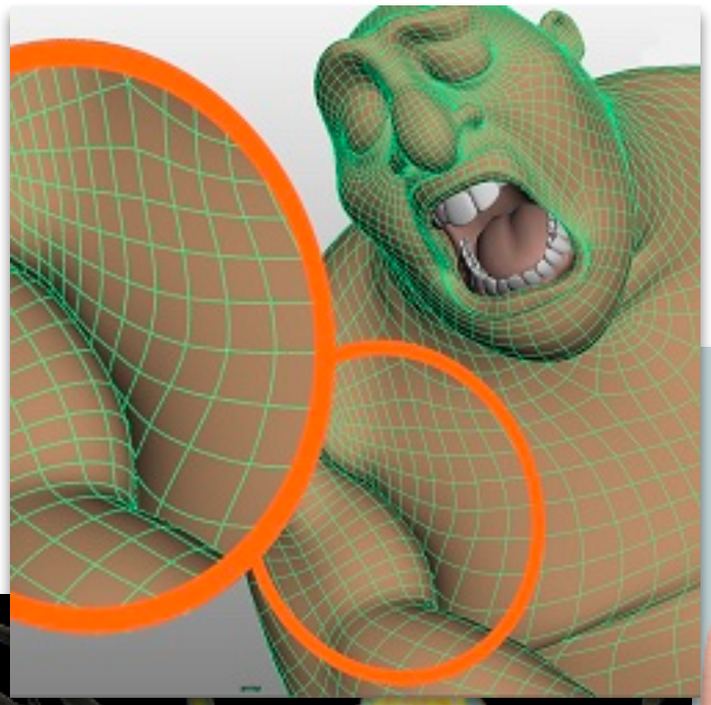
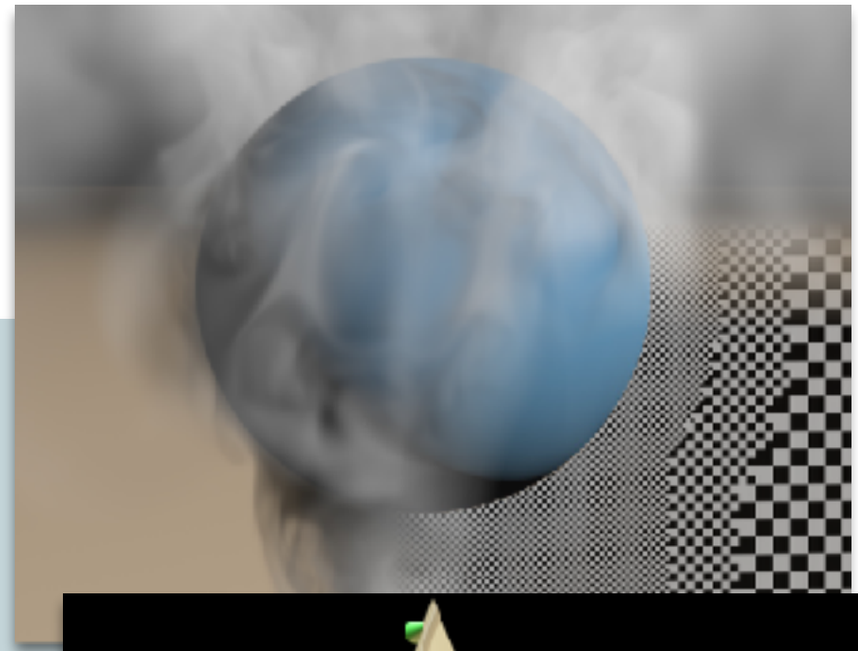
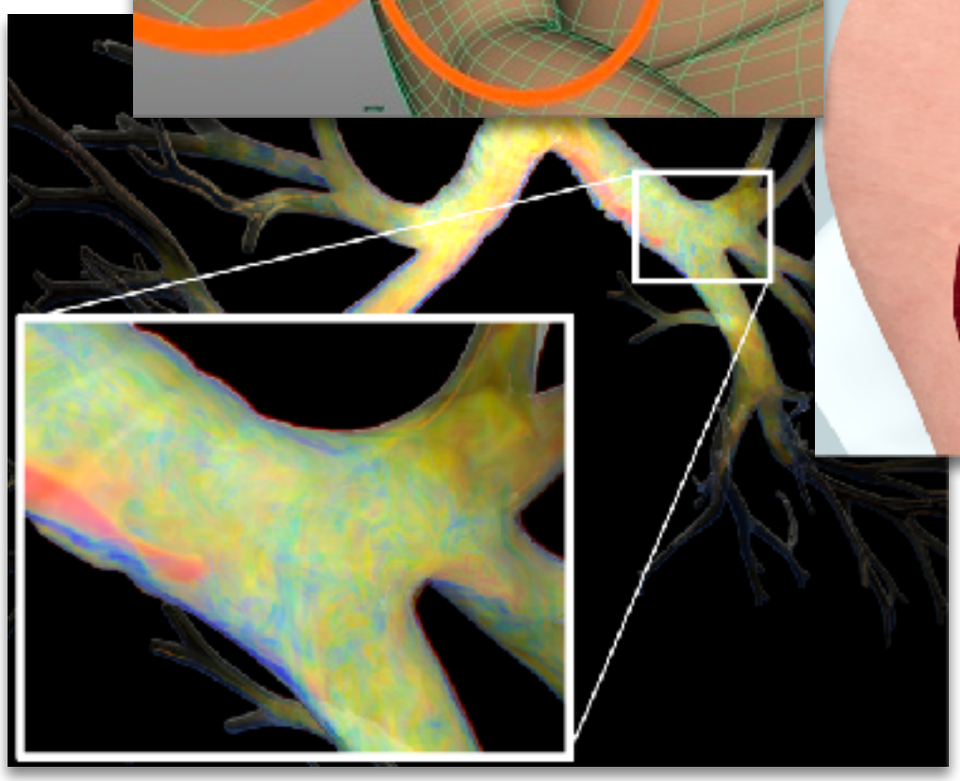
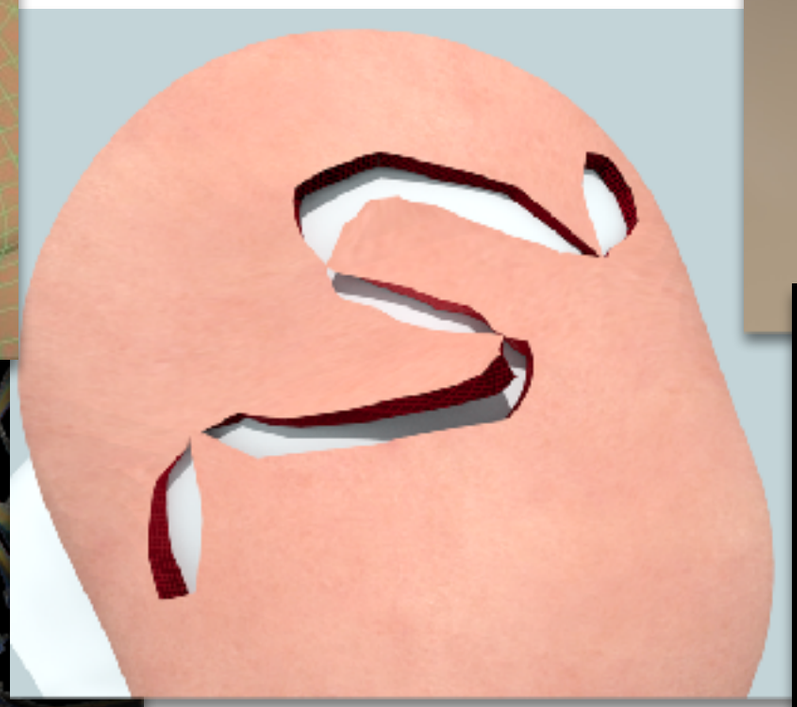
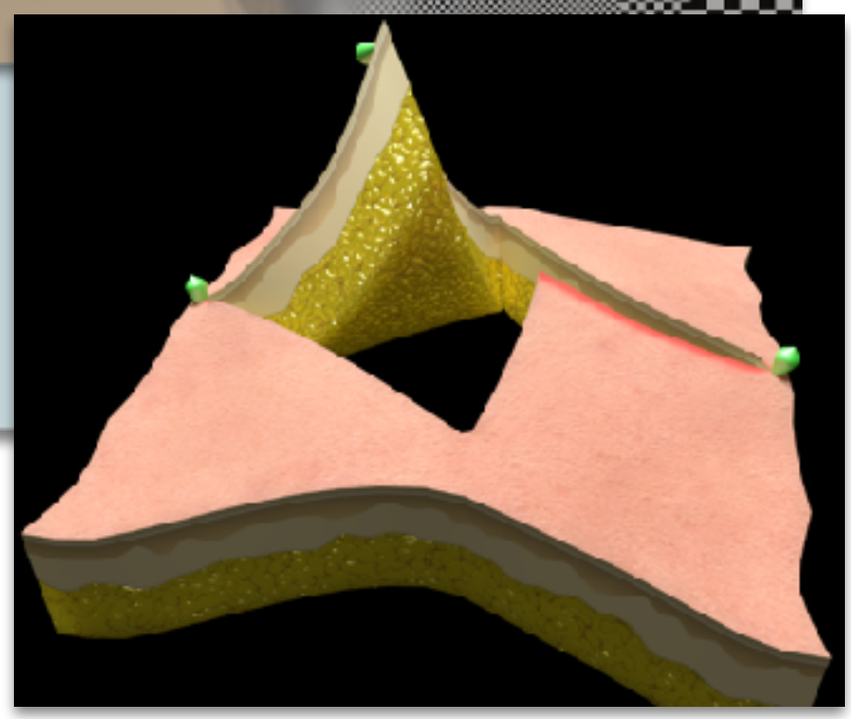










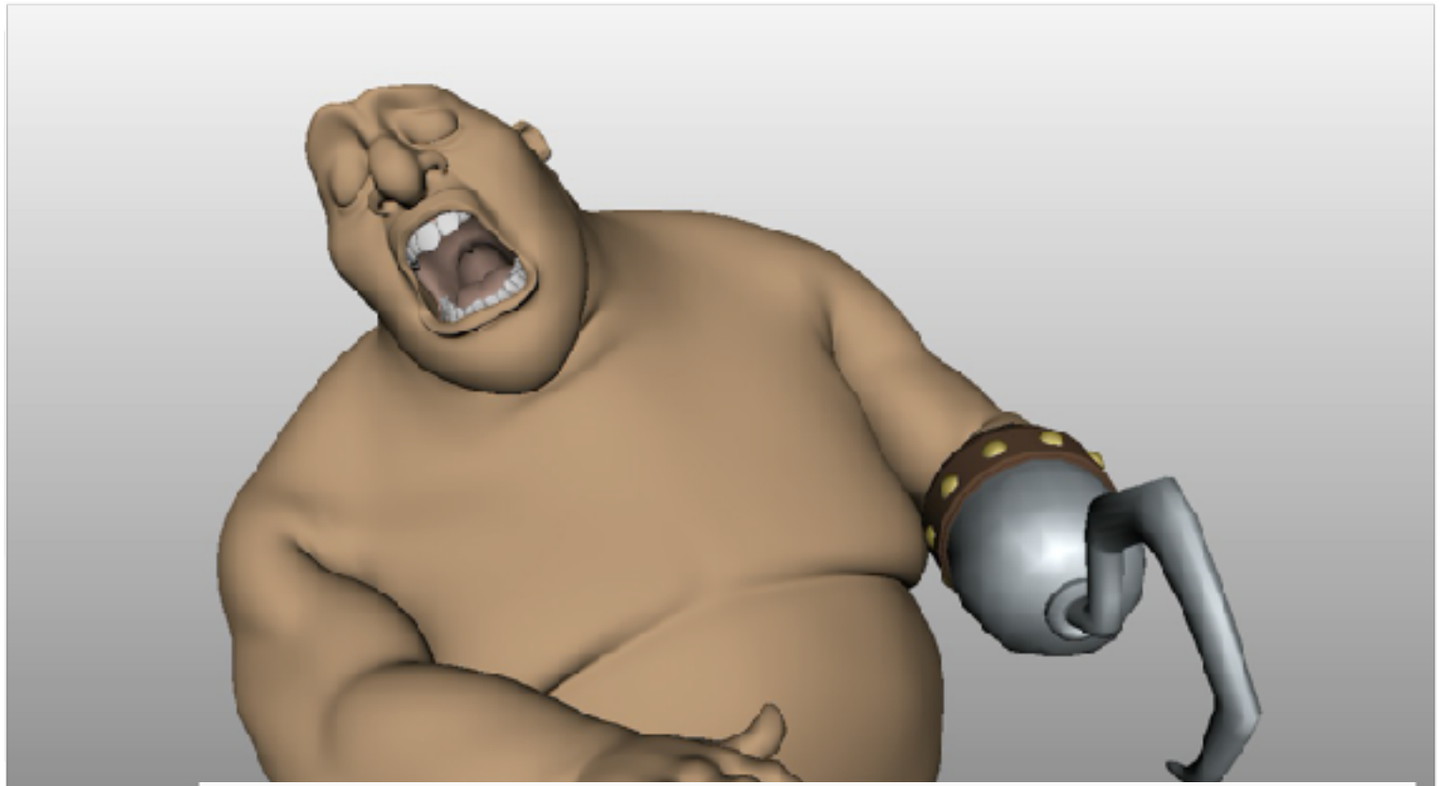
Digital humans, virtual surgery and fast fluids; do they have more in common than their hunger for performance?

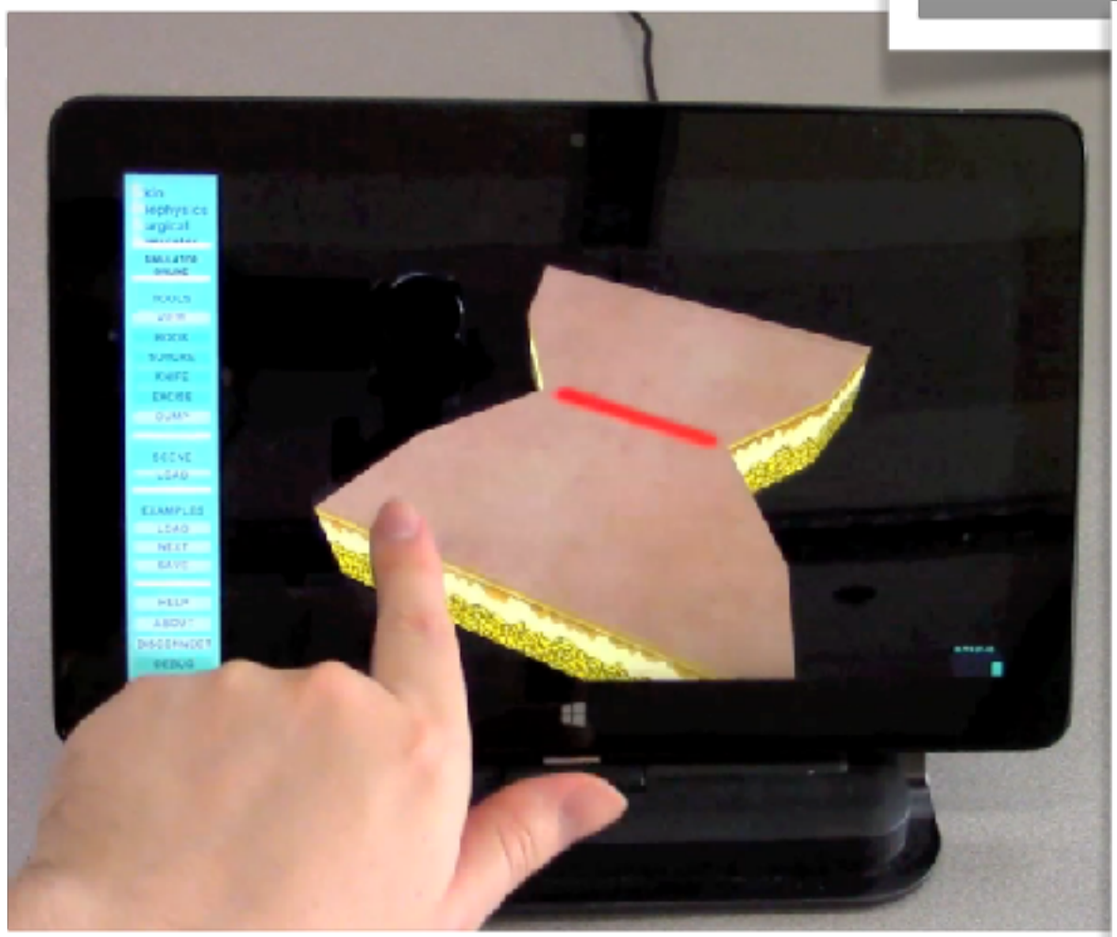
Eftychios Sifakis

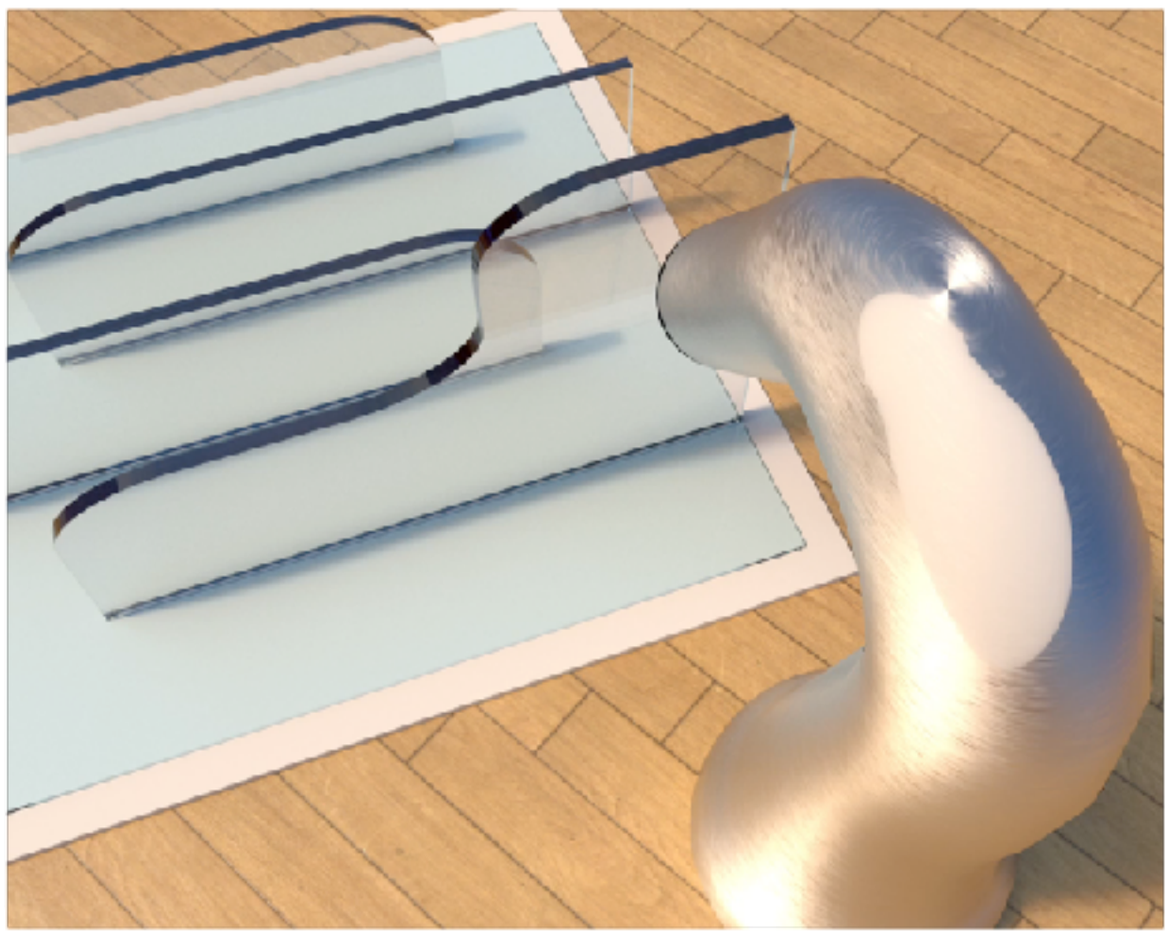
Department of Computer Sciences

University of Wisconsin - Madison

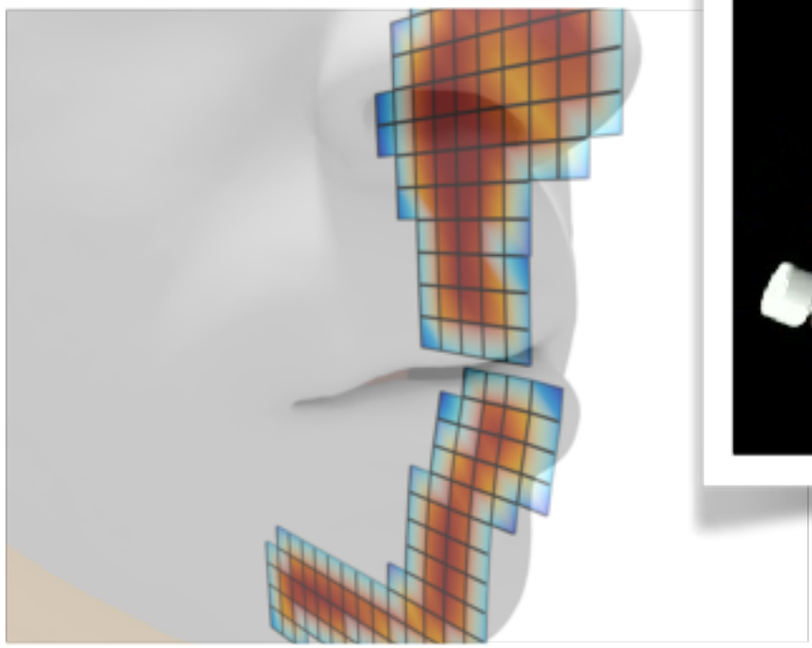
WISCONSIN
UNIVERSITY OF WISCONSIN-MADISON

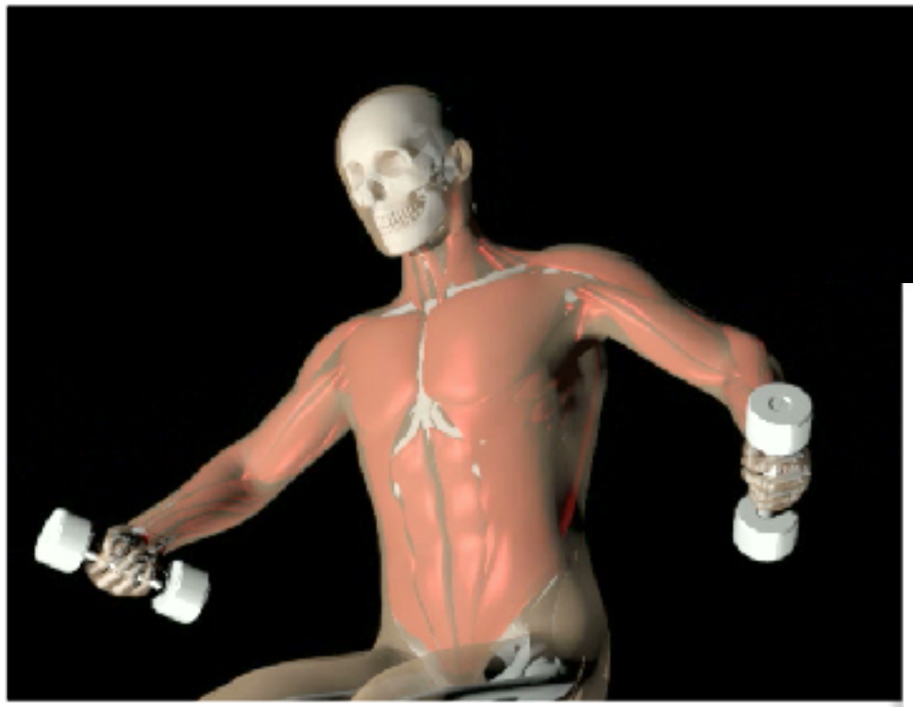


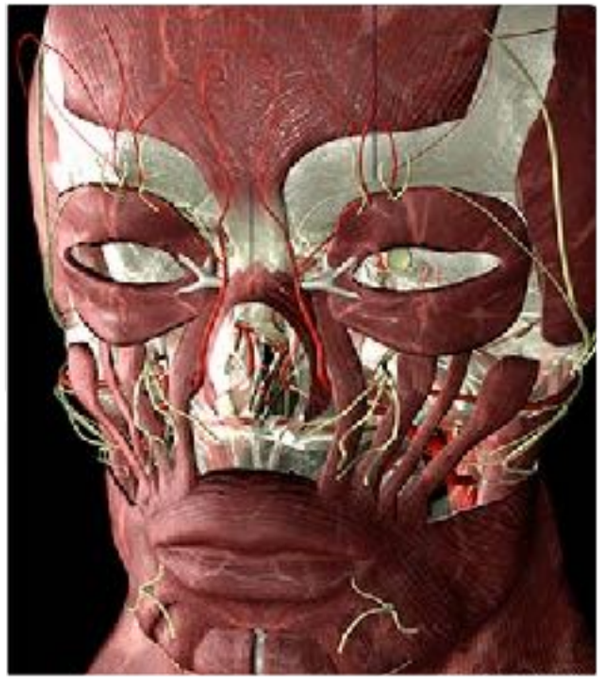




Fast forward to lessons learned …

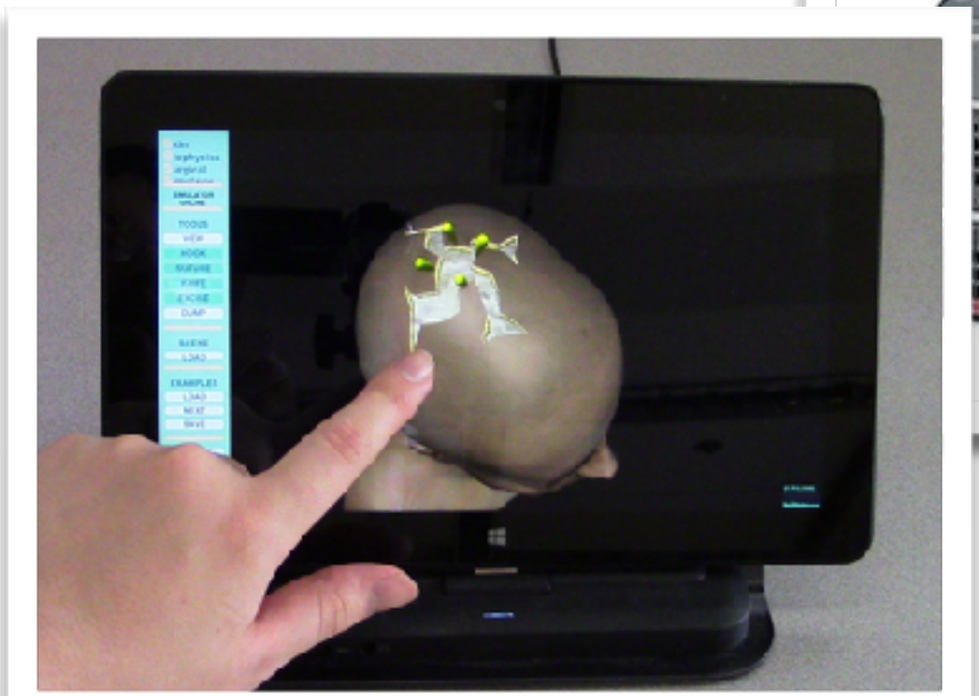






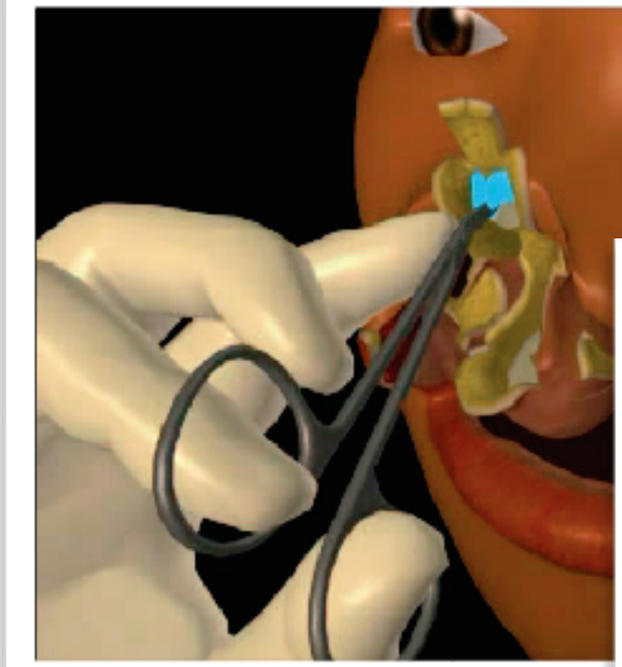
Digital anatomy research yields ***intrinsic benefits****, but is also a* ***catalyst for innovation*** *on synergistic disciplines.*

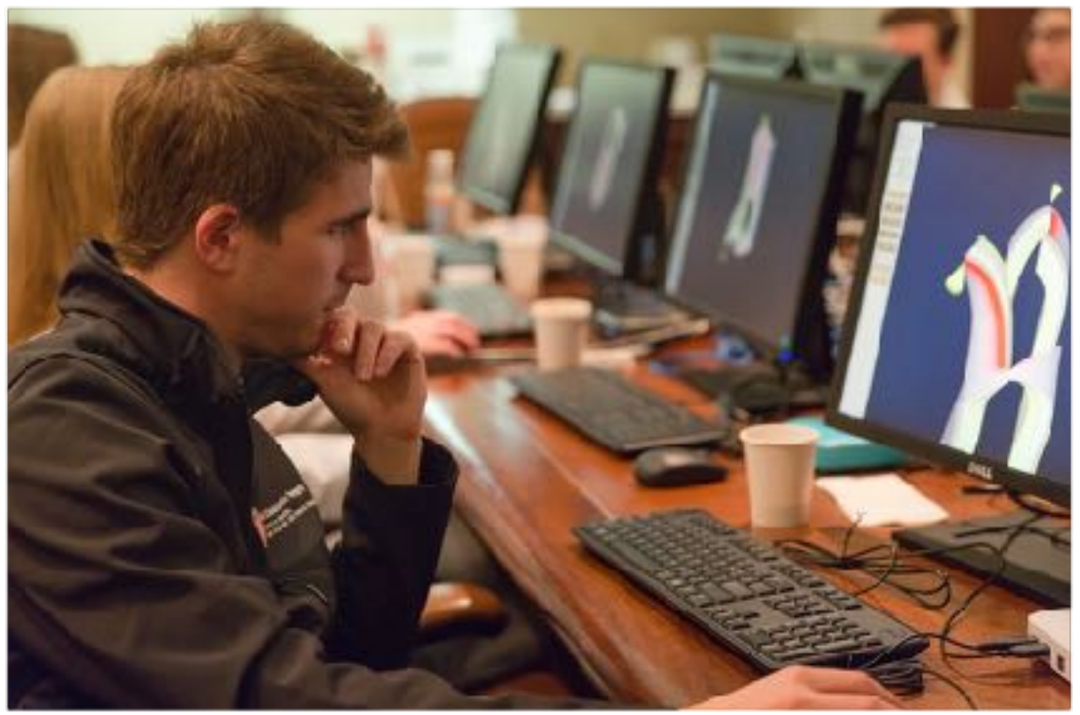
Fast forward to lessons learned …

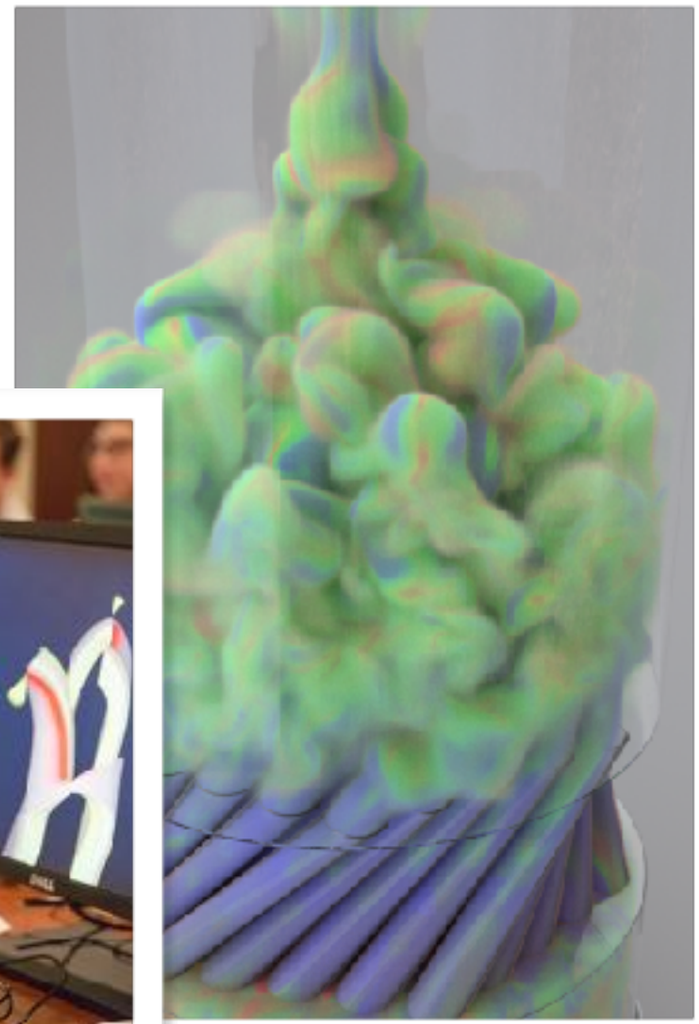


*Openness to **cross-layer interventions** maximizes the opportunity for breakthrough achievements …*

Fast forward to lessons learned …

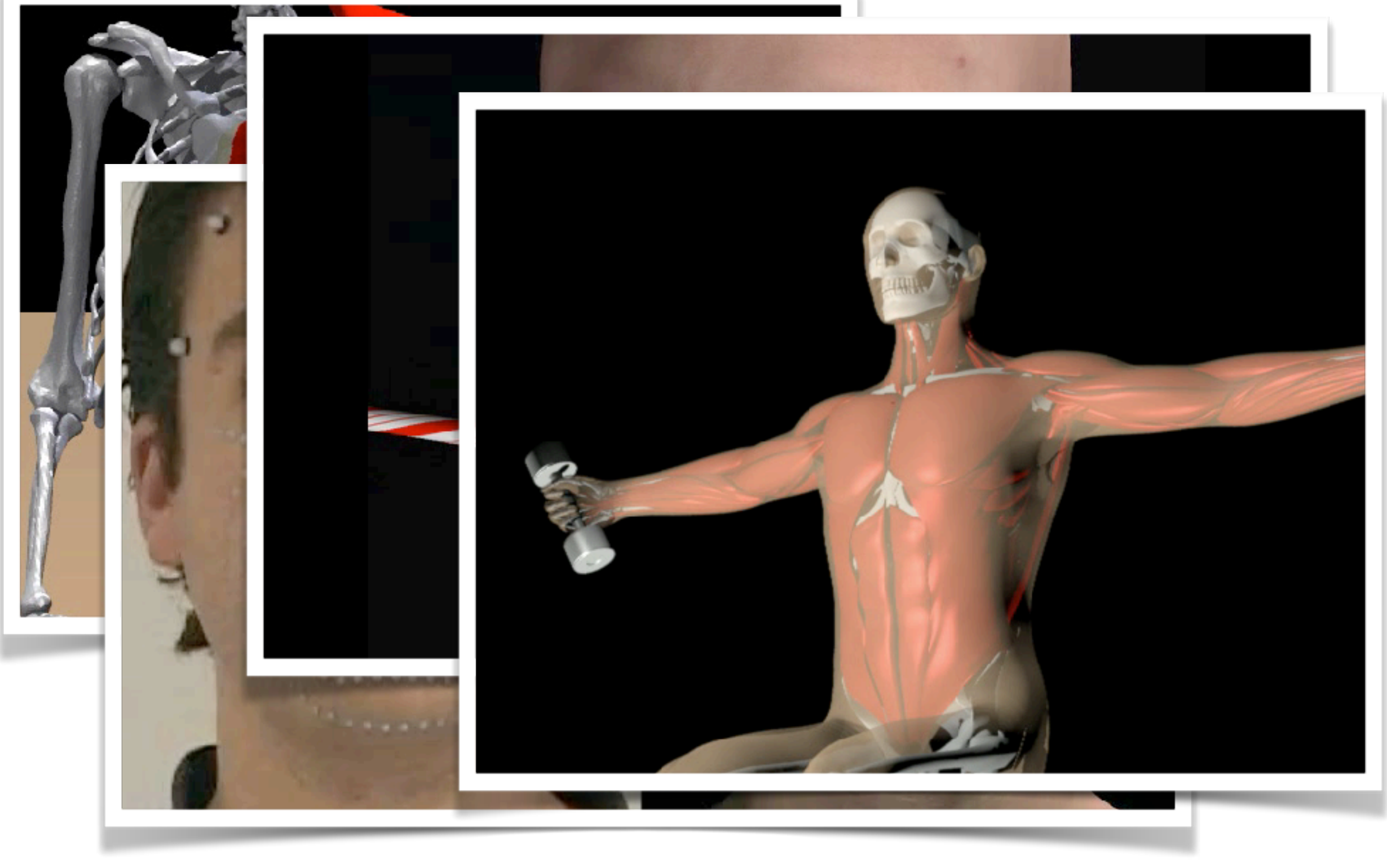




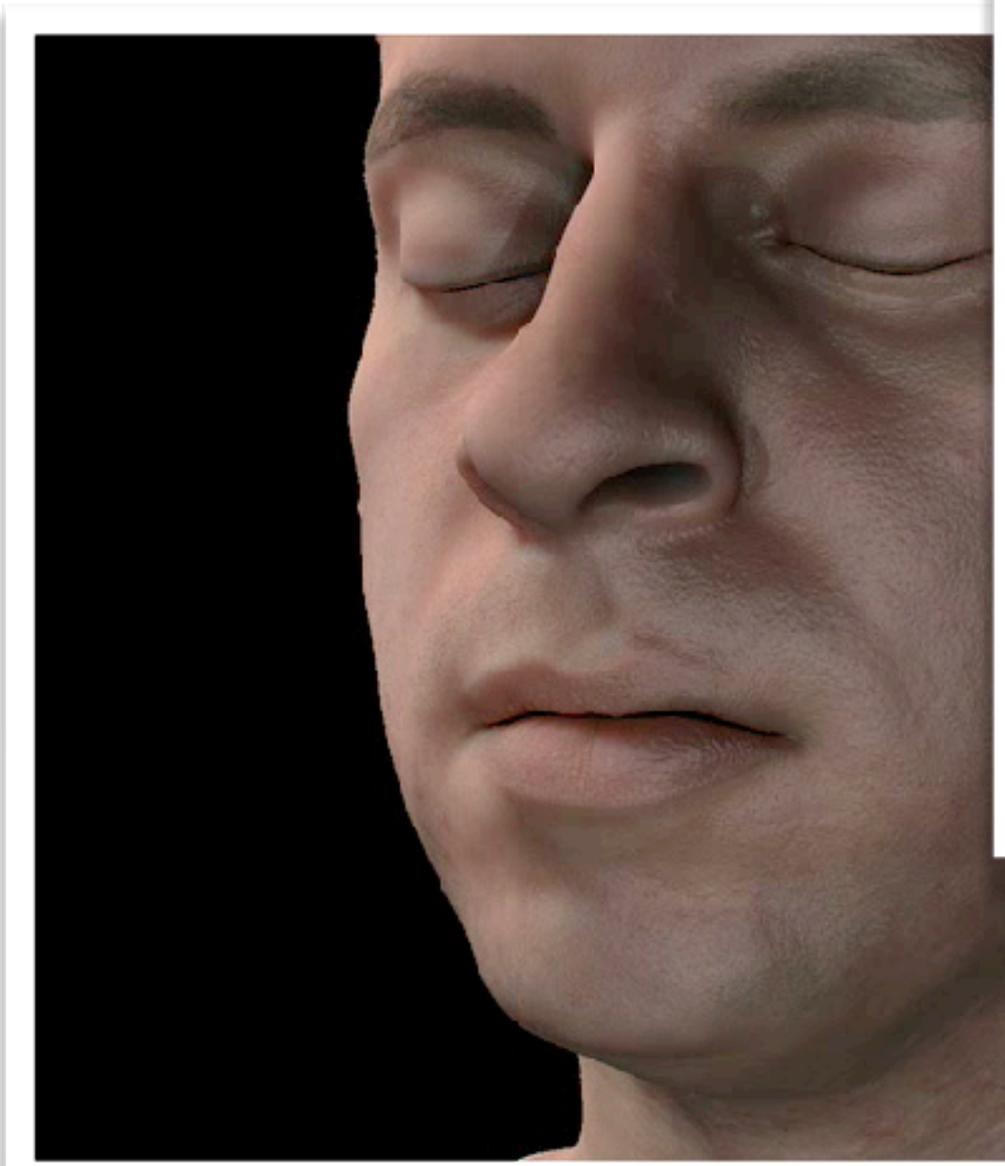


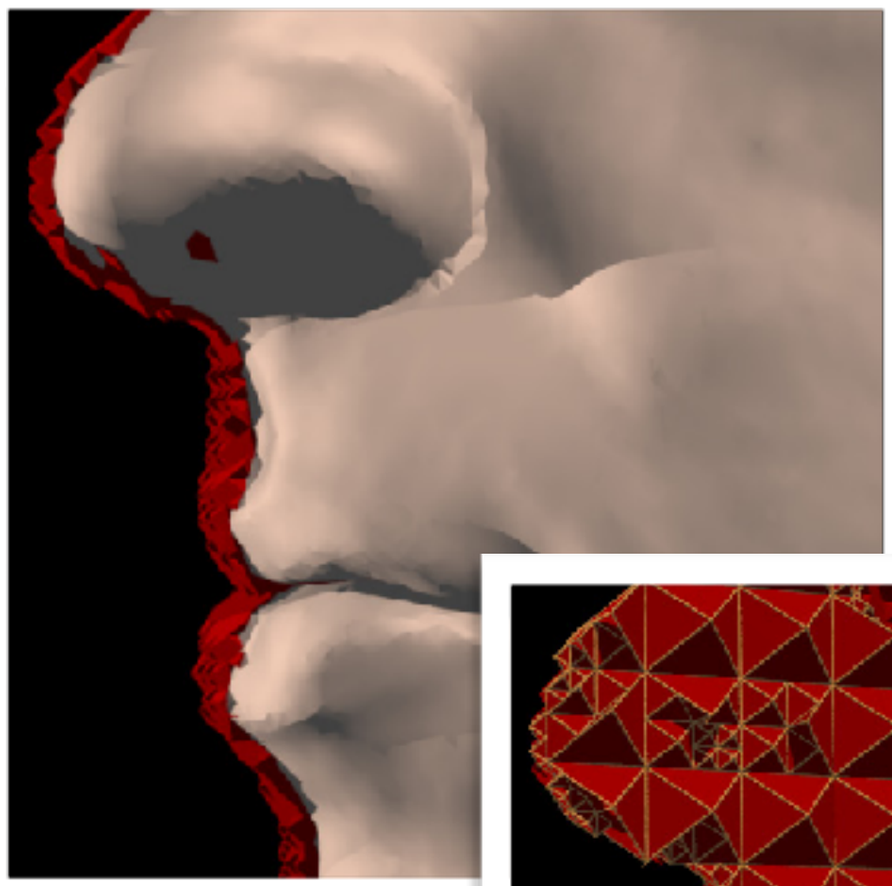
… and the impact of such design philosophy can be have ***benefits beyond modeling*** *(or visual computing).*

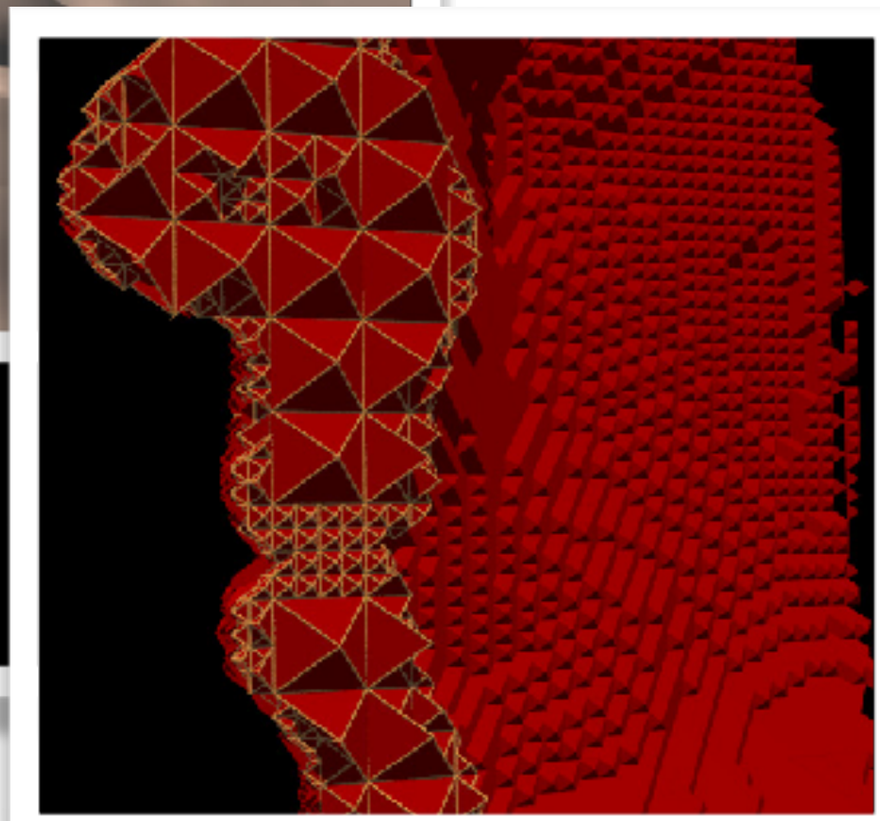
How did this journey start … ?



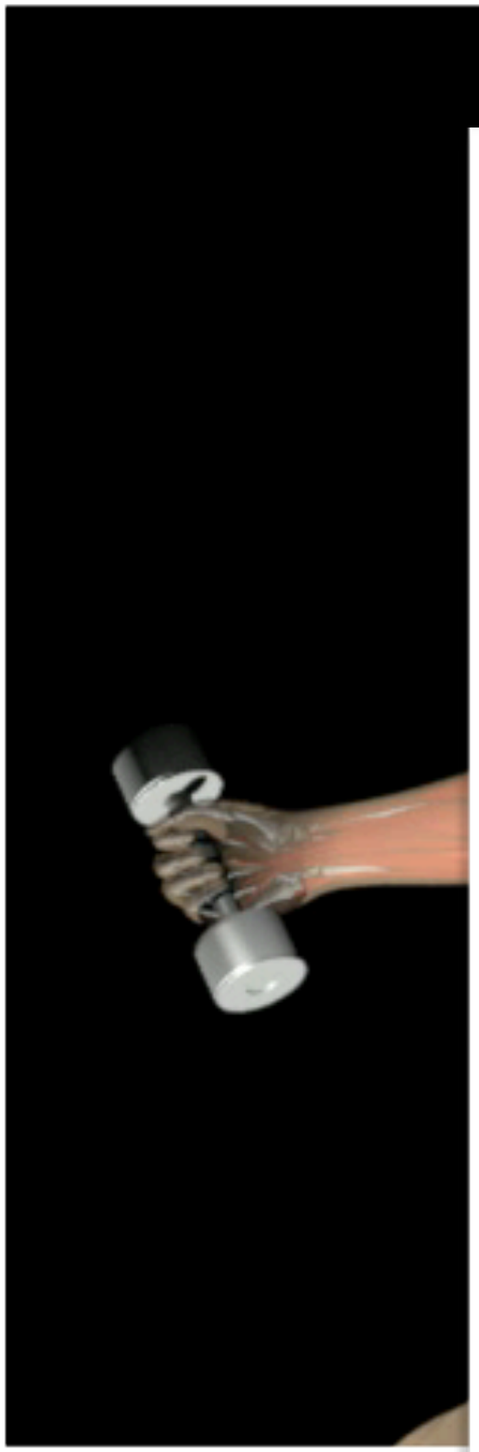
How did this journey start … ?

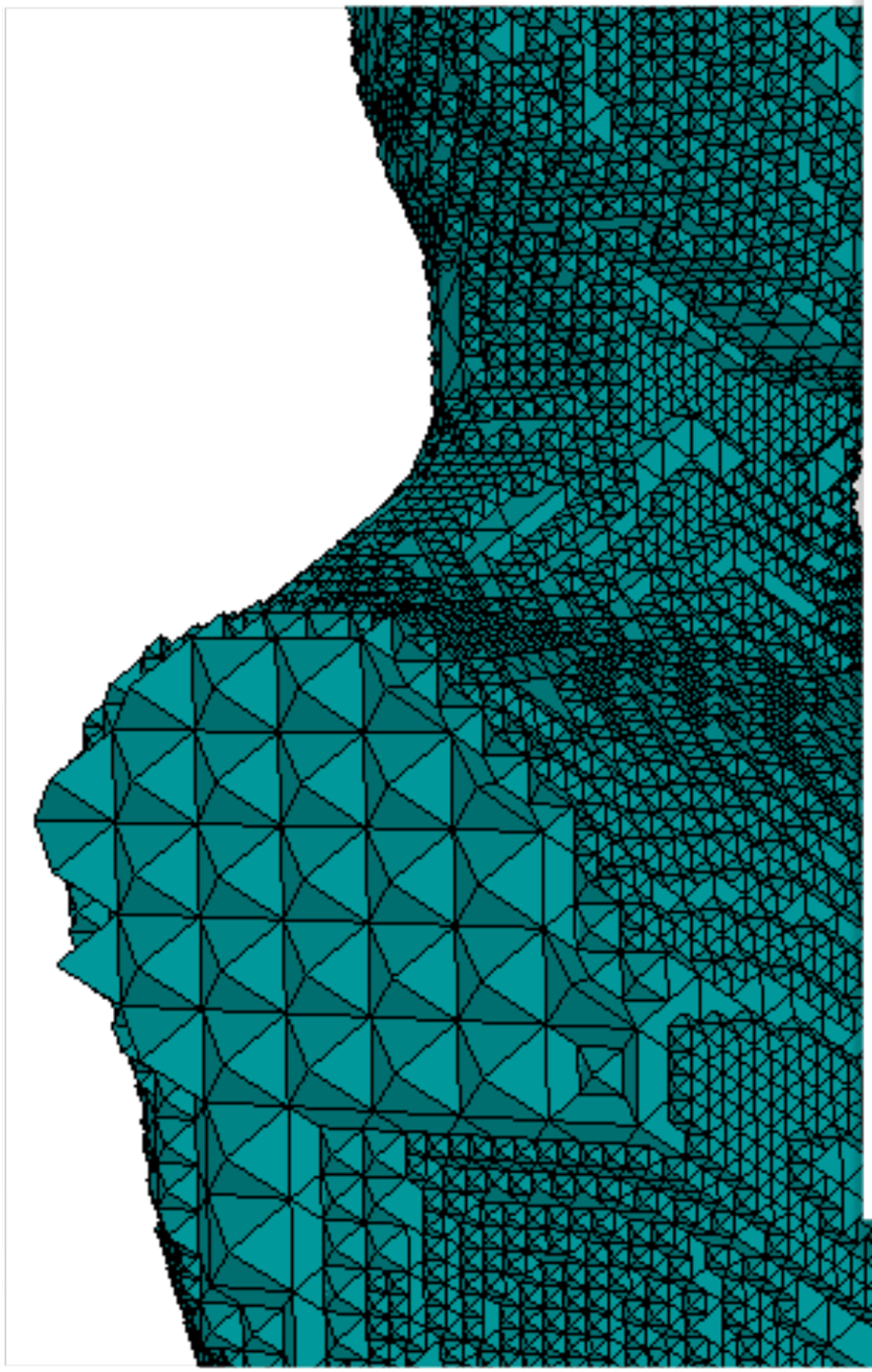






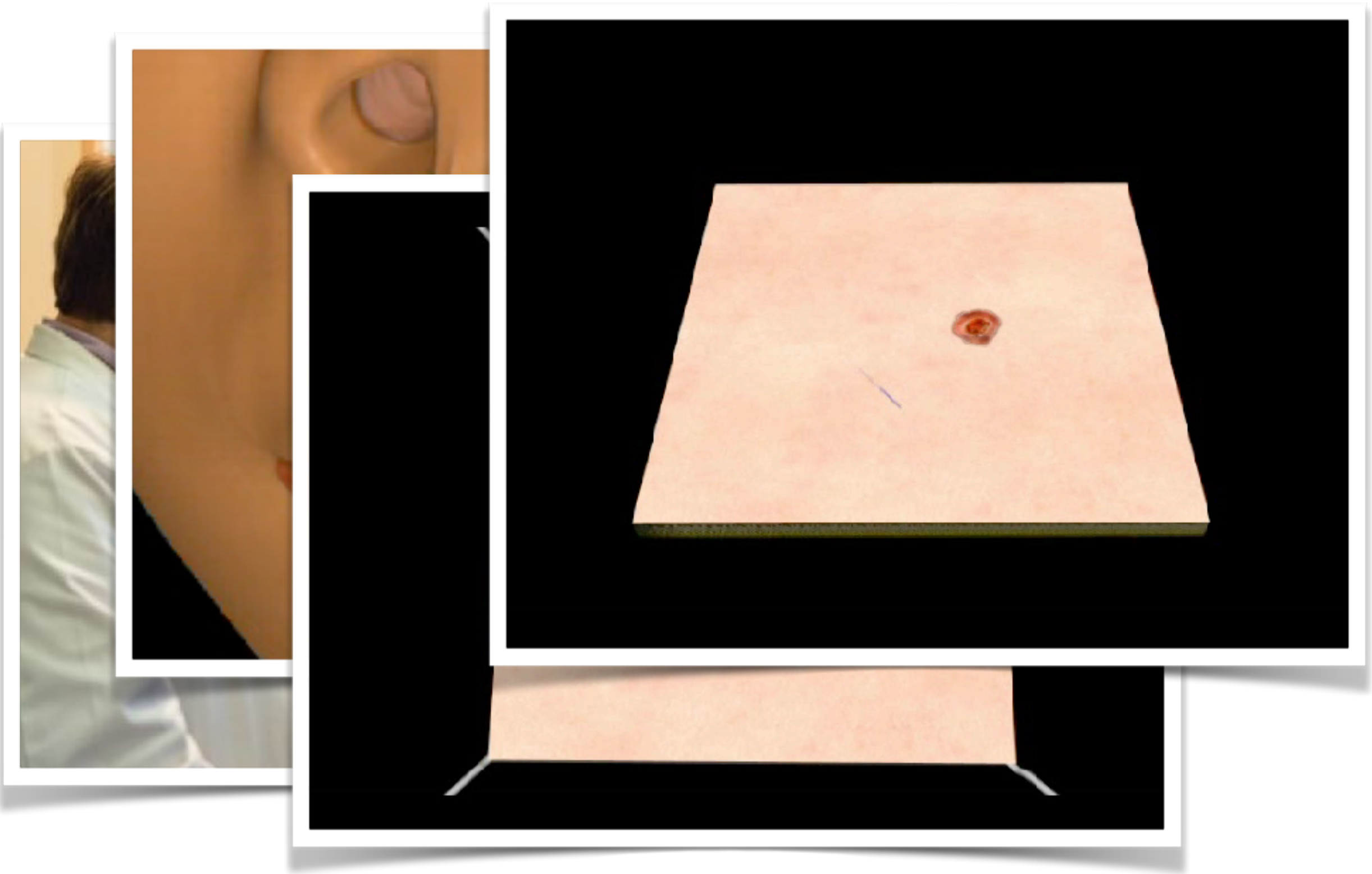
How did this journey start … ?

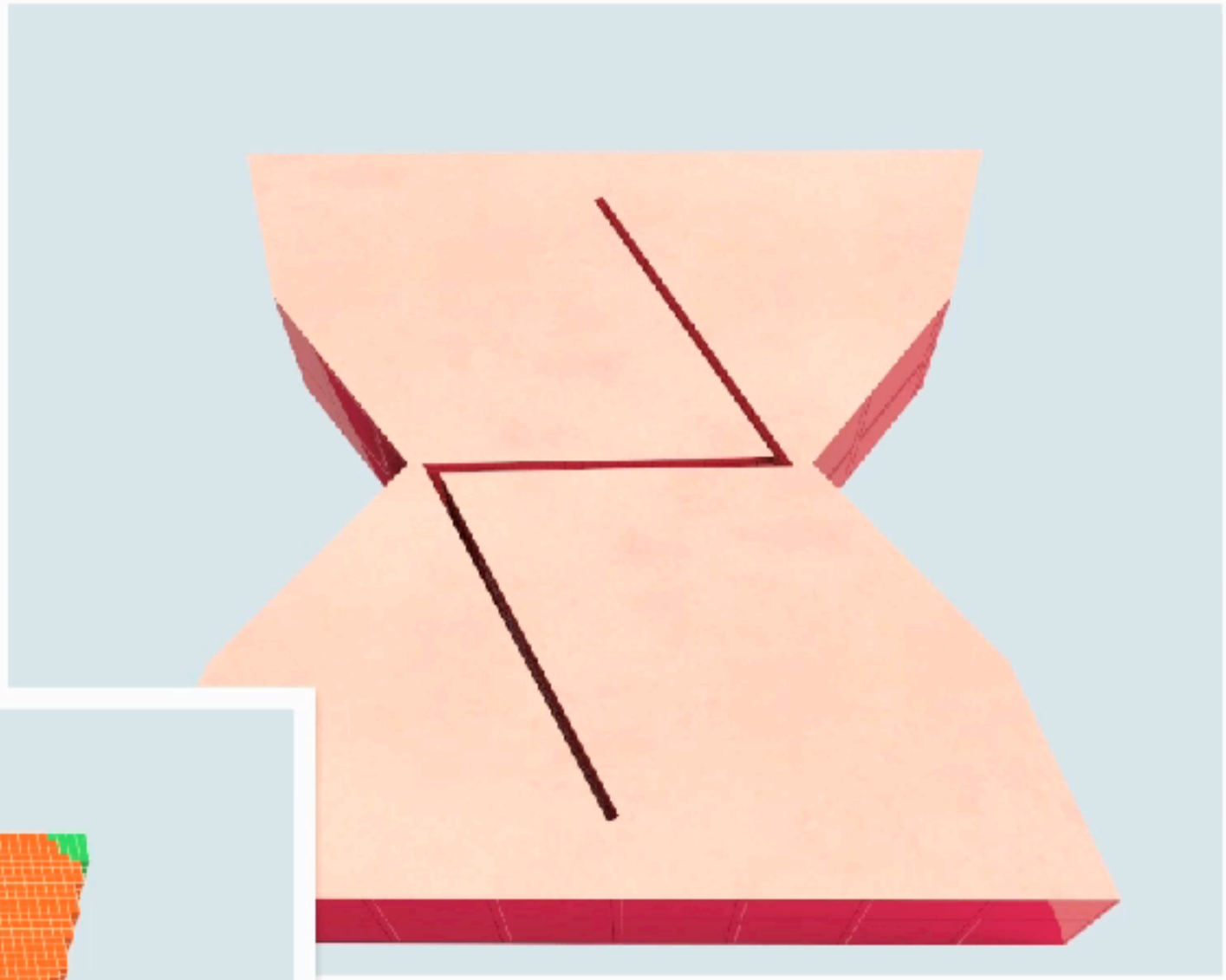


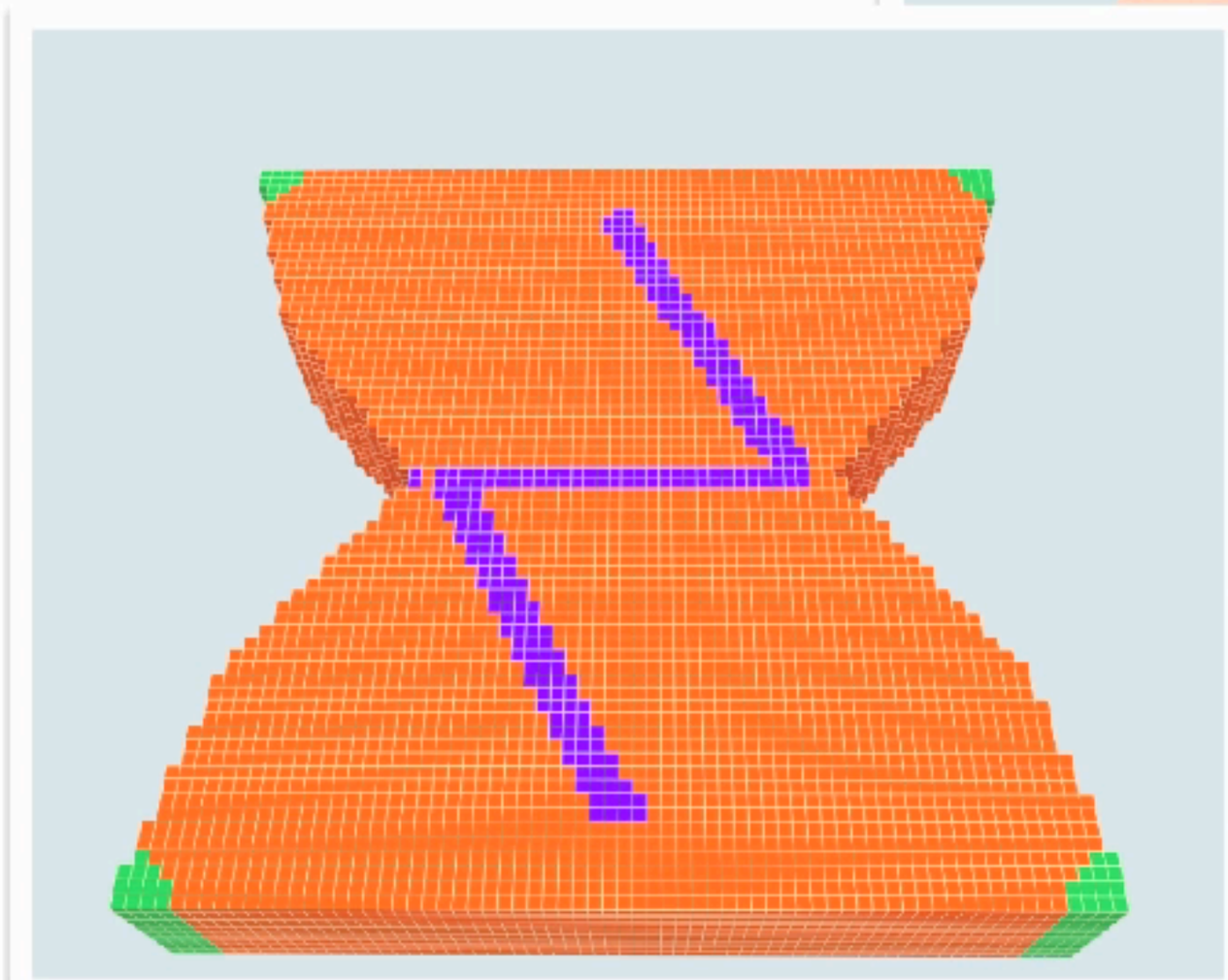


… and how it got into fracture and destruction …

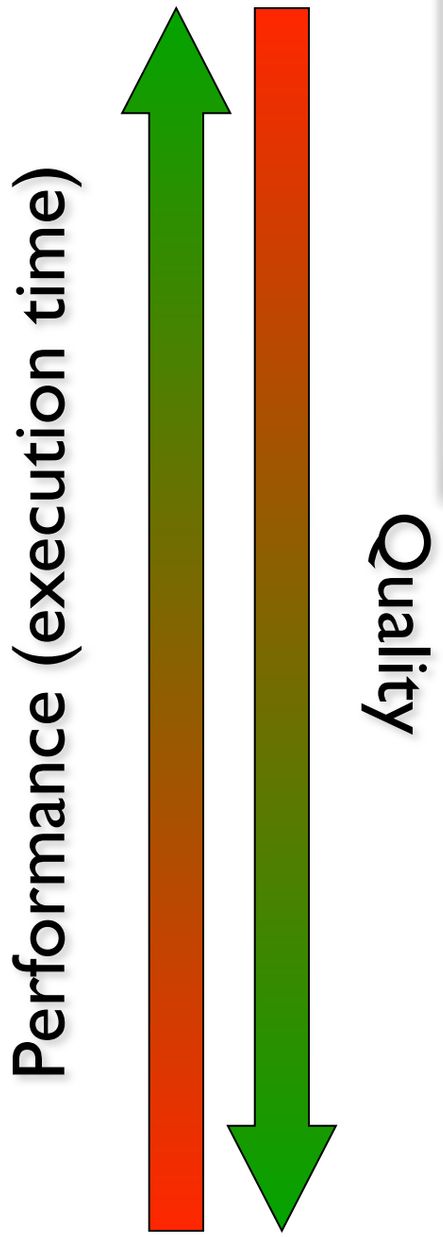
… or cutting and virtual surgery

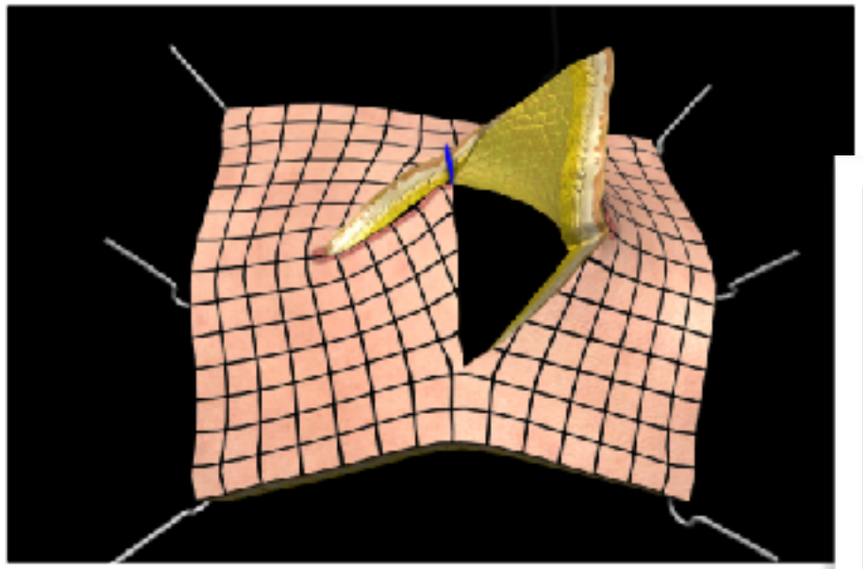


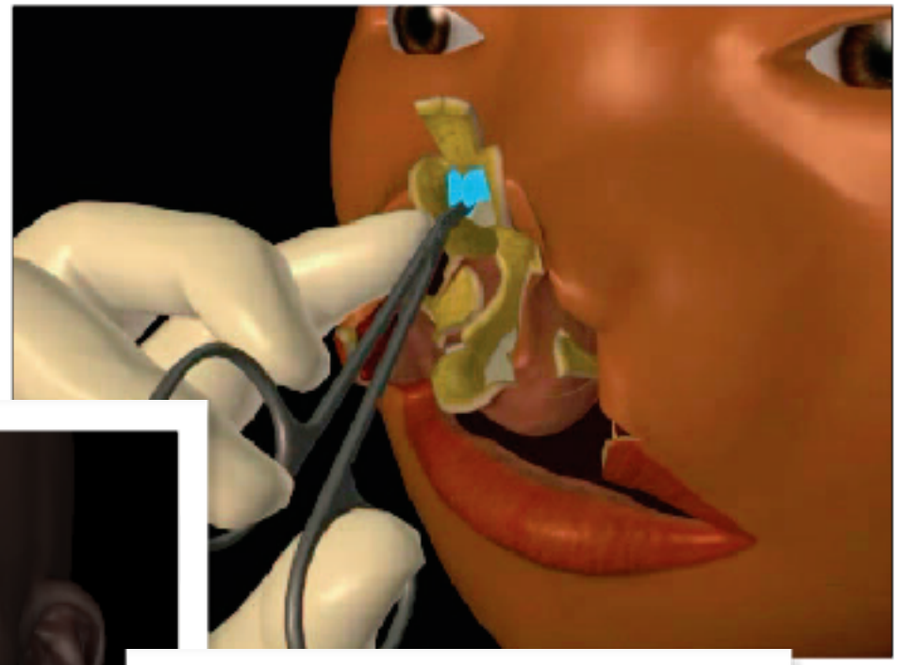


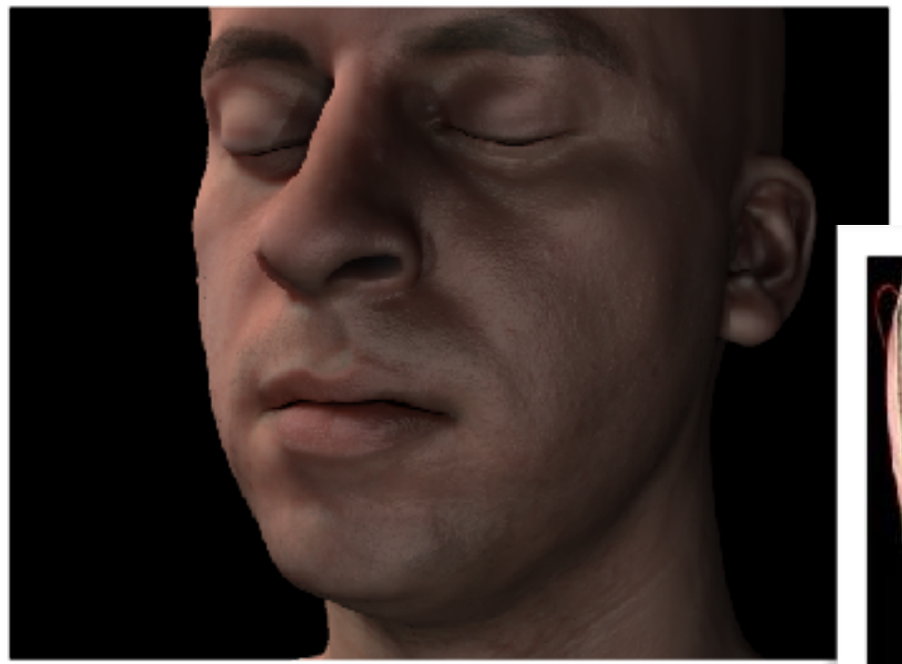


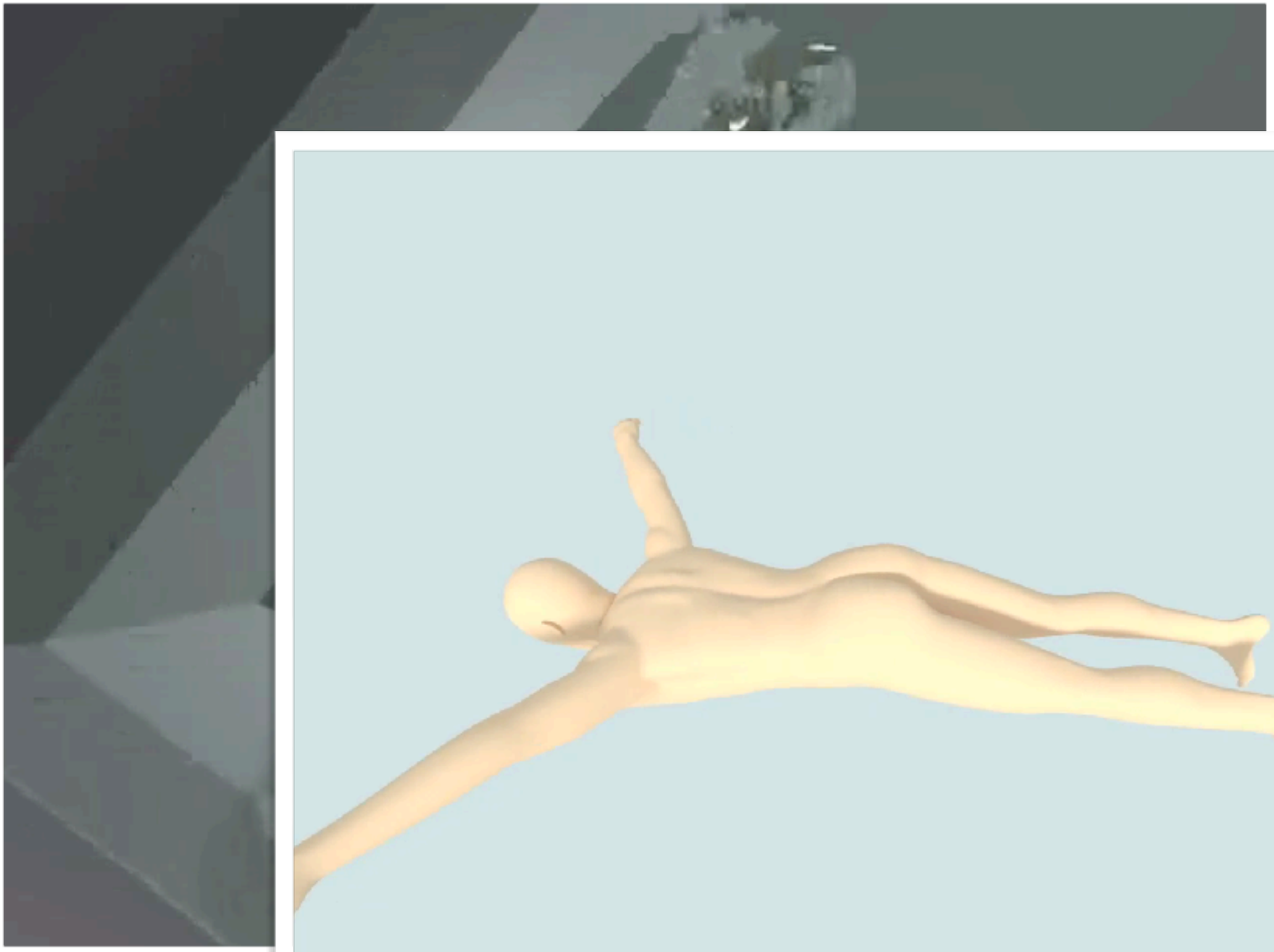
Performance : Incremental benefit or critical feature?











Target performance?

Storage

- Explicit-assembly : 243 coefficients per vertex (give or take …)
- Essential meta-data for matrix-free : 36 floats per vertex

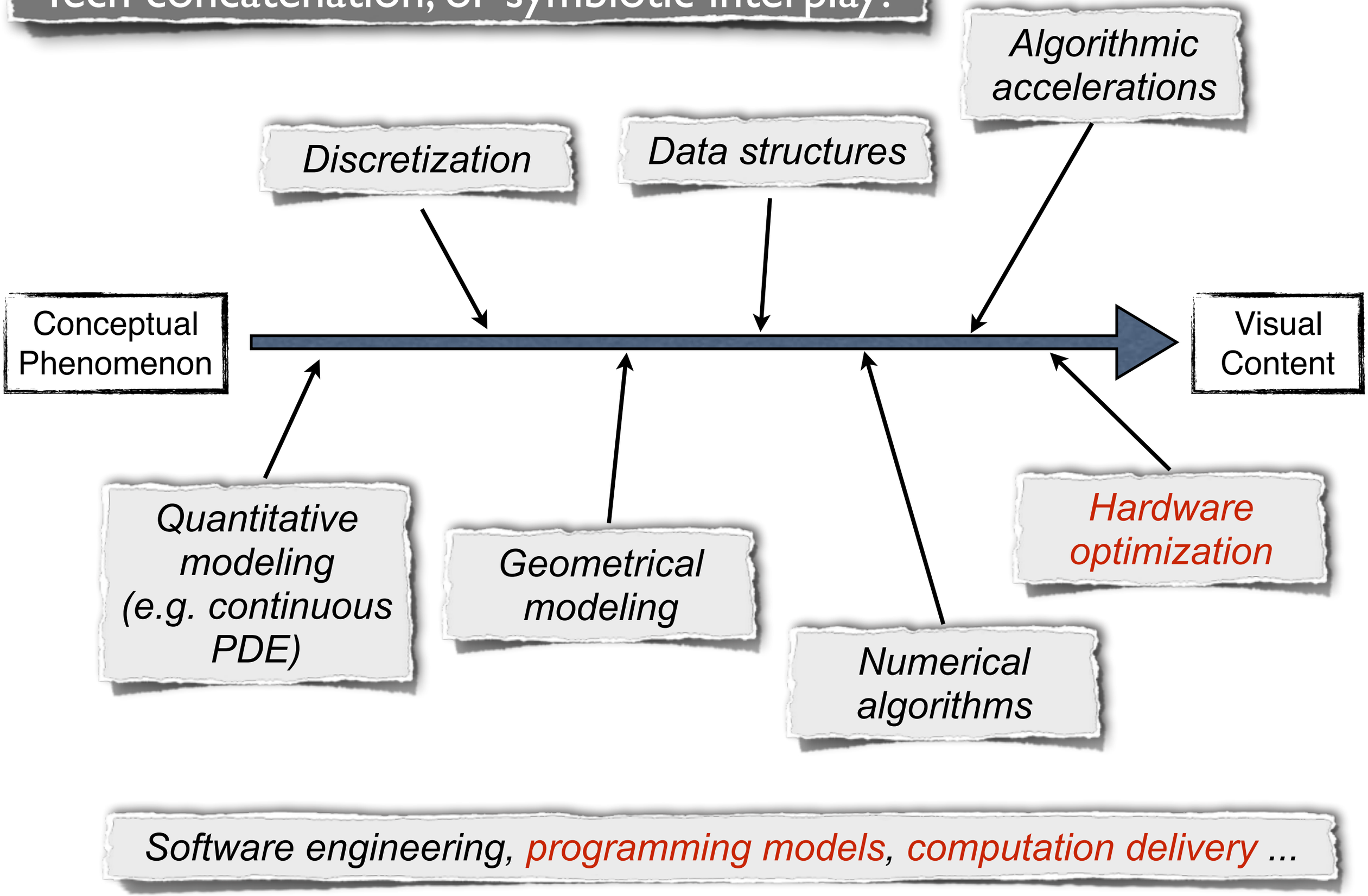
Implicit mat-vec multiply cost:

- About 1.8-2.5x of data access (read/write) cost
- Via SIMD + MultiThread optimizations.

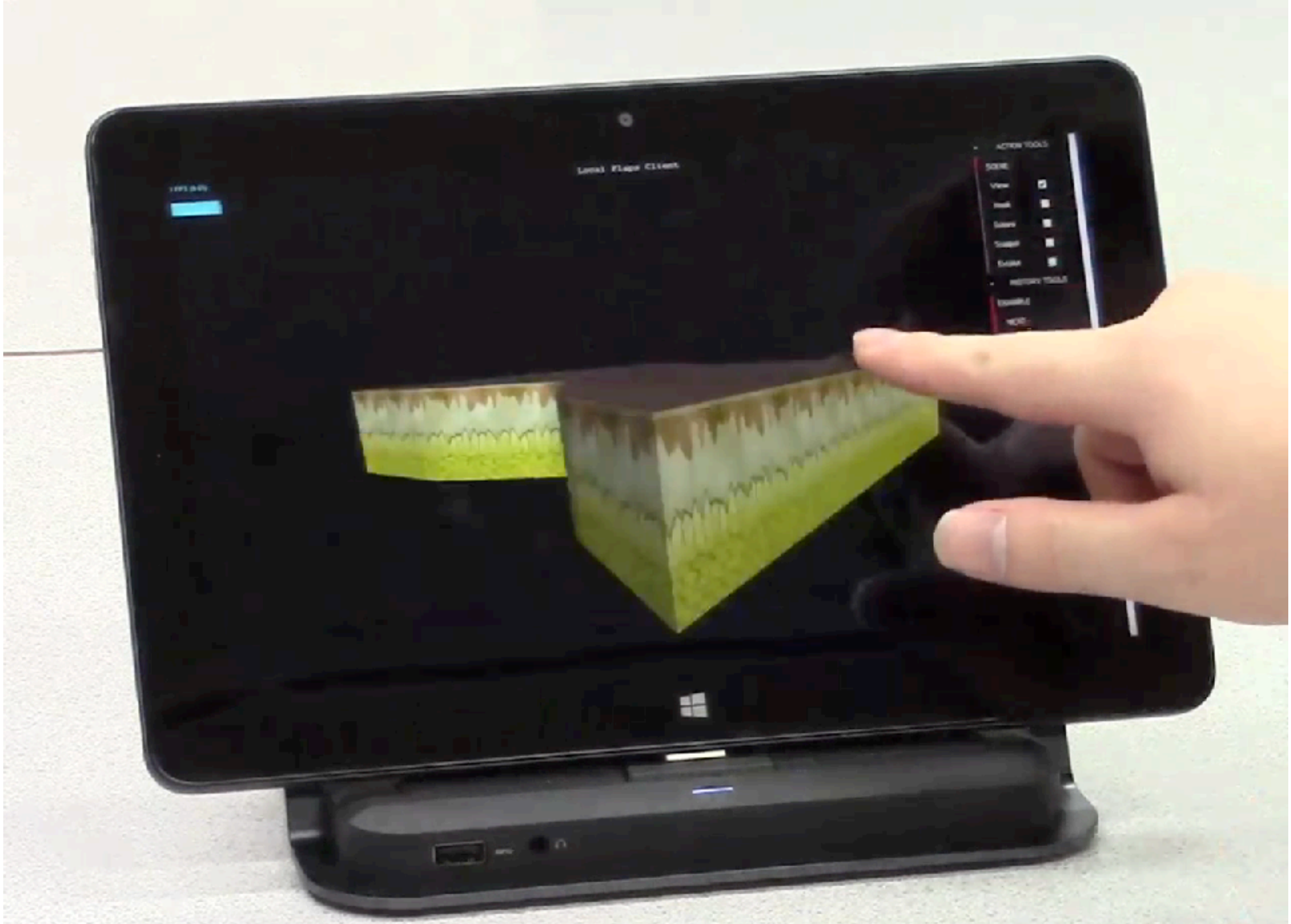
A developer's nightmare?

- Sometimes … see SVD for Disney's PhysGrid.
- But there's a way to find a method to the madness.

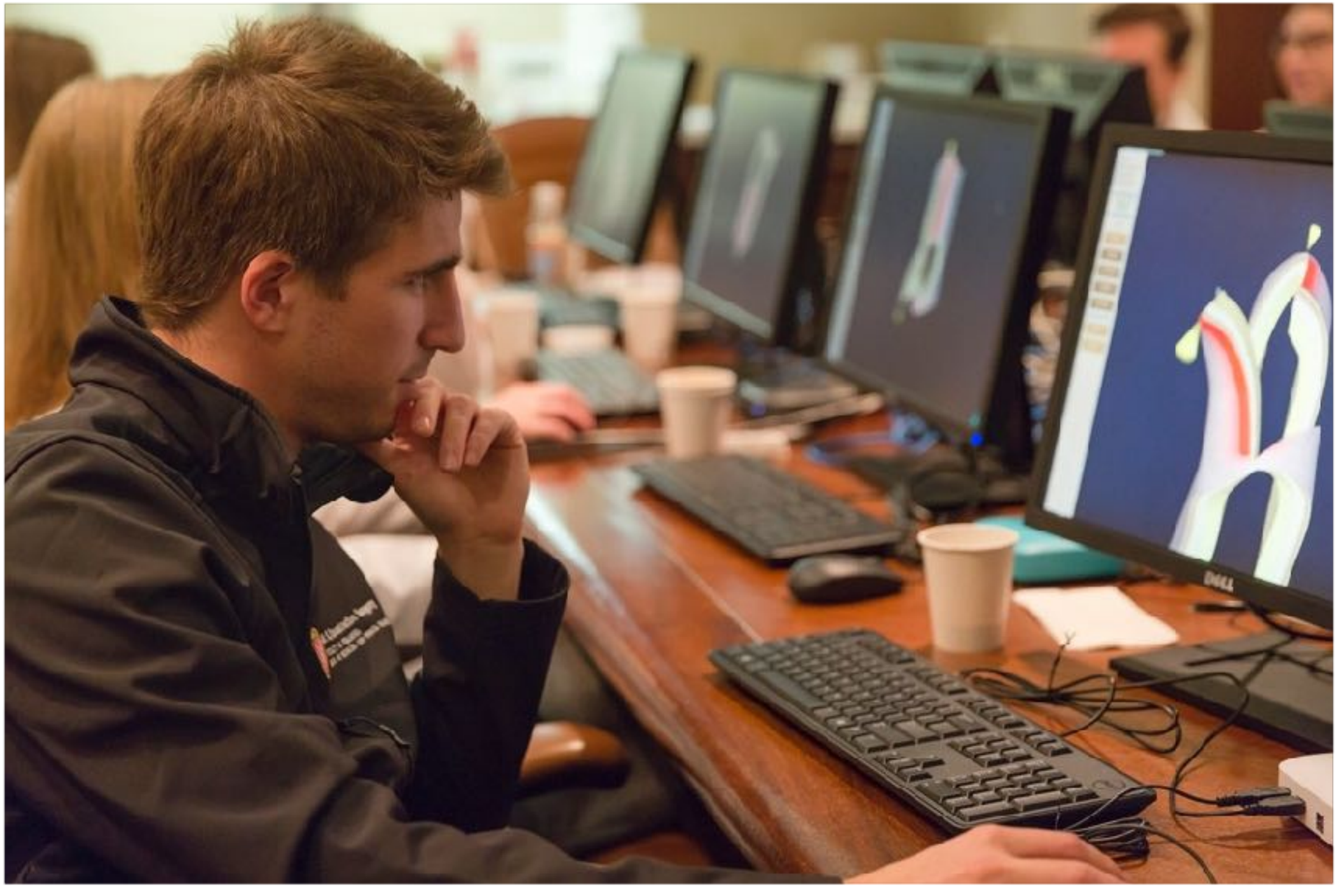
Tech concatenation, or symbiotic interplay?



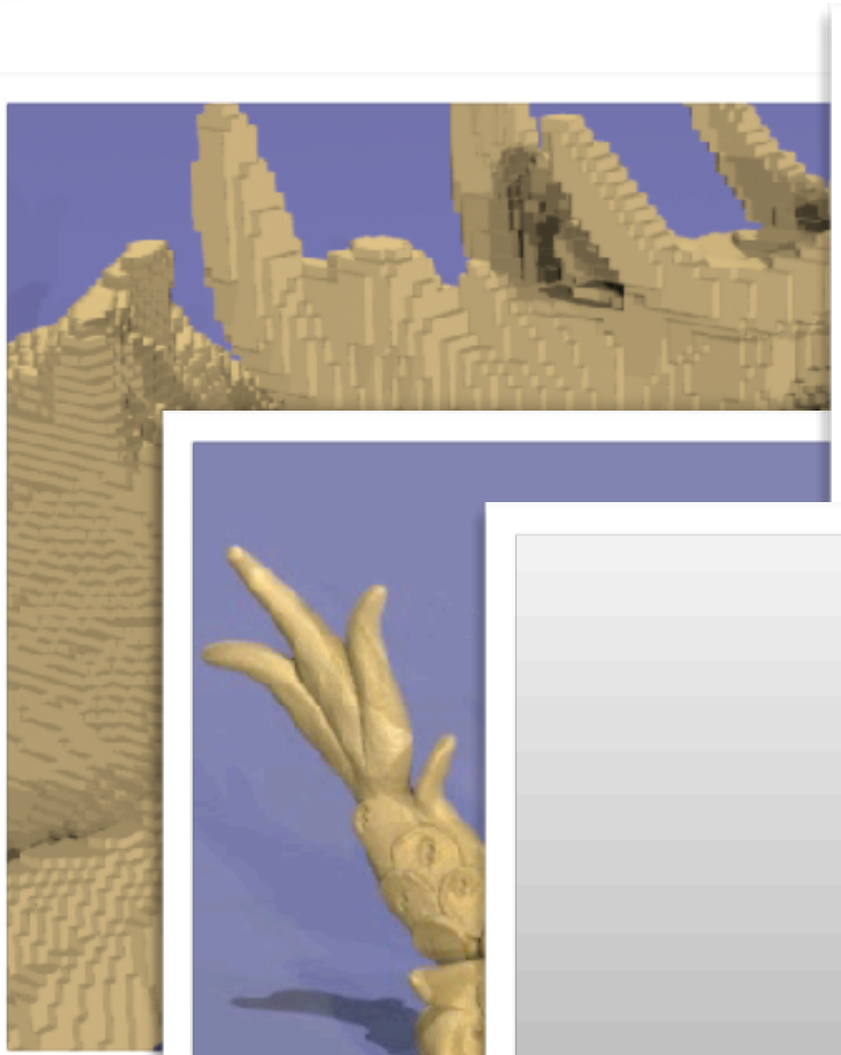
Software engineering, programming models, computation delivery ...

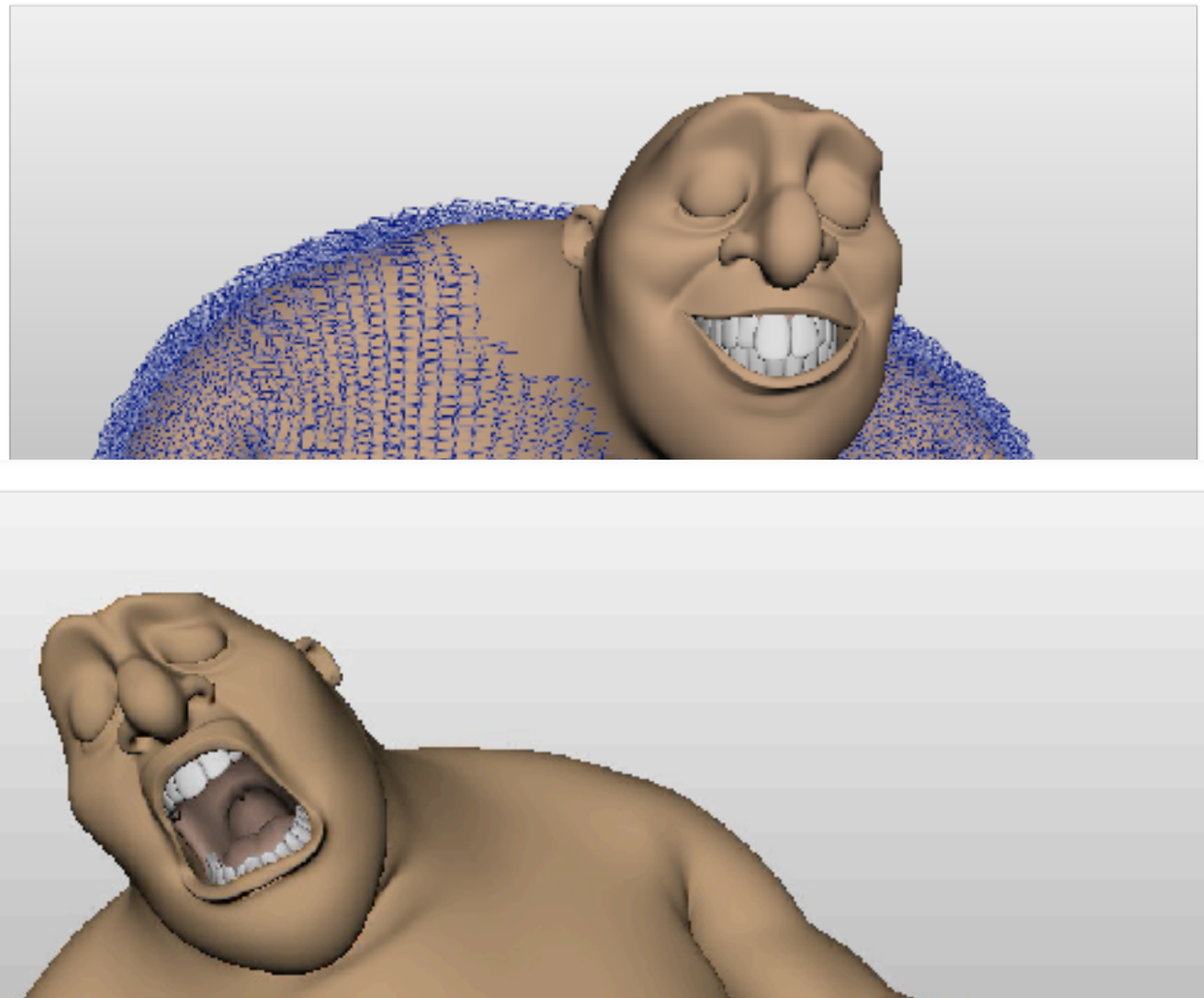

Local Flags Client
ACTION TOOLS
SCENE
View
Mesh
Camera
Transport
Engine
HISTORY TOOLS
DISABLE
MODE

Developments: Soft tissue surgery sim



Bringing hand-optimized code under control?





Target performance?

Storage

- Explicit-assembly : 243 coefficients per vertex (give or take …)
- Essential meta-data for matrix-free : 36 floats per vertex

Implicit mat-vec multiply cost:

- About 1.8-2.5x of data access (read/write) cost
- Via SIMD + MultiThread optimizations.

A developer's nightmare?

- Sometimes … see SVD for Disney's PhysGrid.
- But there's a way to find a method to the madness.

Guided Vectorization

Semantics : Execution of the 3x3 matrix operation $C=A*B^T$
*Arguments : **Streams** of 3x3 matrices $A^{(k)}$, $B^{(k)}$, $C^{(k)}$*
Operation :
for(k=0;k<W;k++) $C^{(k)} := A^{(k)} * [B^{(k)}]^T$

```
template<class T_RAW,class T_DATA,int multiplicity>
void Matrix_Times_Transpose(const T_DATA (&A)[9], const T_DATA (&B)[9], T_DATA (&C)[9]);
```

Guided Vectorization

```
template<class T_RAW,class T_DATA,int multiplicity>
void Matrix_Times_Transpose(const T_DATA (&A)[9], const T_DATA (&B)[9], T_DATA (&C)[9]);
```

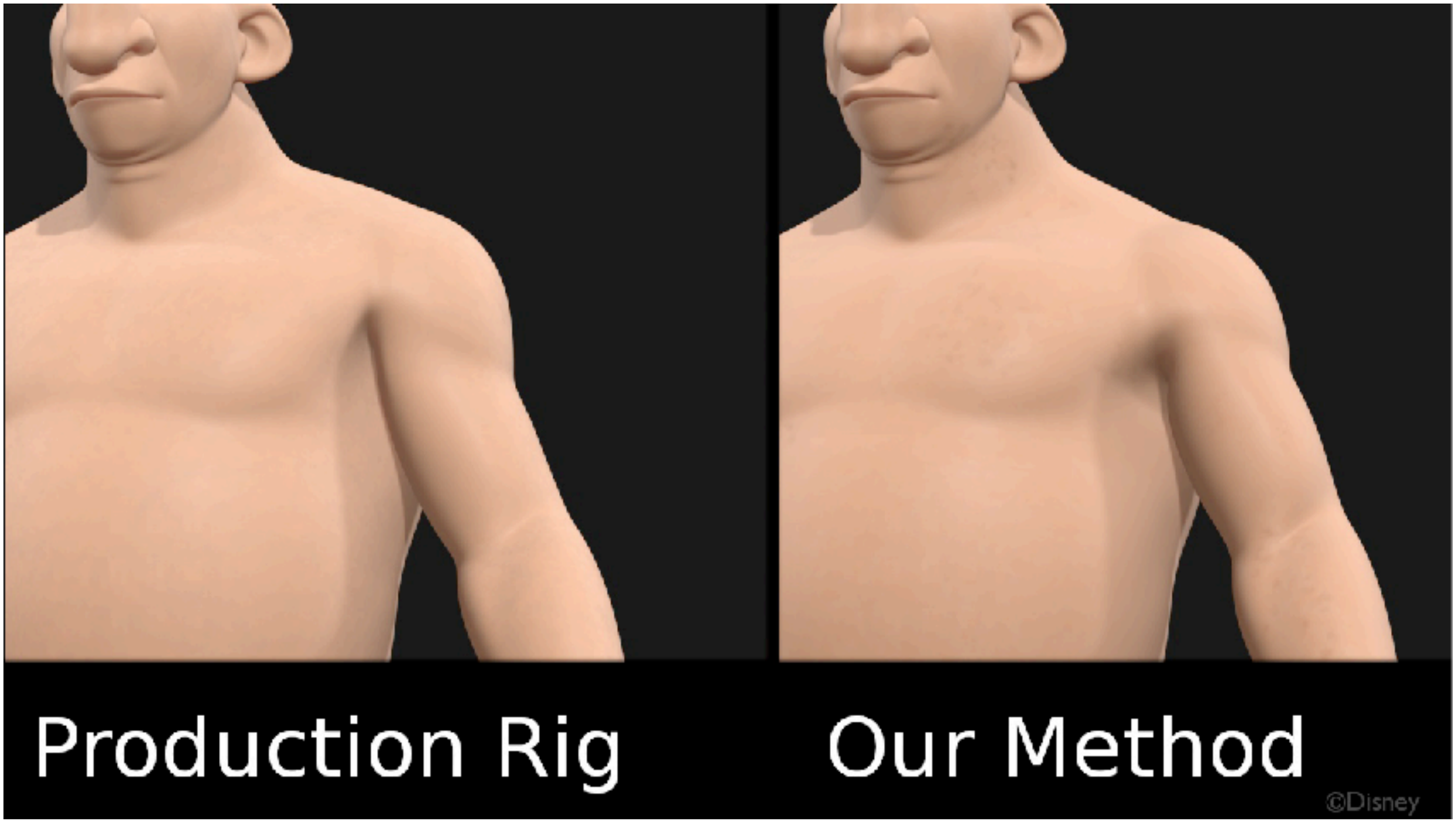
```
template void Matrix_Times_Transpose<float,float>(const float (&A)[3][3],
    const float (&B)[3][3], float (&C)[3][3]);

template void Matrix_Times_Transpose<float,float[8]>(const float (&A)[3][3][8],
    const float (&B)[3][3][8], float (&C)[3][3][8]);

template void Matrix_Times_Transpose<__m128,float[8]>(const float (&A)[3][3][8],
    const float (&B)[3][3][8], float (&C)[3][3][8]);

template void Matrix_Times_Transpose<__m256,float[8]>(const float (&A)[3][3][8],
    const float (&B)[3][3][8], float (&C)[3][3][8]);

template void Matrix_Times_Transpose<__m512,float[16]>(const float (&A)[3][3][16],
    const float (&B)[3][3][16], float (&C)[3][3][16]);

template void Matrix_Times_Transpose<__m512,float[64],4>(const float (&A)[3][3][64],
    const float (&B)[3][3][64], float (&C)[3][3][64]);
```

Guided Vectorization

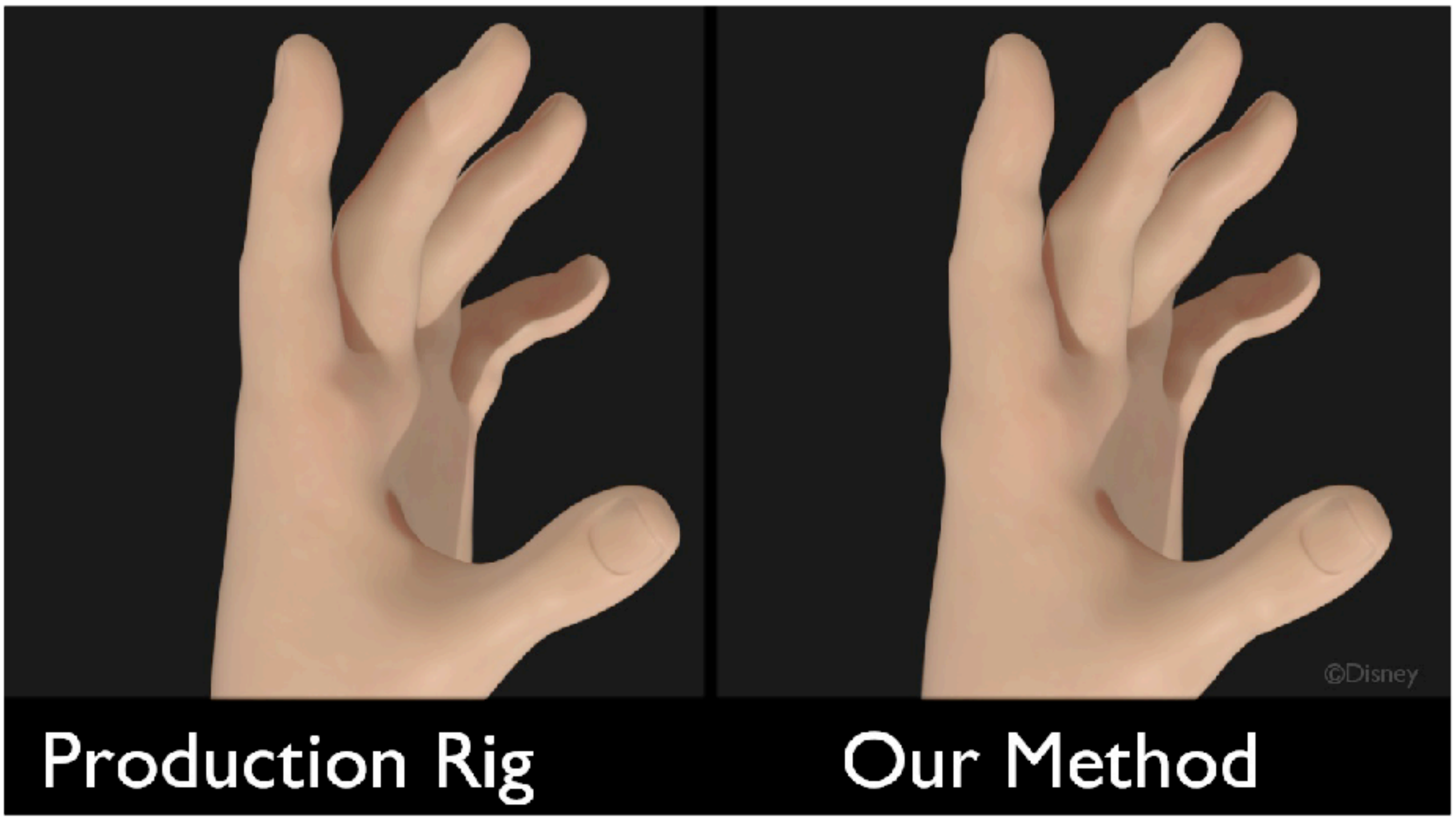
Findings:

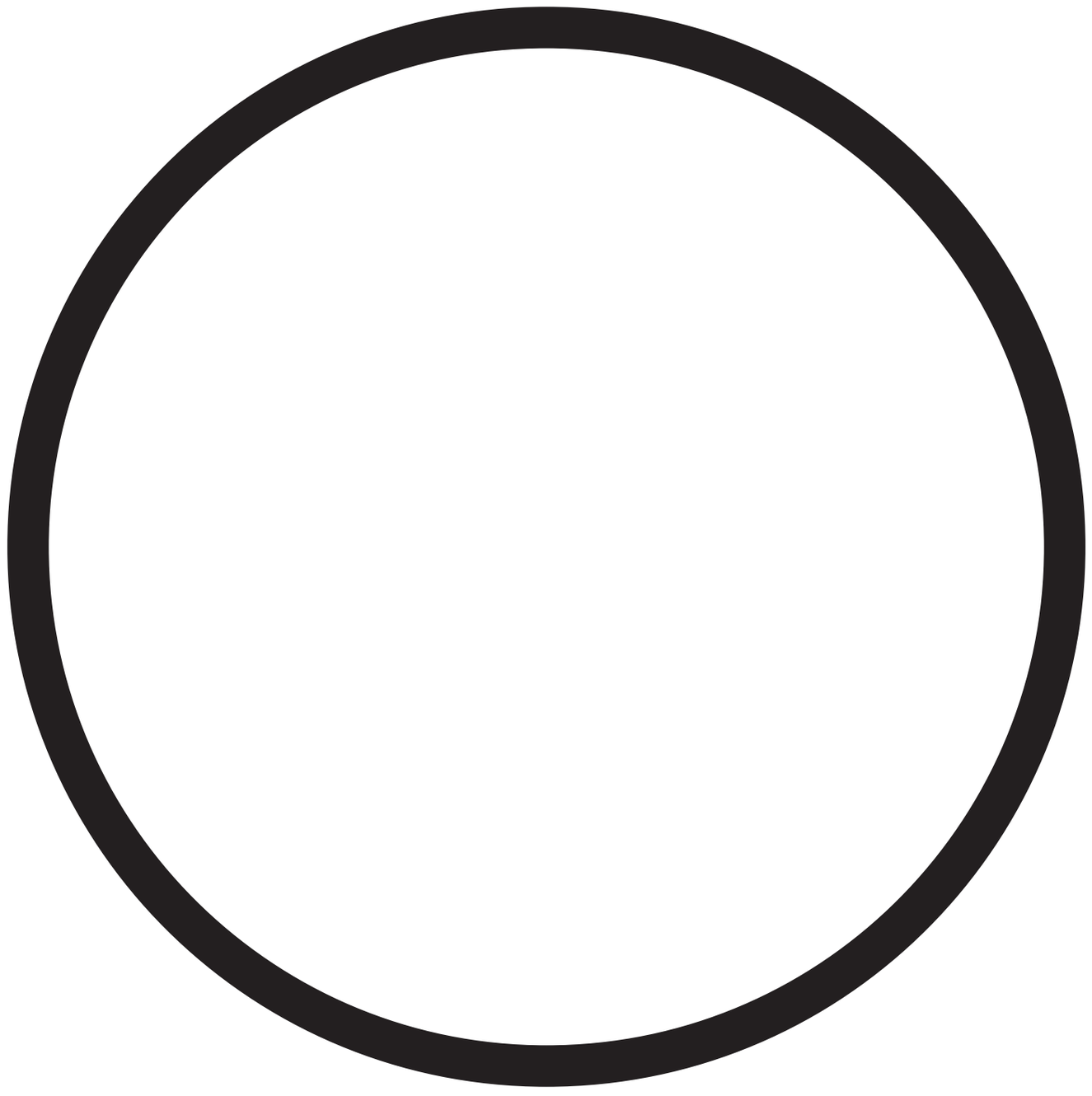
- Matches or exceeds performance of hand-vectorized kernels
- ICC extremely efficient in eliminating temporaries (even auto's)
- No problem with scaling to kernels in the 10,000s of instructions
- Promising performance on automatic fine-grain loop unrolling

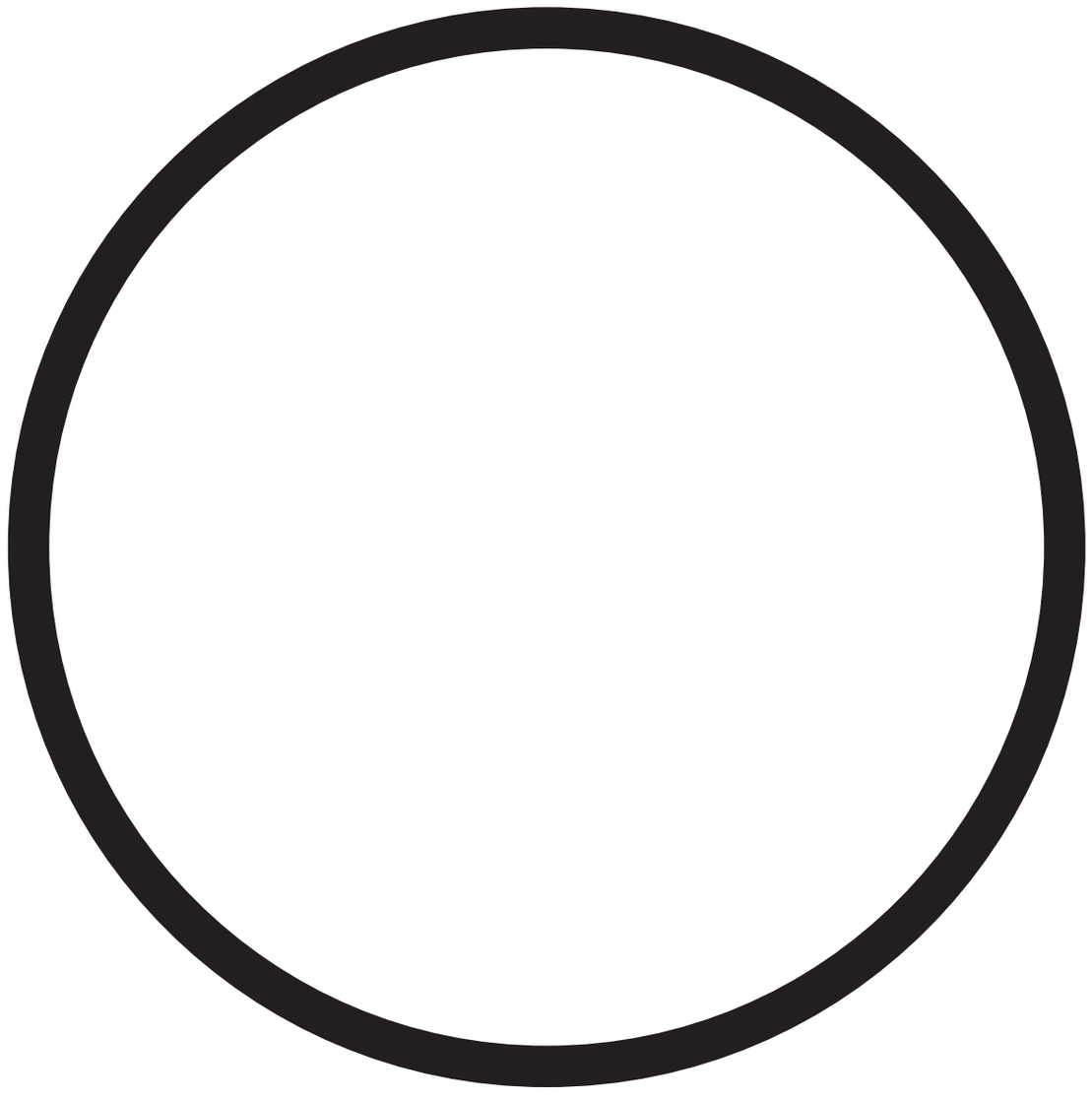
Self collisions : Can we use level sets?



Self collisions : Can we use level sets?





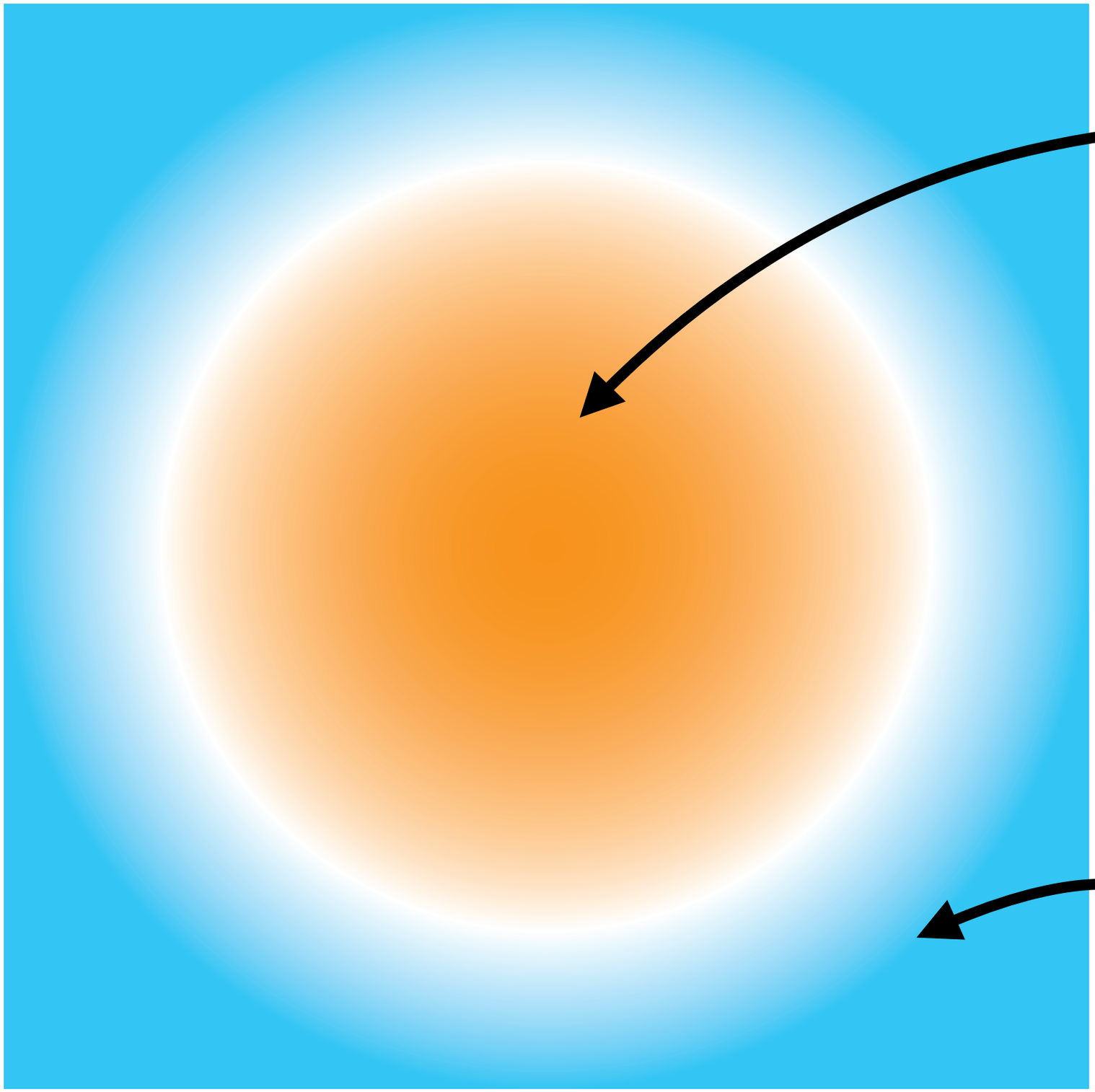


Implicit Surface

$$\mathcal{C} = \{(x, y) | \phi(x, y) = 0\}$$

Signed Distance Field

$$\phi(x, y) = \sqrt{x^2 + y^2} - 1$$



Inside

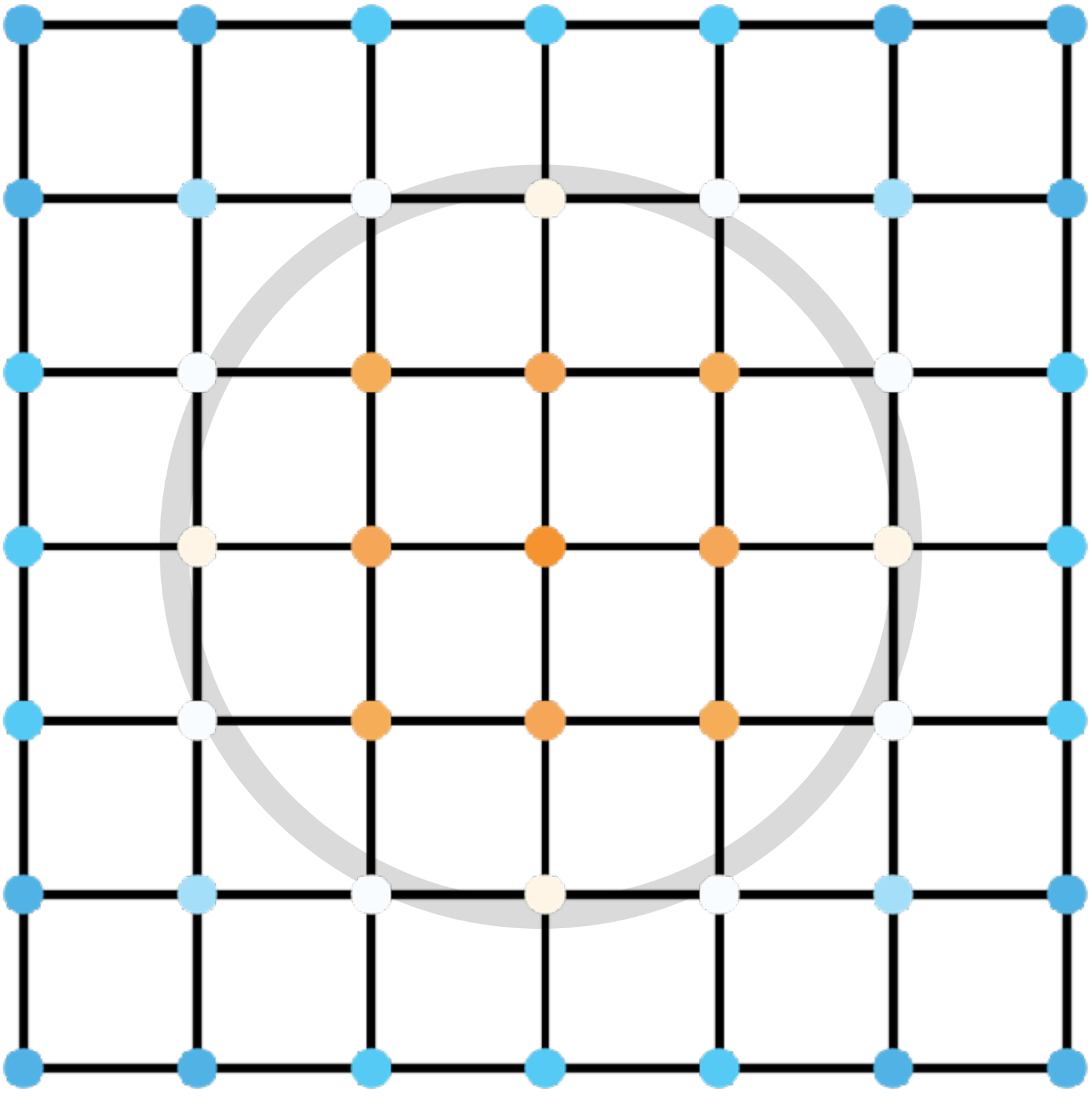
(Negative Φ Values)

Signed Distance Field

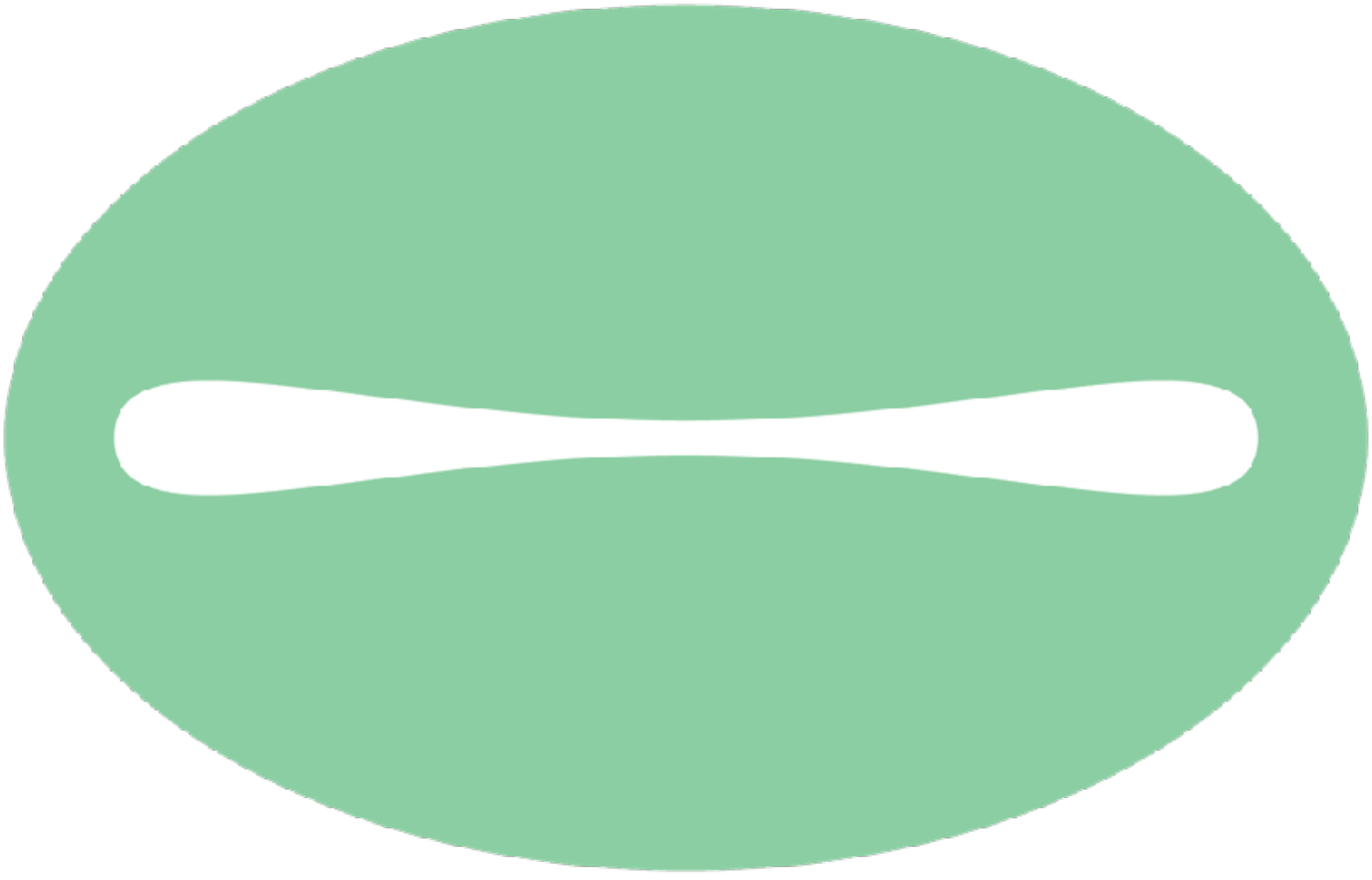
$$\phi(x, y) = \sqrt{x^2 + y^2} - 1$$

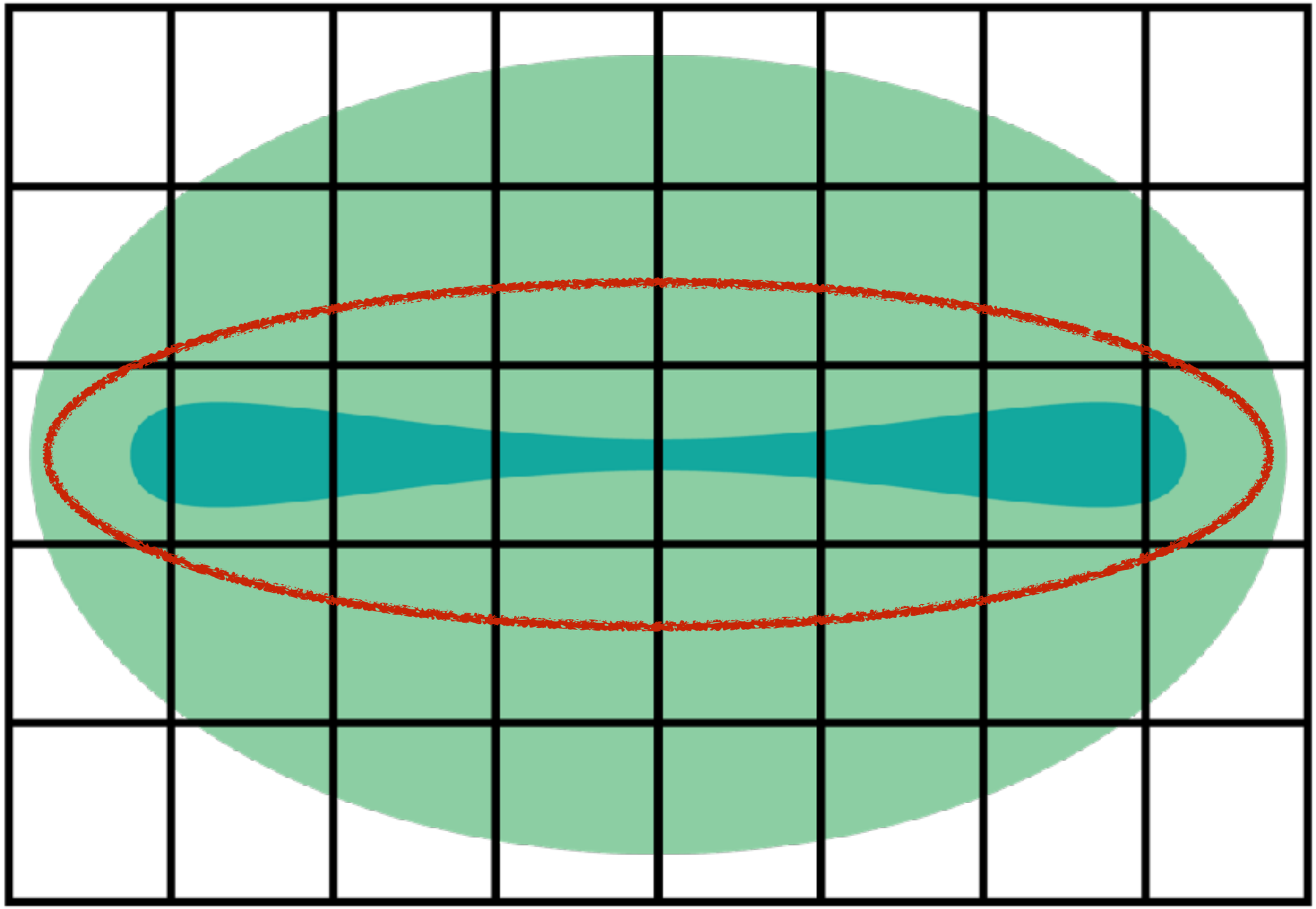
Outside

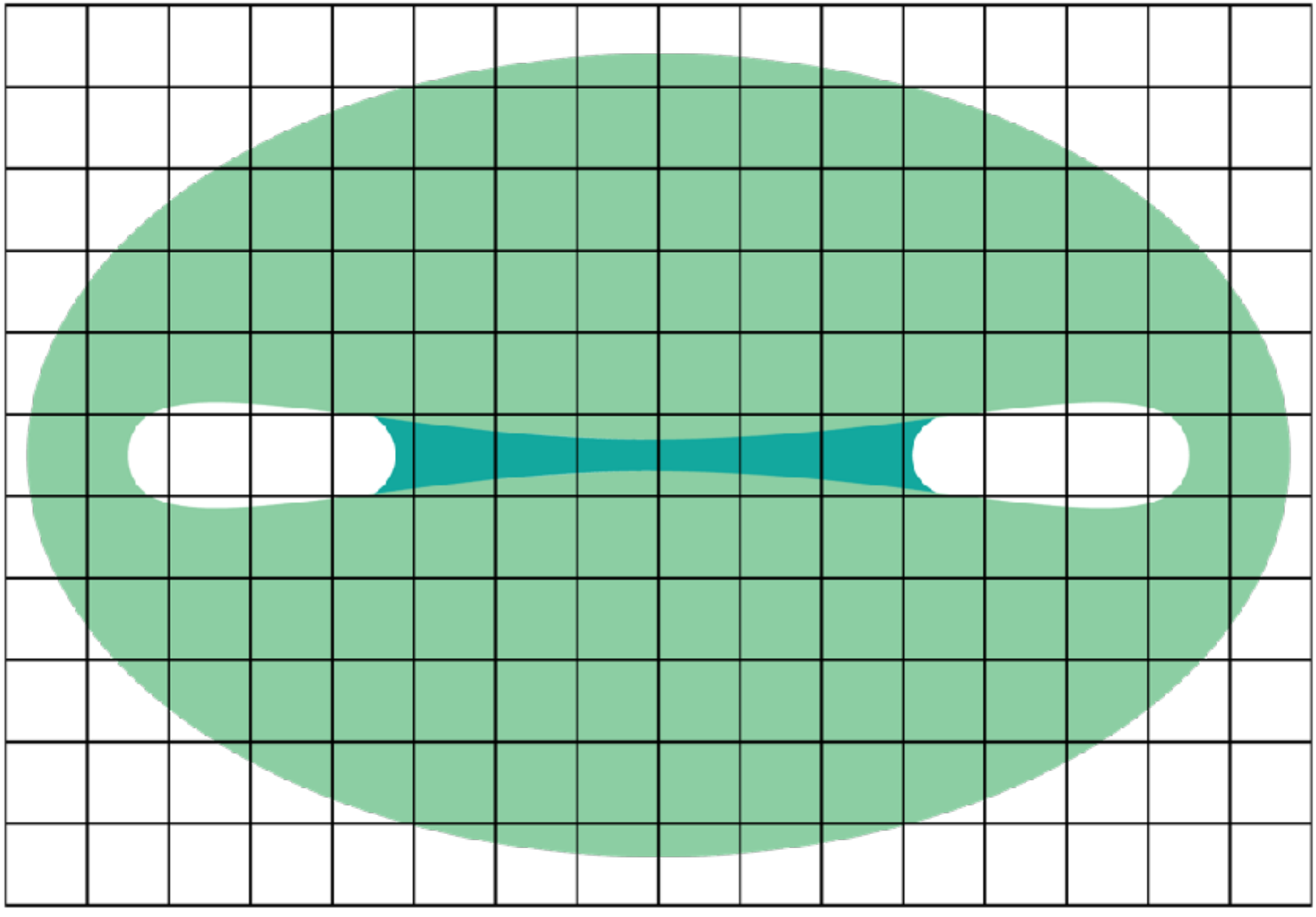
(Positive Φ Values)

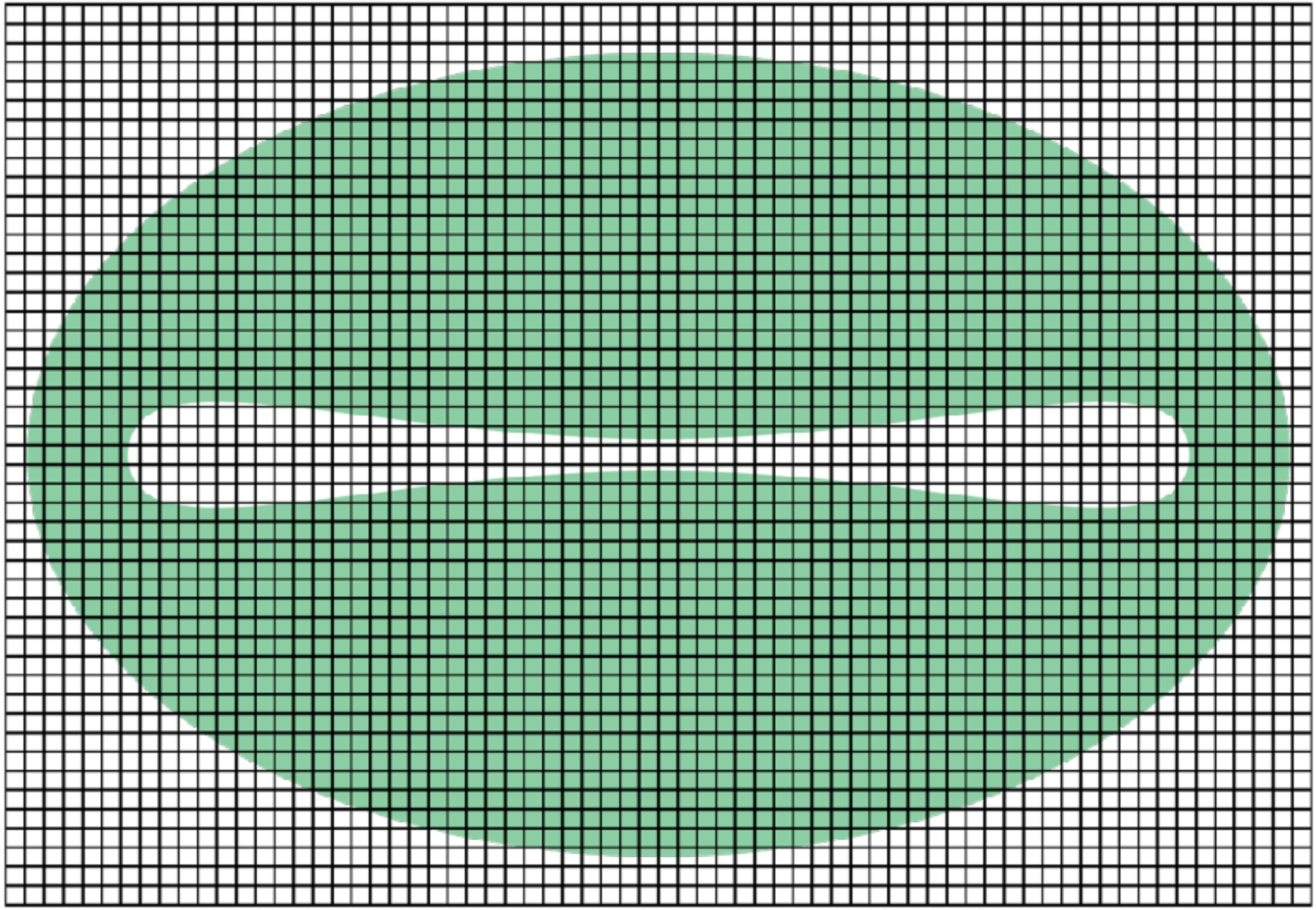


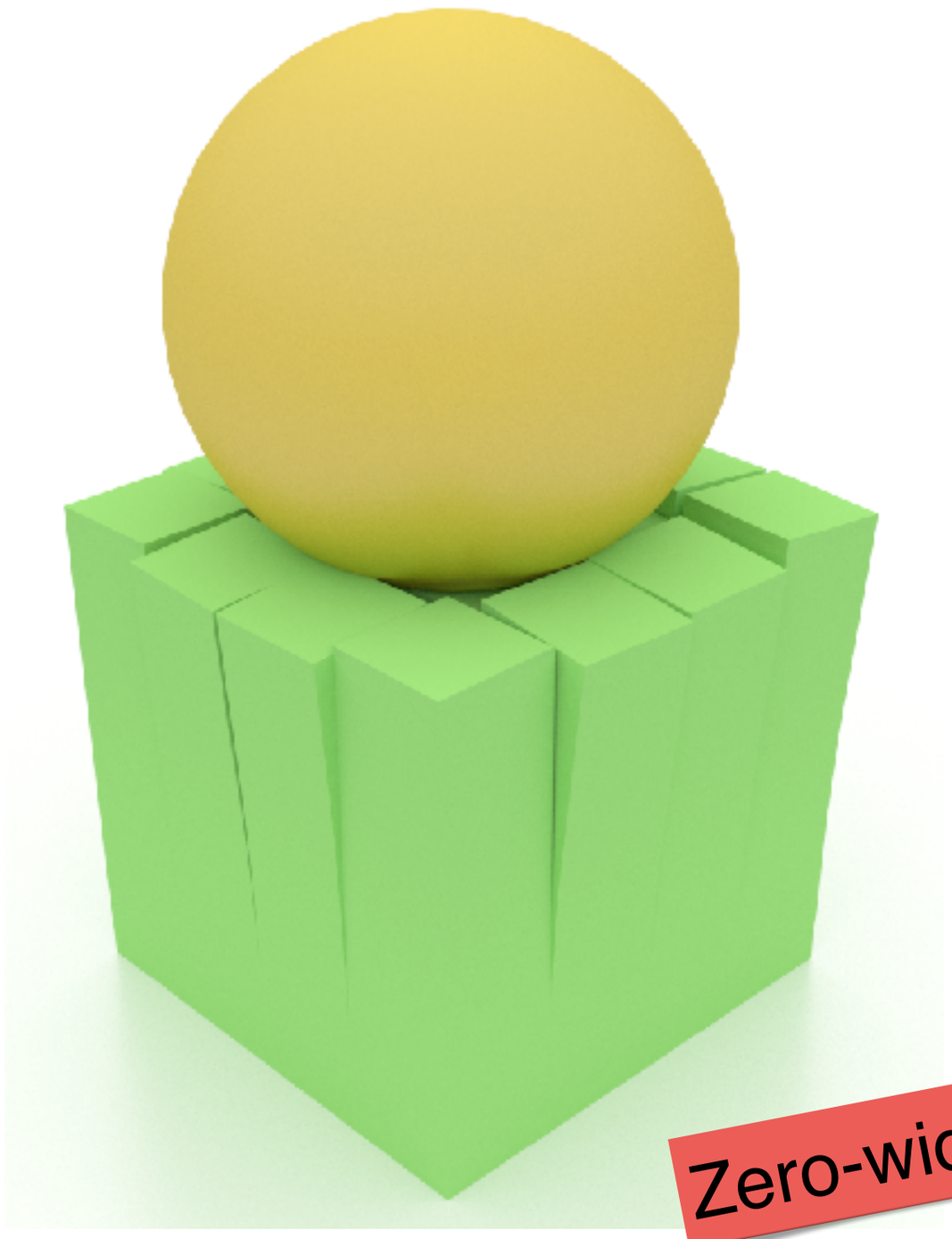
Discrete
Signed Distance Field

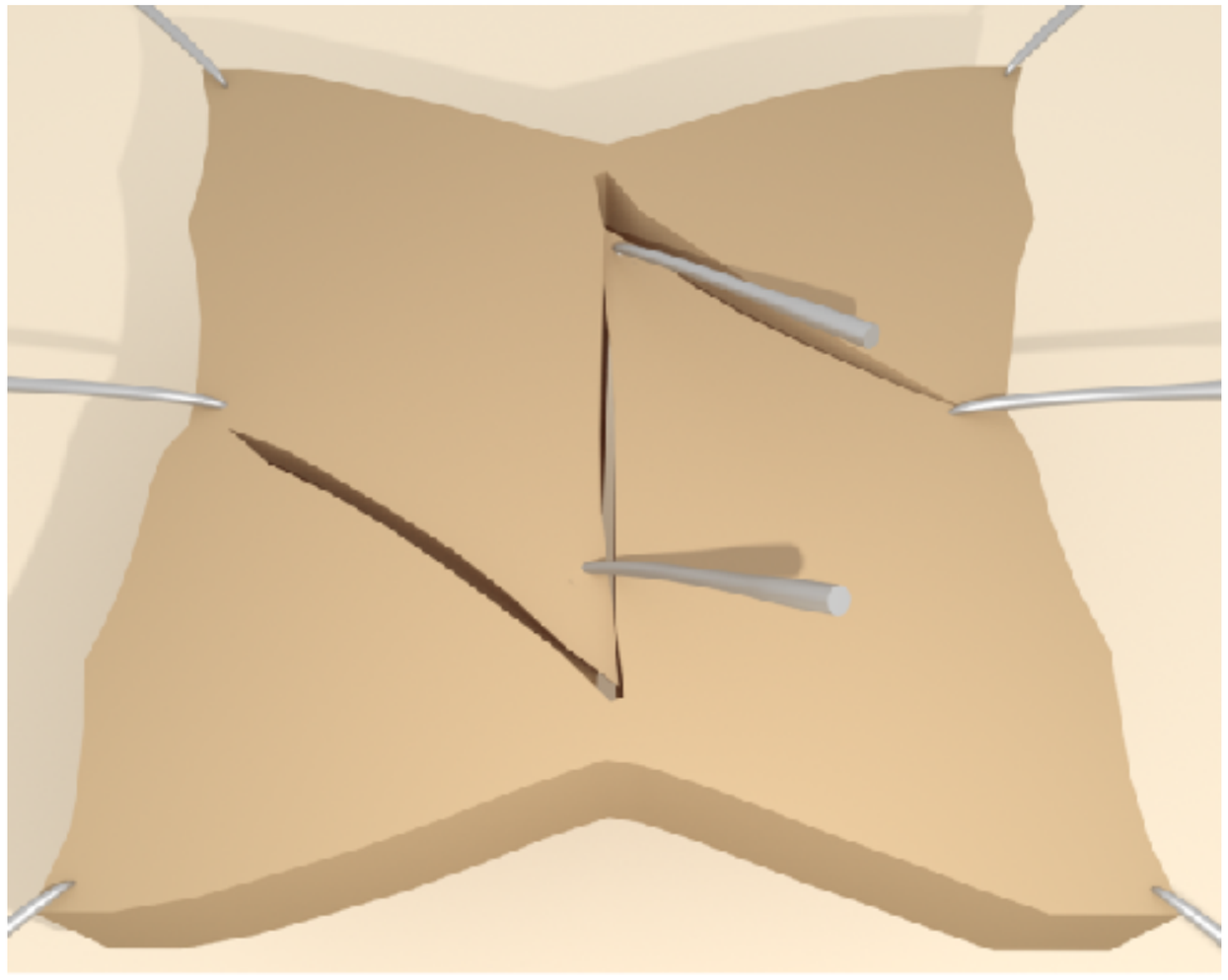






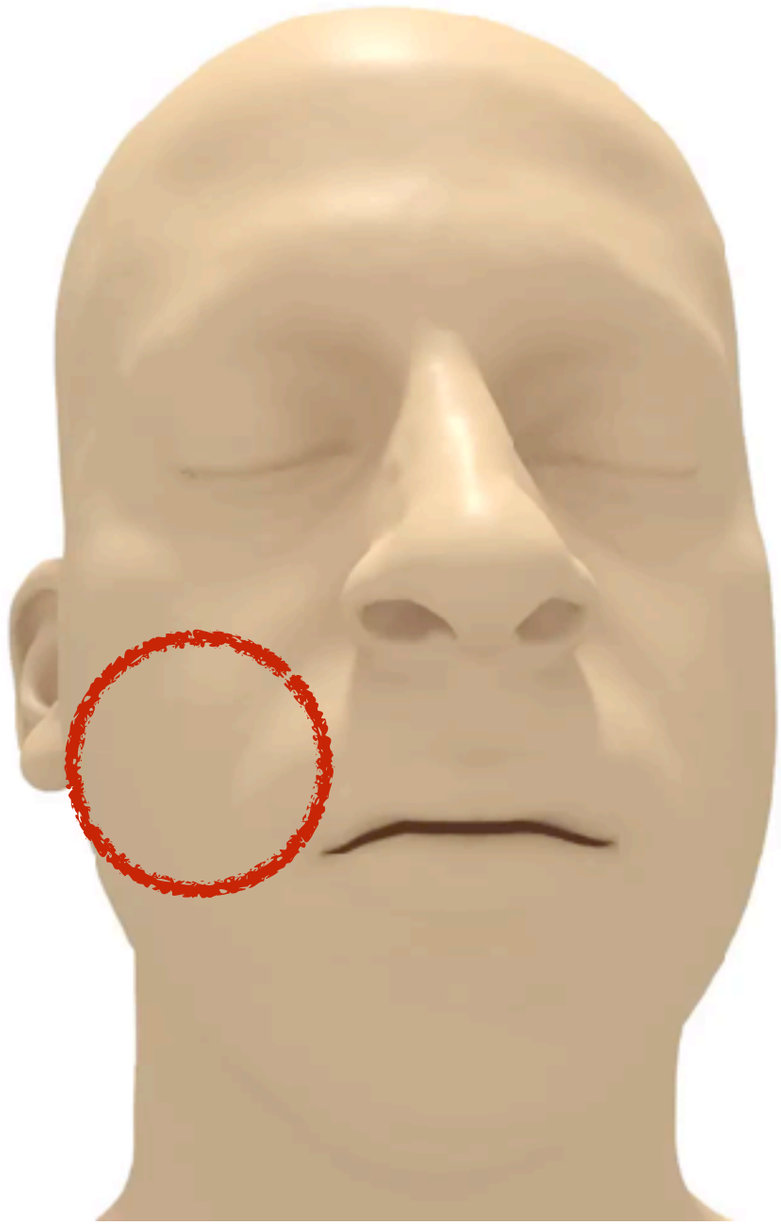


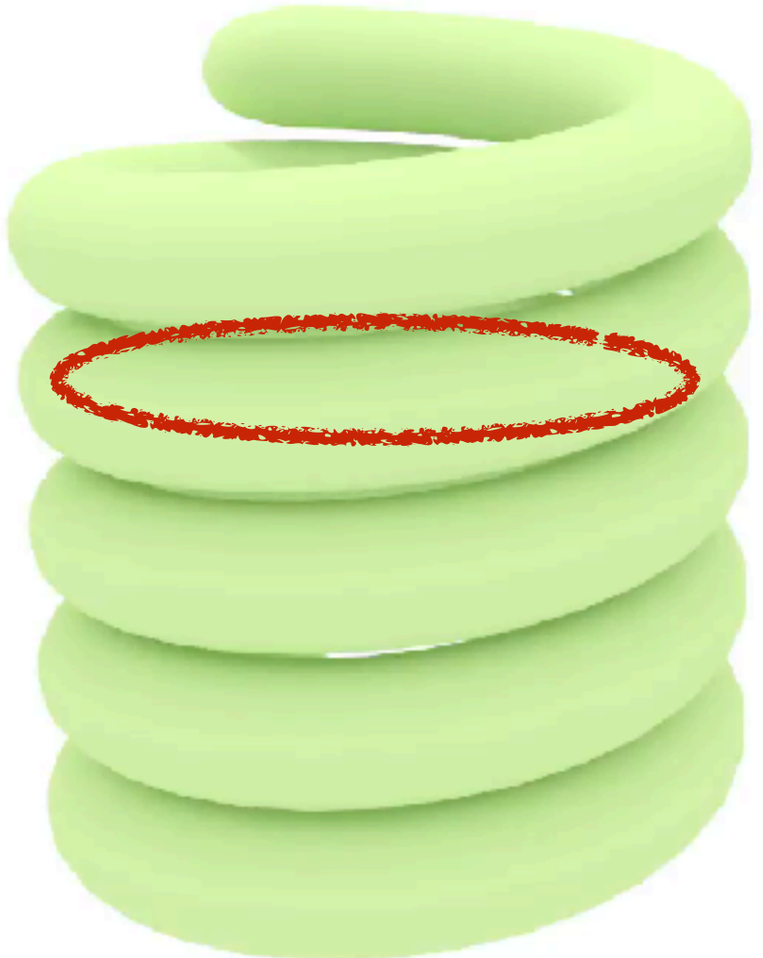




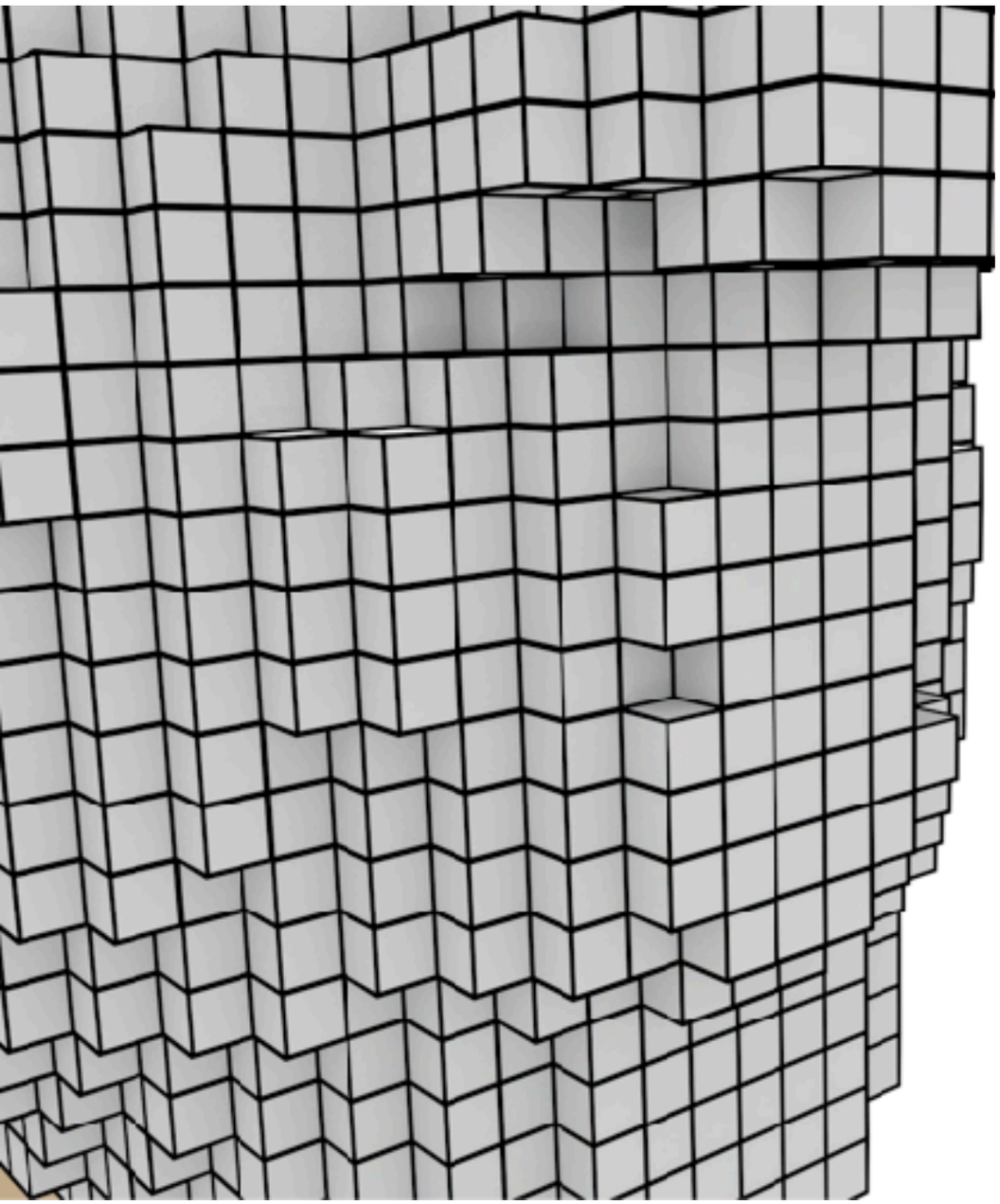
Zero-width cut; extra res won’t help

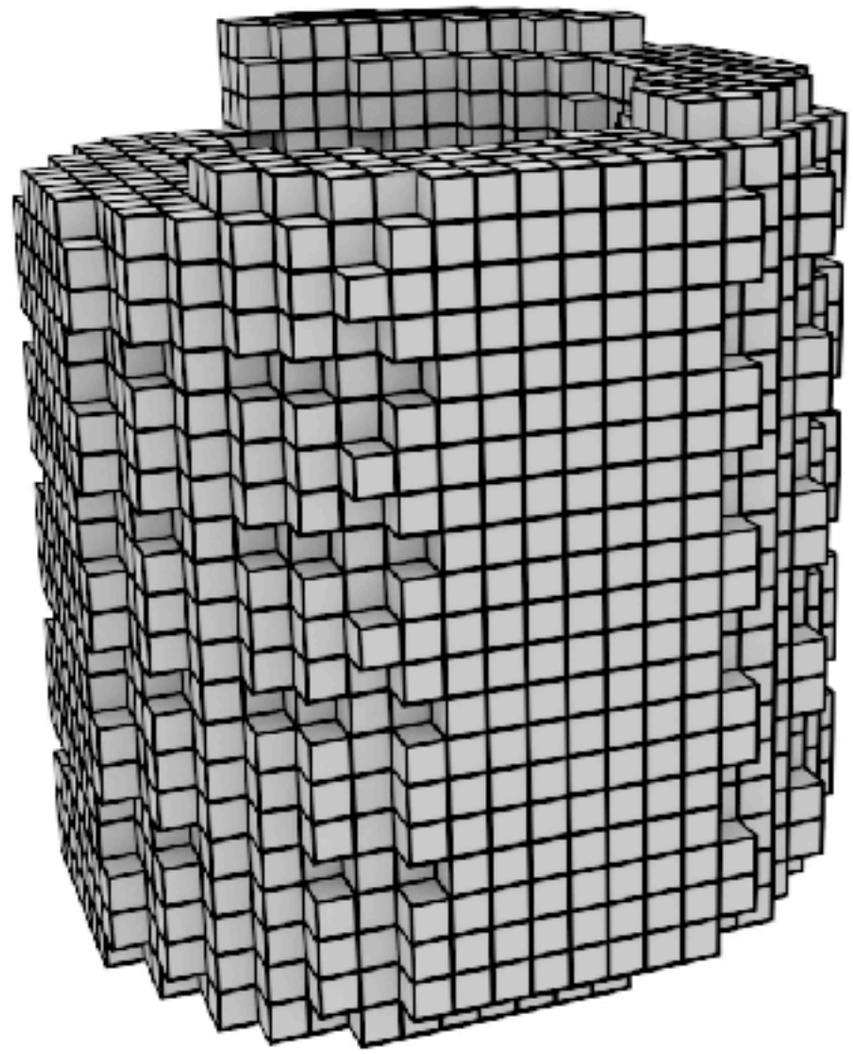
Signed distances on embedding mesh

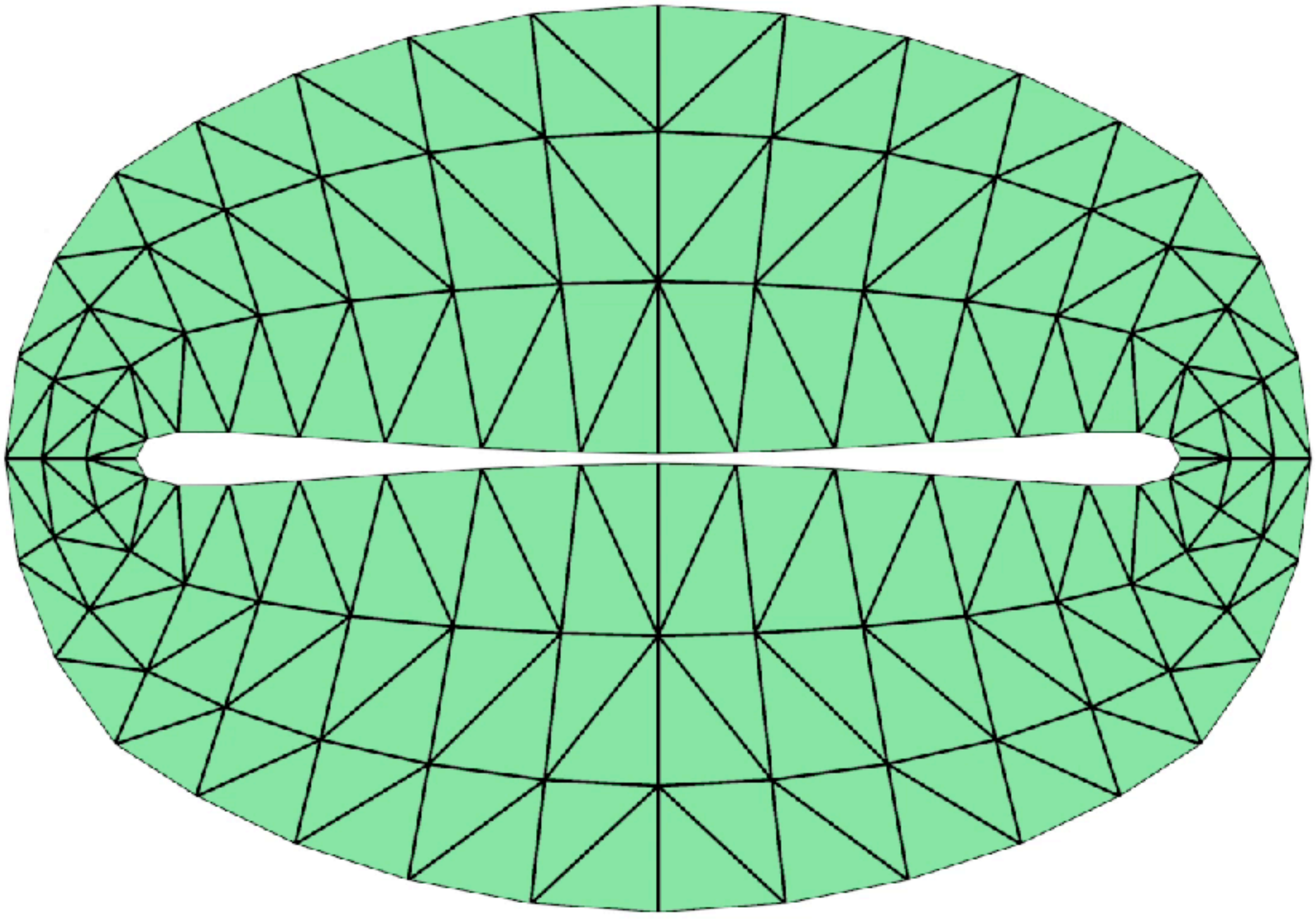


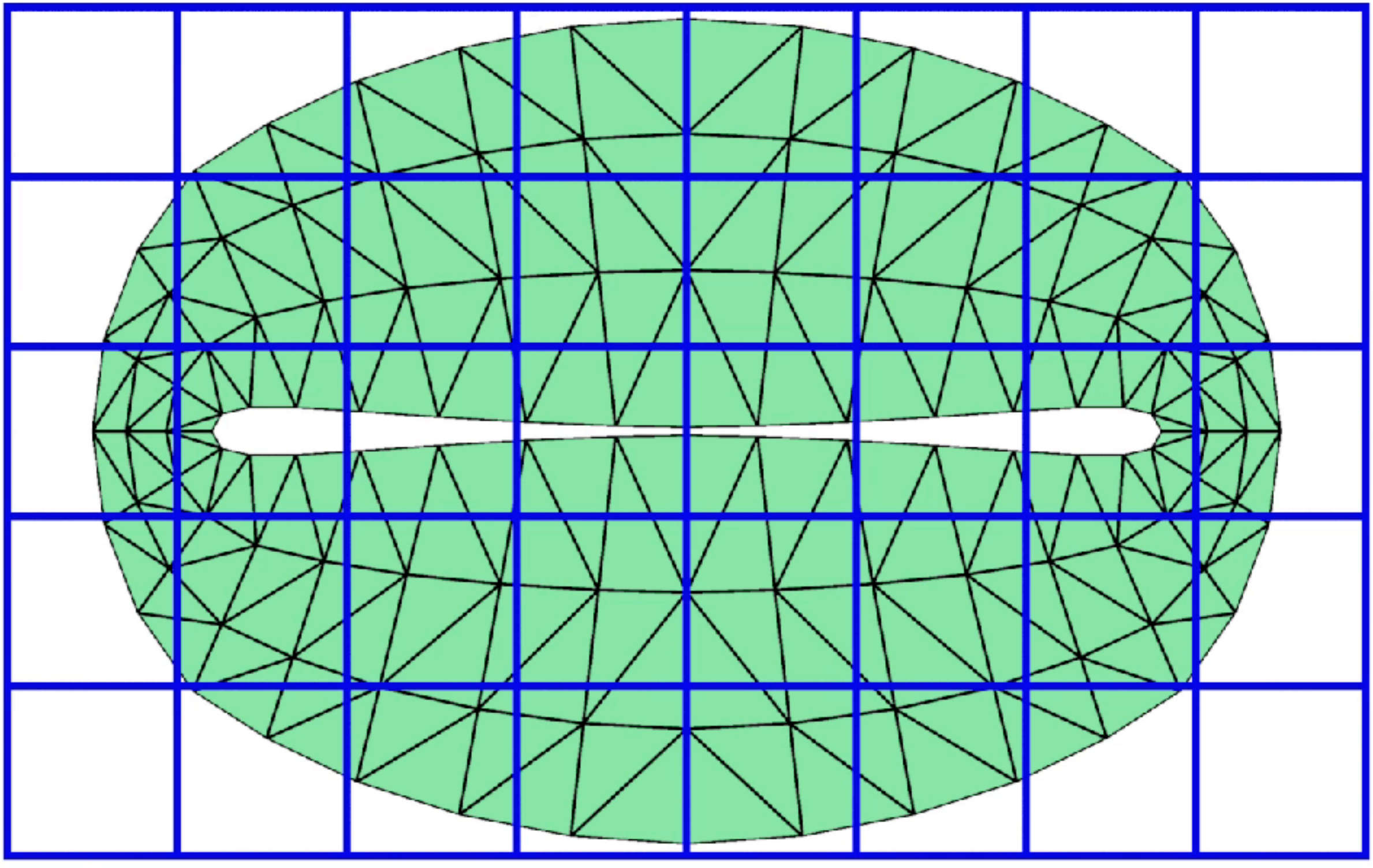


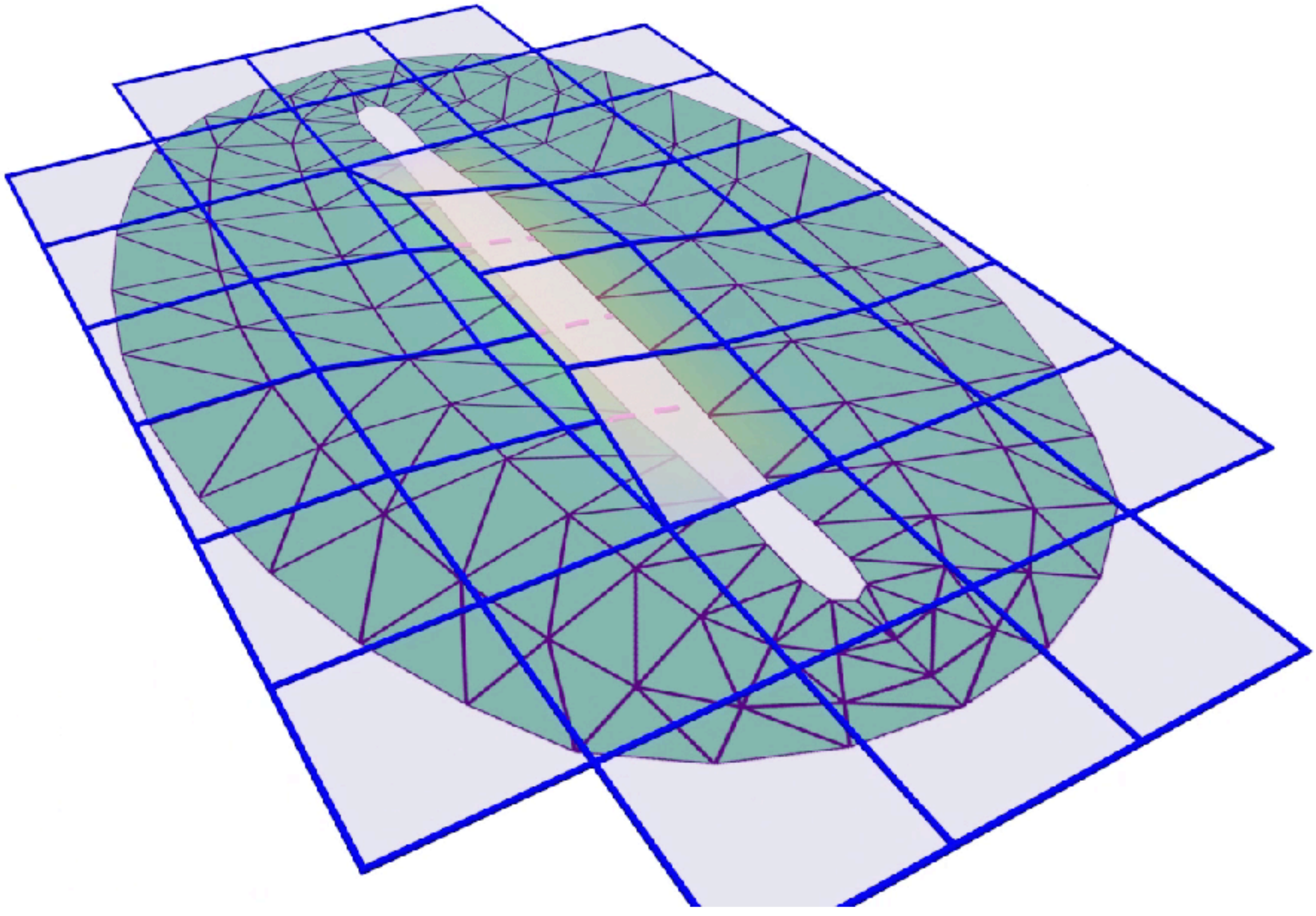
Signed distances on embedding mesh



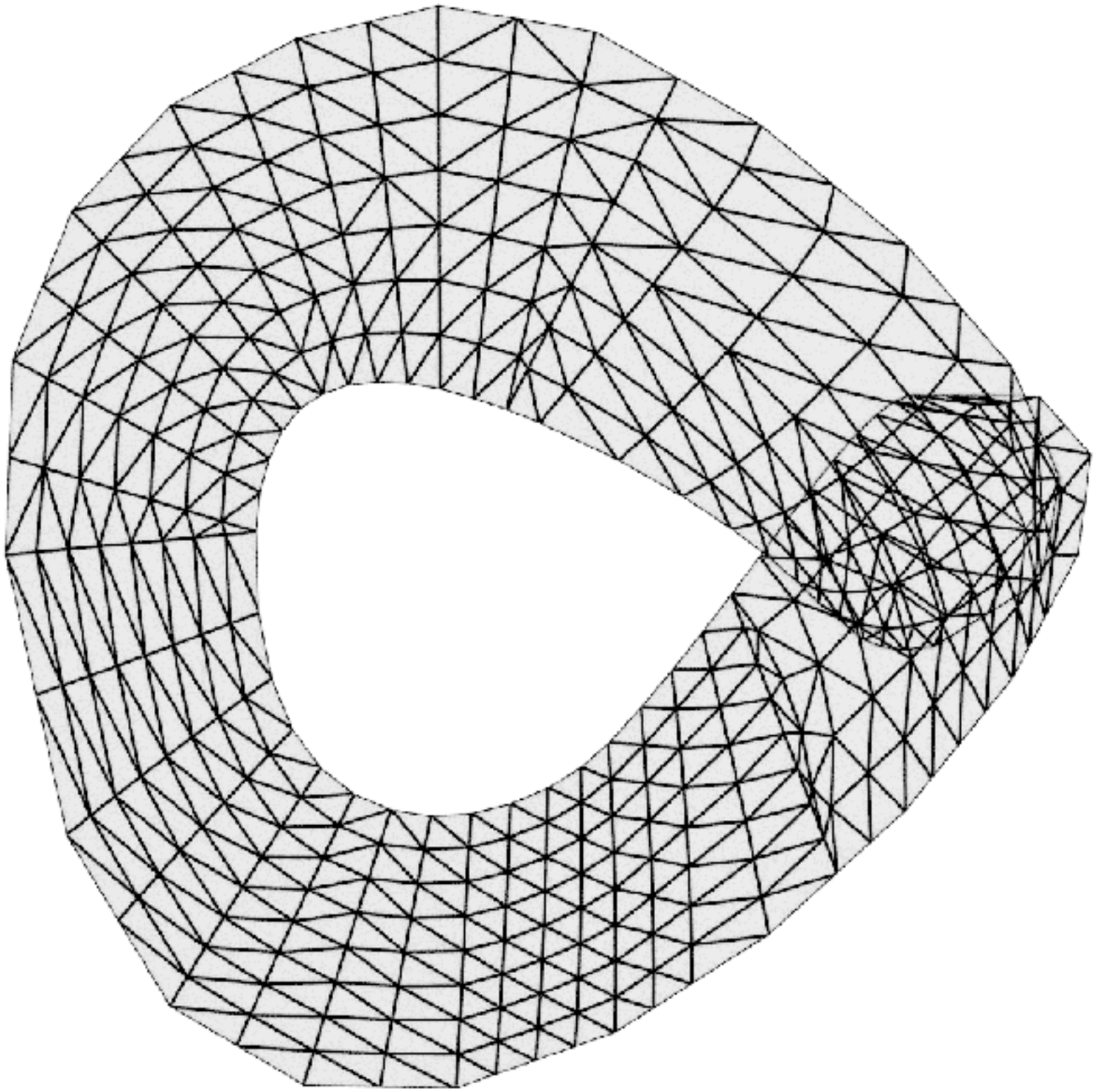




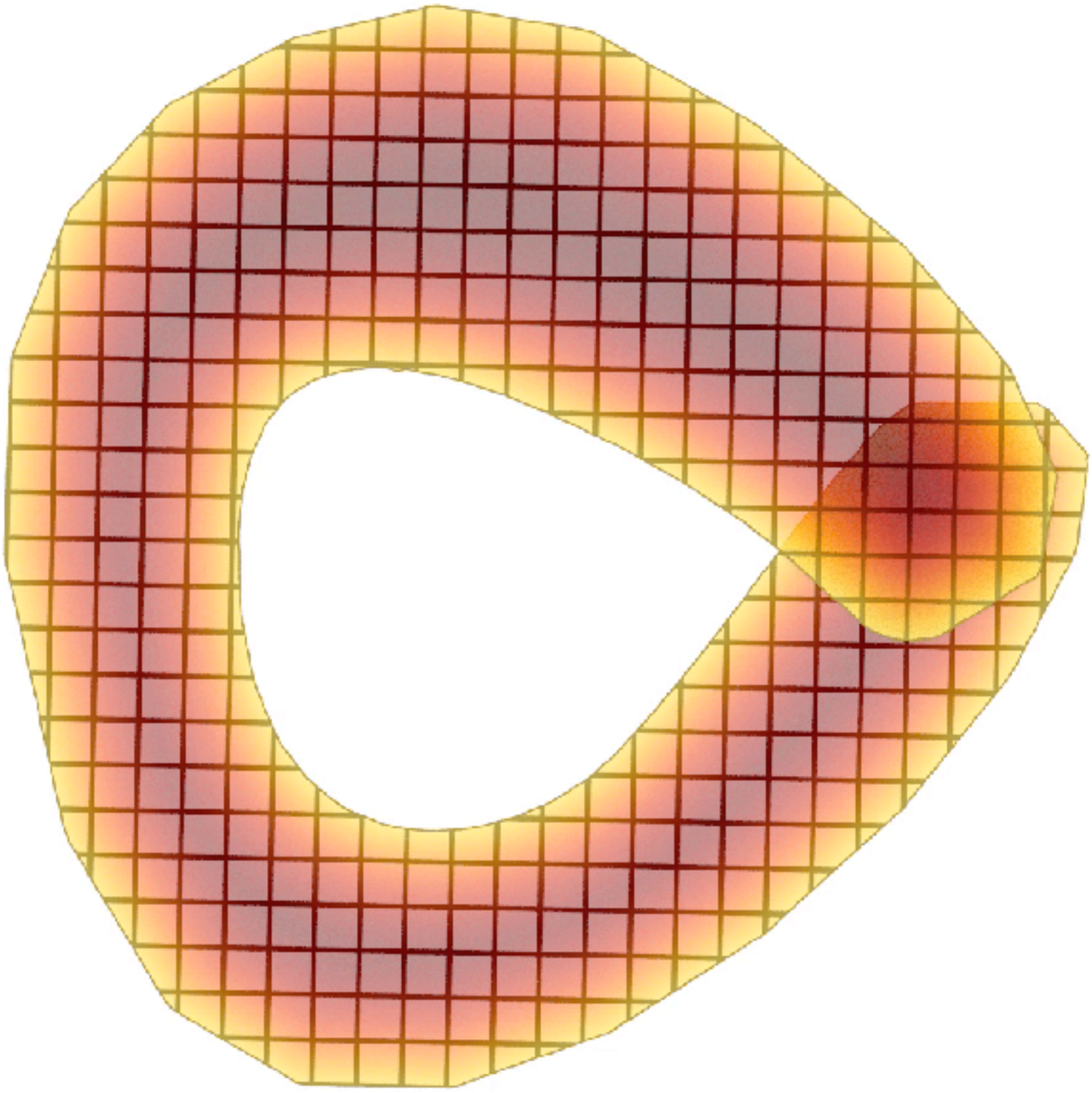




Digression; overlapping mesh works!



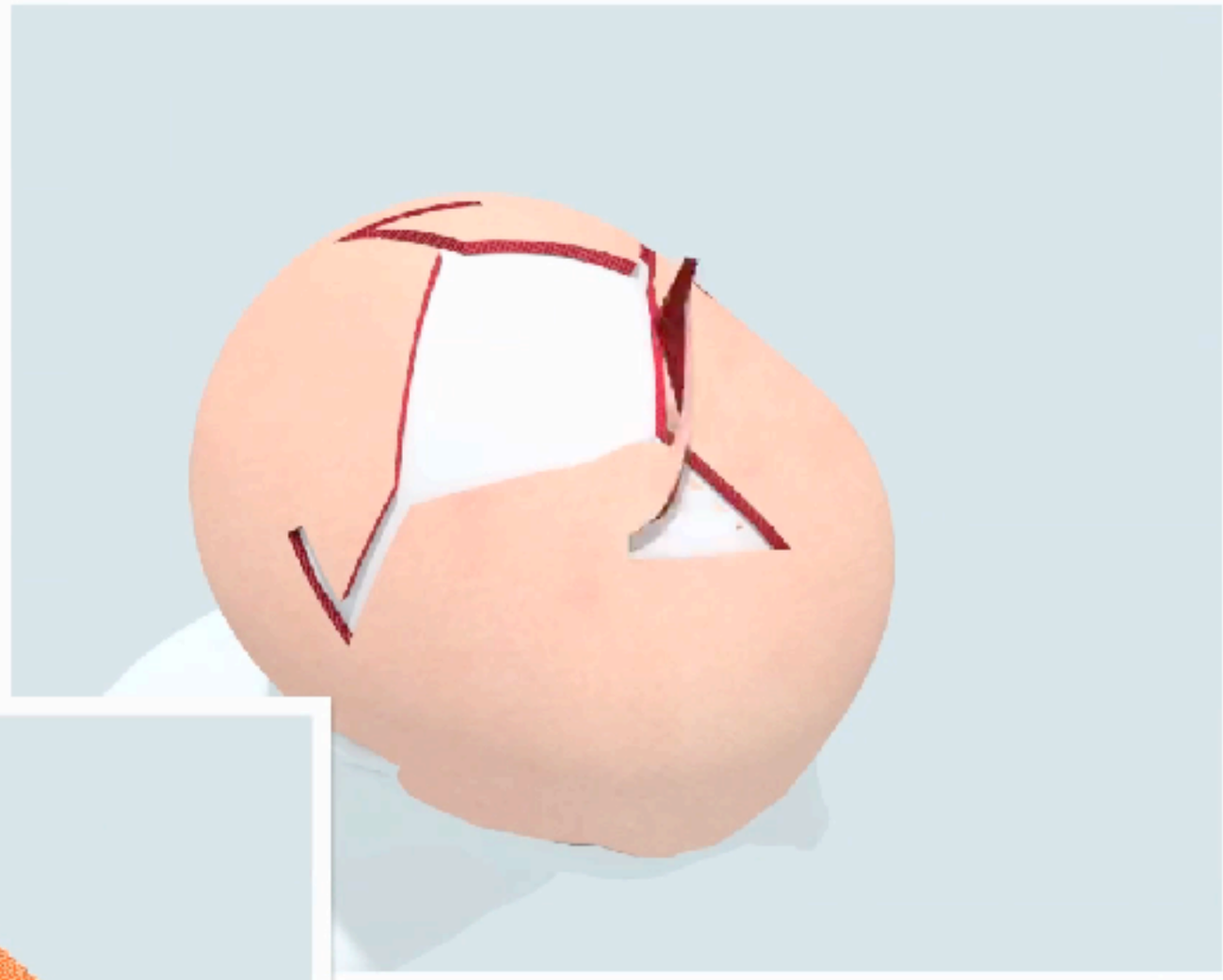
Digression; overlapping mesh works!

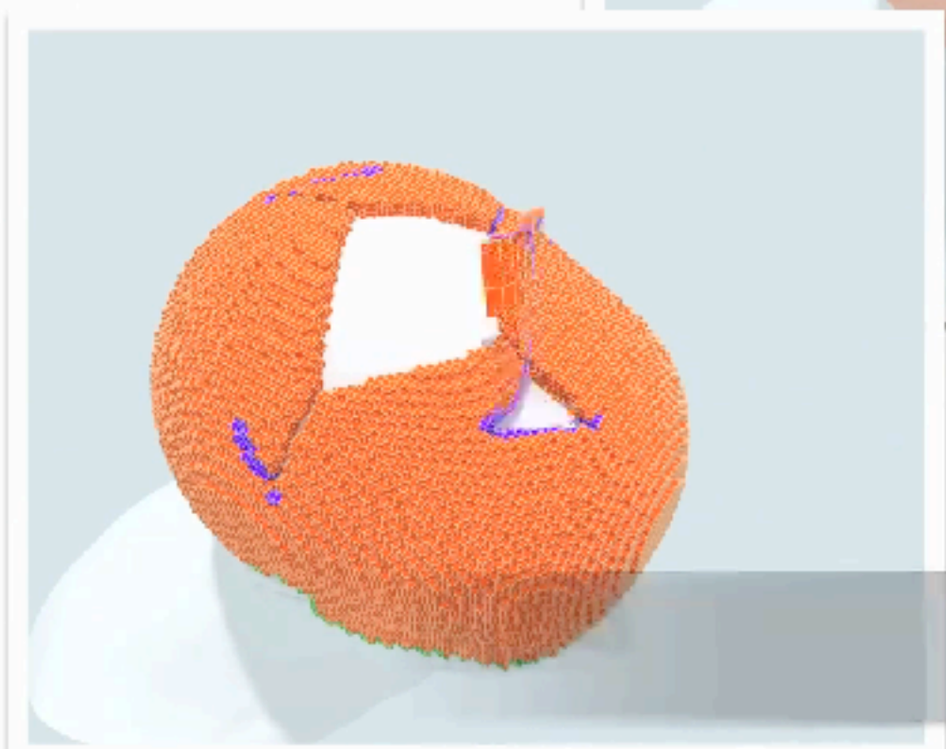


Current challenges?

Materials are inaccurate

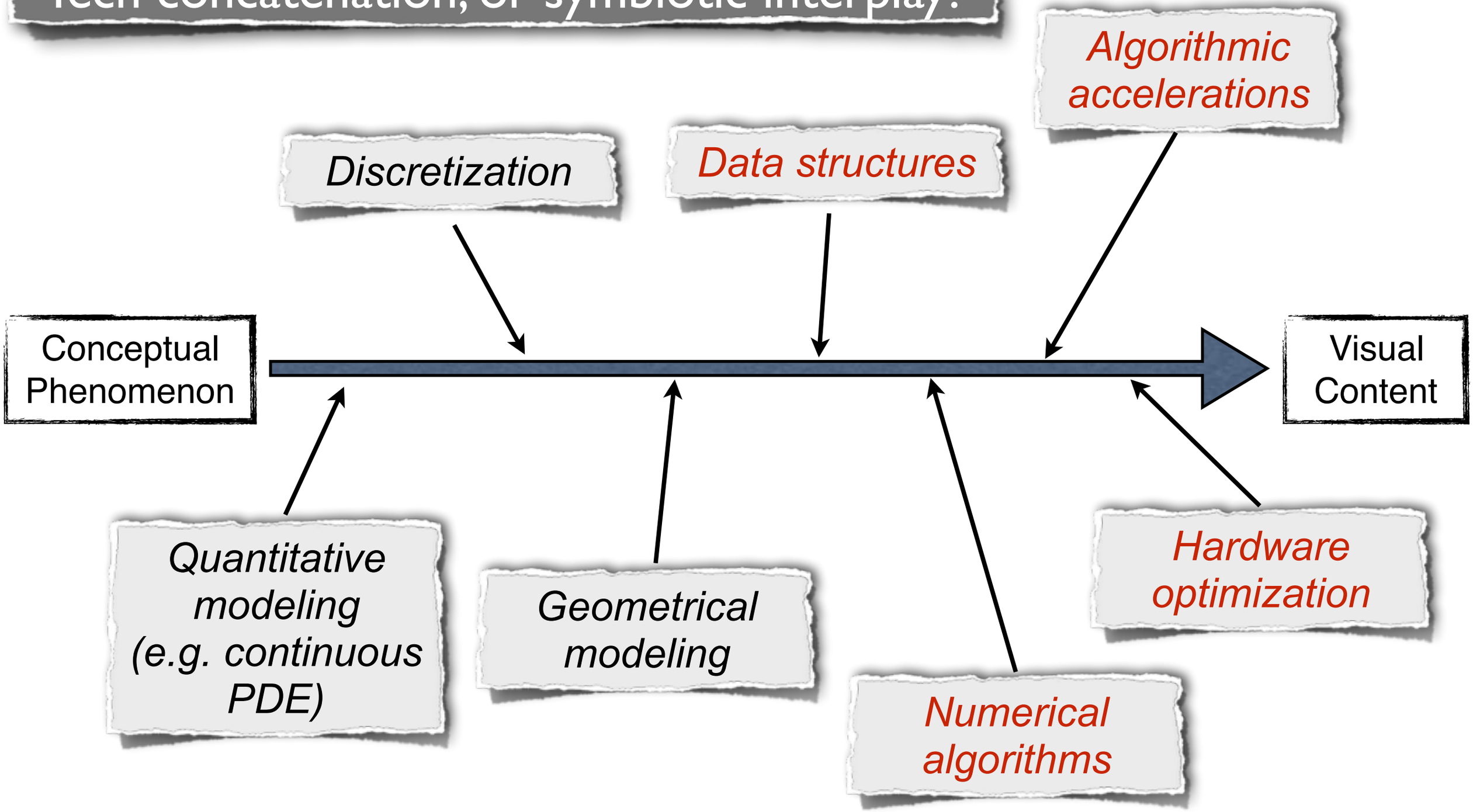
- Way too permissive
- Can't answer *"will it reach?"*
- Stress patterns don't always match our collaborators' experience





Dufourmentel Mouly on Scalp

Tech concatenation, or symbiotic interplay?



Are we pursuing the right efficiency?

RANT ALERT!

Well-intended evaluation practices ...

> *"My serial implementation of algorithm X on machine Y ran in Z seconds. When I parallelized my code, I got a speedup of 15x on 16 cores ..."*

... are sometimes abused like this:

> *"... when I ported my implementation to CUDA, this numerical solver ran 200 times faster than my original MATLAB code ..."*

So, what is wrong with that premise?

Are we pursuing the right efficiency?

RANT ALERT!

Watch for warning signs:

- Speedup across platforms grossly exceeding specification ratios
  - e.g. NVIDIA GTX Titan X vs. Intel Xeon E5-2650v4 (Q1/16)
  - Relative (peak) specifications :
    - GPU has about 6x higher (peak) compute capacity
    - GPU has about 4x higher (peak) memory bandwidth
  - Significantly higher speedups likely indicate:
    - Different implementations on the 2 platforms
    - Baseline code was not optimal/parallel enough
- "Standard" parallelization yields linear speedups on **many** *cores*
  - [Reasonable scenario] Implementation is CPU-bound
  - [Problematic scenario] Implementation is CPU-wasteful

Are we pursuing the right efficiency?

RANT ALERT!

A different perspective ...

> *" ... after optimizing my code, the runtime is about 5x slower than **the best possible performance** that I could expect from this machine ..."*

... i.e. 20% of maximum theoretical ***efficiency!***

Challenge : *How can we tell how fast the best implementation could have been?*
(without implementing it ...)

Are we pursuing the right efficiency?

RANT ALERT!

Example : Solving the quadratic equation

$$ax^2 + bx + c = 0$$

What is the *minimum* amount of time needed to solve this?

Data access cost bound

*"We cannot solve this faster than the time needed to read **a,b,c** and write **x**"*

"We cannot solve this faster than the time needed evaluate the polynomial, for given values of a,b,c and x"
(i.e. 2 ADDs, 2 MULTs plus data access)

Solution verification bound

$$ax^2 + bx + c =$$
$$(ax + b)x + c$$

Equivalent operation bound

"We cannot solve this faster than the time it takes to compute a square root"

Are we pursuing the right efficiency?

RANT ALERT!

What about linear systems of equations?

$$\mathbf{Ax} = \mathbf{b}$$

"Textbook Efficiency" (for elliptic systems)

*It is **theoretically possible** to compute the solution to a linear system (with certain properties) with a cost comparable to **10x the cost of verifying** that a given value **x** is an actual solution*

... or ...

*It is **theoretically possible** to compute the solution to a linear system (with certain properties) with a cost comparable to **10x the cost of computing** the expression **r=b-Ax** and verifying that **r=0***

(i.e. slightly over 10x of the cost of a matrix-vector multiplication)

But what algorithm gets there?

Prime Candidates

- Multigrid
- Domain decomposition

In more specialized (unrealistic?) contexts …

- Fourier Analysis
- Cyclic Reduction
- etc ….

What can spoil this potential?

Systemic (?) Challenges

- Irregular domains
- Adaptive discretizations
- Heterogeneous/nonuniform compute platforms
- Anisotropy
- Inhomogeneous operators

Not-so-systemic (hopefully?) challenges

- Nonlinearity
- Contact (????)

Balancing traits of direct & iterative solvers

- **At core of elastic body simulators**
  - Newton iteration for force balance
  - Need to solve **sparse** linearized system
  - Approximate solution good enough

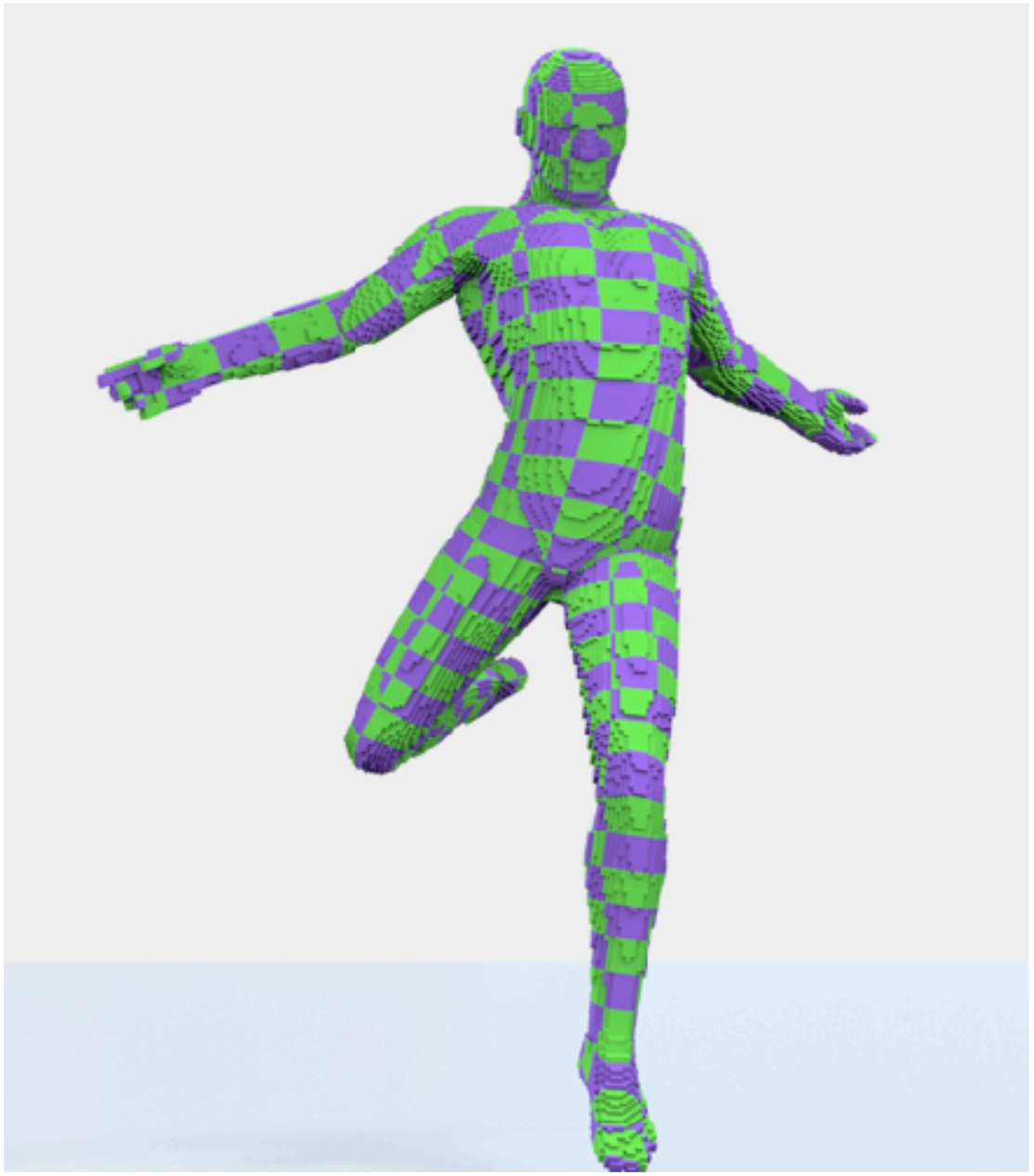
Balancing traits of direct & iterative solvers

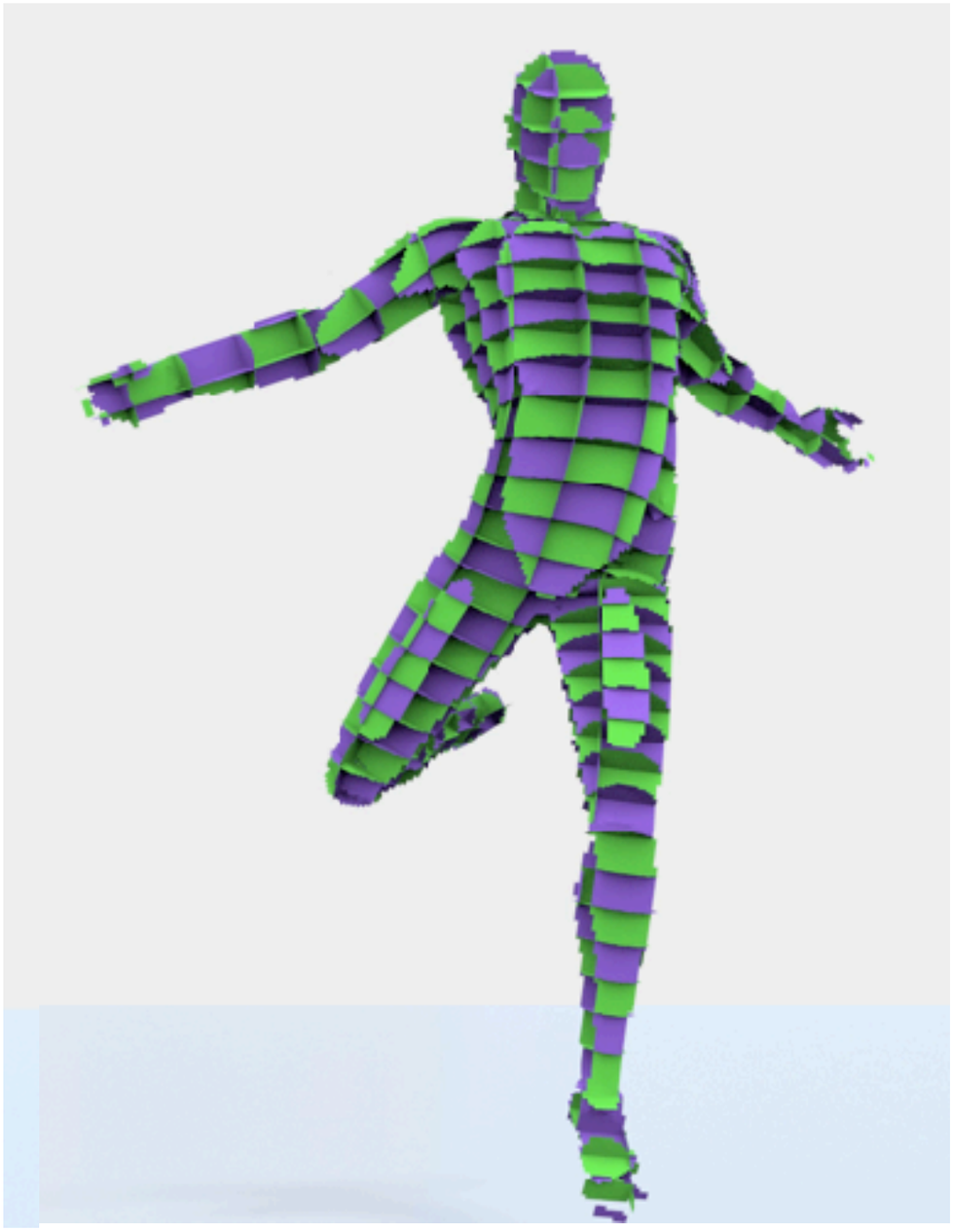
- **Direct or iterative solvers?**
  - Direct methods need little tuning …
  - … but are memory-bound and tough to parallelize
  - Iterative schemes get approximate solution fast …
  - … but require many iterations for large models

Balancing traits of direct & iterative solvers

- **Best of both worlds?**
  - Split work into local problems (elastic blocks)
  - Use direct algebra within blocks
    (and ensure local problem fits in cache)
  - Use iterative solver across block

Balancing traits of direct & iterative solvers

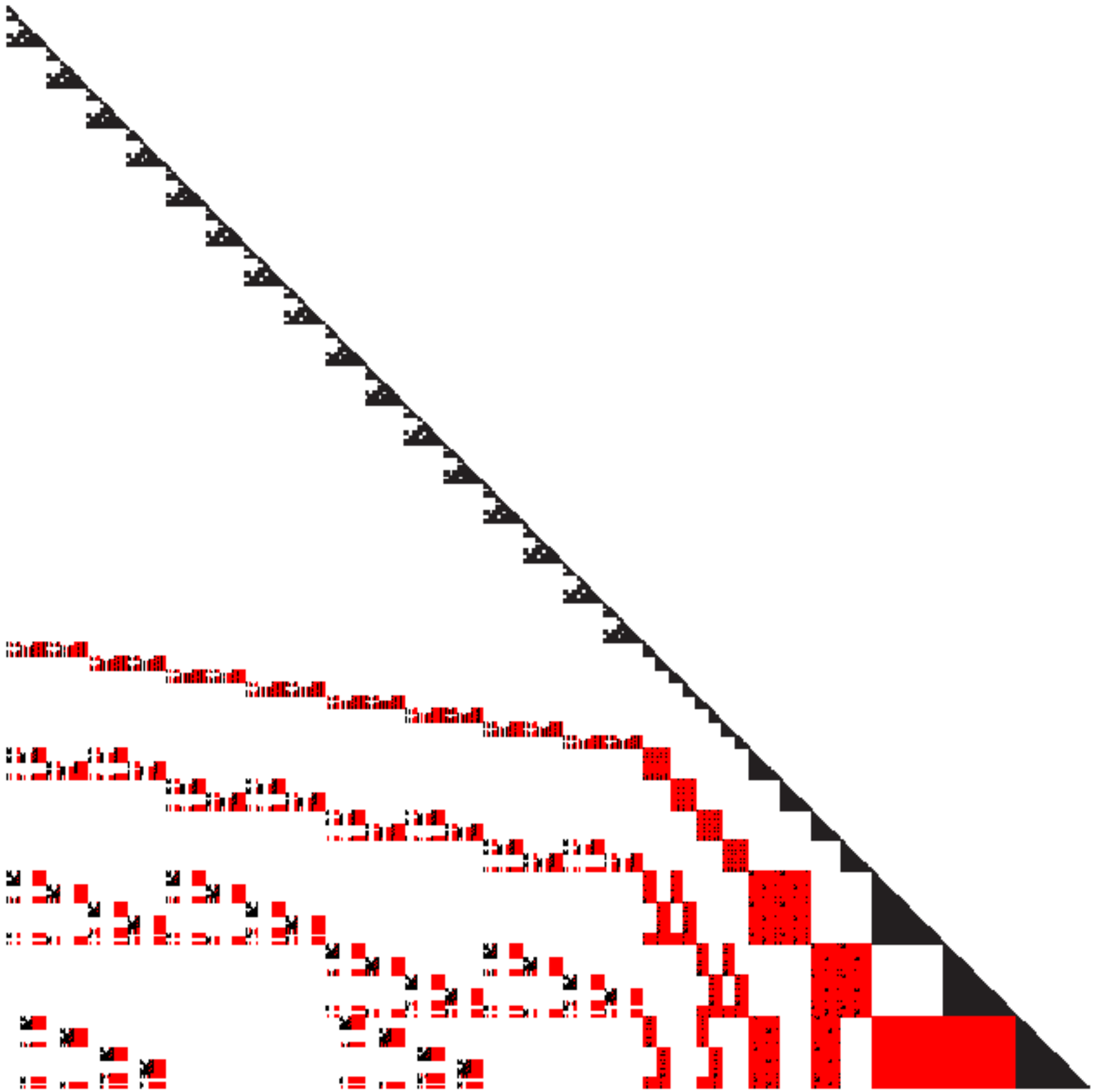




Balancing traits of direct & iterative solvers

- But wait, there's more …
- Direct solver **could** fit in a single core's cache
- But we need to use SIMD for parallelism
- … and be extremely frugal with storage

Balancing traits of direct & iterative solvers



Balancing traits of direct & iterative solvers

- **Challenges**
  - Forward/Backward substitution are inherently serial (at least, textbooks say they are)
  - Uses matrix-vector multiplication (and any of this sort is doomed to be memory bound)
  - No clear opportunity for SIMD (although we are underutilizing computation capability by at least a factor of 100x)

Balancing traits of direct & iterative solvers

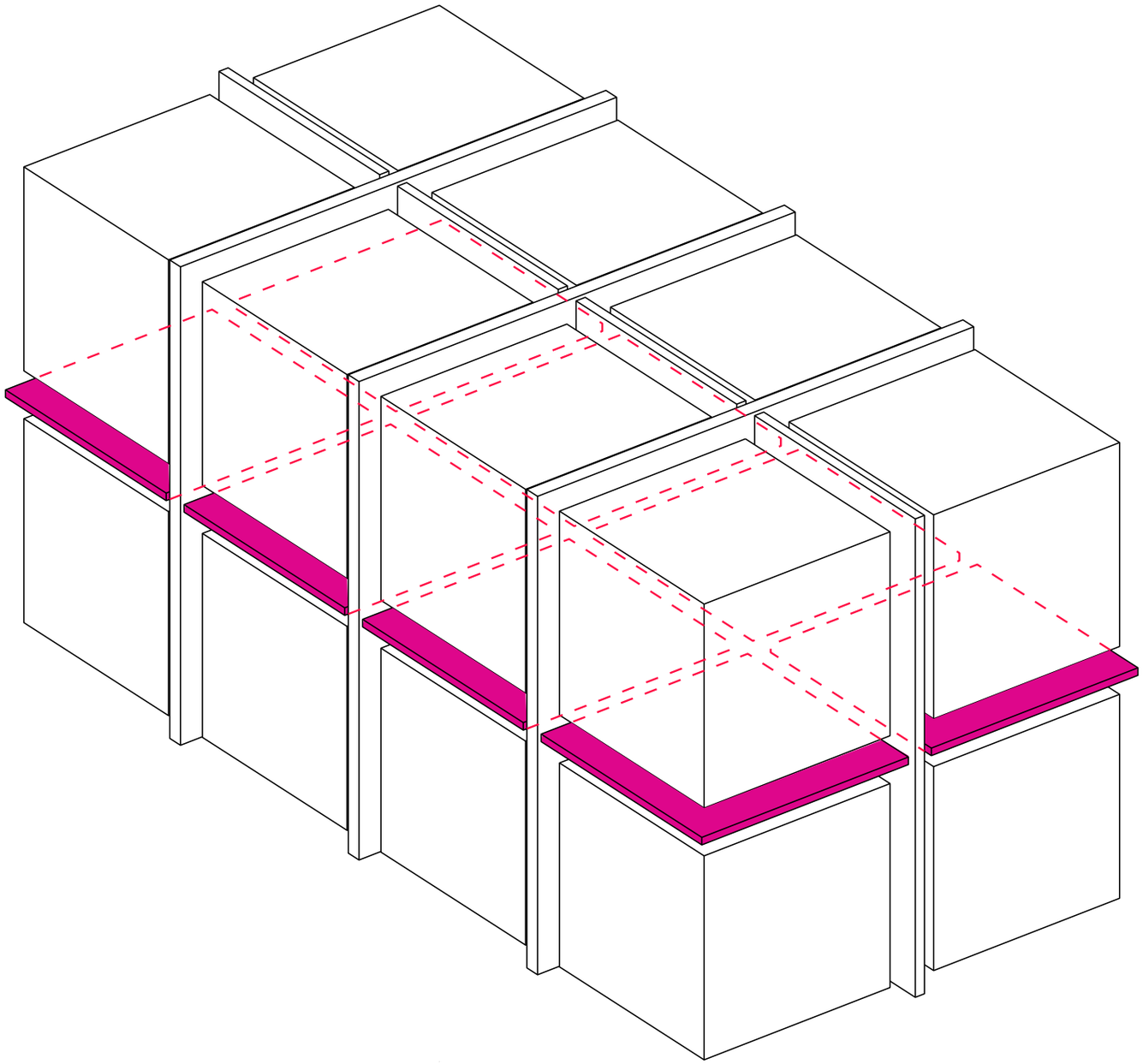
- **Remedies**
  - VPU optimization is <1%.
  - Can easily afford to do 10x more computations,
    if in return we get :
    - 10x reduction in bandwidth requirements, and
    - SIMD opportunities exposed

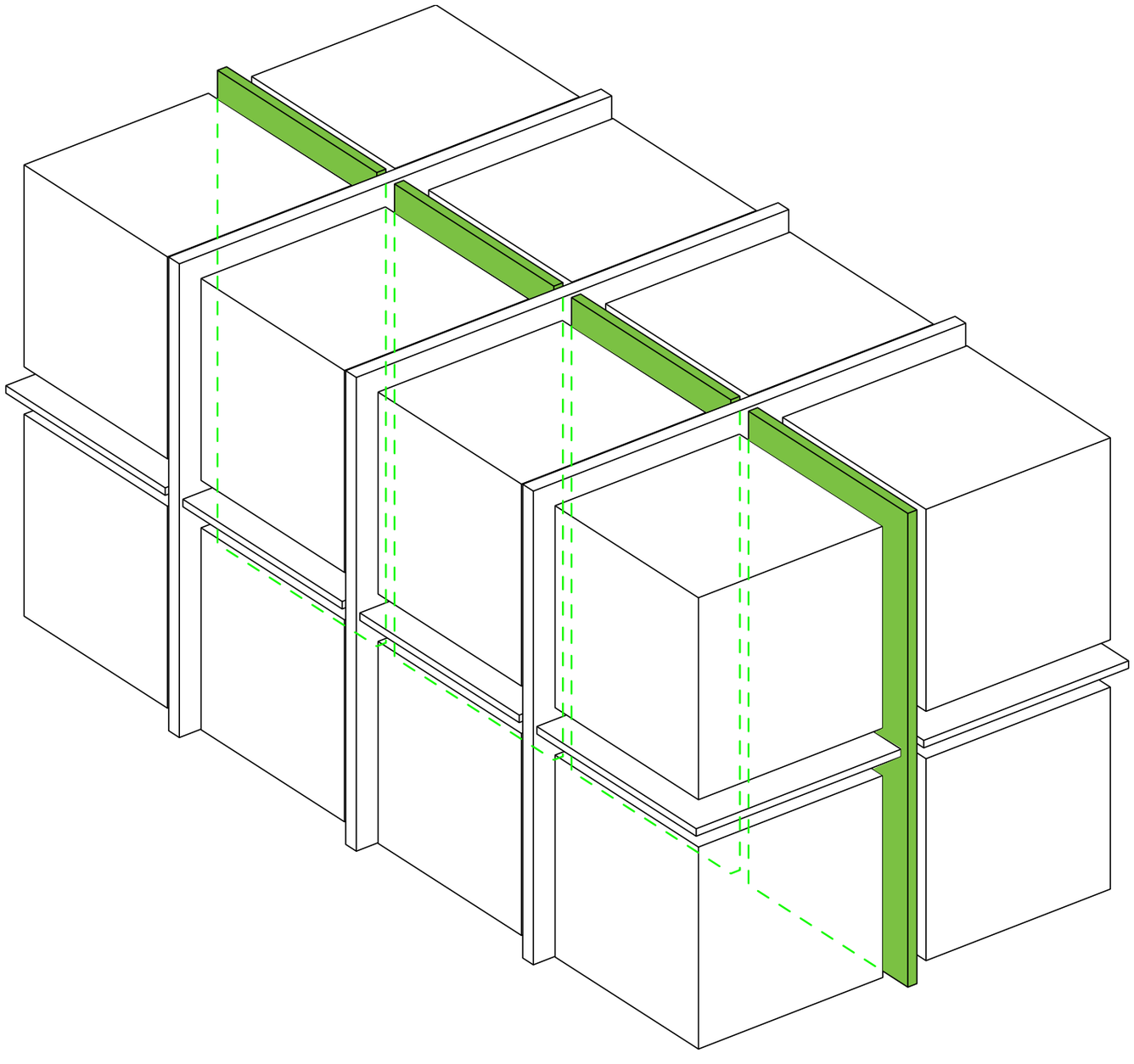
Balancing traits of direct & iterative solvers

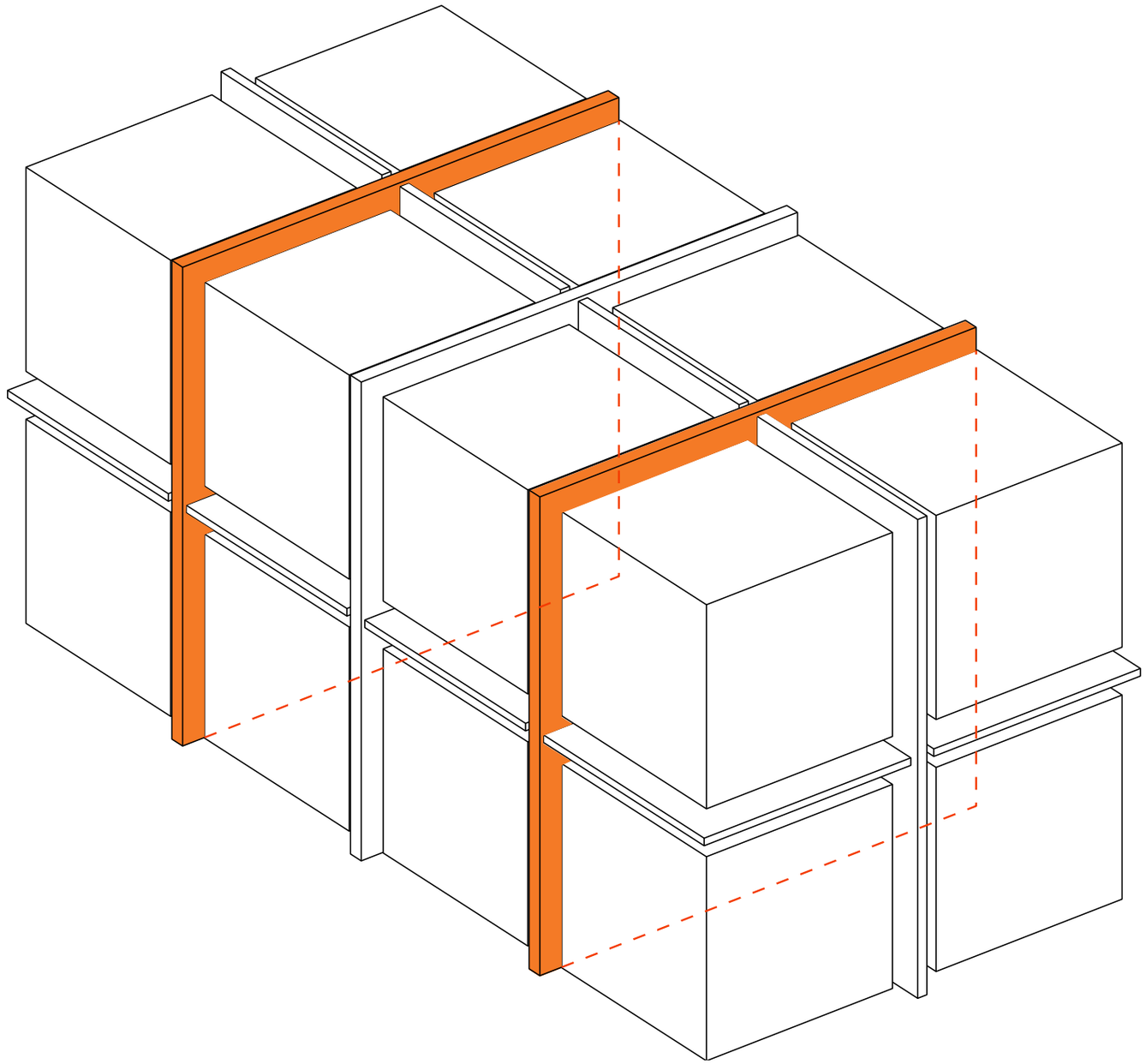
- Invert using divide-and-conquer

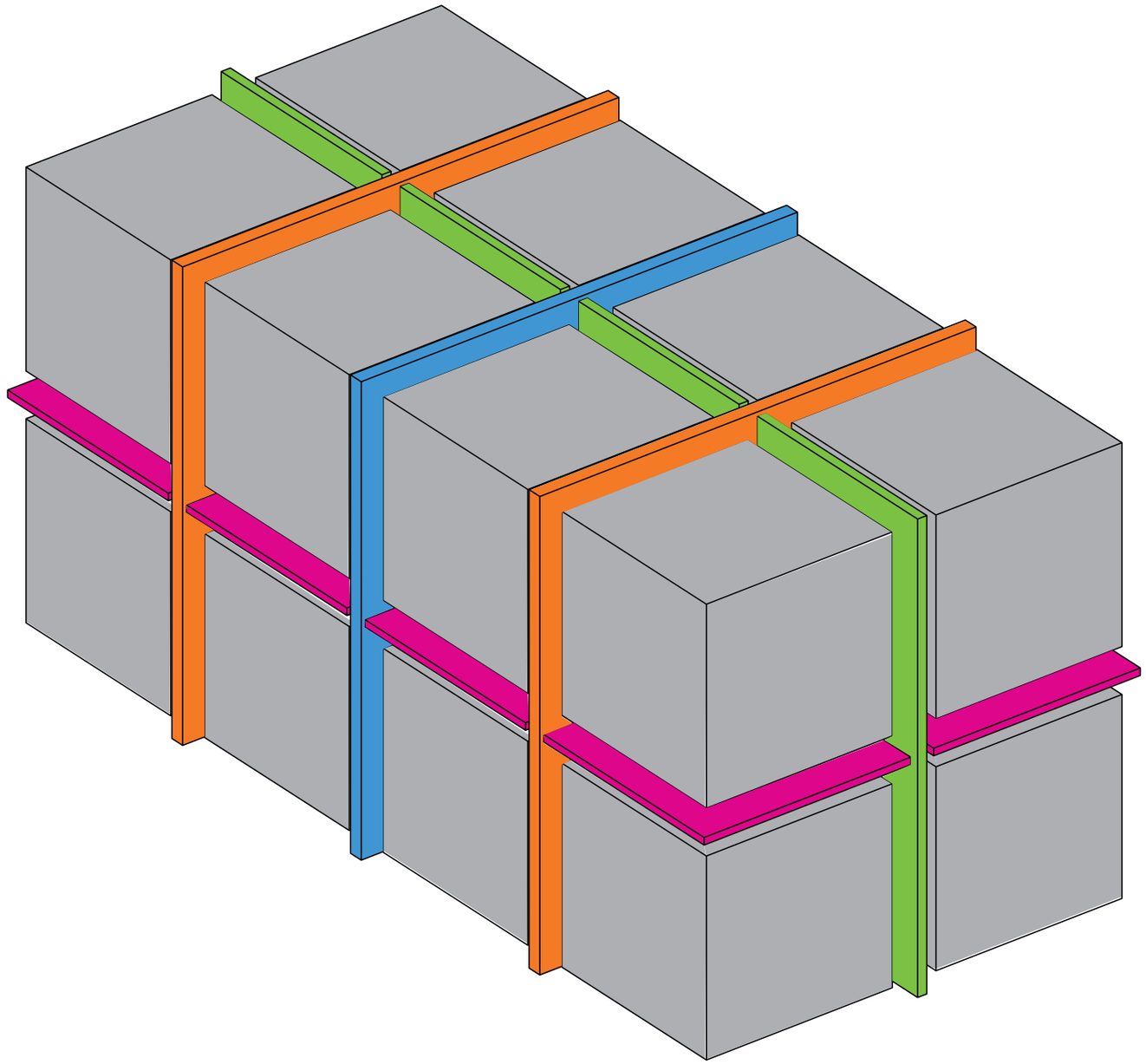
$$\begin{pmatrix} \mathbf{K}_{11} & & \mathbf{K}_{1c} \\ & \mathbf{K}_{22} & \mathbf{K}_{2c} \\ \mathbf{K}_{c1} & \mathbf{K}_{c2} & \mathbf{K}_{cc} \end{pmatrix}^{-1} =$$

$$\begin{pmatrix} \mathbf{I} & & -\mathbf{K}_{11}^{-1}\mathbf{K}_{1c} \\ & \mathbf{I} & -\mathbf{K}_{22}^{-1}\mathbf{K}_{2c} \\ & & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{K}_{11}^{-1} & & \\ & \mathbf{K}_{22}^{-1} & \\ & & \mathbf{C}^{-1} \end{pmatrix} \begin{pmatrix} \mathbf{I} & & \\ & \mathbf{I} & \\ -\mathbf{K}_{c1}\mathbf{K}_{11}^{-1} & -\mathbf{K}_{c2}\mathbf{K}_{22}^{-1} & \mathbf{I} \end{pmatrix}$$

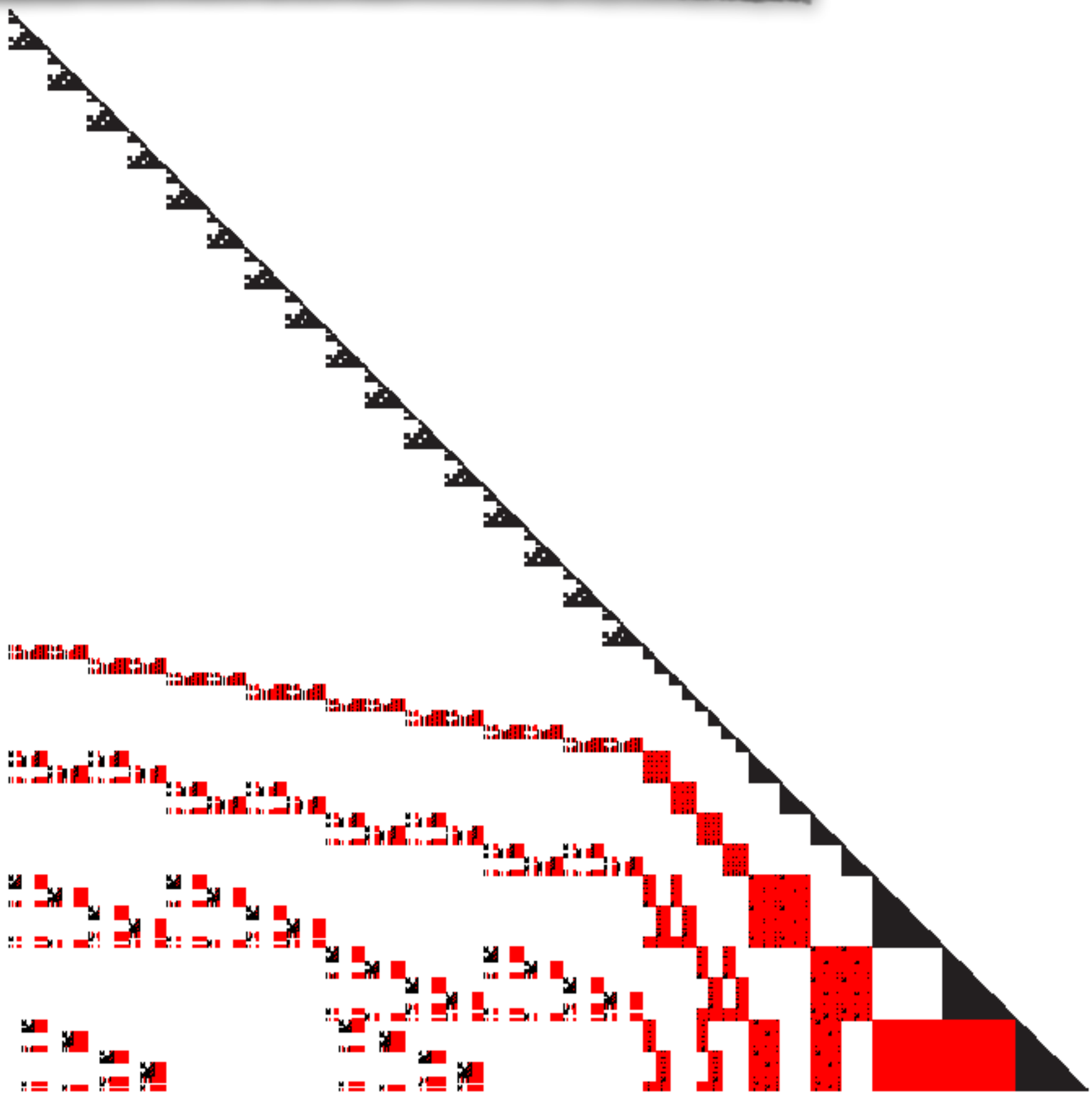




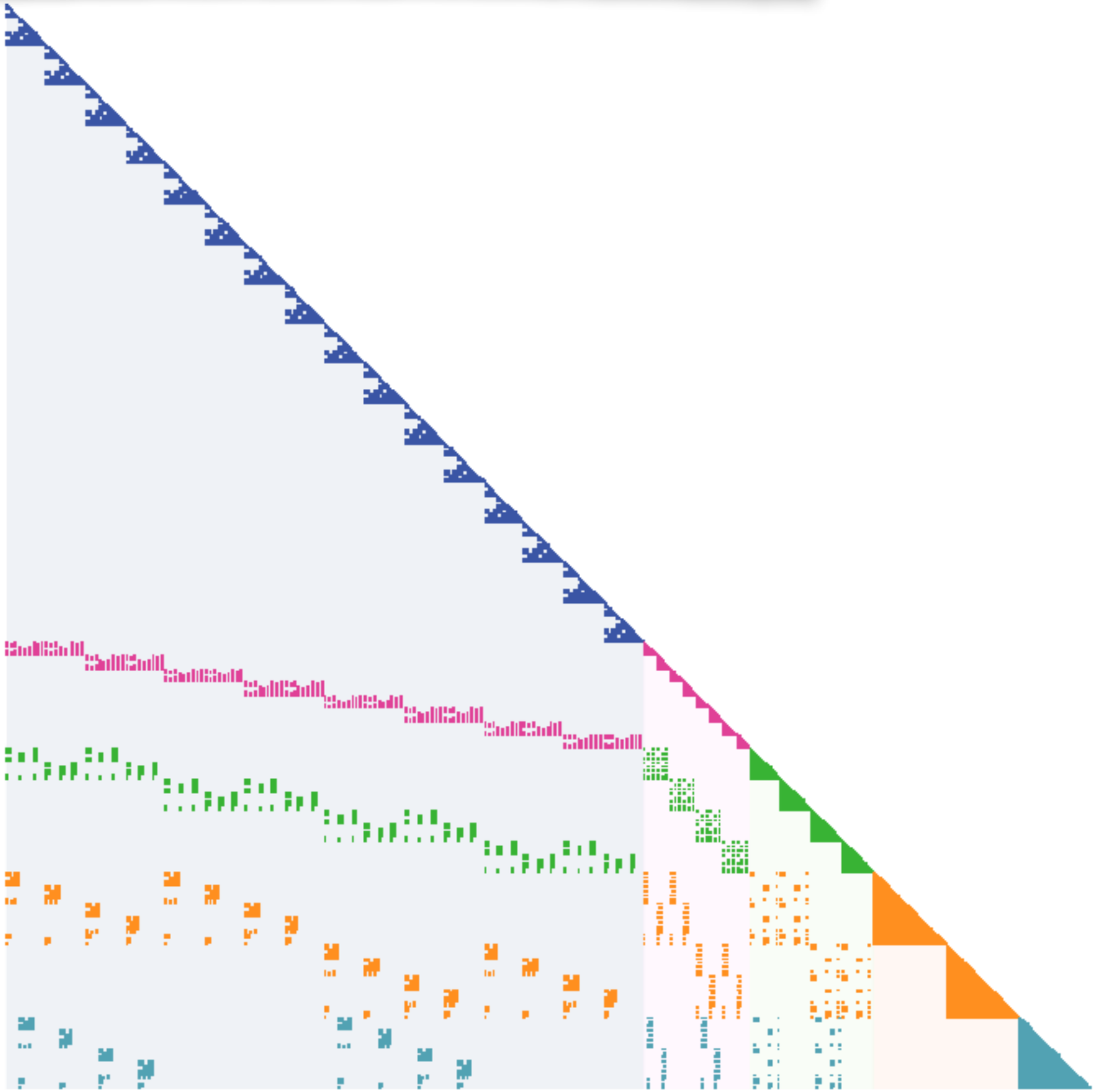




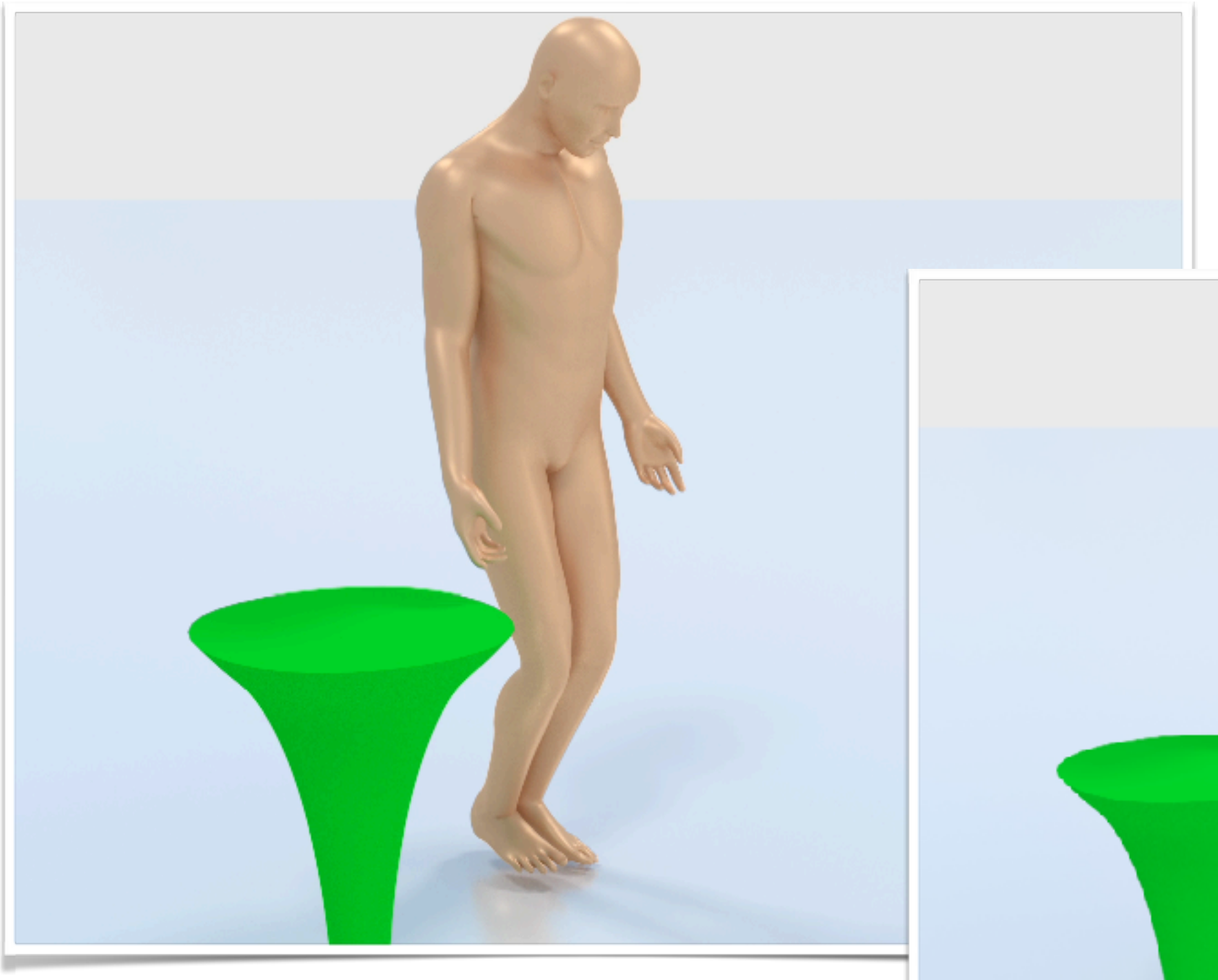
Balancing traits of direct & iterative solvers

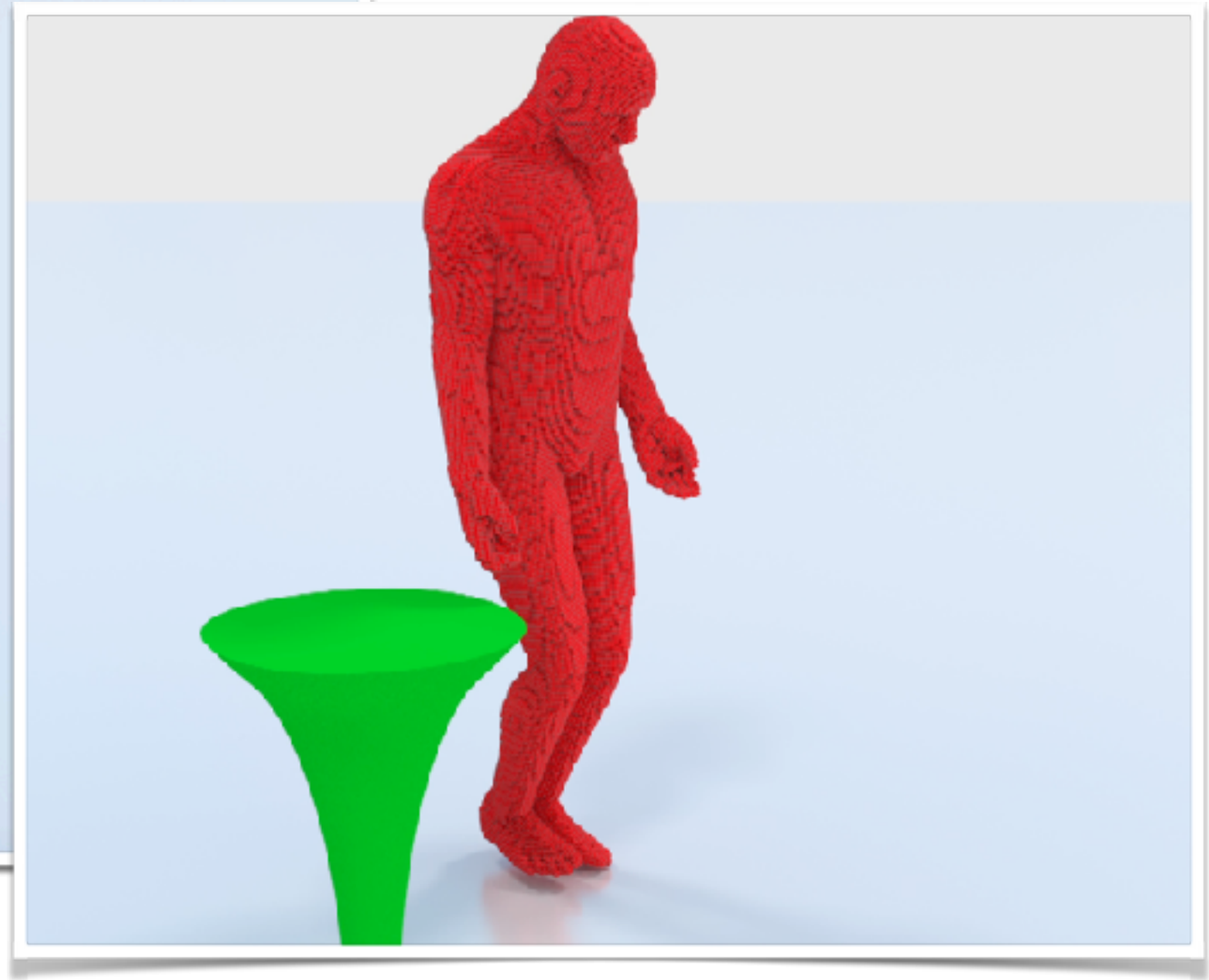


Balancing traits of direct & iterative solvers

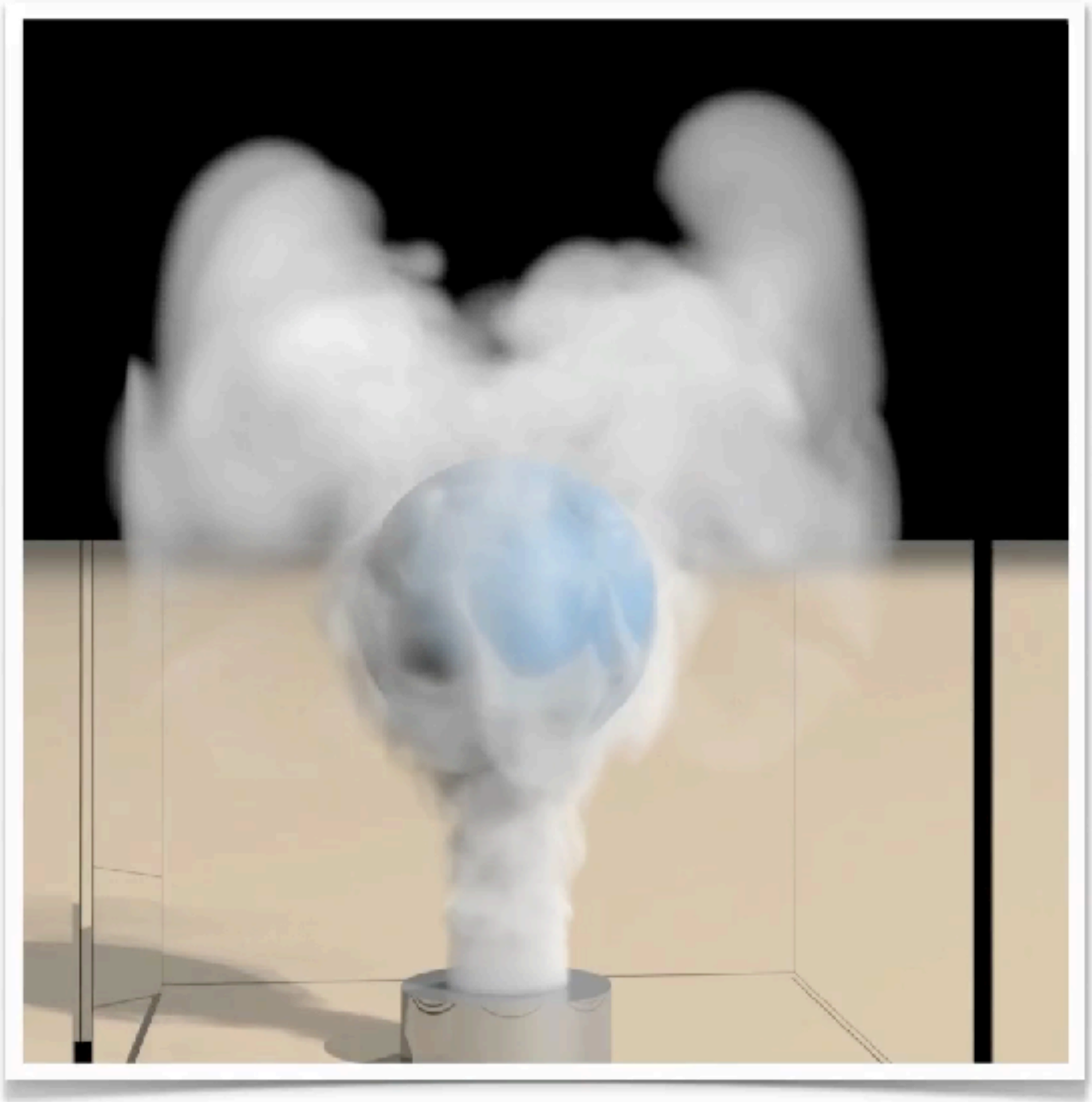


Balancing traits of direct & iterative solvers

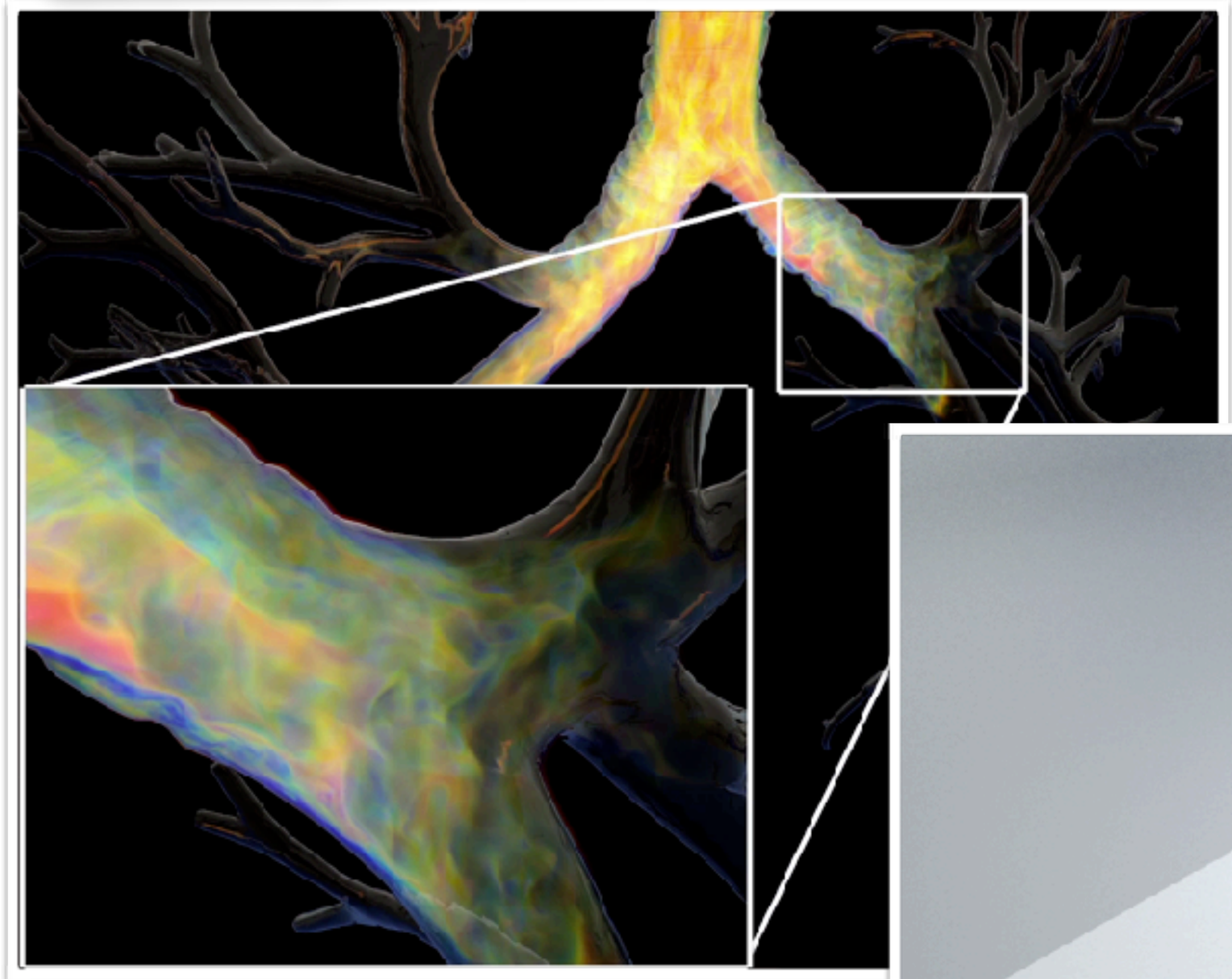


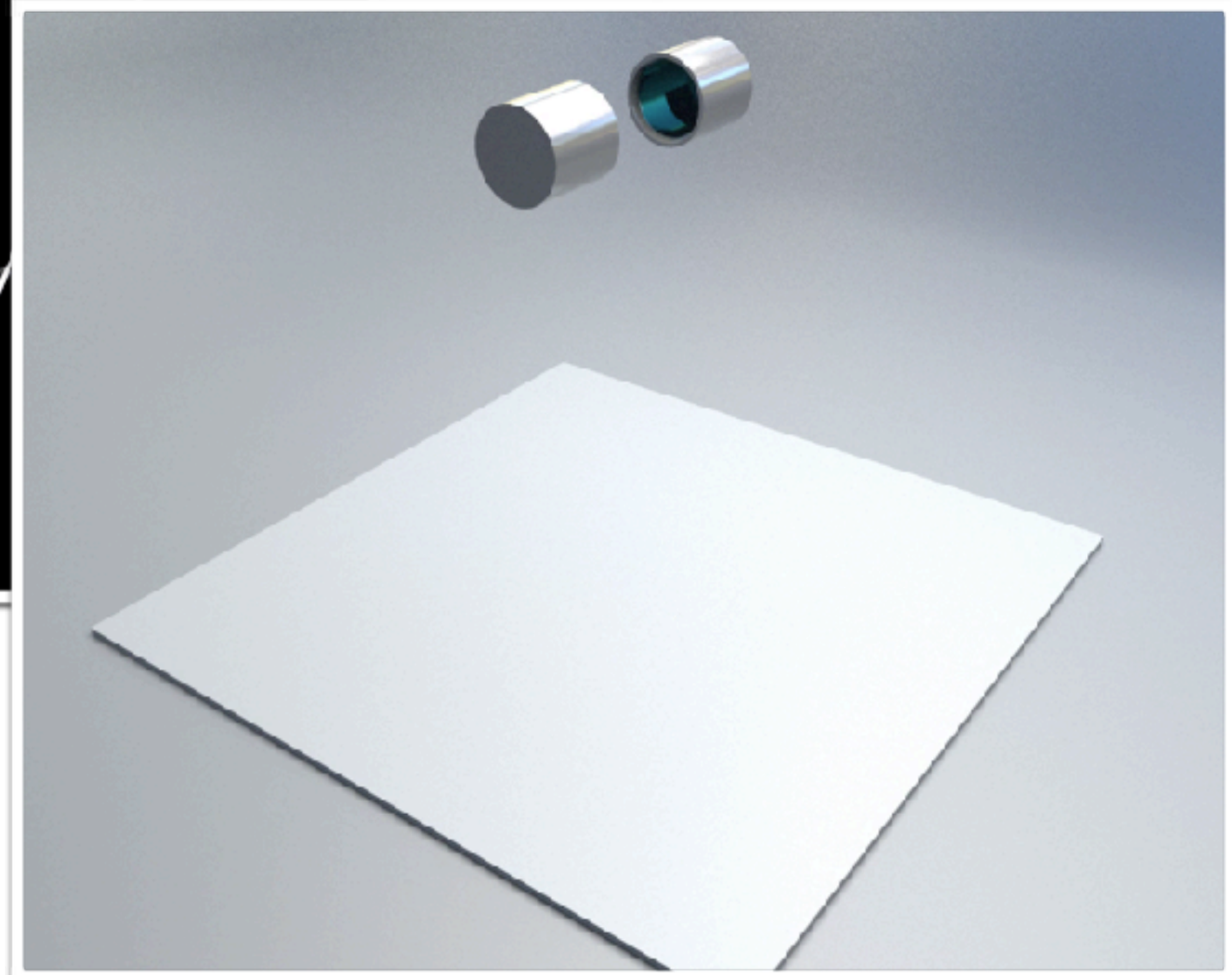


SPGrid - Virtual memory tricks sparse/adaptive grid data

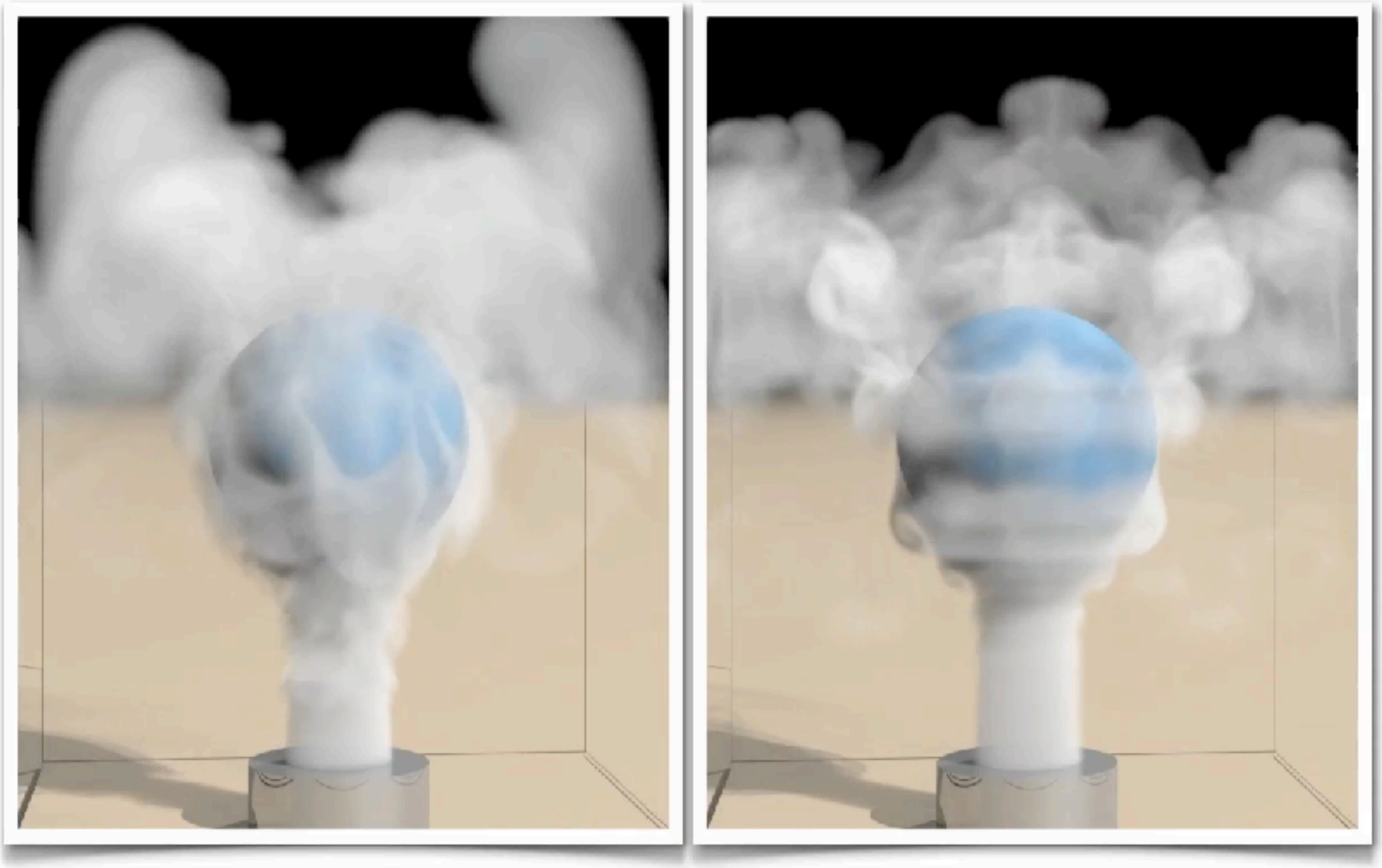


SPGrid - Virtual memory tricks sparse/adaptive grid data





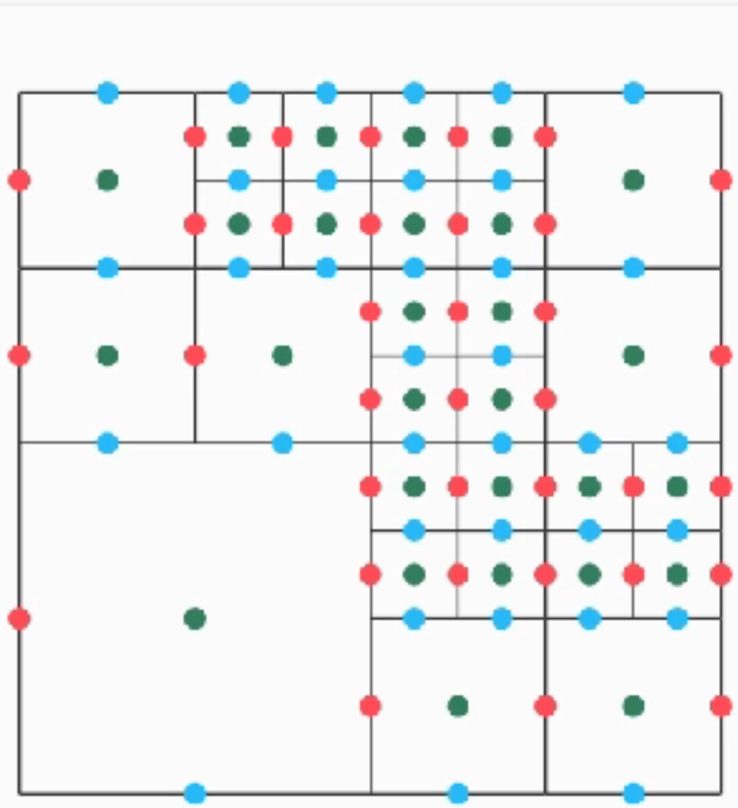
SPGrid - Virtual memory tricks sparse/adaptive grid data



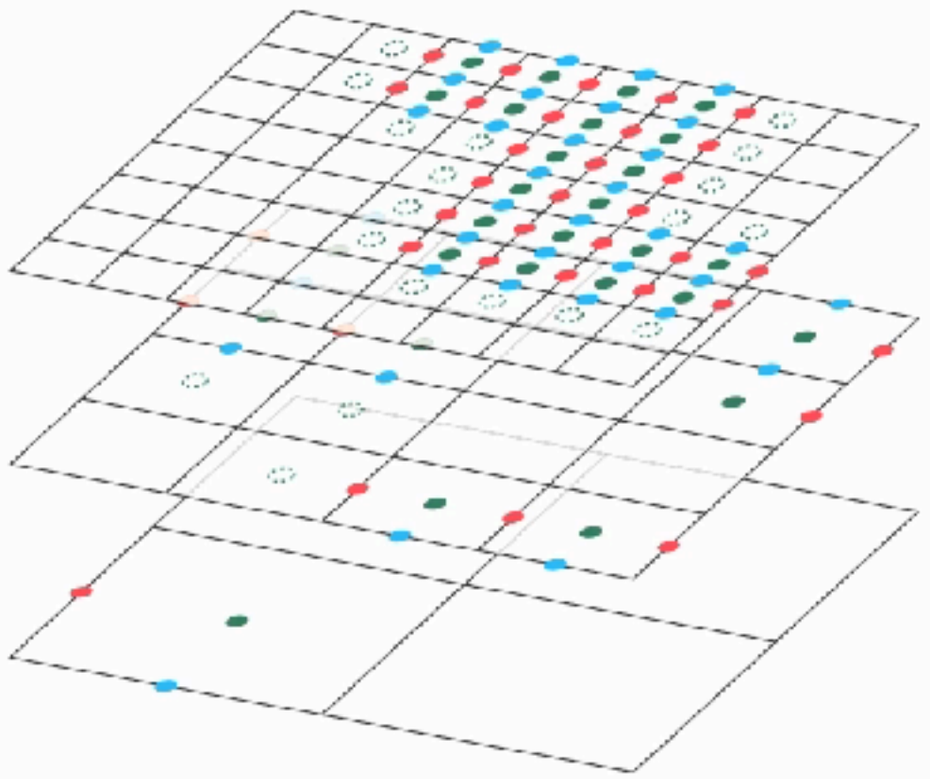
Adaptive
135M cells

Uniform
113M cells

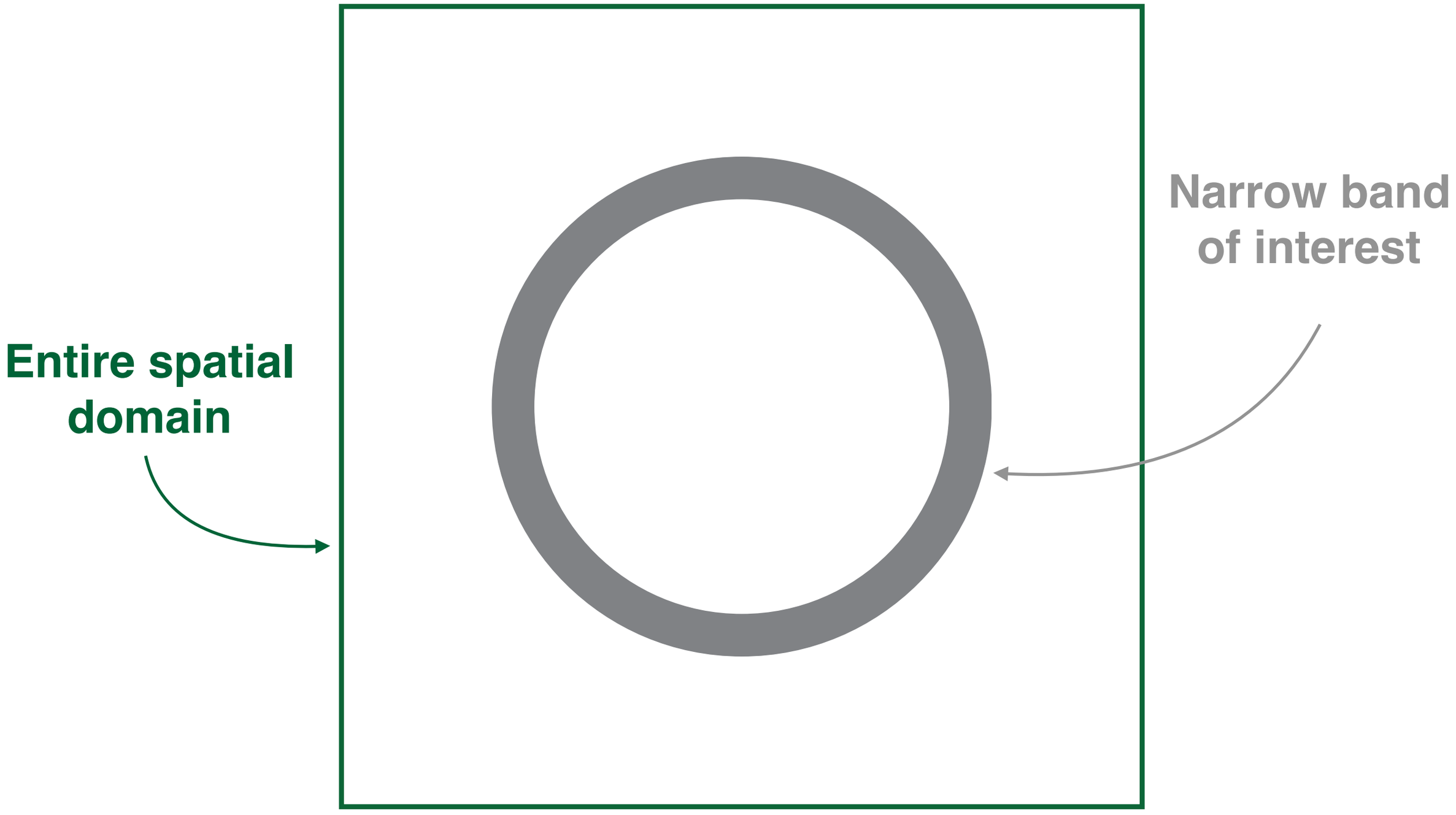
Virtual memory tricks for adaptive simulation



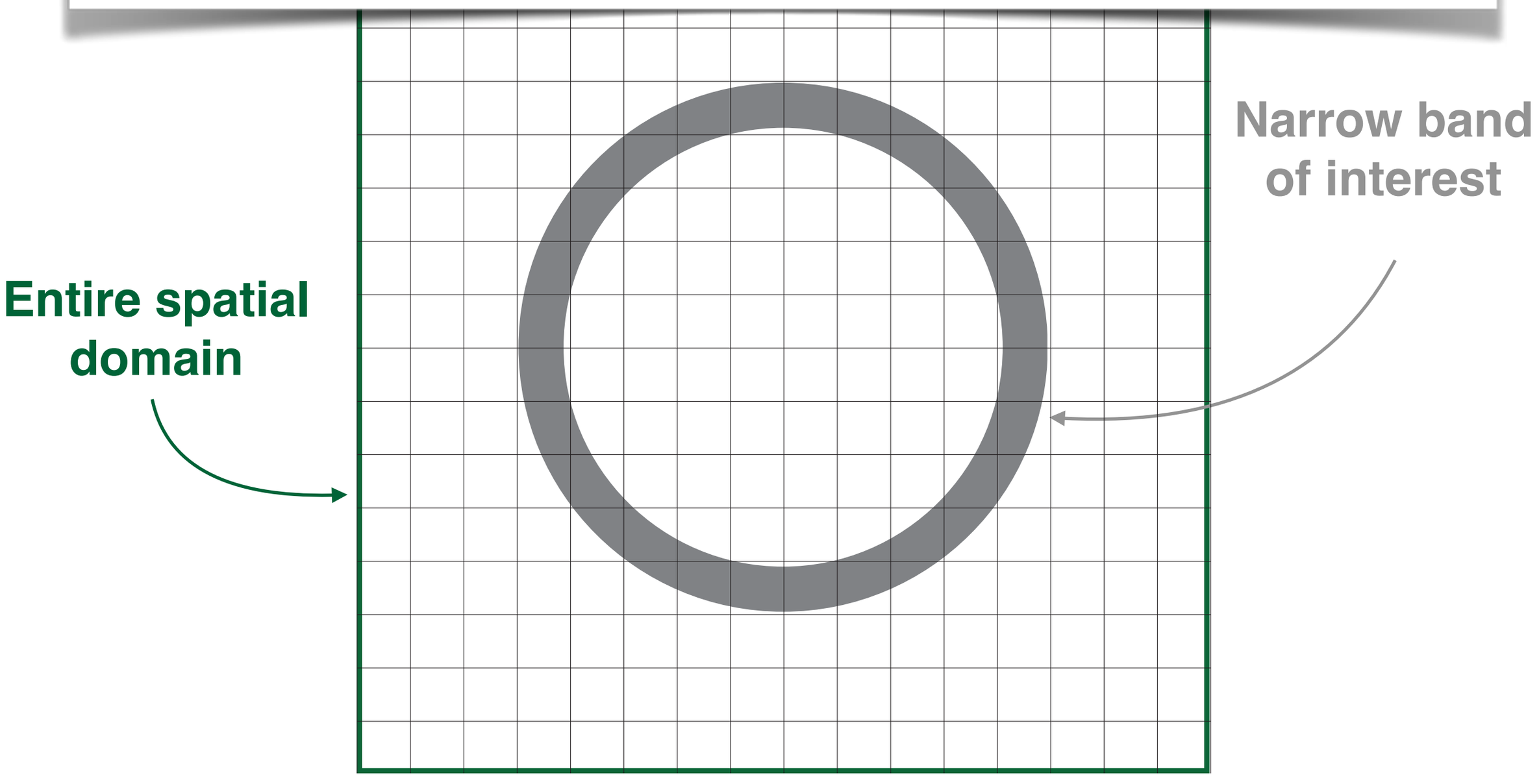
Geometric Octree

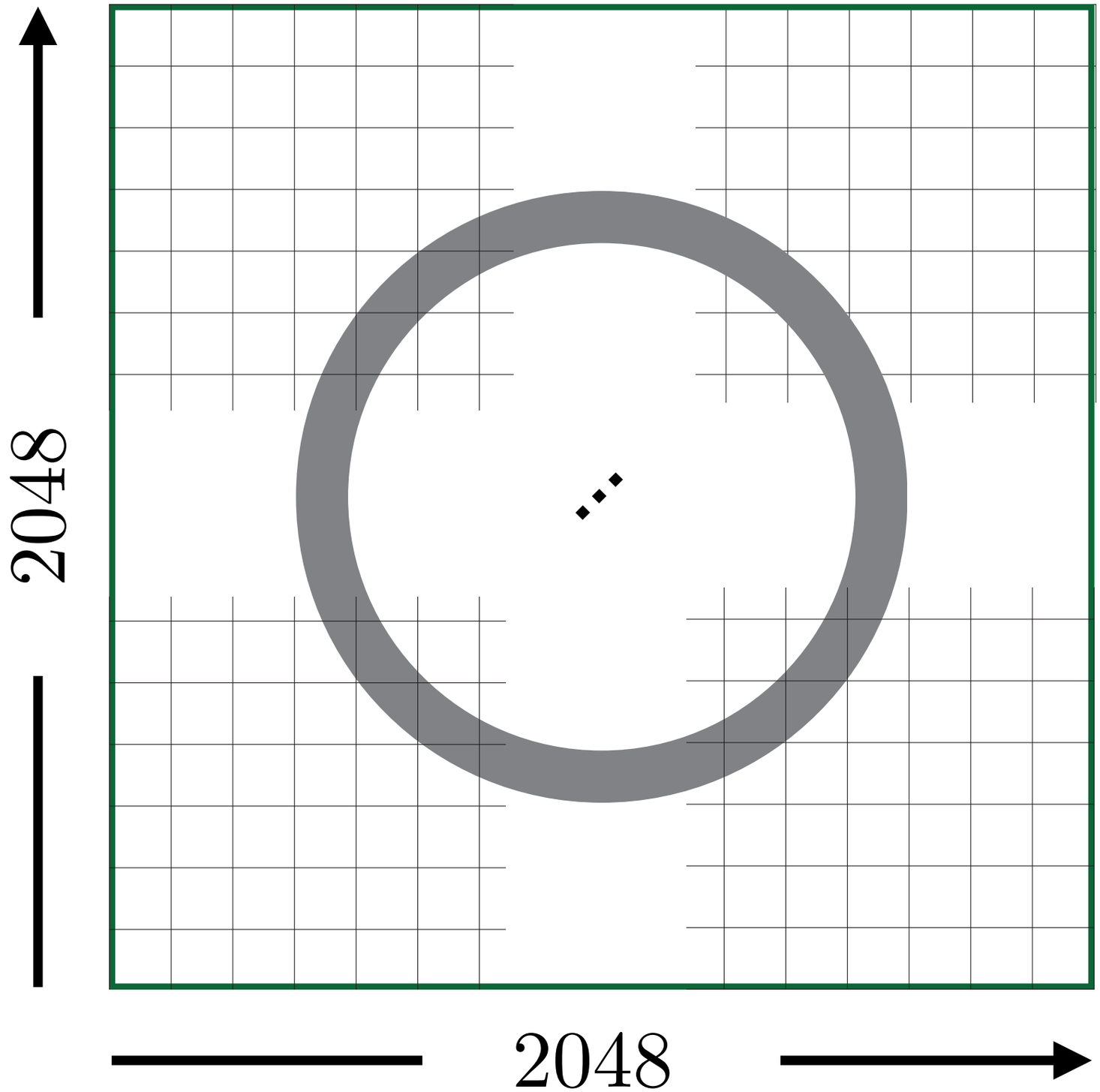


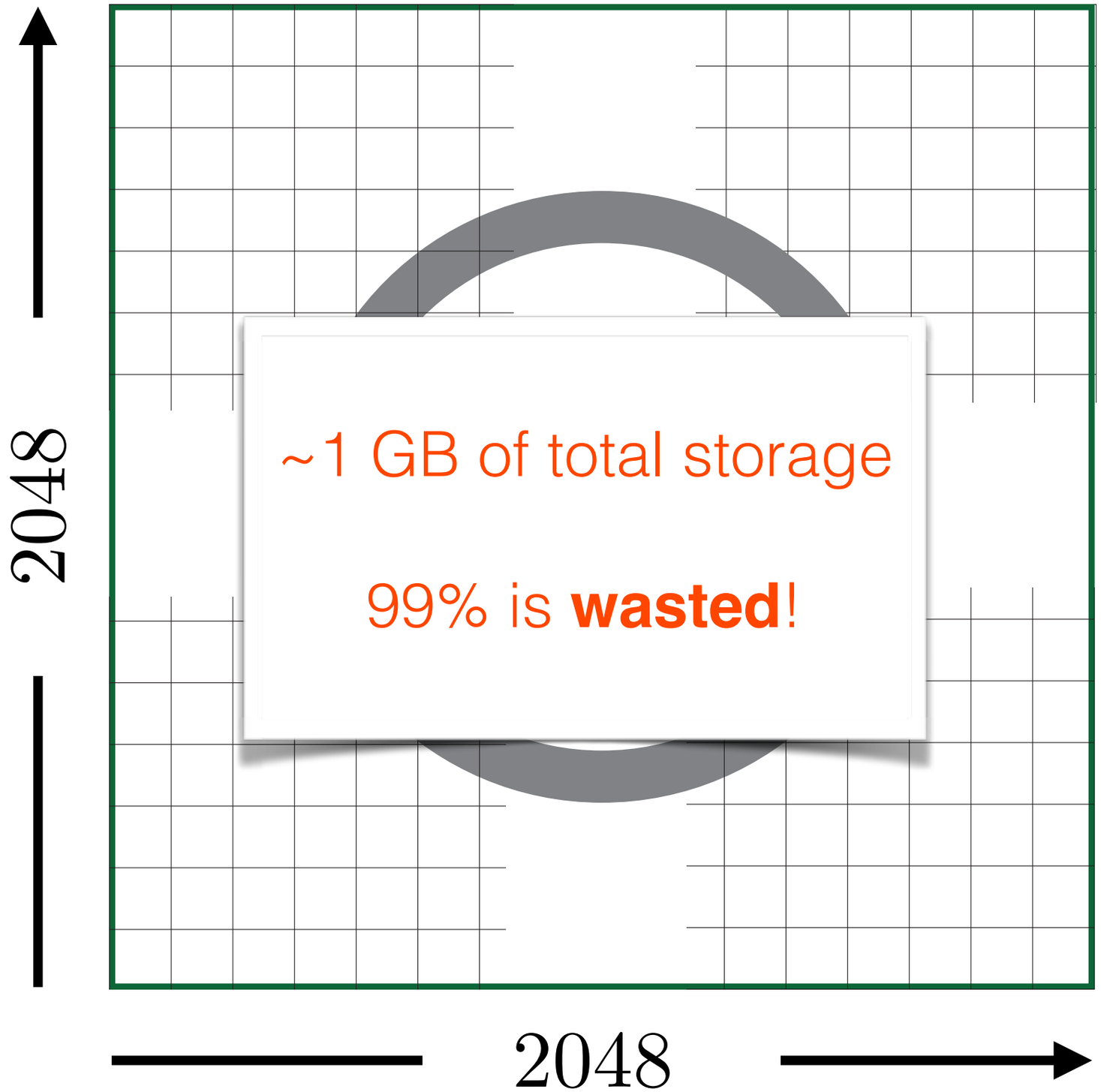
SPGrid Hierarchy


Entire spatial domain
Narrow band of interest

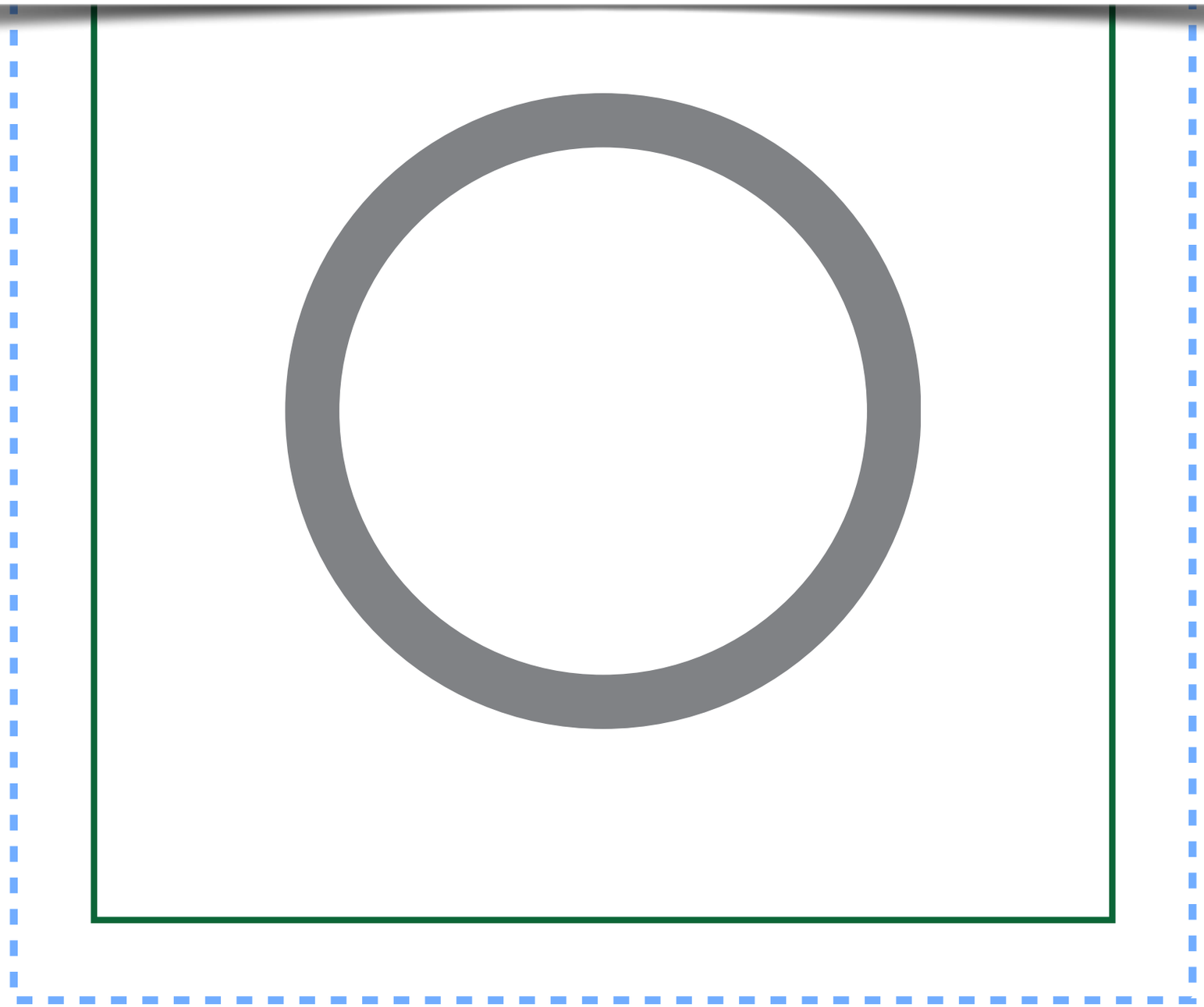
```
float* grid = new float[nx * ny * nz];
```



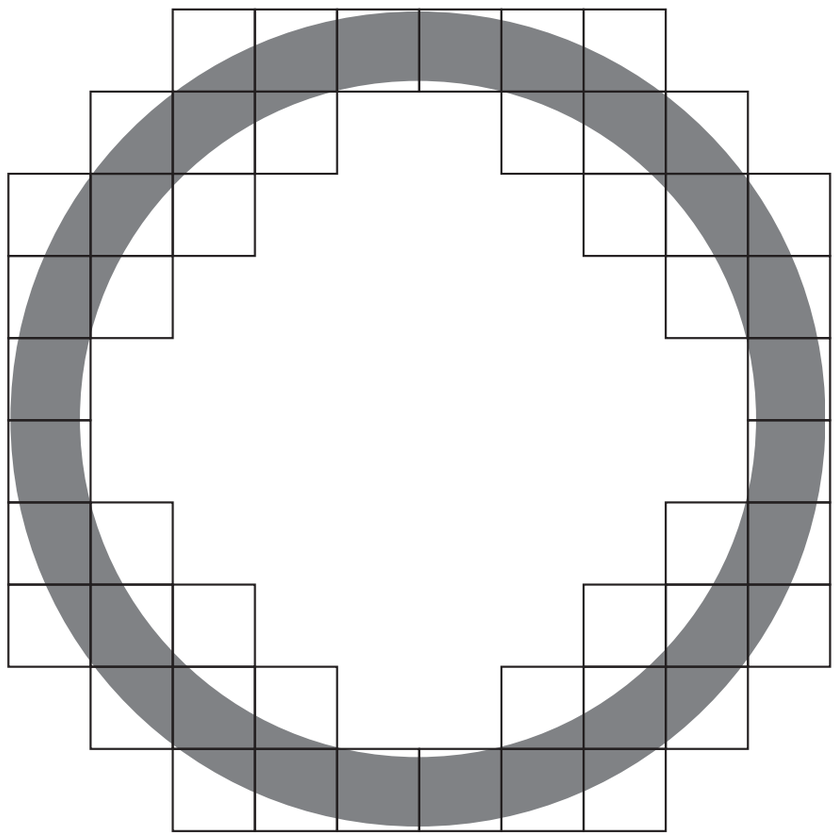

2048
2048


2048
2048
~1 GB of total storage
99% is **wasted**!

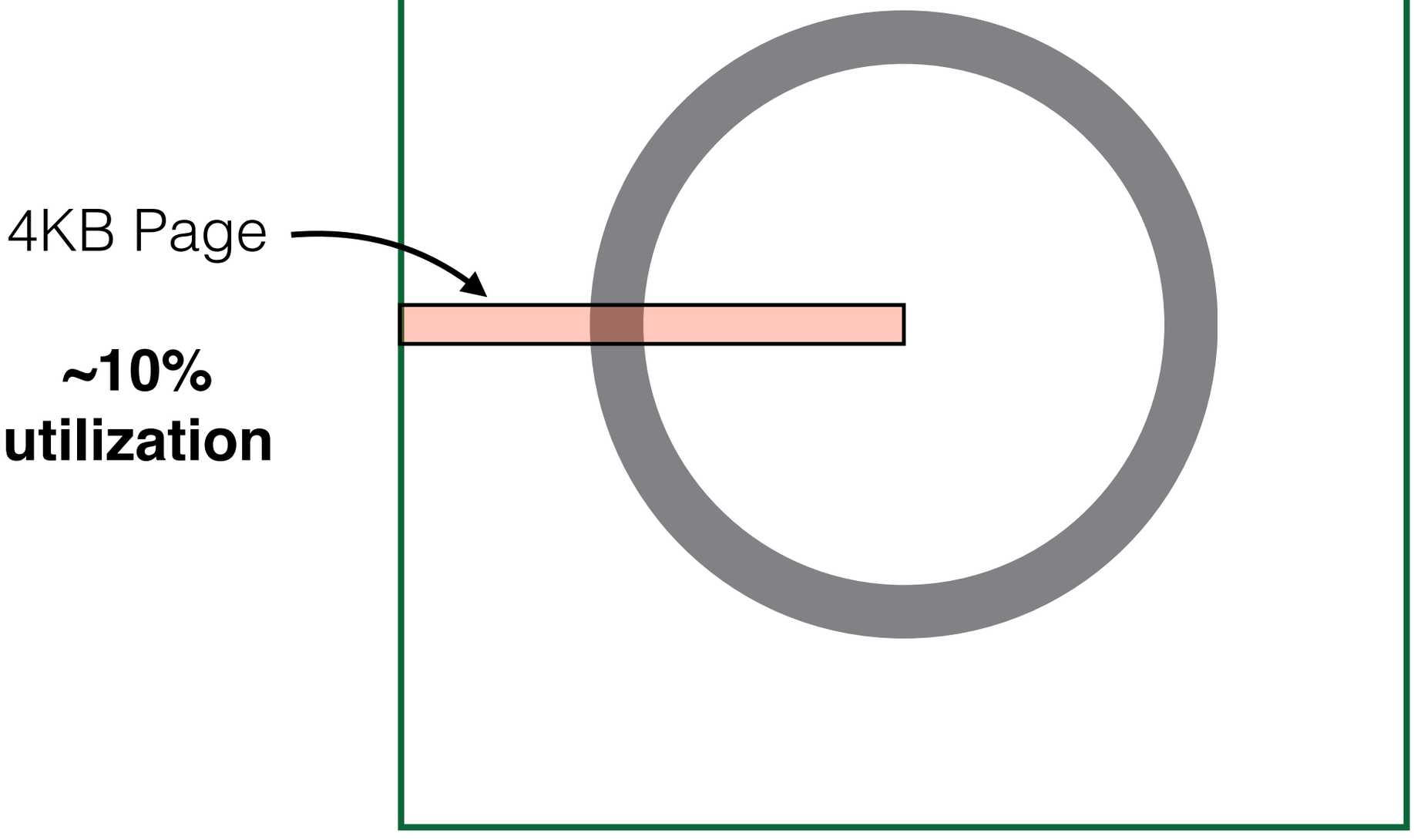
```
float* grid = mmap(nx * ny * nz, …);
```

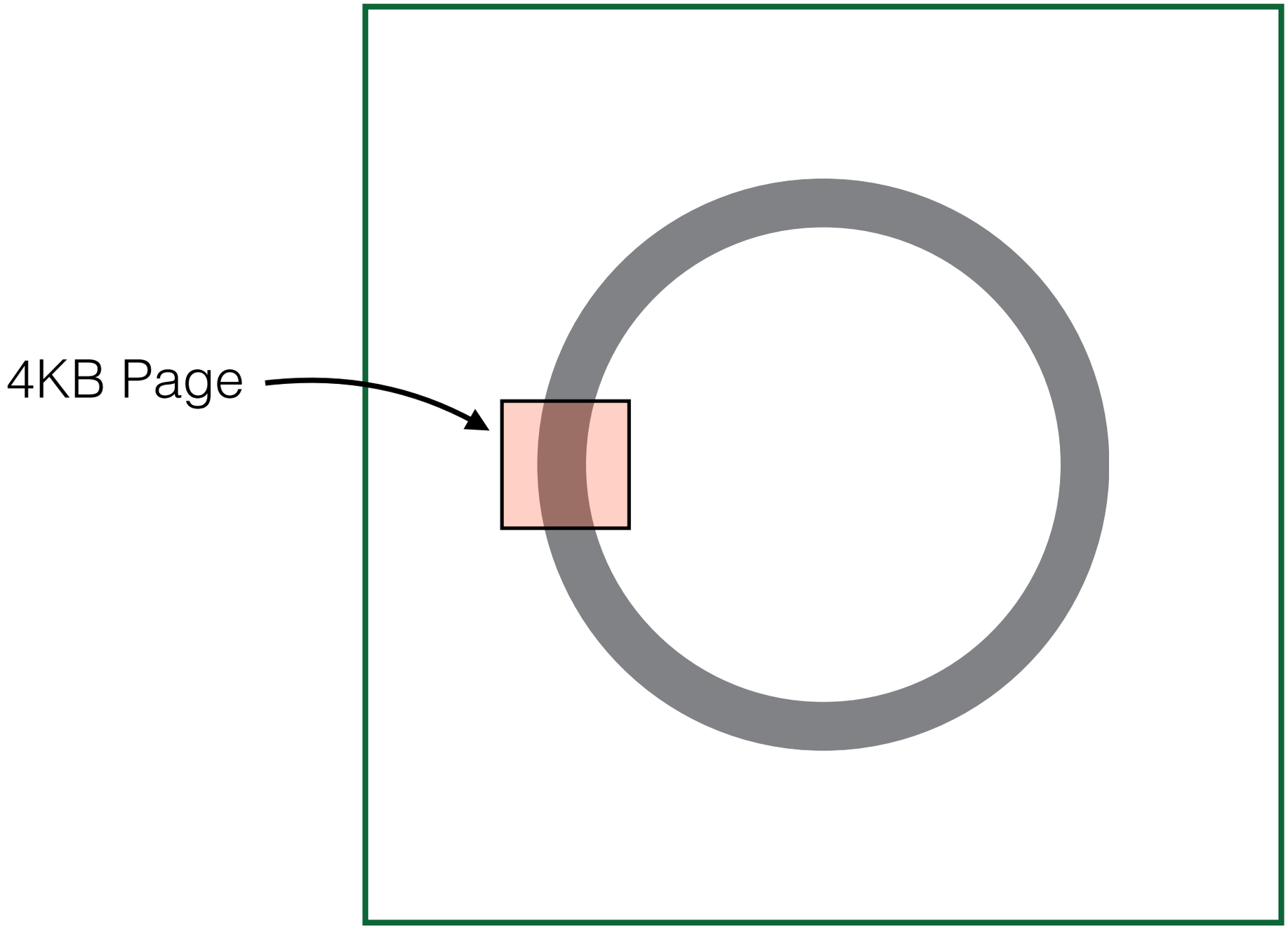


```
float* grid = mmap(nx * ny * nz, …);
```

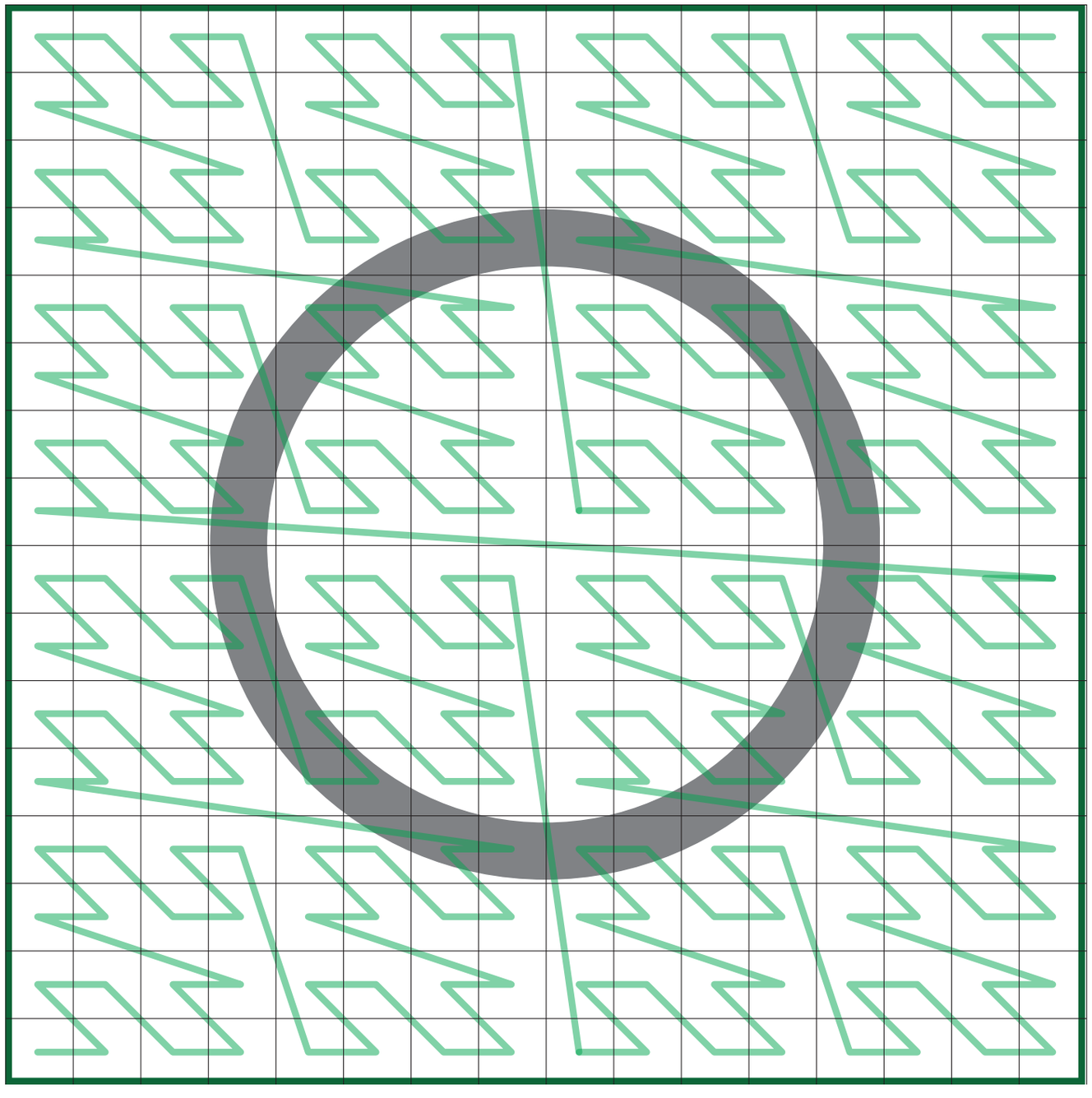


Lexicographical

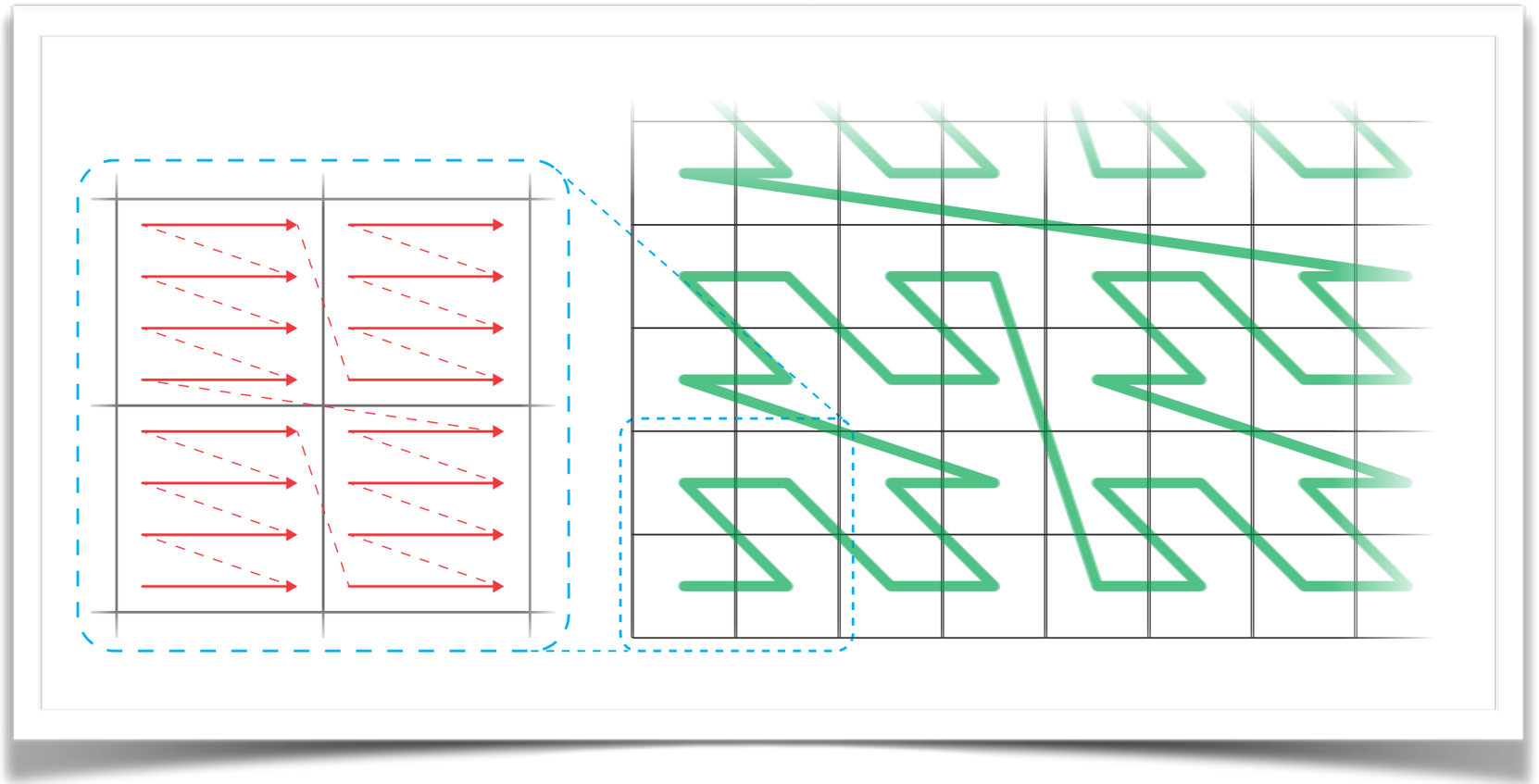



4KB Page

Morton Ordering

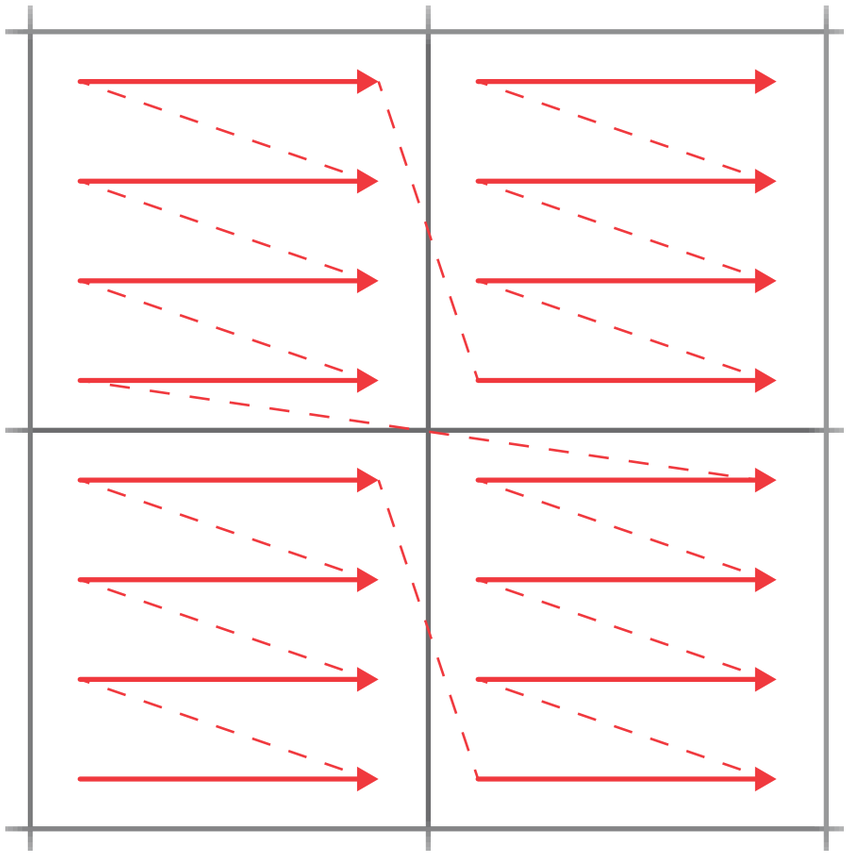


Blocked Grid



Lexicographical traversal **within** block

Z-curve traversal **across** blocks



0	1	2	3	16	17	18	19	64	65
4	5	6	7	20	21	22	23	68	69
8	9	10	11	24	25	26	27	72	73
12	13	14	15	28	29	30	31	76	77
32	33	34	35	48	49	50	51	96	97
36	37	38	39	52	53	54	55	100	101
40	41	42	43	56	57	58	59	104	105
44	45	46	47	60	61	62	63	108	109
128	129	130	131	144	145	146	147	192	193
132	133	134	135	148	149	150	151	196	197

Array Y-coordinate (binary)	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
1001	132	133	134	135	148	149	150	151	196	197
1000	128	129	130	131	144	145	146	147	192	193
0111	44	45	46	47	60	61	62	63	108	109
0110	40	41	42	43	56	57	58	59	104	105
0101	36	37	38	39	52	53	54	55	100	101
0100	32	33	34	35	48	49	50	51	96	97
0011	12	13	14	15	28	29	30	31	76	77
0010	8	9	10	11	24	25	26	27	72	73
0001	4	5	6	7	20	21	22	23	68	69
0000	0	1	2	3	16	17	18	19	64	65

Array X-coordinate (binary)

1001	132	133	134	135	148	149	150	151	196	197
1000	128	129	130	131	144	145	146	147	192	193
0111	44	45	46	47	60	61	62	63	108	109
0110	40	41	42	43	56	57	58	59	104	105
0101	36	37	38	39	52	53	54	55	100	101
0100	32	33	34	35	48	49	50	51	96	97
0011	12	13	14	15	28	29	30	31	76	77
0010	8	9	10	11	24	25	26	27	72	73
0001	4	5	6	7	20	21	22	23	68	69
0000	0	1	2	3	16	17	18	19	64	65
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

0 1 11

0 0 10

Array Y-coordinate (binary)

Array X-coordinate (binary)

~16 cycles on Haswell processor!

= 00101110(2)

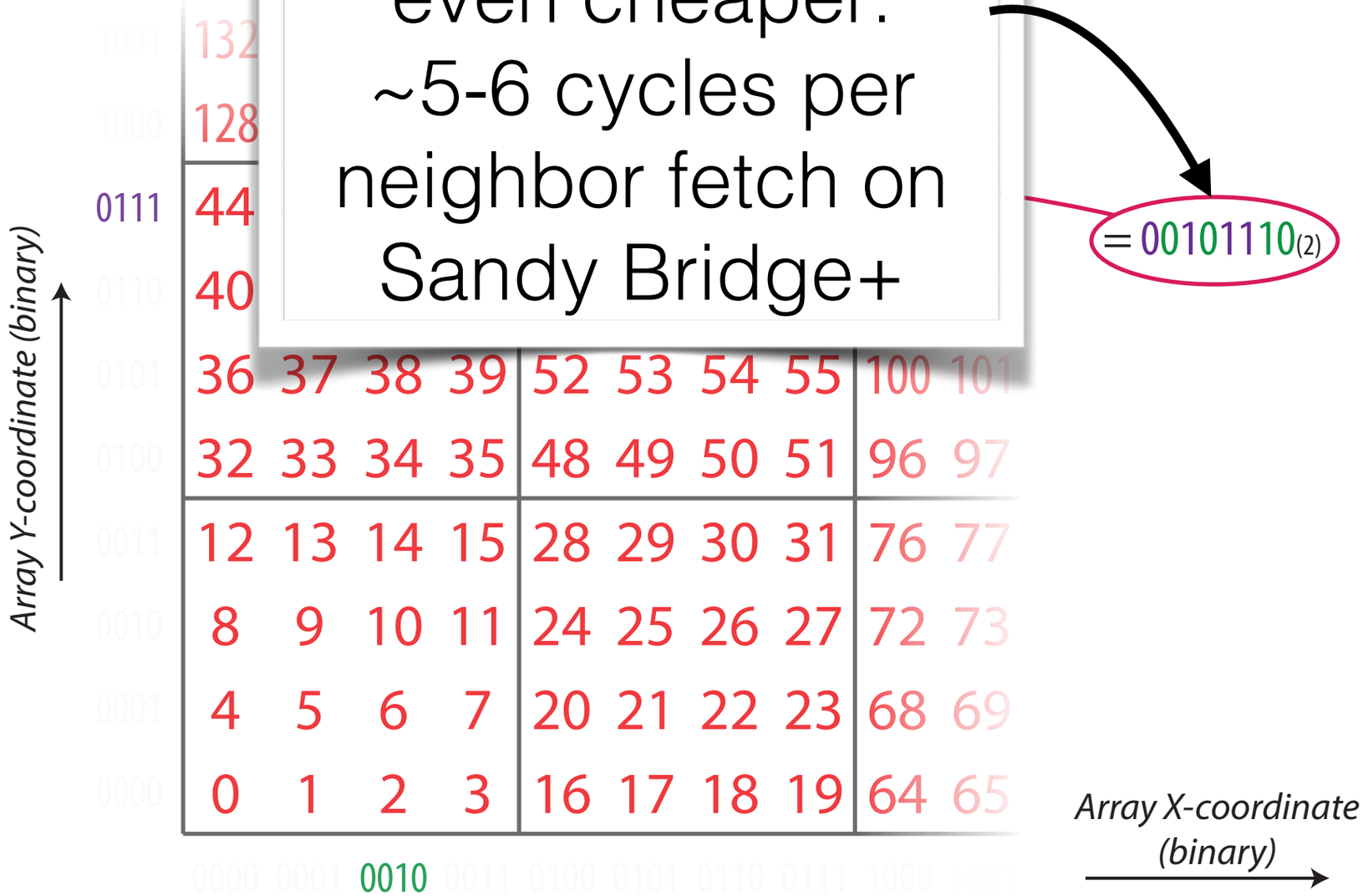
1001	132								104	105
1000	128									
0111	44									
0110	40	41	42	43	56	57	58	59		
0101	36	37	38	39	52	53	54	55	100	101
0100	32	33	34	35	48	49	50	51	96	97
0011	12	13	14	15	28	29	30	31	76	77
0010	8	9	10	11	24	25	26	27	72	73
0001	4	5	6	7	20	21	22	23	68	69
0000	0	1	2	3	16	17	18	19	64	65

0000 0001 0010 0011 0100 0101 0110 0111 1000 1001

Array Y-coordinate (binary)

Array X-coordinate (binary)

Stencil operations even cheaper: ~5-6 cycles per neighbor fetch on Sandy Bridge+



Remaining details …

- Solver : Multigrid-Preconditioned CG
- MG adapted to grid hierarchy, too
- Can do dynamically changing adaptation pattern
- Originally designed for CPU (+Xeon Phi trivially), but GPU analogues implemented to follow-up work

Harnessing platform heterogeneity

- **Heterogeneous platforms**
  - Multi-GPU equipped SMP servers …
  - Small "fast" memory + Large "slow" memory hybrids (e.g.Knights Landing) …
  - Or any reasonably deep memory hierarchy …
  - Best ratio of $$ to Raw Computational Capacity

Harnessing platform heterogeneity

- **Heterogeneous platforms**
  - Multi-GPU equipped SMP servers …
  - Small "fast" memory + Large "slow" memory hybrids (e.g.Knights Landing) …
  - Or any reasonably deep memory hierarchy …
- Best ratio of $$ to Raw Computational Capacity

(110+ TFlops, 768GB RAM, 4.5+GB/s bandwidth : $20k)

Harnessing platform heterogeneity

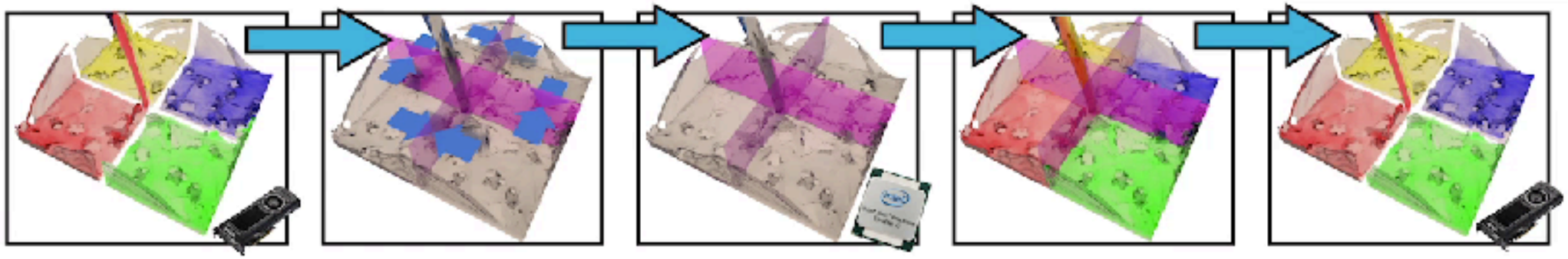
- **Challenges**
  - GFlops/Gload ratio - approaching 100:1 today
    (closer to 12-15:1 in "good old pre-GPU/SIMD days")
  - Access to non-local memory at least one order of magnitude slower
    (PCIe/Infiniband vs DDD4, or KNL fast/slow memory)
  - Even worse scenario :
    Workload ***too slow*** for CPU, ***too large*** *for GPU*
  - Offload overhead often a non-starter for traditional distributed parallelization

Harnessing platform heterogeneity

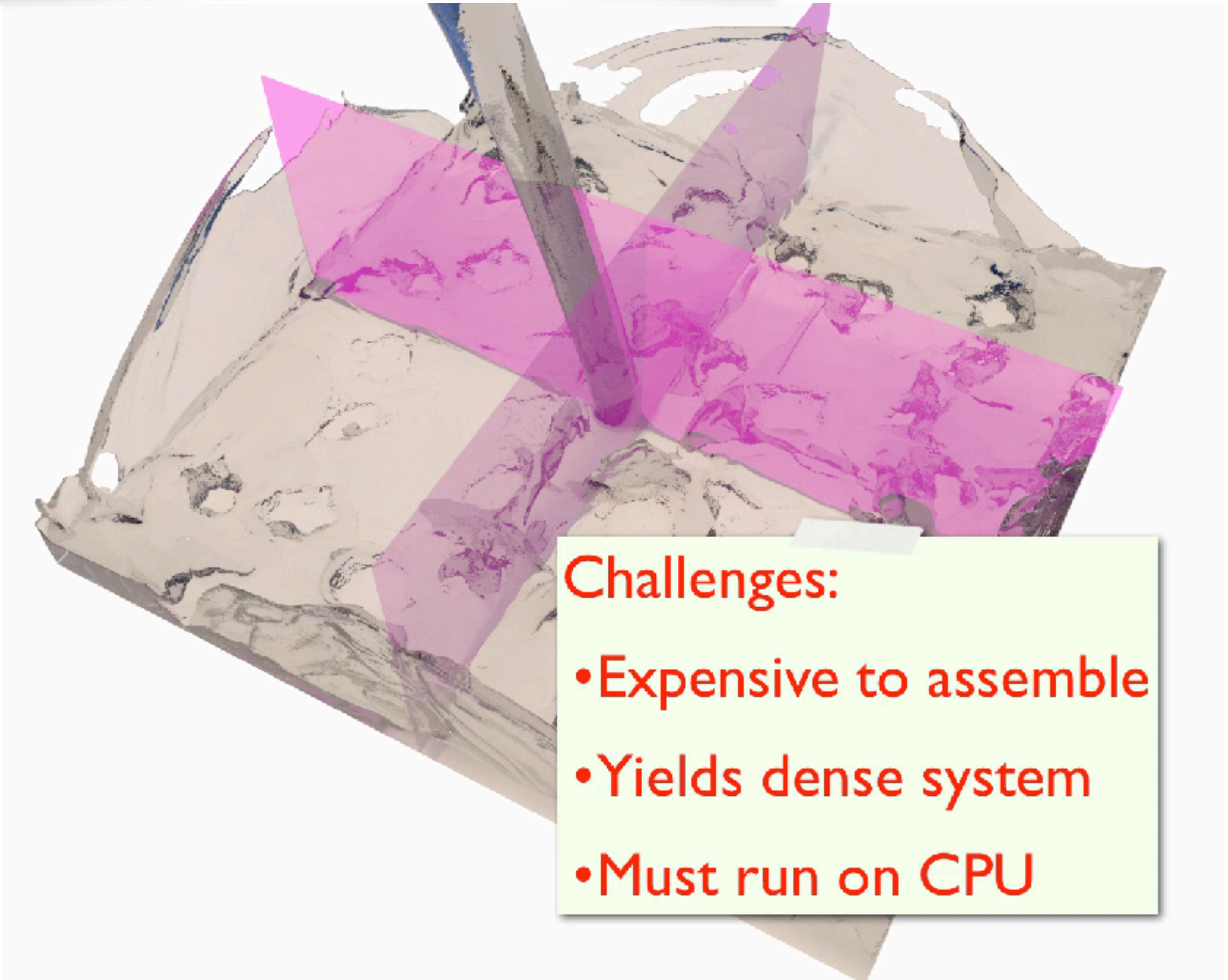
- ***A crisis or an opportunity?***
  - Dream scenario :

    Making an algorithm work on a heterogeneous platform as fast as a homogeneous one with the ***aggregate*** *specs*
  - Certainly not possible *in general* (via automated means)
  - … but doable for many problems in computational dynamics!

Harnessing platform heterogeneity



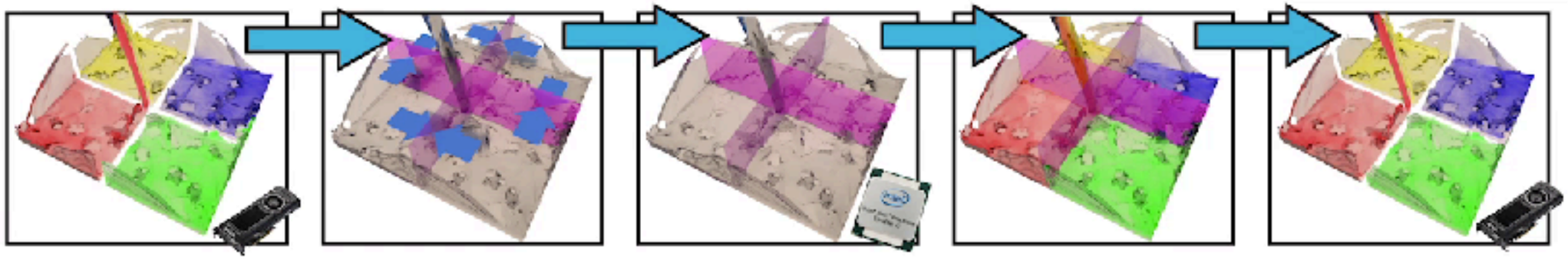
Harnessing platform heterogeneity



Challenges:

- Expensive to assemble
- Yields dense system
- Must run on CPU

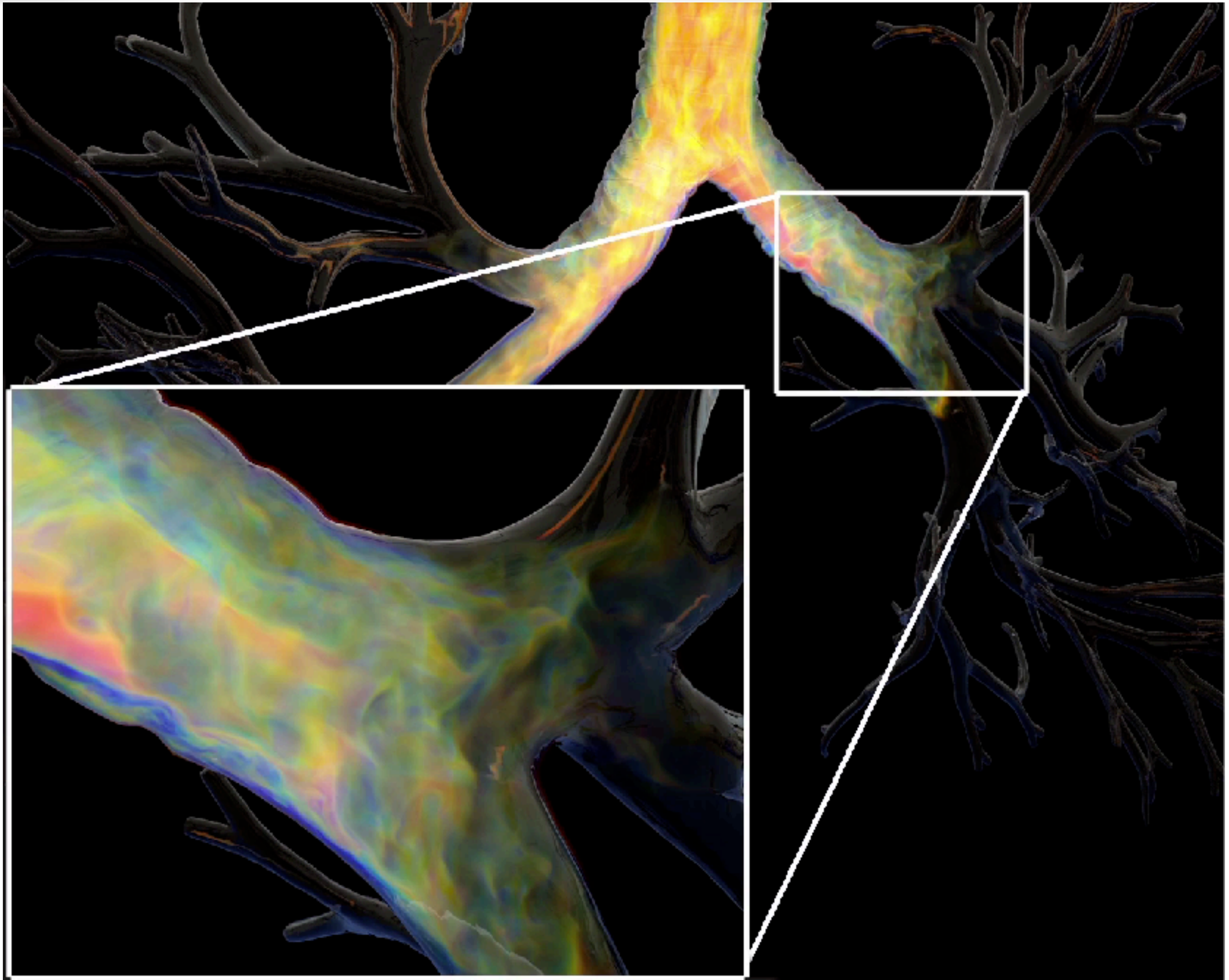
Harnessing platform heterogeneity



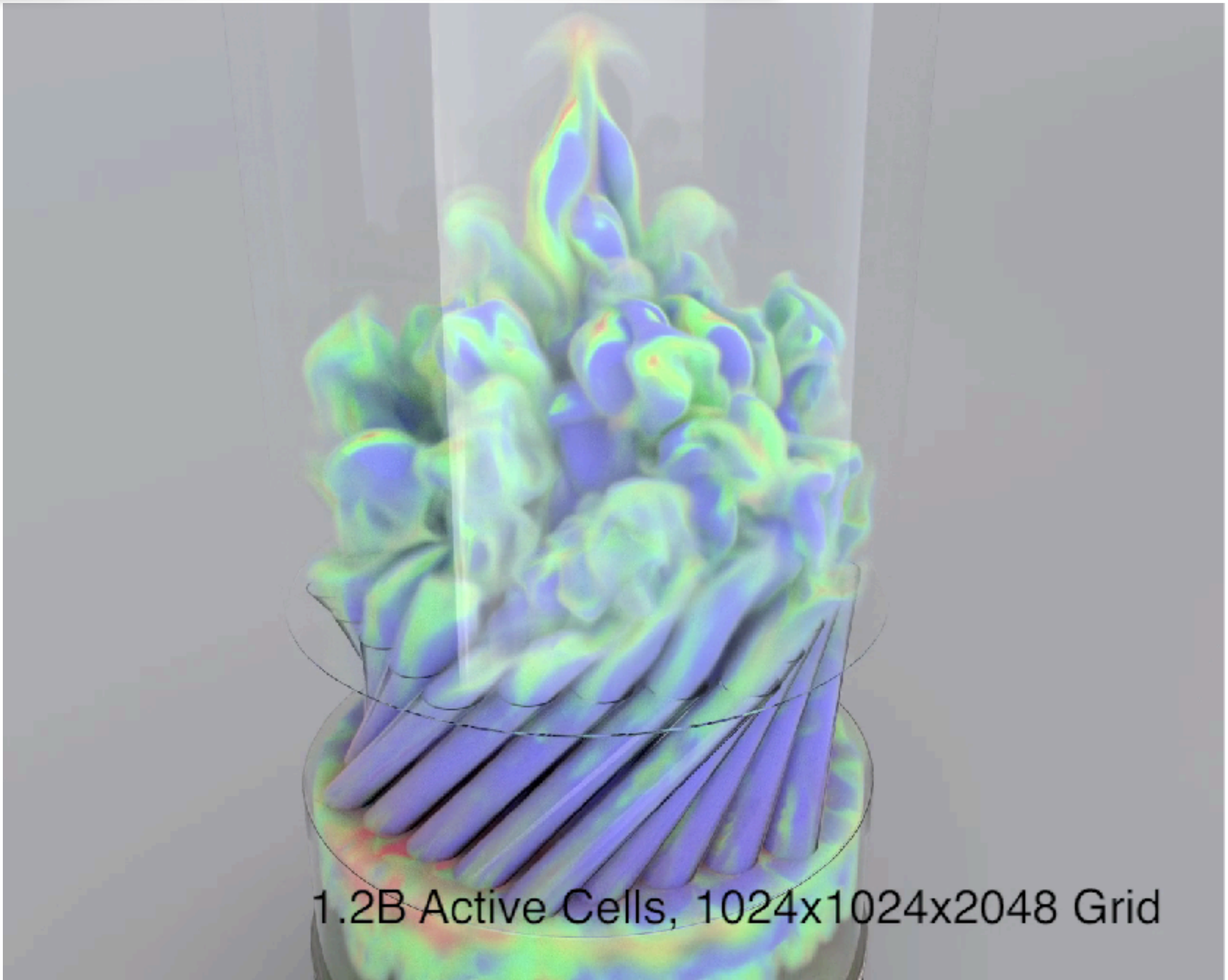
Harnessing platform heterogeneity

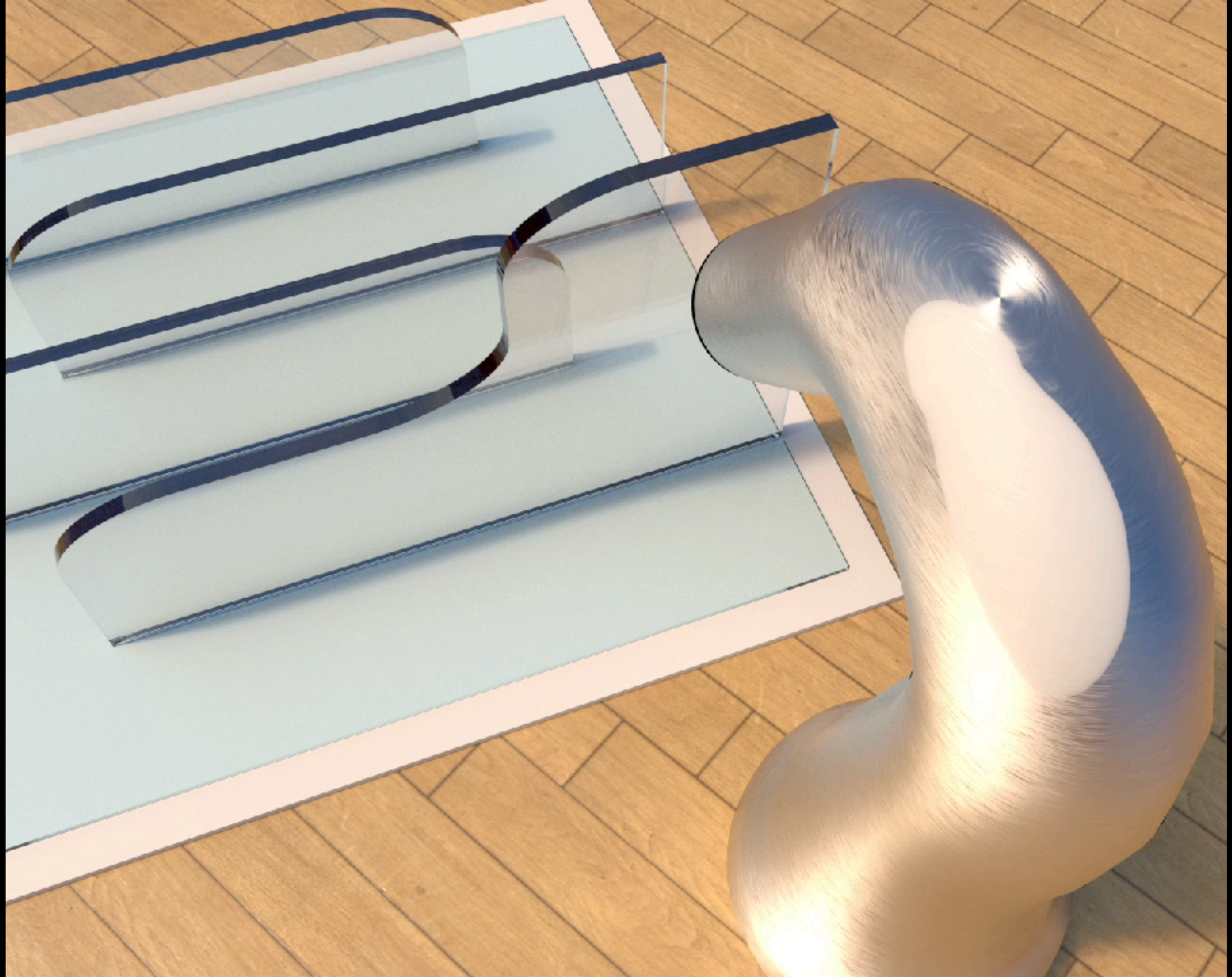
- **Performance restored by:**
  - Making any offloaded tasks are large enough to absorb communication cost (and get something in return)
  - Tweaking the math (adaptivity) to reduce complexity
  - Use an "approximate" divide and conquer design

Harnessing platform heterogeneity



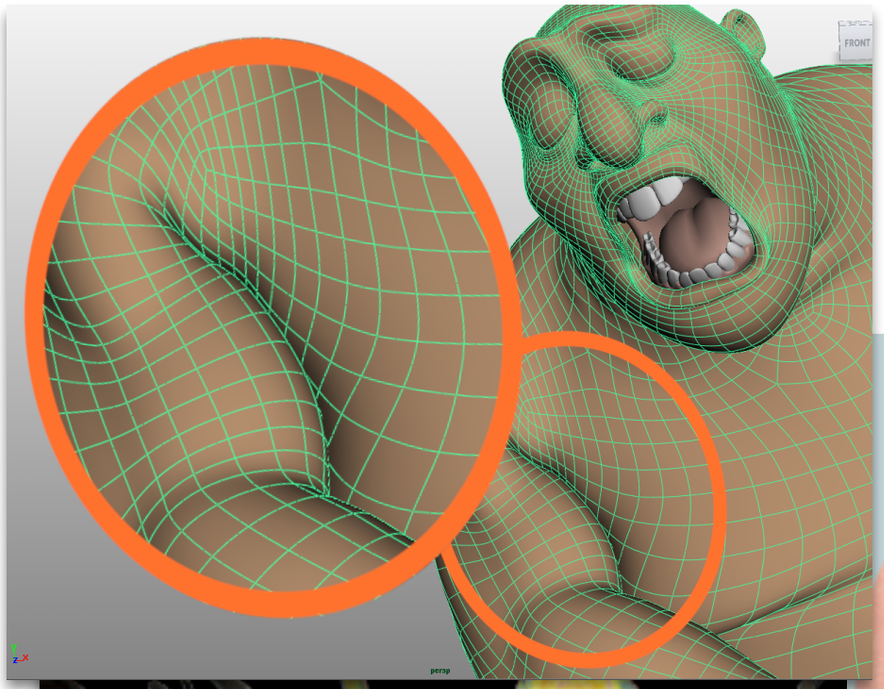
Harnessing platform heterogeneity

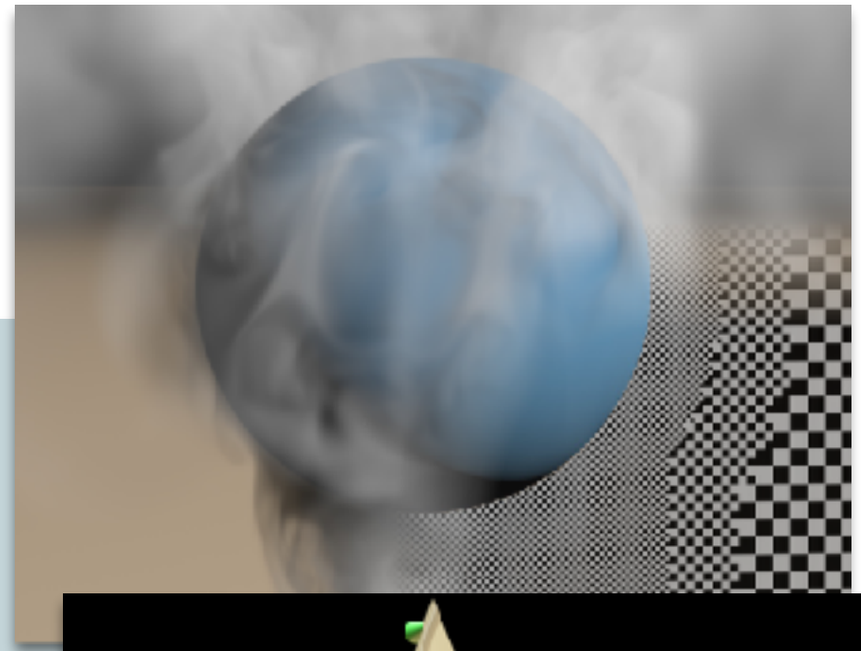


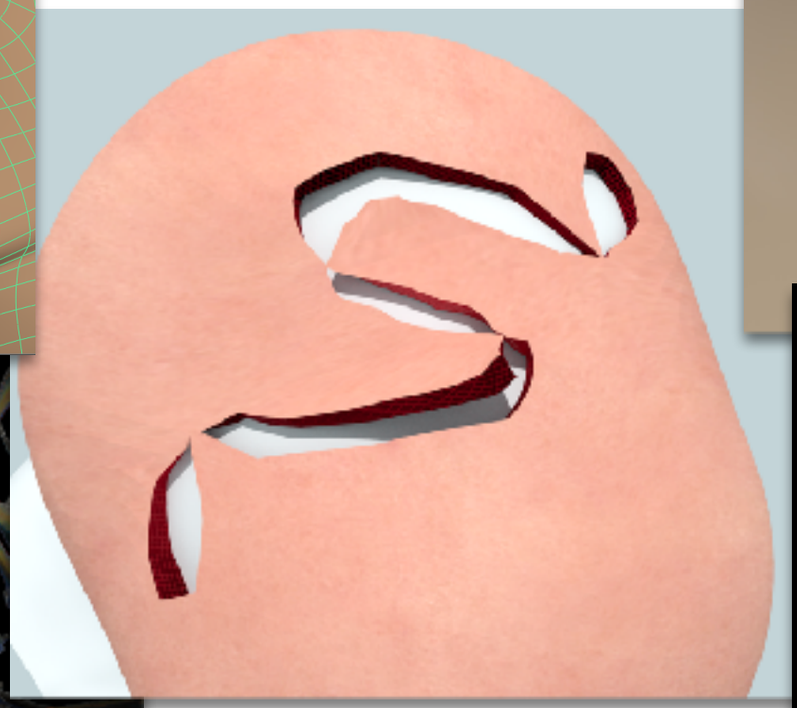


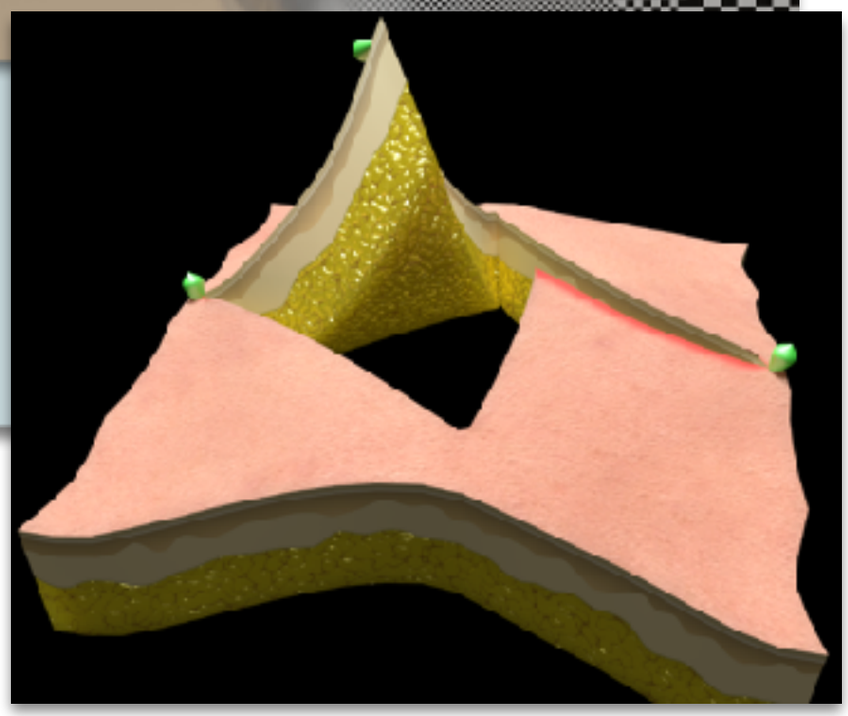
In retrospect …

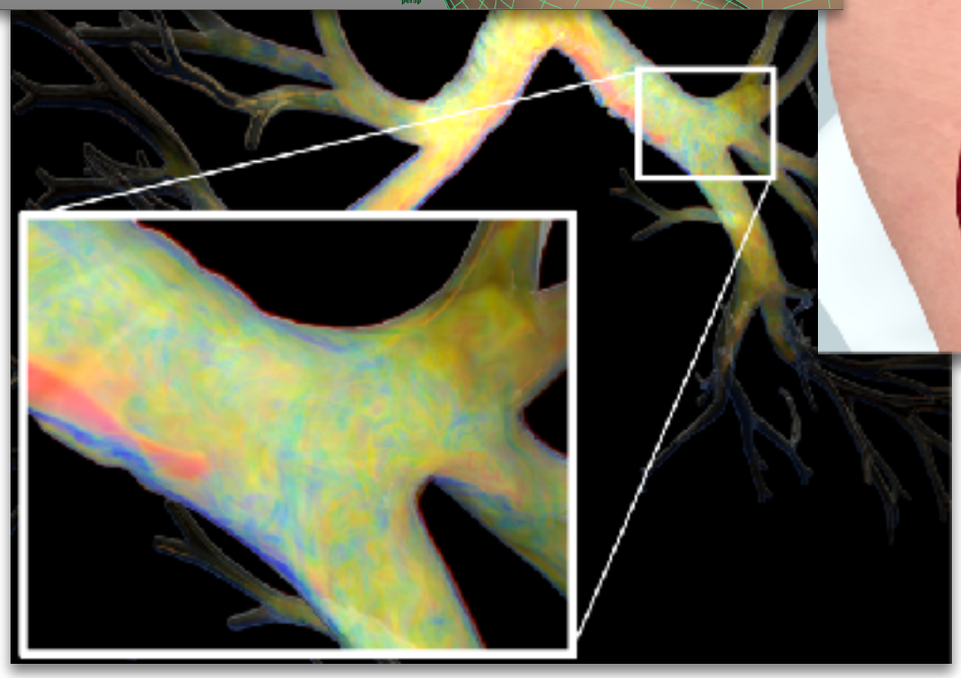
- **Should we be open to cross-layer interventions? Why?**
  - Might be a necessity for competitiveness …
  - Other disciplines appreciate this, too!
    (and graphics researchers have experience doing it)
  - A more fundable vision?
  - Historical precedent …











Digital humans, virtual surgery and fast fluids; do they have more in common than their hunger for performance?

Eftychios Sifakis

Department of Computer Sciences
University of Wisconsin - Madison

WISCONSIN
UNIVERSITY OF WISCONSIN-MADISON