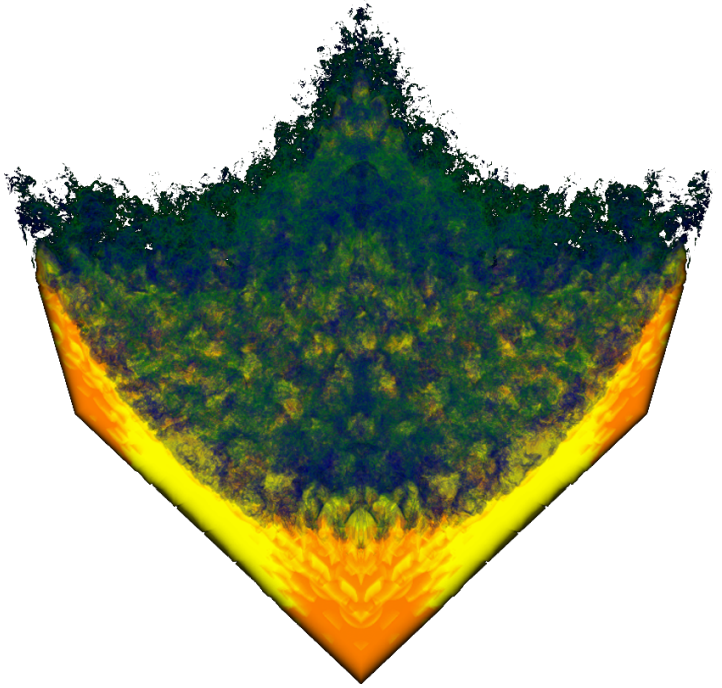
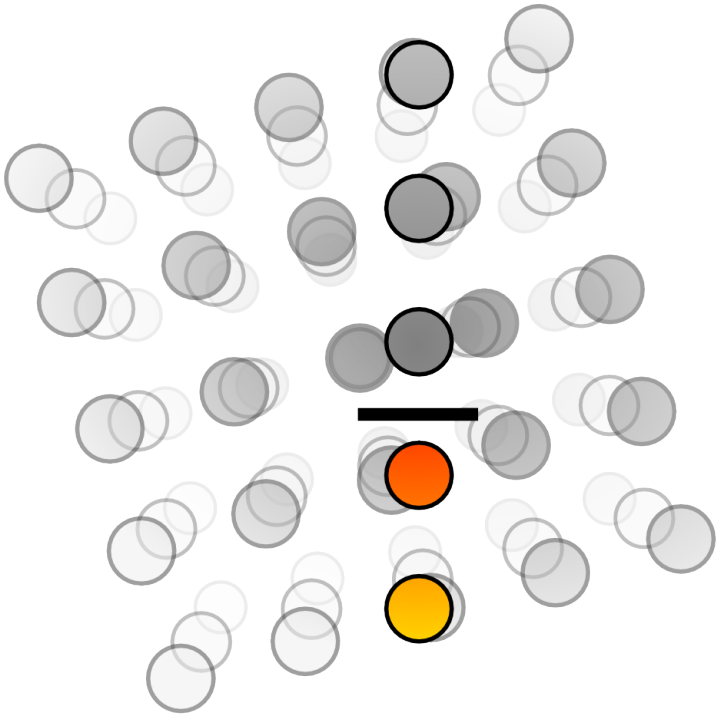
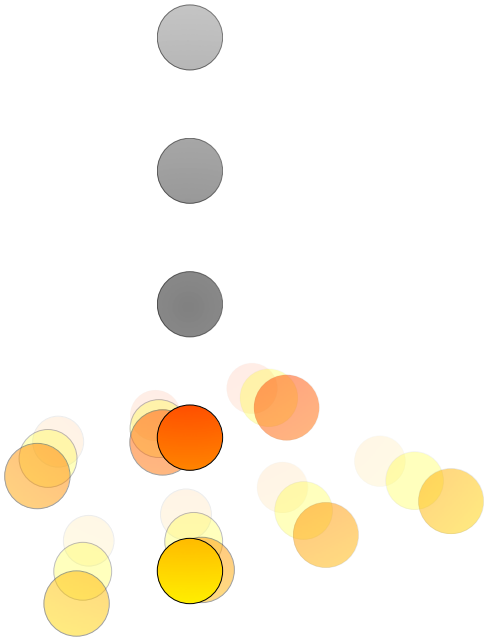
Analysis of Ray-Guided Volume Rendering

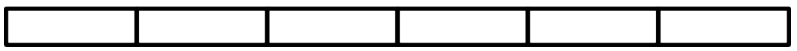
Thomas Fogal, Alexander Schiewe, Jens Krüger



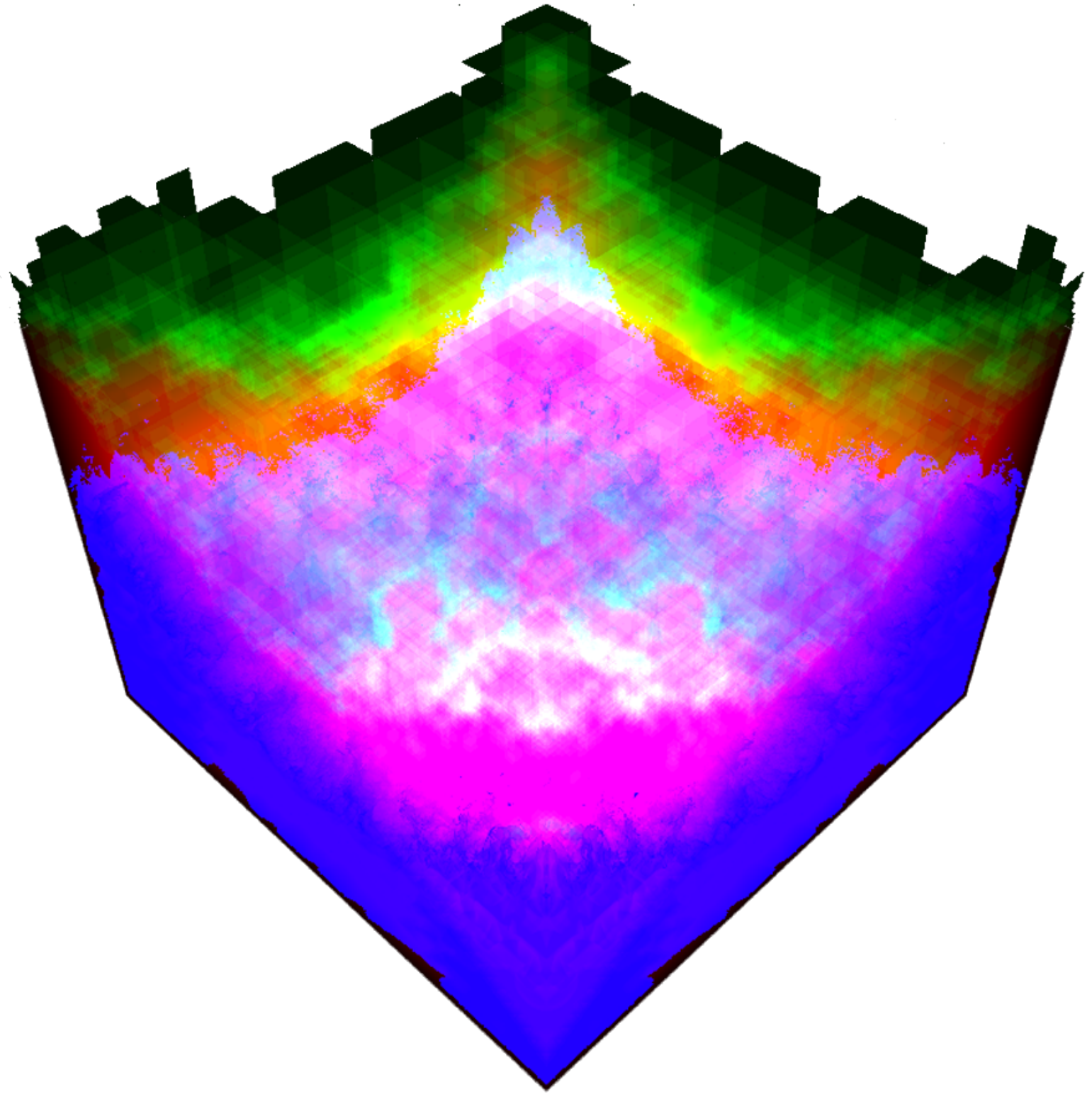
VR Background





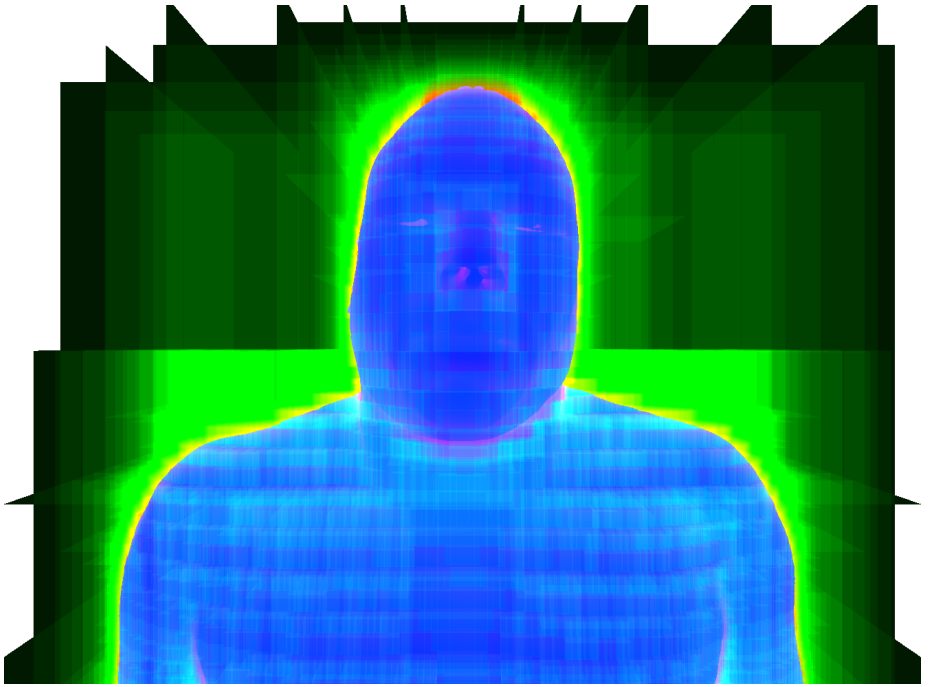


Thanks: Florian Hoffman

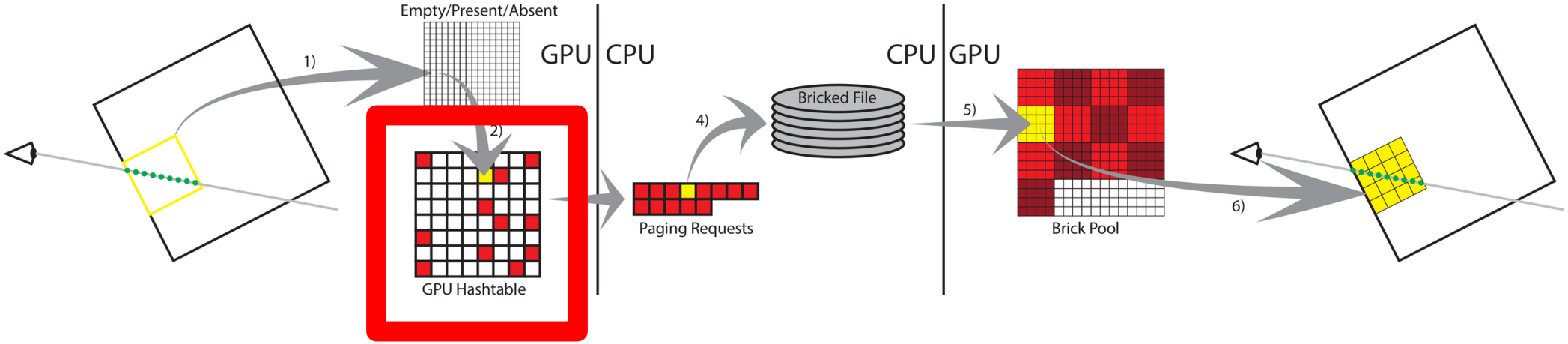


What's Important for Performance

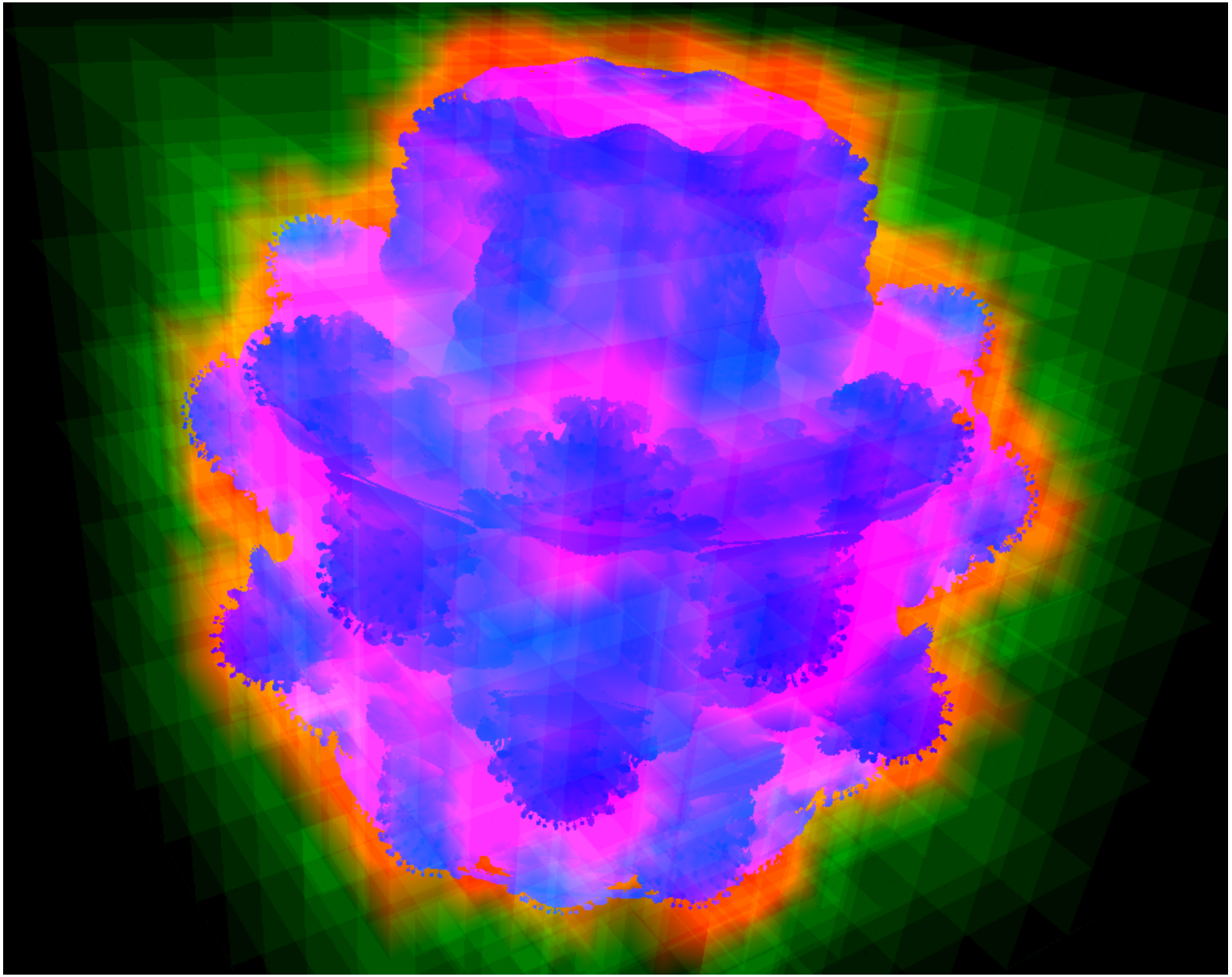
- Identifying densely-sampled regions
- Transition to coarse sampling quickly
- Communicate data needed to IO



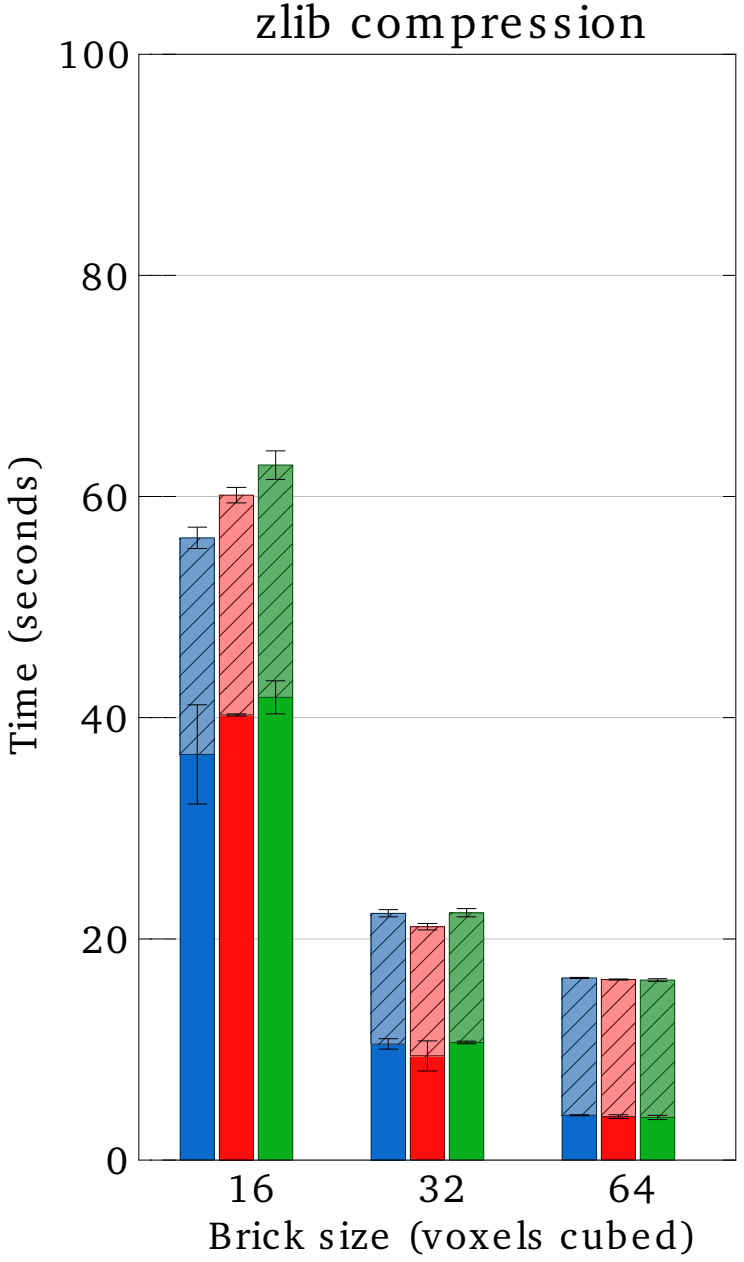
Ray-Guided Rendering

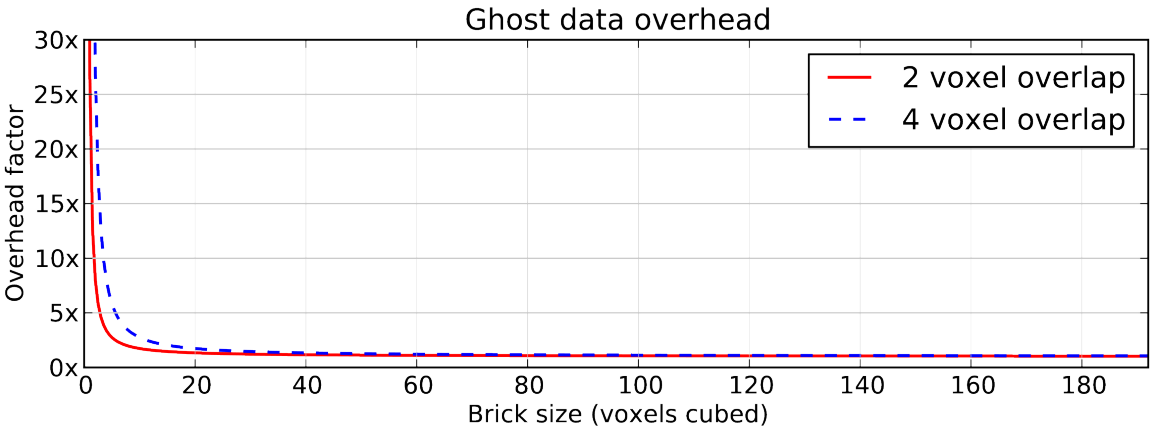


Brick Size

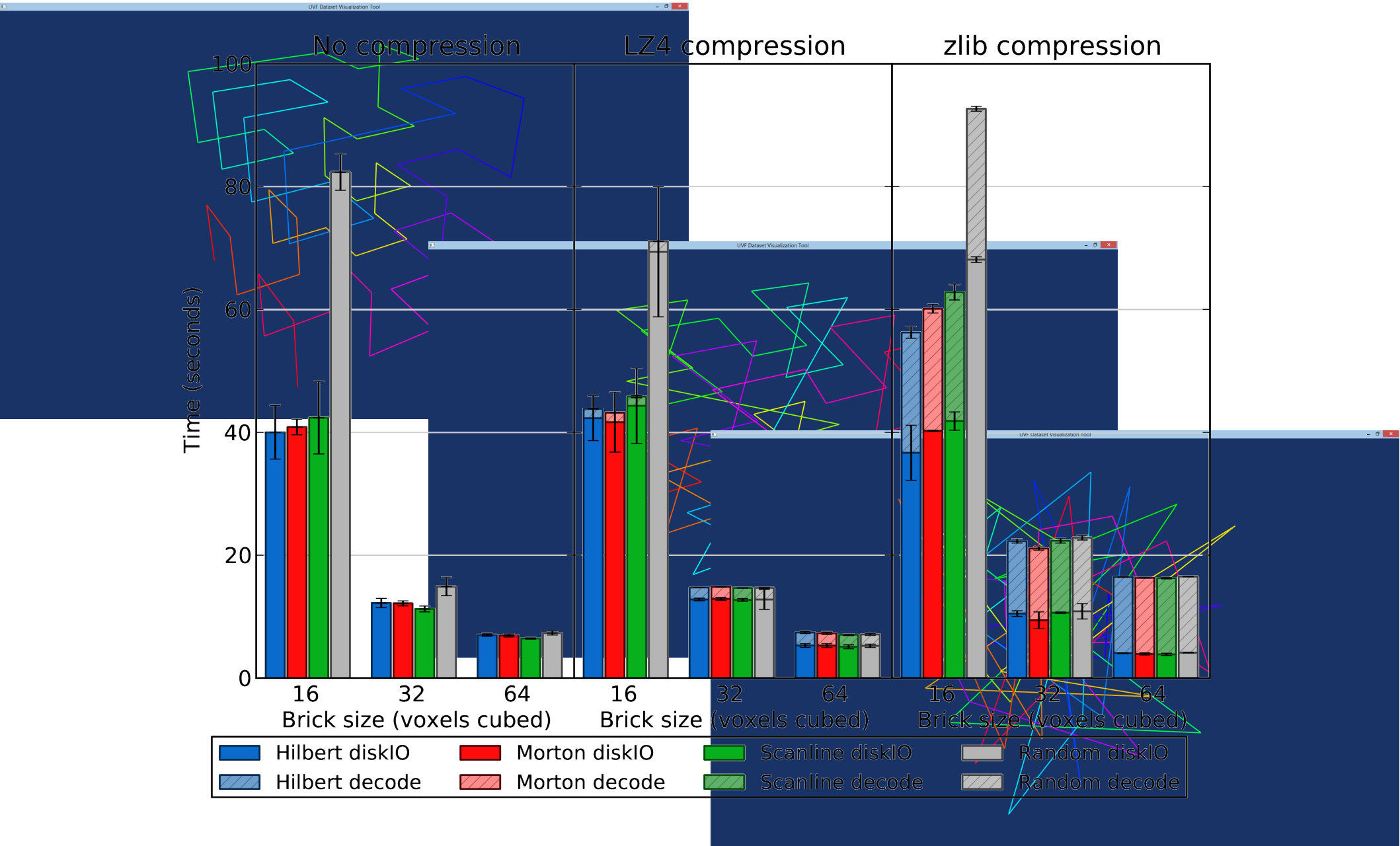


Brick Size: IO

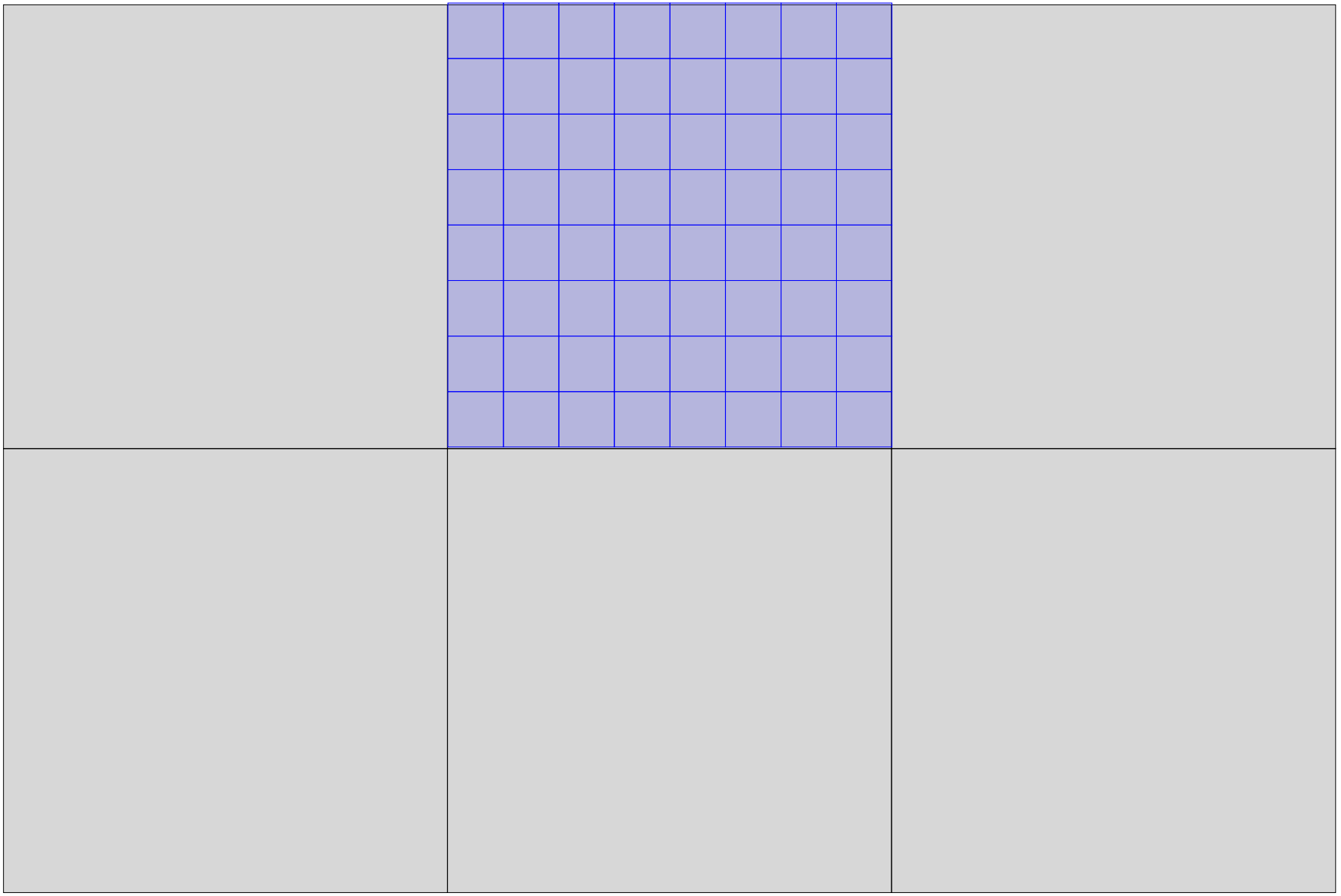




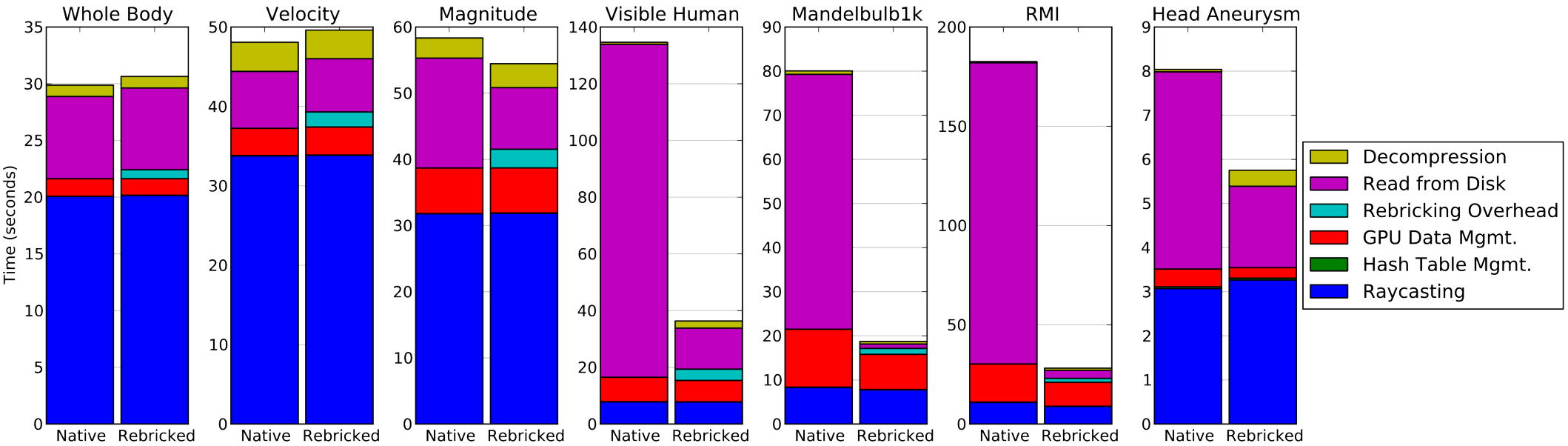
Space-Filling Curves



Dynamic Bricking

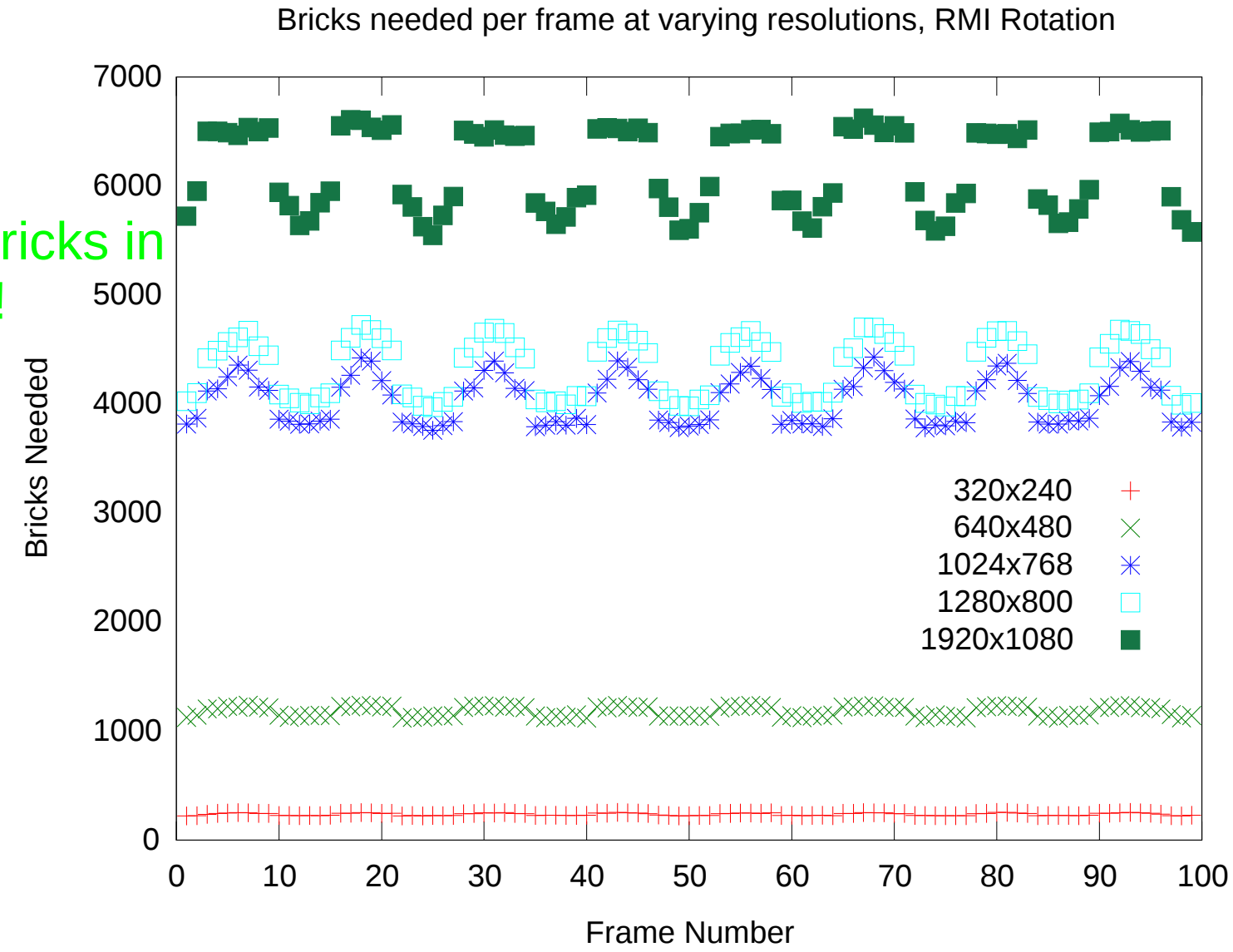


Where Does the Time Go?

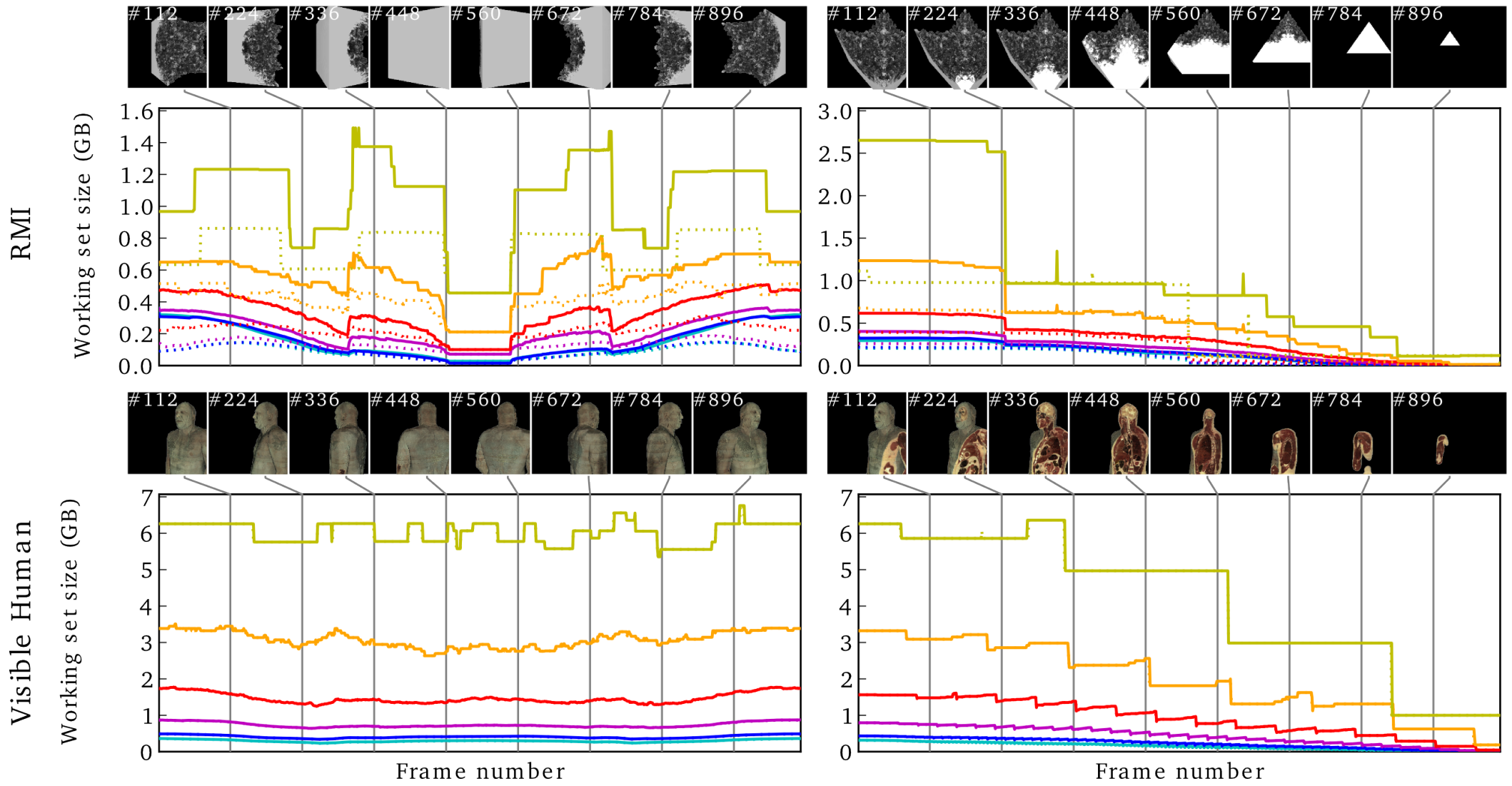


Dynamic Sampling

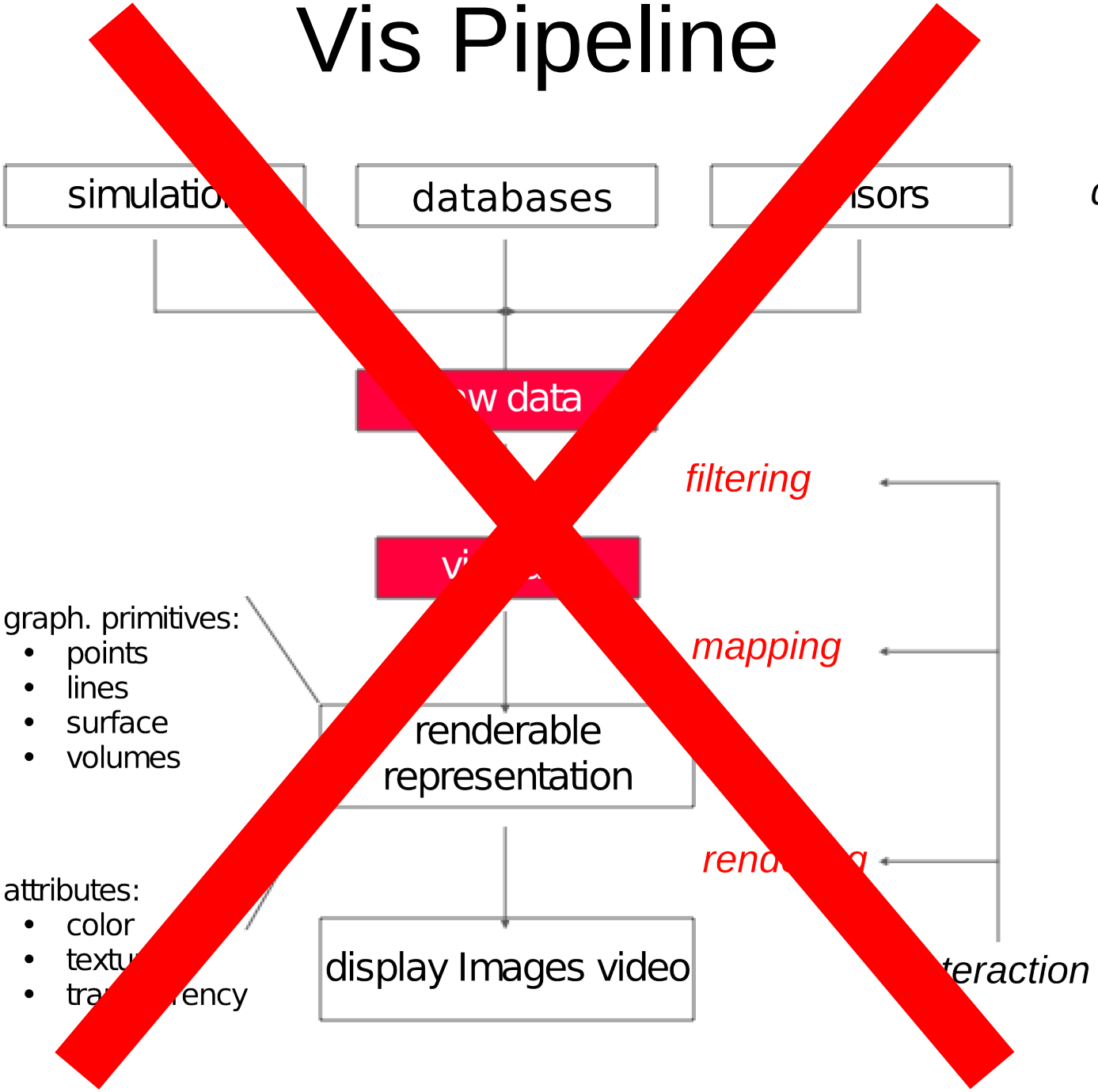
~250k bricks in Dataset!



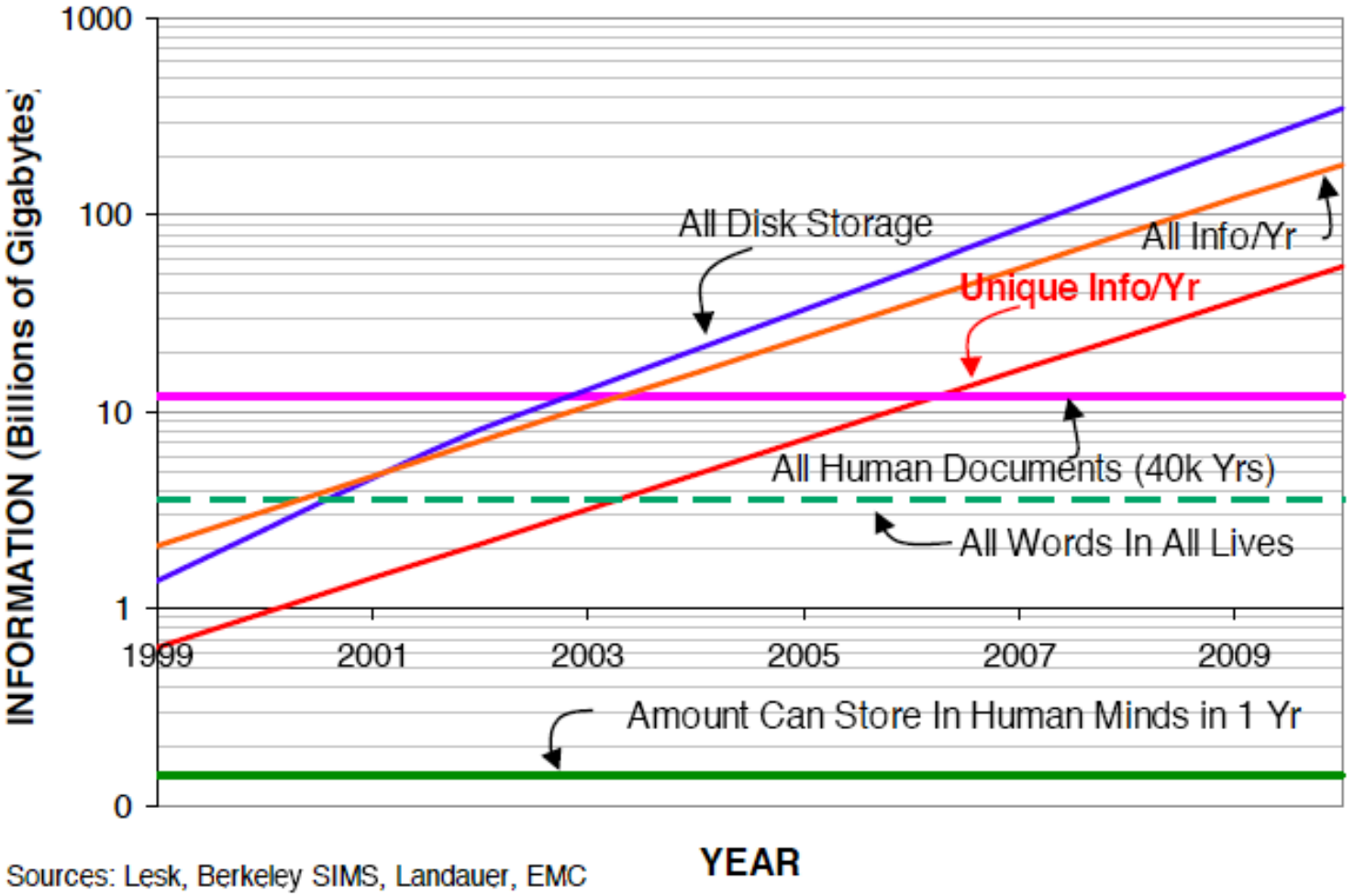
Memory Needed



Vis Pipeline

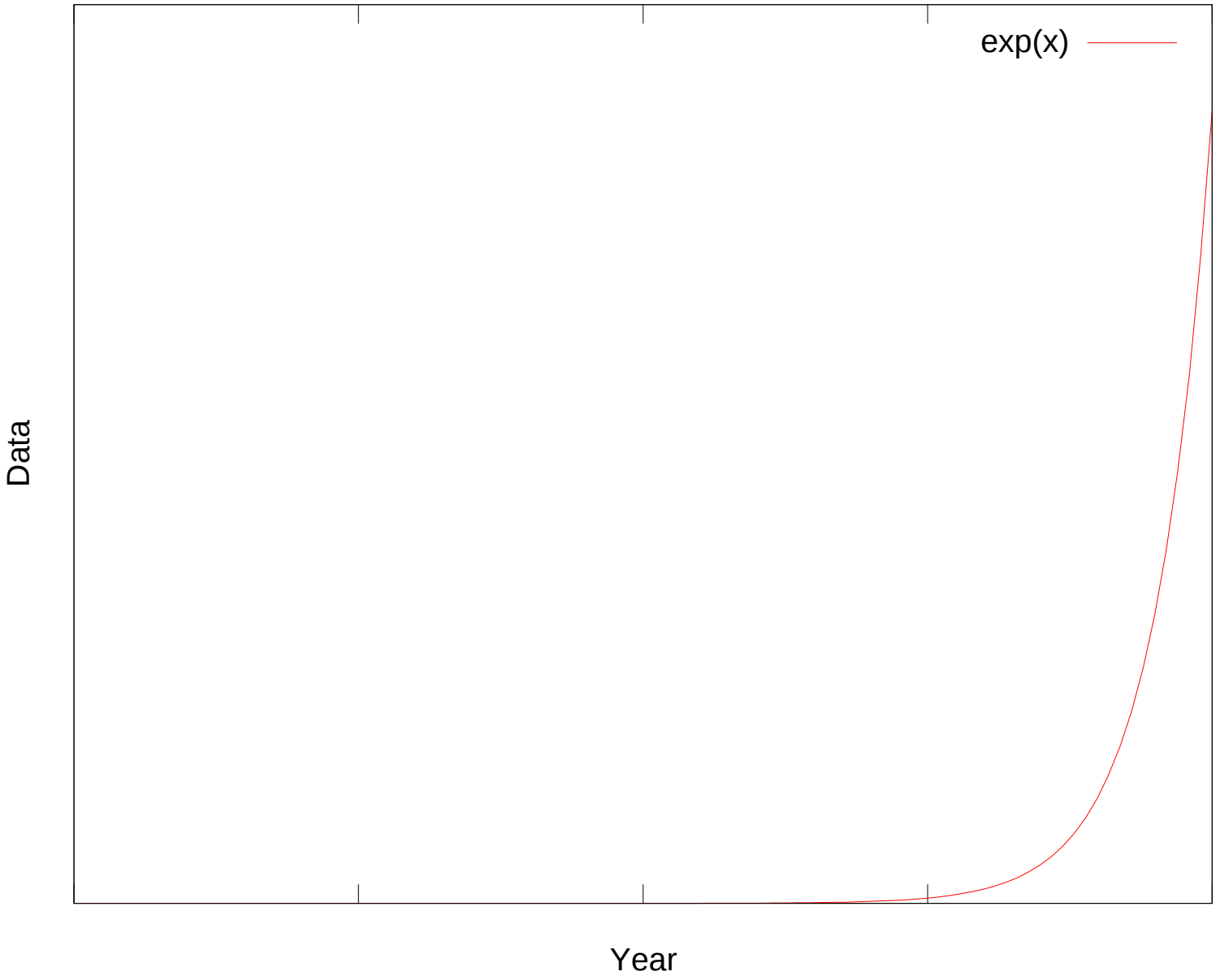


Growth of Data

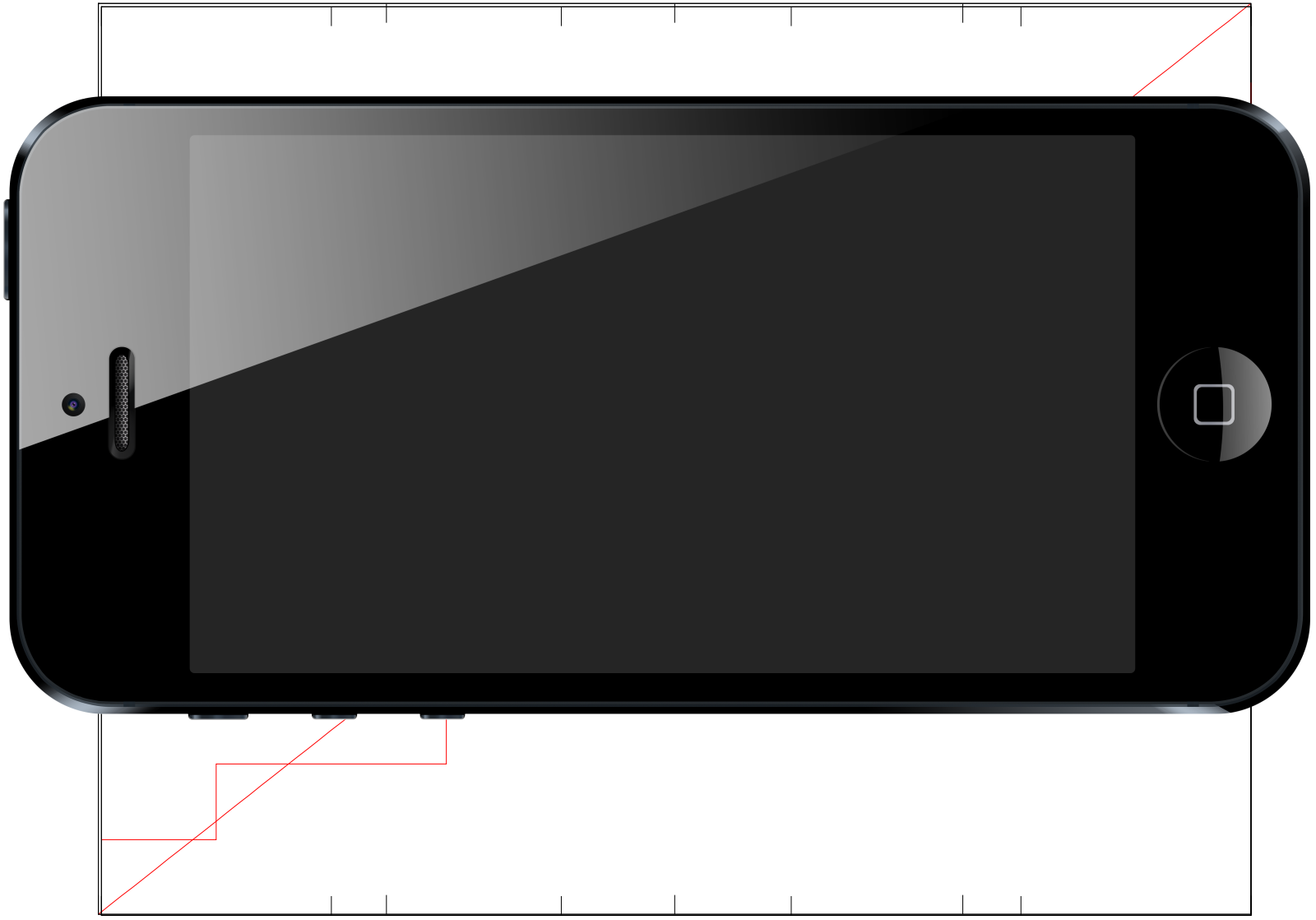


Sources: Lesk, Berkeley SIMS, Landauer, EMC

Growth of Data (2)

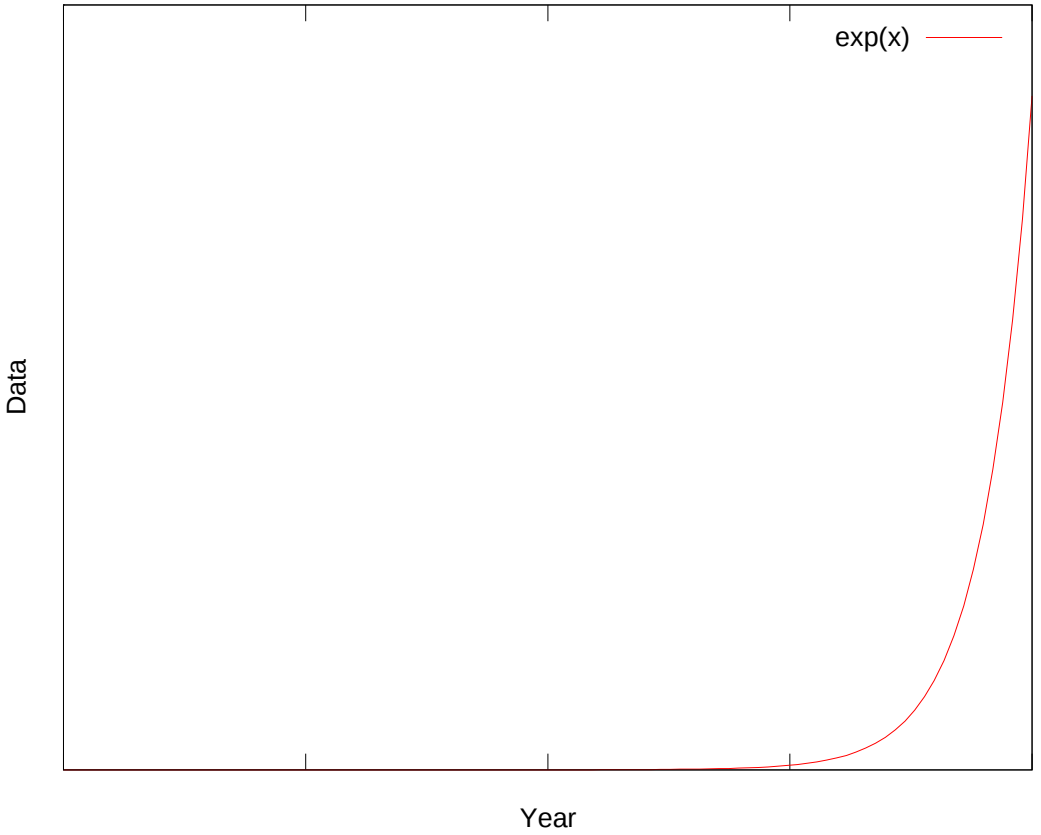


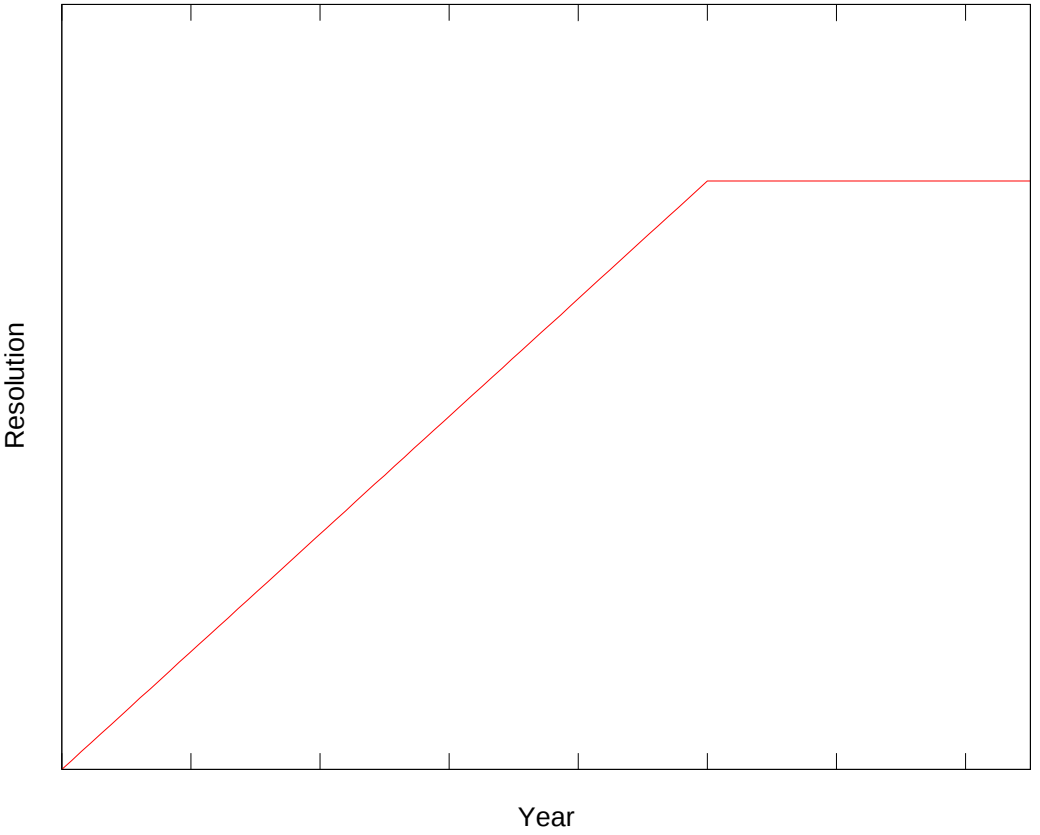
Growth of Display Devices



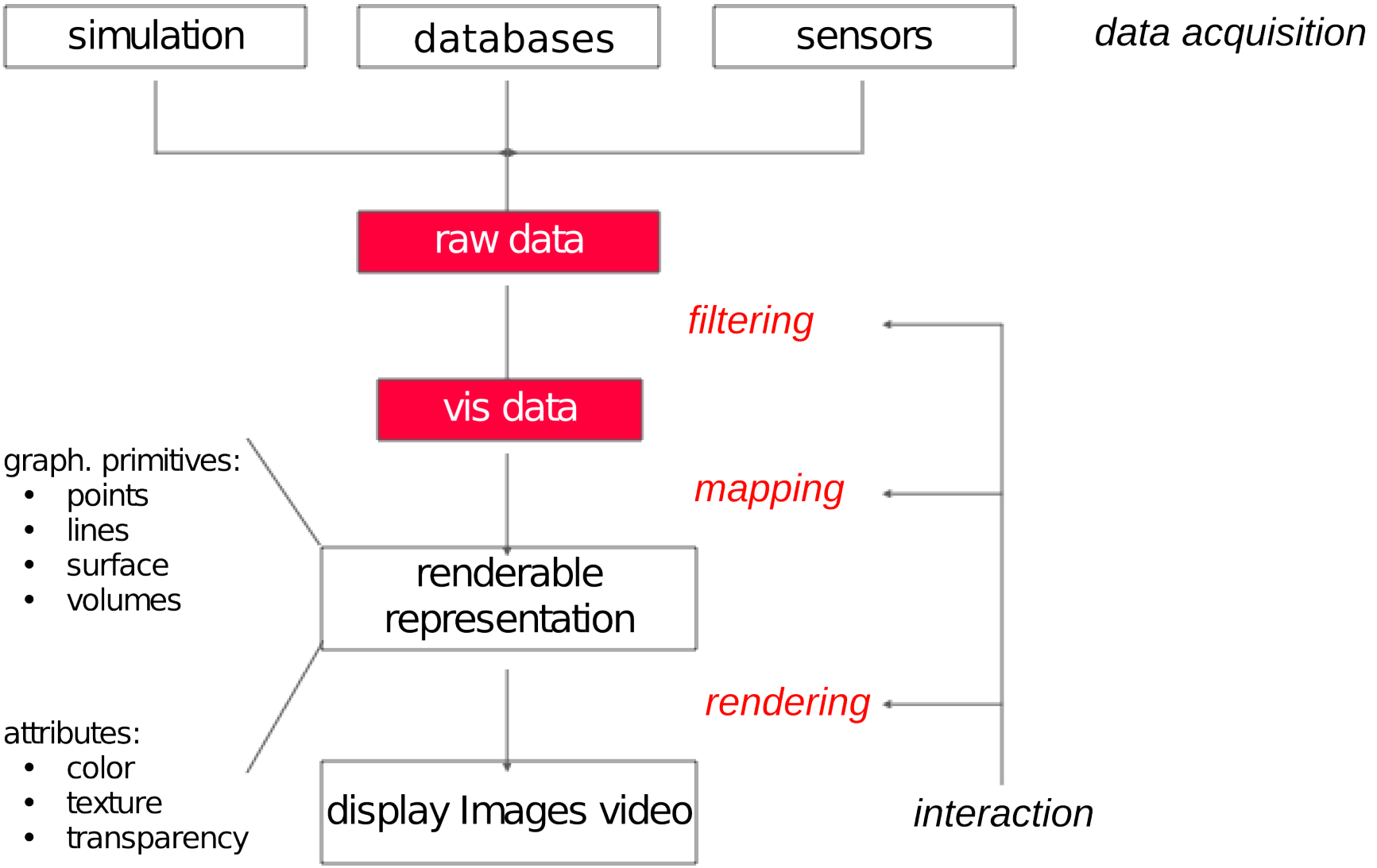
Year

Growth





Vis Pipeline



Rendering

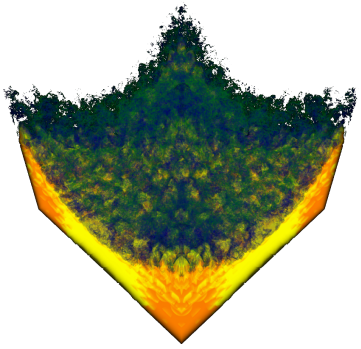
==

Filtering

?

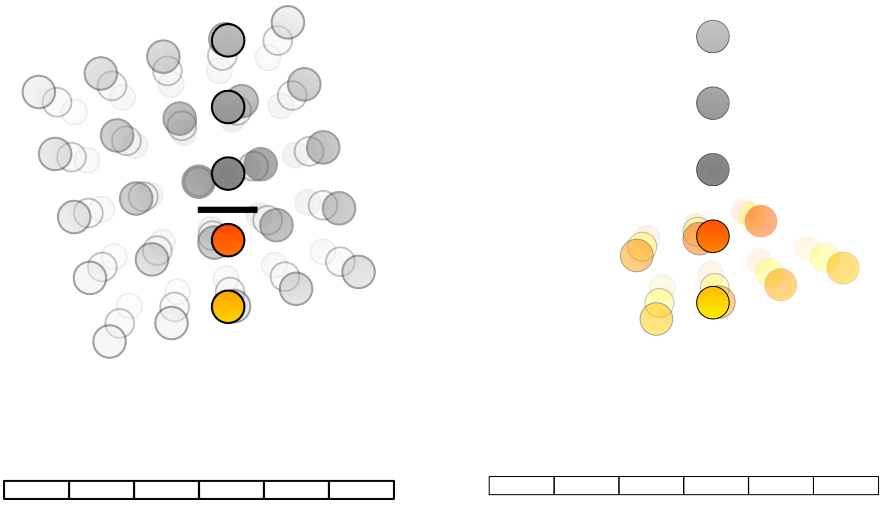
Analysis of Ray-Guided Volume Rendering

Thomas Fogal, Alexander Schiewe, Jens Krüger

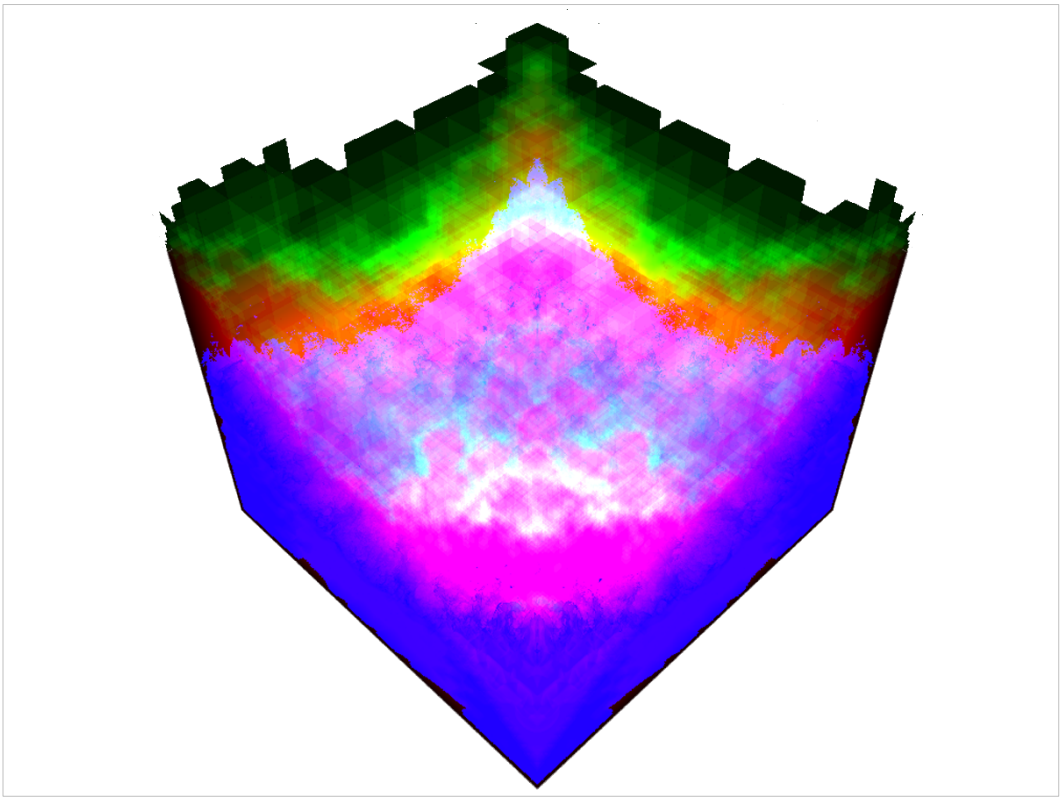


the motivation for this work was partially from these datasets. it seemed like we didn't need to do all that much work to capture these images, and yet they actually require a lot of computation. we scoured the literature: where does a modern volume renderer spend its time? and came up empty. so we started looking ourselves.

VR Background



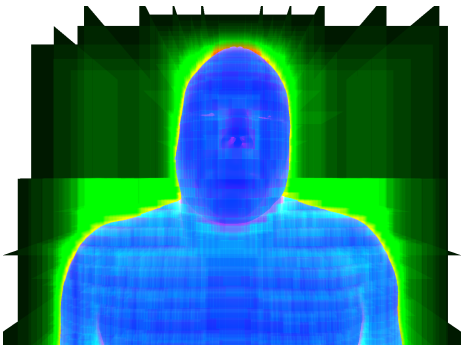
Thanks: Florian Hoffman



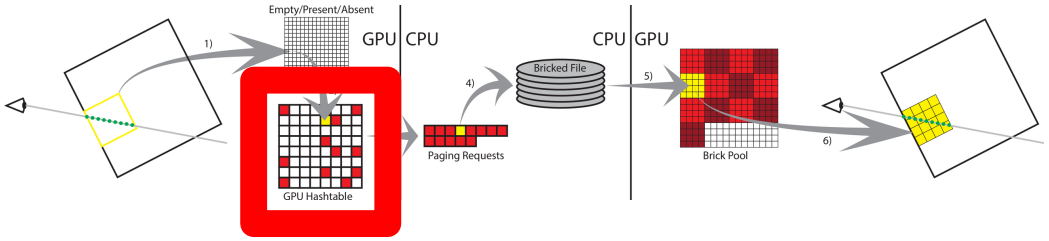
So we measured ourselves.

What's Important for Performance

- Identifying densely-sampled regions
- Transition to coarse sampling quickly
- Communicate data needed to IO

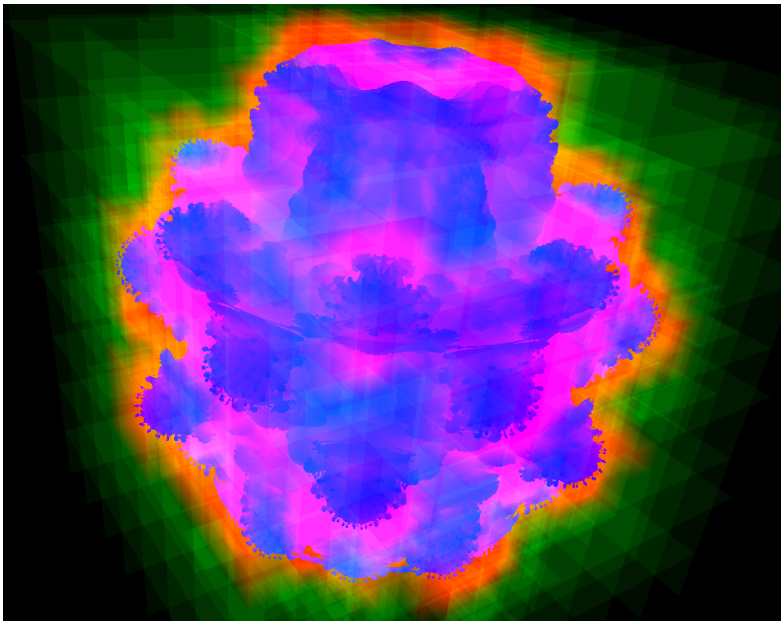


Ray-Guided Rendering



ray-guided rendering is the idea that rendering and identifying which data are needed should be co-computed

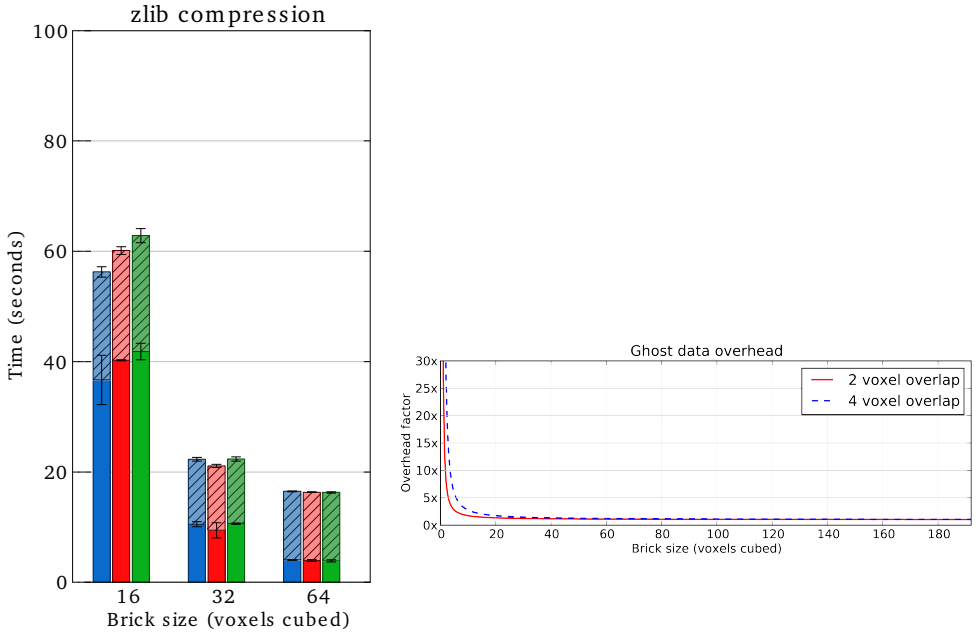
Brick Size



How does brick size effect renderings? In this image we visualize the behavior of a ray. A ray accumulates green if it skips the brick due to empty space leaping. Blue means we terminated the ray on exit from the brick, due to saturation. Red areas were sampled densely. One can see that a lot of the data falls into the 'blue' and 'green' categories: that is, very little work needs to be done for the majority of rays. Only the red areas require lots of computation.

Clearly, we should desire small bricks, to more closely approximate these distinct regions.

Brick Size: IO

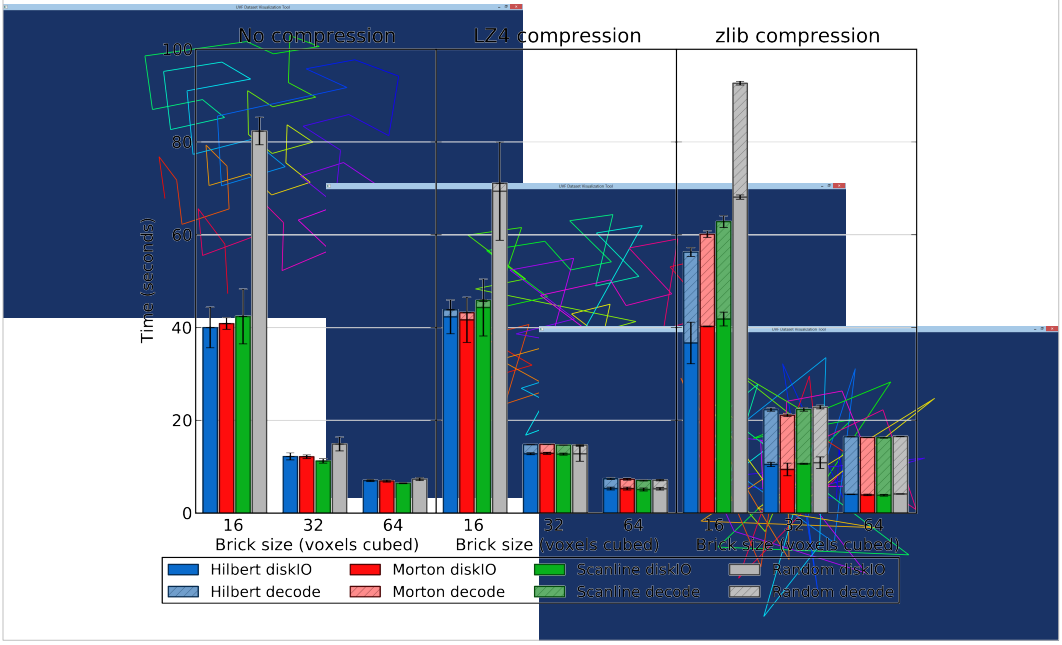


Unfortunately, small bricks give really awful IO performance. As the transfer sizes shrink, we end up paying a lot for each brick. Furthermore, since we need to include ghost data in each brick (thereby copying some data), we can extend the size of the data set pretty dramatically with small brick sizes (50% at 32^3).

Compression is useful in reducing the size of data, but does not actually improve IO time. With compressors like zlib or bzlib, compression can actually have a significant effect on performance.

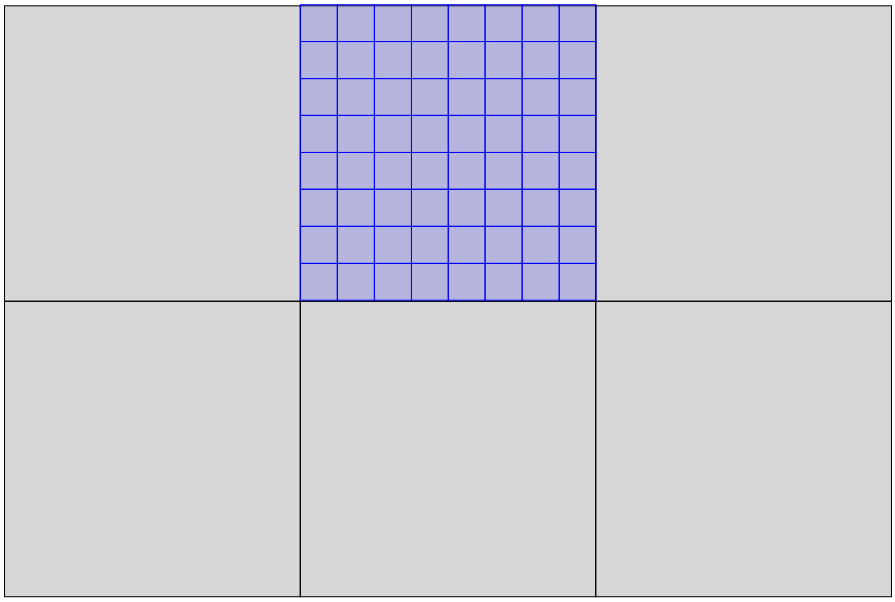
(might want to add transfer vs. seek time image from EGPGV paper)

Space-Filling Curves



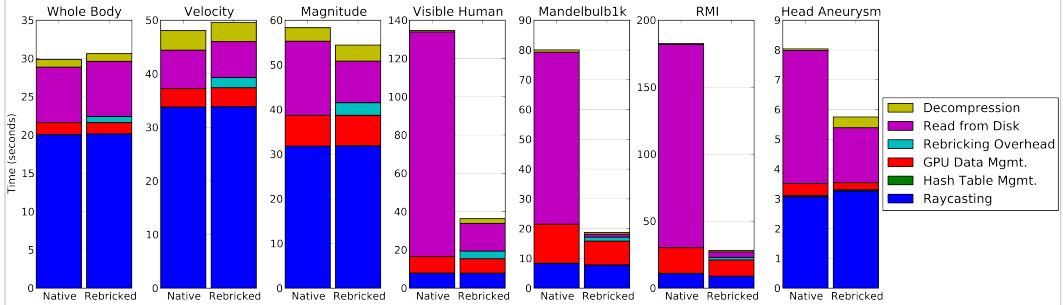
Since IO was a big problem, we applied the conventional wisdom from the community: use a space-filling curve to minimize the 'distance' between two bricks; reading one should automatically put another in cache. If we have good spatiotemporal locality for bricks, this should be a win. Unfortunately this didn't prove to be much of a win over standard 'scanline' ordering.

Dynamic Bricking



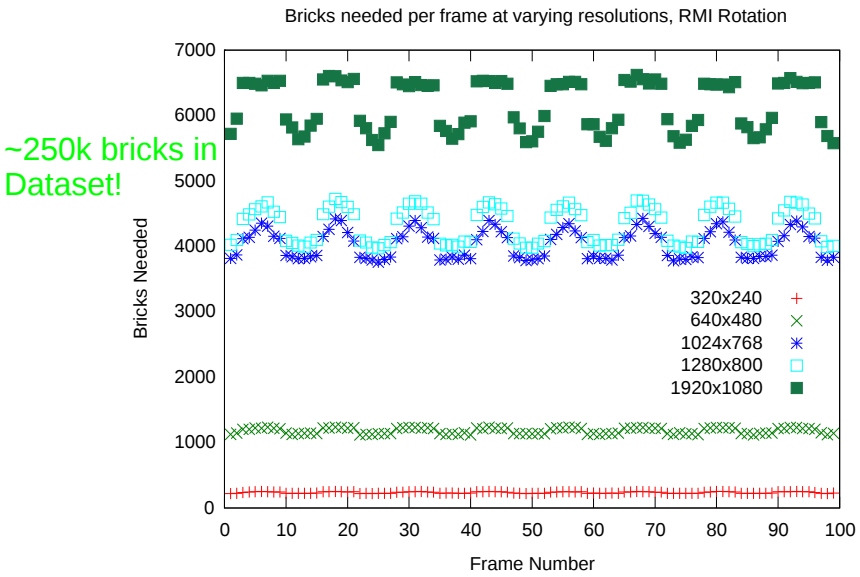
Since the problem was *transfer* time, not seek time, we sought a solution that minimized transfer time, but still gave us small bricks so the renderer would perform well. What we decided on was generating the bricks at runtime. We still do a precomputation up-front; this gave us a base for the data and limits the amount of work done. But we take those large bricks and chop them up into tiny bricks on demand. This means we still do large reads from disk, but our renderer sees the tiny bricks that it wants for e.g. early ray termination.

Where Does the Time Go?



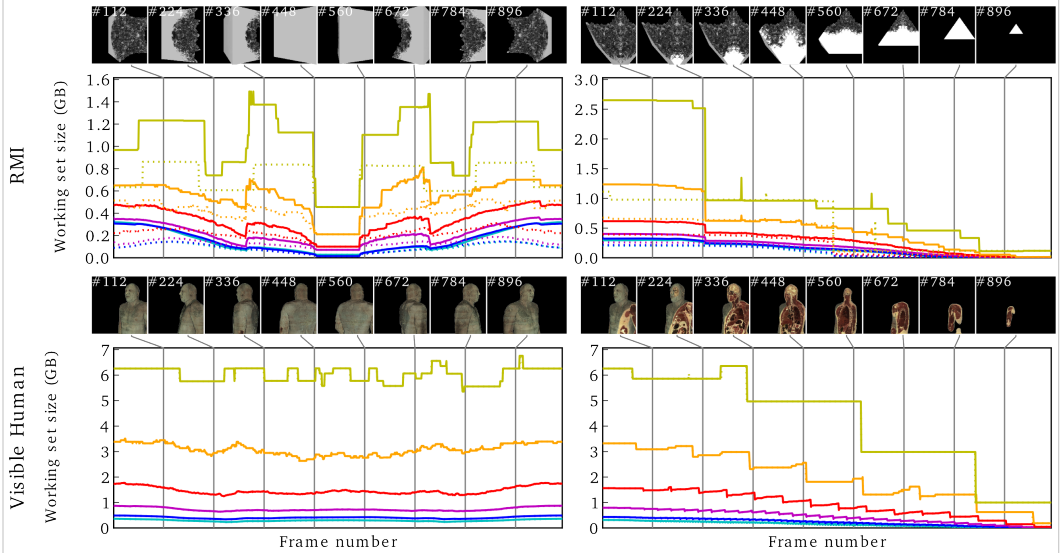
This figure shows where we spend our time. As one might guess, the majority of the time is spent in IO and rendering. Note that the whole body, magnitude, and velocity data sets actually spend a lot of time rendering: this happens with lots of large, thin/transparent structures, which must be sampled densely but do not cause rays to saturate.

Dynamic Sampling

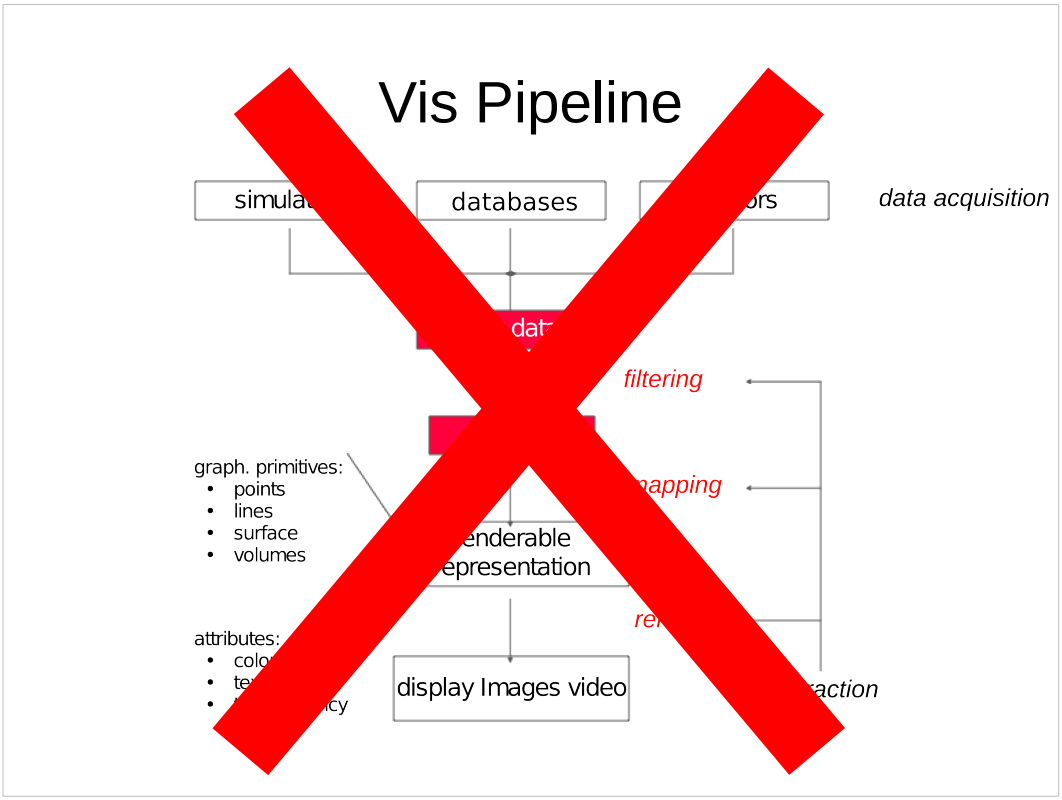


sampling is heavily output-driven: fewer rays means we can sample the data coarsely. this means we can load much less data.
side note: this means we should (as reviewers) require high-resolution output frames
graph generated using a DS bricked via 36^3 bricks: 245760 bricks total

Memory Needed

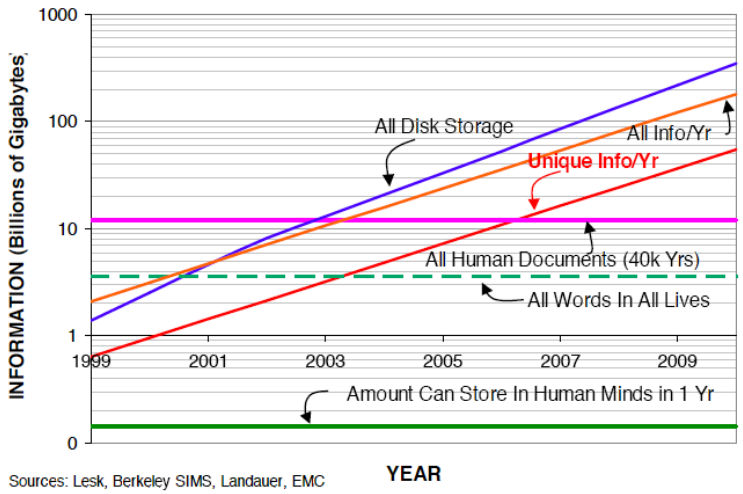


here's a measure of how much data we need, per-frame, for some common operations.
Wholebody: 1.5Gb; RMI: 7.5Gb; VHuman: 12.2Gb
I am sure almost everyone could guess that the data needed is a subset of the overall volume. The point here is how *small* that subset is.

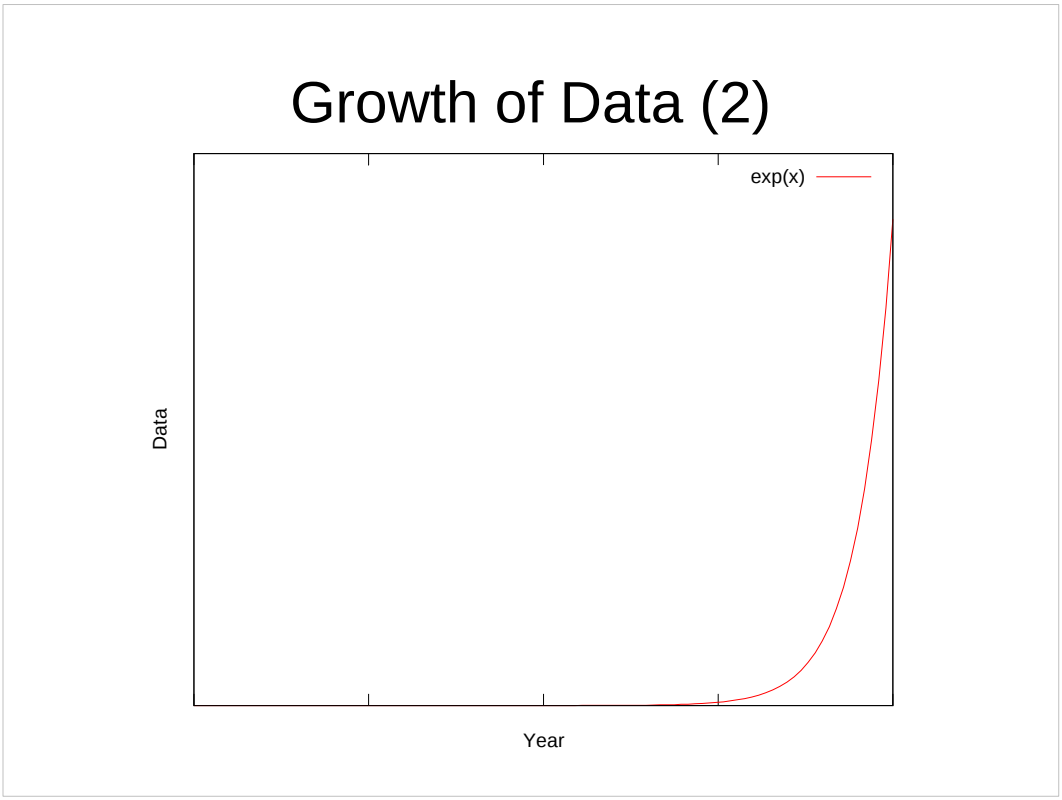


This is the standard vis pipeline. Actually, I stole this directly from the lecture we give graduate students. The problem with this is that it's the wrong way to teach students how to do visualization

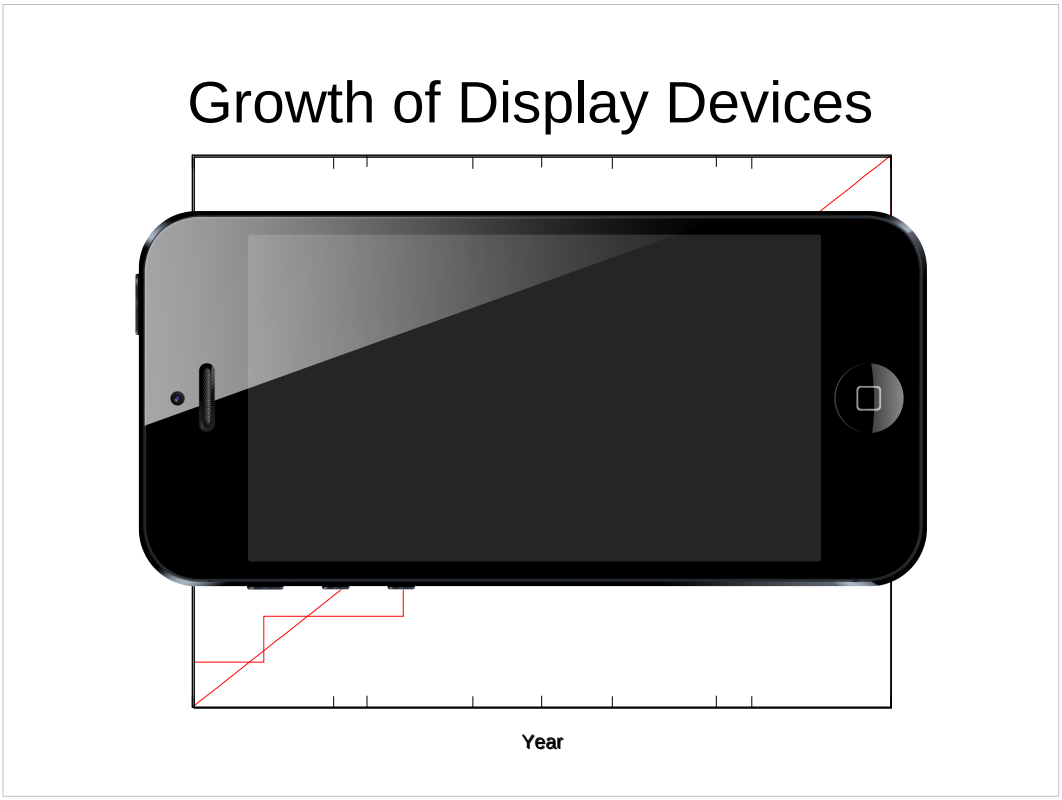
Growth of Data



Let's take a look at how fast data are growing. This graph has a lot of information, but what let's just focus on this “All info per year” line now. And all I really want you to note is that we've got a 'linear' growth on a log scale.

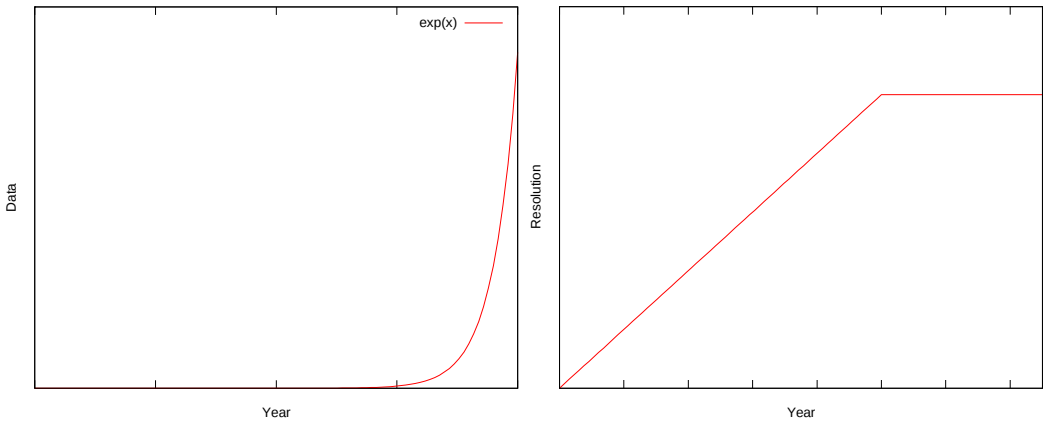


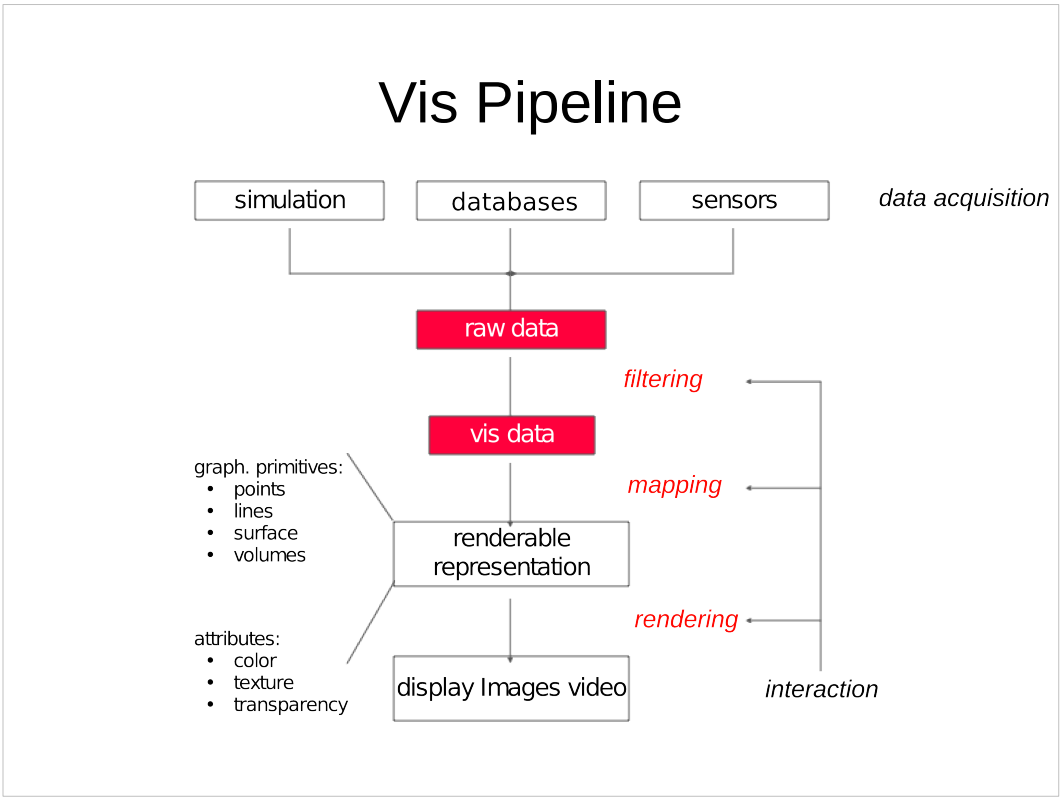
Thus: data are growing exponentially.



What about display devices? Displays look much more like a linear function. Actually, if we want to get technical, it's more of a step function; but either way, it's linear.
And, in recent years, it has hit a peak: so-called “retina resolution”.

Growth





This is the standard vis pipeline. Actually, I stole this directly from the lecture we give graduate students. The problem with this is that it's the wrong way to teach students how to do visualization: it implies this waterfall model which is really awful for performance.

Rendering

==

Filtering

We've got this whole vis pipeline thing upside down. Rendering *is* filtering; they should not be separate operations, else we give up 2 to 3 *orders of magnitude* of performance. We need to stop considering large data as a waterfall, with our job being to direct that flow to where it's desired. Instead, consider large data a lake and our task is to identify where we should insert our 'sampling straw'.