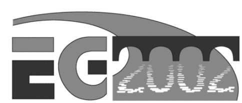
EUROGRAPHICS 2002



Tutorial T4: Programmable Graphics Hardware for Interactive Visualization

Thomas Ertl, VIS, Universität Stuttgart
Daniel Weiskopf, VIS, Universität Stuttgart
Martin Kraus, VIS, Universität Stuttgart
Klaus Engel, VIS, Universität Stuttgart
Manfred Weiler, VIS, Universität Stuttgart
Matthias Hopf, VIS, Universität Stuttgart
Stefan Röttger, VIS, Universität Stuttgart
Christof Rezk-Salama, IMMD9, Universität Erlangen

Published by
The Eurographics Association
ISSN 1017-4565

The European Association for Computer Graphics
23rd Annual Conference

EUROGRAPHICS 2002

Saarbrücken, Germany
September 2–6, 2002

Organized by

EUROGRAPHICS
THE EUROPEAN ASSOCIATION
FOR COMPUTER GRAPHICS

INFORMATIK
Max-Planck-Institut
für Informatik
Saarbrücken, Germany

Universität des Saarlandes
Germany

International Programme Committee Chairs
George Drettakis (France)
Hans-Peter Seidel (Germany)

Conference Co-Chairs
Frits Post (The Netherlands)
Dietmar Saupe (Germany)

Honorary Conference Co-Chairs
Jose Encarnação (Germany)
Wolfgang Straßer (Germany)

Tutorial Chairs
Sabine Coquillart (France)
Heinrich Müller (Germany)

STAR Report Chairs
Dieter Fellner (Germany)
Roberto Scopignio (Italy)

Lab Presentation Chairs
Günther Greiner (Germany)
Werner Purgathofer (Austria)

Industrial Seminar Chairs
Thomas Ertl (Germany)
Bernd Kehrer (Germany)

Günter Enderle Award Committee Chair
François Sillion (France)

Conference Game Chair
Nigel W. John (UK)

John Lansdown Award Chair
Huw Jones (UK)

Conference Director
Christoph Storb (Germany)

Short/Poster Presentation Chairs
Isabel Navazo (Spain)
Philipp Slusallek (Germany)

Local Organization
Annette Scheel (Germany)
Hartmut Schirmacher (Germany)

Eurographics 2002
Tutorial T4
2. September 2002

Programmable Graphics Hardware for Interactive Visualization

Visualization and Interactive Systems Group
University of Stuttgart, Germany
Thomas Ertl

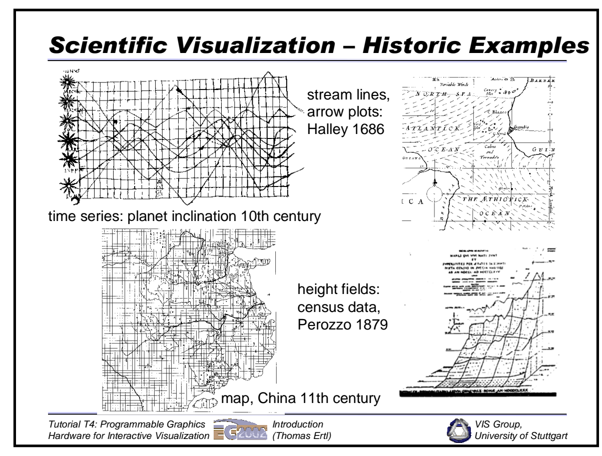
Overview of the Tutorial - Morning

09.00 – 09.30	Introduction to the Tutorial	Thomas Ertl
09.30 – 10.30	Introduction to Programmable Graphics Hardware	Martin Kraus
10.30 – 11.00	*Coffee Break*	All
11.00 – 11.45	Hardware-Accelerated Volume Rendering for Rectilinear Grids	Christof Rezk-Salama
11.45 – 12.30	Volume Graphics on Consumer PC Hardware	Klaus Engel
13.30 – 14.00	*Lunch Break*	All

Overview of the Tutorial - Afternoon

14.00 – 14.20	Pre-Integrated Cell Projection	Stefan Röttger
14.20 – 14.40	Hardware-Based Cell Projection	Manfred Weiler
14.40 – 15.00	Hardware-Accelerated Terrain Rendering by Adaptive Slicing	Stefan Röttger
15.00 – 15.30	Visualization of 2D Flow Fields by Texture Advection	Daniel Weiskopf
15.30 – 16.00	Coffee *Break*	All
16.00 – 16.40	Interactive NPR of Technical Illustrations	Daniel Weiskopf
16.40 – 17.10	Hardware-Accelerated Filtering	Matthias Hopf
17.10 – 17.30	Texture Compression	Martin Kraus

Scientific Visualization – Historic Examples



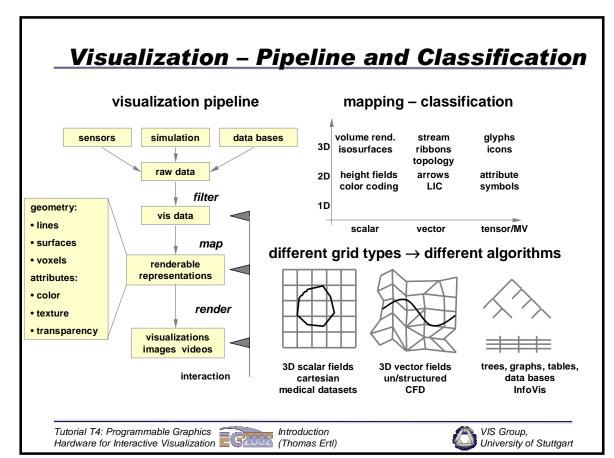
Modern Scientific Visualization

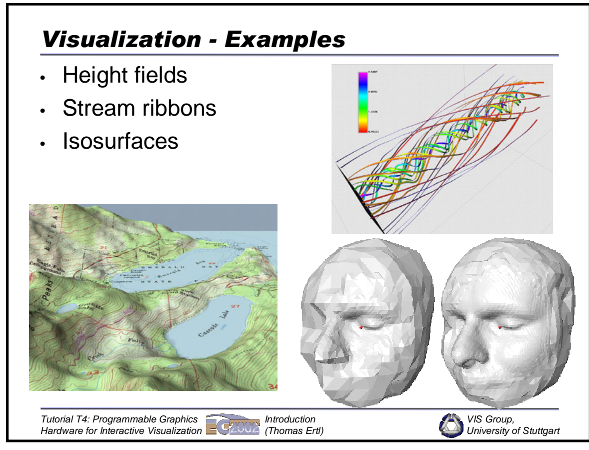
- Traditional plotting techniques are not appropriate for visualizing the huge datasets resulting from
  - computer simulations (e.g. CFD, physics, chemistry, ...)
  - sensoric measurements (e.g. medical, seismic, satellite)

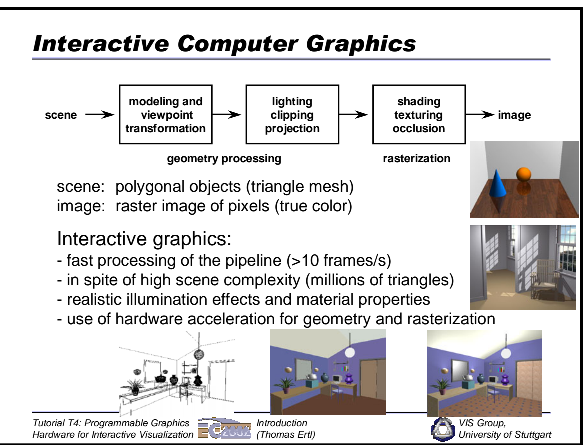
  „The purpose of computing is insight not numbers“
- Map abstract data onto graphical representations
- Try to use colorful 3D raster graphics in
  - expressive still images
  - recorded animations
  - interactive visualizations

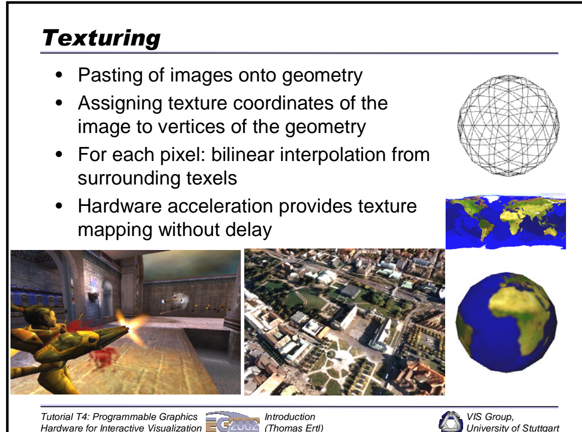
  „To see the unseen“

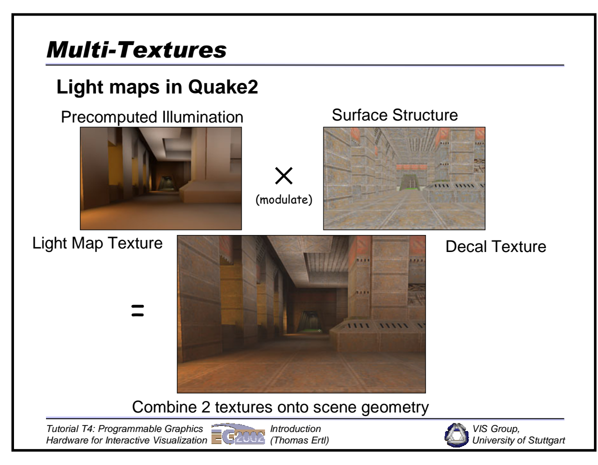
Visualization – Pipeline and Classification

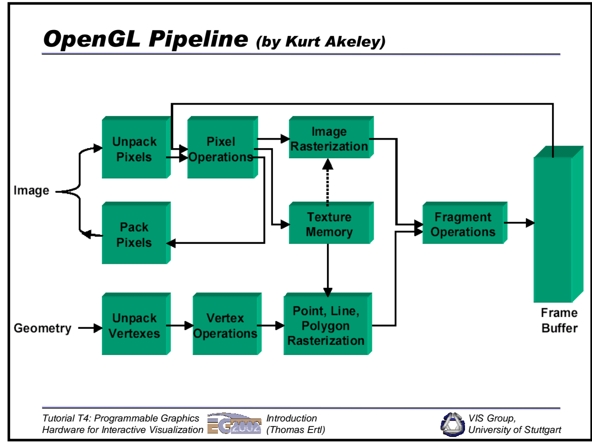












Graphics Hardware Characteristics

- **Performance characteristics**
  - Geometry: shaded triangles per second >> 10 Mio
  - Rasterization: fill rate in pixels per second >> 100 Mio
- **Computational requirements: geometry subsystem**
  - ca. 100 FLOPs per vertex (about 30 for T&L each)
  - 10 Mio. triangles/s T&L performance need 3 GigaFLOPs however only 500.000 triangles in the scene at 20 Hz!
- **Computational requirements: raster subsystem**
  - >10 operations per pixel (without special texturing!)
  - 100 MegaPixel/s fill rate need 1000 MIPS performance
  - at 20Hz and 10 pixel/triangle: 500.000 tris per frame
  - for a 1Kx1K frame buffer 5-fold overdraw of each pixel

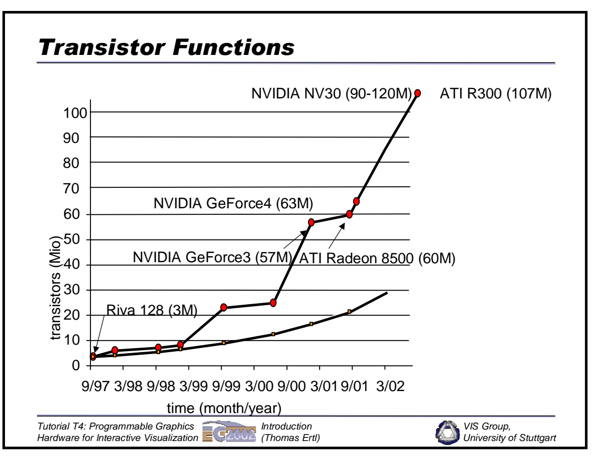
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Introduction (Thomas Ertl)
VIS Group, University of Stuttgart

Graphics Hardware Trend

- Faster development than Moore's law
  - Double transistor functions every 6-12 months
  - Driven by Game industry
- Improvement of performance and functionality
  - Textures, Multi-textures, texture shaders
  - Pixel operations (transparency, blending, pixel shaders)
  - Geometry and lighting modifications (vertex shaders)

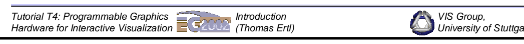
performance
graphics
CPU
network
time

Transistor Functions



20 Years of Graphics Hardware

- **1980s**: Simple rasterization (bitBLT, windows, lines, polygons, text fonts)
- **1990-95:** Geometry engines only for high-end workstations (e.g. SGI O2 vs. Indigo2)
- **1995:** New rasterization functionality (realism with textures) z.B: SGI Infinite Reality
- **1998:** Geometry processing (T&L) for PC graphics cards
- **2000:** PC graphics reaches high-end performance numbers, 3D becomes PC standard
- **2001:** PC graphics offers additional functionality (multi-texturing, vertex and pixel shaders)
- **2002:** Shading Languages: OpenGl 2.0, NVIDIA Cg, DX9 GPUs > 100 Mio. transistors, 8 Pipes and 16 texture units



From Configuration to Programming

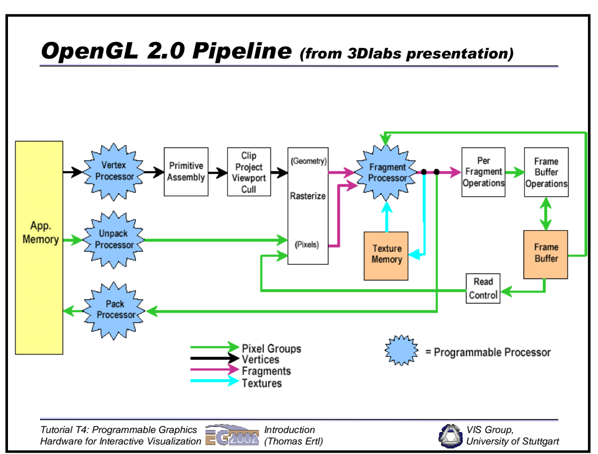
- **Configurability**:
  Select hardware processing options by state changes
  - T&L: various texture generation modes
  - Rasterization: imaging subset
  - Fragment processing: various blending modes
- **Programmability**:
  Download small assembly programs to change hardware behavior
  - T&L: vertex shaders
  - Rasterization: texture shaders
  - Fragment processing: pixel and fragment shaders

Programmable Processors *(from NVIDIA Cg Manual)*

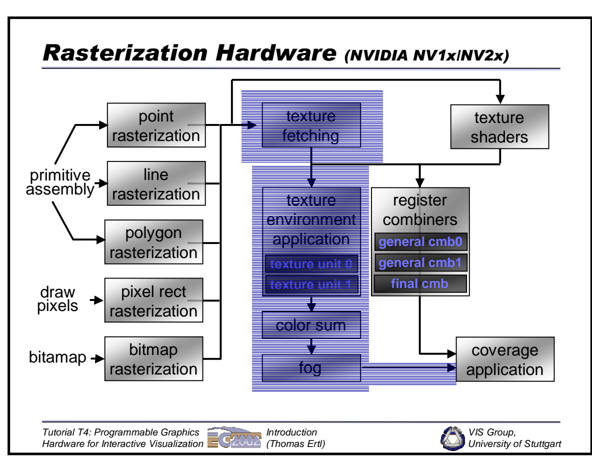
- 2 or more programmable processors per GPU
- Fixed pipeline (with configuration) remains where no flexibility is necessary (or possible)

Matrices, light positions, blend factors, and other uniform parameters
CPU/ Application
Programmable Vertex Processor
Primitive Assembly
Rasterization & Interpretation
Programmable Fragment Processor
Framebuffer Tests & Blending
Vertex Indices
Textures
Framebuffer
Data Flows: Primitive, Vertex, and Fragment Data
Uniform Parameters -- Changes infrequent

OpenGL 2.0 Pipeline *(from 3Dlabs presentation)*

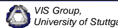


Rasterization Hardware *(NVIDIA NV1x/NV2x)*

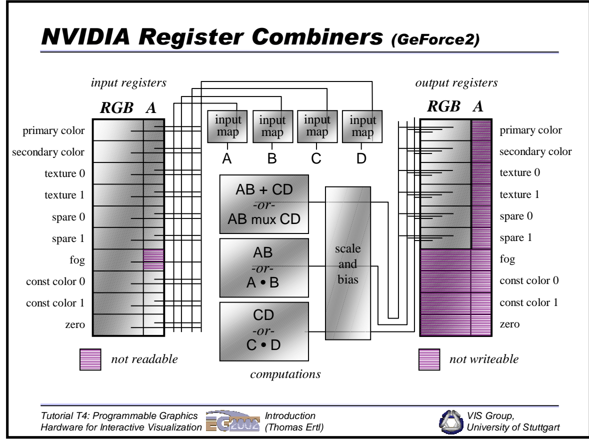


Texture Shaders & Register Combiners

- **Texture shaders**
  (ca. 40 texture programs in NVIDIA GeForce4)
  - Offset textures
  - Dependent textures
  - Dot Product textures
  - 3D textures
  - Up to 4 texture stages
- **Register combiners**
  (8+1 combiners in NVIDIA GeForce4)
  - Simple arithmetic (sum, mult, dot product)
  - Scale, bias, input mappings



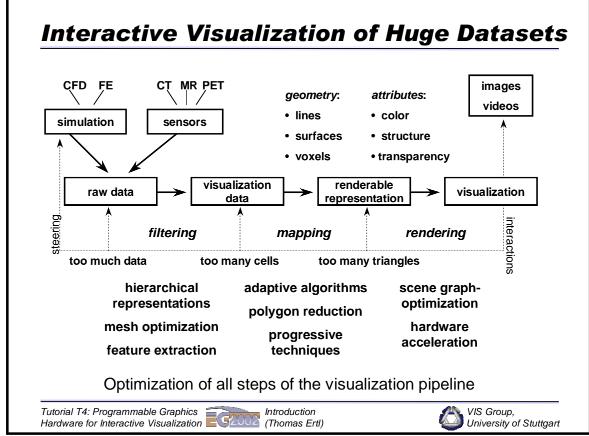
NVIDIA Register Combiners *(GeForce2)*



Vertex Shaders

- **Programmable transformation & lighting**
  - Register architecture with up to 128 instructions
  - Replaces standard transformation pipeline and Phong lighting
  - Special perspective projections (lens effects)
  - Advanced lighting models
  - Automatic generation of texture coordinates
  - Procedural geometry, morphing, skinning, ...

Interactive Visualization of Huge Datasets



Graphics HW and Interactive Visualization

- **First**: Mapping generates polygonal geometry only, colored, lighted and shaded (e.g. isosurfaces, stream ribbons, glyphs)
- **From 1995**: Advanced rasterization functionality, textures and transparency (e.g.LIC, volume rendering)
- **From 2000**: Multi-textures and register combiners
- **From 2002**: Texture shaders and vertex shaders
- **In the future**: Shading languages for visualization
- **Trend**: Graphics hardware on its way up through the visualization pipeline towards the data

Images $\rightarrow$ Renderer $\Rightarrow$ Mapper $\Rightarrow$ Filter $\rightarrow$ Data

Graphics HW and VIS Pipeline Stages

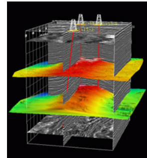
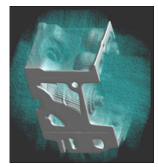
- **Renderer**
  - Texture based techniques (3D textures, LIC, ...)
  - Large textured terrain height fields
- **Mapper**
  - Classification & transfer functions in volume rendering
  - Integrate ray segments (in unstructured volumes)
  - Integrate particle traces (in flow fields)
  - Assign color and transparency for NPR
- **Filtering**
  - Data filtering in graphics memory (e.g. wavelet)
  - Compression/decompression (of textures)

Prog. Graphics HW and VIS Applications

- End users of VIS still use classical Unix workstations (no programmable graphics HW)
- VIS applications (pre- & post processing, toolkits, MVEs) are cross-platform, use minimum funct.
- Texturing and transparency are „advanced“
- Exception: volume rendering
  - Doctors can afford PCs, no Unix workstations
  - Regular data structures profit most
  - Improvements are significant

Volume Visualization

- Abstract 3-dimensional datasets
  - X-ray absorption in material
  - humidity in the atmosphere
  - density distribution in the earth
- Data often given on uniform 3D grid millions of cells (voxel)
- Problem: occlusion

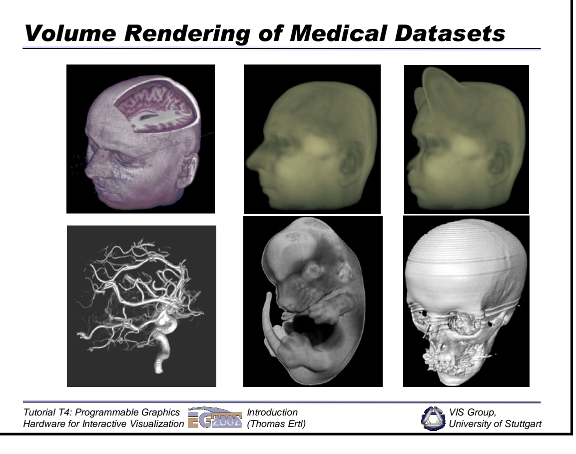
Volume Visualization

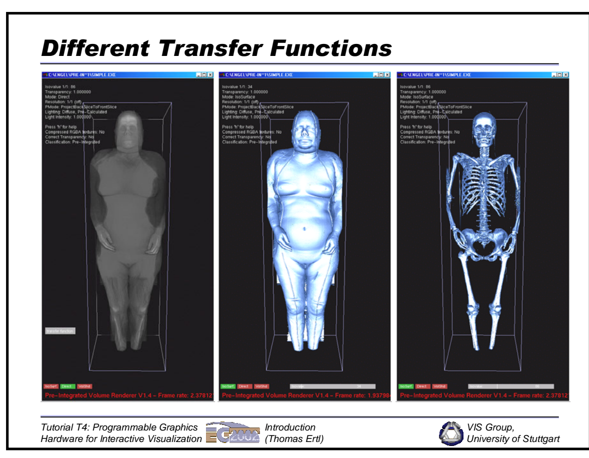
- Focus on 3D scalar fields (e.g. medical data)
  some concepts extend to non-cartesian grids, vector fields,...
- Isosurfaces
  - reconstruction of polygonal surfaces with Marching Cubes
  - fast rendering with OpenGL standard hardware
  - non-interactive for huge datasets (millions of triangles)
- Direct volume rendering
  - for each pixel send a ray into the volume
  - sample volume along ray by interpolation
  - semi-transparent blending along rays
  - transfer functions for color and opacity provide „segmentation“ of structures
  - interactivity even for many trilinear interpolations with hardware support (dedicated or 3D textures)

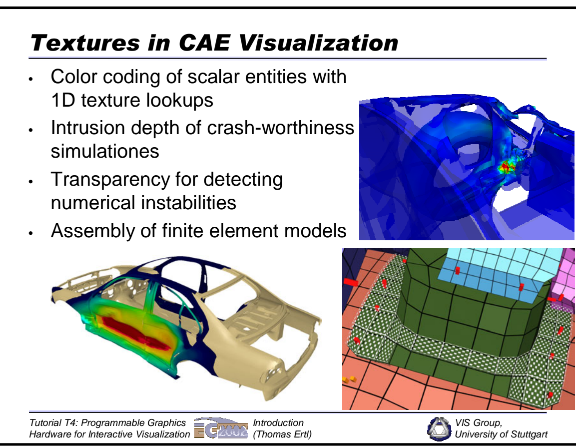
Volume Visualization of Medical Datasets

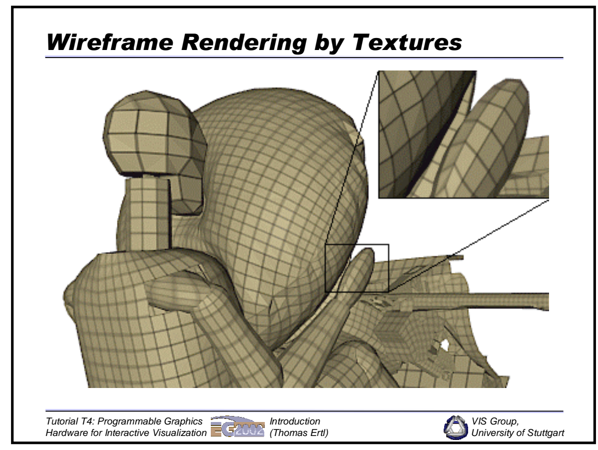
- 2D visualization slice images (MPR)
- *Indirect* 3D visualization isosurfaces (SSD)
- *Direct* 3D visualization volume rendering (DVR)

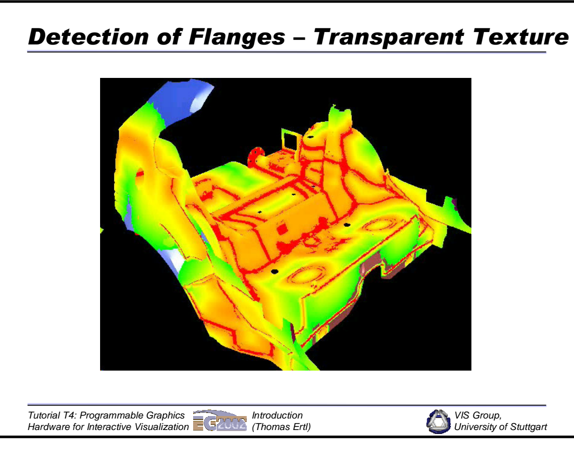
Volume Rendering of Medical Datasets

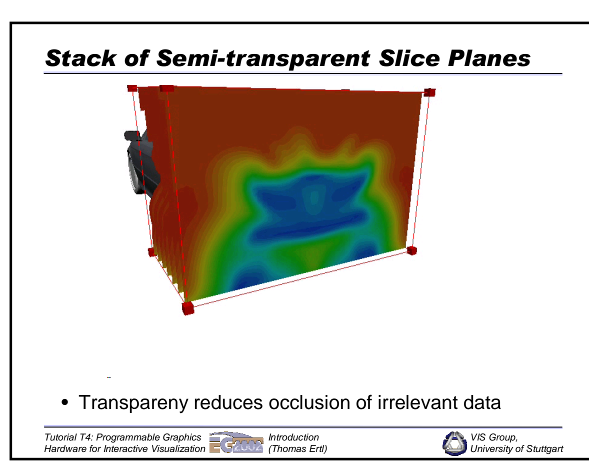


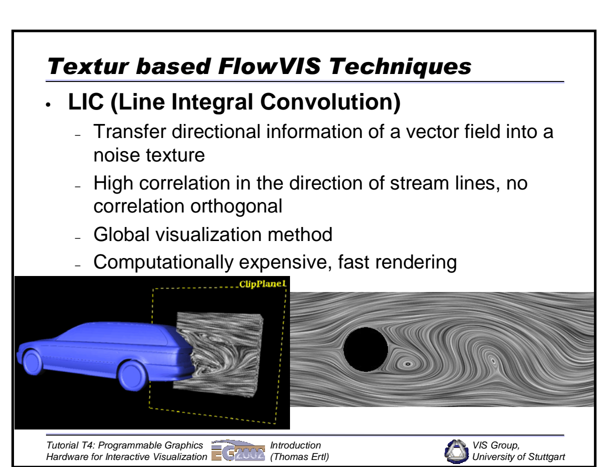

Different Transfer Functions
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Introduction (Thomas Ertl)
VIS Group, University of Stuttgart


Textures in CAE Visualization
Color coding of scalar entities with 1D texture lookups
Intrusion depth of crash-worthiness simulations
Transparency for detecting numerical instabilities
Assembly of finite element models
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Introduction (Thomas Ertl)
VIS Group, University of Stuttgart


Wireframe Rendering by Textures
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Introduction (Thomas Ertl)
VIS Group, University of Stuttgart


Detection of Flanges – Transparent Texture
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Introduction (Thomas Ertl)
VIS Group, University of Stuttgart


Stack of Semi-transparent Slice Planes
Transpareny reduces occlusion of irrelevant data
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Introduction (Thomas Ertl)
VIS Group, University of Stuttgart


Textur based FlowVIS Techniques
LIC (Line Integral Convolution)
Transfer directional information of a vector field into a noise texture
High correlation in the direction of stream lines, no correlation orthogonal
Global visualization method
Computationally expensive, fast rendering
ClipPlane1
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Introduction (Thomas Ertl)
VIS Group, University of Stuttgart

Introduction to Programmable Graphics Hardware

Martin Kraus

Visualization and Interactive Systems Group, Stuttgart

Overview

Contents:

- Programmability of graphics hardware
- Per-vertex operations
- Per-pixel operations
- Outlook

Programmability of Graphics Hardware

Contents:

- **Programmability of graphics hardware**
  - **Programming vs. configuring**
  - **Current hardware**
  - **Low-Level APIs**
  - **High-Level APIs**
- Per-vertex operations
- Per-pixel operations
- Outlook

Programming vs. Configuring

- Standard OpenGL pipeline:
  - *Transform & lighting* (T&L) (per vertex),
  - Rasterization setup (per primitive),
  - *Texturing* (per fragment),
  - Fragment tests (per fragment),
  - Blending with destination (per pixel).
- Transform & lighting:
  - Clear difference between configuring the standard pipeline and programming your own computations.
- Texturing:
  - Seamless transition from configuring multi-textures to programming multi-instruction texturing.

Current Hardware

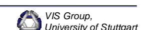
- NVIDIA's nv1x (GeForce256/2)
  - Programmable texture blending (register combiners).
- NVIDIA's nv2x (GeForce 3/4): nfiniteFX engine
  - Programmable T&L (vertex programs).
  - Programmable texturing (texture shader + register combiners).
- ATI's r200 (Radeon 8500): smartshader
  - Programmable T&L (vertex shader).
  - Programmable texturing (fragment shader).
- Many announcements (NVIDIA: nv30, ATI: r300, 3Dlabs: P10, Matrox: Parhelia)

Low-Level APIs

- Microsoft's DirectX (Direct3D):
  - DirectX 8.0: Vertex Shader 1.1, Pixel Shader 1.0,1.1
  - DirectX 8.1: Vertex Shader 1.1, Pixel Shader 1.2-1.4
  - DirectX 9.x: Vertex Shader 2.0, Pixel Shader 2.0
- OpenGL extensions:
  - NVIDIA:
    - GL_NV_vertex_program(1_1)
    - GL_NV_texture_shader(2,3), GL_NV_register_combiners(2),
  - ATI:
    - GL_EXT_vertex_shader
    - GL_ATI_fragment_shader

High-Level APIs

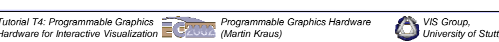
- High-level shading languages:
  - Pixar's PhotoRealistic RenderMan,
  - Michael McCool's SMASH API,
  - Stanford real-time shading language,
  - NVIDIA's Cg (and nvparse),
  - DirectX 9 shading language,
  - OpenGL 2.0 shading language.
- Why should we bother with low-level APIs?
  - Low-level APIs offer best performance & functionality.
  - Help to understand the graphics hardware.
  - Help to understand high-level APIs!



Per-Vertex Operations

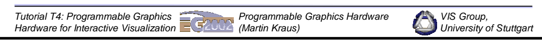
Contents:

- Programmability of graphics hardware
- **Per-vertex operations**
  - **What?**
  - **How?**
    - **DirectX 8: Vertex Shader 1.1**
    - **NVIDIA's OpenGL Extensions**
    - **ATI's OpenGL Extension**
  - **Summary**
- Per-pixel operations
- Outlook


Tutorial T4: Programmable Graphics Hardware for Interactive Visualization Programmable Graphics Hardware (Martin Kraus) VIS Group, University of Stuttgart

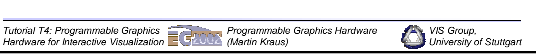
Per-Vertex Operations: What?

- Replace standard vertex transform & lighting by user-defined per-vertex computations, e.g. for
  - modified perspective projection (lenses etc.),
  - advanced lighting,
  - texture coordinate generation,
  - vertex skinning, blending, morphing,
  - procedural geometry.
- In this tutorial:
  - Computations for cell projection (third session).
  - Non-photo-realistic rendering (fourth session).


Tutorial T4: Programmable Graphics Hardware for Interactive Visualization Programmable Graphics Hardware (Martin Kraus) VIS Group, University of Stuttgart

Per-Vertex Operations: What Not?

- Current limitations:
  - No inter-vertex dependencies.
  - Limited number of instructions, registers, etc.
  - No vertex (or primitive) generation.
  - No jumps or loops.
  - No memory access.

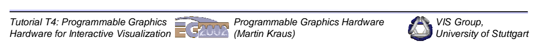

Tutorial T4: Programmable Graphics Hardware for Interactive Visualization Programmable Graphics Hardware (Martin Kraus) VIS Group, University of Stuttgart

Per-Vertex Operations: How?

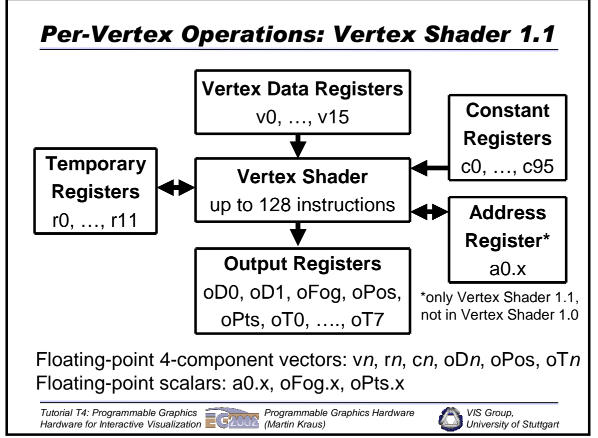
- There are three APIs exposing *very similar* functionality:

	NVIDIA GeForce 3 and 4	ATI Radeon 8500
DirectX 8	Vertex Shader 1.1	Vertex Shader 1.1
OpenGL ext.	GL_NV_vertex_program	GL_EXT_vertex_shader

- Note:
  - For Vertex Shader 1.1 and GL_NV_vertex_program vertex operations are specified with ASCII text.
  - For GL_EXT_vertex_shader vertex operations are specified with OpenGL function calls.


Tutorial T4: Programmable Graphics Hardware for Interactive Visualization Programmable Graphics Hardware (Martin Kraus) VIS Group, University of Stuttgart

Per-Vertex Operations: Vertex Shader 1.1



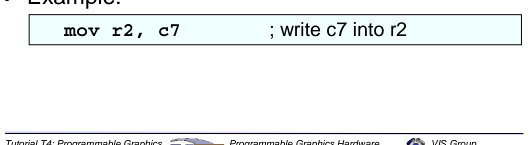
Floating-point 4-component vectors: v*n*, r*n*, c*n*, oD*n*, oPos, oT*n*
Floating-point scalars: a0.x, oFog.x, oPts.x

Tutorial T4: Programmable Graphics Hardware for Interactive Visualization Programmable Graphics Hardware (Martin Kraus) VIS Group, University of Stuttgart

Per-Vertex Operations: Vertex Shader 1.1

- Input Registers:
  - Vertex Registers v0, …, v15 (vectors, read-only)
  - Constant Registers c0, …, c95 (vectors, read-only)
  - Address Register a0 (scalar, read/use-only)
    - Used for relative addressing of constant registers `c[a0.x+n]`
  - Temporary Registers r0, …, r11 (vectors, read/write)
- Example:

```
mov r2, c7          ; write c7 into r2
```



Per-Vertex Operations: Vertex Shader 1.1

- Output Registers:
  - Position Register oPos (vector)
  - Vertex Color Registers oD0, oD1 (vectors)
    Diffuse and specular vertex color.
  - Texture Coordinate Registers oT0,…, oT7 (vectors)
  - Point Size Register oPts (scalar, only x component)
  - Fog Register oFog (scalar, only x component)
    Fog factor, routed to fog table.
- Example:

```
mov oD0, c5         ; write c5 into oD0
```

Tutorial T4: Programmable Graphics Hardware for Interactive Visualization | Programmable Graphics Hardware (Martin Kraus) | VIS Group, University of Stuttgart

Per-Vertex Operations: Vertex Shader 1.1

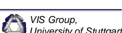
- Modifiers for components of a register *r:*
  - Destination mask: *r*.[x][y][z][w]
  - Source swizzle: *r*.[xyzw][xyzw][xyzw][xyzw]
    - *r.c* is equivalent to *r.cccc*
  - Source negation: -*r*
- Examples:

```
mov r2.xz, c7       ; write c7.x into r2.x and
                    // c7.z into r2.z
mov r2, c7.wzzx     ; write c7.w to r2.x,
                    // c7.z to r2.y and r2.z,…
mov r2, -c7         ; write -c7 into r2
```

Tutorial T4: Programmable Graphics Hardware for Interactive Visualization | Programmable Graphics Hardware (Martin Kraus) | VIS Group, University of Stuttgart

Per-Vertex Operations: Vertex Shader 1.1

- Instruction set:
  - Version instructions:
    `vs, def`
  - General instructions:
    `mov, add, sub, mul, mad, dp3, dp4,`
    `rcp, rsq, dst, lit, expp, logp,`
    `max, min, sge, slt`
  - Macros:
    `exp, log, frc,`
    `m3x2, m3x3, m3x4, m4x3, m4x4`



Tutorial T4: Programmable Graphics Hardware for Interactive Visualization | Programmable Graphics Hardware (Martin Kraus) | VIS Group, University of Stuttgart

Per-Vertex Operations: Vertex Shader 1.1

- Version instruction (first instruction):

```
vs.1.1
```

- Definition of constants:

```
def cn, float0, float1, float2, float3
```

  - After version instruction, before all other instructions.
  - Or use the function `SetVertexShaderConstant.`
- Moving values:

```
mov dest, src       ; dest = src
```

Tutorial T4: Programmable Graphics Hardware for Interactive Visualization | Programmable Graphics Hardware (Martin Kraus) | VIS Group, University of Stuttgart

Per-Vertex Operations: Vertex Shader 1.1

- Addition, subtraction, and multiplication:

```
add dest, src0, src1  ; dest = src0 + src1
sub dest, src0, src1  ; dest = src0 - src1
mul dest, src0, src1  ; dest = src0 * src1
mad dest, src0, src1, src2 ; dest =src0*src1+src2
```

  - Add/subtract/multiply corresponding components.
- 3-component and 4-component dot products:

```
dp3 dest, src0, src1  ; dest = src0 • src1 (x,y,z)
dp4 dest, src0, src1  ; dest = src0 • src1 (x,y,z,w)
```

  - Set all components to the scalar dot product.

Tutorial T4: Programmable Graphics Hardware for Interactive Visualization | Programmable Graphics Hardware (Martin Kraus) | VIS Group, University of Stuttgart

Per-Vertex Operations: Vertex Shader 1.1

- Reciprocals of scalars:

```
rcp dest, src ; dest = 1 / w of src
rsq dest, src ; dest = 1 / sqrt(w of src)
```

  - Set all components to the scalar result.
  - Result is plus infinity if *src* is 0.
- Partial support for computations:

```
dst  dest, src0, src1 ; distance computation
lit  dest, src        ; lighting computation
expp dest, src        ; powers of 2
logp dest, src        ; logarithm to base 2
```

  - See Vertex Shader documentation.

Per-Vertex Operations: Vertex Shader 1.1

- Conditional instructions:

```
max dest, src0, src1; dest = max. of src0 and src1
min dest, src0, src1; dest = min. of src0 and src1
sge dest, src0, src1; dest = 1 if src0>=src1, else 0
slt dest, src0, src1; dest = 1 if src0 < src1, else 0
```

  - Compare and set corresponding components.

Per-Vertex Operations: Vertex Shader 1.1

- Macro scalar instructions:

```
exp dest, src ; dest = 2 to the power of w of src
log dest, src ; dest = log. to base 2 of w of src
```

  - Set all components to the scalar result.

```
frc dest, src ; dest = fractional portion of src
```

  - Sets only x and y components to correspond. results.

Per-Vertex Operations: Vertex Shader 1.1

- Macro matrix instructions:

```
m3x2 dest, src0, src1 ;dest=matrix3x2(src1)•src0
m3x3 dest, src0, src1 ;dest=matrix3x2(src1)•src0
m3x4 dest, src0, src1 ;dest=matrix3x4(src1)•src0
m4x3 dest, src0, src1 ;dest=matrix4x3(src1)•src0
m4x4 dest, src0, src1 ;dest=matrix4x4(src1)•src0
```

  - *src1* has to be a constant register; rows of the matrix are taken from *src1* and following registers.
  - Example: `m4x3 r2, v0, c4` is expanded to:

```
dp4 r2.x, v0, c4
dp4 r2.y, v0, c5
dp4 r2.z, v0, c6
```


Per-Vertex Operations: Vertex Shader 1.1

Summary for DirectX 8 Vertex Shader 1.1:

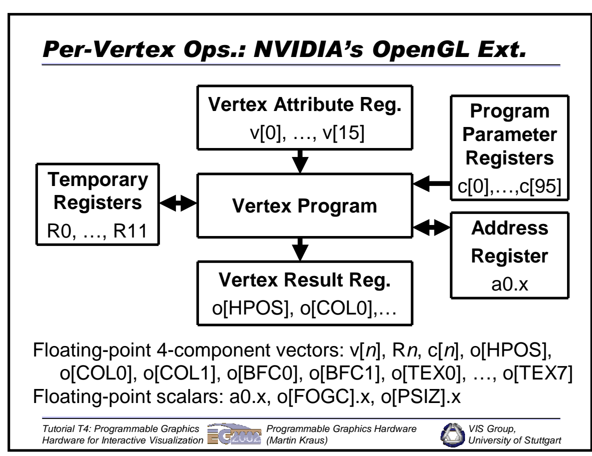
- Not discussed here:
  - Many details. (Restrictions, etc.)
  - Compiling, debugging, activating vertex shaders.
  - Applications. (See third and fourth session.)
- Advantages :
  - Well supported by graphics hardware vendors.
  - Comfortable coding.
- Disadvantages:
  - Not as platform-independent as OpenGL.

Per-Vertex Ops.: NVIDIA's OpenGL Ext.

- NVIDIA's OpenGL extensions:
  - GL_NV_vertex_program(1_1)
  - Nomenclature:

DirectX 8 Vertex Shader 1.1	**NVIDI's OpenGL extensions:**
vertex shader	vertex program
vertex data registers	vertex attribute registers
constant registers	program parameter reg.
output registers	vertex result registers

Per-Vertex Ops.: NVIDIA's OpenGL Ext.



Floating-point 4-component vectors: v[*n*], R*n*, c[*n*], o[HPOS], o[COL0], o[COL1], o[BFC0], o[BFC1], o[TEX0], …, o[TEX7]
Floating-point scalars: a0.x, o[FOGC].x, o[PSIZ].x

Per-Vertex Ops.: NVIDIA's OpenGL Ext.

- Vertex attribute registers:

register	mnemonic	
v[0]	v[OPOS]	position
v[1]	v[WGHT]	weight
v[2]	v[NRML]	normal
v[3]	v[COL0]	primary color
v[4]	v[COL1]	secondary color
v[5]	v[FOGC]	fog coordinate
v[8], ..., v[15]	v[TEX0], ..., v[TEX7]	texture coordinates 0,..., 7

Per-Vertex Ops.: NVIDIA's OpenGL Ext.

- Vertex result registers:

		V.S. 1.1
homogen. clip space position	o[HPOS]	oPos
primary color (front face)	o[COL0]	oD0
secondary color (front face)	o[COL1]	oD1
primary color (back face)	o[BFC0]	
secondary color (back face)	o[BFC1]	
fog coordinate	o[FOGC]	oFog
point size	o[PSIZ]	oPts
texture coordinate set	o[TEX*n*]	oT*n*

- Modifiers: same syntax and semantics as Vertex Shader (destination mask, swizzle, negate)

Per-Vertex Ops.: NVIDIA's OpenGL Ext.

- Instructions:
  - Similar instructions as in Vertex Shader: `MOV, ADD, MUL, MAD, DP3, DP4, RCP, RSQ, DST, LIT, EXP, LOG, MAX, MIN, SGE, SLT`
  - New in GL_NV_vertex_program1_1: `SUB`, `ABS` (absolute value), `DPH` (homogeneous dot product), `RCC` (clamped reciprocal)
  - Syntax: "`;`" for end of instruction, "`#`" for comments
  - Write to `A0.x` not with `MOV`, but with:

```
ARL A0.x, src;        # vertex program
```


Per-Vertex Ops.: NVIDIA's OpenGL Ext.

Summary for GL_NV_vertex_program:

- Not discussed here:
  - Creating, managing, optimizing vertex programs.
  - Specifying vertex attributes and program parameters.
  - Vertex State Programs
- Advantage:
  - Platform-independent OpenGL extension.
- Disadvantage:
  - Single-vendor OpenGL extension.

Per-Vertex Ops.: ATI's OpenGL Extension

- ATI's OpenGL extension for per-vertex ops.:
  - GL_EXT_vertex_shader
  - Nomenclature:

DirectX 8: Vertex Shader 1.1	ATI's OpenGL extension GL_EXT_vertex_shader
instructions	operations
vertex data registers	variants
constant registers	invariants / local constants
temporary registers	locals
output registers	outputs
address register	index (different usage)

Per-Vertex Ops.: ATI's OpenGL Extension

- Output data:

		V.S.1.1
vertex position	OUTPUT_VERTEX	oPos
primary color	OUTPUT_COLOR0	oD0
secondary color	OUTPUT_COLOR1	oD1
fog coordinate	OUTPUT_FOG	oFog
texture coordinate set	OUTPUT_TEXTURE_ COORD*n*	oT*n*

- Modifiers: same semantics as Vertex Shader (destination mask, swizzle, negate)

Per-Vertex Ops.: ATI's OpenGL Extension

- Instruction set:
  - Similar instructions as in Vertex Shader:
    OP_MOV, OP_ADD, OP_SUB, OP_MUL, OP_MADD, OP_DOT3, OP_DOT4, OP_RECIP, OP_RECIP_SQRT, OP_EXP_BASE_2, OP_LOG_BASE_2, OP_FRAC, OP_MAX, OP_MIN, OP_SET_GE, OP_SET_LE, OP_MULTIPLY_MATRIX (= m4x4)
  - Different instructions:
    OP_INDEX, OP_NEGATE, OP_CLAMP, OP_FLOOR, OP_ROUND, OP_POWER, OP_CROSS_PRODUCT

Per-Vertex Ops.: ATI's OpenGL Extension

- Input data for operations has to be declared with **glGenSymbolsEXT, glSetInvariantEXT, glSetLocalConstantEXT, glVariant…EXT, glVariantPointerEXT**
- Specification of operations with

```
glShaderOp1EXT(op, dest, src1);
glShaderOp2EXT(op, dest, src1, src2);
glShaderOp3EXT(op, dest, src1, src2, src3);
glSwizzleEXT(dest, src, outX, outY, outZ);
glWriteMaskEXT(dest, src, outX, outY, outZ);
```

- Many calls, thus rather cumbersome to program.

Per-Vertex Ops.: ATI's OpenGL Extension

Summary for GL_EXT_vertex_shader:

- Not discussed here:
  - Creating and managing vertex shaders.
  - Specifying input data.
- Advantage:
  - Platform-independent multi-vendor OpenGL extension
- Disadvantage:
  - Many OpenGL calls even for small vertex shaders.

Per-Vertex Operations: Summary (1)

- Motivation:
  Replace standard transform & lighting by user-defined per-vertex computations.
- Hardware:
  - NVIDIA GeForce 3/4
  - ATI Radeon 8500
- APIs:
  - DirectX 8: Vertex Shader 1.1
  - OpenGL extensions:
    - GL_NV_vertex_program (NVIDIA)
    - GL_EXT_vertex_shader (ATI)

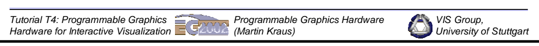
Per-Vertex Operations: Summary (2)

- Why shouldn't you use programmable T&L?
  - You do not need any particular per-vertex operations.
  - You need too complex per-vertex operations.
  - Per-vertex operations are not your bottleneck.
- Why should you use programmable T&L?
  - Exploit specialized hardware: GPU might be faster than CPU.
  - Exploit parallelism: free CPU for other tasks.
  - Avoid bandwidth limitations by computing data in GPU
- Examples will follow in the third and fourth session of this tutorial.

Per-Pixel Operations

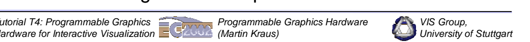
Contents:

- Programmability of graphics hardware
- Per-vertex operations
- **Per-pixel operations**
  - **What?**
  - **How?**
    - **NVIDIA's OpenGL extensions**
    - **DirectX 8.1: Pixel Shader 1.3 and 1.4**
    - **ATI's OpenGL extension**
  - **Summary**
- Outlook



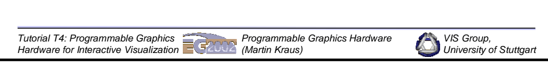
Per-Pixel Operations: What?

- Replace standard fragment texturing and texture blending by user-defined operations for
  - **any per-pixel computation**, e.g.
  - per-pixel lighting, reflection, bump-mapping, environment-mapping, shadow calculations, …
- In this tutorial:
  - Volume visualization,
  - Terrain rendering,
  - Flow visualization,
  - Non photo-realistic rendering,
  - Data filtering and decompression.



Per-Pixel Operations: What Not?

- Current limitations:
  - No single-pass inter-pixel dependencies.
  - No control over rasterization position.
  - Strongly limited number of instructions, registers, texture look-ups, etc.
  - Blending with destination is not programmable.
  - No jumps or loops.

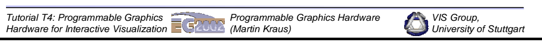


Per-Pixel Operations: How?

- There are four important APIs:

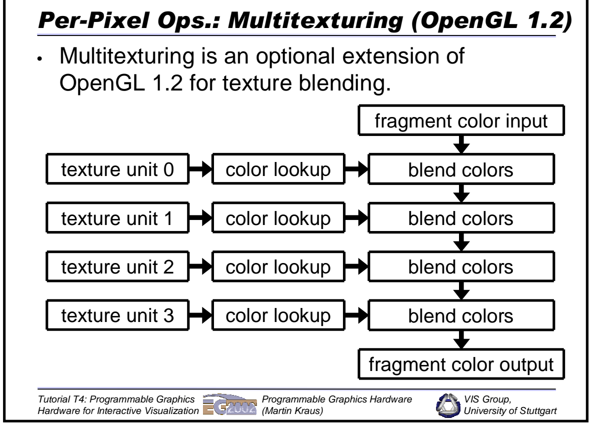
	NVIDIA GeForce 4	**ATI Radeon 8500**
DirectX 8.1	Pixel Shader 1.3	Pixel Shader 1.4
OpenGL extensions	GL_NV_texture_shader3, GL_NV_register_combiners2	GL_ATI_fragment_shader

- Note:
  - NVIDIA GeForce 3 is limited to Pixel Shader 1.2 and GL_NV_texture_shader2, respectively.



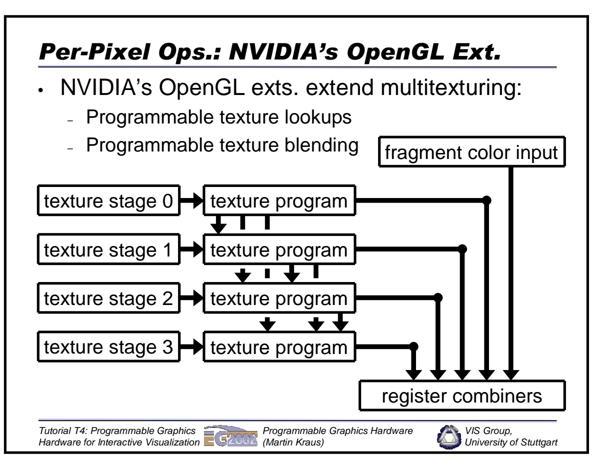
Per-Pixel Ops.: Multitexturing (OpenGL 1.2)

- Multitexturing is an optional extension of OpenGL 1.2 for texture blending.



Per-Pixel Ops.: NVIDIA's OpenGL Ext.

- NVIDIA's OpenGL exts. extend multitexturing:
  - Programmable texture lookups
  - Programmable texture blending



Per-Pixel Ops.: NVIDIA's OpenGL Ext.

- GL_NV_texture_shader (GeForce 3 and 4):
  - Each texture shader program has 2 results:
    - Shader stage result (input for subsequent stages)
    - Texture unit RGBA result (texture color, input for reg. comb.)
  - 21 different texture shader programs.
  - New texture formats: signed colors, texture offset groups, HILO (16 bit, 2 components).
- GL_NV_texture_shader2 (GeForce 3 and 4):
  - 2 new texture shader programs for 3D textures.
- GL_NV_texture_shader3 (GeForce 4):
  - 14 new texture shader programs.
  - new texture format: HILO8 (8 bit with 16 bit filtering).

Per-Pixel Ops.: NVIDIA's OpenGL Ext.

- Basic texture programs: (texture coords (s,t,r,q))
  - **GL_NONE**: RGBA result = (0,0,0,0)
  - **GL_TEXTURE_1D**: 1D texture access via (s/q)
  - **GL_TEXTURE_2D**: 2D texture access via (s/q, t/q)
  - **GL_TEXTURE_RECTANGLE_NV**: 2D rectangular texture access via (s/q, t/q)
  - **GL_CUBE_MAP_ARB**: Cube map texture access via (s,t,r)
  - **GL_PASS_THROUGH_NV**: RGBA result = (s,t,r,q) clamped to [0,1]
  - **GL_CULL_FRAGMENT_NV**: compares s, t, r, q to zero and discards fragment if any result corresponds to the specified cull mode (GL_LESS or GL_EQUAL).

Per-Pixel Ops.: NVIDIA's OpenGL Ext.

- Offset texture programs:
  - The result of a specified previous texture stage has to be a texture offset group (ds, dt) or (ds, dt, mag)
  - Vector (ds, dt) is transformed by specified 2x2 matrix.
  - **GL_OFFSET_TEXTURE_2D_NV**
    Adds transformed vector to texture coordinates (s, t).
  - **GL_OFFSET_TEXTURE_2D_SCALE_N**
    Additionally scales RGB result by mag component.
  - **GL_OFFSET_TEXTURE_RECTANGLE_NV, GL_OFFSET_TEXTURE_RECTANGLE_-SCALE_NV**

Per-Pixel Ops.: NVIDIA's OpenGL Ext.

- Example for **GL_OFFSET_TEXTURE_2D_NV**

fragment color input

texture stage 0: offset texture, (s_0, t_0, r_0, q_0)

GL_TEXTURE_2D lookup at $(s_0/q_0, t_0/q_0)$ — **$RGBA_0$=(0,0,0,0)**

(ds, dt)

texture stage 1: RGBA texture, (s_1, t_1, r_1, q_1), (a1, a2, a3, a4)

GL_OFFSET_-TEXTURE_2D_NV lookup at (s_1+a1***ds**+a2***dt**, t_1+a3***ds**+a4***dt**) — **$RGBA_1$**

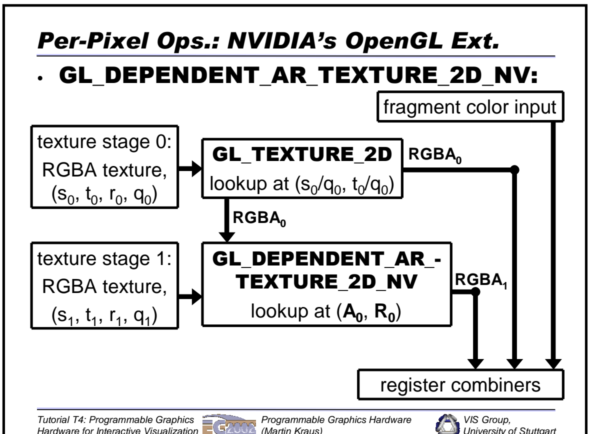
register combiners

Per-Pixel Ops.: NVIDIA's OpenGL Ext.

- Dependent texture programs:
  - **GL_DEPENDENT_AR_TEXTURE_2D_NV**
    Accesses a 2D texture via the (A, R) components of the RGBA result of a specified previous texture stage.
  - **GL_DEPENDENT_GB_TEXTURE_2D_NV**
    Accesses a 2D texture via the (G, B) components of the RGBA result of a specified previous texture stage.

Per-Pixel Ops.: NVIDIA's OpenGL Ext.

- **GL_DEPENDENT_AR_TEXTURE_2D_NV:**



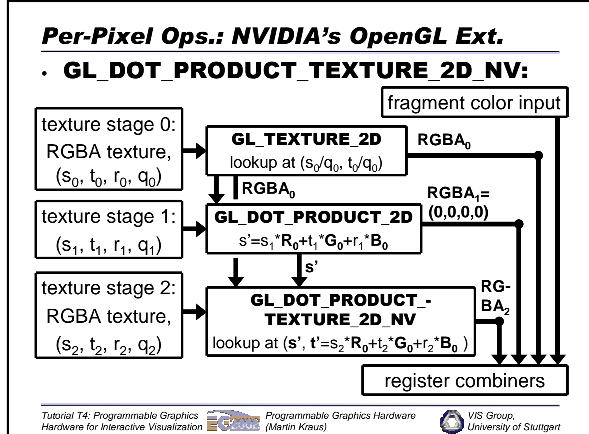
Per-Pixel Ops.: NVIDIA's OpenGL Ext.

- Basic dot product texture programs:
  - **GL_DOT_PRODUCT_NV**
    Does not access a texture map!
    Computes the dot product of (s, t, r) and the (R, G, B) result of a specified previous texture stage.
  - **GL_DOT_PRODUCT_TEXTURE_2D_NV**
    When preceded by GL_DOT_PRODUCT_NV, computes a 2nd dot product (s, t, r) with (R, G, B) result of a specified previous texture stage and accesses a 2D texture map via the 2 dot products.

  Corresponds to a 2x3 matrix multiplication if both programs use the same RGBA result.

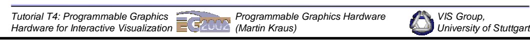
Per-Pixel Ops.: NVIDIA's OpenGL Ext.

- **GL_DOT_PRODUCT_TEXTURE_2D_NV:**



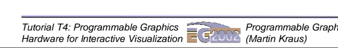
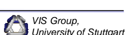
Per-Pixel Ops.: NVIDIA's OpenGL Ext.

- More dot product texture programs:
  - **GL_DOT_PRODUCT_TEXTURE_-RECTANGLE_NV**
  - **GL_DOT_PRODUCT_TEXTURE_-CUBE_MAP_NV**
  - **GL_DOT_PRODUCT_REFLECT_-CUBE_MAP_NV**
  - **GL_DOT_PRODUCT_-CONST_EYE_REFLECT_CUBE_MAP_NV**
  - **GL_DOT_PRODUCT_DIFFUSE_-CUBE_MAP_NV**
  - **GL_DOT_PRODUCT_DEPTH_REPLACE_NV**



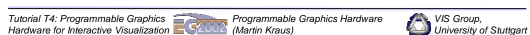
Per-Pixel Ops.: NVIDIA's OpenGL Ext.

- GL_NV_texture_shader2 programs:
  - **GL_TEXTURE_3D**
    Accesses a 3D texture map via (s / q, t / q, r / q).
  - **GL_DOT_PRODUCT_TEXTURE_3D_NV**
    Similar to GL_DOT_PRODUCT_TEXTURE_2D_NV for 3D texture maps.
    (Requires two preceding GL_DOT_PRODUCTS_NV.)

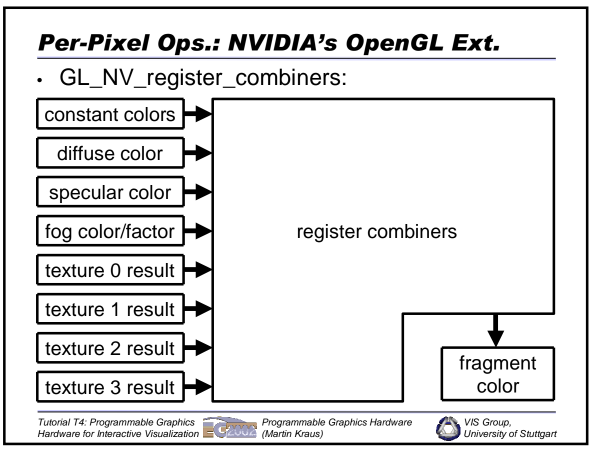
Per-Pixel Ops.: NVIDIA's OpenGL Ext.

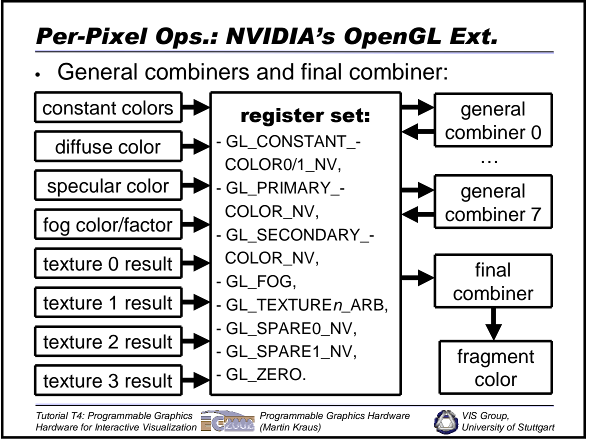
- Offset GL_NV_texture_shader3 programs:
  - **GL_OFFSET_PROJECTIVE_TEXTURE_2D_NV**
  - **GL_OFFSET_PROJECTIVE_TEXTURE_2D-_SCALE_NV**
  - **GL_OFFSET_PROJECTIVE_TEXTURE-_RECTANGLE_NV**
  - **GL_OFFSET_PROJECTIVE_TEXTURE-_RECTANGLE_SCALE_NV**
  - **GL_OFFSET_HILO_TEXTURE_2D_NV**
  - **GL_OFFSET_HILO_TEXTURE_RECTANGLE_NV**
  - **GL_OFFSET_HILO_PROJECTIVE-_TEXTURE_2D_NV**
  - **GL_OFFSET_HILO_PROJECTIVE_TEXTURE-_RECTANGLE_NV**

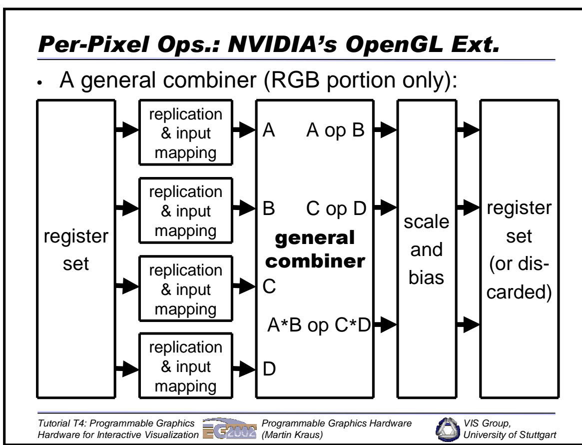


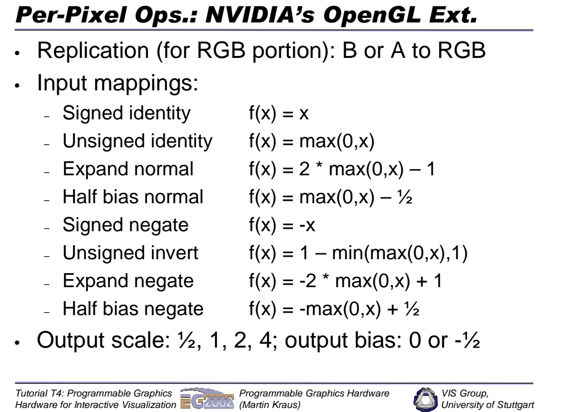
Per-Pixel Ops.: NVIDIA's OpenGL Ext.

- More GL_NV_texture_shader3 programs:
  - **GL_DEPENDENT_HILO_TEXTURE_2D_NV**
  - **GL_DEPENDENT_RGB_TEXTURE_3D_NV**
  - **GL_DEPENDENT_RGB_TEXTURE-_CUBE_MAP_NV**
  - **GL_DOT_PRODUCT_TEXTURE_1D_NV**
  - **GL_DOT_PRODUCT_PASS_THROUGH_NV**
  - **GL_DOT_PRODUCT_AFFINE_DEPTH-_REPLACE_NV**



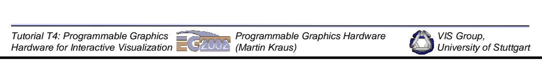






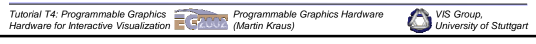
Per-Pixel Ops.: NVIDIA's OpenGL Ext.

- Operations (A op B, C op D, A*B op C*D):
  - A • B, C • D:
    Dot product replicated to all components.
  - A * B, C * D:
    Multiplication of corresponding components.
  - A * B + C * D:
    Sums of products of corresponding components.
  - A * B mux C * D:
    If alpha of register spare0 < ½ : A * B, else: C * D.



Per-Pixel Ops.: NVIDIA's OpenGL Ext.

- Alpha portions of general combiners work similarly but independent.
- Final combiner:
  - Has 7 inputs and computes the fragment color,
  - RGB portion is a little more complicated and specialized for linear interpolation of two colors,
  - A portion is simpler.
- GL_NV_register_combiners2:
  - Introduces 2 constant colors per combiner instead of 2 constant colors in total.



Per-Pixel Ops.: NVIDIA's OpenGL Ext.

Summary for GL_NV_texture_shader(2,3) and GL_NV_register_combiners(2):

- Low-level API for per-pixel shading on GeForce3
- Not discussed here: OpenGL calls for setup.
- Many calls necessary (but "programs" are small).
- Alternative API on same level: nvparse.
- Texture shader: Many powerful texture shader programs (but only 4 texture stages).
- Register combiners: Only simple arithmetics (and only 8 general and 1 final combiner).

Per-Pixel Operations: Pixel Shader 1.3

- DirectX 8.0 introduced
  - Pixel Shader 1.0, 1.1, 1.2 (NVIDIA GeForce 3)
- DirectX 8.1 introduced
  - Pixel Shader 1.3 (NVIDIA GeForce 4)
  - Pixel Shader 1.4 (ATI Radeon 8500)
- Pixel shader 1.3 programs specify the configuration of texture shaders *and* register combiners. Example:

```
ps.1.3        // version instruction
tex t0        // standard texture 0 lookup
mov r0, t0    // copy result to output color
```


Per-Pixel Operations: Pixel Shader 1.3

- Nomenclature:

NVIDIA OpenGL ext.	DirectX 8 Pixel Shader
texture shader programs	texture addressing instructions
register combiners	arithmetic instructions
B or A replication	`.b`/`.a` source register selector
input mapping	source register modifiers
output scale and bias	instruction modifiers
RGB/A portion of general or final combiner	`.rgb`/`.a` destination register write mask (both: `.rgba`)
parallel combiner portions	instruction pairing

Per-Pixel Operations: Pixel Shader 1.3

- Pixel Shader 1.3 instruction sequence:
  - Version instruction:

```
ps.1.3
```

  - Definition of constants (up to 8):

```
def cn, f0, f1, f2, f3
```

  - Up to 4 texture addressing instructions. (Corresponding to 4 texture units.)
  - Up to 8 arithmetic instructions (Corresponding to 8 general combiners.)

Per-Pixel Operations: Pixel Shader 1.3

- Texture addressing instructions:
  `t`*n* specifies the current and `t`*m* a previous tex. stage.

NVIDIA's texture shader programs	Pixel Shader's texture addressing instruction
GL_TEXTURE_1/2/3D	`tex t`*n*
GL_PASS_THROUGH_NV	`texcoord t`*n*
GL_CULL_FRAGMENT_NV	`texkill t`*n*
GL_TEXTURE_OFFSET-_TEXTURE_2D_NV	`texbem t`*n*`, t`*m*
GL_TEXTURE_OFFSET_-TEXTURE_2D_SCALE_NV	`texbeml t`*n*`, t`*m*

Per-Pixel Operations: Pixel Shader 1.3

- Texture addressing instructions:

NVIDIA's texture shader program	Pixel Shader's texture addressing instruction
GL_DEPENDENT_AR-_TEXTURE_2D_NV	`texreg2ar t`*n*`, t`*m*
GL_DEPENDENT_GB-_TEXTURE_2D_NV	`texreg2gb t`*n*`, t`*m*
GL_DEPENDENT_RGB-_TEXTURE_3D_NV	`texreg2rgb t`*n*`, t`*m*

Per-Pixel Operations: Pixel Shader 1.3

- Texture addressing instructions:

NVIDIA's texture shader program	Pixel Shader's texture addressing instruction
GL_DOT_PRODUCT_NV	`texm3x2pad tn, tm` `texm3x3pad tn, tm`
GL_DOT_PRODUCT_-TEXTURE_2D_NV	`texm3x2tex tn, tm`
GL_DOT_PRODUCT_-TEXTURE_3D_NV	`texm3x3tex tn, tm`
GL_DOT_PRODUCT_-DEPTH_REPLACE_NV	`texm3x2depth tn, tm`

Per-Pixel Operations: Pixel Shader 1.3

- Texture addressing instructions:

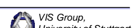
NVIDIA's texture shader program	Pixel Shader's texture addressing instruction
GL_DOT_PRODUCT_-REFLECT_CUBE_MAP_NV	`texm3x3vspec tn, tm`
GL_DOT_PRODUCT_-CONST_EYE_-REFLECT_CUBE_MAP_NV	`texm3x3spec tn, tm`
GL_DOT_PRODUCT-_TEXTURE_1D_NV	`texdp3tex tn, tm`
GL_DOT_PRODUCT_-PASS_THROUGH_NV	`texdp3 tn, tm`

Per-Pixel Operations: Pixel Shader 1.3

- Register nomenclature:

NVIDIA's register combiners	DirectX 8.1 Pixel Shader 1.3
GL_CONSTANT_COLOR0/1_NV (one pair for each combiner)	constant registers `c0`, …, `c7`
GL_PRIMARY_COLOR_NV	color register `v0`
GL_SECONDARY_COLOR_NV	color register `v1`
GL_TEXTURE*n*_ARB	texture reg. `t0`, …, `t3`
GL_SPARE0_NV	temporary register `r0`
GL_SPARE1_NV	temporary register `r1`

Note: The output color is returned in `r0`.



Per-Pixel Operations: Pixel Shader 1.3

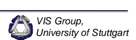
- Source register modifiers (only arithmetic instr.):

NVIDIA's input mappings	source register modifiers	syntax
signed identity	default	*src*
expand normal	signed scaling	*src*`_bx2`
half bias normal	bias	*src*`_bias`
signed negate	negate	`-`*src*
unsigned invert	invert	`1-`*src*
expand negate	signed scaling & negate	`-`*src*`_bx2`
half bias negate	bias and negate	`-`*src*`_bias`
unsigned identity	*indirectly supported*	

Per-Pixel Operations: Pixel Shader 1.3

- Instruction modifiers (only arithmetic instr.):

NVIDIA's scale & bias	instruction modifiers	syntax
no scale, no bias	default	*ins*
scale by ½, no bias	dvide by 2	*ins*`_d2`
scale by 2, no bias	multiply by 2	*ins*`_x2`
scale by 4, no bias	multiply by 4	*ins*`_x4`
any scale with bias	*not supported*	
indirectly supported	saturate (clamp to [0,1])	*ins*`_sat`



Per-Pixel Operations: Pixel Shader 1.3

- Pixel Shader 1.3 arithmetic instructions:

Each instruction corresponds to one general combiner.

```
nop                            // no operation
mov dest, src0                 // dest = src0
add dest, src0, src1           // dest = src0 + src1
sub dest, src0, src1           // dest = src0 - src1
mul dest, src0, src1           // dest = src0 * src1
mad dest, src0, src1, src2     // dest = src0*src1+src2
dp3 dest, src0, src1           // dest = src0 • src1
dp4 dest, src0, src1           // dest = src0 • src1
```

Per-Pixel Operations: Pixel Shader 1.3

- More Pixel Shader 1.3 arithmetic instructions:

```
cmp dest,src0,src1,src2 // dest = src1 if src0>0
                        //        otherwise src2
cnd dest,src0,src1,src2 // dest = src1 if src0>0.5
                        //        otherwise src2
lrp dest,src0,src1,src2 // dest = src0 * src1 +
                        //        (1 - src0) * src2
```

Note:

- `cmp` counts as two instructions.
- *src0* of `cnd` has to be `r0.a` .

Per-Pixel Operations: Pixel Shader 1.3

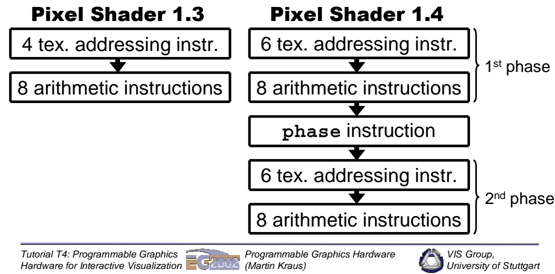
Summary for Pixel Shader 1.3:

- DirectX 8.1 assembler language for per-pixel operations on NVIDIA's GeForce 4.
- Pixel Shader 1.2 for NVIDIA's GeForce 3.
- Functionality very close to NVIDIA's texture shader plus register combiners:
  - 1 texture stage ↔ 1 texture addressing instruction
  - 1 general combiner ↔ 1 arithmetic instruction
- Not discussed here:
  - Many restrictions on particular instructions.
  - Compiling of programs, etc.

Per-Pixel Operations: Pixel Shader 1.4

Pixel Shader 1.4:

- Introduced in DirectX 8.1 for ATI's Radeon 8500.
- Different hardware, thus many changes to 1.3:



Per-Pixel Operations: Pixel Shader 1.4

- 6 temporary registers `r0`, …, `r5` (instead of 2)
  But texture registers `t0`, …, `t5` are no longer writable.
- Less texture addressing instructions:

```
texld rn,tm      // instead of tex
texld rn,rm      // dep. tex. read, only 2nd phase
texcrd rn,tm     // instead of texcoord
texkill rm/tm    // similar to texkill in 1.3
texdepth r5      // instead of texm3x2depth
```

Note:
rn specifies the destination register and texture unit.
rm / *tm* specifies the texture coordinate set.

Per-Pixel Operations: Pixel Shader 1.4

- New source register selectors:
  `src.r, src.g` — replicate r/g
- New source register modifier:
  `src_x2` — multiply by 2
- New source register modifier for `texld`, `texcrd`
  `src_dw, src_dz` — divide by w/z coord.
- New instruction modifiers:
  `ins_x8, ins_d4, ins_d8`
- Arbitrary destination write masks:
  `src.[r][g][b][a]`

Per-Pixel Operations: Pixel Shader 1.4

- New arithmetic instruction:

```
bem rn.rg, src0, src1      // only 1st phase
```

  - Computes
    r*n*.g = *src0*.`r` + M_{00} * *src1*.`r` + M_{01} * *src1*.`g`
    r*n*.b = *src0*.`g` + M_{10} * *src1*.`r` + M_{11} * *src1*.`g`
  - Consumes two instructions.

Per-Pixel Operations: Pixel Shader 1.4

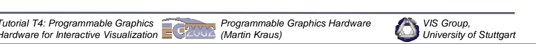
Summary for Pixel Shader 1.4:

- DirectX 8.1 assembler language for per-pixel operations on ATI's Radeon 8500.
- Not discussed here:
  - Many restrictions on particular instructions.
  - Compiling of programs, etc.
- Compared to Pixel Shader 1.3:
  - More texture lookups (12 instead of 4).
  - More arithmetic instructions (16 instead of 8).
  - Arithmetic instructions before texture lookups.
  - Just as ATI Radeon 8500 vs. GeForce 3 and 4.

Per-Pixel Operations: ATI's OpenGL Ext.

- GL_ATI_fragment_shader (Radeon 8500):
  - ATI's OpenGL extension for per-pixel operations.
  - Functionality very similar to Pixel Shader 1.4.
  - Nomenclature:

Pixel Shader 1.4	GL_ATI_fragment_shader
texture addressing instr.	routing instructions
arithmetic instructions	color/alpha fragment instr.
texture registers	interpolators
constant registers	constants
temporary registers	registers
phase	pass



Per-Pixel Operations: ATI's OpenGL Ext.

- Operations specified with OpenGL calls, e.g.

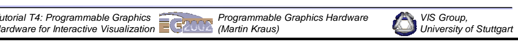
```
tex r1, t2.rgb     // Pixel Shader 1.4
mov r0.rgba, r1.rgba
```

```
// corresponding OpenGL code:
SampleMapATI(GL_REG_1_ATI,
GL_TEXTURE2_ARB, GL_SWIZZLE_SRT_ATI);
ColorFragmentOp1ATI(GL_MOV_ATI,
GL_REG_0_ATI, GL_NONE, GL_NONE,
GL_REG_1_ATI, GL_NONE, GL_NONE);
AlphaFragmentOp1ATI(GL_MOV_ATI,
GL_REG_0_ATI, GL_NONE, GL_NONE,
GL_REG_1_ATI, GL_NONE, GL_NONE);
```

Per-Pixel Operations: ATI's OpenGL Ext.

Summary for GL_ATI_fragment_shader:

- ATI's OpenGL extension for per-pixel operations on Radeon 8500.
- Functionality very similar to Pixel Shader 1.4.
- Programming requires many OpenGL calls.
- Not discussed here: Any details.



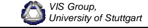
Outlook

Contents:

- Programmability of graphics hardware
- Per-vertex operations
- Per-pixel operations
- **Outlook**
  - **Next generation graphics hardware**
  - **Long-term future**
  - **Consequences for visualization algorithms**

Outlook: Next Generation Graphics HW

- Announced graphics chips:
  NVIDIA: nv30, ATI: r300, 3Dlabs: P10, Matrox: Parhelia
- Likely to support at least a subset of DirectX 9's Vertex Shader 2.0 and PixelShader 2.0.
- Vertex Shader 2.0 (announced features):
  - Displacement maps,
  - Jumps and loops,
  - Subroutines,…
- Pixel Shader 2.0 (announced features):
  - Floating-point precision,
  - Much more instructions, …



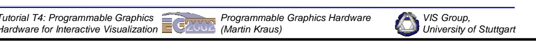
Outlook: Long-Term Future

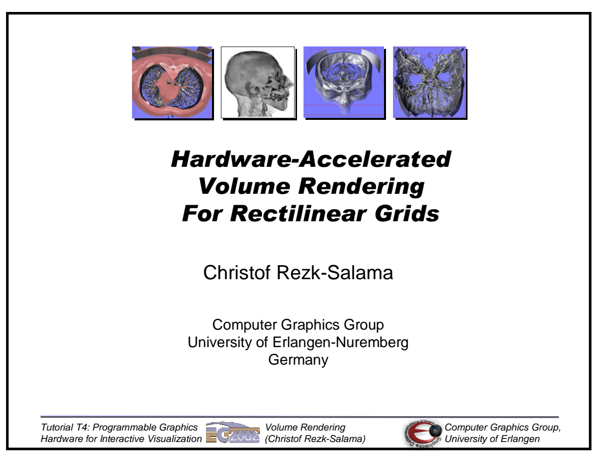
- High-level shading languages will become more important:
  - More comfortable programming.
  - Less than optimal code (acceptable trade-off).
  - NVIDIA's Cg or DirectX 9's shading language?
  - Independent of operating system and graphics HW?
- Unified per-pixel and per-vertex operations.
- Programmable blending with destination pixel.
- Programmable per-primitive operations.

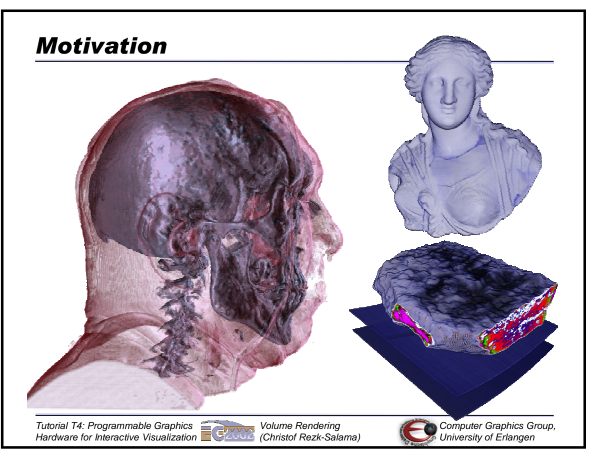
Outlook: Consequences for Visualization

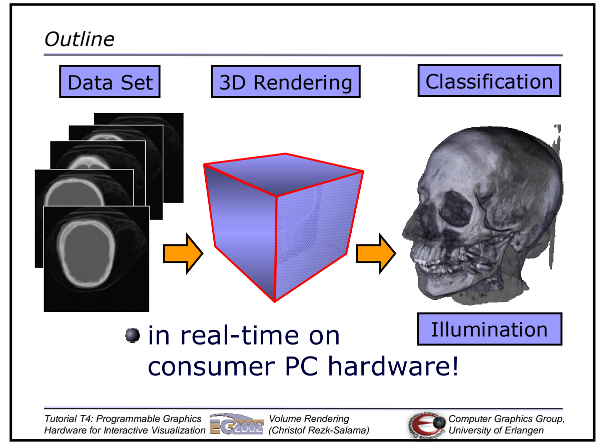
- Many visualization algorithms can benefit from programmable graphics hardware.
- In some cases, programmable graphics hardware allows completely new approaches.
- Several examples will be presented in this tutorial.

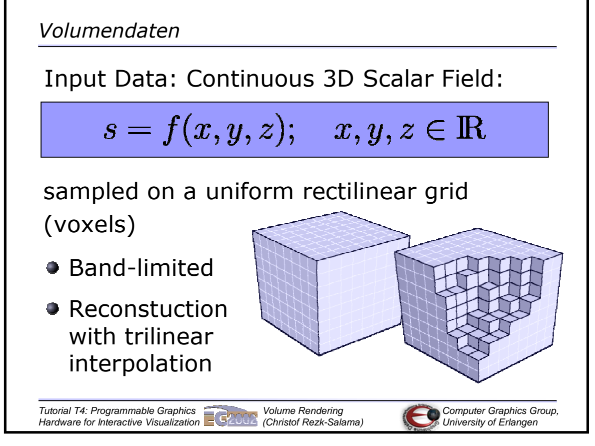
- **There is a lot of potential for visualization research.**

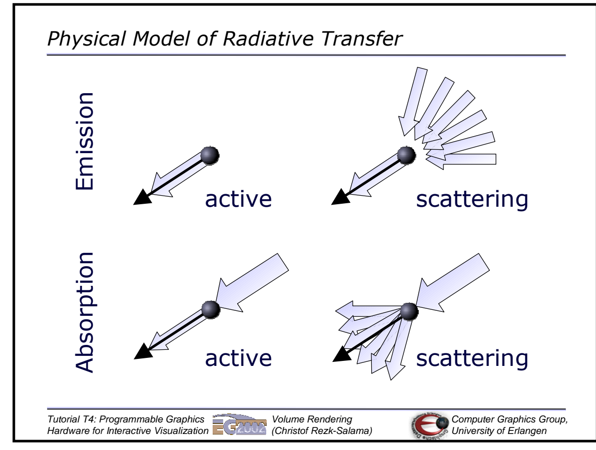


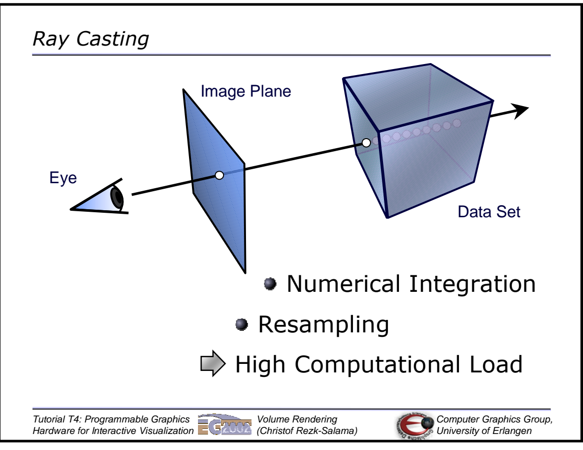

Hardware-Accelerated
Volume Rendering
For Rectilinear Grids
Christof Rezk-Salama
Computer Graphics Group
University of Erlangen-Nuremberg
Germany
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

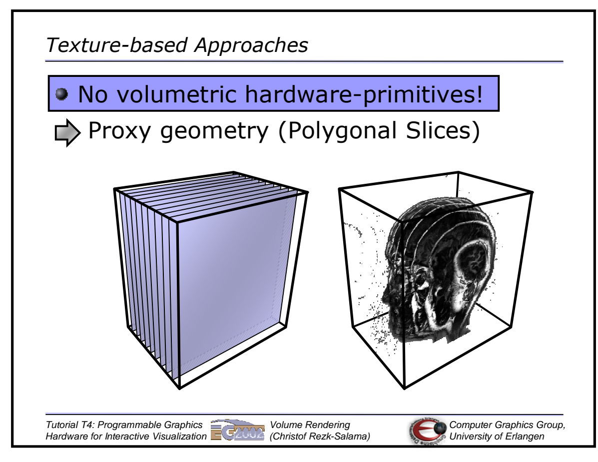

Motivation
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

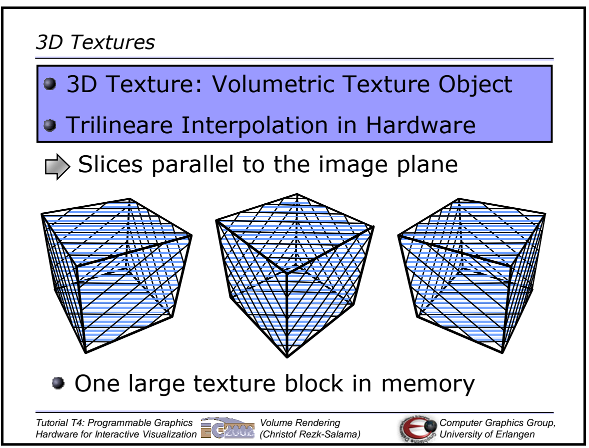

Outline
Data Set
3D Rendering
Classification
Illumination
in real-time on
consumer PC hardware!
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

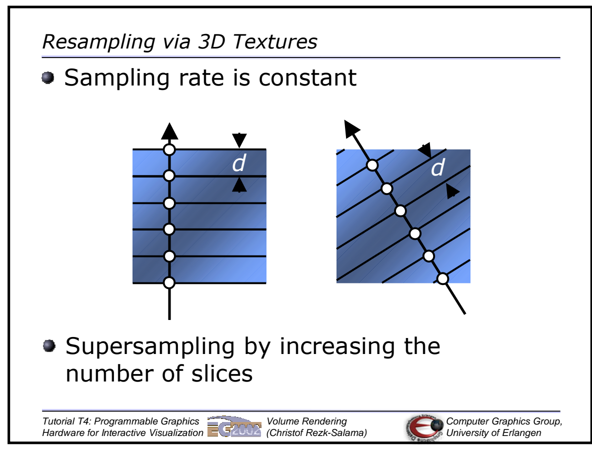

Volumendaten
Input Data: Continuous 3D Scalar Field:
$s = f(x,y,z); \quad x,y,z \in \mathbb{R}$
sampled on a uniform rectilinear grid
(voxels)
Band-limited
Reconstruction
with trilinear
interpolation
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

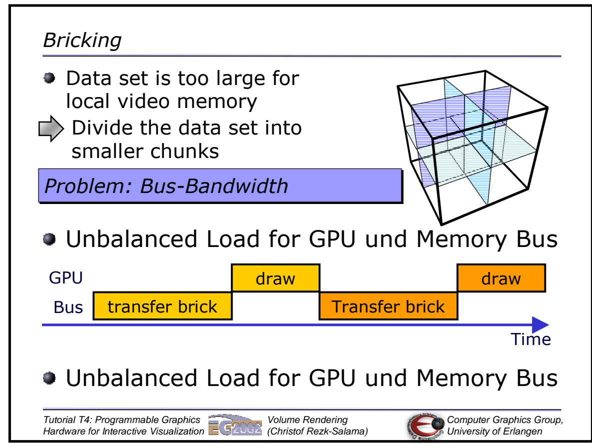

Physical Model of Radiative Transfer
Emission
Absorption
active
scattering
active
scattering
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

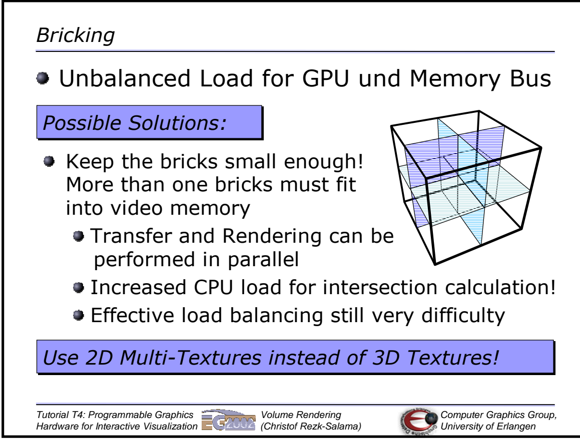

Ray Casting
Image Plane
Eye
Data Set
Numerical Integration
Resampling
High Computational Load
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

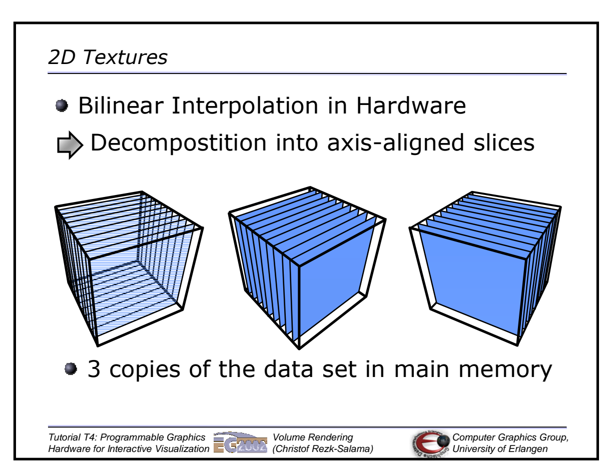

Texture-based Approaches
No volumetric hardware-primitives!
Proxy geometry (Polygonal Slices)
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

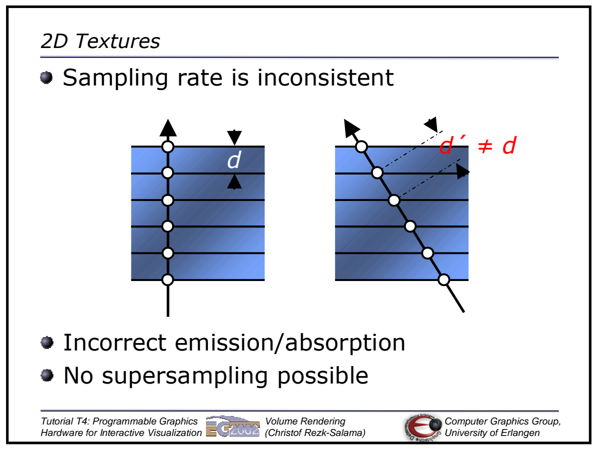

3D Textures
3D Texture: Volumetric Texture Object
Trilineare Interpolation in Hardware
Slices parallel to the image plane
One large texture block in memory
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

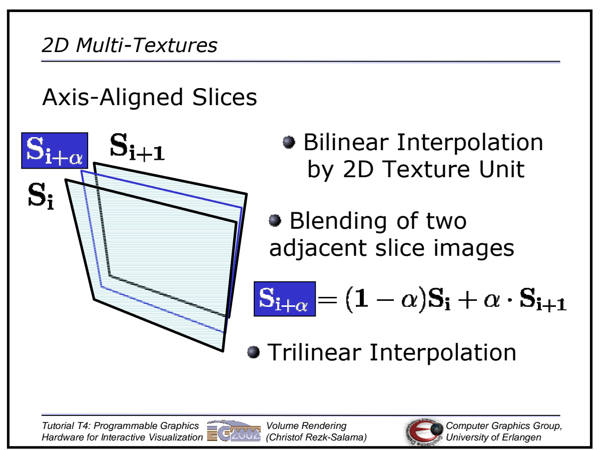

Resampling via 3D Textures
Sampling rate is constant
d
d
Supersampling by increasing the number of slices
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

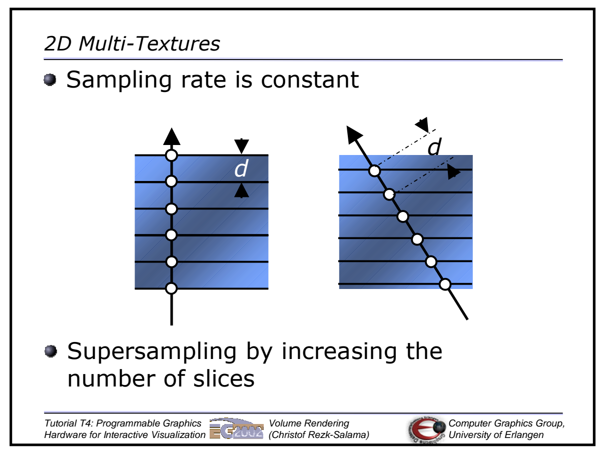

Bricking
Data set is too large for local video memory
Divide the data set into smaller chunks
Problem: Bus-Bandwidth
Unbalanced Load for GPU und Memory Bus
GPU
Bus
draw
draw
transfer brick
Transfer brick
Time
Unbalanced Load for GPU und Memory Bus
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

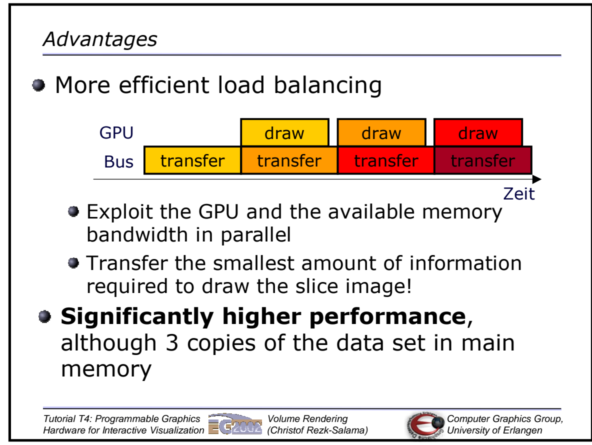

Bricking
Unbalanced Load for GPU und Memory Bus
Possible Solutions:
Keep the bricks small enough! More than one bricks must fit into video memory
Transfer and Rendering can be performed in parallel
Increased CPU load for intersection calculation!
Effective load balancing still very difficulty
Use 2D Multi-Textures instead of 3D Textures!
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

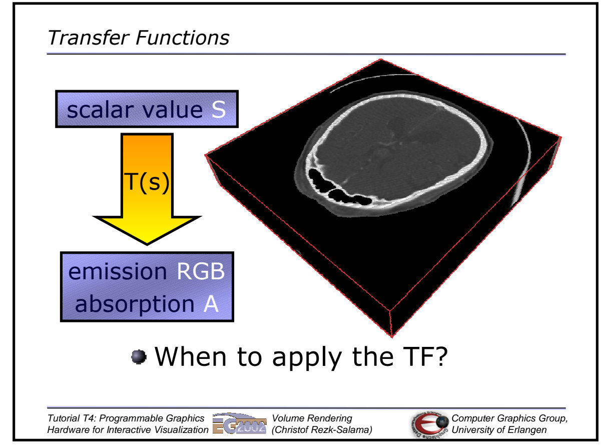

2D Textures
Bilinear Interpolation in Hardware
Decompostition into axis-aligned slices
3 copies of the data set in main memory
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

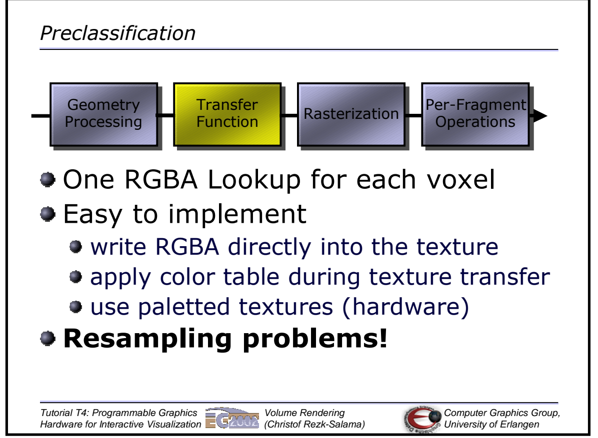

2D Textures
Sampling rate is inconsistent
d
d´ ≠ d
Incorrect emission/absorption
No supersampling possible
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

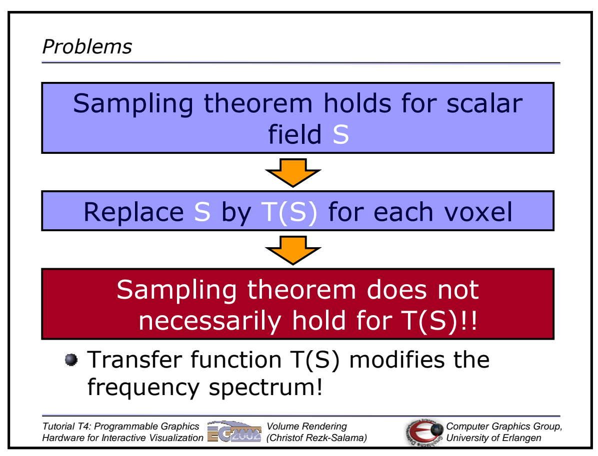

2D Multi-Textures
Axis-Aligned Slices
$S_{i+\alpha}$
S_{i+1}
S_i
Bilinear Interpolation by 2D Texture Unit
Blending of two adjacent slice images
$S_{i+\alpha} = (1-\alpha)S_i + \alpha \cdot S_{i+1}$
Trilinear Interpolation
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

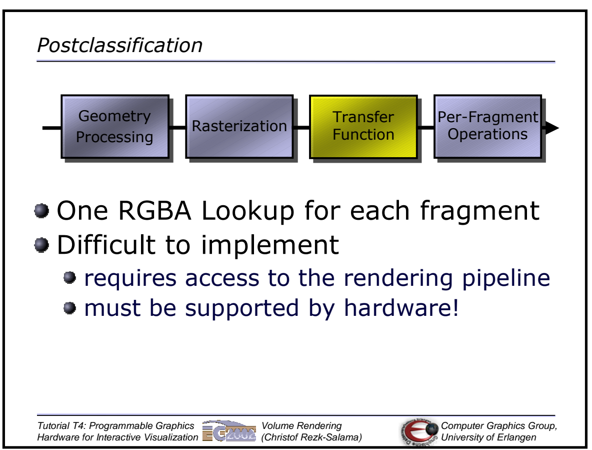

2D Multi-Textures
Sampling rate is constant
d
d
Supersampling by increasing the number of slices
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

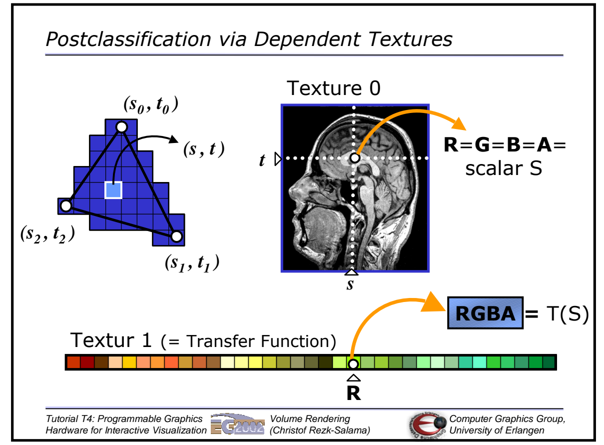

Advantages
More efficient load balancing
GPU
Bus
draw
draw
draw
transfer
transfer
transfer
transfer
Zeit
Exploit the GPU and the available memory bandwidth in parallel
Transfer the smallest amount of information required to draw the slice image!
Significantly higher performance, although 3 copies of the data set in main memory
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

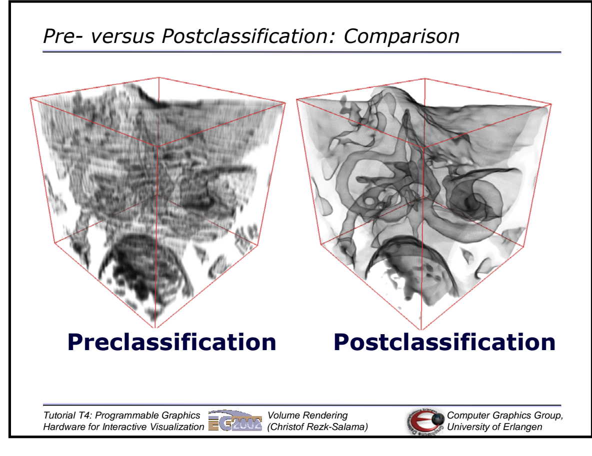

Transfer Functions
scalar value S
T(s)
emission RGB
absorption A
When to apply the TF?
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

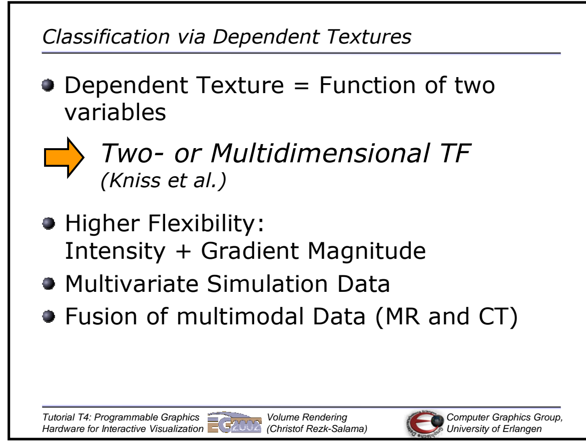

Preclassification
Geometry Processing
Transfer Function
Rasterization
Per-Fragment Operations
One RGBA Lookup for each voxel
Easy to implement
write RGBA directly into the texture
apply color table during texture transfer
use paletted textures (hardware)
Resampling problems!
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

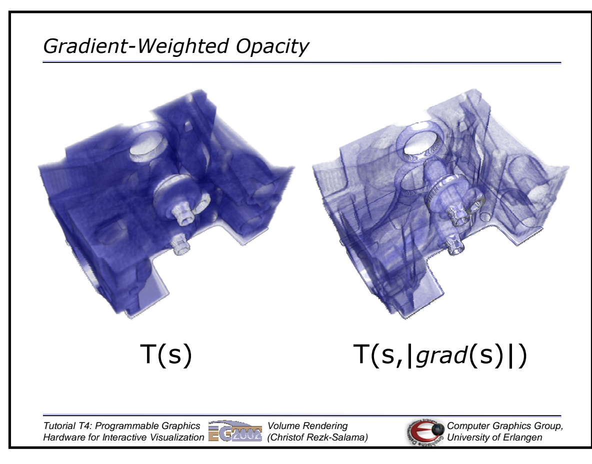

Problems
Sampling theorem holds for scalar field S
Replace S by T(S) for each voxel
Sampling theorem does not necessarily hold for T(S)!!
Transfer function T(S) modifies the frequency spectrum!
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

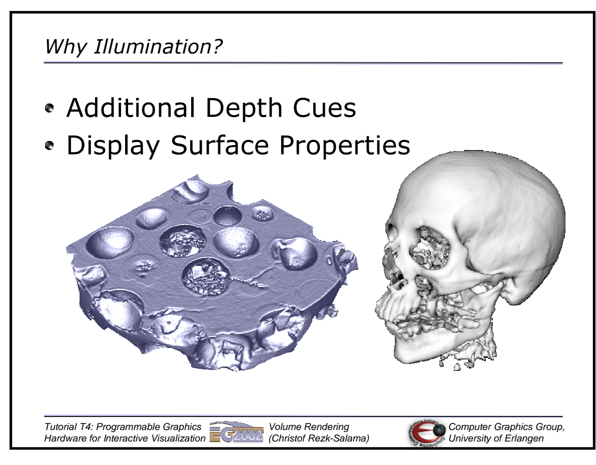

Postclassification
Geometry Processing
Rasterization
Transfer Function
Per-Fragment Operations
One RGBA Lookup for each fragment
Difficult to implement
requires access to the rendering pipeline
must be supported by hardware!
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

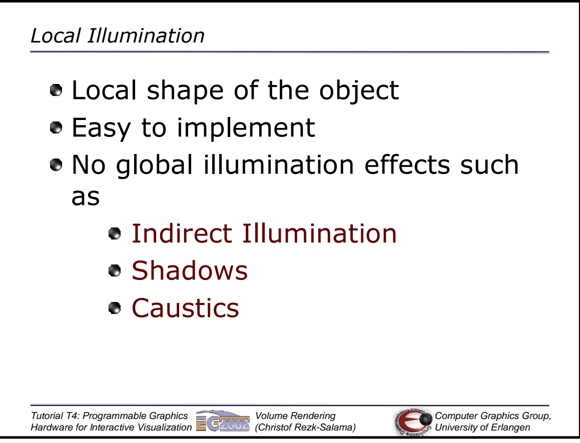

Postclassification via Dependent Textures
(s_0, t_0)
(s, t)
(s_2, t_2)
(s_1, t_1)
Texture 0
t
s
R=G=B=A= scalar S
RGBA = T(S)
Textur 1 (= Transfer Function)
R
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

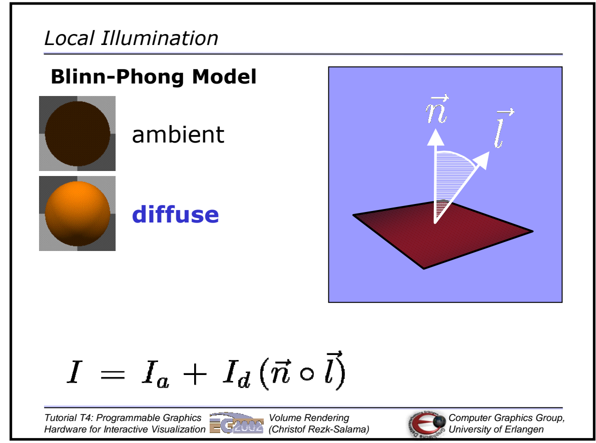

Pre- versus Postclassification: Comparison
Preclassification
Postclassification
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

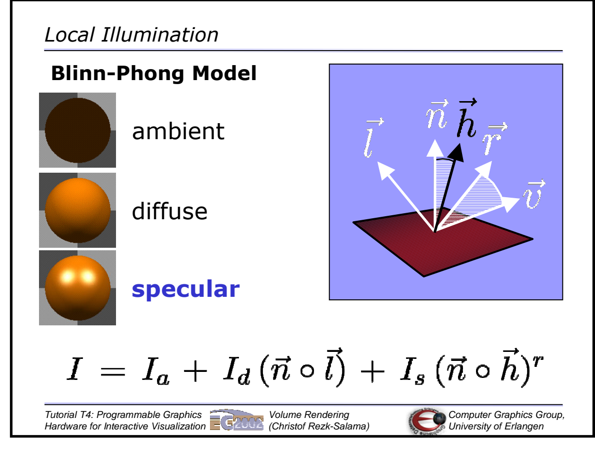

Classification via Dependent Textures
Dependent Texture = Function of two variables
Two- or Multidimensional TF
(Kniss et al.)
Higher Flexibility:
Intensity + Gradient Magnitude
Multivariate Simulation Data
Fusion of multimodal Data (MR and CT)
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

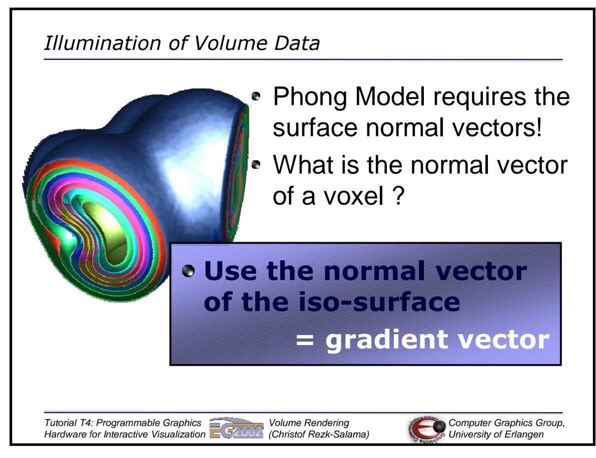

Gradient-Weighted Opacity
T(s)
T(s,|grad(s)|)
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

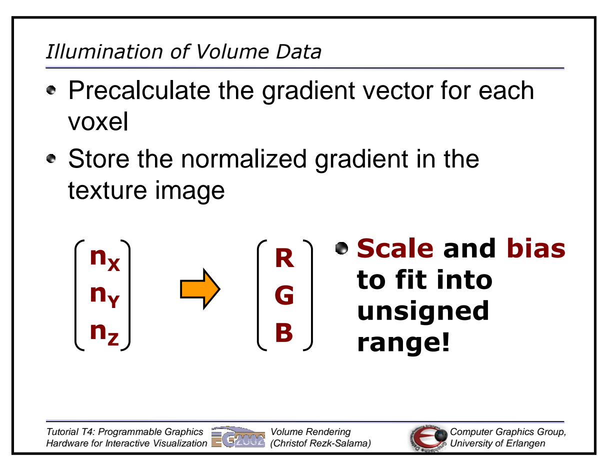

Why Illumination?
Additional Depth Cues
Display Surface Properties
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

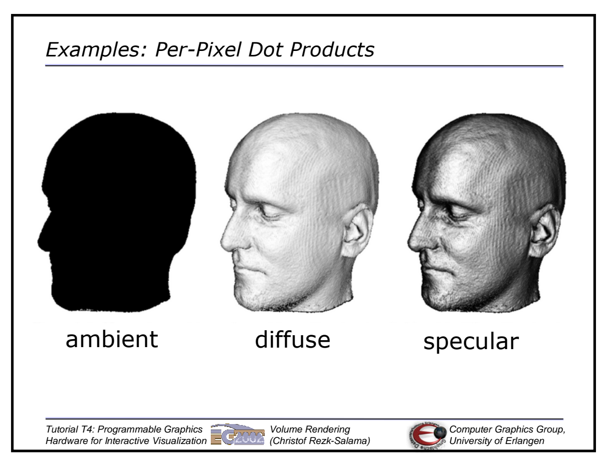

Local Illumination
Local shape of the object
Easy to implement
No global illumination effects such as
Indirect Illumination
Shadows
Caustics
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

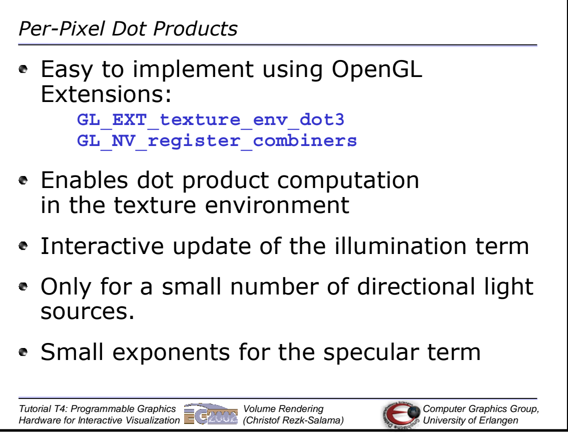

Local Illumination
Blinn-Phong Model
ambient
diffuse
$\vec{n}$
$\vec{l}$
$I = I_a + I_d\,(\vec{n} \circ \vec{l})$
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

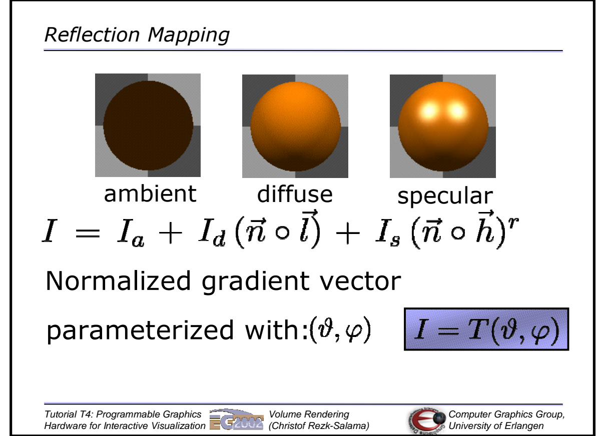

Local Illumination
Blinn-Phong Model
ambient
diffuse
specular
$\vec{l}$
$\vec{n}$
$\vec{h}$
$\vec{r}$
$\vec{v}$
$I = I_a + I_d\,(\vec{n} \circ \vec{l}) + I_s\,(\vec{n} \circ \vec{h})^r$
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

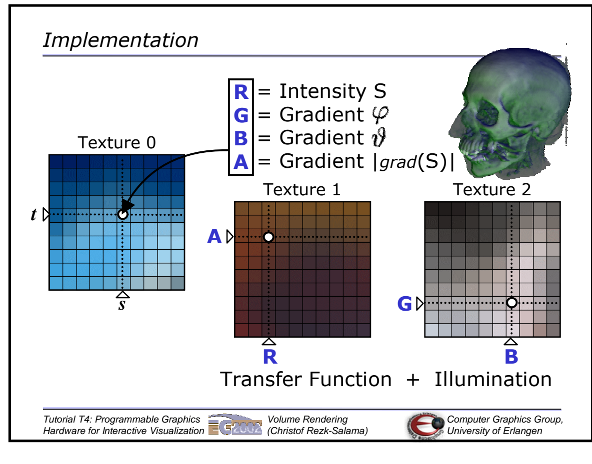

Illumination of Volume Data
Phong Model requires the surface normal vectors!
What is the normal vector of a voxel ?
Use the normal vector of the iso-surface
= gradient vector
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

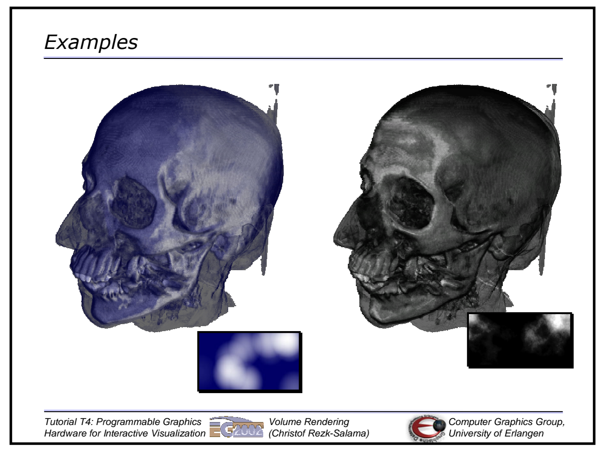

Illumination of Volume Data
Precalculate the gradient vector for each voxel
Store the normalized gradient in the texture image
n_X
n_Y
n_Z
R
G
B
Scale and bias to fit into unsigned range!
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

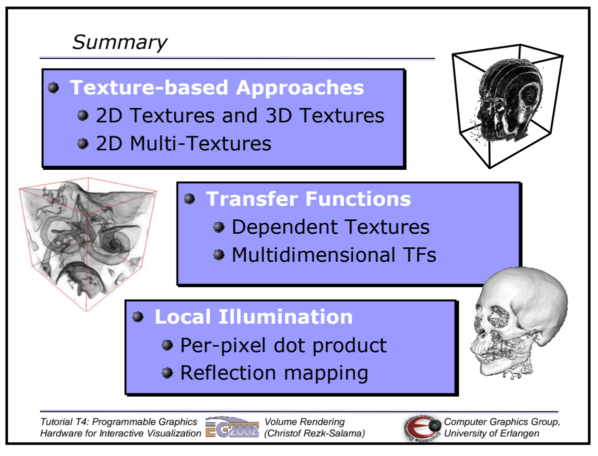

Examples: Per-Pixel Dot Products
ambient
diffuse
specular
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

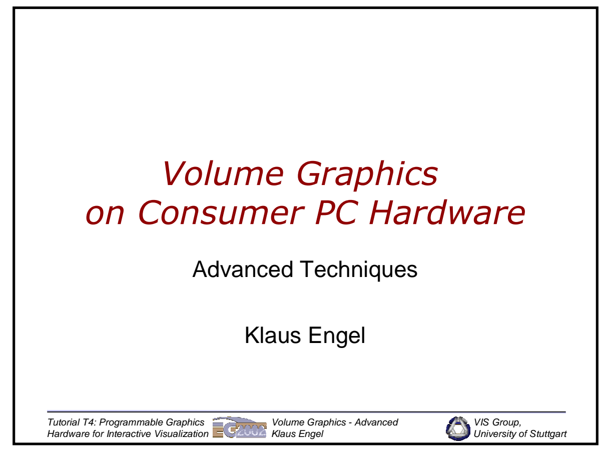

Per-Pixel Dot Products
Easy to implement using OpenGL Extensions:
GL_EXT_texture_env_dot3
GL_NV_register_combiners
Enables dot product computation in the texture environment
Interactive update of the illumination term
Only for a small number of directional light sources.
Small exponents for the specular term
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

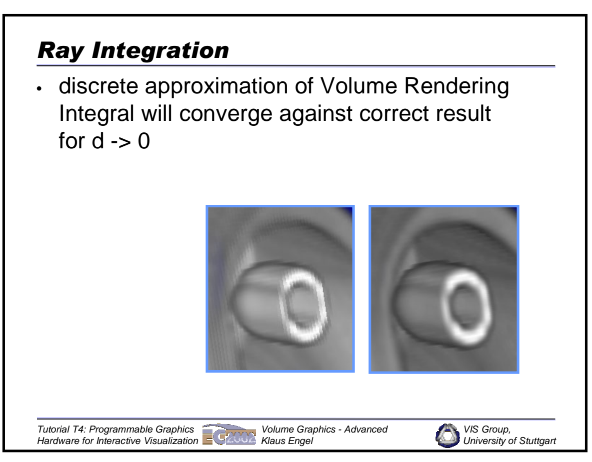

Reflection Mapping
ambient
diffuse
specular
$I = I_a + I_d\,(\vec{n} \circ \vec{l}) + I_s\,(\vec{n} \circ \vec{h})^r$
Normalized gradient vector
parameterized with: (ϑ, φ)
$I = T(\vartheta, \varphi)$
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

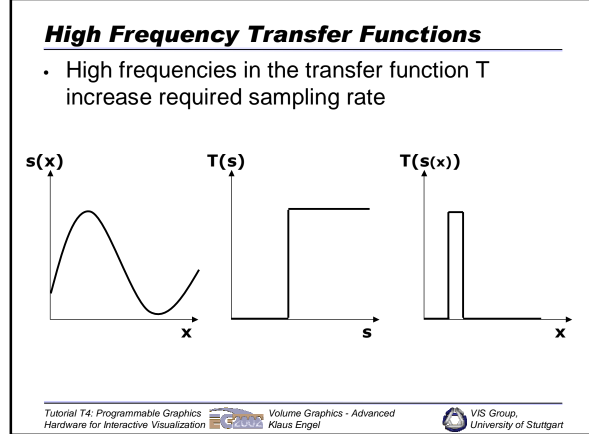

Implementation
R = Intensity S
G = Gradient φ
B = Gradient ϑ
A = Gradient |grad(S)|
Texture 0
t
s
Texture 1
A
R
Texture 2
G
B
Transfer Function + Illumination
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

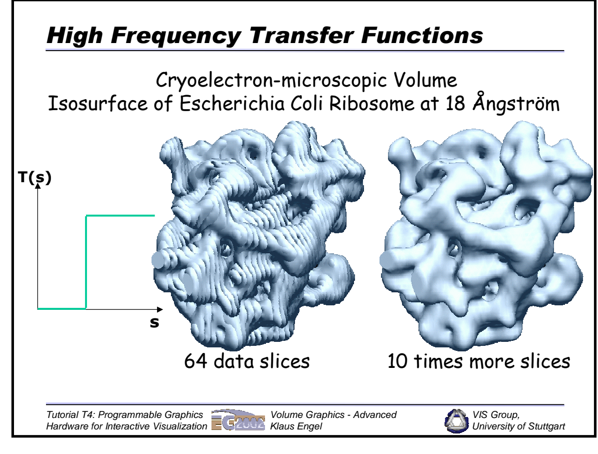

Examples
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

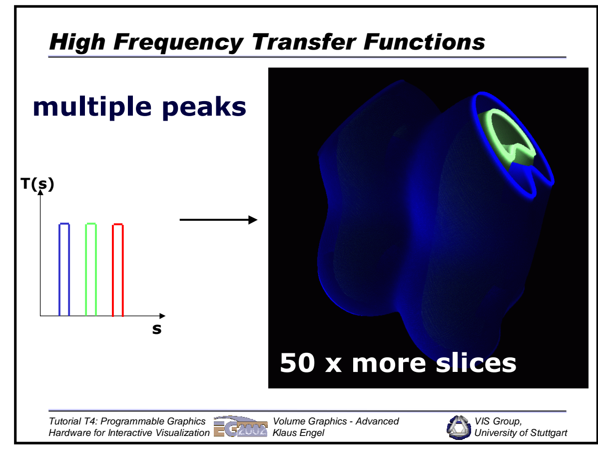

Summary
Texture-based Approaches
2D Textures and 3D Textures
2D Multi-Textures
Transfer Functions
Dependent Textures
Multidimensional TFs
Local Illumination
Per-pixel dot product
Reflection mapping
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Rendering (Christof Rezk-Salama)
Computer Graphics Group, University of Erlangen

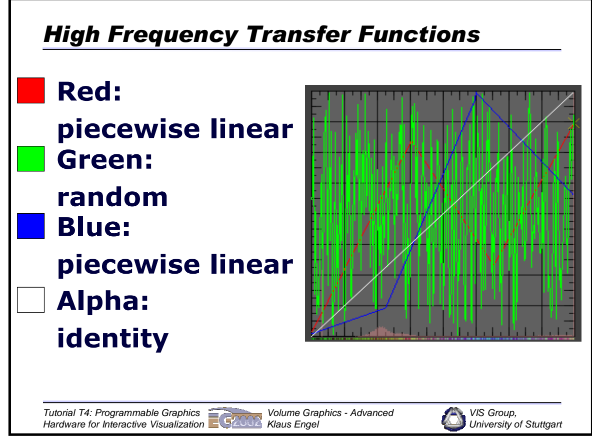

Volume Graphics
on Consumer PC Hardware
Advanced Techniques
Klaus Engel
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

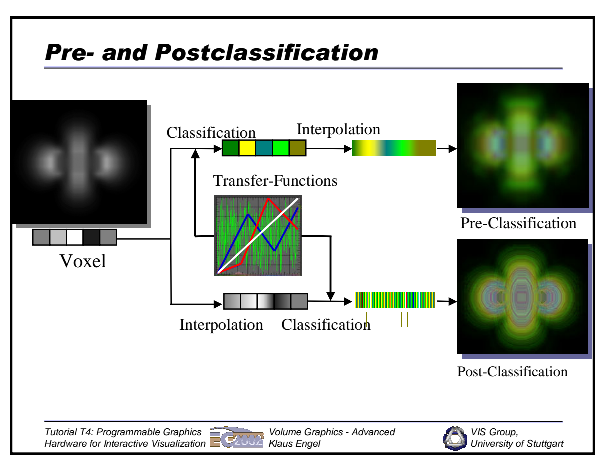

Ray Integration
• discrete approximation of Volume Rendering Integral will converge against correct result for d -> 0
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

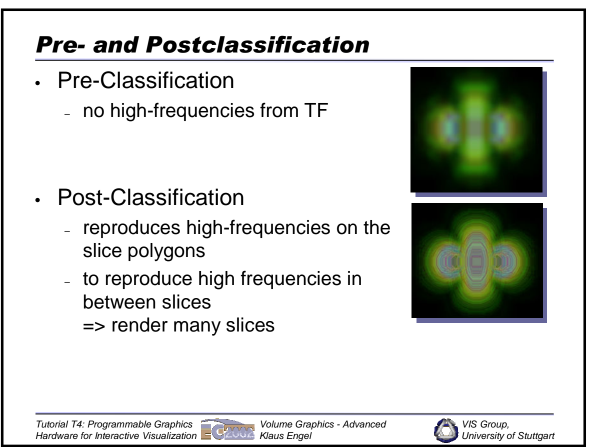

High Frequency Transfer Functions
• High frequencies in the transfer function T increase required sampling rate
s(x)
x
T(s)
s
T(s(x))
x
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

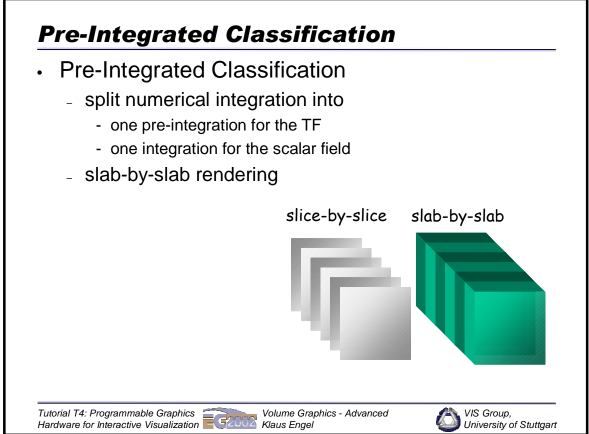

High Frequency Transfer Functions
Cryoelectron-microscopic Volume
Isosurface of Escherichia Coli Ribosome at 18 Ångström
T(s)
s
64 data slices
10 times more slices
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

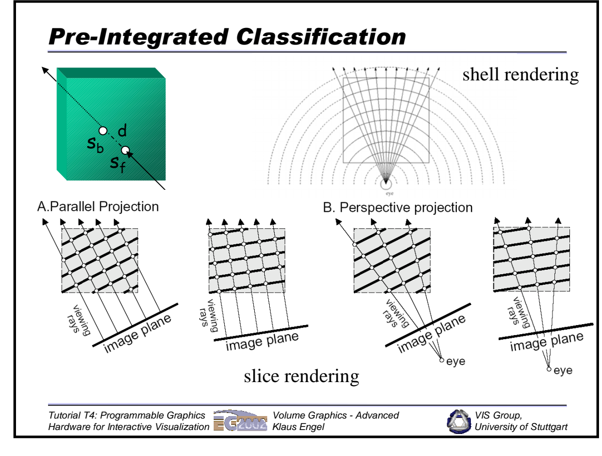

High Frequency Transfer Functions
multiple peaks
T(s)
s
50 x more slices
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

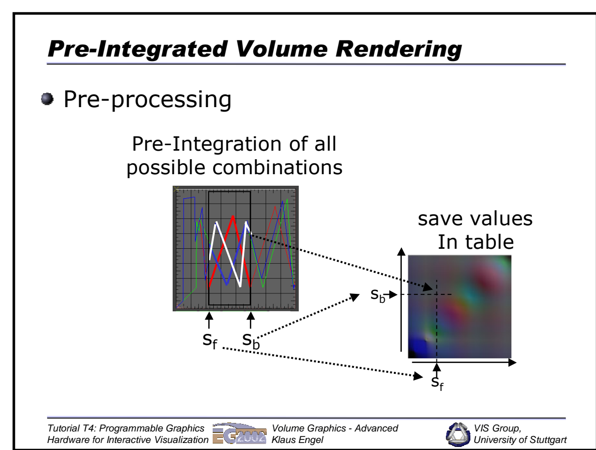

High Frequency Transfer Functions
Red: piecewise linear
Green: random
Blue: piecewise linear
Alpha: identity
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

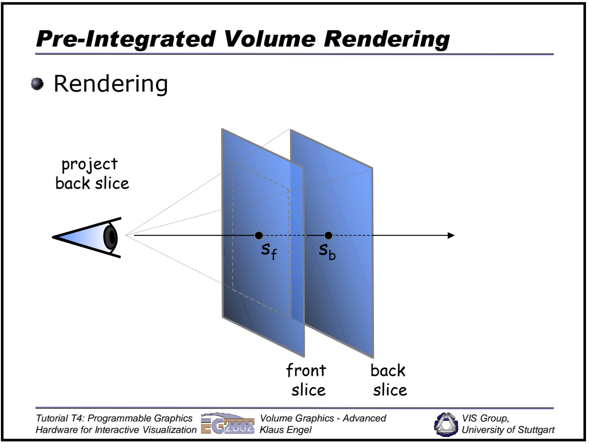

Pre- and Postclassification
Classification
Interpolation
Transfer-Functions
Voxel
Pre-Classification
Interpolation
Classification
Post-Classification
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

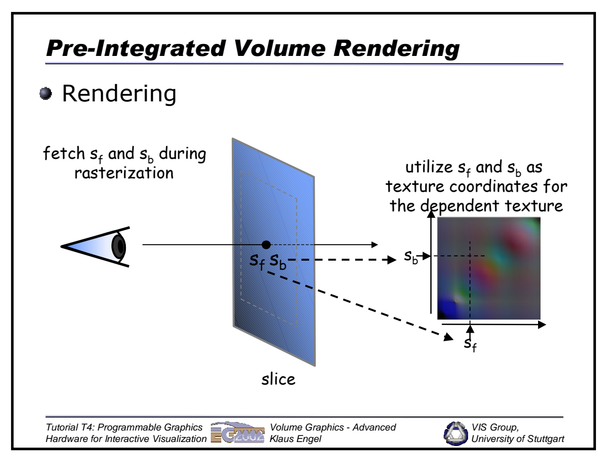

Pre- and Postclassification
• Pre-Classification
– no high-frequencies from TF
• Post-Classification
– reproduces high-frequencies on the slice polygons
– to reproduce high frequencies in between slices
=> render many slices
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

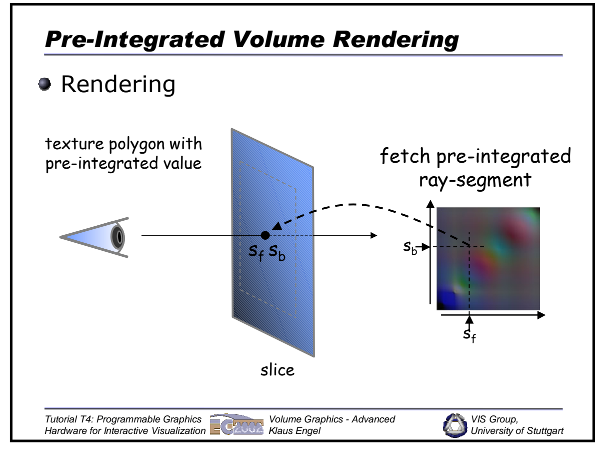

Pre-Integrated Classification
• Pre-Integrated Classification
– split numerical integration into
- one pre-integration for the TF
- one integration for the scalar field
– slab-by-slab rendering
slice-by-slice
slab-by-slab
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

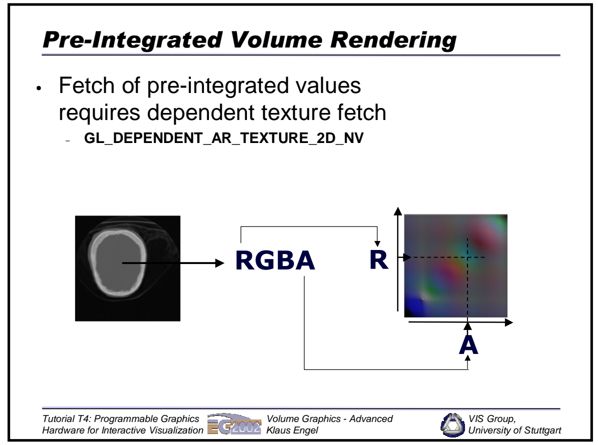

Pre-Integrated Classification
d
shell rendering
A.Parallel Projection
B. Perspective projection
viewing rays
image plane
eye
slice rendering
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

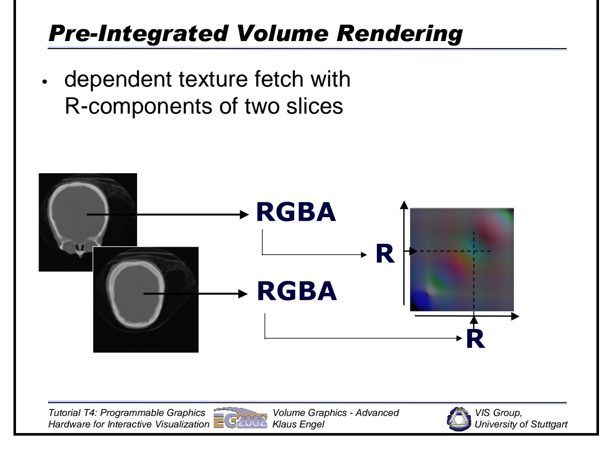

Pre-Integrated Volume Rendering
Pre-processing
Pre-Integration of all possible combinations
save values
In table
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

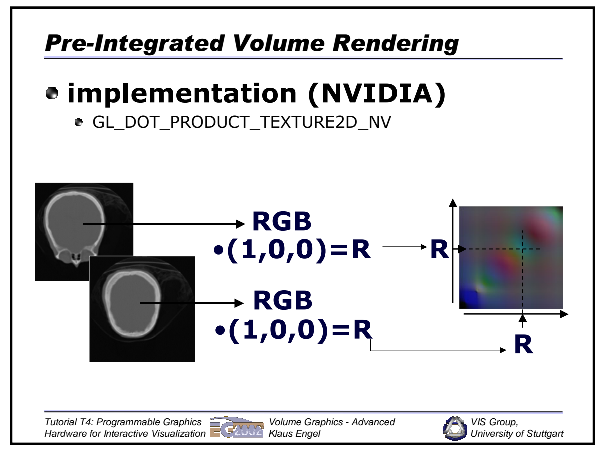

Pre-Integrated Volume Rendering
Rendering
project back slice
front slice
back slice
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

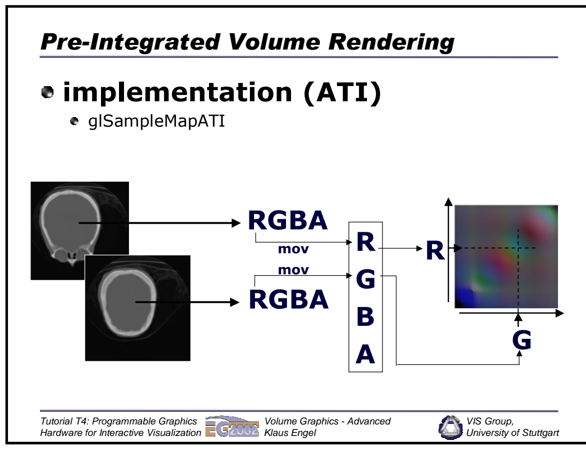

Pre-Integrated Volume Rendering
Rendering
fetch s_f and s_b during rasterization
utilize s_f and s_b as texture coordinates for the dependent texture
s_f s_b
s_b
s_f
slice
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced Klaus Engel
VIS Group, University of Stuttgart

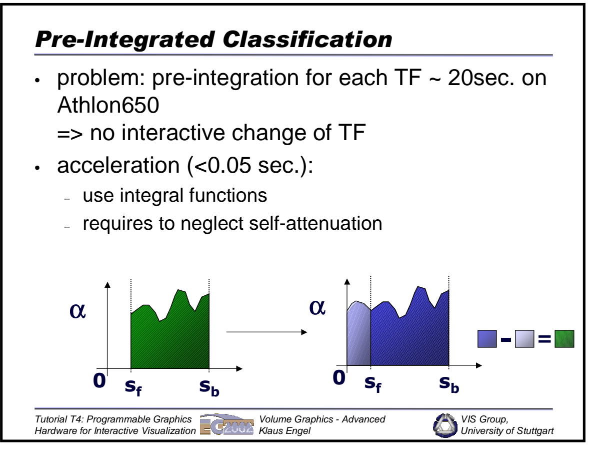

Pre-Integrated Volume Rendering
Rendering
texture polygon with pre-integrated value
fetch pre-integrated ray-segment
s_f s_b
s_b
s_f
slice
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced Klaus Engel
VIS Group, University of Stuttgart

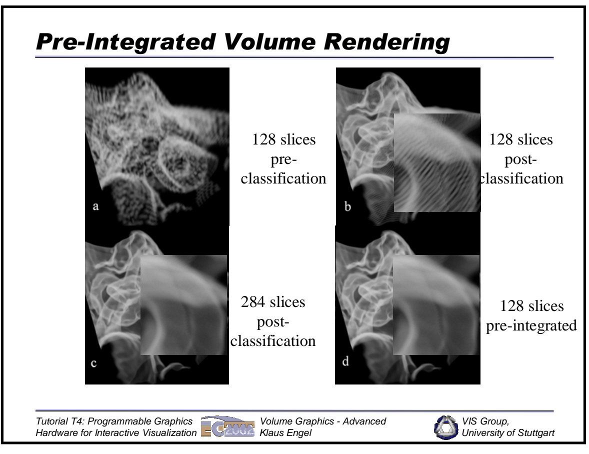

Pre-Integrated Volume Rendering
Fetch of pre-integrated values requires dependent texture fetch
GL_DEPENDENT_AR_TEXTURE_2D_NV
RGBA
R
A
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced Klaus Engel
VIS Group, University of Stuttgart

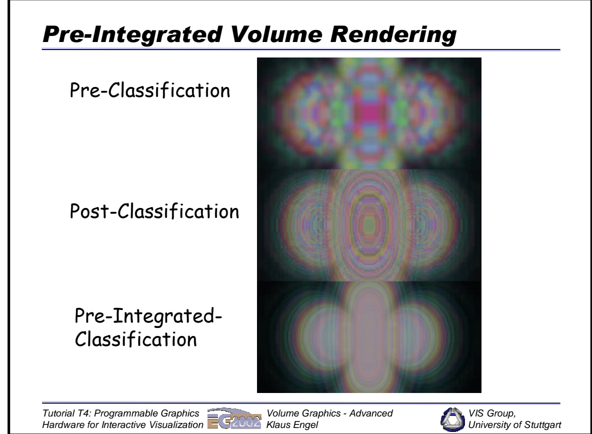

Pre-Integrated Volume Rendering
dependent texture fetch with R-components of two slices
RGBA
R
RGBA
R
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced Klaus Engel
VIS Group, University of Stuttgart

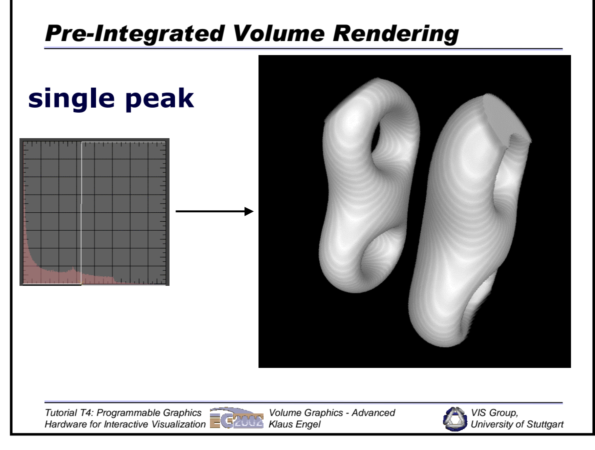

Pre-Integrated Volume Rendering
implementation (NVIDIA)
GL_DOT_PRODUCT_TEXTURE2D_NV
RGB
•(1,0,0)=R
R
RGB
•(1,0,0)=R
R
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced Klaus Engel
VIS Group, University of Stuttgart

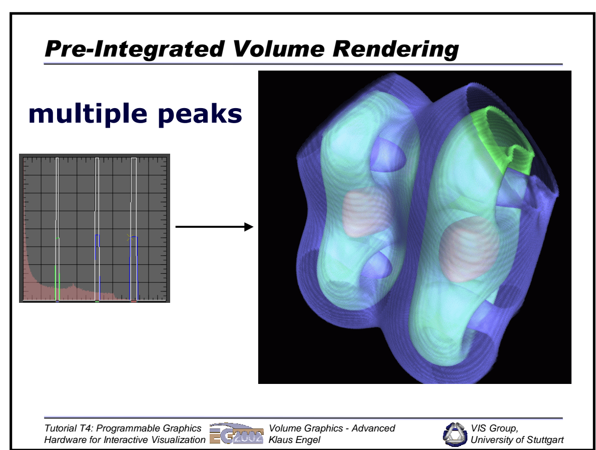

Pre-Integrated Volume Rendering
implementation (ATI)
glSampleMapATI
RGBA
mov
mov
RGBA
R
G
B
A
R
G
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced Klaus Engel
VIS Group, University of Stuttgart

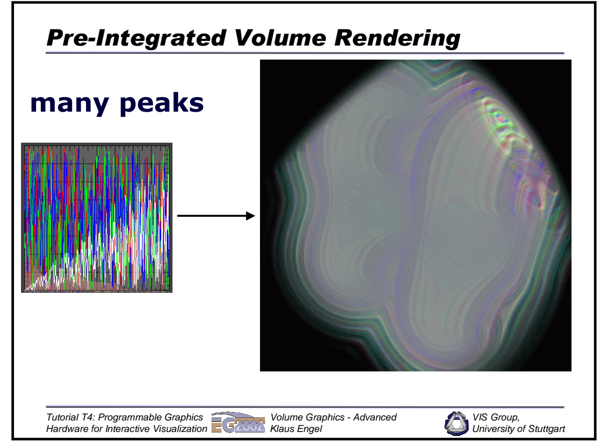

Pre-Integrated Classification
problem: pre-integration for each TF ~ 20sec. on Athlon650
=> no interactive change of TF
acceleration (<0.05 sec.):
use integral functions
requires to neglect self-attenuation
α
0
s_f
s_b
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

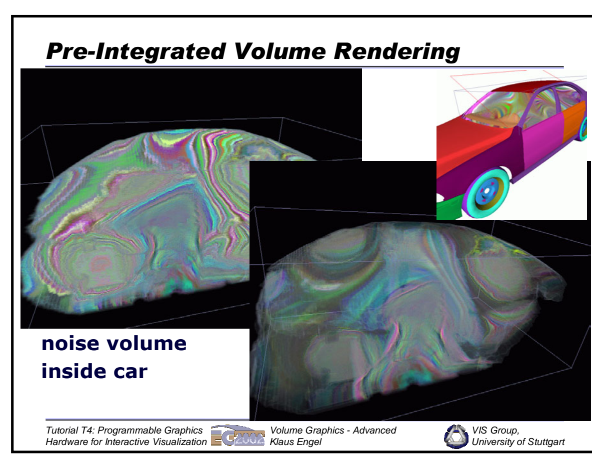

Pre-Integrated Volume Rendering
128 slices pre-classification
128 slices post-classification
284 slices post-classification
128 slices pre-integrated
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

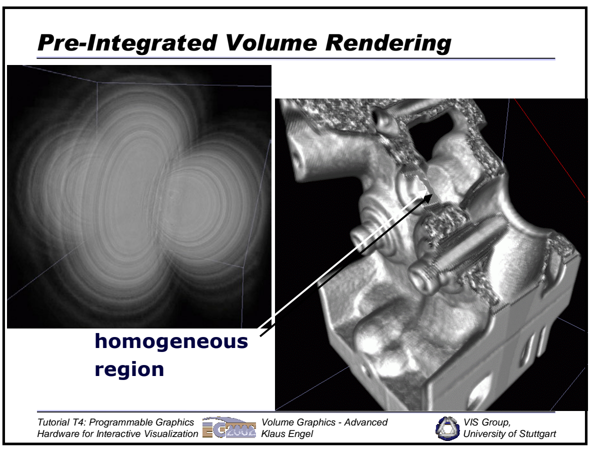

Pre-Integrated Volume Rendering
Pre-Classification
Post-Classification
Pre-Integrated-Classification
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

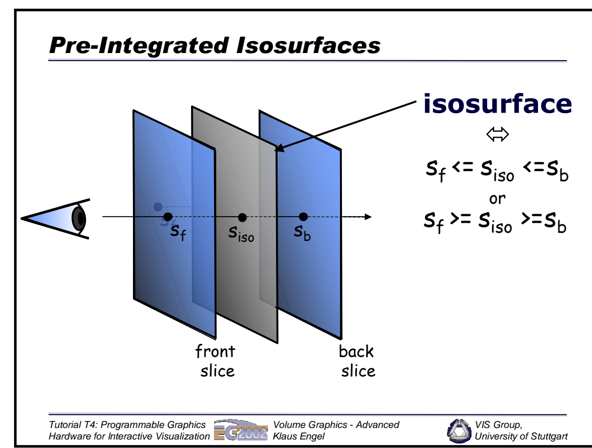

Pre-Integrated Volume Rendering
single peak
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

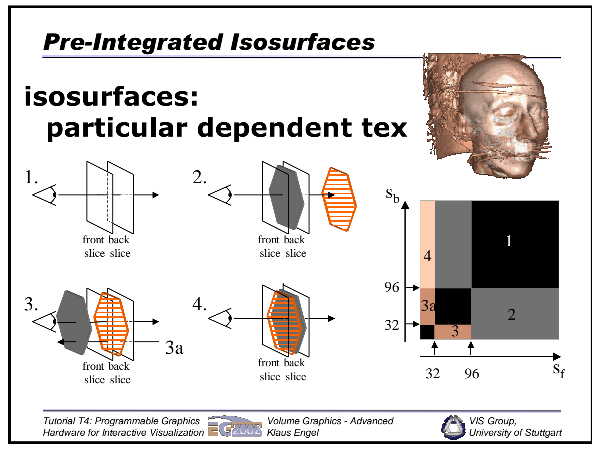

Pre-Integrated Volume Rendering
multiple peaks
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

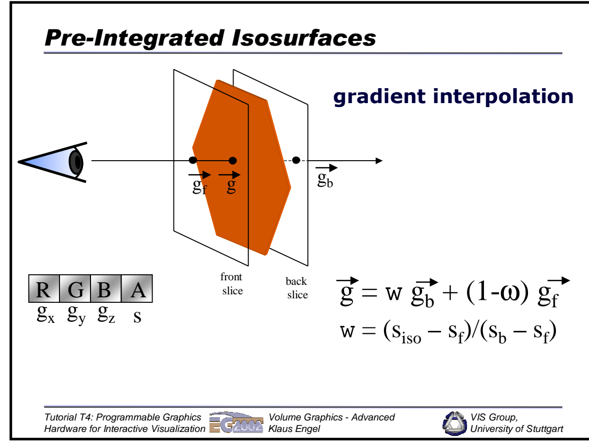

Pre-Integrated Volume Rendering
many peaks
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

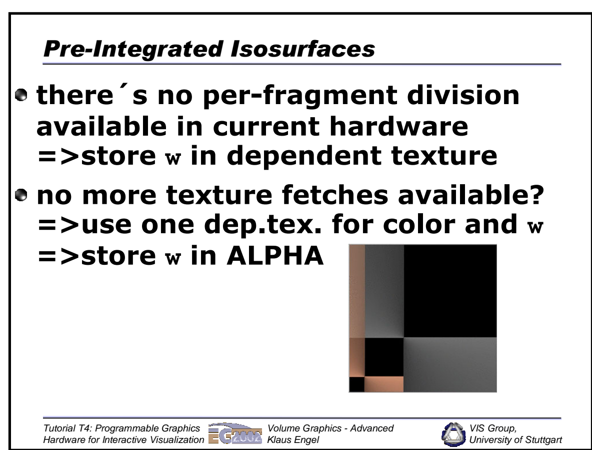

Pre-Integrated Volume Rendering
noise volume
inside car
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

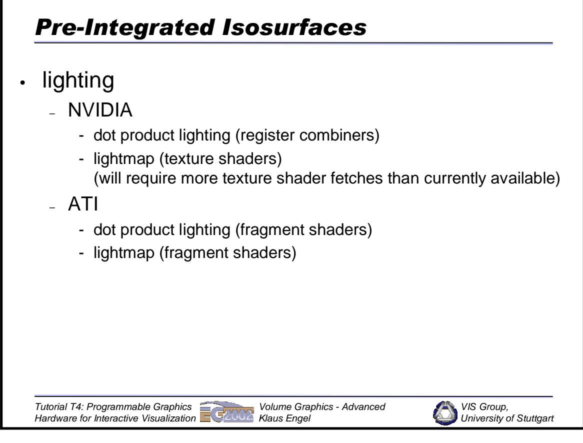

Pre-Integrated Volume Rendering
homogeneous
region
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

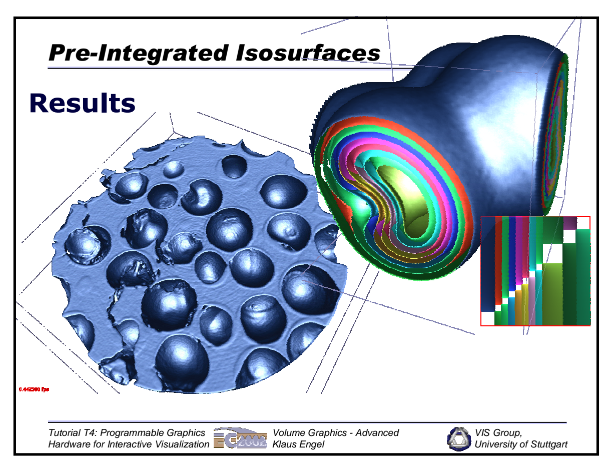

Pre-Integrated Isosurfaces
isosurface
⇔
s_f <= s_{iso} <= s_b
or
s_f >= s_{iso} >= s_b
s_f
s_{iso}
s_b
front
slice
back
slice
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

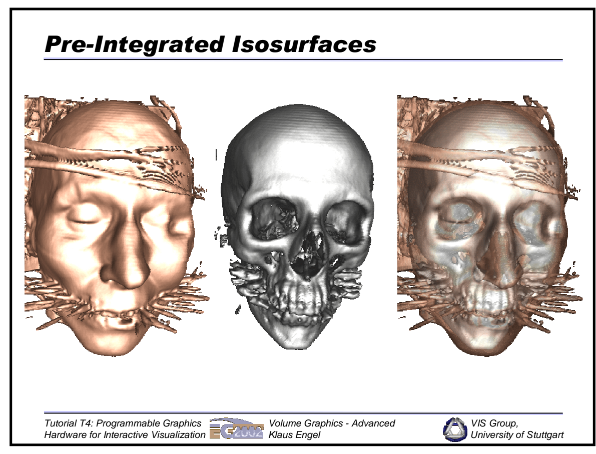

Pre-Integrated Isosurfaces
isosurfaces:
particular dependent tex
1.
2.
3.
4.
front back
slice slice
3a
s_b
s_f
1
2
3
3a
4
96
32
32
96
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

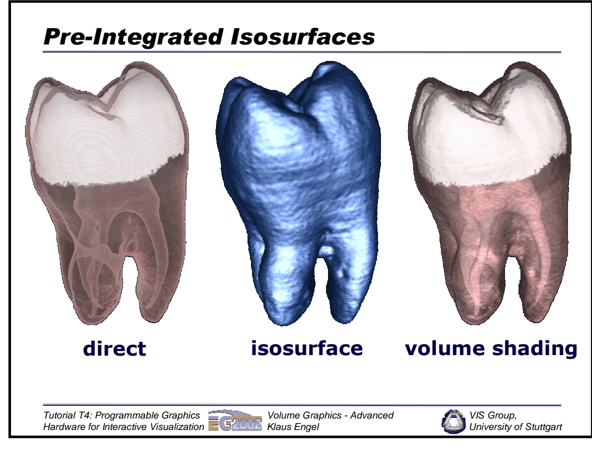

Pre-Integrated Isosurfaces
gradient interpolation
$\vec{g}_f$
$\vec{g}$
$\vec{g}_b$
R G B A
g_x g_y g_z s
front
slice
back
slice
$\vec{g} = w\,\vec{g}_b + (1-\omega)\,\vec{g}_f$
$w = (s_{iso} - s_f)/(s_b - s_f)$
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

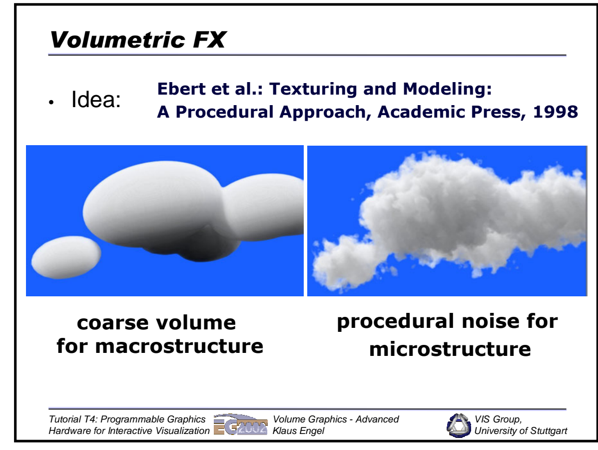

Pre-Integrated Isosurfaces
there´s no per-fragment division available in current hardware
=>store w in dependent texture
no more texture fetches available?
=>use one dep.tex. for color and w
=>store w in ALPHA
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group, University of Stuttgart

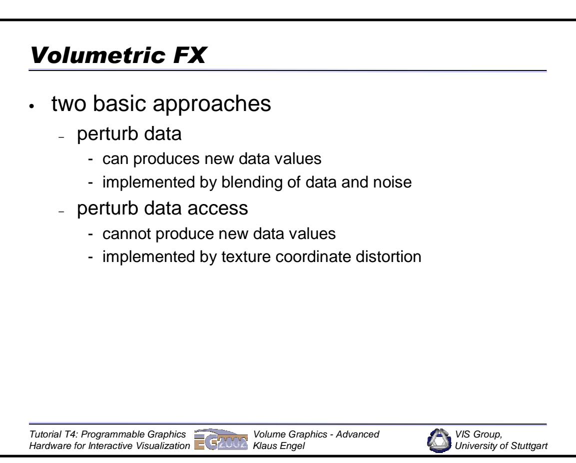

Pre-Integrated Isosurfaces
• lighting
– NVIDIA
- dot product lighting (register combiners)
- lightmap (texture shaders)
(will require more texture shader fetches than currently available)
– ATI
- dot product lighting (fragment shaders)
- lightmap (fragment shaders)
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group,
University of Stuttgart

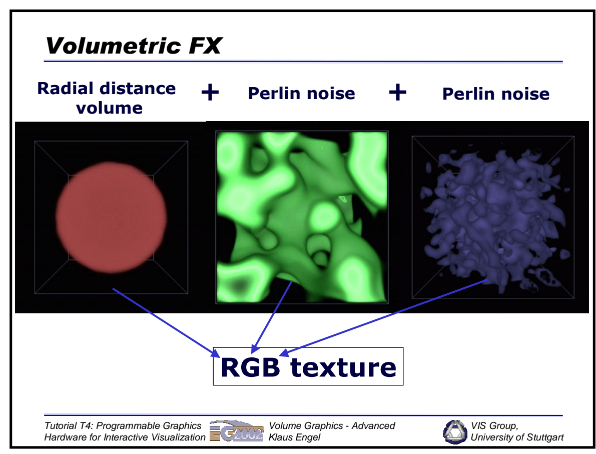

Pre-Integrated Isosurfaces
Results
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group,
University of Stuttgart

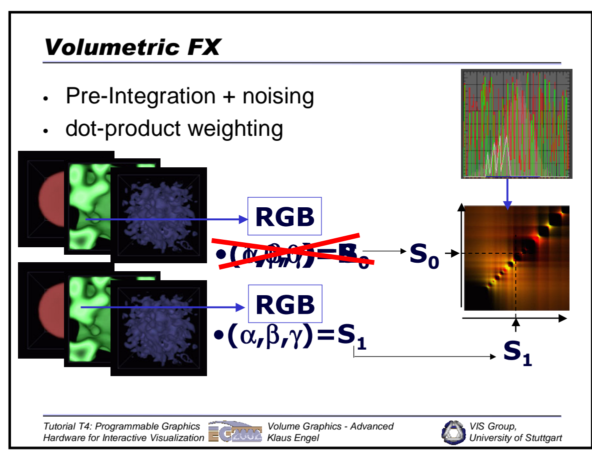

Pre-Integrated Isosurfaces
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group,
University of Stuttgart

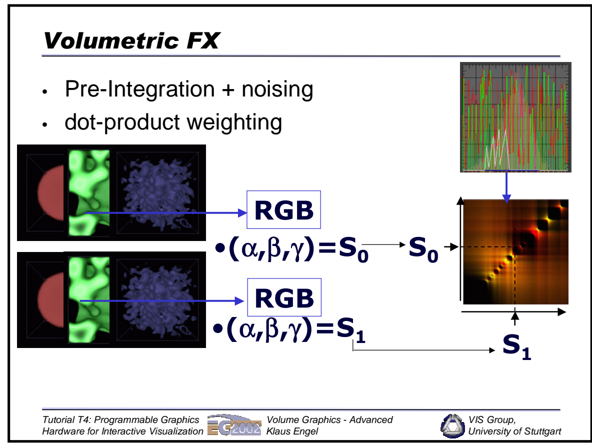

Pre-Integrated Isosurfaces
direct
isosurface
volume shading
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group,
University of Stuttgart

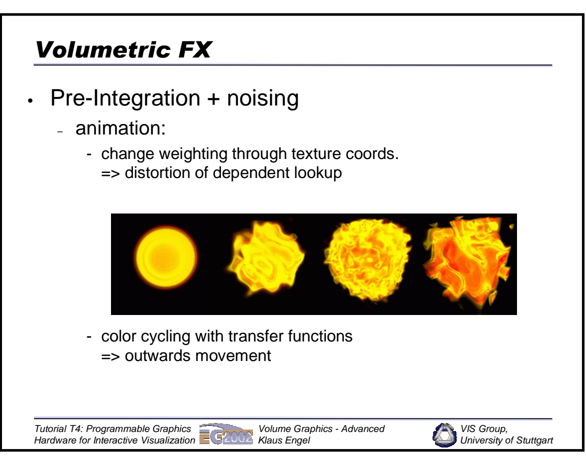

Volumetric FX
• Idea:
Ebert et al.: Texturing and Modeling:
A Procedural Approach, Academic Press, 1998
coarse volume
for macrostructure
procedural noise for
microstructure
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group,
University of Stuttgart


Volumetric FX
• two basic approaches
– perturb data
- can produces new data values
- implemented by blending of data and noise
– perturb data access
- cannot produce new data values
- implemented by texture coordinate distortion
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Volume Graphics - Advanced
Klaus Engel
VIS Group,
University of Stuttgart





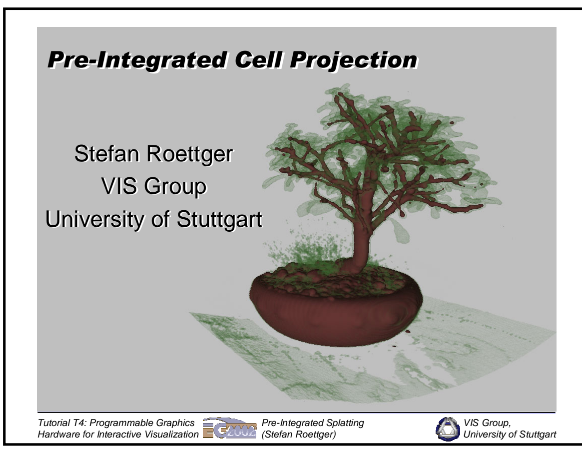




Conclusions

- Pre-Integrated Volume Rendering
  - great quality, less slices
  - less error due to pre-processing
  - reproduces high frequencies from transfer function
  - arbitrary number of lit isosurfaces in a single pass
- Volumetric FX
  - coarse macrostructure, details from noise and TF
  - animate TF and weighting of volumes for dynamics
  - give something back to the gaming industry in return for the new great graphics hardware

Tutorial T4: Programmable Graphics Hardware for Interactive Visualization — Volume Graphics - Advanced, Klaus Engel — VIS Group, University of Stuttgart



Unstructured Volume Rendering

- Given an irregular volumetric mesh
- How can the volume be rendered accurately?

- Obvious: Resampling to regular grid
- Too heavy: Ray tracing

- Ray casting
- Sweep plane algorithms
- Cell projection

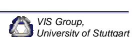
Resampling to Regular Grid

- Hardware-accelerated: Westermann et al. [1]
  - Use graphics hardware to compute the intersection of a set of tetrahedra with a slice plane.

- Comparison with software: Weiler et al. [2]
  - Comparison hardware/software approach.
  - It turns out that software is not significantly slower than a PC hardware based resampling approach.

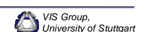
Ray Tracing

- Basic Algorithm:
  - Shoot rays through the volume and integrate the ray integral at each cell intersection in a back to front fashion taking scattering into account.
- Pros:
  - Physically based accurate method
- Cons:
  - Dependend on view port resolution
  - Very slow, but can be parallelized



Ray Casting

- Basic algorithm:
  - Shoot rays through the volume in a front to back fashion and accumulate the ray integral neglecting scattering processes inside the media.
- Pros:
  - Faster because of simplified physical model
  - Early ray termination and space leaping
  - Runs on every platform
- Cons:
  - Dependend on view port size



Sweep Plane Algorithms

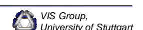
- Basic algorithm:
  - Use a sweep plane to process the cells in a back to front fashion and compose the cells.
  - E.g. ZSWEEP (Farias et al. [3])
- Pros:
  - Runs on every platform
  - Easy to parallelize
- Cons:
  - Dependent on view port size

Splatting

- Basic algorithm:
  - Approximate each cell with a volumetric blob and compose the footprints in a back to front fashion.
- Pros:
  - Good performance, since footprints can be rendered by the graphics hardware.
  - Object order method
- Cons:
  - Footprint is only an approximation -> blurry look

Cell Projection

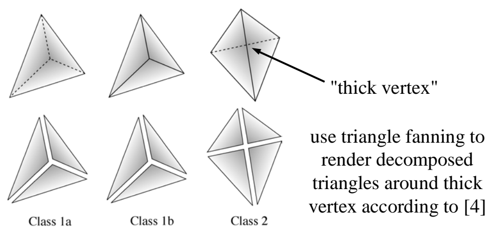
- „Accurate Splatting“
- Basic algorithm:
  - Sort the cells in a back to front fashion and compose the geometric projections of the cells.
- Pros:
  - Hardware-accelerated / Accurate composing
  - Object order method
- Cons:
  - Projection of non-tetrahedral cells is difficult
  - Topological sort needed



Tutorial T4: Programmable Graphics Hardware for Interactive Visualization | Pre-Integrated Splatting (Stefan Roettger) | VIS Group, University of Stuttgart

The PT Algorithm of Shirley and Tuchman

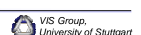
- We distinguish between two different non-degenerate classes of projected tetrahedra:



Tutorial T4: Programmable Graphics Hardware for Interactive Visualization | Pre-Integrated Splatting (Stefan Roettger) | VIS Group, University of Stuttgart

Topological Sorting

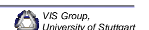
- Numerical (Wittenbrink et al. [5])
  - fast but incorrect
- MPVO (Williams et al. [6])
  - sorts convex polyhedra by processing the hidden relationship via a priority queue
- XMPVO (Silva et al. [7])
- BSP-XMPVO (Comba et al. [8])
  - constructs external dependencies via BSP-tree
- MPVOC (Kraus et al. [9])
  - extension of MPVO which can handle cycles



Tutorial T4: Programmable Graphics Hardware for Interactive Visualization | Pre-Integrated Splatting (Stefan Roettger) | VIS Group, University of Stuttgart

Optical Model of Williams et al.

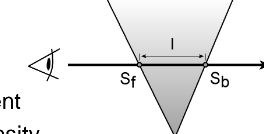
- "Volume density optical model" (Williams et al. [10])
- Emission and absorption along each ray segment depends on scalar density function f(x,y,z).
- The scalar optical density and chromaticity is defined by the transfer functions $\rho(f(x,y,z))$ and $\kappa(f(x,y,z))$.



Tutorial T4: Programmable Graphics Hardware for Interactive Visualization | Pre-Integrated Splatting (Stefan Roettger) | VIS Group, University of Stuttgart

Approximation of Stein et al.

- Approximation of the ray integral by Stein et al. [11] for linear transfer function ρ and a constant transfer function κ:
- $\alpha=1-\exp(-l\tau)$
- l=length of ray segment
- τ=average optical density
- $\tau=(\rho(S_f)+\rho(S_b))/2$



Tutorial T4: Programmable Graphics Hardware for Interactive Visualization | Pre-Integrated Splatting (Stefan Roettger) | VIS Group, University of Stuttgart

Approximation of Stein et al.

- Put alpha(l,τ) into 2D texture and assign (l,τ) as texture coordinates to projected vertices.

- Linear interpolation of texture coordinates yields exponentially interpolated opacities.

Approximation of Stein et al.

- Pros:
  - Hardware-accelerated
  - 2D texture mapping only
  - Equivalent to 1D dependent texture today
- Cons:
  - Restricted application of transfer functions
  - Transfer functions cannot be taken into account inside the tetrahadra

Pre-Integrated Cell Projection

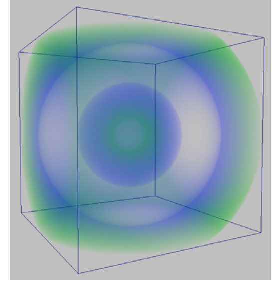
- **Observation**: The ray integral depends only on S_f,S_b, and l.
- Pre-compute the three-dimensional ray integral by numerical integration and store it in a 3D texture.
- Assign appropriate 3D texture coords (S_f,S_b,l) to each projected vertex and use 3D texture mapping to perform per-pixel exact compositing.

- This approach is known as Pre-Integration [12]

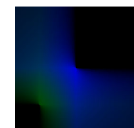
Numerical Pre-Integration

```
iterate over l
  iterate over Sb
    iterate over Sf
      chromaticity C0=0
      transparency T0=1
      iterate over #steps (i=0...n-1)
        S=i/(n-1)Sb+(1-i/(n-1))Sf
        compute emission=κ(S)*l/(n-1)
        compute absorption=exp(-ρ(S)*l/(n-1))
        Ci+1=Ci*absorption+emission
        Ti+1=Ti*absorption
```


Pre-Integration Example

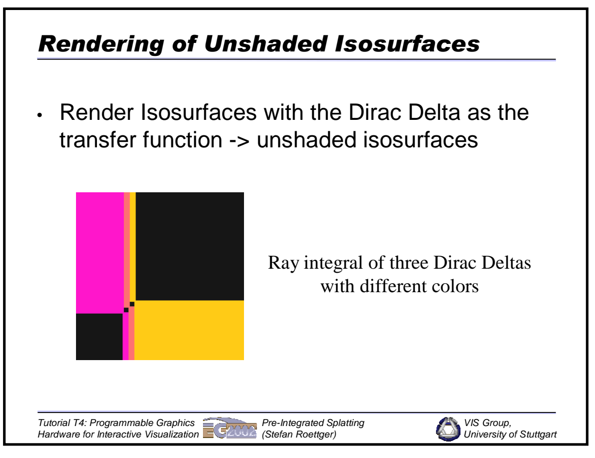


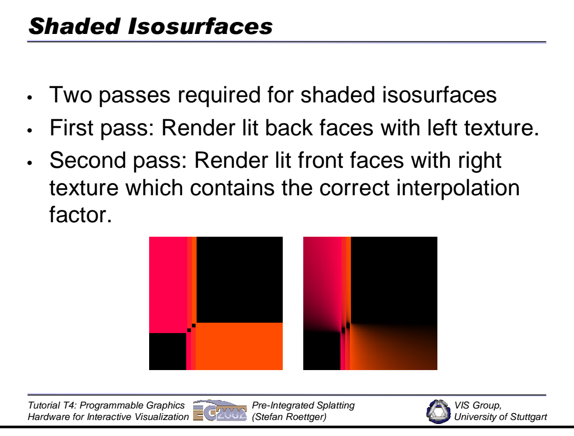
Simple spherical distance volume rendered with piecewise transparent transfer function. The integrated chromaticity is shown below.

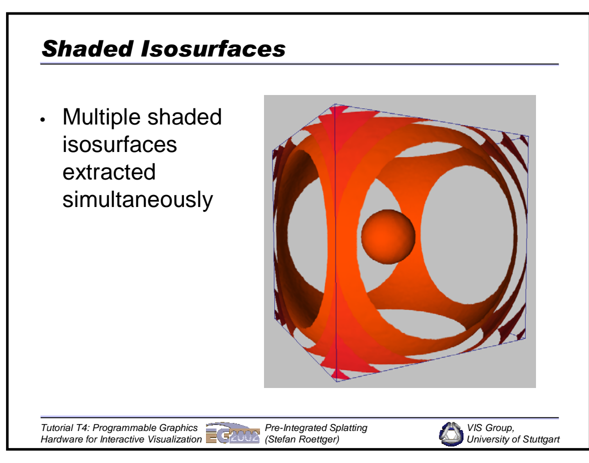


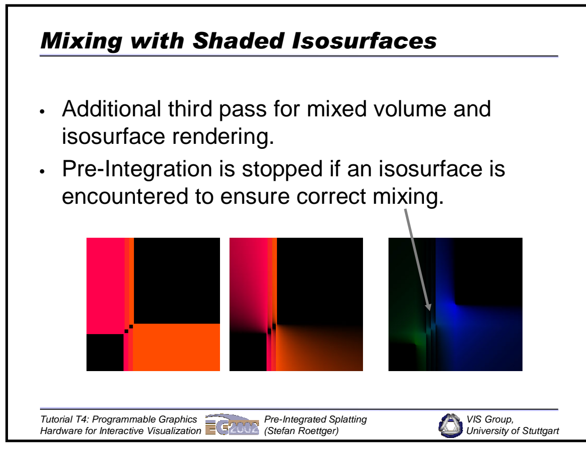
Pre-Integrated Cell Projection

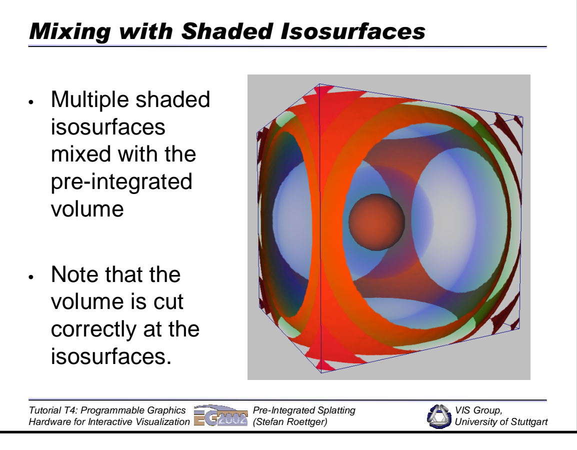
- Pros:
  - Arbitrary transfer functions can be used
  - Accuracy only limited by the size of the pre-intgration table
  - Per-pixel exact rendering
  - Unshaded isosurfaces with Dirac Delta
- Cons:
  - 3D textures not available everywhere
  - Memory consumption of 3D textures

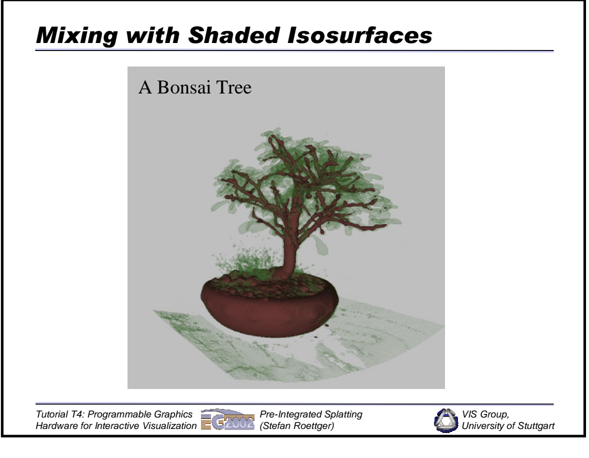

Rendering of Unshaded Isosurfaces
Render Isosurfaces with the Dirac Delta as the transfer function -> unshaded isosurfaces
Ray integral of three Dirac Deltas with different colors
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Pre-Integrated Splatting (Stefan Roettger)
VIS Group, University of Stuttgart


Shaded Isosurfaces
Two passes required for shaded isosurfaces
First pass: Render lit back faces with left texture.
Second pass: Render lit front faces with right texture which contains the correct interpolation factor.
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Pre-Integrated Splatting (Stefan Roettger)
VIS Group, University of Stuttgart


Shaded Isosurfaces
Multiple shaded isosurfaces extracted simultaneously
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Pre-Integrated Splatting (Stefan Roettger)
VIS Group, University of Stuttgart

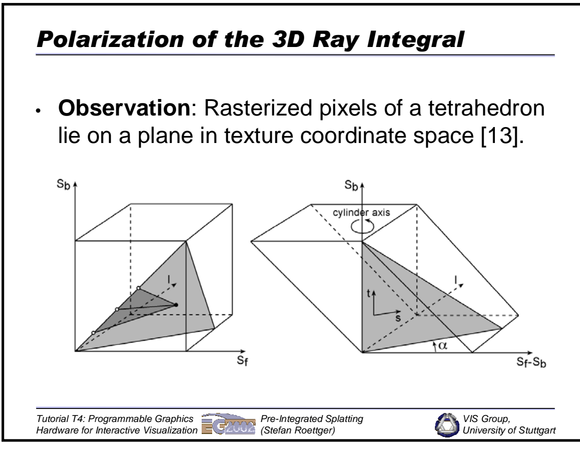

Mixing with Shaded Isosurfaces
Additional third pass for mixed volume and isosurface rendering.
Pre-Integration is stopped if an isosurface is encountered to ensure correct mixing.
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Pre-Integrated Splatting (Stefan Roettger)
VIS Group, University of Stuttgart


Mixing with Shaded Isosurfaces
Multiple shaded isosurfaces mixed with the pre-integrated volume
Note that the volume is cut correctly at the isosurfaces.
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Pre-Integrated Splatting (Stefan Roettger)
VIS Group, University of Stuttgart

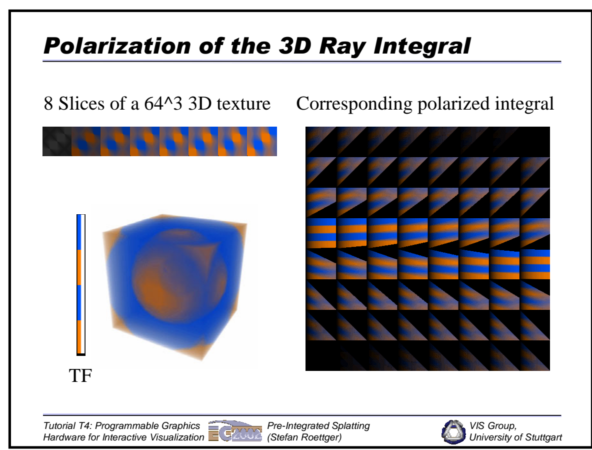

Mixing with Shaded Isosurfaces
A Bonsai Tree
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Pre-Integrated Splatting (Stefan Roettger)
VIS Group, University of Stuttgart

Polarization of the 3D Ray Integral

- **Observation**: Rasterized pixels of a tetrahedron lie on a plane in texture coordinate space [13].

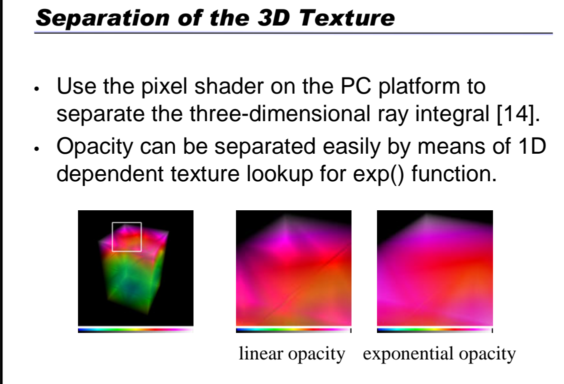


Polarization of the 3D Ray Integral



Separation of the 3D Texture

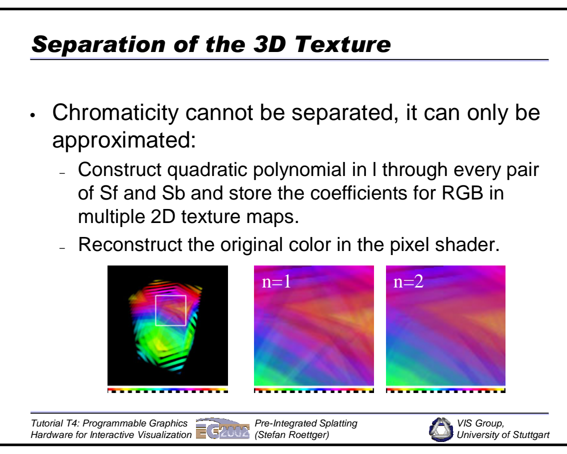
- Use the pixel shader on the PC platform to separate the three-dimensional ray integral [14].
- Opacity can be separated easily by means of 1D dependent texture lookup for exp() function.



linear opacity exponential opacity

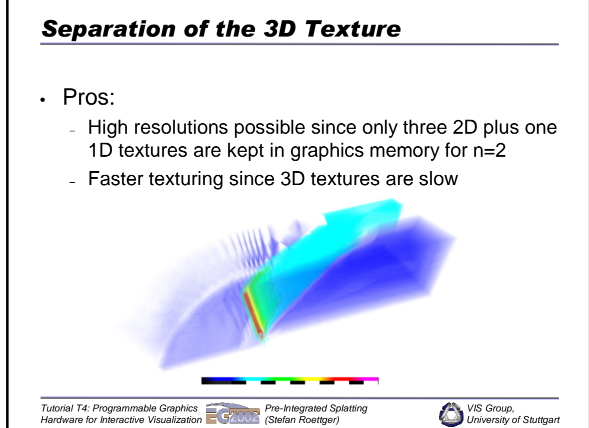
Separation of the 3D Texture

- Chromaticity cannot be separated, it can only be approximated:
  - Construct quadratic polynomial in l through every pair of Sf and Sb and store the coefficients for RGB in multiple 2D texture maps.
  - Reconstruct the original color in the pixel shader.



Separation of the 3D Texture

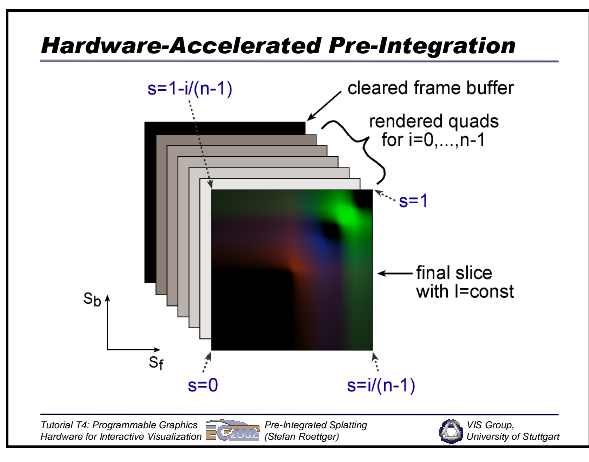
- Pros:
  - High resolutions possible since only three 2D plus one 1D textures are kept in graphics memory for n=2
  - Faster texturing since 3D textures are slow



Hardware-Accelerated Pre-Integration

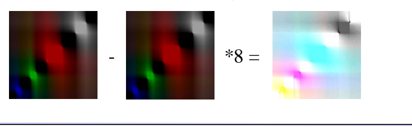
- Accelerate the numerical integration using graphics hardware to maintain interactive updates of the pre-intgration table [13].
- Prerequisite for comfortable exploration of unstructured data sets
- Basic Idea: Put the transfer function in a 1D texture and compute one slice of the 3D texture for l=const in parallel.

Hardware-Accelerated Pre-Integration



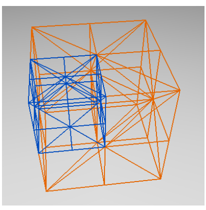
Hardware-Accelerated Pre-Integration

- Pros:
  - Numerical integration takes >1min for a 512x512x64 pre-integration table.
  - With hardware acceleration update rates of <1s are achieved easily.
- Cons:
  - Quantization artifacts but with pixel shader 14 bits [14]



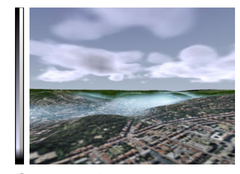
Application to Cloud Rendering

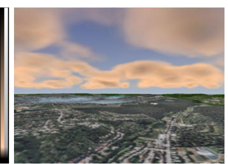
- View-dependent simplification of regular volume
- Screen space error of the simplification is bounded by a user definable threshold.
- Octree hierarchy is decomposed into tetrahedra



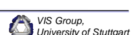
Application to Cloud Rendering

- Clouds generated with 3D Perlin noise





- Ground fog displayed by stacking prisms onto each triangle that is generated by the C-LOD terrain renderer -> avg. 25 fps.

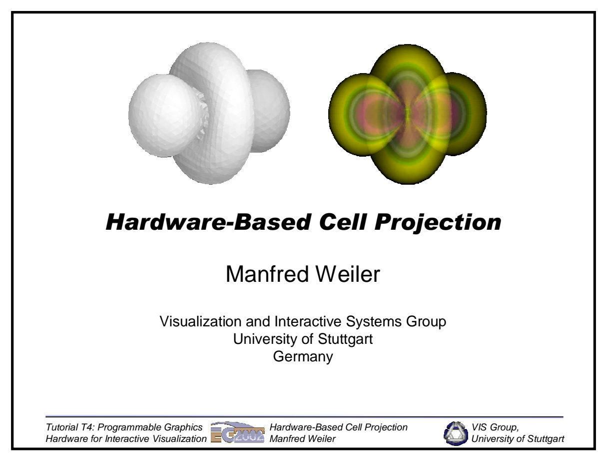


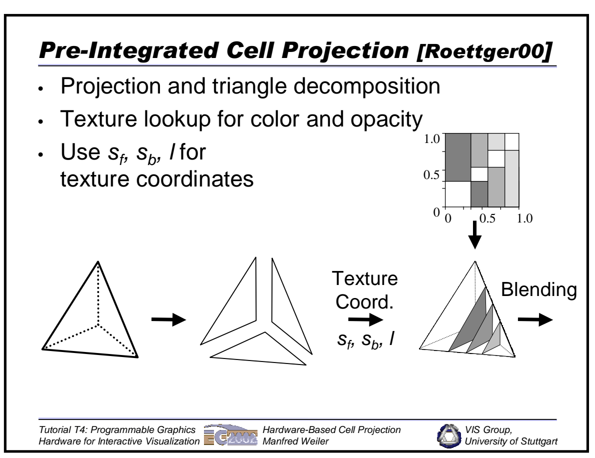
Bibliography (1)

- [1] R. Westermann. The Rendering of Unstructured Grids Revisited. Proc. IEEE VisSym '01. 2001.
- [2] M. Weiler and Th. Ertl. Hardware-Software-Balanced Resampling for the Interactive Visualization of Unstructured Grids. Proc. of IEEE Visualization '01, 2001.
- [3] R. Farias, J. Mitchell, and C. Silva. ZSWEEP: An Efficient and Exact Projection Algorithm for Unstructured Volume Rendering. In Proc. IEEE VolVis '00. 2000.
- [4] P. Shirley and A. Tuchman. A Polygonal Approximation for Direct Scalar Volume Rendering. In Proc. San Diego Workshop on Volume Visualization, pages 63-70, 1990.
- [5] C. Wittenbrink. CellFast: Interactive Unstructured Volume Rendering. In IEEE Visualization '99 Late Breaking Hot Topics, pages 21-24, 1999.
- [6] P. Williams. Visibility Ordering Meshed Polyhedra. ACM Transactions on Graphics, volume 11(2), pages 103-126, 1992.
- [7] C. Silva, J. Mitchell, and P. Williams. An Exact Interactive Time Visibility Ordering Algorithm for Polyhedral Cell Complexes. In Proceedings of the 1998 Symposium on Volume Visualization, pages 87-94. ACM Press, 1998.

Bibliography (2)

- [8] J. Comba, J. Klosowski, N. Max, J. Mitchell, C. Silva, and P. Williams. Fast Polyhedral Cell Sorting for Interactive Rendering of Unstructured Grids. Computer Graphics Forum, volume 18(3), pages 369-376, 1999.
- [9] M. Kraus and Th. Ertl. Cell-Projection of Cyclic Meshes. Proc. of IEEE Visualization '01, pages 215-222, 2001.
- [10] P. Williams and N. Max. A Volume Density Optical Model. 1992 Workshop on Volume Visualization}, pages 61--68, 1992.
- [11] C. Stein, B. Becker, and N. Max. Sorting and Hardware Assisted Rendering for Volume Visualization. In Proc. 1994 Symposium on Volume Visualization, pages 83-90, 1994.
- [12] S. Roettger, M. Kraus, and Th. Ertl. Hardware-Accelerated Volume and Isosurface Rendering Based on Cell-Projection. In IEEE Proc. Visualization 2000, pages 109-116, 2000.
- [13] S. Roettger and Th. Ertl. A Two-Step Approach for Interactive Pre-Integrated Volume Rendering of Unstructured Grids. In Proc. of the 2002 Symposium on Volume Visualization. ACM Press, 2002 (to appear).
- [14] S. Guthe, S. Roettger, A. Schieber, W. Strasser, and Th. Ertl. High-Quality Unstructured Volume Rendering on the PC Platform. In EG/SIGGRAPH Graphics Hardware Workshop '02. 2002 (to appear).





Avoiding the Triangle Decomposition

- Problems:
  - View-dependent
  - Computationally expensive
  - Bandwidth-limited data transfer to graphics adapter
  - Prohibits optimization
    - Display lists
    - Vertex arrays
- Why triangle decomposition?
  - Non-linear variation of
    - Scalar value on the back face
    - Thickness

Ray Casting

- Adopt idea of parametrical line clipping for convex polyhedra
  - Compute ray parameters for every face intersection
  - Classify intersection point (potentially entering/leaving)
  - Exit point: minimum of leaving intersections

$$\vec{r}(t) = \vec{e} + t\vec{d}$$

$$t_i = \frac{(\vec{e} \cdot \vec{n}_i) - a_i}{(\vec{d} \cdot \vec{n}_i)}$$

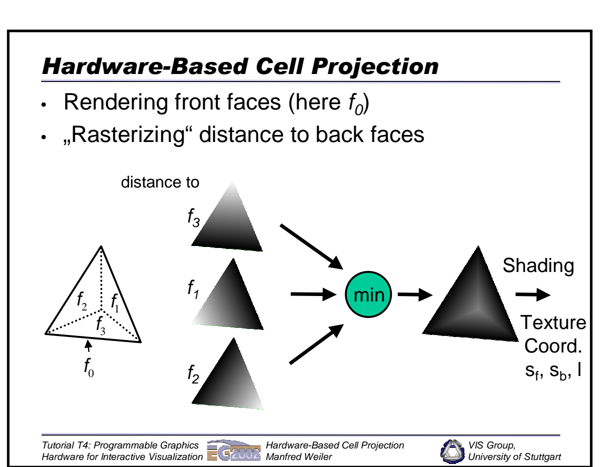
v_1, $\vec{n}_2$, f_2, e, $\vec{d}$, t_0, t_2, t_1, f_0, v_0, f_1, $\vec{n}_1$, v_2

Per-Fragment Ray Casting

- Rendering front faces
- Ray Casting via fragment operations
  - Ray starting at entry point
  - Entry point determined by fragment coordinates
  - Only exit point unknown
  - Leaving intersections correspond to positive ray parameters
- s_f primary color
  linear interpolated
- l equals ray parameter
- s_b from gradient and ray parameter

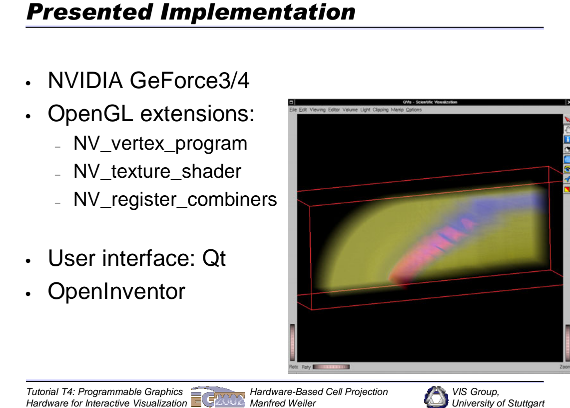
$$s_b = s_f + (\vec{g} \cdot \vec{d})\, t_{\min}$$

v_1, $\vec{n}_2$, $\vec{d}$, $\vec{v}$, f_2, t_2, t_1, f_0, v_0, f_1, $\vec{n}_1$, v_2



Hardware Limitations

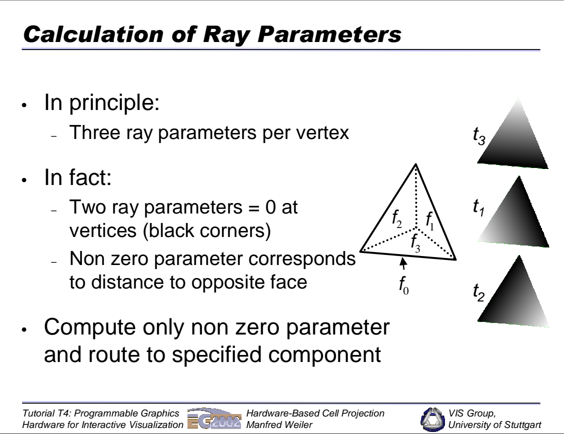
- Single-pass per-fragment ray casting not yet possible
  - Arithmetic operations only after texture lookup (GeForce3)
  - Number of arithmetic operations (Radeon, GeForce3)
- Can be implemented in DirectX9
- Instead: combine vertex and fragment operations
  - Per-Vertex: (vertex shader, vertex program)
    - Compute ray parameters t_i
  - Per-Fragment: (pixel shader, fragment shader, texture shader, register combiners)
    - Compute minimum
    - Evaluate volume rendering integral
- Orthographic projection only!!
  - Because of per-vertex computations



Presented Implementation

- NVIDIA GeForce3/4
- OpenGL extensions:
  - NV_vertex_program
  - NV_texture_shader
  - NV_register_combiners
- User interface: Qt
- OpenInventor

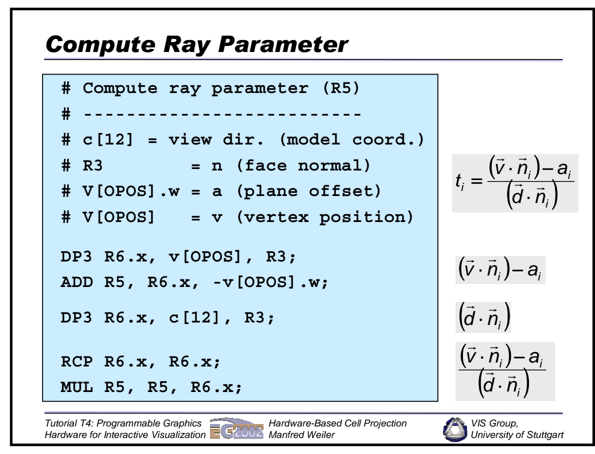
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization | Hardware-Based Cell Projection Manfred Weiler | VIS Group, University of Stuttgart



Calculation of Ray Parameters

- In principle:
  - Three ray parameters per vertex
- In fact:
  - Two ray parameters = 0 at vertices (black corners)
  - Non zero parameter corresponds to distance to opposite face
- Compute only non zero parameter and route to specified component

Tutorial T4: Programmable Graphics Hardware for Interactive Visualization | Hardware-Based Cell Projection Manfred Weiler | VIS Group, University of Stuttgart



Compute Ray Parameter

```
# Compute ray parameter (R5)
# -------------------------
# c[12] = view dir. (model coord.)
# R3        = n (face normal)
# V[OPOS].w = a (plane offset)
# V[OPOS]   = v (vertex position)

DP3 R6.x, v[OPOS], R3;
ADD R5, R6.x, -v[OPOS].w;

DP3 R6.x, c[12], R3;

RCP R6.x, R6.x;
MUL R5, R5, R6.x;
```

$$t_i = \frac{(\vec{v}\cdot\vec{n}_i)-a_i}{(\vec{d}\cdot\vec{n}_i)}$$

$(\vec{v}\cdot\vec{n}_i)-a_i$

$(\vec{d}\cdot\vec{n}_i)$

$\frac{(\vec{v}\cdot\vec{n}_i)-a_i}{(\vec{d}\cdot\vec{n}_i)}$

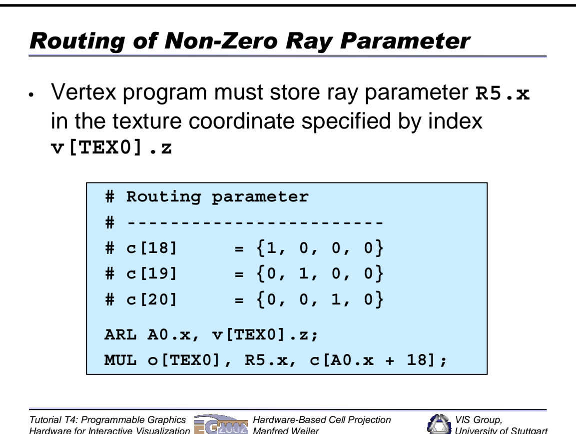
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization | Hardware-Based Cell Projection Manfred Weiler | VIS Group, University of Stuttgart

Invalidate Negative Parameters

```
# Overwrite negatives (R5)
# -----------------------
# R5    = distance
# c[8]  = {1000, ?, ?, 0}

# if (-R5 >= 0)  R10 = 1
# else           R10 = 0
# R5 = R10 * (-R5) + R5
# R5 = R10 * 1000 + R5

SGE R10, -R5, c[8].w;
MAD R5, R10, -R5, R5;
MAD R5, R10, c[8].x, R5;
```

- Overwrite negative ray parameters with large value (1000)
- Simulation of „conditional set“
- Only one additional register required

Tutorial T4: Programmable Graphics Hardware for Interactive Visualization | Hardware-Based Cell Projection Manfred Weiler | VIS Group, University of Stuttgart

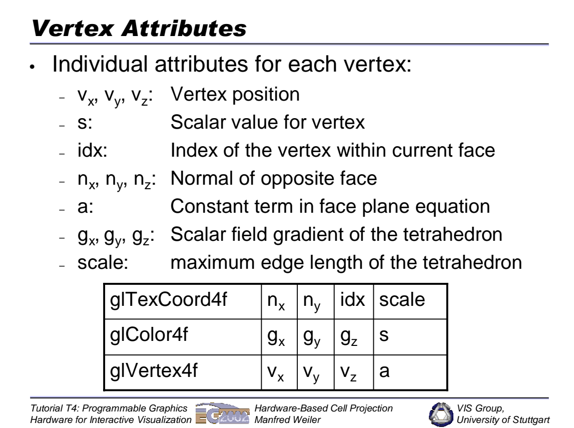


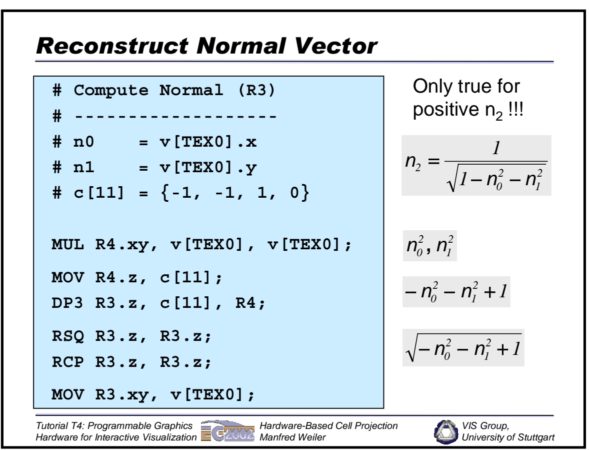
Routing of Non-Zero Ray Parameter

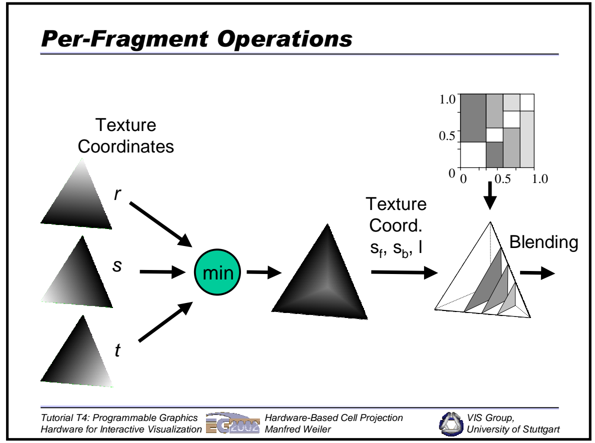
- Vertex program must store ray parameter `R5.x` in the texture coordinate specified by index `v[TEX0].z`

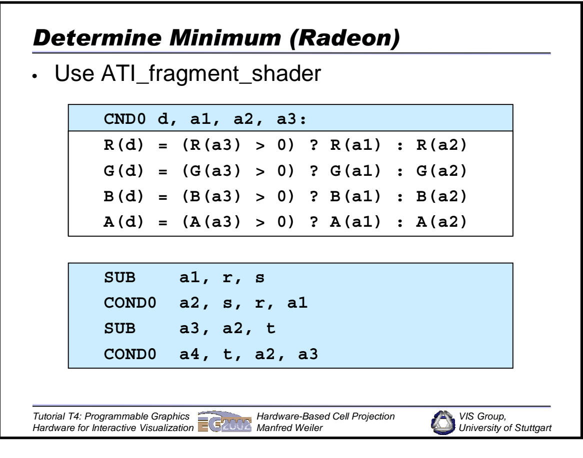
```
# Routing parameter
# -----------------------
# c[18]     = {1, 0, 0, 0}
# c[19]     = {0, 1, 0, 0}
# c[20]     = {0, 0, 1, 0}

ARL A0.x, v[TEX0].z;
MUL o[TEX0], R5.x, c[A0.x + 18];
```

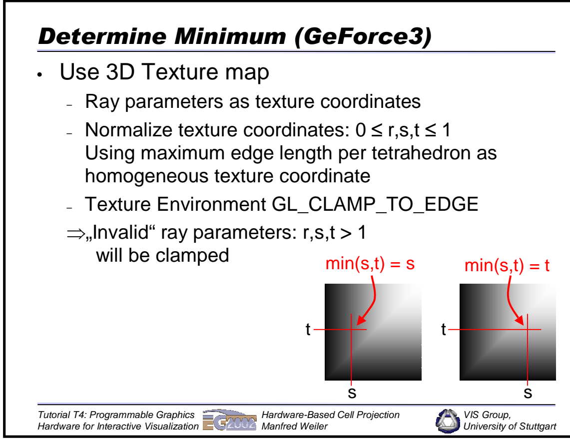
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization | Hardware-Based Cell Projection Manfred Weiler | VIS Group, University of Stuttgart

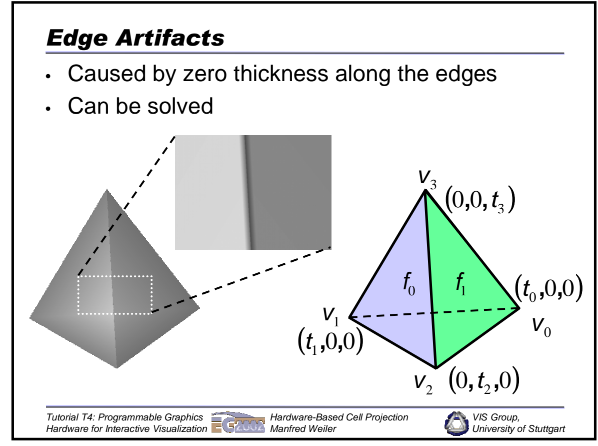

Vertex Attributes
Individual attributes for each vertex:
vx, vy, vz: Vertex position
s: Scalar value for vertex
idx: Index of the vertex within current face
nx, ny, nz: Normal of opposite face
a: Constant term in face plane equation
gx, gy, gz: Scalar field gradient of the tetrahedron
scale: maximum edge length of the tetrahedron
glTexCoord4f | nx | ny | idx | scale
glColor4f | gx | gy | gz | s
glVertex4f | vx | vy | vz | a
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware-Based Cell Projection Manfred Weiler
VIS Group, University of Stuttgart

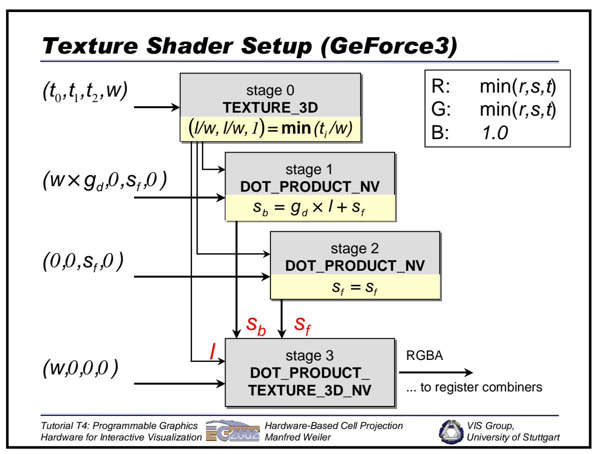

Reconstruct Normal Vector
# Compute Normal (R3)
# ------------------
# n0 = v[TEX0].x
# n1 = v[TEX0].y
# c[11] = {-1, -1, 1, 0}
MUL R4.xy, v[TEX0], v[TEX0];
MOV R4.z, c[11];
DP3 R3.z, c[11], R4;
RSQ R3.z, R3.z;
RCP R3.z, R3.z;
MOV R3.xy, v[TEX0];
Only true for positive n2 !!!
n2 = 1 / √(1 − n0² − n1²)
n0², n1²
− n0² − n1² + 1
√(− n0² − n1² + 1)
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware-Based Cell Projection Manfred Weiler
VIS Group, University of Stuttgart

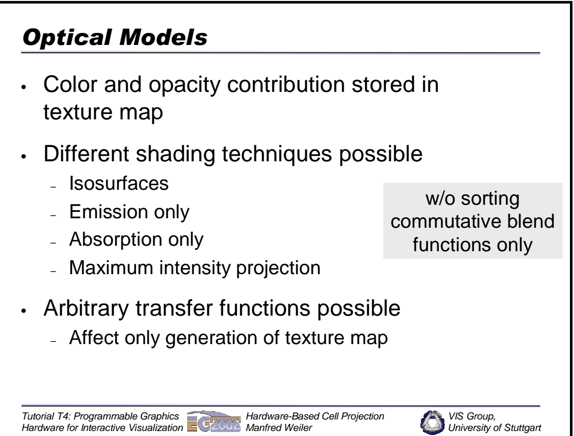

Per-Fragment Operations
Texture Coordinates
r
s
t
min
Texture Coord.
sf, sb, l
Blending
1.0
0.5
0
0.5
1.0
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware-Based Cell Projection Manfred Weiler
VIS Group, University of Stuttgart

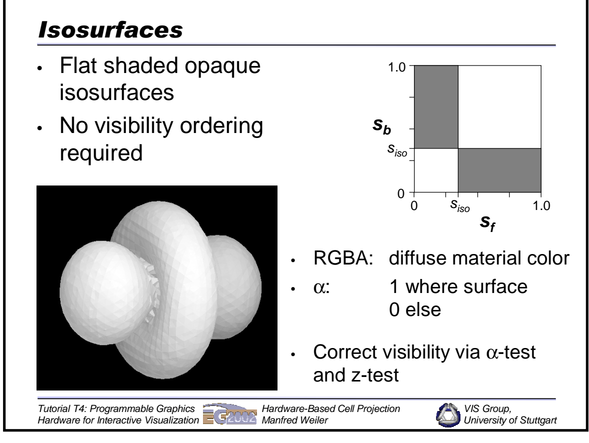

Determine Minimum (Radeon)
Use ATI_fragment_shader
CND0 d, a1, a2, a3:
R(d) = (R(a3) > 0) ? R(a1) : R(a2)
G(d) = (G(a3) > 0) ? G(a1) : G(a2)
B(d) = (B(a3) > 0) ? B(a1) : B(a2)
A(d) = (A(a3) > 0) ? A(a1) : A(a2)
SUB a1, r, s
COND0 a2, s, r, a1
SUB a3, a2, t
COND0 a4, t, a2, a3
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware-Based Cell Projection Manfred Weiler
VIS Group, University of Stuttgart

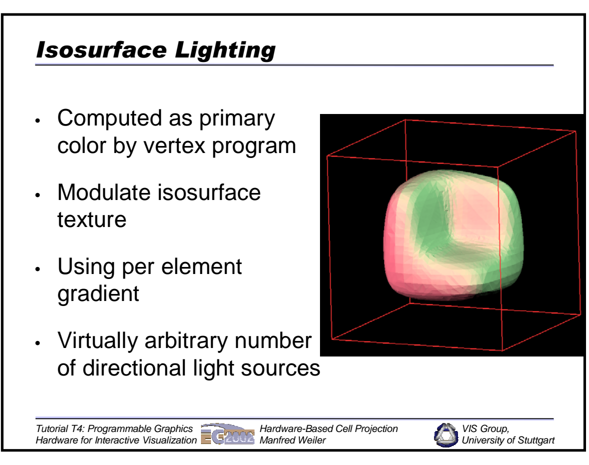

Determine Minimum (GeForce3)
Use 3D Texture map
Ray parameters as texture coordinates
Normalize texture coordinates: 0 ≤ r,s,t ≤ 1
Using maximum edge length per tetrahedron as homogeneous texture coordinate
Texture Environment GL_CLAMP_TO_EDGE
⇒„Invalid“ ray parameters: r,s,t > 1
will be clamped
min(s,t) = s
min(s,t) = t
t
s
t
s
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware-Based Cell Projection Manfred Weiler
VIS Group, University of Stuttgart

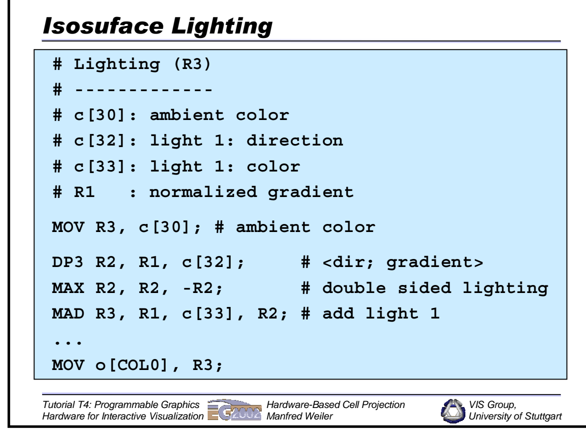

Edge Artifacts
Caused by zero thickness along the edges
Can be solved
V3 (0,0,t3)
f0
f1
(t0,0,0)
V1
V0
(t1,0,0)
V2 (0,t2,0)
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware-Based Cell Projection Manfred Weiler
VIS Group, University of Stuttgart

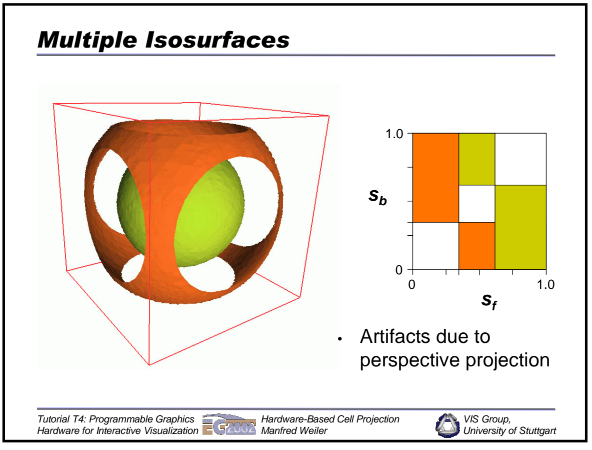

Texture Shader Setup (GeForce3)
(t0,t1,t2,w)
stage 0
TEXTURE_3D
(l/w, l/w, 1) = min (ti/w)
R: min(r,s,t)
G: min(r,s,t)
B: 1.0
(w×gd,0,sf,0)
stage 1
DOT_PRODUCT_NV
sb = gd × l + sf
(0,0,sf,0)
stage 2
DOT_PRODUCT_NV
sf = sf
sb
sf
l
(w,0,0,0)
stage 3
DOT_PRODUCT_
TEXTURE_3D_NV
RGBA
... to register combiners
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware-Based Cell Projection
Manfred Weiler
VIS Group, University of Stuttgart


Optical Models
Color and opacity contribution stored in texture map
Different shading techniques possible
Isosurfaces
Emission only
Absorption only
Maximum intensity projection
w/o sorting
commutative blend functions only
Arbitrary transfer functions possible
Affect only generation of texture map
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware-Based Cell Projection
Manfred Weiler
VIS Group, University of Stuttgart

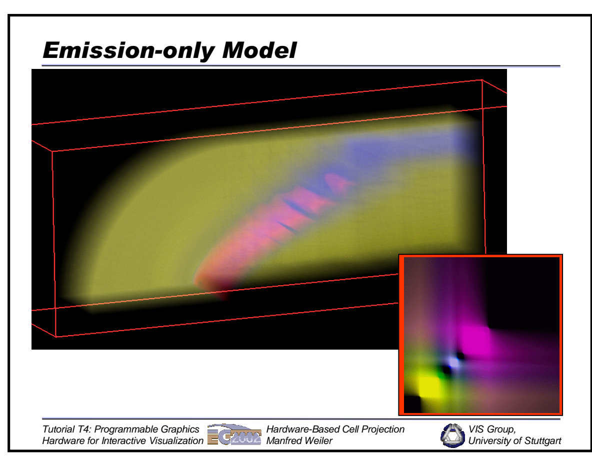

Isosurfaces
Flat shaded opaque isosurfaces
No visibility ordering required
1.0
sb
siso
0
0
siso
sf
1.0
RGBA: diffuse material color
α: 1 where surface
0 else
Correct visibility via α-test and z-test
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware-Based Cell Projection
Manfred Weiler
VIS Group, University of Stuttgart


Isosurface Lighting
Computed as primary color by vertex program
Modulate isosurface texture
Using per element gradient
Virtually arbitrary number of directional light sources
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware-Based Cell Projection
Manfred Weiler
VIS Group, University of Stuttgart

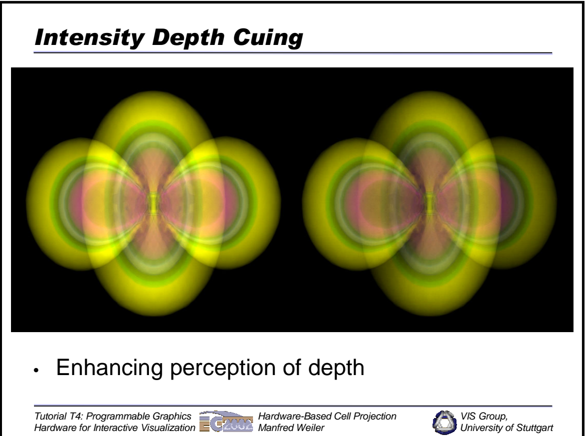

Isosuface Lighting
# Lighting (R3)
# -------------
# c[30]: ambient color
# c[32]: light 1: direction
# c[33]: light 1: color
# R1 : normalized gradient
MOV R3, c[30]; # ambient color
DP3 R2, R1, c[32]; # <dir; gradient>
MAX R2, R2, -R2; # double sided lighting
MAD R3, R1, c[33], R2; # add light 1
...
MOV o[COL0], R3;
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware-Based Cell Projection
Manfred Weiler
VIS Group, University of Stuttgart


Multiple Isosurfaces
1.0
sb
0
0
1.0
sf
Artifacts due to perspective projection
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware-Based Cell Projection
Manfred Weiler
VIS Group, University of Stuttgart

Emission-only Model



Intensity Depth Cuing



- Enhancing perception of depth

Summary

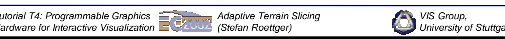
- Render front faces
  - Instead of decomposed projected tetrahedra
  - View-independent
  - Exploit optimizations (display lists, vertex arrays)
- Apply per-fragment ray casting
  - Not yet, due to hardware restrictions
- Implementable solution
  - Using vertex & fragment operations
  - Some hacks required
- Various optical models
  - Texture lookup

Literature

- [Roettger00] S. Röttger, M. Kraus, T. Ertl.
  Hardware-Accelerated Volume And Isosurface Rendering Based On Cell Projection.
  Visualization 2000, Salt Lake City, UT, IEEE.
- [Weiler02] M. Weiler, M. Kraus, T. Ertl.
  Hardware-Based View-Independent Cell Projection.
  to appear in VolVis'02, Boston, MA, IEEE.

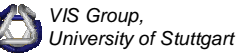
Outline of the Talk

- Previous work by Sim Dietrich (NVIDIA)
- Terrain slicing approach
- Rasterization of horizontally aligned slices
- Rasterization of arbitrary slices
- Adaptive determination of number of slices
- Hybrid approach (slicing / polygonal rendering)
- Results
- Conclusion



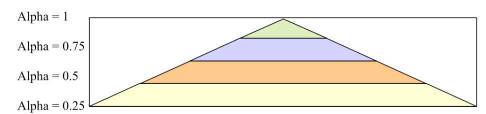
Previous Work

- NVIDIA white paper by Sim Dietrich:
- Store elevation in alpha channel of texture map
- Render set of horizontally aligned slices with alpha test enabled to rasterize the solid part of the terrain

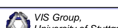


Elevation Maps

- Use a 2D texture that contains the height of each grid point coded into the alpha channel in the range [0,1]
- Draw a set of horizontal slices with an appropriate alpha test enabled
  - glAlphaFunc(GL_GREATER, height of actual slice)



- Artifacts for low view points

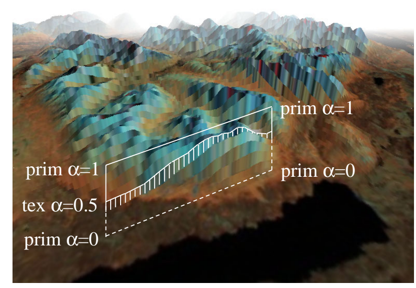


Terrain Slicing

- Solution: Render x-,y-, or z-aligned slices dependent on the view direction
- Assign elevation of the slice vertices as primary alpha
- Use texture combiners to subtract texture alpha from primary alpha
- Enable alpha test to cut off pixels with alpha>0
  -> solid part of each slice is rasterized

Terrain Slicing

- Example terrain rendered with a few slices

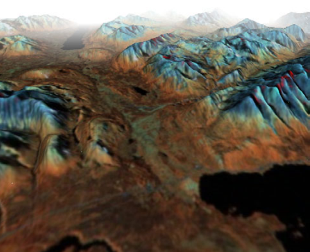


Sub-Pixel Exactness

- Split terrain into several tiles
- Foreach tile:
  - Determine whether x-,y-, or z-slices are needed
  - Estimate the number of parallel slices to achieve sub-pixel exactness
  - Estimation is computed by a screen space projection of the inter-slice distance
  - Rasterize the slices

Sub-Pixel Exactness

- Terrain rendered with a number of slices per tile such that interslice distance maps to less than 1 pixel



Hierarchical Representation

- How big shall the tiles be?
- Too big -> Tiles cannot adapt optimally in a view-dependent fashion (overdraw)
- Too small -> Number of triangles increases
- Solution: Use a hierarchical quadtree representation of the terrain to optimally adapt the number of drawn slices to the viewing distance

Hybrid Approach

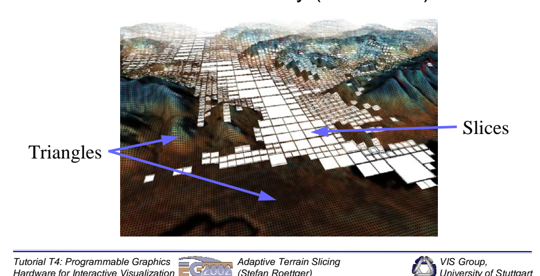
- At each node during the top-down traversal of the octree check whether slicing of the actual node or the four children is faster

  cost = #slices (c1 #pixels + c2 #vertices)
  c1 and c2 are constants specific to the graphics card

- At each leave node check whether terrain slicing or polygon rendering would be faster and choose the appropriate method

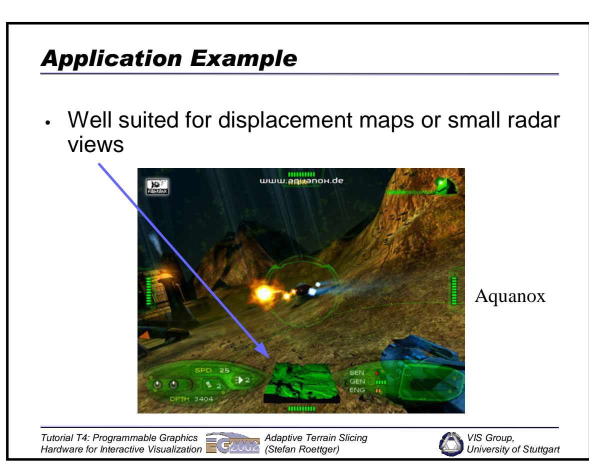
Results

- Approximately 30 fps on a PC with GeForce2 MX (640x480 view port)
- CPU is offloaded efficiently (load<24%)



Conclusion

- Rendering performance depends mainly on screen size (rasterization bound)
- Rendering performance does not depend on the size of the height field
- Algorithmic complexity is low in comparison to C-LOD approaches


Application Example
• Well suited for displacement maps or small radar views
www.aquanox.de
Aquanox
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Adaptive Terrain Slicing (Stefan Roettger)
VIS Group, University of Stuttgart

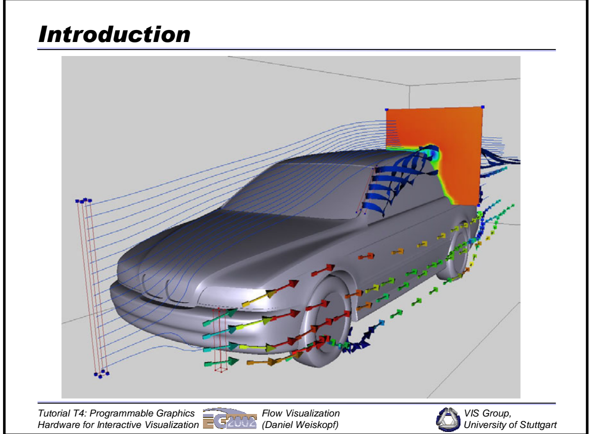
Visualization of 2D Flow Fields by Texture Advection

Daniel Weiskopf
VIS Group, University of Stuttgart

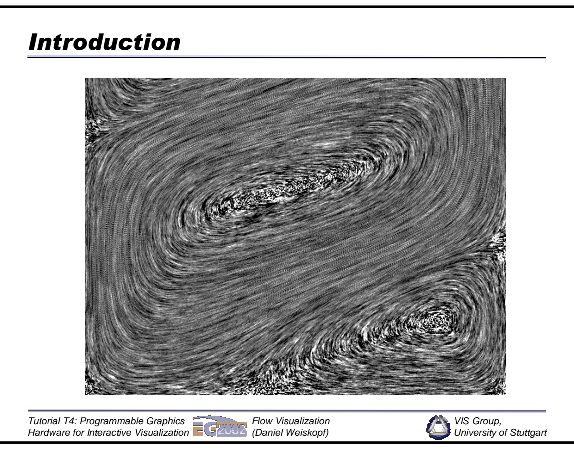
Introduction

- Vector field visualization: indirectly by tracing particles
- Issues:
  - Seed point positioning
  - Dense vs. sparse representation
  - Steady vs. unsteady flow fields
  - Visualization speed

Introduction



Introduction



Related Work

- Mapping/rendering techniques:
  - Streamlines, streaklines, ribbons, etc.
  - Spot noise [Wijk 91]
  - Texture Splats [Crawfis & Max 93]
  - LIC [Cabral & Leedom 93, Stalling & Hege 95], etc.
  - Texture advection [Max et al. 92/96]
  - Lagrangian-Eulerian Advection [Jobard et al. 01]

Related Work

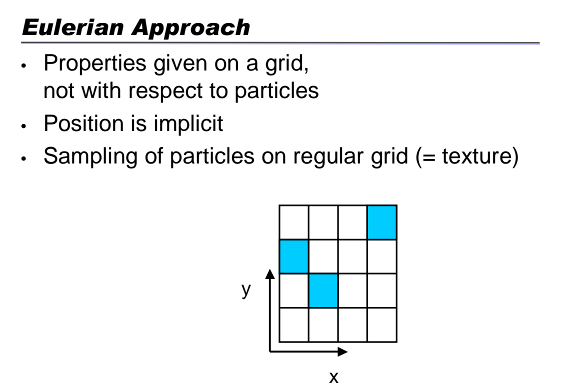
- Hardware-based:
  - LIC [Heidrich et al. 99]
  - Texture advection [Jobard et al. 00]
- Seed point positioning:
  - Topology-guided [Helman & Hesselink 89]
  - Equally spaced streamlines [Turk & Banks 96]
  - Flow-guided streamline seeding [Verma et al. 00]

Lagrangian Approach

- Velocity vector **v**
- Trace massless particles
- Particles identified individually
- Attached are position **r**, velocity **v**, and other properties (color etc.)
- Equation of motion: $$\frac{d\mathbf{r}}{dt} = \mathbf{v}(\mathbf{r}, t)$$
- First order integration: $$\Delta\mathbf{r} = \Delta t \cdot \mathbf{v}(\mathbf{r}, t)$$

Eulerian Approach

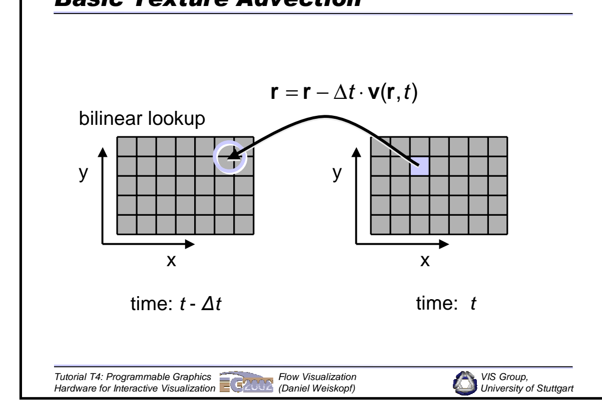
- Properties given on a grid, not with respect to particles
- Position is implicit
- Sampling of particles on regular grid (= texture)



Lagrangian-Eulerian Approach

- Combination of Eulerian and Lagrangian approaches
- Lagrangian-Eulerian Advection (LEA)
- Split in two parts:
  - Lagrangian: Update of coordinates of particles (*coordinate integration*)
  - Eulerian: Particle properties attached to grid (*property advection*)

Basic Texture Advection

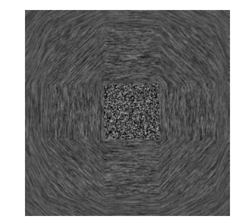


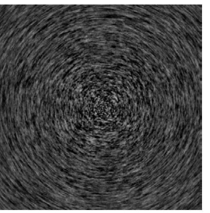
Basic Texture Advection

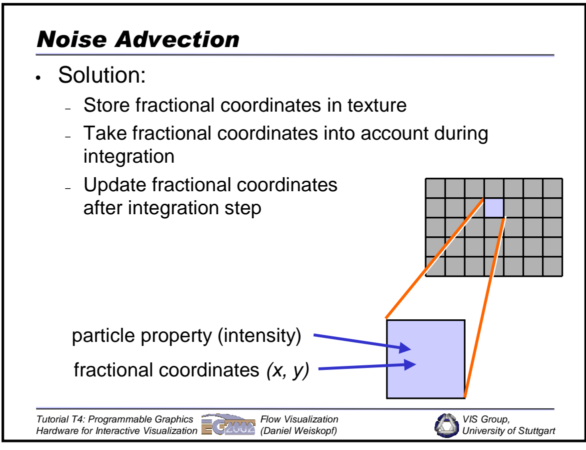
- Forward mapping could produce gaps
- Backward mapping avoids gaps
- Bilinear interpolation:
  - Artificial diffusion
  - Caused by "averaging" via interpolation
  - O.K. for dye advection
  - Unsuitable for noise advection

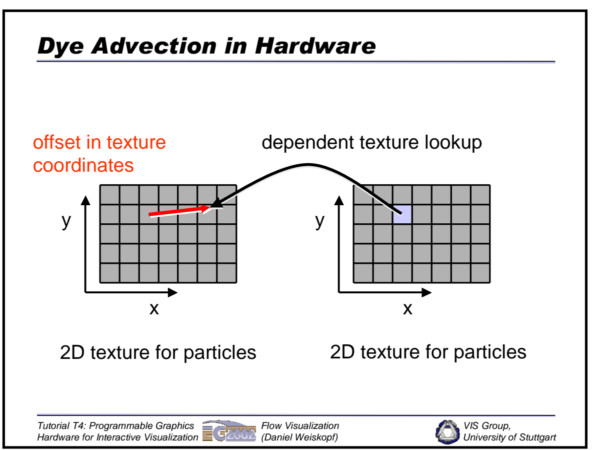
Noise Advection

- Replace bilinear interpolation by nearest neighbor lookup
- Problem: Particles may stay in respective cell for small velocity or step size



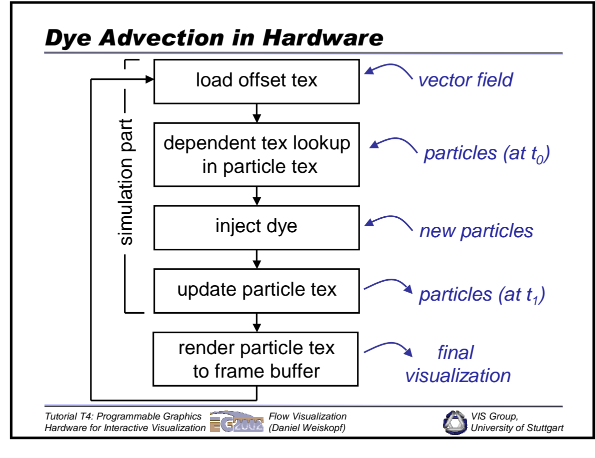






Dye Advection in Hardware

- Representation:
  - Particles by standard 2D RGB texture
  - Vector field by 2D signed texture, with 2 components
  - Computational domain by rendering a rectangle ⇒ generates fragments

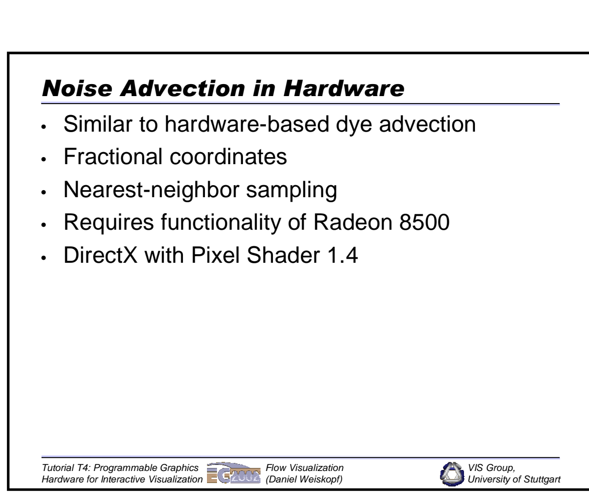


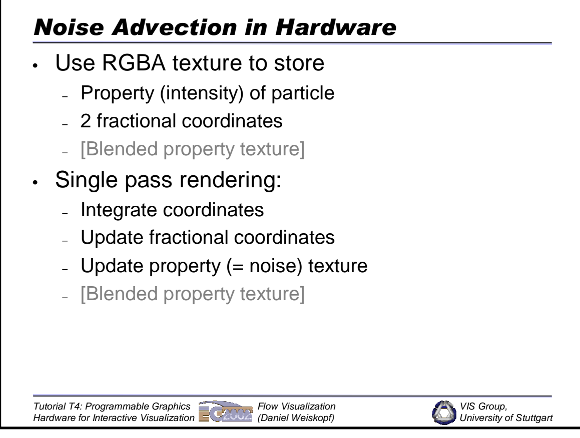
Dye Advection in Hardware

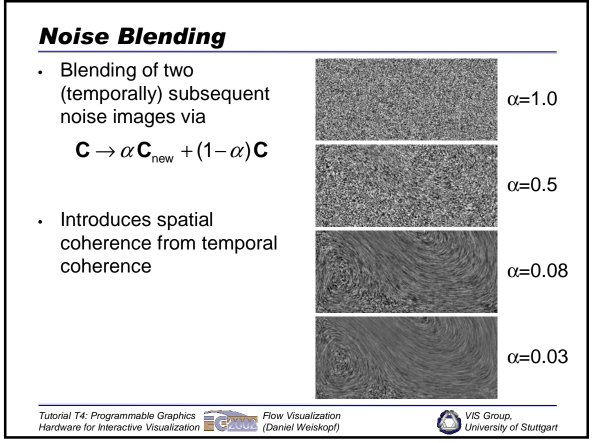
- Pixel Shader 1.0 code:

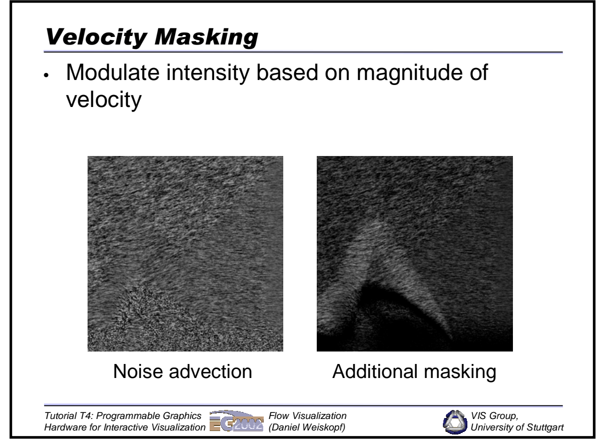
```
tex t0
texbem t1,t0
mov r0, t1
```

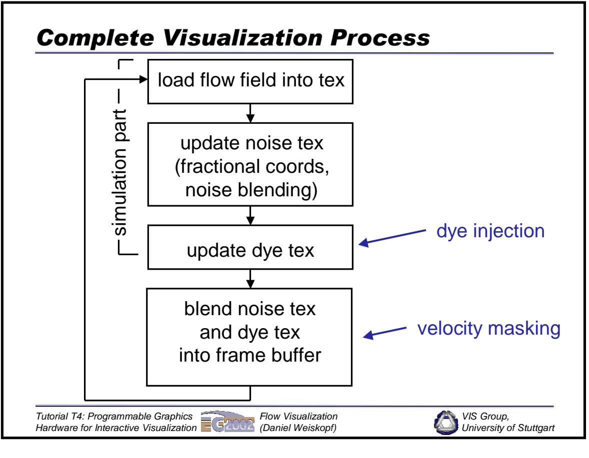
- Runs on GeForce 3/4 or Radeon 8500

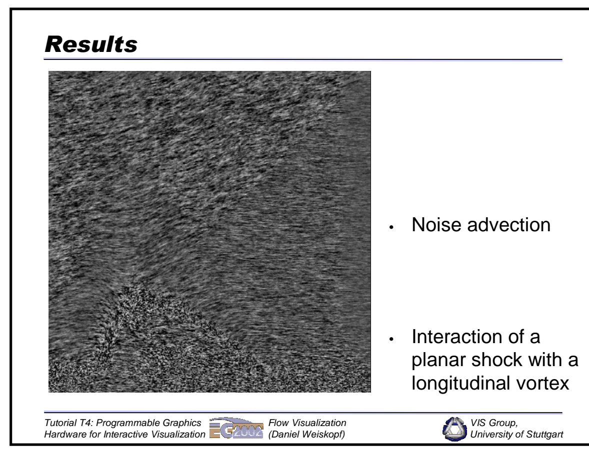


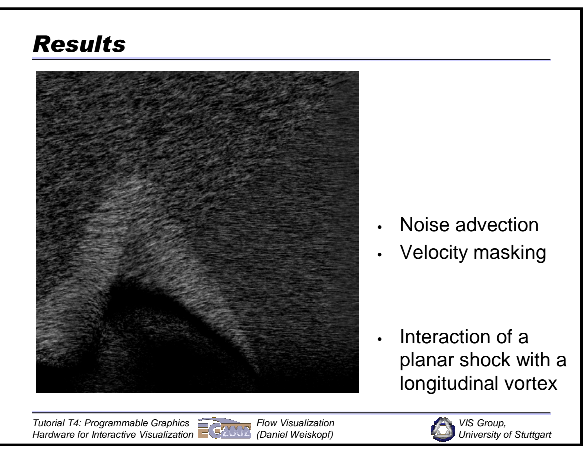

Noise Advection in Hardware
Use RGBA texture to store
Property (intensity) of particle
2 fractional coordinates
[Blended property texture]
Single pass rendering:
Integrate coordinates
Update fractional coordinates
Update property (= noise) texture
[Blended property texture]
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Flow Visualization (Daniel Weiskopf)
VIS Group, University of Stuttgart


Noise Blending
Blending of two (temporally) subsequent noise images via
$\mathbf{C} \rightarrow \alpha \mathbf{C}_{new} + (1-\alpha)\mathbf{C}$
Introduces spatial coherence from temporal coherence
α=1.0
α=0.5
α=0.08
α=0.03
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Flow Visualization (Daniel Weiskopf)
VIS Group, University of Stuttgart

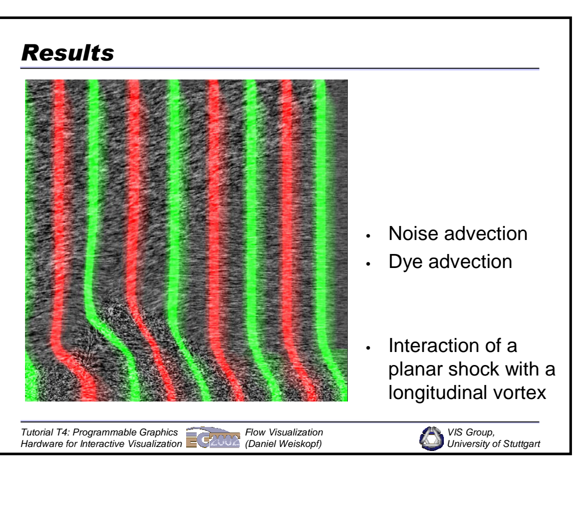

Velocity Masking
Modulate intensity based on magnitude of velocity
Noise advection
Additional masking
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Flow Visualization (Daniel Weiskopf)
VIS Group, University of Stuttgart


Complete Visualization Process
simulation part
load flow field into tex
update noise tex (fractional coords, noise blending)
update dye tex
dye injection
blend noise tex and dye tex into frame buffer
velocity masking
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Flow Visualization (Daniel Weiskopf)
VIS Group, University of Stuttgart


Results
Noise advection
Interaction of a planar shock with a longitudinal vortex
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Flow Visualization (Daniel Weiskopf)
VIS Group, University of Stuttgart


Results
Noise advection
Velocity masking
Interaction of a planar shock with a longitudinal vortex
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Flow Visualization (Daniel Weiskopf)
VIS Group, University of Stuttgart

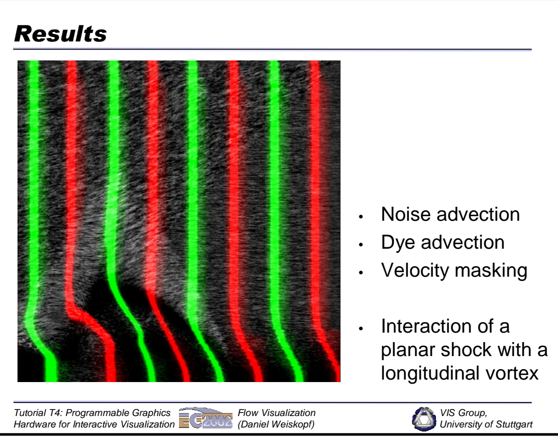
Results



- Noise advection
- Dye advection

- Interaction of a planar shock with a longitudinal vortex

Results



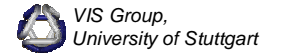
- Noise advection
- Dye advection
- Velocity masking

- Interaction of a planar shock with a longitudinal vortex

Tutorial T4: Programmable Graphics Hardware for Interactive Visualization — Flow Visualization (Daniel Weiskopf) — VIS Group, University of Stuttgart

Further Reading

- Dye advection on GeForce 3 [Weiskopf et al. 01]
- LEA on Radeon 8500 [Weiskopf et al. 02]


Tutorial T4: Programmable Graphics Hardware for Interactive Visualization — Flow Visualization (Daniel Weiskopf) — VIS Group, University of Stuttgart

References

[Cabral & Leedom 93] B. Cabral, L. C. Leedom. Imaging vector fields using line integral convolution. In *SIGGRAPH 1993 Conference Proceedings*, pages 263-272.

[Crawfis & Max 93] R. Crawfis, N. Max. Texture splats for 3D scalar and vector field visualization. In *IEEE Visualization '93*, pages 261-267.

[Heidrich et al. 99] W. Heidrich, R. Westermann, H.-P. Seidel, T. Ertl. Application of pixel textures in visualization and realistic image synthesis. In *ACM Symposium on Interactive 3D Graphics*, pages 127-134,1999.

[Helman & Hesselink 89] J. Helman, L. Hesselink. Representation and display of vector field topology in fluid flow data sets. *IEEE Computer*, 22(8), pages 27-36, August 1989.

[Jobard et al. 00] B. Jobard, G. Erlebacher, M. Y. Hussaini. Hardware-accelerated texture advection for unsteady flow visualization. In *IEEE Visualization 2000*, pages 155-162.

[Jobard et al. 01] B. Jobard, G. Erlebacher, M. Y. Hussaini. Lagrangian-Eulerian advection for unsteady flow visualization. In *IEEE Visualization 2001*, 53-60.

Tutorial T4: Programmable Graphics Hardware for Interactive Visualization — Flow Visualization (Daniel Weiskopf) — VIS Group, University of Stuttgart

References

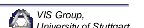
[Max et al. 92] N. Max, R. Crawfis, D. Williams. Visualizing wind velocities by advecting cloud textures. In *IEEE Visualization '92*, pages 171-178.

[Max & Becker 96] N. Max, B. Becker. Flow visualization using moving textures. In *Proc. ICASE/LaRC Symposium on Visualizing Time Varying Data*, D. C. Banks, T. W. Crockett, S. Kathy (eds.), pages 77-87, 1996.

[Stalling & Hege 95] D. Stalling, H.-C. Hege. Fast and resolution independent line integral convolution. In *SIGGRAPH 1995 Conference Proceedings*, pages 249-256.

[Turk & Banks 96] G. Turk. David Banks. Image-guided streamline placement. In *SIGGRAPH 1996 Conference Proceedings*, pages 453-460.

[Verma et al. 00] V. Verma, D. Kao, A. Pang. A flow-guided streamline seeding strategy. In *IEEE Visualization 2000*, pages 163-170.


Tutorial T4: Programmable Graphics Hardware for Interactive Visualization — Flow Visualization (Daniel Weiskopf) — VIS Group, University of Stuttgart

References

[Weiskopf et al. 01] D. Weiskopf, M. Hopf, T. Ertl. Hardware-accelerated visualization of time-varying 2D and 3D vector fields by texture advection via programmable per-pixel operations. In *VMV '01 Proceedings*, pages 439-446, 2001.

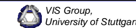
[Weiskopf et al. 02] D. Weiskopf, G. Erlebacher, M. Hopf, T. Ertl. Hardware-accelerated Lagrangian-Eulerian texture advection for 2D flow visualization. Submitted to *VMV '02*.

[Wijk 91] J. van Wijk. Spot noise-texture synthesis for data visualization. In *SIGGRPAPH 1991 Conference Proceedings*, pages 309-318.

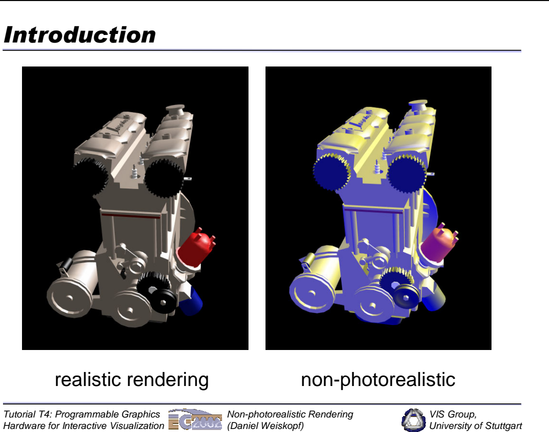
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization — Flow Visualization (Daniel Weiskopf) — VIS Group, University of Stuttgart

Interactive Non-Photorealistic Rendering of Technical Illustrations

Daniel Weiskopf
VIS Group, University of Stuttgart



Introduction



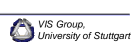
realistic rendering | non-photorealistic

Introduction

- Photorealistic rendering
  - Resemble the output of a photographic camera
- Non-photorealistic rendering (NPR)
  - Convey meaning and shape
  - Emphasize important parts
  - Mimic artistic rendering

Introduction

- Typical drawing styles for NPR
  - Pen-and-ink illustration
  - Stipple rendering
  - Tone shading
  - Cartoon rendering
- Further reading on NPR:
  - SIGGRAPH 1999 Course #17: *Non-Photorealistic Rendering*
  - Gooch & Gooch: *Non-Photorealistic Rendering* [2001]
  - Strothotte & Schlechtweg: *Non-Photorealistic Computer Graphics* [2002]

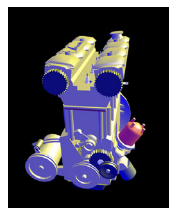


Introduction

- Focus of this talk
  - Technical illustrations
  - Tone shading: cool / warm shading
  - Semi-transparent rendering

Tone Shading

- Cool / warm shading [Gooch et al. 1998]
- Tone shading for technical illustrations
- Lighting model:
  - Uses luminance and hue
  - Indicates surface orientation

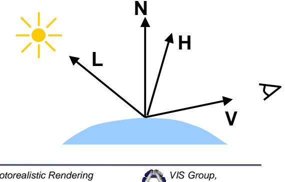


Tone Shading

- Standard Blinn-Phong model

$$I = \underbrace{I_l k_a}_{\text{ambient}} + \underbrace{I_l k_d \max(0, \mathbf{L}\cdot\mathbf{N})}_{\text{diffuse}} + \underbrace{I_l k_s (\max(0, \mathbf{H}\cdot\mathbf{N}))^n}_{\text{specular}}$$

  - Positive term max(0, **L**·**N**)
  - **L** light vector
  - **N** normal vector
  - **V** viewing vector
  - **H** halfway vector

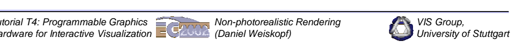


Tone Shading

- Cool / warm shading of a matte object

$$I = \left(\frac{1+\mathbf{L}\cdot\mathbf{N}}{2}\right)k_{\text{warm}} + \left(1-\frac{1+\mathbf{L}\cdot\mathbf{N}}{2}\right)k_{\text{cool}}$$

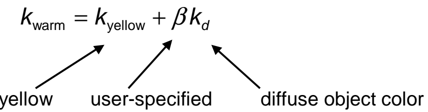
  - Dot product **L**·**N** between normal vector and light vector
  - **L**·**N** can be negative: **L**·**N** $\in [-1,1]$
- Specular highlights like in Blinn-Phong model



Tone Shading

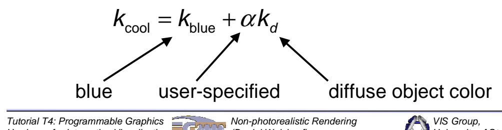
- Mixture of yellow and the object's color k_d :

$$k_{\text{warm}} = k_{\text{yellow}} + \beta k_d$$



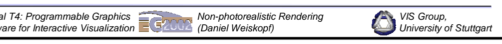
- Mixture of blue and the object's color k_d :

$$k_{\text{cool}} = k_{\text{blue}} + \alpha k_d$$



Implementation on Standard Hardware

- Gouraud shading
- Approximation via two light sources:
  - Light along direction **L** with intensity ($k_{\text{warm}} - k_{\text{cool}}$)/2
  - Light along direction -**L** with intensity ($k_{\text{cool}} - k_{\text{warm}}$)/2
  - Ambient term with intensity ($k_{\text{cool}} + k_{\text{warm}}$)/2
  - Object color has to be white
- Problems:
  - No specular highlights (only in two-pass rendering)
  - Change of material parameters needs change of light color
  - Two light sources needed
  - Negative intensities for the light sources

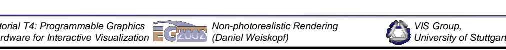


Cool / Warm Shading via Vertex Programs

- Gouraud shading
- Implements cool / warm shading on per-vertex basis
- Includes specular highlights
- Computation in flexible vertex programs

Cool / Warm Shading via Vertex Programs

- Only fragments of the vertex program code
- Already done:
  - Transformation of vertex into eye and clip coordinates
  - Transformation of normal vector into eye coordinates
  - Computation of halfway vector
- Input:
  - Diffuse object color as primary color
  - Parameters for cool / warm model
  - Parameters for specular highlights
  - Color and direction of light



Cool / Warm Shading via Vertex Programs

- First part: computation of geometric components for lighting

```
...
# c[0]   = L: direction of light
# c[1].x = m: shininess for highlight
# R0     = N: normal vector in eye coords
# R1     = H: halfway vector in eye coords
DP3 R2.x,c[0],R0; # L*N
DP3 R2.y,R1,R0;   # H*N
MOV R2.z,c[1].x;  # shininess for highlight
LIT R3,R2;        # R3=weights for Blinn-Phong
...               # model
```


Cool / Warm Shading via Vertex Programs

- Parameters for cool / warm shading

```
# v[COL0] = k_d : diffuse material color
# c[2] = (alpha,beta,0,0): cool/warm params
# c[3] = k_blue: blue tone
MOV R4,c[2];
MAD R5,v[COL0],R4.x,c[3]; #(k_d*alpha + k_blue)
                          # = k_cool
```

- Analogous:

```
# R6 = (k_d*beta + k_yellow) = k_warm
```


Cool / Warm Shading via Vertex Programs

- Geometric coefficients for cool / warm shading

```
# c[4] = (1,0.5,0,0)
# R2.x = L*N
ADD R7,R2.x,c[4].x;  # L*N + 1
MUL R8,R7,c[4].y;    # (L*N + 1)/2
ADD R9,-R8,c[4].x;   # 1 - (L*N + 1)/2
```

- Cool / warm shading

```
# R5 = k_cool
# R6 = k_warm
MUL R10,R5,R9;       # warm part
MAD R10,R6,R8,R10;   # add cool part
```

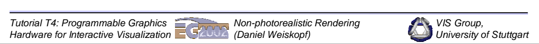

Cool / Warm Shading via Vertex Programs

- Finally add specular part

```
# R3.z = (N*H)^n : specular weight from LIT
# c[5] = light color * specular material color
MAD o[COL0],R3.z,c[5],R10;  # add specular part
```

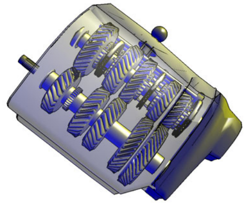

Cool / Warm Shading via Vertex Programs

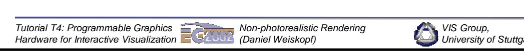
- Advantages:
  - Transparent to the user / developer
  - Cool / warm shading and highlights in single-pass rendering
  - Material properties can be specified per vertex
- Summary:
  - Vertex programs good for per-vertex lighting
  - Gouraud shading only
  - No Phong shading
  - Many more possible applications:
    Toon shading, variations of cool / warm shading, …



Semi-Transparent Illustrations

- Goals of semi-transparent renderings:
  - Reveal information of otherwise occluded interior objects
  - Show spatial relationship between these objects
- Special rendering model for semi-transparent illustrations
  [Diepstraten et al. 2002]



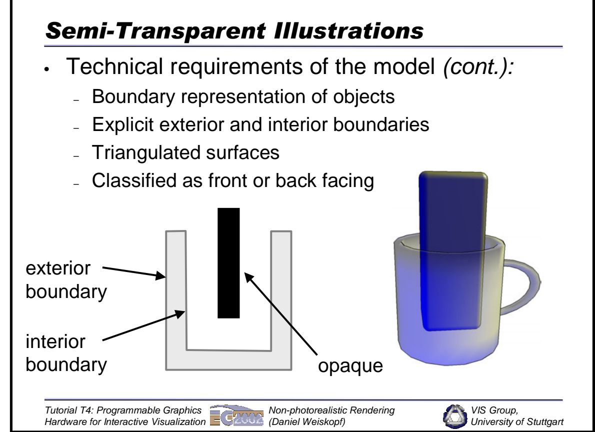


Semi-Transparent Illustrations

- Technical requirements of the model:
  - Only those opaque objects located between the closest and second-closed front-facing transparent surfaces are visible
  - Objects further behind are hidden
  - View-dependent spatial sorting
  - Only partial sorting
  - Need to determine only the closest and second-closest surfaces

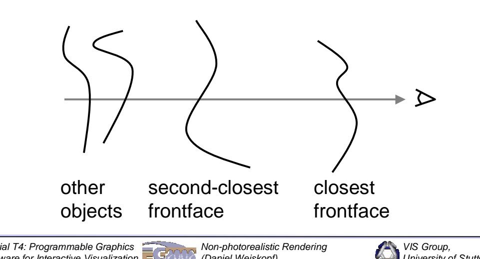
Semi-Transparent Illustrations

- Technical requirements of the model *(cont.):*
  - Boundary representation of objects
  - Explicit exterior and interior boundaries
  - Triangulated surfaces
  - Classified as front or back facing



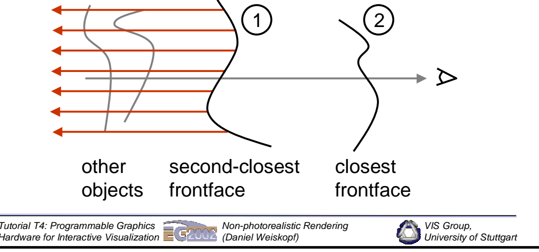
Partial Depth Sorting

- Image-space approach:
  - Requires depth values for closest and second-closest frontface
  - Would require two depth buffers



Partial Depth Sorting

- Image-space approach *(cont.):*
  - Hide objects behind second-closest frontface
  - Render second-closest frontface before closest frontface to achieve correct blending



Partial Depth Sorting

- Basic algorithm:
  1. Render closest frontface (based on standard *z* test)
  2. Store current *z* values
  3. Exclude the closest frontface in following step, based on the *z* values from step 2
  4. Render remaining frontfaces based on *z* test: yields second-closest frontface
  5. Blend closest frontface in front of second-closest front face

Depth Replace Fragment Operations

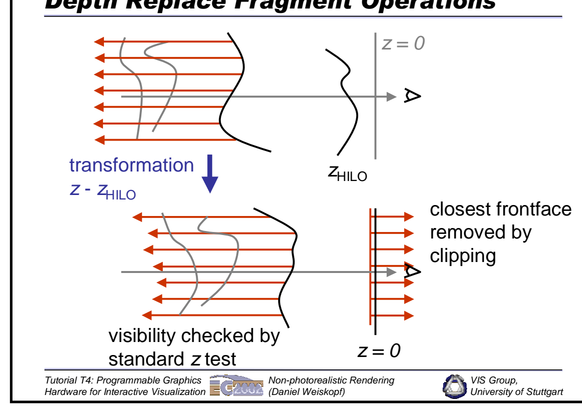
- Problem:
  - No two depth buffers
- Solution:
  - One depth buffer
  - An additional hires texture (HILO texture)
  - Depth replace fragment operations on GeForce 3/4

Depth Replace Fragment Operations

- Basic idea:
  - Store z values of closest frontfaces in HILO texture
  - `Dot Product Depth Replace` changes z value of following fragments to $z - z_{HILO}$
  - Closest frontface then has $z = 0$
    $\Rightarrow$ is removed by clipping against view frustum
  - Second-closest frontface passes z test and clipping test
  - All other faces are rejected by z test

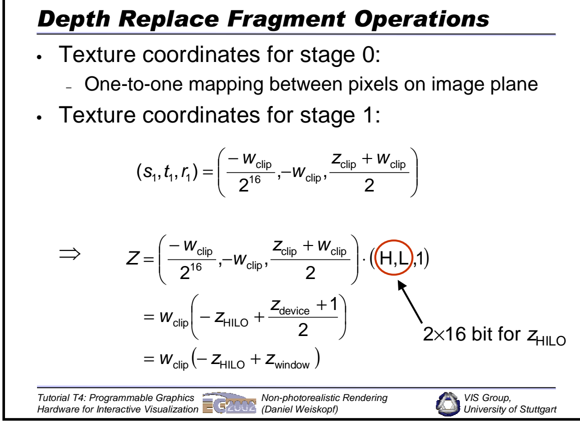
$\Rightarrow$ second-closest frontface is extracted

Depth Replace Fragment Operations



Depth Replace Fragment Operations

- `Dot Product Depth Replace`



Depth Replace Fragment Operations

- Texture coordinates for stage 0:
  - One-to-one mapping between pixels on image plane
- Texture coordinates for stage 1:

$$(s_1, t_1, r_1) = \left(\frac{-w_{clip}}{2^{16}}, -w_{clip}, \frac{z_{clip} + w_{clip}}{2}\right)$$

$$\Rightarrow \quad Z = \left(\frac{-w_{clip}}{2^{16}}, -w_{clip}, \frac{z_{clip} + w_{clip}}{2}\right) \cdot (H, L, 1)$$

$$= w_{clip}\left(-z_{HILO} + \frac{z_{device} + 1}{2}\right)$$

$$= w_{clip}\left(-z_{HILO} + z_{window}\right)$$

2×16 bit for z_{HILO}

Depth Replace Fragment Operations

- Texture coordinates for stage 2:

$$(s_2, t_2, r_2) = (0, 0, w_{clip})$$

$$\Rightarrow \quad W = (0, 0, w_{clip}) \cdot (H, L, 1) = w_{clip}$$

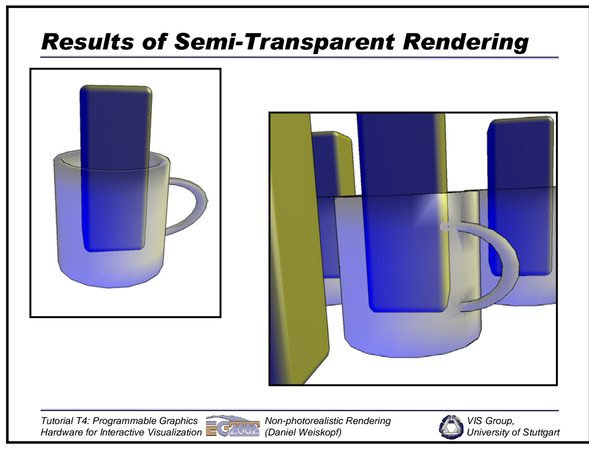
- Final depth value

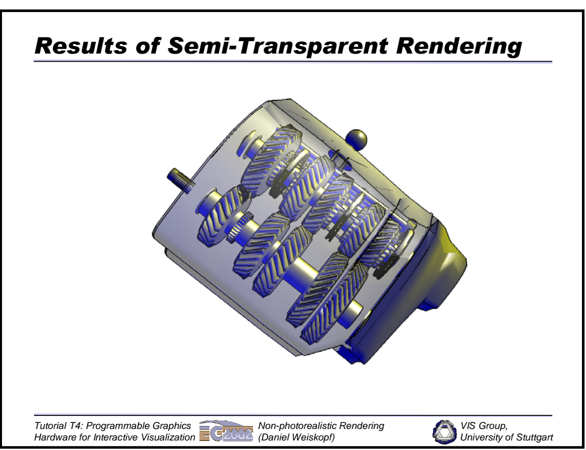
$$z = \frac{Z}{W} = \frac{w_{clip}\left(-z_{HILO} + z_{window}\right)}{w_{clip}}$$

$$= z_{window} - z_{HILO}$$

Depth Replace Fragment Operations

- Algorithm related to *depth peeling* [Everitt 2001]
- Both can be used for complete spatial sorting as well:
  - Multi-pass rendering for each layer, i.e., peeling off different depth layers
  - Order-independent transparency





References

[Diepstraten et al. 2002] J. Diepstraten, D. Weiskopf, T. Ertl. Transparency in interactive technical illustrations. In *Eurographics 2002 Proceedings.*

[Everitt 2001] C. Everitt. Interactive order-independent transparency. White paper, *NVidia*, 2001.

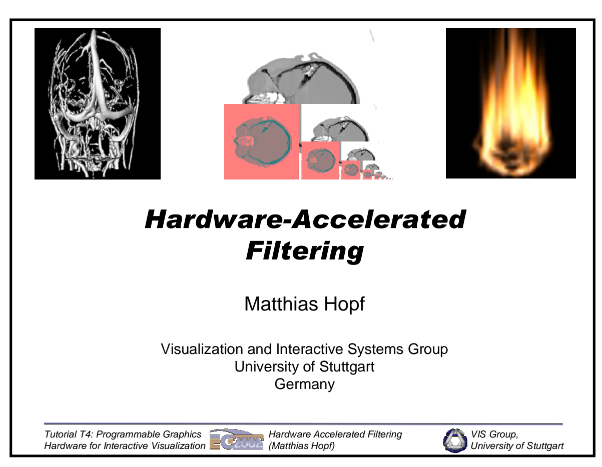
[Gooch et al. 1998] A. Gooch, B. Gooch, P. Shirley, E. Cohen. A non-photorealistic lighting model for interactive technical illustration. In *SIGGRAPH 1998 Conference Proceedings*, pages 101-108.

[Gooch & Gooch 2001]: B. Gooch, A. Gooch. *Non-Photorealistic Rendering*. A. K. Peters, Natick, 2001.

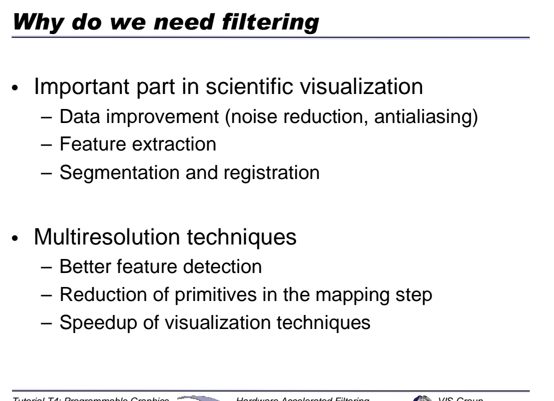
[SIGGRAPH 1999 Course 17] S. Green, D. Salesin, S. Schofield, A. Hertzmann, P. Litwinowicz, A. Gooch, C. Curtis, B. Gooch. *SIGGRAPH 1999 Course 17: Non-Photorealistic Rendering.*

[Strothotte & Schlechtweg 2002] T. Strothotte, S. Schlechtweg. *Non-Photorealistic Computer Graphics.* Morgan Kaufmann Publishers, 2002.

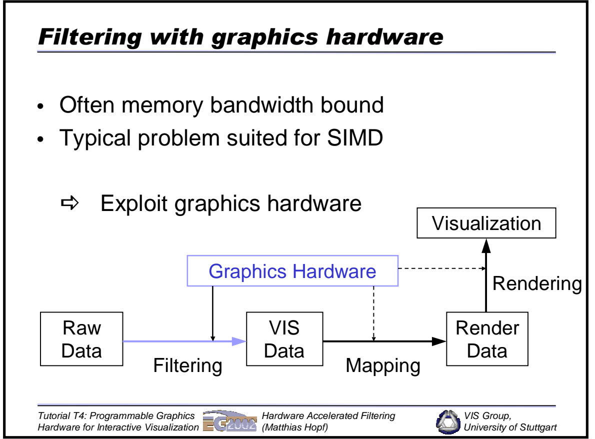
Non-photorealistic Rendering (Daniel Weiskopf)   VIS Group, University of Stuttgart



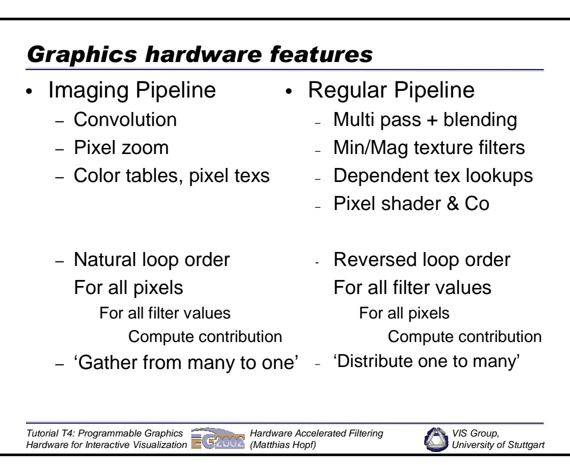
Hardware-Accelerated Filtering
Matthias Hopf
Visualization and Interactive Systems Group
University of Stuttgart
Germany
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware Accelerated Filtering (Matthias Hopf)
VIS Group, University of Stuttgart



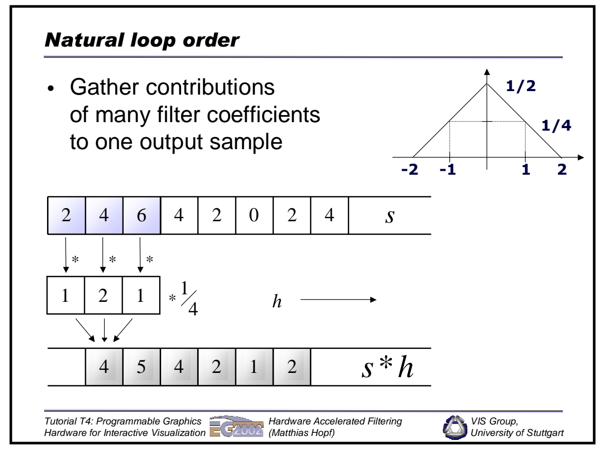
Why do we need filtering
Important part in scientific visualization
Data improvement (noise reduction, antialiasing)
Feature extraction
Segmentation and registration
Multiresolution techniques
Better feature detection
Reduction of primitives in the mapping step
Speedup of visualization techniques



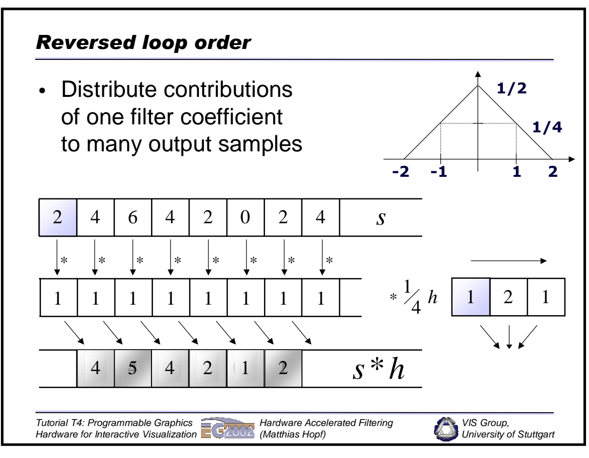
Filtering with graphics hardware
Often memory bandwidth bound
Typical problem suited for SIMD
Exploit graphics hardware
Visualization
Graphics Hardware
Rendering
Raw Data
VIS Data
Render Data
Filtering
Mapping



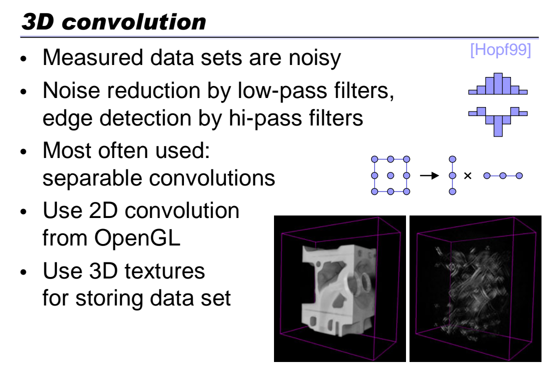
Graphics hardware features
Imaging Pipeline
Convolution
Pixel zoom
Color tables, pixel texs
Regular Pipeline
Multi pass + blending
Min/Mag texture filters
Dependent tex lookups
Pixel shader & Co
Natural loop order
For all pixels
For all filter values
Compute contribution
'Gather from many to one'
Reversed loop order
For all filter values
For all pixels
Compute contribution
'Distribute one to many'

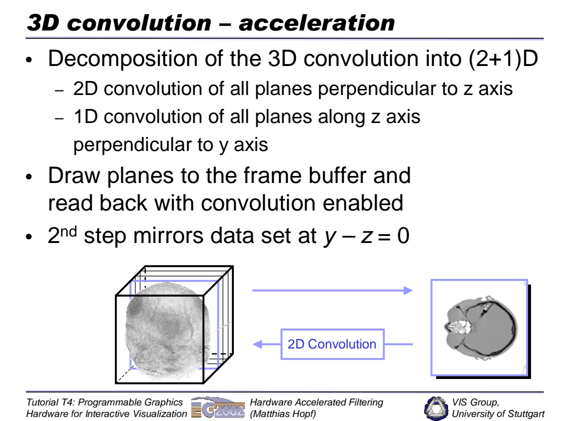


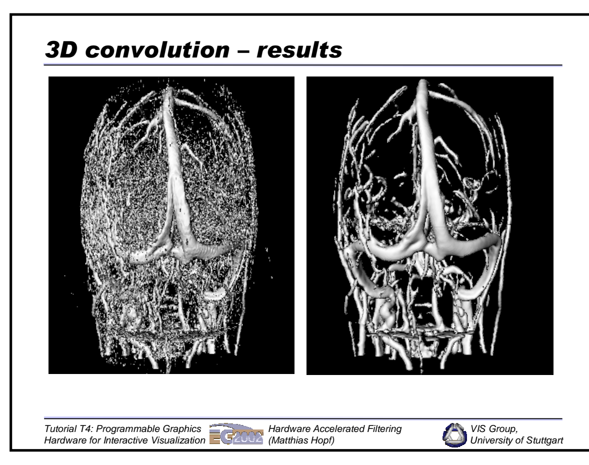
Natural loop order
Gather contributions of many filter coefficients to one output sample
1/2
1/4
-2 -1 1 2
2 4 6 4 2 0 2 4 s
1 2 1 * 1/4
h
4 5 4 2 1 2 s * h

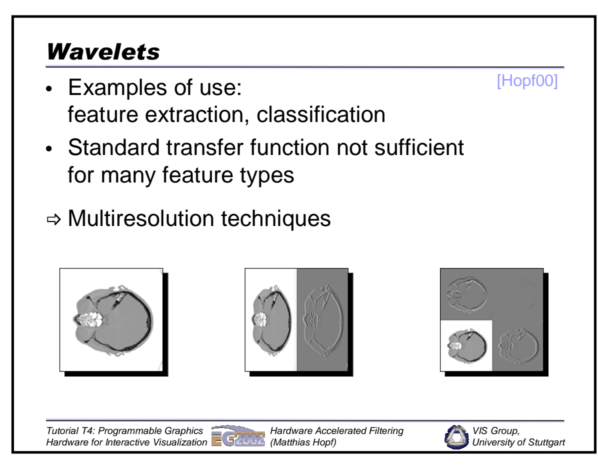


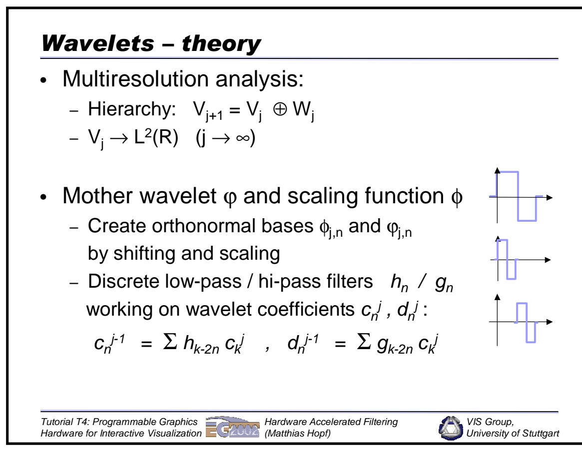
Reversed loop order
Distribute contributions of one filter coefficient to many output samples
1/2
1/4
-2 -1 1 2
2 4 6 4 2 0 2 4 s
1 1 1 1 1 1 1 1
* 1/4 h
1 2 1
4 5 4 2 1 2 s * h

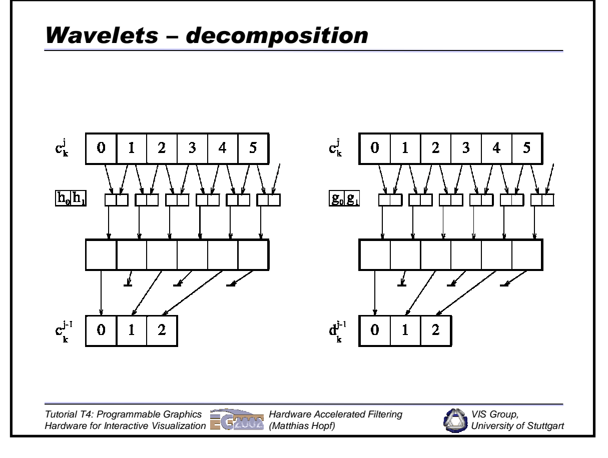

3D convolution
[Hopf99]
Measured data sets are noisy
Noise reduction by low-pass filters, edge detection by hi-pass filters
Most often used: separable convolutions
Use 2D convolution from OpenGL
Use 3D textures for storing data set
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware Accelerated Filtering (Matthias Hopf)
VIS Group, University of Stuttgart

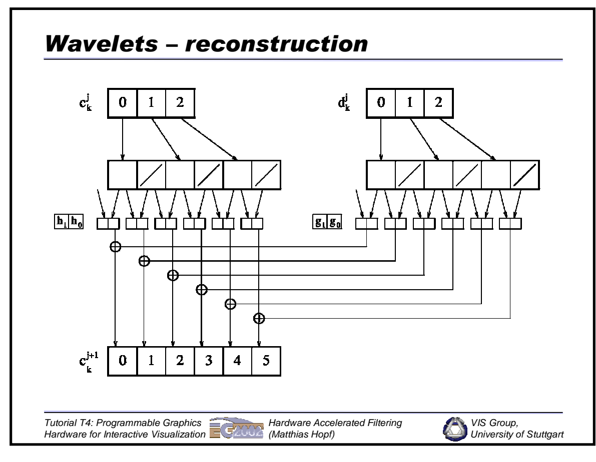

3D convolution – acceleration
Decomposition of the 3D convolution into (2+1)D
2D convolution of all planes perpendicular to z axis
1D convolution of all planes along z axis perpendicular to y axis
Draw planes to the frame buffer and read back with convolution enabled
2nd step mirrors data set at $y - z = 0$
2D Convolution
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware Accelerated Filtering (Matthias Hopf)
VIS Group, University of Stuttgart

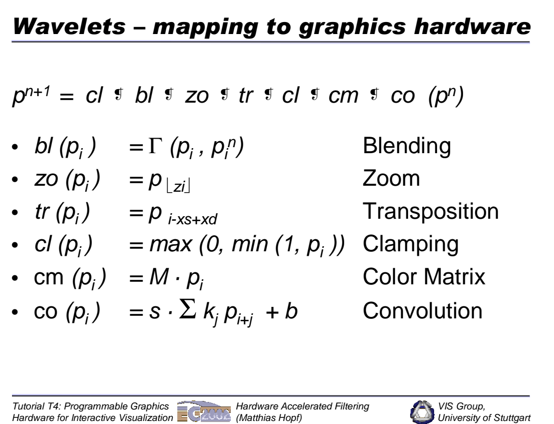

3D convolution – results
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware Accelerated Filtering (Matthias Hopf)
VIS Group, University of Stuttgart

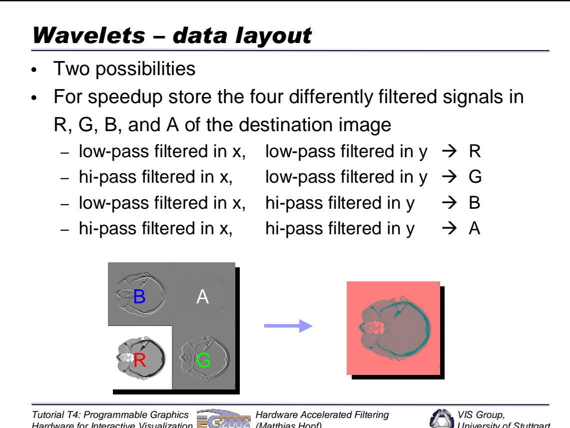

Wavelets
[Hopf00]
Examples of use: feature extraction, classification
Standard transfer function not sufficient for many feature types
⇨ Multiresolution techniques
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware Accelerated Filtering (Matthias Hopf)
VIS Group, University of Stuttgart

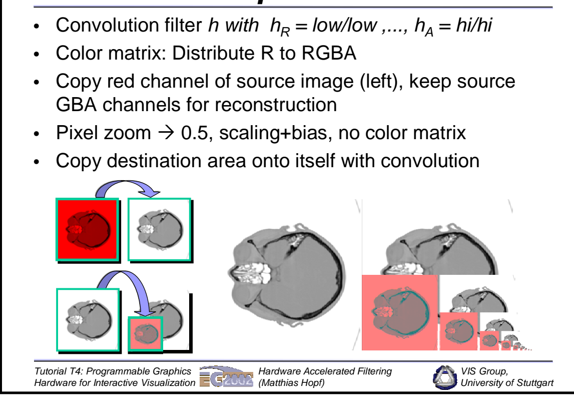

Wavelets – theory
Multiresolution analysis:
Hierarchy: $V_{j+1} = V_j \oplus W_j$
$V_j \to L^2(R) \quad (j \to \infty)$
Mother wavelet φ and scaling function ϕ
Create orthonormal bases $\phi_{j,n}$ and $\varphi_{j,n}$ by shifting and scaling
Discrete low-pass / hi-pass filters h_n / g_n working on wavelet coefficients c_n^j, d_n^j:
$c_n^{j-1} = \Sigma\, h_{k-2n}\, c_k^j \quad , \quad d_n^{j-1} = \Sigma\, g_{k-2n}\, c_k^j$
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware Accelerated Filtering (Matthias Hopf)
VIS Group, University of Stuttgart

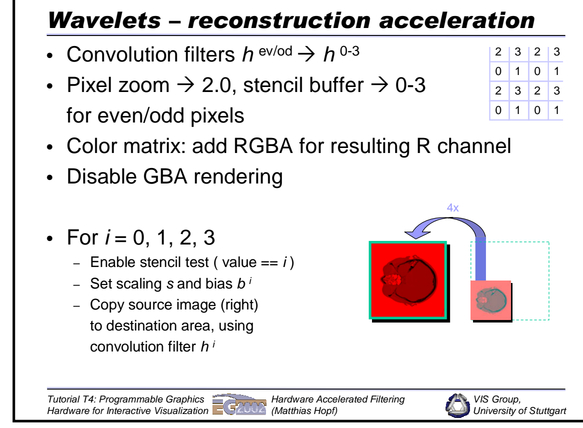

Wavelets – decomposition
c_k^j
0 1 2 3 4 5
$h_0 h_1$
c_k^{j-1}
0 1 2
c_k^j
0 1 2 3 4 5
$g_0 g_1$
d_k^{j-1}
0 1 2
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware Accelerated Filtering (Matthias Hopf)
VIS Group, University of Stuttgart

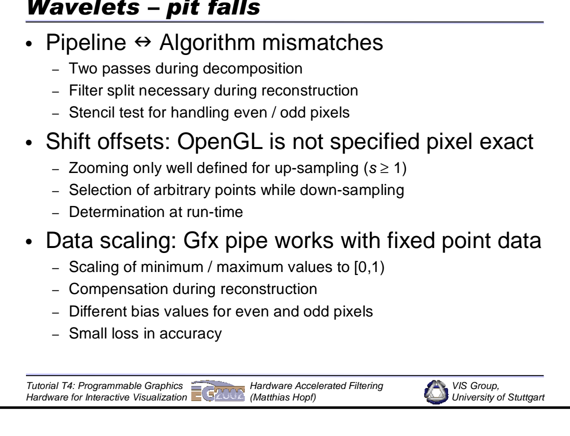

Wavelets – reconstruction
c k j
d k j
0 1 2
h1 h0
g1 g0
c k j+1
0 1 2 3 4 5
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware Accelerated Filtering (Matthias Hopf)
VIS Group, University of Stuttgart


Wavelets – mapping to graphics hardware
p n+1 = cl ¶ bl ¶ zo ¶ tr ¶ cl ¶ cm ¶ co (p n)
bl (pi) = Γ (pi , pi n) Blending
zo (pi) = p ⌊zi⌋ Zoom
tr (pi) = p i-xs+xd Transposition
cl (pi) = max (0, min (1, pi)) Clamping
cm (pi) = M · pi Color Matrix
co (pi) = s · Σ kj pi+j + b Convolution
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware Accelerated Filtering (Matthias Hopf)
VIS Group, University of Stuttgart

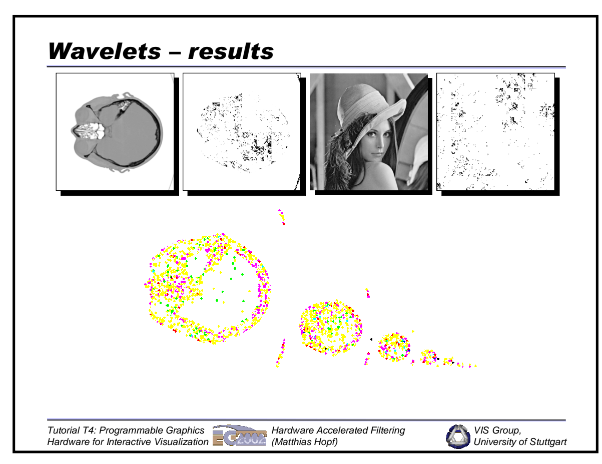

Wavelets – data layout
Two possibilities
For speedup store the four differently filtered signals in R, G, B, and A of the destination image
low-pass filtered in x, low-pass filtered in y → R
hi-pass filtered in x, low-pass filtered in y → G
low-pass filtered in x, hi-pass filtered in y → B
hi-pass filtered in x, hi-pass filtered in y → A
B
A
R
G
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware Accelerated Filtering (Matthias Hopf)
VIS Group, University of Stuttgart


Wavelets – decomposition acceleration
Convolution filter h with hR = low/low ,..., hA = hi/hi
Color matrix: Distribute R to RGBA
Copy red channel of source image (left), keep source GBA channels for reconstruction
Pixel zoom → 0.5, scaling+bias, no color matrix
Copy destination area onto itself with convolution
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware Accelerated Filtering (Matthias Hopf)
VIS Group, University of Stuttgart


Wavelets – reconstruction acceleration
Convolution filters h ev/od → h 0-3
Pixel zoom → 2.0, stencil buffer → 0-3 for even/odd pixels
Color matrix: add RGBA for resulting R channel
Disable GBA rendering
2 3 2 3
0 1 0 1
2 3 2 3
0 1 0 1
For i = 0, 1, 2, 3
Enable stencil test (value == i)
Set scaling s and bias b i
Copy source image (right) to destination area, using convolution filter h i
4x
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware Accelerated Filtering (Matthias Hopf)
VIS Group, University of Stuttgart


Wavelets – pit falls
Pipeline ⇔ Algorithm mismatches
Two passes during decomposition
Filter split necessary during reconstruction
Stencil test for handling even / odd pixels
Shift offsets: OpenGL is not specified pixel exact
Zooming only well defined for up-sampling (s ≥ 1)
Selection of arbitrary points while down-sampling
Determination at run-time
Data scaling: Gfx pipe works with fixed point data
Scaling of minimum / maximum values to [0,1)
Compensation during reconstruction
Different bias values for even and odd pixels
Small loss in accuracy
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware Accelerated Filtering (Matthias Hopf)
VIS Group, University of Stuttgart

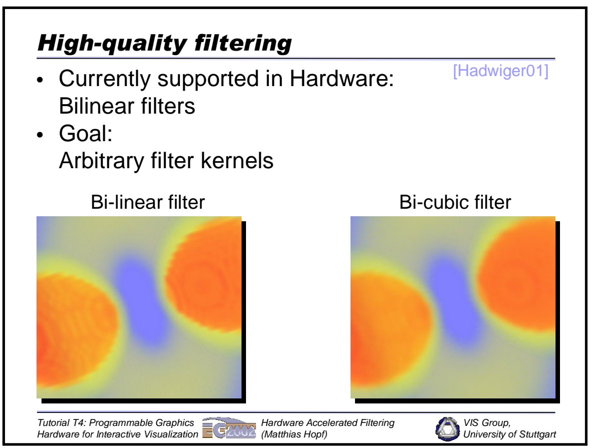
Wavelets – results



High-quality filtering

[Hadwiger01]

- Currently supported in Hardware: Bilinear filters
- Goal: Arbitrary filter kernels



Tutorial T4: Programmable Graphics Hardware for Interactive Visualization — Hardware Accelerated Filtering (Matthias Hopf) — VIS Group, University of Stuttgart

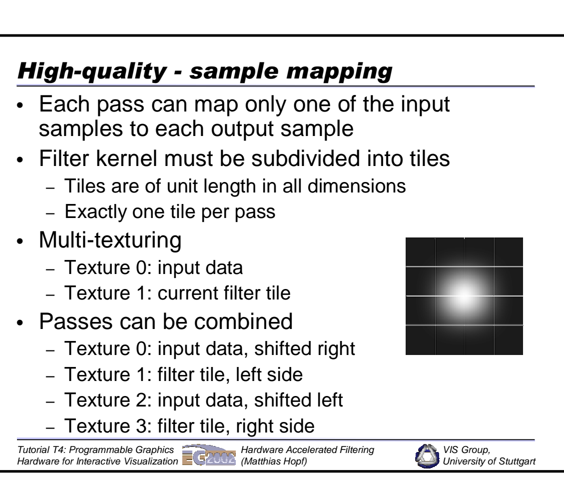
High-quality - mixed loop order

- Distribute contributions of one filter coefficient per output sample to all samples
- Use multiple passes for filter kernel contributions
- Trivial example: 1D linear filter → 2 passes
  - Pass 1:'right-hand neighbors'
  - Pass 2: 'left-hand neighbors'

input samples
input samples

Tutorial T4: Programmable Graphics Hardware for Interactive Visualization — Hardware Accelerated Filtering (Matthias Hopf) — VIS Group, University of Stuttgart

High-quality - sample mapping

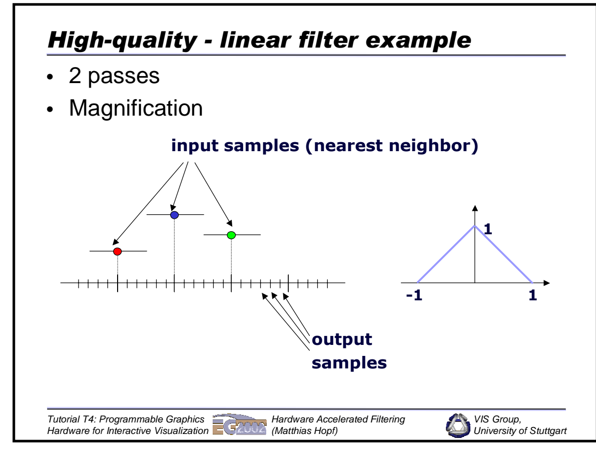


- Each pass can map only one of the input samples to each output sample
- Filter kernel must be subdivided into tiles
  - Tiles are of unit length in all dimensions
  - Exactly one tile per pass
- Multi-texturing
  - Texture 0: input data
  - Texture 1: current filter tile
- Passes can be combined
  - Texture 0: input data, shifted right
  - Texture 1: filter tile, left side
  - Texture 2: input data, shifted left
  - Texture 3: filter tile, right side

Tutorial T4: Programmable Graphics Hardware for Interactive Visualization — Hardware Accelerated Filtering (Matthias Hopf) — VIS Group, University of Stuttgart

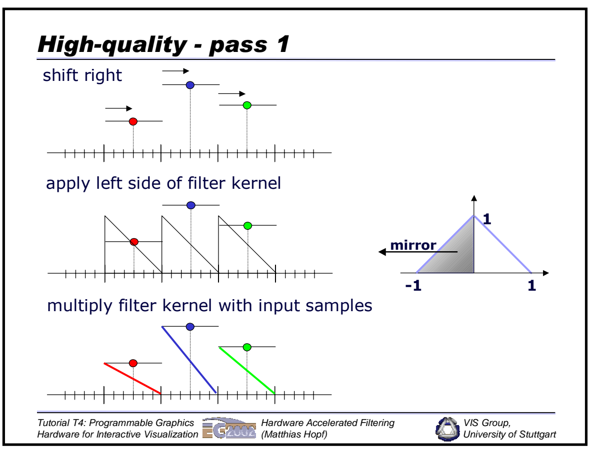
High-quality - linear filter example

- 2 passes
- Magnification

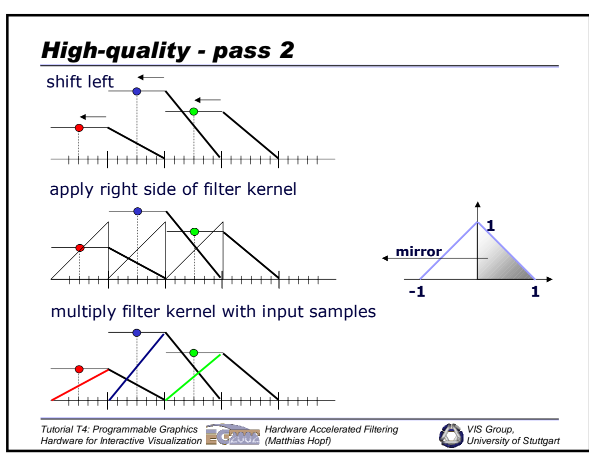


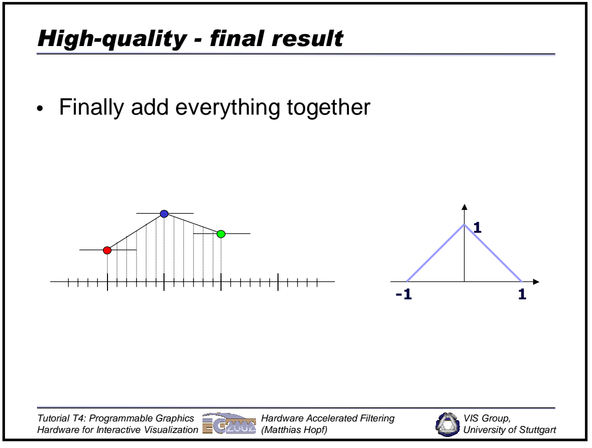
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization — Hardware Accelerated Filtering (Matthias Hopf) — VIS Group, University of Stuttgart

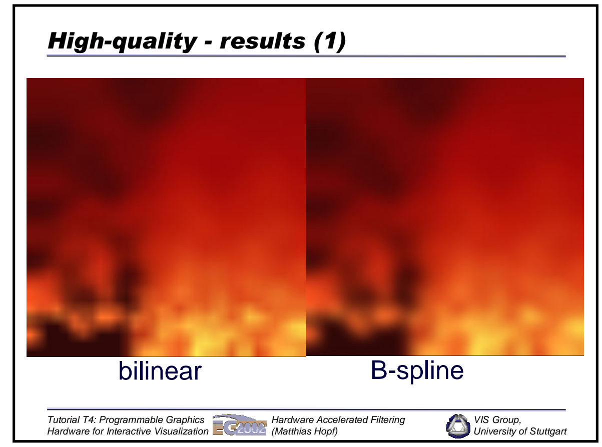
High-quality - pass 1

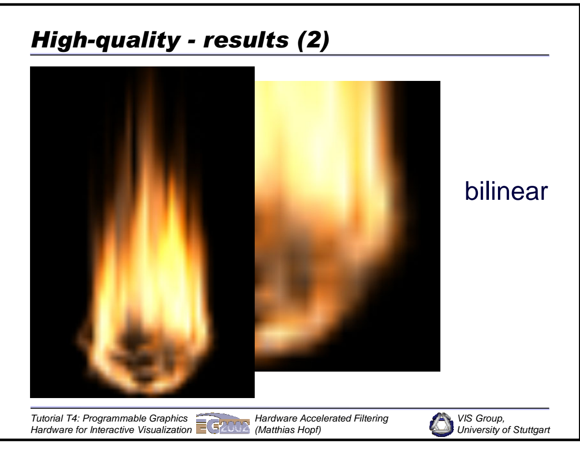


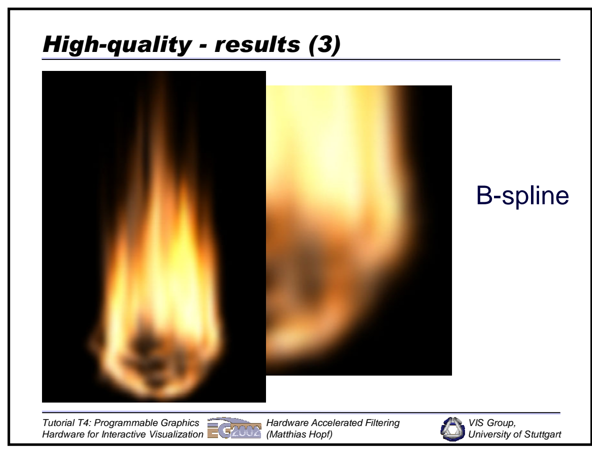
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization — Hardware Accelerated Filtering (Matthias Hopf) — VIS Group, University of Stuttgart

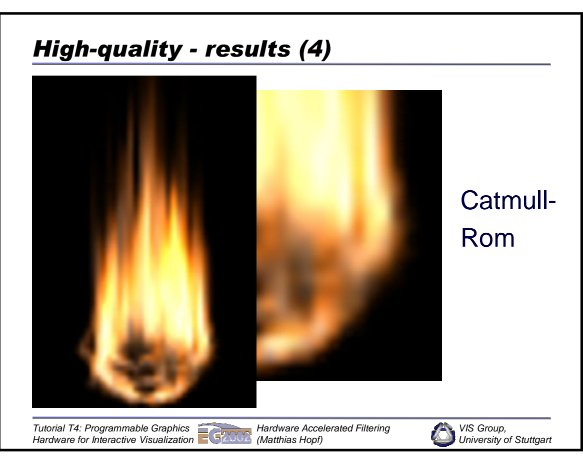

High-quality - pass 2
shift left
apply right side of filter kernel
mirror
1
-1
1
multiply filter kernel with input samples
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware Accelerated Filtering (Matthias Hopf)
VIS Group, University of Stuttgart


High-quality - final result
Finally add everything together
1
-1
1
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware Accelerated Filtering (Matthias Hopf)
VIS Group, University of Stuttgart


High-quality - results (1)
bilinear
B-spline
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware Accelerated Filtering (Matthias Hopf)
VIS Group, University of Stuttgart


High-quality - results (2)
bilinear
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware Accelerated Filtering (Matthias Hopf)
VIS Group, University of Stuttgart


High-quality - results (3)
B-spline
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware Accelerated Filtering (Matthias Hopf)
VIS Group, University of Stuttgart


High-quality - results (4)
Catmull-
Rom
Tutorial T4: Programmable Graphics Hardware for Interactive Visualization
Hardware Accelerated Filtering (Matthias Hopf)
VIS Group, University of Stuttgart

Pipeline pros and cons

- Imaging Pipeline
  - Convolution
  - Pixel zoom
  - Color tables, pixel texs
- Regular Pipeline
  - Multi pass + blending
  - Min/Mag texture filters
  - Dependent tex lookups
  - Pixel shader & Co

Imaging Pipeline:
+ Fewer passes
+ Natural loop order
+ Local memory access
- Less flexible
- Availability

Regular Pipeline:
+ More complex operations
+ Pass collapsing
+ Better support in the future
- Accuracy
- Memory access

Conclusion

- Pros and cons using graphics hardware

  + Computational speed
  + Memory / bus transfer
  - Accuracy
  - Complexity

- And the future?

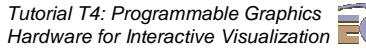
  - Faster processors
  + Even faster graphics hardware
  + Programmable pipes
  + Floating point frame buffers

Literature

- [Hopf99]
  M. Hopf, T. Ertl.
  Accelerating 3D Convolution using Graphics Hardware.
  Visualization '99, San Francisco, CA, IEEE.

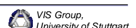
- [Hopf00]
  M. Hopf, T. Ertl.
  Hardware Accelerated Wavelet Transformations.
  VisSym '00, Amsterdam, Netherland, Springer.

- [Hadwiger01]
  M. Hadwiger, T. Theußl, H. Hauser, E. Gröller.
  Hardware-Accelerated High-Quality Filtering on PC Hardware,
  VMV *'01*, Stuttgart, Germany, Infix.



Texture Compression

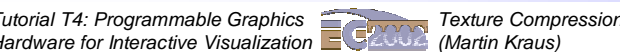
Martin Kraus
Visualization and Interactive
Systems Group, Stuttgart



Texture Compression: Overview

Contents:

- Requirements for texture compression
- S3 Texture Compression (S3TC)
- Vector Quantization (VQ)
- Implementing VQ texture compression with programmable graphics hardware
- Application: Volume Visualization

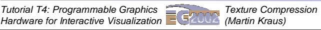


Requirements for Texture Compression

- Encoding/compression of texture maps:
  - Maximize compression rates,
  - Minimize information loss,
  - User-defined trade-off.
- Access/decompression of texture maps:
  - Fast random access,
  - Simple decompression algorithm.

S3 Texture Compression (S3TC)

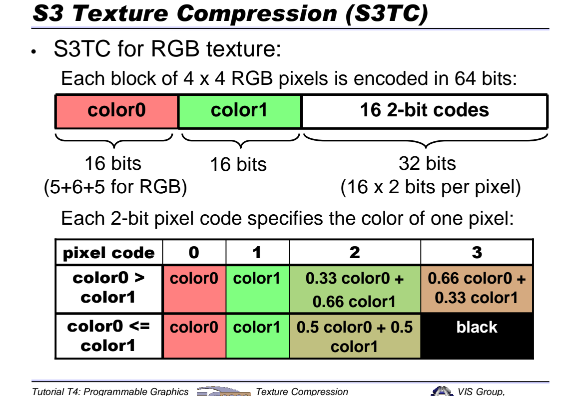
- A.k.a. DXTC in Microsoft's DirectX API.
- OpenGL extension: EXT_texture_compression_s3tc.
- Hardware implementation in several graphics chips, e.g.
  - NVIDIA GeForce series,
  - ATI Radeon 7500.



S3 Texture Compression (S3TC)

- S3TC for RGB texture:

  Each block of 4 x 4 RGB pixels is encoded in 64 bits:



pixel code	0	1	2	3
color0 > color1	color0	color1	0.33 color0 + 0.66 color1	0.66 color0 + 0.33 color1
color0 <= color1	color0	color1	0.5 color0 + 0.5 color1	black

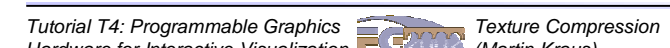
S3 Texture Compression (S3TC)

- Compression ratio in this case:

  original RGB data of 4x4 block: 4 x 4 x 3 = 48 bytes
  compressed data: 64 bits = 8 bytes
  6 : 1 ratio
- More variants for RGBA data.
- Advantages:
  - Simple encoding/decoding,
  - Data-independent.
- Disadvantages:
  - Moderate compression ratios,
  - Block-artifacts, inappropriate for non-smooth data,
  - Fixed compression scheme.

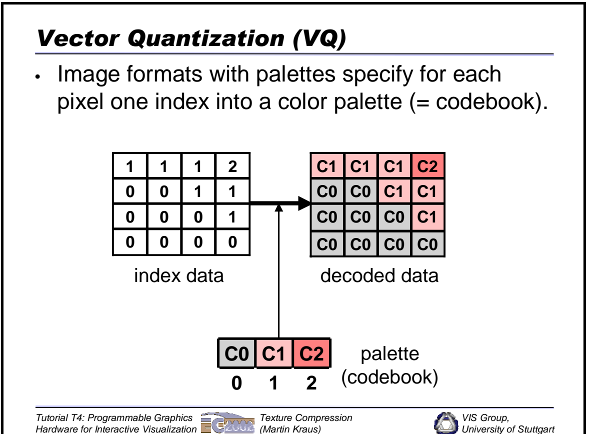
Vector Quantization (VQ)

- Well established lossy compression technique, e.g. image formats with color palettes.
  See [Gersho 1992] for details about VQ.
- Apart from palettes seldom employed for texture maps in hardware, e.g. PowerVR architecture.
- Frequently employed for texture maps in software [Ning 1992, Levoy 1996].



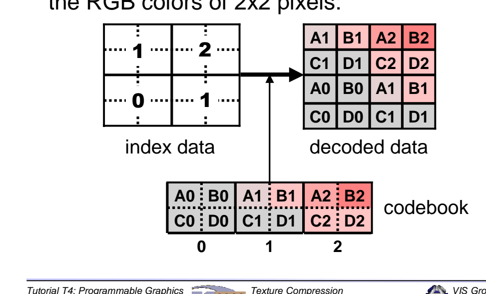
Vector Quantization (VQ)

- Image formats with palettes specify for each pixel one index into a color palette (= codebook).



Vector Quantization (VQ)

- Instead of one 3-D vector for the RGB color of one pixel, we can also encode 12-D vectors for the RGB colors of 2x2 pixels:

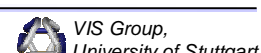


Vector Quantization (VQ)

- Compression ratio in this case:
  original RGB data of 2x2 block: 2 x 2 x 3 = 12 bytes
  index data for codebook of length 256: 1 byte
  12 : 1 ratio
- Advantages:
  - Higher compression ratios,
  - Less compression artifacts,
  - Length of codebook determines quality & comp. ratio.
- Disadvantages:
  - Expensive encoding (codebook generation).
  - Less simple decoding.

Implementing VQ with Prog. Graphics HW

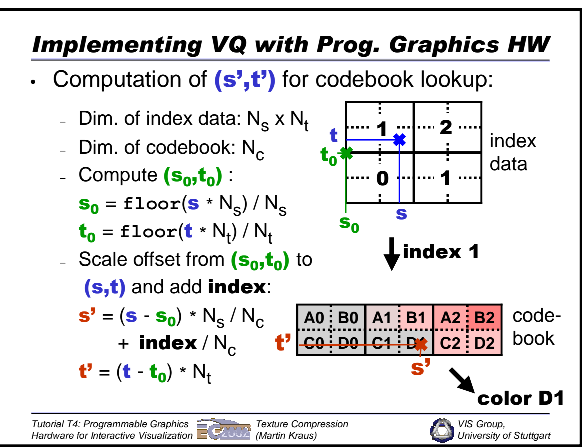
- 2D texture lookup with bilinear interpolation:
  - Nearest-neighbor texture lookup of up to 4 indices,
  - Nearest-neighbor dependent texture lookup of 4 entries of the codebook,
  - Bilinear interpolation.
- Too expensive for current programmable graphics hardware.
- But: nearest-neighbor 2D texture lookup can be implemented with per-pixel operations of current graphics hardware, i.e. ATI Radeon 8500.

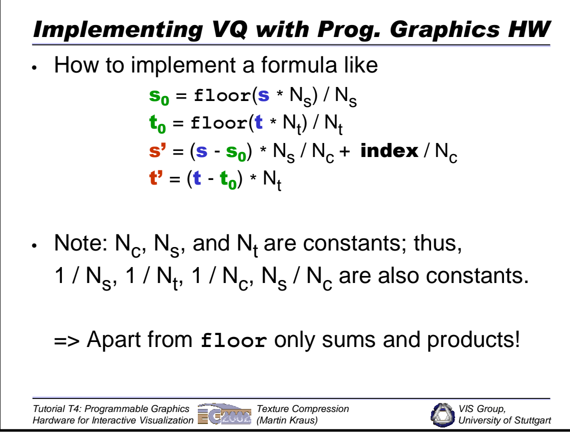


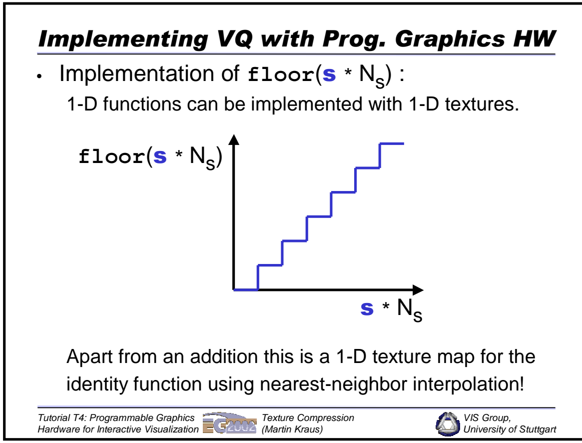
Implementing VQ with Prog. Graphics HW

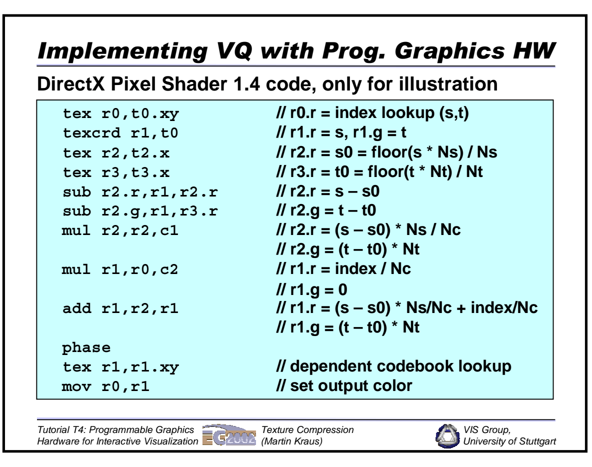
- Nearest-neighbor 2D texture lookup with VQ:
  - Lookup of index:
    Nearest-neighbor 2D texture lookup at **(s,t)**
  - Lookup in codebook:
    Nearest-neighbor, dependent 2D texture lookup at **(s',t')**
  - How to compute **(s',t')** ?

t 1 2 / 0 1 s index data
index 1
A0 B0 A1 B1 A2 B2 / C0 D0 C1 D1 C2 D2 code-book
t' s'
color D1





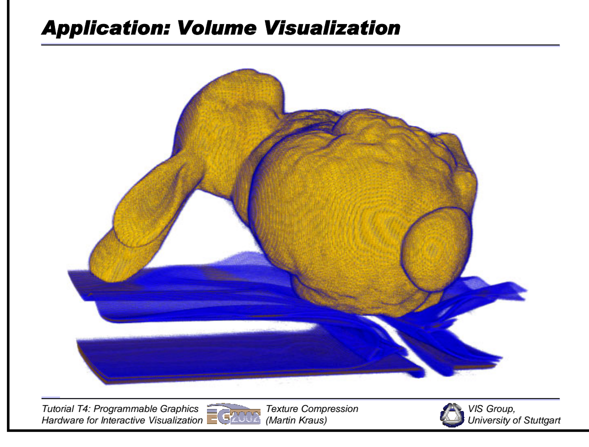




Application: Volume Visualization

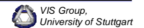
- Approach works also with 3-D textures!
- It may be employed for texture-based volume visualization [Cabral 1994].
- Nearest-neighbor interpolation is often acceptable for large volumes.
- Instead of 2 x 2 pixels, use 2 x 2 x 2 voxels.
- With a codebook of length 256, a 512^3 RGBA-volume is reduced to 256^3 bytes = 16 MBytes, i.e. it fits easily into texture memory.
- See [Kraus 2002] for more details.

Tutorial T4: Programmable Graphics Hardware for Interactive Visualization — Texture Compression (Martin Kraus) — VIS Group, University of Stuttgart



Discussion

- Texture compression with vector quantization on programmable graphics hardware is possible.
- But: Only with nearest-neighbor interpolation.
- Main advantage compared to built-in texture compression: *Programmer is in control!*



References

[Cabral 1994] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Proceedings of 1994 Symposium on Volume Visualization*, pages 91-97.

[Gersho 1992] Allen Gersho and Robert M. Gray. *Vector Quantization and Signal Compresssion.* Kluwer Academic Publiushers, Boston, 1992.

[Kraus 2002] Martin Kraus and Thomas Ertl. Adaptive Texture Maps. In *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware 2002*.

[Levoy 1996] Marc Levoy and Pat Hanrahan. Light Field Rendering. In *Proceedings of SIGGRAPH '96*, pages 31-42.

[Ning 1993] Paul Ning and Lambertus Hesselink. Vector Quantization for Volume Rendering. In *Proceedings of 1992 Workshop on Volume Visualization*, pages 69-74, 1992.