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

PUBLISHED IN THE
MARCH/APRIL 1995 ISSUE OF *The X Journal*.

OPENGL® AND X, COLUMN 3: TEXTURE MAPPING

Mark J. Kilgard *
Silicon Graphics Inc.
Revision : 1.7

November 1, 1995

The OpenGL® graphics interface is now acknowledged as the premier standard for interactive 3D computer graphics. Evidence of OpenGL's success abounds. The *Open Systems Today* trade journal recently proclaimed: "The 3-D graphics API wars are over, and OpenGL has won." And when *PC Magazine* awarded Windows NT 3.5 a Technical Excellence Award, OpenGL was cited as "the most significant" new feature. The `comp.graphics.opengl` Internet newsgroup generates significantly more traffic than the newsgroups dedicated to competitive standards such as PEX. Industry standard OpenGL benchmarks are being finalized so workstation users can competitively evaluate the price-performance of available OpenGL implementations. Future developments such as OpenGL for OS/2 and add-on OpenGL support for Windows 95 promise to further widen OpenGL's appeal.

An important technical reason behind OpenGL's success is its *required* support of advanced rendering features like texture mapping. The focus of this issue's column is to explain texture mapping for OpenGL. Like previous columns, the OpenGL Utility Toolkit (GLUT) is used in the column's example program.

What is Texture Mapping?

Texture mapping (often called simply texturing) is the process of applying an image to the surface of a geometric primitive. In the real world, most surfaces are not of a single color. For example, a wooden table has a wood grain pattern across its surface. If a computer were to render such a table, attempting to render the intricate wood grain surface using simple triangles and rectangles, it would likely be too slow. Instead we can use a 2D image of a wood grain pattern and *texture* the image onto the model of the table. You can think of a texture as a form of "wallpaper" for rendered 3D polygons.

OpenGL is very careful to apply textures in such a way that even if a textured polygon is rotated, translated, scaled, or even projected, the texture is rendered appropriately.

Texturing can make the difference between a realistic scene and one that is flat and obviously computer-generated. If you look at the world around you, most surfaces can be thought of as having some texture. For example, a brick wall, the label on a can of soup, and a reflection in a mirror could all be rendered using textures.

Texture mapping is useful for more purposes than simply adding a surface image to rendered polygons. Texturing can also greatly reduce the geometric complexity of a scene. Imagine a tree in the distance. Rendering such a tree with all its leaves and limbs could take a long time. Instead, you could render a single rectangle and texture the image of a tree onto it. Or a flight simulator could render forest-covered ground by texturing the aerial image of a forest onto a few polygons representing the ground. Figures 1, 4, and 5 show examples of the use of texture mapping.

Using Textures with OpenGL

Textured objects are rendered with the same OpenGL routines used to render geometric primitives. Additional OpenGL calls specify a texture and how the texture is applied to the geometry you render. Using texture mapping can be broken up into four steps:

1. Enabling texture mapping.
2. Specifying the texture.
3. Specifying how the texture is to be applied to the surface.

*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`

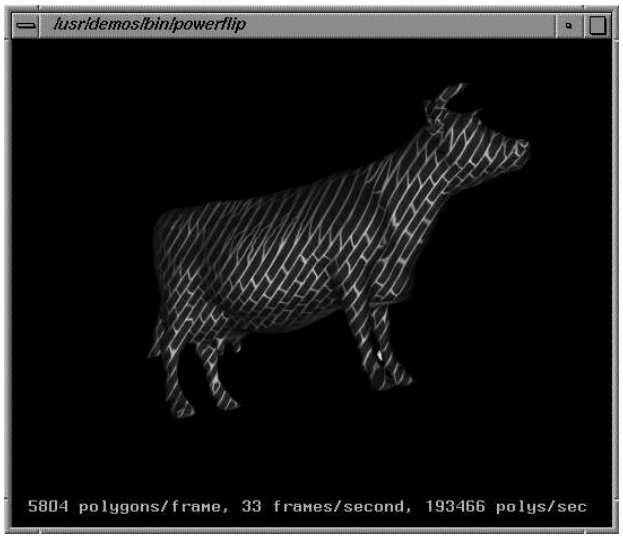

Figure 1: Cow model textured mapped with a red brick texture.

4. Rendering the primitives, supplying both geometric coordinates *and* texture coordinates.

Enabling texturing

Like other OpenGL capabilities, texturing is enabled and disabled using the `glEnable` and `glDisable` calls. Texturing is initially disabled when an OpenGL rendering context is created. The following call enables texturing:

```
glEnable(GL_TEXTURE_2D);
```

Because not everything in your scene may require texturing (and textured primitives render slower than non-textured primitives), disable texturing when not needed.

Specifying a texture

The simplest texture is defined by a single 2D image (1D textures are also possible). The routine `glTexImage2D` specifies a 2D image texture. The parameters to `glTexImage2D` are an array of pixel values for the texture itself along with the texture's height, width, type, format, internal resolution, border width, and level-of-detail. The width and height of a texture must be some (possibly different) power of two.

Where do the texture images themselves come from? They can be programatically generated, but it is more likely they are generated with a paint program, scanner, or digital camera. The texture of my face used in this column's example program (see Figure 4) was generated with a Silicon Graphics Indy workstation's standard digital camera.

The individual cells that make up a texture image are named *texels* (texture elements). OpenGL's texture mapping functionality is supported only for the RGBA color model and not the color index model.

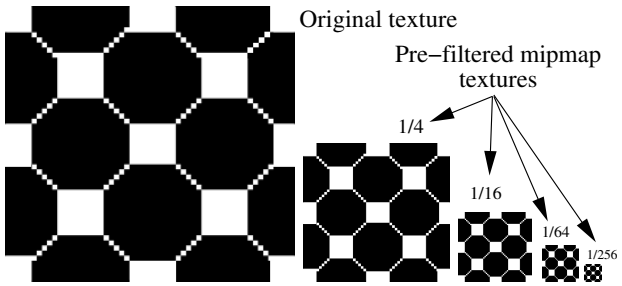

Figure 2: Multiple levels-of-detail for a texture using mipmaps.

Because polygons can be arbitrarily scaled, the final size of a textured polygon on the screen may vary. To avoid the visual artifacts of under-sampling a texture to fit a small polygon in window coordinates (relative to the texture size), OpenGL supports a technique called *mipmapping* that allows several versions of the texture image at smaller sizes to be used. Figure 2 shows how mipmaps are organized as a set of pre-filtered textures at diminishing resolutions. Without mipmapping, small textured objects might shimmer and flash as they move. When mipmap filtering is used, OpenGL automatically picks or interpolates the right *level-of-detail* from the set of available mipmaps. The GLU utility routine `gluBuild2DMipmaps` can assist you building correctly filtered mipmaps.

Because there are a large number of texture formats and types and setting a texture involves OpenGL's pixel path, the details of the format of texture data are beyond the scope of this article. For further information about constructing textures using `glTexImage2D` and `gluBuild2DMipmaps` refer to the OpenGL "red book" [4] or the OpenGL manual pages for these routines.

Specifying texture modes

Once a texture is specified, OpenGL also needs to be informed how you want the texture to be applied to each pixel. OpenGL provides a number of texture filtering methods and texture modes to control how pixels are textured.

OpenGL's texture mode determines how the texture values are combined with the primitive's color values (those generated by lighting calculations or by explicit vertex color assignment) when generating textured fragments. *Fragments* are generated during the rasterization of an OpenGL primitive. A fragment is the bundle of information necessary to update a specific pixel location in the frame buffer. In particular, it determines how the color of a pixel is updated. The *texture environment* that determines how texture values affect the color of a fragment can be one of:

`GL_DECAL` The color of a textured fragment in decal mode is determined directly by the texture's color components. The texture values override the color values for the fragment. Specify decal mode with the following call:

```
glTexEnvi(GL_TEXTURE_ENV,
    GL_TEXTURE_ENV_MODE, GL_DECAL);
```

GL_MODULATE The color of a textured fragment in modulate mode is a combination of the texture color and the color values generated by lighting calculations or explicit vertex color assignment. Using the modulate mode allows texturing to interact with colors determined by OpenGL's lighting model.

```
glTexEnvi(GL_TEXTURE_ENV,
    GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

GL_BLEND Blend mode is useful for a texture that represents luminance (brightness). The color of a textured pixel in blend mode is a combination of the color values generated by lighting calculations or explicit vertex color assignment blended with a *texture environment color* with the blending factor based on the texture's luminance values.

```
glTexEnvi(GL_TEXTURE_ENV,
    GL_TEXTURE_ENV_MODE, GL_MODULATE);
glTexEnvfv(GL_TEXTURE_ENV,
    GL_TEXTURE_ENV_COLOR, &rgba_vec);
```

Typically, if you use an RGB texture, you should use the modulate mode if you want texture and lighting to interact, otherwise use the decal mode. The default texture mode is GL_MODULATE.

OpenGL also allows control over the filtering algorithms used to determine texture values. When the rendered primitive in screen space is smaller than the actual texture, a *minification* filter determines how and which texels determine the texture color for a fragment. For example, the GL_NEAREST filter picks the texel in texture coordinates nearest the center of the pixel, ignoring mipmaps. GL_LINEAR performs a weighted sum of the four texels nearest in texture coordinates to the center of the pixel, ignoring mipmaps. More sophisticated and computationally expensive filtering methods use mipmaps. For example, GL_NEAREST_MIPMAP_LINEAR (the default minification filter) samples the nearest texel in each of the two nearest mipmap images, then uses a weighted sum of the two samples.

Filtering may also be necessary when the rendered primitive is larger in screen space than the texture. The texture *magnification* filter controls this case. Mipmapping works only for minification so the only magnification filtering options are GL_NEAREST and GL_LINEAR (the default).

Better texture filtering can avoid some of the visual artifacts associated with animated texture mapped scenes. A poor filter may allow the texture to "swim" or jitter, particularly when a textured object is moving slowly. Better filtering is traded off with texture rendering performance.

Rendering textured primitives

Once a texture and the necessary texturing parameters are specified, rendering of textured primitives can begin. In addition to object coordinates generated by the glVertex3f family of routines, a textured primitive requires per-vertex *texture coordinates* to be specified. Texture coordinates determine which location within the texture map is assigned to each vertex. The complete texture is mapped onto the surface of the primitive by interpolating the texture coordinates between vertices. Figure 3 shows how texture coordinates map texture information to the surface of a polygon.

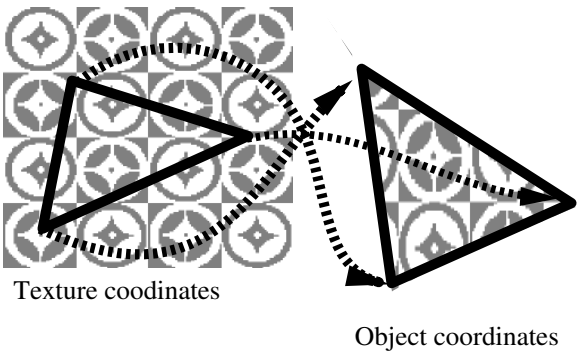

Figure 3: Texture coordinates determine how texels in the texture are mapped to the surface of a triangle in object coordinates.

Texture coordinates are explicitly assigned with the glTexCoord2f family of routines. Usually, texture coordinate values range between 0 and 1. The following is an example of generating texture coordinates for a triangle.

```
glBegin(GL_TRIANGLES);
  glTexCoord2f(0.1, 0.2);
    glVertex3f(0, 1, 3);
  glTexCoord2f(0.2, 0.7);
    glVertex3f(2, 9, 2);
  glTexCoord2f(0.8, 0.5);
    glVertex3f(7, 6, 8);
glEnd():
```

Texels for texture coordinates falling outside the range 0 to 1 can be either be clamped to 0 or 1 or *repeated* by using only the fractional component of the texture coordinates. The texture for a brick pattern probably can be repeated; the texture for a billboard should be clamped.

More texturing features

OpenGL texturing also supports a number of features beyond the scope of this article, but the features are worth mentioning briefly. OpenGL supports the automatic generation of texture coordinates based on the object coordinates of vertices. This saves the overhead of specifying texture coordinates when the texture coordinates can be generated based on the object coordinates. Like modeling coordinates, texture coordinates can be transformed by a *texture matrix* before use. As mentioned previously, OpenGL supports one dimensional textures. 1D textures are useful for scientific visualization where the 1D texture

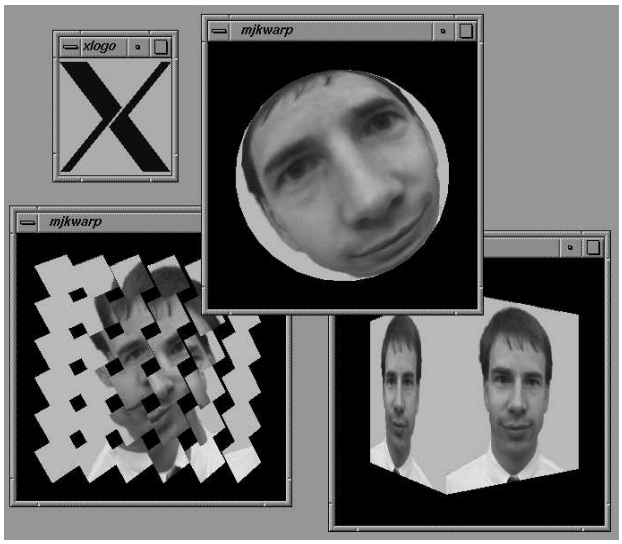

Figure 4: Various modes of `mjkwarp` example. Texturing techniques can be used to rotate, contort, and shatter an image.

coordinate might be used to represent, for example, temperature over the surface of an object. Proposed extensions to OpenGL also permit 3D textures, useful for volumetric rendering of information like 3D seismic and medical imaging data.

If you switch textures repeatedly, it is useful to make a display list for each texture you use frequently. This allows OpenGL to avoid copying the texture and other overhead every time `glTexImage2D` is executed. This is particularly useful when OpenGL direct rendering is not available.

An Example

The `mjkwarp` example program listed at the end of the article demonstrates using texture mapping to rotate, contort, and shatter a 2D image, all while the image spins. The effects are reminiscent of MTV video effects. The point of these effects is to demonstrate a few of the possibilities texture mapping permits. Figure 4 shows `mjkwarp` in its various texturing modes:

Elastic sheet. The scene is a 12 by 12 grid of squares with a portion of my face textured to each square in the grid. The local X and Y vertex components of each square are fixed while the local Z component varies sinusoidally over time, with the amplitude higher near the center of the grid.

The result is a spinning image that appears to stretch inwards and outwards as if the image were affixed to a pulsating elastic sheet of rubber.

By rendering "patches" of a texture onto smaller polygons composing a smooth surface, textures can appear to smoothly deform over a complex surface.

Spinning cube. A simpler texturing effect, my face is textured onto four sides of a spinning cube.

The texture is correctly rendered in perspective even when a cube side is rotated almost out of view.

The distance from the cube to the viewer also varies sinusoidally with time, so the size of the textured image on the screen varies. Mipmapping avoids under-sampling artifacts when the cube's size in the window is small.

Rotating squares. The scene is a 6 by 6 grid of squares, each with a portion of my face. Each square is separately rotated around its center. The result is an image that appears to shatter into a confused pattern of squares, only to rotate back again to the recognizable image of my face.

Along with demonstrating texturing effects, `mjkwarp` allows the minification filtering method to be changed via pop-up menus. Because using a higher quality filtering method is computationally more expensive, better filtering methods are likely to slow the rendering rate of textured geometry.

The speed of texture mapping can vary greatly between OpenGL implementations. Workstations with hardware texturing support achieve much better texturing performance than workstations with no hardware texturing support. Without texturing hardware, texturing calculations (that are expensive because they must be done for each pixel) must be implemented in software. For this reason, `mjkwarp` permits the user to select an appropriate animation rate. The animation advances in real-time with frames calculated no faster than the animation rate via pop-up menus. Without this support, `mjkwarp` can vary wildly in its frame rate as the texturing overhead of the scene varies.

`mjkwarp` is a fairly simple example of using texture mapping. The example uses no lighting effects. For this reason, the `GL_DECAL` texture mode is used. And, only a single texture is used. The actual data for the 256 by 256 pixel RGB image used for the `mjkwarp`'s texture is declared externally and is contained in an `mjkimage.c` source file not listed here since it is merely a large encoded image.

Conclusions

Texture mapping is a versatile rendering technique [1]. It is likely you will hear the term more and more frequently because of the sophistication and realism texturing adds to 3D graphics. For example, texture mapping is becoming quite common in the latest crop of 3D video games. But the utility of texture mapping is hardly limited to video games and visual simulation applications. Texture mapping also has applications in scientific and technical visualization [5] as well as image processing [3]. For those interested in a more detailed technical explanation of texture mapping, see Paul Heckbert's survey of the topic [2].

The example code in this column, like the code for previous articles on OpenGL, is available by anonymous ftp to `sgigate.sgi.com` in the `pub/opengl/xjournal` directory. The next column will discuss another set of OpenGL features: blending, antialiasing, and fog.

References

[1] Paul Haeberli, Mark Segal, "Texture Mapping as a Fundamental Drawing Primitive," *Proceedings of the Fourth Eurographics Workshop on Rendering*, June 1993.

[2] Paul Heckbert, "Survey of Texture Mapping," *IEEE Computer Graphics and Applications*, November 1986.

[3] Carl Korobkin, "Interactive Geometric Image Transformation Using Texture Mapping," *yet unpublished*, Silicon Graphics, 1994.

[4] Jackie Neider, Tom Davis, Mason Woo, *OpenGL Programming Guide: The official guide to learning OpenGL, Release 1*, Addison Wesley, 1993.

[5] Michael Teschner, "Texture Mapping: a new dimension in scientific and technical visualization," *IRIS Universe*, Number 29, 1994.

Figure 5: Forumula 1 racing simulation making extensive use of texture mapping, including texture mapped trees.

A mjkwarp.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>       /* for cos(), sin(), and sqrt() */
#include <GL/glu.h>
#include <GL/glut.h>    /* OpenGL Utility Toolkit */
#include <sys/types.h>
#include <sys/time.h>

/* external texture image array found in mjkimage.c */
extern unsigned char mjk_image[];
extern int mjk_depth, mjk_height, mjk_width; /* 3, 256, 256 */

float tick1 = 0, tick2 = 0, angle, size;
int set_timeout = 0, interval = 100, minify_menu, rate_menu;
#define CUBE 1
#define SQUARES 2
#define DRUM 3
int mode = SQUARES, spinning = 1, scaling = 1, visible = 0;

void animate(int value) {
  if (visible)
    if (spinning || scaling)
      if (value) {
        if (spinning) {
          tick1 += 4 * (interval / 100.0);
          angle = ((int) tick1) % 360;
        }
        if (scaling) {
          tick2 += 2 * (interval / 100.0);
          size = .7 - .5 * sin(tick2 / 20.0);
        }
      }
      glutPostRedisplay();
      set_timeout = 1;
}

#define TIMEDELTA(dest, src1, src2) { \
```

```
        if(((dest).tv_usec = (src1).tv_usec - (src2).tv_usec) < 0) {\
              (dest).tv_usec += 1000000;\
              (dest).tv_sec = (src1).tv_sec - (src2).tv_sec - 1;\
        } else  (dest).tv_sec = (src1).tv_sec - (src2).tv_sec;  }

void redraw(void) {
  struct timeval begin, end, elapsed;
  int i, j;
  float amplitude;

  if (set_timeout) gettimeofday(&begin, NULL);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPushMatrix();
  if (mode != DRUM) glScalef(size, size, size);

  switch (mode) {

#define SQ_COLS 6
#define SQ_TILE_TEX_W (1.0/SQ_COLS)
#define SQ_ROWS 6
#define SQ_TILE_TEX_H (1.0/SQ_ROWS)
  case SQUARES:
    glTranslatef(-SQ_COLS / 2.0 + .5, -SQ_ROWS / 2.0 + .5, 0);
    for (i = 0; i < SQ_COLS; i++)
      for (j = 0; j < SQ_ROWS; j++) {
        glPushMatrix();
        glTranslatef(i, j, 0);
        glRotatef(angle, 0, 1, 1);
        glBegin(GL_QUADS);
        glTexCoord2f(i * SQ_TILE_TEX_W, j * SQ_TILE_TEX_H);
        glVertex2f(-.5, -.5);
        glTexCoord2f((i + 1) * SQ_TILE_TEX_W, j * SQ_TILE_TEX_H);
        glVertex2f(.5, -.5);
        glTexCoord2f((i + 1) * SQ_TILE_TEX_W, (j + 1) * SQ_TILE_TEX_H);
        glVertex2f(.5, .5);
        glTexCoord2f(i * SQ_TILE_TEX_W, (j + 1) * SQ_TILE_TEX_H);
        glVertex2f(-.5, .5);
        glEnd();
        glPopMatrix();
      }
    break;

#define DR_COLS 12
#define DR_TILE_TEX_W (1.0/DR_COLS)
#define DR_ROWS 12
#define DR_TILE_TEX_H (1.0/DR_ROWS)
#define Z(x,y)  (((DR_COLS-(x))*(x) + (DR_ROWS-(y))*(y)) * amplitude) - 28.0
  case DRUM:
    glRotatef(angle, 0, 0, 1);
    glTranslatef(-DR_COLS / 2.0 + .5, -DR_ROWS / 2.0 + .5, 0);
    amplitude = 0.4 * sin(tick2 / 6.0);
    for (i = 0; i < DR_COLS; i++)
      for (j = 0; j < DR_ROWS; j++) {
        glPushMatrix();
        glTranslatef(i, j, 0);
        glBegin(GL_QUADS);
        glTexCoord2f(i * DR_TILE_TEX_W, j * DR_TILE_TEX_H);
        glVertex3f(-.5, -.5, Z(i, j));
        glTexCoord2f((i + 1) * DR_TILE_TEX_W, j * DR_TILE_TEX_H);
        glVertex3f(.5, -.5, Z(i + 1, j));
```

```
        glTexCoord2f((i + 1) * DR_TILE_TEX_W, (j + 1) * DR_TILE_TEX_H);
        glVertex3f(.5, .5, Z(i + 1, j + 1));
        glTexCoord2f(i * DR_TILE_TEX_W, (j + 1) * DR_TILE_TEX_H);
        glVertex3f(-.5, .5, Z(i, j + 1));
        glEnd();
        glPopMatrix();

      }
    break;
  case CUBE:
    glRotatef(angle, 0, 1, 0);
    glBegin(GL_QUADS);
    /* front */
    glTexCoord2f(0.0, 0.0); glVertex3f(-1.0, -1.0, 1.0);
    glTexCoord2f(1.0, 0.0); glVertex3f(1.0, -1.0, 1.0);
    glTexCoord2f(1.0, 1.0); glVertex3f(1.0, 1.0, 1.0);
    glTexCoord2f(0.0, 1.0); glVertex3f(-1.0, 1.0, 1.0);
    /* back */
    glTexCoord2f(0.0, 1.0); glVertex3f(-1.0, 1.0, -1.0);
    glTexCoord2f(1.0, 1.0); glVertex3f(1.0, 1.0, -1.0);
    glTexCoord2f(1.0, 0.0); glVertex3f(1.0, -1.0, -1.0);
    glTexCoord2f(0.0, 0.0); glVertex3f(-1.0, -1.0, -1.0);
    /* left */
    glTexCoord2f(0.0, 0.0); glVertex3f(-1.0, -1.0, -1.0);
    glTexCoord2f(1.0, 0.0); glVertex3f(-1.0, -1.0, 1.0);
    glTexCoord2f(1.0, 1.0); glVertex3f(-1.0, 1.0, 1.0);
    glTexCoord2f(0.0, 1.0); glVertex3f(-1.0, 1.0, -1.0);
    /* right */
    glTexCoord2f(0.0, 1.0); glVertex3f(1.0, 1.0, -1.0);
    glTexCoord2f(1.0, 1.0); glVertex3f(1.0, 1.0, 1.0);
    glTexCoord2f(1.0, 0.0); glVertex3f(1.0, -1.0, 1.0);
    glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, -1.0);
    glEnd();
  }
  glPopMatrix();
  glutSwapBuffers();
  if (set_timeout) {
    set_timeout = 0;
    gettimeofday(&end, NULL);
    TIMEDELTA(elapsed, end, begin);
    if (elapsed.tv_sec >= 1 || elapsed.tv_usec > interval * 1000)
      glutTimerFunc(0, animate, 1);
    else
      glutTimerFunc(interval - (elapsed.tv_usec / 1000), animate, 1);
  }
}

void visibility(int state) {
  if (state == GLUT_VISIBLE) {
    visible = 1;
    animate(0);
  } else
    visible = 0;
}

void minify_select(int value) {
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, value);
  gluBuild2DMipmaps(GL_TEXTURE_2D, mjk_depth, mjk_width, mjk_height,
    GL_RGB, GL_UNSIGNED_BYTE, mjk_image);
  glutPostRedisplay();
```

```
}

void rate_select(int value) {
  interval = value;
}

void menu_select(int value) {
  switch (value) {
  case 1:
    spinning = !spinning;
    if (spinning) animate(0);
    break;
  case 2:
    scaling = !scaling;
    if (scaling) animate(0);
    break;
  case 3:
    mode++;
    if (mode > DRUM) mode = CUBE;
    switch (mode) {
    case CUBE:
      glEnable(GL_CULL_FACE);
      glDisable(GL_DEPTH_TEST);
      break;
    case SQUARES:
      glDisable(GL_CULL_FACE);
      glDisable(GL_DEPTH_TEST);
      break;
    case DRUM:
      glEnable(GL_DEPTH_TEST);
      glDisable(GL_CULL_FACE);
      break;
    }
    glutPostRedisplay();
    break;
  case 666:
    exit(0);
  }
}

void main(int argc, char **argv) {
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
  glutCreateWindow("mjkwarp");
  glutDisplayFunc(redraw);
  glMatrixMode(GL_PROJECTION);
  gluPerspective( /* field of view in degree */ 40.0,
    /* aspect ratio */ 1.0, /* Z near */ 1.0, /* Z far */ 70.0);
  glMatrixMode(GL_MODELVIEW);
  gluLookAt(0.0, 0.0, 5.0,  /* eye is at (0,0,30) */
    0.0, 0.0, 0.0,      /* center is at (0,0,0) */
    0.0, 1.0, 0.);      /* up is in postivie Y direction */
  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
  gluBuild2DMipmaps(GL_TEXTURE_2D, mjk_depth, mjk_width, mjk_height,
    GL_RGB, GL_UNSIGNED_BYTE, mjk_image);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
```

```
  glEnable(GL_TEXTURE_2D);
  glutVisibilityFunc(visibility);
  minify_menu = glutCreateMenu(minify_select);
  glutAddMenuEntry("Nearest", GL_NEAREST);
  glutAddMenuEntry("Linear", GL_LINEAR);
  glutAddMenuEntry("Nearest mipmap nearest", GL_NEAREST_MIPMAP_NEAREST);
  glutAddMenuEntry("Linear mipmap nearest", GL_LINEAR_MIPMAP_NEAREST);
  glutAddMenuEntry("Nearest mipmap linear", GL_NEAREST_MIPMAP_LINEAR);
  glutAddMenuEntry("Linear mipmap linear", GL_LINEAR_MIPMAP_LINEAR);
  rate_menu = glutCreateMenu(rate_select);
  glutAddMenuEntry(" 2/sec", 500);
  glutAddMenuEntry(" 6/sec", 166);
  glutAddMenuEntry("10/sec", 100);
  glutAddMenuEntry("20/sec", 50);
  glutAddMenuEntry("30/sec", 33);
  glutCreateMenu(menu_select);
  glutAddMenuEntry("Toggle spinning", 1);
  glutAddMenuEntry("Toggle scaling", 2);
  glutAddMenuEntry("Switch mode", 3);
  glutAddSubMenu("Minimum frame rate", rate_menu);
  glutAddSubMenu("Minify modes", minify_menu);
  glutAddMenuEntry("Quit", 666);
  glutAttachMenu(GLUT_RIGHT_BUTTON);
  menu_select(3);
  glutMainLoop();
}
```