Manchester Computing Centre
Computer Graphics Unit

UMIST

The Manchester Scene Description Language (MSDL) Version 1.01

Neil Gatenby
Martin Preston

Department of Computer Science, University of Manchester, UK

W.T. Hewitt

Computer Graphics Unit, Manchester Computing Centre, UK

Version 1.01 | CGU 88 | 29^{th} November, 1993

Contents

1 Introduction

The University of Manchester Computer Graphics Unit serves as a centre for computer graphics research, and over the years many novel graphics packages have been developed here. However as each researcher develops something, he or she is forced to cobble together a way of telling their application exactly what it is they want a picture of. Consequently a huge number of ways have been devised of positioning mirrored spheres over chess boards!

In an attempt to reduce the duplication of effort, the CGU research group developed a scene language flexible enough to cope with the requirements of most graphics research programs. The language allows users to create a text file containing a description of a 3 dimensional scene, which could then be read by all users of the language.

To encourage its use a parser was developed to actually read the language files, thereby minimising the amount of effort a programmer needs to expend to use a 'standard' solution. The parser has been in use for some time now, and has proved a useful tool in graphics research.

This report outlines the *Manchester Scene Description Language* (MSDL), designed for describing scenes which are to be used by a user's graphics application package. This document consists of a description of the language, a discussion of some example scenes, an outline of the process of generating the MSDL parser, and information on linking the parser to a users application. Included with the MSDL kit is a `README` file which includes a list of the distributions contents, and information on building the parser for different platforms.

The parser is built using `lex(1)` and `yacc(1)` and consists of a C or C++ library. The users application calls a function in this library to begin reading an MSDL file. As the parser reads the file it calls functions to generate object primitives in the scene. These functions are supplied by the user and placed inside his or her application. Consequently the application is not reliant on the type definitions used internally by the parser, and can define its own, copying the information provided by the reader into them.

MSDL is available principally by anonymous ftp from the Manchester ftp site `ftp.mcc.ac.uk` (`130.88.203.12`), together with some sample MSDL scenes, and other Manchester sourced software. Information on the CGU, MSDL, other products and current research is available on the WorldWideWeb pages at `http://info.mcc.ac.uk/CGU/MSDL/MSDL-intro.html`.

MSDL has been made available for use in the academic community. If you wish to use it for other purposes please contact us. It is supplied *as-is*, we do not guarantee its suitability for any particular use, and is not a supported product of the University of Manchester. However as we use it ourselves we would be interested in your comments and criticisms, which should be emailed to the Manchester general queries address, `cgu-info@mcc.ac.uk`, preferably with MSDL in the subject field.

2 The Manchester SDL

Several scene description languages are in current use, and some of the features of these influenced the design of MSDL. Most of these file formats are keyed to the peculiarities of particular applications, and as MSDL was designed to be primarily used for research purposes no effort was made to make it compatible with these application languages.

- Greg Ward's *Radiance* [5].

- Craig Kolb's *Rayshade.4.0* [4].
- David Buck's *DKBTrace, version 2.12* [1].
- Meiko Ray-Tracer's SDL.
- Eric Haines' *Neutral File Format (NFF)* [2].
- Stefan Müller's extension to NFF.

Possibly the most popular of the non-application specific file formats examined was NFF. This was written with parsing to a local (more complex) file format in mind. Syntax that is close to NFF's is used where it is convenient.

2.1 An overview of MSDL

This section outlines the most important features of MSDL, leaving a more comprehensive treatment of the format for later in this document.

We feel that an important logical separation should be made between the following purposes of any SDL:

- Describing the objects that make up the scene.
- Viewing the scene and specifying rendering parameters that will affect *how* the scene is rendered.

This version of MSDL is concerned primarily with the first goal. However we feel that a 'sample' camera definition should be included with most MSDL files, as without it the user may find it unnecessarily difficult to determine a reasonable view, given that he or she may not know what scale the scene is defined with.

To that end MSDL allows an optional view definition, which the application may ignore if it wishes.

MSDL files are plain text files, designed to be created both by users and their applications. Consequently it is possible to process MSDL files using other programs. Using pre-processors such as `cpp` & `m4` the user can build up libraries of standard objects, which are `#include`'ed into specific scenes. However, for flexibility, we have included the ability to place C-style comments inside MSDL files without having to use a pre-processor.

The ability to assemble standard libraries of objects leads to the notion of three ways to describe an object, each of which has a language keyword associated with it:

defobj – Define an object, but *do not* instantiate it.

object – Define an object, *and* create an instance of it.

instobj – Instantiate a previously defined object.

Other commands allow the user to manage surface properties, and transformations in the same manner.

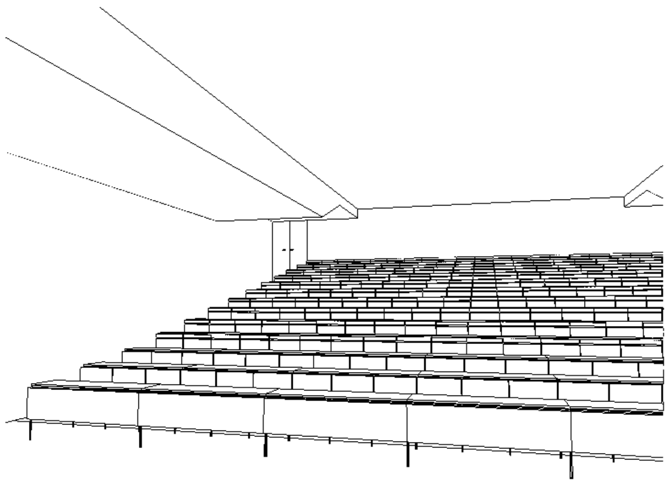

Figure 1: A lecture theatre at Manchester University. The rows and tiers of desks were created by definition and instantiation. Thanks to Richard Andrews for doing all the measuring!

defprops – Define a surface property, but do not instantiate it.

props – Define a surface property, and create an instance of it.

instprops – Instantiate a previously defined surface property

deftrans – Define (only) a transformation

trans – Define and instantiate

insttrans – Instantiate

This define and instantiate approach allows an MSDL user to easily create scenes which contain large numbers of repeated objects. Figure 1 shows a view of an MSDL file representing a lecture theatre in the Computer Science department of Manchester University. To generate it a model for a single seat was constructed, which was instantiated to form a row, and several rows were instantiated to form the seating for the theatre.

A users application employs MSDL by calling a function provided inside the parser supplied in this package. This function (`msdl()`) loads an MSDL text file into its internal store. It then processes the scene, to take into account the effects of instantiating objects, to determine the collection of objects which the application needs to know about.

The parser then goes through each of these objects, calling a corresponding `create()` function. These functions are supplied in the distribution as skeleton code. When the user writes his or her application they fill in these skeleton routines, to perform something meaningful to their application.

For example MSDL scripts can contain NURBS surfaces. However some applications may not be able to cope with these, and so will wish to convert them to something convenient for their internal use. So the application programmer could write a NURBS tessellation routine, which is called inside the `create_nurb()` function, to generate something for the applications use.

This process is described graphically in figure 2 and covered in more detail in section 5.

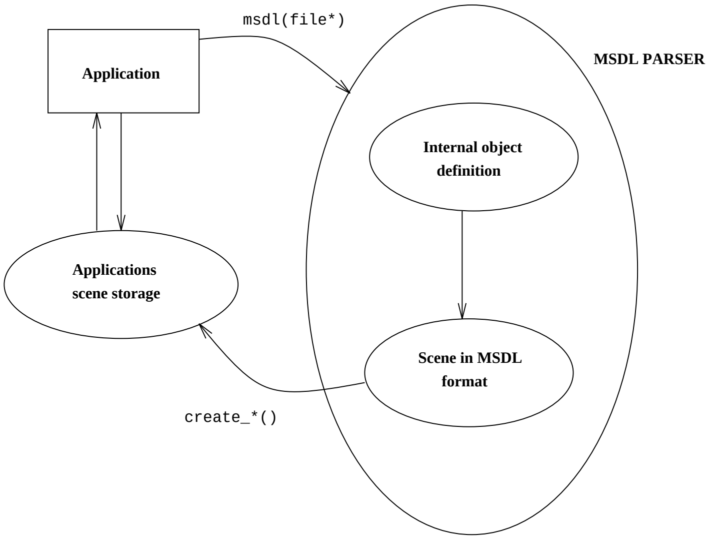

Figure 2: An overview of how an application might use the MSDL parser. The application calls the parser function, which in turn affects the application scene store by calling functions which the application must provide.

Finally it is worth pointing out that the MSDL parser passes the information contained in the script onto the reading application. If the application chooses to use the data in the file for different purposes (for example an application might want to use the specularly reflecting information for something else) then the parser will be none the wiser. This means that users wishing to generate portable MSDL files have to ensure their applications uses the information consistently.

The next section discusses the way in which a user describes a scene in the MSDL language. Following that is a short section on generating some sample scenes. The final sections of the report cover the information an application programmer wishing to link the MSDL parser into his or her system needs to know.

2.2 Objects

2.2.1 Object Primitives

MSDL allows the user to define objects which can be grouped, transformed and have properties attached to them. These objects are defined in terms of primitives, which can be polygons, triangles, spheres, cylinders, cones, discs, boxes, NURBS and polyhedra.

This section consists of a definitions of the different primitives, followed by a BNF syntax. Examples of how on might create one of the primitives is given in typewritten font.

Throughout this report, keywords are shown in **boldface**, parameters are delimited by < .... >, and both are separated by *any* white space characters. Within parameter brackets the type of data expected is either defined, or should be self-apparent.

- **polygon**

  $< \text{nverts} \in \{3, 4, 5, 6, \ldots\} >$
  $< \mathbf{P_1} \in \mathcal{R}^3 >$
  ...
  $< \mathbf{P}_{\text{nverts}} \in \mathcal{R}^3 >$
  < ...options... >

  end

  – Defines a polygon with *nverts* vertices. The vertices should be provided in counter-clockwise order when viewing the *front* of the polygon (Picture a right-handed screw being screwed into the back of the polygon, turned in the direction the vertices were defined – the screw will point along the polygon normal). Also, the angle subtended at $\mathbf{P_2}$ by $\mathbf{P_1}$ and $\mathbf{P_3}$ should be non-zero and convex (i.e., $\theta \in (0, \pi)$), so that the polygon normal can be found using a cross product. (This is consistent with NFF).

  We might define a simple polygon by writing :

```
polygon
4
0.0 0.0 0.0
1.0 0.0 0.0
1.0 1.0 0.0
0.0 1.0 0.0
end
```

The < ...options... > mentioned in the syntax will be described momentarily. First, let's look at a special case of **polygon** which the user can define in an effort to simplify their MSDL files, and perhaps ease the improvement of internal storage and manipulation methods in their implementation:

- **triangle**

  $< \mathbf{P_1}\ \mathbf{P_2}\ \mathbf{P_3} \in \mathcal{R}^3 >$
  < ...options... >

  end

  – Defines a triangle with vertices $\mathbf{P_1}, \mathbf{P_2}, \mathbf{P_3}$. The vertices should be given in counter-clockwise order when viewing the *front* of the triangle, so that the correct patch normal can be found using a cross product. (Again this is consistent with NFF.)

Again, we see < ...options... > appearing in the syntax: There are, in fact, several optional parameters; not all of which are applicable to all objects. They are described in greater detail in the grammar description given later. Two options unique to **triangle**, **polygon**, and **polyhedron** are:

vnorm $< \mathbf{N_1} \ \ldots \ \mathbf{N_{nverts}} \in \mathcal{R}^3 >$

– This associates the given surface normals with the previously given points. These need not be unit vectors.

fnorm $< \mathbf{N} \in \mathcal{R}^3 >$

– This associates the given *facet* normal with the polygon (or polyhedron face). If this is supplied, it will override any 'cross-product' calculation that may be carried out to evaluate the normal.

So if we were defining a triangle, with the vertices normals define we could write something like the following :

```
triangle
0.0 0.0 0.0
5.0 0.0 0.0
0.0 3.0 0.0
vnorm 0.0 0.0 1.0
      0.0 0.0 1.0
      0.0 0.0 1.0
end
```

Let's take a look at the rest of the object primitives available:

- **sphere**
  $< centre \in \mathcal{R}^3 >$ $< radius \in \mathcal{R}^+ \cup \mathcal{R}^- >$
  $[$**up** $< \mathbf{U} \in \mathcal{R}^3 >]$
  $[$**tpz** $< \mathbf{V} \in \mathcal{R}^3 >]$
  $[$**theta** $< start, fin \in [0,1] >]$
  $[$**phi** $< start, fin \in [0,1] >]$
  < ...options... >
  end

  – A sphere with the given centre and radius. Note that the radius can be positive or negative, but not zero. A negative radius results in a sphere whose radius is the modulus of that given, but whose surface is inward-facing. Again, this is NFF-consistent.

  The vectors **up** and **tpz** (Theta, Phi Zero) describe, respectively, the 'North pole' and the point on the 'equator' where *theta* (cf. longitude) is zero(2π). If not given by the user, **up** is taken to be **k**, and **tpz** is taken to be **i** [unit vectors lying along the world coordinate axes: **i**, **j**, **k**]. Note that these are *directions*, not positions on the surface of the sphere. An error will result if $\mathbf{up} \cdot \mathbf{tpz} \neq 0$.

  The keywords **'theta'** and **'phi'** are used to define part-spheres (hemispheres, 'dish' shapes, 'Edam-cheese-with-a-slice-missing' shapes, and so on). The two reals given after each keyword refer to 'how much we sweep out' along the corresponding direction; Θ increases around the equator (0 and 1 at **tpz**), whereas Φ is minimum(0) at the 'South pole' and maximum(1) at the 'North pole'. To clarify; a hemisphere might be defined by:

  ```
  sphere name lampshade
  16.0 20.0 18.77
  3.2
  phi 0.5 1.0
  instprops Transp_light_blue
  end
  ```

  Or, equivalently:

  ```
  sphere name lampshade
  16.0 20.0 18.77
  3.2
  up  0 0 -1
  phi 0 0.5
  instprops Transp_light_blue
  ```

Filling in missing chunks of sphere that will result when the user starts to manipulate Θ and Φ is the user's problem!

Of course many applications may not be able to cope with all the possible shapes one can generate with this primitive. An application should choose a meaningful behaviour if it finds something it can't cope with, e.g., create as much as it can, then display a warning.

- **cylinder**

  $< bottom \in \mathcal{R}^3 >$ [**capped**]
  $< top \in \mathcal{R}^3 >$ [**capped**]
  $<$ radius $\in \mathcal{R}^+ \cup \mathcal{R}^- >$
  [tz $< \mathbf{V} \in \mathcal{R}^3 >$]
  [theta $< start, fin \in [0,1] >$]
  $<$ ...options... $>$

  end

  – A right circular cylinder, extending between the points **bottom** and **top**, with the given cross-sectional radius. The same negative radius rule applies; inward-pointing for negative. The optional keyword **'capped'** indicates whether an endcap should be added to the end of the cylinder associated with the keyword – so **'capped'** might appear 0,1 or 2 times in a **cylinder** definition.

  The vector **tz** (Theta Zero) describes the point on the cylinder's rim where *theta* (cf. longitude) is zero(2π). An error results if $(\mathbf{top} - \mathbf{bottom}) \cdot \mathbf{tz}$ is zero. If not given by the user, **tz** is evaluated as follows:

  1. Find the unit normal along $\mathbf{n} = \mathbf{top} - \mathbf{bottom}$.
  2. Find which of $\mathbf{n}$'s 3 components has the smallest magnitude. Note the unit vector ($\mathbf{l} = \mathbf{i}$, $\mathbf{j}$ or $\mathbf{k}$) which corresponds to this axis.
  3. $\mathbf{tz} = \mathbf{n} \wedge \mathbf{l}$ (normalized).

  Again, filling in missing chunks of cylinder that will result when the user starts to manipulate Θ is the user's problem.

  An example cylinder is shown in section 3.

- **cone**

  $< bottom \in \mathcal{R}^3 >$ [**capped**]
  $< \mathrm{rad}_{\mathrm{bot}} \in \mathcal{R} >$
  $< top \in \mathcal{R}^3 >$ [**capped**]
  $< \mathrm{rad}_{\mathrm{top}} \in \mathcal{R} >$
  [**tz** $< \mathbf{V} \in \mathcal{R}^3 >$]
  [**theta** $< start, fin \in [0,1] >$]
  $<$ ...options... $>$

  end

  – A right circular cone, extending between the points **bottom** and **top**, with the given cross-sectional radii at its ends. The same negative radius rule applies as did for cylinders and spheres. Note that the radii can now take the value zero – i.e., cones which come to a point *are* allowed. If ABS($\mathrm{rad}_{\mathrm{bot}}$) differs from ABS($\mathrm{rad}_{\mathrm{top}}$) then an error results (unless one radius is zero), resulting in a call to one of the error handling routines outlined in section 5.4.

  The keywords **tz** and **theta** serve the same purpose as they do in the cylinder description

- **disc**

  $< centre \in \mathcal{R}^3 >$ $< radius \in \mathcal{R}^+ >$ $< normal \in \mathcal{R}^3 >$
  [$\mathbf{tz} < \mathbf{V} \in \mathcal{R}^3 >$]
  [$\mathbf{theta} < start, fin \in [0,1] >$]
  $<$ ...options... $>$

  end

  – A single-sided flat circular disc, with the given midpoint and radius, orientated according to the given normal vector. The normal vector need not be a unit vector. This has been included largely for compatibility with NFF.

  The keywords **tz** and **theta** mean the same as they did in the cylinder definition, except that now we don't have to evaluate the normal, it is given.

- **box**

  $< bl \in \mathcal{R}^3 >$ $< tr \in \mathcal{R}^3 >$
  $<$[**in**]$>$
  $<$ ...options... $>$

  end

  – An axis-aligned cuboid, with 'bottom-left' and 'top-right' vertices as defined. The optional keyword **in** is used if the user wishes to define the box's surface normals as inwards-pointing. The default is outwards-pointing.

  An example, an useful, box might then be defined

```
defobj unit_cube
    box
      0.0 0.0 0.0 1.0 1.0 1.0
    end
end
```

- **nurbs**

  order $< \text{ord}_u\ \text{ord}_v \in \mathcal{N} >$
  cpts $< \text{ctrl}_u\ \text{ctrl}_v \in \mathcal{N} >$
  knots
  $\{< \text{Uknot}_1 \ldots \text{Uknot}_{ord_u + ctrl_u} \in \mathcal{R} > \}$
  $\{< \text{Vknot}_1 \ldots \text{Vknot}_{ord_v + ctrl_v} \in \mathcal{R} >\}$
  $< P_{1,1} \in \mathcal{R}^3 \text{x} \mathcal{R}^+ >$
  $\ldots$
  $< P_{\text{ctrl}_u,1} \in \mathcal{R}^3 \text{x} \mathcal{R}^+ >$
  $< P_{1,2} \in \mathcal{R}^3 \text{x} \mathcal{R}^+ >$
  $\ldots$
  $\ldots$
  $< P_{\text{ctrl}_u,\text{ctrl}_v} \in \mathcal{R}^3 \text{x} \mathcal{R}^+ >$
  $<$ ...options... $>$

  end

  – This defines a Non-Uniform Rational B-Spline surface. The user must provide the order of the NURBS (in both dimensions), its knot vectors, and its control points. The control points entered u direction first, so if we think of the control point list as an array, the u direction would form the rows, and the information would be entered into the MSDL file as lists of rows.

  You may find the Manchester NURBS library [3] useful in using NURBS in your application. Although MSDL itself does not require this library most NURBS applications will need it!

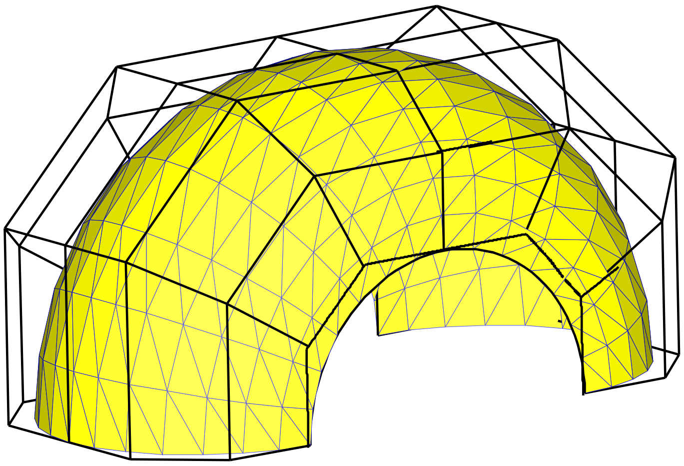

Figure 3: A sample NURBS surface

An example NURBS surface, in this case representing a sphere, would then look like:

```
/* A NURBS sphere, generated by Manchester NURBS library */
nurbs
   order 3    3
   cpts  9    5
   knots
{  /* First the knots for the U direction */
             0
               0
               0
            0.25
            0.25
             0.5
             0.5
            0.75
            0.75
               1
               1
               1}
{
/* Then the knots for V */
               0
               0
               0
             0.5
             0.5
               1
               1
               1}

               0               0           -0.25               1
               0               0       -0.176777        0.707107
```

```
               0           0         -0.25           1
               0           0     -0.176777    0.707107
               0           0         -0.25           1
               0           0     -0.176777    0.707107
               0           0         -0.25           1
               0           0     -0.176777    0.707107
               0           0         -0.25           1
        0.176777           0     -0.176777    0.707107
           0.125       0.125        -0.125         0.5
               0    0.176777     -0.176777    0.707107
          -0.125       0.125        -0.125         0.5
       -0.176777           0     -0.176777    0.707107
          -0.125      -0.125        -0.125         0.5
               0   -0.176777     -0.176777    0.707107
           0.125      -0.125        -0.125         0.5
        0.176777           0     -0.176777    0.707107
            0.25           0             0           1
        0.176777    0.176777             0    0.707107
               0        0.25             0           1
       -0.176777    0.176777             0    0.707107
           -0.25           0             0           1
       -0.176777   -0.176777             0    0.707107
               0       -0.25             0           1
        0.176777   -0.176777             0    0.707107
            0.25           0             0           1
        0.176777           0      0.176777    0.707107
           0.125       0.125         0.125         0.5
               0    0.176777      0.176777    0.707107
          -0.125       0.125         0.125         0.5
       -0.176777           0      0.176777    0.707107
          -0.125      -0.125         0.125         0.5
               0   -0.176777      0.176777    0.707107
           0.125      -0.125         0.125         0.5
        0.176777           0      0.176777    0.707107
               0           0          0.25           1
               0           0      0.176777    0.707107
               0           0          0.25           1
               0           0      0.176777    0.707107
               0           0          0.25           1
               0           0      0.176777    0.707107
               0           0          0.25           1
               0           0      0.176777    0.707107
               0           0          0.25           1
end
```

- **polyhedron**

  {
  $< \mathbf{V_1} \in \mathcal{R}^3 >$
  $\vdots$
  $< \mathbf{V}_{\mathrm{verts}} \in \mathcal{R}^3 >$
  }
  [
  <**vnorm**>
  $< \mathbf{N_1} \in \mathcal{R}^3 >$
  $\vdots$
  $< \mathbf{N}_{\mathrm{verts}} \in \mathcal{R}^3 >$
  ]
  {
  [$<$ vlist$_1$ $>$]
  $\vdots$
  [$<$ vlist$_{\mathrm{faces}}$ $>$]
  }
  [
  <**fnorm**>
  $< \mathbf{M_1} \in \mathcal{R}^3 >$
  $\vdots$
  $< \mathbf{M}_{\mathrm{faces}} \in \mathcal{R}^3 >$
  ]
  $<$ ...options... $>$

  end

  – Used for objects not conveniently described by the other primitives. The user provides the vertices, ($<$ verts $>$ of them), that make up the object, between a pair of braces ({ }). The order in which the vertices are given is important – the polyhedron face information will reference the first vertex listed as 'vertex 1', the second as 'vertex 2', and so on.

  After the vertices, the user gives the $<$ faces $>$ faces that make up the surface of the object. This face information is a list of vertex-lists; each vertex-list describing one face and consisting of a list of integers in the range [1...verts] which reference vertices given in the vertex list. These lists should give the vertices in a counter-clockwise order as one looks at the face, and the angle subtended at the second vertex by the first and third vertices should be non-zero and convex. Each vertex-list (describing one face) is delimited by a newline character in the MSDL file at the end of the list. The list of vertex-lists (describing the whole polyhedron) is delimited by a pair of braces.

  Note that the options **vnorm** and **fnorm** can be used with the polyhedron primitive. An ordered list of $<$ verts $>$ vectors following the keyword **vnorm** will be taken as the normals associated with the listed vertices. An ordered list of $<$ faces $>$ vectors following the keyword **fnorm** will be taken as the normals associated with the described faces.

  A simple example of a polyhedron would be that of a box,

```
/* A unit box represented as a polyhedron */
polyhedron
{
  0.0 0.0 0.0
  1.0 0.0 0.0
```

```
    1.0 1.0 0.0
    0.0 1.0 0.0
    0.0 0.0 0.0
    1.0 0.0 0.0
    1.0 1.0 0.0
    0.0 1.0 0.0
  }
  {
    [ 1 2 3 4 ]
    [ 2 3 7 6 ]
    [ 1 4 8 5 ]
    [ 3 4 8 7 ]
    [ 6 7 8 5 ]
    [ 1 2 6 5 ]
  }
  end
```

2.2.2 Object Transformations

As mentioned earlier, the user can apply transformations to any of the objects. The transformations will be stored hierarchically – i.e., they should be applied to all objects or object primitives that make up the *objects* block in which they appear (see the BNF grammar, and section 2.2.3). They can also be defined (named) and instantiated elsewhere in the MSDL file.

The various transformation options are outlined here:

shift $< vec \in \mathcal{R}^3 >$

– Translates (shifts) the object by the given vector.

rotate $< \mathrm{char} \in \{x, y, z\} >$ $< \theta \in \mathcal{R} >$

– Rotates the object about the given axis (about the origin), through θ degrees, e.g., "rotate x 90".

scale $< vec \in \mathcal{R}^3 >$

– Scales the object by the specified amount (about the origin) along the corresponding axes.

matrix

$< \mathbf{row_1} \in \mathcal{R}^4 >$
$< \mathbf{row_2} \in \mathcal{R}^4 >$
$< \mathbf{row_3} \in \mathcal{R}^4 >$
$< \mathbf{row_4} \in \mathcal{R}^4 >$

– The given matrix (defined by 16 reals) is to be used to transform the object.

Note that if the user wishes to scale or rotate an object about a particular point $[P_x, P_y, P_z]$, they can shift the object by $[-P_x, -P_y, -P_z]$, carry out the desired scale/rotate and shift it back by $[P_x, P_y, P_z]$.

As mentioned earlier, the option to assign names to (presumably complex) transformations, for ease of use elsewhere in the MSDL, has been included. The precise syntax for this is shown in the BNF grammar in appendix A, but it is directly analogous to the `defprops`, `props` and `instprops` syntax already explained.

An example transformation might look like

```
rotate x 20.0      /* Rotate by 20 degree around X axis */
scale 1.0 2.0 1.0  /* Stretch in y */
```

2.2.3 Composite objects

In creating any complicated scenes you will probably want to perform operations (such as rotations) on groups of objects. To allow you to do this MSDL lets you to group a sequence of objects into composites.

MSDL allows you to group any number of objects into a named group. In addition MSDL composites can contain other composites, to generate a hierarchy. This hierarchy is communicated to the parent application by use of the `create_comp()` and `end_comp()` functions. (See section 5 for more details).

A BNF extract for the a composite object would then look like:

compobjblk ::= **compobj** [*id*] {*options*} {objcentre} **end**.

We might then wish to create a composite object by writing :

```
compobj chair

/* Rotate the entire chair by 90 degrees */
rotate x 90.0

instobj legs
instobj back
instobj bottom
end
```

2.2.4 Object Syntax

We want to define a grammar to specify the syntax of this part of the MSDL. The full grammar for the MSDL can be found in appendix A of this report.

objects	::=	*defobjblk*
	|	*objblk*
	|	*instobjblk*
	|	*compobjblk*
	|	*objprims*.
compobjblk	::=	**compobj** [*id*] {*options*} {objcentre} **end**.
defobjblk	::=	**defobj** *id* *objcentre* **end**.
objblk	::=	**object** [*name*] *objcentre* **end**
	|	*objprims*.
instobjblk	::=	**instobj** *id* *options* **end**.
objcentre	::=	{*instobjblk* } {*objprims* } *options*.
options	::=	[*propsblk* | *instpropsblk*] {*transforms*} [*patchesblk*]
objprims	::=	**triangle** [*name*] R3 R3 R3 [**vnorm** R3 R3 R3] [**fnorm** R3] *options* **end**.
	|	**polygon** [*name*] natural R3 .... R3 [**vnorm** R3 .... R3] [**fnorm** R3] *options* **end**.
	|	**sphere** [*name*] R3 R± [**up** R3] [**tpz** R3] [**theta** R_01 R_01] [**phi** R_01 R_01] *options* **end**.
	|	**cylinder** [*name*] R3 [**capped**] R3 [**capped**] R± [**tz** R3] [**theta** R_01 R_01] *options* **end**.
	|	**cone** [*name*] R3 [**capped**] R± R3 [**capped**] R± [**tz** R3] [**theta** R_01 R_01] *options* **end**.
	|	**disc** [*name*] R3 R+ R3 [**tz** R3] [**theta** R_01 R_01] *options* **end**.
	|	**box** [*name*] R3 R3 [in] *options* **end**.
	|	**nurbs** [*name*] **order** natural natural **cpts** natural natural **knots** { R .... R } { R .... R } R3 × R+ .... R3 × R+ *options* **end**.

	|	**polyhedron** [*name*]
		{ R3 ..... R3 }
		[
		vnorm
		R3 ..... R3
		]
		{
		[natural ..... natural]
		....
		[natural ..... natural]
		}
		[
		fnorm
		R3 ..... R3
		]
		options **end**.

name	::=	**name** *id*.
transforms	::=	deftransblk
	|	transblk
	|	insttransblk
	|	{ transprims }.
deftransblk	::=	**deftrans** id { insttransblk } { transprims } **end**.
transblk	::=	**trans** [name id] { insttransblk } { transprims } **end**.
insttransblk	::=	insttransblk id.
transprims	::=	**shift** R3
	|	**scale** R3
	|	**rotate** axis R
	|	**matrix** R4 R4 R4 R4.
patchesblk	::=	**patches** natural [natural [natural]]

Again, this piece of BNF includes some ambiguous parameters; these are:

- R± – The set of positive and negative real numbers.
- R+ – The set of positive real numbers.
- R4 – The set of 4-vectors with real components.
- axis – The set of characters {x,X,y,Y,z,Z}.
- R – The set of real numbers.

2.3 Surface Properties

2.3.1 Surface Property Primitives

The user needs to be able to associate surface properties with object primitives. As mentioned earlier, one can define three different types of *property* block; one that associates a *name* with a particular set of attributes (**defprops**); one that associates an object (and possibly a name) with a particular set of attributes (**props**); and one that merely instantiates an already existing set of attributes (**instprops**).

Before looking at the syntax of how one goes about defining these sets of surface properties, let's consider the different properties we can assign to surfaces in our scene. Many of these properties should be assigned a default value if the user fails to provide one, but this is implementation dependent, and will need to be supplied by the user:

name < string >

– Associate the given name with the surface properties being defined – for possible instantiation elsewhere.

emission $< \mathbf{E} \in (\mathcal{R} - \mathcal{R}^{-}) >$

– The emissivity of the surface being defined, in Joules $\mathrm{sec}^{-1}\mathrm{m}^{-2}$. Included for radiosity calculations.

arc $< \kappa_{\mathbf{a}} \in [0,1]^3 >$

– The Ambient Reflection Coefficient of the surface.

drc $< \kappa_{\mathbf{d}} \in [0,1]^3 >$

– The Diffuse Reflection Coefficient of the surface.

src $< \kappa_{\mathbf{s}} \in [0,1]^3 >$

– The Specular Reflection Coefficient of the surface.

trc $< \kappa_{\mathbf{t}} \in [0,1]^3 >$

– The Transmission Coefficient of the surface – determines the maximum fraction of the incident light that could pass through the surface. Zero implies opaque, one implies transparent.

phexp $< n \in \mathcal{N} >$

– The power to which the '$\cos\theta$' term is raised if a Phong illumination model is used. This parameter is included in MSDL for convenience, many user applications may not use it.

index $< \eta \in \mathcal{R}^{+3} >$

– The refractive index of the material; a 3-vector is needed as the user may wish to account for its variation with wavelength.

texture [**file**] $<$ *name* $>$

– Assigns a texture to the surface. If the keyword **'file'** is included, then a bitmap found in the named file is used, otherwise the name is that of a procedural texture (e.g., WOOD, MARBLE) which would be implementation dependent.

Objects in MSDL are normally defined in some hierarchy (that is you can group objects, and have groups of groups, and so on). If you assign a surface property to a group it will apply to all the members of the group *EXCEPT* for objects in groups which have that property already defined.

As an example of this suppose we define a table which consists of a top, and four legs. If we define the table top to be white, group all the table together, then assign a 'pine' property to the group, the table top will still be white.

2.3.2 Surface Property Syntax

Again, here is a piece of BNF grammar to specify the syntax of this part of the language: Remember, we should be able to do three things; define only; define and instantiate; instantiate only. (The full grammar for the MSDL can be found in appendix A of this report).

properties	::=	*defpropsblk*
	|	*propsblk*
	|	*instpropsblk.*
defpropsblk	::=	**defprops** *id* {*instpropsblk*} {*propsprims*} **end**.
propsblk	::=	**props** [**name** *id*] {*instpropsblk*} {*propsprims*} **end**.
instpropsblk	::=	**instprops** *id*.
propsprims	::=	**emission** *R3nn*
	|	**arc** *R3_01*
	|	**drc** *R3_01*
	|	**src** *R3_01*
	|	**trc** *R3_01*
	|	**phexp** *natural*
	|	**index** *R3nn*
	|	**texture** [**file**] *name*.
id	::=	*letter* {*alpha*}.
letter	::=	a|b|...|z|A|...|Z.
alpha	::=	*letter*|_|0|...|9.

Explanatory note: The terminology used: Square brackets **[....]**, refer to 0 or 1 occurrences of the bracketed item. Braces **{....}**, refer to 0 or more occurrences of the braced item, and the | symbol should be read as **'OR'**. Some undefined types appear in the grammar; these are defined as:

- *R3nn* – The set of 3-vectors with real, non-negative components.
- *R3_01* – The set of 3-vectors with components all in the range $[0, 1] \in \mathcal{R}$.

Note from the grammar that when defining a set of surface properties, we can instantiate simpler, previously-defined sets, rather than write all the information down again. Any conflict due to duplication will be resolved by taking the most recently defined value.

Note that *naming* sets of surface properties defined using **defprops** is compulsory.

As an example lets define a sample surface, which is highly specular, and has a texture file mapped onto it. To demonstrate its use we'll define the properties first, and instantiate them later in the scene.

```
defprops shiny_picture
src 0.8 0.8 0.8
arc 0.1 0.1 0.1
texture file portrait.tif
end

....
  instprops shiny_picture
....
```

2.4 Lights

Lights with zero surface area (point light sources) are discussed here. Area light sources are discussed in section 2.3.1.

Three types of light are supported by the language; **point**, **directional** and **spot** – these are specified in MSDL using text commands followed by parameters, which follow the syntax:

point $< pos \in \mathcal{R}^3 >$ $< intensity \in \mathcal{R}^{+3} >$

– A point light-source, located at *pos*, emitting the specified amount of light uniformly in all directions. The units things like lights & colour are measured in are application specific, but where possible desirable units are recommended. In this case Joules $\mathrm{sec}^{-1}\mathrm{m}^{-2}$ should be used.

directional $< direc \in \mathcal{R}^3 >$ $< intensity \in \mathcal{R}^{+3} >$

– As **point**, except positioned at ∞, so that all objects in the scene see the light along the given direction.

spot $< from \quad towards \in \mathcal{R}^3 >$ $< inten \in \mathcal{R}^{+3} >$
$< \exp \in \mathcal{N} >$ $< \mathrm{cutoff} \in [0, 90] >$

– This is a spotlight, situated at $< from \in \mathcal{R}^3 >$ and pointing towards $< towards \in \mathcal{R}^3 >$. The spread of the beam is controlled by the positive integer $< \exp \in \mathcal{N} >$; which is the power to which the '$\cos\theta$' term is raised, if the applications' lighting model requires it .The intensity of the light (Joules $\mathrm{sec}^{-1}\mathrm{m}^{-2}$) is given by $< inten \in \mathcal{R}^{+3} >$. The final variable ; $< \mathrm{cutoff} \in [0, 90] >$; gives the maximum value of 'θ' (in degrees) that we'll bother evaluating the illumination for; any bigger than that and we *can't see* the source – as if the spot has a conical shutter around it.

2.4.1 Light Syntax

Here is a short BNF grammar to specify the syntax of this part of the MSDL. (The full grammar for MSDL can be found in appendix A of this report).

lights ::= **point** R3 R3+
| **directional** R3 R3+
| **spot** R3 R3 R3+ natural R_090.

Some of the terms that appear in this piece of BNF are not defined in the grammar as shown; these are:

- R3 – The set of 3-vectors with real components.
- R3+ – The set of 3-vectors with real, positive components.
- natural – The set of natural numbers.
- R_090 – The set of real numbers in the range [0,90].

So for example if we wanted to define a spot light pointing towards the origin we might define a light using :

```
spot
0.0 100.0 0.0    0.0 0.0 0.0
1.0 0.0 0.0 /* Red light */
10 45
```

2.4.2 Other options

MSDL has been used in the past extensively in radiosity applications. Consequently an option has been added to MSDL to allow some control over the way in which an application will subdivide a given primitive for radiosity calculations.

The option **patches** $< U_{split} \quad [V_{split} \quad [W_{split}]] >$

can be added to any primitive *except polyhedra*. This information is never used by MSDL, and is simply stored and passed to the routine the parser calls in the application.

2.5 The View

The view syntax used in MSDL forms a superset of the information used by NFF, DKBTrace and PHIGS. MSDL files need not contain a view definition, and is provided merely for the users convenience.

view

from	$< pos \in \mathcal{R}^3 >$
towards	$< pos \in \mathcal{R}^3 >$
[**vpn**	$< direc \in \mathcal{R}^3 >$]
up	$< direc \in \mathcal{R}^3 >$
window	$< bl \quad tr \in \mathcal{R}^2 >$
[**front**	$< \mathrm{dist} \in \mathcal{R} >$]
[**back**	$< \mathrm{dist} \in \mathcal{R} >$]
[**parallel**]	

end

The position **from** defines the projection reference point for the view frustrum, whilst **towards** defines the view reference point – both are given in *world coordinates*.

If the view plane normal (**vpn**) is not supplied, then it is taken to be the unit vector lying along the line (**towards** - **from**).

The **up** vector (which need not be normalized) determines the v-axis of the (u, v, n) view reference coordinate system (**vpn** determines the n-axis). More specifically, the v-axis lies along the orthogonal projection of the **up** vector onto the *view plane* (the plane through **from**, orthogonal to **vpn**). The u-axis of the view reference coordinates is taken to be the vector lying in the view plane which forms an orthogonal right-handed set with the v- and n-axes.

The **window** is defined in the (u, v) coordinates in the view plane. Four reals define the window; (bottom-left.u, bottom-left.v, top-right.u, top-right.v).

The distances **front** and **back** are optional. They define the front and back clipping planes, and the distances are measured from **from**, increasing towards and beyond **towards**. They are included for compatibility with PHIGS.

The keyword **parallel** is used to indicate that a parallel (as opposed to perspective) projection, is desired.

A sample view then might look like

```
view
from 100.0 0.0 0.0
towards 0.0 0.0 0.0
```

```
up 0.0 0.0 1.0
window -10.0 -10.0 10.0 10.0
end
```

3 Generating MSDL files

As a résumé of the capabilities of MSDL we'll now discuss some simple scenes[1].

We'll construct a simple scene from a collection of objects, beginning with a unit box. As we're going to use that several times we'll only define it.

```
defobj unitbox
  box
       0.0 0.0 0.0
       1.0 1.0 1.0
  end
end
```

Now we can instantiate several of these building boxes, moving each up slightly to form a very simple model of a childs building box set.

```
instobj unitbox
  shift 0.0 0.0 0.0
end

instobj unitbox
  shift 0.0 0.0 1.0
end

instobj unitbox
  shift 0.0 0.0 2.0
end
```

To make the scene more interesting we can place surface properties in each of the boxes;

```
/* lowest brick in the tower */
instobj unitbox
  props
    drc 1.0 0.0 0.0
    texture file first_brick.tif
  end
  shift 0.0 0.0 0.0
end

instobj unitbox
  props
    drc 1.0 0.0 0.0
```

[1] More complicated test scenes are available for ftp from the Manchester site

```
    texture file second_brick.tif
  end
  shift 0.0 0.0 1.0
end

/* top brick in the tower */
instobj unitbox
  props
    drc 1.0 0.0 0.0
    texture file third_brick.tif
  end
  shift 0.0 0.0 2.0
end
```

Just to make the simple scene more interesting we'll generate a floor, upon which the tower of bricks rests.

```
polygon
  4
  -4.0 -4.0 0.0
  -4.0 4.0 0.0
   4.0 4.0 0.0
   4.0 -4.0 0.0
end
```

Figure 4 shows the scene if rendered using wire-frame.

So that we can render the scene we'll place a spot light above the tower, pointing down at an angle

```
spot 0.0 0.0 100.0
     10.0 0.0 0.0
     1.0 1.0 1.0 /* Its a white light */
     10.0 60.0
```

Having exhausted the possibilities with simple objects lets move on to composite object; this example is a circular table, with a single central leg, and a square flat base which fixes it to the floor:

```
/* An example of a complex composite */

compobj coffee_table

  /* position the whole thing where we want it in the scene */
  shift 10.0 20.0 15.0

  /* The table top */
  cylinder
    0.0 0.0 0.0 capped 0.0 0.0 1.0 capped 1.0
    scale 10.0 10.0 0.1  shift 0.0 0.0 10.0
```

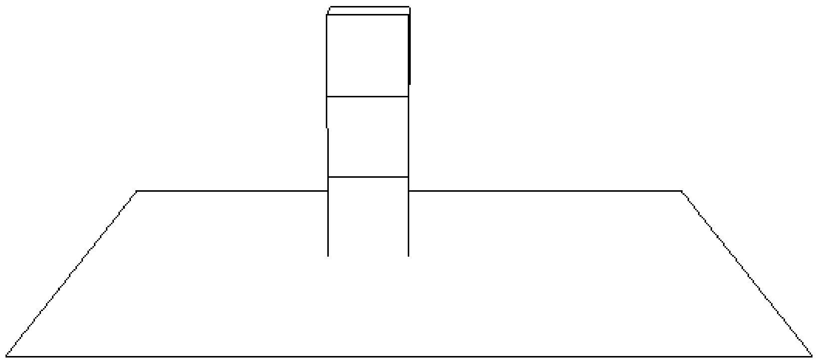

Figure 4: The childs tower.

```
    instprops dark_wood
  end

  /* The leg */
  cylinder
    0.0 0.0 0.0 0.0 0.0 1.0 1.0
    scale 2.0 2.0 9.8  shift 0.0 0.0 0.2
    instprops  metallic_grey
  end

  /* The base */
  box  -2.0 -2.0 0.0    2.0 2.0 0.2
    instprops  metallic_grey
  end


end /* coffee table */
```

Admittedly we haven't defined the properties 'dark_wood' or 'metallic_grey' yet, but we're assuming that's been done elsewhere.

Included with the MSDL kit is a sample scene file, which was used as a test for a radiosity/shadow algorithm here at Manchester. This file contains 3 polygons, one of which is a light source by virtue of having a defined emission. Figure 5 shows the view you should get if you load the file into your application.

3.1 Creating libraries of objects

In any sizeable scene objects will need to be defined and instantiated many times, and in order to save time generating new scenes you may wish to generate a collection of pre-defined objects, which can be added to new scenes. Inside the pre-defined files use **defobj** to define and attach a name to a complicated object, so it can be **instobj**'ed by a scene.

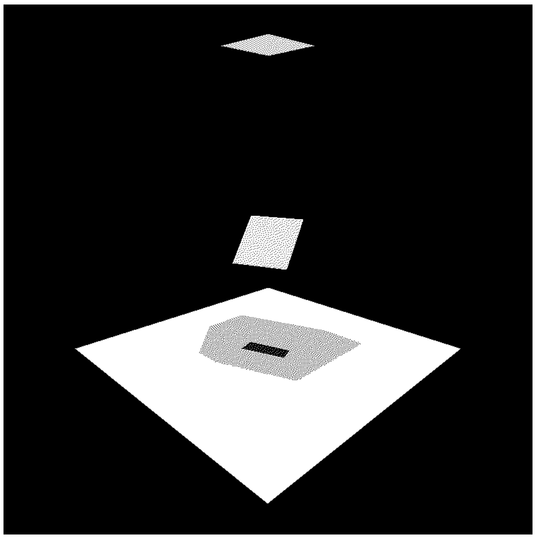

Figure 5: A sample render of `simple.msdl`

An MSDL file which has been constructed from existing definitions might look like:

```
#include "chairs"
#include "tables"

compobj
  name room
  instobj big_comfy_chair
    shift 10 0 10
  end
  instobj nice_coffee_table
    shift 5 0 5
  end
end
```

Then before passing this file to the MSDL reader you must use a pre-processor to create a plain MSDL file. (See section 5 for another, and less clumsy, way of using pre-processors).

As MSDL files are text files it is possible for user applications to write MSDL files. If you intend to use this method to produce libraries to be used in other scenes then we advise you prefix all names with an underscore (_) to help reduce any confusing name clashes.

4 Building `libmsdl.a`

The MSDL kit consists of a collection of C routines which constitute an MSDL parser, together with this documentation and some sample MSDL files.

The Makefile allows you to build a library for linking into your application, and can be compiled for either C or C++. To compile for C simply type `make`. This will call `lex(1)`, `yacc(1)` and `cc(1)` to generate `libmsdl.a`.

To compile the library for C++ edit the `Makefile`, changing the definitions of `$(CC)` and `$(CFLAGS)`. A C++ version of the library will use `new` & `delete` to manage memory, which should avoid some of the memory leakage problems less forgiving C++ compilers can exhibit.

Included with the library is a sample program which illustrates how you link this library with an application, and then call the parser. This can be generated by typing `make tester`. Once compiled the parser can be verified by typing `tester simple.msdl`, which will parse a trivial file. If there are no errors `tester` will terminate normally.

If you wish to debug your applications reader define VERBOSE in `$(CFLAGS)`, and recompile. The reader should display the information it is storing in its internal records for you to compare with your application. If you compile `libmsdl.a` with VERBOSE defined then the library will display debugging information as it fills its internal structures. This may or may not be of use to you!

MSDL creates its own internal representation of the scene, which by default *isn't* `free()`'d at the end. (In earlier versions of MSDL its storage was used by the application itself, and so it made sense to keep it hanging around after parsing). If you wish to clean it up, as you probably should, define CLEANUP in the Makefile.

5 Linking your application to the MSDL reader

The MSDL reader parses the file, filling in its internal data structures as it goes. Once completed the reader traverses its structures calling the relevant `create_*` routines for each data item. These routines are defined as skeletons in `application.c`.

It is anticipated that the application reading MSDL files will have its own internal data format for things like polygons, lights etc, which will need to be filled in. This data can be copied from the MSDL record `Tobject*` passed to the skeleton routine. (The definition of `Tobject` can be found in `msdl_types.h`, but is repeated in this document for your convenience.)

If you are writing your application from scratch, and do not have existing object definitions you *may* wish to use the MSDL types. If you do ensure CLEANUP **isn't** defined when you compile.

The `create_view` routine is only called if the MSDL file being parsed contains a view definition. Otherwise it is up to the application to choose a meaningful default.

Each of these object routines is passed two parameters, and takes the form

```
void create_polygon (Tobject*, Tmatrix );
```

The second item is the *global* transformation which may need to be applied to the object.

The MSDL structures are hierarchical, that is transformations applied to node structures need to be applied to leaf items. As the user application may not employ a hierarchical store the global transform is accumulated for the application and passed as the second parameter. In a hierarchical application this can be ignored, and only the transform stored inside the `Tobject` copied. [2], otherwise apply the global transform.

When the MSDL parser wishes to indicate to the application that the following objects are children of the previous object (i.e., it is stepping one level down in its tree) it calls `create_comp()`, and calls `end_comp()` when stepping back up. This should allow hierarchical applications to generate the correct data.

[2] For our convenience `libmsdl.a(trans.c)` contains the function `transPt3()` which transforms a `Tpt3` by a given matrix. You may wish to use this yourself!

Once the relevant create routines have been written and linked into your application, your program needs to call the function `msdl(FILE *)` to start the parser reading the relevant file.

So, to summarise, in order to enable your application to read MSDL files you will need to;

- make `libmsdl.a`
- copy `application.c` into wherever your main app is stored, and rename it to something meaningful, e.g., ReadMSDL.c
- Fill in that file with `create_*` routines
- Inside your application call something like :

```
FILE *fp;

  /* Initialise any global variables used by your create routines */

fscanf(stdin, "Enter filename : %s\n", filename);
fp = fopen(filename, ``r'');
msdl(fp);
..
```

  This parses the file and calls the skeleton routines. If you intend to use a pre-processor to generate your MSDL files, and are working on a UNIX-like system you may be able to use `popen(3S)` to pre-process as you parse the file. So you will need to write something resembling:

```
fp = popen(``/lib/cpp -P filename.msdl'', ``r'');
msdl(fp);
..
```

- Link the parser into your application.

5.1 Creating objects

These routines are supplied as skeletons in `application.c`. Most applications will not be able to cope with all the primitives defined in MSDL, these applications should leave the relevant create functions as skeletons.

- `create_view()` – create a view in your world.
- `create_comp()` – Start creating a composite object in your world.
- `end_comp()` – Finish creating a composite object.
- `create_triangle()`– create a triangle in your world.
- `create_polygon()` – create a polygon in your world.
- `create_sphere()` – create a sphere in your world.

- `create_cylinder()`– create a cylinder in your world.
- `create_cone()` – create a cone in your world.
- `create_disc()` – create a disc in your world.
- `create_box()` – create a box in your world.
- `create_nurb()` – create a NURBS in your world.
- `create_polyhedron()` – create a polyhedron in your world.
- `create_spot_lt()` – create a spot light in your world.
- `create_directional_lt()` – create a directional light in your world.
- `create_point_lt()` – create a spot light in your world.
- `create_view()` – create a view in your world.
- `create_problem()` – Handle an error

As discussed earlier all the functions concerned with creating objects have 2 arguments passed to them. These are a pointer to the object (`Tobject*`) and a global transformation matrix.

The transformation stored in an object is only intended to be applied to that object and not to it's children and parents. The transformation matrix given is one which would put the object in the correct place if it's parents are ignored.

The object is a structure which contains all the information gathered about the primitive, and is of the form:-

```
typedef struct object {
     .... /* Internal store removed for clarity */

     Tstring          name;       /* what's it called                */
     Tsprops          sprops;     /* surface properties              */
     Tmatrix          trans;      /* transformation matrix           */
     Tptchs           psplit;     /* 3x4, 4x5x6 or what ?            */

     union {                         /* specific stuff */
         struct {               /* polygon-specific stuff */
             Tpolydat       vvnfn;
         } polg;

         struct {               /* tri-strip specific stuff */
             Tpolydat       vvnfn;
         } tri;

         struct {              /* sphere-specific stuff */
             Tpt3      centre,       /* centre                          */
                       up,           /* N pole (from centre) vector     */
                       tpz;          /* THETA=PHI=0 (from centre) vector */
             Tfloat    radius;       /* the radius                      */
             TThetaPhi theta,        /* THETA start and finish          */
```

```
                phi;            /* PHI start and finish            */
     } sph;

    struct {                /* disc specific stuff */
        Tpt3      centre,         /* the centre                      */
                  normal,         /* the surface normal              */
                  theta_zero;     /* the vector indicating theta zero */
        Tfloat    radius;         /* the radius                      */
        TThetaPhi theta;
    } disc;

     struct {               /* cylinder specifics */
        Tpt3      top,            /* the top                         */
                  bottom,         /* the bottom                      */
                  theta_zero;     /* vector indicating theta zero    */
        Tfloat    radius;         /* the radius                      */
        Tbit      tcapped,        /* is the top capped               */
                  bcapped;        /* is the bottom capped            */
        TThetaPhi theta;
     } cyl;

     struct {               /* Cone specifics */
        Tpt3      bottom,         /* the bottom                      */
                  top,            /* the top                         */
                  theta_zero;     /* vector indicating theta zero    */
        Tbit      bcapped,        /* is the bottom capped?           */
                  tcapped;        /* is the top capped ?             */
        Tfloat    bradius,        /* the radius at the bottom        */
                  tradius;        /* the radius at the top           */
        TThetaPhi theta;
     } cone;

     struct {               /* NURBS specifics */
       Tpint      order_u,
                  order_v,
                  no_cpts_u,
                  no_cpts_v;
       Tfloat     *knots_u,       /* an array of the u_knots         */
                  *knots_v;       /* an array of the v_knots         */
       Tpt4       *cpts;          /* an array of u&v control points  */

     } nurb;

     struct {               /* Polyhedron specifics */
       Tpolydat   vvnfn;          /* verts, vert_norms, face_norms   */
       Tfarray    flist;          /* the faces                       */
     } polh;

     struct {               /* Box details */
       Tpt3       bl,             /* bottom - left                   */
                  tr;             /* top    - right                  */
```

```
          Tbit         in;             /* is it pointing inwards          */
        } box;
      } spec;                /* specifics for this object type */
  } Tobject;
```

(This type, and all the structures used by it are defined in `msdl_types.h`).

Although most of the definition should be self-explanatory a few points are worthy of note.

The type `Tpolydat` is a structure containing 3 `Tpint`'s and an array of `Tpt3`'s. The Tpints indicate how many vertices there are, how many vertex normals there are (either equal to the number of vertices or none) and the number of facet normals . The array contains these points and vectors in that order. i.e. the first facet normal will be stored at `vvnfn.data[ vvnfn.no_verts + vvnfn.no_vnorms ]`.

The type `Tfarray` contains an array of pointers to singlefaces and a count of how many faces there are. A single face contains a counter of how many vertices make up a face and an array containing an index to the vertices. The actual co-ordinates of a vertex will have to be retrieved from the `vvnfn` data.

5.2 Creating Lights

There are two ways to create lights in your world, i.e. lights with no surface area (which might be suitable for local illumination models or ray tracing) and objects with surface properties that emit light. If your scene contains point lights, directional lights, and spot lights then separate create functions are called. These are:

- `create_spot_lt()` – create a spot light in your world.
- `create_directional_lt()` – create a directional light in your world.
- `create_point_lt()` – create a spot light in your world.

Note that if you define your lights actually to be objects with emission values then your objects reader must detect them in functions such as `create_polygon`.

Each function is passed a pointer to a `Tlight`.

```
typedef struct lights {
    Tlight             ltype;          /* pt, dir, sp            */

    union { /* specifics for the different light types          */
        struct { /* point light       */
            Tpt3       locn;           /* location               */
            Tspectra   inten;          /* intensity              */
        } point;

        struct { /* directional light */
            Tpt3       along;          /* see light along "along"*/
            Tspectra   inten;          /* intensity              */
        } direc;
```

```
        struct { /* spot light        */
            Tpt3      locn,            /* located at...          */
                      at;              /* pointing at...         */
            Tspectra  inten;           /* intensity              */
            Tpint     exp;             /* cos^exp                */
            Tpint     cutoff;          /* should be in [0,90]    */
        } spot;
    } spec;

    ptr2llist         next;
} Tlights;
```

5.3 Creating the View

The view is optional in your MSDL file and if none is defined this function will not be called.

The create_view routine is passed a pointer to a Tview, which is defined as

```
typedef struct view {
    Tpt3  from, /* viewing FROM here */
          towards, /* looking TOWARDS this point */
          vpn, /* view plane normal */
          up; /* up vector */
    struct {
      TptUV     bl,
                tr;
    } window; /* window in viewing plane */
    Tfloat      fpd, /* front plane distance */
                bpd; /* back plane distance */
    Tboolean    persp; /* TRUE if perspective */
} Tview;
```

5.4 Error handling

If the MSDL parser detects an error while reading a file it calls the function

create_problem (char *prob, int line_number).

This is defined in application.c to print the error and exit(2), though the application writer will probably wish to change this to something more meaningful to the application, e.g., bring up a dialog box.

6 Credits and closing remarks

Initial programming Neil Gatenby

Completion and debugging Alex Knowles, Neil Gatenby, Martin Preston

Documentation Martin Preston, Neil Gatenby, Terry Hewitt, Chris Lilley, Alex Knowles

Testers David Hutchinson, Dean Gammage, Neil Gatenby, Martin Preston, Richard Andrews, Antonio Costa

Particular thanks go to Dr Pete Jinks (Computer Science Dept., University of Manchester) whose help with `lex(1)` and `yacc(1)` has been invaluable, and Terry and Chris, who helped polish up the documentation.

A BNF Grammar for the Manchester Scene Description Language

scene	::=	[*viewblk*] *sceneblk.*
viewblk	::==	**view**
		from R3+
		towards R3+
		[**vpn** R3]
		up R3
		window R2 R2
		[**front** natural]
		[**back** natural]
		[**parallel**].
sceneblk	::=	{*lights* | *defpropsblk* | *objects*}
lights	::=	**point** R3 R3+
	|	**directional** R3 R3+
	|	**spot** R3 R3 R3+ natural R_090.
properties	::=	*defpropsblk*
	|	*propsblk*
	|	*instpropsblk.*
defpropsblk	::=	**defprops** *id* {*instpropsblk*} {*propsprims*} **end.**
propsblk	::=	**props** [**name** *id*] {*instpropsblk*} {*propsprims*} **end.**
instpropsblk	::=	**instprops** *id.*
propsprims	::=	**emission** *R3nn*
	|	**arc** *R3_01*
	|	**drc** *R3_01*
	|	**src** *R3_01*
	|	**trc** *R3_01*
	|	**phexp** *natural*
	|	**index** *R3nn*
	|	**texture** [**file**] *name.*
name	::=	**name** *id.*
transforms	::=	deftransblk
	|	transblk
	|	insttransblk
	|	{ transprims }.
deftransblk	::=	**deftrans** id { insttransblk } { transprims } **end.**
transblk	::=	**trans** [**name** id] { insttransblk } { transprims } **end.**
insttransblk	::=	insttransblk id.
transprims	::=	**shift** R3
	|	**scale** R3
	|	**rotate** axis R.
	|	**matrix** R4 R4 R4 R4
patchesblk	::=	**patches** natural [natural [natural]]

objects	::=	*defobjblk*
	|	*objblk*
	|	*instobjblk*
	|	*compobjblk*
	|	*objprims*.
compobjblk	::=	**compobj** [*id*] {*options*} {objcentre} **end**.
defobjblk	::=	**defobj** *id* *objcentre* end.
objblk	::=	**object** [*name*] *objcentre* **end**
	|	*objprims*.
instobjblk	::=	**instobj** *id* *options* **end**.
objcentre	::=	{*instobjblk* } {*objprims* } *options*.
options	::=	[*propsblk* | *instpropsblk*] {*transforms*} [*patchesblk*]
objprims	::=	**triangle** [*name*] R3 R3 R3 [**vnorm** R3 R3 R3] [**fnorm** R3]
		options **end**.
	|	**polygon** [*name*] natural R3 .... R3 [**vnorm** R3 .... R3] [**fnorm** R3]
		options **end**.
	|	**sphere** [*name*] R3 R±
		[**up** R3] [**tpz** R3] [**theta** R_01 R_01] [**phi** R_01 R_01]
		options **end**.
	|	**cylinder** [*name*] R3 [**capped**] R3 [**capped**] R±
		[**tz** R3] [**theta** R_01 R_01] *options* **end**.
	|	**cone** [*name*] R3 [**capped**] R± R3 [**capped**] R±
		[**tz** R3] [**theta** R_01 R_01] *options* **end**.
	|	**disc** [*name*] R3 R+ R3
		[**tz** R3] [**theta** R_01 R_01] *options* **end**.
	|	**box** [*name*] R3 R3 [in] *options* **end**.
	|	**nurbs** [*name*]
		order natural natural
		cpts natural natural
		knots { R .... R }
		{ R .... R }
		R3 × R+
		....
		R3 × R+
		options **end**.
	|	**polyhedron** [*name*]
		{ R3 ..... R3 }
		[
		vnorm
		R3 ..... R3
		]
		{
		[natural ..... natural]
		....
		[natural ..... natural]
		}
		[
		fnorm
		R3 ..... R3
		]
		options **end**.

letter	::=	a|b|...|z|A|...|Z.
alpha	::=	*letter*|_|0|...|9.

- R – The set of real numbers.
- natural – The set of natural numbers.
- R+ – The set of positive real numbers.
- R± – The set of positive and negative real numbers.
- R_090 – The set of real numbers in the range [0,90].
- R2 – the set of 2-vectors with real components.
- R3 – The set of 3-vectors with real components.
- R4 – The set of 4-vectors with real components.
- R3_01 – The set of 3-vectors with components all in the range $[0,1] \in \mathcal{R}$.
- R3+ – The set of 3-vectors with real, positive components.
- *R3nn* – The set of 3-vectors with real, non-negative components.
- axis – The set of characters {x,X,y,Y,z,Z}.

References

[1] David Buck. DKBTrace, version 2.12. Available from `alfred.ccs.carleton.edu (134.117.1.1)`.

[2] Eric Haines. Neutral file format (NFF). Available from `gondwana.ecr.mu.oz.au (128.250.70.62)`.

[3] W. T. Hewitt and D. Yip. The NURBS Procedure Library. Technical report, Computer Graphics Unit, University of Manchester, 1992. CGU 76, Available from `ftp.mcc.ac.uk (130.88.200.7)`.

[4] Craig Kolb. Rayshade 4.0. Available from `princeton.edu (128.112.128.1)`.

[5] Greg Ward. Radiance. Available from `wuarchive.wustl.edu (128.252.135.4)`.