Rendering Techniques for Hardware-Accelerated Imaged-Based CSG

Florian Kirsch and Jürgen Döllner
1. Overview of Image-Based CSG
2. Optimizing Visibility Transfer
3. Optimizing Depth-Complexity Calculation
4. Results
5. Conclusions
1. Overview of Image-Based CSG

**Constructive Solid Geometry (CSG)**
- Composing shapes by volumetric boolean operations
  - Union ($\cup$)
  - Intersection ($\cap$)
  - Subtraction ($-$)
- CSG tree

![Diagram of CSG tree and boolean operations](image-url)
1. Overview of Image-Based CSG

**Image-Based CSG Rendering**

- Compose *CSG image* in frame-buffer
- Fast $\rightarrow$ good for interactive manipulation

- Requires *CSG tree normalization* into union of partial products
1. Overview of Image-Based CSG

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 $\rightarrow$ calculate visibility of primitive
  2. main z-buffer $\rightarrow$ accumulate results

Layered Goldfeather Algorithm
- Handles depth-layers of partial product
- Good when depth-complexity is small
1. Overview of Image-Based CSG

**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
Overview

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

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
   - \(\alpha_{\text{primitive}} \neq \alpha_{\text{texture}}\)
2. Optimizing Visibility Transfer

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
2. Optimizing Visibility Transfer

**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)
Overview

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

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
3. Optimizing Depth-Complexity Calculation

Occlusion Queries for Layered GF

\[
k = 0
\]
\[
do
\]
Clear frame buffer
\[
begin \text{fragment counting}
\]
Render depth layer \( k \)
\[
end \text{fragment counting, } fc \text{ contains result}
\]
\[
k = k + 1
\]
\[
while \ fc \neq 0
\]
\[
\rightarrow k \text{ contains depth-complexity}
\]

Layered Goldfeather renders each depth-layer anyway
\[
\rightarrow \text{no additional overhead!}
\]

Result:

\[
k=0 \quad k=1 \quad k=2 \quad k=3 \quad k=4 \quad k=5
\]
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., \(a, b, c, d\)):
  \[
  a \ b \ c \ d \ a \ b \ c \ d \ a \ b \ c \ d \ a
  \]
3. Optimizing Depth-Complexity Calculation

Modified Subtraction Phase

\begin{verbatim}
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
\end{verbatim}
1. Overview of Image-Based CSG
2. Optimizing Visibility Transfer
3. Optimizing Depth-Complexity Calculation
4. Results
5. Conclusions
### 4. Results

<table>
<thead>
<tr>
<th></th>
<th>FX5600, old method</th>
<th>FX5600, new method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goldfeather</td>
<td>11</td>
<td>81</td>
</tr>
<tr>
<td>Layered Goldfeather</td>
<td>8</td>
<td>36</td>
</tr>
<tr>
<td>SCS</td>
<td>41</td>
<td>124</td>
</tr>
<tr>
<td>SCS with Depth-Complexity</td>
<td>25</td>
<td>117</td>
</tr>
</tbody>
</table>

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

3-8 times faster!
4. Results

<table>
<thead>
<tr>
<th></th>
<th>FX5600, old method</th>
<th>FX5600, new method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goldfeather</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>Layered Goldfeather</td>
<td>9</td>
<td>38</td>
</tr>
<tr>
<td>SCS</td>
<td>18</td>
<td>24</td>
</tr>
<tr>
<td>SCS with Depth-Complexity</td>
<td>25</td>
<td>95</td>
</tr>
</tbody>
</table>

Up to 8 times faster!

Very fast! Because fewer primitives are subtracted here!
Overview

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

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

**Render-to-texture** is a 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!