

Visibility Driven Rasterization

M. Meißner, D. Bartz, R. Günther and W. Straßer

WSI/GRIS, University of Tübingen, Auf der Morgenstelle 10/C9, D-72076 Tübingen, Germany
meissner@gris.uni-tuebingen.de, bartz@gris.uni-tuebingen.de, guenth@gris.uni-tuebingen.de,
strasser@gris.uni-tuebingen.de

Abstract

We present a new visibility driven rasterization scheme that significantly increases the rendering performance of modern graphic subsystems. Instead of rasterizing, texturing, lighting, and depth-testing each individual pixel, we introduce a two-level visibility mask within the rasterization stage which facilitates the removal of groups of pixels and triangles from rasterization and subsequent pipeline stages.

Local visibility information is stored within the visibility mask that is updated several times during the generation of a frame. The update can easily be accomplished by extending already available (in hardware) occlusion culling mechanisms (i.e. those of HP and SGI), where it is possible to integrate the additional functionality without any additional delay cycles. In addition to these existing hardware based occlusion culling approaches—which cull only geometry contained in bounding volumes determined as occluded—we are able to significantly accelerate the rendering of the geometry determined as visible. However, our approach does not specifically rely on such occlusion culling hardware.

The proposed new rasterization scheme is well suited for hardware implementation, can easily be integrated into low-cost rasterizers, and its scalability can vary upon available chip real estate. Only incremental modifications of modern graphics subsystems are required to achieve a significant improvement in rendering performance.

Keywords: graphics hardware, rasterization, visibility and occlusion culling

1. Introduction

Over the last few years, the complexity and overall rendering bandwidth of graphics subsystems has increased dramatically. Starting from large scale multi-chip systems [1,2], single chip solutions [3,4], and even full graphics processing units (GPUs, which also integrate transformation, lighting, and setup [5]) are available. This trend is based on increased gate counts in integrated circuits and on new memory technologies. However, memory bandwidth and memory access efficiency remains one of the most troublesome issues.

Numerous approaches have been presented to address hiding of memory latency by caching [6], preventing stall cycles by prefetching [7], and interleaving memory such that stalls appear as infrequent as possible [4]. Nevertheless, each pixel can cause *memory stalls* due to memory accesses. Moreover, the dramatic board-to-chip integration, enables more complex dynamic geometry and

richer pixels operations [8]. Richer pixels comprise better filtering techniques, multi-texturing, and per pixel shading models. The associated iteration over multiple light sources and application of multi-texturing increases computational complexity as well as the overall memory inefficiency per pixel, since the multiple memory accesses potentially cause memory stalls. To reduce the overall processing costs of the increasingly expensive per-pixel rendering, we focus on the reduction of redundantly (not visible) processed pixels.

Approaches to reduce the bandwidth and rendering problems include mesh simplification and compression, and visibility and occlusion culling. While the former reduce the overall geometry load, the pixel complexity remains almost unchanged. In contrast, visibility and occlusion culling address geometry and pixel complexity by culling non-visible geometry. However, these algorithms either introduce high computational costs or tend to be of

limited efficiency in scenes with low depth complexity. Therefore, the remaining rasterization and subsequent pixel processing load is frequently beyond the interactive rendering capabilities of current graphics hardware. Our novel visibility driven rasterization significantly reduces the remaining rasterization load (after “traditional” occlusion culling) by introducing a two-level visibility mask. This mask enables the graphics hardware to reject triangles and groups of pixels in non-visible areas and requires only incremental modifications of the graphics subsystem.

Our paper is organized as follows. In the next section, we briefly describe related work on hardware support for visibility and occlusion culling. Section 2 introduces the algorithmic aspects and Section 3 discusses the hardware implications of visibility driven rasterization. In Section 4, we present the results of the evaluation of our novel approach, and finally, we conclude in Section 5.

1.1. Related work

There has been a lot of work in the field of visibility and occlusion culling. An overview can be found in [9,10]. We specifically focus on related work that deals with the hardware aspects.

Greene *et al.* proposed the hierarchical z-buffer algorithm [11] and the related hierarchical polygon tiling algorithm [12]. In both approaches, a polygonal scene is represented in an object- (BSP- and octree, and the axis-aligned bounding boxes of the respective geometry) and image-space hierarchy (z-pyramid, tiling hierarchy) to speed-up occlusion queries. Since the hierarchical z-buffer algorithm implies a high complexity for hardware implementations, Greene suggested a variation of the hierarchical polygon tiling algorithm for a less complex approach [13]. Also in 1999, Xie and Shantz suggested a simplified two-level hierarchical z-buffer approach that is more suitable for implementation in hardware [14]. All the hierarchical z-buffer related approaches access the frame-buffer to obtain the depth information, thus increasing hardware complexity and communication requirements.

A more hardware related approach was proposed by Bartz *et al.* [15], where hardware-based occlusion culling functionality was integrated in the per-fragment-operation stage of the OpenGL rendering pipeline, providing detailed information on the occlusion status of queried geometry.

So far, none of these approaches have been implemented in an existing graphics architecture. Possibly the first commercially available occlusion culling hardware was the Denali GB graphics hardware on the Kubota Pacific graphics workstation, implementing a “z-query” to check the z-buffer for contributions [11]. However, this system—and the company—is not available anymore. The only existing implementations are integrated in the VISUALIZE

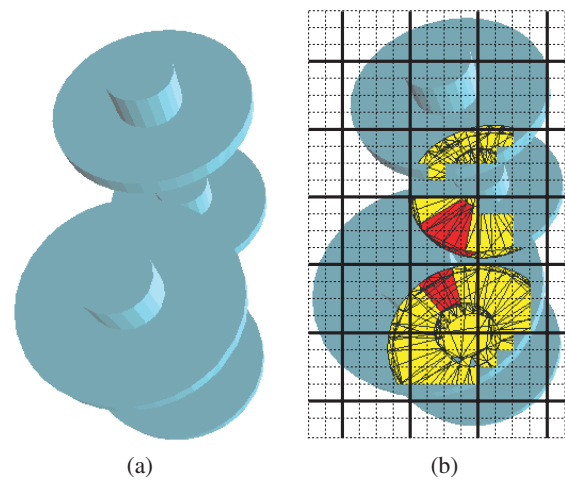


Figure 1: Four wheel hubs of cotton picker scene. (a) Two of the wheel hubs are partially occluded by the front wheel hubs. (b) The culled triangles of the partially occluded wheel hubs are colored in red, the culled pixel groups are colored in yellow. The black grid indicates the applied two-level visibility mask.

fx graphics subsystem series of Hewlett–Packard (HP occlusion culling flag) [16], the graphics subsystem of the Visual PC of SGI [17], and recently announced for nVIDIA’s Geforce3. HP’s and SGI’s implementations check if a bounding volume—which approximates the actual geometry—would be visible if rendered. If the bounding volume is not visible, the bounded geometry is not visible either. These queries are limited to simple yes/no answers and do not allow conclusions concerning how much or which part of a polygonal object is visible. Furthermore, a pipeline synchronization is required to take the result of the query into account.

2. Visibility Driven Rasterization

Visibility driven rasterization represents a new scheme which integrates nicely into existing (“traditional”) occlusion culling schemes. Specifically, it extends rasterization during frame generation in two ways; first, screen-space oriented visibility information is generated several times—during occlusion determination—and stored within the rasterizer in a *visibility mask*, organized as a two-level hierarchy. Second, if contributions to the visibility mask are detected, the established visibility information is exploited additional to “traditional” occlusion culling to cull all triangles and groups of pixels which belong to non-visible areas (see Figure 1).

The visibility information represented in the visibility mask needs to be updated several times per frame to

accomplish good visibility culling performance (in Section 4, we update the visibility mask during every “traditional” occlusion culling test). This update operation (first step) consists of clearing the visibility mask and rendering a set of polygons – i.e. a bounding box or other kinds of bounding volumes – used to determine visibility. The pixels of the rendered polygons which pass the depth test will cause the corresponding bit of the visibility mask to be set. This information is gathered during the “traditional” occlusion culling step, hence no additional z-comparison is required.

In a second step, this visibility information is exploited during rendering of geometry that is located behind the polygons used to establish the visibility. Hence, the visibility mask contains only coverage information based on a set of polygons and the momentary depth test function, but no depth information. The depth information is inherently given, since the geometry (to which the visibility mask is applied) is contained or at least visually blocked by the geometry used to establish the visibility mask. A simple example is to render a bounding box to establish the visibility mask and subsequently render the contained geometry. As we do not store depth but pure visibility information, the visibility mask is very compact. A more detailed description of the algorithm can be found later on in Section 4.

This new approach does not impose any aliasing problems additional to those of the z-buffer. Furthermore, it does not require a front-to-back sorting of the scene objects, although this would improve the culling performance of any image-space occlusion culling algorithm. However, it does require the generation of a scene hierarchy, where bounding volumes (i.e. bounding boxes) are available for each entity. Also, most visibility culling inherent problems are still present with visibility driven rasterization, i.e. translucent objects do not increase occlusion, since they do not occlude other objects. Similar, geometry changing objects (i.e. due to vertex shading or displacement mapping) need to update their respective bounding volume according to the modified geometry.

2.1. Visibility mask

The visibility mask is the core element of our visibility driven rasterization scheme. Each bit of the visibility mask corresponds to a rectangular screen region. A set bit indicates that the geometry used to establish the visibility information is at least partially visible within this area, otherwise it is occluded. A certain granularity has to be selected for the subdivision of the screen space into areas. This is not trivial, since the efficiency of a subdivision scheme depends on the scenes to be rendered. Generally, selecting very large areas per bit of the visibility mask results in a high probability to detect mostly visible areas where

no geometry can be culled from rasterization. On the other hand, selecting a bit for each pixel of the screen space does not achieve much performance improvement in a well balanced pipeline, since at best only idle pipeline cycles can be gained. Furthermore, the finer the visibility mask, the more storage is required (i.e. a viewport of 1024×1024 pixels and one bit per area of 8×8 pixels would require 2 KB to store such a visibility mask). Since the visibility mask has to be accessed at the processing speed of the rasterizer without introducing stall cycles, it must be implemented either as a large register file or as an SRAM module, possibly on-chip. All things considered, the granularity is a trade-off between storage (chip real estate) and culling efficiency.

In the following, we will employ rectangular areas (tiles) of size $n \times m$ for each bit of the visibility mask. The optimal values for n (in x) and m (in y) will be evaluated in Section 4.1, using different polygonal scenes from “real world” applications. To facilitate an efficient implementation in hardware, n and m are chosen as powers of two. Furthermore, since register files and SRAM are organized as addressable space of four, eight, 16, or 32 bit entries, we select 16 bit entries for simplicity. By storing the information of a group of tiles (here $4 \times 4 = 16$ neighboring tiles) in a single entry of the visibility mask, a two-level hierarchy is obtained. This two-level hierarchy can be exploited to cull triangles even more efficiently.

2.2. Culling triangles

The screen space nature of the visibility mask requires that triangles are first transformed and clipped. Once screen space coordinates are available, we can determine whether a triangle resides within a single tile of the visibility mask by testing the address of the vertices of the triangle. If this is the case and the corresponding bit in the visibility mask indicates non-visibility, we can safely cull the triangle, since it resides within one non-visible tile (*trivial reject I*). Depending on the size of tiles and triangles, it frequently occurs that triangles extend over two or more tiles. In these cases, the trivial reject I mechanism will fail. We can avoid this problem for most of the triangles which reside within a group of tiles (4×4 tiles = *tile group*) by using the previously described two-level hierarchy of the visibility mask. If the entire tile group is non-visible then the triangle is culled (*trivial reject II*). Figure 1(b) shows the red colored triangles culled due to trivial reject I and II. Note that for both trivial reject mechanisms, the perspective division of the x and y components is necessary. Only if the triangle cannot be culled, the perspective division for the z component is also required.

Figure 2 illustrates the two-level hierarchy and the trivial reject mechanisms. In this example, triangle “D” is culled by testing it against single tiles (trivial reject I). If all four tile groups are non-visible, all but triangle “B”, which emerges

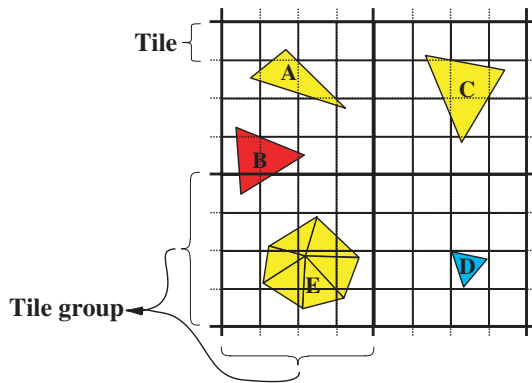


Figure 2: Four tile groups. Each tile represents the area of $n \times m$ pixels on the screen. The visibility information of a tile group (16 tiles, 4×4) is stored in one entry of the visibility mask.

over two tile groups, are culled (trivial reject II). All triangles which cannot be culled will be sent to the rasterization stage, including some partially or entirely non-visible triangles which are addressed in the following section.

Generally, culling of triangles from a well balanced graphics subsystem results in improved performance, if the stages from which load is removed can exploit this additionally available bandwidth. Therefore, to achieve good performance using visibility driven rasterization, the graphics subsystem must be designed with respect to this culling mechanism, i.e. accommodate the bandwidth of the rasterizer and the subsequent pixel processing stages. Nevertheless, performance improvements without changing the balance of the system are achievable, whenever larger triangles (by reducing fill-rate limitations) or multi-texturing (by reducing memory stalls due to page misses) are frequently used.

2.3. Culling groups of pixels

The remaining triangles arriving at rasterization are not necessarily visible, thus we can further save bandwidth on the subsequent pipeline stages by removing groups of pixels of these triangles. Figure 3 illustrates one of the frequent cases where a remaining triangle covers one or more non-visible tiles. During rasterization, the groups of pixels associated with those tiles can safely be culled (see yellow pixels in Figure 1b). After skipping the non-visible tiles, the rasterization continues within visible tiles. Depending on the implemented rasterization scheme (scan-line or stamp based), the exploitation of the resulting idle cycles in existing graphics subsystems requires more subtlety and cleverness which is discussed in the following section.

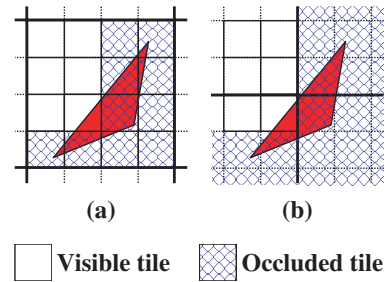


Figure 3: The triangles covering several non-visible tiles of a single tile group (a), of multiple tile groups (b).

3. Hardware Implications

Generally, a rasterizer performs several setup calculations per triangle (such as edge increments for color, texture coordinates, depth value, etc.) and subsequently generates pixels in a certain order. For many years, this was performed in a scan-line based order by stepping along the edges and incrementally generating all pixels between two edge points. Partially driven by the experiences gained in the PixelFlow project [18,19], stamp based rasterization recently became popular. A stamp of i.e. 2×2 pixels is moved across a triangle, potentially generating up to four pixels in each cycle by evaluating three edge equations for each pixel of the stamp [4,20]. Unfortunately, graphics chip companies do not publish details on such implementations. One of the few valuable sources are [20,21], yet how to move the stamp across a triangle within frame-buffer memory pages is only described briefly. Scan-line and stamp based rasterization finally generate pixels including interpolated color, texture coordinates, depth value, etc. In contrast to *visibility driven rasterization*, we will refer to the above described rasterization approaches as *current rasterization*. Figure 4 illustrates the schematic components of a current rasterization scheme. Additionally, the extensions needed for visibility driven rasterization are included.

3.1. Establishing visibility information

From the set of polygons used to establish visibility information, all pixels which pass the momentary depth buffer test will be used to set their corresponding bit of the visibility mask. The address of the tile group and the mask to obtain the corresponding bit can easily be generated using simple logic elements as shift, ANDs, and ORs (see Figure 5 for an example with 16×16 pixel tiles; the four least significant bits of the addresses are ignored to determine the pixel tile). Furthermore, accessing, setting, and writing a bit works well in architectures where at best one pixel passes the depth buffer test per cycle. However, in graphics architectures consisting of multiple pixel processing pipelines, access conflicts need to be resolved. An obvious solution would be to replicate

the visibility mask for each pixel processing pipeline but this increases the overall hardware implementation costs. Alternatively, the accesses can be synchronized by adding a small FIFO buffer to each pixel processing pipeline which stores the tile ID (the ID is a composition of address of the visibility mask entry and the corresponding mask for the specific individual bit) and applying a processing priority given by the number of entries in each FIFO. When reading the value from a FIFO, all other FIFOs having an ID that belongs to the same visibility mask entry will be combined by performing a logical OR operation of the mask bits. To prevent stall cycles due to a full FIFO, each FIFO has a controller that impedes IDs from entering the FIFO, if one of the entries in the FIFO already has the same ID. This mechanism works well as long as the number of pixels per tile is larger than the number of pixels generated per cycle by the rasterizer. (The influence of the tile size onto the overall visibility culling performance is discussed in Section 4.1. However, for the presented datasets tiles of $16 \times 16 = 256$ pixels achieved very good results, while maintaining a good cost/performance ratio.)

The size of the FIFO depends on the number of pixel processing pipelines; i.e. FIFOs of four entries will work sufficiently with four independent pixel processing pipelines. For a higher degree of parallelism in pixel processing, larger FIFOs are necessary. In this case, FIFO controller complexity can be traded versus stall cycles by using a controller which is not fully associative, but only impedes IDs which are equal to the most recent (i.e. four) IDs which entered the FIFO. However, rasterization generally exploits locality of pixels to keep memory accesses efficient and hence, stalls occur infrequently. Furthermore, none of the current single graphics chips processes more than four pixels at a time. The only exception is the non-standard SIMD architecture of the PixelFusion system [22] where the tile size should be tightly related to the size of the SIMD array anyway. Finally, please note that FIFOs are actually not needed for untextured polygons (i.e. the bounding boxes), because only potential texture accesses can put the pixel processing pipelines out of sync.

3.2. Culling triangles

In Section 2.2, we described two trivial reject mechanisms to cull triangles from being rasterized and subsequently processed. Trivial reject II requires the address of the corresponding entry of the tile group in the visibility mask. This address is computed by $\log n$ shift operations of x_{address} , and $\log m$ shift operations of y_{address} for each vertex of the triangle (with $n = m = 16$ in the example in Figure 5). If the resulting bit patterns are identical for all three vertices, the triangle resides within one tile group and assembling the bit patterns generates the address of the corresponding entry in the visibility mask. Finally, non-visibility is given in case the entry is zero and trivial

reject II succeeds. Trivial reject I is evaluated by decoding the corresponding tile of the tile group and checking the resulting bit for non-visibility. A schematic implementation of this mechanism requiring few hardware components is shown in Figure 5. Here, tiles of 16×16 pixels and tile groups of 4×4 are used for a viewport of 1024×1024 pixels. This results in a visibility mask of 256 entries and an overall size of 512 bytes.

3.3. Culling groups of pixels

Removing groups of pixels from the subsequent processing stages during rasterization can be integrated by checking the bit of the corresponding tile in the visibility mask of the pixels to be generated. However, besides avoiding memory stalls possibly due to processing of these pixels, no performance improvement is likely in a well balanced graphics system (see Section 2.3).

To achieve additional performance improvement, the rasterization process must be modified such that skipping non-visible tiles results only in pixels of visible tiles. This skipping mechanism is not trivial, since rasterization is usually performed in an incremental manner and skipping is dependent on the implemented rasterization approach.

In a scan-line based rasterization approach [23,24], multiplications are avoided as often as possible, except in the setup phase. Therefore, while generating pixels between two edge points along a scan-line, color, depth, and other increments (ΔR , ΔG , ΔB , ΔZ) are added to the current values stored in registers (span iteration). A step of a power of two can easily be achieved by shifting the corresponding Δ increment. Steps of 3, 5, 6, 7, etc. are more difficult to implement. We suggest generating an array of increments (Δ , $2*\Delta$, $3*\Delta$, etc.) during the rasterization setup phase and select the according increment to quickly skip non-visible tiles.

For stamp based rasterization, McCormack *et al.* mention that the stamp may also be constrained to generate all fragments in a 2^n by 2^m rectangular “chunk” before moving to the next chunk [20,21]. (The authors state that “(...) chunking is not cheap due to three additional 600-bit save states and associated multiplexers.” but later on refine this statement “(...) we recently discovered that we could have used a single additional wait state.”) Therefore, using chunks of the size of a tile (16×16 pixels), it is possible to directly continue with the next chunk without introducing more than one or a few idle cycles. To avoid colliding read accesses of triangle and pixel group culling stages, this visibility mask should be implemented as dual read memory.

Overall, it is not a trivial task to incorporate such a skipping mechanism into current rasterizers to achieve additional performance improvement, but well worth the benefits as shown in the following section.

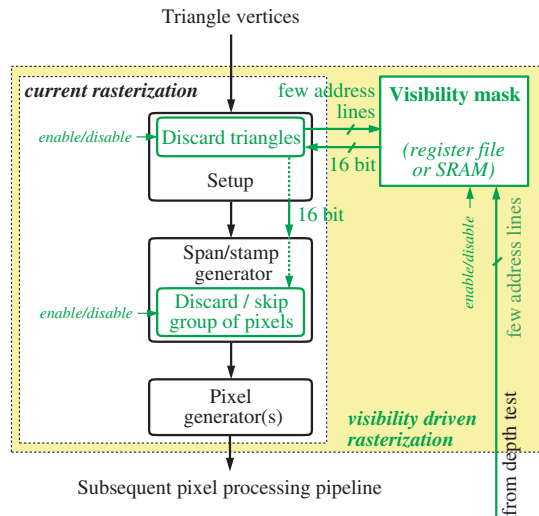


Figure 4: Current and visibility driven rasterization. Yellow areas and green blocks and arrows indicate additionally needed resources for visibility driven rasterization.

4. Results

For the simulation of the potential benefits of our new visibility driven rasterization scheme, we extended the MESA 3D graphics library, an Open Source implementation of the OpenGL graphics API. Two OpenGL modes are added to accommodate the proposed hardware mechanisms; *GL_VISIBILITY_MASK* and *GL_RENDER_VISIBLE*. All entries within the visibility mask will be reset upon enabling *GL_VISIBILITY_MASK*. Subsequently rendered polygons do not affect the content of the frame-buffer, but influence only the visibility mask; for each fragment which passes the momentary depth buffer test, the corresponding bit of the screen area the fragment belongs to will be set. Note that the *GL_VISIBILITY_MASK* mode is almost identical to HP's *GL_OCCLUSION_TEST*, except that we additionally establish the visibility mask, while the graphics pipeline is flushed.

Once the visibility mask is available, it is used in the *GL_RENDER_VISIBLE* mode which applies to the subsequently rendered geometry. The difference between *GL_RENDER* and *GL_RENDER_VISIBLE* is that the latter one employs visibility driven rasterization.

The following code excerpt for the test of a single scene object illustrates the use of these two calls. Modifications to the frame-buffer are impeded by lines 2–3, and enabled again after visibility determination in lines 7–8; the visibility mask is set in lines 4–6, the occlusion information is retrieved in line 9, and the geometry is rendered exploiting the visibility mask in lines 11–13.

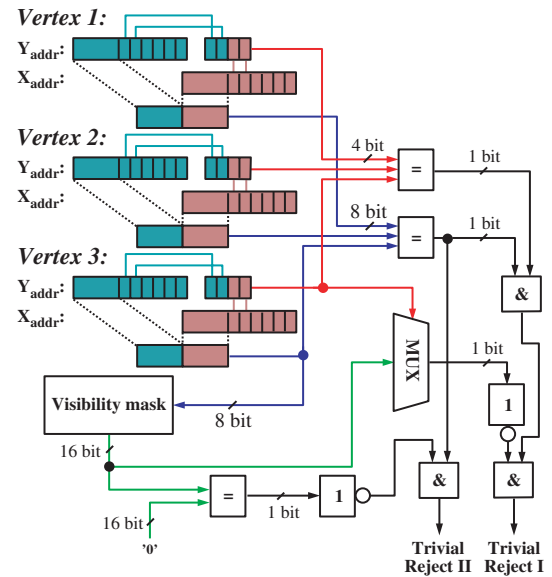


Figure 5: Implementation of the trivial reject I and II mechanisms. Logic for tiles of 16×16 pixels and tile groups of 4×4 tiles is shown. Only very basic and cheap bit operations are required. Test if triangle resides in one tile group (blue arrows), in a single tile (red arrows and subsequent AND), and compare with bit and bit pattern of corresponding visibility mask entry (green arrows).

```

1: glDepthTest(GL_LESS);
2: glDepthMask(GL_FALSE);
3: glColorMask(GL_FALSE, GL_FALSE,
               GL_FALSE, GL_FALSE);
4: glEnable(GL_VISIBILITY_MASK);
5: Render(Bounding Volume);
6: glDisable(GL_VISIBILITY_MASK);
7: glColorMask(GL_TRUE, GL_TRUE,
               GL_TRUE, GL_TRUE);
8: glDepthMask(GL_TRUE);
9: glGetBooleanv(GL_VISIBILITY_MASK,
                 &fResult);
10: if (fResult == TRUE) {
11:   glEnable(GL_RENDER_VISIBLE);
12:   Render(Actual_Geometry);
13:   glDisable(GL_RENDER_VISIBLE);
14: }.

```

4.1. Experiments

We measured the potential quantitative efficiency of our novel visibility driven rasterization using the above described extension of MESA on a sequence of frames

Table 1: Set of scenes (describing the polygonal complexity and the granularity of the respective decomposition tree leaf nodes) used to evaluate the potential benefits of visibility driven rasterization

Scenes / source	Granularity (min, max)	No. of Triangles	No. of Leaf nodes
Cotton picker CAGD	6 9,416	10,605,158	13,257
Screwdriver CAGD	17 13,338	156,424	83
Ventricles MRI/MC	9 29,593	270,882	266
Cathedral CAGD	3 999	391,868	133

of four, quite diverse scenes – summarized in Table 1. Each scene is organized in a decomposition tree where the geometry is stored in the leaf nodes of the tree (each object in an individual leaf node). The respective decomposition trees are derived from the assembly lists of the CAGD models, or specific decomposition algorithms. See [25] for a more detailed discussion on quality of the various approaches.

First, we apply view-frustum culling on the decomposition tree which also requires the computation of the (minimal and maximal) z-values of the tree nodes associated bounding boxes. We use these z-values to depth-sort the leaf nodes of the tree to perform efficient occlusion culling to all geometry (leaf) nodes which are located within the view-frustum. (As already mentioned in Section 2, this depth-sorting is used to increase efficiency; it is not required.) The geometry will be entirely culled, if no contribution to the visibility mask is detected (HP occlusion culling flag by testing fResult). Second, we perform three different tests of visibility driven rasterization (trivial reject I, II, and pixel groups) and measure the culled triangles and pixels to obtain the additional potential culling of our visibility driven rasterization. The average of the remaining triangles and pixels is given in Table 2. Note that all reported culling performance is achieved additionally to “traditional” occlusion culling approaches like [9–11] – or in our case the HP occlusion culling flag based approach as described above [26] – and introduce no additional latency in the modified graphics pipeline. Furthermore, the numbers in Table 2 and Figures 7–9 are referring to the number of triangles and pixels after clipping of the triangles which belong to the geometry nodes that are only partially located within the view-frustum.

Table 2: Number of triangles and pixels (in thousands) remaining after view-frustum and occlusion culling and clipping (tris and pix), as well as remaining pixels (pix1) and triangles (tris1) after trivial reject I and II (using a tile size of 16×16 pixels), and the finally remaining pixels after culling of pixel groups (pix2). These numbers compare to tris3, the number of the actual visible triangles, and pix3, the number of the actual visible pixels (both after z-buffer test)

	Cotton picker (K)	Screwdriver (K)	Ventricles (K)	Cathedral (K)
tris	548	47	10	11
pix	2,572	1,228	3,214	5,021
tris1	408	27	7	9
pix1	2,430	610	2,953	4,924
pix2	1,346	350	1,889	3,220
tris3	305	12	2	2
pix3	341	102	1,089	1,134

4.1.1. Cotton picker

The cotton picker is a “real world” model from an industrial CAGD modeling package which contains 13,257 individual parts in its assembly list (see Figure 6a–c). The decomposition tree has been generated, based on this assembly list. Most of the geometric complexity is located in the six spindle compartments (1,633,137 triangles each) containing the spindle drums (694,113 triangles each) which collect the cotton flakes. These drums in particular are usually occluded by covers and chassis parts. However, from a frontal point of view, all spindle drums are at least partially visible which eliminates the potential culling benefits for a traditional occlusion culling approach. All results are averaged over a sequence of frames of twelve arbitrary views (on a sphere centered around the datasets), where the cotton picker is always completely contained in the view-frustum. An interesting effect is that the percentage of culled triangles of trivial reject I and II is significantly larger than the respective percentage of culled pixels (see Table 2 and Figure 8). This is due to the fact that only smaller triangles, which fit into a single tile or into the tile group, are culled at this stage.

4.1.2. Screwdriver

Similar to the cotton picker, the screwdriver scene is a “real world” model from an industrial CAGD modeling package. It contains 83 individual parts in its assembly list, where the chassis consists of two parts, occluding most of the geometry of this model (see Figure 6d). Based on the assembly list, the decomposition tree has been generated. The sequence of frames uses twelve arbitrary views (on a sphere centered around the datasets), where the screwdriver is always completely contained in the view-frustum.

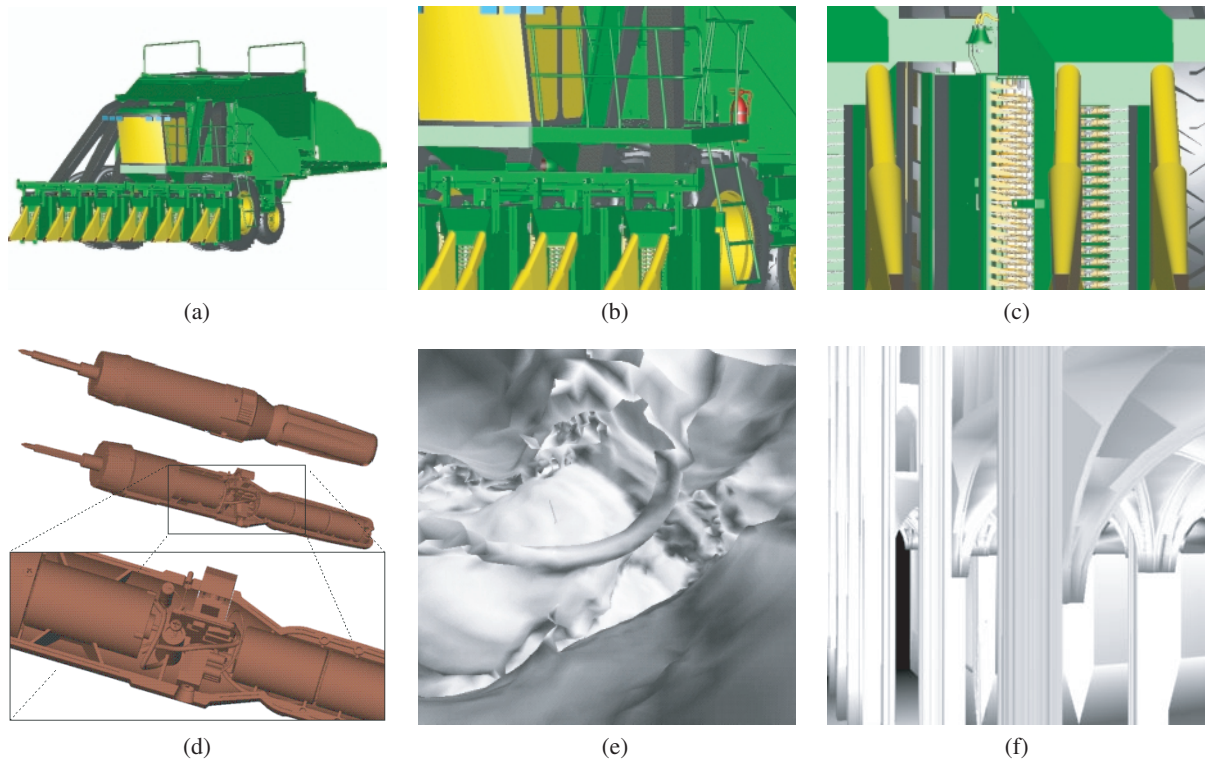


Figure 6: Images of different scenes. (a)–(c) Cotton picker. (b) Close-up of (a). (c) Further close-up of (a) showing the spindle drums. (d) Screwdriver; top: screwdriver with complete chassis; middle: one chassis part removed; bottom: close-up of middle image. (e) Interior view of ventricular system. (f) Interior view of cathedral.

4.1.3. Ventricular system

The ventricular system (ventricles) is extracted from a pre-segmented MRI volume dataset using MarchingCubes (MC, see Figure 6e). The associated decomposition tree has been generated by an octree decomposition of the volume dataset. It is a typical iso-surface model generated from a volumetric representation with a high occlusion depth for interior views and very regular shaped triangles of similar (object space) size. The sequence of frames consist of 150 individual viewpoints which are located inside of the ventricular system along a camera path.

4.1.4. Cathedral

The cathedral is an architectural CAGD scene modeled with a customized modeling package (see Figure 6f). It contains numerous long and narrow triangles of various sizes. The decomposition tree has been generated from the unordered triangles of the model by an automatic decomposition generator [25]. The tall nave and transept of the building impede efficient results using traditional bounding box based occlusion culling approaches due to deep visibility and missing of suitable occluders. The difference of the

percentage of culled triangles and culled pixels (see Table 2 and Figure 8) is due to the small size of the culled triangles (see also discussion of cotton picker model). The sequence of frames consist of 100 individual viewpoints which are located inside of the cathedral.

4.2. Trivial reject I

Using trivial reject I, all triangles residing within a single tile are culled. Figure 7 shows the improvements possible due to trivial reject I for different square tile sizes, after view-frustum and occlusion culling. While up to only 20% of the remaining triangles are culled for the cathedral, almost 40% are culled for the other scenes. The relatively low cull-rate for the cathedral is due to the columns and arches, which are poor occluders. Nevertheless, the triangles behind these occluders are culled using visibility driven rasterization, while traditional occlusion culling can only cull them by applying a finer decomposition of the scene. However, only a certain amount of occlusion culling tests can be performed interactively, due to the high costs of the occlusion query (We use the HP occlusion culling flag, which requires a pipeline synchronization).

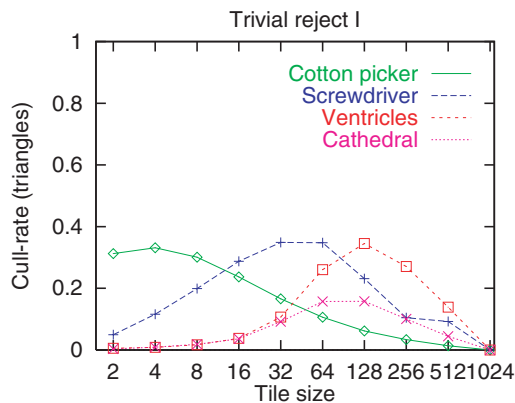


Figure 7: Cull-rate for trivial reject I relative to the triangles which remain after view-frustum and occlusion culling depending on square tile size.

Generally, the peak performance of trivial reject I depends on the size of the triangles of each scene. On average, the cotton picker has very small triangles (2.4 pixels) – due to the spindle drums within the spindle compartments – and therefore, small tiles achieve the best cull-rates. In contrast, the cathedral and the ventricles consist of large triangles (on average 180 and 465 pixels) and for small tiles these triangles reside frequently across multiple tile groups and hence, cannot be culled with trivial reject I only. The screwdriver consists of triangles with on average 149 pixels and achieves best results with tiles of 16×16 to 64×64 .

4.3. Trivial reject I and II

While trivial reject I can only be applied to single tiles, trivial reject II exploits the second level of the visibility mask by testing 16 entries for non-visibility. The results of trivial reject I and II are shown in Figure 8. Compared to trivial reject I, the absolute cull-rates do not increase very much. (Most of the culled triangles of the cotton picker are quite small, hence we only obtain limited pixel savings; see Table 2.) However, for the screwdriver, cathedral, and ventricles the effective cull-rate range has broadened, achieving better cull-rates for a broader set of tile sizes. This stretched cull-rate range is important to determine tile sizes which are efficient for a wide range of different polygonal scenes (i.e. the screwdriver achieves good results already with tiles of 8×8 pixels, the cathedral with 16×16 , and the ventricles with 32×32). Overall, between 20 and 40% of the remaining triangles can be culled using trivial reject I and II.

We can also observe that visibility driven rasterization of the cotton picker model performs best with small tile sizes. This is due to the large majority of very small triangles modeling the highly detailed elements (i.e. spindle drums

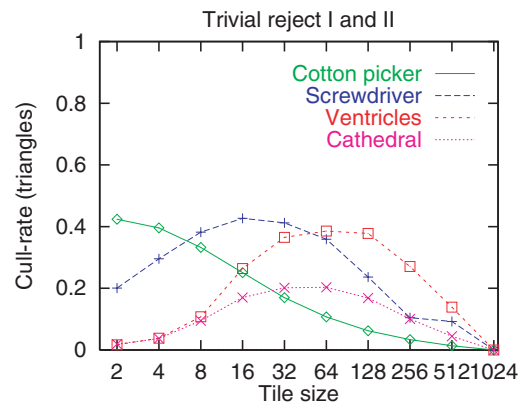


Figure 8: Cull-rate for trivial reject I and II relative to the triangles which remain after view-frustum and occlusion culling depending on square tile size.

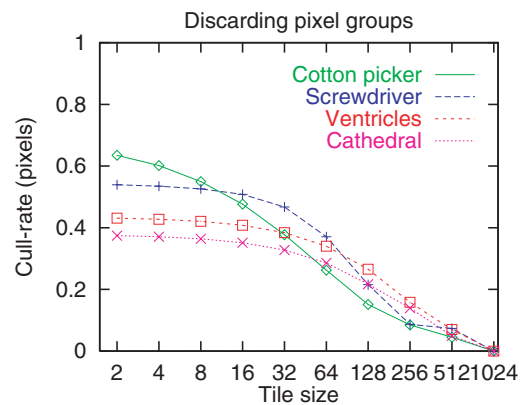


Figure 9: Cull-rate for culling pixel groups relative to the triangles which remain after view-frustum and occlusion culling, and trivial reject I and II depending on square tile size. Absolute values are given in Table 2 (pix2).

in the spindle compartments). However, industrial CAD rendering applications (i.e. EAI Vis MockUp) use a lower level of detail of a multi-resolution representation of the model, which significantly increases the average triangle size. This larger average triangle size results in a cull-rate function drop off at larger tiles, resulting in similar culling curves as for the other scenes.

4.4. Culling groups of pixels

In contrast to trivial reject I and II – which reduce the number of triangles and hence, the setup in the rasterizer – culling groups of pixels removes load from the pixel processors (fill-rate). Figure 9 shows the additional cull-rate of pixels

remaining after view-frustum and occlusion culling, and trivial reject I and II.

Not surprisingly, culling pixel groups works best for smallest tiles and decreases as the size of the tiles increases. For the screwdriver, cathedral, and ventricles, tiles of up to 16×16 pixels achieve almost the same cull-rate as tiles of 2×2 pixels. Only for the cotton picker, tiles of at most 8×8 should be used to maintain a high culling efficiency. This is due to the large number of very small triangles rendered in our visibility driven rasterization system, which does not exploit level-of-detail selection (see Section 4.3).

4.5. Discussion

Generally, tiles of 16×16 (4 Kbit visibility mask) and 32×32 (1 Kbit) achieve good cull-rates, while maintaining a good cost/performance ratio. This translates into a visibility mask of 1–4 Kbit (128–512 bytes) which can be integrated into graphics subsystems using either a large register file or an on-chip SRAM. For the presented scenes, such visibility masks facilitate the culling of up to 40% of all remaining triangles (after “traditional” view-frustum and occlusion culling and clipping) and hence, no rasterization setup needs to be performed for these triangles. Additionally, visibility driven rasterization culls between 40 and 55% of all further remaining pixels (after both trivial rejects). Overall, integrating visibility driven rasterization into graphics subsystems has the ability to significantly improve and potentially doubles the frame-rates for the presented datasets. Moreover, the ability to cull triangles and pixels from the pixel processors is increasingly important when using multi-texturing, where systems are mostly fill-rate limited.

Further improvements for trivial reject II can be achieved by extending the two-level hierarchy to a three- or four-level hierarchy with a few more logic operations (i.e. with the proposed 16 bit, a three-level visibility mask can be realized by additionally checking whether the triangle resides entirely within one of the four disjoint 2×2 tile subgroups). This could naturally be extended to a four-level hierarchy, using a register file with 64 bit entries. The advantage would be that starting out with each bit of the visibility mask representing tiles of 8×8 pixels, even 64×64 large tiles would be covered with the same visibility mask. Overall, the impact of trivial reject II can be increased even more by investing a little more logic.

Tiles consisting of pixel spans are as well feasible by choosing $m = 1$. However, we observed in additional experiments that spans result in an inefficient average triangle and pixel culling performance, and therefore, we focused on quadratic sized tiles.

In Table 2, we compare the achieved triangle and pixel (group) culling with the actual number of visible triangles

and pixels (optimal result). Please note that these numbers are based on the final z-buffer rasterization of the geometry. The discrepancy between these numbers (tris3 and pix3) and the numbers due to visibility driven rasterization (tris1 and pix2) exhibits further culling potential. The coarse bounding volume approximation of the actual geometry (axis-aligned bounding boxes) – in particular the fact that all associated geometry is represented by the nearest z-value of the bounding volume – accounts for most these differences which circumvents the successful pixel (and triangle) culling within a leaf node. Other reasons include triangles which are not aligned on the tile (group) grid.

5. Conclusions and Future Work

We presented a novel visibility driven rasterization scheme, capable of accelerating the rendering of scenes with a high depth complexity. A small and compact two-level visibility mask has been introduced to represent visibility information, which enables our new rasterization scheme to remove a significant number of triangles before rasterization setup and pixel groups during rasterization, in addition to “traditional” view-frustum and occlusion culling techniques.

For a visibility mask of 16×16 pixels, storing the information of 4×4 tiles in one entry of the visibility mask, only 4 Kbit of memory are required, while being able to cull up to 40% of the geometry and 55% of the pixels in “real world” datasets. The saved bandwidth can either be invested into richer pixel operations such as multi-texturing, or in rendering more polygons, if there is sufficient bandwidth available on the vertex bus. Removing triangles from the rasterization and subsequent pipeline stages will be of even stronger importance once higher order primitives (i.e. partial visible NURBS surfaces) are sent right to the graphics subsystem, since the reduced traffic on the front (vertex) bus will also reduce the bottle-neck. Generally, visibility based rasterization achieves best results when correspondingly balancing the graphics subsystem in the first place.

Our future work will focus on how to avoid switching render modes, currently necessary to establish the visibility information. The associated synchronization costs **and latency** could be eased, if we interleave the rendering of the bounding geometry and the actual geometry by providing a double buffered visibility mask. Another improvement is to extend the trivial reject II test to allow various patterns of visible/non-visible tiles within one tile group.

Acknowledgements

This work is supported by project SFB 382 of the German Research Council (DFG), by the MedWis program of the German Federal Ministry for Education and Research, and by the Workstations Systems Lab, Ft Collins, CO of the Hewlett-Packard company. The cotton picker and

screwdriver scenes are courtesy of Engineering Animation Inc. Last but not least, we would also like to thank Michael Doggett for proof-reading and the anonymous reviewers for their helpful comments.

References

1. K. Akeley. RealityEngine graphics. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 109–116. Anaheim, CA, USA, August 1993.
2. J. Montrym, D. Baum, D. Dignam and C. Migdal. Infinite reality: a real-time graphics system. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 293–303. Los Angeles, CA, USA, July 1997.
3. Silicon Graphics Inc.. Silicon Graphics 320, Visual Workstation. *Specification document, available from <http://visual.sgi.com/products/320/index.html>*, 1999.
4. J. McCormack, R. McNamara, C. Gianos, L. Seiler, N. Jouppi and K. Correll. Neon: a single-chip 3D workstation graphics accelerator. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 123–132. Lisboa, Portugal, August 1998.
5. nVIDIA. GeForce. <http://www.nvidia.com/Geforce256.nsf>, 1999.
6. B. Anderson, R. MacAulay, A. Stewart and T. Whitted. Accommodating memory latency in a low-cost rasterizer. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 97–101. Los Angeles, CA, USA, July 1997.
7. H. Igehy, M. Eldridge and K. Proudfoot. Prefetching in a texture cache architecture. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 133–142. Lisboa, Portugal, August 1998.
8. D. Kirk. Unsolved problems and opportunities for high-quality, high-performance 3D graphics on a PC platform. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 11–13. Lisboa, Portugal, August 1998.
9. H. Zhang, D. Manocha, T. Hudson and K. Hoff. Visibility culling using hierarchical occlusion maps. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 77–88. Los Angeles, CA, USA, July 1997.
10. D. Bartz, M. Meißner and T. Hüttner. OpenGL-assisted occlusion culling of large polygonal models. *Computers and Graphics – Special Issue on Visibility – Techniques and Applications*, 23(5):667–679, 1999.
11. N. Greene, M. Kass and G. Miller. Hierarchical z-buffer visibility. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 231–238. Anaheim, CA, USA, August 1993.
12. N. Greene. Hierarchical polygon tiling with coverage masks. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 65–74. New Orleans, USA, August 1996.
13. N. Greene. Occlusion culling with optimized hierarchical buffering. In *Visual Proceedings of ACM SIGGRAPH*, page 261. Los Angeles, CA, USA, July 1999.
14. F. Xie and M. Shantz. Adaptive hierarchical visibility in a tiled architecture. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 75–84. Los Angeles, CA, USA, August 1999.
15. D. Bartz, M. Meißner and T. Hüttner. Extending graphics hardware for occlusion queries in OpenGL. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 97–104. Lisboa, Portugal, August 1998.
16. N. Scott, D. Olsen and E. Gannett. An overview of the VISUALIZE fx graphics accelerator hardware. *The Hewlett-Packard Journal*, 28–34, May 1998.
17. SGI. Silicon Graphics Visual Workstation OpenGL Programming, *Technical Report*, SGI, 1999.
18. J. Eyles, S. Molnar, J. Poulton and T. Greer. PixelFlow: the realization. In *Proc. of Eurographics Workshop on Graphics Hardware*, pages 23–30. Los Angeles, CA, USA, July 1997.
19. S. Molnar, J. Eyles and J. Poulton. PixelFlow: high-speed rendering using image composition. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 231–240. Chicago, USA, July 1992.
20. J. McCormack, R. McNamara, C. Gianos, L. Seiler, N. P. Jouppi, K. Correll, T. Dutton and J. Zurawski. Neon: a (big) (fast) single-chip 3D workstation graphics accelerator, *Technical Report 98.1*, Compaq Western Research Laboratories, 1998.
21. J. McCormack and R. McNamara. Tiled polygon traversal using half-plane edge functions. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 15–21. Interlaken, Switzerland, August 2000.
22. R. McConnell. Massively parallel SIMD computing on a single chip. In *Proc. of the 11th Symposium on High-Performance Chips at Stanford University*. 1999.
23. A. Kugler. The setup for triangle rasterization. In *Proc. of Eurographics Workshop on Graphics Hardware*, pages 49–58. Poitiers, France, August 1996.

24. M. Waller, J. Ewins, M. White and P. Lister. Efficient primitive traversal using adaptive linear edge function algorithms. *Computers & Graphics*, 23:365–375, 1999.
25. M. Meißner, D. Bartz, T. Hüttner, G. Müller and J. Einighammer. Generation of subdivision hierarchies for efficient occlusion culling of large polygonal models, *Technical Report TR WSI-99-13*, University of Tübingen, 1999.
26. D. Bartz and M. Skalej. VIVENDI—a virtual ventricle endoscopy system for virtual medicine. In *Data Visualization*, Proc. of Symposium on Visualization, pages 155–166, 324. 1999.