Apellidos:	Nombre:
Apellidos:	Nombre:

Responder a las preguntas y adjuntad las gráficas y los "trozos" de código pedidos. Entregad este fichero + los ficheros fuente pedidos (animacion.cpp y thor.cpp) en un zip o similar.

En esta tarea vamos a presentar las bases de cómo animar una figura. Aunque el ejercicio planteado será muy sencillo, espero que os sirva para entender el planteamiento de un problema de animación, los problemas a resolver y algunas de sus posibles soluciones.

En principio para crear una animación podríamos hacer como en el cine: mandar cada cierto tiempo (cada frame) una versión "modificada" de los vértices en sus nuevas posiciones a la GPU. En cada instante de tiempo los vértices se habrán movido un poco dando ilusión de movimiento a la figura.

Sin embargo este enfoque sería prohibitivo para cualquier modelo un poco complicado, creando un cuello de botella en la transferencia de datos entre la CPU y la GPU. Además va en contra de nuestra filosofía de mandar los datos una vez y "cambiarlos" en la GPU mientras se dibujan, a base de aplicarles diferentes matrices de transformación.

El problema es que las transformaciones vistas hasta ahora no permiten una animación, ya que mueven nuestro modelo como si fuera un sólido rígido. Lo trasladan o giran en bloque, pero no son capaces de hacer que una parte del modelo tenga un movimiento diferenciado respecto a las otras.

La razón es que la misma matriz M se aplica a todos los vértices del modelo. La solución será dividir el modelo en diferentes partes y aplicarles matrices distintas. En la práctica lo que se hace es asociar cada vértice a lo que se denomina un "hueso" (bone). Al estar el modelo repartido entre diferentes "huesos" podremos asociar un movimiento distinto (de translación o rotación) a cada hueso.

En el "vertex shader" a cada vértice se le aplica una matriz M distinta, que depende del hueso al que esté asociado (que se pasa como un atributo más del vértice, como su posición o su color). Por supuesto a todos los vértices les aplicamos las mismas matrices V / P ya que esas sólo dependen de la cámara.

Compilar y correr el programa GpO_anima.cpp de esta tarea, usando el proyecto habitual. El programa carga un modelo 3D del fichero cilindro3.bix (que como siempre debe estar en el directorio de ejecución, normalmente ./Debug).

Es un modelo muy sencillo de lo que podría ser parte de una "pierna", un cilindro cuyo grosor disminuye ligeramente de arriba a abajo, empezando con un radio de 1.1 en la parte superior (Z=0), y reduciéndose a un radio de 0.8 en la parte inferior (Z=-4). La visualización es en modo "wireframe" (líneas).

El observador se encuentra en la posición x=8, y=8, mirando un poco desde arriba (z=2). Con las teclas de cursores (Izq/der) podéis cambiar la posición del observador y observar el modelo desde otros puntos de vista.

La aplicación (en render_scene) calcula las matrices de proyección P y del cambio de ejes V (basada en la posición del observador). Para la matriz M del movimiento del modelo se usa la matriz identidad en una primera instancia. El producto P·V·M se transfiere a la GPU para que sea usado en el procesador de vértices. Al ser M la identidad nuestro modelo se muestra quieto en pantalla.

Lo primero que haremos es modificar la matriz M, usando translate y rotate para que el modelo gire y se mueva por pantalla. Se trata de:

a) Aplicar una rotación R alrededor del eje Y de forma que los grados girados correspondan a 40·sin(1.5·tt), lo que corresponde a un giro oscilante con una amplitud de +/- 40º.

b) Aplicar una translación T con tx=2·sin(tt), ty=0, tz=0 para impartir un movimiento entre +/-2 a lo largo del eje X.

A partir de R y T construir M como M=T·R (primero rotar, luego trasladar) y usarla en vez de la identidad. Correr el programa. Como la matriz se aplica a todos los vértices en el código del "vertex shader" el cilindro se mueve de forma "solidaria" siguiendo el movimiento descrito (translación + rotación).

Adjuntar captura de la pantalla.

Vamos ahora a crear una articulación en el modelo. En una aplicación real cada vértice recibiría (como un atributo adicional) un índice indicando a qué hueso está "atado". En nuestro caso, como queremos un programa lo más sencillo posible, usaremos directamente la coordenada z para definir el hueso al que pertenece un vértice:

- Tendremos 2 "huesos" en el modelo: los vértices de la mitad para arriba (coordenada Z>=-2) pertenecerán a un hueso y los de la mitad de abajo (Z<-2) al otro. La articulación se sitúa por lo tanto en Z=-2.

- De esta forma, en el programa del vertex shader podemos saber de forma sencilla a que hueso pertenece cada vértice sin más que consultar la variable pos.z.

Añadir las siguientes modificaciones al programa de partida:

En el código del vertex shader:

- Declarar una 2ª matriz (MVP2) también de tipo uniform mat4.
- En el main() calcular una 2ª posición pos2 aplicando MVP2 a la posición original del vértice.
- Para la posición de salida (gl_Position) usar pos1 para los vértices con pos.z>=-2 (parte de arriba) y pos2 para aquellos con pos.z <-2 (parte de abajo). La sintaxis es como en C:

  if (cond) x=aaa; else x=bbb;

  Alternativamente x = (cond)? aaa:bbb; también es aceptado.

En la aplicación (render_scene):

- Usando R2=rotate(θ,eje) crear una matriz de giro para aplicar a la parte de abajo del modelo con (0,1,0) como eje de giro y rotación θ dada por:

  $$\theta = 40*(1-\cos(2*tt)).$$

  Esto corresponde a una rotación que oscila entre 0 y 80º simulando una articulación como una rodilla o codo que solo gira en una dirección.

- Crear las matrices (P·V)·M y (P·V)·R2 y transferirlas a las variables "MVP1" y "MVP2" respectivamente de la GPU. De esta forma la parte superior e inferior del cilindro tendrán movimientos distintos (dados por la matrices M y R2), formándose una articulación en Z=-2. Las matrices P y V son comunes ya que el "observador" es el mismo.

Adjuntar vuestro vertex shader modificado y las 2/3 líneas de código añadidas en el código de la aplicación.

Adjuntar una captura de la pantalla donde se aprecie la "articulación".
¿Qué problemas observáis?

En la práctica los "huesos" de un modelo están organizados en una jerarquía (árbol o esqueleto) , de forma que un hueso hereda los movimientos de su hueso "padre", además de tener (posiblemente) giros propios.

Por ejemplo, una cabeza depende del tronco. Si el tronco se mueve con una translación T la cabeza debe seguirle, pero puede tener su propio movimiento de rotación R. Al expresar el movimiento de los vértices de la cabeza debemos incluir tanto la translación T del cuerpo como su propia rotación R.

En la articulación que nos ocupa, queremos que la parte de abajo tenga un movimiento descrito por el giro R2 respecto a la articulación (centro del modelo). El problema es que la articulación también se mueve de acuerdo con el movimiento M que experimenta la parte superior.

Para saber cuál es la matriz M2 que captura los movimientos de la parte inferior debemos combinar su propio giro (R2) con el movimiento (M) de la parte superior. Si p es la posición de la articulación, el proceso completo es:

- Aplicar M (el 2º hueso comparte movimiento del 1º)
- Translación (-M·p) para llevar la articulación p una vez movida al origen.
- Aplicar rotación R2
- Tras la rotación, deshacer la translación anterior (M·p).

Matricialmente, el movimiento del 2º hueso se expresaría:

$$M2 = T(M \cdot p) \cdot R2 \cdot T(-M \cdot p) \cdot M$$

donde M·p es la posición de la articulación p tras aplicarle el movimiento M.

Podemos calcular M2 en el código de render_scene() con los pasos siguientes:

- Definir vector p (vec3) con las coordenadas de la articulación (0,0,-2).
- Extenderlo a coordenadas homogeneas vec4(p,1) y aplicarle la matriz M.
- Definir Mp como el vector (vec3) con las tres primeras componentes del resultado M·p. Usar vec3() para extraer esas 3 componentes.
- Finalmente calcular la matriz M2 = T(Mp) · R2 · T(-Mp) · M. Usad la función translate para crear las matrices de translación correspondientes a los vectores -Mp y Mp.

La matriz combinada resultante M2 es la que hay que usar para el movimiento del "hueso" inferior, combinando el movimiento M de la articulación superior con el giro propio.

Modificar el código (en la aplicación) para calcular M2 y enviar las nuevas matrices a la GPU (combinándolas con las matrices P y V que son comunes)

Adjuntar modificaciones del código.

Ejecutar el programa y observar como ahora la parte inferior además de girar por su cuenta si que acompaña a la superior en su movimiento. Adjuntar captura de pantalla. ¿Alguna otra cosa que se vea incorrecta? Describidla.

Vamos a tratar de saltar ese último escollo. El problema viene del carácter binario de la asignación de vértices a los huesos, lo que se traduce en una bisagra "mecánica" (pensad en una biela o la articulación de una marioneta): los dos trozos de "pierna" se mueven de forma independiente.

Afortunadamente esto puede arreglarse de forma sencilla (aunque con un poco más de trabajo) en el vertex_shader. La idea es hacer que la asignación de los vértices a los huesos no sea binaria, de forma que vértices cercanos a una articulación pueden estar asociados a ambos huesos simultáneamente

En aquellos vértices asociados a ambos huesos no debemos aplicar MVP1 **o** MVP2 sino ambas. Obviamente esto tiene el problema de que obtendremos dos posiciones pos1 y pos2 para cada vértice. ¿Cuál de ellas se asignará a la posición definitiva gl_Position?

La solución será usar una media ponderada de ambas posiciones. Por ejemplo

```
c1*pos1 + c2 *pos2;
```

De nuevo, en la práctica, los coeficientes de ponderación c1 y c2 se pasarían como atributos del vértice, indicando lo mucho/poco que esté ligado a cada "hueso". En nuestro caso, en vez de pasarse como atributos, los vamos a calcular sobre la marcha en función de su distancia a la articulación.

La idea es que los puntos más alejados de la articulación (en z=-2) sean estrictamente de un hueso (c1≅1, c2≅0) o del otro (c1≅0, c2≅1). Conforme nos acerquemos a la articulación empezarán a mezclarse, hasta que en los puntos que están sobre la articulación tengamos c1≅c2≅0.5.

Para calcular los coeficientes de mezcla en este caso (con la articulación en Z=-2) definiremos una variable t como:

```
t = -Z/4    (t=0 para z=0, arriba y  t=1 para z=-4, abajo)
```

A partir de t podríamos definir los coeficientes de mezcla como c1=(1-t) y c2=t. Pero ni siquiera es necesario, ya que este tipo de "mezclas" son tan comunes en gráficos que el lenguaje GLSL tiene una función llamada mix():

```
y = mix(x1,x2,t)
```

La función mix interpola linealmente entre los datos x1 y x2 dependiendo del parámetro t en el intervalo [0,1]. Si t=0 -> y=x1 y si t=1 -> y=x2. Los datos a interpolar x1 y x2 pueden ser escalares o vectores. En nuestro caso serán las posiciones pos1 y pos2. t es justo el parámetro (-Z/4) definido antes con valores entre 0 y 1 y que toma un valor de 0.5 en la articulación. La salida de mix() es la posición "mezcla" que asignaremos directamente a gl_Position.

Modificar el vertex shader para incluir las modificaciones indicadas.
Adjuntar vertex shader modificado.

Ejecutar el programa y adjuntad captura de la pantalla donde se aprecie la "flexión" de la articulación.

El resultado está todavía lejos de ser realista y se parece a un tubo de goma, con demasiada flexibilidad. La razón es que nos hemos pasado en lo de hacer que los vértices "compartiesen" huesos. Con nuestra definición anterior de t, incluso vértices muy cerca de la parte superior están ya unidos al hueso de abajo y se ven afectados por su movimiento.

Para hacerlo más realista es necesario que la zona de transición entre t=0 y t=1 sea más estrecha alrededor de la articulación. Los vértices por encima o debajo de esa zona tendrán t=0 (estrictamente hueso de arriba) o t=1 (estrictamente hueso de abajo). De nuevo este requerimiento es muy habitual en gráficos y GLSL tiene una función justo para eso:

t = smoothstep(z0,z1,z)

La función smoothstep devuelve 0 si z<z0 y 1 si z>z1. Para valores en el intervalo [z0,z1] devuelve un valor intermedio con una transición suave entre t=0 (z0) y t=1 (z1). La salida de smoothstep puede usarse directamente como el parámetro t de la función mix() anterior

Usar smoothstep definiendo la zona de transición de +/-0.5 por encima y debajo de la articulación.

Adjuntar código del vertex_shader modificado y una captura del resultado.

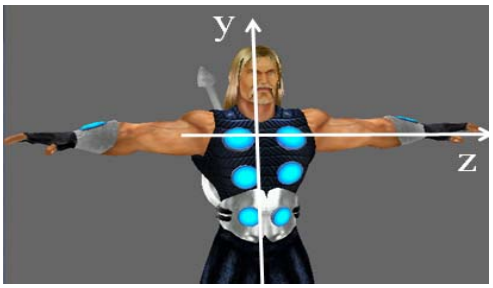
----- **Hasta aquí un 50% de la nota.** -----

(15%) (no necesaria para el resto de la tarea): Completar vuestro código para que usando los cursores arriba y abajo se pueda variar el ancho de la zona de transición. Empezad con el ancho de +/- 0.5 indicado antes y aumentarlo/disminuirlo por 0.01 según se pulsen las teclas up/down (poner un control para que dicho ancho no sea menor que 0.01 ni mayor que 2.00. Fijaros que para que los programas en la GPU sean conscientes de ese valor deberéis definir la correspondiente variable uniform en el fragment_shader y actualizar su valor desde la aplicación.

Adjuntad los trozos de código añadidos en la aplicación y los shaders.

En cualquiera de los casos (con o sin esta última modificación) adjuntad vuestro código fuente (llamarlo animacion.cpp) junto con la hoja de respuestas.

(20%) Finalmente usando el modelo de Thor vamos a simular una animación muy similar a la ensayada en esta práctica. Cargar el modelo de Thor y su textura. Este modelo tiene una altura de unos 1.8m y está diseñado con el eje Y hacia arriba y con el eje Z en la dirección en la que tiene extendidos los brazos.



Elegid la posición del observador / target / campo de visión / vector up de forma que Thor se vea mirando de frente hacia nosotros y se vea una figura como la adjunta (cuidad de que los brazos extendidos entren en la imagen).

Indicad los parámetros usados (obs, target, campo visión, etc) y adjuntad una captura de pantalla.

Como matriz M del movimiento aplicad a todo el modelo un giro alrededor del eje Y de 30º por segundo (usando rotate) como se hizo en clase. El modelo debe verse girar como si estuviese en una plataforma giratoria.

Lo que queremos hacer en este ejercicio es que la figura (además de girar en bloque) doble uno de sus brazos. Aplicaremos al antebrazo un giro adicional alrededor del eje Y de acuerdo a la expresión: $45 \cdot (1-\cos(3 \cdot tt))$ que hace que el antebrazo gire alrededor del codo entre la posición original (0º) y la correspondiente a 90º (apuntando hacia delante).

Lo normal es que el animador nos hubiera pasado a través de atributos, la asignación de los vértices a las distintas partes del cuerpo. Como no tenemos esa información, haremos lo que en la primera parte de la tarea: usaremos la coordenada Z del modelo para decidir la relación de un vértice con la extremidad. El codo del brazo izquierdo corresponde aproximadamente a las coordenadas X=0, Y=1.45, Z=0.48.

Usando el ejercicio anterior como ejemplo para determinar la matriz M2 a usar para el antebrazo a partir de la matriz M del resto del cuerpo y de su giro propio R. Los vértices pertenecientes al antebrazo serán aquellos vértices del modelo cuya coordenada Z sea >= 0.48 (la coordenada Z del codo).
Mostrar una captura donde se aprecie el giro de la articulación.

(15%) Modificar el programa para ampliar el movimiento al otro brazo, haciendo que ambos se muevan con un giro similar. Observaréis que incluso con sólo 2 articulaciones el enfoque de usar la información de la coordenada Z para decidir cómo actuar se va complicando. Esta es la razón de que, en la práctica, la asignación de los vértices a las diferentes articulaciones se haga a través de atributos. Adjuntad una nueva captura.

---- Adjuntar código de vuestro programa como thor.cpp -----