```
/*


  Simple Demo for GLSL

  www.lighthouse3d.com

*/

#include <stdio.h>
#include <stdlib.h>
#include <GL/glew.h>
#include <GL/glut.h>

#include "textfile.h"


GLhandleARB v,f,f2,p;
float lpos[4] = {1,0.5,1,0};

void changeSize(int w, int h) {

        // Prevent a divide by zero, when window is too short
        // (you cant make a window of zero width).
        if(h == 0)
                h = 1;

        float ratio = 1.0* w / h;

        // Reset the coordinate system before modifying
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();

        // Set the viewport to be the entire window
    glViewport(0, 0, w, h);

        // Set the correct perspective.
        gluPerspective(45,ratio,1,1000);
        glMatrixMode(GL_MODELVIEW);


}
float a = 0;

void renderScene(void) {

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glLoadIdentity();
        gluLookAt(0.0,0.0,5.0,
                  0.0,0.0,-1.0,
                      0.0f,1.0f,0.0f);

        glLightfv(GL_LIGHT0, GL_POSITION, lpos);
        glRotatef(a,0,1,1);
        glutSolidTeapot(1);
```



```
        a+=0.1;

        glutSwapBuffers();
}

void processNormalKeys(unsigned char key, int x, int y) {

        if (key == 27)
                exit(0);
}

#define printOpenGLError() printOglError(__FILE__, __LINE__)

int printOglError(char *file, int line)
{
    //
    // Returns 1 if an OpenGL error occurred, 0 otherwise.
    //
    GLenum glErr;
    int    retCode = 0;

    glErr = glGetError();
    while (glErr != GL_NO_ERROR)
    {
        printf("glError in file %s @ line %d: %s\n", file, line,
gluErrorString(glErr));
        retCode = 1;
        glErr = glGetError();
    }
    return retCode;
}


void printInfoLog(GLhandleARB obj)
{
    int infologLength = 0;
    int charsWritten  = 0;
    char *infoLog;

        glGetObjectParameterivARB(obj,
GL_OBJECT_INFO_LOG_LENGTH_ARB,
                                                &infologLength);

    if (infologLength > 0)
    {
        infoLog = (char *)malloc(infologLength);
        glGetInfoLogARB(obj, infologLength, &charsWritten,
infoLog);
                printf("%s\n",infoLog);
        free(infoLog);
    }
}



void setShaders() {
```

```
	char *vs = NULL,*fs = NULL,*fs2 = NULL;

	v = glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
	f = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);
	f2 = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);


	vs = textFileRead("minimal.vert");
	fs = textFileRead("minimal.frag");

	const char * vv = vs;
	const char * ff = fs;

	glShaderSourceARB(v, 1, &vv,NULL);
	glShaderSourceARB(f, 1, &ff,NULL);

	free(vs);free(fs);

	glCompileShaderARB(v);
	glCompileShaderARB(f);

	printInfoLog(v);
	printInfoLog(f);
	printInfoLog(f2);

	p = glCreateProgramObjectARB();
	glAttachObjectARB(p,v);
	glAttachObjectARB(p,f);

	glLinkProgramARB(p);
	printInfoLog(p);

	glUseProgramObjectARB(p);




}



int main(int argc, char **argv) {
	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
	glutInitWindowPosition(100,100);
	glutInitWindowSize(320,320);
	glutCreateWindow("MM 2004-05");

	glutDisplayFunc(renderScene);
	glutIdleFunc(renderScene);
	glutReshapeFunc(changeSize);
	glutKeyboardFunc(processNormalKeys);

	glEnable(GL_DEPTH_TEST);
```



```
	glClearColor(1.0,1.0,1.0,1.0);
	glEnable(GL_CULL_FACE);

	glewInit();
	if (GLEW_ARB_vertex_shader && GLEW_ARB_fragment_shader)
		printf("Ready for GLSL\n");
	else {
		printf("No GLSL support\n");
		exit(1);
	}

	setShaders();

	glutMainLoop();

	return 0;
}

// minimal vertex shader
// www.lighthouse3d.com



void main()
{

	// the following three lines provide the same result

//	gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix *
gl_Vertex;
//	gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
	gl_Position = ftransform();
}

// minimal fragment shader
// www.lighthouse3d.com


void main()
{
	// gl_FragColor = vec4(0.4,0.4,0.8,1.0);
	gl_FragColor = vec4(0.9,0.,0.,1.0);
}
```

```
/*


  Simple Demo for GLSL

  www.lighthouse3d.com

*/

#include <GL/glew.h>
#include <GL/glut.h>
#include <stdio.h>
#include <stdlib.h>

#include "textfile.h"


GLint loc;
GLhandleARB v,f,f2,p;

float lpos[4] = {1.0,0.0,1.0,0.0};


void changeSize(int w, int h) {

        // Prevent a divide by zero, when window is too short
        // (you cant make a window of zero width).
        if(h == 0)
                h = 1;

        float ratio = 1.0* w / h;

        // Reset the coordinate system before modifying
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();

        // Set the viewport to be the entire window
    glViewport(0, 0, w, h);

        // Set the correct perspective.
        gluPerspective(45,ratio,1,100);
        glMatrixMode(GL_MODELVIEW);


}

float a = 0;

void renderScene(void) {

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glLoadIdentity();
        gluLookAt(0.0,5.0,5.0,
                  0.0,0.0,0.0,
                      0.0f,1.0f,0.0f);
```



```
        glLightfv(GL_LIGHT0, GL_POSITION, lpos);
        glRotatef(a,0,1,0);

        glutSolidTeapot(1);
        a+=0.01;

        glutSwapBuffers();
}

void processNormalKeys(unsigned char key, int x, int y) {

        if (key == 27)
                exit(0);
}

#define printOpenGLError() printOglError(__FILE__, __LINE__)

int printOglError(char *file, int line)
{
    //
    // Returns 1 if an OpenGL error occurred, 0 otherwise.
    //
    GLenum glErr;
    int    retCode = 0;

    glErr = glGetError();
    while (glErr != GL_NO_ERROR)
    {
        printf("glError in file %s @ line %d: %s\n", file, line,
gluErrorString(glErr));
        retCode = 1;
        glErr = glGetError();
    }
    return retCode;
}


void printInfoLog(GLhandleARB obj)
{
    int infologLength = 0;
    int charsWritten  = 0;
    char *infoLog;

        glGetObjectParameterivARB(obj,
GL_OBJECT_INFO_LOG_LENGTH_ARB,
                                                &infologLength);

    if (infologLength > 0)
    {
        infoLog = (char *)malloc(infologLength);
        glGetInfoLogARB(obj, infologLength, &charsWritten,
infoLog);
                printf("%s\n",infoLog);
        free(infoLog);
    }
```

```
}


void setShaders() {

        char *vs = NULL,*fs = NULL,*fs2 = NULL;

        v = glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
        f = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);
        f2 = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);


        vs = textFileRead("toonf2.vert");
        fs = textFileRead("toonf2.frag");

        const char * vv = vs;
        const char * ff = fs;

        glShaderSourceARB(v, 1, &vv,NULL);
        glShaderSourceARB(f, 1, &ff,NULL);

        free(vs);free(fs);

        glCompileShaderARB(v);
        glCompileShaderARB(f);

        printInfoLog(v);
        printInfoLog(f);
        printInfoLog(f2);

        p = glCreateProgramObjectARB();
        glAttachObjectARB(p,v);
        glAttachObjectARB(p,f);

        glLinkProgramARB(p);
        printInfoLog(p);

        glUseProgramObjectARB(p);

        loc = glGetUniformLocationARB(p,"time");


}



int main(int argc, char **argv) {
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
        glutInitWindowPosition(100,100);
        glutInitWindowSize(320,320);
        glutCreateWindow("Lighthouse 3D");

        glutDisplayFunc(renderScene);
```



```
        glutIdleFunc(renderScene);
        glutReshapeFunc(changeSize);
        glutKeyboardFunc(processNormalKeys);

        glEnable(GL_DEPTH_TEST);
        glClearColor(1.0,1.0,1.0,1.0);
//      glEnable(GL_CULL_FACE);

        glewInit();
        if (GLEW_ARB_vertex_shader && GLEW_ARB_fragment_shader)
                printf("Ready for GLSL\n");
        else {
                printf("No GLSL support\n");
                exit(1);
        }

        setShaders();

        glutMainLoop();

        return 0;
}


// Vertex shader


varying vec3 lightDir,normal;

void main()
{
        lightDir = normalize(vec3(gl_LightSource[0].position));
        normal = gl_NormalMatrix * gl_Normal;
        // gl_NormalMatrix permet une transformation un objet dans
le repère de la caméra.
        // Pourquoi appliquer ici cette trransformation, c'est
parce que la position de la source
        // est automatiquement transformée dans le repère de la
caméra.

        gl_Position = ftransform();
}

// Frag shader

varying vec3 lightDir,normal;

void main()
{

        float intensity;
        vec4 color;

        // normalizing the lights position to be on the safe side
```

```
	vec3 n = normalize(normal);

	intensity = dot(lightDir,n);

	if (intensity > 0.95)
		color = vec4(1.0,0.5,0.5,1.0);
	else if (intensity > 0.5)
		color = vec4(0.6,0.3,0.3,1.0);
	else if (intensity > 0.25)
		color = vec4(0.4,0.2,0.2,1.0);
	else
		color = vec4(0.2,0.1,0.1,1.0);

	gl_FragColor = color;
}


// textfile.h: interface for reading and writing text files
// www.lighthouse3d.com
//
// You may use these functions freely.
// they are provided as is, and no warranties, either implicit,
// or explicit are given
//////////////////////////////////////////////////////////////////
////

char *textFileRead(char *fn);
int textFileWrite(char *fn, char *s);


// texfile.cpp

// textfile.cpp
//
// simple reading and writing for text files
//
// www.lighthouse3d.com
//
// You may use these functions freely.
// they are provided as is, and no warranties, either implicit,
// or explicit are given
//////////////////////////////////////////////////////////////////
////


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


char *textFileRead(char *fn) {


	FILE *fp;
	char *content = NULL;
```



```
	int count=0;

	if (fn != NULL) {
		fp = fopen(fn,"rt");

		if (fp != NULL) {

      fseek(fp, 0, SEEK_END);
      count = ftell(fp);
      rewind(fp);

			if (count > 0) {
				content = (char
*)malloc(sizeof(char) * (count+1));
				count =
fread(content,sizeof(char),count,fp);
				content[count] = '\0';
			}
			fclose(fp);
		}
	}
	return content;
}

int textFileWrite(char *fn, char *s) {

	FILE *fp;
	int status = 0;

	if (fn != NULL) {
		fp = fopen(fn,"w");

		if (fp != NULL) {

			if (fwrite(s,sizeof(char),strlen(s),fp) ==
strlen(s))
				status = 1;
			fclose(fp);
		}
	}
	return(status);
}
```

```
// Vertex shader

/* ------------------------------------------------------

This shader implements a directional light per vertex using the
diffuse, specular, and ambient terms acoording to "Mathematics of Lighthing"
as found in the book "OpenGL Programming Guide" (aka the Red Book)

António Ramires Fernandes

------------------------------------------------------ */

void main()
{
	vec3 normal, lightDir, viewVector, halfVector;
	vec4 diffuse, ambient, globalAmbient, specular = vec4(0.0);
	float NdotL,NdotHV;

	/* first transform the normal into eye space and normalize the result */
	normal = normalize(gl_NormalMatrix * gl_Normal);

	/* now normalize the light's direction. Note that according to the
	OpenGL specification, the light is stored in eye space. Also since
	we're talking about a directional light, the position field is actually
	direction */
	lightDir = normalize(vec3(gl_LightSource[0].position));

	/* compute the cos of the angle between the normal and lights direction.
	The light is directional so the direction is constant for every vertex.
	Since these two are normalized the cosine is the dot product. We also
	need to clamp the result to the [0,1] range. */

	NdotL = max(dot(normal, lightDir), 0.0);

	/* Compute the diffuse, ambient and globalAmbient terms */
	diffuse = gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse;
	ambient = gl_FrontMaterial.ambient * gl_LightSource[0].ambient;
	globalAmbient = gl_LightModel.ambient * gl_FrontMaterial.ambient;

	/* compute the specular term if NdotL is  larger than zero */
	if (NdotL > 0.0) {

		NdotHV = max(dot(normal, normalize(gl_LightSource[0].halfVector.xyz)),0.0);
		specular = gl_FrontMaterial.specular * gl_LightSource[0].specular *
pow(NdotHV,gl_FrontMaterial.shininess);
	}

	gl_FrontColor = globalAmbient + NdotL * diffuse + ambient + specular;

	gl_Position = ftransform();
```



```
}

// Frag shader

void main()
{
	gl_FragColor = gl_Color;
}
```

file:///C|/Akadi/Partition_E/COURS/OpenGL/Exemples_GL_SL/liste_exples_pdf/dirLightAmbDiffSpec.txt[03/03/2010 16:25:42]

```
// Vertex shader

/* ------------------------------------------------------

This shader implements a point light per pixel using  the
diffuse, specular, and ambient terms acoording to "Mathematics of
Lighthing"
as found in the book "OpenGL Programming Guide" (aka the Red Book)

AntÃ³nio Ramires Fernandes

------------------------------------------------------ */

varying vec4 diffuse,ambientGlobal,ambient;
varying vec3 normal,lightDir,halfVector;
varying float dist;

void main()
{
        vec4 ecPos;
        vec3 aux;

        /* first transform the normal into eye space and normalize
the result */
        normal = normalize(gl_NormalMatrix * gl_Normal);

        /* now normalize the light's direction. Note that
according to the
        OpenGL specification, the light is stored in eye space.
Also since
        we're talking about a directional light, the position
field is actually
        direction */
        ecPos = gl_ModelViewMatrix * gl_Vertex;
        aux = vec3(gl_LightSource[0].position-ecPos);
        lightDir = normalize(aux);

        /* compute the distance to the light source to a varying
variable*/
        dist = length(aux);

        /* Normalize the halfVector to pass it to the fragment
shader */
        halfVector = normalize(gl_LightSource[0].halfVector.xyz);

        /* Compute the diffuse, ambient and globalAmbient terms */
        diffuse = gl_FrontMaterial.diffuse *
gl_LightSource[0].diffuse;
        ambient = gl_FrontMaterial.ambient *
gl_LightSource[0].ambient;
        ambientGlobal = gl_LightModel.ambient *
gl_FrontMaterial.ambient;


        gl_Position = ftransform();
```



```
}

// Frag shader

/* ------------------------------------------------------

This shader implements a point light per pixel using  the
diffuse, specular, and ambient terms acoording to "Mathematics of
Lighthing"
as found in the book "OpenGL Programming Guide" (aka the Red Book)

AntÃ³nio Ramires Fernandes

------------------------------------------------------ */

varying vec4 diffuse,ambientGlobal, ambient;
varying vec3 normal,lightDir,halfVector;
varying float dist;


void main()
{
        vec3 n,halfV,viewV,ldir;
        float NdotL,NdotHV;
        vec4 color = ambientGlobal;
        float att;

        /* a fragment shader can't write a verying variable, hence
we need
        a new variable to store the normalized interpolated normal
*/
        n = normalize(normal);

        /* compute the dot product between normal and ldir */
        NdotL = max(dot(n,normalize(lightDir)),0.0);

        if (NdotL > 0.0) {

                att = 1.0 / (gl_LightSource[0].constantAttenuation
+

gl_LightSource[0].linearAttenuation * dist +

gl_LightSource[0].quadraticAttenuation * dist * dist);
                color += att * (diffuse * NdotL + ambient);


                halfV = normalize(halfVector);
                NdotHV = max(dot(n,halfV),0.0);
                color += att * gl_FrontMaterial.specular *
gl_LightSource[0].specular *
pow(NdotHV,gl_FrontMaterial.shininess);
```

```
        }

        gl_FragColor = color;
}
```



```
// Vertex shader

/* ------------------------------------------------------------

This shader implements a spotlight per pixel using  the
diffuse, specular, and ambient terms acoording to "Mathematics of
Lighthing"
as found in the book "OpenGL Programming Guide" (aka the Red Book)

Antonio Ramires Fernandes

------------------------------------------------------------ */

varying vec4 diffuse,ambientGlobal, ambient;
varying vec3 normal,lightDir,halfVector;
varying float dist;

void main()
{
        vec4 ecPos;
        vec3 aux;

        /* first transform the normal into eye space and normalize
the result */
        normal = normalize(gl_NormalMatrix * gl_Normal);

        /* now normalize the light's direction. Note that
according to the
        OpenGL specification, the light is stored in eye space.*/
        ecPos = gl_ModelViewMatrix * gl_Vertex;
        aux = vec3(gl_LightSource[0].position-ecPos);
        lightDir = normalize(aux);

        /* compute the distance to the light source to a varying
variable*/
        dist = length(aux);

        /* Normalize the halfVector to pass it to the fragment
shader */
        halfVector = normalize(gl_LightSource[0].halfVector.xyz);

        /* Compute the diffuse, ambient and globalAmbient terms */
        diffuse = gl_FrontMaterial.diffuse *
gl_LightSource[0].diffuse;
        ambient = gl_FrontMaterial.ambient *
gl_LightSource[0].ambient;
        ambientGlobal = gl_LightModel.ambient *
gl_FrontMaterial.ambient;


        gl_Position = ftransform();
```

```
}

// Frag shader

/* ------------------------------------------------------

This shader implements a spotlight per pixel using  the
diffuse, specular, and ambient terms acoording to "Mathematics of
Lighthing"
as found in the book "OpenGL Programming Guide" (aka the Red Book)

Antonio Ramires Fernandes

------------------------------------------------------- */

varying vec4 diffuse,ambientGlobal, ambient;
varying vec3 normal,lightDir,halfVector;
varying float dist;


void main()
{
        vec3 n,halfV;
        float NdotL,NdotHV;
        vec4 color = ambientGlobal;
        float att,spotEffect;

        /* a fragment shader can't write a verying variable, hence
we need
        a new variable to store the normalized interpolated normal
*/
        n = normalize(normal);

        /* compute the dot product between normal and ldir */
        NdotL = max(dot(n,normalize(lightDir)),0.0);

        if (NdotL > 0.0) {

                spotEffect =
dot(normalize(gl_LightSource[0].spotDirection),
normalize(-lightDir));
                if (spotEffect > gl_LightSource[0].spotCosCutoff)
{
                        spotEffect = pow(spotEffect,
gl_LightSource[0].spotExponent);
                        att = spotEffect /
(gl_LightSource[0].constantAttenuation +

gl_LightSource[0].linearAttenuation * dist +

gl_LightSource[0].quadraticAttenuation * dist * dist);

                        color += att * (diffuse * NdotL +
ambient);
```



```
                        halfV = normalize(halfVector);
                        NdotHV = max(dot(n,halfV),0.0);
                        color += att * gl_FrontMaterial.specular *
gl_LightSource[0].specular *
pow(NdotHV,gl_FrontMaterial.shininess);
                }
        }

        gl_FragColor = color;
}
```

```
// Vertex shader

void main()
{

        gl_TexCoord[0] = gl_MultiTexCoord0;
        gl_Position = ftransform();
}

// Frag shader

uniform sampler2D tex;

void main()
{
        vec4 color = texture2D(tex,gl_TexCoord[0].st);
        gl_FragColor = color;
}
```



```
varying vec3 lightDir,normal;

void main()
{

        normal = normalize(gl_NormalMatrix * gl_Normal);
        lightDir = normalize(vec3(gl_LightSource[0].position));

        gl_TexCoord[0] = gl_MultiTexCoord0;
        gl_Position = ftransform();
}


// Frag shader


uniform sampler2D tex;

void main()
{
        vec4 color = texture2D(tex,gl_TexCoord[0].st);
        gl_FragColor = color;
}
```

```
// Vertex shader

varying vec3 lightDir,normal;

void main()
{

        normal = normalize(gl_NormalMatrix * gl_Normal);
        lightDir = normalize(vec3(gl_LightSource[0].position));

        gl_TexCoord[0] = gl_MultiTexCoord0;
        gl_Position = ftransform();
}

// Frag shader

varying vec3 lightDir,normal;
uniform sampler2D tex,l3d;

void main()
{
        vec3 ct,cf,c;
        vec4 texel;
        float intensity,at,af,a;

        intensity = max(dot(lightDir,normalize(normal)),0.0);

        cf = intensity * (gl_FrontMaterial.diffuse).rgb +

gl_FrontMaterial.ambient.rgb;
        af = gl_FrontMaterial.diffuse.a;

        texel = texture2D(tex,gl_TexCoord[0].st);

        ct = texel.rgb;
        at = texel.a;

        c = cf * ct;
        a = af * at;

        float coef = smoothstep(1.0,0.2,intensity);
        c += coef *  vec3(texture2D(l3d,gl_TexCoord[0].st));

        gl_FragColor = vec4(c, a);
}
```