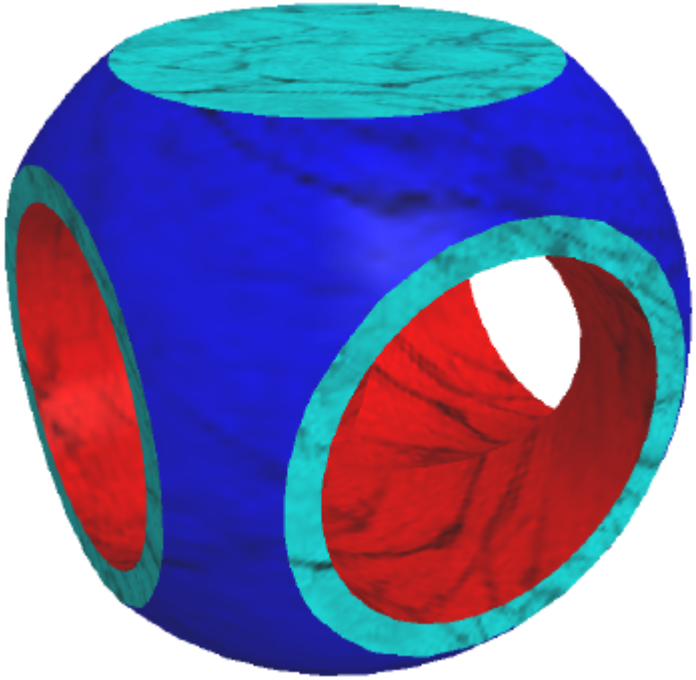
Rendering Techniques for Hardware-Accelerated Imaged-Based CSG



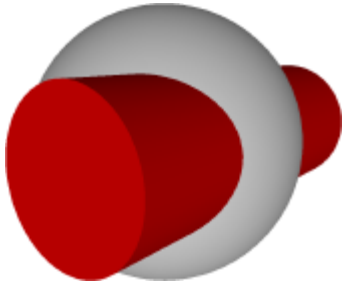
Florian Kirsch and
Jürgen Döllner

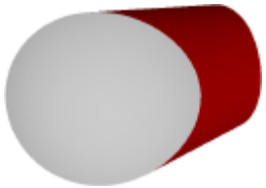
HASSO-PLATTNER-INSTITUT
for Software Systems Engineering
at the University of Potsdam

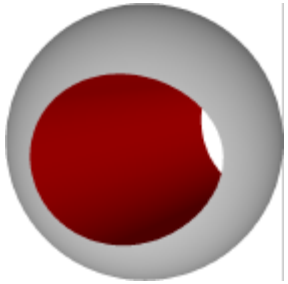
1. **Overview of Image-Based CSG**
2. **Optimizing Visibility Transfer**
3. **Optimizing Depth-Complexity Calculation**
4. **Results**
5. **Conclusions**

Constructive Solid Geometry (CSG)

- Composing shapes by volumetric boolean operations
- Union (È)
- Intersection (Ç)
- Subtraction (-)







- CSG tree

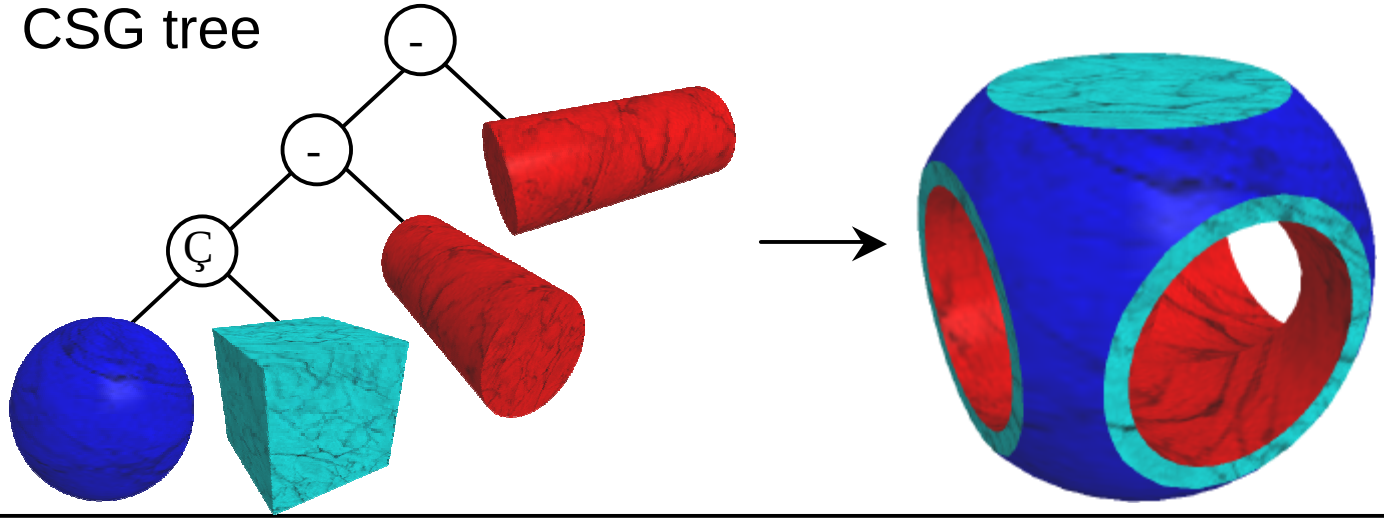
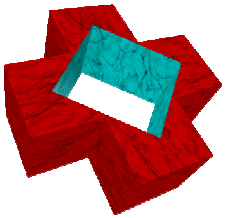
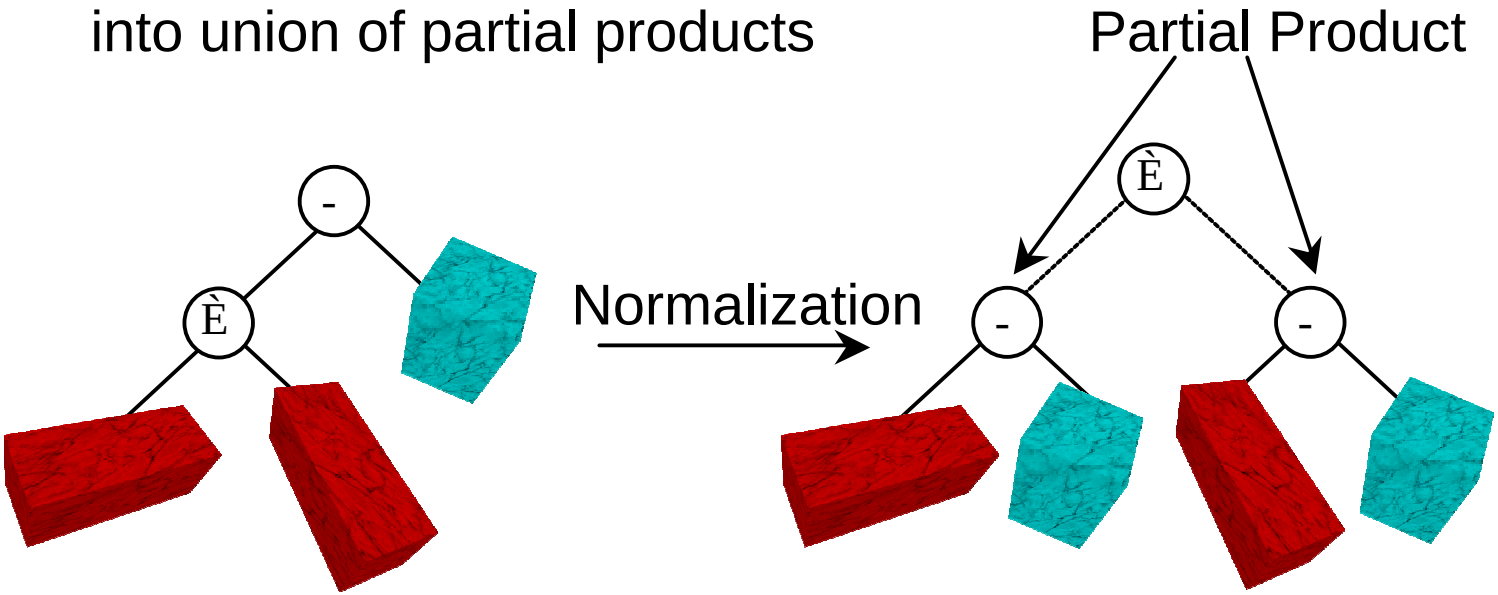


Image-Based CSG Rendering

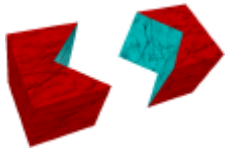


- Compose *CSG image* in frame-buffer
- Fast → good for interactive manipulation

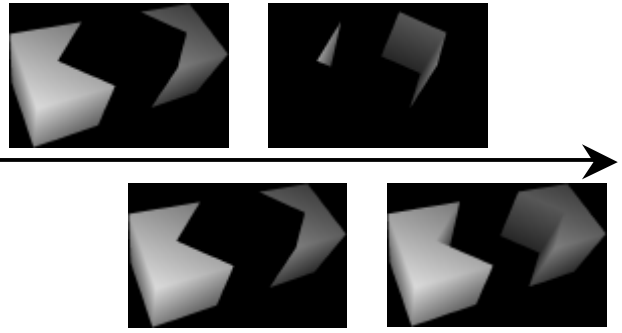
- Requires *CSG tree normalization* into union of partial products



Goldfeather Algorithm (GF)



- Convex and concave primitives
- Based on parity test
- Each (depth layer of a) primitive is handled separately
- Requires two z-buffers:
  1. temporary z-buffer → calculate visibility of primitive
  2. main z-buffer → accumulate results



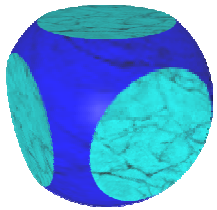
Layered Goldfeather Algorithm

- Handles depth-layers of partial product
- Good when depth-complexity is small

Sequenced Convex Subtraction (SCS)

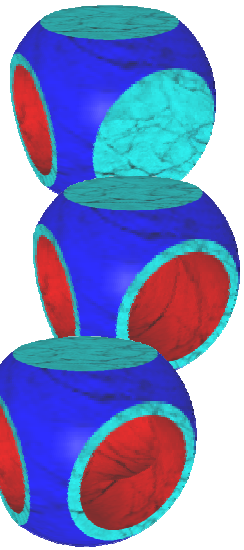
- Convex primitives only
- Complete partial product at once
- Requires temporary and main z-buffer (as GF)

1. Find backmost intersected front surface

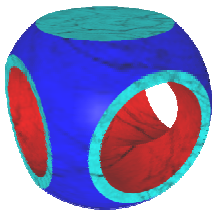


2. Subtraction Phase: “Dig out” subtracted primitives

   O(k*n) subtractions (k: depth-complexity)



3. Clip with intersected back surfaces



Our Contribution

Rendering techniques for image-based CSG

- that take advantage of graphics hardware,
- that fit into Goldfeather and SCS algorithms, and
- that drastically improve performance

1. Overview of Image-Based CSG
2. **Optimizing Visibility Transfer**
3. Optimizing Depth-Complexity Calculation
4. Results
5. Conclusions

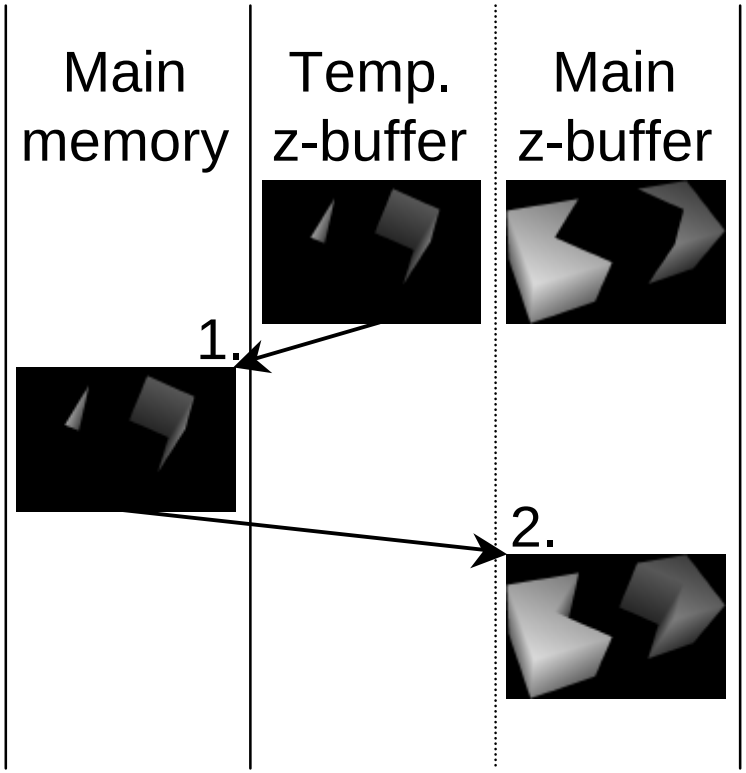
Problem

- Given z-values of (some) primitive in temporary z-buffer
- How to transfer them into main z-buffer?

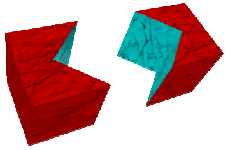
Previous Solution

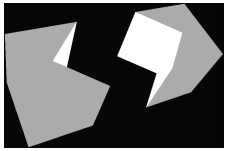
1. Copy z-buffer to main memory
2. Draw z-values from memory to main z-buffer, with “z-less” test

→ Problems: slow and depth artifacts



Our Solution

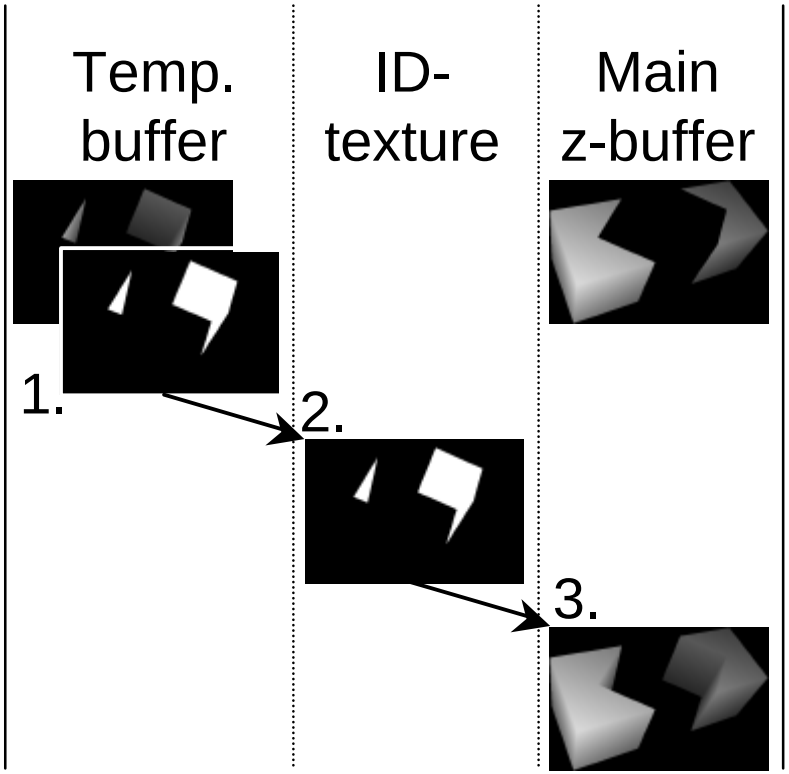


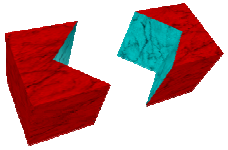


- **P-buffer** as temporary buffer
- Unique **alpha-ID** for each primitive

1. Encode visible parts of primitives with ID
2. Copy **IDs into alpha-texture** (render-texture)
3. Use IDs in texture to update main depth-buffer:

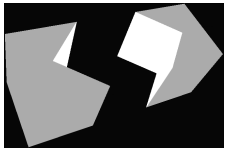
   Draw primitives again; **alpha-test** discards where
   $alpha_{primitive}$!= $alpha_{texture}$





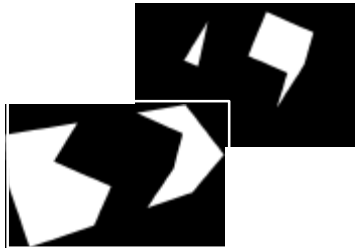
For SCS

- Each primitive in partial product is assigned an ID
- Max. 255 primitives in partial product
- One texture-copy per partial product



For GF

- Only one primitive at a time: one ID / texture
- One texture-copy per primitive



RGBA textures

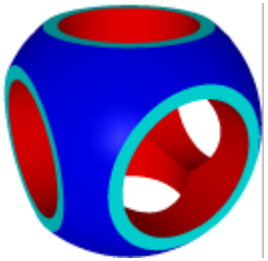
- Dotproduct3 can copy color component to alpha
- Permits “alpha-test” of a color value

Applications

- RGBA texture encodes visibility of
  - 4 primitives (GF)
  - 4 partial products (SCS)
- Up to 2^{32}-1 primitives in partial product (SCS)

1. Overview of Image-Based CSG
2. Optimizing Visibility Transfer
3. **Optimizing Depth-Complexity Calculation**
4. Results
5. Conclusions

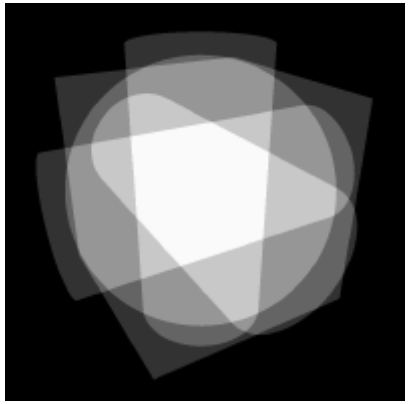
Depth-Complexity



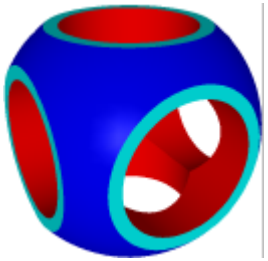
- required for layered Goldfeather
- useful for SCS to shorten subtraction sequence

Common Solution to Determine Depth-Complexity

- Reset stencil buffer
- Render primitives, incrementing stencil value (no z)
- Read stencil buffer to main memory
- Maximum stencil value represents depth-complexity



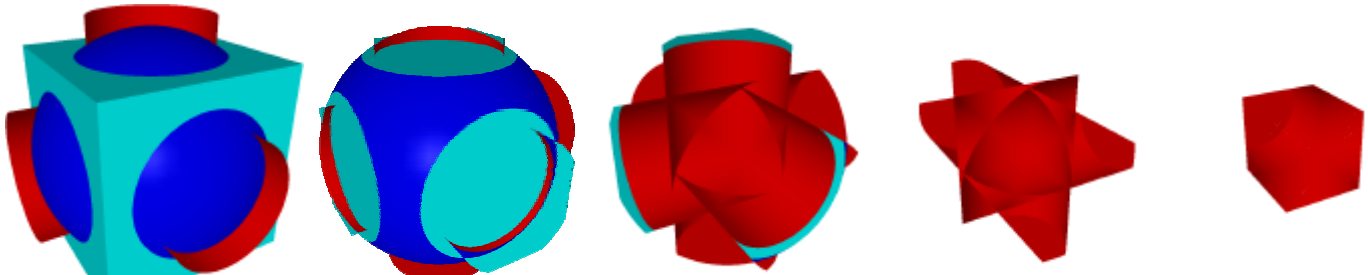
Occlusion Queries for Layered GF



```
k = 0
do
    Clear frame buffer
    begin fragment counting
        Render depth layer k
    end fragment counting, fc contains result
    k = k + 1
while fc != 0
                    → k contains depth-complexity
```

Layered Goldfeather renders each depth-layer anyway

→ no additional overhead!



k=0 k=1 k=2 k=3 k=4 Result: k=5

Occlusion Queries for SCS

Modified subtraction phase

- No depth-buffer updates for all n primitives?
- If yes, terminate subtraction phase.

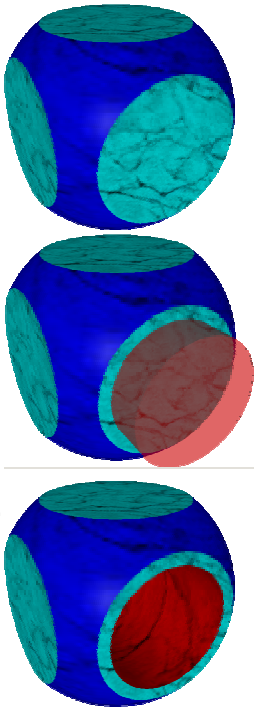
- This does not calculate depth-complexity k…
- … but profits from it anyway!

- Our sequence of subtracted primitives (e.g., *a, b, c, d*):

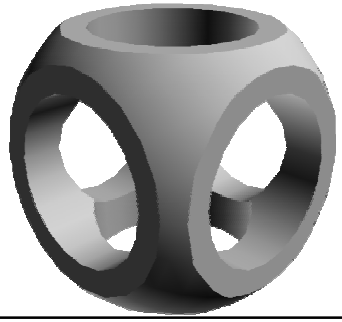
  a b c d a b c d a b c d a

Modified Subtraction Phase

```
for primitives P in subtraction sequence do
    Initialize stencil buffer with 0
    Set stencil buffer to 1 where z-values of P.front < z
    begin fragment counting
        Render P.front
    end fragment counting, fc contains result
    if fc != P.fragment_count then
        Update z where z-values of P.back > z and stencil == 1
        Render P.back
        P.fragment_count = fc
        primitives_without_update = 0
    else
        primitives_without_update++
    end if
    if primitives_without_update >= n then exit algorithm; end if
end for
```



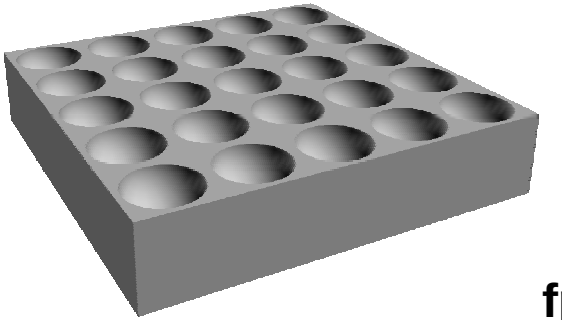
1. Overview of Image-Based CSG
2. Optimizing Visibility Transfer
3. Optimizing Depth-Complexity Calculation
4. **Results**
5. Conclusions



fps	FX5600, old method	FX5600, new method
Goldfeather	11	81
Layered Goldfeather	8	36
SCS	41	124
SCS with Depth-Complexity	25	117

Spends 64% of rendering time reading, 22% writing pixel-data

3-8 times faster!



fps	FX5600, old method	FX5600, new method
Goldfeather	2	16
Layered Goldfeather	9	38
SCS	18	24
SCS with Depth-Complexity	25	95

Up to 8 times faster!

Very fast! Because fewer primitives are subtracted here!

1. Overview of Image-Based CSG
2. Optimizing Visibility Transfer
3. Optimizing Depth-Complexity Calculation
4. Results
5. **Conclusions**

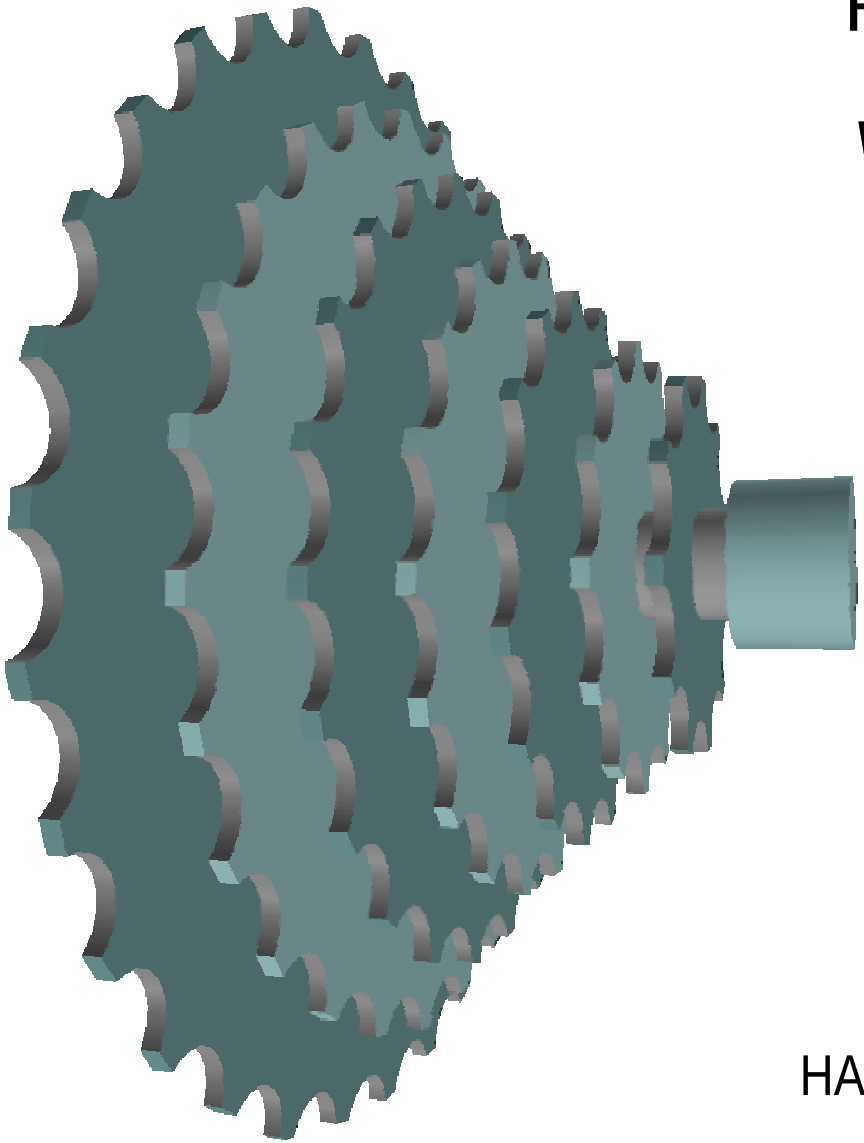
Image-based CSG rendering in **real-time**: Possible now for rather complex shapes

Render-to-texture is valuable tool to avoid slow rendering paths

Occlusion queries not only for occlusion culling, but to lower runtime of geometric algorithms

No depth-artifacts, in contrast to previous methods

SCS faster than Goldfeather, but cannot handle concave primitives



Free implementation:

www.opencsg.org

Thank you!

HASSO-PLATTNER-INSTITUT
for Software Systems Engineering
at the University of Potsdam