The Graphics Processing Unit (GPU) as a high performance computational resource for simulation and geometry processing

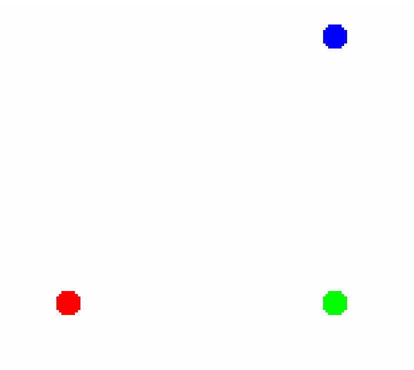
Tor Dokken
Department of Applied Mathematics
SINTEF Information and Communication Technology
www.sintef.no/gpgpu/
tor.dokken@sintef.no

March 10th 2006

Graphics cards as a high-end computational resource

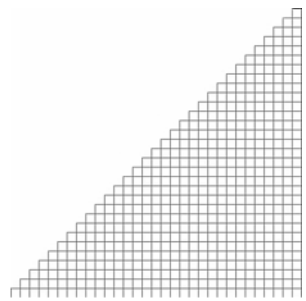
- Project funded by the Research Council of Norway
- Main partner: SINTEF ICT, Department of Applied Mathematics
- Period: 2004-2007
- Cooperation with a number of academic partners in Norway:
  - Center of Mathematics for Application at the University of Oslo, Department of Informatics University of Oslo
    - 3 Ph.D. fellows (1. University of Oslo and 2 SINTEF)
    - 6 master students (January 2006-June 2007) (Will sit at SINTEF)
  - Mathematics Department, University of Bergen
    - 1 Post. Doc
  - Narvik University College, Narvik
    - 3 master students in 2006, 7 in 2005, 1 in 2004

Steps in the graphics pipeline

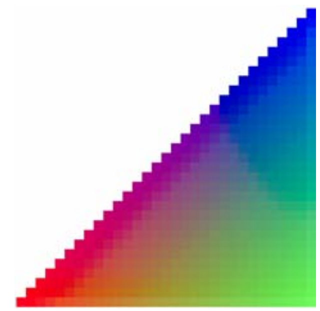


Vertices after transformation and coloring, per vertex operations

(Vertex processor)



Rasterization.



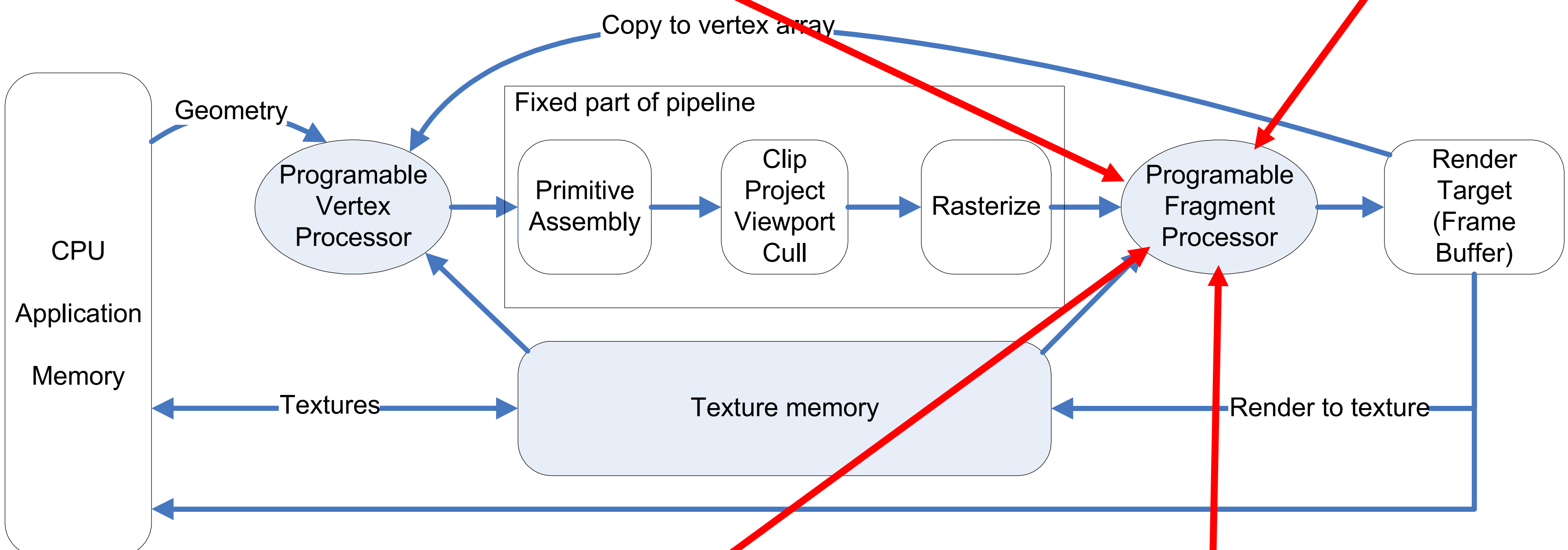
Texturing and coloring, fragment processing.

(Fragment processor)

Why the interest in GPUs?

2003: GPUs with 32 bit floating point arithmetic, programmable vertex and fragment processors introduced.

2004: 16 pipelines x 2 processing units x 4 flops
= Up to 128 flops in parallel
450 MHz: Synthetic test: 53 GFLOPs



2005: NVIDIA GeForce 7800 GTX:
24 pipelines x 2 processing units x 4 flops
= Up to 192 flops in parallel
450 MHz: Synthetic test: 165 GFLOPs

2006: NVIDIA GeForce 7800 GTX 512:
Increased frequency to 550MHz, faster memory

ATI X1900
48 pipelines 650 MHz
How fast ?

Typical price for high performance GPUs 500€.

Vertex Processor

- Vertex processor capabilities:
  - Vertex transformation.
  - Normal transformation.
  - Texture coordinate generation and transformation.
  - Lighting calculations.

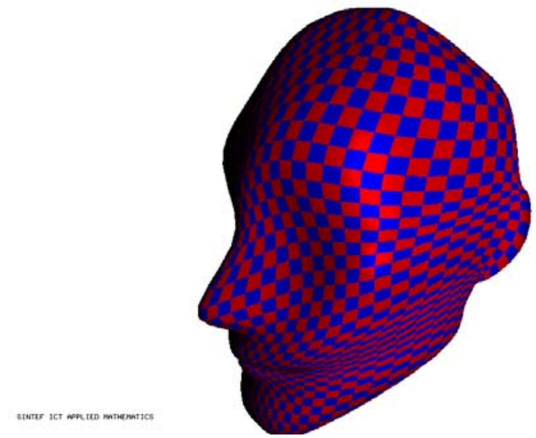
- Fully programmable.
- Processes 4-component vectors (xyzw)
- Can change the position of current vertex.
- Can not read info from other vertices.

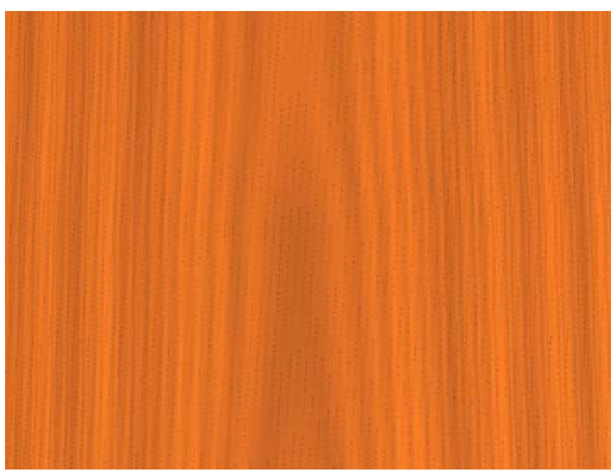
Fragment Processor

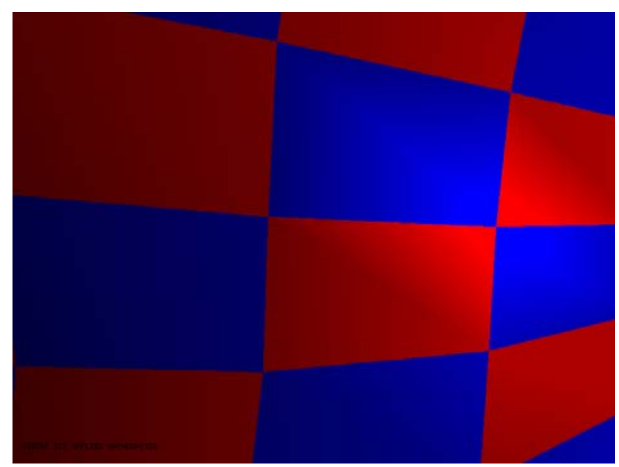
- Main capabilities is to calculate the final color and/or depth to a fragment.
- Fully programmable.
- Processes 4-component vectors (rgba).
- Can read info from other fragments.
- Can not write to more than one pixel in the same buffer.
- Typically more useful than vertex processor for GPGPU purposes.
- Limited number of registers for temporary variables in a fragment shader (currently NVIDIA 32 registers)

Procedural Textures (Fragment Processor)

- Textures created using different algorithms.
- Often used to create a realistic representation of woods, marbles and others.
- Results usually obtained by using different noise functions.







Programming Fragment Processor Specificities

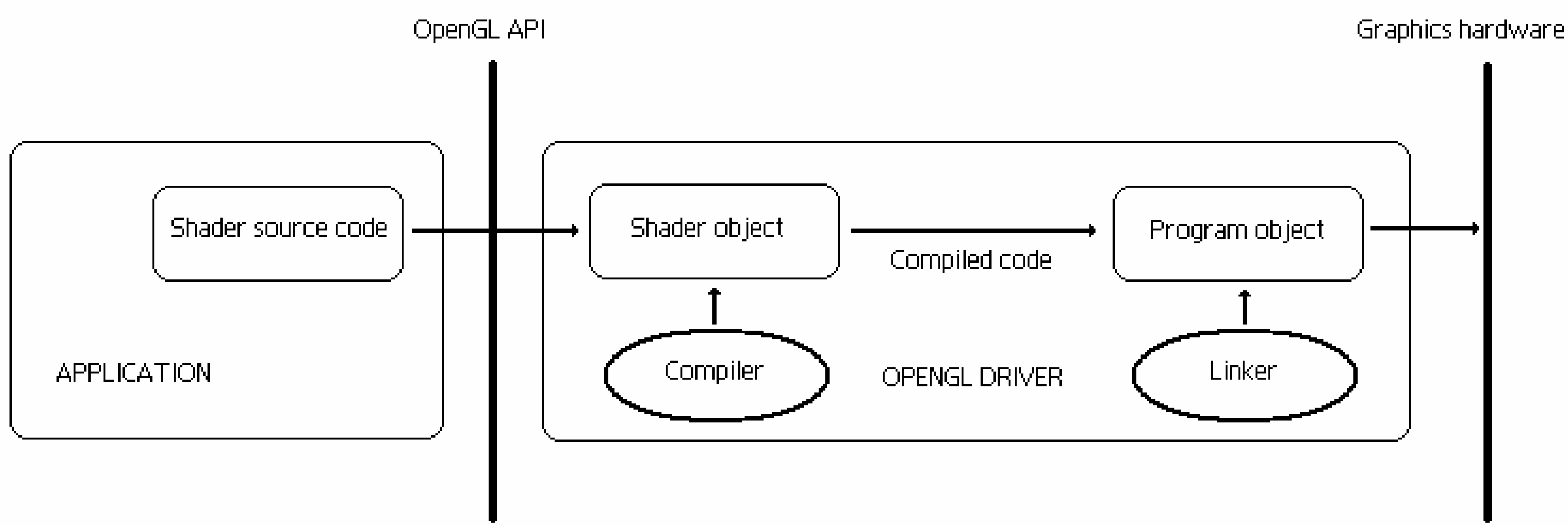
- The programs running on the fragment processor are called ***fragment shaders***
  - A fragment shader can write to up to 4 render targets (textures)
  - A fragment shader can read from many textures,
    - but cannot read from a render target to which it is writing (we do not know the sequence in which the fragments are processed)
  - A render target can be converted to a texture to be used as input in later fragment shaders
- When programming the fragment processor for non-graphical purposes, a default primitive covering the entire viewport should be defined to facilitate the execution of the fragment shaders. High level program languages through DirectX or OpenGL

Shading languages

- High Level Shading Language (HLSL, Microsoft)
  - Part of the DirectX API and only compiles into DirectX code.
  - Hardware independent.
  - Windows only.
  - Game industry.
- ”C for graphics” (Cg, NVIDIA)
  - Platform independent but hardware dependent.
  - Cg and HLSL are very similar languages. Cg/HLSL was co-developed by NVIDIA and Microsoft.
- OpenGL Shading Language (GLSL, ARB)
  - Platform and hardware independent.
  - CAD, scientific visualization, movie industry, academic world...

Using shaders

1. Provide shader source code to OpenGL.
2. Compile shader.
3. Link compiled shaders together.
4. Use program.



The GPU and a double loop

Control structur of "for loops"

For i=1,…,n

F

Input data

Body of "for loop"

• Initiation of a viewport

```
glViewport( 0, 0, n, m);
```

•Initiation of geometry (a quad)

```
glBegin(GL_QUADS);
        glTexCoord2f(0.0f, 0.0f);
        glVertex3f(0.0f, 0.0f, 0.0f);
        glTexCoord2f(1.0f, 0.0f);
        glVertex3f(1.0f, 0.0f, 0.0f);
        glTexCoord2f(1.0f, 1.0f);
        glVertex3f(1.0f, 1.0f, 0.0f);
        glTexCoord2f(0.0f, 1.0f);
        glVertex3f(0.0f, 1.0f, 0.0f);
glEnd();
```

Definition of the clipping volume such that the quad is just inside the clipping volume.

n x m fragments with texture coordinates in [0,1]×[0,1] will be executed

Debugging shaders

- No normal debuggers for GPUs
  - Debugging has to be based on reading values from textures and analyzing these
  - Visualizing the values of computational grid as an image can be helpful for understanding the behavior of the shader.
  - Parallel CPU-implementations often very useful to verify that the shader works properly (and measure performance)
- The reason for an error can be:
  - You may have an error in your algorithm.
  - You may have misunderstood the functionality of the shader language.
  - The driver for the GPU is not working properly.

Why solve PDEs on GPUs?

- Simple grids and data structures.
- Classical finite-difference methods are very simple.
- Embarrassingly parallel.
- Almost perfect speedup expected.
- Best speed up for advance schemes

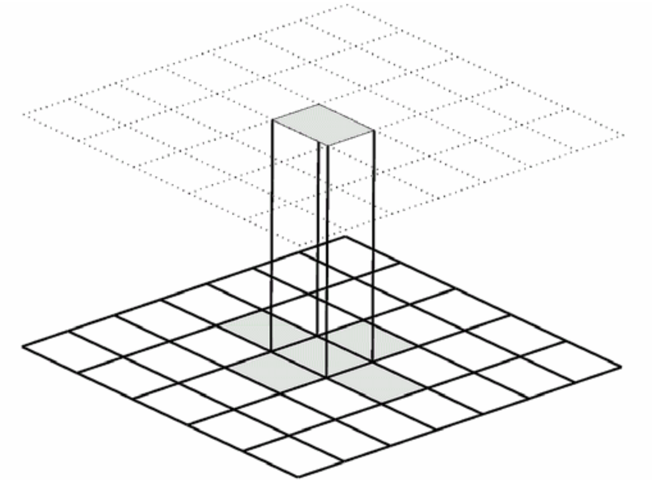
Example: Heat Equation

$$u_t = u_{xx} + u_{yy}$$

Discretication by finite differences over a regular grid:

$$U_{i,j}^{n+1} = U_{i,j}^{n} + \frac{k}{h^2}\left(U_{i+1,j}^{n} + U_{i-1,j}^{n} + U_{i,j-1}^{n} + U_{i,j+1}^{n} - 4U_{i,j}^{n}\right).$$

Each fragment updated as a weighted sum of its nearest five neighbours.



Heat equation shaders

```
[Heat Equation Vertex shader]

varying vec4 texXcoord;
varying vec4 texYcoord;
uniform vec2 dXY;

void main(void)
{
        texXcoord=gl_MultiTexCoord0.yxxx +
                vec4(0.0,0.0,-1.0,1.0)*dXY.x;
        texYcoord=gl_MultiTexCoord0.xyyy +
                vec4(0.0,0.0,-1.0,1.0)*dXY.y;

        gl_Position =
        gl_ModelViewProjectionMatrix*gl_Vertex
        ;
}
```

$$\boxed{U_{i,j}^{n+1} = U_{i,j}^{n} + \frac{k}{h^2}\left(U_{i+1,j}^{n} + U_{i-1,j}^{n} + U_{i,j-1}^{n} + U_{i,j+1}^{n} - 4U_{i,j}^{n}\right).}$$

```
[Heat Equation Fragment shader]

varying vec4 texXcoord;
varying vec4 texYcoord;

uniform sampler2D heatTex;
uniform float r;

void main(void)
{
  vec4 col;
  vec4 tex = texture2D(heatTex, texXcoord.yx);
  vec4 tex0 = texture2D(heatTex, texXcoord.zx);
  vec4 tex1 = texture2D(heatTex, texXcoord.wx);
  vec4 tex2 = texture2D(heatTex, texYcoord.xz);
  vec4 tex3 = texture2D(heatTex, texYcoord.xw);

  col = tex + r*(tex0+tex1-4.0*tex+tex2+tex3);
  gl_FragColor = vec4(col);
}
```

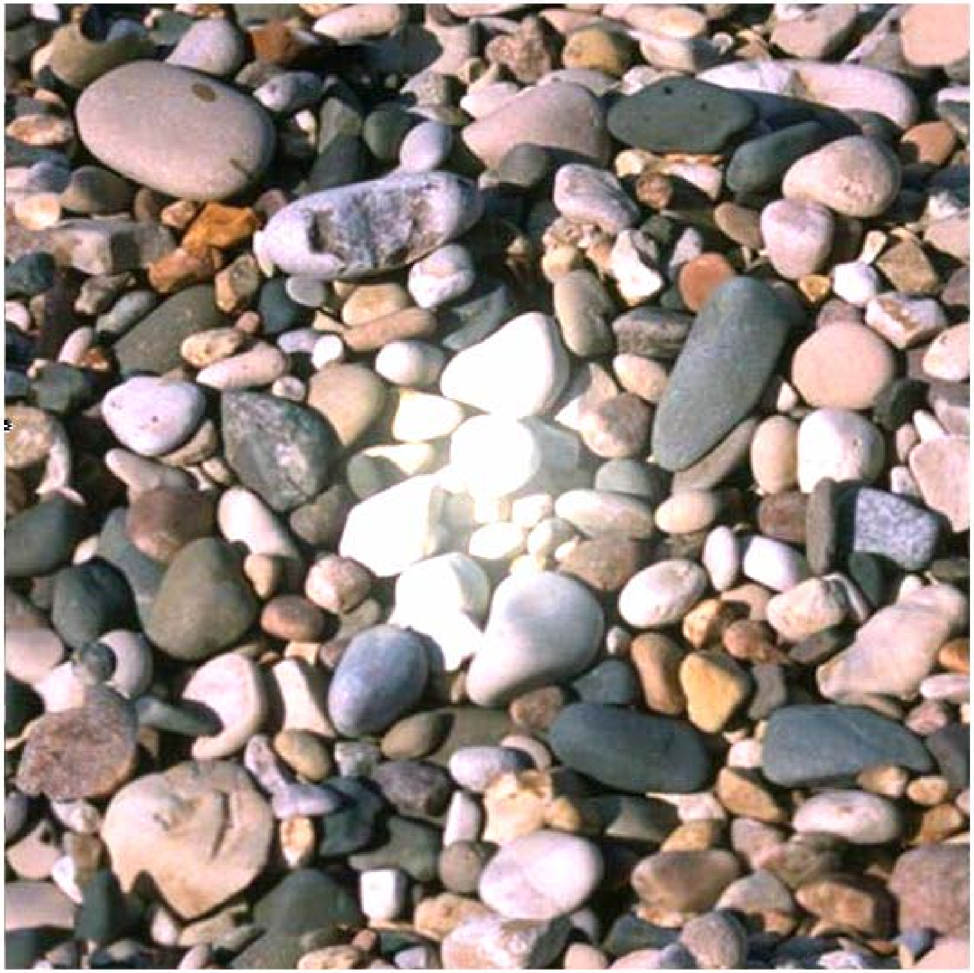
Not using the standard texture coordinates to make more efficient code.

Heat Equation contd.

Wave Equation contd.

SINTEF ICT APPLIED MATHEMATICS

Wave Equation contd.



SINTEF ICT APPLIED MATHEMATICS

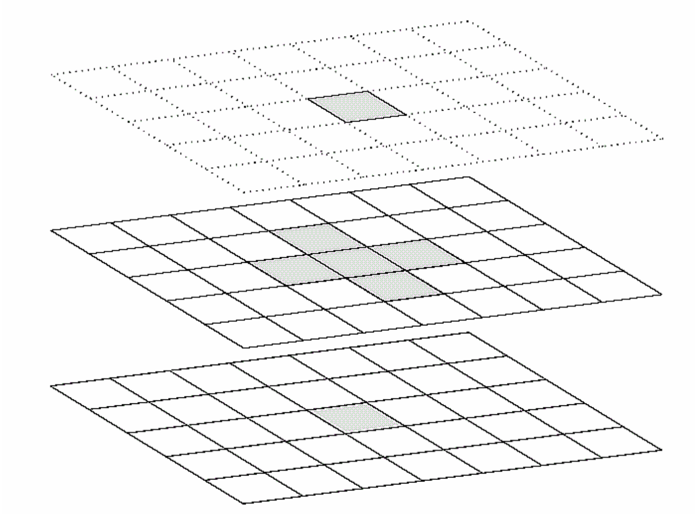
Example: Linear Wave Equation

$$u_{tt} = u_{xx} + u_{yy}$$

Discretication by finite differences over a regular grid:

$$U_{i,j}^{n+1} = 2U_{i,j}^{n} - U_{i,j}^{n-1} + \frac{k^2}{h^2}\left(U_{i+1,j}^{n} + U_{i-1,j}^{n} + U_{i,j-1}^{n} + U_{i,j+1}^{n} - 4U_{i,j}^{n}\right).$$

Almost the heat equation, but needs extra texture to store the value at *n-1*



2. order high-resolution scheme

SINTEF ICT APPLIED MATHEMATICS

Speedup – 2nd order high-resolution

- Runtime per time step and speedup factor for the CPU versus the GPU implementation of *bilinear interpolation* with modified *minmod* limiter for the shock-bubble problem. The results relate to *second-order Runge-Kutta* time stepping.

N	CPU* ms	GPU* ms	Speedup
128x128	30.6	1.27	24.2
256x256	122	4.19	29.1
512x512	486	16.8	28.9
1024x1024	2050	68.3	30.0

* 2.8 GHz Intel Xeon (EM64T)
** GeForce 7800 GTX (450 MHz)

Systems of Conservation Laws

- Fundamental laws of physics: conservation of quantities like mass, momentum and energy.

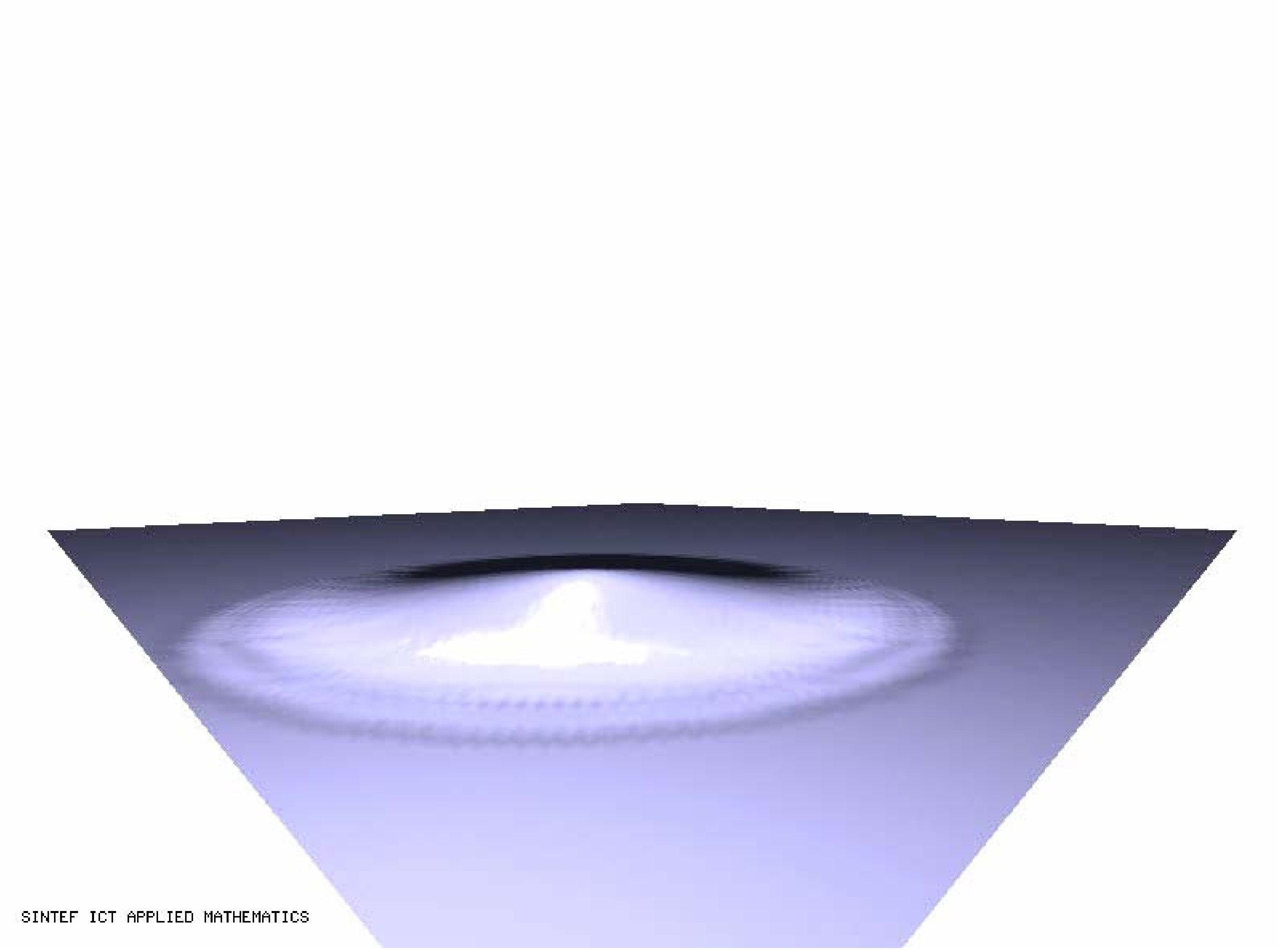
- In arbitrary space dimension this reads:

$$Q_t + \nabla \cdot f(Q) = 0, \qquad\qquad Q(x,0) = Q_0(x)$$

- Example: Shallow water equations

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hu^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Lax-Friedrich



Speedup - *Lax-Friedrichs*

- Runtime per time step and speedup factor for the CPU versus GPU implementation of *Lax-Friedrichs*

N	CPU* ms	GPU** ms	Speedup
128x128	2.22	0.23	9.53
256x256	9.09	0.46	19.8
512x512	37.10	1.47	25.2
1024x1024	148.00	5.54	26.7

* 2.8 GHz Intel Xeon (EM64T)

** GeForce 7800 GTX (450 MHz)

Semi-Discrete High-Resolution Schemes

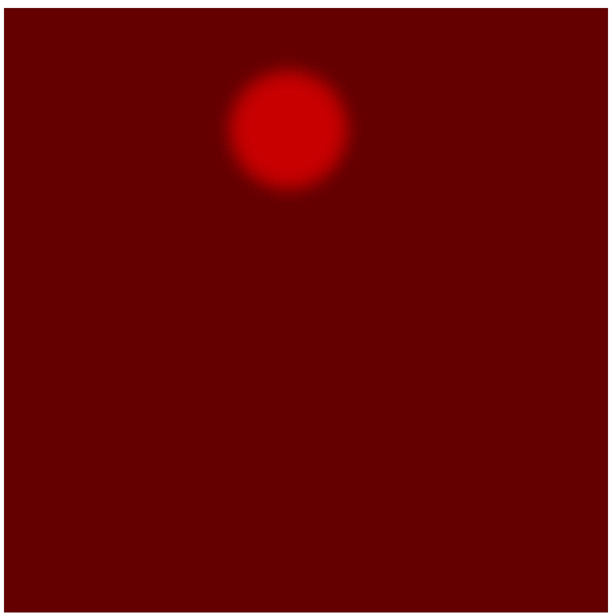
- Evolution of cell averages described by ODEs

$$\frac{d}{dt}U_{ij}(t) = \frac{\left(F_{i-1/2,j}(t) - F_{i+1/2,j}(t)\right)}{\Delta x} + \frac{\left(G_{i,j-1/2}(t) - G_{i,j+1/2}(t)\right)}{\Delta y}$$

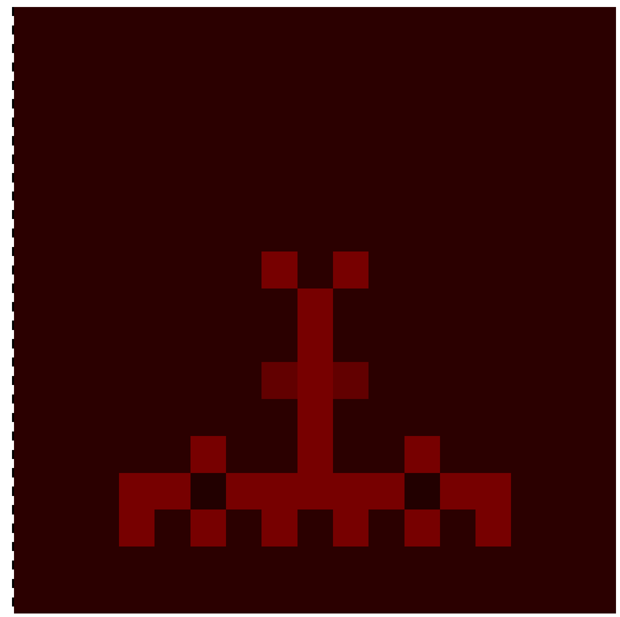
- Steps in the algorithms:
  - Reconstruction of piecewise polynomials from cell averages
  - Evaluation of reconstruction at integration points
  - Numerical computation of edge fluxes
  - Evolution by Runge-Kutta scheme

2nd order high-resolution - Bottom Topography

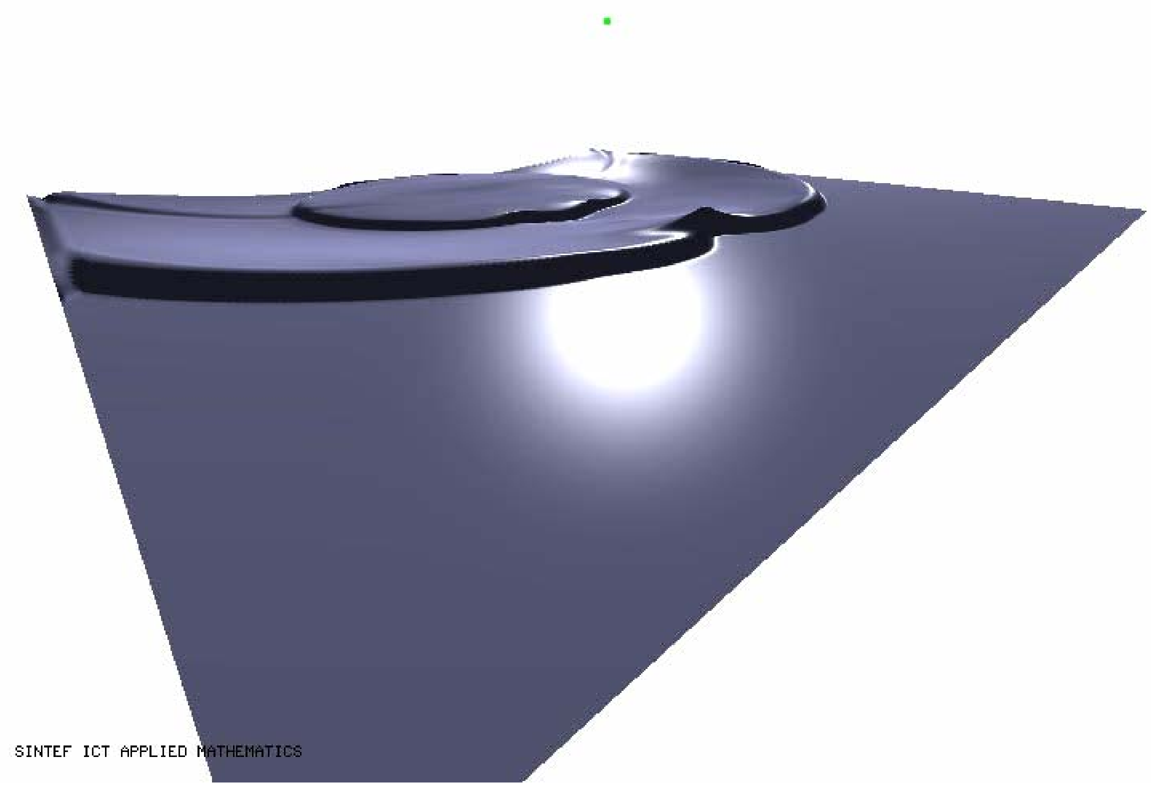
Initial wave map



Initial bottom map



2nd order high-resolution - Bottom Topography

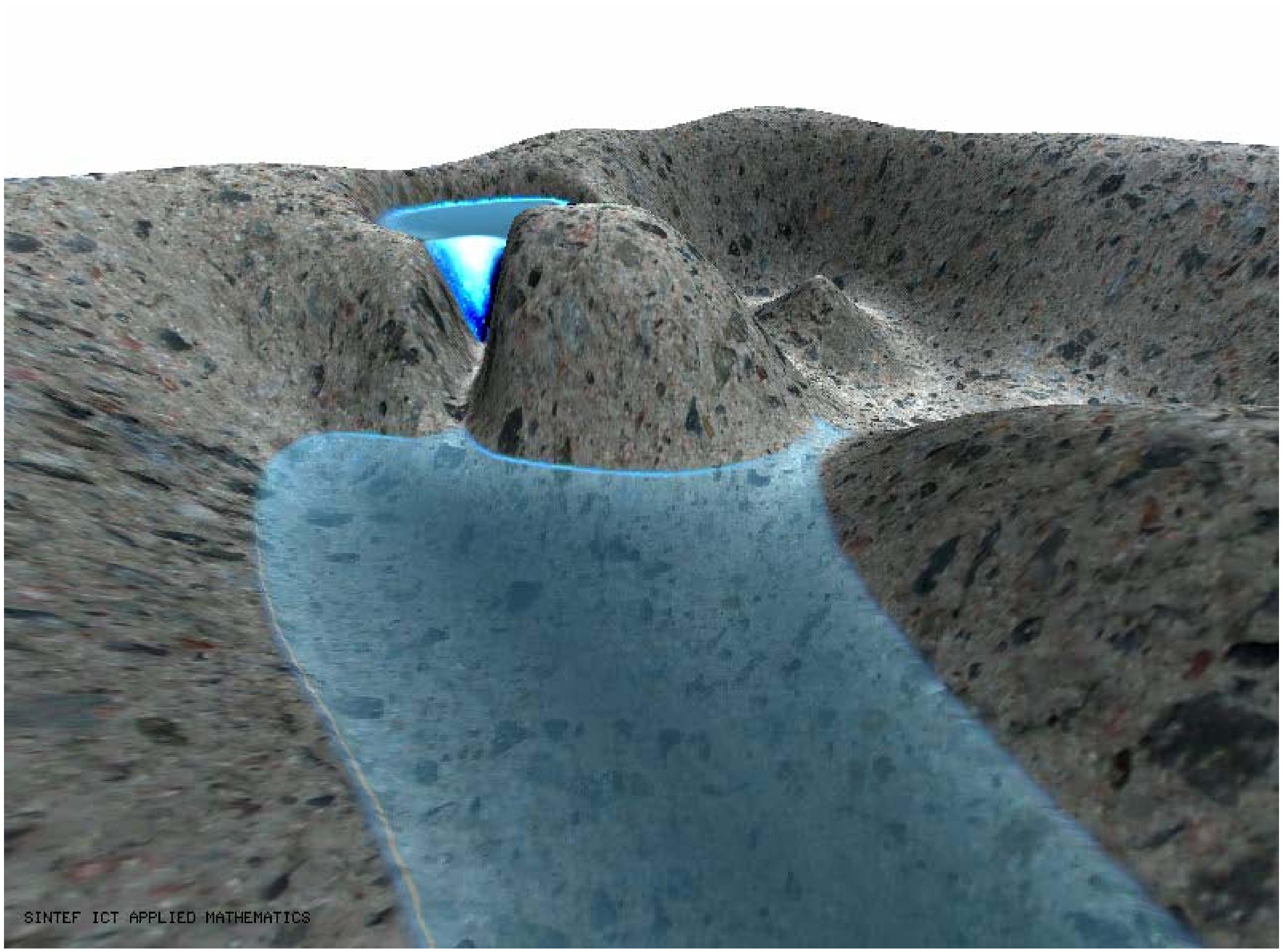


Bilinear Interpolation - Dry States

Initial wave map

Initial bottom map

Bilinear Interpolation - Dry States



Speedup – Dry states

N	CPU* ms	GPU* ms	Speedup
128x128	35.2	2.38	14.7
256x256	143	8.09	17.7
512x512	599	31.9	18.8
1024x1024	3270	142	23.0

* 2.8 GHz Intel Xeon (EM64T)
** GeForce 7800 GTX (450 MHz)

Euler equations

- The two dimensional Euler equations model the dynamics of compressible gasses:

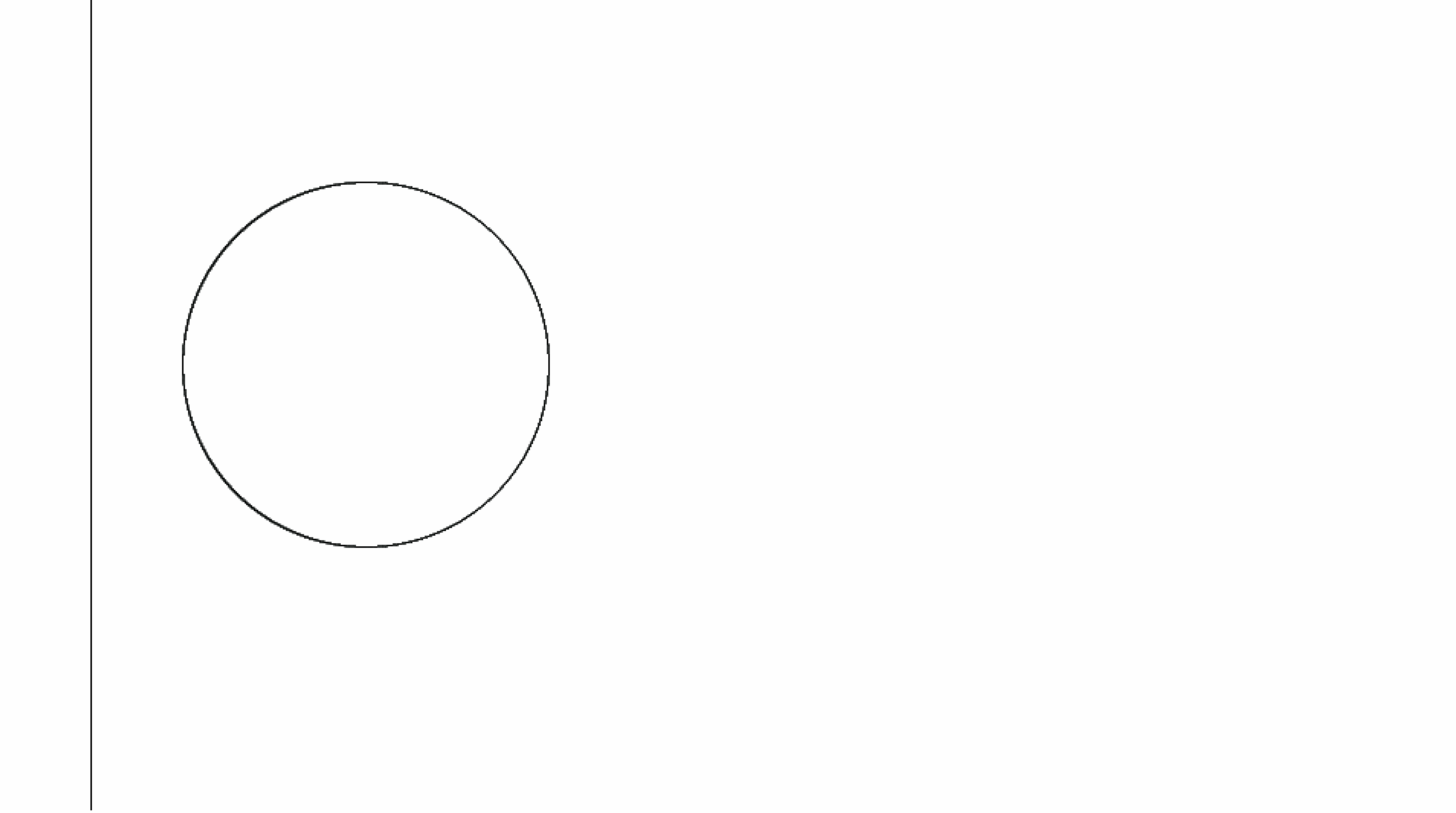
$$\begin{bmatrix} \rho \\ \rho u \\ \rho v \\ E \end{bmatrix}_t + \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho u v \\ u(E+p) \end{bmatrix}_x + \begin{bmatrix} \rho v \\ \rho u v \\ \rho v^2 + p \\ v(E+p) \end{bmatrix}_y = 0$$

ρ denotes density, u and v velocity in x- and y- directions, p pressure and E the total energy.

- The three dimensional

$$\begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ E \end{bmatrix}_t + \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho u v \\ \rho u w \\ u(E+p) \end{bmatrix}_x + \begin{bmatrix} \rho v \\ \rho u v \\ \rho v^2 + p \\ \rho v w \\ v(E+p) \end{bmatrix}_y + \begin{bmatrix} \rho w \\ \rho u w \\ \rho v w \\ \rho w^2 + p \\ w(E+p) \end{bmatrix}_z = \begin{bmatrix} 0 \\ 0 \\ 0 \\ g\rho \\ g\rho w \end{bmatrix}$$

Example: Interaction of a low-density bubble with a shock.



Speedup of 2D shock-bubble on NxN cells

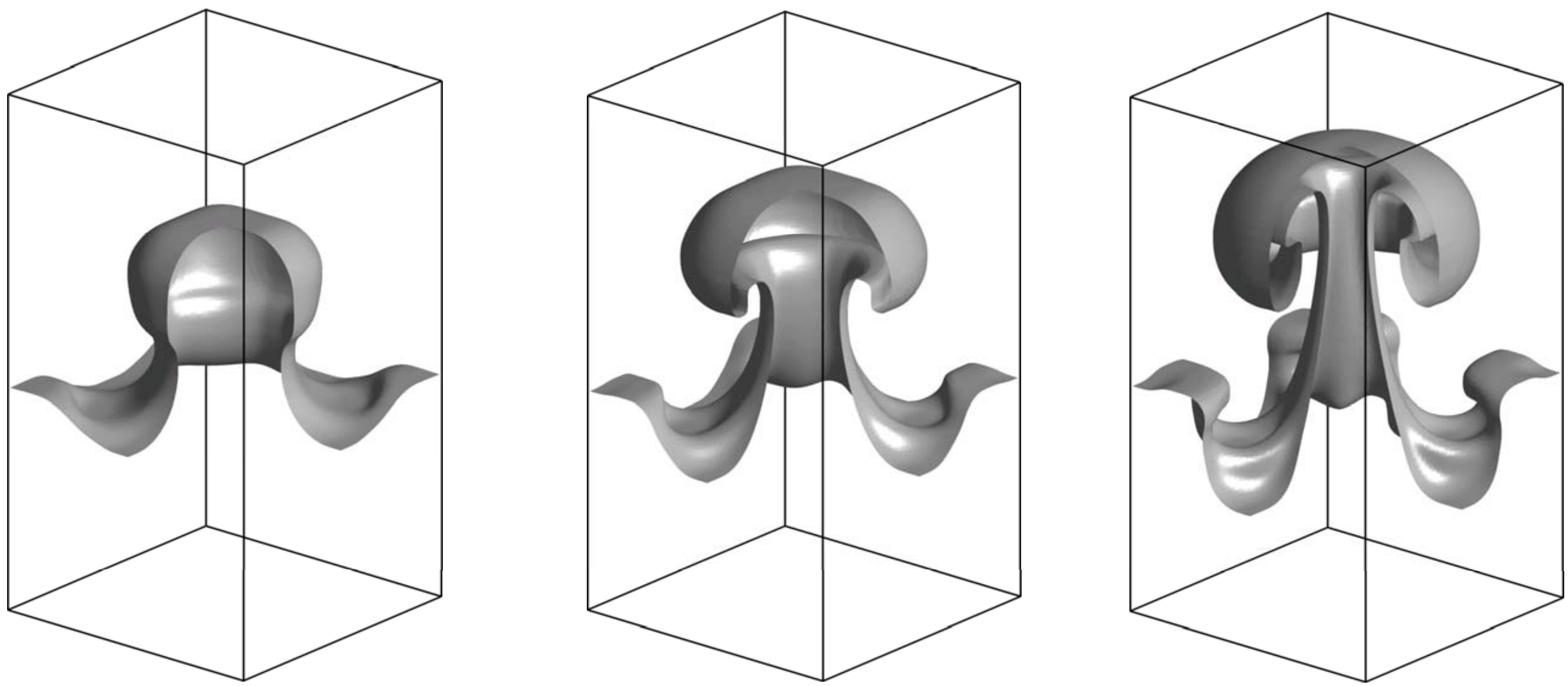
Bilinear reconstruction						
N	**Intel**	**6800**	**speedup**	**AMD**	**7800**	**speedup**
128	4.37e-2	3.70e-3	11.8	1.88e-2	1.38e-3	13.6
256	1.74e-1	8.69e-3	20.0	1.08e-1	4.37e-3	24.7
512	6.90e-1	3.32e-2	20.8	2.95e-1	1.72e-2	17.1
1024	2.95e-0	1.48e-1	19.9	1.26e-0	7.62e-2	16.5

CWENO reconstruction						
N	**Intel**	**6800**	**speedup**	**AMD**	**7800**	**speedup**
128	1.05e-1	1.22e-2	8.6	7.90e-2	4.60e-3	17.2
256	4.20e-1	4.99e-2	8.4	3.45e-1	1.74e-2	19.8
512	1.67e-0	1.78e-1	9.4	1.03e-0	6.86e-2	15.0
1024	6.67e-0	7.14e-1	9.3	4.32e-0	2.99e-1	14.4

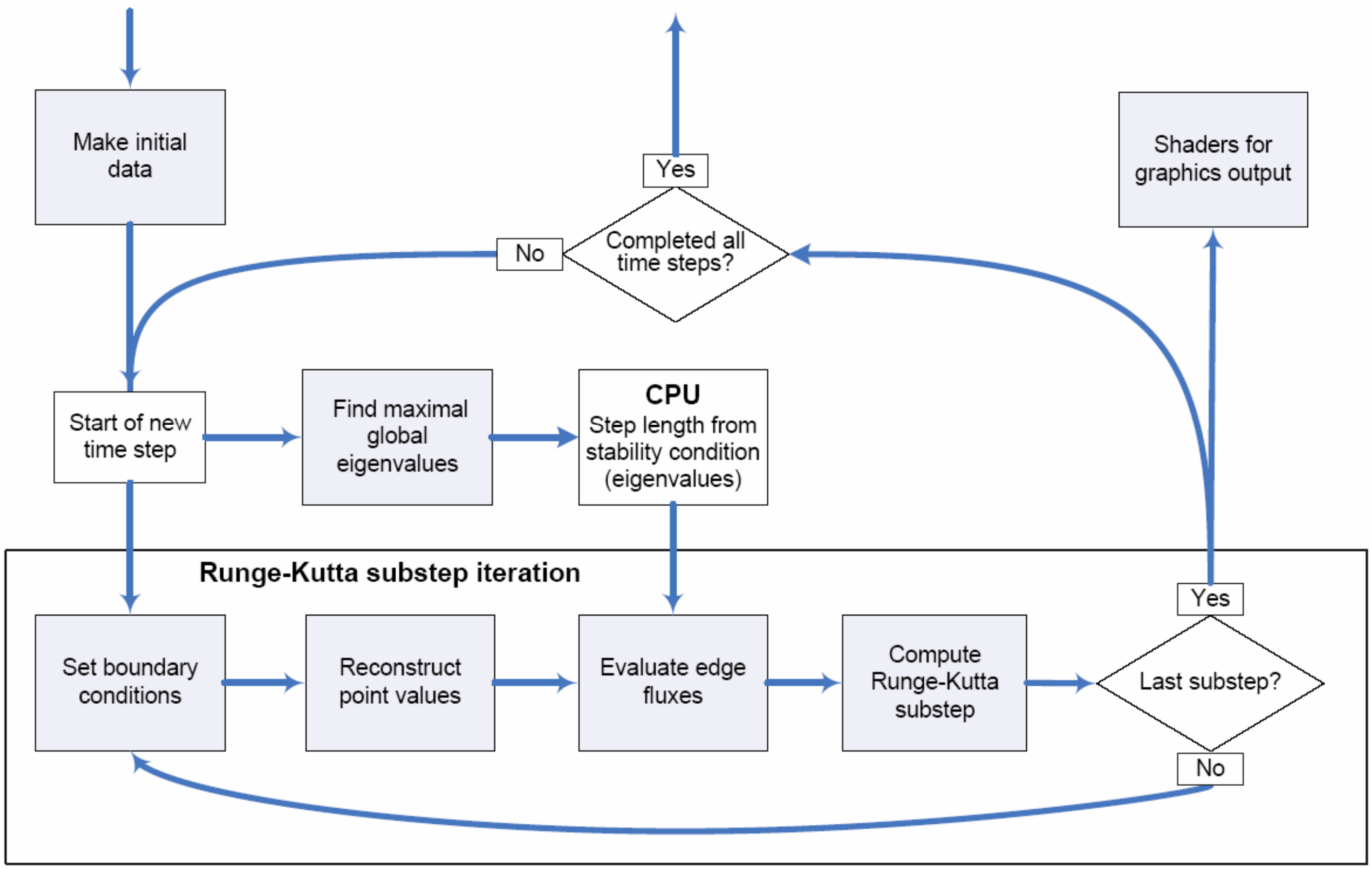
3D Rayleigh–Taylor Instability.

A layer of heavier fluid is placed on top of a lighter fluid and the heavier fluid is accelerated downwards by gravity.

N × N × N grid Average time per time STEP			
N	**AMD**	**7800**	**speedup**
49	5.23e-1	4.16e-2	12.6
64	1.14e-0	8.20e-2	13.9
81	1.98e-0	1.72e-1	11.5



Flow-chart for a GPU implementation of a semi-discrete, high resolution scheme.



Saturation equation

- The saturation equation models transport of two fluid-phases in a porous medium.

$$s_t + \nabla \cdot \big(f(s)(V + \lambda_0(s) g \Delta \rho) \big) = 0,$$

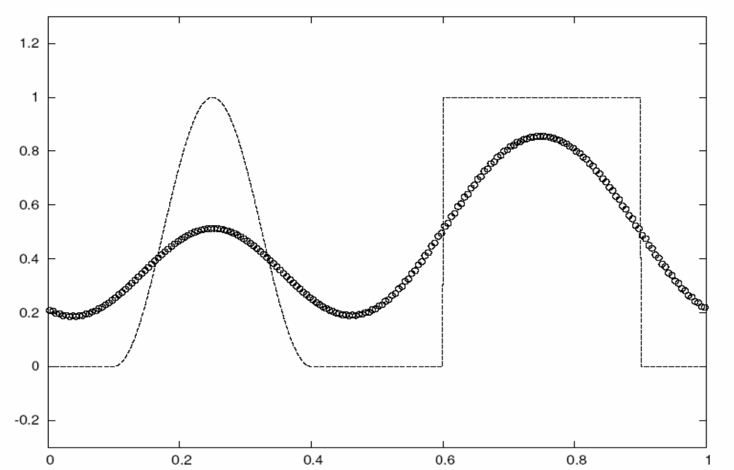
- Example: water injection into a fluvial reservoir filled with oil.

Water injection in a fluvial reservoir

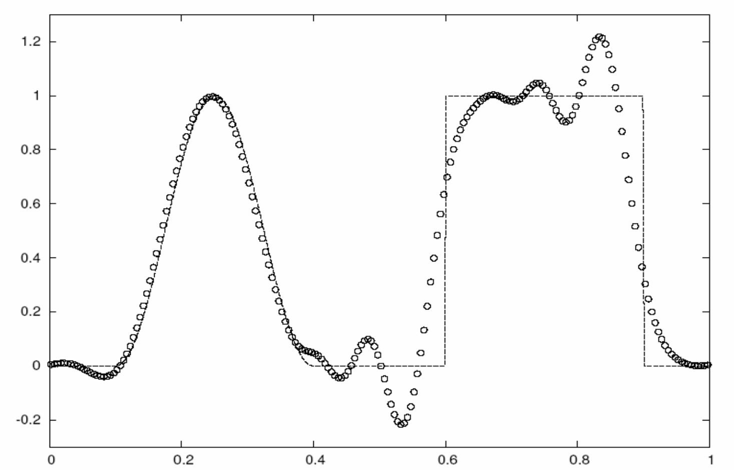
Linear Advection

- Models various transport phenomena of (passive) quantities. Simple equation, but difficult to compute solutions correctly.

1st order scheme: Lax-Friedrich



2nd order scheme: Lax-Wendroff

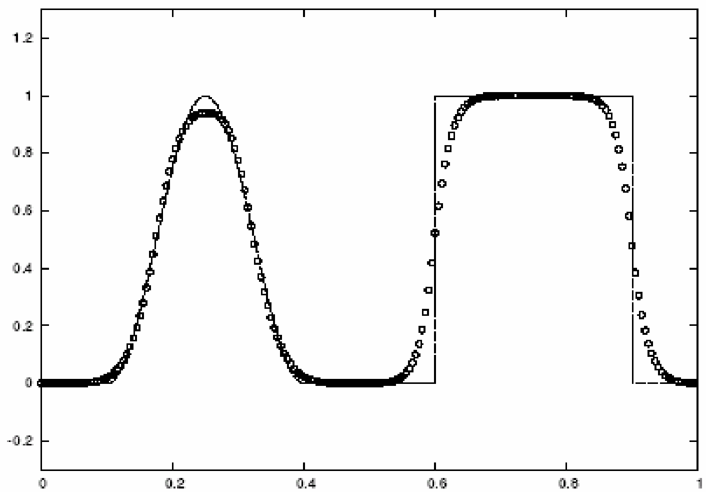


- Classical schemes: excessive smearing or spurious oscillations.
  → Need for more sophisticated schemes.

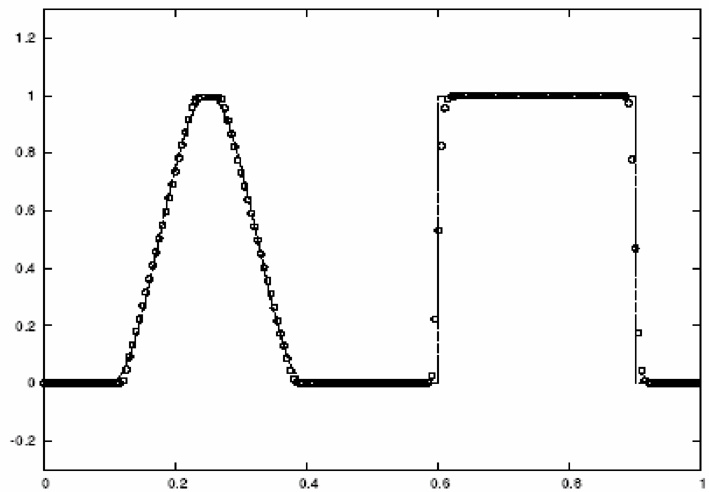
Linear Advection contd.

- Modern high-resolution schemes:

Dissipative



Compressive



- Higher-order approximation of smooth parts and no spurious oscillations at discontinuities.

Semi-Discrete High-Resolution Schemes

- Evolution of cell averages described by ODEs

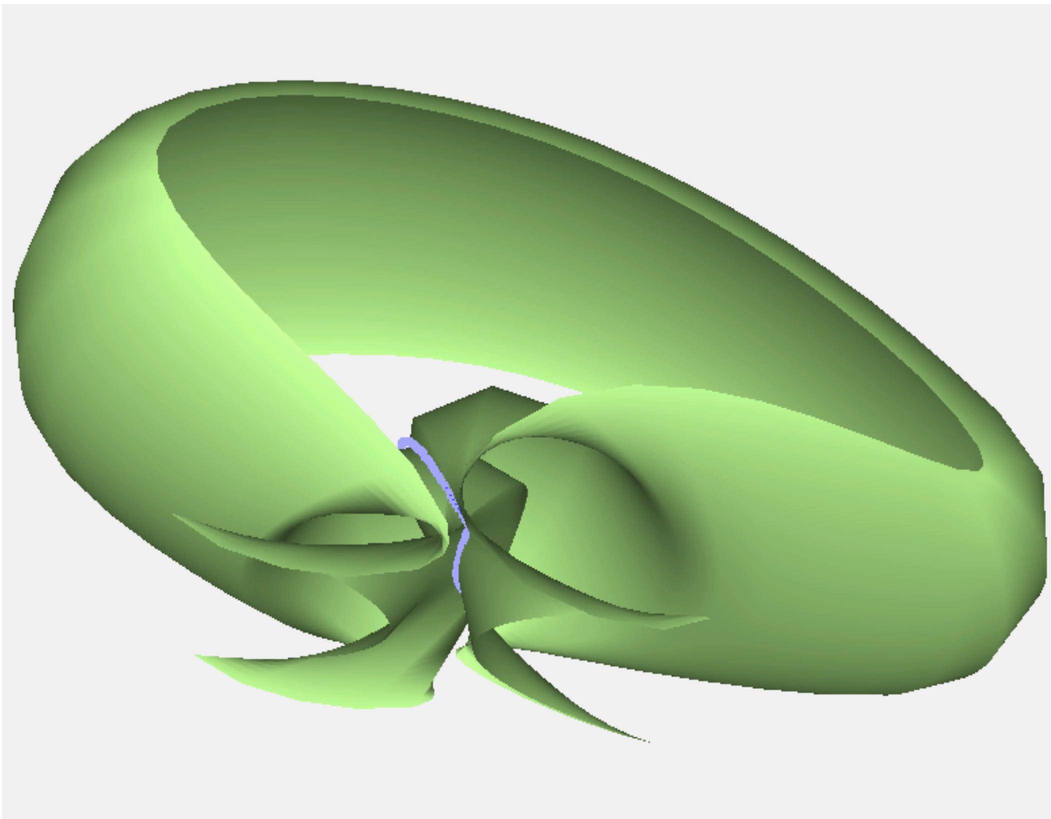
$$\frac{d}{dt} Q_{i,j}(t) = \frac{\left(F_{i-1/2,j}(t) - F_{i+1/2,j}(t)\right)}{\Delta x} + \frac{\left(G_{i,j-1/2}(t) - G_{i,j+1/2}(t)\right)}{\Delta y}$$

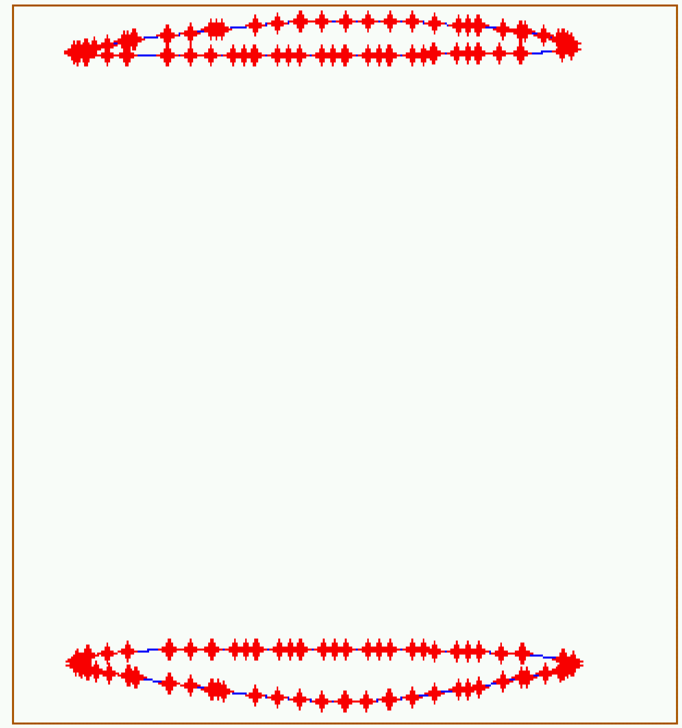
- Steps in the algorithms:
  - Reconstruction of piecewise polynomials from cell averages
  - Evaluation of reconstruction at integration points
  - Numerical computation of edge fluxes
  - Evolution by Runge-Kutta scheme

CAD Intersections on the GPU

- Use the GPU to find if difficult intersection configurations are present and create conjectures on the intersection configuration. (SINTEF has applied for patent)
  - Surface intersection
    - check for loops
    - check for singular intersections
  - Surface self-intersection
    - Is a self-intersection possible?
    - Global self-intersection? Two different parts of the surface intersect
    - Local self-intersection loop?
    - Ridges with vanishing or near vanish surface normal possibly combined with self-intersections.
- The GPU can help to determine if the self-intersection needs advanced intersection algorithms or simpler approaches can be used.

Global Self-intersection

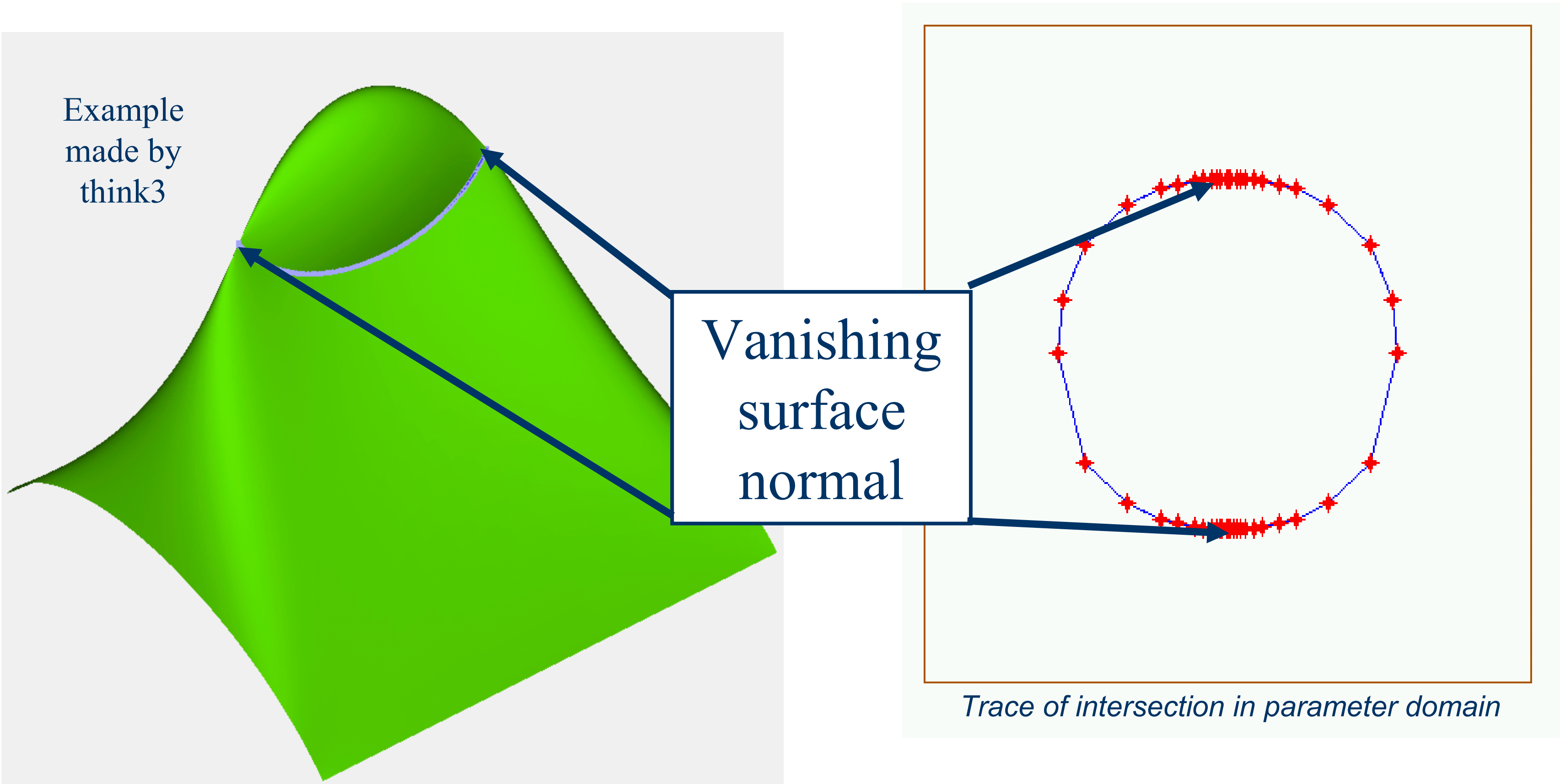




Trace of intersection in parameter domain

Two different parts of the surface intersect. Proper subdivision change the problem to a transversal intersections between two surfaces.

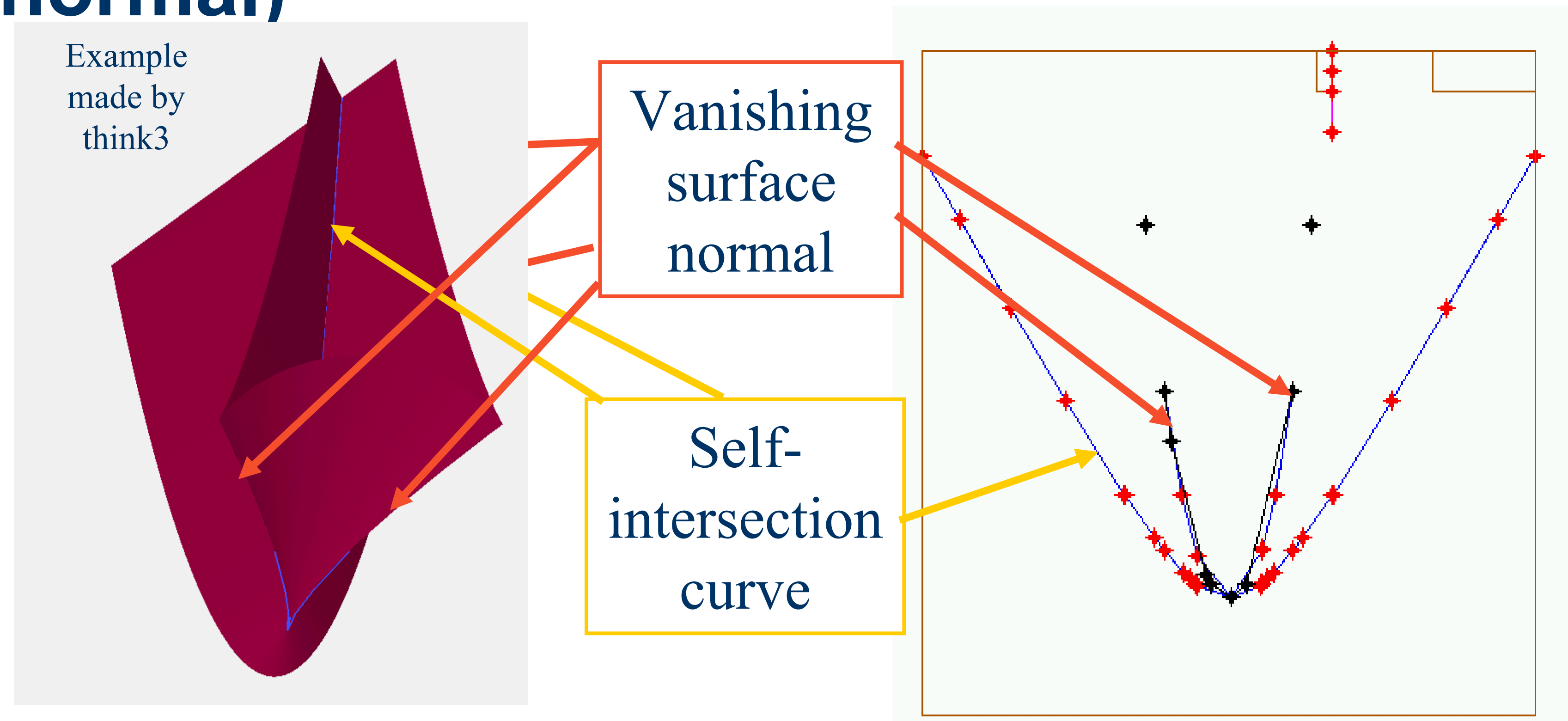
Local self-intersection



Trace of intersection in parameter domain

The self-intersection appears as a small loop, with two singular points. In these the surface normal vanishes. Proper subdivision will change the problem to a near transversal intersection problem.

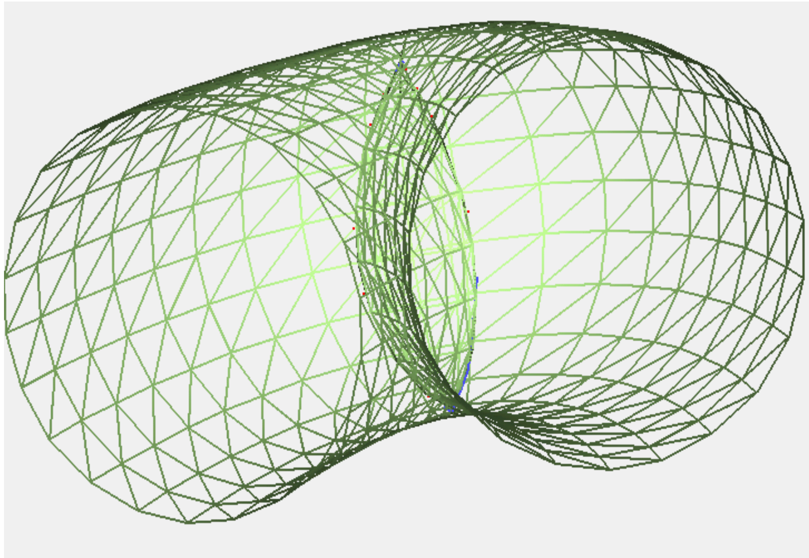
Self-intersection with ridges (Vanishing normal)

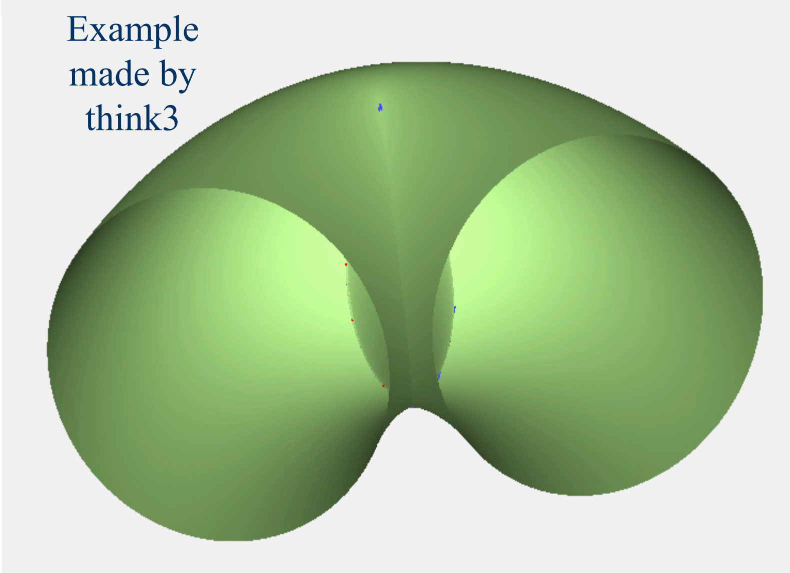


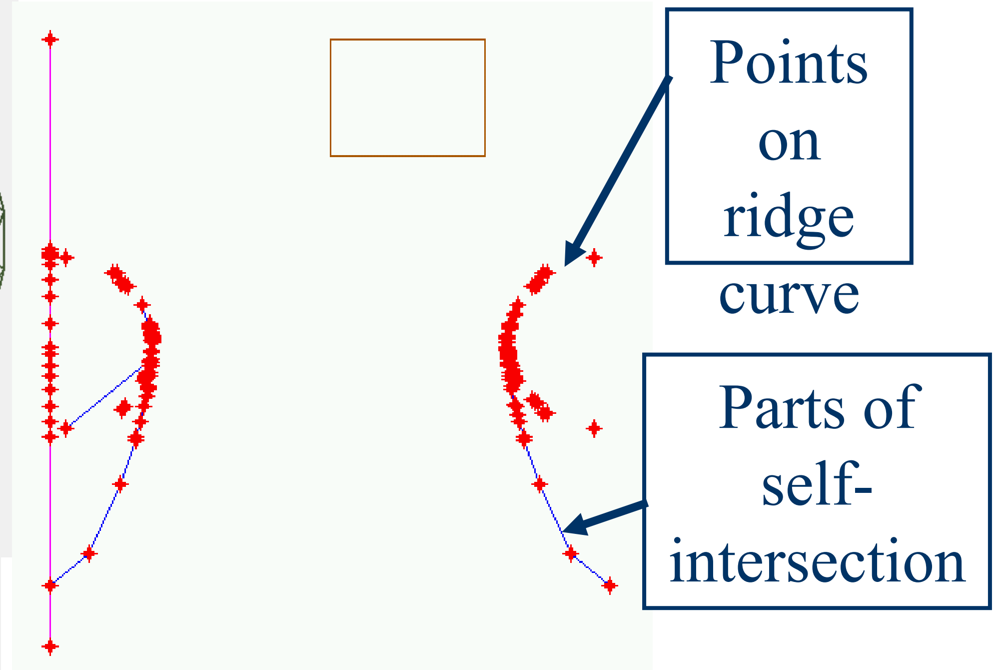
Partial trace of intersection in parameter domain

- The ridges do not in general follow constant parameter lines
- Typical for offset surfaces, duct type surfaces and draft-angle surface cannot be converted to a near-transversal intersection.

Ridges in self-intersection – Pipe surface



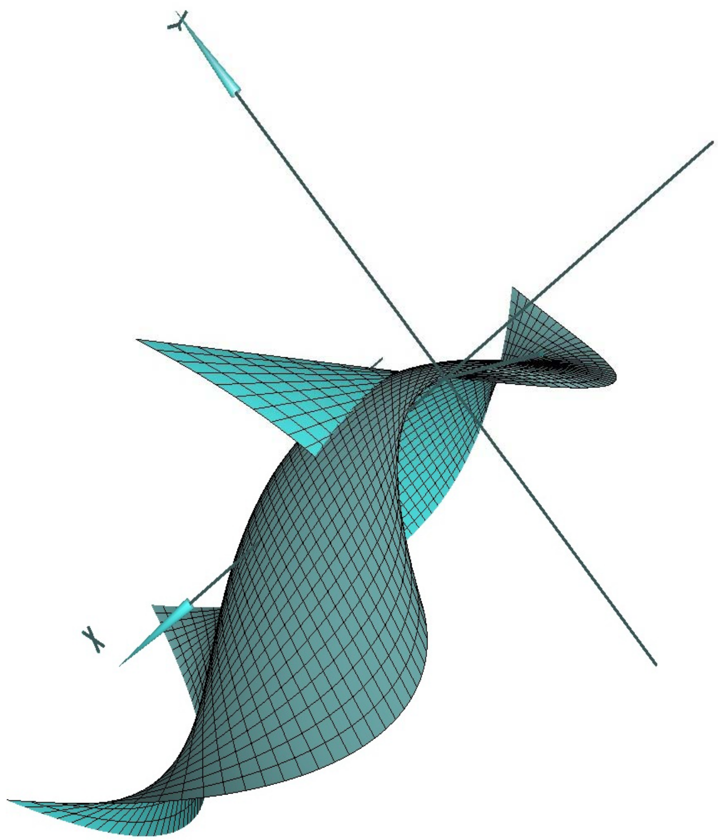


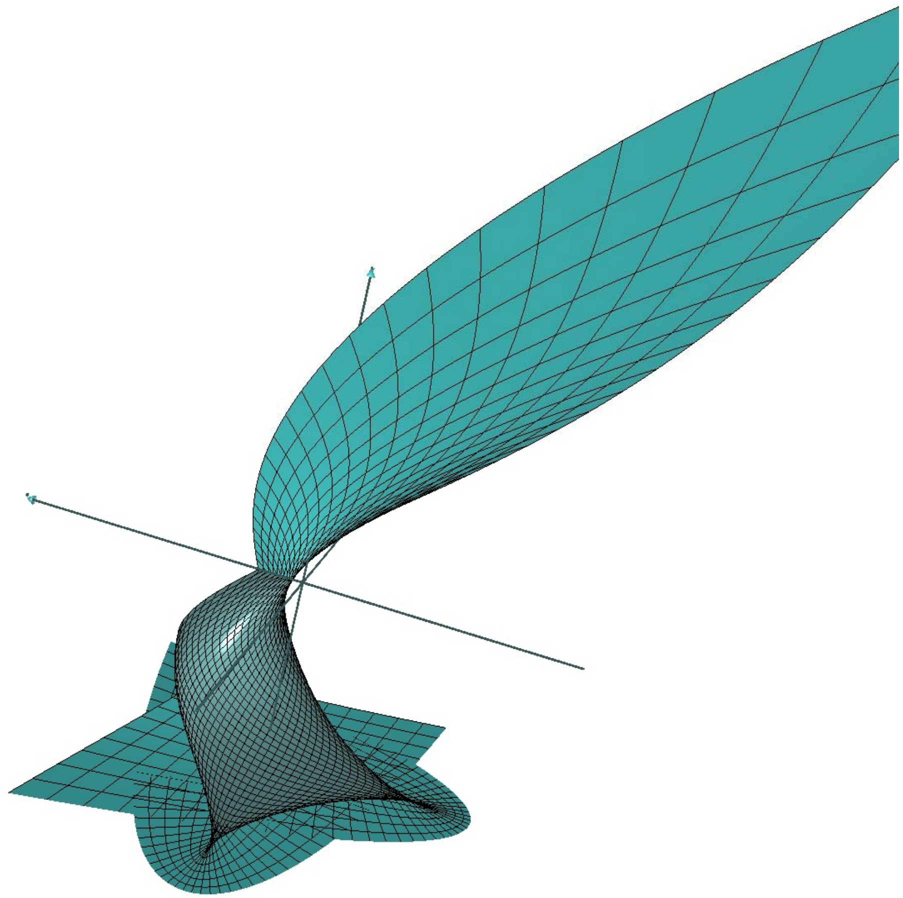


Partial trace of intersection in parameter domain

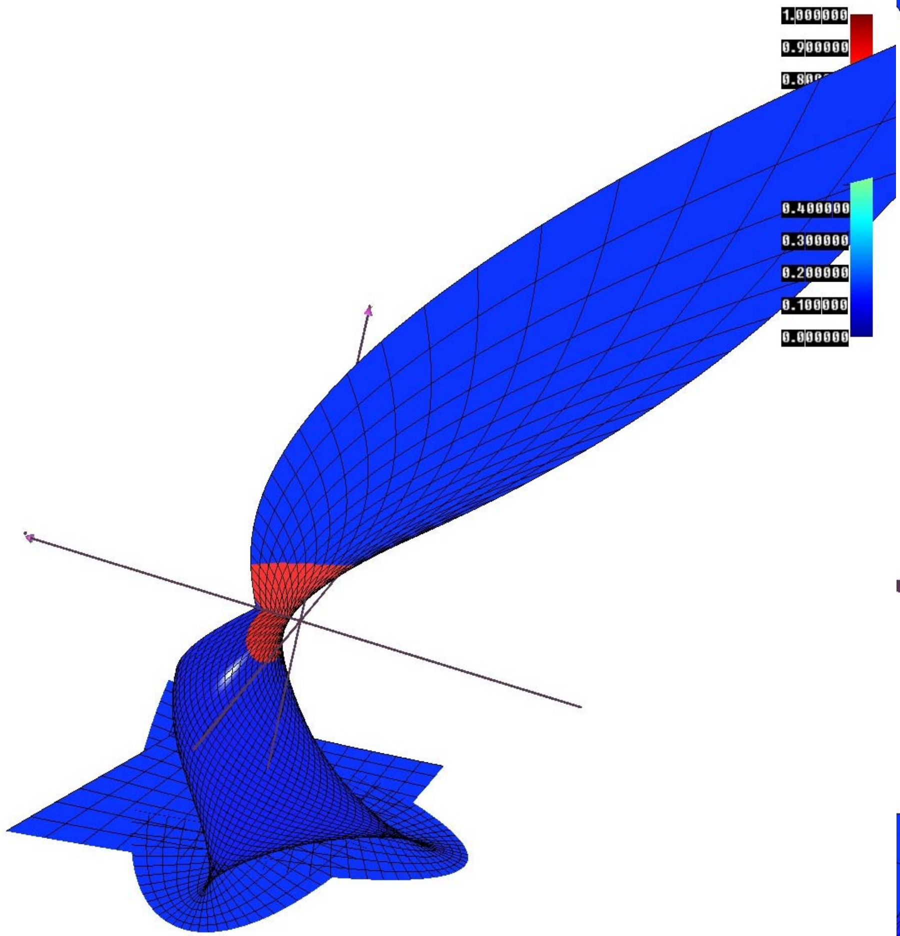
Computationally expensive to solve by recursive subdivision.

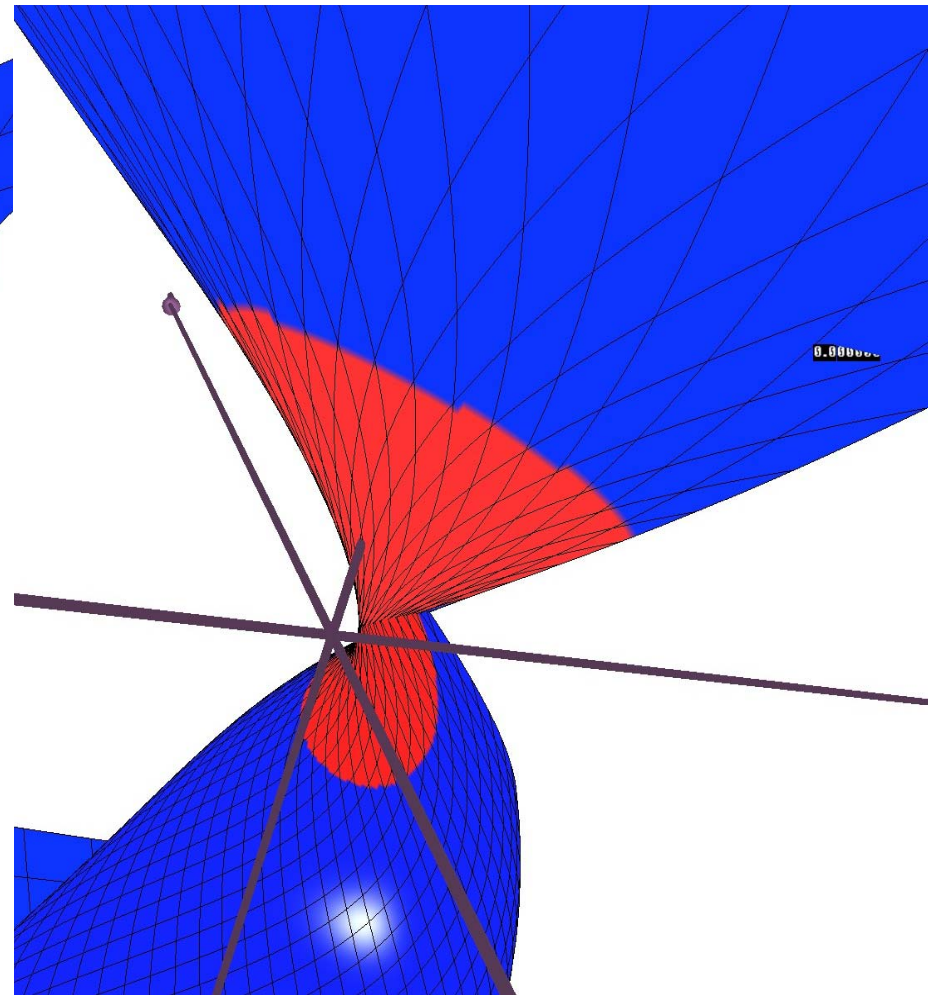
A surface (cubic) & normal surface (quintic)



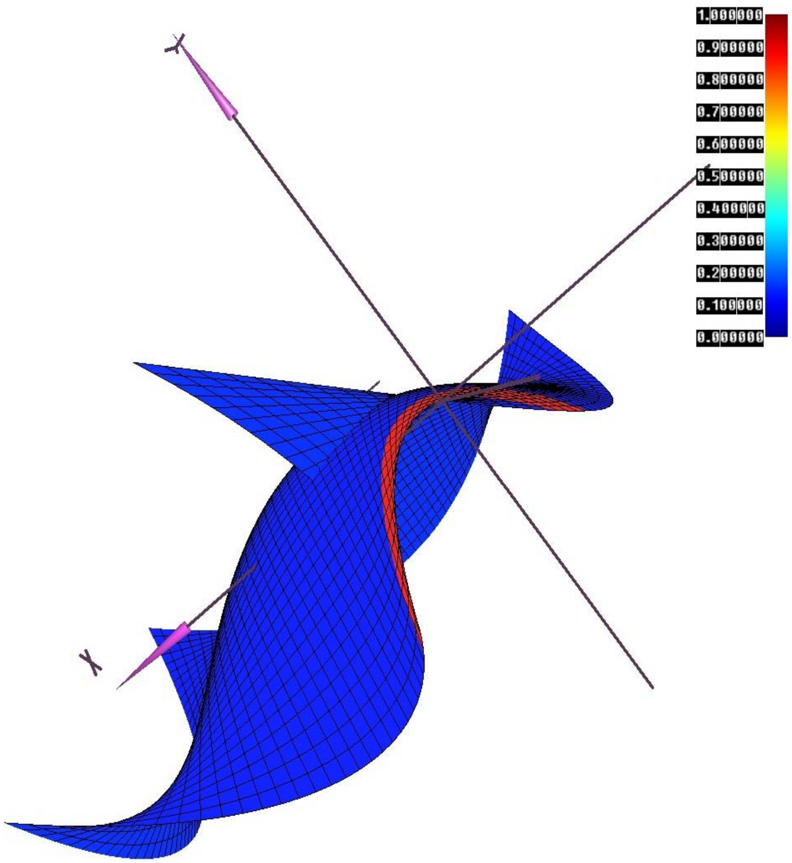


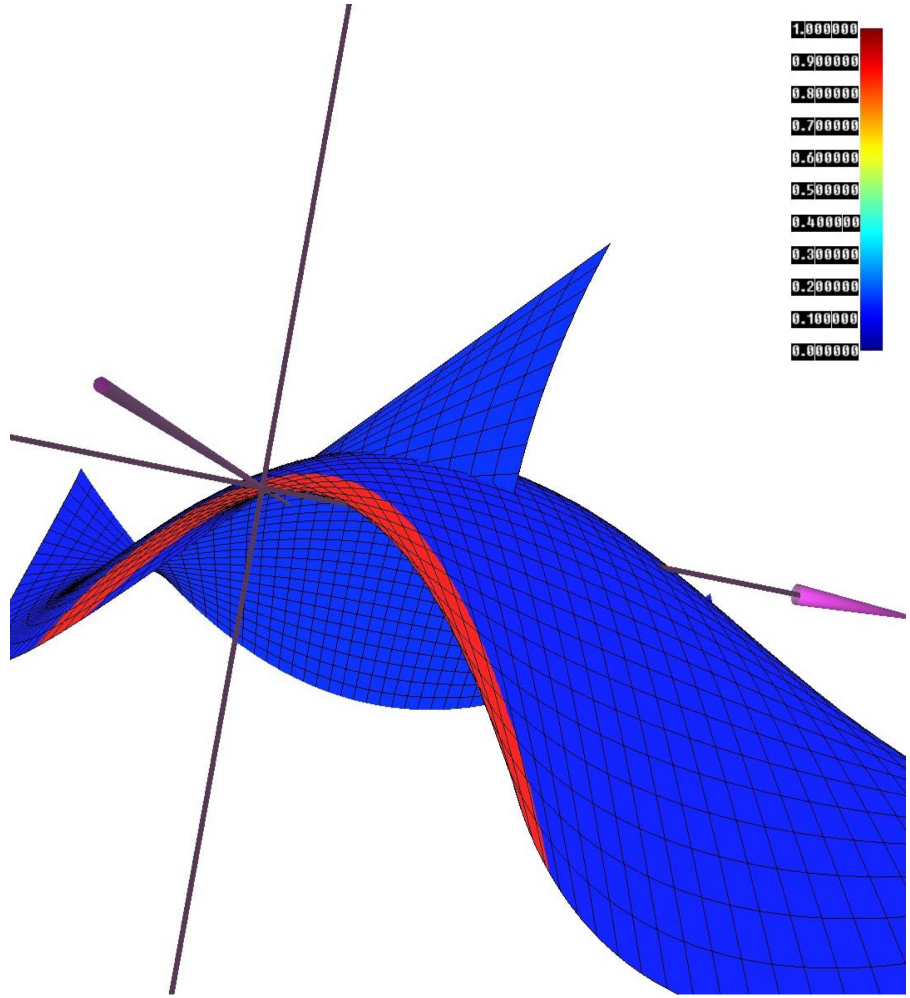
GPU calculated and visualized points on normal surface close to the origin





The regions with near vanishing normal visualized as texture on original surface





Observations

- Most surfaces do not have self-intersection
  - Important to identify most surfaces without self-intersections as early as possible
  - Moderate simplistic subdivision makes the surface and normal cone boxes smaller $O(h^2)$ convergence, and will classify many possible self-intersections as not self-intersecting
- For surface with no vanishing surface normal all self-intersection curves intersect a boundary
- Vanishing surface normal identifies regions with more complex self-intersection topology
  - Moderate simplistic subdivision reduce the size of such regions, and allows to focus a more complex self-intersection algorithms on surface sub-regions.

Comparison of CPU and GPU implementation of simplistic subdivision

Initial process for surface self-intersection

- Subdivision of a bicubic Bezier patch and (quintic) patch representing surface normals into $2^n \times 2^n$ subpatches
- Test for degenerate normals for subpatches
- Computation of the approximate normal cones for subpatches
- Computation of the approximate bounding boxes for subpatches
- Computation of bounding box pair intersections for subpatches

n	Grid	GPU	CPU	Speedup
4	16×16	7.456e-03	6.831e-03	0.9
5	32×32	1.138e-02	7.330e-02	6.4
6	64×64	7.271e-02	1.043e00	14.3
7	128×128	9.573e-01	1.607e01	16.8
8	256×256	1.515e01	2.555e02	16.9

CPU/GPU approach

- For n<5 use CPU
- Refined checks use GPU
- For problems not sorted out use recursive subdivision approach on CPU, or refined subdivision on GPU

Where are we now, and where do we go?

- During the first year (2004) of the project we got experiences on the use of the GPU, and the sort of problems best suited for the GPU
- We try to make the potential of the GPU understandable to personnel outside of computer graphics
  - All concepts related to the GPU has a strong computer graphics flavor
- We will continue investigation on
  - Partial differential equations
  - Geometry problems - intersection
  - Image processing
  - Linear algebra