Práctica 1:

Parte obligatoria

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

Aspect Ratio

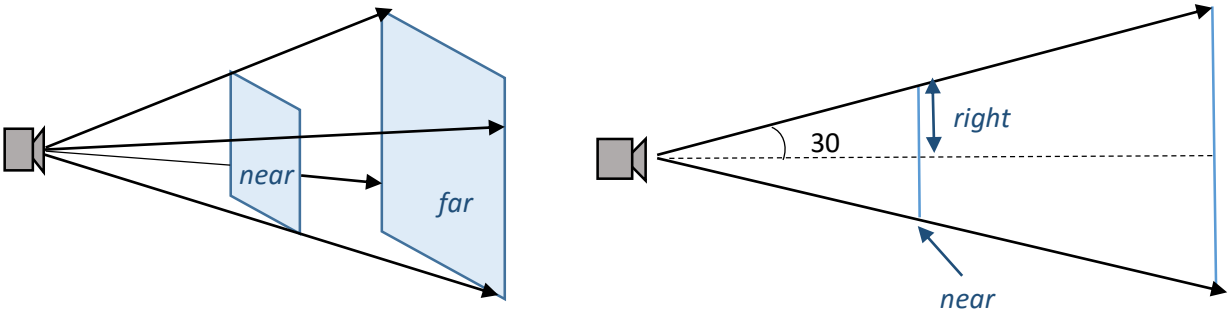
El aspect ratio es el cociente resultante de dividir el ancho entre el alto de una imagen o pantalla.

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

En nuestro enunciado se determinaba que la vista deseada del objeto debía ser de una apertura de 60 grados. Esto se logra, gracias a la matriz perspectiva la encargada de mostrar en la pantalla los objetos visibles en la escena:

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

Al tener que formar 60 grados de apertura, se puede observar que si mirásemos inferiormente la imagen mostrada a la izquierda, obteniendo la imagen de la derecha:



Tendríamos que *right* forma 30 grados con el plano *near.* A su vez se nos ha establecido:

$$right = -left$$

$$top = -bot$$

$$|right| = |-left| = |top| = |-bot|$$

Lo que causa que la matriz proyección cambie ligeramente:

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

Como se mencionó anteriormente *near* y *right* forman 30 ángulos por lo que se llega a la siguiente:

$$\frac{2*near}{2*right} = \frac{near}{right} \rightarrow tan30 = \frac{right}{near} \rightarrow \frac{1}{tan30} = \frac{near}{right};$$

Por lo que a00 se puede cambiar por 1/tan30. En la clase main de la práctica se nos ofrece un método donde podamos hacer que el aspect ratio se mantenga constante para ello se nos ofrece con la siguiente etiqueta:

```
void resizeFunc(int width, int height) { }
```

Se puede observar con ella que se nos proporciona los valores de width y height (ancho y alto), el ancho y el alto están en la unidad de píxeles mientras que top y right están en coordenadas de la pantalla, por lo que haremos una serie de cálculos para poder mantener el aspect ratio se mantenga constante:

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

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

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

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

Con la serie de operaciones anterior llegamos a la conclusión de que para que el aspect ratio se mantenga debemos modificar de nuevo la matriz proyección:

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

Esto en el código de programación se traduciría en lo siguiente:

```
void resizeFunc(int width, int height) {
```

```
 glm::mat4 rproj (0.0f); // se crea la matriz que emplearemos como nueva matriz proyección

 if (height == 0) height = 1; // para prevenir indeterminación impedimos el valor nulo

 float w = width; // para obtener decimales, se cambia el valor de entero a float
 float h = height; // para obtener decimales, se cambia el valor de entero a float
 float aspect = (w / h); // calculamos el cociente de aspect ratio a partir de width y height
 float far = 100.0f;  // establecemos el valor a la variable far
 float near = 0.1f;  // establecemos el valor a la variable near

 rproj[0].x = (1.0f / tan(3.14159f/6.0f)); // el primer valor como antes se explicó es 1/tan30
 rproj[1].y = rproj [0].x*aspect; // el segundo valor como se mencionó arriba es 1/tan30 * ar
 rproj[2].z = -(far + near) / (far - near); //se calcula el valor de acuerdo con la matriz original
 rproj[2].w = -1.0f; //se calcula el valor de acuerdo con la matriz original
 rproj[3].z = -2.0f * far * near / (far - near); //se calcula el valor de acuerdo con la matriz original

 IGlib::setProjMat(rproj); //le indicamos a la clase main que la matriz rproj es la matriz proyección actual

}
```

Añade un nuevo cubo a la escena

En la función Idle, se realizan los movimientos de rotación, traslación y escalado; tanto del cubo dado como recurso de la práctica como del creado más tarde por nosotros.

Para crearlo, ha sido necesario crear un nuevo objeto y llevarlo al mundo virtual. Se ha creado usando la siguiente línea de código, en el método main:

objId2 = IGlib::createObj(cubeNTriangleIndex, cubeNVertex, cubeTriangleIndex, cubeVertexPos, cubeVertexColor, cubeVertexNormal, cubeVertexTexCoord, cubeVertexTangent);

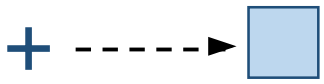
Tras esto, en la función Idle se han declarado las transformaciones geométricas que se le deben aplicar a los dos cubos. En este caso, se han aplicado a vectores columna, como es habitual en gráficos por computador. Para ello, se ha seguido la siguiente ecuación para aplicar, en ese orden, un escalado, rotación sobre sí mismo del objeto entorno a su eje Y, una traslación en el eje X y una rotación en torno al eje Y del origen, donde se sitúa el primer cubo.

La ecuación de la matriz de la transformación sería:

M = Rotación Eje Y Origen * Traslación * Rotación sobre sí mismo * Escalado

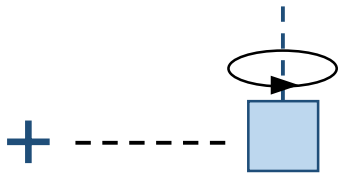
Para ello, las matrices usadas serían:

$$\begin{pmatrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



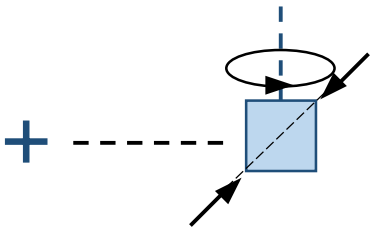
Traslación en el eje X (5 posiciones a la derecha)

$$\begin{pmatrix} \cos\alpha & 0 & sen\,\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -sen\,\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Rotación sobre el eje Y

$$\begin{pmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Escalado uniforme en todas las componentes

Teniendo en cuenta la ecuación anterior, se observa que se aplican dos rotaciones. La matriz de rotación es igual en ambos casos pero con la diferencia de que la primera se aplica en el origen de coordenadas, lo que hace que el objeto gire sobre sí mismo y la otra en torno al eje Y del origen, dando lugar a la rotación que se desea. El escalado es uniforme y reduce su tamaño en todos sus vértices.

Al realizar la operación de multiplicación de las matrices de rotación sobre el eje Y del objeto y el escalado, se observa que es conmutativo, por lo que el orden de estas dos matrices no influye en la ecuación planteada, matemáticamente hablando.

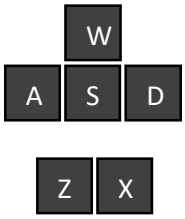
En cambio, esto no puede hacerse con la traslación y la otra rotación, ya que es necesario tener en cuenta la posición de la traslación para las rotaciones en torno al eje Y del origen.

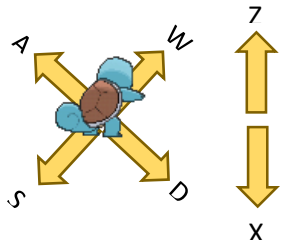
En código, quedaría así:

```
//creamos un angulo de giro estatico para mantenerla de una ejecucion a otra
void idleFunc()
{

//cubo central
	//angulo de rotacion
	static float angle = 0.0f;
	//modificacion del angulo
	angle = (angle <= 2.0f*3.15159) ? angle + 0.001f : 0.0f;
	//declaracion de la matriz del primer modelo
	glm::mat4 model(1.0f);
	//se transforma la matriz del primer modelo en una de rotacion
	model = glm::rotate(model, angle, glm::vec3(1.0f, 1.0f, 0.0f));
	//Se cambia la matriz del modelo por una de rotacion
	IGlib::setModelMat(objId, model);


//cubo rotatorio
	// Declarar la matriz de rotacion
	glm::mat4 model2rot(1.0f);
	// Dar valor a la matriz de rotación
	model2rot = glm::rotate(model2rot, angle, glm::vec3(0.0f, 1.0, 0.0f));
	// Declarar la matriz de traslacion
	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));
	// multiplicar las matrices de las transformaciones
	model2rot *= model2tras*model2rot*escalado2;
	// Se cambia la matriz del modelo por una con las transformaciones(se ha elegido usar
	// la de rotacion ya que se almacenan los valores)

IGlib::setModelMat(objId2, model2rot);
}
```

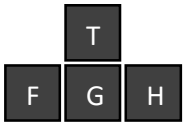
Control de la cámara con el teclado

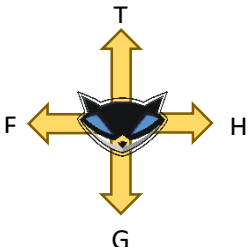
Los movimientos de la cámara se asemejarán a los controles de un videojuego actual, es decir, de la siguiente manera:





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





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

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.

Para realizar la traslación se empleará una simple matriz de traslación:

$$\begin{pmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Siendo dx la cantidad a trasladar en el eje x, dy la cantidad a trasladar en el eje y, y dz la cantidad a trasladar en el eje z. Asignaremos valores diferentes a dx, dy y dz dependiendo de la tecla que se pulse:

Tecla	dx	dy	dz
W	0.0f	0.0f	0.1f
S	0.0f	0.0f	-0.1f
A	0.1f	0.0f	0.0f
D	-0.1f	0.0f	0.0f
Z	0.0f	0.1f	0.0f
X	0.0f	-0,1f	0.0f

Para realizarse la rotación emplearemos dos ángulos, un ángulo alfa para el eje x, y un ángulo beta para el eje y:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(alfa) & -sen\,(alfa) & 0 \\ 0 & sen\,(alfa) & \cos\,(alfa) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} \cos\,(beta) & 0 & sen\,(beta) & 0 \\ 0 & 1 & 0 & 0 \\ -sen\,(beta) & 0 & \cos\,(beta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Esto en lenguaje de programación se traduciría en:

```
void keyboardFunc(unsigned char key, int x, int y) // Control de la cámara con el teclado
{
	// se muestra por consola que tecla se ha pulsado
	std::cout << "Se ha pulsado la tecla " << key << std::endl << std::endl;

	//traslacion
	// se crea una matrix auxiliar para ser empleada como matriz de traslación
	glm::mat4 tras(1.0f);

	//rotacion
	// 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 z la cámara se traslada en dirección abajo
if (key == 'z'){
	// se configura la matriz de traslación para ser trasladada en el eje Y
	tras = glm::translate(tras, glm::vec3(0.0f, 0.1f, 0.0f));
	// se multiplica la matriz de traslación por la matriz view
	view = tras*view;
}
```

```
// si se pulsa la x la cámara se traslada en dirección arriba
if (key == 'x'){
		// se configura la matriz de traslación para ser trasladada en el eje Y
		tras = glm::translate(tras, glm::vec3(0.0f, -0.1f, 0.0f));
		// se multiplica la matriz de traslación por la matriz view
		view = tras*view;
}

// si se pulsa la d la cámara se traslada en dirección derecha
if (key == 'd'){
		// se configura la matriz de traslacion para ser trasladada en el eje X
		tras = glm::translate(tras, glm::vec3(-0.1f, 0.0f, 0.0f));
		// se multiplica la matriz de traslación por la matriz view
		view = tras*view;
}

// si se pulsa la a la cámara se traslada en dirección izquierda
if (key == 'a'){
		// se configura la matriz de traslación para ser trasladada en el eje X
		tras = glm::translate(tras, glm::vec3(0.1f, 0.0f, 0.0f));
		// se multiplica la matriz de traslación por la matriz view
		view = tras*view;
}

// si se pulsa la w la cámara se traslada en dirección frontal
if (key == 'w'){
		// se configura la matriz de traslación para ser trasladada en el eje Z
		tras = glm::translate(tras, glm::vec3(0.0f, 0.0f, 0.1f));
		// se multiplica la matriz de traslación por la matriz view
		view = tras*view;
}

// si se pulsa la s la cámara se traslada en dirección trasera
if (key == 's'){
		// se configura la matriz de traslación para ser trasladada en el eje Z
		tras = glm::translate(tras, glm::vec3(0.0f, 0.0f, -0.1f));
		// se multiplica la matriz de traslación por la matriz view
		view = tras*view;
}

// si se pulsa la tecla t la cámara rota hacia arriba
if (key == 't'){

		// se configura la matriz de rotacion para ser rotada en el eje X
		rot = glm::rotate(rot, -alfa, glm::vec3(1.0f, 0.0f, 0.0f));
		// se multiplica la matriz de rotación por la matriz view
		view = rot*view;
}

// si se pulsa la tecla g la cámara rota hacia abajo
if (key == 'g'){
		// se configura la matriz de rotacion para ser rotada en el eje X
		rot = glm::rotate(rot, alfa, glm::vec3(1.0f, 0.0f, 0.0f));
		// se multiplica la matriz de rotación por la matriz view
		view = rot*view;
}
```

```
// si se pulsa la tecla f la cámara rota hacia la derecha
if (key == 'f'){
        // se configura la matriz de rotacion para ser rotada en el eje Y
        rot = glm::rotate(rot, -beta, glm::vec3(0.0f, 1.0f, 0.0f));
        // se multiplica la matriz de rotación por la matriz view
        view = rot*view;
}

// si se pulsa la tecla h la cámara rota hacia la izquierda
if (key == 'h'){
        // se configura la matriz de rotacion para ser rotada en el eje Y
        rot = glm::rotate(rot, beta, glm::vec3(0.0f, 1.0f, 0.0f));
        // se multiplica la matriz de rotación por la matriz view
        view = rot*view;
}

// se indica a la clase main que la matriz view se actualiza
IGlib::setViewMat(view);
}
```