|  |  |
| --- | --- |
| **Gerb-BMSTU_01** | **Министерство науки и высшего образования Российской Федерации**  Калужский филиал  федерального государственного бюджетного  образовательного учреждения высшего образования  ***«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»***  ***(КФ МГТУ им. Н.Э. Баумана)*** |

**ФАКУЛЬТЕТ** ***ИУК «Информатика и управление»***

**КАФЕДРА** \_\_***ИУК4 «Программное обеспечение ЭВМ, информационные технологии»***

**ЛАБОРАТОРНАЯ РАБОТА №5**

**«Основы наложения текстур в OpenGL»**

**ДИСЦИПЛИНА: «Компьютерная графика»**

|  |  |  |
| --- | --- | --- |
| Выполнил: студент гр. ИУК4-42Б | | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ( Карельский М.К. )  (Подпись) |
| Проверил: | | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ( Глебов С.А. )  (Подпись) |
| Дата сдачи (защиты):  Результаты сдачи (защиты): | | |
|  | - Балльная оценка:  - Оценка: | |

Калуга, 2022

**Цель:** формирование практических навыков по работе с текстурами средствами OpenGL, их наложению на освещенные объекты подверженные проекционному сокращению.

**Задачи:**

* понимать принципы наложения растровых изображений на геометрические объекты,
* уметь реализовывать наложение текстур с использованием возможностей OpenGL,
* научиться использовать наложение множественных текстур,
* уметь создавать фотореалистичные сцены (корректное освещение), на которых присутствуют текстурированные геометрические объекты

**Вариант 7**

**Задание:**

1. Для **Листинга 1** на **параллелепипед** наложить произвольную текстуру из файла в формате tga.
2. Используя **Листинг 2**, для **трех цилиндров, образующих букву «П»**, реализовать вращение и показать действие одномерной текстуры. Цвет меняется плавно от темного до светлого оттенка. Количество градаций цвета не менее 12.
3. Используя **Листинг 3** создать коридор. Для каждой грани использовать свою собственную текстуру т.е. отдельный файл в формате tga.
4. Для **Листинга 4**, используя листинг 6 (листинг 7) из лабораторной работы №3, для исходного тора, сферы и поверхности реализовать наложение текстуры с зеркальными бликами (текстуры: **тор – камень, сфера – листва, поверхность – трава**). Для своих собственных объектов добавить произвольные текстуры из набора: капли на стекле, орнамент, арбуз, каменная стена, кирпичная стена, асфальт, треснутая земля, листва зеленая. реализовать следующие изменения согласно варианту.

**Листинг 1:**

#include "glew.h"

#include "glut.h"

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

namespace Task\_5\_1

{

#pragma warning(disable : 4996)

// Rotation amounts

static GLfloat xRot = 0.0f;

static GLfloat yRot = 0.0f;

// Define targa header.

#pragma pack(1)

typedef struct

{

GLbyte identsize; // Size of ID field that follows header(0)

GLbyte colorMapType; // 0 = None, 1 = paletted

GLbyte imageType; // 0 = none, 1 = indexed, 2 = rgb, 3 = grey, +8 = rle

unsigned short colorMapStart; // First colour map entry

unsigned short colorMapLength; // Number of colors

unsigned char colorMapBits; // bits per palette entry

unsigned short xstart; // image x origin

unsigned short ystart; // image y origin

unsigned short width; // width in pixels

unsigned short height; // height in pixels

GLbyte bits; // bits per pixel (8 16, 24, 32)

GLbyte descriptor; // image descriptor

} TGAHEADER;

#pragma pack(8)

////////////////////////////////////////////////////////////////////

// Allocate memory and load targa bits. Returns poin\_ter to new buffer,

// height, and width of texture, and the OpenGL for\_mat of data.

// Call free() on buffer when finished!

// This only works on pretty vanilla targas... 8, 24, or 32 bit color

// only, no palettes, no RLE encoding.

GLbyte\* gltLoadTGA(const char\* szFileName, GLint

\* iWidth, GLint\* iHeight, GLint\* iComponents, GLenum

\* eFormat)

{

FILE\* pFile; // File pointer

TGAHEADER tgaHeader; // TGA file header

unsigned long lImageSize; // Size in bytes of image

short sDepth; // Pixel depth;

GLbyte\* pBits = NULL; // Pointer to bits

// Default/Failed values

\*iWidth = 0;

\*iHeight = 0;

\*eFormat = GL\_BGR\_EXT;

\*iComponents = GL\_RGB8;

// Attempt to open the fil

pFile = fopen(szFileName, "rb");

if (pFile == NULL)

return NULL;

// Read in header (binary)

fread(&tgaHeader, 18/\* sizeof(TGAHEADER)\*/, 1, pFile);

// Do byte swap for big vs little endian

#ifdef \_\_APPLE\_\_

BYTE\_SWAP(tgaHeader.colorMapStart);

BYTE\_SWAP(tgaHeader.colorMapLength);

BYTE\_SWAP(tgaHeader.xstart);

BYTE\_SWAP(tgaHeader.ystart);

BYTE\_SWAP(tgaHeader.width);

BYTE\_SWAP(tgaHeader.height);

#endif

// Get width, height, and depth of texture

\* iWidth = tgaHeader.width;

\*iHeight = tgaHeader.height;

sDepth = tgaHeader.bits / 8;

// Put some validity checks here. Very simply, I only understand

// or care about 8, 24, or 32 bit targa's.

if (tgaHeader.bits != 8 && tgaHeader.bits != 24 &&

tgaHeader.bits != 32)

return NULL;

// Calculate size of image buffer

lImageSize = tgaHeader.width \* tgaHeader.height \*

sDepth;

// Allocate memory and check for success

pBits = (GLbyte\*)malloc(lImageSize \* sizeof(GLbyte));

if (pBits == NULL)

return NULL;

// Read in the bits

// Check for read error. This should catch RLE or other

// weird formats that I don't want to recognize

if (fread(pBits, lImageSize, 1, pFile) != 1)

{

free(pBits);

return NULL;

}

// Set OpenGL format expected

switch (sDepth)

{

case 3: // Most likely case

\*eFormat = GL\_BGR\_EXT;

\*iComponents = GL\_RGB8;

break;

case 4:

\*eFormat = GL\_BGRA\_EXT;

\*iComponents = GL\_RGBA8;

break;

case 1:

\*eFormat = GL\_LUMINANCE;

\*iComponents = GL\_LUMINANCE8;

break;

};

// Done with File

fclose(pFile);

// Return pointer to image data

return pBits;

}

// Change viewing volume and viewport. Called when window is resized

void ChangeSize(int w, int h)

{

GLfloat fAspect;

// Prevent a divide by zero

if (h == 0)

h = 1;

// Set Viewport to window dimensions

glViewport(0, 0, w, h);

fAspect = (GLfloat)w / (GLfloat)h;

// Reset coordinate system

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

// Produce the perspective projection

gluPerspective(35.0f, fAspect, 1.0, 40.0);

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

}

// This function does any needed initialization on the rendering

// context. Here it sets up and initializes the lighting for

// the scene.

void SetupRC()

{

GLubyte\* pBytes;

GLint iWidth, iHeight, iComponents;

GLenum eFormat;

// Light values and coordinates

// GLfloat whiteLight[] = { 0.05f, 0.05f, 0.05f, 1.0f };

GLfloat whiteLight[] = { 0.5f, 0.5f, 0.5f, 1.0f };

GLfloat sourceLight[] = { 0.75f, 0.75f, 0.75f, 1.0f };

GLfloat lightPos[] = { -10.f, 5.0f, 5.0f, 1.0f };

glEnable(GL\_DEPTH\_TEST);// Hidden surface remov\_al

glFrontFace(GL\_CCW); // Counter clock\_wise polygons face out

glEnable(GL\_CULL\_FACE); // Do not calculate inside of jet

// Enable lighting

glEnable(GL\_LIGHTING);

// Setup and enable light 0

glLightModelfv(GL\_LIGHT\_MODEL\_AMBIENT, whiteLight);

glLightfv(GL\_LIGHT0, GL\_AMBIENT, sourceLight);

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

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

glEnable(GL\_LIGHT0);

// Enable color tracking

glEnable(GL\_COLOR\_MATERIAL);

// Set Material properties to follow glColor val\_ues

glColorMaterial(GL\_FRONT,

GL\_AMBIENT\_AND\_DIFFUSE);

// Black blue background

glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

// Load texture

glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1);

pBytes = (GLubyte\*)gltLoadTGA("liq1.tga", &iWidth, &iHeight, &iComponents, &eFormat);

glTexImage2D(GL\_TEXTURE\_2D, 0, iComponents, iWidth, iHeight, 0, eFormat, GL\_UNSIGNED\_BYTE, pBytes);

free(pBytes);

glTexParameteri(GL\_TEXTURE\_2D,

GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D,

GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S,

GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T,

GL\_CLAMP\_TO\_EDGE);

glTexEnvi(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE,

GL\_MODULATE);

glEnable(GL\_TEXTURE\_2D);

}

// Respond to arrow keys

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

{

if (key == GLUT\_KEY\_UP)

xRot -= 5.0f;

if (key == GLUT\_KEY\_DOWN)

xRot += 5.0f;

if (key == GLUT\_KEY\_LEFT)

yRot -= 5.0f;

if (key == GLUT\_KEY\_RIGHT)

yRot += 5.0f;

xRot = (GLfloat)((const int)xRot % 360);

yRot = (GLfloat)((const int)yRot % 360);

// Refresh the Window

glutPostRedisplay();

}

typedef GLfloat GLTVector3[3]; // Three component floating point vector

// Subtract one vector from another

void gltSubtractVectors(const GLTVector3 vFirst,

const GLTVector3 vSecond, GLTVector3 vResult)

{

vResult[0] = vFirst[0] - vSecond[0];

vResult[1] = vFirst[1] - vSecond[1];

vResult[2] = vFirst[2] - vSecond[2];

}

// Scales a vector by a scalar

void gltScaleVector(GLTVector3 vVector, const GLfloat

fScale)

{

vVector[0] \*= fScale; vVector[1] \*= fScale; vVector[2] \*= fScale;

}

// Gets the length of a vector squared

GLfloat gltGetVectorLengthSqrd(const GLTVector3 vVector)

{

return (vVector[0] \* vVector[0]) + (vVector[1] \* vVector[1]) + (vVector[2] \* vVector[2]);

}

// Gets the length of a vector

GLfloat gltGetVectorLength(const GLTVector3 vVector)

{

return

(GLfloat)sqrt(gltGetVectorLengthSqrd(vVector));

}

// Scales a vector by it's length - creates a unit vector

void gltNormalizeVector(GLTVector3 vNormal)

{

GLfloat fLength = 1.0f / gltGetVectorLength(vNormal);

gltScaleVector(vNormal, fLength);

}

// Calculate the cross product of two vectors

void gltVectorCrossProduct(const GLTVector3 vU, const

GLTVector3 vV, GLTVector3 vResult)

{

vResult[0] = vU[1] \* vV[2] - vV[1] \* vU[2];

vResult[1] = -vU[0] \* vV[2] + vV[0] \* vU[2];

vResult[2] = vU[0] \* vV[1] - vV[0] \* vU[1];

}

// Called to draw scene

// Given three points on a plane in counter clockwise order, calculate the unit normal

void gltGetNormalVector(const GLTVector3 vP1, const

GLTVector3 vP2, const GLTVector3 vP3, GLTVector3

vNormal)

{

GLTVector3 vV1, vV2;

gltSubtractVectors(vP2, vP1, vV1);

gltSubtractVectors(vP3, vP1, vV2);

gltVectorCrossProduct(vV1, vV2, vNormal);

gltNormalizeVector(vNormal);

}

void RenderScene(void)

{

GLTVector3 vNormal;

GLTVector3 vCorners[8] = {

{ 1.0f, 1.0f, -2.0f },

{ 1.0f, 1.0f, 2.0f },

{-1.0f, 1.0f, 2.0f },

{-1.0f, 1.0f, -2.0f },

{ 1.0f, -1.0f, -2.0f },

{ 1.0f, -1.0f, 2.0f },

{-1.0f, -1.0f, 2.0f },

{-1.0f, -1.0f, -2.0f } };

// Clear the window with current clearing color

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

// Save the matrix state and do the rotations

glPushMatrix();

// Move object back and do in place rotation

glTranslatef(0.0f, 0.0f, -10.0f);

glRotatef(xRot, 1.0f, 0.0f, 0.0f);

glRotatef(yRot, 0.0f, 1.0f, 0.0f);

// Draw the object

glColor3f(1.0f, 1.0f, 1.0f);

glBegin(GL\_TRIANGLES);

// Upper edge

glNormal3f(0.0f, 1.0f, 0.0f);

glTexCoord2f(1.0f, 1.0f);

glVertex3fv(vCorners[0]);

glTexCoord2f(0.0f, 1.0f);

glVertex3fv(vCorners[3]);

glTexCoord2f(0.0f, 0.0f);

glVertex3fv(vCorners[2]);

glTexCoord2f(1.0f, 1.0f);

glVertex3fv(vCorners[0]);

glTexCoord2f(0.0f, 0.0f);

glVertex3fv(vCorners[2]);

glTexCoord2f(0.0f, 1.0f);

glVertex3fv(vCorners[1]);

// Bottom edge

glNormal3f(0.0f, -1.0f, 0.0f);

glTexCoord2f(1.0f, 1.0f);

glVertex3fv(vCorners[4]);

glTexCoord2f(0.0f, 0.0f);

glVertex3fv(vCorners[6]);

glTexCoord2f(0.0f, 1.0f);

glVertex3fv(vCorners[7]);

glTexCoord2f(1.0f, 1.0f);

glVertex3fv(vCorners[4]);

glTexCoord2f(1.0f, 0.0f);

glVertex3fv(vCorners[5]);

glTexCoord2f(0.0f, 0.0f);

glVertex3fv(vCorners[6]);

// Left edge

glNormal3f(-1.0f, 0.0f, 0.0f);

glTexCoord2f(1.0f, 1.0f);

glVertex3fv(vCorners[2]);

glTexCoord2f(0.0f, 1.0f);

glVertex3fv(vCorners[3]);

glTexCoord2f(0.0f, 0.0f);

glVertex3fv(vCorners[7]);

glTexCoord2f(1.0f, 1.0f);

glVertex3fv(vCorners[2]);

glTexCoord2f(0.0f, 0.0f);

glVertex3fv(vCorners[7]);

glTexCoord2f(1.0f, 0.0f);

glVertex3fv(vCorners[6]);

// Right edge

glNormal3f(1.0f, 0.0f, 0.0f);

glTexCoord2f(1.0f, 1.0f);

glVertex3fv(vCorners[0]);

glTexCoord2f(0.0f, 1.0f);

glVertex3fv(vCorners[1]);

glTexCoord2f(0.0f, 0.0f);

glVertex3fv(vCorners[5]);

glTexCoord2f(1.0f, 1.0f);

glVertex3fv(vCorners[0]);

glTexCoord2f(0.0f, 0.0f);

glVertex3fv(vCorners[5]);

glTexCoord2f(1.0f, 0.0f);

glVertex3fv(vCorners[4]);

// Front edge

glNormal3f(0.0f, 0.0f, 1.0f);

glTexCoord2f(1.0f, 1.0f);

glVertex3fv(vCorners[1]);

glTexCoord2f(0.0f, 1.0f);

glVertex3fv(vCorners[2]);

glTexCoord2f(0.0f, 0.0f);

glVertex3fv(vCorners[6]);

glTexCoord2f(1.0f, 1.0f);

glVertex3fv(vCorners[1]);

glTexCoord2f(0.0f, 0.0f);

glVertex3fv(vCorners[6]);

glTexCoord2f(1.0f, 0.0f);

glVertex3fv(vCorners[5]);

// Back edge

glNormal3f(0.0f, 0.0f, -1.0f);

glTexCoord2f(1.0f, 1.0f);

glVertex3fv(vCorners[0]);

glTexCoord2f(1.0f, 0.0f);

glVertex3fv(vCorners[4]);

glTexCoord2f(0.0f, 0.0f);

glVertex3fv(vCorners[7]);

glTexCoord2f(1.0f, 1.0f);

glVertex3fv(vCorners[0]);

glTexCoord2f(0.0f, 0.0f);

glVertex3fv(vCorners[7]);

glTexCoord2f(0.0f, 1.0f);

glVertex3fv(vCorners[3]);

glEnd();

// Restore the matrix state

glPopMatrix();

// Buffer swap

glutSwapBuffers();

}

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

{

glutInit(&argc, argv);

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

glutInitWindowSize(800, 600);

glutCreateWindow("Textured Pyramid");

glutReshapeFunc(ChangeSize);

glutSpecialFunc(SpecialKeys);

glutDisplayFunc(RenderScene);

SetupRC();

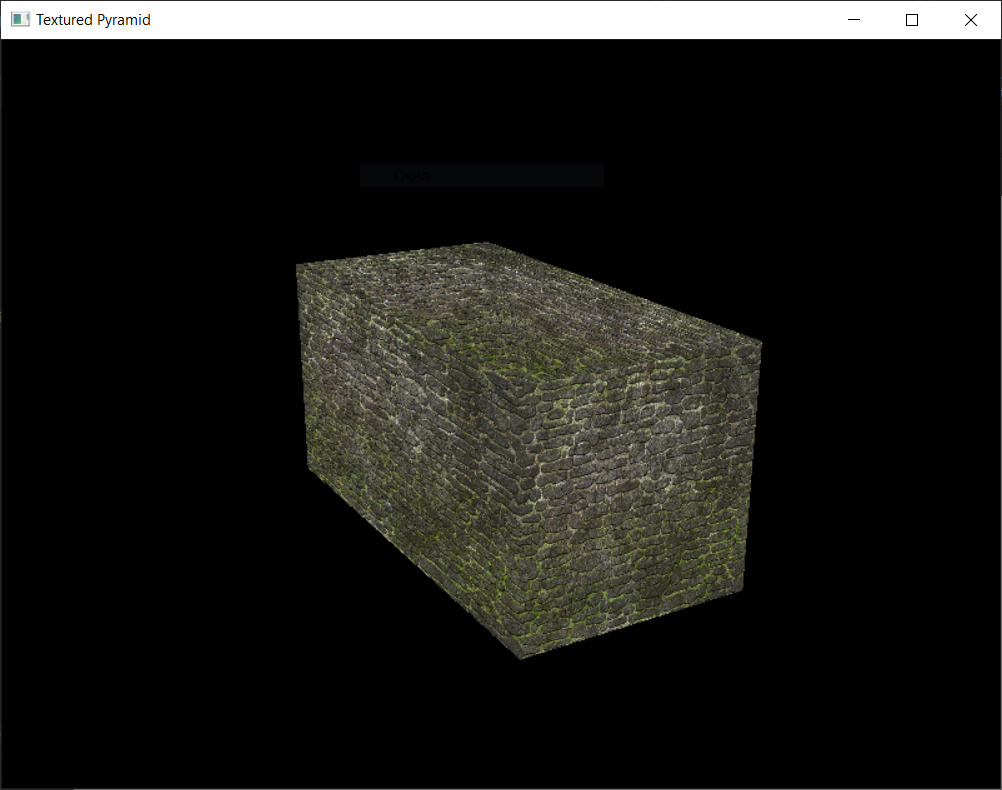
glutMainLoop();

return 0;

}

}

**Результат листинга 1:**



**Рисунок 1.** Результат листинга 1

**Листинг 2:**

#include "glew.h"

#include "glut.h"

#include <math.h>

namespace Task\_5\_2

{

// Rotation angle

static GLfloat yRot = 0.0f;

#define GLT\_PI 3.14159265358979323846

///////////////////////////////////////////////////////

// Some data types

typedef GLfloat GLTVector2[2]; // Two component floating point vector

typedef GLfloat GLTVector3[3]; // Three component floating point vector

typedef GLfloat GLTVector4[4]; // Four component floating point vector

typedef GLfloat GLTMatrix[16]; // A column major 4x4 matrix of type GLfloat

// Adds two vectors together

void gltAddVectors(const GLTVector3 vFirst, const

GLTVector3 vSecond, GLTVector3 vResult) {

vResult[0] = vFirst[0] + vSecond[0];

vResult[1] = vFirst[1] + vSecond[1];

vResult[2] = vFirst[2] + vSecond[2];

}

// Subtract one vector from another

void gltSubtractVectors(const GLTVector3 vFirst,

const GLTVector3 vSecond, GLTVector3 vResult)

{

vResult[0] = vFirst[0] - vSecond[0];

vResult[1] = vFirst[1] - vSecond[1];

vResult[2] = vFirst[2] - vSecond[2];

}

// Scales a vector by a scalar

void gltScaleVector(GLTVector3 vVector, const GLfloat fScale)

{

vVector[0] \*= fScale; vVector[1] \*= fScale; vVector[2] \*= fScale;

}

// Gets the length of a vector squared

GLfloat gltGetVectorLengthSqrd(const GLTVector3 vVector)

{

return (vVector[0] \* vVector[0]) + (vVector[1] \* vVector[1]) + (vVector[2] \* vVector[2]);

}

// Gets the length of a vector

GLfloat gltGetVectorLength(const GLTVector3 vVector)

{

return (GLfloat)sqrt(gltGetVectorLengthSqrd(vVector));

}

// Scales a vector by it's length - creates a unit vector

void gltNormalizeVector(GLTVector3 vNormal)

{

GLfloat fLength = 1.0f / gltGetVectorLength(vNormal);

gltScaleVector(vNormal, fLength);

}

// Transform a point by a 4x4 matrix

void gltTransformPoint(const GLTVector3 vSrcVector,

const GLTMatrix mMatrix, GLTVector3 vOut)

{

vOut[0] = mMatrix[0] \* vSrcVector[0] + mMatrix[4]

\* vSrcVector[1] + mMatrix[8] \* vSrcVector[2] + mMatrix[12];

vOut[1] = mMatrix[1] \* vSrcVector[0] + mMatrix[5]

\* vSrcVector[1] + mMatrix[9] \* vSrcVector[2] + mMatrix[13];

vOut[2] = mMatrix[2] \* vSrcVector[0] + mMatrix[6]

\* vSrcVector[1] + mMatrix[10] \* vSrcVector[2] + mMatrix[14];

}

// Rotates a vector using a 4x4 matrix. Translation column is ignored

void gltRotateVector(const GLTVector3 vSrcVector,

const GLTMatrix mMatrix, GLTVector3 vOut)

{

vOut[0] = mMatrix[0] \* vSrcVector[0] + mMatrix[4]

\* vSrcVector[1] + mMatrix[8] \* vSrcVector[2];

vOut[1] = mMatrix[1] \* vSrcVector[0] + mMatrix[5]

\* vSrcVector[1] + mMatrix[9] \* vSrcVector[2];

vOut[2] = mMatrix[2] \* vSrcVector[0] + mMatrix[6]

\* vSrcVector[1] + mMatrix[10] \* vSrcVector[2];

}

// Get the dot product between two vectors

GLfloat gltVectorDotProduct(const GLTVector3 vU,

const GLTVector3 vV)

{

return vU[0] \* vV[0] + vU[1] \* vV[1] + vU[2] \* vV[2];

}

void toonDramCylinder(GLfloat radius, int edgeNum, GLTVector3 vLightDir)

{

GLTMatrix mModelViewMatrix;

GLTVector3 vNormal, vTransformedNormal;

double step = 2.0f \* GLT\_PI / edgeNum;

// Get the modelview matrix

glGetFloatv(GL\_MODELVIEW\_MATRIX, mModelViewMatrix);

// Normalize the light vector

gltNormalizeVector(vLightDir);

for (int i = 0; i < edgeNum; ++i)

{

GLfloat vx = cos((i + 0.5) \* step);

GLfloat vz = sin((i + 0.5) \* step);

GLfloat vy = 0.0f;

GLTVector3 norm = { vx, vy, vz };

GLfloat x0 = radius \* cos(i \* step);

GLfloat z0 = -radius \* sin(i \* step);

GLfloat y0 = 2.0f;

GLfloat x1 = radius \* cos((i + 1) \* step);

GLfloat z1 = -radius \* sin((i + 1) \* step);

GLfloat y1 = -2.0f;

glBegin(GL\_TRIANGLE\_STRIP);

glTexCoord1f(gltVectorDotProduct(vLightDir, norm));

glVertex3f(x0, y0, z0);

glVertex3f(x0, y1, z0);

glVertex3f(x1, y0, z1);

glVertex3f(x1, y1, z1);

glEnd();

}

GLTVector3 n0 = { 0.0f, -1.0f, 0.0f };

glBegin(GL\_TRIANGLE\_FAN);

glTexCoord1f(gltVectorDotProduct(vLightDir, n0));

glVertex3f(0.0f, 2.0f, 0.0f);

for (int i = 0; i < edgeNum + 1; ++i)

{

GLfloat x = radius \* cos(i \* step);

GLfloat z = -radius \* sin(i \* step);

GLfloat y = 2.0f;

glVertex3f(x, y, z);

}

glEnd();

GLTVector3 n1 = { 0.0f, 1.0f, 0.0f };

glBegin(GL\_TRIANGLE\_FAN);

glTexCoord1f(gltVectorDotProduct(vLightDir, n1));

glVertex3f(0.0f, -2.0f, 0.0f);

for (int i = 0; i < edgeNum + 1; ++i)

{

GLfloat x = radius \* cos(-i \* step);

GLfloat z = -radius \* sin(-i \* step);

GLfloat y = -2.0f;

glVertex3f(x, y, z);

}

glEnd();

}

// Draw a torus (doughnut), using the current 1D texture for light shading

void toonDrawTorus(GLfloat majorRadius, GLfloat minorRadius,

int numMajor, int numMinor, GLTVector3 vLightDir)

{

GLTMatrix mModelViewMatrix;

GLTVector3 vNormal, vTransformedNormal;

double majorStep = 2.0f \* GLT\_PI / numMajor;

double minorStep = 2.0f \* GLT\_PI / numMinor;

int i, j;

// Get the modelview matrix

glGetFloatv(GL\_MODELVIEW\_MATRIX, mModelViewMatrix);

// Normalize the light vector

gltNormalizeVector(vLightDir);

// Draw torus as a series of triangle strips

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

{

double a0 = i \* majorStep;

double a1 = a0 + majorStep;

GLfloat x0 = (GLfloat)cos(a0);

GLfloat y0 = (GLfloat)sin(a0);

GLfloat x1 = (GLfloat)cos(a1);

GLfloat y1 = (GLfloat)sin(a1);

glBegin(GL\_TRIANGLE\_STRIP);

for (j = 0; j <= numMinor; ++j)

{

double b = j \* minorStep;

GLfloat c = (GLfloat)cos(b);

GLfloat r = minorRadius \* c + majorRadius;

GLfloat z = minorRadius \* (GLfloat)sin(b);

// First point

vNormal[0] = x0 \* c;

vNormal[1] = y0 \* c;

vNormal[2] = z / minorRadius;

gltNormalizeVector(vNormal);

gltRotateVector(vNormal, mModelViewMatrix, vTransformedNormal);

// Texture coordinate is set by intensity of light

glTexCoord1f(gltVectorDotProduct(vLightDir, vTransformedNormal));

glVertex3f(x0 \* r, y0 \* r, z);

// Second point

vNormal[0] = x1 \* c;

vNormal[1] = y1 \* c;

vNormal[2] = z / minorRadius;

gltNormalizeVector(vNormal);

gltRotateVector(vNormal, mModelViewMatrix, vTransformedNormal);

// Texture coordinate is set by intensity of light

glTexCoord1f(gltVectorDotProduct(vLightDir, vTransformedNormal));

glVertex3f(x1 \* r, y1 \* r, z);

}

glEnd();

}

}

// Called to draw scene

void RenderScene(void)

{

// Where is the light coming from

GLTVector3 vLightDir0 = { cos(yRot \* GLT\_PI / 180), 0.0f, -sin(yRot \* GLT\_PI / 180) };

GLTVector3 vLightDir1 = { 0.0f, cos(yRot \* GLT\_PI / 180), -sin(yRot \* GLT\_PI / 180) };

// Clear the window with current clearing color

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

glPushMatrix();

glTranslatef(0.0f, 0.0f, -15.0f);

glRotatef(yRot, 0.0f, 1.0f, 0.0f);

glTranslatef(2.0f, 0.0f, 0.0f);

toonDramCylinder(1.0f, 50, vLightDir0);

glTranslatef(-4.0f, 0.0f, 0.0f);

toonDramCylinder(1.0f, 50, vLightDir0);

glTranslatef(2.0f, 3.0f, 0.0f);

glRotatef(90, 0.0f, 0.0f, 1.0f);

toonDramCylinder(1.0f, 50, vLightDir1);

glPopMatrix();

// Do the buffer Swap

glutSwapBuffers();

// Rotate 1/2 degree more each frame

yRot += 1.0f;

}

// This function does any needed initialization on the rendering

// context.

void SetupRC()

{

// Load a 1D texture with toon shaded values

// Green, greener...

GLbyte toonTable[12][3] = {

{ 0, 16, 0 },

{ 0, 32, 0 },

{ 0, 48, 0 },

{ 0, 64, 0 },

{ 0, 80, 0 },

{ 0, 96, 0 },

{ 0, 112, 0 },

{ 0, 128, 0 },

{ 0, 144, 0 },

{ 0, 160, 0 },

{ 0, 176, 0 },

{ 0, 192, 0 },

};

// Bluish background

glClearColor(0.0f, 0.0f, .50f, 1.0f);

glEnable(GL\_DEPTH\_TEST);

glEnable(GL\_CULL\_FACE);

glTexEnvi(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_DECAL);

glTexParameteri(GL\_TEXTURE\_1D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_1D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_1D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP);

glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1);

glTexImage1D(GL\_TEXTURE\_1D, 0, GL\_RGB, 12, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, toonTable);

glEnable(GL\_TEXTURE\_1D);

}

///////////////////////////////////////////////////////////

// Called by GLUT library when idle (window not being

// resized or moved)

void TimerFunction(int value)

{

// Redraw the scene with new coordinates

glutPostRedisplay();

glutTimerFunc(33, TimerFunction, 1);

}

void ChangeSize(int w, int h)

{

GLfloat fAspect;

// Prevent a divide by zero, when window is too short

// (you cant make a window of zero width).

if (h == 0)

h = 1;

glViewport(0, 0, w, h);

fAspect = (GLfloat)w / (GLfloat)h;

// Reset the coordinate system before modifying

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

// Set the clipping volume

gluPerspective(35.0f, fAspect, 1.0f, 50.0f);

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

}

///////////////////////////////////////////////////

// Program entry point

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

{

glutInit(&argc, argv);

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

glutInitWindowSize(800, 600);

glutCreateWindow("Toon/Cell Shading Demo");

glutReshapeFunc(ChangeSize);

glutDisplayFunc(RenderScene);

glutTimerFunc(33, TimerFunction, 1);

SetupRC();

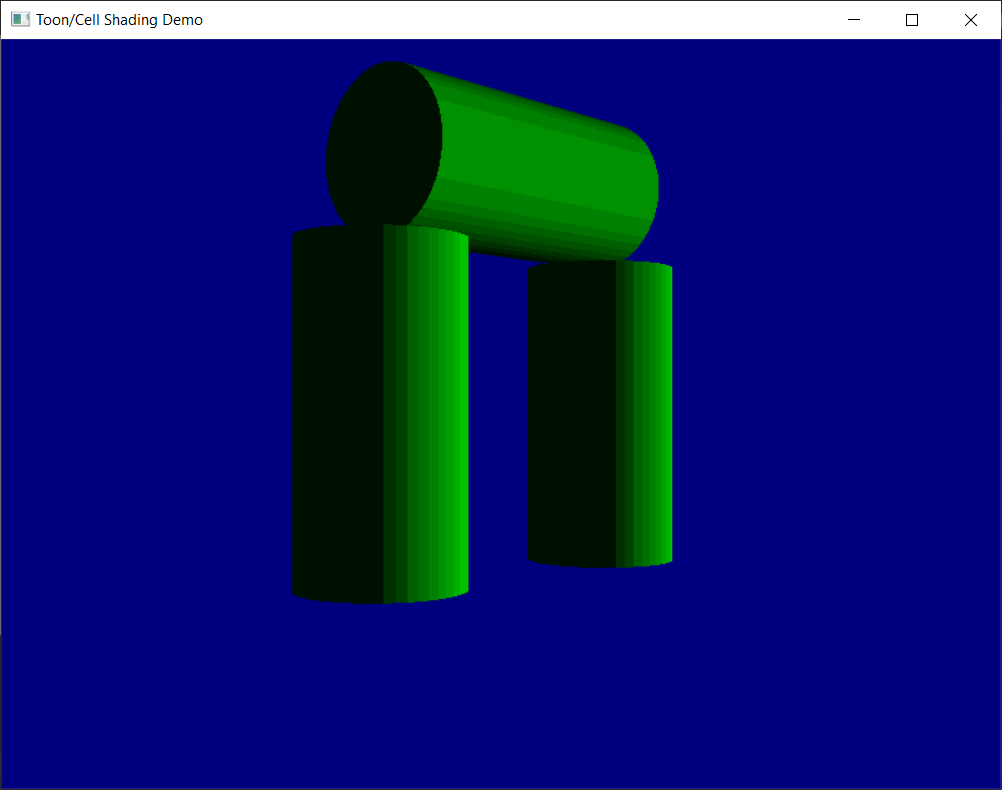
glutMainLoop();

return 0;

}

}

**Результат листинга 2:**



**Рисунок 2.** Результат листинга 2

**Листинг 3:**

#include "glew.h"

#include "glut.h"

#include <math.h>

#include <stdio.h>

#include <stdlib.h>

namespace Task\_5\_3

{

#pragma warning(disable : 4996)

// Rotation amounts

static GLfloat zPos = -60.0f;

// Texture objects

#define TEXTURE\_BRICK 0

#define TEXTURE\_FLOOR 1

#define TEXTURE\_CEILING 2

#define TEXTURE\_COUNT 3

GLuint textures[TEXTURE\_COUNT];

const char\* szTextureFiles[TEXTURE\_COUNT] = { "brick.tga", "floor.tga", "ceiling.tga" };

// Define targa header.

#pragma pack(1)

typedef struct

{

GLbyte identsize; // Size of ID field that follows header(0)

GLbyte colorMapType; // 0 = None, 1 = paletted

GLbyte imageType; // 0 = none, 1 = indexed, 2 = rgb, 3 = grey, +8 = rle

unsigned short colorMapStart; // First colour map entry

unsigned short colorMapLength; // Number of colors

unsigned char colorMapBits; // bits per palette entry

unsigned short xstart; // im\_age x origin

unsigned short ystart; // im\_age y origin

unsigned short width; // width in pixels

unsigned short height; // height in pixels

GLbyte bits; // bits per pixel (8 16, 24, 32)

GLbyte descriptor; // image descrip\_tor

} TGAHEADER;

#pragma pack(8)

/////////////////////////////////////////////////////

///////////////

// Allocate memory and load targa bits. Returns poin\_ter to new buffer,

// height, and width of texture, and the OpenGL for\_mat of data.

// Call free() on buffer when finished!

// This only works on pretty vanilla targas... 8, 24, or 32 bit color

// only, no palettes, no RLE encoding.

GLbyte\* gltLoadTGA(const char\* szFileName, GLint

\* iWidth, GLint\* iHeight, GLint\* iComponents, GLenum

\* eFormat)

{

FILE\* pFile; // File pointer

TGAHEADER tgaHeader; // TGA file header

unsigned long lImageSize; // Size in bytes of image

short sDepth; // Pixel depth;

GLbyte\* pBits = NULL; // Pointer to bits

// Default/Failed values

\*iWidth = 0;

\*iHeight = 0;

\*eFormat = GL\_BGR\_EXT;

\*iComponents = GL\_RGB8;

// Attempt to open the fil

pFile = fopen(szFileName, "rb");

if (pFile == NULL)

return NULL;

// Read in header (binary)

fread(&tgaHeader, 18/\* sizeof(TGAHEADER)\*/, 1,

pFile);

// Do byte swap for big vs little endian

#ifdef \_\_APPLE\_\_

BYTE\_SWAP(tgaHeader.colorMapStart);

BYTE\_SWAP(tgaHeader.colorMapLength);

BYTE\_SWAP(tgaHeader.xstart);

BYTE\_SWAP(tgaHeader.ystart);

BYTE\_SWAP(tgaHeader.width);

BYTE\_SWAP(tgaHeader.height);

#endif

// Get width, height, and depth of texture

\* iWidth = tgaHeader.width;

\*iHeight = tgaHeader.height;

sDepth = tgaHeader.bits / 8;

// Put some validity checks here. Very simply, I only understand

// or care about 8, 24, or 32 bit targa's.

if (tgaHeader.bits != 8 && tgaHeader.bits != 24 &&

tgaHeader.bits != 32)

return NULL;

// Calculate size of image buffer

lImageSize = tgaHeader.width \* tgaHeader.height \*

sDepth;

// Allocate memory and check for success

pBits = (GLbyte\*)malloc(lImageSize \* sizeof(GLbyte));

if (pBits == NULL)

return NULL;

// Read in the bits

// Check for read error. This should catch RLE or other

// weird formats that I don't want to recognize

if (fread(pBits, lImageSize, 1, pFile) != 1)

{

free(pBits);

return NULL;

}

// Set OpenGL format expected

switch (sDepth)

{

case 3: // Most likely case

\*eFormat = GL\_BGR\_EXT;

\*iComponents = GL\_RGB8;

break;

case 4:

\*eFormat = GL\_BGRA\_EXT;

\*iComponents = GL\_RGBA8;

break;

case 1:

\*eFormat = GL\_LUMINANCE;

\*iComponents = GL\_LUMINANCE8;

break;

};

// Done with File

fclose(pFile);

// Return pointer to image data

return pBits;

}

/////////////////////////////////////////////////////

//////////////////////////

// Change texture filter for each texture object

void ProcessMenu(int value)

{

GLint iLoop;

for (iLoop = 0; iLoop < TEXTURE\_COUNT; iLoop++)

{

glBindTexture(GL\_TEXTURE\_2D, textures[iLoop]);

switch (value)

{

case 0:

glTexParameteri(GL\_TEXTURE\_2D,

GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);

break;

case 1:

glTexParameteri(GL\_TEXTURE\_2D,

GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

break;

case 2:

glTexParameteri(GL\_TEXTURE\_2D,

GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST\_MIPMAP\_NEAREST);

break;

case 3:

glTexParameteri(GL\_TEXTURE\_2D,

GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST\_MIPMAP\_LINEAR);

break;

case 4:

glTexParameteri(GL\_TEXTURE\_2D,

GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_NEAREST);

break;

case 5:

default:

glTexParameteri(GL\_TEXTURE\_2D,

GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

break;

}

}

// Trigger Redraw

glutPostRedisplay();

}

/////////////////////////////////////////////////////

/////////////

// This function does any needed initialization on the rendering

// context. Here it sets up and initializes the tex\_ture objects.

void SetupRC()

{

GLubyte\* pBytes;

GLint iWidth, iHeight, iComponents;

GLenum eFormat;

GLint iLoop;

// Black background

glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

// Textures applied as decals, no lighting or co\_loring effects

glEnable(GL\_TEXTURE\_2D);

glTexEnvi(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE,

GL\_DECAL);

// Load textures

glGenTextures(TEXTURE\_COUNT, textures);

for (iLoop = 0; iLoop < TEXTURE\_COUNT; iLoop++)

{

// Bind to next texture object

glBindTexture(GL\_TEXTURE\_2D, textures[iLoop]);

// Load texture, set filter and wrap modes

pBytes = (GLubyte\*)gltLoadTGA(szTextureFiles[iLoop], &iWidth,

&iHeight, &iComponents, &eFormat);

gluBuild2DMipmaps(GL\_TEXTURE\_2D, iComponents,

iWidth, iHeight, eFormat, GL\_UNSIGNED\_BYTE, pBytes);

glTexParameteri(GL\_TEXTURE\_2D,

GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D,

GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D,

GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D,

GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

// Don't need original texture data any more

free(pBytes);

}

}

///////////////////////////////////////////////////

// Shutdown the rendering context. Just deletes the

// texture objects

void ShutdownRC(void)

{

glDeleteTextures(TEXTURE\_COUNT, textures);

}

///////////////////////////////////////////////////

// Respond to arrow keys, move the viewpoint back

// and forth

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

{

if (key == GLUT\_KEY\_UP)

zPos += 1.0f;

if (key == GLUT\_KEY\_DOWN)

zPos -= 1.0f;

// Refresh the Window

glutPostRedisplay();

}

/////////////////////////////////////////////////////

////////////////

// Change viewing volume and viewport. Called when window is resized

void ChangeSize(int w, int h)

{

GLfloat fAspect;

// Prevent a divide by zero

if (h == 0)

h = 1;

// Set Viewport to window dimensions

glViewport(0, 0, w, h);

fAspect = (GLfloat)w / (GLfloat)h;

// Reset coordinate system

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

// Produce the perspective projection

gluPerspective(90.0f, fAspect, 1, 120);

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

}

/////////////////////////////////////////////////////

//

// Called to draw scene

void RenderScene(void)

{

GLfloat z;

// Clear the window with current clearing color

glClear(GL\_COLOR\_BUFFER\_BIT);

// Save the matrix state and do the rotations

glPushMatrix();

// Move object back and do in place rotation

glTranslatef(0.0f, 0.0f, zPos);

for (z = 60.0f; z >= 0.0f; z -= 10)

{

// Floor

glBindTexture(GL\_TEXTURE\_2D, textures[TEXTURE\_FLOOR]);

glBegin(GL\_QUADS);

glTexCoord2f(0.0f, 0.0f);

glVertex3f(-10.0f, -10.0f, z);

glTexCoord2f(1.0f, 0.0f);

glVertex3f(10.0f, -10.0f, z);

glTexCoord2f(1.0f, 1.0f);

glVertex3f(10.0f, -10.0f, z -

10.0f);

glTexCoord2f(0.0f, 1.0f);

glVertex3f(-10.0f, -10.0f, z -

10.0f);

glEnd();

// Ceiling

glBindTexture(GL\_TEXTURE\_2D, textures[TEXTURE\_CEILING]);

glBegin(GL\_QUADS);

glTexCoord2f(0.0f, 1.0f);

glVertex3f(-10.0f, 10.0f, z -

10.0f);

glTexCoord2f(1.0f, 1.0f);

glVertex3f(10.0f, 10.0f, z - 10.0f);

glTexCoord2f(1.0f, 0.0f);

glVertex3f(10.0f, 10.0f, z);

glTexCoord2f(0.0f, 0.0f);

glVertex3f(-10.0f, 10.0f, z);

glEnd();

// Upper Left Wall

glBindTexture(GL\_TEXTURE\_2D, textures[TEXTURE\_BRICK]);

glBegin(GL\_QUADS);

glTexCoord2f(0.0f, 0.0f);

glVertex3f(-20.0f, 0.0f, z);

glTexCoord2f(1.0f, 0.0f);

glVertex3f(-20.0f, 0.0f, z - 10.0f);

glTexCoord2f(1.0f, 1.0f);

glVertex3f(-10.0f, 10.0f, z - 10.0f);

glTexCoord2f(0.0f, 1.0f);

glVertex3f(-10.0f, 10.0f, z);

glEnd();

// Bottom Left Wall

glBindTexture(GL\_TEXTURE\_2D, textures[TEXTURE\_BRICK]);

glBegin(GL\_QUADS);

glTexCoord2f(0.0f, 0.0f);

glVertex3f(-10.0f, -10.0f, z);

glTexCoord2f(1.0f, 0.0f);

glVertex3f(-10.0f, -10.0f, z - 10.0f);

glTexCoord2f(1.0f, 1.0f);

glVertex3f(-20.0f, 0.0f, z - 10.0f);

glTexCoord2f(0.0f, 1.0f);

glVertex3f(-20.0f, 0.0f, z);

glEnd();

// Upper Right Wall

glBegin(GL\_QUADS);

glTexCoord2f(0.0f, 1.0f);

glVertex3f(10.0f, 10.0f, z);

glTexCoord2f(1.0f, 1.0f);

glVertex3f(10.0f, 10.0f, z - 10.0f);

glTexCoord2f(1.0f, 0.0f);

glVertex3f(20.0f, 0.0f, z - 10.0f);

glTexCoord2f(0.0f, 0.0f);

glVertex3f(20.0f, 0.0f, z);

glEnd();

// Bottom Right Wall

glBegin(GL\_QUADS);

glTexCoord2f(0.0f, 1.0f);

glVertex3f(20.0f, 0.0f, z);

glTexCoord2f(1.0f, 1.0f);

glVertex3f(20.0f, 0.0f, z - 10.0f);

glTexCoord2f(1.0f, 0.0f);

glVertex3f(10.0f, -10.0f, z - 10.0f);

glTexCoord2f(0.0f, 0.0f);

glVertex3f(10.0f, -10.0f, z);

glEnd();

}

// Restore the matrix state

glPopMatrix();

// Buffer swap

glutSwapBuffers();

}

/////////////////////////////////////////////////////

// Program entry point

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

{

// Standard initialization stuff

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB);

glutInitWindowSize(800, 600);

glutCreateWindow("Tunnel");

glutReshapeFunc(ChangeSize);

glutSpecialFunc(SpecialKeys);

glutDisplayFunc(RenderScene);

// Add menu entries to change filter

glutCreateMenu(ProcessMenu);

glutAddMenuEntry("GL\_NEAREST", 0);

glutAddMenuEntry("GL\_LINEAR", 1);

glutAddMenuEntry("GL\_NEAREST\_MIPMAP\_NEAREST", 2);

glutAddMenuEntry("GL\_NEAREST\_MIPMAP\_LINEAR", 3);

glutAddMenuEntry("GL\_LINEAR\_MIPMAP\_NEAREST", 4);

glutAddMenuEntry("GL\_LINEAR\_MIPMAP\_LINEAR", 5);

glutAttachMenu(GLUT\_RIGHT\_BUTTON);

// Startup, loop, shutdown

SetupRC();

glutMainLoop();

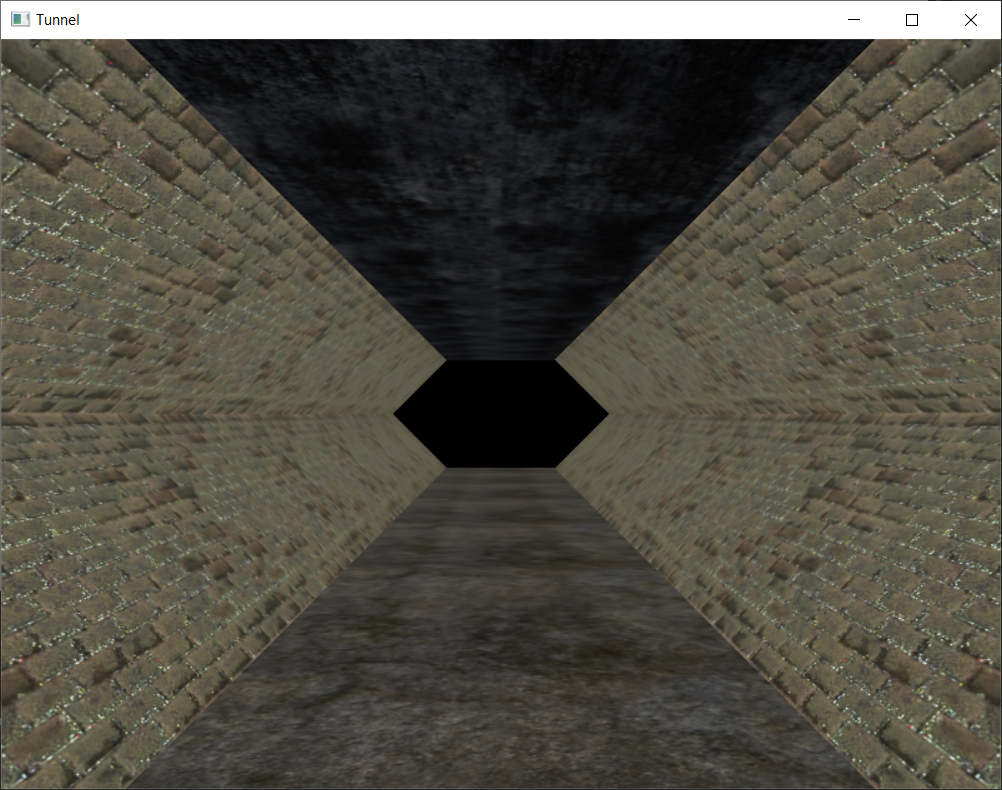
ShutdownRC();

return 0;

}

}

**Результат листинга 3:**



**Рисунок 3.** Результат листинга 3

**Листинг 4:**

#include "glew.h"

#include "glut.h"

#include <math.h>

#include "stdio.h"

#include "stdlib.h"

#include <windows.h>

namespace Task\_5\_4

{

#pragma warning(disable : 4996)

/////////////////////////////////////////////////////

//

// Useful constants

#define GLT\_PI 3.14159265358979323846

#define GLT\_PI\_DIV\_180 0.017453292519943296

#define GLT\_INV\_PI\_DIV\_180 57.2957795130823229

/////////////////////////////////////////////////////

//////////////////////////

// Useful shortcuts and macros

// Radians are king... but we need a way to swap back and forth

#define gltDegToRad(x) ((x)\*GLT\_PI\_DIV\_180)

#define gltRadToDeg(x) ((x)\*GLT\_INV\_PI\_DIV\_180)

/////////////////////////////////////////////////////

//

// Some data types

typedef GLfloat GLTVector2[2]; // Two component floating point vector

typedef GLfloat GLTVector3[3]; // Three compo\_nent floating point vector

typedef GLfloat GLTVector4[4]; // Four component floating point vector

typedef GLfloat GLTMatrix[16]; // A column major 4x4 matrix of type GLfloat

typedef struct { // The Frame of reference container

GLTVector3 vLocation;

GLTVector3 vUp;

GLTVector3 vForward;

} GLTFrame;

#define NUM\_SPHERES 30

GLTFrame spheres[NUM\_SPHERES];

GLTFrame frameCamera;

// Define targa header.

#pragma pack(1)

typedef struct

{

GLbyte identsize; // Size of ID field that follows header(0)

GLbyte colorMapType; // 0 = None, 1 = paletted

GLbyte imageType; // 0 = none, 1 = indexed, 2 = rgb, 3 = grey, +8 = rle

unsigned short colorMapStart; // First colour map entry

unsigned short colorMapLength; // Number of colors

unsigned char colorMapBits; // bits per palette entry

unsigned short xstart; // im\_age x origin

unsigned short ystart; // im\_age y origin

unsigned short width; // width in pixels

unsigned short height; // height in pixels

GLbyte bits; // bits per pixel (8 16, 24, 32)

GLbyte descriptor; // image descrip\_tor

} TGAHEADER;

#pragma pack(8)

/////////////////////////////////////////////////////

///////////////

// Allocate memory and load targa bits. Returns poin\_ter to new buffer,

// height, and width of texture, and the OpenGL for\_mat of data.

// Call free() on buffer when finished!

// This only works on pretty vanilla targas... 8, 24, or 32 bit color

// only, no palettes, no RLE encoding.

GLbyte\* gltLoadTGA(const char\* szFileName, GLint

\* iWidth, GLint\* iHeight, GLint\* iComponents, GLenum

\* eFormat)

{

FILE\* pFile; // File pointer

TGAHEADER tgaHeader; // TGA file header

unsigned long lImageSize; // Size in bytes of image

short sDepth; // Pixel depth;

GLbyte\* pBits = NULL; // Pointer to bits

// Default/Failed values

\*iWidth = 0;

\*iHeight = 0;

\*eFormat = GL\_BGR\_EXT;

\*iComponents = GL\_RGB8;

// Attempt to open the fil

pFile = fopen(szFileName, "rb");

if (pFile == NULL)

return NULL;

// Read in header (binary)

fread(&tgaHeader, 18/\* sizeof(TGAHEADER)\*/, 1,

pFile);

// Do byte swap for big vs little endian

#ifdef \_\_APPLE\_\_

BYTE\_SWAP(tgaHeader.colorMapStart);

BYTE\_SWAP(tgaHeader.colorMapLength);

BYTE\_SWAP(tgaHeader.xstart);

BYTE\_SWAP(tgaHeader.ystart);

BYTE\_SWAP(tgaHeader.width);

BYTE\_SWAP(tgaHeader.height);

#endif

// Get width, height, and depth of texture

\* iWidth = tgaHeader.width;

\*iHeight = tgaHeader.height;

sDepth = tgaHeader.bits / 8;

// Put some validity checks here. Very simply, I only understand

// or care about 8, 24, or 32 bit targa's.

if (tgaHeader.bits != 8 && tgaHeader.bits != 24 &&

tgaHeader.bits != 32)

return NULL;

// Calculate size of image buffer

lImageSize = tgaHeader.width \* tgaHeader.height \*

sDepth;

// Allocate memory and check for success

pBits = (GLbyte\*)malloc(lImageSize \* sizeof(GLbyte));

if (pBits == NULL)

return NULL;

// Read in the bits

// Check for read error. This should catch RLE or other

// weird formats that I don't want to recognize

if (fread(pBits, lImageSize, 1, pFile) != 1)

{

free(pBits);

return NULL;

}

// Set OpenGL format expected

switch (sDepth)

{

case 3: // Most likely case

\*eFormat = GL\_BGR\_EXT;

\*iComponents = GL\_RGB8;

break;

case 4:

\*eFormat = GL\_BGRA\_EXT;

\*iComponents = GL\_RGBA8;

break;

case 1:

\*eFormat = GL\_LUMINANCE;

\*iComponents = GL\_LUMINANCE8;

break;

};

// Done with File

fclose(pFile);

// Return pointer to image data

return pBits;

}

// Adds two vectors together

void gltAddVectors(const GLTVector3 vFirst, const

GLTVector3 vSecond, GLTVector3 vResult) {

vResult[0] = vFirst[0] + vSecond[0];

vResult[1] = vFirst[1] + vSecond[1];

vResult[2] = vFirst[2] + vSecond[2];

}

// Subtract one vector from another

void gltSubtractVectors(const GLTVector3 vFirst,

const GLTVector3 vSecond, GLTVector3 vResult)

{

vResult[0] = vFirst[0] - vSecond[0];

vResult[1] = vFirst[1] - vSecond[1];

vResult[2] = vFirst[2] - vSecond[2];

}

// Scales a vector by a scalar

void gltScaleVector(GLTVector3 vVector, const GLfloat

fScale)

{

vVector[0] \*= fScale; vVector[1] \*= fScale; vVector[2] \*= fScale;

}

// Gets the length of a vector squared

GLfloat gltGetVectorLengthSqrd(const GLTVector3 vVector)

{

return (vVector[0] \* vVector[0]) + (vVector[1] \* vVector[1]) + (vVector[2] \* vVector[2]);

}

// Gets the length of a vector

GLfloat gltGetVectorLength(const GLTVector3 vVector)

{

return

(GLfloat)sqrt(gltGetVectorLengthSqrd(vVector));

}

// Scales a vector by it's length - creates a unit vector

void gltNormalizeVector(GLTVector3 vNormal)

{

GLfloat fLength = 1.0f / gltGetVectorLength(vNormal);

gltScaleVector(vNormal, fLength);

}

// Copies a vector

void gltCopyVector(const GLTVector3 vSource, GLTVector3 vDest)

{

memcpy(vDest, vSource, sizeof(GLTVector3));

}

// Get the dot product between two vectors

GLfloat gltVectorDotProduct(const GLTVector3 vU,

const GLTVector3 vV)

{

return vU[0] \* vV[0] + vU[1] \* vV[1] + vU[2] \* vV[2];

}

// Calculate the cross product of two vectors

void gltVectorCrossProduct(const GLTVector3 vU, const

GLTVector3 vV, GLTVector3 vResult)

{

vResult[0] = vU[1] \* vV[2] - vV[1] \* vU[2];

vResult[1] = -vU[0] \* vV[2] + vV[0] \* vU[2];

vResult[2] = vU[0] \* vV[1] - vV[0] \* vU[1];

}

// Given three points on a plane in counter clockwise order, calculate the unit normal

void gltGetNormalVector(const GLTVector3 vP1, const

GLTVector3 vP2, const GLTVector3 vP3, GLTVector3

vNormal)

{

GLTVector3 vV1, vV2;

gltSubtractVectors(vP2, vP1, vV1);

gltSubtractVectors(vP3, vP1, vV2);

gltVectorCrossProduct(vV1, vV2, vNormal);

gltNormalizeVector(vNormal);

}

// Transform a point by a 4x4 matrix

void gltTransformPoint(const GLTVector3 vSrcVector,

const GLTMatrix mMatrix, GLTVector3 vOut)

{

vOut[0] = mMatrix[0] \* vSrcVector[0] + mMatrix[4]

\* vSrcVector[1] + mMatrix[8] \* vSrcVector[2] + mMatrix[12];

vOut[1] = mMatrix[1] \* vSrcVector[0] + mMatrix[5]

\* vSrcVector[1] + mMatrix[9] \* vSrcVector[2] + mMatrix[13];

vOut[2] = mMatrix[2] \* vSrcVector[0] + mMatrix[6]

\* vSrcVector[1] + mMatrix[10] \* vSrcVector[2] + mMatrix[14];

}

// Rotates a vector using a 4x4 matrix. Translation column is ignored

void gltRotateVector(const GLTVector3 vSrcVector,

const GLTMatrix mMatrix, GLTVector3 vOut)

{

vOut[0] = mMatrix[0] \* vSrcVector[0] + mMatrix[4]

\* vSrcVector[1] + mMatrix[8] \* vSrcVector[2];

vOut[1] = mMatrix[1] \* vSrcVector[0] + mMatrix[5]

\* vSrcVector[1] + mMatrix[9] \* vSrcVector[2];

vOut[2] = mMatrix[2] \* vSrcVector[0] + mMatrix[6]

\* vSrcVector[1] + mMatrix[10] \* vSrcVector[2];

}

// Gets the three coefficients of a plane equation given three points on the plane.

void gltGetPlaneEquation(GLTVector3 vPoint1, GLTVector3 vPoint2, GLTVector3 vPoint3, GLTVector3 vPlane)

{

// Get normal vector from three points. The nor\_mal vector is the first three coefficients

// to the plane equation...

gltGetNormalVector(vPoint1, vPoint2, vPoint3,

vPlane);

// Final coefficient found by back substitution

vPlane[3] = -(vPlane[0] \* vPoint3[0] + vPlane[1]

\* vPoint3[1] + vPlane[2] \* vPoint3[2]);

}

// Determine the distance of a point from a plane, given the pointand the

// equation of the plane.

GLfloat gltDistanceToPlane(GLTVector3 vPoint, GLTVector4 vPlane)

{

return vPoint[0] \* vPlane[0] + vPoint[1] \* vPlane[1]

+ vPoint[2] \* vPlane[2] + vPlane[3];

}

// For best results, put this in a display list

// Draw a sphere at the origin

void gltDrawSphere(GLfloat fRadius, GLint iSlices,

GLint iStacks)

{

GLfloat drho = (GLfloat)(3.141592653589) /

(GLfloat)iStacks;

GLfloat dtheta = 2.0f \* (GLfloat)(3.141592653589)

/ (GLfloat)iSlices;

GLfloat ds = 1.0f / (GLfloat)iSlices;

GLfloat dt = 1.0f / (GLfloat)iStacks;

GLfloat t = 1.0f;

GLfloat s = 0.0f;

GLint i, j; // Looping variables

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

{

GLfloat rho = (GLfloat)i \* drho;

GLfloat srho = (GLfloat)(sin(rho));

GLfloat crho = (GLfloat)(cos(rho));

GLfloat srhodrho = (GLfloat)(sin(rho +

drho));

GLfloat crhodrho = (GLfloat)(cos(rho +

drho));

// Many sources of OpenGL sphere drawing code uses a triangle fan

// for the caps of the sphere. This however introduces texturing

// artifacts at the poles on some OpenGL im\_plementations

glBegin(GL\_TRIANGLE\_STRIP);

s = 0.0f;

for (j = 0; j <= iSlices; j++)

{

GLfloat theta = (j == iSlices) ?

0.0f : j \* dtheta;

GLfloat stheta = (GLfloat)(-

sin(theta));

GLfloat ctheta =

(GLfloat)(cos(theta));

GLfloat x = stheta \* srho;

GLfloat y = ctheta \* srho;

GLfloat z = crho;

glTexCoord2f(s, t);

glNormal3f(x, y, z);

glVertex3f(x \* fRadius, y \* fRadius, z \*

fRadius);

x = stheta \* srhodrho;

y = ctheta \* srhodrho;

z = crhodrho;

glTexCoord2f(s, t - dt);

s += ds;

glNormal3f(x, y, z);

glVertex3f(x \* fRadius, y \* fRadius, z \*

fRadius);

}

glEnd();

t -= dt;

}

}

// Initialize a frame of reference.

// Uses default OpenGL viewing position and orientation

void gltInitFrame(GLTFrame\* pFrame)

{

pFrame->vLocation[0] = 0.0f;

pFrame->vLocation[1] = 0.0f;

pFrame->vLocation[2] = 0.0f;

pFrame->vUp[0] = 0.0f;

pFrame->vUp[1] = 1.0f;

pFrame->vUp[2] = 0.0f;

pFrame->vForward[0] = 0.0f;

pFrame->vForward[1] = 0.0f;

pFrame->vForward[2] = -1.0f;

}

/////////////////////////////////////////////////////

//////////////

// Derives a 4x4 transformation matrix from a frame of reference

void gltGetMatrixFromFrame(GLTFrame\* pFrame, GLTMatrix mMatrix)

{

GLTVector3 vXAxis; // Derived X Axis

// Calculate X Axis

gltVectorCrossProduct(pFrame->vUp, pFrame->vForward, vXAxis);

// Just populate the matrix

// X column vector

memcpy(mMatrix, vXAxis, sizeof(GLTVector3));

mMatrix[3] = 0.0f;

// y column vector

memcpy(mMatrix + 4, pFrame->vUp, sizeof(GLTVector3));

mMatrix[7] = 0.0f;

// z column vector

memcpy(mMatrix + 8, pFrame->vForward, sizeof(GLTVector3));

mMatrix[11] = 0.0f;

// Translation/Location vector

memcpy(mMatrix + 12, pFrame->vLocation, sizeof(GLTVector3));

mMatrix[15] = 1.0f;

}

/////////////////////////////////////////////////////

///////////////

// Apply an actors transform given it's frame of ref\_erence

void gltApplyActorTransform(GLTFrame\* pFrame)

{

GLTMatrix mTransform;

gltGetMatrixFromFrame(pFrame, mTransform);

glMultMatrixf(mTransform);

}

/////////////////////////////////////////////////////

/////////////

// Apply a camera transform given a frame of refer\_ence. This is

// pretty much just an alternate implementation of gluLookAt using

// floats instead of doubles and having the forward vector specified

// instead of a point out in front of me.

void gltApplyCameraTransform(GLTFrame\* pCamera)

{

GLTMatrix mMatrix;

GLTVector3 vAxisX;

GLTVector3 zFlipped;

zFlipped[0] = -pCamera->vForward[0];

zFlipped[1] = -pCamera->vForward[1];

zFlipped[2] = -pCamera->vForward[2];

// Derive X vector

gltVectorCrossProduct(pCamera->vUp, zFlipped,

vAxisX);

// Populate matrix, note this is just the rota\_tion and is transposed

mMatrix[0] = vAxisX[0];

mMatrix[4] = vAxisX[1];

mMatrix[8] = vAxisX[2];

mMatrix[12] = 0.0f;

mMatrix[1] = pCamera->vUp[0];

mMatrix[5] = pCamera->vUp[1];

mMatrix[9] = pCamera->vUp[2];

mMatrix[13] = 0.0f;

mMatrix[2] = zFlipped[0];

mMatrix[6] = zFlipped[1];

mMatrix[10] = zFlipped[2];

mMatrix[14] = 0.0f;

mMatrix[3] = 0.0f;

mMatrix[7] = 0.0f;

mMatrix[11] = 0.0f;

mMatrix[15] = 1.0f;

// Do the rotation first

glMultMatrixf(mMatrix);

// Now, translate backwards

glTranslatef(-pCamera->vLocation[0], -pCamera->vLocation[1], -pCamera->vLocation[2]);

}

/////////////////////////////////////////////////////

////

// March a frame of reference forward. This simply moves

// the location forward along the forward vector.

void gltMoveFrameForward(GLTFrame\* pFrame, GLfloat

fStep)

{

pFrame->vLocation[0] += pFrame->vForward[0] \*

fStep;

pFrame->vLocation[1] += pFrame->vForward[1] \*

fStep;

pFrame->vLocation[2] += pFrame->vForward[2] \*

fStep;

}

/////////////////////////////////////////////////////

////

// Move a frame of reference up it's local Y axis

void gltMoveFrameUp(GLTFrame\* pFrame, GLfloat fStep)

{

pFrame->vLocation[0] += pFrame->vUp[0] \* fStep;

pFrame->vLocation[1] += pFrame->vUp[1] \* fStep;

pFrame->vLocation[2] += pFrame->vUp[2] \* fStep;

}

/////////////////////////////////////////////////////

///

// Move a frame of reference along it's local X axis

void gltMoveFrameRight(GLTFrame\* pFrame, GLfloat

fStep)

{

GLTVector3 vCross;

gltVectorCrossProduct(pFrame->vUp, pFrame->vForward, vCross);

pFrame->vLocation[0] += vCross[0] \* fStep;

pFrame->vLocation[1] += vCross[1] \* fStep;

pFrame->vLocation[2] += vCross[2] \* fStep;

}

/////////////////////////////////////////////////////

////

// Translate a frame in world coordinates

void gltTranslateFrameWorld(GLTFrame\* pFrame, GLfloat

x, GLfloat y, GLfloat z)

{

pFrame->vLocation[0] += x; pFrame->vLocation[1]

+= y; pFrame->vLocation[2] += z;

}

/////////////////////////////////////////////////////

////

// Translate a frame in local coordinates

void gltTranslateFrameLocal(GLTFrame\* pFrame, GLfloat

x, GLfloat y, GLfloat z)

{

gltMoveFrameRight(pFrame, x);

gltMoveFrameUp(pFrame, y);

gltMoveFrameForward(pFrame, z);

}

// Creates a shadow projection matrix out of the plane equation

// coefficients and the position of the light. The return value is stored

// in destMat

void gltMakeShadowMatrix(GLTVector3 vPoints[3],

GLTVector4 vLightPos, GLTMatrix destMat)

{

GLTVector4 vPlaneEquation;

GLfloat dot;

gltGetPlaneEquation(vPoints[0], vPoints[1],

vPoints[2], vPlaneEquation);

// Dot product of plane and light position

dot = vPlaneEquation[0] \* vLightPos[0] +

vPlaneEquation[1] \* vLightPos[1] +

vPlaneEquation[2] \* vLightPos[2] +

vPlaneEquation[3] \* vLightPos[3];

// Now do the projection

// First column

destMat[0] = dot - vLightPos[0] \* vPlaneEquation[0];

destMat[4] = 0.0f - vLightPos[0] \* vPlaneEquation[1];

destMat[8] = 0.0f - vLightPos[0] \* vPlaneEquation[2];

destMat[12] = 0.0f - vLightPos[0] \* vPlaneEquation[3];

// Second column

destMat[1] = 0.0f - vLightPos[1] \* vPlaneEquation[0];

destMat[5] = dot - vLightPos[1] \* vPlaneEquation[1];

destMat[9] = 0.0f - vLightPos[1] \* vPlaneEquation[2];

destMat[13] = 0.0f - vLightPos[1] \* vPlaneEquation[3];

// Third Column

destMat[2] = 0.0f - vLightPos[2] \* vPlaneEquation[0];

destMat[6] = 0.0f - vLightPos[2] \* vPlaneEquation[1];

destMat[10] = dot - vLightPos[2] \* vPlaneEquation[2];

destMat[14] = 0.0f - vLightPos[2] \* vPlaneEquation[3];

// Fourth Column

destMat[3] = 0.0f - vLightPos[3] \* vPlaneEquation[0];

destMat[7] = 0.0f - vLightPos[3] \* vPlaneEquation[1];

destMat[11] = 0.0f - vLightPos[3] \* vPlaneEquation[2];

destMat[15] = dot - vLightPos[3] \* vPlaneEquation[3];

}

/////////////////////////////////////////////////////

//////////////////////////

// Load a matrix with the Idenity matrix

void gltLoadIdentityMatrix(GLTMatrix m)

{

static GLTMatrix identity = { 1.0f, 0.0f, 0.0f,

0.0f,

0.0f, 1.0f,

0.0f, 0.0f,

0.0f, 0.0f,

1.0f, 0.0f,

0.0f, 0.0f,

0.0f, 1.0f };

memcpy(m, identity, sizeof(GLTMatrix));

}

/////////////////////////////////////////////////////

//////////////////////////

// Creates a 4x4 rotation matrix, takes radians NOT degrees

void gltRotationMatrix(float angle, float x, float y,

float z, GLTMatrix mMatrix)

{

float vecLength, sinSave, cosSave, oneMinusCos;

float xx, yy, zz, xy, yz, zx, xs, ys, zs;

// If NULL vector passed in, this will blow up...

if (x == 0.0f && y == 0.0f && z == 0.0f)

{

gltLoadIdentityMatrix(mMatrix);

return;

}

// Scale vector

vecLength = (float)sqrt(x \* x + y \* y + z \* z);

// Rotation matrix is normalized

x /= vecLength;

y /= vecLength;

z /= vecLength;

sinSave = (float)sin(angle);

cosSave = (float)cos(angle);

oneMinusCos = 1.0f - cosSave;

xx = x \* x;

yy = y \* y;

zz = z \* z;

xy = x \* y;

yz = y \* z;

zx = z \* x;

xs = x \* sinSave;

ys = y \* sinSave;

zs = z \* sinSave;

mMatrix[0] = (oneMinusCos \* xx) + cosSave;

mMatrix[4] = (oneMinusCos \* xy) - zs;

mMatrix[8] = (oneMinusCos \* zx) + ys;

mMatrix[12] = 0.0f;

mMatrix[1] = (oneMinusCos \* xy) + zs;

mMatrix[5] = (oneMinusCos \* yy) + cosSave;

mMatrix[9] = (oneMinusCos \* yz) - xs;

mMatrix[13] = 0.0f;

mMatrix[2] = (oneMinusCos \* zx) - ys;

mMatrix[6] = (oneMinusCos \* yz) + xs;

mMatrix[10] = (oneMinusCos \* zz) + cosSave;

mMatrix[14] = 0.0f;

mMatrix[3] = 0.0f;

mMatrix[7] = 0.0f;

mMatrix[11] = 0.0f;

mMatrix[15] = 1.0f;

}

/////////////////////////////////////////////////////

////

// Rotate a frame around it's local Y axis

void gltRotateFrameLocalY(GLTFrame\* pFrame, GLfloat

fAngle)

{

GLTMatrix mRotation;

GLTVector3 vNewForward;

gltRotationMatrix((float)gltDegToRad(fAngle),

0.0f, 1.0f, 0.0f, mRotation);

gltRotationMatrix(fAngle, pFrame->vUp[0], pFrame->vUp[1], pFrame->vUp[2], mRotation);

gltRotateVector(pFrame->vForward, mRotation,

vNewForward);

memcpy(pFrame->vForward, vNewForward, sizeof(GLTVector3));

}

/////////////////////////////////////////////////////

/////

// Rotate a frame around it's local X axis

void gltRotateFrameLocalX(GLTFrame\* pFrame, GLfloat

fAngle)

{

GLTMatrix mRotation;

GLTVector3 vCross;

gltVectorCrossProduct(vCross, pFrame->vUp,

pFrame->vForward);

gltRotationMatrix(fAngle, vCross[0], vCross[1],

vCross[2], mRotation);

GLTVector3 vNewVect;

// Inline 3x3 matrix multiply for rotation only

vNewVect[0] = mRotation[0] \* pFrame->vForward[0]

+ mRotation[4] \* pFrame->vForward[1] + mRotation[8] \*

pFrame->vForward[2];

vNewVect[1] = mRotation[1] \* pFrame->vForward[0]

+ mRotation[5] \* pFrame->vForward[1] + mRotation[9] \*

pFrame->vForward[2];

vNewVect[2] = mRotation[2] \* pFrame->vForward[0]

+ mRotation[6] \* pFrame->vForward[1] + mRotation[10]

\* pFrame->vForward[2];

memcpy(pFrame->vForward, vNewVect, sizeof(GLfloat) \* 3);

// Update pointing up vector

vNewVect[0] = mRotation[0] \* pFrame->vUp[0] +

mRotation[4] \* pFrame->vUp[1] + mRotation[8] \*

pFrame->vUp[2];

vNewVect[1] = mRotation[1] \* pFrame->vUp[0] +

mRotation[5] \* pFrame->vUp[1] + mRotation[9] \*

pFrame->vUp[2];

vNewVect[2] = mRotation[2] \* pFrame->vUp[0] +

mRotation[6] \* pFrame->vUp[1] + mRotation[10] \*

pFrame->vUp[2];

memcpy(pFrame->vUp, vNewVect, sizeof(GLfloat) \*

3);

}

/////////////////////////////////////////////////////

////////

// Rotate a frame around it's local Z axis

void gltRotateFrameLocalZ(GLTFrame\* pFrame, GLfloat

fAngle)

{

GLTMatrix mRotation;

// Only the up vector needs to be rotated

gltRotationMatrix(fAngle, pFrame->vForward[0],

pFrame->vForward[1], pFrame->vForward[2], mRotation);

GLTVector3 vNewVect;

vNewVect[0] = mRotation[0] \* pFrame->vUp[0] +

mRotation[4] \* pFrame->vUp[1] + mRotation[8] \*

pFrame->vUp[2];

vNewVect[1] = mRotation[1] \* pFrame->vUp[0] +

mRotation[5] \* pFrame->vUp[1] + mRotation[9] \*

pFrame->vUp[2];

vNewVect[2] = mRotation[2] \* pFrame->vUp[0] +

mRotation[6] \* pFrame->vUp[1] + mRotation[10] \*

pFrame->vUp[2];

memcpy(pFrame->vUp, vNewVect, sizeof(GLfloat) \*

3);

}

// Light and material Data

GLfloat fLightPos[4] = { -100.0f, 100.0f, 50.0f,

1.0f }; // Point source

GLfloat fNoLight[] = { 0.0f, 0.0f, 0.0f, 0.0f };

GLfloat fLowLight[] = { 0.25f, 0.25f, 0.25f, 1.0f };

GLfloat fBrightLight[] = { 1.0f, 1.0f, 1.0f, 1.0f };

GLTMatrix mShadowMatrix;

#define GROUND\_TEXTURE 0

#define TORUS\_TEXTURE 1

#define SPHERE\_TEXTURE 2

#define PERSONAL\_TEXTURE 3

#define NUM\_TEXTURES 4

GLuint textureObjects[NUM\_TEXTURES];

const char\* szTextureFiles[] = { "grass.tga", "rock.tga", "orb.tga", "brick.tga" };

/////////////////////////////////////////////////////

/////////////

// This function does any needed initialization on the rendering

// context.

void SetupRC()

{

GLTVector3 vPoints[3] = { { 0.0f, -0.4f, 0.0f },

{ 10.0f, -0.4f, 0.0f },

{ 5.0f, -0.4f, -5.0f } };

int iSphere;

int i;

// Grayish background

glClearColor(fLowLight[0], fLowLight[1], fLowLight[2], fLowLight[3]);

// Clear stencil buffer with zero, increment by one whenever anybody

// draws into it. When stencil function is enabled, only write where

// stencil value is zero. This prevents the transparent shadow from drawing

// over itself

glStencilOp(GL\_INCR, GL\_INCR, GL\_INCR);

glClearStencil(0);

glStencilFunc(GL\_EQUAL, 0x0, 0x01);

// Cull backs of polygons

glCullFace(GL\_BACK);

glFrontFace(GL\_CCW);

glEnable(GL\_CULL\_FACE);

glEnable(GL\_DEPTH\_TEST);

glEnable(GL\_MULTISAMPLE\_ARB);

// Setup light parameters

//glLightModelfv(GL\_LIGHT\_MODEL\_AMBIENT, fNoLight);

glLightModeli(GL\_LIGHT\_MODEL\_COLOR\_CONTROL, GL\_SEPARATE\_SPECULAR\_COLOR);

glLightfv(GL\_LIGHT0, GL\_AMBIENT, fLowLight);

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

glLightfv(GL\_LIGHT0, GL\_SPECULAR, fBrightLight);

glEnable(GL\_LIGHTING);

glEnable(GL\_LIGHT0);

// Calculate shadow matrix

gltMakeShadowMatrix(vPoints, fLightPos, mShadowMatrix);

// Mostly use material tracking

glEnable(GL\_COLOR\_MATERIAL);

glColorMaterial(GL\_FRONT,

GL\_AMBIENT\_AND\_DIFFUSE);

glMateriali(GL\_FRONT, GL\_SHININESS, 128);

gltInitFrame(&frameCamera); // Initialize the camera

// Randomly place the sphere inhabitants

for (iSphere = 0; iSphere < NUM\_SPHERES; iSphere++)

{

gltInitFrame(&spheres[iSphere]); // In\_itialize the frame

// Pick a random location between -20 and 20 at .1 increments

spheres[iSphere].vLocation[0] =

(float)((rand() % 400) - 200) \* 0.1f;

spheres[iSphere].vLocation[1] = 0.0f;

spheres[iSphere].vLocation[2] =

(float)((rand() % 400) - 200) \* 0.1f;

}

// Set up texture maps

glEnable(GL\_TEXTURE\_2D);

glGenTextures(NUM\_TEXTURES, textureObjects);

glTexEnvi(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE,

GL\_MODULATE);

for (i = 0; i < NUM\_TEXTURES; i++)

{

GLubyte\* pBytes;

GLint iWidth, iHeight, iComponents;

GLenum eFormat;

glBindTexture(GL\_TEXTURE\_2D, textureObjects[i]);

// Load this texture map

pBytes = (GLubyte

\*)gltLoadTGA(szTextureFiles[i], &iWidth, &iHeight,

&iComponents, &eFormat);

gluBuild2DMipmaps(GL\_TEXTURE\_2D, iComponents,

iWidth, iHeight, eFormat, GL\_UNSIGNED\_BYTE, pBytes);

free(pBytes);

glTexParameteri(GL\_TEXTURE\_2D,

GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D,

GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D,

GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D,

GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

}

}

/////////////////////////////////////////////////////

///////////////////

// Do shutdown for the rendering context

void ShutdownRC(void)

{

// Delete the textures

glDeleteTextures(NUM\_TEXTURES, textureObjects);

}

// For best results, put this in a display list

// Draw a torus (doughnut) at z = fZVal... torus is in xy plane

void gltDrawTorus(GLfloat majorRadius, GLfloat minorRadius, GLint numMajor, GLint numMinor)

{

GLTVector3 vNormal;

double majorStep = 2.0f \* GLT\_PI / numMajor;

double minorStep = 2.0f \* GLT\_PI / numMinor;

int i, j;

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

{

double a0 = i \* majorStep;

double a1 = a0 + majorStep;

GLfloat x0 = (GLfloat)cos(a0);

GLfloat y0 = (GLfloat)sin(a0);

GLfloat x1 = (GLfloat)cos(a1);

GLfloat y1 = (GLfloat)sin(a1);

glBegin(GL\_TRIANGLE\_STRIP);

for (j = 0; j <= numMinor; ++j)

{

double b = j \* minorStep;

GLfloat c = (GLfloat)cos(b);

GLfloat r = minorRadius \* c + majorRadius;

GLfloat z = minorRadius \*

(GLfloat)sin(b);

// First point

glTexCoord2f((float)(i) / (float)(numMajor),

(float)(j) / (float)(numMinor));

vNormal[0] = x0 \* c;

vNormal[1] = y0 \* c;

vNormal[2] = z / minorRadius;

gltNormalizeVector(vNormal);

glNormal3fv(vNormal);

glVertex3f(x0 \* r, y0 \* r, z);

glTexCoord2f((float)(i + 1) / (float)(numMajor),

(float)(j) / (float)(numMinor));

vNormal[0] = x1 \* c;

vNormal[1] = y1 \* c;

vNormal[2] = z / minorRadius;

glNormal3fv(vNormal);

glVertex3f(x1 \* r, y1 \* r, z);

}

glEnd();

}

}

/////////////////////////////////////////////////////

//////

// Draw the ground as a series of triangle strips

void DrawGround(void)

{

GLfloat fExtent = 20.0f;

GLfloat fStep = 1.0f;

GLfloat y = -0.4f;

GLint iStrip, iRun;

GLfloat s = 0.0f;

GLfloat t = 0.0f;

GLfloat texStep = 1.0f / (fExtent \* .075f);

glBindTexture(GL\_TEXTURE\_2D, textureObjects[GROUND\_TEXTURE]);

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S,

GL\_REPEAT);

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T,

GL\_REPEAT);

for (iStrip = -fExtent; iStrip <= fExtent; iStrip

+= fStep)

{

t = 0.0f;

glBegin(GL\_TRIANGLE\_STRIP);

for (iRun = fExtent; iRun >= -fExtent;

iRun -= fStep)

{

glTexCoord2f(s, t);

glNormal3f(0.0f, 1.0f, 0.0f); // All Point up

glVertex3f(iStrip, y, iRun);

glTexCoord2f(s + texStep, t);

glNormal3f(0.0f, 1.0f, 0.0f); // All Point up

glVertex3f(iStrip + fStep, y, iRun);

t += texStep;

}

glEnd();

s += texStep;

}

}

/////////////////////////////////////////////////////

//////////////////

// Draw random inhabitants and the rotating torus/sphere duo

void DrawInhabitants(GLint nShadow)

{

static GLfloat yRot = 0.0f; // Rotation angle for animation

GLint i;

if (nShadow == 0)

{

yRot += 0.5f;

glColor4f(1.0f, 1.0f, 1.0f, 1.0f);

}

else

glColor4f(1.0f, 1.0f, 1.0f, .10f); // Shadow color

// Draw the randomly located spheres

glBindTexture(GL\_TEXTURE\_2D, textureObjects[SPHERE\_TEXTURE]);

for (i = 0; i < NUM\_SPHERES; i++)

{

glPushMatrix();

gltApplyActorTransform(&spheres[i]);

gltDrawSphere(0.3f, 21, 11);

glPopMatrix();

}

glPushMatrix();

glTranslatef(0.0f, 1.0f, -2.5f);

glPushMatrix();

glRotatef(-yRot \* 2.0f, 0.0f, 1.0f, 0.0f);

glTranslatef(1.0f, 0.0f, 0.0f);

gltDrawSphere(0.1f, 21, 11);

glPopMatrix();

glBindTexture(GL\_TEXTURE\_2D, textureObjects[PERSONAL\_TEXTURE]);

glPushMatrix();

glRotatef(-yRot \* 2.0f, 1.0f, 0.0f, 0.0f);

glTranslatef(0.0f, 1.0f, 0.0f);

gltDrawSphere(0.1f, 21, 11);

glPopMatrix();

glPushMatrix();

glRotatef(-yRot \* 2.0f, 0.0f, 0.0f, 1.0f);

glTranslatef(-1.0f, 0.0f, 0.0f);

gltDrawSphere(0.1f, 21, 11);

glPopMatrix();

if (nShadow == 0)

{

// Torus alone will be specular

glMaterialfv(GL\_FRONT, GL\_SPECULAR,

fBrightLight);

}

glRotatef(yRot, 0.0f, 1.0f, 0.0f);

glBindTexture(GL\_TEXTURE\_2D, textureObjects[TORUS\_TEXTURE]);

gltDrawTorus(0.35, 0.15, 61, 37);

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

glPopMatrix();

}

// Called to draw scene

void RenderScene(void)

{

// Clear the window with current clearing color

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

glPushMatrix();

glTranslatef(0.0f, -1.0f, -1.0f);

gltApplyCameraTransform(&frameCamera);

// Position light before any other transformations

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

// Draw the ground

glColor3f(1.0f, 1.0f, 1.0f);

DrawGround();

// Draw shadows first

glDisable(GL\_DEPTH\_TEST);

glDisable(GL\_LIGHTING);

glDisable(GL\_TEXTURE\_2D);

glEnable(GL\_BLEND);

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

glEnable(GL\_STENCIL\_TEST);

glPushMatrix();

glMultMatrixf(mShadowMatrix);

DrawInhabitants(1);

glPopMatrix();

glDisable(GL\_STENCIL\_TEST);

glDisable(GL\_BLEND);

glEnable(GL\_LIGHTING);

glEnable(GL\_TEXTURE\_2D);

glEnable(GL\_DEPTH\_TEST);

// Draw inhabitants normally

DrawInhabitants(0);

glPopMatrix();

// Do the buffer Swap

glutSwapBuffers();

}

// Respond to arrow keys by moving the camera frame of reference

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

{

if (key == GLUT\_KEY\_UP)

gltMoveFrameForward(&frameCamera, 0.1f);

if (key == GLUT\_KEY\_DOWN)

gltMoveFrameForward(&frameCamera, -0.1f);

if (key == GLUT\_KEY\_LEFT)

gltRotateFrameLocalY(&frameCamera, 0.1);

if (key == GLUT\_KEY\_RIGHT)

gltRotateFrameLocalY(&frameCamera, -0.1);

// Refresh the Window

glutPostRedisplay();

}

/////////////////////////////////////////////////////

//////

// Called by GLUT library when idle (window not being

// resized or moved)

void TimerFunction(int value)

{

// Redraw the scene with new coordinates

glutPostRedisplay();

glutTimerFunc(3, TimerFunction, 1);

}

void ChangeSize(int w, int h)

{

GLfloat fAspect;

// Prevent a divide by zero, when window is too short

// (you cant make a window of zero width).

if (h == 0)

h = 1;

glViewport(0, 0, w, h);

fAspect = (GLfloat)w / (GLfloat)h;

// Reset the coordinate system before modifying

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

// Set the clipping volume

gluPerspective(35.0f, fAspect, 1.0f, 50.0f);

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

}

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

{

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB |

GLUT\_DEPTH | GLUT\_STENCIL);

glutInitWindowSize(800, 600);

glutCreateWindow("OpenGL SphereWorld Demo + Texture Maps");

glutReshapeFunc(ChangeSize);

glutDisplayFunc(RenderScene);

glutSpecialFunc(SpecialKeys);

SetupRC();

glutTimerFunc(33, TimerFunction, 1);

glutMainLoop();

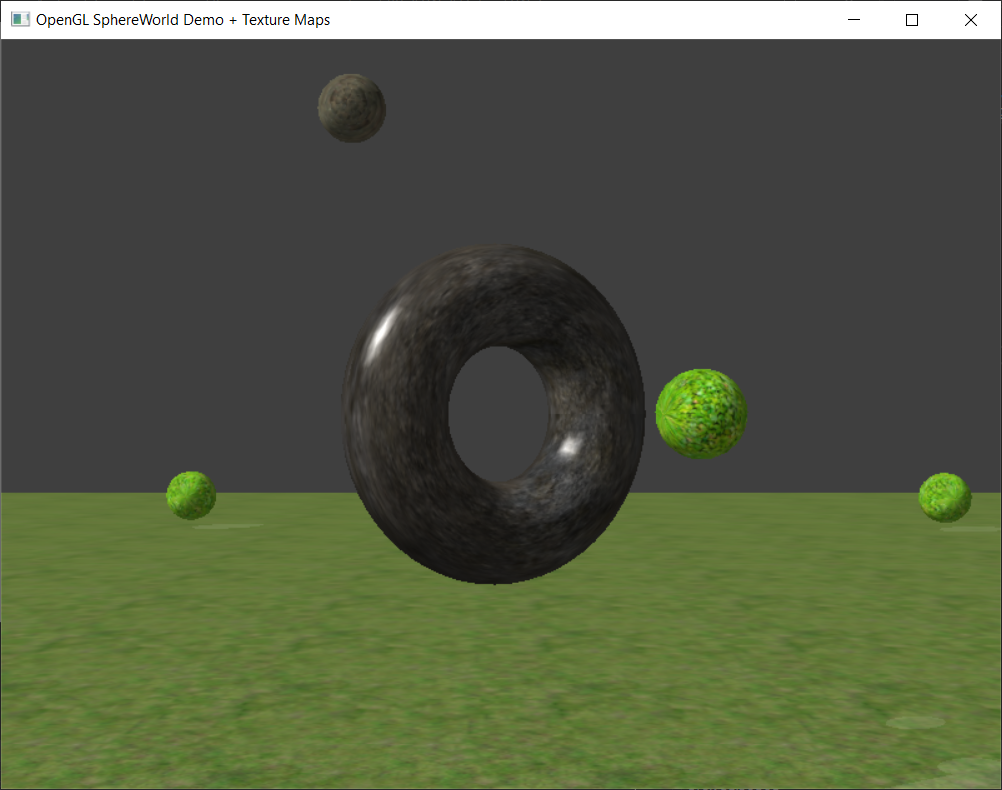
ShutdownRC();

return 0;

}

}

**Результат листинга 4:**



**Рисунок 4.** Результат листинга 4

**Вывод:** в ходе выполнения лабораторной работы были получены практические навыки работы с текстурами, их наложению на освещенные объекты.