Master Thesis

Vladimír Kašpar

Zjednodušování polygonálních modelů ve VRML/X3D

Department of Software Engineering
Advisor: Doc. Ing. Jiří Žára, CSc., Czech Technical University,
Faculty of Electrical Engineering
Study Program: Computer Graphics

PRAGUE 2005
I would like to thank:

• to my advisor Jiří Žára for his support and comments, especially on the text of the thesis.

• to Emil Tcheparov, who helped me with testing software and was always ready to assist with the technical issues.

• to eSZett, especially to Oliver Schwarz and Olaf Zupke for new experiences with virtual reality system.

I declare that I wrote this thesis by myself using only the sources cited. I agree with lending the thesis, but distribution of the source code and developed software is permitted.

Prague, 13th December 2005

Vladimír Kašpar
List of Figures

3.1 Feature edge ............................................................ 13
3.2 Vertex classification .................................................. 14
3.3 Average plane .......................................................... 15
3.4 Vertex clustering ....................................................... 15
3.5 Examples of non-manifold meshes ................................. 16
4.1 VRML tree ............................................................... 18
4.2 Octree ................................................................. 21
4.3 Getting coordinates ................................................... 22
4.4 Error area heuristic .................................................. 23
4.5 Error volume heuristic ............................................... 24
4.6 Compute the range values .......................................... 26
4.7 Inheritance of class “Node” ......................................... 31
4.8 Inheritance of class “Data” ........................................... 33
4.9 Inheritance of class “DataSet” .................................... 34
4.10 Parser ................................................................. 35
5.1 “Stanford bunny” and its bounding box ......................... 43
5.2 Point-to-surface distance ............................................. 45
5.3 The one-sided distances .............................................. 46
5.4 Simplification of the “t-junction” mesh ......................... 51
5.5 Results from comparison 1 ......................................... 52
5.6 Results from comparison 2 ......................................... 52
5.7 Geometric deviation - Rational Reducer ......................... 54
5.8 Geometric deviation - ČVUT - tool ............................... 56
5.9 Geometric deviation - LoDCreator ............................... 58
5.10 All - geometric deviation ........................................... 60
5.11 All - normal deviation ............................................... 61
5.12 Execution speed ..................................................... 62
5.13 Bunny (99% simplification) ....................................... 63
5.14 Bunny - geometric deviation ..................................... 63
5.15 Bunny - normal deviation .......................................... 63
5.16 Color scale .......................................................... 63
6.1 Pipes, original object ................................................. 65
6.2 Pipes, errorrange=10. ................................................. 66
6.3 Cylinders ............................................................... 67

A.1 Menu system. ......................................................... 68
A.2 File menu. ............................................................. 69
A.3 Action menu. ......................................................... 69
A.4 Window menu. .................................................... 70
A.5 Help menu. ............................................................ 71
A.6 Main window. ....................................................... 71
A.7 Status bar. ............................................................ 72
A.8 Tree dialog. ........................................................... 73
A.9 Statistic dialog. ....................................................... 74
A.10 LoD dialog. ........................................................... 75
A.11 “Vertex clustering” dialog. ..................................... 76
A.12 “Vertex decimation” dialog. ................................. 76
A.13 Messages window. ................................................. 77
A.14 Edit dialog. ........................................................... 78
A.15 Preferences dialog. ................................................. 79
List of Tables

5.1 Geometric deviation - Rational Reducer. .......................... 53
5.2 Normal deviation - Rational Reducer. .............................. 53
5.3 Geometric deviation - ČVUT - tool. ................................. 55
5.4 Normal deviation - ČVUT - tool. ................................. 55
5.5 Geometric deviation - LoDCreator (vertex clustering). ......... 57
5.6 Normal deviation - LoDCreator (vertex clustering). .......... 57
5.7 Geometric deviation - LoDCreator (vertex decimation). ..... 59
5.8 Normal deviation - LoDCreator (vertex decimation). ......... 59
5.9 Execution speed. ................................................. 61

B.1 Min diff=20. .................................................. 82
B.2 Min diff=50. .................................................. 82
B.3 Min diff=90. .................................................. 83
B.4 Min diff=100. .................................................. 83
B.5 The situation with error range=0.1%, 2%, 10%, 90% and 100%. .. 83
# Contents

1 Goals of the diploma thesis .................................................. 7

2 VRML1.0 .............................................................................. 9
   2.1 Language Basics ........................................................... 9
   2.2 LOD ............................................................................ 10

3 Theory .................................................................................. 12
   3.1 Mesh simplification ....................................................... 12
      3.1.1 Overview ............................................................. 12
      3.1.2 Vertex decimation ............................................... 13
      3.1.3 Vertex clustering ............................................... 14
   3.2 Level of detail .............................................................. 15
      3.2.1 Discrete LOD ....................................................... 16
      3.2.2 Continuous LOD ............................................... 16
      3.2.3 View-dependent LOD ........................................ 17

4 LoD Creator - Implementation ................................................. 18
   4.1 Algorithms and data structures .................................... 20
      4.1.1 Vertex clustering ............................................... 20
      4.1.2 Vertex decimation ............................................... 24
      4.1.3 Triangulation ..................................................... 25
      4.1.4 Calculating the Range of a LOD ......................... 26
      4.1.5 Joining VRML nodes .......................................... 27
   4.2 Details of object implementation .................................. 30
      4.2.1 Classes derived from superclass “Node” ................. 30
      4.2.2 Classes derived from superclass “Data” ................. 33
      4.2.3 Classes derived from superclass “Dataset” ............ 33
      4.2.4 Other classes ..................................................... 34
   4.3 Parser ........................................................................... 35
      4.3.1 Flex .................................................................... 36
      4.3.2 Yacc-Bison ......................................................... 37
   4.4 Problems ........................................................................ 37
Název práce: Zjednodušování polygonálních modelů ve VRML/X3D

Autor: Vladimír Kašpar

Katedra: Katedra softwarového inženýrství


e-mail vedoucího: zara@fel.cvut.cz

Abstrakt: Potřeba interaktivního zobrazování trojrozměrných dat a virtuálních světů se velmi často objevuje v oblasti počítačové grafiky a inženýrství. V mnoha případech je však rozsáhlost scény omezujícím faktorem pro plynulé generování scény. Jedna z možností, jak zajistit potřebnou rychlost a plynulost generování obrazu, je technika pro dynamické řízení zobrazovaných detailů, nazývaná “stupeň detailu”. Většina programů určených pro zpracování grafických dat však neumí generovat scény se stupněm detailů pro specifické potřeby VRML1.0, který se často používá právě pro interaktivní vizualizaci inženýrských dat. Cílem této práce je navrhnout a implementovat software, který bude schopen zpracovat a generovat stupně detailů pro zadaná grafická data s ohledem na specifické vlastnosti VRML1.0. Výsledkem této diplomové práce je program LODCreator, založený na dvou typech algoritmu pro zjednodušování polygonálních sítí - shlukoření vrcholů a decimace vrcholů.

Klíčová slova: VRML1.0, zjednodušování polygonálních modelů, stupeň detailu

Title: Simplification of polygonal models in VRML/X3D

Author: Vladimír Kašpar

Department: Department of Software Engineering


Supervisor’s e-mail address: zara@fel.cvut.cz

Abstract: Interactive visualization of large three-dimensional objects and virtual worlds is an important topic in the field of computer graphics and visualization of engineering structures. In many cases, the number of primitives in these worlds overwhelms the rendering performance of current graphics systems. One solution for accelerating the displaying of these environments is creation and application method called levels of detail (LOD). However, most algorithms that compute levels of detail do not deal with the special requirements of input data in VRML1.0 format. This thesis works out methodology and develop a tool for generation level of detail for VRML1.0 called LODCreator, based on two simplification algorithms - vertex clustering and vertex decimation, that robustly computes simplifications for objects in VRML1.0.

Keywords: VRML1.0, simplification of polygonal models, level of detail
Chapter 1

Goals of the diploma thesis

The initial idea of examining the geckos with the computer support came. The main goal of this diploma thesis is to find a methodology and develop a tool for automatical generation level of detail for VRML1.0. The next goal is to check the functionality of this tool and compare result with other software for mesh simplification.

During work on the diploma thesis has begun cooperation with company named eSZett dealing with virtual reality technology and the thesis was adjusted and extended to topic "LOD creation" process, only for VRML1.0 (thus not X3D). Communication language with eSZett was English.

Suggestions form eSZett:

Investigate a tool for level of detail generation with the focus on:

- Reorganize VRML1.0 files.
- Define the LOD depends on a predefined distance (like 15 m, 30 m, etc.).
- Define the LOD for each design file separate.
- Produce VRML files compatible with eSZett system.

Note: Contract between the two partners over secrecy, application and protection of the work and ideas. Working language is English.

The basic information about VRML1.0 provides next section - Chapter 2. For deeper insight it is recommend to read the VRML1.0 specification [11]. The overview and theory of two mesh simplification algorithms used in thesis - vertex clustering and vertex decimation - describes Chapter 3. The details of
implementation of the mentioned methods and complete description of functionality, algorithms and data structures of development tool is in Chapter 4. The complete test of development tool and comparison of its results to other software can be found in Chapter 5. The Chapter 6 contains summary of work and also describes some problems, which occur during work and sketch possible solution. Appendices are given in Appendix - the complete reference manual of the LODCreator and tutorial.
Chapter 2

VRML1.0

The Virtual Reality Modeling Language (VRML) is a proposed design based on the Silicon Graphics Open Inventor ASCII file format. Originally called the Inventor VRML, this format has evolved into VRML1.0 format.

VRML is also called a “Markup Language” for reasons that it is used in a fashion similar to HTML (HyperText Markup Language), but for rendering 3D graphics rather than text. VRML, however, is in no way derived from HTML. The first version of VRML allows for the creation of virtual worlds with limited interactive behavior (static world).

2.1 Language Basics

A VRML scene is a hierarchical structure of shapes, transformations, cameras, and lights. Every object-property is a part of this structure and is named “node”.

VRML1.0 uses a Cartesian, right-handed, 3-dimensional coordinate system. The standard unit for lengths and distances specified is meters. The standard unit for angles is radians.

VRML worlds may contain an arbitrary number of local (or “object-space”) coordinate systems, defined by modeling transformations using Translate, Rotate, Scale, Transform, and MatrixTransform nodes.

VRML defines several different classes of nodes. Most of the nodes can be classified into these categories:

**Group nodes**: Group, Separator, Switch, TransformSeparator, WWWAnchor.

**Lighting nodes**: DirectionalLight, PointLight, SpotLight.

**Camera nodes**: OrthographicsCamera, PerspectiveCamera.

**Property nodes**: Coordinate3, FontStyle, Info, LOD, Material, MaterialBinding, Normal, NormalBinding, Texture2, Texture2Transform, TextureCoordinate2, ShapeHints.
2.2 LOD

**Shape nodes:** AsciiText, Cone, Cube, Cylinder, IndexedFaceSet, IndexedLineSet, PointSet, Sphere.

**Transformation nodes:** MatrixTransform, Rotation, Scale, Transform, Translation.

**WWWInline**

Nodes are arranged in hierarchical structures called **scene graphs** which defines an ordering for the nodes. The scene graph has a notion of *state* – nodes earlier in the world can affect nodes that appear later in the world. For example, a Rotation or Material node will affect the nodes after it in the world. A mechanism is defined to limit the effects of properties (separator nodes), allowing parts of the scene graph to be functionally isolated from other parts.

Nodes may contain zero or more fields. Each node type defines the type, name, and default value for each of its fields. The default value for the field is used if a value for the field is not specified in the VRML file. The order in which the fields of a node are read is not important; for example:

```
Cube {
    width 2
    height 4
    depth 6
}
```

and

```
Cube {
    height 4
    depth 6
    width 2
}
```

are equivalent.

More informations can be found on web site “VRML specification” [11].

2.2 **LOD**

Level of detail is used to allow applications to switch between various representations of objects. The LOD node contains a list of nodes, one for each LOD representation, where the highest detail level is given first. LOD must be used as a separator node. The specified center point of the LOD is transformed by the current transformation into world space, and the distance from the transformed
center to the world-space viewpoint is calculated. If the distance is less than the first value in the ranges array, then the first child of the LOD group is drawn. If between the first and second values in the ranges array, the second child is drawn, etc. If there are N-1 values in the ranges array (range1 to range_{n-1}), the LOD group should have N children - n levels of detail (level1 to level_n). Specifying too few children will result in the last child being used repeatedly for the lowest levels of detail; if too many children are specified, the extra children will be ignored. Each value in the ranges array should be less than the previous value, otherwise results are undefined.

FILE FORMAT/DEFAULTS

```
LOD {
    range [ ] # MFFloat
    center 0 0 0 # SFVec3f
}
```

For example, given a range array of [50, 100] (in meters) and three levels of detail, the highest LOD is used when the object’s distance is less than 50 m away, the medium LOD is used between 50 and 100 m, and the lowest LOD is used for all distances over 100 m. Often the lowest detail model is made to be an empty node to allow the object to be invisible beyond a certain distance.

\[
level(d) = \begin{cases} 
    level_1 & \text{where } d < range_1 \\
    level_i & \text{where } range_{i-1} \leq d < range_i \quad \text{for } 1 < i \leq n - 1 \\
    level_n & \text{where } d \geq range_{n-1} 
\end{cases} \quad (2.1)
\]
Chapter 3

Theory

3.1 Mesh simplification

Mesh simplification, sometimes referred to as decimation, is probably the most widely used method of visual LOD implementation. The goal of polygon simplification is to make a polygonal object smaller while minimizing its loss of visual fidelity.

3.1.1 Overview

An assortment of simplification algorithms have been introduced in recent years. Most of those which are applicable to polygonal meshes can be classified into 3 categories.

Vertex clustering methods [4] spatially partition vertices and unify vertices within the same cluster. They are generally very fast and work on arbitrary collections of triangles, but can often produce relatively poor results for some kind of meshes.

Vertex decimation algorithms [1] select unimportant vertices (based on local shape heuristics), remove them, and retriangulate the resulting holes. These methods tend to produce fair results and are reasonably efficient.

Iterative edge contraction [8] has been widely used. Edges are ranked according to their cost, which is typically the amount of error introduced into the model as a result of contracting the edge. At each iteration the lowest-cost edge is contracted and the costs of neighboring edges are updated. The essential difference among the various contraction-based methods is the error metric used for ranking edges.

For this work the first two algorithms were selected: vertex clustering was chosen for its effectiveness and robustness and vertex decimation for effectiveness and reasonable quality output.
3.1.2 Vertex decimation

One of the more widely used algorithms for mesh simplification is vertex decimation, it is an iterative simplification algorithm originally designed by Schroeder [1]. In each step of the decimation process, a vertex is selected for removal, all the faces adjacent to that vertex are removed from the model and the resulting hole is retriangulated. Since the retriangulation usually requires a projection of the local surfaces onto a plane, these algorithms are generally limited to manifold surfaces.

Vertices considered for removal are classified into five categories, illustrated in Figure 3.2: simple, boundary, interior edge, corner, and nonmanifold. A simple vertex is locally manifold, surrounded by a single complete ring of triangles and each of triangle shares a single edge with the vertex. Boundary vertices differ in that the set of triangles does not form a complete ring. The algorithm classifies vertices that do not fit any category as nonmanifold, and does not consider them for removal. Simple vertices can be further classified according to the geometry of the surrounding triangles. Two triangles can form a feature edge, which Schroeder [1] define by the angle between the triangles (Figure 3.1). A simple vertex with less than three feature edges is classified as an interior edge vertex; vertices with three or more feature edges are considered corners.

![Figure 3.1: Feature edge exists if the angle between the surface normals of two adjacent triangles is greater than a user-specified specified “feature angle”.

The distance between the vertex from the average plane (Figure 3.3) of its surrounding triangles is used as the decimation criterion. If this distance is less than some user-specified threshold, the vertex is removed. Note that a threshold of zero will remove only vertices in planar regions. Boundary and interior edge vertices are tested against an average line formed from the two vertices on the boundary or feature edge. The distance of the candidate vertex from this line is tested against the same threshold, and the vertex is deleted if the distance is under threshold. The corner vertices are assumed to represent important
3.1 MESH SIMPLIFICATION

Figure 3.2: Vertices considered for decimation are classified as simple, nonmanifold, boundary, interior edge, or corner.

features and are generally not deleted. However, if the mesh is known to be noisy, with many extraneous feature edges, the user may choose to evaluate corner and interior edge vertices against the average plane criterion used for simple vertices.

For simple vertices, an average plane is computed from the surrounding triangles. The normal of the average plane $\vec{n}$ is an average of the triangle normals weighted by the triangle areas:

$$\vec{N} = \frac{\sum_{i=1}^{m} \vec{n}_i A_i}{\sum_{i=1}^{m} A_i'}, \vec{n} = \frac{\vec{N}}{|\vec{N}|},$$

(3.1)

where $\vec{n}_i$ is a triangle normal and $A_i$ is a triangle area.

A point on the average plane is computed by averaging the centroid of the surrounding triangles, also weighted by area:

$$\vec{x} = \frac{\sum_{i=1}^{m} \vec{x}_i A_i}{\sum_{i=1}^{m} A_i'},$$

(3.2)

where $\vec{x}_i$ is a center of triangle.

A resultant distance to average plane criterion is computed:

$$d = |\vec{n} \bullet (\vec{v} - \vec{x})|,$$

(3.3)

where $\vec{v}$ is a vertex of considered.

3.1.3 Vertex clustering

The key idea is to cluster multiple vertices of the polygonal object that are close in object space into one, and remove all triangles that degenerate or collapse in the process (Figure 3.4). The problem here is that exact control over local detail is not so easily possible, but such an algorithm can robustly deal with any type of input data, and produce arbitrarily high compression. Vertex clustering can
3.2 Level of detail

Levels of details (LODs) are approximations of an object with fewer geometric primitives. The fundamental concept of LOD is simple: when rendering, use a less detailed representation for small, distant, or unimportant portions of the scene. This less detailed representation typically consists of a selection of several versions of objects in the scene, each version is less detailed and faster to render than the one before.

A number of techniques have been developed for various LOD implementations. These include various automatic geometric simplification techniques, billboarded images, and specialized hand edited models with custom distance algorithms.

In all cases, the performance of the scene is dependent upon the algorithm used to determine the LOD to render and the algorithm used to create the LODs.
3.2 LEVEL OF DETAIL

There are a large number of algorithms. Some of the algorithms are designed for compact models, while other work well for extended or large scale objects. There are three basic frameworks for managing level of detail:

### 3.2.1 Discrete LOD

Discrete LOD is implemented by creating a minimal, finite number of varying LODs for an object. The object is then rendered in one of the LODs with respect to some function of the distance from the viewpoint. The object appears as necessary between LODs depending on what it’s distance is in the scene.

Discrete LOD has many advantages. Decoupling simplification and rendering makes this the simplest model to program: the simplification algorithm can take as long as necessary to generate LODs and the run-time rendering algorithm simply needs to choose which LOD to render for each object. Furthermore, modern graphics hardware dedicate to the multiple model versions created by static level of detail, because individual LODs can be compiled during preprocessing to an optimal rendering format. Discrete LOD is used in VRML1.0 (see Chapter 2).

### 3.2.2 Continuous LOD

Continuous LOD changes from the traditional discrete approach. Rather than creating individual LODs during the preprocessing stage, the simplification system creates a data structure encoding a continuous spectrum of detail. The desired level of detail is then extracted from this structure at run-time. A major advantage of this approach is better granularity: since the level of detail for each object is specified exactly rather than selected from a few pre-created options, no more polygons than necessary are used. This frees up more polygons for rendering other objects, which in turn use only as many polygons as needed for

![Figure 3.5: Examples of non-manifold meshes: (a) An edge shared by three triangles. (b) A vertex shared by two otherwise unconnected sets of triangles. (c) An edge of one triangle is spanned by edges from two other triangles called “t-junction”.

There are a large number of algorithms. Some of the algorithms are designed for compact models, while other work well for extended or large scale objects. There are three basic frameworks for managing level of detail:

### 3.2.1 Discrete LOD

Discrete LOD is implemented by creating a minimal, finite number of varying LODs for an object. The object is then rendered in one of the LODs with respect to some function of the distance from the viewpoint. The object appears as necessary between LODs depending on what it’s distance is in the scene.

Discrete LOD has many advantages. Decoupling simplification and rendering makes this the simplest model to program: the simplification algorithm can take as long as necessary to generate LODs and the run-time rendering algorithm simply needs to choose which LOD to render for each object. Furthermore, modern graphics hardware dedicate to the multiple model versions created by static level of detail, because individual LODs can be compiled during preprocessing to an optimal rendering format. Discrete LOD is used in VRML1.0 (see Chapter 2).

### 3.2.2 Continuous LOD

Continuous LOD changes from the traditional discrete approach. Rather than creating individual LODs during the preprocessing stage, the simplification system creates a data structure encoding a continuous spectrum of detail. The desired level of detail is then extracted from this structure at run-time. A major advantage of this approach is better granularity: since the level of detail for each object is specified exactly rather than selected from a few pre-created options, no more polygons than necessary are used. This frees up more polygons for rendering other objects, which in turn use only as many polygons as needed for
the desired level of detail, freeing up more polygons for other objects, and so on. Continuous LOD also supports streaming of polygonal models, in which a simple base model is followed by a stream of refinements to be integrated dynamically. When large models must be loaded from disk or over a network, continuous LOD provides progressive rendering and interruptible loading - often very useful properties.

3.2.3 View-dependent LOD

View-dependent LOD (also called progressive LOD) extends continuous LOD, using view-dependent simplification criteria to dynamically select the most appropriate level of detail for the current view. A complex models representing physically large objects, such as terrains, often cannot be adequately simplified without view-dependent techniques. Creating discrete LODs does not help: the viewpoint is typically quite close to part of the terrain and distant from other parts, so a high level of detail will provide suitable fidelity at unacceptable frame rates, while a low level of detail will provide good frame rate but worse fidelity.
Chapter 4

LoD Creator - Implementation

This chapter contains the specifics of the implementation related to methods that were mentioned in Chapter 3. It also describes all the non-trivial algorithms used in the implementation of the LODcreator. This tool is based on Lodestar [14] algorithm.

As shown in Figure 4.1, the tool replaces each shape node (VRML geometry) by a LOD node and inserts in this node a new shape nodes for each level of detail.

The LODCreator cannot accept some bindings (PER_FACE, PER_PART, PER_VERTEX), these bindings are 1:1 relations and they cannot be maintained if different LODs use the same materials or normals. For that reason these bindings will be transformed into an appropriate indexed binding and indices will be synthesized. In this case it is necessary to write out an additionally MaterialBinding, respectively NormalBinding node.

Because the tool splits non convex faces into triangles, the face type can be set to CONVEX. If the face type is not convex in the actual state, the program generates an additional ShapeHints node for setting the type to convex.

Figure 4.1: A single VRML geometry is converted into a subtree with a single LOD node.
Outline of the algorithm

- Parse the input file with lex and yacc (see Section 4.3).
- Traverse the tree for setting up the dependencies between the nodes. This includes information such as: e.g. which Coordinate3, Material, Normal node uses a IndexedFaceSet node, which bindings are currently used, the current transformation matrix.
- If the camera height angle is not explicit set by the option “camera height angle” a camera is searched by traversing the tree. The program takes the first perspective camera found in the input file. If no camera was found, the default camera is taken (height angle = π/4).
- If option join is specified, the tree is traversed and it is tried to join nodes such as Coordinate3, Material, Normal, IndexedLineSet, IndexedFaceSet and other nodes. Detailed description is in Section “Joining VRML nodes” on page 27.
- Traverse the tree for:
  - Setup the specific data structures for coordinates, materials, normals and texture coordinates:
    * Materials, normals and texture coordinates will be inserted into binary trees. The insert-method removes automatically doublets.
    * Coordinates will be inserted into an octree data structure (for vertex clustering algorithm) or into a linear list (for schroeder’s algorithm). The insert-method removes automatically doublets.
  - If vertex clustering algorithm is used, generate LODs for each IndexedFaceSet and IndexedLineSet::
    1. Reset the data structure with the coordinates which are used by this node to LOD0 and delete all references from this coordinates to faces (lines).
    2. Insert the faces (lines) into an data structure FaceSet (LineSet) for LOD0. The insert-method splits quadrangles into triangles, if they are not convex or are too much distorted in 3D (by options “quad split angle”, see Section “Splitting quadrangles into triangles” on page 26). Faces with more than four vertices will always split into triangles using a triangulator.
    3. Generate the next LOD of the coordinates. The following algorithm is applied to all leaf-nodes which depths is the depth of the octree: Merge the leaf-nodes with the greatest depth into theirs parent node - this is made by selecting a representative of the leaf nodes
(with heuristics *error area, error volume, weighted mean*, see Section 4.1.1). And set to this parent node value for actual LOD. The depth of the resulting tree is decreased by one. If the difference in the count of the coordinates of the new LOD and the old LOD is less then $mindiff \times$ (count of coordinates old LOD) then the algorithm is applied again to the tree.

4. Create a new *FaceSet (LineSet)* for the next LOD and fill this *FaceSet (LineSet)* with the faces (lines) of the next LOD. The faces (lines) of the current *FaceSet (LineSet)* are mapped onto the representatives of the coordinates in the next LOD (see Section “Getting the representative of coordinates” on page 22). Doublets and collapsed faces (lines) will be discarded.

5. return to step 3 until the depth of the octree is 1.

– If Schroeder’s algorithm is used, generate LODs for each *IndexedFaceSet*:

1. Reset the data structure (linear list) with the coordinates which are used by this node and delete all references from this coordinates to faces (lines).
2. Insert the faces into an data structure *FaceSet for LOD_0*.
3. Classify coordinates by Schroeder’s method (see Section 3.1.2).
4. Sort coordinates in linear list (by quicksort).
5. Eliminate the first coordinate form list.
6. Triangulate the result hole and add faces in to the list of faces.
7. Actualize the list of coordinates.
8. Return to step 5, until some termination condition is met (defined by user), or linear list is empty.

– Generate LODs for each *Cone, Cube, Cylinder* and *Sphere*.

- Calculate the ranges of the LODs (detailed description in the section ...).
- Print out the nodes.

### 4.1 Algorithms and data structures

#### 4.1.1 Vertex clustering

The first of algorithm used by LODCreator is vertex clustering based on octree quantization [6].
Octree

Octree quantization was originally developed to select the entries for a color lookup table that optimally represent a given image. Instead of color pixels, the vertices of the model into an octree are entered. Nodes in this tree represent cubic fractions of space. An intermediate node divides his cube into eight subcubes with the same size by halving the sides. This means, the (max 8) successors of the intermediate node are representing subcubes of the cube of this node (Figure 4.2). A cube defined by a leaf node contains exactly one coordinate. The insert algorithm would create automatically intermediate nodes if two coordinates fell into the same cube.

![Inserting Coordinates into an Octree](image)

**Figure 4.2:** Octree.

**Inserting Coordinates into an Octree**

The insert algorithm is realized by a recursive function. Beginning with the root of the tree the function walks through the tree. While the current node is an intermediate node, the function calculates the subcubes and finds out in which subcube the new coordinate lies inside. This determines the right successor for searching again. This is made until a leaf node is reached or the pointer to the successor is NULL.

Then there are three cases:

1. The current node is nil pointer, so the corresponding subvolume is empty. This means the algorithm has found a place to insert. The successor pointer will be set to the new node.

2. A leaf node is reached contains a coordinate equal to the coordinate to which it is inserted in. This means, an equal coordinate already exists. The new coordinate will not be inserted into the tree.
Note: this mechanism automatically sorts out doublets in the vertices, which are a major defect of many VRML models.

3. Two coordinates would fall into the same cube this means the octree is not deep enough. So the octree must be subdivided in that location. The leaf node will be replaced by a new intermediate node. A new intermediate node is created and the old leaf node and a new node containing the new vertex are inserted as children of the new intermediate node.

Getting the representative of coordinates

For generating the next LOD, leaf nodes of an octree will be combined into ancestors. The type of the ancestor will be changed from intermediate to (quasi) leaf. This will be done by changing the LOD value in the node from -1 to the number of the next LOD. Further the data of these parent nodes will be set to the data of the representatives selected out of the leaf nodes. So algorithms are necessary for selecting a representative out of a cluster of coordinates (Figure 4.3).

The remaining problem is which strategy to use select the representative vertex from vertices in the cluster. There are three different heuristics with user defined weight:

Error area algorithm

This heuristic calculates for each coordinate in the cluster the change of the area of all involved faces that will occur if this coordinate is selected as represent.
The involved faces are that faces which use at least one coordinate of the cluster. Each coordinate has a list of faces which uses this coordinate. So the involved faces can be determined.

Figure 4.4 shows what will happen to the involved faces of a cluster of three points if A, B, C or D is selected as the representative. If the error area algorithm is used, D will be chosen, because in this case the change of the area of the involved faces is the smallest. This heuristic is suitable for preserving the contours of an object.

**Error volume algorithm**

This heuristic selects one coordinate of the cluster and calculates the error factor for this coordinate as the sum of volumes. For every involved triangle it is constructed the tetrahedron from the three original vertices and the potential representative (Figure 4.5). The volume of such a tetrahedron is zero if one of the vertices is elected as the representative. The summed volume of all such tetrahedrons is taken as the error volume, and the vertex with the smallest error volume is selected. A disadvantage of this approach is that all volumes are zero if the vertices lie in a plane, so it is useful only in combination with another heuristic.

**Weighted mean algorithm**

This heuristic is an attempt to find the vertex that “best” represents the other vertices: An average vertex is synthesized from the cluster as a weighted mean, where the weights are the vertex counts of the nodes in the cluster. The vertex that is closest to the mean is chosen. The error factors of the points in the cluster are simply the euclidean distances to the mean value calculated above. Unfortunately, this does not take into account any surface properties, so results using this heuristic are not visually as appealing as the other two heuristics.
Figure 4.5: The error volume is computed from the tetrahedron with the original triangle ABC as a base and the chosen representative R as the top. R is the representative of the coordinate A in the next LOD - face ABC is mapped to RBC.

The LOD generation algorithm uses for IndexedLineSet nodes only this selection algorithm, because the other two works only with faces.

Note: The weights for these three algorithms can be changed with the options in “LoD dialog” (Section A.11).

Reconstruction the reduced triangle set

After the number of vertices has been reduced by the desired amount, the set of triangles associated with the reduced vertex set has to be reconstructed. For every triangle, its vertices are replaced by the representative chosen for that vertex. This process may produce triangles with identical vertices (doublets), for which only one instance is kept. Triangles may also collapse into lines, most of which are identical to the edge of another triangle and can be discarded. The remaining lines are usually important for the appearance of the model and are thus saved. Sometimes triangles collapse into points, which are removed from the model.

4.1.2 Vertex decimation

The data structure must contain at least two pieces of information: the geometry, or coordinates of each vertex, and the definition of each triangle in terms of its three vertices. In addition, because ordered lists of triangles surrounding
a vertex are frequently required, it is desirable to maintain a list of the triangles that use each vertex.

It was implemented a space efficient vertex - triangle hierarchical ring structure. This data structure contains hierarchical pointers from the triangles down to the vertices, and pointers from the vertices back up to the triangles using the vertex. Taken together these pointers form a ring relationship. Implementation uses three lists: a list of vertex coordinates (see Section 4.2.3 - class ListCoordSet), a list of triangle definitions, and another list of lists of triangles using each vertex (Section 4.2.4 - class Vertex).

4.1.3 Triangulation

The most level of detail algorithms including LODCreator can work only with triangles as an input. For triangulation is used the extended 2D triangulator based on Seidel’s algorithm [15]. This incremental randomized algorithm runs in expected $O(n \cdot \log(n))$ time on a polygon with $n$ vertices.

The extension converts the polygons from the third into the second dimension. First converts every vertex of a polygon into the second dimension ($xyz \rightarrow xy$). If the new 2D polygon is not correct (crossing lines, the polygon is a line), uses $xz$ and then $yz$. If there is no correct point pair, the polygon will not be triangulated. It is also necessary to check the clock wise order.

Overview of fast polygon triangulation based on Seidel’s Algorithm:

1. **Decompose the polygon into trapezoids** - this operation takes $O(n \cdot \log(n))$ expected time.

2. **Decompose the trapezoids into monotone polygons** \(^1\) - by drawing a diagonal between opposite-side vertices, forming corners of a trapezoid. This is a linear time operation.

3. **Triangulate the resulting monotone polygons** - monotone polygons can be triangulated in $O(n)$ time.

**Note:** the number of possible ways to triangulate polygon is bounded by the elements of the Catalan sequence [3]:

$$C(i) = \frac{1}{i+1} \cdot \binom{2i}{i} = \frac{1}{i+1} \cdot \frac{(2i)!}{i!(2i-i)!} = \frac{1}{i+1} \cdot \frac{(2i)!}{i!i!} = \frac{(2i)!}{(i+1)i!}$$

(4.1)

where $C(i)$ is the number of ways to fill a convex, planar hole with $i + 2$ sides.

\(^1\)If a polygon $P$ is monotone, some line $L$ exists for which no perpendicular will cross more than two edges of $P$. 
Splitting quadrangles into triangles

After a polygon was triangulated, as a side effect, all concave polygons (polygon is concave if at least one of its internal angles is greater than 180 degrees) are removed from the model, which allows the use of algorithms that are simpler, more robust and faster.

The exception are quadrilaterals, for which the error is often small and hence tolerable (i.e., non-visible). It is necessary, though, to check any quadrilaterals for validity after a modification of its vertices. Concave quadrilaterals or quadrilaterals which are distorted in space more than a user-specified threshold (measured as the maximum angle between the normals at the vertices) are split into two triangles.

### 4.1.4 Calculating the Range of a LOD

The selection of a LOD is realized by comparing ranges. A viewer switches to the next LOD if the distance of the object to the viewpoint is greater or equal to a specified value.

The maximum visible error is equal to the maximum distance a vertex can move in screen space due to a mesh simplification. The goal is to compute LOD switching ranges in such a way that this maximum visible error does not exceed a user-defined threshold, that is specified as a percentage of the screen height (Figure 4.6).

![Figure 4.6: Compute the range values.](image)

The viewer must switch to the next LOD if the maximum deviation $s$ projected into the screen is greater than error range specified as a fraction of the height of the screen.

So $s$ is calculated:
\[ s = scale \cdot size \cdot \frac{\sqrt{3}}{2^{\text{level}-1}}, \]

where \( size \) is the diagonal of the bounding box, \( level \) is level of details being computed and \( scale \) is the current scale factor from proceeding \( Scale \) or \( Transform \) nodes.

The height of the screen \( h \) is computed:

\[ h = 2 \cdot \tan \left( \frac{\alpha}{2} \right) \cdot f, \]

where \( \alpha \) is the \textit{camera height angle} and \( f \) is the focal length.

The maximum deferral \( s \) projected into the screen is called \( ps \):

\[ ps = \frac{\text{errorrange}}{100} \cdot h \]

From the relation \( d/s = 1/ps \) get:

\[ d = \frac{s}{\text{errorrange}/100 \cdot 2 \cdot \tan (\alpha/2)} \]

The program adds to this distance \( d \) the radius of the bounding sphere of the vertices.

### 4.1.5 Joining VRML nodes

Sometimes are polygon models broken into smaller parts (in extreme case into triangles) and computing levels of details has a tendency of ripping apart the model and produces useless LODs (a problem also reported by [18]).

Most of these degeneracies can be repaired with a simple joining algorithm, which join sequential \textit{IndexedFaceSets} into one.

Overview of the method:

```
Separator  original tree
Separator   <- this Separator join
Separator   <= this 2 lists
  Coord3
  IndexedFaceSet
Separator <=
  Coord3
  IndexedFaceSet
Separator
Separator
  Coord3
```
IndexedFaceSet

Remove not necessary *Separator* nodes to increase the percentage of joinable nodes:

Separator
  Separator  <- this Separator removes
  Separator  <= this not necessary
  Coord3
  IndexedFaceSet
Separator
  Separator
  Coord3
  IndexedFaceSet

and

Separator
  Separator
  Coord3
  IndexedFaceSet
Separator  <- this Separator removes
  Separator  <= this not necessary separator
  Coord3
  IndexedFaceSet

After join two *Separators* in a list:

Separator  <- this Separator join
  Separator  <= this 2 lists
  Coord3
  IndexedFaceSet
Separator  <=
  Coord3
  IndexedFaceSet

Remove not necessary *Separator*

Separator  <- this Separator removes
  Separator  <= this not necessary Separator
  Coord3
  IndexedFaceSet
And result:

```
Separator
  Coord3
  IndexedFaceSet
```

This join algorithm is also usable with Normal, Texture and Material nodes. For example:

Joining two Separator nodes with different Material sub nodes and possibly different IndexedFaceSet sub nodes.

```
Separator {
  Material { diffuse 0.5 0.6 0.4 } 
  IndexedFaceSet { coordIndex [1,2,3,-1] }
} Separator {
  Material { diffuse 0.3 0.3 0.3 } 
  IndexedFaceSet { coordIndex [4,5,6,-1] }
}
```

becomes

```
Separator {
  Material {
    diffuseColor [0.5 0.6 0.4, 0.3 0.3 0.3]
    ambientColor [0.2 0.2 0.2, 0.2 0.2 0.2]
    specularColor [0 0 0, 0 0 0]
    emissiveColor [0 0 0, 0 0 0]
    shininess [0.2, 0.2]
    transparency [0, 0]
  }
  MaterialBinding { value PER_FACE_INDEXED } 
  IndexedFaceSet { coordIndex [1,2,3,-1,4,5,6,-1]
    materialIndex [0,1] 
  }
}
```

Note is necessary to insert a new MaterialBinding so that the synthesized Material node is put in correct relation to the synthesized IndexedFaceSet. Is also necessary to create a complete Material node (ambientColor, specularColor, emissiveColor, shininess and transparency to the diffuseColor), to join this new created nodes with other nodes.
4.2 Details of object implementation

4.2.1 Classes derived from superclass “Node”

The class Node is used to store and handle VRML nodes - this class is able to contain any type of VRML nodes and many specialized classes are derived from this class (Figure 4.7):

**UseNode** - used for VRML node *USE*.

**GroupNode** - is able to contain subnodes. Used for VRML node *Group* and is superclass for Separator, TransformSeparator (and RootNode):

**RootNode** - has no relationship to VRML - used to store the VRML header and is root of all VRML nodes in a file.

**Separator** - used for VRML node *Separator*.

**TransformSeparator** - used for VRML node *TransformSeparator*.

**RotationNode** - used for VRML node *Rotation*.

**ScaleNode** - used for VRML node *Scale*.

**TransformNode** - used for VRML node *Transform*.

**MatrixTransformNode** - used for VRML node *MatrixTransform*.

**TranslationNode** - used for VRML node *Translation*.

**CoordNode** - used for VRML node *Coordinate3*. This class inserts with print-method all materials defined in this node ("Value" - list of field with field point) in an "OctreeCoordSet - object" or "ListCoordSet - object" (created in constructor) and deletes the list.

**MaterialNode** - used for VRML node *Material*. This class inserts with print-method all materials defined in this node ("Value" - lists of field with these fields: ambientColor, diffuseColor, specularColor, emissiveColor, shininess and transparency) in an "MaterialSet-object" (created in constructor) and deletes the lists.

**MaterialBindingNode** - used for VRML node *MaterialBinding*.

**NormalNode** - used for VRML node *Normal*. This class inserts with print-method all materials defined in this node ("Value" - list of field with field vector) in an "NormalSet-object" (created in constructor) and deletes the lists.

**NormalBindingNode** - used for VRML node *NormalBinding*. 

4.2 Details of Object Implementation

**Figure 4.7:** Inheritance of class “Node”.

**TextureCoordNode** - used for VRML node *TextureCoordinate2*. Inserts with print-method all materials defined in this node (“Value” - list of Field with field point) in an “TextureCoordSet-object” (created in constructor) and deletes the list.

**TextureNode** - used for VRML node *TextureNode*. 
**ShapeNode** - this class is a (abstract) superclass of class “FaceNode“ and class “LineNode“. It contains methods which are common to the next two subclasses:

**FaceNode** - used for VRML node *IndexedFaceSet*. It is able to transform the index-lists to a “FaceSet”, produce and print out the LODs - these actions are performed in the print-method. The print-method calls the method “generateFaceSet” which reads the index-lists and generates a “FaceSet”. Then the method generateLODs will be called. Then the print-method prints out the LODs - additionally some nodes are created temporary and printed out if required (e.g. *MaterialBinding*, *ShapeHints*, *Separator*, . . .)

**LineNode** - used for VRML node *IndexedLineSet*. It is able to transform the index-lists to a “LineSet”, produce and print out the LODs - these actions are performed in the print-method. The print-method calls the method “generateLineSet” which reads the index-lists and generates a “LineSet”. Then the method generateLODs will be called. Then the print-method prints out the LODs - additionally some nodes are created temporary and printed out if required (e.g. *MaterialBinding*, *ShapeHints*, *Separator*, . . .)

**PointNode** - used for VRML node *PointSet*.

**ConeNode** - used for VRML node *Cone*. The “ConeNode” has capabilities required for LOD-generation for this kind of geometry.

**CubeNode** - used for VRML node *Cube*. The “CubeNode” has capabilities required for LOD-generation for this kind of geometry (generates only original *Cube* and empty node).

**CylinderNode** - used for VRML node *Cylinder*. The “CylinderNode” has capabilities required for LOD-generation for this kind of geometry.

**SphereNode** - used for VRML node *Sphere*. The “SphereNode” has capabilities required for LOD-generation for this kind of geometry.

**ShapeHintsNode** - used for VRML node *ShapeHints*.

**UnknownNode** - used to store nodes which are not known by this program. Stores in a list the whole body of the node (evt. incl. subnodes).

**PerspectiveCameraNode** - used for VRML node *PerspectiveCamera* - only necessary for finding this type of camera via “setupCamera” method.

**OrthographicCameraNode** - used for VRML node *OrthographicCamera* - the only difference to class Node is that “setupCamera” method simply produces a warning.
4.2.2 Classes derived from superclass “Data”

Coord - class “Coord” can not be used directly (abstract), it contains all data and methods which are not related to a specific data structure such as an octree or list:

ListCoord - used to store a coordinate in an “ListCoordSet”.
OctreeCoord - used to store a coordinate in an “OctreeCoordSet”.

Material - used to store ambientColor, diffuseColor, specularColor, emissiveColor, shininess and transparency.

Normal - used to store a normal coordinate.

TextureCoord - used to store a texture coordinate.

![Image of class inheritance diagram]

Figure 4.8: Inheritance of class “Data”.

4.2.3 Classes derived from superclass “Dataset”

The class is a predecessor of the classes “CoordSet”, “MaterialSet”, “NormalSet”, “TextureCoordSet”, “FaceSet” and “LineSet”. It contains variables and methods which are common to all of these classes:

CoordSet - class “CoordSet” can not be used directly (abstract), it contains all data and methods which are not related to a specific data structure such as an octree or listCoord. The “CoordSet” has capabilities required for index-searching and LOD-generation.

ListCoordSet - the class “ListCoordSet” is used to store coordinates defined in VRML nodes from type Coordinate3. It removes double defined coordinates. This class is also used to generate LODs of coordinates. Coordinates which will be inserted in this class must be from class “ListCoord”. Used for Schroeder’s algorithm.
**OctreeCoordSet** - the class OctreeCoordSet is used to store coordinates defined in VRML nodes from type *Coordinate3*. It removes double defined coordinates. This class is also used to generate LODs of coordinates. Coordinates which will be inserted in this class must be from class “OctreeCoord”. Used for *vertex clustering algorithm*.

**MaterialSet** - the class “MaterialSet” is used to store materials defined in VRML nodes from type *Material*. It removes double defined Materials. Materials which will be inserted in this class must be from class “Material”.

**NormalSet** - the class “NormalSet” is used to store Normals defined in VRML nodes from type *Normal*. It removes double defined Normals. Normals which will be inserted in this class must be from class “Normal”.

**TextureCoordSet** - the class “TextureCoordSet” is used to store texture-coordinates defined in VRML nodes from type *TextureCoordinate2*. It removes double defined TextureCoords. Texture-coordinates which will be inserted in this class must be from class “TextureCoord”.

![Inheritance of class “DataSet”](image)

**4.2.4 Other classes**

**FaceSet** - the class “FaceSet” is used to store faces defined in VRML nodes from type *IndexedFaceSet*. Removes faces which are equal or part of already inserted faces by the insert method. The faces are inserted internally in a linear list. The FaceSet is also used to produce Levels of Detail. The method “generateNextLOD” creates a new “FaceSet” and fills this “FaceSet” with the faces of the next LOD.

**LineSet** - the class “LineSet” is used to store lines defined in VRML nodes of type *IndexedLineSet* (the class don’t supports polylines so it is necessary to split these polylines into normal lines). This class removes double defined lines (with same coordinates). The lines are inserted internally into a linear list. The “LineSet” is also used to produce levels of detail. The method “generateNextLOD” creates a new “LineSet” and fills this “LineSet” with the lines of the next LOD.
**State** - this class is used to store the current state while traversing the VRML node tree with method setupDepends with the information in this object e.g. the “FaceSet” is able to find the right `Coordinate3, Normal, current transform matrix, . . . and the actual bindings.

**ReferenceArray** - the class “referenceArray” implements an array that can grow at runtime according to the users needs. The array contains objects from type Reference. The class Reference contains only a pointer. This array class is used by the classes derived from DataSet to get an element by an index.

**Vertex** - is a container for pointer to Coord, Material, Normal and TextureCoord.

### 4.3 Parser

![Diagram of Parser](image)

**Figure 4.10: Parser.**

An interpreter for VRML is decomposed into two parts:

1. Read the source code and discover its structure.

2. Process this structure.

Tools Lex and Yacc ([16], [17]) can generate program fragments that solve the first task. The first task of discovering the source structure again is decomposed into subtasks (Figure 4.10):

1. Split the source file into tokens (Lex [16]).

2. Find the hierarchical structure of the source (Yacc [17]).
4.3.1 Flex

Flex is a tool for generating scanners: programs which recognize lexical patterns in text. Flex reads the given input files (or its standard input if no file names are given) for a description of the scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. Flex generates as output a C source file, `lex.yy.c`, which defines a routine `yylex`.

The first phase in a compiler reads the input source and converts strings in the source to tokens. Regular expressions are used for specify patterns to lex that allow it to scan and match strings in the input. Each pattern in lex has an associated action. Typically an action returns a token, representing the matched string, for subsequent use by the parser.

The flex input file consists of three sections, separated by a line with just `%%` in it:

```
definitions
  %
  rules
  %
  user code
```

The *definitions* section contains declarations of simple name definitions to simplify the scanner specification, and declarations of start conditions.

Name definitions have the form:

```
name definition
```

The *rules* section of the flex input contains a series of rules of the form:

```
pattern  action
```

where the pattern must be unindented and the action must begin on the same line.

Finally, the *user code* section is simply copied to `lex.yy.c` verbatim. It is used for companion routines which call or are called by the scanner. The presence of this section is optional; if it is missing, the second `%%` in the input file may be skipped.

When the generated scanner runs, it analyzes its input looking for strings which match any of its patterns. If it finds more than one match, it takes the one matching the most text (for trailing context rules, this includes the length of the trailing part, even though it will then be returned to the input). If it finds two or more matches of the same length, the rule listed first in the flex input file is chosen.

Once the match is determined, the text corresponding to the match (called the token) is made available in the global character pointer `yytext`, and its length in the global integer `yyleng`. The action corresponding to the matched pattern is then executed (a more detailed description of actions follows), and then the remaining input is scanned for another match.
4.3.2 Yacc-Bison

Bison is a general-purpose parser generator that converts a grammar description for an LALR(1) context-free grammar into a C program to parse that grammar.

The input file for the Bison utility is a Bison grammar file. The general form of a Bison grammar file is as follows:

{%
C declarations
%
Bison declarations
%
Grammar rules %
Additional C code
%

The “%%”, ‘%’ and ‘%’ are punctuation that appears in every Bison grammar file to separate the sections.

The C declarations may define types and variables used in the actions. It is also possible to use preprocessor commands to define macros used there, and use #include to include header files.

The Bison declarations declare the names of the terminal and nonterminal symbols, and may also describe operator precedence and the data types of semantic values of various symbols.

The grammar rules define how to construct each nonterminal symbol from its parts.

The additional C code can contain any C code whatsoever. Often the definition of the lexical analyzer yylex goes here, plus subroutines called by the actions in the grammar rules.

4.4 Problems

Problems during development and testing.

Problem with export VRML to Openinventor format

During testing files created by LODCreator was discovered this error:

```plaintext
WWWAnchor {
    name "0 3 626 192"
    Separator {
```
4.4 PROBLEMS

Translation {
translation 180.299988 0.55 -21.962
}
Rotation {
rotation -0.57735 -0.57735 0.57735 2.094395
}
Separator {
DEF green_felt Material {
ambientColor [ 0 0.03 0 ]
diffuseColor [ 0 1 0 ]
specularColor [ 0 0 0 ]
}
LOD {...}
}

export to:

#Inventor V2.1 ascii Separator
Translation {
translation 180.299988 0.55 -21.962
}
Rotation {
rotation -0.57735 -0.57735 0.57735 2.094395
}
Separator {
File { name "wk1.lod/green_felt.inc" }
LOD {...}
}

And appropriate error message form Openinventor:

Inventor read error: Expected '{'; got 'T'

Remark: Any node followed after WWWAnchor in form:

WWWAnchor {
.
.

is exported in non compatible format (without '{' and '}'), as shows example above (in this case is it Separator node).

This problem can be eliminated by writing WWWAnchor nodes in this form:
Conclusion: this error is caused by functionality of tool used for export VRML into the Inventor format.

Center problem

LODCreator computes the center of all polygon and polyline objects (IndexedFaceSet and IndexedLineSet nodes) from theirs bounding spheres. Center field is necessary for correct LOD switching (see Section 2.2). The specified center point of the LOD is transformed by the current transformation into world space, and the distance from the transformed center to the world-space viewpoint is calculated.

For example Rational Reducer does not work with center field. For correct LOD switching (in RR) is necessary to translate each IndexedFaceSet into the (0,0,0) position (accurate: previous Coord3 node) and write the Translation node before Coord3 node (to correct its real global position in scene).

Other VRML geometry (like Cone) are by default placed in the (0,0,0) position and transformed by the current cumulative transformation (transform matrix). RR compute the current center (0,0,0) of an object from the current transform matrix.

Note: in this case is not possible to use feature for joining nodes (see Section 4.1.5), because each of IndexedFaceSet and IndexedLineSet node has own center.

On the other side, system used by eSZett does not apply the current transform matrix on center field value of an object. This mean, is necessary to write exact (including current position changed by transform matrix) global position in scene to center field.

Example:

Original source:

```xml
#Cube with center position at 0 1 5
Separator {
  Translation { translation 0 0 5}
Separator {
    #Center of these coordinates is 0 1 0
    Coordinate3 {
      point [ -0.5 2 0.5,-0.5 2 -0.5, 0.5 2 -0.5,0.5 2 0.5, -0.5 0 0.5,-0.5 0 -0.5,
    }
}
```
Correct export for Rational Reducer:

Separator {
  Translation { translation 0 0 5}
  Separator {
    Translation { translation 0 1 0}
    #Center of these coordinates is 0 0 0
    Coordinate3 {
      point [ -0.5 1 0.5,-0.5 1 -0.5,
             0.5 1 -0.5,0.5 1 0.5,
             -0.5 -1 0.5,-0.5 -1 -0.5,
             0.5 0 -0.5,0.5 -1 0.5 ]
    }
    LOD {
      range [364.581635]
      #there is not "center" field, its function is
      #supplied by "Translation" node above
      IndexedFaceSet {
        coordIndex [ 4,7,3,0,-1,7,6,2,3,-1,
                     6,5,1,2,-1,5,4,0,1,-1,
                     0,3,2,1,-1,5,6,7,4,-1 ]
      }
      Separator {
      }
    }
  }
}

Correct export for system used by eSZett:

Separator {
  Translation { translation 0 0 5}
  Separator {
    Translation { translation 0 1 0}
    #Center of these coordinates is 0 0 0
    Coordinate3 {
      point [ 0.5 0 1 -0.5,0.5 0 0.5 ]
    }
    IndexedFaceSet {
      coordIndex [ 4,7,3,0,-1,7,6,2,3,-1,
                   6,5,1,2,-1,5,4,0,1,-1,
                   0,3,2,1,-1,5,6,7,4,-1 ]
    }
    Separator {
    }
  }
}

Correct export for system used by eSZett:
Note: In VRML1.0 specification is this not correct. In example above is center computed by VRML interpreter to (0, 1, 10). So correct form in VRML1.0 specification is:

Separator {
  Translation { translation 0 0 5}
  Separator {
    Coordinate3 {
      point [ -0.5 1 0.5, -0.5 1 -0.5,
                 0.5 1 -0.5, 0.5 1 0.5,
                 -0.5 -1 0.5, -0.5 -1 -0.5,
                 0.5 0 -0.5, 0.5 -1 0.5 ]
    }
    LOD {
      range [364.581635]
      #Actual center is computing with help of current
      #transform matrix by VRML interpreter (f.e. Cortona)
      center 0 1 0
      IndexedFaceSet {
        coordIndex [ 4, 7, 3, 0, -1, 7, 6, 2, 3, -1,
                      6, 5, 1, 2, -1, 5, 4, 0, 1, -1,
                      0, 3, 2, 1, -1, 5, 6, 7, -1,
                      5, 7, 4, -1 ]
      }
    }
  }
}

There is center position computed with help of current transform matrix $(0, 1, 5)$ by VRML interpreter:

$$P = C \cdot T,$$

where $T$ is the current transform matrix, $C$ is the center field value and $P$ is the new position of center:

$$(0, 1, 5, 1) = (0, 1, 0, 1) \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 5 & 1 \end{pmatrix}$$
Chapter 5

Testing

In this part three tools will be tested - two of them are Rational Reducer and tool from ČVUT, which are dealing only with Indexedfacets and do not create LOD node. VRML models and scenes are arranged in a hierarchical scene graph, so software dealing with single set of polygons is not sufficient. The LODCreator, which is tested as third, uses effective and simple solution by traversing the VRML model and apply the LOD generation to every VRML geometry individually, producing for each new LOD node (Figure 4.1).

For testing was chosen the “Stanford bunny” model with 69451 triangles, 34834 vertices and bounding box $0.156m \times 0.154m \times 0.12m$. The Bunny object is the traditional scanned mesh from Stanford University (Figure 5.1).

Figure 5.1: “Stanford bunny” and its bounding box.
5.1 Measurement of the output quality

If someone is manually creating levels of detail, the only error metric is his judgement. The modeler decides how to create a simple version of a detailed model and may also decide how small it must be on the screen before it looks good. This is an efficient error metric, because it uses the human visual system and human intuition to make an effective decision. But this (human) way is suitable only for LOD system that uses not more than a few discrete levels of details. For large models or scenes is this work labor intensive and is better use some tools for mesh simplification.

The main question is, how to measure output quality of automatic mesh simplification software results. Each of the tool use different error criteria to measure the fitness of approximated surfaces. Usually, these tools do not return measures of the error introduced while simplifying the mesh.

In the next sections will be described method measuring the quality of simplification objects - for the geometric and normal deviation.

5.2 Methodics of measuring

The one from measured software - LODCreator - simplify geometry and ignore the distortion caused to other surface attributes (like colors or textures). Some models are not only geometrically complex, but they may also have various surface properties such as color and texture. A students’ project [7] from the Czech technical university (ČVUT) is able to work with these attributes - color and textures. This tool uses a *quadric error metrics* algorithm [8] for mesh decimation.

In the next testing will be measured only the geometric and normal error, because next attributes are irrelevant in this work. The error representation is normalized for each simplified mesh, so the visual comparison is not possible. The normalized representation of deviations has chosen to highlight all measured values for each simplification software. Visual results are constructed by coloring the reference mesh according to the measured deviation. Figure 5.16 shows the color scale used: the left end is blue, representing minimum deviation, in the center is green which represents medium deviation, and red at the right end shows maximum deviation.

For measuring was used the *Mesh comparison* tool from Michael Roy [2].

5.2.1 Geometric deviation

Geometrical error describes the geometric differences between original and simplified object. This error is reported directly on the mesh, which allows visualization of the local error (Figure 5.14). The used software also returns
5.2 Methodics of Measuring

Numerical values such as mean error. The quality measurement is based on the point-to-surface distance:

**Definition 5.2.1** Let \( p \) be a point and \( S \) a surface. Then the point-to-surface distance \( e(p, S) \) is defined:

\[
e(p, S) = \min_{p' \in S} d(p, p'),
\]

(5.1)

where \( d(p, p') \) is the Euclidian distance between two points in \( \mathbb{R}^3 \).

![Figure 5.2: Point-to-surface distance.](image)

**Definition 5.2.2** The one-sided distance \( E \) between two surfaces \( S_1 \) and \( S_2 \) is defined:

\[
E(S_1, S_2) = \max_{p \in S_1} e(p, S_2).
\]

(5.2)

**Note:** This definition of distance is not symmetric. There exist surfaces such that \( E(S_1, S_2) \neq E(S_2, S_1) \), example is in Figure 5.3. The two-sided (Hausdorff) distance is defined:

**Definition 5.2.3** The two-sided (Hausdorff) distance \( H \) between two surfaces \( S_1 \) and \( S_2 \) is defined:

\[
H(S_1, S_2) = \max(E(S_1, S_2), E(S_2, S_1)).
\]

(5.3)

**Note:** In tests is this distance called “maximum”.

**Definition 5.2.4** The mean of geometric deviation \( E_m \) between two surfaces \( S_1 \) and \( S_2 \) is defined:

\[
E_m(S_1, S_2) = \frac{1}{|S_1|} \sum_{p_i \in S_1} e(p_i, S_2),
\]

(5.4)
where \( |S_1| \) is the sample size of \( S_1 \).

**Definition 5.2.5** The variance \( V \) for geometric deviation between two surfaces \( S_1 \) and \( S_2 \) is defined:

\[
V(S_1, S_2) = \frac{1}{|S_1|} \sum_{p_i \in S_1} (e(p_i, S_2) - E_m(S_1, S_2))^2,
\]

(5.5)

where \( |S_1| \) is the sample size of \( S_1 \).

### 5.2.2 Normal deviation

Normal deviation describes “how much is simplified object deformed”. Vertex normals are estimated averaging face normals in the vertex neighborhood. Measurement is based on the “distance” between two normals:

**Definition 5.2.6** A Given point \( p \in \mathbb{R}^3 \) on surface \( S_1 \) and a surface \( S_2 \), the normal deviation \( n(p, S) \) between point \( p \) and the surface \( S_2 \) is defined:

\[
n(p, S) = \arccos \left( \frac{\overrightarrow{A(p)} \cdot \overrightarrow{A(p_c)}}{|\overrightarrow{A(p)}| \cdot |\overrightarrow{A(p_c)}|} \right), \quad p_c = N(p, S)
\]

(5.6)

where \( \overrightarrow{A(p)} = \overrightarrow{v} \) is normal vector at point \( p \) and \( N(p, S_2) = p_c \) is the nearest point to point \( p \) on surface \( S_2 \).

If several points on the surface \( S_2 \) having the same distance to the point \( p \), the normal deviation is the minimum value between the normal of \( p \) to the normal of the nearest point to \( p \) on \( S_2 \).

Definition of the mean and variance for normal deviation is analogical to geometric deviation:

**Definition 5.2.7** The mean normal deviation \( E_m \) between two surfaces \( S_1 \) and \( S_2 \) is defined:

\[
N_m(S_1, S_2) = \frac{1}{|S_1|} \sum_{p_i \in S_1} n(p_i, S_2),
\]

(5.7)
where \(|S_1|\) is the sample size of \(S_1\).

**Definition 5.2.8** The variance \(V\) for normal deviation between two surfaces \(S_1\) and \(S_2\) is defined:

\[
V(S_1, S_2) = \frac{1}{|S_1|} \sum_{p_i \in S_1} \left( n(p_i, S_2) - N_m(S_1, S_2) \right)^2,
\]

(5.8)

where \(|S_1|\) is the sample size of \(S_1\).

## 5.3 Main testing

### 5.3.1 Rational Reducer

Rational Reducer is high-end tool from the Norwegian company “SIM” for reducing the number of polygons in 3D models. Rational Reducer is currently used by NASA, US Army, Toyota, Airbus and many others. An end user license costs 6559 USD (4850 Euro).

**Main features:**

- RR uses a surface simplification technique called edge collapse.
- Integrated OpenGL-based viewer.
- Multiple simultaneous viewer windows for comparison of models.
- Pre-processing of models to repair cracks and conflicts.
- Automatic optimization, with possibility of manual parameter tuning.
- Choice between fast reduction and high-quality reduction.
- Preservation of the structure and hierarchy of models. Preservation of textures and materials as well.
- Available also as a command-line (batch) option.
- Rational Reducer supports the following file formats:

  - VRML 1.0, VRML 2.0, VRML97
  - 3D Studio (.3ds)
  - AutoCAD DXF (.dxf)
  - OpenFlight (.flt)
5.3 Main testing

System requirements:
- Microsoft Windows 2000/XP.
- SGI IRIX.
- Linux/x86.

Results
Advantages:
- Excellent quality output (see Figures 5.7, 5.10, 5.11, 5.14, 5.15 and Tables 5.1, 5.2).
- Relatively fast (Figure 5.12 and Table 5.9).
- Works with color and texture attributes.

Disadvantages:
- Do not create LOD node.
- Do not work with other VRML geometry (Indexedlineset, Cone, Sphere, . . . )

5.3.2 ČVUT tool
Main features:
- Students’ work from Czech technical university.
- This tool uses a surface simplification technique called quadric error metrics [8].
- This tool supports only one file format:
  - VRML97.

System requirements:
- This tool requires Java Runtime Environment.
5.3 MAIN TESTING

Results

Advantages:

- Very good quality output (see Figures 5.8, 5.10, 5.11, 5.14, 5.15 and Tables 5.3, 5.4).
- Works with color and texture attributes.

Disadvantages:

- Do not create the LOD node.
- Do not work with other VRML geometry (Indexedlineset, Cone, Sphere, ...)
- Take a long time (Figure 5.12 and Table 5.9).

Note: This tool was implemented under the JAVA language, so is a little bit unfair to compare it with other two tools, which were implemented under C++.

5.3.3 LODCreator

Main features

- LODCreator supports only VRML 1.0 file format. Support for VRML97/X3D file formats and next extensions are possible to extend in the future.
- This tool reads a VRML file and generates LODs for each IndexedFaceSet, IndexedLineSet, Cylinder, Sphere, Cone and Cube.
- LoD Creator offers an easy control and work with the simple graphical using interface (GUI). It is possible to create LOD-files by several clicks. The only interaction needed from an user is loading the model, selecting the degree of complexity reduction, pushing the “Create lod” button to create files with LOD.
- LODCreator keeps important data (like WWW Anchor) and file hierarchy. The tool removes double defined materials, coordinates, normals, texture coordinates, faces and lines within the VRML nodes. All textures and materials are preserved as well.
- This tool has two modes for creating levels of details - automatial mode and manual mode.
- LODCreator uses two algorithms for mesh decimation - vertex clustering, which is very fast and robust, but sometimes produces worse output and Schroeder’s algorithm, which is a little bit slower and not too robust, but produces better results.
5.4 Summary

- The vertex clustering algorithm is memory efficient and extraordinarily fast, both crucial attributes for working with massive models.

- VRML models and scenes are arranged in a hierarchical scene graph, so a software dealing with a single set of polygons is not sufficient. The LODCreator uses an effective and simple solution by traversing the VRML model and apply the LOD generation to every VRML geometry individually, producing for each a new LOD node (Figure 4.1).

System requirements:

LODCreator was successfully tested on the following configurations:

- Intel PentiumMMX 200Mhz, 32MB RAM, WIN98
- AMD 1800+, 1.43GHz, 256MB RAM, WIN2000
- Mobile AMD Sempron 3000+ (1.79 GHz), 256MB RAM, WINXP

Vertex clustering

Advantages:

- Robustness, works with large sets of meshes (e.g. see Figure 5.4), which typically involve models from CAD-like systems.
- Execution speed (Figure 5.12 and Table 5.9).

Disadvantages:

- Sometimes poor results (see Figures 5.9, 5.10, 5.11, 5.14, 5.15 and Tables 5.5, 5.6).

Note: This algorithm was carefully tested only with meshes one’s type (non-manifold, “well-connected” - for example “Stanford Bunny”), so it is a little bit unfair to compare with other two tools, which are specialized in this kind of meshes. For another look, see Figure 5.4.

Vertex removal

Advantages:

- Fair quality result (see Figures 5.9, 5.10, 5.11, 5.14, 5.15 and Tables 5.7, 5.8).
- Relatively fast (Figure 5.12 and Table 5.9).

Disadvantages:

- Less robust than vertex clustering algorithm.
5.4 Summary

As it has been written, the vertex clustering algorithm has sometimes average output, but this disadvantage can be eliminated by LOD switching. If displayed in a size corresponding to the computed ranges, the quality degradation is no longer visible (see Figure 5.6 - are there some differences?).

It is possible to use Schroeder’s algorithm, but it is recommended to compare the statistic result (Figure A.9 in Appendix) with vertex clustering algorithm - and if will be markedly worse (mainly functional relation between distance and triangles), it would be better to use the vertex clustering algorithm. The main advantage of vertex clustering algorithm is its robustness suitable for working with data from “real-world” application, which involves “ill-behaved” meshes (Figures 5.4 and 3.5).

Figure 5.4: Simplification of the “t-junction” mesh, first picture is the original mesh (24 triangles), second is simplified from the LODCreator (vertex clustering was used), third is from the RR. (b) and (c) are with the 70% simplification (8 triangles). The ČVUT-tool ended with message “mesh collapse detection” and the LODCreator (with vertex decimation algorithm) generates empty set.

---

Figures 5.14 and 5.15 show the visual results obtained on a mesh simplified by using three tested software packages mentioned above.

Rational Reducer generates a low deviation on the entire mesh, but there are some (small) regions with medium deviation (Figure 5.7 and Tables 5.1 and 5.2). ČVUT-tool generates low deviation (Figure 5.8 and Tables 5.3 and 5.4) on the entire mesh. LODCreator with vertex clustering algorithm generates medium deviation with greater variance (Figure 5.9 and Tables 5.5 and 5.6). Better results has the LODCreator with vertex decimation algorithm, where is result similar as the ČVUT-tool (Figure 5.9 and Tables 5.7 and 5.8).

Figures 5.10 and 5.11 represents the graphs of numerical results obtained from the simplified forms of the previous mesh. Simplified meshes are made using the three above tested software packages. Figure 5.10 shows the mean geometric deviation. Figure 5.11 shows the mean normal deviation. These graphs confirm the remarks made above. Rational Reducer obtained the lowest mean deviation with high percentage mesh simplification (above 60%). With lower values (under 60%) are differences between tested software insignificant (≈ thousandths of percents). Rational Reducer perfectly manages the appearance
Figure 5.5: Results for comparison: the first LOD is original object, the second is with 50% simplification, the third with 80% and the last with 95%. Picture (a) is output from the Rational Reducer and (b) is from the LODCreator (vertex clustering algorithm is used).

Figure 5.6: Results for comparison: the first LOD is original object, the second is with 50% simplification, the third with 80% and the last with 95%. Picture (a) is output from the ČVUT-tool and (b) is from the LODCreator (vertex decimation algorithm is used).

attributes during the simplification process and generates low deviation on the entire mesh.
### 5.5 Tables and Figures

#### Rational Reducer

<table>
<thead>
<tr>
<th>Reduction</th>
<th>minimum</th>
<th>maximum</th>
<th>mean</th>
<th>variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>10% (62507△)</td>
<td>0</td>
<td>5.64 · 10^{-5}</td>
<td>1.66 · 10^{-6}(0.001%)</td>
<td>2.23 · 10^{-11}</td>
</tr>
<tr>
<td>20% (55561△)</td>
<td>0</td>
<td>7.91 · 10^{-5}</td>
<td>4.22 · 10^{-6}(0.003%)</td>
<td>6.74 · 10^{-11}</td>
</tr>
<tr>
<td>30% (48617△)</td>
<td>0</td>
<td>8.24 · 10^{-5}</td>
<td>7.48 · 10^{-6}(0.005%)</td>
<td>13.2 · 10^{-11}</td>
</tr>
<tr>
<td>40% (41671△)</td>
<td>0</td>
<td>11.72 · 10^{-5}</td>
<td>11.4 · 10^{-6}(0.007%)</td>
<td>21.4 · 10^{-11}</td>
</tr>
<tr>
<td>50% (34727△)</td>
<td>0</td>
<td>11.72 · 10^{-5}</td>
<td>16.3 · 10^{-6}(0.011%)</td>
<td>32.2 · 10^{-11}</td>
</tr>
<tr>
<td>60% (27781△)</td>
<td>0</td>
<td>17.37 · 10^{-5}</td>
<td>22.5 · 10^{-6}(0.014%)</td>
<td>49.1 · 10^{-11}</td>
</tr>
<tr>
<td>70% (20836△)</td>
<td>0</td>
<td>22.48 · 10^{-5}</td>
<td>31.43 · 10^{-6}(0.02%)</td>
<td>83.1 · 10^{-11}</td>
</tr>
<tr>
<td>80% (13891△)</td>
<td>0</td>
<td>47.27 · 10^{-5}</td>
<td>44.82 · 10^{-6}(0.029%)</td>
<td>170.1 · 10^{-11}</td>
</tr>
<tr>
<td>90% (6945△)</td>
<td>0</td>
<td>83.02 · 10^{-5}</td>
<td>68.17 · 10^{-6}(0.044%)</td>
<td>373.7 · 10^{-11}</td>
</tr>
<tr>
<td>95% (3473△)</td>
<td>0</td>
<td>170.7 · 10^{-5}</td>
<td>104.0 · 10^{-6}(0.067%)</td>
<td>787.9 · 10^{-11}</td>
</tr>
<tr>
<td>99% (693△)</td>
<td>0</td>
<td>519.8 · 10^{-5}</td>
<td>345.5 · 10^{-6}(0.222%)</td>
<td>9905 · 10^{-11}</td>
</tr>
</tbody>
</table>

Table 5.1: Geometric deviation - Rational Reducer.

<table>
<thead>
<tr>
<th>Reduction</th>
<th>minimum</th>
<th>maximum</th>
<th>mean</th>
<th>variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>10% (62507△)</td>
<td>0</td>
<td>0.1815</td>
<td>4.136 · 10^{-3}</td>
<td>6.534 · 10^{-5}</td>
</tr>
<tr>
<td>20% (55561△)</td>
<td>0</td>
<td>0.2414</td>
<td>8.816 · 10^{-3}</td>
<td>14.74 · 10^{-5}</td>
</tr>
<tr>
<td>30% (48617△)</td>
<td>0</td>
<td>0.5123</td>
<td>13.92 · 10^{-3}</td>
<td>24.67 · 10^{-5}</td>
</tr>
<tr>
<td>40% (41671△)</td>
<td>0</td>
<td>0.5039</td>
<td>19.51 · 10^{-3}</td>
<td>34.36 · 10^{-5}</td>
</tr>
<tr>
<td>50% (34727△)</td>
<td>0</td>
<td>0.4580</td>
<td>25.99 · 10^{-3}</td>
<td>46.83 · 10^{-5}</td>
</tr>
<tr>
<td>60% (27781△)</td>
<td>0</td>
<td>0.5996</td>
<td>33.97 · 10^{-3}</td>
<td>69.3 · 10^{-5}</td>
</tr>
<tr>
<td>70% (20836△)</td>
<td>0</td>
<td>0.5996</td>
<td>44.64 · 10^{-3}</td>
<td>115.2 · 10^{-5}</td>
</tr>
<tr>
<td>80% (13891△)</td>
<td>0</td>
<td>1.3762</td>
<td>58.82 · 10^{-3}</td>
<td>218.8 · 10^{-5}</td>
</tr>
<tr>
<td>90% (6945△)</td>
<td>0</td>
<td>1.2789</td>
<td>84.28 · 10^{-3}</td>
<td>407.3 · 10^{-5}</td>
</tr>
<tr>
<td>95% (3473△)</td>
<td>0.001419</td>
<td>0.7788</td>
<td>112.6 · 10^{-3}</td>
<td>605.2 · 10^{-5}</td>
</tr>
<tr>
<td>99% (693△)</td>
<td>0.006859</td>
<td>1.3631</td>
<td>227.8 · 10^{-3}</td>
<td>2389 · 10^{-5}</td>
</tr>
</tbody>
</table>

Table 5.2: Normal deviation - Rational Reducer.
Figure 5.7: Geometric deviation - Rational Reducer.
5.5 Tables and Figures

ČVUT - tool

<table>
<thead>
<tr>
<th>Reduction</th>
<th>maximum</th>
<th>mean</th>
<th>variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>10% (62504(\triangle))</td>
<td>2.565 \cdot 10^{-5}</td>
<td>0.82 \cdot 10^{-6}(0.0005%)</td>
<td>0.505 \cdot 10^{-11}</td>
</tr>
<tr>
<td>20% (55559(\triangle))</td>
<td>2.987 \cdot 10^{-5}</td>
<td>2.475 \cdot 10^{-6}(0.002%)</td>
<td>1.970 \cdot 10^{-11}</td>
</tr>
<tr>
<td>30% (48614(\triangle))</td>
<td>4.468 \cdot 10^{-5}</td>
<td>4.906 \cdot 10^{-6}(0.003%)</td>
<td>4.511 \cdot 10^{-11}</td>
</tr>
<tr>
<td>40% (41669(\triangle))</td>
<td>6.492 \cdot 10^{-5}</td>
<td>8.264 \cdot 10^{-6}(0.005%)</td>
<td>8.752 \cdot 10^{-11}</td>
</tr>
<tr>
<td>50% (34725(\triangle))</td>
<td>11.48 \cdot 10^{-5}</td>
<td>12.72 \cdot 10^{-6}(0.008%)</td>
<td>17.24 \cdot 10^{-11}</td>
</tr>
<tr>
<td>60% (27780(\triangle))</td>
<td>18.26 \cdot 10^{-5}</td>
<td>20.76 \cdot 10^{-6}(0.013%)</td>
<td>46.98 \cdot 10^{-11}</td>
</tr>
<tr>
<td>70% (20835(\triangle))</td>
<td>78.49 \cdot 10^{-5}</td>
<td>43.50 \cdot 10^{-6}(0.028%)</td>
<td>354.3 \cdot 10^{-11}</td>
</tr>
<tr>
<td>80% (13889(\triangle))</td>
<td>78.49 \cdot 10^{-5}</td>
<td>66.24 \cdot 10^{-6}(0.043%)</td>
<td>558.9 \cdot 10^{-11}</td>
</tr>
<tr>
<td>90% (6945(\triangle))</td>
<td>162.0 \cdot 10^{-5}</td>
<td>144.3 \cdot 10^{-6}(0.092%)</td>
<td>2568 \cdot 10^{-11}</td>
</tr>
<tr>
<td>95% (3471(\triangle))</td>
<td>173.0 \cdot 10^{-5}</td>
<td>222.9 \cdot 10^{-6}(0.143%)</td>
<td>4911 \cdot 10^{-11}</td>
</tr>
<tr>
<td>99% (697(\triangle))</td>
<td>499.7 \cdot 10^{-5}</td>
<td>581.9 \cdot 10^{-6}(0.373%)</td>
<td>28579 \cdot 10^{-11}</td>
</tr>
</tbody>
</table>

Table 5.3: Geometric deviation - ČVUT - tool.

<table>
<thead>
<tr>
<th>Reduction</th>
<th>minimum</th>
<th>maximum</th>
<th>mean</th>
<th>variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>10% (62507(\triangle))</td>
<td>0</td>
<td>0.1894</td>
<td>5.105 \cdot 10^{-3}</td>
<td>11.48 \cdot 10^{-5}</td>
</tr>
<tr>
<td>20% (55561(\triangle))</td>
<td>0</td>
<td>0.3893</td>
<td>10.31 \cdot 10^{-3}</td>
<td>21.48 \cdot 10^{-5}</td>
</tr>
<tr>
<td>30% (48617(\triangle))</td>
<td>0</td>
<td>0.3893</td>
<td>15.73 \cdot 10^{-3}</td>
<td>31.59 \cdot 10^{-5}</td>
</tr>
<tr>
<td>40% (41671(\triangle))</td>
<td>0</td>
<td>0.4656</td>
<td>21.37 \cdot 10^{-3}</td>
<td>44.04 \cdot 10^{-5}</td>
</tr>
<tr>
<td>50% (34727(\triangle))</td>
<td>0</td>
<td>0.4655</td>
<td>27.69 \cdot 10^{-3}</td>
<td>62.10 \cdot 10^{-5}</td>
</tr>
<tr>
<td>60% (27781(\triangle))</td>
<td>0</td>
<td>0.4778</td>
<td>37.75 \cdot 10^{-3}</td>
<td>101.0 \cdot 10^{-5}</td>
</tr>
<tr>
<td>70% (20836(\triangle))</td>
<td>0</td>
<td>0.4699</td>
<td>55.70 \cdot 10^{-3}</td>
<td>195.3 \cdot 10^{-5}</td>
</tr>
<tr>
<td>80% (13891(\triangle))</td>
<td>0</td>
<td>0.7410</td>
<td>79.08 \cdot 10^{-3}</td>
<td>349.7 \cdot 10^{-5}</td>
</tr>
<tr>
<td>90% (6945(\triangle))</td>
<td>0</td>
<td>0.7948</td>
<td>127.4 \cdot 10^{-3}</td>
<td>776.4 \cdot 10^{-5}</td>
</tr>
<tr>
<td>95% (3473(\triangle))</td>
<td>0</td>
<td>0.9063</td>
<td>167.1 \cdot 10^{-3}</td>
<td>1346 \cdot 10^{-5}</td>
</tr>
<tr>
<td>99% (693(\triangle))</td>
<td>0.003915</td>
<td>1.727</td>
<td>287 \cdot 10^{-3}</td>
<td>4015 \cdot 10^{-5}</td>
</tr>
</tbody>
</table>

Table 5.4: Normal deviation - ČVUT - tool.
Figure 5.8: Geometric deviation - ČVUT - tool.
**LODCreator - vertex clustering**

<table>
<thead>
<tr>
<th>Reduction</th>
<th>maximum</th>
<th>mean</th>
<th>variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>10% (62506△)</td>
<td>92.11 · 10^{-5}</td>
<td>3.793 · 10^{-6}(0.002%)</td>
<td>35.65 · 10^{-11}</td>
</tr>
<tr>
<td>20% (55554△)</td>
<td>92.11 · 10^{-5}</td>
<td>11.88 · 10^{-6}(0.008%)</td>
<td>183.3 · 10^{-11}</td>
</tr>
<tr>
<td>30% (48613△)</td>
<td>105.6 · 10^{-5}</td>
<td>21.13 · 10^{-6}(0.015%)</td>
<td>320.6 · 10^{-11}</td>
</tr>
<tr>
<td>40% (41671△)</td>
<td>171.6 · 10^{-5}</td>
<td>31.07 · 10^{-6}(0.02%)</td>
<td>459.8 · 10^{-11}</td>
</tr>
<tr>
<td>50% (34713△)</td>
<td>113.6 · 10^{-5}</td>
<td>41.73 · 10^{-6}(0.026%)</td>
<td>610.9 · 10^{-11}</td>
</tr>
<tr>
<td>60% (27752△)</td>
<td>113.6 · 10^{-5}</td>
<td>51.33 · 10^{-6}(0.033%)</td>
<td>698.1 · 10^{-11}</td>
</tr>
<tr>
<td>70% (20796△)</td>
<td>278.0 · 10^{-5}</td>
<td>71.20 · 10^{-6}(0.046%)</td>
<td>1426 · 10^{-11}</td>
</tr>
<tr>
<td>80% (13882△)</td>
<td>319.8 · 10^{-5}</td>
<td>136.8 · 10^{-6}(0.088%)</td>
<td>4007 · 10^{-11}</td>
</tr>
<tr>
<td>90% (6918△)</td>
<td>325.4 · 10^{-5}</td>
<td>200.1 · 10^{-6}(0.128%)</td>
<td>5807 · 10^{-11}</td>
</tr>
<tr>
<td>95% (3482△)</td>
<td>691.8 · 10^{-5}</td>
<td>487.2 · 10^{-6}(0.312%)</td>
<td>32991 · 10^{-11}</td>
</tr>
<tr>
<td>99% (697△)</td>
<td>1537 · 10^{-5}</td>
<td>1689 · 10^{-6}(1.082%)</td>
<td>608071 · 10^{-11}</td>
</tr>
</tbody>
</table>

Table 5.5: Geometric deviation - LoDCreator (vertex clustering).

<table>
<thead>
<tr>
<th>Reduction</th>
<th>minimum</th>
<th>maximum</th>
<th>mean</th>
<th>variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>10% (62507△)</td>
<td>0</td>
<td>0.557</td>
<td>11.46 · 10^{-3}</td>
<td>58.97 · 10^{-5}</td>
</tr>
<tr>
<td>20% (55561△)</td>
<td>0</td>
<td>1.298</td>
<td>23.43 · 10^{-3}</td>
<td>195.3 · 10^{-5}</td>
</tr>
<tr>
<td>30% (48617△)</td>
<td>0</td>
<td>1.982</td>
<td>34.66 · 10^{-3}</td>
<td>336.3 · 10^{-5}</td>
</tr>
<tr>
<td>40% (41671△)</td>
<td>0</td>
<td>1.999</td>
<td>47.12 · 10^{-3}</td>
<td>446.5 · 10^{-5}</td>
</tr>
<tr>
<td>50% (34727△)</td>
<td>0</td>
<td>1.999</td>
<td>58.96 · 10^{-3}</td>
<td>537.6 · 10^{-5}</td>
</tr>
<tr>
<td>60% (27781△)</td>
<td>0</td>
<td>1.999</td>
<td>71.81 · 10^{-3}</td>
<td>585.1 · 10^{-5}</td>
</tr>
<tr>
<td>70% (20836△)</td>
<td>0</td>
<td>2</td>
<td>87.29 · 10^{-3}</td>
<td>755.3 · 10^{-5}</td>
</tr>
<tr>
<td>80% (13891△)</td>
<td>0</td>
<td>2</td>
<td>118.4 · 10^{-3}</td>
<td>1183 · 10^{-5}</td>
</tr>
<tr>
<td>90% (6945△)</td>
<td>0.0003</td>
<td>2</td>
<td>151.1 · 10^{-3}</td>
<td>1653 · 10^{-5}</td>
</tr>
<tr>
<td>95% (3473△)</td>
<td>0.0009</td>
<td>1.998</td>
<td>239.7 · 10^{-3}</td>
<td>4581 · 10^{-5}</td>
</tr>
<tr>
<td>99% (693△)</td>
<td>0.003463</td>
<td>1.998</td>
<td>322.3 · 10^{-3}</td>
<td>7120 · 10^{-5}</td>
</tr>
</tbody>
</table>

Table 5.6: Normal deviation - LoDCreator (vertex clustering).
Figure 5.9: Geometric deviation - LoDCreator.
## 5.5 Tables and Figures

### LODCreator - vertex decimation

<table>
<thead>
<tr>
<th>Reduction</th>
<th>maximum</th>
<th>mean</th>
<th>variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>10% (62504Δ)</td>
<td>78.32 · 10⁻⁵</td>
<td>0.936 · 10⁻⁶(0.0006%)</td>
<td>9.989 · 10⁻¹¹</td>
</tr>
<tr>
<td>20% (55539Δ)</td>
<td>78.38 · 10⁻⁵</td>
<td>2.768 · 10⁻⁶(0.0018%)</td>
<td>14.87 · 10⁻¹¹</td>
</tr>
<tr>
<td>30% (48576Δ)</td>
<td>92.11 · 10⁻⁵</td>
<td>5.546 · 10⁻⁶(0.004%)</td>
<td>28.13 · 10⁻¹¹</td>
</tr>
<tr>
<td>40% (41632Δ)</td>
<td>92.11 · 10⁻⁵</td>
<td>9.317 · 10⁻⁶(0.006%)</td>
<td>49.37 · 10⁻¹¹</td>
</tr>
<tr>
<td>50% (34683Δ)</td>
<td>113.6 · 10⁻⁵</td>
<td>14.46 · 10⁻⁶(0.009%)</td>
<td>73.27 · 10⁻¹¹</td>
</tr>
<tr>
<td>60% (27728Δ)</td>
<td>117.5 · 10⁻⁵</td>
<td>22.09 · 10⁻⁶(0.014%)</td>
<td>113.3 · 10⁻¹¹</td>
</tr>
<tr>
<td>70% (20728Δ)</td>
<td>151.8 · 10⁻⁵</td>
<td>33.82 · 10⁻⁶(0.022%)</td>
<td>191.6 · 10⁻¹¹</td>
</tr>
<tr>
<td>80% (13837Δ)</td>
<td>151.8 · 10⁻⁵</td>
<td>54.76 · 10⁻⁶(0.035%)</td>
<td>334.1 · 10⁻¹¹</td>
</tr>
<tr>
<td>90% (6895Δ)</td>
<td>151.8 · 10⁻⁵</td>
<td>109.2 · 10⁻⁶(0.07%)</td>
<td>954.7 · 10⁻¹¹</td>
</tr>
<tr>
<td>95% (3440Δ)</td>
<td>188.1 · 10⁻⁵</td>
<td>200.3 · 10⁻⁶(0.128%)</td>
<td>2740 · 10⁻¹¹</td>
</tr>
<tr>
<td>99% (678Δ)</td>
<td>425.8 · 10⁻⁵</td>
<td>720.3 · 10⁻⁶(0.462%)</td>
<td>34622 · 10⁻¹¹</td>
</tr>
</tbody>
</table>

Table 5.7: Geometric deviation - LoDCreator (vertex decimation).

<table>
<thead>
<tr>
<th>Reduction</th>
<th>minimum</th>
<th>maximum</th>
<th>mean</th>
<th>variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>10% (62504Δ)</td>
<td>0</td>
<td>0.524</td>
<td>6.528 · 10⁻³</td>
<td>37.17 · 10⁻⁵</td>
</tr>
<tr>
<td>20% (55539Δ)</td>
<td>0</td>
<td>0.758</td>
<td>14.54 · 10⁻³</td>
<td>97.40 · 10⁻⁵</td>
</tr>
<tr>
<td>30% (48576Δ)</td>
<td>0</td>
<td>0.634</td>
<td>22.57 · 10⁻³</td>
<td>141.9 · 10⁻⁵</td>
</tr>
<tr>
<td>40% (41632Δ)</td>
<td>0</td>
<td>1.419</td>
<td>30.77 · 10⁻³</td>
<td>190.5 · 10⁻⁵</td>
</tr>
<tr>
<td>50% (34683Δ)</td>
<td>0</td>
<td>1.419</td>
<td>40.56 · 10⁻³</td>
<td>243.2 · 10⁻⁵</td>
</tr>
<tr>
<td>60% (27728Δ)</td>
<td>0</td>
<td>1.419</td>
<td>51.74 · 10⁻³</td>
<td>320.6 · 10⁻⁵</td>
</tr>
<tr>
<td>70% (20728Δ)</td>
<td>0</td>
<td>1.99</td>
<td>65.63 · 10⁻³</td>
<td>408.1 · 10⁻⁵</td>
</tr>
<tr>
<td>80% (13837Δ)</td>
<td>0</td>
<td>1.99</td>
<td>82.75 · 10⁻³</td>
<td>533.8 · 10⁻⁵</td>
</tr>
<tr>
<td>90% (6895Δ)</td>
<td>0</td>
<td>1.99</td>
<td>115.8 · 10⁻³</td>
<td>765.9 · 10⁻⁵</td>
</tr>
<tr>
<td>95% (3440Δ)</td>
<td>0</td>
<td>1.99</td>
<td>159 · 10⁻³</td>
<td>1304 · 10⁻⁵</td>
</tr>
<tr>
<td>99% (678Δ)</td>
<td>0.005856</td>
<td>1.998</td>
<td>283.8 · 10⁻³</td>
<td>3360 · 10⁻⁵</td>
</tr>
</tbody>
</table>

Table 5.8: Normal deviation - LoDCreator (vertex decimation).
Summary

Figure 5.10: All - geometric deviation.
Figure 5.11: All - normal deviation.

<table>
<thead>
<tr>
<th>Simplification</th>
<th>Rational r.</th>
<th>CVUT</th>
<th>LoDCreator v.c.</th>
<th>LoDCreator v.d.</th>
</tr>
</thead>
<tbody>
<tr>
<td>10% (62506△)</td>
<td>4.1</td>
<td>23.5</td>
<td>1.5</td>
<td>2.5</td>
</tr>
<tr>
<td>20% (55554△)</td>
<td>5.1</td>
<td>35</td>
<td>1.39</td>
<td>3.1</td>
</tr>
<tr>
<td>30% (48613△)</td>
<td>6.1</td>
<td>45.7</td>
<td>1.4</td>
<td>3.9</td>
</tr>
<tr>
<td>40% (41671△)</td>
<td>7.2</td>
<td>55</td>
<td>1.42</td>
<td>4.6</td>
</tr>
<tr>
<td>50% (34713△)</td>
<td>8.4</td>
<td>66.2</td>
<td>1.44</td>
<td>5.2</td>
</tr>
<tr>
<td>60% (27752△)</td>
<td>9.6</td>
<td>89.2</td>
<td>1.47</td>
<td>6.0</td>
</tr>
<tr>
<td>70% (20796△)</td>
<td>10.8</td>
<td>108</td>
<td>1.47</td>
<td>6.9</td>
</tr>
<tr>
<td>80% (13882△)</td>
<td>11.8</td>
<td>149</td>
<td>1.47</td>
<td>7.5</td>
</tr>
<tr>
<td>90% (6918△)</td>
<td>12.9</td>
<td>170</td>
<td>1.41</td>
<td>8</td>
</tr>
<tr>
<td>95% (3482△)</td>
<td>13.5</td>
<td>177.5</td>
<td>1.42</td>
<td>8.4</td>
</tr>
<tr>
<td>99% (697△)</td>
<td>13.7</td>
<td>178.6</td>
<td>1.39</td>
<td>8.7</td>
</tr>
</tbody>
</table>

Table 5.9: Execution speed.
Figure 5.12: Execution speed.
Figure 5.13: Bunny (99% simplification). Simplified meshes are from Rational Reducer, ČVUT - tool, LODCreator - vertex clustering, LODCreator - vertex decimation (from left to right).

Figure 5.14: Bunny - geometric deviation.

Figure 5.15: Bunny - normal deviation.

Figure 5.16: Color scale.
Chapter 6

Summary and future work

All the defined goals of the diploma thesis were fulfilled - the tool for automatical creating LOD was developed. The LODCreator was tested on real and complex models provided by eSZett. The tool is ready to use, graphic user interface is developed to make the tool more user friendly. The results and function of the LODCreator are satisfactory. However, there will be always something to improve:

Substitution polygonal objects

Sometimes with LOD creation process (especially if the count of LODs is high for each node or there is more Indexedfacesets nodes ) can files increase in size more than two times.

Reasonable solution might be a substitution polygons (Indexedfacesets) by other VRML geometry, if they are “similar” to sphere, cone, cube or cylinder, with user defined parameter, which define limit when two object are similar and when not, for example:

Separator {
    Coordinate3 {
        point [ -0.5 1 0.5, -0.5 1 -0.5, 0.5 1 -0.5, 0.5 1 0.5, -0.5 -1 0.5, -0.5 -1 -0.5, 0.5 -1 -0.5, 0.5 -1 0.5] }
    IndexedFaceSet {
        coordIndex [ 4, 7, 3, 0, -1,7, 6, 2, 3, -1, 6, 5, 1, 2, -1,5, 4, 0, 1, -1, 0, 3, 2, 1, -1,5, 6, 7, 4, -1 ]
    }
}

becomes
Merging objects

On Figure 6.3(a) is cylinder composed of two parts (cylinders). If these structures will occur in scene, LOD switching looks not appropriate (Figure 6.3(b)).

Solution: find out these structures in VRML tree and replace them by uniform object (using the technique mentioned above).

Partially it can help smaller value of error range option (Figure 6.3 (c) and (d) - LOD switching is performed in greater distance).

Distortion of structures

This problem is similar to problem mentioned above: if in a scene is some complex structure (Figure 6.1) composed of more smaller parts of different sizes, LOD switching is performed on each of this small structure in different distances (Figure 6.2), eventually some of these smaller parts are can completely disappear.

The solution for this kind of distortion is more complicated - would require a deep analysis of model structure and geometry.

Figure 6.1: Pipes, original object.
In last two cases can help right (=smaller) value for parameter error range - LOD switching is performed in greater distances. How parameter error range affects the result shows Figure 6.3.

Next possible upgrades (only on eSZett request):

- Extend algorithms for mesh decimation for next attributes (color and texture).
- Improve vertex clustering algorithm (non-uniform clustering).
- Works on GUI - implement threads, openGL viewer with LOD switching, better VRML editor (syntax highlighting), etc.
- Optionally implement a parser for next languages (VRML97, X3D).
Figure 6.3: Cylinders a) original object, b) errorrange=10 c) errorrange=2 d) errorrange=0.5.
Appendix A

User’s manual

Here is the complete reference to all functions available from the LODCreator user interface.

A.1 Menu system

The menu system may be accessed through the menu bar on the top of the main LODCreator window, or through a popup-menu activated by the right mouse button in any file you happen to have in the list of currently loaded files.

A.1.1 Toolbar

The toolbar (see Figure A.1) gives you quick access to the most frequently used menu items.

Open - Opens a new file.

Parse! - Loads a VRML file and setup the specific data structures.

Make lod - Creates a new file(s) with levels of details for the selected file. Call “parse” function before creating LOD. If a name of the file is not specified by the “save scene as” item, you will be asked for a new name.
Delete - Removes the selected model and cleans up allocated memory.

Edit - Opens a window for editing a currently selected file.

Show - Opens a viewer.

### A.1.2 File menu

![File menu](image)

**Open file** - Opens a new file.

**Save file as** - Gives to opened file a new filename.

**Preferences** - Opens the application preferences dialog. See Section “Preferences dialog” on page 78, for further documentation of this menu item.

**Exit** - Quits the application. If there are any opened files which have not been saved yet, you will be asked whether you want to quit without saving them.

### A.1.3 Action menu

Some items from this menu will not be available unless at least one model was loaded and there is naturally no use for them if no file is selected.

![Action menu](image)
**A.1 Menu system**

**Parse!** - Loads a VRML file and setup the specific data structures, usable if you would like to get some information about file, see Section A.2.1 “Main LoDCreator Window”.

**Make lod** - Creates a new file(s) with levels of details for the selected file(s). If the new file name is not specific with the “save scene as” item, you will be asked for new file name. For more information see the Section A.3.3 “LoD dialog”.

**Delete** - Removes the currently selected scene and releases the memory resources it holds.

### A.1.4 Window menu

This menu contains main functions working on loaded files.

![Window menu](image)

**Figure A.4: Window menu.**

**View scene**

**LoD dialog** - Opens the dialog window with controls and options for the LOD creation process. See Section A.3.3 “LoD dialog” for instructions on using the reduction dialog.

**Statistic dialog** - Opens a dialog window which shows information on the currently selected model, like the number of triangle primitives, etc. See Section A.3.2 “Statistic dialog”.

**Messages Windows** - Opens the message window which displays any messages of the processes LODCreator has executed. See Section A.3.5 “Messages window”.

**Edit** - Opens a window for editing the currently selected file.

**View tree** - Shows a VRML tree-viewer for the currently selected file. See Section A.3.1 “The tree dialog”.

---

1 Not implemented yet
A.2 Main LODCreator window

A.1.5 Help menu

![Help menu]

Figure A.5: Help menu.

**Manual** - Opens this manual in default browser.

**About** - Pops up a window with some basic information about the version of the LODCreator application you are running including contact information.

A.2 Main LODCreator window

Apart from the menu bar, the main LODCreator window contains these components:

A.2.1 List of currently available files

Here you can select for which files would like to create levels of details, for which file you want to delete, view, edit, etc. Pressing the right mouse button on any of the line in the model list pops up a menu for quick access to frequently used operations.

![List of currently available files]

Figure A.6: Main window.
After a file has been loaded, it is possible to see its name, path, file size and nothing else - other information are not accessible, because the file is not parsed (LODCreator gets only filename, path and size) and data structures are not created. This feature can be useful for very large models, typically for files of size larger than 100 MB, because the performance of LODCreator may be significantly reduced on computers with limited amount of memory.

Values presented on the right of the “file size” column is the current numbers of VRML primitives present in the model.

A.2.2 Status bar

![Status bar](image)

Figure A.7: Status bar.

The left part of the status bar displays a description of the currently running process (parsing, opening file, ...). The right part displays name of the current selected file.

A.3 The dialogs

A.3.1 Tree dialog

This dialog provides a basic information about the VRML-tree structure (Figure A.8).

A.3.2 Statistic dialog

This window contains a basic statistic information (Figure A.9) about LODs created in current file. From edit boxes below are taken number of triangles for Cone, Sphere and Cylinder necessary for calculation count of triangles in scene.

A.3.3 Level of detail dialog

This dialog offers main options for the LOD creation process (see Figure A.3.3). From here it is possible to set properties and the quality of the output file with LOD. It is possible to set LOD-output for each of VRML shape (Cone, Cylinder, ...) by clicking to tabs. This dialog includes options:

**Full automatically process** The LODCreator computes the number of LODs and theirs ranges automatically, results may be different, depends on few parameters in advanced dialog, see Figure A.11).
Manual settings  In this mode is possible to determine the number of LOD for each kind of VRML shape by hand. Values from edit boxes below are taken for range field in LOD node.

Differentiate by object size - The count of LODs is related to object size.

Notices:

- If mode is “manual”, is not possible to set option min diff, error range and camera height angle in the advanced dialog. These parameters are deactivated, so they have no affect on a result. The other options work normally.
- Slider with “quality” label mean: how many levels will be generated for this VRML primitive, with only two exceptions:
  quality 0: Empty node will be generated.
  quality 1: No levels of details will be generated.

Vertex clustering - “Vertex clustering” algorithm is used for mesh simplification.

Vertex decimation - “Vertex decimation” algorithm is used for mesh simplification.
A.3.4 Algorithms settings

This dialog can be shown by clicking on the button configuration next to the radio buttons Vertex clustering and Vertex decimation, which opens the dialog depend on used algorithm.

**Vertex clustering**

If LOD is creating with this algorithm, degree of simplification could be controllable only indirectly - with “quantization” parameters (Figure A.11):

- **Error area**
  
  default value: 8

- **Error volume**
  
  default value: 1

- **Weighted mean**
  
  default value: 0

The LODCreator uses three algorithms for creation levels of details - error area, error volume and weighted mean. With this three parameters it is possible to change the default weights. The error area algorithm tries to minimize the change in the surface area of the involved polygons (this is the best algorithm). The error volume algorithm tries to minimize the volume which is calculated as the sum of tetraeders (the base areas are the involved faces, the tops are the selected representatives). The weighted mean algorithm simply calculates the deviation to the weighted mean of the points (the weight is the count of points for which a point is already the representative). For IndexedLineSet nodes the weighted mean algorithm is always used. For Sphere, Cylinder, Cone and Cube are not used these algorithms. Detail description is in Section “LODCreator - implementation” on page 20.

- **Min diff (percentage)**
  
  default value: 20

This option specify the minimal difference of points between two LODs
in percent. This does not mean that the difference is exactly equal to this value - the difference can be much higher. e.g. if at first LOD uses 100 points, the next LOD uses a maximum of 80 points.

**Quad split angle (radians)** default value: $\pi/18$ (10 degree)

If a quadrangle is too much distorted in 3D it will be split into two triangles (see also option *no quads*). A quadrangle will be split if one of the angles among the normals (calculated by the program) of the vertices in opposite side is greater than the quad split angle. The action will be performed so that the angle between the two new triangles is minimal. If this parameter is not specified a default value 0.174533 (10 degree) will be used.

**No lines** While generating LODs, faces can collapse into lines. If such a line is equal to an edge of a face or the line is equal to a line already in the set, the line will be eliminated. For each LOD the remaining lines will be written out into an extra *IndexedLineSet* node. If this parameter is specified all lines will be eliminated. Note: this parameter has absolutely nothing to do with generating LODs of *IndexedLineSet* nodes.

**No quads** Normally quadrangles will be split into triangles only if they are not convex or too much distorted in 3D (see option *quad split angle*). If this

![Figure A.10: LoD dialog.](image)

Figure A.10: LoD dialog.
parameter is specified, quadrangles will always be split into two triangles. Polygons with more than four vertices will always be split into triangles which are not dependent on this parameter.

**Vertex decimation**

If LOD is creating with this algorithm (Figure A.12), degree of simplification is controlled by “Decimation step” slider. For example if decimation step is 10% then 10 LODs will be created and first LOD will have 95% vertices of original, second 90%,... If value is 15%, then 6 LODs will be created, etc. Option “Preserve feature edges” turns on/off the preservation of feature edges.
A.3.5 The messages dialog

This dialog prints out messages form LODCreator (Figure A.13).

![Messages window](Image)

**Figure A.13: Messages window.**

**All** - All messages are printed here.

**Messages** - Print out the informative messages.

**Warnings** - Print out the warnings.

**Errors** - Print out the error messages.

**Internal messages** - Print debug information. For example, if somewhere is nil pointer and has not be here.

Check-box “verbose” turn on/off printing detailed messages:

- **Coord** - Print information related to Coordinate3 nodes.
- **Material** - Print information related to Material nodes.
- **Normal** - Print information related to Normal nodes.
- **TexCoord** - Print information related to TextureCoordinate2 nodes.
- **Face** - Print information related to IndexedFaceSet nodes including LOD generation.
- **Line** - Print information related to IndexedLineSet node including LOD generation.
A.3 The dialogs

Join - Print information about joining nodes process.

Triangulator - Print information related to the triangulator (used for polygons with more than four vertices).

A.3.6 Edit dialog

This small editor provides some essential functions for working with *.wrl files like undo, cut, copy, paste, find, replace, etc. Note: be careful, LODCreator parses files directly from harddisk and if file has been edited - in this case, file has to be saved before creating LOD!

![Image of Edit dialog](image)

Figure A.14: Edit dialog.

A.3.7 Preferences dialog

This dialog (Figure A.15) is intended to set properties of the output file.

Float values per line This option specifies the maximum count of floating point values per line. Per default a count of 6 is used.

Integer values per line This option specifies the maximum number of integer values per line in the output file. By default a count of 8 is used.

Extract Used for extracting a specified LOD. Only LODs with this number will be printed. If this parameter is specified the program does not create VRML LOD nodes. Only the IndexedFaceSet, respectively IndexedLineSet node of the specified LOD number will be printed. If a node does not have a LOD with this number (not enough LODs) nothing will be printed. Value 0 is
the original data and if value is -1, this feature is disabled (VRML file is only formatted and cleaned).

**OGL Performer compatible export** If this option is enabled, the LODCreator write the output file in compatible format for OpenGL Performer system used by eSZett. Files in this format are different in some details. Further information is in Chapter 4 “LODCreator - implementation”.

**Rational Reducer compatible export** If this option is enabled, the LODCreator write the output file in format compatible with Rational Reducer.

**Error range (percentage)** default value: 2
This parameter affects the calculation of the ranges of the LODs for switching. This parameter specify the maximum deferral of the points on screen in percent of the height of the viewing window. With a high error range value LOD switching is performed at lesser distances but the image quality is lower. See quick tutorial on page 81.

**Join nodes** The program tries to join nodes. This is usefull if an object consists of many IndexedFaceSets with only few faces inside. Without joining the LOD-generator can not produce proper LODs and the objects tend to
get holes. On the other side separate IndexedFaceSet nodes have separate LODs with separate ranges for switching.

**Camera height angle (radians)**

default value: \( \pi / 4 \)

This angle is needed for calculating the ranges of the LODs. This tool takes the first perspective camera found in the input file. If no camera was found in file, this value is taken. Further information is in Chapter 4 “LODCreator - implementation”.

**No tabs** No indentation of the output is made. Files are smaller.

**Minimize output files** Completely minimize written files. Files are more smaller, but become worse readable (without format).
Appendix B

Quick tutorial

B.1 Loading file

To load a VRML file, you can:

• Press the toolbar button.

• Choose Open in File menu.

These two options will launch “Open” dialog window. Select necessary VRML file (extension .WRL) and press the “Open” button. The file will be loaded shortly and you will see it in main window. For example, load file cow.wrl from the install dir LODCreator. (We will work next with file cow.wrl).

After your file was loaded, you can specify its new name with item “File” → “Save scene as”, but it is not necessary, LODCreator will ask you for a new name after you press the button “Make lod” or “Parse!”.

B.2 Set up the LOD

After the file has been loaded, it will appear automatically in the main window, and you will see a line in the file list frame showing the file name and size. For detection the number of VRML primitives is necessary to click the toolbar button “Parse!”.

To open the “LoD dialog”, you can:

• Push the right mouse button on “main window” (Figure A.6) and choose the item “LoD dialog”.
  
  Note: status bar (Figure A.7) shows which file(s) has been selected.

• Choose “LoD dialog” from the “Window” menu.

There are two possibilities how to create LOD:
B.2 Set up the LOD

**Manual setting** In this mode are disabled these options: *Min diff, error range* and *camera height angle* in “Vertex clustering” dialog and *Decimation step* slider in “Vertex decimation” dialog. The number of LOD is possible to set directly by slider and values for ranges are taken from edit boxes below.

**Full automatically process** In this case LODCreator control the creation process, you can set only few parameters, see Figure A.3.4 - “Vertex clustering” dialog and Figure A.12 “Vertex decimation” dialog.

### Vertex clustering

If LOD is created with this algorithm, degree of simplification could be controllable only indirectly - with “quantization” parameters (Figure A.11). The most interesting options are “*min diff*” and “*error range*” (Figure A.15). How to parameters “*error range*” and “*min diff*” affect result show tables below:

Default value for mindiff is 20 %, that mean: if first LOD uses 1890 points, then the next LOD uses a maximum of 1512 points (Table B.1).

<table>
<thead>
<tr>
<th>Lod</th>
<th>$l_0$</th>
<th>$l_1$</th>
<th>$l_2$</th>
<th>$l_3$</th>
<th>$l_4$</th>
<th>$l_5$</th>
<th>$l_6$</th>
<th>$l_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangles</td>
<td>3778</td>
<td>3033</td>
<td>1715</td>
<td>544</td>
<td>160</td>
<td>30</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>Vertices($l_x$)</td>
<td>1890</td>
<td>1512</td>
<td>839</td>
<td>272</td>
<td>81</td>
<td>18</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>Vertices($l_{x-1}$)-20%</td>
<td>—</td>
<td>1512</td>
<td>1210</td>
<td>671</td>
<td>218</td>
<td>65</td>
<td>14</td>
<td>6</td>
</tr>
</tbody>
</table>

Table B.1: Min diff=20.

<table>
<thead>
<tr>
<th>Lod</th>
<th>$l_0$</th>
<th>$l_1$</th>
<th>$l_2$</th>
<th>$l_3$</th>
<th>$l_4$</th>
<th>$l_5$</th>
<th>$l_6$</th>
<th>$l_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangles</td>
<td>3778</td>
<td>1915</td>
<td>949</td>
<td>467</td>
<td>160</td>
<td>30</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>Vertices($l_x$)</td>
<td>1890</td>
<td>942</td>
<td>467</td>
<td>233</td>
<td>81</td>
<td>18</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>Vertices($l_{x-1}$)-50%</td>
<td>—</td>
<td>945</td>
<td>471</td>
<td>234</td>
<td>117</td>
<td>41</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

Table B.2: Min diff=50.

Example, *mindiff* = 20%:

- The LODCreator begins with the original object (cow.wrl, which contains 3778 triangles and 1890 vertices), this object is generated as $l_0$.

- Next, program has to generate the next level of detail ($l_1$). Value min diff is 20%, so maximum of points in the next level of detail is 1512. LODCreator generates object exactly with 1512 points.
B.3 CREATING LOD

<table>
<thead>
<tr>
<th>Lod</th>
<th>$l_0$</th>
<th>$l_1$</th>
<th>$l_2$</th>
<th>$l_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangles</td>
<td>3778</td>
<td>368</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>Vertices($l_x$)</td>
<td>1890</td>
<td>187</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>Vertices($l_{x-1}$)-90%</td>
<td>—</td>
<td>189</td>
<td>19</td>
<td>1</td>
</tr>
</tbody>
</table>

Table B.3: Min diff=90.

<table>
<thead>
<tr>
<th>Lod</th>
<th>$l_0$</th>
<th>$l_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangles</td>
<td>3778</td>
<td>0</td>
</tr>
<tr>
<td>Vertices($l_x$)</td>
<td>1890</td>
<td>0</td>
</tr>
<tr>
<td>Vertices($l_{x-1}$)-100%</td>
<td>—</td>
<td>0</td>
</tr>
</tbody>
</table>

Table B.4: Min diff=100.

- In the next step ($l_2$), LoDCreator can generates maximum of 1210 vertices, but generates object with 821 vertices.
- Next value ($l_3$) has to be lesser than 671. LODCreator generates object with 218 vertices, etc.
- Next tables shows situation with values 50%, 90% and 100%. With value min diff = 100% LODCreator generates only two nodes - $l_0$ (original object) and $l_1$ (empty node).

With option “error range” is possible to affect ranges for switching the LODs. If value is greater, LOD switching is performed at lesser distances, see Table B.5:

<table>
<thead>
<tr>
<th>Error range</th>
<th>$l_0$</th>
<th>$l_1$</th>
<th>$l_2$</th>
<th>$l_3$</th>
<th>$l_4$</th>
<th>$l_5$</th>
<th>$l_6$</th>
<th>$l_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1%</td>
<td>0m</td>
<td>33.9m</td>
<td>66.5m</td>
<td>131.9m</td>
<td>262.5m</td>
<td>523.8m</td>
<td>1046m</td>
<td>2091m</td>
</tr>
<tr>
<td>2%</td>
<td>0m</td>
<td>2.83m</td>
<td>4.46m</td>
<td>7.73m</td>
<td>14.24m</td>
<td>27.27m</td>
<td>53.24m</td>
<td>105.3m</td>
</tr>
<tr>
<td>10%</td>
<td>0m</td>
<td>1.52m</td>
<td>1.85m</td>
<td>2.50m</td>
<td>3.788m</td>
<td>6.365m</td>
<td>11.42m</td>
<td>21.71m</td>
</tr>
<tr>
<td>90%</td>
<td>0m</td>
<td>2.50m</td>
<td>6.36m</td>
<td>21.9m</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>100%</td>
<td>0m</td>
<td>22.1m</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Table B.5: The situation with error range=0.1%, 2%, 10%, 90% and 100%.
B.3 Creating LOD

To start LOD creation process, you can perform one of the following actions:

- Press toolbar button *Make lod*.
- Select *Make lod* from the *Action* menu.

New file with LOD nodes will be created.

**Minimize the output filesize**

Sometimes (especially if the *lod-quality* slider was set to high value) can files increase in size more than two times. In this case is recommend minimize file size by options in the *preferences* dialog (Figure A.15):

- Parameters *float values per line* and *integer values per line* define the number of values written on each line. Greater values can reduce final file size.
- Option *no tabs* turn off printing the indentations.
- Option *minimize output file size* switch off text formatting.
Bibliography


