DRAWP3D USER'S MANUAL

Joel Welling, Ph.D
Pittsburgh Supercomputing Center

DrawP3D is a subroutine library for creating three dimensional models. If you are writing a program in Fortran or C, or in some other language that can call Fortran or C subroutines, you can use DrawP3D to create 3D models, specify views of the models, and generate images of the models with the given views. The functions of DrawP3D are a lot like those of PHIGS+ or Dore' (TM) , although DrawP3D is simpler and less complete than they .

1. Introduction

When you use DrawP3D, the first thing that you do is specify a renderer. The renderer is the set of software that is used to actually do something with the model you create; without a renderer anything else you might do with DrawP3D would be pretty useless. Several types of renderers are available; see the Renderers section of this document for details. By using an appropriate type of renderer, you can either save your graphics to a file for later viewing or draw it directly to your screen. It is also possible to use more than one type of renderer simultaneously, if desired.

The model that you create is a geometrical object, or GOB for short. It can be composed of other, simpler GOBs. These GOBs might be composed of yet simpler GOBs, or they might be geometrical primitives- things like spheres, polygons, or lines. GOBs can also have attributes like color associated with them; any sub-GOBs making up the GOB with a color will have the same color unless a different color is explicitly associated with the sub-GOB. The primitives which ultimately make up the model will have the color which they inherit from the GOBs which hold them, unless that color is overridden by a color associated with the vertices in the primitive.

Light sources are also GOBs, and you have to have at least one available before you can create an image of your object. You also need to create a camera object before you can create an image; the camera contains information on the point to look from and the point to look at, among other things. Once you have got an object GOB, a light source GOB, and a camera, you can use the 'snap' function to cause an image of your model to be created. (In the case of the P3D renderer, the 'snap' function gets encoded into the P3D metafile, so that when it is rendered the image of the object you specified is created).

So, there are three basic kinds of objects; GOBs, cameras, and renderers. Since there may be more than one of them around at any given time, some method has to exist to keep them straight. This is done by assigning each object a name as it is created. These names are plain text strings which you make up. Cameras and renderers always have names; a GOB must either have a name or be contained within another GOB which has a name. (What this means will be explained in the section on GOBs below). Whenever you want to use an object you specify it by its name. If you create another object of the same type with the same name the old object will be destroyed.

DrawP3D can be called from either Fortran or C. The function names and calling parameters are different for the two languages, and are described in separate sections below. (You could call the Fortran functions from C or vice versa, but it is probably more convenient to do things in the way this manual describes). In the User's Manual section of this document, when a function name is given, we will give both the C and Fortran names. The C names are lower case and begin with 'dp_'; the Fortran names are upper case, begin with 'P', and will usually be given in parenthesis.

All of the routines in DrawP3D are functions that return integers. The value that is returned will either be either 1 (the constant P3D_SUCCESS in C) or 0 (the constant P3D_FAILURE). If the value that comes back is 1, everything went fine with the function call. If it is 0, something went wrong and the call probably had no effect. Note that the these are integer functions that start with 'P', so if you are programming in Fortran you will have to specifically declare the called functions to be integers. We're sorry about that, but all the good letters between 'I' and 'N' were already taken by other subroutine libraries .

2. Objects

2.1 Renderers

A renderer is the simplest object to deal with in using P3D. It is the renderer which transforms the information in the GOBs and cameras you create into images. Renderers are created with the dp_init_ren (PINTRN) function. The renderer creation function takes several strings as arguments. The first string is the name to assign to the renderer, the second string specifies which renderer to use, and the meaning of the third and fourth strings depends on the renderer. The renderer name (the first parameter) can be any character string, but only the first 63 characters of the string given are used.

Several renderers are currently supported, including the P3D renderer, the Painter renderer, and a renderer which uses Silicon Graphics' GL. See the appropriate sections below for information on these renderers.

A renderer can be either open or closed. While it is closed, it does nothing at all; the closed state is used to avoid sending information to the renderer for some reason. Renderers are created open, so unless you explicitly close one with the dp_close_ren (PCLSRN) function you need never think about opening or closing renderers. A closed renderer can be opened again with the dp_open_ren (POPNRN) function.

When your program is finished using DrawP3D, it should call the dp_shutdown (PSHTDN) function. This shuts down all renderers permanently- a different thing from closing them. This is necessary to make sure that any buffered information in the renderer gets dealt with before your program exits. If you wish, renderers can be shut down separately using the dp_shutdown_ren (PSHTRN) function. This must be done carefully, however, because any reference whatsoever to a GOB which was known to a renderer before it was shut down is an error after the renderer is shut down. For this reason, it is probably best just to call dp_shutdown (PSHTDN) at the end of your program and avoid possible problems.

Another interesting function for dealing with renderers is the dp_print_ren (PPRTRN) function. This function causes a text description of the renderer to be written to the standard output.

In the discussion of renderers which follows, character strings are given in double quotes:

```
"string"
```

as they would appear in the C language. Obviously Fortran programmers should use single quotes instead:

```
'string'
```

2.1.1 P3D Renderer

When the second parameter string to the renderer creation function dp_init_ren (PINTRN) is “p3d”, a P3D renderer is created. This renderer just saves the elements of the geometrical model to a P3D metafile. A number of tools exist to produce images or animations from models in this format, either interactively or using high-quality renderers like ray tracers. You may create as many P3D renderers as you want; they can all operate independently at the same time.

When creating a P3D renderer, give the file name of the P3D file to be created in the third parameter to the renderer creation function. The special file name “-” is used to cause output to be written to the Unix standard output. The fourth parameter string is ignored for this renderer type .

2.1.2 Painter Renderer

When the second parameter string to the renderer creation function is “painter”, a Painter renderer is created. This renderer used Painter’s Algorithm to draw three dimensional objects. This means simply that the object is drawn from back to front, so that closer parts of the object are drawn later and hide the parts behind them. This algorithm is fast, but it is not completely accurate- sometimes errors are made in which surfaces which should be hidden are visible or vice versa. Also, for implementation reasons and because of the need for speed, many features of DrawP3D like specular lighting and shading of colors across polygons are not supported by this renderer. Despite all this, it produces quite nice images very fast, and is very useful for scientific models.

In creating a Painter renderer, the third parameter string is used to specify the output device interface to be used. The Painter Renderer uses the output software from the PSC’s GPLOT graphics package, and so can write to any device to which GPLOT can write. The parameter strings for some devices are “xws” or “xl” for the X Window System (xl being better for monochrome displays), “cgmb” and “cgmc” for binary and clear text CGM respectively, and "ps" for postscript. The fourth parameter string specifies the file name to which output is to be written. The special string "-" means to write output to the Unix standard output. This value should also be used for output devices which do not write to a file, such as X.

Because of internal restrictions in the GPLOT device interface, only one Painter renderer can exist at a time. We hope to remove this restriction with a later release of DrawP3D .

2.1.3 GL Renderer

When the second parameter string to the renderer creation function is "gl", an interface to the Silicon Graphics GL graphics library is created. This allows high-speed rendering of DrawP3D models on displays which support GL. You may create as many GL renderers as you want; they can all operate independently at the same time.

In creating a GL renderer, the third parameter string to the renderer creation function is currently unused. The fourth parameter string is composed of a series of keyword entries. Currently supported entries are "title" for the window title and " geometry" for the window size and position (in standard X Window System format). So, the following string would produce a window with the given title and size:

```
title="my_window",geometry="256x256+0+0"
```

Note that the entire character string must be enclosed in quotes, and the internal quotes around the entries must be protected in the usual ways for the programming language used

.

2.2 Graphical Objects and Vertices

2.2.1 Colors

Graphical objects and vertices can have colors associated with them. These colors are usually given as sets of four values between 0.0 and 1.0, these being the red, green, and blue intensities of the color and its opacity. The opacity value means simply that; a color with a 1.0 opacity is completely opaque, while a color with a 0.0 opacity is completely transparent. For historical reasons, the opacity component of color is usually denoted by the letter A, so the red, green, and blue components plus opacity would be denoted RGBA. Not all renderers can draw partly transparent surfaces correctly, but some can, and this mechanism can be used to set the transparency when it is possible to do so. For example, a green surface which let 25% of the light striking it pass through to the other side would have RGBA components (0.0,1.0,0.0,0.75). Future releases of P3D may support other color systems, like HLSA.

The appearance of an object depends on more than just the object's color. The placement of light sources (see below) relative to the object will certainly effect its appearance. Also, the material type (see below) from which the object is made will effect its appearance .

2.2.2 Coordinate Systems and Vertices

P3D assumes that all coordinate systems are right-handed. For example, if the x axis of a coordinate system points to the right of a viewer and the y axis points up, the z axis will point toward the viewer. The signs of rotations are also assumed positive in the right-handed direction. For example, a positive rotation of 90 degrees about the z axis will move a vector initially pointing along the x axis into alignment with the y axis. Rotations and other coordinate transformations are discussed in more detail later in this document.

Vertices are used to define the shapes of certain geometrical primitive objects, for example polygons or polylines. Not all primitives need vertex information; for example spheres do not. A vertex is basically a point in 3D space, but it can carry with it some other information as well. In addition to position, a vertex can carry information about the color of that point or the surface normal vector at that point. No vertex is required to carry this extra information, however; if it is not given DrawP3D will make reasonable assumptions about what values to use.

For example, suppose we wanted to define a five-sided polygon. To do so, we would have to define five vertices, and the polygon would be created to span the space that they enclosed. (To do this, all five points would have to lie in the same plane). If color information was given with each vertex, the renderer would do its best to draw the polygon so that its color shaded continuously from the color of one vertex to that of another as one crosses the polygon between the two vertices. Note that a color given by the vertices of a primitive overrides any color that the primitive may inherit from other GOBs which might contain it (see the section on attribute inheritance below).

DrawP3D allows another mechanism for specifying the colors of vertices, called a color map. A color map is a special function for translating a floating point scalar value into a set of RGBA color values. Any time a color can be given for a vertex, a single scalar can be given instead and the current color map will be used to translate that value into a color. See the section on color maps below for more information.

The meaning of the normal vector given at a vertex is a little more subtle. The normal vector is the direction which the renderer assumes will be perpendicular to the surface of the primitive being drawn at the given vertex. Because the appearance of the light falling on the surface depends on these normal vectors, the normals can be used to change the apparent shape of the surface. This is basically a rendering trick to make surfaces made up of flat polygons look curved.

Any vertex can have colors and/or normals, but not all primitives will make use of the information given. For example, a polyline (a collection of connected line segments) will not use normal information because there is no such thing as a geometrical normal to a line.

Because it is more convenient to specify vertex information in one format in Fortran and another format in C, different mechanisms are used in the two different languages. This is the biggest difference between the Fortran and C interfaces to DrawP3D .

2.2.3 Color Maps

A color map is a special function which can generate a color for a vertex from a single floating point value. This can be very useful, for example when one wants to color a surface based on some physical quantity like pressure at the surface. Note that this only works for vertex colors; you cannot use a color map to specify the color of an entire GOB.

A color map is a genuine function, not a lookup table as those familiar with some other graphics systems might expect. At any given moment there is one valid color map, set by the function dp_set_cmap (PSTCMP). This function takes a minimum and maximum value and a function as parameters. When it becomes necessary to translate a real number into a color, that number is rescaled using the minimum and maximum values and the map function provided by the user is called to convert the rescaled value into a color.

The rescaling works as follows. If the value given is less than the minimum value, it is mapped to 0.0 . If it is greater than the maximum, it is mapped to 1.0 . If it falls in between, it is mapped to (value-minimum)/(maximum-minimum), which lies between 0.0 and 1.0 . The function provided by the user is then required to translate this value, which lies in the range 0.0 to 1.0 inclusive, to a color.

This provides a very powerful, general mechanism for setting up your own color maps, but a simpler mechanism is also available. The function dp_std_cmap (PSDCMP) provides a set of predefined color map functions. It takes the same minimum and maximum values as parameters, plus an integer to specify the map .

2.2.4 Graphical Objects (GOBs)

A graphical object, or GOB, is the basic element out of which geometries are built in DrawP3D. GOBs come in two types; named and unnamed.

Named gobs are created using the dp_open (POPEN) and dp_close (PCLOSE) functions, and are given a name when the user passes a character string to dp_open (POPEN). That name can be used as a handle to refer to the GOB after it has been created. Any character string except the null string ("" in C) or a string consisting of a single space is a valid GOB name. Only the first 63 characters in a GOB name are relevant; names differing after the 63rd character are not treated as being unique.

Between the calls to dp_open (POPEN) and dp_close (PCLOSE), things can be added to the open GOB. For obvious reasons the GOB to which things can be added at any given moment is called the current GOB. These include attributes and transformations, which are described in separate sections below, and other GOBs which become children of the open GOB. Those other GOBs might be previously-created named GOBs, special GOBs containing geometrical primitives, and/or nameless GOBs.

Adding a named GOB as a child of another named GOB is done using the dp_child (PCHILD), which takes the name of the gob to be added as its parameter. The definition of that GOB must already have been closed by the dp_close (PCLOSE) call, as only one named GOB is allowed to be open at a time. Adding geometrical primitive GOBS is accomplished by calling special functions, like dp_sphere (PSPHR) in the case of a sphere primitive. These functions are discussed in their own section below.

To add a nameless GOB as a child of a named GOB, the dp_open (POPEN) and dp_close (PCLOSE) calls are used again. In this case, however, an empty character string is given as a parameter to dp_open (POPEN). (For compatibility with Fortran 77, a character string consisting of a single blank is considered equivalent to an empty string). This tells P3D that the GOB to be created is nameless. Since only one named GOB may be open at a time, it is an error to try to create a new named GOB before the dp_close (PCLOSE) call has been used to close the last named GOB. Thus the open and close calls can be used in a nested way, with the outermost calls being for the named GOB. All more deeply nested pairs of open and close calls will create children of that GOB, or of nameless GOBs which are descendants of that GOB.

GOBs can be rendered, as will be described in a later section. Another interesting thing to do with a GOB is to print it, via the dp_print_gob (PPRTGB) function. This function causes a description of the GOB and all its descendants to be written to the standard output. This can be a very useful debugging tool .

2.2.4.1 GOB Memory Management

A GOB exists as long as it or any GOB of which it is a descendant has a name. A name can be removed from a named GOB either explicitly (via the dp_free (PFREE) function) or by the creation of a new GOB with the same name. This means that the easiest way to destroy a named GOB and all of its nameless descendants is to create a new GOB with the same name. Any descendants which have names will survive, at least until something happens to remove their names. The effect of this mechanism is that the memory associated with a GOB is released as soon as it becomes impossible for the programmer to refer to the GOB again.

For example, suppose we created a GOB with the name 'fred', by calling dp_open (POPEN) with the parameter 'fred' followed by dp_close (PCLOSE). We might then create two new named GOBs, named 'bob' and 'tom' respectively, both of which had the 'fred' GOB as a child. (The child GOB would be added using the dp_child (PCHILD) function). We might then explicitly free the 'fred' GOB, or create a new GOB named 'fred'.

The GOB which originally had the name 'fred' would then be nameless, but it would still be a child of both 'bob' and 'tom' and thus would continue to exist. If the 'bob' GOB were freed or its name were reused, the original 'fred' GOB would continue to exist because it would still be descended from 'tom'. If 'tom' lost its name, however, all memory of the original 'fred' would be lost .

2.2.4.2 Primitives

Primitives are a type of nameless GOB which explicitly contains geometry, rather than being composed of child GOBs which contain geometry. Because these GOBs are nameless, they cannot have attributes or transformations attached to them but will inherit them from parent GOBs. DrawP3D provides the following primitives:

type	C routine	Fortran routine
sphere	dp_sphere	PSPHR
cylinder	dp_cylinder	PCYL
torus	dp_torus	PTORUS
polymarker	dp_polymarker	PPLYMK
polyline	dp_polyline	PPLYLN
polygon	dp_polygon	PPLYGN
triangle strip	dp_tristrip	PTRIST
mesh	dp_mesh	PMESH
Bezier patch	dp_bezier	PBEZP
text	dp_text	PTEXT

Spheres, cylinders, and tori are the simplest primitives. Spheres are always of unit radius and centered at the origin. Cylinders are also of unit radius; they stretch from the origin to the point (0.0,0.0,1.0). Tori lie in the x-y plane, are centered at the origin, and have variable major and minor radii.

All of these primitives are placed at the origin for simplicity. To draw one at another position or of a different size, use a transformation (described below). If a transformation is associated with a GOB which has a primitive as a child, the primitive will be drawn in the position specified by the transformation.

The polymarker, polyline, and polygon primitives are just that: a collection of point markers, a collection of connected line segments, and a polygon. Their shape is defined by a list of vertices. In the case of the polygon, the vertices should be coplanar, because the behavior of the renderer will be hard to predict if they are not. Some renderers cannot handle polygons which are not convex, so they should also be avoided when possible. The 'front' face of a polygon is determined by applying the right hand rule to its list of vertices. This is important for purposes of backface culling, which is described in the section on attributes below.

A triangle strip is also defined by a list of vertices, but in a slightly different way. Given a list of vertices, the triangle strip creation function will draw a triangle through the first three, another through the second three (so that the second vertex is used again), a third through the third three, and so on. This means that if it is given ten vertices, it will end up producing eight triangles. The front face of the first triangle is determined by the right hand rule, the second uses a left hand rule, the third right, and so on. This rule means that the front face is continuous down the length of the strip.

The mesh primitive consists of a list of vertices, and a list of polygons defined using those vertices. It is used quite a bit to create complex shapes. Most renderers will try to make the boundaries between polygons in a mesh look smooth by interpolating the normal vectors of the vertices of the mesh. As with single polygons, the vertices of each polygon in a mesh should be coplanar and have a front face determined by the right hand rule.

A Bezier patch is a special primitive defined by a list of exactly 16 vertices. These vertices act as control points for interpolation of a bicubic spline surface, as illustrated below.

```
*-------*-------*-------*

v1      v2      v3      v4

*-------*-------*-------*

v5      v6      v7      v8

*-------*-------*-------*

v9      v10     v11     v12

*-------*-------*-------*

v13     v14     v15     v16
```

The edge of the patch passes through the outer 12 vertices; the remaining four vertices control the shape of the patch but do not actually fall within the surface. The use of color and normal information associated with the vertices defining a patch is highly renderer-dependent. The 'front' surface of the patch is defined by the right hand rule circulating in the v1-v4-v16-v13 order, so that the above illustration's 'front' direction is into the page.

The text primitive is just that- a string of text. The primitive includes the text itself, a starting location, and two vectors which define the writing direction and the 'up' direction for the text. For example, this user's manual is written with a writing direction which runs from left to right across the page and an up direction which is toward the top of the page. The size of text is effected by the 'text-height' attribute; other text attributes will be added in the future .

2.2.4.3 Composite GOB Routines

Composite GOBs are like primitives in that they can be created with a single DrawP3D call, but on a lower level they are actually composed of several primitives. As a user, you never need to consider the internal structure of a composite GOB, but we make the distinction because there really is a difference in the computer graphics sense. Like primitives, composite GOBs are nameless, and cannot have attributes or transformations attached to them directly. DrawP3D provides the following composite GOBs:

type	C routine	Fortran routine
bounding box	dp_boundbox	PBNDBX

axis	dp_axis	PAXIS
isosurface	dp_isosurface	PISOSF
Z surface	dp_zsurface	PZSURF
random isosurface	dp_rand_isosurf	PRNISO
random Z surface	dp_rand_zsurf	PRNZSF

The bounding box composite function creates a bounding box using the two corners passed to it as opposite ends of the diagonal of the box.

The axis composite function creates an axis. Start and end specify the start and end points of the axis. A text label is written in the direction in which the axis runs, that is, from the start point towards the end point, with the 'up' vector used as usual for the text primitive. The tics hang down from the axis line, toward the text label. The number of tics (including the starting and ending tics) is also specified. Starting and ending values for the axis tic marks are also given, and they are interpolated for other labels to be placed at the remaining tics. There is a label of text placed below the starting val, and it along with the tic labels are scaled by a text_size parameter. Lastly, there is a parameter for the number of decimal places to be displayed, which can range from zero and 8.

The isosurface composite function is used to extract a surface of constant value from a three dimensional array of real data. The best way to understand what this means is to think of a continuous function in three dimensional space. Consider all the points in space for which the function has a given value. Because the function is continuous, the points must group together to form two dimensional closed surfaces in space. These surfaces are iso-valued surfaces, or isosurfaces, of the continuous function. The isosurface function assumes that the grid of data that it is passed samples such a continuous function, and extracts the isosurface of that function at a given value. It can use a second grid of data to provide values to color the surface (via the current color map) if desired. This feature can be used to produce, for example, surfaces of constant pressure which are colored according to the local temperature.

A Z surface is a composite GOB made of a rectangular mesh located in the x-y plane, with the z component specified by the user in an array. The user can specify with each point a scalar value that will be used to color the surface (via the current color map) if desired. The user can also include a user created exclusion function for each point. Each data point in the surface is checked with this function, and if the result so indicates the point is dropped, leaving a hole in the surface. The exclusion function can make its decision based on either the z-value at that point or the x and y indices in the array.

In addition to being able to construct an isosurface from data given on a regular grid, DrawP3D provides a function to construct an isosurface from data given at arbitrary (non-gridded) points in 3D space. The user provides the random isosurface routine with a set of x-y-z coordinate values, and a scalar value associated with each point. The routine calculates the Dirichlet tesselation of the points, which is a way of finding appropriate neighbor relationships between the points, and then applies a 'Marching Tetrahedrons' algorithm to extract an isosurface from the known values and neighbor relationships. The isosurface is bounded by the convex hull of the data, and may be colored by applying the current color map to a second value associated with each point. The calculation of the Dirichlet tesselation is time-consuming and very memory-intensive, so the regular isosurface routine should be used when possible.

A routine is also provided to construct a Z surface from a collection of arbitrary (non-gridded) points in the x-y plane. The user provides the random Z surface function with a set of x-y-z coordinate values for which the z values are known to form a single-valued function of x and y. The routine calculates the Dirichlet tesselation of the points and draws the surface. The surface produced by the routine is bounded in x and y by the convex hull of the given data. The calculation of the Dirichlet tesselation is time-consuming and very memory-intensive, so the regular Z surface routine should be used when possible .

2.2.4.4 Coordinate Transformations

A coordinate transformation is also a characteristic of a GOB. Unlike attributes, transformations accumulate from a GOB to its children. This makes it possible to create an object from an appropriate collection of primitives and transformations, and then move the object around as a whole by adding transformations to a parent GOB of the object.

Transformations are represented as four-by-four matrices, as follows:

```
rotation:

        [ R11 R12 R13 0 ]
```

```
[ R21 R22 R23 0 ]
[ R31 R32 R33 0 ]
[  0   0   0  1 ]
```

```
translation:
```

```
[  1   0   0  Tx ]
[  0   1   0  Ty ]
[  0   0   1  Tz ]
[  0   0   0  1  ]
```

```
scale:
```

```
[ Sx   0   0  0 ]
[  0  Sy   0  0 ]
[  0   0  Sz  0 ]
[  0   0   0  1 ]
```

where R11 through R33 are appropriate rotation components, Tx, Ty, and Tz are translation components in the x, y, and z directions respectively, and Sx, Sy, and Sz are scaling factors in the x, y, and z directions respectively. With these conventions, a point in three dimensional space will be represented by a column vector, the first three components of which are the x, y, and z coordinates of the point and the fourth component of which is always unity. Transforming that point to a new coordinate system then corresponds to left multiplying the column vector by the appropriate transformation matrix.

The function dp_transform (PTRNSFM) exists to allow the user to specify these matrices explicitly, but that is almost never necessary. The transformation functions are:

purpose	C routine	Fortran routine
general transformation	dp_transform	PTRNSFM
translation	dp_translate	PTRANS
rotation	dp_rotate	PROTAT

scale	dp_scale	PSCALE
anisotropic scale	dp_ascale	PASCAL

If you add more than one transformation to a GOB, the transformations accumulate, so that the net transformation is the product of the matrices of the individual transformations. Because matrix multiplication is order dependent, this means that the order in which you add transformations to the GOB is important. Newly added transformations premultiply (left multiply) the transformation currently associated with the GOB, in keeping with the matrix multiplication scheme described above. A GOB to which no transformation has been added effectively has the identity transformation associated with it .

2.2.4.5 Attributes and Attribute Inheritance

Attributes are also characteristics of GOBs. A typical attribute would be the color of the GOB, but just about anything can be an attribute. Attributes have names, which are text strings, and values which can be of a variety of types. The order in which you add attributes to GOBs does not matter; all attributes added apply to the whole GOB. It's a good idea to avoid adding two attributes with the same name to the same GOB, as which one gets used may depend on the renderer.

If a GOB has child GOBs (primitives or other composite GOBs), these GOBs inherit the attributes of the parent GOB. This inheritance can be overridden by an attribute of the same name attached to the child itself. For example, if a GOB has associated with it a 'color' attribute which is red, all of its descendents will also have the 'color' red. If some descendent had a 'color' attribute which was blue, however, the blue 'color' would override the red for it and any GOB descended from it.

The following functions add an arbitrary attribute of a given type to a GOB. Each takes an attribute name and value as its arguments.

type	C routine	Fortran routine
integer	dp_int_attr	PIATT
boolean	dp_bool_attr	PBLATT
float	dp_float_attr	PFATT
string	dp_string_attr	PSTATT

color	dp_color_attr	PCLATT
point	dp_point_attr	PPTATT
vector	dp_vector_attr	PVCATT
transformation	dp_trans_attr	PTRATT
material	dp_material_attr	PMTATT

The integer, boolean, float, and string attribute types are just what their names imply. Color, point, vector and transformation data types have been described earlier in this document, and materials will be described below.

Certain special functions exist to set the values of common attributes. They are:

purpose	C routine	Fortran routine
set GOB color	dp_gobcolor	PGBCLR
set text height	dp_textheight	PTXTHT
control backface culling	dp_backcull	PBKCUL
set GOB surface material	dp_gobmaterial	PGBMAT

Attributes have the same defaults in DrawP3D that they do in P3D. For most attributes, the default is a null value, which means that the renderer will fill in the value for the attribute as needed. Some attributes have explicit defaults, however. They are:

attribute	default
color	opaque white (RGBA = (1.0, 1.0, 1.0, 1.0))
backcull	false
text-height	1.0
material	default material (renderer specific)

2.2.4.6 Materials

It takes more than just a color to determine the appearance of an object. For example, a piece of blue chalk will look different from a piece of blue metal or a piece of blue plastic, even if the blue color is the same in all cases. The reason for this is that, when light strikes an object, it scatters or is reflected in ways that depend on the material from which the object is made.

DrawP3D provides a simple facility to specify the material used to make an object. This is done by associating a material type with an object; the material type is an attribute of the object just as color is an attribute.

At the moment, only a limited set of material types are available through DrawP3D. Material types are provided to produce the following appearances:

material	examples
default	renderer dependent
dull materials	paper
shiny non-metallic materials	plastic or ceramics
metallic	chrome
aluminum	aluminum (less glossy than chrome)
matte	no specular reflection

The default material type exists for backward compatibility, so that models produced before material types were added to DrawP3D can still be drawn correctly .

2.2.4.7 Light Sources

Given all these methods of defining models, we still need light sources to view them by. Lights are GOBs, and can be treated in exactly the same ways as other GOBS. In fact, the same GOB can have a light source and a geometrical primitive as children.

DrawP3D currently supports two types of lighting primitive, ambient and positional light sources. Ambient lights are produced by the dp_ambient (PAMBLT) function. This function takes a color as an argument, and light of that color illuminates all surfaces of the model uniformly. Positional lights are produced with the dp_light (PLIGHT) function, which takes a location as well as a color as arguments. Some renderers may retreat positional light sources to infinite distance, if they can't handle the position as it is given.

In general it is good to use both ambient and positional lighting in a scene. This is done by combining one or more of each light type as children of a GOB, and then using that

GOB to specify the illumination for the scene. For convenience, there is a predefined GOB called 'standard_lights' which contains an ambient light and a single positional light source. To see the exact components of 'standard_lights', one can print its contents using the dp_print_gob (PPRTGB) function .

2.3 Cameras

Now that a model and a collection of light sources are available, all that remains to be specified before a scene can be drawn is the viewing information. This is done by specifying a 'camera', a data structure which contains (among other things) information about what to look at and where to look from. In doing three dimensional graphics, one of the most common sources of trouble is mis-specifying the camera, so if you attempt to render a scene and there is nothing in it, check to be sure that your camera is set up appropriately.

Cameras are created by the dp_camera (PCAMRA) function. As parameters it takes a name to give the new camera, and a variety of information specifying the view. That information includes a point to look at and a point to look from, and an 'up' vector to specify the rotational orientation of the camera around the axis specified by the two points. A fovea is also required; this specifies the opening angle of the view, so that going to a smaller fovea essentially 'zooms in' on the model. The name of the new camera can be any character string, but only the first 63 characters of the name are actually used.

The camera creation function also requires the user to specify hither and yon clipping distances. The hither distance is the distance to the nearest object which will be rendered; the yon distance is the distance to the farthest object rendered. Because the coordinate system at the camera is assumed to be a right handed coordinate system, the hither and yon distances are negative numbers, with the hither distance having the smaller magnitude. Computer graphics renderers generally need to use this information, although there is no corresponding quantity with a real camera. The hither and yon distances can be useful-by setting the hither distance to fall within a model, for example, one can cut the model in half and view its internal structure.

It is a good idea to keep the hither and yon boundaries fairly close to the object being rendered, as some renderers will make errors if the range between them is too large. These errors are characterized by 'banding' across the rendered image, or surfaces which should be hidden showing through the surfaces which should hide them. At the same time, it is a good idea to leave some margin between the boundaries part of the object and the hither and yon clipping planes. This allows the camera to be moved (for example, in viewing a P3D model) without worrying about the clipping limits slicing into the object. A reasonable compromise is to set the hither distance to half the distance to the object, and the yon distance to twice the distance to the object.

The camera also controls the background color a scene has when it is rendered. This may seem a little non-intuitive, but it makes sense when you think about it - the camera controls all aspects of the rendered scene other than the object and the lights, and the background color is certainly not an aspect of the object or the lights. All cameras are created with their background color set to black. To change the background color of a camera, use the dp_camera_background (PCMBKG) function.

For convenience, there is a predefined standard camera with the name 'standard_camera'. It looks at the origin from a point along the positive Z axis. The contents of this or any other camera can be printed using the dp_print_camera (PPRTCM) function .

3. Rendering with the Snap Function

Once the GOBs representing the model and the light sources and an appropriate camera are available, it is an easy matter to render the model. This is done with the dp_snap (PSNAP) function, which takes all three as parameters.

The snap function freezes the view information and sends it off to all the currently open renderers. The P3D renderers will store all the information in their output P3D files, where it can be accessed when the P3D models are viewed .

4. Implementor's Notes

DrawP3D is built in two layers. The lower layer is called P3DGen; it is very object-oriented despite the fact that it is written in standard C. A single module, p3dgen.c, provides a non-object-oriented interface to the P3DGen objects. The upper layer of DrawP3D simply calls entry points in that module; the Fortran and C interfaces are completely disjoint and are coded in drawp3d_fi.c and drawp3d_ci.c respectively. Note that the C language routines of drawp3d_fi.c are linked directly against the user's Fortran code. If someone were inclined to add a new language interface, it would be straightforward to do so.

The enterprising programmer might be inclined to write code which would call the P3DGen objects directly. This is not difficult to do; see p3dgen.c for examples. It is necessary to maintain certain globals which are used by the object routines to identify themselves.

Macros with names beginning with METHOD_ are provided to assist in doing so. Code which fails to use them will generally produce a core dump almost immediately.

Future development in DrawP3D will (hopefully) include the following projects. We may add additional renderers, and additional composite routines will probably continue to be added.

The code for DrawP3D does a number of tricky but legal things, and it has been known to trip up compilers. If your compiler has difficulty with it, consider getting the GNU C compiler (gcc) from the Free Software Foundation. It handles DrawP3D and just about everything else we have seen very well, and produces quite fast code.

The form in which users specify vertex data lists is determined by which of several vertex list objects is used. There is an unused format provided by f_vlist_mthd.c which takes each of the possible data values (x, y, z, normal x, y, z, etc.) as separate parameters. This could be quickly swapped into either the Fortran or C language interfaces if that was desirable.