Rendering Techniques for Hardware-Accelerated Image-Based CSG

Florian Kirsch Jürgen Döllner
Hasso-Plattner-Institute for Software Systems Engineering at the University of Potsdam
Prof.-Dr.-Helmert-Straße 2-3
D-14482 Potsdam, Germany
{kirsch,doellner}@hpi.uni-potsdam.de

ABSTRACT

Image-based CSG rendering algorithms for standard graphics hardware rely on multipass rendering that includes reading and writing large amounts of pixel data from and to the frame buffer. Since the performance of this data path has hardly improved over the last years, we describe new implementation techniques that efficiently use modern graphics hardware. 1) The render-to-texture ability is used to temporarily store shape visibility, avoiding the expensive copy of z-buffer content to external memory. Shape visibility is encoded discretely instead of using depth values. Hence, the technique is also not susceptible to artifacts in contrast to previously described methods. 2) We present an image-based technique for calculating the depth complexity of a CSG shape that avoids reading and analyzing pixel data from the frame buffer. Both techniques optimize various CSG rendering algorithms, namely the Goldfeather and the layered Goldfeather algorithm, and the Sequenced-Convex-Subtraction (SCS) algorithm. This way, these image-based CSG algorithms now operate accelerated by graphics hardware and, therefore, represent a significant improvement towards real-time image-based CSG rendering for complex models.

Keywords
Constructive Solid Geometry, CSG Rendering, Image-Based Rendering, Rendering Algorithms, Solid Modeling

1. INTRODUCTION

CSG modeling, i.e. 3D-shape modeling by means of volumetric Boolean operations, is a powerful tool in many areas of applications, e.g., manufacturing, engineering, and 3D-interactive sculpting. For interactive manipulation and display of CSG shapes, *image-based CSG* rendering algorithms are most suitable. To this category belong the Goldfeather algorithm [Gol86a, Gol89a, Wie96a], the layered Goldfeather algorithm [Ste98a], and the SCS (Sequenced Convex Subtraction) algorithm [Ste00a, Ste02a].

We identified as main bottleneck of these algorithms, as described in previous publications, reading pixel data from the frame buffer to external memory. This operation is needed due to:

- **Visibility transfer**: The algorithms determine the visibility of CSG primitives and store the corresponding depth values in a temporary depth buffer. To hold a copy of the main depth buffer and to merge the temporary depth values with the main depth buffer, depth buffers are saved to external memory and restored from it.
- **Depth-complexity calculation**: The algorithms, in general, calculate the depth complexity of a CSG shape by counting the overdraw in the stencil buffer, and then reading the stencil values to find the maximum overdraw.

Today, the performance of image-based CSG rendering is mainly bound by the throughput of this data path.

In our approach, we propose a solution for performing visibility transfer and depth-complexity calculation by the graphics hardware on its own. Thereby, the overall performance of CSG algorithms is drastically improved. In addition, our solution for visibility transfer is insusceptible to artifacts in contrast to previous methods.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Journal of WSCG, Vol.12, No.1-3., ISSN 1213-6972
WSCG'2004, February 2-6, 2004, Plzen, Czech Republic.
Copyright UNION Agency – Science Press

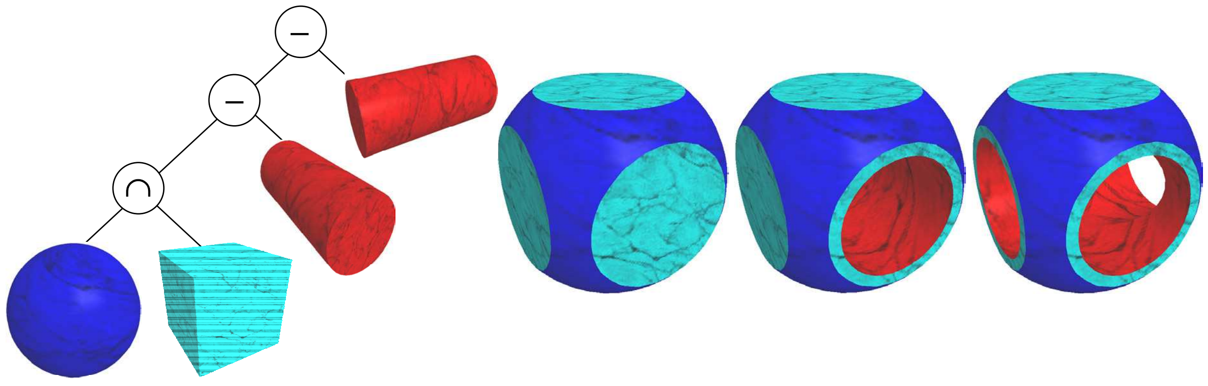

Figure 1: A sample CSG tree (a). It consists of four primitives; the sphere and the box are intersected; the cylinders are subtracted. The intermediate results (b, c) and the final image of the CSG expression (d).

Section 2 summarizes related work and introduces basic concepts of image-based CSG. Section 3 discusses our technique for visibility transfer. Section 4 explains the technique calculating depth complexity. Section 5 gives a detailed performance analysis for test models, and Section 6 draws conclusions.

2. RELATED WORK

Constructive Solid Geometry (CSG) represents a powerful and expressive approach to geometric 3D modeling [Req80a]. In CSG, complex shapes are built from simple shapes by volumetric Boolean operations, i.e., union, intersection, and subtraction. A complex shape is specified by a CSG expression, which is commonly stored as a CSG tree whose leaf nodes represent basic shapes (primitives such as sphere, cylinder, and box) and inner nodes denote Boolean operations (see Figure 1).

Image-based CSG algorithms are a category of algorithms for z-buffer graphics hardware that generate "just the image" of a CSG shape without calculating a description of the final object geometry. Compared to object-based algorithms, image-based CSG algorithms offer a number of advantages in many areas of applications. For example, they allow for interactively composing and manipulating a CSG shape. In general, they also produce less visual artifacts than a possibly approximated 3D geometry.

We are concentrating only on those image-based CSG algorithms that can be implemented on standard graphics hardware, because we aim at massive hardware-acceleration available on today's GPUs. Algorithms that rely on specialized graphics hardware, such as the Trickle algorithm [Eps89a], are not considered.

The Goldfeather Algorithm

Goldfeather et al. presented a CSG rendering algorithm for the Pixel-Planes graphics hardware [Gol86a, Gol89a]. Their work includes the notion of *tree normalization*, a set of equations to transform a generic CSG tree into an equivalent union of one or more partial products, whereby a *partial product* is built by intersection and subtraction of an arbitrary number of primitives. The normalization ensures that CSG expressions can be rendered effectively using z-buffer supported graphics hardware. Today, all image-based CSG rendering algorithms rely on it.

Goldfeather et al. also observed that only front faces of intersected and back faces of subtracted primitives in a partial product are potentially visible. The Goldfeather algorithm separately tests the visibility of each potentially visible depth layer L of a (possibly concave) primitive P. If P is convex, obviously only one depth layer of P must be considered.

Visibility testing of L works as follows: The z-values of L are rendered into a temporary z-buffer. Then, a *parity test* is performed for all other primitives Q in the partial product to discard fragments of L that are not visible. The parity test counts the number of front and back depth layers of Q with less or equal depth as L. For visible parts of L, that number must be odd if Q is intersected, and similarly it must be even if Q is subtracted. When all parity tests for L have been performed, the temporary z-buffer contains the correct z-values for visible fragments of L. The z-values are merged with the content of the main z-buffer using a "z-less" test. The Goldfeather algorithm has a quadratic runtime behavior with respect to the number of primitives in a partial product.

Stewart et al. [Ste98a] observed that the depth complexity k of the primitives in a partial product is typically much smaller than the number of primitives n, and they proposed the *layered Goldfeather algorithm* that takes advantage of this fact. The idea is to test the visibility of a depth layer of the partial product instead that of a single primitive. The theoretical runtime of this algorithm is $O(n \cdot k)$. The problem of rendering artifacts that commonly occurred was solved later [Erh00a] (also see Section 3).

Recently Guha et al. applied two-sided depth testing – enabled by hardware support of shadow mapping on modern graphics hardware – to a variant of the layered Goldfeather algorithm [Guh03a]. By depth peeling [Eve01a], their algorithm applies the parity test for depth layers of the partial product in front to back order, whereby a stencil mask rejects visibility updates where the visibility of a CSG layer already has been determined.

The SCS Algorithm

Stewart et al. [Ste00a] developed the SCS algorithm and they later described a refined version [Ste02a] to which we will refer to. The SCS Algorithm directly handles convex primitives only; concave primitives can be processed if they are subdivided into a set of convex primitives.

To determine the z-values of a partial product, the SCS algorithm uses three stages:

- First, the front surface of all intersected primitives in the partial product is determined. Two principles are applied to achieve this task: First, the visible front face of the intersection must be further away from the viewer than all other front faces of intersected shapes. Second, *n* back faces of intersected shapes must be behind the furthest front face; otherwise the furthest front face is not visible.
- In the next stage, a sequence of subtracted primitives is subtracted from the z-buffer. A subtraction removes a primitive *P* from a temporary front surface, i.e., where the front surface of the subtracted primitive *P* is closer and the back surface further away than the temporary surface, the z-values are replaced by the z-values of the back surface of *P*. In general, the sequence of subtracted primitives must ensure that all permutations of primitives reside in the sequence in sorted order, such that all possible dependencies of primitives are correctly handled. A sequence that has this property is called *permutation embedding*.
- At last, z-values of the subtracted primitives are clipped to the back faces of the intersected primitives. This is necessary because currently visible back faces of subtracted primitives can be situated behind the back of the intersection of all intersected primitives in the partial product. To mark these spots as invisible, it is necessary to render all back faces of the intersected primitives, resetting the z-value of fragments that are closer than the current z-value.

The first and the last stage have linear runtime; the subtraction stage has quadratic runtime with respect to the number of subtracted primitives *n* because of the size of the permutation-embedding sequence. When the depth complexity *k* of the subtracted primitives is known, a shorter subtraction sequence of $n{\cdot}k$ primitives can be used [Ste00a]. Furthermore, the object-space arrangement of subtracted primitives can be analyzed to shorten the subtraction sequence in certain cases [Ste03a].

3. VISIBILITY TRANSFER

Both the Goldfeather and the SCS algorithms use two z-buffers: a temporary z-buffer used to compute the depth image of (part of) a partial product, and a final z-buffer to accumulate the results. But standard z-buffer graphics hardware does not support two simultaneous z-buffers for a single fragment. Wiegand proposed a workaround for the Goldfeather algorithm [Wie96a], which was later also applied to SCS: His method saves the main z-buffer into main memory. After calculating the visibility of some CSG primitives, it restores the z-buffer and merges the temporary result. Unfortunately, this solution has important shortcomings:

- In general, the original z-values do not exactly match the copied z-values, since the OpenGL standard does not guaranty such exactness (due to the conversion of the data format). As a consequence, z-artifacts occur [Erh00a].
- Z-values are temporarily copied from graphics memory to main memory. Even under ideal circumstances, performance of this approach will be moderate due to bandwidth limitations. We expect that the throughput of this data path will hardly increase in future hardware (For measurements of the throughput on today's graphics hardware see Section 5).

Erhart and Tobler omit z-buffer copies. Instead, they copy IDs for shapes that are stored in the stencil-buffer [Erh00a]. This solves the z-artifacts problem; it does not solve the performance constraint since stencil-buffer copies are not significantly faster than z-buffer copies.

Render-To-Texture

In our technique, visibility transfer is based on p-buffers [Wyn01a], which represent off-screen frame buffers. Rendering processes linked to other frame buffers can use the p-buffer color channel as 2D-texture. This functionality, commonly denoted as *render-to-texture*, omits copying color data between main memory and graphics memory and, therefore, is well suited for real-time rendering.

For our purpose, the p-buffer has the same size as the frame buffer[1]. Using automatically generated texture coordinates, the texture that corresponds to the p-buffer can be projected on the frame buffer in such a way that every texel in the texture has a one-to-one mapping to a pixel in the frame buffer. This idea has been previously applied to depth-textures to implement depth-peeling [Eve01a].

ID Textures for Visibility of Primitives

For convex primitives, only the front face of intersected primitives and the back face of subtracted primitives are potentially visible. This means that the visibility information of a convex primitive can be encoded by marking the position with a unique bit code ultimately.

3.2.1 SCS Algorithm

For the SCS algorithm, our technique uses p-buffers to determine the pixel positions at which a primitive in a partial product is visible. Thereby, we use the alpha channel of the p-buffer to store temporary results, i.e., the alpha value of a pixel in the p-buffer indicates which primitive is visible at a position. For this purpose, each primitive in the partial product is assigned a unique ID. Every time the depth buffer is updated to denote visibility of a primitive, we also store the ID of the primitive in the alpha buffer. Where no primitive is visible at all, we store the special ID 0.

After computing the partial product, the alpha channel of the p-buffer holds all necessary visibility information for the primitives in the partial product. For the visibility transfer, we use the alpha channel of the p-buffer as texture, and we project this texture on the main frame buffer. Then, with "z-less" test, we render all potentially visible layers of primitives in the partial product (i.e., back or front faces), discarding all fragments with an alpha texture-value that differs from the ID of the current primitive. This is easily done with the alpha test. Finally, the z-buffer in the back buffer contains the z-values for all partial products computed so far.

3.2.2 Goldfeather Algorithm

The Goldfeather algorithm cannot deploy shape IDs in the same way as the SCS algorithm, because it calculates the visibility of a single primitive at a time. So it would mark a primitive as potentially visible, and then the parity test would determine which parts would be really visible, remarking all other parts as invisible. In this step, the visibility IDs for primitives that were calculated earlier would have to be restored. This is not possible without temporarily storing them elsewhere, which would impose an enormous overhead. Because of that, we cannot reasonably use IDs for primitives in the same way as in the SCS algorithm.

Instead, we store the visibility information for only a single primitive in the alpha channel of the p-buffer. The alpha buffer contains a value of 0 exactly if the primitive is not visible − the visibility transfer is possible in the same way as in the SCS algorithm.

We can apply the same approach for concave primitives. The Goldfeather algorithm calculates their visibility layer by layer, so if we store the number of the layer along with the visibility information, these information suffice for the visibility transfer into the main z-buffer. More exactly, for concave primitives, the alpha buffer contains a value of 0 exactly if the given layer of the primitive is not visible. In the same way, we deploy ID textures to the layered Goldfeather algorithm.

3.2.3 RGBA Textures

Besides the alpha channel, we can store visibility information also in the red, green, and blue color channels. The algorithm projects an RGBA-texture onto the frame buffer and needs a kind of 'alpha testing based on color values'. That testing is achieved with the ARB_combine_dot3 extension of OpenGL. Using the dot product of the texture-color with a constant color ((1,0,0), (0,1,0), or (0,0,1)), the texture environment is configured to move the information of a color channel into the alpha channel. The alpha test, then, is applied as usual. For the Goldfeather algorithm, this way one texture encodes the visibility of four primitives; for the SCS algorithm, the RGBA-texture even encodes the visibility of four partial products.

3.2.4 Limitations and Benefits

For the SCS algorithm, ID textures limit the number of primitives that can be contained in a partial product to the number of different IDs. An alpha buffer of 8 bit allows for 255 primitives. If more primitives are required, the IDs can be spread over the RGBA channels, allowing for $2^{32}-1$ different IDs.

Z-artifacts do not occur with our approach because ID textures contain binary information only.

4. DEPTH COMPLEXITY

The layered Goldfeather algorithm calculates the depth complexity k of a partial product to determine the maximum number of depth layers which must be handled by the algorithm. In analogy, calculating the depth complexity is advantageous for the SCS

[1] If non-power-of-two textures are not supported by the graphics hardware, we use a greater p-buffer and restrict application of the texture to that area that is covered by the frame buffer, by using the texture matrix.

algorithm because it allows to shorten the length of the subtraction sequence from n^2 to $n \cdot k$.

Calculating the depth complexity, as described in previous publications [Ste98a], is a costly operation: The overdraw of all pixels is determined by rendering the primitives of the partial product into the stencil buffer incrementing stencil values. Next, the stencil buffer is copied into main memory, and the maximum value is determined, which finally represents the maximum depth complexity of the partial product.

Generic Calculation of Depth Complexity

The occlusion-query capability of today's graphics hardware can easily determine the depth complexity of a partial product without reading back the stencil buffer. Occlusion queries count the number of fragments that have passed the stencil test and depth test while rasterizing a given set of primitives. In OpenGL, the corresponding NV_occlusion_query extension [Kil03a] has found widespread support from different hardware vendors, e.g., from NVidia and ATI.

To determine the depth complexity k, the partial product is rendered layer by layer, counting the number of rendered fragments for each layer. The number of the initial layer for which the number of rendered fragments is zero equals $k+1$.

At first glance, the approach appears to be inefficient because it requires rendering $n \cdot k$ primitives to determine the depth complexity, instead of only n primitives with the conventional approach of reading back the stencil buffer. However, the layered Goldfeather algorithm requires rendering each layer of the partial product in any case. Therefore, for this algorithm an occlusion query can be performed while rendering the next layer, and, if the number of fragments in this layer is zero, the algorithm terminates.

Depth Complexity for SCS

The SCS algorithm, during the subtraction stage, deploys occlusion queries to test whether the depth buffer has not changed during rasterization of the last n primitives. In this case, the subtract-stage is finished (exit condition), i.e., subtracting the remaining primitives in the subtraction sequence is not required anymore. This approach requires to use an alternative subtraction sequence.

4.2.1 An Alternative Subtraction Sequence

For subtracting primitives $P_1, \ldots, P_n$ Stewart et al. propose the following permutation-embedding sequence [Ste00a, Ste02a]

$$\underbrace{P_1P_2\ldots P_{n-1}P_nP_{n-1}\ldots P_2P_1P_2\ldots P_{n-1}P_nP_{n-1}\ldots P_2P_1P_2\ldots}_{n^2-n+1 \text{ shapes}}$$

However, we use the following sequence:

$$S_n = \underbrace{P_1P_2\ldots P_n \quad . \; . \; . \; . \quad P_1P_2\ldots P_n}_{n-1 \text{ times}} \; P_1$$

The length of S_n is also $n \cdot (n-1) + 1 = n^2 - n + 1$. First, we show by inductive proof that this sequence is really permutation embedding. For the induction base $n = 1$ the proposition is evident; and for $n = 2$ it is also because the sequence $S_2 = P_1P_2P_1$ obviously contains all permutations of P_1 and P_2, i.e., P_1P_2 and P_2P_1. So lets presume that the proposition is true for $P_1, \ldots, P_{n-1}$, i.e., the sequence S_{n-1}, build by $n-1$ times $P_1 \ldots P_{n-1}$ followed by P_1, contains all permutations of $P_1, \ldots, P_{n-1}$. The length of S_{n-1} is $(n-1)^2 - (n-1) + 1 = n^2 - 3n + 3$.

Consider S_n for the primitives $P_1, \ldots, P_n$ now. We choose an arbitrary primitive P_i. Then we construct a sub-sequence S' of S_n by removing those (at most $n-1$) primitives from S_n that are left of the first occurrence of P_i in S_n, and by removing all occurrences of P_i. P_i is removed exactly $n-1$ times if $i \neq 1$; if $i=1$, P_i is removed n times but then no other elements are removed from S_n. So, the size of S' is at least the size of S_n minus $2 \cdot (n-1)$, i.e., it is at least $n^2 - n + 1 - 2 \cdot (n-1) = n^2 - 3n + 3$. By construction, the beginning of S' has the same form as S_{n-1}, and S' is at least as long as S_{n-1}. Therefore, the first $n^2 - 3n + 3$ elements of S' embed all permutations of $P_1, \ldots, P_n$ without P_i (inductive hypothesis). But S' is a sub-sequence of S_n, hence permutations of $P_1, \ldots, P_n$ that start with P_i are embedded in S_n. Our choice of P_i was arbitrary, so all permutations of $P_1, \ldots, P_n$ are embedded in S_n, which means that S_n is permutation embedding.

Therefore, our permutation-embedding sequence is as effective and efficient as the sequence given by Stewart et al. Its advantage for our algorithm, however, is that when n primitives out of the subtraction sequence have been rendered, n different primitives have been rendered. We apply this property, which would be wrong for the sequence of Stewart et al., for the exit condition of our modified subtraction algorithm below.

4.2.2 Modified Subtraction Algorithm

In the subtraction stage, we assign a value $P_{.fragment\_count}$ to each subtracted primitive P that holds the number of rendered fragments. The value is initialized with zero. Then, for all primitives in the sequence, the following enhanced subtraction is performed (additions to the original SCS subtraction are set in bold):

Algorithm 1 modified subtraction of $P_1, \ldots, P_n$

```
Generate sequence $S_n$ from $P_1, \ldots, P_n$
*primitives_without_update* = 0
for primitives $P$ in $S_n$ do
    Initialize stencil buffer with 0
    Set stencil buffer to 1 where z-values of $P_{.front}$ < z
    begin fragment counting
        Render $P$
    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 $P$
        $P_{.fragment\_count}$ = $fc$
        primitives_without_update = 0
    else
        primitives_without_update++
    end if
    if primitives_without_update >= $n$ then
        exit algorithm (exit condition)
    end if
end for
```

The front surface of *P* is rendered, with "z-less" test and without update of the z-buffer, but setting a stencil-bit where the z-test passes. **Additionally, the number of fragments that pass the z-test is counted. Only if this number does not equal the number stored with *P*,** the back surface of *P* is rendered where the stencil-bit is set and with "z-greater" test, updating z-values where stencil and z-test pass**.** Otherwise, the visibility of the front of *P* did not change compared to the last subtraction of *P*, so that the currently stored z-values already match the visible back surface of *P* and updating the z-buffer is not necessary.

The exit condition can be defined as follows: If for *n* consecutive subtractions the z-buffer was not updated, the subtraction stage is terminated immediately. In this case, the modified subtraction sequence has ensured that all *n* subtracted primitives in the partial product have been tested without updating the z-buffer, so that all dependencies between different subtracted shapes have been handled.

The pseudo code of our algorithm, which is outlined in Algorithm 1, can be optimized in an obvious way by iterating over the stencil reference values from 1 to 255 and only clearing the stencil buffer for every 255th primitive.

5 RESULTS

The performance of reading and writing pixel data from the frame buffer to external memory has been hardly accelerated over the last years. The measurements (Table 1) show that the older TNT2 can read pixels as fast as the newer ATI 9700. The ATI 9700 is even considerably slower than the TNT2 writing pixel data. The FX5600 performs better in these disciplines, however in comparison to the TNT2, the improvements do not come near to the comparable increase in fillrate or triangle throughput.

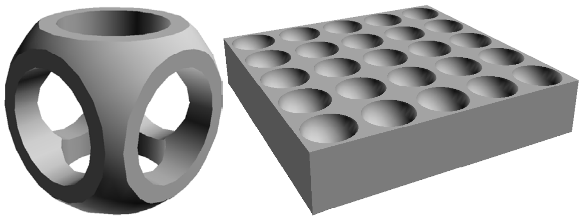
Figure 2: The widget and the grid model used for performance analysis.

Pixel transfer rate (Mpixels/sec)	NVidia TNT2 (1998)	Nvidia FX5600 (2003)	ATI 9700 (2003)
Read	26	41	26
Write	42	118	17

Table 1: Pixel-transfer rates of different generations of graphics hardware.

Our techniques have been implemented in C++ based on OpenGL. The implementation does not contain the "frame start optimization" [Wie96a]: For the first primitive (respectively partial product) in a frame, this optimization omits saving and restoring the z-buffer, because at this time the main z-buffer does not yet contain any data. Even though this is a useful optimization, it complicates analyzing the rendering performance: Algorithms that perform well for one partial product may perform worse for two or more partial products.

We measured the rendering performance at a resolution of 800x600, both on the FX5600 and the ATI 9700; results are given in frames-per-second. The performance results show that the presented implementation techniques lead to a drastically better performance for all tested CSG algorithms. In particular, the Goldfeather algorithm gets about eight times faster on the FX5600, and the layered Goldfeather as the SCS algorithms with depth-complexity sampling get about four times faster. The performance improvements for the ATI 9700 are not shown here in detail, but they are even more evident, because due to the slower pixel transfer performance, the ATI 9700 is significantly slower than the FX5600 in the conventional code path.

In the conventional code path, CSG rendering performance is bound by the pixel-transfer performance, even on graphics hardware such as the FX5600, where pixel transfer is comparably fast: For example, the widget model is composed of five

3d-model: widget	FX5600 conven-tional	FX5600 new	ATI 9700 new
Goldf.	11	81	114
Lay. Goldf.	8.2	36	40
SCS	41	124	228
SCS (DS)	25	117	220

Table 2: Performance results for the widget model.

3d-model: grid	FX5600 conven-tional	FX5600 new	ATI9700 new
Goldf.	2.0	16	24
Lay. Goldf.	8.9	38	50
SCS	18	24	73
SCS (DS)	25	95	198

Table 3: Performance results for the grid model.

primitives so that the conventional Goldfeather algorithm requires to save and restore the depth buffer five times to render the widget. At a resolution of 800x600 and 11 fps, these are about 26.4 Mpixels that are copied in a second. In relation to the maximum pixel-transfer rate of 41 Mpixels/sec for reading and 118 Mpixels/sec for writing, the Goldfeather algorithm spends about 64% of its rendering time for saving the depth buffer and 22% for restoring it. Only the remaining 14% are actually used for rendering in this example.

The SCS algorithm with depth-complexity sampling performs exceptionally well compared to the standard SCS algorithm; for the grid model the performance improvement are apparent. On the other hand, the widget model generally favours algorithms that do not determine the depth complexity, because all primitives overlap. But the SCS algorithm with depth-complexity sampling is hardly slower than the standard SCS algorithm. The overhead of the occlusion queries appear to be very small. Additionally, we found a similar effect as Guha et al. in their CSG algorithm [Guh03a]: In practice, the SCS algorithm with modified subtraction algorithm often requires rendering far fewer primitives than expressed by the run-time complexity of O($n \cdot k$) in the worst case. This is due to the exit condition that dynamically detects when the frame buffer has not been changed for a time long enough.

6 CONCLUSIONS

By taking advantage of features of modern graphics hardware, we can transfer core tasks in image-based CSG rendering to graphics hardware. On the one hand, occlusion queries determine depth complexity, on the other hand, p-buffers and textures transfer visibility information.

In terms of performance, both the Goldfeather and the SCS algorithm benefit from these techniques by a huge amount. Overall, the SCS algorithm remains faster. However, the Goldfeather algorithm is suitable for handling concave primitives which the SCS algorithm does not support directly.

The required graphics capabilities are included on current and future graphics hardware. The ARB_occlusion_query extension has been included in the OpenGL 1.5 specification; rendering to a texture will be exposed and optimized by ARB superbuffers [Mac03a]. Therefore, the presented techniques are well suited for real-time enabled image based CSG rendering on future graphics hardware.

7 REFERENCES

[Eps89a] Epstein, D., Jansen, F., and Rossignac, J. Z-Buffer Rendering from CSG: The Trickle Algorithm. IBM Research Report RC 15182, 1989.

[Erh00a] Erhart, G., and Tobler, R.F. General Purpose Z-Buffer CSG Rendering with Consumer Level Hardware. VRVis Technical Report 003, 2000.

[Eve01a] Everitt, C. Interactive order-independent transparency. Technical report, NVidia Corporation, 2001.

[Gol86a] Goldfeather, J., Hultquist, J. P. M., and Fuchs, H. Fast Constructive Solid Geometry

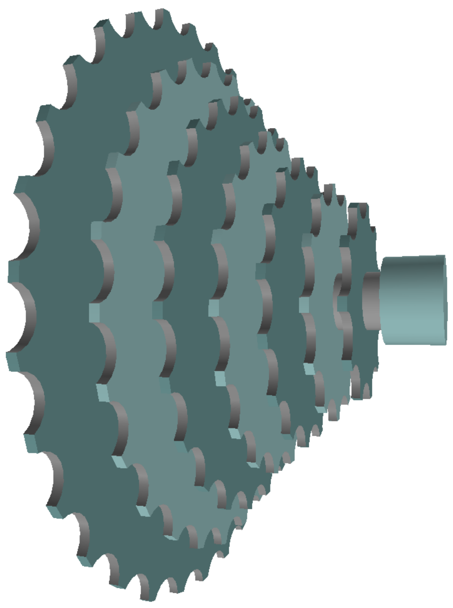

Figure 3: Cassette of a bicycle. This CSG model is composed of 150 primitives. With the SCS algorithm and depth-complexity sampling, the model is rendered interactively with 12fps on the GeForce FX5600.

Display in the Pixel-Powers Graphics System. ACM Computer Graphics (SIGGRAPH '86 Proceedings), 20(4):107-116, 1986.

[Gol89a] Goldfeather, J., Molnar, S., Turk, G., and Fuchs, H. Near Realtime CSG Rendering Using Tree Normalization and Geometric Pruning. IEEE Computer Graphics and Applications, 9(3):20-28, 1989.

[Guh03a] Guha, S., Krishnan, S., Munagala, K., and Venkatasubramanian, S. Application of the Two-Sided Depth Test to CSG Rendering. ACM SIGGRAPH 2003 Symposium on Interactive 3D Graphics, 177-180, 2003.

[Mac03a] Mace, R. OpenGL ARB Superbuffers. 2003. `http://developer.nvidia.com/docs/IO/8230/GDC2003_OGL_ARBSuperbuffers.pdf`.

[Kil03a] Kilgard, M. J. (editor). NVIDIA OpenGL Extension Specifications, June 2003. `http://developer.nvidia.com`.

[Req80a] Requicha, A. A. G. Representations for Rigid Solids: Theory, Methods, and Systems. ACM Computing Surveys, 12(4):437-464, 1980.

[Ste98a] Stewart, N., Leach, G., and John, S. An Improved Z-Buffer CSG Rendering Algorithm. 1998 Eurographics / SIGGRAPH Workshop on Graphics Hardware, ACM, 25-30, 1998.

[Ste00a] Stewart, N., Leach, G., and John, S. A CSG Rendering Algorithm for Convex Objects. Journal of WSCG, 8(2):369-372, 2000.

[Ste02a] Stewart, N., Leach, G., and John, S. Linear-time CSG Rendering of Intersected Convex Objects. Journal of WSCG, 10(2):437-444, 2002.

[Ste03a] Stewart, N., Leach, G., and John, S. Improved CSG Rendering using Overlap Graph Subtraction Sequences. Proceedings of GRAPHITE 2003, 47-53, 2003.

[Wie96a] Wiegand, T.F. Interactive Rendering of CSG Models. Computer Graphics Forum, 15(4):249-261, 1996.

[Wyn01a] Wynn, C. Using P-Buffers for Off-Screen Rendering in OpenGL. Technical report, NVidia Corporation, 2001.