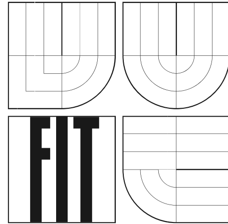
BRNO UNIVERSITY OF TECHNOLOGY

FACULTY OF INFORMATION TECHNOLOGY



3ds file format loading library

MSc. Thesis

2006 Jaroslav Přibyl

3ds file format loading library

Submitted in partial fulfillment of the requirements for MSc. Thesis apology at
27. February 2006

© Jaroslav Přibyl, 2006

The author thereby grants to Brno University of Technology Faculty of Information Technology persmission to reproduce and distribute copies of this thesis document in whole or in part.

Statutory Declaration

I hereby declare that the thesis has been written by myself under the supervision of Ing. Jan Pečiva. I also declare that, as required by these rules and conduct, I have fully cited and referenced all materials and results that are not original to this wok.

....
Jaroslav Přibyl
27. February 2006

Abstract

This paper presents 3ds file format analysis results and describes the process that leads towards them. We used Open Inventor library to write convertor from 3ds file format to Open Inventor file format iv. In fact this MSc. Thesis offers you complex analysis and detailed view to 3ds file format contents.

Keywords

Open Inventor, Coin3D, 3D Studio, 3ds file format, 3ds loading library, smoothing groups, bump mapping, keyframe, hierarchic structure, scale bug, two sided lightning, back face culling, testing, analysis

Acknowledgements

Na tomto místě bych chtěl poděkovat především svým rodičům za jejich podporu při studiu a také panu Ing. Janu Pečivovi za jeho odbornou pomoc při řešení této diplomové práce.

Abstrakt

Tento dokument předkládá výsledky získané na základě analýzy 3ds file formátu a naznačuje cestu, kterou byly dosaženy. V projektu je využívána knihovna Open Inventor. S její pomocí jsme napsali převodník z formátu 3ds do formátu iv. Tato práce tedy nabízí komplexní analýzu a detailní pohled na 3ds file formát, včetně softwarové podpory.

Klicova slova

Open Inventor, Coin3D, 3D Studio, 3ds souborový formát, 3ds knihovna, smoothing groups, bump textury, klíčové snímky, hierarchická struktura, scale bug, two sided lightning, back face culling, testování, analýza

Contents

Contents 6

1 Introduction 8

2 Theory 9
- 2.1 Virtual scenes rendering . . . 9
- 2.2 Scene's geometry . . . 9
  - 2.2.1 Base object's geometry . . . 9
  - 2.2.2 Computing normals . . . 10
- 2.3 Materials and Lights . . . 11
- 2.4 3ds file format structure . . . 11

3 Resources for realization 14
- 3.1 3D Studio MAX . . . 14
- 3.2 Open Inventor . . . 14
- 3.3 Lib3ds library . . . 14

4 Convertor implementation 15
- 4.1 Program structure . . . 15
- 4.2 Object's geometry . . . 15
- 4.3 Computing normals with smooth groups . . . 16
- 4.4 Creating Open Inventor scene graph . . . 18
  - 4.4.1 The generall scene graph . . . 18
  - 4.4.2 Application transformations . . . 18
  - 4.4.3 Application materials . . . 19
  - 4.4.4 Application textures . . . 20

5 Library implementation 22
- 5.1 Passing through 3ds file . . . 22
- 5.2 Loading object's geometry . . . 23
- 5.3 Loading materials . . . 24
- 5.4 Loading keyframe part . . . 24

6 Analysis and solutions 27
- 6.1 Fundamental principles . . . 27
- 6.2 3D Studio export principles . . . 27
- 6.3 Object as an instance . . . 29
- 6.4 Hierarchic scene construction . . . 29

6.4.1 Finding the independent matrices 32
6.4.2 Transforming vertices 34

7 Testing **35**
7.1 Scale bug in 3ds file 35
7.2 Translation to pivot point 36
7.3 Hierarchy system and instances 36
7.4 Texture optimalization 37
7.5 Where is the truth 37

8 Conclusion **38**

List of Figures **40**

A API for the Lib3ds library **41**

B Howto using the 3ds2iv **42**

C Companion cdrom **43**

Chapter 1

Introduction

Preliminary I would like to denote short chracteristics of our problem in wider context and describe contents of each chapter.

Virtual scenes rendering has been a complex problem. With respect to different complexity, we'll try to describe just some parts of this process. The main objective for this project is to create convertor from the 3D Studio 3ds file format to the Open Inventor iv file format. In conjunction with this task, we make deep 3ds file format analysis. We need this analysis so as a good convertor could be written. Well this is the second objective for our work. The last task for this project was to write the 3ds file loading library lib3ds. We need this library to obtain information about scene from a 3ds binary file. This library also transfrom binary data from a 3ds file into data structures witch are simpler to use.

The 3D Studio is one of the best software for modeling and consequently rendering virtual scenes, simple or complex objects and animation. We can export scene from the 3D Studio into many formats. The important format for us will be the 3ds file format. But there is a lot of limitations and hidden processes during exporting a scene into the 3ds file format. Our analysis attempt to explain these limitations and processes.

There are some limitations in the Open Inventor too. But the Open Inventor hadn't been primary software for virtual scenes rendering as the 3D Studio can be. It isn't our goal to rendering scenes look like real virtual scenes with shadows, light effects etc. Our goal is to reconstruct scene from a 3ds file based on geometry, materials and partial also on animation. In spite of these limitations the Open Inventor and the 3ds file format are enough for our purposes.

In the next chapters we'll continue from our year and term projects results. These results are a fundamental source for whole 3ds file format analysis and this thesis. There we'll presents these results and show the way how we got them. Next we'll describe project's resources including the Open Inventor or the 3D Studio MAX and describe implementation of the 3ds2iv convertor and the lib3ds library. We make detailed analysis of all wrong loaded models and try to load them right. Finally we show the results of wrong models analysis, describe the 3ds2iv convertor features and show possibilities for future work.

Chapter 2

Theory

In this chapter we learn the 3D graphics fundamentals such as geometry, normals and materials, that are need for successful solving this project. Next this chapter will include short 3ds file format description and the main principles of this format.

2.1 Virtual scenes rendering

Virtual scenes rendering is generally a term of wide comprehension and we can see on it in several ways. You have to wise up the rendering scene will be used for many application. The output from the rendering program will be different for a realistic photography and ie. for fast game's objects rendering. It is clearly to see that for a realistic photography or animation frame we want best realistic appearance. On the other hand for fast game rendering we want best performance expressed by frames per second (FPS).

The virtual scene rendering is compound of several important parts. So next paragraph will include their description.

2.2 Scene's geometry

The first necessary step for create virtual scene is to create object's geometry. There are many modelling techniques which may be useful for this purpose. Each technique is used for creating specific objects. In example for modelling a table we can use a technique named box modelling. On the other hand for modelling a glass we can use splines. The right way selection of modelling technique is important because of time needed for modelling object's form and because of the number of triangles from witch is the object compound. If there is load of triagles in virtual scene, then heftiness of the scene grow up and scene is going to less applicable. Practically we are offen obliged to make compromises between scene's complexity and scene's dynamism. There has been also possibility how to improve the scene's dynamism with highly detailed models at the same time. One of these methods is level of detail (LOD)[4]. This method is based on triangles count reduction for models, that are further from camera.

2.2.1 Base object's geometry

Each virtual scene is composed from many various objects. But each of them is composed from set of triangles. If we put these triangles together, then we obtain object's body (surface). This

surface composed from triangles is marked as mesh. However there are methods for change object's appearance unlike redefinition the triagle set. First method for changing object's appearance is mesh shading. Choosing the right object's shading we can obtain smooth objects appearance. Objects shading will be described in next subsection. The second method intended for change object's appearance is bump mapping[2, 11]. This method is based on getting normals from normal map. Each pixel in scene has defined own normal. Bump map principles will be described later. Essentially there was the normal computing problem ever. We have to compute normals for each object in virtual scene.

Mesh is composed from triangles which is named faces. As you know each triangle has three vertices, but there is the vertices ordering very important, because it defines all normals orientation. Then the orientation of each normal will be important for computing visibility and lightning in scene.

2.2.2 Computing normals

We can compute object's normals in several ways. The end result of object's appearance is different for each used method.

No shading

For this shading method there are no one normals for computing. This shading style causes 2D scene's appearance. There are a same color for all triangles which haven't defined any shading style. Generally we can define color in each vertex of triangle. Color for all other triangle's pixels is defined by interpolation. For no shading style we can recognize only outline object's edges, because there are no inside edges visible, because color of all triangle's pixels are the same.

Flat shading

If we need to get hight frame per rate for visualised scene, we can use constant shading (flat shading) method. This shading method can be used for working models or for preview models. There is for each triangle computed one normal. We can mark this as a term "one normal per face". There are important the vertex numbering, because the normal direction. For computing face's normal we can use equation marked as a cross product:

$$\vec{n} = \vec{u} \times \vec{v} = \hat{n_x}(u_y \cdot v_z - u_z \cdot v_y) + \hat{n_y}(u_z \cdot v_x - u_x \cdot v_z) + \hat{n_z}(u_x \cdot v_y - u_y \cdot v_x)$$

$$\text{where } \vec{u} = V_2 - V_1 = (u_x, u_y, u_z) \text{ and } \vec{v} = V_3 - V_1 = (v_x, v_y, v_z)$$

The computed vector $\vec{n}$ represents perpendicular to plane represened by vectors $\vec{u}$ and $\vec{v}$.

Smooth shading (Gouraud shading)

This shading style is used for simulation smooth passing between two faces. So there isn't visible edge between triangles, in contrast to flat shading, but the color isn't the same for all pixels, in constrast with No Shading method. There is computed three normals one for each vertex for this shading type. The vertex normal is computed as arithmetic average of all face's normals which this vertex belongs to. There is in practise some cases when we want, that some edges between faces could be visible. Such case can be in example a cylinder. There we want the edge between cylinder base and lateral surface of cylinder is visible. Well for right computing the cylinder base's vertex normals we shall use only base faces and for right computing vertex normals for the lateral surface of cylinder we shall use only lateral surface faces.

Phong shading

This shading method offers to us better visual results than Gouraud shading. But the difference indicating price between visual quality and computational requirements is much higher than requirements for previous three methods. Phong shading method is based on interpolation of normal vectors. The normals are interpolated at the same time with the surface rasterization is doing. Based ond this principle is determining color shade for each pixel. The difference between Gouraud and Phong shading was detailed described in [8].

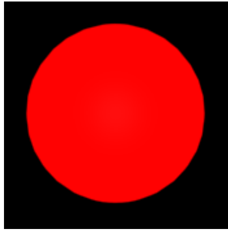


Figure 2.1: No Shading

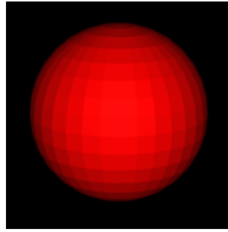


Figure 2.2: Flat shading

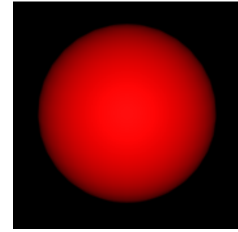


Figure 2.3: Smooth shading

2.3 Materials and Lights

The next important step in virtual scene rendering process is definition of parameters for materials. Parameters for materials closely relating to lighting model and to optical surface properties as are color, glossines and so on. Selected lightning model determine resulting quality for rendered virtual scene.

There is the object's color determined based on reflective surface characteristics in real world and based on amount of incident radiation of light too. In computer graphics we often use light model reduced to three base elements of spectrum, because modeling material's characteristics for whole light spectrum is very difficult. The three base elements of light are ambient, diffuse and specular color. Ambient light represents omnidirectional light. This light is comming from all directions and is same in all point in scene. Diffuse light depends on angle of fall. After fall on object's surface is dispersed to the all directions. Also this light element is independent on point of view. Diffusily reflected light brings information about surface color to the observer. Specular element of light represents light, that after falling on the surface is reflected to only one direction. His chracteristic attribute is directionality. When the surface is more glassy, then directionality increases.

2.4 3ds file format structure

The main 3ds file format structure is quite simple. The file is composed from binary data blocks. These blocks are marked as *chunks*. Each chunk has a header that contains two items. The first item we'll mark as *chunk identifier* (chunk id). This item identify the data block type. It can be mesh, material, keyframe and so on. The second item defines the data block size including header and all *sub-chunks*. Each chunk in 3ds file has this pair of information, therefore *chunk id* and *chunk size*.

There is hierarchical data structure in 3ds file format. It means that some chunks are able to have next sub-chunks. Sub-chunk data block has the same data format as a chunk. It means they have a header and a body. A chunk data block can be a parent for defined number of sub-chunks.

Generally the chunks and sub-chunks can including end data, but they can including a next sub-chunks too. How we are allredy wrote each chunk and each sub-chunk offer us two important information. So we define their sizes in bytes now. The chunk identifier is two bytes length. The chunk size identifier is four byte length. Whole header size is also six bytes length. The chunk body and sub-chunk body length is also defined by chunk size identifier. Generall 3ds file structure could look like in this example.

```
chunk 1
    sub-chunk 1
        data 1
        data 2
        data 3
        sub-chunk 3
            data 1
            data 2
    sub-chunk 2
        data 1
        data 2
        data 3
chunk 2
    data 1
    data 2
  .
  .
  .
```

If you can get data from sub-chunk number two, you must first read header from the sub-chunk number one. There you can find the sub-chunk number one's size identifier (four bytes) in header. Then you can jump to the header of sub-chunk number two, because knowing the sub-chunk number one size. The advantage in this approach is that you generally didn't have to know the chunk or sub-chunk data structure. If you don't need to perform some chunks or some sub-chunks, you can simply skip them.

The real base 3ds file format structure is shown bellow in the next page. First there is defined MAIN CHUNK determinings the 3ds file. The number 0x4D4D is chunk id. This is the first parameter included in each header. The second parameter cannot be written here, because the size depend on concrete 3ds file contents. Next the 3D EDITOR CHUNK determine data and objects created in 3D Studio editor including materials and animation. The name 3D EDITOR is remainder from Autodesk 3D Studio Release for DOS. There was 2D Shaper, 3D Lofter, 3D Editor, Keyframer and Material editor in this old 3D Studio software.

The next important chunk in each 3ds file is OBJECT CHUNK. This chunk is parent for all physical objects created in 3D Studio. The most important sub-chunk are TRIANGULAR MESH, which including whole scene geometry. There is defined MATERIAL CHUNK too. This MATERIAL CHUNK contains definition for all materials created and used in scene. The last usefull chunk for us is the KEYFRAME CHUNK. This chunk contains information about geometric transformations for all objects. Lately we'll use this transformations for correct some 3ds file format mistakes. This chunk also contains information about all animation key frames.

So if we want to create the lib3ds library, we should know all the neccessary chunks in 3ds format. This chunk's official description we can get from Autodesk company or acctually from this [3] internet adress.

```
MAIN CHUNK 0x4D4D
  3D EDITOR CHUNK 0x3D3D
    OBJECT CHUNK 0x4000
      TRIANGULAR MESH 0x4100
        VERTICES LIST 0x4110
        FACES DESCRIPTION 0x4120
        MAPPING COORDINATES LIST 0x4140
          SMOOTHING GROUP LIST 0x4150
        LOCAL COORDINATES SYSTEM 0x4160
      LIGHT 0x4600
        SPOTLIGHT 0x4610
      CAMERA 0x4700
    MATERIAL CHUNK 0xAFFF
      MATERIAL NAME 0xA000
      AMBIENT COLOR 0xA010
      DIFFUSE COLOR 0xA020
      SPECULAR COLOR 0xA030
      TEXTURE MAP 0xA200
      BUMP MAP 0xA230
    KEYFRAMER CHUNK 0xB000
      MESH INFORMATION BLOCK 0xB002
      FRAMES (START AND END) 0xB008
        OBJECT NAME 0xB010
        OBJECT PIVOT POINT 0xB013
        POSITION TRACK 0xB020
        ROTATION TRACK 0xB021
        SCALE TRACK 0xB022
        HIERARCHY POSITION 0xB030
```

Chapter 3

Resources for realization

3.1 3D Studio MAX

3D Studio MAX[9, 13] is one of the best sofware for modeling and consequently rendering virtual scenes, simple or complex objects and virtual scene animation. The 3D Studio is most important for our task, because a created scene in this software can be exported into 3ds file format. We'll derive benefit from this and we'll create small models because the 3ds analysis. With this small models we are able to make deep and exact analysis. We analyse some large objects too, but we'll not create them. We'll get some models from internet and other sources and find models that propably contains some problems. You can be sure that such models exists and there are swarms of them. There is important for us the animation part (keyframe) in 3D Studio. All of wrong loaded models we'll try to analyse and then improve the 3ds2iv convertor written as part of the last year project.

3.2 Open Inventor

Open Inventor is object oriented library for 3D graphics visualization developed by Silicon graphics (SGI)[5]. This library offers us the solution for programming interactive graphics application. The programming model is based on constructing scene graph[12]. Programming application with this library is dramatically simpler, because the programming model. There exists a lot of open source Open Inventor implementations. We'll use the Coin3D[1] implementation of Open Inventor, which is fully compatible with SGI Open Inventor 2.1. Open Inventor is the standard for 3D visualization and simulation software for scientific and engineering community today. More about Open Inventor you can get from books[6, 7] or from internet[10].

3.3 Lib3ds library

To create this library is one of the main goals for this thesis. Our lib3ds library will appear from the 3d studio file toolkit library 3dsftk created by Autodesk company and which was modified by Cyberloonies company. This 3dsftk library wasn't released as public domain or similar source code free licence. We'll plan to release this document and 3ds2iv convertor as an open source so we need to write our own open source library for loading 3ds file binary data. So we marked this library as lib3ds. With the help of the lib3ds library we can load information about scene from binary 3ds file. In near future we would like to improve this lib3ds library for better and wider working with loaded data, because not everybody would like to use this library with 3ds2iv convertor together.

Chapter 4

Convertor implementation

4.1 Program structure

There in this chapter we'll describe the fundamentals needed to write 3ds2iv convertor. We won't make any detailed description how to work with Open Inventor. We'll show to you only how to create a scene graph. It isn't the main objective for this thesis to learn working with Open Inventor. On the other hand you'll get knowledge how to interpreting 3ds file's data for creating scene geometry, materials and textures. At first we'll show to you how is the main algorithm constructed. For this purpose we use the following flowchart.

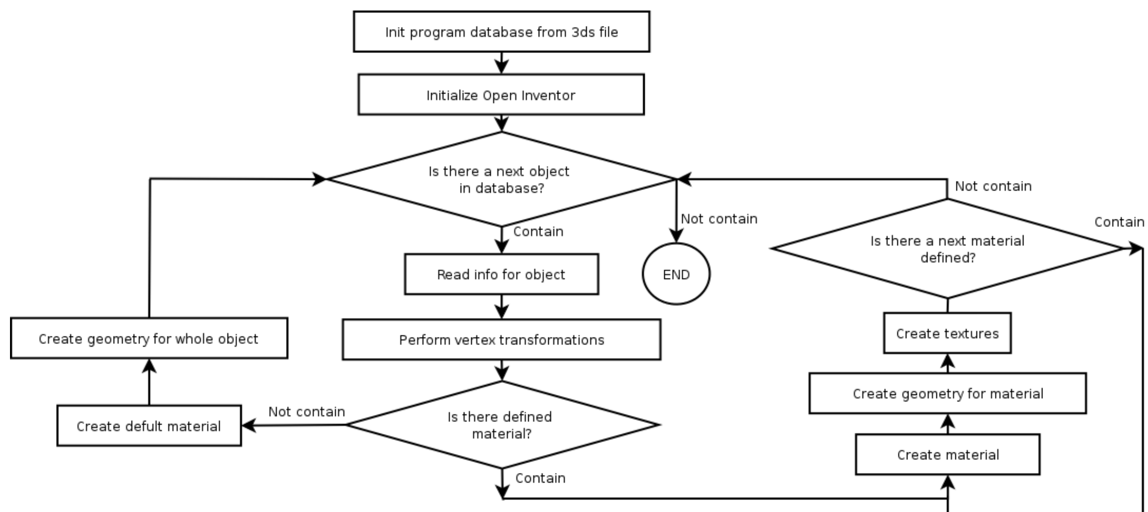


Figure 4.1: Convertor Flow chart

4.2 Object's geometry

If you can create geometry by correct way, you should make more work than inserting vertices and triangles definition into scene. The 3ds file format define a special method for creating smoothly shaded objects. This method bring to us a lot of complications, because the poor documentation of this format. First in the next figure 4.2 we'll show the base 3ds file format content.

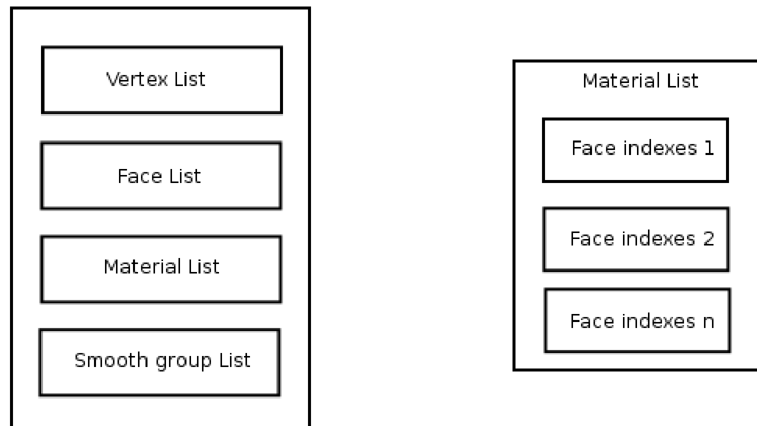


Figure 4.2: Base 3ds file format content for creating geometry

There is only one right way how to create object's geometry. The main chunk in 3ds file is material list as you can se at figure 4.2.

By the way there are also defined vertex list too. Is is clearly to see, that it contain all vertices for one object (mesh).

Next there are defined face list containing all faces definition for actual object. That means for each face there are defined three indexes into vertex array. These three vertices define one triangle. The indexes ordering is important for computing face normal orientation. It could be clocwise or counterclockwise.

The material list section is divided by number of defined materials for a given object. For each material there is defined face index set. This face index set including indexes into face array. That means that each object (mesh) in scene is divided by materials and we have information about faces that belongs to this given materials. On the basis of this knowledge we can create all objects. But if we use only these data for creating scene, we'll get only flat shading models in fact.

Generally there can be each face defined as smooth shaded in 3D Studio. Well how it could be distinguished which faces will be smooth shaded and which faces will be flat shaded? For this purpose we can use the smooth group list. This list containt 32 bits information about smoothing groups for each face. Why there is defined only 32 bits (32 smoothing groups)? Because there was defined in 3D Studio, that we can define up to 32 smooth groups. If some face belongs to at least one smoothing group, it would be rendered by smooth shading. Withal there is possibility to create and apply material with parameter *faceted*. This parameter causes that all faces related to this material will be rendered using the flat shading method. This is applied for face that belongs to some smoothing group too.

Unfortunetely 3ds file format didn't include normals. So if we can use smooth shading style failing from flat shading, we have to compute normals manually with usage of smooth group list. In conjunction with faces definition and division mesh by materials is more difficult to compute normals. How we can compute normals we show to you in the next section.

4.3 Computing normals with smooth groups

We wrote the general concepts for computing normals in the subsection 2.2.2. But this is only generally principles. Now we try to explain how to compute normals for smooth shading practically.

First we have to notify how to computing normals for flat shading. This can be done more easilly than smooth shading. We'll just know only vertex coordiantes and vertex ordering. In this case smooth groups isn't important for us. Thus if we create the Open Inventor scene graph, we only compute the normals by this following way. We have to write the loop over the defined materials. Then for each material we take a look into face index set for this given material. There is defined component faces belong to this given material. Well we know the faces indexes. Now we can simply take indexes from face index set and based on this indexes we can found faces definition in face list. At this moment it is simply to compute the face normal, because the vertices coordinates is known. This procedure is described with using the pseudocode written bellow.

```
for (i = 0; i != number of defined materials; i++) {
  getFaceIndexSetFromMaterialList(i);
  for (j = 0; j != number of indexes in face index set; j++) {
    getFaceFromFaceList(j);
    computeFaceNormal();
  }
}
```

So we know how to compute the face normals. Now we can begin to explain how to compute the vertex normals (smooth shading). We'll use the smooth groups list. This list is defined once for each object (mesh). We have to define the algorithm at first. The base for this algorithm is diferent compared to the flat shading algorithm.

First we need know for each vertex in which faces belongs, because for compute vertex normal we need all faces normals that belong to this given vertex. So we have to create auxiliary array that contain reverse information compared to face array. Then if we select some vertex from this array, we'll get a list of faces in which this given vertex belongs. This auxiliary array contain more than face indexes list. For each face index is there attached smoothing group identifier. Through the use of this identifier we can determine the smoothing groups in which a given face belong. Now we can compute the vertex normals.

```
createAuxiliaryArray()
DivideFacesBySmoothGroups();
for (all smooth groups - 1 .. 32)  {
  for (j = 0; j != all faces in one of 32 divided faces array; j++)  {
    for (all three vertex indexes for given face)  {
      GetFaceListForGivenVertexFromAuxiliaryArray();
      for (all faces belong to actual smooth group)  {
        ComputeFacesNormals();
      }
      ComputeVertexNormalsFromFacesNormals();
    }
  }
}
```

After we create an auxiliary array described above, we'll create next arrays. We need divide all faces by smoothing groups into separate arrays, because creating the scene graph in Open Inventor and the visible edges between smoothing groups. Then we can compute normals independently on

materials and based on defined smoothing groups. So step by step for smoothing groups we proceed the divided face arrays and compute normals in the following way.

At this moment we proceed ie. the divided array for smooh group number one. Next we proceed all faces in this array. For each vertex for given face we get the faces list from the auxiliary array. With this face list we get the information in which smoothing group individual faces from this list belongs to. So we can select the faces corresponding to the processing smooth group number one. For these selected faces we compute the face normals and from this normals we can compute the vertex normal for a given vertex. This procedure we have to perform for all smooth groups defined for a given mesh.

4.4 Creating Open Inventor scene graph

There in this section we'll describe the Open Inventor scene graph construction. Please dont't expect any instruction how to work with Open Inventor. This section is attempt only to show the process of creating a scene graph. If you can learn more about Open Inventor you can read the book Inventor Mentor[7].

4.4.1 The generall scene graph

The general scene graph in figure 4.3 shown the basic scene construction. You can see that the scene is divided based on defined mesh's objects. Each mesh is composed from vertices represented by SoCoordinate3 node and separator classes named Material. Generally each separator class isn't a terminating object (node) in scene and have to be next expanded. So the title Material 1 and Material n express, that each object is divided based on materials. This way of creating scene is described in section 4.3. Each mesh is divided acoording to number of defined materials for him.

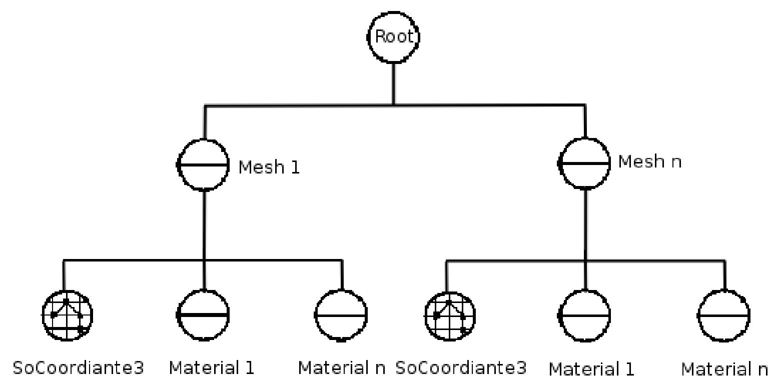


Figure 4.3: The generall scene graph. Scene is composed from vertices nodes and material separator class which will be next expanded

4.4.2 Application transformations

Now we add the transformation node into the scene. This node represents geometric transformations over vertices. This transformations meaning will be explained later. But there is the place in scene graph to put the transformations. Each mesh class is then modified as is shown in figure 4.4.

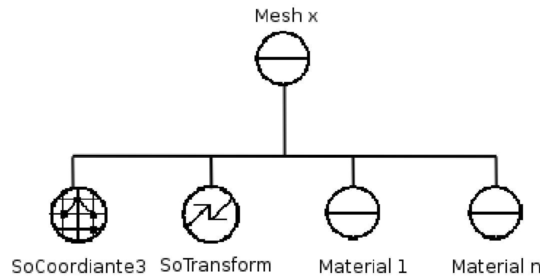


Figure 4.4: We add the SoTransform node into the scene. Transformation is perform over the SoCoordinate3 vertices

4.4.3 Application materials

As we know creating object's geometry depends on number of used materials for this object. At this moment we add to scene graph the material definition and the scene is slowly going to be applicable. Thus each material separator class will be contain material properties nodes and the Normal separator class that will be next expanded. The expanded material look like in figure 4.5.

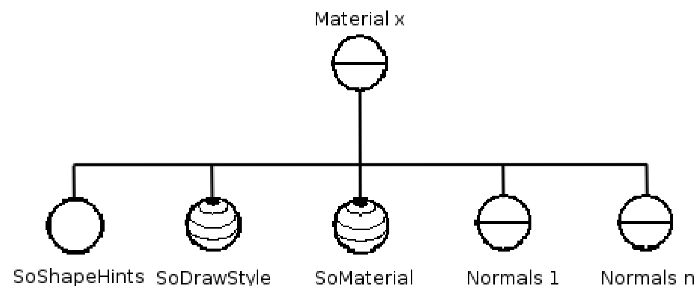


Figure 4.5: We add the material properties nodes and the class separator for normals into scene graph

At least we can expand the normal class. Performing this action we get the more or less functional scene graph. Each Normals class will including three nodes. The first node SoNormalBinding defines way of application normals. We can define normals per vertex (smooth shading) or we can define normal per face (flat shading). It depends on computed normals and used shading for a given object (mesh). The second node SoNormals contains the computed normals. The last node SoIndexedFaceSet contains the faces definition. The number of Normal separator class depends on number of used smoothing groups for a given mesh and material. You can see this scene graph on figure 4.6. If there is used ie. five smoothing groups for a given mesh and material, then there will be five Normals separator classes.

At this moment we try to explain the contents of each Material class. The main node for material definition is the SoMaterial node. This node brings up to the scene settings of basic material attributes such colors, shinniness, transparency etc. If there wasn't defined material in 3ds file, we'll use the self defined default material. This newly created material we'll assign to a given object. If a given object has defined some material in 3ds file, then we'll use this one for settings parameters of the SoMaterial node. The material's definition will be inserted into the scene graph at the moment of creating Material separator class. We can see this on figure 4.5. Next we add into the scene graph

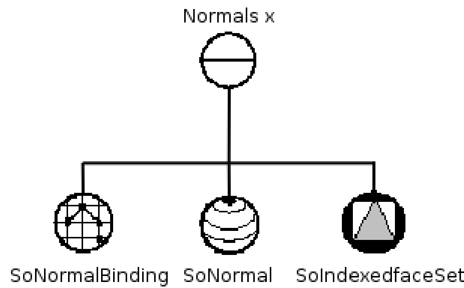


Figure 4.6: We expand the Normals separator class and get the a functional scene graph

two new nodes. The first is SoShapeHints node and the second is SoDrawStyle node. The first SoShapeHints node is used for settings two sided lightning and backface culling properties. For this purpose we can use the ShapeType and vertexOrdering SoShapehints parameters. The second SoDrawStyle node is used for specifying common rendering properties for meshes. The draw style properties can be lines style, points style and filled shape style.

There was discovered that some models contain bug in 3ds file. This bug we can simply describe as negative scale around the x axis. This bug will be explained later in chapter 6. We can solve this problem by turning the object around the x axis. But this operation causes that the model has all faces invisible, because the backfaceculling. So we have to set the vertexOrdering to CLOCKWISE for meshes with this bug.

4.4.4 Application textures

The Open Inventor represented by Coin3D library allows us relatively good working with textures. We can use the SoTexture2 node and additional nodes SoTextureCoordinate2, SoTextureCoordinateBinding and SoTexture2Transform. There has been supported bump mapping since Coin 3D version 2.2. For this feature we can use the SoBumpMap node, SoBumpMapCoordinate node and SoBumpMapTransform nodes. The scene graph with included texture nodes look like at figure 4.7.

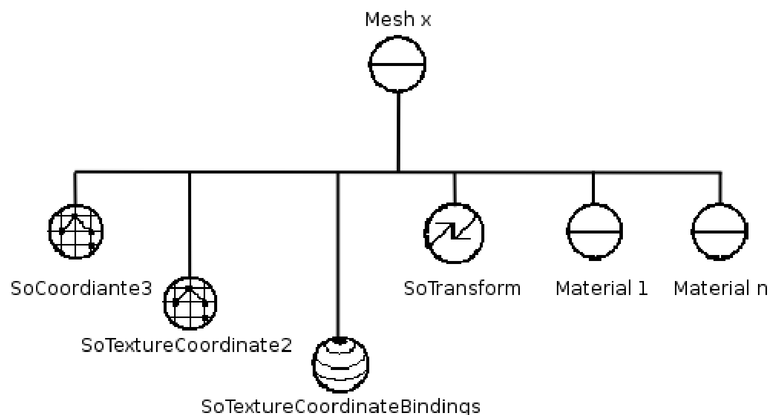


Figure 4.7: Scene graph with included texture's nodes in Mesh class

The last step to get complete scene graph with all needed nodes is shown at figure 4.8. We add to scene in the last paragraph mentioned SoTexture2 and SoTexture2Transform nodes. Then we add the SoBumpMap and SoBumpMapTransform nodes into the scene graph.

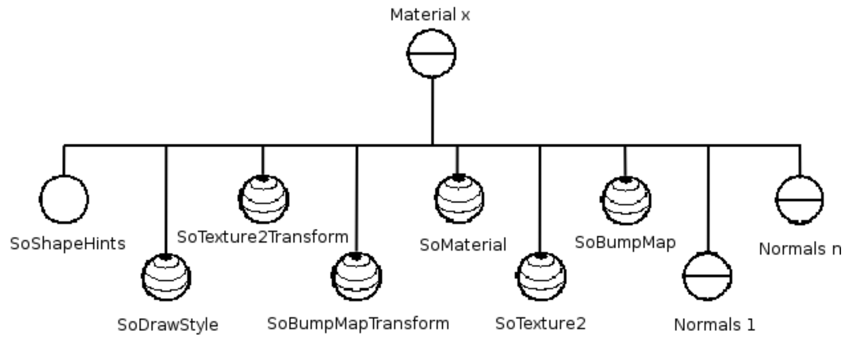


Figure 4.8: Scene with included other needed nodes in Material class

This next figure 4.9 is a bonus and it shows the complete scene graph for one mesh with one defined material. There are a lot of meshes with a lot of defined materials included in scene generally.

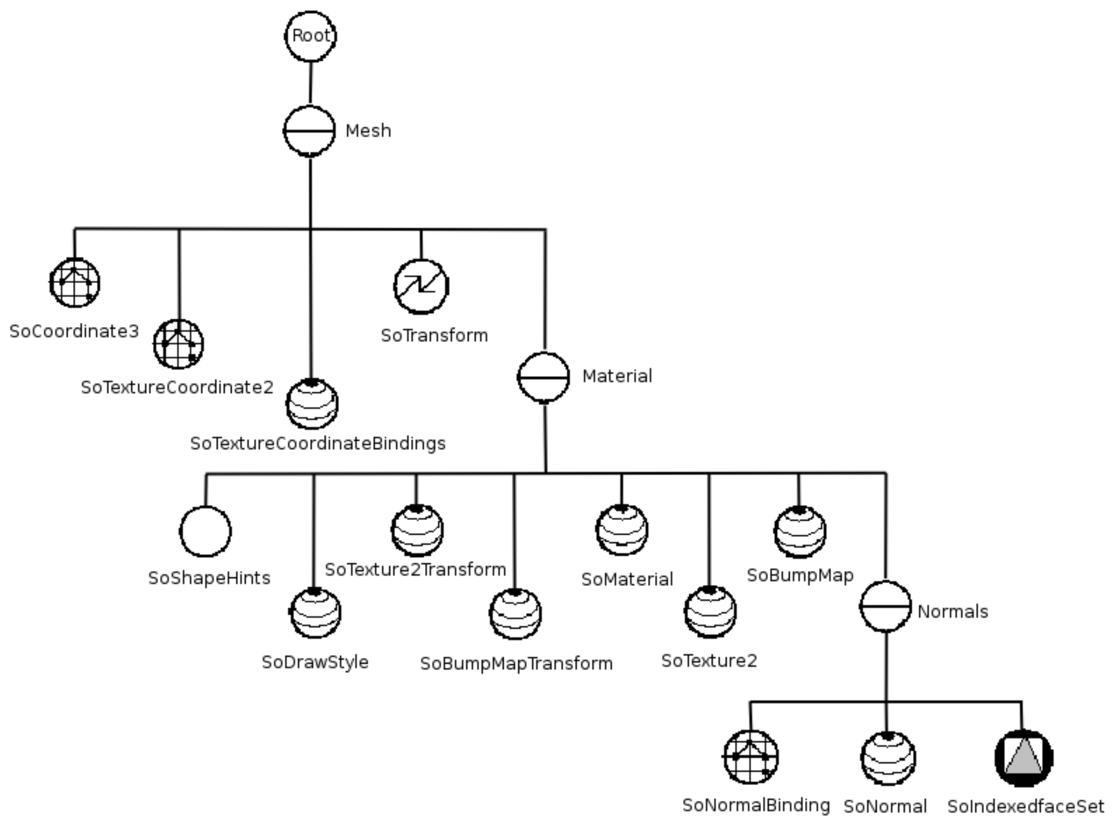


Figure 4.9: The complete scne graph for one mesh and one defined material

Chapter 5

Library implementation

If we began implement this lib3ds library, we would result from the 3dsftk library. The 3dsftk library is more robust, but for our purposes it is unnecessary. We try to preserve the principles of working with this library, but the implementation is more diferent, because our lib3ds library is written for better working with the 3ds2iv convertor.

The library implements all necessary parts for right work with all 3ds files. Before we stared to describe the library implementation, we had to become acquainted with 3ds file format structure. This file format principles was mentioned in section 2.4 and in official 3ds file format documentation.

So the 3ds library implementation can be divided into four separately parts. Next in this chapter we'll try to describe them subsequently.

5.1 Passing through 3ds file

As we wrote in this chapter's introduction, the whole library implementation is divided into four parts. Now we can describe the first part marked as passing through the 3ds file.

The passing through the 3ds file is implemented by reading the chunks and processing their contents. This part implements the library interface represented by functions and data structures.

Before we started to load data from 3ds file, we need to create three empty internal databases for storing the loaded data. We need this databases for better working with data using the library functions. We need informations about complex scene for implement functions provided by library. So first we create the mesh database, material database and at finally the key frame database. These databases initialization provide their conctructors and at this point they are all empty.

Now is the time for passing through the 3ds file. The base idea about passing thourgh 3ds file lies in each chunk's header. The header contain two unique informations the chunk id and the chunk size. You can get detailed info about this two chunks in chapter 2.4. If we know each cnunk's size, we can simply skip this chunk. In expamle we don't need information about geometry flag array. So we read this chunk's header and after it we simply jump along the file to next chunk header by using the chunk size identifier. If we can proceed the given chunk content, we must know the given chunk's specific structure. We can get each chunks description in official 3ds file format documentation from Autodesk. Processing of each chunk's contents is performed by functions implemented by the mesh database, material database and key frame databases.

The pseudocode bellow shows you the base framework for passing thru the 3ds file.

```
// Go thru 3ds file and read all known chunks
while( ftell(inputFile) < fileSize )  {
  // next chunk started on actual position in inputFile
  chunkStart = ftell(inputFile);

  chunkId = read first two bytes, they determining chunk id
  chunkLenght= read next four bytes, they determining chunk lenght

  switch (chunkId)  {
    .
    .
    case 0x4110: meshDatabase->addPointArray(); break;
    case 0x4111: meshDatabase->addPointFlagArray(); break;
    case 0x4120: meshDatabase->addFaceArray(); break;
    .
    .
    case 0xA010: matDatabase->setColorType(ambient); break;
    case 0xA020: matDatabase->setColorType(diffuse); break;
    case 0xA030: matDatabase->setColorType(specular); break;
    .
    .
    case 0xB010: kframeDatabase->addHeader(); break;
    case 0xB013: kframeDatabase->addPivot(); break;
    case 0xB011: kframeDatabase->addInstanceName(); break;
    .
    .
    default: jump(chunkLenght - (CHUNK_ID_LEN + CHUNK_LENGHT_LEN));
  }
}
```

For chunks that aren't enumerated in chunkId switch will be applied the defult rule. This rule cause that the given chunk will be skipped. In this 3ds file passing are called the functions from relevant databases. These databases functions then fill up the data structures which will be used by the library's interface functions.

5.2 Loading object's geometry

First we read from 3ds file an information about object's geometry and we create the list of them. This list contain all names of object's. All objects in 3ds file have own unique name. Based on this name we are able to distinguish each object (mesh).

Well the process of creating mesh database will be like this next. We get the name of current mesh and add its name into the list. Then we setting the default values for all supported mesh properties in our library. Next we read from 3ds file the vertices array for current mesh. But before this reading we must read the number of vertices for this given mesh. Then we can read the exact number of vertices. On the base of this principle we are able to read the next Flags array, the Face array, the MatGroup array, the Smooth array, the Texture vertex array and the Local matrix array. We have to cooperate with the 3ds file format documentation. There are described all chunks and partially their meaning too. The main idea is still to create mesh list and then for each mesh read the usefull data from 3ds file. After creating this list we can write the function for read data for selected mesh from database. We select the mesh using its name. This function is important for our 3ds2iv convertor.

So the first library function for work with meshes is the getMeshList function. This function can return the list of mesh's names included in database. The second function provided by our library is the getMesh function that return data for a given mesh's name. We also can get access for all parameters stored in database for a given mesh. These two function are sufficient for our purposes.

5.3 Loading materials

The principle for loading materials is similar to loading object's geometry. We need to create the list of material's names too. During the 3ds file passing process, we get all the material names and its parameters. So if we get a new material name, we add this material to the database. Than we set to this newly added material the default parameters.

Next we go through the 3ds file and get information about the new material. Based on this information we can modify the default values for material's properties.

There in this part of processing the 3ds file, we proceed the textures too. We can get from the 3ds file the name and path of used texture files. For each texture, we can read a few parameters from 3ds file, which can be usefull.

The library provide two functions for work with materials. The first is getMaterialList. This function return the list of materials used for given object. The second function named getMaterial return data for given name of material. There is defined only distinct material names in 3ds file. So we can use the material name to distinquish between two other materials.

5.4 Loading keyframe part

In this last part we'll load information about animation represented by key frames and information about hierarchic structure between objects in 3ds file.

There can be defined for all distinguished objects (mesh, camera, light, etc.) an animation (hierarchic structure). You can imagine the light connected with camera. If the camera fly throw a scene, we need the light will be shinning to the part of scene where is the camera flying throw. The next example will be animation of robot. If the robot turn his shoulder, we want to turn his forearm too. We can simply define the hierarchic structure in 3D Studio and this will be exposed into the 3ds file. More important than light and camera animation is for us the mesh animation and hierarchic structure. The other objects will be skipped.

Implementation of this part was more difficult, because a litle experience with this part. So the second big complication for us was the bug in 3dsftk library, which was the base line for our lib3ds library. The bug lay in incorrect loading hierarchic structure for models created in old version of 3D Studio. The old 3D Studio means the 3D Studio without MAX annex, so this 3D Studio had been developed from Autodesk and was destinated primary for DOS.

So in new 3D Studio with MAX annex is a hierarchic structure defined by ID number and by distinct name. In old 3D Studio there is hierarchical structure defined only by name and the ordering of keyframe part in 3ds file. So the 3dsftk library propably worked only with ID numbers and distinct names. But the ID didn't defined in old 3ds files and for this reason appear the 3dsftk library bug. Our library solve this 3D Studio version diferences and load object's hierarchy by right way. The situation is better described by using the following example.

```
ChunkOBJECT_NODE_TAG (b002H)      Chunk OBJECT_NODE_TAG (b002H)
  Length is 189 (bdH)               Length is 191 (bfH)

  Chunk NODE_ID (b030H)             Chunk NODE_HDR (b010H)
  Length is 8 (8H)                   Length is 17 (11H)
   Node ID: 0                       Object name: roof
                                    Flags 1: 4020
  Chunk NODE_HDR (b010H)            ATKEY2
   Length is 17 (11H)               ATKEYFLAGS
  Object name: roof                 PRIMARY_NODE
  Flags 1: 4000                     Flags 2: 0
  PRIMARY_NODE                      No Parent
  Flags 2: 0
  No Parent
```

There is on the left side in example the 3D Studio MAX output and on the right size is the old 3D Studio output. You can see that for the old 3D Studio there is missing the NODE_ID chunk. There is the NODE_ID defined as the ordinal number of OBJECT_NODE_TAG chunk in the keyframe part. The parent can be eventually searched base on the ordinal number. In 3D Studio MAX we'll search parent on the base of the correspondence between NODE_ID and the PARENT property node.

The situation about searching object's parent is slightly more complicated and we'll try to explain why. There is possibility to create objects as an instances in 3D Studio generally. These objects have the same geometry as their parents, but they have different position, rotation or scale transformation. In 3ds file these objects have the same name as their parents, but they have an instance name node added. If there is more than one instance for one parent, then the instance name distinguish them. The instance name node distinguish between the parent and the instance too.

We can found definition for the instance objects only in the key frame part. They have defined only the transformations considering to the parent in the key frame part. So if we go throw the list of mesh names using the function getMeshList, we don't get listed the instance objects. For this purpose there are defined two functions. The first is getMeshInstanceCount. This function return for the given mesh the number of its instances. If there isn't any instances then its return zero. The second function for working with instances is getKeyFrameInstanceByIndex. This function return for a given mesh's name and for a given index the indexth instance for a given mesh's name. Also we can use these two function together. There are a thirth function too. This function is named getKeyFrame and returns key frame data for given mesh's name.

There are a special object in 3D Studio marked as Dummy. This object didn't have any geometry. This object is only auxiliary object for working with animation and didn't inherit any geometry. The Dummy object define only some tranformations and it can be a parent to all other objects.

Next we can show to you a small example about instances. Imagine you create in 3D Studio an object marked as Box01. Then you create an object Box02 as an instance of object Box01. And finally you create an object Box03 as instance of object Box02. So the Box01 mesh is the main object and have defined own geometry and transformations in key frame part. In 3ds file is the next mesh defined again as Box01, but has defined the instance name node with value Box02. It means that the object Box01 with instance name Box02 inherit geometry and transformations from object Box01 and adds only its transformations to inherited values from Box01. The last object Box03 is instance of Box02, but is marked as Box01 too. If we created Box03, we create a connection between Box02 and Box03 too. So the Box02 is parent for Box03. The Box03 parent name has Box01.Box02 value. You can see this described example output from 3ds file using 3dsftk library

bellow this paragraph.

```
Chunk OBJECT_NODE_TAG (b002H)
  Length is 168 (a8H)

  Chunk NODE_ID (b030H)
  Length is 8 (8H)
   Node ID: 0

  Chunk NODE_HDR (b010H)
  Length is 18 (12H)
   Object name: Box01
   Flags 1: 4000
   PRIMARY_NODE
   Flags 2: 0
   No Parent

Chunk OBJECT_NODE_TAG (b002H)
  Length is 180 (b4H)

  Chunk NODE_ID (b030H)
  Length is 8 (8H)
   Node ID: 1

  Chunk NODE_HDR (b010H)
  Length is 18 (12H)
   Object name: Box01
   Flags 1: 0
   Flags 2: 0
   No Parent

  Chunk INSTANCE_NAME (b011H)
  Length is 12 (cH)
   Instance name: Box02

Chunk OBJECT_NODE_TAG (b002H)
  Length is 180 (b4H)

  Chunk NODE_ID (b030H)
  Length is 8 (8H)
   Node ID: 2

  Chunk NODE_HDR (b010H)
  Length is 18 (12H)
   Object name: Box01
   Flags 1: 0
   Flags 2: 0
   Parent 1

  Chunk PARENT_NAME (80f0H)
  Length is 0 (0H)
   Parent name: Box01.Box02

  Chunk INSTANCE_NAME (b011H)
  Length is 12 (cH)
   Instance name: Box03
```

We have to refers how had beed the parent names created. So the situation in our example is houpfully understendable. As you can see the parent for the Box03 is created based on the parent name and instance. These two information about parent is put together and the result is like this Box01 dot Box02. This principle is easy, but be aware with the Dummy object. For this object is the parent name created slight differently, because the Dummy objects have a special name introduced by the $$$DUMMY string and allways have define an instance name specifyiny which Dummy object is used as a parent. So in our library we have to solve this specific dummy object automatically.

Chapter 6

Analysis and solutions

There in this chapter we will describe analysis and the way of solution for discovered problems. When we had working in our year project, we would have discover some problematic models. Generally there is problems with keyframe part, hierarchic structure and textures for these models. A certain number of these models was created and exported into 3ds file from an old version of 3D Studio. You can imagine in exmaple 3D Studio compounds of 2D Shaper, 3D Lofter and 3D Editor i.a. running in DOS operating system. This can be the old 3D Studio definition.

Models exported from this old 3D Studio had include some additional information. In example information about working space and other unnecessary chunks. So if we load this old created 3ds file with new 3D Studio MAX, then these chunks will be cutted. But this isn't the only one differences between old and new 3ds file. There are many stranges features in this 3ds format and we'll try to show its for you.

6.1 Fundamental principles

So if we create any object in 3D Studio, then this object has defined its own primary local coordinate system. So we can mark this for our purpose as transform gizmo. We can apply on this gizmo only the translation. This translation is then stored in chunk marked as PIVOT. This chunk contains the threesomes x, y and z coordinates. The transform gizmo is shown in figure 6.1.

However there is a way how to perform transformations for local coordinate system. We can do this using the pivot point. You can see it on figure 6.2. Originally the pivot point is the same as transform gizmo. But we can perform a transformations for this one. This is usefull with performing object transformations according to pivot point. On the figure 6.3 is shown a local coordinate system (transform gizmo) and rotated pivot point in the same point. Local coordinate system and the pivot point have always the same position in world coordinate system. In fact, the pivot point stands for local coordinate system rotation. This pivot point transformation is stored in chunk marked as LOCAL_MATRIX. This is classic transformation matrix including translation, rotation and scale.

6.2 3D Studio export principles

So look at scene created in 3D Studio showed in figure 6.4. There are two boxes. The first is created on origin of world coordinates. This box has yellow color. The second box is created and translated by a few units in comparsion with the first box. Then we take the pivot point and drag him between the first and the second box. This scene composition is shown on figure 6.4. This scene composition

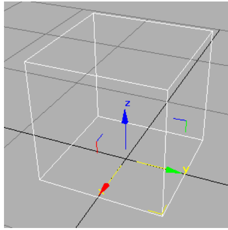


Figure 6.1: The local coordinate system (transform gizmo)

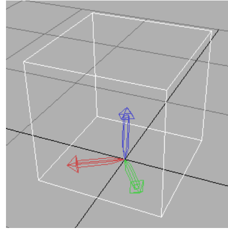


Figure 6.2: Rotated pivot point. Transform gizmo is hidden

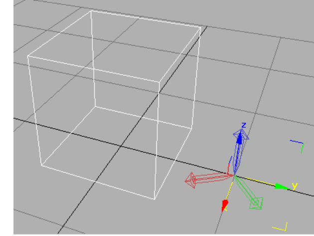


Figure 6.3: Translated local coordinate system and pivot point

we can marked as what you see is what you get. Althrough we move the pivot point, the scene stay static. But if we export this scene into 3ds file, unfortunately the scene get changed.

If we perform any number of pivot point transformations, then during the export process will be these transformations applied on to all related vertices. If somebody who is creating scene in 3D Studio transform the pivot points, there is a good recommendation for resetting pivot points after the scene is created. For our scene is the second box during the export process translated into its pivot point. Then if we read only the vertices without the respect for transformations, we get scene as is shown in figure 6.5.

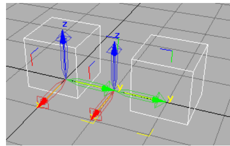


Figure 6.4: The scene in 3D Studio with two boxes. The second Box has translated pivot point

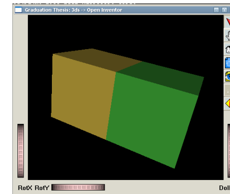


Figure 6.5: During the exporting is unexpectedly performed the translation

Next we added the rotation for pivot point to the scene shown in figure 6.4. During the export process from 3D Studio into a 3ds file is performed translation for the second cube and after it is perfomed the rotation. So it is clearly now, but what is get changed in a 3ds file?

There are a several choices for manipulating with objects (applying transformations). You can see a panel from 3D Studio at figure 6.6, where you can switch between these choices for manipulating with objects. For us are importatnt the “Affect Pivot Only” and the “Affect Object Only” choices.

We can select one of the two choices or no one. If we select the affect pivot only choice, then all the transformations will be applied for the pivot point only. If we perform translation, then then local coordinate system is affected too. But the model stays unmodified. These pivot point transformations are stored in LOCAL_MATRIX. The translation is stored into the PIVOT chunk node.

If we select the affect object only, then all the transformations will be applied for the model only. The local coordinate system and the pivot point stay both unchanged. The transformation are stored into the key frame part position, rotation and scale track tags.

There is in 3D Studio selected no chodce by default. That means for all transformations is affected object and pivot point together. So performing transformation with this mode is affected all important chunks in a 3ds file. So this is how we can work with local coordinate system and and

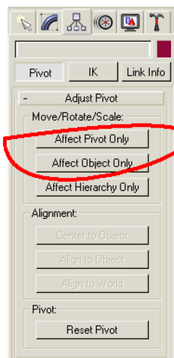


Figure 6.6: The tools for changing the method of applicate transformations

pivot point in 3D Studio.

In the key frame section is stored the object transformations. These object's transformations are related to its local coordinate system represented by transform gizmo (shown at figure 6.1). There in this section is also the translation chunk, rotation chunk and the scale chunk. There are of course the next other chunks, but these three are more important. There can be more than one transformation for each separate transformation (translation, rotation, scale). It depend on the number of defined key frames in scene. There appear a question about object's hierarchy, but about this problem we can tell you later in section 6.4.

What is also the sequence of transformations when the 3D Studio exports the scene into 3ds format? This sequence of steps describe the next list.

1. First a given object is translated into its pivot point.
2. Second a given object is rotated based on its pivot point rotation. Next is applied the object rotation related to the local coordinate system marked as transform gizmo.
3. Last is performed the scale for object.

6.3 Object as an instance

We have a certain alternative while we creating objects in 3D Studio. We can create object as an instance of one other object which was previously created in scene. The next figure 6.7 show the dialog for creating instances.

Object created as an instance hasn't its own geometry definition. There are only information about its parents and model's transformations related to the local coordinate system in key frame part. This is valid only if we didn't perform any transformations with the instance's parent. If we ie. rotate the parent for some instance, then the instance will have defined its own geometry and gets for independent object.

6.4 Hierarchic scene construction

For the present we talk about independent object created in 3D Studio. But with only this feature we won't be able to construct complex scene and animation especially. For this purposes the 3D

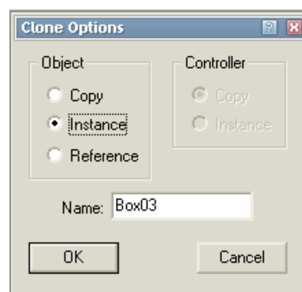


Figure 6.7: Dialog for creating instances in 3D Studio

Studio define a hierarchic structure. This structure is also included in 3ds file and at this time we can introduce it.

There in key frame part is stored the model transformations. For the present we talk about it, but we didn't define them. So next we can define the important chunk in key frame part. We know the PIVOT chnuk representing the translation between local coordinate system and his original position defined during the object creation (defined by 3D Studio internally). Next we know the LOCAL_MATRIX described above. Now we create a list including all the distinct chnuks in key frame part with short description.

1. PIVOT chunk represents translation between local coordinate system and local coordinate system's original position, defined by 3D Studio internally during creation of given object. It is usually a geometric center of object.

2. POS_TRACK_TAG chunk represents translation of pivot point considering to the world coordinate system for independent object. If the given object has a parent, then the POS_TRACK_TAG chunk represents translation of pivot point considering to its parent's pivot point position.

3. ROT_TRACK_TAG chunk repressents rotation of model considering to local coordinate system. If the object have a parent, then the rotation depends on the parent's model rotation related to parent's local coordinate system. The rotation trag tag then stand for the change of rotation, which would cause the model is independent (without parent).

4. SCL_TRACK_TAG chunk represents scale factor of model. If the object have a parent, then this chunk represent the scale diference, which would cause the model is independent (without parent).

I have to recall that for the translation transformations are the pivot point and the local coordinate system the same points. So there is one big example, where we can try to explain all the mentioned chunks in practise.

So try to imagine this example. There are two boxes. The first is Box01 and has a green color. Its position is (-10,0,0). The second box is marked as Box02 with yellow color and its position in world is (15,15,0). Next we affect the pivot only for the Box02 object and translate the local coordinate system from point (15,15,0) to point (15,15,20). Now we can write all the important chunks for better view of the problem and then make the scene more complex.

```
Box01:

PIVOT at            0.000000,  0.000000,  0.000000
POS_TRACK_TAG at  -10.000000,  0.000000,  0.000000

Box02:

PIVOT at            0.000000,  0.000000, 20.000000
POS_TRACK_TAG at   15.000000, 15.000000, 20.000000
```

At this moment we add to the scene some rotation transformations. Performing this step grow up the scene complexity, but it is still quiet good for understand. At first we rotate the pivot point for Box02 around the axis x and axis y by 45°. There in 3ds file get changed only the local matrix for storing the pivot point transformations. We won't show this matrix, because it is not so important at this moment. There in 3ds file get changed the vertices too, and it is unfortunatelly the 3D Studio export feature. The other chunks stay unchanged, because the affect pivot only.

The next step is rotation of objects as a compex unit. That means rotation of object and its pivot points together. We rotate the Box01 object around the axis z by 60°. Then we rotate the Box02 object by the 25° around the axis y. So we rotate the pivot point by given angle and the object related to local coordinate system by the same given angle together. The 3ds file contents will look like this.

```
Box01:

PIVOT at            0.000000,  0.000000,  0.000000
POS_TRACK_TAG at  -10.000000,  0.000000,  0.000000
ROT_TRACK_TAG at    0.000000,  0.000000, -1.000000, 1.047197°

Box02:

PIVOT at            0.000000,  0.000000, 20.000000
POS_TRACK_TAG at   15.000000, 15.000000, 20.000000
ROT_TRACK_TAG at    0.000000, -1.000000,  0.000000, 0.436332°
```

Now we perform the scale transformation for Box01 by 150 percentage factor around axis x. For the Box02 we perform scale around the axis z by percentage factor 200. The 3ds file then contains these informations.

```
Box01:

PIVOT at            0.000000,  0.000000,  0.000000
POS_TRACK_TAG at  -10.000000,  0.000000,  0.000000
ROT_TRACK_TAG at    0.000000,  0.000000, -1.000000, 1.047197°
SCL_TRACK_TAG at    1.500000,  1.000000,  1.000000

Box02:

PIVOT at            0.000000,  0.000000, 20.000000
POS_TRACK_TAG at   15.000000, 15.000000, 20.000000
ROT_TRACK_TAG at    0.000000, -1.000000,  0.000000, 0.436332°
SCL_TRACK_TAG at    1.000000,  1.000000,  2.000000
```

At this time we create at the scene a hierarchic structure between the two boxes. We define the Box01 object as a parent for Box02 object. The we export the model into 3ds file and dump the binary data in to the text data using the 3dsftk library. Well here is the diference between scene with defined hirarchy (above) and scene without hierarchy system (bellow).

```
Box01:

PIVOT at            0.000000,  0.000000,  0.000000
POS_TRACK_TAG at  -10.000000,  0.000000,  0.000000
ROT_TRACK_TAG at    0.000000,  0.000000, -1.000000, 1.047197°
SCL_TRACK_TAG at    1.500000,  1.000000,  1.000000

Box02:

PIVOT at            0.000000,  0.000000, 20.000000
POS_TRACK_TAG at   16.990000,-14.150000, 20.000000
ROT_TRACK_TAG at   -0.215236, -0.330777,  0.918836, 1.127305°
SCL_TRACK_TAG at    1.000000,  0.670000,  2.000000
PARENT is Box01
```

We can see on the result, that the parent Box01 stay unchanged. But the child object Box02 had changed the position, the rotation and the scale chunks. Our method of processing scene with defined hierarchy is based on division the hierarchy models into separate objects. Next we can explain a way leading towards these results.

6.4.1 Finding the independent matrices

If we hadn't defined hierarchy in scene, then we only need apply some transformations described bellow this text in subsection 6.4.2. But if there is defined a hierarchy in 3ds file, then the position, rotation and scale chunks from key frame part are related to the parent's local coordinate system. There is true, that each object can have a parent. But this parent can have a next parent too. So the base object can have in fact one or more parents. We also need to find these parents and apply their transformations on the base object.

For our example there is one parent for the object Box02. We should also find the position, rotation and translation of parent Box01 object and apply them on all the child Box02 position,

rotation and scale track tags. By this way we are able to use these transformed chunks for repeat the Box02 performed transformations.

Position track tag finding

At first we transform the given child Box02 position track tag. So we find the position track tag for its parent Box01. We need rotation and scale track tags for object Box01 too. These three track we can use for creating a three transformation matrices. It will be translation matrix, rotation matrix and scale matrix. These three matrices will be applied on the Box02 position track tag.

For our example we apply step by step the Box01 scale (1.5, 1.0, 1.0), rotation around z axis by 60° and at last the transaltion (-10.0, 0.0, 0.0) on the Box02 position track tag. After applying these three transformations is the Box02 translation the same as the scene without hierarchy.

```
Child translation         16.990000,-14.150000, 20.000000
After parent scale        25.490000,-14.150000, 20.000000
After parent rotation     25.000000, 15.000000, 20.000000
After parent translation  15.000000, 15.000000, 20.000000
```

Rotation track tag finding

Well we just know the child position track tag. Next we would get the rotation track tag by similar way. So we have the parent's rotation matrix. We simply multiply this parent's (Box01) rotation matrix with the child 's(Box02) roation matrix. We create the Box02 rotation matix from the Box02 rotation track tag.

For our example we take the parent rotation around the z axis by 60° and multiply this matrix with the child rotation matrix. The result is rotation for child, that means the child is detached from parent.

```
Parent rotation         0.000000,  0.000000, -1.000000, 1.047197°
Child rotation         -0.215236, -0.330777,  0.918836, 1.127305°
After multiplication    0.000000, -1.000000,  0.000000, 0.436332°
```

Scale track tag finding

At last we find the scale track tag for our child Box02 object. We have to rotate the parent's scale using the child's rotation matrix. By this way we transform the parent's scale into child's local coordinate system. Then we simply multiply this transformed scale vector with the child's scale vector.

For our example the parent scale is (1.0, 0.67, 2.0). We can apply on this vector the child rotation matrix. The result from this action is transformed scale (1.0, 1.5, 1.0). The we simply multiply the child scale (1.0, 0.67, 2.0) with the transformed parent scale (1.0, 1.5, 1.0).

```
Parent scale               1.500000,  1.000000,  1.000000
After child rotation       1.000000,  1.500000,  1.000000
Multiply with child scale  1.000000,  1.000000,  2.000000
```

You can be confused about the rotation angle sign. The 3D Studio turns the sign for rotation during the export process. So for some case we have to first turn the sign and after it use the rotation transformation.

6.4.2 Transforming vertices

So at this moment we know the key frame informations for all objects in scene. We can simply apply the described principles about scene construction. That means we have to perform transformations for vertices at first. These transformations don't depend on any other object and stay for the independent object transformations.

For all vertices are applied the pivot point transformations during the export process. So we have to eliminate these transforamtions. We use the LOCAL_MATRIX chunk for this purpose. But this matrix can include the models transformations too. If we apply the LOCAL_MATRIX inverse transformation, we have to perform subsequently all the object transformations after apply the inversion transformation (except translation). The result is scene without transformations. So we have to perform the transformations by key frame informations. The transforming vertices process have a four stages. Next we try to describe them.

1. The first stage stays for performing translation of vertices. These tranlation is related to instances in 3ds files. Each created instance inherit parent's geometry and the LOCAL_MATRIX too. Also the instance can have a different position considering to the parent. The translation differs for instances and for all their childrens. The translation is computed as difference between the object's translation track tag and the LOCAL_MATRIX translation. This LOCAL_MATRIX translation stays for the pivot point translation. For objects without defined instance, is this difference equal to zero. This doesn't need be true for some models created in old version of 3D Studio.

2. The second stage stays for performing rotation. We have to rotate the model with negative angle around the object's pivot point. The rotation is performed with center in pivot point. We have to use the negative scale, because during the 3D Studio export is the angle's sign turned around.

3. The third stage stays for performing scale. In this stage we have to perform two steps. The fist is to eliminate scale bug in 3ds file. This acale bug appear if we use a negative scale in odd number of coordinate axis. That means in one or three coordinate axis. If this bug appear, the model is scaled around the x axis by $180°$. So if we can solve this bug, we must perform reverse scale around the x axix. That means negative scale around the x axis by $180°$. So now we can perform the object's scale. Also we simply apply the scale track tag with center in pivot point.

4. The last fourth stage stays for performing inverse transformation of LOCAL_MATRIX. At this time we perform translation from pivot point back to the original position defined by original local coordinate system.

Chapter 7

Testing

There in this chapter we will show to you some problematic models. These models are usefull when we solving and discovering the 3ds file format principles. We can show on these problematic model the difference between the 3ds2iv convertor created as part of our year project and between this new 3ds2iv convertor improved on our graduation thesis.

7.1 Scale bug in 3ds file

This test shows the scale bug in 3ds file format. This bug appear allways when we use negative scale for odd number of coordinate axes. That means we use ie. a negative scale around the x axis by -150 percentage. There in the figures bellow you can see the scale bug side effect for tennis racket model. On the left side is scale bug and on the right side is this scale bug solved.

The scale bug can be discovered using the linear vector algebra and the dot product equation. If we compute normal for the xy plane, we get the normal vector corresponding to the z axix vector. But if we perform ie. negative scale around the x axis, we get the reverse not corresponding to the z axis vector. So for corresponding vectors is the dot product greater than zero. But for non corresponding vectors is the dot product result less then zero. In this case we perform the negative scale around the x axis. This method for solving this bug we partially get from 3D Studio MAX SDK. Originaly we solved this problem only for scale, but there is hidden possibility of this bug appear in other cases. This dot product solve all those cases.

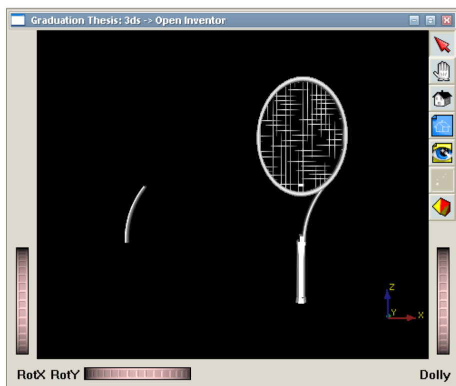


Figure 7.1: Example of scale bug in 3ds file format

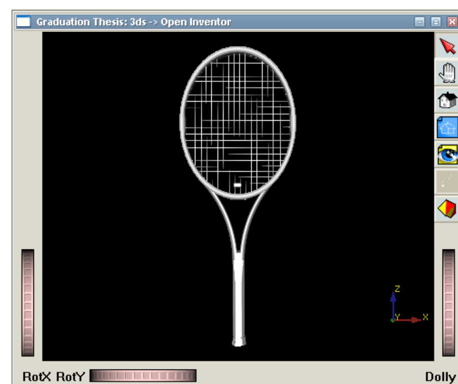


Figure 7.2: Solved scale bug

7.2 Translation to pivot point

During the export process is all objects in scene translated into its pivot points. We can show you a small example. There are all ducks translated into its pivot point. On the left sided figure **??** is the pivot point problem and on the right sided figure 7.4 is this problem solved with our new 3ds2iv convertor.

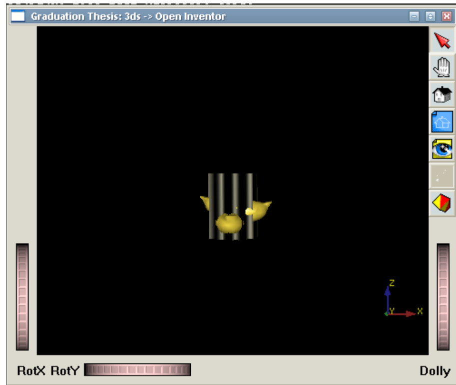


Figure 7.3: The ducks are translated to pivot points

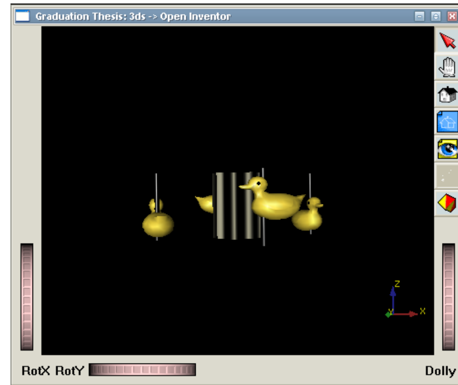


Figure 7.4: Back translation to their original positions

7.3 Hierarchy system and instances

In this test we can see almost all problems which we succesfully solved. There is the pivot point translation problem on the figure 7.5. Next there is the problem with instances (the four arms for the four choppers). There is defined a old hierarchy structure too. So we have to solve this old hierarchy loading in our library lib3ds. On the right figure 7.6 is the right loaded model with our new 3ds2iv convertor.

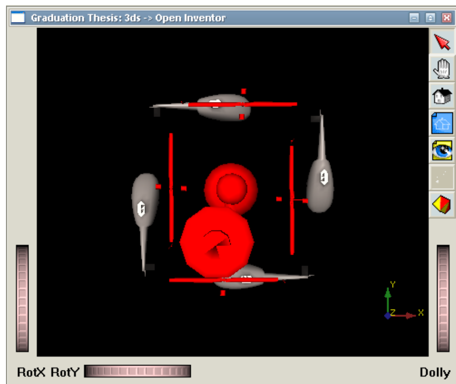


Figure 7.5: Complicated scene with the distinct problems

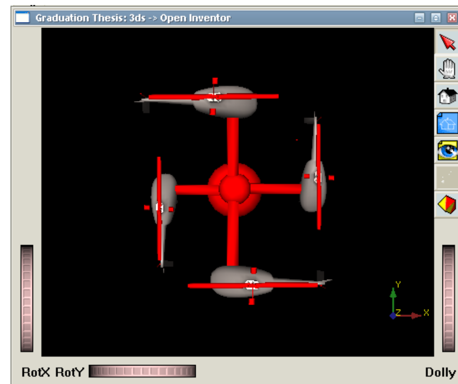


Figure 7.6: The result of our analysis

7.4 Texture optimalization

This test show a difference with textures between old 3D Studio and a new 3D Studio. If we take a original model of can, we get the model shown in figure 7.7. So if we import this same 3ds model into new 3D Studio, we get the model on figure 7.8. The new 3D Studio solve this problem by hidden way. Actually we try to solve this problem. This is the our last known problem with 3ds models.The principle is in augumentation the scene of new vertices, that cover the texture vertices size optimalization. So this is a rare problem and all the main problems described above this paragraph are solved succesfully.

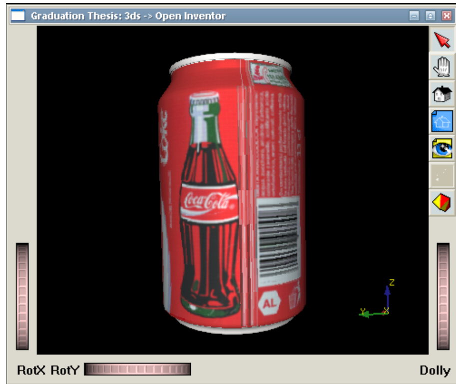


Figure 7.7: Model with texture wrapping problem

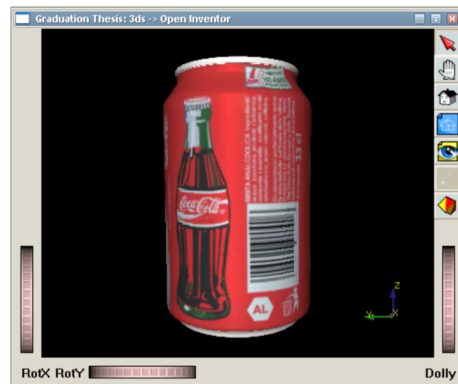


Figure 7.8: After exporting from new 3D Studio

7.5 Where is the truth

Some model can look like they are rendered by wrong way. But if we try import these models into 3D Studio, we get the same “bad” scene. These model are created by authors who didn’t know anything about the 3ds format. After they export the model into 3ds format, they didn’t load them back in to 3D Studio for get a check its correctness. There are a lot of models with these “errors”, but there is no one distinct way to determine their correstness. Next we can show you some of these models.

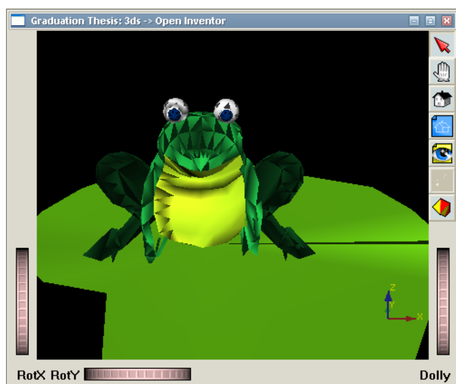


Figure 7.9: Frog with many flipped normals

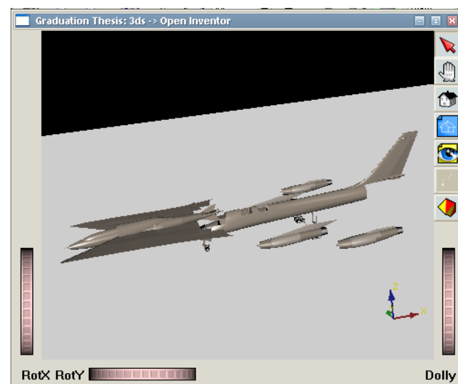


Figure 7.10: Bad created model of aircraft

Chapter 8

Conclusion

There was created a program for convert 3D Studio 3ds file format into the Open Inventor iv file format. Simultaneously was performed the 3ds file format analysis.

I cannot help but I have to started this chapter with the 3ds file format critique. First and foremost I have to claim, that the 3ds file format documentation is for programmers absolutely unusable. I suppose that programmers can write professional programs, but with this documentation it is impossible. I'm sure for simple application is this documentation sufficient and everyone is able to write 3ds file loader in short time without previous knowledges.

In this thesis was described and solved all the main problems that had been appear around the 3ds file format ever. After reading this thesis is everybody going to be able to named the main 3ds file format problems and going to be able to solve them easilly. I think this is the main contribution of my work.

Due to the analysis we programmed the 3ds2iv convertor. The main features of this convertor are loading and transforming geometry, loading and applying materials, solving smoothing groups, computing normals, solving hierarchy system and animation, solving the scale bug and textures with bump mapping. The other free 3ds loaders implement only loading the geometry and materials.

During the work on this thesis we tested about tens of models. Approximate rate of models with one or more described problems was about fifteen percent. Absolute majority of these problematic models are succesfully solved thanks to our analysis.

The possibilities of expanding the 3ds file format is propably minimal rather there are no possibilities. But there are still a lot of created models on the internet and it is a good idea to convert these models into Open Inventor for future and simpler use. On the other hand the 3ds file format is still alive in CAD systems and is still supported for all professional programs from this line.

There are also the next possibilities of development. It could be written the complex convertor from 3ds to many other formats. Next there are a limitations in the lib3ds library ie. not implemented lights or not implemented camera. The next possibility for future work is automatic repairing for the bad created models. There are a technique in 3D Studio named unifying normals, that may by applied onto these models.

This 3ds2iv convertor is embeded into the Lexocad application developed by the CADWork company. It shows the focus on the 3ds file format from the companies and give this work an other purpose.

Bibliography

[1] Coin 3D. Open source implementation of open inventor and developer page. `http://coin3d.org`.

[2] ATI. Utility for generate normal maps for bump mapping. `http://www.ati.com/developer/tools.html`.

[3] Cyberloonies. Official 3ds file format documentation and 3dsftk library. `http://cyberloonies.com/3dsftk.html`.

[4] Luebke D. *Level of detail for 3D graphics: Application and Theory*. Morgan Kaufmann Publishers Inc, 2003. ISBN 15-586-0838-9.

[5] Silicon Graphics. Official page. `http://sgi.com`.

[6] Open Inventor Architecture Group. *Open Inventor C++ Reference Manual*. Addison-Wesley Professional, 1994. ISBN 02-016-2491-5.

[7] Wernecke J. *The Inventor Mentor: Programming Object Oriented 3D graphics With Open Inventor*. Addison-Wesley Professional, 1994. ISBN 02-016-2495-8.

[8] Petr Felkel Jiří Žára, Bedřich Beneš. *Moderní počítačová grafika*. Computer press, 1998. ISBN 80-722-6049-9.

[9] Jan Kříž. *3ds max 6 praktické postupy*. Computer press, 2004. ISBN 80-251-0329-3.

[10] SGI Techpub Library. Electronic version of some books and more usefull information. `http://techpubs.sgi.com`.

[11] OpenGL. Sobell filter for compute magnitude and edges orientation. `http://www.opengl.org/`.

[12] Ing. Jan Pečiva. Seriál článků o open inventoru. `http://root.cz/clanky/open-inventor`.

[13] David Řeháček. *3D Studio 4.0/MAX*. Computer press, 1997. ISBN 80-858-9687-7.

List of Figures

2.1 No Shading 11
2.2 Flat shading 11
2.3 Smooth shading 11

4.1 Convertor Flow chart 15
4.2 Base 3ds file format content for creating geometry 16
4.3 The generall scene graph. Scene is composed from vertices nodes and material separator class which will be next expanded 18
4.4 We add the SoTransform node into the scene. Transformation is perform over the SoCoordinate3 vertices 19
4.5 We add the material properties nodes and the class separator for normals into scene graph 19
4.6 We expand the Normals separator class and get the a functional scene graph 20
4.7 Scene graph with included texture's nodes in Mesh class 20
4.8 Scene with included other needed nodes in Material class 21
4.9 The complete scne graph for one mesh and one defined material 21

6.1 The local coordinate system (transform gizmo) 28
6.2 Rotated pivot point. Transform gizmo is hidden 28
6.3 Translated local coordinate system and pivot point 28
6.4 The scene in 3D Studio with two boxes. The second Box has translated pivot point 28
6.5 During the exporting is unexpectedly performed the translation 28
6.6 The tools for changing the method of applicate transformations 29
6.7 Dialog for creating instances in 3D Studio 30

7.1 Example of scale bug in 3ds file format 35
7.2 Solved scale bug 35
7.3 The ducks are translated to pivot points 36
7.4 Back translation to their original positions 36
7.5 Complicated scene with the distinct problems 36
7.6 The result of our analysis 36
7.7 Model with texture wrapping problem 37
7.8 After exporting from new 3D Studio 37
7.9 Frog with many flipped normals 37
7.10 Bad created model of aircraft 37

Appendix A

API for the Lib3ds library

int createModel(char *name)

This function loads the model from file named "name" into the memory. Later you can manipulate with the loaded model. If the model is loaded correctly, then the function return 1 otherwise it return 0.

void releaseModel()

This function releases all allocated memory for a given object.

void getMeshList(meshListLib3 **meshList)

This function return a list of all mesh names that are included in 3ds file.

void getMesh(char *name, meshLib3 **mesh)

This function return a mesh from "name". The first parameter "name" you can get from the meshList.

void getMaterialList(materialListLib3 **materialList)

This function return a list of all material names that are included and used in 3ds file.

void getMaterial(char *name, materialLib3 **material)

This function return a material from "name". The first parameter "name" you can get from the materialList.

void getKeyFrame(char *name, kfmeshLib3 **kfmesh)

This function return a key frame informations for mesh named "name".

void getKeyFrameInstanceByIndex(char *name, int index, kfmeshLib3 **kfmesh)

This function return a key frame informations for mesh named "name" and for it's indexth instantce.

int getMeshInstanceCount(char *name)

This function return count of instances for given mesh named "name". The parameter you can get from the meshList. If there is no instance in 3ds file defined, then is returned the one value (it stays for the given mesh). If there is no animation defined for the given mesh then zero is returned.

Appendix B

Howto using the 3ds2iv

```
Usage: 3ds2iv [options] infile [outfile]
  -e : Use Examine viewer to show the model
  -f : Compute normals using flat shading
  -h : Print this message (help)
  -s : Force two sided materials
  -t : Do not apply textures
  -v : Do not create outfile, only use Examine viewer to
       show the model

If no output file name is specified, stdout will be used.
```

Appendix C

Companion cdrom

There is described the cdrom directory tree.

- `/bin` – binaries and libraries for win32/linux platform
- `/src` – source codes for lib3ds library and 3ds2iv convertor for win32/linux
- `/models` – example and testing models
- `/doc` – electronic version of this report (pdf and latex source too)
- `/utilities` – Coin3d installation for win32/linux platform

The detailed description for example models and for compilation from source codes is described in given directories.