*Simulation in Computer Graphics*

# *Image-Space Collision Detection*

Matthias Teschner

Computer Science Department
University of Freiburg

# *Acknowledgements*

- Parts of this slide set are courtesy of Bruno Heidelberger.

# Outline

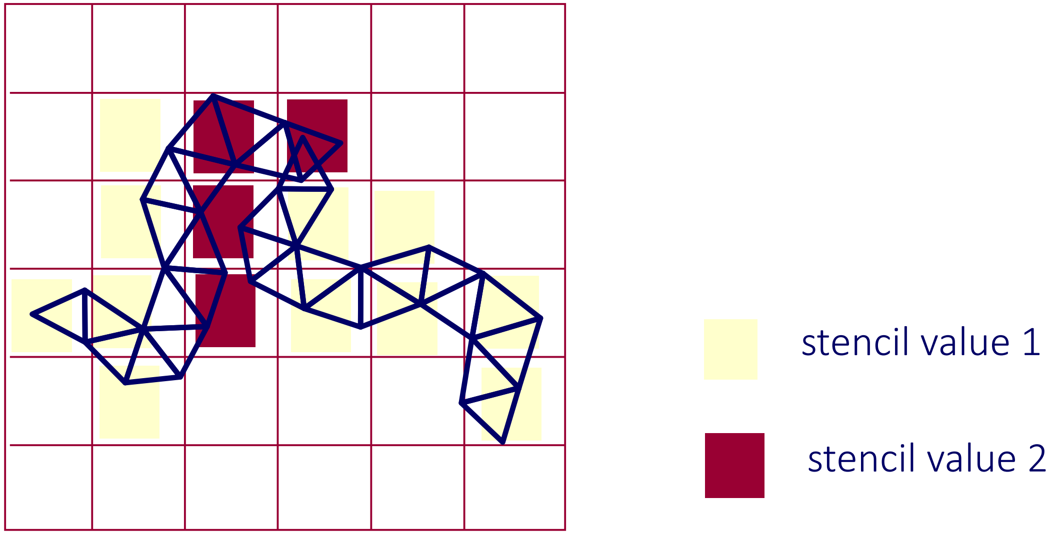
- motivation
- algorithms
- performance
- application
- discussion

# Graphics Hardware for 2D Collision Detection

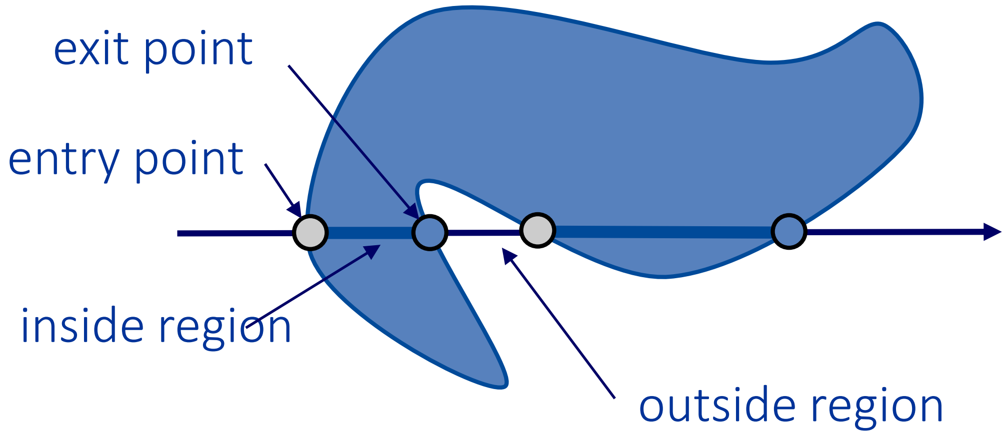
- rendering corresponds to placing all object primitives into the according cell (pixel) of a uniform rectangular 2D grid (frame buffer)
- rendering determines all pixels in a frame buffer affected by the object
- at each pixel position, information can be processed (color, depth, stencil)

# Graphics Hardware for 2D Collision Detection

- [Kenneth Hoff, UNC]
- stencil-buffer for collision detection
- clear stencil buffer
- increment stencil buffer for each rendered object
- intersection for stencil buffer value larger 1



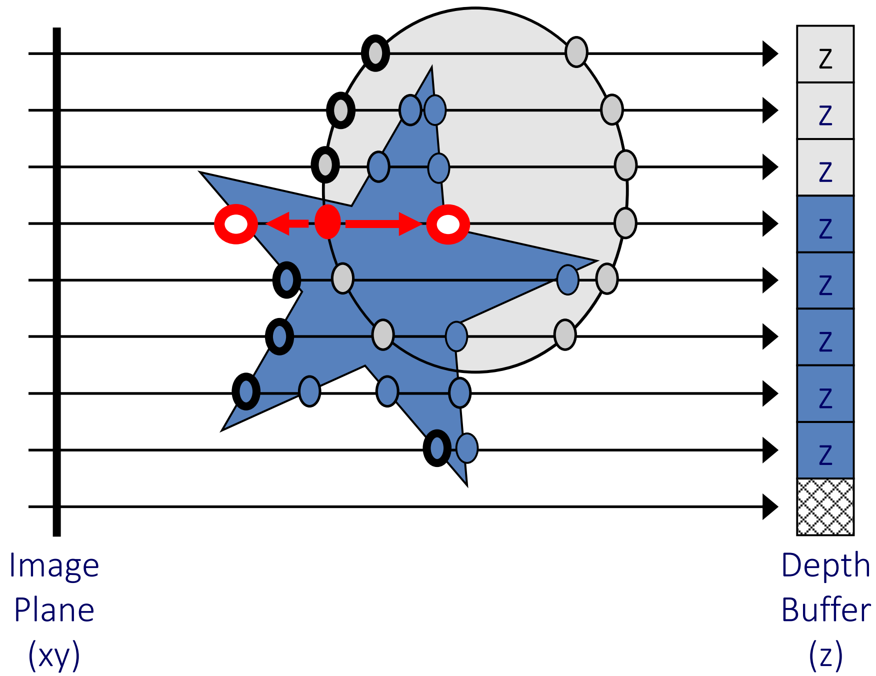
# Closed Objects




- number of entry points equals the number of exit points
- in case of convex objects, one entry point and one exit point
- inside and outside are separated by entry or exit point
- entry point is at a front face
- exit point is at a back face
- front and back faces alternate

# Collision Detection with Graphics Hardware

- exploit rasterization of object primitives for intersection test
- benefit from graphics hardware acceleration

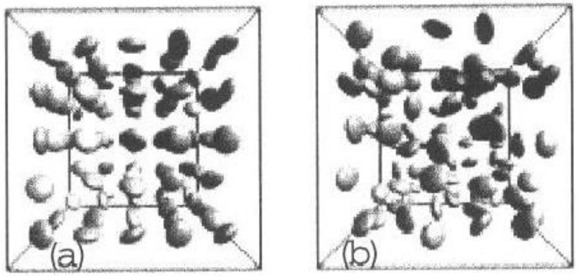


# Collision Detection with Graphics Hardware

- Idea
  - computation of entry and exit points can be accelerated with graphics hardware
  - computation corresponds to rasterization of surface primitives
  - all object representations that can be rendered are handled
  - parallel processing on CPU and GPU
- Challenges
  - restricted data structures and functionality
- Drawbacks
  - approximate computation of entry and exit points

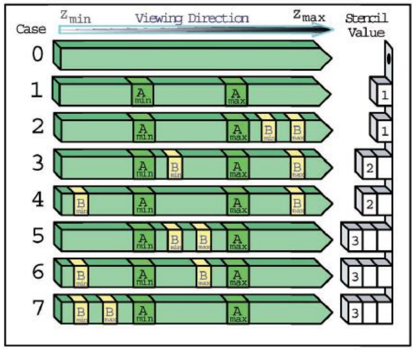
# *Early approaches*

[Shinya, Forgue 1991]
image-space collision detection for
*convex objects*



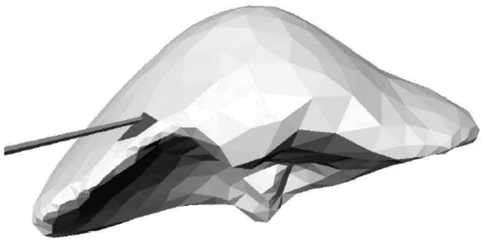

[Myszkowski, Okunev, Kunii 1995]
collision detection for *concave objects*
*with limited depth complexity*

[Baciu, Wong 1997]
hardware-assisted collision detection for
*convex objects*

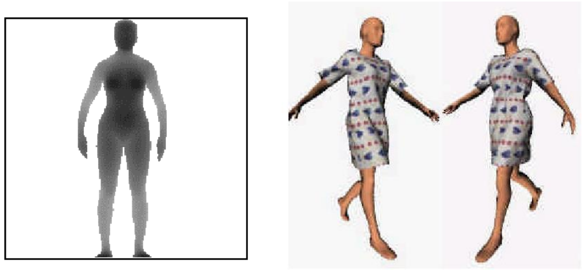


# *More approaches*

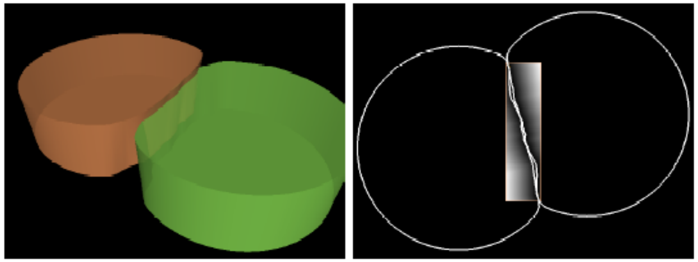
[Lombardo, Cani, Neyret 1999]
intersection of *tool with deformable tissue*
by rendering the interior of the tool



[Vassilev, Spanlang, Chrysanthou 2001]
image-space collision detection applied to
cloth simulation and *convex avatars*

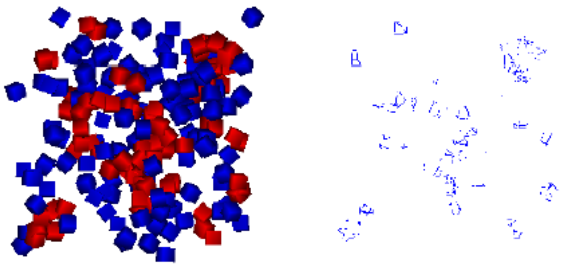


[Hoff, Zaferakis, Lin, Manocha 2001]
proximity tests and penetration
depth computation, *2D*

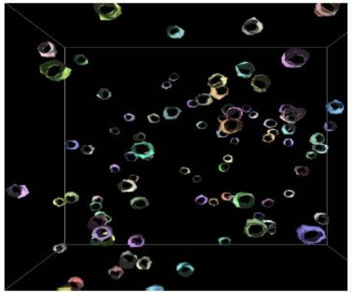


# Recent approaches

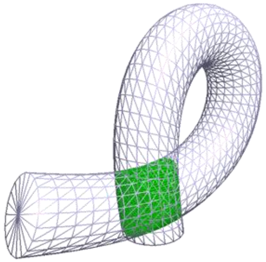
[Knott, Pai 2003]
intersection of edges with surfaces



[Govindaraju, Redon, Lin, Manocha 2003]
object and sub-object pruning based on
occlusion queries

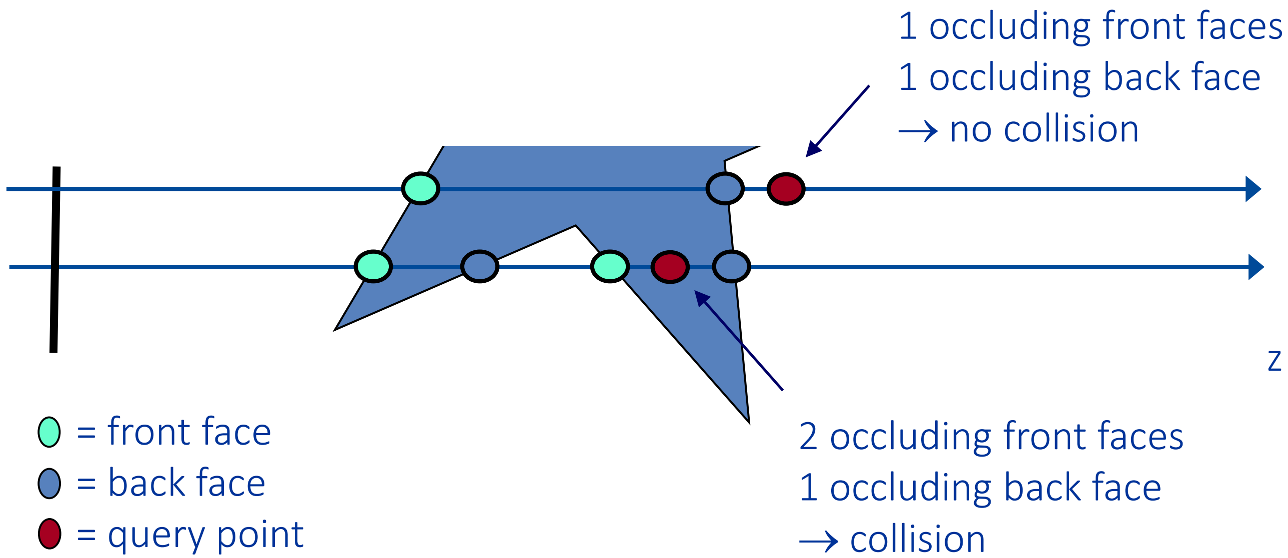


[Heidelberger, Teschner 2004]
explicit intersection volume and
self-collision detection based on LDIs



# Image-Space Collision Detection [Knott, Pai 2003]

- render all query objects (e. g. edges) to depth buffer
- count the number f of front faces that occlude the query object
- count the number b of back faces that occlude the query object
- iff f - b == 0 then there is no collision



# Image-Space Collision Detection

- clear depth buffer, clear stencil buffer
- render query objects to depth buffer
- disable depth update
- render front faces with stencil increment
- if front face is closer than query object, then increment stencil
- depth buffer is not updated
- result: stencil represents number of occluding front faces
- render back faces with stencil decrement
- if back face is closer than query object, then decrement stencil
- depth buffer is not updated
- result: stencil represents diff. of occluding front and back faces
- stencil buffer not equal to zero → collision

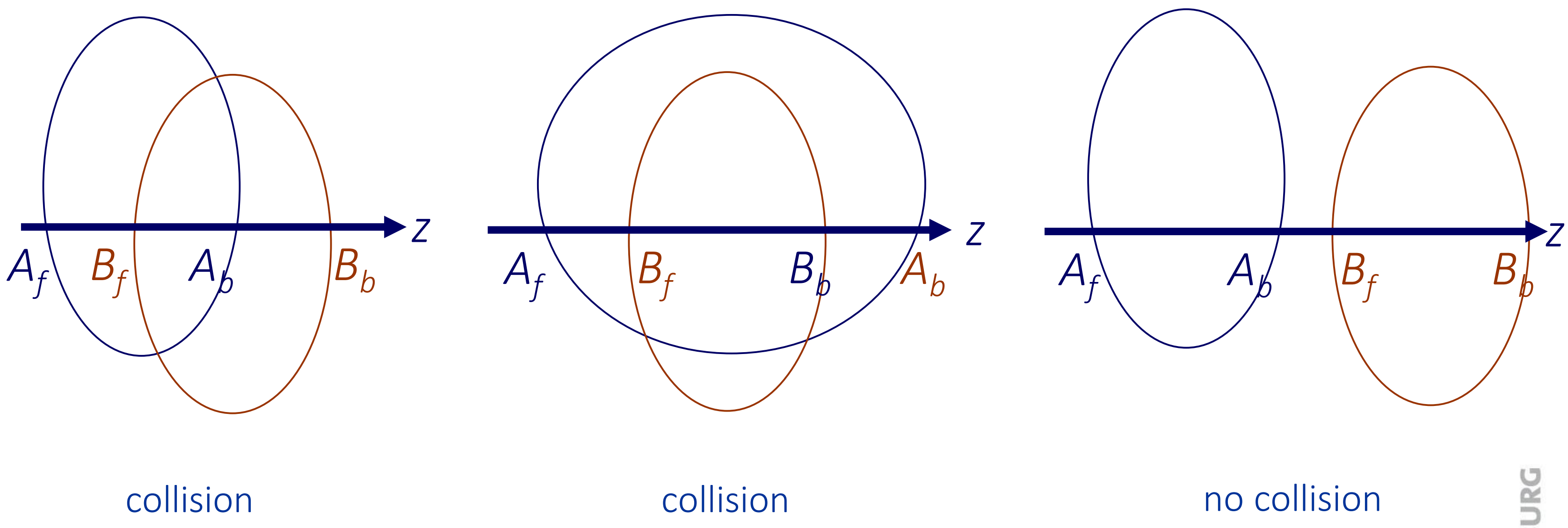
# Image-Space Collision Detection

- works for objects with closed surface
- works for n-body environments
- works for query objects that do not overlap in image space
- numerical problems if query object is part of an object
- offset in z-direction required

- [Video]

# *Image-Space Collision Detection [Baciu 2000]*

- RECODE – REndered COllision DEtection
- works with pairs of closed convex objects A and B
- one or two rendering passes for A and B
- algorithm estimates overlapping z intervals per pixel



# First Rendering Pass

- clear depth buffer
- clear stencil buffer
- enable depth update
- render back faces of A with stencil increment
- if nothing has been rendered → stencil=0
- if something has been rendered → stencil=1
- depth buffer contains depth of back faces of A
- disable depth update
- render B with stencil increment
- if stencil==1 and B occludes back face of A → stencil+=1
- depth buffer is not updated
- stencil-1 = number of faces of B that occlude A

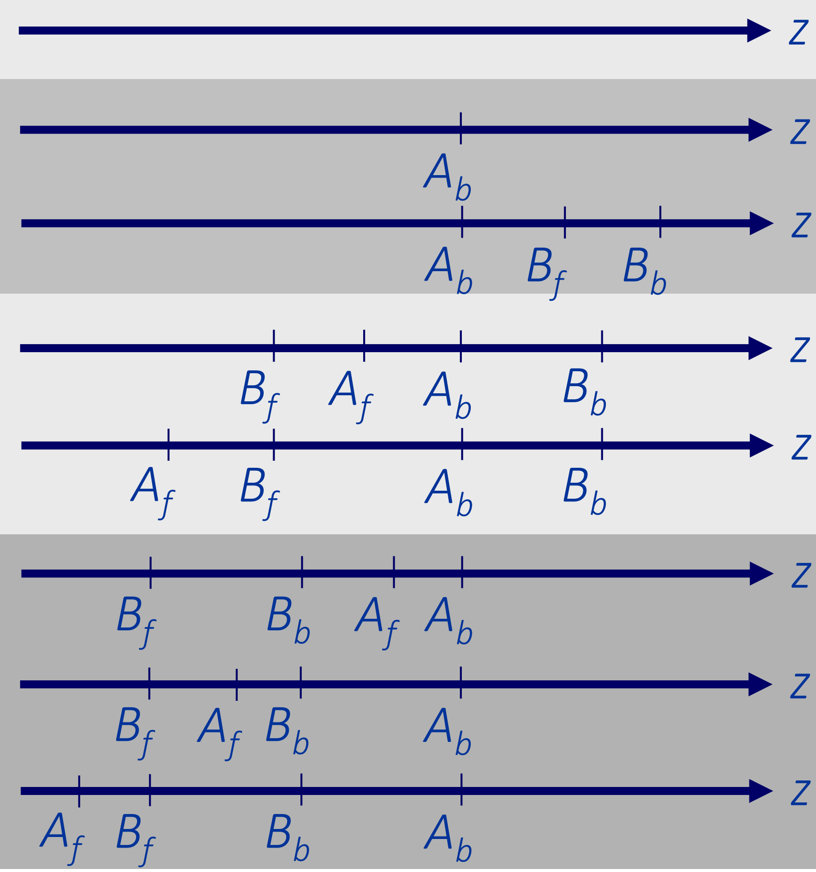


UNI FREIBURG

# First Rendering Pass

- first pass collision query

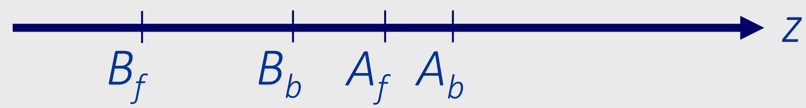
- stencil 0 $\rightarrow$ no collision
- stencil 1 $\rightarrow$ no collision
  - no fragment of B occludes back face of A (2 cases)
- stencil 2 $\rightarrow$ collision
  - front face of B occludes back face of A (2 cases)
- stencil 3 $\rightarrow$ second pass
  - front and back face of B occlude back face of A (3 cases)



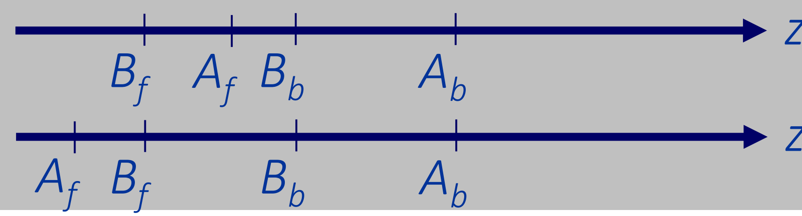
# Second Rendering Pass

- render back faces of object B, count occluding faces of A
  - corresponds to first pass with A and B permuted
  - only 3 cases based on the result of the first rendering pass

- stencil 1 → no collision
  - no fragment of A occludes back face of B (1 case)




- stencil 2 → collision
  - front face of A occludes back face of B (2 cases)

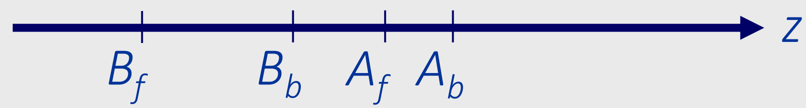



- done

# Second Rendering Pass [Myszkowski 1995]

- render front faces of object A, count occluding faces of B
  - corresponds to first pass, front faces are rendered instead of back faces
  - only 3 cases based on the result of the first rendering pass

- stencil 3 $\rightarrow$ no collision
  - front and back face of B occlude front face of A




- stencil 2 $\rightarrow$ collision
  - front face of B occludes front face of A


- stencil 1 $\rightarrow$ collision
  - no fragment of B occludes front face of A


- done

# *Image-Space Collision Detection for Concave Objects [Myszkowski 1995]*

- collision detection for pairs of concave objects A and B with limited depth complexity (number of entry / exit points)
- faces have to be sorted with respect to the direction of the orthogonal projection (e. g. BSP tree)

- objects are rendered in front-to-back or back-to-front order
- alpha blending is employed:

$$\text{color}_{\text{framebuffer}} = \text{color}_{\text{object}} + \alpha \cdot \text{color}_{\text{framebuffer}}$$

- color of A is zero, color of B is $2^{k-1}$, $k$ is the number of bits in the frame buffer, $\alpha = 0.5$



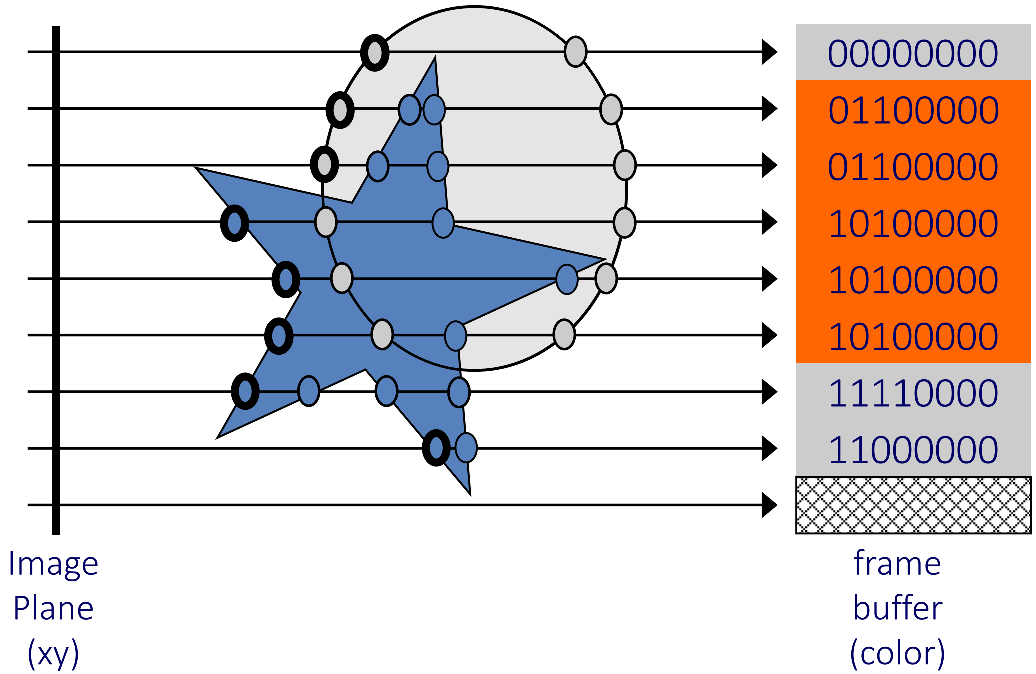
UNI FREIBURG

# Image-Space Collision Detection for Concave Objects

- example: $k = 8$
- color A = 0, color B = $2^7$
- sequence of faces B1 $A_1$ $A_2$ $B_2$ $B_3$ $B_4$ rendered back to front:
- $c_{fb} = 00000000_2$
- render $B_4$: $c_{fb} = 2^7 + \alpha \cdot c_{fb} = 10000000_2 + 0.5 \cdot 00000000_2 = 10000000_2$
- render $B_3$: $c_{fb} = 10000000_2 + 0.5 \cdot 10000000_2 = 11000000_2$
- render $B_2$: $c_{fb} = 10000000_2 + 0.5 \cdot 11000000_2 = 11100000_2$
- render $A_2$: $c_{fb} = 00000000_2 + 0.5 \cdot 11100000_2 = 01110000_2$
- render $A_1$: $c_{fb} = 00000000_2 + 0.5 \cdot 01110000_2 = 00111000_2$
- render $B_1$: $c_{fb} = 10000000_2 + 0.5 \cdot 00111000_2 = 10011100_2$
- resulting bit sequence represents order of faces of A (0) and B (1)
- odd number of adjacent zeros or ones indicates collision

# Image-Space Collision Detection for Concave Objects

- example

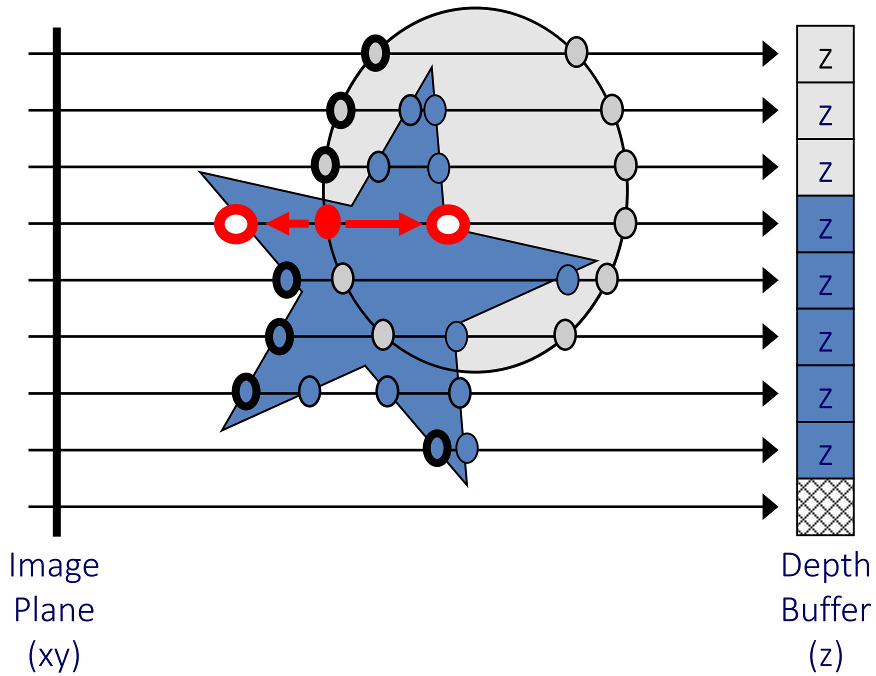


# *Image-Space Collision Detection [Heidelberger 2003]*

- works with pairs of closed arbitrarily-shaped objects
- three implementations
  - n+1 hardware-accelerated rendering passes
    where n is the depth complexity of an object
  - n hardware-accelerated rendering passes
  - 1 software rendering pass
- three collision queries
  - intersection volume (based on intersecting z intervals)
  - vertex-in-volume test
  - self-collision test
- basic idea and implementation for convex objects has been proposed by Shinya / Forgue in 1991

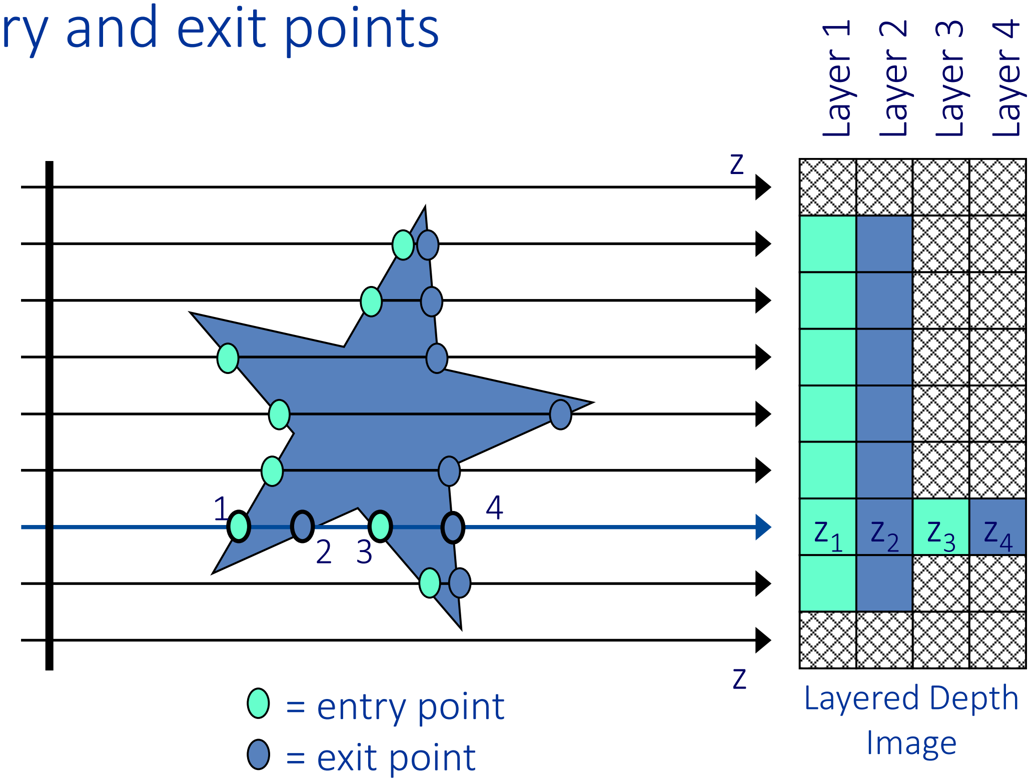
# Collision Detection with Graphics Hardware

- exploit rasterization of object primitives for intersection test
- benefit from graphics hardware acceleration



# Layered Depth Image

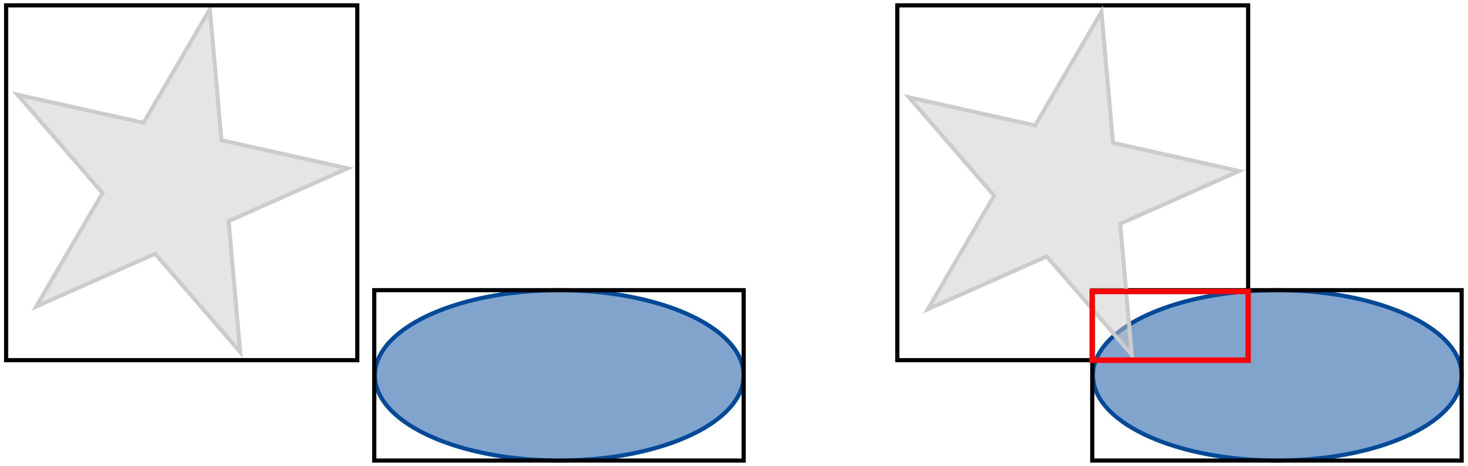
- compact, volumetric object representation [Shade et al. 1998]
- represents object as layers of depth values
- stores entry and exit points



# Algorithm Overview

Algorithm consists of 3 stages:

Stage 1: Check for bounding box intersection

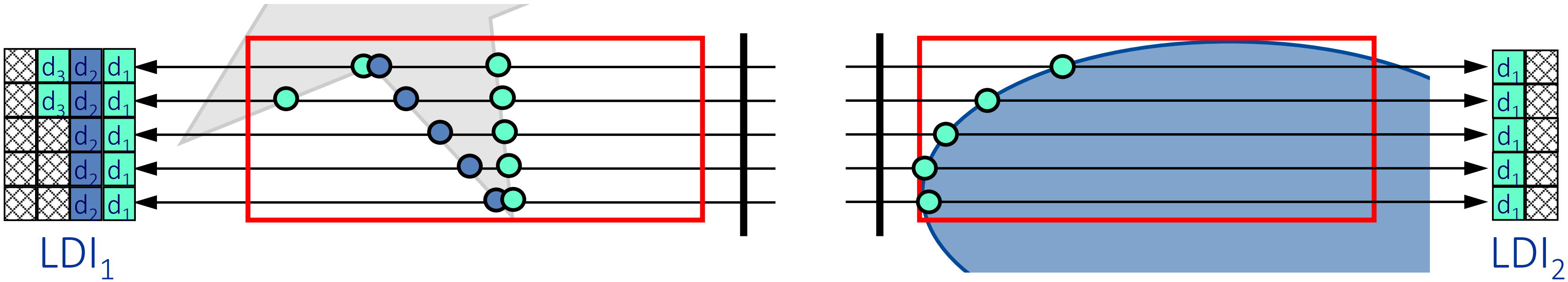


a) Very fast detection of trivial “no collision” cases

b) Overlapping area defines volume of interest (VoI) for step 2 & 3

# Algorithm Overview

Stage 2: Generate the layered depth images (LDI)




Step 3: Perform the collision tests

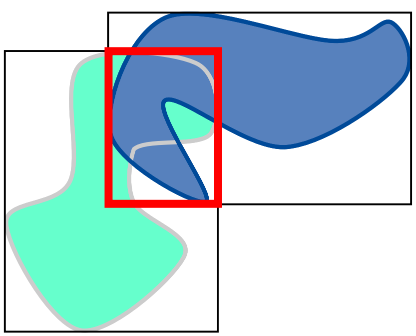
a) test object primitives of one object against LDI of the other

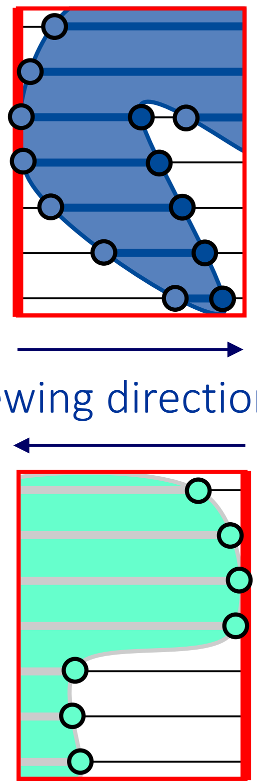
b) combine both LDI to get overlapping volume

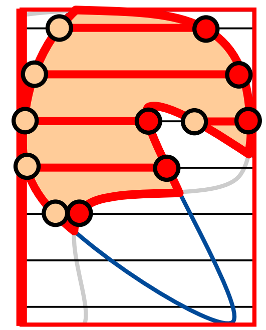
c) self-intersection test

# Algorithm Overview

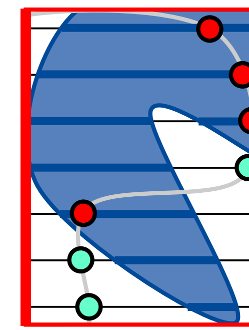
| Stage 1 | Stage 2 | Stage 3 |
|---|---|---|
| Volume-of-interest | LDI generation | Collision query |



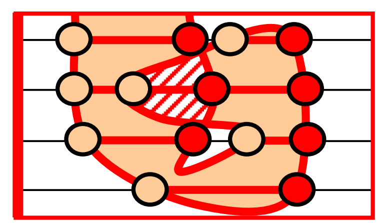


a) LDI intersection

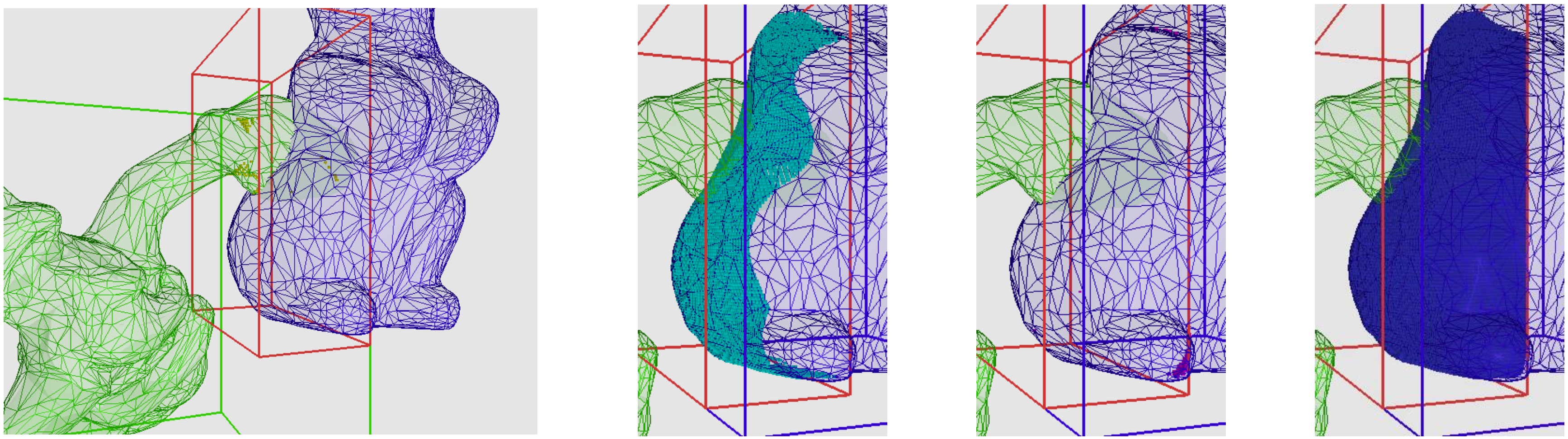


b) Vertex-in-volume



c) Self-collision

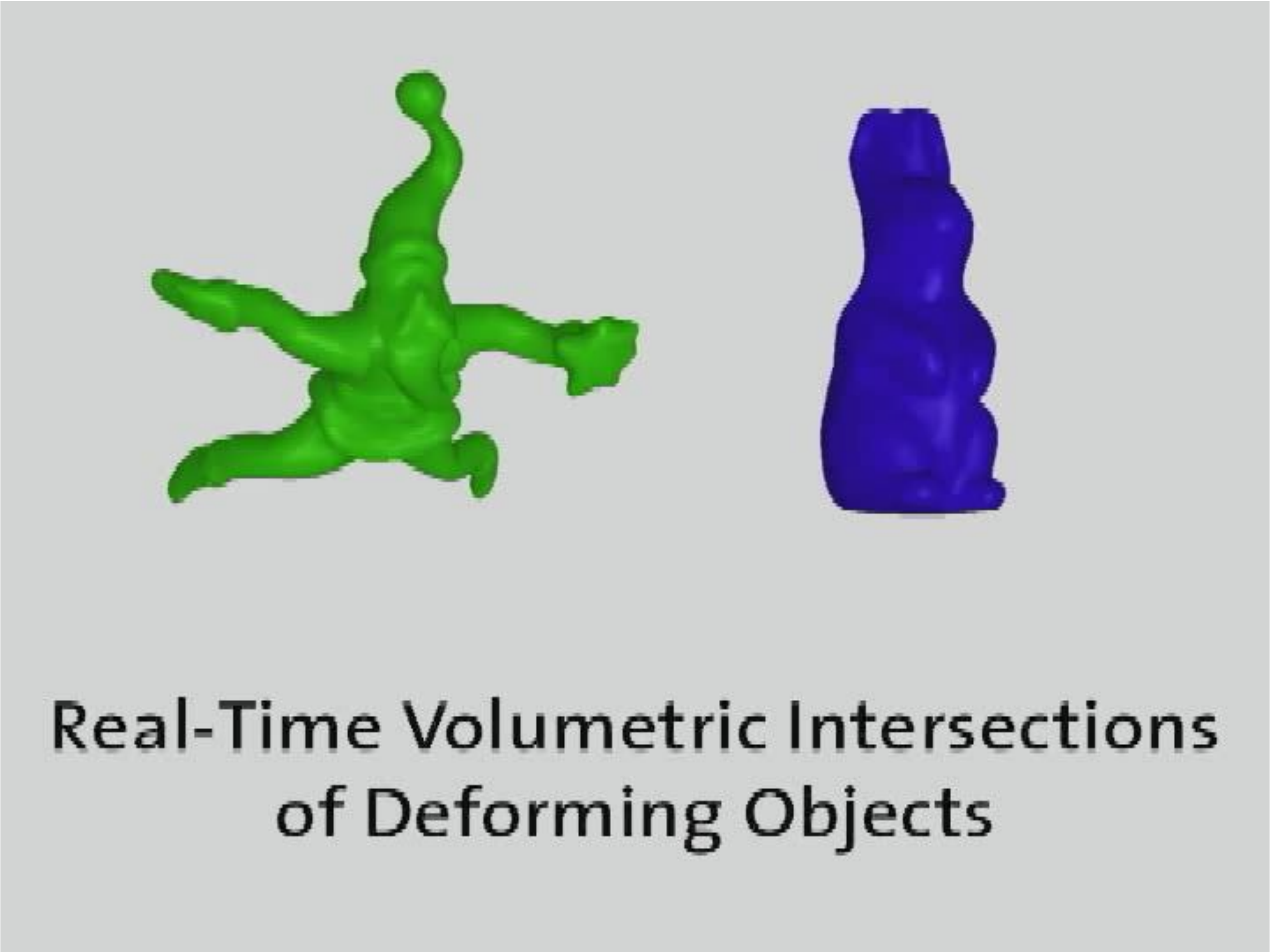
# Algorithm Overview



volume of interest VoI | layer 1 | layer 2 | volume

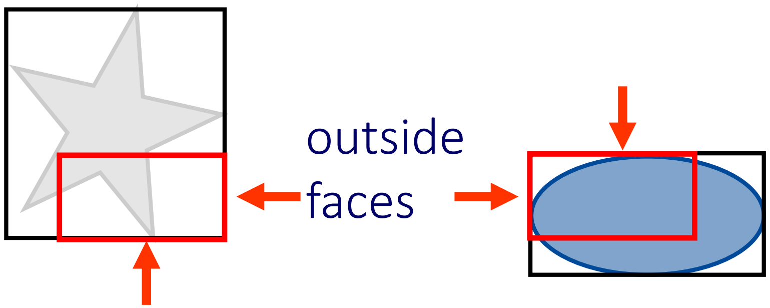
collision queries

# Algorithm Overview

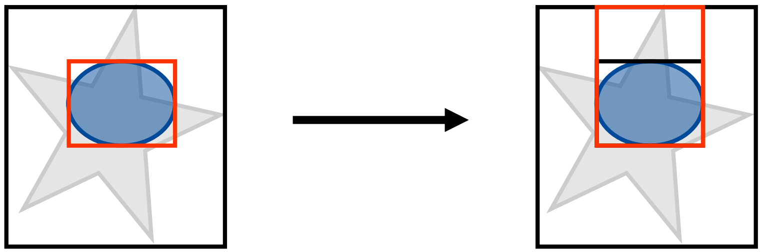


# Volume of Interest

1. evaluation of trivial rejection test: VoI == Ø → no collision!
2. choice of *opposite* render directions for LDI generation




possible enlargement of VoI to guarantee valid directions



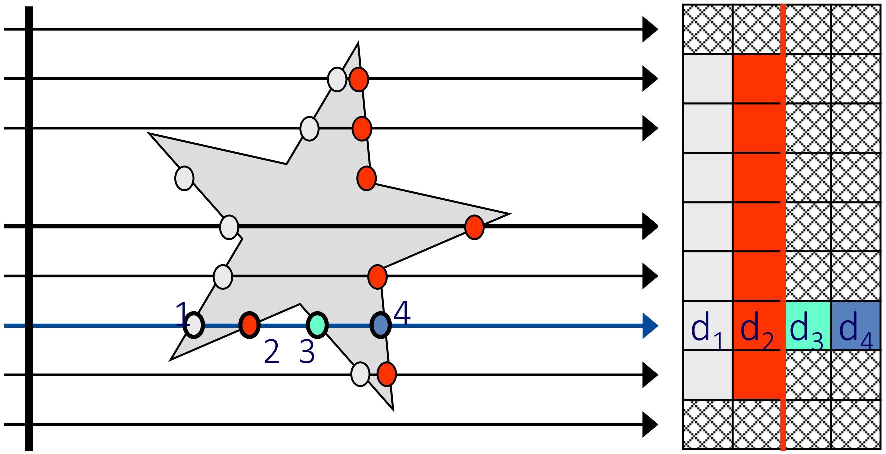
**outside faces are outside the object**

**-> guarantees that first intersection point is an entry point**

# LDI Generation on the GPU
# Depth Peeling

- object is rendered once for each layer in the LDI
- two separate depth tests per fragment are necessary:
- fragment must be farther than the one in the previous layer (d2)
- fragment must be the nearest of all remaining fragments (d3 & d4)

example: pass #3




→ second depth test is realized using shadow mapping
extended depth-peeling approach [Everitt 2001]

# Shadow Mapping

Idea:

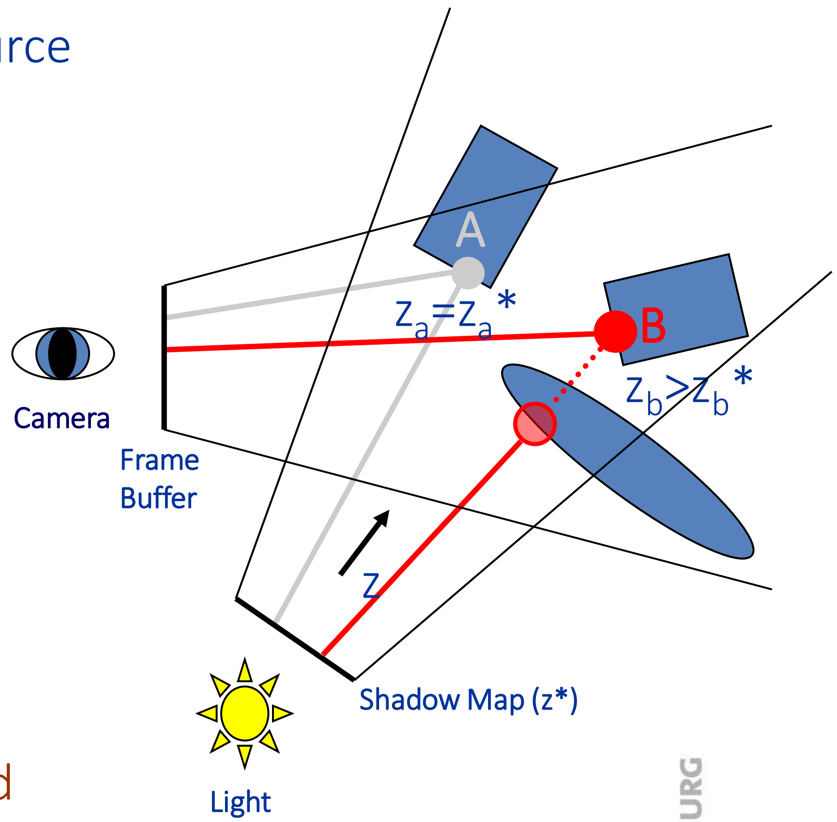
- for each fragment to be rendered:
  check if it is visible from the light source

Algorithm:

- render scene from the light source:
  store all distances to the visible (=lit) fragments in a "shadow map"
- render scene from the camera:
  compare the distance *z* of each fragment to the light with the value *z** in the shadow map:

$z = z^* \rightarrow$ fragment is lit

$z > z^* \rightarrow$ fragment is shadowed

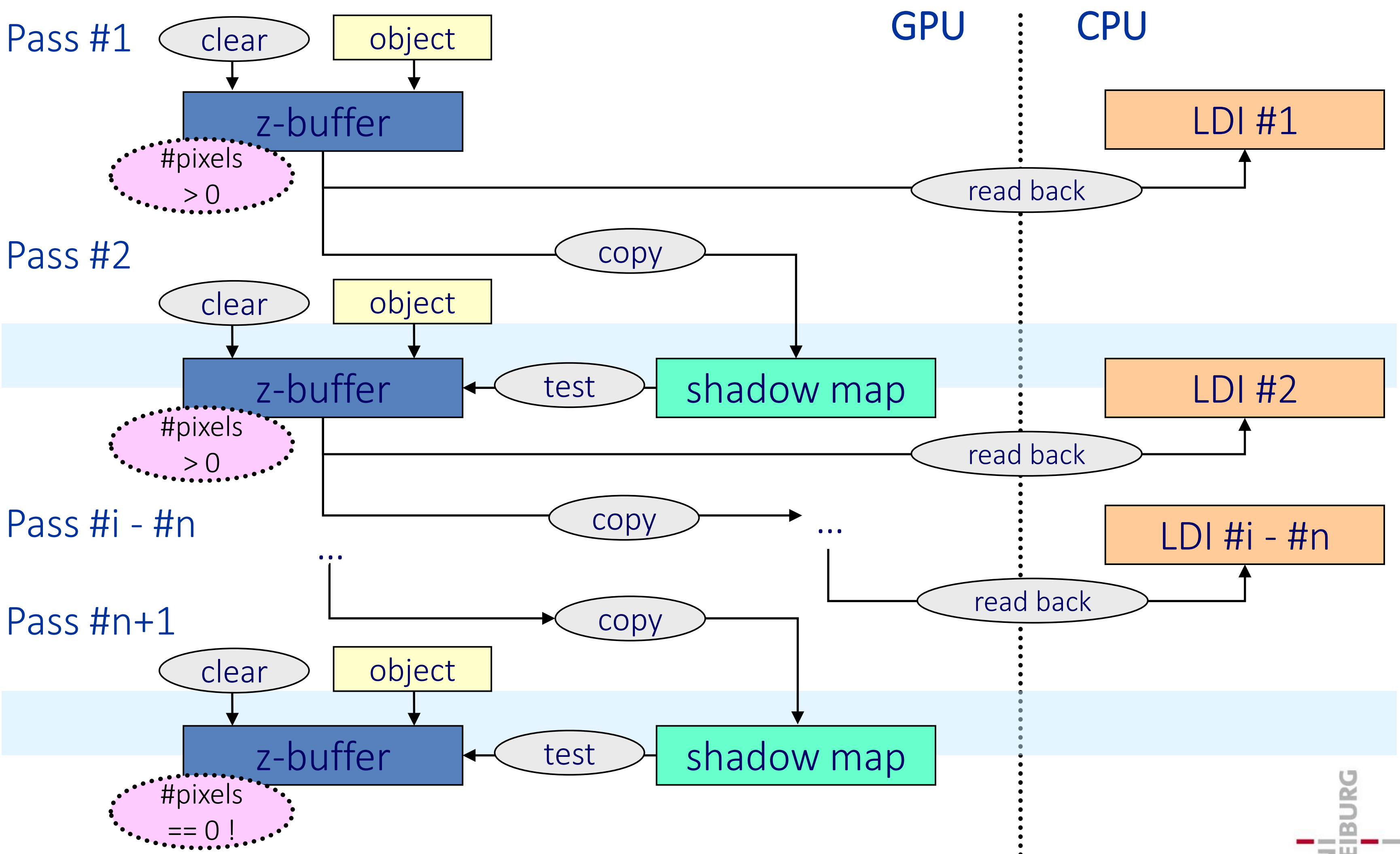


# *Shadow Mapping as Depth Test*

- Differences to regular depth test:
  - shadow mapping depth test is not tied to camera position
  - shadow map (depth buffer) is *not writeable during depth test*
  - shadow mapping *does not discard fragments*

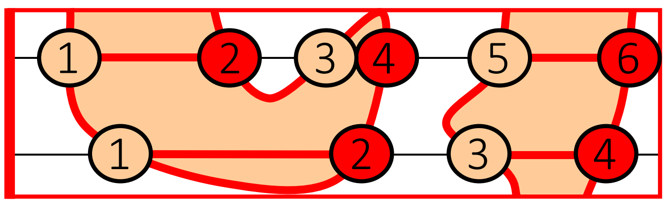
- Depth test setup for LDI generation:
  - fragment must be **farther** away than fragment in previous depth layer → shadow map test
  - fragment must be **the nearest** of all remaining fragments → regular depth test

# Multipass LDI Generation



# *Result of LDI Generation*

- multipass LDI generation results in an ordered LDI representation of the VoI




VoI

| | | | | | |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 2 | 3 | 4 | | |

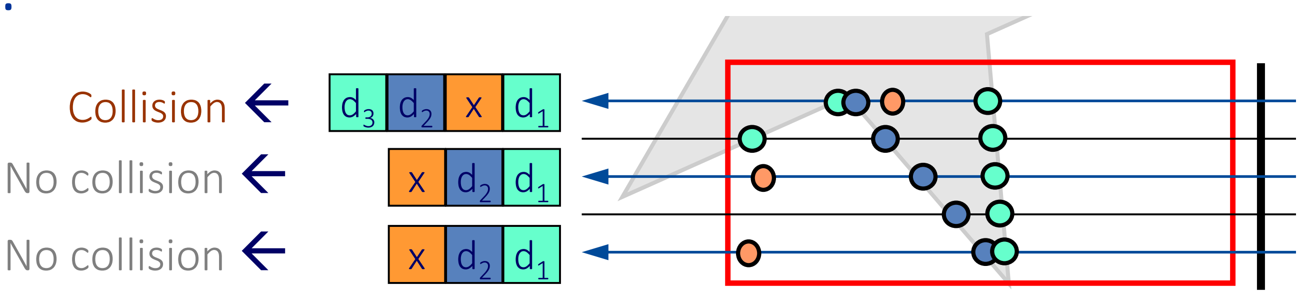
ordered LDI

- requires one rendering pass per depth layer
- requires shadow mapping functionality

# Collision Detection Test

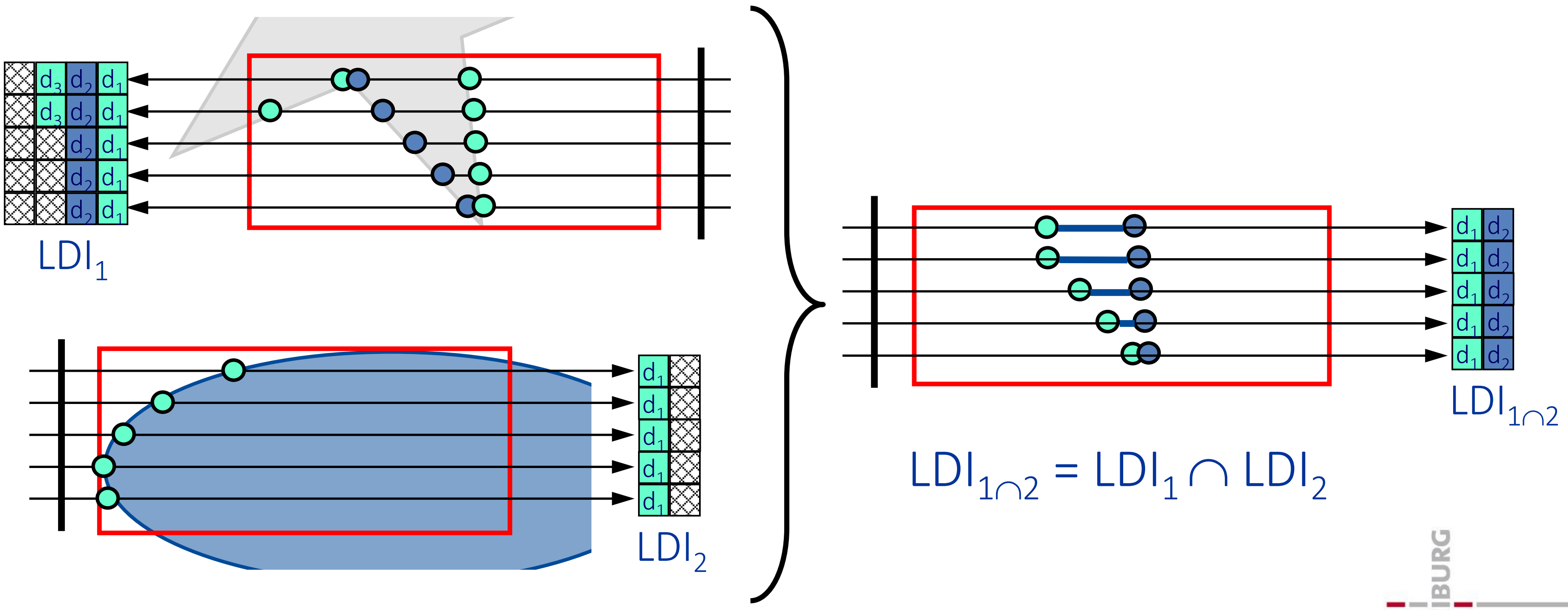
- test object primitives of one object against LDI of the other object (and vice versa)
- vertex-in-volume test

example:



# LDI Combination

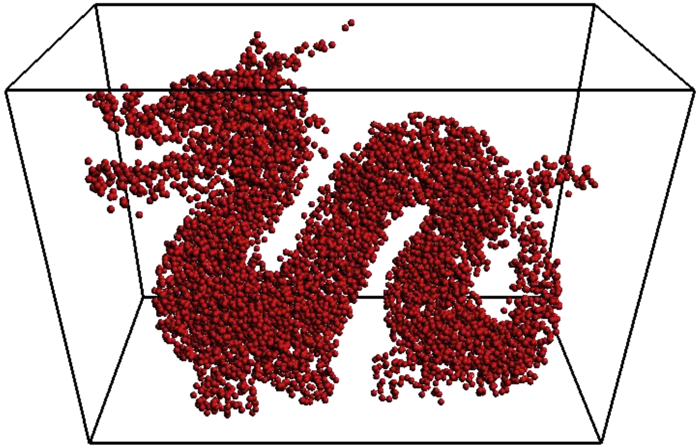
- intersect both LDI to get the overlapping volume
- provides an explicit intersection volume
- other boolean operations (union, difference) are also possible
  → constructive solid geometry (CSG)

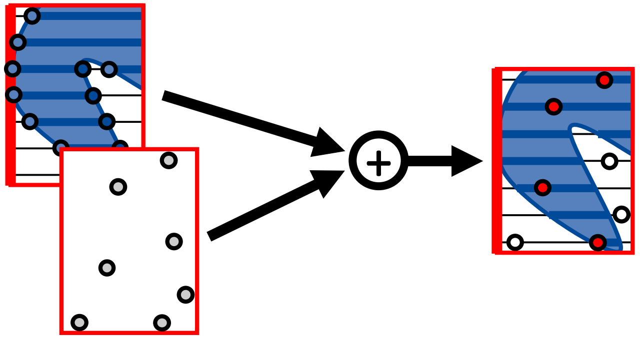



$LDI_{1\cap 2} = LDI_1 \cap LDI_2$

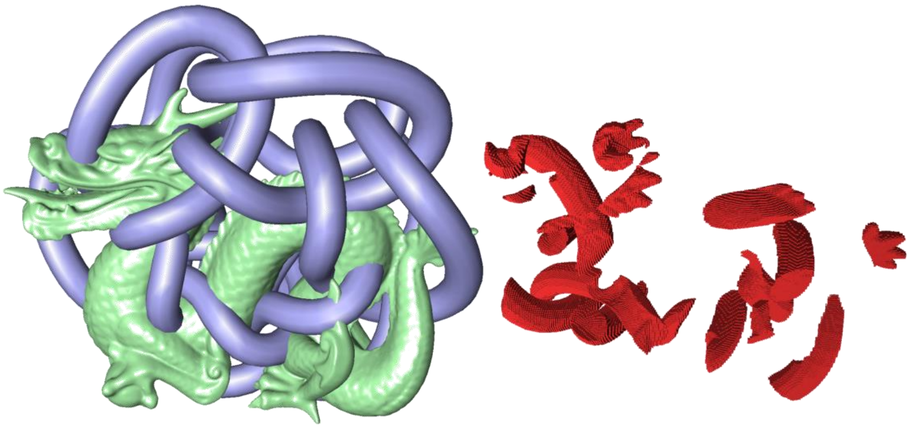
# *Collision queries*

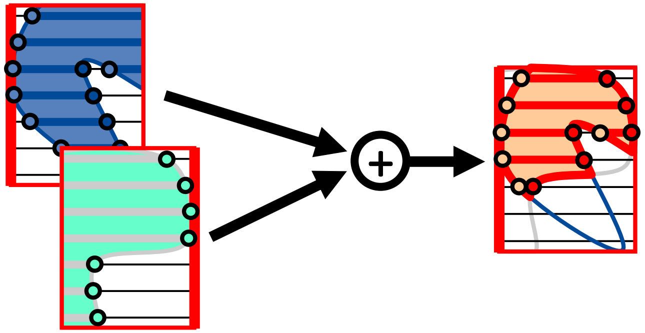
## Vertex-in-volume test





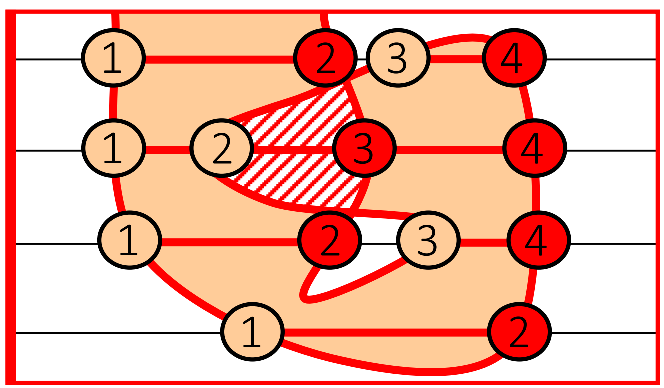
## Explicit intersection volume





# Self-collision query

- check for incorrect ordering of front and back faces
- if front and back faces do not alternate → self collision



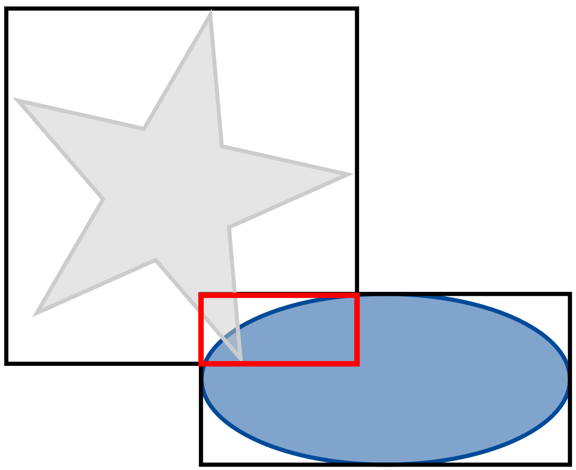

VoI

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 1 | 2 | 3 | 4 |
| 1 | 2 | 3 | 4 |
| 1 | 2 | | |

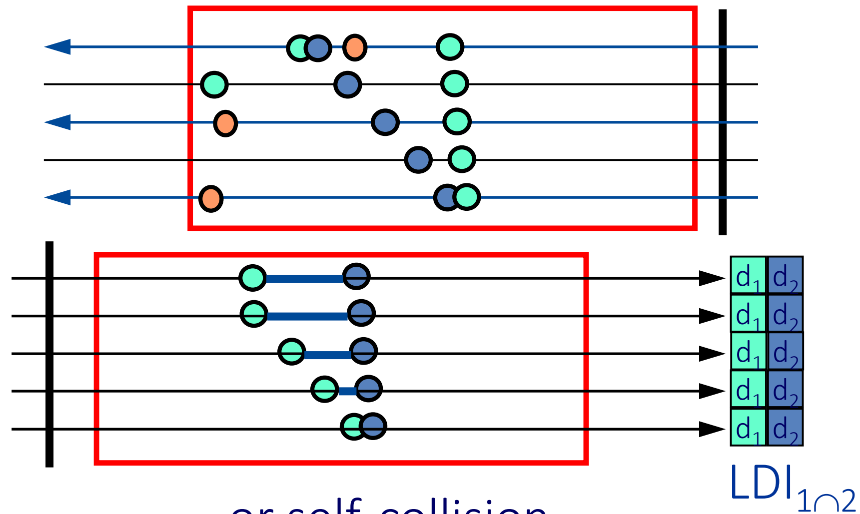
LDI

# Algorithm Summary

(1) Volume of interest

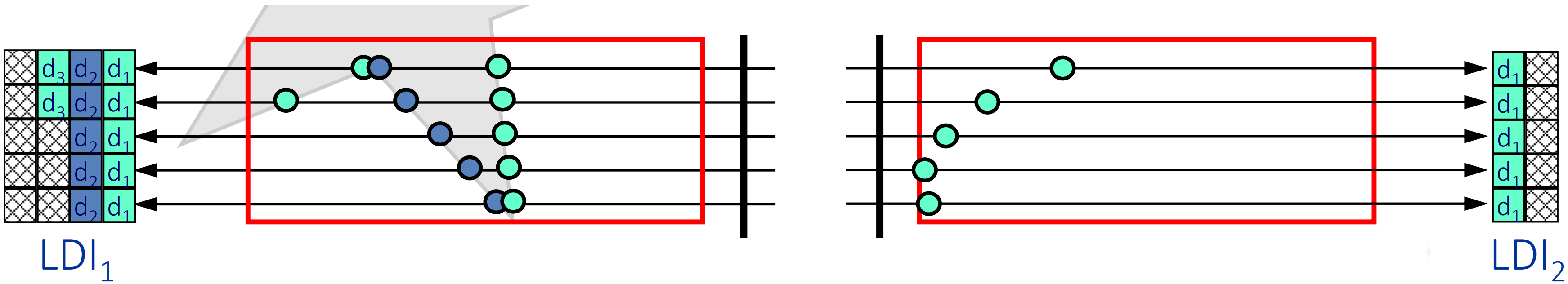


(3) Collision detection test



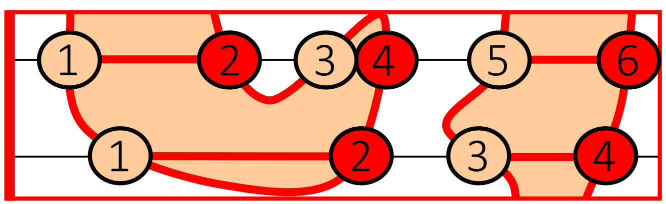

or self-collision

(2) LDI generation



# *Unordered LDI Generation*

- alternative method for LDI generation
- GPU generates unsorted LDI
  - fragments are rendered in the same order in each rendering pass
  - stencil buffer is used to get n-th value in the n-th pass
- CPU generates ordered LDI
  - depth complexity is known for each fragment
    (how many values are rendered per pixel)




VoI

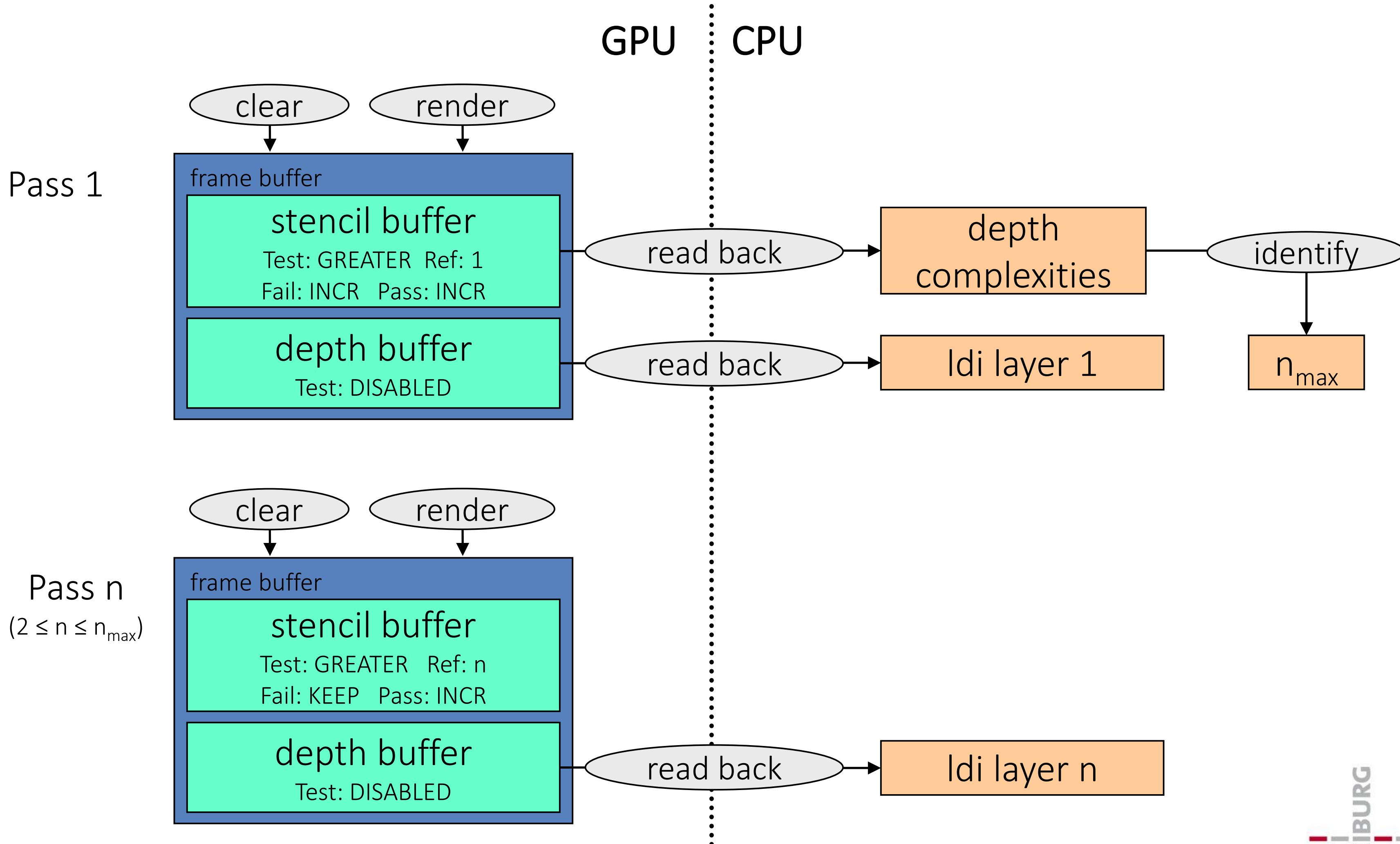
| | | | | | |
|---|---|---|---|---|---|
| 5 | 3 | 2 | 1 | 4 | 6 |
| 4 | 1 | 3 | 2 | 2 | 2 |

unsorted LDI (GPU)

| | | | | | |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 2 | 3 | 4 | 2 | 2 |

sorted LDI (CPU)

# Unordered LDI Generation



# *Limitations*

- performance is dependent on:
  - depth complexity of objects in volume of interest
  - read back delay for simple objects
  - rendering speed for complex objects

- requires graphics hardware

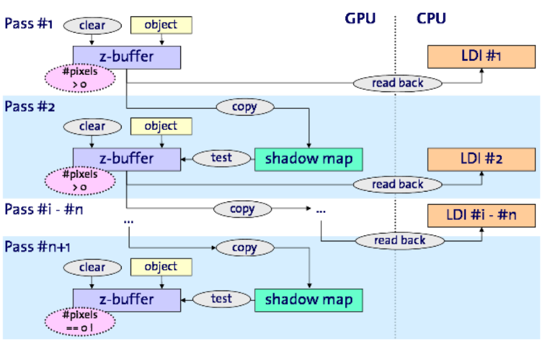
# Ordered LDI Generation on CPU

- Motivation
  - buffer read-back from GPU
    can be performance bottleneck
  - GPU requires multiple passes
  - CPU can store fragments directly into LDI

- Simplified software-renderer
  - rasterization of triangle meshes
  - frustum culling
  - face clipping
  - orthogonal projection

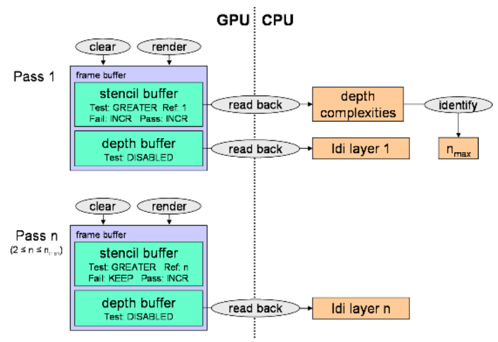
# LDI Generation - Summary

### *Ordered LDI (GPU)*




- n+1 passes
- complex setup
- two depth tests
- shadow map
- OpenGL extensions

### *Unordered LDI (GPU)*




- n passes
- simple setup
- no depth test
- stencil buffer
- plain OpenGL 1.4

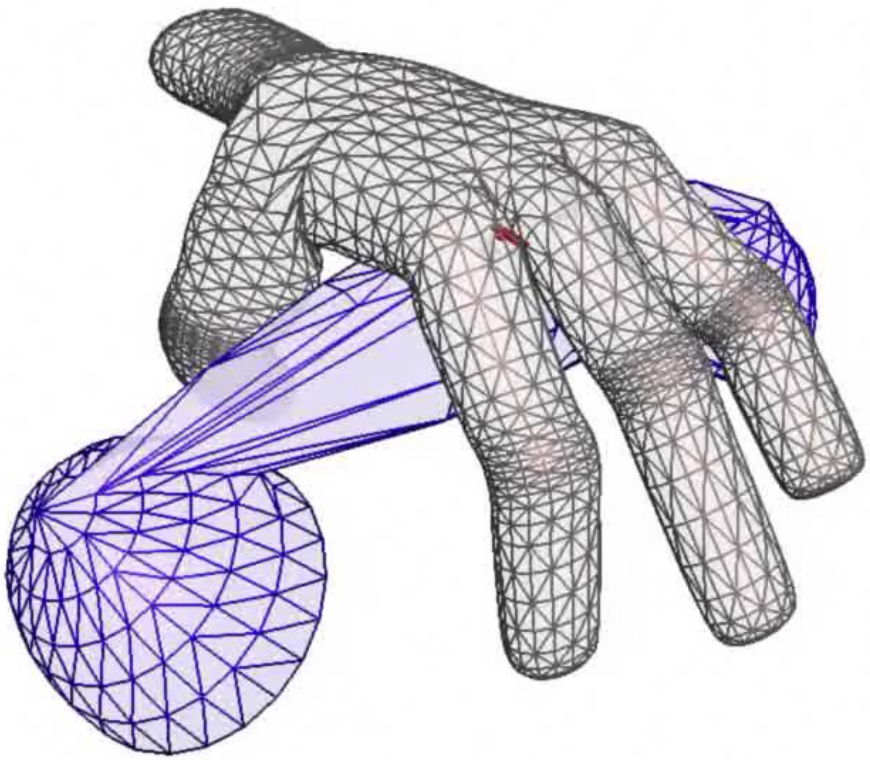
### *Ordered LDI (CPU)*

rasterize

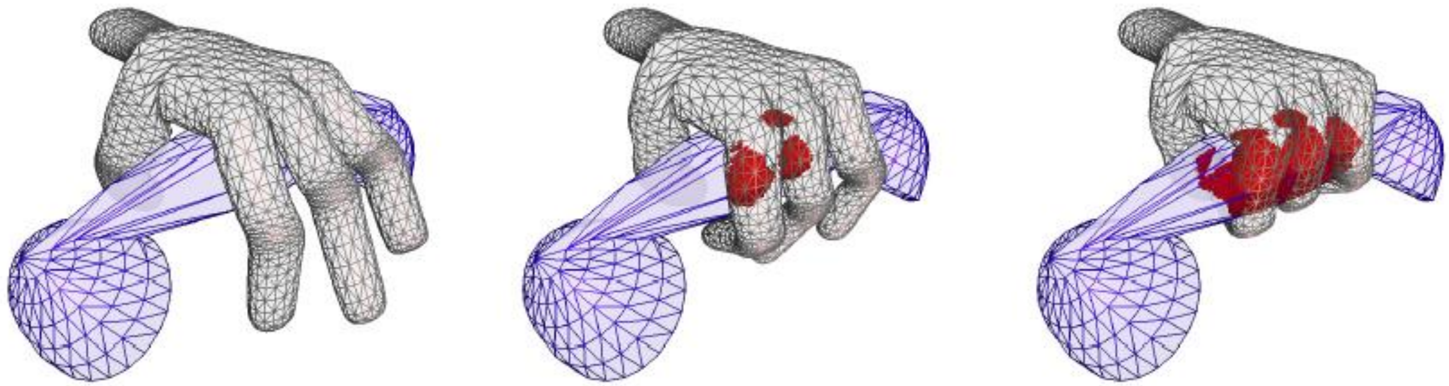
- 1 pass
- simple setup
- no depth test

# Performance - Intersection Volume

- hand with 4800 faces
- phone with 900 faces
- two LDIs
- intersection volume for collision detection
- analysis of front / back face ordering for self-collision



# Performance – Intersection Volume



| method | collision min / max | self collision min / max | overall min / max |
|---|---|---|---|
| ordered (GPU) | 28 / 37 | 40 / 54 | 68 / 91 |
| unordered (GPU, CPU) | 9 / 12 | 12 / 18 | 21 / 30 |
| software (CPU) | 3 / 4 | 5 / 7 | 8 / 11 |

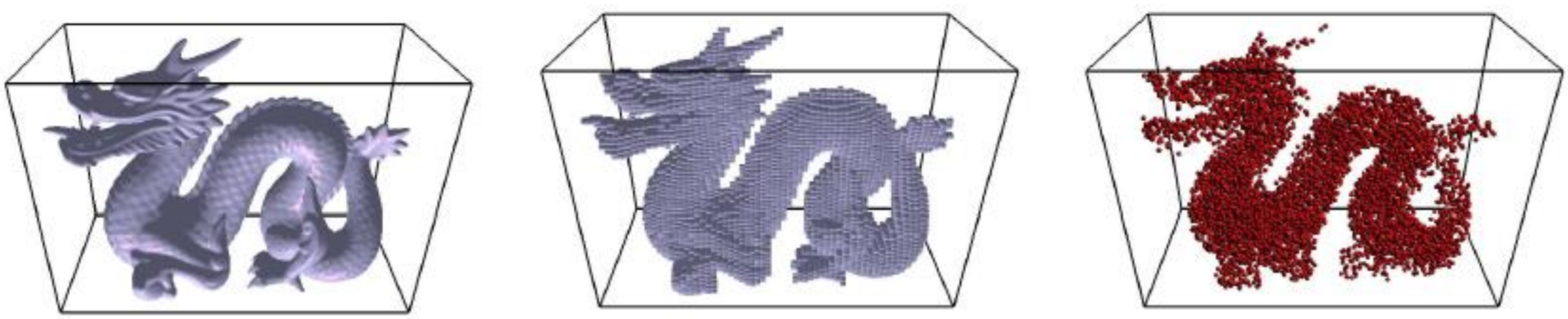
3 GHz Pentium 4, GeForce FX Ultra 5800

hand with 4800 faces
phone with 900 faces
measurements in ms

# Performance – Vertex-in-Volume

- santa with 10000 faces
- 20000 particles
- one LDI
- test vertices against inside regions of the LDI

# *Performance – Vertex-in-Volume*



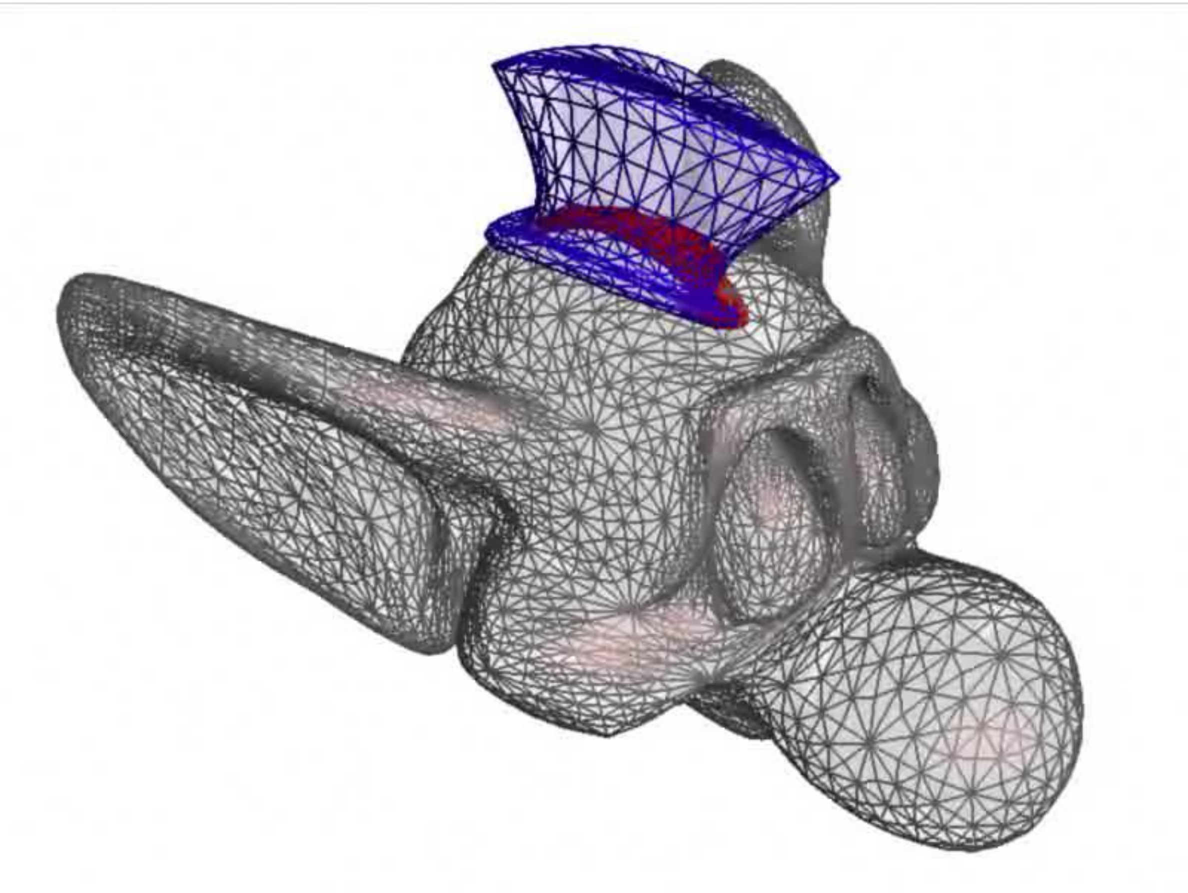
| method | 520k faces<br>100k particles | 150k faces<br>30k particles | 50k faces<br>10k particles |
|---|---|---|---|
| ordered (GPU) | 450 | 160 | 50 |
| unordered (GPU, CPU) | 225 | 75 | 25 |
| software (CPU) | 400 | 105 | 35 |

3 GHz Pentium 4, GeForce FX Ultra 5800

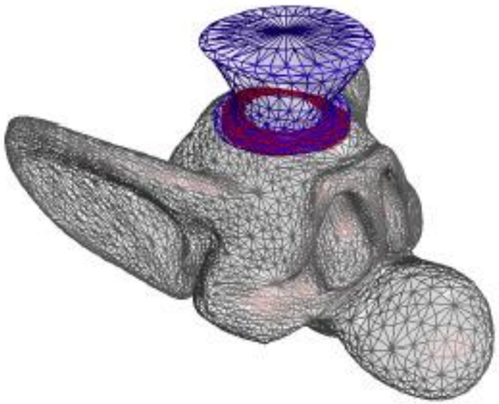
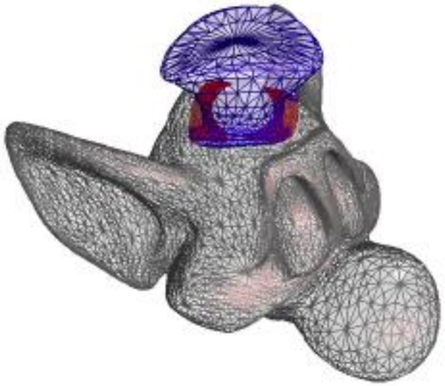
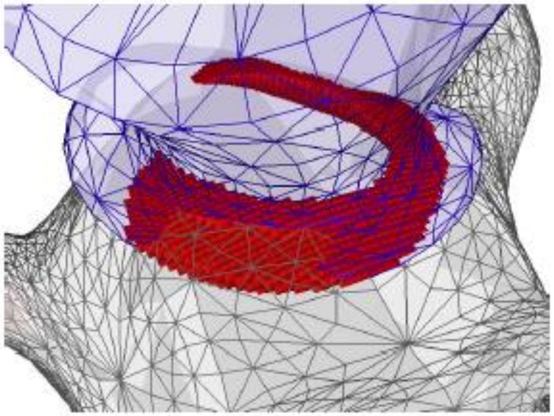
LDI resolution 64 x 64
measurements in ms

# Performance – LDI resolution

- mouse with 15000 faces
- hat with 1500 faces
- two LDIs
- intersection volume for collision detection



# Performance – LDI resolution

| method | 32 x32 | 64 x 64 | 128 x128 |
|---|---|---|---|
| ordered (GPU) | 24 | 26 | 51 |
| unordered (GPU, CPU) | 8 | 9 | 17 |
| software (CPU) | 2 | 3 | 6 |

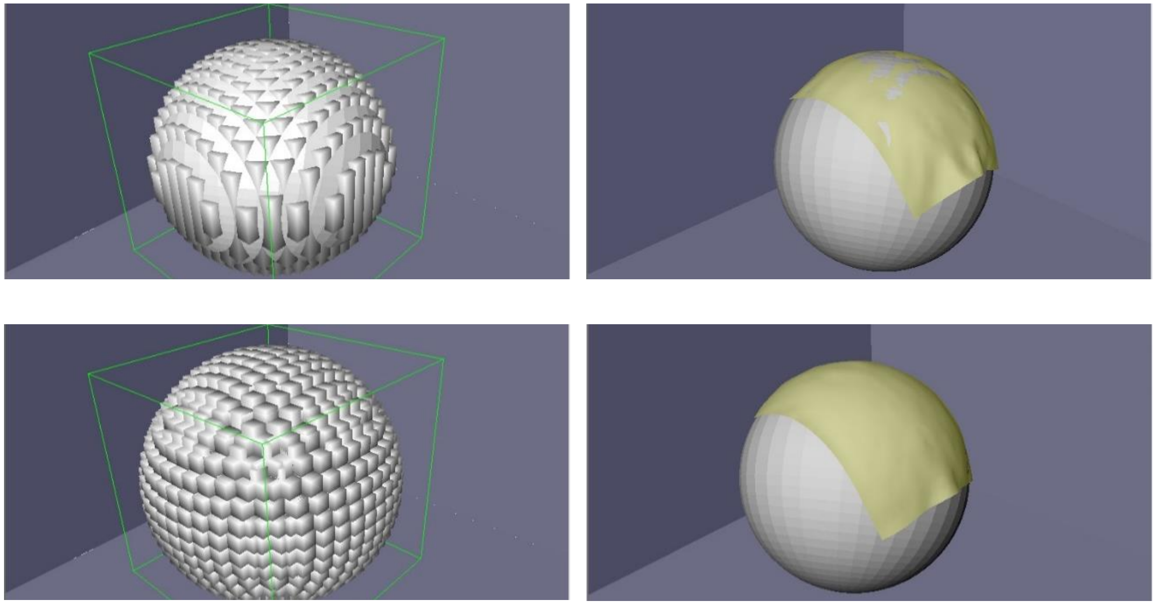
3 GHz Pentium 4, GeForce FX Ultra 5800

mouse with 15000 faces
hat with 1500 faces
measurements in ms

# Applications – Cloth Modeling

LDI

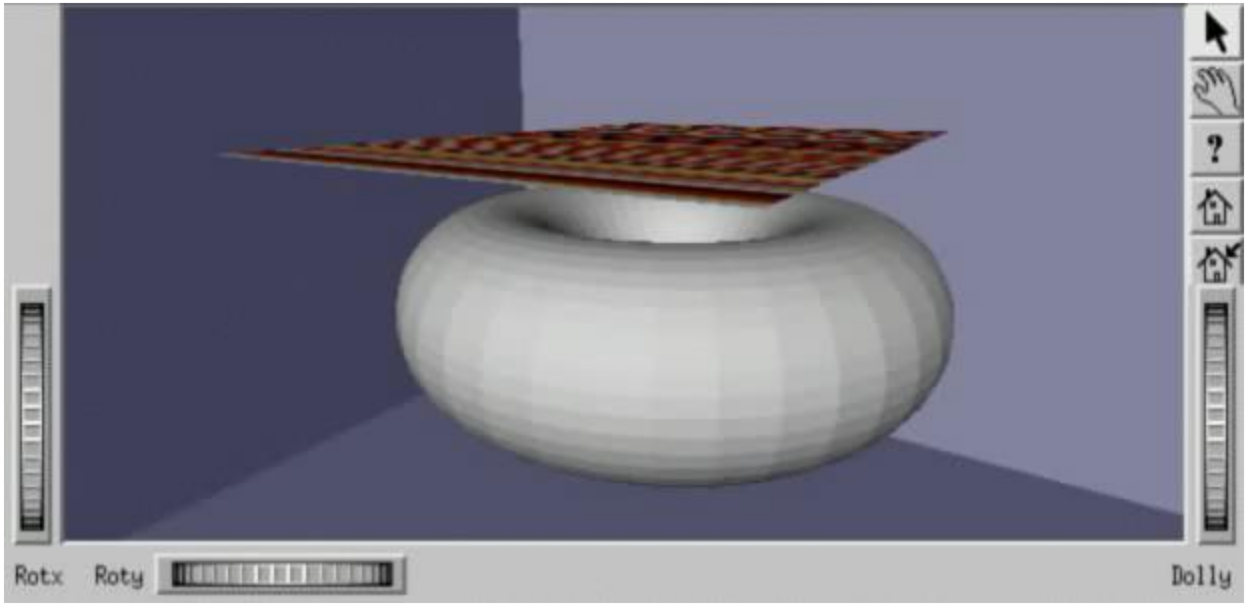
3 orthogonal dilated LDIs





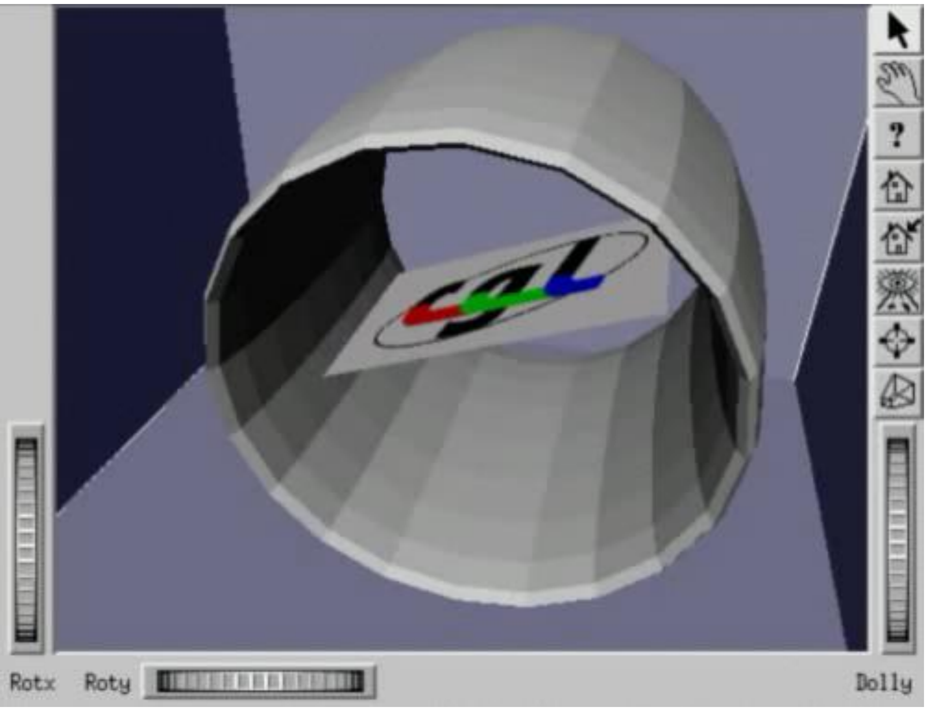
UNI FREIBURG

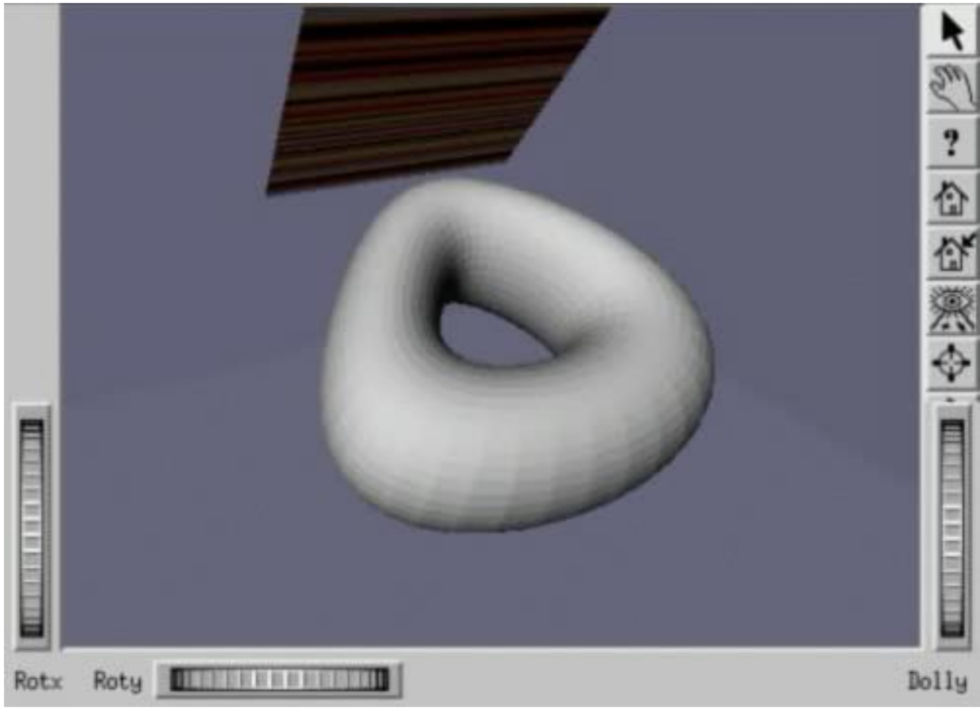
# Real-Time Cloth Simulation with Collision Handling




stable collision handling

# *Real-Time Cloth Simulation with Collision Handling*




concave transforming object

concave deforming object

# Summary

- image-space technique
- detection of collisions and self-collisions
- handling of rigid and deformable closed meshes
- no pre-processing
- CPU: 5000 / 1000 faces at 100 Hz
- GPU: 520000 faces / 100000 particles at 4 Hz
- application to cloth simulation
- limitations
  - closed meshes
  - accuracy
  - collision information for collision response

# References


- M. Shinya, M. Forgue, “Interference Detection through rasterization,” *Journal of Visualization and Computer Animation*, vol. 2, pp. 132-134, 1991.
- K. Myszkowski, O. Okunev, T. Kunii, “Fast collision detection between complex solids using rasterizing graphics hardware,” The Visual Computer, vol. 11, no. 9, pp. 497-512, 1995.
- J. C. Lombardo, M.-P. Cani, F. Neyret, “Real-time Collision Detection for Virtual Surgery,” *Proc. of Comp. Anim.*, pp. 82-91, 1999.
- G. Baciu, S. K. Wong "Image-Based Techniques in a Hybrid Collision Detector," *IEEE Trans on Visualization and Computer Graphics*, Jan 2002.
- D. Knott, D. Pai: “Cinder: Collision and interference detection in real-time using graphics hardware,” *Proc. Graphics Interface, 2003*.
- B. Heidelberger, M. Teschner, M. Gross, *“Volumetric Collision Detection for Deformable Objects,” Proc. VMV’03*, pp. 461-468, 2003.
- B. Heidelberger, M. Teschner, M. Gross, “Detection of Collisions and Self-collisions Using Image-space Techniques,” *Proc. WSCG’04*, pp. 145-152, 2004.