App. B) GE3D "Graphics Engine 3D"

The GE3D library - Graphics Engine 3D - was designed as a machine independent, immediate mode, 3D graphics interface.

The first version of GE3D was developed together with Michael Hofer, whose work is acknowledged here.

The functionality of GE3D includes:

- manipulation of vectors and matrices, and a stack of transformation matrices
- camera definition, both perspective and orthographic
- definition of light sources
- double buffering (two screen pages)
- drawings in wire frame, hidden line, flat shaded and smooth shaded
- drawing of 3D faces (polygons) and polyhedra
- drawing some 2D primitives: lines, rectangles, arcs, circles, output of text

The library was implemented in (standard) C atop the GL graphics library of Silicon Graphics, with a view for future portation to Open GL. Header and implementation file contain macros to switch between ANSI C and Kernighan-Ritchie C by defining the pre-processor symbol GE3D_PROTOTYPES.

As it can be seen from the list above, GE3D's functionality is on a higher level than typical machine dependent libraries like GL or Starbase. For example a polyhedron represents a set of faces with the same edge and fill colour. The drawing modes (from wire frame to smooth shading) do not bother the user of the library with correct settings of flags for hidden surface elimination, filling, usage of z-Buffer and so on. Other functions, for example for drawing lines and rectangles, and for manipulations of the matrix stack, have a corresponding counterpart in low level graphic libraries.

One could ask for the reason why implementing yet another graphics interface (moreover when it is closely related to GL). Beside the mentioned enlarged functionality the machine independence of the interface increases the *portability* of the programs. As the header file is completely independent of any other header file of graphics interfaces another implementation of the GE3D library can be linked at any time without changes or recompilation of existing programs.

B.1 Type Definitions

File <ge3d/vectors.h> contains the definitions for 3D *points* and *vectors* and a set of pre-processor macros for vector operations.

```
typedef struct
{ float x, y, z;
} vector3D, point3D;

typedef float matrix [4][4];
```

Type *matrix* is used for transformations (4D homogeneous coordinates). There is no assumption on the ordering of elements in the matrix (row major or column major). Functions of GE3D should be used to build and to concatenate the matrices, they can be stored and pushed onto the transformation stack, but should not be manipulated.

The type *face*, specifying a polygon in 3D space is defined in file <ge3d/face.h>. See procedure ge3d_polyhedron for further explanation of fields.

```
typedef struct
{ int num_faceverts,    /* number of vertices in face */
      num_facenormals;  /* number of normals in face */
  int *facevert,        /* array of indices of face vertices */
      *facenormal;      /* ... and of the normals of the face */
  vector3D normal;      /* normalised, outward face normal */
} face;
```

The four supported *drawing modes* are defined in <ge3d/ge3d.h>:

```
enum ge3d_mode_t
{ ge3d_wireframe,
  ge3d_hidden_line,
  ge3d_flat_shading,
  ge3d_smooth_shading
};
```

B.2 Functions

B.2.1 Opening the Graphics Device

There are two ways for opening the graphics device. Either GE3D is asked to open a window - this is done with the function

```
void ge3d_openwindow ();
```

Otherwise the main program is responsible for opening an output window. A call to

```
void ge3d_init_ ();
```

will initialise the GE3D library, including clearing the screen and setting default values. A call of ge3d_open_window automatically causes ge3d_init_ to be called.

B.2.2 Display Control

```
void ge3d_clearscreen ();
```

clears the window (with the current background colour) and the z-Buffer (if hidden surface elimination is activated).

```
void ge3d_swapbuffers ();
```

The GE3D library uses double-buffering (if available). This means that there are two screen pages for graphic output: a *visible* and an *active* one. All drawings are done on the *in*visible page. ge3d_swapbuffers must be called to display a drawn picture by swapping the two pages. The purpose is to avoid flickering on animations. If no double-buffering is available, this function is intended to flush any cached drawings onto the window.

B.2.3 Drawing Modes and Attributes

```
void ge3d_setmode (ge3d_mode_t mode);
```

sets the drawing mode for shapes, which should be one of:

ge3d_wireframe,
ge3d_hidden_line,
ge3d_flat_shading, or
ge3d_smooth_shading.

```
void ge3d_setbackgcolor (float R, float G, float B);
```

Sets the background colour[1]). All *colours* are specified as triples of RGB values in range 0.0 to 1.0.

[1]) Please excuse the mixture of British and American English spelling. Sorry.

void **ge3d_setfillcolor** (float R, float G, float B);

Sets the fill colour (RGB values). Only relevant in modes flat and smooth shading. Also sets the edge colour to (R, G, B).

void **ge3d_setlinecolor** (float R, float G, float B);

Sets the colour (RGB) for drawing lines and polygon edges. Usually there is a performance penalty for drawing polygons with different line and fill colour; if needed, ge3d_setlinecolor has to be called after ge3d_setfillcolor.

void **ge3d_setlinestyle** (short pattern);

Sets the linestyle pattern, which is specified as a 16-bit integer. For example 0xffff (or -1) is a solid line, 0x0f0f a dashed line.

void **ge3d_setlinewidth** (short width);

Sets the line width in pixels. There may be a performance penalty for drawing lines with widths greater than one or nonsolid lines.

B.2.4 The Transformation Matrix Stack

All drawing routines are called with so called *modelling coordinates*, also called object coordinates, because it is the coordinate system in which graphical objects are defined (or modelled). Cameras and light sources are specified in the *world coordinate* system (or scene coordinates). Figure B.1 gives an overview over the coordinate systems from object to window coordinates.

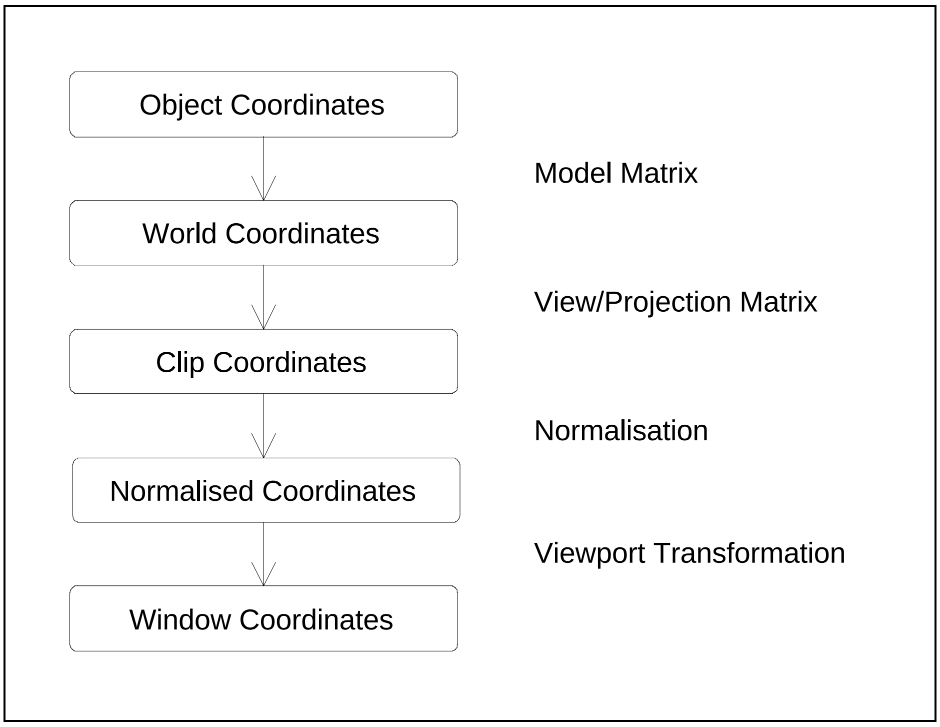

Figure B.1: coordinate systems

The other coordinate spaces (below world coordinates) are entirely handled by the graphics library. The camera transformation transforms world coordinates to *clip coordinates* - the view frustum transforms into an axis aligned cube, depth clipping is done in this coordinate system. For z-buffering (hidden surface elimination) the cube is normalised into coordinates in range 0.0 to 1.0 - these are called *normalised coordinates.* At last a simple scaling and translation maps the normalised coordinates to the *window coordinates*.

The transformation from modelling to world coordinates is done with the current *transformation matrix*. All transformation matrices use 4 by 4 homogeneous coordinates and describe *affine* transformations (linear plus translation). Affine transformations include translation, rotation, scaling, and shearing. The matrix need not be built by the user - the GE3D library contains functions to compute it from values for translation, rotation and scaling. (These functions are discussed in the next section.)

Transformation matrices can be stacked on the *transformation matrix stack*. The current transformation matrix is the top matrix on the stack. Other transformation matrices can be pushed onto the stack and later removed. On pushing, the matrix it may be concatenated with the old top matrix, meaning that the transformations are relative to the old one, thus allowing hierarchical description of objects. A typical limit for the maximum depth of the stack set by the underlying graphic library is 64.

There are several routines for handling the transformation stack:

void **ge3d_push_matrix** ();

Pushes down the transformation stack by *copying* the old top matrix. (If the stack was empty an identity matrix is pushed).

```
void ge3d_push_this_matrix (matrix mat);
```

Pushes down the transformation stack by *pre-concatenating* the matrix mat with the old top matrix. (On an empty stack the matrix is pushed unchanged.)

```
void ge3d_push_new_matrix (matrix mat);
```

Pushes down the transformation stack and puts the matrix mat *unchanged* onto the top of the stack.

```
void ge3d_transform_mc_wc (float in_x, float in_y, float in_z,
                           float* o_x, float* o_y, float* o_z)
```

Transforms the point (in_x, in_y, in_z), given in modelling coordinates, with the current transformation matrix to the point (*o_x, *o_y, *o_z) in world coordinates. This transformation is applied implicitly to all 3D points when calling any drawing function.

```
void ge3d_transformvector_mc_wc (float in_x, float in_y, float in_z,
                           float* out_x, float* out_y, float* out_z);
```

Same as ge3d_transform_mc_wc, but for *vectors*. Note: a translation of a vector makes no sense, therefore the transformation is applied without translation.

```
void ge3d_print_cur_matrix ();
```

Prints the values of the current transformation matrix to stderr. Can be used as debugging tool. Must not be called on an empty stack.

```
void ge3d_get_and_pop_matrix (matrix mat);
```

Stores the current transformation matrix in mat and *pops* it from the stack (mat is a reference parameter, because matrix is a float array). Must not be called when the stack is empty.

```
void ge3d_pop_matrix ();
```

Pops the current transformation matrix from the stack. Must not be called when the matrix stack is empty.

B.2.5 Building Transformation Matrices

The functions *ge3d_translate*, *ge3d_rotate_axis* and *ge3d_scale* build and concatenate transformation matrices for the most common affine transformations: translation, rotation about a coordinate axis and scaling. The computed transformation matrix is *pre-multiplied* to the current transformation matrix.

That means the transformations take effect in the order the functions are called, for example first a translation and then a rotation. Mathematically the transformations have to be applied in the reverse order to get the right result, like first rotating and then translating in the example.

It is an error to call these functions on an empty stack. Function *ge3d_push_matrix* () should be used first to push an identity matrix onto the stack.

void **ge3d_translate** (float x, float y, float z);

Does a translation by the vector (x, y, z).

void **ge3d_rotate_axis** (char axis, float angle);

Does a rotation about the axis *axis* ('x', 'y' or 'z') by angle *angle*. The angle is measured in degrees, counter clockwise when looking along the axis towards the origin.

void **ge3d_scale** (float sx, float sy, float sz, float all);

Does scaling with the factor sx along the x axis, sy along y and sz along z, and an overall scaling with the factor all.

B.2.6 Text

void **ge3d_text** (float x, float y, float z, const char* s);

Output of a text string s, beginning at position (x, y, z). The text is written horizontally on the window, beginning at the transformed position. The current line colour is used.

B.2.7 Line Primitives

Some of the functions discussed in this section only have arguments for two dimensions. They are used primarily for 2D drawings, but are also three-dimensional. They draw into the plane z = 0, but are also affected by the current transformation matrix, which can be used to translate and rotate the drawing into the desired position and orientation.

```
void ge3d_moveto (float x, float y, float z);
```

Moves the current position to the point (x, y, z).

```
void ge3d_lineto (float x, float y, float z);
```

Draws a line from the current 3D position to the point (x, y, z), using the current line colour, style, and width. Then the current 3D position is updated to (x, y, z) for further calls to ge3d_lineto.

```
void ge3d_rect (float x0, float y0, float x1, float y1);
```

Draws the outline of an axis-aligned rectangle with opposite points (x0, y0) and (x1, y1) in the plane z = 0. Current line attributes are used.

```
void ge3d_wirecube (float x0, float y0, float z0,
                    float x1, float y1, float z1);
```

Draws an axis-aligned cube with opposite vertices (x0, y0, z0) and (x1, y1, z1) as wire frame (regardless of the current drawing mode), using the current line attributes.

```
void ge3d_circle (float x, float y, float r);
```

Draws the outline of a circle with midpoint (x, y, 0) and radius r in the plane z = 0, using the current line attributes.

```
void ge3d_arc (float x, float y, float r,
               float startangle, float endangle);
```

Draws an arc, which is defined as the part of a circle with midpoint (x, y, 0) and radius r (in plane z = 0), beginning at startangle, ending at endangle in counter-clockwise direction. The two angles are given in degrees, measured CCW from the positive x-axis. The current line attributes are used.

```
void ge3d_wirepolyhedron (point3D* vertexlist, vector3D* normallist,
                          int numfaces, face* facelist);
```

Draws a wire frame model of a polyhedron (in any drawing mode). The arguments have the same meaning as for *ge3d_polyhedron* (see next section).

B.2.8 Solid Primitives

```
void ge3d_circf (float x, float y, float z);
```

Draws a circle, filled with the current fill colour (regardless of the current drawing mode). The circle is drawn with midpoint (x, y, 0) and radius r in the plane z = 0.

Filling depends on the current drawing mode (set with ge3d_mode). In mode *wire frame* only the outline of faces is drawn. *Hidden line* does a hidden line elimination which may be achieved by filling the face with the background colour.

Flat and smooth shading take the light sources into account. *Flat shading* uses a single (constant) colour for each face. *Smooth shading* (Gouraud shading) requires normal vectors for the vertices and interpolates the colour smoothly over the face. These two modes also include a hidden surface elimination.

```
void ge3d_polygon (point3D* vertexlist, int nvertices,
                   int* vertexindexlist,
                   vector3D* normallist, int nnormals,
                   int* normalindexlist,
                   vector3D* f_normal);
```

Draws a polygon. Parameters:

vertexlist	an array of 3D vertex coordinates (modelling coordinates).
nvertices	the number of vertices of the polygon.
vertexindexlist	an array of nvertices integer indices, telling which vertex of vertexlist is the first vertex of the polygon, the second and so on., in counter clockwise order when seen from outside. The first vertex of vertexlist has index 0.
f_normal	outward normal vector (face normal), used for flat shading. This normal has to be provided for efficiency (to avoid recomputation including normalisation at each drawing).

The other parameters are only used in mode *smooth shading* and if nvertices = nnormals. In this case vertex normals must be provided. If the mode smooth shading is active, but nnormals is less than nvertices, the polygon is flat shaded.

normallist	an array of vertex normal vectors (outward, modelling coordinates).
nnormals	the number of vertex normals of the polygon.
normalindexlist	an array of integer indices, telling which normal vector to use for the first vertex of the polygon, which for the second one and so on. The first normal of normallist has index 0.

That means that the polygon with vertices *vertexlist [vertexindexlist [0]]* to *vertexlist [vertexindexlist [nvertices-1]]* is drawn and automatically closed. In mode smooth shading and if *nvertices = nnormals* the normal vectors *normallist [normalindexlist [0]]* to *normallist [normalindexlist [nnormals-1]]* are used as vertex normals to calculate the colour at the vertices for shading.

For correct results the polygon has to be *convex*. The drawing of a concave or non-simple polygon is undefined but lies within the convex hull. Polygons are *single sided*, therefore the vertices must be given in counter clockwise order when seen from the front.

The number of vertices per face may be limited by the graphics library (e.g. to 255). Normal vectors should be *normalised* to a length of 1.0 for correct shading.

```
void ge3d_polyhedron (point3D* vertexlist, vector3D* normallist,
                int numfaces, face* facelist);
```

Draws a polyhedron. The parameters are:

vertexlist	an array of vertex coordinates (modelling coordinates).
normallist	an array of vertex normal vectors (outward, modelling coordinates).
numfaces	the number of polygons (faces) of the polyhedron.
facelist	an array of numfaces faces.

A *face* contains all the data for one polygon of the polyhedron (see also B.1 Type Definitions):

num_faceverts	the number of vertices of a face/polygon.
num_facenormals	the number of vertex normals.
facevert	an array of integer indices into the vertexlist.
facenormal	an array of integer indices to the normallist.

Note that the vertexlist and normallist (triples of floats) are shared among all polygon faces. Only an integer index is used to specify which vertex or vertex normal is used.

A call of *ge3d_polyhedron* (vertexlist, normallist, numfaces, facelist) leads to the same drawings as a call to *ge3d_polygon* for all faces *facelist [0]* to *facelist [numfaces - 1]* as a loop like

```
int i;
face* faceptr;
for (i = 0, faceptr = facelist;  i < numfaces;  i++, faceptr++)
{
  ge3d_polygon (vertexlist, faceptr->num_faceverts,
              faceptr->facevert,
              normallist, faceptr->num_facenormals,
              faceptr->facenormal,
              &faceptr->normal);
}
```

The use of *ge3d_polyhedron* is in most cases faster because it avoids unnecessary function calls and can draw all faces at once in the proper drawing mode.

Example: A tetrahedron with ground plane in z = 0.

```
static point3D vertexlist [] =         /* vertices */
  {{0, 0, 0}, {1, 0, 0}, {0.5, 0.8, 0}, {0.5, 0.5, 1}};
static int facevert [][3] =            /* vertex indices */
  {{2, 1, 0}, {0, 1, 3}, {1, 2, 3}, {2, 0, 3}};
static vector3D normal [] =            /* face normals */
  {{0, 0, -1}, {0, -0.894, 0.447},
   {0.837, 0.523, 0.157}, {-0.837, 0.523, 0.157}
  };
face facelist [4], *fptr;
int i;

for (i = 0, fptr = facelist;  i < 4;  i++, fptr++)
{ fptr->num_faceverts = 3;
  fptr->facevert = facevert [i];
  fptr->num_facenormals = 0;           /* no vertex normals:*/
  fptr->facenormal = NULL;             /* flat shading */
  fptr->normal = normal [i];
}

ge3d_polyhedron (vertexlist, NULL, 4, facelist);
```

B.2.9 Camera Definition

GE3D supports a perspective and an orthographic camera model. The perspective camera is appropriate for 3D drawings and the orthographic camera is simpler to specify for 2D drawings. The usage of the procedures is not restricted to these cases, since all drawings are made in 3D space.

The *perspective* camera is defined by a viewpoint position (eye point) and a reference point (lookat) in world coordinates. The distance between eye point and view plane is called focal length and determines together with the aperture (height of the camera window on the view plane) and the aspect ratio (width/height) the field of view.

The orthographic camera also uses a viewpoint position and a reference point for specifying the line of sight, which is projected to the midpoint of the window. The size of the viewport is given by its width and height.

In both cases *depth clipping* is done with two clipping planes called hither and yon. The camera is *untilted* with the y axis as up direction in a right-handed coordinate system.

void **ge3d_setcamera** (point3D pos, point3D ref, float aper,
                    float focal, float aspect,
                    float hither, float yon);

Sets up a perspective camera. The arguments have the following meaning:

pos	position of the view point (eye) of the camera (in world coordinates).

ref	reference point (in world coordinates), pos and ref together determine the line of sight, which projects to the midpoint of the viewport.
aper	the height of the camera window on the view plane.
focal	the distance between the viewpoint position and the view plane.
aspect	the aspect ratio of the camera window (width/height, e.g. 4/3).
hither	distance of near clipping plane from viewpoint position, must be > 0.
yon	distance of far clipping plane from viewpoint position, must be > hither.

For a perspective projection *depth clipping* cannot be turned off, because the visible part of the view pyramid, often called view frustum, has to be transformed into a cube. This is also necessary for hidden line elimination with the z-Buffer. If (almost) no depth clipping is wished, set hither to a very low and yon to a large enough number, but setting it too high can cause more rounding errors in the z-Buffer hidden surface algorithm.

The aspect ratio used for setting up the camera should match the aspect ratio of the output window (width devided by height). Otherwise the drawing will be distorted.

```
void ge3d_ortho_cam (point3D pos, point3D ref,
                float width, float height,
                float hither, float yon);
```

Sets up an orthographic camera with the following parameters:

pos	position of the view point (eye) of the camera (in world coordinates).
ref	reference point (in world coordinates), the line of sight goes from pos to ref - as in the perspective camera model.
width	the width of the camera viewport.
height	the height of the camera viewport.
hither	distance of near clipping plane from viewpoint position, arbitrary.
yon	distance of far clipping plane from viewpoint position, must be > hither.

For an orthographic camera *depth clipping* is not restricted to regions in front of the eye. It also cannot be turned off, but also drawings behind the eye may be visible.

Example: drawing a diagonal line over the whole window.

```
point3D pos = {0.0, 0.0, 1.0};
point3D ref = {0.0, 0.0, 0.0};
point3D leftbot = {-1.0, -1.0);
point3D rightup = {1.0, 1.0};

ge3d_ortho_cam (pos, ref, 2.0 /* -1 to 1 */, 2.0 /* -1 to 1 */,
            -10, 10);
ge3d_moveto (leftbot);
ge3d_lineto (rightup);
```

B.2.10 Light Sources

For shaded drawings, light sources have to be defined. The library uses *positional* light sources in a *diffuse* lighting model. Diffuse means that the colour of a surface is independent of the current viewing position and there are no highlights.

Light sources must be first registered with an unique index and can then be turned on and off using that index. Note that the graphic library will limit the number of usable light sources (e.g. only eight in GL).

There are two routines handling the light sources:

```
void ge3d_setlightsource (int index, float R, float G, float B,
                          float x, float y, float z)
```

Registers a light source with a (unique) index *index*. The colour-intensities are given as R, G, and B, and the light is placed at position (x, y, z) (in world coordinates). The index (a small positive integer) is later used to switch on and off the light source. The light is not switched on automatically on registering.

```
void ge3d_switchlight (int index, int state)
```

Switches the light source with index *index* on (if state is not 0) or off (if state is 0).

B.2.11 Closing the Graphics Device

```
void ge3d_close ();
```

Closes the graphics device.

App. C) SDF File Format

For storing the data for the Hyper-G 3D viewer a file format was designed, which describes a 3D scene in a single file.

The SDF *scene description file* format is based on several ASCII files generated with the *Wavefront Advanced Visualizer* software [Wave91], in particular with the programs Model, Property, and Preview.

Single 3D objects are modelled with program *Model*. The output of model is an ASCII file, describing the objects as polyhedrons. These files are referred here as object files.

The colours and other properties (like textures) of object surfaces are built with *Property*. Property generates an ASCII file, which here is called material file. Light sources are also defined with Property, and stored in a light file (also ASCII).

Finally the scene is composited with *Preview* (see also Section 5.5.). This includes the placement of the objects and light sources, and the definition of cameras. The output of Preview is a binary file, but Preview includes commands for writing selected data to ASCII files.

When using Wavefront's Advanced Visualizer as an animation software package (what it is originally intended), Preview is used to set up the animation. The objects, lights, and cameras are placed for a number of key frames. Finally, *Image* (the rendering program) is used to compute the single frames of the animation.

As we are interested in a single static scene (movement is done interactively by the user), in our case the preview file contains only a single frame.

First the individual files are discussed, they are:

actor file	listing the objects of the scene
position file	defining the transformations for each object
camera files	specifying the camera
light files	specifying the light sources
material file	including all materials used in the scene
object files	containing the data of polyhedrons

Finally the composition of the files to a single SDF-file is shown.

C.1 Actor File (ACT)

The actor file lists the objects (actors) of the scene. It is generated in Wavefront's Preview with the command: `ls -O >file.act`. This is an example:

```
Obj                          Rot  Tran  Disp
Num Name      File Name  Typ Par. Prior Prior Stat Chans Colour
--- ---------- ---------- --- ---- ----- ----- ---- ----- -------
1   viewcam    dumcam.obj cam   0   xyz   rts   ON    6   RED
2   light1     white.lgt  lgt   0   xyz   trs   ON    6   GREEN
3   cube.a     cube.obj   obj   0   xyz   trs   ON   10   YELLOW
4   cube3      cube3.obj  obj   0   xyz   trs   ON    6   CYAN
5   pi         pi.obj     obj   0   xyz   trs   ON    6   BLUE
6   light2     white.lgt  lgt   0   xyz   trs   ON    6   LTBLUE
```

The fields have the following meaning:

ObjNum	number of the object (should be continuous, beginning with 1).
Name	name of the object (comment), may be used for anchor encoding (see below).
File Name	name of the object file (if type = obj) or name of the light file (if type = lgt) or ignored otherwise. Note: This field is modified before being merged to the SDF file (see C.7).
Typ	determines the type of object (obj = geometrical object, cam = camera, lgt = light, dum = dummyobject).
Parent	object number of the parent (0 means no parent).
Rot Prior	defines the order of the axis-rotations.
Tran Prior	determines the order of transformations (translation, rotation, scaling).
Disp Stat	display status of objects and lights (ON/OFF); for cameras: the first camera with Disp Stat ON is taken as default camera.
Chans	the number of channels *present* in the posfile for this object.
Colour	ignored (colour coding in Preview).

For the usability tests a standalone version of the 3D viewer was used. Therefore the anchors had to be encoded in the object names. The syntax is:

```
(filename)<Type>
```

where <Type> is a single character, 'T' for texts (default extension .txt), 'I' for TIFF raster images (default extension .tif), and '3' for a 3D scene (SDF file, default extension .sdf).

When the Hyper-G 3D viewer is fully integrated into Hyper-G the scene file will not contain any anchor description any longer. All links are then kept in an extra database (see Chapter 4.).

C.2 Position File (POS)

The position file tells the transformation of the objects. It is generated by Wavefront's PreView with the commands `o *` and `ls >file.pos`. Here is a (shortened) example:

```
   | XTRAN   YTRAN   ZTRAN    XROT    YROT    ZROT ...  ZROT
Frame|   in.     in.     in.    deg.    deg.    deg. ...  deg.
  #  |   1/ 1    1/ 2    1/ 3    1/ 4    1/ 5    1/ 6 ...   6/ 6
-----------------------------------------------------...--------
 1   | 0.0000  0.0000  41.286  0.0000  0.0000  0.0000 ... 0.0000
```

As only display static scenes are displayed, the position file contains only the data for one frame (thus the first column frame #1). Each column holds all the information for *one* channel of *one* object.

The first row specifies the kind of transformation associated with that channel. Valid targets are [XYZ]TRAN (translation), [XYZ]ROT (rotation), [XYZ]SCALE (scaling), and SCALE (overall scale).
The second row specifies the unit of measurement (ignored).
The third row determines to which object this channel belongs. N/M means that the channel value of the Mth channel of the Nth object is present. (It is assumed that the order of the objects and the number of channels of each object in the actor file correspond to the data in the position file.)
The fourth row is ignored.
Finally the fifth row specifies the channel value (a float).

C.3 Camera File (CAM)

The camera file specifies all cameras. It is generated with the PreView command
`ls -C >file.cam`. For example:

```
Obj        Proj Focal      Aspect ----------- Viewport ----------
Num Name   Type Lngth Aper Ratio Left Right Bot  Top Hither Yon
-- ------- ---- ----- ---- ----- -------------------------------
1  viewcam  P   1.00 0.860 1.330 0.00 1.00 0.24 0.78 1.00 10000.0
```

The fields mean:

ObjNum	object number of the camera (same as in actor file).
Name	name of the camera (same as in actor file).
Proj Type	should be P for perspective (O ... orthographic).
Focal Length	distance between viewpoint and view plane.
Aper	width of the viewport (at distance focal length).
Aspect Ratio	ratio with to height of the viewport.
Left, Right, Bot, Top	ignored for perspective cameras.

Hither, Yon	near and far clipping plane, measured from viewpoint.

Note: Both *position* and *direction* of the camera (the line of sight) are derived from the channel values. A camera with position (0, 0, 1), looking towards (0, 0, 0) is transformed like all other objects (see C.2 position file).

C.4 Light File (LGT)

Light files are edited with Wavefront's Property. For example:

```
intensity 0.7653 0.7635 0.7651
```

Field *intensity* specifies the RGB values of the light source. The *position* is derived from the channel values as translation from the origin (0, 0, 0) (see C.2 position file).

The current 3D Viewer supports directional light sources only, positional light sources would be a possible extension. Other fields specify spot lights or attenuation. As they can currently not be simulated in real time they are not discussed here.

C.5 Material File (MTL)

Material files are also edited with program Property. Here is an example:

```
newmtl green
Ka 0.0 0.1 0.0
Kd 0.0 0.5 0.0
illum 1


newmtl brownmetal
Ka 0.1469 0.0287 0.0000
Kd 0.2000 0.0392 0.0000
Ks 0.5020 0.5020 0.5020
illum 2
Ns 60.0000
```

The data lines have the following meaning:

newmtl xxx	start definition of material xxx.
Ka R G B	ambient RGB colour (ignored).
Kd R G B	defines the diffuse RGB colour of a face.
Ks R G B	specular RGB colour (highlight, ignored).
illum N	WF coding of the illumination model (ignored).

Ns N	specular reflection (ignored).

In the current lighting model only the *diffuse* colour is relevant. The Wavefront file format also includes specifying *textures*. They are not supported in the current version, as only very few hardware platforms are able to render textures in real-time.

C.6 Object File (OBJ)

Object files are the ASCII output of Wavefront's Model. They describe an object model with polygons. Here is a (shortened) example:

```
# Tue Jun  8 13:01:11 1993
#
#

mtllib rgb.mtl
g
v 0.000000 6.000000 0.500000
v 0.000000 5.000000 0.500000
v 5.000000 5.000000 0.500000
v 5.000000 6.000000 0.500000
...
# 24 vertices

# 0 vertex parms

# 0 texture vertices

# 0 normals

g top
usemtl ltgreen
f 1 2 3 4
f 8 7 6 5
usemtl green
f 4 3 7 8
f 5 1 4 8
f 5 6 2 1
f 2 6 7 3
g left
...
# 18 elements
```

The description of the syntax follows:

#	lines beginning with # are comment lines
v X Y Z	definition of a 3D vertex (X, Y, Z)
n X Y Z	definition of a 3D normal (X, Y, Z)

mtllib LL	material definitions are taken from file LL (default extension .mtl)
usemtl MM	use material MM for the following faces
g	grouping of vertices/faces for manipulation in Model (ignored)
f vertexindices	define a faces by a list of vertex indices
f vi1 vi2 vi3 ...	defines a face by indices of vertices/normals, given in counter clockwise order; the vertexindices can have the following formats:
f v1 v2 ...	vertex indices only (running from 1)
f v1//n1 v2//n2	vertex and vertex normal indices (running from 1; for smooth shading)

Other face formats v/t and v/t/n involve texture indices, which are ignored by the current version.

C.7 Scene Description File (SDF)

The additional ASCII files described above needed for the scene description can be generated from the (binary) Preview file with a single call of *Preview* (pv) with the following command sequence under UNIX:

```
NAME=myscene

pv $NAME.pv <<!
# Object list ("actor file")
ls -O > $NAME.act
# Channel values ("position file")
o *
ls -k > $NAME.pos
# Camera settings ("camera file")
ls -C > $NAME.cam
!
```

The combination of all these individual files to a single SDF-file also includes a modification of the *actor file* to *share object data*. There can be many identical objects in a scene (like chairs or tables in a room), but each object is only stored once in the SDF-file and the actor file gives a reference to that object for all the other ones.

When the new actor file has been built, the SDF-file is generated by concatenating all scene files into one big file, separating the parts with a sentinel character (@ was chosen). The order of the sub files in the SDF-file is:

# heading lines (comment)
@
modified actor file: for each filename of an object (polyhedron) that already has been encountered, say at object number N, the filename is replaced with &N.
@
position file: channel values for transformations (see C.2).
@
one camera file (see C.3) for each camera in the actor file (separated with @).
@
all material definitions used for the objects (see C.5).
@
one light file (see C.4) for each light source in the actor file (separated with @)
@
one object file (see C.5??) for each polyhedron, but only the first time (i.e. not when the filename begins with & in the modified actor file).
@

This can be done with a shell script including some calls to *awk* (a string processing UNIX command supporting fields and associative arrays). Here is a listing of the script:

```
#! /bin/sh

# acttosdf
#
# combines actor, position, camera, material, light, object data
# into a single scene description file ("sdf")
#
# Author : Michael Pichler
#
# created: 17 May 1993
#
# changed:  2 Jun 1993


### Argument check ###

if [ -z "$1" ]
then
  echo "acttosdf. call:  acttosdf FILE"
  echo "  to merge FILE.act, FILE.cam, FILE.pos, and"
  echo "  material, light, and object files (shared)"
  echo "  into a single scene description file FILE.sdf"
  exit 0
fi

if [ ! -f $1.act ]
then
  echo "acttosdf. error: $1.act not found."
  exit 1
fi


### Varialbes ###
```

```
# destination file
sdffile=$1.sdf
# program version
version="acttosdf, Version 1.4"
# sentinel character for separating file parts
sentinel="@"
# directory of files
directory=`dirname $1`


### Function append FILE(S)

append ()
{
# echo "- appending $* to $sdffile"
  cat $* >> $sdffile
  echo $sentinel >> $sdffile
}


### generate modified actor file (shared objects)

# objno [FILE] ... first number of object with file FILE, say NN
# further occurrences of FILE are encoded &NN

awk '
BEGIN    { i = 0
           print "# modified actor file (shared objects)"
         }
/^[1-9]/  { i++
           if ($4 == "obj" && $8 == "ON")
           { if (objno [$3] != "")
               $3 = "&" objno [$3]
             else
               objno [$3] = i
             print
           }
           else
             print
         }
/^[^1-9]/ { print }
' $1.act > $$.act


### merging part ###

echo "$version. generating $sdffile ..."

echo "# Hyper-G 3D scene description file" > $sdffile
echo "# generated with $version" >> $sdffile
echo "# DO NOT EDIT!" >> $sdffile
echo $sentinel >> $sdffile

# (modified) actor file
append $$.act

# position file
```

```
append $1.pos

# camera file
append $1.cam

# material file(s)
append $directory/[!,]*.mtl

# light files
files=`awk '
BEGIN    { ORS = " " }
/^[1-9]/ { if ($4 == "lgt" && $8 == "ON")
             print $3;
         }
' $1.act`

for a in $files
do
  echo "# $a" >> $sdffile
  append $directory/$a
done

# object files (shared)
files=`awk '
BEGIN    { ORS = " " }
/^[1-9]/ { if ($4 == "obj" && $8 == "ON")
             if (substr ($3, 1, 1) != "&")
               print $3;
         }
' $$.act`

for a in $files
do
  echo "# $a" >> $sdffile
  append $directory/$a
done


### clean up

rm $$.act


exit 0
```

The SDF-file is the only file that is needed for the Hyper-G 3D viewer. The act-, cam-, and pos-files can be deleted because they can always be reconstructed from the binary Preview file.

Example 1: modified actor file

This is the modified actor file of the usability test environment generated with the program above (the file was shortened and manually tabified since *awk* does not preserve input whitespace when changing fields):

```
# modified actor file (shared objects)
Obj                          Rot  Tran  Disp
Num Name       File Name    Typ Par.Prior Prior Stat Chans Colour
--- ---------- ------------ --- --- ----- ----- ---- ----- ------
1   viewcam    dumcam.obj   cam  0   xyz   rts   ON    6  PURPLE
2   light1     white.lgt    lgt  0   xyz   trs   ON    6  GREEN
3   room       ihci.obj     obj  0   xyz   trs   ON    6  YELLOW
4   TOPCAM     dumcam.obj   cam  0   xyz   rts   ON    6  WHITE
5   light2     white.lgt    lgt  0   xyz   trs   ON    3  MAGENTA
6   light3     white.lgt    lgt  0   xyz   trs   ON    6  PURPLE
7   light4     white.lgt    lgt  0   xyz   trs   ON    6  CYAN
8   tableA1    table.obj    obj  0   xyz   trs   ON    3  WHITE
9   A2(a.txt)T &8           obj  0   xyz   trs   ON    3  MAGENTA
10  tableB     &8           obj  0   xyz   trs   ON    3  LTBLUE
11  tableC     &8           obj  0   xyz   trs   ON    3  ORANGE
12  D(lady30)l &8           obj  0   xyz   trs   ON    3  RED
...
20  chairA1    b_chair.obj  obj  0   xyz   trs   ON    3  WHITE
21  chairA2    chair.obj    obj  0   xyz   trs   ON    3  MAGENTA
22  chairB     &21          obj  0   xyz   trs   ON    3  LTBLUE
23  C(c.txt)T  &20          obj  0   xyz   trs   ON    3  ORANGE
...
28  labelB     lisa.obj     obj  0   xyz   trs   ON    7  LTBLUE
29  labelC     lucy.obj     obj  0   xyz   trs   ON    7  ORANGE
30  labelE     keith.obj    obj  0   xyz   trs   ON    7  CYAN
31  labelD     wilma.obj    obj  0   xyz   trs   ON    7  RED
...
52  (collg)l   manual.obj   obj  0   xyz   trs   ON    4  WHITE
53  man2       &52          obj  0   xyz   trs   ON    4  GREEN
54  (man.txt)T &52          obj  0   xyz   trs   ON    4  LTBLUE
```

Example 2: SDF-file

Here is an example of a complete SDF-file (material and object file portions shortened):

```
# Hyper-G 3D scene description file
# generated with acttosdf, Version 1.4
# DO NOT EDIT!
@
# modified actor file (shared objects)
Obj                        Rot  Tran  Disp
Num Name       File Name  Typ Par. Prior Prior Stat Chans Colour
--- ---------- ---------- --- ---- ----- ----- ---- ----- -------
1   viewcam    dumcam.obj cam   0   xyz   rts   ON    6   RED
2   light1     white.lgt  lgt   0   xyz   trs   ON    6   GREEN
3   cube.a     cube.obj   obj   0   xyz   trs   ON    10  YELLOW
4   cube3      cube3.obj  obj   0   xyz   trs   ON    6   CYAN
5   pi         pi.obj     obj   0   xyz   trs   ON    6   BLUE
6   light2     white.lgt  lgt   0   xyz   trs   ON    6   LTBLUE
@
```

```
    | XTRAN   YTRAN   ZTRAN   XROT    YROT    ZROT  ...  ZROT
Frame|    in.     in.     in.    deg.    deg.    deg. ...   deg.
  #  |    1/ 1     1/ 2    1/ 3    1/ 4    1/ 5    1/ 6 ...   6/ 6
------------------------------------------------------...---------
 1   | 0.0000  0.0000  41.286  0.0000  0.0000  0.0000 ... 0.0000
@
Obj       Proj Focal      Aspect ----------- Viewport ----------
Num Name  Type Lngth Aper Ratio Left Right Bot  Top Hither Yon
-- ------ ---- ----- ---- ----- --------------------------------
1  viewcam P   1.00 0.860 1.330 0.00 1.00 0.24 0.78 1.00 10000.0
@
newmtl green
Ka 0.0 0.1 0.0
Kd 0.0 0.5 0.0
illum 1

...

newmtl ltcyan
Ka 0.0 0.2 0.2
Kd 0.0 1.0 1.0
@
# white.lgt
intensity 0.7653 0.7635 0.7651
attenuate linear 10.000000 0.000
@
# white.lgt
intensity 0.7653 0.7635 0.7651
attenuate linear 10.000000 0.000
@
# cube.obj
# Tue Jun  8 13:07:09 1993
#
#

mtllib rgb.mtl
g
v -2.500000 2.500000 2.500000
...
v 2.500000 2.500000 -2.500000
# 8 vertices
# 0 vertex parms
# 0 texture vertices
# 0 normals

g cube
usemtl ltblue
f 1 2 3 4
...
f 2 6 7 3
# 6 elements
@
# cube3.obj

...

@
# pi.obj
```

```
...

@
```

Bibliography

[Bourne82] S. R. BOURNE: *The UNIX System*. Addison Wesley (1982).

[Card91] S. K. CARD, G. G. ROBERTSON, J. D. MACKINLAY: *The Information Visualizer, an Information Workspace*. In: ACM Proceedings - CHI (1991), pp. 181 - 188.

[Chen88] M. CHEN, S. J. MOUNTFORT, A. SELLEN: *A Study in Interactive 3-D Rotation Using 2-D Control Devices*. In: ACM Computer Graphics 22/4 (August 1988), pp. 121 - 129.

[Foley90] J. FOLEY ET.AL.: *Computer Graphics - Principles and Practice*. Addison-Wesley (1990).

[Hill90] F. S. HILL, JR.: *Computer Graphics*. Macmillan Publishing Company (1990).

[Kappe91] F. KAPPE: *Aspects of a Modern Multi-Media Information System*. Dissertation. IICM, Graz University of Technology, Austria (June 1991).

[Kappe93a] F. KAPPE, H. MAURER, N. SHERBAKOV: *Hyper-G: A Universal Hypermedia System*. In: Journal of Educational Multimedia and Hypermedia. 2/1 (1993), pp. 39 - 66.

[Kappe93b] F. KAPPE, H. MAURER: *Hyper-G: Ein großes universelles Hypermediasystem und einige Spin-offs*. IICM - TU Graz und IMMIS - Joanneum Research. In: Informationstechnik und Technische Informatik 35/2 (1993), pp. 39 - 46.

[Kern88] B. W. KERNIGHAN, D. M. RITCHIE: *The C Programming Language, 2nd Edition*. Prentice-Hall (1988).

[Linton87] M. A. LINTON, P. R. CALDER: *The Design and Implementation of InterViews.* Proceedings of the USENIX C++ Workshop (Nov. 1987), pp. 256 - 267.

[Linton89] M. A. LINTON, J. M. VLISSIDES, P. R. CALDER: *Composing User Interfaces with InterViews*. IEEE Computer 22/2 (Feb. 1989), pp. 8 - 22.

[Mack90] J. D. MACKINLAY, S. K. CARD, G. G. ROBERTSON: *Rapid Controlled Movement Through a Virtual 3D Workspace*. XEROX PARC. In: ACM Computer Graphics Vol. 24, No. 2 (August 1990), pp. 171 - 176.

[Nielsen90] J. NIELSEN, R. MOLIC: *Heuristic Evaluation of User Interfaces*. In: CHI'90 Proceedings, Seattle, Washington (April 1990), pp. 249 - 256.

[Nielsen93] J. NIELSEN: *Usability Engineering.* Academic Press, London (1993).

[OpenGL93] *Open GL Reference Manual*. Addison Wesley (1993).

[PHIGS89] *Information Processing Systems - Computer Graphics - Programmers Hierarchical Interactive Graphics System (PHIGS). Part 2 - Archive File Format*. ISO/IEC 9592-2 (1989).

[Pimen93] K. PIMENTEL, K. TEIXEIRA: *Virtual Reality - Through the New Looking Glass.* Windcrest Books (1993).

[Reilly88] T. O'REILLY ET.AL.: *X Window System User's Guide (for Version 11).* O'Reilly & Associates, Inc., Newton, Massachusets (1988).

[Robert93] G. G. ROBERTSON, S. K. CARD, J. D. MACKINLAY: *Information Visualization Using 3D Interactive Animation*. In: Communications of the ACM (April 1993), pp. 57 - 71.

[Upstill] S. UPSTILL: *The Renderman Companion*. Addison Wesley.

[Scheif86] R. W. SCHEIFLER, J. GETTIS: *The X Window System.* ACM Transactions on Graphics, Vol. 5, No. 2 (Apr. 1986), pp. 79 - 109.

[Strous91] B. STROUSTRUP: *The C++ Programming Language, 2nd Edition.* Addison-Wesley (1991).

[Ware90] C. WARE, S. OSBORNE: *Exploration and Virtual Camera Control in Virtual Three Dimensional Environments*. University of New Brunswick. In: ACM Siggraph (1990), pp. 175 - 183.

[Wave91] WAVEFRONT TECHNOLOGIES: *Advanced Visualizer User's Guide.* Santa Barbara, California. Wavefront Technologies Inc. (1991).