Contents

List of Figures

Chapter 1

The Scene File Format

1.1 Introduction

The major hurdle for most of the researchers while using synthetic data is that the tools developed for data generation are not scalable. Such tools cannot be used by other researchers to generate data with some other scene setup. This limits the data that can be shared among the researchers. The datasets cannot be changed very easily. Some researchers have tried to over come this hurdle by specifying the constraints and methods to be used for acquiring the datasets [**?**]. A lot of time goes into configuring these tools to generate the data which is acceptable for testing the algorithms. If such data generation tools provide some method of sharing the scene created for generating the datasets, the users would be able to make minor modifications like changing the objects in the scene and use the same configuration of cameras, lights etc to generate new datasets.

Many ray tracing softwares support very powerful high level scene description languages. Users can represent complex scenes which mimic the real world with ease. These scene description languages are generally in ASCII format (human readable format), but they are very complex to understand at the first look. Since complex mesh objects require a large number of vertexes and texture coordinates, the amount of data that is present in these scene description files cannot be understood without the help of some kind of scene graph editor. Many 3D authoring tools have the capability to generate the scene description of the scenes setup using them. As stated in the earlier chapters, Though such tools have been very effective in generating high resolution images with photo realistic quality, It is very unintuitive to extend these softwares to generate any other representation which may be useful for creating datasets,

crucial for testing CV and IBR algorithms.

A data generation tool would require a more robust and abstract representation of the world. A representation that can be extended to incorporate any new extensions made to the tool. Care has to be taken not to make such scene files human readable and at the same time extend-able. Human readability of the files is an important since it enables the user to make modifications to the scene file to match his/her requirements. The scene files have to be independent of the data they have been created to generate. Such abstraction would enable generating different kinds of data using the same scene file. Due to the lack of such standard scene format which supports both dynamic and static objects, we have formulated our own new file format for such scene description.

1.2 Design

Based on our analysis of the requirements of the scene description format, we have designed an ASCII file format for storing the scene that the users setup using our tool. The properties of the objects such as their positions, name of the 3D model file are to be stored in the scene file. The information about the data generated or that can be generated should not be stored in this file. We have designed the file format so that it is very simple to parse. It's also very easy to generate scene files using the models that are already available with the user. The scene files used by our tool have a ".scene" suffix.

The over all structure of a scene file is of the structure:

```
(Object Type) (Object Details)
```

The scene file always starts with a number describing the total number of objects that are in the scene. The whole scene file is divided into two blocks. The first block describes the information about the models, lights and cameras that are present in the scene. The second block again contains two or more blocks (each representing a key frame) based on the number of key frames. By default the user has to mark a minimum of two key frames. This constraint has been imposed so as to make use of the same rendering pipeline in our tool for both dynamic and static scenes. The key frame block has the following structure:

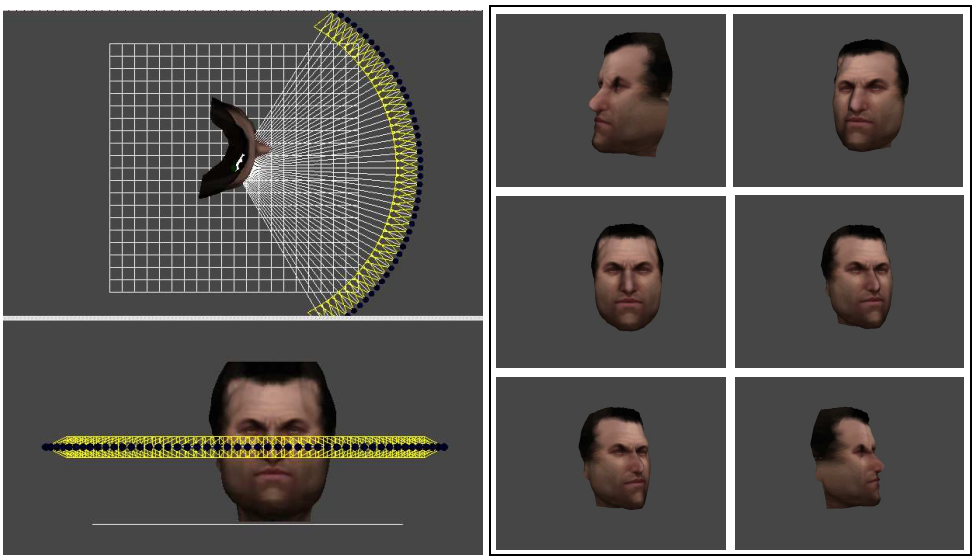
```
(Frame Number)
```

```
(Details of all objects, one in each line)
```

The order in which the object details are specified in the key frames have to be in the same order as they have been specified in the first block of the file. This is important so as to maintain consistency across different key frames. The following is the basic structure of the whole scene file. $\%f$ indicates floating point value, $\%d$ indicates an integer value and $\%s$ indicates a string value.

```
%d (num objects)
Object Type: <object parameters>
...
...
%d (num key frames)
%d (frame number)
<object parameters>
...
...
%d (frame number)
...
...
```

The first line in the file describes the number of objects present in the scene. The lines following this number are the details about each object. We have defined special tags for each object type describing the kind of object that follows, to make the file more readable and to perform extra operations on the scene file. For example, the camera object has the location and the look at values. These values are represented by two objects and hence require the tool to read another object in the file when it finds a camera object. This is a simple strategy we have adopted to ensure easy extendability of the tool to support different input formats. There are some predefined object types that are currently supported in our file format. $glObject\_Light$ refers to light objects, $glObject\_AC$ to represent an AC3D file or a povray scene file, $glObject\_Camera$ for specifying the camera details, $glObject\_3DS$ for 3ds objects and $glObject\_Md2$ for md2 objects. Following this information is the number of key frames in the scene. This gives the total number of key frames that have been setup by the user in the scene. What follows is a sequence of frame number and object details blocks. Each of these are important building blocks of the dynamic scene to be generated using the tool. Each key frame represents a time instant.



(a) (b)

Figure 1.1 (a) A semi circular setup of 48 cameras for acquiring the images around the face model. The cameras in this scene were added not using the tool but through another program which generated the center and look at points of all the 48 cameras. Such large number of cameras cannot be added manually into a scene as it is very time consuming process. (b) Images of some of the views of the camera setup.

For all the time instances in between these key frames, the tool has to interpolate the object properties based on their values at the key frames.

The file format does not have any limitation on the number of objects of any type that can be placed in the scene file. Many objects, cameras, light sources can be present in a single scene file. Due to the freedom over the number of objects (cameras especially) users will be able to create complex camera configurations like hemisphere or circular etc. Figure 1.2 is an example of complex camera setups possible using our tool by specifying the cameras externally in the scene file. This is an important feature because specifying parameters of each camera in the GUI can be inconvenient. Since we are using OpenGL for rendering purposes, the common number of lights supported on most of the graphics hardware is eight. This is the current limitation on the number of lights that can be put in a scene.

1.3 Implementation

We have implemented a simple parser for parsing the input scene files and setting up the scenes in the tool. The integer value in the first line is read and based on this number, the rest of the data is loaded. As

Figure 1.2 An outdoor scene created and rendered using our data generation tool. This scene has multiple objects and a single camera unlike the above figure.

stated before every line describes an object type followed by the details about that particular object. The tokens describing the object id are used for creating an instance of the corresponding class implemented for loading that object type. For example, we have implemented a class called $acObject$ which inherits a base class called $glObject$, for loading the ac3d models. Similarly we have implemented $md2Object$, $Model\_3DS$ for loading MD2 and 3DS models respectively. While loading the scene files, the tool loads all the key frames in the scene. These key frames are shown in the time line just like they were when the actual scene was created. While loading the information about the key frames we assume that the order in which the object details appear is same as the order in which the object model details appear in the first block of the scene file.

The following is an example scene file:

```
3
glObject_AC: tuxedo.ac
glObject_AC: world.ac
glObject_Camera: ( 60, 0.1, 1000, 1.33 )
```

```
[ 2.45 0 -7.24999, -4.5 0 4.09998, 0 1 0 ]
2
0
< 0 0 0 > { 1.00 , < 0, 0, 0 > }
< 0 0 0 > { 1.00 ,< 0, 0, 0 > }
< 2.45 0 -7.24999 > { 1.0000 ,< 0, 0, 0 > }
< -4.5 0 4.09998 > { 1.00000 ,< 0, 0, 0 > }
2
< 0 0 0 > { 1.00 , < 0, 0, 0 > }
< 0 0 0 > { 1.00 ,< 0, 0, 0 > }
< 2.45 0 -7.24999 > { 1.0000 ,< 0, 0, 0 > }
< -4.5 0 4.09998 > { 1.00000 ,< 0, 0, 0 > }
```

The scene file shown above describes a scene with two 3d models (ac3d files tuxedo.ac and world.ac) along with a camera. During design of our tool, we have combined the rendering pipeline for both static and dynamic scenes. Due to this design methodology, every scene file should have two key frames even though there is no motion in the scene. The above scene file has two key frames one at frame 0 and another at frame 2. You will observe that there is no change in the position of the objects in the two frames.