Granular Visibility Queries on the GPU

Thomas Engelhardt Carsten Dachsbacher*
University of Stuttgart

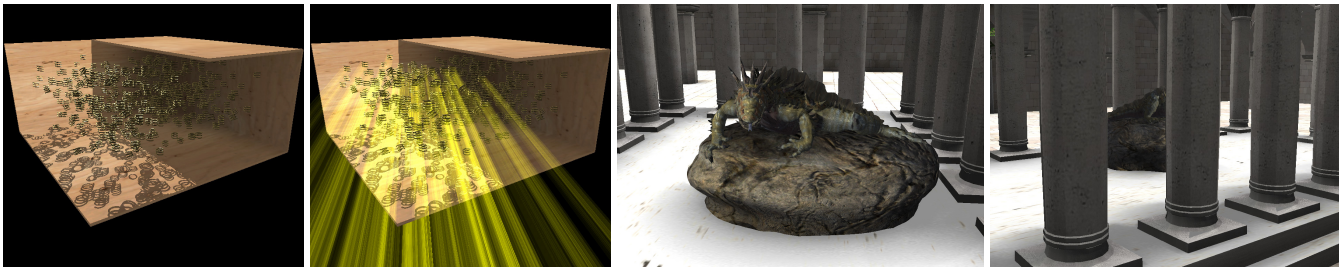


Figure 1: *Left: Shadow volumes are extruded in a geometry shader and granular visibility queries are used directly on the GPU to exclude unlit objects from the shadow volume generation. Right: Visibility determination on the GPU is also beneficial in costly rendering techniques such as displacement mapping to cull individual, occluded primitives.*

Abstract

Efficient visibility queries are key in many interactive rendering techniques, such as occlusion culling, level of detail determination, and perceptual rendering. The occlusion query mechanism natively supported by GPUs is carried out for batches of rendered geometry. In this paper, we present two novel ways of determining visibility by intelligently querying summed area tables and computing a variant of item buffers. This enables visibility queries of finer granularity, e.g., for sub-regions of objects and for instances created within a single draw call. Our method determines the visibility of a large number of objects simultaneously which can be used in geometry shaders to cull triangles, or to control the level of detail in geometry and pixel shaders under certain rendering scenarios. We demonstrate the benefits of our method with two different real-time rendering techniques.

CR Categories: I.3.7 [Three-Dimensional Graphics and Realism]: Visible line/surface algorithms;

Keywords: real-time rendering, GPUs, visibility queries

1 Introduction

Visibility determination is a widely studied topic in computer graphics and is required for many applications such as walk-throughs for large scale environments [Funkhouser et al. 1996], computer games and CAD modeling systems. Its ultimate goal is to identify occluded portions of a scene which do not contribute to the final image. Many algorithms have been proposed to solve this *occlusion culling* problem.

*e-mail: {engelhardt, dachsbacher}@visus.uni-stuttgart.de

Recent approaches in real-time computer graphics often deploy *hardware occlusion queries* (HOQs) which allow the user to determine the number of visible pixels of rasterized geometry, e.g. bounding volumes, against the depth buffer. The query result is read back to the application and a decision made whether to render the objects or not. The drawback of reading back the result, and thus synchronizing the GPU and the CPU, is eliminated by conditional rendering available with the OpenGL extension `GL_NV_conditional_render` (the Direct3D counterpart is called occlusion predicate rendering). First, the geometry is rasterized to generate the predicates, and later predicated draw calls are issued: rendering is automatically omitted, if the query result is available and full occlusion has been detected, i.e., predicates are non-stalling and are therefore not guaranteed. Conditional rendering is perfectly suited for per-object occlusion culling, and HOQs have proven very effective for occlusion culling methods using stop-and-wait approaches [Bittner et al. 2004; Mattausch et al. 2008]. The latter are also used in methods using quantitative visibility, e.g. for aggressive culling, however, in such cases attention has to be paid to hide the read back latencies.

Both mechanisms have two restrictions in common. First of all, the finest granularity is a single draw call, i.e. multiple queries are required if the visibility of parts of an object is to be determined, possibly resulting in a large number of draw calls. Furthermore, when using instanced geometry a single draw call creates multiple object instances and it is not possible to query the instances' visibility separately. Consequently, culling instances individually is not possible at all. Second, the visibility results are not directly available in shaders (with predicates they are not available at all), where they can be useful to cull object parts, or to control the level of detail of tessellation and involved rendering techniques. Lastly, the results of a query depend on the contents of the depth buffer at the time when the geometry is rasterized and thus on the rendering order. Thus the scene has to be rendered twice to resolve mutual occlusion of objects (z-only pass, followed by the visibility test).

In this paper, we first address the granularity of the queries and present a method to determine visibility of sub-regions of objects and of instanced geometry. The results are generated and directly available on the GPU and can be accessed in shaders. We believe that this has a range of applications, such as the previously mentioned culling in geometry shaders and LOD control in geometry and pixel shaders under certain rendering scenarios. Determining the visibility of objects (in an output-sensitive manner) is basically

a problem of counting pixels of rasterized geometry. We analyze two different approaches to this problem: The first approach intelligently assigns objects to color channels and renders them in pure red, green, blue, or into the alpha channel of an off-screen color texture. Summed area tables (SATs) [Hensley J. and Lastra 2005] allow a quick retrieval of the sum of pixels in an image region and thus to count the pixels covered by an object or sub-regions thereof. Our second approach extends the well-known item buffer such that IDs are automatically generated for object instances and sub-regions. By using a fast counting method on the GPU (similar to computing image histograms) and mip mapping, we can efficiently perform visibility queries.

Please note that our method is expedient in particular situations only: Inherent to pixel counting approaches is that query and occluder objects need to be rendered before determining visibility. Futher, GPUs also cull at fine granularity: Occluded fragments are discarded prior to shader execution using a hierarchical depth buffer. Consequently our method can improve pixel shader performance only if this mechanism becomes disabled, e.g. due to depth writes or discards in pixel shaders. Thus the main application of our method is to speed up rendering techniques with costly geometry shaders (GS). Again, geometry generated in the GS itself cannot be used instantaneously as an occluder, because pixel counting is based on the result of a preceding render pass. We discuss two exemplary applications for our method: culling of individual prisms in displacement mapping and shadow volume extrusion for instanced objects (see teaser). We analyze the benefits and problems of our *granular visibility queries* (GVQs) and the implementation on current GPUs.

2 Previous Work

Visibility determination is of central interest in computer graphics and has become increasingly important with the growth in complexity of models, especially for interactive applications. The amount of research conducted in this field is enormous and covering the entire literature is beyond the scope of this paper. A classification of visibility problems and comprehensive overviews are given by [Cohen-Or et al. 2003] and [Bittner and Wonka 2003].

Apart from hardware occlusion queries (HOQs) various other methods exist which determine visibility in image space in on output-sensitive way: The hierarchical z-buffer (HZB) [Greene et al. 1993] refers to a resolution pyramid of a depth buffer allowing the testing of primitives as a whole for occlusion efficiently. Optimizations with regard to hardware implementations have been presented in [Greene 2001] and similar concepts are implemented on GPUs to cull blocks of pixels before pixel shader execution. A different hierarchical representation of depth maps has been presented by [Décoret 2005] which can be queried in GPU programs. In contrast to an image pyramid this method stores the value and position of local depth extrema in an image cube. Hierarchical occlusion maps [Zhang et al. 1997] store opacity and not depth information which allows approximate visibility culling by using an opacity threshold. For this occluders and occludees are distinguished and the visibility test is decoupled into an overlap test (do objects overlap in screen space), and a depth test (are occluders closer to the viewer than occludees). Occupancy maps [Staneker et al. 2003] aim at reducing the number of visibility queries: A low-resolution version of the frame buffer is stored as a bitmask to determine efficiently if an object is visible or possibly occluded (which is then verified with a standard HOQ). A well-known technique is the item buffer (e.g. used by [Klosowski and Silva 2000]): A unique color is assigned to each object (or triangle) and after the geometry is rasterized its the visibility is determined by counting the pixels with the respective color. [Scheuermann and Hensley 2007] demonstrate how to efficiently compute histograms, i.e. effectively counting pixels, on contemporary graphics hardware, and it is to be expected that even faster implementations are possible with scatter operations on upcoming GPUs.

Recent GPUs support predicated or conditional rendering [OpenGL Extension Registry 2008]: The application sends the bounding volume of an object to the pipeline to test its visibility. Afterwards, preferably at a much later time, the GPU is instructed to render the object itself and automatically excludes it from rendering if no pixel of the bounding volume passed the depth test (without any feedback to the application). However, if the result of the predicate test is not yet available then the object is rendered in any case.

The HOQ mechanism is designed to accelerate *from-point* visibility determination and is a substantial building block for many such algorithms. For example, many implementations of cell and portal algorithms [Luebke and Georges 1995] determine the visibility of portals with HOQs. [Yoon et al. 2003] and [Bittner et al. 2004; Mattausch et al. 2008] demonstrate how to use HOQs efficiently and how to exploit spatial and temporal coherence of visibility. [Guthe et al. 2006] apply a statistical model to describe occlusion probability of queries in order to reduce the number of wasted queries. *From-region* visibility determination is considered harder and recent conservative algorithms only reach interactive performance for moderately complex scenes. Thus many of these methods include an off-line preprocessing phase whereas recent methods, such as [Leyvand et al. 2003], also apply HOQs to speed up the precomputation.

The remainder of this paper is structured as follows: In the next section we present our methods in detail, followed by the description of its application to different rendering techniques. In Section 4 we present results which we discuss in Section 5.

3 Granular Visibility Queries

The main goal of our method is to provide efficient means for counting the number of pixels covered by the query objects on the screen. However in contrast to HOQs, we want to be able to query the visibility of sub-regions of objects individually, and to make the results available to shaders directly on the GPU.

We analyze two possible approaches for fast pixel counting: First, we use summed area tables to retrieve the average color of an image region, and thus retrieve the screen coverage of query objects once we rendered them appropriately colored. Second, we introduce a new variant of item buffers which can be quickly rendered and evaluated on contemporary GPUs.

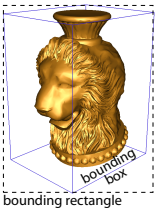
3.1 Pixel Counting with Summed Area Tables

The basic idea of our first approach is to render query objects into a color texture and count the pixels belonging to each object afterwards. Two GPU-friendly methods exist to accelerate this process: summed area tables (SATs) and mip maps allow us to retrieve the accumulated or average color within a rectangular image region efficiently. Both treat color channels independently and by rendering query objects in pure red, green, blue, or into the alpha channel, we can distinguish pixels of up to four objects per image region (given a standard four channel texture). The crucial point for a practical algorithm is to develop a good heuristic for coloring the objects and constructing an as minimal as possible set of texture accesses to determine the visibility of an object. We focus our discussion in this section on SATs and will comment on mip maps in Sect. 3.1.4.

Our heuristic is a three-step process performed every frame: First, we assign query objects to color channels such that no overlapping

and fusing in image space occurs (step 1). Obviously, this is not possible if more query objects overlap than color channels are available. In this case we cannot assign such *conflict objects* as a whole to a color channel – instead we assign portions of these objects to different color channels (step 2). In step 3, we determine a set of texture accesses to the SAT for each query object to retrieve the visibility. Step 1 to 3 are executed on the CPU and require access to the objects' transformations and bounding boxes. The SAT computation takes place on the GPU, and the visibility queries are then possible in any shader program.

3.1.1 The Conflict Graph



The goal of the first step of our algorithm is to assign as many query objects as possible to color channels in a way such that fusion in image space does not occur. For this we build a conflict graph: Each vertex of the graph corresponds to one query object. A *conflict* between two objects exists if their bounding rectangles in image space overlap; in this case the graph contains an edge between the two corresponding vertices. We chose this conservative overlap test particularly with regard to the use of SATs: In favor of fewer texture accesses we want to avoid non-rectangular regions in advance. Please note that we do not assume a depth sorting of the objects as this does not provide any advantages unless we perform an occlusion culling procedure beforehand.

For coloring the conflict graph's vertices, we adapt the Chaitin-Briggs algorithm from the domain of compiler design [Muchnick 1997]: In its original application it is used for register allocation, In our case, the algorithm must not assign two objects in conflict to the same color channel. During graph coloring, conflict objects are identified and separated for special treatment afterwards. Optimal coloring is a NP-complete problem, the graph coloring algorithm (possibly yielding sub-optimal results) is $O(n^2)$. The first phase of the algorithm is the graph decomposition: We find the vertex with the least incident edges, remove this vertex and all its edges from the graph, and store it on a stack. This is repeated until the entire graph has been decomposed. The second phase reconstructs the graph and assigns colors; the following steps are repeated until the stack is empty:

- Take the top-most vertex from the stack, reinsert it into the graph and restore edges to adjacent vertices, if they have already been reconstructed.
- If the vertex does not have any neighbors yet, assign it to an arbitrary color channel, otherwise select a channel which is not used by one of its neighbors.
- If no color channel is available, mark this vertex as *conflict object* and do not insert it into the graph (it will undergo special treatment afterwards).

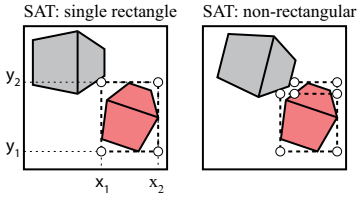


Figure 2: *Left: The query of a rectangular region from a SAT requires 4 texture lookups (shown as circles). Right: For non-rectangular query regions samples can often be reused.*

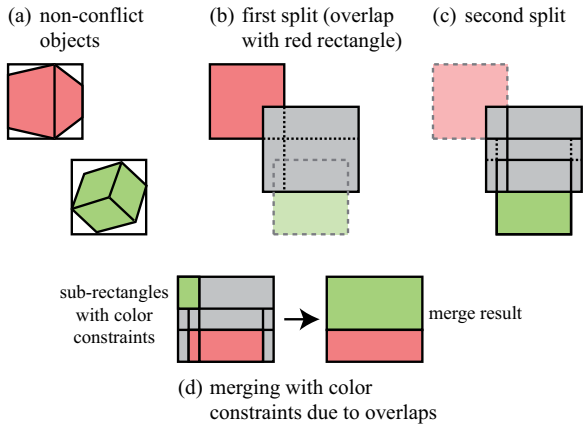


Figure 3: *A conflict object is split recursively (b and c); each rectangle keeps track of free color channels (there are only two channels in this example). (d) Adjacent sub-rectangles are merged iteratively if they can be assigned to the same color channel.*

This algorithm removes vertices with fewer incident edges with higher priority. During graph reconstruction the vertices are reconstructed in the opposite order, effectively coloring objects in more difficult configurations first, and coloring unproblematic objects (with few or no conflicts) later. The output of the graph coloring are two sets of objects: *Non-conflict objects* which have been assigned to color channels as a whole, and *conflict objects* which could not be assigned.

3.1.2 The Region Set

A SAT allows us to compute the sum of pixel colors in an axis-aligned rectangular region $[x_1, x_2] \times [y_1, y_2]$ in image space, with $0 \leq x_1 \leq x_2 \leq 1$ and $0 \leq y_1 \leq y_2 \leq 1$ easily. Each query of a rectangular region requires four lookups to the SAT. Non-rectangular shapes can be approximated with multiple rectangles and often samples can be shared among the queries (see Fig. 2). Of course, the SAT will be generated at a finite resolution (1024^2 in our examples) and $x_i, y_i \in [0; 1]$ are mapped to pixels accordingly.

A *region set* represents the information where to lookup the SATs in order to determine the area covered by an object, i.e. it contains $i \geq 1$ SAT-lookups consisting of a rectangle and an associated color channel: $([x_{i,1}, x_{i,2}] \times [y_{i,1}, y_{i,2}], c_i)$.

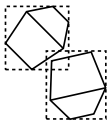
The graph coloring assigns non-conflict objects to color channels and the visibility for every such object corresponds to a single SAT query: The rectangular region corresponding to the object's bounding rectangle (Fig. 3a). Next, we need to split and assign parts of the conflict objects to color channels. All operations are performed on the bounding rectangles in image space only, thus keeping the necessary operations simple and efficient. We process one conflict object at a time by splitting its bounding rectangle and keeping track of free and occupied color channels for each sub-rectangle. The following steps are performed for each conflict object:

- Find the next overlapping SAT lookup (that has already been created) and split the conflict object into axis-aligned rectangles (Fig. 3b shows a split operation)
- For each sub-rectangle we store flags indicating which color channels are already occupied by other objects. The flags of the overlap region (of the object to be inserted) are updated accordingly after the split.

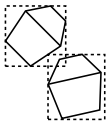
- We proceed recursively for all sub-rectangles and test for overlaps with the remaining SAT-lookups (Fig. 3c).
- The sub-rectangles form a partition of the initial bounding rectangle and in order to reduce the number of SAT queries for the conflict object, we iteratively merge sub-rectangles sharing an edge if their flags indicate that they can be assigned to the same color channel (Fig. 3d).

Difficulties and Optimizations So far the outlined algorithm is based on the assumption that there is always a free color channel for each sub-rectangle. However, configurations exist where this is not the case and they occur more frequently with an increasing number of conflict objects and higher depth complexity. Obtaining accurate results with rectangular queries may also create an exceedingly large number of SAT lookups and thus splitting is to be terminated. By looking at the silhouette of the objects' bounding volumes in image space we distinguish the three possible overlap configurations that may occur in these cases and handle them – possibly by sacrificing accurracy – as follows.

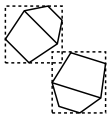
Overlap conflict Both objects intersect the overlap region. Since we omit further splitting we overestimate visibility and share the overlap region across both objects, i.e., pixels in that region contribute to the visibility of both objects.



Overlap no conflict If only one object intersects the overlap region then we split the rectangle of the respective other object. In favor of less SAT queries, we can over-estimate visibility in this case as well and abandon the rectangle split.



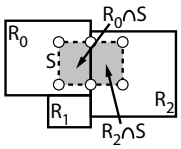
Empty overlap The simplest case is when no object intersects the overlap region. If we detect this case, no rectangle needs to be split.



Creating the Summed Area Table The region set for each object is sent to the GPU via shader constants. Non-conflict objects are rasterized into the query texture into a single color channel, whereas rendering the conflict objects is slightly more involved: In a fragment shader, we determine for each pixel to which rectangle it belongs and choose color accordingly. The summed area table itself is computed on the GPU using the fast generation algorithm by [Hensley J. and Lastra 2005].

3.1.3 Querying the Visibility of Sub-Regions

We can now query the visibility of arbitrary sub-regions of objects, e.g. a rectangular sub-region $S = [x_1, x_2] \times [y_1, y_2]$. We determine the intersections of S and the region set rectangles $R_i = [x_{i,1}, x_{i,2}] \times [y_{i,1}, y_{i,2}]$, with



$$S \cap R_i = [max(x_1, x_{i,1}), min(x_2, x_{i,2})] \times [max(y_1, y_{i,1}), min(y_2, y_{i,2})].$$

All non-empty $S \cap R_i$ are evaluated using the SAT and their accumulated contribution yields the visibility of the sub-region.

3.1.4 Pixel Counting with Mip-Maps

Similar to SATs we can use mip map pyramids (MMPs) to retrieve the average color of larger image regions efficiently and we implemented this approach for comparison. However, lookups with MMPs are less flexible due to their generation scheme. During the construction of the region set, we keep track of the occupancy of texels in the MMP using one quad-tree for each color channel such that each quad-tree node corresponds to one color channel in one texel in the MMP.

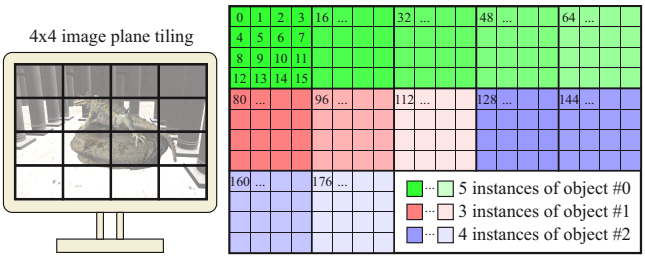


Figure 4: *A 2D histogram (computed from a hierarchical item buffer) with 192 bins due to a 4×4 image plane tiling and 12 instances from 3 different objects. Mip-mapping can be used to query the visibility of instances in individual screen tiles up to the whole screen. Figures in the colored boxes denote the ID of the instance/tile pairs.*

At first sight, MMPs are tempting as few lookups allow us to query large image regions and this proved right in scenes with few objects. However it turned out that the region sets, and with it the CPU overhead for maintaining the quad-tree updates, become very large when the number of query objects increases. In the end, the SAT approach outperformed MMPs for all reasonably complex test scenes and we consequently refrain from a detailed description.

3.2 Hierarchical Item Buffers

Our second approach is an extension to the well-known item buffer algorithm. In its original formulation a unique ID (a color) is assigned to each object and the scene is rendered to a buffer. Then the visibility can be determined by counting the pixels storing the respective IDs. [Scheuermann and Hensley 2007] demonstrate a fast histogram computation on GPUs which can be used for item buffer counting: The item buffer is bound to the input stage of the graphics pipeline and interpreted as a point list. The vertex shader scatters each point by computing a bin index from each pixel and a point primitive is rendered (with additive blending turned on) into the histogram texture.

We adapted their method to the needs in our visibility determination. First, we use a special layout and compute 2D histogram textures such that we can query the visibility of objects (and instances) with a granularity of a predefined screen tiling. To this end, we virtually divide the screen into $2^t \times 2^t$, $t \geq 0$, tiles. Thus the 2D histogram texture (Fig. 4) contains one square region of $2^t \times 2^t$ bins for each object instance. When rendering into the item buffer, the ID can be computed in a pixel shader from the base ID of the object, the instance index and the screen tile where the pixel resides:

$$ID = baseID + instanceIndex \cdot 2^{2t} + x + y \cdot 2^t$$

where $x, y \in [0..2^t - 1]$ denote the horizontal and vertical screen tile location of the pixel. Please note that an even finer granularity is imaginable, e.g. by assigning different IDs to triangle groups or clusters within an instance.

Next the histogram is computed from the item buffer; each ID used in the item buffer refers to a bin in the 2D histogram as shown in Fig. 4. In item buffers pixels containing the same ID are often clumped together. During the histogram generation these pixels are sent to the same histogram bin, thus hindering GPU parallelization

(this problem occurs rarely for image histograms). To alleviate this, we apply a simple reordering trick to the scattering operations and process the item buffer pixels in randomized order using a precomputed index buffer.

Fig. 4 shows an example for a 2D histogram texture with a layout for $t = 2$ and a total of 15 instances from 3 different objects. It can be used to directly lookup the visibility of each object instance within each screen tile. We call this method the *hierarchical item buffer* (HIB), as we can also use the 2D histogram texture together with its mip maps to efficiently query visibility of both whole objects and sub-regions: The HIB allows us, for example, to query the visibility of the first instance in the screen tiles 2, 3, 6, and 7 with a single lookup into mip level 1 (the full resolution texture is mip level 0), or the total visibility of instances in the mip level 2. The mip maps can be automatically generated by the GPU with very little overhead.

An interesting fact is that the granularity has no negative effect on the speed of the histogram generation. If at all, finer granularity creates less GPU stalls and is even faster. The only consideration is the amount of texture memory that is required for storing the histogram when many instances are to be queried.

The HIB method is very easy to implement and does not introduce CPU overhead, since all ID computation and pixel counting is offloaded to the GPU. It is worth noting that it is the only method that can be used to query the individual visibility of instanced geometry fully on the GPU. This is particularly tempting as – depending on the number of draw calls, shader cost and pipeline throughput – a tremendous increase in rendering performance due to instancing has been reported (e.g. see NVIDIA's "skinned instancing" demo).

4 Results and Applications

In this section we present timings and results of our method. We applied granular visibility queries to a shadow volume implementation using the GPU to extrude the objects' silhouettes, and to a displacement mapping technique intensively using geometry shaders.

4.1 Shadow Volume Culling

For testing our method in practical applications, we used it to speed up the shadow volume algorithm [Crow 1977], which is best suited for rendering accurate shadows from point lights, but has also been extended to generate soft-shadows [Assarsson et al. 2003]. Geometry shaders allow the generation of the shadow volumes directly on the GPU which is beneficial for real-time applications, especially with dynamic or deforming geometry. In addition to the geometry shader, the rasterization of the shadow volumes is costly as well, as it consumes a lot of fill-rate. One solution to reduce this cost has been presented in [Lloyd et al. 2004] and a GPU-friendly version is described by [Eisemann and Décoret 2006]. The culling of superfluous shadow volumes can be easily implemented using our methods. In the teaser (left images) we show an example scene where we determined the visibility of each object with respect to the light source in the geometry shader. Shadow volumes are excluded from rendering if an object is fully in shadow. Table 1 shows the rendering performance and compares it to predicated/conditional occlusion queries; Table 2 details the CPU timings for the SAT method.

4.2 Image Space Culling: Displacement Mapping

In our second example we use granular visibility queries to speed up rendering by culling sub-regions of objects. This is possible, as SATs and HIBs do not only contain information about the number of visible pixels per object, but also of their spatial distribution.

	NVIDIA 280GTX				ATI HD4870X2			
objs	nc	SAT	HIB	pred.	nc	SAT	HIB	pred.[1]
100	53	217	194	217	100	149	225	75
200	27	99	121	113	55	102	156	38
300	19	63	87	70	39	69	108	24
400	14	38	48	37	30	43	65	18
500	11	26	31	27	24	29	43	12

Table 1: *Average frames per second for the shadow volume example achieved with the SAT and HIB with instancing for a varying number of randomly placed objects. For comparison, we used Direct3D10 occlusion predicates and no culling (nc). All culling techniques use a 512^2 resolution render target for determining visibility. The rendering resolution was 1600×1050, each object consists of 606 triangles.* [1] *presumably a driver related problem was responsible for the bad performance.*

#obj	overlap test	coloring	split& merge	conflicts	⌀error pixels	⌀error percent
100	0.01	0.05	0.1	10	1,6	2,3
200	0.35	0.17	0.8	50	1,7	2,6
300	0.6	0.33	3,7	116	2,7	4,4
400	1.1	0.5	11	166	6,3	9,8
500	1.6	0.7	23	249	7,4	11,6

Table 2: *Detailed timings in milliseconds of the CPU tasks when using the SAT approach in the shadow volume example. We also give the average relative and absolute pixel error of the queries.*

We use the granular visibility queries for a displacement mapping technique [Wang et al. 2004], specifically its adaptation in the DirectX10 SDK (shown in the teaser), where the geometry shader extrudes triangles and splits the resulting prism into three tetrahedra. Therein gradients are constant and a fragment shader computes the intersection of a view ray and the height field. We modified the geometry shader such that it determines the bounding rectangle of each prism, and next, it determines if any pixel belonging to the object is visible within this rectangle. If no visible pixels are found, the geometry shader terminates without creating geometry. Please note that we conservatively extrude the triangle mesh in the first render pass (to create the SAT or HIB) by moving its vertices in normal direction such that it encloses the displaced surface. When using SATs the visibility determination computes the intersection with every query in the region set as described in Section 3.1.3. For the HIBs implementation we test for visibility in all overlapped screen tiles. Table 3 shows the measured rendering performances.

culling method	full vis.	partial vis.	occluded
none	71 / 22	90 / 42	90 / 45
predicates, 594×12 triangles	25 / 23	47 / 50	149 / 62
predicates, 891×8 triangles	20 / 22	30 / 32	110 / 42
item buffer, 8×8 tiles	61 / 22	80 / 41	186 / 158
item buffer, 16×16 tiles	61 / 22	105 / 63	186 / 158
item buffer, 32×32 tiles	61 / 22	123 / 82	186 / 158
item buffer, 90×90 tiles	66 / 24	131 / 87	186 / 158
summed area table	57 / 19	89 / 50	112 / 67

Table 3: *Frames per second measured on a NVIDIA 280GTX/ATI HD3870 for the lizard displacement mapping for full and partial visibility (see teaser). Again, due to driver problems we could not use a ATI HD4870X2 for comparison. The results indicate that HIBs do not suffer from many IDs. Timings for full occlusion are given to estimate the overhead of the methods. The static scene part consists of 134414 triangles, the lizard has 7132 triangles.*

method	granularity	CPU load	GPU load	instancing	accuracy
HOQ	draw call	very low	render proxy [1]	no	exact
Predicated OQ	draw call	very low	render proxy [1]	no	no feedback
Coloring + SAT	arbitrary sub-regions	high	additional render target	no[2]	approximate
Hierarchical Item Buffer	fixed (screen) tiling	very low	additional render target	yes	exact

Table 4: *This table compares the properties and possibilities of natively supported visibility queries and our methods.* [1] *the scene has to be rendered twice to resolve mutual occlusion.* [2] *instancing is possible, if the instance transformations are evaluated on the CPU.*

5 Discussion

Differentiation Our method is meant to query the visibility of objects efficiently after rasterizing many objects, as it is common for all item buffer methods. That is, stop-and-wait algorithms, e.g. [Bittner et al. 2004], are orthogonal to our approaches. Similar to GVQs, a software implementation of the hierarchical z-buffer method allows the culling of portions of objects. However, the HZB is bound to the fixed construction scheme as mip maps are, and thus requires many texture lookups to test for visibility of arbitrary regions, and counting visible pixels is impractical.

Accuracy As expected the query results when using SATs show deviations which are due to the SAT resolution, but mainly due to approximative region sets (Sect. 3.1.2). However, we believe that this negligible for many applications, such as those using aggressive culling strategies. The conservative region sets return more pixels than actually visible and the deviation and the CPU load grows with the number of conflicts (see Table 2). Thus the SAT approach can be recommended for a moderate number of query objects only. Please note that multiple render targets, and thus more than 4 color channels, can be used to reduce the number of conflict objects.

Non-Binary Visibility Typically the visibility function for a pixel is of a binary nature – it is visible or not. Interestingly the SAT method does away with this restriction: Instead of writing purely colored pixels to query textures, we can also output pixels with any brightness. We believe that this has various tempting applications, e.g. to estimate blocking from a transparent occluder, or to accumulate the tolerable visibility threshold as used with perceptual rendering methods [Drettakis et al. 2007]. A similar result can be obtained with the item buffer approach by storing a (scalar) value in addition to the ID per pixel. During histogram computation, we sum up the values instead of incrementing the bin by one.

Querying the Visibility on the GPU By using predicates and standard occlusion queries (with a read back to the application) we can prevent geometry from being sent to the graphics pipeline at all, and this is obviously the fastest method for culling large chunks of geometry. The strengths of GVQs lie in the culling of individual primitives and the controlling of shader level of detail in (partly) occluded regions.

Geometry shaders are often used to extrude geometry and subdivide triangles. The rendering performance does not only depend on the shader instructions, but also on the amount of geometry that is output. GVQs can be used to determine local visibility information in order to omit primitives and shader output as demonstrated in our examples.

If early-z culling becomes deactivated, e.g. due to shaders that output depth, GVQs can be used to avoid costly pixel shader executions: Geometry can be omitted in geometry shaders, or "culled" in vertex shaders by transforming it outside the view frustum to prevent rasterization. As mentioned before, it is imaginable to estimate the number of occluded pixels per primitive to obtain a fractional visibility and by this control the accuracy or detail in pixel shader computations. Table 4 summarizes properties of the different query mechanisms.

6 Conclusions and Future Work

We presented two methods for granular visibility queries which allow us to determine the visibility of objects and sub-regions of objects in shader programs on the GPU. We demonstrated how costly rendering techniques, in particular those relying on geometry shaders, gain significant performance increases. We also introduced a hierarchical item buffer which enables efficient visibility queries for instanced objects and sub-regions simultaneously.

Granular visibility queries and non-binary visibility functions are particularly interesting for more involved rendering methods, such as real-time perceptual rendering and level of detail control. We believe that our method also has potential use in other GPU rendering techniques which rely on visibility determination of some kind.

References

ASSARSSON, U., DOUGHERTY, M., MOUNIER, M., AND AKENINE-MÖLLER, T. 2003. An Optimized Soft Shadow Volume Algorithm with Real-Time Performance. In *HWWS '03: Proceedings of the conference on Graphics hardware*, 33–40.

BITTNER, J., AND WONKA, P. 2003. Visibility in Computer Graphics. *Environment and Planning B: Planning and Design 30*, 5, 729–756.

BITTNER, J., WIMMER, M., PIRINGER, H., AND PURGATHOFER, W. 2004. Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. *Computer Graphics Forum 23*, 3, 615–624.

COHEN-OR, D., CHRYSANTHOU, Y. L., SILVA, C. T., AND DURAND, F. 2003. A Survey of Visibility for Walkthrough Applications. *IEEE Transactions on Visualization and Computer Graphics 09*, 3, 412–431.

CROW, F. C. 1977. Shadow Algorithms for Computer Graphics. *SIGGRAPH Comput. Graph. 11*, 2, 242–248.

DÉCORET, X. 2005. N-Buffers for Efficient Depth Map Query. *Computer Graphics Forum 24*, 3.

DRETTAKIS, G., BONNEEL, N., DACHSBACHER, C., LEFEBVRE, S., SCHWARZ, M., AND VIAUD-DELMON, I. 2007. An Interactive Perceptual Rendering Pipeline using Contrast and Spatial Masking. In *Rendering Techniques 2007 (Proceedings of the Eurographics Symposium on Rendering)*.

EISEMANN, E., AND DÉCORET, X. 2006. Fast Scene Voxelization and Applications. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM SIGGRAPH, 71–78.

FUNKHOUSER, T., TELLER, S., SQUIN, C., AND KHORRAMABADI, D. 1996. The UC Berkeley System for Interactive Vi-

sualization of Large Architectural Models. In *Presence: Journal of Virtual Reality and Teleoperators*.

GREENE, N., KASS, M., AND MILLER, G. 1993. Hierarchical Z-Buffer Visibility. In *SIGGRAPH '93*, 231–238.

GREENE, N. 2001. Occlusion Culling with Optimized Hierarchical Buffering. In *ACM SIGGRAPH Course Notes #30*.

GUTHE, M., BALÁZS, A., AND KLEIN, R. 2006. Near Optimal Hierarchical Culling: Performance Driven Use of Hardware Occlusion Queries. In *Eurographics Symposium on Rendering 2006*, T. Akenine-Möller and W. Heidrich, Eds.

HENSLEY J., SCHEUERMANN T., S. M., AND LASTRA, A. 2005. Interactive Summed-Area Table Generation for Glossy Environmental Reflections. In *ACM SIGGRAPH sketches*, ACM Press.

KLOSOWSKI, J. T., AND SILVA, C. T. 2000. The Prioritized-Layered Projection Algorithm for Visible Set Estimation. *IEEE Transactions on Visualization and Computer Graphics 6*, 2, 108–123.

LEYVAND, T., SORKINE, O., AND COHEN-OR, D. 2003. Ray Space Factorization for From-Region Visibility. *ACM Trans. Graph. 22*, 3, 595–604.

LLOYD, B., WENDT, J., GOVINDARAJU, N., AND MANOCHA, D. 2004. CC Shadow Volumes. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Sketches*, 146.

LUEBKE, D., AND GEORGES, C. 1995. Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, 105–ff.

MATTAUSCH, O., BITTNER, J., AND WIMMER, M. 2008. CHC++: Coherent Hierarchical Culling Revisited. *Computer Graphics Forum (Proceedings Eurographics 2008) 27*, 2 (Apr.), 221–230.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, August.

OPENGL EXTENSION REGISTRY, 2008. NV_conditional_render. http://www.opengl.org/registry/.

SCHEUERMANN, T., AND HENSLEY, J. 2007. Efficient histogram generation using scattering on gpus. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, 33–37.

STANEKER, D., BARTZ, D., AND MEISSNER, M. 2003. Improving Occlusion Query Efficiency with Occupancy Maps. In *PVG '03: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, 15–ff.

WANG, X., TONG, X., LIN, S., HU, S.-M., GUO, B., AND SHUM, H.-Y. 2004. Generalized Displacement Maps. In *Rendering Techniques*, 227–234.

YOON, S.-E., SALOMON, B., AND MANOCHA, D. 2003. Interactive View-Dependent Rendering with Conservative Occlusion Culling in Complex Environments. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, 22.

ZHANG, H., MANOCHA, D., HUDSON, T., AND HOFF, III, K. E. 1997. Visibility Culling using Hierarchical Occlusion Maps. In *SIGGRAPH '97*, 77–88.