EFFICIENT IMPLEMENTATION OF LODSTRIPS

Francisco Ramos, Miguel Chover, Oscar Ripolles, Carlos Granell
Departamento de Lenguajes y Sistemas Informáticos
Universitat Jaume I, Campus de Riu Sec, 12071, Castellón, Spain
{Francisco.Ramos, chover, oripolle, Carlos.Granell}@uji.es

ABSTRACT
In earlier work, we introduced LodStrips, a new multiresolution model for fast visualisation of polygonal meshes. This model was based on triangle strips and it defined a continuous sequence of level-of-detail managed on demand. In this paper, we present new data structures and algorithms for efficient implementation of the LodStrips model and its applications. This implementation is based on GPU characteristics and results demonstrate noticeable improvements in spatial and rendering costs.

KEY WORDS
Real-time graphics, 3D modelling, level of detail, multiresolution, triangle strips.

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 modelling by level of detail have been developed that attempt to adapt the number of polygons of the objects to their importance within the scene. According to [1], we can distinguish between discrete and continuous models. Discrete models typically store a few LODs and tend to suffer from popping artifacts when a LOD is changed. Continuous multiresolution models are more exact and two consecutive LODs differ by only a few triangles.

In previous work [2], we introduced a new continuous multiresolution model that improved existing solutions. LodStrips consists of an arbitrary mesh M^n and a set of *n* records that indicate how to simplify it. Each record encodes the information that enable us to modify the mesh according to the LOD required.

Following the same philosophy of the original paper [2], we present new data structures and algorithms allowing the efficient implementation of LodStrips. This implementation is adapted to the graphics hardware and it reduces the spatial cost and accelerates rendering. Results show the performance with some models and in applications, as shown in figure 1 and figure 2.

The paper is organized as follows. In section 2, we revise some related work. The LodStrips model is reviewed in section 3. Section 4 describes the technical background of the model and section 5 our basic data structures. Section 6 explains the process of traversing the levels of detail in a LodStrips mesh. Finally, section 7 shows implementation results and section 8 summarizes the paper.

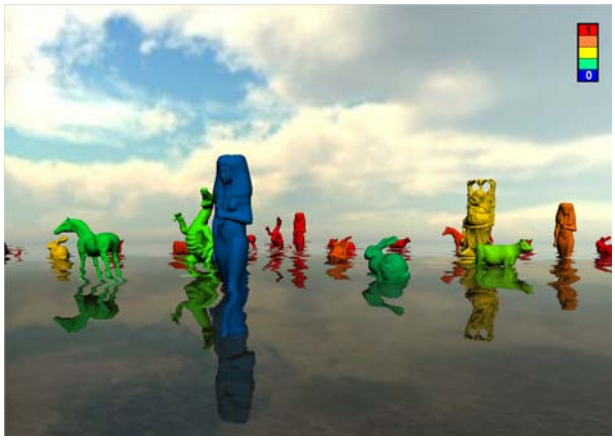


Figure 1. Efficient implementation of LodStrips in a computer game engine: Ogre3D.

2. 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 makes 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.

The best known continuous multiresolution model is Progressive Meshes [3], 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 and triangle fans, have been developed. Many continuous models based on this type of primitives have been recently developed [2][4-7].
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[12][13], and the new extensions of graphics libraries [14] that allow visualization of a whole mesh with only a few instructions are examples of these new techniques.

3. Review of LodStrips

This multiresolution model represents a mesh as a set of multiresolution strips. Let M be the original polygonal surface and M^r its multiresolution representation. M^r can be defined as:

$$M^r = \{ V, S \} \qquad (1)$$

where V is the set of all the vertices and S the triangle strips used to represent any resolution. We can express S as a tuple $\{S_0, C\}$, where S_0 consists of the set of triangle strips at the highest level of detail and C is the set of simplifications required to refine them.

$$S_0 \xrightarrow{C_0} S_1 \xrightarrow{C_1} \ldots \xrightarrow{C_{n-2}} S_{n-1} \qquad (2)$$

In particular, we will consider every simplification in its minimal expression, that is, one simplification implies a vertex collapse. Moreover, we arrange simplifications in such manner that simplification *i* means collapsing the vertex *i*. Thus, to build *S* with its *n* levels of detail, we apply *n-1* iterations of a progressive simplification method. Each simplification C_i generates a new level of detail S_{i+1}, where $0<i<n-1$ and it may be represent by the tuple $\{vf'_i, S'_i\}$ where vf'_i is the vertex where collapses the vertex *i*, and S'_i is the subset of strips modified by this collapse.

4. Technical Background

Various algorithms were used to build the model: an algorithm to obtain a sequence of edge-collapses from the highest level of detail to the lowest and an algorithm to construct efficient triangle strips.

In order to obtain a sequence of edge-collapses, we apply the QSLIM algorithm[8], it is based on the iterative contraction of vertex pairs, a generalization of edge contraction. It produces simplified versions of polygonal models. In this manner, this information is stored into the model data structures, as shown in section 4, allowing transitions among levels of detail.

The model presented is wholly based on triangle strips, which are generated from a polygonal model at the highest level of detail. NvTriStrip [9] utility is used with the purpose of maximizing vertex reuse when rendering a mesh, but it has a limitation of indices at 16 bits; hence Stripe utility [10] is applied to large models.

Information about simplification and triangle strips at the highest level of detail is saved into the model data structures. Later, information about changes to manage level of detail, is also calculated and stored.

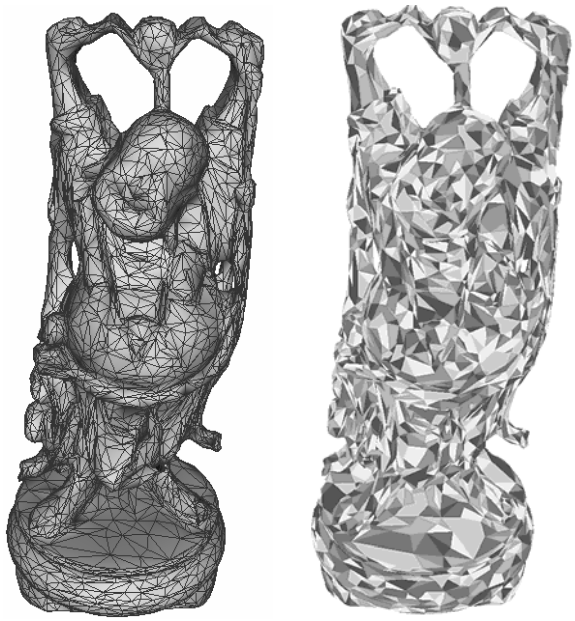


Figure 2. Buda model. At the highest level of detail: 543644 vertices and 31596 triangle strips. a) The lowest level of detail: 5438 vertices. b) Strips at the lowest level of detail.

5. Data Structures

To display a polygonal mesh at the highest level of detail only two basic data structures are required: *Vertices* and *Strips*. *Vertices* stores the 3D coordinates for each vertex on the mesh and *Strips* stores the mentioned mesh , that is, a set of strips, where each strip contains a set of indices to the vertex, in *Vertices*, which refers to.

To change the level of detail, we also need to store the vertex that will be collapsed for each LOD. That information is obtained from a pre-process. Thus, for each vertex in the *Vertices* data structure we also store the vertex where the collapse will take place.

Moreover, in order to avoid the problem of having to search the vertex to be collapsed in each strip, we first store the strip that changes in the data structures and then store the exact position of the vertex to be collapsed in that triangle strip.

However, an accumulation of identical vertices is produced as the model moves towards coarser LODs. Sending these vertex repetitions to the graphics hardware does not contribute to the final scene at all. This problem is solved by detecting those patterns before rendering, avoiding the application of filters in visualisation. Thus, we have proven that most vertex repetitions follow patterns like *aa(a)+* or *ab(ab)+*, where *a* and *b* are vertices of the model. Patterns *aa(a)+* are replaced by *aa*, and *ab(ab)+* by *ab*.

Figure 3 shows main LodStrips data structures in a C++ implementation.

```
struct Vertex {
          Point point;          //(x,y,z) coordinates
          int Collapse;         //index to the vertex to collapse
};

struct Strip {
          int *indices;         //indices to Vertices
};

struct RecordChange {
          int strip;            //strip affected by one change
          int #collapses;       //number of collapses to apply
          int #resizes;         //number of resizes to apply
          int *data;            //positions, in the strip, where
                                //collapses and resizes take place
};

struct Change {
          int #StripsAffected;   //number of records
          RecordChange *Records; //array of records to apply
};

struct LodStripsMesh {
          Array<Vertex> Vertices;       //vertices of the mesh
          Array<Strip> Strips;          //triangle strips of the
mesh
          Array<Change> Changes;        //Records to change the
mesh
};
```

Figure 3. LodStrips data structures.

In figure 4 we can observe the simple process to fill those data structures. First of all, we have the strips at the highest level of detail and, from the simplification, we know where collapses every vertex. We start from LOD 0 and, we collapse vertex 0 to vertex 3, the result is the strip 3 2 3 1 5 4, which has no repetitions, and therefore, no resizes. Finally, we fill *Change* data structure with 1 strip affected by this collapse and a pointer to the *RecordChange* data structure, which contains the strip affected, in this case 0 and, only 1 collapse and 0 resizes. In the *data* data structure we have the position where the collapse takes place in the strip, that is 0. The process continues in the same fashion though every level of detail. Thus, we fill data structures by collapsing vertices and removing repetitions, as shown in that figure, where we calculate transitions to three levels of detail.

6. Level of Detail Management

Management of level of detail implies two essential tasks in multiresolution modelling: extraction of the LOD required and visualisation of the resultant geometry.

6.1 Updating Geometry

Updating geometry means adapt LodStrips meshes to a desired level of detail of complexity, Figure 5 shows different levels of detail from a model.

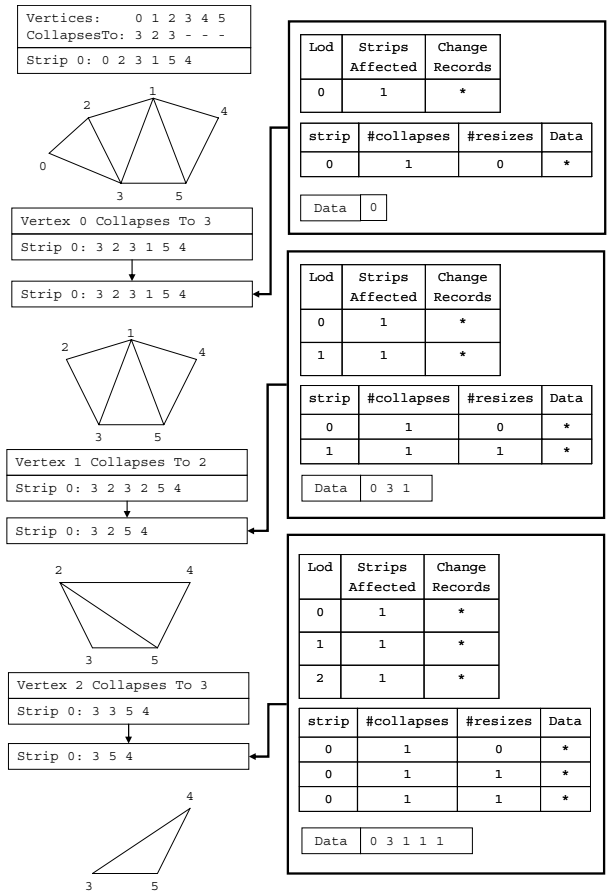


Figure 4. Construction of LodStrips data structures.

6.2 Rendering

In the model, triangle strips are changing as much as change the LOD required by applications. Therefore, a data structure to manage strips, with fast constant time

insertions and deletions is required. On the other hand, access time to these type of data structures is penalized. Our model implements an array of displayed strips, where access is very fast. Thus, if no LOD is required, the mesh is rendered with the highest performance possible, and if a change of LOD is demanded, this array is updated only for those strips modified.

In an implemented view, triangle strips has two representations, on the one hand, *hwStrips*, which contains the strips at the current LOD, and, on the other hand, *disStrips*, which contains the same as *hwStrips* but with a hardware-oriented implementation. Thus, if an application maintains the LOD, *disStrips* offers the maximum performance. When a change of LOD is processed, all display strips affected are updated. It is easily managed by a dirty flag.

LOD 1 LOD 0.5 LOD 0

Figure 5. Three levels of detail from the Al Capone model..

7. Results

This model was submitted to several tests, all of which were aimed at evaluating the rendering time in a real-time application by applying different acceleration techniques. Tests designed to compare multiresolution models follow the ones introduced by [11] and the one carried out in this study was the linear test: this consists in extracting the LODs of the model in a linear and proportionately increasing or decreasing way.

To carry out the tests, three well-known meshes from the Stanford 3D Scanning Repository were taken as a reference, so as to make it easy to compare this model with other well-developed models.

Test were carried out using a PC with an Intel Pentium Xeon 2.8 GHz processor, 1024 Mb RAM and a NVIDIA GeForce FX 6600 256 Mb graphics card. C++ was employed for the implementation, using the graphics library OpenGL, and it is completely portable.

All these tests were carried out in order to evaluate the different acceleration techniques. Initially, utilisation of coherence was evaluated, later; hardware acceleration techniques were compared, by applying the immediate mode, vertex array mode and the OpenGL extension: vertex buffer objects with glDrawRangeElements and glMutiDrawElements. After that, the effect of utilising cache reutilisation techniques was tested with the NvTriStrip library, which creates cache-optimised triangle strips and then compared to the Stripe utility and finally, a comparison to some well-know multiresolution models was obtained and compared.

7.1 Multiresolution Models Comparison

Table 1 shows a comparison of rendering and storage of different well-know multiresolution models, this information has been taken from the implemented models and they are compared to our efficient implementation. We can observe an improvement of around a 30% when compared to the original LodStrips. Moreover, we can observe how rendering is even faster in immediate mode.

Model	Render (ms)	Spatial Cost (Mb)	#ratio
Skip Strips	3.56	4.36	2.83
MTS	4.37	7.55	4.90
LodStrips	2.43	4.30	2.79
Efficient	2.07	3.20	2.08
Vertices+Strips		1.54	1.00

Table 1. Render and spatial cost comparison of multiresolution models (MTS[6], SkipStrips[4] and LodStrips[2]) for the bunny model. Render column consist of the time to extract LODs in a linear way plus the time for visualising the result in OpenGL immediate mode. Ratio column shows how many original meshes fits into the strips model at the highest level of detail..

7.2 Spatial Cost

In table 2 we can observe the spatial cost for some models. It is important to underline the storage obtained with large models.

Model	Vertices	Original Model (Mb)	Spatial Cost (Mb)
Cow	2904	0.13	0.31
Bunny	34834	1.54	3.20
Dragon	54296	2.39	6.01
Buda	543644	24.11	49.83

Table 2.Spatial cost data.

7.3 Stripification Techniques

The NVTriStrip library was unable to generate strips for the happy buddha object. It is due to a limitation of the library, which can not manage objects which exceed 35635 indices. However, by using Stripe, the model can be loaded and managed by the efficient implementation with 30 fps in the worst case, by using vertex buffer objects (extension DrawRangeElements) technique, as shown in figure 6. With MultiDraw extension we obtained 70 fps at the highest level of detail.

7.4 Hardware Acceleration

The results of the hardware acceleration tests are shown in figures 7 and 8; on the left side we can see an image of the object with strips, while on the right side and in the upper part of the table, some data about characteristics of the model are shown. In the lower part the total rendering time is shown first, and after that the table shows the percentage of this time used in extracting the level of detail and in drawing the resulting mesh.

As can be seen in these figures, the performance of the model grows exponentially when a hardware acceleration technique is applied. By applying both acceleration techniques and the immediate mode we obtain a significant improvement in performance.
It is important to underline the suitability of the model for applying hardware acceleration techniques. This model spends a small percentage of time on extracting the level of detail, which leads to good rendering times due to the lower extraction times and, moreover, it benefits the application of those techniques.

8. Conclusions and Future Work

We have described an efficient implementation of the LodStrips model introduced in earlier work. Efficient data structures and algorithms permit fast iteration through the LodStrips approximations.
Spatial cost has been improved, as well as rendering times and the model has been successfully implemented in a computer game engine: Ogre3D. Moreover, the efficiency of the geometric acceleration techniques was tested on this implementation. To verify the increase in performance, a series of tests, besides to some acceleration techniques, were carried out to evaluate the ability of the model to manage the changes in the level of detail.
One of the most important conclusions that must be highlighted is that this implementation shows a total integration with GPU. Hardware acceleration techniques allow us to increase the performance of the models with dynamic geometry. In this sense, the model noticeably increased its performance. This rise is mainly due to the optimised design of the model for the hardware, where level of detail extraction times are very low and so graphic acceleration is greatly benefited.
Moreover, cache reutilisation techniques have shown good rendering times when cache-optimised triangle strips, generated from the NvTriStrip library, have been utilised, although improvements, in this way, are required in order to manage multiresolution schemes.

References

[1] J. Ribelles, A. López, Ó. Belmonte, I. Remolar, M. Chover, Multiresolution modeling of arbitrary polygonal surfaces: a characterization, Computers & Graphics, vol. 26, n.3 2002.
[2] F. Ramos, M. Chover, LodStrips, 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.
[3] Hoppe H. Progressive Meshes. Computer Graphics (SIGGRAPH), v. 30:99-108, 1996.
[4] 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
[5] Michael Shafae, Renato Pajarola. DStrips: Dynamic Triangle Strips for Real-Time Mesh Simplification and Rendering. Proceedings Pacific Graphics Conference, 2003.
[6] O. Belmonte, I. Remolar, J. Ribelles, M. Chover, M. Fernández. 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.
[7] A. James Stewart: Tunneling for Triangle Strips in Continuous Level-of-Detail Meshes. Graphics Interface 2001: 91-100.
[8] M. Garland and P. Heckbert. 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.
[9] NvTriStrip Library, NVIDIA Corporation (2002). Available in Internet at following URL http://developer.nvidia.com/object/ nvtristrip_library.html.
[10] F. Evans, S. Skiena and A. Varshney, Optimising Triangle Strips for Fast Rendering, IEEE Visualization ’96, 319-326, 1996. http://www.cs.sunysb.edu/~stripe.
[11] J. Ribelles, M. Chover, A. Lopez and J. Huerta. A First Step to Evaluate and Compare Multiresolution Models, Short Papers and Demos EUROGRAPHICS’99, 230-232, 1999.
[12] A. Bogomjakov, C. Gostman. Universal Rendering Sequences for Transparent Vertex Caching of Progressive Meshes. Proceedings of Graphics Interface 2001.
[13] H. Hoppe, “Optimization of Mesh Locality for Transparent Vertex Caching”, ACM SIGGRAPH 1999, pp. 269-276.
[14] Silicon Graphics, “Vertex Buffer Object Specification”, 2003, http://oss.sgi.com /projects /ogl-sample /registry /ARB/vertex_buffer_object.txt.

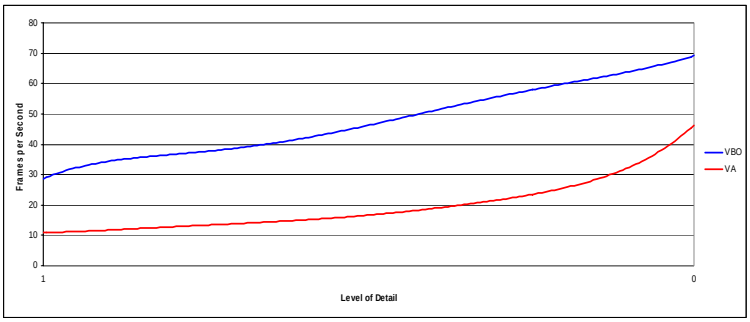
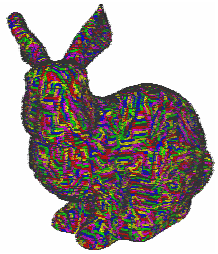
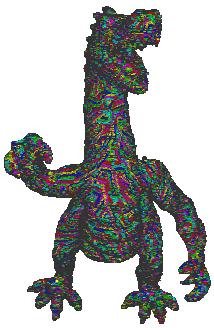


Figure 6. OpenGL DrawRangeElements and vertex array results obtained from the happy buddha object with strips generated from the Stripe utility.



Bunny object			
Vertices 34834	Strips 6194	Storage 3.9 Mb	
Test	**Hardware Technique**	**Render (ms)**	
		% rec	**% drw**
Linear Test	Immediate Mode	2.064565	
		0.19	99.81
	VBO: DrawRange	0.519380	
		0.68	99.32
	VBO: MultiDraw	0.250492	
		1.26	98.74

Figure 7. Results obtained from the bunny model by applying hardware acceleration techniques.



Dragon object			
Vertices 54296	Strips 8799	Storage 6.0 Mb	
Test	**Hardware Technique**	**Render (ms)**	
		% rec	**% drw**
Linear Test	Immediate Mode	3.638561	
		0.11	99.89
	VBO: DrawRange	0.734888	
		0.52	99.48
	VBO: MultiDraw	0.370881	
		1.03	98.97

Figure 8. Results obtained from the dragon model by applying hardware acceleration techniques.