UC Irvine
ICS Technical Reports

Title
Cached geometry manager for view-dependent LOD rendering

Permalink
https://escholarship.org/uc/item/4jt597s5

Authors
Lario, Roberto
Pajarola, Renato
Tirado, Francisco

Publication Date
2004

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

ICS

TECHNICAL REPORT

Cached Geometry Manager for View-dependent LOD rendering

Roberto Lario, Renato Pajarola, Francisco Tirado

UCI-ICS Technical Report No. 04-07
Department of Computer Science
University of California, Irvine

April 2004

Information and Computer Science
University of California, Irvine

Cached Geometry Manager for View-dependent LOD rendering

Roberto Lario

Dpto. Arquitectura de
Computadores y Automática
Universidad Complutense Madrid, Spain

Renato Pajarola

Computer Graphics Lab
Computer Science Department
University of California Irvine, USA

Francisco Tirado

Dpto. Arquitectura de
Computadores y Automática
Universidad Complutense Madrid, Spain

Abstract

The new generation of commodity graphics cards with significant on-board video memory has become widely popular and provides high-performance rendering and flexibility. One of the features to be exploited is the use of the on-board video memory to store geometry information. This strategy significantly reduces the data transfer overhead from sending geometry data over the (AGP) bus interface from main memory to the graphics card. However, taking advantage of cached geometry is not a trivial task because the data models often exceed the memory size of the graphics card. In this paper we present a dynamic cached geometry manager (CGM) to address this issue. We show how this technique improves the performance of real-time view-dependent level-of-detail (LOD) selection and rendering algorithms of large data sets. Our approach has been analyzed over two view-dependent progressive mesh (VDPM) frameworks: one for rendering of arbitrary manifold 3D meshes, and one for terrain visualization.

1. Introduction

The functionality and speed of graphics hardware has increased significantly in last few years, making the GPU a programmable stream processor with sufficient power and flexibility to perform intensive calculations. Despite the advances in the graphics hardware, the data transfer from main memory to the graphics card remains the major bottleneck [HCH03]. This restriction prevents the full exploitation of the potential computational horsepower of the GPU and introduces significant overhead in short data transfers [THO02].

View-dependent level-of-detail (LOD) algorithms can significantly reduce the amount of data transfer as the geometric scene complexity is adaptively minimized using a view-dependent error metric [LRC*03]. The adaptive nature of such methods introduces frequent but small geometric changes between consecutive frames. Our goal is to take advantage of this fact using the video memory of the modern consumer graphics hardware as geometry cache.

The rendering performance can greatly be improved if the geometric data of a given scene is stored in video memory. However, the limited size of available video memory restricts the complete storage of big data models. The use of view-dependent LOD algorithms can provide a solution to this problem because the geometric information required for rendering a scene at a certain LOD is in general only a small fraction of the full resolution model. This portion of geometry information can be cached on the graphics card using video memory (see Figure 1) and is updated every frame when the viewpoint location of the camera or the resolution is changing. In order to efficiently handle the frequent video memory updates, a *Cached Geometry Manager* (CGM) is needed. The continuous adaptive LOD changes guarantee that only a small amount of the cached geometry in the video memory has to be updated between consecutive frames.

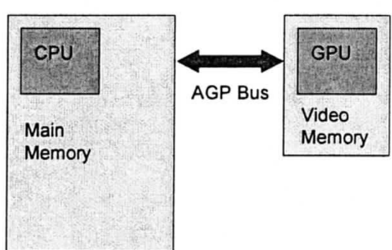


Figure 1: *CPU/GPU communication diagram.*

In this paper we describe several strategies to implement an efficient geometry-cache manager. Two *view-dependent progressive mesh* (VDPM) frameworks

are used to test the proposed techniques and to show the speed-up in rendering performance when applied to a general view-dependent LOD algorithm. The first framework is *FastMesh* [Paj01], it uses an efficient view-dependent and adaptive LOD method for rendering arbitrary 3D meshes in real-time. The general concepts of this framework are common to most similar VDPMs, e.g. such as [XV96], [Hop97], [LE97], [DMP97] or [KL01]. The second framework is *QuadTIN* [PAL02], an efficient quadtree-based triangulation approach for irregular terrain height-fields that provides fast quadtree-based adaptive triangulation, view-dependent LOD-selection and real-time rendering. Many interactive terrain visualization systems, e.g. such as [SS92], [LKR*96], [DMP96], [Pup96], [Paj98], [BAV98] or [EKT01], exhibit a similar top-down LOD triangulation and rendering approach.

The remainder of the paper is organized as follows: Section 2 presents a very brief overview of related work. Section 3 describes the Cached Geometry Manager. In Section 4 the two VDPM frameworks are presented to test the CGM approach. Experimental results are presented in Section 5 and Section 6 ends the paper with some conclusions.

2. Related Work

Despite the extensive work on level-of-detail (LOD) techniques [LRC*03], only very few methods that use cached geometry have been proposed recently. One possible reason for this lack is that only recent generations of graphics cards allow the application program to manage large amounts of video memory systematically for storing geometry.

In [Lev02] an terrain rendering algorithm is presented that operates on clusters of cached geometry called aggregate triangles. The dynamically generated aggregate triangles are kept in the geometry cache for several frames to improve rendering performance. A similar concept is followed in [CGC*03, CGC*03*] where a LOD hierarchy of simplified height-field triangle patches is generated in a pre-process. At run-time the appropriate LOD triangle patches are selected for rendering and a LRU strategy is used for caching. In [LPT03] square patches of a quadtree-based hierarchical terrain triangulation are used for fast rendering and caching in video memory. A common limitation of the above methods is that they are restricted special-purpose solutions for terrain rendering and not applicable in general to other VDPM frameworks. In contrast, the concepts presented in this paper are directly applicable to a wide range of VDPM frameworks.

A remarkable approach to provide seamless geometric LODs is provided by GLOD [CLD*03]. It allows advanced users to define discrete LOD objects as well as specify the use of video memory for patches of the geometry. In contrast to GLOD, the proposed Cached Geometry Manager interacts directly with VDPM frameworks that dynamically generate continuously adaptive LOD meshes and provides transparent use of video memory.

3. Cached Geometry Manager (CGM)

Most view-dependent simplification frameworks represent the geometry in a hierarchical data structure called *vertex hierarchy* (see Figure 2). The nodes located near the root correspond to low-resolution vertices while those located farther away represent high-resolution detail vertices. The vertex hierarchy is dynamically queried to perform the view-dependent simplification. The *front* of active nodes divides the current nodes used to generate the simplified scene from the rest. This frontier is continuously updated for every rendered frame.

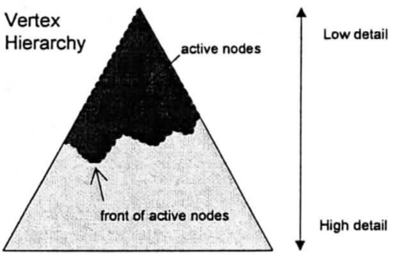


Figure 2. *Vertex hierarchy diagram.*

By far most of the vertices of a scene remain active beetwen consecutive frames and just a small fraction changes its state. Hence most vertices can be stored and kept continuously in video memory in order to improve rendering performance. Each frame only few vertices require a read operation from main memory, to transfer to video memory, when they change their state from inactive to active. A remove operation is needed when the video memory is full and new vertices have to be added. Inactive but cached vertices are the candidates to be deleted from video memory in this case. A video memory manager is requiered to carry out these operations.

3.1. Vertex Arrays

Indexed vertex buffers, called vertex arrays or VARs in OpenGL, are the best way to take advantage of modern graphics accelerator (see section 11.4.5 in [MH02]). The application puts the data into specific buffers and gives the pointers to the driver, which accesses the data directly, as shown in Figure 3. Hence vertex arrays need much fewer OpenGL funtion calls for rendering than the classic *immediate mode vertex submission* (using

glBegin()/...glVertex().../glEnd() blocks*)*. In [Mar00], several methods to optimize submission of vertex data in OpenGL are described. Our Cached Geometry Manager takes advantage of vertex arrays in combination with the OpenGL extension *NV_vertex_array_range* [Kil99].

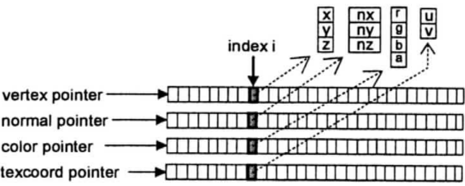


Figure 3: *Indexed Vertex Arrays (VAR).*

As illustrated in Figure 4, the order and positions of vertices is different in the cached VAR from the main memory VAR. Thus the vertex indices of an indexed triangle mesh must be transformed accordingly for rendering. However, the rendering speedup will compensate for this extra re-indexing required by the CGM.

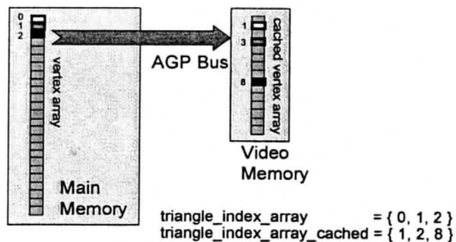


Figure 4: *Different index arrays for the global VAR and for the cached-VAR.*

3.2. CGM Strategies

In this and the following section we describe three basic caching strategies to implement the video memory manager. These three strategies are going to be discussed for two variants of VDPM frameworks in order to cover the range of applications: those which calculate an explicit front of active nodes by incremental updates between consecutive frames, and those which implicitly define the active front by selecting the active nodes top-down for each frame (see Figure 2).

In this section we first discuss the more general case of implicit active front VDPM frameworks. Note that a non-explicit front does not mean it does not exist, in fact the front always implicitly exists in any view-dependent LOD framework. The implicitly-defined refers to the behaviour of the VDPM framework that has no other information than if a vertex is selected or not for each rendered frame.

From here on we will refer to both video memory and geometry cache as the same concept.

The basic two tasks of the cache manager are: (1) to determine that a vertex is already resident (cached) in the video memory, and (2) to find and use an open slot in the cache to store a new vertex. Task (1) can efficiently be determined by a cross indexing: each vertex in main memory has a field that indicates the cache index where it was last stored, and each cache slot has an index field indicating which vertex is stored. More complicated is task (2) for which we describe viable strategies below.

***First-Available* Strategy (FA):** This simple strategy uses the video memory as a linear list of slots with flags. This list is traversed from the beginning to the first non-used slot (First Available) every time a new vertex must be cached. This slot is then marked as used. The process continues while there are vertices to cache, a pointer is moved from the head to the end searching the next available open slot. Owing to the fact that the list of slots is sequentialy traversed this strategy can be implemented using a simple *array*, as illustrated in Figure 5 a). Each slot is considered used when it stores a vertex used in the current or last frame. This policy considers the fact that it is very likely that a vertex used in frame i will be also required in frame i+1. Hence each slot flag is an integer counter which stores the last frame that cached vertex was used. This strategy is simple to implement, but has one potential drawback: unused slots near the beginning of the list will immediately be overwritten when a new vertex has to be cached while unused slots at the end may cache an unused vertex for a long time. This bias of re-using cache slots based on their position is not necessarily the best solution. The next strategy addresses this problem.

***LRU* Strategy:** As mentioned above, the FA strategy considers any empty slot in the cache as equally good. If a slot has not been used in the current or last frame it is considered available. However, there is an intuitive reason that more recently used vertices are more likely to be used again as vertices that have not been used for a long time. Hence a more refined policy is to take into account the age of the unused slots and use a last-recently-used (LRU) strategy. The LRU parameter is directly obtained from the frame counter associated with each slot. One possible data structure to make use of this strategy is a *doubly-linked-list*. Two pointers (*head* and *tail*) are needed for the proposed implementation as shown in Figure 5 b).

The *head* points to the youngest slot, and the *tail* points to the oldest slot. New vertices are cached in the slot pointed to by *tail* which is then moved to the *head*. Reused slots of rendered vertices already in cache are simply moved from their current position in the linked

list to the *head*. Consequentially, unused slots automatically move towards the *tail* which points always to the oldest slot entry. Note that these operations do not imply a displacement of the actual vertex data in video memory, it is just a mechanism for the cache manager to maintain access to the last-recently-used open slot. Each slot in this linked list corresponds to a fixed memory location in the cache.

***LRU + Error-PriorityQueue* Strategy:** Figure 2 shows clearly that the vertices near the top of the hierarchy are more significant as they correspond to lower LOD information. Consequently these vertices are included in the mesh representation before any vertices of higher LODs. For this reason, for a new vertex it is more suitable to choose among the empty slots the one that corresponds to an old vertex which represents a high level of detail. In order to add this new feature to the CGM we propose to categorize the age of the unused slots and introduce a priority-queue for the oldest-category vertices. The oldest category vertices are naturally and compactly stored at the end of the LRU list described above. Hence as shown in Figure 5 c) we manage the last section of the LRU list in a priority-queue with the LOD error-metric parameter as key. Note that it is not advisable to choose a big priority queue size since this data structure is more costly than the doubly-linked-list of the simple LRU approach.

As with the LRU approach, reused slots are moved from the current location to the *head* and unused slots slowly sink towards the *tail*. The *tail* marker also indicates the bounds of the oldest-category. Thus elements at the *tail* are moved to the priority-queue as soon as their age has reached a certain limit and the priority-queue is not at maximal capacity. When a new vertex has to be inserted into the cache the top slot of the priority-queue is used and moved to the *head*.

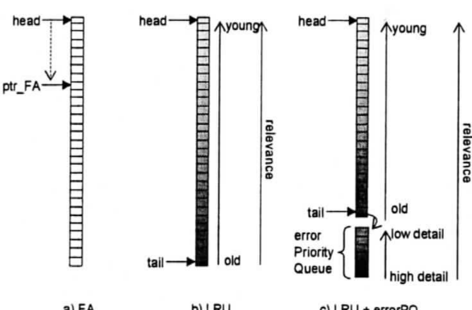


Figure 5: *Data structures for the CGM strategies. The relevance of a cached vertex is defined as low probability to be removed when a new vertex in going to be inserted in video memory.*

3.3. CGM Strategies for Front-Frameworks

The strategies described in the previous section can be refined if the VDPM framework has explicit knowledge of which vertices have been removed from and which ones have been added to the current LOD triangle mesh. This feature is typical in LOD systems that maintain an explicit active front for the current frame and update this front incrementally as illustrated in Figure 6. The newly activated vertices are called *added* (+) vertices, and those deactivated are called *removed* vertices (-). For each frame the *added* vertices have to be inserted into video memory, if not already cached, while the *removed* vertices remain cached but change their slot flag to be unused. Note that the *removed* vertices have always been active in the previous frame.

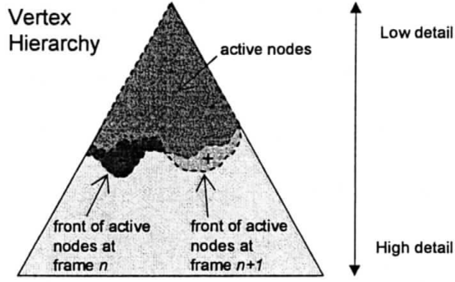


Figure 6: *Vertex hierarchy diagram of a front-framework..*

***First-Available* Strategy for front-framework:** This strategy, while obviously suboptimal when information about both *added* and *removed* vertices is explicitly provided by the LOD system, applies without changes to explicit-front frameworks.

***LRU* Strategy for front-framework:** The LRU policy described previously can be improved using a third pointer, called *frontier* in Figure 7, to divide the active slots from the inactive ones. The slots of *removed* vertices are moved to just below the *frontier* while slots of *added* vertices change their position from the *tail* to the *head*. Advantage can be taken for vertices that were already active in the previous frame because their corresponding slot in the LRU list is not affected by any move operation. Note that these reused vertices are by far the largest part of active vertices. Therefore, compared to the basic LRU cache algorithm, linked-list operations are limited to the few *removed* and *added* vertices.

Like other terrain visualization systems, QuadTIN selects the active vertices for a LOD of a particular viewpoint in a top-down quadtree traversal for each frame. Hence QuadTIN belongs to the category of VDPM with implicitly defined active front and does not provide explicitly the *removed* or *added* vertices between consecutive frames. Experimental results using the CGM strategies given in Section 3.2 are reported in Section 5.2 for this frame.

5. Experimental Results

Experimental results were performed on a 3.0 GHz Pentium 4 with 1GB RAM using an NVIDIA GeForce4 Ti 4400 graphics card. In all the scenes a 45° vertical field-of-view camera followed several trajectories as described below.

5.1. FastMesh Results

The models used with the FastMesh rendering system are given in Table 1. The rendering experiments were averaged over 1000 frames in a window of 800 x 800 pixels using an error tolerance equal to one pixel (projective tolerance of geometric error projected on screen).

model	female	hand	dragon
vertices	302948	327323	437645
faces	605086	654666	871414

Table 1: *3D models used with FastMesh.*

Three different camera trajectories have been analyzed to examine the impact of the Cached Geometry Manager in the FastMesh framework as illustrated in Figure 9:

- Circular camera trajectory.
- Small camera rotations.
- Straight line camera trajectory.

These camera movements are very common as they are typical movements observers normally execute to explore a 3D object. The camera is pointing to the center of the model in all the trajectories.

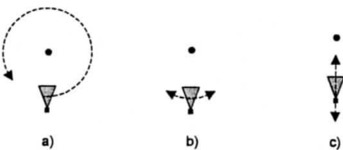


Figure 9: *3D object rendering camera trajectories: a) circular trajectory. b) small rotations. c) straight line trajectory (zoom in/out).*

The chosen CGM size (slot-list length) was 2^{17} = 131072 because all the models required at least 2^{16} = 65536 vertices to be rendered from every tested viewpoint. Of course, in this context no LOD mesh can have more vertices than available in the geometry cache, and the application program must make sure that the best LOD for the limited number of vertices is selected. If this mesh size exceeds the cache size then the application should disable the CGM and render the mesh in normal mode, or possibly render the mesh using the CGM in multiple passes. We are only studying the effect of the CGM in this paper and do not address the latter issues in this work.

In the experiments, the size of each vertex element is 36 bytes, consisting of: 3 floats (position) + 3 floats (normal) + 3 floats (color).

We have focused the numerical results on the biggest model (the Dragon) to avoid excessive and repeated information. Statistical data for the other models is given in Table 3. Figure 10 shows the per-frame timing results (in miliseconds) of Dragon model for the three different camera trajectories. The total time per frame has been divided into three parts: the *rendering* time, the time needed to construct the vertex array (*build VAR*), and the time required to perform the error metrics and updating the LOD mesh (*others*). Our CGM techniques only affects the vertex array construction and rendering time but not any other tasks of the VDPM framework.

As we mentioned in section 4.1, the initial version of FashMesh traverses a linked-list of triangles to render the model in *immediate mode vertex submission*. This mode has been included in our tests in the first column of Figure 10 a), b), and c). The second column represents the standard VAR mode (characterized by the OpenGL function *glDrawElements*). This mode is more time expensive than the previous due to the transformation from a linked-list of triangles to an indexed triangle array.

The next columns report the CGM modes. The *LRU+errorPQ%10* strategy employs an error priority queue size equal to 10 percent of the total cache size. As expected, the rendering improvement is significant. The build VAR time for these modes increases the workload of the CGM. However, the global speedup obtained for the three strategies easily compensates the extra CGM cost. In fact, the actual rendering cost, which is the only cost affected by the CGM, is improved by a factor of up to 3 (including the VAR build time).

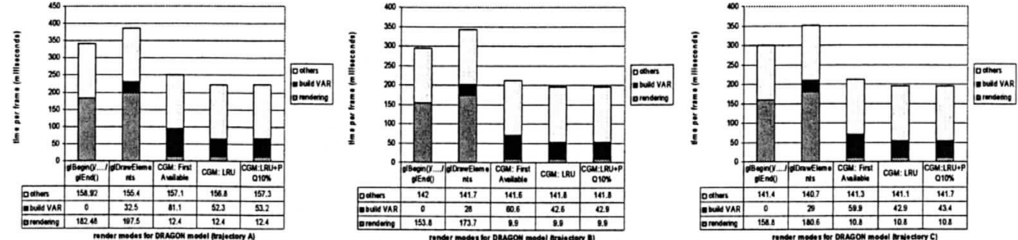


Figure 10: *Per frame timing results (in miliseconds) for the Dragon model for: a) circular camera trajectory, b) small camera rotations and c) straight line camera trajectory. The build VAR time contains the time consumed by the cache memory manager in modes where the CGM is enabled.*

Despite three different camera trajectories, all show almost the same behavior. More information may be obtained taking into account the number of vertices inserted in video memory. It allows to undestand which strategy makes better use of the cached geometry since more inserted vertices involves more data transfer from main to video memory. This information is given in Table 2.

CGM TYPE	Trajectory A	Trajectory B	Trajectory C
First Available	1286367	1115502	260863
LRU	1068700	132966	122049
LRU+PQ10%	1030306	134891	122049

Table 2: *Vertices inserted into video memory for the different CGM modes over 1000 frames (Dragon model).*

The *First Available* strategy clearly makes the worst use of cached geometry, especially for small camera rotations. In contrary to our initial expectations, in most cases the simple *LRU* strategy outperforms the *LRU + Error-PriorityQueue* strategy. The latter gives the best result only for the circular camera trajectory, and even in this case, the lower data transfer rate does not compensate for the more expensive priority queue operations.

Figure 11 b) and c) show the inefficiency of the *First Available* strategy for the last two camera trajectories, where the insertion of new vertices is unnecessary after a certain number of frames. Recall the most expensive frame is always the first because the cache is empty of vertices.

Table 3 lists the speedups achieved by the different CGM strategies with respect to the original *immediate mode vertex submission* FastMesh version. The first column corresponds to the variant with a standard main-memory VAR but no CGM. The last three columns indicate the speedup factors for the three implemented CGM strategies. Note that these speedup factors take the entire process into account, including the LOD selection phase which is not accelerated by the use of a CGM. The individual speedups for the rendering stage reach factors up to 3, see Figure 10, which shows the real impact of the Cached Geometry Manager. The Figure 15 shows three Fastmesh rendering examples for straight line camera trajectory.

Model	camera trajectory	CGM render modes			
		std VAR (NO CGM)	First Available	LRU	LRU + PQ10%
female	A	0.89	1.42	1.62	1.6
	B	0.87	1.42	1.57	1.56
	C	0.85	1.47	1.61	1.61
hand	A	0.86	1.38	1.56	1.56
	B	0.88	1.48	1.62	1.61
	C	0.88	1.5	1.61	1.64
dragon	A	0.89	1.36	1.54	1.53
	B	0.86	1.39	1.52	1.52
	C	0.86	1.42	1.54	1.53

Table 3: *FastMesh speed-ups for different CGM strategies.*

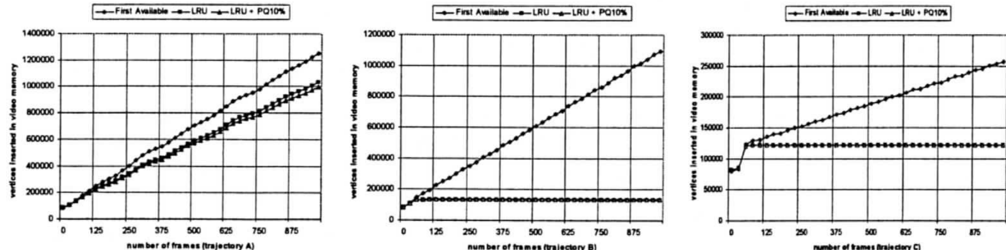


Figure 11: *Vertices inserted into video memory for the Dragon model for: a) circular camera trajectory, b) small camera rotations and c) straight line camera trajectory.*

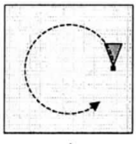
5.2. QuadTIN Results

The height-field model used for the QuadTIN rendering experiment is the well known Puget Sound data set (2563548 vertices, after QuadTIN-preprocess error tolerance = 6 meters). The results were averaged over 3000 frames in a window of 1024 x 768 pixels using an error tolerance equal to one pixel (projective tolerance of geometric error projected on screen).

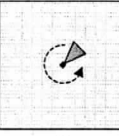
The chosen CGM size was $2^{16} = 65536$ following the same criteria applied as for the experiments with FastMesh. The size of each vertex element in this case is 32 bytes: 3 floats (position) + 3 floats (normal) + 2 floats (texture coordinate).

The camera trajectories tested to perform the CGM analysis with the QuadTIN rendering system are the following (see Figure 12):

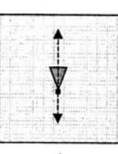
- Circular camera trajectory.
- Camera rotation with fixed eye.
- Straight line camera trajectory.



a)



b)



c)

Figure 12: *Terrain rendering camera trajectories: a) circular trajectory. b) camera rotation with fixed eye. c) straight line trajectory. The rectangle represents the height field.*

The CGM strategies applied to QuadTIN are given in Section 3.2. The total time per frame has been divided in three parts: the *rendering* time, the *CGM* cost, and the time needed to select the vertices and build the triangle strip (*others*). The QuadTIN system constructs an indexed triangle strip as required for a standard VAR approach. Note that due to the implicit definition of the active front, no information about *added* or *removed* vertices is provided, QuadTIN only reports which vertices are selected for a particular frame.

Figure 13 shows four columns, one for each rendering strategy, for each camera trajectory: the first column for the standard VAR mode, and the three others for the different CGM strategies. In this case, the best result is achieved by the *First Available* (*FA*) strategy.

Despite the fact that the *FA* strategy still makes the worst use of the geometry cache (see Table 4), its simple and fast data structure are still advantageous over the doubly-linked list of slots in the two *LRU* CGM strategies. This result is not completely surprising as the minor data transfer overhead of *FA* is amortized by the simple and fast array data structure for slots.

CGM TYPE	Trajectory A	Trajectory B	Trajectory C
First Available	171873	302288	290747
LRU	169928	286331	66453
LRU+PQ10%	169972	281707	66436

Table 4: *Vertices inserted in video memory for the different CGM modes over 3000 frames (Puget Sound model).*

The straight line camera trajectory deserves a special discussion (see Figure 14 c)) since the *FA* strategy remains the fastest despite its bad reuse of cached geometry. The *LRU* and *LRU+PQ10%* strategies require more computation time but much less data transfer. Depending on the CPU speed in relation to the AGP bus bandwith this result may slightly change, and the *LRU* strategies may win over the *FA* strategy for certain configurations. The relation between CPU speed and AGP bus bandwith of the system will decide which is the best strategy.

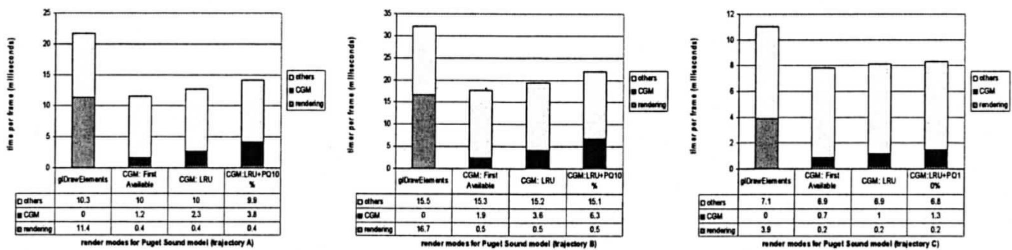


Figure 13: *Per frame timing results (in miliseconds) for the Puget Sound data set for: a) circular camera trajectory, b) camera rotation with fixed eye and c) straight line camera trajectory.*

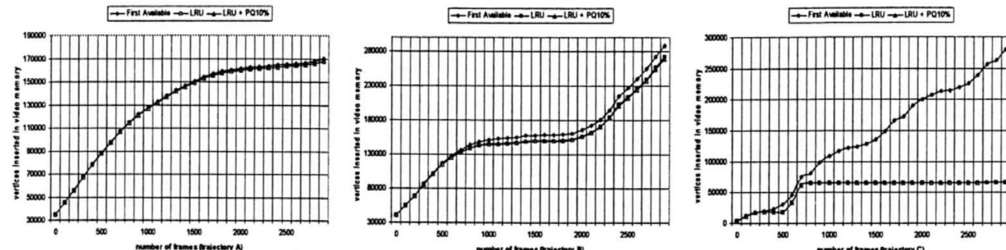


Figure 14: *Vertices inserted into video memory for the Puget Sound data set for: a) circular camera trajectory, b) camera rotation with fixed eye and c) straight line camera trajectory.*

The QuadTIN overall speedup factors in Table 5 obtained using our CGM are even better than those achieved for 3D meshes using FastMesh. This is because the QuadTIN system, as most other terrain visualization systems, has a simpler LOD selection process compared to arbitrary 3D mesh VDPM frameworks. The speedup of the rendering stage itself, which is the only stage affected by the CGM, reaches factors up to 5 or higher in some cases as seen in Figure 13. This dramatically shows the potential of using a CGM in a view-dependent LOD rendering system. The Figure 16 illustrates the QuadTIN wireframe-rendering examples for the three camera trajectories.

camera	CGM render modes		
trajectory	First Available	LRU	LRU + PQ10%
A	1.87	1.71	1.54
B	1.82	1.67	1.47
C	1.41	1.36	1.33

Table 5: *QuadTIN speedups for different CGM modes.*

6. Conclusion

This work presents several strategies to implement an efficient Cached Geometry Manager that takes advantage of on-board video card memory for caching vertex data. It provides effective solutions to manage the video memory as a geometry cache in order to dramatically reduce the vertex data transfer rate from main to video memory for each rendered frame. The proposed techniques can be applied to a wide range of view-dependent LOD rendering frameworks, and allow the efficient reuse of cached geometry information stored on the video graphics card.

The presented approaches significantly improve the overall performance of various view-dependent LOD rendering applications with little extra implementation effort. Experimental results on two different VDPM frameworks have confirmed the suitability and the effectiveness of our approach to accelerate the rendering stage. In fact, the rendering stage itself is dramatically accelerated by at least 3, and up to 6 times.

7. Acknowledgments

This research was supported by the Spanish research grant TIC 2002-750 and the New Del Amo award UCDM-33657.

References

[BAV98] BALMELLI L., AYER S., VETTERLI M.. Efficient algorithms for embedded rendering of terrain models. *Proceedings IEEE International Conference on Image Processing ICIP 98* (1998), pp. 914-918.

[CGC*03] CIGNONI P., GANOVELLI F., GOBBETTI E., MARTON F., PONCHIO F., SCOPIGNO R.. BDAM Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization. *Proceedings EG/IEEE TCVG Symposium on Visualization 2003*.

[CGC*03*] CIGNONI P., GANOVELLI F., GOBBETTI E., MARTON F., PONCHIO F., SCOPIGNO R.: Planet-Sized Batched Dynamic Adaptive Meshes (P-BDAM). *Proceedings IEEE Visualization 2003*, pp. 147–154.

[CLD*03] COHEN J., LUBKE D., DUCA N., SCHUBERT B.: GLOD: Level of Detail for the Masses. URL http://www.cs.jhu.edu /~graphics/ TR/TR03-4.pdf.

[DMP96] DE FLORIANI L., MARZANO P., PUPPO E.: Multiresolution models for topographic surface description. *The Visual Computer* (August 1996), pp. 317-345.

[DMP97] DE FLORIANI L., MAGILLO P., PUPPO E.: Building and traversing a surface at variable resolution. *Proceedings IEEE Visualization 97* (1997), pp. 103-110.

[EKT01] EVANS W., KIRKPATRICK D., TOWNSEND G.. Right-triangulated irregular networks. *Algorithmica* (March 2001), pp. 264-286.

[HCH03] HALL J. D., CARR N. A., HART J. C.: Cache and Bandwidth Aware Matrix Multiplication on the GPU. Technical Report UIUCDCS-R-2003-2328. University of Illinois at Urbana-Champaign Computer Science Department. April 2003.

[Hop97] HOPPE H.: View-dependent refinement of progressive meshes. *Proceedings SIGGRAPH 97* (1997), pp. 189-198.

[Kil99] KILGARD M. J.. NVIDIA OpenGL Extension Specification NV_vertex_array_range. URL: http://www.nvidia.com/dev_content/nvopenglspecs/GL_NV_vertex_array_range.txt.

[KL01] KIM J., LEE S.. Truly selective refinement of progressive meshes. *Proceedings Graphics Interface 2001*, pp. 101-110.

[LE97] LUEBKE D., ERIKSON C.. View-dependent simplification of arbitrary polygonal environments. *Proceedings SIGGRAPH 97* (1997) pp. 199-208.

[Lev02] LEVENBERG J.. Fast view-dependent level-of-detail rendering using cached memory. *Proceedings IEEE Visualization 2002*, pp. 259-265.

[LKR*96] LINDSTROM P., KOLLER D., RIBARSKY W., HODGES L. F., FAUST N., TURNER G. A.. Real-time, continuous level of detail rendering of height fields. *Proceedings SIGGRAPH 96* (1996), pp. 109-118.

[LPT03] LARIO R., PAJAROLA R., TIRADO F.. Hyperblock-QuadTIN: Hyper-block quadtree based triangulated irregular networks. *Proceedings IASTED Invernational Conference on Visualization, Imaging and Image Processing* (VIIP 2003), pp. 733-738.

[LRC*03] LUEBKE D., REDDY M., COHEN J., VARSHNEY A, WATSON B., HUEBNER R.. Level of detail for 3D graphics. Morgan Kaufman. 2003.

[Mar00] MARSELAS H.: Optimizing Vertex Submission for OpenGL. Game Programming Gems, pp. 353-360. Charles River Media. 2000.

[MH02] MÖLER T., HAINES E,. Real-time rendering. 2nd edition. A K Peters. 2002.

[Paj98] PAJAROLA R.: Large scale terrain visualization using the restricted quadtree triangulation. *Proceedings IEEE Visualization 98* (1998), pp. 19-26 and 515.

[Paj01] PAJAROLA R.. FastMesh: Efficient View-dependent Meshing. *Proceedings Pacific Graphics 2001*, pp. 22-30.

[PAL02] PAJAROLA R., ANTONIJUAN M, LARIO R.: QuadTIN: quadtree based triangulated irregular networks. *Proceedings IEEE Visualization 2002*, pp. 395–402.

[Pup96] PUPPO E.: Variable resolution terrain surfaces. *Proceedings of the 8th Canadian Conference on Computational Geometry* (1996), pp. 202-210.

[SS92] SIVAN R, SAMET H.. Algorithms for constructing quadtree surface maps. *Proceedings 5th International Symposium on Spatial Data Handling* (August 1992), pp. 361-370.

[THO02] THOMPSON C. J., HAHN S., OSKIN M.: Using modern graphics architectures for general-purpose computing: a framework and analysis. *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture* (2002), pp. 306-317.

[XV96] C. XIA J. C., VARSHNEY A.. Dynamic view-dependent simplification for polygonal models. *Proceedings IEEE Visualization 96* (1996), pp. 327-334.

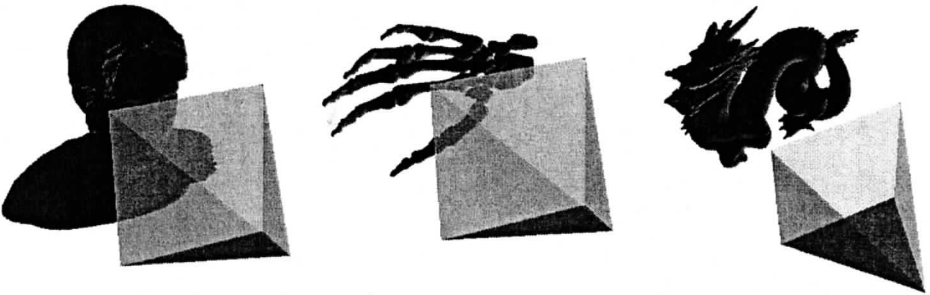
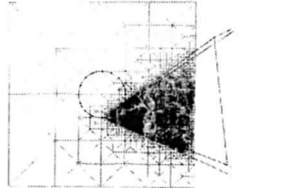
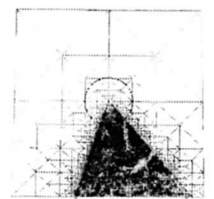


Figure 15: *Fastmesh rendering examples for straight line camera trajectory for:* ***left)*** *female,* ***center)*** *hand,* ***right)*** *dragon. The images show the medium distance from the center of the model to the viewpoint used for the three camera trajectories.*





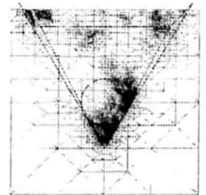


Figure 16: *QuadTIN wireframe-rendering examples for:* ***left)*** *circular camera trajectory,* ***center)*** *camera rotation with fixed eye, and* ***right)*** *straight line camera trajectory.*