Un videojuego sencillo con Gosu y Ruby

Original por Julian Raschke
Traducción de Belén Albeza

22 de marzo de 2008

Resumen

Esta es una traducción del tutorial de iniciación a Gosu disponible en el wiki oficial[1]. Muestra el funcionamiento básico de la librería Gosu mediante la creación de un pequeño juego de ejemplo.

Preparación

El **código fuente** del juego completo, junto con los archivos de imágenes y sonidos que hacen falta, se puede encontrar en la distribución de Gosu que hayas descargado (`examples/Tutorial.rb`).

Si has instalado Gosu vía RubyGems, los ejemplos están en tu carpeta `gems` con el resto de la librería. Por ejemplo, en OS X 10.5, los ejemplos se encuentran en `/Library/Ruby/Gems/1.8/gems/gosu-<version>/examples`.

Si no has instalado Gosu vía RubyGems, tienes que copiar `gosu.so` (o `gosu.bundle`) en el directorio de los ejemplos, y a continuación ejecutar `Tutorial.rb`.

Si no dispones de un editor que soporte ejecución directa (TextMate, SciTE…), sitúate desde consola en el directorio y ejecuta el tutorial con `ruby Tutorial.rb`.

1. Sobrecargando la clase Window

La manera más fácil de crear una aplicación en Gosu es crear una nueva clase que **herede** de `Gosu::Window` (consulta la referencia para una descripción completa de su interfaz). Una aplicación básica con nuestra propia clase `GameWindow` podría ser esta:

[1] http://code.google.com/p/gosu/wiki/RubyTutorial
[2] http://www.opensource.org/licenses/mit-license.php

```
require 'gosu'

class GameWindow < Gosu::Window
  def initialize
    super(640, 480, false)
    self.caption = "Tutorial de Gosu"
  end

  def update
  end

  def draw
  end
end

window = GameWindow.new
window.show
```

El **constructor** llama al de la clase base `Gosu::Window`. Los parámetros indicados crean una **ventana** (de ahí el `false`, para desactivar la pantalla completa) de 640×480 píxeles. Después cambia el **título** de la ventana, que estaba vacío hasta entonces.

Los métodos `update` y `draw` son sobrecargas de métodos propios de la clase base `Gosu::Window`. El método **`update`** se llama 60 veces por segundo (por defecto) y es el que contiene la lógica principal del juego: movimiento de objetos, manejo de colisiones, etc.

El método **`draw`** se llama después de `update` y también cuando la ventana requiera re-dibujarse por cualquier motivo. También puede ser que se omita una llamada a `draw` si los FPS (frames por segundo) caen muy bajo. Este método debería contener el código para redibujar la pantalla entera, dejando a un lado toda la parte lógica.

A continuación aparece el **programa principal**. Se crea una ventana y se llama a su método `show`, que no efectúa su `return` hasta que se indique desde código o el usuario cierre la ventana. *Et voilà!* Ahora tienes una ventanita negra con un título a tu elección.

2. Usando imágenes

2.1. El fondo

```
class GameWindow < Gosu::Window
  def initialize
    super(640, 480, false)
    self.caption = "Tutorial de Gosu"
```

```
    @background_image = Gosu::Image.new(self, "media/Space.png"
                                        true)
  end

  def update
  end

  def draw
    @background_image.draw(0, 0, 0);
  end
end
```

El constructor de la clase `Gosu::Image` tiene tres argumentos. El primero, como todos los recursos multimedia, está asociado a una **ventana** en concreto (`self` en este caso). Todos los recursos de Gosu necesitan un objeto de clase `Window` para su inicialización, y mantendrán una referencia a esa ventana. El segundo parámetro es el nombre del **fichero** con la imagen. El tercer parámetro especifica si la imagen se crea o no con **bordes duros** (ver la sección de *"Conceptos Básicos"* en el wiki[3]).

Como hemos dicho anteriormente, el método `draw` de `Window` (o sus clases derivadas) es el lugar para dibujar todo, así que dibujamos ahí nuestra imagen de fondo. Los argumentos son obvios: la imagen se dibuja en las coordenadas `(0,0)`; el tercer parámetro marca la coordenada `Z` (ver la sección de *"Conceptos Básicos"*).

2.2. El jugador y sus movimientos

Aquí está una clase sencilla para el jugador:

```
class Player
  def initialize(window)
    @image = Gosu::Image.new(window, "media/Starfighter.bmp", false)
    @x = @y = @vel_x = @vel_y = @angle = 0.0
  end

  def warp(x, y)
    @x, @y = x, y
  end

  def turn_left
    @angle -= 4.5
  end

  def turn_right
    @angle += 4.5
  end
```

[3] http://code.google.com/p/gosu/wiki/BasicConcepts

```
  def accelerate
    @vel_x += Gosu::offset_x(@angle, 0.5)
    @vel_y += Gosu::offset_y(@angle, 0.5)
  end

  def move
    @x += @vel_x
    @y += @vel_y
    @x %= 640
    @y %= 480

    @vel_x *= 0.95
    @vel_y *= 0.95
  end

  def draw
    @image.draw_rot(@x, @y, 1, @angle)
  end
end
```

Hay un par de cosas a comentar sobre los movimientos:

- **`accelerate`** hace uso de las funciones `offset_x` y `offset_y`, que funcionan de forma similar a como algunos emplean los senos y cosenos: por ejemplo, si algo se mueve 100 píxeles en un ángulo de 30º, se movería `offset_x(30,100)` píxeles horizontalmente y `offset_y(30,100)` `píxeles verticalmente`. Esto se muestra en la imagen de la figura 1.
- Cuando se cargan **ficheros BMP**, Gosu reemplaza el color `0xFF00FF` (el rosa fosforito feo) con píxeles transparentes[4].
- Fíjate en que **`draw_rot`** sitúa el centro de la imagen en las coordenadas `(x,y)`, y no la esquina superior izquierda (como hace `draw`).
- El jugador se dibuja con la coordenada `z=1`; es decir, por encima del fondo. Reemplazaremos estos numeritos mágicos por algo mejor más adelante.
- Lee la *"Referencia para Ruby"*[5] para ver todos los métodos de dibujo y sus argumentos.

2.3. Integrando al jugador

[4]También se pueden cargar imágenes PNG con canal alpha
[5]http://code.google.com/p/gosu/wiki/RubyReference

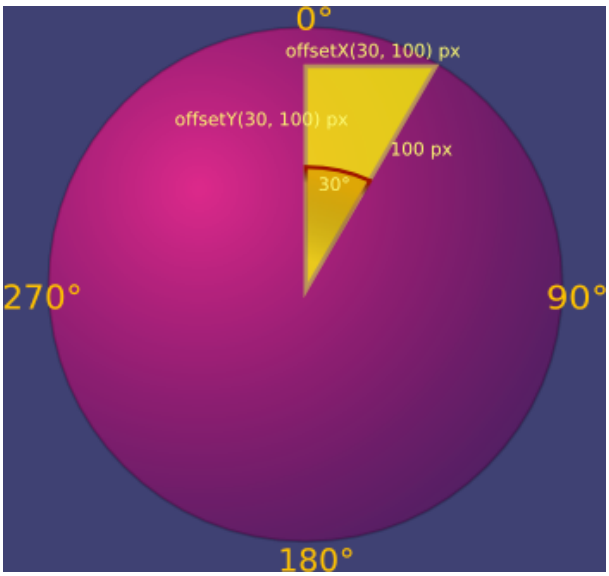

Figura 1: offset_x y offset_y

```
class GameWindow < Gosu::Window
  def initialize
    super(640, 480, false)
    self.caption = "Tutorial de Gosu"

    @background_image = Gosu::Image.new(self, "media/Space.png",
                                        true)

    @player = Player.new(self)
    @player.warp(320, 240)
  end

  def update
    if button_down? Gosu::Button::KbLeft or
       button_down? Gosu::Button::GpLeft then
      @player.turn_left
    end
    if button_down? Gosu::Button::KbRight or
       button_down? Gosu::Button::GpRight then
      @player.turn_right
    end
    if button_down? Gosu::Button::KbUp or
    button_down? Gosu::Button::GpButton0 then
      @player.accelerate
    end
    @player.move
  end
```

```
  def draw
    @player.draw
    @background_image.draw(0, 0, 0);
  end

  def button_down(id)
    if id == Gosu::Button::KbEscape
      close
    end
  end
end
```

Como puedes ver, ¡hemos introducido entrada por `teclado` y por `mando`! De forma similar a `update` y `draw`, la clase `Gosu::Window` nos proporciona dos metodos que pueden ser sobrecargados y que por defecto están vacíos: `button_down(id)` y `button_up(id)`. Aquí empleamos uno para **cerrar la ventana** cuando el usuario presiona la tecla Escape (para obtener una lista con todas las constantes de botones predefinidas, consulta la *"Referencia para Ruby"*).

Auqnue conocer cuándo se pulsan ciertos botones es adecuado para eventos que sólo se ejecutan una vez, como las interacciones con la interfaz, saltar o teclear, es bastante inútil para acciones que se expanden durante varios frames (por ejemplo, moverse manteniendo pulsado un botón). Y es aquí cuando el método `update` entra en juego, el cual simplemente llama a los métodos de **movimiento** de la clase del jugador.

Si ejecutas el código de este apartado, deberías poder volar con la nave.

3. Animaciones simples

En primer lugar, vamos a librarnos de los numeritos mágicos para las coordenadas Z reemplazándolos por las siguientes constantes:

```
module ZOrder
  Background, Stars, Player, UI = *0..3
end
```

¿Qué es una **animación**? Pues una secuencia de imágenes, así que usaremos los arrays de Ruby para almacenarlas. En un juego real, a veces no hay más remedio que escribir nuestras propias clases para satisfacer nuestras necesidades específicas, pero por ahora emplearemos esta solución sencilla.

Vamos a presentar a las **estrellas**, que son las protagonistas de esta sección. Las estrellas aparecen de la nada en una posición aleatoria de la pantalla y viven una vida animada hasta que el jugador las recolecta. La clase de las estrellas, `Star` es bastante sencilla:

```
class Star
  attr_reader :x, :y

  def initialize(animation)
    @animation = animation
    @color = Gosu::Color.new(0xff000000)
    @color.red = rand(255 - 40) + 40
    @color.green = rand(255 - 40) + 40
    @color.blue = rand(255 - 40) + 40
    @x = rand * 640
    @y = rand * 480
  end

  def draw
    img = @animation[Gosu::milliseconds / 100 % @animation.size];
    img.draw(@x - img.width / 2.0, @y - img.height / 2.0,
        ZOrder::Stars, 1, 1, @color, :additive)
  end
end
```

Puesto que no queremos que cada estrella cargue de nuevo la animación, no podemos hacer eso en su constructor. En su lugar, hemos de pasársela desde otro sitio (la ventana cargará la animación dentro de unos cuantos párrafos).

Para mostrar un *frame* diferente de la **animación** cada 100 milisegundos, el tiempo que devuelve `Gosu::milliseconds` se divide entre 100 y se le efectúa un módulo para mantenerlo dentro del rango del número de imágenes que componen la animación. Esta imagen se dibuja de forma aditiva, centrada en la posición de la estrella, y con un color aleatorio generado en el constructor.

Ahora vamos a añadir un código simple para hacer que el jugador **recolecte** las estrellas (que se encuentran almacenadas en un array).

```
class Player
  # [...]
  def collect_stars(stars)
    stars.reject! do |star|
      Gosu::distance(@x, @y, star.x, star.y) < 35
    end
  end
end
```

Y a continuación ampliamos la clase `GameWindow` para cargar la animación, crear las estrellas, hacer que el jugador las recolecte y dibujar las que aún estén en pantalla:

```
class Window < Gosu::Window
  def initialize
```

```
    super(640, 480, false)
    self.caption = "Tutorial de Gosu"

    @background_image = Gosu::Image.new(self, "media/Space.png",
                                        true)

    @player = Player.new(self)
    @player.warp(320, 240)

    @star_anim = Gosu::Image::load_tiles(self, "media/Star.png",
                                         25, 25, false)
    @stars = Array.new
  end

  def update
    # ...
    @player.move
    @player.collect_stars(@stars)

    if rand(100) < 4 and @stars.size < 25 then
      @stars.push(Star.new(@star_anim))
    end
  end

  def draw
    @background_image.draw(0, 0, ZOrder::Background)
    @player.draw
    @stars.each { |star| star.draw }
  end

  # ...
end
```

¡Hecho! Ahora puedes recoger todas esas estrellas.

4. Textos y sonidos

Por último, queremos dibujar la **puntuación** del jugador usando una **fuente** bitmap y reproducir un **sonido** de *beep* cada vez que el jugador recoja una estrella. La ventana manejará la parte del texto, cargando una fuente de 20 píxeles de alto:

```
class GameWindow < Gosu::Window
  def initialize
    # ...
    @font = Gosu::Font.new(self, Gosu::default_font_name, 20)
  end

  # ...
```

```
  def draw
    @background_image.draw(0, 0, ZOrder::Background)
    @player.draw
    @stars.each { |star| star.draw }
    @font.draw("Score: #{@player.score}", 10, 10, ZOrder::UI, 1.0,
               1.0, 0xffffff00)
  end

  # ...
end
```

¿Y qué queda para el jugador? Exacto: un contador para la puntuación, cargar el sonido y reproducirlo:

```
class Player
  attr_reader :score

  def initialize(window)
    @image = Gosu::Image.new(window, "media/Starfighter.bmp", false)
    @beep = Gosu::Sample.new(window, "media/Beep.wav")
    @x = @y = @vel_x = @vel_y = @angle = 0.0
    @score = 0
  end

  # ...

  def collect_stars(stars)
    stars.reject! do |star|
      if Gosu::distance(@x, @y, star.x, star.y) < 35 then
        @score += 10
        @beep.play
        true
      else
        false
      end
    end
  end
end
```

Como ves, cargar y reproducir efectos de sonido no podría ser más fácil. Consulta la *"Referencia para Ruby"* para conocer otras formas más potentes de reproducir sonidos (trastear con el volumen, *panning* y el tono).

¡Y esto es todo! Lo demás queda a tu imaginación. Si consideras que este tutorial no es suficiente para crear juegos, mira la sección de *"Usuarios de Gosu"*[6] en el wiki oficial para descargar código fuente y echarle un vistazo.

[6] http://code.google.com/p/gosu/wiki/GosuUsers