EG UK Theory and Practice of Computer Graphics (2010)
John Collomosse, Ian Grimstead (Editors)

An Improved Discrete Level of Detail Model Through an Incremental Representation

J. Ribelles[1,3], A. López[2,3] and O. Belmonte[1,3]

[1]Departamento de Lenguajes y Sistemas Informáticos
[2]Departamento de Ingeniería y Ciencia de los Computadores
[3]Instituto de Nuevas Tecnologías de la Imagen
Universitat Jaume I, Castellón, Spain

Abstract

Real-time applications such as computer and video games, virtual reality and scientific simulation require rendering of complex models for realism. Graphics rendering engines include multiresolution modelling techniques to accelerate the visualization process. The Discrete Level of Detail framework (DLoD) is usually the most popular while the Continuous Level of Detail framework (CLoD) is still not as widely used by software developers. In this paper, we first discuss the benefits and drawbacks of both frameworks. Then, we present a model based on coding a discrete number of levels of detail (LoDs), with more LoDs coded than is usual in DLoD, and with an incremental representation, which is often used in CLoD. This model obtains a performance similar to DLoD by providing optimized LoDs for efficient visualization, while the popping effect is imperceptible. We present specific proposals for each of the three main stages involved in multiresolution processing: geometry simplification, construction of the incremental representation and retrieval of either uniform or view-dependent LoDs.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling

1. Introduction

Multiresolution modelling has been a matter of growing interest in the last decade. One of its main goals is to accelerate the visualization process [LRC*03]. Multiresolution modelling allows for adjusting the level of detail (LoD) of the scene so maintaining a constant frame rate and assuring interactivity to the final user. Nowadays, real-time applications such as computer and video games, virtual reality or scientific simulation are among the most demanding and tightly optimized graphics applications. As these applications require rendering of complex models for realism, graphics rendering engines include multiresolution modelling techniques, which have become widely used.

The multiresolution modelling techniques presented in the literature are classified under two main frameworks for managing level of detail: Discrete LoD (DLoD) and Continuous LoD (CLoD). DLoD is the most widely used. This framework manages a small number of independent levels of detail (LoDs), where each approximation or LoD represents the original object using a different number of faces. CLoD is introduced as an alternative which provides a wide range (virtually a continuous range [PS97]) of different approximations, such that the LoD can be adapted to the application requirements with a high degree of accuracy. CLoD has been extended to provide view-dependent LoDs, which is sometimes considered as a third framework [LRC*03].

In order to introduce our proposal we first compare DLoD and CLoD to understand their benefits and drawbacks. We have arranged a series of processing stages (figure 1):

Off-line process

- Simplification. The object is processed using a simplification tool. In case of DLoD, this process gives the LoDs separately. In case of CLoD, this process gives the se-

DOI: 10.2312/LocalChapterEvents/TPCG/TPCG10/059-066

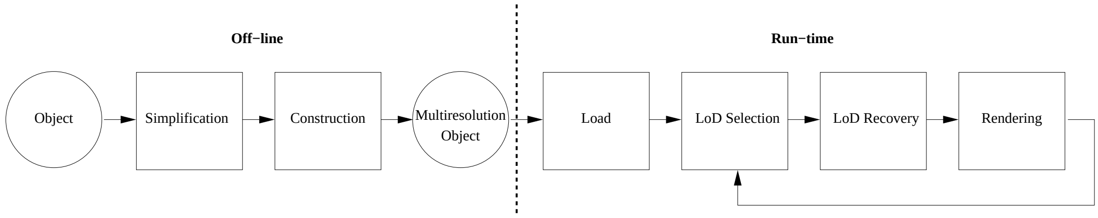


Figure 1: *Order of operations.*

quence of removals necessary to construct a multiresolution representation that provides a wide range of LoDs.

- Construction.

  DLoD. This stage consists of only processing each LoD for maximum efficiency in the rendering process by using features depending on the particular hardware targeted, i.e. triangle strips. After completion, standard file formats can be used to store data for each LoD. Consequently, LoDs can be easily used in a wide range of graphic engines.

  CLoD. First, a process constructs the multiresolution representation according to the CLoD model to be used. Second, data is processed to take advantage of hardware rendering features. After completion, data is stored in a proper file format, which is generally non standard.

At run-time

- Load.

  DLoD. LoDs are stored in main memory through commonly used data structures and they are easily compiled into hardware command streams as static buffer objects.

  CLoD. This framework requires special data structures to arrange data in such a way that LoDs can be retrieved as efficient as possible. These structures differ depending on the particular CLoD model. As in DLoD, data are compiled into hardware command streams.

- LoD Selection. An algorithm selects the most appropriate level of detail to display. This stage is analogous for both frameworks.
- LoD Recovery.

  DLoD. As the LoD has been optimized in the off-line process and compiled for efficient rendering in the load stage, no more process is required. However, view-dependent LoDs cannot be retrieved.

  CLoD. An algorithm traverses the data structures to recover either uniform or view-dependent LoDs. In order to use the current hardware capabilities an optimization process must be done, thus incurring in an overload, which is even higher in case of view-dependent LoDs.

- Rendering.

  DLoD. This framework displays more or less detail than actually required, and presents visual discontinuities when changing between LoDs, known as the popping effect. However, data is provided to GPU highly optimized and maximum performance is obtained.

  CLoD. This framework displays the level of detail just required. Popping effect is not perceptible by the user as the difference between consecutive LoDs is very small. However, maximum performance is hard to achieve because LoD optimization is computed at runtime and it is not as good as off-line optimization.

We can conclude that DLoD presents drawbacks as the popping effect and the impossibility of providing view dependent LoDs, and CLoD requires a higher effort in almost every stage and does not reach the performance of DLoD. Furthermore, as graphics hardware evolve, changes or advent of new features require minor changes in DLoD, but major changes, even redesign, in CLoD.

In this paper, we aim to obtain a performance similar to DLoD by providing optimized LoDs for efficient rendering, while the popping effect is imperceptible. We present an improved DLoD (iDLoD) based on the following features: (1) to increase the number of LoDs, such that it depends on the fame rate of the application (which is a constant) and it is independent on the complexity of the object; (2) to encode LoDs using an incremental representation to reduce the storage cost, so that consecutive LoDs share a high number of faces; and (3) the difference between two consecutive LoDs is forced to be a surface patch (see figure 2), so that it can be optimized off-line for rendering.

Consequently, a LoD is a reduced set of surface patches, each of which is optimized off-line. At the load stage, data are compiled into static hardware command streams. The complexity of the algorithm for LoD recovery depends on the number of LoDs, which is a constant instead of depending on the size of the object. The provided LoDs fit the required level of detail better than DLoD, thus avoiding the popping effect. Furthermore, in spite of the low granularity, iDLoD can provide view-dependent LoDs.

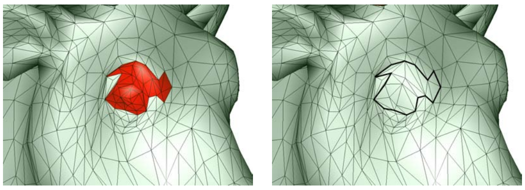


Figure 2: *The left image shows in red the surface patch (61 triangles) selected for simplification at the current LoD. The right image shows the LoD, which is generated by substituting the red triangles by a simplified surface (24 triangles). There is a difference of 37 triangles between these two consecutive LoDs.*

2. Previous work

The original DLoD was presented by Clark in 1976 [Cla76]. This technique began to be used with the main aim of increasing the performance of the graphic system and this was accomplished in applications such as walkthroughs in virtual environments. Recent work has addressed the problem of the popping effect performing smooth transitions between them by means of geo-morphing or blending [SG03, BGB*05]. Numerous methods for CLoD have been presented in the last years. For an overview on the various schemes proposed see [Gar99] [RLB*02].

El-Sana et al. [ESAV99] presented the Skip Strips model, which maintains a data structure to store the strips that avoids the need to calculate them in real-time. The MTS model [BRR*04] uses triangle strips as both the storage and the visualization primitive. It consists of a set of multiresolution strips, each of which represents a triangle strip and all its levels of detail. LodStrips model [RCRG06] is entirely based on optimized hardware primitives, triangle strips, and deals with the apparition of degenerate triangles by applying pre-calculated filters.

Ji et al. [JWLL05] suggest a method to select and display several LoDs by using the GPU. In particular, they encode the geometry in a quadtree based on a LoD atlas texture. Masking Strips [RCG*09] uses a cache-aware stripification technique to diminish bus traffic and also to apply LoD update routines which remove all the unnecessary degenerate triangles.

Many of the GPU-based continuous models are aimed at view-dependent rendering of massive models. Researchers have recently proposed methods for moving the granularity of the representation from triangles to triangle patches in order to offer view-dependent capabilities for rendering out-of-core models [CGG*04, SM05]. With a similar objective but with a further GPU exploitation, the GoLD method [BGB*05] introduces a hierarchy of geometric patches for very detailed meshes with high resolution textures. Recently, Hu et al. [HSH09] presented a fully-GPU implementation of Progressive Meshes [Hop97].

3. Heuristic determination of the number of LoDs

In the context of real-time applications, an object that is far away from the viewer will be represented by the lowest level of detail. As the distance decreases the level of detail increases. Despite thousands of LoDs are available, the application usually requires one LoD per frame at most. Let us consider a frame rate of 50 images per second. If a distant object gets very near of the viewer one second later, for instance, only 50 LoDs would be displayed. In case the distant object gets slowly near to the viewer, ten seconds later for example, the application would require 500 LoDs at most when using the same frame rate. However, we can suspect in these cases that the user is not focusing his attention on this object. This perceptual factor is often used to extend the life of a LoD over a number of frames [LRC*03]. In case the application decides to maintain the LoD during 5 frames, only 100 LoDs would be displayed, just two times the frame rate. In case of maintaining the LoD during only 2 frames, 250 LoDs would be enough, just 5 times the frame rate. In any case, the amount of displayed LoDs is very far from the granularity that CLoD provides.

Therefore, we propose the number of LoDs to be a multiple of the frame rate. Experiments have been carried out using a factor of 2, and three different frame rates: 25, 50 and 70 images per second. Thus each object stores 50, 100 and 140 different LoDs.

4. Simplification

A simplification process converts a polygonal surface into another surface with a smaller number of polygons. As in our multiresolution model we need a sequence of simplified surface patches, methods based on vertex clustering or merging regions are well suited [Gar99]. However, we used the method proposed by Garland and Heckbert [GH97], which is based on edge collapse, plus a merging process to form surface patches. There is a public domain implementation of the simplification method of Garland and Heckbert so-called *Qslim* [Gar04], and we use it due to its speed and the quality of the generated meshes.

Our proposal is based on ***regions*** that are created and merged until they reach a given simplification error, E. For each edge collapse in the sequence provided by *Qslim*, a new region is created and checked to be merged with any of the previously created regions. Each region is assigned a simplification error, which is the sum of the simplification errors of all the edge collapses grouped for that region. When the simplification error of a region reaches E, the region is not further considered for merging.

Let us consider an edge collapse as the removal and creation of triangles in the polygonal surface. For example, the collapse of the edge between triangles a and j in figure 3(a) and 3(b) would remove triangles $\{a,b,c,d,i,j\}$ and would

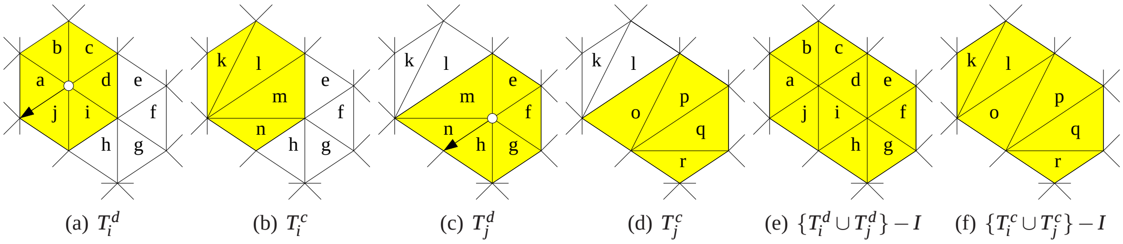


Figure 3: *Creating and merging regions. First, a region R_i is created, (a) $T_i^d = \{a,b,c,d,i,j\}$, (b) $T_i^c = \{k,l,m,n\}$. Second, a new region R_j is created, (c) $T_j^d = \{e,f,g,h,m,n\}$, (d) $T_j^c = \{o,p,q,r\}$. As $I = \{T_i^c \cap T_j^d\} = \{m,n\}$, then $R_k = merge(R_i,R_j)$, (e) $T_k^d = \{T_i^d \cup T_j^d\} - I = \{a,b,c,d,e,f,g,h,i,j\}$, (f) $T_k^c = \{T_i^c \cup T_j^c\} - I = \{k,l,o,p,q,r\}$.*

create triangles $\{k,l,m,n\}$. For each edge collapse, we define a region, R_i, as the tuple $\{T_i^d, T_i^c\}$ where T_i^d is the set of removed triangles (figure 3(a)) and T_i^c is the set of created triangles (figure 3(b)). Once the new region is created, we check the previously created regions to find those regions suitable to be merged with it. So, given two regions, R_i and R_j, where region R_i was previously created and R_j is a new region, we define the *check* function as:

$$I = \text{check}(R_i, R_j) = \{T_i^c \cap T_j^d\}$$

If $I \neq \emptyset$ then R_i and R_j are merged. That is, in case there are some triangles created by region R_i that are deleted by region R_j, then both regions must be merged. In the example of figure 3, first region R_i was created (figures 3(a), 3(b)), then a new region R_j is created from the collapse of edge between faces n and h (figures 3(c), 3(d)). As the intersection between T_i^c and T_j^d is $\{m,n\}$, both regions must be merged. We define the *merge* function as follows:

$$R_k = merge(R_i, R_j) = \{\{T_i^d \cup T_j^d\} - I, \{T_i^c \cup T_j^c\} - I\}$$

Figures 3(e) and 3(f) show the new T_k^d and T_k^c sets of region R_k as a result of the *merge* function. And the simplification error of the resulting region is the sum of the simplification errors of both regions. When a region reaches E, it is not further considered for the merging process. These completed regions are stored in a result list so that, at the end of the process, the list contains the sequence of regions in the completion order. Finally, for each region R_i in the result list, a LoD of the multiresolution representation is established, that is, $M_i - T_i^d + T_i^c = M_{i-1}$. Figure 4 outlines the data structures and the algorithm used for this process.

```
class Region {
  vector<int> DeletedFaces
  vector<int> CreatedFaces
  double error
}
list <Region> result
list <Region> temp

for each edge collapse e {
  construct new region R(e)
  for each region Ri in temp  {
    if (check(Ri, R)≠ ∅) {
      R= merge (Ri, R)
      R.error = R.error + Ri.error
      temp.remove(Ri)
    }
  }
  if (R.error > E)
    result.push_back(R)
  else
    temp.push_back(R)
}
```

Figure 4: *Algorithm for creating regions. The* `result` *list finally contains the sequence of regions in order of creation.*

Let $M = M_n$ be the original mesh. Let $\{R_n, R_{n-1}, \ldots, R_1\}$, where $R_i = \{T_i^d, T_i^c\}$, the regions obtained in the simplification process. We construct each LoD as follows:

$$\begin{aligned} M_n - T_n^d + T_n^c &= M_{n-1} \\ &\cdots \\ M_1 - T_1^d + T_1^c &= M_0 \end{aligned}$$

Instead of storing every LoD independently, an incremental representation stores faces that compose the lower detailed LoD, T_0, plus the sequence of updates that allow to construct every LoD, which in our scheme are determined by the regions obtained in the simplification process.

$$M_r = \{\{T_n^d, T_n^c\}, \{T_{n-1}^d, T_{n-1}^c\}, \ldots, \{T_1^d, T_1^c\}, T_0\} \quad (1)$$

5. Construction

One of the characteristics that a multiresolution model for real-time applications should fulfil is efficient information processing [RLB*02]. That is, if the multiresolution representation stores n different LoDs, then the information should be organized in such a way that, during execution, when any of the n LoDs is requested, the retrieval algorithm should extract data as fast as possible.

This representation would store many faces twice because a face created at T_i^c most probably is deleted in some T_j^d with $i > j$, but it can also remain until the lower detailed LoD (it belongs to T_0). Therefore, we can rewrite M_r as a sequence of removed faces plus T_0, that is:

$$M_r = \{T_n^d, T_{n-1}^d, \ldots, T_1^d, T_0\} \quad (2)$$

Each face is assigned a label which identifies the LoD where the face appears for the first time in the sequence $\{M_n, T_n^c, \ldots, T_1^c\}$. Faces that belong to M_n are assigned the label n, those that belong to T_n^c are assigned the label $n-1$, and so on, until those that belong to T_1^c, which are assigned the label zero. Consequently, each set T_i^d probably contains triangles with different labels. Then, faces in each set T_i^d are clustered depending on their label, so that the label is assigned to each cluster. Therefore, each set T_i^d is composed of a set of labelled clusters, $T_{i_l}^d$. For a required LoD, M_i, a $T_{j_l}^d$, with $i \geq j$, belongs to M_i only when $l \geq i$. Finally, in order to take advantage of graphics hardware features, each labelled cluster is processed to obtain triangle strips.

The basic data structure (figure 5) is composed of two vectors: one contains the vertices, and the other contains the sets of deleted faces plus T_0 (equation 2), that is, one set per LoD. Each set consists of an ordered vector of clusters. The number of clusters in each set T_i^d depends on the number of labels in the set. Each cluster is composed of its label l plus the set $T_{i_l}^d$ previously stripified. To speed up the recovery process, the clusters are stored in decreasing order of l.

```
class Vertex {float x, y, z}
class TriangleStrip {vector <uint> indices}
class Cluster {
  vector<TriangleStrip> triangleStrips
  int label
}
class Set {vector<Cluster> clusters}
class Mr {
  vector<Vertex> vertices
  vector<Set> sets
}
```

Figure 5: *Basic data structure to represent M_r.*

6. Uniform LoD recovery

The algorithm to retrieve a given LoD is very simple. Let M_i, $n \geq i \geq 0$, the required LoD to be rendered. As sets T_j^d, with $n \geq j > i$, are the sets of previously deleted faces, none of these faces belongs to M_i. So, the algorithm traverses the data structure starting from T_i^d until T_0. Then, for each set T_j^d, $i \geq j \geq 0$, the clusters, $T_{j_l}^d$, such that $l \geq i$, belong to M_i. Let us remark that the algorithm does not need to check every triangle, only the label of each cluster. Also the ordering of clusters allows to speed up the process. The algorithm is shown in figure 6.

```
int First= sets.length()-i-1
int Last= sets.length()
for (j= First; j< Last; j++) {
  int nClusters= sets[j].clusters.length()
  int k= 0
  while ((k< nClusters) and
         (sets[j].clusters[k].label ≥ i)) {
    Draw(sets[j].clusters[k])
    k= k +1
  }
}
```

Figure 6: *Algorithm to render a uniform LoD M_i, $n \geq i \geq 0$.*

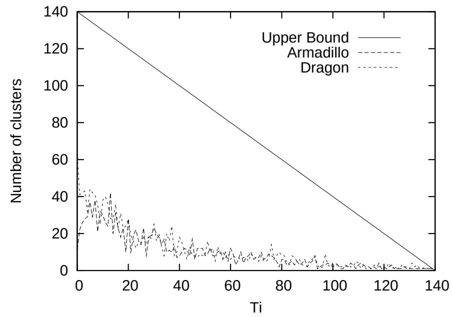


Figure 7: *Number of clusters per set T_i^d, $0 \leq i \leq 140$.*

In order to analyze the efficiency of this algorithm, we must analyze the number of clusters per set, the total number of clusters in M_r, and the total number of clusters that form a given LoD, M_i. First, let us analyze how many clusters, $T_{i_l}^d$, are there for each set T_i^d. Let L_i be the number of clusters in the set T_i^d. Theoretically, the upper bound of L_i is $n+1-i$. The experimental measurements (figure 7) show that L_i does not depend on the size of the object as expected. The variation of L_i is very similar for all the objects and, in all cases, the values of L_i are much less than the theoretical upper bound.

Second, the sum of every L_i is bounded by $n^2/2 + n/2$. Consequently, the maximum theoretical number of stored clusters is independent of the size of the object. Experimental results show that the total number of clusters is around $10n$ for the tested objects (see table 1).

Third, let us analyze how many clusters form one LoD, M_i. The theoretical upper bound of the number of clusters that form M_i is $i*(n+1-i)$, where $n \geq i \geq 0$. As n depends on the desired frame rate, this amount is independent of the size of the object. Figure 8 shows the number of clusters per LoD of several objects, and the theoretical upper bound. The theoretical and experimental values coincide approximately in M_n and M_0, but they are very different for the rest of LoDs. We can also observe that, on average, the maximum number of clusters that form a LoD is around $5n$, which is very far

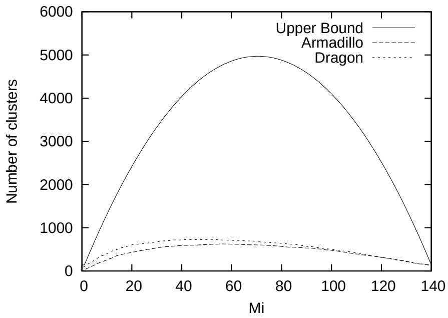


Figure 8: *Number of clusters for each M_i, $0 \leq i \leq 140$.*

from the theoretical curve. Although the maximum differences between the objects are located around intermediate LoDs, they are almost insignificant, specially if we take into account, for example, that the dragon model has near 2 times the number of polygons of the armadillo model.

```
drawCluster (int i, vector<int> Labels) {
  int nClusters= sets[i].clusters.length()
  for (j= 0; j< nClusters; j++)
    if (sets[i].clusters[j].label ∉ Labels)
      Draw(sets[i].clusters[j])
}

checkDependence (int i, vector<int> Labels) {
  int nClusters= sets[i].clusters.length()
  int k= 0
  while ((k< nClusters) and
         (sets[i].clusters[j].label ∉ Labels))
      k= k + 1
  return (k< nClusters)
}

main () {
  vector<int> Labels= ∅
  int First= 0
  int Last= sets.length()
  for (i= First; i< Last; i++) {
    bool pDetail= preserveDetail(i)
    bool dependence= checkDependence(i,Labels)
    if ((pDetail is True) or
         (dependence is True)) {
      drawCluster(i,Labels)
      Labels.pushback(n-1-i)
    }
  }
}
```

Figure 9: *Algorithm to render a view-dependent LoD.*

7. View-Dependent LoD recovery

The aim is to represent an object with several levels of detail coexisting along the surface. The decision about which areas of the surface should be visualized in high or low detail depends exclusively on the criterion or set of critera required by the application, for example, local illumination, view frustum, silhouette and so on [Hop97].

The algorithm to retrieve a view-dependent LoD evaluates each set T_i^d in order to decide whether its faces must be preserved or removed in the required LoD, by using a function that evaluates the criterion defined by the application. Therefore, the algorithm traverses the whole data structure starting from T_n^d until T_0 (see equation 2). Let us note that, although this might seem a drawback, the vector length is independent of the object complexity, and the experimental results show it is around $10n$. If the evaluation function finds that some face in T_i^d must be preserved, all faces in T_i^d are preserved. That is, the simplification coded by T_i^d is not performed. Therefore, none of the faces in T_i^c can belong to the required LoD, because these faces are created only if the simplification coded by T_i^d is performed. Then, if one set T_j^d, $j \neq i$, contains any of the faces in T_i^c, the simplification coded by T_j^d can not be performed, as it depends on the simplification coded by T_i^d. Consequently, faces in T_j^d not belonging to T_i^c are preserved too. Actually, only sets T_j^d with $i > j$ can suffer this dependence and, in case the dependence ocurrs, T_j^d also generates new dependences with subsequent sets. As faces in T_j^d have been clustered and labelled, and each label indicates the LoD of creation, none of the faces in cluster $T_{j_l}^d$ with $l \neq i-1$ belong to T_i^c, only faces in $T_{j_{i-1}}^d$ do. Therefore, for checking dependence between T_i^d and T_j^d, $i > j$, it is enough to check if any of the labels in T_j^d is equal to $i-1$. The algorithm is shown in figure 9. In summary, each set T_i^d, $n \geq i \geq 0$, is preserved in any of these cases:

- The evaluation function decides to preserve detail.
- There exists a dependence: there is one set T_k^d, $k > i$, that was preserved and there is a cluster $T_{i_l}^d$, such that $l = k-1$.

This process uses the same data structure shown in figure 5. However, the algorithm uses a vector of labels to store those which produce dependences in the recovered LoD. The length of this vector is $n+1$ in the worst case, where n is a constant multiple of the frame rate. The algorithm traverses the *sets* vector entirely. As the length of this vector is small (50, 100 and 140 in our experiments), the traversal is done very fast. Probably, the most expensive operation in the algorithm is the evaluation function for preserving detail. So, the interest of the algorithm depends on whether this cost is lower than the cost in rendering the most detailed LoD using the algorithm shown in figure 6.

8. Results

The experiments were carried out on a personal computer with Linux operating system, Pentium D CPU and NVIDIA GeForce 7600-GT GPU. The model was coded in C++ and

	Geometry. M_n		M_r, 140 LoDs		M_r, 100 LoDs		M_r, 50 LoDs	
	#Vertices	#Triangles	#Indices	#Strips	#Indices	#Strips	#Indices	#Strips
Armadillo	172,974	345,944	1,021,755	1,330	1,010,339	951	975,797	436
Dragon	359,173	715,933	2,069,974	1.577	2,053,243	1,171	2,005,531	535

Table 1: *Three iDLoD representations of Armadillo and Dragon objects, with 140 LoDs, 100 LoDs and 50 LoDs respectively.*

	M_{100}		M_{75}		M_{50}		M_{25}		M_0	
	#Vertices	#Indices	#Vertices	#Indices	#Vertices	#Indices	#Vertices	#Indices	#Vertices	#Indices
Armadillo	172,974	549,196	129,956	421,305	86,939	285,610	43,921	145,241	904	3,042
Dragon	359,173	1,030,325	270,463	841,201	181,770	580,827	93,072	301,434	3,710	11,250

Table 2: *DLoD representations of Armadillo and Dragon objects with 5 LoDs, each one reducing the geometry in 25%.*

the *OpenGL* graphics library was used. In our experiments we enabled one light source and the size of the viewport is 1024x768 pixels. For each vertex we send its coordinates and its normal, and we scale the object to the bigger size inside of the frustum. We used the GL_TIME_ELAPSED extension, which provides a query mechanism to determine the amount of time used for completing a set of GL tasks without stalling the rendering pipeline. We used the *NvTriStrip* library for vertex cache aware stripification of geometry. Experiments show that better performance is obtained when triangle strips are stitched together using degenerate triangles. Therefore, triangle strips have been stiched in the off-line process for both DLoD and iDLoD.

Table 1 shows the characteristics of the polygonal models used in the experiments as well as the number of strip indices. The number of clusters in M_r is equal to the number of strips since they are stiched for each cluster. The experiments carried out aim to obtain the performance of the proposed model, iDLoD, and even more important, to compare with the performance of a DLoD that consists of 5 LoDs, denoted as reference LoDs, each one reducing the geometry in 25%. Table 2 show the characterisitcs of the DLoD constructed for each object.

We used two different implementations of the iDLoD model. First, a unique multiresolution representation with the n LoDs, denoted as iDLoD. Second, a sequence of four multiresolution representations, denoted as iDLoD-b, such that each one provides LoDs between two consecutive reference LoDs. Therefore, each multiresolution representation in the iDLoD-b model provides a quarter of the n LoDs. Figure 10 shows the results of the three considered models with two different objects. All of the plots show similar behaviour. As expected, the DLoD produces the highest frame rate. The iDLoD almost reaches DLoD performance: the models with 50 LoDs approach DLoD performance more than the ones with 140 LoDs. The performance of the iDLoD-b implementation goes through the reference values of DLoDs. This is because this implementation provides LoDs formed by a quarter of the clusters at most, which produces the increment of performance respect to the iDLoD implementation.

9. Conclusions

Since J. Clark presented it in 1976, DLoD has been the multiresolution framework mostly used. During the last years, many well-known works have been proposed as an alternative. Mainly, the efforts have been directed to improve CLoD and View-Dependent techniques. However, in the field of real time applications, software developers still prefer the DLoD framework. In this paper we have presented a model based on coding a discrete number of levels of detail, with more LoDs coded than is usual in DLoD, and with an incremental representation, which is often used in CLoD. The difference between two consecutive LoDs is forced to be a surface patch so that it can be optimized off-line for rendering. Experimental results show that this model obtains a performance similar to DLoD by providing optimized LoDs for efficient visualization, while the popping effect is imperceptible. In addition, view-dependent LoDs can be retrieved in spìte of the low granularity.

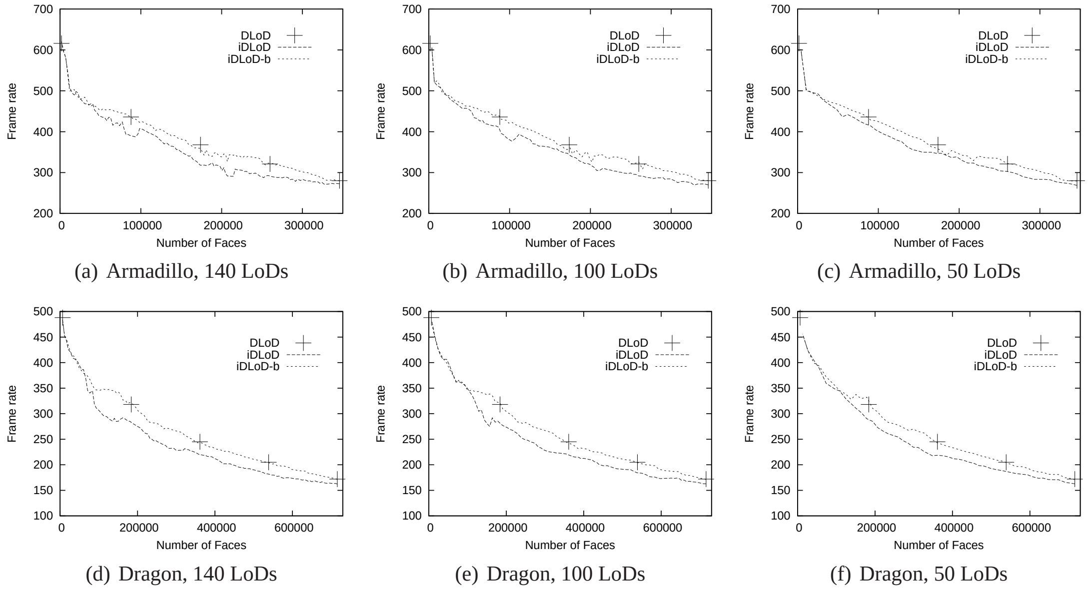
Acknowledgments

This research is supported by the Spanish Ministry of Science an Innovation Grant No DPI2008-06548-C03-01 and CONSOLIDER INGENIO 2010 (CSD2007-00018); and by Fundació Caixa Castelló-Bancaixa Grant No: P1- 1B2009-50.

References

[BGB*05] BORGEAT L., GODIN G., BLAIS F., MASSICOTTE P., LAHANIER C.: Gold: interactive display of huge colored and textured models. *ACM Trans. Graph. 24*, 3 (2005), 869–877.

[BRR*04] BELMONTE O., REMOLAR I., RIBELLES J., CHOVER M., FERNANDEZ M.: Efficiently using connectivity information between triangles in a mesh for real-time rendering. *Future Generation Computer System 20* (2004), 1263–1273.



(a) Armadillo, 140 LoDs
(b) Armadillo, 100 LoDs
(c) Armadillo, 50 LoDs
(d) Dragon, 140 LoDs
(e) Dragon, 100 LoDs
(f) Dragon, 50 LoDs

Figure 10: *Results for the Armadillo and Dragon objects with different number of LoDs.*

[CGG*04] Cignoni P., Ganovelli F., Gobbetti E., Marton F., Ponchio F., Scopigno R.: Adaptive tetrapuzzles: Efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. *ACM Transactions on Graphics 23* (2004), 796–803.

[Cla76] Clark J. H.: Hierarchical geometric models for visible surface algorithms. *Commun. ACM 19*, 10 (1976), 547–554.

[ESAV99] El-Sana J., Azanli E., Varshney A.: Skip strips: Maintaining triangle strips for view-dependent rendering. In *Proc. IEEE Visualization '99* (1999), IEEE Computer Society Press, pp. 131–138.

[Gar99] Garland M.: Multiresolution modeling: Survey & future opportunities. In *State of the Art Reports of EUROGRAPHICS '99* (1999), pp. 111–131.

[Gar04] Garland M.: Qslim. http://graphics.cs.uiuc.edu/~garland/software/qslim.html.

[GH97] Garland M., Heckbert P.: Surface simplification using quadric error metrics. In *Proc. SIGGRAPH '97* (1997), pp. 209–216.

[Hop97] Hoppe H.: View-dependent refinement of progressive meshes. *Computer Graphics 31* (1997), 189–198.

[HSH09] Hu L., Sander P., Hoppe H.: Parallel view-dependent refinement of progressive meshes. In *Proc. of the 2009 symposium on Interactive 3D graphics and games* (2009), pp. 169–176.

[JWLL05] Ji J., Wu E., Li S., Liu X.: Dynamic lod on gpu. *Computer Graphics International Conference 0* (2005), 108–114.

[LRC*03] Luebke D., Reddy M., Cohen J. D., Varshney A., Watson B., Huebner R.: *Level of Detail for 3D Graphics*. Elsevier Science, 2003.

[PS97] Puppo E., Scopigno R.: Simplification, lod and multiresolution principles and applications. *Tutorial Notes of Eurographics'99 16*, 3 (1997).

[RCG*09] Ripolles O., Chover M., Gumbau J., Ramos F., Puig-Centelles A.: Rendering continuous level-of-detail meshes by masking strips. *Graphical Models 71*, 5 (2009), 184–195.

[RCRG06] Ramos F., Chover M., Ripolles O., Granell C.: Continuous level of detail on graphics hardware. In *DGCI* (2006), pp. 460–469.

[RLB*02] Ribelles J., López A., Belmonte O., Remolar I., Chover M.: Multiresolution modeling of arbitrary polygonal surfaces: a characterization. *Computers & Graphics 26*, 3 (June 2002), 449–462.

[SG03] Southern R., Gain J. E.: Creation and control of real-time continuous level of detail on programmable graphics hardware. *Comput. Graph. Forum 22*, 1 (2003), 35–48.

[SM05] Sander P., Mitchell J.: Progressive buffers: view-dependent geometry and texture lod rendering. *Eurographics Symposium on Geometry Processing* (2005), 129.