Lập trình Shader

Mục lục

Mục tiêu

- Làm quen với ngôn ngữ lập trình GLSL phiên bản 1.10 và chuỗi dựng hình của OpenGL 2.0.
- Giới thiệu thư viện lập trình GLEW để nạp các module mở rộng cho thư viện OpenGL, cho phép sử dụng v*ertex shader*, f*ragment shader* và s*hader program* trong chương trình ứng dụng.
- Xây dựng các chương trình v*ertex shader* và f*ragment shader* để tô bóng các vật thể 3D.

Yêu cầu

Kiến thức về các phép biến đổi hình ảnh (tịnh tiến, xoay, tỷ lệ), các kỹ thuật lập trình OpenGL cơ bản và thư viện lập trình FreeGLUT, GLEW. Sử dụng thành thạo ngôn ngữ lập trình C/C++ và các khái niệm cơ bản của ngôn ngữ lập trình GLSL [1].

Thư viện quản lý chương trình Shader

Để tiện cho việc sử dụng các chương trình shader trong các bài tập, tránh phải viết đi viết lại nhiều lần các câu lệnh OpenGL để quản lý các đối tượng *vertex* và *fragment shader*, ta thiết kế một thư viện hỗ trợ các công việc thường hay phải thực hiện khi sử dụng shader như nạp, biên dịch và liên kết. Các tính năng của thư viện quản lý chương trình shader gồm có:

- Định nghĩa kiểu dữ liệu vector để tương tác dễ dàng với shader.

- Tạo/hủy một đối tượng chương trình shader.
- Nạp, biên dịch mã nguồn cho đối tượng *vertex shader* và *fragment shader*.
- Liên kết *vertex shader* và *fragment shader* vào chương trình shader và kích hoạt chương trình shader.
- Hiển thị các thông báo lỗi biên dịch shader (nếu có)
- Truy xuất đến các biến *uniform* cơ bản bên trong *vertex shader* và *fragment shader*. Các biến *uniform* được hỗ trợ hiện tại chỉ gồm biến thực, bộ 3 tọa độ (x, y, z) và vector 3 chiều. Ta có thể mở rộng để hỗ trợ đầy đủ hơn các biến *uniform* và *attribute* để khả thi hơn trong thực tế (biến nguyên, biến bool, vector 4 chiều, ma trận 2×2, 3×3 và 4×4...

- Tập tin SHADER.H

```
#ifndef __SHADER_H__
#define __SHADER_H__
#include <GL/glew.h>
/* Định nghĩa kiểu dữ liệu vec3 tương đương với GLSL */
typedef struct {
    union { struct { float x, y, z; };      // truy xuất thông qua xyz
            struct { float r, g, b; };      // truy xuất thông qua rgb
            float data[3];                  // truy xuất thông qua chỉ số
    };
} vec3;
/* kiểu đối tượng shader cần thao tác */
typedef enum glslShader { VERTEX_SHADER, FRAGMENT_SHADER } GLSL_SHADER;
/* đối tượng chương trình shader GLSL*/
typedef struct glslProgram {
    GLuint program;
    GLboolean linked;
} GLSL_PROGRAM;
/* các hàm xử lý trên đối tượng chương trình shader */
GLSL_PROGRAM * glslCreate();
void      glslDestroy(      GLSL_PROGRAM * p);
GLboolean glslCompileFile(  GLSL_PROGRAM * p, GLSL_SHADER type,
                            const GLchar * fileName);
GLboolean glslCompileString(GLSL_PROGRAM * p, GLSL_SHADER type,
                            const GLchar * source);
GLboolean glslLink(         GLSL_PROGRAM * p);
GLboolean glslActivate(     GLSL_PROGRAM * p);
void      glslDeactivate(   GLSL_PROGRAM * p);
/* các hàm thao tác trên biến uniform */
GLint     glslGetUniform(   GLSL_PROGRAM * p, const GLchar * name);
GLboolean glslSetUniform1f( GLSL_PROGRAM * p, const GLchar * name, GLfloat v);
GLboolean glslSetUniform3f( GLSL_PROGRAM * p, const GLchar * name,
                            GLfloat x, GLfloat y, GLfloat z);
GLboolean glslSetUniform3fv(GLSL_PROGRAM * p, const GLchar * name, const vec3 v);

#endif
```

- Tập tin SHADER.C

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include "shader.h"

GLSL_PROGRAM * glslCreate() {
    GLSL_PROGRAM * p;
    if (!(p = (GLSL_PROGRAM *) malloc(sizeof(GLSL_PROGRAM)))) return NULL;
    p->linked = GL_FALSE;
    if (!(p->program = glCreateProgram())) {
        free(p);
        p = NULL;
    }
    return p;
}

void glslDestroy(GLSL_PROGRAM * p) {
    if (!p) return;
    glUseProgram(0);
    glDeleteProgram(p->program);
    free(p);
}

GLboolean glslCompileFile(GLSL_PROGRAM * p, GLSL_SHADER type, const GLchar * fileName)
{
    FILE        * fp = fopen(fileName, "rt");
    GLchar      * source = NULL;
    GLint       count = 0;
    GLboolean   result;

    if (!fp || !p) return GL_FALSE;
    fseek(fp, 0, SEEK_END);
    count = ftell(fp);
    rewind(fp);
    if (count > 0) {
        source = (GLchar *) malloc(sizeof(GLchar) * (count+1));
        count = fread(source, sizeof(GLchar), count, fp);
        source[count] = '\0';
    }
    fclose(fp);
    result = glslCompileString(p, type, source);
    free(source);
    return result;
}

GLboolean glslCompileString(GLSL_PROGRAM * p, GLSL_SHADER type, const GLchar * source)
{
    GLuint shader = 0;
    GLint result = GL_FALSE;

    if (!p) return GL_FALSE;
    switch (type) {
    case VERTEX_SHADER:   shader = glCreateShader(GL_VERTEX_SHADER);   break;
    case FRAGMENT_SHADER: shader = glCreateShader(GL_FRAGMENT_SHADER); break;
    }
```

```
    glShaderSource(shader, 1, &source, NULL);
    glCompileShader(shader);
    glGetShaderiv(shader, GL_COMPILE_STATUS, &result);
    if (result == GL_FALSE) {
        int logLength = 0;
        glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &logLength);
        if (logLength) {
            char * log = (char * ) malloc(logLength);
            int size = 0;
            glGetShaderInfoLog(shader, logLength, &size, log);
            printf("%s\n", log);
            free(log);
        }
    }
    else {
        glAttachShader(p->program, shader);
        glDeleteShader(shader);
    }
    return result;
}

GLboolean glslLink(GLSL_PROGRAM * p) {
    GLint result = GL_FALSE;
    if (!p) return GL_FALSE;
    if (p->linked) return GL_TRUE;
    glLinkProgram(p->program);
    glGetProgramiv(p->program, GL_LINK_STATUS, &result);
    p->linked = GL_TRUE;
    return result;
}

GLboolean glslActivate(GLSL_PROGRAM * p) {
    if (!p || !p->linked) return GL_FALSE;
    glUseProgram(p->program);
    return GL_TRUE;
}

void glslDeactivate(GLSL_PROGRAM * p) {
    glUseProgram(0);
}

GLint glslGetUniform(GLSL_PROGRAM * p, const GLchar * name) {
    return glGetUniformLocation(p->program, name);
}

GLboolean glslSetUniform1f(GLSL_PROGRAM * p, const GLchar * name, GLfloat v) {
    GLint loc = glslGetUniform(p, name);
    if (loc < 0) return GL_FALSE;
    glUniform1f(loc, v);
    return GL_TRUE;
}

GLboolean glslSetUniform3f( GLSL_PROGRAM * p, const GLchar * name,
```

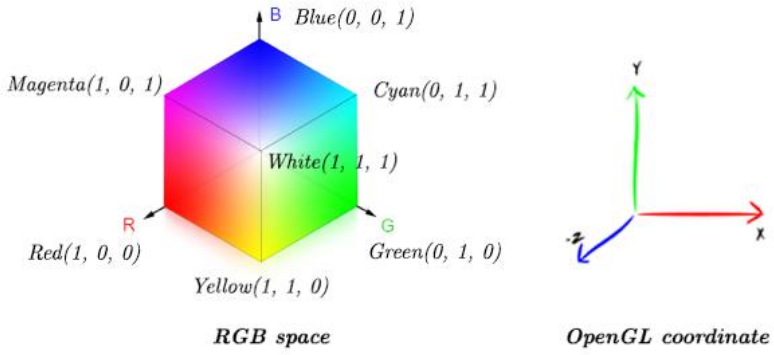
```
                            GLfloat x, GLfloat y, GLfloat z) {
    GLint loc = glslGetUniform(p, name);
    if (loc < 0) return GL_FALSE;
    glUniform3f(loc, x, y, z);
    return GL_TRUE;
}

GLboolean glslSetUniform3fv(GLSL_PROGRAM * p, const GLchar * name, const vec3 v) {
    return glslSetUniform3f(p, name, v.x, v.y, v.z);
}
```

Bài tập

Bài 1: Shader ánh xạ vị trí RGB

Các đỉnh của một vật thể đều có tọa độ 3 chiều (x, y, z) trong không gian tạo bởi các trục *Ox*, *Oy* và *Oz*. Ta có thể biểu diễn các tọa độ này thông qua không gian màu RGB trong một khối lập phương như Hình 1. Hãy thực hiện các yêu cầu sau:

- Hãy viết *vertex shader* và *fragment shader* thực hiện việc ánh xạ các tọa độ của vật thể này vào không gian màu RGB, với tọa độ mỗi đỉnh của đối tượng sẽ là một tọa độ (r, g, b) tương ứng trong không gian RGB.
- Xây dựng một ứng dụng OpenGL hiển thị một vật thể sử dụng các shader đã tạo.
- Cho phép xoay đối tượng bằng chuột để quan sát, sử dụng các phím $1, 2, 3, 4$ để thay đổi các vật thể 3D bao gồm vật thể lập phương, vật thể cầu, vật thể torus và ấm trà. Sử dụng mã nguồn thư viện `SHADER` đã được cung cấp để nạp, biên dịch và liên kết các shader trong ứng dụng.



Hình 1. Không gian RGB và hệ trục tọa độ OpenGL.

Hướng dẫn:

- Chuẩn hóa tọa độ các đỉnh của vật thể về đoạn $(0, 1)$.
- Tịnh tiến các đỉnh này một đoạn $(x = 0.5, y = 0.5, z = 0.5)$ để đưa toàn bộ vật thể về phía dương của không gian để cho $x \geq 0, y \geq 0, z \geq 0, \forall x, y, z$.
- Đảo trục để tọa độ (z, y, x) tương đương với giá trị màu (r, g, b).

Do ta thực hiện ánh xạ 1:1 mỗi đỉnh trên vật thể vào không gian màu RGB, shader này còn được gọi là shader ánh xạ vị trí (*Position Mapping*).

$$f\colon\ z, y, x\ \rightarrow (r, g, b)$$

Hình 2. Kết xuất của Bài 1: Shader ánh xạ vị trí

- Tập tin `VERTEX.GLSL`

```
// POSITION TO RGB MAPPING: VERTEX SHADER
varying vec4 color; // gía trị màu dùng cho fragment shader
void main() {
    // chuẩn hóa tọa độ đỉnh và ánh xạ vào không gian RGB
    color = (normalize(gl_Vertex.zyxw) + vec4(0.5, 0.5, 0.5, 1.0));
    // tính tọa độ chiếu cho mỗi đỉnh
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

- Tập tin `FRAGMENT.GLSL`

```
// POSITION TO RGB MAPPING: FRAGMENT SHADER
varying vec4 color; // giá trị màu đã tính từ vertex shader
void main() {
    // gán giá trị màu này cho pixel
    gl_FragColor = color;
}
```

- Tập tin `MAIN.C`

```
#include <GL/glew.h>
#include <GL/glut.h>
#include <stdio.h>
#include <stdlib.h>
#include "util.h"
#include "shader.h"

#ifdef _MSC_VER                         /* for MSVC */
#   pragma comment (lib, "opengl32.lib")
#   pragma comment (lib, "glu32.lib")
#   pragma comment (lib, "glew32.lib")
#   pragma comment (lib, "freeglut.lib")
#endif

#define WND_WIDTH   640                   /* chiều rộng cửa sổ */
#define WND_HEIGHT  360                   /* chiều cao cửa sổ */
#define WND_TITLE   "Lab05-1"             /* tiêu đề */
#define VS_FILE     "vertex.glsl"         /* tập tin mã nguồn vertex shader */
```

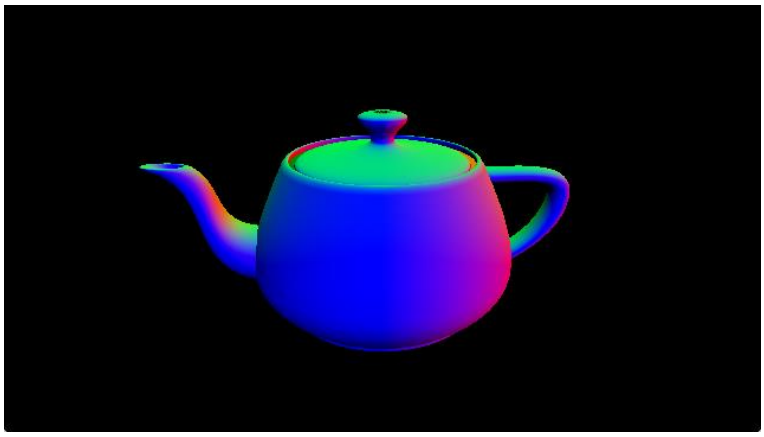
```
#define FS_FILE     "fragment.glsl"         /* tập tin mã nguồn fragment shader */

typedef enum objectMode { CUBE, SPHERE, TORUS, TEAPOT } OBJECT_MODE;

/* các biến toàn cục */
float xAngle    = 0.0f;                     /* các góc xoay vật thể */
float yAngle    = 0.0f;
float zAngle    = 0.0f;
int xMotion = 0;                            /* theo dõi chuyển động chuột */
int yMotion = 0;

GLSL_PROGRAM * prog = NULL;                 /* chương trình shader */
OBJECT_MODE obj = TEAPOT;                   /* vật thể hiển thị hiện thời */

void resize(int width, int height) {  /* thay đổi kích thước cửa sổ */
    if (height == 0) height = 1;
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glViewport(0, 0, width, height);
    gluPerspective(45.0f, (float) width/height, 1.0f, 100.0f);
    glMatrixMode(GL_MODELVIEW);
}

void render() {  /* dựng hình */
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt(0.0f, 0.0f, 4.0f,  0.0f, 0.0f, 0.0f,  0.0f, 1.0f, 0.0f);
    glRotatef(xAngle, 1.0f, 0.0f, 0.0f);
    glRotatef(yAngle, 0.0f, 1.0f, 0.0f);
    glRotatef(zAngle, 0.0f, 0.0f, 1.0f);

    switch(obj) {
    case CUBE:      glutSolidCube(1.5f); break;
    case SPHERE:    glutSolidSphere(1.0f, 24, 24); break;
    case TORUS:     glutSolidTorus(0.5, 1.0f, 24, 24); break;
    case TEAPOT:    glutSolidTeapot(1.0f); break;
    }
    glutSwapBuffers();
}

void input(unsigned char key, int x, int y) { /* xử lý sự kiện bàn phím */
    switch (key) {
    case 27: exit(0);
    case '1':   obj = CUBE;   break;
    case '2':   obj = SPHERE; break;
    case '3':   obj = TORUS;  break;
    case '4':   obj = TEAPOT; break;
    default:    break;
    }
}

void mouse(int button, int state, int x, int y) { /* xử lý sự kiện chuột */
```

```
    if (state == 0 && button == 0) { /* phím trái nhấn */
        xMotion = x;
        yMotion = y;
    }
}

void motion (int x, int y) { /* hàm xử lý chuyển động chuột */
    if (xMotion) {
        if (xMotion > x) yAngle -= 2.0f;
        if (xMotion < x) yAngle += 2.0f;
        xMotion = x;
    }
    if (yMotion) {
        if (yMotion > y) xAngle -= 1.0f;
        if (yMotion < y) xAngle += 1.0f;
        yMotion = y;
    }
}

void idle() { /* hàm xử lý trong thời gian chờ */
    glutPostRedisplay();
}

GLboolean init(int argc, char ** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowPosition(-1, -1);
    glutInitWindowSize(WND_WIDTH, WND_HEIGHT);
    glutCreateWindow(WND_TITLE);
    glutDisplayFunc(render);
    glutIdleFunc(idle);
    glutReshapeFunc(resize);
    glutKeyboardFunc(input);
    glutMotionFunc(motion);
    glutMouseFunc(mouse);

    if (GLEW_OK != glewInit()) return GL_FALSE;
    if (!(prog = glslCreate())) printf("No GLSL supported.\n");
    glslCompileFile(prog, VERTEX_SHADER, VS_FILE));
    glslCompileFile(prog, FRAGMENT_SHADER, FS_FILE));
    glslLink(prog);
    glslActivate(prog);
    glEnable(GL_DEPTH_TEST);
    return GL_TRUE;
}

void done() {
    glslDestroy(prog);
}

void run() {
    glutMainLoop();
}
```

```
int main(int argc, char ** argv) {
    init(argc, argv);
    run();
    return 0;
}
```

Bài 2: Shader ánh xạ pháp tuyến RGB

Tương tự Bài 1: Shader ánh xạ vị trí, sử dụng lại mã nguồn của bài tập này, hãy viết *vertex shader* và *fragment shader* để ánh xạ hướng của pháp tuyến các mặt trên vật thể vào không gian RGB (Hình 3).



Hình 3. Kết xuất của Bài 2: Shader ánh xạ pháp tuyến

- Tập tin `VERTEX.GLSL`

```
// NORMAL TO RGB MAPPING: VERTEX SHADER
varying vec3 normal; // vector pháp tuyến nội suy được sử dụng cho fragment shader
void main() {
    normal = normalize(gl_NormalMatrix * gl_Normal);
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

- Tập tin `FRAGMENT.GLSL`

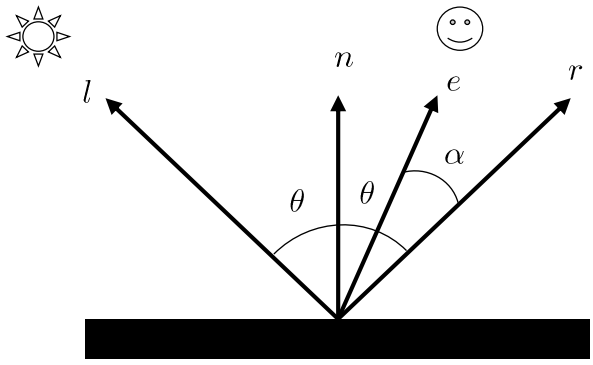
```
// NORMAL TO RGB MAPPING: FRAGMENT SHADER
varying vec3 normal;  // vector pháp tuyến nhận từ vertex shader
void main() {
    // ánh xạ hướng pháp tuyến vào không gian RGB
    gl_FragColor = vec4(normal, 1.0);
}
```

Bài 3: Shader chiếu sáng theo mô hình Phong.

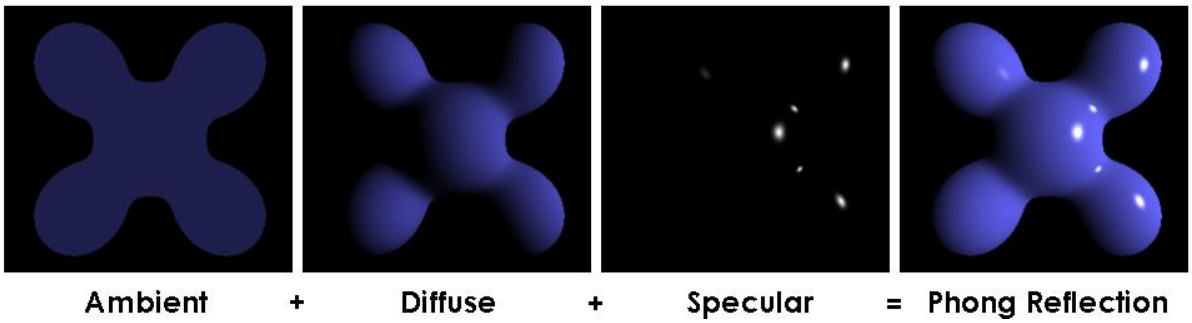
Mô hình chiếu sáng Phong (*Phong Illumination Model*) [2] (Hình 4) là mô hình được sử dụng rộng rãi trong lĩnh vực đồ họa máy tính. Mô hình này cải tiến việc chiếu sáng các vật thể dựa trên việc nội suy vector pháp tuyến tại từng điểm trên bề mặt các tam giác trên vật thể đó, nhằm tạo ra hiệu quả chiếu sáng trung thực hơn mô hình chiếu sáng trước đó do Henri Gouraud đề xuất [3], vốn chỉ nội suy giá trị cường độ sáng của các điểm ảnh trong tam giác dựa trên pháp tuyến tại 3 đỉnh của tam giác đó. Kỹ thuật chiếu sáng với mô hình này còn có tên gọi là chiếu sáng ở mức pixel (*per-pixel lighting*).

Mô hình chiếu sáng Phong được xây dựng dựa trên phương trình chiếu sáng sau đây, Hình 5 cho ta cái nhìn trực quan hơn về phương trình chiếu sáng này. Trong đó, K_a là hệ số của thành phần *ambient*, I_a là giá trị màu ambient. Tương tự, K_d, I_d, K_s, I_s lần lượt là hệ số và giá trị màu của các thành phần *diffuse* và *specular*. Giá trị *sh* biểu diễn hệ số trơn láng của bề mặt chiếu sáng.

$$I = K_a I_a + K_d \max\ 0, n \cdot l\ + K_s I_s \max\ 0, r \cdot l\ ^{sh}$$



Hình 4. Mô hình chiếu sáng Phong



Hình 5. Trực quan hóa các thành phần trong phương trình Phong

Hình 6. Kết xuất của Bài 3: Shader chiếu sáng theo mô hình Phong.

Mở rộng mã nguồn Bài 1: Shader ánh xạ vị trí RGB để hỗ trợ thêm các tính năng sau:

- Cho phép di chuyển góc nhìn (*camera*) bằng các phím điều khiển `W`, `A`, `S`, `D`.
- Di chuyển vị trí nguồn sáng đầu tiên của OpenGLbằng các phím `X`, `Y`, `Z`. Thể hiện nguồn sáng bằng một hình cầu bán kính nhỏ tại đúng vị trí.

- Thay đổi giá trị của hệ số `Ka`: phím `<`, `>`.
- Thay đổi giá trị của hệ số `Kd`: phím `[`, `]`.
- Thay đổi giá trị của hệ số `Ks`: phím `{`, `}`.
- Thay đổi hệ số bóng bề mặt `shininess`: phím `H`, `h`.
- Thay đổi giá trị màu `diffuseColor`: phím `R`, `r`, `G`, `g`, `B`, `b`.

- Tập tin `VERTEX.GLSL`

```
// PHONG SHADING: VERTEX SHADER
varying vec3 N;         // vector pháp tuyến nội suy, chuyển cho fragment shader
varying vec3 v;         // vị trí của đỉnh trong không gian góc nhìn

void main(void) {
    // tính tọa độ đỉnh trong không gian góc nhìn
    v = vec3(gl_ModelViewMatrix * gl_Vertex);
    // nội suy pháp tuyến
    N = normalize(gl_NormalMatrix * gl_Normal);
    // tính tọa độ chiếu của đỉnh
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

- Tập tin `FRAGMENT.GLSL`

```
// PHONG SHADING: FRAGMENT SHADER FOR ONE LIGHT
uniform vec3 lightPosition; // vị trí nguồn sáng, từ chương trình chính
uniform vec3 ambientColor;  // màu ambient, từ chương trình chính
uniform vec3 diffuseColor;  // màu diffuse, từ chương trình chính
uniform vec3 specularColor; // màu specular, từ chương trình chính
uniform float Ka;           // hệ số ambient, từ chương trình chính
uniform float Kd;           // hệ số diffuse, từ chương trình chính
uniform float Ks;           // hệ số specular, từ chương trình chính
uniform float shininess;    // hệ số bóng bề mặt, từ chương trình chính
varying vec3 N;             // pháp tuyến nhận từ vertex shader
varying vec3 v;             // tọa độ đỉnh từ vertex shader

void main (void) {
    vec3 L = normalize(lightPosition.xyz - v);  // xác định vector hướng sáng
    vec3 E = normalize(-v);                     // xác định vector hướng nhìn
    vec3 R = normalize(-reflect(L,N));          // xác định vector phản chiếu
    // tính thành phần ambient
    vec4 Iamb = vec4(ambientColor, 1.0);
    // tính thành phần diffuse
    vec4 Idiff = vec4(diffuseColor * max(dot(N, L), 0.0), 1.0);
    // tính thành phần specular
    vec4 Ispec = vec4(specularColor * pow(max(dot(R, E),0.0), shininess);
    // cập nhật giá trị pixel dựa trên phương trình Phong
    gl_FragColor = Ka * Iamb + Kd * Idiff + Ks * Ispec;
}
```

- Tập tin `MAIN.C`

```
#include <GL/glew.h>
#include <GL/glut.h>
#include <stdio.h>
#include <stdlib.h>
#include "util.h"
#include "shader.h"
```

```
#ifdef _MSC_VER                             /* for MSVC */
#   pragma comment (lib, "opengl32.lib")
#   pragma comment (lib, "glu32.lib")
#   pragma comment (lib, "glew32.lib")
#   pragma comment (lib, "freeglut.lib")
#endif

#define WND_WIDTH   640                     /* chiều rộng cửa sổ */
#define WND_HEIGHT  360                     /* chiều cao cửa sổ */
#define WND_TITLE   "Lab05-3"               /* tiêu đề cửa sổ */
#define VS_FILE     "vertex.glsl"           /* tập tin mã nguồn vertex shader */
#define FS_FILE     "fragment.glsl"         /* tập tin mã nguồn fragment shader */

typedef enum objectMode { CUBE, SPHERE, TORUS, TEAPOT } OBJECT_MODE;

/* tham số cho fragment shader */
vec3 light      = {0.0f, 1.0f, 1.0f};       /* vị trí nguồn sáng  */
vec3 ambient    = {0.1f, 0.1f, 0.1f};       /* màu ambient */
vec3 diffuse    = {1.0f, 0.5f, 0.0f};       /* màu diffuse */
vec3 specular   = {1.0f, 1.0f, 1.0f};       /* màu specular */
float Ka        = 0.1f;                     /* hệ số ambient */
float Kd        = 1.0f;                     /* hệ số diffuse */
float Ks        = 1.0f;                     /* hệ số specular */
float shininess = 32.0f;                    /* hệ số bóng bề mặt */
vec3 camera     = {0.0f, 0.0f, 5.0f};       /* vị trí camera */
float xAngle    = 0.0f;                     /* góc quay vật thể */
float yAngle    = 0.0f;
float zAngle    = 0.0f;
int xMotion = 0;                            /* theo dõi di chuyển chuột */
int yMotion = 0;

GLSL_PROGRAM * prog = NULL;                 /* đối tượng chương trình shader */
OBJECT_MODE obj = TEAPOT;                   /* vật thể hiện tại đang hiển thị */

void resize(int width, int height) {  /* thay đổi kích thước cửa sổ */
    if (height == 0) height = 1;
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0f, (float) width/height, 1.0f, 100.0f);
    glViewport(0, 0, width, height);
    glMatrixMode(GL_MODELVIEW);
}

void render() { /* dựng hình */
    /* truyền các tham số vào fragment shader */
    glslSetUniform3fv(prog, "lightPosition",  light);
    glslSetUniform3fv(prog, "diffuseColor",  diffuse);
    glslSetUniform1f (prog, "shininess", shininess);
    glslSetUniform1f(prog, "Ka", Ka);
    glslSetUniform1f(prog, "Kd", Kd);
    glslSetUniform1f(prog, "Ks", Ks);

    glClearColor(0.05f, 0.1f, 0.2f, 1.0f);
```

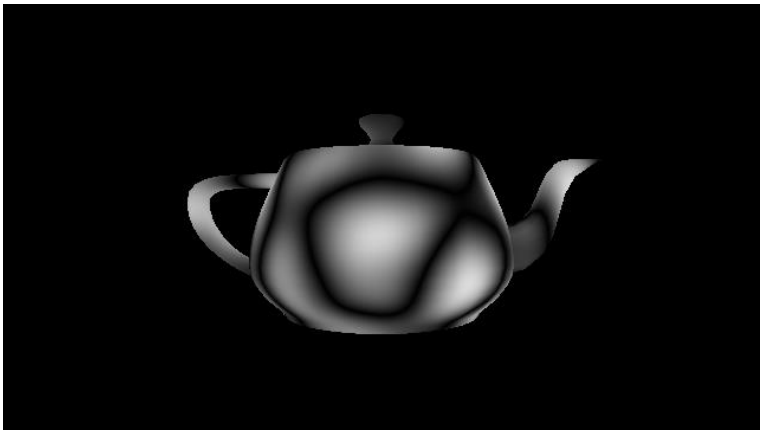
```
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    gluLookAt(camera.x, 0.0f, camera.z,
              camera.x, 0.0f, camera.z-1.0f,
              0.0f, 1.0f,  0.0f);
    /* vẽ nguồn sáng */
    glslDeactivate(prog);
    glPushMatrix();
    glLoadIdentity();
    glTranslatef(light.x, light.y, light.z);
    glColor3f(1.0, 1.0, 1.0);
    glutSolidSphere(0.05, 4, 4);
    glPopMatrix();
    glslActivate(prog);

    /* vẽ khung cảnh */
    glRotatef(xAngle, 1.0f, 0.0f, 0.0f);
    glRotatef(yAngle, 0.0f, 1.0f, 0.0f);
    glRotatef(zAngle, 0.0f, 0.0f, 1.0f);
    switch(obj) {
    case CUBE:      glutSolidCube(1.5f); break;
    case SPHERE:    glutSolidSphere(1.0f, 24, 24); break;
    case TORUS:     glutSolidTorus(0.5, 1.0f, 24, 24); break;
    case TEAPOT:    glutSolidTeapot(1.0f); break;
    }
    glutSwapBuffers();
}

void input(unsigned char key, int x, int y) { /* xử lý sự kiện bàn phím */
    switch (key) {
    case 27: exit(0);
    case 'a':
    case 'A':   camera.x -= 0.1f; light.x += 0.1f; break;
    case 'd':
    case 'D':   camera.x += 0.1f; light.x -= 0.1f; break;
    case 'w':
    case 'W':   camera.z -= 0.1f; light.z += 0.1f; break;
    case 's':
    case 'S':   camera.z += 0.1f; light.z -= 0.1f; break;
    case 'X':   light.x += 0.1f; break;
    case 'Y':   light.y += 0.1f; break;
    case 'Z':   light.z += 0.1f; break;
    case 'x':   light.x -= 0.1f; break;
    case 'y':   light.y -= 0.1f; break;
    case 'z':   light.z -= 0.1f; break;
    case 'R':   if (diffuse.r < 1.0f)   diffuse.r += 0.01f; break;
    case 'G':   if (diffuse.g < 1.0f)   diffuse.g += 0.01f; break;
    case 'B':   if (diffuse.b < 1.0f)   diffuse.b += 0.01f; break;
    case 'r':   if (diffuse.r >= 0.0f)  diffuse.r -= 0.01f; break;
    case 'g':   if (diffuse.g >= 0.0f)  diffuse.g -= 0.01f; break;
    case 'b':   if (diffuse.b >= 0.0f)  diffuse.b -= 0.01f; break;
    case 'H':   if (shininess < 200.0f) shininess += 1.0f;  break;
```

```
    case 'h':   if (shininess > 6.0f)   shininess -= 1.0f;  break;
    case '[':   if (Kd >= 0.0f) Kd -= 0.05f; break;
    case ']':   if (Kd <  1.0f) Kd += 0.05f; break;
    case '{':   if (Ks >= 0.0f) Ks -= 0.05f; break;
    case '}':   if (Ks <  1.0f) Ks += 0.05f; break;
    case '<':   if (Ka >= 0.0f) Ka -= 0.05f; break;
    case '>':   if (Ka <  1.0f) Ka += 0.05f; break;
    case '1':   obj = CUBE;   break;
    case '2':   obj = SPHERE; break;
    case '3':   obj = TORUS;  break;
    case '4':   obj = TEAPOT; break;
    default:    break;
    }
}

void mouse(int button, int state, int x, int y) { /* xử lý sự kiện chuột */
    if (state == 0 && button == 0) { /* nút trái được nhấn */
        xMotion = x;
        yMotion = y;
    }
}

void motion (int x, int y) { /* theo dõi chuyển động chuột */
    if (xMotion) {
        if (xMotion > x) yAngle -= 2.0f;
        if (xMotion < x) yAngle += 2.0f;
        xMotion = x;
    }
    if (yMotion) {
        if (yMotion > y) xAngle -= 1.0f;
        if (yMotion < y) xAngle += 1.0f;
        yMotion = y;
    }
}

void idle() { /* xử lý khi không có sự kiện gì xảy ra */
    glutPostRedisplay();
}

GLboolean init(int argc, char ** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowPosition(-1, -1);
    glutInitWindowSize(WND_WIDTH, WND_HEIGHT);
    glutCreateWindow(WND_TITLE);
    glutDisplayFunc(render);
    glutIdleFunc(idle);
    glutReshapeFunc(resize);
    glutKeyboardFunc(input);
    glutMotionFunc(motion);
    glutMouseFunc(mouse);

    glEnable(GL_DEPTH_TEST);
```

```
    if (GLEW_OK != glewInit()) return GL_FALSE;
    if (!(prog = glslCreate())) printf("No GLSL supported.\n");
    glslCompileFile(prog, VERTEX_SHADER, VS_FILE));
    glslCompileFile(prog, FRAGMENT_SHADER, FS_FILE));
    glslLink(prog);
    glslActivate(prog);
    glslSetUniform3fv(prog, "ambientColor",  ambient);
    glslSetUniform3fv(prog, "specularColor", specular);

    return GL_TRUE;
}

void done() {
    glslDestroy(prog);
}

void run() {
    glutMainLoop();
}

int main(int argc, char ** argv) {
    init(argc, argv);
    run();
    return 0;
}
```

Bài 4: Shader Nhiễu Simplex

Mở rộng mã nguồn của Bài 1: Shader ánh xạ vị trí RGB, xây dựng *vertex shader* và *fragment shader* để biểu diễn nhiễu Simplex [4] trên bề mặt vật thể 3D. Cho phép sử dụng các phím *R, r, G, g, B, b* để thay giá trị màu của vân nhiễu trên bề mặt vật thể.



Hình 7. Kết xuất của Bài 4: Shader Nhiễu Simplex

- Tập tin `VERTEX.GLSL`

```
// SIMPLEX NOISE SHADER: VERTEX SHADER
varying vec3 position; // tọa độ đỉnh dùng cho fragment shader

void main() {
    position = gl_Vertex.xyz;
```

```
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

- Tập tin FRAGMENT.GLSL

```
// SIMPLEX NOISE SHADER: FRAGMENT SHADER
uniform float time;     // giá trị thời gian hiện hành để biến đổi nhiễu
uniform vec3 shade;     // màu của nhiễu
varying vec3 position;  // tọa độ đỉnh từ vertex shader
const float speed = 1.0;

// BỘ PHÁT SINH NHIỄU SIMPLEX
vec4 permute(vec4 x) {
    return mod(((x*34.0)+1.0)*x, 289.0);
}

vec4 taylorInvSqrt(vec4 r)  {
    return 1.79284291400159-0.85373472095314 * r;
}

float snoise(vec3 v) {
    const vec2  C = vec2(1.0/6.0, 1.0/3.0);
    const vec4  D = vec4(0.0, 0.5, 1.0, 2.0);

    vec3 i  = floor(v + dot(v, C.yyy) );
    vec3 x0 =   v - i + dot(i, C.xxx) ;
    vec3 g = step(x0.yzx, x0.xyz);
    vec3 l = 1.0 - g;
    vec3 i1 = min( g.xyz, l.zxy );
    vec3 i2 = max( g.xyz, l.zxy );
    vec3 x1 = x0 - i1 + 1.0 * C.xxx;
    vec3 x2 = x0 - i2 + 2.0 * C.xxx;
    vec3 x3 = x0 - 1. + 3.0 * C.xxx;

    i = mod(i, 289.0 );
    vec4 p = permute(permute( permute(
                 i.z + vec4(0.0, i1.z, i2.z, 1.0 ))
               + i.y + vec4(0.0, i1.y, i2.y, 1.0 ))
               + i.x + vec4(0.0, i1.x, i2.x, 1.0 ));

    float nn = 1.0/7.0;
    vec3 ns = nn * D.wyz - D.xzx;
    vec4 j = p - 49.0 * floor(p * ns.z *ns.z);
    vec4 xx = floor(j * ns.z);
    vec4 yy = floor(j - 7.0 * xx );
    vec4 x = xx *ns.x + ns.yyyy;
    vec4 y = yy *ns.x + ns.yyyy;
    vec4 h = 1.0 - abs(x) - abs(y);
    vec4 b0 = vec4( x.xy, y.xy );
    vec4 b1 = vec4( x.zw, y.zw );
    vec4 s0 = floor(b0)*2.0 + 1.0;
    vec4 s1 = floor(b1)*2.0 + 1.0;
    vec4 sh = -step(h, vec4(0.0));
    vec4 a0 = b0.xzyw + s0.xzyw*sh.xxyy ;
```

```
    vec4 a1 = b1.xzyw + s1.xzyw*sh.zzww ;
    vec3 p0 = vec3(a0.xy,h.x);
    vec3 p1 = vec3(a0.zw,h.y);
    vec3 p2 = vec3(a1.xy,h.z);
    vec3 p3 = vec3(a1.zw,h.w);

    vec4 norm = taylorInvSqrt(vec4( dot(p0,p0),  dot(p1,p1),
                                    dot(p2, p2), dot(p3,p3)));
    p0 *= norm.x; p1 *= norm.y;
    p2 *= norm.z; p3 *= norm.w;

    vec4 m = max(0.6 - vec4(dot(x0,x0), dot(x1,x1),
                            dot(x2,x2), dot(x3,x3)), 0.0);
    m = m*m*m*m;
    return 42.0 * dot(m, vec4(  dot(p0,x0), dot(p1,x1),
                                dot(p2,x2), dot(p3,x3) ) );
}

void main() {
    vec3 v = position + time * speed;
    gl_FragColor = vec4(shade, 1.0) * vec4(abs(vec3(snoise(v))), 1.0);
}
```

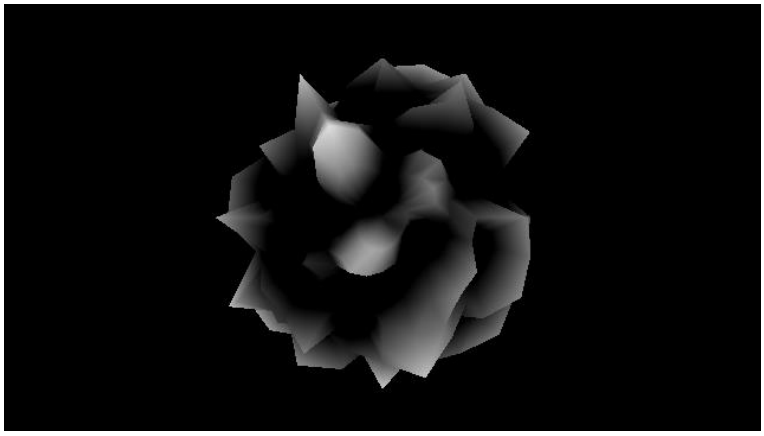
- Tập tin `MAIN.C`

```
/* các phần mã tương tự Bài 1 sẽ không được trình bày ở đây */

/* các tập tin header, ta bổ sung time.h */
/* . . . */
#include <time.h>

/* các biến toàn cục */
vec3  shade = {1.0, 1.0, 1.0};  /* giá trị màu của vân nhiễu */

/* . . . */

void render() {  /* dựng hình khung cảnh */
    /* truyền các giá trị cần thiết cho fragment shader */
    glslSetUniform1f(prog, "time", clock()/1000.0f);
    glslSetUniform3fv(prog, "shade", shade);

    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt(0.0f, 0.0f, 4.0f,  0.0f, 0.0f, 0.0f,  0.0f, 1.0f, 0.0f);
    /* vẽ khung cảnh */
    glRotatef(xAngle, 1.0f, 0.0f, 0.0f);
    glRotatef(yAngle, 0.0f, 1.0f, 0.0f);
    glRotatef(zAngle, 0.0f, 0.0f, 1.0f);
    switch(obj) {
    case CUBE:      glutSolidCube(1.5f); break;
    case SPHERE:    glutSolidSphere(1.0f, 24, 24); break;
    case TORUS:     glutSolidTorus(0.5, 1.0f, 24, 24); break;
    case TEAPOT:    glutSolidTeapot(1.0f); break;
```

```
    }
    glutSwapBuffers();
}

void input(unsigned char key, int x, int y) /* xử lý sự kiện bàn phím */ {
    switch (key) {
    case 27: exit(0);
    case '1':   obj = CUBE;   break;
    case '2':   obj = SPHERE; break;
    case '3':   obj = TORUS;  break;
    case '4':   obj = TEAPOT; break;
    case 'R':   if (shade.r < 1.0 ) shade.r += 0.05; break;
    case 'r':   if (shade.r >= 0.0) shade.r -= 0.05; break;
    case 'G':   if (shade.g < 1.0 ) shade.g += 0.05; break;
    case 'g':   if (shade.g >= 0.0) shade.g -= 0.05; break;
    case 'B':   if (shade.b < 1.0 ) shade.b += 0.05; break;
    case 'b':   if (shade.b >= 0.0) shade.b -= 0.05; break;
    default:    break;
    }
}

/* . . . */
```

Bài 5: Shader Displacement

Mở rộng mã nguồn của Bài 1: Shader ánh xạ vị trí RGB, xây dựng *vertex shader* và *fragment shader* áp dụng thuật toán nhiễu Simplex [4] để biến đổi vị trí các đỉnh của vật thể. Cho phép sử dụng các phím *R, r, G, g, B, b* để thay giá trị màu của vân nhiễu trên bề mặt vật thể, phím [,] để thay đổi độ mạnh của hiệu ứng displacement.



Hình 8. Kết xuất từ Bài 5: Shader Displacement

- Tập tin VERTEX.GLSL

```
// DISPLACEMENT SHADER: VERTEX SHADER
uniform float time;         // trị thời gian biến đổi nhiễu
uniform float displacement; // độ mạnh của hiệu ứng
varying float noise;        // giá trị màu cho fragment shader
const float speed = 1.0;    // tốc độ hoạt cảnh
const float scale = 2.0;    // tỷ lệ vân nhiễu
```

```
// SIMPLEX NOISE GENERATOR
vec4 permute(vec4 x) {
    return mod(((x*34.0)+1.0)*x, 289.0);
}

vec4 taylorInvSqrt(vec4 r)  {
    return 1.79284291400159-0.85373472095314 * r;
}

float snoise(vec3 v) {
    const vec2  C = vec2(1.0/6.0, 1.0/3.0);
    const vec4  D = vec4(0.0, 0.5, 1.0, 2.0);

    vec3 i  = floor(v + dot(v, C.yyy) );
    vec3 x0 =   v - i + dot(i, C.xxx) ;
    vec3 g = step(x0.yzx, x0.xyz);
    vec3 l = 1.0 - g;
    vec3 i1 = min( g.xyz, l.zxy );
    vec3 i2 = max( g.xyz, l.zxy );
    vec3 x1 = x0 - i1 + 1.0 * C.xxx;
    vec3 x2 = x0 - i2 + 2.0 * C.xxx;
    vec3 x3 = x0 - 1. + 3.0 * C.xxx;

    i = mod(i, 289.0 );
    vec4 p = permute(permute( permute(
                 i.z + vec4(0.0, i1.z, i2.z, 1.0 ))
               + i.y + vec4(0.0, i1.y, i2.y, 1.0 ))
               + i.x + vec4(0.0, i1.x, i2.x, 1.0 ));

    float nn = 1.0/7.0;
    vec3 ns = nn * D.wyz - D.xzx;
    vec4 j = p - 49.0 * floor(p * ns.z *ns.z);
    vec4 xx = floor(j * ns.z);
    vec4 yy = floor(j - 7.0 * xx );
    vec4 x = xx *ns.x + ns.yyyy;
    vec4 y = yy *ns.x + ns.yyyy;
    vec4 h = 1.0 - abs(x) - abs(y);
    vec4 b0 = vec4( x.xy, y.xy );
    vec4 b1 = vec4( x.zw, y.zw );
    vec4 s0 = floor(b0)*2.0 + 1.0;
    vec4 s1 = floor(b1)*2.0 + 1.0;
    vec4 sh = -step(h, vec4(0.0));
    vec4 a0 = b0.xzyw + s0.xzyw*sh.xxyy ;
    vec4 a1 = b1.xzyw + s1.xzyw*sh.zzww;
    vec3 p0 = vec3(a0.xy,h.x);
    vec3 p1 = vec3(a0.zw,h.y);
    vec3 p2 = vec3(a1.xy,h.z);
    vec3 p3 = vec3(a1.zw,h.w);

    vec4 norm = taylorInvSqrt(vec4( dot(p0,p0),  dot(p1,p1),
                                    dot(p2, p2), dot(p3,p3)));
    p0 *= norm.x; p1 *= norm.y;
    p2 *= norm.z; p3 *= norm.w;
```

```
    vec4 m = max(0.6 - vec4(dot(x0,x0), dot(x1,x1),
                            dot(x2,x2), dot(x3,x3)), 0.0);
    m = m*m*m*m;
    return 42.0 * dot(m, vec4(  dot(p0,x0), dot(p1,x1),
                                dot(p2,x2), dot(p3,x3) ) );
}

void main() {
    noise = snoise(normalize(gl_Vertex.xyz)*scale+(time*speed));
    vec3 pos = gl_Vertex.xyz+gl_Normal*noise*vec3(clamp(displacement, 0.0, 1.0));
    gl_Position = gl_ModelViewProjectionMatrix*vec4(pos, 1.0);
}
```

- Tập tin FRAGMENT.GLSL

```
// DISPLACEMENT SHADER: FRAGMENT SHADER
uniform vec3 shade;     // màu của vân nhiễu, từ chương trình chính
varying float noise;    // giá trị màu của pixel nhiễu, từ vertex shader

void main() {
    gl_FragColor = vec4(shade * clamp(noise, 0.0, 1.0 ), 1.0 );
}
```

- Tập tin MAIN.C

```
/* các phần mã tương tự Bài 1 sẽ không được trình bày ở đây */

/* các tập tin header, ta bổ sung time.h */
/* . . . */
#include <time.h>

/* các biến toàn cục */
float displacement = 0.5f;                  /* độ mạnh của hiệu ứng */
vec3  shade     = {1.0, 1.0, 1.0};          /* giá trị màu của vân nhiễu */

/* . . . */

void render() {  /* dựng hình khung cảnh */
    /* truyền các giá trị cần thiết cho fragment shader */
    glslSetUniform1f(prog, "time", clock()/1000.0f);
    glslSetUniform1f(prog, "displacement", displacement);
    glslSetUniform3fv(prog, "shade", shade);
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt(0.0f, 0.0f, 4.0f,  0.0f, 0.0f, 0.0f,  0.0f, 1.0f, 0.0f);
    /* draw scene */
    glRotatef(xAngle, 1.0f, 0.0f, 0.0f);
    glRotatef(yAngle, 0.0f, 1.0f, 0.0f);
    glRotatef(zAngle, 0.0f, 0.0f, 1.0f);
    switch(obj) {
    case CUBE:      glutSolidCube(1.5f); break;
    case SPHERE:    glutSolidSphere(1.0f, 24, 24); break;
    case TORUS:     glutSolidTorus(0.5, 1.0f, 24, 24); break;
```

```
    case TEAPOT:    glutSolidTeapot(1.0f); break;
    }
    glutSwapBuffers();
}

void input(unsigned char key, int x, int y) /* xử lý sự kiện bàn phím */ {
    switch (key) {
    case 27: exit(0);
    case '1':   obj = CUBE;   break;
    case '2':   obj = SPHERE; break;
    case '3':   obj = TORUS;  break;
    case '4':   obj = TEAPOT; break;
    case 'R':   if (shade.r < 1.0 ) shade.r += 0.05; break;
    case 'r':   if (shade.r >= 0.0) shade.r -= 0.05; break;
    case 'G':   if (shade.g < 1.0 ) shade.g += 0.05; break;
    case 'g':   if (shade.g >= 0.0) shade.g -= 0.05; break;
    case 'B':   if (shade.b < 1.0 ) shade.b += 0.05; break;
    case 'b':   if (shade.b >= 0.0) shade.b -= 0.05; break;
    case '[':   displacement -= 0.01f; break;
    case ']':   displacement += 0.01f; break;
    default:    break;
    }
}

/* . . . */
```

Mã nguồn

Để tải về mã nguồn của các bài tập này, vui lòng truy cập vào địa chỉ: https://github.com/dzutrinh/CMP211-LAB.

Tài liệu tham khảo

[1] J. Kessenich, The OpenGL Shading Language, The Khronos Group Inc., 2017.

[2] B. T. Phong, “Illumination for computer generated pictures,” The University of Utah, 1973.

[3] H. Gouraud, “Computer display of curved surfaces,” The University of Utah, 1971.

[4] K. Perlin, "Noise Hardware," in *Real-Time Shading SIGGRAPH Course Notes*, 2001.