Interactive Methods in Scientific Visualization

SCI INSTITUTE

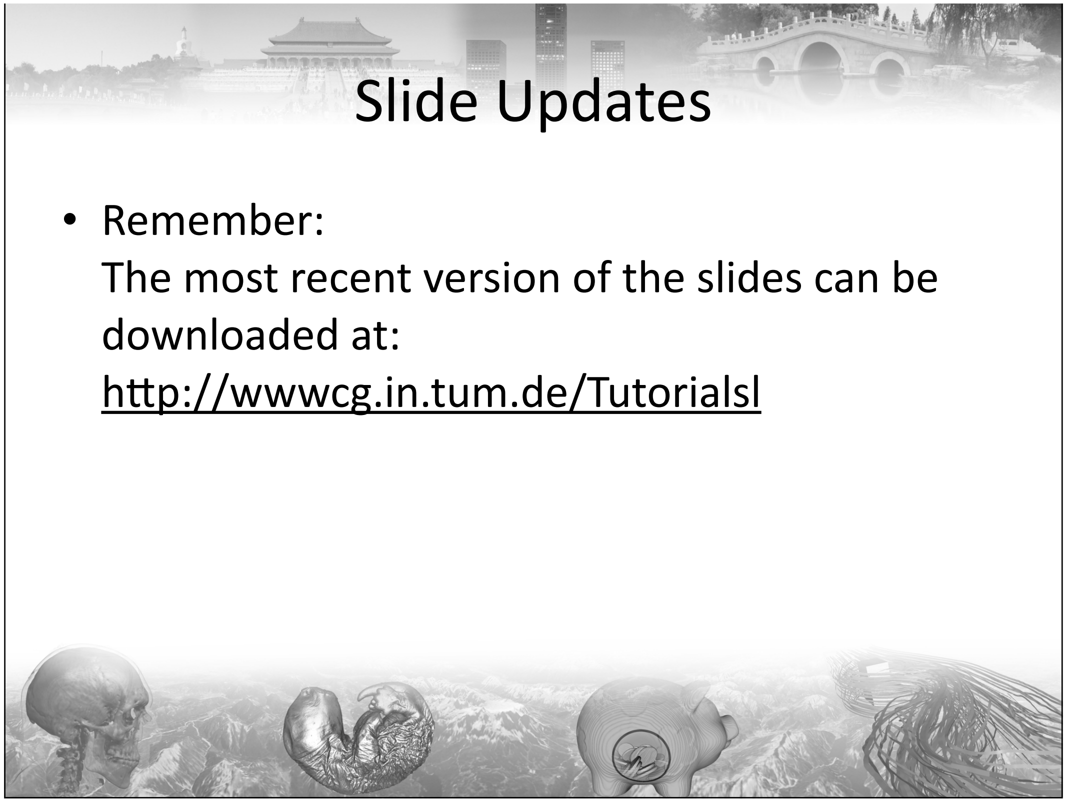
CG

tum.3D computer graphics & visualization

Part 2:
Vector Field Visualization

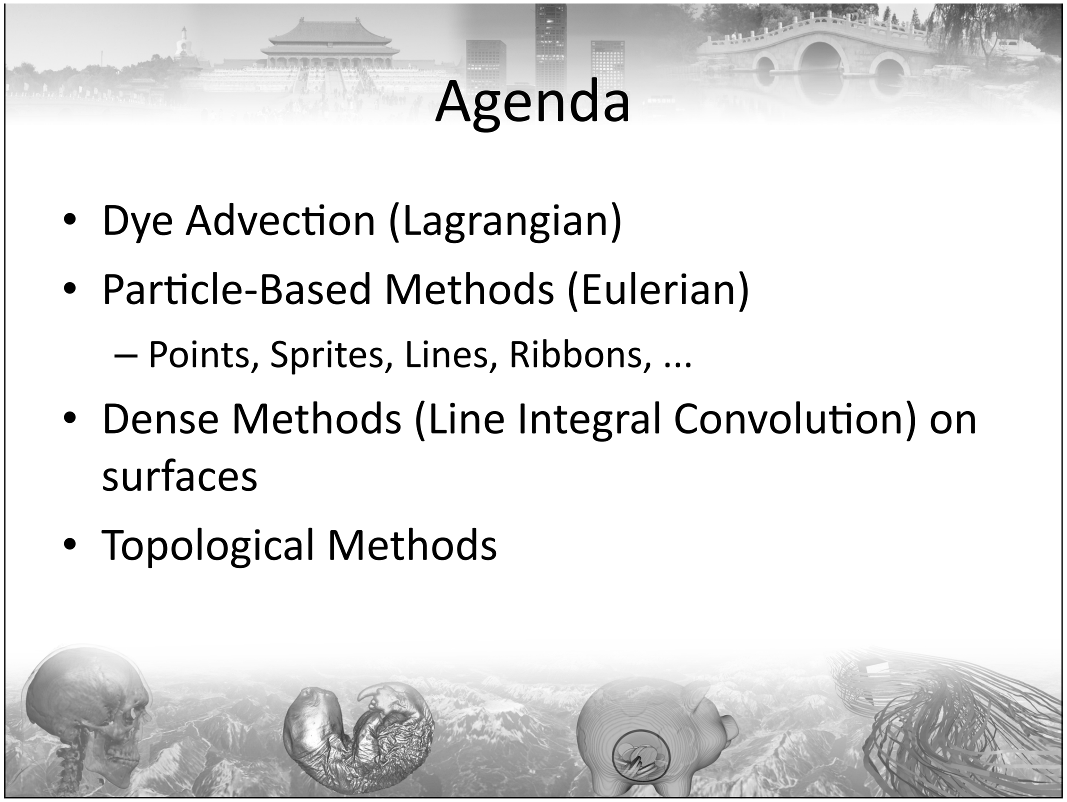
Jens Krüger

SCI INSTITUTE



Slide Updates

- Remember:
  The most recent version of the slides can be downloaded at:
  http://wwwcg.in.tum.de/Tutorialsl



Agenda

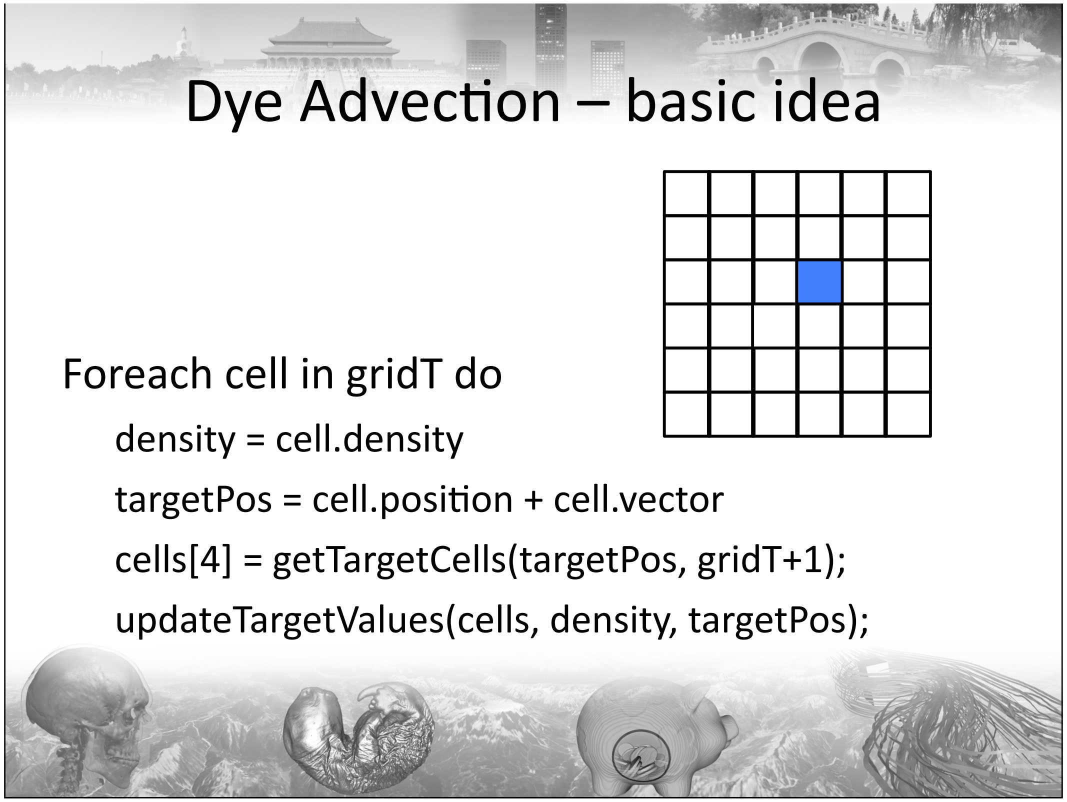
- Dye Advection (Lagrangian)
- Particle-Based Methods (Eulerian)
  - Points, Sprites, Lines, Ribbons, ...
- Dense Methods (Line Integral Convolution) on surfaces
- Topological Methods

This talk will cover a set of approaches to interactively explore flow fields (given on regular grids, while the methods themselves also work for unstructured grids but for the sake of simplicity most of the implementations are given for regular grids)

Dye Advection

NVIDIA SDK http://developer.nvidia.com/page/home.html

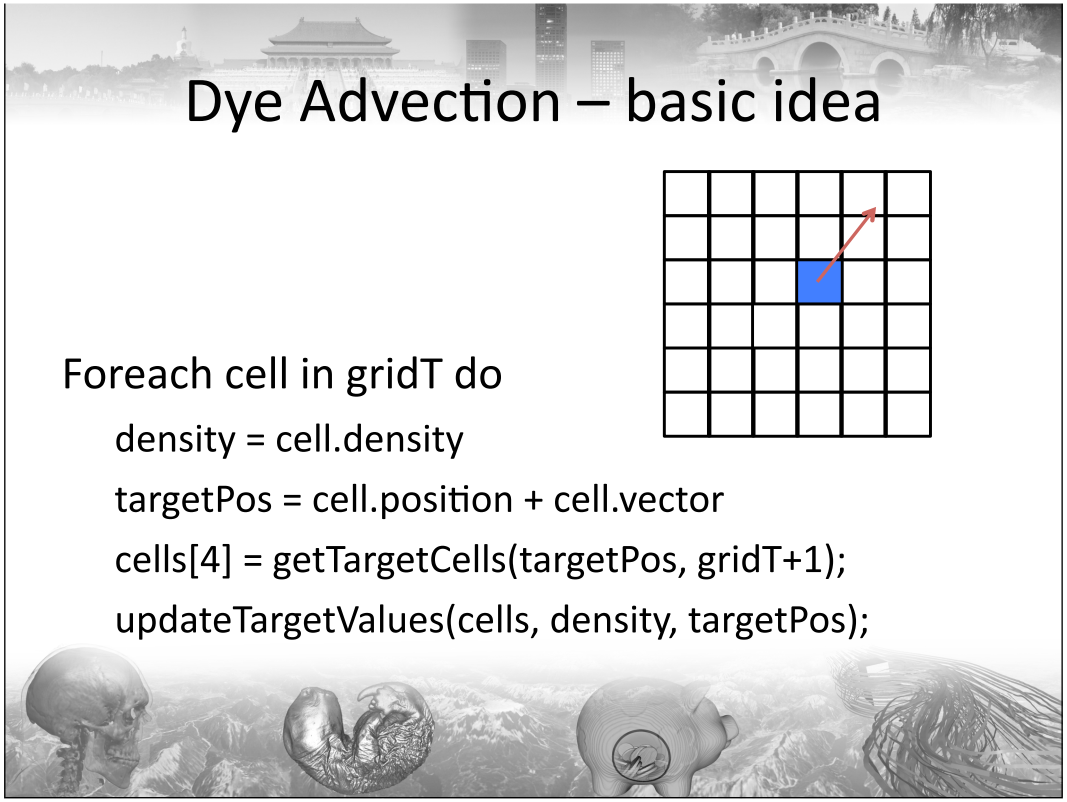
The first method in this talk is the dye advection, that simulates continuous smoke or dye in a flow field.



Dye Advection – basic idea

```
Foreach cell in gridT do
    density = cell.density
    targetPos = cell.position + cell.vector
    cells[4] = getTargetCells(targetPos, gridT+1);
    updateTargetValues(cells, density, targetPos);
```

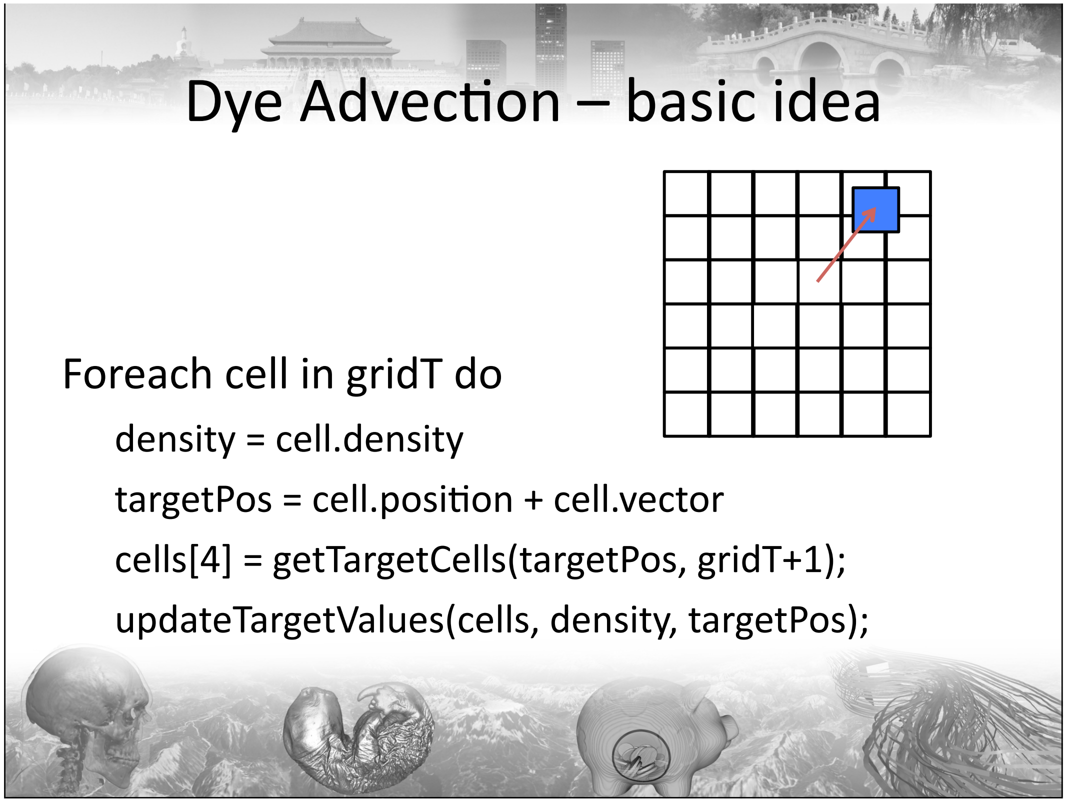
For this method we discretize the domain (for the sake of explanation we use a 2D grid but the method extends trivial to higher dimensions). In this grid we visit every cell in ever iteration ad move (advect) it‘s contentes along the flow. Note that the updateTargetValues call requires some interpolation or splating as we will in general not hit a single taget cell right in the center.



Dye Advection – basic idea

```
Foreach cell in gridT do
    density = cell.density
    targetPos = cell.position + cell.vector
    cells[4] = getTargetCells(targetPos, gridT+1);
    updateTargetValues(cells, density, targetPos);
```

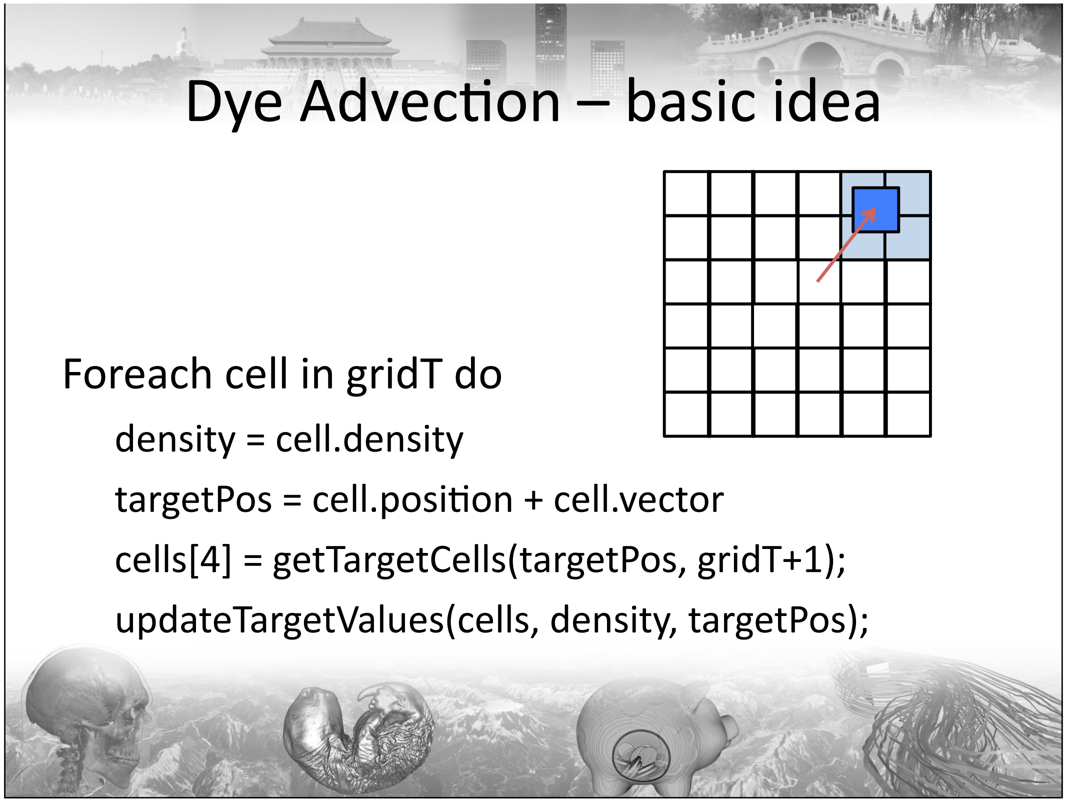
For this method we discretize the domain (for the sake of explanation we use a 2D grid but the method extends trivial to higher dimensions). In this grid we visit every cell in ever iteration ad move (advect) it‘s contentes along the flow. Note that the updateTargetValues call requires some interpolation or splating as we will in general not hit a single taget cell right in the center.



Dye Advection – basic idea

```
Foreach cell in gridT do
    density = cell.density
    targetPos = cell.position + cell.vector
    cells[4] = getTargetCells(targetPos, gridT+1);
    updateTargetValues(cells, density, targetPos);
```

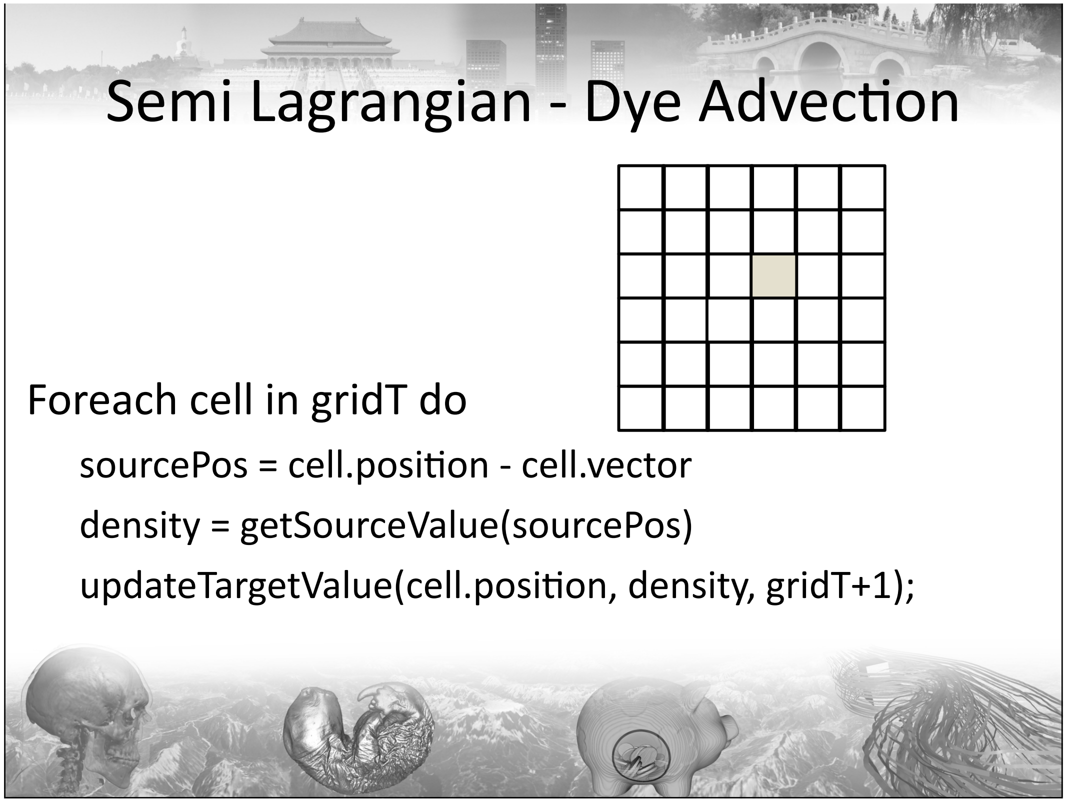
For this method we discretize the domain (for the sake of explanation we use a 2D grid but the method extends trivial to higher dimensions). In this grid we visit every cell in ever iteration ad move (advect) it‘s contentes along the flow. Note that the updateTargetValues call requires some interpolation or splating as we will in general not hit a single taget cell right in the center.



Dye Advection – basic idea

```
Foreach cell in gridT do
    density = cell.density
    targetPos = cell.position + cell.vector
    cells[4] = getTargetCells(targetPos, gridT+1);
    updateTargetValues(cells, density, targetPos);
```

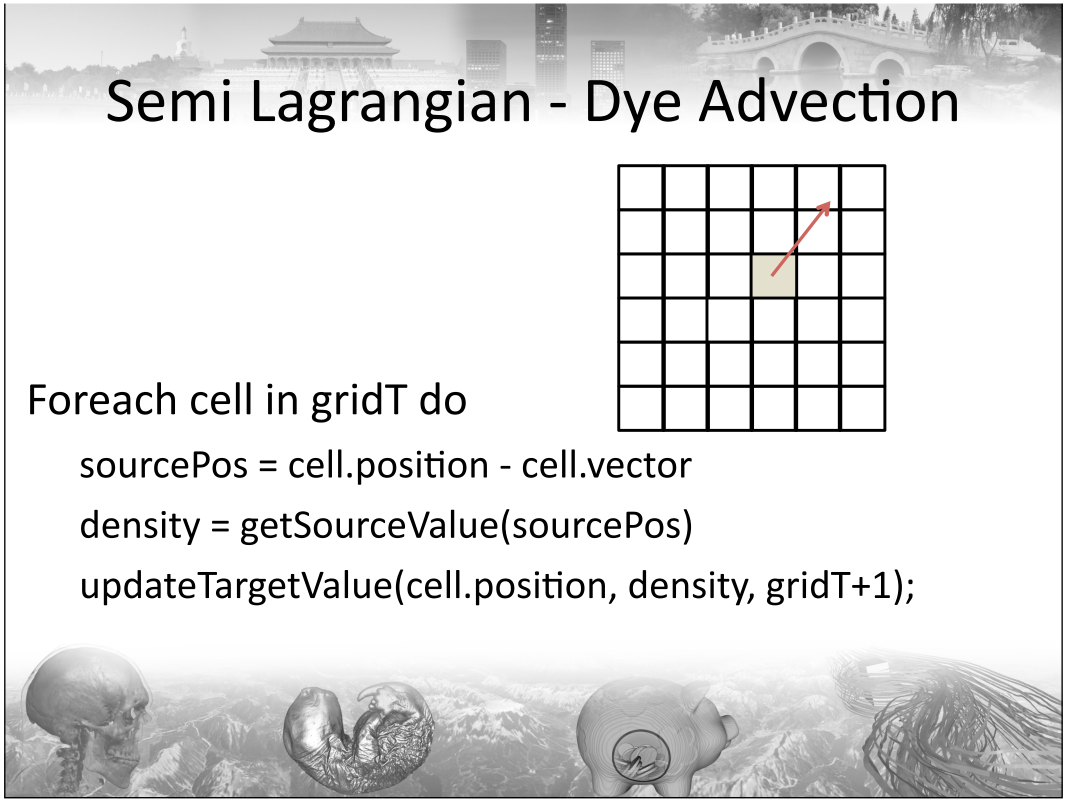
For this method we discretize the domain (for the sake of explanation we use a 2D grid but the method extends trivial to higher dimensions). In this grid we visit every cell in ever iteration ad move (advect) it‘s contentes along the flow. Note that the updateTargetValues call requires some interpolation or splating as we will in general not hit a single taget cell right in the center.



Semi Lagrangian - Dye Advection

```
Foreach cell in gridT do
    sourcePos = cell.position - cell.vector
    density = getSourceValue(sourcePos)
    updateTargetValue(cell.position, density, gridT+1);
```

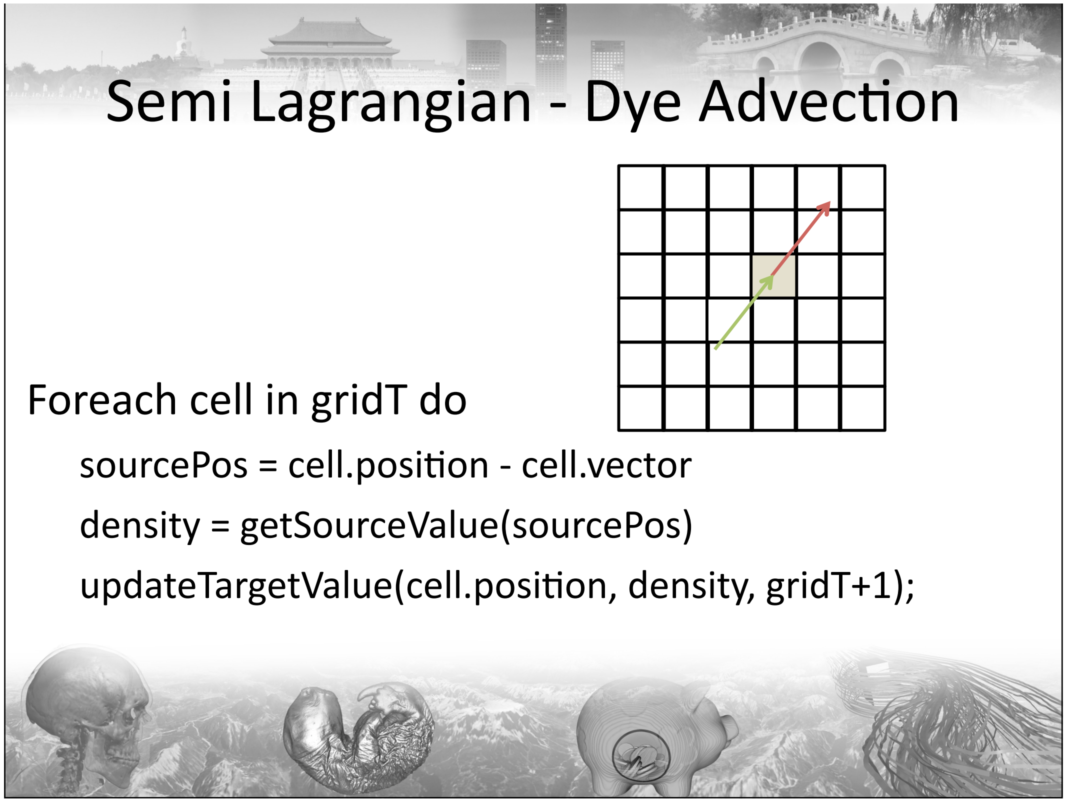
To make this method more cache efficient (and also easier to implement on a GPU) we use the sem-lagrangian approach. In this approach instead of splatting or pushing the dye from the current cell to its target cells we look backwards in time and pull dye to the current cell.



Semi Lagrangian - Dye Advection

```
Foreach cell in gridT do
    sourcePos = cell.position - cell.vector
    density = getSourceValue(sourcePos)
    updateTargetValue(cell.position, density, gridT+1);
```

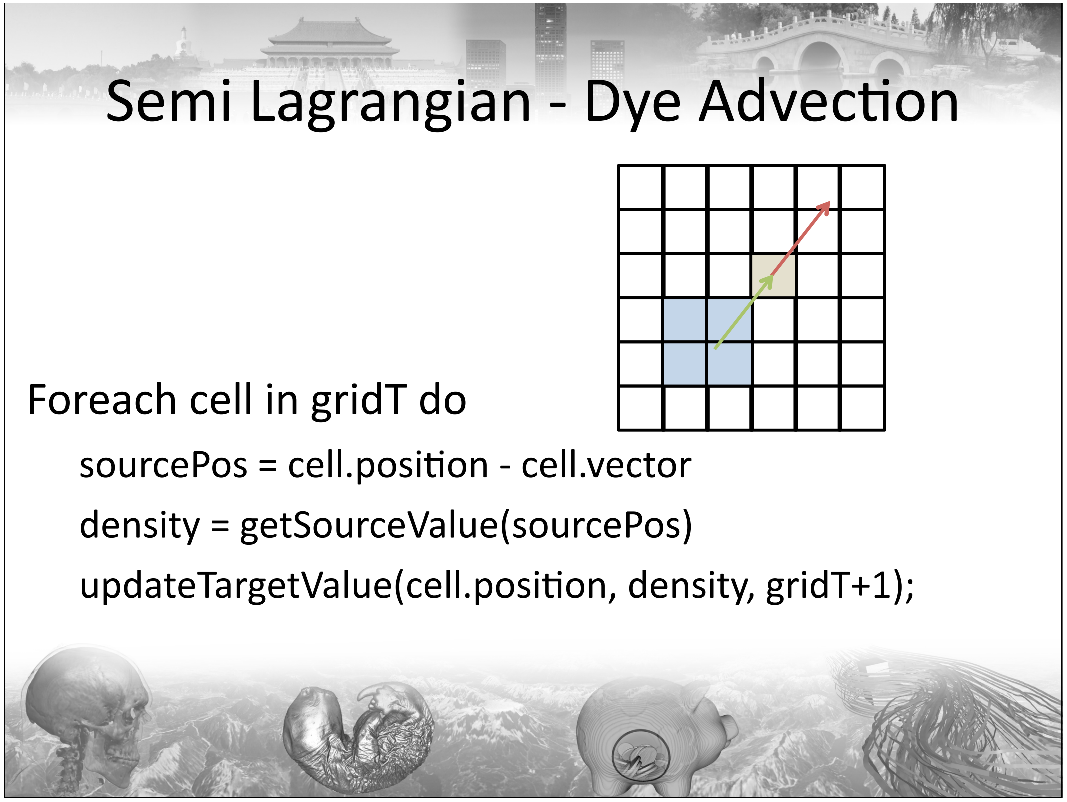
To make this method more cache efficient (and also easier to implement on a GPU) we use the sem-lagrangian approach. In this approach instead of splatting or pushing the dye from the current cell to its target cells we look backwards in time and pull dye to the current cell.



Semi Lagrangian - Dye Advection

```
Foreach cell in gridT do
    sourcePos = cell.position - cell.vector
    density = getSourceValue(sourcePos)
    updateTargetValue(cell.position, density, gridT+1);
```

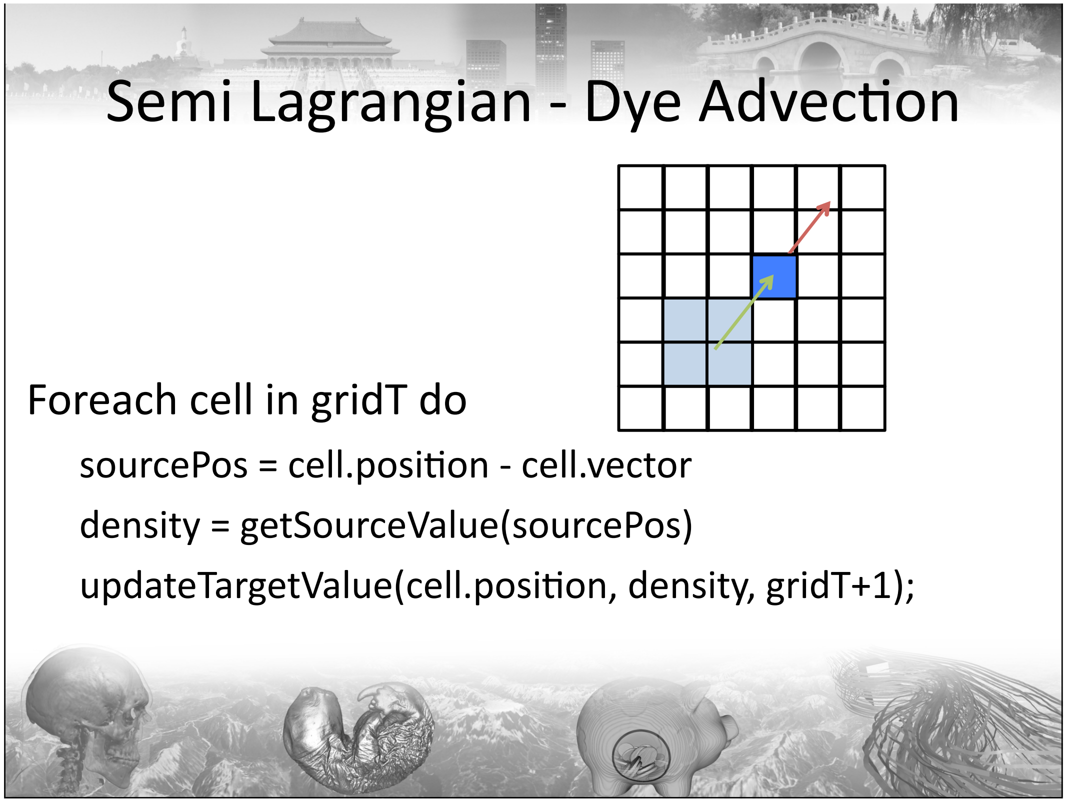
To make this method more cache efficient (and also easier to implement on a GPU) we use the sem-lagrangian approach. In this approach instead of splatting or pushing the dye from the current cell to its target cells we look backwards in time and pull dye to the current cell.



Semi Lagrangian - Dye Advection

```
Foreach cell in gridT do
    sourcePos = cell.position - cell.vector
    density = getSourceValue(sourcePos)
    updateTargetValue(cell.position, density, gridT+1);
```

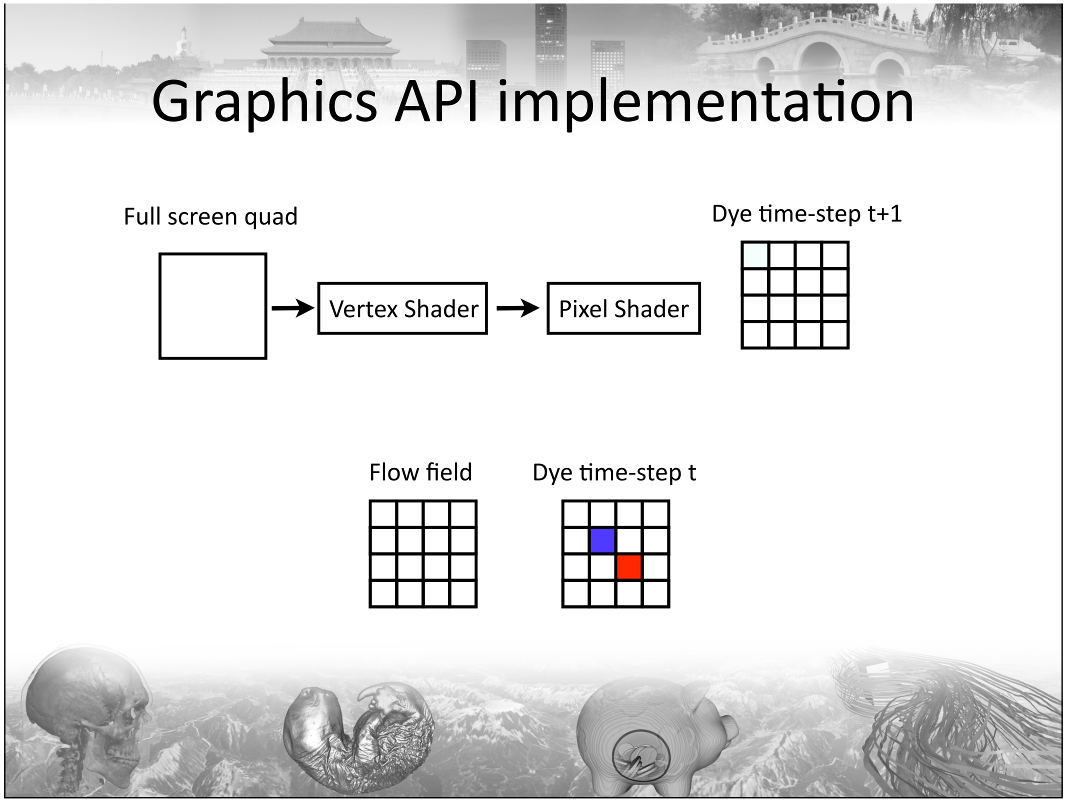
To make this method more cache efficient (and also easier to implement on a GPU) we use the sem-lagrangian approach. In this approach instead of splatting or pushing the dye from the current cell to its target cells we look backwards in time and pull dye to the current cell.



Semi Lagrangian - Dye Advection

```
Foreach cell in gridT do
    sourcePos = cell.position - cell.vector
    density = getSourceValue(sourcePos)
    updateTargetValue(cell.position, density, gridT+1);
```

To make this method more cache efficient (and also easier to implement on a GPU) we use the sem-lagrangian approach. In this approach instead of splatting or pushing the dye from the current cell to its target cells we look backwards in time and pull dye to the current cell.



Graphics API implementation

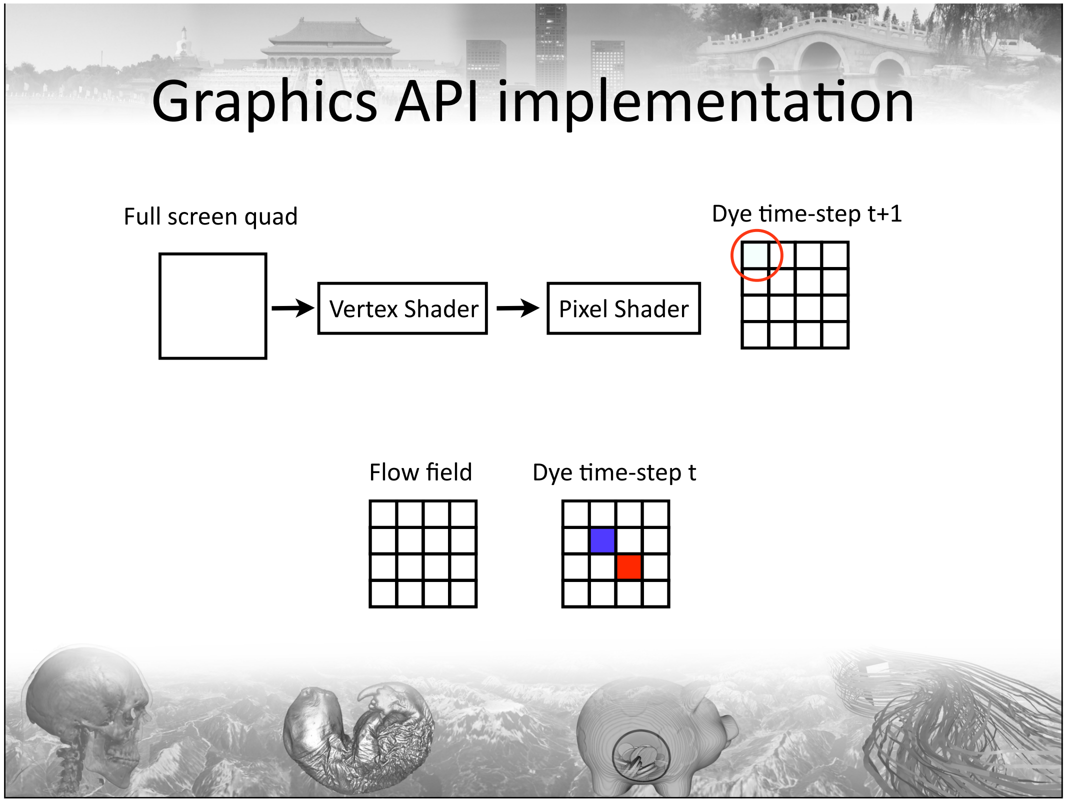
Full screen quad → Vertex Shader → Pixel Shader

Dye time-step t+1

Flow field

Dye time-step t

In particluar the semi-lagrangian approach is very easy to implement on the GPU in 2D. In every time step a quad is rendered that covers the entire domain, in the pixel shader for ever fragment (=texel) the flow field is fetched, the flow vector is added to the current texture coordinate and with this the last time-step‘s dye is fetched and stored at the current position.



Graphics API implementation

Full screen quad

Vertex Shader

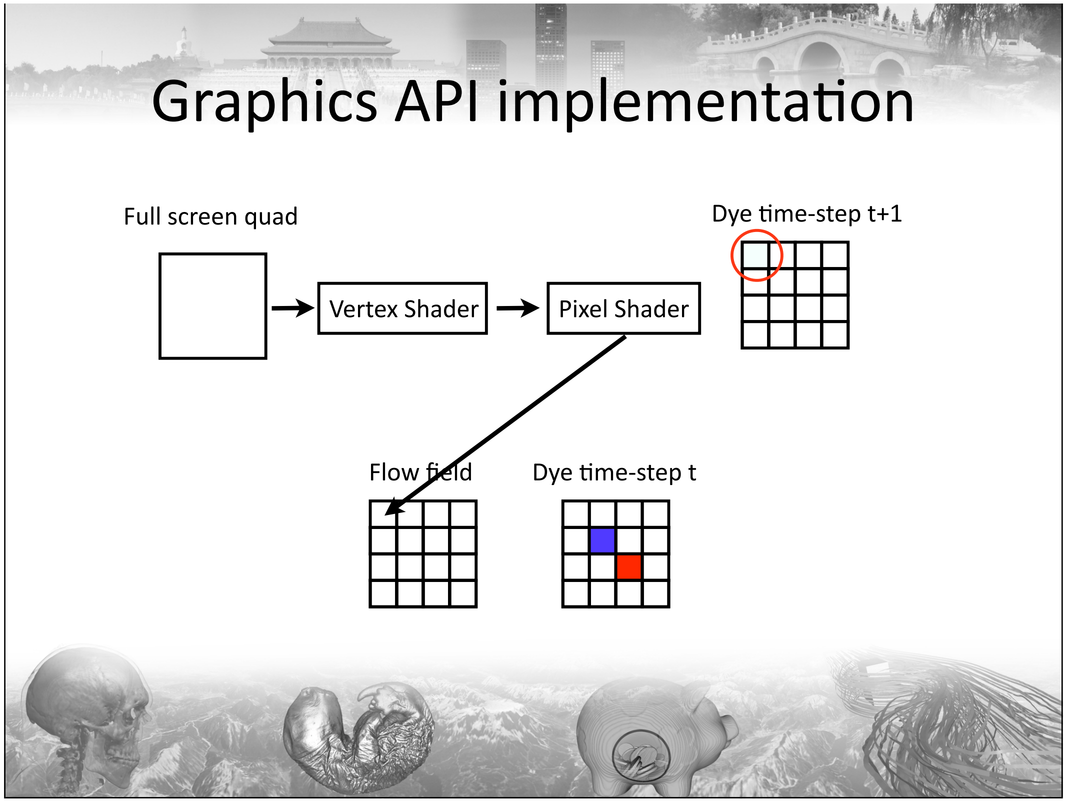
Pixel Shader

Dye time-step t+1

Flow field

Dye time-step t

In particluar the semi-lagrangian approach is very easy to implement on the GPU in 2D. In every time step a quad is rendered that covers the entire domain, in the pixel shader for ever fragment (=texel) the flow field is fetched, the flow vector is added to the current texture coordinate and with this the last time-step‘s dye is fetched and stored at the current position.



Graphics API implementation

Full screen quad

Vertex Shader

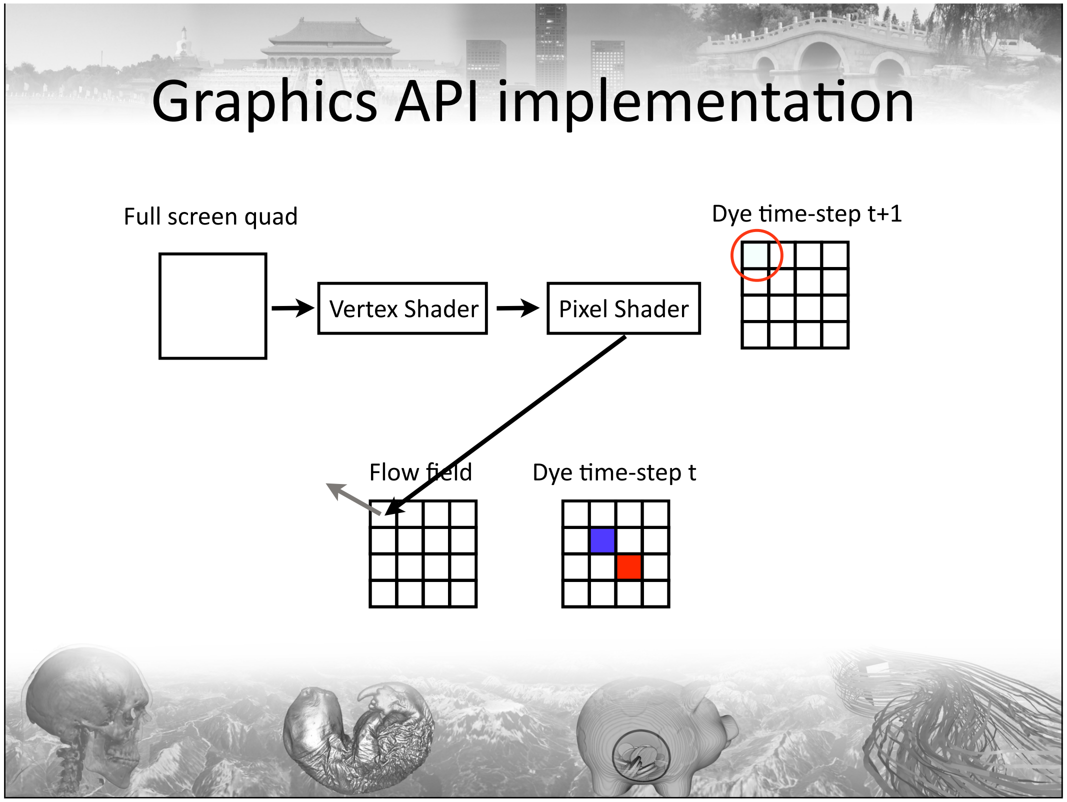
Pixel Shader

Dye time-step t+1

Flow field

Dye time-step t

In particluar the semi-lagrangian approach is very easy to implement on the GPU in 2D. In every time step a quad is rendered that covers the entire domain, in the pixel shader for ever fragment (=texel) the flow field is fetched, the flow vector is added to the current texture coordinate and with this the last time-step‘s dye is fetched and stored at the current position.



Graphics API implementation

Full screen quad

Vertex Shader

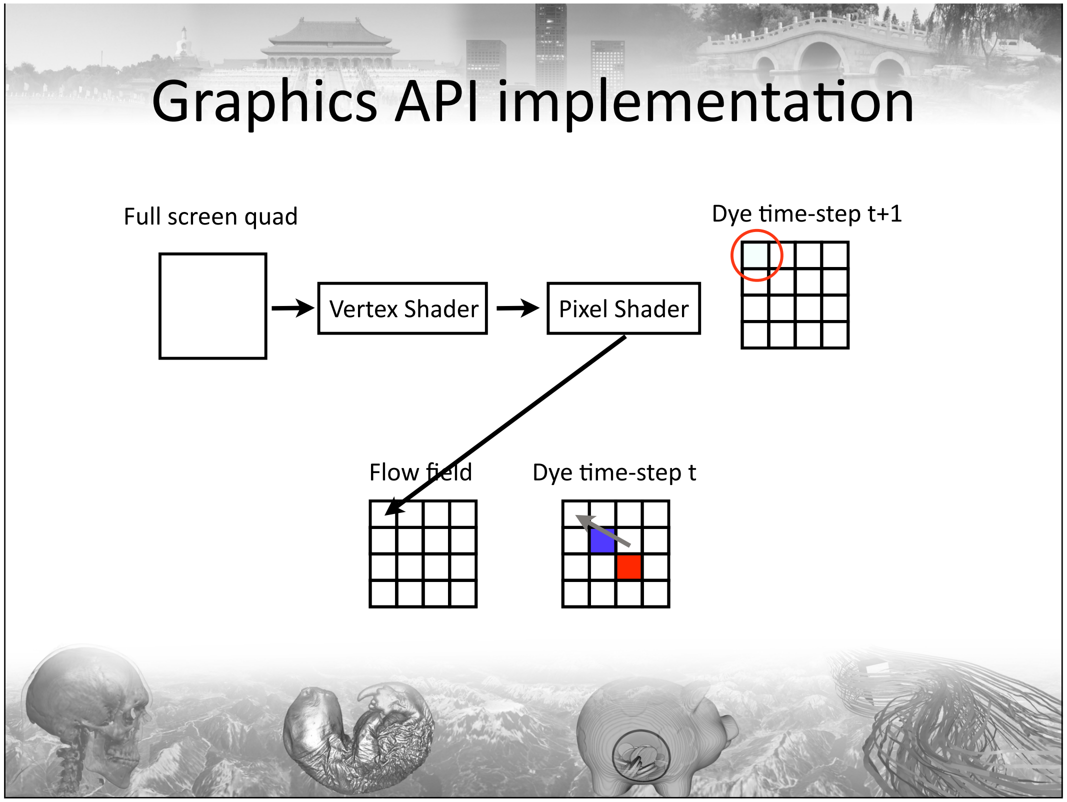
Pixel Shader

Dye time-step t+1

Flow field

Dye time-step t

In particluar the semi-lagrangian approach is very easy to implement on the GPU in 2D. In every time step a quad is rendered that covers the entire domain, in the pixel shader for ever fragment (=texel) the flow field is fetched, the flow vector is added to the current texture coordinate and with this the last time-step‘s dye is fetched and stored at the current position.



Graphics API implementation

Full screen quad

Vertex Shader

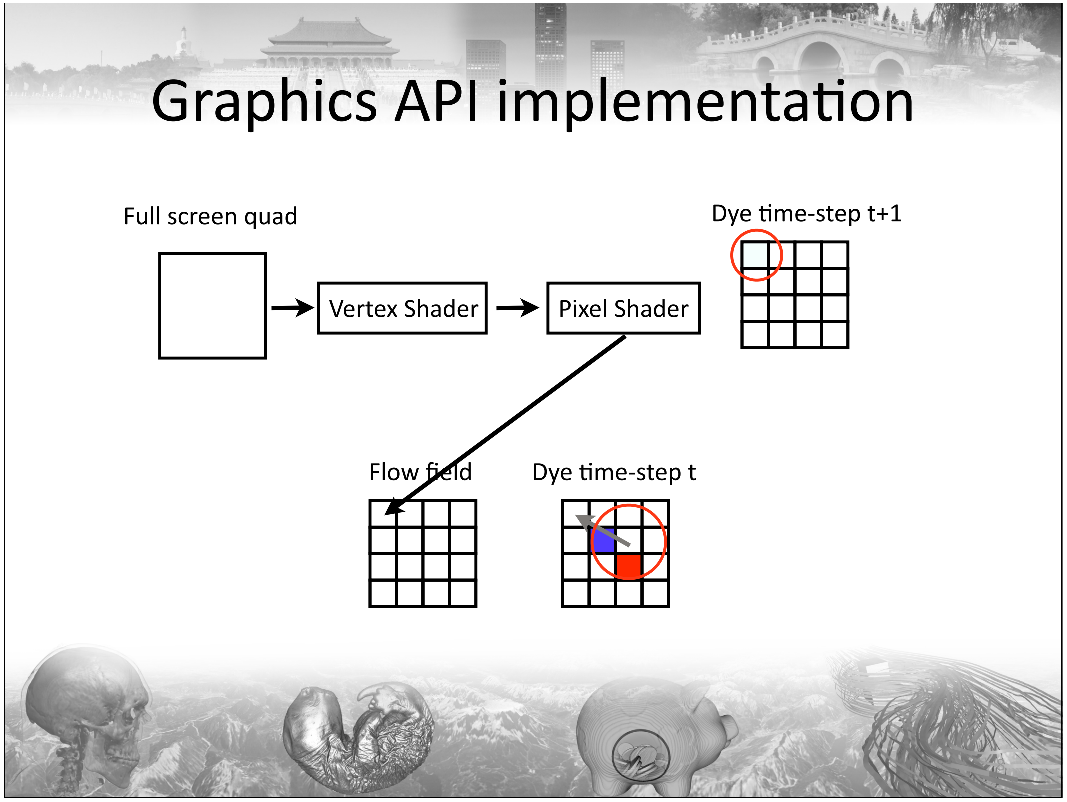
Pixel Shader

Dye time-step t+1

Flow field

Dye time-step t

In particluar the semi-lagrangian approach is very easy to implement on the GPU in 2D. In every time step a quad is rendered that covers the entire domain, in the pixel shader for ever fragment (=texel) the flow field is fetched, the flow vector is added to the current texture coordinate and with this the last time-step‘s dye is fetched and stored at the current position.



Graphics API implementation

Full screen quad

Vertex Shader

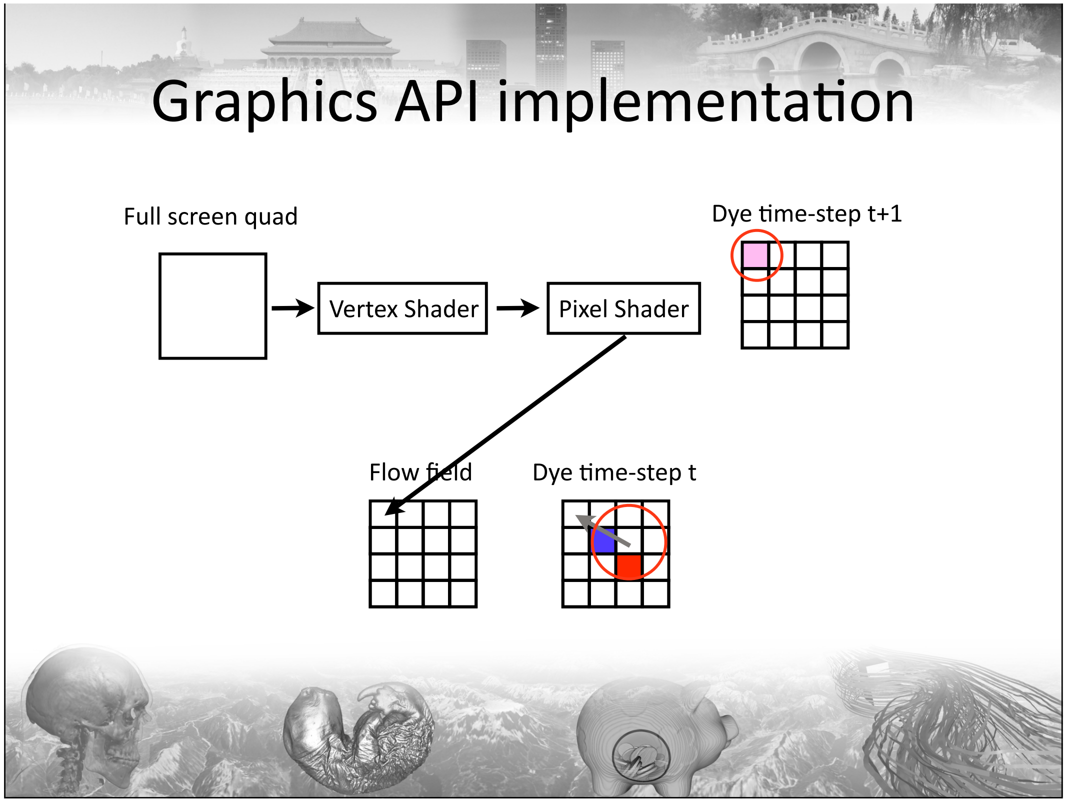
Pixel Shader

Dye time-step t+1

Flow field

Dye time-step t

In particluar the semi-lagrangian approach is very easy to implement on the GPU in 2D. In every time step a quad is rendered that covers the entire domain, in the pixel shader for ever fragment (=texel) the flow field is fetched, the flow vector is added to the current texture coordinate and with this the last time-step‘s dye is fetched and stored at the current position.

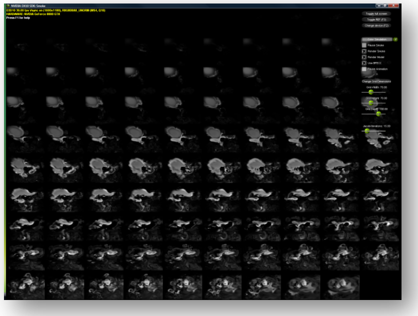


Graphics API implementation

In particluar the semi-lagrangian approach is very easy to implement on the GPU in 2D. In every time step a quad is rendered that covers the entire domain, in the pixel shader for ever fragment (=texel) the flow field is fetched, the flow vector is added to the current texture coordinate and with this the last time-step‘s dye is fetched and stored at the current position.

Dye Advection

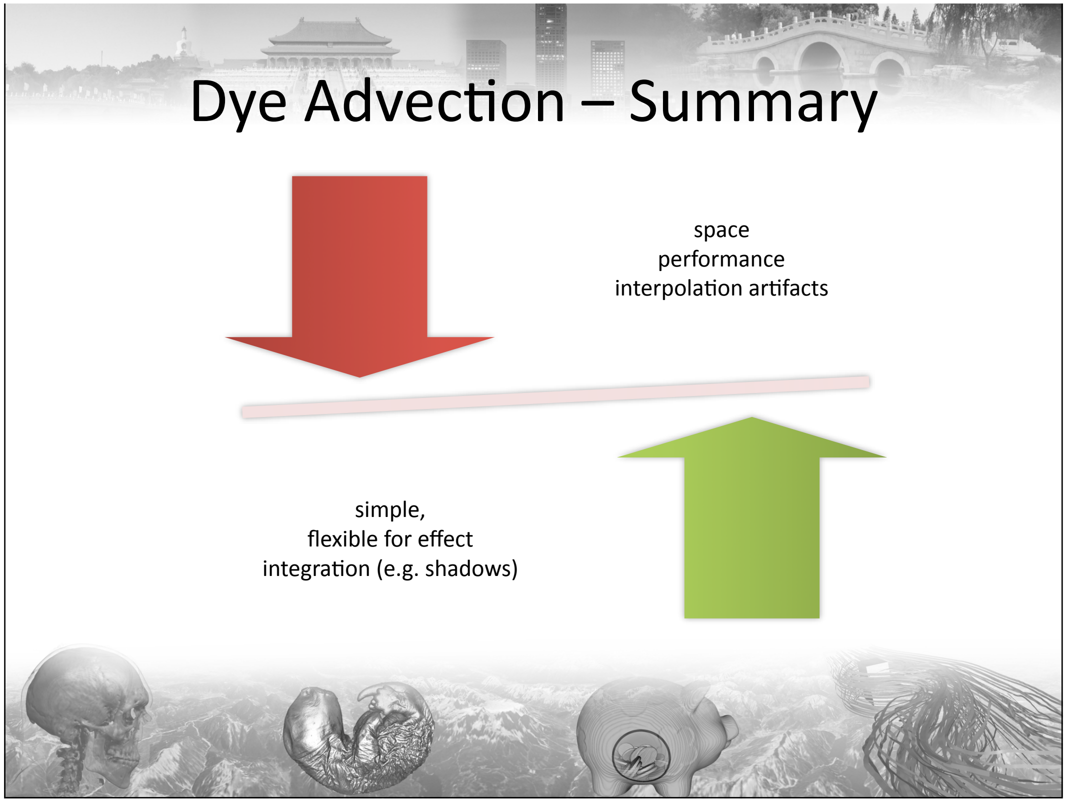
- GPU friendly for 2D
- 3D Requires either
  - flat textures
  - render to 3D texture
    - OpenGL extension DX10
  - GPGPU API
    - CUDA, OpenCL, DX11 compute



NVIDIA SDK http://developer.nvidia.com/page/home.html

For a 2D grid the implementation of dye advection on the GPU is really straight forward, on a 3D lattice, however, the update of the volume becomes less easy to implement. To handle 3D grids three approaches can be used:

- use a stack or an atlas of 2D grids (this limits the size of the volume to whatever fits into a 2D texture, and makes interpolation more complex)
- use render to 3D texture if available on the target platform
- use arrays in a GPGPU API such as CUDA, OpenCL, or DirectX11 compute shader API

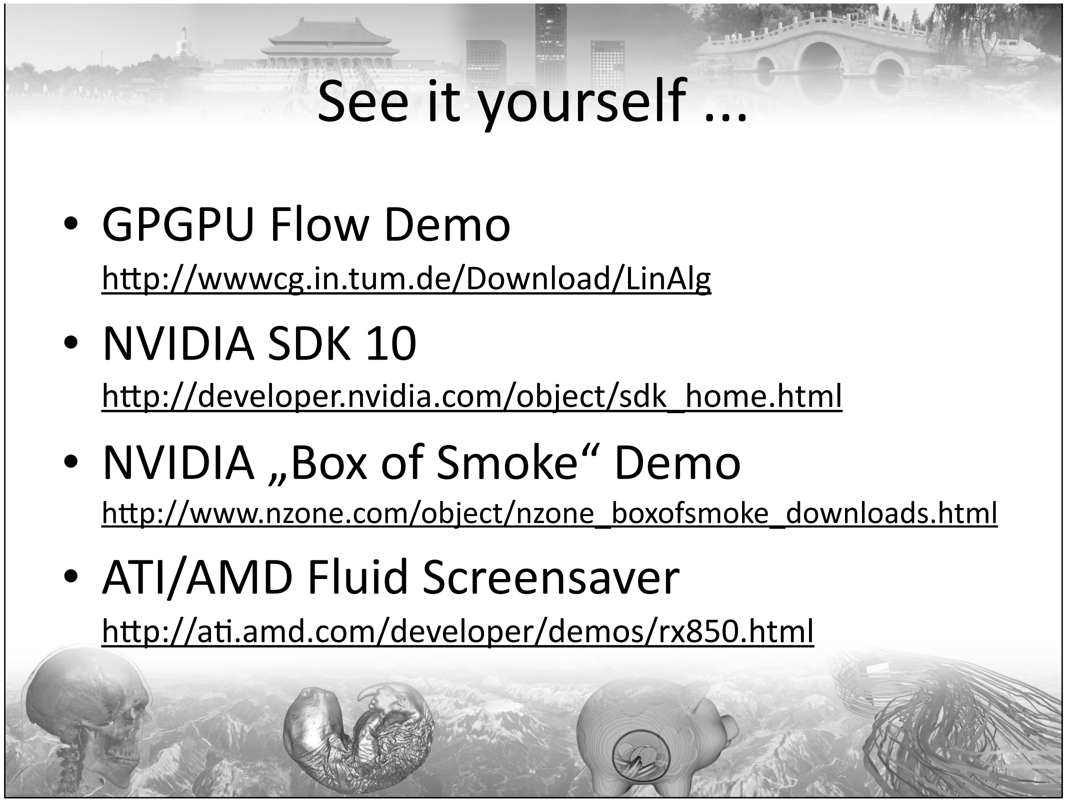


Dye Advection – Summary

space
performance
interpolation artifacts

simple,
flexible for effect
integration (e.g. shadows)

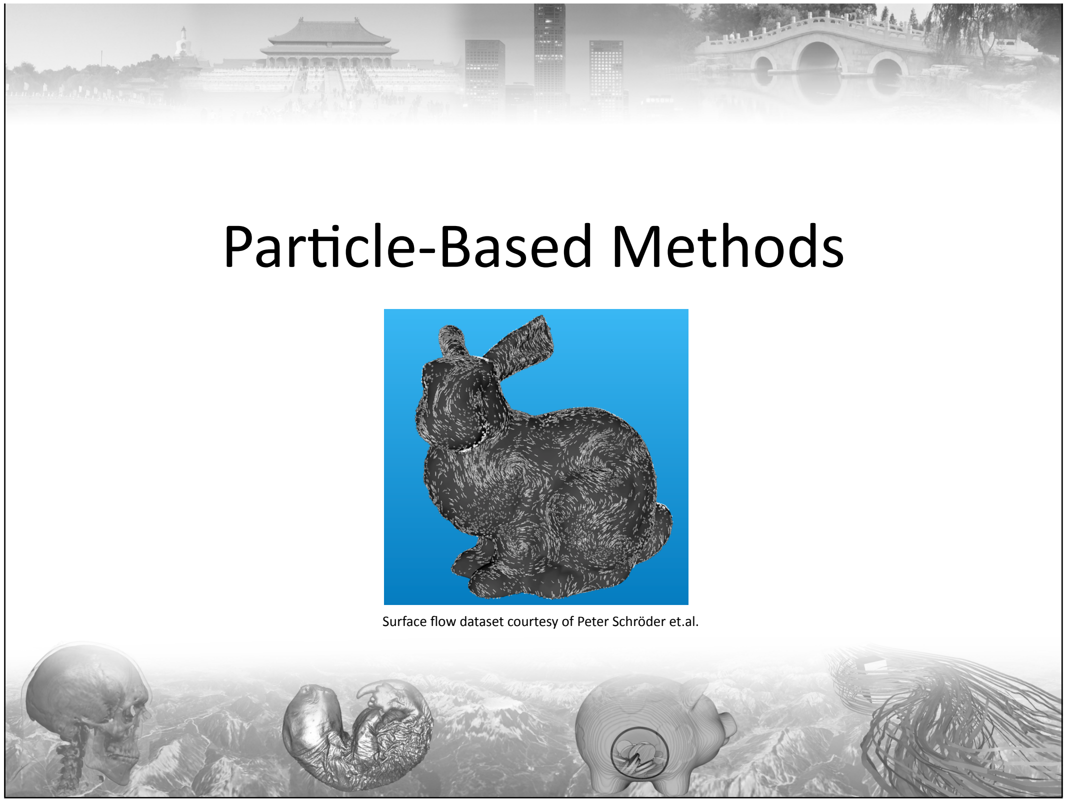
In a nutshell dye advection (at least for 2D) is the most simple approach (try it yourself, this method can be coded in minutes) and it allows for the integration of some „standart“ effects, such as volumetric shadows, thus it works best in scenarios where eye candy is needed, e.g. games. On the downside this method requires a lot of space (i.e. a high resolution dye volume) and since in every frame this entire volume must be updated, the performance may be slow. To tackle this aften low resolution grids are used but in that case interpolation artefacts become visible during rendering.



See it yourself ...

- GPGPU Flow Demo
  http://wwwcg.in.tum.de/Download/LinAlg
- NVIDIA SDK 10
  http://developer.nvidia.com/object/sdk_home.html
- NVIDIA „Box of Smoke“ Demo
  http://www.nzone.com/object/nzone_boxofsmoke_downloads.html
- ATI/AMD Fluid Screensaver
  http://ati.amd.com/developer/demos/rx850.html

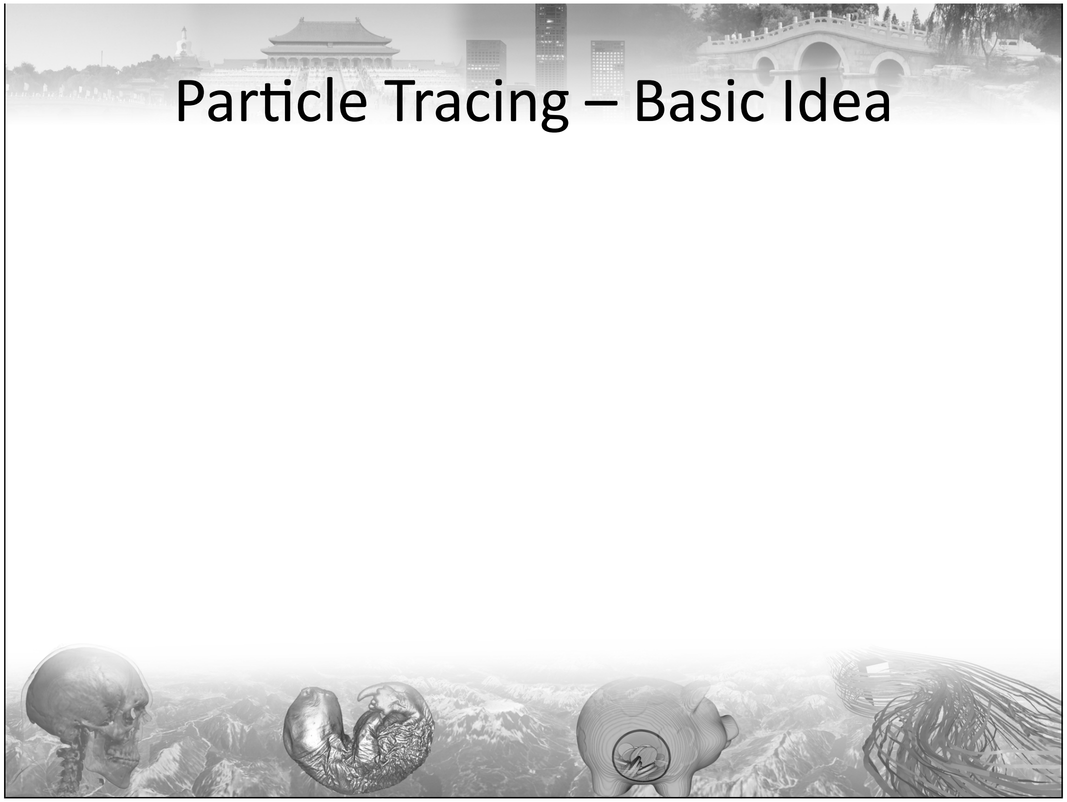
To see the dye advection technique in action, visit these sides, and download and try these demo programs.



Particle-Based Methods

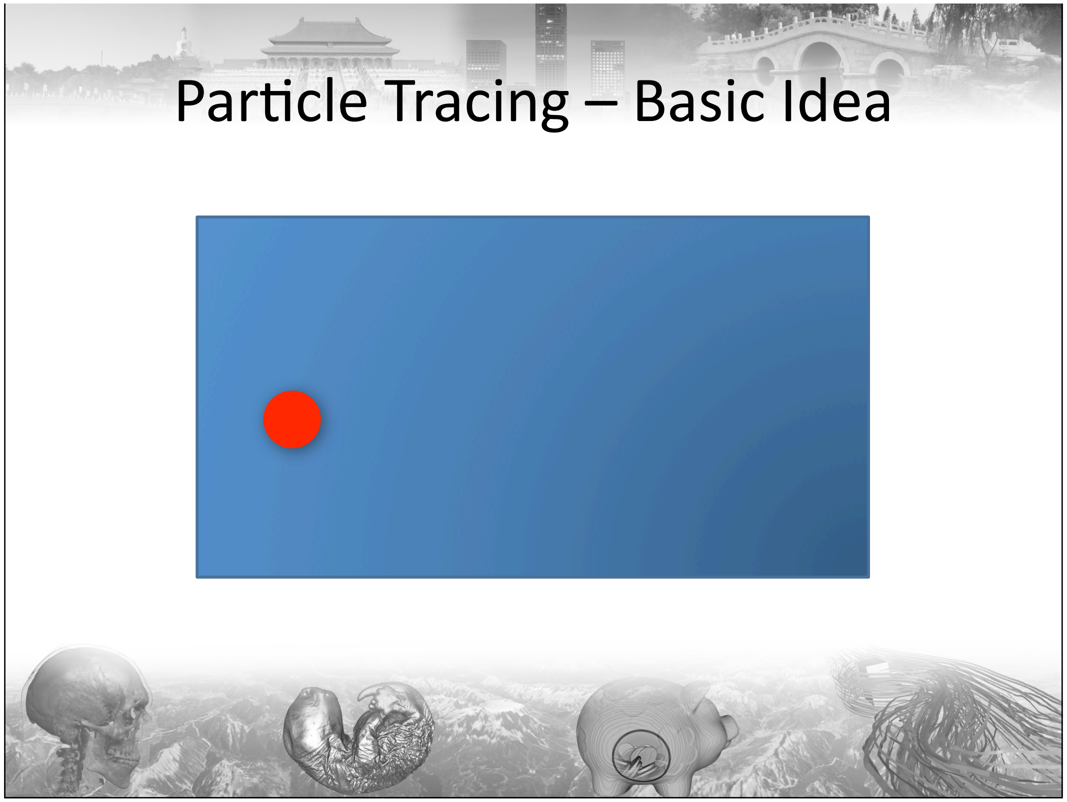
Surface flow dataset courtesy of Peter Schröder et.al.

The next family of methods is based on an Eulerian approach, that takes discrete particle quantities in a continuos domain into account.



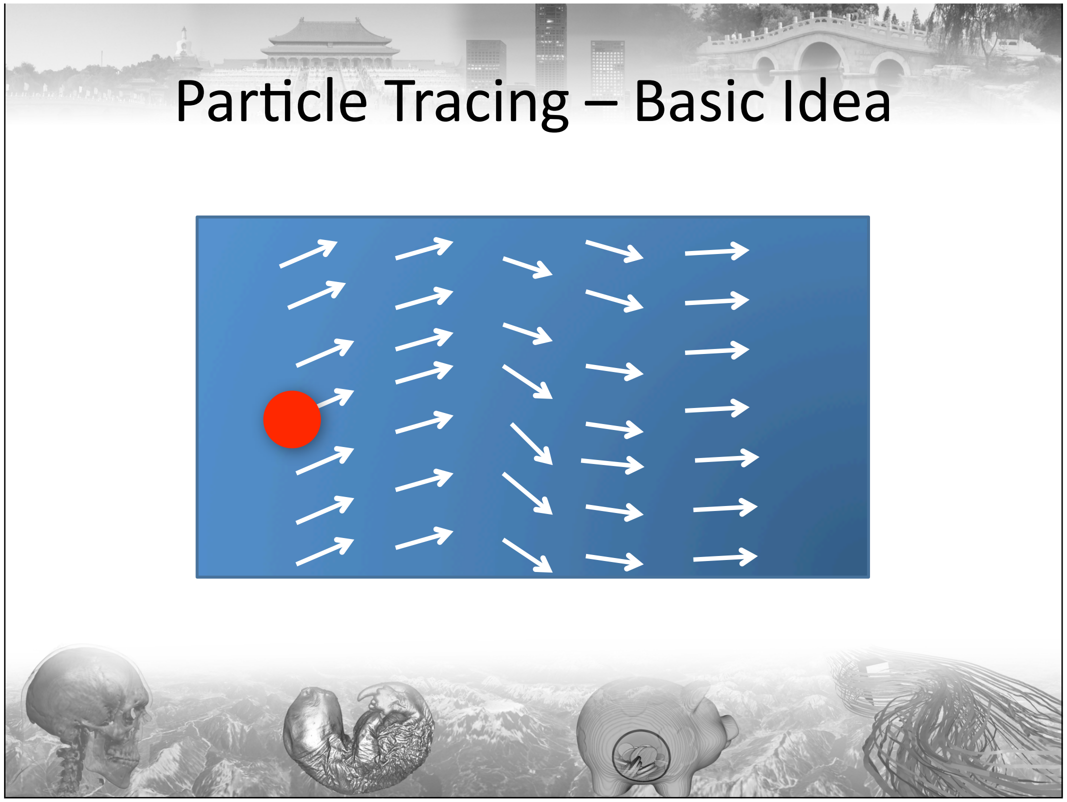
Particle Tracing – Basic Idea

This animation demonstrates the basic idea of particle tracing and shows that to advect a particle in the flow a simple ODE needs to be solved, it also shows that for this method to be useful as a visualization technique, many particles are required.



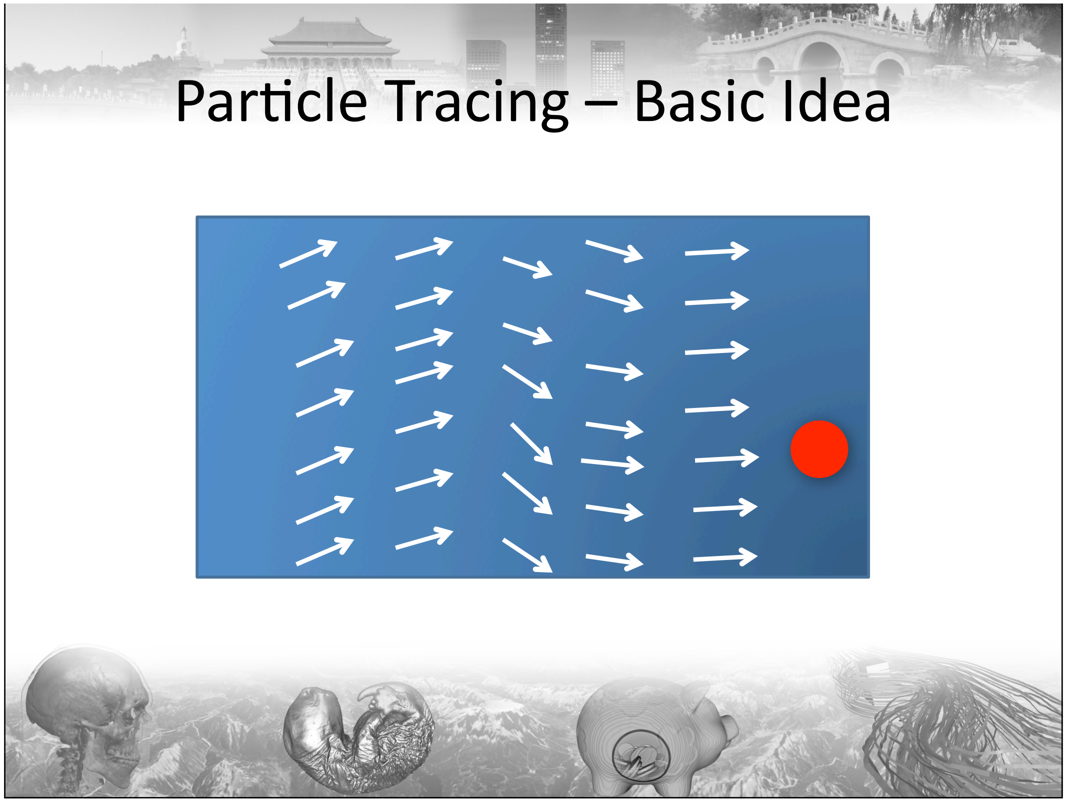
Particle Tracing – Basic Idea

This animation demonstrates the basic idea of particle tracing and shows that to advect a particle in the flow a simple ODE needs to be solved, it also shows that for this method to be useful as a visualization technique, many particles are required.



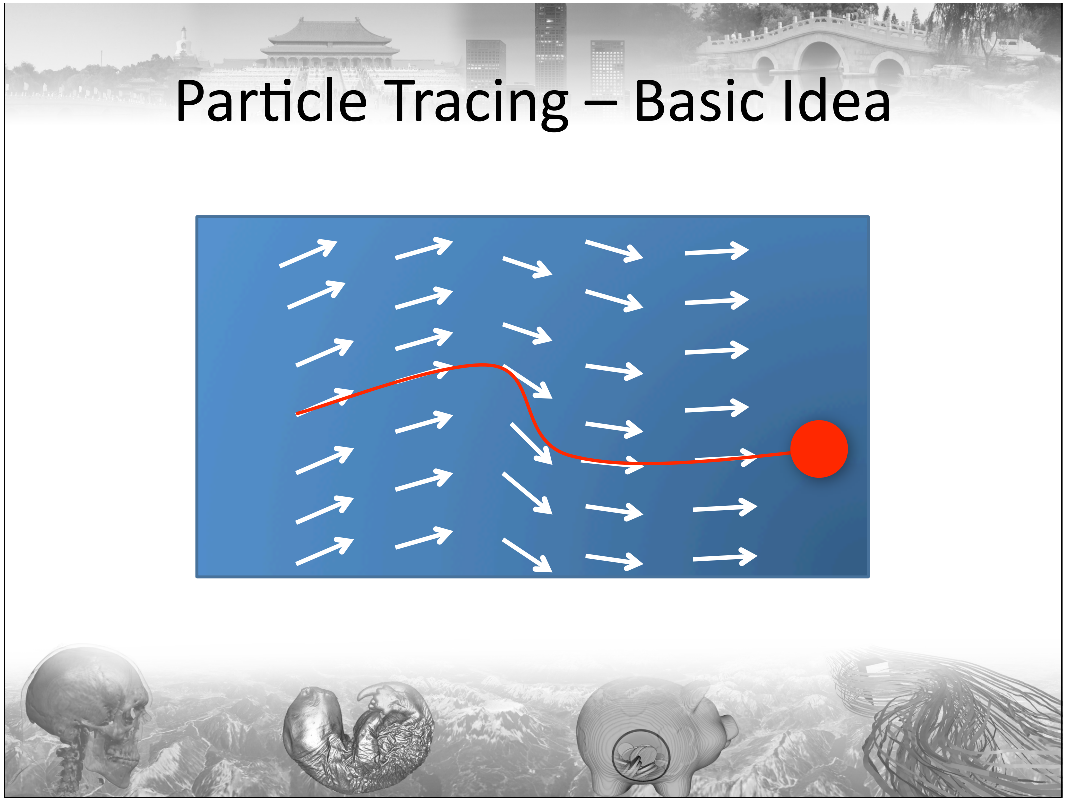
Particle Tracing – Basic Idea

This animation demonstrates the basic idea of particle tracing and shows that to advect a particle in the flow a simple ODE needs to be solved, it also shows that for this method to be useful as a visualization technique, many particles are required.



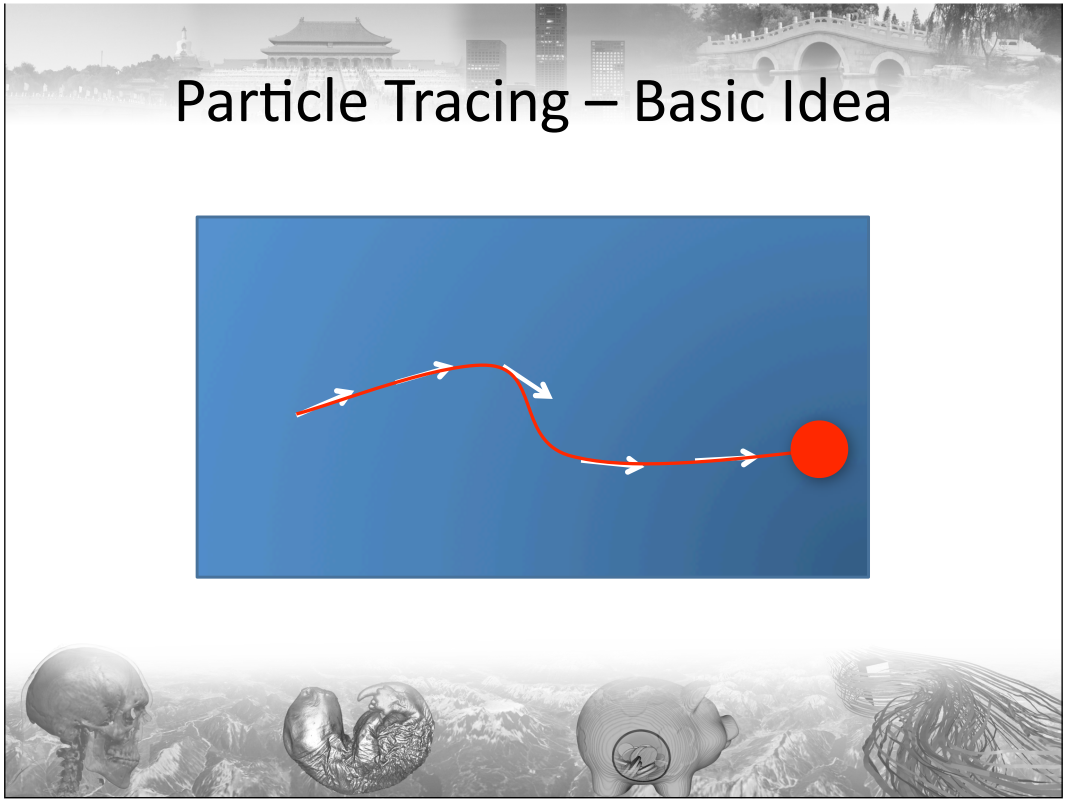
Particle Tracing – Basic Idea

This animation demonstrates the basic idea of particle tracing and shows that to advect a particle in the flow a simple ODE needs to be solved, it also shows that for this method to be useful as a visualization technique, many particles are required.



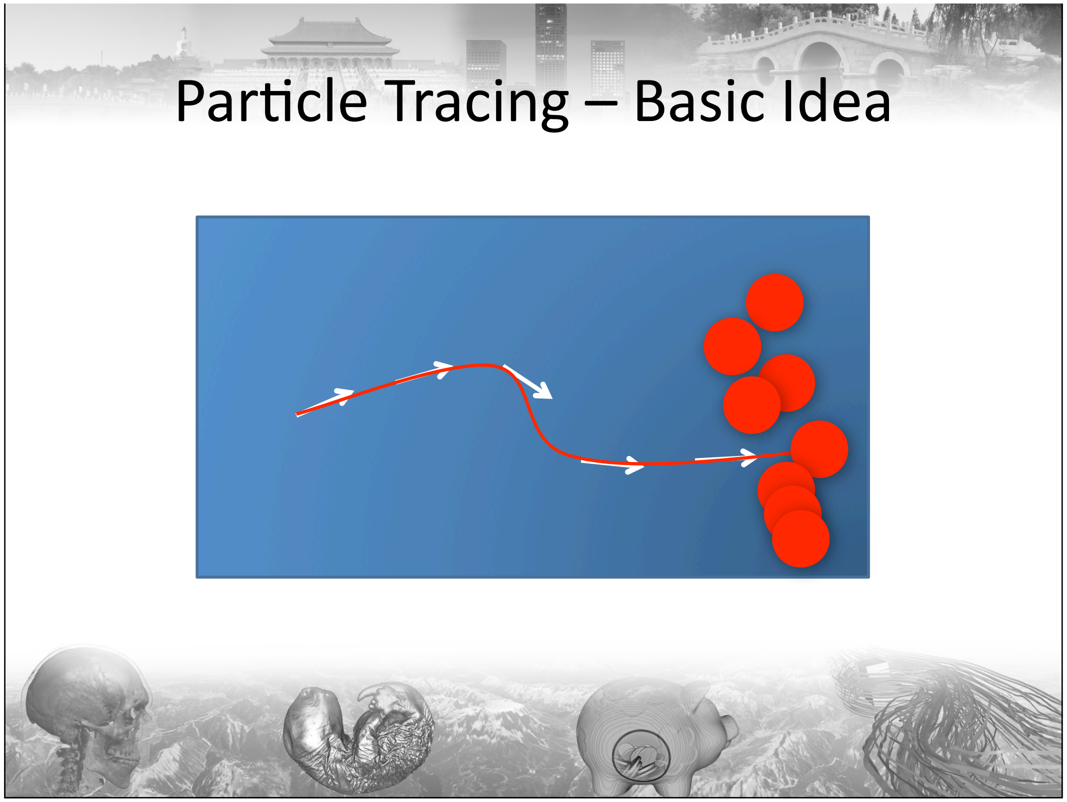
Particle Tracing – Basic Idea

This animation demonstrates the basic idea of particle tracing and shows that to advect a particle in the flow a simple ODE needs to be solved, it also shows that for this method to be useful as a visualization technique, many particles are required.



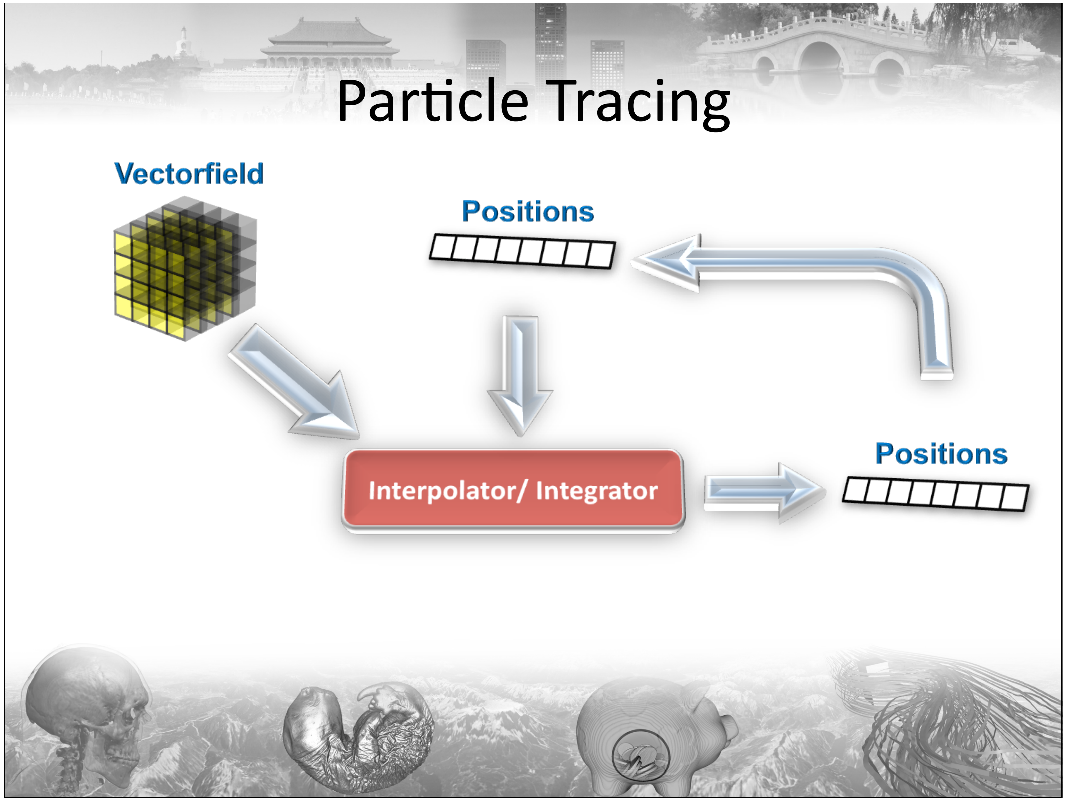
Particle Tracing – Basic Idea

This animation demonstrates the basic idea of particle tracing and shows that to advect a particle in the flow a simple ODE needs to be solved, it also shows that for this method to be useful as a visualization technique, many particles are required.



Particle Tracing – Basic Idea

This animation demonstrates the basic idea of particle tracing and shows that to advect a particle in the flow a simple ODE needs to be solved, it also shows that for this method to be useful as a visualization technique, many particles are required.



Particle Tracing

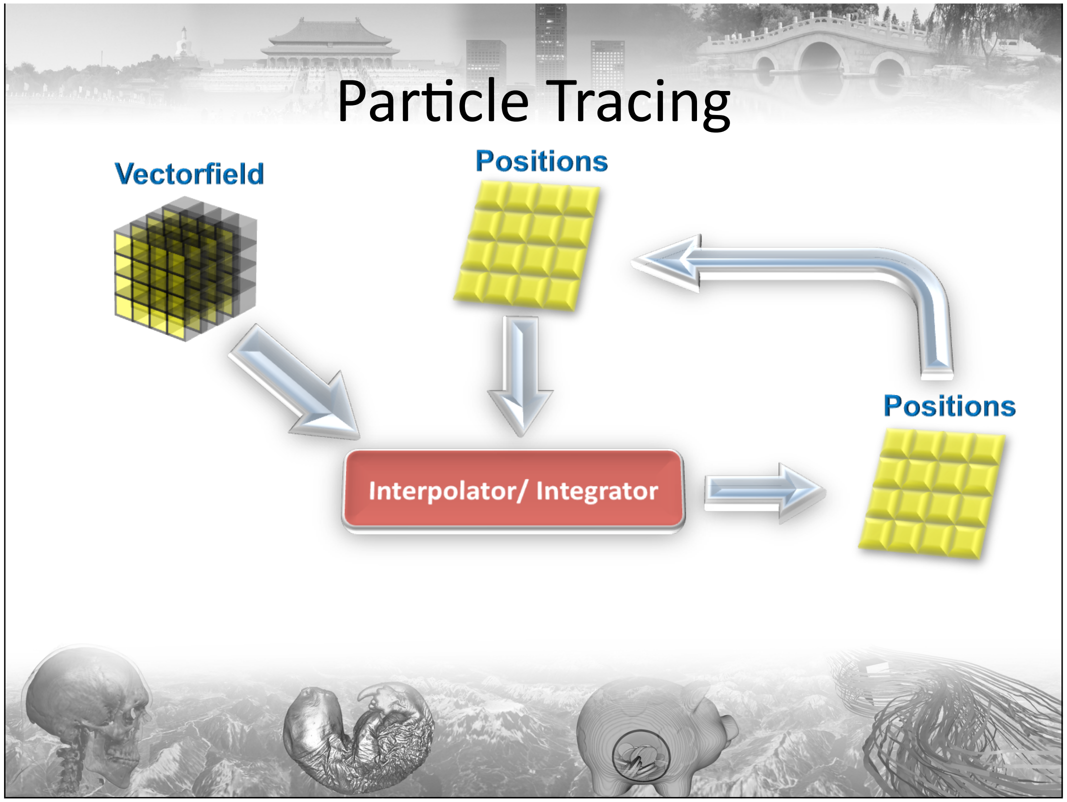
Vectorfield

Positions

Interpolator/ Integrator

Positions

This slide and the next slide demonstrate how particle tracing is achieved on a graphics card. In essence the pixel shader is used to access the vector-field and update the position texture. Note that on newer DirectX 10 class GPUs this process can be done in the geometry shader to update the vertex positions directly, instead of first updating a position texture and then displacing vertices with this texture.



Particle Tracing

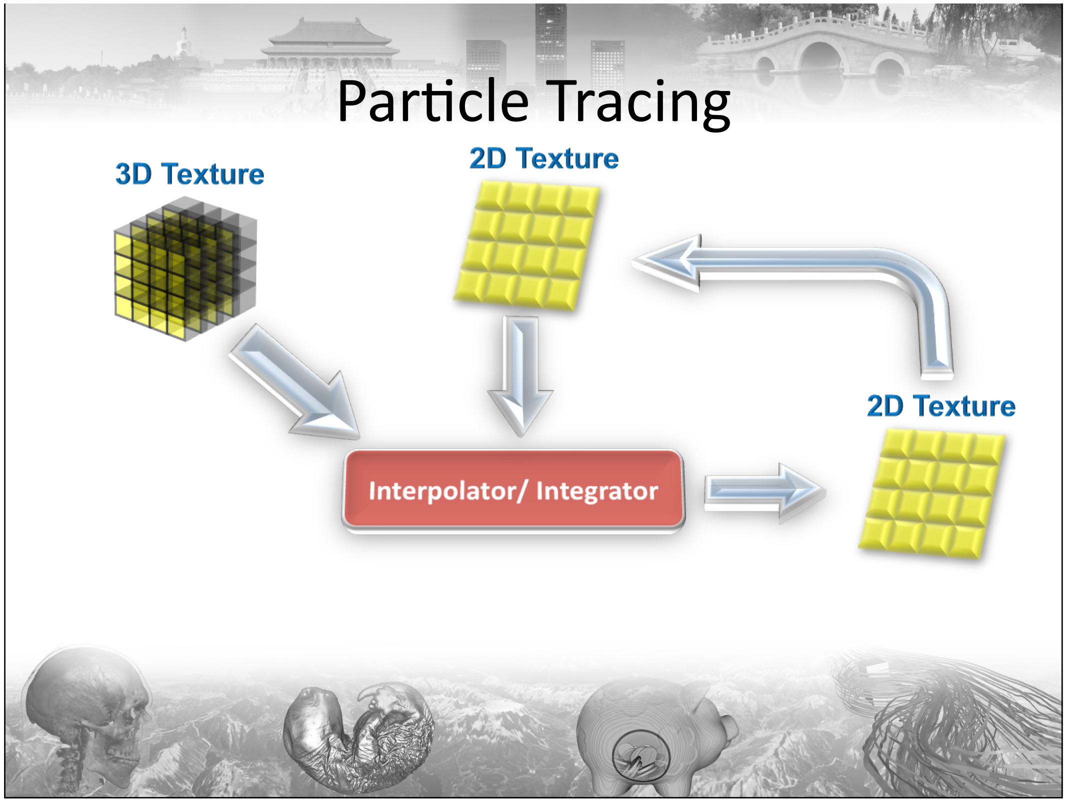
Vectorfield

Positions

Interpolator/ Integrator

Positions

This slide and the next slide demonstrate how particle tracing is achieved on a graphics card. In essence the pixel shader is used to access the vector-field and update the position texture. Note that on newer DirectX 10 class GPUs this process can be done in the geometry shader to update the vertex positions directly, instead of first updating a position texture and then displacing vertices with this texture.



Particle Tracing

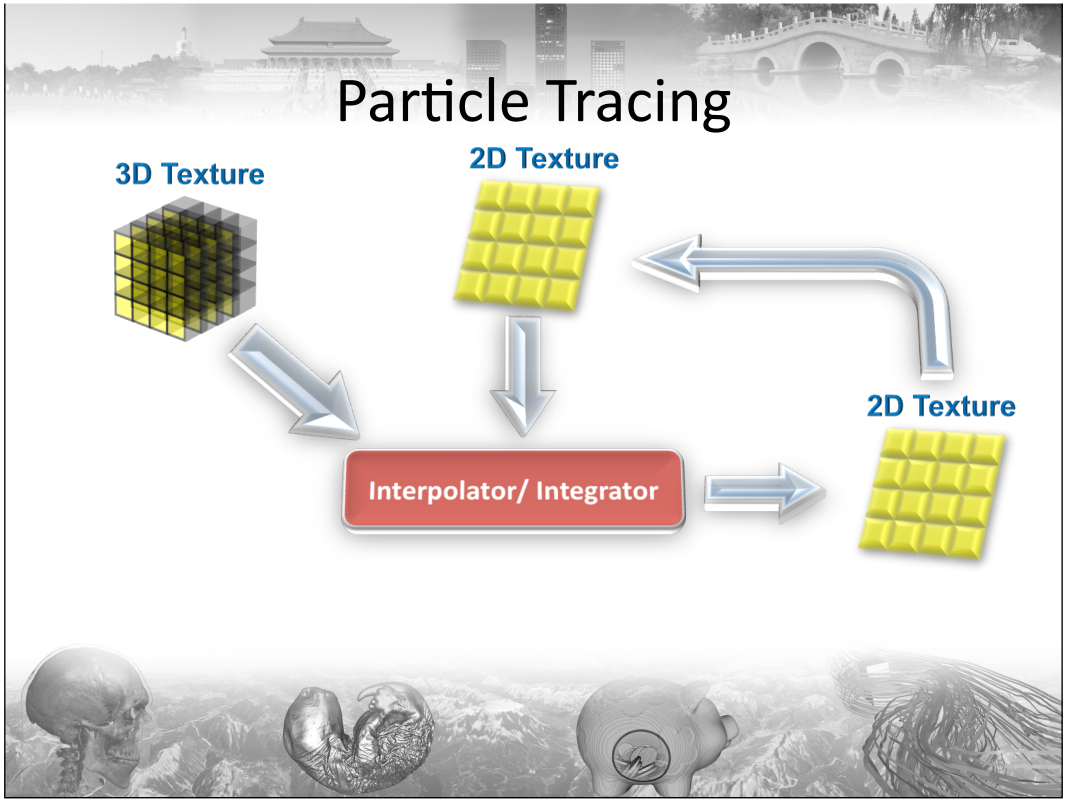
3D Texture

2D Texture

Interpolator/ Integrator

2D Texture

This slide and the next slide demonstrate how particle tracing is achieved on a graphics card. In essence the pixel shader is used to access the vector-field and update the position texture. Note that on newer DirectX 10 class GPUs this process can be done in the geometry shader to update the vertex positions directly, instead of first updating a position texture and then displacing vertices with this texture.



Particle Tracing

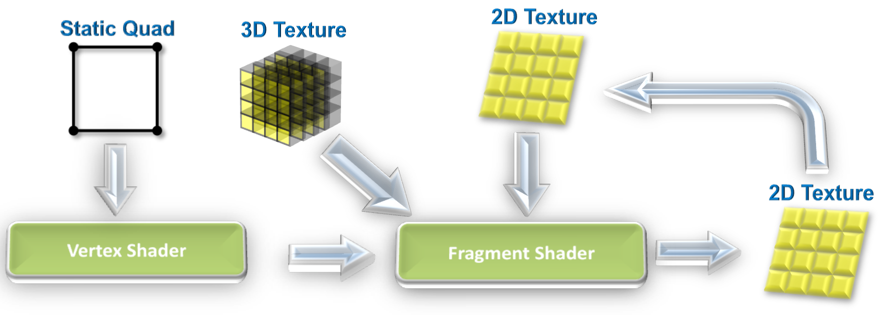
3D Texture

2D Texture

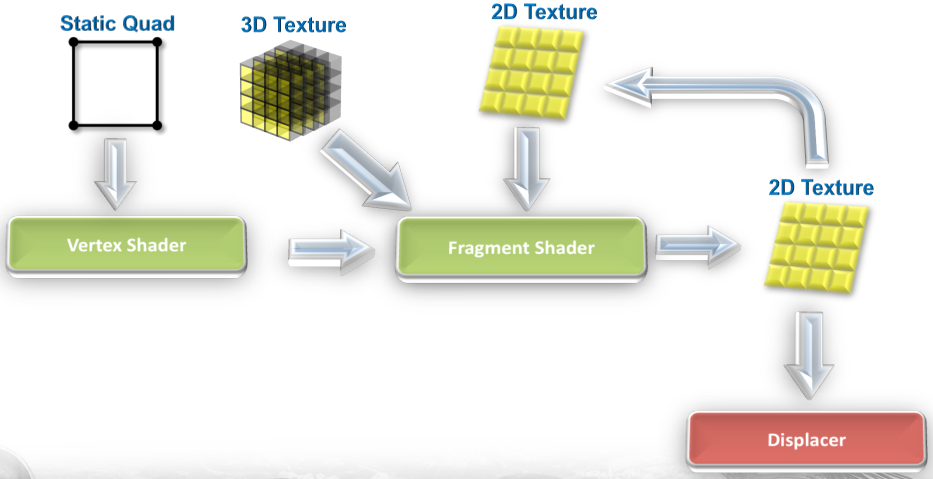
2D Texture

Interpolator/ Integrator

Particle Tracing



Particle Tracing



Static Quad

3D Texture

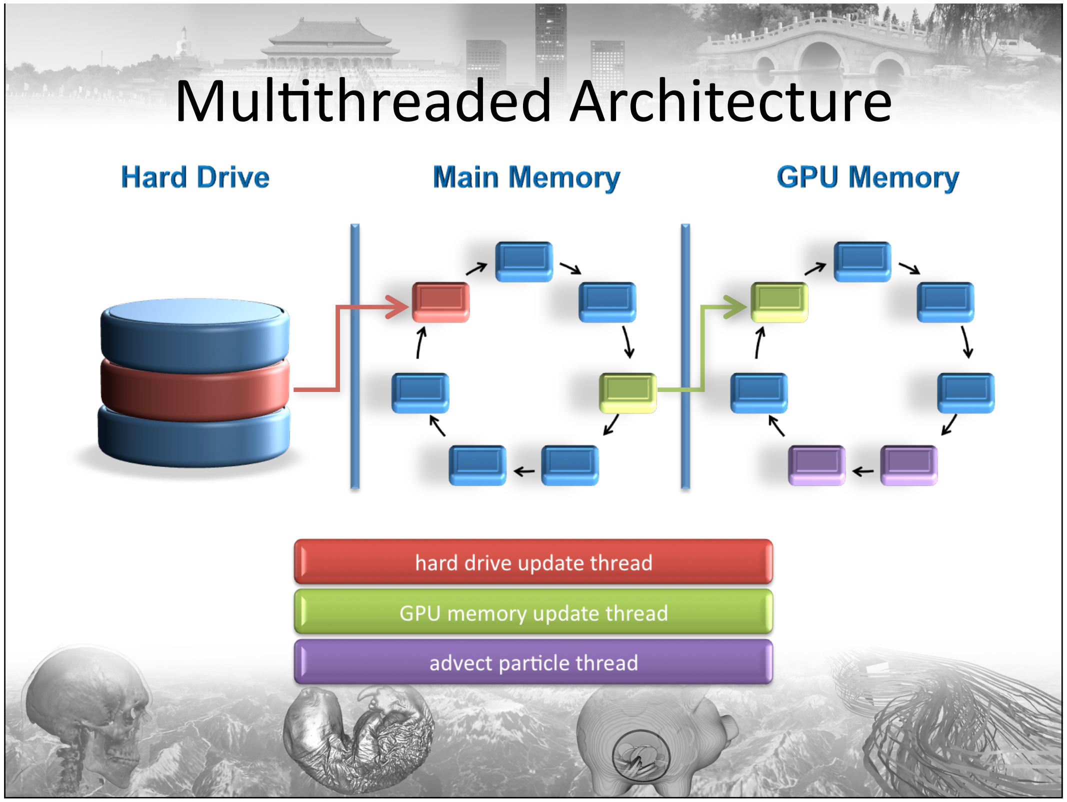
2D Texture

Vertex Shader

Fragment Shader

2D Texture

Displacer



Multithreaded Architecture

Hard Drive

Main Memory

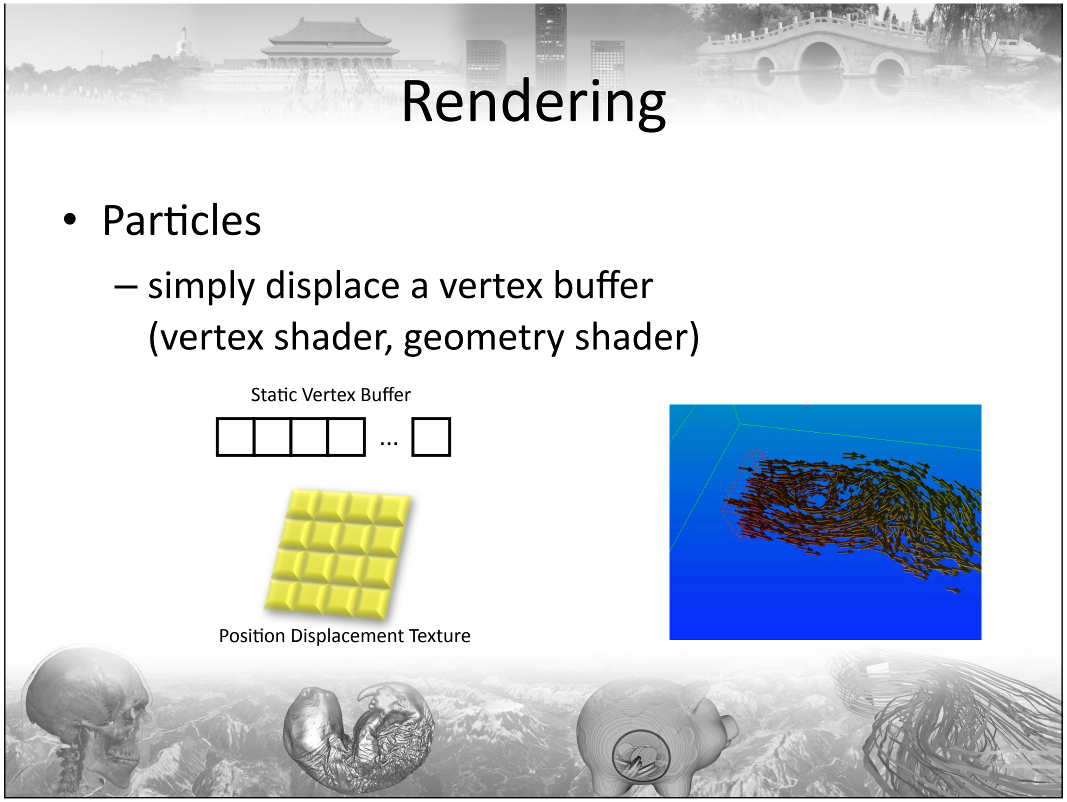
GPU Memory

hard drive update thread

GPU memory update thread

advect particle thread

This is just the general theme for large out of core rendering/processing for much more details on this see the large-data talk of this tutorial later today.



Rendering

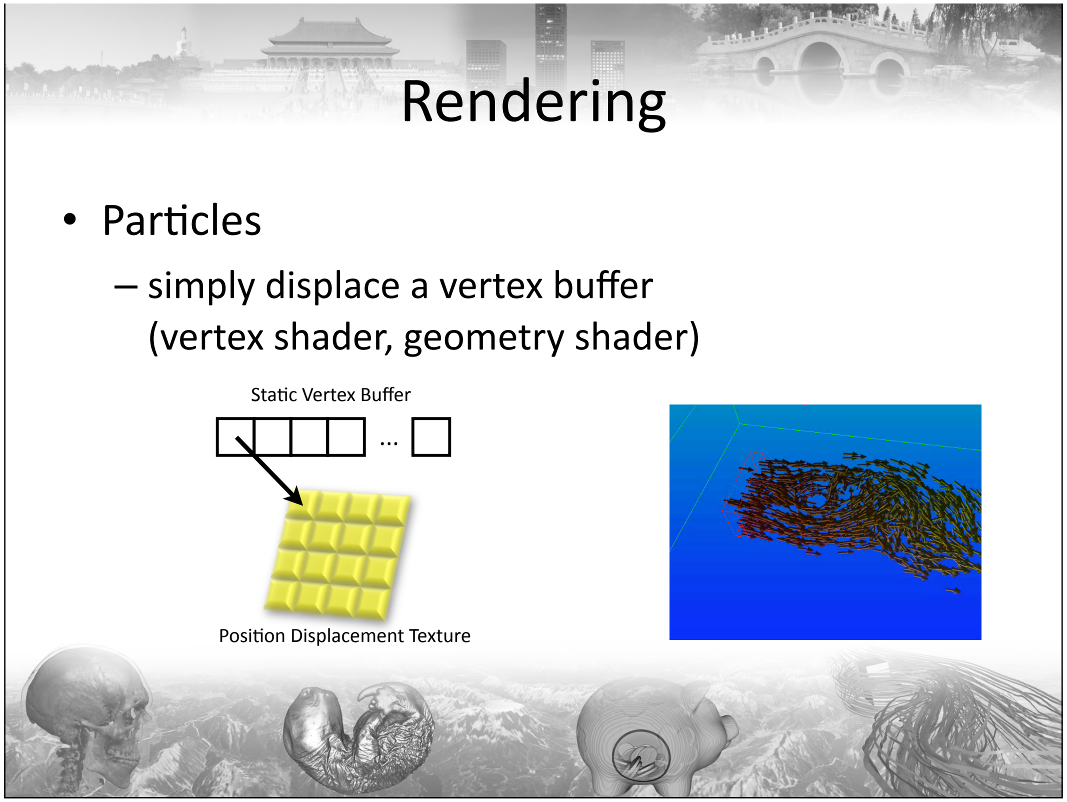
- Particles
  - – simply displace a vertex buffer (vertex shader, geometry shader)

Static Vertex Buffer

…

Position Displacement Texture

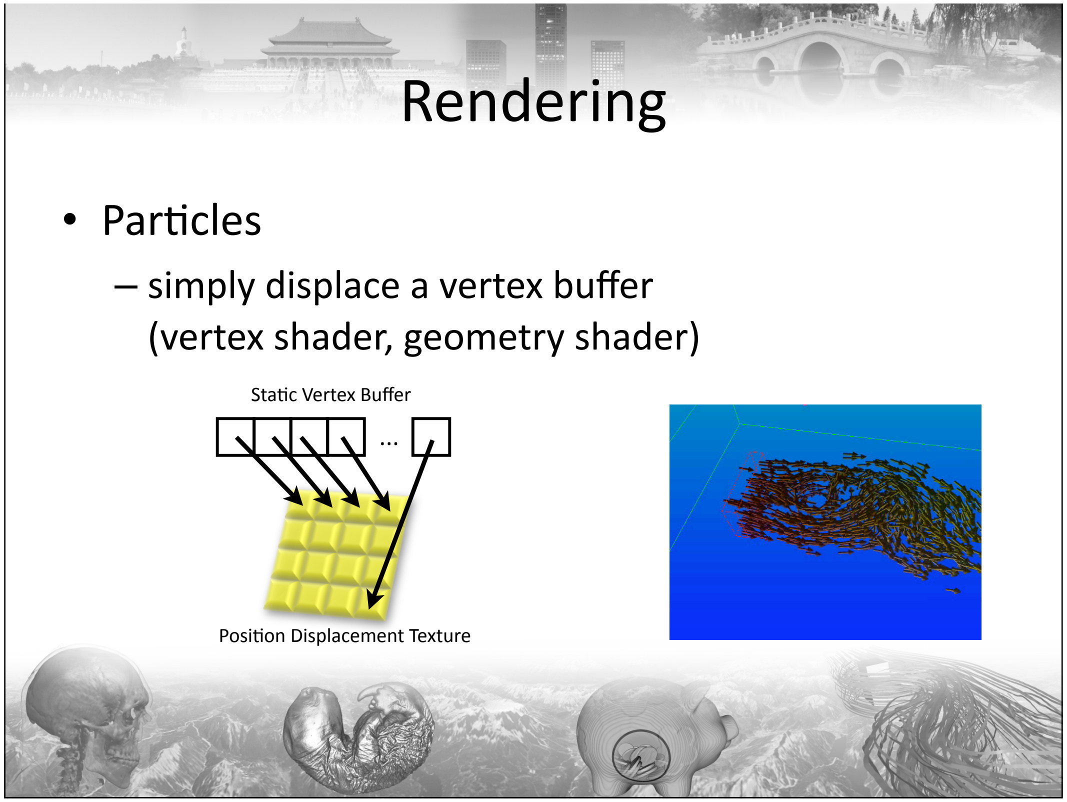
Most simple method, Idea:
Send a static vertex buffer into the pipeline / every element in the vertex shader fetches it‘s current position from the displacement texture / render displaced vertices as point sprites
If the graphics hardware is DirectX 10 class:
Instead of using the 2D textures in the previous step a vertex buffer can be used in the geometry shader directly in this case no explicit displacement is necessary



Rendering

- Particles
  - simply displace a vertex buffer (vertex shader, geometry shader)

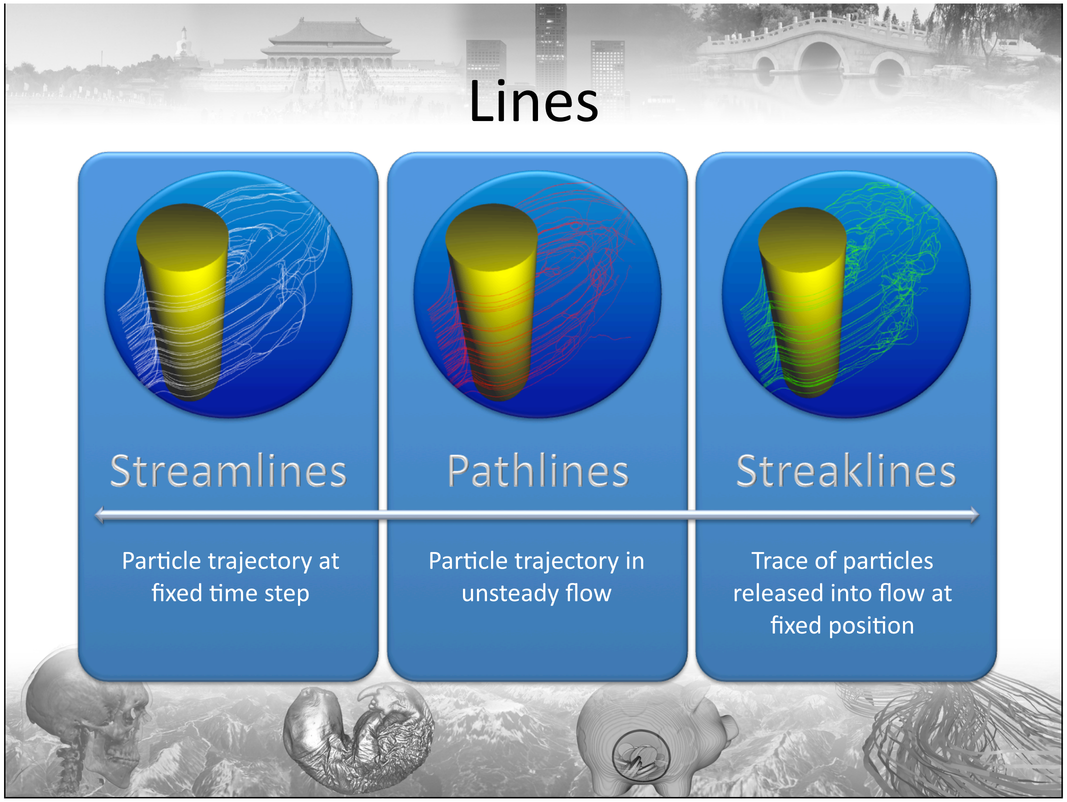
Most simple method, Idea:
Send a static vertex buffer into the pipeline / every element in the vertex shader fetches it‘s current position from the displacement texture / render displaced vertices as point sprites
If the graphics hardware is DirectX 10 class:
Instead of using the 2D textures in the previous step a vertex buffer can be used in the geometry shader directly in this case no explicit displacement is necessary



Rendering

- Particles
  - simply displace a vertex buffer (vertex shader, geometry shader)

Most simple method, Idea:
Send a static vertex buffer into the pipeline / every element in the vertex shader fetches it‘s current position from the displacement texture / render displaced vertices as point sprites
If the graphics hardware is DirectX 10 class:
Instead of using the 2D textures in the previous step a vertex buffer can be used in the geometry shader directly in this case no explicit displacement is necessary



Lines

Streamlines

Particle trajectory at fixed time step

Pathlines

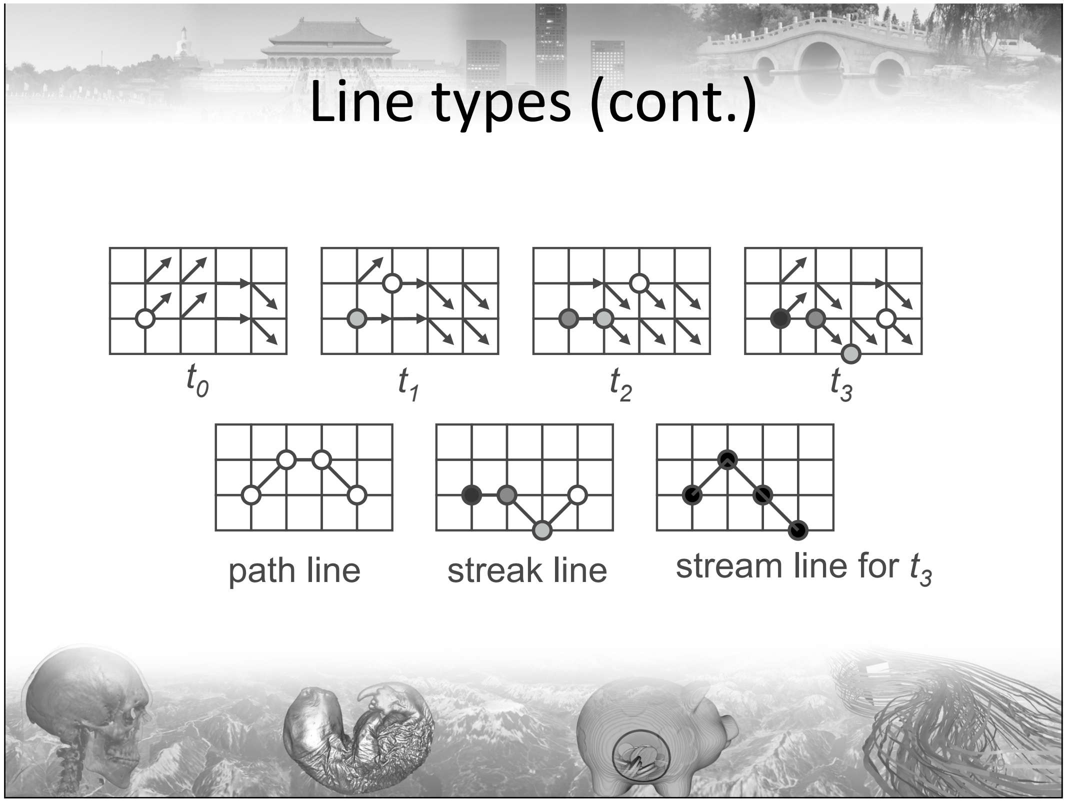
Particle trajectory in unsteady flow

Streaklines

Trace of particles released into flow at fixed position

In an unsteady flow field there are three different possibilities to trace lines:
Streamlines, where the entire line is recomputed for every time-step (a somehwat artificial method)
Pathlines, that use the time t during the advection to access the corresponding time-step in the flow field (correspond to the path taken by massless particles in the flow)
Streaklines, release a new particle every time-step in the flow and always advect all particles (corresponds to ink, advected in the flow)

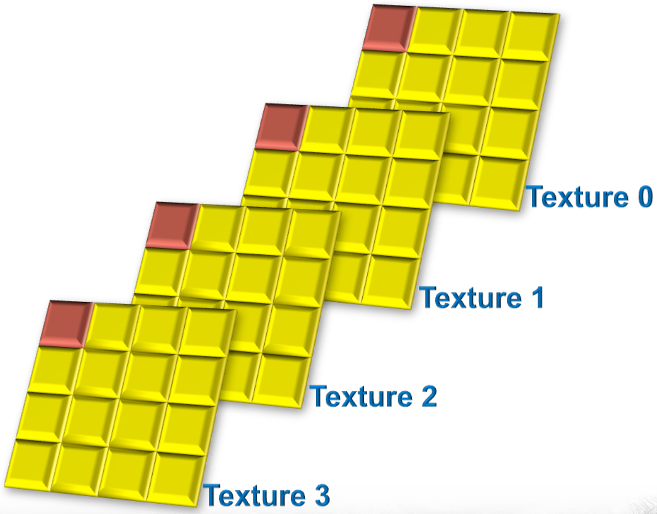
Note that in a steady flow field these three approaches are the same.



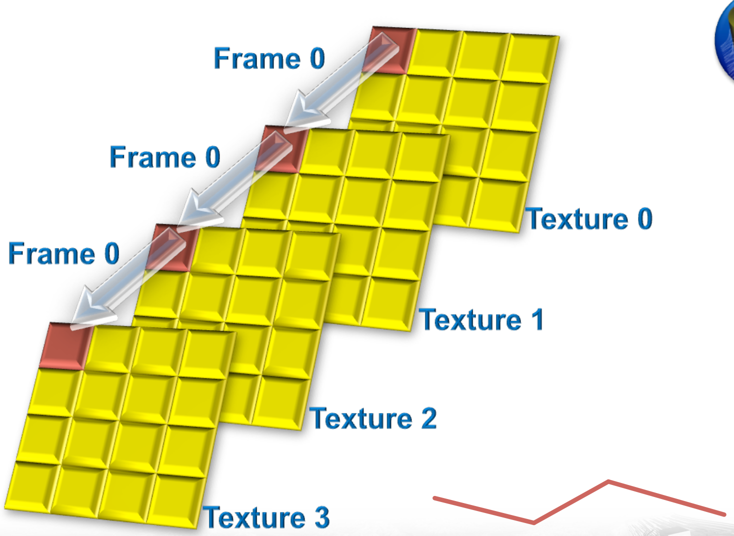
Line types (cont.)

This slide shows the difference of the three methods for a sequence of four time steps in a time dependent flow.

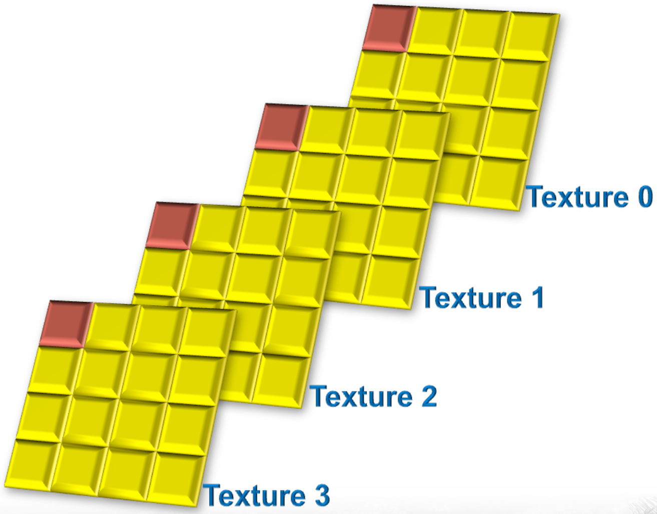
Streamlines



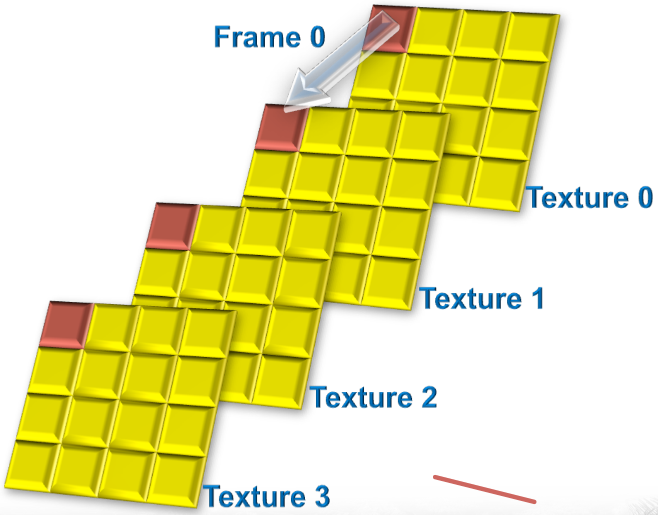
Streamlines



Pathlines



Pathlines



Frame 0

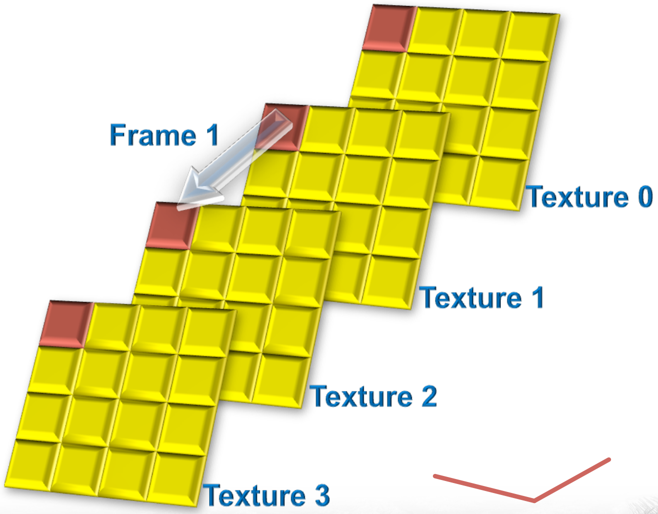
Texture 0

Texture 1

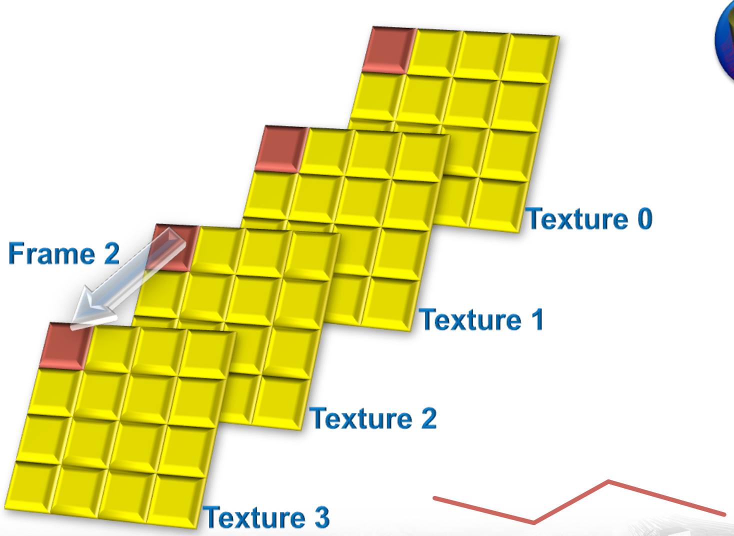
Texture 2

Texture 3

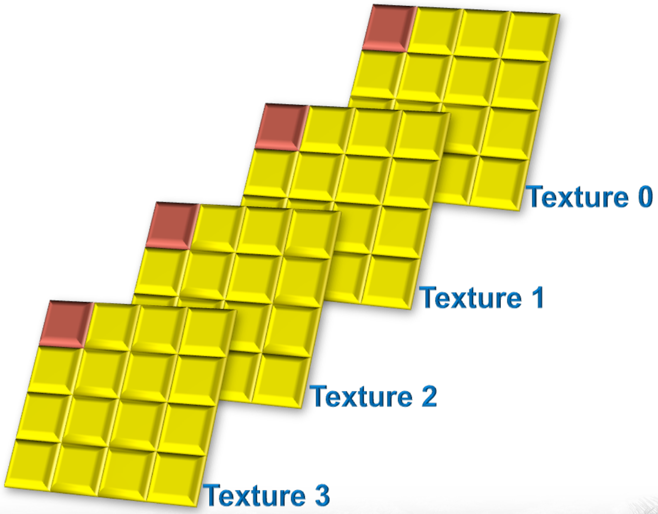
Pathlines



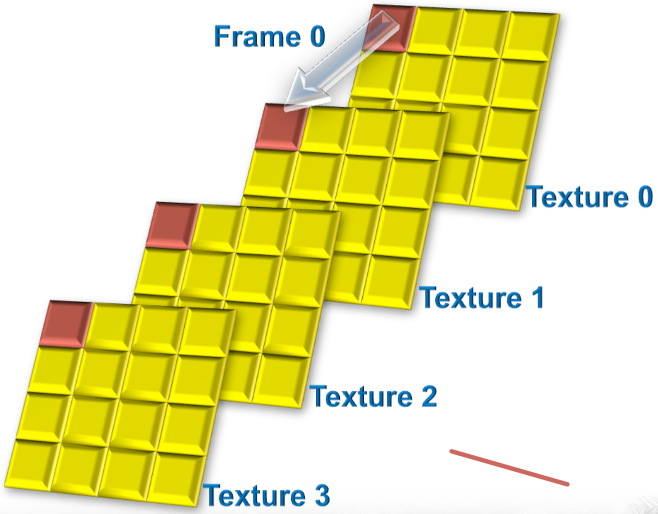
Pathlines



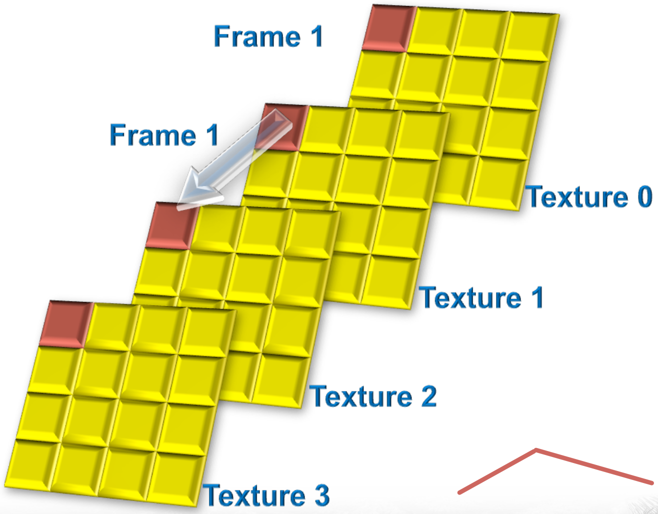
Streaklines



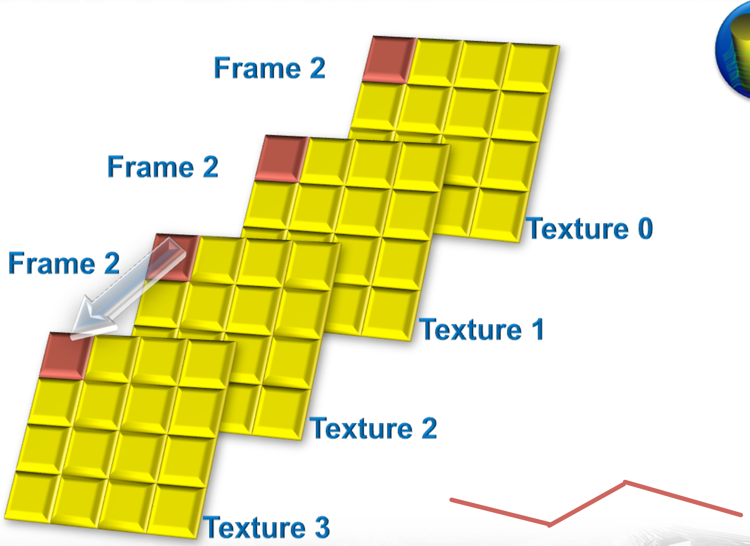
Streaklines



Streaklines



Streaklines



Frame 2

Frame 2

Texture 0

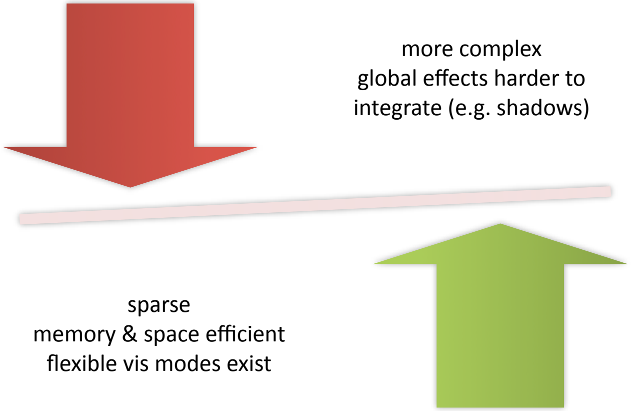
Frame 2

Texture 1

Texture 2

Texture 3

Particle Tracing – Summary



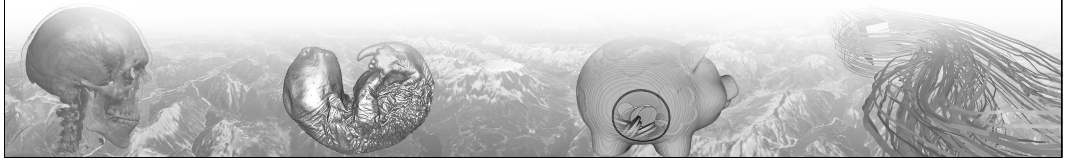
more complex
global effects harder to
integrate (e.g. shadows)

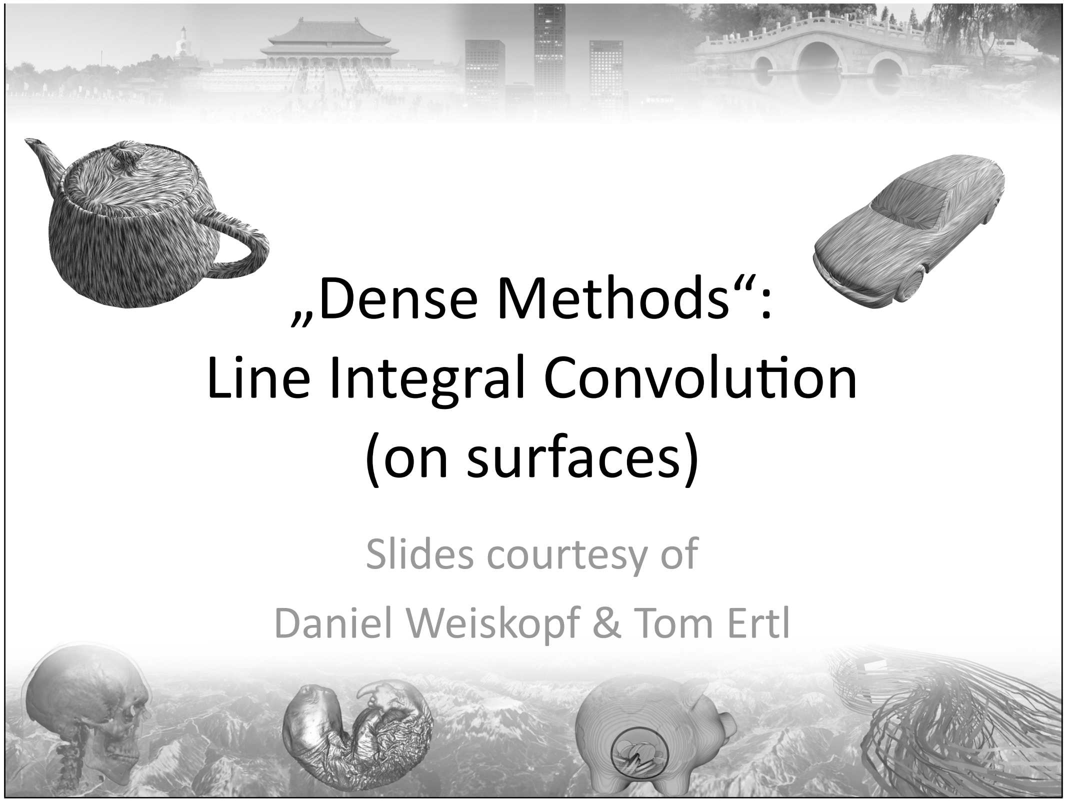
sparse
memory & space efficient
flexible vis modes exist

See it yourself ...

- tum.3D Particle Engine
  http://wwwcg.in.tum.de/Download/PE
- DirectX SDK

  http://msdn.microsoft.com/en-us/directx/default.aspx





„Dense Methods“: Line Integral Convolution (on surfaces)

Slides courtesy of

Daniel Weiskopf & Tom Ertl

Goals

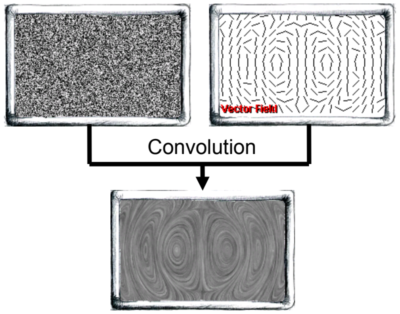
- On curved surfaces
  - Triangle meshes (triangle soup)
  - Tangential vector fields
- Dense representation
- GPU support for interactive visualization

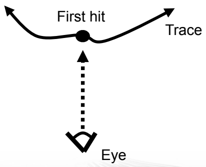
Basic Algorithm

- Noise-based dense visualization
  - LIC integral

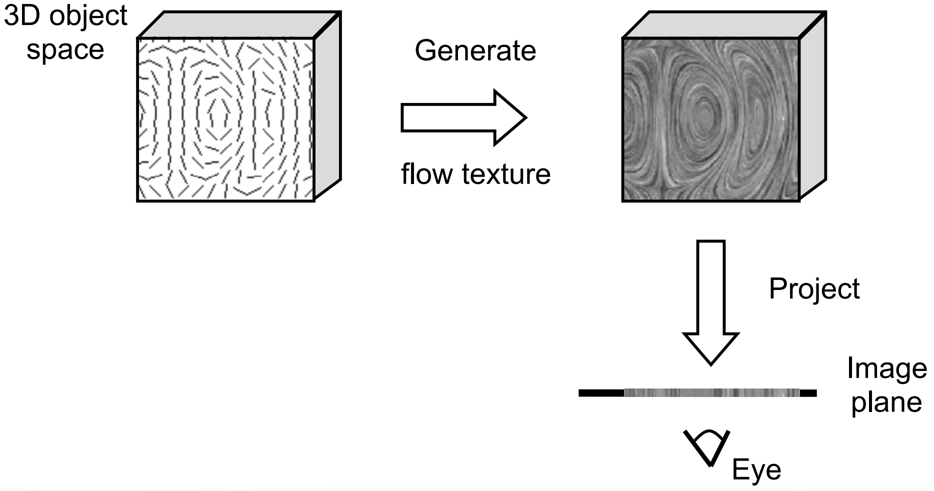
$$I = \int_{s_0-L}^{s_0+L} k(s-s_0)\times N(\mathbf{r}(s))\,ds$$

- Particle integration
  - Compute $\mathbf{r}(s)$
- Starting point: First hit with object
- Accumulate noise contribution $N(\mathbf{r})$ along traces





Texture-Based Flow Vis on Surfaces



3D object space

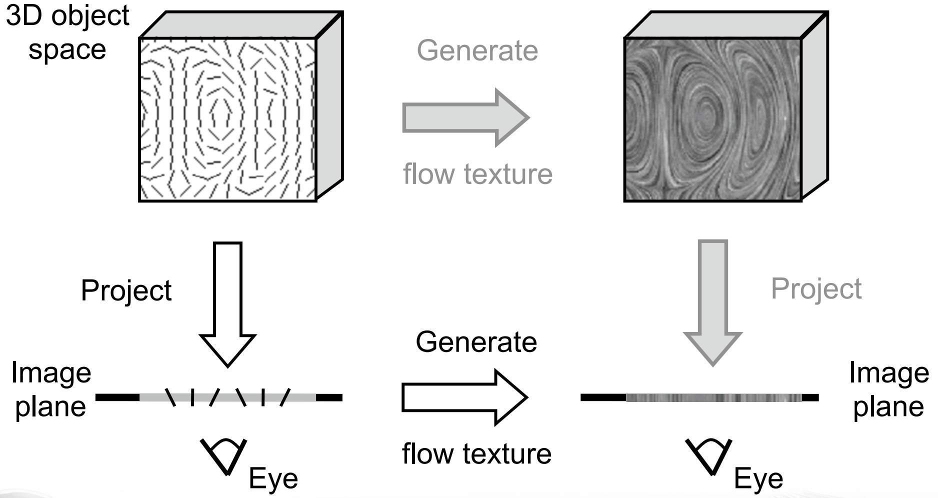
Generate flow texture

Project

Image plane

Eye

Texture-Based Flow Vis on Surfaces

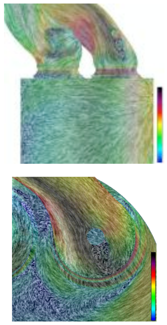


Advection in Image-Space

- Advantages
    - + No mesh connectivity or parameterization required
    - + Uniform noise density in image space
    - + Simple mapping to GPU
    - + Performance determined by viewport size, not mesh complexity

Advection in Image-Space

- Disadvantages
  - No or only limited frame-to-frame coherence
  - Blurring (numerical diffusion)
  - Only exponential filter
  - Silhouette lines are problematic inflow regions
  - Special detection of boundary lines needed
    to avoid cross-
    boundary flow



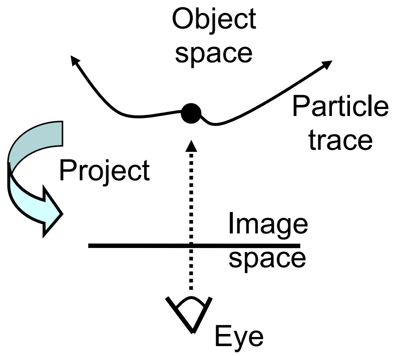
Courtesy of
Bob Laramee

Lagrangian Particle Tracing

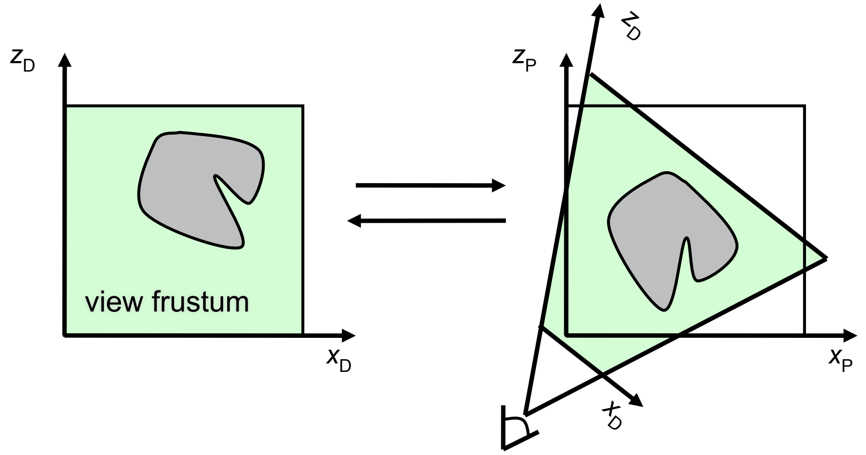
- Velocity vector **v**
- Trace massless particles
- Equation of motion: $\frac{d\mathbf{r}}{dt} = \mathbf{v}(\mathbf{r}, t)$
- First order integration: $\Delta\mathbf{r} = \Delta t \times \mathbf{v}(\mathbf{r}, t)$

Image-Space and Object-Space Aspects

- Reduce information to image space
  - Project vector field onto image plane
    - Render object geometry
    - Vector field in 3D texture or attached to vertices of the mesh
  - Starting position
- Keep some 3D object-space information
  - Current position along particle trace
  - Noise (solid texture)
- Coupling:
  - Simultaneous computation of positions in object and image spaces

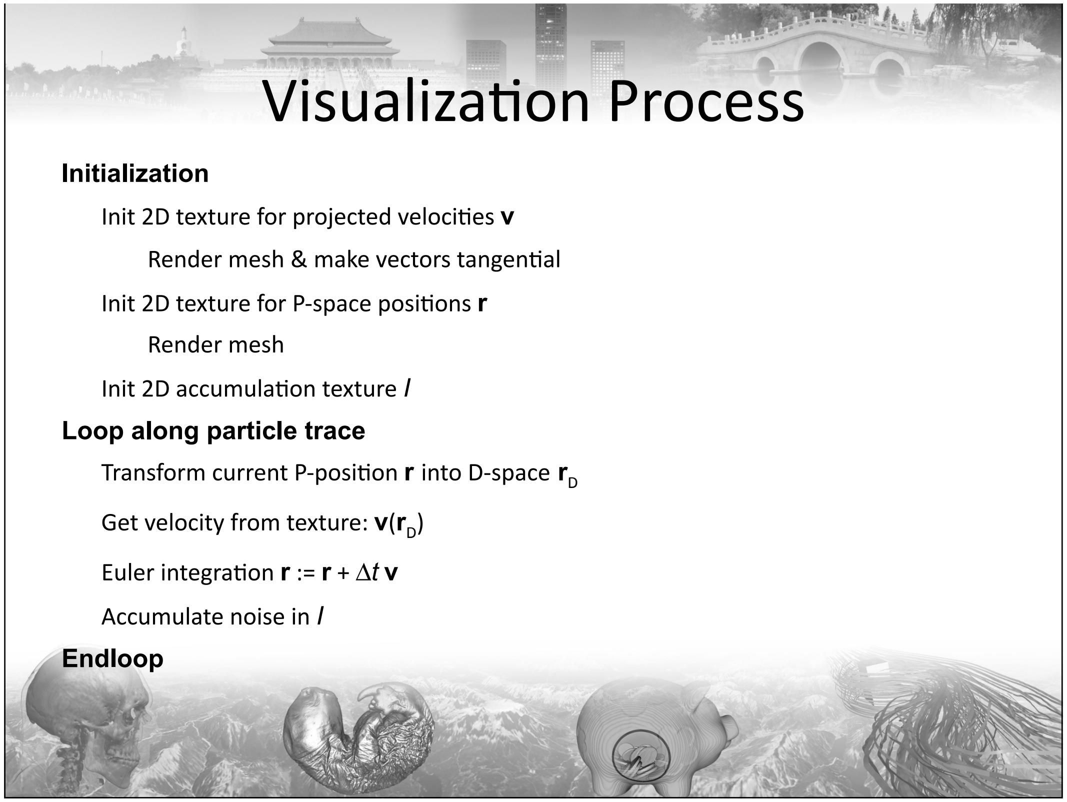


D-Space vs. P-Space for Coordinates



- Device (D) space
- On image plane

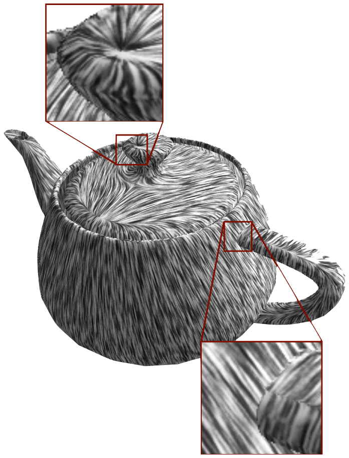
- Physical (P) space
- Object space



Visualization Process

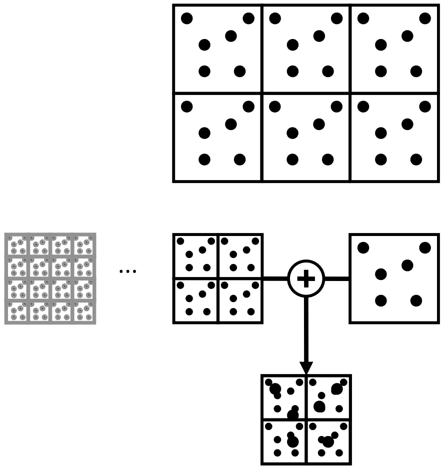
```
Initialization
    Init 2D texture for projected velocities $\mathbf{v}$
        Render mesh & make vectors tangential
    Init 2D texture for P-space positions $\mathbf{r}$
        Render mesh
    Init 2D accumulation texture $I$
Loop along particle trace
    Transform current P-position $\mathbf{r}$ into D-space $\mathbf{r}_D$
    Get velocity from texture: $\mathbf{v}(\mathbf{r}_D)$
    Euler integration $\mathbf{r} := \mathbf{r} + \Delta t\, \mathbf{v}$
    Accumulate noise in $I$
Endloop
```

Flow Across Silhouette Lines



Noise

- 3D solid texture
  - Frame-to-frame coherence
  - Large amount of texture space?
  - Aliasing?
- 3D texture repeat
  - Typically 64^3 or 128^3
  - No artifacts if surface is (slightly) curved
- Distance-based scaling of noise
  - Powers of two
  - Interpolate between two closest levels (similarly to mipmapping)
- Time dependency: van Wijk [2002]



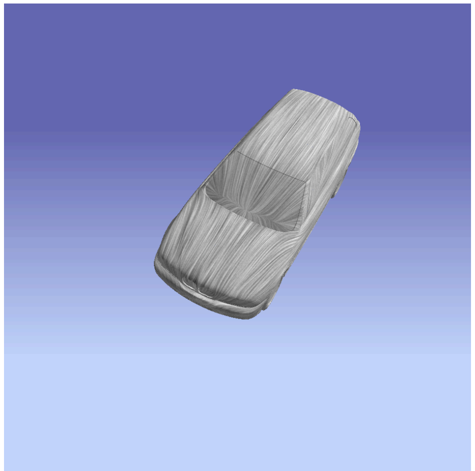
Implementation

- Mapping to GPUs
  - DirectX 9.0
  - HLSL and fx files

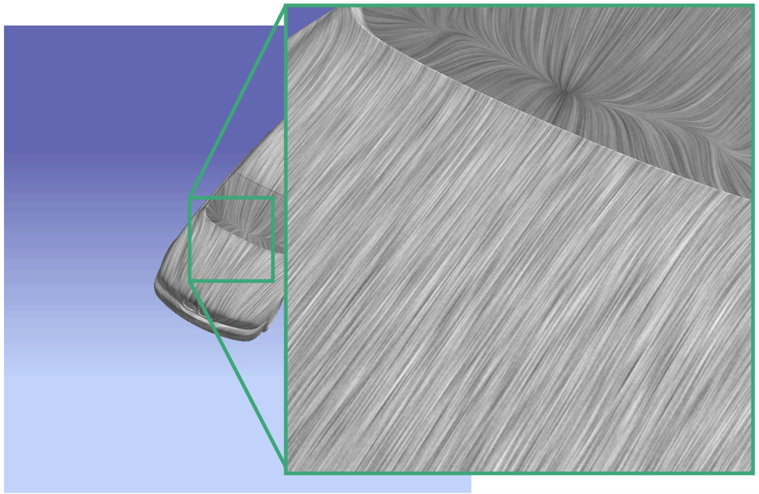
http://www.vis.uni-stuttgart.de/texflowvis

- Textures
  - 32 bit floating-point for coordinates (2D tex)
  - 16 bit fixed-point for accumulated gray values (2D tex)
  - 16 bit fixed or floating point for vector field (3D tex)
  - 8 bit fixed point for noise field (3D tex)
- Early z-test to process only visible surface elements

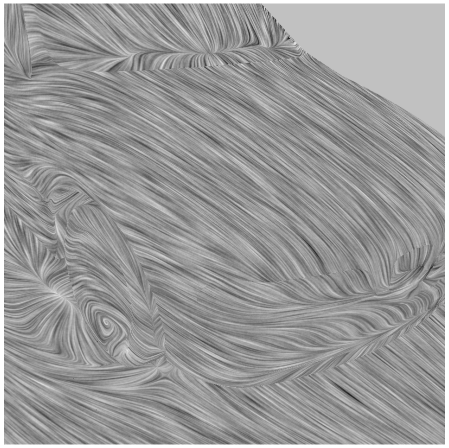
Noise Scaling



Noise Scaling

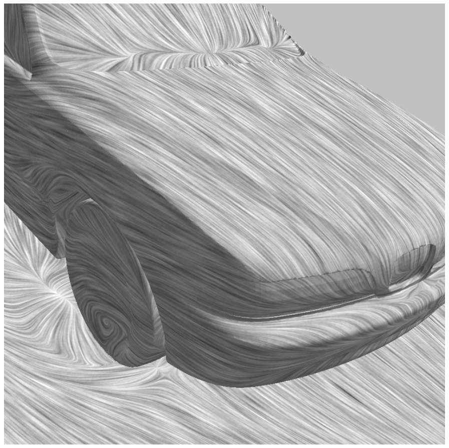


Rendering



No surface shading

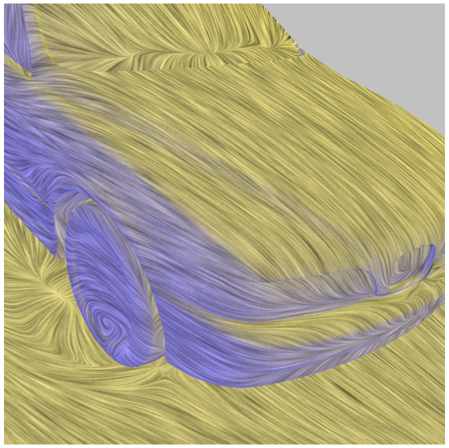
Rendering



No surface shading

Gray surface

Rendering

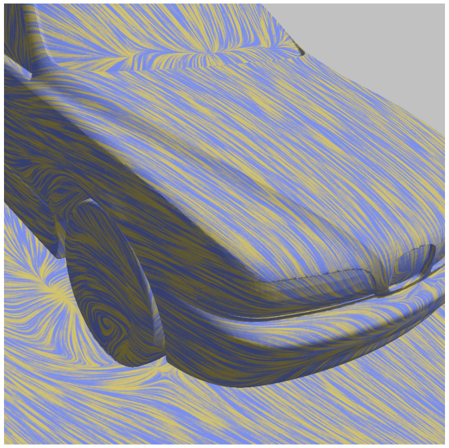


No surface shading

Gray surface

Cool/warm shading for surface

Rendering



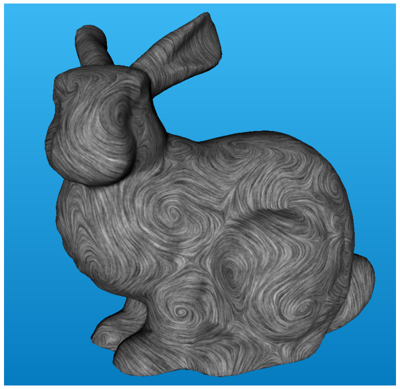
No surface shading

Gray surface

Cool/warm shading
for surface

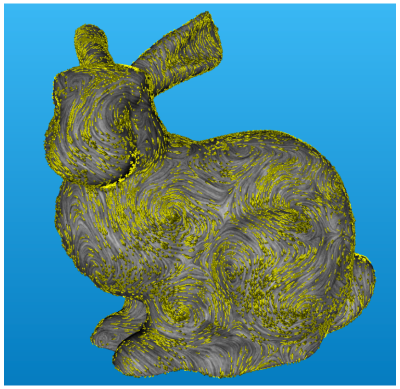
Gray surface
Yellow/blue flow

Synthesis



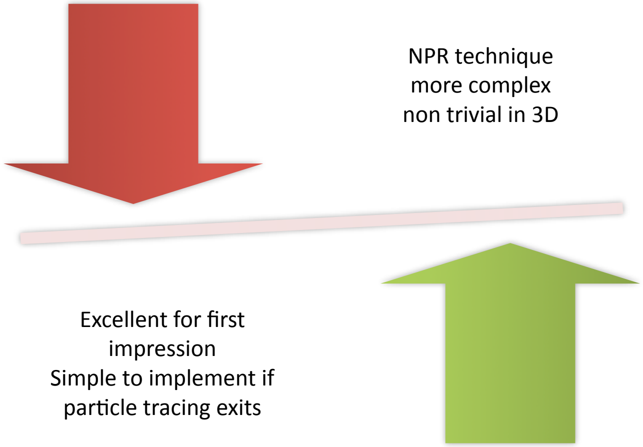
Surface dataset courtesy of Peter Schröder e.al.

Synthesis



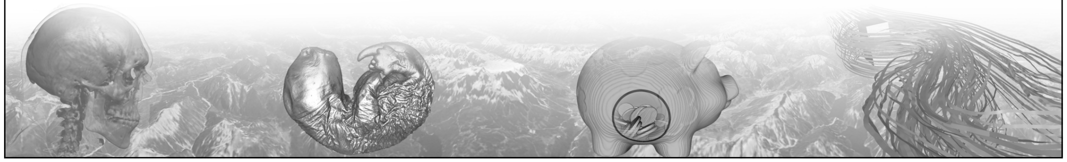
Surface dataset courtesy of Peter Schröder e.al.

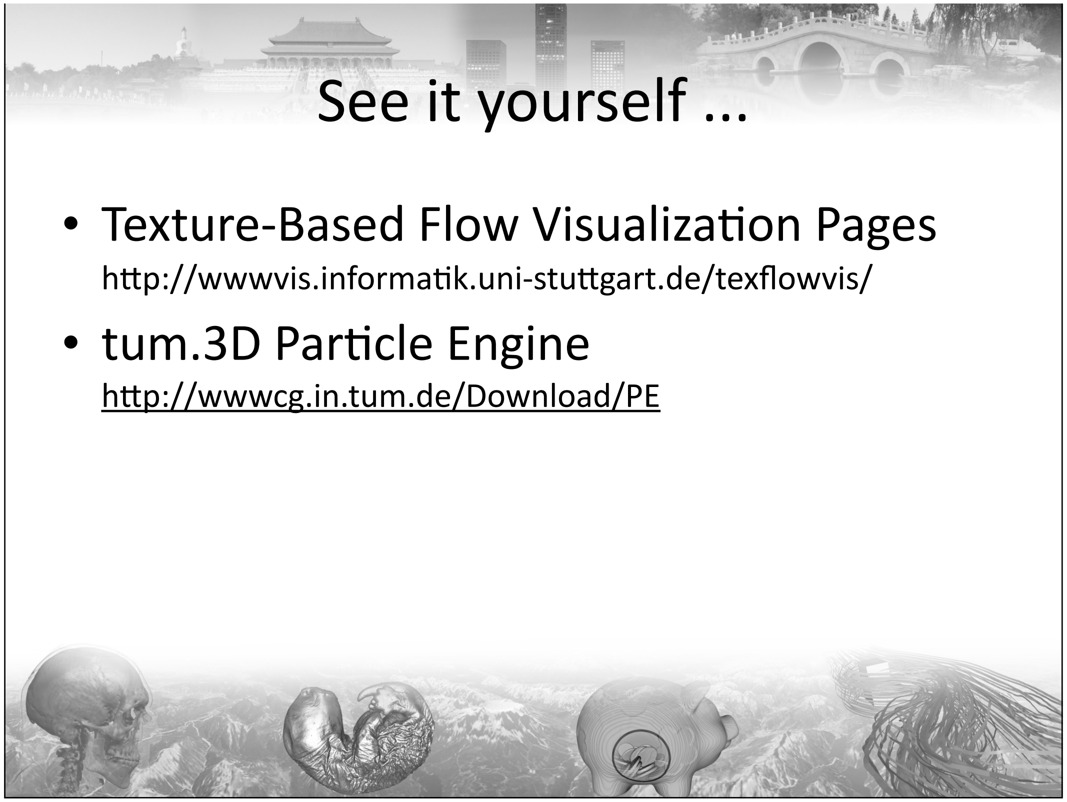
LIC



NPR technique
more complex
non trivial in 3D

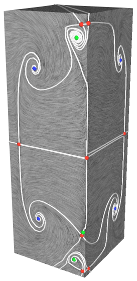
Excellent for first impression
Simple to implement if particle tracing exits





See it yourself ...

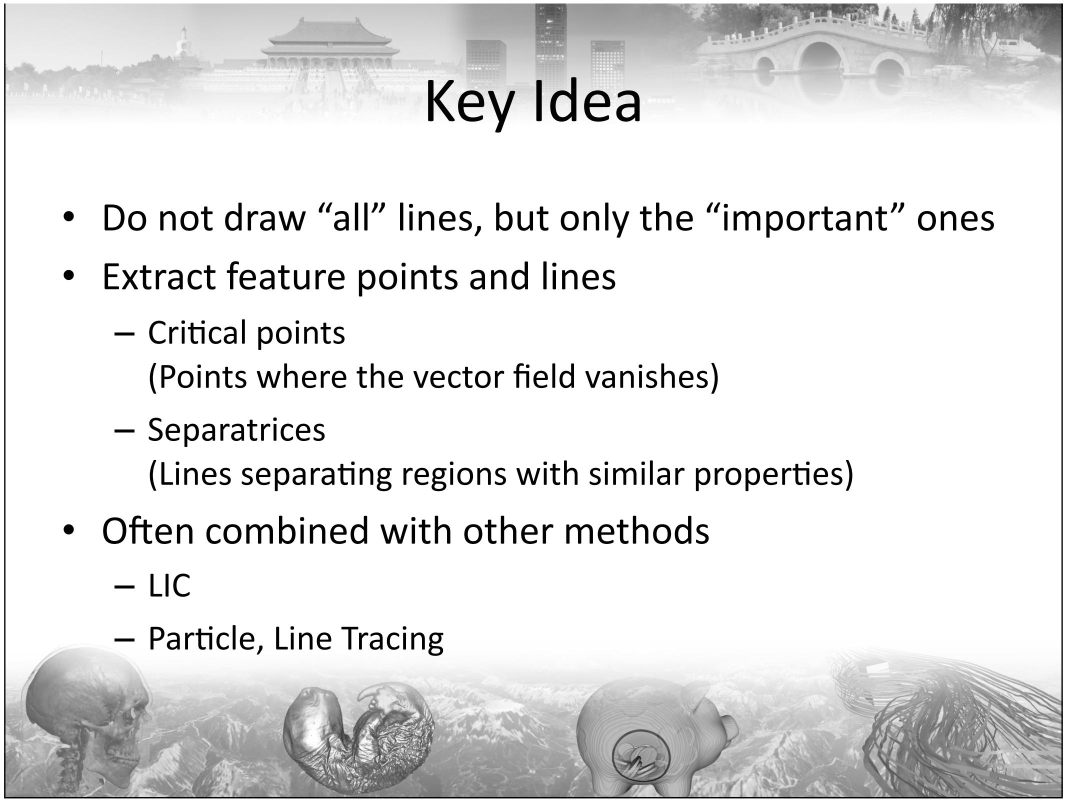
- Texture-Based Flow Visualization Pages
  http://wwwvis.informatik.uni-stuttgart.de/texflowvis/
- tum.3D Particle Engine
  http://wwwcg.in.tum.de/Download/PE



Topological Methods

Slides courtesy of

Gerik Scheuermann

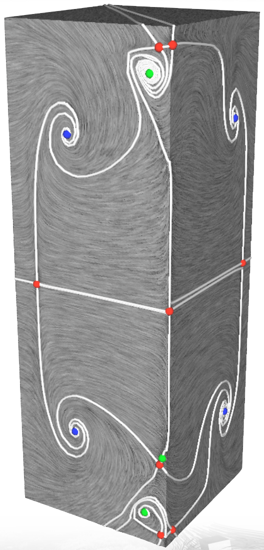


Key Idea

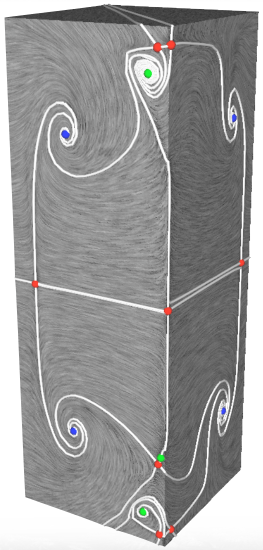
- Do not draw “all” lines, but only the “important” ones
- Extract feature points and lines
  - Critical points
    (Points where the vector field vanishes)
  - Separatrices
    (Lines separating regions with similar properties)
- Often combined with other methods
  - LIC
  - Particle, Line Tracing

Topological Methods

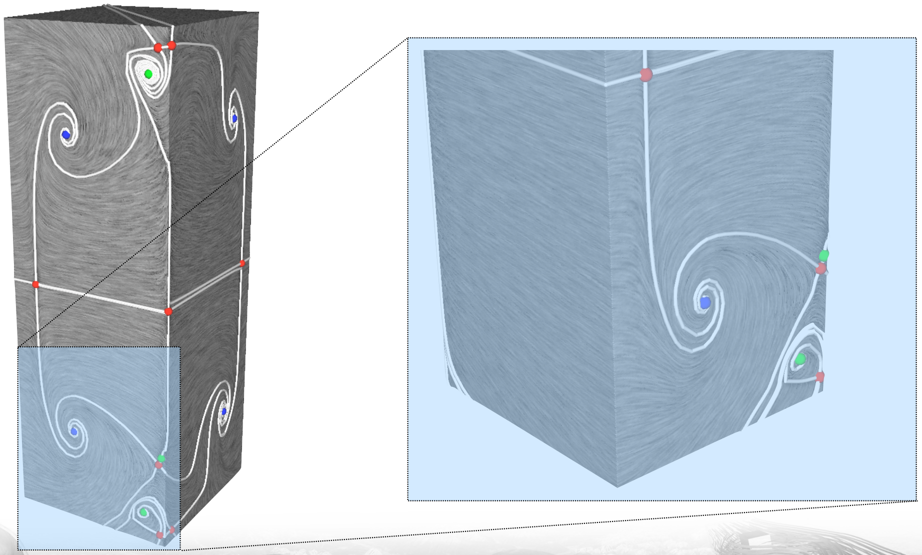
- Flow around a cube
  - Surface LIC
  - Flow topology on surface
    - attractive
    - repelling
    - saddle
    - separatrices



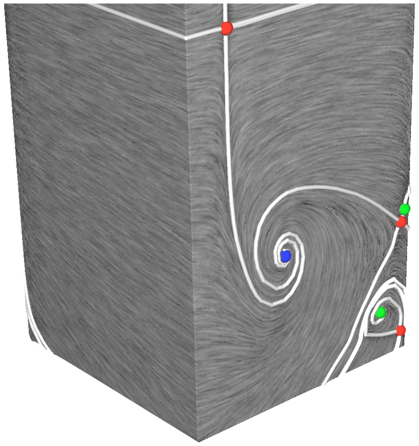
Topological Methods



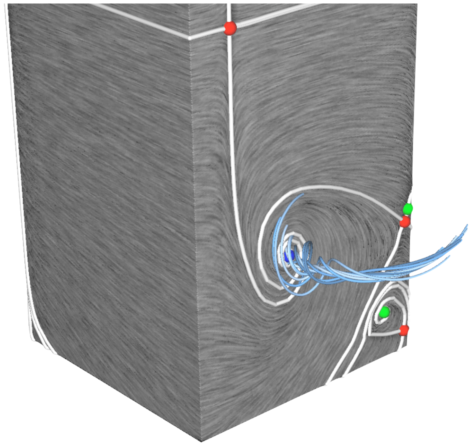
Topological Methods



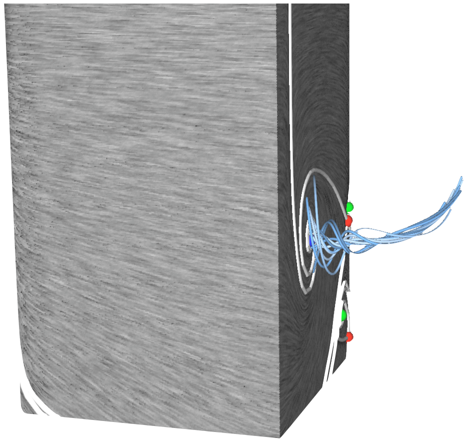
Topological Methods



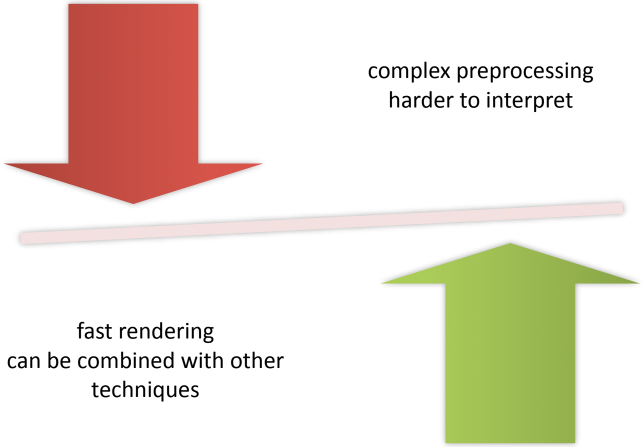
Topological Methods



Topological Methods

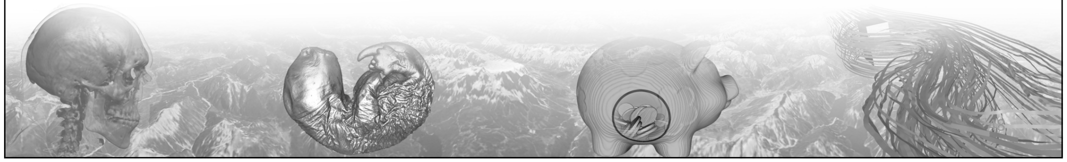


Topological Methods



complex preprocessing
harder to interpret

fast rendering
can be combined with other techniques



References

GPU Particles and Lines

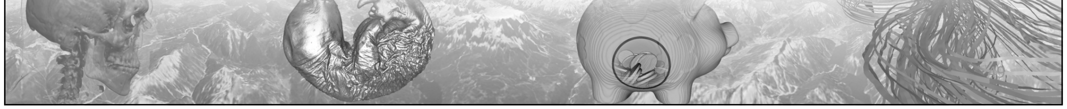
- **UberFlow: A GPU-Based Particle Engine**
  P. Kipfer, M. Segal, R. Westermann, Graphics Hardware 2004
- **A Particle System for Interactive Visualization of 3D Flows**
  J. Krüger, P. Kipfer, P. Kondratieva, R. Westermann, IEEE Transactions on Visualization and Computer Graphics Vol. 11, No. 6
- **Interactive Visual Exploration of Unsteady 3D Flows**
  K. Bürger, J. Schneider, P. Kondratieva, J. Krüger, R. Westermann, EuroVis 2007
- **Importance-Driven Particle Techniques for Flow Visualization**
  K. Bürger, P. Kondratieva, J. Krüger, R. Westermann, Pacific Vis 2008
- **Smoke Surfaces: An Interactive Flow Visualization Technique Inspired by Real-World Flow Experiments**, W. von Funck, T. Weinkauf, H. Theisel and H.-P. Seidel, In Proc. IEEE Visualization 2008, 2008

LIC

- **Image based flow visualization**
  Jarke J. van Wijk, ACM Trans. Graphics 2002
- **Texture Based Flow Vis**
  Daniel Weiskopf http://wwwvis.informatik.uni-stuttgart.de/texflowvis/

Topological Methods

- **The State of the Art in Flow Visualisation: Feature Extraction and Tracking**
  Frits H. Post , Benjamin Vrolijk , Helwig Hauser , Robert S. Laramee and Helmut Doleisch
  http://www.cs.swan.ac.uk/~csbob/research/star/post03state.pdf



The end!

Any questions?