PRACTICA 3

Introducción a la programación en OpenGL

Parte obligatoria

Adrián David Morillas Marco y Guillermo Meléndez Morales
Diseño y Desarrollo de Videojuegos + Ingeniería de Computadoras

Control de la intensidad y la Iluminación a través del teclado

Para poder definir una luz se han de definir una posición donde se ubicará la luz, y a continuación la intensidad que emitirá la luz que se vaya a crear. Estas dos cualidades son definidas en el shader de fragmentos como se muestra a continuación:

```
vec3 Id = vec3 (1.0);                // vector intensidad
vec3 lpos = vec3 (0.0);              // vector posición
```

Para poder manipular los valores de la iluminación y de la intensidad, será necesario llevarlos a la clase main del proyecto. Para ello lo primero que se hará es declarar un vector posición y un vector iluminación, a través de los cuales manipularemos la iluminación y la intensidad de la luz:

```
glm::vec3 posicion = glm::vec3 (0.0f, 0.0f, 8.0f);      // vector posición
glm::vec3 iluminacion = glm::vec3 (1.0f);               // vector intensidad
```

A continuación se declararán las variables globales que permitirán acceder a las variables uniform, al igual que se realizó con las matrices ModelViewMat, ModelViewProjMat y NormalMat:

```
int uintensidad;          // uniform intensidad
int upl;                  // uniform posición
```

Una vez creadas las variables, es necesario crear los identificadores de dichas variables:

```
uintensidad = glGetUniformLocation(program, "Id");          // identificador de intensidad
upl = glGetUniformLocation(program, "lpos");                // identificador de posición
```

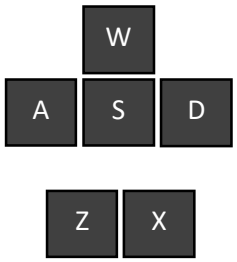
Actualmente las variables uniform intensidad e iluminación carecen de valores, por lo que se les ha de definir uno. En este caso los valores que se desea que contengan, son los vectores que se declararon al inicio “posicion” e “iluminación”, los cuales serán los operados al pulsar una de las teclas de control, para ello nos dirigiremos a la función de renderizado:

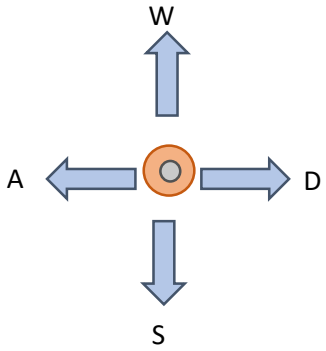
```
glm::vec3 lpos = posicion;
glm::vec3 Id = iluminacion;

if (upl != -1)
  glUniform3f(upl, lpos.x, lpos.y, lpos.z);
if (uintensidad != -1;
glUniform3f(uintensidad2, Id.x, Id.y, Id.z);
```

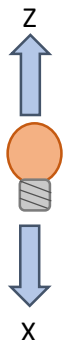
A continuación en el shader de fragmentos colocamos como entrada uniform los vectores lpos e Id y los cambiamos por los actuales:

```
uniform vec3 lpos;
uniform vec3 Id;
```

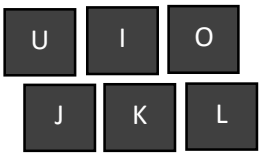
Finalmente en el método keyboardFunc de la clase main establecemos los controles de la manipulación de la intensidad y de la iluminación:

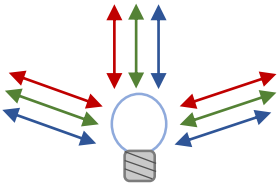






$W \rightarrow mover\ hacie\ el\ frente$	$A \rightarrow mover\ hacia\ la\ izquierda$	$X \rightarrow mover\ hacia\ abajo$
$S \rightarrow mover\ hacia\ atrás$	$D \rightarrow mover\ hacia\ la\ derecha$	$Z \rightarrow mover\ hacia\ arriba$





$U \rightarrow aumentar\ intensidad\ roja$	$I \rightarrow aumentar\ intensidad\ verde$	$O \rightarrow aumentar\ intensidad\ azul$
$J \rightarrow disminuir\ intensidad\ roja$	$K \rightarrow disminuir\ intensidad\ verde$	$L \rightarrow disminuir\ intensidad\ azul$

Esto en código se traduce a:

```
// posición
if (key == 'w'){
        posicion.z -= 0.1;
}
if (key == 's'){
        posicion.z += 0.1;
}
if (key == 'a'){
        posicion.x -= 0.1;
}
if (key == 'd'){
        posicion.x += 0.1;
}
if (key == 'z'){
        posicion.y += 0.1;
}
if (key == 'x'){
        posicion.y -= 0.1;
}
```

```
// intensidad
if (key == 'u'){
		if (iluminacion.r < 1){
			iluminacion.r += 0.1;
		}
}
if (key == 'j'){
		if (iluminacion.r > 0){
			iluminacion.r -= 0.1;
		}
}
if (key == 'i'){
		if (iluminacion.g < 1){
			iluminacion.g += 0.1;
		}
}
if (key == 'k'){
		if (iluminacion.g > 0){
			iluminacion.g -= 0.1;
		}
}
if (key == 'o'){
		if (iluminacion.b < 1){
			iluminacion.b += 0.1;
		}
}
if (key == 'l'){
		if (iluminacion.b > 0){
			iluminacion.b -= 0.1;
		}
}
```

Matriz de proyección que conserve el aspect ratio

El aspect ratio es el cociente resultante de dividir el ancho entre el alto de una imagen o pantalla

$$ar = \frac{ancho}{alto}$$

Para poder conservar el aspect ratio será necesario definir una matriz de proyección, al igual que se realizó en la primera práctica, es decir, una matriz con la siguiente estructura:

$$\begin{pmatrix} \frac{1}{tan30} & 0 & 0 & 0 \\ 0 & \frac{1}{tan30} * ar & 0 & 0 \\ 0 & 0 & -\frac{far + near}{far - near} & \frac{2 * far * near}{far - near} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Resultante de aplicar el siguiente razonamiento en la matriz de proyección:

$$ar = \frac{width}{height} = \frac{right - left}{top - bot} \rightarrow (tenemos\ una\ variable\ desconocida, (top - bot)) \rightarrow$$

$$(top - bot) = \frac{right - left}{ar} \rightarrow (sustituimos\ (top - bot)\ en\ a11) \rightarrow \frac{2 * near}{\frac{right - left}{ar}} \rightarrow$$

$$\frac{2 * near}{right - left} = 1/tan30$$

$$\frac{1}{tan30} * ar = a11$$

Dicha matriz en lenguaje de programación se traduce a:

```
float w = width;
float h = height;
float aspect = w / h;
glm::mat4 rproj(0.0f);
rproj[0].x = (1.0f / tan(3.14159f / 6.0f));
rproj[1].y = rproj[0].x*aspect;
rproj[2].z = -(100 + 0.1) / (100 - 0.1);
rproj[2].w = -1.0f;
rproj[3].z = -2.0f * 100 * 0.1 / (100 - 0.1);
```

Para poder adquirir la matriz proyección, el contenido de esta nueva matriz, simplemente se deberá hacer una igualación:

```
proj = rproj;
```

Añadir un nuevo cubo a la escena

Para añadir un nuevo cubo a la escena será necesario inicialmente crear una nueva matriz model:

```
glm::mat4   model2 = glm::mat4(1.0f);
```

En la función initObj se inicializa el objeto cubo configurando sus atributos. Tras ello se inicializan las matrices del objeto:

```
// 1.3 inicializamos la matriz model del cubo
model2 = glm::mat4(1.0f);
```

A continuación, como se realizó con el primer cubo, en la función de renderizado se pintará el cubo, para ello subimos las matrices requeridas por el shader, del segundo cubo:

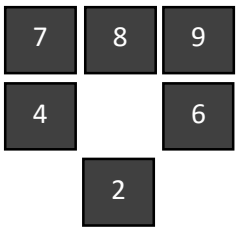
```
// 1.3 añadimos el cubo a la escena
        modelView = view * model2;
        modelViewProj = proj * view * model2;
        normal = glm::transpose(glm::inverse(modelView));

        if (uModelViewMat != -1)
                glUniformMatrix4fv(uModelViewMat, 1, GL_FALSE,
                &(modelView[0][0]));

        if (uModelViewProjMat != -1)
                glUniformMatrix4fv(uModelViewProjMat, 1, GL_FALSE,
                &(modelViewProj[0][0]));

        if (uNormalMat != -1)
                glUniformMatrix4fv(uNormalMat, 1, GL_FALSE,
                &(normal[0][0]));

        glBindVertexArray(vao);
        glDrawElements(GL_TRIANGLES, cubeNTriangleIndex * 3,
                GL_UNSIGNED_INT, (void*)0);
```

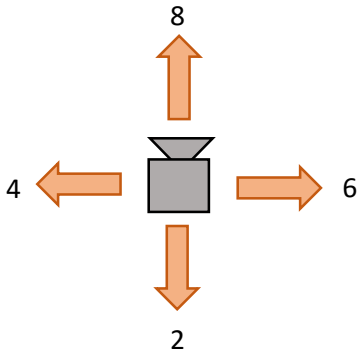
Para definir el movimiento del cubo, será necesario definirlo en el método Idle ():

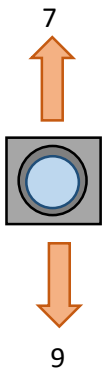
```
// 1.3 definimos el movimiento del cubo
        model2 = glm::mat4(1.0f);
        //Dar valor a la matriz de rotación
        model2 = glm::rotate(model2, angle, glm::vec3(0.0f, 1.0, 0.0f));
        //Declarar la matriz de traslación
        glm::mat4 model2tras(1.0f);
        //Dar valor a la matriz de traslacion
        model2tras = glm::translate(model2tras, glm::vec3(5.0f, 0.0f, 0.0f));
        //Declarar la matriz de escalado
        glm::mat4 escalado2(1.0f);
        //Dar valor a la matriz de escalado
        escalado2 = glm::scale(escalado2, glm::vec3(0.5f, 0.5f, 0.5f));
        // aplicar transformaciones
        model2 *= model2tras*escalado2*model2;
```

Control de la cámara con el teclado

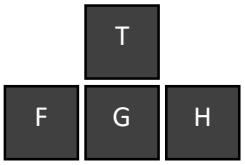
A la hora de implementar los movimientos de la cámara, se ha optado por emplea la misma estructura empleada en la primera práctica, con un leve cambio de los controles:

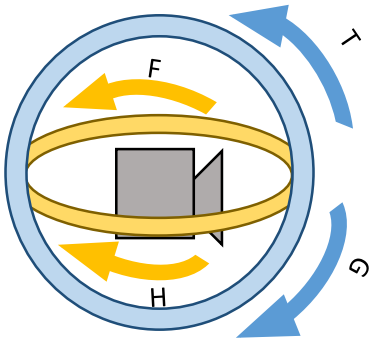






$8 \rightarrow mover\ hacie\ el\ frente$	$4 \rightarrow mover\ hacia\ la\ izquierda$	$9 \rightarrow mover\ hacia\ abajo$
$2 \rightarrow mover\ hacia\ atrás$	$6 \rightarrow mover\ hacia\ la\ derecha$	$7 \rightarrow mover\ hacia\ arriba$





$T \rightarrow mirar\ hacia\ arriba$	$F \rightarrow mirar\ hacia\ la\ izquierda$
$G \rightarrow mirar\ hacia\ abajo$	$H \rightarrow mirar\ hacia\ la\ derecha$

El movimiento de la cámara es producirá a medida que modifiquemos la matriz view, la cual es la encargada de transformar de coordenadas del mundo a coordenadas de la cámara. La matriz view será modificada por la operación de matrices rotación y matrices traslación.

En código de programación se traduciría a:

```
// se crea una matriz auxiliar para ser empleada como matriz de traslación
glm::mat4 tras(1.0f);
// se crea una matriz auxiliar para ser empleada como matriz de rotación
glm::mat4 rot(1.0f);
// se crea un ángulo auxiliar para la rotación en el eje x
static float alfa = 0.01f;
// se crea un ángulo auxiliar para la rotación en el eje y
static float beta = 0.01f;
// si se pulsa la 7 la cámara se traslada en dirección abajo
if (key == '7'){
		tras = glm::translate(tras, glm::vec3(0.0f, 0.1f, 0.0f));
		view = tras*view;
}
// si se pulsa la 9 la cámara se traslada en dirección arriba
if (key == '9'){
		tras = glm::translate(tras, glm::vec3(0.0f, -0.1f, 0.0f));
		view = tras*view;
}
// si se pulsa la 6 la cámara se traslada en dirección derecha
if (key == '6'){
		tras = glm::translate(tras, glm::vec3(-0.1f, 0.0f, 0.0f));
		view = tras*view;
}
// si se pulsa  el 4  la cámara se traslada en dirección izquierda
if (key == '4'){
		tras = glm::translate(tras, glm::vec3(0.1f, 0.0f, 0.0f));
		view = tras*view;
}
// si se pulsa la 8 la cámara se traslada en dirección frontal
if (key == '8'){
		tras = glm::translate(tras, glm::vec3(0.0f, 0.0f, 0.1f));
		view = tras*view;
}
// si se pulsa la 2 la cámara se traslada en dirección trasera
if (key == '2'){
		tras = glm::translate(tras, glm::vec3(0.0f, 0.0f, -0.1f));
		view = tras*view;
}
// si se pulsa la tecla t la cámara rota hacia arriba
if (key == 't'){
		rot = glm::rotate(rot, -alfa, glm::vec3(1.0f, 0.0f, 0.0f));
		view = rot*view;
}
// si se pulsa la tecla g la cámara rota hacia abajo
if (key == 'g'){
		rot = glm::rotate(rot, alfa, glm::vec3(1.0f, 0.0f, 0.0f));
		view = rot*view;
}
// si se pulsa la tecla f la cámara rota hacia la derecha
if (key == 'f'){
		rot = glm::rotate(rot, -beta, glm::vec3(0.0f, 1.0f, 0.0f));
		view = rot*view;
}
```

```
// si se pulsa la tecla h la cámara rota hacia la izquierda
if (key == 'h'){
        rot = glm::rotate(rot, beta, glm::vec3(0.0f, 1.0f, 0.0f));
        view = rot*view;
}
```

Para evitar que se muevan las luces al mover la cámara, será necesario multiplicar los vectores L de las luces por la matriz View, para ello emplearemos el mismo procedimiento empleado con la iluminación y la intensidad.

Se declara la variable global:

```
GLint uView = -1;
```

Creamos el identificador de view:

```
// 1.4 asignamos la variable view
uView = glGetUniformLocation(program, "View")
```

Asignamos a la matriz view, la matriz view anterior:

```
glm::mat4 View = view;
```

A continuación en el shader de fragmentos, colocamos como entrada uniform, la matriz view, y se la multiplicamos al vector L

```
uniform mat4 View;

vec3 L = normalize(vec3(View*vec4(lpos, 1.0)-pos));
```