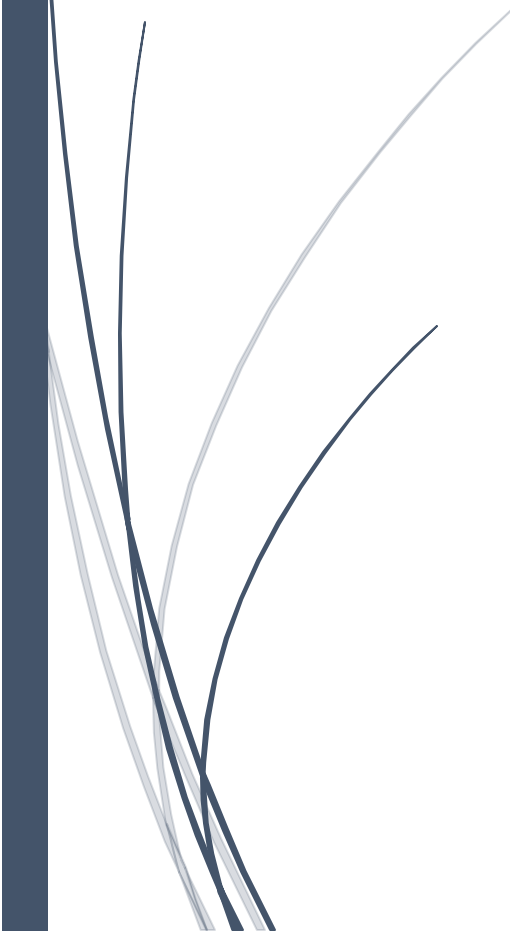


A dark blue vertical bar on the left side of the page. A blue arrow points to the right from the bar, containing the date.

2-12-2015

Práctica Opcional 3

Informática Gráfica

Several thin, curved lines in dark blue and light grey originate from the bottom left corner and curve upwards and to the right.

Sebastián Guisado Barreras y Susana Pineda De
Luelmo

1. Modifica las propiedades (intensidad y posición) de la luz a través de teclado.

Para poder modificar la intensidad y la posición del foco a través de teclado necesitamos una variable que pase la información de la posición y la intensidad del foco del main al shader de fragmentos, de esta forma lo que estamos haciendo es subir la información de memoria principal a la GPU.

Para pasar la información del main al shader de fragmentos creamos una variable uniform para cada uno de los valores.

En el shader de fragmentos las declaramos de la siguiente forma:

```
uniform vec3 posFoco;  
uniform vec3 Id;
```

Una vez declaradas las variables las utilizamos para el cálculo de la iluminación de la escena:

```
vec3 shade()  
{  
    vec3 c = vec3(0.0);  
    c = Ia * Ka;  
    vec3 L = normalize (posFoco - pos);  
    vec3 diffuse = Id * Kd * dot (L,N);  
    c += clamp(diffuse, 0.0, 1.0);  
  
    vec3 V = normalize (-pos);  
    vec3 R = normalize (reflect (-L,N));  
    float factor = max (dot (R,V), 0.01);  
    vec3 specular = Is*Ks*pow(factor,alpha);  
    c += clamp(specular, 0.0, 1.0);  
  
    c+=Ke;  
  
    return c;  
}
```

Una vez declaradas las variables uniform en el shader de fragmentos pasamos a implementarlo en el main.

Lo primero que tenemos que hacer es declarar dos nuevas variables que se encarguen de almacenar la posición y la intensidad del foco:

```
glm::vec3 posicionFoco = glm::vec3(0.0f, 0.0f,8.0f);  
glm::vec3 iluminacion = glm::vec3(1.0f);
```

Después declaramos las dos variables uniform encargadas de comunicar el shader de fragmentos y el main:

```
int uintensidad;  
int uPosFoco;
```

Una vez declaradas las variables encargadas de dar valor a la posición y a la intensidad del foco y las variables uniform encargadas de comunicar el shader con el main vamos a asociarlas, de forma que el valor de uintensidad y uPosFoco tengan el mismo valor que posicionFoco e iluminación, en un primer momento. Para ello en el Main en la función initShader introducimos las siguientes líneas de código:

```
uintensidad = glGetUniformLocation(program, "Id");  
uPosFoco = glGetUniformLocation(program, "posFoco");
```

En la función `renderFunc` vamos a indicar que si las variables `uniform` no tienen un valor asignado tome los valores de la variable `posFoco` o `Id`, dependiendo de cual de ellas se encuentre sin valores iniciales. De esta forma inicializamos las variables `uniform` al valor de las variables creadas para dar valor a la posición y a la intensidad del foco.

Nuevas variables que utilizan `posicionFoco` e iluminación para poder utilizar estas variables sin modificar su valor fuera de la función.

```
glm::vec3 posFoco = posicionFoco;  
glm::vec3 Id = iluminacion;
```

Indicamos que si las variables `uniform` no tienen un valor asignado le asignamos el valor de `posFoco` o de `Id` que hace referencia al valor de `posicionFoco` e iluminación. De esta forma inicializamos la posición y la intensidad del foco al comienzo de la ejecución de nuestro programa.

```
if (uintensidad != -1)  
    glUniform3f(uintensidad, Id.x, Id.y, Id.z);  
  
if (uPosFoco != -1)  
    glUniform3f(uPosFoco, posFoco.x, posFoco.y, posFoco.z);
```

Por último pasamos a implementar la función `keyboardFunc()` a partir de la cual modificaremos las propiedades del foco con el teclado.

En dicha función a partir de la variable `key`, que nos indica la tecla pulsada, vamos a crear un `switch` que se encargue de disminuir o aumentar la intensidad del foco dependiendo de las teclas “r” “g” “b” “t” “h” “n”. Utilizaremos las teclas “r” y “t” para aumentar y disminuir el valor de la coordenada `x` del vector encargado de la iluminación (`iluminacion`), las teclas “g” y “h” para la coordenada `y` del vector, y “b” y “n” para la coordenada `z` de dicho vector.

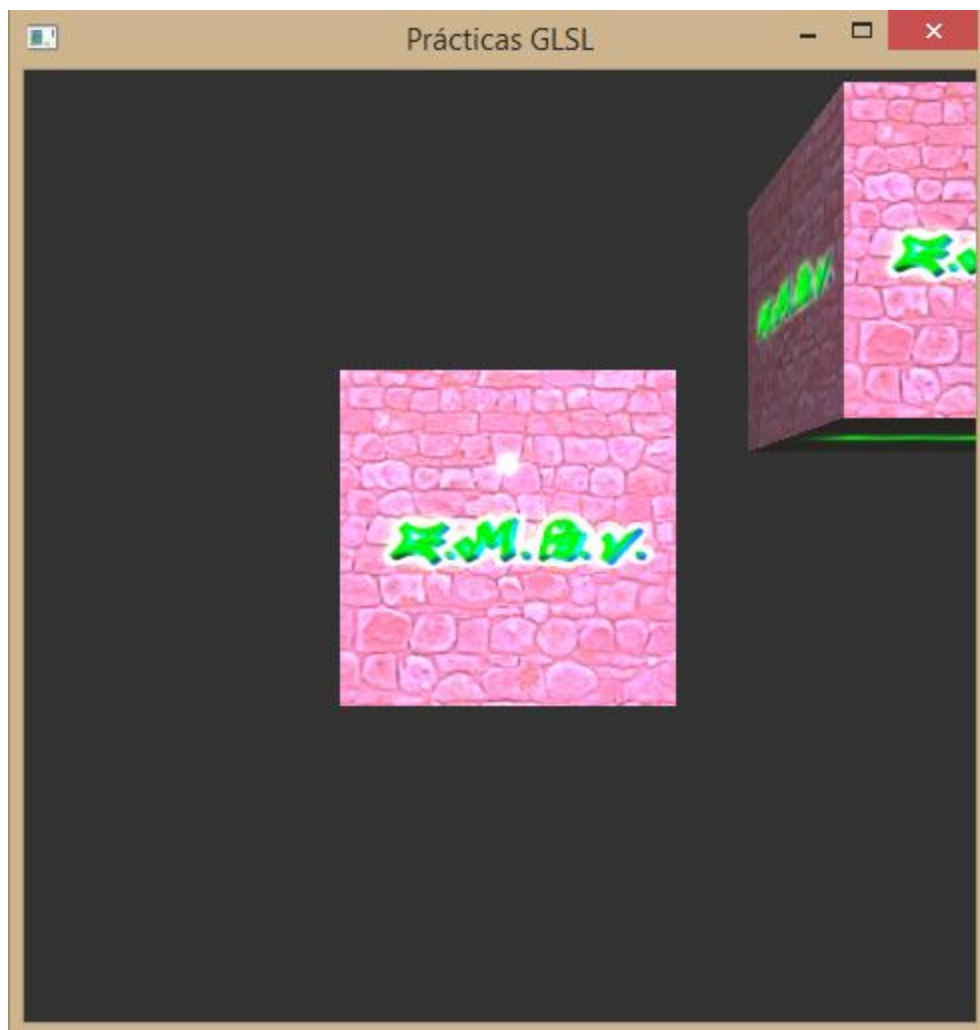
```
switch (key) {  
    //Posicion del foco  
    case 'i': // Movimiento acercar  
        posicionFoco.z -= 0.1;  
        break;  
    case 'k': // Movimiento alejar  
        posicionFoco.z += 0.1;  
        break;  
    case 'j': // Movimiento lateral izquierdo  
        posicionFoco.x -= 0.1;  
        break;  
    case 'l': // Movimiento lateral derecho  
        posicionFoco.x += 0.1;  
        break;  
    case 'o': // Movimiento arriba  
        posicionFoco.y += 0.1;  
        break;  
    case 'u': // Movimiento abajo  
        posicionFoco.y -= 0.1;  
        break;  
  
    //intensidad del foco  
    case 'r': // r++  
        iluminacion.x += 0.1;
```

```

        break;
case 't': // r--
    iluminacion.x -= 0.1;
    break;
case 'g': // g++
    iluminacion.y += 0.1;
    break;
case 'h': //g--
    iluminacion.y -= 0.1;
    break;
case 'b': // b++
    iluminacion.z += 0.1;
    break;
case 'n': // b--
    iluminacion.z -= 0.1;
    break;
}

```

De esta forma queda implementado el movimiento y el control de la intensidad del foco por teclado.



2. Define una matriz de proyección que conserve el aspect ratio cuando cambiamos el tamaño de la ventana.

Para que el aspect ratio se conserve cuando cambiamos el tamaño de la ventana tenemos que crear una nueva matriz de proyección que relacione el tamaño de la ventana en pixeles y en coordenadas de la pantalla.

En pixeles contamos con el alto de la pantalla (h) y el ancho de la pantalla (w)

En coordenadas de la pantalla contamos con el alto de la pantalla calculado como t-b y el ancho calculado como r-l.

Con esto podemos demostrar que el aspect ratio se relaciona con la siguiente formula:

$$aspect\ ratio = \frac{w}{h} = \frac{r-l}{t-b}$$

Por otra parte contamos con la matriz de proyección proporcionada por OpenGL:

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

En esta matriz podemos realizar diferentes cambios para que se ajuste al tamaño de la pantalla:

- Por un lado sabemos que $\frac{n}{r} = \frac{1}{\tan 30}$ debido al volumen de visualización ya que la apertura de la cámara es de 60º calculamos el alto del plano de proyección mediante la tangente de 30.

Contando con esta información podemos sustituirlo en la posición a00.

Por otra parte la posición a11 podemos transformarla mediante la fórmula del aspect ratio, ya que t - b va a ser un valor que desconozcamos. Esta posición podemos transformarla de la siguiente forma:

$$ar = \frac{w}{h} = \frac{r-l}{t-b} \Rightarrow t-b = \frac{r-l}{ar}$$

Sustituimos este resultado en la fórmula de la posición a11 y operamos:

$$\frac{\frac{2n}{r-l}}{ar} = \frac{2n * ar}{r-l} = \frac{ar}{\tan 30}$$

- Por otro lado podemos demostrar que las posiciones a02 y a22 van a dar 0.
- De esta forma la matriz de proyección nos quedaría de la siguiente forma:

$$\begin{pmatrix} \frac{1}{\tan 30} & 0 & 0 & 0 \\ 0 & \frac{ar}{\tan 30} & 0 & 0 \\ 0 & 0 & \frac{-(n+f)}{f-n} & \frac{-2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Esta nueva matriz la vamos a implementar en la función `resizeFunc()`, la cual será llamada cuando se detecte un cambio en el tamaño de la pantalla.

Lo primero que tenemos que hacer es declarar las variables que vamos a necesitar dentro de esta función.

Tenemos que transformar los int de entrada a float para que las divisiones sean más exactas:

```
float w = width;  
float h = height;
```

Declaramos la variable aspect en la que guardaremos w/h para no tener que calcularlo cada vez que lo necesitemos. Declaramos también la nueva matriz view que vamos a utilizar:

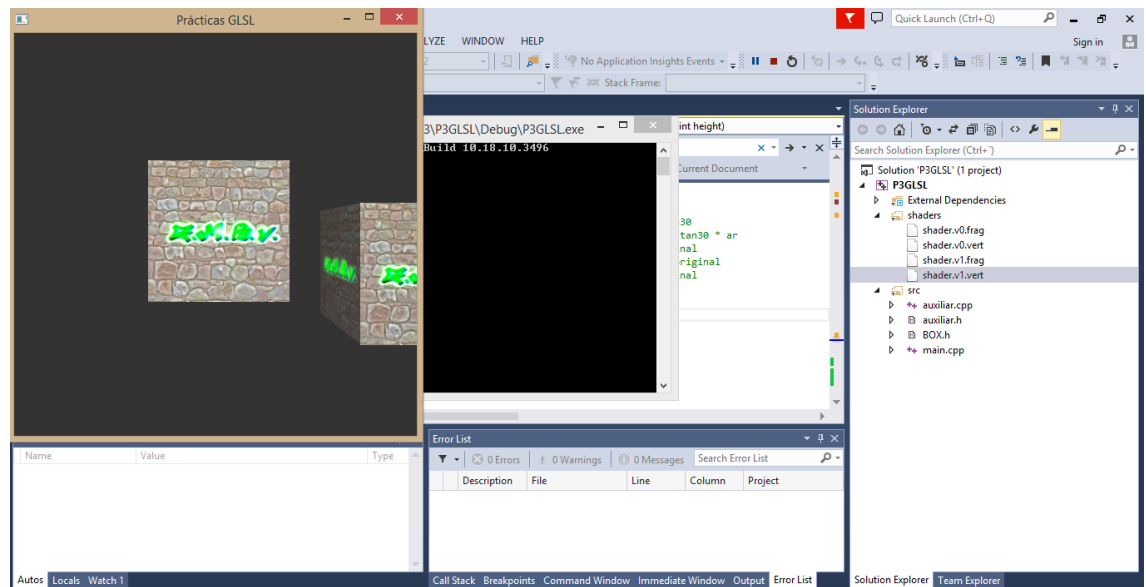
```
float aspect = w / h;  
glm::mat4 rproyec(0.0f);
```

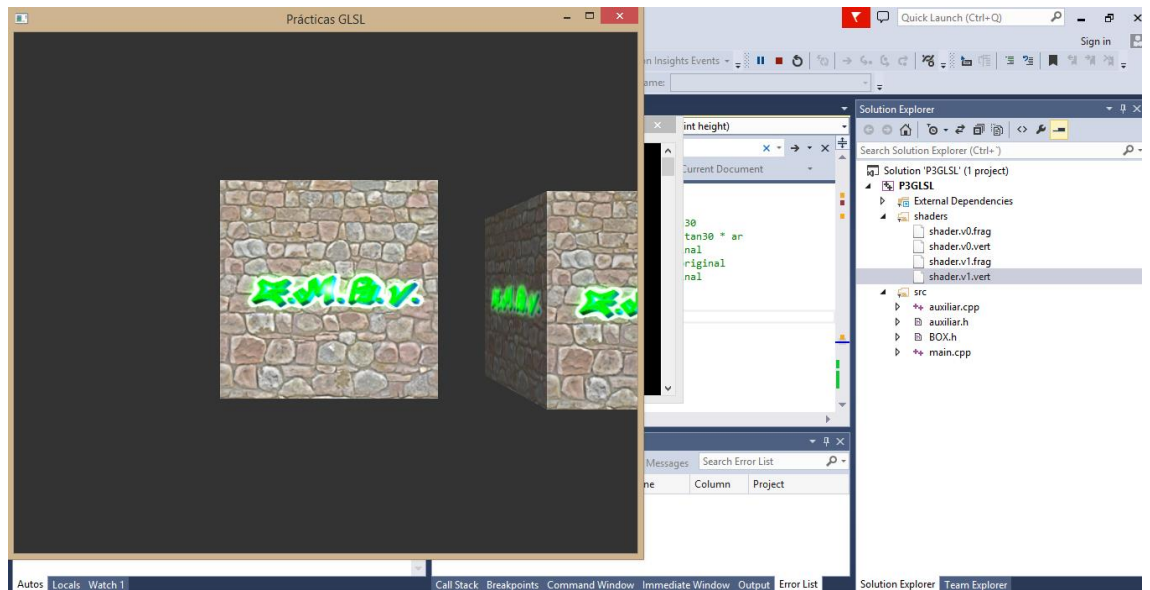
A continuación inicializamos la matriz con los valores que hemos calculado teóricamente antes:

```
rproyec[0].x = (1.0f / tan(3.14159f / 6.0f));  
rproyec[1].y = rproyec[0].x*aspect;  
rproyec[2].z = -(100 + 0.1) / (100 - 0.1);  
rproyec[2].w = -1.0f;  
rproyec[3].z = -2.0f * 100 * 0.1 / (100 - 0.1);
```

Por ultimo asignamos el valor de esta nueva matriz a la matriz proyeccion:

```
proyeccion = rproyec;
```





Como podemos observar, ahora se conserva la proporción de los objetos cuando cambiamos el tamaño de la ventana de visualización.

3. Añade un nuevo cubo a la escena. El segundo cubo deberá orbitar alrededor del primero describiendo una circunferencia a la vez que rota sobre su eje Y.

Para pintar el segundo cubo no tenemos que crear un segundo cubo, si no que tenemos que crear otra matriz model para pintarla con el mismo VAO.

Lo primero que tenemos que hacer es declarar la nueva matriz model para el segundo cubo:

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

En la función initObj() asociamos la nueva matriz (model2) a una matriz que más tarde daremos valor.

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

Por ultimo en la función de renderizado (renderFunc) pasamos a definir las propiedades de la matriz model2, y por consiguiente del segundo cubo. Para ello hacemos lo mismo que con el primer cubo pero con la nueva matriz model (model2):

```
//segundo cubo
modelView = view * model2;
modelViewProyeccion = proyeccion * view * model2;
normal = glm::transpose(glm::inverse(modelView));

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

if (uModelViewProyeccion != -1)
    glUniformMatrix4fv(uModelViewProyeccion, 1, GL_FALSE,
        &(modelViewProyeccion[0][0]));

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

//pintamos el segundo cubo:
glBindVertexArray(vao);
glDrawElements(GL_TRIANGLES, cubeNTriangleIndex *
3, GL_UNSIGNED_INT, (void*)0);
glDrawElements(GL_TRIANGLE_STRIP, cubeNTriangleIndex,
GL_UNSIGNED_SHORT, (void
*) (cubeNTriangleIndex*sizeof(GLushort)));
```

Por ultimo definimos el movimiento del Segundo cubo en la función idleFunc() :

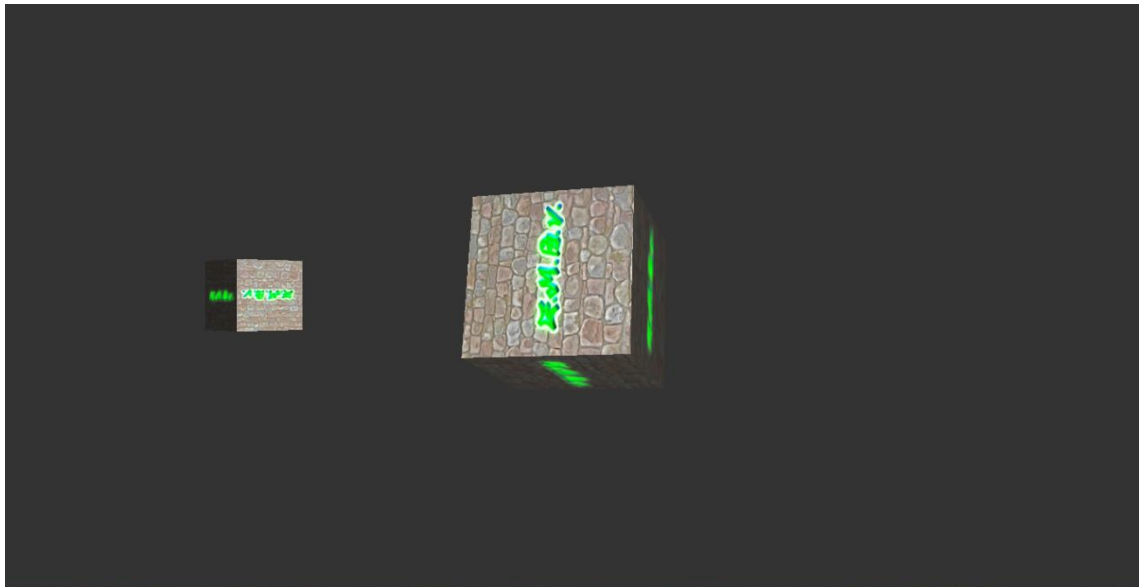
```
model2 = glm::mat4(1.0f);

//traslacion y escalado
model2 = glm::rotate(model2, angle, glm::vec3(0.0f, 1.0, 0.0f)
glm::mat4 model2tras(1.0f);
model2tras = glm::translate(model2tras, glm::vec3(5.0f, 0.0f, 0.0f));

glm::mat4 escalado2(0.5f);
escalado2 = glm::scale(escalado2, glm::vec3(0.5f, 0.5f, 0.5f));

model2 *= model2tras*escalado2*model2;
```


Lo primero que hacemos es definir la matriz model otra vez para poder utilizarla en la función `idleFunc()`. Una vez que la tenemos otra vez definida, en la siguiente línea de código, damos valor a la matriz de rotación. Vamos a rotar y a trasladar para hacer que el cubo orbite alrededor del primer cubo. En la tercera línea de código declaramos la matriz de traslación y en la siguiente línea de código le vamos a dar valor. En la quinta línea de código declaramos una matriz de escalado para el segundo cubo y en la línea siguiente le damos valor. Por ultimo lo único que tenemos que hacer es multiplicar estas matrices en orden y asociar el resultado a la matriz `model2` que será a partir de la cual pintemos nuestro segundo cubo.



4. Control de la cámara con el teclado. La posición de la luz debe ser invariante con respecto a la posición de la cámara.

Para controlar la cámara por teclado, haremos como en la primera práctica. Para ello, en la función `keyboardFunc()` ampliaremos el switch implementado en la primera parte de la practica con el movimiento de la cámara.

```
switch (key) {
//Giro de la camara
case '8': // Giro camara arriba
view = glm::rotate(glm::mat4(1.0f), -0.1f, glm::vec3(1.0f, 0.0f,
0.0f)) * view;
break;

case '5': // Giro camara abajo
view = glm::rotate(glm::mat4(1.0f), 0.1f, glm::vec3(1.0f, 0.0f, 0.0f))
* view;
break;

case '4': // Giro camara derecha
view = glm::rotate(glm::mat4(1.0f), -0.1f, glm::vec3(0.0f, 1.0f,
0.0f)) * view;
break;

case '6': // Giro camara izquierda
view = glm::rotate(glm::mat4(1.0f), 0.1f, glm::vec3(0.0f, 1.0f, 0.0f))
* view;
break;

//movimiento de la camara
case 'q': // Movimiento adelante
view = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, 0.1f)) *
view;
break;

case 'e': // Movimiento detras
view = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, -0.1f)) *
view;
break;

case 'a': // Movimiento lateral izquierdo
view = glm::translate(glm::mat4(1.0f), glm::vec3(-0.1f, 0.0f, 0.0f)) *
view;
break;

case 'd': // Movimiento lateral derecho
view = glm::translate(glm::mat4(1.0f), glm::vec3(0.1f, 0.0f, 0.0f)) *
view;
break;

case 'w': // Movimiento arriba
view = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.1f, 0.0f)) *
view;
break;

case 's': // Movimiento abajo
view = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, -0.1f, 0.0f)) *
view;
break;
}
```

Una vez que tenemos implementado el movimiento por teclado de la cámara tenemos que hacer que las luces no se muevan con el movimiento de la cámara, para ello tenemos que multiplicar los vectores L de las luces (shader de fragmentos) por la matriz view. Para ello creamos una variable uniform que comunique el shader de fragmentos con el Main.

Lo primero que tenemos que hacer es declarar la nueva variable uniform en el Main:

```
GLint View = -1;
```

Creamos un identificador para la view en la función initShader():

```
uView = glGetUniformLocation(program, "View");
```

Y por último asignamos esta matriz a la matriz view en la función renderFunc():

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

De esta forma queda inicializada la variable uniform en el Main. A continuación pasamos a multiplicarla por los vectores L en el shader de fragmentos.

Declaramos la matriz uniform en el shader:

```
uniform mat4 View;
```

Por ultimo multiplicamos L por la variable uniform en la función shade()

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