GPU Construction and Transparent Rendering of Iso-Surfaces

Peter Kipfer

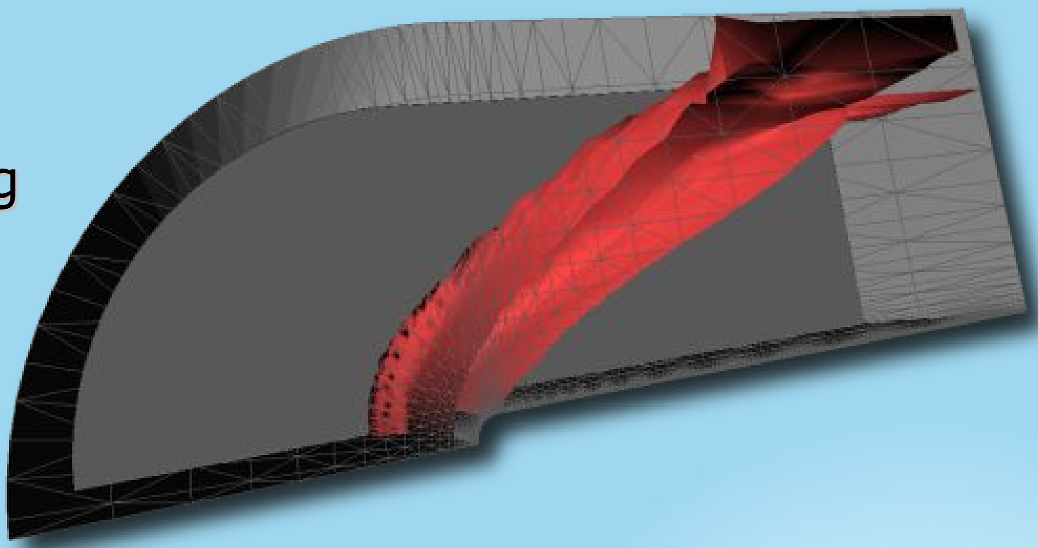
Rüdiger Westermann

computer graphics & visualization

Introduction

- Indirect volume visualization: display of an iso surface

- Fragment-based
  direct volume rendering with special transfer function for immediate display



- Geometry-based
  compute level set inside grid cell and store for further processing and display

Previous work

Marching tetrahedra for indirect volume rendering

- Works for all grids by splitting other element types
- Simple classification with less cases than marching cubes

- Only first order approximation of level set
- More elements

- Perform extraction on the GPU
- Avoids bus transfer bottleneck and exploits memory bandwidth and parallelism for interactive rendering
- Expect a speedup !

Previous GPU work

Implementation of the element classification

- in the vertex shader [Reck et al. 2004]
  - compatible with CPU acceleration structures
  - send all element data all the times
- in the fragment shader [Klein et al. 2004]
  - more compute power and memory bandwidth
  - allows to store the surface vertices using OpenGL SuperBuffers
  - interpolation of vertex attributes very expensive
  - hardware restrictions (shader length)
  - no acceleration structures

Marching Tetrahedra revisited

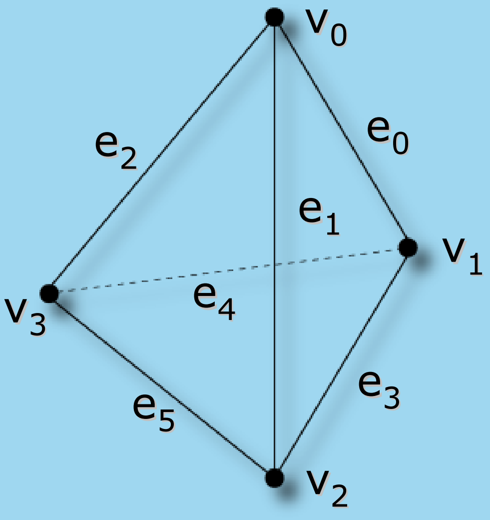
Classic approach: element-centric classification

1. mark element vertices wrt. iso value
2. lookup intersected edges according to marker
3. interpolate surface position along selected edges

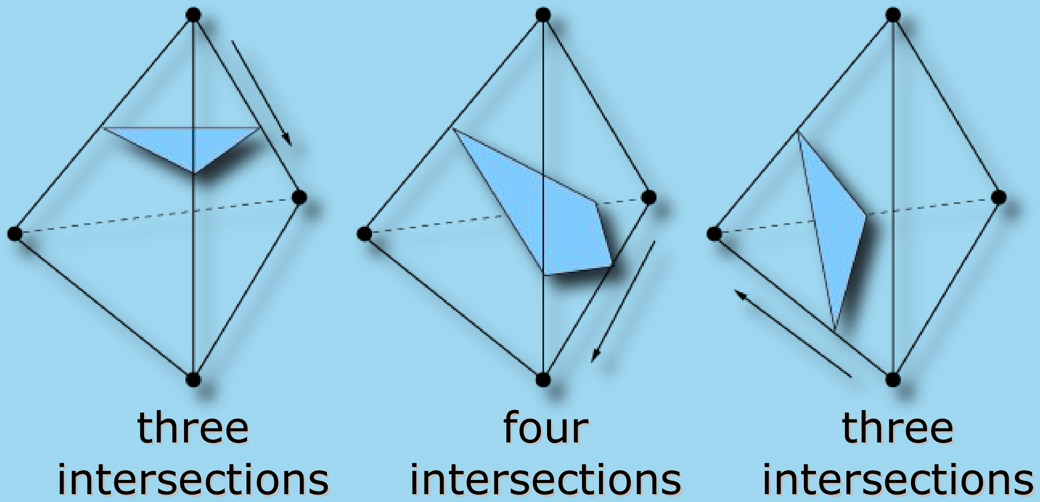
- To avoid redundant interpolations, needs to store and access intermediate results
- On GPUs, vertex/fragment processing is independent
- Repeated classification and interpolation

Marching Tetrahedra revisited

New approach: edge-based processing

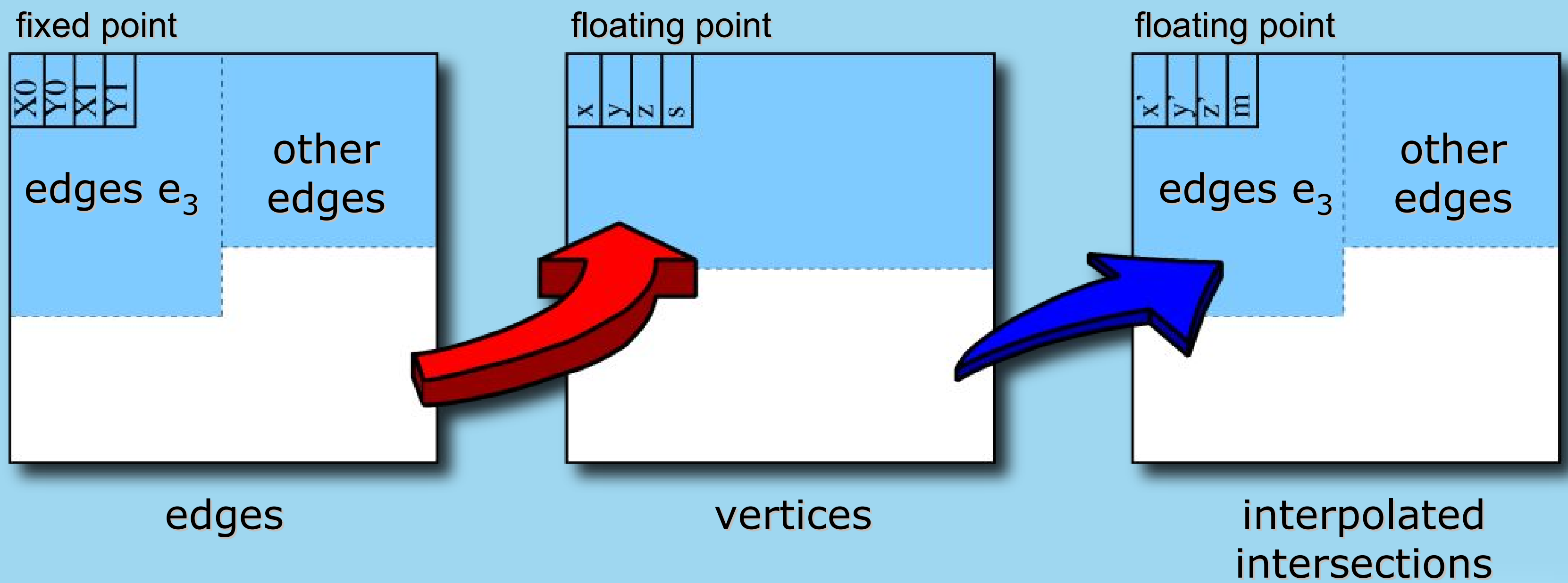


$v_0 \leq v_1 \leq v_2 \leq v_3 \leq v_4$



- Surface uniquely defined by edge intersections
- If vertices are sorted, element is implicitly classified by intersection status of edge e_3

Pass 1: Geometry (interpolation)



- Compute intersection along each edge and mark

-1	iso value smaller than v_1
[0;1]	valid intersection between v_1 and v_2
-2	iso value larger than v_2

Pass 1: Geometry (interpolation)

- Simple and short shader code

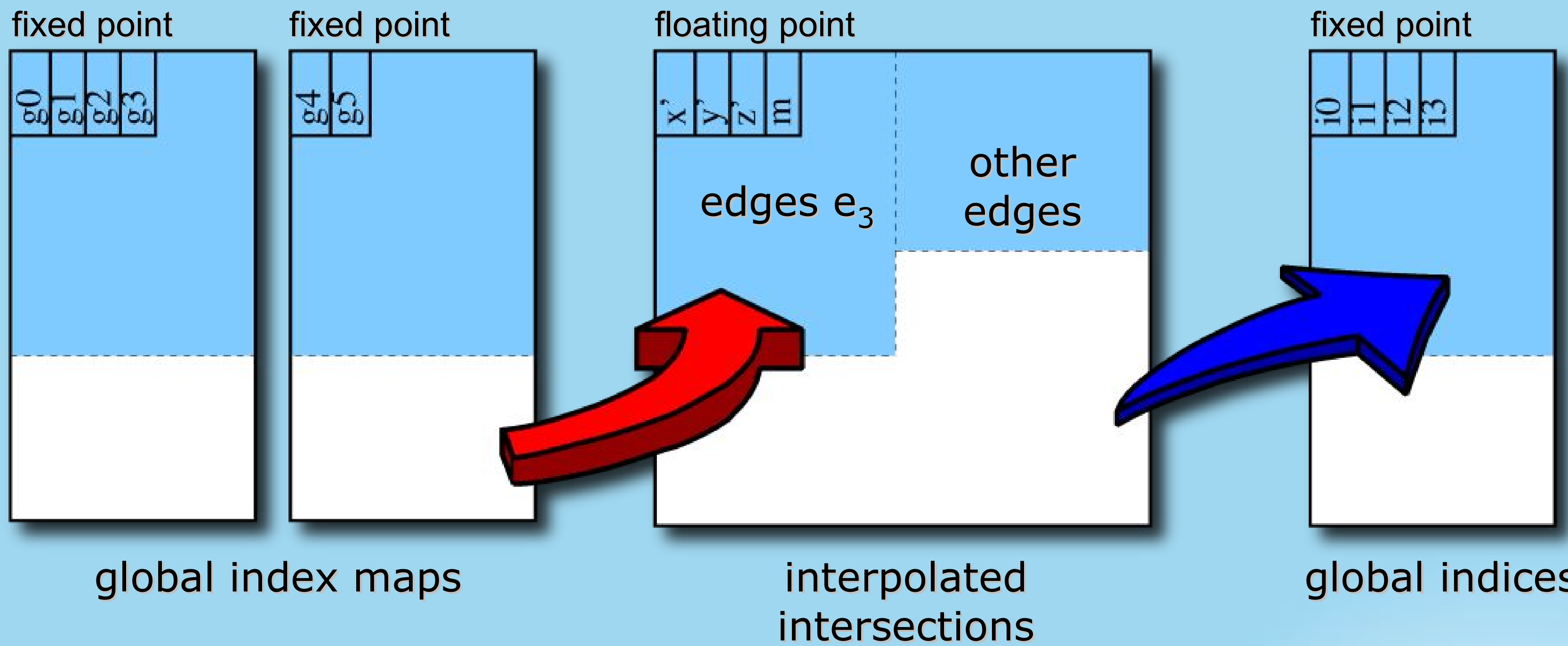
```
edge = tex2D(Edges,TCoord[0]);
v0 = tex2D(Vertices,edge.xy);
v1 = tex2D(Vertices,edge.zw);

// we know that v0 has smaller scalar (stored in w comp.)
d = max(v1.w - v0.w, epsilon);
i = clamp((Iso - v0.w) / d);

result = lerp(v0,v1,i);

if (Iso > v1.w) result.w = -2;
else if (Iso < v0.w) result.w = -1;
else result.w = i;
```

Pass 2: Topology (global indices)



- Fetch global indices according to local index

`marker = -1` : **`[ 0 1 2 2 ]`** triangle-quad

`marker in [0;1]` : **`[ 1 2 3 4 ]`** quad

`marker = -2` : **`[ 2 2 4 5 ]`** triangle-quad

Pass 2: Topology (global indices)

- Short and efficient shader code (17 ARB instr.)

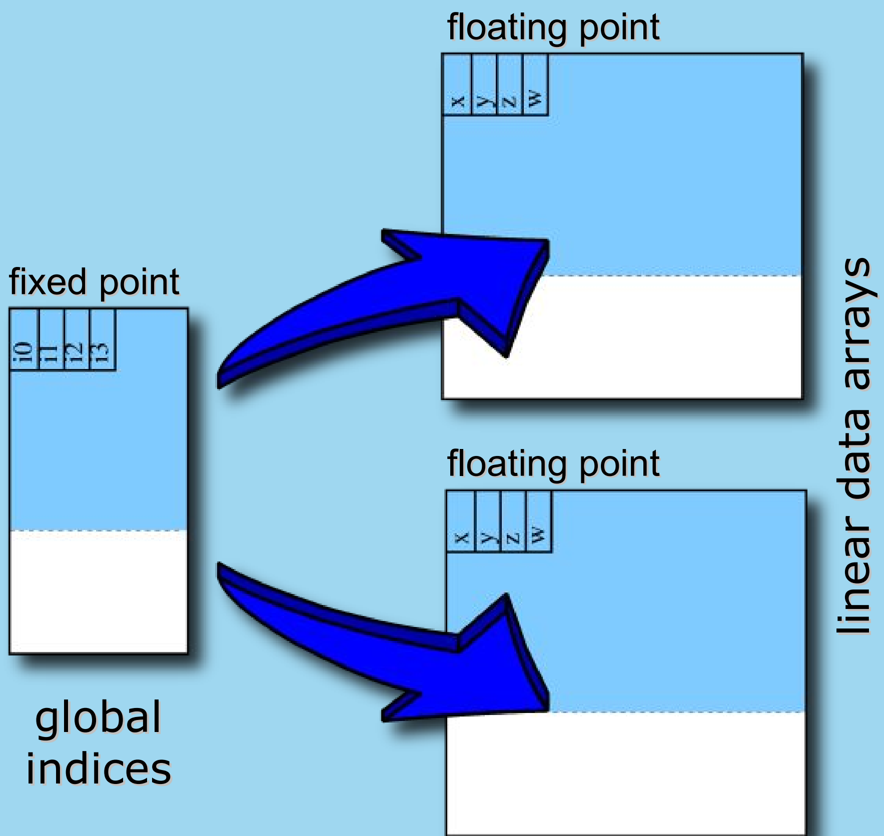
```
v = tex2D(InterpVtx, TCoord[0]);

if (v.w == -1)
  idx = [0, 1, 2, 2]; // iso smaller than values at edge3
else if (v.w == -2)
  idx = [2, 2, 4, 5]; // iso larger than values at edge3
else
  idx = [1, 2, 4, 3]; // flip last two for GL_QUAD draw

// get global edge indices of tet
map0 =  tex2D(Map0, TCoord[0]*[2,1]);
map1 =  tex2D(Map1, TCoord[0]*[2,1]);

res = map1.yyyy;
res = (idx < 5) ? map1.xxxx : res;
res = (idx < 4) ? map0.wwww : res;
res = (idx < 3) ? map0.zzzz : res;
res = (idx < 2) ? map0.yyyy : res;
res = (idx < 1) ? map0.xxxx : res;
```

Pass 3: Linearization (optional)



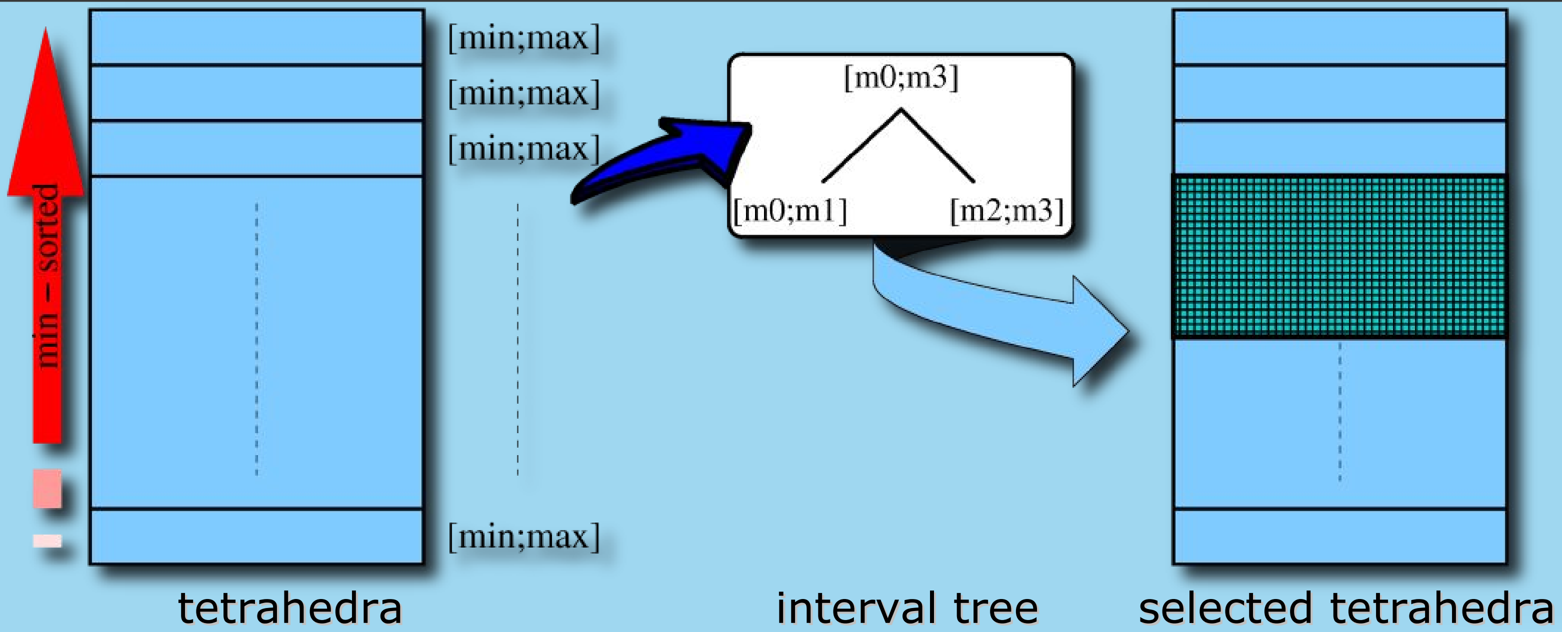
- Assemble linear arrays by indexed lookup into the interpolated vertices
- Create two array buffers for better aspect ratio
- Can write to multiple render targets if supported by shader output bandwidth

Properties

- Resulting geometry and topology buffer are in native OpenGL format
  - direct usability in any application that uses indexed drawing (**`glDrawElements`**)
  - optional third pass creates linear arrays for array-based drawing (**`glDrawArrays`**)
- Passes 1 and 3 can be carried out with any vertex attribute
  - interpolate any attribute on the surface (color, texture coordinates, normals, ...)
  - pass 2 (global indices) necessary only once
- Short shader code and good cache coherence

computer graphics & visualization

Acceleration structure



- Don't process non-contributing tetrahedra:
  - Global minimum scalar sort
  - Store min/max scalar value per row in an interval tree
  - Traverse tree for iso value to get contributing rows
  - Row-range valid for all passes

Processing

- Pre-processing
  - Sort tetrahedra, create canonical vertex ordering, split field into regions if too much tetrahedra for given index Bit-width
  - Build interval tree
- Extraction
  - Traverse interval tree to determine active regions and quad sizes
  - Perform extraction passes
  - Draw surface

Performance

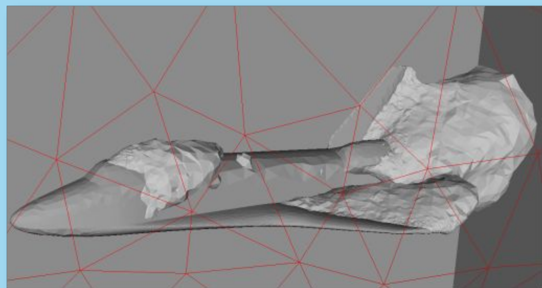
million tets per second in GPU memory	Interval tree disabled		Interval tree enabled	
Extract (2 Pass)	65	112	83	143
Extract (3 Pass)	21	44	52	69
Extract + Render	15	29	42	57
	ATI 9800Pro	ATI X800 XT	ATI 9800Pro	ATI X800 XT

Memory requirements

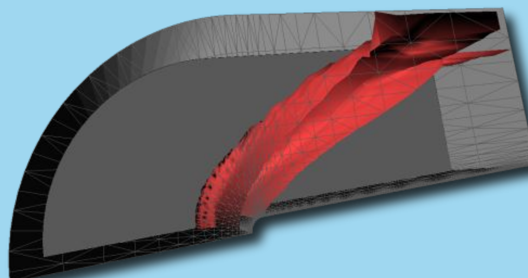
	element-based [Klein et al. 2004]	edge-based this approach
vertex data	128 Bit	128 Bit
element data	224 Bit	128 Bit
texture read bandwidth	216 Bit	130 Bit*

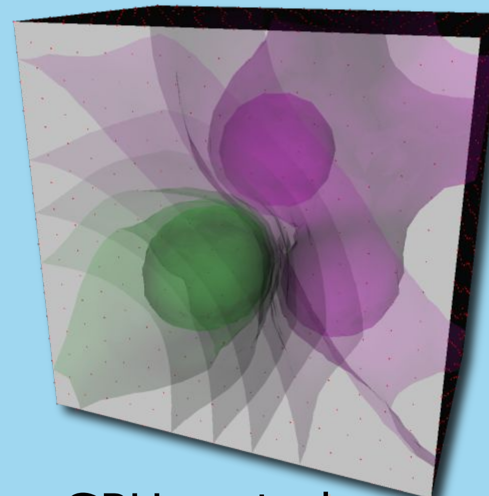
* edge valence = 6

Demo

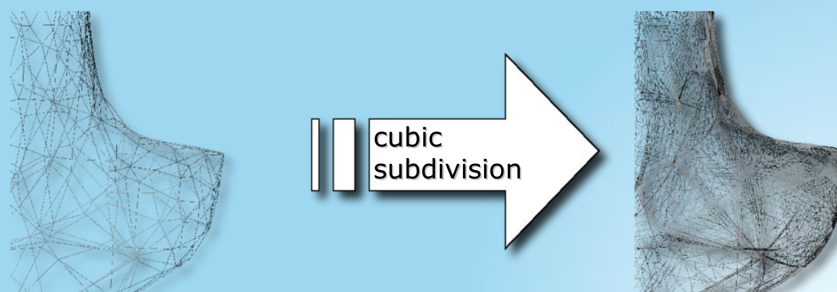


Linear interpolated
vertex attributes





GPU sorted
transparency



OpenGL extensions
third-party shader

Thanks for listening !

Questions ?

Demo + Infos

http://wwwcg.in.tum.de