Copyright ©1995 Mark J. Kilgard. All rights reserved.

PUBLISHED IN THE
JULY/AUGUST 1995 ISSUE OF *The X Journal*.

# OPENGL® AND X, COLUMN 5: EXTENSIONS TO OPENGL

Mark J. Kilgard *
*Silicon Graphics Inc.*
*Revision* : 1.3

April 18, 1995

Previous OpenGL® columns surveyed interesting aspects of OpenGL's core rendering functionality. This column is different; it focuses on how OpenGL implementations are being extended for *even better* rendering functionality.

For new readers, the OpenGL graphics system is a standard programming interface for 3D graphics. OpenGL is window system independent and is available for the X Window System, Windows NT, and OS/2. Implementations for the Macintosh and Windows 95 are planned.

Compared with other similar graphics standards, OpenGL mandates a rather high baseline of supported rendering functionality. This is good because it means you will not find OpenGL outdated in a year or two. OpenGL's rendering functionality is state-of-the-art for even today's most powerful graphics workstations, but computer graphics is a rapidly evolving field. For this reason, OpenGL has been designed so it can be extended with additional rendering capabilities.

## OpenGL Extension Naming

An OpenGL extension defines additional OpenGL functionality beyond what the OpenGL core defines. An OpenGL implementation may implement zero or more OpenGL extensions. Each extension has a unique name. While OpenGL implementors are free to implement their own extensions, OpenGL's governing body, the OpenGL Architectural Review Board, works to coordinate standard OpenGL extensions and make sure vendor-specific extensions do not clash.

Names for extensions are designed to make it clear whether an extension is proposed by only a single vendor or if multiple vendors have endorsed the extension. The `EXT_` prefix is used to indicate multiple vendors have endorsed the extension. This may or may not mean that multiple implementations of the extension currently exist. Vendor specific extensions are prefixed by a vendor identifier like `IBM_` or `SGI_`. For vendor specific extensions, further conventions may exist to distinguish classes of vendor specific extensions. For example, SGI uses a trailing `S` after its vendor identifier (like `SGIS_`) to indicate the extension is likely to only be supported on certain hardware configurations. A trailing `X` after SGI's vendor identifier (like `SGIX_`) indicates that the extension is considered experimental and may not be supported in future releases.

The OpenGL extensions supported by an OpenGL implementation are determined by calling:

```
extensions = glGetString(GL_EXTENSIONS);
```

The returned string is a space-separated list of OpenGL extension names supported by the implementation. By convention, every extension name listed in the string is prefixed by `GL_`. The application program is responsible for parsing the string to determine if a particular extension is supported before using it.

An extension adds new routines, constants, and/or types to the OpenGL API. The names of added routines, types, or constants are suffixed with the extension name's prefix. A routine, constant, or type with an `EXT_` prefix would be named, respectively:

```
glRoutineEXT
GL_CONSTANT_EXT
GLtypeEXT
```

The specification for an extension documents how the extension affects the OpenGL state machine. An OpenGL extension is not limited to adding new routines to establish new functionality. An extension may provide new enumerants to be passed to existing OpenGL routines. An extension may also establish new

*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`

OpenGL state that can affect the operation of existing OpenGL routines.

## Available Extensions

There are currently several dozen specified OpenGL extensions for which implementations exist or are planned. Extensions are proposed for numerous reasons including adding new capabilities, generalizing existing functionality, improving rendering quality, improving performance, and overcoming design shortcomings. Vendors are only now beginning to release OpenGL implementations that implement extensions. Discussing every proposed OpenGL extension is beyond the scope of this column. Instead, we survey only a few extensions:

`EXT_agbr` is a simple extension to augment the available host-memory image formats. It provides a reverse-order alternative to the RGBA image format. AGBR refers to the order that the red, green, blue, and alpha components are represented in memory. The predecessor to OpenGL used the AGBR image format, so many existing applications and their file formats rely on an image format reversed from OpenGL's default. This extension allows OpenGL to efficiently support images in the AGBR format.

`EXT_polygon_offset` allows depth values of polygon fragments to be displaced so that lines, points, and polygons that lie in the same plane can be rendered without interaction. The polygon offset extension is useful to avoid artifacts that occur when lines are used to draw outlines around polygons or when multiple co-planar polygons need to be rendered. This extension is covered in more detail later because this column's example demonstrates the `EXT_polygon_offset` extension.

`SGIS_multisample` provides a mechanism to antialias all primitives. The technique is to sample all primitives multiple times at different locations within each pixel (rather than just at the pixel center). The color sample values are resolved to a single displayable color each time a pixel is updated, so antialiasing appears automatic at the application level. Multisampling permits extremely high quality animated scenes. Currently, multisampling is implemented only in high-end graphics hardware such as SGI's RealityEngine.

`EXT_texture3D` supports three-dimensional texture mapping. In the core OpenGL, texture coordinates are already logically manipulated as (homogeneous) 3D coordinates. This extension defines in-memory formats for 3D images, and adds pixel formats to support them. 3D textures are useful for volumetric rendering. For example, a 256x256x256 cube of medical or seismic data could be sliced into arbitrarily if represented as a 3D texture (potentially rendered in real-time if 3D textures are supported by hardware!).

`EXT_texture_object` supports named texture objects whose contents and parameters may be changed after they are defined. Texture objects are designed to make up for deficiencies in efficiently using display lists to package textures. For machines with special texture memories, the extension also provides simple texture memory management.

`EXT_subtexture` allows a contiguous portion of an already-existing texture image to be redefined without affecting the remaining portion of the image or any of the other state that describes the texture. One application of subtexture loading is to support "roaming" through image data viewed as a texture.

`EXT_convolution` adds 1D and 2D convolution operations to OpenGL's pixel transfer process. Thus pixel drawing, reading, and copying, as well as texture image definition, are all candidates for convolution. Convolution is an important image processing operation.

`SGIS_sharpen_texture` introduces texture magnification filters that sharpen the resulting image when greatly magnified. Instead of simply magnifying the texture's highest level of detail (the level 0 mipmap), by extrapolating from the texture's second highest level of detail (the level 1 mipmap) to the texture's highest level of detail (the level 0 mipmap), an apparently sharper resulting texture can be rendered. Texture sharpening can be specified on a per-component basis. Often the alpha component of a texture is used to define a pattern like a tree shape. Texture sharpening of the alpha component of such "shaped" textures can be used to avoid blurring shape edges that otherwise result from a simpler sampling of the alpha values at the edges.

`EXT_vertex_array` adds the ability to specify multiple geometric primitives with very few subroutine calls. Instead of calling an OpenGL procedure for each individual vertex, normal, or color, arrays of vertexes, normals, and colors are prespecified and dispatched via a single subroutine call. The vertex array extension promises to improve performance for *extremely* fast graphics hardware where host overhead may be a potential bottleneck, and on CPUs where the cost of a subroutine call is relatively expensive (like the Intel x86 architecture).

## Extension Interoperability

OpenGL is logically a client/server system. For a program to use an OpenGL extension, the extension must be supported by both the OpenGL client library and the OpenGL server implementation. A given extension might not be supported for a number of reasons. For example, the OpenGL server implementation might be running a previous release of OpenGL that predated the extension, or the OpenGL server might be from a vendor that does not support the extension. OpenGL programs that use extensions should be careful to make sure they properly query the existence of extensions they will be using. This is no different than what X programs should do before using X extensions.

In general, applications should ensure they operate when a given extension is not available. For example, a program using the `EXT_vertex_array` extension can fall back to the more conventional means of generating vertex, normal, and color data. Programs using extensions like `SGIS_multisample` or `SGIS_sharpen_texture` can generally operate without the

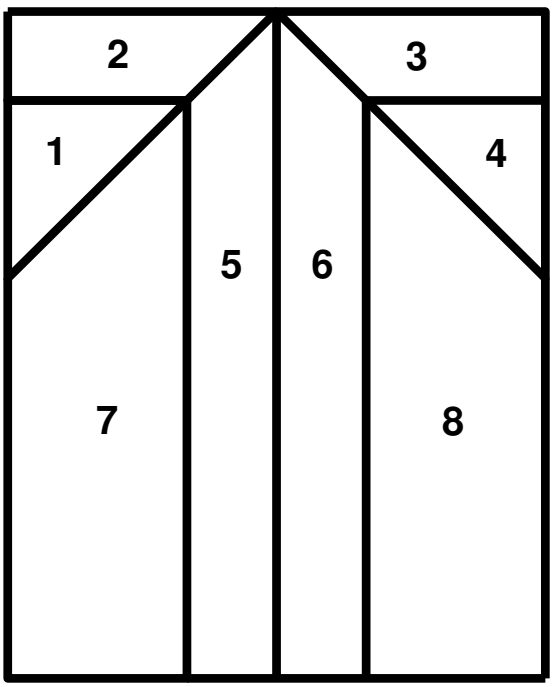


Figure 1: The paper airplane is divided into eight polygons defined by the fold lines of the folded plane.

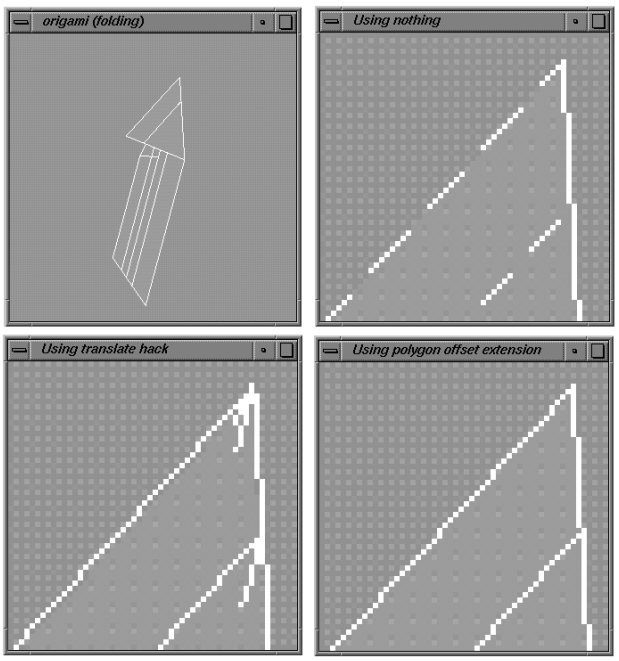


Figure 2: The `origami` example program and three different outline modes. Notice that using the polygon offset extension gives the cleanest result. Rendering the fold and edge outlines using either "nothing" or the "translate hack" method results in noticeable artifacts seen in the magnified views.

extension, admittedly at reduced image quality.

## An Extension Example

This column's example folds and unfolds a paper airplane as the airplane spins in 3D. The `origami` program works by modeling the paper airplane using eight polygons as pictured in Figure 1. The program then translates and rotates these polygons to animate the folding and unfolding of the paper plane. The airplane polygons are rendered in a single color, so outlines around each polygon help delineate the folds and edges of the airplane.

The `EXT_polygon_offset` extension is used (if available) to properly outline the folds and edges of the paper. Without the polygon offset extension, unsightly artifacts are generated when rendering the outline folds and edges as seen in Figure 2. If the polygons and outlines are rendered naively with the same vertex coordinates, each polygon may obscure its associated outline in some orientations because of the way the depth buffering sorts fragments. One solution to avoid these artifacts is to slightly translate the outline closer to the viewer, but this can lead to different artifacts. When translated slightly forward, the outline may sometimes appear to "float" above its polygon at some orientations. The polygon offset extension can be used to avoid both types of outlining artifacts by providing a way to offset depth values for polygons without changing the polygon vertices.

The polygon offset extension is enabled like other OpenGL modes. The following statements enable and disable operation of the polygon offset extension:

```
glEnable(GL_POLYGON_OFFSET_EXT);
glDisable(GL_POLYGON_OFFSET_EXT);
```

When enabled, the polygon offset extension displaces the depth values of fragments generated by rendering polygons by an amount that is proportional to the maximum absolute value of the depth slope of the polygon, measured and applied in window coordinates. This displacement allows lines (or points) and polygons in the same plane to be rendered without interaction, meaning the lines can be rendered either completely in front of or behind the polygons (depending on the sign of the offset factor). It also allows multiple coplanar polygons to be rendered without interaction, if different offset factors are used for each polygon.

The extension's `glPolygonOffsetEXT` routine takes two parameters specifying the scaling *factor* which is used to create a variable depth offset for each polygon and the *bias* which is a constant depth offset. For example:

```
glPolygonOffsetEXT(/*factor*/ 0.5,
                   /*bias*/ 0.001);
```

Initially, the factor and bias are both zero. Polygon offset is useful for rendering hidden line images, for applying decals to surfaces, and for rendering surfaces with highlighted edges.

Notice in `origami.c` that all code relating to using the polygon offset extension is guarded by `#ifdef` C preprocessor directives. This ensures the code will cleanly compile on an

OpenGL library implementation that does not provide the polygon offset extension. By convention, each system's OpenGL header file advertises what extensions the system's OpenGL client library provides based on C preprocessor `#defines`. The `GL_EXT_polygon_offset` name is defined only if the polygon offset extension is supported by the client library OpenGL implementation.

In additional to the compile-time check, a run-time check should also be made if the OpenGL server supports the polygon offset extension. As a rule, do not call OpenGL extension routines until you have verified the extension is supported. The `glutExtensionSupported` routine (part of the OpenGL Utility Toolkit) makes it easy to query if a named extension is supported. When the extension is supported, `origami` will use it; when not supported, `origami` will still run but uses the "translate hack" instead of the cleaner polygon offset extension.

## GLX Extensions

So far we have focused only on OpenGL extensions, but it is also possible to extend the OpenGL's window system support API, for example, GLX in the case of the X Window System. The original GLX 1.0 did not provide support for extensions, so GLX 1.1 added this capability by introducing a few new GLX routines:

**`glXGetClientString`** returns vendor information, version information, and the GLX extensions supported by the client-side GLX library.

**`glXGetServerString`** returns vendor information, version information, and the GLX extensions supported by the server-side GLX implementation.

**`glXQueryExtensionsString`** returns a string describing the GLX extensions supported by the connection. This is the intersection of the supported client and server extensions.

GLX 1.1 does not actually add any extensions itself; it just provides a mechanism for using extensions. While an official OpenGL standard, GLX 1.1 implementations are only now becoming available.[1] At compile time, the C preprocessor name `GLX_VERSION_1_1` is defined if GLX 1.1 is supported. At run time, you can use the GLX 1.0 `glXQueryVersion` routine to determine if your GLX implementation has 1.1 support.

GLX extensions follow the same conventions as OpenGL extensions for naming prefixes and suffixes. The functionality of GLX extensions, unlike OpenGL extensions, involves interfacing OpenGL with the window system and/or operating system. Examples of currently specified GLX extensions are:

`SGIS_multisample` provides the mechanism for selecting X Visuals that support multisampling for use with the `SGIS_multisample` OpenGL extension.

`SGI_make_current_read` allows OpenGL pixel operations to read pixel data from one drawable and draw into the buffers of another. For example, pixels can be copied from one window to another. Previously, GLX's `glXMakeCurrent` routine allowed pixel copies involving only memory and a single drawable.

`SGI_video_sync` provides a means for synchronization with the video frame rate of the monitor. This is important for real-time graphics applications with multiple threads of execution that must stay in step with graphics output.

`SGI_swap_control` modifies the semantics of `glXSwapBuffers` to allow a minimum number of monitor vertical retraces between buffer swaps. This helps animation applications maintain a constant frame rate.

## Extensions as OpenGL's Future

It is likely that a future major or minor revision to the OpenGL specification will "roll up" successful OpenGL extensions into the core of OpenGL's functionality. In this sense, OpenGL extensions are a testing ground for new functionality and a glimpse at what a future revision to OpenGL might contain.

Advances in graphics hardware and rendering support happen quickly. The OpenGL extension mechanism provides a framework that lets OpenGL vendors develop new rendering functionality in a controlled and consistent manner. To find out more about the extensions supported by a given OpenGL implementation, consult the vendor's documentation. Information on support extensions is often found on the `glIntro` and `glXIntro` man page.

Next issue's column explores the wide variety of ways graphics hardware can accelerate OpenGL rendering. The focus will be to provide you with enough knowledge about 3D graphics hardware architecture to select the right graphics hardware for your needs and budget.

[1] IRIX 5.3 from Silicon Graphics does support GLX 1.1.

# A origami.c

```
#include <stdlib.h>
#include <stdio.h>
#include <GL/glut.h>

enum {
  FLAT,                 /* completely flat sheet of paper */
  FLAP1,                /* left flap being folded in */
  FLAP2,                /* right flap being folded int */
  CENTER2,              /* right side folded up at center */
  WING2,                /* right wing folded down */
  CENTER1,              /* left side folded up at center */
  WING1,                /* left wing folded down */
  FOLDED                /* fully folded paper airplane */
} States;

#if defined(GL_EXT_polygon_offset)
int polygon_offset;
#endif
int motion = 1, spinning = 1, state = FLAT;
int click = 0, delay = 0, direction;
float flap1_angle = 0, flap2_angle = 0, center1_angle = 0;
float center2_angle = 0, wing1_angle = 0, wing2_angle = 0;

typedef GLfloat Point[2];
Point poly1[] =
{ {-1, 0}, {-1/3.0, 2/3.0}, {-1, 2/3.0} };
Point poly2[] =
{ {-1, 1}, {-1, 2/3.0}, {-1/3.0, 2/3.0}, {0, 1} };
Point poly3[] =
{ {0, 1}, {1, 1}, {1, 2/3.0}, {1/3.0, 2/3.0} };
Point poly4[] =
{ {1/3.0, 2/3.0}, {1, 2/3.0}, {1, 0} };
Point poly5[] =
{ {-1/3.0, 2/3.0}, {0, 1}, {0, -1.5}, {-1/3.0, -1.5} };
Point poly6[] =
{ {0, 1}, {1/3.0, 2/3.0}, {1/3.0, -1.5}, {0, -1.5} };
Point poly7[] =
{ {-1, 0}, {-1/3.0, 2/3.0}, {-1/3.0, -1.5}, {-1, -1.5} };
Point poly8[] =
{ {1, 0}, {1/3.0, 2/3.0}, {1/3.0, -1.5}, {1, -1.5} };

void
polydlist(int dlist, int num, Point points[])
{
  int i;

  glNewList(dlist, GL_COMPILE);
  glBegin(GL_POLYGON);
  for (i = 0; i < num; i++)
    glVertex2fv(&points[i][0]);
  glEnd();
  glEndList();
}

void
idle(void)
{
  if (spinning) click++;
```

```
  switch (state) {
  case FLAT:
    delay++;
    if (delay >= 80) {
      delay = 0;
      state = FLAP1;
      glutSetWindowTitle("origami (folding)");
      direction = 1;
    }
    break;
  case FLAP1:
    flap1_angle += 2 * direction;
    if (flap1_angle >= 180) state = FLAP2;
    else if (flap1_angle <= 0) state = FLAT;
    break;
  case FLAP2:
    flap2_angle += 2 * direction;
    if (flap2_angle >= 180) state = CENTER2;
    else if (flap2_angle <= 0) state = FLAP1;
    break;
  case CENTER2:
    center2_angle += 2 * direction;
    if (center2_angle >= 84) state = WING2;
    else if (center2_angle <= 0) state = FLAP2;
    break;
  case WING2:
    wing2_angle += 2 * direction;
    if (wing2_angle >= 84) state = CENTER1;
    else if (wing2_angle <= 0) state = CENTER2;
    break;
  case CENTER1:
    center1_angle += 2 * direction;
    if (center1_angle >= 84) state = WING1;
    else if (center1_angle <= 0) state = WING2;
    break;
  case WING1:
    wing1_angle += 2 * direction;
    if (wing1_angle >= 84) state = FOLDED;
    else if (wing1_angle <= 0) state = CENTER1;
    break;
  case FOLDED:
    delay++;
    if (delay >= 80) {
      delay = 0;
      glutSetWindowTitle("origami (unfolding)");
      direction = -1;
      state = WING1;
    }
    break;
  }
  glutPostRedisplay();
}

void
draw_folded_plane(void)
{
  glPushMatrix();
    glRotatef(click, 0, 0, 1);
    glRotatef(click / 5.0, 0, 1, 0);
    glTranslatef(0, .25, 0);
```

```
    glPushMatrix();
      glRotatef(center1_angle, 0, 1, 0);
      glPushMatrix();
        glTranslatef(-.5, .5, 0);
        glRotatef(flap1_angle, 1, 1, 0);
        glTranslatef(.5, -.5, 0);
        glCallList(2);
      glPopMatrix();
      glCallList(5);
      glPushMatrix();
        glTranslatef(-1 / 3.0, 0, 0);
        glRotatef(-wing1_angle, 0, 1, 0);
        glTranslatef(1 / 3.0, 0, 0);
        glCallList(7);
        glPushMatrix();
          glTranslatef(-.5, .5, 0);
          glRotatef(flap1_angle, 1, 1, 0);
          glTranslatef(.5, -.5, 0);
          glCallList(1);
        glPopMatrix();
      glPopMatrix();
    glPopMatrix();
    glPushMatrix();
      glRotatef(-center2_angle, 0, 1, 0);
      glPushMatrix();
        glTranslatef(.5, .5, 0);
        glRotatef(-flap2_angle, -1, 1, 0);
        glTranslatef(-.5, -.5, 0);
        glCallList(3);
      glPopMatrix();
      glCallList(6);
      glPushMatrix();
        glTranslatef(1 / 3.0, 0, 0);
        glRotatef(wing2_angle, 0, 1, 0);
        glTranslatef(-1 / 3.0, 0, 0);
        glCallList(8);
        glPushMatrix();
          glTranslatef(.5, .5, 0);
          glRotatef(-flap2_angle, -1, 1, 0);
          glTranslatef(-.5, -.5, 0);
          glCallList(4);
        glPopMatrix();
      glPopMatrix();
    glPopMatrix();
  glPopMatrix();
}

void
display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
  glColor3ub(67, 205, 128);
#if defined(GL_EXT_polygon_offset)
  if (polygon_offset) glEnable(GL_POLYGON_OFFSET_EXT);
#endif
  draw_folded_plane();
  glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
  glColor3ub(255, 255, 255);
#if defined(GL_EXT_polygon_offset)
```

```
  if (polygon_offset) {
    /* XXX a bug in the unpatched IRIX 5.3 OpenGL posts
       GL_INVALID_ENUM when GL_POLYGON_OFFSET_EXT is disabled;
       please ignore it. */
    glDisable(GL_POLYGON_OFFSET_EXT);
  } else
#endif
    {
      glPushMatrix();
      glTranslatef(0, 0, .1);
    }
  draw_folded_plane();
#if defined(GL_EXT_polygon_offset)
  if(!polygon_offset)
#endif
    glPopMatrix();
  glutSwapBuffers();
}

void
visible(int state)
{
  if (state == GLUT_VISIBLE && motion) glutIdleFunc(idle);
  else glutIdleFunc(NULL);
}

void
menu(int value)
{
  switch (value) {
  case 1:
    direction = -direction;
    if (direction > 0) glutSetWindowTitle("origami (folding)");
    else glutSetWindowTitle("origami (unfolding)");
    break;
  case 2:
    motion = 1 - motion;
    if (motion) glutIdleFunc(idle);
    else glutIdleFunc(NULL);
    break;
  case 3:
    spinning = 1 - spinning;
    break;
  }
}

void
main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH | GLUT_DOUBLE);
  glutCreateWindow("origami");
  glutDisplayFunc(display);
  glutVisibilityFunc(visible);
  glClearColor(.488, .617, .75, 1.0);
  glMatrixMode(GL_PROJECTION);
  gluPerspective(35.0, 1.0, 0.2, 7.0);
  glMatrixMode(GL_MODELVIEW);
  gluLookAt(0, 0, 5.5, 0, 0, 0, 0, 1, 0);
  glEnable(GL_DEPTH_TEST);
```

```
  glDepthFunc(GL_LEQUAL);
  glLineWidth(1.0);
  polydlist(1, sizeof(poly1) / sizeof(Point), poly1);
  polydlist(2, sizeof(poly2) / sizeof(Point), poly2);
  polydlist(3, sizeof(poly3) / sizeof(Point), poly3);
  polydlist(4, sizeof(poly4) / sizeof(Point), poly4);
  polydlist(5, sizeof(poly5) / sizeof(Point), poly5);
  polydlist(6, sizeof(poly6) / sizeof(Point), poly6);
  polydlist(7, sizeof(poly7) / sizeof(Point), poly7);
  polydlist(8, sizeof(poly8) / sizeof(Point), poly8);
  glutCreateMenu(menu);
  glutAddMenuEntry("Reverse direction", 1);
  glutAddMenuEntry("Toggle motion", 2);
  glutAddMenuEntry("Toggle spinning", 3);
#if defined(GL_EXT_polygon_offset)
  polygon_offset = glutExtensionSupported("GL_EXT_polygon_offset");
  if(polygon_offset) glPolygonOffsetEXT(0.8, 0.0);
#endif
  glutAttachMenu(GLUT_RIGHT_BUTTON);
  glutMainLoop();
}
```