



UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS



Tecnicatura en diseño
y programación de videojuegos

UNL VIRTUAL



Modelos y algoritmos para videojuegos I

Unidad 1

¿Cuestión de nombres?

Docentes

Sebastián Rojas Fredini

Patricia Schapschuk

CONTENIDOS

1. ¿CUESTIÓN DE NOMBRES?.....	2
1.1. Arquitectura	2
1.1.1. Gráficos.....	2
1.1.2. Periféricos de control (Input)	4
1.1.3. Audio	4
1.1.4. Y... ¿nada más?.....	5
1.2. Engines, Middleware... ¿bibliotecas de alto nivel?	5
1.2.1. Middleware	6
1.2.2. Engines	7
1.2.3. SFML.....	7
1.3. Configuremos el entorno.....	8
1.3.1. El árbol de archivos.....	8
BIBLIOGRAFÍA.....	10

1. ¿CUESTIÓN DE NOMBRES?

1.1. Arquitectura

Para comenzar con esta unidad, haremos un repaso de la arquitectura que comúnmente se utiliza a la hora de desarrollar videojuegos. Si bien ésta varía de plataforma a plataforma, aquí presentaremos un esquema de alto nivel, es decir, bastante abstracto, pero que se puede aplicar a la mayoría de las plataformas en las cuales se desarrollan videojuegos. Este esquema está compuesto por diversos componentes que interactúan entre sí para el funcionamiento del videojuego.

1.1.1. Gráficos

El componente más popular de la arquitectura es –en general– el visual. Sin importar la plataforma, el encargado de dibujar nuestros juegos en la pantalla es una placa de video.

Placa de video
Pieza de hardware dedicado que se encarga de dibujar todo lo que aparece en pantalla.

El programador no interactúa directamente con la placa de video, sino que lo hace a través de unas bibliotecas de bajo nivel. Si bien en *Introducción a la programación* se explicó qué es una biblioteca, lo recordaremos brevemente.

Biblioteca
Conjunto de clases y rutinas.

Estas bibliotecas proveen funciones y clases para que el usuario pueda interactuar con el hardware gráfico de manera accesible, ya que de otra forma sería demasiado complejo comunicarse con la placa de video. Estas librerías difieren de plataforma a plataforma. Por ejemplo, en PC existen dos alternativas: Direct3D y OpenGL.

Direct3D es parte de DirectX, que es un conjunto de librerías de Microsoft®, las cuales –por supuesto– sólo funcionan bajo sistemas operativos Windows®.

DirectX es la tecnología más utilizada por los juegos AAA, ya que es muy potente y su uso está tan difundido, al punto que no existe en el mercado una placa de video que no la soporte.

DirectX es desarrollado por Microsoft y los fabricantes de placas de video suministran **drivers** que entienden estas librerías.

Driver
El controlador de dispositivo (<i>driver</i> en español) es un programa informático que permite al sistema operativo interactuar con un periférico.

Por otro lado, OpenGL es una especificación, no una implementación. A saber:

Especificación
Un documento que describe un conjunto de funciones y clases. Esta descripción incluye los comportamientos, definiciones y tipos que utiliza. No incluye detalles de cómo es implementada internamente cada función/clase.

Esta definición difiere de lo que llamamos implementación:

Implementación
Realización en código de un programa o biblioteca. Programación.

Un programa/videojuego puede ser descrito de muchas maneras antes de ser implementado, y la ingeniería del software y las teorías de game design nos proveen de herramientas para esto.

El producto de estas herramientas suelen denominarse *documentos de diseño*, que describen partes o todo lo que se va a desarrollar. Una especificación de biblioteca, por ejemplo, es un tipo de documento de diseño.

En ellos no se brindan detalles acerca de cómo se programarán los elementos descritos. Esto significa que constituyen una descripción de alto nivel, mucho más abstracta que la implementación. Luego será tarea del programador realizar la mejor implementación posible que cumpla con esos documentos.

Para ilustrar lo dicho anteriormente, tenemos el caso de OpenGL. La especificación OpenGL es mantenida actualmente por el grupo Khronos, que es un consorcio conformado por miembros de la industria, enfocados en la creación de un estándar abierto para la manipulación de gráficos.

Estándar abierto
Especificación disponible públicamente. Esto significa que cualquier persona lo puede usar sin pagar regalías o rendir condiciones al grupo que lo publica.

Al ser un estándar abierto, OpenGL apunta a aumentar la compatibilidad e interoperabilidad de aplicaciones y hardware.

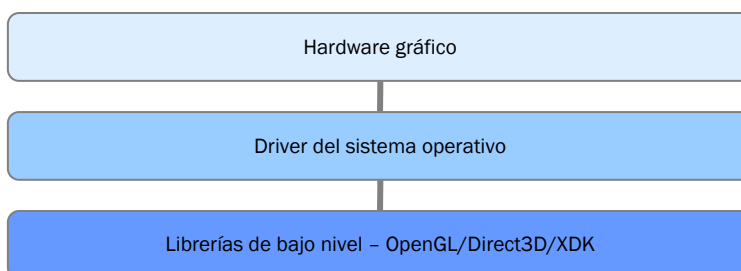
En la práctica esto no suele ser siempre verdad, ya que, al ser una especificación, alguien debe implementar la librería.

En este caso, la implementación está a cargo de los fabricantes de placas de video y esto, generalmente, resulta ineficiente o parcial.

Por ello, un videojuego programado en OpenGL a veces funciona bien con una placa de video determinada y tiene problemas con una similar de otro fabricante. Si a esto le sumamos que las ultimas versiones no han sido capaces de estar a la altura de las necesidades gráficas de la industria, especialmente la de los videojuegos, nos podemos ir haciendo una idea de por qué no son tan populares en este sector.

En las consolas se usan librerías similares. En Playstation® 3, por ejemplo, se utiliza una implementación de OpenGL, mientras que en Xbox360® se emplea una versión extendida de DirectX, denominada XDK.

Sinteticemos, ahora, la jerarquía de los elementos descritos anteriormente en el siguiente gráfico:



Antes de finalizar con este tema, es importante aclarar las diferencias entre DirectX, OpenGL y Direct3D.

Por un lado, DirectX es un conjunto que agrupa varias bibliotecas dentro de las cuales se encuentra Direct3D, que es la que interactúa con el hardware gráfico. Por otro lado, OpenGL sólo es una biblioteca gráfica y, en la jerarquía, está a la misma altura de Direct3D, pero no de DirectX.

1.1.2. Periféricos de control (Input)

Cuando hablamos de periféricos de control, nos referimos a toda pieza de hardware que sirve para establecer una interacción entre el videojuego y el jugador. Los ejemplos más comunes son el joystick, el teclado y el mouse, si bien en los últimos tiempos estamos asistiendo a la producción de tecnologías de detección de movimiento que fueron revolucionando la jugabilidad. De cualquier manera, en esta unidad nos enfocaremos particularmente en los tres periféricos mencionados, que son los más accesibles para todos.

En este caso, también necesitamos una biblioteca que interactúe con el hardware y nos permita leer lo que hace el jugador de manera medianamente sencilla.

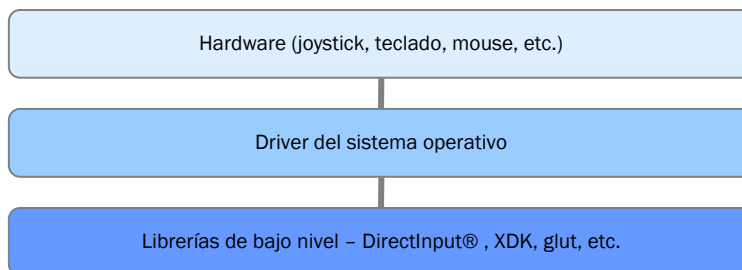
Nuevamente, una de las bibliotecas más populares para PC es DirectInput®, que también forma parte del conjunto de bibliotecas DirectX®.

Si uno realiza un juego con OpenGL en esta plataforma, necesita obtener alguna biblioteca que reemplace a DirectInput, ya que OpenGL no trae soporte para dispositivos de entrada. Así, una de las alternativas más populares para trabajar en conjunto con OpenGL es *glut*.

Por su parte, las consolas proveen soluciones particulares.

En el caso de Xbox360®, la misma utiliza una versión de DirectInput extendida, parte del XDK. En el caso de PlayStation® y Wii®, éstas proveen implementaciones particulares en cada caso y dependen, obviamente, del fabricante.

Si realizamos nuevamente un gráfico de jerarquías, observaremos que la situación es muy similar a la de los componentes gráficos. Más aún: esta jerarquía se repite en casi todos los componentes que veremos.



1.1.3. Audio

El audio es una parte fundamental de un videojuego, ya sea para la música o para los sonidos. Nos brinda la ambientación, nos hace sentir los golpes, nos transmite sensaciones como tristeza, alegría y miedo.

Como podrán imaginar, no vamos a interactuar directamente con el hardware de sonido porque realmente es complicado hacerlo. Utilizaremos, en cambio, bibliotecas que proveen funcionalidad para reproducir sonidos, controlar su posición espacial, modificar su volumen, etc.

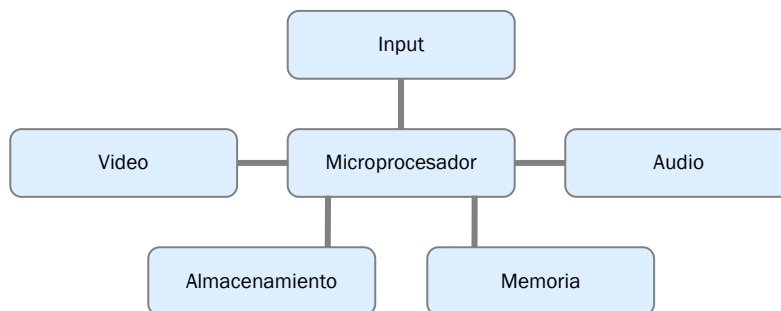
Ahora bien, para PC, DirectX® provee DirectSound® y DirectMusic®, mientras que en otras plataformas o consolas se utilizan soluciones particulares y librerías propias de los fabricantes. De todas maneras, como veremos más adelante, generalmente no se interactúa con estas librerías de bajo nivel, sino que se utilizan soluciones más potentes.

1.1.4. Y... ¿nada más?

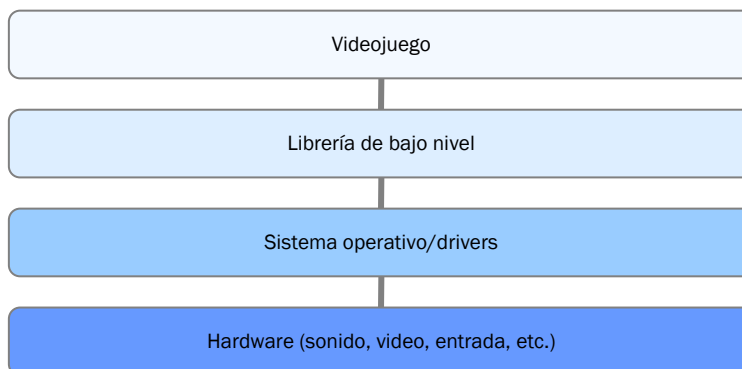
Claramente, existen otros componentes que intervienen en una arquitectura, como los soportes de almacenamiento para guardar las partidas, el contenido descargable, etc. También es muy importante tener en cuenta la memoria RAM y el microprocesador del que se dispone, sus características y prestaciones.

Si bien esto escapa a los contenidos de la materia, vamos a ver un esquema que integra todo lo comentado anteriormente. El mismo no se repite exactamente así en todas las plataformas de videojuegos, pero las diferencias son despreciables en relación a los contenidos de esta materia. Dicho esquema nos sirve para explicar de manera general el funcionamiento e interacción de los componentes.

A nivel de hardware, las comunicaciones e interacciones entre los componentes suelen pasar siempre por el microprocesador, que es el que ejecuta nuestro videojuego y se encarga de manejar todos los elementos que lo integran.



Observemos a continuación cómo quedaría el esquema de interacción a nivel lógico o de software:



En el nivel más alto observamos el videojuego que desarrollamos. Éste se comunica con unas librerías de bajo nivel, como vimos anteriormente (DirectX, OpenGL, etc.), que a su vez dialogan con el sistema operativo, el cual –en última instancia– es el encargado de interactuar con el hardware mediante los drivers.

Este esquema es básico y, como veremos en el próximo párrafo, suele existir otra instancia entre nuestro videojuego y las librerías de bajo nivel.

1.2. Engines, Middleware... ¿bibliotecas de alto nivel?

En el párrafo anterior dijimos que entre nuestro videojuego y las bibliotecas de bajo nivel se suele ubicar, al menos, una instancia más del esquema.

Esta nueva capa se encarga de proveer facilidades para el desarrollo del videojuego, ya que programar directamente sobre una biblioteca de bajo nivel, como DirectX® u OpenGL®, es una tarea compleja y trabajosa.

Entonces, para aliviarle esta tarea al desarrollador y permitirle trabajar de manera más rápida y productiva, surge una biblioteca de más alto nivel, que provee mayor funcionalidad que las que se encuentran jerárquicamente debajo, acortando así los tiempos de desarrollo.

Ahora bien, ¿de qué hablamos cuando decimos *nivel alto o bajo* de una biblioteca? Nos referimos a la funcionalidad que provee. Es decir, las bibliotecas de alto nivel poseen funciones y clases que llevan a cabo tareas más complejas, o más elaboradas; generalmente agrupan en cada función una o más *llamadas* a librerías de bajo nivel, ahorrándonos de esta forma mucho trabajo. En efecto, si estuviésemos programando directamente contra la biblioteca de bajo nivel, deberíamos hacer todas las llamadas.

Por el contrario, las bibliotecas de bajo nivel suelen proveer funcionalidad más básica, tornándose difícil programar directamente sobre ellas. Así, por ejemplo, para crear una ventana en una biblioteca de alto nivel sólo basta indicar el tamaño con un comando. En cambio, si utilizáramos directamente las bibliotecas de bajo nivel, conseguir el mismo resultado nos llevaría varias líneas de código.

No obstante, si bien las bibliotecas de alto nivel se presentan como la mejor alternativa, esto no siempre es cierto, ya que frecuentemente no nos permiten tener un control total sobre lo que hacemos, sino que muchas tareas se realizan de manera automática.

Esta capa intermedia que explicamos, y llamamos genéricamente *biblioteca de alto nivel*, en realidad se suele clasificar en distintos tipos, dependiendo de la funcionalidad que provean. Los más comunes y utilizados en videojuegos son dos: *Engine* y *Middleware*. Estas bibliotecas pueden abarcar uno o más componentes de un videojuego (video, audio, entrada, etc.).

1.2.1. Middleware

Middleware es un término adoptado de la ingeniería del software, pero el significado que adquirió en la industria de videojuegos fue levemente distinto. Para la ingeniería del software, se define como:

Middleware
Software de conectividad que ofrece un conjunto de servicios que hacen posible el funcionamiento de aplicaciones distribuidas sobre plataformas heterogéneas.

Sin embargo, cuando hablamos de *Middleware* en videojuegos, hacemos referencia especialmente al aspecto de conectividad.

Por lo tanto, en este caso, no es necesario que existan aplicaciones distribuidas y plataformas heterogéneas, sino sólo un software que sirva de conectividad entre una aplicación de alto nivel (videojuego) y otra de bajo nivel, como las bibliotecas gráficas. Este software provee funcionalidad a la aplicación de alto nivel y le permite cumplir con sus tareas de manera más sencilla. En otras palabras, posee mucha funcionalidad implementada que sólo tiene utilidad cuando se la incorpora a un videojuego.

Por lo tanto, *Middleware* podría definirse como:

Middleware
Biblioteca de alto nivel que provee funcionalidades parciales para ser integrada en nuestro videojuego.

Como decíamos anteriormente, el *middleware* puede ser gráfico, de sonido o inclusive abarcar partes específicas de un videojuego, como es el caso de *Havok*, un *middleware* que provee funcionalidad de simulación física. Por sí mismo, éste no soluciona el problema del comportamiento físico del personaje, pero posee potentes herramientas que, combinadas, pueden brindarnos la solución.

No obstante, comparados con los *engines*, los *middleware* constituyen sólo una solución parcial.

1.2.2. Engines

Los motores o *engines* son bibliotecas de alto nivel que proveen funcionalidad para el desarrollo integral de uno o más componentes de un videojuego; implementan métodos y clases que responden a las necesidades de las distintas áreas de desarrollo.

Por ejemplo, existen motores gráficos que poseen funciones que se encargan de dibujar todo lo que vemos en pantalla y nos automatizan muchas tareas.

También los hay de audio, de entrada, etc., y aquellos que abarcan más de un componente, comúnmente llamados *game engines*, que proveen una solución completa para todas las áreas involucradas en un videojuego y constituyen la forma más sencilla de empezar a desarrollar.

Además de proveer estas herramientas de programación, los engines suelen incluir editores visuales que, de manera sencilla y rápida, nos permiten diseñar el mundo, los personajes, el sonido, la música, las cinemáticas y todo lo que compone un juego.

Los game engines suelen usar middleware en su funcionamiento. Es decir, podemos pensar que un motor está compuesto por distintas bibliotecas que proveen funcionalidades parciales.

De este modo, podríamos definirlos de la siguiente manera:

Game engine
Una serie de rutinas de programación que permite el diseño, la creación y la representación de un videojuego.

El uso de la palabra *motor* surge porque esta serie de rutinas se puede pensar como el mecanismo que hace mover nuestro videojuego. Es la pieza central y fundamental sobre la cual lo construimos. Como bien marca la definición, un motor comprende la creación (mediante las rutinas), el diseño (generalmente a través de los editores visuales) y la representación del videojuego.

Por lo tanto, podemos afirmar que donde exista un videojuego habrá un motor. Y aunque éste no sea desarrollado explícitamente, las rutinas que implementamos para que el juego funcione cumplen la definición dada y, por ende, componen un motor.

1.2.3. SFML

Para el desarrollo del curso utilizaremos un middleware o librería de alto nivel, denominada SFML. Esta librería nos ayudará a desarrollar de manera sencilla y rápida nuestros prototipos de videojuegos, permitiéndonos enfocarnos en las cuestiones centrales.

SFML es una librería gratuita orientada a objetos y escrita en C++, que ofrece acceso de alto y bajo nivel a los componentes gráficos, de sonido, entrada, red, etc. Dentro de las principales ventajas que nos ofrece, podemos destacar las siguientes:















- *Portable* La librería compila y puede ser ejecutada en plataformas Windows®, GNU/Linux, Unix y Mac OS X.
- *Orientada a objetos* Utiliza las ideas del diseño orientado a objetos.
- *Flexible* Es un middleware que abarca varios componentes básicos de un videojuego, que pueden ser utilizados de manera independiente.
- *Fácil de usar* Provee una sintaxis clara y breve para que podamos concentrarnos en los conceptos y no tengamos inconvenientes técnicos innecesarios.

1.3. Configuremos el entorno

En esta materia utilizaremos la versión 1.6 de SFML, que puede ser obtenida de <http://www.sfml-dev.org/download.php>.

Cuando ingresemos a este sitio, veremos una lista de descargas separadas por sistema operativo y lenguaje, ya que SFML se puede utilizar desde más de un lenguaje, si bien nosotros sólo lo haremos desde C++. El menú que encontraremos es similar a la siguiente figura:

Official SFML libraries

C++ version 1.5		
SFML full SDK (headers / libraries / documentation / sources / samples / external libraries) ←		
 Windows - MinGW (Code::Blocks) (27.1 Mb)	 Linux - 32 bits (12.4 Mb)	 Mac OS X (16.0 Mb)
 Windows - Visual C++ 2005 (22.9 Mb)	 Linux - 64 bits (12.4 Mb)	
 Windows - Visual C++ 2008 (23.1 Mb) ←		
SFML development files (headers / libraries / external libraries)		
 Windows - MinGW (11.1 Mb)	 Linux - 32 bits (713 Kb)	 Mac OS X (4.24 Mb)
 Windows - Visual C++ 2005 (6.89 Mb)	 Linux - 64 bits (723 Kb)	
 Windows - Visual C++ 2008 (7.09 Mb)		
SFML documentation (HTML / CHM)		
  All systems (3.26 Mb)		

Allí podemos observar tres categorías adicionales de descarga para la versión C++ 1.6 de SFML.

Debemos bajar la primera categoría, donde reza *SFML full SDK*, tal como indica la flecha roja.

Dentro de esta categoría observamos que hay varias columnas. Cada una de ellas pertenece a los distintos sistemas operativos. Aquí debemos elegir el que corresponda al nuestro y bajar el archivo.

Nótese que en el caso de Windows® tenemos que elegir la versión que corresponda con nuestro compilador.

En esta materia utilizaremos el IDE Microsoft Visual C++ express 2008, por lo que deberemos bajar el tercer link de la primera columna.

Para cualquier otro IDE libre, generalmente deberán usar el que dice *MinGW*. No obstante, queda a criterio de ustedes utilizar otro IDE o sistema operativo, pero se proveerán las consideraciones necesarias en los casos en que resulte pertinente.

1.3.1. El árbol de archivos

Una vez que hemos bajado SFML, deberemos descomprimirlo en alguna carpeta queelijamos y que será la base para nuestro repositorio de archivos.

A los fines de mantener homogeneidad durante esta materia, extraeremos la librería en C:\SFML-1.6.

Al terminar de descomprimirla, si navegamos con un explorador de archivos y vemos su contenido, observaremos la siguiente estructura:



Ahora explicaremos qué contiene cada carpeta de este árbol.

En primer lugar, tenemos el directorio *build*. El mismo contiene los archivos de proyecto para Visual Studio®, necesarios para compilar SFML. Nosotros no los utilizaremos, ya que la versión que bajamos se encuentra compilada.

En el directorio *doc* está la documentación de la librería dentro de la cual hay una referencia de todas las funciones y clases que la componen con descripciones de los parámetros necesarios para invocarlas.

En *extlibs* se encuentran librerías de terceros compiladas (.lib) que SFML necesita para funcionar.

El directorio *include* contiene todos los archivos de encabezado de C++ con extensión .h. Como recordarán, estos archivos son los que debemos incluir en nuestro programa para poder utilizar la librería. Es importante que conozcamos muy bien los encabezados, ya que son nuestro punto de acceso a la librería y sus rutinas.

En el directorio *.lib* están las librerías propiamente dichas, las cuales utilizaremos para linkear con nuestro programa. Los encabezados describen qué es lo que contienen estas librerías y nos permiten enlazar con ellas.

En el directorio *samples* se encuentra el código fuente de diversos ejemplos de la funcionalidad de SFML, listos para compilar y probar. Es interesante verlos porque demuestran gran parte de la funcionalidad en fragmentos de código sencillos.

Por último, *src* contiene el código fuente con el cual fueron generados los archivos del directorio *lib*. Aquí tampoco usaremos estos archivos porque, como se dijo anteriormente, disponemos de una versión lista para usar en nuestro sistema operativo.

BIBLIOGRAFÍA

Alonso, Marcelo; Finn, Edward. *Física: Vol. I: Mecánica*, Fondo Educativo Interamericano, 1970.

Altman, Silvia; Comparatone, Claudia; Kurzrok, Liliana. *Matemática/ Funciones*. Buenos Aires, Editorial Longseller, 2002.

Botto, Juan; González, Nélica; Muñoz, Juan C. *Fís Física*. Buenos Aires, Editorial tinta fresca, 2007.

Díaz Lozano, María Elina. *Elementos de Matemática*. Apuntes de matemática para las carreras de Tecnicaturas a distancia de UNL, Santa Fe, 2007.

Gettys, Edward; Sélér, Frederick. J.; Skove, Malcolm. *Física Clásica y Moderna*. Madrid, McGraw-Hill Inc., 1991.

Lemarchand, Guillermo; Navas, Claudio; Negroti, Pablo; Rodríguez Usé, Ma. Gabriela; Vásquez, Stella M. *Física Activa*. Buenos Aires, Editorial Puerto de Palos, 2001.

Sears Francis; Zemansky, Mark; Young, Hugh; Freedman, Roger. *Física Universitaria*. Vol. 1, Addison Wesley Longman, 1998.

SFML Documentation, <http://www.sfml-dev.org/documentation/>

Stroustrup, Bjarne. *The Design and Evolution of C++*. Addison Wesley, Reading, MA. 1994.

Stroustrup, Bjarne. *TheC++ Programming Language*. Addison Wesley Longman, Reading, MA. 1997. 3° ed.

Wikipedia, <http://www.wikipedia.org>