UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf


# Didáctica espacial en la NINTENDO DS™

**PROYECTO FINAL DE CARRERA**

*Director:* Manuel Agustí Melchor

*Alumno:* Héctor Cuñat Núñez

**Valencia, Junio 2011**


# DIDÁCTICA ESPACIAL EN LA NINTENDO DS™

**PROYECTO FINAL DE CARRERA**

**DIRECTOR:** ***Manuel Agustí Melchor***
**ALUMNO:** ***Héctor Cuñat Núñez***

**VALENCIA, JUNIO 2011**

Escuela Técnica Superior de Ingeniería **Informática**

etsinf

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

# ÍNDICE

# INTRODUCCIÓN

El presente proyecto surge como una continuación del trabajo realizado en la asignatura *Integración de Medios digitales*, de la titulación de *Ingeniero Técnico en Informática de Sistemas*, de la *Universidad Politécnica de Valencia.*

Dicho trabajo (el cual se recoge como anexo en la presente memoria) consistió en la implementación de una aplicación educativa para *NintendoDS* donde los alumnos de *Educación Secundaria* puedan educar la visión espacial mientras juegan.

Gracias a la base técnica adquirida durante el trabajo de la asignatura, a la ayuda del director del presente proyecto y a la colaboración de los alumnos del centro en el que trabajo como profesor (*figura 1*), pudieron alcanzarse objetivos aún más ambiciosos.


*Fig. 1*: Alumnado del CEIP Eliseo Vidal, el cual ha sido consultado en repetidas ocasiones durante la elaboración del presente proyecto.

Así pues, a continuación, se reproduce el documento que se presentó a la *Escuela Superior de Ingeniería Informática* como propuesta de proyecto final de carrera de tipo B y donde se recogen dichos objetivos:

# PROPUESTA DE PROYECTO FINAL DE CARRERA:

## "MoonseeDS: *DIDÁCTICA ESPACIAL EN LA NintendoDS*"

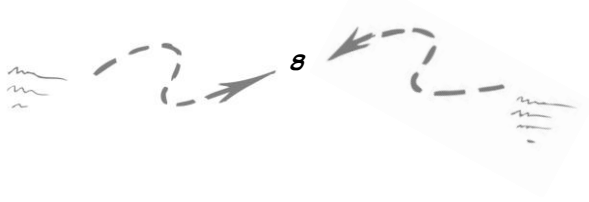
# DATOS DEL PROYECTO

| | |
|---|---|
| *Título:* | MoonseeDS: Didáctica espacial en la NintendoDS. |
| *Tipo:* | B. |
| *Director:* | Manuel Agustí Melchor. |
| *Alumno:* | Héctor Cuñat Núñez. |

# INTRODUCCIÓN

Con el paso de los años los videojuegos han ido cobrando importancia, siendo un factor clave en el desarrollo del individuo desde edades muy tempranas. Por otro lado, no todo el mundo posee una visión espacial adecuada (existen casos en que los tres ejes de coordenadas que se dibujan en una pizarra no son vistos espacialmente, sino como una confluencia plana de tres caminos incidentes).

El presente proyecto consistirá en la implementación de una aplicación didáctica basada en la visualización y manipulación de distintos modelos tridimensionales, ayudando así al desarrollo de una temprana educación de la visión espacial, necesaria para el desarrollo personal, tanto en la vida diaria como en campos académicos: geometría, arte, arquitectura… La aplicación mostrará figuras tridimensionales, realizando al mismo tiempo preguntas sobre las mismas. El programa contestará, en cada caso, si la respuesta es correcta o no.



De esta forma, el presente proyecto se concibe como una contribución a la "didáctica del espacio" que busca, además de resultar lúdica y atrayente, ser lo más viable posible. Por ello, se pretende realizar el desarrollo sobre la plataforma *NintendoDS*, ya que, tras encuestar al alumnado de Enseñanza Secundaria Obligatoria del CEIP Eliseo Vidal, ésa era la consola que la mayoría poseía.

Por último cabe señalar que el punto de partida de este proyecto es continuar con el trabajo realizado por el alumno en la asignatura *Intregración de Medios Digitales*, perteneciente al núcleo de la intensificación *Multimedia* de la titulación de *Ingeniero Técnico en Informática de Sistemas* de la *Universidad Politécnica de Valencia*.

# OBJETIVOS

## OBJETIVOS GENERALES

En este plano, los objetivos son los de construir:

• Una herramienta didáctica sobre una plataforma lúdica.

• Un material utilizable en las clases de Tecnología y/o dibujo Técnico de la Enseñanza Secundaria Obligatoria.

• Unos contenidos formadores de la educación de la visión espacial del alumnado.

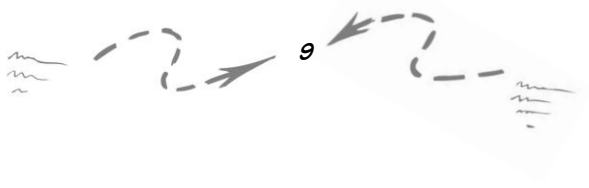
## OBJETIVOS TÉCNICOS

En el plano técnico, la aplicación desarrollada debe conseguir:

• La visualización y manipulación en tiempo real de modelos 3D texturizados.

• Una interfaz atrayente y usable que:

- Permita al usuario voltear los modelos cómodamente.
- Facilite responder a las preguntas formuladas de modo intuitivo y sencillo.
- Subraye de modo sonoro los fallos y los aciertos.



• Estudiar mecanismos que posibiliten la carga externa de los distintos modelos así como de las preguntas a realizar, de forma que el docente pueda personalizar, sin poseer conocimientos de programación, el cuestionario a realizar por la aplicación.

# PLATAFORMAS Y SOFTWARE A UTILIZAR

Además de una *NintendoDS* con cartucho para desarrollo (*flashcart*), será necesario el uso de un ordenador personal con:

• ***S.O. donde se desarrolla:*** Windows 7.
• ***Editor de texto:*** *Notepad++*, para la redacción y edición de código fuente.
• ***Lenguaje de programación:*** C/C++, además de la versión de OpenGL implementada en la NintendoDS, para la programación gráfica 3D.
• ***Compilador:*** *DevkitARM*, compilador multiplataforma *ARM*, incluido en el *toolchain DevKitPro* y basado en el conjunto de herramientas *GCC*.
• ***Librerías a emplear:*** *libnds 1.4.10*, *libFAT 1.0.9*, *PALib 100707* y *MaxMod*.
• ***Emulador de NintendoDS:*** *NO$GBA 2.6*.
• ***Flashcart:*** *SuperCardSD*, con *firmware v1.85*.

# BIBLIOGRAFÍA


• Documentación de las liberías Libnds (http://libnds.devkitpro.org).
• Documentación de las liberías PAlib (http://www.palib-dev.com/manual.html).
• Web de Chishm (http://chishm.drunkencoders.com).
• Web de la librería de sonido MaxMod (http://www.maxmod.org).
• Foro de *El Otro Lado* (www.elotrolado.net).

# HARDWARE NDS

El 12 de Marzo de 2005 *Nintendo* sacó a la venta en España la consola *NintendoDS* (*Dual Screen*), siendo la primera consola en incorporar 2 pantallas (de 3'' cada una, con una resolución de 256x192 píxeles), siendo una de ellas táctil y la otra una pantalla LCD estándar, tal como se muestra en la *figura 2*:


*Fig. 2*: En un extremo está situado el *pad* direccional, en el lado contrario se sitúan cuatro botones que se corresponden con el estándar marcado por los modelos de sobremesa como *SuperNintendo*: *X*, *Y*, *B* y *A*, incluyendo también los botones *SELECT* y *START* (Imagen obtenida de http://zephirothspals.blogspot.com).

El sistema incluye dos procesadores: un procesador principal *ARM9* (de 67Mhz, encargado de la ejecución del juego y su representación en pantalla) y un coprocesador *ARM7* (de 33MHz, encargado del sonido y del soporte WiFi).

El controlador de vídeo es capaz de realizar las siguientes operaciones 3D por hardware: transformaciones e iluminación, transformación textura-coordenada 3D, *tiling*, *alpha blending*, *antialiasing*, *cel-shading* y *z-buffering*. El sistema puede teóricamente generar alrededor de 120.000 triángulos por segundo a 60*fps*, pero sólo puede representar como máximo 2.048 en cada *frame*.

Dicho hardware 3D está diseñado para dibujar una sola pantalla por ciclo de reloj, por lo que cuando se dibujan imágenes 3D diferentes en ambas pantallas, la velocidad disminuye notablemente.

La memoria de vídeo (*VRAM*) es de 656 KB y está estructurada según el mapa de memoria mostrado en la *figura 3*:

| Name | Start Address | Stop Address | Size |
|---|---|---|---|
| **Video RAM** | | | |
| Main OAM | 0x07000000 | 0x070003FF | 1KB |
| Sub OAM | 0x07000400 | 0x070007FF | 1KB |
| Main Palette | 0x05000000 | 0x050003FF | 1KB |
| Sub Palette | 0x05000400 | 0x050007FF | 1KB |
| Bank A | 0x06800000 | 0x0681FFFF | 128KB |
| Bank B | 0x06820000 | 0x0683FFFF | 128KB |
| Bank C | 0x06840000 | 0x0685FFFF | 128KB |
| Bank D | 0x06860000 | 0x0687FFFF | 128KB |
| Bank E | 0x06880000 | 0x0688FFFF | 64KB |
| Bank F | 0x06890000 | 0x06983FFF | 16KB |
| Bank G | 0x06894000 | 0x06897FFF | 16KB |
| Bank H | 0x06898000 | 0x0689FFFF | 32KB |
| Bank I | 0x068A0000 | 0x068A3FFF | 16KB |

*Fig. 3*: Mapa de memoria de la VRAM (imagen obtenida de http://dev-scene.com).

Donde los bancos de memoria A,B,C y D pueden utilizarse para almacenar texturas (disponiendo así de 512KB para ello, dando lugar a un tamaño máximo de textura de 1024x1024 píxeles).

Tiene compatibilidad con *WiFi IEEE 802.11b*. La unidad también soporta un protocolo especial inalámbrico creado por *Nintendo* utilizado en el programa de chat *PictoChat*). La conexión *WiFi* se usa para acceder a Internet o para su uso en el modo multijugador de algunos juegos.

# OTROS MODELOS

***NintendoDS Lite***

Consola portátil fabricada por *Nintendo* en 2006 y desarrollada para suceder a la *NintendoDS*. Las únicas novedades fueron la estética, mucho más estilizada y pulida, y la posibilidad de elegir entre cuatro niveles de brillo.

***NintendoDSi***

Nuevo modelo con un diseño muy similar al de la *DS Lite*, pero con pantallas ligeramente más grandes y con dos cámaras interactivas de 0.3 megapíxeles que pueden ser utilizadas para tomar fotografías. Posee una nueva interfaz al estilo *Wii* de *Nintendo*, ranura de tarjeta SDHC, navegador de Internet, conexión a la Tienda *NintendoDSi* y posibilidad de subir fotos a *Facebook* directamente desde la consola.

***NintendoDSiXL***

Modelo idéntico al anterior salvo por un tamaño de pantalla notablemente mayor ( 4,2'' frente a las 3,25" de la *DSi* y las 3" de la *DSLite*).

# POTENCIA GRÁFICA

Para tener una idea concreta de la potencia gráfica 3D de la *NintendoDS*, se crearon dos ejecutables a partir del código del ejemplo 6 (*figura 4*) de la famosa serie de tutoriales sobre *OpenGL* de *NeHe* (http://nehe.gamedev.net). Ambos emplean las mismas instrucciones de *OpenGL* y muestran la misma escena por pantalla, mostrando además el número de *frames* por segundo. Sin embargo, el primero fue escrito para ejecutarse en una en una *NintendoDS* y el segundo para ejecutarse en diferentes *PCs*.

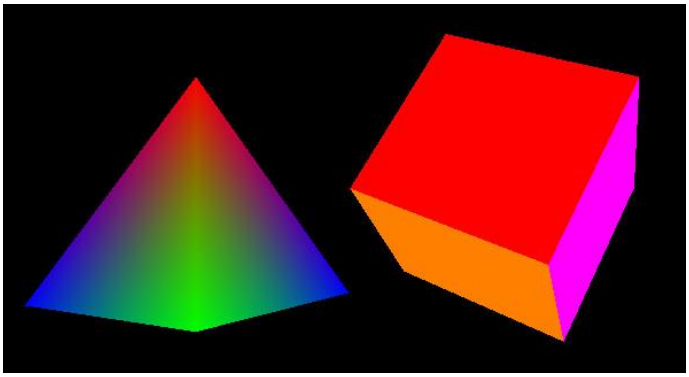


*Fig. 4*: Al ejecutar el ejemplo 6 de *NeHe*, se muestran por pantalla una pirámide y un cubo girando sobre sí mismos.

Así pues, comparando el número de *frames* por segundo empleado por cada ejecutable para representar las escena podemos hacernos una idea de la potencia relativa de cada una de las plataformas en que se ha ejecutado, tal como se muestra en la *figura 5*:

| **FPS** | 72 *fps* | 7684 *fps* | 925 *fps* | |
|---|---|---|---|---|
| **PLATAFORMA** | NDS | PC SOBREMESA | PC PORTÁTIL | |
| | ARM9, 67MHz | Pentium 4, 3200MHz | Mobile Dual Core 1800MHz | ***Procesador*** |
| | 656KB | 3GB DDR2 | 3GB DDR2 | ***RAM*** |
| | | ATI Radeon X700 (128MB) | Mobile Intel (1GB) | ***Gráfica*** |
| | | Windows XP | Windows 7 | ***S.O.*** |

*Fig. 5:* Resultados obtenidos en diferentes tipos de equipos.

A pesar de que a simple vista queda patente la poca potencia que presenta una *NintendoDS*, ésta es aún menor en realidad, puesto que el portátil y el ordenador de sobremesa representaron la escena a una resolución de *1280x1024* píxeles mientras que la *NintendoDS* únicamente a una resolución de *255x192* píxeles.

Así pues, dada la poca potencia del dispositivo elegido, resulta de extrema importancia la optimización en todo aquello relativo a la programación gráfica 3D, como se detallará en los apartados *Modelado y Texturas* y *Cargador de *.obj's* de la presente memoria.

# ENTORNO DE DESARROLLO

Tal como se especifica en el capítulo de introducción, la presente aplicación se programó en un ordenador personal bajo el sistema operativo *Windows 7*.

Para la escritura y edición de código fuente, en lenguaje *C*, se utilizó el programa *Notepad++* (*figura 6*), compilando dicho código mediante el compilador *DevkitARM,* incluido en la cadena de herramientas *DevKitPro.*

```
749  //
750  //
751  // Function: main()
752  int main()
753  {
754      PA_Init();    // Initializes PA_Lib
755
756      PA_Init8bitBg(1, 0); // We'll draw the characters on the screen...
```

*Fig. 6: Notepad++* reconoce y colorea la sintaxis del código escrito en lenguaje *C*.

Las distintos ejecutables fueron testados en el emulador *NO$GBA 2.6* y ejecutados directamente sobre una *NintendoDS Lite* mediante el uso de los *flashcarts SupercardSD* y *M3iZero*.

## LIBRERÍAS UTILIZADAS

***LibNDS (v1.4.10)***

*LibNDS* es una librería creada por Michael Noland (*joat*) y Jason Rogers (*dovoto*) y actualmente mantenida por Dave Murphy (*WinterMute*). Pretende ser una alternativa *open source* al *SDK* comercial de *Nintendo* para la consola *NintendoDS*. *Libnds* da soporte a prácticamente todas las funcionalidades de la *DS*, incluyendo pantalla táctil, micrófono, hardware 3D, hardware 2D y *WiFi*.

Es importante destacar que éstas son librerías de bajo nivel, incluidas en *DevKitPro* y que su API (http://libnds.devkitpro.org) contiene las instrucciones de *OpenGL* que permitirán la programación gráfica 3D.

***LibFAT (v1.0.9)***

Incluidas también dentro de la cadena de herramientas *DevKitPro*, estas librerías permiten el acceso al sistema de archivos del *flashcart* utilizado, permitiendo cargar ficheros de forma externa al ejecutable.

Fueron creadas, y están actualmente mantenidas, por *Chishm* (http://chishm.drunkencoders.com).

***ASLib (v1.0)***

Librerías creadas por Noda (http://nodadev.wordpress.com) y utilizadas en la reproducción de música de fondo en formato *mp3* y de efectos sonoros en formato *raw*.

Estas librerías fueron utilizadas en lugar de las *MaxMod*, al contrario de lo que inicialmente se planeó, debido a su sencillez, a la funcionalidad de reproducir archivos de audio en formato *mp3* y por no requerir la elaboración previa de una biblioteca de sonidos.

***PAlib (beta 100707)***

Librerías creadas por *Mollusk* y actualmente abandonadas. Al ser unas librerías construidas sobre las *libNDs*, proveen funciones de más alto nivel que facilitan considerablemente la programación para *NintendoDS*.

Gracias a la inclusión de estas librerías (no se había hecho uso de ellas en el proyecto realizado en la asignatura IMD) se ha podido añadir a la aplicación funcionalidades tan interesantes como acceso al reloj del sistema, teclado táctil,... de una forma casi inmediata.

# MAKEFILE UTILIZADO

Un *makefile* es un archivo de texto utilizado en la gestión de compilación de programas. En él se recogen las dependencias entre las diferentes partes de un proyecto. Todos los *makefiles* están ordenados en forma de reglas, especificando qué ha de hacer el compilador para obtener un módulo en concreto.

Sin embargo, como se detalla en el anexo de la presente memoria (en el punto *Integración de las distintas funcionalidades en una única aplicación*), una de las dificultades que más tiempo consumió fue la elaboración de un *makefile* que integrase las dependencias requeridas por las distintas funcionalidades de la aplicación. En concreto, podemos leer:

> *"La primera gran dificultad radica en que el Makefile que nos permite cargar los modelos en 3D no está preparado para trabajar con GRIT, y el Makefile del proyecto de ejemplo que permite cargar fondos, no permite la manipulación de figuras tridimensionales".*

Afortunadamente, la instalación de las librerías *PAlib* soluciona este problema de forma inmediata, pues proporciona un *makefile* "universal" donde intenta recoger el conjunto de dependencias y reglas requeridas por los distintos tipos posibles de fichero. Para ello, dichos ficheros deben encontrarse en la carpeta adecuada dentro del directorio del proyecto, tal como se muestra en la *figura 7*:

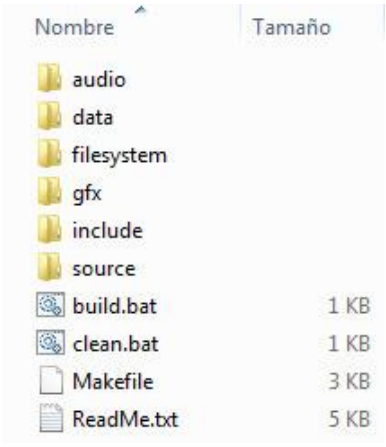



*Fig. 7*: Contenido de una carpeta de proyecto.

Incluiremos los ficheros de código fuente en la carpeta *source*, los fondos y los *sprites* en la carpeta *gfx (figura 8)*, los archivos de sonido en la carpeta *audio* y los modelos tridimensionales y sus correspondientes texturas en la carpeta *data*.

Con *build.bat* generaremos el ejecutable, creado por el compilador a partir de las dependencias especificadas en el *makefile*.

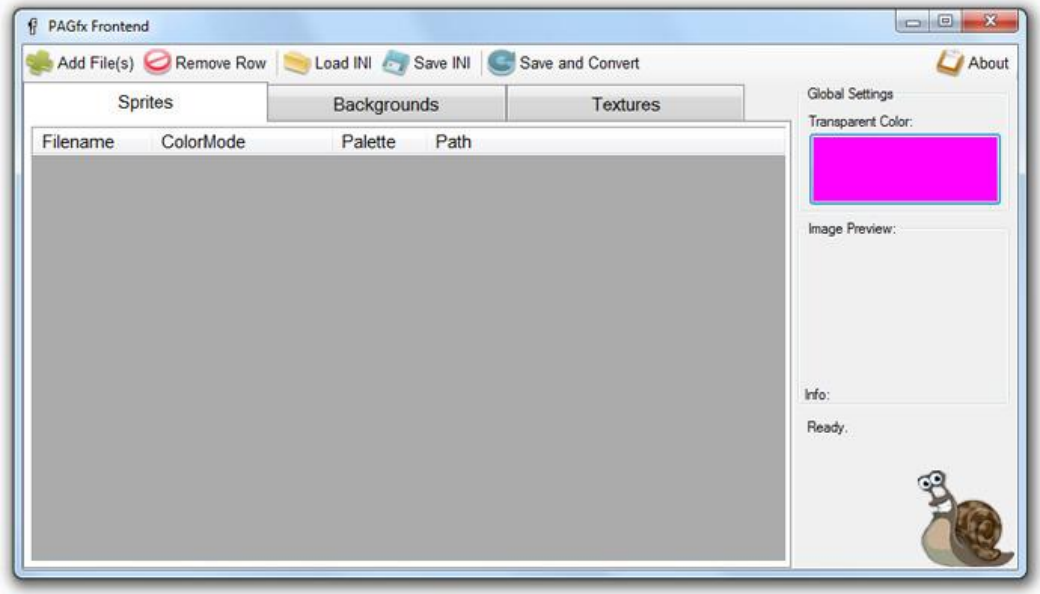



*Fig. 8*: Dentro de la carpeta *gfx* podemos encontrar la aplicación *PAGfx*, la cual nos permitirá transformar los *sprites*, los fondos y las texturas a un formato compatible con la NintendoDS.

Abriendo dicho *makefile* con un editor de texto, podemos acceder e incluso editar la siguiente información de interés (*figura 9*):

*#Especificamos el core a cargar en el procesador secundario, en este caso la opción por #defecto:* **arm7_mp3.bin**, *la cual nos proporcionará funcionalidades de reproducción de #archivos de sonido en *.mp3.*

```
...
ifeq ($(strip $(ARM7_SELECTED)), ARM7_MP3)
      ARM7BIN    := $(PAPATH)/lib/arm7_mp3.bin
      ARM7_IS_OK := YES
endif
...
```

*#Seleccionamos ahora aquellas librerías que queramos incluir en nuestro proyecto.*

```
...
ifeq ($(strip $(ARM7_SELECTED)), ARM7_MP3)
      LIBS := -lfilesystem -lfat -lnds9
endif
...
```

Vemos cómo para incluir una nueva librería externa a *PAlib*, basta con añadir su nombre, precedido de un guión, a la lista de librerías. Es necesario explicar que no se hace referencia a la librería de sonido *ASlib* por estar incluida dentro de la versión utilizada de *PAlib*.

*#Indicamos dónde se encuentran los archivos de código fuente...*

```
...
CFILES   := $(foreach dir,$(SOURCES),$(notdir $(wildcard
$(dir)/*.c)))
CPPFILES := $(foreach dir,$(SOURCES),$(notdir $(wildcard
$(dir)/*.cpp)))
...
```

*# ...y los distintos recursos incluidos en la aplicación (fondos, música, modelos,...).*

```
...
BINFILES := $(foreach dir,$(SOURCES),$(notdir $(wildcard
$(dir)/*.bin)))
MP3FILES := $(foreach dir,$(SOURCES),$(notdir $(wildcard
$(dir)/*.mp3)))
...
```

Cabe señalar que no se hace referencia directa a archivos de imagen ni a modelos tridimensionales y texturas porque éstos han sido convertidos previamente a formato **.bin* (mediante el programa *PAGfx* o los programas *NDS_Mesh_Converter* y *bmp2bin*) tal como se verá más adelante en el punto *Modelado y Texturas* de la presente memoria.

*# Por último especificamos las reglas para la construcción de...*
*# ...el ejecutable:*

```
#---------------------------------------------------------------------
%.nds: %.bin
#---------------------------------------------------------------------
	@ndstool -c $@ -9 $(TARGET).bin -7 $(ARM7BIN) -b $(ICON)
"$(TEXT1);$(TEXT2);$(TEXT3)" $(FILESYSTEM) > /dev/null
	@echo
	@echo Built: $(notdir $@)
	@echo
```

*# ...el código fuente:*

```
#---------------------------------------------------------------------
%.o: %.cpp
#---------------------------------------------------------------------
	@echo $(notdir $<)
	@$(CXX) -MMD -MP -MF $(DEPSDIR)/$*.d $(CXXFLAGS) -c $< -o $@

#---------------------------------------------------------------------
%.o: %.c
#---------------------------------------------------------------------
	@echo $(notdir $<)
	@$(CC) -MMD -MP -MF $(DEPSDIR)/$*.d $(CFLAGS) -c $< -o $@
```

*# ...y los distintos recursos...*

```
#---------------------------------------------------------------------
%.o: %.bin
#---------------------------------------------------------------------
	@echo $(notdir $<)
	@$(bin2o)
#---------------------------------------------------------------------
%.o: %.mp3
#---------------------------------------------------------------------
	@echo $(notdir $<)
	@$(bin2o)
```

*# ...los cuales hacen uso de la macro* **bin2o***:*

```
define bin2o
	cp $< $*
	bin2s $* | $(AS) -o $@
	rm $*

	echo "extern const u8" $*"[];" > $*.h
	echo "extern const u32" $*_size";" >> $*.h
endef
```

*Fig. 9*: Fragmentos del *makefile* incluido en PALib.

# FLASHCARTS Y DLDI

Los *flashcarts* son dispositivos utilizados para ejecutar programas caseros (*homebrew*) en la *NintendoDS*, ya que ésta no se vende con ningún medio regrabable de almacenamiento. Por tanto será necesario el uso de uno de estos dispositivos para poder ejecutar la aplicación programada directamente en una *NintendoDS*.

Cabe remarcar la extrema importancia de conseguir ejecutar la aplicación desarrollada directamente en una *NintendoDS*: Además de que el emulador no emula el hardware de la videoconsola de forma exacta, no podrán utilizarse aquellas funciones que consulten datos internos de la consola, como puede ser el reloj del sistema. Además, obviamente, tampoco podremos acceder al sistema de archivos del *flashcart* para cargar elementos externos al ejecutable.

Básicamente hay dos tipos de *flashcarts* (*figura 8*): los más antiguos, que utilizan la ranura para los juegos de GameBoyAdvance (SLOT-2) y que necesitan de algún dispositivo de arranque en el SLOT-1 (la ranura para los juegos de NintendoDS); y las más modernas, que utilizan directamente el SLOT-1.


*Fig. 10*: Las dos *flashcarts* utilizadas a lo largo del presente proyecto: *SupercardSD*, de tipo SLOT-2 y *M3iZero*, de tipo SLOT-1 (Imágenes obtenidas de http://images.discoazul.com/ y http://www.dsiconsolas.com.

Como ya se ha comentado con anterioridad, se pretende que la aplicación del presente proyecto cargue las figuras problema de forma externa al ejecutable (de forma que cada profesor pueda incluir sus propias figuras o variar las existentes cada cierto tiempo). Para ello, es necesario acceder al sistema de archivos del *flashcart* mediante las librerías *libFAT*.

Con el propósito de que nuestra aplicación fuese compatible con el máximo número posible de *flashcarts*, deberíamos incluir los drivers de todos ellos en nuestro ejecutable. Sin embargo, cabe remarcar que cuando saliesen al mercado nuevos modelos, nuestra aplicación probablemente no sería compatible con ellos.

Como consecuencia de esto, el mismo autor de las libFAT (Chishm) desarrolló también las DLDI (*Dynamically Linked Device Interface for libfat*), librerías en las cuales en lugar de incluir los drivers de los distintos *flashcarts* en el ejecutable, se incluye un contenedor de 32Kb. Cada usuario, mediante el programa *DLDIPatcher* (disponible para *Windows* y *Linux*, ver *figura 11*) introduce en dicho contenedor el driver del *flashcart* en el que va a ejecutar la aplicación. De esta manera y con un solo binario (archivo **.nds*) se podrá tener compatibilidad con todos aquellos *flashcarts* que tengan soporte con las librerías DLDI.

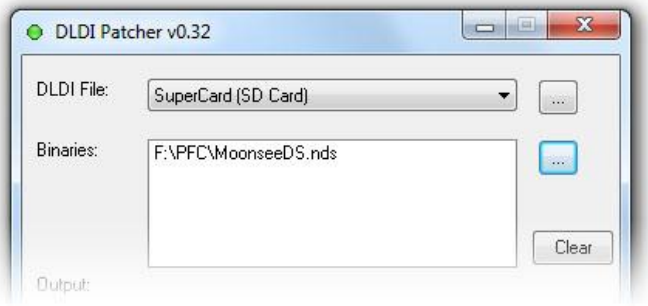



*Fig. 11*: Menú del DLDIPatcher utilizado para ejecutar *homebrew* en la *SupercardSD*, la más antigua de las 2 flashcart utilizadas en el presente proyecto.

Afortunadamente, la gran mayoría de los *flashcart* actuales incluyen parcheo DLDI automático (como la *M3iZero*, utilizada en el presente proyecto) no haciendo falta incluir manualmente el driver para poder ejecutar la aplicación.

# DISEÑO GAMEPLAY

No ha de perderse de vista en ningún momento que la presente aplicación tiene una finalidad didáctica. Sin embargo, es de vital importancia que no resulte aburrida a los alumnos. Por tanto, fue necesario diseñar el *gameplay* del juego poniendo especial atención a que resultase atractivo y adictivo a la vez que permitiese adquirir al alumno las competencias deseadas.

Las dos principales fuentes de inspiración fueron los siguientes juegos comerciales (*figura 12*), donde cada uno representó una de las dos vertientes entre las que se debía encontrar un equilibrio: diversión y educación.

***CANABALT***

***BIG BRAIN ACADEMY***


*Fig. 12*: *Canabalt* (*AdamAtomic*, para navegador web y *iPhone/iPod*) es un juego sencillo pero extremadamente adictivo donde el jugador ha de recorrer la máxima distancia posible mientras esquiva obstáculos. Se pudo comprobar con el alumnado lo adictivo que resultaba y cómo fomentaba la competitividad por ver quién recorría mayor distancia.

*BigBrainAcademy* (*Nintendo*, para *NintendoDS* y *Wii*) es un conjunto de puzles que pretende medir la inteligencia y/o tiempo de reacción del jugador.

Imágenes obtenidas de http://www.gman.tv/ y http://cdn1.gamepro.com/

Así pues, se diseñó un sistema de juego en el que el alumno debe ayudar al protagonista a completar una carrera donde los distintos obstáculos son preguntas relativas a las vistas (alzado, planta y perfil) de un grupo de figuras. Dibujando correctamente el alzado, planta o perfil de las mismas, sortearemos el obstáculo y proseguiremos con la huída.

Una vez definido el *gameplay* básico y tras repetidas consultas al alumnado de primer ciclo de Secundaria del CEIP Eliseo Vidal, con cada una de las sucesivas versiones de la presente aplicación, se definieron las siguientes funcionalidades a implementar:

- Los alumnos deben dibujar la solución, obligándolos a tomar un papel activo y evitando una respuesta tipo test que pueda ser acertada accidentalmente.

- Las preguntas han de aparecer aleatoriamente, de forma que el alumnado deba razonar la respuesta y no pueda memorizar la secuencia de respuestas correctas.

- Las preguntas acertadas, así como el tiempo empleado en contestarlas y su dificultad, han de acelerar o decelerar al protagonista en su carrera. Esto afectará a la larga a la distancia recorrida, fomentando así la competitividad por ver quién recorre una mayor distancia.

- Para fomentar aún más dicha competitividad, es conveniente incluir un registro con los 5 mejores jugadores y sus marcas.

- El test ha de ser difícil de completar, mostrando un final vistoso sólo a unos pocos que sean capaces de superarlo. Esto animará al resto del alumnado a esforzarse por conseguirlo también.

- La música ha de ser relajada y poética, de forma que no precipite al alumno en su respuesta.

Teniendo en cuenta dichas funcionalidades, se diseñó el *gameplay* descrito en la *figura13*:

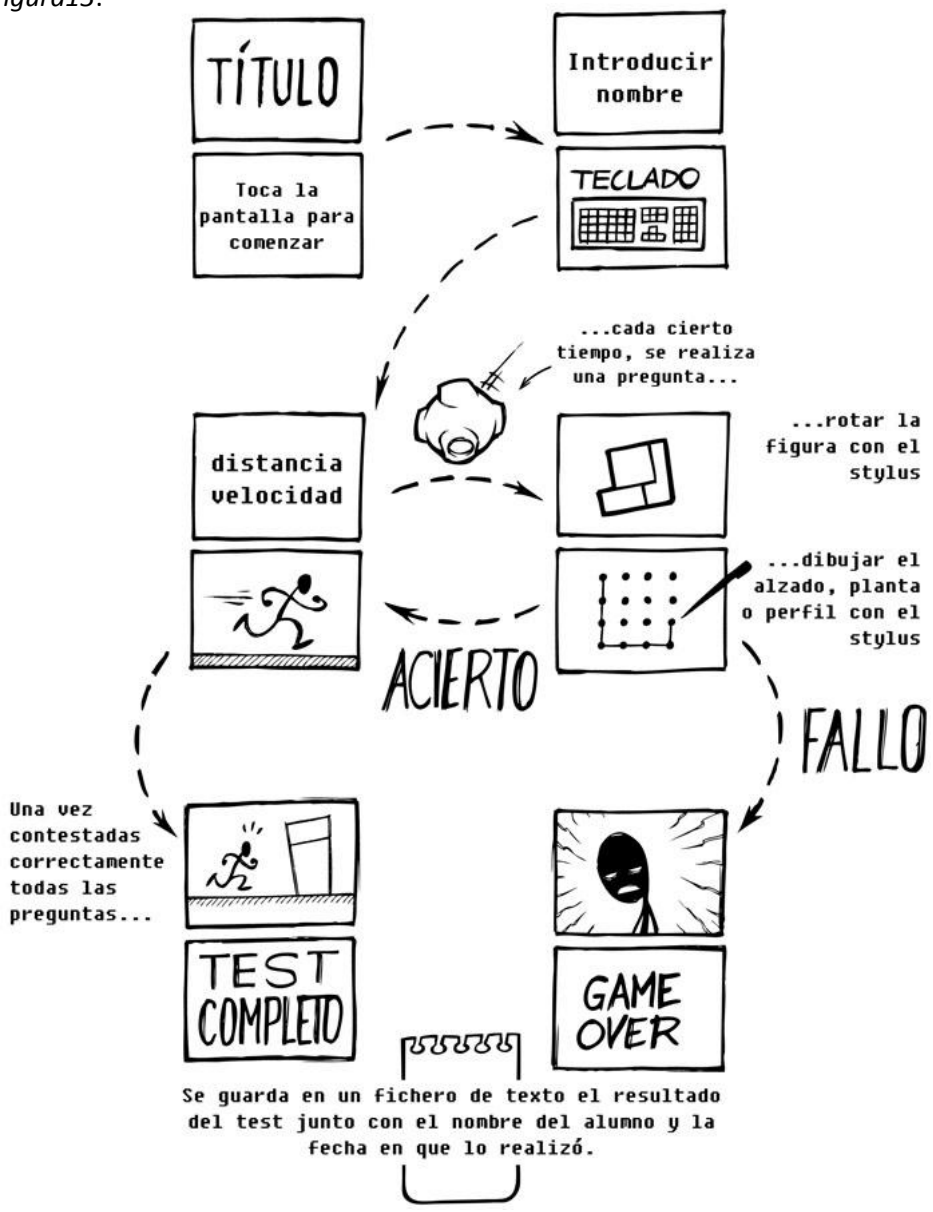



*Fig. 13*: Esquema del *gameplay* de la aplicación.

# DISEÑO ARTÍSTICO

Dada la iniciativa didáctica de la aplicación, se buscó una estética sencilla que permitiese al alumno concentrarse en la solución técnica de la pregunta planteada y que no desvíe su atención hacia otros aspectos menos importantes.

Para poder dirigir dicha atención, se pretendió resaltar con un determinado color los elementos de interés, representando el resto con colores de una misma gama muy distinta a la del color elegido, tal como se muestra en la *figura 14*:


*Fig. 14*: Técnica similar a la descrita utilizada por *Frank Miller* en su novela gráfica *Sin City* (Imagen obtenida de http://www.mtv.com/movies).

Por otra parte, se escogió como música de fondo el tema *Spiegel im Spiegel*, de *Arvo Part,* que permitía ser reproducida cíclicamente a partir de una pista de audio relativamente corta (con el consiguiente ahorro de memoria) y que resultaba especialmente poética y relajante.

Por último, cabe decir que el alumnado votó y opinó acerca de cada uno de los bocetos e imágenes realizados (*figuras 15 y 16*), cada uno de los cuales representaba una posible estética a emplear en la aplicación. Es interesante mencionar además que la gran mayoría opinó que la estética empleada en el proyecto de la asignatura *Integración de Medios Digitales* resultaba demasiado infantil.

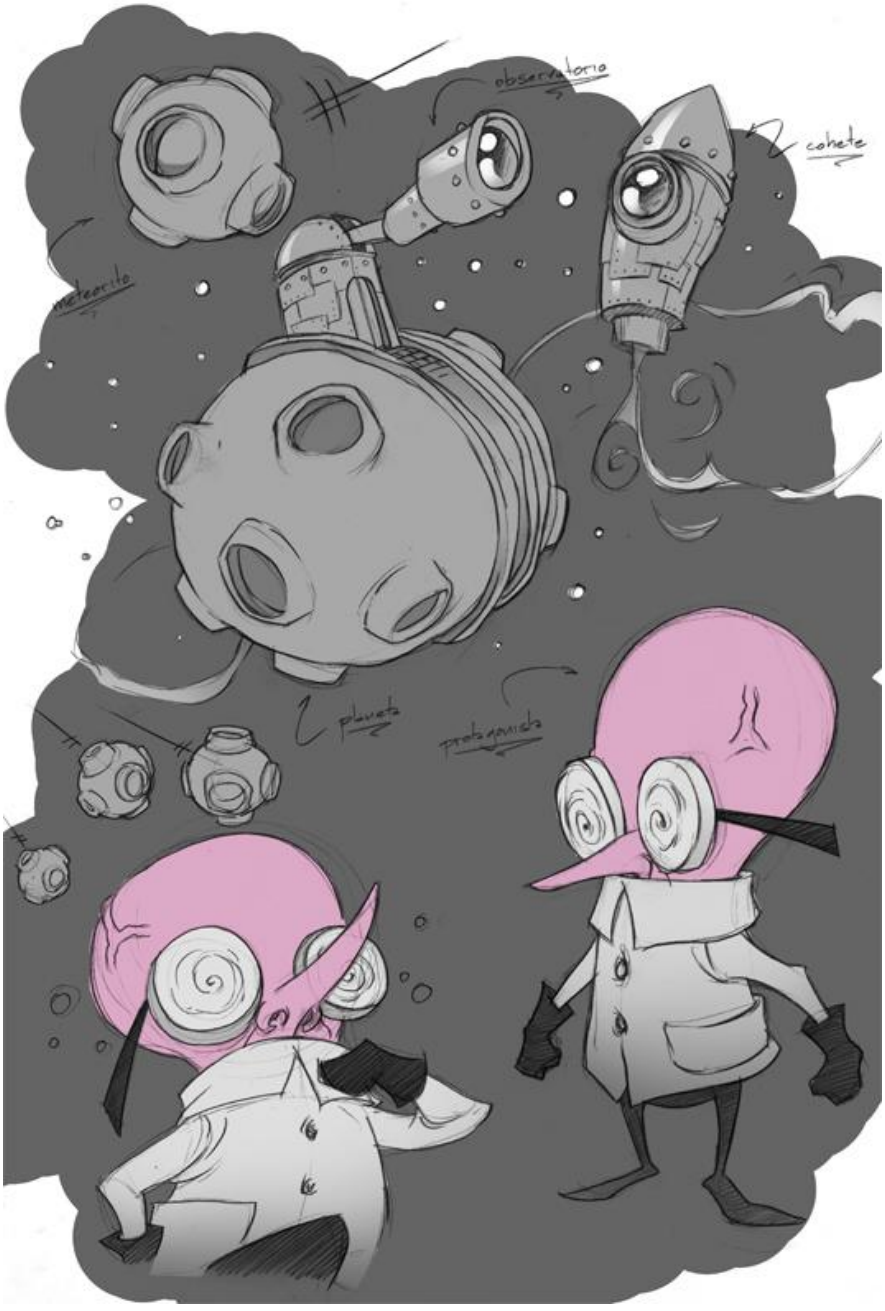



*Fig. 15*: Bocetos que lograron una mayor aceptación entre el alumnado.


*Fig. 16*: Dibujos coloreados en *Photoshop*, a partir de los bocetos realizados, para a su uso como fondos en la aplicación

# MODELADO Y TEXTURAS

## CREACIÓN DE LOS MODELOS TRIDIMENSIONALES

Una vez diseñados los elementos del juego, es hora de crearlos haciendo uso de un paquete de modelado tridimensional, como *Autodesk 3DSMAX* en el caso del presente proyecto.

En primer lugar, conviene tener en todo momento una referencia del diseño elaborado, para asegurarnos que el modelo creado se ajusta al mismo. Para ello, resulta conveniente modelar primero dos planos verticales y perpendiculares entre sí a los que se les aplica como textura los diseños realizados (por ejemplo el protagonista de frente y de perfil, como se muestra en la *figura 17*).

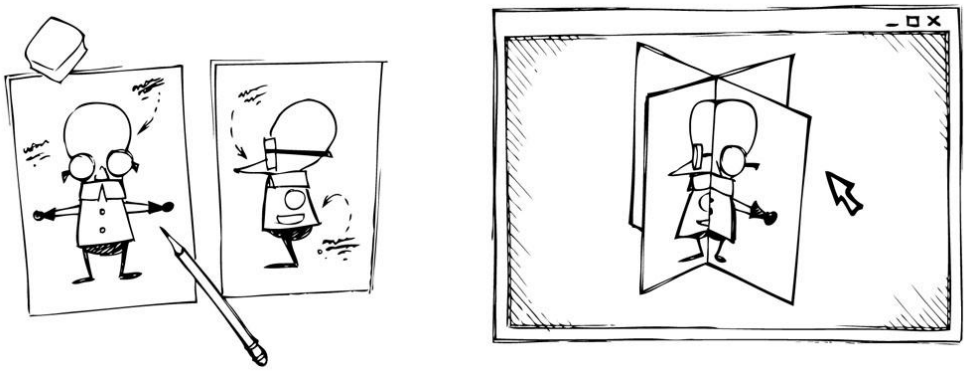


*Fig. 17*: Creación de las referencias artísticas en el paquete de modelado 3D.

Así pues, con dichas referencias como plantilla, se modelaron cada uno de los elementos tridimensionales de la aplicación, utilizando para ello la técnica conocida como *box modeling,* detallada en el siguiente punto.

# BOX MODELING Y QUADS

La técnica de *box modeling* consiste en el modelado de figuras relativamente complejas. Se parte de primitivas gráficas sencillas a cuyas caras se les aplican sucesivas subdivisiones, extrusiones y diversas transformaciones espaciales, tal como se muestra en la *figura 18*:

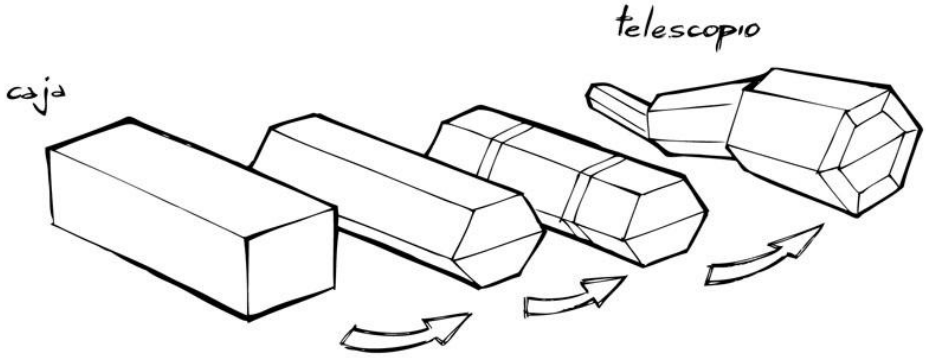



*Fig. 18*: Proceso de modelado del telescopio mediante la técnica *de box modeling*.

Este método, además de ser relativamente sencillo, resulta notablemente más rápido que la manipulación individual de cada uno de los vértices.

El elemento fundamental en la técnica de *box modeling* son las caras de cuatro lados, comúnmente llamadas *quads*, las cuales permiten resultados predecibles y consistentes, ya que pueden subdividirse en 2 o 4 triángulos (trazando una o dos diagonales) que poseen aproximadamente la misma dirección de la normal (recordemos que la gran mayoría de los motores de videojuegos interpreta las geometrías como tiras de triángulos).

Así pues, intentaremos que nuestros modelos estén siempre formados por *quads* en la medida de lo posible, siendo esto estrictamente obligatorio en el caso de las figuras problema, pues éstas se representarán en pantalla mediante el código descrito en la figura 29 (punto *Cargador de *.obj's*) cuya funcionalidad se limita a la representación de figuras integradas por *quads*.

Adicionalmente, en el punto: *Importación, manipulación y texturización de figuras 3D* (contenido en el *Anexo I* de la presente memoria) se muestra el código necesario para la manipulación y texturización de un *quad* en pantalla y se plantea cómo crear figuras más complejas a partir del mismo, tales como el laberinto tridimensional mostrado en la *figura 19:*

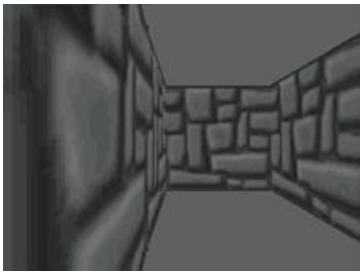


*Fig. 19*: Laberinto similar al del juego *Wolfenstein3D* (*ID software*, 1992) formado por una serie de *quads* texturizados.

Por último, tal como se refleja en el apartado de hardware de la presente memoria, la NDS resulta un dispositivo relativamente poco potente a la hora de procesar gráficos tridimensionales. Así pues, se buscó que los modelos creados fuesen sencillos (con un número de polígonos nunca superior a 1000 (*figura 20*)).

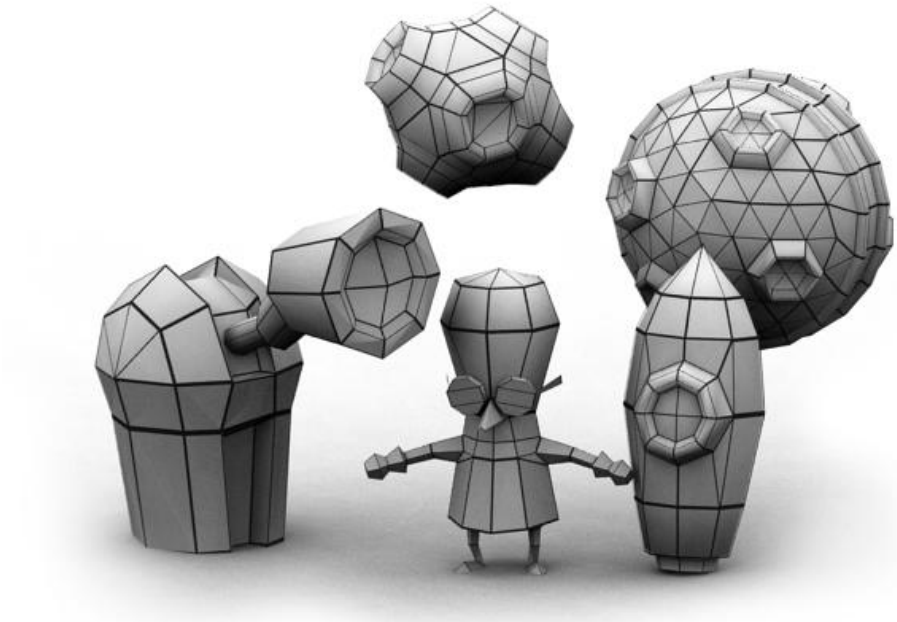


*Fig. 20*: Modelos empleados en la aplicación, mostrando su sencilla malla geométrica.

# TEXTURAS

Como se vio en el apartado de hardware de la presente memoria, la memoria de vídeo de la *NintendoDS* (*VRAM*) tiene 4 bancos de memoria de 128KB en los que puede almacenar texturas (entre otras cosas). Dichos 128KB permiten almacenar una textura de 512x512 píxeles, o bien cuatro texturas con una resolución de 256x256 píxeles o menor.

Así pues, una *NintendoDS* puede como máximo representar en pantalla 16 texturas de 256x256 píxeles simultáneamente (o una única textura de 1024x1024 que ocupe los 4 bancos).

Además, es extremadamente importante tener en cuenta que dichos bancos tienen otras funcionalidades, como la de almacenar *sprites*, fondos,… y que cuanto más espacio de los mismos ocupemos con texturas, menos elementos de otro tipo podremos incluir en nuestra aplicación.

Por tanto, la representación de texturas en pantalla puede resultar aún más limitante a la hora de representar gráficos tridimensionales que la complejidad geométrica de los modelos representados. Así, fue necesario desarrollar estrategias de optimización con el fin de reducir el espacio ocupado por las texturas:

***SIMETRÍA***

En todo momento se buscó que los modelos fueran simétricos (protagonista, planeta,...) con el fin de cubrir la superficie del modelo con una textura cuya área fuese la mitad, tal como puede verse en la *figura 21*:

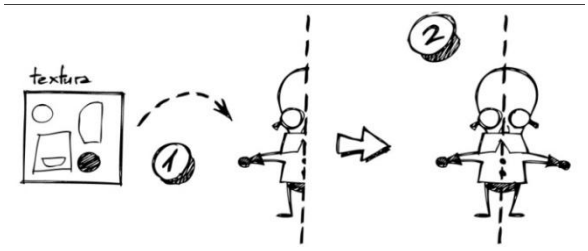



Fig. 21: Ahorro de memoria en la textura realizando un *mirroring* del modelo.

***TILES***

Del mismo modo, en aquellos modelos de grandes dimensiones (planeta), se optó por texturizarlos mediante *tiles* (texturas de tamaño reducido pero que al ser puestas unas al lado de otras dan sensación de continuidad), con el fin de cubrir toda su superficie sin que ello conllevase una pérdida en la resolución (figura 22).

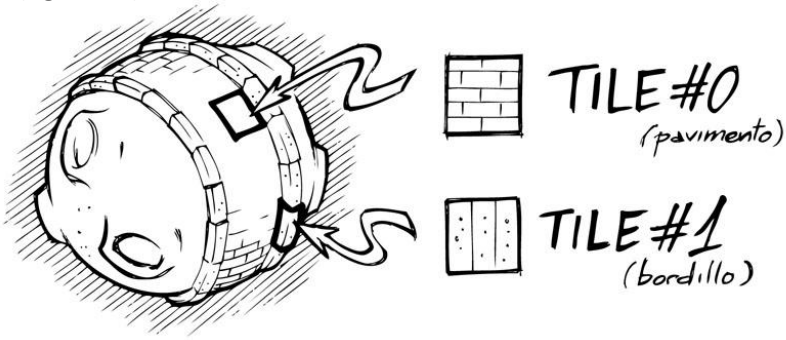



*Fig. 22*: Uso de *tiles* en la texturización del planeta.

***OBJETOS BLANCOS***

Se aprovechó también el hecho de que los modelos a los cuales no se les asigna una textura son representados en blanco, de esta forma las estrellas (todas ellas blancas) no tienen asociada una textura que pueda ocupar memoria.

***DETALLES***

Por último, se le dio mayor tamaño en la textura del modelo a los detalles en los que se pretendía una mayor resolución (*figura 23*), procurando colorear el resto con colores planos, los cuales permanecen invariables al ser escalados.

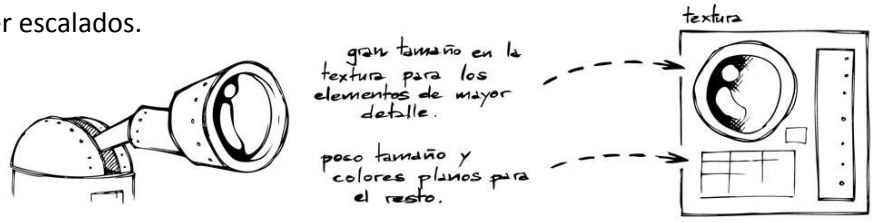



*Fig. 23*: Se pretendió que la lente del observatorio tuviese especial detalle.

Finalmente, el conjunto de modelos y texturas, quedó tal como se muestra en la *figura 24*:

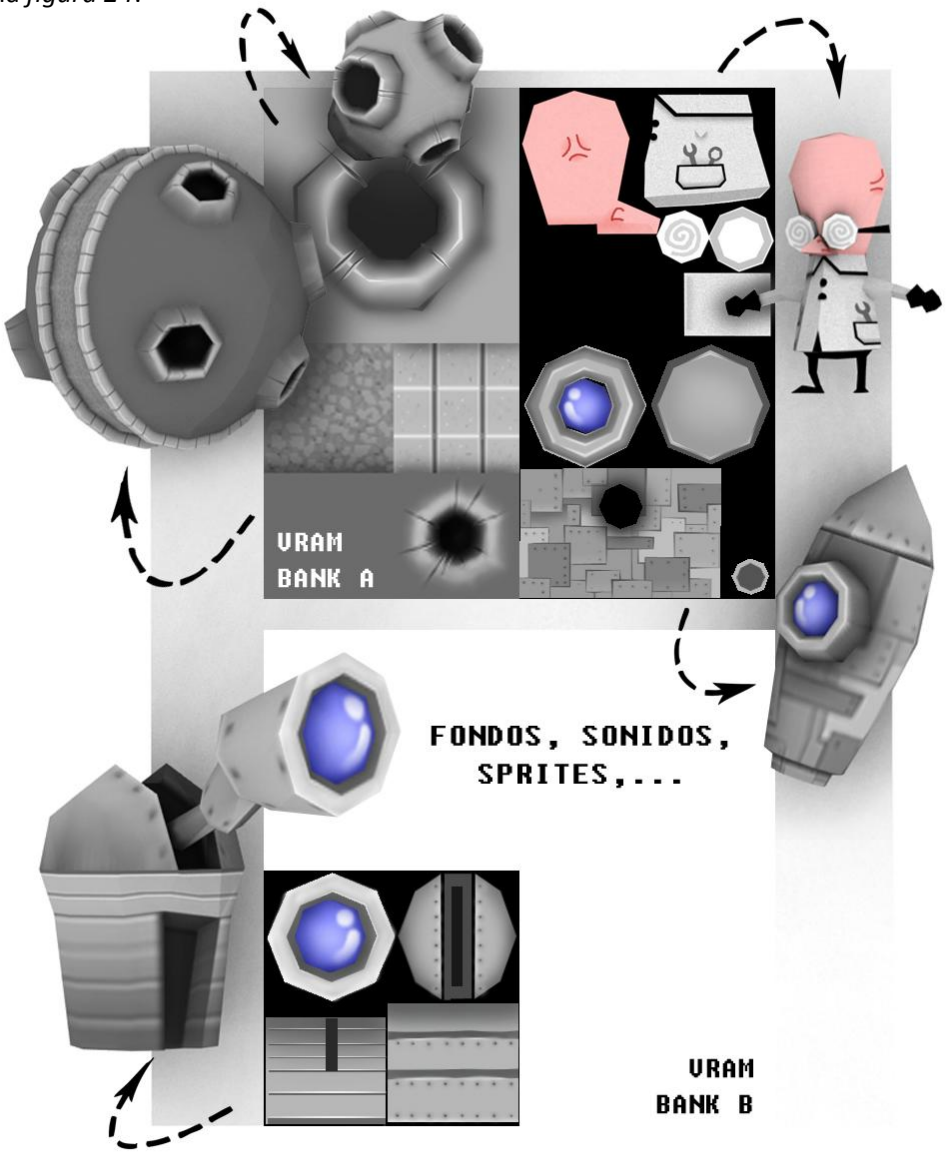



*Fig. 24*: Modelos con sus correspondientes texturas, mostrando el tamaño relativo de éstas y el banco de memoria en el que se ubican.

# INTEGRACIÓN EN LA APLICACIÓN

Para cargar en nuestra aplicación los objetos y texturas creados, es necesario convertirlos primero a un formato compatible con la *NintendoDS* y las librerías utilizadas. Para ello, se hizo uso del programa *NDS_Mesh_Converter* de *PadrinatoR* en el caso de los modelos (guardados previamente en formato **.3ds*) y del programa *bmp2bin* en el caso de las texturas (guardadas previamente como mapas de bits con una profundidad de color de 24 bits), tal como se muestra en la *figura 25*:

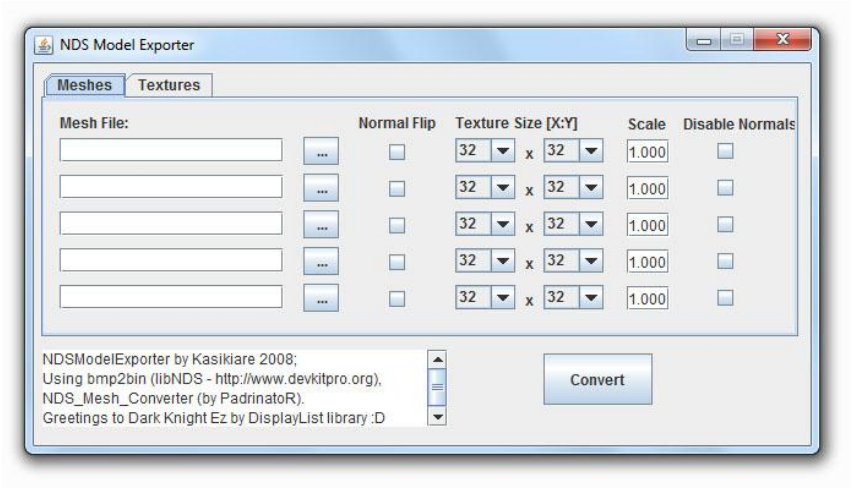



*Fig. 25*: *Frontend 'NDSModelExporter'*, de *Kasikiare* que proporciona una sencilla interfaz Java a los 2 programas nombrados.

Debido a un fallo de *NDS_Mesh_Converter,* es necesario volver a convertir los modelos una vez más, mediante el programa *DLFixer,* de *AntonioND,* para que éstos se vean afectados por las luces *OpenGL* que podamos definir posteriormente.

Una vez convertidos todos nuestros modelos y texturas, debemos almacenarlos en la carpeta *data* de nuestro proyecto y hacer referencia a ellos desde nuestro código:

```
#include "modelo00.h"
#include "textura00.h"
#include "modelo01.h"
#include "textura01.h"
...
```

Así pues, he aquí el código mínimo necesario para representar por la pantalla de la *NintendoDS* un modelo con su correspondiente textura (*figura 26*):

```
#include <nds.h>   //Incluimos las ndslib
#include "modelo.h"    //Incluimos el modelo...
#include "textura.h"   //...y su correspondiente textura

int main(){

	int textureID;   //ID de la textura

	videoSetMode(MODE_0_3D);   //Modo 3D en el background 0

	glInit();      //Inicializamos OpenGL

	glEnable(GL_TEXTURE_2D);   //Permitimos el uso de texturas

	glViewport(0,0,255,191);   //Especificamos el tamaño de ventana (el de la
                               //pantalla de la NintendoDS)
	glClearColor(0,0,0,31);    //Especificamos el color de fondo
	glClearPolyID(63);
	glClearDepth(0x7FFF);

	vramSetBankA(VRAM_A_TEXTURE); //Almacenaremos la textura en el banco A
                                  //de la VRAM
	//Especificamos la textura, su tamaño y su ID
	glGenTextures(1, &textureID);
	glBindTexture(0, textureID);
	glTexImage2D(0,0,GL_RGB, TEXTURE_SIZE_256 , TEXTURE_SIZE_256,
	0, TEXGEN_TEXCOORD, (u8*)textura);

	//Especificamos los parámetros y posición de la cámara que representará la escena
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluPerspective(35, 256.0 / 192.0, 0.1, 40);
	gluLookAt(   0.0, 0.0, 1.0,
	             0.0, 0.0, 0.0,
	             0.0, 1.0, 0.0);
```

```
while (1){

            glPushMatrix();

                  //Alejamos el modelo de la cámara, una unidad, según el eje Z
                  glTranslate3f32(0, 0, floattof32(-1));

                  //Especificamos la opacidad (0-transparente, 31-opaco)
                  //del modelo, y le pedimos que no compute los vértices
                  //que no sean visibles por la cámara
                  glPolyFmt(POLY_ALPHA(31) | POLY_CULL_BACK) ;

                  //vinculamos la textura con el modelo que se cargará en la
                  //siguiente instrucción
                  glBindTexture(0, textureID);

                  //cargamos el modelo
                  glCallList((u32*)modelo);

                  glEnd();

            glPopMatrix(1);

            //Limpiamos la pantalla
            glFlush(0);

            PA_WaitForVBL();
      }

      return 0;
}
```

*Fig. 26*: Código que muestra por pantalla un modelo con su correspondiente textura.

Nótese que, como se verá en el siguiente punto, las transformaciones realizadas entre las sentencias **glPushMatrix()** y **glPopMatrix(1)** afectan únicamente al modelo que se haya cargado, mediante la llamada **glCallList()**, entre dichas dos sentencias.

Una vez visto el código básico de representación de objetos tridimensionales, el resto de funcionalidades se basa en añadir el código necesario para el control de dicha representación a lo largo de la ejecución, haciendo uso de las funciones que nos proporcionan las bibliotecas que hayamos incluido.

# ANIMACIÓN DE LOS MODELOS

En aquellos elementos animados cuya animación estaba basada únicamente en una transformación de movimiento o de escala, ésta se generó por código, mediante instrucciones de *OpenGL*, de forma similar a la vista en el punto anterior, tal como se muestra en la *figura 27*:

```
...
int posX=0, posY=0;

while (1){
            glPushMatrix();

                  glTranslate3f32(posX, 0, floattof32(-1));
                  glPolyFmt(POLY_ALPHA(31) | POLY_CULL_BACK) ;
                  glBindTexture(0, textureID[0]);
                  glCallList((u32*)modelo00);
                  glEnd();

            glPopMatrix(1);

            glPushMatrix();

                  glTranslate3f32(0, posY, floattof32(-1));
                  glPolyFmt(POLY_ALPHA(31) | POLY_CULL_BACK) ;
                  glBindTexture(1, textureID[1]);
                  glCallList((u32*)modelo01);
                  glEnd();

            glPopMatrix(1);

            glFlush(0);
            PA_WaitForVBL();

            posX++; posY++;

      }
...
```

*Fig. 27*: Fragmento de código que mostraría el desplazamiento del *modelo00* a lo largo del eje X y del *modelo01* a lo largo del eje Y.

Sin embargo, cuando la animación de un modelo implicaba además un cambio en la posición relativa de sus vértices (como por ejemplo el ciclo de correr del protagonista, *figura 28*), se hizo necesario generar dicha animación dentro del paquete de modelado e importarla posteriormente, fotograma a fotograma.

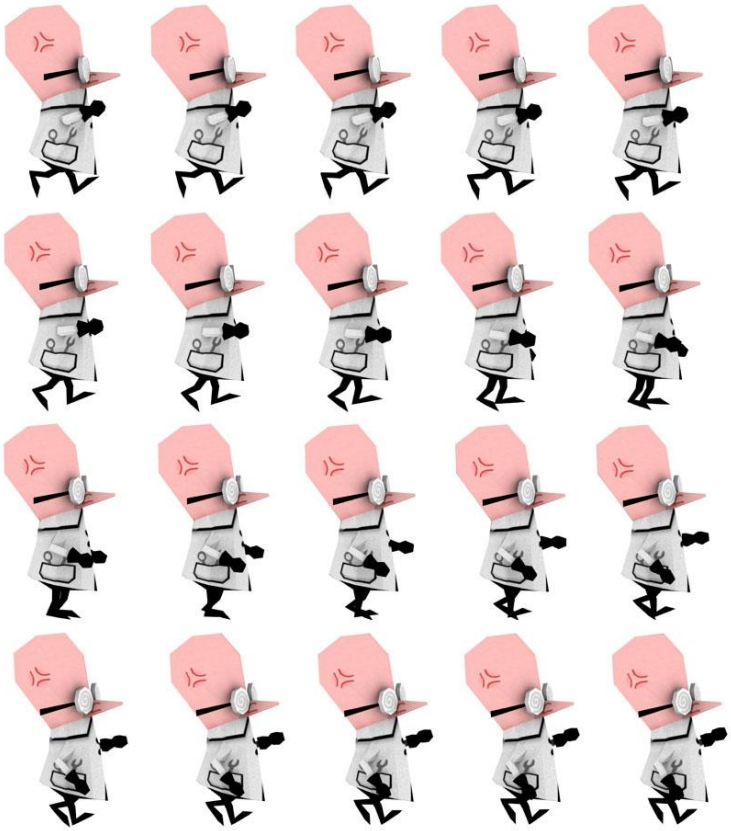


*Fig. 28*: Animación de 20 fotogramas que reproducidos alternativamente en sentidos contrarios (del 1 al 20, del 20 al 1, del 1 al 20, del 20 al 1...) provocan la sensación de carrera del protagonista.

# CARGADOR DE *.OBJ's

## FORMATO *OBJ*

Recordemos una vez más la importancia de que nuestra aplicación posea la funcionalidad de cargar de forma externa las distintas figuras en las que se basará el test, dotando así a la aplicación de gran versatilidad y de un tiempo de vida virtualmente ilimitado.

*OBJ* es un formato de archivo utilizado para la definición espacial de modelos tridimensionales. En un inicio fue desarrollado por *Wavefront Technologies* para ser usado en aplicaciones propias, pero debido a ser un formato abierto, fue adoptado también por otras muchas aplicaciones 3D, siendo hoy en día un formato universalmente adoptado.

La elección de dicho formato para nuestras figuras problema viene dado por el hecho de que un **.obj* es en realidad un archivo de texto que contiene la descripción de una superficie y que permite, por tanto, la lectura y edición desde un simple editor de texto. En la *figura 29* vemos el contenido de un archivo **.obj*, que contiene la geometría de un cubo creado en *3DSMAX*, abierto desde el *bloc de notas*:

```
# object cubo

#Listado de los 8 vértices del cubo

v  -12.5000 -12.5000 -12.5000
v  -12.5000 12.5000 -12.5000
v  12.5000 12.5000 -12.5000
v  12.5000 -12.5000 -12.5000
v  -12.5000 -12.5000 12.5000
v  12.5000 -12.5000 12.5000
v  12.5000 12.5000 12.5000
v  -12.5000 12.5000 12.5000
# 8 vertices
```

*#Listado de las normales de las 6 caras del cubo*

```
vn 0.0000 0.0000 -1.0000
vn 0.0000 -0.0000 1.0000
vn 0.0000 -1.0000 -0.0000
vn 1.0000 0.0000 -0.0000
vn 0.0000 1.0000 0.0000
vn -1.0000 0.0000 -0.0000
# 6 vertex normals
```

*#Listado de las coordenadas de texturización del cubo*

```
vt 1.0000 0.0000 0.0000
vt 1.0000 1.0000 0.0000
vt 0.0000 1.0000 0.0000
vt 0.0000 0.0000 0.0000
# 4 texture coords
```

*#Listado con la relación de vértices, normales y coordenadas #de texturización asociadas a*
*# cada cara del cubo*

```
g cubo
f 1/1/1 2/2/1 3/3/1 4/4/1
f 5/4/2 6/1/2 7/2/2 8/3/2
f 1/4/3 4/1/3 6/2/3 5/3/3
f 4/4/4 3/1/4 7/2/4 6/3/4
f 3/4/5 2/1/5 8/2/5 7/3/5
f 2/4/6 1/1/6 5/2/6 8/3/6
# 6 polygons
```

*#Siendo, cada tripleta, (***vértice/coord_de_text/normal***).*
*#Así, por ejemplo, la última cara (***f 2/4/6 1/1/6 5/2/6 0/3/6***) está formada por los vértices (***2,1,5,0***) con las coordenadas de texturización (***4,1,2,3***) asociadas y la normal de dicha cara es la* ***6****.*

*Fig. 29*: Contenido del archivo cubo.obj.

# CARGADOR DE **.OBJ's*

Así pues, nuestro cargador de **obj's* consistirá básicamente en una aplicación que deberá:

*1)* Localizar el archivo deseado en el sistema de archivos del *flashcart*.
*2)* Abrirlo como un archivo de texto.
*3)* Leer sus vértices y normales uno a uno, representando en pantalla las distintas caras que dichos vértices forman.

En la *figura 30* se muestra el código de la función implementada para conseguir dicha funcionalidad:

```
...
//Necesario utilizar las LibFAT para acceder al sistema de archivos
#include <fat.h>
...

...
//matrices donde almacenaremos los vértices, las coordenadas de texturización y las caras
//(límite máximo: objeto de 250 vértices o 100 caras)
float v[249][3], vn[249][3];
int faces[99][4][2];
int numfaces;
//Nombre de la figura a cargar
char figura[20];
...

void leer_figura(){

    //Abrimos la figura como si fuera un fichero de texto
    FILE* f = fopen (figura, "rb");

    char oneline[255];
    char primera[255];

    int i, numverts=0, numnorms=0;
    int v1, vn1, v2, vn2, v3, vn3, v4, vn4;
    float xf, yf, zf;

    numfaces=0;
```

```
  while (!feof(f)){

    fgets(oneline,255,f);
    sscanf(oneline, "%s ", primera);


     if(strcmp(primera, "v")==0){
      sscanf(oneline, "v %f %f %f", &xf, &yf, &zf);
      v[numverts][0]=xf; v[numverts][1]=yf; v[numverts][2]=zf;
      numverts++;
     }


     if(strcmp(primera, "vn")==0){
      sscanf(oneline, "vn %f %f %f", &xf, &yf, &zf);
      vn[numnorms][0]=xf; vn[numnorms][1]=yf; vn[numnorms][2]=zf;
      numnorms++;
     }


     if(strcmp(primera, "f")==0){
      sscanf(oneline, "f %d//%d %d//%d %d//%d %d//%d",
            &v1, &vn1, &v2, &vn2, &v3, &vn3, &v4, &vn4);
      faces[numfaces][0][0]=v1; faces[numfaces][0][1]=vn1;
      faces[numfaces][1][0]=v2; faces[numfaces][1][1]=vn2;
      faces[numfaces][2][0]=v3; faces[numfaces][2][1]=vn3;
      faces[numfaces][3][0]=v4; faces[numfaces][3][1]=vn4;
      numfaces++;
     }

    }//del while
      fclose(f);
}
```

*Fig. 30*: Código de la función encargada de cargar los *.obj externos al ejecutable.

Nótese que en ningún momento se ha almacenado la información relativa a la texturización de las figuras. Esto permite un ahorro considerable de memoria además de evitar al usuario la tarea de texturizar las figuras creadas por él (*figura 31*).

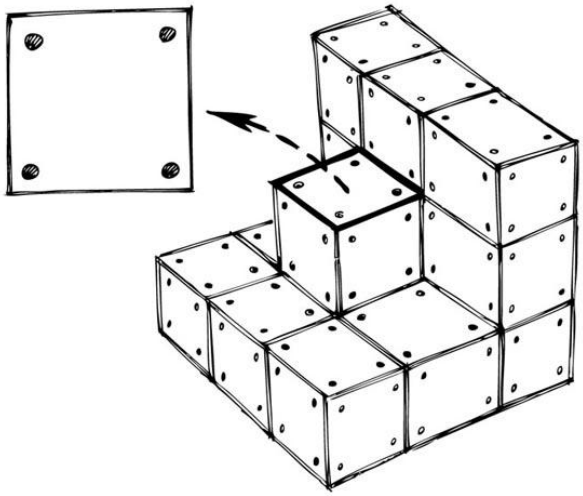


*Fig. 31*: La aplicación texturiza automáticamente la figura importada, aplicándole a cada cara una textura predefinida y aprovechando así una vez más las ventajas ofrecidas por el uso de *tiles*.

Para mostrar por pantalla la figura importada, basta con utilizar la información adquirida por la función **`leer_figura()`** para reconstruir el modelo mediante las funciones de *OpenGL* necesarias, tal como se describe en la *figura 32*

```
...
//asociamos la textura predefinida al quad a dibujar en pantalla
glBindTexture(0, textureID);

//dibujamos las numfaces caras
for(i=0; i<numfaces; i++){

        //Dibujamos caras de 4 vértices
        glBegin(GL_QUADS);

        //La normal a dicha cara
        glNormal3f(floattov16(vn[faces[i][0][2]-1][0]),
        floattov16(vn[faces[i][0][2]-1][1]),
        floattov16(vn[faces[i][0][2]-1][2]));
```

```
        //1er vértice del quad
        //le asignamos unas coordenadas de texturización
        GFX_TEX_COORD = (TEXTURE_PACK(0, inttot16(64)));
        //y especificamos el vértice propiamente dicho
        glVertex3v16(floattov16(v[faces[i][0][0]-1][0]),
        floattov16(v[faces[i][0][0]-1][1]),
        floattov16(v[faces[i][0][0]-1][2]));

        //2do vértice del quad
        //le asignamos unas coordenadas de texturización
        GFX_TEX_COORD = (TEXTURE_PACK(inttot16(64),inttot16(64)));
        //y especificamos el vértice propiamente dicho
        glVertex3v16(floattov16(v[faces[i][1][0]-1][0]),
        floattov16(v[faces[i][1][0]-1][1]),
        floattov16(v[faces[i][1][0]-1][2]));

        //3er vértice del quad
        //le asignamos unas coordenadas de texturización
        GFX_TEX_COORD = (TEXTURE_PACK(inttot16(64), 0));
        //y especificamos el vértice propiamente dicho
        glVertex3v16(floattov16(v[faces[i][2][0]-1][0]),
        floattov16(v[faces[i][2][0]-1][1]),
        floattov16(v[faces[i][2][0]-1][2]));

        //4to vértice del quad
        //le asignamos unas coordenadas de texturización
        GFX_TEX_COORD = (TEXTURE_PACK(0, 0));
        //y especificamos el vértice propiamente dicho
        glVertex3v16(floattov16(v[faces[i][3][0]-1][0]),
        floattov16(v[faces[i][3][0]-1][1]),
        floattov16(v[faces[i][3][0]-1][2]));
}
glEnd();
...
```

*Fig. 32*: Código que muestra por pantalla la figura cargada.

Como se ha podido ver en el código, y como se explicó en el punto *Box Modeling y Quads* de la presente memoria, la aplicación únicamente muestra figuras formadas íntegramente por *quads*, y así ha de tenerlo en cuenta el usuario a la hora de crear y exportar sus propias figuras (ver *figura 33*). Además, al exportar la figura, también deberá especificar que no es necesario que el *.obj contenga información relativa a las coordenadas de texturización.

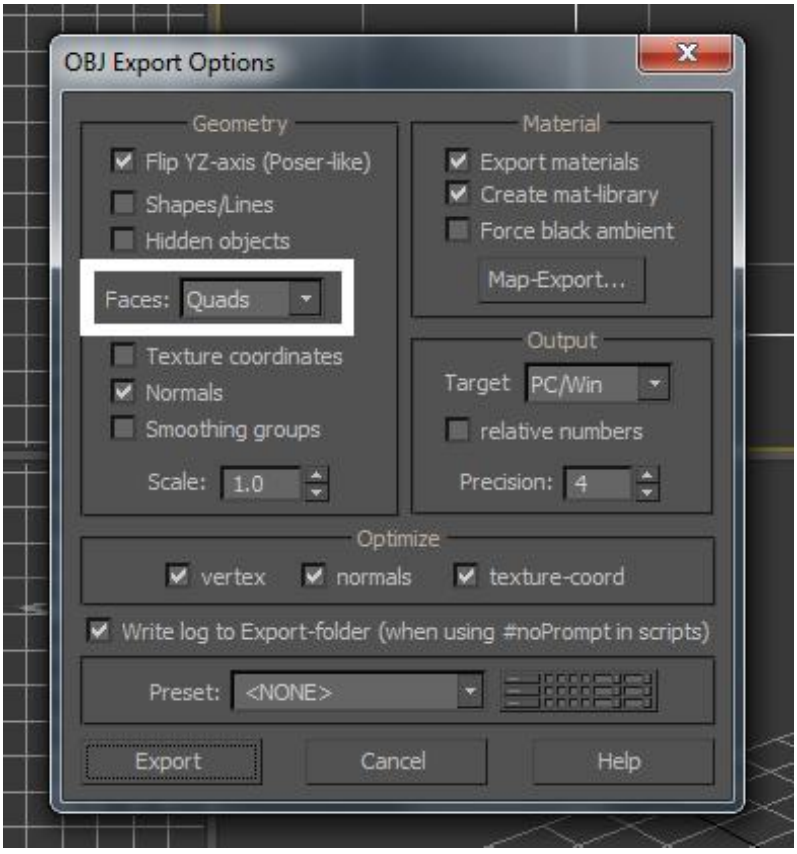



*Fig. 33*: Imagen del exportador de OBJ's incluido en 3DSMAX, donde se especifica que el *.obj creado esté formado únicamente por *quads*.

El usuario deberá tener en cuenta la escala e intentar que las figuras creadas tengan un tamaño similar a la figuras ejemplo que acompañan la aplicación (contenidas en un cubo de aproximadamente 1x1x1 unidades). Del mismo modo, deberá centrar sus figuras en el origen de coordenadas de la aplicación donde las modeló, pues las mismas rotarán alrededor de este punto.

Finalmente, debe tenerse muy en cuenta que distintos paquetes 3D pueden generar archivos **.obj* con una sintaxis ligeramente distinta, lo cual podría afectar a nuestro cargador de *OBJs*. Sin embargo, éste fue programado de forma que su funcionamiento se basase en la sintaxis común a los paquetes 3D a los que se ha tenido acceso. Así pues, también pueden utilizarse paquetes como *Blender* (interesante alternativa *open source* actualmente muy extendida) para la creación y exportación de las figuras problema (ver *figura 34*).

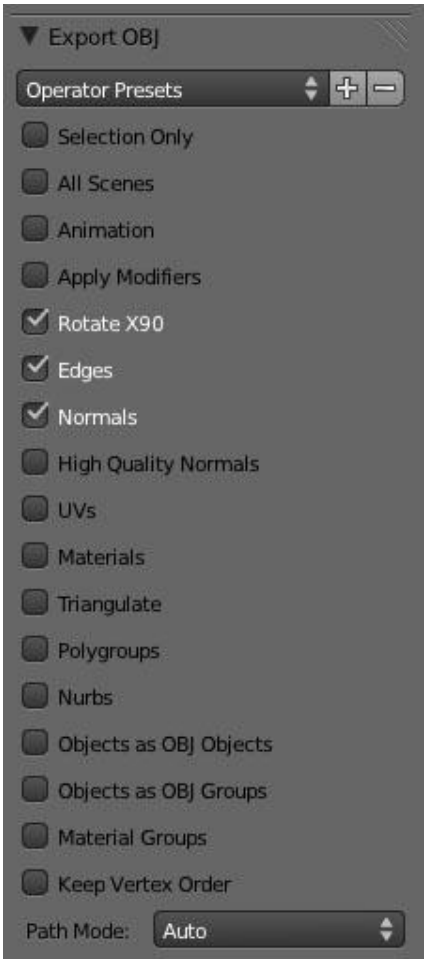



*Fig. 34*: Parámetros utilizados en la exportación de *OBJs*, en Blender .

# SISTEMA DE RESPUESTA

Recordemos una vez más que una de las funcionalidades más importantes de la presente aplicación es la posibilidad de que el docente pueda, sin tener conocimientos de programación, personalizar el test que realizará a sus alumnos.

Para dicha personalización, basta con editar el archivo *test.xml* (situado dentro del *flashcart* en la misma ruta que la *rom* de la aplicación y las figuras problema) cuyo formato es el mostrado en la *figura 35*:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<test>

<numpreguntas>6</numpreguntas>

<pregunta>
 <id>1</id>
 <enunciado>alzado </enunciado>
 <figura>figura001.obj </figura>
 <aristas>1 1 1 0 1 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 0 1</aristas>
 <tiempo>30</tiempo>
 <dificultad>1</dificultad>
</pregunta>

<pregunta>
 <id>2</id>
 <enunciado>planta </enunciado>
 <figura>figura001.obj </figura>
 <aristas>1 1 1 1 1 1 0 1 0 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 0 1</aristas>
 <tiempo>30</tiempo>
 <dificultad>1</dificultad>
</pregunta>
      ...

</test>
```

*Fig. 35*: Sintaxis del XML que contiene los enunciados y las soluciones.

Por otro lado, se pretende que el alumno dibuje con el *stylus* la vista de la figura indicada (*figura 36*), pues sólo entonces es seguro que ha adquirido las competencias deseadas en visión espacial.

*Fig. 36*: Sintaxis del XML que contiene los enunciados y las soluciones.

Se hace necesario, por tanto, un sistema que permita la expresión del dibujo de la vista preguntada en un formato entendible por nuestra aplicación (*figura 37*) y plasmable en el fichero *xml:*

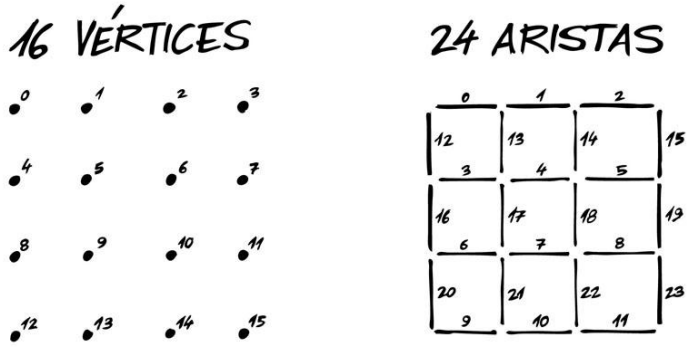



*Fig. 37*: Identificación de los posibles vértices y aristas que forman la vista solución.

El alumno deberá unir los vértices entre sí, dibujando un subconjunto de las 24 aristas indicadas arriba. Guardaremos un **1** en la posición **n** de un vector de enteros de tamaño 24, llamado **aristas[]** si el alumno ha dibujado la arista **n**. Del mismo modo, guardaremos un **1** en la posición **n** de otro vector de enteros del mismo tamaño (**solucion[]**) si la arista **n** forma parte de la solución (vector que el profesor especificará en el *xml* entre los *tags* **<aristas>** y **</aristas>**). La corrección de la pregunta consistirá básicamente en comparar ambos vectores:

```
bool corregir(){
	int i;
	for(i=0; i<24; i++) if(solucion[i]!=aristas[i]) return false;
	return true;
}
```

De este modo, y atendiendo al *xml* de ejemplo que se muestra al principio de este capítulo, la pregunta con id **1** consistirá en dibujar el alzado de la **figura001** en un tiempo inferior a **30** segundos. La dificultad estimada del ejercicio es de **1** y la solución correcta corresponde a la mostrada en la *figura 38*:

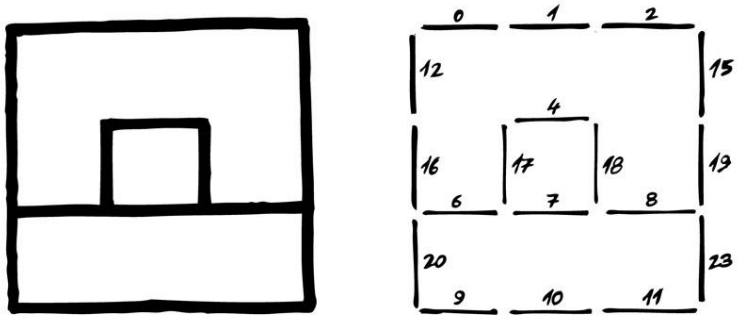



*Fig. 38:* `<aristas>1 1 1 1 1 1 0 1 0 1 1 1 1 0 0 1 1 1 1 1 1 0 0 1</aristas>`
Recordemos que dentro del atributo aristas, un **1** indica que la arista situada en dicha posición (de la 0 a la 23) forma parte de la solución.

Durante la ejecución de la aplicación, ésta deberá leer los atributos de cada pregunta de una forma similar a la utilizada para importar las figuras en el capítulo anterior, tal como se muestra en la *figura 39*:

```
#include <PA9.h> //Necesario utilizar las PAlib para generar el número aleatorio  y
                                                    //utilizar las funciones de sprites
#include <Fat.h> //Necesario utilizar las LibFAT para acceder al sistema de archivos

...
//Guardaremos en solución[] el vector especificado dentro del atributo aristas del xml
int aristas[24], solucion[24], preguntadas[99];  numpreguntas;
tiempo=0, id=0, dificultad=0;
char enunciado[255];
...
```

```
//leemos el test y almacenamos los atributos de la pregunta en las variables de arriba
void leer_test(){

	char linea[255];
	int i;
	int aleatorio=0;
	FILE* f = fopen ("test.xml", "rb");

	fgets(linea,255,f); //la cabecera
	fgets(linea,255,f); //la línea en blanco
	fgets(linea,255,f); //<test>
	fgets(linea,255,f); //la línea en blanco
	fgets(linea,255,f); //leemos el número de preguntas
	sscanf(linea, "<numpreguntas>%d</numpreguntas>",
	&numpreguntas);

	//nos aseguramos de que el orden de las preguntas sea aleatorio y de que ninguna
	//pregunta salga más de una vez
	do{
	aleatorio=PA_RandMinMax(1,numpreguntas);
	}while(preguntadas[aleatorio-1]==1);

	preguntadas[aleatorio-1]=1;

	//para cada pregunta...
	for(i=0; i<aleatorio; i++){
		fgets(linea,255,f); // la línea en blanco
		fgets(linea,255,f); //leemos <pregunta>
		fgets(linea,255,f); //leemos <id>
		sscanf(linea, "    <id>%d</id>", &id);
		fgets(linea,255,f); //leemos <enunciado>
		sscanf(linea, "    <enunciado> %s </enunciado>",
		enunciado);
		fgets(linea,255,f); //leemos <figura>
		sscanf(linea, "    <figura> %s </figura>", figura);
		fgets(linea,255,f); //leemos <aristas>
```

```
            sscanf(linea, "    <aristas>%d %d %d %d %d %d %d %d %d
            %d %d %d %d %d %d %d %d %d %d %d %d %d %d d</aristas>",
            &solucion[0], &solucion[1], &solucion[2], &solucion[3],
            &solucion[4], &solucion[5], &solucion[6], &solucion[7],
            &solucion[8], &solucion[9], &solucion[10],
            &solucion[11], &solucion[12], &solucion[13],
            &solucion[14], &solucion[15], &solucion[16],
            &solucion[17], &solucion[18], &solucion[19],
            &solucion[20], &solucion[21], &solucion[22],
            &solucion[23]);
            fgets(linea,255,f); //leemos <tiempo>
            sscanf(linea, "    <tiempo>%d</tiempo>", &tiempo);
            fgets(linea,255,f); //leemos <dificultad>
            sscanf(linea, "    <dificultad>%d</dificultad>",
            &dificultad);
            fgets(linea,255,f); //leemos </pregunta>
      }
      fclose(f);
}
```

*Fig. 39*: Código de la función encargada de leer el *xml* con los enunciados y las soluciones.

Para saber qué aristas está dibujando en cada momento el alumno, se crean al iniciar la aplicación 16 *sprites* circulares (correspondientes a los 16 vértices ya nombrados en este punto) y se detecta cuáles está uniendo el alumno con el *stylus* mediante el código mostrado en la *figura 40*:

```
int main(){
...
while(1){
...
PA_8bitDraw(1, 1); //Dibujamos el paso del stylus sobre la pantalla...

if(Stylus.Released){  //Al levantar el stylus de la pantalla...
      PA_Clear8bitBg(1); //borramos el paso del stylus por la pantalla
      tocado=false; //Aún no hemos tocado el 1er punto de los 2 necesarios para
      p=-1;                                          //dibujar una arista
      p1=0; p2=0;
      Stylus.X=0; Stylus.Y=0;
```

```
}else{
	p=punto_pulsado();
	if(p!=-1){ //Si el stylus está sobre un punto...
			if(!tocado){ //...y es el primer punto tocado
				p1=p;
				tocado=true;
				x1=PA_GetSpriteX(1,p);
				y1=PA_GetSpriteY(1,p);
			}
		//...si es el segundo punto pulsado, y es distinto al anterior, dibujamos la
		//arista y el nuevo punto pulsado pasa a ser el último punto pulsado
		if((PA_GetSpriteX(1,p)!=x1)||PA_GetSpriteY(1,p)!=y1){
				PA_Clear8bitBg(1);
				p2=p;
				dibuja_arista(p1,p2);
				p1=p2;
				x1=PA_GetSpriteX(1,p);
				y1=PA_GetSpriteY(1,p);
		}
	}
}
...
} //del bucle principal
...
} //del main
```

*Fig. 40*: Código que detecta qué vértices está uniendo el jugador con el *stylus*.

Dicho código hace uso de las siguientes funciones (*figura 41*), además de las proporcionadas por las librerías *PAlib* para el tratamiento de *sprites*:

```
//detectamos el punto pulsado por el stylus y devolvemos -1 cuando no se pulsa ninguno
int punto_pulsado(){
	int i=0;
	for(i=0; i<16; i++){
		if (PA_SpriteTouchedPix(i)) return i;
	}
	return -1;
}
```

```
//metemos un 1 en la posición correspondiente de aristas[], a partir de los 2 puntos unidos
void dibuja_arista(int p1, int p2){
//horizontales...
if(((p1==0)&&(p2==1))||((p1==1)&&(p2==0))) aristas[0]=1;
if(((p1==1)&&(p2==2))||((p1==2)&&(p2==1))) aristas[1]=1;
if(((p1==2)&&(p2==3))||((p1==3)&&(p2==2))) aristas[2]=1;

if(((p1==4)&&(p2==5))||((p1==5)&&(p2==4))) aristas[3]=1;
if(((p1==5)&&(p2==6))||((p1==6)&&(p2==5))) aristas[4]=1;
if(((p1==6)&&(p2==7))||((p1==7)&&(p2==6))) aristas[5]=1;

if(((p1==8)&&(p2==9))||((p1==9)&&(p2==8))) aristas[6]=1;
if(((p1==9)&&(p2==10))||((p1==10)&&(p2==9))) aristas[7]=1;
if(((p1==10)&&(p2==11))||((p1==11)&&(p2==10))) aristas[8]=1;

if(((p1==12)&&(p2==13))||((p1==13)&&(p2==12))) aristas[9]=1;
if(((p1==13)&&(p2==14))||((p1==14)&&(p2==13))) aristas[10]=1;
if(((p1==14)&&(p2==15))||((p1==15)&&(p2==14))) aristas[11]=1;

//verticales
if(((p1==0)&&(p2==4))||((p1==4)&&(p2==0))) aristas[12]=1;
if(((p1==1)&&(p2==5))||((p1==5)&&(p2==1))) aristas[13]=1;
if(((p1==2)&&(p2==6))||((p1==6)&&(p2==2))) aristas[14]=1;
if(((p1==3)&&(p2==7))||((p1==7)&&(p2==3))) aristas[15]=1;

if(((p1==4)&&(p2==8))||((p1==8)&&(p2==4))) aristas[16]=1;
if(((p1==5)&&(p2==9))||((p1==9)&&(p2==5))) aristas[17]=1;
if(((p1==6)&&(p2==10))||((p1==10)&&(p2==6))) aristas[18]=1;
if(((p1==7)&&(p2==11))||((p1==11)&&(p2==7))) aristas[19]=1;

if(((p1==8)&&(p2==12))||((p1==12)&&(p2==8))) aristas[20]=1;
if(((p1==9)&&(p2==13))||((p1==13)&&(p2==9))) aristas[21]=1;
if(((p1==10)&&(p2==14))||((p1==14)&&(p2==10))) aristas[22]=1;
if(((p1==11)&&(p2==15))||((p1==15)&&(p2==11))) aristas[23]=1;
}
```

*Fig. 41*: Funciones escritas para su utilización en el código que se encarga de dibujar las aristas.

Finalmente, deberá llamarse en cada ejecución del bucle principal a alguna función que muestre por pantalla las aristas dibujadas (almacenadas en el vector **aristas[ ]**) tal como se muestra en la *figura 42*:

```
void muestra_aristas(){

//sumamos 4 a cada coordenada para que dibuje la línea desde el centro del sprite (de 8x8 píxeles), ya que las funciones PA_GetSpriteX() y PA_GetSpriteY() devuelven la posición de la esquina superior izquierda del mismo.
  if(aristas[0]==1){
    PA_Draw8bitLineEx(1, PA_GetSpriteX(1,0)+4,
    PA_GetSpriteY(1,0)+4, PA_GetSpriteX(1,1)+4,
    PA_GetSpriteY(1,1)+4, 1 /* color de la línea */, 2 /* grosor de la línea */);
  }

  if(aristas[1]==1){
    PA_Draw8bitLineEx(1, PA_GetSpriteX(1,1)+4,
    PA_GetSpriteY(1,1)+4, PA_GetSpriteX(1,2)+4,
    PA_GetSpriteY(1,2)+4, 1, 2);
  }
...
}
```

*Fig. 42*: Función que muestra las aristas dibujadas hasta el momento por pantalla.

# OTRAS FUNCIONES UTILIZADAS

Hasta el momento se han presentado los tres principales módulos que componen la aplicación y que mayor tiempo de trabajo han consumido. Sin embargo, ésta no estaría completa sin otras muchas funcionalidades cuya implementación ha resultado menos costosa gracias la utilización de funciones de alto nivel proporcionadas por las siguientes librerías:

## PALIB

***Mostrar texto por pantalla***

```
void PA_OutputText(
        screen,  //Pantalla inferior o superior (0 o 1).
        x,  //Coordenada X (0-31) del inicio del texto.
        y,  //Coordenada X (0-31) del inicio del texto.
        text,  //String a mostrar por pantalla.
        ...   )  //Para mostrar variables dentro del String, usar: %s para otros
                   strings, %d para enteros y %fX para floats con X dígitos.
```

*Ejemplo:* PA_OutputText(0,2,1,1,"Me llamo %s y peso %d kilos", nombre, peso);

***Mostrar fondos***

```
PA_LoadBackground(
        screen,  //Pantalla en la que mostrar el fondo (0-inferior, 1-superior).
        priority,  //Prioridad (capa) del fondo (0 - 3).
        background_name)  //Fondo a mostrar.
```

*Ejemplo:* PA_LoadBackground(1, 2, &pantalla_acierto);

### *Variar el brillo de la pantalla*

Las distintas transiciones entre pantallas se programaron variando el brillo de cada una de ellas.

```
void PA_SetBrightness(
        screen, //Pantalla inferior o superior (0 o 1).
        bright) //Nivel de brillo, desde -32 hasta 32, siendo el 0 neutral.
```

### *Acceso al reloj del sistema*

Puede accederse en todo momento al reloj del sistema (por ejemplo para calcular un incremento de tiempo transcurrido) consultando las siguientes variables:

**PA_RTC.Day** *(día)*
**PA_RTC.Month** (mes)
**PA_RTC.Year** *(año)*
**PA_RTC.Hour** *(hora)*
**PA_RTC.Minutes** *(minutos)*

### *Generación de números aleatorios*

Gracias a la siguiente función, puede generarse un entero aleatorio entre otros dos enteros dados (para que, por ejemplo, varíe el orden de las preguntas en cada partida).

```
PA_RandMinMax(
        min, //valor mínimo (inclusive)
        max) //valor máximo (inclusive)
```

### *Teclado alfanumérico*

PAlib incorpora un teclado alfanumérico (para que, por ejemplo, un jugador introduzca su nombre) que una vez iniciado puede mostrarse y retirarse mediante las siguientes funciones...

```
PA_KeyboardIn(
        x, // Coordenada X de la esquina superior izquierda del teclado.
        y) //Coordenada Y de la esquina superior izquierda del teclado.

PA_KeyboardOut(void)
```

...pudiendo consultar la tecla pulsada en cada momento mediante la función `PA_CheckKeyboard(void)`.

### *Manejo de sprites*

Para crear un *sprite* cuyo gráfico está contenido en la carpeta *gfx* de nuestro proyecto, utilizaremos la función...

```
PA_CreateSprite(
        screen, //Pantalla en la que mostrar el sprite.
        obj_number, //ID del sprite a mostrar.
        obj_data, //gráfico correspondiente a dicho sprite.
        obj_shape,//especificar el tamaño en píxeles.
        obj_size, //especificar el tamaño en píxeles.
        color_mode, //(0 - 256 colores, 1 - 16 colores).
        palette, //Paleta a  utilizar (0-15).
        x, // Coordenada X de la esquina superior izquierda del sprite.
        y) // Coordenada Y de la esquina superior izquierda del sprite.
```

...pudiendo detectar en todo momento si está siendo tocado por el stylus mediante la siguiente función:

```
PA_Sprite16cTouchedPix(
        sprite) //ID del sprite a detectar.
```

# LIBFAT

Una vez inicializadas las librerías al comienzo del *main* mediante **`fatInitDefault()`**, podemos acceder a los archivos externos al ejecutable con las funciones tradicionales de manejo de ficheros en *C*: **`fopen, fclose,...`**

# ASLIB

Para la reproducción de archivos de sonido en formato *mp3*, habrá que inicializar la librería con los siguientes parámetros...

```
PA_VBLFunctionInit(AS_SoundVBL);
AS_Init(AS_MODE_MP3); //especificamos el formato mp3.
AS_SetDefaultSettings(AS_PCM_8BIT, 11025, AS_SURROUND); //parámetros
                                                        //del archivo mp3.
```

...pudiendo comenzar la reproducción del archivo de sonido mediante las instrucciones...

```
AS_MP3DirectPlay((u8*)nombre_del_archivo, (u32)tamaño_del_archivo);
AS_SetSoundVolume(0,127); //canal 0, volumen al máximo (0-127).
AS_SetMP3Loop(true); //el archivo se reproduce cíclicamente, de forma indefinida.
```

...pudiendo reproducir además los efectos sonoros, en formato *raw*, cuando lo deseemos, mediante la orden:

```
AS_SoundQuickPlay(nombre_del_archivo_raw);
```

# CONCLUSIONES Y POSIBLES AMPLIACIONES

Una vez finalizada la aplicación, puede comprobarse con satisfacción que se han cumplido todos los objetivos recogidos en la introducción del presente proyecto. De este modo, se ha conseguido desarrollar una herramienta potente y versátil que, tal como se pretendía, resulta de gran utilidad en la educación de la visión espacial y adecuada para su uso en los centros docentes. Véanse (*figura 43*) algunas instantáneas de la aplicación en ejecución.

Además, durante el desarrollo del presente proyecto el autor ha tomado conciencia de las dificultades que entraña la programación de aplicaciones en una plataforma tan limitada como la *NintendoDS*, resultando crítica la optimización en prácticamente todos los aspectos.

Por otra parte, el trato continuo con el director del proyecto, y paralelamente, con los alumnos, ha resultado una experiencia altamente enriquecedora: del primero se obtuvo consejo , apoyo técnico e involucración en todo momento, y de los segundos, la sinceridad crítica, frescura de ideas y participación que sólo se obtiene de gente tan joven y que sin embargo todo usuario debería proporcionar.

Sin embargo, tras la programación de la aplicación, siempre surgen nuevas ideas que merecen formar parte de la misma. Resulta necesario, por tanto, recogerlas como futuras ampliaciones, como es el caso de la implementación de un *ranking* donde los alumnos puedan consultar las mejores marcas obtenidas hasta el momento por el resto del alumnado, fomentando así una vez más el afán de superación a través de una sana rivalidad entre los alumnos.


*Fig. 43*: Distintas imágenes que muestran la aplicación en ejecución.

# Nintendo3DS

Durante la elaboración del presente proyecto (concretamente el 25 de Marzo de 2011) *Nintendo* sacó al mercado la consola portátil sucesora de la *NintendoDS*: la *Nintendo3DS* (ver *figura 44*).

Pese a su gran parecido externo con la *NintendoDS* (incorpora también dos pantallas, una de ellas táctil) sus especificaciones técnicas indican una potencia mucho mayor: su *CPU* es un *ARM11* y su *GPU*, un *Pica200* (también con soporte *OpenGL*) que trabaja a 200MHz. La memoria principal es de 128MB y la de vídeo de 4MB. La pantallas, pese a ser de un tamaño similar a las de *NintendoDS*, tienen una resolución de 800(400 para cada ojo)x240 y de 320x240 píxeles respectivamente.


*Fig. 44*: Foto de la Nintendo3DS, de aspecto muy similar a los últimos modelos de *NintendoDS* (Imagen tomada de http://ac3ds.info).

Sin embargo, su principal atractivo se basa en la capacidad de representar imágenes 3D estereoscópicas sin necesidad de utilizar gafas: la pantalla superior se comporta de forma similar a un *estereograma*, dirigiendo la mitad de las líneas verticales hacia un ojo y la otra mitad hacia el otro, proyectando dos imágenes de una misma realidad que difieren levemente y que por tanto producen sensación de profundidad.

Con todo esto, resulta evidente que las futuras ampliaciones del presente proyecto deberán basarse en el desarrollo de la aplicación para esta nueva plataforma: Además de aprovechar un mayor potencial técnico (evitando muchas de las limitaciones impuestas por la *NintendoDS*) podrían visualizarse las figuras problema como figuras 3D estereoscópicas que parecen emerger de la pantalla.

Además, no sólo la *Nintendo3DS* garantiza retrocompatibilidad con los juegos de *NintendoDS*, sino que además es posible ejecutar en ella, desde el momento de su lanzamiento, *homebrew* desarrollado para *NintendoDS* mediante el uso de algunos *flashcarts*, como son los nombrados en la *figura 45*:

*EZ FLASH Vi*
*M3i ZERO*
*R4i-SDHC 3DS*
*R4LI*
*R4i GOLD*
*R4iTT*
*CycloDSI Evolution*
*Acerkard 2i*
*Supercard dstwo*
*SuperCard DSONEi*
*GEi*

*Fig. 45*: Listado comprobado de *flashcarts* compatibles hasta el momento con la *Nintendo3DS* según *www.elotrolado.net*

Desafortunadamente, al no tratarse de un simple filtro, no puede conseguirse el efecto 3D estereoscópico en modo DS. Así pues, se hace necesario (al menos de momento) el SDK oficial para Nintendo3DS que permita incluir dicho efecto en los juegos programados para tal fin. La alternativa consiste en esperar a que la *scene* de Nintendo3DS crezca. Sin embargo, esto puede demorarse indefinidamente puesto que aún hay características del modelo *DSi*, del cual ha tomado N3DS su sistema de seguridad (incluyendo mejoras), a las que aún no se ha podido acceder.

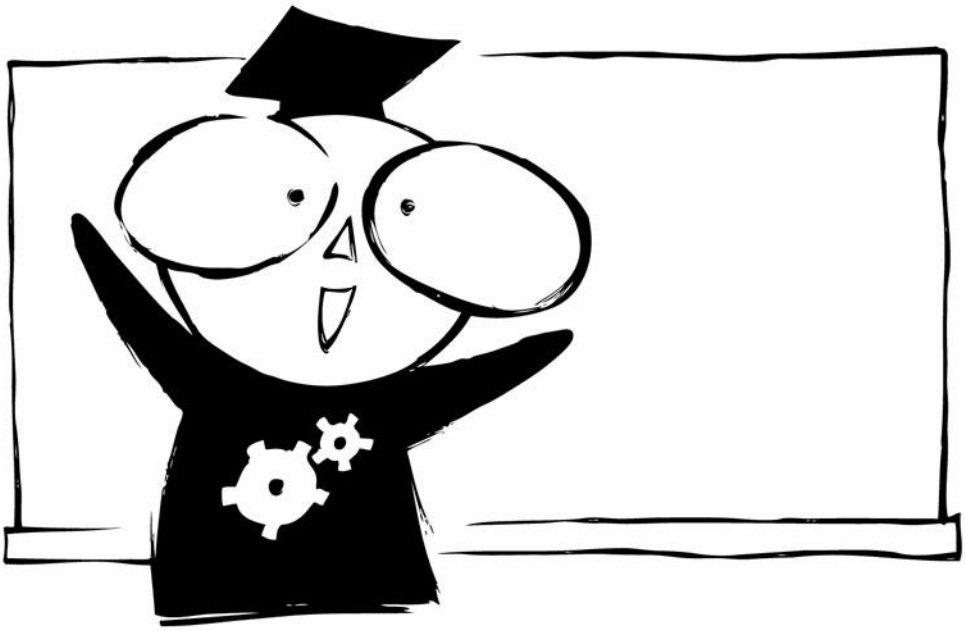
# BIBLIOGRAFÍA


- Documentación de las liberías Libnds (http://libnds.devkitpro.org).
- Documentación de las liberías PAlib (http://www.palib-dev.com/manual.html).
- Web de Chishm (http://chishm.drunkencoders.com).
- Web de NeHe (http://nehe.gamedev.net).
- Web de Noda (http://nodadev.wordpress.com).
- Foro de *El Otro Lado* (www.elotrolado.net).

# ANEXO I

## DESARROLLO DE UNA APLICACIÓN DIDÁCTICA PARA *NintendoDS* MEDIANTE EL USO DE LAS LIBRERÍAS *Ndslib* EN *DevKitPro.*

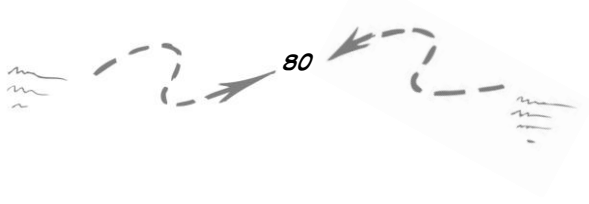
***Asignatura:*** **IPM.**
***Curso:*** **2010-2011.**
***Profesor:*** **D. Manuel Agustí.**



***Héctor Cuñat Núñez.***

# ÍNDICE





# AUTOR


- **Nombre:** Héctor Cuñat Núñez
- **Correo:** *heccuanu@topo.upv.es*
- **Asignatura:** Integración de Medios Digitales.
- **Titulación:** Ingeniería Técnica en Informática de Sistemas.


# RESUMEN


Aplicación didáctica para la videoconsola *NintendoDS* basada en la visualización y manipulación de distintos modelos tridimensionales, en la pantalla superior. Al mismo tiempo, se realizarán preguntas sobre los mismos en la pantalla inferior. La aplicación contestará, en cada caso, si la respuesta es correcta o no.


# OBJETIVOS

No todo el mundo posee una visión espacial adecuada. Existen casos en que los tres ejes de coordenadas que se dibujan en una pizarra no son vistos espacialmente, sino como una confluencia plana de tres caminos incidentes. El presente trabajo pretende ser una herramienta didáctica que ayude a una temprana educación de la visión tridimensional, necesaria para el desarrollo personal, tanto en la vida diaria como en campos académicos: geometría, arte, arquitectura. En este plano, los objetivos son los de construir:

- Una herramienta didáctica sobre una plataforma lúdica.
- Un material utilizable en las clases de Tecnología de la ESO.
- Unos contenidos formadores de la educación de la visión espacial del alumnado.

Esta contribución a la "didáctica del espacio", pretende además de resultar lúdica y atrayente ser lo más viable posible. Por ello, se ha desarrollado en la plataforma *NintendoDS*, ya que, tras encuestar al alumnado (a un grupo de alumnos de 1º y 2º de la ESO), ésa era la consola que la mayoría poseía.

En el plano técnico, la aplicación desarrollada debe conseguir:

- La visualización y manipulación en tiempo real de modelos 3D texturizados.
- Una interfaz atrayente y muy usable que:
  - Permita al usuario voltear los modelos cómodamente.
  - Facilite responder a las preguntas formuladas de modo intuitivo y sencillo.
  - Subraye de modo sonoro los fallos y los aciertos.

# REQUERIMIENTOS

- DevkitPro (*devkitARM r32*) con *libnds 1.4.8*.
- NintendoDS con cartucho para desarrollo y/o emulador.

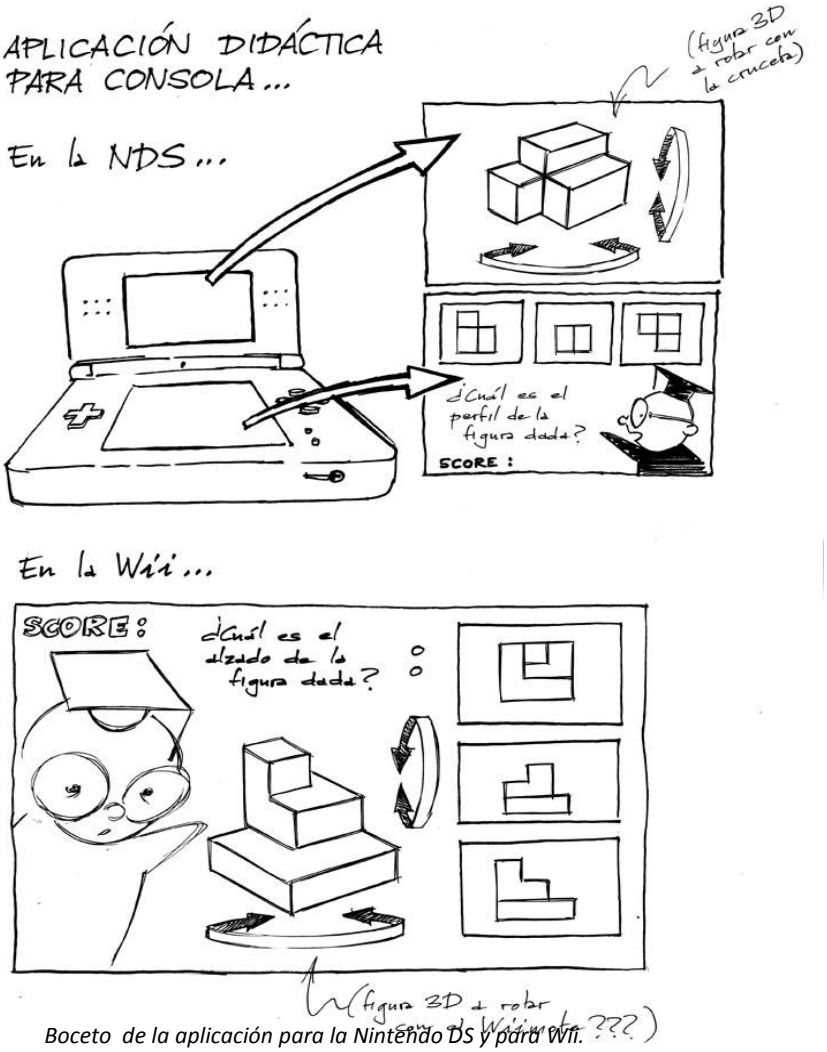
## Elección del dispositivo.

Debido a intereses profesionales –me dedico a la enseñanza-, desde muy temprano tuve la intención de realizar este proyecto para alguna de las videoconsolas del mercado, dada su popularidad entre el alumnado.

Pronto, el campo de las posibles consolas en las que podía desarrollar la investigación propuesta, se restringió a dos: la *NintendoDS* o la *Wii*, de *Nintendo*. Ambas eran muy populares y tenían la propiedad de que la interacción de la videoconsola con el usuario se realizaba de una forma novedosa, cómoda e intuitiva (ya fuera con el *stylus* o mediante el uso del *wiimote*).

Además, quería que mi trabajo tuviese una aplicación didáctica, que fuese una herramienta que facilitase mi trabajo como profesor de tecnología en enseñanza secundaria.

Con estas premisas, realicé un boceto sobre la posible interfaz de la aplicación para cada una de las plataformas elegidas (véase la figura anexa).




*Boceto de la aplicación para la Nintendo DS y para Wii.*

Como muestra el texto de los bocadillos, la aplicación pensada (al menos en un principio) para reforzar el ámbito del dibujo técnico, pretende que el alumno manipule un objeto tridimensional sobre el que se realizan algunas preguntas relativas a su alzado, planta o perfil.

Finalmente, consideré que dado el objetivo didáctico del proyecto, un factor clave en la decisión de la consola para la cual realizar el desarrollo debía ser la disponibilidad de la misma por parte del alumnado. Es preciso remarcar que no sólo es necesario que el alumno disponga de la videoconsola, sino también que esta esté preparada para ejecutar *homebrew*.

Así pues, realicé una encuesta al alumnado de secundaria del CEIP Eliseo Vidal de Valencia con los siguientes resultados:

| ***Total alumnos*** | ***Con NDS*** | ***NDS + cartucho carga backups*** | ***Wii*** | ***Wii pirateada*** |
|---|---|---|---|---|
| **56** | 48 | 44 | 18 | 12 |
| **100%** | 86% | 79% | 32% | 21% |

*Tabla estadística DS / Wii.*

Los resultados de la encuesta hicieron que me inclinara por la Nintendo DS, ya que, como puede verse en la encuesta, resultó que casi todos los alumnos poseían la DS y pocos la Wii. Además, el porcentaje de consolas en las que se podría ejecutar *homebrew* era mucho mayor en el caso de la *NintendoDS*.

Así pues, procedí a la instalación de las herramientas de desarrollo necesarias (*DevkitPro (devkitARM r32)* con *libnds 1.4.8.* http://devkitpro.org).

Con la finalidad de probar las distintas versiones de la aplicación, utilicé principalmente el emulador *NO$GBA 2.6* y puntualmente una *NintendoDS* (modelo *lite*) con el dispositivo de carga de *homebrew* y *backups, SuperCardSD*.

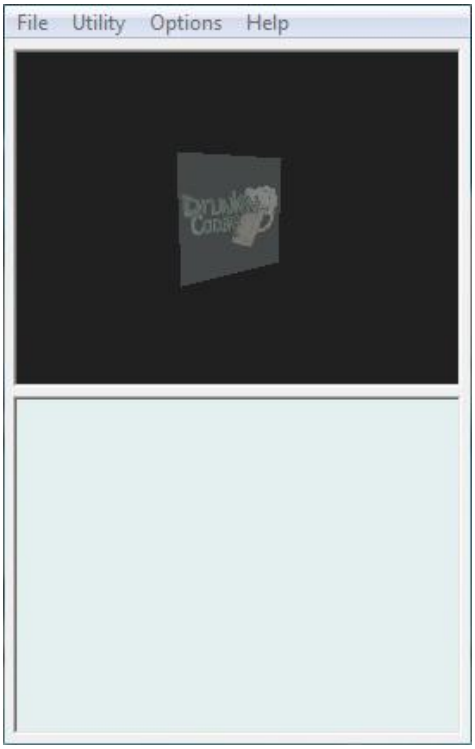
## Importación, manipulación y texturización de figuras 3D

Con el objeto de tomar contacto con la funcionalidad de representar modelos tridimensionales en la pantalla de la NintendoDS, partí del ejemplo que viene con *devkitpro* llamado *textured_quad*:

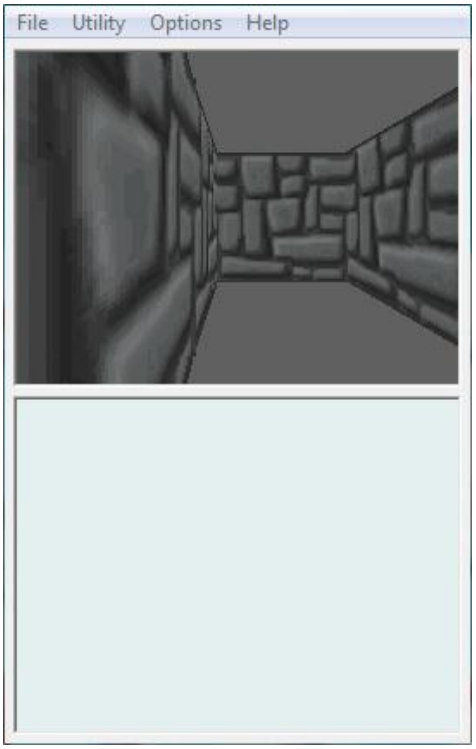
```
#include <nds.h>
#include <stdlib.h>

//texture_bin.h is created automagicaly from the texture.bin placed in arm9/resources
//texture.bin is a raw 128x128 16 bit image.  I will release a tool for texture conversion
//later
#include "texture_bin.h"

int main() {

    int textureID;

    float rotateX = 0.0;
    float rotateY = 0.0;

    //set mode 0, enable BG0 and set it to 3D
    videoSetMode(MODE_0_3D);

    // initialize gl
    glInit();

    //enable textures
    glEnable(GL_TEXTURE_2D);

    // enable antialiasing
    glEnable(GL_ANTIALIAS);

    // setup the rear plane
    glClearColor(0,0,0,31); // BG must be opaque for AA to work
    glClearPolyID(63); // BG must have a unique polygon ID for AA to work
    glClearDepth(0x7FFF);

    //this should work the same as the normal gl call
    glViewport(0,0,255,191);

    vramSetBankA(VRAM_A_TEXTURE);

    glGenTextures(1, &textureID);
    glBindTexture(0, textureID);
    glTexImage2D(0, 0, GL_RGB, TEXTURE_SIZE_128 , TEXTURE_SIZE_128, 0, TEXGEN_TEXCOORD, (u8*)texture_bin);


    //any floating point gl call is being converted to fixed prior to being implemented
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(70, 256.0 / 192.0, 0.1, 40);

    gluLookAt(  0.0, 0.0, 1.0,      //camera possition
                0.0, 0.0, 0.0,      //look at
                0.0, 1.0, 0.0);     //up
```

```
    while(1) {
        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();

        //move it away from the camera
        glTranslate3f32(0, 0, floattof32(-1));

        glRotateX(rotateX);
        glRotateY(rotateY);


        glMaterialf(GL_AMBIENT, RGB15(16,16,16));
        glMaterialf(GL_DIFFUSE, RGB15(16,16,16));
        glMaterialf(GL_SPECULAR, BIT(15) | RGB15(8,8,8));
        glMaterialf(GL_EMISSION, RGB15(16,16,16));

        //ds uses a table for shinyness..this generates a half-ass one
        glMaterialShinyness();

        //not a real gl function and will likely change
        glPolyFmt(POLY_ALPHA(31) | POLY_CULL_BACK);

        scanKeys();

        u16 keys = keysHeld();

        if((keys & KEY_UP)) rotateX += 3;
        if((keys & KEY_DOWN)) rotateX -= 3;
        if((keys & KEY_LEFT)) rotateY += 3;
        if((keys & KEY_RIGHT)) rotateY -= 3;

        glBindTexture(0, textureID);

        //draw the obj
        glBegin(GL_QUAD);
            glNormal(NORMAL_PACK(0,inttov10(-1),0));

            GFX_TEX_COORD = (TEXTURE_PACK(0, inttot16(128)));
            glVertex3v16(floattov16(-0.5), floattov16(-0.5), 0 );

            GFX_TEX_COORD = (TEXTURE_PACK(inttot16(128),inttot16(128)));
            glVertex3v16(floattov16(0.5),  floattov16(-0.5), 0 );

            GFX_TEX_COORD = (TEXTURE_PACK(inttot16(128), 0));
            glVertex3v16(floattov16(0.5),  floattov16(0.5), 0 );

            GFX_TEX_COORD = (TEXTURE_PACK(0,0));
            glVertex3v16(floattov16(-0.5), floattov16(0.5), 0 );

        glEnd();

        glPopMatrix(1);

        glFlush(0);

        swiWaitForVBlank();
    }

    return 0;
}//end main
```

De este ejemplo, y con la ayuda de la API de *ndslib* (http://libnds.devkitpro.org) comprendemos cómo implementar la funcionalidad de "pintar" en pantalla un *quad* y aplicarle una textura (una imagen raw de 16bits y *128x128px*). Obsérvese que también se puede ver la funcionalidad de leer los botones de la consola y voltear la pieza en función de cuál esté pulsado.




*Ejemplo textured_quad de devkitpro en ejecución.*




*Laberinto generado a partir del ejemplo anterior.*

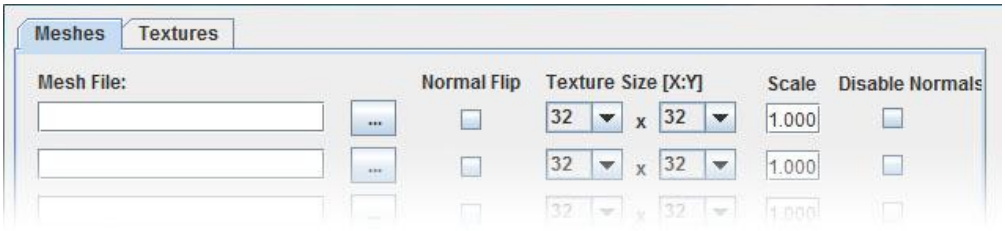
A partir de este código, y para familiarizarme con las funciones de las librerías *ndslib* vistas, construí con *quads* un laberinto similar al del *Wolfenstein3D* (http://es.wikipedia.org/wiki/Wolfenstein_3D). Además, importé una textura propia (para ello me fue necesario el uso del programa *bmp2bin*, que convierte el mapa de bits tratado en *Photoshop* al formato apropiado para la *NintendoDS*).

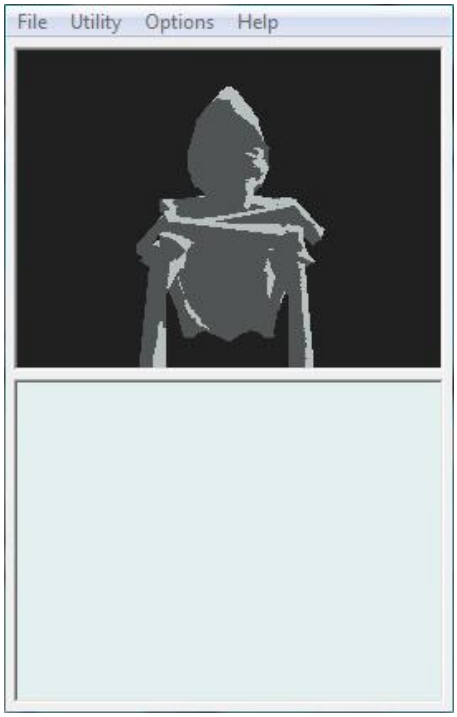
Sin embargo, la construcción de un modelo tridimensional especificando para cada *quad* las coordenadas de sus vértices resulta largo y tedioso. Por ello, Busqué una solución alternativa y descubrí que *devkitpro* incorpora otro ejemplo, llamado *toon_shading*, en el cual se importa un archivo que contiene la geometría de un modelo 3d ya creado. Además, incorpora una función (*get_pen_delta()*) que permite voltear el modelo desplazando el *stylus* sobre la pantalla táctil en lugar de presionando botones, facilidad que me pareció más usable e intuitiva.

```c
#include <nds.h>

//NB: This would look better if the object had a bit of texturing too (eyes, nose etc)
#include "statue_bin.h"

static void get_pen_delta( int *dx, int *dy )
{
    static int prev_pen[2] = { 0x7FFFFFFF, 0x7FFFFFFF };

    u32 keys = keysHeld();

    touchPosition touchXY;

    if( keys & KEY_TOUCH )
    {

        touchRead(&touchXY);
        if( prev_pen[0] != 0x7FFFFFFF )
        {
            *dx = (prev_pen[0] - touchXY.rawx);
            *dy = (prev_pen[1] - touchXY.rawy);
        }

        prev_pen[0] = touchXY.rawx;
        prev_pen[1] = touchXY.rawy;
    }
    else
    {
        prev_pen[0] = prev_pen[1] = 0x7FFFFFFF;
        *dx = *dy = 0;
    }
}

int main() {

    int rotateX = 0;
    int rotateY = 0;

    //set mode 0, enable BG0 and set it to 3D
    videoSetMode(MODE_0_3D);

    // initialize gl
    glInit();

    // enable antialiasing
    glEnable(GL_ANTIALIAS);

    // setup the rear plane
    glClearColor(0,0,0,31); // BG must be opaque for AA to work
    glClearPolyID(63); // BG must have a unique polygon ID for AA to work
    glClearDepth(0x7FFF);

    //this should work the same as the normal gl call
    glViewport(0,0,255,191);
    //toon-table entry 0 is for completely unlit pixels, going up to entry 31 for completely lit
    //We block-fill it in two halves, we get cartoony 2-tone lighting
    glSetToonTableRange( 0, 15, RGB15(8,8,8) );
    glSetToonTableRange( 16, 31, RGB15(24,24,24) );

    //any floating point gl call is being converted to fixed prior to being implemented
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(70, 256.0 / 192.0, 0.1, 40);

    //NB: When toon-shading, the hw ignores lights 2 and 3
    //Also note that the hw uses the RED component of the lit vertex to index the toon-table
    glLight(0, RGB15(16,16,16) , 0,        floattov10(-1.0),       0);
    glLight(1, RGB15(16,16,16),   floattov10(-1.0),    0,      0);

    gluLookAt(  0.0, 0.0, -3.0,     //camera possition
                0.0, 0.0, 0.0,      //look at
                0.0, 1.0, 0.0);     //up
```

```
    while(1)
    {

        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();
            glRotateXi(rotateX);
            glRotateYi(rotateY);

            glMaterialf(GL_AMBIENT, RGB15(8,8,8));
            glMaterialf(GL_DIFFUSE, RGB15(24,24,24));
            glMaterialf(GL_SPECULAR, RGB15(0,0,0));
            glMaterialf(GL_EMISSION, RGB15(0,0,0));

            glPolyFmt(POLY_ALPHA(31) | POLY_CULL_BACK | POLY_FORMAT_LIGHT0
                                     | POLY_FORMAT_LIGHT1 | POLY_TOON_HIGHLIGHT);

            scanKeys();
            u32 keys = keysHeld();

            if( keys & KEY_UP ) rotateX += 1;
            if( keys & KEY_DOWN ) rotateX -= 1;
            if( keys & KEY_LEFT ) rotateY += 1;
            if( keys & KEY_RIGHT ) rotateY -= 1;

            int pen_delta[2];
            get_pen_delta( &pen_delta[0], &pen_delta[1] );
            rotateY -= pen_delta[0];
            rotateX -= pen_delta[1];

            glCallList((u32*)statue_bin);
            glPopMatrix(1);

        glFlush(0);

        swiWaitForVBlank();
    }

    return 0;
}//end main
```

La primera experimentación con el código de ejemplo implicaba poder importar objetos propios. Para ello, se modeló y texturizó en *3DSMAX 2010* una figura sencilla que luego sería exportada en formato **.3ds* y convertida a **.bin*, formato legible por la DS, mediante el *'NDS Model Exporter'* de *PadrinatoR* (http://sites.google.com/site/lawebdeunfriki). La textura, por otra parte, fue convertida mediante el programa *bmp2bin* anteriormente citado.






*Ejecución del ejemplo toon_shading.*

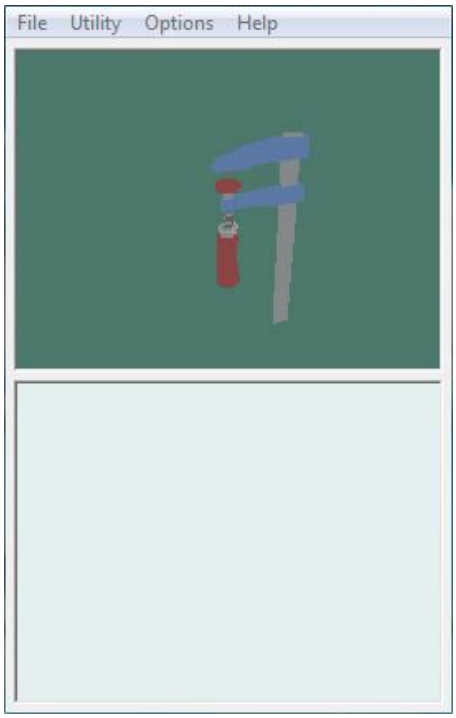



*Figura propia importada.*

## Cargar y mostrar fondos.

Con el objeto de mostrar fondos en la otra pantalla de la *NintendoDS* se recurrió una vez más a un ejemplo de *devkitpro* (*256_color_bmp*) en el cual se carga y muestra como fondo un **.png* . De este modo, experimenté con el código para ver si podía utilizarlo para cargar **.png*'s propios.

Mediante el análisis del ejemplo, descubrí que no es necesario convertir las imágenes de los fondos a formato **.bin*, pues el ejemplo hace uso de *GRIT* (*GBA Image Transmogrifier*), una herramienta de conversión de imágenes para ser cargadas en una *GameBoy Advance* o en una *NintendoDS* (http://www.coranac.com/man/grit/html/grit.htm).

```c
#include <nds.h>

#include <stdio.h>

// grit adds a nice header we can include to access the data
// this has the same name as the image
#include "drunkenlogo.h"

int main(void)
{
    // set the mode for 2 text layers and two extended background layers
    videoSetMode(MODE_5_2D);
    vramSetBankA(VRAM_A_MAIN_BG_0x06000000);

    consoleDemoInit();

    iprintf("\n\n\tHello DS devers\n");
    iprintf("\twww.drunkencoders.com\n");
    iprintf("\t256 color bitmap demo");

    int bg3 = bgInit(3, BgType_Bmp8, BgSize_B8_256x256, 0,0);

    dmaCopy(drunkenlogoBitmap, bgGetGfxPtr(bg3), 256*256);
    dmaCopy(drunkenlogoPal, BG_PALETTE, 256*2);

    while(1)swiWaitForVBlank();

    return 0;
}
```


*Ejecución del ejemplo 256_color_bmp.*

## Integración de las distintas funcionalidades en una única aplicación.

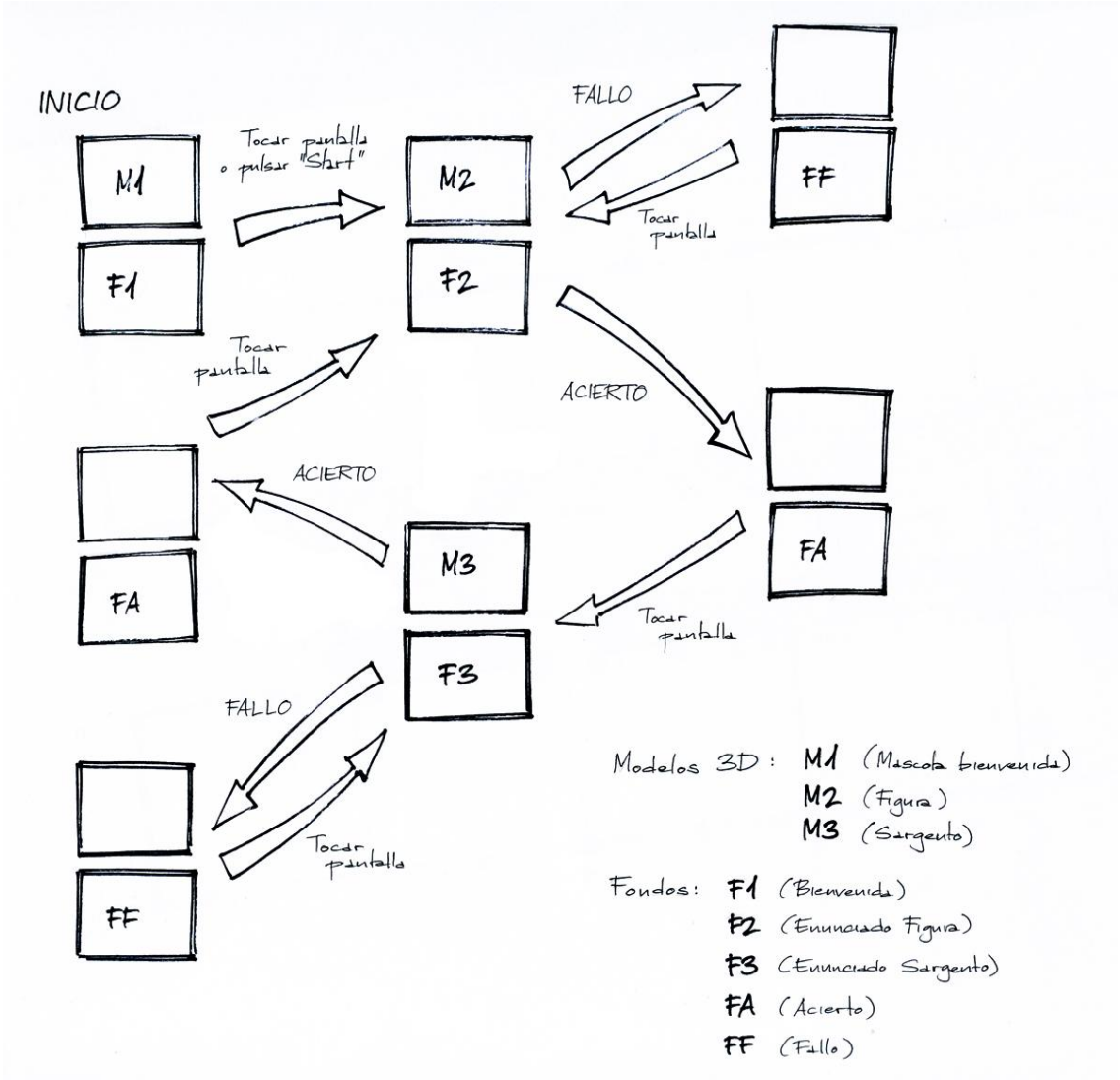
Una vez dominadas las funcionalidades de cargar objetos 3D y de mostrar fondos por pantalla, el siguiente paso es integrarlas en una misma aplicación.

La primera gran dificultad radica en que el *Makefile* que nos permite cargar los modelos en 3D no está preparado para trabajar con *GRIT*, y el *Makefile* del proyecto de ejemplo que permite cargar fondos, no permite la manipulación de figuras tridimensionales. Por tanto, se tomó la decisión de modificar el *Makefile* con funcionalidades 3D para que pudiese trabajar con *GRIT*, siguiendo el siguiente tutorial: http://www.coranac.com/man/grit/html/gritmake.htm.

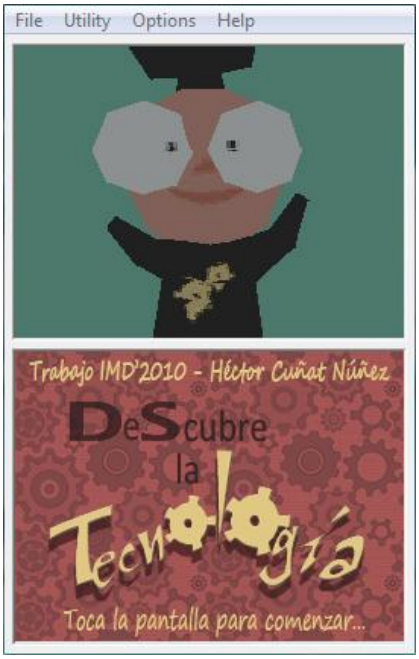
Además, era necesario que se mostrase el fondo y la figura 3D en pantallas distintas y simultáneamente. La NintendoDS consta de 2 procesadores, llamados *MainEngine* y *SubEngine*, asociados por defecto, con la pantalla superior e inferior respectivamente. Así, en el código de la carga del fondo es necesario especificar al SubEngine que muestre la imagen en la pantalla inferior, sustituyendo las funciones *videoSetMode()* y *bgInit()* por *videoSetModeSub()* y *bgInitSub()* y escogiendo un banco de memoria asociado al *SubEngine* (*VRAM_C_SUB* en lugar de *VRAM_A_MAIN*).

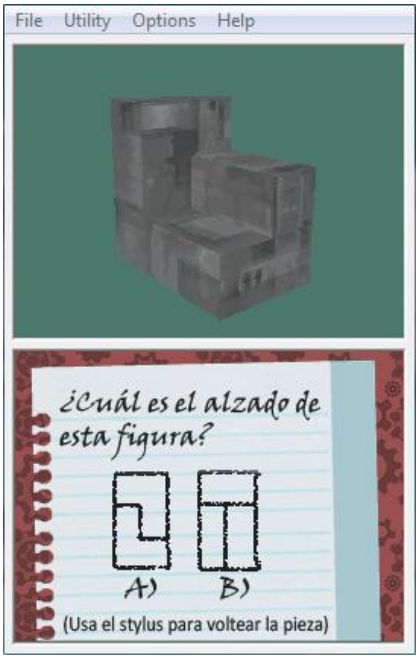
```
#include <nds.h>
#include <stdio.h>

// git adds a nice header we can include to access the data
// this has the same name as the image
#include "drunkenlogo.h"

int main(void)
{
    // set the mode for 2 text layers and two extended background layers
    //videoSetMode(MODE_5_2D);
    //vramSetBankA(VRAM_A_MAIN_BG_0x06000000);

    videoSetModeSub(MODE_5_2D | DISPLAY_BG3_ACTIVE);
    vramSetBankC(VRAM_C_SUB_BG);

    int bg3 = bgInitSub(3, BgType_Bmp8, BgSize_B8_256x256, 0,0);

    dmaCopy(drunkenlogoBitmap, bgGetGfxPtr(bg3), 256*256);
    dmaCopy(drunkenlogoPal, BG_PALETTE_SUB, 256*2);


    while(1)swiWaitForVBlank();

    return 0;
}
```

Finalmente, se crearon una serie de fondos y de figuras tridimensionales, de forma que la aplicación mostrase unas u otras, en cada una de las pantallas, en función de las respuestas del usuario. En el siguiente diagrama se muestra una visión global del funcionamiento y relación entre las diferentes pantallas:




*Esquema de funcionamiento del programa.* Para *voltear la figura se empleará el Stylus. Para escoger la respuesta correcta, se empleará el botón correspondiente.*





En la presente página, se muestran algunas de las pantallas que se visualizan al ejecutarse el programa. Para observar la ejecución completa de la aplicación desarrollada, consultése la dirección: http://www.youtube.com/watch?v=vaO6xPadYUk

## Sonido

Los conceptos se retienen mucho mejor cuando el emisor realza los mensajes mediante el aplauso (sonido de víctoria o éxito) y la desilusión (sonido de fracaso). Por eso, consideré importante desde el comienzo el ser capaz de incorporar sonidos al juego.

Además, una tonadilla sencilla y particular sirve también para identificar a un producto por lo que también pensé que sería conveniente poseer una música de fondo que identificase de modo sonoro la aplicación.

Para incorporar sonido a la aplicación, utilicé las funciones y el banco de sonidos suministrados por el ejemplo *basic_sound* de devkitpro.

```
#include <nds.h>
#include <maxmod9.h>
#include <stdio.h>

#include "soundbank.h"
#include "soundbank_bin.h"

int main() {

    consoleDemoInit();

    mmInitDefaultMem((mm_addr)soundbank_bin);

    // load the module
    mmLoad( MOD_FLATOUTLIES );

    // load sound effects
    mmLoadEffect( SFX_AMBULANCE );
    mmLoadEffect( SFX_BOOM );

    // start playing module
    mmStart( MOD_FLATOUTLIES, MM_PLAY_LOOP );

    mm_sound_effect ambulance = {
        { SFX_AMBULANCE } ,         // id
        (int)(1.0f * (1<<10)), // rate
        0,      // handle
        255,    // volume
        0,      // panning
    };

    mm_sound_effect boom = {
        { SFX_BOOM } ,          // id
        (int)(1.0f * (1<<10)), // rate
        0,      // handle
        255,    // volume
        255,    // panning
    };

    // ansi escape sequence to clear screen and home cursor
    // /x1b[line;columnH
    iprintf("\x1b[2J");

    // ansi escape sequence to set print co-ordinates
    // /x1b[line;columnH
    iprintf("\x1b[0;8HMaxMod Audio demo");
    iprintf("\x1b[3;0HHold A for ambulance sound");
    iprintf("\x1b[4;0HPress B for boom sound");

    // sound effect handle (for cancelling it later)
    mm_sfxhand amb = 0;
```

```
    do {

        int keys_pressed, keys_released;

        swiWaitForVBlank();
        scanKeys();

        keys_pressed = keysDown();
        keys_released = keysUp();

        // Play looping ambulance sound effect out of left speaker if A button is pressed
        if ( keys_pressed & KEY_A ) {
            mmEffectEx(&ambulance);
        }

        // Play explosion sound effect out of right speaker if B button is pressed
        if ( keys_pressed & KEY_B ) {
            mmEffectEx(&boom);
        }

    } while( 1 );
}
```

Por último, es importante reseñar que dicho ejemplo hace uso de las librerías de sonido *MaxMod* (http://www.maxmod.org/ref/tut/dsprog.html) y que por tanto debemos modificar de nuevo el *Makefile* de nuestro proyecto para que las incluya.

# CÓDIGO FUENTE DEL PROTOTIPO

Tras las consideraciones anteriores, el código fuente final de la solución implementada es el siguiente:

```
#include <nds.h>
#include <stdlib.h>
#include <stdio.h>
#include <maxmod9.h>
//modelos (pantalla superior)
#include "sargento_bin.h"
#include "texturesargento_bin.h"
#include "figura_bin.h"
#include "texturefigura_bin.h"
#include "mascota_bin.h"
#include "texturemascota_bin.h"
//fondos (pantalla inferior)
#include "acierto.h"
#include "bienvenida.h"
#include "fallo.h"
#include "figura.h"
#include "sargento.h"
#include "gracias.h"
//bancos de sonidos
#include "soundbank.h"
#include "soundbank_bin.h"

static void get_pen_delta( int *dx, int *dy, u32 keys)
{
    static int prev_pen[2] = { 0x7FFFFFFF, 0x7FFFFFFF };

    touchPosition touchXY;

    if( keys & KEY_TOUCH )
    {

        touchRead(&touchXY);
        if( prev_pen[0] != 0x7FFFFFFF )
        {
            *dx = (prev_pen[0] - touchXY.rawx);
            *dy = (prev_pen[1] - touchXY.rawy);
        }

        prev_pen[0] = touchXY.rawx;
        prev_pen[1] = touchXY.rawy;
    }
    else
    {
        prev_pen[0] = prev_pen[1] = 0x7FFFFFFF;
        *dx = *dy = 0;
    }
}

int main()
{
    bool figura=false;
    bool sargento=false;
    bool acierto_figura=false;
    bool fallo_figura=false;
    bool acierto_sargento=false;
    bool fallo_sargento=false;
    bool gracias=false;
    bool inicio=true;
```

```
int texture[2];

float rotateX = 0.0;
float rotateY = 0.0;

//sonido
mmInitDefaultMem((mm_addr)soundbank_bin);
mmLoad( MOD_FLATOUTLIES );
mmLoadEffect( SFX_BOOM );
mmStart( MOD_FLATOUTLIES, MM_PLAY_LOOP);

mm_sound_effect boom = {
    { SFX_BOOM } ,          // id
    (int)(1.0f * (1<<10)), // rate
    0,      // handle
    255,    // volume
    255,    // panning
};


//2D
videoSetModeSub(MODE_5_2D);
vramSetBankC(VRAM_C_SUB_BG);
vramSetBankA(VRAM_A_TEXTURE);

int bg3 = bgInitSub(3, BgType_Bmp8, BgSize_B8_256x256, 0,0);


//3D
videoSetMode(MODE_0_3D);
glInit();
glEnable(GL_TEXTURE_2D);
glViewport(0,0,255,191);
glEnable(GL_ANTIALIAS);


glClearColor(3,15,11,31);
glClearPolyID(63);
glClearDepth(0x7FFF);

vramSetBankA(VRAM_A_TEXTURE);

glGenTextures(3, texture);

glBindTexture(0, texture[0]);
glTexImage2D(0, 0, GL_RGB, TEXTURE_SIZE_128 , TEXTURE_SIZE_128,
                    0, TEXGEN_TEXCOORD, (u8*)texturefigura_bin);

glBindTexture(0, texture[1]);
glTexImage2D(0, 0, GL_RGB, TEXTURE_SIZE_128 , TEXTURE_SIZE_128,
                    0, TEXGEN_TEXCOORD, (u8*)texturesargento_bin);

glBindTexture(0, texture[2]);
glTexImage2D(0, 0, GL_RGB, TEXTURE_SIZE_128 , TEXTURE_SIZE_128,
                    0, TEXGEN_TEXCOORD, (u8*)texturemascota_bin);
```

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(35, 256.0 / 192.0, 0.1, 40);

gluLookAt(  0.0, 0.0, 1.0,
            0.0, 0.0, 0.0,
            0.0, 1.0, 0.0);

glLight(0, RGB15(31,31,31) , 0, floattov10(-1.0), 0);

while(1) {

    glPushMatrix();

    glTranslate3f32(0, 0, floattof32(-1));

    glRotateX(rotateX);
    glRotateY(rotateY);

    glMatrixMode(GL_TEXTURE);
    glLoadIdentity();

    glMatrixMode(GL_MODELVIEW);

    glMaterialf(GL_AMBIENT, RGB15(16,16,16));
    glMaterialf(GL_DIFFUSE, RGB15(31,31,31));
    glMaterialf(GL_SPECULAR, BIT(15) | RGB15(8,8,8));
    glMaterialf(GL_EMISSION, RGB15(3,3,3));

    glMaterialShinyness();

    glPolyFmt(POLY_ALPHA(31) | POLY_CULL_BACK | POLY_FORMAT_LIGHT0) ;

    scanKeys();

    u32 keys = keysHeld();
    int pen_delta[2];
    get_pen_delta( &pen_delta[0], &pen_delta[1], keys);

    if(inicio){
    dmaCopy(bienvenidaBitmap, bgGetGfxPtr(bg3), 256*256);
    dmaCopy(bienvenidaPal, BG_PALETTE_SUB, 256*2);
    glFlush(0);
    glBindTexture(0, texture[2]);
    glCallList((u32*)mascota_bin);
        if((keys & KEY_START)|(keys & KEY_TOUCH)) { inicio=false; figura=true; }
    }

    if(figura){
    //Voltear con la cruceta
    if((keys & KEY_UP)) rotateX += 3;
    if((keys & KEY_DOWN)) rotateX -= 3;
    if((keys & KEY_LEFT)) rotateY += 3;
    if((keys & KEY_RIGHT)) rotateY -= 3;
    //Voltear con el stylus
    rotateY -= pen_delta[0]/10;
    rotateX -= pen_delta[1]/10;
```

```
        dmaCopy(figuraBitmap, bgGetGfxPtr(bg3), 256*256);
        dmaCopy(figuraPal, BG_PALETTE_SUB, 256*2);
        glFlush(0);
        glBindTexture(0, texture[0]);
        glCallList((u32*)figura_bin);
            if((keys & KEY_A)) { mmEffectEx(&boom); figura=false; acierto_figura=true;  }
            if((keys & KEY_B)) { mmEffectEx(&boom); figura=false; fallo_figura=true; }
        }

        if(acierto_figura){
        dmaCopy(aciertoBitmap, bgGetGfxPtr(bg3), 256*256);
        dmaCopy(aciertoPal, BG_PALETTE_SUB, 256*2);
        glFlush(0);
            if((keys & KEY_TOUCH)) { acierto_figura=false; sargento=true; }
        }

        if(fallo_figura){
        dmaCopy(falloBitmap, bgGetGfxPtr(bg3), 256*256);
        dmaCopy(falloPal, BG_PALETTE_SUB, 256*2);
        glFlush(0);
            if((keys & KEY_TOUCH)) { fallo_figura=false; figura=true; }
        }

        if(sargento){
        //Voltear con la cruceta
        if((keys & KEY_UP)) rotateX += 3;
        if((keys & KEY_DOWN)) rotateX -= 3;
        if((keys & KEY_LEFT)) rotateY += 3;
        if((keys & KEY_RIGHT)) rotateY -= 3;
        //Voltear con el stylus
        rotateY -= pen_delta[0]/10;
        rotateX -= pen_delta[1]/10;

        dmaCopy(sargentoBitmap, bgGetGfxPtr(bg3), 256*256);
        dmaCopy(sargentoPal, BG_PALETTE_SUB, 256*2);
        glFlush(0);
        glBindTexture(0, texture[1]);
        glCallList((u32*)sargento_bin);
            if((keys & KEY_A)) { mmEffectEx(&boom); sargento=false; acierto_sargento=true; }
            if((keys & KEY_B)) { mmEffectEx(&boom); sargento=false; fallo_sargento=true; }
            if((keys & KEY_X)) { mmEffectEx(&boom); sargento=false; fallo_sargento=true; }
            if((keys & KEY_Y)) { mmEffectEx(&boom); sargento=false; fallo_sargento=true; }
        }

        if(acierto_sargento){
        dmaCopy(aciertoBitmap, bgGetGfxPtr(bg3), 256*256);
        dmaCopy(aciertoPal, BG_PALETTE_SUB, 256*2);
        glFlush(0);
            if((keys & KEY_TOUCH)) { acierto_sargento=false; figura=true; }
        }

        if(fallo_sargento){
        dmaCopy(falloBitmap, bgGetGfxPtr(bg3), 256*256);
        dmaCopy(falloPal, BG_PALETTE_SUB, 256*2);
        glFlush(0);
            if((keys & KEY_TOUCH)) { fallo_sargento=false; sargento=true; }
        }

         glEnd();

         glPopMatrix(1);

         swiWaitForVBlank();
    }

    return 0;
}//del main
```

# CONCLUSIONES Y PROPUESTAS DE MEJORA

## Conclusiones

La experiencia ha sido positiva y muy interesante, tanto por lo que he aprendido como por las posibilidades del proyecto. De hecho, lo que comenzó como un trabajo para la asignatura de IPM se ha ido perfilando como un proyecto que puede alcanzar mayor envergadura en un desarrollo posterior. En caso de abordarse, el proyecto de perfeccionamiento de la aplicación debería contener: *1)* mejoras técnicas y funcionales que se me han ido ocurriendo a medida que avanzaba en su realización. *2)* mejoras propuestas por los usuarios consultados.

## Propuesta de mejoras funcionales y técnicas

- Inclusión de gran variedad de figuras.
- Mayor versatilidad mediante la carga de archivos externos.
- Creación de un banco de sonidos propio.
- Que las preguntas salgan aleatoriamnete sin un orden definido (para que no puedan memorizarse las respuestas según el orden de aparición de las preguntas).
- Que haya un límite de tiempo para contestar las preguntas.
- Que sólo pueda darse una respuesta o que puntúen menos los aciertos después de haber fallado previamente.
- Que al pulsar START se pause el juego y se muestren las estadísticas y puntuaciones del jugador.

## Mejoras propuestas por los usuarios (alumnos de Secundaria del Eliseo Vidal)

Una vez finalizado el prototipo, se consultó a los alumnos del CEIP Eliseo Vidal sobre qué les había gustado y cómo les gustaría que evolucionase la apliación, recogiendo los siguientes comentarios y sugerencias:

- Llamarle *lápiz* al *stylus* (en general, el alumnado no interpreta qué es el stylus).

- Prefieren que se responda también con el stylus y no con los botones.

- Que la dificultad de las preguntas se reajuste según los resultados obtenidos hasta el momento.

- Que el jugador pueda crear un perfil al estilo de los Mii's de la Wii. Además, les gustaría que, como en algunos foros, a cada jugador se le llame de una forma según su índice de aciertos.

- Que el jugador se mueva por un mapa con los distintos mundos en los que se agrupen las preguntas según su temática (mundo del dibujo técnico, mundo de la electricidad,...).

- Que puedas acceder a una pizarrita donde hacer dibujos en sucio con el stylus.

# BIBLIOGRAFÍA


- *http://libnds.devkitpro.org*
- *http://devkitpro.org*
- *http://www.dev-scene.com*
- *http://www.elotrolado.net*
- *http://patater.com*
- *http://www.maxmod.org*

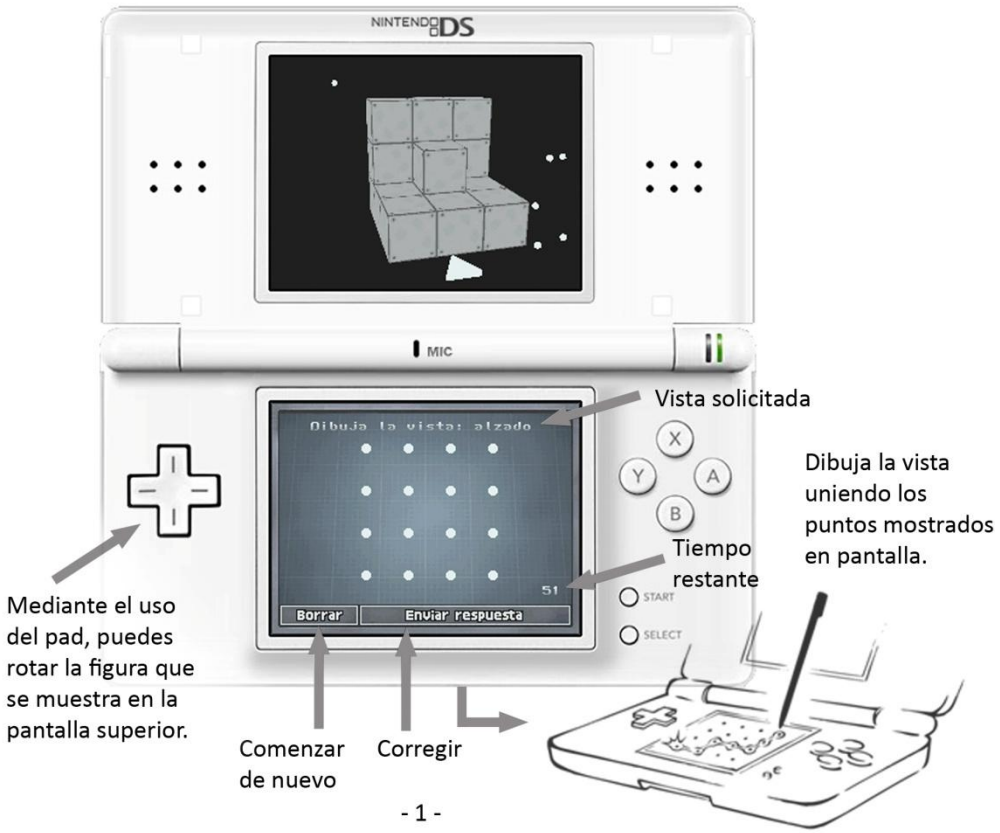
# ANEXO II

## MANUAL DEL USUARIO


Para ejecutar la aplicación, debes situar los archivos suministrados: el ejecutable (*.nds), las figuras (*.obj) y el XML que contiene las preguntas del test, en el directorio raíz de tu *flashcart*. Basta con encender la NintendoDS y seleccionar el ejecutable desde el menú de tu *flashcart* para comenzar el test.

## REALIZAR EL TEST

Una vez introducido el nombre del jugador, la aplicación realizará una serie de preguntas de forma automática. En ellas, el jugador deberá dibujar en la pantalla táctil (inferior) la vista indicada de la figura mostrada en la pantalla superior.



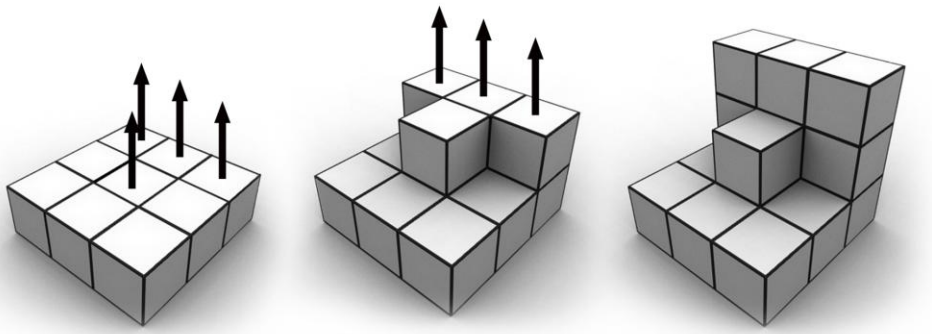
# PERSONALIZACIÓN DEL TEST

Puedes diseñar tu propio test sustituyendo las figuras suministradas (*.obj) por las figuras que desees y editando el fichero XML que contiene los enunciados de las distintas preguntas:

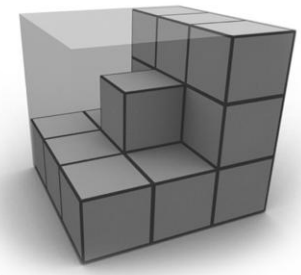
# CREACIÓN DE FIGURAS PROBLEMA

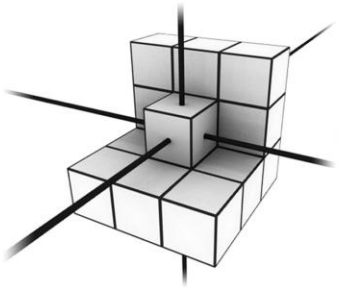
Puedes construir tus propias figuras en el programa de modelado que desees. Para ello, se recomienda realizar sucesivas extrusiones sobre la cara de un cubo hasta alcanzar la figura deseada.

Recuerda que todas las caras han de tener cuatro lados.



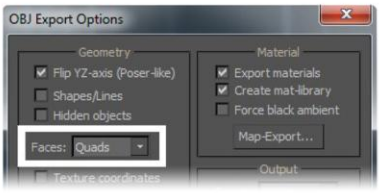
Para que tus figuras tengan el tamaño adecuado deben caber dentro de un cubo cuya longitud de arista sea de 1 unidad (puedes tomar como referencia una de las figuras suminstradas).





Centra tu figuras en el origen de coordenadas, pues girarán alrededor de ese punto.

Por último, recuerda exportar tus figuras en formato *.obj, y con todas sus caras como *quads*.



# EDICIÓN DEL ENUNCIADO

Una vez creadas las figuras deseadas, debe editarse el archivo XML suministrado y que contiene las distintas preguntas formuladas:

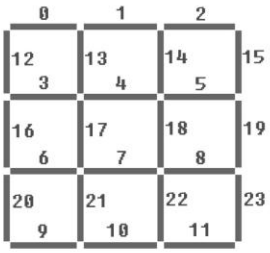
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<test>
<numpreguntas>6</numpreguntas>
<pregunta>
 <id>1</id>
 <enunciado>alzado </enunciado>
 <figura>figura001.obj </figura>
 <aristas>1 1 1 0 1 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 0 0 1</aristas>
 <tiempo>30</tiempo>
 <dificultad>1</dificultad>
</pregunta>
      ...
</test>
```

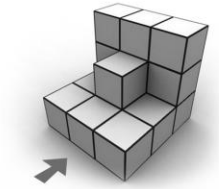
Escribe aquí el número total de preguntas del test.

Escribe aquí el número de la pregunta (aunque recuerda que se mostrarán aleatoriamente), la vista a dibujar, y la figura de la cual dibujar dicha vista.

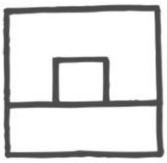
Escribe aquí el tiempo (en segundos) disponible para responder la pregunta y la dificultad asociada a la misma.

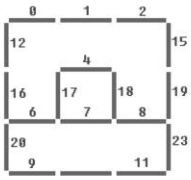
Para especificar la solución a la vista pedida, consulta en el esquema de la derecha la numeración de las distintas aristas e introduce un 1 en aquellas posiciones en las que dicha arista forme parte de la solución.



Por tanto, la pregunta del ejemplo consistiría en dibujar el alzado de la figura creada en la página anterior, en menos de 30 segundos. Dicho ejercicio tiene asignada una dificultad de grado 1.






# RESULTADOS

Cada vez que un alumno realiza el test, se archivan sus resultados en un archivo de texto generado por la aplicación en la carpeta raíz. En dicho archivo podemos encontrar: el nombre del alumno, la fecha y hora en la cual realizó el test, el número de preguntas acertadas, el tiempo empleado para ello,...

# ÍNDICE ANALÍTICO

## *T*



## *V*



## *X*

# ÍNDICE DE FIGURAS