Scene Description Issues for Computer Graphics

Martin Preston, Paul Lever

Computer Graphics Unit, Manchester Computing,
University of Manchester, Manchester, M13 9PL
Email: *{preston | lever}@mcc.ac.uk*
WWW: *http://info.mcc.ac.uk/CGU/*

Abstract

Whilst most scene description languages were developed to be produced and consumed by programs, they are often maintained by users. Unfortunately this tends to be a time-consuming enterprise, as the languages provide little assistance for even moderately sized scenes. This paper advocates the application of existing programming paradigms (including object orientation, procedural descriptions and functional composition) to this scene description process, and highlights how they may be used to considerably simplify scene manipulation.

Keywords Scene Description, Object Orientation, Functional Programming, Procedures.

1. Introduction

Most 3D graphics applications utilise some form of scene description file, which contains a static description of the virtual world. Whilst these are conceptually just ways of providing the illusion of persistence, and so should not be of interest to users, the lack of standards for inter-application scene transfer, the immaturity of scene editing programs and the inevitable desire to 'cut-corners', means that users want to be able to manipulate these files directly.

We can classify existing scene formats into two groups, the first consisting of those that were designed without any thought given to editing the scene manually, and secondly those that are intended to assist in human maintenance. The first group tend to have a fixed syntax, and contain little, if any, descriptive text to enable a reader to relate components of the file to portions of the 3D world. Examples of this sort of file include NFF [6], DXF [3] & MGF [4]. The second group differ primarily in their use of a more flexible layout, which is often parsed using an elaborate syntactical analyser (such as yacc), and the provision of labelling. Common files of this style include RIB [10], MSDL [8] & VRML [11].

Unfortunately, even using the most flexible of languages from the second group, scene maintenance is a time consuming and problematic exercise. We can identify three principal reasons for this:

- Realistic scenes require large descriptions. Manipulating large text files, whose contents often owe more to the application specific properties of the scene (such as whether NURBS or polyhedra are employed) rather than the 'meaning' of the scene, is awkward.
- Large scenes are likely to contain errors, and as visualising the effects of a modification

to the textual description is difficult to visualise in terms of the 3D scene, scene debugging often requires trial-and-error.

- Generating large scenes from the form of basic components provided is extremely difficult, and frequently requires an in-depth knowledge of current (low level) computer graphics techniques.

It is the thesis of this paper that the problems exhibited by contemporary scene description languages are similar to those identified in programming languages, such as difficulties of code reuse, and long timescales. As computer science has developed a number of successful strategies to alleviate these problems, it is worth considering whether they may be usefully employed in easing the process of scene description. In this paper we first identify the most useful paradigms produced for software engineering, and propose mechanisms which allow them to be adopted in a conventional scene description languages. Though we use MSDL [8] as the basis for our work, we feel that the techniques developed here are likely to be relevant to most scene description formats.

The rest of this paper is structured as follows. In Section 2 we discuss the current state-of-the-art in scene description languages, and identify the particular strengths and weaknesses of the most popular formats, as well as highlighting related attempts to use software engineering paradigms in computer graphics. Section 3 highlights the most promising coding paradigms available. Following this sections 4-6 discuss first how each of these techniques relates to scene description, and then (for each) describes how they can be included in our base language (MSDL). Finally Section 7 discusses the strengths of this approach, and identifies areas for future work.

2. Related Work

In this section we first (in 2.1) discuss the most popular contemporary scene description languages, before discussing how programming paradigms have been exploited in CG in Section 2.2.

2.1. Existing Scene Description Languages

As described earlier, there are a great many scene description languages, though they may be neatly categorised into two large groups. The first of these were designed for use as a low level scene transferral mechanism, and so little thought was given to easing manual maintenance. The second group has made some attempt to ease this process, but in practice this only assists in extremely small scenes.

The first group, which we can term the 'basic' languages, contain a great many application specific formats. Indeed, until the development of basic ray-tracers by the amateur community becomes less fashionable, this situation is likely to worsen. However there are a few of the more portable (or at least widely used) formats that are worth discussing.

NFF [6] is one of the most popular scene description formats in computer graphics research, largely as it was used to define a number of 'standard' scenes, which have since been used as a mechanism for benchmarking software. Its principal contribution to convenient scene manipulation (apart from its supplied reader) is the use of procedural definition to assist in the description of repetitive structures (such as trees). It has been extended into ENFF, and has acted as an influence on other languages, but unfortunately lacks many of the features (such as NURBS) which are required in modern languages.

DXF [3] was produced by Autodesk to allow their popular Autocad program to save scenes. DXF has been through various revisions, and as Autodesk have not encouraged the widespread use of DXF, tends not to be backwards compatible. However, given the large number of scenes available, and the benefits from using a commercial CAD program for scene generation, its various versions have attracted use by other applications.

OBJ [5] shares the commercial background of DXF, but is designed to encode more widespread data, and includes surface descriptions. OBJ was produced to enabled Wavefronts' (now Alias|Wavefront) line of graphics software to exchange scenes, and has benefited from the availability of documentation describing its format. OBJ is often used as a way of transferring scenes between high-end commercial graphics packages, and is probably the strongest of the basic languages.

MGF [4] (Materials and Geometry Format) was developed to allow researchers in the field of global illumination to share a scene description which includes enough data to allow more advanced rendering algorithms to be employed. It serves as a good example of the tendency to include data in scene files which are specific to particular uses, though (given the target market) MGFs' peculiarities are understandable.

The second group of languages, which often employ more elaborate scene parsing mechanisms, are usually more portable, though there is a tendency to extend them for specific circumstances. We can identify three representative formats, though there are many more in use throughout the graphics community.

RIB [10] (the Renderman Interface Bytestream) is the scene format recognised by the Renderman rendering architecture proposed by Pixar. RIB includes a wide range of primitives, comments, flexible layout, and so requires parsing. A small number of reading applications do exist (such as BMRT), but the scarcity of public domain parsers has hindered its widespread acceptance. RIB was originally marketed as a "3D PostScript" but has not gained widespread acceptance. The success of Renderman in commercial animation rendering is due primarily to the shading language (which is not part of RIB).

MSDL [8] (The Manchester Scene Description Language) is one of the public domain scene languages which supply their own parser (built from lex and yacc) to help researchers integrate it with their work. It includes a wide collection of primitives (including NURBS), and provides basic support for object definition and instantiation, but suffers from the weaknesses that we have previously described. It is not intended to provide support to any particular mechanism for storing or rendering scenes, unlike most other scene languages.

VRML [11] (the Virtual Reality Modelling Language) was originally conceived as a mechanism to allow VR work to be integrated with the World Wide Web. Following extensive discussion on email lists this admirable goal was shelved, and instead the developers concentrated on finding a form of scene description language which would be portable, usable and also suitable for later extension. Following a voting process the SGI Inventor file format was adopted, which provided much of the functionality required with minimal extension (simply adding URL tags to enable links to be embedded in scenes). However, in the process some problems were caused: a scene data structure which matched Inventors was assumed, no effort was made to enforce standard meanings for many of the tags (such as lighting values), and curved surfaces were removed. VRML 1.0 has been used for the basis of many competing commercial extensions (many of which are simply Inventor files) and has been used in certain fields as a 'lowest-common-denominator' scene transferral format. Given the widespread adoption of versions of VRML it is likely that it will continue to develop rapidly, but it is not clear whether a single 'standard' will remain.

As VRML adopts a layout and syntax closely tied to the object oriented nature of the data structure which produced it, editing a file by hand is an awkward process, as one is forced to struggle with a layout which often has little relation to the visible properties of the scene. Consequently VRML is unusual in our second group of languages in combining the disadvantages of the basic languages (awkward syntax) with few of the advantages of the second group.

The file formats we have discussed have been chosen simply because they are representative of the sort of file formats in regular use today. A more extensive list, including documentation for each, is available [1].

2.2. Graphics Paradigms

Whilst conventional procedural paradigms have been used for many years in computer graphics, some considerable effort has been expended in developing more suitable abstractions to assist in application development. SGI's Inventor [9] (whose file format is essentially VRML 1.0) has successfully used many object oriented ideas in a graphics API. The ability to form inheritance graphs considerably simplifies many jobs. Considerable research has been performed in applying functional programming methods to graphics applications, though little of this has reached the marketplace.

These advances are interesting, and imply that such paradigms have a place in the future of graphics, but do not relate directly to static scene description files. However some work has been performed in using higher level scene description methods in CAD, with one of the more remarkable examples of this being Donikian & Hegrons' [2] system which allows basic statements about the scene to be made, from which the CAD system must infer the actual contents of the scene. For example their system allows statements such as "The chair is in front of the table" and "the table occupies half of the floor space" to be used to guide scene production. Methods based on this technique could in theory be used as the basis for scene description files, but they require considerable computation, and so remain an area for further work.

3. Programming Paradigms for Scene Description

In previous sections we have identified the problems faced by contemporary scene description formats, and highlighted the relative strengths and weaknesses of a representative sample of popular formats. Now we can turn to identifying the most significant features of programming languages, before going on to adapt them to scene description in subsequent sections.

In searching for promising features it is helpful to relate existing scene description formats to software engineering. The basic languages we identified in Section 2 correspond to machine code programming, where no abstraction whatsoever hides the programmer/scene author from the way in which the file will be executed. The second group of languages improve little on this state, and so can be thought of as 'macro assemblers', i.e., simple pieces of code which translate easily to machine code.

The advent of higher level programming languages considerably eased the software development process. The first successful paradigm which was applied to such an engineering task was 'procedural definition', i.e., blocks of code which achieve a definable goal, and which may be employed by other segments of the program without the caller needing to be aware 'how' the goal is reached. Therefore it is worth considering how reusable procedural definition may be used in scene description. Note that previous 'procedural scene languages' have merely used this ability to generate repetitive structures, little effort has been made to produce reusable parameterised scene definition.

A second advance in software engineering can be adequately termed object based programming, where the program is constructed from chunks of existing code which may be easily modified and combined (unlike procedural languages) to produce larger, and more functional, applications. Object oriented methods have, as previously described, been used in assisting 3D graphics work, but the most significant OO SDL (VRML) does not assist in scene editing. Therefore it is worth considering how this situation may be improved, and whether OO definition may be usefully employed in scene maintenance.

Finally we can identify another programming paradigm, functional programming, which attempts to shift the focus of programming to "what the program does" (i.e., its function) rather than "how does the program do this". Functional programming has yet to gain widespread acceptance in commercial code production, but is worth investigating for scene description, as scene authors should be more concerned with the purpose of the scene, rather than the relevant graphics primitives used to represent it.

3.1. The Manchester Scene Description Language (MSDL)

In the following three sections we describe how each of these paradigms may be exploited in scene production, and then present a sample syntax for MSDL to demonstrate how this might be achieved. In order to be able to relate the new terms to MSDL we first outline the features of the existing language.

MSDL [8] was produced to aid in the transfer of scenes between research applications which belonged to widely differing branches of graphics work. Consequently much of the initial development was concerned with identifying those properties which were common to all the identified branches (principally radiosity, path tracing, animation & surface manipulation), and those which were too specific to be of common use. The publicly available versions of the language have reflected this concern, though in the latest version facilities have been added to enable application specific instructions to be added to files in a manner which won't prevent the bulk of the scene being used elsewhere.

Text within MSDL files may be freely arranged, as the parser uses lex and yacc to parse the scene. Instructions within MSDL scenes tend to fall into one of two categories, either definition or instantion of items (where items include cameras, lights, geometric objects and transformations). Consequently the language allows items to be defined and named.

MSDL files contain a wide variety of primitives, including polygons, triangle, spheres, cones and NURBS. Unusually MSDL presents the parsed scene to an application in a manner which doesn't impose any decisions on the format of scene datastructure, and so the language designers were free to disregard any computational efficiencies in attempting to produce a language which was convenient to write, rather than quick to read into a particular application. However, as previously described, these mechanisms break down for even moderately sized files, and so the language is only partly successful.

4. Object Oriented Scene Languages

The concept of object based working, where the code developer deals with objects which have defined properties and purposes with a level of abstraction suitable to the problem, maps cleanly to the concept of a geometric object. However, such a trivial mapping forces the scene developer to concern him or herself with the graphics primitives that the reading application can support (as in VRML 1.0 [11]). Consequently we use the term object (in this context) to refer to a collection of primitives which *together* constitute some item meaningful at a scene description level.

The object oriented paradigm encompasses a great many programming mechanisms, including transparent message passing, data hiding, dynamic call resolution, persistence and active objects, but we will concentrate our efforts on applying a subset of the functionality, principally static inheritance.

This operates by allowing a programmer to define objects in terms of other objects, i.e., “this button in the dialog box is the same as that one, except....”. This inheritance is static, as the effect of the inheritance is not dynamically alterable. This adapts quite cleanly to scene description, and allows us to construct scenes using such abstractions as “This chair is the same as that one, except its legs are wooden”.

This improvement is largely due to us grouping primitive data items, and in order to make full use of this we can also add the OO abstraction of ‘object construction’, which is essentially just a way of defining default values for each primitive within the object.

4.1. Object Oriented MSDL

To add this basic object oriented functionality to MSDL we need only make some minor, yet powerful, extensions. These are

- The strong binding of the ‘name’ of an object to its definition. The current MSDL parser attaches the (optional) name of objects to their data, but this name plays no part in the parsing process (so multiple objects may have the same name). We therefore use the name to search for whether we are ‘overriding’ another object we have previously defined. This means that the user must either specify the name of a primitive, or rely on the default name (the rules for the creation of which we must then publicise).

- We add an ‘=’ operator to allow these names to be changed (so particular names to instances of previously defined objects may be imposed).

- The addition of scope to name searching. When we define an object composed of primitives (a compobj), then we only search this object to see whether we need to override previously defined objects.

- Default values are defined for each of the primitives. When a primitive is defined then, either these values (which we must therefore publish) are copied, or the user supplied values override them.

- The ability to define composite objects in terms of previously defined objects. When an object such as this is instantiated the hierarchy is traced, and all the component graphics primitives are copied, with any overriding being performed. We use the colon to indicate which objects we are inheriting from.

Given these basic extensions we may construct scene definitions which resemble the following extract

```
defobj chair_legs
   cylinder leg1 ....
   cylinder leg2 ....
   cylinder leg3 ....
   cylinder leg4 ....
endobj
```

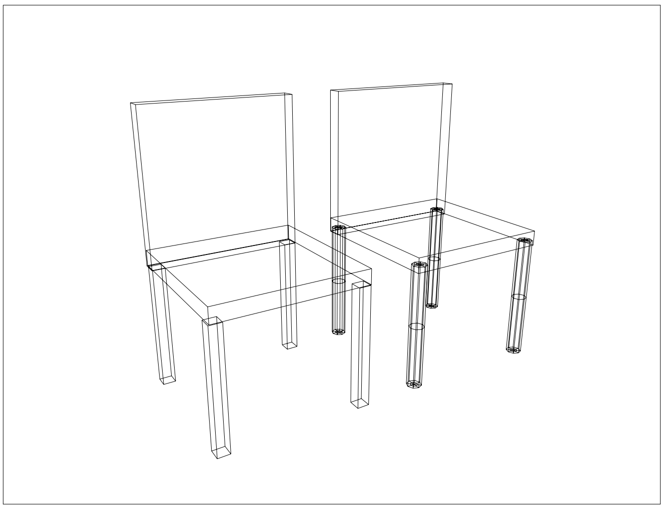

Figure 1 *On the right we see an* instobj*'ed* chair*, and on the left a* square_legged_chair

```
defobj chair
   back = instobj chair_back; /* Previously defined */
   seat = instobj chair_seat;
   legs = instobj chair_legs;
endobj

defobj square_legs
   cube leg1 ...
   cube leg2 ...
   cube leg3 ...
   cube leg4 ...
endobj

defobj square_legged_chair : chair
   legs = instobj square_legs
endobj
```

Here we first define the chair in terms of a number of basic primitives (one of which, the legs, actually consists of more primitives). We can then define a new chair, which shares many of the features of the existing chair, except a set of legs are used to override the previous definition. The effect of this code is shown in Figure 1.

5. Reusable Procedural Definition

Existing scene description files have already used some form of procedural description, but as previously discussed, these have principally been intended to assist in constructing repetitive structures. Here we wish to use procedural definition to construct reusable, parametrisable, scene constructs. The need to be able to parameterise our procedures implies that they are capable of deciding what object primitives are created dependent on context, and so requires

Table 1 Basic variables defined for each object

Variable	Meaning
centre	x,y & z of the defined centre of the object
max	Maximum x,y & z of the bounding box
min	Minimum x,y & z of the bounding box
rot	The Euler rotation that the body is under

the addition of the concept of 'state'.

This is achieved as follows. First, during the scene reading process, a collection of objects are defined and then instantiated. When the objects are instantiated their values are added to the collection of variables which the parser will use. For example, if we define a primitive sphere called "Snooker_ball", then the scene parser will create a number of variables like Snooker_ball.radius, and Snooker_ball.centre.x which it will later use.

As part of the scene language a number of procedures are defined. These procedures consist of excerpts from a simple programming language, which instantiate objects dependent on both the parameters passed to the procedure, and the content of previously defined variables. So, for example, we might write a piece of code which creates a snooker ball to the left of another, which is passed to the procedure as Parameter1:

```
instantiate object "Snooker_ball"
Snooker_ball.centre.x = Parameter1.centre.x - Parameter1.radius
```

Given such constructs it is possible to define quite elaborate scene description mechanisms. For example, in defining a room we may wish to arrange a number of chairs along a wall. With existing scene description languages we would have to measure the wall ourselves, and manually create each of the objects. With procedural definition it is possible to define a procedure which replicates objects along the length of some other object, and this same procedure may be used to align any primitive along any other primitive.

5.1. Procedural MSDL

During the scene parsing process we create, for each primitive *instantiated*, a collection of variables. There are a number of variables which are created for all primitive types (shown in Table 1), and a number of variables which are specific to the primitive, such as the radius of spheres. Where primitives form part of objects (as described in Section 4) we may access them as such.

We also allow the scene author to define a procedure (using the defproc command) and execute it using the instproc instruction. The excerpts of code are written in a simple language based on Tcl [7], which contains instructions to allow

- Definition and modification of variables which are local to the procedure
- Looping using these local variables
- Conditions based on local or global variables, such as those defined for previously

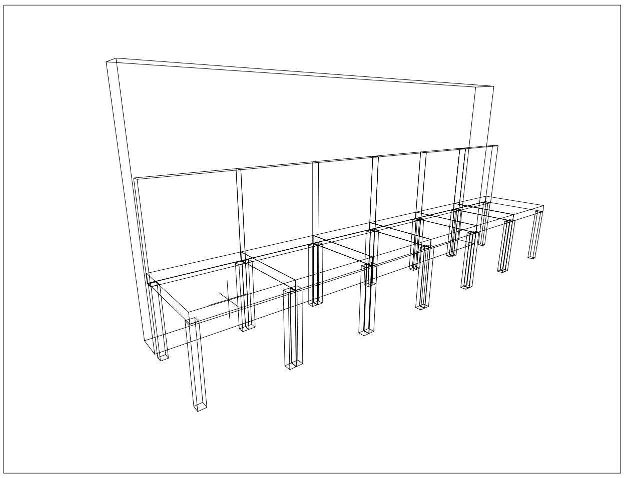

Figure 2 *Here the procedure* arrange_along *replicates a chair along a wall.*

instantiated primitives.

- Instantiation of any previously defined object or primitive.
- Examination and modification of any variables defined for any previously instantiated primitive.

Parameters passed to these procedures must either be constants, axis names, or names of previously defined or instantiated objects. This allows the following extract of MSDL to be produced:

```
defproc arrange_along item alongside
    set number [expr (alongside.max.x - alongside.min.x) /
                            (item.max.x - item.min.x)]
    set number [expr floor($number)]
    for {set i = 0} {i < number} {incr i}
        instobj item
            set object.x [expr $position.x + ($i*$object.max.x)]
            ......
        endinst
endproc

instproc arrange_along chair wall
```

This piece of code defines a procedure called arrange_along, which takes two parameters, firstly the item which it replicates (which must be a defined object in MSDL), and the object that it will range alongside. Inside the procedure a piece of Tcl code calculates how many chairs it can fit along the X axis space occupied by alongside, and replicates the objects using the MSDL instobj instruction. For clarity the code to calculate where to place the row has been removed.The effect of this code is shown in Figure 2.

6. Functional Decomposition

The functional programming paradigm [12] allows us to attain the goal of writing scene descriptions which convey meaning rather than implementation issues. Decomposition of the scene construction process allows us to define simpler and more meaningful functions which achieve a small understandable goal. By providing a range of functions, with the ability to define new functions within a scene, it becomes possible to combine these functions into more elaborate processes.

As with procedural definition, parameters may be provided to the functions which allow creation dependant on context, as well as specific scene requirements. An example of this, is the function which generates a row of objects, with the parameters being: the geometric object to repeat, the number of instances required, a direction vector to distribute the objects along, and a value for the spacing between objects.

By using partial (or *curried*) functions, a function may be passed as a parameter to another function which will provide further parameters, once some computation has been performed. Extending the earlier example, take a function which generates a row of objects which will fit along the length of another given object (e.g., a wall.) Although, the function understands the concept of a row, and hence how the objects are positioned along it, it does not need to recreate the process of generating the row. Instead it calculates the number of objects that can fit and the appropriate spacing; these parameters are then appended to the original row_of parameters, hence returning a more specialised function.

In addition, functional languages easily provide the basis for overloading of datatypes, and hence geometric primitive types. One common function required for all geometric types, is the bounds function. By associating a function with each geometric type, scene constructor functions can call the appropriate bounds function for whatever type has been passed. The result of this function is a known datatype, and hence the function is independent of geometric types.

In conclusion, a functional language paradigm, provides a more flexible approach than standard procedural definition, as complex composite functions can be developed easily, due in part to the confidence instilled by using previously developed functions, and not re-implementing them for every new function. One drawback though is when a required function is very complex: this is better suited to a procedural paradigm, as it may be easier to develop code to perform a specific operation, than trying to combine many simpler operations to achieve the same result. However, doing this produces a function which is not readily reusable within the functional environment.

6.1. Functional Definition in MSDL

In order to incorporate the principal features of functional programming as part of MSDL we need:

- The ability to define functions, which take named parameters which may include other functions (deffun .... endfun).
- Each instance of these functions produces a composite object, so any primitives instanced by a function are grouped together. This means that such functions can be thought to return a composite group.
- These parameters may be typed loosely (in which case all geometry types will match) or typed specifically (i.e., a scene author may wish to define a function which operates on NURBS)

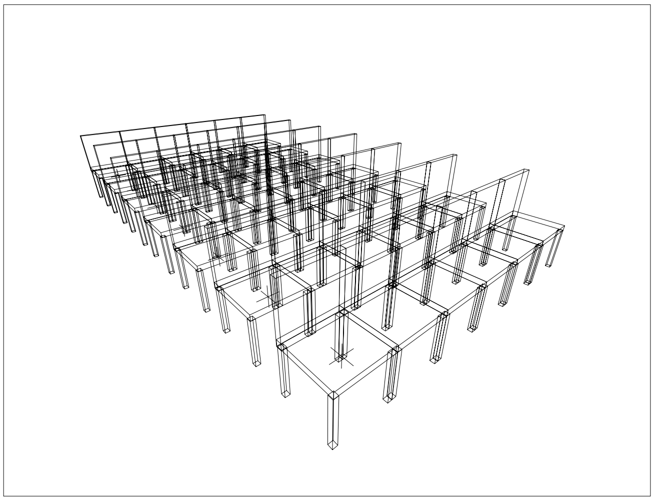

Figure 3 *Using the function* row_of() *we can create multiple copies of a single object.*

- Many of the procedural programming features outlined in the previous section are required, such as the ability to modify object variables.
- The ability to instantiate these functions as part of the scene description, with the relevant type-checking, and execution of nested functions (instfun). This use of an explicit instantiation token fits in with the MSDL style, but may not be necessary in other scene languages.

Having identified the new commands added to MSDL it is now possible to see how they might be used in a larger example, in this case the definition of the seating arrangement in a large seminar room.

```
defobj conference_chair
    ..... /* Define the chair using normal MSDL primitives */
enddef

deffun row_of (any_object parameter, direction, number, spacing)
    for {set i = 0} {i < number} {incr i}
        instobj parameter
            set object.x [expr $i*direction.x*spacing.x]
        endinst
endfun

seating_block = instfun row_of(instfun(instobj conference_chair, X, 5,0.0),
                                         Y, 7, 1.0)
```

Here we have defined a simple object, our conference chair, and a function row_of(), which takes the object that it manipulates, a direction that the user wants the object to be repeated in, and the number of repetitions required. We create a composite object which contains 7 composite objects, who are arranged in Y, each of which contains 5 composite objects, each of which contains a chair. The result of executing this piece of scene description is shown in Figure 3.

7. Conclusions

Scene description languages, whilst normally conceived as a quick solution to the problem of scene interchange, are frequently used as the graphics researchers primary means of scene definition. Unfortunately existing scene languages tend to be unsuitable for use in maintaining all but the most trivial models. This paper has presented a number of ways of addressing this problem, by identifying the similarity of scene description to programming. Reusable procedural definition, object orientation and functional definition have all been investigated for use in constructing more pleasant SDLs. This initial research leads us to believe that these mechanisms may ease scene maintenance, and our current work is concerned with evaluating them for use with realistic scenes.

Acknowledgements

We would like to thank the staff and students of Manchester's Computer Graphics Unit for their support, and all the various people who have contacted us with queries or suggestions about MSDL over the years. Your assistance has been much appreciated!

References

[1] CICA 3D File Formats Page. *http://www.cica.indiana.edu/graphics/3D.objects.html*

[2] S. Donikian and G. Hegron. A Declarative Design Method for 3D Scene Sketch Modeling. *Computer Graphics Forum* (Proceedings of Eurographics '93) 12(3):223-236.

[3] DXF File Format. *http://www.cica.indiana.edu/graphics/object_specs/DXF.format.txt*

[4] MGF File Format. *ftp://hobbes.lbl.gov/www/mgf/HOME.html*

[5] OBJ File Format. *http://www.cica.indiana.edu/graphics/object_specs/OBJ.format.ps*

[6] NFF File Format. *http://www.cica.indiana.edu/graphics/object_specs/NFF.format.txt*

[7] J. Ousterhout. Tcl and the Tk Toolkit. Addison-Wesley, 1994.

[8] M.Preston, N.Gatenby, W.T. Hewitt. The Manchester Scene Description Language V1.1. *http://info.mcc.ac.uk/CGU/MSDL/*

[9] P.S. Strauss and R. Carey. An Object Oriented 3D Graphics Toolkit. *Computer Graphics* 26(2): 341-349.

[10] S. Upstill. The Renderman Companion. Addison-Wesley.

[11] VRML File Format. *http://www.eit.com/vrml/vrmlspec.html*

[12] Åke Wikström. Functional Programming Using Standard ML. Prentice-Hall. 1987