Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

One of the main problems of interactive graphics applications, such as computer games or virtual reality, is the geometric complexity of the scenes they represent. In order to solve this problem, different modelling techniques by level of detail have been developed, trying to adapt the number of polygons of the objects to their importance inside 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*, etc., where models with continuous levels of detail, based mainly on Progressive Meshes [Hop96], are introduced. Figure 1.1 shows the ogre head model and an approximation at a medium level of detail which would be rendered much faster.

The tendency in the 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.

Nevertheless, the multiresolution models available nowadays are not always suitable for all kind of meshes. Many of the current interactive applications such as flight simulators, virtual reality environments or computer

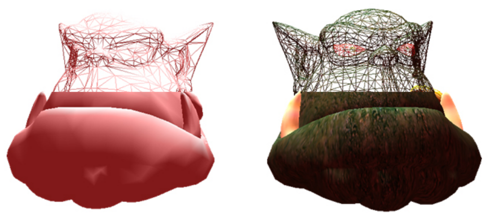


Figure 1.1: Original Ogre head model and a less detailed approximation.

games take place in outdoor scenes, where the vegetation is an essential component. The lack of trees and plants can detract from their realism. Tree modelling has been widely investigated [PL90] [LD99], and its representation is very realistic (figure 1.2). However, tree models are formed by such a vast number of polygons that real-time visualization of scenes with trees is practically impossible, and it is necessary to resort to some method that diminishes the number of polygons that form the object, such as multiresolution modeling. But the multiresolution models that have appeared up to now deal with general meshes and do not work properly with this kind of mesh.

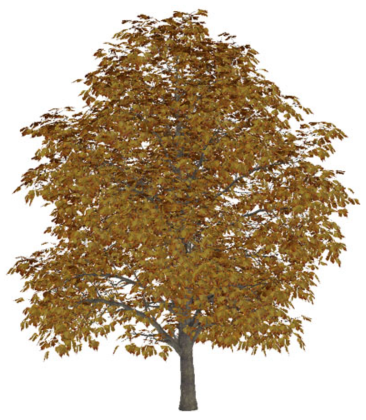


Figure 1.2: Aesculus hippocastanum. Tree modelled with the commercial modelling tool Xfrog. 192.179 triangles.

1.2 Developed models

We have developed two different multiresolution models: a model for general meshes (*LodStrips*) and a model for plants and trees (*LodTrees*). Both models can make a selection of the LoD in running time in order to establish a balance between the number of polygons with which the object will be represented and the amount of time needed to visualize it. When modeling vegetation, we will need to do a separate processing. Branches, including the trunk, can be considered as general meshes and will therefore be handled by the general mesh model. Leaves, on the contrary, will be represented using their own specific model.

The construction of a multiresolution model involves the simplification of the original model, in order to obtain the sequence of different approximations that compose the model diminishing the number of polygons while maintaining their appearance. This way, we have developed several algorithms to perform the simplification of the meshes following diverse criteria. Moreover, it is important to mention that the LodStrips model works with a stripified mesh. Therefore, we will also need a method to convert polygonal meshes, which are geometrically composed of triangles, to triangle strips.

1.3 Document organization

This document is organized according to the developed models commented previously. The supporting modules that are necessary for the correct performance of this workpackage will be described first, and the multiresolution modules will be described last.

Chapter 2 addresses the problem of finding a good set of strips for a given triangulation and the solution we have implemented. Although this solution offers good results, there is still room for further improvement.

Chapter 3 describes the different simplification algorithms we have developed, for both general meshes and trees and plants. For the simplification of general meshes, we propose the use of two different algorithms, one based on the geometric simplification of the 3D models, and a new image based simplification algorithm which obtains simplified meshes that appear more similar to a human observer. The algorithm for the simplification of foliage uses a new method, leaf collapse: two leaves disappear to create a new one, so that the leaves obtained after collapsing preserve an area similar to that of the collapsed leaves.

Finally, chapters 4 and 5 offer the scientific description of the multiresolution models. Chapter 4 presents the multiresolution model for general meshes, and chapter 5 discusses the model for trees and plants.

Chapter 2

Quality strips for models with level of detail

2.1 Introduction

Multiresolution models are widely employed in computer graphics applications as they allow us to reduce the amount of geometry information sent to the graphics system, which results in an improvement in performance. The use of triangle strips in these models offers further improvement, since it adds a compact representation of the connectivity existing in a triangle mesh and enables faster rendering. The main problem of multiresolution models based on strips arises when, starting from a set of strips representing the initial mesh at maximum detail and applying the successive simplifications, the strips start to include a large quantity of degenerated triangles, which have no mathematical area and imply sending information for triangles that will not be rendered. An example of these low-quality strips can be observed in figure 2.1, where the strip in the middle is collapsed after two simplification steps, where edges 0, 3 and 1, 2 are also collapsed.

One possible way to overcome this problem is to use strips which are dynamically generated for each level of detail. Research has been conducted on this approach, and it is possible to find methods of building and maintaining a good set of triangle strips like the one proposed by Stewart [Ste01], and also multiresolution models based on dynamic strips [Hop96][SP03]. But the additional cost involved in generating the strips for every level of detail is high; therefore the use of static strips, despite their limitations, can turn out to be more suitable.

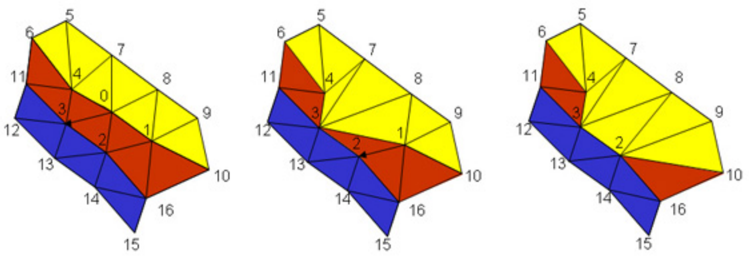


Figure 2.1: Collapse of a strip.

We propose an algorithm for building triangle strips for static models which avoids working with low quality strips. It constructs the strips from the minimum to the maximum level of detail following the simplification sequence, while maintaining the original appearance of the 3D model.

2.2 Previous work

The use of the triangle strip primitive allows us to greatly accelerate the visualization of geometry. Although finding an optimum set of strips from a given triangulation is an NP-complete problem [ESV96], there are different solutions which, though not optimum, maximize its performance following diverse criteria.

Among the many studies carried out we can highlight the methods introduced in [ESV96][AHB90][XHM99] to generate strips in a static way, as well as the one proposed by Stewart [Ste01] for the dynamic generation of strips. The suggested algorithms show differences in generation and rendering speed, in the use of memory or in the number of strips generated, which make them more suitable for a specific use. It is also important to comment on the studies which make optimum use of the vertex cache, in order to maximize references to vertices already loaded in registers inside the cache. In this regard, methods such as Hoppe's [Hop99] or the one devised by Nvidia [BD02] have appeared in recent years. The company referenced before has created its own library in order to find strips that derive the maximum benefit from vertex caches and from the spatial locality of vertex buffers.

Finally, mention should be made of the algorithm proposed by Belmonte [BRRC00], which, as the method presented here, also considers the generation of strips following a simplification criterion. But in this case, as in
]

the rest of the algorithms, the generation starting from the maximum level of detail and its subsequent simplification causes degeneration of the strips obtained at levels of low detail.

2.3 Our approach

The objective of the algorithm we present is to find triangle strips that are optimum for multiresolution models, avoiding the strips to be cut when simplified.

With this intention, we start out with the mesh simplified to the minimum level of detail, which means we may start with just a few triangles. For every step of our algorithm we will need to know the vertices that split, the two new triangles that appear and the set of existing triangles that must be modified. It is possible to collect all this information during the simplification process of the original mesh.

We can consider two main cases. The first of them, shown in figure 2.2, represents a refinement along a border edge between two strips. In this situation, two triangles will be modified and a total of four new vertices for two new triangles will be inserted, as each strip will need a swap operation. In this way, following the example in figure 2.1, strips will be initially made up of vertices 1, 2, 3, 4 and 1, 5, 3, 6. After the split, they will be made up of vertices 1, 2, 7, 2, 3, 4 and 1, 5, 7, 5, 3, 6. We should mention that it might be possible to find this case with just one strip, this only being possible if the strip includes a fan.

The second case can also be observed in figure 2.2. It shows a split along an edge which is not a border. This makes it impossible to add the two new triangles into the existing strip without resorting to degenerated triangles that would increase the number of vertices required. In this case it is necessary to create a new strip, since we will be able to insert only one of the new triangles into the existing strip. The insertion of these two new triangles will involve a rise of five units in the total number of vertices. Thus, the strip that was initially made up of vertices 1, 2, 3, 4, 5, 6 will now contain edges 1, 2, 3, 2, 7, 4, 5, 6 and a new strip will appear with vertices 3, 7, 5.

With the two general cases introduced, the algorithm to build strips will be similar to the one presented in figure 2.3. The two new triangles share an edge with one of the triangles that will be modified, and it will therefore be through this edge that we will locate the triangle or triangles where we will

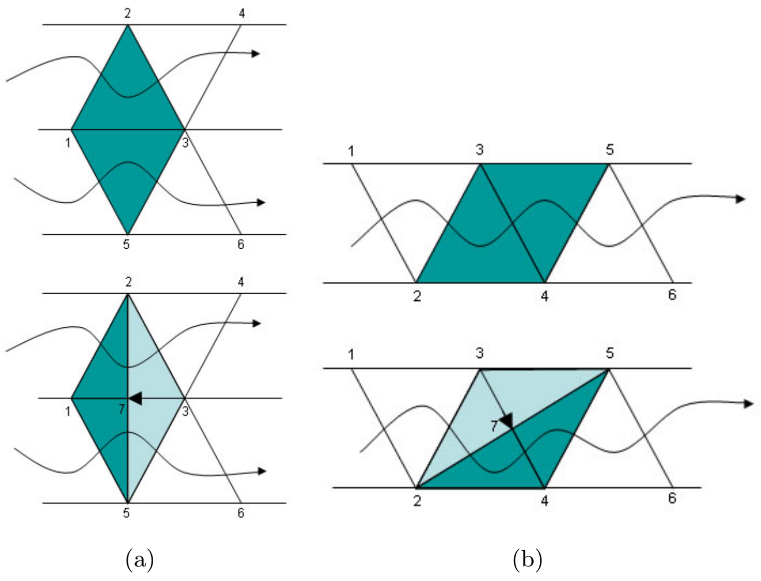


Figure 2.2: a)Vertex split along a border edge between two strips. b) Edge expansion along a non-border edge inside a strip.

be able to insert them. Once the edge is found, we will simply insert the new triangle taking care to choose the right side of the edge. If we do not find that edge, we will be obliged to create a new strip. Finally, we will always have to check that all the modified triangles have been changed correctly. We must respect all the changes implied in each step, since otherwise we will not obtain the correct polygonal model when we reach the maximum level of detail.

```
if find_edge() do
  choose_side();
  insert_triangle();
else
  create_strip();
  check_modifications();
end if
```

Figure 2.3: Strip generation algorithm.

2.3.1 Optimizations

With the algorithm proposed, most steps involve the insertion of four new vertices for the two new triangles. These four insertions allow us to obtain a 30% saving with respect to the three vertices per triangle that would be necessary if we represented the model with a triangle mesh. To improve these results, the algorithm has been extended so that, in each step, no repeated vertices or edges and no unnecessary edges are inserted. Furthermore, it is possible to improve the results if, each time we must create a new strip, we try to insert the new triangle at the end of an already existing strip.

On many occasions we have no choice but to insert a new triangle as a new strip. In successive iterations we may have to add a new triangle next to this one. But, depending on how we have inserted it, we will be able to do the new insertion or not. This is due to the fact that, when inserting a new triangle, one of its three edges will not be explicitly reflected on the strip and then we will not be able to find it in the search for edges in our algorithm. In order to avoid this situation as many times as possible, we have developed a function that predicts the usefulness of the three edges. This prediction is carried out by following their evolution throughout the remaining refinement steps. With this information we will be able to decide which of the three edges is less useful when it comes to inserting the new triangle as a new strip. At this point, we have to choose whether it is better to eliminate the edge that will be used sooner, or the one that will be used later. Our experiments have proven that penalizing the edge we will use sooner offers better results, since it allows the new triangles to be inserted into strips in the last steps of the process.

2.4 Results

In order to analyze the strips generated as a result of the algorithm presented here, we have conducted a study of the vertices sent to the graphics card for different levels of detail. Results are shown from table 2.1 to 2.2. These data have been compared with those obtained using a simple triangle mesh for each level of detail and with a multiresolution model based on strips that uses a simplification method involving edge collapse, such as Skip-Strips [ESAV99], MTS [BRR$^+$01] or LodStrips [RC04]. The experiments were carried out using Windows XP on a Dell PC with a processor at 2.8 Ghz, 1 GB RAM and an Nvidia GeForce 6600 graphics card with 256MB RAM.

Figure 2.4 shows the results obtained for three different polygonal models. They show the number of vertices sent to the graphics processor for each level

Model	Triangles
Cow	5804
Al Capone	7124
Bunny	69451

Table 2.1: Number of triangles of the geometric models used in the experiments.

	Triangles	SMM	Strips
Cow	25.272.930	15.752.211	13.777.800
Al Capone	38.127.687	23.158.015	20.883.500
Bunny	3.619.981.407	2.163.219.828	1.976.880.000

Table 2.2: Results in total number of vertices sent going from the minimum to the maximum level of detail.

	Triangles	SMM	Strips
Cow	100%	62.3%	54.5%
Al Capone	100%	60.7%	54.7%
Bunny	100%	59.7%	54.6%

Table 2.3: Percentages of vertices sent going from the minimum to the maximum level of detail.

of detail, considering 100% as the maximum detail and 0% as the minimum, although in the tests 10% was taken as the minimum since a lower level of detail would entail complete loss of the original shape of the 3D model. It should be pointed out that the information marked in the figures with the name Triangles refers to a model that uses triangle meshes, SMM is an abbreviation of strip-based multiresolution model and Strips are the result of the algorithm proposed here.

As we expected, this algorithm sends fewer vertices than a triangle mesh. It can be observed that for high levels of detail the example multiresolution model sends fewer vertices. But if we consider the total number of vertices necessary to go through all the levels of detail, from the minimum to the maximum, our method involves less information traffic, as shown in table 2.3. In this way, for more than 60% of the levels the algorithm presented sends less geometric information to the GPU.

In figure 2.5 the resulting stripification of the cow model using our algorithm is presented. In addition to the strips representing the model at maximum level of detail, two more images taken during the process are also offered, for a 33% and a 66% of the total detail.

2.5 Conclusions

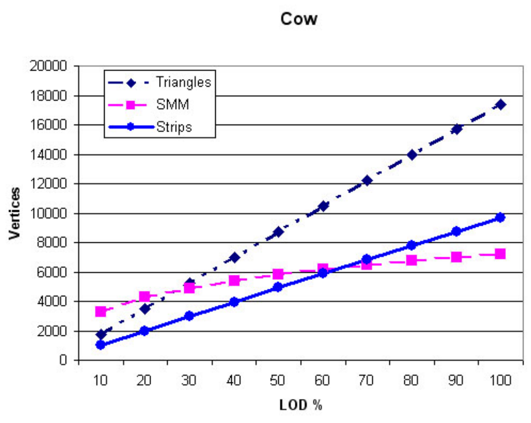
We have presented a new method for strips generation in which we obtain a set of triangle strips that will maintain its quality throughout the simplification process. We improve on the results offered by previous stripification algorithms, since all of them offer low quality for levels of coarser detail. In contrast, our algorithm needs a larger number of triangle strips at levels of high detail. But, in general, the total number of vertices covering from the minimum to the maximum level of detail is about 15% lower than the multiresolution model ours was compared with and this means a saving of around 50% with respect to the original triangle mesh.

Through the results it can be observed how, as the level of detail is reduced, the strips used by the multiresolution model worsen, reaching a point near 20% of detail, where it is even better to use the triangle mesh instead of the strips the model offers. This lends further support to the idea that has encouraged the investigation of this new algorithm, which avoids working with low quality strips.

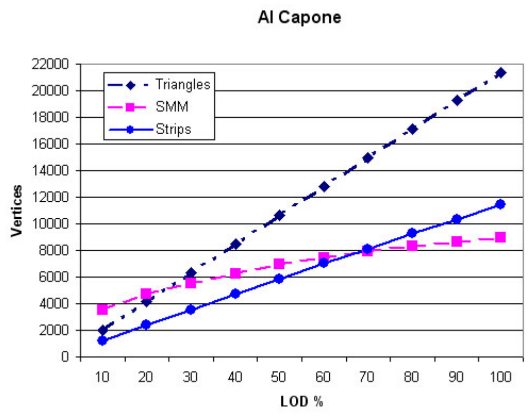
From the results obtained we can also infer that our design loses quality as the models increase in size. In many applications, such as games, it is usual to work with models which are not as complex as the ones analyzed, where it is easy to offset this low polygonal complexity with a correct treatment of illumination or other aspects of the visualization of geometry.

2.6 Further optimizations

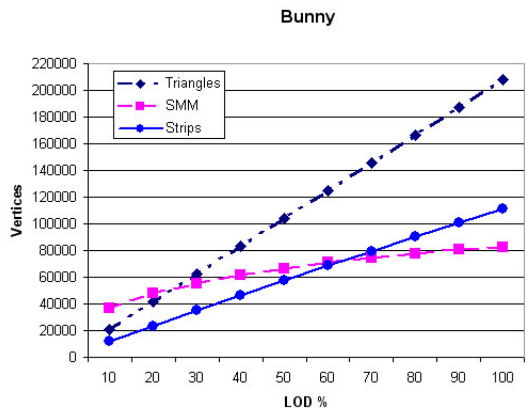
The correct management of the vertex cache is a very important issue for future improvements in the algorithm. Thus, grouping closer strips may result in an increase of the performance of the set of strips generated. In the same way, we consider that a significant improvement can be achieved by utilizing a prediction system that better fits the evolution of simplification, since this simple prediction method already offers a 5% improvement in the number of vertices sent.



(a) Results for the cow model.

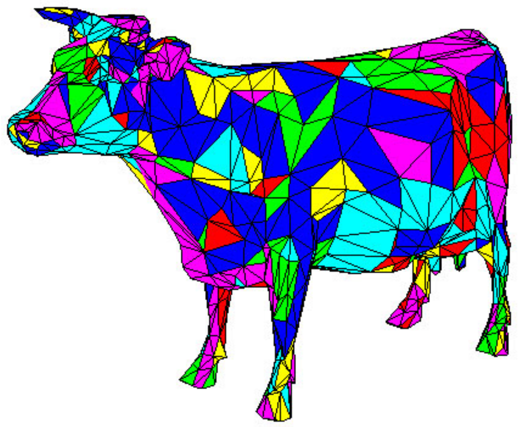


(b) Results for the Al Capone model.

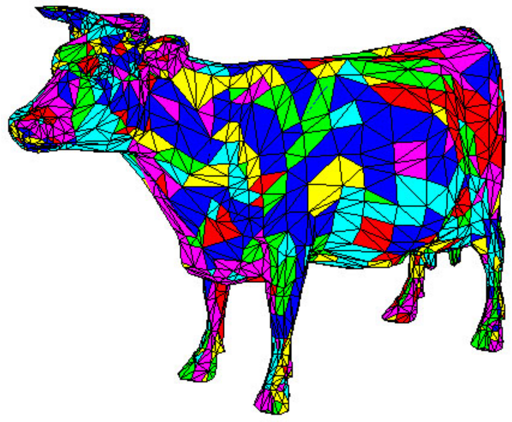


(c) Results for the bunny model.

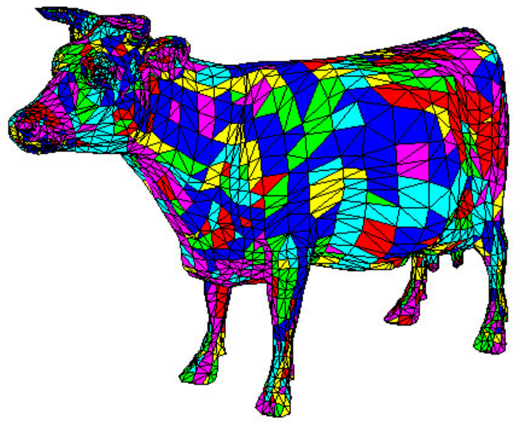
Figure 2.4: Some results obtained with the algorithm.



(a)



(b)



(c)

Figure 2.5: Stripification of the cow model for a) 33% b) 66% and c) 100% of the maximum detail.

Chapter 3

Simplification

3.1 Introduction

The construction of a multiresolution representation is based on two main elements. On the one hand, the original geometry of the object at its maximum level of detail. On the other, the simplification sequence that permits the generation of the different levels of detail. This chapter deals with the simplification methods that we use to construct our multiresolution models LodStrips and LodTrees.

The chapter starts revising the methods used to simplify general meshes (non-manifold surfaces), considering the geometric and the image-based method. Finally, we introduce our algorithm to obtain the simplification sequence for tree models.

3.2 Geometric simplification

3.2.1 Previous work

Algorithms for polygonal mesh simplification can be categorized into the following classes:

Vertex Decimation [Sch97, CCMS96]. These methods basically use an iterative vertex selection for removal. Once a vertex is removed, all faces that share that vertex are also removed and the resulting hole is triangulated. This type of algorithms are limited to manifold meshes due to its retriangularization schemes.

Vertex Clustering [LT97, RB93]. These methods are based on a bounding box divided into a grid. All vertices contained in a single cell are mapped into a single vertex, and faces indices are modified to reflect the changes. These methods tend to be really fast, but the quality of the resulting simplified mesh is quite low.

Edge Contraction [GH97]. These methods use an iterative edge selection for removal. At each step, a new edge (or vertex pair) is selected and removed. One of the vertices is also removed and all affected faces are mapped into the other vertex. Degenerated faces are also removed from the mesh.

Morphological operations [NT03]. These methods apply morphological operations (similar to those used in 2D images but applied to 3D voxels). These methods, such as erosion and dilation, are quite fast and offer fairly good results, taking out detail from the original mesh.

The most extended and accurated methods used for surface simplification, as for example [HG97],[PS97] or [Gar99], use techniques based on Iterative Edge Contractions to simplify models, and maintains surface error approximations using quadric metrics. These methods allow us to merge vertices and modify the edges that use these vertices in order to keep the connectivity with the other vertices of the edges.

The simplification criterion used in this method is calculated per-vertex from the triangles that share that vertex. These methods set a decimation factor for each edge, so that the edge with the lowest value will be removed first. The decimation cost reflects how much the appearance of the object will change, so that the edges with lower values are first contracted. The edge decimation cost is calculated from the decimation cost of the two vertices involved.

For each edge contraction (v_1,v_2)$\rightarrow \overline{v}$ the following steps are followed:

1. Delete vertex v_1.
2. Remove all triangles that share the edge.
3. Remap all triangles shared by v_1 to v_2.
4. Move the vertex to an optimum position.
5. Recalculate the cost (decimation coefficient) for the vertex based on the new connectivity information.

To calculate the decimation cost, the basic idea is to calculate the distance between the vertex and all planes (triangles sharing that vertex). That value will represent how much that contraction would change the shape of the mesh.

It is important to note that a contraction pair is not restricted to a triangle edge, but to a pair of vertices that are closer than a given distance.

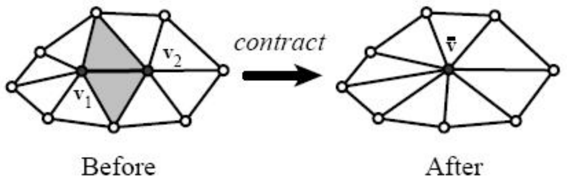


Figure 3.1: Example of edge contraction.

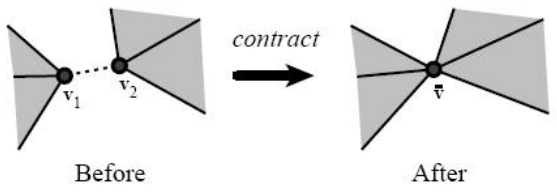


Figure 3.2: Example of pair contraction based on a distance threshold.

3.2.2 General description

We have implemented several improvements on the aforementioned simplification algorithm. The obtained result from a high resolution mesh is a simplified mesh with a simplification sequence. The simplification sequence contains for each simplification step the following information:

- The vertices of the collapsed edge
- The removed faces
- The displacement of the non-removed vertex
- A list of the modified faces

Global Simplification

The methods in the literature are efficient algorithms to simplify surfaces with a unique mesh. However, if a surface is composed by a set of individual submeshes without any physical interconnection information, the simplification process tends to produce artifacts like holes (see figure 3.5(b), and figure 3.7(b)) between the patches. This way, the final simplified model does not preserve the appearance from the original object.

We have implemented an improvement for this method, which consists of three steps:

- A pre-process step, which merges all the submeshes in a unique virtual mesh, storing all the information about the original connectivity of the meshes. This step is based on the distance of the vertices of different meshes in the original object. Moreover, if two neighbour submeshes are composed of different materials, the edges between them are marked as virtual boundaries, so that the simplification step keeps the original shape.
- The simplification step, which simplifies the whole mesh, using the algorithm improved in two directions: the interpolation of texture coordinates and the generation of normals per vertex.
- The post-process step retrieves the simplified mesh and splits it into different submeshes, using the interconnectivity information of the original submeshes stored in the pre-process step.

The result is the simplification of each single submesh maintaining the original appearance as much as possible and avoiding the artifacts produced by a plain contraction method.

This is especially useful for models which are often composed of a set of small patches without any interconnection, except appearance connectivity.

Local Simplification (sub-mesh simplification)

The implemented method presents also the possibility of a local simplification of the model. That is, if the object is composed by more than a single mesh, each mesh could be simplified to a different level of detail. For more details on this method, see algorithm 3.3.

It has to be considered that simplifying a submesh produces a minimum simplification of the neighbour meshes in order to avoid creating holes in the surface.

```
// Clear the edge buffer
simp_edges = [];
for i<mesh.edges.size() do
  if (mesh.submesh.has_edge(i)) then
    simp_edges.add(i);
  end if
simplify_edges(simp_edges);
end for
```

Figure 3.3: General scheme for local simplification.

Boundaries preservation

Using the information stored in the pre-process step, the implemented algorithm allows the preservation of the boundaries, so that when an edge with a vertex in a boundary has to be remapped, the other vertex will overlap the first one. This way, the original shape of the virtual boundary remains unchanged while possible. A general scheme to this approach can be seen in algorithm 3.4.

```
vertex_to_move = None;
if (mesh.boundaries.has_vertex($v_1$)) then
  if (mesh.boundaries.has_vertex($v_2$)) then
    vertex_to_move = normal_placement($v_1$,$v_2$);
  else
    vertex_to_move = $v_2$;
  end if
else if (mesh.boundaries.has_vertex($v_2$)) then
  vertex_to_move = $v_1$;
else
  vertex_to_move = normal_placement($v_1$,$v_2$);
end if
```

Figure 3.4: General scheme for virtual boundaries preservation.

If a vertex in the edge is classified as a virtual boundary to preserve, algorithm 3.4 is executed. If both vertices are in a boundary (or different boundaries) we compute a normal vertex placement. If only one vertex is in a virtual boundary, the other vertex of the edge is the one to be moved. Else, if no vertices are on a boundary, we also compute a normal vertex placement.

Results of this method can be observed in figure 3.6.

Texture coordinates interpolation

At each simplification step texture coordinates for each modified vertex need to be recalculated in order to maintain the mapping appearance. The new texture coordinate is calculated based on the displacement of the mapped vertex using a linear interpolation.

To obtain the offset that must be applied to the texture coordinates of the modified vertex we propose the following system of equations:

$$\left\{\begin{array}{l} Q^{'}_x = \alpha U_x + \beta V_x + \gamma N_x \\ Q^{'}_y = \alpha U_y + \beta V_y + \gamma N_y \\ Q^{'}_z = \alpha U_z + \beta V_z + \gamma N_z \end{array}\right. \tag{3.1}$$

having $\vec{Q'} = \vec{Q} - \vec{P^1}$, $\vec{U} = \vec{P^2} - \vec{P^1}$, $\vec{U} = \vec{P^3} - \vec{P^1}$, where $\{\vec{P^1}, \vec{P^2}, \vec{P^3}\}$ are the three vertices of a modified triangle, $\vec{Q}$ is the new position of the modified vertex and $\vec{N}$ is the triangle normal.

Resolving this system of equations (3.1), we obtain:

$$\alpha = \frac{-(N_zQ^{'}_yV_x - N_yQ^{'}_zV_x - N_zQ^{'}_xV_y + N_xQ^{'}_zV_y + N_yQ^{'}_xV_z - N_xQ^{'}_yV_z)}{-N_zU_yV_x + N_yU_zV_x + N_zU_xV_y - N_xU_zV_y - N_yU_xV_z + N_xU_yV_z}$$

$$\beta = \frac{-(-N_zQ^{'}_yU_x + N_yQ^{'}_zU_x + N_zQ^{'}_xU_y - N_xQ^{'}_zU_y - N_yQ^{'}_xU_z + N_xQ^{'}_yU_z)}{-N_zU_yV_x + N_yU_zV_x + N_zU_xV_y - N_xU_zV_y - N_yU_xV_z + N_xU_yV_z}$$

$$\gamma = \frac{-(-Q^{'}_zU_yV_xQ^{'}_yU_zV_x + Q^{'}_zU_xV_y - Q^{'}_xU_zV_y - Q^{'}_yU_xV_z + Q^{'}_xU_yV_z)}{N_zU_yV_x - N_yU_zV_x - N_zU_xV_y + N_xU_zV_y + N_yU_xV_z - N_xU_yV_z}$$

where α, β and γ are the coordinates of $\vec{P}$ expressed in the triangle coordinate system. They also expresses how much the modified vertex has been moved in triangle coordiante system units, so they can be used to calculate the perturbed texture coordinate for the modified vertex. We use the following formula:

$$\left\{\begin{array}{l} T^{res}_u = \alpha(T^2_u - T^1_u) + \beta(T^3_u - T^1_u) + T^1_u \\ T^{res}_v = \alpha(T^2_v - T^1_v) + \beta(T^3_v - T^1_v) + T^1_v \end{array}\right. \tag{3.2}$$

where $\vec{T^1}$,$\vec{T^2}$ and $\vec{T^3}$ are the original texture coordinates of the three vertices, $\vec{T^{res}}$ is the new texture coordinate for the modified vertex and α, β are the coefficients calculated from equation 3.1.

Note that we only use α and β from equation 3.1 because texture coordinates are bidimensional vectors contained in the plane formed by the triangle. As γ is the displacement along the normal vector of the triangle, we do not need it.

Normals generation

The simplification step keeps track of the normal indices at each edge contraction, so that in the post-process step normals are generated using this information. Thus, the simplified mesh has on each part of the model the same type of normals (vertex or face normals) that the original model.

3.2.3 Results

The following examples show the improvements made to the current contraction methods. The images that are shown as examples are generated using the Qslim algorithm, which uses an edge contration system.

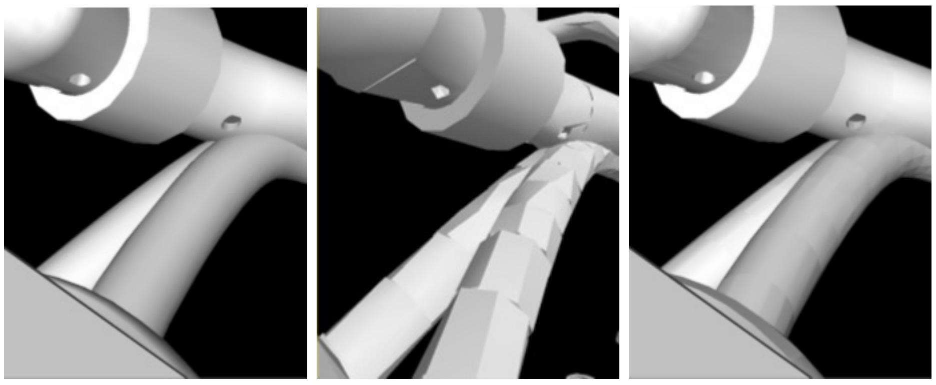
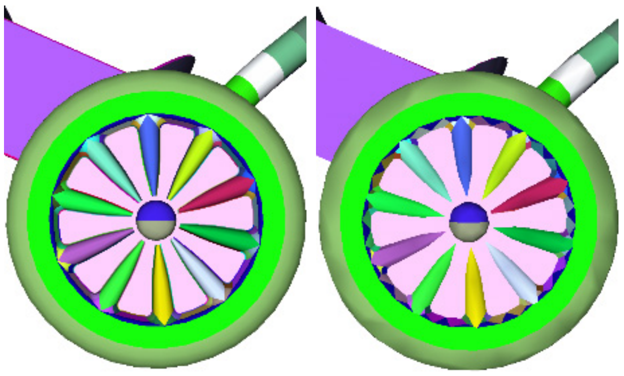

(a) Original model (b) QSlim (LOD 0.3) (c) Our method (LOD 0.3)

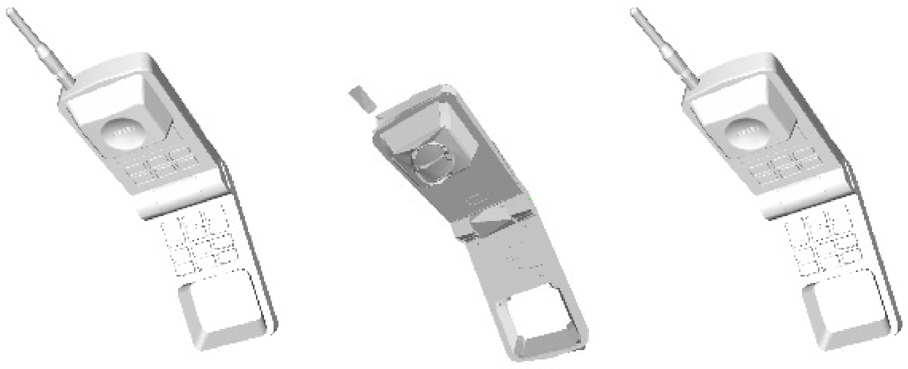
Figure 3.5: Example of global simplification.

In figure 3.5 we can see how our implementation improves the quality of the simplified model when the model is composed of a lot of small patches. The image on the left is the original model. The image in the center is the model produced by the QSlim algorithm at 30% of polygons. We can see the holes and artifacts produced during the simplification. The image on the right shows the same model simplified with our improved technique. We can see how all the previous artifacts have disappeared, and the mesh has a smooth transition through all the patches. Moreover, we can see in figure 3.6 that the boundaries of the original object are respected as much as possible.



(a) Original model (b) Our method (LOD 0.3)

Figure 3.6: Example of boundaries preservation.



(a) Original model (b) QSlim (LOD 0.3) (c) Our method (LOD 0.3)

Figure 3.7: Example of global simplification.

On the left of figure 3.7 the original model is shown, which is composed of a high number of submeshes. In the center the simplified model with the QSlim is shown. In this image it can be observed the wide quantity of holes and artifacts that has been appeared with a simplification of 30%. And on the right we can observe the same object simplified to 30% also with our method. This simplification preserves much better the original shape than the QSlim. Moreover, the object does not present any artifact or hole.

An example of local simplification is shown in figure 3.8, where only one of the submeshes is simplified to 10%.

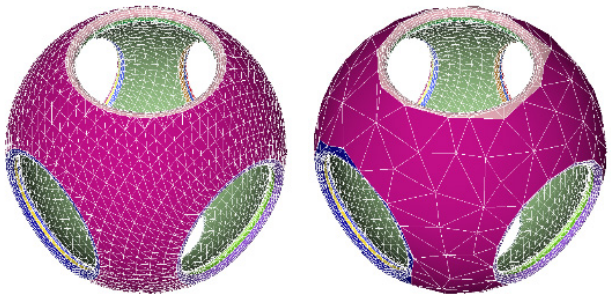

(a) Original model (b) Our method (LOD 0.1)

Figure 3.8: Example of local simplification.

In figure 3.9 it can be observed an example of the texture coordinates interpolation. On the left of the image we can see the original model. In the center the simplified model without interpolation of the texture coordinates is shown. The texture presents high deformations, but with the interpolation of the texture coordinates (image on the right) the simplification produces almost no deformation.

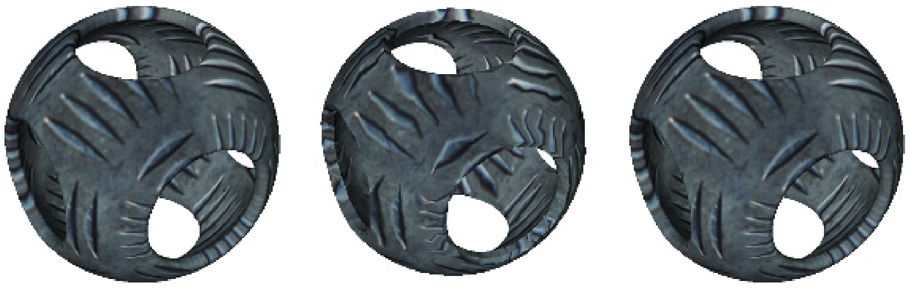

(a) Original model (b) Model without the texture coordinates interpolated (c) Model with the texture coordinates interpolated

Figure 3.9: Example of texture coordinates interpolation.

A comparative of simplification times is shown in figure 3.10, considering different levels of detail of models with different number of polygons. It can be observed that the times increase exponentially.

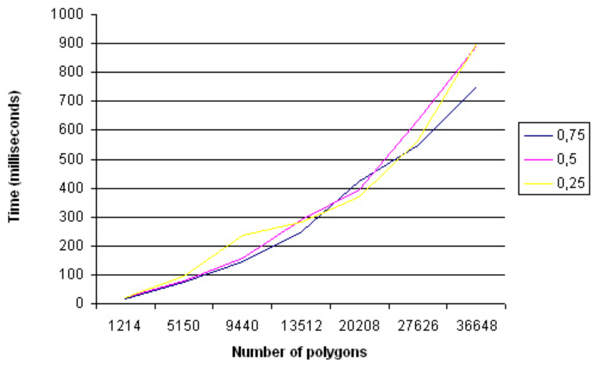


Figure 3.10: Comparative of simplification times.

3.2.4 Conclusions

We have implemented an improvement on the current contraction methods so that it could be used correctly for multiresolution models avoiding the generation of artifacts or holes in the surface of the object. Moreover, the texture coordinates interpolation and the normal generation allow us to obtain a simplified model more similar to the original one. Thus, with this method it can be possible to present a more realistic simplified object than the one obtained with the current algorithms, saving a lot of geometric information.

3.3 Image-based simplification

Most common simplification methods use some technique based on a geometric distance as a quality measure between an original mesh and the one obtained from simplification. By using these methods we can obtain meshes that are visually very similar to the original. The image-based simplification tries to carry out a simplification guided by differences between images more than by geometric distances. The goal is to create simplified meshes that appear similar to a human observer.

Many purely geometric methods only take into account the position of vertices, edges and faces, although lately some incorporate surface properties such as colour and textures coordinates. The most common way of incorporating such properties is to add some weighted sum of deviations to the geometric distance. However, these weights are arbitrarily chosen by the user. One of the objectives of the image-based methods is to manipulate in a natural way the different interactions between the properties of a mesh in only one metric.

The applications that can be benefited by using image-based simplification are those in which the main requirement is visual similarity. Examples of such applications are video games, vehicle simulations, building walk-throughs, etc.

By using the image-based simplification we can achieve meshes with a maximum simplification in hidden zones. A reduced number of applications require exact geometric tolerances with regard to the original model. For this type of applications it would be better to consider some simplification method based on a purely geometric measure. Examples of such applications include collision detection and path planning for part insertion and removal.

3.3.1 Previous work

At the moment there is a unique algorithm by Lindstrom and Turk [LT00] that carries out an image-based simplification. Basically, this method determines the cost of and edge collapse operation by rendering the model from several viewpoints.

The algorithm compares the rendered images to the original ones and sums up the mean-square error in luminance across all pixels of all images. It sorts all edges by the total error they induce in the images and chooses the edge collapse that induces least error. The calculation of the error induced by an edge collapse is very expensive. Lindstrom and Turk use 20 viewpoints in their implementation to compute that error. In order to reduce the calculation cost they take advantage of the fact that the edge collapse operation typically affects a small region of the image and thus only a few triangles. The main advantage of Lindstrom and Turk method is that their metric provides a natural way to balance the geometric and shading properties without needing to use an arbitrary weight of the geometric and shading attributes by the user. On the other side, its main disadvantage is the speed. Despite the optimizations mentioned, it is a very expensive method. For that reason Lindstrom and Turk propose to make two passes. First, simplify the original model by using a geometric method and then apply the image-based method.

3.3.2 General description

The information theory has been used as a basis for the elaboration of metrics that have been successfully applied in image-based modelling. For example:

the viewpoint entropy that is based on Shannon entropy has been used to compute the best viewpoints of an object.

Our goal is to develop new metrics that can be used to evaluate the cost of an edge collapse. To address this, we have proposed the mutual information as a new metric for the elaboration of image-based simplification algorithms.

We define the mutual information for a scene S and a viewpoint x as:

$$I(x,S) = \sum_{i=1}^{N_f} w(x,i)\log(\frac{w(x,i)}{a_i}) + \log(A_t) \tag{3.3}$$

where N_f is the number of faces of the scene, $w(x,i)$ is the projected area (number of pixels) of the face i divide by the total projected area, a_i is the real area of the face i and finally A_t is the real area of all faces of the scene.

This metric doesn't take into account in its current definition the colour information and neither the texture coordinates, although it is possible to incorporate them.

3.3.3 Techniques for Computing the Mutual Information

In order to compute the mutual information, we need the number of pixels covered for each visible triangle from a particular camera position. This number will give us the projected area. Next we analyze several techniques that allow us to compute those areas.

OpenGL Histogram

The OpenGL histogram was first used to compute the viewpoint entropy in [VS04]. The OpenGL histogram lets us analyze the colour information of an image. Basically, it counts the appearances of a colour value of a particular component. However, we can also use it to calculate the area of triangles that are visible from a viewpoint, without reading the buffer. Since version 1.2, OpenGL includes an extension called glHistogram. This extension is part of the image processing utilities. The OpenGL histogram is hardware-accelerated, although there are just a few graphics cards that actually support it (for instance, 3DLabs WildCat). Usually, it is a driver manufacturer's responsibility, as generally happens with the OpenGL extensions, and it is often implemented in software.

In order to obtain the area of each visible triangle by means of the OpenGL histogram, we need to assign a different colour to each triangle. An important limitation is that histograms have a fixed size, normally of 256 different values. This is the most common value in many graphics cards. The glGetHistogram command returns a table that counts each colour value separated into channels. If we use the 4 RGBA colour channels, a 256 item table of 4 integer values will be returned, where each integer is the number of pixels this component has. Thus, if we want to detect a triangle, this should be codified using one single channel. This gives us a total of 1020 different values. That is to say, for channel R (1,0,0,0) up to (255,0,0,0), for channel G (0,1,0,0) up to (0,255,0,0), for channel B((0,0,1,0) up to (0,0,255,0) and finally for channel A (0,0,0,1) up to (0,0,0,255). The value (0,0,0,0) is reserved for the background.

Obviously the main drawback of this technique is that for objects with more than 1020 triangles, several rendering passes are needed. In each pass, we will obtain the area of 1020 different triangles of the object.

Using histograms with a higher number of items, and making a rendering off-screen, will increase the number of colours, and therefore making necessary less rendering passes. However, this possibility is outside the OpenGL specification and it is hardware dependent. It was not possible for us to use a larger size histogram in the several graphics cards we tested.

Hybrid Software and Hardware Histogram

The OpenGL histogram allows us to obtain the area of each visible triangle. However, as we have seen in the previous section, several rendering passes are needed for objects with more than 1020 triangles. Currently, new symmetric buses have appeared such as the PCI Express. In this new bus, the buffer read operation is not as expensive as before. Therefore, it is possible to obtain a histogram avoiding making several rendering passes. The way to get it is very simple. A different colour is assigned to each triangle and the whole object is sent for rendering. Next, a buffer read operation is done, and we analyze this buffer pixel by pixel retrieving data about its colour. Using a RGBA colour codification with a byte value for each channel, up to 256*256*256*256 triangles can be calculated with only one single rendering pass.

Occlusion Query

This OpenGL extension is normally used to identify which scene objects are hidden by others, and therefore we shouldn't send them to render. In fact, what we do is just render the bounding box of an object and, if it is not visible, the object is not send for rendering. However, it can also be used to compute the area of the triangles that are visible from a particular camera position.

The OpenGL ARB_occlusion_query extension returns the number of visible pixels. This feature allows us directly obtain the area of the visible triangles. In order to compute the area of each visible triangle from an object using this technique we will proceed as follows. First, the whole object is sent for rendering and the depth buffer is initialized. Second, we independently send each triangle for rendering. With this procedure it is necessary to make $n + 1$ rendering passes, being n the number of triangles in an object. We must mention that only in the first pass it is necessary to render all the geometry. In the following passes, only one triangle is rendered. However, a high number of renderings can significantly penalize this technique. In order to improve the results, this extension can be used asynchronously in contrast to its predecessor HP_OCCLUSION_QUERY. That is to say, it does not use a "stop-and-wait" execution model for multiple queries. This allows us to issue many occlusion queries before asking for the result of any one. But we must be careful using this feature because, as we mentioned above, this extension was not designed to deal with thousands of multiple queries. Thus, we can have some limitations depending on the graphics card.

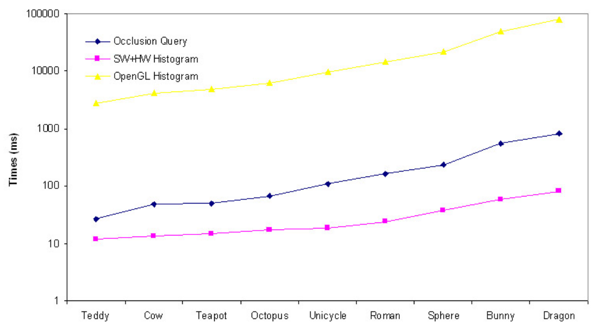


Figure 3.11: Comparison of results obtained from the different analyzed techniques.

Among these techniques, the hybrid histogram by Software and Hardware has the least temporal cost, as we can see in Figure 3.11. These results have been obtained using an NVIDIA GeForce 6800 GT 256 MB card. Thus, we have used this technique for mutual information calculation.

3.3.4 Algorithm

The applied simplification process is similar to the one used by Lindstrom and Turk. We compute the error induced by each edge collapse as the difference between the mutual information of the model before simplifying with the mutual information of the model after simplifying. Then we choose the edge collapse that has the least cost. This error has been calculated as a sum of differences in absolute value of the mutual information for each viewpoint. We have done measurements with 6, 12 and 20 viewpoints. We have checked that the accuracy of the results is better with many viewpoints than with just a few, although the computational cost is higher.

The edge cost c has to be evaluated in each iteration for the entire set of remaining edges. An edge collapse in our algorithm could, initially, affect the cost of any remaining edge. But it does not always happen to each edge.

The current implementation completely recalculates in each iteration the errors induced by all the edge collapses. Thus, it always chooses the optimum. However, it may also be possible to recalculate the error only for a group of edges from the triangles involved in an edge collapse (the neighbourhood of the transformation) in order to accelerate the algorithm. In Figure 3.12 we can see a summary of the algorithm described above.

3.3.5 Results

Due to the high temporal cost of the algorithm, we have done our tests with low complexity models. We have done measurements with some scanned meshes and with meshes created by means of CAD programs. In order to know the quality of our results, we have compared the results obtained at the same simplification level to the results obtained from Qslim [GH97], a simplification algorithm based on a geometric metric developed by Garland and Heckbert.

All models simplified by our image-based simplification method shown below were obtained from 6 viewpoints.

As we can see in figure 3.13(c) our image-based simplification method keeps some spokes in the unicycle wheel. However, Qslim deletes all these spokes (see figure 3.13(b)).

```
// Compute the mutual information for each viewpoint
compute mutual information $I(x,S)x = \{1,\ldots,n\}$

// Generate the initial priority queue of edge collapse.
for ($\forall e \in M$) do
  perform collapse $e$;
  compute mutual information $I^{'}(x,S^{'})/x = \{1,\ldots,n\}$;
  compute cost $c = \sum_{x=1}^{n} \left| I(x,S) - I^{'}(x,S^{'}) \right|$;
  insert the duple $(e,c)$ in queue;
  undo collapse $e$;
end for
// Collapse the mesh
while (queue not empty) do
  delete from queue $e$ with lowest $c$;
  perform collapse of $e$;
  recalculate cost of every edge in queue $e$;
end while
```

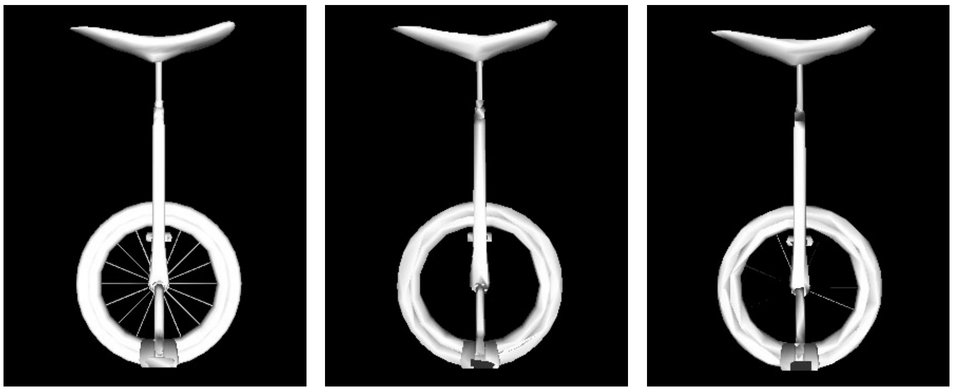
Figure 3.12: Pseudocode of the algorithm for our image-based simplification method.

In figure 3.13(f) we can distinguish how the fish keeps the tail shape while several holes appear in the model simplified by Qslim (see figure 3.13(e)).

In figure 3.14(a) we show the results for a scanned model (Cow) that previously has been presimplified by Qslim up to the level of detail shown. From this one, we have applied both methods. We can check that the results are similar, although we can notice a slight loss of quality in the result obtained from our image-based simplification method (see figure 3.14(c)) compared to the simplified model by Qslim. Anyway, the results can be improved if we increase the number of viewpoints.

3.3.6 Conclusions

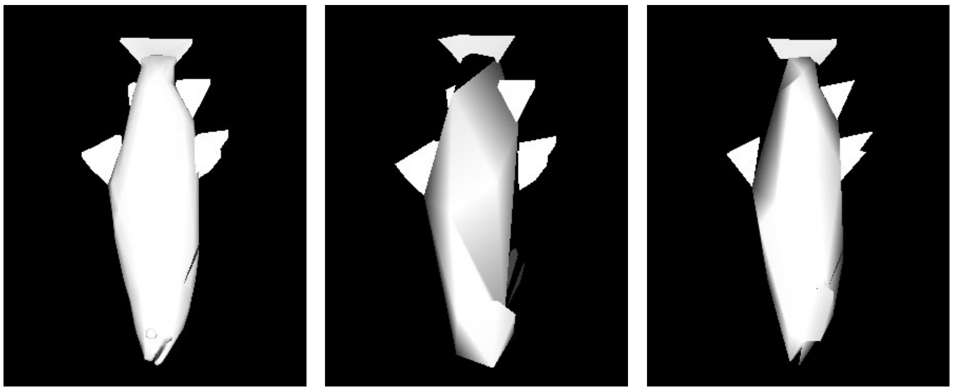
The proposed simplification method based on images mainly obtains very good results. These are composed of different pieces that are assembled together, thus presenting a lot of hidden zones and that is where our proposed algorithm attacks more. The results are similar to those obtained from geometric methods for scanned objects where all triangles are uniform and there are not hidden zones due to the assembly. Although we must take into account that currently the temporal cost is very high in our image-based simplification method.



(a) Original Model. T=3,192.

(b) Qslim. T=1,000.

(c) Image-based Simplification. C=6. T=1,000.



(d) Original Model. T=815.

(e) Qslim. T=100.

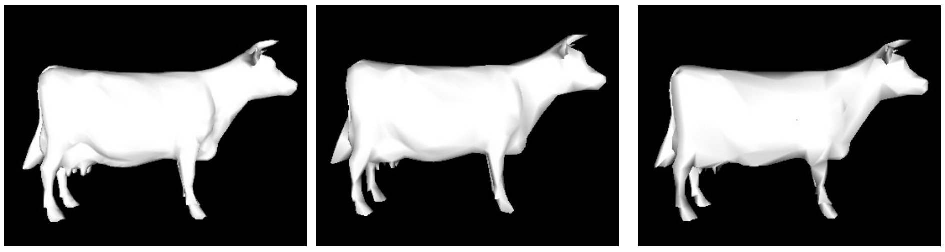
(f) Image-based Simplification. C=6. T=99.

Figure 3.13: Some examples with the unicycle and the fish model.

3.4 Geometric Simplification of Foliage

Many automatic simplification methods have appeared up to now. Applying them to trees, these obtain acceptable results with the meshes of polygons that represent the trunk and the branches. However, they do not work properly with the foliage. The existing simplification methods generally eliminate polygons, so that the appearance of the crown after an automatic process of simplification is that it has been pruned. The number of leaves is less than before, so the tree appears less leafy. The images obtained with these methods are not very realistic and for this reason it is necessary to introduce new solutions.

The method presented for the automatic simplification of foliage diminishes the number of polygons that form the crown, while maintaining its



(a) Original Model. T=3,192. (b) Qslim. T=1,000. (c) Image-based Simplification. C=6. T=1,000.

Figure 3.14: Some examples with the cow model.

leafy appearance. The key to the algorithm is leaf collapse. Two leaves are transformed into a single one, so that the area of the new leaf is similar to the area formed by the two leaves initially. An error function is the way of determining the pair of leaves that will be simplified to create a new one.

3.4.1 Previous Work

Because previous work on geometric simplification has recently been reviewed in several papers [PS97], [HG97], we review the different existing methods of simplification by analyzing the results that would be produced on the mesh of isolated triangles that constitutes the foliage of the trees.

According to [HG97], one of the methods traditionally used to generate simplified versions of an object, is the manual method. The user generates several levels of detail by hand. Simplified versions of trees and plants can be obtained, in the case of using L-systems, by limiting the number of polygons at the time of generating the object. This is one of the most widely used methods for tree geometry simplification. The commercial software Xfrog [LD99] has an additional tool, denominated XfrogMLOD, to generate these levels of detail. The user determines the number of leaves or branches that conform the tree during its modelling. Varying this parameter, different levels of detail of a same tree are obtained. But these tools cannot simplify any geometric description of a tree. They only simplify trees that have been generated with that software.

Another method is Vertex Clustering [RB93], [LT97]. It partitions the vertex set spatially into clusters and unifies all vertices within the same cluster. This produces simplified trees that appear to have been pruned.

Region Merging [KT96] and Wavelet Decomposition [SDS96] methods do not work properly with meshes of isolated polygons. Region Merging is generally restricted to manifold surfaces. Wavelet Decomposition is adequate for surfaces with subdivision connectivity.

Vertex Decimation [CCMS96] [Sch97] does not produce good results either. In each step of the decimation process, a vertex is selected for removal, all the faces adjacent to that vertex are removed from the model, and the resulting hole is re-triangulated. In our case, each leaf is formed by two triangles with an image textured on it. If one of them is eliminated, it would cause the image to be disfigured. The Iterative Contraction [GH97] [Lin00] [Hop96] method would produce the same effects as those mentioned with regard to vertex decimation.

The Foliage Simplification Algorithm has been developed in order to generate different levels of detail of a same tree without losing similarity with the original model.

3.4.2 Tree geometry simplification

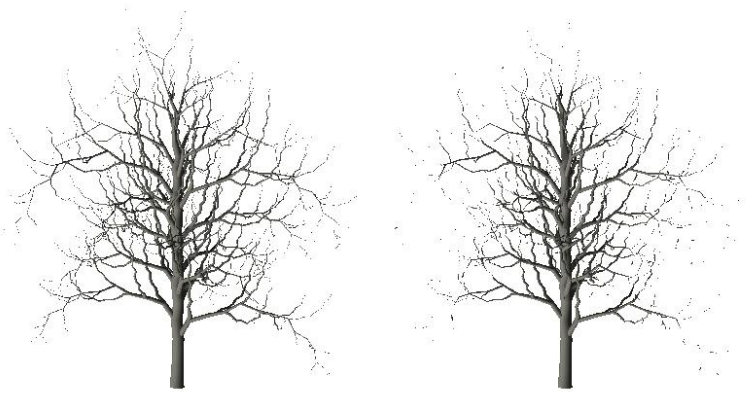
The trees used in our study are modelled by the Xfrog application [LD99] (figure 1.2). They are very realistic, but are generally formed by more than 50.000 polygons each. This is a disadvantage when it comes to generating images in an interactive way.

The trees can be separated into two different parts: the solid component of the tree, i.e. the trunk and the branches, and the sparse component, the foliage or leaves. The trunk and the branches are represented by triangle meshes and the foliage by a set of isolated polygons where each of the leaves is a textured quadrilateral.

The trunk is formed by a set of meshes of polygons. A great number of automatic simplification algorithms existing in the literature deal with this type of objects (figure 3.15).

Secondarily, the foliage of the tree is formed by a set of independent polygons. The automatic simplification algorithms that have appeared up to now do not work properly with this type of representation. Figure 3.16 shows the crown of a tree after application of the automatic algorithm. It can be seen that, in this image, the tree is less leafy.

The algorithm for foliage simplification has been defined with the purpose of diminishing the number of polygons that form the foliage of the tree without losing the leafy appearance. The algorithm is described below.



(a) 38.781 polygons. (b) 12.771 polygons.

Figure 3.15: Image of the original trunk and a simplified version of the same trunk.

3.4.3 Foliage Simplification Algorithm

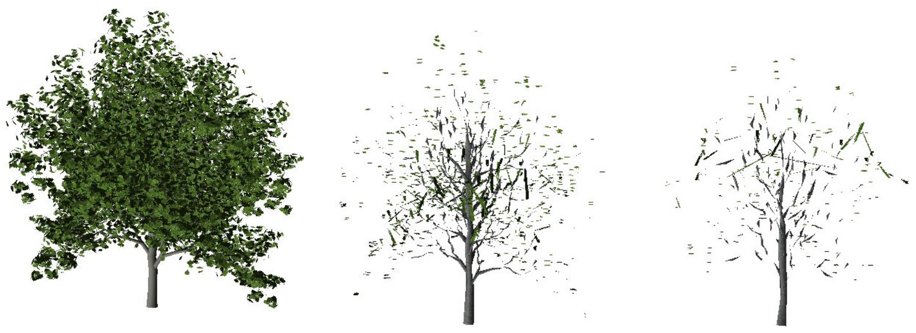
The tree leaves defined with the Xfrog application are represented by quadrilaterals formed by two triangles. The final aspect is obtained by texturing these quadrilaterals with the image of a leaf. The method of simplification presented here repeatedly selects a pair of leaves, which minimises an error function. These leaves disappear and a new one is obtained. The collapsed leaves are eliminated from the list of candidates, and next, the new leaf is evaluated with the leaves that remain in the foliage.

The main idea of this simplification algorithm is that the leaf obtained after collapsing maintains an area similar to that of the collapsed leaves. This is done in order to preserve the appearance of the foliage when the number of leaves is reduced.

The simplification method is characterised by two elements:

- the measurement that specifies the cost of collapsing two leaves, and
- the position of the vertices that form the newly created leaf.

These two questions are discussed in the following sections



(a) 20.376 leaves. (b) 779 leaves. (c) 236 leaves.

Figure 3.16: Example of a simplified foliage from 20.376 leaves to 779 and 236 leaves with the Qslim method.

Leaf Collapse Cost

Given a set of candidate leaves to be collapsed, a pair will be chosen so that the error function is diminished. This function combines distance and planarity between the pair of evaluated leaves.

Assuming that l_t and l_u are two leaves pertaining to a certain level of detail, the error function is as follows:

$$E(l_t, l_u) = E_{dist}(l_t, l_u) + E_{pl}(l_t, l_u) \tag{3.4}$$

where $E_{dist}(l_t, l_u)$ is the distance between leaves, and $E_{pl}(l_t, l_u)$ the planarity level.

The fist term of the function takes into account the measurement of the distance between leaves, according the following equation:

$$E_{dist}(l_t, l_u) = (D_H(l_t, l_u))^2 \tag{3.5}$$

where $D_H(l_t, l_u)$ is the Hausdorff distance. This measurement makes geometric comparisons between two sets of points. Let L_t y L_u be the geometric set of vertices that respectively form the leaves l_t and l_u , the *Hausdorff distance* is defined as:

$$D_H(l_t, l_u) = max(D_H(L_t, L_u), D_H(L_u, L_t)) \tag{3.6}$$

where $D_H(L_t, L_u)$ is:

$$D_H(L_t, L_u) = max_{l_i \in L_t} min_{l_j \in L_u} \|l_i - l_j\| \tag{3.7}$$

In addition, we also measure the planarity between two leaves. This is done in the second term of the equation:

$$E_{pl}(l_t, l_u) = P(l_t, l_u) D_H(l_t, l_u) \tag{3.8}$$

Giving priority to the nearest leaves, the angles formed by the normal vectors of the leaves are compared. Firstly leaves with a similar planarity will be collapsed.

Vertex placement.

The simplification algorithm does not introduce new vertices in the model, as shows figure 3.17. The vertices of the new leaf will be two vertices of each of the collapsed leaves. For this reason, the two vertices of each leaf that are furthest from the other leaf would be chosen. This method will allow us to maintain an area similar to the two original leaves. However, the two triangles that will form the new leaf are not generally in the same plane.

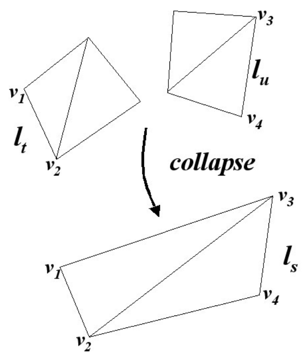


Figure 3.17: Simplification of two leaves to create a new one. The vertices of the original leaves remain.

3.4.4 Algorithm Overview

Data structure

The main data structure of the algorithm is the *Leaf* class, shown in figure 3.18. Each of the leaves in the foliage is represented by an object of this class.

```
class Leaf {
  int Vertices[4];
  float Normal[3];
  bool exists;
  float error;
  int id_couple;
  int lnumber;
};
```

Figure 3.18: Main C++ data structure.

Firstly, all the polygons that make up the foliage are possible candidates to be simplified. For this reason a flag, *exists*, has been introduced which initially would be true in all the polygons.

Each of the leaves will be evaluated with the rest according to the error function $E(l_t, l_u)$. Two data will be stored in them:

- the leaf that makes the error function as low as possible, *id_ couple*, and
- the value of this function, *error*.

The *lnumber* field indicates the number of leaves that have been collapsed to create this leaf. In order to prevent some leaves from growing in an unbalanced form with respect to others, we imposed the condition that two leaves will only be collapsed if the values of their lnumber differ by one. In this way, the coexistence of excessively large leaves and the original, much smaller, leaves is avoided. When all the leaves have been evaluated, the pair of leaves that make the lowest error function is chosen. This selected pair will be taken away from the candidate leaves putting flag *exists* to false, and the new leaf will enter on this list.

Storing cost

As it is said, the leaves in the foliage are formed by four vertices different from the ones formed other leaf. Let $|L|$ and $|V|$ be the total number of leaves an vertices in the foliage. Note that:

$$|H| = \frac{|V|}{4} \tag{3.9}$$

Let us suppose that the storage cost of an integer, real or pointer is one word. The final cost is the following:

$$3|V| + 11|H| = 3|V| + 11\frac{|V|}{4} = 3|V| + 2.75|V| = 5.75|V| \qquad (3.10)$$

Summarizing, we can say that the storage cost of the data structure is $O(V)$.

Algorithm

Considering L_s as the set of leaves that initially form the crown of the tree, the core of the traversal algorithm is summarised in figure 3.19.

```
for each leaf $l_s \in L_s$ do
  $l_s$.error= MAXERROR;
  for each leaf $l_j \in L_s$ do
    if $l_j$.exists and ($l_j$ <> $l_s$) then
      if ($l_j$.lnumber-$l_s$.lnumber) $<$ 2 then
        P = Planarity($l_s$,$l_j$);
        D = Hausdorff($l_s$,$l_j$);
        E = Error (P,D);
        if $l_s$.error $<$ E then
          $l_s$.error = E;
          $l_s$.id_couple = $l_j$;
        end if
      end if
    end if
  end for

end for
```

Figure 3.19: Pseudocode of the algorithm that calculates the error.

When a leaf collapse is performed, the new leaf will be evaluated with the rest of the leaves in the same way as described above. The same process will be done for the candidate leaves that store in the *id_ couple* field one of the two leaves that have just been collapsed.

The number of leaves that make up the simplified crown is determined a priori by the user. This will condition the number of iterations of the algorithm.

Computing cost

Each leaf in the foliage has to be evaluated with the rest of leaves in order to obtain the error induced in their collapse. This action conditions that the computing cost is $O(|H|^2)$.

3.4.5 Results

The method developed was implemented with OpenGL on a PC with Windows 2000 operating system. The computer used was a dual Pentium Intel Xeon at 1.8GHz. with an NVIDIA Quadro4 700XGL graphics processor with 64MB.

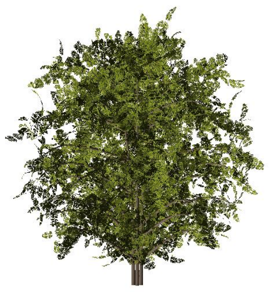
The tree models have been modelled with the Xfrog 2.1 program. They have been converted from the Xfrog format to standard obj format.

Figures 3.20 to 3.23 shows different simplified versions of the tree models used for the experiments. They have been simplified with the algorithm presented in this section. It can be observed that the images obtained maintain the appearance although their leaf number diminishes. In these images, we can observe an image composition of the simplified tree versions according to the distance to the viewer.

(a) 24.839 leaves.

(b) 18.629 leaves.

(c) 12.419 leaves.


(d) 6.209 leaves.

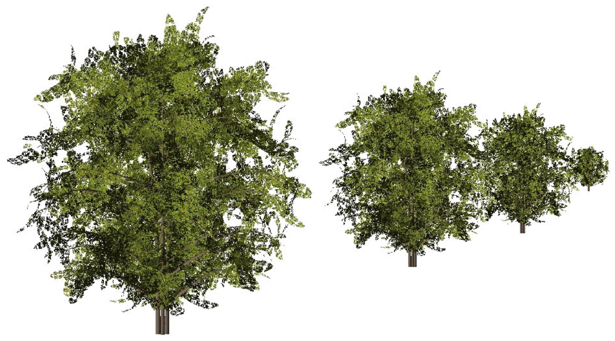
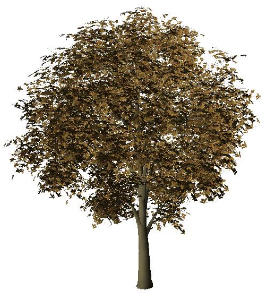
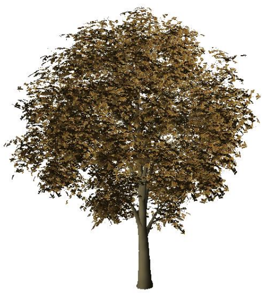
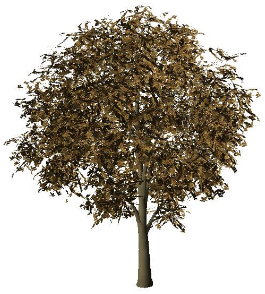
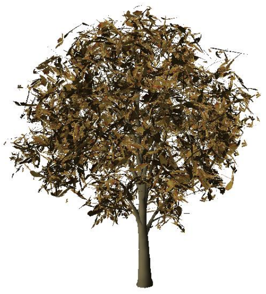

(e) Composition of the different aproximations following the distance-to-the-viewer criteria.

Figure 3.20: Results obtained for the tree model *Sorbus Aucuparia*.


(a) 29.534 leaves.


(b) 22.150 leaves.


(c) 14.767 leaves.


(d) 7.383 leaves.

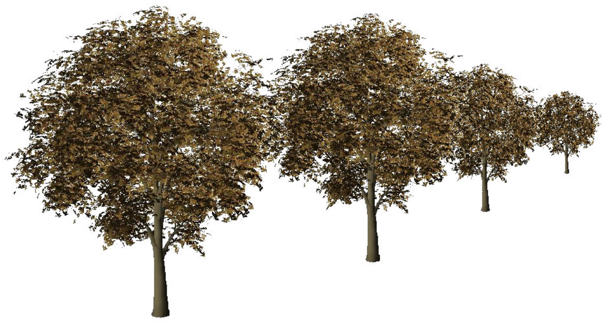
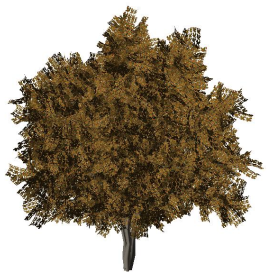
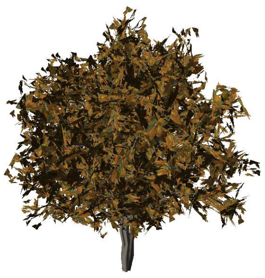

(e) Composition of the different aproximations following the distance-to-the-viewer criteria.

Figure 3.21: Results obtained for the tree model *Aesculus Hippocastanum.*


(a) 48.160 leaves.

(b) 36.120 leaves.

(c) 24.080 leaves.


(d) 12.040 leaves.

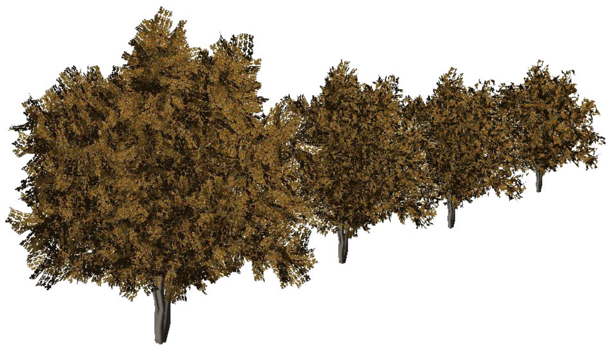
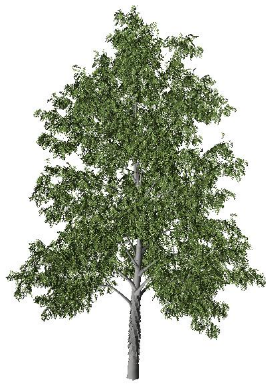
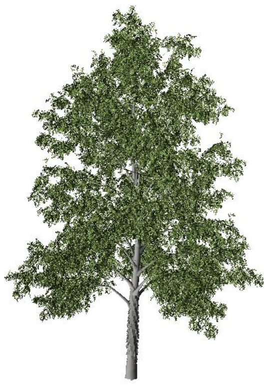
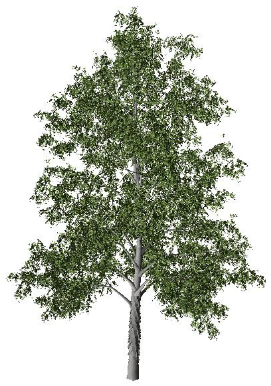
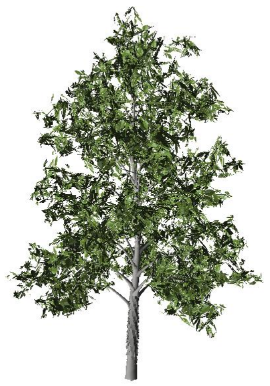

(e) Composition of the different aproximations following the distance-to-the-viewer criteria.

Figure 3.22: Results obtained for the tree model *Taxus Baccata*.


(a) 114.114 leaves.


(b) 85.585 leaves.


(c) 57.057 leaves.


(d) 28.528 leaves.

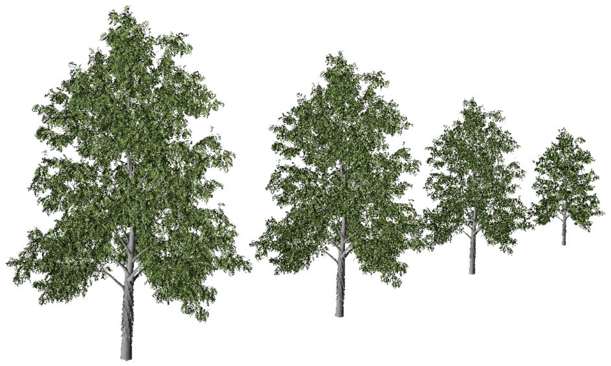

(e) Composition of the different aproximations following the distance-to-the-viewer criteria.

Figure 3.23: Results obtained for the tree model *Carya ovata*.

Chapter 4

LodStrips

4.1 Introduction

A common way to deal with the problem of rendering large 3D scenes is the use of multiresolution modeling techniques. According to Garland [GH97] a multiresolution model represents an object through a set of approximations at different levels of detail, as shown in figure 4.1, and allows us to recover any of them on demand. In recent years, multiresolution models have progressed substantially. In the early days, discrete models were employed in graphics applications due mainly to the low degree of complexity involved in implementing them. These models were based on a relatively small number of approximations (normally between 5 and 10) [ESV96]. Nevertheless, the increase in realism in graphics applications made it necessary to find solutions which were more exact in their approximations, which did not call for high storage costs and which had faster visualization times. This fact led to the appearance of continuous models, where two consecutive levels of detail only differ by a few polygons and which are capable of solving the problems of interactive visualization, progressive transmission, geometric compression and variable resolution. A comprehensive description of multiresolution models can be found in Ribelles et al. [RLB$^+$02]

Working with the current multiresolution models poses the problem of dealing with high level of detail extraction time and excessive storage cost. The continuous uniform resolution model we present noticeably improves existing models in terms of storage and visualization costs. The model is based entirely on optimized hardware primitives, triangle strips, and it is conceived in such a manner that mesh updating is fast and efficient.

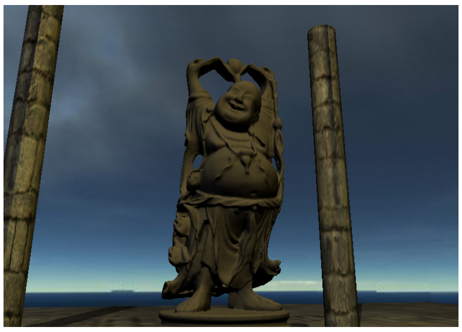


Figure 4.1: Happy_buddha model at the highest level of detail (543699 vertices and 31596 triangle strips) running in the Ogre 3D Engine.

4.2 Previous work

One of the first models to benefit from the triangle strip primitive was the one presented by Hoppe [Hop96], known as Progressive Meshes and included in Microsoft's DirectX library. The main drawback of Progressive Meshes was that it used triangles during the change of level of detail and during the rendering step. After this model appeared, many works were presented with the intention of improving its performance. In figure 4.2, we can observe three levels of detail of a model based on triangle strips primitives.

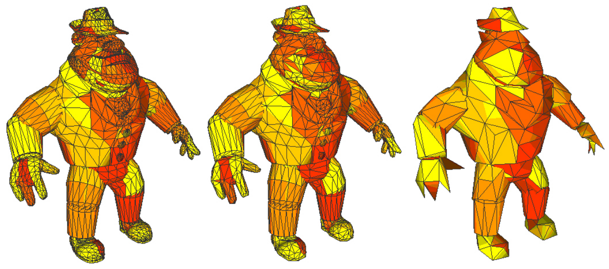


Figure 4.2: Three levels of detail from the AlCapone model.

The first multiresolution model to take full advantage of the connectivity information among triangles in a mesh was the model introduced by Ribelles et al. called MOM-Fan [RLR$^+$00], which used the triangle fan primitive in its data structures. The main problem of this model was the high number of degenerated triangles, although they were purged before the rendering stage. Another disadvantage of this model was that the average number of triangles in each triangle fan was small.

Later, El-Sana et al. [ESAV99] presented the Skip Strips model, which was the first model to maintain a data structure to store the strips that avoided the need to calculate them in real time. But this model still uses triangles to adjust the geometry at each level of detail.

The MTS model [BRR+02] uses triangle strips both as 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; only the ones that are modified when changing the level of detail are updated before being rendered.

Some time later Dstrips [SP03] appeared, which is a method that tries to maintain the initially calculated strips, modifying the existing ones and searching for new strips only when a specific zone of the model requires it.

4.3 Multiresolution model

The LodStrips 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:

$$M^r = V, S^r \tag{4.1}$$

where V is the set of all the vertices,

$$V = v_1, \ldots, v_n, v_i \in \Re^3 \tag{4.2}$$

and S^r is the set of all the triangle strips used for the representation of any of the different approximations that M^r stores.

The construction of this multiresolution model is made up of a set of different processes. First of all, as we are working with a model based on triangle strips, we will need to have a stripified mesh. We will also need to choose a simplification method for our 3D model, since we need the simplification information to generate the different levels of detail. With the information about the strips and the simplification sequence, the real construction of the model begins by filling the data structures with all the required information. Figure 4.3 offers the data flow diagram associated with the global construction process.

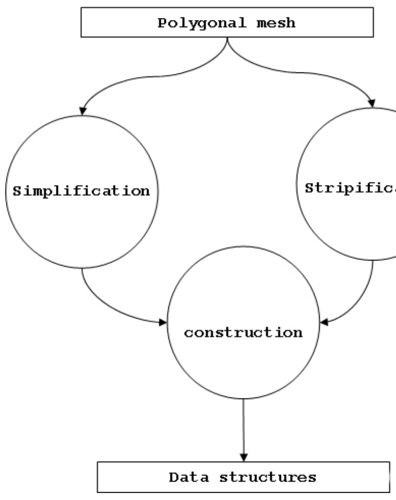


Figure 4.3: Model construction.

There are several mesh simplification methods [GH97, Lue01], but those based on iterative edge contractions are the ones employed on well-known multiresolution models such as [ESAV99, SP03, BRR$^+$02, RLR$^+$00, Ste01] and on our model.

The simplification process allows us to obtain versions of the input polygonal mesh at different levels of detail. The fundamental information that this process supplies consists of a sequence of collapses that are needed to simplify the polygonal mesh. For every collapse of the simplification process we need to know the vertices that split, the two triangles that disappear and the set of existing triangles that will be modified.

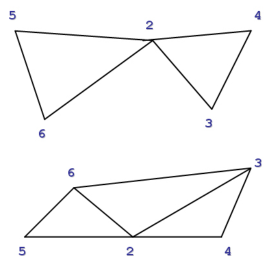
As the multiresolution model presented here is wholly based on triangle strip primitives, we will need to apply a stripification process consisting in converting polygonal meshes, which are geometrically composed of triangles, to triangle strips.

Many works can be found in the literature where the problem of converting a triangularized mesh into triangle strips is solved [ESV96, AHB90, XHM99]. This process can be carried out in either a dynamic or a static way. Dynamic stripification involves generating the triangle strips in real time, that is, for each level of detail new strips are generated. Static stripification entails creating triangle strips just once and then working with versions of these original strips through all levels of detail. In the multiresolution model proposed, static strips will be used since the cost of creating strips for every level of detail is too high. Figure 4.4 offers the resulting stripification of the Athena model.

Figure 4.4: Stripification example.

The main problem of static stripification is that strips tend to present vertex repetitions that do not add any geometric information to the final scene. Models like [ESAV99] solve this problem by applying filters in visualization, thus preventing those vertices from being sent at the moment of rendering. The approach we will follow is also based on the application of filters, as it runs a pre-process that detects them early on and then stores that information to eliminate them from the strips before visualizing them.

We have proven that most vertex repetitions follow patterns like $aa(a)+$ or $ab(ab)+$. Patterns $aa(a)+$ are replaced by aa, and $ab(ab)+$ by ab. Figure 4.5 shows an example of both kinds of patterns, and it can be observed how the final geometry of the strips is not altered after removing these patterns.



Strip: 5 6 **2 2 2** 3 4
Replace aa(a)+ by aa
Strip: 5 6 2 2 3 4

Strip: 5 6 **2 3 2 3** 4
Replace ab(ab)+ by ab
Strip: 5 6 2 3 4

Figure 4.5: Type of patterns removed by model data structures.

4.3.1 Construction

Once the information about the vertices to be simplified for each level of detail has been obtained from the simplification process, and the triangle strips at the highest level of detail have been generated, we then proceed to the construction of the model.

In this process, vertices are reordered according to the simplification order, that is, the first vertex to be collapsed will be zero, the second will be one, and so on. Obviously, it will also be necessary to modify the strips according to the changes made. This step also stores the ordered vertices and the triangle strips within the model data structures. Finally, and as we have mentioned before, the method takes into account the existence of degenerated triangles and applies filters to avoid their appearance in the triangle strips.

With the information gathered in the previous step it would already be possible to build a multiresolution model that traverses through the levels of detail. However, whenever a change in level of detail occured, it would be necessary to search among all the strips for the vertices that would collapse, and this operation would have a high cost. Thus, a further process is required that pre-computes and stores this information in another data structure.

This process computes the strips that change for each level of detail and the exact location of the vertex to be simplified in every strip. This allows the levels of detail in the model to be crossed rapidly, offering optimum performance. This is the information that enables a fast level of detail extraction time and makes this multiresolution model quick and efficient.

4.4 Data structure

Only two data structures are needed to visualize a polygonal mesh at the highest level of detail: *Strips* and *Vertices*. *Vertices* stores the 3D coordinates for each vertex in the mesh, and *Strips* is a set of triangle strips where each strip contains a sequence of indexes to *Vertices*.

To change the level of detail, we also need to store the vertex that will be collapsed for each LOD. Thus, for each vertex in the *Vertices* data structure we also store the vertex where the collapse will take place. Figure 4.6 shows a scheme of the model data structures.

In order to avoid the problem of having to search the vertex to be collapsed in each strip, we first store the strip that change in the data structures

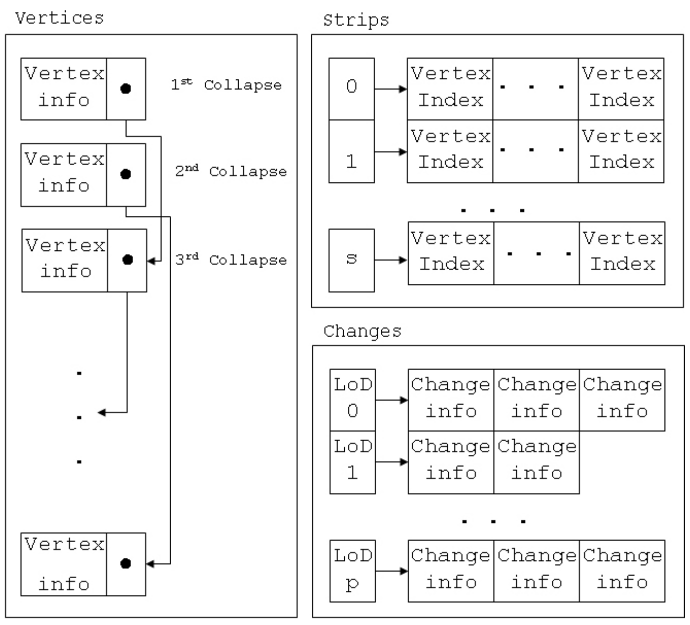


Figure 4.6: LodStrips data structures.

and then store the exact position of the vertex to be collapsed in the triangle strip. However, as mentioned in the previous section, 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.

In summary, we need some additional data structures to support the aforementioned aspects, that is to say, to index the vertex to be collapsed and to remove the most frequent patterns. This information is stored in *Change Info* within the *Changes* data structure.

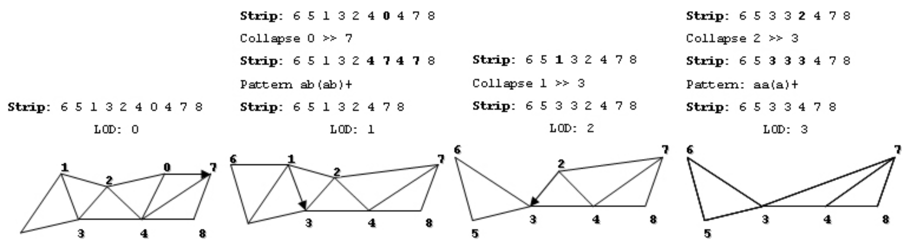


Figure 4.7: Model construction example.

Figure 4.7 offers three steps of the construction process of a triangle strip.

We will start having *Strips* and *Vertices* filled with the suitable information, while the data structure *Changes* will be empty. As we mentioned before, the level of detail will be related to the vertex that is collapsed in each LOD change, and *Vertices* offers the vertex it collapses to. This way, we know that vertex 0 collapses with vertex 7. We calculate that vertex 0 is located in position 6 and only appears once. The collapse of this vertex compels the appearance of a vertex repetition in position 5, allowing us to eliminate two vertices of the strip. With all this information we can fill the *Changes* data structures for all the LOD changes.

4.4.1 Level of detail extraction

The level of detail extaction algorithm works by changing the vertices of all strips. Algorithm 4.8 offers a pseudo algorithm to move from LOD n to LOD $n+1$. It consists in replacing the vertex n by the vertex it collapses to in every strip where it appears, and removing vertex repetitions following both patterns.

```
for lod = currentLOD to demandedLOD
  //Compute the number of changes to apply in this lod
  nChanges = TotalChanges(lod);
  //Collapse vertices and remove patterns calculated in the pre-process
  for i = 0 to nChanges - 1
    CollapseVertices(Changes[lod][i]);
    RemovePatterns(Changes[lod][i]);
  end for i
end for lod
```

Figure 4.8: Level of detail extraction from a LOD to a coarser one.

4.4.2 Optimizations in visualization

Every multiresolution strip has two representations with the same information. The first one is the already mentioned *Strips*, a data structure with a constant time in insertions and deletions. The second one is *visStrips*, which is efficient and fast in access and allows us to exploit coherence in visualization. This representation can be allocated either in the main memory or directly in the graphics hardware, producing great acceleration.

The information stored in *visStrips* must be updated according to the changes made in *Strips*. Thus, every time a strip is modified in the level of detail extraction process, it is necessary to communicate it to the visualization process through the *stripChanged* flag array. The method for visulization is presented in figure 4.9.

```
for  i = 0 to lStrips.size()
  //Update visStrips when proceed
  if (stripChanged[i])
    visStrips[i]=lStrips[i];
  //Send strips to GPU
  visStrips[i].Draw();
end for i
```

Figure 4.9: Visualization algorithm.

4.5 Results

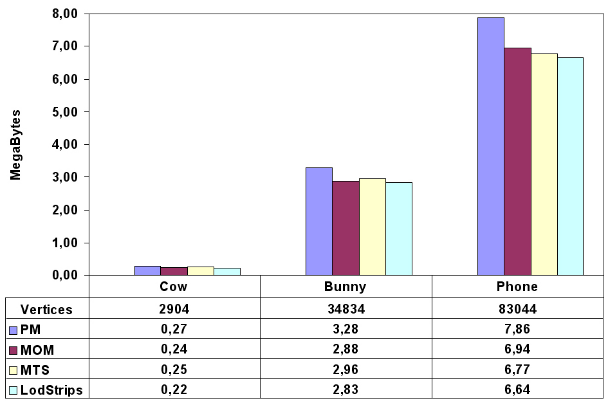
This model has been submitted to several tests in order to analyze the main features that must be taken into account when selecting a multiresolution model. To carry out the tests, some well-known meshes from the Stanford 3D Scanning Repository were taken as a reference in order to make it easier to compare this model with other well-developed models. The spatial cost of these models is shown in table 4.1. The computer where the tests were conducted was a PC with an Intel Pentium Xeon 2.8 GHz processor, 1024 Mb RAM and an NVIDIA GeForce FX 6600 256 Mb graphics card. C++ was employed for the implementation, using the graphics library OpenGL.

	Cow	Bunny	Dragon	Phone	Buddha
Vertices	2.904	34.834	54.294	83.044	543.699
Strips	551	6.194	8.799	1.747	31.596
Mb	0,17	2,64	4,01	5,08	35,51

Table 4.1: Spatial cost of some models.

Figure 4.10 shows a comparison of spatial costs among the most important present-day continuous uniform resolution models: PM [Hop96], MOM [RLR+00] and MTS [BRR+02]. As can be observed, the presented model offers the best spatial cost. On average, the model presented here fits in 1.5 times the original mesh in triangles.

In figures from 4.11(a) to 4.14(a), a composition of three images is shown, being the first one the stripification of the model at the highest level of detail,



	Cow	Bunny	Phone
Vertices	2904	34834	83044
🟦 PM	0,27	3,28	7,86
🟥 MOM	0,24	2,88	6,94
🟨 MTS	0,25	2,96	6,77
🟦 LodStrips	0,22	2,83	6,64

Figure 4.10: Spatial cost comparison.

the second one the strips at the lowest level of detail and the last one an illuminated version of the original object.

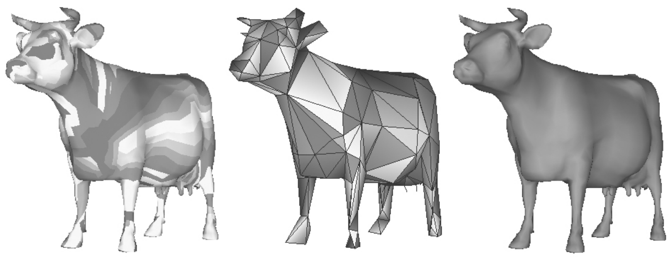
In continuous multiresolution models, level of detail management entails two fundamental tasks: level of detail extraction and visualization of the resultant geometry. We can observe in figures from 4.11(b) to 4.14(b) that the model presented here offers a low extraction time. This is mainly due to the effect of using coherence in the extraction algorithm and also to the implementation of efficient data structures that manage the level of detail. The experiment consists in extracting one hundred levels of detail between 0 and 1, where 0 represents the highest lod and 1 the lowest one.

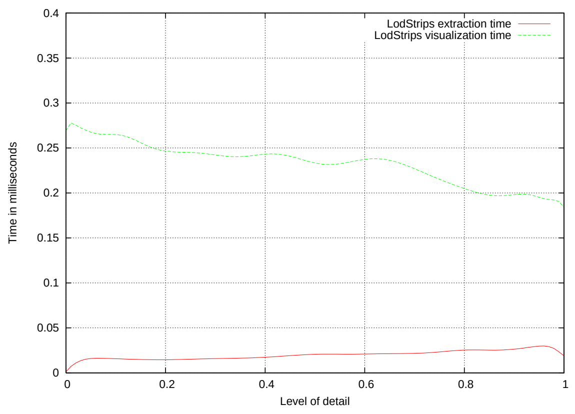
Finally, in figures from 4.11(c) to 4.14(c) a comparison of the vertices sent to the GPU is also offered.

4.6 Conclusions

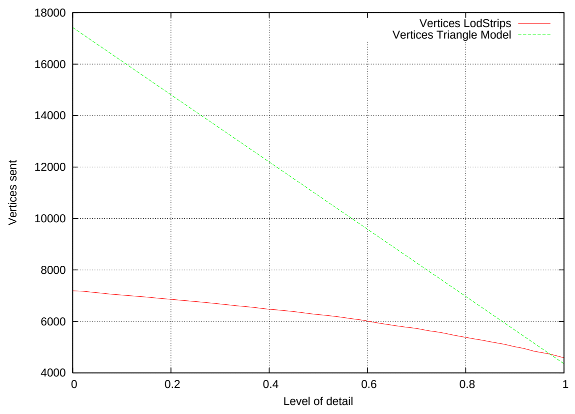
The LodStrips model offers many advantages and it should be underlined that it is a model with only three simple data structures and it is easy to implement. Moreover, it offers a fast LOD extraction which allows us to obtain smooth transitions between LODs, as well as very good rendering times, because extraction is usually an important part of the total rendering time. This model is wholly based on triangle strips, which leads to an important reduction in storage and rendering costs.

Moreover, this model features: an easy adaptation to the graphics hardware, optimized hardware primitives, vertex cache exploitation and low spatial cost.


(a) Images of the model.

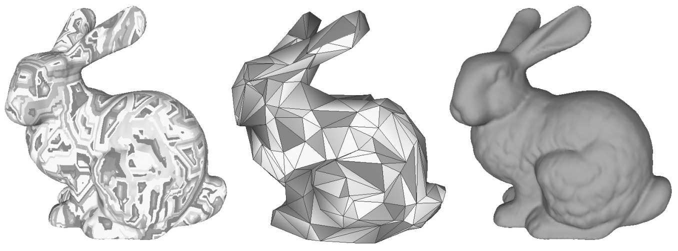


(b) Extraction and visualization time.

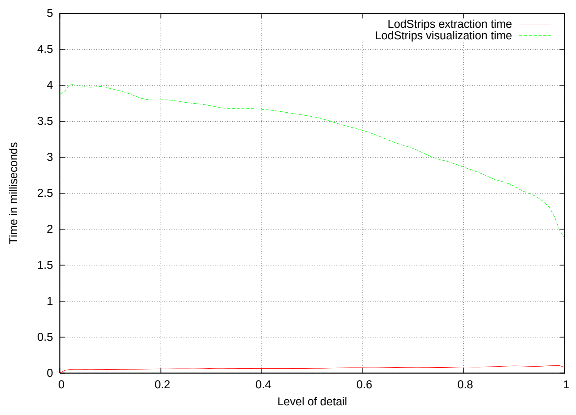


(c) Comparison of the vertices sent, from the highest LOD (1) to the lowest (0).

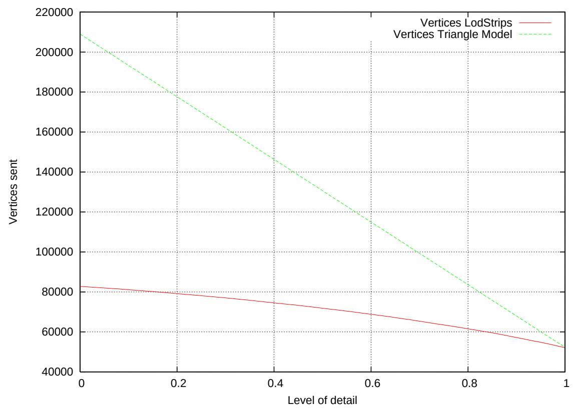
Figure 4.11: Results for the cow model.



(a) Images of the model.

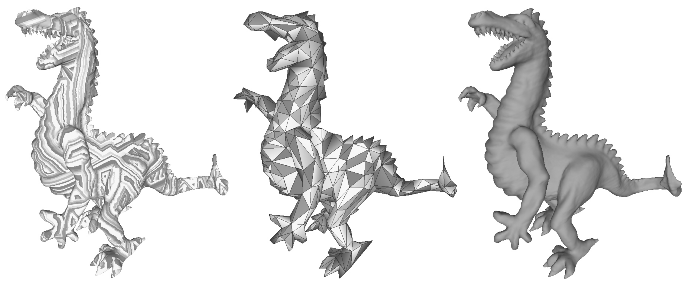


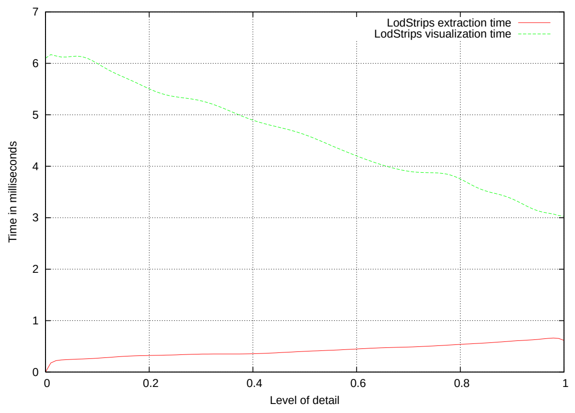
(b) Extraction and visualization time.



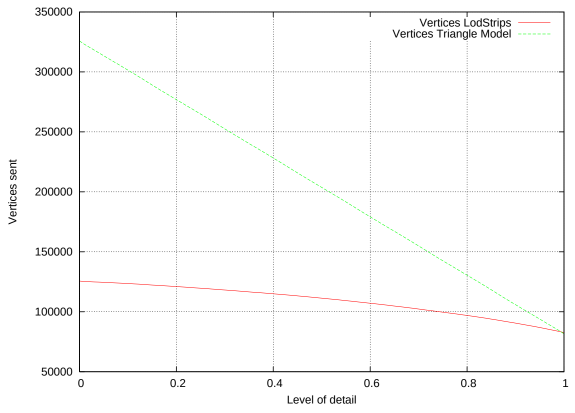
(c) Comparison of the vertices sent, from the highest LOD (1) to the lowest (0).

Figure 4.12: Results for the bunny model.


(a) Images of the model.

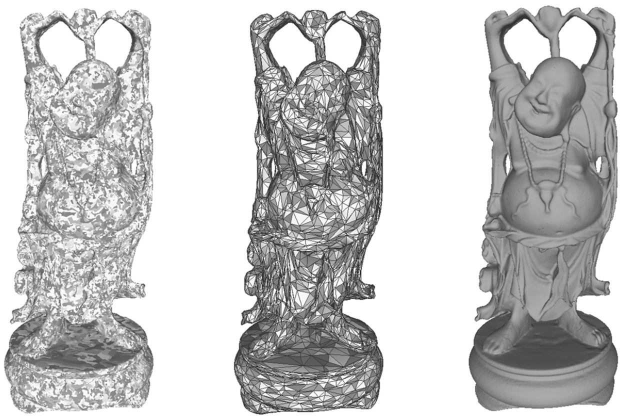


(b) Extraction and visualization time.

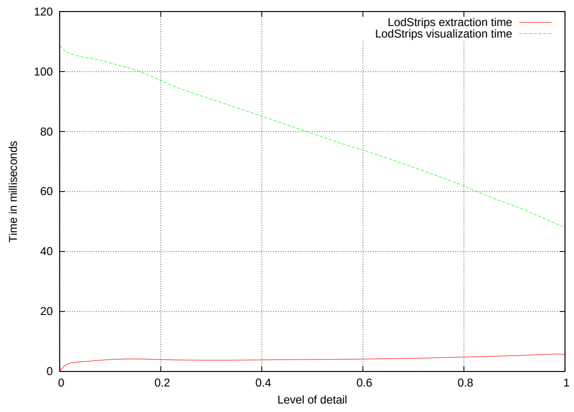


(c) Comparison of the vertices sent, from the highest LOD (1) to the lowest (0).

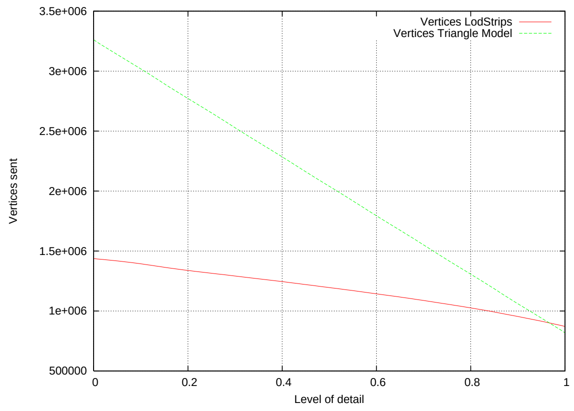
Figure 4.13: Results for the dragon model.



(a) Images of the model.



(b) Extraction and visualization time.



(c) Comparison of the vertices sent, from the highest LOD (1) to the lowest (0).

Figure 4.14: Results for the happy buddha model.

Chapter 5

LodTrees

5.1 Introduction

In general, multiresolution models are built from the original geometry and from the approximations obtained through the simplification process. In section 3.4, we analyzed the automatic foliage simplification algorithm, which offers as a result the simplification sequence of leaves for a given model. This section introduces the use of the simplification sequence and related algorithms for the visualization of foliage models in real time.

5.2 Previous work

Research aimed at vegetation can be divided in two major fields: the generation of plants and trees, and their visualization. Vegetation modelling has been explored extensively. The most important works in this field are Lindermayer-systems [PL90], used for generating realistic models of trees. Other solutions combine grammar based modelling with a graph description [LD99]. Apart from the great number of works that have appeared in the literature, some commercial applications have been developed for modelling trees. Some of the most important are OnyxTree (www.onyxtree.com), AMAP (www.bionatics.com), Xfrog (www.greenworks.de) and SpeedTreeRT (www.idvinc.com).

The analysis of previous work related with our approach can be divided as geometric representation and image-based rendering methods.

Geometric representation: Level of detail rendering [Jak00] is one of the most popular methods to reduce the complexity of polygonal data sets in a smart manner.

The continuous multiresolution models presented thus far deal with general meshes and cannot be applied effectively to such scenes. Hoppe [Hop97], Xia [XV96] and El-Sana [ESV99] use methods based on the union of pairs of vertices in the construction process simplification. Luebke [LE97] uses a method based on vertex clustering: a set of vertices is collapsed into one vertex. These methods can not process the foliage without degradation of appearance [DCSD02].

Another technique in interactive visualization of complex plant models uses pointbased rendering based on the idea of substituting the basic primitive triangle by points or lines. Reeves and Blau [RB85] rendered trees using small disks representing the foliage, and Weber and Penn [WP95] used sets of points for the leaves and lines for the tree skeleton. Stamminger and Dettrakis [SD01] visualize plants with a random sample set of points. One of the most recent works in this field has been presented by Deussen et al. [DCSD02]. Their approach combines geometry with points and lines.

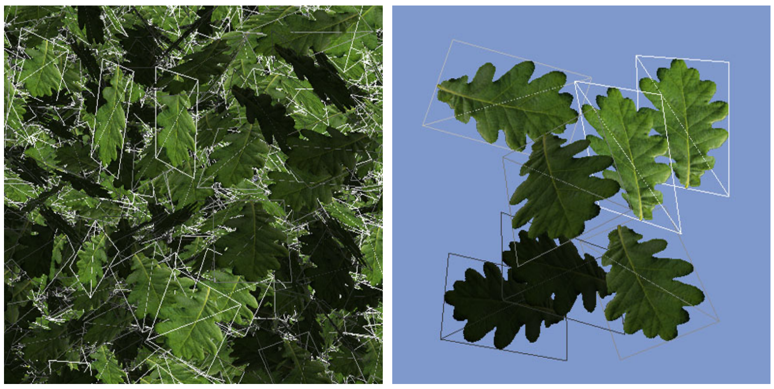
Image-Based Rendering methods: Billboarding is one of the most frequently used techniques due to its simplicity. The trees are reduced to images textured on polygons, which always maintain their orientation towards the observer. However this technique has great deficiencies, because the models are represented in two dimensions. When the observer moves toward the object, the lack of details produces a loss of realism in the scene.

Layered depth images [SSHS98], LDI, store in each pixel of the image a 2D array of depth pixels. In each depth pixel are stored, in proximity order to the point of view, the surfaces that appear in that image. But the LDI files created for trees are excessively large. Another similar method however using Z-buffers is presented by Max [MO95].

Jakulin [Jak00] presents a method based on images with alpha-blended textured polygons. Lluch et al. [LCV04] present a method based on a hierarchy of images obtained from pre-processing the botanical tree structure (a L-system) and storing the information in a texture data tree.

5.3 Multiresolution model for foliage

The construction of the multiresolution model for a tree entails two different processes. On the one hand, the trunk of the tree is formed by a continuous mesh, classified inside the arbitrary surfaces, and its multiresolution model will be built using the LodStrips model we have just presented. On the other hand, as tree foliage is represented by a set of isolated polygons where each leaf is formed by the union of two triangles (see figure 5.1), most of the existing multiresolution models are not appropriate for its visualization [DCSD02] [Jak00]. The multiresolution model for foliage has been developed in order to allow the interactive visualization of the crown of trees.



(a) Detail of the crown. (b) Detail of the leaves.

Figure 5.1: Detail of the foliage in a tree.

Let F be the original crown of the tree, and F^r its multiresolution representation. Considering that each leaf is composed of a pair of triangles that determine the quadrilateral where the texture is applied, F and F^r can be defined as:

$$F = \{V, L\} \qquad F^r = \{V^r, L^r\} \tag{5.1}$$

where V and L are the set of vertices and leaves that form the original object, and V^r and L^r are the set of all vertices and leaves used to represent any of the different approximations stored in F^r.

As mentioned before, a multiresolution object is usually built from the original geometry and the sequence of simplifications that reduces the detail of the object. In this case, the simplifications will be those obtained through the simplification algorithm explained in section 3.4. In order to construct the multiresolution representation with n approximations, we will need $n-1$ simplification steps. This way, starting from the maximum level of detail, F_0, we will be obtain the sequence of approximations $F_1, F_2, \ldots, F_{n-1}$. Finally, we can conclude that the original model will be formed by the vertices and leaves of the representation with the maximum detail:

$$V = V_0 \qquad L = L_0 \tag{5.2}$$

and, considering the characteristics of the simplification method employed that does not add new vertices during the process of obtaining the different approximations, the multiresolution representation F^r can be expressed as:

$$V^r = V_0 \tag{5.3}$$

$$L^r = L_0 \cup l_0 \cup ... \cup l_{n-2} = L_0 \ \cup \ \bigcup_{i=0}^{n-2} l_i, n \geq 1 \tag{5.4}$$

where l_i is the leaf added to F_i to create the approximation F_{i+1}.

In order to change the level of detail, two geometric operations have been defined, as shown in figure 5.2:

- **Leaf collapse**: Decreases the level of detail of a given representation. This operation replaces two leaves by a new one, diminishing the number of leaves of the crown. Following the example in figure 5.2, leafs l_t and l_u will be replaced with l_s in the new level of detail.
- **Leaf split**: It is the inverse operation of the previous one. It replaces a leaf by the two leaves that it represents, increasing the level of detail. Following the same example, l_s will be replaced with l_t and l_u.

This multiresolution model has the following features:

- Input data contains the polygons that form the crown: a set of isolated quadrilaterals, where each one is represented with two triangles.
- Stores attributes like normals and texture coordinates.
- The model is based on the automatic foliage simplification algorithm. The basic simplification operation is the *leaves collapse*.
- The model allows real time visualization using levels of detail with uniform resolution.

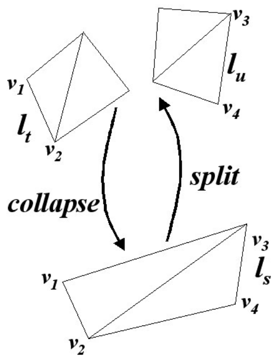


Figure 5.2: Example of leaf *collapse* and *split* operation.

5.4 Data Structure

The data of the multiresolution model are arranged in *forests* of binary trees. This structure reflects the way the simplification operation works, which establishes a hierarchical relation between leaves (see figure 5.3). Each new leaf, l_s, is created from two existing ones, l_t and l_u. Considering each leaf as a node of the foliage structure, l_s represents the parent node of l_t and l_u.

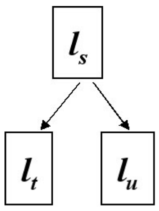


Figure 5.3: Hierarchical relationship between leaves conditioned by the *collapse* operation.

This hierarchical structure is constructed from the bottom up. Thus, we firstly process F_0, and then the sequence of $n-1$ simplification steps where a collapse is represented by a new node in the structure. Let's suppose a foliage F_0 is initially made up of 9 leaves ($|L_0| = 9$). The simplification sequence produces a new diminished foliage F_{n-1} made up of 3 leaves with $n-1$ steps. Let's suppose the following simplification operations:

- Leaf 10 is created from collapsing leaves 6 and 7: nodes 6 and 7 will be children of the new node labeled 10.
- Leaf 11 is created from collapsing leaves 1 and 2, proceeding in the same way.

- Leaf 12 is created from collapsing leaves 5 and 10.
- Leaf 13 is created from collapsing leaves 11 and 3.
- Leaf 14 is created from collapsing leaves 4 and 12.
- Leaf 15 is created from collapsing leaves 8 and 9.

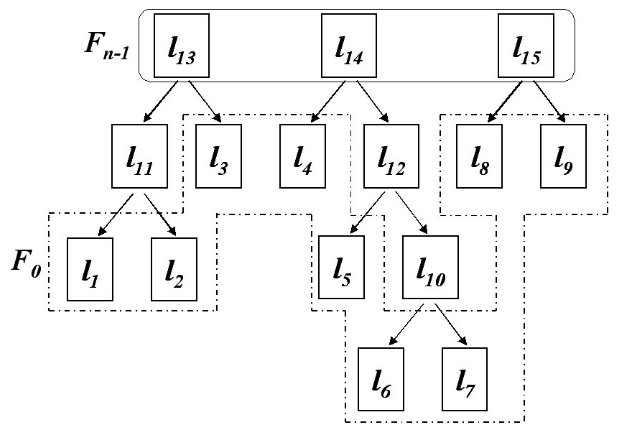


Figure 5.4: Example of data structure F^r.

The resulting structure F^r is shown in figure 5.4. The leaves of the whole forest represent the most detailed object F_0 and the roots of each tree denote the leaves of the lowest level of detail F_{n-1}.

Data structures

The data structures required to store the hierarchy of leaves are shown in figure 5.5.

The geometric data that forms F^r are stored in the *FoliageData* structure. *Leaf* data structure represents a node, and stores information about its geometry and the hierarchical relationships with other leaves.

Foliage structure represents the sequence of leaves that composes a certain level of detail. This structure stores a pointer to all data needed to represent F^r, and a double linked list of visible leaves for a given foliage resolution.

The design of these data structures are oriented towards the creation of ecosystems. In nature, it is common to find the same plant species several times in the same habitat. The organisation of data shown in figure 5.5 permits the representation of the same species as different objects with different levels of detail, but storing just once the geometric data that compose it.

```
struct Vertex {
  float coordinates[3];
};
struct Leaf {
  int VLeaf[4];
  int parent;
  int lchild;
  int rchild;
};
struct FoliageData {
  struct Vertex *VerticesFoliage;
  struct Leaf *Leaves;
};
struct ActiveLeaves {
  int next;
  int prev;
};
struct Foliage {
  FoliageData *MyFoliage;
  ActiveLeaves *Active;
};
```

Figure 5.5: Basic data structure of the multiresolution model.

Storage cost

For the study of the storage cost, we assume that the storing cost of an integer, a pointer or a real is a word. The original model will consist of three real numbers by vertex, and four indices by leaf:

$$4|L_0| + 3|V_0| \tag{5.5}$$

As each leaf is formed by four independent vertices,

$$|L_0| = \frac{|V_0|}{4} \tag{5.6}$$

the total storage cost of the original model F is $4|V_0|$.

For calculating the storing cost of a representation F^r, the *Leaf* and *Vertex* elements will cost 7 and 3 words respectively. This way, the data stored will cost:

$$7|L^r| + 3|V^r| \tag{5.7}$$

where $|L^r|$ and $|V^r|$ are the number of leaves and vertices stored in the multiresolution structure. The number of leaves for a given resolution level are the number of leaves of the maximum level plus those coming from collapse operations, so:

$$|V^r| = |V_0| \qquad |L^r| = |L_0| + (n-1) \tag{5.8}$$

The cost is influenced by the number n of approximations stored in the representation F^r. In the best case, the model only stores one approximation, $n = 1$. On the contrary, in the worst case the model allows approximation formed by a single leaf, storing $n = |L_0|$ approximations. In this case, we will have to add $|L_0| - 1$ new leaves to the original representation, as from two existing leaves we will generate a new one.

In equation 5.6 it was shown that the number of initial leaves corresponds with a quarter of the number of vertices stored. Thus, in the best case, the number of leaves and vertices stored will be:

$$n = 1 \quad \longrightarrow |L^r| = |L_0|, \qquad |V^r| = |V_0| \tag{5.9}$$

with a cost of:

$$7|L_0| + 3|V_0| = 7\frac{|V_0|}{4} + 3|V_0| \approx 4,75|V_0| \tag{5.10}$$

In the worst case, $|L_0|$ approximations will be stored and the total number of vertices and leaves will be:

$$n = |L_0| \quad \longrightarrow |L^r| = 2|L_0| - 1, \qquad |V^r| = |V_0| \tag{5.11}$$

with a cost of:

$$7(2|L_0| - 1) + 3|V_0| = 7(2\frac{|V_0|}{4} - 1) + 3|V_0| \approx 6,5|V_0| \tag{5.12}$$

With all th equations shown in this section, we may say that, in summary, the storage cost of the model is O($|V_0|$).

Results

The experiments have been carried out in an environment with the following characteristics:

Hardware. Dual Pentium, Intel Xeon at 1.8 Gz with 512 Mb of RAM. graphics card Quadro4 700XGL with 64 MB.

Software. The implementation has been developed in *C++* and the graphics library *OpenGL*.

Data. Trees have been obtained with the *Xfrog* 2.1 program (http://www.greenworks.de/), proposed by [LD97].

Table 5.1 summarizes the characteristics of the foliage of the trees used in the experiments, and the storing cost is shown in table 5.2.

For each tree, table 5.1 shows the number of vertices and leaves that form the original model. It is shown its storing cost (in Mb) assuming a data structure based on a vertices list and a leaf list. It is assumed that a word (integer, float or pointer) has a 4 bytes cost.

Table 5.2 shows the data of the multiresolution representation. The storing cost of this representation is between 1,56 and 1,62 times higher than the original model.

	Original			
	Vertices	Polygons	Leaves	MB.
Betula Populifolia	32.560	16.280	8.140	0'50
English Oak	81.504	40.752	20.376	1'24
Sorbus Aucuparia	99.360	49.680	24.840	1'52
A. Hippocastanum	118.140	59.070	29.535	1'80
Taxus Baccata	192.640	96.320	48.160	2'94
C. Lawsoniana	194.064	97.032	48.516	2'96
Fagus Sylvatica	194.784	97.392	48.696	2'97
Carya ovata	456.456	228.228	114.114	6'96

Table 5.1: Some trees used in the expermiments, with their characteristics and original storing cost.

	LodTrees			Ratio
	n	Leaves	MB.	
Betula Populifolia	8.019	16.158	0,80	1,62
English Oak	20.090	40.465	2,01	1,62
Sorbus Aucuparia	24.498	49.337	2,45	1,62
A. Hippocastanum	25.145	54.679	2,81	1,56
Taxus Baccata	47.520	95.679	4,76	1,62
C. Lawsoniana	47.651	96.166	4,79	1,62
Fagus Sylvatica	48.039	96.734	4'81	1,62
Carya ovata	110.099	224.212	11'21	1,61

Table 5.2: Trees used in the experiments, with their characteristics and storing cost.

5.5 Level of detail extraction

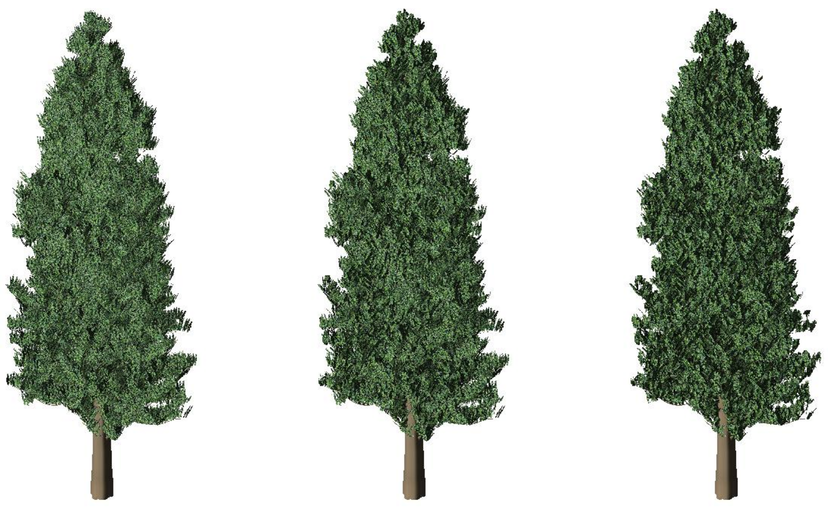
The multiresolution model representation allows us to easily visualize and dynamically vary the level of detail of the foliage, thanks to the data structure employed.

Figure 5.6 shows the algorithm that can be used to adapt dynamically the number of leaves of the foliage. Initially, the application calculates the number of leaves that should form the foliage based on the distance to the observer and the current number of leaves.

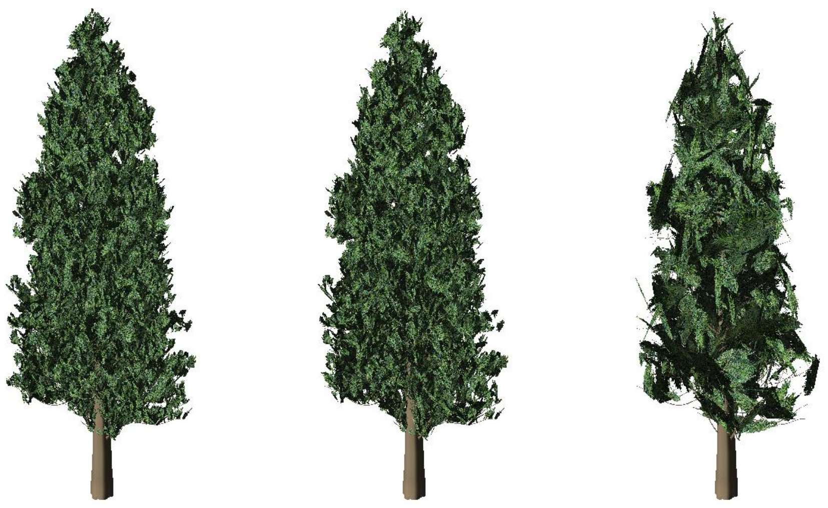
```
// Compute the number of leaves to visualize
nleaves = Number_Leaves_LoD (foliage, camera);
// Compute the number of leaves currently visualized
nl_active = Current_Number_Leaves (foliage);
// Compute the max number of the visualized leaves
lastleaf = Max_Number_Active_Leaf (foliage);

if nleaves < nl_active then
  // Diminish the number of leaves to visualize
  while ( nl_active - nleaves) > 0 do
    Collapse (lastleaf);
    nl_active = nh_active - 1;
  end while
else
  // Increase the number of leaves to visualize
  while (nleaves - nl_active) > 0 do
    Split (lastleaf);
    nl_active = nl_active + 1;
  end while
end if
```

Figure 5.6: Algorithm of the level of detail extraction.

The collapse operation is equivalent to diminishing in one the number of active leaves (the algorithm adds two leaves but removes one). In the same way, the split operation involves increasing in one the number of leaves visualized. These operations are applied sequentially until the desired level of detail is reached, increasing or diminishing the number of leaves according to the necessities of the application. The simplification sequence establishes the order of the operations, so that, at every moment, the number of leaves that forms the foliage is adapted efficiently. This algorithm works increasing or decreasing the level of detail uniformly, adding or removing one leaf at a time. At figure 5.7 different representations of the same object are shown, varying uniformly the number of leaves.



(a) 48.515 leaves. (b) 32.515 leaves. (c) 20.515 leaves.



(d) 8.515 leaves. (e) 4.515 leaves. (f) 2.866 leaves.

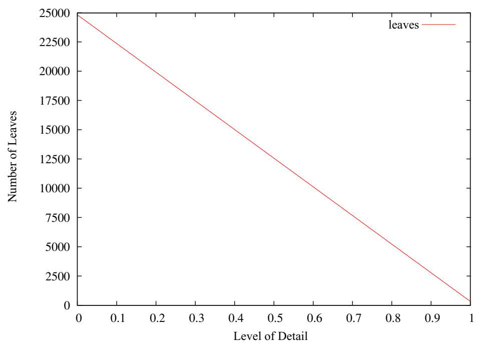
Figure 5.7: Different approximations of the *Chamaecyparis Lawsoniana* tree.

5.6 Results

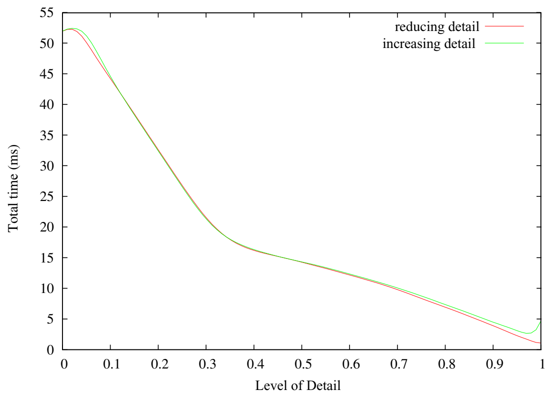
The experiments have been carried out using the same conditions as in section 5.4. In this case, the obtained approximations vary in the interval $[0, 1]$, where 0 represents the most detailed approximation and 1 the worst one. The number of leaves that form the different levels of detail vary in a lineal form following a *step*, defined as a number proportional to the difference that exists between the approximations F_{n-1} y F_0.

We have defined 30 intervals. In this case, the step changes between the different tree models we have used in the tests. Results are shown from figure 5.8 to 5.11.

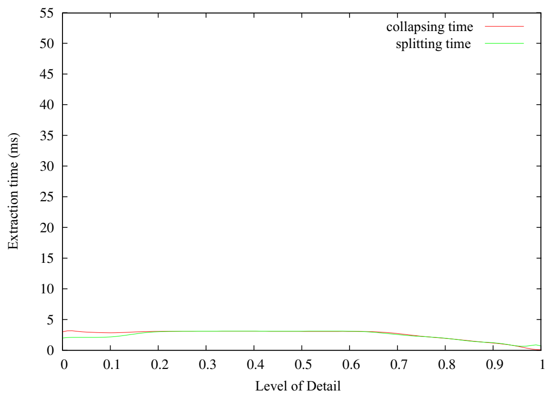
(a) *Sorbus Aucuparia.*



(b) Number of leaves in a level of detail.



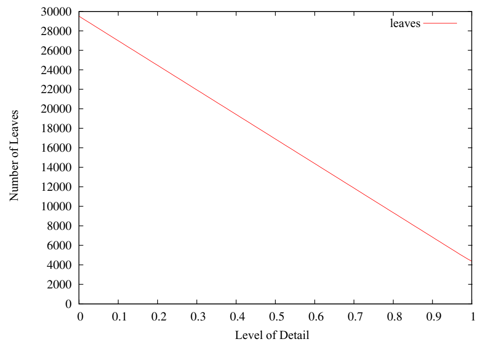
(c) Total time.



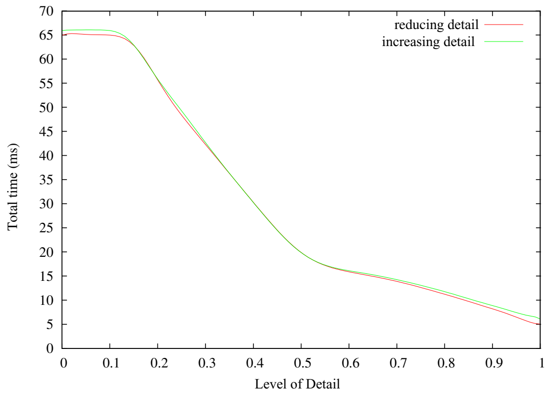
(d) Extraction time.

Figure 5.8: Results obtained for the tree *Sorbus Aucuparia.*

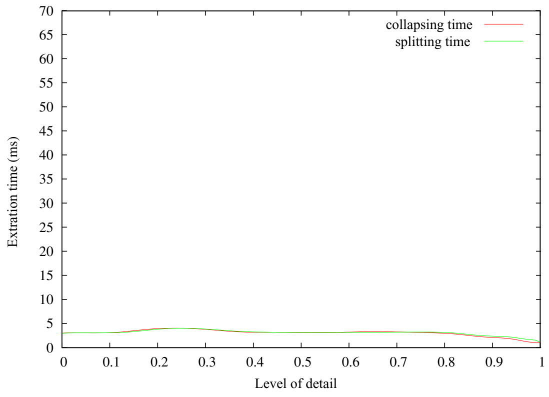
(a) *Aesculus Hippocastanum.*



(b) Number of leaves in a level of detail.

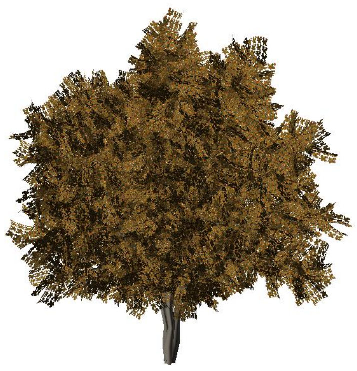


(c) Total time.

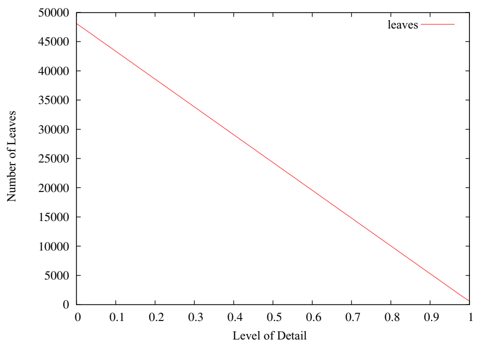


(d) Extraction time.

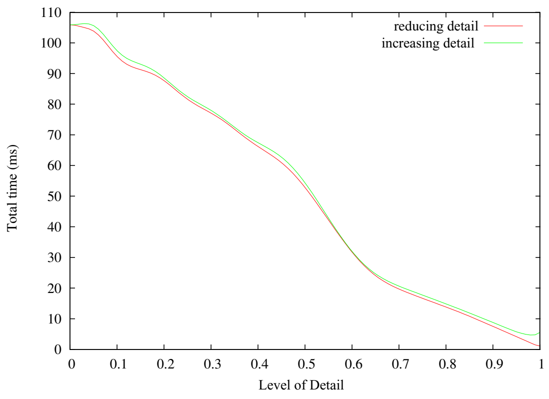
Figure 5.9: Results obtained for the tree *Aesculus Hippocastanum.*



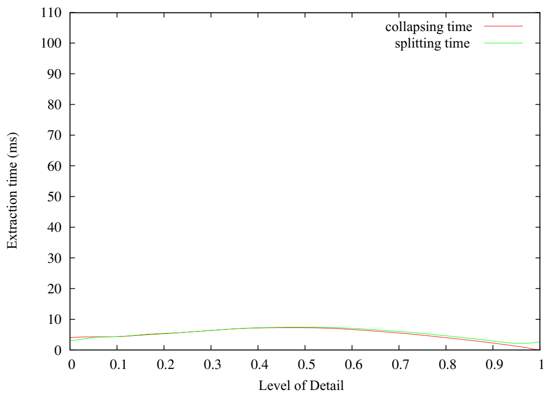
(a) *Taxus Baccata.*



(b) Number of leaves in a level of detail.

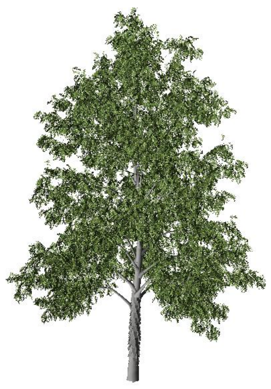


(c) Total time.

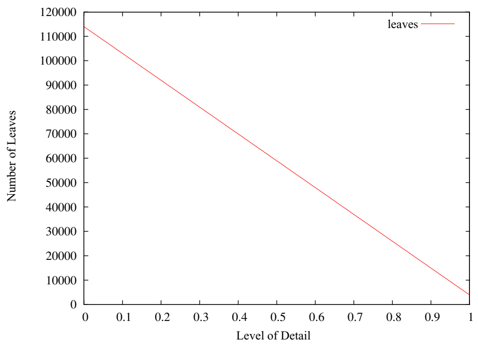


(d) Extraction time.

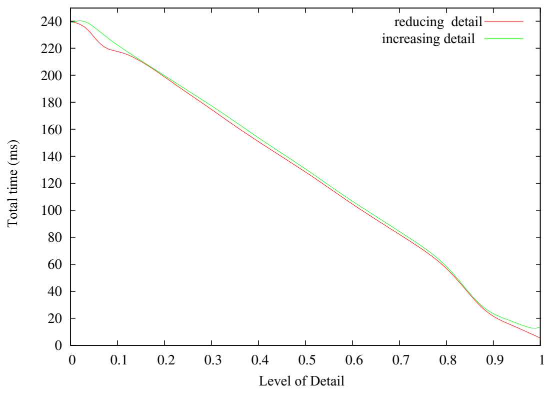
Figure 5.10: Results obtained for the tree *Taxus Baccata.*



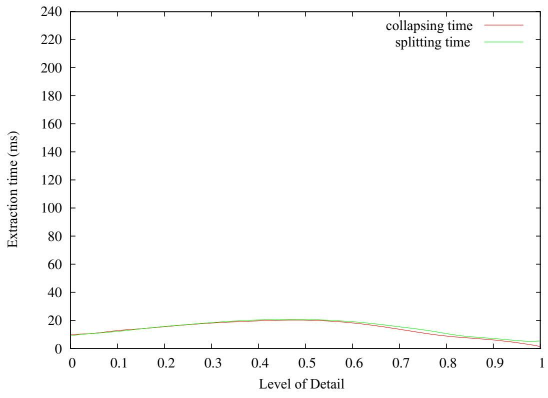
(a) *Carya ovata*.



(b) Number of leaves in a level of detail.



(c) Total time.



(d) Extraction time.

Figure 5.11: Results obtained for the tree *Carya ovata*.

5.7 Conclusions

The algorithm and its data structure are designed to visualize ecosystems where the same species of a tree appears several times in the same habitat. The data structure allows us to visualize many trees with different levels of detail, sharing the same data. The multiresolution model for foliage has proven to be useful to render foliage with uniform resolution, where the level of detail can be changed in real time and the number of leaves can be adjusted to meet the requirements of the application.

Bibliography

[AHB90] K. Akeley, P. Haeberli, and D. Burns. The tomesh.c program. Technical Report SGI Developer's Toolbox CD, Silicon Graphics, 1990.

[BD02] C. Beeson and J. Demer. Nvtristrip, library version. `http://developer.nvidia.com/view.asp?IO=nvtristrip_library` , 2002.

[BRR$^+$01] O. Belmonte, I. Remolar, J. Ribelles, M. Chover, C. Rebollo, and M. Fernandez. Multiresolution triangle strips. In *Proceedings IASTED Invernational Conference on Visualization, Imaging and Image Processing (VIIP 2001*, pages 182–187, 2001.

[BRR$^+$02] O. Belmonte, I. Remolar, J. Ribelles, M. Chover, and M. Fernández. Efficient implementation of multiresolution triangle strips. In *ICCS '02: Proceedings of the International Conference on Computational Science-Part II*, pages 111–120, London, UK, 2002. Springer-Verlag.

[BRRC00] O. Belmonte, J. Ribelles, I. Remolar, and M. Chover. Búsqueda de tiras de triángulos guiadas por un criterio de simplificación. In *Actas del X Congreso Español de Informática Gráfica (CEIG 2000)*, pages 51–64, 2000.

[CCMS96] A. Ciampalini, P. Cignoni, C. Montani, and R. Scopigno. Multiresolution decimation based on global error. Technical report, Centre National de la Recherche Scientifique, Paris, France, 1996.

[DCSD02] O. Deussen, C. Colditz, M. Stamminger, and G. Drettakis. Interactive visualization of complex plant ecosystems. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 219–226. IEEE Computer Society, 2002.

[ESAV99] J. El-Sana, E. Azanli, and A. Varshney. Skip strips: maintaining triangle strips for view-dependent rendering. In *VIS '99: Proceedings of the conference on Visualization '99*, pages 131–138. IEEE Computer Society Press, 1999.

[ESV96] F. Evans, S. Skiena, and A. Varshney. Optimizing triangle strips for fast rendering. In *IEEE Visualization*, pages 319–326, 1996.

[ESV99] J. El-Sana and A. Varshney. Generalized view-dependent simplification. *Computer Graphics Forum*, 18(3):83–94, 1999.

[GH97] M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 209–216. ACM Press/Addison-Wesley Publishing Co., 1997.

[HG97] P. Heckbert and M. Garland. Survey of polygonal surface simplification algorithms. Technical report, Multiresolution Surface Modeling Course Notes of SIGGRAPH'97, 1997.

[Hop96] H. Hoppe. Progressive meshes. *Computer Graphics*, 30(Annual Conference Series):99–108, 1996.

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

[Hop99] H. Hoppe. Optimization of mesh locality for transparent vertex caching. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 269–276. ACM Press/Addison-Wesley Publishing Co., 1999.

[Jak00] A. Jakulin. Interactive vegetation rendering with slicing and blending. In A. de Sousa and J.C. Torres, editors, *Proc. Eurographics 2000 (Short Presentations)*. Eurographics, 2000.

[KT96] A. D. Kalvin and R. H. Taylor. Superfaces: Polygonal mesh simplification with bounded error. *IEEE Computer Graphics Application*, 16(3):64–77, 1996.

[LCV04] J. Lluch, E. Camahort, and R. Vivo. An image based multiresolution model for interactive foliage rendering. *Journal of WSCG'04*, 12(3):507–514, 2004.

[LD97] B. Lintermann and O. Deussen. A modelling method and user interface for creating plants. In *Proceedings of the conference on Graphics interface '97*, pages 189–197. Canadian Information Processing Society, 1997.

[LD99] B. Lintermann and O. Deussen. Interactive modeling of plants. *IEEE Computer Graphics Application*, 19(1):56–65, 1999.

[LE97] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 199–208. ACM Press/Addison-Wesley Publishing Co., 1997.

[Lin00] P. Lindstrom. Out-of-core simplification of large polygonal models. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 259–262. ACM Press/Addison-Wesley Publishing Co., 2000.

[LT97] K. Low and T. Tan. Model simplification using vertex-clustering. In *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 75–ff. ACM Press, 1997.

[LT00] P. Lindstrom and G. Turk. Image-driven simplification. *ACM Transaction Graphics*, 19(3):204–241, 2000.

[Lue01] D. Luebke. A developer's survey of polygonal simplification algorithms. *IEEE Computer Graphics Application*, 21(3):24–35, 2001.

[MO95] N. Max and K. Ohsaki. Rendering trees from precomputed z-buffer views. In Pat Hanrahan and Werner Purgathofer, editors, *Rendering Techniques '95, Proceedings of the Eurographics Workshop*, pages 74–81. Springer-Verlang, 1995.

[NT03] F. Nooruddin and G. Turk. Simplification and repair of polygonal models using volumetric techniques. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):191–205, 2003.

[PL90] P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants.* Springer-Verlag New York, Inc., 1990.

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

[RB85] W. Reeves and R. Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 313–322. ACM Press, 1985.

[RB93] J. Rossignac and P. Borrel. Multi-resolution 3d approximations for rendering complex scenes. In B. Falcidieno and T. Kunii, editors, *Modeling in Computer Graphics: Methods and Applications*, pages 455–465. Springer-Verlag, 1993.

[RC04] J. F. Ramos and M. Chover. Lodstrips: Level of detail strips. In *International Conference on Computational Science*, pages 107–114, 2004.

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

[RLR$^+$00] J. Ribelles, A. López, I. Remolar, O. Belmonte, and M. Chover. Multiresolution modelling of polygonal surface meshes using triangle fans. In *DGCI '00: Proceedings of the 9th International Conference on Discrete Geometry for Computer Imagery*, pages 431–442, London, UK, 2000. Springer-Verlag.

[Sch97] W. J. Schroeder. A topology modifying progressive decimation algorithm. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 205–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[SD01] M. Stamminger and G. Drettakis. Interactive sampling and rendering for complex and procedural geometry. In S.Gortler and C.Myszkowski, editors, *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 151–162. Springer-Verlag, 2001.

[SDS96] E. J. Stollnitz, T. D. Derose, and D. H. Salesin. *Wavelets for computer graphics: theory and applications.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[SP03] M. Shafae and R. Pajarola. Dstrips: Dynamic triangle strips for real-time mesh simplification and rendering. In Wenping Wang Jon Rokne and Reinhard Klein, editors, *Proceedings Pacific Graphics 2003*, pages 271–280. IEEE, 2003.

[SSHS98] J. Shade, S.Gortler, L. He, and R. Szeliski. Layered depth images. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 231–242. ACM Press, 1998.

[Ste01] A. J. Stewart. Tunneling for triangle strips in continuous level-of-detail meshes. In *GRIN'01: No description on Graphics interface 2001*, pages 91–100, Toronto, Ont., Canada, Canada, 2001.

[VS04] P. P. Vazquez and M. Sbert. On the fly best view detection using graphics hardware. In *Visualization, Imaging, and Image Processing (VIIP 2004)*. ACTA Press, 2004.

[WP95] J. Weber and J. Penn. Creation and rendering of realistic trees. In Robert Cook, editor, *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 119–128. ACM Press, 1995.

[XHM99] X. Xiang, M. Held, and J. Mitchell. Fast and effective stripification of polygonal surface models. In *SI3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 71–78, New York, NY, USA, 1999. ACM Press.

[XV96] J. Xia and A. Varshney. Dynamic view-dependent simplification for polygonal models. In *VIS '96: Proceedings of the 7th conference on Visualization '96*, pages 327–334. IEEE Computer Society Press, 1996.