Dpto. de Lenguajes y Sistemas Informáticos
Escuela Técnica Superior de Ingenierías Informática y Telecomunicación

Problemas en la codificación

Autor:
Domingo Martín

Curso 2016/17

1. Introducción

La codificación de un programa mediante un lenguaje necesita de conocimiento, habilidad y experiencia. Es relativamente fácil hacer programas que funcionen pero no debemos conformarnos con eso sino que debemos intentar escribir programas que sean eficientes, así como fáciles de leer y mantener.

Aunque la experiencia y el aprendizaje es una cuestión personal, se pueden indicar algunas normas generales que nos puedan ayudar a la hora de implementar un código mejor. La mayoría de estas reglas se pueden enunciar como preguntas a las que la respuesta nos indicará si estamos en el buen camino o no, y en cualquier caso, nuestras decisiones deben estar justifcadas y deben poder ser defendidas, en otro caso, la selección no ha sido correcta.

Por ejemplo, una de las posibilidades que veremos es la de generar nueva funcionalidad replicando código, operación que se puede ver beneficiada por la facilidad de copiar y pegar. Diría que la regla en este caso es que si la cantidad de código es pequeña, puede ser una solución valida frente al uso de una función, pero si el número de líneas de código sobrepasa una cierta cantidad (es dificil establecer un umbral pues probablemente es una preferencia personal), lo mejor es crear una función.

A continuación iremos comentado distintos casos, algunos de caracter general y otros relacionados con las prácticas de la asignatura Informática Gráfica.

2. Ejemplos

La primera práctica de IG, en la que se propone que se dibujen un tetraedro y un cubo, pero que sobre todo se distinga entre lo que es modelar y lo que es visualizar da pié a que se produzcan numeros problemas de codificación.

La diferencia entre modelar y visualizar es clara: el modelo es el conjunto de datos que representan al objeto, la visualización es una de las posibles acciones que se pueden hacer con dichos datos. Si asumimos que el modelo no cambia (salvo por las tranformaciones geométricas), es fácil entender que la creación del modelo sólo se realiza una vez mientras que la visualización se puede llevar a cabo tantas veces como sea necesario.

En el código que se entrega, la funcion **draw_scene()** se llama cada vez que es necesario dibujar la escena, implicando la limpieza de la ventana, el posicionamiento de la cámara, el dibujado de los ejes, y el dibujado de los objetos que hayamos definido.

```
void draw_scene(void)
{
  clear_window();
  change_observer();
  draw_axis();
```

```
  draw_objects();
  glutSwapBuffers();
}
```

El primer problema que nos podemos encontrar es el siguiente:

```
void draw_objects()
{
  float Vertices[8][3]={{1,0,1},...};
  ...
}
```

Este código implica que se va a repetir la creación de 24 flotantes cada vez que se llame la función para dibujar, aunque los valores no cambien. Con 8 vértices el efecto no se nota, pero pongamos que se tienen que inicializar un millón de vértices y en seguida veremos lo ineficiente de la implementación. Esta opción sólo la usaremos como una forma rápida de probar algo.

Existen otras variantes del mismo tema:

```
void draw_objects()
{
  vector<_vertex3f> Vertices;

  Vertices.push_back(_vertex3f(0.5,0.5,0.5));
  ...
}
```

También se comete el mismo error aunque el vector se cree una sola vez si finalmente se le introducen los datos cada vez que se dibuja.

```
vector<_vertex3f> Vertices;

void draw_objects()
{
  Vertices.clear();
  Vertices.push_back(_vertex3f(0.5,0.5,0.5));
  ...
}
```

En algunas implementaciones, la función de dibujado necesita que se le pasen los vectores de los vértices y de los triángulos, y dependiendo de un modo, que puede ser otro parámentro o una variable global, dibujar de una manera u otra:

```
void draw(vector<_vertex3f> &Vertices1,vector<_vertex3i> &Triangles1,
    int Mode)
{
  switch (Mode){
```

```
  case VERTICES:
    ...
    break;
  ...
  }
  ...
}
```

Dado que la función es única para todos los objetos, una de las posibles soluciones erróneas consiste en copiar los datos del objeto seleccionado a unos vectores genéricos

```
vector<_vertex3f> &Vertices_cube;
vector<_vertex3f> &Vertices_tetrahedron;
vector<_vertex3f> &Vertices;

vector<_vertex3i> &Triangles_cube;
vector<_vertex3i> &Triangles_tetrahedron;
vector<_vertex3i> &Triangles;

void draw_objects()
{
  switch (Object){
  case CUBE:
    Vertices=Vertices_cube;
    Triangles=Triangles_cube;
    draw(Vertices,Triangles,Mode);
    break;
  ...
  }
}
```

En este caso la solución es sencilla:

```
vector<_vertex3f> &Vertices_cube;
vector<_vertex3f> &Vertices_tetrahedron;

vector<_vertex3i> &Triangles_cube;
vector<_vertex3i> &Triangles_tetrahedron;

void draw_objects()
{
  switch (Object){
  case CUBE:
    draw(Vertices_cube,Triangles_cube,Mode);
    break;
  ...
  }
}
```

El mismo problema ocurre si lo que se igualan son objetos:

```
_object3D Cubo;
_object3D Tetraedro;
_object3D Objeto;

void draw_objects()
{
  switch (Object){
  case CUBE:
    Objeto=Cubo;
    draw(Objeto,Mode);
    break;
  ...
  }
}
```

Un alivio al problema consiste en usar punteros, pero esta solución introduce el bien conocido problema de la gestion de punteros, y sobre todo, hay mejores soluciones.

Otro error consiste en utilizar una variable booleana para determinar si el objeto se debe crear o no

```
_object3D Cubo;
_object3D Tetraedro;
_object3D Objeto;

void draw_objects()
{
  switch (Object){
  case CUBE:
    if (Cubo_created==false){
      create_cube();
      Cubo_created==true;
    }
    draw(Cube,Mode);
    break;
  ...
  }
}
```

No es un problema grave, pero la cuestión es: ¿se puede hacer mejor de una forma sencilla? La respuesta es sí, basta con crear los objetos al principio, por ejemplo en el main.

Siguiendo en la misma línea, suele ser común encontrar el siguiente código para la práctica segunda.

```
void draw_objects()
{
  switch (Objet){
  case PLY:
    open_file("ant.ply");
    read_ply(Coordinates,Positions);
    create_ply(Coordinates,Positions);
    draw(Object_ply,Mode);
    break;
  ...
  }
}
```

¿Por qué hay que leer y crear el objeto PLY cada vez que se dibuja?

Otro conjunto de problemas son aquellos relacionados con la lectura del código. Debemos intentar que nuestro código sea fácil de leer, de seguir. ¿Qué objeto se quiere dibujar cuando se indica con el número 1? ¿A qué modo de visualización nos referimos cuando ponemos el número 2?

```
void draw_objects()
{
  switch (Objet){
  case 0:
    open_file("ant.ply");
    read_ply(Coordinates,Positions);
    create_ply(Coordinates,Positions);
    draw(Object_ply,Mode);
    break;
  case 1:
  ...
  }
}

void draw()
{
  switch (Mode){
  case 0:
    glBegin(GL_POINTS);
    ...
    break;
  case 1:
    glBegin(GL_TRIANGLES);
    ...
    break;
  ...
  }
```

```
}
```

Se puede y deben usar defines o constantes para tener valores textuales en vez de numéricos.

```
typedef enum{TETRAHEDRON,CUBE,PLY} _object_type;
typedef enum{VERTICES,EDGES,SOLID} _rendering_mode;

void draw_objects()
{
  switch (Objet){
  case TETRAHEDRON:
    draw(Tetrahedron,Mode);
    break;
  ...
  }
}

void draw()
{
  switch (Mode){
  case VERTICES:
    glBegin(GL_POINTS);
    ...
    break;
  case EDGES:
    glBegin(GL_TRIANGLES);
    ...
    break;
  case SOLID:
    glBegin(GL_TRIANGLES);
    ...
    break;
  ...
  }
}
```

Si nos fijamos, en este último ejemplo, al querer introducir el modo en el que se dibuja a la vez como vértices, aristas y sólido, la solución pasa por repetir el código de cada modo.

```
typedef enum{VERTICES,EDGES,SOLID,ALL} _rendering_mode;

void draw()
{
  switch (Mode){
  case VERTICES:
```

```
    glBegin(GL_POINTS);
    ...
    break;
  case EDGES:
    glBegin(GL_TRIANGLES);
    ...
    break;
  case SOLID:
    glBegin(GL_TRIANGLES);
    ...
    break;
  case ALL:
    glBegin(GL_POINTS);
    ...
    glBegin(GL_TRIANGLES);
    ...
    glBegin(GL_TRIANGLES);
    ...
    break;
  ...
  }
}
```

Es fácil ver que cualquier modificación se complica bastante. La solución pasa por crear funciones para cada modo de visualización.

```
typedef enum{VERTICES,EDGES,SOLID,ALL} _rendering_mode;

void draw()
{
  switch (Mode){
  case VERTICES:
    draw_vertices();
    break;
  case EDGES:
    draw_edges();
    break;
  case SOLID:
    draw_solid();
    break;
  case ALL:
    draw_vertices();
    draw_edges();
    draw_solid();
    break;
  ...
  }
```

```
}
```

Otra forma de complicar la comprensión de un programa consiste en usar cadenas largas de if-else.

```
typedef enum{VERTICES,EDGES,SOLID,ALL} _rendering_mode;

void draw()
{
  if (Mode==VERTICES) draw_vertices();
  else
  if (Mode==EDGES) draw_edges();
  else
  if (Mode==SOLID) draw_edges();
  else
  if (Mode==ALL) draw_all();
  else
  ...
}
```

Es fácil ver por qué se creo el switch. Usémoslo cuando tengamos 3 o más opciones.

Ahora un problema de eficiencia. Consiste en usar la funcion **push_back()** con los vectores. Como sabemos, los vectores son capaces de crecer automáticamente cuando ya no disponen de más espacio. Para ello, solicitan el doble de espacio al sistema, copian el contenido al espacio nuevo y liberan el antiguo. Dado el mínimo que tenga la STL, Min, una aproximacion del número de veces que necesitaremos solicitar memoria es, dando el número de elementos que necesitemos N $\log_2 N - \log_2 Min$. Por ejemplo, sin Min es 16 y necesitamos 1024 elementos, tenemos $\log_2 1024 - \log_2 16 \rightarrow 10 - 4 = 6$. Necesitaremos duplicar el tamaño 6 veces, y en la última copiaremos 512 elementos. Es fácil observar que se producirá una degradación en el funcionamiento. Sólo usaremos **push_back()** cuando no conozcamos el número de elementos que vamos a introducir.

Otro problema en la forma de codificar que he detectado es en la estructuración del código. La forma estandar de estructurar el código implica que la definiciones están serparadas de las implmentaciones. En las primeras se indica cual es la interfaz y en la segunda se define cómo se lleva a cabo. Por ejemplo, es común definir unos ficheros con las cabeceras y otros con el código. En C, los primeros serían los ficheros acabados en .h y los segundos serían los ficheros acabados en .c. Para C++ podemos encontrarnos distintas formas: .h, .hxx, .hpp, .cc, .cpp, .cxx, etc.

Lo que no tiene sentido es tener una estructura en la que las definiciones se encuentran en un fichero del tipo .h y las implmentaciones se encuentran en un .hxx, y que los segundos sean incluidos en el primero. Por ejemplo:

object3D.h

```
#include <vector>
```

```
#include "vertex3f.h"

class _object3D
{
public:
        _object3D();
  void  draw_points()

  std::vector<_vertex3f> Vertices;
}

#include "object3D.hxx"
```

object3D.hxx

```
_object3D::_object3D()
{
  ...
}

void _object3D::draw_points()
{
  ...
}
```

Esta forma de codificar implica que cada vez que se incluya el .h se incluirá y compilará la implementación, lo cual es un esfuerzo inútil puede llevar a simbolos duplicados. Lo lógico es poner el código de esta forma:

object3D.h

```
#include <vector>
#include "vertex3f.h"

class _object3D
{
public:
        _object3D();
  void  draw_points()

  std::vector<_vertex3f> Vertices;
}
```

object3D.cc

```
_object3D::_object3D()
```

```
{
  ...
}

void _object3D::draw_points()
{
  ...
}
```

y reflejarlo en el Makefile o project.

¡Importante! Está regla de uso sólo nos la podremos saltar cuando no quede más remedio, como por ejemplo cuando usamos plantillas (*templates*). En tal caso, sólo se nos permite crear un fichero .h que contiene la definición y la implementación. Es como lo hace la *Standard Template Library*.

En línea con lo comentado, no hay que olvidar que hay que controlar que el contenido de un punto .h sólo se carge una vez. Para ello se recurre a la siguiente construcción:

object3D.h

```
#ifndef _OBJECT3D_H
#define _OBJECT3D_H

#include <vector>
#include "vertex3f.h"

class _object3D
{
public:
        _object3D();
  void  draw_points()

  std::vector<_vertex3f> Vertices;
}

#endif
```