#include<windows.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

#include <GL/glut.h>

#define c3.14/180

#ifndef M\_PI

#define M\_PI 3.14159265

#endif

//#define PI 3.14

//#define TWO\_PI 2.0 \* PI

//#define RAD\_TO\_DEG 180.0 / PI

#define GREY 0

#define RED 1

#define GREEN 2

#define BLUE 3

#define CYAN 4

#define MAGENTA 5

#define YELLOW 6

#define BLACK 7

/\* Variable controlling various rendering modes. \*/

static int stencilReflection = 1, stencilShadow = 1, offsetShadow = 1;

static int renderShadow = 1, renderDinosaur = 1, renderReflection = 1;

static int linearFiltering = 0, useMipmaps = 0, useTexture = 1;

static int reportSpeed = 0;

static int animation = 1;

static GLboolean lightSwitch = GL\_TRUE;

static int directionalLight = 1;

static int forceExtension = 0;

/\* Time varying or user-controled variables. \*/

static float jump = 0.0;

static float lightAngle = 0.0, lightHeight = 20;

GLfloat angle = -150; /\* in degrees \*/

GLfloat angle2 = 30; /\* in degrees \*/

int moving, startx, starty;

int lightMoving = 0, lightStartX, lightStartY;

float djump = 0.0,ang=0,fxincr=0.1,fzincr,fx,fz,theta=0.01,temp,theta1;

enum {

MISSING, EXTENSION, ONE\_DOT\_ONE

};

int polygonOffsetVersion;

static GLdouble bodyWidth = 3.0;

/\* \*INDENT-OFF\* \*/

static GLfloat body[][2] = { {0, 3}, {1, 1}, {5, 1}, {8, 4}, {10, 4}, {11, 5},

{11, 11.5}, {13, 12}, {13, 13}, {10, 13.5}, {13, 14}, {13, 15}, {11, 16},

{8, 16}, {7, 15}, {7, 13}, {8, 12}, {7, 11}, {6, 6}, {4, 3}, {3, 2},

{1, 2} };

static GLfloat arm[][2] = { {8, 10}, {9, 9}, {10, 9}, {13, 8}, {14, 9}, {16, 9},

{15, 9.5}, {16, 10}, {15, 10}, {15.5, 11}, {14.5, 10}, {14, 11}, {14, 10},

{13, 9}, {11, 11}, {9, 11} };

static GLfloat leg[][2] = { {8, 6}, {8, 4}, {9, 3}, {9, 2}, {8, 1}, {8, 0.5}, {9, 0},

{12, 0}, {10, 1}, {10, 2}, {12, 4}, {11, 6}, {10, 7}, {9, 7} };

static GLfloat eye[][2] = { {8.75, 15}, {9, 14.7}, {9.6, 14.7}, {10.1, 15},

{9.6, 15.25}, {9, 15.25} };

static GLfloat lightPosition[4];

static GLfloat lightColor[] = {1.0, 1.0, 1.0, 1.0}; /\* green-tinted \*/

static GLfloat skinColor[] = {0.1, 1.0, 0.1, 1.0}, skinColor1[] = {0, 0, 1.0, 1.0}, skinColor2[]={0.5,0.5,0.5,0.5},skinColor3[]={1,0,0,1},eyeColor[] = {1.0, 0.2, 0.2, 1.0};

//static int moving = 1;

static float materialColor[8][4] =

{

{0.8, 0.8, 0.8, 1.0},

{0.8, 0.0, 0.0, 1.0},

{0.0, 0.8, 0.0, 1.0},

{0.0, 0.0, 0.8, 1.0},

{0.0, 0.8, 0.8, 1.0},

{0.8, 0.0, 0.8, 1.0},

{0.8, 0.8, 0.0, 1.0},

{0.0, 0.0, 0.0, 0.6},

};

static int useLighting = 1;

static int useQuads = 1;

static int useRGB = 1;

//Coordinates for the chassis of the car

float p[]={5.5,-2.5,1},q[]={5.5,-7.5,1},r[]={10.7,-7.5,1},s[]={10.7,-2.5,1};

float p1[]={10.7,-9,3},s1[]={12.7,-9,3},q1[]={10.7,-1,3},r1[]={12.7,-1,3};

float p2[]={0.5,-1,1},s2[]={5.5,-1,1},q2[]={0.5,-9,1},r2[]={5.5,-9,1};

float p3[]={-15,-6.5,1},q3[]={-15,-3.5,1},r3[]={0.5,-2.5,1},s3[]={0.5,-7.5,1};

float p4[]={-13,-6.5,1},q4[]={-13,-6.5,2.5},r4[]={0.5,-7.5,3.5},s4[]={0.5,-7.5,1};

float p5[]={-13,-3.5,1},q5[]={-13,-3.5,2.5},r5[]={0.5,-2.5,3.5},s5[]={0.5,-2.5,1};

float p6[]={5.5,-2.5,1},q6[]={5.5,-2.5,3.5},r6[]={10.7,-2.5,3.5},s6[]={10.7,-2.5,1};

float p7[]={5.5,-7.5,1},q7[]={5.5,-7.5,3.5},r7[]={10.7,-7.5,3.5},s7[]={10.7,-7.5,1};

float p8[]={5.5,-7.5,3.5},q8[]={10.7,-7.5,3.5},r8[]={10.7,-6,3.5},s8[]={5.5,-6,3.5};

float p9[]={5.5,-2.5,3.5},q9[]={5.5,-4,3.5},r9[]={10.7,-4,3.5},s9[]={10.7,-2.5,3.5};

float p10[]={5.5,-4,3.5},q10[]={10.7,-4,3.5},r10[]={10.7,-5,4.5},s10[]={5.5,-5,5.5};

float p11[]={5.5,-6,3.5},q11[]={10.7,-6,3.5},r11[]={10.7,-5,4.5},s11[]={5.5,-5,5.5};

float p12[]={10.7,-9,2},q12[]={10.7,-9,4},r12[]={12.7,-9,4},s12[]={12.7,-9,2};

float p13[]={10.7,-1,2},q13[]={10.7,-1,4},r13[]={12.7,-1,4},s13[]={12.7,-1,2};

float p14[]={0.5,-1,1},q14[]={0.5,-1,3},r14[]={5.5,-1,3},s14[]={5.5,-1,1};

float p15[]={0.5,-9,1},q15[]={0.5,-9,3},r15[]={5.5,-9,3},s15[]={5.5,-9,1};

float p16[]={0.5,-1,1},q16[]={0.5,-1,3},r16[]={0.5,-2.5,3.5},s16[]={0.5,-2.5,1};

float p17[]={0.5,-7.5,1},q17[]={0.5,-7.5,3.5},r17[]={0.5,-9,3},s17[]={0.5,-9,1};

float p18[]={5.5,-1,1},q18[]={5.5,-1,3},r18[]={5.5,-2.5,3.5},s18[]={5.5,-2.5,1};

float p19[]={5.5,-7.5,1},q19[]={5.5,-7.5,3.5},r19[]={5.5,-9,3},s19[]={5.5,-9,1};

float p20[]={10.7,-7.5,1},q20[]={10.7,-7.5,3.5},r20[]={10.7,-2.5,3.5},

s20[]={10.7,-2.5,1};

float p21[]={4,-2.5,3.5},q21[]={5.5,-2.5,3.5},r21[]={5.5,-7.5,3.5},s21[]={4,-7.5,3.5};

char KEY; //Variable that stores key pressed by user

//float angle; //Rotation angle for car

float carx=0,cary=570; //Variables that specify position of the car

int rot=0; //rotation angle for the wheels

int start=0;

/\* \*INDENT-ON\* \*/

/\* Nice floor texture tiling pattern. \*/

static void

setColor(int c)

{

if (useLighting) {

if (useRGB) {

glMaterialfv(GL\_FRONT\_AND\_BACK,

GL\_AMBIENT\_AND\_DIFFUSE, &materialColor[c][0]);

} else {

glMaterialfv(GL\_FRONT\_AND\_BACK,

GL\_COLOR\_INDEXES, &materialColor[c][0]);

}

} else {

if (useRGB) {

glColor4fv(&materialColor[c][0]);

} else {

glIndexf(materialColor[c][1]);

}

}

}

static void

drawCheck(int w, int h, int evenColor, int oddColor)

{

static int initialized = 0;

static int usedLighting = 0;

static GLuint checklist = 0;

if (!initialized || (usedLighting != useLighting)) {

static float square\_normal[4] =

{0.0, 0.0, 1.0, 0.0};

static float square[4][4];

int i, j;

if (!checklist) {

checklist = glGenLists(1);

}

glNewList(checklist, GL\_COMPILE\_AND\_EXECUTE);

if (useQuads) {

glNormal3fv(square\_normal);

glBegin(GL\_QUADS);

}

for (j = 0; j < h; ++j) {

for (i = 0; i < w; ++i) {

square[0][0] = -1.0 + 2.0 / w \* i;

square[0][1] = -1.0 + 2.0 / h \* (j + 1);

square[0][2] = 0.0;

square[0][3] = 1.0;

square[1][0] = -1.0 + 2.0 / w \* i;

square[1][1] = -1.0 + 2.0 / h \* j;

square[1][2] = 0.0;

square[1][3] = 1.0;

square[2][0] = -1.0 + 2.0 / w \* (i + 1);

square[2][1] = -1.0 + 2.0 / h \* j;

square[2][2] = 0.0;

square[2][3] = 1.0;

square[3][0] = -1.0 + 2.0 / w \* (i + 1);

square[3][1] = -1.0 + 2.0 / h \* (j + 1);

square[3][2] = 0.0;

square[3][3] = 1.0;

if (i & 1 ^ j & 1) {

setColor(oddColor);

} else {

setColor(evenColor);

}

if (!useQuads) {

glBegin(GL\_POLYGON);

}

glVertex4fv(&square[0][0]);

glVertex4fv(&square[1][0]);

glVertex4fv(&square[2][0]);

glVertex4fv(&square[3][0]);

if (!useQuads) {

glEnd();

}

}

}

if (useQuads) {

glEnd();

}

glEndList();

initialized = 1;

usedLighting = useLighting;

} else {

glCallList(checklist);

}

}

static char \*circles[] = {

"....xxxx........",

"..xxxxxxxx......",

".xxxxxxxxxx.....",

".xxx....xxx.....",

"xxx......xxx....",

"xxx......xxx....",

"xxx......xxx....",

"xxx......xxx....",

".xxx....xxx.....",

".xxxxxxxxxx.....",

"..xxxxxxxx......",

"....xxxx........",

"................",

"................",

"................",

"................",

};

static void

makeFloorTexture(void)

{

GLubyte floorTexture[16][16][3];

GLubyte \*loc;

int s, t;

/\* Setup RGB image for the texture. \*/

loc = (GLubyte\*) floorTexture;

for (t = 0; t < 16; t++) {

for (s = 0; s < 16; s++) {

if (circles[t][s] == 'x') {

/\* Nice green. \*/

loc[0] = 0x1f;

loc[1] = 0x8f;

loc[2] = 0x1f;

} else {

/\* Light gray. \*/

loc[0] = 0xaa;

loc[1] = 0xaa;

loc[2] = 0xaa;

}

loc += 3;

}

}

glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1);

if (useMipmaps) {

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER,

GL\_LINEAR\_MIPMAP\_LINEAR);

gluBuild2DMipmaps(GL\_TEXTURE\_2D, 3, 16, 16,

GL\_RGB, GL\_UNSIGNED\_BYTE, floorTexture);

} else {

if (linearFiltering) {

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

} else {

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);

}

glTexImage2D(GL\_TEXTURE\_2D, 0, 3, 16, 16, 0,

GL\_RGB, GL\_UNSIGNED\_BYTE, floorTexture);

}

}

enum {

X, Y, Z, W

};

enum {

A, B, C, D

};

/\* Create a matrix that will project the desired shadow. \*/

void

shadowMatrix(GLfloat shadowMat[4][4],

GLfloat groundplane[4],

GLfloat lightpos[4])

{

GLfloat dot;

/\* Find dot product between light position vector and ground plane normal. \*/

dot = groundplane[X] \* lightpos[X] +

groundplane[Y] \* lightpos[Y] +

groundplane[Z] \* lightpos[Z] +

groundplane[W] \* lightpos[W];

shadowMat[0][0] = dot - lightpos[X] \* groundplane[X];

shadowMat[1][0] = 0.f - lightpos[X] \* groundplane[Y];

shadowMat[2][0] = 0.f - lightpos[X] \* groundplane[Z];

shadowMat[3][0] = 0.f - lightpos[X] \* groundplane[W];

shadowMat[X][1] = 0.f - lightpos[Y] \* groundplane[X];

shadowMat[1][1] = dot - lightpos[Y] \* groundplane[Y];

shadowMat[2][1] = 0.f - lightpos[Y] \* groundplane[Z];

shadowMat[3][1] = 0.f - lightpos[Y] \* groundplane[W];

shadowMat[X][2] = 0.f - lightpos[Z] \* groundplane[X];

shadowMat[1][2] = 0.f - lightpos[Z] \* groundplane[Y];

shadowMat[2][2] = dot - lightpos[Z] \* groundplane[Z];

shadowMat[3][2] = 0.f - lightpos[Z] \* groundplane[W];

shadowMat[X][3] = 0.f - lightpos[W] \* groundplane[X];

shadowMat[1][3] = 0.f - lightpos[W] \* groundplane[Y];

shadowMat[2][3] = 0.f - lightpos[W] \* groundplane[Z];

shadowMat[3][3] = dot - lightpos[W] \* groundplane[W];

}

/\* Find the plane equation given 3 points. \*/

void

findPlane(GLfloat plane[4],

GLfloat v0[3], GLfloat v1[3], GLfloat v2[3])

{

GLfloat vec0[3], vec1[3];

/\* Need 2 vectors to find cross product. \*/

vec0[X] = v1[X] - v0[X];

vec0[Y] = v1[Y] - v0[Y];

vec0[Z] = v1[Z] - v0[Z];

vec1[X] = v2[X] - v0[X];

vec1[Y] = v2[Y] - v0[Y];

vec1[Z] = v2[Z] - v0[Z];

/\* find cross product to get A, B, and C of plane equation \*/

plane[A] = vec0[Y] \* vec1[Z] - vec0[Z] \* vec1[Y];

plane[B] = -(vec0[X] \* vec1[Z] - vec0[Z] \* vec1[X]);

plane[C] = vec0[X] \* vec1[Y] - vec0[Y] \* vec1[X];

plane[D] = -(plane[A] \* v0[X] + plane[B] \* v0[Y] + plane[C] \* v0[Z]);

}

void

extrudeSolidFromPolygon(GLfloat data[][2], unsigned int dataSize,

GLdouble thickness, GLuint side, GLuint edge, GLuint whole)

{

static GLUtriangulatorObj \*tobj = NULL;

GLdouble vertex[3], dx, dy, len;

int i;

int count = (int) (dataSize / (2 \* sizeof(GLfloat)));

if (tobj == NULL) {

tobj = gluNewTess(); /\* create and initialize a GLU

polygon tesselation object \*/

// gluTessCallback(tobj, GLU\_BEGIN, glBegin);

// gluTessCallback(tobj, GLU\_VERTEX, glVertex2fv); /\* semi-tricky \*/

gluTessCallback(tobj, GLU\_END, glEnd);

}

glNewList(side, GL\_COMPILE);

glShadeModel(GL\_SMOOTH); /\* smooth minimizes seeing

tessellation \*/

gluBeginPolygon(tobj);

for (i = 0; i < count; i++) {

vertex[0] = data[i][0];

vertex[1] = data[i][1];

vertex[2] = 0;

gluTessVertex(tobj, vertex, data[i]);

}

gluEndPolygon(tobj);

glEndList();

glNewList(edge, GL\_COMPILE);

glShadeModel(GL\_FLAT); /\* flat shade keeps angular hands

from being "smoothed" \*/

glBegin(GL\_QUAD\_STRIP);

for (i = 0; i <= count; i++) {

/\* mod function handles closing the edge \*/

glVertex3f(data[i % count][0], data[i % count][1], 0.0);

glVertex3f(data[i % count][0], data[i % count][1], thickness);

/\* Calculate a unit normal by dividing by Euclidean

distance. We \* could be lazy and use

glEnable(GL\_NORMALIZE) so we could pass in \* arbitrary

normals for a very slight performance hit. \*/

dx = data[(i + 1) % count][1] - data[i % count][1];

dy = data[i % count][0] - data[(i + 1) % count][0];

len = sqrt(dx \* dx + dy \* dy);

glNormal3f(dx / len, dy / len, 0.0);

}

glEnd();

glEndList();

glNewList(whole, GL\_COMPILE);

glFrontFace(GL\_CW);

glCallList(edge);

glNormal3f(0.0, 0.0, -1.0); /\* constant normal for side \*/

glCallList(side);

glPushMatrix();

glTranslatef(0.0, 0.0, thickness);

glFrontFace(GL\_CCW);

glNormal3f(0.0, 0.0, 1.0); /\* opposite normal for other side \*/

glCallList(side);

glPopMatrix();

glEndList();

}

/\* Enumerants for refering to display lists. \*/

typedef enum {

RESERVED, BODY\_SIDE, BODY\_EDGE, BODY\_WHOLE, ARM\_SIDE, ARM\_EDGE, ARM\_WHOLE,

LEG\_SIDE, LEG\_EDGE, LEG\_WHOLE, EYE\_SIDE, EYE\_EDGE, EYE\_WHOLE

} displayLists;

static void

makeDinosaur(void)

{

extrudeSolidFromPolygon(body, sizeof(body), bodyWidth,

BODY\_SIDE, BODY\_EDGE, BODY\_WHOLE);

extrudeSolidFromPolygon(arm, sizeof(arm), bodyWidth / 4,

ARM\_SIDE, ARM\_EDGE, ARM\_WHOLE);

extrudeSolidFromPolygon(leg, sizeof(leg), bodyWidth / 2,

LEG\_SIDE, LEG\_EDGE, LEG\_WHOLE);

extrudeSolidFromPolygon(eye, sizeof(eye), bodyWidth + 0.2,

EYE\_SIDE, EYE\_EDGE, EYE\_WHOLE);

}

static void

drawDinosaur(void)

{

glPushMatrix();

if(fxincr!=0)

theta1=(atan(fzincr/fxincr)\*180)/3.141;

else if(fzincr>0)

theta1=-90.0;

else theta1=90.0;

if(fxincr>0&&fzincr<0)

{

theta1=-theta1;

}

else if(fxincr<0&&fzincr<0)

{

theta1=180-theta1;

}

else if(fxincr<0&&fzincr>0)

{

theta1=-180-theta1;

}else if(fxincr>0&&fzincr>0)

{

theta1=-theta1;

}

/\* Translate the dinosaur to be at (0,8,0). \*/

glTranslatef(fx-3, jump, fz);

glRotatef(theta1,0,1,0);

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, skinColor);

glCallList(BODY\_WHOLE);

glPushMatrix();

glTranslatef(0.0, 0.0, bodyWidth);

glCallList(ARM\_WHOLE);

glCallList(LEG\_WHOLE);

glTranslatef(0.0, 0.0, -bodyWidth - bodyWidth / 4);

glCallList(ARM\_WHOLE);

glTranslatef(0.0, 0.0, -bodyWidth / 4);

glCallList(LEG\_WHOLE);

glTranslatef(0.0, 0.0, bodyWidth / 2 - 0.1);

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, eyeColor);

glCallList(EYE\_WHOLE);

glPopMatrix();

glPopMatrix();

}

static void

makeDinosaur1(void)

{

extrudeSolidFromPolygon(body, sizeof(body), bodyWidth,

BODY\_SIDE, BODY\_EDGE, BODY\_WHOLE);

extrudeSolidFromPolygon(arm, sizeof(arm), bodyWidth / 2,

ARM\_SIDE, ARM\_EDGE, ARM\_WHOLE);

extrudeSolidFromPolygon(leg, sizeof(leg), bodyWidth / 4,

LEG\_SIDE, LEG\_EDGE, LEG\_WHOLE);

extrudeSolidFromPolygon(eye, sizeof(eye), bodyWidth + 0.1,

EYE\_SIDE, EYE\_EDGE, EYE\_WHOLE);

}

static void

drawDinosaur1(void)

{

glPushMatrix();

if(fxincr!=0)

theta1=(atan(fzincr/fxincr)\*180)/3.141;

else if(fzincr>0)

theta1=-90.0;

else theta1=90.0;

if(fxincr>0&&fzincr<0)

{

theta1=-theta1;

}

else if(fxincr<0&&fzincr<0)

{

theta1=180-theta1;

}

else if(fxincr<0&&fzincr>0)

{

theta1=-180-theta1;

}else if(fxincr>0&&fzincr>0)

{

theta1=-theta1;

}

/\* Translate the dinosaur to be at (0,8,0). \*/

glTranslatef(fx-3, jump, fz);

glRotatef(theta1,0,1,0);

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, skinColor);

glCallList(BODY\_WHOLE);

glPushMatrix();

glTranslatef(0.0, 0.0, bodyWidth);

glCallList(ARM\_WHOLE);

glCallList(LEG\_WHOLE);

glTranslatef(0.0, 0.0, -bodyWidth - bodyWidth / 4);

glCallList(ARM\_WHOLE);

glTranslatef(0.0, 0.0, -bodyWidth / 4);

glCallList(LEG\_WHOLE);

glTranslatef(0.0, 0.0, bodyWidth / 2 - 0.1);

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, eyeColor);

glCallList(EYE\_WHOLE);

glPopMatrix();

glPopMatrix();

}

void cylinder(float r,float l)

{

float x,y,z; int d,c;

glBegin(GL\_QUAD\_STRIP);

for( d=0;d<=362;d+=1)

{

x=r\*cos(c\*d);

z=r\*sin(c\*d);

y=0;

glVertex3f(x,y,z);

y=l;

glVertex3f(x,y,z);

}

glEnd();

}

void alloy(float R1,float R2)

{

float X,Y,Z;int y,c;

glColor3f(1.0,0.5,0.4);

glBegin(GL\_QUAD\_STRIP);

for(y=0;y<=361;y+=1)

{

X=R1\*cos(c\*y);

Z=R1\*sin(c\*y);

Y=0;

glVertex3f(X,Y,Z);

X=R2\*cos(c\*y);

Z=R2\*sin(c\*y);

Y=0;

glVertex3f(X,Y,Z);

}

glEnd(); }

//Function to draw the spokes of the wheel

void actall(float R1,float R2)

{

float X,Y,Z; int i,c;

glBegin(GL\_QUADS);

for(i=0;i<=361;i+=120)

{

glColor3f(1.0,0.8,0);

X=R1\*cos(c\*i);

Y=0;

Z=R1\*sin(c\*i);

glVertex3f(X,Y,Z);

X=R1\*cos(c\*(i+30));

Y=0;

Z=R1\*sin(c\*(i+30));

glVertex3f(X,Y,Z);

X=R2\*cos(c\*(i+30));

Y=0;

Z=R2\*sin(c\*(i+30));

glVertex3f(X,Y,Z);

X=R2\*cos(c\*i);

Y=0;

Z=R2\*sin(c\*i);

glVertex3f(X,Y,Z);

}

glEnd();

}

//Function to draw a circle

void circle(float R)

{

float X,Y,Z;int i,c;

glBegin(GL\_POLYGON);

for(i=0;i<=360;i++)

{

X=R\*cos(c\*i);

Z=R\*sin(c\*i);

Y=0;

glVertex3f(X,Y,Z);

}

glEnd();

}

//Function to draw a quadrilateral

void rect(float p[],float q[],float r[],float s[])

{

glBegin(GL\_POLYGON);

glVertex3fv(p);

glVertex3fv(q);

glVertex3fv(r);

glVertex3fv(s);

glEnd();

}

//Function to generate car driver

void driver()

{

//Legs

glPushMatrix();

glTranslatef(3,-3.5,1.5);

glRotatef(90,0,0,1);

cylinder(0.4,3);

glPopMatrix();

glPushMatrix();

glTranslatef(3,-6.5,1.5);

glRotatef(90,0,0,1);

cylinder(0.4,3);

glPopMatrix();

//Hands

glPushMatrix();

glTranslatef(3,-3.5,2.5);

glRotatef(90,0,0,1);

cylinder(0.4,3);

glPopMatrix();

glPushMatrix();

glTranslatef(3,-6.5,2.5);

glRotatef(90,0,0,1);

cylinder(0.4,3);

glPopMatrix();

//Head

glPushMatrix();

glTranslatef(3,-5,4);

glutSolidSphere (1.0, 20, 16);

glPopMatrix();

//Body

glPushMatrix();

glTranslatef(3,-5,1);

glRotatef(90,1,0,0);

cylinder(1,2);

glPopMatrix();

//Circle

glPushMatrix();

glTranslatef(3,-5,3);

glRotatef(90,1,0,0);

circle(1);

glPopMatrix();

}

void tri(float a[],float b[],float z[])

{

glBegin(GL\_TRIANGLES);

glVertex3fv(a);

glVertex3fv(b);

glVertex3fv(z);

glEnd();

}

//Function that has calls to other functions to generate wheels along with axle

void wheels()

{

//axle

glColor3f(1,0,0);

cylinder(0.4,9);

//1st Wheel

cylinder(2,2);

alloy(2,1.4);

actall(1.4,0.8);

glColor3f(1,0,0);

circle(0.8);

glPushMatrix();

glTranslatef(0,2,0);

alloy(2,1.4);

actall(1.4,0.8);

circle(0.8);

glPopMatrix();

//2nd Wheel

glPushMatrix();

glTranslatef(0,8,0);

glColor3f(1,0,0);

cylinder(2,2);

alloy(2,1.4);

actall(1.4,0.8);

glColor3f(1,0,0);

circle(0.8);

glPopMatrix();

glPushMatrix();

glTranslatef(0,10,0);

actall(1.4,0.8);

alloy(2,1.4);

glColor3f(1,0,0);

circle(0.8);

glPopMatrix();

}

//Function that generates the chassis of the car

void chassis()

{

//Parameters For glMaterialfv() function

GLfloat specular[] = { 0.7, 0.7, 0.7, 1.0 };

GLfloat ambient[]={1,1,1,1},diffuse[]={0.7,0.7,0.7,1};

GLfloat full\_shininess[]={50.0};

//Material Properties

glMaterialfv(GL\_FRONT,GL\_AMBIENT,ambient);

glMaterialfv(GL\_FRONT,GL\_SPECULAR,specular);

glMaterialfv(GL\_FRONT,GL\_DIFFUSE,diffuse);

glMaterialfv(GL\_FRONT,GL\_SHININESS, full\_shininess);

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, skinColor3);

rect(p,q,r,s);

rect(p2,q2,r2,s2);

rect(p3,q3,r3,s3);

rect(p4,q4,r4,s4);

rect(p5,q5,r5,s5);

rect(q5,q4,r4,r5);

rect(p6,q6,r6,s6);

rect(p7,q7,r7,s7);

rect(p8,q8,r8,s8);

rect(p9,q9,r9,s9);

glColor3f(1.0,0.6,1.0);

rect(p1,q1,r1,s1);

rect(q5,q4,p3,q3);

tri(p4,q4,p3);

tri(p5,q5,q3);

rect(p10,q10,r10,s10);

rect(p11,q11,r11,s11);

rect(r16,r18,q18,q16);

rect(q17,q19,r19,r17);

rect(p21,q21,r21,s21);

glColor3f(1,0,0);

rect(p12,q12,r12,s12);

rect(p13,q13,r13,s13); rect(p14,q14,r14,s14);

rect(p15,q15,r15,s15);

rect(p16,q16,r16,s16);

rect(p17,q17,r17,s17);

rect(p18,q18,r18,s18);

rect(p19,q19,r19,s19);

rect(r18,q19,p19,s18);

rect(p20,q20,r20,s20);

}

//Function that that has function calls to chassis(),tyrea(),

//tyreb(),driver() to generate the car with wheels rotating

void car()

{

glPushMatrix();

glRotatef(180,0,0,1);

chassis();

glPushMatrix();

glTranslatef(8,-10,1);

glRotatef(rot,0,1,0);

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, skinColor2);

wheels();

glPopMatrix();

glPushMatrix();

glTranslatef(-12,-10,1);

glRotatef(rot,0,1,0);

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, skinColor2);

wheels();

glPopMatrix();

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, skinColor1);

driver();

rot+=90;

if(rot>360) rot-=360;

glPopMatrix();

}

static GLfloat floorVertices[4][3] = {

{ -20.0, 0.0, 20.0 },

{ 20.0, 0.0, 20.0 },

{ 20.0, 0.0, -20.0 },

{ -20.0, 0.0, -20.0 },

};

/\* Draw a floor (possibly textured). \*/

static void

drawFloor(void)

{

glDisable(GL\_LIGHTING);

if (useTexture) {

glEnable(GL\_TEXTURE\_2D);

}

glBegin(GL\_QUADS);

glTexCoord2f(0.0, 0.0);

glVertex3fv(floorVertices[0]);

glTexCoord2f(0.0, 16.0);

glVertex3fv(floorVertices[1]);

glTexCoord2f(16.0, 16.0);

glVertex3fv(floorVertices[2]);

glTexCoord2f(16.0, 0.0);

glVertex3fv(floorVertices[3]);

glEnd();

if (useTexture) {

glDisable(GL\_TEXTURE\_2D);

}

glEnable(GL\_LIGHTING);

}

static GLfloat floorPlane[4];

static GLfloat floorShadow[4][4];

static void

redraw(void)

{

int start, end;

if (reportSpeed) {

start = glutGet(GLUT\_ELAPSED\_TIME);

}

/\* Clear; default stencil clears to zero. \*/

if ((stencilReflection && renderReflection) || (stencilShadow && renderShadow)) {

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT | GL\_STENCIL\_BUFFER\_BIT);

} else {

/\* Avoid clearing stencil when not using it. \*/

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

}

/\* Reposition the light source. \*/

lightPosition[0] = 12\*cos(lightAngle);

lightPosition[1] = lightHeight;

lightPosition[2] = 12\*sin(lightAngle);

if (directionalLight) {

lightPosition[3] = 0.0;

} else {

lightPosition[3] = 1.0;

}

shadowMatrix(floorShadow, floorPlane, lightPosition);

glPushMatrix();

/\* Perform scene rotations based on user mouse input. \*/

glRotatef(angle2, 1.0, 0.0, 0.0);

glRotatef(angle, 0.0, 1.0, 0.0);

/\* Tell GL new light source position. \*/

glLightfv(GL\_LIGHT0, GL\_POSITION, lightPosition);

if (renderReflection) {

if (stencilReflection) {

/\* We can eliminate the visual "artifact" of seeing the "flipped"

dinosaur underneath the floor by using stencil. The idea is

draw the floor without color or depth update but so that

a stencil value of one is where the floor will be. Later when

rendering the dinosaur reflection, we will only update pixels

with a stencil value of 1 to make sure the reflection only

lives on the floor, not below the floor. \*/

/\* Don't update color or depth. \*/

glDisable(GL\_DEPTH\_TEST);

glColorMask(GL\_FALSE, GL\_FALSE, GL\_FALSE, GL\_FALSE);

/\* Draw 1 into the stencil buffer. \*/

glEnable(GL\_STENCIL\_TEST);

glStencilOp(GL\_REPLACE, GL\_REPLACE, GL\_REPLACE);

glStencilFunc(GL\_ALWAYS, 1, 0xffffffff);

/\* Now render floor; floor pixels just get their stencil set to 1. \*/

drawFloor();

/\* Re-enable update of color and depth. \*/

glColorMask(GL\_TRUE, GL\_TRUE, GL\_TRUE, GL\_TRUE);

glEnable(GL\_DEPTH\_TEST);

/\* Now, only render where stencil is set to 1. \*/

glStencilFunc(GL\_EQUAL, 1, 0xffffffff); /\* draw if ==1 \*/

glStencilOp(GL\_KEEP, GL\_KEEP, GL\_KEEP);

}

glPushMatrix();

/\* The critical reflection step: Reflect dinosaur through the floor

(the Y=0 plane) to make a relection. \*/

glScalef(1.0, -1.0, 1.0);

/\* Reflect the light position. \*/

glLightfv(GL\_LIGHT0, GL\_POSITION, lightPosition);

/\* To avoid our normals getting reversed and hence botched lighting

on the reflection, turn on normalize. \*/

glEnable(GL\_NORMALIZE);

glCullFace(GL\_FRONT);

/\* Draw the reflected dinosaur. \*/

glPushMatrix();

glRotatef(-90,1,0,0);

glTranslatef(0,0,2);

car();

//drawDinosaur();

glPopMatrix();

glPushMatrix();

glTranslatef(-5,0,5);

drawDinosaur1();

glPopMatrix();

/\* Disable noramlize again and re-enable back face culling. \*/

glDisable(GL\_NORMALIZE);

glCullFace(GL\_BACK);

glPopMatrix();

/\* Switch back to the unreflected light position. \*/

glLightfv(GL\_LIGHT0, GL\_POSITION, lightPosition);

if (stencilReflection) {

glDisable(GL\_STENCIL\_TEST);

}

}

/\* Back face culling will get used to only draw either the top or the

bottom floor. This let's us get a floor with two distinct

appearances. The top floor surface is reflective and kind of red.

The bottom floor surface is not reflective and blue. \*/

/\* Draw "bottom" of floor in blue. \*/

glFrontFace(GL\_CW); /\* Switch face orientation. \*/

glColor4f(0.1, 0.1, 0.7, 1.0);

drawFloor();

glFrontFace(GL\_CCW);

if (renderShadow) {

if (stencilShadow) {

/\* Draw the floor with stencil value 3. This helps us only

draw the shadow once per floor pixel (and only on the

floor pixels). \*/

glEnable(GL\_STENCIL\_TEST);

glStencilFunc(GL\_ALWAYS, 3, 0xffffffff);

glStencilOp(GL\_KEEP, GL\_KEEP, GL\_REPLACE);

}

}

/\* Draw "top" of floor. Use blending to blend in reflection. \*/

glEnable(GL\_BLEND);

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

glColor4f(0.7, 0.0, 0.0, 0.3);

glColor4f(1.0, 1.0, 1.0, 0.3);

drawFloor();

glDisable(GL\_BLEND);

if (renderDinosaur) {

/\* Draw "actual" dinosaur, not its reflection. \*/

glPushMatrix();

glRotatef(-90,1,0,0);

glTranslatef(0,0,2);

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, skinColor2);

car();

// drawDinosaur();

glPopMatrix();

glPushMatrix();

glTranslatef(-5,0,5);

drawDinosaur1();

glPopMatrix();

}

if (renderShadow) {

/\* Render the projected shadow. \*/

if (stencilShadow) {

/\* Now, only render where stencil is set above 2 (ie, 3 where

the top floor is). Update stencil with 2 where the shadow

gets drawn so we don't redraw (and accidently reblend) the

shadow). \*/

glStencilFunc(GL\_LESS, 2, 0xffffffff); /\* draw if ==1 \*/

glStencilOp(GL\_REPLACE, GL\_REPLACE, GL\_REPLACE);

}

/\* To eliminate depth buffer artifacts, we use polygon offset

to raise the depth of the projected shadow slightly so

that it does not depth buffer alias with the floor. \*/

if (offsetShadow) {

switch (polygonOffsetVersion) {

case EXTENSION:

#ifdef GL\_EXT\_polygon\_offset

glEnable(GL\_POLYGON\_OFFSET\_EXT);

break;

#endif

#ifdef GL\_VERSION\_1\_1

case ONE\_DOT\_ONE:

glEnable(GL\_POLYGON\_OFFSET\_FILL);

break;

#endif

case MISSING:

/\* Oh well. \*/

break;

}

}

/\* Render 50% black shadow color on top of whatever the

floor appareance is. \*/

glEnable(GL\_BLEND);

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

glDisable(GL\_LIGHTING); /\* Force the 50% black. \*/

glColor4f(0.0, 0.0, 0.0, 0.5);

glPushMatrix();

/\* Project the shadow. \*/

glMultMatrixf((GLfloat \*) floorShadow);

// drawDinosaur();

car();

drawDinosaur1();

glPopMatrix();

glDisable(GL\_BLEND);

glEnable(GL\_LIGHTING);

if (offsetShadow) {

switch (polygonOffsetVersion) {

#ifdef GL\_EXT\_polygon\_offset

case EXTENSION:

glDisable(GL\_POLYGON\_OFFSET\_EXT);

break;

#endif

#ifdef GL\_VERSION\_1\_1

case ONE\_DOT\_ONE:

glDisable(GL\_POLYGON\_OFFSET\_FILL);

break;

#endif

case MISSING:

/\* Oh well. \*/

break;

}

}

if (stencilShadow) {

glDisable(GL\_STENCIL\_TEST);

}

}

glPushMatrix();

glDisable(GL\_LIGHTING);

glColor3f(1.0, 1.0, 0.0); //arrow color

if (directionalLight) {

/\* Draw an arrowhead. \*/

/\* glDisable(GL\_CULL\_FACE);

glTranslatef(lightPosition[0], lightPosition[1], lightPosition[2]);

glRotatef(lightAngle \* -180.0 / M\_PI, 0, 1, 0);

glRotatef(atan(lightHeight/12) \* 180.0 / M\_PI, 0, 0, 1);

glBegin(GL\_TRIANGLE\_FAN);

glVertex3f(0, 0, 0);

glVertex3f(2, 1, 1);

glVertex3f(2, -1, 1);

glVertex3f(2, -1, -1);

glVertex3f(2, 1, -1);

glVertex3f(2, 1, 1);

glEnd();

/\* Draw a white line from light direction. \*/

/\*glColor3f(1.0, 1.0, 1.0);

glBegin(GL\_LINES);

glVertex3f(0, 0, 0);

glVertex3f(5, 0, 0);

glEnd();

glEnable(GL\_CULL\_FACE);

} else {

/\* Draw a yellow ball at the light source. \*/

glTranslatef(lightPosition[0], lightPosition[1], lightPosition[2]);

// glutSolidSphere(1.0, 5, 5);

}

glEnable(GL\_LIGHTING);

glPopMatrix();

glPopMatrix();

glPushMatrix();

glScalef(-20,-20,-20);

glTranslatef(0,-0.3,0.7);

drawCheck(6,6,RED,YELLOW);

glPopMatrix();

if (reportSpeed) {

glFinish();

end = glutGet(GLUT\_ELAPSED\_TIME);

printf("Speed %.3g frames/sec (%d ms)\n", 1000.0/(end-start), end-start);

}

glutSwapBuffers();

}

/\* ARGSUSED2 \*/

static void

mouse(int button, int state, int x, int y)

{

if (button == GLUT\_LEFT\_BUTTON) {

if (state == GLUT\_DOWN) {

moving = 1;

startx = x;

starty = y;

}

if (state == GLUT\_UP) {

moving = 0;

}

}

if (button == GLUT\_MIDDLE\_BUTTON) {

if (state == GLUT\_DOWN) {

lightMoving = 1;

lightStartX = x;

lightStartY = y;

}

if (state == GLUT\_UP) {

lightMoving = 0;

}

}

}

/\* ARGSUSED1 \*/

static void

motion(int x, int y)

{

if (moving) {

angle = angle + (x - startx);

angle2 = angle2 + (y - starty);

startx = x;

starty = y;

glutPostRedisplay();

}

if (lightMoving) {

lightAngle += (x - lightStartX)/40.0;

lightHeight += (lightStartY - y)/20.0;

lightStartX = x;

lightStartY = y;

glutPostRedisplay();

}

}

/\* Advance time varying state when idle callback registered. \*/

static void

idle(void)

{

static float time = 0.0;

time = glutGet(GLUT\_ELAPSED\_TIME) / 500.0;

jump = 4.0 \* fabs(sin(time)\*0.5);

if (!lightMoving) {

lightAngle += 0.03;

}

glutPostRedisplay();

}

enum {

M\_NONE, M\_MOTION, M\_LIGHT, M\_TEXTURE, M\_SHADOWS, M\_REFLECTION, M\_DINOSAUR,

M\_STENCIL\_REFLECTION, M\_STENCIL\_SHADOW, M\_OFFSET\_SHADOW,

M\_POSITIONAL, M\_DIRECTIONAL, M\_PERFORMANCE

};

static void

controlLights(int value)

{

switch (value) {

case M\_NONE:

return;

case M\_MOTION:

animation = 1 - animation;

if (animation) {

glutIdleFunc(idle);

} else {

glutIdleFunc(NULL);

}

break;

case M\_LIGHT:

lightSwitch = !lightSwitch;

if (lightSwitch) {

glEnable(GL\_LIGHT0);

} else {

glDisable(GL\_LIGHT0);

}

break;

case M\_TEXTURE:

useTexture = !useTexture;

break;

case M\_SHADOWS:

renderShadow = 1 - renderShadow;

break;

case M\_REFLECTION:

renderReflection = 1 - renderReflection;

break;

case M\_DINOSAUR:

renderDinosaur = 1 - renderDinosaur;

break;

case M\_STENCIL\_REFLECTION:

stencilReflection = 1 - stencilReflection;

break;

case M\_STENCIL\_SHADOW:

stencilShadow = 1 - stencilShadow;

break;

case M\_OFFSET\_SHADOW:

offsetShadow = 1 - offsetShadow;

break;

case M\_POSITIONAL:

directionalLight = 0;

break;

case M\_DIRECTIONAL:

directionalLight = 1;

break;

case M\_PERFORMANCE:

reportSpeed = 1 - reportSpeed;

break;

}

glutPostRedisplay();

}

/\* When not visible, stop animating. Restart when visible again. \*/

static void

visible(int vis)

{

if (vis == GLUT\_VISIBLE) {

if (animation)

glutIdleFunc(idle);

} else {

if (!animation)

glutIdleFunc(NULL);

}

}

/\* Press any key to redraw; good when motion stopped and

performance reporting on. \*/

/\* ARGSUSED \*/

static void keys(unsigned char c, int x, int y)

{

if (c == 27)

exit(0); /\* IRIS GLism, Escape quits. \*/

}

// glutPostRedisplay();

/\* Press any key to redraw; good when motion stopped and

performance reporting on. \*/

/\* ARGSUSED \*/

static

void special(int key, int x, int y)

{

switch (key)

{

case GLUT\_KEY\_LEFT :/\*temp=fxincr;

fxincr=fxincr\*cos(theta)+fzincr\*sin(theta);

fzincr=-temp\*sin(theta)+fzincr\*cos(theta);

fx+=fxincr;

fz+=fzincr;

\*/ temp=fxincr;

fxincr=fxincr\*cos(theta)+fzincr\*sin(theta);

fzincr=-temp\*sin(theta)+fzincr\*cos(theta);

break;

case GLUT\_KEY\_RIGHT :

/\*temp=fxincr;

fxincr=fxincr\*cos(-theta)+fzincr\*sin(-theta);

fzincr=-temp\*sin(-theta)+fzincr\*cos(-theta);

fx+=fxincr;

fz+=fzincr;

\*/ temp=fxincr;

fxincr=fxincr\*cos(-theta)+fzincr\*sin(-theta);

fzincr=-temp\*sin(-theta)+fzincr\*cos(-theta);

break;

case GLUT\_KEY\_UP :fx+=fxincr;

fz+=fzincr;break;

case GLUT\_KEY\_DOWN :fx-=fxincr;

fz-=fzincr; break;

}

glutPostRedisplay();

}

static int

supportsOneDotOne(void)

{

const char \*version;

int major, minor;

version = (char \*) glGetString(GL\_VERSION);

if (sscanf(version, "%d.%d", &major, &minor) == 2)

return major >= 1 && minor >= 1;

return 0; /\* OpenGL version string malformed! \*/

}

int

main(int argc, char \*\*argv)

{

int i;

glutInit(&argc, argv);

for (i=1; i<argc; i++) {

if (!strcmp("-linear", argv[i])) {

linearFiltering = 1;

} else if (!strcmp("-mipmap", argv[i])) {

useMipmaps = 1;

} else if (!strcmp("-ext", argv[i])) {

forceExtension = 1;

}

}

glutInitDisplayMode(GLUT\_RGB | GLUT\_DOUBLE | GLUT\_DEPTH | GLUT\_STENCIL | GLUT\_MULTISAMPLE);

#if 1

/\* In GLUT 4.0, you'll be able to do this an be sure to

get 2 bits of stencil if the machine has it for you. \*/

glutInitDisplayString("samples stencil>=2 rgb double depth");

#endif

glutCreateWindow("Shadowy Leapin' Lizards");

if (glutGet(GLUT\_WINDOW\_STENCIL\_SIZE) <= 1) {

printf("dinoshade: Sorry, I need at least 2 bits of stencil.\n");

exit(1);

}

/\* Register GLUT callbacks. \*/

glutDisplayFunc(redraw);

glutMouseFunc(mouse);

glutMotionFunc(motion);

glutVisibilityFunc(visible);

glutKeyboardFunc(keys);

glutSpecialFunc(special);

glutCreateMenu(controlLights);

glutAddMenuEntry("Toggle motion", M\_MOTION);

glutAddMenuEntry("-----------------------", M\_NONE);

glutAddMenuEntry("Toggle light", M\_LIGHT);

glutAddMenuEntry("Toggle texture", M\_TEXTURE);

glutAddMenuEntry("Toggle shadows", M\_SHADOWS);

glutAddMenuEntry("Toggle reflection", M\_REFLECTION);

glutAddMenuEntry("Toggle dinosaur", M\_DINOSAUR);

glutAddMenuEntry("-----------------------", M\_NONE);

glutAddMenuEntry("Toggle reflection stenciling", M\_STENCIL\_REFLECTION);

glutAddMenuEntry("Toggle shadow stenciling", M\_STENCIL\_SHADOW);

glutAddMenuEntry("Toggle shadow offset", M\_OFFSET\_SHADOW);

glutAddMenuEntry("----------------------", M\_NONE);

glutAddMenuEntry("Positional light", M\_POSITIONAL);

glutAddMenuEntry("Directional light", M\_DIRECTIONAL);

glutAddMenuEntry("-----------------------", M\_NONE);

glutAddMenuEntry("Toggle performance", M\_PERFORMANCE);

glutAttachMenu(GLUT\_RIGHT\_BUTTON);

makeDinosaur();

#ifdef GL\_VERSION\_1\_1

if (supportsOneDotOne() && !forceExtension) {

polygonOffsetVersion = ONE\_DOT\_ONE;

glPolygonOffset(-2.0, -1.0);

} else

#endif

{

#ifdef GL\_EXT\_polygon\_offset

/\* check for the polygon offset extension \*/

if (glutExtensionSupported("GL\_EXT\_polygon\_offset")) {

polygonOffsetVersion = EXTENSION;

glPolygonOffsetEXT(-0.1, -0.002);

} else

#endif

{

polygonOffsetVersion = MISSING;

printf("\ndinoshine: Missing polygon offset.\n");

printf(" Expect shadow depth aliasing artifacts.\n\n");

}

}

glEnable(GL\_CULL\_FACE);

glEnable(GL\_DEPTH\_TEST);

glEnable(GL\_TEXTURE\_2D);

glLineWidth(3.0);

glMatrixMode(GL\_PROJECTION);

gluPerspective( /\* field of view in degree \*/ 40.0,

/\* aspect ratio \*/ 1.0,

/\* Z near \*/ 20.0, /\* Z far \*/ 100.0);

glMatrixMode(GL\_MODELVIEW);

gluLookAt(0.0, 8.0, 60.0, /\* eye is at (0,8,60) \*/

0.0, 8.0, 0.0, /\* center is at (0,8,0) \*/

0.0, 1.0, 0.); /\* up is in postivie Y direction \*/

glLightModeli(GL\_LIGHT\_MODEL\_LOCAL\_VIEWER, 1);

glLightfv(GL\_LIGHT0, GL\_DIFFUSE, lightColor);

glLightf(GL\_LIGHT0, GL\_CONSTANT\_ATTENUATION, 0.1);

glLightf(GL\_LIGHT0, GL\_LINEAR\_ATTENUATION, 0.05);

glEnable(GL\_LIGHT0);

glEnable(GL\_LIGHTING);

makeFloorTexture();

/\* Setup floor plane for projected shadow calculations. \*/

// findPlane(floorPlane, floorVertices[1], floorVertices[2], floorVertices[3]);

glutMainLoop();

return 0; /\* ANSI C requires main to return int. \*/

}