CURSO DE PROGRAMACIÓN DE VIDEJUEGOS CON C++ Y ALLEGRO

artebinario@netexplora.com
http://artebinario.cjb.net
http://artebin.sf.net

Revisa la nueva sección **Introducción::Licencia** e **Introducción::Agradecimientos y Dedicatorias**

Capítulo 1 : INTRODUCCIÓN

Licencia

Espero no hacer esta sección muy relevante porque este manual lo entrego sin fin de lucro, es casi un regalo, un acto de altruismo esperando a cambio un flujo constante y creciente de sugerencias, de expansión, de colaboración, de correcciones.

También espero que menciones el sitio desde donde lo sacaste (http://artebinario.cjb.net), que te hagas miembro de la lista de correo (http://www.elistas.net/lista/programacion_juegos) y que juegues mucho.

Agradecimientos y dedicatorias

Agradecimientos a todas las personas que han descargado este curso que se han pasado el dato “de boca en boca”. A las personas que desarrollan Allegro, Djgpp, RHIde, Linux, allegro.cc, Pixelate, a los creadores del libro “*Cutting Edge: Java Game Programming*”, a mirax y en general a todos los que han prestado ayuda a Artebinario desde sus albores.

Una especial dedicatoria al amor de mi vida, su nombre empieza con “E” y prefiere de todas maneras el anonimato (no es tan ególatra como yo :), le pido disculpas por las horas que no le dedico en pos de este curso.

Motivación

He visto muchas páginas (demasiadas) relacionadas con la programación de videojuegos, casi todas están en inglés y no son comprensibles muchos de los términos que utilizan. De los sitios que he visto en castellano sobre programación de videojuegos muchos se quedan en "esta página está en construcción", "próximamente más artículos", etc. eso no me gusta y yo no quiero caer en ese tipo de temática. Este curso lo terminaré y les entregaré a Uds. todos los conocimientos que he adquirido hasta ahora, así como los libros e incluso aportes de muchos navegantes. Quiero terminar este curso también a la par de la primera versión del juego ADes, porque los que han seguido de cerca el proyecto saben que queda totalmente estancado en los semestres de clases y muchas veces también en las vacaciones, yo les digo a ellos que ahora tengo más documentación y motivación para emprender esta tarea, he conocido más sobre el diseño de videojuegos: frameworks para videojuegos, UML, tarjetas CRC, etc. Muchas cosas que tú también aprenderás.

Buena suerte a mi y a todos los que quieren emprender este desafío (desafío 99... solo algunos saben lo que significa eso) ;-)

Requisitos

Antes que todo, Uds. deben saber los requisitos para que avancemos a un ritmo considerable. Lo más importante es tener ganas y saber “C”, pero ni siquiera eso, porque los que son demasiado "maestros" en C tienden a encontrar dificultades para aprender programación orientada al objeto:

- Conocimiento de C
- Conocimiento de la programación orientada al objeto
- Conocimiento de Djgpp y las bibliotecas allegro
- Conocimiento del editor RHide
- (Opcional) Conocimiento de VC++ y compilación con DirectX.
- (Opcional) Conocimiento de editores de imágenes tales como PhotoShop, PaintShop Pro, Fireworks, etc.

Objetivos

Aquí viene la parte en que Ud. se entusiasman y empiezan a aportar en este proyecto, es la parte en que sueñan con algún día hacer un juegos desde el principio hasta el final Los objetivos son:

- Comprender que es el diseño orientado al objeto, orientado a los videojuegos
- Aprender a crear un espacio de trabajo
- Conocer como trabaja Allegro y Djgpp
- Aprender a compilar, ejecutar y depurar un programa con Allegro y Djgpp
- Aprender a ocupar el editor RHIDE
- Aprender a implementar el diseño de las clases
- Crear un videojuego basado en la orientación a objetos.

Creo que esos son los principales objetivos de mi sitio y de este curso. Debo advertirles además que principalmente me orientaré a la programación de juegos de naves (shoot'em up) como ADes, pero con los frameworks (ya verán lo que eso) no tendrán problema para adaptarlo a su propio videojuego.

C++: El Lenguaje

Tendría que esforzarme demasiado para decir algo nuevo sobre C++, probablemente repetiré lo que he venido diciendo durante mucho tiempo o lo que siempre leo: "C++ es un lenguaje orientado al objeto ocupado en la mayoría de las aplicaciones de diseño de software. El caso de nuestro objetivo lo ocuparemos para programar unos videojuegos, de cualquier tipo, ya sea de scroll, de luchas (¡street fighter!) o 3d.

Si aprenden C++ bien serán parte de una comunidad de “elite” dentro de los programadores incluso podrán ganar mucho dinero creando software de bajo nivel, desde los cuales se sustentarán sistemas mucho más complejos. Pero, nuevamente, los voy a traer hasta nuestra realidad: establecer una base para crear nuestros futuros videojuegos. Ahora que nombre esto de "crear una base para nuestros futuros vieojuegos" les paso a explicar la ventaja más excitante de programar orientado al objeto. Si nosotros creamos un diseño relativamente bueno que sustente a nuestro videojuego podremos en un futuro seguir ocupando estas "clases" de objeto (ya veremos este concepto), es decir, no tendremos que invertar la rueda de nuevo (un error frecuente en un programador). Otra característica de C++ es que ofrece la robustez de nuestro diseño mediante la ocultación de información, es decir, no todas las personas que ocupen nuestro código podrán acceder a todo el código. Por ejemplo, no podrán acceder a las estructuras que guardan la información en un tipo de dato lista, podremos restringuir el acceso del programador que ocupará nuestras clases a las funciones canónicas insertar, eliminar o borrar un elemento. Bueno, mejor leánse un libro o visiten la página de un tipo que escribió el mejor libro de C++ y lo dejó abierto para ser leído gratuitamente (http://mindview.com) y el libro se llama "Thinking in C++" y está en dos tomos. Yo les recomiendo de todas formas leerse un libro o por lo menos saber los conceptos de la programación orientada al objeto.

Si sabes programar en algún otro lenguaje orientado a objetos como SmallTalk, Java no deberías tener problemas con C++.

Compiladores de C++

Hay muchos compiladores para el lenguaje C++ en todos los sistemas operativos (todos), porque casi todos los sistemas operativos están hechos en C o C++. Sin embargo, solo algunos de ellos se pueden usar conjuntamente con Allegro en su versión más reciente (3.9.35) algunos de ellos son Djgpp, Visual C++, Watcom y Borland C++. En todo caso si vas al sitio Allegro (http://alleg.sourceforge.net/) puedes ver todos los compiladores que soporta Allegro.

Yo personalmente me voy ha basar en el compilador Djgpp ya que es gratuito y tiene una comunidad grande de personas que lo ocupan, además de que se basa en el compilador de Unix, es decir, puedo pasar compilar fácilmente un programa hecho en Djgpp en Unix y por consecuencia en Linux. Djgpp también trata de estar siempre acorde con la estandarización ANSI. El código fuente trataré de que sea compatible con todos los siguientes compiladores: GCC Unís, Visual C++ y Djgpp.

Ambientes de Desarrollo Integrados (IDEs)

Un IDE (Integrated Development Enviroment), como se conoce comúnmente a un editor, permite integrar varias herramientas de desarrollo tales como: compilador, enlazador (linkeador) y depurador. Además también puede permitir crear proyectos y escribir los archivos que luego realizaran automáticamente las tareas de compilación (archivo de *makefile*).

Yo también daré mis ejemplos en el IDE RH (RHIde) creado por Robert Höhne que permite hacer todo lo que les dije anteriormente. Ahí crearemos el proyecto y terminaremos creando el ejecutable.

Visual C++ tiene un potente IDE y no debería ser problema aplicarlo a este proyecto.

Herramientas de desarrollo para Allegro

Además de ocupar las bibliotecas de Allegro para las primitivas en nuestro juego (líneas, puntos, triángulos, rellenos, sprites, música, etc.) también ocuparemos las herramientas de desarrollo que trae Allegro que nos permitirán configurar el sonido, los periféricos y todo lo que se necesitará para jugar nuestro juego, además de una magnífica herramienta que nos permitirá comprimir todos nuestros elementos de Allegro (música, imágenes, texto, sonido) en un solo archivo y además comprimirlo.

No te alarmes por todas estas cosas que he presentado, todas ellas le iremos viendo a su debido tiempo.

Capítulo 2 :
CREANDO NUESTRO PRIMER JUEGO

Creando un espacio de trabajo

Lo primero que debes aprender al empezar un proyecto medianamente grande es ser organizado, y créeme: cuesta mucho aprender a serlo. Yo te puedo dar algunos consejos, no hay problema, pero tú debes continuar y hacer del orden tu forma de vivir por lo menos en la programación porque ante todo te permitirá mantener en un lugar seguro todos los activos de tu proyecto, fáciles de ubicar, fáciles de rehusar, fáciles de arreglar, en general, fáciles de mantener.

Estos algunos de los pasos que yo te recomiendo y que también seguiré en mi proyecto de juego, así también estarán ubicados los códigos y encabezados de mi proyecto. Esta organización se puede dar como una especie de árbol comúnmente conocido como directorio. Por ejemplo,

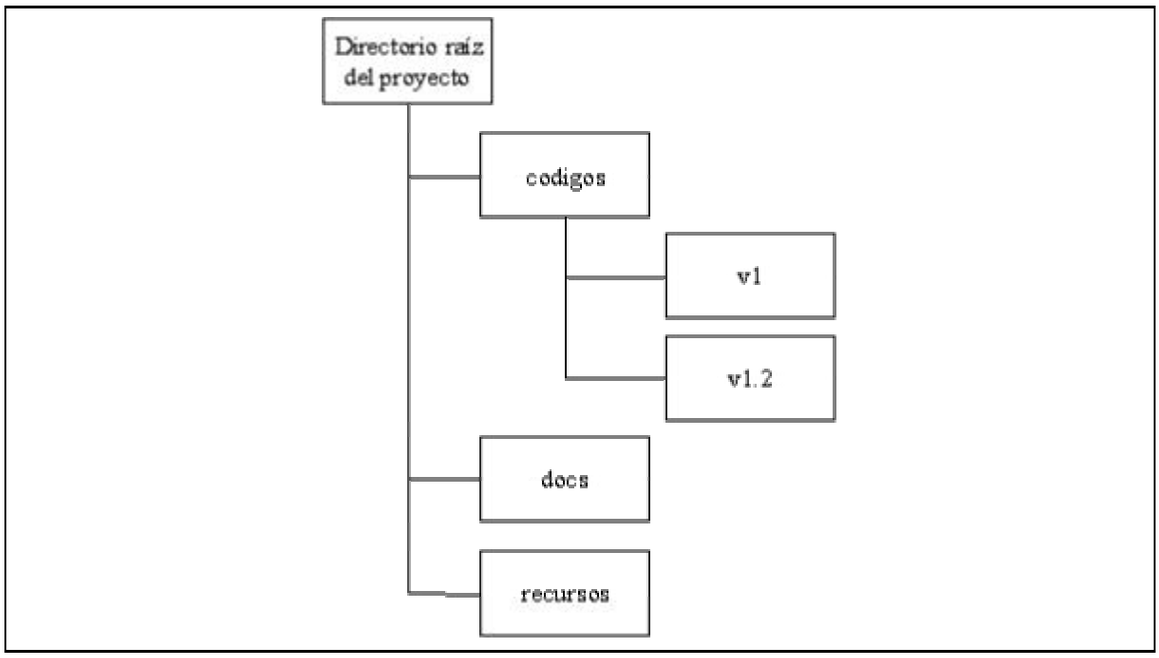


Figura 1

El directorio raíz de nuestro proyecto puede estar en cualquier parte de nuestro disco duro o disco de red. Además dentro del directorio códigos puede considerarse una subclasificación entre las versiones de nuestro proyecto, por ejemplo la versión 1, versión 1.2, etc.

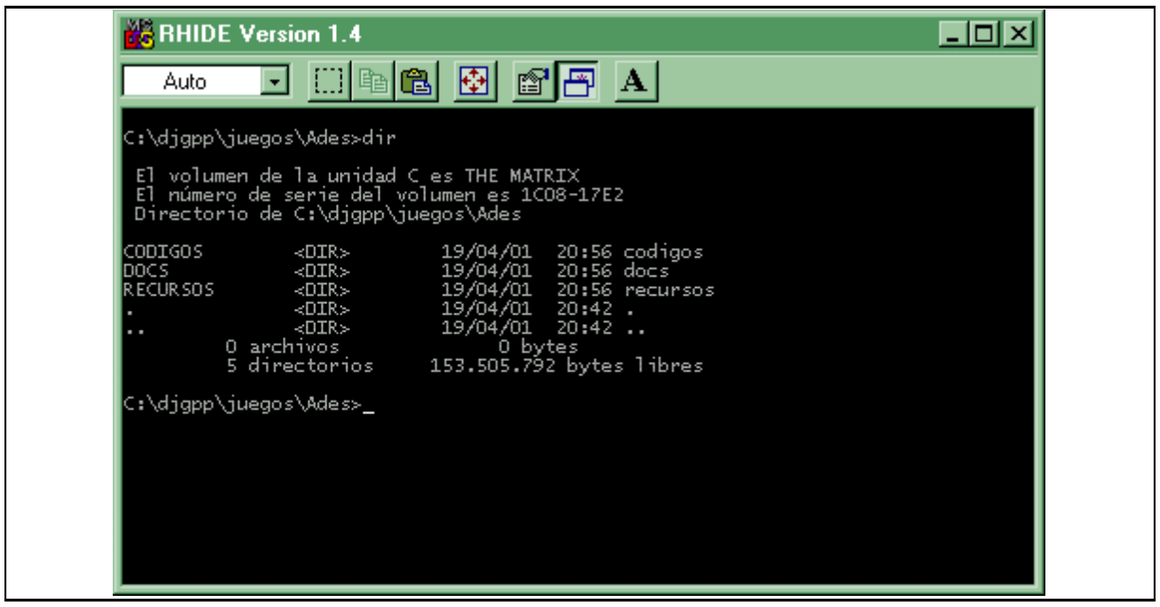


Figura 2

Ahora que tenemos creado la estructura de directorios entonces creamos el proyecto en RHIDE.

Desde ahora en adelante me referiré a los directorios en forma relativa al directorio raíz del proyecto, por ejemplo, si hablo del archivo códigos/prueba.cpp entonces me referiré al archivo que está, en el caso mío, en c:\djgpp\juegos\Ades\codigos\prueba.cpp (MS-DOS)

Entramos al directorios código y escribimos RHIDE.

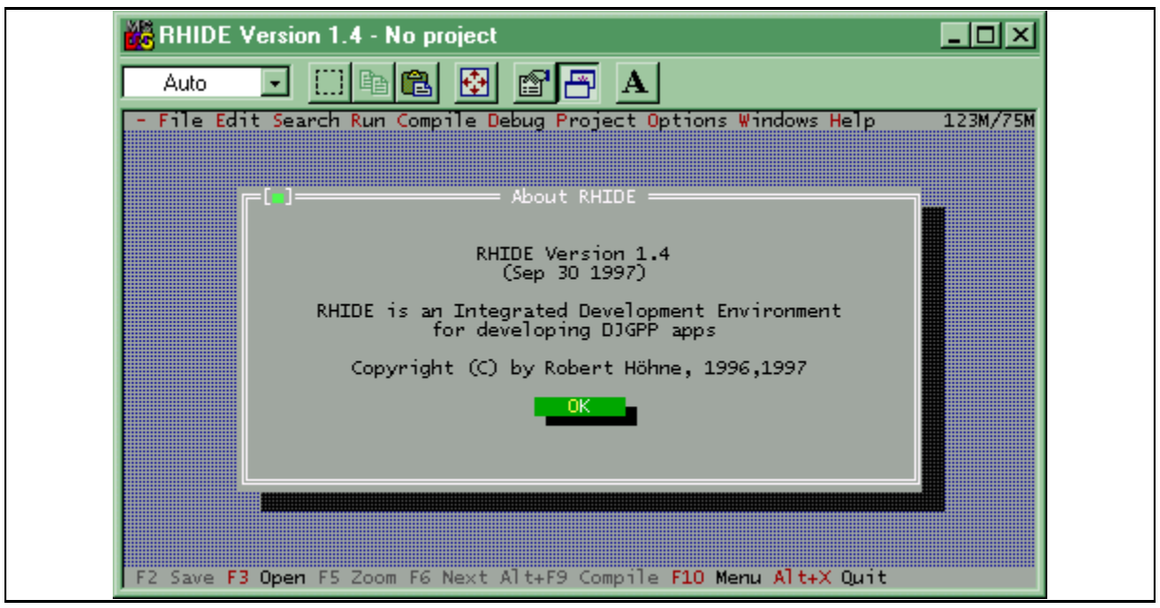


Figura 3

Ahora que nos encontramos dentro de RHIDE podemos ver varios menús, uno de ellos es el menú de proyectos->abrir proyecto (proyect->open proyect, **desde ahora no traduciré los menús por lo que tu debes entender de cual estoy hablando**). Entonces saldrá un dialogo que nos preguntará cual es el nombre del proyecto. Este debe ser un nombre explicativo, que diga algo, por ejemplo a mi proyecto le pondré ades y el archivo correspondiente será *ades.gpr*.

Para editar los códigos fuentes yo te recomiento personalmente usar las siguientes opciones: anda al menú opciones->ambiente->opciones del editor y haz clic en las siguientes y deja el tamaño de la tabulación así:

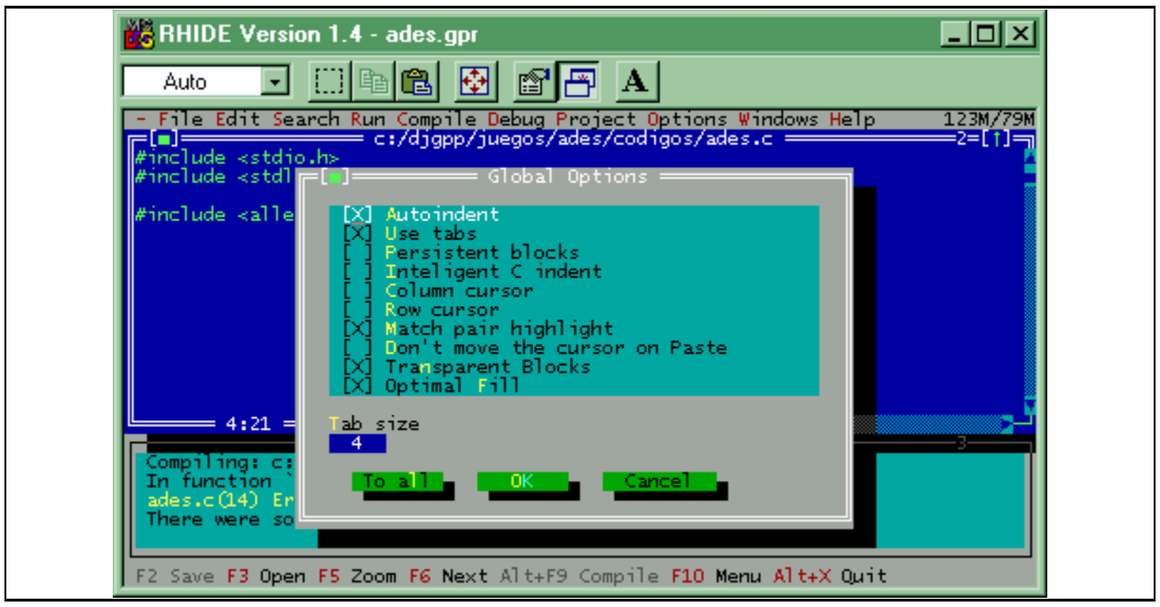


Figura 4

Ahora, para compilar nuestro código con las bibliotecas Allegro le diremos al compilador que las enlace a nuestro proyecto. En el menú opciones ->bibliotecas ingresamos el nombre *alleg* y dejamos seleccionada la casilla, tal como lo muestra la Figura 5.

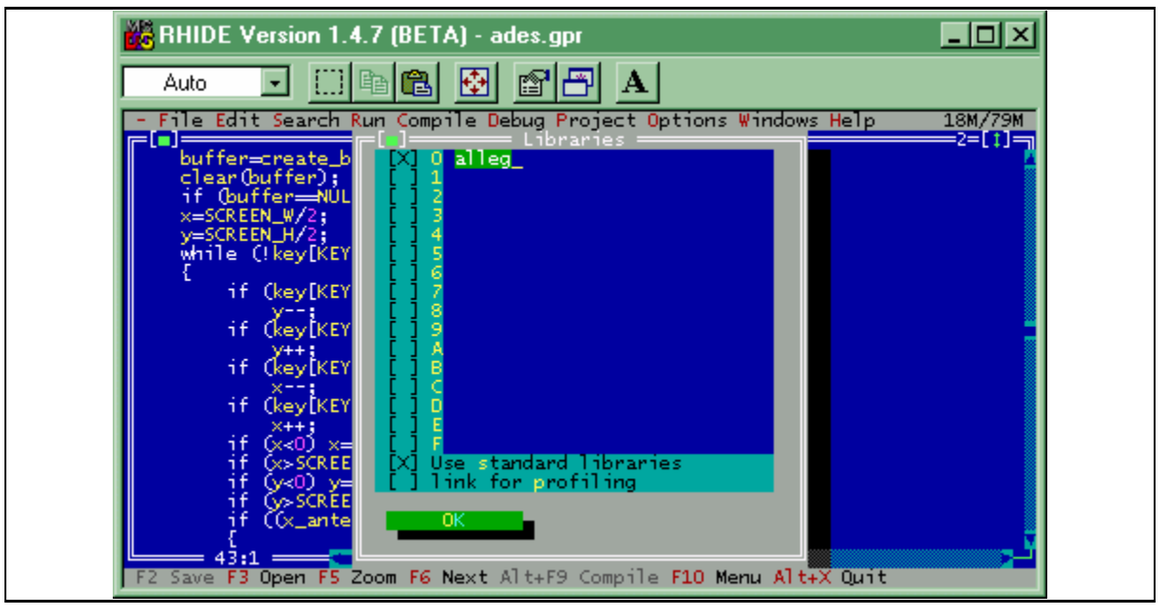


Figura 5

Luego de ingresar el nombre aparecerá una ventana del proyecto que en un futuro mostrará todas las fuentes involucrados en el proyecto. Por ahora crearemos un nuevo archivo desde el menú *archivo->nuevo.*

Ahora introduciremos un código más o menos canónico para comprobar que todo está trabajando en Allegro. Introduce lo siguiente:

Código Fuente 1.

```
#include <stdio.h>
#include <stdlib.h>
#include <allegro.h>

int main()
{
    allegro_init();
    install_keyboard();
    if (set_gfx_mode(GFX_AUTODETECT,320,200,0,0)<0)
    {
        printf("error al iniciar modo grafico\n");
        allegro_exit();
        exit(-1);
    }
    textout(screen, font, "Mi primer programa", 0, 0, 15);
    readkey();
    allegro_exit();
    return 0;
}
```

Ahora que tienes todo esto escrito entonces guardalo con el nombre *ades.c*, luego anda al menú *proyecto->agregar ítem* y selecciona el archivo que acabas de guardar, luego anda *opciones->bibliotecas* y agrega la biblioteca *alleg* (todos estos pasos son explicados con más detalles en la página Artebinario, como parte de la instalación y ejecución de un programa en Allegro). Anda a *compilar->construir todo* y ya tienes listo el programa para ejecutarlo en *ejectuar->ejecutar*. Deberías ver algo como esto:

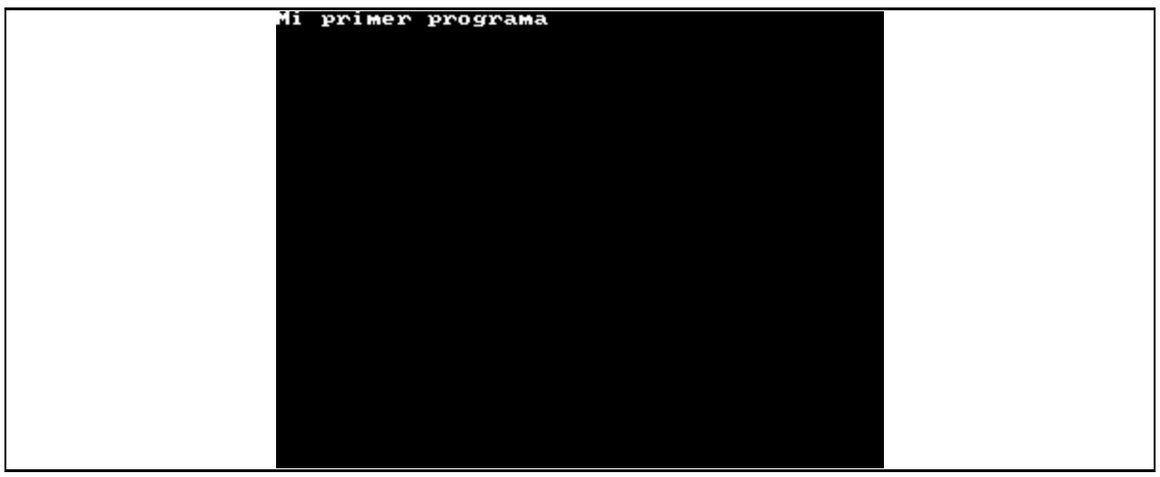


Figura 6

Ahora cada vez que entres a RHIDE desde el directorio de códigos se abrirá por defecto el proyecto.

Estableciendo un estándar en el desarrollo

Es bueno siempre establecer un estándar a través del desarrollo, esto quiere decir que se debe explicitar cuales son los formatos de archivo, cuales son las extensiones de cada tipo, cuales son las bibliotecas que se usarán, cuales son las opciones de compilación, que no se permitirá, etc. Todo esto sirve cuando se trabaja con muchas personas, un grupo de programadores sobre un mismo proyecto. Como hacer que un programador no se intrometa en mi código y como yo no me intrometo en su código o en la parte del proyecto que el está actualmente desarrollando.

En Internet hay muy buenos artículos sobre como comentar, indentar y documentar nuestro código, nuestras clases o en un futuro nuestras propias bibliotecas de desarrollo.

A modo de ejemplo yo estableceré las extensiones y directorios que tendrá cada parte de mi proyecto.

- Los códigos fuentes en C++ tendrán la extensión *.cpp*.

- Los códigos fuentes en C tendrán la extensión *.c*.
- Los archivos de inclusión que van en la cabecera tendrán la extensión *.h*
- Etc.

Lo anterior fue un pequeño ejemplo de cómo establecer tus propias directivas en el desarrollo de un proyecto. Más adelante se verán algunas reglas más específicas, pero desde ahora debes ir pensando en *tu propia estandarización*.

Estableciendo los requerimientos de nuestro proyecto

Quizá antes de empezar a probar que si Allegro funciona, que la estandarización, que esto y que lo otro, sería mejor establecer **que es lo que queremos hacer**. A simple vista puede parecer una pregunta sencilla, pero muchos de los que han programado y están empezando recién a crear su juego se encuentran ante un gran abanico de posibilidades de desarrollo de entre las cuales son muy pocas las que parecen desechables, es decir, queremos que nuestro juego tenga todas las características, todos los modos de juego, todos géneros, ocupe todos lo periféricos, etc. Sería mucho mejor establecer un objetivo final como por ejemplo lo sería mi proyecto:

- ADes es un juego de naves con deslizamiento (scroll) vertical en 2D.
- 1 o 2 jugadores
- El juego se divide en misiones
- Varios tipos de armas con niveles de poder.
- Soporte de distintos periféricos como control
- Múltiples capas de juego, incluye agua, tierra y aire.
- Efectos de sombre, luces, huella y polvo.

Ud. dirán “¡Pero todavía es demasiado pedir para mi proyecto!”, yo les digo: sí, es demasiado pedir. Pero quien les ha dicho que haremos todo de una vez, para eso se inventó el tiempo, para gastarlo, gastarlo de buena forma. Lo que fije anteriormente fueron los objetivos, que es lo que yo espero que tenga el juego final. Deben sabes que muchos de los videojuegos comerciales quizá nunca alcanzan el objetivo inicial que fue fijado y todavía están a mucha distancia de hacerlo, pero en eso se trabaja y ese trabajo es lo que se denomina la **versión de un juego**.

Dentro de nuestros objetivos nosotros estableceremos metas de corto plazo, por ejemplo para este juego la primera meta será sencilla: hacer que aparezca una nave en pantalla y que pueda moverse con el teclado. Esa si que es una *meta* sencilla para **alcanzar** un *objetivo*. Trabajemos en eso.

Primer pedazo de código: un rápido vistazo para alimentar las esperanzas.

Muchas veces en el transcurso de mi aprendizaje me he topado con gente que dice que primero que todo hay que sentarse en un escritorio, tomar papel y lápiz y crear un buen diseño. Muy bien, felicito a la persona que puede hacerlo, pero yo cómo buen programador no me puedo pasar la vida esperando ver mi nave o mi personaje surcar la pantalla al son de las teclas.

Un justificación un poco más elaborada para esta sección sería decir que alentará al ansioso lector a seguir con está insondable tarea de crear juegos.

Ya tenemos nuestro código que permite iniciar el espacio de trabajo de Allegro, bueno ahora seguiremos extendiendo eso para permitir una pequeña nave en movimiento. Separaremos las secciones de iniciación y término de las Allegro en dos procedimientos *iniciar* y *terminar*.

Código Fuente 2.

```
int iniciar()
{
    allegro_init();
    install_keyboard(); /* nos permitira utilizar el teclado */
    set_color_depth(16);
    if (set_gfx_mode(GFX_AUTODETECT,640,480,0,0)<0)
    {
        printf("error al iniciar modo grafico\n");
        allegro_exit();
        exit(-1);
    }
}

void terminar()
{
    allegro_exit();
}
```

Ahora podríamos llamar sencillamente a estos procedimientos dentro del procedimiento *main*:

Código Fuente 3.

```
int main()
{
    iniciar();
    /*
       aquí va el código para mover la nave
    */
    terminar();
}
```

Ahora explicaré algunas funciones que ocupamos en este pequeño ejemplo. La función *allegro_init* permite incializar todas las variables globales que ocupa Allegro así como crear memoria, dejar contadores en cero, etc. Por otro lado la función allegro_exit hace todo lo contrario: libera memoria, cambia modo gráfico al original, etc. La función *set_gfx_mode* permite iniciar el modo gráfico dándole el ancho y el alto en pixeles de la pantalla, en este caso son 640x480 píxeles, pero antes de llamar a esta función debemos establecer la profundidad de color mediante la función *set_color_depth*

Ven que sencillo resulta ir poco a poco encalando en complejidad. Ahora por supuesto que se complicará más, sin embargo ya tendremos probado la parte de inicialización. Debo advertirte también que quizá este código no tenga nada que ver con el resultado final que obtendremos en la sección principal de juego, pero nos permitirá adentrarnos en el método de desarrollo, en los pasos que nos permitirán llegar a resultados satisfactorios.

Dibujando una imagen, nuestra nave

Ahora veremos un poco de código específico de Allegro, pero ya te debes ir acostumbrando. Todo esto es tiempo y experiencia, programa y programa, analiza juegos hechos por otras personas, en la red hay varios códigos dando vueltas por ahí.

Nuestra nave por el momento será una imagen simple que se moverá en la pantalla según las teclas que presionemos. Más adelante veremos que la nave es generalmente visto como un *actor* en una *escena*, y el presionar una tecla es un *evento* que *gatilla* una *acción* en ese actor, que en este caso es moverse.

El archivo que contiene la imagen es *nave.pcx* y la copiaremos a nuestra carpeta *recursos*, debes editar el código fuente *codigos/ades.c* dejando las includes y separando el código.

La forma de abrir una imagen en Allegro es muy sencilla, solo debes declarar una variable como puntero al tipo *BITMAP* y asignarla al valor que retorna la función *load_bitmap*. Como estamos trabajando en una resolución de 320x200 con la opcion *GFX*_AUTODETECT generalmente eligirá el modo a 256 colores, es por eso que también debemos cargar la paleta de colores de la imagen declarando una nueva variable de tipo *PALETTE*. Por ejemplo, en nuestro proyecto crearemos ahora un nuevo procedimiento llamado *realizar_juego* que contendrá todo el código que nos permitirá mover esa nave por la pantalla.

Código Fuente 4.

```
void realizar_juego()
{
    BITMAP *nave;
    PALETTE *paleta;

    nave=load_bitmap("../recursos/nave.pcx",paleta);
    draw_sprite(screen,nave,0,0);
    readkey();
    destroy_bitmap(nave);
}
```

Nuestro procedimiento *realizar_juego* entonces cargará la nave y la mostrará en pantalla mediante la funcion *draw_sprite* en la posición (0,0) de la pantalla (*screen* es un puntero a la posición de *memoria de video* que contiene la pantalla visual). Es importante entender como son las coordenadas en la pantalla, cosa que veremos en la próxima sección.

Coordenadas de la pantalla: el mundo al revés

Las personas que han tenido cursos de matemáticas vectoriales muy bien saben que el plano cartesiano se divide en 4 sección: x>0 & y>0, x<0 & y>0, x<0 & y<0, finalmente x>0 & y<0. Que se representa de la siguientes forma:

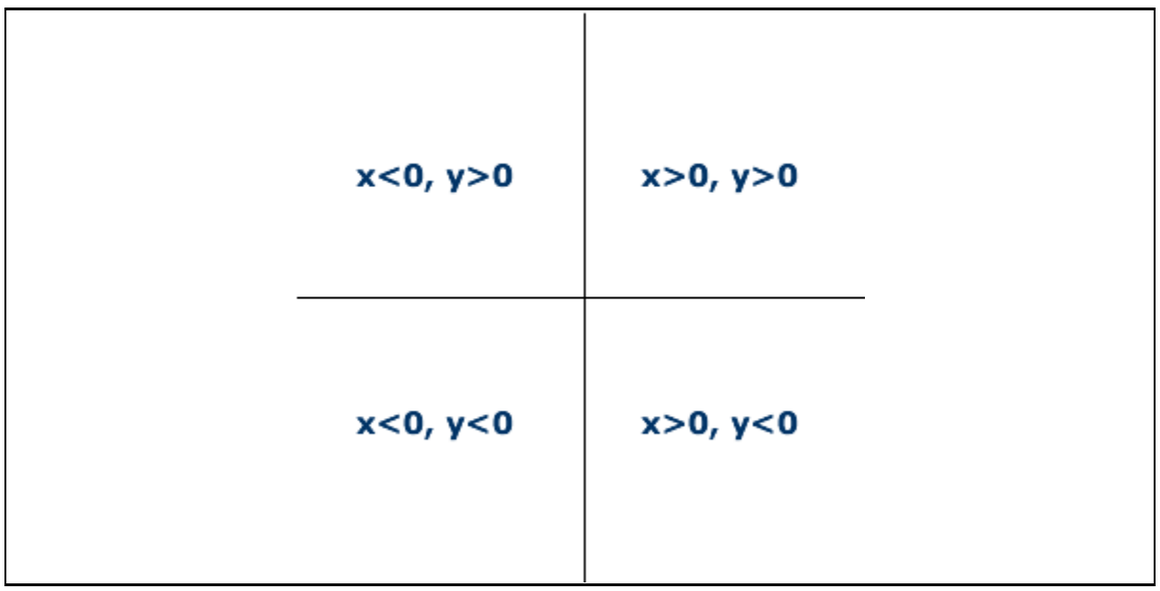


Figura 7

Bueno, ahora debes cambiar tu paradigma y pensar que la pantalla se guía por las siguientes coordenadas:

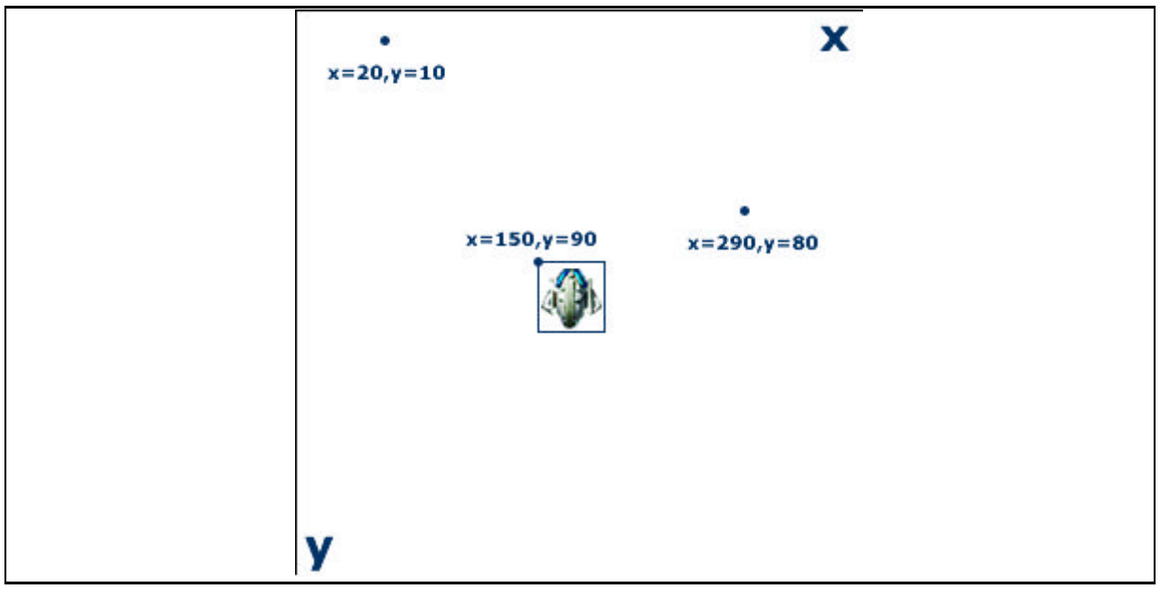


Figura 8

Tal como vez en la Figura 8 tenemos tres puntos. El primero en la esquina superior izquierda que tiene la coordenada *x=20 & y=10*, es decir, suponiendo que la pantalla empieza en los bordes superior izquierdo de la figura entonces ese punto estaría *20 pixeles* por debajo del borde superior y *10 pixeles* a la derecha del borde izquierdo. Así esta cuadriculada la pantalla, en pixeles; la unidad mínima gráfica.

Por otro lado vemos a la nave en x=*150 & y=90* pero ella continúa imprimiéndose más bajo de esa coordenada, eso es porque la imagen de la nave tiene varios pixeles de ancho y largo. Siempre que le decimos a un gráfico que se dibuje damos a la coordenada superior izquierdo del gráfico.

Es decir “el mundo al revés”, vemos el primer cuadrante del plano cartesiano (cuando x>0 & y>0) al revés, si la nave se mueve a la izquierda su coordenada *x* disminuye hasta llegar a cero o incluso seguir disminuyendo a través de número negativos, en tanto la coordenada *y* disminuye si la nave se mueve hacia arriba llegando a números negativos.

Ahora, por notación debes aceptar que cuando hablo de (0,0) estoy diciendo que *x=0 & y=0* y cuando hablo de (4,2) estoy diciendo *x=4 & y=2*.¿Muy básico?. Sí, pero no volveré a hablar de x=? e y=?. Si te manejas bien con el cálculo vectorial no deberías tener problemas de ninguna clase cuando luego resolvamos el problema de dirigir las balas de nuestros enemigos a nuestra nave. Si no entendiste no importa, porque es otro intento de alentarte a seguir en este desafio99.

Dando movimiento a la nave

Ahora que ya entiendes como son las coordenadas de la pantalla (o del monitor para los más duros de cabeza) intentaremos decirle que responda a algún periférico, por el momento será nuestro teclado.

Leyendo desde el teclado

Cuando inicializamos Allegro llamamos al procedimiento *install_keyboard* que lo que hace es instalar una interrupción que revisa a cada instante los eventos del teclado. Al llamar a este procedimiento también dejamos inutilizadas las funciones propias de C/C++ como gets, scanf, etc. Pero no te preocupes porque Allegro las reemplaza con otras que se adaptan perfectamente a nuestras necesidades.

¿Te haz fijado que cuando lees del teclado no puedes hacerlo continuamente?. Eso es porque el teclado no está programado inicialmente para leer varias cientos de veces por segundo, es decir, si mantenemos presionada una tecla esta tendrá un retardo y luego una repetición con un retardo demasiado grande entre cada carácter, esto no es usable en un juego. Por eso, al instalar el teclado dejamos una interrupción que nos permite leer cientos de veces por segundo el estado del teclado.

Para leer desde el teclado se necesita consultar un arreglo global que tiene Allegro, es decir, que es alcanzable por cualquier función o procedimiento en cualquier parte de nuestro código. Este arreglo es *key* que contiene 256 posiciones desde el 0 al 255. Pero nosotros no necesitamos aprendernos cada uno de los códigos de las teclas, para eso están definidos, globalmente también, algunas teclas comunes. Últimamente también se soportan caracteres UNICODE con lo que se permiten hasta 65536 diferentes teclas.

Por ejemplo si queremos saber si la tecla *Enter* está presionada entonces hacemos el siguiente código:

Código Fuente 5.

```
if (key[KEY_ENTER])
{
    printf("Tecla enter fue presionada\n");
}
```

¡Muy fácil!. Si dejamos esta sentencia dentro de un bucle, por ejemplo *while* se imprimirá muchas veces en la pantalla, sin retardo y sin espera por el retardo.

Estableciendo las teclas necesarias

En esta sección estableceremos las teclas que ocupará nuestro pequeño primer juego, más adelante todas estar opciones pueden estar automatizadas con algún menú de configuración en nuestro juego.

Las teclas de movimiento serán arriba, abajo, derecha e izquierda, y la tecla de salida de nuestro juego será la tecla de escape. Enumeradas en los códigos de Allegro serían:

- Arriba: KEY_UP
- Abajo: KEY_DOWN
- Derecha: KEY_RIGHT
- Izquierda: KEY_LEFT
- Escape: KEY_ESC

Como pueden ver los nombres son muy intuitivos por lo que no deberían haber mayores problemas. El formato a seguir es lógico, algunos ejemplo:

- Tecla A hasta la Z: KEY_A ... KEY_Z
- Tecla 0 hasta el 9: KEY_0 ... KEY_9

Estableciendo las direcciones de movimientos

Voy a hacer una revisión exhaustiva de cómo debe moverse la nave o que valores deben darse a las variables cada vez que se presione una u otra tecla.

Declaramos dos variables que contendrán la posición x e y de la nave en la pantalla:

Código Fuente 6.

```
void realizar_juego()
{
    BITMAP *nave;
    PALETTE paleta;
    int x,y,
...
```

Pensemos, ¿Qué pasa si presiono la flecha hacia arriba en mi teclado?.Intuitivamente concluimos que la nave debería “moverse” hacia arriba, pero ¿En que se traduce esa frase “moverse hacia arriba”?. ¡Claro!, en cambiar el valor de la variable *y* de algún modo.

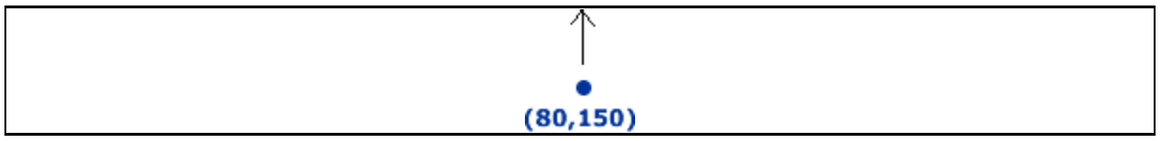


Figura 9

Supongamos que la posición inicial de la nave es (80,150), como lo muestra la Figura 9, entonces cuando presionemos la flecha arriba, es decir, cuando sea verdadera la expresión *key[KEY_UP]* debería *disminuir* el valor de la variable *y* (recuerda que en la pantalla el mundo es al revés, si algo sube su posición en la pantalla disminuye). Si no entendiste te recomiendo que repases esta parte porque es muy importante ver la relación entre el movimiento en la pantalla y las variables, imagina como se complica esto en un entorno en 3D en donde hay tres variables de posición (x, y, z).

Entonces el pedazo de código que ejecute lo requerido debería ser algo así:

Código Fuente 7.

```
...
    if (key[KEY_UP])
        y--;
...
```

Ahora hagámoslo para todas las direcciones:

- Baja la nave: y++
- Hacia la derecha: x++
- Hacia la izquierda: x—

Código Fuente 8.

```
...
    if (key[KEY_DOWN])
        y++;
    if (key[KEY_LEFT])
        x--;
    if (key[KEY_RIGHT])
        x++;
...
```

Todo esto iría dentro de un loop hasta que presionaramos la tecla *Escape*. Es decir:

Código Fuente 9.

```
...
    while (!key[KEY_ESC])
    {
        /* codigo para modificar las variables */
    }
...
```

Dibujando la nave en movimiento

Ahora que tenemos todo listo para que nuestra nave se mueva por la pantalla solo necesitamos dibujarla. Como en el ejemplo anterior la nave permanecía estática en la posición (0,0) no nos preocupábamos de actualizar el dibujo.

Cuando actualizamos la pantalla para que la nave aparezca en otra posición nos debemos preocupar de que el dibujo anterior se borre porque recuerden que la pantalla que vemos en el monitor es solo *memoria de video* que puede almacenarse mientras estemos en ese modo gráfico, es decir, necesitamos de algún modo borrar esa memoria para dibujar la nueva imagen. En el mercado existen muchos métodos para realizar esto pero por el momento solo guardaré la parte de la pantalla que voy a modificar para luego reponerla. Más adelante se verán con más detalles otros *algoritmos* que realicen en forma eficiente (lo más rápido posible) este trabajo.

Se declarará otra variable que guardará la sección de la pantalla que modificó con *draw_sprite*:

Código Fuente 10.

```
void realizar_juego()
{
    /* demas variables */
    BITMAP *buffer;
...
```

Este *buffer* debe tener las mismas dimensiones que la imagen que guardará, en este caso la variable *nave*. Para obtener las dimensiones de esta imagen se hace uso del campo *h* (height, alto) y *w* (width, ancho) luego de que esta se ha cargado por supuesto. La función de Allegro que permite crear una imagen con un formato específico es *create_bitmap*.*Voy* a mostrar todo el código del procedimiento *realizar_juego* hasta aquí:

Código Fuente 11.

```
void realizar_juego()
{
    BITMAP *nave;
    PALETTE paleta;
    int x,y;
    BITMAP *buffer;

    nave=load_bitmap("../recursos/nave.pcx", paleta);
    buffer=create_bitmap(nave->w,nave->h);
    clear(buffer);
```

Luego de crearlo lo limpiaremos porque puede tener basura la posición de memoria que le fue asignada, esto lo hacemos mediante la función *clear*. Alguno dirá "Pero si no se cargó correctamente el archivo a la variable imagen, como lo sabremos", excelente acotación. Antes de trabajar con cualquier imagen deberíamos comprobar que cargó correctamente. Así lo haremos:

Código Fuente 12.

```
...
    nave=load_bitmap("../recursos/nave.pcx", paleta);
    if (nave==NULL) terminar();
    buffer=create_bitmap(nave->w,nave->h);
    if (buffer==NULL) terminar();
...
```

En general para cualquier función de Allegro que cree memoria, retornará NULL si ha fallado. Pero alguien dirá "¡Pero que pérdida de tiempo!", en absoluto. Siempre es bueno dejar cerrado el círculo de desarrollo, es decir, comprobar que todo esté trabajando bien para encapsular esta parte del código, dejarla tranquila por el resto del proyecto.

Razonemos. Siempre debemos mantener la pantalla limpia, es decir, tal como empezó al iniciar el modo gráfico. Entonces lo que debemos hacer primero es guardar lo que se nos entregó antes de hacer alguna modificación. Declararemos dos variables que guardarán la posición anterior del sprite. Expresado en código sería:

Código Fuente 13.

```
...
    blit(screen,buffer,x_anterior,y_anterior,0,0,buffer->w,buffer->h);
    draw_sprite(screen, nave, x, y);
...
```

La primera línea copia el pedazo de pantalla que ocupará nuestra nave y lo lleva a *buffer*. La función *blit* permite copiar una sección de la pantalla completamente sin saltarse los píxeles transparentes (máximo rojo y azul en más de 256 colores, es el un color fucsia). Ahora que tocamos ese tema lo explicaré mejor.

Si ya has probado la función *draw_sprite* debes saber que ella dibuja un imagen en la pantalla pero saltándose los píxeles transparentes dando la sensación de que no es una imagen cuadrada. Si quieres puedes abrir la imagen *nave.pcx* en un editor como Photoshop o Fireworks y verás que los píxeles que no aparecen con la función *draw_sprite* son fucsias. En el caso de los modos a 256 colores (viejos modo, pero muy usados y prácticos) este color es el que está ubicado en la posición cero de la paleta (*paleta[0]*). Puedes también probar como funciona esto enviando a la pantalla las imágenes con las dos funciones *blit* y *draw_sprite* para que veas la diferencia. Entonces ocupamos *blit* porque nos permite copiar todo, sin distinción de píxeles.

Ahora debemos pensar en la posibilidad de que movamos la nave hasta que esta se salga de la pantalla por cualquiera de los cuatro bordes. Con algo de matemática básica llegamos a las siguientes conclusiones.

- Cuando llegue al borde izquierda entonces sucederá que:

Código Fuente 14.

```
x<0
```

- Cuando llegue al borde derecha entonces:

Código Fuente 15.

```
x>SCREEN_W-nave->w
```

- Cuando llegue al borde superior:

Código Fuente 16.

```
y<0
```

- Cuando llegue al borde inferior:

Código Fuente 17.

```
y>SCREEN_H-nave->h
```

Si te resulta complicado recuerda que la nave se representa por un punto en la pantalla (que en este caso esta en la esquina superior izquierda de la imagen) que tiene una coordenada (x, y). Lo que debe hacer finalmente la nave cuando se enfrente a cualquiera de estas situaciones es mantenerse ahí mismo, es decir:

Código Fuente 18.

```
if (x<0) x=x_anterior;
if (x>SCREEN_W-nave->w) x=x_anterior;
if (y<0) y=y_anterior;
if (y>SCREEN_H-nave->h) y=y_anterior;
```

Finalmente todo el procedimiento es así:

Código Fuente 19.

```
void realizar_juego()
{
    BITMAP *nave;
    PALETTE paleta;
    int x,y, x_anterior, y_anterior;
    BITMAP *buffer;

    nave=load_bitmap("../recursos/nave.pcx", paleta);
    set_palette(paleta);
    if (nave==NULL) terminar();
    buffer=create_bitmap(nave->w,nave->h);
    clear(buffer);
    if (buffer==NULL) terminar();
    x=SCREEN_W/2;
    y=SCREEN_H/2;
    while (!key[KEY_ESC])
    {
        if (key[KEY_UP])
            y--;
        if (key[KEY_DOWN])
            y++;
        if (key[KEY_LEFT])
            x--;
        if (key[KEY_RIGHT])
            x++;
        if (x<0) x=x_anterior;
        if (x>SCREEN_W-nave->w)    x=x_anterior;
        if (y<0) y=y_anterior;
        if (y>SCREEN_H-nave->h) y=y_anterior;
        if ((x_anterior!=x) || (y_anterior!=y))
        {
            blit(buffer,screen, 0, 0, x_anterior,y_anterior,buffer->w,buffer-
>h);
            blit(screen,buffer,x,y,0,0,buffer->w,buffer->h);
            draw_sprite(screen, nave, x, y);
        }
        x_anterior=x;
        y_anterior=y;
    }
}
```

Si compilas todo lo que llevamos hasta ahora te darás cuenta que al moverla la nave esta tintinea. ¿Cómo solucionamos eso?. Lo veremos mucho más adelante. Por ahora me parece bien haberte mostrado estos pedazos de códigos para entusiasmarte.

Resumen

En este capítulo vimos paso a paso el inicio de un proyecto, como crearlo en RHIDE y configurar correctamente todas las partes que tendrá. Además me permití mostrar como hacer un código utilizando las bibliotecas Allegro.

Capítulo 3 :
PROGRAMACIÓN ORIENTADA AL OBJETO, EL "NUEVO" PARADIGMA

¿Sabes y entiendes lo que realmente te entrega la programación orientada al objeto?. En este capítulo trataré de convencerte de las inmensas buenas características que te entrega el paradigma de POO.

El progreso de la abstracción

Todos los lenguajes de programación proveen de abstracciones. Se puede discutir que la complejidad del problema que estás tratando de resolver esta relacionado directamente con el tipo y calidad de la abstracción. Por "tipo" yo entiendo, "¿Qué es lo que estoy abstrayendo?", por ejemplo el lenguaje ensamblador es una pequeña abstracción de la máquina subyacente. Hay muchos lenguajes llamados "imperativos" que siguen lo que fue esta abstracción de la máquina (tales como Fortran, Basic y C). Estos lenguajes tienen grandes mejoras sobre el lenguaje ensamblador, pero esa abstracción primaria continúa requiriendo que tu pienses en términos de la estructura del computador más que la estructura del problema que estás tratando de resolver. La programación debe establecer una asociación entre el modelo de la máquina (en el "espacio de la solución", la modelación del problema, como un computador por ejemplo) y el modelo del problema que esta actualmente resolviéndose (en el "espacio del problema", donde el problema existe). El esfuerzo requerido para realizar este mapeo y el hecho que esto es intrínseco al lenguaje de programación, produce programas que son difíciles de escribir y caros de mantener.

La aproximación orientada al objeto va un paso más allá proveyendo herramientas para que el programador represente los elementos del espacio del problema. Esta representación es generalmente suficiente para que el programador no esté limitado para ningún tipo particular de problema. Nos referimos a los elementos en el espacio del problema y las representaciones en el espacio de la solución como "objetos" (Por supuesto que también necesitamos otros objetos que no tengan analogías en el espacio del problema). La idea es que el programa tenga permitido adaptarse él mismo al idioma del problema agregando nuevos tipos de objetos, así cuando tú leas el código describiendo la solución, estarás leyendo la solución que también expresa el problema. Esto es más flexible y poderoso que la abstracción del lenguaje que teníamos antes. Así, la POO te permite describir el problema en términos del problema más que en términos del computador en donde la solución se ejecutará. Sin embargo, allí todavía hay una conexión con un computador. Cada objeto se puede ver como un pequeño computador, tiene un estado y tiene operaciones que tu puedes saber y realizar. Sin embargo, no parece una mala analogía comparar los objetos con el mundo real; ellos tienen características y comportamientos.

- **Todo es un objeto.** Piensa en un objeto como una variable imaginaria; ella guarda datos, pero tu puedes "hacer consultas" a ese objeto, realizando estas operaciones sobre si mismo cuando le preguntas. Puedes tomar cualquier componente conceptual del problema que estás tratando de resolver (perros, construcciones, servicios, naves, tanques, etc.) y representarlos como un objeto en el programa.

- **Un programa es un conjunto de objetos diciéndole a otro qué hacer con el mensaje que le enviaste.** Para hacer una petición a un objeto, tu "envías un mensaje" a ese objeto. Más concretamente, puedes pensar un mensaje como un requerimiento para llamar a una función que pertenece a un objeto particular.
- **Cada objeto tiene su propia memoria aparte de otro objeto**. Desde otro punto de vista, tú creas un nuevo tipo de objeto para hacer un paquete que contiene un objeto existente. Así, tu puedes construir complejidad en un programa mientras escondiéndolo detrás de objetos simples.
- **Todos los objetos tienen un tipo**. Cada objeto es una *instancia de* una *clase*, en la cual "clase" es sinónimo de tipo. La más importante característica que se puede distinguir de una clase es "¿Qué mensaje puedo enviarle?".
- **Todos los objetos de un tipo particular pueden recibir el mismo mensaje**. Por ejemplo, un objeto de tipo círculo es también un objeto de tipo "figura", un circulo garantiza que puede recibir mensajes que una figura acepta. Esto significa que puedes escribir código que le hable a una figura y automáticamente enganche cualquier cosa que se ajuste a la descripción de figura. Esta *sustitubilidad* es uno de los más poderosos conceptos en la POO.

Características de la POO

Un objeto tiene una interfaz.

La interfaz establece *que* solicitudes se pueden hacer a un objeto en particular. Sin embargo, debe haber un código en alguna parte que satisfaga esa solicitud. Esto, junto con los datos escondidos, comprenden la *implementación*. Desde un punto de vista de programación procedural, esto no es tan complicado. Un tipo tiene una función asociada para cada posible solicitud y cuando tu hacer una solicitud particular a un objeto, esa función es llamada. Este proceso es usualmente resumido diciendo que "envías un mensaje" (haces una solicitud) a un objeto y el objeto entiende que hacer con ese mensaje (ejecuta el código).

La implementación está oculta

Antes de continuar haremos una distinción entre el *creador de la clase* y el *programador cliente*. El primero es el que crea el nuevo tipo de dato(la clase) y el segundo es el que consume la clase, quien usa los tipos de datos en sus aplicaciones. La meta de un programador cliente es coleccionar una caja de herramientas llenas de clases para usarlas en el rápido desarrollo de aplicaciones. La meta del creador de la clase es construirlas exponiendo solo lo que es necesario que el programador cliente sepa, manteniendo todo lo demás escondido. ¿Porqué?. Porque si esta escondido el programador cliente no puede usar eso que esta escondido lo cual significa que el creador de la clase puede cambiar la porción escondida en el futuro sin que le preocupe el alguien más ese cambio. La porción escondida representa usualmente lo más delicado de un objeto que podría fácilmente ser corrompido por un programador cliente descuidado o sin información, así la implementación oculta reduce los errores.

En cualquier relación es importante tener límites que son respetados por todas las partes involucradas. Cuando creas una biblioteca, estableces una relación con el programador cliente, quien es también un programador, pero que está combinando en una aplicación tus bibliotecas posiblemente para construir una más grande.

Si todos los miembros de una clase están disponibles para cualquiera, entonces el programador cliente puede hacer cualquier cosa con esa clase y no habría forma de establecer reglas. Aún cuando podrías realmente preferir que el programador cliente no manipule directamente alguno de los miembros de tu clase, sin control de acceso no hay forma de impedirlo. Todo está desnudo ante el mundo.

De esta forma, la primera razón para tener control de acceso es mantener al programador cliente con las manos fuera de la porción que no puede tocar, partes que son necesariamente para la maquinación interna del tipo de dato pero no de la interfaz que el usuario usará para resolver un problema particular. Esto es actualmente un servicio para los usuarios porque ellos pueden fácilmente ver que es importante para ellos y que pueden ignorar.

La segunda razón para controlar el acceso es permitir al diseñador de las bibliotecas cambiar el trabajo interno de la clase sin preocupase sobre como esto afectaría al programador cliente. Por ejemplo, puedes implementar una clase particular de una manera simple y luego descubrir que necesitas rescribirlo para hacer que se ejecute más rápido. Si la interfaz y la implementación están claramente separadas y protegidas puedes hacerlo fácilmente y requerir solo de un reenlace por el usuario.

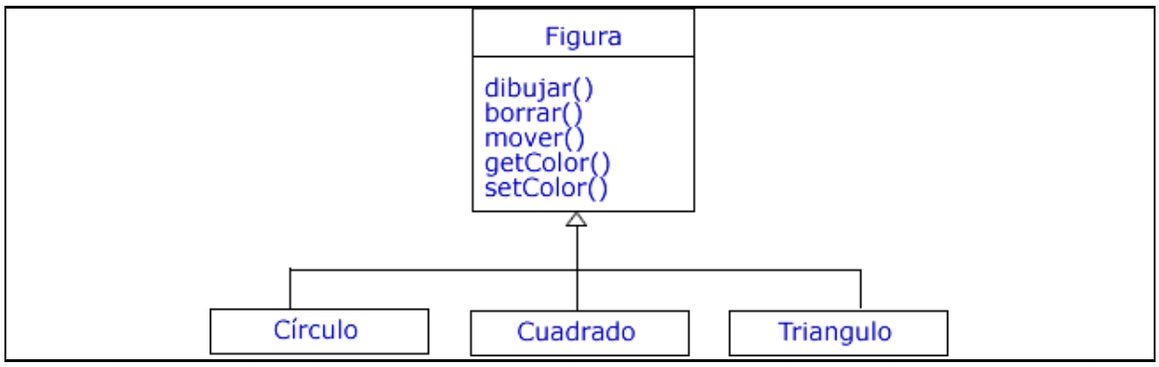
C++ usa tres palabras claves explícitas para establecer los límites en una clase: ***public, private y protected.*** Si alguien trata de acceder a un miembro ***private***, él obtiene un error en tiempo de compilación. ***Protected*** actual como ***private***, con la excepción de que una clase heredada tiene acceso a miembros ***protected***, pero no a miembros ***privados***. La herencia será introducida luego.

Ocupando nuevamente la implementación

Una vez que la clase está creada y testeada, esto podría (idealmente) representar una unidad de código útil. Esto significa que re-usar el código nos puede ayudar de alguna forma, este pedazo de código toma experiencia y comprensión produciendo un buen diseño. El código re-usado es una de las más grandes ventajas que los LOO proveen.

Por ejemplo, una maquina recicladora de basura ordena pedazos de basura. El tipo base es “basura” y cada pedazo de basura tiene un alto, un valor y así sucesivamente, y estos pedazos pueden ser rotos, fundidos o descompuestos. Des allí, tipos más específicos de basura pueden ser derivados y podrían también tener características (una botella tiene un color) o comportamientos adicionales (el aluminio puede ser roto o un imán puede estar magnetizado). Usando la herencia, puedes construir tipos de jerarquía que expresan el problema que estas tratando de resolver en términos de sus tipos.

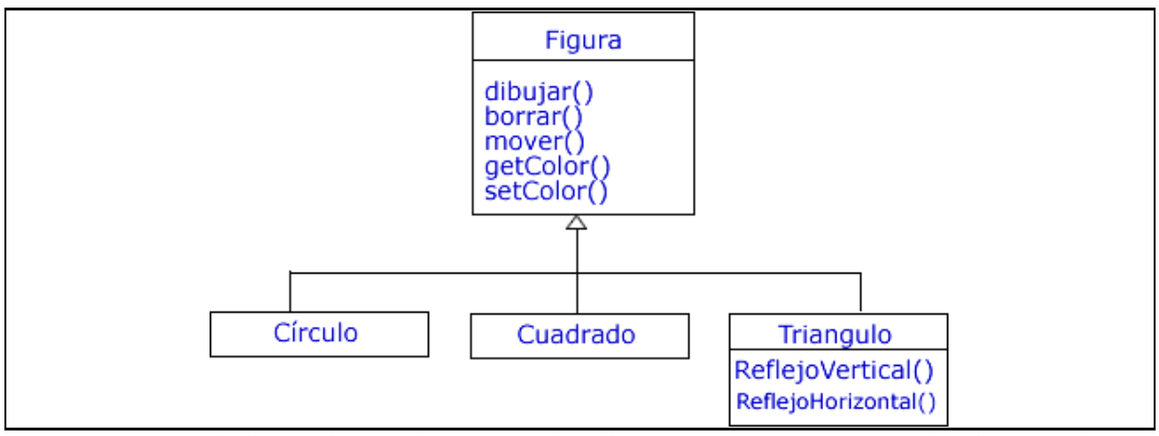
Un segundo ejemplo es el clásico de la “figura”, quizá usado en diseño de sistemas asistido por computador o simulación de juegos. El tipo base “figura” y cada figura tiene un tamaño, un color, una posición y así sucesivamente. Cada figura puede ser dibujada, borrada, movida, coloreada, etc. Desde aquí, un tipo específico de figura puede ser derivado (heredado): circulo, cuadrado, triangulo, y así, cada uno de los cuales puede tener características y comportamientos adicionales. Ciertos tipos de figuras pueden ser rotados, por ejemplo. Algunos comportamientos pueden ser diferentes, tal como cuando quieres calcular el área de una figura. El tipo de jerarquía contiene las similitudes y diferencias entre las figuras.



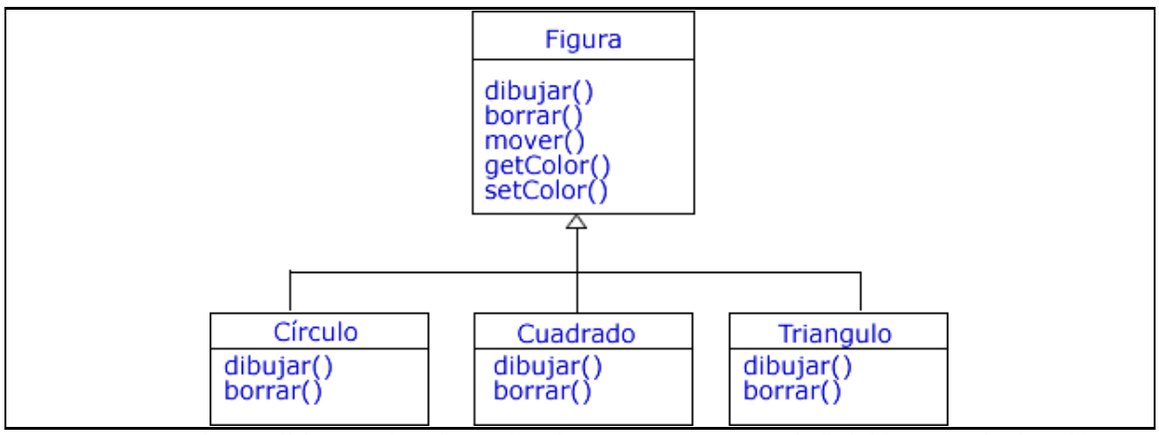
Cuando tu heredas creas un nuevo tipo. Este nuevo tipo contiene no solo todo los miembros del tipo existente (aunque los *private* están escondidos e inaccesibles), pero más importante es duplicar la interfaz de la clase base. Esto es, todos los mensajes que puedes enviar al objeto de la clase base puedes también enviárselo al objeto de la clase derivada. Sabemos el tipo de una clase por los mensajes que podemos enviarle, esto significa que una clase derivada *es del mismo tipo que la clase base*. En un ejemplo anterior, “un círculo es una figura”. Este tipo de equivalencia vía herencia es una de las entradas fundamentales para entender el significado de la programación orientada al objeto.

Desde que la clase base y la derivada tiene la misma interfaz, debe haber alguna implementación para acceder a esa clase. Esto es, debe haber algún código que ejecutar cuando un objeto recibe un mensaje particular. Si tu simplemente heredas una clase y no haces nada más, los métodos desde la interfaz de la clase base están exactamente en la clase derivada. Esto significa que los objeto derivados no solo tienen el mismo tipo, ellos también tienen los mismo comportamientos, lo cual no es particularmente interesante.

Tienes dos formas para diferenciar tus nuevas clases derivadas de la clase base original. La primera es fácil: simplemente agregas marcas a las nuevas funciones de la clase derivada. Estas nuevas funciones no son parte de la interfaz de la clase base. Esto significa que la clase base simplemente no hace tanto como lo que quieres, así agregas más funciones. Este simple y primitivo uso de la herencia es, a veces, la solución perfecta para tu problema. Sin embargo, deberías mirar más de cerca la posibilidad de que tu clase base necesite también esas funciones. Este proceso de descubrimiento e iteración de tu diseño sucede regularmente en la POO.



Aunque la herencia puede a veces implicar que agregas nuevas funciones a la interfaz, no es necesariamente simpre verdadero. La segunda y más importante forma de diferenciar tus nuevas clases es *cambiando* los comportamientos de una función de la clase base que existe. Esto es referido como *sobrescribir* esa función.

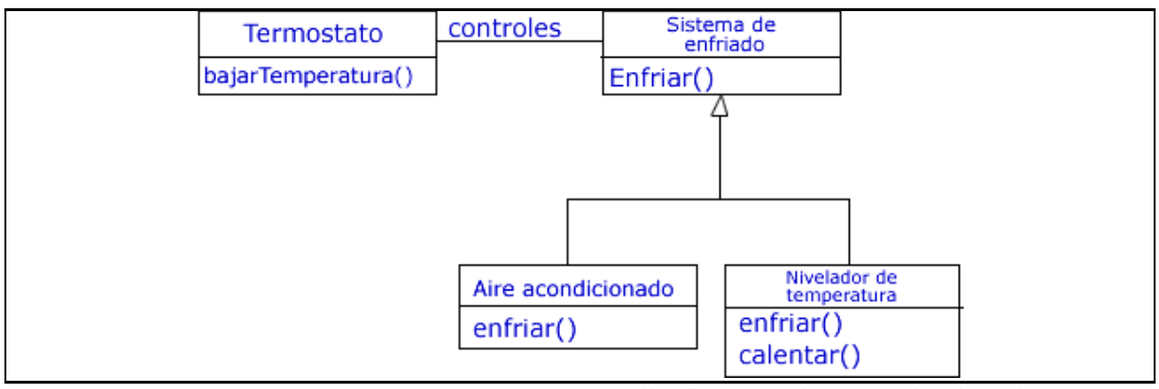


Para sobrescribir una función, simplemente crea una definición para la función en la clase derivada. Estarás diciendo, “Estoy usando la misma interfaz de la función aquí, pero espero hacer algo diferente para mi nuevo tipo”.

Relación es-un vs. es-como-un

Hay cierto debate que puede ocurrir acerca de la herencia: ¿Debe la herencia sobrescribir *solo* funciones de la clase base (y no agregar nuevas funciones miembros que no están en la clase base)? Esto podría significar que el tipo derivado es *exactamente* el mismo tipo que la clase base desde que ésta tiene exactamente la misma interfaz. Como resultado, puedes sustituir exactamente un objeto de la clase derivada por un objeto de la clase base. Esto se puede pensar como una *sustitución pura*, y es referido a veces como *principio de sustitución*. En un sentido, esta es la forma ideal de tratar la herencia. A veces nos referiremos a la relación entre la clase base y las clases derivadas, en este caso, como una relación *es-un*, porque tu dirás “un circulo *es una* figura.” Una prueba para la herencia es determinar si puedes enunciar la relación *es un* respecto de las clases, y que tenga esto sentido.

Hay veces en que debes agregar nuevos elementos a la interfaz de un tipo derivado, de esta forma extiendes la interfaz y creas un nuevo tipo. El nuevo tipo puede continuar siendo substituido por el tipo base, pero la sustitución no es perfecta porque tus nuevas funciones no son accesibles desde el tipo base. Esto puede ser descrito como una relación *es-como-un*; El nuevo tipo tuvo la interfaz del antiguo tipo pero este también contiene otras funciones, no puedes realmente decir que es exactamente lo mismo. Por ejemplo, consideremos un sistema de aire acondicionado. Supone que tu casa está cableado con todos los controles para el enfriador; es decir, tiene una interfaz que te permite controlar el enfriador. Imagina que el sistema de aire acondicionado se avería y lo reemplazas con un nivelador de temperatura, el cual puede calentar y enfriar. El nivelador de temperatura *es-como-un* sistema de aire acondicionado, pero puede hacer más. Porque el sistema de control de tu casa está diseñado solo para controlar el enfriamiento, esta restringido a comunicarse con solo la parte enfriadora del nuevo objeto. La interfaz del nuevo objeto está extendida, y el sistema existente no entiende sobre nada excepto la interfaz original.



Por supuesto que una vez que veas este diseño será claro que la clase base “sistema de enfriado” no es suficientemente general, y debería ser renombrada como “sistema controlador de temperatura”, así este también puede incluir el calentador – en donde funcionaría el principio de sustitución. Sin embargo, el diagrama de arriba es un ejemplo de lo que puede suceder en el diseño y en el mundo real.

Resumen

En este capítulo aprendimos algunas de las formas más comunes de pensar en el sentido de los objetos. Ver todo como un objeto que pertenece a una clase y que tiene comportamientos y datos específicos y separados de otros objetos nos permite encapsular mucho mejor nuestro problema al compararlo con el enfoque procedural de los antiguos lenguajes de programación.

También se vieron algunos ejemplos de la herencia y de las relaciones *es un* y *es como un*.

Capítulo 4 :
ANÁLISIS Y DISEÑO ORIENTADO A OBJETOS

El paradigma orientado al objeto es una nueva y diferente forma de pensar en la programación y muchos amigos han tenido problemas al hacer proyectos con esta metodología. Una vez que sabes todo lo que supone un objeto y aprendes a pensar más en el estilo orientado a objeto, puedes empezar a crear “buenos” diseños que aprovechan todos los beneficios que la POO ofrece.

Un *método* (o llamado *metodología*) es un conjunto de procesos y heurísticas usadas para reducir la complejidad de un problema programable. Muchos métodos de la POO han sido formulados desde la aparición de este paradigma. Esta sección orientará para que trates de terminar (porque muchos proyectos no terminan) cuando usas un método.

Especialmente en la POO la metodología es un campo muy experimental, y es importante entender qué problema está tratando de resolver el método antes de considerar adoptar uno. Esto es particularmente cierto en C++, en el cual el lenguaje de programación es reduce la complejidad (comparado con C) envuelta en expresar un programa. Esto puede en efecto aliviar la necesidad de metodología cada vez más complejas.

Es también importante realzar que el término “metodología” es a veces demasiado grande y prometedor. Lo que sea que tu uses cuando diseñas y escribes un programa ahora, es un método. Este podría ser tu propio método y podrías no ser consciente de lo que hace, pero es lo que sucede cuando estás creando. Si no es un proceso efectivo, podría necesitar una pequeña afinación para trabajar con C++. Si no estás satisfecho con la productividad y la forma en que tus programas resultan, esperarás considerar una metodología formal o elegir partes de algunos métodos formales.

Mientras estás en el proceso de desarrollo, la cosa más importante es esto: no te sientas perdido. Es fácil caer en eso. Muchos de los métodos de análisis y diseño son hechos para problemas muy generales. Recuerda que la mayoría de los proyectos no se ajustan a esa categoría, de esta forma puedes tener un diseño y análisis con un pequeño subconjunto del método recomendado.

Es fácil quedarse pegado, caer dentro del “parálisis en el análisis”, donde sientes que no puedes avanzar porque no sellaste todos los pequeños detalles de la etapa actual. Recuerda, no importa cuanto análisis hagas, hay algunas cosas sobre el sistema que no se relevarán hasta el proceso de diseño e incluso algunas cosas no serán reveladas hasta escribir el código, o hasta que el programa está efectivamente ejecutándose. A causa de esto, es crucial avanzar rápidamente a través del análisis y el diseño, e implementar una prueba del sistema propuesto.

Si estás buscando una metodología que contenga sendos detalles y que sugiera muchos pasos y documentos, continúa siendo dificultoso el saber cuando detenerse. Mantén en mente que estás tratando de descubrir:

1. ¿Cuáles son los objetos? (¿Cómo dividir tu proyecto en sus partes componentes?)
2. ¿Cuáles son las interfases? (¿Qué mensajes necesitas tener disponibles para enviar a cada objeto?)

Si tu te encuentras en estos momentos, con nada más que objetos y sus interfases, puedes escribir un programa. Por varias razones necesitarás de más descripciones y documentos que eso.

El proceso puede ser abordado en cinco fases y una fase 0 que solo es el compromiso inicial para usar algún tipo de estructura.

Fase 0: Construir un plan

Debes primero decidir que pasos darás en tu proceso. Esto suena simple (de hecho, *todo* esto suena simple) y todavía hay gente que a veces no tiene ningún plan antes de empezar a escribir el código fuente. Si tu plan es “sentarse y empezar a escribir”, esta bien. (Muchas veces esto es apropiado cuando tu problema lo entiendes a cabalidad). A lo menos debes estar de acuerdo en que esto es un plan.

Puedes también decidir en esta fase que estructuras adicionales son necesarias para el proceso, pero no todo. Hay cosas comprensibles, como por ejemplo, que a algunos programadores les guste trabajar en “el modo vacación”, donde no hay una estructura impuesta en el proceso de desarrollo de su trabajo; “esto terminará cuando termine”. Puede ser atractivo por algún momento, pero he encontrado que es bueno establecer ciertas metas dentro del proyecto en vez de pensar en una gran meta final llamada “terminar el proyecto”. En suma, dividir el proyecto en piezas más pequeñas y hacer que parezcan menos amenazantes (además, las metas ofrecen más oportunidades para celebrar).

Cuando empecé a estudiar una estructura de una historia para un libro (cuando escribí una novela), me resistí inicialmente a la idea de “estructura”, porque sentía que cuando escribo simplemente dejo fluir mis pensamientos a través de las páginas. Pero después de realizado esto, cuando escribí sobre los computadores, la estructura estaba lo suficientemente clara, así que no pensaba mucho en eso. Pero continúo estructurando mi trabajo, a pesar de que esté inconscientemente en mi cabeza, aún si piensas que tu plan es solo empezar a programar.

Establecer una misión

Cualquier sistema que construyas, no importa cuan complicado, tiene un propósito fundamental, una necesidad básica que será satisfecha. Si puedes mirar detrás de la interfaz de usuario, el hardware o los detalles específicos del sistema, el código fuente de los algoritmos y los problemas de eficiencia, eventualmente encontrarías el corazón de la existencia del sistema. Como es llamado en Hollywood, el “concepto superior”, la descripción en una o dos oraciones. Esta pura descripción es un punto de partida.

El concepto superior es muy importante porque le da el rumbo a tu proyecto; es el establecimiento de una misión. Quizá no tendrás este concepto claro la primer vez (más tarde en tu proyecto puede ser todo muy claro), pero mantente intentando hasta que sientas que lo encontraste. Por ejemplo, en un sistema de control de tráfico aéreo puedes empezar con un concepto superior enfocado sobre el sistema que estas construyendo: “El programa de la torre administra las pistas de los aviones”. Pero considera lo que sucede cuando disminuye el sistema a un pequeño aeropuerto; quizá hay solo un humano controlando o quizá nadie. Un modelo más práctico que no se preocupe de la solución que estás creando tanto como se describe en el problema: “Las aviones arriban, descargan, recargan y emprenden el viaje”, ese podría ser un mejor concepto superior.

Fase 1: ¿Qué estamos haciendo?

En la generación anterior del diseño de un programa (llamado *diseño procedural*), lo que “estamos haciendo” es llamado “creando el análisis de los requerimientos y especificación del sistema.” Esto, por supuesto, fue hecho para perder; documentos con nombres que intimidan. Sin embargo, su intención fue buena. El análisis de los requerimientos decía “Haz una lista de directivas que ocuparemos para saber cuando el trabajo este listo y el consumidor esté satisfecho”. La especificación del sistema decía “Aquí hay una descripción de *que* hará el programa (no *como*) para satisfacer los requerimientos.” El análisis de los requerimientos es realmente un contrato entre tu y el consumidor (aun si el consumidor trabaja en tu compañía o es algún otro objeto o sistema). La especificación del sistema es una exploración de alto nivel dentro del problema y en algún sentido es un descubrimiento de que tanto este puede hacer y cuanto tiempo tomará en hacerlo. Desde que ambas cosas requieren consenso entre la gente (y porque esas personas tendrán poco tiempo), pienso que es mejor mantenerse tan abierto como sea posible para ahorrar tiempo. Quizá lo más importante es empezar un proyecto con un montón de entusiasmo.

Es necesario mantener enfocado en nuestro corazón lo que estás tratando de lograr en esta fase: determinar lo que el sistema se supone que hace. La herramienta más valiosa para esto son los “casos de uso.” Los casos de uso identifican características claves en el sistema que revelarán algunas de las clases fundamentales que usarás. Se encuentran esencialmente respuestas descriptivas a preguntas como:

- ¿Quién usará el sistema?
- ¿Qué pueden esos actores hacer con el sistema?
- ¿Cómo lo hace este actor para hacer esas cosas?
- ¿Los mismos actores tienen un objetivo diferente? (se revelan variaciones)
- ¿Qué problemas pueden pasar mientras haces cosas con el sistema? (se revelan excepciones)

Fase 2: ¿Cómo construiremos eso?

En esta fase debes aparecer con un diseño que describa lo que las clases son y como interactuarán. Una técnica excelente para determinar las clases e interacciones son las tarjetas de *Clase-Responsabilidad-Colaboración*(CRC). Parte del valor de esta herramienta es que es de baja tecnología: comienzas con una serie de tarjetas blancas de 3” por 5” y escribes sobre ellas. Cada tarjeta representa una clase simple y sobre la ella escribes:

1. El nombre de la clase. Es importante que el nombre capture la esencia de lo que la clase hace.
2. Las “responsabilidades” de las clases: qué deberían hacer. Pueden ser listadas solo con los nombres de las funciones miembro (puesto que los nombre deberían ser descriptivos en un buen diseño), pero no prohíbe otras anotaciones. Si necesitas empezar el proceso de escribir las clases, piensa en esto: ¿Qué objetos podrías querer que aparecieran mágicamente para resolver tu problema?
3. Las “colaboraciones” de las clases: ¿Qué otras clases interactúan con la actual? “Interactuar” es un término dicho intencionalmente; esto podría significar agregar un método o simplemente otro objeto existente que nos provea servicios para un objeto de la clase. Por ejemplo, si estas creando la clase *fuego artificial*, ¿quién irá a observarla?, ¿Un químico o un espectador? El primero querría saber que químicos van dentro de la construcción, y el último respondería que colores y figuras se forman al explotar.

Podrías pensar que las tarjetas deberían ser un poco más grandes para toda la información que quieres poner ahí, pero ellas son intencionalmente pequeñas, no solo para mantener tus clases pequeñas sino para mantener todos los detalles alejados del diseño. Si no puedes ajustar todo en una sola tarjeta, tu diseño es demasiado complejo (o bien, estás siendo demasiado detallista o deberías crear más clases). La clase ideal es entendida a primera vista. La idea de las tarjetas CRC es asistirte al empezar con el primer diseño para así tener una idea global para luego refinar tu diseño.

Uno de los grandes beneficios de las tarjetas CRC es la comunicación. Es realmente en tiempo real, en un grupo, sin computadores. Cada persona toma responsabilidades de muchas clases (las cuales empiezan sin nombre u otra información). Realizas una simulación real resolviendo un escenario a la vez, decidiendo qué mensajes son enviados a los demás objetos satisfaciendo de esta forma cada uno de esos escenarios. Como tu estarás en este proceso, descubrirás las clases que necesitas, con sus responsabilidades y colaboradores, y llenarás tu tarjeta con esa información. Cuando te revises todos los casos de uso, podrías tener un completo primer acercamiento de tu diseño.

Cinco etapas en el diseño de objetos

El tiempo de vida del diseño de un objeto no está limitado a la época en que escribiste el programa. Los objetos, también tienen sus patrones que emergen a través del conocimiento, el uso y el re-uso.

1. **Descubrimiento del objeto.** Esta etapa sucede durante el análisis inicial de un programa. Los objetos pueden ser descubiertos observando factores externos y límites, duplicación de elementos en el sistema y las pequeñas unidades conceptuales. Algunos objetos son obvios si los tienes listos en una biblioteca de clases. Comúnmente las clases bases y las herencias pueden aparecer al instante o después en el proceso de diseño.
2. **Ensamblaje del objeto.** Como estas construyendo un objeto descubrirás la necesidad de un nuevo miembro que no aparece durante el descubrimiento del objeto. La necesidad interna de un objeto puede requerir otras clases que soporten esas necesidades.
3. **Construcción del sistema.** De nuevo, otros requerimientos para un objeto pueden aparecer en esta tardía etapa. La necesidad por comunicar e interconectarte con otros objetos en el sistema puede cambiar las necesidades de tus clases o requerir nuevas clases. Por ejemplo, puedes descubrir la necesidad por clases que te ayudan, tales como listas enlazadas, y que contienen información pequeña o sin estado y simplemente ayudan a otras clases.
4. **Extensión del sistema.** Al agregar nuevas características a tu sistema, puedes descubrir que el anterior diseño no soporta extensiones simples del sistema. Con esa nueva información, puedes re-estructurar las partes del sistema, posiblemente agregando nuevas clases o derivando.
5. **Reutilización del objeto.** Este es la prueba real más dura para un objeto. Si alguien trata de re-usarlo en una situación enteramente nueva, probablemente descubrirá algunos defectos. Al cambiar la clase para adaptarla a nuevos programas, los principios generales de la clase serán claros y finalmente tendrás un tipo realmente reusable. Sin embargo, no anticiparás que más objetos en el diseño del sistema son reusables – esto es perfectamente aceptable para un diseño de objetos realizado específicamente para un solo sistema. Los tipos reusables tienen a ser menos comunes, y ellos deben resolver problemas más generales para ser más reusables.

Directrices para el desarrollo de objetos

Estas etapas siguieren algunas directrices cuando piensas en el desarrollo de tus clases:

1. Deja a un problema específico generar una clase, luego deja a la clase crecer y madurar durante la solución de otros problemas.
2. Recuerda: descubrir las clases que necesitas (y sus interfases) es el mayor trabajo que debes realizar en el diseño de un sistema. Si ya tienes estas clases, este podría ser un proyecto fácil.
3. No te fuerces a ti mismo para saber todo desde un comienzo; aprende a medida que avanzas. Esto sucederá de todas formas en algún momento.
4. Parte programando; toma algún trabajo para así poder aprobar o desaprobar tu diseño. No temas si terminas con un tipo de código al estilo procedural. Las clases malas no se vuelven mágicamente “buenas clases”.

5. Siempre mantén todo lo menos complicado posible. Objetos pequeños y claros con una utilidad obvia más que grandes y complicadas interfases. Cuando los puntos de decisión aparecen, usa esta aproximación: considera las alternativas y selecciona la que es más simple, siempre las clases más simples serán las mejores. Empieza simple y poco detallado, y podrás expandir las interfases de las clases cuando las entiendas mejor, pero como el tiempo corre, es difícil remover elementos desde una clase.

Fase 3: Construyendo el núcleo.

Tu meta es buscar el corazón de la arquitectura de tu sistema que necesita ser implementada de tal forma de generar un sistema que "corra", no importando cuan incompleto este el sistema inicialmente. Puedes estas creando un marco de trabajo (*framework*) para todo tu sistema, que se ejecute y realice iteraciones. Probablemente también descubrirás cambios y mejoras que puedes hacer a tu arquitectura original – cosas que podrían no haber aprendido si no implementabas el sistema.

Asegúrate de que tu prueba verifica los requerimientos y casos de uso. Cuando el núcleo del sistema este estable, estás listo para seguir adelante y agregar más funcionalidad.

Fase 4: Iterando los casos de uso.

Una vez que el marco de trabajo núcleo está ejecutándose, cada conjunto de característica que agregas es un pequeño proyecto por si mismo. Agregas características durante la *iteración*, un periodo razonablemente corto de desarrollo.

¿Cuán grande es una iteración?. Idealmente, cada iteración toma una o dos semanas (esto puede variar de acuerdo al lenguaje de implementación). Al final de ese periodo tienes un sistema integrado y probado con más funcionalidad que antes. Pero lo que es particularmente importante para la iteración: ya agregaste un simple caso de uso. No solo esto hace darte una mejor idea del alcance de tu caso de uso, sino que también da más validación a la idea del caso de uso desde que el concepto no es descartado después del análisis y diseño, en lugar de eso es una unidad fundamental de desarrollo a lo largo del proceso de construcción del software.

Tu terminas de iterar cuando funcionalmente logras apuntar o llegar a una fecha de tope y el consumidor puede estar satisfecho con la versión actual. (Recuerda, el software es un negocio de suscripción.) Ya que el proceso es iterativo, tienes muchas oportunidades para enviar un producto que todavía puede seguir extendiéndose; los proyectos de código abierto trabajan exclusivamente en un ambiente iterativo y de alta retroalimentación en el cual es preciso hacer esto correctamente.

Un proceso de desarrollo iterativo es válido por muchas razones. Puedes advertir y resolver riesgos tempranamente, los consumidores tendrán una amplia oportunidad de cambiar sus métodos, la satisfacción del programador es alta y el proyecto puede ser guiado con más precisión.

Fase 5: Evolución

Este es el punto en el ciclo de desarrollo que es tradicionalmente llamado "mantención", un término que puede significar todo desde "poner a trabajar el programa en la forma en que fue ideado inicialmente" hasta "agregar características que los consumidores olvidaron mencionar" o hasta cosas más tradicionales como "reparar errores". Quizá halla un termino mejor para describir esto.

Usaré el término *evolución*. Esto es, "No podrás tenerlo todo la primera vez, así tendrás la latitud para aprender, volver atrás y realizar cambios." Puedes necesitar hacer un montón de cambios y así, aprender y entender más profundamente el problema.

Quizá la más importante cosa que debes recordar es que por defecto – realmente por definición – si modificas una clase, luego sus superclases y subclases continúan funcionando. Necesitas hacer modificaciones sin miedo (especialmente si tienes un conjunto dentro de una unidad de prueba que verifica el correcto funcionamiento de tus modificaciones). Las modificaciones no necesariamente destruirán el programa y cualquier cambio en el resultado estará limitado a la subclase y/o a colaboradores específicos de la clase que cambiaste.

Resumen

En este capítulo se abordaron temas importantísimos a la hora de hacer el análisis y diseño de un software de calidad basado en la orientación a objetos.

Se dan pasos específicos y de mucha ayuda a la hora de empezar un proyecto: como empezar y como termina.

Algunas herramientas conocidas y recomendadas son los *casos de uso* y las *tarjetas CRC* .

Por último, este capítulo nos hace recordar las estadísticas: más del 70% de los proyectos nunca se terminan.

Capítulo 5 : ESTABLECIENDO LOS REQUERIMIENTOS Y ANÁLISIS INICIALES DEL JUEGO

En esta sección intentaremos definir los requerimientos del sistema, en nuestro caso un juego de video, tal como si se tratase de un cliente que nos pide esta tarea. Por supuesto que la ventaja principal de ser nosotros mismos nuestro cliente es que conocemos todo con mucho detalle, una ventaja que un programador siempre desearía tener. Solo quiero dejar en claro que todo esto es preliminar y se ira aumentando la complejidad en un proceso iterativo de desarrollo.

Lo que se denomina como requerimientos del juego es como el año cero en el proceso de desarrollo del software. Aquí se establece qué esperamos de nuestro juego, conocido como "¡quiero que en mi juego mi nave este eludiendo proyectiles como loco!", eso ya es un requerimiento... y muy complejo.

En esta etapa de especificación de requerimientos se puede producir muchas veces en un juego. Por ejemplo, en Street Fighter 1 el requisito era "hacer un juego de peleas y vagar por el mundo con Ryu en busca de contrincantes", luego en la última versión quizá se este pensando en "seleccionar de entre *n* jugadores y tener como contrincantes *m* peleadores que dependan del personaje elegido, desarrollándose la acción en un entorno tridimensional... bla, bla, bla". Como pueden ver el juego sigue siendo el mismo y a lo largo de los años se han definido distintos requerimientos o *parchando* errores haciendo aumentar el número de la versión y de la revisión respectivamente y, porque no, la entretención.

Requerimientos del sistema (Fase 0 y Fase 1)

Ahora nos pondremos en el caso de una compañía que pide que le hagan un video juego, ella exige lo que quiere y el desarrollador le entrega el producto dentro de un plazo razonable.

Comencemos:

"El nombre del juego es '*Alfa Destructio, Interventor*' y *Alfa* hace alusión a la versión. Si vemos que el juego a quedado bien entonces haremos la versión Beta, es decir, '*Beta Destructio, Interventor'.* Desde ahora en adelante abreviaré *'Alfa Destructio'* como ADES.

ADES será un juego de naves del género conocido popularmente como '*shoot'em up'*. Se podrá jugar de hasta dos personas.

Cada persona controlará una nave en la pantalla. La nave que controlamos es del modelo denominado *Destructio* (ficción) y tiene energía , distintos tipos de arma y cada arma con distintos niveles de poder.

Estas naves deben ir avanzando a través de etapas enfrentándose a distintas clases de unidades enemigas tales como tanques, naves, lanchas, helicópteros, etc.

Las etapas estarán creadas con un *editor de etapas*. Este editor de etapas establecerá dónde y cuándo aparecerá una nave enemiga y qué recorrido hará por la pantalla. También en él se crearán los elementos de juego, es decir, unidades enemigas, unidades aliadas, proyectiles, ítems de energía, ítems de poder, etc.”

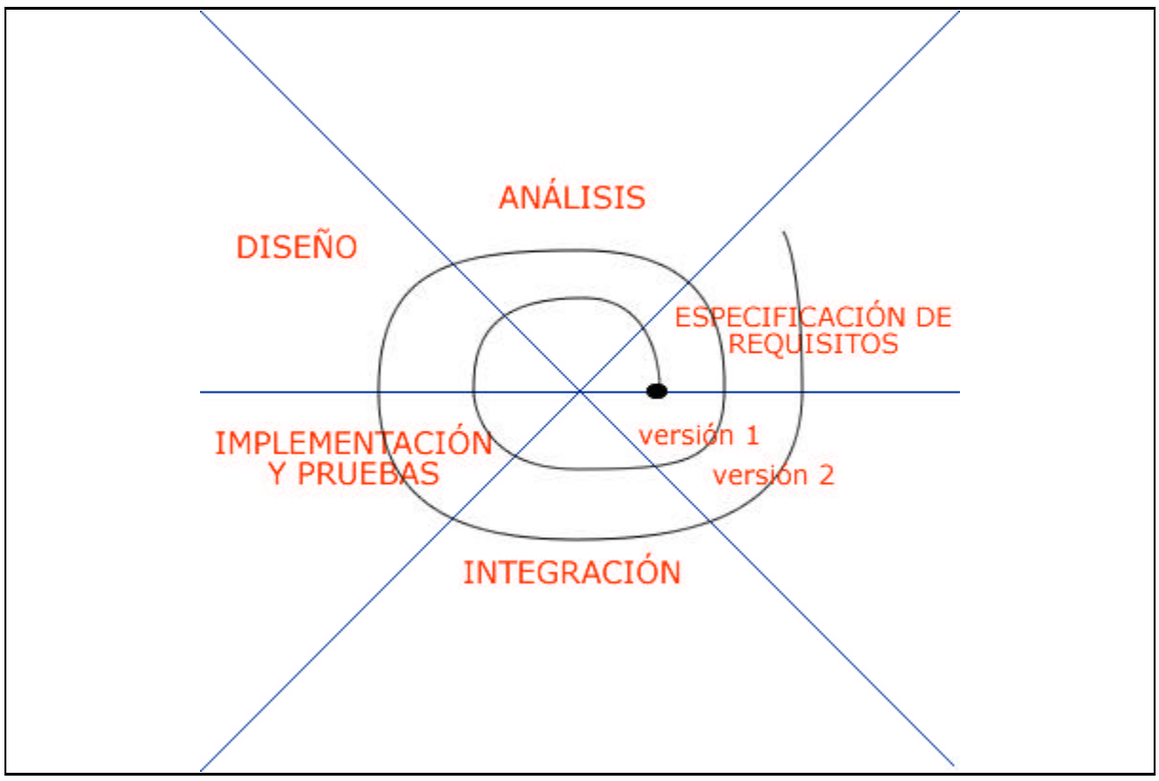
Ahora que ya se han especificado los requerimientos iniciales del sistema en este pequeño documento debemos empezar el trabajo del análisis, luego el diseño, luego la implementación y las pruebas; la integración y el lanzamiento de la esperada primera versión.

Por supuesto que para cumplir con todos estos requisitos debemos crear casi un juego completo. Entonces, ¿Qué hacemos?. Tal como se explica en el capítulo anterior debemos crear pequeñas metas para cumplir el objetivo final del programa que cumple con todos estos requisitos iniciales, es decir, debemos crear sub requisitos, ir iterando a través del tiempo e integrando nuevas características a nuestro programa.

Por ejemplo, en una primera realización podríamos tener la pantalla en modo gráfico y nada más. Eso ya sería un avance si esa sección del juego funciona perfectamente. No habría porque seguir preocupándose de la sección “inicialización del modo gráfico”.

Entonces lo que tendrá esta primera versión del juego ADES serán loos requerimientos anteriormente definidos: la *versión 1.0* . El decimal se incrementará a medida que avancemos hasta el entero. Definiremos de algún modo qué secciones del juego, al ser desarrolladas, permitirán avanzar el decimal algunas unidades. Por ejemplo, si desarrollamos el sistema de controles, avanzaremos el decimal en *0.2*

Para que les quede claro como es el proceso de desarrollo de un programa, les mostraré este diagrama:



Siempre para desarrollar una versión se parte desde la especificación de requisitos, luego se produce el análisis, el diseño, implementación, integración y sale a luz una nueva versión del software.

Haciendo un resumen y comparando con los pasos sugeridos en el capítulo anterior, podemos ver que la fase denominada 0 es en la cual hacemos un plan de trabajo individual o en un grupo de personas. Mi plan es “terminar el juego (versión 1) junto con este curso antes de 3 meses trabajando semanalmente en conjunto con otras personas que puedan aportan ideas tanto al diseño del juego como a la programación misma”. La fase 1 entonces es el análisis de los requerimientos que presenté anteriormente como si un cliente nos pidiera que desarrolláramos el juego.

Análisis orientado al objeto (Fase 2)

Ahora viene la fase 2 en la que analizamos el sistema para encontrar las clases y las relaciones entre las clases que puedan existir.

Existen varias técnicas para llevar a cabo esta etapa. Ocuparemos las tres más sencillas para hacer un análisis de los requerimientos. Estas tres técnicas son:

Casos de Uso:

- Es una técnica de modelado utilizada para describir lo que un nuevo sistema debe hacer o lo que un sistema existente ya hace.
- Los casos de uso representan una vista externa del sistema.
- Un modelo de casos de uso se construye mediante un proceso iterativo durante las reuniones entre los desarrolladores del sistema y los clientes (y/o los usuarios finales) conduciendo a una especificación de requisitos sobre la que todos coinciden.
- Un caso de uso captura algunas de las acciones y comportamientos del sistema y de los actores.
- El modelado con casos de uso fue desarrollado por Ivar Jacobson.

Descripción informal en lenguaje natural

- Subrayar los nombre y verbos de la descripción del problema
- Los **nombres** representan los **objetos** candidatos
- Los **verbos** representan las **operaciones** candidatas.

Tarjetas CRC (Clases/Responsabilidades/Colaboradores)

- Es una forma simple de analizar escenarios
- Son muy útiles para la enseñanza del AOO, DOO y POO
- Facilitan las “tormentas de ideas” y la comunicación entre desarrolladores
- Las tarjetas CRC pueden disponerse espacialmente para representar relaciones de colaboración
- Las tarjetas CRC también se pueden colocar para expresar jerarquías de generalización / especialización

Casos de Uso

Empecemos entonces con los casos de uso de nuestro juego.

1. Caso de Uso 1: El jugador inicia el juego
    a. El jugador ejecuta el juego
    b. Selecciona desde un menú empezar a jugar o elegir algunas opciones
    c. Si elige jugar, realizar el juego hasta que se le acaben las vidas.
    d. Vuelve al menú principal.

Como ven ahí tenemos nuestro primer caso de uso, totalmente primitivo y abstracto, pero siguiendo los prácticos consejos de sencillez lograremos ir profundizando en el concepto final.

Supongamos ahora que el usuario quiere seleccionar algunas opciones. Algunas de ellas serán por el momento nivel de dificultad y selección de los controles del juego:

2. Caso de Uso 2: El jugador selecciona el menú de opciones
    a. Se muestra en pantalla otro menú con las opciones de cambiar controles o cambiar nivel de dificultad.
    b. Si cambia alguna de los parámetros para las opciones, entonces se procede a actualizar el juego con los nuevos parámetros.
    c. Se vuelve al menú principal

Nuevamente la sencillez. Lo más tranquilo posible. Ahora analizaremos el tercer caso de uso: empieza a jugar.

3. Caso de uso 3: Empieza a jugar.
    a. Se establece el número inicial de vidas y energía por cada vida.
    b. Empieza la etapa inicial desde el principio con cero puntaje y algunos proyectiles secundarios
    c. La nave se sitúa en medio de la pantalla mirando hacia arriba.

Un último caso de uso es cuando el usuario dentro del juego presiona la tecla *escape*.

4. Caso de uso 4: El jugador presiona la tecla *escape* dentro del juego

a. Se despliega un dialogo en pantalla que pregunta si el jugador quiere salir del juego actual.
b. Si selecciona *si*, el juego termina y se vuelve al menú principal, sino se borra el diálogo y se continúa con la ejecución del juego.

Por ahora tenemos lo que se llama el armazón principal del juego: El inicio, el medio y el final. Cada uno de los casos de uso representa un "*que pasa cuando*" y nos ayudarán más adelante a descubrir tanto nuevo objetos como nuevas responsabilidades de estos.

Descripción informal en lenguaje natural

Procederemos con la técnica de la descripción informal en lenguaje natural destacando los ***sujetos*** que representan objetos candidatos y verbos que representan ***operaciones*** candidatas.

Este análisis de sujetos y verbos se puede aplicar a algún análisis independiente o incluso sobre los casos de uso. En nuestro caso lo aplicaremos a los casos de uso, es decir, encontraremos los sujetos y verbos existentes en los 4 casos de usos anteriores.

La lista de sujetos es la siguiente:

- Jugador
- Juego
- Menú
- Vidas
- Pantalla
- Controles
- Dificultad
- Energía
- Puntaje
- Proyectiles
- Tecla
- Diálogo

La lista de operaciones candidatas (obtenida de los verbos) son los siguientes:

- Iniciar el juego
- Ejecutar el juego
- Seleccionar desde menú
- Realizar el juego
- Mostrar en pantalla
- Cambiar controles
- Cambiar nivel de dificultad
- Actualizar el juego
- Establecer número de vidas
- Establecer nivel de energía
- Situar la nave
- Presionar la tecla

Ahora que tenemos la lista completa de posibles candidatos a clases y operaciones candidatas procedemos en una primera instancia a hacer una refinación eliminando o reemplazando los nombres para hacerlos más descriptivos.

- Jugador
- Juego
- Menú
- Vidas
- Pantalla
- Controles ➔ Control
- Dificultad ➔ Nivel de Dificultad
- Energía
- Puntaje
- Proyectiles ➔ Proyectil
- Tecla ➔ Componente de periférico
- Diálogo ➔ Menú

Se han cambiado algunos nombres de operaciones candidatas para hacerlas más entendibles. Controles pasa ha ser una clase genérica llamada Control. Tecla a pasado a ser un componente de periférico como por ejemplo el clic derecho sería un componente del mouse. Diálogo también es de tipo menú.

La segunda fase es eliminar cualquier frase que describa cosas fuera del sistema, sobre las cuales el sistema no tiene control:

- Jugador (X)

Pero reemplazamos la clase Jugador con Unidad Controlable, que es la representación en el juego del humano que manipula la unidad. Para que quede más claro aún: se elimina jugador porque no podemos darle la orden, por ejemplo, de salir del juego, de elegir tal nivel de dificultad, etc. Finalmente se agrega la siguiente clase:

- Unidad Controlable

La tercera fase es eliminar las frases que representan atributos de otras clases:

- Nivel de Dificultad (atributo de Juego)
- Puntaje (atributo de Juego)
- Energía (atributo de Actor Controlable)
- Vidas (atributo de Juego)

Además, y ya que se introdujo el sujeto *periférico* agregaremos esta clase. Así, la lista final es:

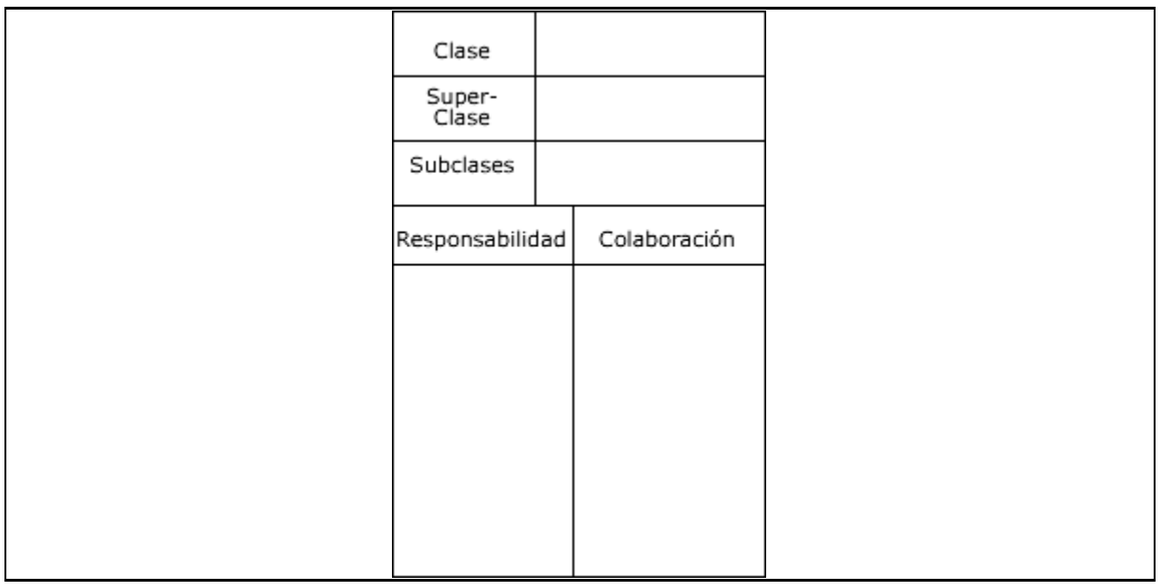
- Juego
- Menú
- Pantalla
- Control
- Proyectil
- Unidad Controlable
- Periférico

El análisis de las operaciones se hará más adelante con ayuda de las tarjetas CRC.

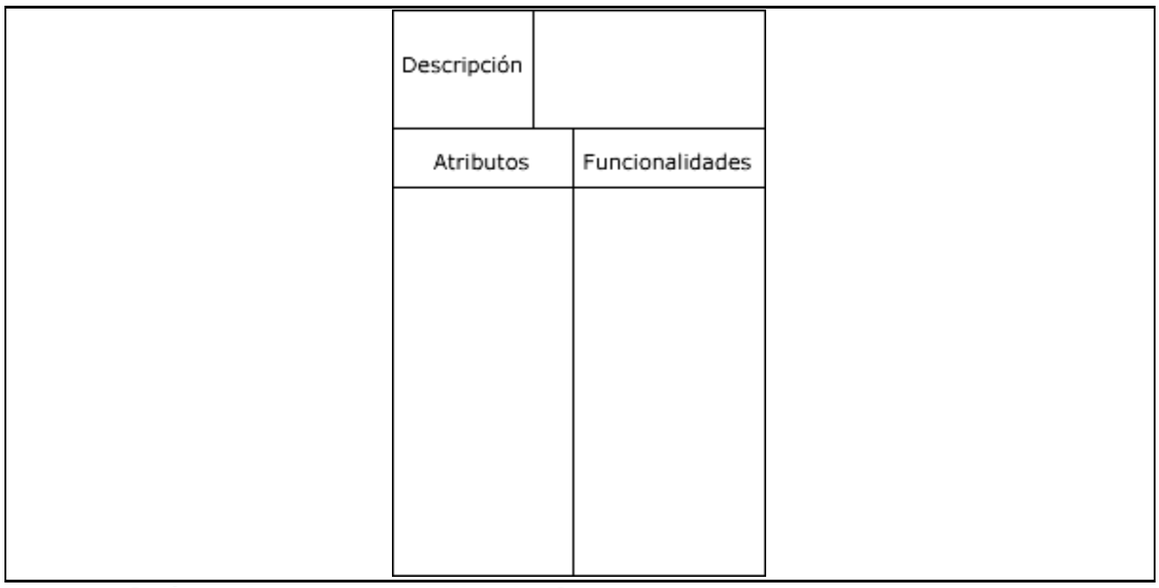
Tarjetas CRC

Ahora que ya tenemos individualizadas las clases es el tiempo de introducir las tarjetas de Clase, Responsabilidad y Colaboración.

Una tarjeta CRC generalmente se confecciona en una ficha de 10x15 cm. como lo muestra la figura:



También como opcional, ya que algunos autores lo señalan, es el reverso de la tarjeta como la parte “que no se ve” de la interfaz de la clase:



¿Qué son las responsabilidades?

- El conocimiento que mantiene un objeto
- Las acciones que un objeto puede realizar.

Las responsabilidades son la forma de dar sentido al propósito de un objeto y su lugar en el sistema.

Los convenios entre dos clases representan una lista de servicios que en una instancia de una clase (el cliente) puede requerir de una instancia de otra clase (el servidor). Las responsabilidades solo abarcan los servicios públicos. Por lo tanto, definir las responsabilidades es un nivel de implementación independiente. Recuerda que los convenios entre servidores y clientes especifican qué cosas se pueden obtener y no cómo se obtienen.

¿Cómo se obtienen las responsabilidades?

Las responsabilidades las obtendremos de los casos de uso y de las clases que ya obtuvimos. Hay que listas todos los verbos (las operaciones candidatas) y utilizando nuestro juicio determinaremos cual de ellos representan claramente las acciones que un objeto efectúa en el sistema.

La información que un objeto puede mantener y manipular también determina las responsabilidades del mismo.

¿Cómo se asignan las responsabilidades?

- Distribuya la “inteligencia” del sistema igualitariamente.
- Comience con responsabilidades tan genéricas como sea posible.
- Una las funciones con la información relacionada (si hay)
- Ponga la información de una cosa en un solo lugar.
- Comporta responsabilidades entre objetos relacionados.

Responsabilidades adicionales podrían ser identificadas si se examinan las relaciones entre clases. Tres tipos de relaciones son utilizadas particularmente al respecto:

- Es del tipo
- Es análogo a
- Es parte de

Por ahora no llevaremos a cabo el análisis con las tarjetas CRC ya que la mayor parte de las clases que lleva un videojuego ya ha sido ideada antes. Solo emplearemos esta herramienta cuando estemos frente a algo nuevo y lo adaptaremos a lo creado antes.

Estas clases de las que les hablo forman parte de los marcos de trabajo que introduciremos en el siguiente capítulo, los cuales toman el juego como un sistema de tiempo real en la que intervienen actores dentro de una escena determinada. Los actores pueden ser las naves, tanques, proyectiles, etc. Y la escena puede ser el espacio, la selva, etc.

Ya he introducido en este análisis algunas de las clases típicas en el diseño de un juego como son *juego, menú, control, periférico y pantalla*. Dos esas clases forman parte del mundo físico (periférico y pantalla) y son precisamente esas clases las que son más fáciles de descubrir.

Resumen

En este capítulo vimos la parte esencial y más difícil a la hora de empezar un juego: el comienzo. Establecer que vamos a hacer y en cuanto tiempo es una tarea difícil sobre todo cuando se trabaja solo o por entretención.

Se desarrollaron algunos puntos sugeridos en el capítulo anterior (presentados como 5 fases), particularmente las fases de requerimientos y análisis.

En la etapa de análisis se ocuparon las herramientas de casos de uso y tarjetas CRC, quedando estas últimas a la espera del marco de trabajo, una técnica muy utilizada para el desarrollo de juegos.

Capítulo 6 :
MARCO DE TRABAJO PARA UN JUEGO

¿Que es un marco de trabajo?

Los marcos de trabajo son como contenedores de conocimiento, conocimiento sobre una tarea particular. En nuestro caso, la tarea es escribir un videojuego. La clave detrás de un marco de trabajo es una antigua máxima: no reinventes la rueda. La idea es escribir el código una vez y usarlo varias veces nuevamente, muchas veces en diferentes circunstancias.

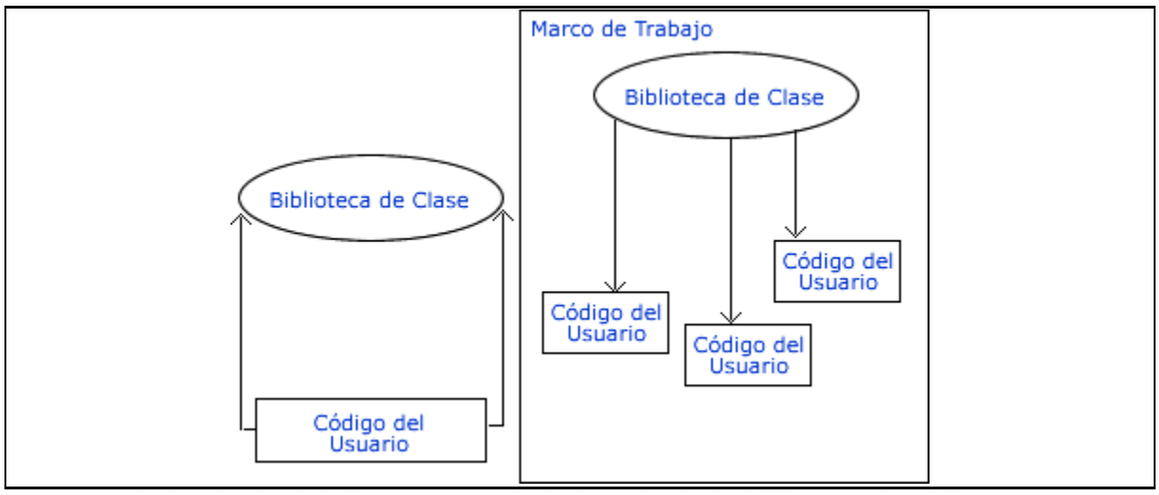
Ahora, debes estar pensando que los marcos de trabajo son como bibliotecas de código (como Allegro). Las bibliotecas de códigos, como las DLL de Windows, también contienen código escrito para realizar una tarea, entonces ¿Cuál es la diferencia?

Los marcos de trabajo son el próximo paso lógico desde las bibliotecas de códigos. Son a veces descritos como bibliotecas de código activas. *Activa* es la palabra clave aquí. Los marcos de trabajo contienen bibliotecas de código, pero, más que tener que escribir código extra para que las puedas usar, ellas hacen suposiciones a tu favor.

Para hacer esto, a veces los marcos de trabajo vienen pre-configurados, asumen de qué forma quieres usarlos. Típicamente tendrás que instanciar una clase o dos (crear objetos a partir de ellas) desde el marco de trabajo y eso es todo, puede funcionar perfectamente así. Hacer eso no sería nada particularmente interesante, pero estaría trabajando, es decir, se podría decir que se está ejecutando un juego (aunque sea mostrando una pantalla negra).

Si esperas que algo diferente suceda, eliges otra configuración para el marco de trabajo. Tu tarea principal como programador es seleccionar los comportamientos del marco de trabajo y *agregar* tus propios comportamientos. El marco de trabajo luego usará tu reelección para hacer las cosas que quieras. Frecuentemente, un marco de trabajo bien escrito contendrá el código necesario para hacer muchas de tus tareas. Solo necesitas decirle al marco de trabajo que piezas de código quieres ejecutar. El marco de trabajo toma un rol activo.

Desde otro punto de vista los marcos de trabajo son bibliotecas de código al revés. Tú usas bibliotecas de código escribiendo otro código que llama a estas bibliotecas; usas un marco de trabajo escribiendo el código que el marco de trabajo llama. Ve la figura siguiente:



A la izquierda del diagrama está el ejemplo de la biblioteca de código. Esta muestra una gran porción de trozo de código que llama repetida mente a las bibliotecas de códigos. A la derecha es el ejemplo del marco de trabajo. Aquí, el código escrito por el usuario son pequeños (métodos sobrescritos) que son llamados por el marco de trabajo. Ambos ejemplo potencialmente hacen el mismo trabajo.

En conjunto, el marco de trabajo descarga un montón de tareas de los hombros del programador. Este proceso es a veces simple y menos propenso a errores que escribir el código para llamar a la biblioteca de código. Por otro lado hay cosas que suceden y que no quieres. Esas son cosas que sobrescribirás y, posiblemente, escribirás por ti mismo. La buena noticia, a pesar de todo, es que en el principio puedes avanzar un montón de camino rápidamente. Si el marco de trabajo está bien diseñado y hace muchas cosas que necesitarás, la parte de personalización se minimizará y verás como has ganado tiempo.

Creando el corazón del marco de trabajo

Empezaremos a aplicar los conceptos de los capítulos anteriores. Tomando ventaja tanto de la POO como de el marco de trabajo llegaremos a un diseño re-usable casi en su totalidad, por lo que no se extrañen que varios juegos futuros tengan “el mismo aspecto” (en código fuente, claro está).

Las partes fundamentales del marco de trabajo que crearemos son tres clases abstractas: Game, StageManager, Actor, ActorManager.

Nota: En el marco de trabajo que vamos a crear, cualquier objeto que pertenezca a la interfaz (no a la implementación) tendrá un nombre en inglés por razones obvias. El objetivo que se persigue es crear un marco de trabajo para Allegro como proyecto paralelo para que pueda servir a muchas personas en la comunidad.

Todo el análisis orientado a objetos se puede resumir de alguna forma en una tarjeta CRC. Es por eso que en este caso solo mostraremos la tarjeta y no todo el análisis porque esta clase es prácticamente obvia.

Clase Actor

Los actores son los elementos del juego que nosotros movemos o vemos moverse a través de la pantalla. Ellos son las piezas en un juego de ajedrez, son las cartas en un juego como “el solitario” y son las naves, tanques y proyectiles en nuestro juego. La clase Actor es la responsable de mantener la apariencia y posición del actor. La tarjeta CRC para el actor es la siguiente:

Clase	Actor
Super-Clase	
Subclases	
Responsabilidad	Colaboración
1.- Mantener la posición del actor. 2.- Dibujar al actor	

El código fuente de la interfaz (*actor.*h) es el siguiente:

Código Fuente 20. Archivo actor.h

```
#ifndef ACTOR_H
#define ACTOR_H

#include <allegro.h>

class Actor
{
      public:

                Actor();
                virtual ~Actor();

                virtual void draw(BITMAP *bmp);
                virtual void update();
                void set_x(int pos_x) { x=pos_x; }
                void set_y(int pos_y) { y=pos_y; }
                int get_x() { return x; }
                int get_y() { return y; }
      protected:

                int x, y;
};

#endif ACTOR_H
```

La primera sentencia es para que no se cometa el error de incluir dos veces el archivo *actor.h*. Es una vieja técnica. En este archivo se implementan *inline* las funciones *get_x* y *get_y*, y los procedimiento *set_x* y *set_y*

Ahora veremos la implementación que, sencillamente, no hace nada porque es una clase abstracta, es decir, desde ella solo de derivará.

Código Fuente 21. Archivo actor.cpp

```
#include "actor.h"

Actor::Actor()
{
}

Actor::~Actor()
{
}

void Actor::draw(BITMAP *bmp)
{
}

void Actor::update()
{
}
```

Por supuesto que esta no es la interfaz final de la clase Actor, más adelante agregaremos más características como una interfaz para el soporte de periféricos, de imágenes, etc. Por ahora se dibuja en un BITMAP recibido como parámetro en el procedimiento *draw*.

Clase ActorManager

La clase ActorManager es muy simple: mantiene una lista de actores. Como ya vimos anteriormente, la clase StageManager usa la clase ActorManager para recorrer y obtener la lista de actores. El juego guarda la clase ActorManager. A continuación se muestra la tarjeta CRC:

Clase	ActorManager
Super-Clase	
Subclases	
Responsabilidad	Colaboración
1.- Mantener una lista de actores.	

La interfaz es la siguiente:

Código Fuente 22. archivo actormanager.h

```
#ifndef ACTORMANAGER_H
#define ACTORMANAGER_H

#include <algorithm>
#include <list>

class Actor;
class Game;

class ActorManager
{
      public:
                ActorManager(Game *g);
                ~ActorManager();

                void add(Actor *a);
                void del(Actor *a);
                void rewind();
                Actor *next();
                Actor *current();
                void update();
                int num_actors();

     protected:
                Game *game;

                list<Actor*> actors;
                list<Actor*>::iterator actors_iter;
};

#endif
```

Y la implementación es:

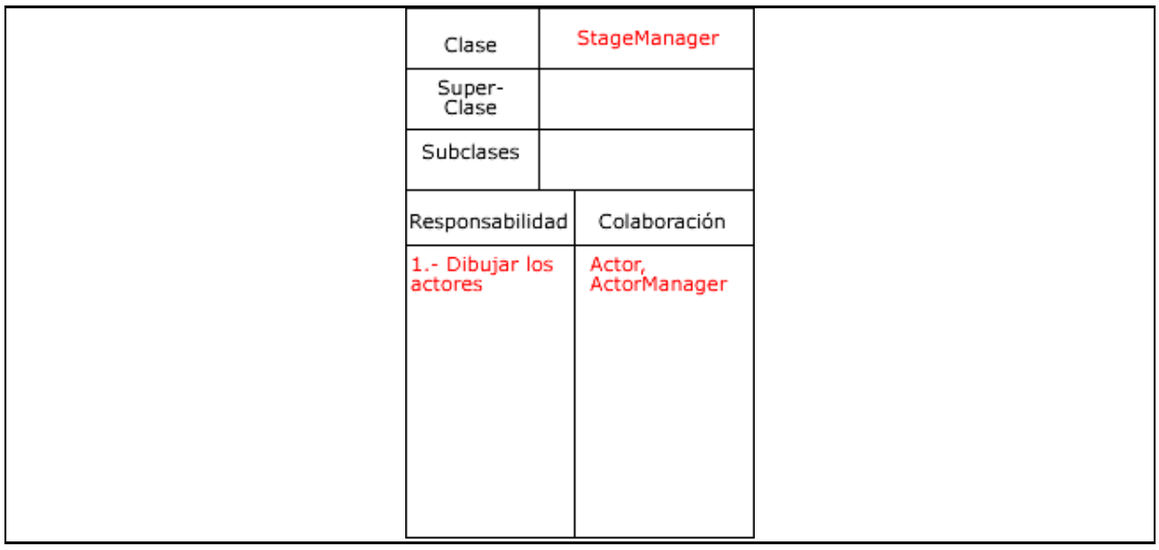
Código Fuente 23. archivo actormanager.cpp

```
#include "actor.h"
#include "actormanager.h"

ActorManager::ActorManager(Game *g)
{
	game=g;
}

ActorManager::~ActorManager()
{
	for (actors_iter=actors.begin(); actors_iter!=actors.end();
actors_iter++)
		delete (*actors_iter);
}


void ActorManager::add(Actor *a)
{
	actors.push_back(a);
}

void ActorManager::del(Actor *a)
{
	list<Actor*>::iterator tmp_actors_iter;

	tmp_actors_iter=find(actors.begin(), actors.end(), a);
	if (tmp_actors_iter!=actors.end())
		actors.erase(tmp_actors_iter);
}

void ActorManager::rewind()
{
	actors_iter=actors.begin();
}

Actor *ActorManager::next()
{
	Actor *tmp_a;

	tmp_a=*actors_iter;
	if (actors_iter==actors.end()) return NULL;
	actors_iter++;
	return tmp_a;
}

Actor *ActorManager::current()
{
	if (actors_iter==actors.end()) return NULL;
	else
		return *actors_iter;
}

int ActorManager::num_actors()
{
	return actors.size();
}

void ActorManager::update()
{
	list<Actor*>::iterator tmp_iter;
	for (tmp_iter=actors.begin(); tmp_iter!=actors.end(); tmp_iter++)
		(*tmp_iter)->update();
}
```

Espero que los nombres de los procedimientos sean lo suficientemente claros como para no estar explicandolos exahustivamente. Luego, cuando vean la implementación de StageManager, verán como se ocupan los procedimientos *rewind*, *next* y *current* de ActorManager.

Clase StageManager

La escena (interpretación de la palabra stage en este contexto) es donde todos los actores están. Un objeto de StageManager será responsable de dibujar la escena. Este conoce los actores que están en la escena y, en general, todos los objetos que se dibujarán en ella. También será responsable más adelante de la optimización al dibujar la escena. La tarjeta CRC es la siguiente:



La interfaz es la siguiente:

Código Fuente 24. archivo stagemanager.h

```
#ifndef STAGEMANAGER_H
#define STAGEMANAGER_H

#include <allegro.h>

class Game;

class StageManager
{
	public:

		StageManager(Game *g, int w, int h);
		~StageManager();
		int w();
		int h();
		void update();
		void draw();

	protected:

		Game *game;
		BITMAP *buffer;
		int width, height;
};

#endif
```

Y la implementación es:

Código Fuente 25. archivo stagemanager.cpp

```
#include "game.h"
#include "actor.h"
#include "actormanager.h"
#include "stagemanager.h"

StageManager::StageManager(Game *g, int w, int h)
{
	game=g;
	width=w;
	height=h;
	buffer=create_bitmap(SCREEN_W, SCREEN_H);
}

StageManager::~StageManager()
{
	destroy_bitmap(buffer);
}

int StageManager::w()
{
	return width;
}

int StageManager::h()
{
	return height;
}

void StageManager::update()
{
	draw();
}

void StageManager::draw()
{
	Actor *tmp;
	game->actor_manager->rewind();
	clear(buffer);
	while ((tmp=game->actor_manager->next())!=NULL)
	{
		tmp->draw(buffer);
	}
	blit(buffer, screen, 0,0,0,0,SCREEN_W, SCREEN_H);
}
```

Por el momento se cuenta con una burda optimización al dibujar que consiste en dibujar todo en un mapa de bits auxiliar para luego dibujarlo en la pantalla. Es muy lento, pero más adelante veremos algunas técnicas como *dirty rectangle* y adaptaciones de ella.

Como pueden observar el método *update* lo único que hace por el momento es llamar a *draw* que finalmente recorre la lista de actores y que a su vez llama al método *draw* de Actor.

Todos los administradores (managers) tienen una referencia al juego al que pertenecen, reflejado en la variable protegida *game*.

Clase Game

La clase Game se encarga de mantener toda la información respecto del juego actual que se está desarrollando. Esta contiene a los administradores y les releva trabajo. Por el momento contendrá a StageManager y ActorManager.

Clase	Game
Super-Clase	
Subclases	

Responsabilidad	Colaboración
1.- Crear y mantener a los administradores 2.- Relevar tareas a los administradores 3.- Inicializar las funciones de Allegro necesarias	StageManager, ActorManager

La interfaz es la siguiente:

Código Fuente 26. archivo game.h

```
#ifndef GAME_H
#define GAME_H

#include <string>
class ActorManager;
class StageManager;

class Game
{
	public:

		Game();
		virtual ~Game();
		ActorManager *actor_manager;
		StageManager *stage_manager;
		virtual void init(int gfx_mode, int w, int h, int col);
		virtual void main();
		void set_name(string name);
		string get_name();

	protected:

		string name;
		void update();

		int gfx_w,gfx_h;
		int colors;
	private:

		void start();
		void shutdown(string message="Gracias por jugar");
		void create_actormanager();
		void create_stagemanager();
};

#endif
```

Y la implementación es la siguiente:

Código Fuente 27. archivo game.cpp

```
#include <allegro.h>
#include "actormanager.h"
#include "stagemanager.h"
#include "game.h"


Game::Game()
{
     actor_manager=NULL;
     stage_manager=NULL;
}

Game::~Game()
{
}

void Game::init(int gfx_mode, int w, int h, int col)
{
     allegro_init();
     install_keyboard();
     set_color_depth(col);
     if (set_gfx_mode(gfx_mode,w, h, 0,0)<0)
     {
          shutdown("No se pudo inicializar modo grafico");
          return;
     }
     create_actormanager();
     create_stagemanager();
     start();
}

void Game::shutdown(string message="Gracias por jugar")
{
     delete actor_manager;
     delete stage_manager;
     set_gfx_mode(GFX_TEXT,0,0,0,0);
     cout << name << endl;
     cout << message << endl;
     allegro_exit();
}

void Game::create_actormanager()
{
     actor_manager = new ActorManager(this);
}

void Game::create_stagemanager()
{
     stage_manager = new StageManager(this, gfx_w, gfx_h);
}
void Game::start()
{
     main();
     shutdown();
}

void Game::main()
{
     while (!key[KEY_ESC]);
}

void Game::set_name(string n)
{
     name=n;
}

string Game::get_name()
{
     return name;
}

void Game::update()
{
     stage_manager->update();
```

```
        actor_manager->update();
    }
```

Ahora el desafío es probar este pequeño marco de trabajo para hacer juego del capítulo 2. Continuemos.

Creando nuestro primer juego II

Tal como dije antes para hacer los juegos con un marco de trabajo solo es necesario heredar clases o sobrescribir algunos métodos.

Primero que todo heredaremos desde Actor una nueva clase que llamaremos AirCraft (nave). Ahora si haremos un pequeño análisis:

"Quiero que la nave tenga asociada una imagen y que se pueda mover con las teclas de dirección", dice el *boss*. Un pequeño requerimiento después de todo.

De esta forma podemos obtener una lista de candidatos a clases desde este requerimiento.

- Imagen
- Teclas de Dirección

Y otra lista de posibles métodos:

- Asociar imagen
- Mover

La tarjeta CRC que se obtiene es la siguiente:

Clase	AirCraft
Super-Clase	Actor
Subclases	
Responsabilidad	Colaboración
1.- Asociar una imágen 2.- Moverse mediante las teclas	

Claramente la nave *tiene* una imagen asociada, es decir, una variable protegida BITMAP dentro de la interfaz. Para entregarle a la nave esa imagen agregaremos un método llamado *set_image* que tendrá como parámetro un puntero BITMAP. Ya tenemos solucionadas las partes de asociar una imagen y guardarla.

Para solucionar el asunto de las teclas y la responsabilidad de moverse sobrescribiremos el método *update* de Actor para realizar:

Código Fuente 28.

```
    void AirCraft::update()
    {
        if (key[KEY_UP]) y-=2;
        if (key[KEY_DOWN]) y+=2;
        if (key[KEY_LEFT]) x-=2;
        if (key[KEY_RIGHT]) x+=2;
        if (x<0) x=0;
        if (x>SCREEN_W-image->w) x=SCREEN_W-image->w;
        if (y<0) y=0;
        if (y>SCREEN_H-image->h) y=SCREEN_H-image->h;
    }
```

Debemos recordar que este método es llamado cada vez que realizamos la actualización del juego (*Game::update*) mediante ActorManager.

Para dibujarse solo se sobrescribe el método *draw*:

Código Fuente 29.

```
void AirCraft::draw(BITMAP *bmp)
{
     draw_sprite(bmp, image, x,y);
}
```

La interfaz final es la siguiente:

Código Fuente 30.

```
class AirCraft : public Actor
{
     public:
               AirCraft();

               void draw(BITMAP *bmp);
               void update();
               void set_image(BITMAP *bmp);

     protected:

               BITMAP *image;
};
```

Y la implementación de los demás métodos y el constructor es:

Código Fuente 31.

```
AirCraft::AirCraft()
{
     image=NULL;
}

void AirCraft::set_image(BITMAP *bmp)
{
     image=bmp;
}
```

Para hacer nuestro primer juego, que llamaremos *test_framework* simplemente lo heredaremos desde *Game* sobrescribiendo el método *main* que es llamado luego de la inicialización. La interfaz es:

Código Fuente 32.

```
class TestFrameWork : public Game
{
     public:
               void main();
};
```

Y la implementación:

Código Fuente 33.

```
void TestFrameWork::main()
{
     BITMAP *bmp;
     PALETTE tmp;
     AirCraft *a=new AirCraft;

     bmp=load_bitmap("nave.pcx", tmp);
     a->set_image(bmp);
     a->set_x(0);
     a->set_y(0);
     actor_manager->add(a);
     while (!key[KEY_ESC]) update();
     destroy_bitmap(bmp);
}
```

Como pueden ver, en el método *TestFrameWork::main* siempre debe llamarse a *update* cada vez que podamos para que el juego se actualice. El juego *testFrameWork* se ejecuta hasta que se presiona la tecla escape.

Probando el potencial del marco de trabajo

Ahora veremos cuan simple es agregar nuevos tipos de actores a nuestro juego. Vamos a poner algunas estrellas en el fondo de la pantalla, como en el espacio. El actor entonces lo llamaremos Star.

"Quiero que en el fondo de la pantalla avancen estrellas a diferentes velocidades y en posiciones al azar", dice *the boss*.

Esta es una clase mucho más simple y la tarjeta CRC es la siguiente:

Clase	Star
Super-Clase	Actor
Subclases	
Responsabilidad	Colaboración
1.- Tomar una posición al azar y reinicializarse al llegar abajo	

La interfaz e implementación son:

Código Fuente 34. clase Star

```
class Star : public Actor
{
	public:

		Star();

		void update();
		void draw(BITMAP *bmp);
	protected:

		void reinit();
		int vy;
};

Star::Star()
{
	reinit();
}

void Star::reinit()
{
	x=rand()%SCREEN_W;
	y=0;
	vy=1+rand()%4;
}

void Star::update()
{
	y+=vy;
	if (y>SCREEN_H)
	{
		reinit();
	}
}

void Star::draw(BITMAP *bmp)
{
	putpixel(bmp,x,y,makecol(255,255,255));
}
```

Como pueden ver, nuevamente se sobrescriben los métodos *update* y *draw* solo que ahora lo que se dibujará es un simple píxel blanco en la posición (x, y).

Ahora solo modificamos el método *TestFrameWork::main* para que cree las estrellas y listo!.

Código Fuente 35.

```
...
      Star *star_tmp;
      for (int i=0; i<200;i++)
      {
               star_tmp=new Star;
               star_tmp->set_y(rand()%SCREEN_H);
               actor_manager->add(star_tmp);
      }
...
```

Para crear las estrellas lo hacemos mediante algunas funciones estandar para la generación de número al azar. Recuerda que para ocupar *rand()* ,*srand()* y *time()* hay que incluir *ctime* y *cstdlib*. *Rand* retorna un número al azar, *srand* crea la semilla del generador de números al azar y ocuparemos el parámetro *time* para hacerlo.

Una captura de pantalla del juego creado:

Finalmente el código fuente de nuestro juego es:

Código Fuente 36. archivo test_framework.cpp

```
#include <allegro.h>
#include <ctime>
#include <cstdlib>
#include "game.h"
#include "actor.h"
#include "actormanager.h"

class AirCraft : public Actor
{
     public:
               AirCraft();

               void draw(BITMAP *bmp);
               void update();
               void set_image(BITMAP *bmp);

     protected:

               BITMAP *image;
};

AirCraft::AirCraft()
{
     image=NULL;
}

void AirCraft::draw(BITMAP *bmp)
{
     draw_sprite(bmp, image, x,y);
}

void AirCraft::update()
{
     if (key[KEY_UP]) y-=2;
     if (key[KEY_DOWN]) y+=2;
     if (key[KEY_LEFT]) x-=2;
     if (key[KEY_RIGHT]) x+=2;
     if (x<0) x=0;
     if (x>SCREEN_W-image->w) x=SCREEN_W-image->w;
     if (y<0) y=0;
     if (y>SCREEN_H-image->h) y=SCREEN_H-image->h;
}

void AirCraft::set_image(BITMAP *bmp)
{
     image=bmp;
}

class Star : public Actor
{
     public:

               Star();

               void update();
               void draw(BITMAP *bmp);
     protected:

               void reinit();
               int vy;
};

Star::Star()
{
     reinit();
}

void Star::reinit()
{
     x=rand()%SCREEN_W;
     y=0;
     vy=1+rand()%4;
}

void Star::update()
```

```
{
	y+=vy;
	if (y>SCREEN_H)
	{
		reinit();
	}
}

void Star::draw(BITMAP *bmp)
{
	putpixel(bmp,x,y,makecol(255,255,255));
}

class TestFrameWork : public Game
{
	public:
		void main();
};

void TestFrameWork::main()
{
	BITMAP *bmp;
	PALETTE tmp;
	AirCraft *a=new AirCraft;
	Star *star_tmp;
	for (int i=0; i<200;i++)
	{
		star_tmp=new Star;
		star_tmp->set_y(rand()%SCREEN_H);
		actor_manager->add(star_tmp);
	}

	bmp=load_bitmap("nave.pcx", tmp);
	a->set_image(bmp);
	a->set_x(SCREEN_W/2);
	a->set_y(SCREEN_H/2);
	actor_manager->add(a);
	while (!key[KEY_ESC]) update();
	save_bitmap("screen.bmp",screen,tmp);
	destroy_bitmap(bmp);
}

int main()
{
	TestFrameWork game;
	srand(time(NULL));
	game.set_name("Test del Marco de Trabajo");
	game.init(GFX_AUTODETECT, 640,480,16);
}
```

Resumen

En este capítulo vemos lo básico en la creación de un marco de trabajo para luego probar como funciona éste intentando recrear lo hecho en el capítulo 2. Además, para probar la fácil extensibilidad del modelo orientado a objetos, agregamos nuevas características en un par de líneas.

Capítulo 7 :
VELOCIDAD CONSTANTE DE JUEGO

Al probar el marco de trabajo del capítulo anterior en distintos computadores, nos damos cuenta de que, dependiendo de la velocidad de ellos, la nave y los elementos de la pantalla se moverán más rápido o más lento. Por diferentes razones ocurre esto: velocidad del procesador, tarjeta gráfica, velocidad de respuesta de los periféricos, etc. En la mayoría de los casos se debe a la velocidad del procesador y, en los juegos 3D, a la tarjeta gráfica.

Nuestro juego debe moverse de igual modo en cualquier computador que lo probemos ya que fácilmente una “persona mal intencionada” podría ejecutarlo en un computador lento para que el juego se moviese lento. Al moverse lento el juego las reacciones son muchos mejores, ¡sería un superdotado de los videojuegos!.

Solucionando el problema de la velocidad

Establezcamos nuestro requerimiento: queremos que el juego se mueva a la misma velocidad en cualquier computador. Pero, ¿Qué es eso de “que se mueva a la misma velocidad”?, significa que el tiempo que pasa entre una actualización (una oportunidad para que nuestra nave se mueva) y otra, sea constante.

De este modo debemos llamar al método que actualiza toda la lógica del juego una cantidad de veces por segundo constante. Lógica del juego es todo lo no gráfico, es decir, los movimientos de los objetos, la lectura del teclado o joystick, la actualización de los puntajes, etc.

El método *Game::update()*, visto en el capítulo anterior, debe ser llamado cada vez que nosotros podamos, éste es el que controla la velocidad del juego ya que llama a las dos partes de nuestro marco de trabajo: el ciclo lógico y el ciclo gráfico, representados por *ActorManager::update()* y *StageManager::update()*, respectivamente.

En las bibliotecas Allegro existe un procedimiento que nos permite crear un *timer* el cual es un procedimiento que se llama *automáticamente* una determinada cantidad de veces por segundo o cada varios segundos. Este procedimiento es el que ocuparemos para tener un conteo objetivo del tiempo. El procedimiento *timer* aumentará una variable llamada *tick*, es decir, la variable *tick* aumentará su valor una cantidad determinada de veces por segundo.

Lo que primero debemos establecer es a que cantidad de ciclos por segundo queremos que nuestro juego se mueva. Estableceré en 70 la velocidad. Para entender esto, un ejemplo. Si nuestra nave se mueve un píxel cada vez que es llamado el método *AirCraft::update()*, la velocidad final sería de 70 píxel por segundo; si se mueve 2 pixel cada vez entonces la velocidad es de 140 píxel por segundo.

Cuando el ciclo lógico se ejecuta a una razón de 70 veces por segundo entonces se puede llamar al ciclo gráfico. En seudo-código sería algo así:

```
Si (numero_de_ciclos_logicos <= timer) entonces
bloque-si
        hacer_ciclo_lógico()
        numero_de_ciclos_lógicos++
fin-si
Si (numero_de_ciclos_lógicos >= timer) entonces
```

```
bloque-si
        hacer_ciclo_gráfico()
fin-si
```

Ahora introduciremos un nuevo concepto conocido como *salto de cuadros* (frameskip).

Salto de cuadros

Cada vez que no se puede ejecutar un ciclo gráfico entonces se contabiliza un *salto de cuadro* que representa un ciclo en el cual se prioriza la actualización lógica y se posterga la gráfica. Esta contabilización se puede llevar a cabo modificando el seudo-código presentado anteriormente:

```
Si (numero_de_ciclos_logicos <= timer) entonces
bloque-si
        hacer_ciclo_lógico()
        numero_de_ciclos_lógicos++
fin-si
Si (numero_de_ciclos_lógicos >= timer) entonces
bloque-si
        hacer_ciclo_gráfico()
        salto_de_cuadros=0
sino
        salto_de_cuadros++
fin-si
```

Ahora que tenemos contabilizado el salto de cuadros podríamos establecer un tope de estos. Por ejemplo, siempre esperamos que nuestro juego sea jugable en la mayor cantidad de computadores. Si se juega en un computador demasiado lento, quizá siempre estaría saltando cuadros en las escenas en las que se requiere de demasiado despliegue gráfico, entones ¿Nuestro jugador solo podría ver las escenas que tienen poco despliegue gráfico?. Por supuesto que no.

Para hace que de todas formas, independiente de la potencia gráfica del computador, siempre se pueda ver lo que pasa en pantalla, se establecerá un máximo de salto de cuadros. Si se cumple ese máximo, es obligatorio mostrar algo en pantalla. En seudo-código es:

```
Si (numero_de_ciclos_logicos <= timer) entonces
bloque-si
        hacer_ciclo_lógico()
        numero_de_ciclos_lógicos++
fin-si
Si ((numero_de_ciclos_lógicos >= timer) ||
    (salto_de_cuadors>max_salto_de_cuadros)) entonces
bloque-si
        hacer_ciclo_gráfico()
        Si (salto_de_cuadros>=max_salto_de_cuadros) entonces
        bloque-si
                numero_de_ciclos_logicos=timer
        fin-si
        salto_de_cuadros=0

sino
        salto_de_cuadros++
fin-si
```

La variable *max_salto_cuadro* representa la mayor cantidad de cuadros que pueden saltarse dentro del desarrollo del juego. Dentro del segundo "*Sí ...*" existe el condicional:

```
Si (salto_de_cuadros>=max_salto_de_cuadros) entonces
bloque-si
        numero_de_ciclos_logicos=timer
fin-si
```

Como pueden ver, aquí solo se entrará en el caso que el salto de cuadros sea máximo y la acción que se realiza (igualar el número de ciclos con el *timer*) permite seguir jugando como “si no hubiese pasado nada”. En caso de que esta sentencia no estuviese, al pasar por el cuello de botella gráfico, se realizaría un “gran salto de cuadros” para corregir el ciclo lógico con el gráfico.

De alguna manera, la velocidad constante de juego no puede realizarse correctamente porque siempre es necesario que en computadores antiguos algo pueda verse en pantalla.

Implementación en Allegro

Lo anterior fue un poco de teoría, ahora veamos como se hace en Allegro.

En el código fuente *game.cpp* agregaremos al inicio algunas “feas” variables globales y un procedimiento *timer* que se llamará *tick_count*, de la siguiente forma:

Código Fuente 37.

```
...
void tick_count();
volatile int tick;

void tick_count()
{
    tick++;
}
END_OF_FUNCTION(tick_count);
...
```

La macro *END_OF_FUNCTION* agrega código al procedimiento que será un *timer*, es algo técnico que no vale la pena explicar. Además, para las variables que se modificarán con el *timer*, en este caso *tick*, hay que declararlas como volatile. Debemos además indicarle a Allegro que este procedimiento y esta variable deben estar en un lugar de la memoria que nadie más puede modificar. Para hacer todo esto el método constructor de *Game* queda de la siguiente forma:

Código Fuente 38.

```
...
Game::Game()
{
     allegro_init();
     install_keyboard();
     actor_manager=NULL;
     stage_manager=NULL;
     install_timer();
     LOCK_VARIABLE(tick); // protege la variable
     LOCK_FUNCTION(tick_count); // protege procedimiento
     install_int(&tick_count, 14); // instala interrupción
}
...
```

Nota: A diferencia de la implementación anterior de la clase Game, la inicialización de Allegro se realiza en el constructor y no en el método Game::init()

Para ocupar las funciones de *timer* en Allegro debemos llamar al procedimiento *install_timer*.

LOCK_VARIABLE y *LOCK_FUNCTION* no necesitan mayor explicación. El procedimiento de más abajo es *install_int* que finalmente instala una interrupción con los siguientes parámentros:

- Dirección de memoria del procedimiento que será llamado cada vez que se interrumpa, y;
- Cada cuantos milisegundos será llamado

En nuestro caso lo fijamos en 14 para que el ciclo lógico sea llamado 70 veces por segundo... algo de matemáticas:

$$retardo\_de\_ciclos = \frac{1000}{ciclos\_deseados} = \frac{1000}{70} = \sim 14$$

Desde el momento que declaramos *install_int(&tick_count, 14)* el procedimiento *tick_count* está siendo llamado 70 veces por segundo, es decir, la variable *tick* aumenta su valor en 70 cada segundo.

Declararemos una nueva variable *privada* para la clase *Game* que llevará la cuenta de los ticks actuales de nuestro ciclo lógico, otra que tendrá el salto de cuadros actual y finalmente otra que mantendrá el máximo de salto de cuadro que estamos dispuestos a aceptar:

Código Fuente 39.

```
...
private:

     int actual_tick, frame_skip, max_frame_skip;
...
```

Estamos listos para aplicar el “algoritmo”, ahora solo nos falta ajustar el tiempo actual con el real, esto lo haremos cuando se inicie el juego: en el método *Game::start()*.

Código Fuente 40.

```
...
void Game::start()
{
     actual_tick=tick; // se alinea con el timer real
     old_tick=tick;
     max_frame_skip=15; // estable el salto de cuadro por defecto
     main();
     shutdown();
}...
```

Modifiquemos entonces el método *Game::update()*:

Código Fuente 41.

```
...
void Game::update()
{
    if (actual_tick<=tick)
    {
        actor_manager->update(); // ciclo lógico
        actual_tick++;
    }
    if ((actual_tick>=tick) || (frame_skip>max_frame_skip))
    {
        stage_manager->update(); // ciclo gráfico
        if (frame_skip>max_frame_skip) actual_tick=tick;
        graphic_tick++;
        frame_skip=0;
    }
    else
    {
     frame_skip++;
    }
}
...
```

Para poder establecer el máximo de salto de cuadros agregaremos el método *Game::set_max_frame_skip()*:

Código Fuente 42.

```
     protected:
...
        void set_max_frame_skip(int max_fs);
...
```

Y su implementación trivial:

Código Fuente 43.

```
void Game::set_max_frame_skip(int max_fs)
{
     max_frame_skip=max_fs;
}
```

Eso es todo, nuestra nave se moverá a 70 píxeles por segundo en cualquier computador.

Ahora nos interesaría saber cuantos ciclos gráficos realiza nuestro juego, en palabras simples “si se mueve bien”.

Los ciclos gráficos: cuadros por segundo

En muchos análisis de juegos 3D se da como referencia los *cuadros por segundo* (o *frames per second, fps*) que desarrolla, lo que es distinto de los ciclos lógicos. Los cuadros por segundo reflejan cuantas veces se está dibujando por segundo la escena en pantalla, así en un momento del juego podríamos tener 1000 cuadros por segundo y bajar drásticamente a 30 al aparecer en medio de la pantalla un monstruo gigante que tira fuego por la boca, lo importante es que nuestro héroe no pierda velocidad de movimiento. Agregaremos algunas cosas a marco de trabajo para conocer cuantos cuadros por segundo estamos desarrollando.

Lo lógico es contar cuantas veces se dibuja nuestra escena en pantalla, es decir, cuantas veces se llama al método *StageManager::update()*, en seudo-código:

```
Si (numero_de_ciclos_lógicos>=timer) entonces
        Hacer_ciclo_gráfico()
        Numero_de_ciclos_graficos++
Si (timer – antiguo_timer >= 70) entonces
        Imprimir(“cuadros por segundo:”)
        Imprimir(numero_de_ciclos_graficos)
        Numero_de_ciclos_graficos=0
        antiguo_timer=timer
```

Las nuevas variables se declaran en la clase *Game* como privadas:

Código Fuente 44.

```
...
private:
      int graphic_tick, old_tick;
...
```

Ahora hacemos una pequeña modificación a *Game::start ()*

Código Fuente 45.

```
...
void Game::start()
{
     actual_tick=tick; // se alinea con el timer real
     old_tick=tick;
     main();
     shutdown();
}
...
```

Y finalmente en *Game::update()* mostramos además el salto de cuadros:

Código Fuente 46.

```
...
void Game::update()
{
...
    if (tick-old_tick >=70) // se cumplió un segundo
    {
        rectfill(screen,0,0,200,14,0); // borrar el antigüo marcador
        textprintf(screen, font, 0,0,-1, "fps: %u frameskip:%u", graphic_tick,
frame_skip);
        graphic_tick=0;
        old_tick=tick;
    }
...
}
```

Y para que el marcador aparezca en pantalla movemos un poco el área de dibujo del objeto StageManager:

Código Fuente 47. En archivo stagemanager.cpp

```
...
void StageManager::draw()
{
      Actor *tmp;
      game->actor_manager->rewind();
      clear(buffer);
      while ((tmp=game->actor_manager->next())!=NULL)
      {
            tmp->draw(buffer);
      }
      // 14 pixels abajo para el marcador de fps
      blit(buffer, screen, 0,0,0,14,SCREEN_W, SCREEN_H);
}
...
```

Si todo está bien, en pantalla habrá un marcador que nos indique cuantos cuadros por segundo se están realizando.

En mi computador *Pentium Celeron 333 Mhz,* Tarjeta Gráfica *Intel 740 8MB* el indicador es de ~77 cuadros por segundos lo que es demasiado poco.... ya veremos otra forma de acelerar el dibujo en pantalla.

Resumen

En el capítulo se resuelve el problema de la velocidad de juego cuando lo ejecutamos en diferentes computadores. Se presenta una técnica simple, pero ampliamente usada para controlar el movimiento de los objetos haciendo uso de las potentes funciones de tiempo de Allegro.

Por el final del capítulo se muestra la forma de conocer los cuadros por segundo que está realizando nuestro juego.

Capítulo 8 :
MANEJO DE LOS CONTROLES

Nuestros juegos serían muy aburridos si dejamos que los actores hagan las cosas por si solos. Después de todo, esperamos *jugar*. Nosotros, omnipotentes, necesitamos ejercer algo de control sobre los actores.

Hasta el momento este proceso fue llevado a cabo dentro del método *update* de la clase *AirCraft*, la cual preguntaba por el estado de algunos *componentes* del *teclado*, si encontraba algún cambio *AirCraft* modificaba sus variables para realizar la *acción* deseada (avanzar, retroceder, etc.) El problema surge cuando queremos cambiar esas teclas que nos permitían movernos. Para hacerlo cambiábamos cuatro sentencias condicionales que consultaban los estados de cuatro teclas, cada una de las cuales representaba una de las cuatro direcciones. Luego de modificarlas debíamos recompilar esa parte del código y enlazarlo. ¿Haz tenido que hacer eso cuando quieres disparar con la tecla *"D"* en vez de la tecla "*Control"* en el Quake?. No.

Todas estos requerimientos los resolveremos en este capítulo, para ello crearemos un *Controlador de Controles (ControlManager)* que administrará tanto los *controles* como los *periféricos*. La diferencia entre controles y periféricos se explicará a continuación.

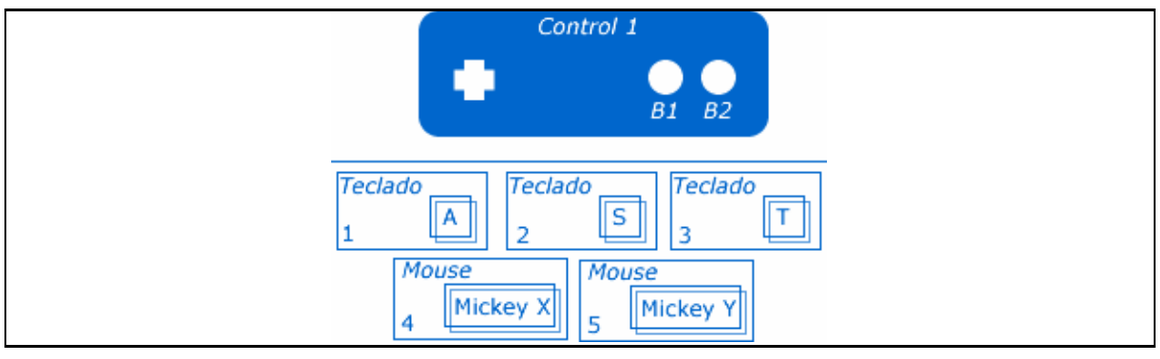
Diferencia entre un control y un periférico

Para la mayoría de los juegos es suficiente que podamos controlar una nave con el teclado, otra nave con un joystick, etc. En este caso los periféricos son el teclado y el joystick, y los controles son el "control para la nave 1" y "control para la nave 2".

Pongámonos en el caso de un juego como Quake en que el control del personaje se realiza con dos periféricos a la vez (para los más experimentados): teclado y Mouse. Mientras que con el teclado podemos agacharnos, movernos por el terreno, abrir puertas; con el Mouse podemos apuntar, disparar y saltar. En este caso el control 1 pide ayuda a dos periféricos.

Por supuesto que nuestro marco de trabajo debe tratar de ser lo más general posible abarcando la mayor cantidad de enfoques en lo que se refiere a los controles de los actores.

Para abstraernos un poco de la situación podríamos pensar que un control es un gran súper-periférico que permite tener *porciones* de otros periféricos dentro de si. Para ilustrarlo, veamos la siguiente figura:

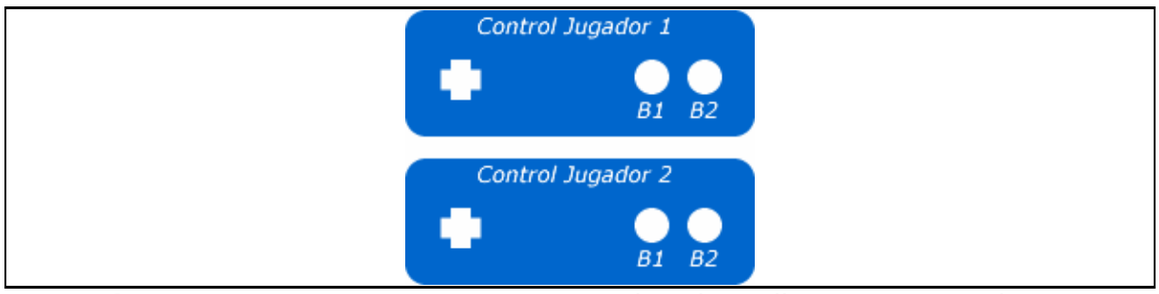


Nuestra abstracción es lo que llamamos el *control 1* (arriba), y este se relaciona con los periféricos mediante la abstracción que llamaremos *asociación*. Por ejemplo, algunas asociaciones que podríamos formar son: B1 con 1 (para 1 el periférico es teclado y el componente de ese periférico es la tecla A) y B2 con 2(periférico teclado y tecla 2), además podríamos asociar los movimientos del control con el mouse: izquierda con 4 (periférico mouse, *mickey x)*. *Mickey* es una variable asociada al mouse que indica cuantos píxeles se movió esté desde la última consulta a su estado.

Cómo trabaja el controlador de controles

La responsabilidad de saber el estado de un periférico ya no recaerá dentro del actor que necesita moverse, sino que la delegaremos a otra clase que llamaremos *control*. Está clase tendrá una lista de asociaciones y preguntará a los objetos *peripheral*, que están dentro de las asociaciones, si sus componentes han cambiado de estado. Si han cambiado entonces *Control* envía al actor correspondiente la acción a realizar, asociado a ese componente del control. ¿Muy complicado?. Vamos lento.

Por ejemplo, en un juego de dos naves existen solo dos controles. Un control para el jugador 1 y otro para el jugador 2. Es decir, por ahora tenemos:



Supongamos que queremos que nuestro jugador 1 dispare con B1. Primero se define qué acción se realizará, en este caso es *disparar*. Luego de define a qué periférico lo asociaremos. Por ejemplo, asociémoslo al periférico teclado. Un tercer miembro en nuestra asociación es el *componente* del periférico, por ejemplo, la tecla A. Finalmente, definiremos con que evento se gatillará esa acción, por ejemplo "*cuando el botón se presione*" (evento conocido en inglés por "*on press"*). Enumeremos entonces los miembros de una asociación:

- **Acción:** Disparar
- **Periférico:** Teclado
- **Componente:** tecla A
- **Evento:** al presionar

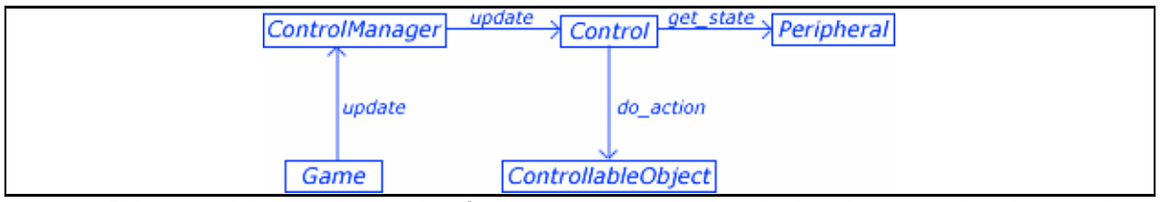
Agregaremos otro campo que describa la asociación:

- **Nombre:** Botón 1 disparo principal

A nuestro control debemos asignarle un propietario, que en nuestro caso es el jugador 1. Este propietario es quien recibirá las órdenes (acciones que debe realizar) enviadas por el control. El jugador es un actor, por supuesto, pero un tipo especial de actor dado que puede recibir órdenes desde un control, también podemos concluir que no sólo un actor puede recibir órdenes desde un control sino también otras clases de objetos. Por ejemplo, un menú puede cambiar cuando nosotros presionemos la tecla *escape*. Llamaremos genéricamente a las clases que pueden recibir órdenes desde un control como *objetos controlables*, creados a partir de la nueva clase *ControllableObject*

Toda esta información estará contenida en lo que llamamos un controlador de controles, tanto los periféricos como los controles. El controlador de controles estará contenido en la clase *Game*.

El proceso en el cual los actores se mueven al presionar una tecla se puede ver en esta figura:



Primero el juego llama al método *update* del controlador de controles, luego el controlador de controles llama al método *update* de todos los controles que tiene en su lista. Cada control pregunta por el estado de los componentes de las asociaciones a los periféricos correspondientes, si cumplen con el evento necesario para realizar la acción, entonces llama al método *do_action* de la clase *ControllableObject* diciéndole que realice la acción respectiva.

Diseño del controlador de controles

Ahora veremos las tarjetas CRC y la implementación final en Allegro del controlador de controles.

Periférico

Clase	Peripheral
Super-Clase	
Subclases	
Responsabilidad	Colaboración
Conocer el estado de todos los componentes.	

Esta responsabilidad que denominamos “conocer el estado de todos los componentes” se refiere en general a distinta información que podemos obtener del periférico, tal cómo: que componentes han cambiado, el estado de *X* componente, etc.

La interfaz de esta clase es la siguiente:

Código Fuente 48. archivo peripheral.h

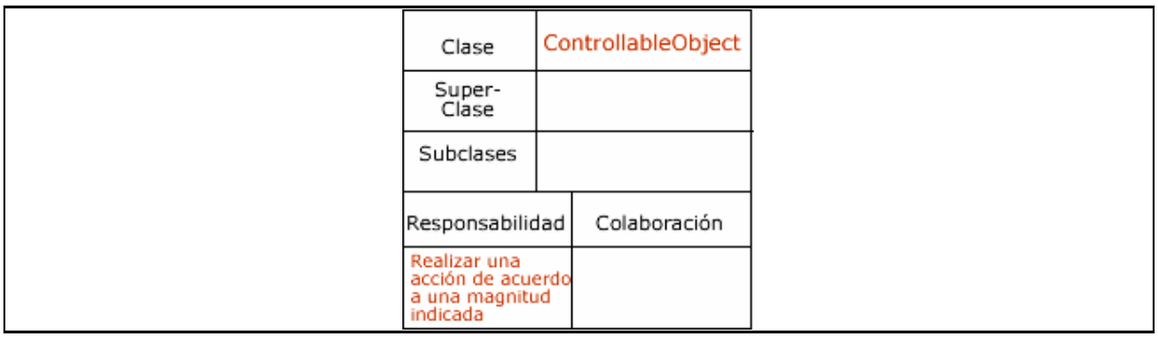
```
#ifndef PERIPHERAL_H
#define PERIPHERAL_H

#include <string>

#define INVALID_STATE        -2
#define INVALID_COMPONENT    -2

class Peripheral
{
public:
    typedef enum event_t
    {
        NO_EVENT,
        ON_PRESS,
        ON_RELEASE,
        ON_PRESSING,
        ON_RELEASSING,
    };
    typedef int state_t;
    typedef int component_t;

    Peripheral();
    virtual state_t get_state(component_t comp);
    virtual component_t get_change();
    virtual string get_component_name(component_t comp);
    virtual void reset();
};

#endif
```

Dentro de la clase se declaran algunos tipos de datos como *state_t* (la letra “t” al final indica que es un tipo de dato, más que una instancia) que representa el estado de algún componente y *component_t* que es para representar un componente.

Los tipos de eventos son representador por *event_t*, dentro de ellos los más importantes: ON_PRESS, cuando una tecla es presionada; y ON_RELEASE, cuando una tecla es soltada.

El método *get_state* retorna el estado del componente *comp*.

Objeto Controlable



La interfaz es muy sencilla:

Código Fuente 49. archivo controllableobject.h

```
#ifndef CONTROLLABLEOBJECT_H
#define CONTROLLABLEOBJECT_H

class ControllableObject
{
public:

     typedef int action_t;
     virtual void do_action(action_t action, int magnitude);
};

#endif
```

Si no queda claro que es la magnitud de una acción, imagina que de acuerdo a la velocidad que mueves el mouse es como el personaje rotará, es decir, la acción es rotar y la magnitud será lo que indica el *mickey* del mouse. Más adelante será más claro mediante un ejemplo.

Control

Clase	Control
Super-Clase	
Subclases	

Responsabilidad	Colaboración
Conocer las asociaciones	Peripheral, ControllableObject
Saber a quien pertenece	ControlableObject
Revisar los periféricos y enviar las acciones asociadas	Peripheral, ControllableObject

Esta clase tiene varias responsabilidades. Primero debe saber todas las asociaciones (acción, periférico, componente y nombre) y permitir que otras clases agreguen nuevas asociaciones, permita establecer a que objeto pertenece el mismo, revisar todas las asociaciones y enviar las acciones al objeto que es dueño del control.

La interfaz es la siguiente:

Código Fuente 50. archivo control.h

```
#ifndef CONTROL_H
#define CONTROL_H

#include <list>
#include <string>
#include "peripheral.h"
#include "controllableobject.h"

class Control
{
public:

    typedef struct
    {
        ControllableObject::action_t act;
        string name;
        Peripheral *peri;
        Peripheral::component_t comp;
        Peripheral::event_t old_event;
        Peripheral::event_t event;
    }association_t;

    void add_action_name(ControllableObject::action_t act, string str);
    void add_association(association_t asoc);
    void set_actionperipheral(ControllableObject::action_t act,
                        Peripheral* peri, Peripheral::component_t comp,
                        Peripheral::event_t e);
    void set_owner(ControllableObject*);
    void update();
    string get_name_action(ControllableObject::action_t);
    void reset();

protected:

    ControllableObject *owner;
    list<association_t> associations;
    list<association_t>::iterator associations_iter;
};
#endif
```

Primero se declara una estructura llamada *association_t* que contiene los componentes de una asociación además de su nombre. Luego viene un método llamado *add_action_name* que solo inserta una acción y un nombre descriptivo de la acción muy útil cuando iniciemos el juego pero no sepamos qué componentes de qué periférico gatillarán tal evento. La siguiente es *add_association* que permite agregar una asociación directamente en el caso de que sepamos todos los componentes de esta. Si agregamos un nombre de acción mediante *add_action_name* esperaríamos luego asociar esa acción a un componente de algún periférico.

El método *set_owner* establece a que objeto se le enviarán las acciones que genere ese control. El objeto dueño está representado por el atributo protegido *ControllableObject *owner*.

El método *update* será llamado por el controlador de controles y permitirá revisar todas las asociaciones para comprobar si hay que enviar alguna orden.

Finalmente *get_name_action* retorna el nombre de la acción

La implementación es la siguiente:

Código Fuente 51. archivo control.cpp

```
#include <allegro.h>
#include "peripheral.h"
#include "control.h"

void Control::add_action_name(ControllableObject::action_t act, string str)
{
    association_t asoc;

    asoc.act = act;
    asoc.name= str;
    asoc.peri= NULL;
    asoc.comp= -1;
    associations.push_back(asoc);
}

void Control::set_actionperipheral(ControllableObject::action_t act,
                                   Peripheral* peri,
                                  Peripheral::component_t comp,
                                  Peripheral::event_t e)
{
    for (associations_iter=associations.begin();
         associations_iter!=associations.end();
         associations_iter++)
    {
        if (associations_iter->act == act)
        {
            associations_iter->peri=peri;
            associations_iter->comp=comp;
            associations_iter->event=e;
            associations_iter->old_event=Peripheral::NO_EVENT;
            return;
        }
    }
}

void Control::add_association(Control::association_t assoc)
{
    associations.push_back(assoc);
}

void Control::set_owner(ControllableObject* co)
{
    owner=co;
}

void Control::update()
{
    int do_action_order;
    Peripheral::state_t tmp_state;
    Peripheral::event_t tmp_old_event;
    for (associations_iter=associations.begin();
        associations_iter!=associations.end();
        associations_iter++)
    {
        tmp_state=associations_iter->peri->get_state(associations_iter->comp);
        tmp_old_event=associations_iter->old_event;
        do_action_order=false;

        switch(associations_iter->event)
        {
        case Peripheral::ON_PRESSING:
            if (tmp_state!=INVALID_STATE)
            {
                do_action_order=true;
            }
            break;
        case Peripheral::ON_PRESS:
            if ((tmp_old_event==Peripheral::ON_RELEASE) &&
                (tmp_state!=INVALID_STATE))
            {
                associations_iter->old_event=Peripheral::ON_PRESS;
                do_action_order=true;
            }
            else
                if (tmp_state==INVALID_STATE) associations_iter-
```

```
    >old_event=Peripheral::ON_RELEASE;
                break;
            case Peripheral::ON_RELEASE:
                if ((tmp_old_event==Peripheral::ON_PRESS) &&
                    (tmp_state==INVALID_STATE))
                {
                    associations_iter->old_event=Peripheral::ON_RELEASE;
                    do_action_order=true;
                }
                else
                    if (tmp_state!=INVALID_STATE) associations_iter-
    >old_event=Peripheral::ON_PRESS;
                break;
            case Peripheral::ON_RELEASSING:
                if (tmp_state==INVALID_STATE)
                {
                    do_action_order=true;
                }
                break;
            default:
                break;
            }

            if (do_action_order)
            {
                owner->do_action(associations_iter->act, tmp_state);
            }
        }
    }

    string Control::get_name_action(ControllableObject::action_t act)
    {
        for (associations_iter=associations.begin();
             associations_iter!=associations.end();
             associations_iter++)
        {
            if (associations_iter->act == act)
            {
                return associations_iter->name;
            }
        }
        return "";
    }
```

Controlador de Controles

Ahora analizaremos la clase que será contenedora de todos los objetos anteriores, sabrá la cantidad de periféricos y controles, y otras cosas que veremos mediante su tarjeta CRC. Dejaré en claro que al agregar un nuevo periférico o control no será posible eliminarlo, esto es porque los periféricos disponibles son generalmente leídos desde un archivo como *allegro.cfg* y se mantienen disponibles durante toda la ejecución de un juego, así también un control se mantiene disponible por todo el juego.

Clase	ControlManager
Super-Clase	
Subclases	

Responsabilidad	Colaboración
Contener todos los periféricos y controles, permitiendo acceder a ellos	Peripheral, Control
Actualizar las asociaciones de los controles	Control
Indicar los cambios que se producen en los periféricos	Peripheral

La primera responsabilidad se implementará mediante cuatro métodos: *add_peripheral, get_peripheral, add_control, get_control*. Los periféricos y controles se guardarán en vectores permitiendo acceder a ellos por su posición dentro del vector, es decir, cuando agreguemos un periférico con el método *add_peripheral* se nos retornará la posición en la cual fue insertado para luego poder obtenerlo mediante *get_peripheral* con la posición que nos fue entregada. La última responsabilidad se refiere a la capacidad que tendremos de configurar los controles sabiendo exactamente que componente de qué periférico ha cambiado, otra característica de la cual no necesitaremos por ahora.

Veamos la interfaz:

Código Fuente 52. archivo controlmanager.h

```
#ifndef CONTROLMANAGER_H
#define CONTROLMANAGER_H

#include <vector>
#include "control.h"
#include "peripheral.h"

#define MAXPERIPHERALS 5

class ControlManager
{
public:
	ControlManager();
	~ControlManager();

	typedef struct
	{
		Peripheral *p;
		Peripheral::component_t comp;
	}change_t;

	change_t get_change();

	int add_control(Control *ctrl);
	int add_peripheral(Peripheral *periph);
	Control *get_control(int number);
	Peripheral *get_peripheral(int number);
	void update();

protected:
	vector<Control*> controls;
	vector<Control*>::iterator controls_iter;
	vector<Peripheral*> peripherals;
	vector<Peripheral*>::iterator peripherals_iter;
	int old_state[MAXPERIPHERALS];
};

#endif
```

Las propiedades *change_t* ,*get_change* y *old_state* (protegido) nos permitirá más adelante hacer el típico menú con el cual configuraremos los controles.

La implementación es la siguiente:

Código Fuente 53. archivo controlmanager.cpp

```
#include <vector>
#include <allegro.h>

#include "peripheral.h"
#include "controlmanager.h"

ControlManager::ControlManager()
{
    for (int i=0;i<MAXPERIPHERALS;i++)
        old_state[i]=FALSE;
}

ControlManager::~ControlManager()
{
    for (peripherals_iter=peripherals.begin();
         peripherals_iter!=peripherals.end();
         peripherals_iter++)
    {
        delete(*peripherals_iter);
    }
    for (controls_iter=controls.begin();
        controls_iter!=controls.end();controls_iter++)
    {
        delete(*controls_iter);
    }
}

int ControlManager::add_control(Control *c)
{
    controls.push_back(c);
    return controls.size()-1;
}

int ControlManager::add_peripheral(Peripheral *p)
{
    peripherals.push_back(p);
    return peripherals.size()-1;
}

Control *ControlManager::get_control(int number)
{
    return controls[number];
}

Peripheral *ControlManager::get_peripheral(int number)
{
    return peripherals[number];
}

ControlManager::change_t ControlManager::get_change()
{
    change_t ret;
    Peripheral::component_t comp;
    int pos=0;

    for (peripherals_iter=peripherals.begin();
         peripherals_iter!=peripherals.end();
         peripherals_iter++)
    {
        pos++;
        comp=(*peripherals_iter)->get_change();
        if (comp!=INVALID_COMPONENT)
        {
            ret.comp=comp;
            ret.p=*peripherals_iter;
            old_state[pos]=TRUE;
        }
        else
            old_state[pos]=FALSE;
    }

    return ret;
}

void ControlManager::update()
```

```
    {
        for (controls_iter=controls.begin();
             controls_iter!=controls.end(); controls_iter++)
             (*controls_iter)->update();
    }
```

Para que todo funcione debemos hacer que nuestra clase *game* delegue la responsabilidad de los controles a *ControlManager* tal como lo hizo anteriormente con *StageManager* y *ActorManager*.

Agregando el nuevo controlador a nuestro marco de trabajo

Hay que hacer cambios básicamente en la clase *game*, agregando un nuevo componente *ControlManager* en las propiedades públicas:

NOTA: Se mostrarán solo los cambios que se han producido en el código de la interfaz e implementación

Código Fuente 54. cambios en archivo game.h

```
    #ifndef GAME_H
    #define GAME_H

    #include <string>
    #include "controlmanager.h" /* <- Cambio */
    ....
    ....
                  ActorManager *actor_manager;
                  StageManager *stage_manager;
                  ControlManager *control_manager; /* <- Cambio */
    ...
    ...
                   void create_actormanager();
                  void create_stagemanager();
                  void create_controlmanager(); /* <- Cambio */
    ...
```

y en la implementación:

Código Fuente 55. cambios en archivo game.cpp

```
Game::Game()
{
...
     control_manager=NULL;
...
}
...
void Game::init(int gfx_mode, int w, int h, int col)
{
...
     create_stagemanager();
     create_controlmanager(); /* <- Cambio */
...
}
...
void Game::shutdown(string message)
{
     delete actor_manager;
     delete stage_manager;
     delete control_manager; /* <- Cambio */
...
}
...
void Game::create_controlmanager()
{
     control_manager = new ControlManager();
}
...
void Game::update()
{
     /* Ciclo logico */
     if (actual_tick<=tick)
    {
        actor_manager->update();
        control_manager->update(); /* <- Cambio */
        actual_tick++;
    }
...
```

Integrando las nuevas características a nuestro juego

Por el momento nosotros necesitamos que nuestra nave sea de tipo *ControllableObject*, es decir, se derivará desde esa clase. Pero nuestra nave no solo es un tipo de objeto controlable sino también es un *actor*, para que sea de ambos tipo ocuparemos la herencia múltiple.

Defensa de la herencia múltiple (o “háganme caso o sugieran algo mejor”)

La herencia múltiple es una característica que tiene C++ y que está ausente en otros lenguajes como Java. Permite que una clase tenga como ancestros a dos clases o más. Hasta el momento estábamos trabajando con la herencia simple, es decir, heredando a partir de una sola clase.

Hay muchos detractores de la herencia múltiple porque le agrega, según ellos, “complejidad *innecesaria*”. Por el momento no me parece innecesaria esta complejidad porque es lógico pensar de la siguiente manera (en el espacio del problema, por supuesto): El jugador es un actor y un objeto controlable. A mi me parece muy natura esa abstracción.

Pero fundamentalmente ocupo la herencia múltiple porque no solo un actor es un tipo de objeto controlable sino un panel o el menú inicial del juego. Espero derivar desde *ControllableObject* tales clases.

De todas formas pensé como hacerlo con herencia simple. Se crearía una clase *ControllableActor* heredada desde *Actor* y se trabajaría con esa clase en *Control* y *ControlManager*. Luego la clase *AirCraft*, más que estar derivada desde las clases *Actor y ContollableObject*, lo sería solo desde *ControllableActor*. Pero si quisiéramos algo parecido para un panel o menú deberíamos crear las clases *ControllablePanel* y *ControllableMenu* en vez de derivarla desde *ControllableObject*. Trabajo de más.

La nueva clase AirCraft

Crearemos una nueva clase para simplificar las cosas que se llamará *ControllableActor* y que no tendrá ningún atributo más que sus ancestros *Actor* y *ControllableObject*:

Código Fuente 56. archivo controllableactor.h

```
#ifndef CONTROLLABLEACTOR_H
#define CONTROLLABLEACTOR_H

#include "actor.h"
#include "controllableobject.h"

class ControllableActor : public Actor, public ControllableObject
{
};

#endif
```

Ahora derivaremos la clase *AirCraft* desde ControllableActor:

Código Fuente 57. clase AirCraft

```
class AirCraft : public ControllableActor
{
     public:

               AirCraft();

               typedef enum
               {
                        DOWN,
                        UP,
                        LEFT,
                        RIGHT
               }action_t;


               void draw(BITMAP *bmp);
               void update();
               void set_image(BITMAP *bmp);
               void do_action(ControllableObject::action_t act, int magnitude);

     protected:

               BITMAP *image;
}
```

Como pueden ver es muy lógico: la clase *AirCraft* es del tipo *ControllableActor* y tiene una nuevo tipo de dato denominado *action_t* que enumera todas las acciones que pueden realizar las instancias de esta clase. Además se sobrescribe el método *do_action* (perteneciente a la clase *ControllableObject*) en la cual serán controlados las acciones descritas en *action_t* (abajo, arriba, izquierda y derecha).

El método *do_action* es el siguiente:

Código Fuente 58. método do_action

```
void AirCraft::do_action(ControllableObject::action_t act, int magnitude)
{

      switch (act)
      {
               case DOWN:
                        y+=4;
                        break;
               case UP:
                        y-=4;
                        break;
               case LEFT:
                        x-=4;
                        break;
               case RIGHT:
                        x+=4;
                        break;
      }
      if (x<0) x=0;
      if (x>SCREEN_W-image->w) x=SCREEN_W-image->w;
      if (y<0) y=0;
      if (y>SCREEN_H-image->h) y=SCREEN_H-image->h;
}
```

Recuerden que el método *update* también es llamado por *actor_manager* en cada ciclo, pero ya no tendrá la responsabilidad de revisar los periféricos, así que por el momento está vacío.

Código Fuente 59. método AirCraft::update

```
void AirCraft::update()
{
}
```

Creando el periférico teclado

Veremos las bondades de la programación orientada al objeto (nuevamente) creando el periférico *Keyboard*. Para ello heredamos y sobrescribimos todos los métodos de la clase *Peripheral*:

Código Fuente 60. archivo keyboard.h

```
#ifndef KEYBOARD_H
#define KEYBOARD_H

#include <allegro.h>
#include "peripheral.h"

class Keyboard : public Peripheral
{
public:
    Keyboard();
    state_t get_state(component_t comp);
    component_t get_change();
    string get_component_name(component_t comp);
    void reset();

protected:
    int old_state[KEY_MAX];
};

#endif
```

Y la implementación es la siguiente:

Código Fuente 61. archivo keyboard.cpp

```
#include <allegro.h>
#include <string>
#include "keyboard.h"

Keyboard::Keyboard()
{
    for (int i=0; i<KEY_MAX;i++) old_state[i]=key[i];
}

Peripheral::state_t Keyboard::get_state(state_t comp)
{
    if (keyboard_needs_poll())
        poll_keyboard();
    if ((comp>=0) && (comp<KEY_MAX) && (key[comp])) return TRUE;
    else
        return INVALID_STATE;
}

Peripheral::component_t Keyboard::get_change()
{
    for (int i=0;i<KEY_MAX;i++)
    {
        if (key[i]!=old_state[i])
        {
            old_state[i]=key[i];
            return i;
        }
    }
    return INVALID_COMPONENT;
}

void Keyboard::reset()
{
   for (int i=0; i<KEY_MAX;i++) old_state[i]=key[i];
}

string Keyboard::get_component_name(component_t comp)
{
    string ret;

    switch (comp) {
        case KEY_A:
            ret="A";
            break;
        case KEY_B:
            ret="B";
            break;
        case KEY_C:
            ret="C";
            break;
        case KEY_D:
...
    return ret;
}
```

El último método *Keyboard::get_component_name* por supuesto que es largísimo porque tiene el nombre de todas las teclas (componentes).

De esta misma forma es posible crear un periférico *Joystick o Mouse*. Eso podría ser una tarea para ti.

Combinando todo y agregándolo al juego TestFrameWork

Son agregadas algunas interfaces al archivo *testframework.cpp*:

Código Fuente 62. cambio en cabeceras testframework.cpp

```
#include <allegro.h>
#include "game.h"
#include "actor.h"
#include "actormanager.h"
#include "controllableactor.h"
#include "control.h"
#include "keyboard.h"
#include <ctime>
#include <cstdlib>
```

Y agregamos tanto el control para la instancia de *AirCraft* como el periférico desde el cual son generadas las órdenes. Todo esto, por el momento, se realizará en el método *TestFrameWork::main*.

Creamos una instancia de *Control* y agregamos las acciones básica de la siguiente manera:

Código Fuente 63. creación de un Control

```
Control *control_p1=new Control;

control_p1->add_action_name(AirCraft::DOWN, "Bajar");
control_p1->add_action_name(AirCraft::UP,   "Subir");
control_p1->add_action_name(AirCraft::LEFT, "Izquierda");
control_p1->add_action_name(AirCraft::RIGHT,"Derecha");
```

Agregamos cuatro acciones al control y un nombre para cada acción. La identificación de las acciones las obtenemos desde la clase *AirCraft* porque para objetos de ese tipo es creado este control.

Creamos un periférico para asociar los componentes de este a las acciones correspondientes:

Código Fuente 64. creacion de un periférico keyboard

```
Keyboard *kboard=new Keyboard;
```

Tenemos creada la instancia de un teclado *kboard* y hacemos las asociaciones correspondientes:

Código Fuente 65. creación de las asociaciones

```
control_p1->set_actionperipheral(AirCraft::DOWN, kboard, KEY_DOWN,
Peripheral::ON_PRESSING);
control_p1->set_actionperipheral(AirCraft::UP, kboard, KEY_UP,
Peripheral::ON_PRESSING);
control_p1->set_actionperipheral(AirCraft::LEFT, kboard, KEY_LEFT,
Peripheral::ON_PRESSING);
control_p1->set_actionperipheral(AirCraft::RIGHT, kboard ,KEY_RIGHT,
Peripheral::ON_PRESSING);
```

La explicación es muy sencilla, por ejemplo, la acción *AirCraft::DOWN* es ordenada cuando el componente *KEY_DOWN* del periférico *kboard* responde al evento *ON_PRESSING* (algo así como “mientras esté presionado”). Por el momento todas las acciones están asociadas a componentes de un solo periférico, pero potencialmente se podrían combinar varios periféricos en un solo control.

Finalmente se le da un dueño a este control y se agregan tanto el control como el periférico al controlador de controles.

Código Fuente 66. dando un dueño a control_p1

```
AirCraft *a=new AirCraft;
...
control_p1->set_owner(a);
control_manager->add_control(control_p1);
control_manager->add_peripheral(kboard);
```

El método *TestFrameWork::main* queda de la siguiente manera:

Código Fuente 67. método TestFrameWork::main

```
    void TestFrameWork::main()
    {
          BITMAP *bmp;
          PALETTE tmp;
          AirCraft *a=new AirCraft;
          Star *star_tmp;

          Control *control_p1=new Control;

          control_p1->add_action_name(AirCraft::DOWN, "Bajar");
          control_p1->add_action_name(AirCraft::UP, "Subir");
          control_p1->add_action_name(AirCraft::LEFT, "Izquierda");
          control_p1->add_action_name(AirCraft::RIGHT, "Derecha");

          Keyboard *kboard=new Keyboard;

         control_p1->set_actionperipheral(AirCraft::DOWN, kboard, KEY_DOWN,
    Peripheral::ON_PRESS);
         control_p1->set_actionperipheral(AirCraft::UP, kboard, KEY_UP,
    Peripheral::ON_PRESS);
         control_p1->set_actionperipheral(AirCraft::LEFT, kboard, KEY_LEFT,
    Peripheral::ON_PRESSING);
         control_p1->set_actionperipheral(AirCraft::RIGHT, kboard ,KEY_RIGHT,
    Peripheral::ON_PRESSING);

          control_p1->set_owner(a);
          control_manager->add_control(control_p1);
          control_manager->add_peripheral(kboard);

          for (int i=0; i<100;i++)
          {
                   star_tmp=new Star;
                   star_tmp->set_y(rand()%SCREEN_H);
                   actor_manager->add(star_tmp);
          }

          bmp=load_bitmap("nave.pcx", tmp);
          a->set_image(bmp);
          a->set_x(SCREEN_W/2);
          a->set_y(SCREEN_H/2);
          actor_manager->add(a);
          while (!key[KEY_ESC]) update();
          destroy_bitmap(bmp);
    }
```

Resumen

En este capítulo vimos conceptos muy importantes dentro de un marco de trabajo: el manejo de los controles. La abstracción que logramos a través de estos párrafos nos permitirá más adelante comprender completamente la forma en como podemos ofrecer una alternativa a la implementación de esta clase de sistema vistos desde los *eventos*. Más que eventos, y en pos de la velocidad de ejecución del marco de trabajo, nos enfocamos en las acciones que envían los controles a los actores, y en generalmente, a los objetos que son *controlables*.

Siempre pensando en la implementación inmediata de los conceptos aprendidos, para lograr una comprensión eficaz y rápida, agregamos estas clases como controladores a nuestra clase *game*. De esta forma ya, en el capítulo actual, hemos delegado las responsabilidades de administración de actores, de la escena (dibujo y disposición de objetos) y de los controles.

Capítulo 9 :
SEPARACIÓN DE GRÁFICA Y CONTROL

Siguiendo con la idea de delegar funciones a clases específicas o especialistas es que ahora analizaremos y diseñaremos una forma de crear en forma separada la vista del actor.

La vista del actor es la *representación gráfica* que se dibujará en la pantalla, es *la parte gráfica del actor*. Esta abstracción nos permite dividir muy bien lo que es la vista (lo que finalmente se muestra) y el control (el comportamiento del actor), un paradigma muy usado en informática. De esta forma podemos darle una nueva cara a la clase *AirCraft* sin necesidad de modificar la clase misma sino solo su vista. Por ejemplo, para hacer un juego más simple, podríamos hacer que *AirCraft* no fuera un mapa de bits sino un simple cuadrado dibujado mediante el procedimiento Allegro *rect*.

El nombre que crearemos para la clase que se encarga de la representación gráfica será *ActorGraphic*.

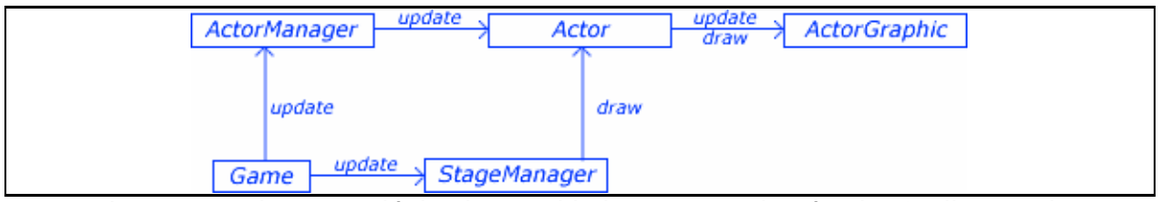
Análisis y diseño de la gráfica del actor

Veamos el siguiente requerimiento para nuestra clase: “La gráfica de un actor se encarga de la forma en que se dibujará este en la pantalla y también, potencialmente, de como se animará este dibujo.” Podemos obtener la siguiente tarjeta CRC:

Clase	ActorGraphic
Super-Clase	
Subclases	
Responsabilidad	Colaboración
1.- Dibujar al actor en pantalla 2.- Conocer la información del Actor	Actor

De esta forma la clase *Actor* delega la responsabilidad de dibujarse a la nueva clase *ActorGraphic*, es decir, dentro del método *Actor::draw* se llamará al nuevo método que crearemos *ActorGraphic::draw*. La segunda responsabilidad es la tener conocimiento acerca del *Actor* que está dibujando *ActorGraphic*, esto lo haremos creando una referencia (un puntero) a *Actor* desde *ActorGraphic*.

El proceso de actualización de la sección gráfica ahora es el siguiente:



Lo que en el proceso lógico ha cambiado es que, además al actualizarse el actor, se llama al método *update* de su representación gráfica. También ha cambiado el ciclo gráfico porque el actor llama al método *draw* de su representación gráfica cuando realiza su propio método *draw*.

Implementación de la clase ActorGraphic

La interfaz es la siguiente:

Código Fuente 68. archivo actorgraphic.h

```
#ifndef ACTORGRAPHIC_H
#define ACTORGRAPHIC_H

#include <allegro.h>

class Actor;

class ActorGraphic
{
    public:

        ActorGraphic(Actor *a);
        virtual ~ActorGraphic();

        virtual void init();
        virtual void update();
        virtual void draw(BITMAP *bmp);
        virtual int get_w();
        virtual int get_h();
        virtual int get_x();
        virtual int get_y();

    protected:
        Actor *owner;
};

#endif
```

El constructor establece cual es el *Actor* que está siendo representado por un objeto de tipo *ActorGraphic*.

La implementación:

Código Fuente 69. archivo actorgraphic.cpp

```
#include <allegro.h>
#include "actorgraphic.h"
#include "actor.h"

ActorGraphic::ActorGraphic(Actor *a)
{
    owner=a;
}

ActorGraphic::~ActorGraphic()
{
}

void ActorGraphic::update()
{
}

void ActorGraphic::init()
{
}

void ActorGraphic::draw(BITMAP *bmp)
{
}

int ActorGraphic::get_w()
{
    return 0;
}

int ActorGraphic::get_h()
{
    return 0;
}

int ActorGraphic::get_x()
{
    return (int)(owner->get_x());
}

int ActorGraphic::get_y()
{
    return (int)(owner->get_y());
}
```

Por el momento la clase *ActorGraphic* no dibujará nada en pantalla y establecerá su ancho y largo (gráfico) en cero, su posición (x,y) es la misma del Actor al cual representa. Los métodos *init* y *update* quedarán más claros adelante cuando implementemos la vista de una animación.

Modificaciones necesarias en el marco de trabajo

Modificación de la clase Actor

La clase Actor debe sufrir algunas modificaciones para permite la separación del control y la vista. El control en este caso está representado por el método *update* de control que permite realizar toda la lógica del objeto. Para la implementación de la vista se agrega un nuevo atributo *agraph* (una instancia de *ActorGraphic*) y el método correspondiente para establecer el gráfico *set_actor_graphic*.

Ya que dentro del controlador de actores (*ActorManager*) no podremos tener información del objeto que represente su parte gráfica, la clase *Actor* es la que deberá darnos esa información. Los métodos que agregaremos a la clase *Actor* son los siguientes:

- *get_w*: retorna el ancho gráfico del actor
- *get_h:* retorna la altura gráfica del actor
- *get_graph_x:* retorna la posición x desde donde empieza la gráfica del actor
- *get_graph_y:* retorna la posición y desde donde empieza la gráfica del actor

- *set_actor_graphic:* Establece la parte gráfica del actor

Otra modificación que realizaremos es al método *update* del Actor. Antiguamente en este método nosotros realizábamos la lógica del actor. Ahora el nombre del método que nos permitirá realizar esta tarea es *move* ya que *update* llamará a *ActorGraphic::update* y despues a *move*, es decir, realiza la actualización tanto de la lógica de la gráfica como del mismo Actor. Pero tu te preguntarás, ¿Exxiste una parte lógica de una clase netamente gráfica como *ActorGraphic*?. Sí.

Supongamos que creamos una clase llamada *Sprite* heredada desde *ActorGraphic* que nos permita representar una animación de varios cuadros e imágenes. La contabilización del tiempo y la determinación del cuadro correcto que debe desplegarse en pantalla cuando se llame a *Sprite::draw* debe ser realizada dentro del método *Sprite::update*. No te preocupes, luego implementaremos esta clase para que te quede más claro cual puede ser el proceso lógico dentro de *ActoGraphic*.

Creamos además un nuevo método *init* que permite inicializar todas las cosas del Actor antes de entrar en escena. Su utilización puede quedar suprimida por el constructor, sin embargo *init* es llamado cuando el Actor ingresa al controlador de controles a diferencia del constructor que se llama cuando se crea la instancia (el objeto).

De esta forma, la interfaz de la clase *Actor* queda de la siguiente manera:

Código Fuente 70. archivo actor.h

```
#ifndef ACTOR_H
#define ACTOR_H

#include <allegro.h>
#include "game.h"
#include "actorgraphic.h"

class Actor
{
    public:

        Actor();
        virtual ~Actor();

        virtual void draw(BITMAP *bmp);
        virtual void update();
        virtual void move();
        virtual void init();
        virtual int get_w();
        virtual int get_h();

        void set_x(int pos_x);
        void set_y(int pos_y);
        void set_actor_graphic(ActorGraphic *ag);
        int get_x();
        int get_y();
        int get_graph_x();
        int get_graph_y();

    protected:

        int x, y;
        ActorGraphic *agraph;
};

#endif ACTOR_H
```

Y la implementación:

Código Fuente 71. archivo actor.cpp

```
#include "actor.h"

Actor::Actor()
{
    agraph=NULL;
}

Actor::~Actor()
{
    if (agraph) delete agraph;
}

void Actor::draw(BITMAP *bmp)
{
    agraph->draw(bmp);
}

void Actor::update()
{
    agraph->update();
    move();
}

void Actor::init()
{
    agraph->init();
}

void Actor::move()
{
}

void Actor::set_x(int pos_x)
{
    x=pos_x;
}

void Actor::set_y(int pos_y)
{
    y=pos_y;
}

void Actor::set_actor_graphic(ActorGraphic *ag)
{
    agraph=ag;
}

int Actor::get_x()
{
    return x;
}
int Actor::get_y()
{
    return y;
}

int Actor::get_w()
{
    return agraph->get_w();
}

int Actor::get_h()
{
    return agraph->get_h();
}

int Actor::get_graph_x()
{
    return agraph->get_x();
}

int Actor::get_graph_y()
{
    return agraph->get_y();
}
```

Modificación de ActorManager

ActorManager necesita la pequeña modificación al agregar un *Actor* al juego, una llama a *Actor::init*. Entonces, en el método *ActorManager::Add*:

Código Fuente 72. modificación de actormanager.cpp

```
...
void ActorManager::add(Actor *a)
{
    actors.push_back(a);
    a->init(); // <- cambio
}
...
```

Creación de distintos gráficos para actores

Ahora viene la parte más interesante, ya que podremos crear distintas representaciones gráficas para los actores. Partamos por el actor más simple que creamos en el último juego: la clase *Star*. Recordemos su interfaz e implementación:

Código Fuente 73. antigua interfaz e implementación de la clase Star

```
class Star : public Actor
{
    public:

        Star();

        void update();
        void draw(BITMAP *bmp);
    protected:

        void reinit();
        int vy;
};

Star::Star()
{
    reinit();
}

void Star::reinit()
{
    x=rand()%SCREEN_W;
    y=0;
    vy=1+rand()%8;
}

void Star::update()
{
    y+=vy;
    if (y>SCREEN_H)
    {
        reinit();
    }
}

void Star::draw(BITMAP *bmp)
{
    putpixel(bmp,x,y,makecol(255,255,255));
}
```

Ahora esta clase solo realizará el control del actor, es decir, definirá el *comportamiento* de una estrella.

Lo escencial en la estrella es que siempre bajaba a una velocidad constante que se definía en la creación y cada vez que llegaba al borde inferior de la pantalla, la parte gráfica estará descrita por una especialización de *ActorGraphic*.

A partir de *ActorGraphic* heredaremos una nueva clase *Pixel*. La interfaz e implementación es la siguiente:

Código Fuente 74. clase Pixel

```
class Pixel : public ActorGraphic
{
    public:
        Pixel(Actor *aowner, int col);
        void draw(BITMAP *bmp);
        int get_w();
        int get_h();
    protected:
        int color;
};

Pixel::Pixel(Actor *aowner, int col) : ActorGraphic(aowner)
{
    color=col;
}

void Pixel::draw(BITMAP *bmp)
{
    putpixel(bmp, get_x(), get_y(), color);
}

int Pixel::get_w()
{
    return 1;
}

int Pixel::get_h()
{
    return 1;
}
```

Como pueden ver en el constructor de *Pixel*, además de definir a qué actor corresponde esta representación gráfica, se define el color del píxel. En el método *draw* se llama al procedimiento Allegro *putpixel* que dibuja un pixel en mapa de bits *bmp* en la posición *get_x* y *get_y*. Recordemos que al llamar *get_x* y *get_y* en realidad estamos llamando a la implementación de *ActorGraphic::get_x* y *ActoGraphic::get_y* que retornaban la posición del actor dueño de esta representación gráfica:

Código Fuente 75.

```
...
int ActorGraphic::get_x()
{
    return (int)(owner->get_x());
}

int ActorGraphic::get_y()
{
    return (int)(owner->get_y());
}
...
```

Ahora realicemos el cambio a la clase *Star*:

Código Fuente 76. nueva interfaz e implementación de la clase Star

```
class Star : public Actor
{
    public:
        Star();
        void move();
    protected:
        void reinit();
        int vy;
};

Star::Star()
{
    reinit();
}

void Star::reinit()
{
    x=rand()%SCREEN_W;
    y=0;
    vy=1+rand()%8;
}

void Star::move()
{
    y+=vy;
    if (y>SCREEN_H)
    {
        reinit();
    }
}
```

En vez de sobrescribir el método *update* se sobrescribe *move* para realizar el movimiento de la estrella. Como ven, nada gráfico.

Ahora al crear una instancia de *Star* también debemos asignarle una representación gráfica:

Código Fuente 77.

```
...
Star *star_tmp;
Pixel *pixel_tmp;
for (int i=0; i<100;i++)
{
    star_tmp=new Star();
    pixel_tmp=new Pixel(star_tmp, makecol(255, 255, 255));
    star_tmp->set_actor_graphic(pixel_tmp);
    star_tmp->set_y(rand()%SCREEN_H);
    actor_manager->add(star_tmp);
}
...
```

Primero creamos los punteros *star_tmp* y *pixel_tmp* luego, en cada iteración, creamos una instancia de estrella y luego una instancia de *Pixel*. El parámetro de pixel es la estrella que acabamos de instanciar. Para crear el color llamamos al procedimiento Allegro *makecol* que toma tres valores en RGB como (0-255, 0-255, 0-255). El color (255, 255, 255) es el blanco (todo el rojo, todo el verde y todo el azul). Finalmente le decimos al Actor que su representación gráfica es la instancia de *Pixel* con *star_tmp->set_actor_graphic(pixel_tmp)*. Lo demás es conocido.

Continuemos entonces con la creación de nuevas representaciones gráficas para la nave.

Clase Bitmap

La clase *bitmap* la definiremos como una representación gráfica de mapa de bits. Para ir rápido, la interfaz e implementación:

Código Fuente 78. clase Bitmap

```
class Bitmap : public ActorGraphic
{
    public:
        Bitmap(Actor *aowner, BITMAP *bmp);
        void draw(BITMAP *bmp);
        int get_w();
        int get_h();
    protected:
        BITMAP *bitmap;
};

Bitmap::Bitmap(Actor *aowner, BITMAP *bmp) : ActorGraphic(aowner)
{
    bitmap=bmp;
}

void Bitmap::draw(BITMAP *bmp)
{
    draw_sprite(bmp, bitmap, get_x(), get_y());
}

int Bitmap::get_w()
{
    return bitmap->w;
}

int Bitmap::get_h()
{
    return bitmap->h;
}
```

Ahora en el constructor en vez de definir un simple color, como en la clase *Pixel*, definimos el *BITMAP* que se dibujará en pantalla.

Para que pueda ser ocupado por nuestra nave, debemos eliminar los métodos gráficos y dejar solo los lógicos en la clase *AirCraft*.

Código Fuente 79. clase AirCraft

```
class AirCraft : public ControllableActor
{
    public:

        AirCraft();

        typedef enum
        {
            DOWN,
            UP,
            LEFT,
            RIGHT
        }action_t;

        void do_action(ControllableObject::action_t act, int magnitude);
};

AirCraft::AirCraft()
{
}

void AirCraft::do_action(ControllableObject::action_t act, int magnitude)
{

    switch (act)
    {
        case DOWN:
            y+=4;
            break;
        case UP:
            y-=4;
            break;
        case LEFT:
            x-=4;
            break;
        case RIGHT:
            x+=4;
            break;
    }
    if (x<0) x=0;
    if (x>SCREEN_W-get_w()) x=SCREEN_W-get_w();
    if (y<0) y=0;
    if (y>SCREEN_H-get_h()) y=SCREEN_H-get_h();
}
```

¡Muy sencilla!. Ahora la creación de la instancia y la asignación de la representación gráfica *Bitmap* es un tanto distinta:

Código Fuente 80.

```
...
AirCraft *airc=new AirCraft();
BITMAP *bmp;
bmp=load_bitmap("nave.pcx", NULL);
Bitmap *bitm=new Bitmap(airc, bmp);
airc->set_actor_graphic(bitm);
actor_manager->add(airc);
...
```

Al llamar al constructor de *Bitmap* le asignamos el nuevo mapa de bits que cargamos desde un archivo. Lo demás es trivial.

Probando el potencial de las representaciones gráficas

Siempre es bueno, al enseñar una materia, dejar a la audiencia (Uds.) sorprendidos por lo que han aprendido, decir “¡He creado un monstruo!” es la meta mía. Es por eso que ahora Uds. dirán “eso no puedo haberlo creado yo”, pero así ha sido.

A través de estos capítulos hemos llegado a un motor de juego algo robusto y fácil de expandir a base de rigurosidad y sin asco al admitir que me he equivocado con la consecuencia de rediseñar grandes secciones de código, ahora solo estamos recogiendo los frutos.

No es muy atractivo que la nave este quieta todo el rato, se necesita algo de movimiento. Para empezar, le pondré unas llamas saliendo de sus turbinas, dos sencillos movimientos. ¿Como realizarlo? Pensemos que una animación es otra representación gráfica

Clase Sprite

Un *Sprite* será para nosotros (por el momento) una secuencia de mapas de bits que se suceden con distintos tiempos entre si. El conjunto de mapa de bits y tiempo, lo llamaremos *Cuadro*. Veamos la interfaz:

Código Fuente 81. interfaz de Sprite

```
class Sprite : public ActorGraphic
{
    public:
        Sprite(Actor *aowner);
        void draw(BITMAP *bmp);
        void update();
        void init();
        int get_w();
        int get_h();
        int get_x();
        int get_y();
        void add_frame(BITMAP *bmp, int cx, int cy, int ticks);
    protected:
        typedef struct Frame
        {
            BITMAP *bmp;
            int cx;
            int cy;
            int ticks;
        };
        vector<Frame> frames;
        int actual_tick, actual_frame;
};
```

Se ha creado una nueva estructura de datos *Frame* que representa un cuadro dentro de la animación. Este cuadro esta conformado por un mapa de bits, la posición del centro (*cx* y *cy*) y la cantidad de *ticks* que es mostrado ese cuadro.

Los cuadros en el *Sprite* son guardados en un vector de cuadros, manteniéndose también como atributos el conteo actual de *ticks* (*actual_tick*) y el cuadro actual que está siendo desplegado (*actual_frame*). Para agregar un cuadro sólo llamamos al método *add_frame* con los parámetros correspondientes a cada atributo de *Frame*.

En esta clase, por ejemplo, se hace uso del método *init* el cual reiniciará la animación y se sobrescriben los métodos *get_x* y *get_y* porque ya no representarán la misma posición del actor propietario.

La implementación es la siguiente:

Código Fuente 82. implementación de Sprite

```
Sprite::Sprite(Actor *aowner) : ActorGraphic(aowner)
{
}

void Sprite::init()
{
    actual_frame=0;
    actual_tick=0;
}

void Sprite::add_frame(BITMAP *bmp, int cx, int cy, int ticks)
{
    Frame tmp_frame;
    tmp_frame.bmp=bmp;
    tmp_frame.cx=cx;
    tmp_frame.cy=cy;
    tmp_frame.ticks=ticks;
    frames.push_back(tmp_frame);
}


void Sprite::update()
{
    if (frames.size()<=1) return;
    if (actual_tick>frames[actual_frame].ticks)
    {
        actual_tick=0;
        actual_frame++;
        if (actual_frame>=(int)frames.size()) actual_frame=0;
    }
    else
    {
        actual_tick++;
    }
}

void Sprite::draw(BITMAP *bmp)
{
    draw_sprite(bmp, frames[actual_frame].bmp, get_x(), get_y());
}

int Sprite::get_w()
{
    return frames[actual_frame].bmp->w;
}

int Sprite::get_h()
{
    return frames[actual_frame].bmp->h;
}

int Sprite::get_x()
{
    return owner->get_x()-(frames[actual_frame].cx);
}

int Sprite::get_y()
{
    return owner->get_y()-(frames[actual_frame].cy);
}
```

Por ejemplo, crearemos una nueva representación gráfica para la nave que tu controlas con dos mapas de bits “*nave1.pcx*” y “*nave2.pcx*”. La creación se hace de la siguiente manera:

Código Fuente 83.

```
    BITMAP *bmp1, *bmp2;
    AirCraft *airc=new AirCraft();

    bmp1=load_bitmap("nave1.pcx", NULL);
    bmp2=load_bitmap("nave2.pcx", NULL);
    Sprite *sp=new Sprite(airc);
    sp->add_frame(bmp1, bmp1->w/2,     bmp1->h/2,     20);
    sp->add_frame(bmp2, bmp2->w/2,     bmp2->h/2,     20);
    airc->set_x(SCREEN_W/2);
    airc->set_y(SCREEN_H/2);
    airc->set_actor_graphic(sp);
    actor_manager->add(airc);
```

Primero se crea la instanca de *AirCraft*, luego se cargan los mapas de bits. Se crea la instancia de *Sprite* y se asigna de propietario la instancia de *AirCraft airc*. Se agregan los cuadros y se le da un tiempo de 20 *ticks* a cada uno. Dado que el juego se ejecuta a 70 cuadros por segundo, entonces 20 *ticks* son aproximadamente 30 centésimas de segundo.

El código fuente final de la nueva prueba del marco de trabajo es:

Código Fuente 84. archivo testframework.cpp

```
#include <ctime>
#include <cstdlib>
#include <allegro.h>
#include "game.h"
#include "actor.h"
#include "actormanager.h"
#include "controllableactor.h"
#include "control.h"
#include "keyboard.h"
#include "actorgraphic.h"

class Sprite : public ActorGraphic
{
    public:
        Sprite(Actor *aowner);
 ...
};

Sprite::Sprite(Actor *aowner) : ActorGraphic(aowner)
{
}
...

class Pixel : public ActorGraphic
{
    public:
        Pixel(Actor *aowner, int col);
...
};

Pixel::Pixel(Actor *aowner, int col) : ActorGraphic(aowner)
{
    color=col;
}
...
class AirCraft : public ControllableActor
{
    public:
        AirCraft();
...
};

AirCraft::AirCraft()
{
}
...
class Star : public Actor
{
    public:
        Star();
...
};

Star::Star()
{
    reinit();
}
...

class TestFrameWork : public Game
{
    public:
        void main();
};

void TestFrameWork::main()
{
    BITMAP *bmp1, *bmp2;
    AirCraft *airc=new AirCraft();

    Control *control_p1=new Control;

    control_p1->add_action_name(AirCraft::DOWN, "Bajar");
    control_p1->add_action_name(AirCraft::UP, "Subir");
    control_p1->add_action_name(AirCraft::LEFT, "Izquierda");
```

```
        control_p1->add_action_name(AirCraft::RIGHT, "Derecha");

        Keyboard *kboard=new Keyboard;


        control_p1->set_actionperipheral(AirCraft::DOWN, kboard, KEY_DOWN,
    Peripheral::ON_PRESSING);
        control_p1->set_actionperipheral(AirCraft::UP, kboard, KEY_UP,
    Peripheral::ON_PRESSING);
        control_p1->set_actionperipheral(AirCraft::LEFT, kboard, KEY_LEFT,
    Peripheral::ON_PRESSING);
        control_p1->set_actionperipheral(AirCraft::RIGHT, kboard ,KEY_RIGHT,
    Peripheral::ON_PRESSING);

        control_p1->set_owner(airc);
        control_manager->add_control(control_p1);
        control_manager->add_peripheral(kboard);

        Star *star_tmp;
        Pixel *pixel_tmp;
        for (int i=0; i<100;i++)
        {
            star_tmp=new Star();
            pixel_tmp=new Pixel(star_tmp, makecol(255, 255, 255));
            star_tmp->set_actor_graphic(pixel_tmp);
            star_tmp->set_y(rand()%SCREEN_H);
            actor_manager->add(star_tmp);
        }

        bmp1=load_bitmap("nave1.pcx", NULL);
        bmp2=load_bitmap("nave2.pcx", NULL);
        Sprite *sp=new Sprite(airc);
        sp->add_frame(bmp1, bmp1->w/2,      bmp1->h/2,      20);
        sp->add_frame(bmp2, bmp2->w/2,      bmp2->h/2,      20);
        airc->set_x(SCREEN_W/2);
        airc->set_y(SCREEN_H/2);
        airc->set_actor_graphic(sp);
        actor_manager->add(airc);

        while (!key[KEY_ESC]) update();
        destroy_bitmap(bmp1);
        destroy_bitmap(bmp2);
    }

    int main()
    {
        TestFrameWork game;
        srand(time(NULL));
        game.set_name("Test del Marco de Trabajo");
        game.init(GFX_AUTODETECT, 640,480,16);
    }
```

Podrías tratar de hacer una nueva representación gráfica para las estrellas. Por ejemplo, unos círculos con el procedimiento *circle* de Allegro. Tómalo como una “tarea”.

Resumen

En este capítulo hemos dado un paso más en la flexibilidad del marco de trabajo: la separación de la vista y el control. Dado que este paradigma nos permite una facilidad de cambio, de reutilización y extensibilidad es que nuestro marco de trabajo gozará de estos mismos beneficios.

Primero partimos con una pequeña implementación de una representación gráfica de *Pixel* para luego terminar con una animación de varios cuadros. Las posibilidades son muchas sin perjuicio de la eficiencia en la ejecución. El próximo será la creación de un compilado de representaciones gráficas: píxeles que cambian de color, rectángulos, círculos, mapas de bits que cambiad de tamaño o rotan, etc.

Capítulo 10 : Detección de Colisiones

Una de las características más importantes en nuestro marco de trabajo se refiere a la detección de colisiones ya que nos permitirá saber a qué actores estamos atacando o que actores nos están atacando, por supuesto en los juegos en que lo requiramos.

La detección de colisiones se suma a la lista de administradores que tiene nuestro marco de trabajo, liberando de esta forma al objeto *Game* de la tarea de dar por ganador a uno u otro actor mediante las colisiones. El objeto *Game* entonces delegará esta función a un nuevo objeto de la clase *CollisionManager*.

La forma en que se tomarán decisiones al producirse una colisión es a través de una nueva interfaz de la clase *Actor* que será invocada que se *detecte una colisión*.

Análisis y diseño del administrador de colisiones

Ordenando nuestras ideas advertimos que lo único que hará el administrador de Actores será intersectar en parejas a todos los actores pertenecientes al juego. Si la intersección es verdadera entonces se llamará a un método del actor que le avisará *quien* ha chocado con él y *cuanto daño* le ha causado. Veamos entonces la tarjeta CRC de la nueva clase *CollisionManager*:

Clase	CollisionManager
Super-Clase	
Subclases	
Responsabilidad	**Colaboración**
Intersectar la parte grafica de todos los actores del controlador de actores	Actor, ActorManager

En realidad el administrador de colisiones necesitará la colaboración de la clase *Game* ya que finalmente desde ella obtendrá el controlador de actores.

En el siguiente punto veremos las modificaciones necesarias al marco de trabajo para que, en el caso que no necesitemos un administrador de colisiones, lo podamos personalizar correctamente.

Ocupando solo los administradores necesarios

Muchos de Uds. se preguntarán que hacer en el caso de que nuestro juego no necesite de un administrador de colisiones o quizá de un administrador de controles como, por ejemplo, un juego de Ajedrez (valioso aporte y discusión de Juan José Pastor). Para ello se ha ideado una forma de personalizar el marco de trabajo de tal forma de no crear estos controladores y, por supuesto, de no invocarlos.

El paradigma que seguimos para agregar controladores al juego es el de definir 3 cosas en la clase *Game*, la que delega tareas a los controladores): primero, un método que cree el controlador (con nombre al estilo de *create_controlmanager, create_actormanager*, etc.); segundo, agregar una línea al método *Game::update* de tal forma de invocar al controlador; y tercero, de destruir el controlador en el método *Game::shutdown*.

En este capítulo agregaremos un administrador de colisiones, pero supongamos que en un potencial juego no queramos ocuparlo (juego de cartas, damas, dados, etc.). La técnica es personalizar nuestro juego, mediante la herencia, y sobrescribir el método que crea el controlador. Para sobrescribir un método necesitamos que la clase ancestral lo declare de tipo *virtual*. Entonces modifiquemos la interfaz de nuestra clase *Game* para que así sea:

Código Fuente 85. cambios en la clase Game

```
...
        virtual void create_actormanager();
        virtual void create_stagemanager();
        virtual void create_controlmanager();
        virtual void create_collisionmanager();
...
```

Ahora en el método *Game::update* nos preocuparemos de que realmente esté creado el controlador para invocarlo, es decir, que tenga memoria asignada:

Código Fuente 86. modificación del método Game::update

```
...
    if (actual_tick<=tick)
    {
        actor_manager->update();
        // Comprobación para collision_manager
        if (collision_manager) collision_manager->update();
        // Coprobación para control_manager
        if (control_manager) control_manager->update();
        actual_tick++;
    }
...
```

Además asegurémonos de que, en caso de no crearse el controlador, estos objetos apunten a *NULL*, algo que ya se había previsto, pero de todas formas lo mostraremos:

Código Fuente 87. constructor de la clase Game

```
Game::Game()
{
    allegro_init();
    install_keyboard();
    actor_manager=NULL;
    stage_manager=NULL;
    control_manager=NULL;
    collision_manager=NULL;
...
```

Finalmente agregamos las comprobaciones necesarias al método *Game::shutdown*:

Código Fuente 88. comprobación de que fuerón creados los controladores antes de destruirlos

```
void Game::shutdown(string message)
{
    if (actor_manager) delete actor_manager;
    if (stage_manager) delete stage_manager;
    if (control_manager) delete control_manager;
    if (collision_manager) delete collision_manager;
...
```

*Nota: aunque en el C++ estándar cuando si un puntero es NULL y es destruido con la instrucción **delete**, esta no produce ningún efecto colateral, pero prefiero asegurarme sabiendo que muchas veces no se sigue el estándar.*

En este punto ya ha sido agregado el administrador de colisiones en el marco de trabajo, entonces crearemos un juego que no haga uso de él para que se vea la metodología para hacerlo. El nombre: *GameWithoutCollision*.

Código Fuente 89.

```
class GameWithoutCollision : public Game
{
...
 private:
...
   void create_collisionmanager();
};

void GameWithoutCollision::create_collisionmanager()
{
  // No se escribe nada para no ocupar el administrador de colisiones
}
```

Entonces si ahora creamos un juego a partir de *GameWithoutCollision* no comprobará las colisiones.

Interfaz del administrador de colisiones

La simple interfaz del controlador de colisiones que nos permitirá agregar esta característica al marco de trabajo es:

Código Fuente 90. archivo controlmanager.h

```
#ifndef COLLISIONMANAGER_H
#define COLLISIONMANAGER_H

#include "actormanager.h"
#include "game.h"
#include "mask.h"

class CollisionManager
{
public:
    CollisionManager(Game *g);

    typedef enum
    {
        BOUNDING_BOX,
        PP_COLLISION
    }collision_method_t;

    void update();

protected:
    Game *game;
};

#endif
```

Dado el análisis que se desprende de la tarjeta CRC se necesita dentro de la clase *CollisionManager* un referencia al objeto *Game* al que pertenece, de esta forma obtendremos los actores para hacerlos intersectar en parejas (a través de *Game->actor_manager*) . Por otro lado el método *CollisionManager::update* realizará las tareas necesarias de comprobación y la enumeración *collision_method_t* identificará el método de comprobación de colisión que se llevará a cabo.

Nuestro marco de trabajo soportará dos métodos: Bounding Box y Detección Perfecta por Píxeles, métodos que analizaremos más adelante.

Una decisión importante de diseño es darle la capacidad a los *actores* para seleccionar el método que se ocupará para detectarlos en vez de seleccionar un solo método de detección para todo el juego en curso ya que se podría haber optado por hacer una generalización del controlador de colisiones para luego heredarlos en *PerfectPixelCollisionManager* y *BoundingBoxCollisionManager*.

Como se realiza la detección de colisiones

Ahora veremos a grandes rasgos algunos aspectos técnicos de la detección de colisiones para los dos métodos que soporta nuestro marco de trabajo.

Método *Bounding Box*

Este es el método más rápido y menos preciso de detección abstrayendo los actores a simples rectángulos. Cuando se han obtenido las coordenadas del rectángulo que encierra a los actores se procede a interceptarlos, en caso de que los rectángulos estén solapados (uno encima de otro) la detección *Bounding Box* es verdadera. Veamos el siguiente diagrama para que se vea más claro esto:

Entonces con el siguiente código podríamos interceptar dos actores y ver si se solapan sus rectángulos:

Código Fuente 91.

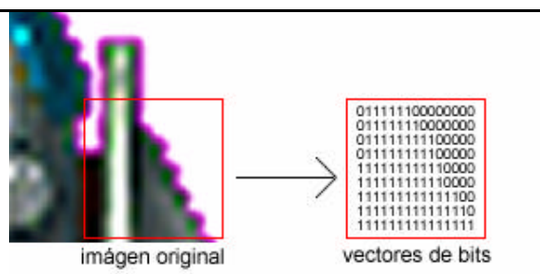
```
if (!((actor1->get_graph_x()>actor2->get_graph_x()+actor2->get_w())
    || (actor2->get_graph_x() > actor1->get_graph_x()+actor1->get_w())
    || (actor1->get_graph_y() > actor2->get_graph_y()+actor2->get_h())
    || (actor2->get_graph_y() > actor1->get_graph_y()+actor1->get_h())))
{
    // Código que se ejecute cuando estan colisionando
}
```

Para mejoras en la velocidad de comprobación de los límites del rectángulo se trabaja con solo “o” (comparación que se hace mediante “||”) y un negador al inicio de la comparación.

Método de Colisión Perfecta al Píxel

Este método es el mejor y el más lento, se comprueba cada píxel de la representación gráfica de un actor con la del otro.

Si refinamos un poco más nuestro algoritmo nos damos cuenta de que no necesitamos saber de qué color es el píxel, solo necesitamos conocer si “hay” o “no hay” un píxel”. Es por este motivo que abstraeremos la representación gráfica a un conjunto de *vectores de bit* que indicarán en donde se encuentra un píxel en la representación gráfica original. A continuación veremos como se hace esta transformación:



La representación gráfica de ejemplo es una imagen sencilla que es transformada a varios vectores de bits. Por ejemplo, el primer vector de bits (línea horizontal en la figura arriba de *vectores de bits*) es:

[0111110000000000]

Es decir, desde el bit 1 hasta el 6 existen píxeles en la imagen original. El conjunto de vectores de bits lo denominaremos *máscara de bits* de la representación gráfica.

Ahora supongamos que tenemos otra imagen que ha sido transformada a vectores de bits y que el primer renglón de su máscara de bits es:

[0000100000010000]

Si solo vemos este primer renglón de este vector y el anterior, ¿podríamos decir que las representaciones gráficas originales colisionaban? La respuesta es: sí.

Esto sucede porque en la posición 5 del vector de bits de ambas máscaras existe un 1, es decir, en la representación gráfica original había un píxel.

Ahora podemos hacer uso de algunos operandos que nos ofrece C/C++ para comparar bit a bit. Un operando del que haremos uso es “&” que retorna *verdadero* si encuentra en la misma posición de ambos vectores de bits un 1. Por ejemplo:

SI ((0001) & (0111)) ESCRIBIR(“Están superpuestos”)

En C/C++ no podemos trabajar tan fácilmente con bits, pero es sólo un ejemplo.

Por todo lo anteriores estamos en condiciones de dar paso a la creación de una nueva clase llamada *Mask* que nos proveerá de todas las funciones involucradas en la creación y detección de superposición de mascaras.

La clase Mask

Desde ahora en adelante cada representación gráfica de un actor tendrá asociada una máscara. Esta máscara contendrá el alto y ancho de la representación gráfica; y un arreglo bidimensional de bits (los vectores de bits) que indican, cada uno, la existencia o no de un píxel.

Este arreglo bidimensional de bits es en realidad un arreglo de enteros. Aprovechándonos de la facilidad para manejar los datos a nivel de bits en C/C++ es que utilizaremos el tipo de dato *unsigned long* para crear un vector de 32 bits.

De esta forma cada representación gráfica tendrá una máscara de bits representada por enteros de 32 bits que se ajustan en forma mínima a su tamaño. Por ejemplo, una representación gráfica de 40x80 píxel debe ser ajustada por un arreglo bidimensional de 2 *unsigned long* por 80 de alto.

Generalmente sobrarán bits horizontalmente al crear una máscara. Por ejemplo, para ajustarnos a los 40 píxeles de ancho anteriores necesitábamos dos *unsigned long* que podían cubrir un ancho de 64 píxeles (32 bits x 2).

Otra responsabilidad que asignaremos a la clase *Mask* es la entregarnos métodos para la detección por *bounding box* y *perfect pixel detection*.

Entonces, veamos la interfaz de la clase *Mask*:

Código Fuente 92. archivo mask.h

```
#include <allegro.h>

using namespace std;

#ifndef MASK
#define MASK

class Mask;

class Mask
{
public:
    Mask();
    ~Mask();

    static int check_ppcollision(Mask *m1, Mask *m2, int x1, int y1, int x2,
int y2);
    static int check_bbcollision(Mask *m1, Mask *m2, int x1, int y1, int x2,
int y2);
    void create(BITMAP *bmp);

    int get_bb_height() { return bb_height; }
    int get_bb_width()  { return bb_width;  }
    int get_max_chunk() { return max_chunk; }
    int get_num_y() { return num_y; }
    unsigned long int get_sp_mask(int i, int j) { return sp_mask[i][j]; }

protected:
    int bb_height;
    int bb_width;
    int max_chunk;
    int num_y;
    unsigned long int **sp_mask;

};

#endif
```

El primer método (*check_ppcollision*) realiza una detección perfecta por píxel. Recibe la mascara, la posición horizontal y vertical de un un par de actor de los cuales necesitamos saber si están colisionando. Retorna cero cuando no lo están haciendo y distinto de cero cuando sí.

El segundo método (*check_bbcollision*) hace lo mismo que *check_ppcollision* solo que mediante el método de colisión bounding box. El tercer método crea una máscara a partir de un mapa de bit. Los demás métodos retornan la información contenida en el objeto *Mask*. Los atributos *bb_height* y *bb_width* tienen la información sobre el alto y el ancho de la representación gráfica.

El último atributo (*sp_mask*) es el arreglo bidimensional que contiene la máscara del bits.

La parte de la implementación de esta clase es un tanto engorrosa y se presentará al final de esta sección de “detección de colisiones” dada la complejidad en la programación de bajo nivel en C++.

La implementación es:

Código Fuente 93. archivo mask.cpp

```
#include <allegro.h>
#include "game.h"
#include "mask.h"

Mask::Mask()
{
    sp_mask=NULL;
    bb_height=0;
    bb_width=0;
    max_chunk=0;
    num_y=0;
}

Mask::~Mask()
{
    if (sp_mask!=NULL)
    {
        for (int j=0; j<num_y; j++)
            delete[] sp_mask[j];
        delete[] sp_mask;
    }
}

void Mask::create(BITMAP *bmp)
{
    // Creando la memoria para la mascara
    sp_mask = new unsigned long int *[bmp->h];
    for (int j=0; j<bmp->h; j++)
        sp_mask[j]=new unsigned long int[((bmp->w)>>5)+1];

    // creación de la mascara de bits

    // Desde aquí hacia abajo este método funciona como "caja negra" para mi
    int x1, y1, z; /* Used to span through pixels within the sprite */
    int p; /* Holds return value from getpixel() */

    for (y1=0; y1<(bmp->h); y1++) /* Now go through each pixel of the sprite
*/
    {
        for(z=0; z<(int)((bmp->w)/32)+1; z++){
            sp_mask[y1][z]=0;
            for (x1=(z*32); x1<((z*32)+32); x1++)
            {
                p=getpixel(bmp,x1,y1);
                if ((p!=bitmap_mask_color(bmp)) && (p>=0))
                {
                    if (z>max_chunk) (max_chunk = z);
                    if (y1>bb_height) bb_height=y1;
                    if (x1>bb_width) bb_width=x1;
                    sp_mask[y1][z]+=0x80000000 >> (x1 - ((z+1)*32));
                }
            }
        }
    }
    num_y=bmp->h;
}

int Mask::check_bbcollision(Mask *m1, Mask *m2, int x1, int y1, int x2, int
y2)
{

    if ((x1>x2+m2->get_bb_width()) || (x2 > x1+m1->get_bb_width()) ||
        (y1>y2+m2->get_bb_height()) || (y2> y1+m1->get_bb_height()))
    {
        return 0;
    }
    else
    {
        return 1;
    }
}

int Mask::check_ppcollision(Mask *m1, Mask *m2, int x1, int y1, int x2, int
y2)
{
```

```
      int dx1, dx2, dy1, dy2, ddy1, ddy2;
      int spr1_chunk, spr2_chunk;
      int dx1_chunk, dx2_chunk;

      if ((x1>x2+m2->get_bb_width()) || (x2 > x1+m1->get_bb_width()) ||
       (y1>y2+m2->get_bb_height()) || (y2> y1+m1->get_bb_height()))
      {
           return 0;    /* No collision */
      }
      else   /* if bounding box reports collision do pixel-perfect check*/
      {
     if (x1>x2) {
       dx1=0;             //don't need to shift sprite 1
       dx2=x1-x2;  //we need to shift sprite 2 left
     }
     else {
       dx1=x2-x1;  //we need to shift sprite 1 left
       dx2=0;             //don't need to shift sprite 2
     }
     if (y1>y2) {
       dy1=0;             //we don't need to skip any rows on sprite 1
       dy2=y1-y2;  //we need to skip this many rows on sprite 2
     }
     else {
       dy1=(y2-y1);  //we need to skip this many rows on sprite 1
       dy2=0;           // we don't need to skip any rows on sprite 2
     }


     spr1_chunk = dx1 / 32;
     spr2_chunk = dx2 / 32;
     dx1_chunk = dx1 - (32 * spr1_chunk);
     dx2_chunk = dx2 - (32 * spr2_chunk);

    while((spr1_chunk <= m1->get_max_chunk()) & (spr2_chunk <= m2-
>get_max_chunk())){
      ddy1 = dy1;
      ddy2 = dy2;

      while((ddy1<=m1->get_bb_height())&&(ddy2<=m2->get_bb_height())) {
       if ((m1->get_sp_mask(ddy1,spr1_chunk)<<dx1_chunk)&(m2-
>get_sp_mask(ddy2,spr2_chunk)<<dx2_chunk))
        return 1;
       ddy1++;
       ddy2++;
      }

      if((!dx1_chunk) && (!dx2_chunk)){  /*In case both sprites are lined up on
x axis */
       spr1_chunk++;
       spr2_chunk++;
      }
      else{
       if(!dx1_chunk){  /*Set up the next 32 bit chunk in the mask for
detection*/
        spr2_chunk++;
        dx1_chunk = 32 - dx2_chunk;
        dx2_chunk = 0;
       }
       else if(!dx2_chunk){
        spr1_chunk++;
        dx2_chunk = 32 - dx1_chunk;
        dx1_chunk = 0;
       }
      }
     }

     return 0; /* no collision */
   }
  }
```

Nota: *La implementación de la clase Mask y más específicamente los métodos Mask::create, Mask::check_bbcollision y Mask::check_ppcollision funcionan como caja negra para mi y no puedo ofrecer un soporte acusioso de estos métodos. Es mi obligación mencionar el lugar de donde he obtenido estos algorítmos: http://www.geocities.com/SiliconValley/Park/1077/rout.htm*

Modificación a ActorGraphic

Entonces ahora veremos las modificaciones que debemos hacer a la clase *ActorGraphic* para que nos permita obtener la máscara de representación gráfica:

Código Fuente 94. modificación de la clase ActorGraphic

```
    class ActorGraphic
    {
        public:
    ...
            virtual Mask* get_mask();
    ...
    };
```

Se agrega el método *get_mask* que será ocupado por el administrador de colisiones para rescatar la mascara de todos los actores en pantalla. En realidad la máscara se obtendrá de todos los objetos de la clase actor que están en el administrador de actores mediante un método llamado *get_mask* y que solo será un puente al mismo método de su representación gráfica.

Perfeccionando el criterio de detección

En todo juego se necesita diferenciar a los actores. Por ejemplo, un proyectil que lance un aliado no debería colisionar conmigo, que soy también aliado. De esta necesidad nace el soporte de *equipos* en nuestro marco de trabajo.

Cada actor pertenece a un equipo. La lista de equipos disponibles se puedo obtener desde la clase *Game* con el atributo que llamaremos *team_t*. El controlador de colisiones solo comprobará una superposición cuando dos actores sean de distintos equipos.

Se establecerán por defecto dos equipos: los aliados y los enemigos. Veamos entonces el nuevo atributo para la clase *Game*.

Código Fuente 95. modificación de la clase Game

```
    class Game
    {
        public:
    ...
            typedef enum
            {
                ALLY,
                ENEMY
            }team_t;
    ...
```

Supongamos que un actor no quiere ser detectado por ningún otro elemento, es decir, ser indetectable. Se agrega otro criterio de detección representado por el atributo *is_detectable*. Cuando este atributo está en *verdadero* se procede a comprobar la detección. Ambos actores deben tener el atributo *is_detectable* en verdadero para ser comprobada la superposición y además ser de distinto equipo.

Modificando la clase Actor y ActorManager

Actor

Una nueva responsabilidad de la clase actor es proporcionarnos la máscara de su representación gráfica actual mediante un método llamado *get_graph_mask*. Además nos permitirá modificar los atributos que permiten comprobar la detección (*is_detectable*) y establecer el equipo al que pertenece el actor (*team*).

Además la clase *Actor* podrá establecer el método de colisión que se empleará para la detección (*set_collision_*method y *get_collision_method*) y el daño que produce cuando colisiona con otro actor *(set_power* y *get_power*.)

Un actor será notificado sobre una colisión a través del método *hit*. Este método tendrá dos parámetros, el primero indicando *quién* lo colisiona y *cuanto daño* le produce.

Código Fuente 96. modificaciones a la interfáz de la clase Actor

```
...
#include "collisionmanager.h"

class Actor
{
    public:
...

        Mask *get_graph_mask();
        void set_is_detected(bool tf);
        bool get_is_detected();
        void set_power(int pow);
        int get_power();
        void set_collision_method(CollisionManager::collision_method_t cm);
        CollisionManager::collision_method_t get_collision_method();
        virtual void hit(Actor *who, int damage);

        void set_team(Game::team_t tm);
        Game::team_t get_team();

    protected:
...
        int power;
        Game::team_t team;
        bool is_detectable;
        CollisionManager::collision_method_t collision_method;
...
};

#endif ACTOR_H
```

Ahora para crear un actor se deben seguir los siguientes pasos:

1. Crear el objeto a partir de alguna clase de actor enviándole de parámetro el juego al que pertenece;

```
AirCraft *airc=new AirCraft(this);
```

2. Establecer la representación gráfica del actor y la posición en pantalla;

```
bmp1=load_bitmap("nave1.pcx", NULL);
bmp2=load_bitmap("nave2.pcx", NULL);
Sprite *sp=new Sprite(airc);
sp->add_frame(bmp1, bmp1->w/2,     bmp1->h/2,     20);
sp->add_frame(bmp2, bmp2->w/2,     bmp2->h/2,     20);
airc->set_x(SCREEN_W/2);
airc->set_y(SCREEN_H/2);
airc->set_actor_graphic(sp);
```

3. Establecer si es detectable o no, a qué equipo pertenece y que tipo de detección emplearemos con él;

```
airc->set_is_detected(true);
airc->set_team(ENEMY);
airc->set_collision_method(CollisionManager::PP_COLLISION);
```

4. Finalmente agregarlo al administrador de actores.

```
actor_manager->add(airc);
```

Lo único agregado desde las implementaciones anteriores es el punto 3.

ActorManager

El controlador de actores nos proveía una forma lineal de recorrer todos los actores del juego. Ahora se necesita que el controlador de colisiones visite de par en par todos los actores. Debido a esto agregaremos algunos métodos que permitirán tener control total sobre la forma en cómo visitamos a los actores:

Código Fuente 97. modificación sobre la clase ActorManager

```
    class ActorManager
    {
        public:
...
            list<Actor*>::iterator get_begin_iterator();
            list<Actor*>::iterator get_end_iterator();
...
    };

    #endif
```

El método *get_begin_iterator* retornará el *iterador* (concepto de las clases estándares de C++ STL) inicial de la lista de actor, de la misma forma *get_end_iterator* retorna el iterador de la posición final.

Otro cambio que se ha realizado en la implementación de *ActorManager* es la capacidad que tiene un actor de eliminarse a sí mismo. Por ejemplo, supongamos que un actor se ha salido de los límites de la pantalla y debe eliminarse por esta causa:

Código Fuente 98. ejemplo de actor que se elimina al salir de la pantalla

```
    void ActorPrueba::move()
    {
     if (x<0) game->actor_manager->del(this);
    }
```

En este caso *ActorPrueba* puede decirle al administrador de actores que lo elimine. Sin embargo, el administrador de actores no puede eliminar al actor inmediatamente sino que debe dejar esperándolo en una cola de actores “eliminables”. Al final de cada llamada al método *ActorManager::update* el administrador de actores elimina a todos los que esperaban en esa cola.

De la misma forma puede hacerse para actores que quieren crearse en medio del juego, para ellos también se implementa una cola de espera de actores “insertables” .

La interfaz para ambos requerimientos:

Código Fuente 99. modificación de la clase ActorManager

```
    class ActorManager
    {
...
        protected:
...
            list<Actor*> to_del;
            list<Actor*> to_create;

            void add_all_to_create();
            void del_all_to_del();
    };
```

La implementación para los métodos nuevos y modificados de *ActorManager*:

Código Fuente 100. implementación de nuevos métodos de ActorManager

```
#include "actor.h"
#include "actormanager.h"

...
void ActorManager::add(Actor *a)
{
    to_create.push_back(a);  // se agrega a la cola de espera
}

void ActorManager::add_all_to_create()
{
    if (to_create.size()==0) return;
    list<Actor *>::iterator tmp_iter;
    for (tmp_iter=to_create.begin(); tmp_iter!=to_create.end(); tmp_iter++)
    {
        actors.push_back(*tmp_iter);
        (*tmp_iter)->init();
    }
    to_create.clear();
}

void ActorManager::del(Actor *a)
{
    to_del.push_back(a); // ahora se agrega a la cola de espera
}


void ActorManager::del_all_to_del()
{
    if (to_del.size()==0) return;
    list<Actor*>::iterator tmp_iter;
    list<Actor*>::iterator tmp_actors_iter;

    for (tmp_iter=to_del.begin(); tmp_iter!=to_del.end(); tmp_iter++)
    {
        tmp_actors_iter=find(actors.begin(), actors.end(), *tmp_iter);
        if (tmp_actors_iter!=actors.end())
        {
            actors.erase(tmp_actors_iter);
            delete (*tmp_iter);
          }
    }
    to_del.clear();
}

...

void ActorManager::update()
{
    list<Actor*>::iterator tmp_iter;

    add_all_to_create(); // llamada para agregar todos los actores en espera
    for (tmp_iter=actors.begin(); tmp_iter!=actors.end(); tmp_iter++)
        (*tmp_iter)->update();
    del_all_to_del(); // lamada para eliminar todos los actores en espera
}

list<Actor*>::iterator ActorManager::get_begin_iterator()
{
    return actors.begin();
}

list<Actor*>::iterator ActorManager::get_end_iterator()
{
    return actors.end();
}
```

Haciendo detectables las representaciones gráficas

Ahora debemos hacer que las representaciones gráficas creen sus propias máscaras. Como un ejemplo, veamos como crea su máscara la representación gráfica *Pixel*.

Primero se agrega cómo atributo a la clase *Pixel* una máscara *mask* además de sobreescribir el método *get_mask*:

Código Fuente 101. archivo pixel.h

```
#ifndef PIXELGRAPH_H
#define PIXELGRAPH_H

#include "game.h"
#include "actorgraphic.h"
#include "mask.h"

class Pixel : public ActorGraphic
{
    public:
        Pixel(Actor *aowner, int col);
        ~Pixel();

        void draw(BITMAP *bmp);
        int get_w();
        int get_h();
        Mask *get_mask();

    protected:
        int color;
        Mask *mask;
};

#endif
```

La creación de la máscara se hace en el constructor de esta clase:

Código Fuente 102. archivo pixel.cpp

```
#include "pixel.h"

Pixel::Pixel(Actor *aowner, int col) : ActorGraphic(aowner)
{
    color=col;
    // Se creara la mascara de un pixel a partir de un bitmap vacio
    BITMAP *tmp_bmp=create_bitmap(1,1);
    clear_to_color(tmp_bmp, 1); // cualquier color
    mask=new Mask;
    mask->create(tmp_bmp);
    destroy_bitmap(tmp_bmp);
}
```

Dado que una máscara se crea a partir de un mapa de bits, se crea un mapa de bits temporal de 1x1 píxels relleno con cualquier color, menos el de la máscara. Después de eso se crea una instancia de *Mask* y se llama a su método *Mask::create* para terminar con la creación de la máscara. Finalmente se destruye el mapa de bits temporal.

La implementación del método *Pixel::get_mask* es trivial:

Código Fuente 103. implementacion de get_mask

```
Mask *Pixel::get_mask()
{
    return mask;
}
```

En el caso que la representación gráfica tenga varios cuadros, como por ejemplo la clase *Sprite*, se debe crear una máscara para cada uno de ellos.

Cuando implementamos la clase *Sprite* hace algún tiempo atras definimos un tipo de dato llamado *Frame* que representaba a un cuadro. Lo que haremos simplemente es agregar un nuevo campo al tipo de dato *Frame* guardando la máscara de ese cuadro actual:

Código Fuente 104. archivo sprite.h

```
#ifndef SPRITE_H
#define SPRITE_H

#include "actorgraphic.h"
#include "actor.h"

class Sprite : public ActorGraphic
{
    public:
...
        Mask *get_mask();

    protected:
        typedef struct Frame
        {
            BITMAP *bmp;
            Mask *mask;
            int cx;
            int cy;
            int ticks;
        };
...
};
```

Cada vez que creemos una representación gráfica debemos sobreescribir el método *get_mask* para adaptarlo al comportamiento de nuestra clase. Por ejemplo, en la clase *Sprite* el método *get_mask* retornará la máscara del cuadro actual que se está desplegando en pantalla:

Código Fuente 105. metodo get_mask

```
Mask *Sprite::get_mask()
{
    return frames[actual_frame].mask;
}
```

La creacion de las mascaras se hace justo en el momento en que se agregan los cuadros a un objeto *Sprite*, es decir, en el método *Sprite::add_frame*:

Código Fuente 106. nuevo metodo add_frame

```
void Sprite::add_frame(BITMAP *bmp, int cx, int cy, int ticks)
{
    Frame tmp_frame;
    tmp_frame.bmp=bmp;
    tmp_frame.cx=cx;
    tmp_frame.cy=cy;
    tmp_frame.ticks=ticks;
    // Creacion de la mascara
    tmp_frame.mask=new Mask;
    tmp_frame.mask->create(bmp);
    ////////////////////////
    frames.push_back(tmp_frame);
}
```

Ahora no es necesario crear un mapa de bits temporal para la el objeto *Mask* ya que la clase *Sprite* trabaja con este tipo de dato.

Un pequeño juego con detección de colisiones

Ahora crearemos, a partir del juego que hemos venido desarrollando, una aventura en la cual debes destruir las estrellas que vienen hacia ti... no es una gran historia, pero les sirve para probar la detección de colisiones en terreno.

En nuestro juego crearemos tres clases actores:

- Aircraft
- Explosion
- Star

Un una clase representación gráfica:

- CircleWithZoom

Si queremos trabajar con la detección en alguna clase debemos sobreescribir el método *hit*, por ejemplo, en la conocida clase *AirCraft*:

Código Fuente 107. interfaz de aircraft

```
class AirCraft : public ControllableActor
{
    public:

        AirCraft(Game *g);

        typedef enum
        {
            DOWN,
            UP,
            LEFT,
            RIGHT
        }action_t;

        void do_action(ControllableObject::action_t act, int magnitude);
        void hit(Actor *a, int damage);
};
```

La clase *Explosion* es un actor que se creará cuando un estrella *(Star)* colisione con un enemigo, en este caso un objeto de *AirCraft*.

De esta forma, cuando una estrella es colisionad, se elimina del administrador de actores:

Código Fuente 108.

```
void Star::hit(Actor *a, int damage)
{
 game->actor_manager->del(this);
}
```

Y cuando un *AirCraft* es colisionado se crea un actor explosión:

Código Fuente 109.

```
void AirCraft::hit(Actor *a, int damage)
{
    Explosion *exp=new Explosion(game, 10);
    CircleWithZoom *cwz=new CircleWithZoom(exp, 10);
    exp->set_x(a->get_x()-10+rand()%20);
    exp->set_y(a->get_y()-10+rand()%20);
    exp->set_actor_graphic(cwz);
    exp->set_is_detected(false);

    game->actor_manager->add(exp);
}
```

Ahora veamos el cuerpo principal del método *TestFrameWork::main* que modificará los asignará los equipos a cada objeto que realice, les dirá si son detectable y, eventualmente, les podría decir cual es método de colisión que se empleará con ellos:

Código Fuente 110. clase TestFrameWork

```
class TestFrameWork : public Game
{
    public:
        void main();
};

void TestFrameWork::main()
{
    BITMAP *bmp1, *bmp2;
    AirCraft *airc=new AirCraft(this);

    Control *control_p1=new Control;

    srand(time(0));

    control_p1->add_action_name(AirCraft::DOWN, "Bajar");
    control_p1->add_action_name(AirCraft::UP, "Subir");
    control_p1->add_action_name(AirCraft::LEFT, "Izquierda");
    control_p1->add_action_name(AirCraft::RIGHT, "Derecha");

    Keyboard *kboard=new Keyboard;


    control_p1->set_actionperipheral(AirCraft::DOWN, kboard, KEY_DOWN,
Peripheral::ON_PRESSING);
    control_p1->set_actionperipheral(AirCraft::UP, kboard, KEY_UP,
Peripheral::ON_PRESSING);
    control_p1->set_actionperipheral(AirCraft::LEFT, kboard, KEY_LEFT,
Peripheral::ON_PRESSING);
    control_p1->set_actionperipheral(AirCraft::RIGHT, kboard ,KEY_RIGHT,
Peripheral::ON_PRESSING);

    control_p1->set_owner(airc);
    control_manager->add_control(control_p1);
    control_manager->add_peripheral(kboard);

    Star *star_tmp;
    Pixel *pixel_tmp;
    for (int i=0; i<10;i++)
    {
        star_tmp=new Star(this);
        pixel_tmp=new Pixel(star_tmp, makecol(255, 255, 255));
        star_tmp->set_actor_graphic(pixel_tmp);
        star_tmp->set_y(rand()%SCREEN_H);
        star_tmp->set_is_detected(true);
        star_tmp->set_team(ALLY);
        actor_manager->add(star_tmp);
    }

    bmp1=load_bitmap("nave1.pcx", NULL);
    bmp2=load_bitmap("nave2.pcx", NULL);
    Sprite *sp=new Sprite(airc);
    sp->add_frame(bmp1, bmp1->w/2,     bmp1->h/2,     20);
    sp->add_frame(bmp2, bmp2->w/2,     bmp2->h/2,     20);
    airc->set_x(SCREEN_W/2);
    airc->set_y(SCREEN_H/2);
    airc->set_actor_graphic(sp);
    airc->set_is_detected(true);
    airc->set_team(ENEMY);
    airc->set_collision_method(CollisionManager::PP_COLLISION);
    actor_manager->add(airc);

    while (!key[KEY_ESC]) update();
    destroy_bitmap(bmp1);
    destroy_bitmap(bmp2);
}
```

Resumen

En estos artículos de detección de colisiones nos hemos paseado por una de las características más importantes en un juego que nos permitirá tener un control de los acontecimientos del campo de batalla.

Por el final de esta serie de artículos pudimos observar la facilidad para crear un juego ya que la detección de colisiones funciona en forma transparente para nosotros.