ADVANCED TOOLS FOR DEVELOPING HIGHLY REALISTIC COMPUTER GAMES

WORKING MODULES FOR GEOMETRY

Document identifier:	**GameTools-4-D4.3-03-1-1-Working Modules for Geometry**
Date:	**15/05/2006**
Work package:	**WP04: Geometry**
Partner(s):	**UJI, UPV**
Leading Partner:	**UJI**
Document status:	**DRAFT**
Deliverable identifier:	**D4.3**

Abstract: This technical report describes the initially working modules of the geometry work package.

Delivery Slip

	Name	Partner	Date	Signature
From		UJI		
Reviewed by				
Approved by				

Document Log

Issue	Date	Comment	Author

Document Change Record

Issue	Item	Reason for Change

Files

Software Products	User files / URL
Word	gametools-ist-2-004363-4-d4.3-03-1-1-working modules for geometry.doc

CONTENT

1. INTRODUCTION

OBJECTIVES OF THIS DOCUMENT

This document describes the initially working modules for the Geometry Work Package. Its aim is to describe the modules and explain how they work.

DOCUMENT AMENDMENT PROCEDURE

Any project partner may request amendments but each amendment must be analysed and approved by the GameTools Project Coordinator or Project Manager.

TERMINOLOGY

Glossary

GTP	GameTools Project
PC	Project Coordinator
PM	Project Manager
WP	Work Package
LOD	Level Of Detail

2. DESCRIPTION OF THE MODULES

2.1. STRIPIFICATION MODULE

2.1.1. Description

This module is a geometry tool that makes triangle strip models from a list of triangles. Drawing 3D models using triangle strips has always been more efficient than just using a triangle list. This module only affects the triangle indices, the vertices are not affected by stripification changes.

This module uses the *tri-stripper* algorithm to accomplish the task of making triangle strips. This algorithm has been chosen because it is fast and it takes advantage of vertex caches found in most 3D cards.

The module was designed to support easy integration of other stripification algorithms. A future extension of this module is to implement different stripification algorithms that could be chosen from the GUI.

2.1.2. Module Structure

The source code of the stripification module can be found in two files: a file header named GeoMeshStripifer.h and a source file named GeoMeshStripifer.cpp. There is a parent class named `MeshStripifer` that contains the basic structure developed for classes implementing particular stripification methods. The tri-stripper method is implemented in the `CustomStripifer` class. Here is a brief description of the stripification module (for more information read the GameTools Geometry Modules Reference Manual 0.1):

`class MeshStripifier`

- `MeshStripifier`(): Constructor, initializes basic structures and gets the mesh to stripify.
- `Stripify()`: Make triangle strips of the mesh model.
- `GetMesh ()`: Gets the stripified mesh model.
- `setSubMeshLeaves()`: Used to select the leaf sub-mesh of a mesh model representing a tree.

2.2.3. Using the Module

Using strips to describe the geometry of the models accelerates the graphics visualization of the models. Representing a mesh model using triangle strips is more compact than representing the model using triangle lists. The former requires less storage and bus bandwidth than the latter. A tri-stripper-based algorithm has been implemented that exploits the vertex cache of the 3D cards and provides more efficient triangle strips than a stripper algorithm that does not use the cache.

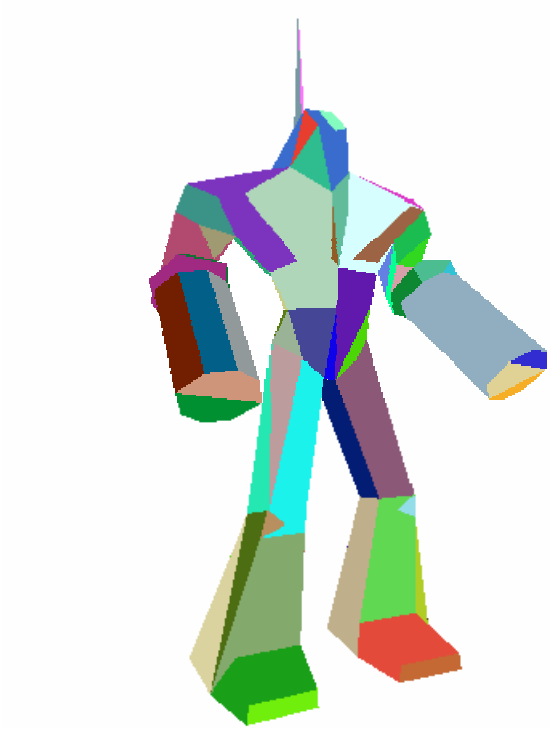


Figure 1: The mighty stripified robot

Figure 2: The Athena statue stripified

Figure 3: The dwarf model stripified using the GeoTool

2.2. SIMPLIFICATION MODULE

The multiresolution representation of a mesh model has two main parts: the original geometry of the object at its maximum level of detail, and the simplification sequence that supports the generation of different levels of detail. This Section deals with the simplification methods we use to construct our multiresolution models LODStrips and LODTrees.

First, we introduce the simplification methods we developed. Then, we present the structure of these methods. Finally, we describe how to use the module.

2.2.1. Description

We have developed three different simplification methods. Depending on the purpose of the simplification the user can apply any of the three methods. These methods are based on: geometric simplification of meshes, viewpoint-driven simplification and geometric simplification of foliage.

In the following sub-sections, we explain each of these methods.

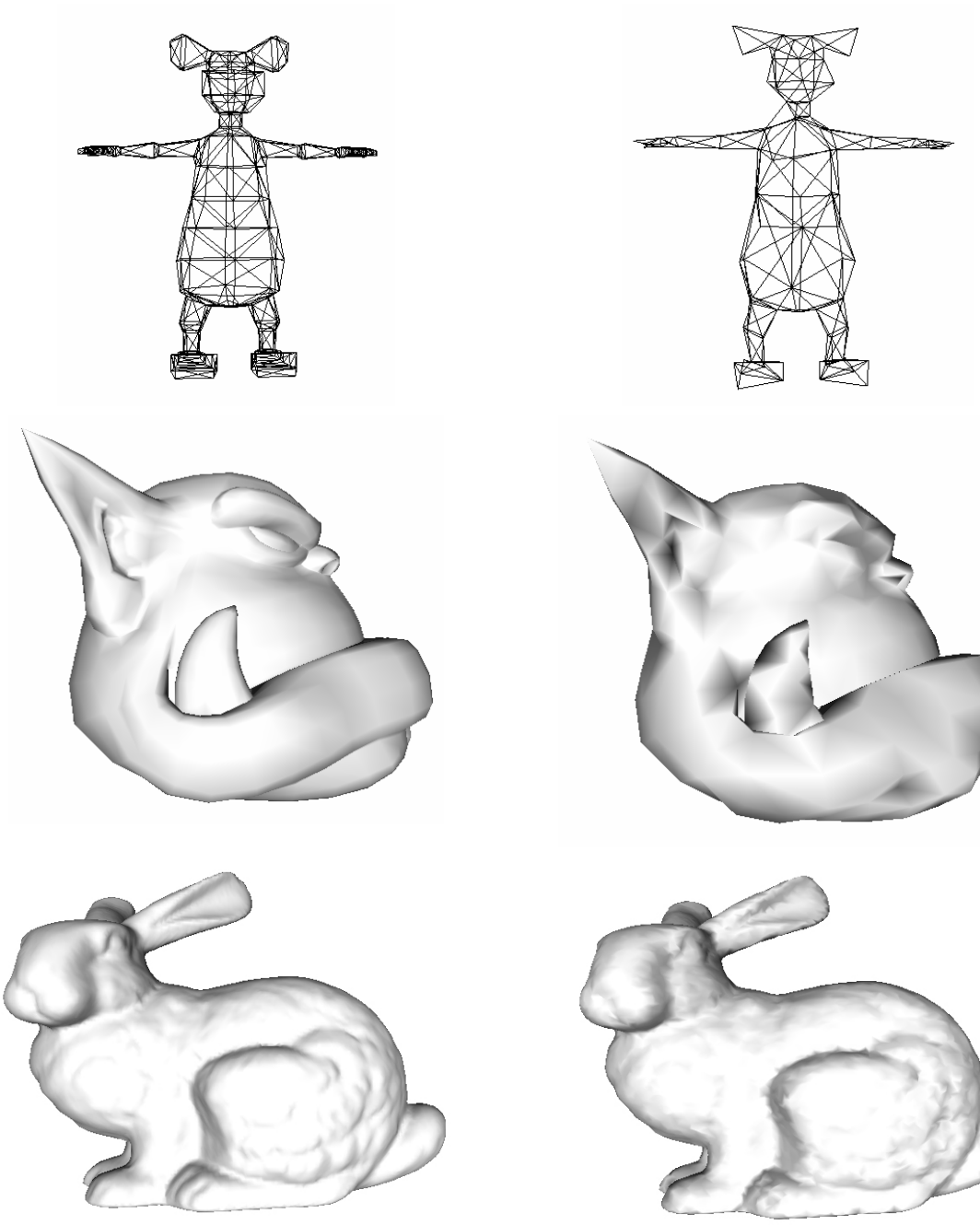
2.2.1.1. Geometric Simplification of Meshes

The method returns a simplified model given a high resolution model. Moreover, it also computes the simplification sequence. The simplification sequence represents the steps needed to transform the original mesh into its simplified form.

Our method is based on *edge contractions*, that is, it iteratively selects edges for removal. At each step, a new edge (or vertex pair) is selected and removed. One of the vertices is also removed and the indices of all affected faces are re-mapped to the other vertex. Faces that become degenerate during this process are also removed.

The method runs as follows. First, we merge all the sub-meshes of the model into a unique virtual mesh, storing all the information about the original connectivity of the meshes. The decision of merging two vertices that belong to different sub-meshes is based on the distance between those vertices. Moreover, if two neighbouring sub-meshes are composed of different materials, the edges between them are marked as virtual boundaries, and the simplification step keeps the original shape.

Next, the virtual mesh is simplified. We assign a decimation cost to each edge using a quadric error metric. This cost is used to sort the edges for simplification. The edge with the lowest decimation cost is the first edge simplified and removed. Edges with higher costs follow.



(a) High resolution models

(b) Low resolution models (geometry at 30%)

Figure 4: Example of geometric mesh simplification

Finally, the method retrieves the simplified mesh and splits it into different sub-meshes using the interconnectivity information stored before.

The method supports selecting edges to preserve mesh boundaries. This is done by assigning a higher cost to the edges of the boundaries.

Figure 4 shows an example of our geometric mesh simplification method applied to various models. 8(a) and 8(b) respectively show the high resolution models and the simplified models at 30% of the original geometry.

2.2.1.2. Mutual Information-Driven Simplification

We have developed a viewpoint-based simplification method to produce simplifications that minimize the changes of the simplified model for a given human observer.

The method returns a simplified model of the original model, including the simplification sequence.

A set of cameras is uniformly placed on a sphere around the object that contains the object's bounding box. Each camera renders the object and computes the area of the rendered triangles. With this area mutual information is performed. The mutual information is a metric that compares the error between two probabilistic distributions. This is a very wide concept that is applied here to the relation between the polygons and its projected areas. This method gives a good metric because it's very sensitive to visual changes from a point of view.

(a) High resolution models (b) Low resolution models (geometry at 5%)

Figure 5: Example of mutual information-driven simplification

For each edge a contraction is done and the mutual information is recomputed assigning a decimation cost. With this information, the edges are stored sorted by decimation cost.

Then, the contractions are computed. After each contraction, the contraction cost of the neighboring edges (based on a threshold) is recalculated.

Figure 5 shows an example of this kind of simplification. In 9(a) the high resolution models can be observed. 9(b) shows a simplified version with 5% of the original geometry. For this simplification we used 42 cameras located around the object.

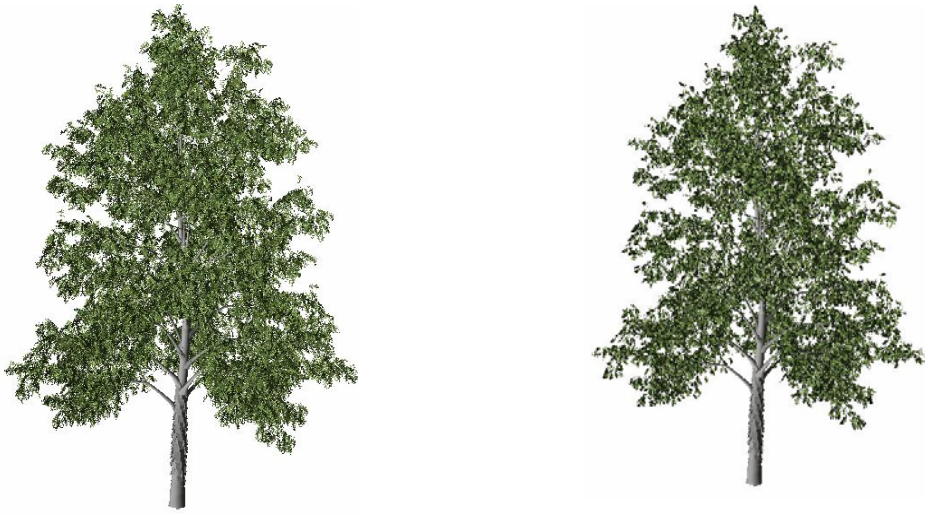
2.2.1.3. Geometric Simplification of Foliage

Given a tree model, we produce a simplified tree by generating an appropriate simplification sequence.

Simplification of a tree model is generated using two different simplification methods: one for the trunk and branches and the other for the leaves. This is because of the incorrect leaf simplification one gets when using geometric simplification of meshes for the leaves. So, the trunk and the branches are simplified with the simplification method for meshes, and the leaves are simplified with another method that works as follows.

The method assigns collapse costs to the leaves. This cost is based on a combination of distances and planarity.

Leaf collapses are computed without generating new vertices. The original vertices of the leaves are used instead.



(a) High resolution model

(b) Low resolution model (geometry at 50%)

Figure 6: Example of foliage simplification

2.2.2. Module Structure

The simplification methods are managed by classes. The base classes are:

`class MeshSimplifier`

This module is used by both types of models, general mesh models and plant and tree models. For plant and tree models we use this module for the trunk and the branches. It contains functions that generate simplified versions of 3D objects made out of triangles. Given a 3D object, this module computes a sequence of geometric transformations that reduce the object's geometric detail while preserving its appearance. For each simplification step, it returns a simplification sequence containing the edge to be collapse, the two triangles being removed and the new triangles remapped to the model.

Its input is a pointer to the `Geometry::Mesh` object containing the 3D model to be simplified. Its outputs are the simplified model, contained in a `Geometry::Mesh` object, and the simplification sequence, represented by a `Geometry::MeshSimplificationSequence` object.

The `MeshSimplifier` class is implemented as an abstract class that represents the interface of a simplification method. Its functions are:

- `MeshSimplifier()`: constructor, takes a pointer to a valid `Geometry::Mesh` object to simplify.
- `Simplify (Geometry::Real)`: Starts the simplification process. Receives as parameter the LOD factor in the range of [0,1]. This is a pure virtual method and must be overloaded in derived classes implementing simplification algorithms.
- `Simplify (Geometry::uint32)`: Starts the simplification process. Receives as parameter the number of vertices of the resulting mesh. This is a pure virtual method and must be overloaded in derived classes implementing simplification algorithms.
- `GetMesh()`: Returns the simplified mesh.
- `GetSimplificationSequence()`: Returns the simplification sequence for general meshes.
- `setMeshLeaves()`: Selects the mesh that stores the leaves.

`class ViewPointDrivenSimplifier`

This class implements a simplification algorithm based on a viewpoint evaluation technique. This class inherit from `MeshSimplifier` class. Its functions are:

- `ViewPointDrivenSimplifier()`: Class constructor.

- `Simplify (Geometry::Real)`: Starts the simplification process. Receives as a parameter the LOD factor in the range of [0,1]. Implements the `Simplifier::Simplify` method to perform an image-based simplification.

- `Simplify (Geometry::uint32)`: Receives as parameter the number of vertices of the resulting mesh. Implements the `Simplifier::Simplify` method to perform an image-based simplification.

`class GeometryBasedSimplifier`

This class implements a simplification algorithm based on a classic geometry evaluation technique. This class inherits from `MeshSimplifier`. Its functions are:

- `GeometryBasedSimplifier`: Class constructor.

- `Simplify (Geometry::Real)`: Starts the simplification process. Receives as parameter the LOD factor in the range of [0,1]. Implements the `Simplifier::Simplify` method to perform geometry-based simplification.

- `Simplify (Geometry::uint32)`: Starts the simplification process. Receives as a parameter the number of vertices of the resulting mesh. Implements the `Simplifier::Simplify` method to perform geometry-based simplification.

`class TreeSimplifier`

This module is used by LODTree to simplify the leaves of a tree. It contains functions that generate simplified versions of 3D objects made out of quads (represented as pairs of texture-mapped triangles). Given a 3D object, this module computes a sequence of geometric transformations that reduce the object's geometric detail while preserving its appearance.

For each simplification step, the module returns a simplification sequence containing the leaf collapsed, the two leaves being removed, and the leaf resulting from that contraction.

Its input is a pointer to the `Geometry::Mesh` object containing the tree to be simplified. Its outputs are the simplified mesh, contained in a `Geometry::Mesh` object, and the simplification sequence, represented by a `Geometry::TreeSimplificationSequence` object.

Its functions are:

- `TreeSimplifier()`: Class constructor. Retrieves a pointer to a valid mesh object to simplify.

- `Simplify()`: Starts the simplification process. Receives as parameter the LOD factor in the range of [0,1].

- `GetMesh()`: Returns the simplified mesh.
- `GetSimplificationSequence()`: Returns the simplification sequence for the leaves.

`class MeshSimplificationSequence`

This class stores the simplification sequence applied to a given mesh. It maintains information about the simplification process. For each simplification step it gives the vertex that collapses, the vertex that remains unchanged, the two triangles that disappear and the new triangles that come out.

It also offers a method to generate a file in mesh simplification sequence format. This file begins with a line stating the name of the mesh file it refers to. The other lines contain the following information:

- Vertex that collapses.
- Vertex that remains unchanged.
- The triangles that disappear.
- & as a separator.
- The list of new triangles.

The functions of the `MeshSimplificationSequence` class are:

- `MeshSimplificationSequence`: Class constructor.
- `Load()`: Loads a simplification sequence from a Serializer.
- `Save()`: Saves the contents of the data structures.
- `putMeshName()`: Inserts the mesh name into the `MeshSimplificationSequence`.
- `getMeshName()`: Gets the mesh name.

The members of the `MeshSimplificationSequence` class are:

- `Step`: Represents a simplification step in the sequence.
- `mSteps`: Stores all the simplification steps.
- `meshName`: Mesh name.

`class TreeSimplificationSequence`

This class represents the simplification sequence applied to a given mesh representing the leaves of a tree. It maintains information about the simplification process. For each simplification step it gives the four triangles that compose the two leaves that will be collapsed and the two triangles of the new leaf.

It also offers a method to generate a file in tree simplification sequence format. This file begins with a line stating the name of the mesh file it refers to.

The file containing the tree simplification data (*.lodt*) stores for each leaf collapse operation one line with the following format:

- The four triangles that represent the two leaves that will be collapsed.
- & as a separator.
- The two triangles of the new collapsed leaf.

The functions of the `TreeSimplificationSequence` class are:

- `TreeSimplificationSequence`: Class constructor.
- `Load()`: Loads a simplification sequence from a Serializer.
- `Save()`: Saves the contents of the data structures.

- `putMeshName()`: Inserts the mesh name into the `TreeSimplificationSequence`.
- `getMeshName()`: Gets the mesh name.

The members of the `TreeSimplificationSequence` class are:

- `Step`: Represents a simplification step in the sequence.
- `mSteps`: Stores all the simplification steps.
- `meshName`: Mesh name.

2.2.3. Using the Module

With this module, simplified models can be generated from a high resolution one. The goal is to obtain models that are quite similar to the original ones but use less storage and processing power.

The application offers three possible simplification options: mesh geometric simplification, viewpoint-based simplification and foliage simplification.

If we want a fast and geometry-based simplification method for meshes we can use mesh geometric simplification. If we want simplification that minimizes the differences between the simplified model and the original one for a human observer, we can use viewpoint-based simplification. And finally, if we want to simply a tree's leaves, we can use foliage simplification.

Moreover, for each simplification our simplification methods generate a simplification sequence. So, in each step of the simplification some information about the operation is stored. This way we can easily change from one level of detail of the model to the next or another one. This is especially useful for creating multiresolution models.

2.3. LODSTRIPS CONSTRUCTION MODULE

2.3.1. Description

LODStrips is a multi-strip-based continuous multiresolution model. The LODStrips construction process takes as input a manifold mesh and extracts the simplification information needed to transform the original mesh into the simplified one. This simplification sequence is provided by the simplification module. Thus, the original mesh must be simplified first to a certain level of detail given by the user. Once the model has been simplified, and the simplification steps extracted, the next step is the stripification of the mesh. This process requires no user input and works with hard-coded fixed options. The stripification process is needed because the LODStrips model works with stripified meshes to minimize the number of vertices to be sent to the GPU. When the model has been stripified the process of LODStrips construction starts, involving vertex sorting and other low-level processes that will not be explained here because they are beyond the scope of this document.

The resulting multiresolution model is store in two files:

- One file represents the stripified geometry of the original model stored in a standard Ogre *.mesh* file. As each sub-mesh in the stripified model contains a high number of triangle strips, these strips are joined using degenerated triangles. Thus we transform a model composed of *n* sub-meshes and *m* triangle strips per sub-mesh into a model formed by *n* sub-meshes and only *1* triangle strip for each sub-mesh. This makes the model faster to be drawn, because it's important for the renderer to minimize the number of render calls. Moreover, this is the only way to provide a standard Ogre *.mesh* file with many strips per sub-mesh.
- The other file stores the multiresolution information needed to perform fast level of detail switching in real time. This information is saved to a *.lods* file, a text file containing:
    - The simplification order for each vertex, coded in the file with the format: "*v #vertexid*".
    - The lines beginning with a 'd' character contain: the identifier of the triangle strip to be modified, the number of collapses in this strip, the number of vertex and edge repetitions, and a flag indicating if the current collapse forces the next one. This flag is used to avoid holes in the mesh.
    - Those lines beginning with a 'p' character contain the number of strips affected by every LOD change.
    - Lines beginning with a 'b' character contain all information needed to compute the collapses and repetitions.

2.3.2. Module Structure

The LODStrips constructor is implemented in the `LodStripsConstructor` class. This class encapsulates the processes needed to construct a LODStrips multiresolution model from a mesh simplification sequence and the original mesh. This class inherits from the class `Serializable` which provides the functionality needed to be saved to disk.

Constructor

`LodStripsConstructor` takes two class construction parameters: the original stripified mesh and a `MeshSimplificationSequence` object representing the decimation sequence provided by the geometry simplifier. Moreover, there are two optional parameters: one is used to discard any sub-mesh from the construction process. This is useful to reuse the `LodStripsConstruction` module in the LODTree construction process that will be described later. The other optional parameter is used to provide feedback about the progress made so far by the construction process.

Methods

The main method of this class is `Save()`. It is used to construct and save the multiresolution model in the *.mesh* and *.lods* files.

- `GetMesh()`: returns the current mesh of the constructed model, once stripified.
- `SetSubMeshLeaves()`: is used to give the identifier of the sub-mesh to be ignored, as in the optional parameter of the constructor.

2.3.3. Using the module

Any client application (like the GeoTool) wanting to construct a LODStrips model with this class has to:

1. Load a mesh into a `Geometry::Mesh` object.
2. Generate a `Geometry::MeshSimplificationSequence` given by any of the simplification methods provided by the GTGeometry library.
3. Create a `Geometry::LodStripsConstructor` object from the two previously created objects: the mesh and the simplification sequence.
4. Call the `Save()` method to construct the model and output the information into a `Serializer` object (like a file). This method does the following:
    - It writes the *.lods* file with the simplification sequence.
    - It modifies a copy of the original mesh to reflect the changes occurred in the construction process.

2.4. LODSTRIPS MODULE

2.4.1. Module Description

A common way to handle the problem of rendering large 3D scenes is to use multiresolution modeling. Multiresolution models store different levels of detail of an object in order to optimize the rendering cost for each view of the scene.

The LODStrips model represents a mesh as a set of multiresolution strips. For each level of detail a set of indices is selected for rendering without changing the original geometry. LODstrips supports continuous multiresolution because it stores each simplification change generated by each simplification step.

So, this module supports generating a multiresolution model with its simplification information and its corresponding simplification sequence. With this sequence it is possible to smoothly change between levels of detail, just by performing the changes stored in the simplification sequence.

Figure 7. Example of many LODStrips models at different levels of detail

2.4.2. Module Structure

The base class of the module is the class `LodStripsLibrary` which represents a multiresolution model managed by a LODStrips algorithm.

`class LodStripsLibrary`

This module contains functions that handle the levels of detail of the input multiresolution objects made of polygonal meshes. For any given resolution and object, this module returns a set of triangle strips representing the object at that resolution, that is, at the level of detail requested. These models use triangle strips to reduce storage usage and to speed up realistic rendering.

Its input is a file describing a multiresolution object. And its output is a strip set that represents the level of detail requested.

Its functions are:

- `LodStripsLibrary`: Constructor, receives as a parameter the name of the file containing the multiresolution object.
- `MaxLod()`: Returns the highest LOD.
- `MinLod()`: Returns the lowest LOD.
- `GoToLod()`: Returns de current LOD and changes to the specified LOD.
- `TrimByLod()`: Establishes the new LOD range. Only the LODs in that range are stored and used.
- `MaxFaces()`: Returns the number of triangles of the highest LOD.
- `MinFaces()`: Returns the number of triangles of the lowest LOD.
- `MaxVertices()`: Returns the number of vertices of the highest LOD.
- `MinVertices()`: Returns the number of vertices of the lowest LOD.
- `GetStripCount()`: Gets the number of strips.
- `GetIndexCountByStrip()`: Gets the number of indices by strip.

Its structures are:

- `mStrips`: Vector of strips.

Figure 8 shows an example of a model that fully exploits the LODStrips algorithm.

Figure 8. Example of a LODStrips model at different levels of detail

2.4.3. Using the Module

The usage of the `LodStripsLibrary` by a client application is quite straightforward. The client needs to create an instance object of the `LodStripsLibrary` class, which is the class that manages all the data and the multiresolution processes. Then, simply by calling the `GoToLod()` function, the library updates the current state of the model and adapts it to the requested level of detail. Finally, to render the state of the actual multiresolution model, the application can query these data from the `LodStripsLibrary` class using the `mStrips` vector structure.

We are working on a new design of the data management used in the LODStrips library. This design will improve performance and data storage. It will allow the client application to store the data in the most efficient way. Here, the term *efficient* is a client-dependent quality because different applications may need to store the data in completely different ways. For example, an OpenGL application stores the information in a different way compared to a Direct3D application. Our new approach will allow the application to be the container of the data, while the library just provides an *algorithm* that knows how to use the data. This will also avoid the need to query data from the library, thus increasing the overall performance.

2.4.3.1. LODStrips manager

Managing a single LODStrips model in a scene is quite straightforward. However, dealing with a scene with a high number of LODStrips models may be a bit more complicated. Since extracting a given level of detail is not a constant-time operation, it takes same time to change the level of detail of an object. When there is only one LOD model in the scene this really doesn't matter, but when a scene is populated of multiresolution models of this type it would be a problem if a large number of LOD models need to change its level of detail. This could cause frame rate droppings during the game play. Thus, any sort of LODManager is needed to ensure that the LOD extraction only occurs in a separated time space. The general process would be:

1- A given LODStrips model asks the LODStrips Manager to change its LOD.

2- The manager assigns a pripority to that request and stores it

3- When an X time has passed since the last LOD petition was served, the manager servers the next priority-based petition.

That would ensure a constant frame rate with no droppings of the game play experience.

2.4.4. Working with Ogre

The integration of the *LODStrips* library within a game engine entails mainly one task: provide support to access the data structure where the triangle strips are stored. This way, the rest of the multiresolution tasks would be carried out by the main library, and we would only need an intermediate library to access the triangle strips of the model in the most appropriate way. In the `LODStripsLibrary` module, this data structure is called `mStrips` and contains the different triangle strips that make up the polygonal mesh.

Now we need a method to initially fill this data structure with the original triangle strips. Then, the library can apply the changes to the triangle strips. Once the data structure is filled, we can just use the

`GoToLod()` function and recover the new triangle strips from `mStrips`. Using those strips we can then fill the game engine's data structures.

The only problem we found when using the `LodStripsLibrary` in the Ogre game engine comes from a limitation in the engine. This limitation forces us to use only one triangle strips per sub-mesh. This way, we need to unify all the different triangle strips into a single one. We do that adding degenerate triangles every time we update the level of detail. Due to this requirement we experimented a slight decrease in the model performance. Still running time remained good on average.

In the demo we have developed for the *GameTools* project, we have included an initial version of this intermediate class, called `OgreLodStripsLibrary`. In this library we have maintained almost all of the original functions. We had to modify the creation of the data structure to read the strips from the original *Ogre* mesh. We also had to change the level-of-detail update routine, where we added code necessary to change the `IndexBuffers` where the strips are stored.

Figure 9: A LODStrips-based model at a 35% level of detail

The demo application allows loading any desired model and adjusting the level of detail according to the distance to the viewer criterion. We have also added a color-coded material to reflect the level of detail. It is possible to modify the name of the model and the distances taken into account for selecting the detail. Apart from the option of moving around the scene, the demo also offers:

- keys Z and X for scaling the model,
- keys W and S for modifying the position of the model, and
- keys A and D for rotating the model.

Figure 10: A textured LODStrips model in Ogre at a 50% level of detail

2.5. LODTREES CONSTRUCTION MODULE

2.5.1. Description

LODTrees combines a continuous multiresolution model designed exclusively to represent leaves and a continuous multiresolution model conceived to manage manifold smooth meshes: LODStrips, which is used for the trunk and branches of the tree.

As in the LODStrips construction module, the constructor starts from a mesh representing a tree. There is one restriction: the tree can be made of any number of sub-meshes, but the leaves must fit into just one sub-mesh. This sub-mesh must also be exclusively devoted to storing the leaves. This is so because the leaves and the trunk are rendered using different types of rendering primitives and this makes them exclusive.

The overall process is similar to the one described in the LODStrips construction section with some additions. First, the user must select manually the sub-mesh containing the leaf geometry. After that, the construction process begins by simplifying the initial tree to a certain level of detail given by the user. Finally, the simplification sequence generated during the simplification process is saved. Since the tree has two differentiated parts represented by two different multiresolution models, the result is two different simplification sequences. The simplification methods for both types of geometry have already been explained in detail in Section 2.2.

A LODTree model is stored in three files: a *.mesh* object file containing the geometry for both the trunk and branches and the leaves, a *.lods* file containing the multiresolution model for the trunk and branches, and a *.lodt* file containing the simplification sequence for the leaves. The latter is a text file with the following format.

Each line in a *.lodt* file represents a leaf collapse containing the four triangles being collapsed (two triangles per leaf) and the indices for the resulting collapsed leaf.

2.5.2. Module Structure

The LODTree constructor is implemented in the `LodTreeConstructor` class. Like the LODStrips construction module, this class encapsulates the processes needed to construct and save a LODTree multiresolution model. This class inherits from the class `Serializable` which provides the functionality needed to be saved to disk..

`class LodTreeConstructor`

`LodTreeConstructor` takes two class construction parameters: the original stripified mesh and a `MeshSimplificationSequence` object representing the decimation sequence provided by the geometry simplifier. Moreover, there are two optional parameters: one is used to discard any sub-mesh from the construction process. This is useful to reuse the `LodStripsConstruction` module in the LODTree construction process that will be described later. The other optional parameter is used to provide feedback about the progress made so far by the construction process.

The methods of `LodTreeConstructor` are:

- `LodTreeConstructor():` class constructor. Takes two parameters: the original stripified mesh and a `MeshSimplificationSequence` object.
- `Save():` does all the computations and outputs the results to the corresponding *.mesh*, *.lods* and *.lodt* files as explained above. This method receives a `Serializer` object to redirect the output to the corresponding place.

2.5.3. Using the Module

This process is very similar to the LODStrips construction process. The only difference is the requirement of manually selecting the sub-mesh of the model that represents the leaves. After that, the process is the same for the client application:

1. Load a mesh into a `Geometry::Mesh` object.
2. Generate a `Geometry::MeshSimplificationSequence` given by any of the simplification methods provided by the GTGeometry library. This generates and saves the simplification sequence for the trunk and branches of the tree.
3. Generate a `Geometry::TreeSimplificationSequence` provided by a `Geometry::TreeSimplifier` object to save the simplification sequence for the leaves.
4. Create a `Geometry::LodTreeConstructor` object from the three previously created objects, the mesh and the two simplification sequences.
5. Call the `Save()` method to construct the model and output the information to a `Serializer` object (like a file). This method does the following:
    - Write the *.lods* file with the simplification sequence.
    - It modifies a copy of the original mesh to reflect the changes occurred in the construction process.

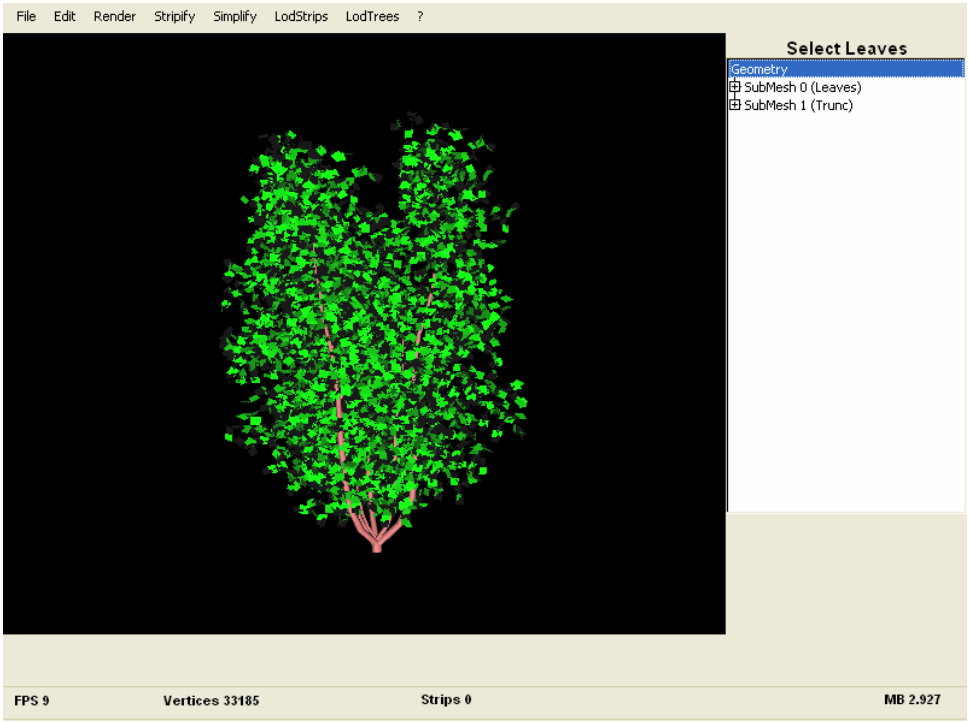
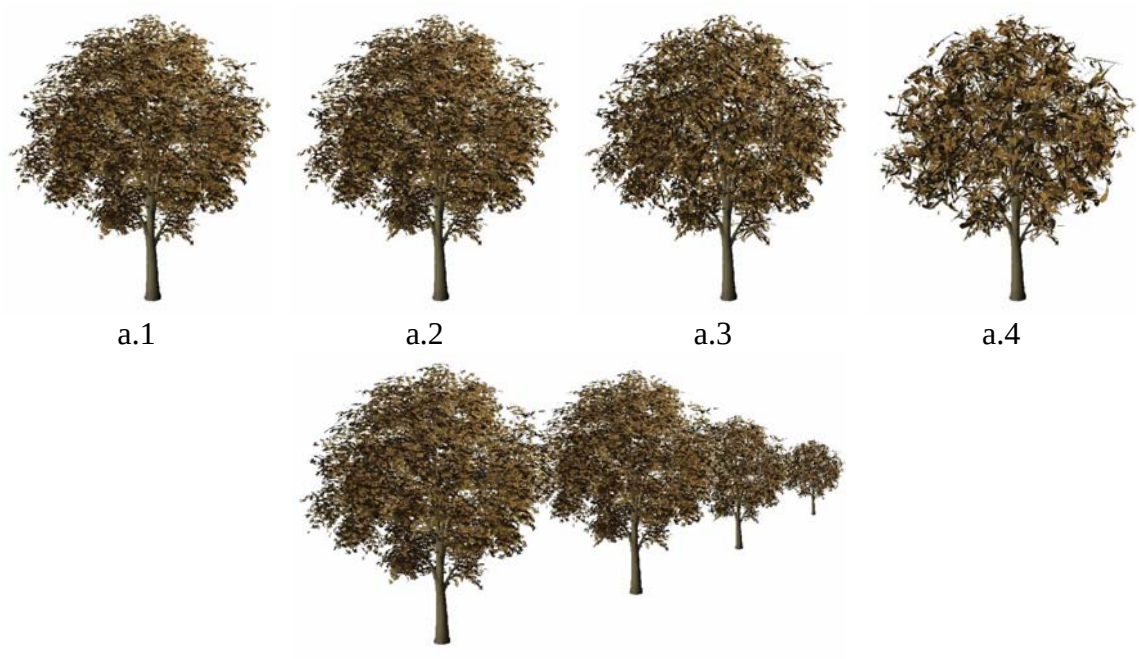


Figure 11: A tree about to be converted into a LODTree object. The tree is rendered without textures to see the real geometry

a.1 a.2 a.3 a.4

a.5

Figure 12: Results of simplification of the Sorbus aucuparia. Models with 24.839 leaves (a1), 18.629 (s2), 12.419 (a3) and 6.209 leaves (a4). a.5) shows a composition of the models according to the distance.



a.1 a.2 a.3 a.4

a.5

Figure 13: Results of simplification of the Aesculus hippocastanum. Figure a.1 shows the original model with 29.534 leaves. The simplified models are formed by a.2) 22.150 leaves, a.3) 14.767 leaves and a.4) 7.383 leaves. Figure a.5) shows a composition of the models according to the distance.

2.6. LODTREES MODULE

2.6.1. Description

A multiresolution model must provide an efficient framework for interactive applications to manage levels of detail that maintain acceptable frame rates. To guarantee that, models usually take advantage of the fact that polygonal models do not need the same polygonal complexity when they move away from the camera.

LODStrips models provide an efficient way of changing the level of detail of an object by rendering it with an efficient rendering primitive: triangle strips. However, the leaves of plants can not be treated in the same manner as smooth and continuous surfaces. This is due to the spurious nature of vegetation.

To solve this, the LODTree multiresolution model was developed. It can process leaf data using a special simplification method based in leaf collapses. This method can be configured to use interchangeable metrics such as leaf proximity and visual impact.

Given a target level of detail, the LODTrees library can extract a list of leaves to be drawn as well as connectivity information and the vertices needed for each leaf. Like LODStrips, only the list of indices is recomputed, not its vertex data.

While the leaves are handled using this leaf-collapse-based multiresolution method, the trunk and the branches are handled using the previously described LODStrips model.

To handle the required information, a multiresolution LODTree model must be represented using three files: a *.mesh* file, a *.lods* file and a *.lodt* file. They store the geometric data and the simplification sequences for the trunk and branches and the leaves, respectively.

Since using three different types of files to manage a single multiresolution tree can be tedious or annoying, we are working on integrating all the required information into a single file called *extended .mesh file*. This file is a file in standard Ogre mesh file format with additional special chunks of data that represent the multiresolution data. Such a file format would be transparent for standard Ogre-based loaders because the Ogre engine just ignores unknown chunk ids. In this case, software loading a file would only load a ‘static’ tree file. On the other hand, a ‘prepared/modified’ Ogre-based loader could load the entire simplification model from the extended file by reading the additional data chunks labeled with the new chunk ids.

We are also planning on integrating bones to support skeletal animations of the tree which could be useful to simulate wind effects or collisions between branches and leaves.

2.6.2. Module Structure

The base class that handles a LODTree model is `LodTreeLibrary`. This class builds a multiresolution LODTree object from the parameters passed to the constructor.

Constructor

The class constructor takes as parameters the files needed to load the simplification sequences for the trunk and branches and the leaves. It also takes a `Geometry::Mesh` object to handle the geometry and an identifier to select the sub-mesh that represents the leaves.

Methods for Managing LODs

The following methods are used to specify the levels of detail of the two parts of the model: the trunk and branches and the leaves.

- `GoToTrunkLod():` Sets the trunk and branch target vertex count to *newlod*.
- `GoToFoliageLod():` Sets the foliage target vertex count to *newlod*.

Methods to Retrieve LOD Data

These methods are useful to get information about the current LOD state of an object.

- `MinTrunkLod():` Returns the lowest possible LOD for the trunk and branches.
- `MaxTrunkLod():` Returns the highest possible LOD for the trunk and branches.
- `MinFoliageLod():` Returns the lowest possible LOD for the foliage.
- `MaxFoliageLod():` Returns the highest possible LOD for the foliage.

Methods to Retrieve Geometry Data

These methods are used to retrieve vertex data from the multiresolution model.

- `CurrentLOD_Trunk_StripCount():`Returns the active strip count for the current LOD of the trunk and branches.
- `CurrentLOD_Trunk_IndexCountByStrip():` Returns the number of active indices for a given triangle strip of the trunk and branches.
- `Get_Foliage_MaxIndexCount():` Returns the maximum number of active indices of the foliage.
- `CurrentLOD_Foliage_IndexCount():` Returns the current number of active indices of the foliage.
- `CurrentLOD_Foliage_Indices():` Returns the active indices of the foliage.

2.6.3. Using the Module

The usage of the `LodTreeLibrary` by a client application is very similar to the usage of the `LodStripsLibrary`. The client needs to create an instance object of the `LodTreeLibrary` class, which is the class that manages all the data and multiresolution processes, including the foliage and the trunk and branches. Then, simply by calling the `GoToTrunkLod()` and `GoToFoliageLod()` methods, the library updates the current state of each part of the model (trunk and branches, and foliage) and adapts it to the requested level of detail.
Finally, to render the state of the actual multiresolution model, the application can query these data from the `LodStripsLibrary` class using the appropriate methods explained above.

As described in the LODStrips module section, this may not be the most efficient way to do it. So we use the same interface described for the LODStrips module. The interface has the following features:

- It avoids duplicity of the data: the data is kept by the client application; however the `LodStripsLibrary` has the ability to change that data.
- It removes *dumping* times: since there are not duplicates of the data, it does not need to be copied anywhere.

2.6.3.1. LODTree manager

As argued in the LODStrips section, a LODManager is needed for an entire forest populated with LODTrees to perform well, with no droppings of the game play. The same approach could be also used with the LODTrees. However, the spurious nature of the trees (the leaves) could inspire some extensions to the latter approach. A valid extension would assign a priority not only based on the distance of the model to the camera but also taking into account the percentage of the model not occluded by other models. Moreover, this would help to assign not only a priority of a LOD petition but also to perturb the LOD factor so that the visibility of the model is taken into account.

2.6.4. An Example in Ogre

The following images show examples of integration of the LODTree library with a client application, in this case the Ogre engine. It consists of a LOD forest populated with some trees that change their LODs independently.

Figure 14: An LOD forest. The further is the tree, the lower the level of detail

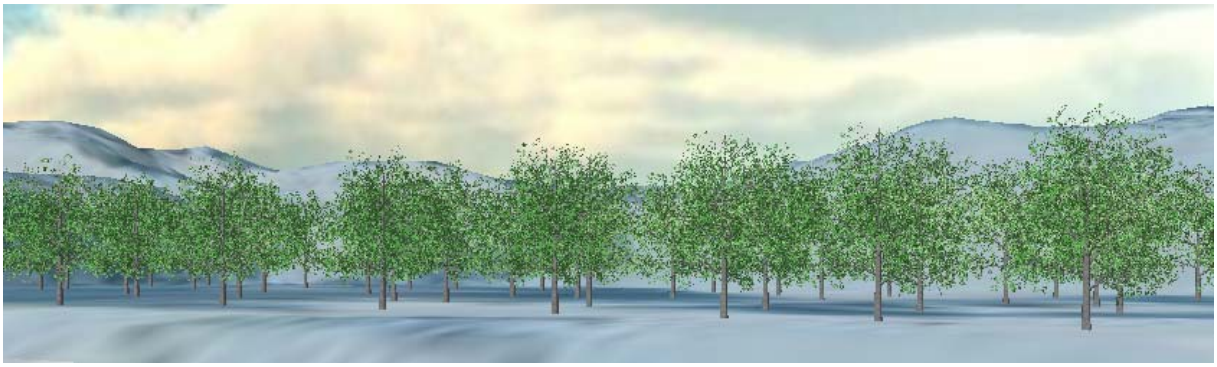


Figure 15: Our LOD forest seen from far away

Figure 16: Some examples of another type of trees using the LODTree library

Figure 17: A complete LOD scene: LODStrips models (ninjas) in front of a LODTree-based forest

2.7. GEOTOOL APPLICATION

This is a standalone application used both as a usage demonstrator of the GameTools Geometry Library and a useful tool to quickly generate multiresolution models.

2.7.1. Description

GeoTool is a multiplatform, portable and engine independent tool useful to manipulate meshes and build multiresolution models. It also allows more basic operations such as mesh simplification and stripification. The application uses the FLTK toolkit to provide a portable graphical user interface, and the OpenGL real-time rendering API to perform every rendering call.

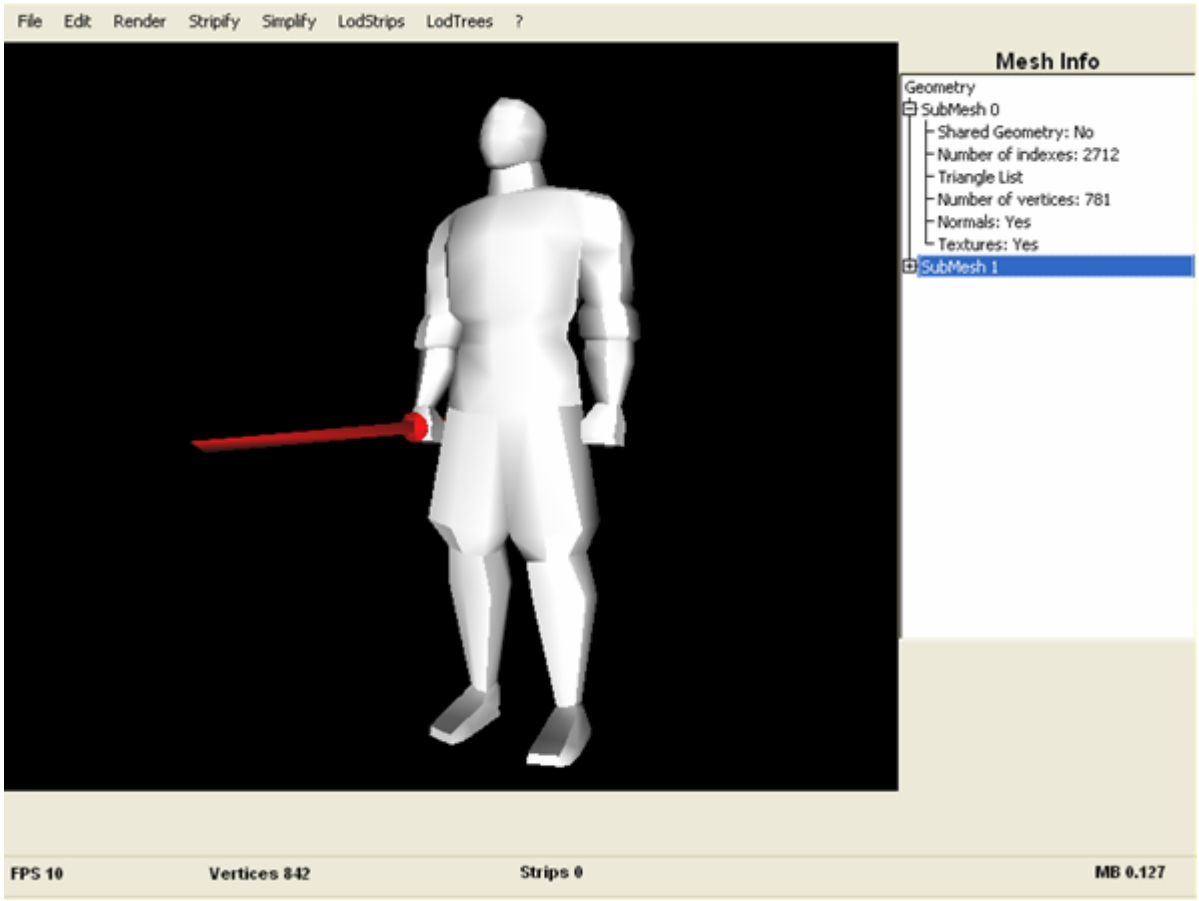


Figure 18: A screenshot of the GeoTool application

The application uses the Ogre mesh file format to load and store geometry data. This file format supports mesh models composed by any number of sub-meshes. Each sub-mesh can be represented by any rendering primitive (a triangle list or a triangle strip). This is useful to store trees with the LODTree model, because the trunk must be represented by triangle strips and the leaves by triangle lists. Moreover, the Ogre file format supports bones and skeletal animations. In future versions the application will also support them as well as texture coordinates.

The application also needs a place to store the simplification sequence needed to build and visualize the implemented multiresolution models. These data is stored in separate files, associated with an Ogre mesh file, together with the multiresolution data. LODStrips use a single sequence, and LODTree uses two sequences: one for the trunk and the other for the leaves. This will be explained in more detail in the LODTree construction section.

GeoTool allows the user to perform three different types of operations:

- Visual operations: these operations do not affect the mesh itself but the way it is rendered. The rendering primitive can be changed (wire mode, solid mode, triangle strips) as well as the lighting surface parameters (flat and smooth). The rendering viewpoint can also be changed in order to focus on the desired region of the model. Moreover, the application can load a previously computed LODStrips or LODTree model and render it.
- Basic operations: basic operations involve mesh stripification and simplification. These are catalogued as basic operations because they are done in a single step and return a transformed standalone mesh. Two different simplification approaches are available (as introduced in the following section): geometry-driven simplification and viewpoint-driven simplification. They will be explained in more detail in section 2.3.
- Complex operations: these operations are LODStrip construction and LODTree construction. Internally, these complex operations perform some basic operations such as stripification, simplification and vertex reordering. They are much more time consuming than basic operations. They take a mesh as input, construct a new mesh and an associated multiresolution sequence, and save the result to disk. We explain this process in more detail in the following sections.

Because the use of multiple files representing a single multiresolution model can be confusing, a new approach has been developed: the Ogre file format is a binary format composed of chunks of data. Each chunk contains info depending of its chunk type, so a new chunk type containing the multiresolution data (simplification sequences) was included in the Ogre mesh file format. This is a very transparent approach, because if a standard Ogre-based application tries to load an extended mesh file, then it will ignore the extended chunk. However, an Ogre-based application which makes use of the GTGeometry modules could successfully load the improved chunk in order to load the multiresolution data.

Summarizing, GeoTool integrates all the Geometry WorkPackages providing:

- ‘HowTo’ sample code for using all the Geometry WP modules.
- A handy application that integrates mesh simplification and stripification, and multiresolution model construction and rendering in a single compact tool.

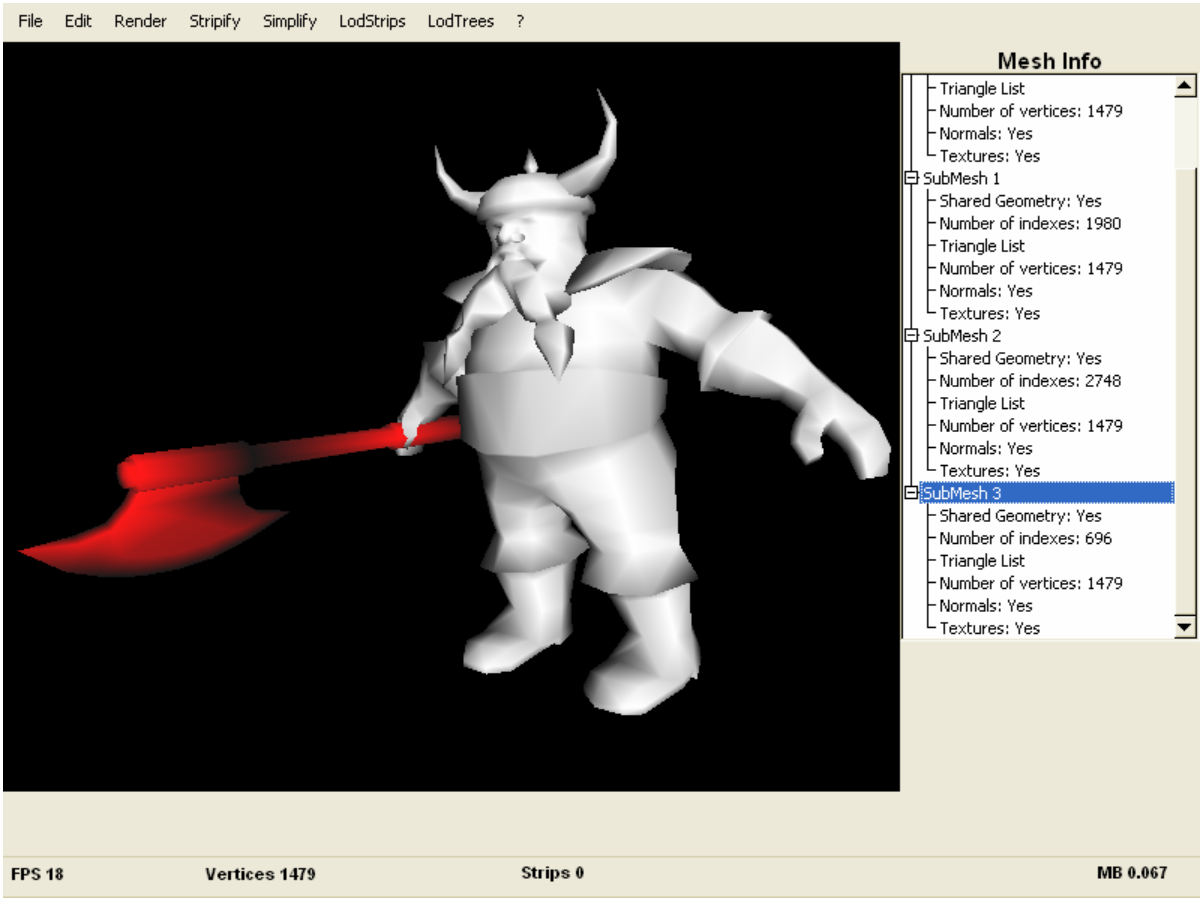


Figure 19: Getting mesh information from the GeoTool

2.7.2. User Interface

The user interface of GeoTool has been designed to be easy to use. The menu bar across the top of Figure 19 manages all the operations that can be performed on a mesh.

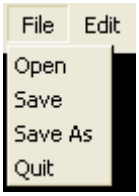
The main window in the center shows the current render state, which can be changed using the *Render* menu. The panel on the right shows a more detailed view of the current selected action. For example, when the simplify option becomes selected, the panel on the right shows more information and options about the selected action.

The status bar on the bottom of the application shows some information about the loaded model, such as its vertex, strip and triangle counts.

2.7.2.1. GeoTool Menu System

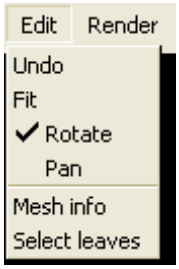
This section will explain all menu options and panels, and how they are organized.

The File Menu

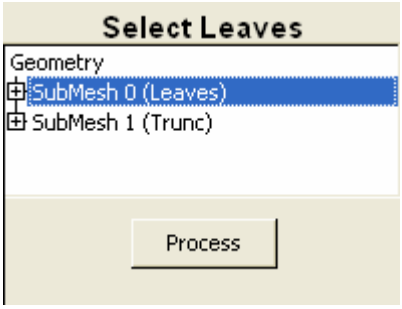


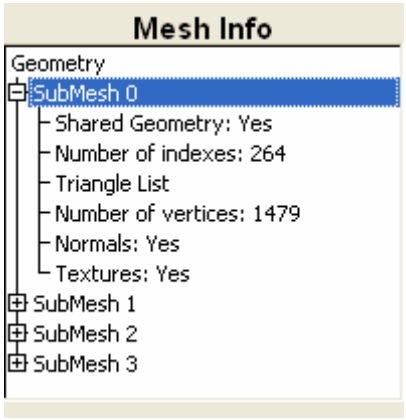
- Open: Shows a dialog to open an Ogre mesh file and load it into the application.
- Save (as): Saves the current mesh into an Ogre mesh file.
- Quit: Terminates the application.

The Edit Menu



- Undo: Gets the current mesh back to its previous state.
- Fit: Modifies the current view to fit the loaded mesh inside the screen.
- Rotate/Pan: Selects the action to be taken when the user drags the mouse pointer.
- Mesh info: Configures the right panel to show mesh information, such as its vertex and triangle counts, the rendering primitive type and its sub-mesh count.
- Select leaves: Configures the right panel to show a sub-mesh selector. This allows the user to select the sub-mesh that represents the leaves of a tree.

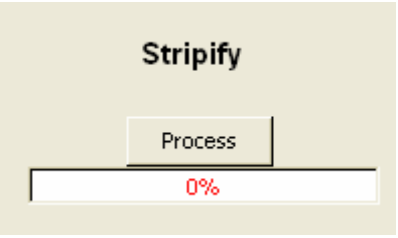




The Render Menu

- Wire / Solid: Selects the geometry rendering mode: wireframe or solid.
- Flat / Smooth: Selects the surface shading mode: flat or smooth (Gouraud).

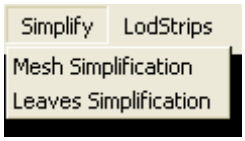
The Stripification Menu



This menu has no popup menu associated. Instead, it immediately opens the Stripification panel and begins the stripification process (see figure on the left).

The progress bar will show the stripification status.

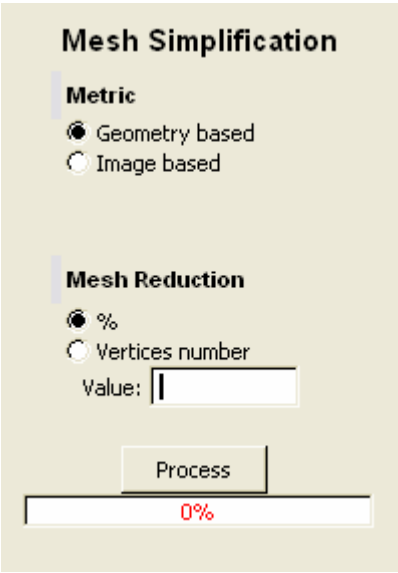
The Simplify Menu:



- Mesh simplification: Performs general geometric simplification using one of these two methods:
  - Geometry-based simplification
  - Viewpoint-based simplification

  The selection of the simplification method is done using the right panel shown in the figure shown below under Simplification Panel.
- Leaves simplification: Shows the leaf simplification panel that allows leaf simplification.

The Simplification Panel:



This is the panel shown when the Mesh simplification menu is selected. This panel allows the user to select the simplification metric to be Geometry- or Image-based (Viewpoint-based simplification).

While geometry-based simplification is much faster than the image-based simplification, the latter tends to produce better simplified meshes due to the nature of the simplification algorithm.

The panel also allows the user to select the simplification limits, which can be specified as either a percentage or a number giving the target vertex count.

During the simplification process, the progress bar shows the simplification status at any given time.

This panel also works for the leaf simplification method. However, it internally performs leaf-collapse simplification instead of the edge-collapse simplification typically used for meshes.

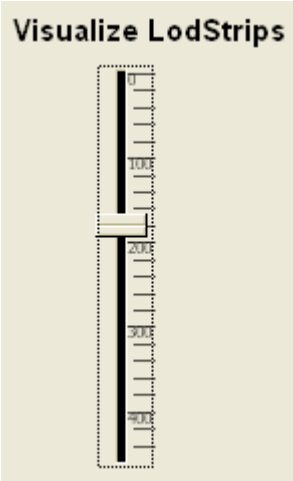
The LODStrips Menu:

- Generate: shows the LODStrips construction panel.
- Visualize: loads a LODStrips model from disk. To do so it allows the user to search for the *.mesh* and *.lods* files. Then it shows the LOD control panel.

El LODTree Menu:

- Generate: shows the LODTree construction panel.
- Visualize: loads a LODTree model from disk. To do so it allows the user to search for the *.mesh*, *.lods* and *.lodt* files. Then it shows the LOD control panel.

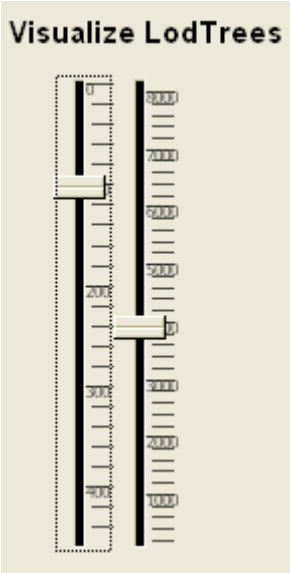
LOD Control Panels:



These two figures show the multiresolution visualization panels. They allow the user to change the level of detail used to render the current model.

The figure on the left shows the panel used for LODStrips models. The figure on the right shows the panel used for LODTree models.

The panel of the right has two slider controls because the foliage and the trunk meshes can have different levels of detail.



2.7.3. Using the Application

This section describes the general process needed to perform operations such as loading a mesh, simplifying a mesh, and complex operations like building and visualizing a LODTree model.

2.7.3.1. Basic Application Usage

The first thing we generally do is load a mesh from disk. This is done by selecting the *Open* option in the *File* menu. This action will bring up a file chooser dialog window where we can browse for the file we want to load. This dialog will search for files with .mesh extension, i.e. Ogre mesh files.

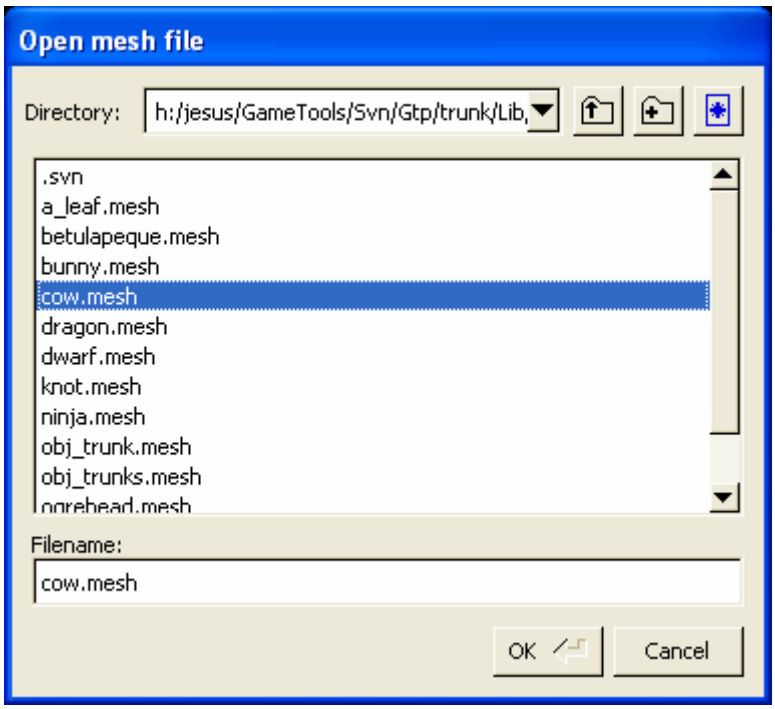


Figure 20: The 'open mesh' file browser

Once the mesh is loaded into the application we can apply some operations to it. We can change the point of view, retrieve mesh information or change mesh rendering options using the menu system explained above.

2.7.3.2. Elemental Operations

Elemental operations take a mesh as input and transform it to another output mesh so that this resulting mesh can be used as input to another elemental operation.

Two elemental operations are supported by GeoTool: mesh simplification and mesh stripification. Geotool provides a handy way to easily apply these operations to a mesh and use the mesh for other operations.

To simplify the mesh we select the *Simplify* option of the menu bar and select *Simplify mesh*. This will bring up the simplification panel as shown in the previous section. There, we can introduce the simplification factor. For example, the *cow* model simplified to 20% of its original size has the following appearance compared to the original one:

Figure 21: The cow model simplified with GeoTool

The stripification process is done in a similar way, so we do not explain it any further.

2.7.3.3. Advanced Operations: Multiresolution Model Construction

Advanced operations are more complex than basic ones because they involve several processes. However, GeoTool encapsulates them into a simple interface to make them easy to use. There are two advanced operations related to multiresolution model construction: LODStrips model construction and LODTree model construction.

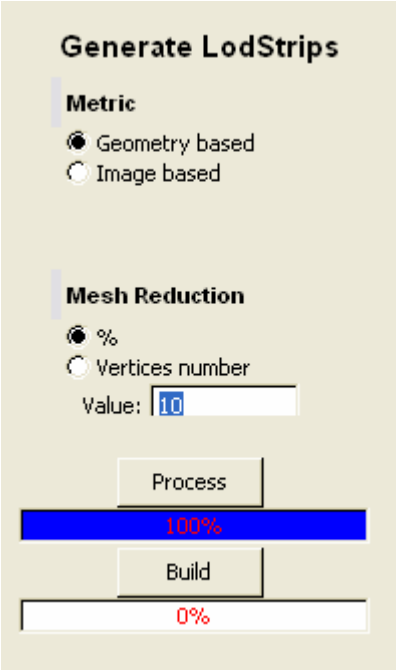
LODStrips Model Construction

As will be explained later, a LODStrips model takes as input a single mesh and transforms it to another mesh associated to a multiresolution sequence file. This file is used to perform the level of detail operations. The contents of each file will be described in detail in the LODStrips construction section.

Once a mesh has been opened in our application we build an LODStrips model as follows. Select the *Generate* option under the LODStrips menu. This will bring up a panel similar to the simplification panel. This is necessary because the LODStrips constructor needs to simplify the mesh to a given level of detail (which will be the minimum LOD) and save the simplification steps needed to take the original mesh to its simplified form. This process is independent of the simplification method as well as the metric used.

Now the application allows the user to select the simplification options that better suit his/her needs. Figure 22 shows the LODStrips control panel.

When the user clicks on the *Process* button, the simplification step is performed and the resulting simplified mesh shown on the screen.



The construction process stops here to give the user the opportunity to see the resulting mesh at the minimum LOD of the multiresolution model. Thus, if the simplification options are wrong, the user can undo the simplification step (*undo* option under the *Edit* menu), and perform the simplification step again.

Once the user agrees with the resulting simplified mesh, the final step can be started by clicking on the *Build* button. First, the application asks the user for the filename to be used to save the results. These results are stored in two new files: one containing the stripified original mesh and the other containing the simplification sequence. These have extensions *.mesh* and *.lods* respectively, with the file name that the user entered in the previous dialog box.

The LOD construction process is quite slow at the time this report is being written. So we are looking for optimizations to workaround this issue.

Figure 22: LODStrips generation panel

LODTree Model Construction

The LODTree model construction process is similar to the LODStrips construction process. However, an extra step is required from the user: the selection of the leaves of the tree. This is a required step. In fact, if the user tries to generate a LODTree model without having the leaves selected, the program will ask the user to perform such task. This is done using the *Select leaves* option under the *Edit* menu. When the option is selected, a control panel opens allowing the user to choose any sub-mesh of the tree as the *leaf sub-mesh*. The rest of sub-meshes will be assumed to be part of the trunk.

Once the leaves are selected, the user can start the generation process by selecting the *Generate* option under the *LodTrees* menu. A generation panel identical to the LodStrips one opens, showing the same options like the LODStrips panel. This is because the trunk of the tree is encoded as a LODStrips object, and it requires the same options as a LODStrips model to be generated. The leaves are simplified using a special leaf-collapse method that requires no user input (except for the minimum leaf count). So this is the only option for leaf simplification in the panel.

The process to generate the LODTree consists of the same steps described for LODStrips models. First, the entire tree is simplified to the lowest possible level of detail. Then, if the user agrees with the result, the process continues and generates the LODTree model, saving it to disk.

A LODTree model in disk is stored in 3 different files: a *.mesh* file containing the geometry, a *.lods* file representing the simplification sequence for the trunk, and a *.lodt* file containing the simplification sequence for the leaves. Later we explain this in more detail.

2.7.3.4. Multiresolution Model Visualization

Once a multiresolution model is constructed and saved to disk, we can display it using the *visualize* option under the *LODStrips* and *LODTrees* menus. The application will ask the user to select and load the files of the model: a .mesh, a .lods and a .lodt file for a LODTree model, or a .mesh and a .lods file for a LODStrips model). That will bring up panels for manipulating the level of detail of the model. In the case of a LODTree model, the panel looks like the one in the following figure.

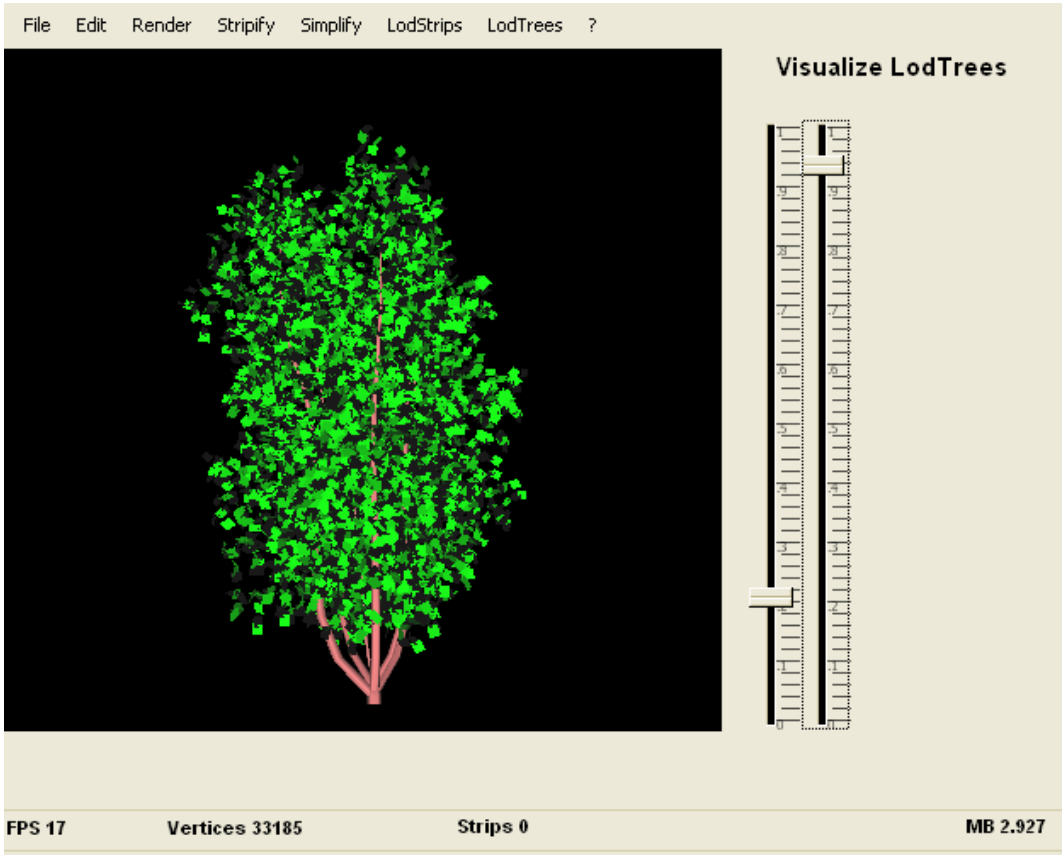


Figure 23: LODTree visualization panel in GeoTool

The panel contains two sliders: the left one allows changing the LOD of the trunk and the right one allows changing the LOD of the leaves. The LODStrips LOD control panel is similar to this one but has only one slider to manipulate de LOD of the mesh.