

Continuous Level of detail on Graphics Hardware

Francisco Ramos, Miguel Chover, Oscar Ripolles, and Carlos Granell

Universitat Jaume I, Depto. Lenguajes y Sistemas Informaticos
12071 Castellon, Spain

{Francisco.Ramos,chover,oripolle,canut}@uji.es

Abstract. Recent advances in graphics hardware provide new possibilities to successfully integrate and improve multiresolution models. In this paper, we present a new continuous multiresolution model that maintains its geometry, based on triangle strips, in high-performance memory in the GPU. This model manages the level of detail by performing fast strip updating operations. We show how this approach takes advantage of the new capabilities of GPUs in an efficient manner.

1 Introduction

One of the main problems of interactive graphic applications, such as computer games or virtual reality, is the geometric complexity of the scenes they represent. In order to solve this problem, different techniques for modeling by level of detail have been developed that attempt to adapt the number of polygons of the objects to their importance within the scene.

The application of these techniques is common in standards such as X3D, graphic libraries such as OpenInventor, OSG, and even in game engines such as Torque, CryEngine, and so forth, where models with continuous levels of detail, based mainly on Progressive Meshes [1], are introduced.

The tendency in recent years has been to improve the features of continuous models by using the possibilities offered by the graphics hardware to the maximum, with the intention of competing with the discrete models that, although more limited, are perfectly adapted to current graphics hardware. Specifically, they have worked on the representation of multiresolution models which use triangle strips to accelerate visualization by means of vertex arrays located in the GPU. The fundamental problem of these techniques is the fact that a continuous model needs to make changes in the list of indexes of the primitives it draws and carrying out this kind of operations causes graphics hardware to lower its performance.

1.1 Related Work

In recent years, multiresolution models have progressed substantially. At the beginning, discrete models were employed in graphics applications, due mainly

to the low degree of complexity involved in implementing them, which is the reason why nowadays they are still used in applications without high graphics requirements. Nevertheless, the increase in realism in graphics applications make it necessary to use multiresolution models which are more exact in their approximations, which do not call for high storage costs and which are faster in visualization. This has given way to continuous models, where two consecutive levels of detail only differ by a few polygons and where, additionally, the duplication of information is avoided to a considerable extent, thus improving on the spatial cost offered by the discrete models.

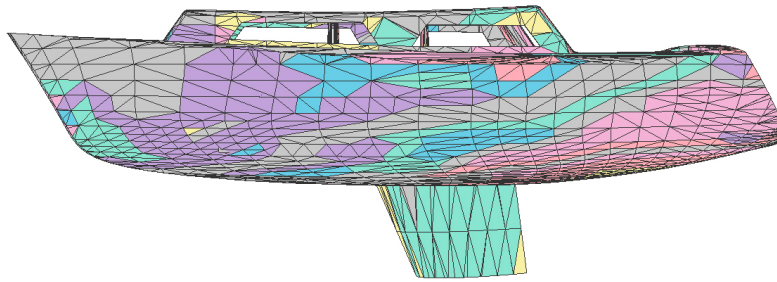


Fig. 1. Boat model in triangle strips

The best known continuous multiresolution model is Progressive Meshes [1], included in Microsoft Corporation's graphic library DirectX. This model offers excellent results in visualization in real time, although it is based on triangle primitives.

Advances have been made in the use of new graphics primitives which minimize the data transfer between the CPU and the GPU, apart from trying to make use of the connectivity information given by a polygonal mesh. For this purpose, graphics primitives with implicit connectivity, such as triangle strips (see figure 1) and triangle fans, have been developed. Many continuous models based on this type of primitives have been recently developed [2-7].

In these last few years, graphics hardware performance has evolved outstandingly, giving rise to new techniques which allow the continuous models to accelerate even more. The use of stripification algorithms, which try to take the maximum advantage of the GPU cache, and the new extensions of graphics libraries that allow visualization of a whole mesh with only a few instructions are examples of these new techniques.

Nowadays GPUs offer new capabilities that, when exploited to the maximum, can offer very good results in several aspects. One of them involves storing information directly in the high speed memory located in the GPU. This characteristic allows information to be managed in the GPU while avoiding data

transfer between the CPU and the GPU, and taking the maximum advantage of the proximity of the memory and the graphics processor. There are a number of related works which make use of the new capabilities of the current GPUs, such as [8], which implements a discrete model manager that puts geomorphing into practice by using vertex shaders; another work is [9], which creates different shaders depending on the level of detail.

1.2 Motivation

In general, the main problem with continuous models lies in the high cost of extracting the level of detail, which usually takes about 20% of the total visualization cost. Apart from extraction, the use of AGP buses poses the problem of their being much better optimized to upload data than to download it, thus favoring the use of the memory of the graphics card to store static objects that do not change their geometry. But the appearance of the PCI-Express bus makes it possible to use a symmetric bus, which allows data to be uploaded and downloaded to the GPU at the same speed, so that it is possible to work with the GPU memory in a reliable way and without penalizations in data download.

1.3 Contributions

In this article we present a new multiresolution model that is integrated into the graphics hardware. This model makes use of the present capabilities of GPUs to store its data structures inside them. The fundamental idea on which the model is based is the creation of efficient data structures that can be integrated into the GPU and which, at the same time, offer an optimum performance with respect to both visualization and spatial cost. The model works directly with the GPU memory, appreciable improvements being obtained, as can be seen in the results section.

Hence, what this model offers is complete integration into the graphics hardware, a low cost of extraction of the level of detail, by exploiting the coherence between levels of detail, and a low spatial cost.

The implemented model features different characteristics:

- Wholly based on triangle strips.
- Simplification based on progressive edge collapses.
- Static stripification. Triangle strips are only generated once, at the highest level of detail, using a method that takes advantage of the GPU cache.
- Geometric information of the model is maintained and stored in the GPU.
- Level of detail management is performed by a data structure, LOD-Manager, which allows fast updating of strips and removal of degenerated triangles.

2 Fundamentals

2.1 Multiresolution Models

To construct a continuous multiresolution model based on primitives with implicit connectivity, such as triangle strips, certain requirements must be fulfilled.

On the one hand, a mesh made up of this kind of primitives must be available and, on the other hand, the simplification method that should be employed in order to generate the different levels of detail must be selected, an example is shown in figure 2.

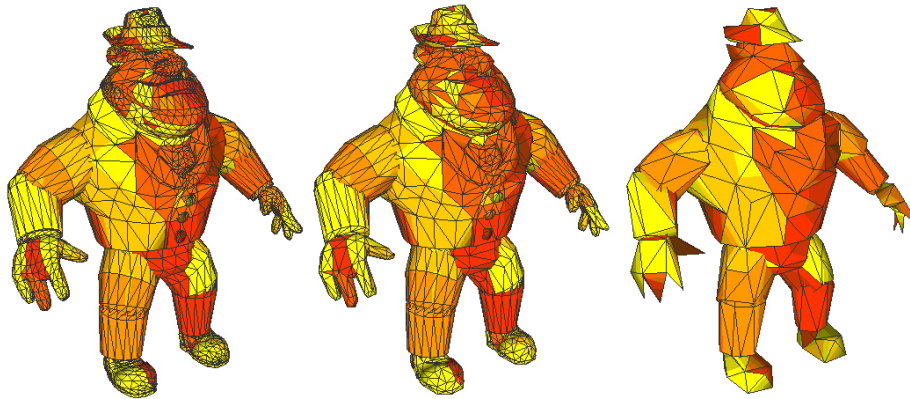


Fig. 2. Three levels of detail from the AL model (LOD=1,0.5 and 0, respectively)

There are several mesh simplification methods [10][11], but one of the most important in progressive mesh simplification is [1]. This method is based on iterative edge contractions, and it is the one employed in well-known multiresolution models such as [2-7].

Many works can be found in the literature where the problem of converting a polygonal mesh made up of triangles into triangle strips is solved [12][14]. This process is commonly called stripification, and it can be carried out in a dynamic or static way. Dynamic stripification involves generating the triangle strips in real time, that is, for each level of detail new strips are generated. On the other hand, static stripification entails first creating triangle strips and then working with versions of the original strips. There are several models that use dynamic stripification [3][4], especially variable resolution models. Other models such as [2][5-7], however, use static stripification techniques.

The main problem of static stripification models can be observed in Figure 3. As the model reaches lower levels of detail, it presents vertex repetitions that do not add any information to the final scene but nevertheless involve higher data traffic between the CPU and the GPU. Models like [2][7] solve this problem by applying filters to eliminate degenerated triangles. The first employs filters in visualization, thus avoiding sending those vertices at the moment of rendering, and the second runs a preprocess that detects them early on, and then stores that information and eliminates them from the strips before visualizing them.

Given the architecture of present-day GPUs, it is better to employ static stripification techniques since we thereby avoid strip creation and destruction in the GPU, which would imply an additional cost that would make the model

much less competitive. Furthermore, there is an additional cost stemming from the calculation of the new triangle strips at each level of detail, which also penalizes the use of these techniques. Moreover, it is preferable to eliminate degenerated triangles before visualization, which allows a considerable degree of acceleration to be accomplished by resizing strips, apart from also enabling a better implementation of the model in the GPU by avoiding the need to create a specific code for the filters. Nowadays, a variety of acceleration techniques have appeared, which, if integrated into a multiresolution model, would also become essential to improving its performance. Basically, we can observe stripification techniques oriented toward exploiting vertex caches [12] and hardware acceleration techniques based on graphics library extensions [13].

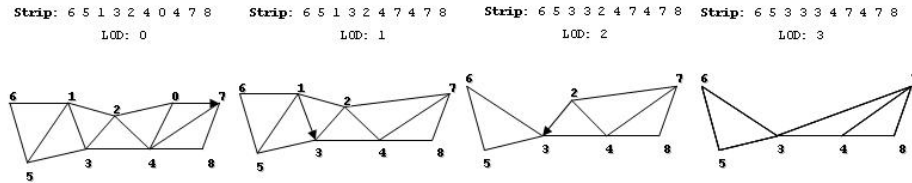


Fig. 3. Multiresolution triangle strips

2.2 High-performance memory in GPUs

A vertex buffer object is a feature that enables us to store data in high-performance memory in the GPU. The basic idea is to provide some buffers, which will be available through identifiers. There are different ways to interact with buffers:

- Bind a buffer: this activates the buffer in order to be used by the application.
- Put and get data: this allows us to copy data between a client's area and a buffer object in the GPU.
- Map a buffer: you can get a pointer to a buffer object in the client's area, but this can lead to the driver's waiting for the GPU to finish its operations.

There are two kinds of vertex buffer objects: array buffers and element array buffers. On the one hand, array buffers contain vertex attributes, such as vertex coordinates, texture coordinates data, per-vertex color data and normals. On the other hand, element array buffers contain only indexes to elements in array buffers. The ability to switch between various element buffers while keeping the same vertex array allows us to implement level of detail schemes by changing the elements buffer while working on the same array of vertices.

In order to implement the model on graphics hardware, we used different functions which interact with buffer objects. Among them, we can highlight:

- glBindBufferARB: this function sets up internal parameters so that the next operations work on this current buffer object.
- glBufferDataARB: this function is an abstraction layer between the memory and the application. Basically, this function copies data from the client memory to the buffer object bound.
- glBufferSubDataARB and glGetBufferSubDataARB: its purpose consists in replacing or obtaining, respectively, data from an existing buffer.

3 Implementation Details

3.1 General Framework

A brief outline of the model is shown in Figure 4. At the beginning, information about vertices and strips, at the highest level of detail, is uploaded into the GPU. Later, by means of the LOD-Manager data structure, strips are updated in accordance with the current level of detail.

In our approach we first perform two essential tasks: generation of triangle strips at the highest level of detail and calculation of vertex-collapse simplification.

At runtime, we upload information about vertices and strips into the GPU. Then, depending on application demands, we perform vertex-split or edge-collapse operations directly on the strips. This task is executed by the LOD-Manager. More specifically, when a level of detail transition is required, it downloads the strips affected by these changes from the GPU. Later, it modifies and uploads the updated strips to the graphics system. Lastly, strip information in the GPU is then used for display.

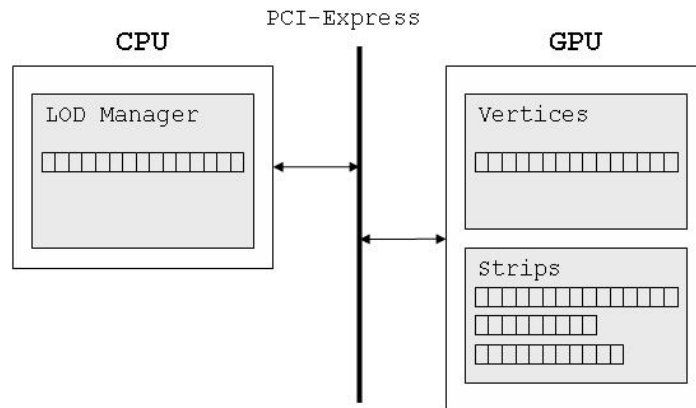


Fig. 4. Model architecture

3.2 LOD-Manager Data Structures

The main function of LOD-Manager consists in serving the level of detail demands required by applications. It is able to quickly change the geometric information located in the GPU by applying a series of pre-calculated records. These records store mainly two kinds of information: simplifications and filters.

Simplification information contains data about which strips change for each level of detail, and where the vertices to be split or collapsed are located. It allows us to quickly locate information to be modified when we move from one level of detail to another. However, as the model moves to coarse LODs, an accumulation of identical vertices is produced. Sending these vertex repetitions to the graphics hardware does not contribute at all to the final scene because it is equivalent to send degenerated triangles, as is shown in Figure 3. We have proved that most vertex repetitions can be removed, following patterns like $aa(a)^+$ or $ab(ab)^+$. Patterns $aa(a)^+$ are replaced by aa , and $ab(ab)^+$ by ab . Figure 5 shows an example for each kind of pattern, and it can be observed that the final geometry of strips does not change after removing these patterns.

3.3 GPU Data Structures

Two essential data structures for the performance of the model are stored in the GPU: vertices and strips, which constitute the polygonal mesh. On the one hand, vertices are stored in a vertex array buffer. On the other hand, we might allocate each strip in an element buffer. However, we have observed that creating as many buffers as strips leads to noticeable decreases in performance due to bind operations. A solution to this problem, with optimum results, consists in creating a single element buffer, where every strip to be rendered is located. In this way, we avoid the need for continuous bind operations to assign an element buffer for each strip.

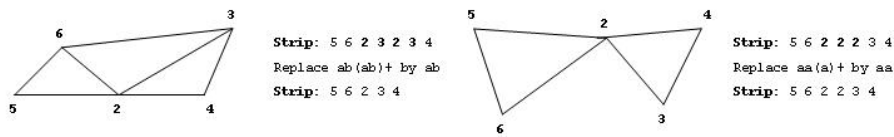


Fig. 5. Removed patterns

3.4 Controlling Level of Detail

In continuous multiresolution models, level of detail management entails two fundamental tasks: level of detail extraction required by applications and visualization of resulting geometry.

Level of Detail Extraction. At a high level, the pseudo algorithm for moving from LOD n to LOD $n+1$ would consist in downloading, from the GPU, the chunks of memory corresponding to the strips affected by the change in the level of detail. After that, we replace vertex n by the vertex it collapses to, in every strip where it appears. Later, derived vertex repetitions must be removed. Finally, the strip is uploaded to the GPU for visualization. Figure 6 shows the algorithm.

```

for LOD = currentLOD to demandedLOD
  for Strip = StripsAffected(LOD).Begin() to StripsAffected(LOD).End()
    auxStrip=DownloadFromGPU(Strip);
    CollapseOrSplit(auxStrip,LOD);
    UploadToGPU(auxStrip);
  end for
end for

```

Fig. 6. Level of detail extraction from a LOD to a coarse one

Visualization. Figure 7 shows the visualization algorithm. This algorithm takes advantage of the capabilities of the latest GPU capabilities. It stores and manages strips to be visualized directly from the graphics hardware memory.

```

for IndexStrip = 0 to NumberOfStrips - 1
  glDrawRangeElements (
    GL_TRIANGLE_STRIP,
    currentLOD,
    NumberOfVertices - 1,
    StripBufferManager(IndexStrip).size(),
    GL_UNSIGNED_INT,
    (const void*)(StripBufferManager(IndexStrip).Offset()*sizeof(EnteroUn)),
  end for

```

Fig. 7. Visualization algorithm

4 Results


Figure 8 shows a comparison of spatial costs. On average, the model presented in this paper fits in 1.5 times the original mesh in triangles and 2.3 times in triangle strips.

Two well-known utilities to generate strips were tested in this multiresolution model: Stripe Utility [14] and NVTriStrip Library [12]. Triangle strips for different objects were generated using both utilities. The model generated from the NVTriStrip Library shows better frame-per-second rates than the Stripe object when the level of detail is higher; this behavior is shown in Figure 9(right).

Results of visualization are shown in Figure 9(left), where our approach is compared to other models. It can be seen that our model offers the best visualization times due to its being integrated into the hardware.

5 Conclusions

We have presented a uniform resolution model that noticeably improves existing models in terms of both storage and visualization cost. This model features: total graphics hardware integration with implementation in high-performance memory, optimized hardware primitives, vertex cache exploitation and low spatial cost.



	Cow	AI	Bunny	Panther	Dragon	Phone	Buddha
Vertices	2904	3618	34834	38911	54294	83044	543699
Faces	5804	7124	69451	69397	108588	165963	1085634
Size Tris kb	113.4	140.0	1358.2	1421.2	2120.9	3242.5	21217.6
Size Strips kb	73.5	91.4	867.6	971.1	1387.6	1999.5	14107.9
Model Cost Mb	0.16	0.20	2.07	2.00	3.59	4.71	33.53
Ratio Triangles	1.5	1.5	1.6	1.4	1.7	1.5	1.6
Ratio Strips	2.3	2.2	2.4	2.1	2.7	2.4	2.4

Fig. 8. Spatial cost comparison

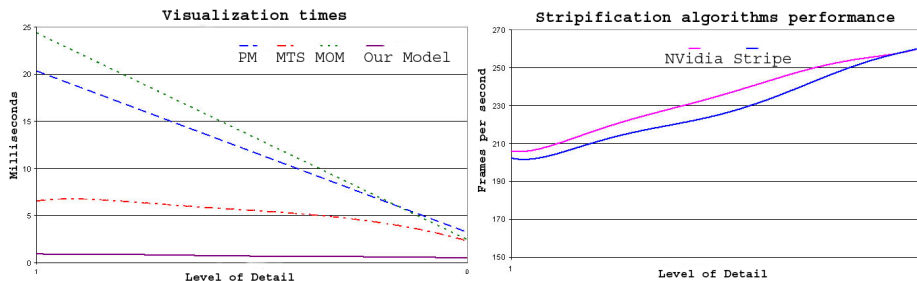


Fig. 9. Results obtained from the bunny object. On the left, multiresolution models comparison of PM[1], MTS[5], MOM[6] and our model. On the right, stripification techniques performance in our approach

Bibliography

- [1] Hoppe H.: Progressive Meshes. *Computer Graphics (SIGGRAPH)*, 30:99-108, 1996.
- [2] El-Sana J., Azanli E., Varshney A.: Skip strips: maintaining triangle strips for view-dependent rendering. In: *Proceedings of Visualization 99*, 1999. p.131-137.
- [3] Shafae M., Pajarola R.: DStrips. *Dynamic Triangle Strips for Real-Time Mesh Simplification and Rendering*. *Proceedings Pacific Graphics Conference*, 2003.
- [4] A. James Stewart: Tunneling for Triangle Strips in Continuous Level-of-Detail Meshes. *Graphics Interface 2001*: 91-100.
- [5] Belmonte O., Remolar I., Ribelles J., Chover M., Fernandez M.: Efficient Use Connectivity Information between Triangles in a Mesh for Real-Time Rendering, *Future Generation Computer Systems*, Special issue on Computer Graphics and Geometric Modeling, 2003. ISSN 0167-739X.
- [6] Ribelles J., Lopez A., Remolar I., Belmonte O., Chover M.: Multiresolution Modeling of Polygonal Surface Meshes Using Triangle Fans. *Proc. of 9th DGC 2000*, 431-442, 2000. ISBN 3-540-41396-0.
- [7] Ramos J.F., Chover M.: LodStrips. *Level of Detail Strips*, Lecture notes in Computer Science, *Proc. of Computational Science ICCS 2004*, Springer, ISBN/ISSN 3-540-22129-8, Krakow (Poland), vol. 3039, pp. 107-114, June, 2004.
- [8] Olano, Marc, Kuehne B., Simmons M.: Automatic Shader Level of Detail. *Proceedings of Graphics Hardware 2003*, Eurographics/ACM SIGGRAPH, July 2003.
- [9] Gain J., Southern R.: Creation and Control of Real-time Continuous Level of Detail on Programmable Graphics Hardware. *Computer Graphics Forum*, March 2003
- [10] Garland M., Heckbert P.: Surface simplification using quadric error metrics. In *Proceedings of SIGGRAPH '97* (Los Angeles, CA), *Computer Graphics Proceedings, Annual Conference Series*, pages 209 - 216. ACM SIGGRAPH, ACM Press, August 1997.
- [11] Luebke P.: A Developer's Survey of Polygonal Simplification Algorithms, *IEEE CGA*, June, 2001
- [12] NvTriStrip Library, NVIDIA Corporation (2002). Available in Internet at following URL http://developer.nvidia.com/object/nvtristrip_library.html.
- [13] ARB_vertex_buffer_object Specification. http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_buffer_object.txt
- [14] Evans F., Skiena S., Varshney A.: Optimising Triangle Strips for Fast Rendering, *IEEE Visualization '96*, 319-326, 1996. <http://www.cs.sunysb.edu/stripe>