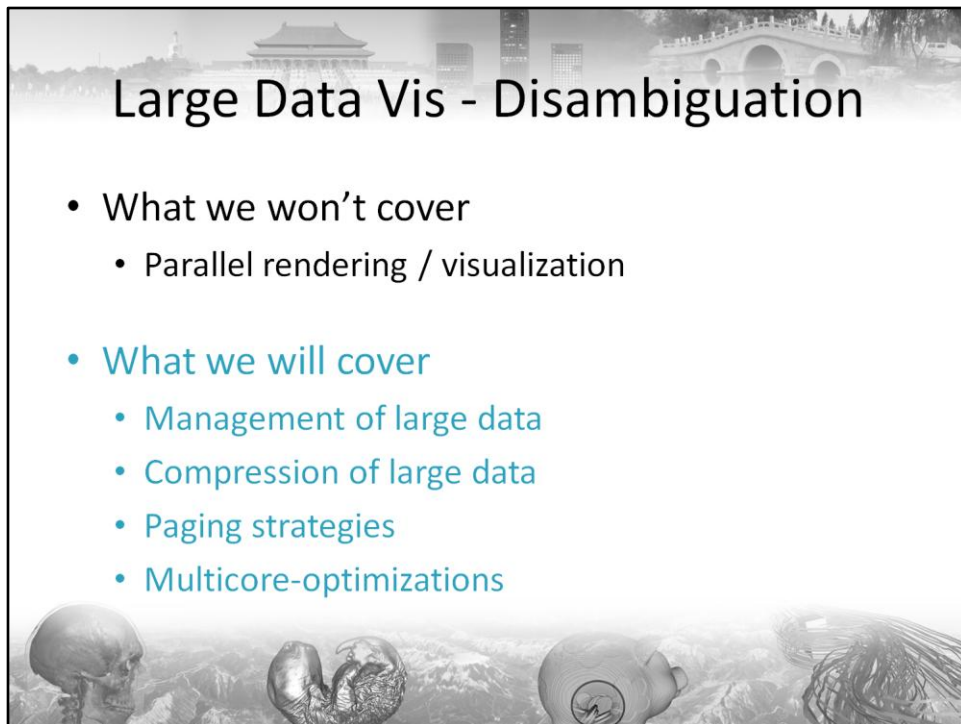


This is the part „Large Data“ of the tutorial „Interactive Methods in Scientific Visualization“ held at the IEEE Pacific Visualization Symposium 2009 by Jens Schneider.



# Interactive Methods in Scientific Visualization





When talking about the visualization of large data, many associate this topic with parallel rendering and visualization. While they are certainly right to do so, we will not cover this topic in this part of the tutorial.

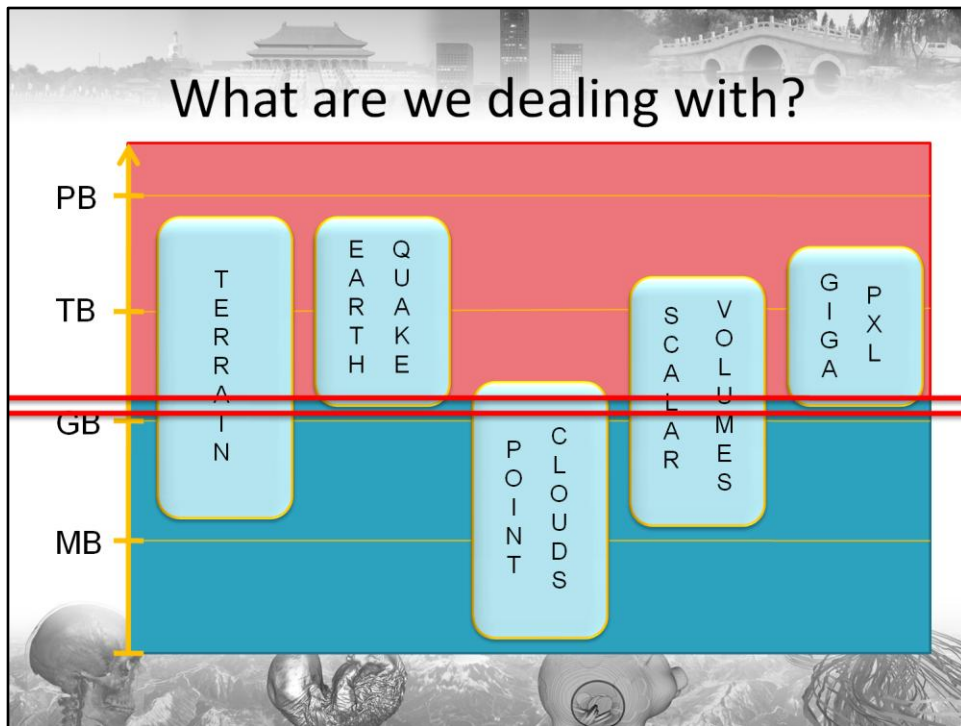
However, we will address the management of large data including data compression. We will also address various paging strategies and will mention multicore-optimizations. Our target system for all the presented examples and data types is a „standard“ desktop machine, equipped with a multi-core processor, a decent amount of RAM (4-8 GB), as many harddrives as you can grab (not a severe restriction anymore at 10ct / GB) and a fair GPU (typically DirectX 10 compliant, choose your favourite vendor).



# Large Data – Road Map

1. **What are we dealing with?**
2. Know your system!
3. Terrain Data
4. Terashake 2.1 Simulation
5. Point Clouds
6. Scalar Volumes
7. Gigapixel Images
8. Conclusions & Remarks

First, we will quickly show you what „large“ data means by today, and keep in mind to think big enough for the future.



Let's have a look what we mean by large data by today.

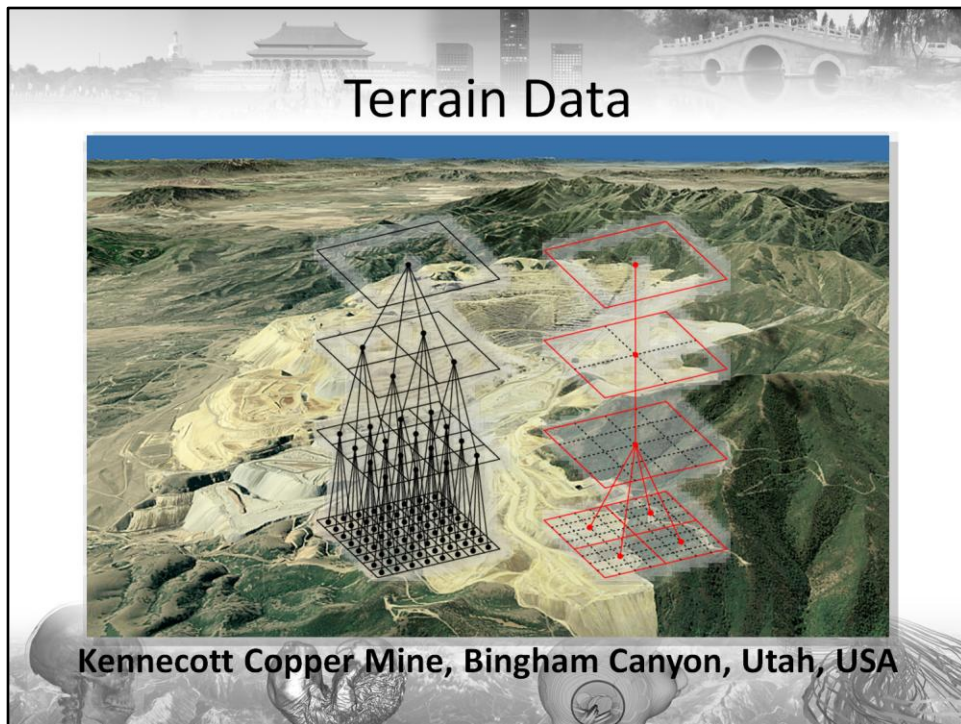
Firstly, Terrain data is known to be quite large. Typical sizes range from a few dozens of MB to hundreds of TB. For instance, China resolved at 0.25m per sample amounts to roughly 700 TB for both DEM and texture.

Secondly, geological sciences tend to generate fairly large data. For instance, the Terashake simulation conducted by the San Diego Supercomputing Centre amounts to about 20GB per time step and consists of several hundred time steps.

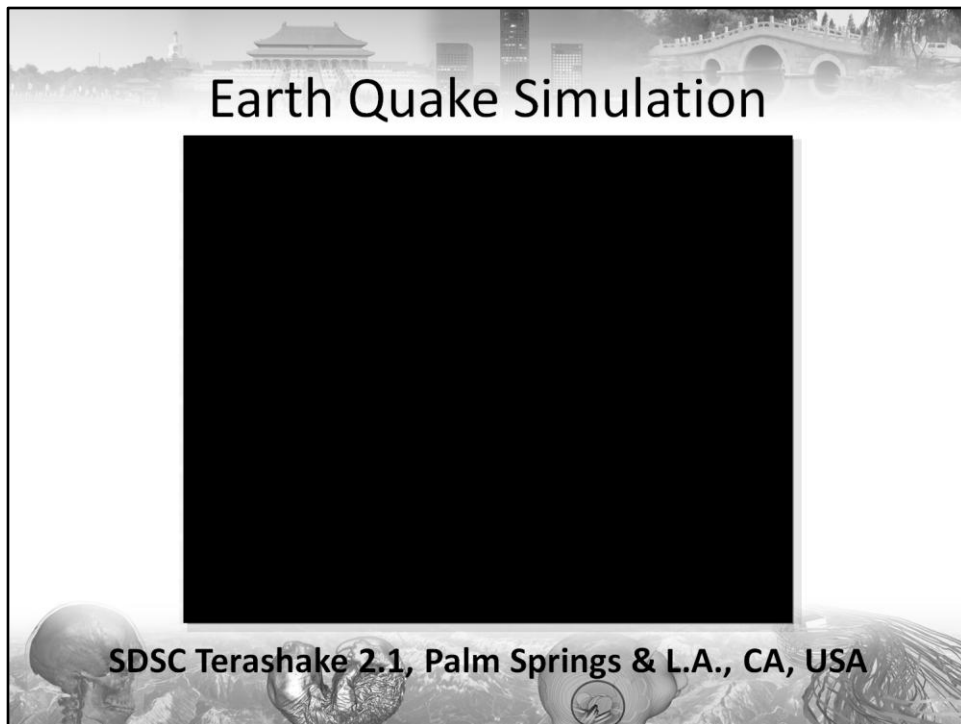
Another example for large data are point scans and meshes. The Stanford Computer Graphics group acquired point scans of various statues by Michelangelo that range from about 7 million points to up to 350 million points. This amounts to several GB of data.

Current state-of-the-art medical scanners generate scalar volumes that range from a few megabytes to some terabytes. Especially the medical data in everyday-use is expected to grow at significant rates in the near future.

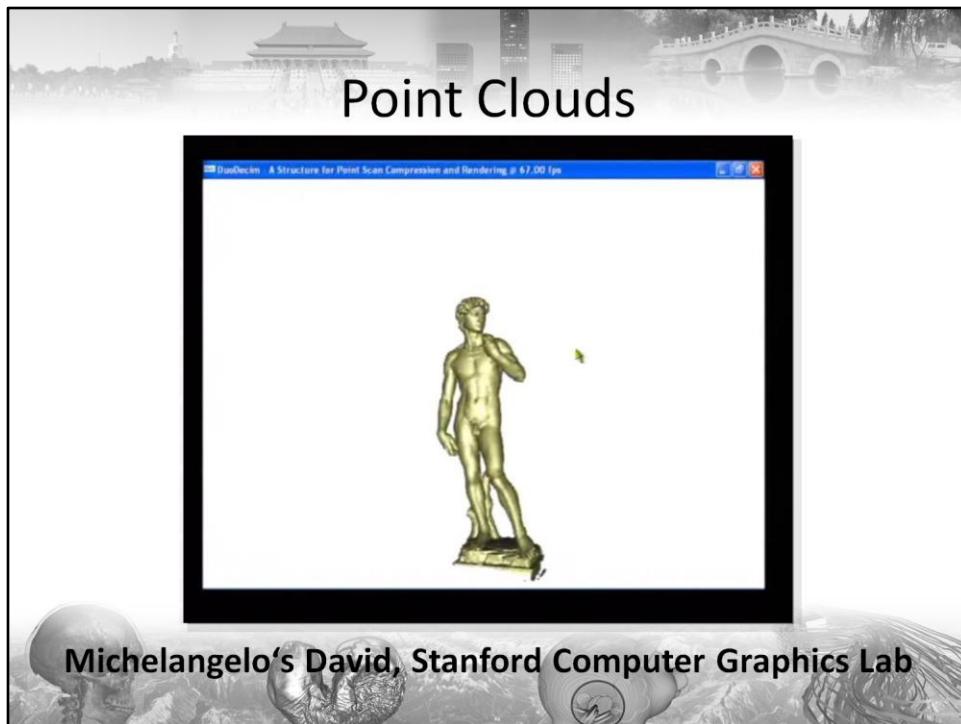
Last but not least, there has been a certain trend to Gigapixel images recently. Naturally these images consume some GB in raw format and are expected to grow into the mid TB region soon, since acquisition methods scale very well.



Terrain Data will be our first example, here you see the Kennecott Copper Mine in Utah. The picture above is a very small piece of a data set comprising the entire State of Utah, USA. While my colleague Christian Dick will explain in detail how to render this data, I will focus on the data management and paging strategies required during pre-processing and run time.



This is a video of our visualization of the SDSC Terashake 2.1 Simulation. We choose to somewhat counter-intuitively visualize the deformation in the upper layers of the surface by particles. By doing so, you can observe many structures that are really hard to grasp with other visualization methods. We attribute this to the fact that particles in a sense „accumulate“ deformations, thereby forming structures that are symptomatic for the underlying wave types.



This small video demonstrates the smallest of the meshes from the Digital Michelangelo Project, David, which with its ~28M points is already considered large by many. However, there are much larger data sets in the repository, for instance the unfinished Atlas statue comprising about 255M points.





Here we show you the fullcolor version of the visible human. The data set was rendered using S3TC encoded textures in a bricked renderer. Note that the tattoos have not been seen for over a decade, since they were to our knowledge not visualized before. This image was rendered using the Image 3D package from SCI, Utah.

## Gigapixel Images



Image by courtesy of J. Kopf and O. Deussen

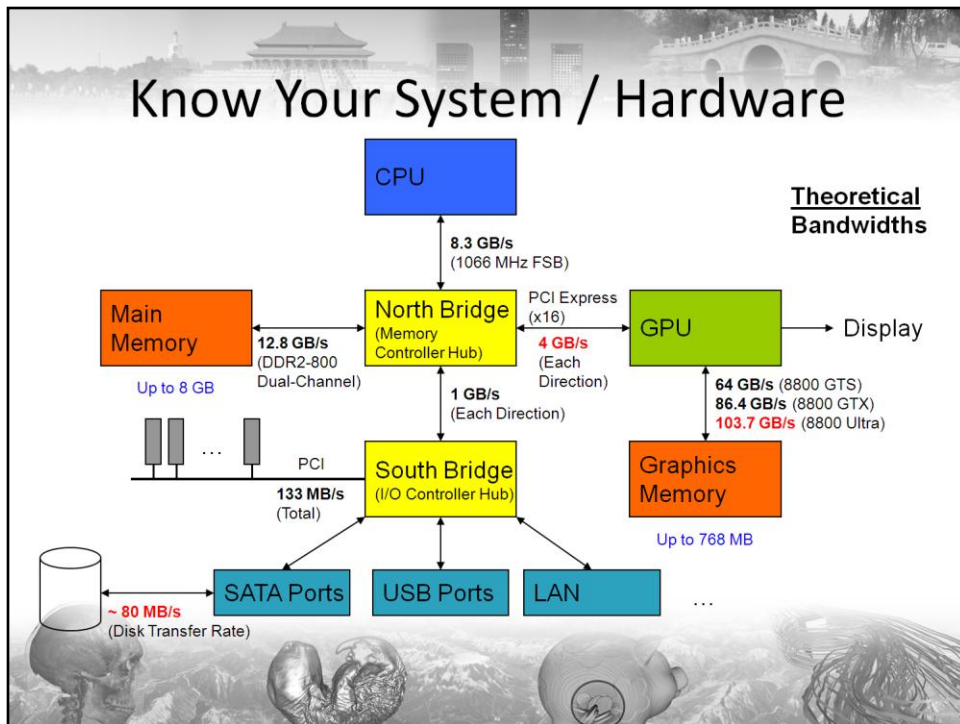
As a last data type we will show some Gigapixel images. Here we just demonstrate the range of zoom-in possible on this 4.5 Gigapixel image kindly provided by J. Kopf and O. Deussen.



# Large Data – Road Map

1. What are we dealing with?
- 2. Know your system!**
3. Terrain Data
4. Terashake 2.1 Simulation
5. Point Clouds
6. Scalar Volumes
7. Gigapixel Images
8. Conclusions & Remarks

In order to generate all these pretty pictures, we first have to deal with the average desktop system's ugly depths. Knowing your system is the key to design efficient pre-processing methods. Most people regard pre-processing time as expendable (and we did, too, before encountering the data sets presented in the following). But a clever pre-processing strategy can easily mean the difference between waiting days or waiting weeks. If you're under pressure, because data is coming in at a high rate (as terrain data was in our lab some months ago), you will want to see your data immediately. Thus, pre-processing can also mean the difference between buying a moderately equipped desktop machine at 4KU\$ or a highend workstation at 50KU\$.



In order to successfully deal with data larger than your main memory, you will have to know your hardware well. Especially the knowledge about the bandwidths, latencies, and capacities of the various components in your system are inexpressible.

While most people know these properties, the good news is that it typically needs computer scientists or engineers to understand and react to the implications of these numbers. That basically means that we all are inexpressible, too, and that we are hopefully facing well-paid jobs.

The basic von Neumann architecture connects the various system components by busses, some of which have remained not quite fast enough through the last four or so decades. Therefore, memory hierarchies have been introduced, consisting of external storage, main memory, and caches. Throwing in a GPU and talking about interactive visualization, we have another layer on top, namely the GPU memory.

# Know Your System / Hardware

- Here specifically MS Windows
  - Disk page sizes
  - Memory mapped files
  - Amount of available physical RAM
  - Type of OS (Use 64Bit OS to allocate >2GB!)
- CPU
  - Partitioned data is “embarrassingly parallel”
  - Benefit of multi-core technology!

Furthermore, specific knowledge about the OS is fairly useful. Here we will discuss numbers based on MS Windows, but other systems such as Linux or MacOS are fairly similar. Since we will have to process most data out-of-core, we need to know the OS disk page size. We will also discuss memory mapped files as a convenient and quick way to load and stream data. Last but not least, we need to know the available amount of physical memory. While modern OSes do everything to hide that number from the casual user, you definitely do not want to over-allocate, since the OS paging data other than your own will vastly diminish performance. As a side note, you should be using a 64Bit OS if possible in any way, since it exposes all your memory to your app instead of 2-3GB.

If you can partition your data, do so! Partitioned data is „embarrassingly parallel“ and can benefit of multi-core technology without much ado. Often it's a matter of using some library implementing a parallel-do and your application runs 3.5 times faster! This might well cut your data processing times from weeks to a few days.

Now we always argued that these times are a pre-process and hence time did not matter. However, if you are given large data on a weekly basis, you will want to see it as quickly as possible. Often, weeks of processing on a single machine are not acceptable, so you argue that in theory you could throw in more machines. However, rumor has it that there are still some customers out there who do not have access to the same amount of hardware you have. On the other hand most of them will have multi-core / hyperthreaded machines (or they already have), so why not use it?

# Know Your System / Hardware

- Determining the page size (Windows)

```
#include <windows.h>
```

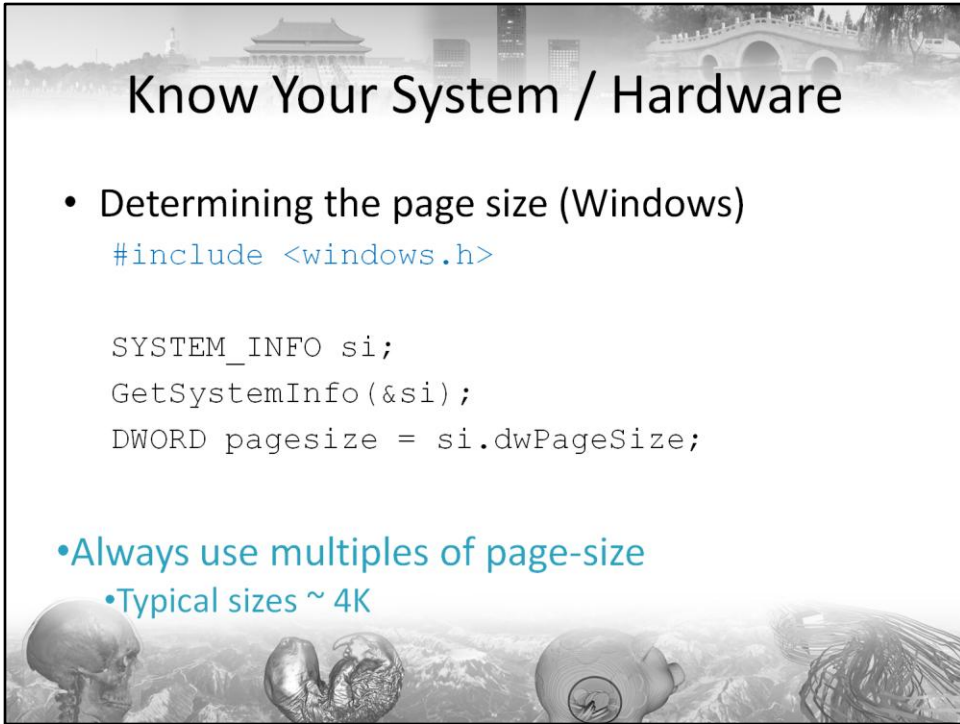
```
SYSTEM_INFO si;
```

```
GetSystemInfo(&si);
```

```
DWORD pagesize = si.dwPageSize;
```

- Always use multiples of page-size

- Typical sizes ~ 4K



Here's an example of how to reliably determine the page size under windows – for completeness sake.

# Know Your System / Hardware

- Determining amt. of physical RAM (Windows)

```
#include<windows.h>
```

```
MEMORYSTATUS memInfo;
```

```
memInfo.dwLength = sizeof(memInfo);
```

```
GlobalMemoryStatus(&memInfo);
```

```
size_t availBytes = memInfo.dwAvailPhys;
```

- Allocate 60% - 75% at initialization

- Dynamic allocation / release affects performance!



Here's a code snippet for determining the amount of available physical memory. Be sure to check out the MEMORYSTATUS struct, if you haven't already – there are lots of useful things in there.



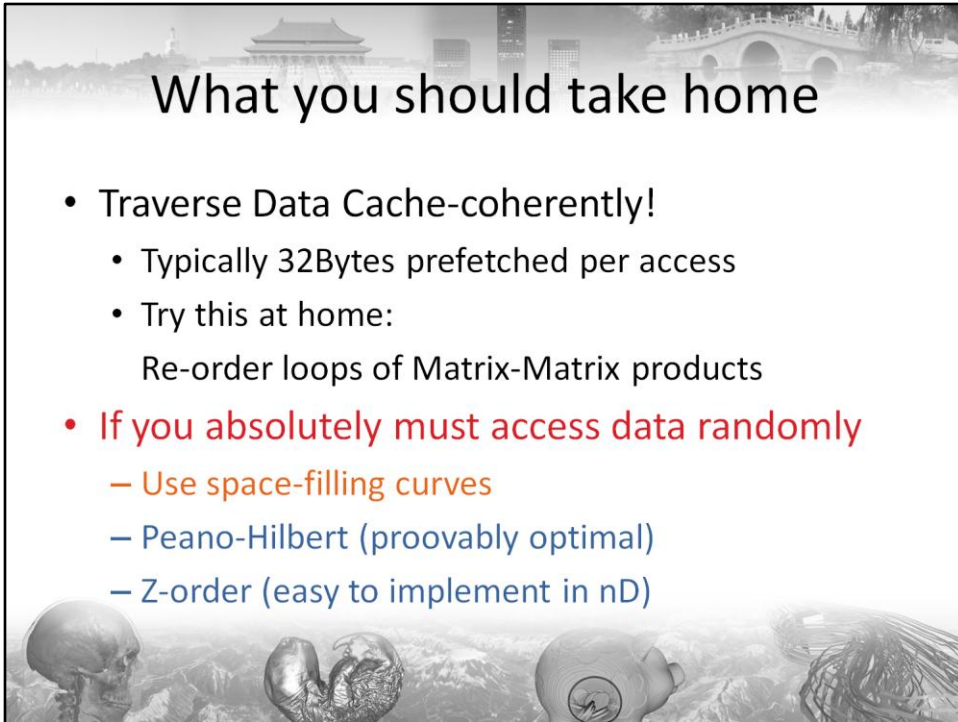
## Know Your System / Hardware

|              | Latency         | Bandwidth   |      |
|--------------|-----------------|-------------|------|
| GPU Memory   | „it depends“    | ~ 110 GB/s  |      |
| CPU L1 Cache | 1.25 ns (3 cl.) | ~ 200 GB/s  |      |
| CPU L2 Cache | 7.9 ns (19 cl.) | ~ 60 GB/s   |      |
| Main Memory  | 124 ns          | < 12.5 GB/s | 100x |
| HDD to CPU   | ~ 13 ms         | ~ 80 MB/s   | 170x |
| SSD to CPU   | ~ 75 $\mu$ s    | ~ 140 MB/s  |      |

**Typical peak performances (Q6600QC 2.4GHz, 8800GTX)**


In this diagram we list typical peak bandwidths and „peak“ latencies – „peak“ of course meaning „best-possible“. Note that the cache is typically two orders of magnitude faster than „normal“ memory access, and that average cached RAM access is again several orders of magnitude faster than HDD access. This is somewhat alleviated by the recent improvement of SSDs – but they will cost you! Still, RAM is more than 300x faster in terms of latency than is SSD access!





# What you should take home

- Traverse Data Cache-coherently!
  - Typically 32Bytes prefetched per access
  - Try this at home:  
Re-order loops of Matrix-Matrix products
- If you absolutely must access data randomly
  - Use space-filling curves
  - Peano-Hilbert (provably optimal)
  - Z-order (easy to implement in nD)



This said, cache coherent traversal is mandatory. Do everything possible to ensure that as much data as possible resides in cache.

- Use compression to squeeze more data into the caches
- Use space-filling curves to make pseudo-random accesses more coherent

While the Peano-Hilbert curve is provably optimal for truly random access patterns, it is a pain to implement in any dimension greater than 2. Instead, use the Z-order curve. This can be conveniently implemented using a bit-shuffle in logarithmic time:

Let your „real-world“ index be  $X := (X_1, X_2, \dots, X_n)$ . First pad the components of  $X$  by leading 0s such that all indices required in your operation are of the same bit length. Then simply „shuffle“ them as follows. Let  $B_1(X_i)$  be the first (left-most) bit of index  $X_i$ ,  $B_j(X_i)$  the  $j$ th bit of  $X_i$  (again from left). Then the Z-order index is

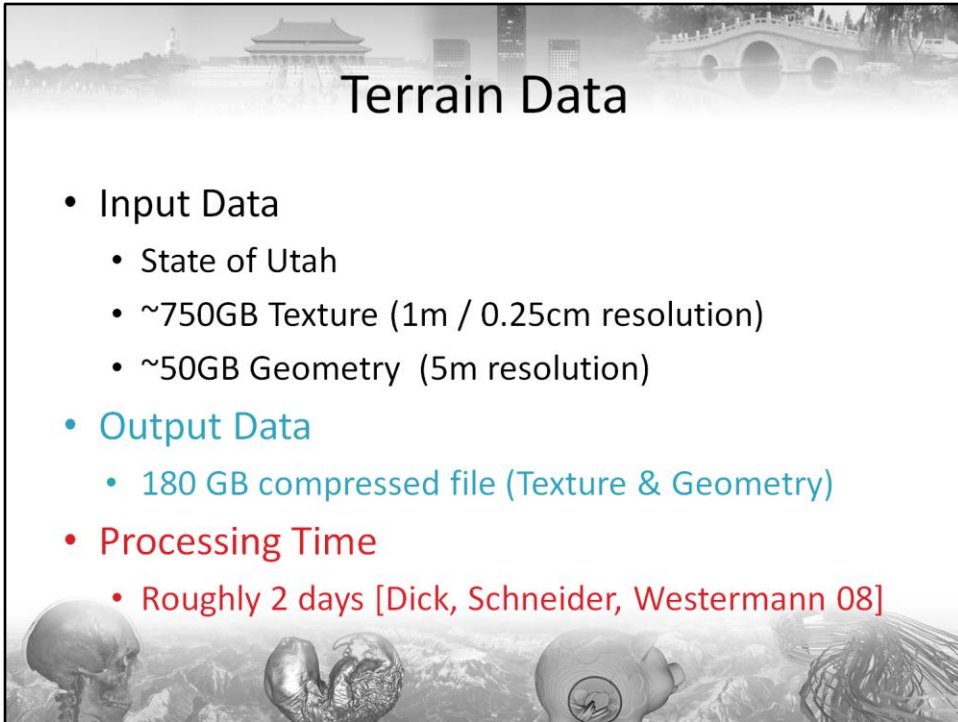
$Z = B_1(X_1).B_1(X_2) \dots B_1(X_n).B_2(X_1).B_2(X_2) \dots B_2(X_n) \dots B_m(X_n)$ , where  $m$  is the number of bits used for each index. „.“ denotes concatenation, here. To obtain the index vector  $X$  of any Z-order index  $Z$ , simply „unshuffle“ the bits.



# Large Data – Road Map


1. What are we dealing with?
2. Know your system!
- 3. Terrain Data**
4. Terashake 2.1 Simulation
5. Point Clouds
6. Scalar Volumes
7. Gigapixel Images
8. Conclusions & Remarks

The first real example is terrain data.



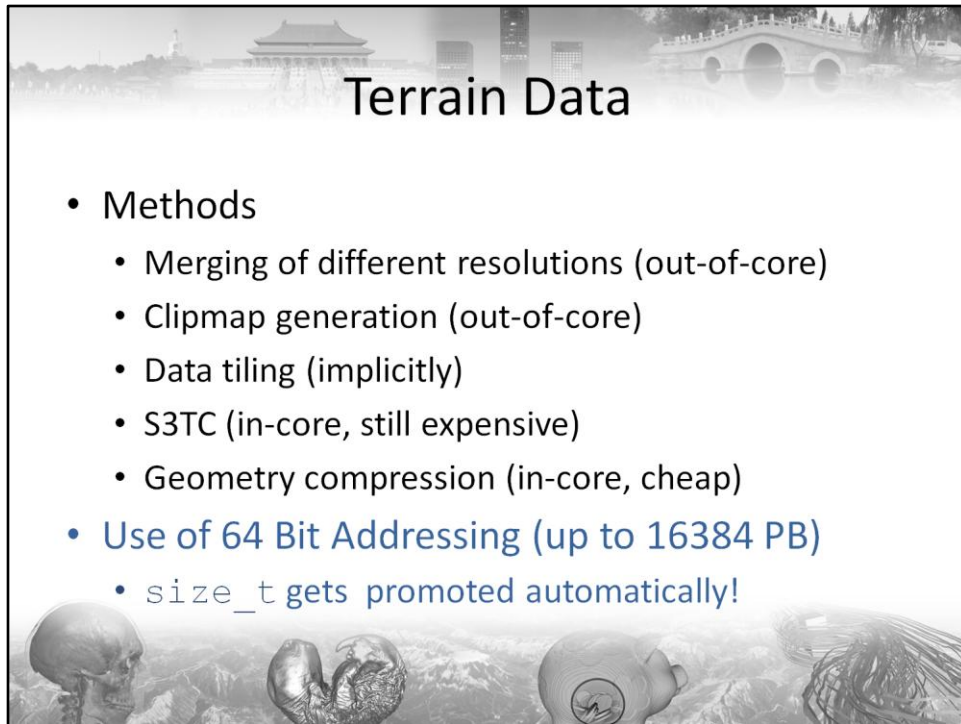
# Terrain Data

- Input Data
  - State of Utah
  - ~750GB Texture (1m / 0.25cm resolution)
  - ~50GB Geometry (5m resolution)
- Output Data
  - 180 GB compressed file (Texture & Geometry)
- Processing Time
  - Roughly 2 days [Dick, Schneider, Westermann 08]



When I showed the Kennecott Copper Mine I mentioned that this is part of a larger data set. The State of Utah makes its geospatial data publicly available, including an ortho-rectified texture. This texture is the major difference to what the USGS offers for the entire USA – and the resolution. If you get all the data available, it will be about 800GB of texture and geometry, with texture dominating the total size – as of now. In my personal opinion we will soon see geometry as fine-grained as 1m in large areas and textures with about 0.25cm spacing. The amount of data will then be gigantic – think of several TBs of data, not to mention additional geospatial information.

We compressed this data down to 180GB (will be explained in detail later on and by Christian), and the processing took roughly 2 days. Again, this time was mainly spent on texture compression (meshing and geometry encoding is actually a matter of a few minutes). If we had not used Quad-core parallelization, this would have cost us 8 days! Speeding up this process is especially critical during development – you simply cannot afford to wait days before you see that something went terribly wrong.



Here's a quick overview of the methods employed. First, we merge different geometric resolutions using a scattered data interpolation scheme – if such resolutions were provided. After that, we generate a single, huge clipmap of both the geometry and the texture. This has to be done out-of-core (as well as the merging step), and whether or not implementing this is painful depends on your ability to abstract from the actual amount of data by automatically paging data structures. Rather invest a few hours or days more into your data structures and make it scale instead of keeping re-writing it whenever the next larger chunk of data arrives.

Data tiling is not much to talk about, tiling is done implicitly. These (in our case 512x512) data chunks are then S3TC encoded – use the Squish Library of Simon Brown or NVIDIA's CUDA Library to do so. Parallelize this with the amount of cores in your system. This is really easy. Geometry compression is very cheap in comparison, because it does not involve any optimization process.

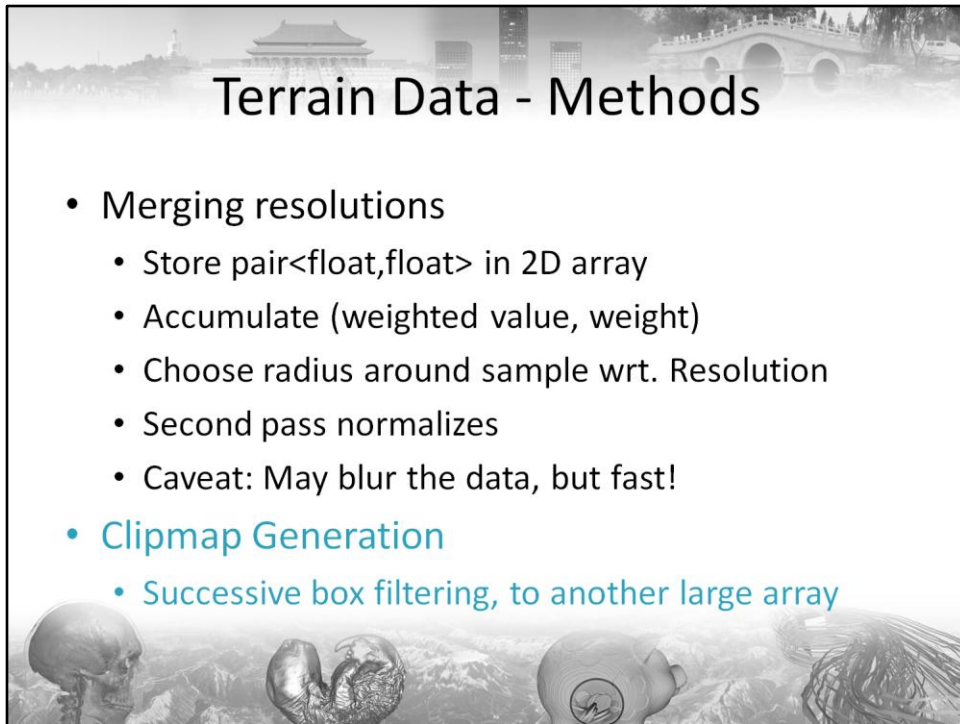
As a side note, use 64 Bit addressing, it really makes life easier. Also note that the `size_t` data type gets promoted automatically to 64bit for many different compilers. Use this data type to make your code a bit more scalable wrt. 32/64bit OSes – but be careful, it's still an unsigned ;)



## Terrain Data - Methods

- Tiles may come in different resolutions
  - Need uniform resolution for rendering
  - Use splatting to combine them
- Large 2D Array abstraction
  - Behave like C-Arrays
  - Memory mapped file wrapper
  - Process data logical-page-wise! (here: 4MB)
  - Code: <http://wwwwcg.in.tum.de/Tutorials>

The abstraction layer we implemented to handle large 2D data pages automatically and uses multiple threads to perform pre-fetching (which, honestly, did not provide the expected speed-up, but improved performance just a little bit.) These arrays use Z-order data layout on disk and may behave just like 2D C-Arrays. But even if they do, process your data page-wise! After experimenting a bit, we found that using a logical page size of 4MB offers a good trade-off between disk latency to position the next page and the disk bandwidth to retrieve the next page. Code is available on the tutorial home page.



These large data arrays are implemented as templates, so you can store inside whatever you want. For instance a `std::pair` of floats during the merging step of the pre-processing.

Clipmap generation is implemented simply as successive box filtering, but you may choose your downsampling algorithm of choice, here.



# Terrain Data - Methods

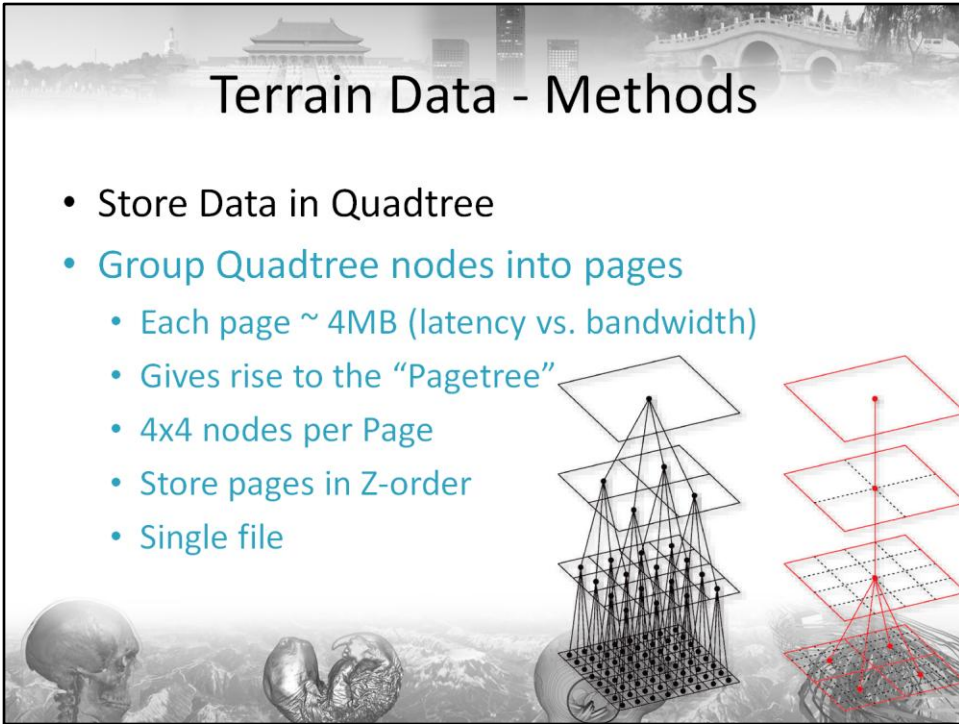
- Geometry Compression: Christian Dick
- Data Tiling
  - Gather full  $512^2$  mipmaps from various files
- S3 Texture Compression
  - About 0.5s for  $512^2$  [Simon Brown's Squish Lib]
  - Good speed-up on multi-cores (3.7x on 4 cores)
  - Reason: Almost sync-free
  - Recommendation: Use intel's TBB Library

Geometry compression will be discussed by my colleague Christian Dick. As already mentioned, S3TC is the real bottleneck during compression. We implemented a threaded encoder the „hard way“, using only WIN32 calls. From this experience we really recommend using some library that conveniently encapsulates the parallelisation – and prevents you from experiencing the abysmal „fun“ of thread debugging . Intel just recently presented their „Threading Bulding Blocks“, which comes in a neat, STL-like design, scales well with the amounts of cores, and costs absolutely nothing. How great is that?



# Terrain Data - Methods

- Store Data in Quadtree
- Group Quadtree nodes into pages
  - Each page ~ 4MB (latency vs. bandwidth)
  - Gives rise to the “Pagetree”
  - 4x4 nodes per Page
  - Store pages in Z-order
  - Single file



Data is naturally stored in a quadtree, where nodes correspond to 512x512 samples of our multi-res hierarchy. To avoid latencies arising from disk seek times, we further group 4x4 nodes into a page. These pages are approximately 4MB each, and pages are always loaded contiguously. They are stored in Z-order in a single file on the disk (geometry and texture). Thereby we avoid managing multiple handles and ensure that seek times are minimal.



# Terrain Data - Methods

- CPU Memory Management
  - Allocate big chunk of memory
  - Buddy system [Knuth 97]
  - Page in data on request (and send to GPU)
  - Circular prefetching [Ng et al 05]
- GPU Memory Management
  - Buddy system [Knuth 97]
  - LRU strategy



To avoid frequent re-allocation – a process that is quite expensive on most Desktop OSes, we allocate a single, big chunk of main memory and manage this chunk ourselves. This requires to hold an index structure. To avoid internal fragmentation, we utilize a buddy system. If a new page is to be loaded, we first check if a chunk is available into which the page tightly fits (i.e.  $\text{chunksize} > \text{pagesize}$  but  $2 * \text{pagesize} < \text{chunksize}$ ). Then, of all these pages, we choose the least recently used to be purged.

Circular prefetching is used to hide seek latencies from the user and to allow for rapid shoulder views. Note that the amount of data to be streamed is only slightly higher than with the more commonly employed fan-regions, albeit at the cost of an increased total memory usage (about a factor 2-3).

The GPU memory system works in the exact same manner for geometry buffers. The texture tiles used in our system are all of the same size, making managing them a rather trivial task. Here, a LRU strategy is also employed.

- C.-M. Ng, C.-T. Nguyen, D.-N. Tran, T.-S. Tan, S.-W. Yeow, „Analyzing pre-fetching in large-scale visual simulation“, Proc. Computer Graphics International, pp. 100—107, 2005
- D. E. Knuth, „Fundamental Algorithms, The Art of Computer Programming“, 3rd Edition, Addison-Wesley, 1997



## Large Data – Road Map

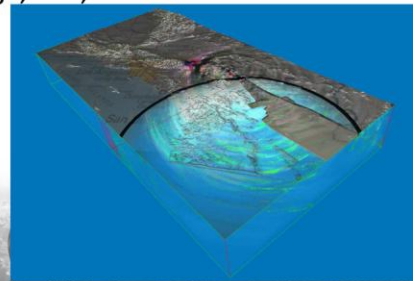
1. What are we dealing with?
2. Know your system!
3. Terrain Data
- 4. Terashake 2.1 Simulation**
5. Point Clouds
6. Scalar Volumes
7. Gigapixel Images
8. Conclusions & Remarks

Now to something completely different – the visualization of the Terashake 2.1 Simulation that was conducted at the San Diego Super Computing Centre.

# Terashake 2.1

- Simulation performed by SDSC
  - About 20GB per time step
  - Hundreds of time steps
  - “Challenge” of the IEEE Vis Contest 2006
  - Originating at Palm Springs, CA, USA
  - 3D Displacements
  - “Basins” in scalar volume

<http://wwwcg.in.tum.de/Research/Projects/vis-contest06>



The Terashake 2.1 Simulation comprises data of a magnitude 7.7 earth quake simulation. The grid is finely resolved (200m spacing) and covers the area from Palm Springs to Los Angeles and beyond. This data is huge! The original resolution comes in 20GB per time step chunks, and there are several hundreds of time steps. Participants of the IEEE Visualization Contest 2007 were given a still remarkable amount of 70GB of data. The original data was reduced using frame skipping and resampling, but it was still enough to pose serious trouble. In this data set, you can see the earth quake originating at Palm Springs, then following the St. Andreas fault to Los Angeles, where it keeps rumbling for a significant period of time (and with still remarkable strength). Participants were given a set of questions from domain experts and an additional scalar data set that contains measured wave phase speeds for the ground. By reconstructing an isosurface in this scalar data set (participants were told the proper iso-value) one can reconstruct so-called basins – geological structures at whose boundaries the waves of the earthquake react in various manners. These basins are the cause for the fact that an earth quake originating in Palm Springs is a serious threat for Los Angeles’ residents.

# Terashake 2.1 - Methods

- First insight:
  - Data is massive! In fact too large to stream...
  - Subsampling. We know it's evil, but:
    - Data becomes a lot smaller
    - Interactive preview possible
- Streaming [Bürger et al 07]
  - Asynchronous GPU ring buffer
  - Hides latency from user



The first insight is that you cannot possibly stream the original 20GB of data per time step during an interactive visualization. Albeit evil by itself, downsampling is a viable method to reduce the data before the visualization. Once done, we used the streaming approach described in Bürger et al 2007. This approach implements a ring buffer on the GPU to hide latencies from the user. It tries to prefetch as many time steps as possible, and purges each time step as soon as it is no longer needed for the visualization. Furthermore, it also performs a prefetching to core memory in order to smooth out occasional disk-“hiccups”. The picture in this slide shows the mechanism. While rendering from times  $t_0, t_1$ , additional steps are loaded to the GPU asynchronously.

- K. Bürger, J. Schneider, P. Kondratieva, J. Krüger, R. Westermann, „Interactive Visual Exploration of Unsteady 3D Flows“, EG/IEEE-VGTC Symposium on Visualization, 2007

## Terashake 2.1 - Methods

- Visualization requires 4D interpolation
  - Just a matter of timing:
    - Always have at least two timesteps resident
    - Fetch data w/ trilinear interpolation
    - Estimate loading time for next step
    - Gives 4th interpolation weight (Shader)
  - Visualization
    - Particle tracing very powerful...
    - Though advection non-standard

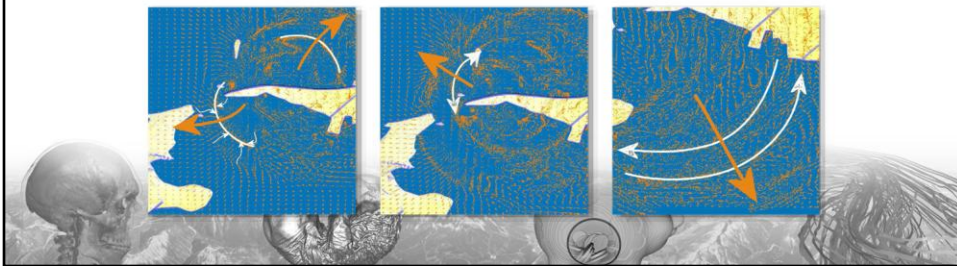


To visualize time-dependent data, a 4D-interpolation is required. In terms of interactive visualization this boils down to quadri-linear tensor product interpolation. While tri-linear interpolation can be done in hardware, interpolation along the time-axis can generally not be performed in hardware. We tried to estimate the time needed to load another time slice and use the time index relative to the point where the last step became resident as an interpolation weight. Then, two tri-linearly fetched vectors were interpolated in the Shader.

Although a particle-based visualization is physically not meaningful (since you simply cannot advect through displacements) it proved to be extremely insightful. This is due to the fact that during the advection displacements are accumulated and very characteristic wave patterns form in the particles.


## Terashake 2.1 - Methods

- Lessons learned
  - Particles offer good/intuitive insights
  - Even if physically not visualization of choice
  - Fast disks are a good thing
  - DirectX 10 asynchronous mechanisms tricky



These characteristic patterns are shown in the three figures. Features to look out for are the general wave front propagation direction (orange) that can be tracked by local particle density maxima. The secondary feature is the oscillation of particles (white) relative to these peaks. There are two kinds of oscillations, one with low amplitude (middle) and one with high amplitude (right). The third type of wave present in earthquakes can be diagnosed by particles moving from both sides to form a local density maximum (left). Actually, there is a fourth wave type, which cannot easily be determined using particles, but three out of four can!

Among the lessons learned was that particles offer intuitive insights (to the surprise of domain experts), even if the visualization is not physically permitted. Another thing we learned was that fast, cheap hard disks are one of the major benefits of our time. Spend a few bucks more for speed, whenever you can, or use RAID's – it is worth the effort. As a side note, DirectX 10's asynchronous mechanisms are somewhat tricky, as even with ThreadPumps it is not always guaranteed that buffers are resident when the upload call returns. They can as well be batched, and dedicated asynchronous mechanisms as present in OpenGL are not really available for DirectX. So getting the upload „hiccup“-free might require some skillful implementation – but recent driver releases show significant improvements in these issues.



## Large Data – Road Map

1. What are we dealing with?
2. Know your system!
3. Terrain Data
4. Terashake 2.1 Simulation
- 5. Point Clouds**
6. Scalar Volumes
7. Gigapixel Images
8. Conclusions & Remarks

Now to point clouds. By point clouds we mean point-sampled surfaces (presumably 2-manifold), not dense sets of points as they arise in physics or numerical simulation. The reason is that we're still working on that latter type of data and we're trying to figure out how to deal with them most efficiently.



## Point Clouds - Methods

- Resampling (MLS surface)
  - Map to local heightfields [Ochotta and Saupe 04]
  - Use 2D Tensorproduct Wavelet Encoding
  - Advantage: General Coding, good bit rate (2.3bpp)
  - Disadvantage: Resampling surface expensive



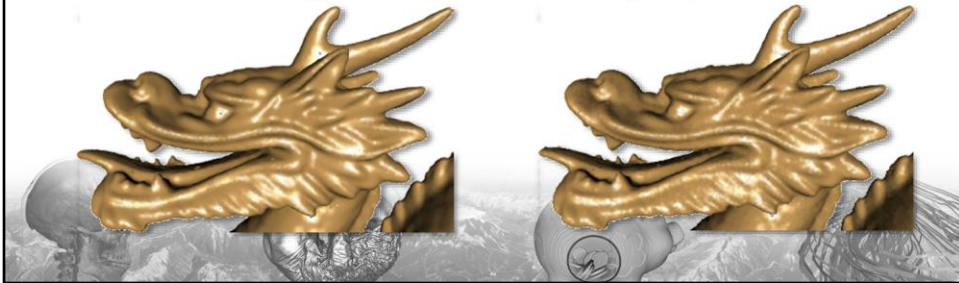
The point cloud is reconstructed using a Moving Least Squares Surface and is projected into a local tangent plane. In this plane, it is regularly resampled to yield a height field. These height fields are then encoded using standard 2D tensorproduct wavelet approaches (using SPIHT as a backend). The advantages of this method are the fairly general encoding process and the good bitrate of only about 2.3 bpp. (points need to be stored, only). However, finding the local planes and resampling the points into it require spatially coherent portions of the point cloud to be resident in main memory. Also, this process is fairly costly. The decoding cannot be performed trivially on the GPU, as SPIHT encoded wavelets do not offer trivial random access, but rather an  $O(\log N)$  process is required to decode a single point. It could be suitable, though, to encode one height field after another. Another more subtle drawback is that due to the use of tangent planes the resampling resolution varies across the mesh and might lead to artifacts.

- T. Ochotta and D. Saupe, „Compression of point based 3D models by shape-adaptive wavelet coding of multi-height fields“, Eurographics Symposium on Point Based Graphics, 2004



# Point Clouds - Methods

- Multi-resolution decomposition
  - Iterative Clustering [Waschbüsch et al 04]
  - Predictive Differential Encoding
  - Advantage: Bitrates  $\sim 2\text{-}6\text{bit} + \sim 6\text{-}13\text{bit}$  for normal
  - Disadvantage: Iterative Clustering expensive

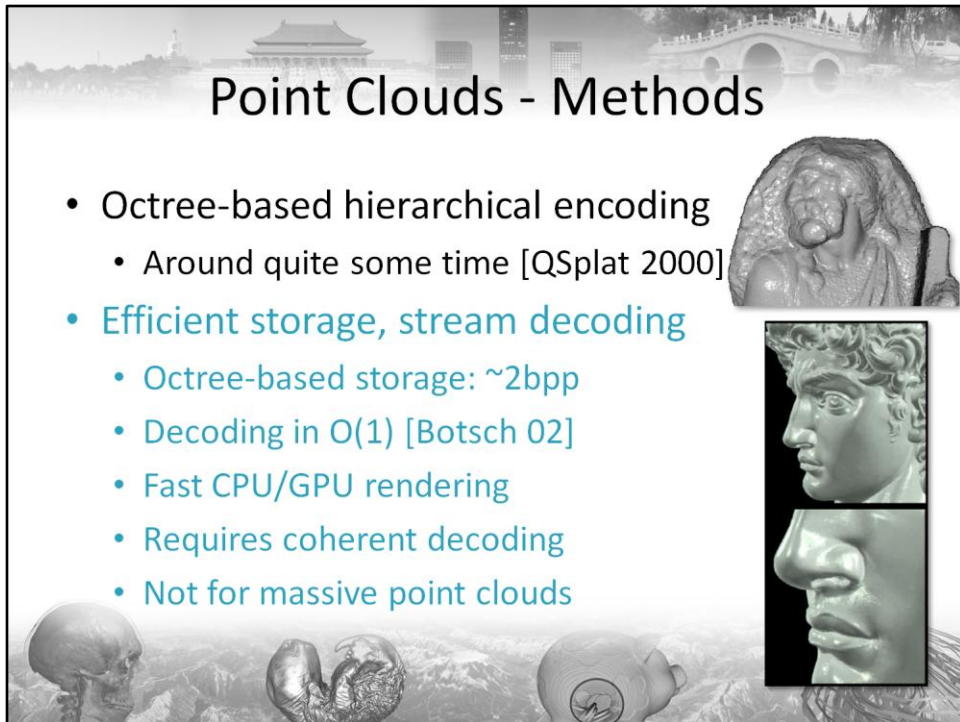


The method proposed by Waschbüsch et al. consists of iterative clustering to obtain a multi-resolution point cloud. During this step, the hierarchy is built bottom up by successive point contractions. Bitrates are largely dependent on the smoothness of the surface implied by the point set. For smooth surfaces, bitrates are typically around 2 bit, while normals require another 6 -13 bits (in rare cases up to 17). For the dragon, the left picture shows the original, while the right one was encoded using 6.4bpp. Note that the problem of varying resampling is entirely avoided, although it this method is still very likely to result in very awkward and expensive GPU decompression if one tried so.

- M. Waschbüsch, S. Würmlin, E.C. Lamboray, F. Eberhard, M. Gross, „Progressive Compression of point-sampled models“, Eurographics Symposium on Point Based Graphics, 2004

# Point Clouds - Methods

- Octree-based hierarchical encoding
  - Around quite some time [QSplat 2000]
- Efficient storage, stream decoding
  - Octree-based storage: ~2bpp
  - Decoding in  $O(1)$  [Botsch 02]
  - Fast CPU/GPU rendering
  - Requires coherent decoding
  - Not for massive point clouds



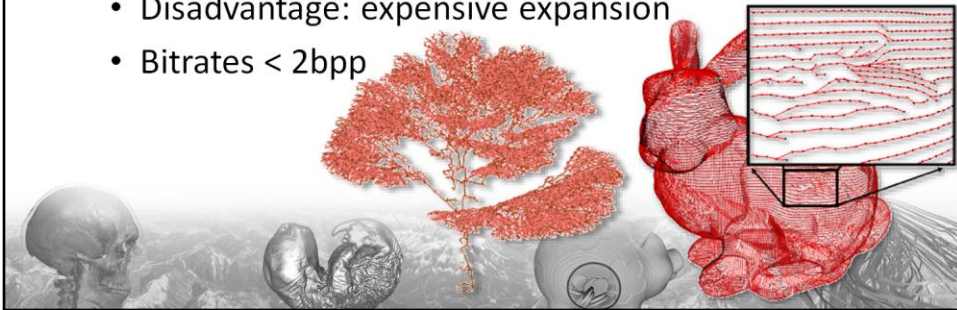
An approach that has been around for quite some while is an octree-based encoding of points. Among the first papers was Rusinkewitz and Levoy's Qsplat approach that showed remarkable results for its time.

Later on, Botsch et al showed that by storing points in an octree, coherent decoding is an  $O(1)$  process, and that 2bpp suffice to store points. Normals, however, require significant more bits. The authors proposed a vector quantizer obtained by successive octahedron refinement – a technique that avoids any codebook storage, but requires slightly more bits than other, LBG-based approaches. All these approaches are not well suited for random access, since the points have to be decoded sequentially, starting with the coarsest resolution. Also, large point sets could pose problems if the data cannot reside in main memory anymore.

- S. Rusinkewitz, M. Levoy, „QSplat: A multi-resolution point rendering system for large meshes“, SIGGRAPH 2000, pp. 343-352
- M. Botsch, A. Wiratanaya, L. Kobbelt, „Efficient High Quality Rendering of Point Sampled Geometry“, EG Workshop on Rendering, 2002

# Point Clouds - Methods

- Coherent path traversal
  - Tree containing each point [Gumhold et al 04]
  - Two prediction methods: constant, linear
  - Graph-theoretical formalisms possible
  - Disadvantage: expensive expansion
  - Bitrates  $< 2\text{bpp}$

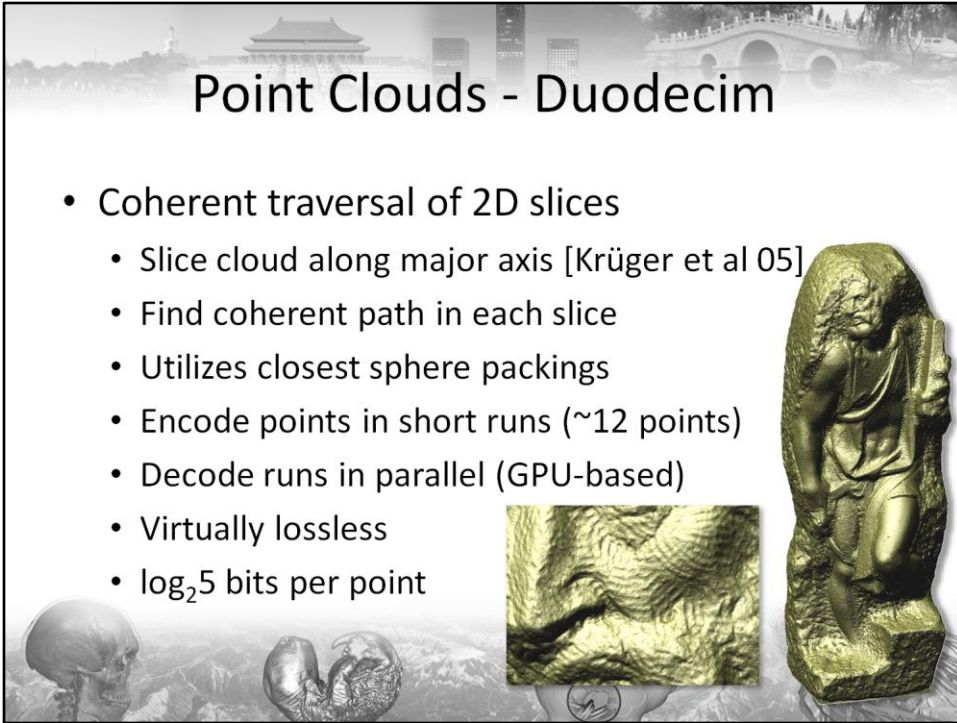


Another method to compress point clouds is to find a coherent path (or, as in this work, a spanning tree) to traverse all points. This allows graph-theoretical formalisms to be employed – which is nice, because they have been studied for decades. Then, successive points along these paths are predicted either constantly or linearly, and the offset between prediction and actual point position is encoded. Although the decompression is somewhat expensive and is likely NOT to be decoded by any of today's GPUs efficiently, the method provides a nice theoretical prediction/correction framework and achieves good bit rates.

•S. Gumhold, Z. Karni, M. Isenburg, H.-P. Seidel, „Predictive Point-Cloud Compression“, SIGGRAPH Sketch, 2004

# Point Clouds - Duodecim

- Coherent traversal of 2D slices
  - Slice cloud along major axis [Krüger et al 05]
  - Find coherent path in each slice
  - Utilizes closest sphere packings
  - Encode points in short runs (~12 points)
  - Decode runs in parallel (GPU-based)
  - Virtually lossless
  - $\log_2 5$  bits per point



In a similar style, we presented the Duodecim approach that is exceptionally well suited for models exhibiting a major axis, such as statues or CT scans. Again, the key insight is that data is highly coherent along spatially coherent paths. Unlike previous work, however, the data is sliced into 2D slabs and points falling into these slabs are inserted into a closest sphere packing grid. We then first compute a spanning tree of adjacent occupied cells and transform this tree (or more precisely the forest of such spanning trees in case of multiple connected components) into one or more paths. Then, these paths are cut into conveniently-sized chunks of about 12 points (depending on the decoding GPU's capabilities). Points along these paths can be encoded differentially by storing a symbol 1..6 denoting the neighboring cell through which the path passes next. If we require the path to leave through a different cell than it entered, we can further reduce the symbols to only 5 choices. Note that this is no restriction, since we already encoded the entering neighbor. Consequently, we can encode each point differentially at  $\log_2(5)$  bits – except for the first one, which is plainly encoded using the respective grid cells 2D index.

Although good fidelity is achieved at very low bitrates, we still resample the data – albeit at a scale comparable to the Nyquist rate of the scan.

• J. Krüger, J. Schneider, R. Westermann, „Duodecim – A Structure for Point Scan Compression and Rendering“, EG Symposium on Point-Based Graphics, 2005

# Point Clouds - Duodecim

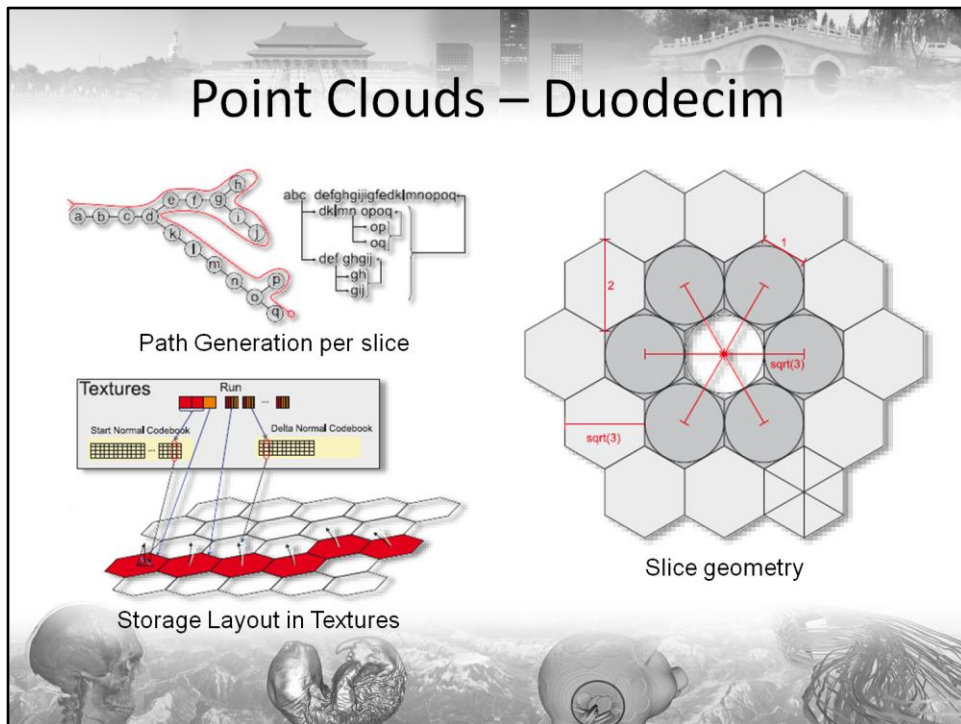
- Sort points along major axis
  - Out-of-core sorting [Sedgewick 98]
- “Slab” points perpendicular to major axis
  - Efficient stream operation on sorted points!
  - Grid points per slab into hexagonal grid
  - Find spanning tree between connected points
  - Convert tree into path
  - Encode differentially



To efficiently slab the data, we perform an out-of-core sorting along the major axis. This is a pre-process that simply permutes indices and vertex positions. Hence, it is lossless with respect to the render-ability and process-ability of the input mesh. The sorted data set can thus replace the original data without any second thoughts.

- R. Sedgewick, „Algorithms in C++ Parts 1—4“, 3rd Edition, Addison-Wesley Longman, Amsterdam, 1998





Here are some figures from the paper demonstrating how to convert the spanning tree into a path, an example of the slice geometry, and the storage layout in textures. For a full description, please consult the paper.

# Point Clouds - Duodecim

- GPU-based decoding
  - Page in required textures (driver-based paging)
  - Decode runs in parallel (requires state!)
  - Write decoded positions & normals to vertex array
  - Render vertex array
  - High Quality rendering [Botsch03, Krüger06]
- LOD based on multires hierarchy
  - Introduces storage overhead



The GPU-based decoder pretty much relies on the driver to provide him with the data necessary to render the model. This can be afforded, since the entire Digital Michelangelo Project (19GB) fit into video memory on modern GPUs (the DMP occupies about 500MB in compressed form). On today's GPUs, rendering is quite straightforward, decoding the short runs in parallel and writing decoded positions to a vertex array. Of course it is even simpler to implement this decoding in the Geometry Shader, but we cannot whole-heartedly recommend doing so with respect to performance.

- M. Botsch, L. Kobbelt, „High-Quality Point-Based Rendering on Modern GPUs“, Pacific Graphics, pp 335-343, 2003
- J. Krüger, J. Schneider, R. Westermann, „Compression and Rendering of Iso-Surfaces and Point Sampled Geometry“, The Visual Computer, Vol. 22, No. 8, August 2006

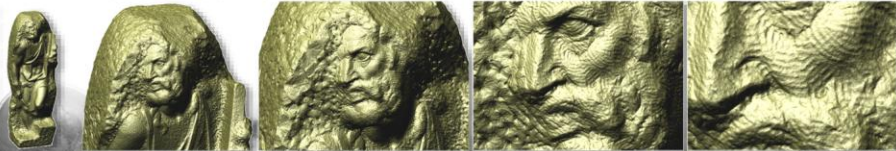
# Point Clouds – Duodecim Roundup

- Compression

- Positions: 2.32 bits
- Normals: ~ 5bits (spherical encoding)
- Colors: 8-11 bits (less coherent than normals)

- Performance

- Entire Michelangelo Project (19GB) fits into GPU memory!
- Sustained frame rates of 30+ Hz.



Here's some statistics on bit rates and performance as well as a zoom on the unfinished Atlas statue. This model is particularly interesting, because:

- It is, unlike Michelangelo's other statues, not polished. Consequently, you can observe the chisel directions and gain some insights into how these statues were made.
- It is large. 255M points to be precise, occupying roughly 10GB as a ply-file.
- Its surface is quite rough, so you can judge the efficiency and fidelity of your compression algorithm very well.

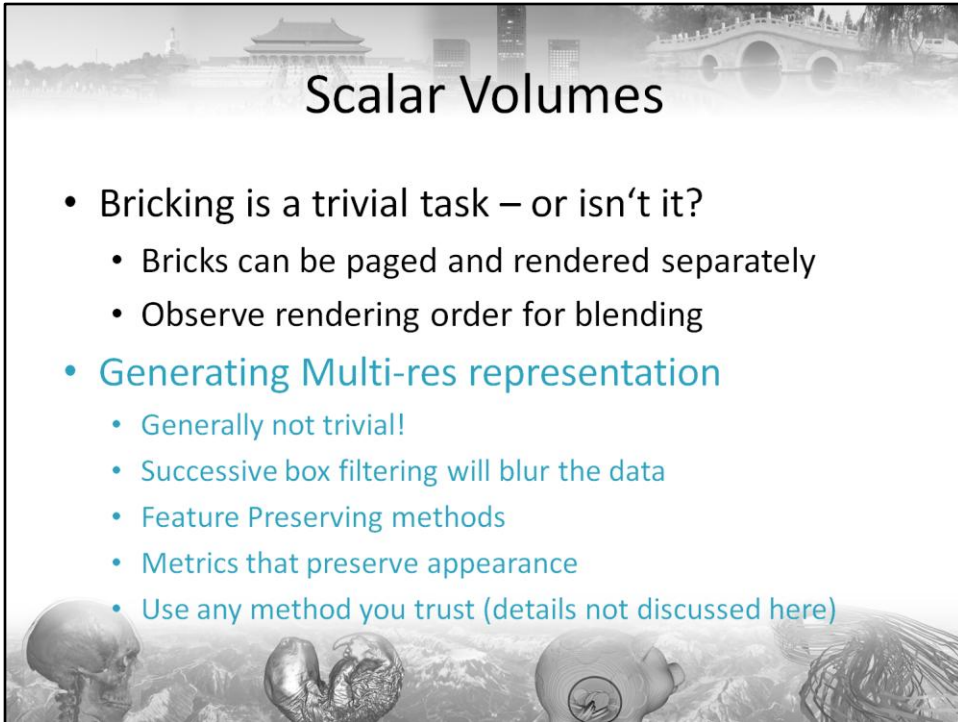




# Large Data – Road Map

1. What are we dealing with?
2. Know your system!
3. Terrain Data
4. Terashake 2.1 Simulation
5. Point Clouds
- 6. Scalar Volumes**
7. Gigapixel Images
8. Conclusions & Remarks

Rendering of Medical Volumes will be presented by Christof Resz-Salama in this tutorial. I will just very briefly discuss bricking and data management.



# Scalar Volumes

- Bricking is a trivial task – or isn't it?
  - Bricks can be paged and rendered separately
  - Observe rendering order for blending
- **Generating Multi-res representation**
  - Generally not trivial!
  - Successive box filtering will blur the data
  - Feature Preserving methods
  - Metrics that preserve appearance
  - Use any method you trust (details not discussed here)

Bricking sounds easy – but this is not always the case! If you want to generate a multi-res representation for your volumes, there is a plethora of literature ranging from box filters, over histogram-based approaches to appearance preserving methods. While we will not discuss them here, be warned that due to the volume rendering integral otherwise reliable techniques to filter data can (and will) fail.

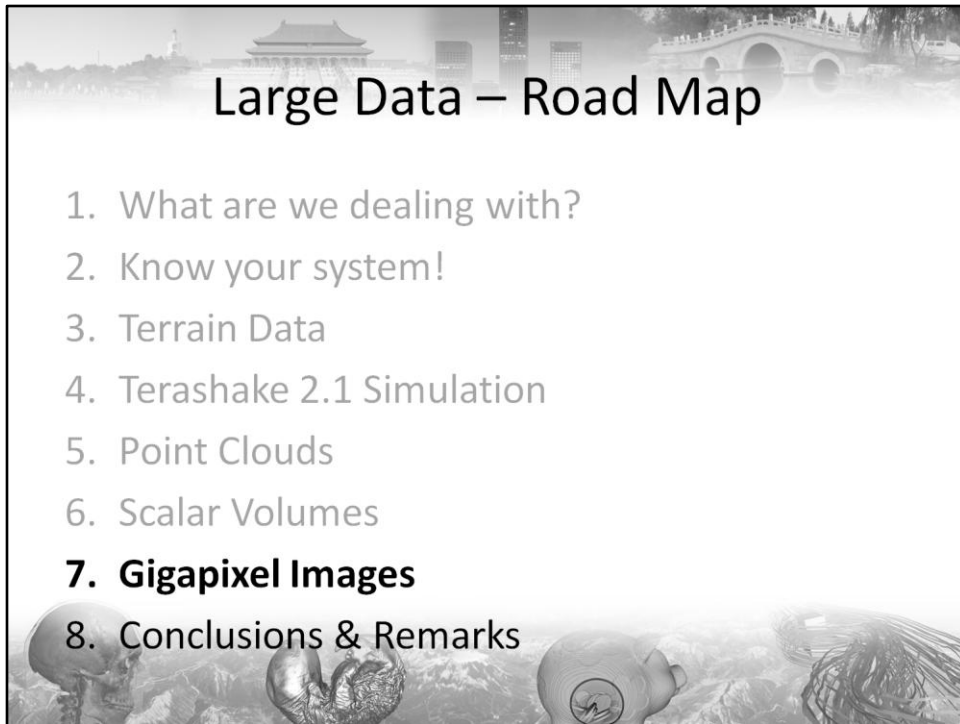
# Scalar Volumes

- Rendering: Christof Rezk-Salama
- Memory Management
  - Example: ImageVis3D, SCl, University of Utah
  - Paging of Bricks using tightest-fit re-use of blocks
  - If one (of many) tightest fitting blocks is found, use MRU (!)
- Rationale behind MRU
  - LRU results in excessive paging if RAM scarce
  - Reason: Entire data set will be paged each frame!



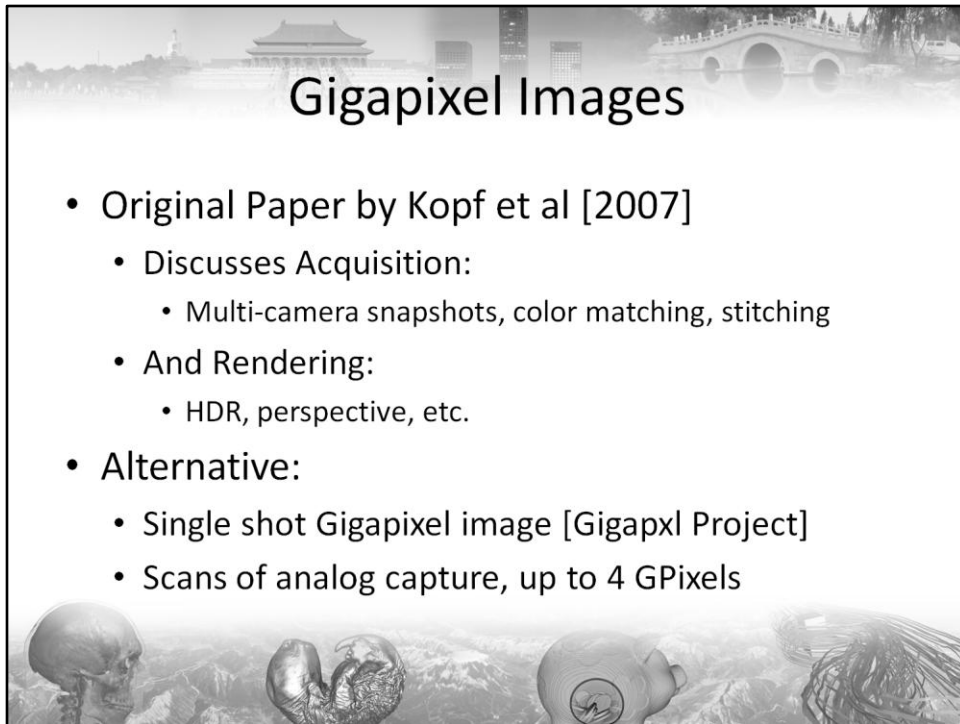
Assuming that you now generated your multi-res representation using a method of choice, memory management is easy – with a quirk!

Assume that you have less memory than your data set would naturally require. Now if you use an LRU paging strategy, you will be constantly paging your entire volume in each frame. If, however, you use an MRU strategy, you will effectively partition your data into a part that is always resident and a part that will always be paged. This is a great advantage when compared to paging all your data, although the choice of MRU might seem counter-intuitive.



Gigapixel Images have become quite a hype recently. I attribute this to their amazing zoom range. On a PC high-resolution display of 2560x1600 (4MPixels), you can zoom in about a factor of 30+ (4.5GPixels) before you finally arrive at the true resolution of the image. By doing so, you will encounter lots and lots of details hidden in these gigantic images.

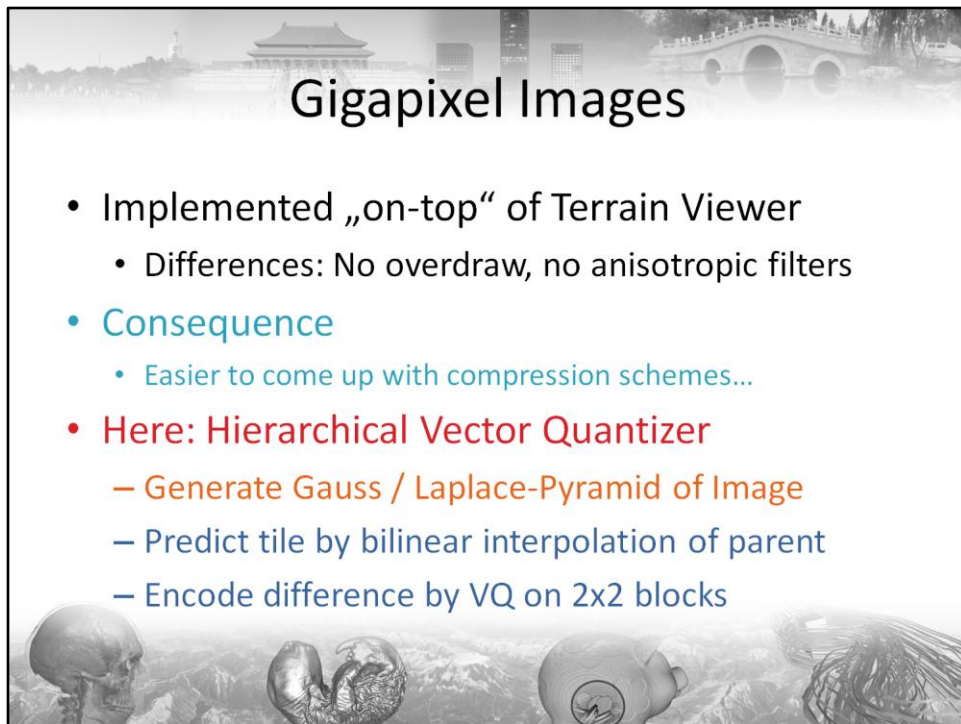
But are they different from terrain data? Well, they do not have geometry, but otherwise... You can in fact use your favorite terrain engine, look straight down from atop and render dummy geometry to view them. But this aside, does this data allow for more efficient methods? Let's have a look at that.



The concept of Gigapixel images was only recently introduced to the graphics community by Kopf et al. Their paper discusses the tiled acquisition using multiple cameras and multiple snapshots. Also, the paper addresses HDR-based rendering and how to use perspective to generate more immersive images.

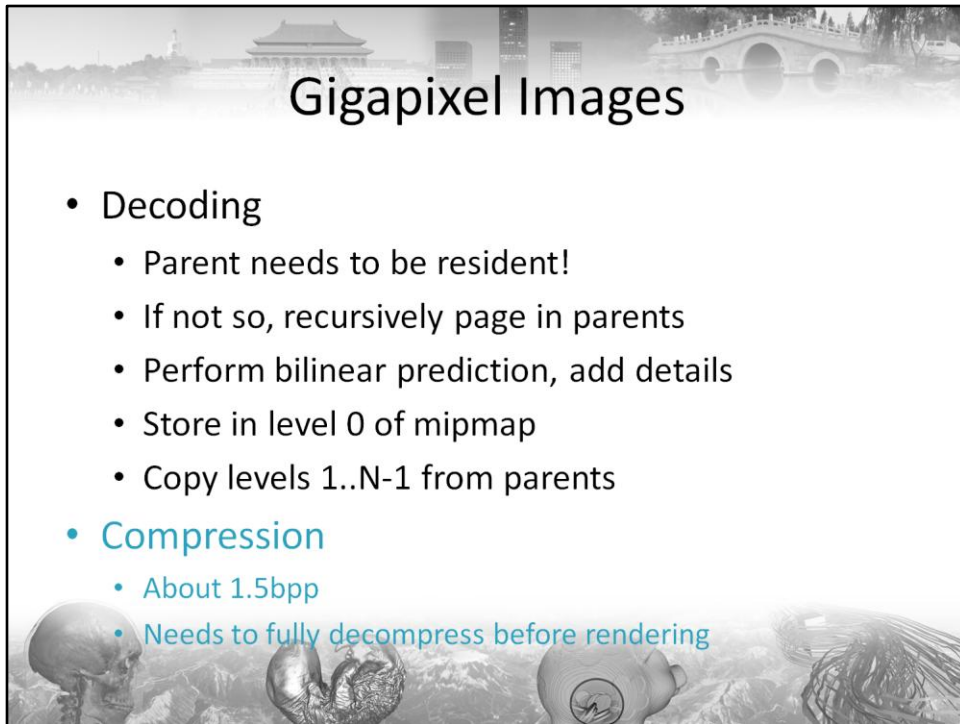
As an alternative we would like to point out that there are single shot Gigapixel images. These are obtained using an analog camera, scanning the analog output. These shots are currently limited somewhere around 4GPixels, but they are still impressive.

- J. Kopf, M. Uyttendaele, O. Deussen, M. Cohen, „Capturing and Viewing Gigapixel Images“, ACM Transactions on Graphics, Vol. 26, No. 3, Art.No. 96, 2007
- [www.gigapxl.org](http://www.gigapxl.org)



As mentioned, we implement our Gigapixel viewer „on-top“ of our terrain viewer. We just lock the camera at straight atop, use an orthographic projection and throw away unnecessary geometry. However, since the camera is locked, you no longer need anisotropic filters, and you do not generate any overdraw any more. Sweet!

While the working set of textures for the terrain viewer corresponds to roughly 350MB uncompressed texture for 4MPixel view ports due to perspective and overdraw, for the Gigapixel viewer this is exactly 12MB + mipmaps. Consequently it does not really hurt to have a texture format from which the GPU cannot immediately render but which allows a significantly more compact representation of the data. Realizing this, we utilized a hierarchical vector quantization scheme based on Gauss / Laplace-Pyramids that uses bilinear-interpolation from coarser levels as a predictor and encodes only the difference. This difference is then blocked into 2x2 pixel groups and encoded using a 12D vector quantizer. At a visual fidelity that is subjectively hard to distinguish from the original photograph, we can encode typical images at around 1.5bpp (including overhead for the mip hierarchy!), thereby rivaling standard jpg compression. Unlike jpg compression, however, we still maintain the ability to decode the image on the GPU.



This involves slight changes to the terrain engine, since we now need to access parents. Consequently, we fetch them recursively and cache them in video memory. This works exceptionally well and requires only minimal bandwidth overhead.

As a side note, by counting the internal nodes vs. the leaf nodes of the quadtree, we can prove that there exists a pre-fetching strategy that has to load at most 1 additional tile for any coherent movement. Implementing this strategy, however, requires a lot more memory in the worst-case, and consequently we live with occasional bandwidth peaks. They do not generally endanger smooth movements.



# Gigapixel Images

- Open Challenges
  - Editing: requires update of pyramid
  - Multi-Layers / Multi-Spectral ?
  - Image Operations [Kazhdan08]



Some of the remaining challenges are clearly editing, new data types such as multi-spectral images (due to issues with the compressor) and image operations.

- M. Kazhdan, H. Hoppe, „Streaming multigrid for gradient-domain operations on large images“, ACM Transactions on Graphics, Vol. 27, No. 3, Art.No. 21, 2008



# Large Data – Road Map

1. What are we dealing with?
2. Know your system!
3. Terrain Data
4. Terashake 2.1 Simulation
5. Point Clouds
6. Scalar Volumes
7. Gigapixel Images
8. Conclusions & Remarks

# Conclusions & Remarks

- Important Messages
  - Know your system!
  - Know your data!
- Type of data allows for shortcuts
  - Images fundamentally different from terrains
- Use custom encoders
  - Hierarchies greatly help in rendering/managing
  - Tiling/Bricking greatly helps in pre-processing





Questions ?

Contact me:

[jens.schneider@in.tum.de](mailto:jens.schneider@in.tum.de)

Visit the webpage:

<http://wwwwcg.in.tum.de/Tutorials>

