NAME

VOGLE − A very ordinary graphics learning environment.

DESCRIPTION

VOGLE is a library of C routines for doing line drawings and polygon fills in 2 and 3 Dimensions. It handles circles, curves, arcs, patches, polygons, and software text in a device independent fashion. Simple hidden line removal is also available via polygon backfacing. Access to hardware text and double buffering of drawings depends on the driver. There is also a FORTRAN interface but as it goes through the C routines FORTRAN users are warned that arrays are in row-column order in C. A SUN Pascal interface has also been provided. The following is a brief summary of the VOGLE subroutines.

Include files.

There are two include files provided with vogle: vogle.h and Vogle.h. The lowercase vogle.h is the C header file which most vogle C programs would include. The uppercase Vogle.h is the SUN Pascal header file which contains the forward/external declarations for Pascal. This header file should be included just after the program statement of a Pascal program.

The following is a brief summary of the VOGLE subroutines.

Using X toolkits and Sunview

For X11 and Sunview based applications, it is posible for VOGLE to use a window that is supplied by that application's toolkit. Under these circumstances, the toolkit is is responsible for handling of all input events, and VOGL simply draws into the supplied window. These calls are only available from C. Also see the directories examples/xt, examples/xview and examples/sunview.

For X based toolkits the following three calls may be used:

vo_xt_window(display, xwin, width, height)

Tells VOGLE to use the supplied window *xwin*

```
vo_xt_window(display, xwin, width, height)
        Display  *display;
        Window xwin;
        int      width, height;
```

This routine should be called before calling "vinit()".

vo_xt_win_size(width, height)

Tells VOGLE that the supplied window has changed size.

```
vo_xt_win_size(width, height)
        int      width, height;
```

vo_xt_set_win(display, xwin, width, height)

Just tells VOGLE to draw into that particular 'Drawable'

```
vo_xt_set_win(display, xwin, width, height)
        Display  *display;
        Drawable xwin;
        int      width, height;
```

vo_xt_get_display()

Returns the current display that VOGLE beleives it is using.

```
Display *
vo_xt_get_display()
```

vo_xt_get_window()

Returns the current window that VOGLE beleives it is using.

```
Window
vo_xt_get_window()
```

vo_xt_get_GC()

Returns the current Graphics Context that VOGL beleives it is using.

```
GC
vo_xt_get_GC()
```

For sunview based applications the following two calls may be used:

vo_sunview_canvas(canvas, width, height)

Tells VOGLE to use the supplied sunview canvas *canvas*

```
vo_sunview_canvas(canvas, width, height)
        Canvas  canvas;
        int     width, height;
```

This routine should be called before calling "ginit()".

vo_sunview_canvas_size(width, height)

Tells VOGLE that the supplied canvas has changed size.

```
vo_sunview_canvas_size(width, height)
        int     width, height;
```

Device routines.

vinit(device)

Initialise the device.

Fortran:

```
subroutine vinit(device)
character *(*) device
```

C:

```
vinit(device);
char    *device;
```

Pascal:

```
procedure Vinit(device: string_t)
```

Note 1 :- Current available devices are:

tek - tektronix 4010 and compatibles
hpgl - HP Graphics language and compatibles
hpgt - HP Graphics Terminals and compatibles
dxy - roland DXY plotter language

postscript - mono postscript devices
ppostscript - mono postscript devices (portrait mode)
cps - colour postscript devices
pcps - colour postscript devices (portrait mode)
sun - Sun workstations running sunview
X11 - X windows (SUN's OPenwindows etc etc)
next - NeXTStep and other NeXT platforms
decX11 - the decstation window manager
(basically obsolete).
apollo - Apollo workstations
(basically obsolete).
hercules - IBM PC hercules graphics card
cga - IBM PC cga graphics card
ega - IBM PC ega graphics card
vga - IBM PC vga graphics card
sigma - IBM PC sigma graphics card.
mswin - IBM PC Microsoft Windows.
os2pm - IBM PC OS/2 Presentation Manager
grx - the contributed GRX graphics library that
is available with DJGPP (GNU for PCs).

Sun, X11, apollo, grx, mswin, os2pm, next, hercules, cga and ega support double buffering.

Note 2 :- If device is a NULL or a null string the value of the environment variable "VDEVICE" is taken as the device type to be opened.

Note 3 :- after init it is wise to explicitly clear the screen.

e.g.: in C
```
color(BLACK);
clear();
```

or in Fortran
```
call color(BLACK)
call clear
```

or in Pascal
```
Color(BLACK);
Clear;
```

vexit() Reset the window/terminal (must be the last VOGLE routine called)

Fortran:
```
subroutine vexit
```

C:
```
vexit()
```

Pascal:
```
procedure Vexit;
```

voutput(path)

Redirect output from *next* vinit to file given by path. This routine only applies to devices drivers that write to stdout e.g. postscript and hpgl.

Fortran:

```
subroutine voutput(path)
character*(*) path
```

C:

```
voutput(path)
        char    *path;
```

Pascal:

```
procedure Voutput(path: string_t)
```

vnewdev(device)

Reinitialize VOGLE to use a new device without changing attributes, viewport etc. (eg. window and viewport specifications)

Fortran:

```
subroutine vnewdev(device)
character *(*) device
```

C:

```
vnewdev(device)
        char *device;
```

Pascal:

```
VnewDev(device: string_t)
```

vgetdev(device)

Gets the name of the current VOGLE device. The C version of the routine also returns a pointer to it's argument.

Fortran:

```
subroutine vgetdev(device)
character *(*) device
```

C:

```
char *
vgetdev(device)
        char *device;
```

Pascal:

```
procedure VgetDev(var device: string_t)
```

pushdev(device)

Initialize a new device without changing attributes, viewport etc, but save the previously initialised device on a stack.

Note, this is intended to completely change the device, it won't work if you pushdev the same device that you are already running. (This will be fixed at a later date).

Fortran:
```
subroutine pushdev(device)
character *(*) device
```

C:
```
pushdev(device)
        char *device;
```

Pascal:
```
PushDev(device: string_t)
```

popdev()

Pops a device off the device stack and reinstates the previously pushed device.

Fortran:
```
subroutine popdev()
```

C:
```
popdev()
```

Pascal:
```
PopDev
```

getdepth()

Returns the number of bit planes (or color planes) for a particular device. The number of colors displayable by the device is then 2**(nplanes-1)

Fortran:
```
integer function  getdepth()
```

C:
```
int
getdepth()
```

Pascal:
```
function GetDepth(): integer;
```

Routines For Setting Up Windows.

Some devices are basically window orientated - like sunview and X11. You can give vogle some information on the window that it will use with these routines. These can make your code very device independant. Both routines take arguments which are in device space. (0, 0) is the top left hand corner in device space. To have any effect these routines must be called before vinit. For the X11 device, an entry may be made in your .Xdefaults file of the form vogle.Geometry =150x500+550+50 (where you specify your geometry as you please).

prefposition(x, y)

Specify the preferred position of the window opened by the *next* vinit.

Fortran:
```
subroutine prefposition(x, y)
integer x, y
```

C:

prefposition(x, y)
int x, y;

Pascal:

procedure PrefPosition(x, y: integer)

prefsize(width, height)

Specify the preferred width and height of the window opened by the *next* vinit.

Fortran:

subroutine prefsize(width, height)
integer width, height

C:

prefsize(width, height)
int width, height;

Pascal:

procedure PrefSize(width, height: integer)

General Routines.

clear() Clears the screen to the current colour.

Fortran:

subroutine clear

C:

clear()

Pascal:

procedure Clear

color(col)

Set the current colour. The standard colours are as follows:

black = 0 red = 1 green = 2 yellow = 3
blue = 4 magenta = 5 cyan = 6 white = 7.

Fortran:

subroutine color(col)
integer col

C:

color(col)
int col;

Pascal:

procedure Color

mapcolor(indx, red, green, blue)

Set the color map index indx to the color represented by (red, green, blue). If the device has no color map this call does nothing.

Fortran:

```
subroutine mapcolor(indx, red, green, blue)
integer indx, red, green, blue
```

C:

```
mapcolor(indx, red, green, blue)
        int     indx, red, green, blue;
```

Pascal:

```
procedure MapColor(indx, red, green, blue: integer)
```

clipping(onoff)

Turn clipping on or off. Non-zero is considered on. Note: on some devices turning clipping off may not be a good idea.

Fortran:

```
subroutine clipping(onoff)
logical onoff
```

C:

```
clipping(onoff)
        int     onoff;
```

Pascal:

```
procedure Clipping(onoff: boolean)
```

getkey()

Return the ascii ordinal of the next key typed at the keyboard. If the device has no keyboard getkey returns -1.

Fortran:

```
integer function getkey
```

C:

```
int
getkey()
```

Pascal:

```
function GetKey(): integer;
```

checkkey()

Returns zero if no key is pressed or the ascii ordinal of the key that was pressed.

Fortran:

```
integer function checkkey()
```

C:

```
int
checkey()
```

Pascal:

```
function CheckKey(): integer;
```

getstring(bcol, string)

Read in a string, echoing it in the current font, using the current color and the current transformation. bcol is the background color which is used for erasing characters after a backspace or a delete key is received. Getstring interprets the Backspace key (ascii 8) and the Del key (ascii 127) as erasing characters. An EOT (ascii 4) or a Carraige return (ascii 13) will terminate input. Getstring returns the number of characters read. Getstring does not check for overflow in the input buffer string

Fortran:
```
integer function getstring(bcol, string)
integer bcol
character *(*) string
```
C:
```
int
getstring(bcol, string)
        int     bcol;
        char    *string;
```

Pascal:
```
function GetString(bcol: integer; var string: string_t): integer;
```

locator(xaddr, yaddr)

Find out where the cursor is. xaddr and yaddr are set to the current location in world coordinates. The function returns a bit pattern which indicates which buttons are being held down eg. if mouse buttons 1 and 3 are down locator returns binary 101 (decimal 7). The function returns -1 if the device has no locator capability. Note: if you have been doing a lot of 3-D transformations xaddr and yaddr may not make a lot of sense. In this case use slocator.

Fortran:
```
integer function locator(xaddr, yaddr)
real xaddr, yaddr
```
C:
```
int
locator(xaddr, yaddr)
        float   *xaddr, *yaddr;
```

Pascal:
```
function Locator(var xaddr, yaddr: real): integer;
```

slocator(xaddr, yaddr)

Find out where the cursor is. xaddr and yaddr are set to the current location in screen coordinates. The return value of the function is set up in the same way as with locator. If the device has no locator device *slocator* returns -1.

Fortran:
```
integer function slocator(xaddr, yaddr)
real xaddr, yaddr
```
C:
```
int
slocator(xaddr, yaddr)
        float   *xaddr, *yaddr;
```

Pascal:
```
function Slocator(var xaddr, yaddr: real): integer;
```

Routines for controling flushing or syncronisation of the display.

On some devices (particularly X11) considerable speedups in display can be achieved by not flushing each graphics primitive call to the actual display until necessary. VOGLE automatically delays flushing under in following cases:

- Within a callobj() call.
- Within curves and patches.
- Within Hershey software text.
- When double buffering (the flush is only done within swapbuffers).

There are two user routines that can be used to control flushing.

vsetflush(yesno)

Set global flushing status. If yesno = 0 (.false.) then don't do any flushing (except in swapbuffers(), or vflush()). If yesno = 1 (.true.) then do the flushing as described above.

Fortran:
```
subroutine vsetflush(yesno)
logical yesno
```

C:
```
void
vsetflush(yesno)
        int     yesno;
```

Pascal:
```
procedure VsetFlush(yesno: boolean);
```

vflush() Call the device flush or syncronisation routine. This forces a flush.

Fortran:
```
subroutine vflush
```

C:
```
void
vflush();
```

Pascal:
```
procedure Vflush;
```

Viewport Routines.

viewport(left, right, bottom, top)

Specify which part of the screen to draw in. Left, right, bottom, and top are real values in screen coordinates (-1.0 to 1.0).

Fortran:
```
subroutine viewport(left, right, bottom, top)
real left, right, bottom, top
```

C:
```
viewport(left, right, bottom, top)
        float   left, right, bottom, top;
```

Pascal:
```
procedure ViewPort(left, right, bottom, top: real);
```

pushviewport()

Save current viewport.

Fortran:
```
subroutine pushviewport
```

C:
```
pushviewport()
```

Pascal:
```
procedure PushViewPort;
```

popviewport()

Retrieve last viewport.

Fortran:
```
subroutine popviewport
```

C:
```
popviewport()
```

Pascal:
```
procedure PopViewPort;
```

getviewport(left, right, bottom, top)

Returns the left, right, bottom and top limits of the current viewport in screen coordinates (-1.0 to 1.0).

Fortran:
```
subroutine getviewport(left, right, bottom, top)
real left, right, bottom, top
```

C:
```
getviewport(left, right, bottom, top)
        float    *left, *right, *bottom, *top;
```

Pascal:
```
procedure GetViewPort(var left, right, bottom, top: real)
```

Getting the aspect details

Often the screen is not perfectly square and it would be nice to use the extra space without having to turn clipping off. The following routines are provided to get the values needed to adjust the calls to viewport, etc as needed.

getaspect()

Returns the ratio height over width of the display device.

Fortran:
```
real function getaspect()
```

C:
```
float
getaspect()
```

Pascal:

```
function GetAspect(): real;
```

getfactors(wfact, hfact)

Returns wfact as the width over min(width of device, height of device) and hfact as the height over min(width of device, height of device).

Fortran:

```
subroutine getfactors(w, h)
real w, h
```

C:

```
getfactors(w, h)
        float   *w, *h;
```

Pascal:

```
procedure GetFactors(var w, h: real)
```

getdisplaysize(w, h)

Returns the width and height of the device in pixels in w and h respectively.

Fortran:

```
subroutine getdisplaysize(w, h)
real w, h
```

C:

```
getdisplaysize(w, h)
        float   *w, *h;
```

Pascal:

```
procedure GetDisplaySize(var w, h: real)
```

expandviewport()

When Vogle does viewport calculations, it will normally begin by using the largest square it can fit onto the actual display device. This call says to use the whole device... however you must then take into account any distortion that will occur due to the non square mapping. Thus, a viewport of (-1.0, 1.0, -1.0, 1.0) will map into the whole display device.

Fortran:

```
subroutine expandviewport
```

C:

```
expandviewport()
```

Pascal:

```
procedure ExpandViewport
```

unexpandviewport()

Does the reverse of *expandviewport.* Basically, it returns vogle to using the largest square of the device for it's viewport calculations.

Fortran:

```
subroutine unexpandviewport
```

C:

unexpandviewport()

Pascal:

procedure UnExpandViewport

Attribute Stack Routines.

The attribute stack contains the following details:
current color index
filling/hatching status
hatch pitch and angle
Linestyle and linestyle status
Doublebuffer status
Text rotation
Text size
Text justification
Text fixedwidth status
Text status (hardware or software)
Text font.
If you need to prevent object calls form changing these, use *pushattributes* before the call and *popattributes* after.

pushattributes()

Save the current attributes on the attribute stack.

Fortran:

subroutine pushattributes

C:

pushattributes()

Pascal:

procedure PushAttributes;

popattributes()

Restore the attributes to what they were at the last *pushattribute()*.

Fortran:

subroutine popattributes

C:

popattributes()

Pascal:

procedure PopAttributes;

Projection Routines.

All the projection routines define a new transformation matrix, and consequently the world units. Parallel projections are defined by ortho or ortho2. Perspective projections can be defined by perspective and window.

ortho(left, right, bottom, top, near, far)

Define x (left, right), y (bottom, top), and z (near, far) clipping planes. The near and far clipping planes are actually specified as distances along the line of sight. These distances can also be negative. The actual location of the clipping planes is z = -near_d and z = -far_d.

Fortran:

```
subroutine ortho(left, right, bottom, top, near_d, far_d)
real left, right, bottom, top, near_d, far_d
```

C:

```
ortho(left, right, bottom, top, near_d, far_d)
    float    left, right, bottom, top, near_d, far_d;
```

Pascal:

```
procedure Ortho(left, right, bottom, top, near_d, far_d: real)
```

ortho2(left, right, bottom, top)

Define x (left, right), and y (bottom, top) clipping planes.

Fortran:

```
subroutine ortho2(left, right, bottom, top)
real left, right, bottom, top
```

C:

```
ortho2(left, right, bottom, top)
    float    left, right, bottom, top;
```

Pascal:

```
procedure Ortho2(left, right, bottom, top: real)
```

perspective(fov, aspect, near, far)

Specify a perspective viewing pyramid in world coordinates by giving a field of view, aspect ratio and the distance from the eye of the near and far clipping plane.

The fov, specifies the field of view in the y direction. It is the range of the area that is being viewed. The aspect ratio is the ratio x/y (width/height) which determines the fov in the x direction.

Fortran:

```
subroutine perspective(fov, aspect, near, far)
real fov, aspect, near, far
```

C:

```
perspective(fov, aspect, near, far)
    float    fov, aspect, near, far;
```

Pascal:

```
procedure Perspective(fov, aspect, near, far: real)
```

window(left, right, bot, top, near, far)

Specify a perspective viewing pyramid in world coordinates by
give the rectangle closest to the eye (ie. at the near clipping
plane) and the distances to the near and far clipping planes.

Fortran:

subroutine window(left, right, bot, top, near, far)
real left, right, bot, top, near, far

C:

window(left, right, bot, top, near, far)
float left, right, bot, top, near, far;

Pascal:

procedure Window(left, right, bot, top, near, far: real)

Matrix Stack Routines.

pushmatrix()

Save the current transformation matrix on the matrix stack.

Fortran:

subroutine pushmatrix

C:

pushmatrix()

Pascal:

procedure PushMatrix

popmatrix()

Retrieve the last matrix pushed and make it the current transformation matrix.

Fortran:

subroutine popmatrix

C:

popmatrix()

Pascal:

procedure PopMatrix

Viewpoint Routines.

Viewpoint routines alter the current tranformation matrix.

polarview(dist, azim, inc, twist)

Specify the viewer's position in polar coordinates by giving the distance from the viewpoint to the world origin, the azimuthal angle in the x-y plane, measured from the y-axis, the incidence angle in the y-z plane, measured from the z-axis, and the twist angle about the line of sight.

Fortran:

subroutine polarview(dist, azim, inc, twist)
real dist, azim, inc, twist

C:

polarview(dist, azim, inc, twist)
float dist, azim, inc, twist;

Pascal:

```
procedure PolarView(dist, azim, inc, twist: real)
```

up(x, y, z)

Specify the world up. This can be used to prevent lookat's sometimes annoying habit of turning everything upside down due to the line of sight crossing the appropriate axis.

Fortran:
```
subroutine up(x, y, z)
real x, y, z
```

C:
```
up(x, y, z)
        float   x, y, z;
```

Pascal:
```
procedure Up(x, y, z: real)
```

lookat(vx, vy, vz, px, py, pz, twist)

Specify the viewer's position by giving a viewpoint and a reference point in world coordinates. A twist about the line of sight may also be given.

The viewpoint is at (vx, vy, vz). The reference (or viewed) point is at (px, py, pz). ie the line of site is from v to p. The twist parameter is a righthand rotation about the line of site.

Fortran:
```
subroutine lookat(vx, vy, vz, px, py, pz, twist)
real vx, vy, vz, px, py, pz, twist
```

C:
```
lookat(vx, vy, vz, px, py, pz, twist)
        float   vx, vy, vz, px, py, pz, twist;
```

Pascal:
```
procedure LookAt(vx, vy, vz, px, py, pz, twist: real)
```

Move Routines.

move(x, y, z)

Move current graphics position to (x, y, z). (x, y, z) is a point in world coordinates.

Fortran:
```
subroutine move(x, y, z)
real x, y, z
```

C:
```
move(x, y, z)
        float   x, y, z;
```

Pascal:
```
procedure Move(x, y, z: real)
```

rmove(deltax, deltay, deltaz)

Relative move. deltax, deltay, and deltaz are offsets in world units.

Fortran:

```
subroutine rmove(deltax, deltay, deltaz)
real deltax, deltay, deltaz
```

C:

```
rmove(deltax,deltay)
        float   deltax, deltay, deltaz;
```

Pascal:

```
procedure Rmove(deltax, deltay, deltaz: real)
```

move2(x, y)

Move graphics position to point (x, y). (x, y) is a point in world coordinates.

Fortran:

```
subroutine move2(x, y)
real x, y
```

C:

```
move2(x, y)
        float       x, y;
```

Pascal:

```
procedure Move2(x, y: real)
```

rmove2(deltax, deltay)

Relative move2. deltax and deltay are offsets in world units.

Fortran:

```
subroutine rmove2(deltax, deltay)
real deltax, deltay
```

C:

```
rmove2(deltax, deltay)
        float       deltax, deltay;
```

Pascal:

```
procedure Rmove2(deltax, deltay: real)
```

smove2(x, y)

Move current graphics position in screen coordinates (-1.0 to 1.0).

Fortran:

```
subroutine smove2(x, y)
real x, y
```

C:

```
smove2(x, y)
        float       x, y;
```

Pascal:

```
procedure Smove2(x, y: real)
```

rsmove2(deltax, deltay)

Relative smove2. deltax, and deltay are offsets in screen units (-1.0 to 1.0).

Fortran:

```
subroutine rsmove2(deltax, deltay)
real deltax, deltay
```

C:

```
rsmove2(deltax, deltay)
        float   deltax, deltay;
```

Pascal:

```
procedure Rsmove2(deltax, deltay: real)
```

Linestyles.

Linestyles are specified by giving a nominal length of a single dash and a character string consisting of 1's and 0's (zeros) that specify when to draw a dash and when not to draw a dash. Linestyles will follow curves and "go around" corners. If a linestyle is set or reset, the accumulated information as to where on a curve (or line) a dash is to be draw is also reset.

For example, with a nominal view of -1 to 1, setting the dash length to 0.5, and the linestyle to '11010' would draw a line(or curve) with a 1.0 unit solid part, followed by a 0.5 unit blank part followed by a 0.5 unit solid part followed by a 0.5 unit blank part. The linestyle would then repeat itself.

The dash sizes are affected by the current viewport/transformation scaling factors, meaning that in perspective, the dashes look smaller the farther away they are.

setdash(dashlen)

Set the current dash length (in world units) to be dashlen.

Fortran:

```
subroutine setdash(dashlen)
real dashlen
```

C:

```
setdash(dashlen)
        float   dashlen;
```

Pascal:

```
procedure SetDash(dashlen: real)
```

linestyle(style)

Set the current linestyle to style.

Fortran:

```
subroutine linestyle(style)
character *(*) style
```

C:

```
linesyle(style)
        char    *style;
```

Pascal:

procedure LineStyle(style: string_t)

Drawing Routines.

draw(x, y, z)

Draw from current graphics position to (x, y, z). (x, y, z) is a point in world coordinates.

Fortran:

```
subroutine draw(x, y, z)
real x, y, z
```

C:

```
draw(x, y, z)
        float    x, y, z;
```

Pascal:

```
procedure Draw(x, y, z: real)
```

rdraw(deltax, deltay, deltaz)

Relative draw. deltax, deltay, and deltaz are offsets in world units.

Fortran:

```
subroutine rdraw(deltax, deltay, deltaz)
real deltax, deltay, deltaz
```

C:

```
rdraw(deltax, deltay, deltaz)
        float   deltax, deltay, deltaz;
```

Pascal:

```
procedure Rdraw(deltax, deltay, deltaz: real)
```

draw2(x, y)

Draw from current graphics position to point (x, y). (x, y) is a point in world coordinates.

Fortran:

```
subroutine draw2(x, y)
real x, y
```

C:

```
draw2(x, y)
        float    x, y;
```

Pascal:

```
procedure Draw2(x, y: real)
```

rdraw2(deltax,deltay)

Relative draw2. deltax and deltay are offsets in world units.

Fortran:

```
subroutine rdraw2(deltax, deltay)
real deltax, deltay
```

C:
```
rdraw2(deltax, deltay)
        float   deltax, deltay;
```

Pascal:
```
procedure Rdraw2(deltax, deltay: real)
```

sdraw2(x, y)

Draw in screen coordinates (-1.0 to 1.0).

Fortran:
```
subroutine sdraw2(x, y)
real x, y
```

C:
```
sdraw2(x, y)
        float       x, y;
```

Pascal:
```
procedure Sdraw2(x, y: real)
```

rsdraw2(deltax, deltay)

Relative sdraw2. delatx and deltay are in screen units (-1.0 to 1.0).

Fortran:
```
subroutine rsdraw2(deltax, deltay)
real deltax, deltay
```
C:
```
rsdraw2(deltax, deltay)
        float       deltax, deltay;
```

Pascal:
```
procedure Rsdraw2(deltax, deltay: real)
```

Arcs and Circles.

circleprecision(nsegs)

Set the number of line segments making up a circle. Default is currently 32. The number of segments in an arc or sector is calculated from nsegs according the span of the arc or sector. This replaces the routine arcprecision.

Fortran:
```
subroutine circleprecision(nsegs)
integer   nsegs
```
C:
```
circleprecision(nsegs)
        int         nsegs;
```

Pascal:
```
procedure CirclePrecision(nsegs: integer)
```

arc(x, y, radius, startang, endang)

Draw an arc. x, y, and radius are values in world units.

Fortran:

```
subroutine arc(x, y, radius, startang, endang)
real x, y, radius;
real startang, endang;
```

C:

```
arc(x, y, radius, startang, endang)
        float  x, y, radius;
        float  startang, endang;
```

Pascal:

```
procedure Arc(x, y, radius, startang, endang: real)
```

sector(x, y, radius, startang, endang)

Draw a sector. x, y, and radius are values in world units. Note: sectors are regarded as polygons, so if polyfill or polyhatch has been called with 1, the sectors will be filled or hatched accordingly.

Fortran:

```
subroutine sector(x, y, radius, startang, endang)
real x, y, radius;
real startang, endang;
```

C:

```
sector(x, y, radius, startang, endang)
        float  x, y, radius;
        float  startang, endang;
```

Pascal:

```
procedure Sector(x, y, radius, startang, endang: real)
```

circle(x, y, radius)

Draw a circle. x, y, and radius are values in world units. Note: circles are regarded as polygons, so if polyfill or polyhatch has been called with 1, the circle will be filled or hatched accordingly.

x and y real coordinates in user units.

Fortran:

```
subroutine circle(x, y, radius)
real    x, y, radius
```

C:

```
circle(x, y, radius)
        float   x, y, radius;
```

Pascal:

```
procedure Circle(x, y, radius: real)
```

Curve Routines.

curvebasis(basis)

Define a basis matrix for a curve.

Fortran:

```
subroutine curvebasis(basis)
real basis(4,4)
```

C:

```
curvebasis(basis)
        float   basis[4][4];
```

Pascal:

```
procedure CurveBasis(basis: Matrix44_t)
```

curveprecision(nsegs)

Define the number of line segments used to draw a curve.

Fortran:

```
subroutine curveprecision(nsegs)
integer nsegs
```

C:

```
curveprecision(nsegs)
        int     nsegs;
```

Pascal:

```
procedure CurvePrecision(nsegs: integer)
```

rcurve(geom)

Draw a rational curve.

Fortran:

```
subroutine rcurve(geom)
real geom(4,4)
```

C:

```
rcurve(geom)
        float   geom[4][4];
```

Pascal:

```
procedure Rcurve(geom: Matrix44_t)
```

curve(geom)

Draw a curve.

Fortran:

```
subroutine curve(geom)
real geom(3,4)
```

C:

```
curve(geom)
        float   geom[4][3];
```

Pascal:

```
procedure Curve(geom: Matrix43_t)
```

curven(n, geom)

Draw n - 3 overlapping curve segments. Note: n must be at least 4.

Fortran:

```
subroutine curven(n, geom)
integer n
real geom(3,n)
```

C:

```
curven(n, geom)
        int     n;
        float   geom[][3];
```

Pascal:

```
procedure Curven(n: integer; geom: GeomMat_t)
```

Rectangles and General Polygon Routines.

rect(x1, y1, x2, y2)

Draw a rectangle.

Fortran:

```
subroutine rect(x1, y1, x2, y2)
real x1, y1, x1, y2
```

C:

```
rect(x1, y1, x2, y2)
        float   x1, y1, x2, y2;
```

Pascal:

```
procedure Rect(x1, y1, x2, y2: real)
```

polyfill(onoff)

Set the polygon fill flag. This will always turn off hatching. A non-zero (.true.) turns polyfill on.

Fortran:

```
subroutine polyfill(onoff)
logical onoff
```

C:

```
polyfill(onoff)
        int     onoff;
```

Pascal:

```
procedure PolyFill(onoff: boolean)
```

polyhatch(onoff)

Set the polygon hatch flag. This will always turn off fill. A non-zero (.true.) turns polyhatch on. Note that hatched polygons must initially be defined parrallel to the X-Y plane.

Fortran:

```
subroutine polyhatch(onoff)
logical   onoff
```

C:

```
polyhatch(onoff)
        int     onoff;
```

Pascal:

```
procedure PolyHatch(onoff: boolean)
```

hatchang(angle)

Set the angle of the hatch lines.

Fortran:

```
subroutine hatchang(angle)
real angle
```

C:

```
hatchang(angle)
        float   angle;
```

Pascal:

```
procedure HatchAng(angle: real)
```

hatchpitch(pitch)

Set the distance between hatch lines.

Fortran:

```
subroutine hatchpitch(pitch)
real pitch
```

C:

```
hatchpitch(pitch)
        float   pitch;
```

Pascal:

```
procedure HatchPitch(pitch: real)
```

poly2(n, points)

Construct an (x, y) polygon from an array of points provided by the user.

Fortran:

```
subroutine poly2(n, points)
integer   n
real points(2, n)
```

C:

```
poly2(n,  points)
        int     n;
        float   points[][2];
```

Pascal:

```
procedure Poly2(n: integer; points: Poly2_array_t)
```

poly(n, points)

Construct a polygon from an array of points provided by the user.

Fortran:

```
subroutine poly(n, points)
integer   n
real points(3, n)
```

C:

```
poly(n,  points)
        int     n;
        float   points[][3];
```

Pascal:

```
procedure Poly(n: integer; points: Poly3_array_t)
```

makepoly()

makepoly opens up a polygon which will then be constructed by a series of move-draws and closed by a closepoly.

Fortran:

```
subroutine makepoly
```

C:

```
makepoly()
```

Pascal:

```
procedure MakePoly
```

closepoly()

Terminates a polygon opened by makepoly.

Fortran:

```
subroutine closepoly
```

C:

```
closepoly()
```

Pascal:

```
procedure ClosePoly
```

backface(onoff)

Turns on culling of backfacing polygons. A polygon is backfacing if it's orientation in *screen* coords is clockwise, unless a call to backfacedir is made.

Fortran:

```
subroutine backface(onoff)
integer onoff
```

C:

```
backface(onoff)
        int     onoff;
```

Pascal:

```
procedure BackFace(onoff: boolean)
```

backfacedir(clockwise)

Sets the backfacing direction to clockwise or anticlockwise depending on whether clockwise is 1 or 0.

Fortran:

```
subroutine backfacedir(clockwise)
integer clockwise
```

C:

```
backfacedir(clockwise)
        int     clockwise;
```

Pascal:

```
procedure BackFaceDir(clockwise: boolean)
```

Text Routines.

VOGLE supports hardware and software fonts. The software fonts are based on the character set digitized by Dr Allen V. Hershey while working at the U. S. National Bureau of Standards. Exactly what hardware fonts are supported depends on the device, but it is guaranteed that the names "large" and "small" will result in something readable. For X11 displays the default large and small fonts used by the program can be overridden by placing the following defaults in the ~/.Xdefaults file:

vogle.smallfont: <font name>
vogle.largefont: <font name>

It is noted here that text is always assumed to be drawn parallel to the (x, y) plane, using whatever the current z coordinate is. The following software fonts are supported:

astrology	cursive	cyrillic	futura.l
futura.m	gothic.eng	gothic.ger	gothic.ita
greek	markers	math.low	math.upp
meteorology	music	script	symbolic
times.g	times.i	times.ib	times.r
times.rb	japanese		

A markers font "markers" is also provided for doing markers - you need to have centertext on for this to give sensible results - with the markers starting at 'A' and 'a'.

If the environment variable "VFONTLIB" is set VOGLE looks for the software fonts in the directory given by this value.

font(fontname)

Set the current font

Fortran:
```
subroutine font(fontname)
character*(*) fontname
```
C:
```
font(fontname)
    char    *fontname
```
Pascal:
```
procedure Font(fontname: string_t)
```

numchars()

Return the number of characters in the current font. Applicable only to software fonts.

Fortran:
```
integer function numchars
```
C:
```
int
numchars()
```
Pascal:
```
function NumChars: integer;
```

textsize(width, height)

Set the maximum size of a character in the current font. Width and height are values in world units. This only applies to software text. This must be done after the font being scaled is loaded.

Fortran:
```
subroutine textsize(width, height)
real width, height
```
C:
```
textsize(width, height)
```

float width, height;

Pascal:
procedure TextSize(width, height: real)

textang(ang)

Set the text angle. This angles strings and chars. This routine only affects software text.

Fortran:
subroutine textang(ang)
real ang

C:
textang(ang)
float ang;

Pascal:
procedure TexTang(ang: real)

fixedwidth(onoff)

Turns fixedwidth text on or off. Non-zero (.true.) is on. Causes all text to be printed fixedwidth. This routine only affects software text.

Fortran:
subroutine fixedwidth(onoff)
logical onoff

C:
fixedwidth(onoff)
int onoff;

Pascal:
procedure FixedWidth(onoff: boolean)

centertext(onoff)

Turns centertext text on or off. Non-zero (.true.) is on. This centres strings and chars. This routine only affects software text. All other justification is turned off. The text is centered in both the x and y directions.

Fortran:
subroutine centertext(onoff)
logical onoff

C:
centertext(onoff)
int onoff;

Pascal:
procedure CenterText(onoff: boolean)

textjustify(val)

General (direct) control of text justification. The value of val is made up of the logical OR of the following predefines constants in vogle.h (FOR C only). LEFT, RIGHT, XCENTERED, TOP, BOTTOM, YCENTERED. Centering takes priority, as does RIGHT and TOP justification (if you were silly enough to set it to LEFT|RIGHT for example that is). A value of 0 (zero) (in all languages) resets the textjustification to the default.

Fortran:

```
subroutine textjustify(val)
integer val
```

C:

```
textjustify(val)
        unsigned val;
```

Pascal:

```
procedure CenterText(val: integer)
```

leftjustify()

Left justifies text. The text string will begin at the current position and extend to the notional right. Right justifcation and X centering are turned off.

Fortran:

```
subroutine leftjustify
```

C:

```
leftjustify()
```

Pascal:

```
procedure LeftJustify
```

rightjustify()

Right justifies text. The text string will begin at a point to the notional left of the current position and finish at the current position. Left justification and X centering are turned off.

Fortran:

```
subroutine rightjustify
```

C:

```
rightjustify()
```

Pascal:

```
procedure RightJustify
```

xcentertext()

Centers text in the X direction. The text string will begin at a point to the notional left of the current position and finish at a point to the right of the current position. Left justification and Right justification are turned off.

Fortran:

```
subroutine xcentertext
```

C:

```
xcentertext()
```

Pascal:

```
procedure XcenterText
```

topjustify()

Top justifies text. The text string will be drawn with it's upper edge aligned with the current Y position. Bottom justification and Y centering are turned off.

Fortran:
```
subroutine topjustify
```

C:
```
topjustify()
```

Pascal:
```
procedure TopJustify
```

bottomjustify()

Bottom justifies text. The text string will be drawn with it's lower edge aligned with the current Y position. Top justification and Y centering are turned off.

Fortran:
```
subroutine bottomjustify
```

C:
```
bottomjustify()
```

Pascal:
```
procedure BottomJustify
```

ycentertext()

Centers text in the Y direction. The text string will so that it's center line is aligned with the current y position. Top justification and Bottom justification are turned off.

Fortran:
```
subroutine ycentertext
```
C:
```
ycentertext()
```

Pascal:
```
procedure YcenterText
```

getcharsize(c, width, height)

Get the width and height of a character. At the moment the

getcharsize(c, width, height)

Get the width and height of a character. At the moment the height returned is always that of the difference between the maximum descender and ascender.

Fortran:
```
subroutine getcharsize(c, width, height)
character*1 c
real width, height
```
C:
```
getcharsize(c, width, height)
	char	c;
	float	*width, *height;
```

Pascal:

```
procedure GetCharSize(c: char; var width, height: real)
```

getfontsize(width, height)

Get the maximum width and height of a character in a font.

Fortran:

```
subroutine getfontsize(width, height)
real width, height
```

C:

```
getfontsize(width, height)
        float   *width, *height;
```

Pascal:

```
procedure GetFontSize(var width, height: real)
```

drawchar(c)

Draw the character c. The current graphics position represents the bottom left hand corner of the character space, unless centertext has been enabled, where it represents the "centre" of the character.

Fortran:

```
subroutine drawchar(c)
character c
```

C:

```
drawchar(str)
        char c;
```

Pascal:

```
procedure DrawChar(c: char)
```

drawstr(str)

Draw the text in str at the current position. The current graphics position represents the bottom left hand corner of the character space, unless centertext has been enabled, where it represents the "centre" of the string.

Fortran:

```
subroutine drawstr(str)
character*(*) str
```

C:

```
drawstr(str)
        char *str;
```

Pascal:

```
procedure DrawStr(str: string_t)
```

strlength(str)

Return the length of the string s in world units.

Fortran:

```
real function strlength(str)
character*(*) str
```

C:

```
float
strlength(str)
        char *str;
```

Pascal:

```
function StrLength(str: string_t): real;
```

boxtext(x, y, l, h, s)

Draw the string s so that it fits in the imaginary box defined with bottom left hand corner at (x, y), length l, and hieght h. This only applies to software text.

Fortran:

```
subroutine boxtext(x, y, l, h, s)
real x, y, l, h
character*(*)
```

C:

```
boxtext(x, y, l, h, s)
        float   x, y, l, h;
        char    *s;
```

Pascal:

```
procedure BoxText(x, y, l, h: real; s: string_t)
```

boxfit(l, h, nchars)

Set scale for text so that a string of the biggest characters in the font will fit in a box l by h. l and h are real values in world dimensions. This only applies to software text.

Fortran:

```
subroutine boxfit(l, h, nchars)
real l, h
integer nchars
```

C:

```
boxfit(l, h, nchars)
        float   l, h
        int     nchars
```

Pascal:

```
procedure BoxFit(l, h: real; nchars: integer)
```

Transformations Routines.

All transformations are cumulative, so if you rotate something and then do a translate you are translating relative to the rotated axes. If you need to preserve the current transformation matrix use pushmatrix(), do the drawing, and then call popmatrix() to get back where you were before.

translate(x, y, z)

Set up a translation.

Fortran:

```
subroutine translate(x, y, z)
real x, y, z
```

C:

```
translate(x, y, z)
        float   x, y, z;
```

Pascal:
procedure Translate(x, y, z: real)

scale(x, y, z)
Set up scaling factors in x, y, and z axis.

Fortran:
```
subroutine scale(x, y, z)
real x, y, z
```
C:
```
scale(x, y, z)
        float   x, y, z;
```

Pascal:
```
procedure Scale(x, y, z: real)
```

rotate(angle, axis)
Set up a rotation in axis axis. Where axis is one of 'x', 'y', or 'z'.

Fortran:
```
subroutine rotate(angle, axis)
real angle
character axis
```
C:
```
rotate(angle, axis)
        float   angle;
        char    axis;
```

Pascal:
```
procedure Rotate(angle: real; axis: char)
```

Patch Routines.

patchbasis(tbasis, ubasis)
Define the t and u basis matrices of a patch.

Fortran:
```
subroutine patchbasis(tbasis, ubasis)
real tbasis(4, 4), ubasis(4, 4)
```
C:
```
patchbasis(tbasis, ubasis)
        float   tbasis[4][4], ubasis[4][4];
```

Pascal:
```
procedure PatchBasis(tbasis, ubasis: Matrix44_t)
```

patchprecision(tseg, useg)
Set the minimum number of line segments making up curves in a patch.

Fortran:
```
subroutine patchprecision(tseg, useg)
integer tseg, useg
```
C:
```
patchprecision(tseg, useg)
```

```
int	tseg, useg;
```

Pascal:

```
procedure PatchPrecision(tseg, useg: integer)
```

patchcurves(nt, nu)

Set the number of curves making up a patch.

Fortran:

```
subroutine patchcurves(nt, nu)
integer nt, nu
```

C:

```
patchcurves(nt, nu)
	int	nt, nu;
```

Pascal:

```
procedure PatchCurves(nt, nu: integer)
```

rpatch(gx, gy, gz, gw)

Draws a rational patch in the current basis, according to the geometry matrices gx, gy, gz, and gw.

Fortran:

```
subroutine rpatch(gx, gy, gz, gw)
real  gx(4,4), gy(4,4), gz(4,4), gw(4,4)
```

C:

```
rpatch(gx, gy, gz, gw)
   float  gx[4][4], gy[4][4], gz[4][4], gw[4][4];
```

Pascal:

```
procedure Rpatch(gx, gy, gz, gw: Matrix44_t)
```

patch(gx, gy, gz)

Draws a patch in the current basis, according to the geometry matrices gx, gy, and gz.

Fortran:

```
subroutine patch(gx, gy, gz)
real  gx(4,4), gy(4,4), gz(4,4)
```

C:

```
patch(gx, gy, gz)
	float  gx[4][4], gy[4][4], gz[4][4];
```

Pascal:

```
procedure Patch(gx, gy, gz: Matrix44_t)
```

Point Routines.

point(x, y, z)

Draw a point at x, y, z

Fortran:

```
subroutine point(x, y, z)
real x, y, z
```

C:

```
point(x, y, z)
```

```
        real    x, y, z;
```

Pascal:

```
procedure Point(x, y, z: real)
```

point2(x, y)

Draw a point at x, y.

Fortran:

```
subroutine point2(x, y)
real x, y
```

C:

```
point2(x, y)
        float   x, y;
```

Pascal:

```
procedure Point2(x, y: real)
```

Object Routines.

Objects are graphical entities created by the drawing routines called between *makeobj* and *closeobj.* Objects may be called from within other objects. When an object is created most of the calculations required by the drawing routines called within it are done up to where the calculations involve the current transformation matrix. So if you need to draw the same thing several times on the screen but in different places it is faster to use objects than to call the appropriate drawing routines each time. Objects also have the advantage of being saveable to a file, from where they can be reloaded for later reuse. Routines which draw or move in screen coordinates, or change device, cannot be included in objects.

makeobj(n)

Commence the object number n.

Fortran:

```
subroutine makeobj(n)
integer n
```

C:

```
makeobj(n)
        int     n;
```

Pascal:

```
procedure MakeObj(n: integer)
```

closeobj()

Close the current object.

Fortran:

```
subroutine closeobj()
```

C:

```
closeobj()
```

Pascal:

```
procedure CloseObj
```

genobj()

Returns a unique object identifier.

Fortran:
```
integer function genobj()
```
C:
```
int
genobj()
```
Pascal:
```
function GenObj: integer
```

getopenobj()

Return the number of the current object.

Fortran:
```
integer function getopenobj()
```
C:
```
int
getopenobj()
```
Pascal:
```
function GetOpenObj: integer
```

callobj(n)

Draw object number n.

Fortran:
```
subroutine callobj(n)
integer n
```
C:
```
callobj(n)
    int     n;
```
Pascal:
```
procedure CallObj(n: integer)
```

isobj(n)

Returns non-zero if there is an object of number n.

Fortran:
```
logical function isobj(n)
integer n
```
C:
```
int
isobj(n)
    int     n;
```
Pascal:
```
function IsObj(n: integer): boolean;
```

delobj(n)

Delete the object number n.

Fortran:

```
subroutine delobj(n)
integer n
```

C:

```
delobj(n)
        Object    n;
```

Pascal:

```
procedure DelObj(n: integer);
```

loadobj(n, filename)

Load the object in the file filename a object number n.

Fortran:

```
subroutine loadobj(n, filename)
integer n
character*(*) filename
```

C:

```
loadobj(n, filename)
        int     n;
        char    *filename;
```

Pascal:

```
procedure LoadObj(n: integer; filename: string_t)
```

saveobj(n, filename)

Save the object number n into the file filename. This call does not save objects called inside object n.

Fortran:

```
saveobj(n, filename)
integer   n
character*(*) filename
```

C:

```
saveobj(n, filename)
        int         n;
        char        *filename;
```

Pascal:

```
procedure SaveObj(n: integer; filename: string_t)
```

Double Buffering.

Where possible VOGLE allows for front and back buffers to enable things like animation and smooth updating of the screen. The routine backbuffer is used to initialise double buffering.

backbuffer()

Make VOGLE draw in the backbuffer. Returns -1 if the device is not up to it.

Fortran:

integer function backbuffer

C:

backbuffer()

Pascal:

function BackBuffer:integer

frontbuffer()

Make VOGLE draw in the front buffer. This will always work.

Fortran:

subroutine frontbuffer

C:

frontbuffer()

Pascal:

procedure FrontBuffer

swapbuffers()

Swap the front and back buffers.

Fortran:

subroutine swapbuffers

C:

swapbuffers()

Pascal:

procedure SwapBuffers

Position Routines.

getgp(x, y, z)

Gets the current graphics position in world coords.

Fortran:

subroutine getgp(x, y, z)
real x, y, z

C:

getgp(x, y, z)
float *x, *y, *z;

Pascal:

procedure GetGp(var x, y, z: real)

getgpt(x, y, z, w)

Gets the current *transformed* graphics position in world coords.

Fortran:

subroutine getgpt(x, y, z, w)
real x, y, z, w

C:

```
getgpt(x, y, z, w)
        float *x, *y, *z, *w;
```

Pascal:

```
procedure GetGpT(var x, y, z, w: real)
```

getgp2(x, y)

Gets the current graphics position in world coords.

Fortran:

```
subroutine getgp2(x, y)
real x, y
```

C:

```
getgp2(x, y)
        float *x, *y;
```

Pascal:

```
procedure GetGp2(var x, y: real)
```

sgetgp2(x, y)

Gets the current screen graphics position in screen coords (-1 to 1)

Fortran:

```
subroutine sgetgp2(x, y)
real x, y
```

C:

```
sgetgp2(x, y)
        float *x, *y;
```

Pascal:

```
procedure SgetGp2(var x, y: real)
```

BUGS

We had to make up the font names based on some books of type faces.

Polygon hatching will give unexpected results unless the polygon is initially defined in the X-Y plane.

Double buffering isn't supported on all devices.

We don't recommend the use of the smove/sdraw routines.

The yobbarays may be turned on or they may be turned off.