Vision, Modeling, and Visualization (2016)
M. Hullin, M. Stamminger, and T. Weinkauf (Eds.)

Topological Triangle Sorting for Predefined Camera Paths

Christoph Weber[†1] and Marc Stamminger[‡1]

[1]Friedrich-Alexander Universität Erlangen-Nürnberg, Germany

Abstract

We present a preprocessing pipeline for triangle meshes that topologically sorts all triangles for a given camera and scene animation in front-to-back or back-to-front order. This allows us to efficiently render a given animation without depth buffer, and to include transparency. We also remove non-contributing triangles, thus improving render time, especially when applying anti-aliasing. To this end we first record the visible triangles of a sequence of frames. For every frame we create a directed graph storing occlusion information. After a topological sort of this graph, all triangles are sorted properly. The contribution of this paper is the reduction of redundancy by merging the graphs of all frames. The result of our pipeline is a single sorted index buffer, over which we slide a window that yields sorted index buffers for each single frame. Circular dependencies are broken by placing duplicates of the affected triangles in the index buffer. Our sliding window then displays only frame specific triangles in their proper order. We conclude by demonstrating the benefits of removing invisible triangles and disabling the hardware visibility test.

1. Introduction

The premise of this paper is the acknowledgment that depth testing is costly, or rather that the synchronization of fragments belonging to the same pixel is and always will be a bottleneck in rasterization pipelines. We concede that contemporary graphics cards render this problem negligible. However when considering low-cost, low-power mobile GPUs such contemplations become worthwhile yet again. Hence we coin the notion of *automotive visualization*, by which we describe the rendering of geometry, namely cars, on mobile platforms such as are integrated in dashboards of premium-segmented vehicles. Car manufacturers strive to present interactive experiences to increase the perceived value of cars or to address young audiences familiar with interactive media. The scenario ranges from a personalized rendering of an owners car, to guided tours or interactive manuals that show the correct button for the filler cap via a rendered close-up of the same. Such a scenario offers specific challenges and well defined restriction that can be exploited:

· Our rendering applies a fixed camera and animation path as well as a fixed frame frequency, in short: every resulting image must be perfectly deterministic.

· The quality of the rendering must satisfy strict demands, aliasing is unacceptable.

· The available resources are limited and shared with other, more critical applications.

An immediate solution to this problem are pre-rendered videos, however integrating movies into the rendering context requires the storage, decoding and streaming of high-resolution content within the rendering context. While such an option is clearly possible, we want to explore the possibility of exploiting the knowledge associated with predefined camera paths and optimize the visibility determination of triangles. In an initial experiment, we implemented a *Geometry (G)-buffer* video, which allows for configurable rendering of the final scene at peak performance. Such an approach, albeit offering a very lean and fast rendering pipeline is prohibitively memory expensive, assuming a minimal precision of 5 bytes per pixel for material, normal and depth (1-2-2). The key problem of such a G-buffer video approach is compression, which should be lossless, as variations can cause noticeable shading errors, thus offering very little compression ratios. However, the core idea of video compression, namely the reduction of redundant and recurring content, should carry over to rendering of determinate sequences. Hence, we propose a rendering pipeline that combines certain ideas of pre-rendered content with the effectiveness of contemporary rendering pipelines. Our final contribution is a sliding window over a sorted index buffer, thus eliminating the need for costly visibility computation or online sorting of geometry.

2. Previous Work

The removal of hidden surfaces is a fundamental problem for rendering pipelines that apply rasterization by projecting geometry

† christoph.weber@cs.fau.de
‡ marc.stamminger@informatik.uni-erlangen.de

DOI: 10.2312/vmv.20161354

on an image plane. Early solutions relied on sorting the primitives [SSS73], however sorting is a global problem and requires complex solutions to resolve overlaps and circular dependencies. Using a depth- or *z-buffer*, introduced by Straßer [Str74] and Catmull [Cat74] to resolve visibility on the pixel level, has been proven to be the most effective solution with regard to modern rendering architectures, as it is robust in the face of circular dependencies. Several improvements have arisen from the development of modern graphics hardware such as *early-z* and hierarchical-z, the basic principle remains unchanged.

While depth-buffer approaches will always provide the correct solution, they largely ignore the spatial coherence inherent in the three dimensional scene structure. With the introduction of binary space partitioning [FKN80], triangles can be hierarchically clustered. The resulting tree can then be used to quickly discard large groups of primitives.

The dependency relation between triangles itself can be formulated using graphs, as is shown in this paper. The original considerations by Williams [Wil92] describe the problem of occlusion and inter-triangles relations. Other proposals use various kinds of space partitioning techniques, such as feudal priority lists [CW96] or clustering approaches such as voronoi diagrams [FS06]. Compared to these general purpose implementation, we consider the merging of graphs of consecutive, temporally coherent viewing angles as our area of interest.

The complements to primitive ordering are fragment sorting algorithms. The simplest idea introduced by Everitt [Eve01] uses the z-buffer to identify the closest layer of fragments. By rendering the same configuration multiple times and *peeling* off the closest layer, we acquire sorted fragment lists for every pixel. Various optimizations have accelerated the technique, starting with the immediate progression of peeling multiple layers per pass [LWX06] and peaking with GPU friendly approaches that acquire all layers in a single pass and sort them in a second, using per pixel linked lists [BKSSk11]. We adopt the peeling idea in our preprocessing step to acquire the dependency of visible triangles. However we have no interest, in acquiring a complete list.

Visibility computation is a many-faceted problem and is not restricted on determining the triangle order. The multitude of visibility related algorithms address the problem of removing entire clusters of virtual scenes. A major concern is the data transfer from disk to main and graphics memory. Sajadi et al [SHDG*09] propose a caching structure, that clusters the memory layout instead of the actual scene data Their method allows the interactive exploration of very large environmental scenes. Memory transfer is especially problematic if the rendered geometry has to be loaded via network. The approach of Limper et. al [LJB*13] allows the progressive upload of geometry to provide an immediate visualization during the loading of a web page. The same problem is addressed by Behr et. al [BJFS12]. They accelerate the visualization of web content by separating the upload of structured scene information and unstructured vertex data. Because the latter is dependent on the viewing angle, it is not necessary in its entirety and can be streamed as required. s A pipeline with very similar motivation to ours has been proposed by Chen et al. [CSN*12]. They also propose to sort triangles but aim to provide viewport independence. As a consequence, they not only duplicate triangle faces to account for front and back faces, but they also decide during rendering, which triangles to draw. Our method differs in so far, that we restrict camera movement to a predefined path and use this knowledge to make rendering as simplistic as possible. Instead of selecting triangles during rendering, we provide a "sliding window" over the index buffer, that shows a set of sorted triangles applicable for each frame. Their algorithm also only considers statical models and breaks upon entering the bounding volume of geometry. Because we sort triangles solely based on their observed visibility, we can easily implement deformations and forgo elaborate culling pipelines. Another concept by Ernst et al. [EFG04] also removes triangles based on observed visibility. They assume that certain areas are never observed, and therefore decimate the omitted triangles using various sampling methods to gauge visibility. However, they argue that their method requires user interaction and proper configuration to ensure that actually visible triangles are not removed. Similar in concept is the algorithm of Eikel et al. [EJF10]. By sampling large scenes in a preprocessing step, they can determine the number of visible triangles, thus culling the majority of a scene. Thusly reduced, the culled scene can be rendered on low-performance rendering hardware, e.g. for showcasing purposes. Another comparable approach that also creates index buffers based on viewing angles, is the clustering approach from Han and Sander [HS16]. They create distinct ordered index buffers for various viewing directions and switch to the appropriate one during rendering. Their technique is very similar in concept to ours but does not address the problem of reusability of triangles across viewpoints. Instead they cluster the viewing angles that are valid for a specific index buffer and store the triangles redundantly, just as we do if change in viewing angle inverts the triangle ordering.

Dynamic scenes introduce an additional uncertainty to the visibility problem, as occlusion relations change independently from the viewing transformation. As a result, the caching of visibility must be done on a frame-to-frame basis and should therefore be fast. Garanzha [Gar09] devises a fast bounding volume hierarchy builder, by restricting range of possible deformations and assuming that triangle clusters remain connected.

Most depth sorting algorithms apply a global ordering of triangles or provide elaborate space partition structures to accelerate the visibility decision during rendering. An absolute restriction of transformation freedom is yet to be exploited.

3. Rendering of Triangle Dependency Graph Sequences

In standard rasterization, the closest fragment is identified via depth testing. By applying early-z culling, new fragments, farther from the image plane, are ignored and not shaded. Rendering is therefore most efficient if the triangles are ordered, either because depth testing can be disabled (when rendering back to front using painters algorithm, which is appropriate if the fragment evaluation is simple) or because early-z can cull most fragments (when rendering front to back, which is appropriate if the fragment evaluation is expensive). Certain render modes, such as alpha blending or alpha based anti-aliasing, require disabling standard depth testing and rely on ordered geometry.

The occlusion relationship of triangles can be stored in a graph

or rather a forest of directed trees. Since most triangles do not overlap, they are independent of each other. Also, most triangles are invisible, either because of sampling properties, or because they are occluded.

Triangle dependencies are view specific. Each frame in an animation sequence has its own dependency graph. The most trivial solution to our fixed camera pipeline is to store and traverse individual graphs. Yet, we can expect that during an animation the dependency graphs of successive frames would be similar. It is very likely that the union of two graphs can be used to reconstruct/render the two belonging frames. A union of two graphs introduces the possibility of circular dependencies, but because most triangles are independent of each other, we should be able to merge the majority of them and retain the conflicting parts. Instead of two largely redundant graphs, we store three: One large graph containing the similar elements and two minor ones, containing view-specific dependencies.

Merging graphs for an entire camera sequence, with regard to traversal order and occlusion dependencies, is the capstone of our contribution and explained in Section 4.2. To simplify, we initially ignore triangle dependencies entirely. Thus we explain in detail how to sample (Figure 1), store (Figure 2), and render (Figures 8, 9) visible triangles. Occlusion dependencies are incorporated over the course of the explanations (Figure 7). Once the basic concepts have been introduced, we elucidate how to merge graphs and remove circles (Figure 12).

3.1. Recording Triangle Graphs

Recording the *visible* triangles involves two steps (see Figure 1):

We start by accumulating the set of triangles that are visible at least once during the entire camera/animation path. Invisible triangles are never entered into the individual graphs, thus speeding up the later merging process by relaxing constraints and reducing the chance for circular dependencies.

Using *depth peeling*, we strip layers from front to back, and we also accumulate the opacity/alpha for every pixel. Once the alpha-value of a pixel (p) is saturated, we ignore every triangle that lies behind p. Note that the opacity of transparent materials can be set to 0, to capture all triangles. The actual opacity value can be adapted during rendering. Having acquired the visibility information, we create the actual graph. The graph nodes are the triangle IDs, with every node maintaining a list of occluding triangles, as well as its occluders.
During depth peeling we add the visible triangles to the graph. The ordering of triangles is implicit in the sequence of depth layers. Note that this approach not only ignores occluded triangles, as per design, but also those that are never sampled during rasterization. The resulting surfaces are only true for the specific camera position from which they are recorded. Even a slight shift will yield a perforated surface. This proves problematic when applying multi-sample anti-aliasing, but is easily solved by repeating the process for every sample position and adding the additional triangles to the graph. This problem of sampling a "water tight" surface presents a fundamental obstacle when merging graphs of different views. We solve this by creating a set of *ignored* triangles, namely those that are never covered by a fragment.

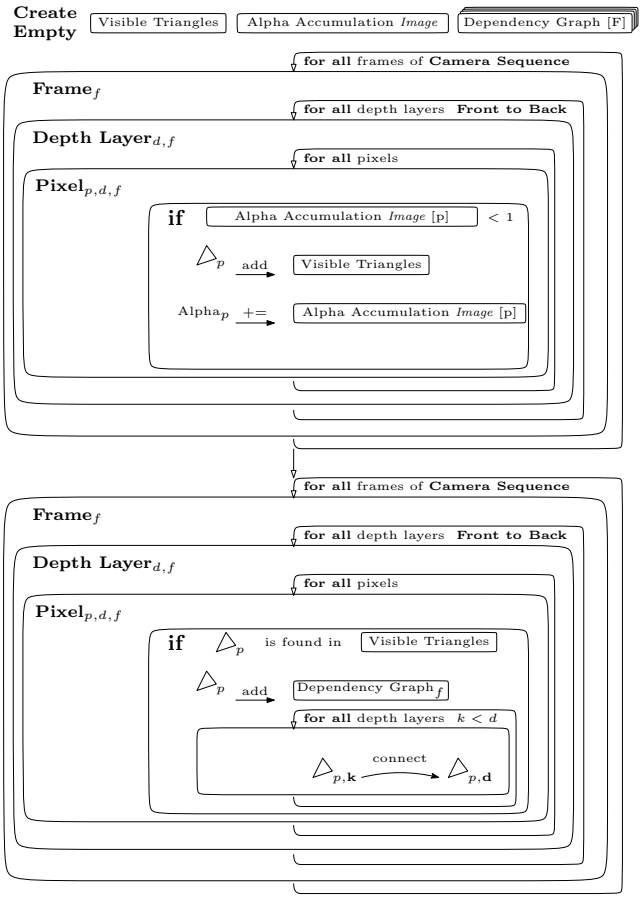


Figure 1: Dependency Graphs are created with a two step approach. First, we acquire triangle visibility. Only triangles that are visible enter the graph. However, triangles that are *temporarily* occluded must also be considered, as our graph traversal of the upper triangular matrix relates graphs to one another. Then, we capture all visible triangles and create a connected dependency graph.

If we were to create a triangle graph for a single frame, our pipeline would be completed. The rendering of such a reduced model is, expectably, very fast (see Table 1). Next, we introduce the idea of merging triangle graphs from many frames.

3.2. Merging Triangle Graphs across Frames

Our algorithm creates one graph per frame. The final camera sequence is easily visualized as a list of graphs (Figure 2, left). However, such a graph sequence is highly redundant, and in particular graphs of similar images from adjacent camera positions are nearly congruent. Here we introduce our concept of arranging the dependency graphs in an *upper triangular matrix* (see Figure 2, right). Each node (i, j) of that matrix contains a graph of triangles visible from frame i up to frame j. We start our reduction algorithm by placing the original, redundant graphs on the diagonal. Our algorithm then removes redundancies by "shifting" triangles (Figure 4)

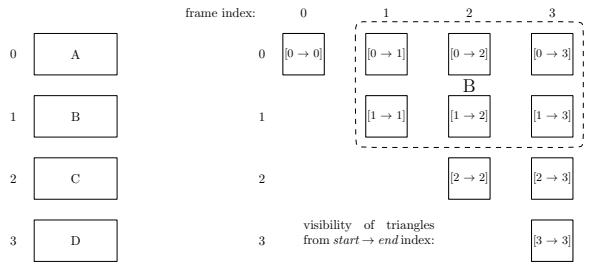
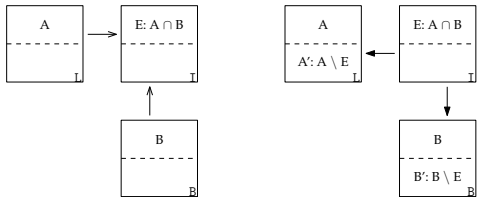


Figure 2: The original graphs lie on the diagonal of an upper triangular matrix. By merging graphs, triangles are "shifted" towards the top right. By shifting, we indicate that the triangle is visible over more frames, as indicated by the *start* and *end* index. The original graphs are stretched over the triangular matrix (here graph "B"). The sliding window for frame+1 /graph "B" contains all graphs with *start* index $\leq 1 \leq$ *end* index.

from the diagonal towards the upper right corner of the matrix. The more remote a triangle is from the diagonal, the longer it is visible during the sequence. Triangles in the top-right are consequently always visible. The reconstruction of the original frame requires the traversal of every graph to the right and overhead of the diagonal element (Figure 2, dashed box on the right).

Removing triangles from the diagonal requires merging the pairs of adjacent graphs (see Figure 3, Intersection). We refer to these dyads as sources (`S`) and discriminate between the left (`L`) and the bottom or below (`B`) graph. The result of this merging operation of two graphs is termed *intersection* (`I`). The merging is explained in detail in the next section. After merging, the duplicate triangles are accepted in the intersection graph and must be removed from the diagonal (see Figure 3, Subtraction). Without triangle dependencies, our merging yields intersections as true. However, to merge graphs with regard to occlusion and traversal order, we relax and elaborate our notion of intersection. To this effect, we will later merge the graphs into a union and will truncate until satisfied. The truncated union then ultimately yields our "intersection".



I. Intersection II. Substraction

Figure 3: Two-step merging of graphs: I. finding of the intersection (`I`) of two graphs left and below (`L`, `B`); II. Removing the intersection graph from the original graphs.

The merging operation between two frames is applied repeatedly, for every diagonal, until all graphs are processed (see Figure 4). Merging pairs of graphs is "embarrassingly parallel", however, note that almost every source graph adjoins two intersections. Removing both intersections from the source graphs must be serialized.

If our rendering pipeline were to ignore triangle dependency, our algorithm would be finished. We can effectively remove any invisible triangle of any given frame sequence, thus speeding up rendering compared to the original scene. Next, we show the particularities of occlusion dependencies (Figure 7) and how rendering the upper triangular matrix controls the merging operation (Figure 12).

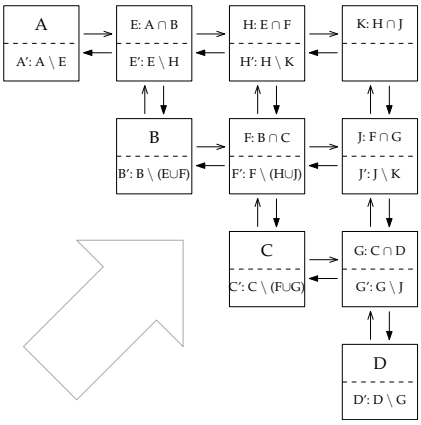


Figure 4: Graph merging for a 4-frame sequence: Elements of each diagonal can be intersected in parallel. Removing the intersection elements from the source graphs can be done in parallel as well, while dual dependencies (e.g. B\E∪F) must be processed sequentially. Merging is simplest if triangles have no order relation, in which case the result is in an actual intersection. By implementing triangle dependencies and graph-relations the merging becomes more complex.

4. Triangle Dependencies

The order of triangles can be described using graphs. If triangles share no dependencies, that is they not occluding each other, such a graph is disconnected. A disconnected graph can consist of connected sub-graphs and disconnected nodes that share no connection. Circular dependencies are illegal, and the triangles are therefore preprocessed and self-intersections must be removed. Experience shows that removing self-intersections is integral to the method as virtually no mesh is without flaw. However, we will not delve into this particular problem any further as it is well understood, although hard to implement.

4.1. Ignored Triangles - Visibility Sampling

Before explicating on the merging of dependencies, we refine our recording of visible triangles. Initially, we strove to collect every triangle visible from the camera, either by computing an analytical solution involving partial coverage, or by sampling the mesh. Both approaches prove ineffective, the former due to numerical limitations, the latter from its inherent bias of insufficient samples. Thus we cannot create a water-tight surface of closest geometry, which severely impedes the effectiveness of our proposed merging algorithm (see Figure 5).
We extend our algorithm and introduce an *ignore list*, thus turning the described handicap into a feature. Because knowledge of the entire sampling pattern is evident, ignoring triangles that miss the raster actually reduces the rendering load. A comparison between

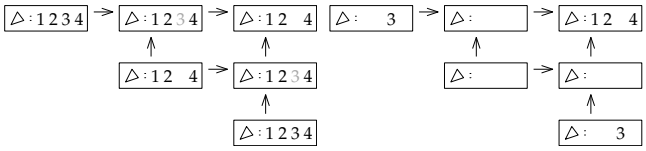


Figure 5: Missing triangles prevent the removal of redundancies. Here, triangle 3 is missing in the second graph on the diagonal.

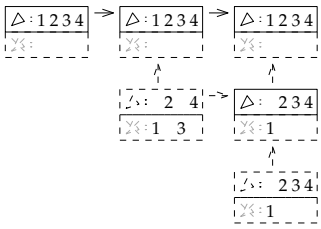


Figure 6: Merged/intersected elements are removed from the *ignore lists* ($\triangle:3$). The merging/intersection operation is applied for *ignored* triangles as well ($\triangle:1$).

Figures 5 and 6 shows that all four triangles are merged in the top right corner. A subsequent reduction step in Figure 6 removes every triangle from the lower diagonals.

4.2. Merging Triangle Dependencies

Up to this point, inter-triangle dependencies have been largely ignored, namely total and partial occlusions. Yet, by merging two graphs we also combine inter-triangle relations, thus adding complexity to the merging problem. Two immediate problems when combining are the creation of circular dependencies and abridging transitive dependencies (see Figure 7: `III`, `IV`). While circles are an obvious problem, shortcuts are problematic due to the traversal requirements: the final rendering requires a breadth-first traversal, ambiguous cases as in graph `III` would allow triangle 3 to be rendered before triangle 1. As such, the intersection operation must implement restrictions that prohibit merging if dependencies of either source graph are violated. Yet, any relation that does not transgress the ordering of the source graphs is allowed.

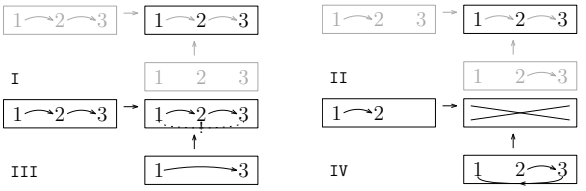


Figure 7: Merging of dependencies. Shortcuts must be avoided, because of breadth-first graph traversal. Circular dependencies are illegal.

Additional restrictions occur when rendering multiple graphs, where the ordering of `L`, `I` and `B` becomes relevant (Figure 12).

5. Rendering & Multi-Graph-Merging Strategies

The objective of our preprocessing stage is to accelerate the rendering without increasing the complexity of the rendering pipeline. That is, a just in-time traversal of graphs is impossible. Instead we create a single *index buffer* by serializing all graphs of our triangular matrix and concatenating the individual results line after line. The traversal of directed graphs follows a *breadth-first* directive (Figure 8, serialization of graph$_{i,j}$). Note that the ordering of triangles affects the traversal of the graph as well.

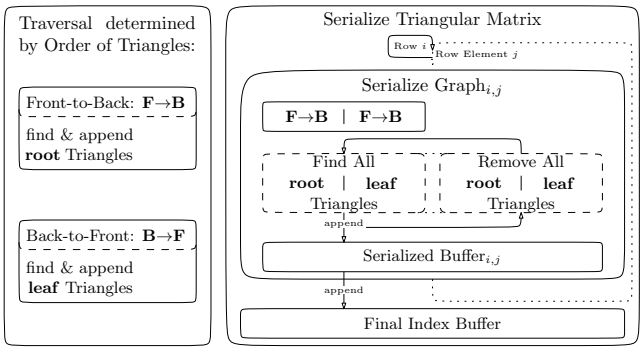


Figure 8: Individual graphs are serialized via breadth traversal. Serializing triangular matrices complies with a line-wise concatenation of graphs. The origin of traversal, whether the root or the leaves, depends on the rendering order of triangles: front to back or vice versa.

Again, because the rendering is to be fast, the triangles *should* be merged in a single buffer. Note that while the upper triangular matrix is stored sequentially and in close succession (line-wise storage), the actual reconstruction requires a scattered access of several (coherent) buffers containing subgraphs (see Figure 9). The

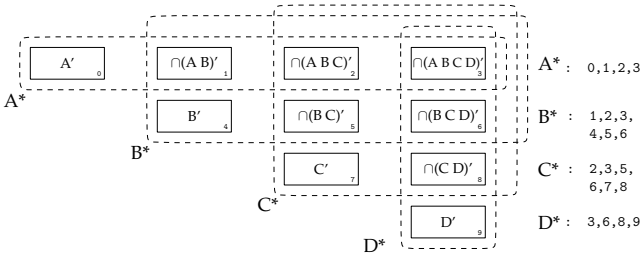
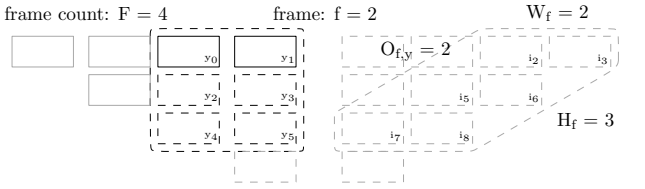


Figure 9: The original graphs A-D are distributed across the upper triangular matrix. The reconstruction of diagonal elements requires the traversal of all elements above and to the right of the original graph.

final rendering thus involves drawing several buffers. Currently, OpenGL 4.3 and beyond provide so-called *multi-indirect* drawing capabilities, which can combine multiple draw calls into one, thus eliminating any additional CPU overhead caused by rendering multiple buffers. The draw-call requires an additional GPU array beyond the vertex and index buffer, listing the offsets (see Figure 10) and triangle counts of the graphs in our triangular matrix. We name such buffer a *command buffer*. Note that every frame requires a dedicated command buffer. The sliding window is thus implemented by submitting a distinct *multi-indirect* buffer per frame, thereby trading memory for performance gains.



window width:	$W_f = F - f$	row:	$r = y/W_f + 1$
window height:	$H_f = f+1$	offset:	$O_{f,y} = f \times r - \frac{r \times r-1}{2}$
graph:	$y = 0 \rightarrow W_f \times H_f$	final index:	$i = O_{f,y}+y$

Figure 10: The serialization of our sliding windows requires an offset computation. The offset of every each row is determined via the upper left triangular matrix.

5.1. Merging Strategies

Our explanations so far have cycled around the idea of merging two source graphs (L, B) into an intersection (I) and then expanded the idea hierarchically using our upper triangular matrix (Figure 4). The order relation between the graphs L,I & B themselves, which is dictated by the ultimate traversal and serialization for rendering, has yet to be addressed. To combine three graphs there are six possible combinations: LIB, BIL, ILB, IBL, LBI, BLI.

The merging strategies applied to the graphs and by extension the matrix, determine the traversal order within the sliding window. Figure 11 illustrates two merging strategies. Both imply different visibility persistence and traversal orders. The LIB strategy demands that L must be drawn before I, before B. This, transfered to the sliding window, requires line-wise window traversal. An ILB strategy draws the corner element first and commands a diagonal window traversal. The selection of strategies solely depends on the scene, for example:

· LIB works best, if triangles that leave the focus are never seen again.

· ILB is suited for graphs with similar depth relation, placing the closest in the upper right.

· LBI will render unique dependencies first. The effectiveness also depends on the traversal order, whether front-to-back or vice versa. Selecting the best practice for any given scene is basically trial and error, but is easily automated.

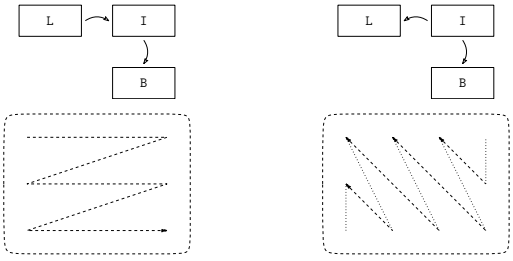


Figure 11: Two different inter-graph dependencies (LIB left, ILB right) that determine merging strategies and the resulting serialization order of distributed Graphs. An inversion of dependencies requires a reversed serialization sequence.

Our graph merging strategy is summarized as follows: shifting triangles (Figure 4) is only possible if the original *intra-graph* dependencies are not violated by the new *inter-graph* dependency.

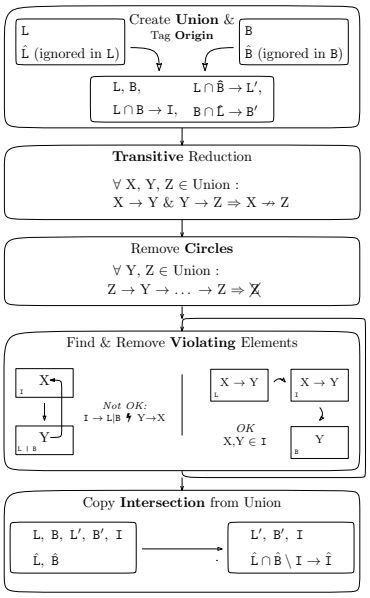


Figure 12: Merging source graphs via a reduction of **Union**. The union contains all triangles and dependencies. Triangles are tagged with their **Origin**, the origin is used to identify violations. Shortcuts are removed via **Transitive** reduction. **Circles** introduced through union are broken. **Violating** elements are identified and removed until none remaining. Violation depends on the chosen merging strategy, here ILB & LIB. The final **Intersection** consists of all remaining elements. The elements still ignored in both, are carried over.

The fundamental feature of our algorithm is the identification and removal of violating elements. A violation occurs, if the order of the matrix traversal contradicts the order of triangles. This will happen, if the order of triangles between graphs is inverted, for example by rotating the camera. Figure 12 shows a sketch of our merging algorithm. To create an intersection between source graphs, we create the union and remove undesirable elements until the "intersection" remains. The fundamental decision rule as to whether a triangle can remain in the union is as follows:

- If the intersection (I) precedes a source graph (S), the triangle (t_I) may not have any successors that are not in the intersection or not ignored ($\hat{S}$).
- If the intersection (I) is rendered after a source graph (S), the triangle (t_I) may not have any predecessor that are not in the intersection or not ignored ($\hat{S}$).
- If a triangle (t_I) possesses illegal connections, it is removed and put back in its source graph.

The algorithm is repeated for all intersection triangles of the union until no violations remain. Because each cleansing operation may alter the entire union graph, the algorithm must be restarted after each cleansing occurs; optimizations are ignored for ease of explanation. Note the two steps prior to the removal loop in Figure 12, in which we cull the aforementioned shortcuts and circles.

5.2. Modeling & Rendering Considerations

Our rendering pipeline requires a singular set of triangles and one command buffer per frame. This specification requires the material/model ID to be stored on a per-triangle basis. The material itself can be adjusted during rendering, which also applies for lighting. Normals are stored in object space, allowing utmost flexibility of transformation, while an octahedral normal [MSS*10] decreases the memory footprint. Transparency can be achieved via over-blending or alpha saturate operations, because the triangles are always sorted, separate render passes are not necessary. The scene can be animated via rigid transformations, either in full, by moving the camera, or partially, for instance, by opening a car door. Because our algorithm captures triangle visibility on a per-frame basis, the actual position of triangles can be ignored, as long as recording and rendering transformations are the same. Because every transformation is known beforehand, everything can be stored on the video memory, superseding any synchronization between GPU and CPU. Should start-up time be critical, the relevant information, namely the final index buffer (see Figure 8), can be streamed just-in time. Our pipeline is especially effective when applying multi sampling as we waste no fragments on invisible triangles.

6. Results

The experiments to be presented were executed on an NVIDIA® GeForce GTX970 GPU and an NVIDIA® GeForce GTX560 GPU. The scene is a model of a car, featuring complex geometry such as meshes and windows, as well as self intersecting geometry that was resolved before recording. The driver door was rigged for an open and close animation.

6.1. Render Timings of Sorted Triangles

The first column in Table 1 shows the timings for rendering a single frame, regularly and after applying our method without occluded triangles. We illustrate the results of various visibility functions applied to our sorted triangles, namely alpha blending (B→F), alpha saturation and depth-test (F→B). We also show the timings of 4× and 16× multi-sampling. We show that renderings are significantly faster due to a reduced triangle count. Transparency can be achieved using saturate or alpha blending operations (Figure 13). We also see that disabling depth testing achieves an additional speedup due no depth writes on the GTX970. Contrary to that we see a deterioration when using the GTX560.

6.2. Triangle Merging Effectiveness

Our rendering pipeline trades memory for performance by providing a pre-sorted index buffer. The graph in Figure 14 exemplifies the size of the individual graphs in our upper triangular matrix. Note that the top-right element cannot be displayed as it contains 170k triangles. Multi-viewpoint renderings introduce redundancies due to irreconcilable order relations between frames. The efficacy of our algorithm is directly tied to the camera placement, as opposite viewing angles of the same geometry will provide little grounds for merging, especially when considering non-convex geometry and translucency effects, such as car interiors. Purely translational camera movements allow for an effective merging, with all triangles grouped in the top right of the triangular matrix.

Table 1: Comparison between the rendering of all unordered triangles (default) vs. the rendering of minimal sets (source) vs. the sliding windows of our upper triangular matrix (merged). Values in braces indicate renderings using Multi-Sampling. Depth and saturate define the visibility test. Rendering times in ms. Resolution: 1024 × 1024. The top half shows the timings using an NVIDIA® GeForce GTX970 GPU, the bottom half shows the timings of an NVIDIA® GeForce GTX560 GPU.

frame	Default Pipeline (4× MSAA)	Sorted Source Graphs (4× MSAA)		Sorted Merged Graphs (4× MSAA)	
NVIDIA® GeForce GTX970					
	Depth	Depth	Saturate/Alpha	Depth	Saturate/Alpha
0	4.90 (7.60)	0.10 (0.27)	0.10 (0.25)	0.27 (0.52)	0.27 (0.51)
27	5.17 (6.40)	0.30 (0.48)	0.29 (0.46)	0.49 (1.01)	0.49 (0.97)
49	5.17 (6.20)	0.34 (0.56)	0.34 (0.54)	0.46 (0.88)	0.46 (0.79)
NVIDIA® GeForce GTX560					
	Depth	Depth	Saturate/Alpha	Depth	Saturate/Alpha
0	15.1 (15.1)	0.62 (0.9)	0.72 (0.90)	2.11 (2.4)	2.11 (2.50)
27	7.0 (6.40)	0.82 (1.25)	1.51 (3.30)	2.11 (2.60)	2.50 (3.95)
49	6.3 (6.65)	1.0 (1.53)	1.90 (4.26)	1.60 (2.0)	2.10 (3.8)

7. Conclusion

We have presented a pipeline for the specific purpose of rendering scenes with *predefined* camera and animation paths.
We capture the visible triangles per frame and store the dependencies of each in a separate graph. These graphs are subsequently merged, thus reducing redundancies and preventing circular dependencies without violating individual triangle orders. The final graph structure of an entire frame sequence is interpreted as an upper triangular matrix. The position of each element determines the visibility of triangles during the camera animation. To reconstruct the original graphs from the merged structure we implemented a sliding window. This window contains all graphs with a start-index lower than or equal to the frame index to be rendered and an ending index greater than or equal to the same frame index. The final draw call issues a buffer containing the offsets to the individual graphs of the sliding window.

The goal of our implementation was to make the rendering of a fixed camera path as fast as possible, which meant removing any invisible triangles beforehand and rendering a sorted set of triangles. The challenge of such an proposition was to merge different view-specific graphs while allowing a “hardware friendly” traversals and rendering.
Our experiments showed that the removal of invisible triangles significantly speeds up the rendering. We also showed that disabling the visibility test can yield additional benefits, depending on the number of fragments.

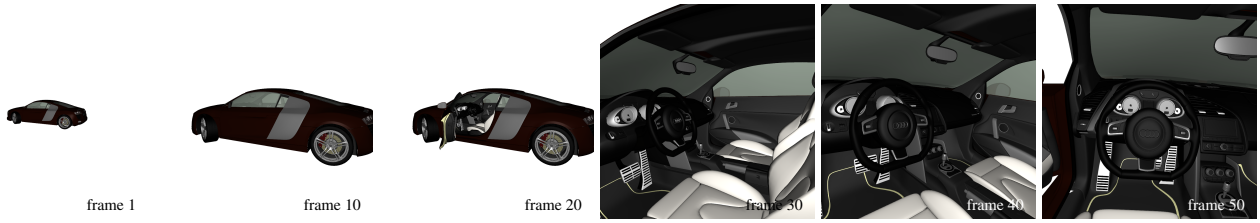


Figure 13: Camera Animation Sequence. Zooming, opening the door, translating and rotating the camera.

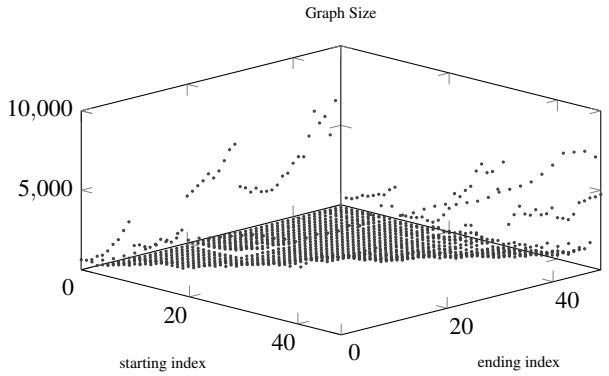


Figure 14: Triangle Distribution for 50 frames. Top-right graph contains 174,583 triangles. Original approximate size of source graphs on diagonal: ≈100,000. Mesh triangle count: 5,339,724. Every mark is a graph of our upper triangular matrix. The start and end index of a graph indicate the frames between which triangles are visible. The top-right graph[0,49] is always visible.

Acknowledgements

This work was partly supported by the Research Training Group 1773 "Heterogeneous Image Systems", funded by the German Research Foundation (DFG).

References

[BJFS12] BEHR J., JUNG Y., FRANKE T., STURM T.: Using images and explicit binary container for efficient and incremental delivery of declarative 3d scenes on the web. In *Proceedings of the 17th International Conference on 3D Web Technology* (New York, NY, USA, 2012), Web3D '12, ACM, pp. 17–25. 2

[BKSSk11] BARTA P., KOVÃĄCS B., SZÃĽCSI S. L., SZIRMAY-KALOS L.: Order independent transparency with per-pixel linked lists, 2011. 2

[Cat74] CATMULL E. E.: *A Subdivision Algorithm for Computer Display of Curved Surfaces.* PhD thesis, The University of Utah, 1974. AAI7504786. 2

[CSN*12] CHEN G., SANDER P. V., NEHAB D., YANG L., HU L.: Depth-presorted triangle lists. *ACM Trans. Graph. 31*, 6 (Nov. 2012), 160:1–160:9. 2

[CW96] CHEN H.-M., WANG W.-T.: The feudal priority algorithm on hidden-surface removal. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1996), SIGGRAPH '96, ACM, pp. 55–64. 2

[EFG04] ERNST M., FIRSCHING F., GROSSO R.: Entkerner: A system for removal of globally invisible triangles from large meshes. In *Proceedings of the 13th International Meshing Roundtable, IMR 2004, Williamsburg, Virginia, USA, September 19-22, 2004* (2004), pp. 449–458. 2

[EJF10] EIKEL B., JÄHN C., FISCHER M.: Preprocessed global visibility for real-time rendering on low-end hardware. In *Proceedings of the 6th International Conference on Advances in Visual Computing - Volume Part I* (Berlin, Heidelberg, 2010), ISVC'10, Springer-Verlag, pp. 622–633. 2

[Eve01] EVERITT C.: Interactive order-independent transparency, 2001. 2

[FKN80] FUCHS H., KEDEM Z. M., NAYLOR B. F.: On visible surface generation by a priori tree structures. *SIGGRAPH Comput. Graph. 14*, 3 (July 1980), 124–133. 2

[FS06] FUKUSHIGE S., SUZUKI H.: Voronoi diagram depth sorting for polygon visibility ordering. In *Proceedings of the 4th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia* (New York, NY, USA, 2006), GRAPHITE '06, ACM, pp. 461–467. 2

[Gar09] GARANZHA K.: The Use of Precomputed Triangle Clusters for Accelerated Ray Tracing in Dynamic Scenes. *Computer Graphics Forum* (2009). 2

[HS16] HAN S., SANDER P. V.: Triangle reordering for reduced overdraw in animated scenes. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2016), I3D '16, ACM, pp. 23–27. 2

[LJB*13] LIMPER M., JUNG Y., BEHR J., STURM T., FRANKE T. A., SCHWENK K., KUIJPER A.: Fast, progressive loading of binary-encoded declarative-3d web content. *IEEE Computer Graphics and Applications 33*, 5 (2013), 26–36. 2

[LWX06] LIU B., WEI L.-Y., XU Y.-Q.: *Multi-Layer Depth Peeling via Fragment Sort*. Tech. Rep. MSR-TR-2006-81, Microsoft Research, June 2006. 2

[MSS*10] MEYER Q., SÜSSMUTH J., SUSSNER G., STAMMINGER M., GREINER G.: On floating-point normal vectors. In *Proceedings of the 21st Eurographics Conference on Rendering* (Aire-la-Ville, Switzerland, Switzerland, 2010), EGSR'10, Eurographics Association, pp. 1405–1409. 7

[SHDG*09] SAJADI B., HUANG Y., DIAZ-GUTIERREZ P., YOON S.-E., GOPI M.: A novel page-based data structure for interactive walkthroughs. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2009), I3D '09, ACM, pp. 23–29. 2

[SSS73] SUTHERLAND I. E., SPROULL R. F., SCHUMACKER R. A.: Sorting and the hidden-surface problem. In *Proceedings of the June 4-8, 1973, National Computer Conference and Exposition* (New York, NY, USA, 1973), AFIPS '73, ACM, pp. 685–693. 2

[Str74] STRASSER W.: *Schnelle Kurven- und Flächendarstellung auf graphischen Sichtgeräten.* PhD thesis, Technical University of Berlin, 1974. 2

[Wil92] WILLIAMS P. L.: Visibility-ordering meshed polyhedra. *ACM Trans. Graph. 11*, 2 (Apr. 1992), 103–126. 2