Fundamentos de programación gráfica
Práctica 7 - Renderizado de texto

Aprovechando el soporte para *texture atlas* incluido en el motor gráfico, vamos a incluir soporte para el renderizado de textos. La práctica se divide en dos partes.

Primera parte (0.3 puntos)

La imagen **"data/monospaced.png"** es un texture atlas de 16x16 frames. Cada frame contiene la representación en una fuente monoespaciada del carácter ASCII correspondiente (de 0 a 255). Los caracteres están dibujados con color blanco para que se pueda tintar el texto con `Renderer::SetColor`. El fondo de la imagen es transparente.

Para manejar el renderizado de textos, crearemos una nueva clase `Font` en el motor. Como su funcionalidad básica es análoga a la de una imagen con 256 frames, vamos a hacer que `Font` sea una clase derivada de `Image`, y aprovecharemos de esta forma la funcionalidad ya implementada. La cabecera de la clase debe ser como sigue:

```
class Font : public Image {
public:
	Font(const String& filename);

	virtual uint16 GetSize() const;
	virtual uint32 GetTextWidth(const String& text) const;
	virtual uint32 GetTextHeight(const String& text) const;

	virtual void Render(const String& text, double x, double y);
};
```

El constructor únicamente recibe el nombre de la imagen que contiene el texture atlas con los caracteres. Al llamar al constructor base, se deben especificar los 16x16 frames que contiene la imagen.

El método `GetSize()` devuelve el tamaño de la fuente. Es lo mismo que llamar a `GetHeight()` (método heredado de la clase `Image`). En la fuente utilizada como ejemplo, el alto es de 16 puntos.

Los métodos `GetTextWidth()` y `GetTextHeight()` devolverán el tamaño en píxeles que ocupa un determinado texto. El ancho es igual al tamaño de la fuente por el número de caracteres, y el alto es igual al alto de la fuente.

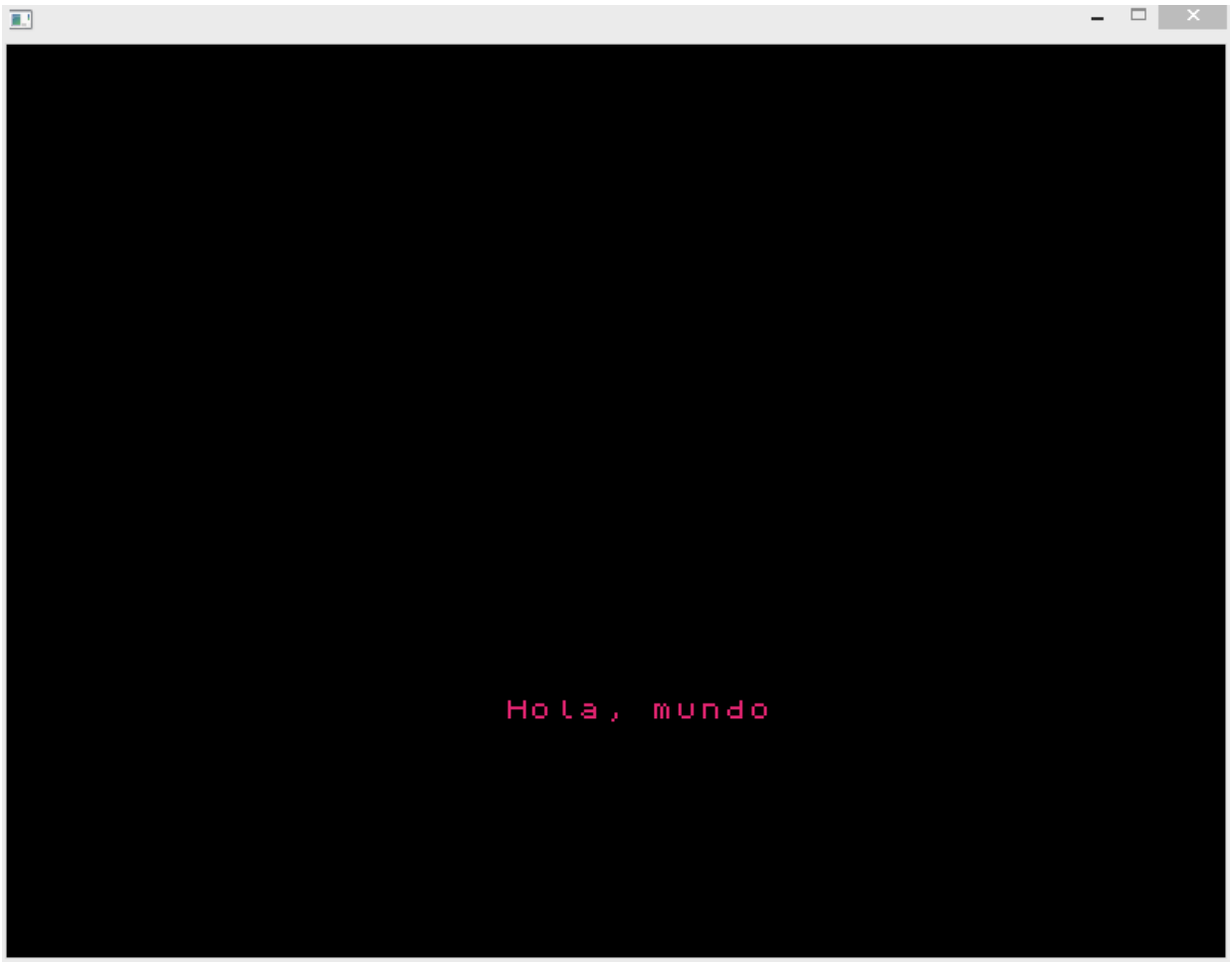
El método `Render` debe dibujar todos los frames de la imagen correspondientes a los caracteres del texto pasado como parámetro. Utilizará el método `DrawImage` de la clase `Renderer` para efectuar el pintado.

En la clase `ResourceManager`, activaremos los métodos correspondientes a la carga y limpiado de fuentes. Debemos asegurarnos de que el método `FreeResources` libera también las fuentes cargadas.

Una vez creada la interfaz e implementación de la nueva clase (e incluida su cabecera en los módulos del motor que corresponda), realizaremos una aplicación para probar su funcionamiento. Las características de la aplicación deben ser las siguientes:

- La aplicación debe dibujar el texto **“Hola, mundo”** en pantalla.
- En el arranque, obtendremos una velocidad de desplazamiento en X y en Y al azar dentro del rango [128, 255]. El texto debe moverse a esta velocidad por la pantalla.
- Cuando el texto choca con los límites izquierdo o derecho de la pantalla, la velocidad de desplazamiento en X se debe invertir.
- Cuando el texto choca con los límites inferior o superior de la pantalla, la velocidad de desplazamiento en Y se debe invertir.
- Cuando el texto choca con cualquier de los límites, se debe escoger un nuevo color para el texto al azar.

La aplicación resultante debe mostrarse de la siguiente manera:



Segunda parte (0.7 puntos)

Para poder utilizar tipografías de ancho variable, vamos a implementar detección de glifos dentro de cada frame del *texture atlas*. Introduciremos información sobre el tamaño del glifo dentro de la propia imagen, de la siguiente manera:

Dentro de cada frame, un píxel de color amarillo **(RGB=255,255,0)** marca las coordenadas de origen del glifo. Un píxel de color rojo **(RGB=255,0,0)** marca el final del glifo. Debemos crear una clase `Glyph` que almacene estas coordenadas, y la clase `Font` contendrá una lista de glifos con las coordenadas de cada carácter dentro del frame.

Para leer el valor de los píxeles, dentro del constructor de la clase `Font` podemos cargar el buffer de imagen con `stbi_load_image` (al igual que se hace en la clase `Image`) y buscar los píxeles adecuados dentro de cada frame. Además, hay que modificar los píxeles del buffer de la siguiente manera:

- Los píxeles amarillos y rojos deben cambiar su valor de alpha a cero (de forma que no son visibles al imprimir texto).

- Los píxeles negros (**RGB=0,0,0**) deben ser convertidos en transparentes, de forma que podamos cargar fuentes sin transparencia (que se ven con mayor claridad si abrimos con un visor de imágenes la fuente), y el motor tomará estos píxeles como transparentes.

Una vez modificado el buffer, utilizaremos de nuevo la función `glTexImage2D` para sustituir la textura de OpenGL generada por la clase `Image` por una nueva generada a partir de este buffer.

Los métodos `GetTextWidth` y `GetTextHeight` de clase clase `Font` también deben ser modificados para que tengan en cuenta el tamaño de cada glifo.

A la hora de dibujar los caracteres, hay que espaciarlos y posicionarlos correctamente en función de las dimensiones de cada glifo.

Se debe modificar el ejercicio de la parte anterior para que cargue la fuente **"data/ arial16.png"**, de anchura variable, y renderice el texto adecuadamente. El resultado debe quedar como sigue: