Programación de sistemas de animación
Práctica

La práctica de programación de sistemas de animación está dividida en dos partes, cada una orientada a implementar en el motor soporte para un tipo de animación.

Primera parte (0,5 puntos)

El objetivo de esta parte consiste en introducir en la clase `Sprite` del motor soporte para animación basada en frames.

La clase `Image` ya viene con el soporte para definir en el momento de su carga el número de frames que tiene una imagen, y el método `DrawImage` de la clase `Renderer` ya hace el cálculo para extraer las coordenadas UV por nosotros y dibujar el frame apropiado.

Lo que queremos añadir es la posibilidad de que, indicando a un sprite su frame de animación mínimo y máximo, y una velocidad especificada en frames por segundo (fps), en el método `Update` se vaya actualizando la información del frame de animación a pintar basado en el tiempo transcurrido desde que seleccionó el último frame.

Los métodos a implementar son `SetFPS`, `GetFPS`, `SetFrameRange`, `GetFirstFrame`, `GetLastFrame`, `SetCurrentFrame`, `GetCurrentFrame`.

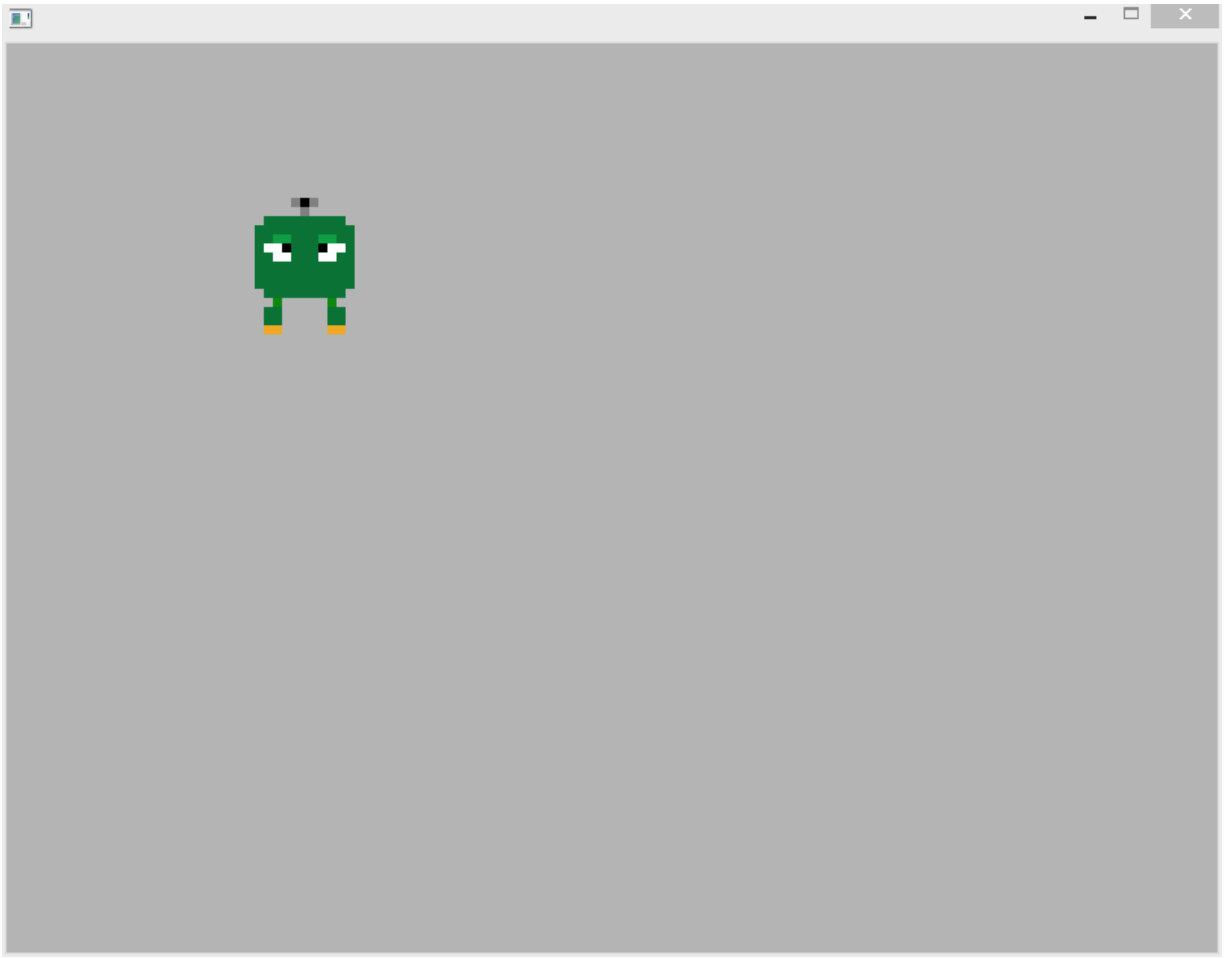
Y, como ya se ha dicho, debemos asegurarnos de que en el método `update` se va actualizando el frame actual a la velocidad establecida por `SetFPS`. También debe estar implementado el método `Render` de forma que se pinte el frame actual.

Partiremos del resultado de la práctica avanzada de sprites, en el que dibujábamos un pequeño alien en la pantalla que podía ser movido con los cursores. Hay que modificar la práctica (o, en caso de no haber desarrollado la parte avanzada de la práctica de sprites, implementarla desde cero), de forma que ahora la imagen cargada sea **“data/alienanim.png”**, con 8 frames de animación en horizontal y uno en vertical.

Se debe indicar una velocidad de 16 frames por segundo para la animación.

Si se había implementado la parte avanzada de la práctica de sprites, se debe mantener la rotación del alien al girar a izquierda y derecha. Si no se había realizado esta parte, no es necesario incluir la rotación en esta práctica.

Un ejemplo de resolución de la práctica es el siguiente:



Segunda parte (0,5 puntos)

En esta segunda parte, implementaremos un sistema básico de animación basado en huesos. Implementaremos una clase `SkeletonSprite` que contendrá una jerarquía de huesos, cada uno con una secuencia de frames que representarán los keyframes de la animación. Todos los frames restantes se deben de calcular con los valores del keyframe anterior y el siguiente.
La interfaz de la clase `SkeletonSprite` es la siguiente (se proporciona en los recursos de la práctica junto con una implementación parcial):

```
class SkeletonSprite : public Sprite {
public:
```

```
	SkeletonSprite(const String& filename);
	virtual ~SkeletonSprite();

	virtual const Bone* GetRootBone() const;

	virtual void Update(double elapsed, Map* map = NULL);
	virtual void Render() const;
protected:
	virtual void UpdateCollisionBox();
private:
	Bone* root;
};
```

El sprite contiene un puntero al hueso raíz y un método `GetRootBone` para obtenerlo. Además, se redefinen dos métodos heredados de la clase padre `Sprite`.

En el constructor de la clase, se utiliza la clase auxiliar `SkeletonData` para cargar los datos del XML. Esta clase es de uso interno, y simplemente se ha añadido para evitar al alumno tener que trabajar sobre ficheros XML en esta práctica. La clase simplemente contiene un vector de objetos `BoneData` con los huesos leídos del fichero y los métodos para acceder a estos huesos de la lista.

Respecto a esta clase `BoneData`, contiene todos los datos de un hueso contenidos en el XML, y un vector con todos los frames de ese hueso. La clase `SkeletonSprite` debe crear una jerarquía de objetos de tipo `Bone` a partir de los objetos `BoneData` (que no están organizados de forma jerárquica, sino en una lista, tal y como aparecen en el fichero de datos).

Parte de la funcionalidad de la clase `SkeletonSprite` viene ya implementada, pero el alumno deberá realizar una serie de tareas:

En el constructor, se crea el objeto `SkeletonData` a partir del fichero para tener acceso a todos los datos de la animación, y a continuación se crea el hueso raíz (es la raíz del árbol jerárquico de los huesos). Después entramos en un bucle `for` que itera por cada hueso cargado en el fichero, y debemos de ir generando un objeto `Bone` por cada `BoneData` iterado y añadiéndolo correctamente a la jerarquía de huesos del sprite:

- Obtenemos el hueso con el valor del índice iterado actual.
- Encontramos el objeto hueso padre del que estamos leyendo. El nombre del hueso padre lo obtenemos con el método `GetParentName` de la clase `BoneData`. Si el nombre es **“world”**, entonces el nuevo hueso se añadirá directamente como hijo del hueso raíz. En cualquier otro caso, debemos utilizar el método `FindChild` sobre el hueso raíz para encontrar el hueso al que añadiremos el nuevo.
- Obtenemos el nombre de la imagen del `BoneData`, le quitamos la ruta con la función `ExtractDir` de la clase `String`, y la cargamos.

- Creamos un nuevo objeto `Bone` con la imagen cargada, el hueso padre obtenido anteriormente, y pasamos en el constructor los datos del objeto `BoneData`.
- A continuación iteramos por todos los frames del `BoneData` y los añadimos al `Bone`.
- Por último, utilizamos el método `AddChild` sobre el hueso padre para añadir el hueso que acabamos de crear.

Al final del constructor se busca el último frame de animación para establecer la longitud de la secuencia, y se borra el objeto `SkeletonData`, ya sólo es necesario para la generación del esqueleto.

En el método `Update`, debemos llamar al método padre, y a continuación al método `Update` del hueso raíz pasándole el frame actual del sprite.

En el método `Render`, estableceremos el blending mode y el color de pintado del sprite, y realizaremos las siguientes acciones:

- Añadimos la matriz de OpenGL activa a la pila de matrices con `glPushMatrix`.
- Trasladamos la matriz activa a las coordenadas del sprite con `glTranslated`.
- Llamamos al método `render` del hueso raíz, que se dibujará con la transformación definida.
- Restauramos la matriz actual con la matriz que hemos añadido a la pila utilizando la función `glPopMatrix`.

Los huesos pertenecen a la clase `Bone`, que también se entrega parcialmente implementada, y que cuenta con la siguiente interfaz:

```
class Bone {
public:
	Bone();
	Bone(const String& id, Image* image, double pivotX, double pivotY, double handleX, double handleY);

	virtual const String& GetID() const;
	virtual const Image* GetImage() const;

	virtual void AddChild(const Bone& bone);
	virtual uint32 CountChildren() const;
	virtual const Bone* GetChild(uint32 index) const;
	virtual Bone* GetChild(uint32 index);
	virtual const Bone* FindChild(const String& id) const;
	virtual Bone* FindChild(const String& id);

	virtual void AddFrame(const Frame& frame);
	virtual uint32 CountFrames() const;
```

```
	virtual const Frame* GetFrame(uint32 index) const;
	virtual const Frame* FindFrame(uint32 id) const;

	virtual void TranslationForFrame(int32 f, double* x, double* y) const;
	virtual double RotationForFrame(int32 f) const;
	virtual void ScaleForFrame(int32 f, double* x, double* y) const;

	virtual void Update(int32 currentFrame);
	virtual void Render();
protected:
	virtual void GetFrame(int32 f, const Frame** frame, const Frame**
prevFrame, const Frame** nextFrame) const;
	virtual double Interpolate(int32 id, int32 prevId, int32 nextId, double
prevVal, double nextVal) const;
private:
	String id;
	Image* image;
	double pivotX, pivotY;
	double handleX, handleY;
	Array<Bone> children;
	Array<Frame> frames;

	double currentX, currentY;
	double currentRotation;
	double currentScaleX, currentScaleY;
};
```

En esta clase, implementaremos el método `Interpolate`, que es utilizado por `TranslationForFrame`, `RotationForFrame` y `ScaleForFrame` para interpolar entre dos keyframes cuando el frame actual no está en la lista. También implementaremos los métodos `Update` y `Render`.

En `Interpolate`, recibimos un identificador con la posición que buscamos dentro del rango, otros dos identificadores que representan las posiciones mínima y máxima dentro del rango, y los valores en la posición mínima y máxima. Por ejemplo, si el rango empieza en la posición 0 con un valor de 0.0 y termina en la posición 10 con un valor de 5.0, y usamos `interpolate` para obtener el valor de la posición 5, éste será 2.5. Este funcionamiento se representa con el siguiente pseudocódigo:

```
	valor = prevVal + (nextVal - prevVal) * (id - prevId) / (nextId - prevId)
```

Hay que tener en cuenta que se deben realizar las conversiones necesarias para garantizar que la división que aparece se haga con valores decimales y no enteros.

En el método `update`, debemos actualizar los valores de las variables miembro `currentX`, `currentY`, `currentRotation`, `currentScaleX` y `currentScaleY` utilizando los métodos `TranslationForFrame`, `RotationForFrame` y `ScaleForFrame`. A continuación, iteramos por todos los hijos del hueso y llamamos a su método `Update` para que puedan también actualizarse.

En el método `Render`, haremos las siguientes tareas:

- Apilamos la transformación de la matriz actual con `glPushMatrix`.
- Trasladamos la matriz a la posición `currentX` y `currentY` con `glTranslated`.
- Rotamos la matriz sobre el eje Z con `glRotated(currentRotation, 0, 0, -1)`.
- Si el hueso actual tiene imagen (es decir, si su variable miembro `image` tiene un valor distinto de `NULL`) hacemos lo siguiente:
    - Modificamos su handle con los valores `handleX` y `handleY` del hueso, que están normalizados, con lo que hay que multiplicarlos por el ancho y el alto de la imagen.
    - Dibujamos la imagen en la posición 0, 0, con el frame 0, y como escala utilizamos `currentScaleX` y `currentScaleY` multiplicados por el ancho y el alto de la imagen.
    - Colocamos el pivote de referencia para los huesos hijos utilizando `glTranslated`, utilizando los valores normalizados `pivotX` y `pivotY` multiplicados por el ancho y el alto de la imagen.
- Dibujamos los huesos hijos (independientemente de si el actual tenía imagen o no) iterando por todos ellos y llamando a su método `render`.
- Restauramos las transformaciones de la matriz recuperando la que habíamos apilado con `glPopMatrix`.

La clase `Frame`, que contiene las transformaciones para un hueso en sus keyframes la tenemos completamente implementada ya, y su interfaz es la siguiente:

```
class Frame {
public:
	Frame();
	Frame(int32 id, double translationx, double translationy, double
rotation, double scalex, double scaley);
	virtual int32 GetId() const;
	virtual double GetTranslationX() const;
	virtual double GetTranslationY() const;
	virtual double GetRotation() const;
	virtual double GetScaleX() const;
	virtual double GetScaleY() const;
private:
	int32 id;
	double translationx, translationy;
```

```
        double rotation;
        double scalex, scaley;
};
```

Una vez añadida la funcionalidad de animación por huesos al motor, comprobaremos que está correctamente implementada creando un `SkeletonSprite` a partir del fichero **“data/ animation.xml”**, estableciendo su velocidad de animación a 32 frames por segundo, y dibujando el sprite en las coordenadas del ratón, lo cual nos dará el siguiente resultado:

La animación muestra una cadena de huesos que se comporta como un látigo que se agita hacia la derecha y vuelve al origen.