Massive Model Visualization using Real-time Ray Tracing

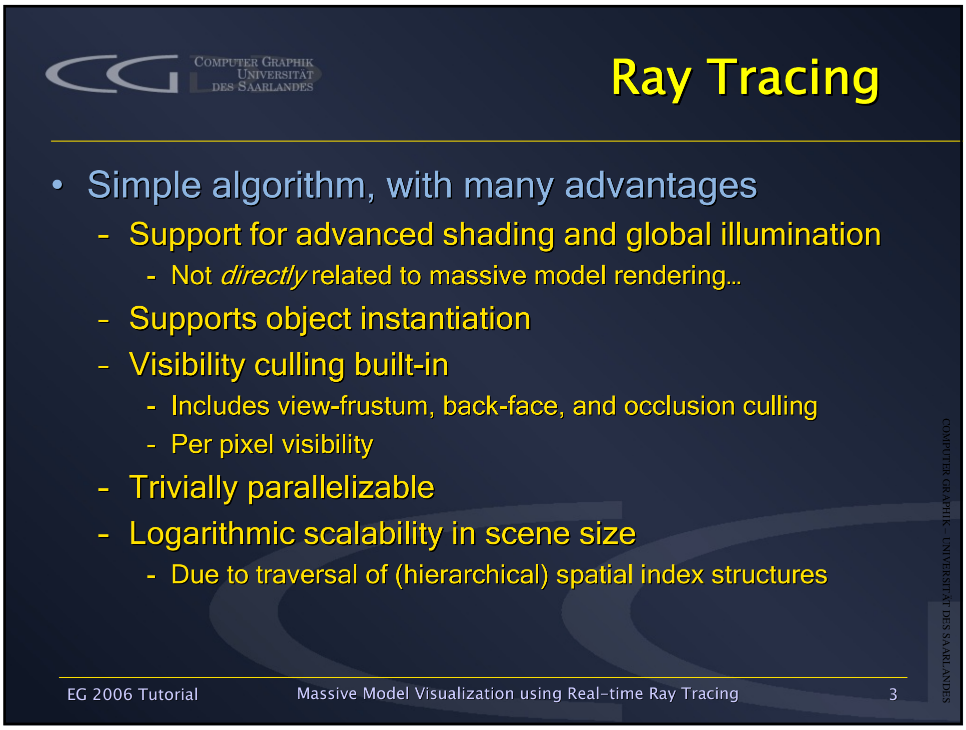
Eurographics 2006 Tutorial:
Real-time Interactive Massive Model Visualization

Andreas Dietrich Philipp Slusallek

Saarland University & inTrace GmbH

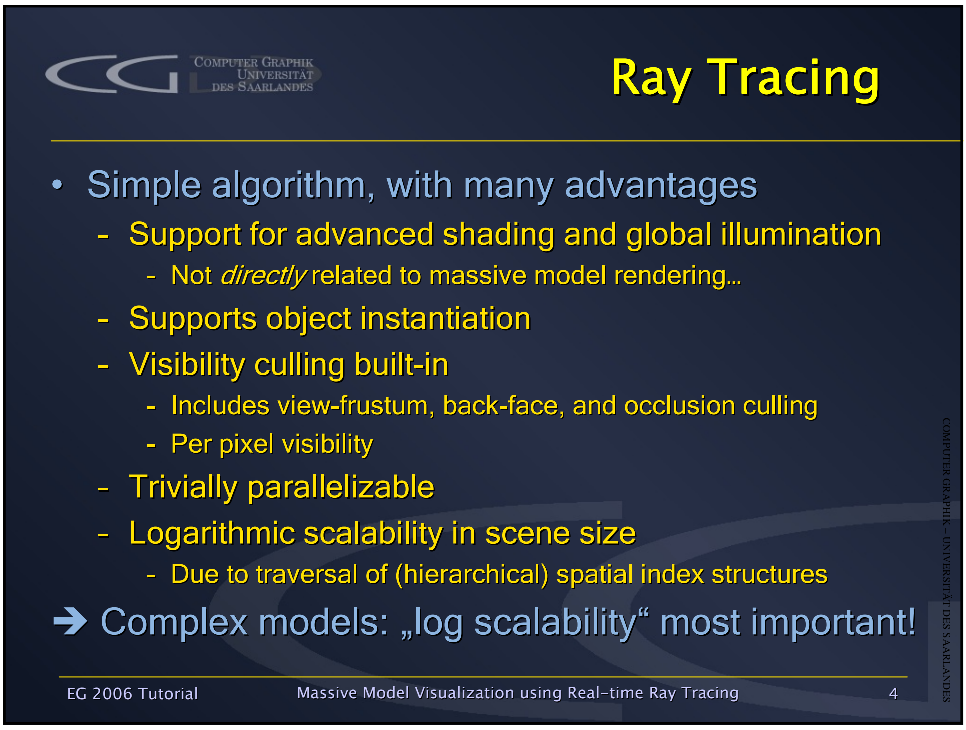
Overview

1. Simplicity of data preparation for ray tracing complex scenes
    - Efficient spatial index structures for ray tracing
    - Off-line construction of index structures
2. Adapting ray tracing to complex models
    - Active memory management
    - Asynchronous data loading
    - Level-of-detail management
3. Photorealistic rendering and lighting of highly complex models
    - Flexible combination and integration of different shading algorithms
    - Efficient integration of environmental lighting
4. Review of hardware-trends for real-time ray tracing
    - Comparing multi-core CPUs, GPUs, Cell processor, and custom ray tracing hardware



Ray Tracing

- Simple algorithm, with many advantages
  - Support for advanced shading and global illumination
    - Not *directly* related to massive model rendering…
  - Supports object instantiation
  - Visibility culling built-in
    - Includes view-frustum, back-face, and occlusion culling
    - Per pixel visibility
  - Trivially parallelizable
  - Logarithmic scalability in scene size
    - Due to traversal of (hierarchical) spatial index structures



Ray Tracing

- Simple algorithm, with many advantages
  - Support for advanced shading and global illumination
    - Not *directly* related to massive model rendering…
  - Supports object instantiation
  - Visibility culling built-in
    - Includes view-frustum, back-face, and occlusion culling
    - Per pixel visibility
  - Trivially parallelizable
  - Logarithmic scalability in scene size
    - Due to traversal of (hierarchical) spatial index structures

➔ Complex models: „log scalability“ most important!

Ray Tracing for Massive Models

- Visibility not the main problem

  - Proof by example: “Forest” scene
    - 1.5 billion triangles
    - Plus shadows, textures, transparency, …
    - Rendered interactively on few PCs

Storage Problem

- Number of visible triangles not main problem

  ➔Main problem: Efficient scene storage and access !
- The storage problem
  - Logarithmic cost: Assumes all data is in memory
  - “Forest” example:
    - Only possible through instantiation → special case !
  - For general complex models usually not the case
    - Boeing 777: 30-40 GB on disk (including spatial index)

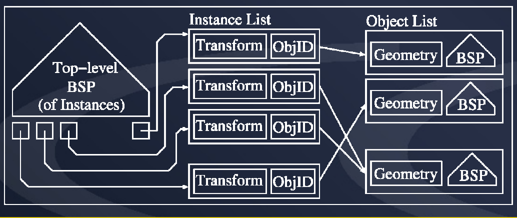
➔Need efficient data handling for preprocessing and rendering

Part I
Simplicity of Data Preparation for Ray Tracing Complex Scenes

---

Spatial Index
Hierarchy and Instancing

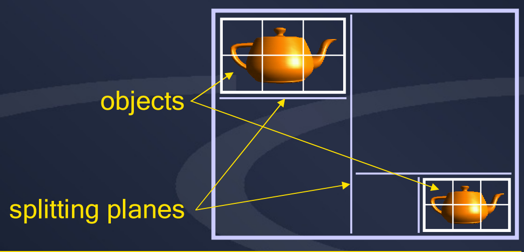
- Two-level k-d tree scheme [Wald et al. 2003]
  - Accelerates ray-object intersection computation
  - Low-level k-d tree for each object type
  - High-level k-d tree organizes instances
    - Object references
    - Object bounding boxes
    - Transformation matrices



Spatial Index

Hierarchy and Instancing

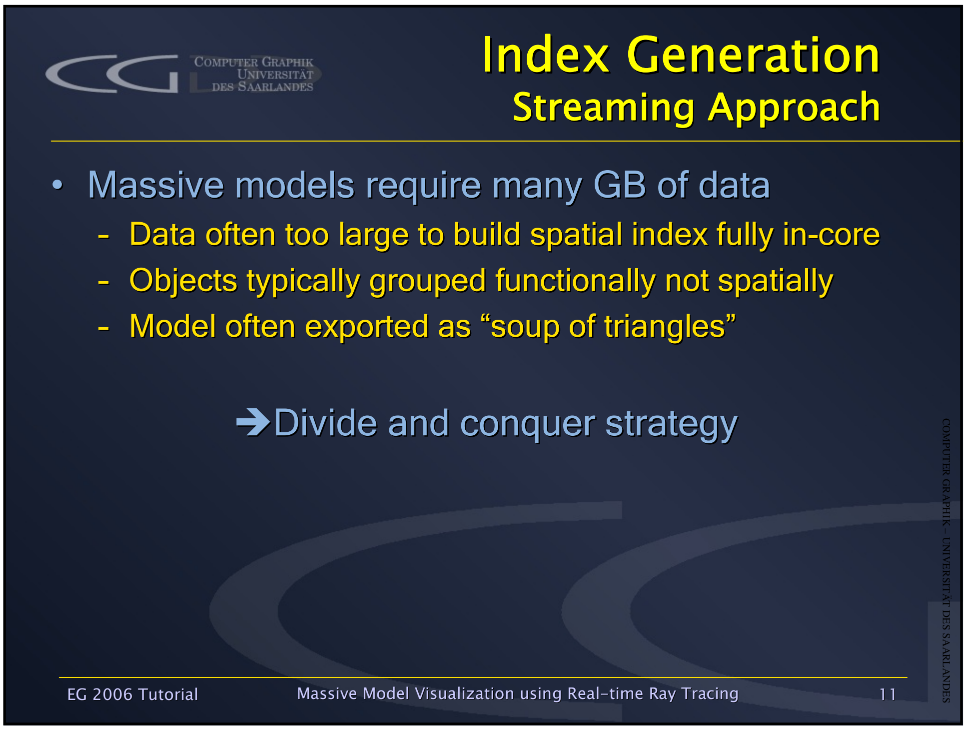
- Two-level k-d tree scheme enables
  - Rigid-body dynamics
    - Only high-level k-d tree has to be rebuilt
  - Efficient instancing
    - Low-level objects can be reused with little memory overhead



Index Generation

- Simple case
  - Source model grouped/partitioned into individual objects
  - Object boxes are not extensively overlapping
  - Data for single object fits into memory

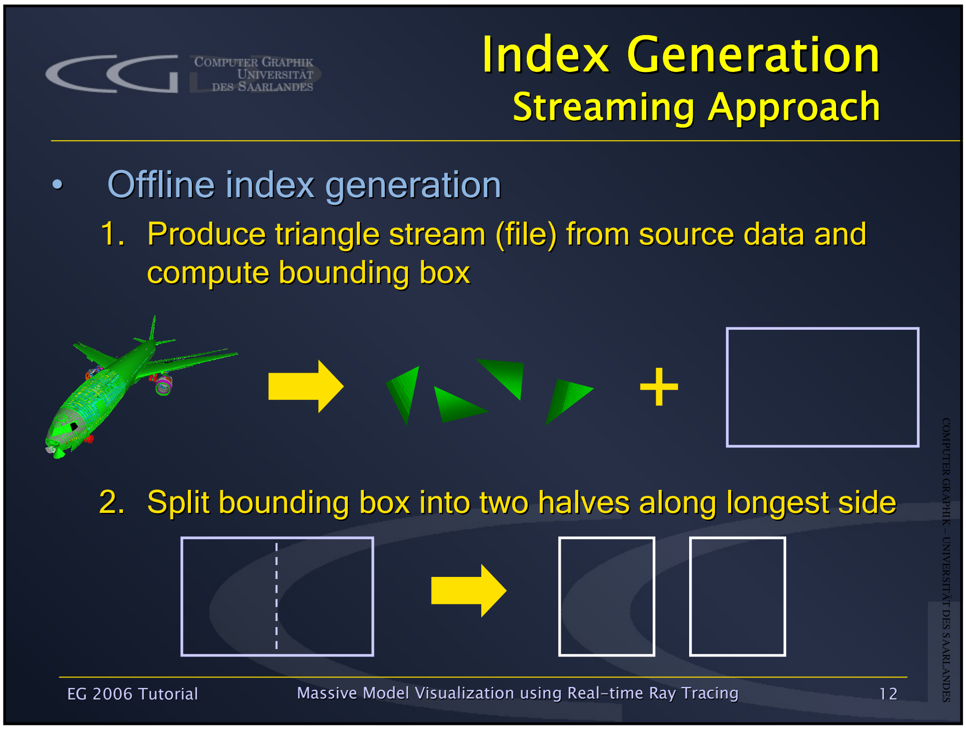
➔Build k-d trees independently
  - Build low-level object k-d trees one after another
  - Build high-level scene k-d tree based on objects boxes



Index Generation
Streaming Approach

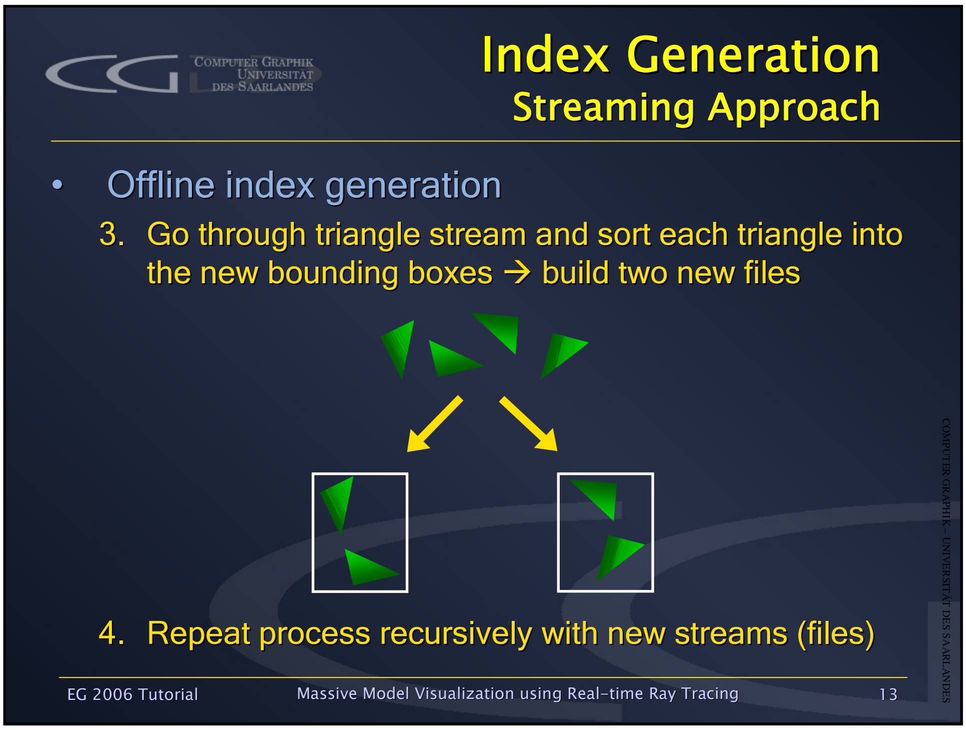
- Massive models require many GB of data
  - Data often too large to build spatial index fully in-core
  - Objects typically grouped functionally not spatially
  - Model often exported as “soup of triangles”

➔Divide and conquer strategy



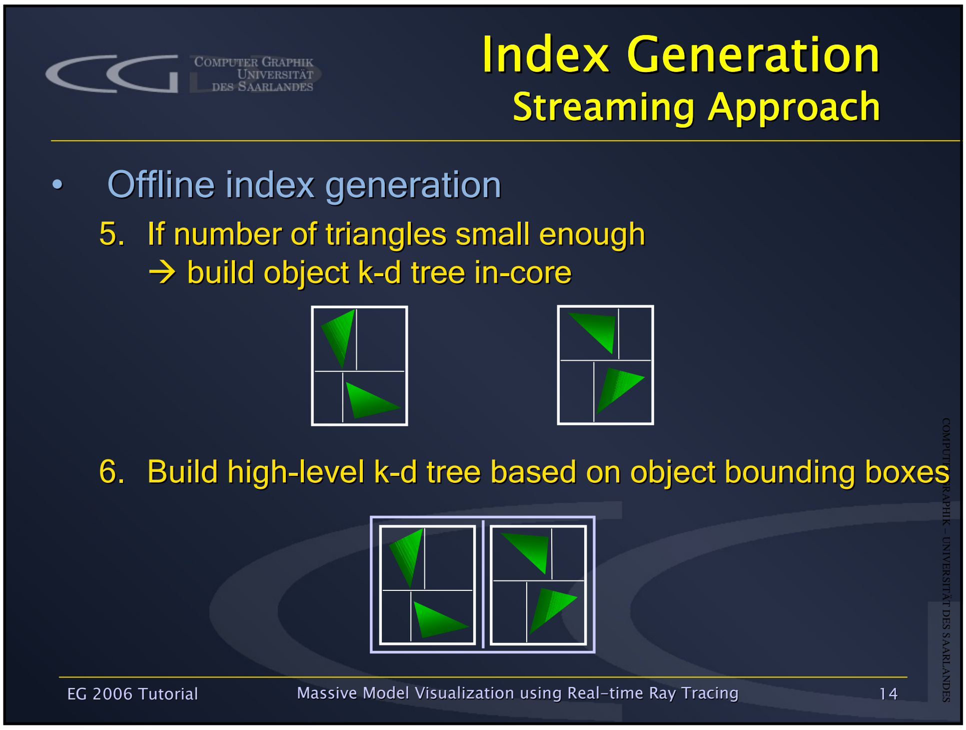
Index Generation
Streaming Approach

- Offline index generation
  1. Produce triangle stream (file) from source data and compute bounding box
  2. Split bounding box into two halves along longest side



Index Generation
Streaming Approach

- Offline index generation
  3. Go through triangle stream and sort each triangle into the new bounding boxes → build two new files
  4. Repeat process recursively with new streams (files)



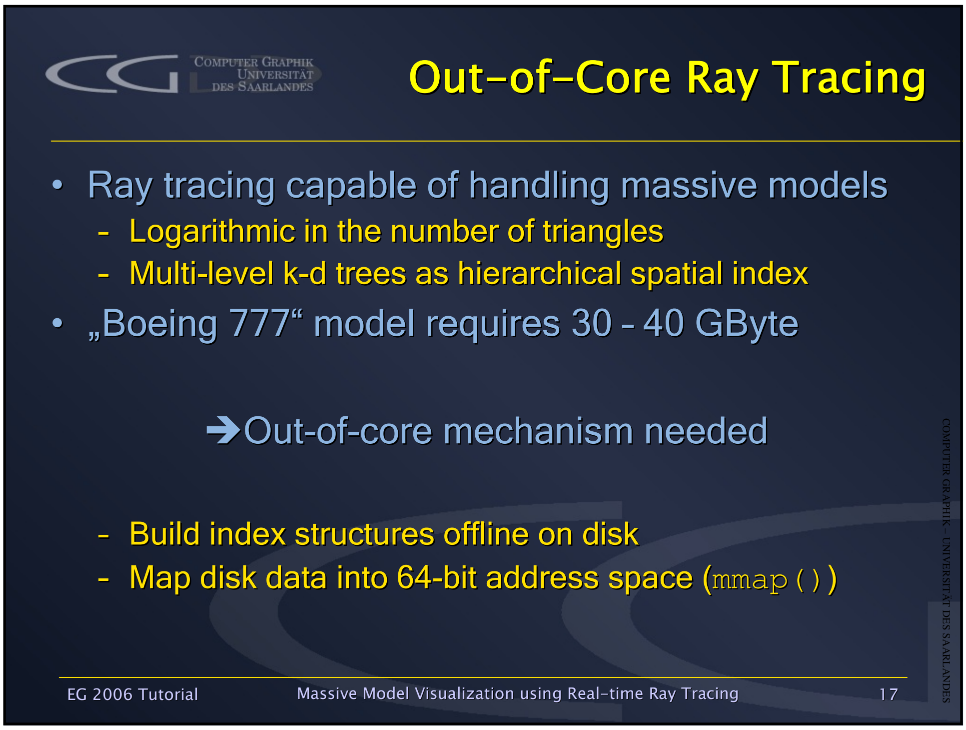
Index Generation
Streaming Approach

- Offline index generation
  5. If number of triangles small enough → build object k-d tree in-core
  6. Build high-level k-d tree based on object bounding boxes

Index Generation
Streaming Approach

- Optimizations
  - Remove vertex shading data from triangle stream
    - Normals, UV-coordinates, vertex colors, etc.
    - Do sorting only with vertex position data
    - Reconstruct full triangles after sorting
  - Compute better splitting planes
    - Use cost functions to determine plane position e.g., Surface Area Heuristics (SAH) [McDonald et al. 1989]
  - Parallelize sorting
    - Start extra processes for new streams

Part II
Adapting Ray Tracing To Complex Models

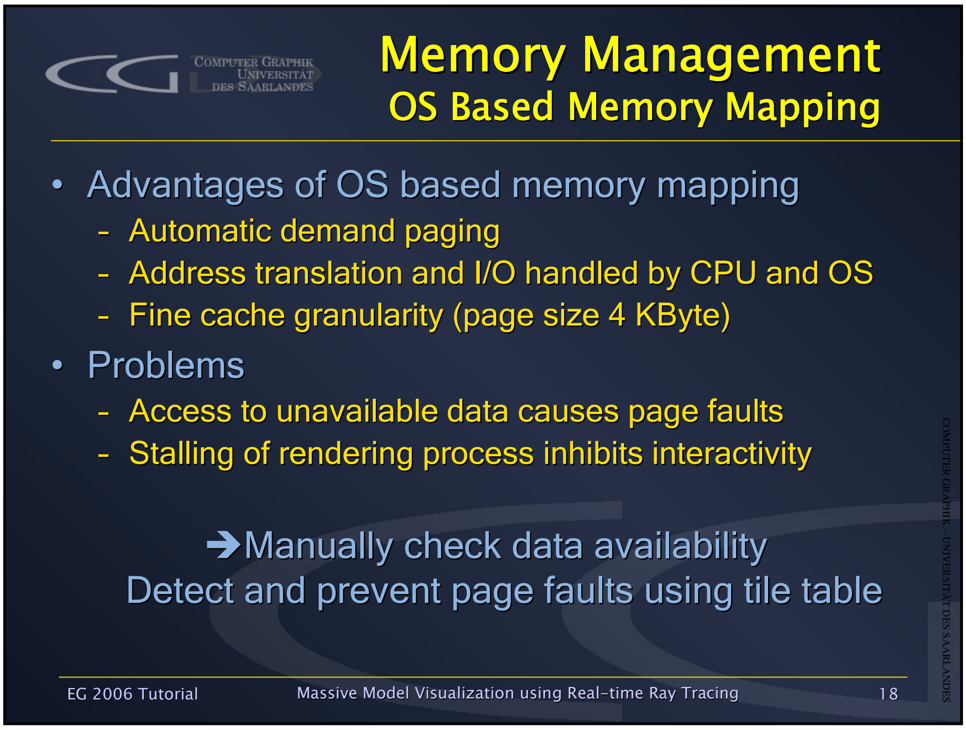


Out-of-Core Ray Tracing

- Ray tracing capable of handling massive models
  - Logarithmic in the number of triangles
  - Multi-level k-d trees as hierarchical spatial index
- „Boeing 777“ model requires 30 - 40 GByte

➔Out-of-core mechanism needed

  - Build index structures offline on disk
  - Map disk data into 64-bit address space (`mmap()`)

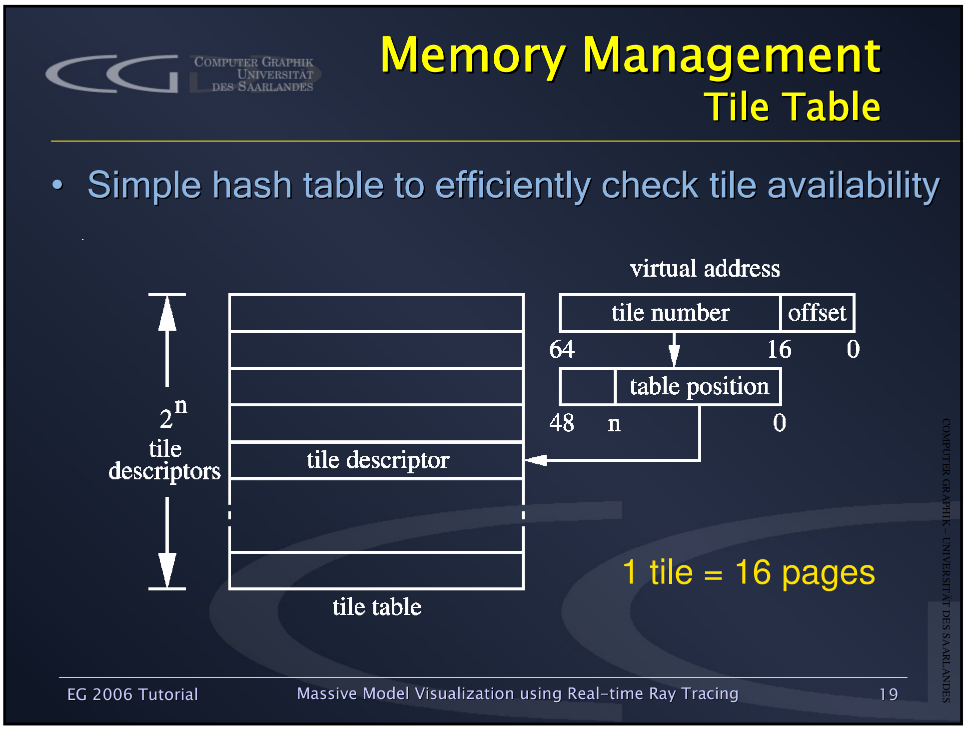


Memory Management

OS Based Memory Mapping

- Advantages of OS based memory mapping
  - Automatic demand paging
  - Address translation and I/O handled by CPU and OS
  - Fine cache granularity (page size 4 KByte)
- Problems
  - Access to unavailable data causes page faults
  - Stalling of rendering process inhibits interactivity

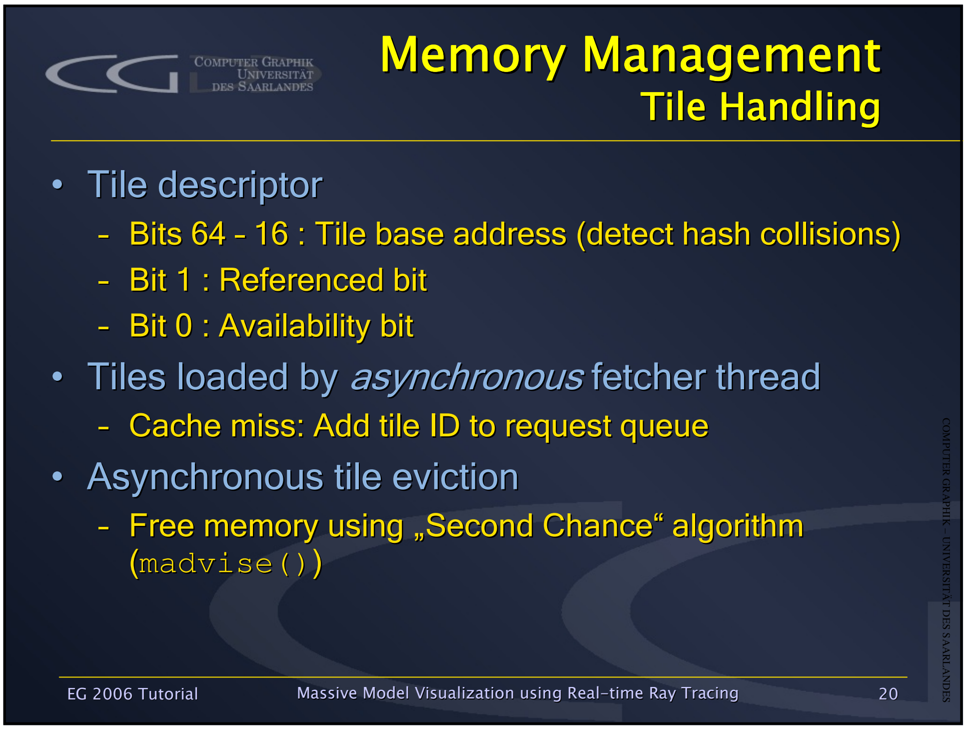
➔Manually check data availability
Detect and prevent page faults using tile table



Memory Management
Tile Table

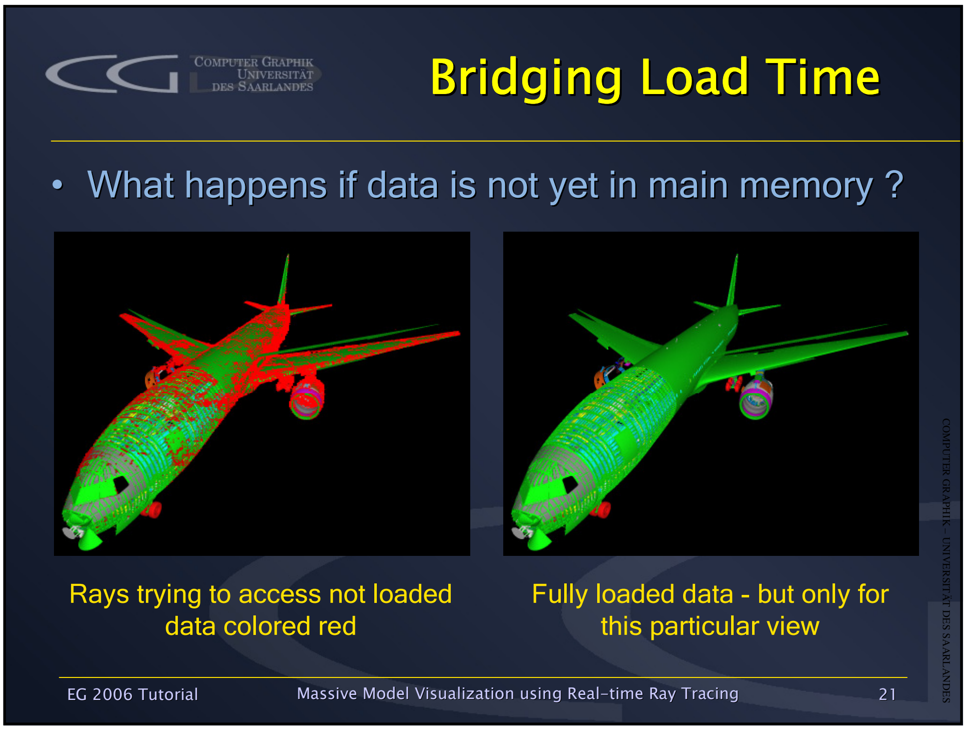
- Simple hash table to efficiently check tile availability

1 tile = 16 pages



Memory Management
Tile Handling

- Tile descriptor
  - Bits 64 - 16 : Tile base address (detect hash collisions)
  - Bit 1 : Referenced bit
  - Bit 0 : Availability bit
- Tiles loaded by *asynchronous* fetcher thread
  - Cache miss: Add tile ID to request queue
- Asynchronous tile eviction
  - Free memory using „Second Chance“ algorithm (`madvise()`)

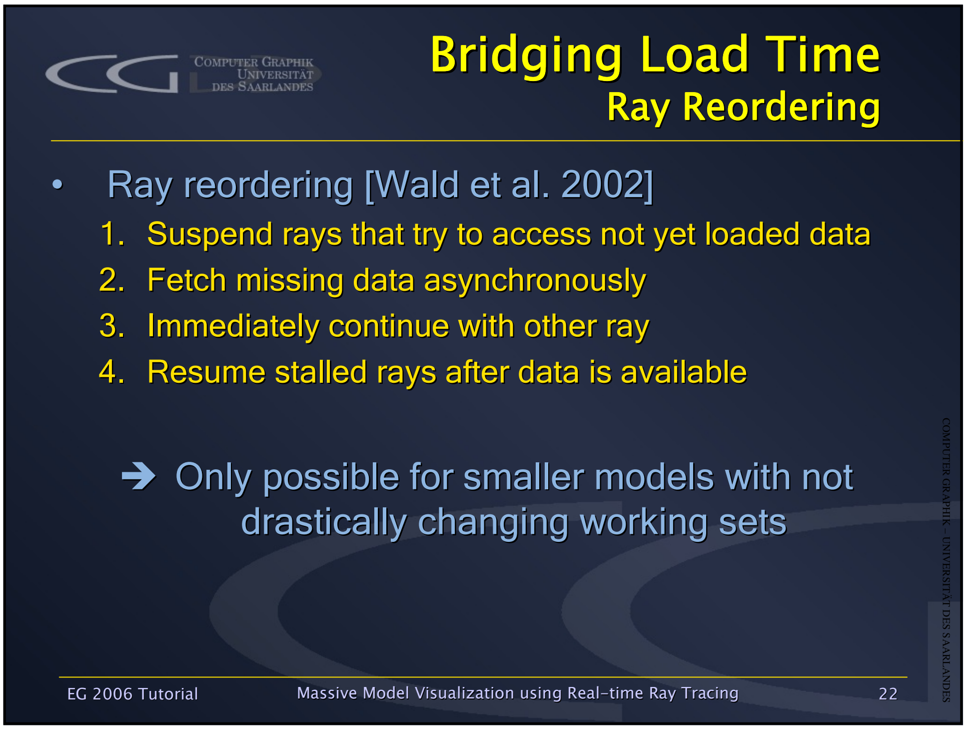


Bridging Load Time

- What happens if data is not yet in main memory ?

Rays trying to access not loaded data colored red

Fully loaded data - but only for this particular view

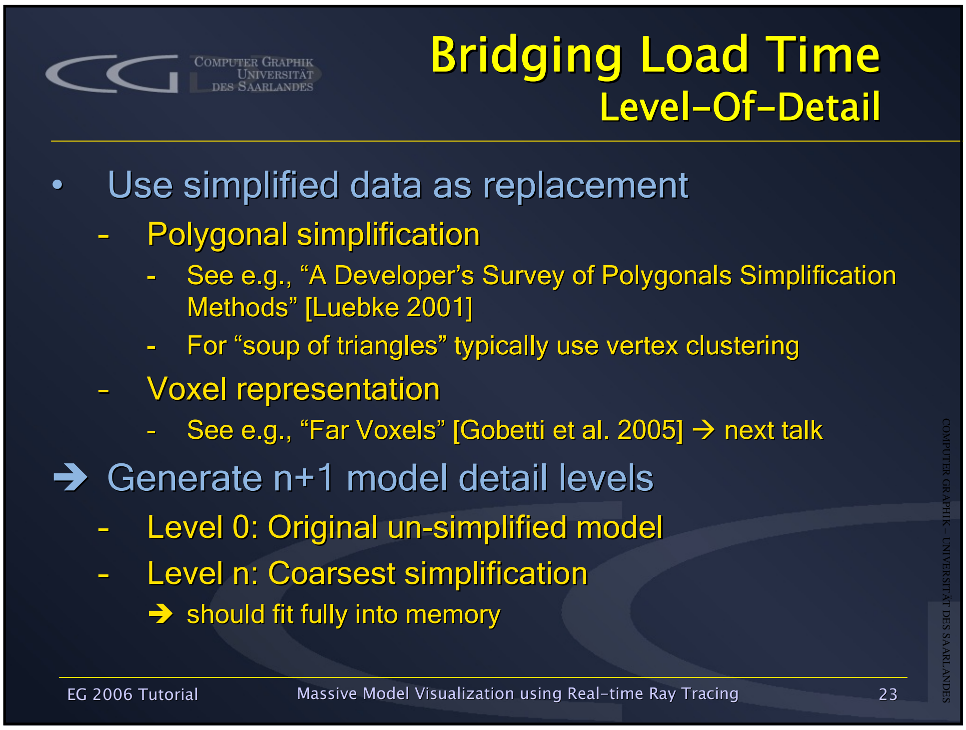


Bridging Load Time

Ray Reordering

- Ray reordering [Wald et al. 2002]
  1. Suspend rays that try to access not yet loaded data
  2. Fetch missing data asynchronously
  3. Immediately continue with other ray
  4. Resume stalled rays after data is available

➔ Only possible for smaller models with not drastically changing working sets

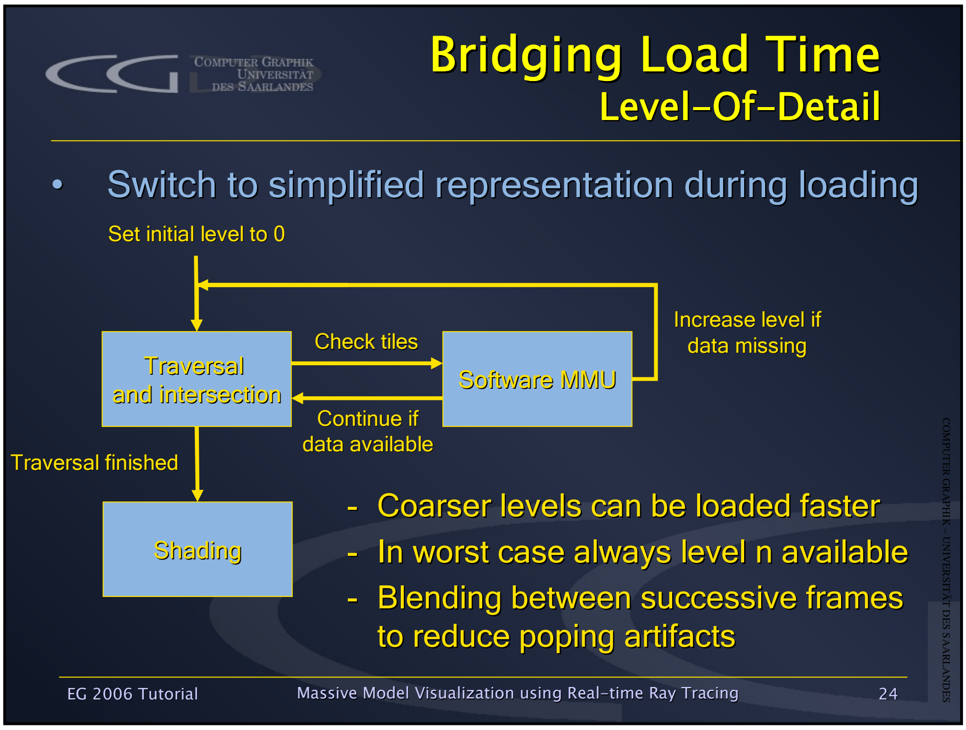


Bridging Load Time

Level-Of-Detail

- Use simplified data as replacement
  - Polygonal simplification
    - See e.g., “A Developer’s Survey of Polygonals Simplification Methods” [Luebke 2001]
    - For “soup of triangles” typically use vertex clustering
  - Voxel representation
    - See e.g., “Far Voxels” [Gobetti et al. 2005] → next talk

➔ Generate n+1 model detail levels
  - Level 0: Original un-simplified model
  - Level n: Coarsest simplification
    ➔ should fit fully into memory



Bridging Load Time

Level-Of-Detail

- Switch to simplified representation during loading

Set initial level to 0

Traversal and intersection

Check tiles

Software MMU

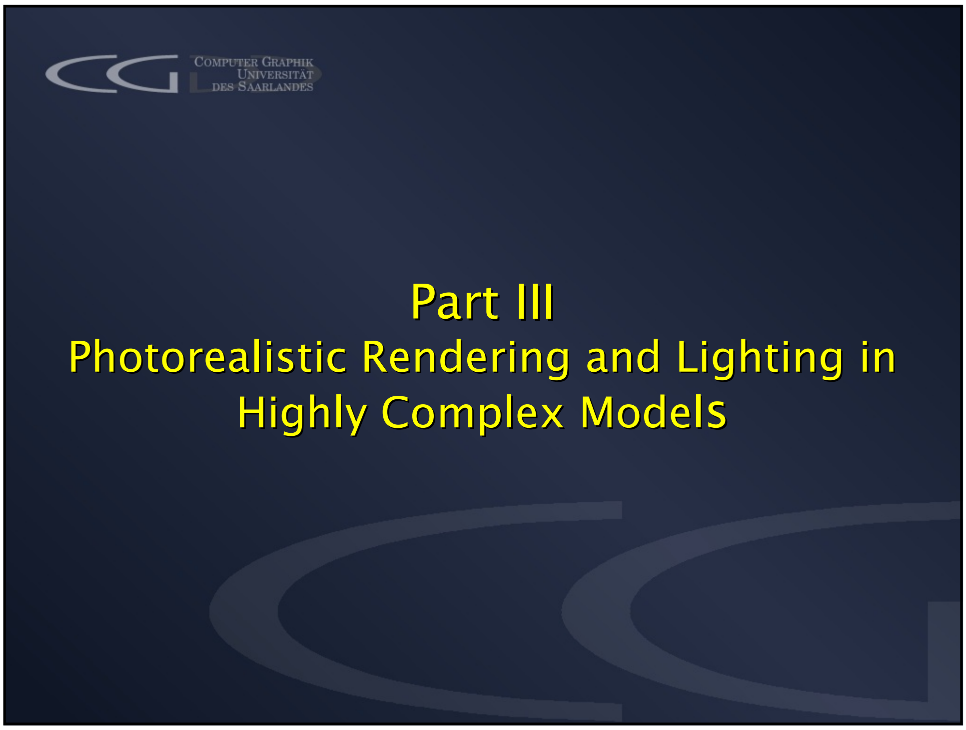
Increase level if data missing

Continue if data available

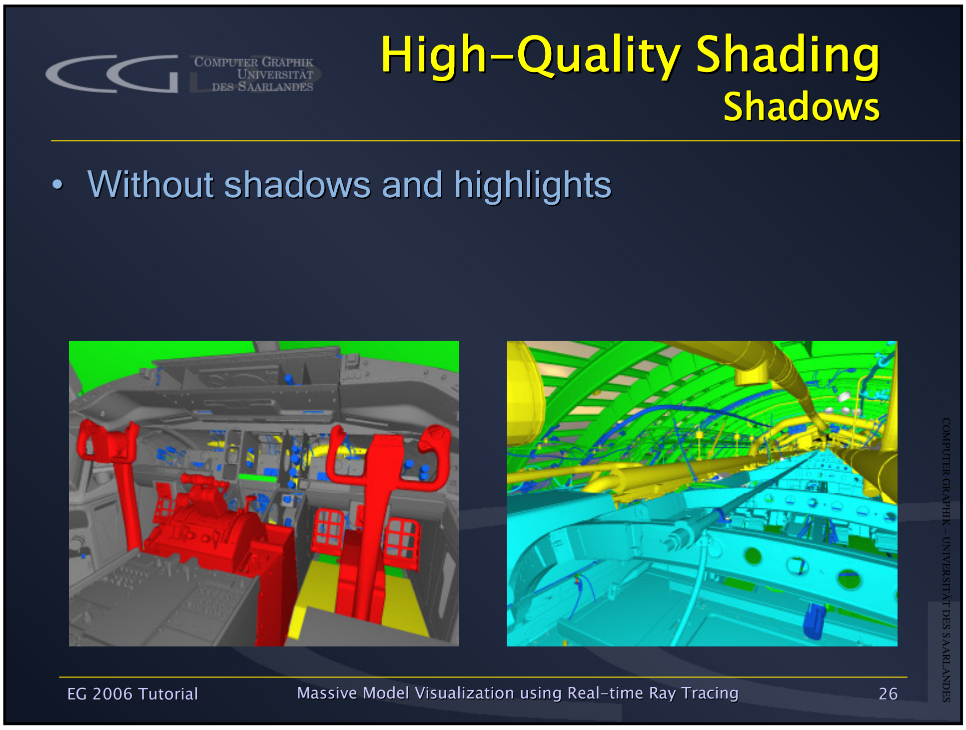
Traversal finished

Shading

- Coarser levels can be loaded faster
- In worst case always level n available
- Blending between successive frames to reduce poping artifacts



Part III
Photorealistic Rendering and Lighting in Highly Complex Models

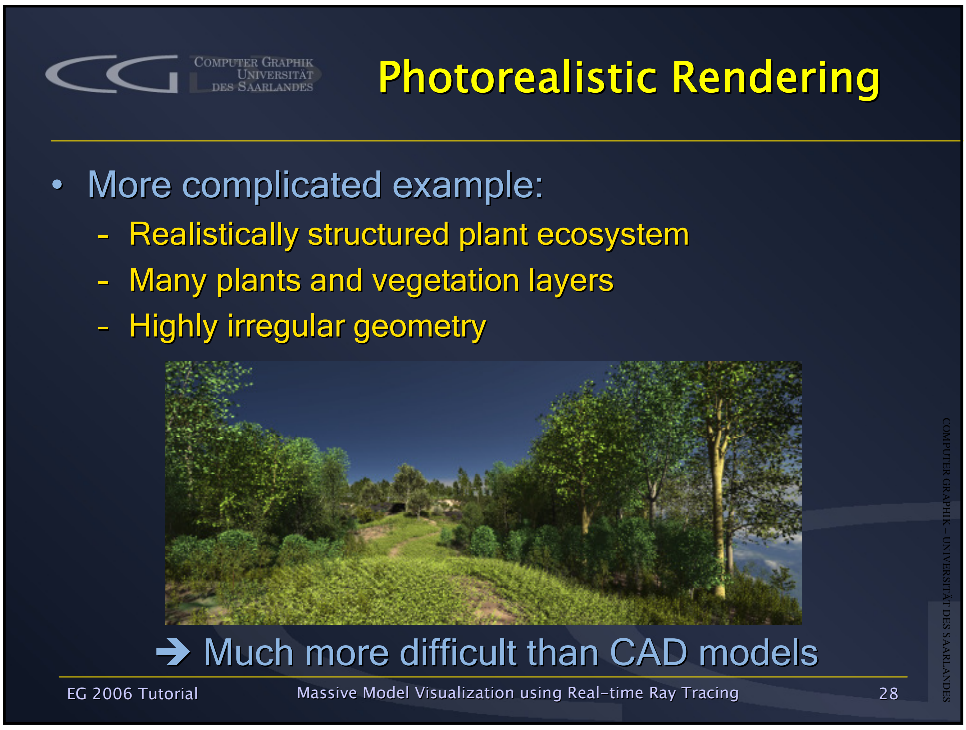


High-Quality Shading
Shadows

- Without shadows and highlights

High–Quality Shading
Shadows

- Pixel-accurate shadows and highlights
  - Simple integration into a ray tracer
    - Shader (plug-in) is called when a ray hits a surface
    - Shaders can fire arbitrary rays (for shadows, reflection, …)



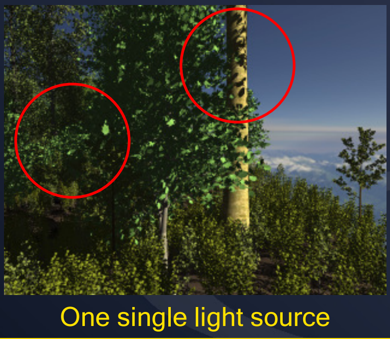
Photorealistic Rendering

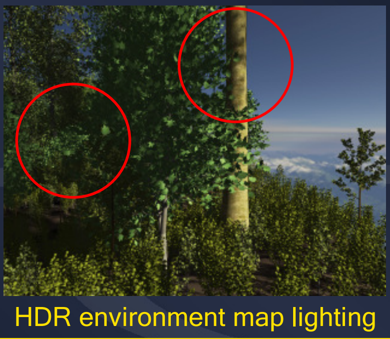
- More complicated example:
  - Realistically structured plant ecosystem
  - Many plants and vegetation layers
  - Highly irregular geometry

➔ Much more difficult than CAD models

Realistic Lighting
Environmental Illumination

- Realistic Illumination of outdoor scenes
  - Depends heavily on environmental illumination
  - One single directional light not sufficient
    - Cannot capture subtle effects, e.g. soft shadows

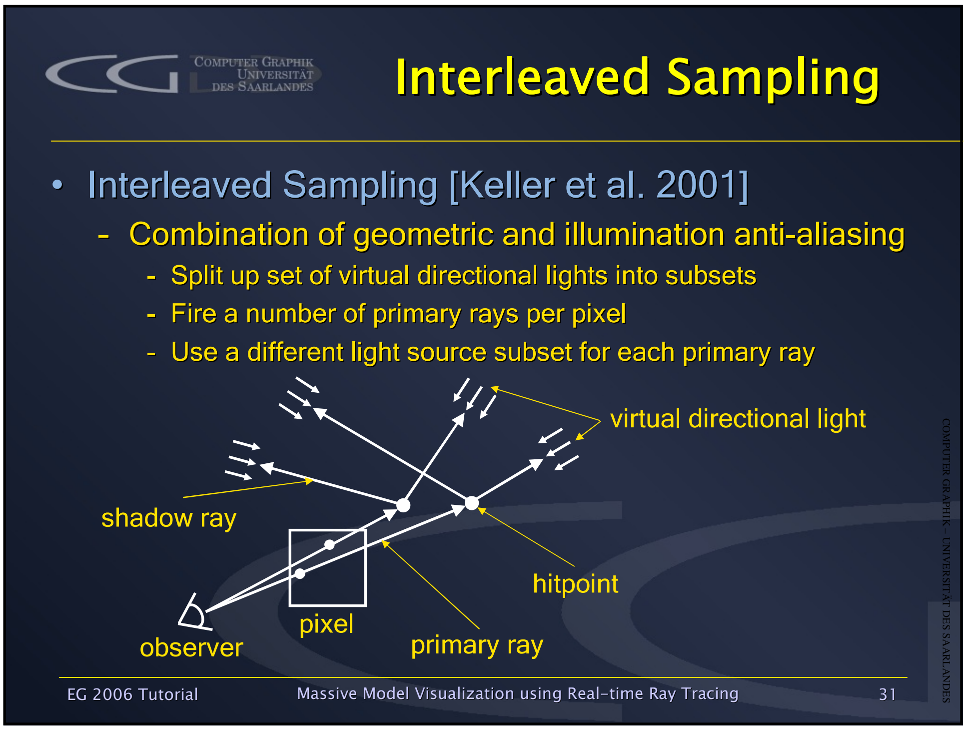

One single light source


HDR environment map lighting

Realistic Lighting
HDRI Approximation

- Pre-computation e.g. PRT not practical
  - Scene too complex → memory limitations
  - Difficult to use with instantiation

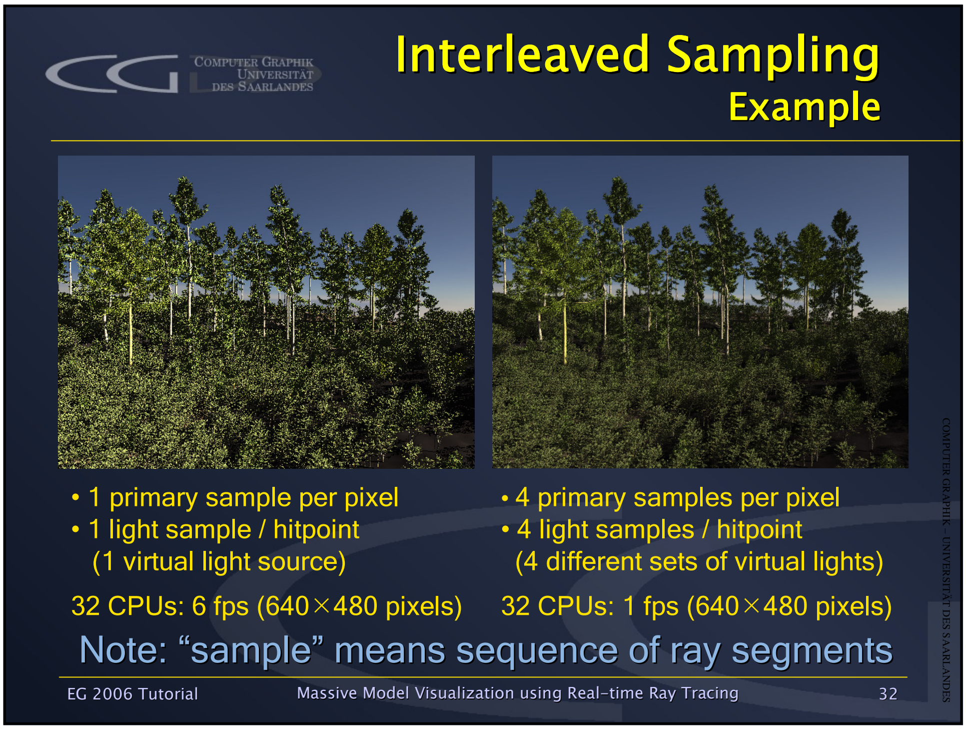
→Approximate HDR environmental illumination
  - Approximate with large number of directional lights
    - Generated from HDR environment maps
      e.g., similar to [Kollig et al. 2003] or [Agarwal et al. 2003]
  - Randomly pick subsets from these virtual lights
    - Use as targets for shadow rays
    - Interleave shadow rays with primary rays



Interleaved Sampling

- Interleaved Sampling [Keller et al. 2001]
  - Combination of geometric and illumination anti-aliasing
    - Split up set of virtual directional lights into subsets
    - Fire a number of primary rays per pixel
    - Use a different light source subset for each primary ray

---



Interleaved Sampling
Example

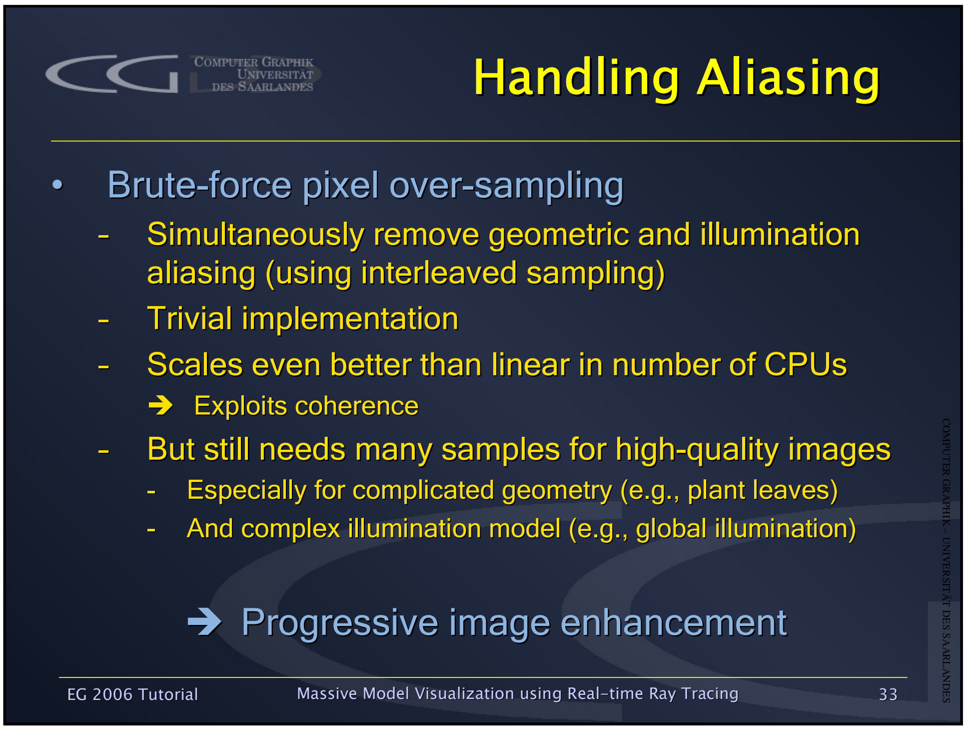
- 1 primary sample per pixel
- 1 light sample / hitpoint
  (1 virtual light source)

32 CPUs: 6 fps (640×480 pixels)

- 4 primary samples per pixel
- 4 light samples / hitpoint
  (4 different sets of virtual lights)

32 CPUs: 1 fps (640×480 pixels)

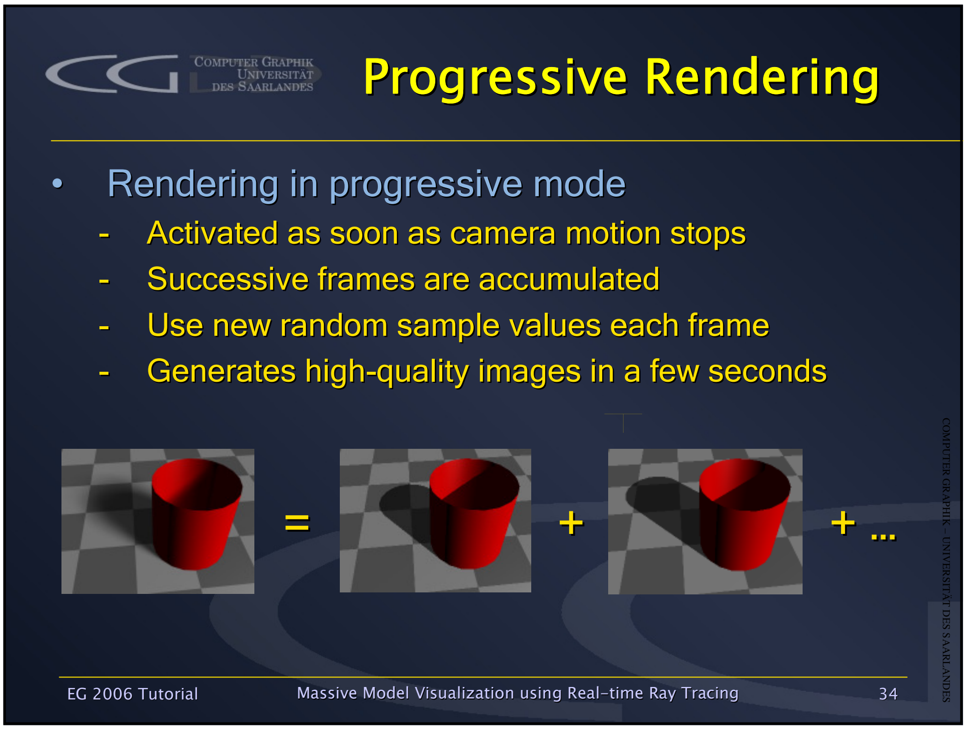
Note: “sample” means sequence of ray segments



Handling Aliasing

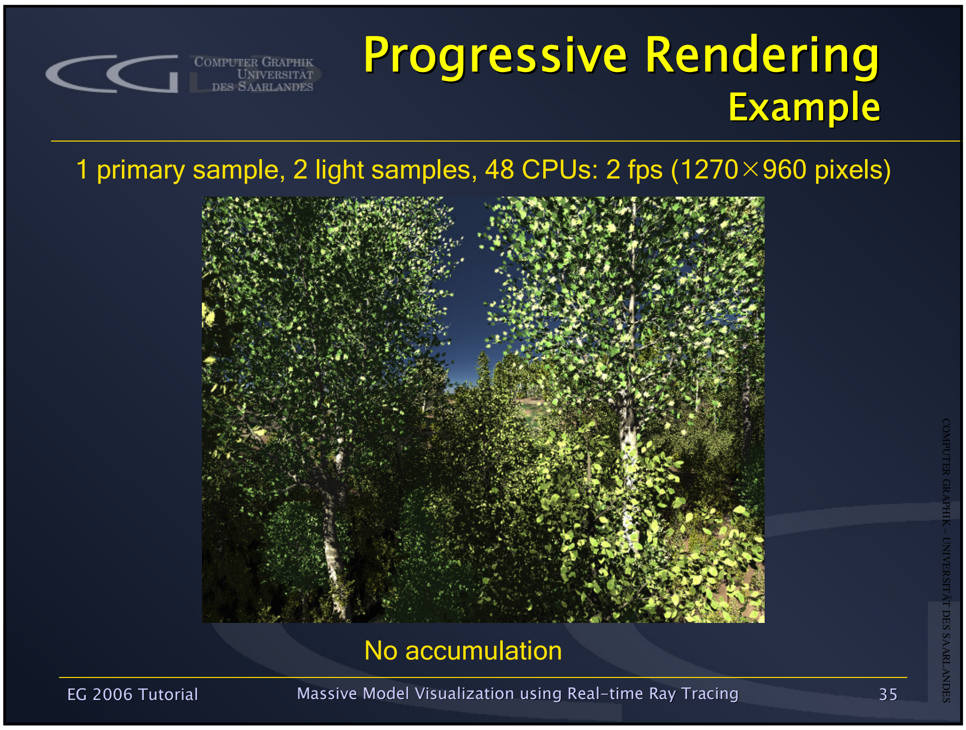
- Brute-force pixel over-sampling
  - Simultaneously remove geometric and illumination aliasing (using interleaved sampling)
  - Trivial implementation
  - Scales even better than linear in number of CPUs
    - ➔ Exploits coherence
  - But still needs many samples for high-quality images
    - Especially for complicated geometry (e.g., plant leaves)
    - And complex illumination model (e.g., global illumination)

➔ Progressive image enhancement



Progressive Rendering

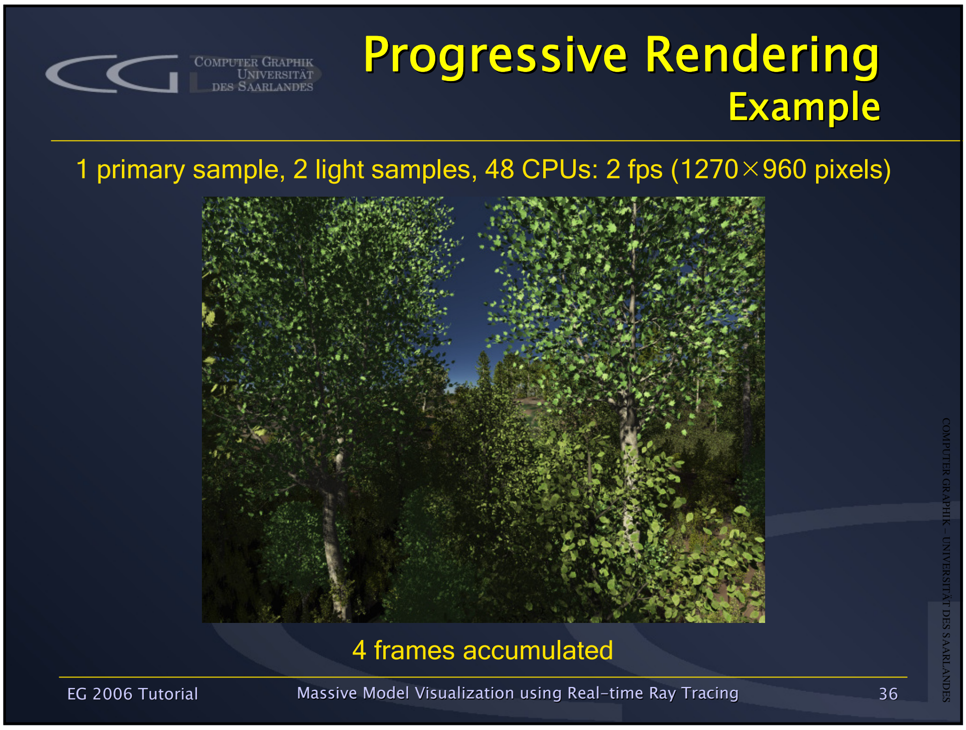
- Rendering in progressive mode
  - Activated as soon as camera motion stops
  - Successive frames are accumulated
  - Use new random sample values each frame
  - Generates high-quality images in a few seconds



Progressive Rendering
Example

1 primary sample, 2 light samples, 48 CPUs: 2 fps (1270×960 pixels)

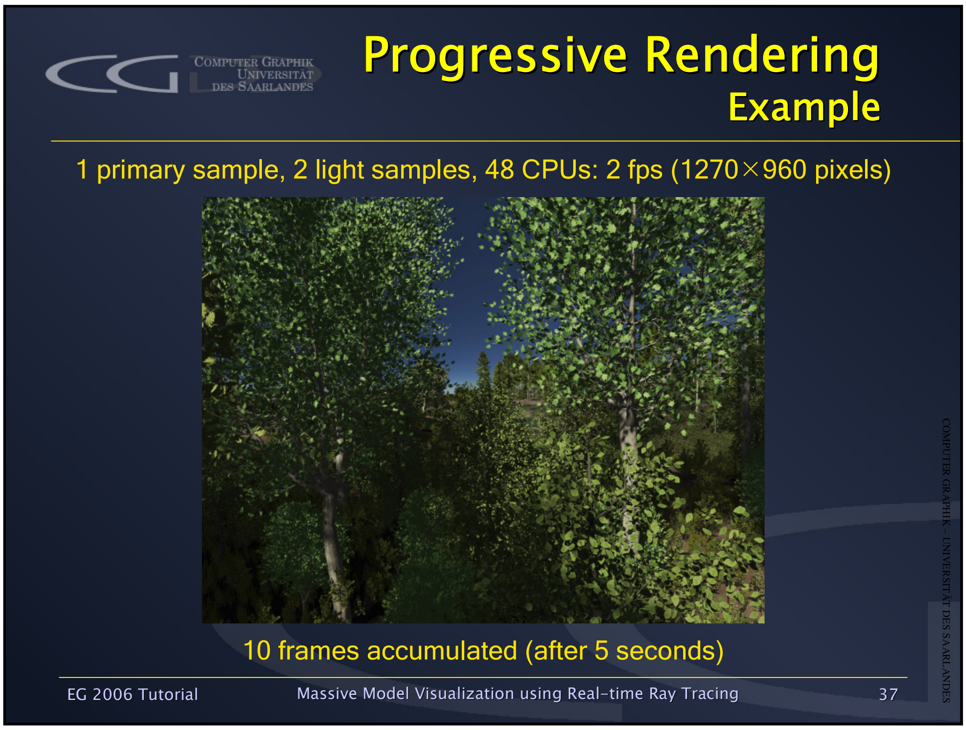
No accumulation



Progressive Rendering
Example

1 primary sample, 2 light samples, 48 CPUs: 2 fps (1270×960 pixels)

4 frames accumulated

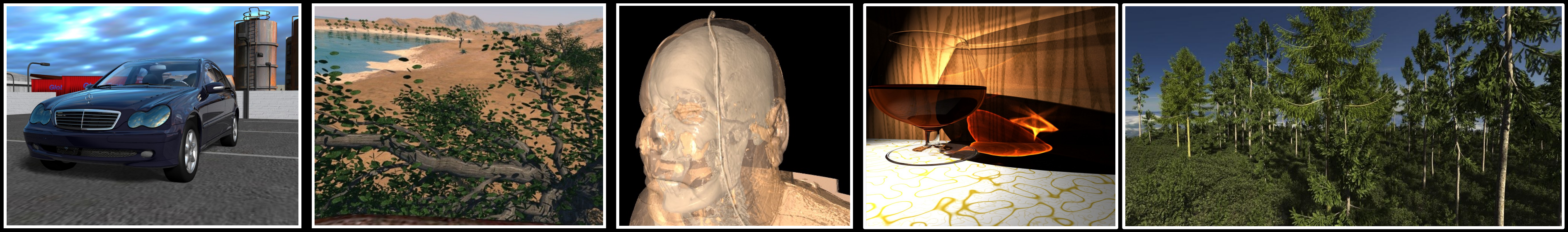


Progressive Rendering

Example

1 primary sample, 2 light samples, 48 CPUs: 2 fps (1270×960 pixels)

10 frames accumulated (after 5 seconds)

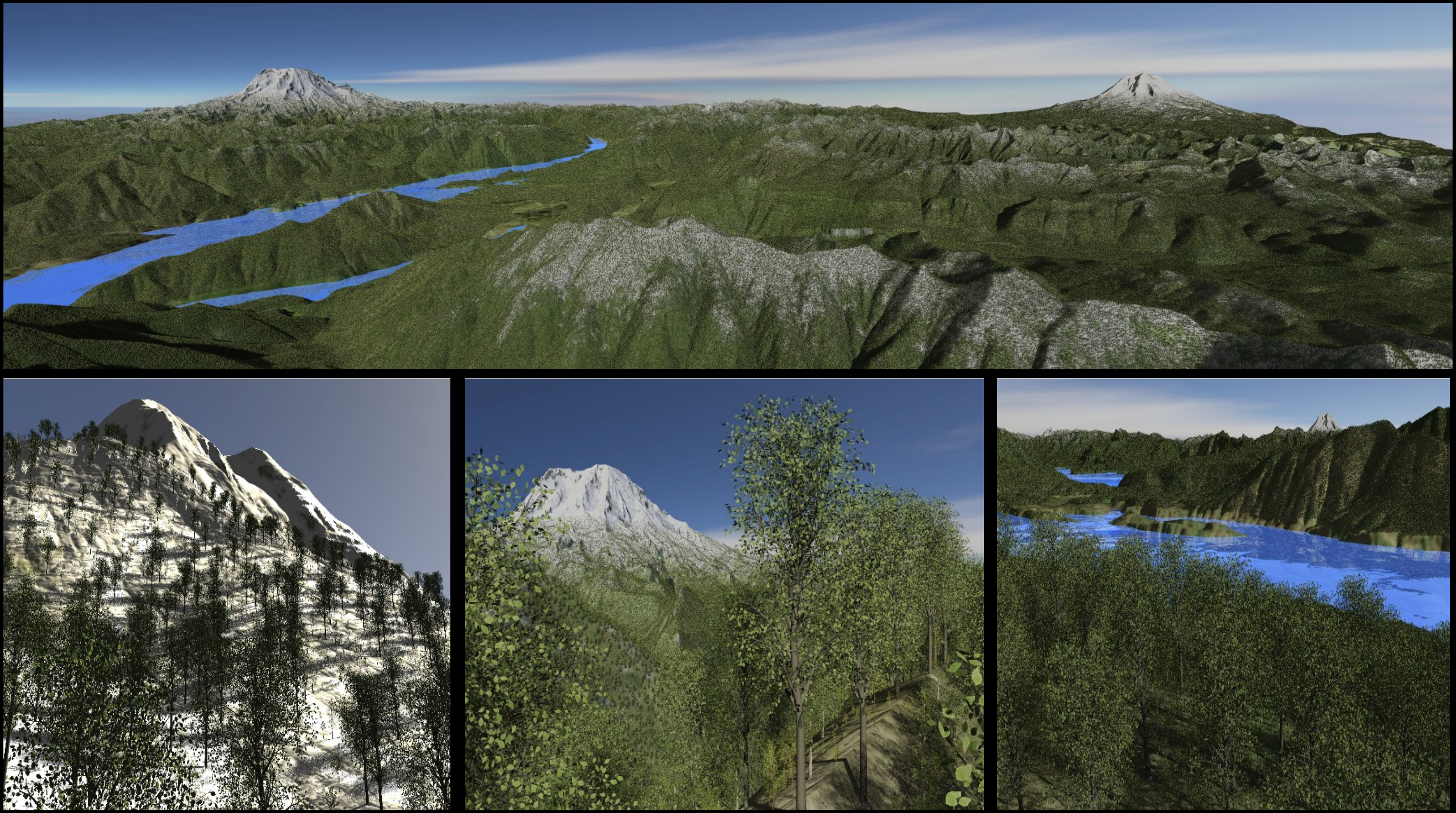


Hardware Trends for Realtime Ray Tracing

Philipp Slusallek
Saarland University, Germany

informatik
saarland.

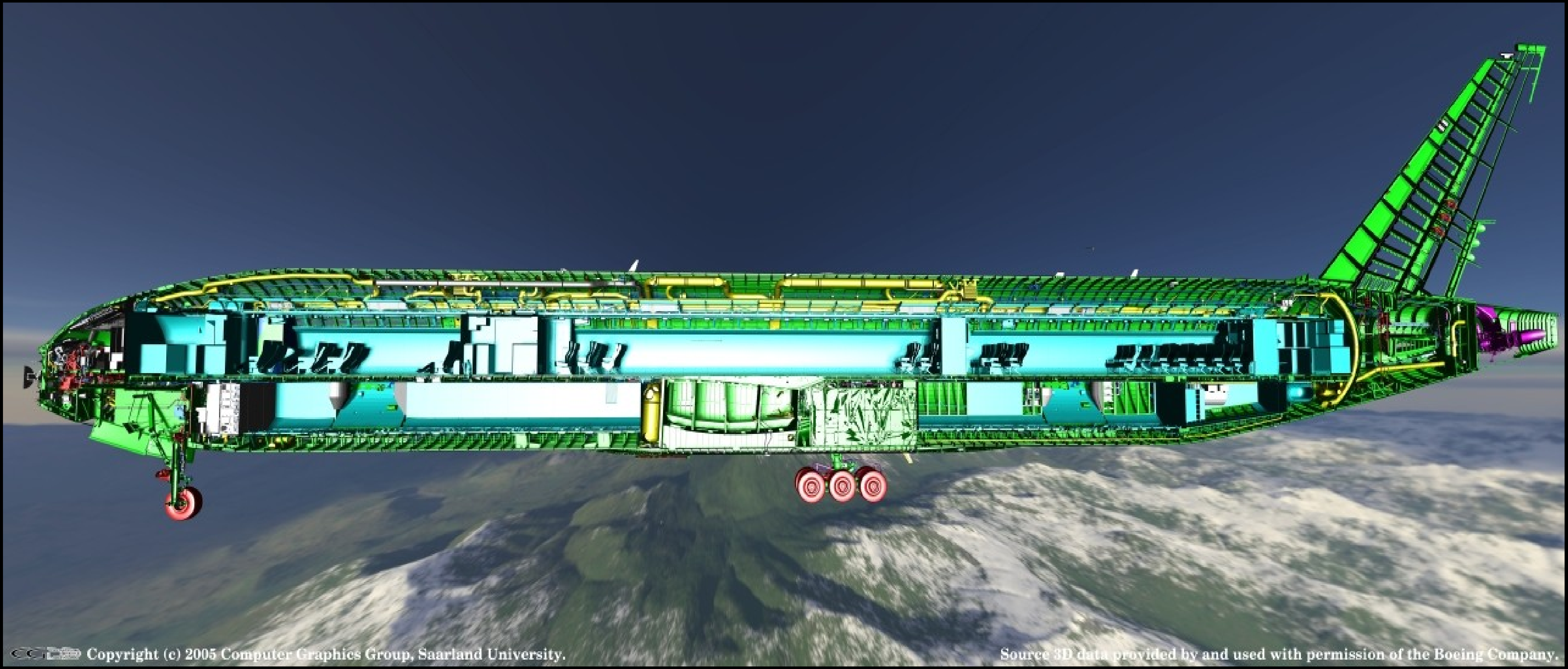
Outdoor Environments with full Sky Illumination



Outdoor Environments with full Sky Illumination

71 Trillion Triangles

Large Model Visualization at Boeing



Copyright (c) 2005 Computer Graphics Group, Saarland University.

Source 3D data provided by and used with permission of the Boeing Company.

CATIA Model of Boeing 777:
350 million triangles, 30 GB on disk, 2-3 fps on Dual-Opteron

VW Visualization Center
by inTrace GmbH

VW Visualization Center
by inTrace GmbH

VW Visualization Center
by inTrace GmbH

Lighting Simulation at EADS

Product Visualization at EADS

Ray Traced Games

Realtime Requirements

- Minimum Number of Rays
  - 1 megapixel screen
  - 30 frames per second
  - 10 rays per pixel (anti-aliasing, lighting, ...)
  - ➔ 300 million rays per second
- But
  - Larger screens (2x), higher frame rate (2x)
  - Complex lighting (10x)
- Promising: Adaptive space-time sampling

Ray Tracing on Multi-Core

- Advantages:
  - High-performance implementations are available
  - Highly flexible environment
  - Scales nicely with # of cores (~10 Mrays/s per core)
- Disadvantage
  - Need 30 cores for minimum requirements
- Not for the mass market any time soon

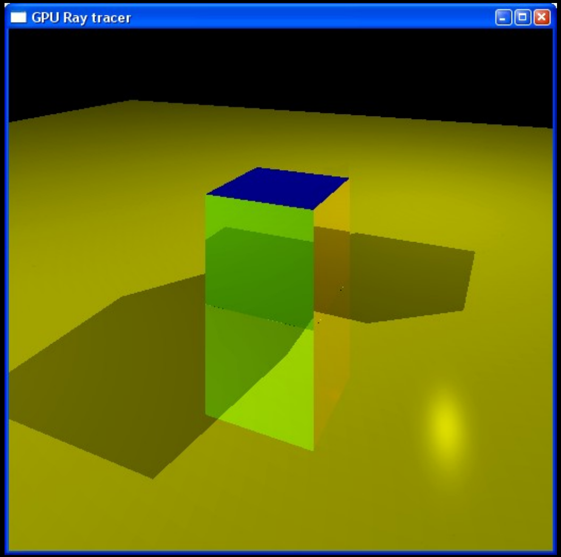
Ray Tracing on Multi-Core

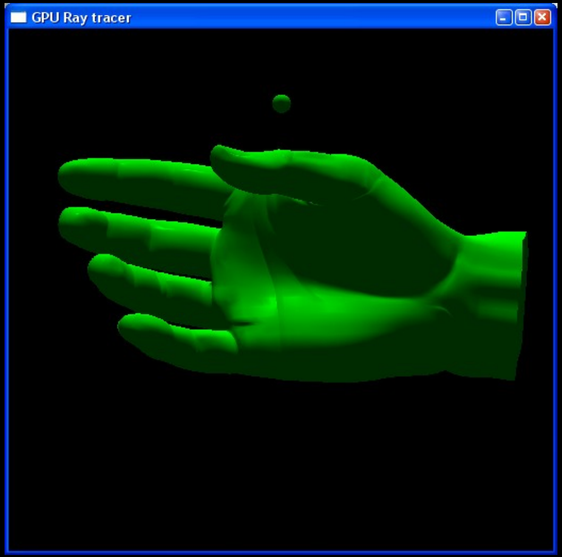
- Advantages:
  - High-performance implementations are available
  - Highly flexible environment
  - Scales nicely with # of cores (~10 Mrays/s per core)
- Disadvantage
  - Need 30 cores for minimum requirements
- Not for the mass market any time soon
  - But high-end systems are becoming available
    - Opteron-System (8 CPUs x Quad-Core) → 32 cores

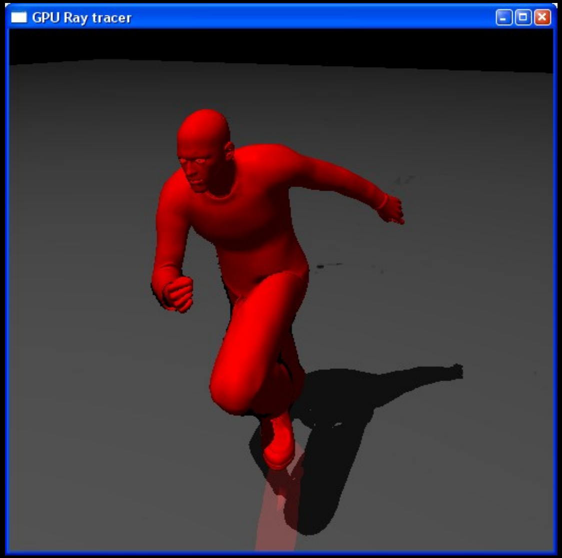
Ray Tracing on GPUs

- Increasingly Implemented as an Add-On
  - Volume rendering by ray casting [Krüger '03]
  - Displacement mapping [Wang '04]
  - Approximate refractions on GPU [Weyman '05]
  - Screen space caustics [Krüger '06]
- Not well supported by GPUs
  - So far, less efficient than CPUs
    - Even though they have higher raw performance

Ray Tracing on GPUs: Performance @ 1024 x1024







Scene	Triangles	ATI x1900
Cube	16	5.0
Hand	17k	5.5
Ben	72k	1.1

Ray Tracing on Cell

- Advantages:
  - Already 8 compact but powerful cores (SPUs)
  - Highly efficient SIMD instruction set
  - DMA and full control over caches in LS
  - C/C++ compiler
- Disadvantages
  - Still hard to program, non-optimal compilers
  - Needs another programming approach
    - No good, high-level data parallel languages available
  - Complex and costly memory handling

Ray Tracing on Cell: Performance @ 1024 x1024

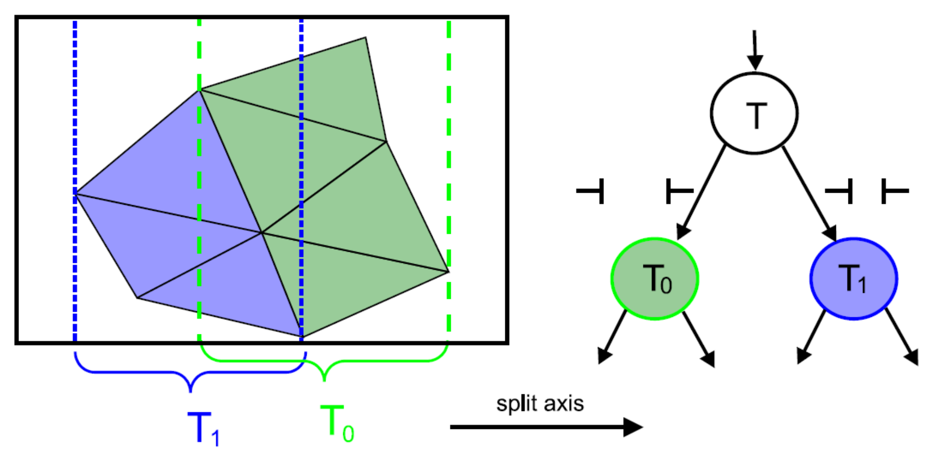
Scene	Triangles	Single-Cell	Dual
ERW6	800	58.1	110.9
Conference	280k	20.0	37.3
Beetle	680k	16.2	30.6

D-RPU Approach

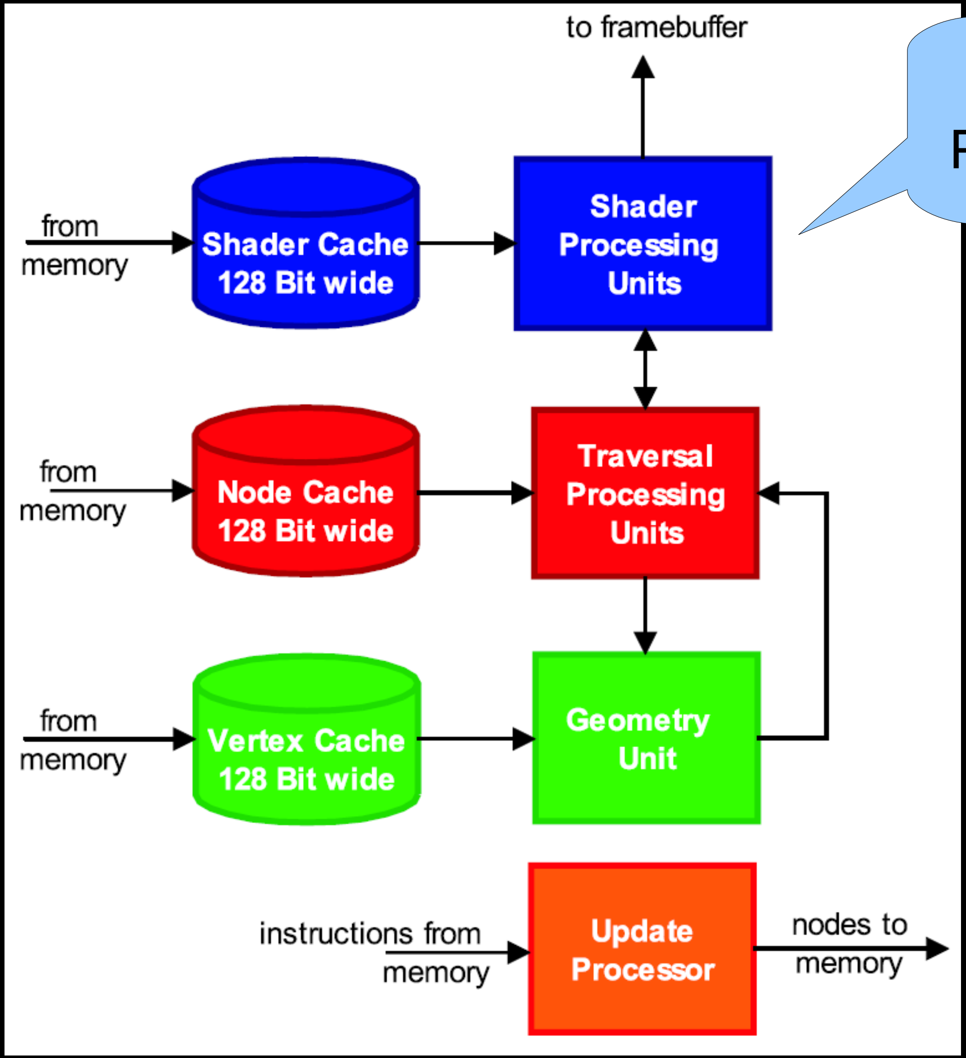
- **Shading processor**
  - Design similar to fragment processors on GPUs
  - Support for full recursion even with SIMD
  - Highly parallel, highly efficient
- **Improved programming model**
  - Add highly efficient recursion, conditional branching
  - Add flexible memory access (beyond textures)
- **Custom traversal and intersection hardware**
  - High-performance kd-tree traversal & triangle intersection

D-RPU: Dynamic Scenes [GH'06]

- Bounding KD-Trees (B-KD Trees)
  - Combining the best of two worlds
    - Traversal efficiency of kd-trees
    - Update efficiency of bounding volume hierarchies
  - Efficient for coherent motion with fixed topologies
  - Supports general rays
  - Good for empty space
- Implemented in HW
  - Traversal & update

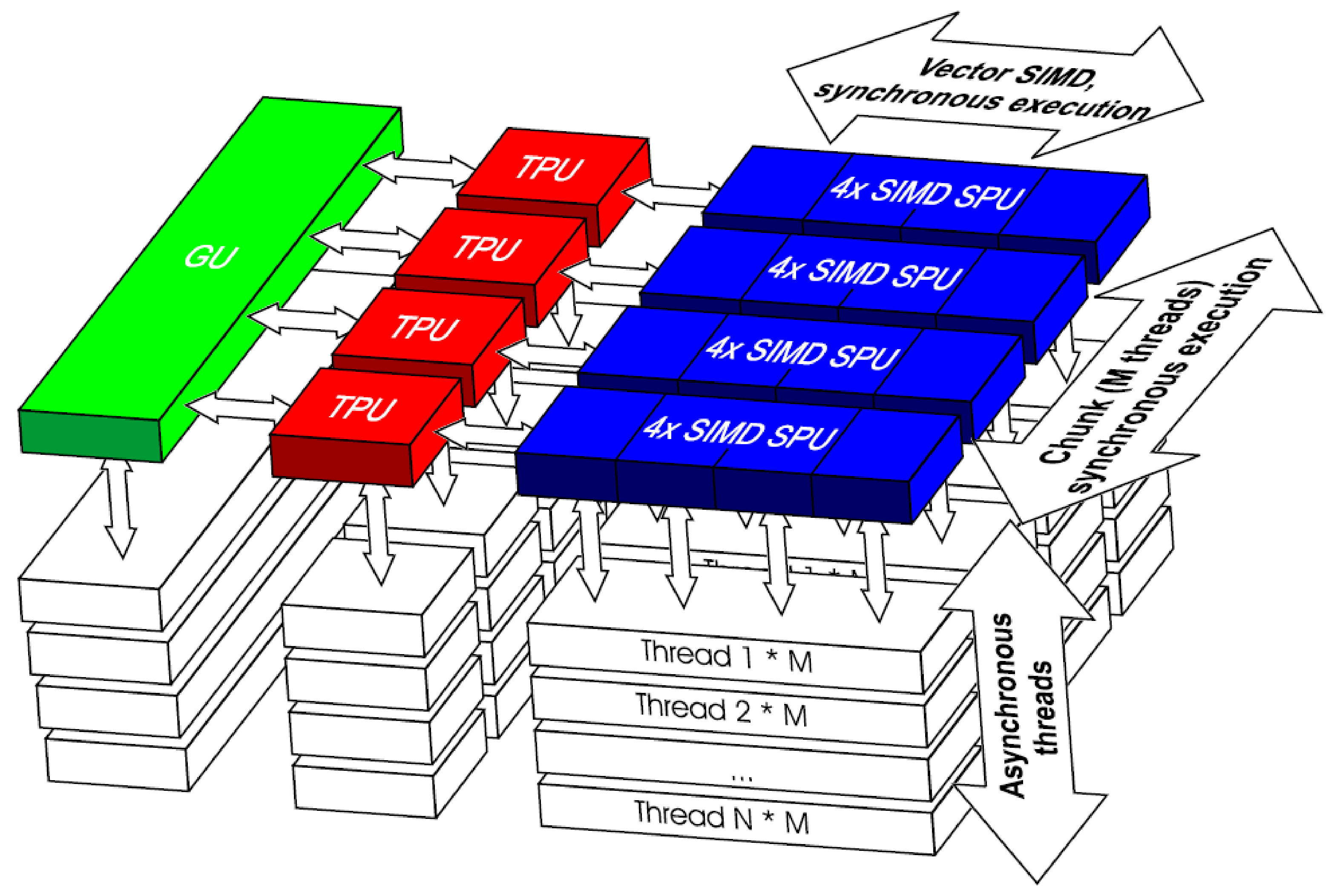


D-RPU: High-Level Architecture

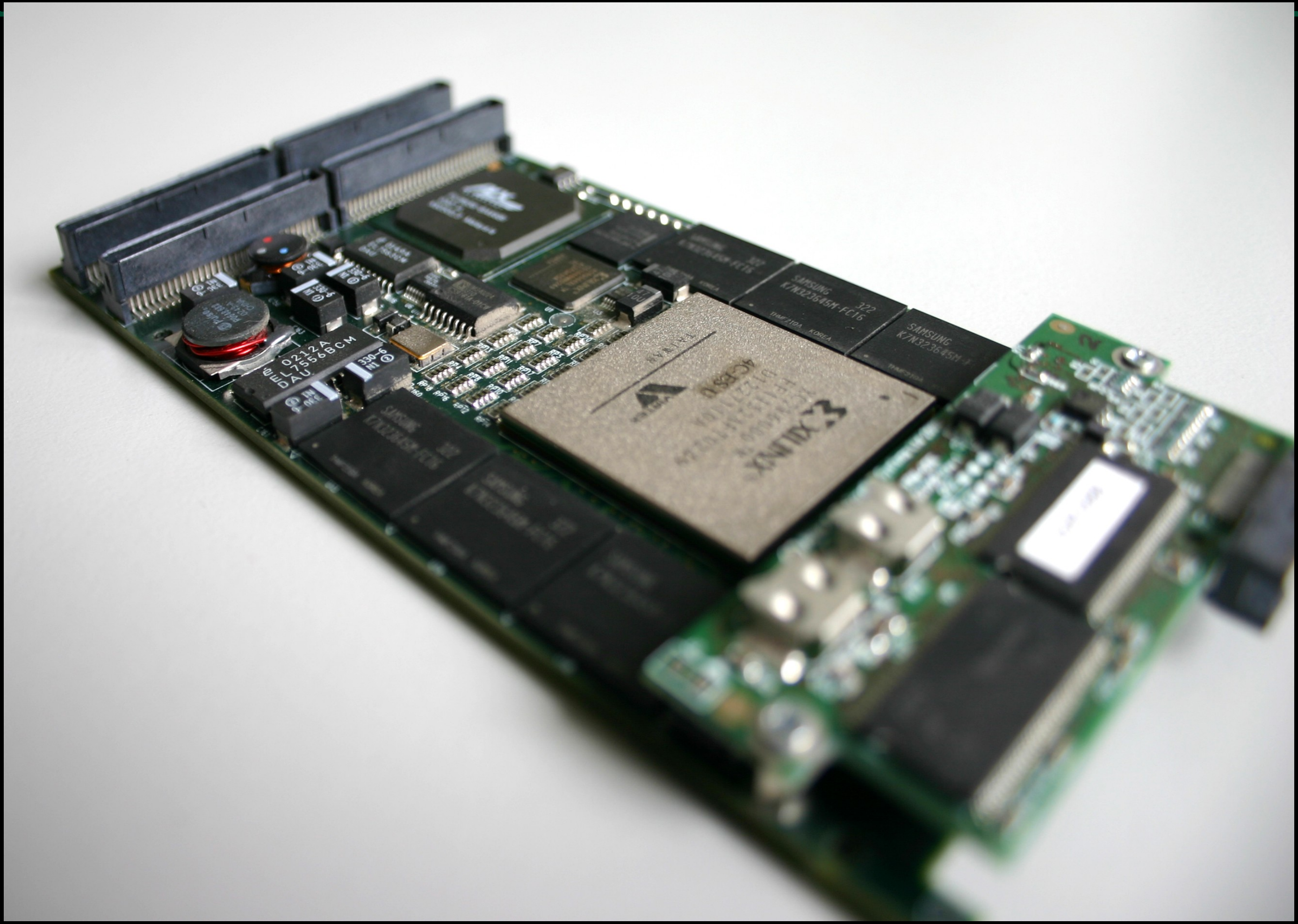


Fully Programmable

D-RPU: Hardware Architecture



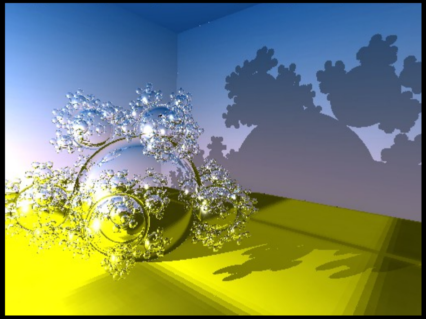
Hardware Implementation



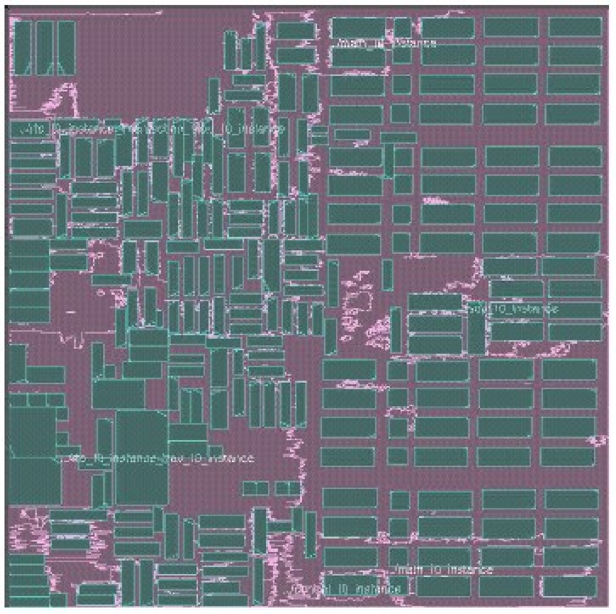
D-RPU Implementation

- Xilinx Virtex-4 LX160
  - 128 MB RAM, .5 GB/s @ 66 MHz
  - 7.5 GFLOP/s @ 24 bit
  - Usage: 99% logic, 60% memory
  - 32 threads per SPU 60% usage
  - Chunk size of 4 95% efficiency
  - 12 kB caches in total 90% hit rate
- Performance
  - 40-70% faster than OpenRT
  - OpenRT on CPU with 40x clock rate

➔ 60x „more efficient“



D-RPU Implementation



- D-RPU ASIC
  - Synthesized from HWML
    - With HW evaluation for clock rate
  - Larger caches (3x 16 KB)
    - 4-way associative
  - 130 nm process from UMC: 49 mm^2, 266 MHz
    - 30 GFLOP/s @ 32 bit (post-layout timing)
    - 2.1 GB/s required to external memory

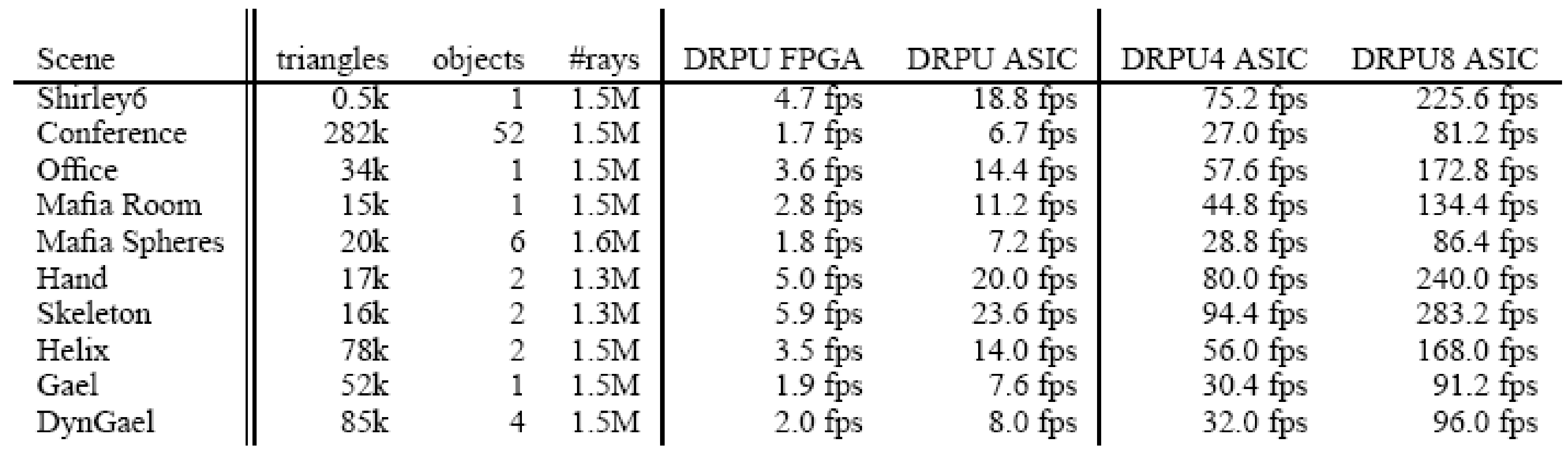
Projections

- ATI R-520: 288 mm² in 90 nm process

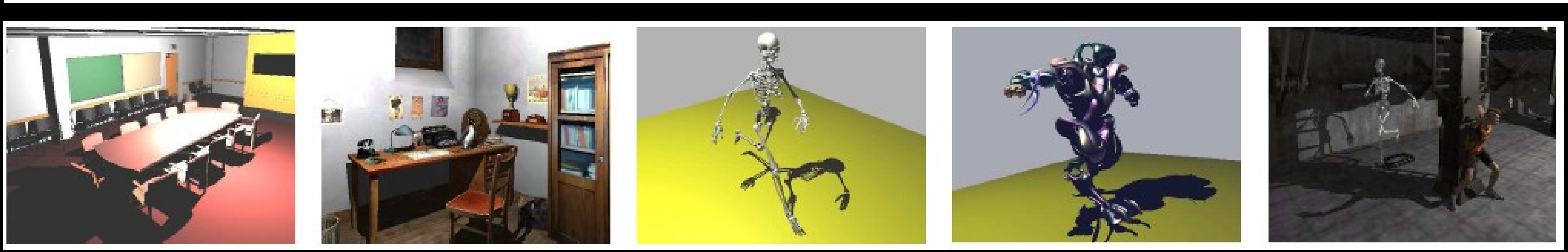
- D-RPU-4: 196 mm², 130 nm
  - 120 GFLOP/s @ 266 MHz (constant field scaling)
  - 8.5 GB/s (DDR2 memory?)
- D-RPU-8: 186 mm², 90 nm
  - 361 GFLOP/s @ 400 MHz (constant field scaling)
  - 25.6 GB/s (multi-channel DDR-2 or XDR memory)

Performance @ 1024 x 768

(shadows, full Phong shading, textures)



Scene	triangles	objects	#rays	DRPU FPGA	DRPU ASIC	DRPU4 ASIC	DRPU8 ASIC
Shirley6	0.5k	1	1.5M	4.7 fps	18.8 fps	75.2 fps	225.6 fps
Conference	282k	52	1.5M	1.7 fps	6.7 fps	27.0 fps	81.2 fps
Office	34k	1	1.5M	3.6 fps	14.4 fps	57.6 fps	172.8 fps
Mafia Room	15k	1	1.5M	2.8 fps	11.2 fps	44.8 fps	134.4 fps
Mafia Spheres	20k	6	1.6M	1.8 fps	7.2 fps	28.8 fps	86.4 fps
Hand	17k	2	1.3M	5.0 fps	20.0 fps	80.0 fps	240.0 fps
Skeleton	16k	2	1.3M	5.9 fps	23.6 fps	94.4 fps	283.2 fps
Helix	78k	2	1.5M	3.5 fps	14.0 fps	56.0 fps	168.0 fps
Gael	52k	1	1.5M	1.9 fps	7.6 fps	30.4 fps	91.2 fps
DynGael	85k	4	1.5M	2.0 fps	8.0 fps	32.0 fps	96.0 fps



Outlook:
Hardware for Ray Tracing

- Symmetric & Asymmetric Multi-Core CPUs
  - Current: ~10 Mrays/s (per core)
  - Future: many cores per chip, SHM
- High Performance Parallel GPUs
  - Not competitive (yet?), limited programming model
- Custom Ray Tracing Hardware
  - Current: 5-9 Mrays/s (FPGA, 66 MHz)
  - Future: >300 Mrays/s (ASIC, 285 MHz)

Interested?
Questions?

informatik
saarland.

Informatik Saarland	http://www.informatik-saarland.de
Computergraphik	http://graphics.cs.uni-sb.de
Ray Tracing	http://www.OpenRT.de
Direct Email	slusallek@cs.uni-sb.de
inTrace GmbH	http://www.inTrace.com info@inTrace.com