Московский авиационный институт
Факультут прикладной физики и математики
Кафедра вычислительной матемитики и программирования

# Курсовая работа по курсу
# «Компьютерная графика»

Студент: Забарин Н.И.
Преподаватель:
Дата:
Оценка:
Подпись:

Москва, 2017

## Введение

Реалистичная визуализация водной поверхности один из самых эффективных способов сделать 3D приложение привлекательным. Но многие алгоритмы синтеза поверхности, как правило, сложны в реализации и требовательны к аппаратуре, поэтому к вопросу выбора алгоритма стоит подойти с особым вниманием.

Ранние методы визуализации воды в реальном времени, основывались на предположении о том, что водная поверхность плоская. Иллюзия волн создавалась за счет рельефного текстурирования с использованием заранее сгенерированных карт для нормалей и высот. Такой подход до сих пор часто используется.

В моей работе я попытался реализовать воду с помощью неплоской поверхности, изменяющейся в зависимости от времени.

## Описание работы

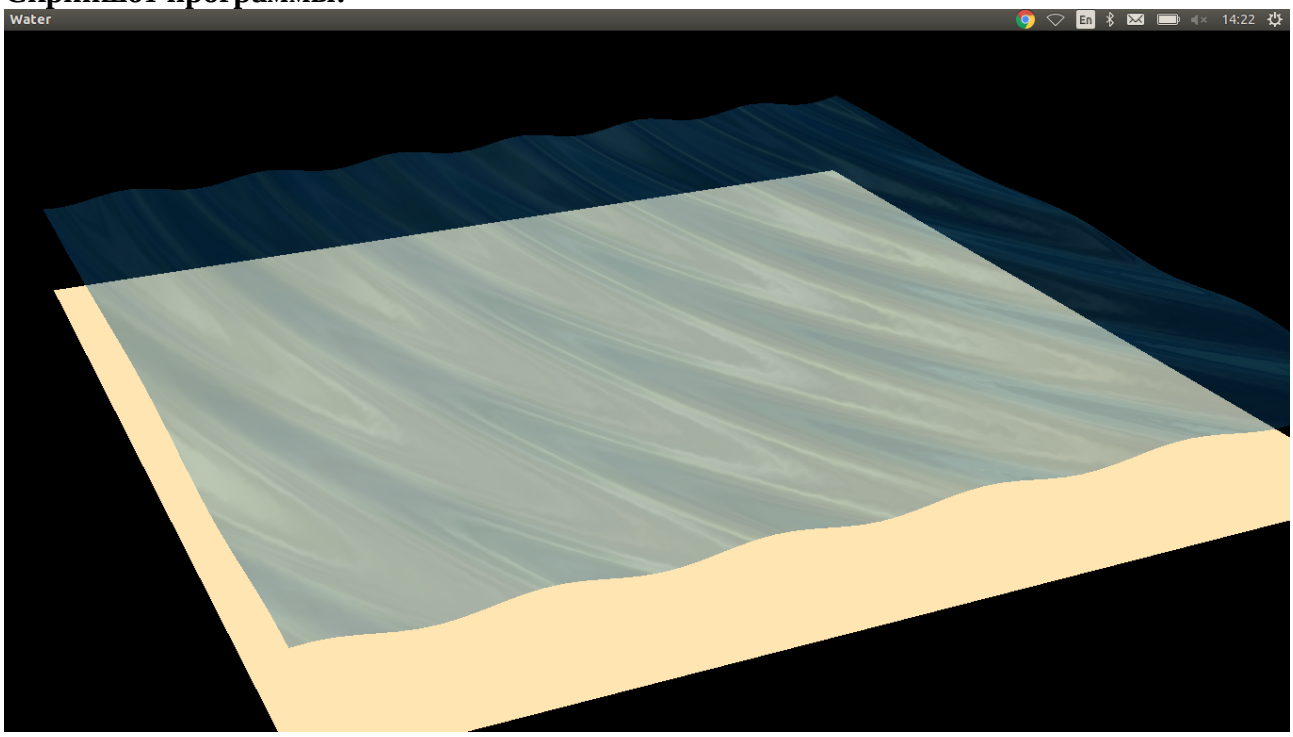
Функция задающая поверхность воды: 2sin(20sqrt((x-3)^2 + (y+1)^2) -4t). Поскольку t находится вне синуса она задает только смещение поверхности, что создает эффект волны.

Поверхность я составлял из множества треугольных граней.

Далее вся функция отрисовки картинки сводится к простым действиям: вычислить поверхность в данных момент времени, вычистить для всех граней вектора нормалей, натянуть текстуру.

Так же я дополнил сцену дном водоема с простенькой текстурой.

**Скриншот программы:**




**Листинг программы:**

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

#define  RESOLUTION 64
```

```
static GLuint texture;

static int left_click = GLUT_UP;
static int right_click = GLUT_UP;
static int wire_frame = 0;
static int normals = 0;
static int xold = 0;
static int yold = 0;
static float rotate_x = 30;
static float rotate_y = 15;
static float translate_z = 4;

static float surface[6 * RESOLUTION * (RESOLUTION + 1)];
static float normal[6 * RESOLUTION * (RESOLUTION + 1)];


static float z (const float x, const float y, const float t) {
   const float x2 = x - 3;
   const float y2 = y + 1;
   const float xx = x2 * x2;
   const float yy = y2 * y2;
   return ((2 * sinf (20 * sqrtf (xx + yy) - 4 * t)) / 200);
}
// Function to load a Jpeg file.
int         load_texture (const char * filename, unsigned char * dest, const int format, const unsigned int size) {
   FILE *fd;
   struct jpeg_decompress_struct cinfo;
   struct jpeg_error_mgr jerr;
   unsigned char * line;

   cinfo.err = jpeg_std_error (&jerr);
   jpeg_create_decompress (&cinfo);

   if (0 == (fd = fopen(filename, "rb"))) {
      return 1;
   }

   jpeg_stdio_src (&cinfo, fd);
   jpeg_read_header (&cinfo, TRUE);
   if ((cinfo.image_width != size) || (cinfo.image_height != size)) {
      return 1;
   }

   if (GL_RGB == format) {
      if (cinfo.out_color_space == JCS_GRAYSCALE) {
         return 1;
      }
   } else {
      if (cinfo.out_color_space != JCS_GRAYSCALE) {
         return 1;
      }
   }
   jpeg_start_decompress (&cinfo);

   while (cinfo.output_scanline < cinfo.output_height) {
      line = dest + (GL_RGB == format ? 3 * size : size) * cinfo.output_scanline;
      jpeg_read_scanlines (&cinfo, &line, 1);
   }
   jpeg_finish_decompress (&cinfo);
   jpeg_destroy_decompress (&cinfo);
   return 0;
}
```

```
// Function called to update rendering
void DisplayFunc (void) {
   const float t = glutGet (GLUT_ELAPSED_TIME) / 1000.;
   const float delta = 2. / RESOLUTION;
   const unsigned int length = 2 * (RESOLUTION + 1);
   const float xn = (RESOLUTION + 1) * delta + 1;
   unsigned int i, j;
   float x, y, l;
   unsigned int indice;
   unsigned int preindice;

   float v1x, v1y, v1z;
   float v2x, v2y, v2z;
   float v3x, v3y, v3z;
   float vax, vay, vaz;
   float vbx, vby, vbz;
   float nx, ny, nz;


   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   glLoadIdentity ();
   glTranslatef (0, 0, -translate_z);
   glRotatef (rotate_y, 1, 0, 0);
   glRotatef (rotate_x, 0, 1, 0);

   /* Vertices */
   for (j = 0; j < RESOLUTION; j++) {
      y = (j + 1) * delta - 1;
      for (i = 0; i <= RESOLUTION; i++) {
         indice = 6 * (i + j * (RESOLUTION + 1));

         x = i * delta - 1;
         surface[indice + 3] = x;
         surface[indice + 4] = z (x, y, t);
         surface[indice + 5] = y;
         if (j != 0) {
            /* Values were computed during the previous loop */
            preindice = 6 * (i + (j - 1) * (RESOLUTION + 1));
            surface[indice] = surface[preindice + 3];
            surface[indice + 1] = surface[preindice + 4];
            surface[indice + 2] = surface[preindice + 5];
         } else {
            surface[indice] = x;
            surface[indice + 1] = z (x, -1, t);
            surface[indice + 2] = -1;
         }
      }
   }

   /* Normals */
   for (j = 0; j < RESOLUTION; j++) {
      for (i = 0; i <= RESOLUTION; i++) {
         indice = 6 * (i + j * (RESOLUTION + 1));

         v1x = surface[indice + 3];
         v1y = surface[indice + 4];
         v1z = surface[indice + 5];

         v2x = v1x;
         v2y = surface[indice + 1];
         v2z = surface[indice + 2];
```

```
if (i < RESOLUTION) {
   v3x = surface[indice + 9];
   v3y = surface[indice + 10];
   v3z = v1z;
} else {
   v3x = xn;
   v3y = z (xn, v1z, t);
   v3z = v1z;
}

vax =  v2x - v1x;
vay =  v2y - v1y;
vaz =  v2z - v1z;

vbx = v3x - v1x;
vby = v3y - v1y;
vbz = v3z - v1z;

nx = (vby * vaz) - (vbz * vay);
ny = (vbz * vax) - (vbx * vaz);
nz = (vbx * vay) - (vby * vax);

l = sqrtf (nx * nx + ny * ny + nz * nz);
if (l != 0) {
   l = 1 / l;
   normal[indice + 3] = nx * l;
   normal[indice + 4] = ny * l;
   normal[indice + 5] = nz * l;
} else {
   normal[indice + 3] = 0;
   normal[indice + 4] = 1;
   normal[indice + 5] = 0;
}


if (j != 0) {
   /* Values were computed during the previous loop */
   preindice = 6 * (i + (j - 1) * (RESOLUTION + 1));
   normal[indice] = normal[preindice + 3];
   normal[indice + 1] = normal[preindice + 4];
   normal[indice + 2] = normal[preindice + 5];
} else {
   /*          v1x = v1x; */
   v1y = z (v1x, (j - 1) * delta - 1, t);
   v1z = (j - 1) * delta - 1;

   /*          v3x = v3x; */
   v3y = z (v3x, v2z, t);
   v3z = v2z;

   vax = v1x - v2x;
   vay = v1y - v2y;
   vaz = v1z - v2z;

   vbx = v3x - v2x;
   vby = v3y - v2y;
   vbz = v3z - v2z;

   nx = (vby * vaz) - (vbz * vay);
   ny = (vbz * vax) - (vbx * vaz);
   nz = (vbx * vay) - (vby * vax);

   l = sqrtf (nx * nx + ny * ny + nz * nz);
```

```
            if (l != 0) {
               l = 1 / l;
               normal[indice] = nx * l;
               normal[indice + 1] = ny * l;
               normal[indice + 2] = nz * l;
            } else {
               normal[indice] = 0;
               normal[indice + 1] = 1;
               normal[indice + 2] = 0;
            }
         }
      }
   }
   /* The ground */
   glPolygonMode (GL_FRONT_AND_BACK, GL_FILL);
   glDisable (GL_TEXTURE_2D);
   glColor3f (1, 0.9, 0.7);
   glBegin (GL_TRIANGLE_FAN);
   glVertex3f (-1, 0, -1);
   glVertex3f (-1, 0,  1);
   glVertex3f ( 1, 0,  1);
   glVertex3f ( 1, 0, -1);
   glEnd ();

   glTranslatef (0, 0.2, 0);

   /* Render wireframe? */
   if (wire_frame != 0) {
      glPolygonMode (GL_FRONT_AND_BACK, GL_LINE);
   }

   /* The water */
   glEnable (GL_TEXTURE_2D);
   glColor3f (1, 1, 1);
   glEnableClientState (GL_NORMAL_ARRAY);
   glEnableClientState (GL_VERTEX_ARRAY);
   glNormalPointer (GL_FLOAT, 0, normal);
   glVertexPointer (3, GL_FLOAT, 0, surface);

   for (i = 0; i < RESOLUTION; i++) {
      glDrawArrays (GL_TRIANGLE_STRIP, i * length, length);
   }

   /* Draw normals? */
   if (normals != 0) {
      glDisable (GL_TEXTURE_2D);
      glColor3f (1, 0, 0);
      glBegin (GL_LINES);
      for (j = 0; j < RESOLUTION; j++) {
         for (i = 0; i <= RESOLUTION; i++) {
            indice = 6 * (i + j * (RESOLUTION + 1));
            glVertex3fv (&(surface[indice]));
            glVertex3f (surface[indice] + normal[indice] / 50,
                        surface[indice + 1] + normal[indice + 1] / 50,
                        surface[indice + 2] + normal[indice + 2] / 50);
         }
      }

      glEnd ();
   }

   /* End */
   glFlush ();
```

```
    glutSwapBuffers ();
    glutPostRedisplay ();
}
// Function called when the window is created or resized
void ReshapeFunc (int width, int height) {
    glMatrixMode(GL_PROJECTION);

    glLoadIdentity ();
    gluPerspective (20, width / (float) height, 0.1, 15);
    glViewport (0, 0, width, height);

    glMatrixMode(GL_MODELVIEW);
    glutPostRedisplay();
}
// Function called when a key is hit
void KeyboardFunc (unsigned char key, int x, int y) {
    xold = x;
    yold = y;
    switch (key) {
        case 'q': case 'Q':
            exit (0);
            break;
        case 't': case 'T':
            wire_frame = !wire_frame;
            break;
        case 'n': case 'N':
            normals = !normals;
            break;
    }
}
// Function called when a mouse button is hit
void MouseFunc (int button, int state, int x, int y) {
    if (GLUT_LEFT_BUTTON == button) {
        left_click = state;
    }
    if (GLUT_RIGHT_BUTTON == button) {
        right_click = state;
    }
    xold = x;
    yold = y;
}
// Function called when the mouse is moved
void MotionFunc (int x, int y) {
    if (GLUT_DOWN == left_click) {
        rotate_y = rotate_y + (y - yold) / 5.0;
        rotate_x = rotate_x + (x - xold) / 5.0;
        if (rotate_y > 90) {
            rotate_y = 90;
        }
        if (rotate_y < -90) {
            rotate_y = -90;
        }
        glutPostRedisplay ();
    }
    if (GLUT_DOWN == right_click) {
        rotate_x = rotate_x + (x - xold) / 5.;
        translate_z = translate_z +
                    (yold - y) / 50.;
        if (translate_z < 0.5) {
            translate_z = 0.5;
        }
        if (translate_z > 10) {
            translate_z = 10;
```

```
      }
      glutPostRedisplay ();
   }
   xold = x;
   yold = y;
}



int         main (int narg, char ** args) {
   unsigned char total_texture[4 * 256 * 256];
   unsigned char alpha_texture[256 * 256];
   unsigned char caustic_texture[3 * 256 * 256];
   unsigned int i;

   /* Creation of the window */
   glutInit (&narg, args);
   glutInitDisplayMode (GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
   glutInitWindowSize (500, 500);
   glutCreateWindow ("Water");

   /* OpenGL settings */
   glClearColor (0, 0, 0, 0);
   glEnable (GL_DEPTH_TEST);
   glEnable (GL_BLEND);
   glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
   glPolygonMode (GL_FRONT_AND_BACK, GL_FILL);
   glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);

   /* Texture loading  */
   glGenTextures (1, &texture);
   if (load_texture ("alpha.jpg", alpha_texture, GL_ALPHA, 256) != 0 ||
         load_texture ("reflection.jpg", caustic_texture, GL_RGB, 256) != 0) {
      return 1;
   }
   for (i = 0; i < 256 * 256; i++) {
      total_texture[4 * i] = caustic_texture[3 * i];
      total_texture[4 * i + 1] = caustic_texture[3 * i + 1];
      total_texture[4 * i + 2] = caustic_texture[3 * i + 2];
      total_texture[4 * i + 3] = alpha_texture[i];
   }
   glBindTexture (GL_TEXTURE_2D, texture);
   gluBuild2DMipmaps (GL_TEXTURE_2D, GL_RGBA, 256, 256, GL_RGBA,
                      GL_UNSIGNED_BYTE,  total_texture);
   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);
   glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
   glEnable (GL_TEXTURE_GEN_S);
   glEnable (GL_TEXTURE_GEN_T);
   glTexGeni (GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
   glTexGeni (GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);

   /* Declaration of the callbacks */
   glutDisplayFunc (&DisplayFunc);
   glutReshapeFunc (&ReshapeFunc);
   glutKeyboardFunc (&KeyboardFunc);
   glutMouseFunc (&MouseFunc);
   glutMotionFunc (&MotionFunc);

   /* Loop */
   glutMainLoop ();
```

```
    /* Never reached */
    return 0;
}
```