# Olvídate de la SDL: Desarrollo de juegos en C++ con Gosu

José Tomás Tocino García - <theom3ga@gmail.com>

Licencia CC - Reconocimiento - No comercial - Compartir igual.
http://creativecommons.org/licenses/by-nc-sa/2.5/es/

¿Cuántas líneas hacen falta para hacer aparecer una imagen en pantalla usando la biblioteca SDL? ¿Cuántas veces has implementado una clase `Imagen` para usar orientación a objetos con la SDL? ¿Has visto algún juego hecho con SDL ir más allá de los 70fps?

Si has usado SDL para programar juegos, conocerás las respuestas a estas preguntas, y habrás sufrido una agonía de bajo nivel repleta de `SDL_Surface`, `SDL_DisplayFormatAlpha` y demás engendros del demonio. Una vez que tenías el motor del juego más o menos hecho, con suerte te quedaban fuerzas para hacer a lo sumo un par de niveles del juego.

Pero esto, señores, **s'acabao**.


Gosu (`http://www.libgosu.org`) es una librería para el desarrollo de videojuegos 2D utilizando Ruby y C++, y disponible para Mac OS X, Windows y, oh sí, **Linux**. Incluye la gestión de gráficos (con **aceleración 3D por OpenGL**), gestión de la entrada y reproducción de música y sonidos, así como funciones de red.

La librería está hecha en **C++**, por lo que la orientación a objetos es total. Además, está completamente adaptada a la **creación de juegos**, adaptando el flujo de ejecución al game loop típico consistente en gestionar la entrada de usuario, actualizar la lógica del juego y finalmente imprimir los gráficos en pantalla.

## 1. Requisitos previos

Es necesario que sepas C++. Es fácil, es bonito, mola. Si no sabes C++, corre a la biblioteca.

Al ser una librería poco conocida (al menos por ahora, pero esperad a que este tutorial se haga famoso, MWAHAHA), no está disponible en los repositorios de las distribuciones más famosas, ni tampoco está empaquetada, así que hay que instalarla partiendo del código fuente.

Afortunadamente es un proceso muy sencillo que no reviste dificultad alguna. Antes de nada, hay que instalar las dependencias de la librería, que son unas cuantas. Suponiendo que estás en un sistema basado en Debian:

**Código 1** Instalación de dependencias

```
sudo apt-get install g++ libgl1-mesa-dev libpango1.0-dev libboost1.40-dev \
     libsdl-mixer1.2-dev libsdl-ttf2.0-dev
```

Una vez hecho, crea una carpeta para tu proyecto, por ejemplo `miJuego`, descarga el código fuente de Gosu desde la página de descargas (`http://code.google.com/p/gosu/downloads/list`) y descomprímelo. Debería aparecer una carpeta `gosu` dentro de la carpeta de tu proyecto.

Entra en `gosu` y luego en el subdirectorio `linux`, y teclea:

Ya está compilada y lista para usarse. Gosu suele adjuntarse con cada proyecto, en lugar de instalarla a nivel de sistema, y se enlaza estáticamente. Aunque de eso hablaremos más adelante.

**Código 2** Compilación de Gosu

```
./configure
make
```

### 1.1. Conociendo boost

A lo largo del tutorial, y en todo el código de la propia librería, se hace uso de una parte de las librerías **Boost**. Para los que no lo conozcan, Boost es un conjunto de librerías open source que amplían las posibilidades del lenguaje C++. Después de la STL, se trata del conjunto de librerías **más importante** que existe para este lenguaje; tanto es así que la próxima versión de C++ (C++0x) incluirá por defecto partes de Boost de forma estándar.

Las partes de Boost que se utilizarán son los **punteros inteligentes** (smart pointers). Estos punteros se diferencian principalmente de los punteros tradicionales en que simplifican la gestión de la memoria, mediante tareas como la **liberación automática** de la misma al finalizar su uso. En nuestro caso usaremos concretamente dos tipos:

- **Scoped pointers:** Un scoped pointer (`boost::scoped_ptr`) es un puntero que no puede copiarse, es decir, el objeto al que apunta no puede ser referenciado por ningún otro puntero, asegurándonos de que el único punto de acceso al mismo sea nuestro puntero original. Un scoped pointer pertenece a un ámbito (scope), y es liberado cuando el ámbito termina. Por ejemplo, si es un miembro de una clase, el puntero será liberado cuando se llame al destructor de la misma.
- **Shared pointers:** Un shared pointer (`boost::shared_ptr`) es parecido a un scoped pointer, se diferencia de él en que puede ser copiado, siendo éste su punto fuerte. El shared pointer llevará la cuenta de cuántos punteros están haciendo referencia al objeto creado, liberándose automáticamente cuando ya no quede ningún puntero.

  Esto resulta muy útil a la hora de compartir un recurso entre varios objetos o clases, ya que de esta forma podemos olvidarnos de liberar el recurso cuand ya ningún objeto esté usándolo.

Cuando aparezcan en el tutorial, explicaremos la sintaxis de estos punteros, aunque realmente no se diferencian en nada de los punteros tradicionales, ya que también hacen uso del operador flecha.

## 2. Empezando por el principio

Bien, parece que quieres hacer un juego. Lo primero es hacer que aparezca una ventana, porque no pretenderás que los muñequitos vayan por la pantalla sin orden ni concierto.

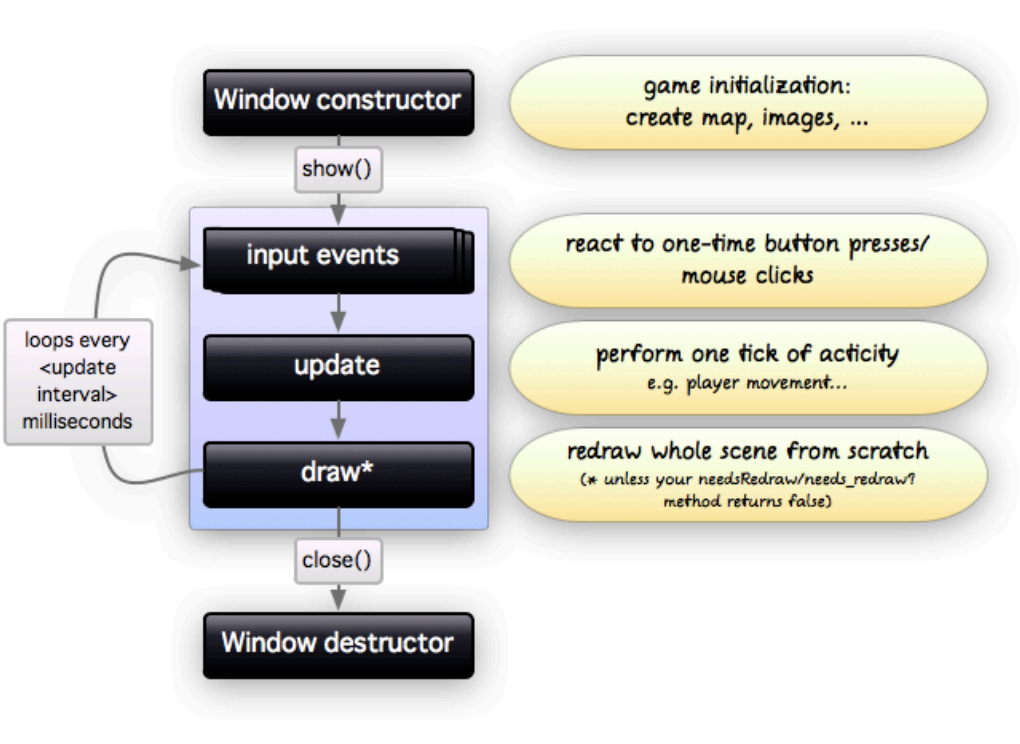
El punto de partida para todas las aplicaciones Gosu es la clase `Gosu::Window`. Se trata de una clase abstracta de la que debemos heredar para empezar a hacer nuestro juego. Mucha palabrería pero en realidad, nada del otro mundo. Abre un nuevo fichero `main.cpp` y guárdalo en la carpeta principal de tu proyecto.

**Código 3** Estructura inicial

```
#include <iostream>
#include "Gosu/Gosu.hpp"

using namespace std;

class Juego : public Gosu::Window{

public:

    Juego() : Gosu::Window(640, 480, false){
        cout << "Constructor" << endl;
    }

    void update(){
        // Aquí va la lógica del juego
    }

    void draw(){
        // Dibujado
    }

    void buttonDown(Gosu::Button boton){
        if(boton == Gosu::kbEscape){
            close();
        }
    }
};

int main(int argc, char *argv[])
{
    Juego J;
    J.show();

    return 0;
}
```

Nuestra clase `Juego` hereda de `Gosu::Window`, que se inicializa con el ancho y alto de la ventana, un booleano que indica si queremos pantalla completa, y un cuarto parámetro opcional para controlar los fps. Esta es otra **ventaja** de Gosu sobre la SDL: automáticamente controla los fps del juego, así que no tenemos que andar creándonos nuestra propia función que cuente los `ticks` ni nada parecido.

Normalmente, tras crear la subclase, se instancia y se llama al método `show`, que hará comenzar el Game Loop. La secuencia de eventos viene bien reflejada en el siguiente gráfico:




Existen dos métodos que hay que implementar:

- En `update` hemos de escribir la lógica del juego: cálculos, colisiones, etcétera.
- En `draw` incluiremos el código para mostrar las imágenes en pantalla.

whateverljsahlfkjhsadlfk lkjsadhflsakjdhflksadhflksahdf whateverljsahlfkjhsadlfk lkjsadhflsakjdhflksadhflksahdf whateverljsahlfkjhsadlfk lkjsadhflsakjdhflksadhflksahdf whateverljsahlfkjhsadlfk lkjsadhflsakjdhflksadhflksahdf whateverljsahlfkjhsadlfk lkjsadhflsakjdhflksadhflksahdf whateverljsahlfkjhsadlfk lkjsadhflsakjdhflksadhflksahdf