GAMETOOLS

ADVANCED TOOLS FOR DEVELOPING HIGHLY REALISTIC COMPUTER GAMES

FINISHED MODULES FOR GEOMETRY

Document identifier:	**GameTools-4-D4.4-02-1-1-Finished Modules for Geometry**
Date: (use "update field" Word function, right mouse button)	**13/03/2007**
Work package:	**WP04: Geometry**
Partner(s):	**UJI, UPV**
Leading Partner:	**UJI**
Document status:	**Approved**
Deliverable identifier:	**D4.4**

Abstract: This final technical report describes the different algorithms used on the implementation and finalization of the Geometry module.

Delivery Slip

	Name	Partner	Date	Signature
From	Miguel Chover	UJI	10-03-2007	
Reviewed by	Moderator and reviewers	ALL		
Approved by	Moderator and reviewers	ALL		

Document Log

Issue	Date	Comment	Author
1-0	02-03-2007	First draft	Miguel Chover
1-1	08-03-2007	Final Version	Miguel Chover

Document Change Record

Issue	Item	Reason for Change

Files

Software Products	User files / URL
Word	gametools-ist-2-004363-4-d4.4-02-1-1-finished modules for geometry.doc (use “update field” Word function)

Finished modules for geometry

Contents

Chapter 1

Introduction

1.1 Overview

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, are introduced.

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 discrete models that, although more limited, are perfectly adapted to current graphics hardware. Specifically, researchers 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 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, and its representation is very realistic. 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 meshes.

1.2 Developed models

We have developed two different multiresolution models to face this problem: a model for generic meshes (LodStrips) and a model specifically designed to handle trees and plants (LodTrees). We also needed to create a set of stripification and simplification algorithms, as a base for the creation of our multiresolution models. We also have developed a module designed to efficiently use the previous multiresolution algorithms in massive scenes while preserving the overall performance and avoiding sudden stalls in the graphics pipeline. Finally, a module (LodManager) to handle efficiently large scenes composed of hundreds of multiresolution objects has been developed.

To make easy the construction of multiresolution objects a stand-alone application has been developed: the GeoTool. This application can also be used as a front-end tool to use the developed modules: stripification, simplification, and multiresolution construction and visualization.

In order to prove the validity of our algorithms in real world engines and software, we have developed some demos that demonstrate the integration in the Ogre and Shark engine.

1.3 Document organization

This document is organized according to the developed modules commented previously. For each module, an explanation about the theory involved is provided as well as the specification of the final API.

First, a quick guide that shows how to use the Geotool is presented, including a chapter showing how to create multiresolution models. After that, the multiresolution programming guide is presented. This shows how to integrate the multiresolution run-time modules into existing aplications. the next chapter describes the integration of our modules into the Ogre and Shark 3D engines. The rest of the chapters are intended as a reference guide of the most important classes of the geometry modules.

Chapter 2

GeoTool usage

2.1 Introduction

This is a standalone application used to review the performance of the Game-Tools Geometry Library and a useful tool to quickly generate multiresolution models.

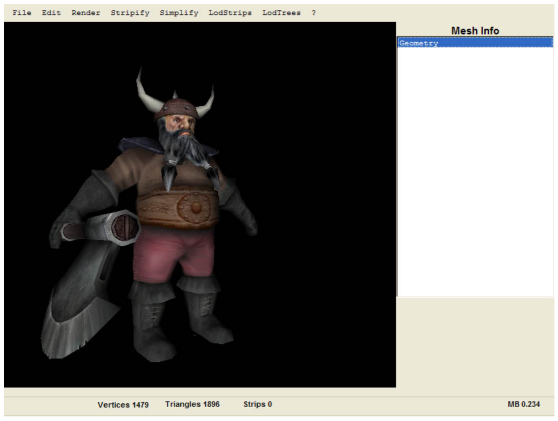
2.2 Description

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

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.

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

Basic operations: these operations involve file, edit and render operations. The rendering primitive can be changed (wire mode, solid mode) 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 LOD-



Vertices 1479 Triangles 1896 Strips 0 MB 0.234

Figure 2.1: The GeoTool application

Strips or LODTree model and render it.

Simple operations: these operations involve mesh stripification and simplification. These are catalogued as simple operations because they are done in a single step and return a transformed standalone mesh. Two different simplification approaches are available: geometry-driven simplification and viewpoint-driven simplification.

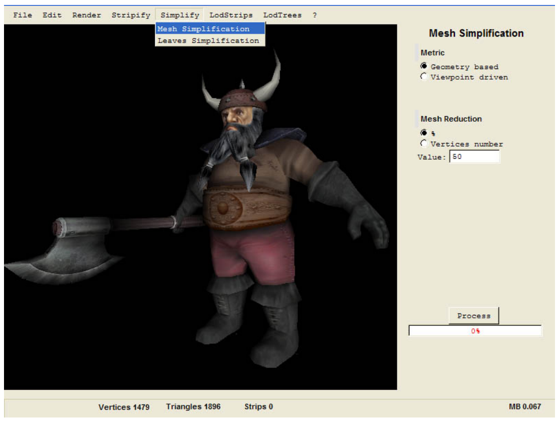
Complex operations: these operations are LODStrip construction and LODTree construction. Internally, these complex operations perform some simple operations such as stripification, simplification and vertex reordering. It is important to note that they are much more time consuming than basic operations. They take a mesh as input, then they construct a new mesh with the associated multiresolution sequence, and save the result to disk. We will explain this process in more detail in the following sections.

2.3 User Interface

The user interface of GeoTool has been designed to be easy to use. The menu bar across the top 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 vertices, strips and triangle count.



Mesh Simplification

Metric

[✓ Geometry based] Viewpoint driven

Mesh Reduction

[✓ %] Vertices number

Value: 50

0%

Vertices 1479 Triangles 1896 Strips 0 MB 0.067

Figure 2.2: The simplify menu

2.3.1 Basic operations

This section the explains basic operations performed by the GeoTool application. The first three menus *File, Edit and Render*, perform basic operations.

- **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.

  Load Textures: Allows to select the texture of the entire model. In addition, it allows us to select the texture of for a single submesh.

  Quit: Terminates the application.

- **Edit Menu**

  Undo: Gets the current mesh back to its previous state.

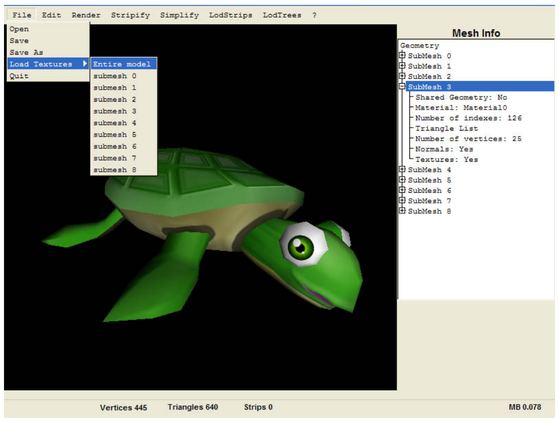
  Fit: Modifies the current view to fit the loaded mesh inside the screen.

  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 tree-top. Pushing the process button the folliage is selected.

- **Render Menu**



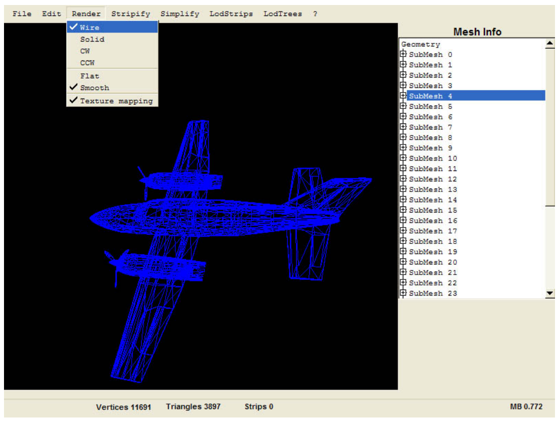
```
Geometry
⊞ SubMesh 0
⊞ SubMesh 1
⊞ SubMesh 2
⊟ SubMesh 3
   ├ Shared Geometry: No
   ├ Material: Material0
   ├ Number of indexes: 126
   ├ Triangle List
   ├ Number of vertices: 25
   ├ Normals: Yes
   └ Textures: Yes
⊞ SubMesh 4
⊞ SubMesh 5
⊞ SubMesh 6
⊞ SubMesh 7
⊞ SubMesh 8
```

```
Vertices 445    Triangles 640    Strips 0                MB 0.078
```

Figure 2.3: The open menu

Wire / Solid: Selects the geometry rendering mode: wireframe or solid.

Flat / Smooth: Selects the surface shading mode: flat or smooth (Gouraud).



```
Geometry
⊞ SubMesh 0
⊞ SubMesh 1
⊞ SubMesh 2
⊞ SubMesh 3
⊞ SubMesh 4
⊞ SubMesh 5
⊞ SubMesh 6
⊞ SubMesh 7
⊞ SubMesh 8
⊞ SubMesh 9
⊞ SubMesh 10
⊞ SubMesh 11
⊞ SubMesh 12
⊞ SubMesh 13
⊞ SubMesh 14
⊞ SubMesh 15
⊞ SubMesh 16
⊞ SubMesh 17
⊞ SubMesh 18
⊞ SubMesh 19
⊞ SubMesh 20
⊞ SubMesh 21
⊞ SubMesh 22
⊞ SubMesh 23
```

```
Vertices 11691    Triangles 3897    Strips 0                MB 0.772
```

Figure 2.4: The render menu

2.3.2 Simple operations

The simple operations are performed in a single step of the Geometry Game Tools Library and modify the given mesh.

- **Stripify Menu:** This menu has no popup menu associated. Instead, it immediately opens the Stripification panel. The process button will start the stripification. The progress bar will show the stripification status. Only manifold meshes can be stripified. If a mesh is not manifold an error message is shown.

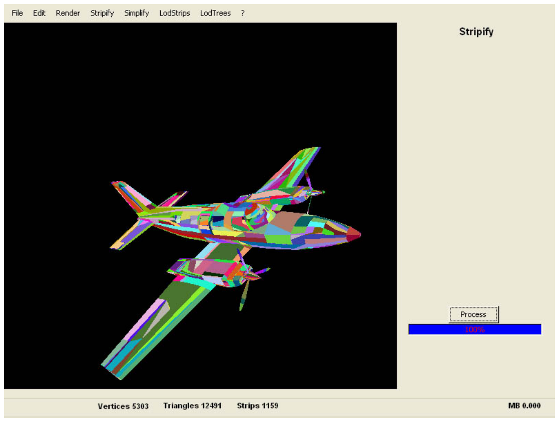


Figure 2.5: The stripify operation

- **Simplify Menu** The simplification of a mesh object can be accomplished with one of the two following simplification modes:

  Mesh simplification: Performs edge collapse simplification. Moreover, there are two ways to perform simplification in edge collapse.

    Geometry-based: Fast method of simplification. Simplify mesh objects based on geometry relation between triangles.

    Viewpoint-based: Simplify the mesh object based on image processing.

  Leaves simplification: Performs leaves collapse simplification. Only simplifies the mesh that was chosen as foliage with the *Edit/Select Leaves* menu.

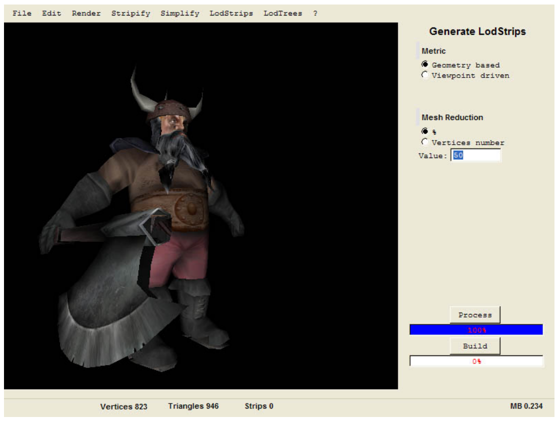
  In addition, the mesh reduction factor can be chosen as a percentage value or as a number of vertices.

2.3.3 Complex operations

The complex operations visually need several of simple geometry operations. There are two types:

- **LODStrips Menu**

Generate: Generate a mesh with its correspondent LOD (Level Of Detail) sequence. The process to accomplish this is similar to the simplification, but a new step is added. In the right panel appear a new Build button that performs the stripification and the build process. The build process gets the simplification sequence done by the simplification and the stripified model, then generates the new mesh with its correspondent LOD sequence.



Metric
(● Geometry based) ○ Viewpoint driven

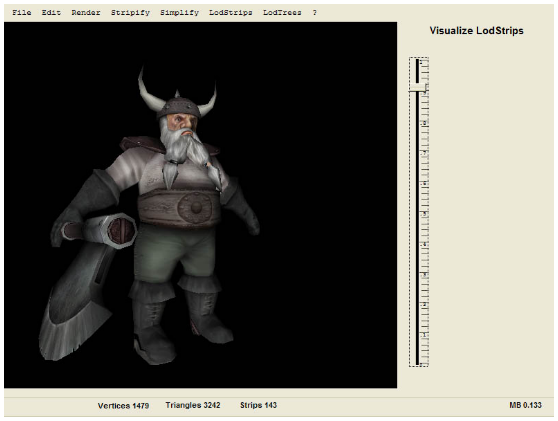
Mesh Reduction
(● %) ○ Vertices number
Value: 50

Process: 100%
Build: 0%

Vertices 823 Triangles 946 Strips 0 MB 0.234

Figure 2.6: Generating LODStrips

Visualize: This allows us to see the result of the LODStrip process. The level of detail of the object can be changed with a slide bar that appears in right panel.



Vertices 1479 Triangles 3242 Strips 143 MB 0.133

Figure 2.7: Visualize LODStrips

- **LODTrees Menu**

Generate: This process gets a tree object with the LODStrip of the trunk. The foliage should be selected the *Edit/Select Leaves* menu. The user interface operation is the same that the LODStrips one. First, simplify the foliage selecting the percent or the number of triangles desired. Finally pushing the build button the LODTree is generated.

Visualize: Two slidebars appears in right panel. One to change the level of detail of the trunk and the other to change the level of detail of the foliage.

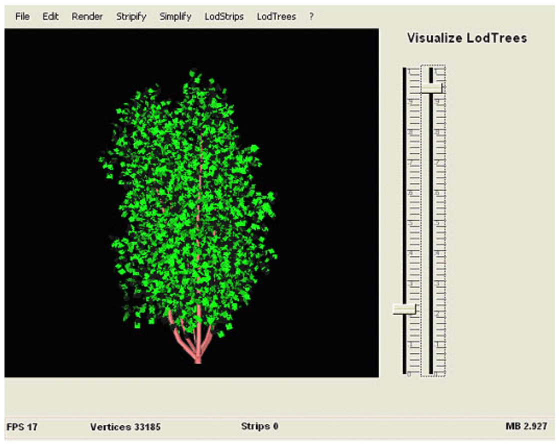


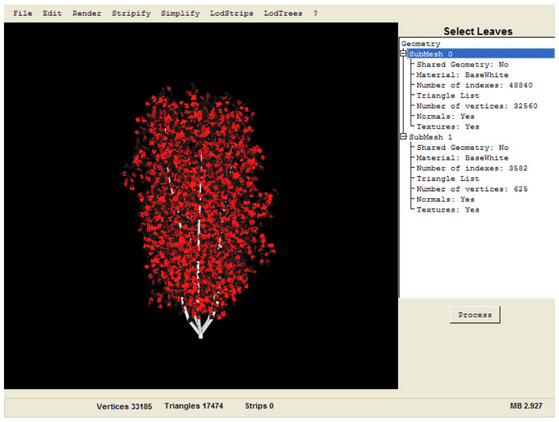
Figure 2.8: Visualize LODTree

2.4 LOD Generation

This section explains the steps that are necessaries to build a LOD object. The LOD generation involves mesh stripification and mesh simplification.

The first step is to open a mesh object. We do this with the *Open* option of the *File* menu. Next step is select option *Generate* of the *LODStrip* menu to begin the generation process (see figure 2.6). At right appears the *Generate LODStrips* panel, in which we can choose the simplification method between *Geometry base* and *Viewpoint Driven*. In addition, we must specify mesh reduction by percent or by the number of triangles to obtain. Pushing the *Process* button the simplification is performed and its result can be viewed at screen. At this point, if the result of simplification doesn't convince us, we can undo this step and change simplification options. In the other hand, if the results of the simplification are OK, we can build the LOD object pushing the *Build* button and filling the name of this one. If we can view the result of build process can do this by selecting the option *Visualize* of the menu *LODStrips* that makes a slide bar appear in the right panel (see figure 2.7). By moving the slide bar we can change the Level Of Detail of the object.

First of all to perform *LODTree* we must select the *Select Leaves* option of the *Edit* menu (see figure 2.9). This makes that the *Select Leaves* panel appears. In this panel there is a mesh info browser that shows the submeshes in which is divided the object and relevant geometric information. Clicking a submesh in the browser paints it in red in the view window. We must choose the treetop submesh and push the button *Process* to make the selection of leaves. Next we can go to do the LODStrip procress as described before (see figure 2.6). Finaly the result of the LODStrip process is used as input of the LODTree one. Using the *LODTree* menu we can perform the LODTree building following the same steps than a LODStrip process.



```
Geometry
⊟SubMesh 0
 ├Shared Geometry: No
 ├Material: BaseWhite
 ├Number of indexes: 48840
 ├Triangle List
 ├Number of vertices: 32560
 ├Normals: Yes
 └Textures: Yes
⊟SubMesh 1
 ├Shared Geometry: No
 ├Material: BaseWhite
 ├Number of indexes: 3582
 ├Triangle List
 ├Number of vertices: 625
 ├Normals: Yes
 └Textures: Yes
```

Vertices 33185 Triangles 17474 Strips 0 MB 2.927

Figure 2.9: Select Treetop

Chapter 3

The Geometry Programming guide

3.1 Introduction

This chapter is conceived as a guide to use the LOD models constructed with the GeoTool in external applications. The Ogre Engine is used in the examples.

A LOD model is represented in the GameTools Geometry Package with the *Geometry::LodObject* class. There are two types of LOD objects: those that represent a general LOD model (implemented in the *Geometry::LodStripsLibrary* class) and those that represent trees (*Geometry::LodTreeLibrary* class). Both LOD model types store its data in a binary file format, extended from the Ogre Mesh file format specification. These extensions involve the creation of two new chunks of data that are appended to the Mesh file. These two new chunks are transparent for standard Ogre Mesh loaders, and therefore they will only be detected by our software. These chunks contain the LOD information required to change the level of detail of an object in real time.

3.2 Concepts

The *Geometry::LodStripsLibrary* class represents a general mesh which is able to change its level of detail. The mesh can be composed of several sub-meshes. The LOD algorithm affects the whole object, it's not possible to restrict to any submesh.

The *Geometry::LodTreeLibrary* represents a tree or a plant which is able to change its level of detail. One of those submeshes is dedicated completely to store the foliage. The rest are dedicated to store the trunk. The trunk itself uses internally the LodStrips algorithm to manage the LOD the trunk. The class offers methods to change the level of detail of the foliage and the trunk separately.

3.3 Usage

First of all, an extended mesh file must be created using the GeoTool application. This file must contain the required LOD info to run the LodStrips algorithm (see section 2.4 for more details). To use this file into an external application as a LOD object the general process will be:

1. Initialization:
    (a) Load the geometry information from the extended mesh file.
    (b) Load the LOD information.
    (c) Create an instance of LOD model feeding it with the LOD data.
2. To change the level of detail:
    (a) Call the GoToLOD method.

The following sections will explain this process in more depth.

3.3.1 The IndexData interface

The *Geometry::LodStripsLibrary* and *Geometry::LodTreeLibrary* classes can calculate the changes in the level of detail of an object. These classes have been designed to be API independent. However, updating the indices of a mesh is an API dependant task, because it's dependant on how the client application stores the indices to render the geometry. For example, the client application can use OpenGL or Direct3D, which manage their indices in a very different way, but our library must support them.

Therefore, the *Geometry::LodStripsLibrary* and *Geometry::LodTreeLibrary* classes know how to calculate the new set of indices to render the geometry, but they do not know how or where to store the resulting indices. To solve this, we have developed our library using an IndexData abstraction interface. The LodStrips and LodTree algorithms use this interface to communicate with the client code to set the indices at a given LOD.

The user must inherit a custom class from the IndexData interface and implement its virtual methods to provide the desired functionality. Thus, an instance of the user custom IndexData class will be passed to the *Geometry::LodStripsLibrary* and *Geometry::LodTreeLibrary* classes at creation time. The code bellow shows the IndexData class interface.

```
class IndexData
{
public:
  IndexData(void){}
  virtual ~IndexData(void){}

  virtual void Begin(unsigned int submesh, unsigned int numinds)=0;
  virtual void SetIndex(unsigned int i, unsigned int index)=0;
  virtual void End(void)=0;
  virtual void BorrowIndexData(const IndexData *)=0;
};
```

The functions above will be called when a LodStrips or LodTree instance must change the level of detail. The meaning of each method is described below:

- `Begin(subm,numi)` indicates that the following `numi` indices must be modified on the submesh `subm`. Hint: This is a good place to *lock* an index buffer.
- `SetIndex(i,idx)` specifies the new value for the index at the position `i`.
- `End()` indicates that the changes made to this submesh are finished. Hint: This is a good point to *unlock* an index buffer.
- `BorrowIndexData(indexdata)` indicates that the object must use temporarily the new `indexdata` set. This is only used by the LodManager. If you're not planning to use it, you can leave this function unimplemented.

3.3.2 Initialization

The Geometry module includes a class to load an extended mesh file to extract all information it contains. This class is *Geometry::MeshLoader* and it is simply used creating an instance of the class and using its `load()` method. Here is an example:

```
Geometry::GeoMeshLoader meshloader;
Geometry::Mesh *mesh = meshloader.load("sphere.mesh");
```

This call returns a *Geometry::Mesh* object which describes the geometrical information of a mesh (vertices, indices, bones, ...). To be able to create LOD object, the LOD information of the file must be extracted. The method `GetLodStripsData()` of the *Geometry::MeshLoader* class can be used to obtain the LOD information. If the function returns `NULL` then the loaded file didn't contain LOD info and it can't be used as a multiresolution object.

If it doesn't return `NULL` a LOD object can be instantiated using the *Geometry::LodStripsLibrary* class. Thus, a LOD object is created this way:

```
Geometry::LodStripsLibrary *lodobj;
lodobj=new Geometry::LodStripsLibrary(meshloader.GetLodStripsData(),
                                      mesh,
                                      new CustomIndexData(ogreMesh));
```

A LodTree object is instantiated in a similar way. The main difference is that we must also extract the LodTree-related LOD info (the foliage simplification sequence) as well as the LodStrips related info (used for the trunk). This LodTree info is a requirement to instantiate a *Geometry::LodTreeLibrary* object. The code below shows how to load the info from a file and instantiate a LodTree.

```
// load LOD info from the object
meshloader=new Geometry::GeoMeshLoader;
Geometry::Mesh *mesh = meshloader->load("tree.mesh");

if (!meshloader->GetLodStripsData())
  OGRE_EXCEPT(1, "LOD info for the trunk not found","LOD Demo");
if (!meshloader->GetTreeSimpSeq())
  OGRE_EXCEPT(1, "LOD info for the foliage not found","LOD Demo");

myLodTree = new Geometry::LodTreeLibrary(meshloader->GetLodStripsData(),
                                         meshloader->GetTreeSimpSeq(),
                                         mesh,
                                         new CustomIndexData(ogreMesh));
```

3.3.3 Changing the LOD

At this moment we have already instantiated a multiresolution model. The LOD changing involves finding the new set of indices that describe the object at a given LOD. Both classes, *Geometry::LodStripsLibrary* and *Geometry::LodTreeLibrary* calculate this set of indices and apply them to the mesh with the IndexData user interface implemented by the user. To change the LOD of a multiresolution object the user must call the method `GoToLod()` specifying the desired level of detail in the range [0,1] (0 for the minimum and 1 for the full LOD).

```
myLodObject->GoToLod(lodfactor); // lodfactor \in [0,1]
```

After calling this function, the target mesh should have changed effectively its level of detail and it should be ready to be rendered. The following code shows an example implementation (used in the Ogre demo applications) as a bridge between the IndexData interface and the index buffers used by Ogre to handle the geometry of the target mesh.

```
class CustomIndexData : public Geometry::IndexData
{
private:
  Ogre::Mesh *targetMesh;
  Ogre::HardwareIndexBufferSharedPtr ibuf;
  Ogre::RenderOperation mRenderOp;
  unsigned long* pIdx;
public:
  CustomIndexData(Ogre::Mesh *ogremesh):Geometry::IndexData(){
    targetMesh=ogremesh;
    pIdx=NULL;
  }
  virtual ~CustomIndexData(void){}

  virtual void Begin(unsigned int submeshid, unsigned int indexcount){
    targetMesh->getSubMesh(submeshid)->_getRenderOperation(mRenderOp,0);
    ibuf = mRenderOp.indexData->indexBuffer;
    mRenderOp.indexData->indexCount = indexcount;
    pIdx = static_cast<unsigned long*>(ibuf->lock(Ogre::HardwareBuffer::HBL_NORMAL));
  }
  virtual void SetIndex(unsigned int i, unsigned int index){
    pIdx[i] = index; //lodStripsLib->dataRetrievalInterface->GetIndex(k+offset);
  }
  virtual void End(){
    ibuf->unlock();
  }
  virtual void BorrowIndexData(const Geometry::IndexData *){}
};
```

The code above is quite straightforward. The `Begin()` function is called each time a submesh is going to be modified. Thus, the example locks the appropiate index buffer to modify it in the following `SetIndex()` calls. When the mesh update is complete, the `End()` function is called and we use it to unlock the previously opened index buffer. The funcion `BorrowIndexData` is only used by the LodManager, so if one does not need it, the funcion can remain unimplemented.

Chapter 4

Runtime module integration

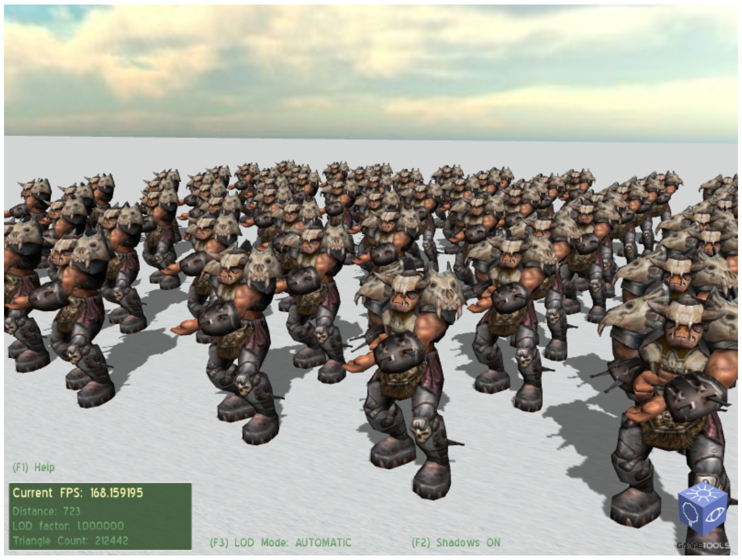
4.1 Introduction

Integration of our modules in existing applications and game engines is a very important task. The geometry modules have been designed as flexible as possible, so that they can be successfully integrated in any situation.

This chapter is organized as follows: first, the integration into Ogre will be explained and, later, all issues about the integration into Shark will be exposed as well as the solutions provided to facilitate the task.

4.2 Integration into Ogre

The integration of our modules into Ogre-based applications is a very straightforward task. This is due to the fact that the GameTools Geometry Modules were initially designed having this engine in mind. The file format used to store multiresolution models is an extension of the Ogre mesh file format which has backwards compatibility with the standard file format. The Ogre mesh file format is a chunk based format. When a multiresolution model is constructed, the results are stored in the file in a specificaly designed chunk. This way, both geometry and multiresolution data are stored in a compact file which has backwards compatibility with standard Ogre applications, as the default Ogre behaviour is to ignore unknown chunks. For more information about the integration of the geometry modules into the Ogre Rendering Engine read chapter 3.



```
(F1) Help
Current FPS: 168.159195
Distance: 723
LOD factor: 1.000000
Triangle Count: 212442
(F3) LOD Mode: AUTOMATIC        (F2) Shadows ON
```

Figure 4.1: Screenshot of the LodStrips demo.

4.2.1 Demos

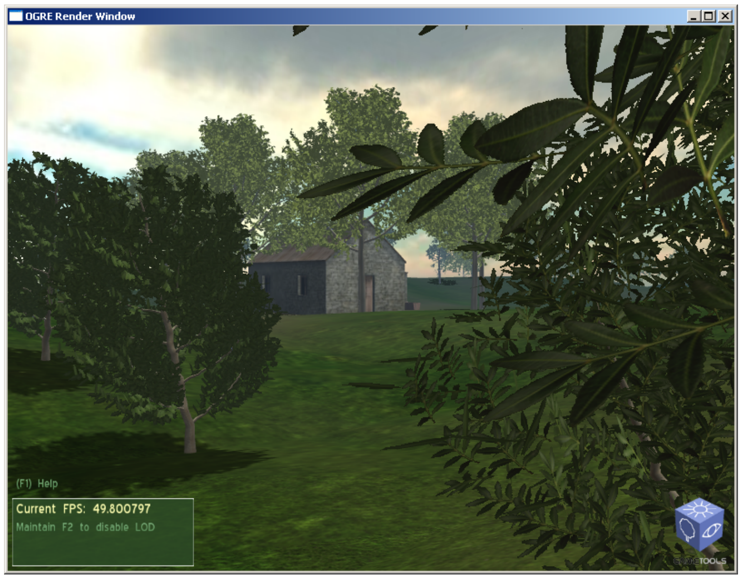
We have developed some demos to demonstrate the geometry modules and its integration into a graphics rendering engine: The Ogre Rendering Engine.

4.2.1.1 LodStrips demo

The first demo demonstrate the LodStrips multiresolution run-time library (*Geometry::LodStripsLibrary*). The application (figure 4.1) shows a group of models which are able to change their level of detail depending on the distance of the group of objects to the camera. The information panel on the bottom-left corner of the screen shows the current LOD factor, frames per second and the amount of geometry sent to the renderer. It can be seen how the frame rate increases as the LOD decreases.

The level of detail can be calculated in two ways: automatic LOD (based on the distance of the group to the camera) and manual LOD (the user changes the level of detail independently from the distance), which changes the level of detail of the objects manually. This last mode is useful to see the meshes in detail even when their level of detail is set to the minimum.

The demo also shows how LOD operations can be minimized by grouping some instances of the same model to be managed by a single LodStrips multiresolution model. This is useful when some models need similar levels of detail and will improve the overall performance.



```
Current FPS: 49.800797
Maintain F2 to disable LOD
```

Figure 4.2: Screenshot from the LodTrees demo

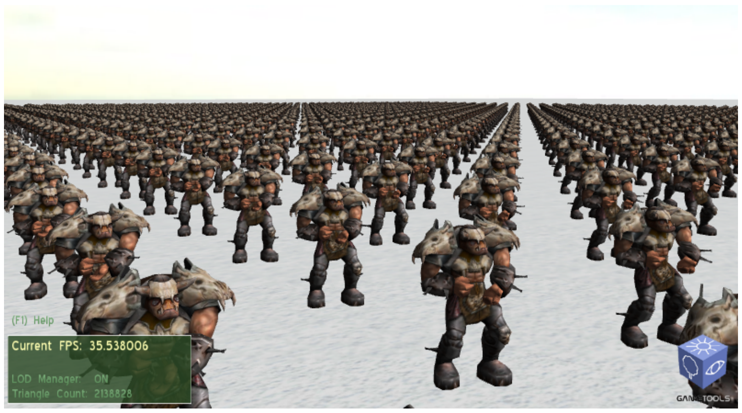
4.2.1.2 LodTrees demo

This demo presents the LodTrees multiresolution run-time library (*Geometry::LodTreeLibrary*). The demo is composed of some groups of multiresolution trees. The LOD of each group, composed by some trees of the same type, is managed by a single LodTree instance to optimize performance. The result is a forest of multiresolution trees which change its level of detail depending on the distance to the camera of each one of these groups. Figure 4.2 shows an image of the LodTrees demo.

4.2.1.3 LodManager demo

The LodManager demo features a massively populated scene composed of more than a thousand models. Each one of them is attached to an inpedendent multiresolution model instance that manages its level of detail. To manage the level of detail of such a vast scene, we introduce the use of the LodManager, which decides whether an object can freely change its level of detail or just has to borrow an already calculated LOD snapshot. For more details about this technique and its advantages see chapter 7.

The demo allows the user to enable or disable the LodManager capabilities to show the difference in performance. The LodManager is able to keep the bottleneck of the application in the graphics engine, not in the LOD calculations.



```
Current FPS: 35.538006

LOD Manager: ON
Triangle Count: 2138828
```

Figure 4.3: A view of the massive scene of the LodManager demo.

4.3 Integration into Shark3D

The integration into Shark 3D has been more complicated compared to the integration into Ogre. This is completely reasonable because our geometry classes are based on Ogre: the file format used in our algorithms is the Ogre mesh file format extended (which can be directly read from Ogre) and the class design is very similar to the Ogre's design (class Mesh, class SubMesh, class VertexBuffer).

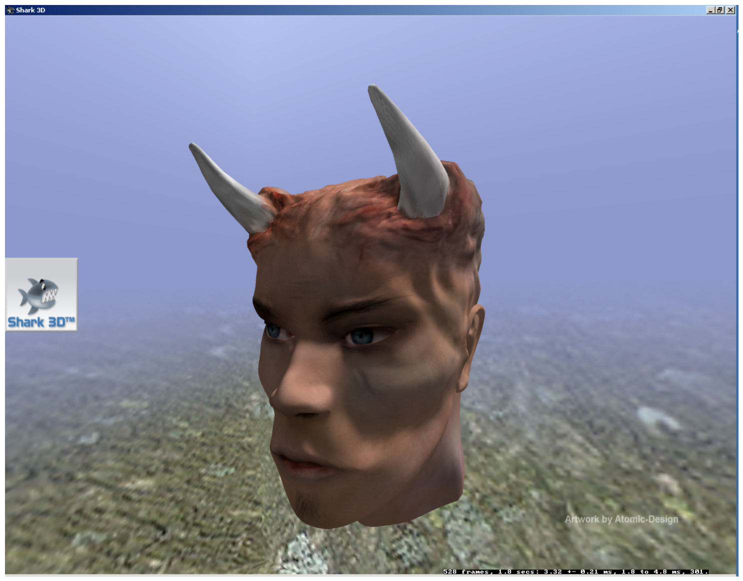
However, the architechture and design of Shark 3D also poses some inconvenients for the integration. The main problem is the fact that the current version of Shark does not offer support for triangle strips. This problem forced us to develop a new rendering module for Shark 3D which lets the engine use the triangle strips drawing primitive, because our LodStrips algorithm is based on triangle strips.

Moreover Shark 3D mesh file format and the production pipeline of Shark 3D mesh files are not suitable for our purposes. It was necessary to implement a new mesh loader module for Shark 3D to allow the engine to load multiresolution models in our own file format.

To summarize, the following list presents the modules we needed to develop to integrate our multiresolution model into Shark 3D:

- **A module to load the *.mesh* file format.** This module loads multiresolution and geometry data from our own file format. The geometry data is then passed to Shark 3D and the multiresolution data is loaded into memory to be used by our run-time multiresolution library.

- **A new renderer module.** This is necessary because the standard renderers for Shark 3D did not support triangle strips, which is a mandatory



```
528 frames, 1.8 secs: 3.32 +- 0.21 ms, 1.8 to 4.8 ms, 301.
```

Figure 4.4: A shot of the Devil's Head demo used as an example of the integration.

feature for using the LodStrips multiresolution model. Shark 3D rendering power rely mostly in its Shader Component model. We use this Shader Component system to extract the level of detail before each frame is about to be rendered. The LOD factor is calculated based on the distance of the object to the camera. Therefore, a new renderer module has been developed to both change the level of detail and to render the object using the triangle strips rendering primitive.

4.3.1 Demo

Once all necessary modules have been developed, the demo can finally be constructed. The demo shows the integration of the LodStrips algorithm into the Shark 3D game engine. For the demo we have used the original Devil's Head demo built by Spinor changing the Devil's Head model with a multiresolution version of the same model.

The demo features a multiresolution model which is able to change its level of detail depending on the distance to the camera. As we wanted to meassure performance improvements when using our multiresolution algorithm we have reduced the complexity of per-pixel effects in the demo. Therefore, the model is rendered using a per-vertex lighting method, instead of the original bump-mapped light technique. Figure 4.4 shows a screenshot of the demo.

Chapter 5

LodStrips multiresolution module

5.1 Introduction

Multiresolution modelling is a useful way to deal with the rendering of large scenes. Multiresolution modelling allows us to change the level of detail of the objects in the scene in order to optimize the amount of geometry sent to the renderer. There are several criteria to decide which level of detail must be used to represent an object. Mainly, the most used criterion is the visual importance of the object, using for example the distance of the object to the camera or the size of the object's bounding volume in screen space.

Working with the current multiresolution models poses the problem of dealing with high level of detail extraction times and excessive storage costs. 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.

5.2 Geometry::LodStripsLibrary Class Reference

The **LodStripsLibrary** (p. 29) interface class.

`#include <GeoLodStripsLibrary.h>`

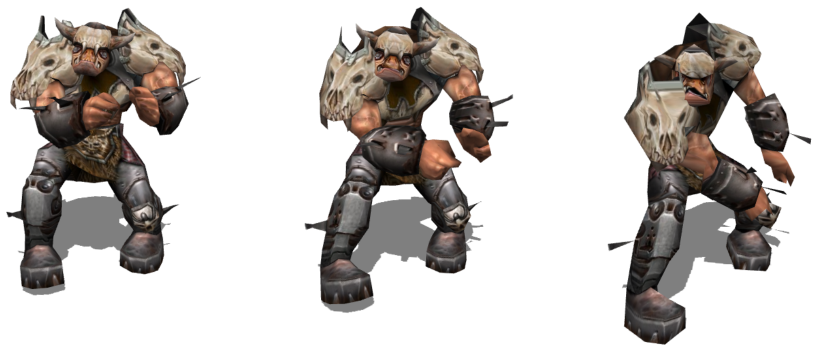


Figure 5.1: An animated LodStrips model. The figure shows how a mesh can be animated while changing the level of detail.

Public Member Functions

- **LodStripsLibrary** (const LodStripsLibraryData ∗, **Mesh** ∗geomesh, IndexData ∗userindexdata)

  Class constructor.

- ∼**LodStripsLibrary** (void)

  Class destructor.

- virtual void **GoToLod** (Real)

  Changes the level of detail of the object to a specified factor.

- uint32 **MaxFaces** () const

  Returns the number of triangles of the highest LOD.

- uint32 **MinFaces** () const

  Returns the number of triangles of the lowest LOD.

- uint32 **GetValidIndexCount** (int submeshid) const

  Returns the index count at the current LOD of a certain submesh.

- uint32 **GetTotalStripCount** (void) const

  Retrieves the total number of strips of all submeshes.

- uint32 **GetSubMeshtripCount** (int submeshid) const

  Returns the number of strips of a given submesh.

- uint32 **GetCurrentTriangleCount** (void) const

Gets the triangle count at the current LOD.

- virtual Real **GetCurrentLodFactor** (void) const

  Gets the current LOD factor.

5.2.1 Detailed Description

The **LodStripsLibrary** (p. 29) interface class.

This module contains functions that handle the levels of detail of the input multiresolution polygonal meshes. For any given resolution of an 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.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 Geometry::LodStripsLibrary::LodStripsLibrary (const LodStripsLibraryData ∗, Mesh ∗ *geomesh*, IndexData ∗ *userindexdata*)

Class constructor.

Constructs a LodStrips multiresolution object from:

- The LodStrips decimation information (this can be obtained using the class Geometry::MeshLoader).
- The **Mesh** (p. 56) object defining the geometry of the model at its most detailed level of detail (this can be obtained using the class Geometry::MeshLoader).
- A user-defined Geometry::IndexData instance.

5.2.3 Member Function Documentation

5.2.3.1 virtual void Geometry::LodStripsLibrary::GoToLod (Real) `[virtual]`

Changes the level of detail of the object to a specified factor.

The value specified to change the LOD must be in the range [0,1] ([min,max]). After the LOD is calculated, this function automatically updates the indices using the IndexData interface provided in the constructor.

Chapter 6

LodTree multiresolution module

6.1 Introduction

Research on real-time representation of trees and plants can be grouped into two groups: algorithms based on geometric primitives and those called image-based rendering methods. On the one hand, image-based methods are more cheap and easy to use and offer good quality at far distances. However, they suffer from parallax and ghosting effects when rendered closer to the viewer. On the other hand, geometry-based methods are more realistic because they use real geometry primitives to represent its contents but they are more expensive to use. However, the rapid evolution that the graphics hardware is experiencing facilitates the use of geometry-based methods over image-based methods for obtaining high-quality representations.

Our library provides a geometry-based method which is able to change its level of detail in a continuous way. A LodTree multiresolution object is composed of two different sub-components: the trunk, whose level of detail is managed using a LodStrips (see chapter 5) object as it can be represented as a general mesh, and the foliage, which uses our special multiresolution model for leaves.

Our multiresolution model for leaves uses the leaf collapse as its basic operation to calculate the whole set of approximations. The information generated during simplification is used in run-time to manage the level of detail of the foliage. Figure 6.2 shows the *Chamaecyparis Lawsoniana* tree at different levels of detail using our LodTree multiresolution library.

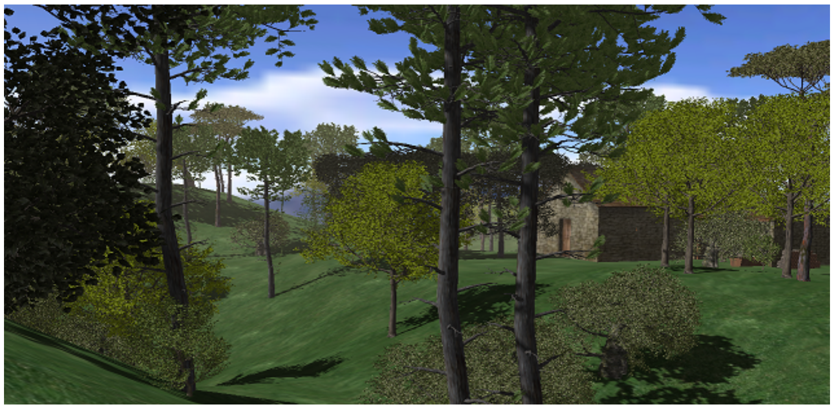
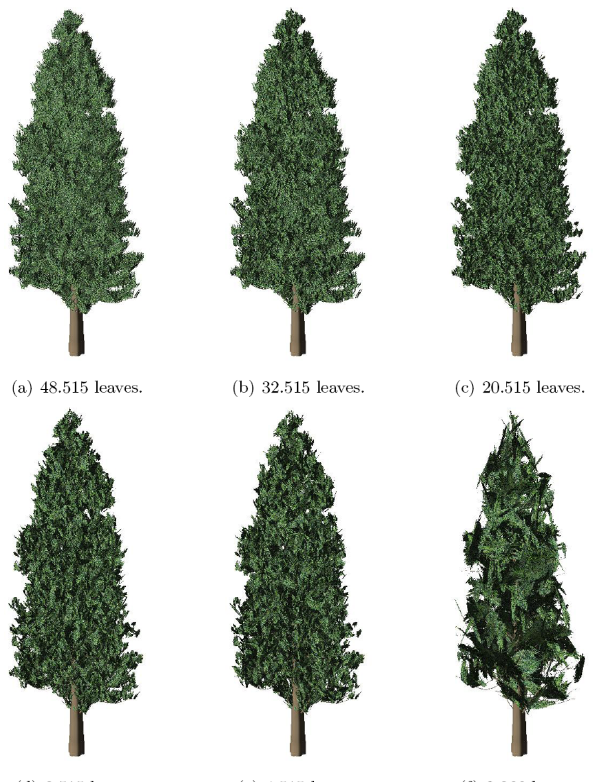


Figure 6.1: A level of detail forest



(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 6.2: Different approximations of the *Chamaecyparis Lawsoniana* tree
.

6.2 Geometry::LodTreeLibrary Class Reference

This class represents a tree object that is able to change its level of detail.

`#include <GeoLodTreeLibrary.h>`

Public Member Functions

- **LodTreeLibrary** (const LodStripsLibraryData ∗, const **TreeSimplificationSequence** ∗, **Mesh** ∗treeGeoMesh, IndexData ∗user_indexdata)

  Class constructor.

- ∼**LodTreeLibrary** (void)

  Class destructor.

- virtual void **GoToLod** (Real)

  Changes the lod of the entire object (trunk and leaves).

- void **GoToTrunkLod** (Real)

  Changes the lod of the trunk.

- void **GoToFoliageLod** (Real)

  Changes the lod of the foliage.

- virtual Real **GetCurrentLodFactor** (void) const

  Retrieves the current real LOD factor of the object.

- virtual IndexData ∗ **GetIndexDataInterface** (void)

  Retrieves a reference to the IndexData interface that manages the foliage, provided in the constructor.

- const IndexData ∗ **CurrentLOD_Trunk_Indices** (void) const

  Retrieves a reference to the IndexData interface that manages the trunk.

- uint32 **GetValidTrunkIndexCount** (int isubmesh) const

  Retrieves the index count of the trunk at the current level of detail.

- uint32 **Get_Foliage_MaxIndexCount** (void) const

  Gets the index count of the foliage at its maximum level of detail.

- uint32 **CurrentLOD_Foliage_IndexCount** (void) const

  Retrieves the index count of the foliage at the current level of detail.

- uint32 **GetLeavesSubMesh** (void) const

  Specifies which submesh of the ***Mesh*** *(*p. 56*) (provided through the constructor) represents the foliage.*

6.2.1 Detailed Description

This class represents a tree object that is able to change its level of detail.

It uses internally a LodStrips object to manage the level of detail of the trunk. The level of detail of the object initially is 1.0 (maximum accuracy).

6.2.2 Constructor & Destructor Documentation

6.2.2.1 Geometry::LodTreeLibrary::LodTreeLibrary (const LodStripsLibraryData $*$, const TreeSimplificationSequence $*$, Geometry::Mesh $*$ *treeGeoMesh*, Geometry::IndexData $*$ *user_indexdata*)

Class constructor.

Constructs an object from:

- LodStrips decimation info (for the trunk).
- LodTree simplification info (for the foliage).
- The **Mesh** (p. 56) containing the geometry of the tree.
- A user-defined Geometry::IndexData instance. For more information about how to use the IndexData interface see chapter 3.

6.2.3 Member Function Documentation

6.2.3.1 virtual void Geometry::LodTreeLibrary::GoToLod (Real) `[virtual]`

Changes the lod of the entire object (trunk and leaves)

The value specified to change the LOD must be in the range [0,1] ([min,max]) After the LOD is calculated, this function automatically updates the indices using the IndexData interface provided in the constructor.

6.2.3.2 void Geometry::LodTreeLibrary::GoToTrunkLod (Real)

Changes the lod of the trunk.

The value specified to change the LOD must be in the range [0,1] ([min,max]) After the LOD is calculated, this function automatically updates the indices using the IndexData interface provided in the constructor.

6.2.3.3 void Geometry::LodTreeLibrary::GoToFoliageLod (Real)

Changes the lod of the foliage .

The value specified to change the LOD must be in the range [0,1] ([min,max]) After the LOD is calculated, this function automatically updates the indices using the IndexData interface provided in the constructor.

6.2.3.4 uint32 Geometry::LodTreeLibrary::GetValidTrunkIndexCount (int *isubmesh*) const `[inline]`

Retrieves the index count of the trunk at the current level of detail.

6.2.3.5 uint32 Geometry::LodTreeLibrary::Get_Foliage_MaxIndexCount (void) const

Gets the index count of the foliage at its maximum level of detail.

6.2.3.6 uint32 Geometry::LodTreeLibrary::GetLeavesSubMesh (void) const `[inline]`

Specifies which submesh of the **Mesh** (p. 56) (provided through the constructor) represents the foliage.

The rest submeshes are considered as part of the trunk. This function is useful because the foliage must be rendered with triangle lists and the trunk must be rendered with triangle strips.

Chapter 7

LodManager module

7.1 Introduction

Changes in the level of detail have associated a CPU consumption time, needed to calculate and update the object to its new rendering state. This issue is specially problematic when dealing with scenes with lots (some hundreds or even thousands) of LOD objects. In this case, changing the level of detail of the objects without any control could cause the application interactivity to drop. Many articles were written in the early days of the GPUs when it was advisable to spend some CPU processing time to optimize the GPU rendering process. Nowadays, due to the great scalability of the graphics cards, we must revise all that related work to provide an updated and practical viewpoint of that situation: overloading the CPU is a delicate task that in most cases will cause it to be a bottleneck for the graphics hardware.

Therefore, the aim of this module is to develop a LOD manager with very low CPU requirements, freeing the CPU by minimizing the number of real changes in levels of detail. Nowadays, the GPUs have experienced a great evolution in their gross horsepower. That has provoked that real-world real-time applications tend to be CPU bounded, i.e. the CPU limits the GPU. Thus, developing heuristics that involve high CPU processing times can be counterproductive. Thus, the objective of this chapter is to provide a simple yet effective method that lowers the CPU usage in order to keep the bottleneck on the GPU.

7.2 Geometry::LodManager Class Reference

This class implements a LOD manager for level of detail objects such as Lod-Strips and LodTree objects.

`#include <GeoLodManager.h>`

Public Member Functions

- **LodManager** (Real near, Real far, const Geometry::Vector3 &campos, int numslots=10)

    LodManager *(*p. *39) constructor.*

- ∼**LodManager** (void)

    Class destructor.

- void **AddLodObj** (const std::string &name, LodObject ∗, const Geometry::Vector3 &)

    Adds a new LOD object (a LodStrips or a LodTree object).

- void **UpdateLOD** (void)

    Performs all needed LOD updates.

- void **UpdateCamera** (const Geometry::Vector3 &)

    Updates the camera position.

- void **UpdateLODObjectPos** (LodObject ∗, const Geometry::Vector3 &)

    Updates the position of a certain LodObject.

Public Attributes

- bool **always_calculate_lod**

    Set it to true to force all LOD calculations (disable the ***LodManager*** *(*p. *39) features).*

- bool **force_highest_lod**

    If set to true, all objects will be set to their highest level of detail.

7.2.1 Detailed Description

This class implements a LOD manager for level of detail objects such as LodStrips and LodTree objects.

This class automatically manages the level of detail of all objects added to the LOD manager. The key idea is to minimize the number of real changes in levels of detail because they consume CPU and can cause stalls and CPU bottlenecks in massive applications. The **LodManager** (p. 39) solves this issue by deciding which objects have to change the level of detail and which objects can just share an already calculated one. The system also decides whether an object should change its level of detail or whether it is not necessary because it would not affect the performance. This drastically reduces the CPU usage. in a completely transparent way to the user.

7.2.2 Constructor & Destructor Documentation

7.2.2.1 Geometry::LodManager::LodManager (Real *near*, Real *far*, const Geometry::Vector3 & *campos*, int *numslots* = `10`)

LodManager (p. 39) constructor.

Constructs a **LodManager** (p. 39) object from the following parameters:

- The near and far distances that define the active LOD range.
- The initial camera position.
- The number of LOD slots in the snapshot list (this parameter is optional).

7.2.3 Member Function Documentation

7.2.3.1 void Geometry::LodManager::AddLodObj (const std::string & *name*, LodObject *, const Geometry::Vector3 &)

Adds a new LOD object (a LodStrips or a LodTree object).

When an object is added to the system, its level of detail is automatically managed, and the client application does not need to manually change its level of detail. The parameters needed to add an object to the system are:

- The class name of the object. The class name is the type of object. This is used by the system to know which objects are compatible for sharing levels of detail.
- A reference to the LOD Object itself (a LodStrips or a LodTree object).
- The initial position in the 3D space of the added object.

7.2.3.2 void Geometry::LodManager::UpdateLOD (void)

Performs all needed LOD updates.

This function should be called once per frame (or at other reasonable interval). It internally performs all needed steps to change the level of detail of the objects.

Chapter 8

Stripification module

8.1 Introduction

Stripification algorithms enable for a compact representation of meshes and they are a key feature of our multiresolution algorithm LodStrips. The main problem using triangle strips in a multiresolution model is that they very degenerated as the level of detail of an object decreases. These low-quality strips imply sending more information to the renderer. For this reason, we have implemented our own triangle stripification algorithm which searches for high quality strips to avoid degenerated triangles as much as possible in the whole range of levels of detail.

Figure 8.1 shows an example of triangle strip degeneration obtained after changing the level of detail. Figure 8.2 shows an example of a mesh stripification.

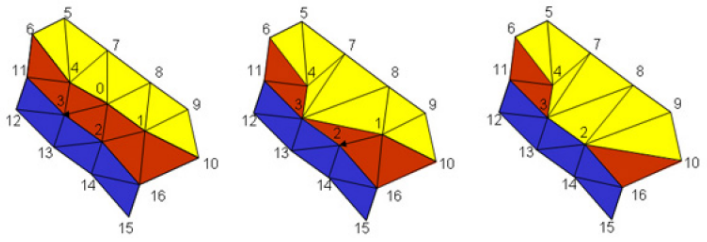


Figure 8.1: Example of triangle strip degeneration.

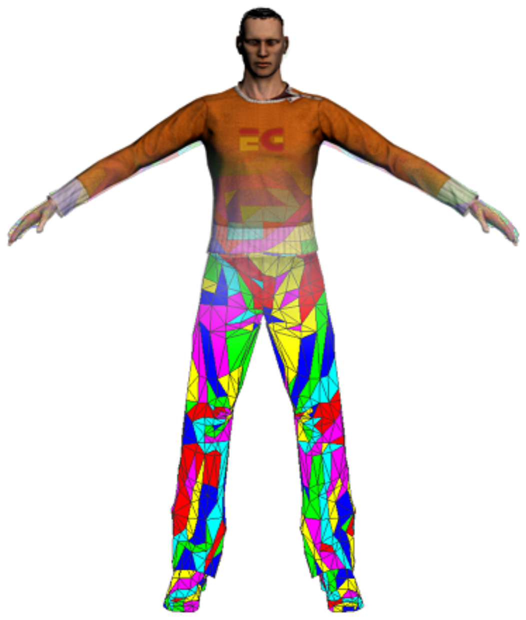


Figure 8.2: Mesh stripification.

8.2 Geometry::MeshStripifier Class Reference

Stripifier abstract interface class.

`#include <GeoMeshStripifier.h>`

Public Member Functions

- **MeshStripifier** (const **Mesh** ∗)

  Class constructor, receives as a parameter a const pointer to the object that describes a mesh.

- virtual ∼**MeshStripifier** (void)

  Virtual class destructor.

- virtual int **Stripify** (void)=0

  Starts the stripification process. This is a pure virtual method and must be overloaded in a derived class that implements a stripification algorithm.

- **Mesh** ∗ **GetMesh** (void)

  Returns the stripified mesh.

8.2.1 Detailed Description

Stripifier abstract interface class.

This module implements methods that extract triangle strips from a triangle mesh described by a **Geometry::Mesh** (p. 56) object.

The class **Geometry::MeshStripifier** (p. 44) is an abstract base class which is designed to be implemented by classes that represent custom stripifier methods.

The class Geometry::CustomStripifier inherits from this class and implements the stripifier used in the geometry modules. However, its interface is not described here because it is identical to the **Geometry::MeshStripifier** (p. 44) interface.

This module receives a pointer to a **Geometry::Mesh** (p. 56) object containing the model to be stripified, and outputs the stripified mesh, contained also in a **Geometry::Mesh** (p. 56) object.

Chapter 9

Simplification module

9.1 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 allows the generation of the whole family of levels of detail. Therefore, a simplification method is needed to construct a multiresolution object to generate the simplification sequence.

9.1.1 Simplification for general meshes

Algorithms for polygonal mesh simplification can be categorized into different classes: vertex decimation, vertex clustering, edge contractions and morphological operations. The most extended and accurate methods are those based on iterative edge contraction. Figure 9.1 shows an example of the edge collapse operation. On each operation, one or two triangles are removed and one vertex is erased.

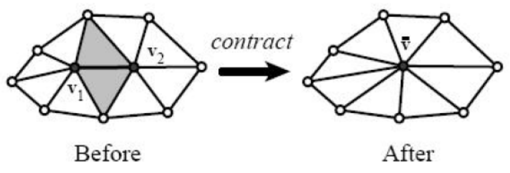


Figure 9.1: Example of edge contraction.

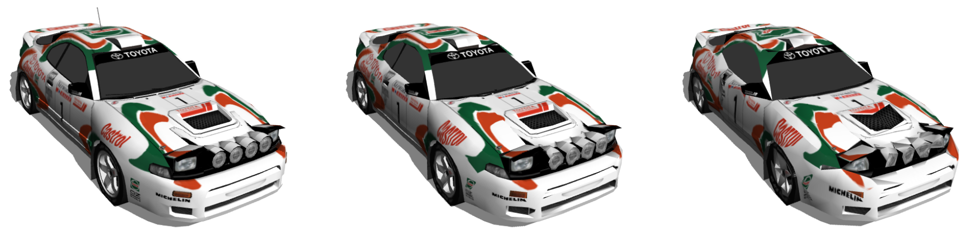


Figure 9.2: Example of geometry simplification. From left to right, the original object, the object simplified to 66% and the object simplified to 33%, respectively.

We have developed two different simplification methods to deal with general mesh simplification: Geometry-based and Viewpoint-driven simplification algorithms.

Although both methods are based on edge collapse operations, they use very different metrics to decide the order of the edges to collapse. The following sections summarize the basics of each method.

9.1.1.1 Geometry-based simplification

Our geometry-based simplification algorithm is based on edge contractions and uses a quadric error metric to decide the order of collapses. Our method is specifically designed to simplify meshes tipically used in games and real-time applications, taking into account its particularities such as the topological discontinuities due to replication of vertices with multiple attributes (i.e. normals and texture coordinates). Figure 9.2 shows an example of simplification made with our geometry-based simplification algorithm.

9.1.1.2 Viewpoint-driven simplification

Unlike our geometry-based algorithm, which has a local error metric, our viewpoint-driven simplification is based on a global error metric. This means that the error metric can take into account changes produced along the entire mesh. In addition, the metric is a visual error metric which studies the whole mesh from a certain number of fixed points of view. This is useful to preserve the global appearance of the model.

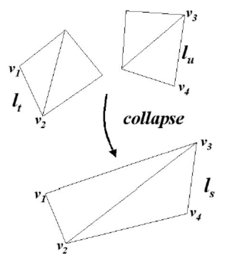


Figure 9.3: Simplification of two leaves to create a new one.

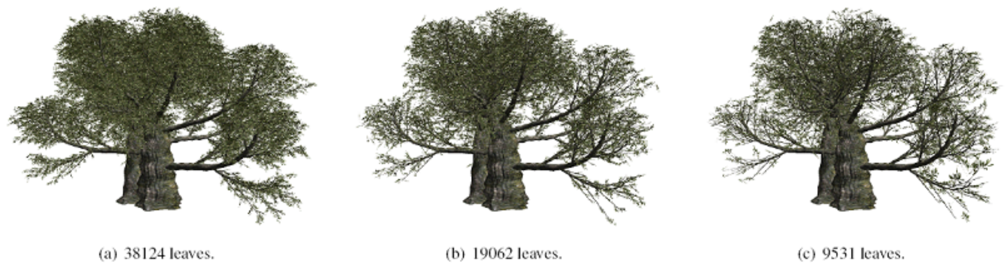


Figure 9.4: Foliage simplification.

9.1.2 Foliage simplification

We have developed an specific simplification algorithm for the foliage in which our LodTree multiresolution algorithm is based on. This algorithm uses on the leaf-collapse operation as its atomic operation. Using this technique, a certain metric decides to collapse two different leaves into a single one which preserves the area and overall shape of the initial two leaves. Figure 9.3 shows a graphical representation of the leaf collapse operation. Figure 9.4 shows an example of foliage simplificacion using our algorithm.

9.2 Geometry::GeometryBasedSimplifier Class Reference

Implementation of a simplification algorithm based on geometry information.

`#include <GeoMeshSimplifier.h>`

Public Member Functions

- **GeometryBasedSimplifier** (const **Mesh** ∗geoMesh, TIPOFUNC upb=0)

  Class constructor. Receives as a parameter the mesh to be simplified.

- ∼**GeometryBasedSimplifier** (void)

  Class destructor.

- void **Simplify** (Geometry::Real)

  Starts the simplification process.

- void **Simplify** (Geometry::uint32)

  Starts the simplification process.

- **Mesh** ∗ **GetMesh** (void)

  Returns the simplified mesh.

- MeshSimplificationSequence ∗ **GetSimplificationSequence** ()

  Returns the simplification sequence for general meshes.

9.2.1 Detailed Description

Implementation of a simplification algorithm based on geometry information.

This class implements a simplification algorithm based on a classic geometry evaluation technique.

It receives in the constructor the mesh to be simplified. The resulting simplified mesh can be obtained using the **GetMesh()** (p. 52) inherited method.

9.2.2 Constructor & Destructor Documentation

9.2.2.1 Geometry::GeometryBasedSimplifier::GeometryBasedSimplifier (const Mesh ∗ *geoMesh*, TIPOFUNC *upb* = 0)

Class constructor. Receives as a parameter the mesh to be simplified. The second is an optional parameter used to specify a 'status' function which is called internally by the simplifier to periodically indicate the completion percentage to the callee.

9.2.3 Member Function Documentation

9.2.3.1 void Geometry::GeometryBasedSimplifier::Simplify (Geometry::Real) `[virtual]`

Starts the simplification process.

Receives as a parameter the LOD factor in a range of [0,1]. Implements the Simplifier::Simplify method to perform a geometry based simplification.

Implements **Geometry::MeshSimplifier** (p. **??**).

9.2.3.2 void Geometry::GeometryBasedSimplifier::Simplify (Geometry::uint32) `[virtual]`

Starts the simplification process.

Receives as a parameter the number of vertices of the resulting mesh. Implements the Simplifier::Simplify method to perform an image based simplification.

Implements **Geometry::MeshSimplifier** (p. **??**).

9.3 Geometry::ViewPointDrivenSimplifier Class Reference

Implementation of a simplification algorithm based on visual information.

`#include <GeoMeshSimplifier.h>`

Inheritance diagram for Geometry::ViewPointDrivenSimplifier::

Public Member Functions

- **ViewPointDrivenSimplifier** (const **Geometry::Mesh** *geoMesh, Geometry::TIPOFUNC upb=0)

  Class constructor. Will call Simplifier class constructor.

- ~**ViewPointDrivenSimplifier** (void)

  Class destructor.

- void **Simplify** (Geometry::Real)

  Starts the simplification process.

- void **Simplify** (Geometry::uint32)

  Starts the simplification process.

- **Mesh** ∗ **GetMesh** (void)

    Returns the simplified mesh.

- MeshSimplificationSequence ∗ **GetSimplificationSequence** ()

    Returns the simplification sequence for general meshes.

9.3.1 Detailed Description

Implementation of a simplification algorithm based on visual information.

This class implements a simplification algorithm based on a brand-new viewpoint-driven evaluation technique.

It receives in the constructor the mesh to be simplified. The resulting simplified mesh can be obtained using the **GetMesh()** (p. 52) inherited method.

9.3.2 Member Function Documentation

9.3.2.1 void Geometry::ViewPointDrivenSimplifier::Simplify (Geometry::Real) `[virtual]`

Starts the simplification process.

Receives as a parameter the LOD factor in a range of [0,1]. Implements the Simplifier::Simplify method to perform a viewpoint-driven simplification.

Implements **Geometry::MeshSimplifier** (p. ??).

9.3.2.2 void Geometry::ViewPointDrivenSimplifier::Simplify (Geometry::uint32) `[virtual]`

Starts the simplification process.

Receives as a parameter the number of vertices of the resulting mesh. Implements the Simplifier::Simplify method to perform a viewpoint-driven simplification.

Implements **Geometry::MeshSimplifier** (p. ??).

9.4 Geometry::TreeSimplifier Class Reference

Tree Simplifier interface.

`#include <GeoTreeSimplifier.h>`

Public Member Functions

- **TreeSimplifier** (const **Geometry::Mesh** ∗, Geometry::TIPOFUNC upb=0)

  Class constructor. Receives as a parameter the mesh to be simplified.
  .

- ∼**TreeSimplifier** (void)

  Class destructor.

- void **Simplify** (Geometry::Real, Geometry::Index)

  Starts the foliage simplification process.

- **Mesh** ∗ **GetMesh** ()

  Returns the simplified mesh.

- **TreeSimplificationSequence** ∗ **GetSimplificationSequence** ()

  Returns the simplification sequence for leaves.

9.4.1 Detailed Description

Tree Simplifier interface.

This module is used by LODTree to simplify 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 objectŠs 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 resulting leaf for that contraction.

Receives as input a pointer to the **Geometry::Mesh** (p. 56) object containing the tree to be simplified, and outputs:

1. The simplified mesh, contained in a **Geometry::Mesh** (p. 56) object.
2. Simplification sequence, represented by a **Geometry::TreeSimplificationSequence** (p. ??) object.

9.4.2 Constructor & Destructor Documentation

9.4.2.1 Geometry::TreeSimplifier::TreeSimplifier (const Geometry::Mesh *, Geometry::TIPOFUNC *upb* = 0)

Class constructor. Receives as a parameter the mesh to be simplified.

.

The second and optional parameter is used to specify a 'status' function which is called internally by the simplifier periodically to indicate the completion percentage to the callee.

9.4.3 Member Function Documentation

9.4.3.1 void Geometry::TreeSimplifier::Simplify (Geometry::Real, Geometry::Index)

Starts the foliage simplification process.

Receives as a parameter the LOD factor in a range of [0,1]. The second parameter represents an integer pointing to the submesh containing the foliage data. Implements the Simplifier::Simplify method to perform an geometry based simplification.

Chapter 10

Helper classes reference

10.1 Introduction

This chapter is intended to be a reference guide for those classes that are used by the main modules described in previous chapters. Not all the classes of the geometry modules are detailed here. Just those classes which are referred by the main classes are included in this section.

10.2 Geometry::Mesh Class Reference

Mesh (p. 56) class interface.

`#include <GeoMesh.h>`

Inherits **Geometry::Serializable**.

Public Member Functions

- **Mesh** ()

  Constructor.

- ~**Mesh** ()

  Destructor.

- **Mesh** (const **Mesh** &)

  Copy constructor.

- **Mesh** & **operator=** (const **Mesh** &)

  Assignment operator.

- void **Load** (**Serializer** &s)

  Loads data from a ***Serializer*** *(*p. **??***).*

- void **Save** (**Serializer** &s)

  Saves data to a ***Serializer*** *(*p. **??***).*

Public Attributes

- **VertexBuffer** ∗ **mVertexBuffer**

  Shared ***VertexBuffer*** *(*p. *60).*

- **SubMesh** ∗ **mSubMesh**

  Array of subMeshes.

- size_t **mSubMeshCount**

  Total count of subMeshes.

- MeshBounds **mMeshBounds**

  Mesh *(*p. *56) bounds.*

10.2.1 Detailed Description

Mesh (p. 56) class interface.

10.3 Geometry::SubMesh Class Reference

SubMesh (p. 58) interface.

`#include <GeoSubMesh.h>`

Inheritance diagram for Geometry::SubMesh::

Public Member Functions

- **SubMesh** ()

  Default constructor.

- ~**SubMesh** ()

  Default destructor.

- void **Load** (**Serializer** &s)

  *Loads data from a **Serializer** (p. ??).*

- void **Save** (**Serializer** &s)

  *Saves data to a **Serializer** (p. ??).*

Public Attributes

- **VertexBuffer** ∗ **mVertexBuffer**
- bool **mSharedVertexBuffer**

  *true if **VertexBuffer** (p. 60) it's shared with **Mesh** (p. 56) and other SubMeshes*

- Index ∗ **mIndex**

  Array of Index.

- size_t **mIndexCount**

  Index count.

- Index ∗∗ **mStrip**

  Array of pointers to mIndex that represents each strip.

- size_t **mStripCount**

  number of Strips

- MeshType **mType**

Type of mesh.

- char **mMaterialName** [255]

  Material name.

10.3.1 Detailed Description

SubMesh (p. 58) interface.

SubMesh (p. 58) is part of a **Mesh** (p. 56), and stores vertex and index geometric information.

10.3.2 Member Data Documentation

10.3.2.1 VertexBuffer∗ Geometry::SubMesh::mVertexBuffer

VertexBuffer (p. 60) used to store vertex Data. Is a reference to a shared VertexData if mSharedVertexBuffer == true, and must be not deallocated in that case.

10.4 Geometry::VertexBuffer Class Reference

VertexBuffer (p. 60) interface Class.

`#include <GeoVertexBuffer.h>`

Inherits **Geometry::Serializable**.

Inheritance diagram for Geometry::VertexBuffer:Collaboration diagram for Geometry::VertexBuffer:

Public Member Functions

- **VertexBuffer** ()

  Default Constructor.

- ~**VertexBuffer** ()

  Default destructor, releases allocated memory.

- **VertexBuffer** ∗ **Clone** () const

  *Returns a new **VertexBuffer** (p. 60) with the same data.*

- void **Load** (**Serializer** &s)

  *Fills this **VertexBuffer** (p. 60) from a **Serializer** (p. ??).*

- void **Save** (**Serializer** &s)

  Stores data.

Public Attributes

- unsigned int **mVertexInfo**

  Type of info stored by vertex.

- size_t **mVertexCount**

  Number of vertices.

- Vector3 ∗ **mPosition**

  Position array of each Vertex, only valid if (vertexInfo & VERTEX_-POSITON) == true.

- Vector3 ∗ **mNormal**

  Normal array of each Vertex, only valid if (vertexInfo & VERTEX_-NORMAL) == true.

- Vector2 ∗ **mTexCoords**

  Texture Coordinates array of each Vertex, only valid if (vertexInfo & VERTEX_ TEXCOORDS) == true.

10.4.1 Detailed Description

VertexBuffer (p. 60) interface Class.

This Structure holds the vertex information used by Meshes and SubMesehs.