Lógica del videojuego

Jordi Duch i Gavaldà
Heliodoro Tejedor Navarro

PID_00188520

UOC
Universitat Oberta de Catalunya

Índice

Introducción

Lo que diferencia un videojuego de cualquier otro tipo de aplicación no son sus gráficos ni su sonido, sino el conjunto de retos y reglas que estimulan la interacción con el usuario. Con los elementos explicados hasta ahora aún no podemos implementar ningún juego, sino que se trata solamente de complementos que principalmente van a ayudar a que el usuario se sienta más integrado dentro del juego y que pueda tener acceso a la mayor cantidad de información posible para poder desarrollar la partida.

La parte donde se controla el desarrollo del juego es lo que llamamos el motor de lógica. Esta parte, tan o más importante que todas las descritas anteriormente, incluye la descripción de los atributos de todos los elementos que participan, y de todas las reglas y condiciones que hemos situado en el juego. Continuamente mira las acciones que han realizado los jugadores y los elementos controlados por la inteligencia artificial y decide si estas acciones se pueden llevar a cabo y cuál es el resultado de ejecutarlas.

Dentro del motor lógico de juego se juntan tres elementos muy importantes:

- La integración de todos los componentes que hemos visto hasta ahora
- La gestión de todos los datos de la partida
- La aplicación de las reglas del juego

Veremos con detalle cómo se implementan estos tres elementos, que serán el corazón de nuestro juego.

Además, la parte del motor lógico es la que se encuentra más ligada al proceso creativo del juego y es donde intervienen algunos perfiles no tan familiarizados con la programación avanzada. Para separar la parte creativa de la programación se utilizan los lenguajes de *script*, que veremos en la última parte del módulo.

Objetivos

Con el material de este curso podréis alcanzar los objetivos siguientes:

1. Repasar los conceptos de ingeniería del software necesarios para el desarrollo de la estructura del videojuego.

2. Implementar un diagrama de estados para gestionar las transiciones entre los diferentes estados de un juego.

3. Diseñar el bucle principal de un juego.

4. Organizar la información en la vista lógica para que el motor lógico y la inteligencia artificial conozcan el desarrollo de la partida.

5. Implementar el motor lógico para garantizar que las reglas del juego se cumplen.

6. Almacenar los datos necesarios para el juego de una forma óptima y estructurada.

7. Utilizar un lenguaje de *scripting* para facilitar el trabajo del diseñador del juego.

1. Ingeniería del software aplicada a los videojuegos

En el primer apartado de este módulo vamos a hacer un repaso de la teoría básica de ingeniera del software y cómo se introducen los conceptos clásicos de esta teoría en el desarrollo de un videojuego.

Es muy importante conocer la mayoría de estos conceptos, sobre todo para poder desarrollar aplicaciones a gran escala de una forma modular y estructurada.

Aprender estas técnicas de ingeniería de software adicionalmente nos va a permitir:

- Resolver problemas comunes con un procedimiento óptimo y ya probado.
- Ser más productivos porque no tenemos que volver a pensar cómo hacer un algoritmo y porque sabemos que el que implementamos ya ha sido probado para resolver otros problemas con buenos resultados.
- Complementar la documentación con el propio código (aunque nunca la sustituye).
- Entender código ajeno con sólo ver su estructura.

En la ingeniería del software se han ido desarrollando nuevas formas de organización a la hora de crear nuevas aplicaciones. Una de estas formas, que es muy interesante conocer a la hora de desarrollar aplicaciones, son los patrones de diseño.

Para entender bien el concepto de los patrones de diseño, veremos primero la evolución que ha sufrido el arte de la programación y los diferentes paradigmas que han ido apareciendo.

A partir del repaso a estos paradigmas, conoceremos el concepto de patrón de diseño y estudiaremos los más importantes y que nos serán útiles a la hora de desarrollar un videojuego.

1.1. Paradigmas de programación

Los paradigmas de programación más destacados y que vamos a repasar a continuación son los siguientes:

- Programación no estructurada
- Programación estructurada

- Programación modular
- Programación usando tipos abstractos de datos
- Programación orientada a objetos

1.1.1. Programación no estructurada

Es la primera y más básica forma de programación. En este tipo sólo pretendemos hacer que un algoritmo funcione sin tener en cuenta ningún aspecto extra: estructuración, seguridad, mantenimiento, etc.

Utilizamos variables globales, no prestamos atención a los identificadores de las variables.

El control del flujo de las instrucciones la realizamos básicamente con la instrucción "goto" y el bucle "if".

ejemplo.c

```
#include <stdio.h>
float a, aa, aaa;
int main()
{
inicio:
   a = 3.1415926f;
   scanf("%f\n", &&aa);
   if (aa == 0.0f) goto fin;
   aaa = 2 * a * aa;
   printf("El perimetro de una circumferencia de radio %f es %f\n", aa, aaa);
   goto inicio;
fin:
   return 0;
}
```

1.1.2. Programación estructurada

Una vez hemos aprendido a desarrollar pequeños algoritmos, empezamos a utilizar estructuras que nos permiten mantener mejor nuestro código. Además, nos damos cuenta de que este código es más legible para los demás desarrolladores.

Evitamos las variables globales, intentamos dividir los algoritmos en pequeños procedimientos, utilizamos nombres de variables más correctos.

Se utilizan bucles "for" y "while" y se evita la instrucción "goto".

ejemplo.c

```
#include <stdio.h>
float leer_radio()
{
   float radio;
   scanf("%f", &radio);
   return radio;
}
float calcular_perimetro(float radio)
{
   float PI = 3.1415926f;
   return 2.0 * PI * radio;
}
int main()
{
   float radio;
   float perimetro;
   do
   {
      radio = leer_radio();
      if (radio != 0.0f)
      {
         perimetro = calcular_perimetro(radio);
         printf("El perimetro de una circunferencia de radio %f es %f\n", radio, perimetro);
      }
   } while (radio != 0.0f);
   return 0;
}
```

1.1.3. Programación modular

En este nuevo paradigma, los programadores necesitamos cada vez más desarrollar un código más complejo y vemos que es necesario volver a escribir determinados algoritmos. Empezamos a utilizar módulos (o bibliotecas) donde podemos ir almacenando código que podemos utilizar en otros programas.

Por ejemplo, tenemos un módulo llamado "circunferencia" con las siguientes funciones:

Circunferencia.h

```
float leer_radio();
```

```
float calcular_perimetro(float radio);
void imprimir_info(float radio, float perimetro);
```

Y un programa que lo utiliza:

Ejemplo.c

```
#include <circunferencia.h>
int main()
{
   float radio;
   float perimetro;
   do
   {
      radio = leer_radio();
      if (radio != 0.0f)
      {
         perimetro = calcular_perimetro(radio);
         imprimir_info(radio, perimetro);
      }
   } while (radio != 0.0f);
   return 0;
}
```

1.1.4. Programación usando tipos abstractos de datos

Una vez conocidas las ventajas de la programación modular, nos damos cuenta de una fuente de errores bastante común. Los datos se suelen agrupar formando una entidad (por ejemplo, un cliente tiene un DNI, un nombre, un teléfono...). Estos datos debemos tratarlos en conjunto y evitar en lo posible que funciones externas al módulo puedan modificar esta información. Con esta nueva forma de trabajar, evitaremos que los datos de una entidad dejen de ser consistentes (una función externa puede cambiar solamente el nombre de un cliente, pero seguiría teniendo los mismos apellidos; y puede que sea un comportamiento no deseado).

Podemos solucionar este problema trabajando con los datos de una entidad dentro de su módulo y que el cliente de éste sólo pueda acceder a ellos utilizando las funciones que el módulo le proporciona.

Por ejemplo, desarrollamos el siguiente módulo para trabajar con un avatar de un videojuego:

Avatar.h

```
typedef void* avatar_t;
avatar_t avatar_create(string name, point_t location, state_t state);
```

```
void avatar_delete(avatar_t avatar);
string avatar_get_name(avatar_t avatar);
point_t avatar_get_location(avatar_t avatar);
void avatar_move(avatar_t avatar, vector_t distance);
...
```

Y un programa que utiliza este módulo:

Game.c

```
#include <avatar.h>
#include <game.h>
#include <list.h>
void update_state(game_t game)
{
   list_t list = game_get_avatars(game);
   while (list_has_next(list))
   {
      avatar_t cur_avatar = (avatar_t) list_get_next(list);
      vector_t distance = game_calculate_movement(game, cur_avatar);
      avatar_move(cur_avatar, distance);
      list_move_next(&list);
   }
}
```

1.1.5. Programación orientada a objetos

A partir de la programación usando tipos abstractos de datos, nos damos cuenta de que estos módulos los podemos organizar usando jerarquías (un avatar y un coche son dos objetos animados). Es en este paso cuando nos adentramos en la programación orientada a objetos (POO), que nos permite añadir muchas más funcionalidades a la programación con tipos abstractos de datos.

Este paradigma nos proporciona muchos mecanismos para programar código más seguro, funcional y reutilizable. Sus principales ventajas son:

- Los paquetes de objetos bien construidos los podemos utilizar en otros proyectos aumentando su productividad. Fomenta la reutilización de código.

- Si los objetos se reutilizan, aumentamos la seguridad, puesto que estos objetos ya han sido probados en otras aplicaciones.

- Los mecanismos de la POO nos permiten crear jerarquías de objetos que se pueden ampliar con menor coste que los sistemas tradiciones de programación.

- La correcta utilización de la POO nos permite reaprovechar las mismas líneas de código de un algoritmo para diferentes tipos de datos (objetos). A menor líneas de código, menor probabilidad de errores y tenemos un mantenimiento menor.

- La POO permite resolver problemas trabajando con los mismos términos del problema (dominio); no se utilizan términos del ordenador (podemos hablar de avatares, enemigos, coches, pelotas...).

- Al igual que en la utilización de tipos abstractos de datos, la POO nos permite modificar un determinado algoritmo sin necesidad de reprogramar el resto de la aplicación, siempre y cuando la interfaz (contrato entre el responsable de un objeto y sus usuarios) se mantenga.

- La POO nos facilita la programación en un grupo de trabajo. Una vez se concluye la fase de diseño, cada miembro del grupo puede ir programando los diferentes objetos que componen el problema con apenas interferencias de los demás miembros. También nos facilita la creación de tests unitarios para cada uno de los objetos (útiles tanto en la fase de diseño como en la de depuración).

- Un código creado con POO es más legible y fácil de entender para un programador medio comparándolo con lenguajes más tradicionales (básicamente, C). No es necesario contratar a programadores expertos para la mayoría de problemas y se pueden rebajar costes (en la mayoría de casos, sólo es necesario contratar programadores expertos en C++ para programar el núcleo de un videojuego).

- La POO genera un mercado de compra/venta de componentes y una base para que en un futuro haya un procedimiento para interactuar entre varios objetos de diferentes programadores (y entre diferentes lenguajes). Varios intelectuales no consideran la programación como una ingeniería puesto que no hay ningún procedimiento estándar que permita reutilizar código (compárese con las demás ciencias técnicas).

Como todos los paradigmas, la POO también tiene sus desventajas:

- La primera y más importante es que es muy difícil de traducir a un lenguaje orientado a objetos el código que ya existe. En el caso concreto de los videojuegos, este punto no es muy importante, ya que la mayoría de desarrollos se empiezan casi desde cero.

- Se debe formar a los grupos de desarrollo con las nuevas técnicas de programación, lo que genera costes.

- Los desarrolladores ya formados en programación tradicional suelen ser reacios a aprender nuevas formas de programación. Se debe cambiar la forma de pensar en el problema y eso no suele gustarle a mucha gente.

- Aunque en la mayoría de aplicaciones no suele ser crítico, la ejecución de un código realizado con POO suele ser más lenta que usando técnicas de programación tradicional.

1.1.6. Características de la POO

La POO no sólo amplía los conceptos de la programación tradicional, modifica la forma de pensar para atacar los problemas.

Los puntos básicos a tener en cuenta cuando se trata de aprender el paradigma de la POO son los siguientes:

Modelo de objeto

Definiremos una clase como la definición de una estructura que contiene información (datos) y una declaración de algoritmos (o métodos) que modificarán la información.

Un objeto es una zona de memoria que cumple con la estructura de una clase concreta y se puede deducir que un objeto sólo existe mientras esté en ejecución la aplicación.

Podemos comparar una clase como un tipo de datos de la programación tradicional y un objeto como una variable del tipo definido por la clase.

Un objeto puede seguir viéndose como un tipo abstracto de datos.

Abstracción

Podemos definir la abstracción como la representación de las características esenciales de cualquier cosa sin incluir detalles irrelevantes. Todos tenemos un modelo de un coche, pero pocos (mecánicos) saben exactamente cómo funciona. Nosotros hemos abstraído un modelo simplificado que representa las características que necesitamos conocer de un coche para hacerlo funcionar.

De la misma manera, podemos crear una clase (un objeto en movimiento, por ejemplo) que represente un detalle concreto del problema y que los usuarios de nuestra clase sólo necesiten saber las funcionalidades que necesitan (hacer que se mueva en una dirección, consultar su posición actual...).

Encapsulamiento

Al igual que con los tipos abstractos de datos (una clase no deja de serlo), necesitamos evitar que el usuario de nuestra clase pueda acceder a la información interna, hacer mal uso de ella y crear inconsistencia en nuestros datos.

Debemos actuar como si cada clase fuese una caja negra, donde no se sabe qué se guarda ni cómo se trabaja con la información; sólo necesitamos saber que hace la función, cuánto tarda en hacer su trabajo y que lo hace bien (en el caso de que no fuese así, el responsable de la clase debería solucionarlo).

Herencia

La herencia es el mecanismo que consiste en extender los comportamientos de una clase y poder compartir los métodos y atributos (datos) con sus subclases (clases que heredan de ella). En el ejemplo de los objetos en movimiento, podemos crear una herencia sencilla que explicará este apartado:

- Un objeto en movimiento se define como una posición, una velocidad y una aceleración.
- Un objeto en movimiento tiene como un método "update" al que se le pasa el número de segundos que han transcurrido y debe modificar su posición.

MovingObject.h

```
class MovingObject
{
protected:
   Point position;
   Vector3D velocity;
   Vector3D acceleration;
public:
   void update(float seconds)
   {
      position += velocity * seconds;
      velocity += acceleration * seconds;
   }
   Point& getPosition() { return position; }
   Type getType() { return MovingObjectType; }
   ...
};
```

A partir de aquí, un avatar es un objeto en movimiento:

Avatar.h

```
#include <MovingObject.h>
```

```
class Avatar
   : public MovingObject
{
private:
   Vector3D desiredVelocity;
public:
   void update(float seconds)
   {
      velocity = velocity * 0.5f + desiredVelocity * 0.5f;
      position += velocity * seconds;
   }
   Type getType() { return AvatarType; }
};
```

Con el ejemplo anterior, podemos crear la clase siguiente:

Game.h

```
#include <list>
class Game
{
private:
   std::list<MovingObject*> movingObjects;
public:
   Game()
   {
      movingObjects.push_back(new Avatar("Malo"));
      movingObjects.push_back(new Avatar("Bueno"));
      movingObjects.push_back(new MovingObject("Piedra de Indiana Jones"));
      movingObjects.push_back(new MovingObject("Balancín"));
   }
   void update(float seconds)
   {
      std::list<MovingObject*>::iterator it;
      for (it = movingObjects.begin(); it != movingObjects.end(); it++)
              (*it)->update(seconds);
      prepareDraw();
      for (it = movingObjects.begin(); it != movingObjects.end(); it++)
              draw((*it)->getPosition(), (*it)->getType());
      drawOverlays();
      finalizeDraw();
   }
};
```

Polimorfismo

Permite que varios objetos de una misma clase puedan tener comportamientos diferentes. Éste suele ser un concepto más difícil de explicar que de entender con un ejemplo. La idea básica la podemos explicar con el ejemplo anterior, donde el método "update" tiene dos comportamientos diferentes según sea un movingObject o un avatar. En la clase Game se llama indistintamente a cada uno de los dos comportamientos según el objeto que hay en la lista (movingObjects) sea de una clase o de otra (se define el tipo al crear la lista).

Ligadura dinámica

La ligadura dinámica es el mecanismo que tiene el lenguaje para decidir qué método debe ejecutarse de todos los que implementa un objeto. En el ejemplo anterior, ¿cómo ha sabido el lenguaje qué método "update" tenía que ejecutar de los dos posibles? El algoritmo encargado de decirlo se llama ligadura dinámica.

1.2. Patrones de diseño

Un patrón de diseño consiste en la definición de un problema recurrente y de su solución simple y elegante.

Este concepto se utiliza en varias disciplinas, pero no se usará en la ingeniería del software hasta la publicación del libro *Desing Pattern*, escrito por Erich Gamma, Richard Helm, Ralph Johnson y John Vlisides (también conocidos por "Gang of Four", o GoF). En este libro se definen veintitrés patrones recurrentes y proponen soluciones sencillas.

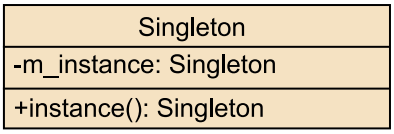
De estos veintitrés, veremos los más destacados y que podremos utilizar a la hora de diseñar e implementar código de un videojuego. Los patrones que hemos seleccionado son:

- Singleton
- Factory Method
- Proxy
- Iterator
- Observer

Aparte de estos patrones, también se explicará con detalle el patrón modelo-vista-controlador, muy útil para desarrollar todo tipo de aplicaciones que tienen interacción con el usuario.

1.2.1. Singleton

Este patrón garantiza que una clase sólo pueda tener una instancia dentro de la aplicación y establece su punto de acceso global.



Un ejemplo de su uso puede ser el controlador de algún elemento de hardware que necesitemos en nuestra aplicación, por ejemplo, el ratón, el teclado, el sonido, etc.

Otro uso es mantener el estado del videojuego (posiciones de los elementos móviles, estado de los avatares...).

Hay dos formas básicas de implementar un singleton:

- "eager instantiation": la instancia se crea al inicio de la aplicación.
- "lazy instantiation": la instancia no se crea hasta que no se necesite por primera vez.

Ejemplo de "eager instantiation":

MouseController.h

```
class MouseController
{
private:
   static MouseController m_instance;
   MouseController()
   {
      // init...
   }
public:
   static MouseController& getInstance() { return m_instance; }
};
```

MouseController.cpp

```
#include <MouseController.h>
static MouseController MouseController::m_instance;
```

Ejemplo de "lazy instantion":

MouseController.h

```
class MouseController
{
private:
   static MouseController* m_pInstance;
   private MouseController() {
      // init
   }
public:
   static MouseController& getInstance() {
      if (NULL == m_pInstance)
         m_pInstance = new MouseController();
      return *m_pInstance;
   }
};
```

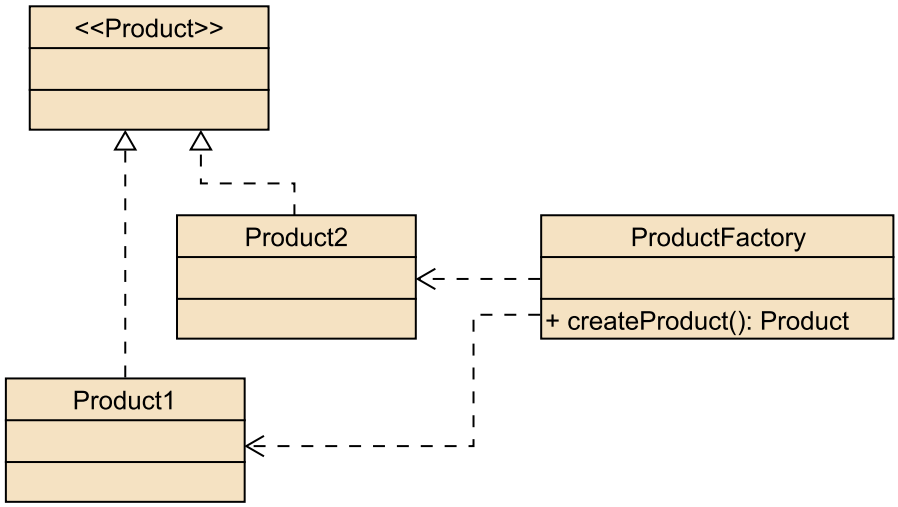
MouseController.cpp

```
#include <MouseController.h>
static MouseController* MouseController::m_pInstance = NULL;
```

En la API del engine Ogre3D es muy común el uso de este patrón.

1.2.2. Factory Method

Este patrón define la forma en que deben de crearse determinados objetos a partir de un objeto llamado fábrica (o Factory). Por ejemplo, al leer una cierta imagen desde un fichero, podemos delegar la elección del algoritmo de descompresión (gif, png, jpeg...) a un objeto Factory y que éste nos devuelva una instancia a un objeto imagen:

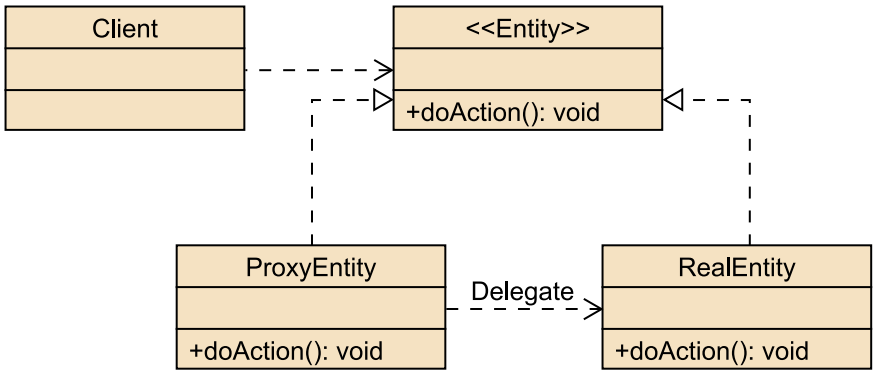


ImageFactory.h

```
#include <string>
class ImageFactory
{
private:
   bool testExtension(const std::string& filename, const std::string& ext) const
   {
      return filename.length() == filename.rfind(ext) + ext.length();
   }
public:
   Image* createImageFromFile(const std::string& filename)
   {
      if (testExtension(filename, ".png"))
         return new PNGImage(filename);
      if (testExtension(filename, ".gif"))
         return new GIFImage(filename);
      return NULL;
   }
};
```

1.2.3. Proxy

Este patrón permite el acceso indirecto a un objeto, permitiendo controlar las acciones que se realizan sobre el objeto real. Dos de estas acciones pueden ser: control de acceso de seguridad (*login/password*) y acceso a objetos remotos (implementación sencilla de programación distribuida).



Para implementar este patrón necesitamos de una interfaz que establezca el comportamiento de un objeto, una implementación real de la interfaz y un delegado:

IController.h

```
class IController {
public:
   virtual void keyPressed(char ch) = 0;
```

```
    virtual void keyReleased(char ch) = 0;
};
```

NetController.h

```
#include <IController.h>
class MainController
    : public IController
{
public:
    MainController() { ... }
// IController members
protected:
    virtual void keyPressed(char ch) {
        // update state
    }
    virtual void keyReleased(char ch) {
        // update state
    }
};
```

NetController.h

```
#include <IController.h>
class NetController
    : public IController
{
private:
    NetSocket m_ns;
public:
    NetController(NetSocket ns)
        : m_ns(ns)
    {
        m_ns.connect();
    }
// IController members
protected:
    virtual void keyPressed(char ch)
    {
        m_ns.executeCommand("keyPressed", ch);
    }
    virtual void keyReleased(char ch)
    {
        m_ns.executeCommand("keyReleased", ch);
    }
};
```

Una vez diseñada esta estructura, el siguiente código:

Ejemplo.cpp

```
class GUI
{
private:
   IController* m_controller;
public:
   GUI(IController* controller)
      :m_controller(controller)
   {
   }
   void onKeyPressed(char ch)
   {
      m_controller->keyPressed(ch);
   }
   ...
};
```

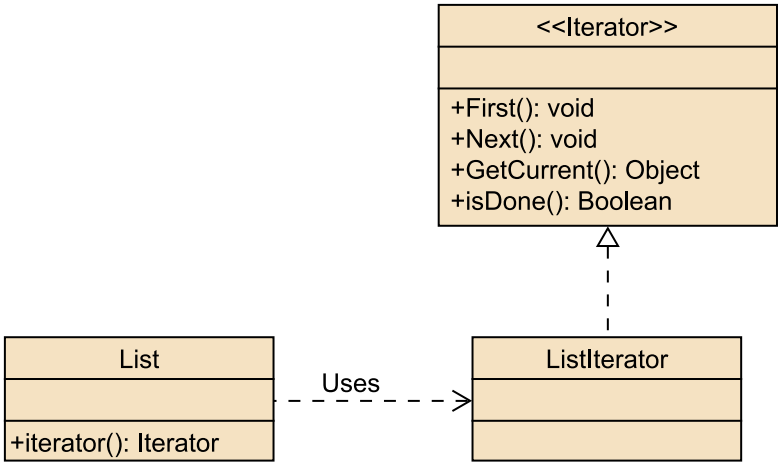
Funcionará sin ningún tipo de modificación para un juego standalone como para un juego en red:

```
class GUIFactory
{
public:
   static GUI* createSinglePlayer() {
      return new GUI(new MainController());
   }
   static GUI* createNetPlayer(NetSocket ns) {
      return new GUI(new NetController(ns));
   }
};
```

1.2.4. *Iterator*

El patrón *iterator* soluciona el problema de iterar sobre una serie de elementos. La solución inicial presentada por GoF indica que se debe crear una interfaz con los siguientes métodos:

- First(): inicia la enumeración.
- Next(): itera sobre el siguiente elemento.
- GetCurrent(): retorna una instancia al elemento actual.
- IsDone(): retorna un valor booleano indicando si se ha llegado al final de la enumeración.



En la librería estándar de C++, el patrón de iteración usado no sigue estas pautas. Cada tipo de contenedor de objetos (vector, list, queue, hash, set) define una clase interna llamada "iterator". Los métodos "begin" y "end" devuelven una instancia de la clase "iterator" y definen el inicio y el final de la iteración. Existen dos métodos análogos llamados "rbegin" y "rend" que permiten crear una iteración inversa y, en cuyo caso, se utiliza la clase "reserve_iterator".

La clase "iterator" y su análoga "reserve_iterator" contienen los siguientes métodos:

- operator*: retorna el elemento actual
- operator++: se mueve al siguiente elemento
- operator– –: se mueve al anterior elemento

Un ejemplo de la creación y recorrido de una lista en C++:

Ejemplo.cpp

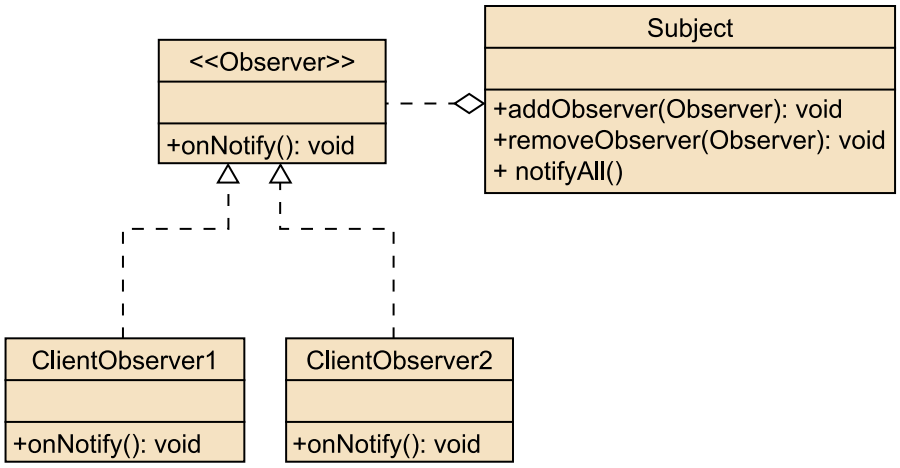
```
#include <iostream>
#include <list>
int main()
{
   std::list<int> l;
   for(int i = 0; i < 10; i++)
      l.push_back(i);
   std::list<int>::iterator it = l.begin();
   std::cout << "Todos:";
   while (it != l.end())
   {
      std::cout << " " << *it;
      it++;
   }
   std::cout << std::endl;
   std::cout << "Pares:";
```

```
    it = l.begin();
    while (it != l.end())
    {
       std::cout << " " << *it;
       std::advance(it, 2);
    }
    std::cout << std::endl;
    return 0;
}
```

Este patrón se usa bastante en diversas librerías que se pueden utilizar en la creación de videojuegos.

1.2.5. Observer

Este patrón, también llamado publish/subscribe, permite que varios objetos observen el estado de otro.



Una posible implementación es la siguiente:

- Creación de una interfaz que establezca los mecanismos de llamada.
- Un objeto que mantiene su estado e informa a los objetos que tiene suscritos de posibles cambios utilizando la interfaz creada.
- Varios objetos que se suscriben al anterior e implementan la interfaz.

Un ejemplo muy común en la programación gráfica con el que se puede utilizar este patrón es el de las librerías OpenGL y GLUT. Podemos crear el siguiente ejemplo:

DisplayListener.h

```
class DisplayListener
{
public:
```

```
    virtual void onDisplay() = 0;
    virtual void onKeyboard(unsigned char key, int x, int y) = 0;
    virtual void onKeyboardUp(unsigned char key, int x, int y) = 0;
    virtual void onIdle() = 0;
};
```

Display.h

```
#include <glut.h>
#include <set>
class Display
{
private:
    static Display m_instance;
    Display();
    std::set<DisplayListener*> m_listeners;
public:
    static Display& getInstance();
    void addListener(DisplayListener* listener);
    void removeListener(DisplayListener* listener);
private:
    static void static_display() { getInstance().display(); }
    static void static_keyboard(unsigned char key, int x, int y)
        { getInstance().keyboard(key, x, y); }
    static void static_keyboardUp(unsigned char key, int x, int y)
        { getInstance().keyboardUp(key, x, y); }
    static void static_idle() { getInstance().idle(); }
private:
    void display();
    void keyboard(unsigned char key, int x, int y);
    void keyboardUp(unsigned char key, int x, int y);
    void idle();
};
```

Display.cpp

```
#include <Display.h>
Display::Display()
{
    glutDisplayFunc(static_display);
    glutKeyboardFunc(static_keyboard);
    glutKeyboardUpFunc(static_keyboardUp);
    glutIdleFunc(static_idle);
}
Display& Display::getInstance()
{
    if (NULL == m_instance)
```

```
      m_instance = new Display();
   return *m_instance;
}
void Display::addListener(DisplayListener* listener)
{
   m_listeners.insert(listener);
}
void Display::removeListener(DisplayListener* listener)
{
   m_listeners.remove(listener);
}
void Display::display()
{
   std::set<DisplayListener*>::iterator it = listeners.begin();
   while (it != listeners.end())
   {
      (*it)->onDisplay();
      it++;
   }
}
...
```

Y una sencilla forma de utilizar el patrón:

Game.h

```
#include <DisplayListener.h>
class Game
   : public DisplayListener
{
public:
   Game()
   {
      Display::getInstance().addListener(this);
   }
   virtual ~Game()
   {
      Display::getInstance().removeListener(this);
   }
// DisplayListener members
protected:
   void onDisplay();
   void onKeyboard(unsigned char key, int x, int y);
   void onKeyboardUp(unsigned char key, int x, int y);
   void onIdle();
};
```

1.2.6. Modelo-vista-controlador

Este patrón no pertenece a la lista que propusieron los GoF. Empezó a utilizarse con el lenguaje SmallTalk y se definió a finales de los años setenta. Su arquitectura básica se descompone de tres elementos:

- Modelo: define el estado de la aplicación. En este elemento se mantiene toda la información necesaria. Esta información puede guardarse (serialize) para mantener el estado de la aplicación entre ejecuciones (guardar una partida podría hacer uso de esta característica).

- Vista: elemento que presenta la información del modelo (*render*) y recibe los eventos de la interfaz (teclado, ratón...).

- Controlador: trata los eventos de la vista y modifica el estado del modelo. También informa a la vista de los posibles cambios ocurridos en el modelo.

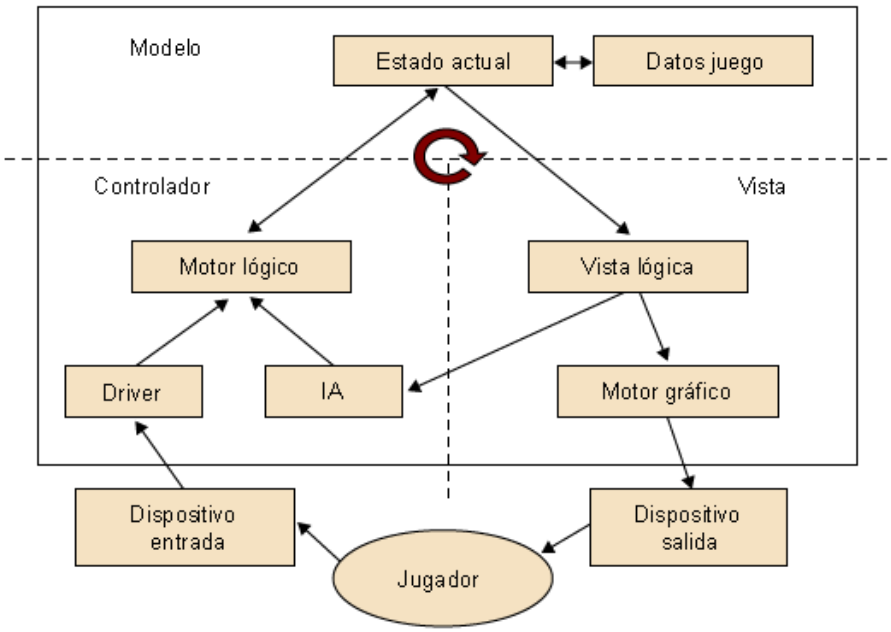


Figura 1. Esquema modelo-vista-controlador

En el caso de un videojuego, estos tres elementos los podemos emparejar con las siguientes definiciones:

- Modelo: información que define el estado del videojuego (lista de elementos móviles, avatares, puntos, pistas descubiertas...) y contiene los datos sobre el mundo donde se está desarrollando la acción.

- Controlador: entidad que se encarga de la lógica del videojuego (implementa todas las operaciones que puede o no puede hacer un determinado avatar, motor de física...). Aquí también entraría toda la parte de inteligencia artificial y toma de decisiones.

- Vista: entidad encargada de renderizar a partir de la información del modelo lo que el jugador debe de ver, oír y sentir. A partir de la entrada de la interfaz (teclado, ratón...), informará al controlador para que modifique el estado.

2. El programa principal de un videojuego

El programa principal de un videojuego es el encargado de coordinar todos los diferentes componentes que hemos visto en los módulos anteriores (sonido, entrada de datos, gráficos, red) con el motor lógico, que veremos en el apartado siguiente y la inteligencia artificial, que veremos en el próximo módulo.

En este apartado vamos a explicar en primer lugar cómo se segmenta un juego en diferentes estados. Los estados nos permiten separar ciertas tareas que son necesarias para la puesta a punto del mismo.

Dentro de cada uno de los estados se encuentra un bucle que se ejecuta continuamente. El estado que se encuentre activo nos indicará si estamos en el menú principal esperando que el usuario seleccione cómo quiere jugar o bien si nos encontramos en el estado "de juego", donde transcurre la acción de la partida.

Para finalizar el apartado, vamos a analizar paso a paso la estructura interna del bucle principal del estado "de juego", para entender cómo se integra la información que recibimos y enviamos a todos los componentes que utilizamos durante el juego, básicamente: entrada, sonido, gráficos y red.

2.1. Estados de un videojuego

Entendemos por un estado de un videojuego un módulo independiente que realiza una funcionalidad específica y que tiene su propio gestor de eventos, su bucle principal y su *engine* gráfico para mostrar la información por pantalla.

Nota

No confundir estos estados con los estados que se utilizan en una entidad de inteligencia artificial para la toma de decisiones o con el estado global de una partida, que se refiere al valor que tienen las variables en cada momento concreto del juego.

La subdivisión en estados depende de cómo vayamos a estructurar nuestro juego. No existen reglas en las que basarse para decidir cuándo es conveniente separar dos tareas en dos estados diferentes, pero principalmente dependerá de si estás pueden compartir gestores de eventos y los diferentes motores del juego, o no.

Éstos son algunos ejemplos de posibles estados que podemos definir en un videojuego:

- La configuración del sistema.
- El establecimiento de la conexión de la red.

- Los vídeos de presentación o de transición, donde el usuario no tiene ningún tipo de control.
- El bucle principal del juego (o *mainloop*), donde se desarrolla la acción de la partida.

Por un lado, cada estado tiene un gestor de eventos propio que se encarga de procesar todos los eventos que estén permitidos mientras nos encontremos en el estado. Por ejemplo, en el estado de conexión de red tendremos un sistema que simplemente tratara los eventos de red, e ignorará otro tipo de eventos, o en un estado en que tenemos pausado el juego, nuestro gestor de eventos tan sólo reaccionará cuando se pulse una determinada orden.

Además, cada estado tiene que renderizar la pantalla para mostrar la información necesaria para que el usuario pueda interactuar en cada estado. En el caso del menú nos puede bastar con un simple *engine* 2D que sea capaz de mostrar las opciones al usuario; en cambio, cuando estemos en el bucle principal del juego posiblemente necesitaremos usar un *engine* gráfico mucho más complejo, como los estudiados en el módulo de gráficos 3D.

2.1.1. Diagrama de estados

Un diagrama (o máquina) de estados finitos define una serie de estados y una relación entre ellos. De todos los estados, sólo uno de ellos está activo y, a partir de una serie de entradas, se realizan transiciones entre estados y cada cambio genera una serie de salidas.

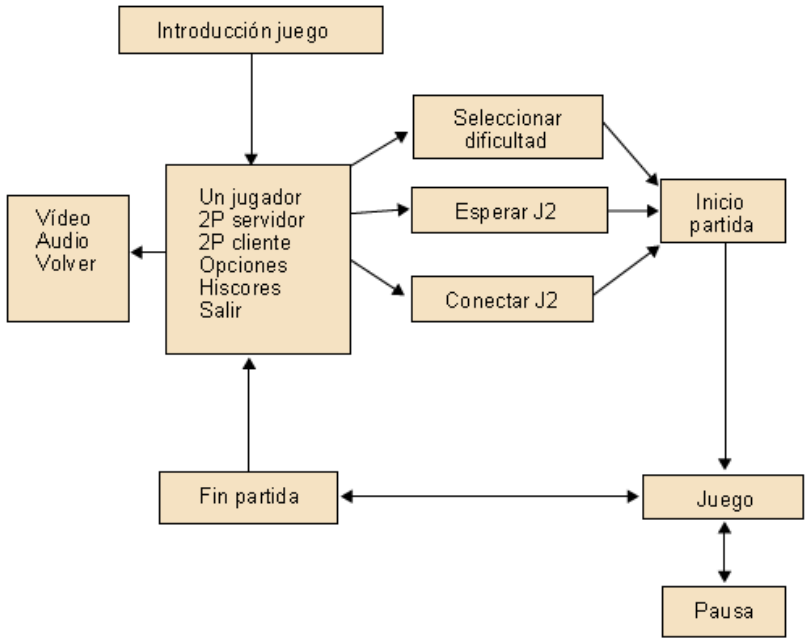
Podemos decir que la máquina de estados finitos más estudiada es la máquina de Turing, una máquina teórica desarrollada por Alan Turing en 1936 con la cual se pudo definir formalmente el concepto de algoritmo.

Las características básicas de este tipo de estructuras:

- Se pueden implementar con un código muy sencillo y limpio.
- Son estructuras fáciles de depurar.
- El código de control es mínimo y no generan "overhead".
- Son muy intuitivas.

Las máquinas de estado finito se pueden representar mediante grafos, donde cada vértice corresponde con un estado y las aristas definen las transiciones entre estados. En las aristas podemos incluir información como, por ejemplo, condiciones que deben cumplirse.

Mediante un diagrama de estados podemos visualizar las posibles transiciones que existen entre los diferentes estados del juego. Éste es un ejemplo de un posible diagrama de estados para un juego:



- Después de mostrar algunos vídeos introductorios en un primer estado, el estado que se carga es el del menú, donde tratamos sólo aquellos eventos que nos permitan seleccionar qué queremos hacer en el juego: jugar nosotros solos, crear un servidor de juegos, conectarnos a un servidor, modificar las opciones de juego o salir.

- En las pantallas del menú se configura el comportamiento que tendrá nuestro juego. Podemos modelar cada una de estas opciones dentro de un estado propio o ponerlas en un estado más genérico que incluya toda la configuración del juego.

- Una vez el usuario haya seleccionado un tipo de juego, podemos empezarlo mostrando una animación o vídeo (para posicionar los personajes, etc...), y finalmente pasamos el control al estado que permite jugar al usuario. Allí todo gira alrededor del bucle principal, que es donde se encuentra implementado el auténtico juego.

- Una vez finalice el juego, devolveremos el control al estado del menú.

Nota

Los diagramas de estado también se pueden utilizar para la implementación de la lógica de todas las entidades de un juego (enemigos, puertas, puzles...). Para los objetos sencillos es una buena opción (puerta cerrada - abriendo - abierta - cerrando), pero para entidades más complejas, mejor utilizar otras técnicas como las que se verán en el punto 2.1.3. o las que se verán en el contenido de *Inteligencia artificial*.

2.1.2. El gestor de estados

La forma tradicional de implementar un sistema de estados ha sido siempre utilizando comandos bucles 'if' y 'case'. Sólo tenemos un programa principal, y en cada paso del bucle buscamos el estado en el que estamos y allí ejecutamos el código para tratarlo:

main.c

```
int main()
{
   // Inicializaciones...
   for(;;) // Bucle principal
```

```
    {
       switch(estadoActual)
       {
          case MENU_JUEGO:...
          case OPCIONES:...
          case JUEGO:...
       }
    }
    return 0;
}
```

El problema de esta implementación es que no nos permite tener varios motores gráficos o gestores para cada uno de los estados del juego. Además, cada vez que recibimos una entrada de teclado o de red, tenemos que mirar si esto provoca un cambio de estado, con lo que es más fácil tener incoherencias en el sistema. Todo esto hace que el programa principal sea difícil de seguir y, por tanto, difícil de mantener y depurar.

Otra alternativa más eficiente es utilizar la ingeniería del software explicada en el apartado anterior para diseñar e implementar un gestor de estados que nos permita modular el comportamiento de los mismos. Un gestor de estados se compone de dos partes:

- Cada estado es una clase que implementa una serie de funciones comunes: iniciar el estado, finalizar el estado, actualizar la pantalla, gestionar determinados eventos,...

- Tenemos un controlador de estados que se encarga de indicar cuál es el estado que se encuentra activo en todo momento y permite cambiar a otro estado.

Para crear los diferentes estados podemos basarnos en la siguiente interfaz:

IState.h

```
class IState
{
public:
public:
   virtual void enter() = 0;
   virtual void leave() = 0;
   virtual void pause() = 0;
   virtual void resume() = 0;

   virtual void update(double timeElapsed) = 0;
};
```

A partir de esta interfaz, podemos crear una serie de clases que definan nuestros estados, como por ejemplo: MenuState, JugarState, OpcionesState,...

MenuState.h

```
#include <IState.h>
class MenuState
   : public IState
{
private:
   StateManager* m_manager;
   int options;
public:
   MenuState()
{ }
// IState members
public:
   void enter() { /* Dibujar menú */ }
   void leave() { /* Limpiar menú */ }
   void update(double timeElapsed)
   {
      // Mirar entrada de datos y actuar
   }
};
```

Para coordinar todos los estados, deberemos crear un controlador de estados, al que podemos llamar StateManager. El StateManager debe proporcionar funciones para seleccionar el estado actual o guardar un estado temporalmente (por ejemplo, si hacemos una pausa). Pasamos a ver un ejemplo de la implementación de un StateManager:

StateManager.h

```
class StateManager
{
private:
   std::hash<std::string, IState*> m_states;
   IState* m_current;
public:
   StateManager() : m_current(NULL) { }
   void registerState(const std::string& name, IState* state)
   {
      m_states[name] = state;
   }
   void changeTo(const std::string& name)
   {
      if (m_current != NULL)
```

```
            m_current->leave();
        m_current = m_states[name];
        m_current->enter();
    }
    void update(double timeElapsed)
    {
        if (m_current != NULL)
            m_current->update(timeElapsed);
    }
};
```

2.1.3. Diagrama de estados y subestados

Hemos visto que un diagrama de estados mantiene un estado concreto y define qué transiciones existen entre varios estados y qué acciones deben realizarse en estos cambios.

Es posible que dentro de un estado queramos definir varios tipos de subestados. Por ejemplo, podemos tener un estado que define la navegación por el menú del juego. En el menú de opciones, podemos presentar varios subestados: sonido activo - desactivando - desactivado - activando (haciendo *fade-in* y *fade-out* del sonido como parte de una animación).

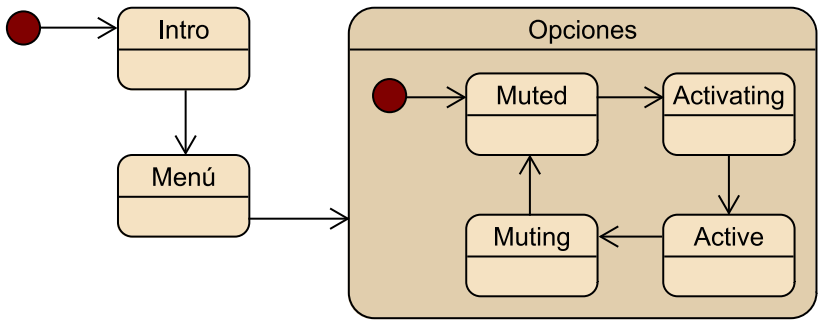


Figura 2. Ejemplo de un diagrama de estados con subestados

Este tipo de estructuras necesitan de una implementación más compleja. Para ello, es una buena opción utilizar librerías que permitan facilitar esta tarea. Por ejemplo, usando *boost statechart*.

Sistemas con múltiples estados simultáneos

En muchos de los juegos podemos encontrarnos que tenemos diferentes estados para un objeto, como por ejemplo saltar y golpear, donde, en cada uno de estos estados, el objeto requiere de un tipo de animación, tiene su propia lógica y tiene una duración asociada al mismo. El problema reside en que muchas veces queremos que el objeto pueda encontrarse en dos estados de forma simultánea, como por ejemplo saltar y golpear a la vez, lo que implica que debemos controlar qué pasa en ambos estados.

Hay varias formas de tratar con sistemas con estados simultáneos:

- La más simple es eliminar la concurrencia y transformar las combinaciones de estados en nuevos estados exclusivos. Por ejemplo, podríamos separar saltar, golpear y saltar+golpear en tres estados diferentes con sus transiciones entre ellos y usar una máquina de estados sencilla. Esta solución no requiere código extra, pero en cambio es poco escalable: si hay muchas opciones la máquina de estados puede crecer exponencialmente.

- Una segunda opción es utilizar los diagramas de estados con subestados como los vistos en el apartado 2.1.3. La complejidad en este caso se encuentra en la programación de los estados, pero en cambio es un sistema muy extensible y fácil de controlar.

- La tercera opción es utilizar un sistema de *scripting* como el que se presenta en el apartado 5.

2.2. El controlador principal

Si utilizamos un sistema basado en estados, necesitamos crear otra clase especial que nos permita acceder a los componentes únicos del sistema (el gestor de entrada, de red,...) y a los datos globales del sistema. A esta clase la llamaremos el Controlador Principal, y tiene dos requisitos básicos:

- Sólo puede existir una instancia de la clase, que es creada por el programa principal.
- Debe ser visible desde todos los estados que hayamos definido en el sistema para que éstos puedan utilizar los recursos compartidos.

Si utilizamos esta clase como el elemento central de nuestro programa, el código del programa principal nos quedará extremadamente simple:

main.cpp

```
#include <CoreManager.h>
int main()
{
   CoreManager* core = new CoreManager();
   core->run();
   delete core;
}
```

Por otro lado, os presentamos una implementación de un controlador principal. Fijaos que simplemente cumple con los requisitos que hemos detallado al principio. Se trata de un singleton, con lo que sólo se puede instanciar una vez, e incluye todos los recursos globales y las funciones para acceder a ellos.

CoreManager.h

```
class InputManager;

class CoreManager
   : public Singleton<CoreManager>
{
public:
   static CoreManager* getSingletonPtr();
   static CoreManager& getSingleton();

public:
   CoreManager();      // Inicialización de los recursos
   ~CoreManager();     // Finalización de los recursos

public:
   int run();          // Inicio del gestor de estados

public:
   // Funciones para acceder a los recursos globales
   InputReader* getInputReader() { return m_inputReader; }

private:
   // Recursos globales
   InputReader* m_inputReader;
}
```

En la programación de la clase tenemos que poner especial énfasis en dos funciones: el constructor de la clase, donde vamos a inicializar todos los recursos, y la función "run", que es llamada por el programa principal para encender al gestor de estados y pasar el control del juego al primer estado.

2.3. El bucle principal del juego

El bucle principal del juego es la parte que se ejecuta continuamente y es donde se produce todo el desarrollo de la partida.

El bucle se puede ejecutar de dos formas posibles:

- Podemos ejecutar el bucle cada vez que se genera un evento especial, como por ejemplo cada vez que tenemos que refrescar la pantalla.

- Otra opción es estar ejecutándolo todo el tiempo utilizando alguna técnica multihilo.

En ambos casos es necesario que sincronicemos las repeticiones del bucle con un *timer*, ya que en la mayoría de los juegos necesitamos mantener una velocidad constante de ejecución que sea independiente de la capacidad gráfica y de los recursos que tenga el sistema (es decir, que el juego funcione igual independientemente del hardware que tenga el sistema).

Una forma de implementar el *timer* es añadir al principio del bucle una comprobación que mire cuánto tiempo ha pasado desde la última vez que actualizamos el estado global Si no ha pasado el suficiente tiempo, podemos simplemente saltarnos el bucle y esperar hasta que toque actualizar. El número de veces que se ejecuta el bucle debería ser mayor de 25 veces por segundo para tener fluidez en la representación gráfica.

Nota

Parece ser que el ojo humano es capaz de ver un vídeo sin saltos cuando se emiten un mínimo de 25 imágenes por segundo.

Antes de empezar a ejecutar el bucle principal del estado donde vamos a jugar la partida, es muy importante que se cumplan los siguientes requisitos:

- Cada elemento hardware necesita su propio *driver* específico para que se puedan interpretar todas las señales que se originen en el controlador.

- Todos los recursos deben estar cargados e inicializados.

- Todos los sistemas deben estar inicializados y tener acceso a los datos que necesitan para poder trabajar (Lógica, Gráficos, Red, Sonido, IA, Física, etc.).

El bucle principal está compuesto por tres fases principales: la recogida de datos, el cálculo del nuevo estado global del juego y la redistribución de los cambios. Estas fases las podemos desglosar más detalladamente en la siguiente secuencia:

- Recoger la entrada de datos: lo primero que hacemos es mirar si el usuario ha realizado alguna acción desde la última vez que entramos en el bucle. En el caso de un sistema *buffered* comprobaremos si ha habido algún cambio en el estado del *buffer*. En el caso de un sistema *unbuffered* comproba-

remos si tenemos algún aviso de alguna entrada que nos haya dejado el controlador de entrada.

- Recoger los datos de la red: si tenemos un juego multijugador, comprobaremos si se han recibido nuevos datos. Miraremos el *buffer* de paquetes y nos quedaremos con aquellos que nos interesen en el estado en el que nos encontremos. Es importante recordar que no bloquearemos el sistema esperando los paquetes, tan sólo vamos a tratar aquellos que ya hayan llegado.

- Recoger las decisiones de la inteligencia artificial: en esta fase recibimos las nuevas posiciones de los objetos que se encuentran controlados por la inteligencia artificial. Existen dos formas de obtener esta información: la primera es lanzar una función que nos determine las decisiones de la IA en este momento, y la segunda es tener un proceso paralelo que siempre esté calculando decisiones y al que vamos a consultar qué es lo mejor que ha decidido hasta este momento.

- Procesar todas las entradas con el motor lógico del juego. Esta fase sólo se puede ejecutar una vez hemos recibido la entrada de las tres fases anteriores. La podemos subdividir en dos subfases:
  - Control de física y de colisiones: calculamos si las posiciones y acciones que se piden se pueden llevar a cabo o si existen interacciones entre los elementos. En este caso el sistema de física decide cuál es el movimiento real de los elementos. La salida de esta fase indica las posiciones finales de los objetos en el mundo. Los cálculos de esta fase siempre se realizan en un tiempo máximo definido que permita la sensación de fluidez al usuario.

  - Control de la lógica del juego: miramos que se estén cumpliendo las reglas del juego y que el comportamiento de los elementos esté dentro de los márgenes. También miramos si el juego ha finalizado en este último paso. Finalmente, actualizamos todas las variables globales de la partida: tiempos, puntuaciones...

- Distribución de los resultados: la salida del sistema de lógica (posiciones, variables...) se tiene que hacer llegar a todos los componentes que lo necesiten, al sistema gráfico, a la inteligencia artificial y a todos los jugadores conectados por red.

- Actualización de los dispositivos de salida: finalmente reproducimos los sonidos que corresponda y actualizamos el motor gráfico para que muestre todos los cambios. Puede ser necesario que tengamos que hacer algunos cálculos extras, como por ejemplo recolocar la cámara para que siga a nuestro personaje.

La implementación de este bucle principal se debe realizar íntegramente dentro de cada estado donde tenga lugar el desarrollo de la partida. En otros estados podemos eliminar algunos de estos puntos (normalmente el de lógica).

A continuación presentamos un ejemplo de la implementación de un bucle de juego principal:

GameState.h

```
#include <IState.h>

class GameState
   : public IState
{
private:
   StateManager* m_manager;
   int options;
public:
   MenuState()
{ }
// IState members
public:
   void enter() { /* Inicializar nivel del juego */ }
   void leave() { /* Liberar recursos del nivel del juego */ }
   void update(double timeElapsed)
   {
      CoreManager& core = CoreManager::singleton();
      core.lecturaDatosEntrada(timeElapsed);
      core.lecturaDatosDeRed(timeElapsed);
      core.lecturaDatosIA(timeElapsed);
      core.getFisica().update(timeElapsed);
      core.getLogica().updateReglasDeJuego(timeElapsed);
      core.enviarCambiosDeEstadoPorRed(timeElapsed);
      core.modificarDispositivosSalida(timeElapsed);
   }
};
```

3. El motor lógico de un videojuego

En los módulos anteriores hemos tratado algunas de las tecnologías que son necesarias para poder desarrollar un videojuego. La mayoría de estas tecnologías son genéricas y pueden utilizarse en otro tipo de aplicaciones.

En el primer módulo del curso definimos un juego como un conjunto de objetivos que tenemos que conseguir siguiendo unas reglas. La implementación de estas reglas, y por tanto del juego propiamente dicho, se lleva a cabo en lo que llamamos el motor lógico de un videojuego (también llamado motor de lógica). Tecnológicamente hablando, el motor lógico es el componente más importante de un videojuego y se encarga, entre otras cosas, de:

- Inicializar el contenido del juego y crear el mundo donde se desarrollará la acción.
- Determinar cuándo empieza y cuándo termina una partida.
- Implementar las reglas que controlan las acciones de los jugadores y controlar que estas reglas se cumplen durante el desarrollo de la partida.
- Permitir que los jugadores interactúen con el entorno sin romper estas reglas.
- Controlar el estado de todos los elementos que están participando en la partida, desde las posiciones de los avatares hasta las variables globales de la partida: tiempo de juego, puntuación, resultados...

3.1. El mundo lógico del juego

El motor lógico utiliza su propia descripción del mundo donde se desarrolla la acción, completamente diferente del que hemos visto anteriormente en el apartado gráfico. Este mundo está compuesto principalmente de las descripciones y los atributos de los elementos que se encuentran en él. Por ejemplo, los personajes los representamos mediante cajas con posiciones y vectores, sin tener en cuenta la forma real que tienen ni la animación que se esté reproduciendo en este momento.

Esta simplificación es necesaria para que el ordenador pueda interpretar en todo momento el estado del juego. Por ejemplo, para el sistema de inteligencia artificial no es necesario conocer si un personaje es un orco de color verde o rojo para tomar una decisión de qué hacer con él, simplemente está interesado en el tamaño, la posición y la dirección en la que está mirando este orco.

Reflexión

Durante el desarrollo de un videojuego, a veces es interesante poder observar el mundo lógico y el mundo físico/gráfico por separado o conjuntamente. Esto permite trabajar con más detalle con uno u otro y depurar mejor los posibles problemas existentes en cada uno, ya que, de otra manera, podríamos tener un exceso de información en pantalla.

La representación visual del mundo lógico se hace normalmente con gráficos sencillos basados en líneas y textos. Esta representación se tiene que diseñar con cuidado, intentando simplificar la información que mostramos, pero que a la vez podamos ver representado todo aquello que el sistema está observando del juego.

3.1.1. Discretización del mundo

Una de las primeras tareas que tenemos que hacer para definir el mundo lógico es realizar una discretización del mundo real que vamos a utilizar. Dependiendo del tipo de juego y de los recursos del sistema, utilizaremos una simplificación con más o menos detalle.

Nota

Ya vimos técnicas parecidas de discretización del mundo (octrees, BSP) en los módulos que trataban los videojuegos 2D y 3D.

La discretización permite organizar con mayor eficiencia la distribución de los objetos en el mundo y nos permite asignar diferentes propiedades a los polígonos que componen el mundo discreto. Por ejemplo, en un juego de coches podemos asignar una propiedad a cada polígono para representar la fricción del suelo.

Nota

Los polígonos que utilizamos para la discretización del mundo y los polígonos que utilizamos en los gráficos pueden coincidir, aunque esto no es una condición necesaria. En algunos casos nos puede interesar tener polígonos más grandes o más pequeños en el mundo lógico para proporcionar información extra al sistema.

Mediante la discretización, también podemos crear un grafo que determine los posibles caminos entre dos puntos del mundo lógico, y mediante algoritmos de búsqueda de caminos podemos calcular las rutas de los elementos móviles del sistema.

Existen dos técnicas comunes para discretizar el mundo, sobreponer un *grid* regular encima del mundo o extraer una malla de navegación a partir de la geometría:

Grid regular

Un *grid* es una teselación de polígonos. Cuando lo sobreponemos encima del mundo, podemos determinar qué parte del mundo está dentro o fuera de nuestro *grid*, y además podemos segmentar la parte interior en polígonos regulares.

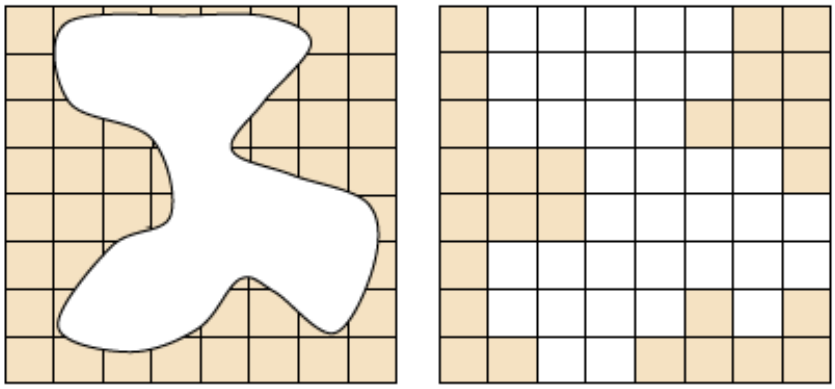


Figura 3. Ejemplo de un mapa con un *grid* rectangular (izquierda) y del resultado de la discretización (derecha)

Cuanto más detalle utilicemos para el *grid*, mayor será la resolución de nuestro sistema, pero también necesitamos más espacio para guardarlo y mayor será el tiempo para calcular caminos sobre el mismo. También podemos utilizar algunas de las técnicas vistas anteriormente en el módulo de gráficos para tener diferentes niveles de resolución, como por ejemplo octrees o BSPs, aunque tendremos que tener en cuenta que el uso de diferentes niveles jerárquicos puede complicar el proceso de búsqueda de caminos.

Nota

Un *grid* regular sólo puede estar compuesto por tres tipos de polígonos: triángulos equiláteros, cuadrados o hexágonos.

Para el cálculo de caminos, conectaremos con una línea el centro de todo par de polígonos adyacente. Para esto necesitaremos el grafo de todas las rutas existentes en este *grid*. En el caso del *grid* cuadrangular, es importante considerar si podemos ir directamente a las casillas que se encuentran en diagonal.

Malla de navegación

Una malla de navegación es una partición de la geometría del mundo lógico en polígonos convexos como los que acabamos de describir. En otras palabras, es un conjunto de polígonos que cubre por completo el mapa, donde cada par de polígonos sólo comparte dos puntos y una línea. Cada polígono representa un punto de un camino que está.

El hecho de que sean convexos garantiza que sólo nos podamos mover en línea recta dentro de un polígono. Hay gran cantidad de algoritmos que nos permiten realizar esta subdivisión en polígonos convexos, sobre todo a nivel de triángulos.

En este caso podemos asignar los puntos de los caminos desde dos posiciones, en el centro de cada polígono o en el centro de cualquier arista que conecte dos polígonos diferentes.

Nota

Hay varios tipos de triangulación, pero quizás el más utilizado sea el que cumple la condición de Delaunay.

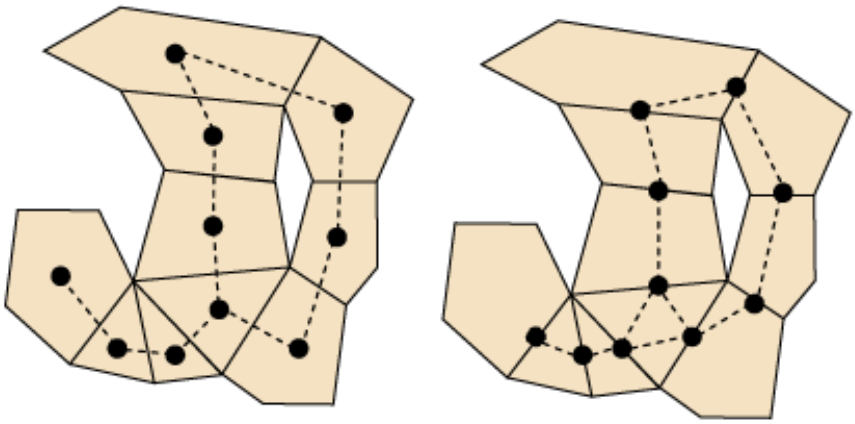


Figura 4. Ejemplo de una malla de navegación con los caminos usando los centros de los polígonos (izquierda) o las aristas (derecha)

3.2. Tareas del motor de lógica

El motor de lógica es el encargado de controlar el desarrollo del juego. Recibe de todos los sistemas de entrada (jugadores o IA) las acciones que han tomado los jugadores en un turno, decide qué sucede con ellas, calcula el nuevo estado global del juego y devuelve este estado al sistema de lógica y a los dispositivos de salida para informar de lo que ha ocurrido.

Vamos a estudiar con más detalle una posible secuencia de todo lo que sucede en el motor de lógica cada vez que tenemos que actualizar el sistema en cada iteración o 'turno'. El orden de las tareas que ponemos a continuación no es estricto y se puede reestructurar para adaptarse a las necesidades de cada juego en concreto.

3.2.1. Aplicar movimiento de los elementos del juego

Para poder cambiar el estado del juego, el motor de lógica necesita recibir por parte de todos los elementos 'móviles' o 'activos' del juego la información de las acciones que han tomado en cada momento (si es que han decidido realizar alguna).

Una de las primeras tareas que debemos realizar en cada iteración de este motor es calcular las nuevas posiciones de todos los elementos. Tenemos dos posibles orígenes de movimiento:

- Los elementos controlados por jugadores: la información de movimiento puede venir dada de varias maneras (posiciones relativas, absolutas, velocidad o aceleración...). Esto depende de cómo interpretemos la entrada del usuario y el estado actual del jugador. Por ejemplo, si tenemos un coche moviéndose con una cierta velocidad, aunque el usuario no realice ninguna acción en esta iteración tenemos que considerar que el elemento se sigue moviendo.

- Los elementos no controlados por jugadores: estos movimientos nos lo proporciona el sistema de inteligencia artificial. Algunos de ellos son movimientos cíclicos, con una ruta prefijada, y por tanto fácil de calcular. Otros movimientos son más complejos y requieren de la observación del estado del mundo para poder tomar decisiones. El sistema de inteligencia artificial puede enviar la información de varias maneras, así que en este caso también tendremos que interpretar el movimiento y traducirlo a posiciones reales para cada elemento.

Una vez hemos calculado todas las posiciones, entra en escena el cálculo de colisiones entre los elementos. Aunque lo hayamos introducido en el módulo de gráficos, la física y el control de colisiones los realiza el motor lógico usando la información contenida en su representación del mundo y no el motor gráfico.

Nota

Las técnicas básicas para saber si dos objetos colisionan se encuentran explicadas en el módulo que trata de videojuegos 2D.

La función que calcula las colisiones nos devuelve las posiciones finales de todos los movimientos realizados en esta iteración. Sólo queremos remarcar que es importante decidir qué elementos tienen prioridad sobre otros al calcular las nuevas posiciones, ya sea utilizando sus propiedades físicas (un objeto con más masa y velocidad ocupará el lugar de otro con menos) o utilizando prioridades que asignemos nosotros a los objetos.

Nota

Tenemos libertad, ¡estamos implementando un videojuego y no un simulador de física!

Una vez pasado este filtro, falta una última comprobación donde miramos que las posiciones finales de los elementos cumplan con las reglas del juego, es decir, que la posición final de cada elemento se encuentre dentro de una zona permitida para él.

3.2.2. Aplicar acciones de los elementos del juego

La segunda tarea del motor de lógica es la de llevar a cabo todas las otras acciones que realizan los elementos descritos en el apartado anterior. La naturaleza de estas acciones puede ser muy variada, ya sea lanzar un hechizo, chutar un balón o construir una granja.

Lo primero que tenemos que hacer es ver si las acciones se pueden llevar a cabo. Para eso tenemos que comprobar que el elemento que realiza la acción tiene todo lo necesario para llevarla a cabo. Por ejemplo, en el caso del hechizo, que el personaje tenga los atributos u objetos necesarios (energía, reactivos,...), o en el caso del jugador, que tenga la posesión del balón.

Reflexión

Debemos tener en cuenta que el estado de un elemento puede intervenir en el estado de otro. Así, si tenemos un jugador que está dando un puñetazo mientras salta y otro objeto lo mata, el jugador ya no podrá herir a nadie mientras cae al suelo.

En segundo lugar, tenemos que mirar que la acción no infrinja las reglas del juego y que, por tanto, se pueda efectuar. Por ejemplo, aunque tengamos suficiente dinero para construir la granja, el sistema tiene que comprobar que la construimos en una posición donde esté permitida por las reglas del juego.

Reflexión

Es importante decidir quién realiza primero cada acción, sobre todo en el caso de que la acción afecte a otros elementos del juego. Una acción puede cambiar los atributos de otro elemento y evitar que éste pueda llevar a cabo la suya. Por ejemplo, si dos jugadores se atacan entre sí, es posible que el primero acabe con el segundo y que éste ya no puede devolver el golpe.

Por lo tanto, es muy importante decidir la prioridad de las acciones de los elementos para no perjudicar a nadie. Existen varias formas de decidir el orden de las acciones:

- Una forma es utilizar un sistema *round-robin*, en el que siempre se utiliza el mismo orden de procesamiento de acciones.

Round-robin

Round-robin es un algoritmo para repartir un recurso entre varios procesos de manera equitativa.

- Otra posibilidad es seleccionar el orden en que se llevan a cabo las acciones de forma aleatoria. En este caso sólo utilizaremos la aleatoriedad cuando las acciones se solapen, en otro caso podemos seguir con un *round-robin*.

- Finalmente, la opción más elegante es asignar prioridades a las acciones. Esta opción se puede complementar con las dos anteriores.

Una vez realizada cada acción, se computan los cambios del entorno debidos a la misma y se guardan en todos los elementos que se han visto afectados.

Reflexión

Cada vez es más importante tener en cuenta la programación multihilo porque los nuevos ordenadores y videoconsolas se montan con varios procesadores.

Si se quiere aprovechar al máximo la potencia de estos sistemas, es necesario trabajar con algoritmos concurrentes y tener mucho cuidado con los errores comunes que tienen asociados.

3.2.3. Cálculo del nuevo estado del juego

Finalmente, una vez todos los elementos se han movido y han actuado, es necesario recalcular el estado del sistema y actualizar las variables globales. Este paso se realiza al final de todas las acciones para poder integrar el resultado de la actuación de los elementos de esta iteración. Algunas de las tareas que debemos realizar en este paso final son:

- Incrementar el tiempo del juego proporcionalmente. Por ejemplo, hay juegos donde es importante la distinción entre noche y día.

- Comprobar si el juego ha llegado a su fin. En este caso tendremos que informar al bucle principal que tenemos que cambiar de estado para dejar de procesar las entradas del usuario.

- Comprobar si se ha cumplido algún objetivo intermedio y, por lo tanto, si es necesario cambiar.

Toda esta información se deja almacenada en las estructuras de datos del sistema y se avisa a los componentes que lo necesiten (IA, gráficos,...) de todos los cambios que se han producido para que se puedan actualizar con el nuevo estado del juego.

4. Gestión de los datos de un videojuego

Los datos que vamos a utilizar en un videojuego son tan importantes como el motor lógico explicado en el punto anterior. El motor lógico implementa las reglas, pero sin estos datos no tendremos nada con que jugar.

Los datos del juego se pueden subdividir en dos grandes grupos:

- En primer lugar, tenemos todos aquellos datos que se utilizan en los motores gráfico y sonoro. Estos datos pueden incluir música, efectos, o modelos tridimensionales, texturas o animaciones. A estos datos los llamaremos **entidades**.

- Por otro lado, tenemos otro grupo de datos que describe todos los parámetros de los elementos que intervienen en el juego. A estos datos los llamaremos **objetos** y pueden incluir desde posiciones, direcciones y velocidades, hasta atributos de los personajes. Estos datos son utilizados por el motor lógico para poder evaluar continuamente el estado del juego.

Cada estado del juego tendrá sus propias entidades y objetos locales. Además, podrá utilizar todos aquellos que se encuentren definidos de forma global a través del uso del controlador principal del juego que hemos explicado anteriormente.

En este apartado vamos a describir los diferentes tipos de objetos y entidades, cómo se implementan en el código del programa y, finalmente, cómo se almacenan en el disco.

4.1. Objetos y entidades

Para cualquier tipo de juego existen una serie de objetos comunes que normalmente utilizamos.

- Objeto terreno o entorno: sólo existe una copia de este objeto y define la base de nuestro juego. Normalmente este objeto está compuesto por una única entidad grande (por ejemplo, la malla donde se posicionan el resto de objetos) y en él incluimos variables globales que influyen en el comportamiento global del sistema (por ejemplo, la gravedad).

- Objetos estáticos: objetos que se encuentran sobre el terreno con unas características (posición, tamaño, etc., prefijadas desde el principio). Incluyen una o varias entidades para poderlos representar en la pantalla. Pueden estar animados o no.

- Objetos móviles: se trata del principal grupo de objetos del juego. Aquí incluiremos aquellos objetos que se desplazan e interactúan con el entorno, tanto si están controlados por un jugador como si están controlados por la inteligencia artificial. Están compuestos por una o varias entidades, además de tener incorporados varias animaciones para cada una de las acciones que realizan. Estos objetos incorporan varios niveles de información para el sistema lógico que dependerá de su importancia. Por ejemplo, si se trata de una caja que podemos desplazar, tendremos propiedades físicas como su masa o la resistencia a romperse. En cambio, si es un personaje tendremos muchísimos más datos asociados a este objeto (vidas, armas, atributos, etc.).

Nota

Es una buena idea implementar los objetos móviles usando máquinas de estado finitas como los vistos en el punto 2.

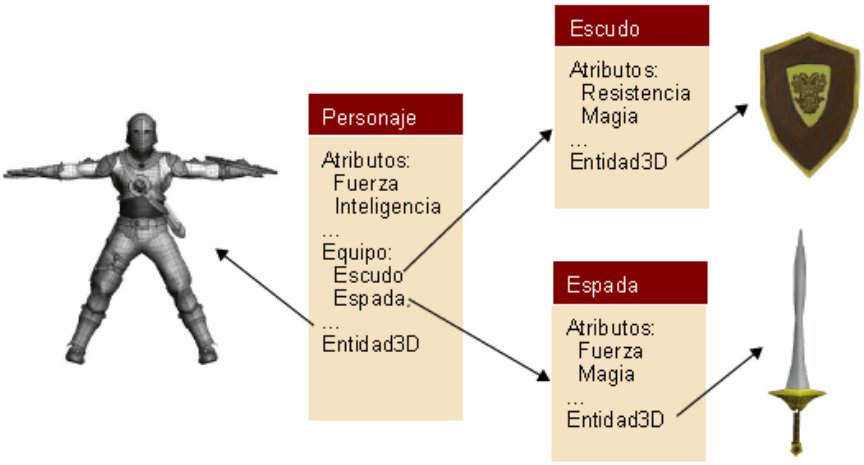
- Objetos de control: los objetos de control son objetos que no están relacionados con ninguna entidad y por tanto son invisibles al usuario. Se trata de objetos que son necesarios para que sistemas como la IA o el motor lógico del juego puedan conocer más detalles sobre el mundo donde se desarrolla la partida. Por ejemplo, en un juego de carreras podemos tener un objeto de control que nos indique el camino óptimo dentro de la pista para que la IA pueda calcular las posiciones de los coches.

Reflexión

Para poder almacenar y organizar todos los objetos y entidades que tenemos activos en nuestro juego, lo más fácil es utilizar las estructuras de datos que se utilizan en todo tipo de aplicaciones informáticas:

- Para representar secuencias de datos, podemos usar vectores, matrices, listas, pilas o colas.
- Para representar una jerarquía u organización, tenemos los árboles y los *heaps*.
- Para representar las relaciones entre objetos o entidades, podemos utilizar los grafos.

La gestión de las estructuras de datos las podemos programar nosotros mismos, pero quizás es recomendable el uso de alguna API externa que sea contratada. Si programamos usando C++, podemos utilizar la librería STL (*standard template library*), que incluye estructuras de datos genéricos que nos permiten gestionar todo tipo de elementos mediante el uso de *templates*.



Los objetos y las entidades normalmente se encuentran enlazados entre sí formando estructuras más complejas. Esto permite mucha más flexibilidad, ya que podemos combinar objetos y entidades para describir los elementos que participan en el juego.

4.1.1. Implementación de un objeto

La forma más simple de implementar los objetos es utilizando estructuras de datos. En una estructura de datos describimos todas las variables que están asociadas a un objeto, tanto las que se usan en la parte lógica como las que se usan para gráficos o sonido.

```
struct orco
{
   int posicion_x, posicion_y, posicion_z;
   int armadura, int arma;
   int vida_total, vida_restante;
   Entidad3D *malla;
   Onda *sonidos;
}
```

Como hemos visto en el primer apartado de este módulo, el problema de estas estructuras de datos es que son muy poco flexibles y seguras. Además, las funciones que utilizan estos datos se implementan de forma independiente a la estructura, con lo que la organización en módulos es más difícil de controlar.

Para facilitar el acceso a los datos de una forma más segura, la mejor opción es implementar una clase que los abstraiga y que nos proporcione una interfaz para poder acceder a ellos. De esta forma podremos controlar mejor el acceso a los datos. También tendremos juntos los datos y las funciones que trabajen directamente sobre ellos (como por ejemplo las funciones encargadas de actualizar la animación hacia el siguiente paso).

Por ejemplo, la clase Orco debe disponer de un constructor donde inicialicemos el estado de todas las variables, funciones para leer y escribir estas variables (ya sea de forma directa o indirecta), y funciones que permitan actualizar todo el sistema.

```
class Orco
{
public:
   Orco(const std::string& fichero_datos);
   ~Orco();

   void setPosicion_x(int position) { posicion_x = position; };
   int getPosicion_x() { return posicion_x };
   ...
```

```
    void update();      // Actualizar el objeto: animaciones...
private:
    int posicion_x, posicion_y, posicion_z;
    int armadura, int arma;
    int vida_total, vida_restante;
    Entidad3D *malla;
    Onda *sonidos;
};
```

Cada vez que queramos una copia de este objeto, simplemente necesitaremos crear una nueva instancia. Esta instancia cargará todos los datos que están especificados en el fichero que le pasamos como parámetro al constructor e inicializará todas las variables del objeto. Una vez tengamos la clase creada, ya podemos acceder a los datos.

Nota

Recordad que las funciones de estas clases deben estar muy optimizadas ya que se llamarán continuamente y desde diferentes puntos del juego (gráficos, sonido, IA,...).

Otra ventaja de la utilización de clases es que podemos utilizar la jerarquía de clases para representar la jerarquía de objetos que hemos comentado anteriormente y las relaciones entre los diferentes datos. Mediante el uso de interfaces y la herencia de clases, es mucho más fácil definir cómo acceder a los datos entre clases de una forma clara y segura.

Reflexión

Un error que se comete continuamente es la pérdida de memoria debido al mal uso de la memoria dinámica. Para minimizar el riesgo, es muy recomendable utilizar estructuras que abstraigan al programador de la tarea de reservar y liberar memoria dinámica. Por ejemplo, podemos utilizar un patrón Factory que no sólo cree las instancias de los objetos, sino que mantenga una lista de los objetos creados y que permita destruirlos.

En el siguiente ejemplo definimos la clase Orco, que contendrá el constructor y el destructor privados. Con esta condición sólo será posible crear un objeto de la clase Orco dentro de la propia clase o desde la clase ObjectManager, ya que indicamos que es una clase amiga:

Orco.h

```
#pragma once

#include <Vector4.h>

class ObjectManager;

class Orco
{
private:
    friend class ObjectManager;
    Orco();
    virtual ~Orco();

public:
```

```
    const Vector4& pos() const { return m_pos; }
    Vector4& pos() { return m_pos; }

    void update();

private:
    Vector4 m_pos;
};
```

Orco.cpp

```
#include <Orco.h>

#include <iostream>

Orco::Orco()
{
    std::cout << "I: Orco::Orco()" << std::endl;
}

Orco::~Orco()
{
    std::cout << "I: Orco::~Orco()" << std::endl;
}

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

Con el ejemplo anterior, el siguiente código nos dará errores de compilación:

error.cpp

```
#include <Orco.h>

int main()
{
    Orco *orco = new Orco(); // error, el constructor es privado
    delete orco;             // error, el destructor es privado
    Orco otroOrco;           // dos errores, tanto constructor como
    return 0;                // destructor son privados
}
```

A partir del código anterior, podemos crear una clase ObjectManager que permita crear y destruir objetos Orco:

ObjectManager.h

```
#pragma once

#include <Orco.h>

#include <set>

class ObjectManager
{
public:
   ObjectManager();
   virtual ~ObjectManager();

public:
   Orco *createOrco();
   void destroyOrco(Orco *orco);

private:
   typedef std::set<Orco*> OrcoSet;
   OrcoSet m_orcos;
};
```

Y su código:

ObjectManager.cpp

```
#include <ObjectManager.h>
#include <iostream>

ObjectManager::ObjectManager()
{
}

ObjectManager::~ObjectManager()
{
   for(OrcoSet::iterator it = m_orcos.begin(); it != m_orcos.end(); ++it)
   {
      std::cout << "W: Borrando orco desde el destructor del manager"
                << std::endl;
      delete *it;
   }
}

Orco *ObjectManager::createOrco()
{
   Orco *result = new Orco();
```

```
    m_orcos.insert(result);
    return result;
}

void ObjectManager::destroyOrco(Orco *orco)
{
    OrcoSet::iterator found = m_orcos.find(orco);
    if (found == m_orcos.end())
        std::cout << "E: No se puede borrar orco, no existe" << std::endl;
    else {
        delete *found;
        m_orcos.erase(found);
    }
}
```

Ahora, el siguiente código trabaja con orcos sabiendo que se han minimizado los fallos de memoria:

Ejemplo1.cpp

```
#include <ObjectManager.h>
#include <Orco.h>

int main()
{
    ObjectManager om;
    Orco *orco1 = om.createOrco();
    orco1->update();
    om.destroyOrco(orco1);

    return 0;
}
```

La salida que genera el código anterior:

```
I: Orco::Orco()
I: Orco::~Orco()
```

Mientras que el siguiente código:

Ejemplo2.cpp

```
#include <ObjectManager.h>
#include <Orco.h>

int main()
{
```

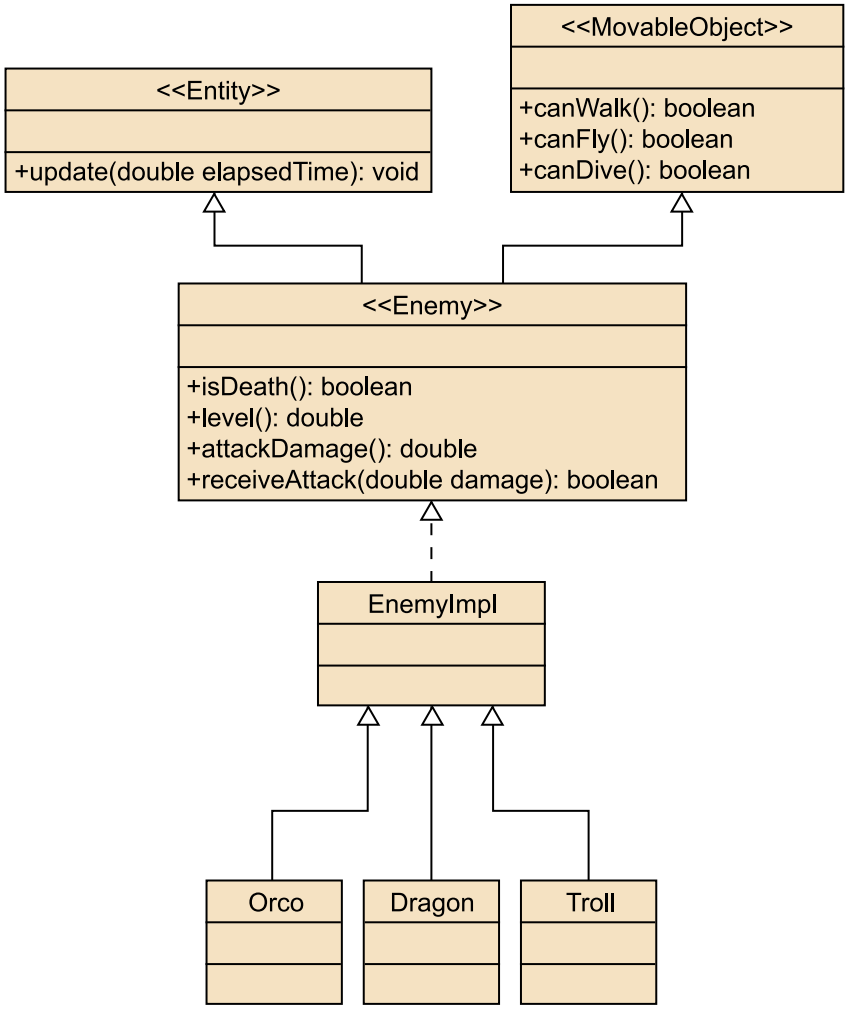
```
    ObjectManager om;
    Orco *orco1 = om.createOrco();
    Orco *orco2 = orco1;
    Orco *orco3 = om.createOrco();

    om.destroyOrco(orco1);
    om.destroyOrco(orco2);

    return 0;
}
```

Genera la siguiente salida:

```
I: Orco::Orco()
I: Orco::Orco()
I: Orco::~Orco()
E: No se puede borrar orco, no existe
W: Borrando orco desde el destructor del manager
I: Orco::~Orco()
```

4.1.2. Implementación de objetos usando herencia

En este punto definiremos a tres enemigos (Troll, Dragon y Orco) con características similares y configurables por interfaz. También definiremos a un jugador. Estos cuatro elementos son entidades que habitan en un mundo virtual. En el siguiente diagrama se detallan las interfaces, clases y sus relaciones que definen el ejemplo.



Para empezar, definiremos la interfaz entidad (se omite el código necesario para el dibujo):

Entity.h

```
#pragma once


class Entity
{
public:
   virtual ~Entity() { }
   virtual void update(double elpasedTime) = 0;
};
```

También definimos una interfaz que indique si una entidad puede volar, bucear o andar.

MovableObject.h

```
#pragma once
```

```
class MovableObject
{
public:
   virtual ~MovableObject() { }
   virtual bool canWalk() const = 0;
   virtual bool canFly() const = 0;
   virtual bool canDive() const = 0;
};
```

Se define un enemigo:

Enemy.h

```
#pragma once

#include <Entity.h>
#include <MovableObject.h>

class Enemy
   : public Entity
   , public MovableObject
{
public:
   virtual ~Enemy() { }

public:
   virtual bool isDeath() const = 0;
   virtual double level() const = 0;
   virtual double attackDamage() const = 0;
   // returns "true" if this attack kills himself
   virtual bool receiveAttack(double damage) = 0;
};
```

Se proporciona también una implementación básica de un enemigo en el siguiente código:

EnemyImpl.h

```
#pragma once

#include <Enemy.h>

class EnemyImpl
   : public Enemy
{
public:
   EnemyImpl();
```

```
   virtual ~EnemyImpl();

// Enemy partial implementation
public:
   bool isDeath() const { return m_health <= 0; }
   bool receiveAttack(double damage);

private:
   double m_health;
};
```

EntityImpl.cpp

```
#include <EnemyImpl.h>

EnemyImpl::EnemyImpl() { }
EnemyImpl::~EnemyImpl(){ }

bool EnemyImpl::receiveAttack(double damage)
{
   if (isDeath())
      return false;
   m_health -= damage;
   return isDeath();
}
```

Para definir el comportamiento concreto de cada enemigo:

Orco.h

```
#pragma once

#include <EnemyImpl.h>

class Orco
   : public EnemyImpl
{
public:
   Orco();
   virtual ~Orco();

// Entity implementation
public:
   void update(double elapsedTime);

// MovableObject implementation
public:
```

```
    bool canWalk() const { return true; }
    bool canFly() const { return false; }
    bool canDive() const { return false; }

// Enemy implementation
public:
    double level() const { return 10; }
    double attackDamage() const { return 0.2; }
};
```

Dragon.h

```
[...]
class Dragon
    : public EnemyImpl
{
[...]
// MovableObject implementation
public:
    bool canWalk() const { return false; }
    bool canFly() const { return true; }
    bool canDive() const { return false; }

// Enemy implementation
public:
    double level() const { return 30; }
    double attackDamage() const { return 0.4; }
};
```

Troll.h

```
[...]
class Troll
    : public EnemyImpl
{
[...]

// MovableObject implementation
public:
    bool canWalk() const { return true; }
    bool canFly() const { return false; }
    bool canDive() const { return true; }

// Enemy implementation
public:
    double level() const { return 20; }
    double attackDamage() const { return 0.25; }
```

```
};
```

Por último, la declaración de un jugador:

Player.h

```
#pragma once

#include <Entity.h>
#include <MovableObject.h>

class Player
   : public Entity
   , public MovableObject
{
public:
   Player();
   virtual ~Player();

// Entity implementation
public:
   void update(double elapsedTime);

// MovableObject implementation
public:
   bool canWalk() const { return true; }
   bool canFly() const { return false; }
   bool canDive() const { return true; }
}
```

El controlador de objetos:

ObjectManager.h

```
#pragma once

#include <Enemy.h>
#include <Player.h>

#include <set>

class ObjectManager
{
public:
   ObjectManager();
   virtual ~ObjectManager();
```

```
public:
   template<class T> Enemy *createEnemy();
   void destroyEnemy(Enemy *enemy);

   Player *createPlayer();
   void destroyPlayer(Player *player);

public:
   void update(double elapsedTime);

private:
   typedef std::set<Enemy*> EnemySet;
   typedef std::set<Entity*> EntitySet;
   EnemySet m_enemies;
   EntitySet m_entities;
};

template<class T> Enemy *ObjectManager::createEnemy()
{
   Enemy *enemy = new T();
   m_enemies.insert(enemy);
   m_entities.insert(enemy);
   return enemy;
}
```

ObjectManager.cpp

```
#include <ObjectManager.h>

ObjectManager::ObjectManager()
{
}

ObjectManager::~ObjectManager()
{
}

void ObjectManager::destroyEnemy(Enemy *enemy)
{
   m_enemies.erase(enemy);
   m_entities.erase(enemy);
}

Player *ObjectManager::createPlayer()
{
   Player *player = new Player();
   m_entities.insert(player);
```

```
    return player;
}

void ObjectManager::destroyPlayer(Player *player)
{
    m_entities.erase(player);
    delete player;
}

void ObjectManager::update(double elapsedTime)
{
    for(EntitySet::iterator it = m_entities.begin(); it != m_entities.end(); ++it) {
        Entity* curr = *it;
        curr->update(elapsedTime);
    }
}
```

Y un código de ejemplo que lo utiliza:

main.cpp

```
#include <ObjectManager.h>

#include <Dragon.h>
#include <Orco.h>
#include <Troll.h>
#include <Player.h>

#include <iostream>

int main()
{
    ObjectManager om;

    Enemy *orco1 = om.createEnemy<Orco>();
    Enemy *orco2 = om.createEnemy<Orco>();
    Enemy *troll = om.createEnemy<Troll>();
    Enemy *dragon = om.createEnemy<Dragon>();
    Player *player = om.createPlayer();

    std::cout << "First update:" << std::endl;
    om.update(1);

    om.destroyEnemy(orco1);
    om.destroyEnemy(orco2);
    om.destroyEnemy(troll);
    om.destroyEnemy(dragon);
```

```
    std::cout << "Second update:" << std::endl;
    om.update(1);

    om.destroyPlayer(player);

    std::cout << "Last update:" << std::endl;
    om.update(1);

    return 0;
}
```

La ejecución del ejemplo genera la siguiente salida por la consola:

```
First update:
I: Orco::update()
I: Orco::update()
I: Troll::update()
I: Dragon::update()
I: Player::update()
Second update:
I: Player::update()
Last update:
```

4.2. Niveles del juego

Se considera como un nivel de un juego una sección o una parte del mismo. Normalmente, cada nivel tiene unos objetivos particulares, los cuales se tienen que llevar a cabo antes de poder pasar al siguiente.

Técnica de *prefetching*

Para dar al usuario la sensación de continuidad, en algunos juegos se utiliza la técnica de *prefetching*, que carga por anticipado los datos del siguiente nivel sin que el usuario se de cuenta.

Un nivel es simplemente un conjunto de datos que recopila toda la información de todos los elementos que intervienen en esta sección del juego:

- El terreno base para el nivel y todas las propiedades asociadas.
- Los objetos estáticos y dinámicos. Su colocación en la escena, sus atributos, sus propiedades lógicas.
- Los componentes adicionales para los gráficos (situación de las luces, cámaras, etc.).
- Los objetos de control y las reglas que determinan el funcionamiento global del nivel.

Los datos de cada nivel acostumbran a ser diferentes, tanto a nivel lógico como gráfico o sonoro. Por tanto, la segmentación de un juego en niveles nos permite organizar mucho mejor los datos y no tener que tratar con todo el volumen a la vez.

4.2.1. Implementando un nivel

Existen dos formas de definir un nivel dentro de nuestro juego, introduciéndolo todo como código fuente o combinando parte de código fuente con los datos de un fichero externo. La primera opción es sólo factible en un juego pequeño y simple donde no existan muchos objetos ni necesitemos realizar un balanceo final de los mismos (por ejemplo, en un juego de ajedrez probablemente podremos organizar todos los objetos dentro del propio código).

Normalmente, la opción más utilizada para implementar un nivel se realiza separando los datos del código principal. Tan sólo necesitaremos programar una clase que nos permita leer los datos externos necesarios para un determinado nivel y distribuirlos por la escena para poder empezar a jugar.

```
#ifndef __Nivel__
#define __Nivel__

#include <string>

class Nivel
{
public:
   Nivel(const std:string& name, const std::string& filename);
   ~Nivel();

public:
   void cargar();
   void descargar();

private:
   const std::string m_name;
   const std::string m_filename;

   Terrain* m_terreno;
   Objeto* m_listaObjetos;
   ObjetoMovil* m_listaObjetosMoviles;
};

#endif // __Nivel__
```

Y posteriormente necesitaremos un controlador que nos permita gestionar los diferentes niveles y acceder al nivel actual.

```
#ifndef __NivelManager__
#define __NivelManager__

class NivelManager
```

```
{
public:
   static NivelManager* getSingletonPtr();
   static NivelManager& getSingleton();

private:
   NivelManager();
   ~ NivelManager();

public:
   void seleccionarNivel(const std::string& name);
   void cargarNivelSeleccionado();
   void descargarNivelSeleccionado();

   Nivel& nivelActual() { assert(m_currentTrack); return *m_currentTrack; }
private:
   std::map<std::string, Nivel*> m_tracks;
   std::string m_selectedTrackName;
   Nivel* m_currentTrack;
};

#endif // __NivelManager__
```

4.3. Almacenamiento de los datos

Como acabamos de comentar, es importante separar los datos del código principal. La mejor forma es guardarlos en un fichero aparte y leerlos durante la ejecución del código cuando sea necesario.

Nota

También existe la posibilidad de guardar los datos en una base de datos. En este caso se requiere de una interfaz que nos permita consultarlos y guardarlos en nuestras clases objeto.

Los datos los almacenamos en diferentes tipos de fichero dependiendo de su contenido. Los datos binarios (imágenes, sonidos,...) necesitan de un formato propio que nos permita codificarlos de forma óptima. Los datos no binarios (normalmente descripciones en forma de texto) podemos guardarlos de diferentes formas dependiendo del nivel de estructura que queramos incluir en el fichero.

4.3.1. Datos binarios

Para nuestro juego tendremos que escoger uno (o varios) tipos de formatos con los que vamos a guardar la información binaria.

Los datos binarios los guardamos en el disco utilizando un formato que optimice su relación calidad/tamaño. Podemos utilizar formatos propios si necesitamos guardar algún tipo de característica especial, pero lo más normal es utilizar formatos conocidos y que existan librerías disponibles para poderlos leer y escribir. Algunos ejemplos de formatos:

- Sonido: WAV, MP3, Midi, OGG,...
- Imágenes rasterizadas: JPEG, BMP, GIF, PNG, TIFF,...
- Imágenes vectoriales: EPS, SVG,...
- Vídeo: MOV, MPEG, OGG, FLA,...
- Objetos 3D: 3DS, MESH, MAX, SMD,...

Lo más normal es tener un fichero para cada recurso que pongamos en nuestro sistema. Para poder organizar toda esta información, una buena opción es agrupar todos los ficheros que se utilicen dentro de un mismo nivel (o incluso todos los recursos del juego) en un solo fichero. Adicionalmente, podemos utilizar algún tipo de compresión en este fichero para reducir aún más el tamaño de los recursos.

Comprimir los datos

Si queremos utilizar la opción de comprimir los datos en un fichero único, necesitaremos también alguna librería que nos permita realizar la descompresión desde dentro del juego

Un ejemplo de agrupación de ficheros de recursos en los videojuegos los encontramos en el formato .PAK, utilizado en juegos como Quake o Half-life. En estos ficheros encontramos comprimidos imágenes, sonidos, objetos y el resto de datos.

Reflexión

Otro aspecto que habrá que tener en cuenta cuando guardemos los datos en el disco es saber cómo se van a leer por parte de la versión final del juego.

Si tenemos los datos disponibles en algún soporte de lectura aleatoria (como un disco duro), lo mejor es tener la organización en paquetes o directorios. Al existir menos limitaciones de espacio podemos utilizar recursos de mayor calidad.

En cambio, si tenemos los datos en un soporte óptico (CD o DVD), es importante la organización de los mismos. En este caso hay que decidir en qué parte del disco colocamos los datos. Lo más normal es poner los datos que se leen más cerca del centro (donde se leen más deprisa) y los que se utilizan mucho menos, en la parte externa del disco.

4.3.2. Datos no binarios

Aunque la solución más simple sería guardar todos los datos de forma secuencial en un fichero, lo mejor es añadir un poco de información extra para organizarlos. De este modo los datos guardados estarán bien estructurados y serán más fáciles de leer y de tratar.

Para realizar esta tarea de catalogación de la información, podemos usar un lenguaje descriptivo o de marcado. Un lenguaje descriptivo incorpora etiquetas o marcas que contienen información adicional acerca de la estructura del texto o su presentación.

Con los lenguajes descriptivos podemos almacenar tanto los niveles como los objetos que hemos descrito en el apartado anterior:

- De los objetos podemos guardar sus entidades y cómo se estructuran jerárquicamente sus sonidos y algunas variables propias del objeto que no cambian de un objeto a otro (o poner algunos valores por defecto).

- De los niveles podemos guardar todos los objetos que intervienen en un determinado nivel, sus posiciones, atributos, la lógica del sistema... En este caso es mucho más importante la estructuración del fichero, ya que el volumen de datos probablemente será mucho mayor.

El formato XML

Siempre que se ha creado un programa, se ha generado una serie de información que ha convenido guardar. Pasando por alto partes de la historia, esta información se guardaba en ficheros y, normalmente, en un formato que definía el programador.

De este modo, cada software generaba un formato de fichero que sólo el programador conocía.

Por éste y otros motivos, IBM encargó a Charles F. Goldtab una estructura que permitiese guardar cualquier tipo de información. Charles creó GML (*generalized markup language*), pero fue demasiado extenso y no tuvo éxito.

En 1986, la ISO adoptó el trabajo de Goldtab y creó el estándar SGML (*standard generalized markup language*). Aunque fue un estándar, la gente continuó trabajando con formatos propietarios (WordStar, dBase, WordPerfect...).

El SGML se basa en una serie de elementos que definen el inicio y el final de una información. Esta información se guarda en un fichero de texto, por ejemplo:

Ejemplo.sgml

```
<-- comentario -->
<libro>
  <titulo>Derecho de la función pública</titulo>
  <autor>Miguel Sanchez Moron</autor>
  <capitulo>
    <titulo>La función pública y su evolución histórica</titulo>
    <seccion>
      <titulo>Qué es la función pública</titulo>
      <parrafo>El vasto complejo organizativo que hoy componen las...</parrafo>
      <parrafo>Algunas de estas personas han sido elegidas...</parrafo>
    </seccion>
  </capitulo>
</libro>
```

Para definir la estructura correcta de un fichero SGML, se utiliza la especificación DTD (*document type definition*):

libro.dtd

```
<!ELEMENT libro - - (titulo?, autor?, capitulo+)>
<!ELEMENT titulo - - (#PCDATA)>
<!ELEMENT autor - - (#PCDATA)>
<!ELEMENT capitulo - - (titulo?, seccion+)>
<!ELEMENT seccion - - (titulo?, parrafo+)>
<!ELEMENT parrafo – 0 (#PCDATA)>
```

En el ejemplo anterior, la marca de finalización de párrafo no sería necesario ponerla.

La especificación de SGML es demasiado extensa y compleja, así que la W3C y varias empresas definieron un nuevo lenguaje de marcas basado en SGML, pero más sencillo, al que llamaron XML: eXtended Markup Language.

Un XML tiene como limitaciones principales sobre el SGML:

- una marca debe tener su correspondiente de cierre.
- una marca debe de estar incluida completamente en otra, de esta forma se genera un árbol donde se tiene un nodo inicial al que se llama "root".

El ejemplo de SGML anterior también sería un fichero XML válido.

El lenguaje XML es una tecnología sencilla, pero su punto fuerte está en que proporciona un sistema de organización de la información que puede ser fácilmente interpretado por cualquier aplicación. Se compone de dos partes, por un lado un fichero DTD (*document type definition*), que especifica la estructura que pueden tener las etiquetas y por tanto el fichero; y por otro lado, los ficheros XML que utilizan estas etiquetas para almacenar y describir la información. Asimismo, existen infinidad de librerías que nos proporcionan la lectura y escritura de ficheros usando XML.

Un ejemplo de fichero en formato XML que define un objeto de nuestro juego puede ser el siguiente:

Datos_orco1.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<objeto>
  <nombre>orco</nombre>
<variable>
  <nombre>vida_total</nombre>
  <tipo>int</tipo>
  <valor>30</valor>
</variable>
<variable>
  <nombre>arma</nombre>
  <tipo>int</tipo>
  <valor>10</valor>
</variable>
<variable>
  <nombre>armadura</nombre>
  <tipo>int</tipo>
  <valor>8</valor>
</variable>
<objeto3d>orco.3d</objeto3d>
<sonidos>orco.wav</sonidos>
</objeto>
```

Los programas encargados de leer e interpretar la entrada se conocen como analizadores sintácticos (en inglés, *parser*). Estos programas permiten navegar por el contenido del fichero como si estuviéramos explorando una estructura

en forma de árbol. Normalmente el *parser* se llama dentro de los constructores de los objetos, los cuales reciben como parámetro el fichero donde están contenidos los datos del objeto que queremos crear.

4.4. Editor de objetos y de niveles

En el caso de que queramos trabajar con los objetos y los niveles de forma independiente del código del programa, necesitaremos una aplicación para poder tratar con todos estos datos.

Nota

A un editor de niveles también se lo conoce por un editor de campañas, mapas o escenarios.

La solución más elegante es desarrollar lo que se conoce por un editor de niveles. Aunque algunos juegos lo llevan incorporado dentro del propio programa, lo más normal es que se trate de un programa completamente independiente.

En un editor de niveles debemos poder organizar todos los elementos que compone un nivel, añadir nuevos o quitarlos. Adicionalmente, algunos editores permiten editar los objetos que tenemos definidos dentro del nivel.

La comunicación entre el editor de niveles y el juego se realiza a través de los ficheros de datos. En este caso es altamente recomendable el uso de los lenguajes descriptivos explicados anteriormente para garantizar que ambos sistemas comparten la información sin problemas.

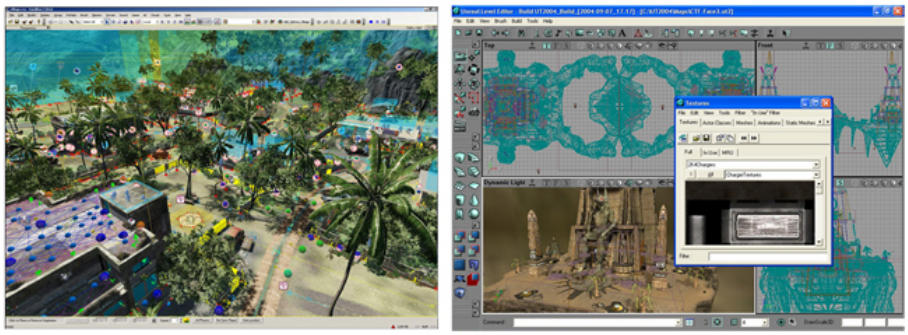


Figura 5. Sandbox Editor del juego Crysis y UnrealEd del juego Unreal. Unreal Tournament © Electronic Arts y Epic Games respectivamente

5. Lenguajes del *scripting*

El problema principal en el desarrollo de un juego es que existe una gran cantidad de perfiles diferenciados que contribuyen a la creación del mismo, y muchos de estos perfiles no tienen los conocimientos de programación necesarios que les permitirían escribir clases y funciones. En este caso, la única opción es dar las explicaciones de lo que tiene que hacer el juego a los programadores y esperar que ellos traduzcan sus ideas en forma de código dentro del programa.

Este problema lo podemos resolver con el uso de los lenguajes de *scripting*. La principal característica de los lenguajes de *scripting* es que nos permiten separar el proceso de creación de un juego de los motores y API utilizadas para interactuar con el sistema. Esto permite que todos aquellos que no dominen la programación puedan participar en los procesos más importantes de la creación de un juego y su posterior balanceo.

A diferencia de los lenguajes que utilizamos para programar los diferentes motores gráficos, lógicos y los otros componentes (en nuestro caso hemos trabajado con el lenguaje C++), el *scripting* se realiza mediante un lenguaje interpretado. Se conoce como un lenguaje interpretado (o de *script*) aquel que se interpreta en tiempo de ejecución y no es necesario compilarlo previamente.

La inclusión de un lenguaje de *scripting* facilita sobre todo lo que conocemos por la parametrización del juego, es decir, la definición de todos los elementos que intervienen, todos sus atributos y todas las reglas que se aplican en el juego. Veremos más adelante que esta parametrización es clave para poder crear fácilmente nuevos componentes y sobre todo para poder realizar una de las tareas más críticas de todo el proceso: el balanceo final.

5.1. Diferencia entre un lenguaje interpretado y un lenguaje compilado

Vamos a poner un ejemplo que nos permitirá ver más claramente la diferencia existente entre un código compilado y uno que utilice un *script*.

Diferencia entre *script* y código

En un nivel conceptual, se dice que un *script* es de alto nivel porque dice al juego lo que tiene que hacer, mientras que un código es de bajo nivel porque dice al juego cómo lo tiene que hacer.

En un lenguaje compilado ponemos todo lo que va a realizar nuestro programa dentro del código fuente y generamos un fichero ejecutable binario compilándolo. Cuando queremos ver el resultado, tan sólo tenemos que lanzar el ejecutable y el sistema interpreta el binario para darnos el resultado. Cada vez que queremos realizar un cambio, tenemos que volver a repetir el proceso: editar, compilar y después ejecutar.

Calculadora.c

```
#include <stdio.h>
int main()
{
   int a = 5;
   int b = 10;
   int c = a * b;
   printf("%d x %d = %d\n", a, b, c);
   return 0;
}
```

En un lenguaje interpretado el funcionamiento es diferente. Nosotros escribimos un código utilizando la sintaxis del lenguaje de *scripting* y guardamos el fichero fuente. Cuando queremos ejecutar el código, lo que hacemos es ejecutar lo que se conoce por el intérprete y le pasamos nuestro código como parámetro. A medida que el intérprete va leyendo el código, va ejecutando las diferentes acciones. Si queremos cambiar algo, únicamente tenemos que editar el fichero fuente y volver a lanzar el intérprete, sin necesidad de compilar nada. Hasta es posible tener el intérprete en ejecución e indicarle que vuelva a cargar el fichero de *script* y lo ejecute.

En este caso vamos a enseñaros un ejemplo hecho en Python, uno de los lenguajes descriptivos más utilizados. Fijaos además en el siguiente ejemplo donde la sintaxis es mucho más simple e intuitiva que en el ejemplo anterior:

Calculadora.py

```
a = 5
b = 10
c = a * b
print a,"x",b,"=", c
```

Finalmente, la tercera opción que tenemos es la de integrar un código compilado con uno interpretado. El secreto reside en añadir dentro del código que compilamos los mecanismos necesarios para llamar al intérprete externo, normalmente mediante la utilización de una librería externa:

Calculadora.c

```
#include <stdio.h>
#include <Python.h>

int main()
{
   FILE *fp;
   // Inicializar integración con intérprete
   Py_Initialize();
   // Interpretar fichero externo
```

```
    PyRun_SimpleFile(fp, "calculadora.py");
return 0;
}
```

Si escogemos esta tercera opción, podemos ejecutar todo el código que se encuentra dentro del *script*, como hemos mostrado en el ejemplo anterior; o bien podemos llamar sólo una función en particular que se encuentre dentro del *script*. En este segundo caso se tiene que ir con mucho cuidado de que la información que se transfiera en forma de parámetros entre los dos códigos sea coherente.

5.2. Ventajas y desventajas del *scripting*

Vamos a analizar las principales ventajas y desventajas que conlleva integrar un lenguaje de *scripting* dentro de nuestro videojuego. Empezaremos detallando algunas de las principales ventajas:

- El punto más importante de los lenguajes de *scripting* es su facilidad de uso. Tal y como veremos posteriormente, los buenos lenguajes tienen una estructura muy intuitiva, lo que nos permite hacer códigos que se pueden entender con tan sólo leerlos.

- Hacen que el juego se encuentre más orientado a los datos y a las reglas, y que los motores sean un algoritmo simple que trabaja con todos estos datos y reglas.

- Permiten hacer pruebas muy rápidamente, lo que se conoce por "in-game tweaking", ya que no hace falta recompilar el código que cambiemos. Sólo realizando unas pequeñas modificaciones podemos ver el resultado obtenido de forma inmediata (por ejemplo, pulsando una combinación de teclas dentro del juego, sin necesidad de pararlo y volverlo a abrir).

- Permiten que los usuarios puedan desarrollar *mods* (o extensiones) que pueden extender, mejorar o modificar completamente el juego. Esto permite que alrededor del juego se cree un entorno social muy importante.

- Las actualizaciones y los parches para arreglar ciertos problemas normalmente son más fáciles de distribuir, ya que muchas veces no se requiere que se modifiquen los ejecutables principales.

- Existe una gran variedad de lenguajes de *scripting* públicos con una extensa documentación, algunos de ellos se han utilizado con gran éxito en algunos de los juegos más vendidos de la historia.

Y ahora vamos a detallar algunas de las desventajas asociadas con los lenguajes de *scripting*.

- Quizás el principal punto débil de estos lenguajes se encuentra en su rendimiento. El código se lee en tiempo de ejecución por parte del intérprete y éste lo traduce a órdenes, pero este proceso es siempre más lento que si estuviera compilado (normalmente un compilador optimiza el código para que se ejecute más rápido). Como no vamos a programar ninguna tarea crítica usando *scripting* (por ejemplo los gráficos), este aspecto no debería preocuparnos demasiado.

- La resolución de los problemas es más compleja que en un lenguaje compilado. En primer lugar, en el lenguaje compilado nos damos cuenta de los errores cuando compilamos (antes de empezar a ejecutar la aplicación). En cambio, en los lenguajes de *scripting* sólo nos damos cuenta de que hay un error cuando llegamos a la interpretación del mismo. También las herramientas de depuración para detectar los errores son mucho menos detalladas que las herramientas para lenguajes compilados.

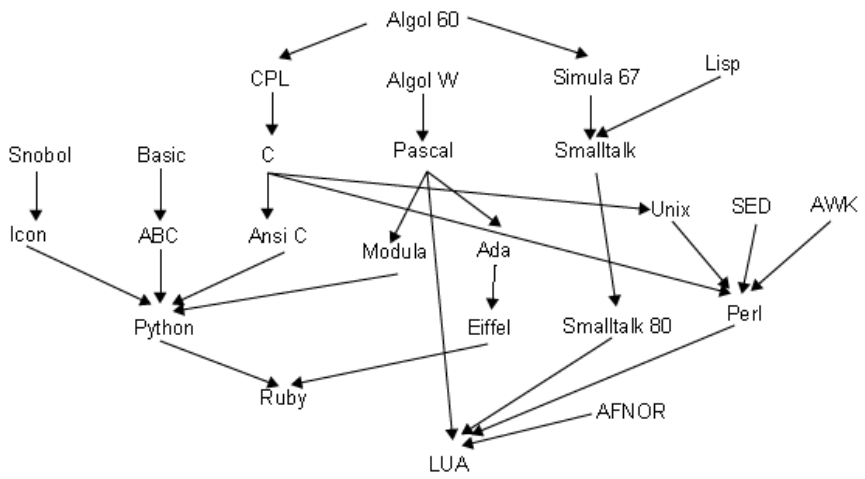
- Estos lenguajes no están preparados para tratar con códigos muy largos ni muy complejos. Si tenemos alguna función con muchas líneas, quizás deberíamos pensar en integrarla dentro del código compilado. La discusión de si debe estar dentro (en el lenguaje compilado) o fuera (en el interpretado) continúa viva.

5.3. Lenguajes de *scripting*

Existen dos posibilidades a la hora de integrar un lenguaje de *scripting* en nuestra aplicación: desarrollar nuestro propio lenguaje o utilizar alguno público. Como siempre, ambas opciones tienen sus ventajas y desventajas.

En el caso de que queramos desarrollar nuestro propio lenguaje, debemos tener en cuenta que la tarea es compleja porque tendremos que programar el intérprete del lenguaje. En este caso, recomendamos repasar la literatura acerca de la programación de compiladores de código, donde se explica cómo transformar una secuencia de comandos de un texto en una serie de órdenes del procesador. Las ventajas de realizar esta tarea son numerosas, ya que podemos adaptar perfectamente el lenguaje a las necesidades específicas de nuestro juego.

Por otro lado, existe una cantidad importante de lenguajes interpretados que se pueden utilizar como lenguajes de *scripting* en videojuegos: LUA, Python, Ruby, JavaScript, Perl,... Estos lenguajes han evolucionado a partir de la sintaxis de otros lenguajes, simplificando el trabajo del programador para poder realizar códigos más simples e intuitivos.



5.3.1. Introducción a LUA

Para explicar con más detalle la integración de un lenguaje de *scripting* con el motor lógico del juego, hemos decidido utilizar LUA. Se trata de uno de los lenguajes más utilizados en la programación de videojuegos y, a diferencia de los otros, tan sólo puede utilizarse integrado dentro de otro lenguaje (la mayoría de los lenguajes interpretados vienen acompañados de un interprete externo para desarrollar programas completos).

Nota

Existe la posibilidad de compilar los *scripts* con LUA para hacerlos un poco más eficientes, obteniendo ficheros binarios con formato LUB.

LUA es un lenguaje interpretado creado por unos profesores de la universidad de Río de Janeiro. Se ha utilizado en multitud de aplicaciones y juegos, como por ejemplo Monkey Island 3-4, World of Warcraft, Homeworld 2 o FarCry. Además, es compatible con muchas videoconsolas, como por ejemplo para PS2 o PSP. En la página del propio lenguaje destacan los siguientes aspectos:

- Fácil de utilizar para no programadores
- Rápido, no ralentiza el juego
- El tamaño de los *scripts* ejecutables normalmente es bastante pequeño
- Dispone de mucha documentación y ejemplos prácticos
- Proporciona una interfaz limpia y flexible

La sintaxis de un código en LUA es bastante simple y muy parecida a la utilizada en C. Existen multitud de páginas disponibles en Internet con tutoriales y referencias de las funciones del lenguaje.

Lectura recomendada

Recomendamos visitar la guía de referencia oficial del lenguaje disponible en:

http://www.lua.org/manual/5.1/es

Ejemplo. LUA

```
function factorial(n)
   if n == 0 then
      return 1
   end
   return n * factorial(n - 1)
end
```

```
-- Comentario
print "Hello, World!"
print factorial(10)
```

5.3.2. Integración de LUA con C

Vamos a estudiar cómo se integra LUA con una aplicación desarrollada en C.

En primer lugar tenemos que aprender cómo se puede llamar un código en LUA desde dentro de un código en C. Para eso tenemos que hacer dos cosas:

- Crear una referencia a un nuevo intérprete mediante la función lua_open(int).
- Incluir todas aquellas funciones que vamos a utilizar en nuestro código.

Nota

A partir de la versión 5.2., la función para crear un nuevo intérprete (estado) es *luaL_newstate.* Por otro lado, la inclusión de las librerías más comunes se puede hacer con *luaL_openlibs*."

initLUA.c

```
#include <stdio.h>
#include <lua.h>

int main(int argc, char* argv[])
{
   // Inicializamos el intérprete
   lua_State* luaVM = lua_open(0);
   if (NULL == luaVM)
   {
      printf("Error Initializing lua\n");
      return -1;
   }
   // Inicializamos las librerías que necesitemos de LUA
   lua_baselibopen(luaVM);
   lua_iolibopen(luaVM);
   lua_strlibopen(luaVM);
   lua_mathlibopen(luaVM);

   // Aquí pondremos lo que queramos hacer con LUA

   // Cerramos el interprete
   lua_close(luaVM);
   return 0;
}
```

Una vez tenemos el intérprete inicializado, ya podemos empezar a ejecutar *scripts* escritos con LUA. Para leer un *script* contenido en un fichero utilizamos la función lua_dofile:

```
lua_dofile(luaVM , "script.lua");
```

Si lo que nos interesa es hacer llamadas desde un *script* de LUA a funciones de C con parámetros, hemos de utilizar una estructura de datos de tipo pila (o *stack*) que proporciona el intérprete de LUA y que nos permitirá la comunicación entre lenguajes.

En primer lugar tenemos que registrar aquellas funciones de C que vamos a llamar desde LUA:

```
lua_register(luaVM , "nombre de la funcion", mifuncion);
```

Los parámetros utilizados en las funciones se encuentran dentro de la pila que hemos comentado, así que tenemos que acceder a ella para leer los parámetros de entrada utilizados al llamar la función y escribir los resultados de salida:

```
int mifuncion(lua_State *L)
{
   // Miramos cuantos parametros hay dentro
   int argc = lua_gettop(L);

   for ( int n=1; n<=argc; ++n )
   {
      printf( " %s \n", lua_tostring(L, n));
   }

   // Escribimos un resultado en la pila
   lua_pushnumber(L, 123);
   // Devolvemos cuantos resultados hay en la pila
   return 1;
}
```

Esta es la forma más simple de integrar ambos lenguajes. En el siguiente punto veremos cómo utilizar la librería *luabind* para realizar llamadas más complejas.

5.3.3. Integración de LUA con C++ usando *luabind*

La API que proporciona la librería original de LUA es demasiado simple para poder aprovechar al máximo el lenguaje y, a la vez, minimizar el tiempo de desarrollo. Para ello, existen varias librerías que facilitan la interacción entre C++ y LUA. Una de ellas es *luabind* y será la que vamos a explicar brevemente en este punto.

Los puntos básicos que se quieren solventar con este tipo de librerías son:

- Traducción automática de los prototipos de funciones de C a la estructura de LUA. Así nos "podemos despreocupar" de controlar cómo se pasan los parámetros y los resultados entre el código en C y el código en LUA.

- Proporcionar clases y objetos de C++ para ser utilizados en los scripts LUA. De esta forma podemos acceder desde LUA a todos los objetos y recursos que ya se encuentran creados dentro del juego.

La librería basa toda su simpleza en el uso de *templates* de C++ y la sobrecarga de operadores ([], +) para hacer más legible al programador lo que se está definiendo. Por este motivo, es necesario que el alumno entienda estos conceptos para poder resolver los posibles errores de compilación que puedan aparecer.

Lo primero que se tiene que hacer es inicializar la librería *luabind* usando la función *luabind::open* después de haber inicializado el intérprete de LUA.

A partir de entonces, se pueden crear tantos módulos como se quiera usando el *template luabind::module*. Un ejemplo, y fijaos que, a diferencia del código anterior, en las funciones ya no utilizamos la pila para pasar los parámetros:

```
using namespace luabind;

static double suma(double x, double y) {
   return x + y;
}

static double cuadrado(double x) {
   return x * x;
}

int main() {
   lua_State *L = luaL_newstate();
   luaL_openlibs(L);
   open(L);
   module(L) [
      def("suma", &suma),
      def("cuadrado", &cuadrado)
   ];
   ...
}
```

Y en LUA podríamos usar estas funciones:

```
r = cuadrado(suma(3,4))
```

La librería también nos permite definir y usar clases de C++ de una manera muy simple. Suponiendo que tenemos una clase de C++ llamada "Car" donde están definidos los métodos: accelerate, brake, turnLeft, turnRight, celerity y wheelDegrees, la definimos:

```
module(L) [
   class_<Car>("Car")
      .def(constructor<>())
      .def(constructor<int>())
      .def("accelerate", &Car::accelerate)
      .def("brake", &Car::brake)
      .def("turnLeft", &Car::turnLeft)
      .def("turnRight", &Car::turnRight)
      .property("celerity", &Car::celerity)
      .property("wheelDegrees", &Car::wheelDegrees),
   def("suma", &suma),
   def("cuadrado", &cuadrado)
];
```

Y el código LUA que utiliza esta clase:

```
c = Car()
c:accelerate()
print(c.celerity)
```

Este ejemplo es válido para crear objetos en LUA, pero lo interesante del scripting es poder acceder a los objetos que ya han sido creados por el juego (siguiendo el ejemplo, los coches que realmente están en la carrera), y no crear objetos nuevos.

Para ello, una solución simple es crear una función en C++ que acceda a un objeto singleton que contenga la información de la carrera y sea este objeto el que se proporcione con los scripts de LUA. Por ejemplo:

```
class Race {
public:
   static Race &instance();
   Car &car(int id);
   ...
};
```

Se define en el módulo para poder ser llamada desde LUA:

```
module(L) [
   def("curRace", &Race::instance),
   class_<Race>("Race")
      .def("car", &Race::car)
```

```
];
```

Y ya se puede acceder a los objetos del juego actual:

```
r = curRace()
c = r:car(3)
c:accelerate() -- accede al objeto de C++
```

5.4. Usos del *scripting*

Una vez visto que el *scripting* nos proporciona una gran herramienta para hacer el sistema mucho más modular, vamos a ver algunas partes de un videojuego que se acostumbra a utilizar con *scripting*.

5.4.1. Almacenamiento de datos

Aunque ésta no es su tarea principal, podemos definir los parámetros de un elemento mediante variables del *scripting*. En este caso solamente usaríamos la parte estática del *scripting*.

Nos puede servir como sustituto de los lenguajes descriptivos vistos anteriormente.

Nivel.lua

```
crearTerreno(desierto)
addNPC(orco1, pos1, arma1, armadura1)
addNPC(orco2, pos2, arma1, armadura1)
addNPC(orco3, pos3, arma2, armadura2)
addNPC(orco4, pos4, arma2, armadura2)
colocarLuz(posLuz)
colocarCamara(posCamara)
colocarPersonaje(posPersonaje)
```

En este caso suponemos que todas las funciones que llamamos desde este *script* de ejemplo deberán estar definidas dentro del código principal del programa, tal como hemos explicado anteriormente.

5.4.2. Grafos de estado de la IA y lógica del juego

Una de las funciones principales de LUA es la de proporcionar un sistema que permite diseñar nuestra IA del juego a medida que la vamos probando en tiempo real. También el equipo que se encarga de la lógica del juego acostumbra a realizar gran parte de su trabajo mediante lenguajes de *scripting*.

iaCoche.lua

```
function nextMovement(c)
   if c.wheelDegrees == 0 then
      if c.remainDistance > 10 then
         c:accelerate()
      else
         c:brake()
      end
   elseif c.nextCurveToRight then
      c:turnRight()
   elseif c.nextCurveToLeft then
      c:turnLeft()
   end
end

r = curRace()
for i = 0, r.carCount - 1 do
   local c = r:car(i)
   nextMovement(c)
end
r:run()
```

5.4.3. Definición y control de interfaces de usuario

El uso de un lenguaje de *scripting* hace que la interfaz de usuario sea muy flexible y adaptable a las necesidades de cada usuario.

Mediante el uso de un lenguaje de *scripting*, podemos ofrecer al usuario más o menos cantidad de información según las necesidades del mismo. El secreto reside en proporcionar una plataforma accesible desde LUA para poder acceder a todos los atributos de los objetos del juego. Después, mediante el uso de LUA, podemos leer esta información, tratarla convenientemente y organizarla para que el usuario la pueda disfrutar en su pantalla.

El desarrollo de extensiones de la interfaz gráfica se está incrementando en los juegos para PC (donde existe mucha más flexibilidad para modificarlos), ya que un mayor número de juegos incorpora LUA u otros lenguajes parecidos.

Figura 5. Interfaz modificada con extensiones desarrolladas con LUA, World of Warcraft © Blizzard Entertainment

5.4.4. Guiones de vídeos

Finalmente, otra de las tareas que se suele realizar con un *script* se conoce por 'vídeos de transición'. Estos vídeos se muestran normalmente al principio de la partida y de cada nuevo nivel, y nos sirven para situar al jugador en el estado actual de la partida.

Estos vídeos se pueden reproducir mediante ficheros AVI o equivalentes, aunque la tendencia es cada vez más usar el propio motor gráfico del juego para mostrar una secuencia programada de movimientos de los objetos y las cámaras, dando la sensación al jugador de que está viendo una película pregrabada.

Nota

Por ejemplo, en Half Life la presentación de la historia se realizaba con el propio motor del juego y el usuario podía cambiar el punto de vista, imposible de hacer con vídeo tradicional

Este tipo de vídeos también se acostumbra a describir utilizando lenguajes de *scripting*. En este caso tenemos que programar dentro del código en LUA el guión de los acontecimientos paso a paso. Cuando llega el momento de reproducirlo, el juego deja de tratar todos los eventos de entrada y va interpretando lo que está en el fichero de *scripting*.

Video.lua

```
moverOrco(orco1, pos1, pos2)
reproducirAnimacion(orco1, "caminar")
moverCamara(pos3, pos4)
escribirDialogo("Has entrado en las tierras de los orcos")
reproducirSonido(gritoOrco)
```

Resumen

En este módulo didáctico hemos introducido uno de los principales componentes de un videojuego, el motor de lógica, que es lo que va a diferenciar nuestro juego de cualquier otro tipo de aplicación.

En primer lugar hemos dado un repaso a algunas técnicas básicas de programación que se utilizan en el desarrollo de videojuegos para estructurar correctamente todos los componentes que necesitamos (API, gestores de eventos, controladores...). Estas técnicas son imprescindibles para el desarrollo de proyectos medianos o grandes, ya que nos ayudan a separar el trabajo en diferentes módulos especializados.

Posteriormente, hemos explicado cómo utilizar estas técnicas de programación para implementar el programa principal de un juego. Hemos comentado cómo integrar las diferentes API y hacerlas visibles a todos los componentes del juego. También hemos explicado el funcionamiento de un diagrama de estados y cómo éste nos ayuda a segmentar las diferentes partes de un juego. Para terminar, hemos comentado cómo funciona el bucle principal del juego donde se realizan tres tareas clave: recoger la información de entrada, calcular el nuevo estado del juego y actualizar todos los dispositivos de salida.

Hemos introducido el motor de lógica como el componente más importante de un juego, donde definimos sus reglas y controlamos el estado de todos los elementos que lo componen. El motor lógico trabaja sobre lo que llamamos la vista lógica, donde se encuentran todos los parámetros del juego necesarios para saber cómo se desarrolla una partida. Hemos explicado cómo el motor lógico usa esta información y cómo se aplican las reglas de decisión sobre los elementos del juego.

Otro elemento importante de un juego son todos los datos que lo componen. Hemos explicado los diferentes tipos de datos que se usan y cómo podemos almacenar estos datos. Hemos hablado de XML como un ejemplo de un lenguaje descriptivo que nos permite almacenar la información de forma estructurada.

En la última parte del libro hemos introducido los lenguajes de *scripting*. Los lenguajes de *scripting* nos permiten separar la parte compilada del código, donde normalmente programamos las funciones más básicas de la parte de diseño del propio juego, abriendo la posibilidad a otro tipo de perfiles sin grandes conocimientos de programación para que puedan intervenir en la creación de la parte lógica. Hemos explicado el funcionamiento y la integración del lenguaje LUA, uno de los más utilizados en el mundo de los videojuegos, así como las tareas principales que se pueden llevar a cabo con este tipo de lenguajes.

Actividades

1. Estudiad diferentes maneras de sincronizar el bucle principal. Por ejemplo, cómo poder ofrecer siempre el mismo *rate* constante o cómo poder ofrecer el máximo refresco de pantalla sin sufrir desincronizaciones.

2. Como implementaríais un sistema para que los diseñadores del juego tuvieran que saber lo mínimo posible de programación, es decir, que puedan introducir ordenes del tipo:

```
Encuentro[Jugador,Ladron] --> Robar[Ladron,Random[Jugador.contents]]
```

3. Estudiad los diferentes algoritmos para realizar la discretización del mundo. Implementad un sistema capaz de realizar algunas de las discretizaciones vistas en el libro, ya sea en forma de *grid* o de malla de navegación.

4. Comprobad si existen otros polígonos que se puedan utilizar para hacer un *grid*. El dibujo de un *grid* hexagonal no se puede realizar de forma trivial, inventad un algoritmo que sea capaz de pintarlo por pantalla.

5. Pensad y redactad de qué forma guardaríais la información para crear una aventura gráfica basada en algún cuento infantil (por ejemplo, *La caperucita roja, Juan sin miedo, Los tres cerditos*...). No es necesaria su implementación.

6. Pensad y redactad cómo implementaríais las reglas del juego del punto anterior. No es necesaria su implementación.

7. Elaborad un diagrama de estados de un jugador del juego Street Fighter. Definidlo en UML e implementadlo usando Boost Statechart.

Glosario

Delaunay *f* Boris Nikolaevich Delone, matemático ruso que inventó en 1934 la condición de Delaunay (traducción al francés de su apellido, un guiño a sus antecesores).

diagrama de estados *m* Se usan para representar gráficamente máquinas de estados finitos, donde se describen los estados del sistema y las posibles transiciones entre ellos.

GoF *m* Gang of Four. Se refiere a Erich Gamma, Richard Helm, Ralph Johnson y John Vlisides, autores del libro *Desing Patterns*.

grid *m* Matriz de triángulos, rectángulos o hexágonos regulares que se utiliza para discretizar el mundo.

lenguaje descriptivo o de marcas *m* Se trata de un lenguaje que nos permite organizar la información mediante el uso de etiquetas.

lenguaje interpretado *m* Lenguaje que se ejecuta leyendo directamente el código e interpretándolo paso a paso, sin necesidad de compilarlo previamente.

LUA *m* Lenguaje de *scripting* utilizado principalmente en el desarrollo de videojuegos.

malla de navegación *f* Discretización del mundo utilizando polígonos convexos, donde los polígonos adyacentes comparten una arista.

patrón de diseño *m* Definición de un problema recurrente y de su solución simple y elegante.

POO *f* Programación orientada a objetos. Paradigma de programación donde se encapsula en una entidad (objeto) la información y los algoritmos que operan sobre ella.

round-robin *m* Algoritmo para repartir un recurso entre varios procesos de manera equitativa.

timer *m* Mecanismo utilizado para sincronizar las iteraciones de un bucle o un proceso.

XML *m* Sigla de eXtended Markup Language, lenguaje de marcas definido por la W3C que permite el intercambio de información estructura.

Bibliografía

Bjork, S.; Holopaien,J. (2004). *Patterns in Game Design*. Charles River Media.

Flynt, J. P.; Salem, O. (2004). *Software Engineering for Game Developers*. Course Technology PTR.

Gamma, E.; Helm, R.; Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series.

Gutschmidt, T. (2003). *Game Programming with Python, Lua, and Ruby*. Course Technology PTR.

Penton, R. (2002). *Data Structures for Game Programmers*. Muska & Lipman/Premier-Trade.

Sanchez-Crespo Dalmau, D. (2003). *Core Techniques and Algorithms in Game Programming*. New Riders.

Schuytema, P.; Manyen, M. (2005). *Game Development with Lua*. Charles River Media.

Varanese, A. (2002). *Game Scripting Mastery*. Course Technology PTR.