IMAGE CAMPUS

- OTRA EDUCACIÓN -

Requisitos sobre el código para entregas de proyectos

Desarrollo de Videojuegos I - 2022

Sergio Baretto

1) Usar C++ y no C

Por ejemplo:

- Si se hacen includes de librerías estándar de C (math, etc.) usar `<cmath>` en vez de `<math.h>`
- Si se crean estructuras usar `struct` en vez de `typedef struct`
- Si se debe hacer un casteo (por ejemplo para convertir un float a un int de manera de evitar un warning por pérdida de precisión) usar `static_cast<int>(num)` en vez de `(int)num`

2) Evitar `#defines` para definir constantes

Reemplazar por `const` y `enum`.

3) Ser consistente en el código

Por ejemplo:

- Si se abren las llaves arriba, abrirlas arriba para todo
- Si se usan nombres en inglés, no poner algo en inglés y algo en castellano

4) Evitar números mágicos

Poner constantes o enumeradores con un nombre claro de lo que representan.

5) Modularizar el código

Crear archivos de código para diferentes partes, contenidos o módulos del proyecto.

Por ejemplo:

- Para las escenas
- Para el juego en sí (game, app o similar)
- Para los elementos visibles en el juego

6) Por cada .cpp (a excepción del main) debe haber un .h del mismo nombre

El header (.h) debe contener las cabeceras y declaraciones.

7) Main mínimo

La función `main` debe ser mínima, por ejemplo haciendo que llame a una función que ejecute el juego y nada más.

Reducir también en el main los includes a los mínimos necesarios.

8) Función `main` en archivo main.cpp

La función `main` debe estar en un archivo llamado main.cpp (todo en minúsculas) ya que es un nombre estándar para marcar el punto de entrada.

Este archivo debe estar suelto en la carpeta src y sin filtros en Visual Studio.

9) Incluir .h sólo si se usa en el código que lo incluye

Solo se debe hacer `#include` de un determinado header (`iostream`, librerías, otro archivo de código del proyecto, etc.) si en el archivo de código que lo incluye se van a utilizar las funciones declaradas en dicho header.

Lo mismo aplica para los namespaces: solo usar using namespace si dicho namespace va a ser utilizado.

10) Poner en .h solamente lo que es accedido desde otros archivos

Las declaraciones de funciones y variables que se usan solamente en un archivo .cpp no deberían estar en su .h (como parte de la interfaz que este archivo expone).

Por lo tanto, estas deberían estar sólo en los .cpp en donde se usan y deberían marcarse como `static`.

Nota para los que están viendo programación orientada a objetos: esto no aplica si se declara una clase en el header.

11) Mantener orden de funciones en .h y .cpp

El orden en que se declaran las funciones en un archivo .h debe repetirse cuando estas funciones se definen en el .cpp.

Por ejemplo, si en el .h encontramos:

```
void funcA();

void funcB();
```

En el .cpp entonces debe encontrarse la definición de las funciones siguiendo el mismo orden (primero A y luego B).

A su vez, procurar que el ordenamiento siga un orden lógico.

Por ejemplo, si las funciones son init, update y close, deberían estar en este orden y no primero el close y luego el init.

12) Contener todo el código del proyecto dentro de un namespace

Opcionalmente puede haber más sub-namespaces dentro de este.

Pero todo el código (excepto la función main) debe estar contenido dentro de al menos un mismo namespace.

13) Mantener una lógica y ordenamiento al hacer include

Ordenar los diferentes includes que se hacen en el código (de librería estándar, de otras librerías, de otros archivos .h del proyecto, etc.) de la siguiente manera:

```
#include "player.h"

#include <iostream>
#include <string>
#include <cmath>

#include "raylib.h"

#include "weapon.h"
#include "inventory.h"

using namespace std;

...
```

1ro: poner include del .h correspondiente a este .cpp
Dejar espacio

2do: poner includes de librería estándar (iostream, etc.) si se usa

Dejar espacio
3ro: poner includes de otras librerías (raylib, por ejemplo)
Dejar espacio

4to: poner includes de otros archivos del proyecto

Dejar espacio
5to: namespaces
Dejar espacio
Resto del código

14) Poner #include guards en todos los archivos .h (o usar #pragma once)

La #include guard es una técnica para evitar que un header (.h) sea incluido múltiples veces, evitando problemas que esto podría traer.

Por lo tanto, en todos los .h deben incluirse directivas de `#ifndef #define #endif` relacionadas con el nombre del archivo.

Por ejemplo, si el archivo se llama `main_menu.h`, debe hacerse:

```
#ifndef MAIN_MENU_H
#define MAIN_MENU_H

// declaraciones

#endif // MAIN_MENU_H
```

Todo el contenido del .h debe estar comprendido dentro de estas cláusulas.

Dejar un espacio al comienzo entre el #define y las declaraciones, así como al final entre las declaraciones y el #endif.

15) Evitar .. al hacer include

Se debe poder leer y entender claramente lo que se incluye, por eso en vez de tener:

```
#include “../../scenes/menu.h”
```

se debe tener:

```
#include “scenes/menu.h”
```

Para esto es necesario poner como directorio de inclusión adicional a la carpeta src del proyecto.

16) Nombres de archivo de código fuente no deben tener espacios y preferentemente ser todos en minúscula

Los archivos .h y .cpp no deben contener espacios.

Lo mismo aplica para las carpetas dentro de src.

Cambiar los espacios por un guión bajo si llevan más de una palabra.

Por ejemplo: en vez de mainmenu.h llamarlo main_menu.h para facilitar la lectura.

También preferir nombres de archivo todo en minúsculas: main_menu.h en vez de MainMenu.h o Main_menu.h o mainMenu.h.

NOTA: no se aclaran acá otros requisitos de buenas prácticas sobre el código que a esta altura se dan por sabidos y que se continúan evaluando, tales como:

- tener nombres de variables y funciones claros
- hacer las funciones lo más específicas y acotadas posibles
- etc.

Links útiles: algunos recursos que pueden resultar de utilidad:

- https://google.github.io/styleguide/cppguide.html
- https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
- https://isocpp.org/wiki/faq/coding-standards