To be published in the
July/August issue of *The X Journal.*

OpenGL™ and X, Part 3: Integrating OpenGL with Motif

Mark J. Kilgard *
Silicon Graphics Inc.
Revision : 1.18

April 15, 1994

Abstract

The OpenGL™ graphics system can be integrated with the industry-standard OSF/Motif user interface. This article discusses how to use OpenGL within a Motif application program. There are two approaches to using OpenGL with Motif. One is to render into a standard Motif drawing area widget, but this requires each application window to use a single visual for its window hierarchy. A better approach is to use the special OpenGL drawing area widget allowing windows used for OpenGL rendering to pick freely an appropriate visual without affecting the visual choice for other widgets. An example program demonstrates both approaches. The X Toolkit's work procedure mechanism animates the example's 3D paper airplanes. Handling OpenGL errors is also explained.

1 Introduction

OSF/Motif is the X Window System's industry-standard programming interface for user interface construction. Motif programmers writing 3D applications will want to understand how to integrate Motif with the OpenGL™ graphics system. This article, the last in a three-part series about OpenGL, describes how to write an OpenGL program within the user interface framework provided by Motif and the X Toolkit.

Most 3D applications end up using 3D graphics primarily in one or more "viewing" windows. For the most part, the graphical user interface aspects of such programs use standard 2D user interface objects like pulldown menus, sliders, and dialog boxes. Creating and managing such common user interface objects is what Motif does well. The "viewing" windows used for 3D are where OpenGL rendering happens. These windows for OpenGL rendering can be constructed with standard Motif drawing area widgets or OpenGL-specific drawing area widgets. Bind an OpenGL rendering context to the window of a drawing area widget and you are ready for 3D rendering.

Programming OpenGL with Motif has numerous advantages over using "Xlib only" as described in the first two articles in this series [2, 3]. First and most important, Motif provides a well-documented, standard widget set that gives your application a consistent look and feel. Second, Motif and the X Toolkit take care of routine but complicated issues such as *cut and paste* and window manager conventions. Third, the X Toolkit's work procedure and timeout mechanisms make it easy to animate a 3D window without blocking out user interaction with your application.

This article assumes you have some experience programming with Motif and you have a basic understanding of how OpenGL integrates with X.

Section 2 describes how to use OpenGL rendering with either a standard Motif drawing area widget or an OpenGL-specific drawing area widget. Section 3 discusses using X Toolkit mechanisms for seamless animation. Section 4 provides advice on how to debug OpenGL programs by catching OpenGL errors. Throughout the discussion, a Motif-based OpenGL program named `paperplane` is used as an example. The complete source code for `paperplane` is found in the appendix. The program animates the 3D flight paths of virtual paper airplanes. The user can interact with the program via Motif controls. The program can be compiled to use either a standard Motif drawing area widget or an OpenGL-specific drawing area widget. Figure 1 shows `paperplane` running.

*Mark graduated with B.A. in Computer Science from Rice University and is a Member of the Technical Staff at Silicon Graphics. He can be reached by electronic mail addressed to `mjk@sgi.com`

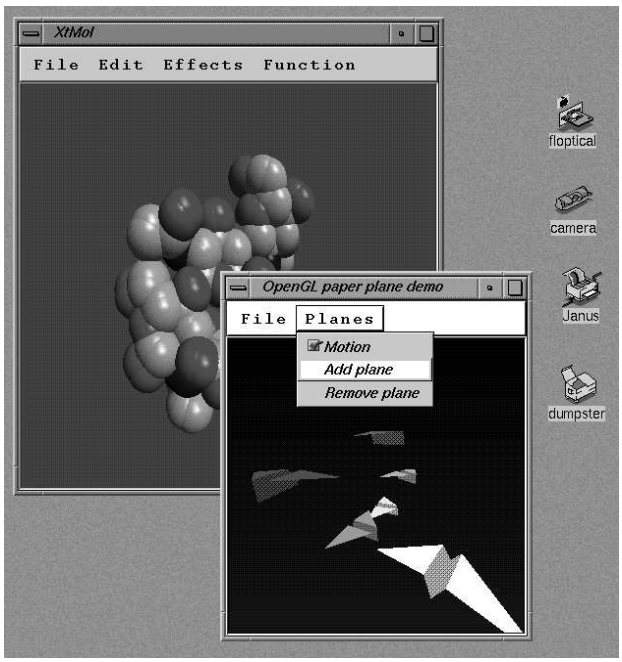

Figure 1: Screen snapshot of `paperplane` with another OpenGL Motif program for molecular modeling.

2 OpenGL with Widgets

Your application's 3D viewing area can be encapsulated by an X Toolkit widget. There are two approaches to rendering OpenGL into a widget. You can render OpenGL into a standard Motif drawing area widget, or you can use a special OpenGL drawing area widget.

The Motif drawing area widget would seem a natural widget for OpenGL rendering. Unfortunately, the X Toolkit's design (upon which Motif relies) allows programmers to specify a widget's visual only if its class is derived from the shell widget class. Shell widgets are often called "top level" widgets because they are designed to communicate with the window manager and act as containers for other widgets. Non-shell widgets inherit the depth and visual of their parent widget. The Motif drawing area widget class (like most widget classes) is not derived from the shell widget class. It is impossible (without resorting to programming widget internals) to set the visual of a standard non-shell Motif widget differently than its ancestor shell widget.

But in OpenGL, the X notion of a visual has expanded importance for determining the OpenGL frame buffer capabilities of an X window. In many cases, an application's 3D viewing area is likely to demand a deeper, more capable visual than the default visual which Motif normally uses.

There are two options:

1. Use the standard Motif drawing area widget for your OpenGL rendering area and make sure that the top level shell widget is created with the desired visual for OpenGL's use.

2. Use an OpenGL drawing area widget that is specially programmed to overcome the limitation on setting the visual and depth of a non-shell widget.

Either approach works.

The `paperplane` example in the appendix is written to support either scheme depending on how the code is compiled. By default, the code compiles to use the OpenGL-specific widget. If the `noGLwidget` C preprocessor symbol is defined, the standard Motif drawing area widget will be used, forcing the use of a single visual throughout the example's widget hierarchy. The code differences between the two schemes in the `paperplane` example constitute seven changed lines of code.

The preferable approach is to use the OpenGL-specific widget, since you can run most of the application's user interface in the default visual and use a deeper, more capable visual only for 3D viewing areas. Limiting the use of deeper visuals can save memory and increase rendering speed for the user interface windows. If you use a 24-bit visual for your 3D viewing area and use the same visual for your entire application, that means that the image memory for pixmaps used by non-OpenGL windows is four times what it would be for an 8-bit visual.[1] Some X rendering operations might also be slower for 24-bit windows compared with 8-bit windows.

There can be advantages to running your entire application in a single visual. Some workstations with limited colormap resources might not be capable of showing multiple visuals without colormap flashing. Such machines which support OpenGL should be rare. Even if running in a single visual is appropriate, nothing precludes doing so using an OpenGL-specific widget.

2.1 The OpenGL-specific Widget

There are two OpenGL-specific drawing area widget classes. One is derived from the Motif primitive widget class (*not* the Motif drawing area widget class). The other is derived from the X Toolkit core widget class. Both have the same basic functionality; the main difference is that the Motif-based widget class gains capabilities of the Motif primitive widget class. If you use Motif, you should use the Motif OpenGL widget. If you use a non-Motif widget set, you can use the second widget for identical functionality.

The Motif OpenGL widget class is named `glwMDrawingAreaWidgetClass`; the non-Motif OpenGL widget class is named `glwDrawingAreaWidgetClass` (the difference is the lack of an `M` in the non-Motif case). Since

[1] Even though a 24-bit pixel requires only three bytes of storage, efficient manipulation of the pixels demands each pixel is stored in an even 4 bytes.

the Motif OpenGL widget is subclassed from the Motif primitive widget class, the Motif OpenGL widget inherits the capabilities of the primitive class like a help callback and keyboard traversal support (keyboard traversal is disabled by default for the Motif OpenGL widget). The `paperplane` example uses the Motif widget by default but the non-Motif widget can be used by defining the `noMotifGLwidget` C preprocessor symbol when compiling `paperplane.c`. The difference is two changed lines of code with no functional difference in the program.

When you create either type of widget, you need to specify the visual to use by supplying the widget's `GLwNvisualInfo` resource. The attribute is of type `XVisualInfo*` making it easy to find an appropriate visual using `glXChooseVisual` which returns a `XVisualInfo*` for a visual with the capabilities you request.

Although this practice is not recommended, the widgets also allow you to specify the OpenGL capabilities you desire for the widget directly using widget resources. Because the X Toolkit widget creation process is not expected to fail, there is no way for a widget creation routine to indicate failure. If a visual that matches the desired OpenGL capabilities cannot be found, the widget code prints an error and exits without giving the program a chance to handle the failure. If you request a specific `XVisualInfo*` that has already been determined to be acceptable using `glXChooseVisual` or calls to `glXGetConfig`, you will not have this problem. As a rule, always specify the visual using the `GLwNvisualInfo` resource.

The OpenGL widgets also do extra work that might go unnoticed. Because the OpenGL widget uses a different visual, the widget's creation code creates a colormap matching the visual. It also posts an ICCCM `WM_COLORMAP_WINDOWS` top level window property to let the window manager know that the program uses multiple colormaps.

More information about the OpenGL widgets can be found in the Silicon Graphics *OpenGL Porting Guide* [4] and the widgets' man pages.

2.2 The Motif Drawing Area Widget

Using the standard Motif drawing area widget with OpenGL has some extra caveats. The main caveat is that you must create the top level widget with the correct visual for the program's OpenGL rendering.

When you start a widget program, there is generally a call to `XtAppInitialize` to establish the connection to the X server and create the top level widget. Both steps are done in the same routine. So how can we call `glXChooseVisual` to know what visual the top level widget should use until we have established a connection to the X server?

It would appear that it is impossible to create the top level widget with an appropriate visual for OpenGL.

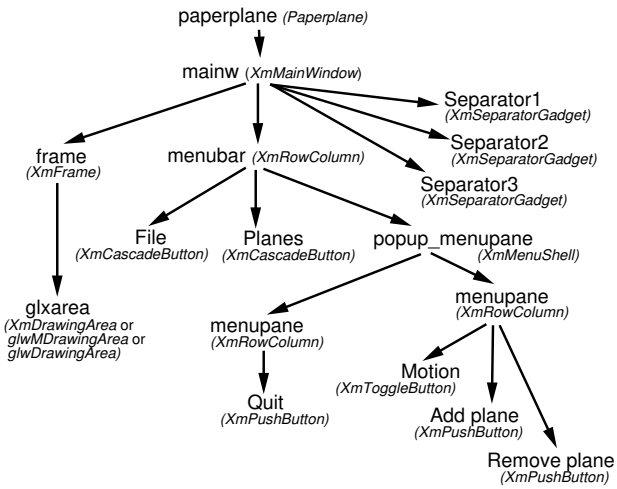

Figure 2: Diagram of the widget hierarchy for `paperplane`. The `glxarea XmDrawingArea` widget is the only widget rendered using OpenGL.

`XtAppInitialize` connects to the X server and creates the top level widget, but it does not *realize* the top level widget. The X window for the top level widget is not created until `XtRealizeWidget` is called. This allows `XtSetValues` to be used after the top level widget's creation (and before its realization) to specify the widget's visual. The `paperplane` sample code in the non-OpenGL widget case demonstrates this.

A second caveat is due to the X Toolkit's inconsistent inheritance of the visual, depth, and colormap widget resources. The default visual of a widget's window is copied from its *parent window*'s visual. But the default colormap and depth of a widget are copied from the widget's *parent widget*.[2]

This means that if you create a widget derived from the shell widget and the widget's parent uses a non-default depth or colormap for a non-default visual, you will need to specify the same visual as the new widget's parent widget. If you do not, a `BadMatch` X protocol error will result. For this reason the `paperplane` example's `XmCreatePulldownMenu` calls specify the visual of the created widget's parent widget in the Motif drawing area version of `paperplane`.

Realize that it is not possible to bind an OpenGL rendering context to a widget's window until the widget has been realized. Until the widget is realized, the widget's window does not yet exist. Notice `paperplane` does not call `glXMakeCurrent` until after `XtRealizeWidget` has been called.

To see how the 3D viewing area widget fits into the `paperplane` widget hierarchy example, Figure 2 shows the complete hierarchy including widget class names.

[2]If the widget has no parent, the depth and colormap are determined by the default depth and colormap of the screen.

These caveats are not unique to OpenGL. The problem comes from using non-default visuals with the X Toolkit. PEXlib 5.1 programs have a similar need for non-default visuals and require the same jumping through hoops[1]. Fortunately, if you use the OpenGL drawing area widgets, you can avoid the caveats of using the standard Motif drawing area.

2.3 Drawing Area Callbacks

Applications using the Motif drawing area widget or the OpenGL drawing area widgets for their 3D rendering will want to register routines to handle expose, resize, and input callbacks using `XtAddCallback`. In `paperplane.c`, the `draw`, `resize`, and `input` routines handle these callbacks.

`paperplane`'s drawing area adjusts OpenGL's viewport by calling `glViewport`. Note how the `made_current` variable is used to protect against calling `glViewport` before we have done the `glXMakeCurrent` to bind to the drawing area window. In the X Toolkit, the resize callback can be called before the `XtRealizeWidget` routine returns. Since the program does not call `glXMakeCurrent` until after the program returns from `XtRealizeWidget`, the OpenGL rendering context would not be bound. Calling an OpenGL routine before a context is bound has no effect but generates an ugly warning message.[3] An example of when the resize callback can be called before `XtRealizeWidget` returns is when a `-geometry` command line option is specified.

Note that `glXMakeCurrent` is defined to set a context's viewport to the size of the first window it is bound to. (This happens only on the context's first bind.) This is why `paperplane.c` makes no initial call to `glViewport`; `glXMakeCurrent` sets the viewport implicitly.

The `paperplane` example uses a single window for OpenGL rendering. For this reason, `glXMakeCurrent` is called only once to bind the OpenGL context to the window. In a program with multiple OpenGL windows, each expose and resize callback should make sure that `glXMakeCurrent` is called so that OpenGL rendering goes to the correct window.

The `draw` callback routine issues the OpenGL commands to draw the scene. If the window is double buffered, `glXSwapBuffers` swaps the window's buffers. If the context is not direct, `glFinish` is called to avoid the latency from queuing more than one frame at a time; interactivity would suffer if we allowed more than one frame to be queued. Direct rendering involves direct manipulation of the hardware so it generally has less latency than a potentially networked indirect OpenGL context.

Note that you can render OpenGL into *any* widget (as long as it is created with an OpenGL capable visual). There is nothing special about the Motif or OpenGL-specific drawing area widgets, though drawing area widgets tend to be the most appropriate widget type for a 3D viewing area.

[3] The exact behavior is undefined by the OpenGL specification.

2.4 Handling Input

The `input` routine handles X events for the drawing area. Input events require no special handling for OpenGL. But remember that the coordinate systems for X and OpenGL are distinct, so pointer locations need to be mapped into OpenGL's coordinate space. OpenGL generally assumes that the origin is in the lower left-hand corner, while X always assumes an origin at the upper left-hand corner.

3 Animation Via Work Procedures

The X Toolkit's work procedure facility makes it easy to integrate continuous OpenGL animation with Motif user interface operation. Work procedures are application supplied routines that execute while the application is idle waiting for events. Work procedures should be used to do small amounts of work; if too much time is spent in a work procedure, X events will not be processed and program interactivity will suffer.

Rendering a single frame of OpenGL animation is a good use for work procedures. `XtAppAddWorkProc` and `XtRemoveWorkProc` are used to add and remove work procedures. `XtAppAddWorkProc` is passed a function pointer for the routine to be called as a work procedure. The function to be called returns a `Boolean`. If the function returns `True`, the work procedure should be removed automatically; returning `False` indicates the work procedure should remain active. `XtAppAddWorkProc` returns an ID of type `WorkProcId` which can later be given to `XtRemoveWorkProc` to remove the work procedure.

The `paperplane` example uses a work procedure to manage the update of its 3D scene. In response to changing the state of the "Motion" toggle button on the "Planes" pulldown menu, the `toggle` callback routine will add and remove the `animate` work procedure.

The `animate` routine calls `tick` which advances the position of each active plane; `animate` then calls `draw` to redraw the scene with the new plane locations. Finally, `animate` returns `False` to leave the work procedure installed so that the animation will continue.

Because `paperplane` uses a work procedure, animation of the scene does not interfere with window resizing and user input. The X Toolkit manages both the animation and events from the X server.

3.1 Handling Iconification

When the `paperplane` window is open, we want the `animate` work procedure to update the 3D scene continuously. If the user iconifies the window, it would be wasteful to continue animating a no longer visible scene. To avoid wasting resources rendering to an unmapped window, `paperplane` installs an event handler called `map_state_changed` for the top-level widget to notice `UnmapNotify` and `MapNotify` events. The handler makes sure the work procedure is removed or added to reflect the map state of the window.

3.2 Timeouts

X Toolkit timeouts are similar to work procedures, but instead of being activated whenever event dispatching is idle, they are called when a given period of time has expired. The `XtAppAddTimeout` and `XtRemoveTimeOut` routines can be used to add and remove X Toolkit timeouts.

OpenGL programmers may find timeouts useful to maintain animation at rates slower than "as fast as OpenGL will render." Timeouts can be used to give animation a sustained frame rate. Timeouts can also be used to redraw a scene with higher detail when the user has stopped interacting with the program. For example, a 3D modeling program might redraw its model with lighting enabled and finer tessellation after the program has been idle for two seconds. Timeouts can also be used to trigger simple real-time state changes useful for visual simulation.

4 Debugging Tips

As well as demonstrating the use of widgets with OpenGL, `paperplane` also demonstrates detection of OpenGL errors for debugging purposes. Some debugging code has been added to the bottom of `paperplane`'s `draw` function to test for any OpenGL errors. A correct OpenGL program should not generate any OpenGL errors, but while debugging it is helpful to check explicitly for errors. A good time to check for errors is at the end of each frame. Errors in OpenGL are not reported unless you explicitly check for them, unlike X protocol errors which are always reported to the client.

OpenGL errors are recorded by setting "sticky" flags. Once an error flag is set, it will not be cleared until `glGetError` is used to query the error. An OpenGL implementation may have several error flags internally that can be set (since OpenGL errors might occur in different stages of the OpenGL rendering pipeline). When you look for errors, you should call `glGetError` repeatedly until it returns `GL_NO_ERROR` indicating that all of the error flags have been cleared.

The OpenGL error model is suited for high performance rendering, since error reporting does not slow down the error-free case. Because OpenGL errors should not be generated by bug-free code, you probably want to remove error querying from your final program since querying errors will slow down your rendering speed.

When an OpenGL error is generated, the command which generated the error is not recorded, so you may need to add more error queries into your code to isolate the source of the error.

The `gluErrorString` routine in the OpenGL Utility library (GLU) converts an OpenGL error number into a human readable string and helps you output a reasonable error message.

5 Conclusion

OpenGL and Motif are a powerful combination. Using both APIs allow X applications programmers to get the most out of both Motif and OpenGL.

Still another way to integrate OpenGL rendering with widgets is the Open Inventor object-oriented 3D graphics toolkit which renders using OpenGL and integrates with X Toolkit widgets. Open Inventor allows you to specify 3D scenes in an object-oriented fashion instead of low-level OpenGL rendering primitives. If you are interested in object-oriented 3D, check out the recently published *Inventor Mentor* [5].

The source code presented in this series is available by anonymous ftp to `sgigate.sgi.com` in the `pub/opengl/xjournal` directory.

Acknowledgments

Writing these three articles on OpenGL required the assistance from numerous engineers and managers at Silicon Graphics. In particular I would like to thank Kurt Akeley, David Blythe, Simon Hui, Phil Karlton, Mark Segal, Kevin Smith, Joel Tesler, Tom Weinstein, Mason Woo, and David Yu.

A paperplane.c

```
/*
 * paperplane can be compiled to use a "single visual" for the entire window
 * hierarchy and render OpenGL into a standard Motif drawing area widget:
 *
 *  cc -o sv_paperplane paperplane.c -DnoGLwidget -lGL -lXm -lXt -lX11 -lm
 *
 * Or paperplane can be compiled to use the default visual for most of
 * the window hierarchy but render OpenGL into a special "OpenGL widget":
 *
 *  cc -o glw_paperplane paperplane.c -lGLw -lGL -lXm -lXt -lX11 -lm
 */
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <math.h>
#include <Xm/MainW.h>
#include <Xm/RowColumn.h>
#include <Xm/PushB.h>
#include <Xm/ToggleB.h>
#include <Xm/CascadeB.h>
#include <Xm/Frame.h>
#ifdef noGLwidget
#include <Xm/DrawingA.h>        /* Motif drawing area widget */
#else
#ifdef noMotifGLwidget
#include <GL/GLwDrawA.h>        /* pure Xt OpenGL drawing area widget */
#else
#include <GL/GLwMDrawA.h>       /* Motif OpenGL drawing area widget */
#endif
#endif
#include <X11/keysym.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glx.h>

static int dblBuf[] = {
    GLX_DOUBLEBUFFER, GLX_RGBA, GLX_DEPTH_SIZE, 16,
    GLX_RED_SIZE, 1, GLX_GREEN_SIZE, 1, GLX_BLUE_SIZE, 1,
    None
};
static int *snglBuf = &dblBuf[1];
static String  fallbackResources[] = {
#ifdef IRIX_5_2_or_higher
    "*sgiMode: true",           /* try to enable IRIX 5.2+ look & feel */
    "*useSchemes: all",         /* and SGI schemes */
#endif
    "*title: OpenGL paper plane demo",
    "*glxarea*width: 300", "*glxarea*height: 300", NULL
};
Display     *dpy;
GLboolean    doubleBuffer = GL_TRUE, moving = GL_FALSE, made_current = GL_FALSE;
XtAppContext app;
XtWorkProcId workId = 0;
Widget       toplevel, mainw, menubar, menupane, btn, cascade, frame, glxarea;
GLXContext   cx;
XVisualInfo *vi;
#ifdef noGLwidget
Colormap     cmap;
```

```
#endif
Arg          menuPaneArgs[1], args[1];

#define MAX_PLANES 15

struct {
    float           speed;      /* zero speed means not flying */
    GLfloat         red, green, blue;
    float           theta;
    float           x, y, z, angle;
} planes[MAX_PLANES];

#define v3f glVertex3f /* v3f was the short IRIS GL name for glVertex3f */

void draw(Widget w)
{
    GLfloat         red, green, blue;
    int             i;

    glClear(GL_DEPTH_BUFFER_BIT);
    /* paint black to blue smooth shaded polygon for background */
    glDisable(GL_DEPTH_TEST);
    glShadeModel(GL_SMOOTH);
    glBegin(GL_POLYGON);
        glColor3f(0.0, 0.0, 0.0);
        v3f(-20, 20, -19); v3f(20, 20, -19);
        glColor3f(0.0, 0.0, 1.0);
        v3f(20, -20, -19); v3f(-20, -20, -19);
    glEnd();
    /* paint planes */
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_FLAT);
    for (i = 0; i < MAX_PLANES; i++)
        if (planes[i].speed != 0.0) {
            glPushMatrix();
                glTranslatef(planes[i].x, planes[i].y, planes[i].z);
                glRotatef(290.0, 1.0, 0.0, 0.0);
                glRotatef(planes[i].angle, 0.0, 0.0, 1.0);
                glScalef(1.0 / 3.0, 1.0 / 4.0, 1.0 / 4.0);
                glTranslatef(0.0, -4.0, -1.5);
                glBegin(GL_TRIANGLE_STRIP);
                    /* left wing */
                    v3f(-7.0, 0.0, 2.0); v3f(-1.0, 0.0, 3.0);
                    glColor3f(red = planes[i].red, green = planes[i].green,
                              blue = planes[i].blue);
                    v3f(-1.0, 7.0, 3.0);
                    /* left side */
                    glColor3f(0.6 * red, 0.6 * green, 0.6 * blue);
                    v3f(0.0, 0.0, 0.0); v3f(0.0, 8.0, 0.0);
                    /* right side */
                    v3f(1.0, 0.0, 3.0); v3f(1.0, 7.0, 3.0);
                    /* final tip of right wing */
                    glColor3f(red, green, blue);
                    v3f(7.0, 0.0, 2.0);
                glEnd();
            glPopMatrix();
        }
    if (doubleBuffer) glXSwapBuffers(dpy, XtWindow(w));
    if(!glXIsDirect(dpy, cx))
        glFinish(); /* avoid indirect rendering latency from queuing */
```

```
113 #ifdef DEBUG
114     { /* for help debugging, report any OpenGL errors that occur per frame */
115         GLenum error;
116         while((error = glGetError()) != GL_NO_ERROR)
117             fprintf(stderr, "GL error: %s\n", gluErrorString(error));
118     }
119 #endif
120 }

121 void tick_per_plane(int i)
122 {
123     float theta = planes[i].theta += planes[i].speed;
124     planes[i].z = -9 + 4 * cos(theta);
125     planes[i].x = 4 * sin(2 * theta);
126     planes[i].y = sin(theta / 3.4) * 3;
127     planes[i].angle = ((atan(2.0) + M_PI_2) * sin(theta) - M_PI_2) * 180 / M_PI;
128     if (planes[i].speed < 0.0) planes[i].angle += 180;
129 }

130 void add_plane(void)
131 {
132     int i;

133     for (i = 0; i < MAX_PLANES; i++)
134         if (planes[i].speed == 0) {

135 #define SET_COLOR(r,g,b) \
136         planes[i].red=r; planes[i].green=g; planes[i].blue=b; break;

137             switch (random() % 6) {
138             case 0: SET_COLOR(1.0, 0.0, 0.0); /* red */
139             case 1: SET_COLOR(1.0, 1.0, 1.0); /* white */
140             case 2: SET_COLOR(0.0, 1.0, 0.0); /* green */
141             case 3: SET_COLOR(1.0, 0.0, 1.0); /* magenta */
142             case 4: SET_COLOR(1.0, 1.0, 0.0); /* yellow */
143             case 5: SET_COLOR(0.0, 1.0, 1.0); /* cyan */
144             }
145             planes[i].speed = (random() % 20) * 0.001 + 0.02;
146             if (random() & 0x1) planes[i].speed *= -1;
147             planes[i].theta = ((float) (random() % 257)) * 0.1111;
148             tick_per_plane(i);
149             if (!moving) draw(glxarea);
150             return;
151         }
152     XBell(dpy, 100); /* can't add any more planes */
153 }

154 void remove_plane(void)
155 {
156     int             i;

157     for (i = MAX_PLANES - 1; i >= 0; i--)
158         if (planes[i].speed != 0) {
159             planes[i].speed = 0;
160             if (!moving) draw(glxarea);
161             return;
162         }
163     XBell(dpy, 100); /* no more planes to remove */
164 }
```

```
165 void resize(Widget w, XtPointer data, XtPointer callData)
166 {
167     Dimension       width, height;

168     if(made_current) {
169         XtVaGetValues(w, XmNwidth, &width, XmNheight, &height, NULL);
170         glViewport(0, 0, (GLint) width, (GLint) height);
171     }
172 }

173 void tick(void)
174 {
175     int i;

176     for (i = 0; i < MAX_PLANES; i++)
177         if (planes[i].speed != 0.0) tick_per_plane(i);
178 }

179 Boolean animate(XtPointer data)
180 {
181     tick();
182     draw(glxarea);
183     return False;               /* leave work proc active */
184 }

185 void toggle(void)
186 {
187     moving = !moving; /* toggle */
188     if (moving)
189         workId = XtAppAddWorkProc(app, animate, NULL);
190     else
191         XtRemoveWorkProc(workId);
192 }

193 void quit(Widget w, XtPointer data, XtPointer callData)
194 {
195     exit(0);
196 }

197 void input(Widget w, XtPointer data, XtPointer callData)
198 {
199     XmDrawingAreaCallbackStruct *cd = (XmDrawingAreaCallbackStruct *) callData;
200     char            buf[1];
201     KeySym          keysym;
202     int             rc;

203     if(cd->event->type == KeyPress)
204         if(XLookupString((XKeyEvent *) cd->event, buf, 1, &keysym, NULL) == 1)
205             switch (keysym) {
206             case XK_space:
207                 if (!moving) { /* advance one frame if not in motion */
208                     tick();
209                     draw(w);
210                 }
211                 break;
212             case XK_Escape:
213                 exit(0);
214             }
215 }
```

```
216 void map_state_changed(Widget w, XtPointer data, XEvent * event, Boolean * cont)
217 {
218     switch (event->type) {
219     case MapNotify:
220         if (moving && workId != 0) workId = XtAppAddWorkProc(app, animate, NULL);
221         break;
222     case UnmapNotify:
223         if (moving) XtRemoveWorkProc(workId);
224         break;
225     }
226 }

227 main(int argc, char *argv[])
228 {
229     toplevel = XtAppInitialize(&app, "Paperplane", NULL, 0, &argc, argv,
230                                fallbackResources, NULL, 0);
231     dpy = XtDisplay(toplevel);
232     /* find an OpenGL-capable RGB visual with depth buffer */
233     vi = glXChooseVisual(dpy, DefaultScreen(dpy), dblBuf);
234     if (vi == NULL) {
235         vi = glXChooseVisual(dpy, DefaultScreen(dpy), snglBuf);
236         if (vi == NULL)
237             XtAppError(app, "no RGB visual with depth buffer");
238         doubleBuffer = GL_FALSE;
239     }
240     /* create an OpenGL rendering context */
241     cx = glXCreateContext(dpy, vi, /* no display list sharing */ None,
242         /* favor direct */ GL_TRUE);
243     if (cx == NULL)
244         XtAppError(app, "could not create rendering context");
245     /* create an X colormap since probably not using default visual */
246 #ifdef noGLwidget
247     cmap = XCreateColormap(dpy, RootWindow(dpy, vi->screen),
248         vi->visual, AllocNone);
249     /*
250      * Establish the visual, depth, and colormap of the toplevel
251      * widget _before_ the widget is realized.
252      */
253     XtVaSetValues(toplevel, XtNvisual, vi->visual, XtNdepth, vi->depth,
254                   XtNcolormap, cmap, NULL);
255 #endif
256     XtAddEventHandler(toplevel, StructureNotifyMask, False,
257                       map_state_changed, NULL);
258     mainw = XmCreateMainWindow(toplevel, "mainw", NULL, 0);
259     XtManageChild(mainw);
260     /* create menu bar */
261     menubar = XmCreateMenuBar(mainw, "menubar", NULL, 0);
262     XtManageChild(menubar);
263 #ifdef noGLwidget
264     /* Hack around Xt's unfortunate default visual inheritance. */
265     XtSetArg(menuPaneArgs[0], XmNvisual, vi->visual);
266     menupane = XmCreatePulldownMenu(menubar, "menupane", menuPaneArgs, 1);
267 #else
268     menupane = XmCreatePulldownMenu(menubar, "menupane", NULL, 0);
269 #endif
270     btn = XmCreatePushButton(menupane, "Quit", NULL, 0);
271     XtAddCallback(btn, XmNactivateCallback, quit, NULL);
272     XtManageChild(btn);
273     XtSetArg(args[0], XmNsubMenuId, menupane);
274     cascade = XmCreateCascadeButton(menubar, "File", args, 1);
```

```
275     XtManageChild(cascade);
276 #ifdef noGLwidget
277     menupane = XmCreatePulldownMenu(menubar, "menupane", menuPaneArgs, 1);
278 #else
279     menupane = XmCreateP
```