Tratamiento de archivos BLOB en Clipper 5.3 y Visual Objects

Por Antonio Quirós

Tanto en Visual Objects como en Clipper 5.3, se incorpora un RDD especial para poder trabajar con campos memo, de forma que el soporte que recibimos para dichos campos mejore ostensiblemente respecto al que teníamos con los archiconocidos y nunca bien vituperados archivos DBT. Este nuevo RDD es parte de un producto comercial denominado FlexFile y sus principales virtudes se reflejan en la siguiente relación de propiedades del mismo:

- Tamaño máximo de archivo: 4.2 Gb.
- Tamaño mínimo de bloque: 1 byte.
- Técnica eficiente para el reciclado de espacio no usado.
- Posibilidad de almacenar cualquier tipo de datos, a excepción de codeblocks y objectos.
- Conjunto de funciones (BLOB) para manipular los archivos memo con sus correspondientes campos.

A través del *RDDBLOB* podemos disponer de tres posibilidades de trabajo:

- Archivos memo .FPT de bases de datos DBFCDX.
- Archivos memo .DBV de bases de datos DBFNTX o DBFMDX
- Archivos BLOB independientes que no necesitan estar conectados a ninguna base de datos para funcionar.

Hasta ahora estamos acostumbrados a *asociar* archivos memo a bases de datos, sin embargo existe una tercera posibilidad que es la de trabajar directamente con un sólo archivo *DBV* que contiene lo que se denominan datos de tipo *BLOB (Binary Large Objects)*. A pesar del nombre, el contenido de este archivo puede ser el mismo que el de los *FPT* o *DBV* vistos hasta ahora. La diferencia radica en el sistema de localización de cada dato que en cada uno de ellos se sigue. En los primeros la localización se lleva a cabo a través de un puntero que se guarda, según el sistema tradicional, en el registro del *DBF* asociado al dato *memo*, mientras que aquí al no existir esa asociación somos nosotros los que hemos de ocuparnos de mantenerla.

Para ello disponemos de una maquinaria basada en lo siguiente. Cada archivo BLOB se divide en dos partes: *datos* y *raíz*. Para acceder a cada dato de la parte de datos hay que conocer el puntero por el que el sistema reconoce a cada uno de los BLOB allí archivados. Este puntero puede guardarse en un array de punteros en el área raíz. Recuerde el lector que en los archivos de *FlexFile* vamos a poder guardar cualquier tipo de dato, lo que incluye los arrays. La figura 1 representa esta arquitectura. Nótese como el área raíz se ubica al final del archivo BLOB, de forma que la incorporación de datos a la misma no suponga recomposición del área de datos.

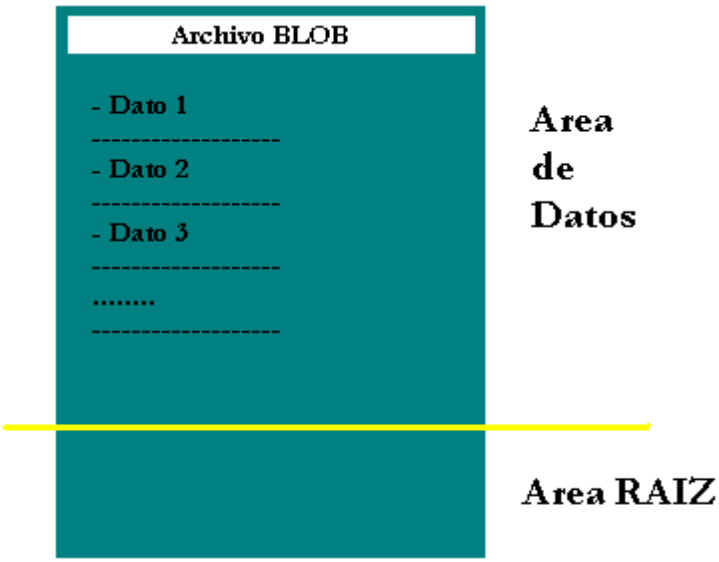

Figura 1: Arquitectura para la localización de datos

El caso de Clipper

Para trabajar con archivos BLOB tenemos que incorporar el siguiente *INIT PROCEDURE:*

```
ANNOUNCE RddSys
INIT PROCEDURE RddInit
   REQUEST DBFBLOB
   RDDSetDefault("DBFBLOB")
   RETURN
```

En el fuente 1 podemos ver un ejemplo para crear un archivo DBV a través del RDD BLOB:

```
// ---Fuente 1: Creablob.prg -------------------------------------
FUNCTION Main()

   LOCAL aDbf := {}

   AADD(aDbf, {"NOTAS","M",10,0})

   DbCreate("BLOBOS",aDbf,"DBFBLOB")

RETURN NIL

ANNOUNCE RddSys
INIT PROCEDURE RddInit
   REQUEST DBFBLOB
   RDDSetDefault("DBFBLOB")
   RETURN
*------------------------------------------------------------------
```

Este programa crea un archivo denominado *BLOBOS.DBV.* Lo único digno de mencionarse en él es el modo que tenemos con *DbCreate()* de crear un BLOB. Nótese como en el array de

estructura ponemos un solo campo, de nombre *NOTAS* y de tipo memo. Esto es indiferente, ya que en cualquier caso se creará un archivo DBV vacío y listo para que pongamos cosas en sus áreas raíz y de datos.

La compilación del fuente puede llevarse a cabo con:

```
CLIPPER creablob /n/w
```

Y el enlazado con:

```
EXOSPACE FI creablob LIB DBFBLOB
```

Ahora vamos a incorporar información a dicho archivo. Lo que haremos será incorporar un array de mil elementos y el puntero que nos indica donde se guarda el dato será almacenado en el área raíz en otro array. Si ejecutamos en sucesivas ocasiones, por cada una de ellas se añadirá un nuevo array con mil elementos al área de datos y el puntero a ese nuevo array se almacenará en una nueva posición del array que guardamos en el área raíz. De este modo tenemos listos nuestros dos arrays. El del área raíz contiene los punteros a todos y cada uno de los arrays del área de datos, de forma que a través de dichos punteros podemos acceder a cada uno de los arrays.

Actuamos así porque en la arquitectura de los BLOB para acceder al área de datos es necesario conocer el puntero donde se ubican los datos, mientras que al área raíz podemos acceder directamente, pero sólo podemos tener un dato guardado en la misma. El fuente 2 muestra todo este proceso.

```
*--Fuente 2: Fillblob.prg ------------------------------------------
#Include "Blob.Ch"

FUNCTION Main()
   LOCAL aClientes := {}
   LOCAL aRefs     := {}
   LOCAL nAscii := 65
   LOCAL n      := 0
   LOCAL dFecha := Date()-10000
   LOCAL nPunt  := 0

   USE Blobos

   CLS
   @ 10,10 SAY "Voy por el: "

   FOR n:= 1 TO 1000

      AAdd(aClientes, {n,;
                       Replicate(Chr(nAscii),40),;
                       dFecha++})

      IF ++nAscii > 90
         nAscii := 65
      ENDIF

      @ 10,22 SAY AllTrim(Str(n))
   END FOR

   @ 10,10 SAY "Añadiendo al BLOB                      "
   nPunt := Blobos->(BLOBDirectPut(0,aClientes))
```

```
   @ 10,10 SAY "El puntero es: " +Str(nPunt)

   IF ValType(Blobos->(BLOBRootGet())) == "A"
      aRefs := Blobos->(BLOBRootGet())
      AAdd(aRefs, nPunt)
    ELSE
      AAdd(aRefs, nPunt)
   ENDIF

   Blobos->(BLOBRootPut(aRefs))

RETURN NIL

ANNOUNCE RddSys
INIT PROCEDURE RddInit
   REQUEST DBFBLOB
   RDDSetDefault("DBFBLOB")
   RETURN
*------------------------------------------------------------------------
```

Nótese el *include* a *Blob.Ch*; esto es muy importante, ya que las funciones que se emplean en el fuente, tal como *BLOBRootPut()* son realmente seudofunciones definidas a través del preprocesador y no funciones reales de las librerías. Nótese en el fuente cómo usamos las funciones *BLOBRootGet()* y *BLOBRootPut()* para leer y escribir en el área raíz y las funciones *BLOBDirectGet()* y *BLOBDirectPut()* para leer y escribir en el área de datos. Por lo demás, es interesante notar cómo la manipulación del DBV en cuanto a la apertura del mismo, etc., se hace a través de los mandatos y funciones estándar de Clipper para manipular bases de datos.

La compilación del fuente puede llevarse a cabo con:

```
CLIPPER fillblob /n/w
```

Y el enlazado con:

```
EXOSPACE FI fillblob LIB DBFBLOB
```

para modo protegido y:

```
BLINKER FI fillblob LIB DBFBLOB
```

para modo real.

Por último vamos a ver el fuente 3 que se encarga de la lectura y visualización de los datos antes grabados:

```
*-- Fuente 3: Leeblob.prg ---------------------------------------------
#Include "Blob.Ch"
#Include "Inkey.Ch"

FUNCTION Main()
   LOCAL aClientes := {}
```

```
   LOCAL aRefs  := 0
   LOCAL n      := 0
   LOCAL nRef   := 0
   LOCAL nRow   := 3
   LOCAL nTecla

   USE Blobos
   aRefs := Blobos->(BLOBRootGet())

   CLS
   @ 01,02 SAY "CODIGO--"
   @ 01,10 SAY "NOMBRE-----------------------------------"
   @ 01,50 SAY "--FECHA------"
   @ 24,10 SAY "[Esc] = Terminar"

   FOR nRef := 1 TO Len(aRefs)

       aClientes := Blobos->(BLOBDirectGet(aRefs[nRef]))

       FOR n:= 1 TO Len(aClientes)
           @ nRow, 03 SAY StrZero(aClientes[n,1],5)
           @ nRow, 10 SAY aClientes[n,2]
           @ nRow, 52 SAY aClientes[n,3]

           IF ++nRow == 22
              nRow := 3
              nTecla := Inkey(0)
              IF nTecla == K_ESC
                 CLS
                 EXIT
              END IF
           END IF

       END FOR
       IF nTecla == K_ESC
          EXIT
       ENDIF
   END FOR

RETURN NIL


ANNOUNCE RddSys
INIT PROCEDURE RddInit
   REQUEST DBFBLOB
   RDDSetDefault("DBFBLOB")
   RETURN
*------------------------------------------------------------------------
```

El procedimiento seguido es relativamente sencillo. Se trata sólo de recorrerse el array de punteros que guardamos en el área raíz del DBV e ir leyendo cada array del área de datos asociado al puntero antes recuperado.

La compilación del fuente puede llevarse a cavo con:

```
CLIPPER viewblob /n/w
```

Y el enlazado con:

```
EXOSPACE FI viewblob LIB DBFBLOB
```

para modo protegido y:

```
BLINKER FI viewblob LIB DBFBLOB
```

para modo real.

La mecánica de trabajo con los BLOB es muy útil cuando deseamos trabajar con algún tipo de formato de longitud variable. En conjunción con los arrays anidados de Clipper podemos construir potentísimas estructuras de almacenamiento no sujetas al corsé de la definición de campos en una base de datos.

El caso de Visual Objects

Los BLOB desde Visual Objects se manipulan de forma muy similar a Clipper. Veamos, no obstante, algunas de sus características:

- No es necesario indicar que se va a usar un RDD especial más que a través de la cláusula *VIA* o de *RDDSetDefault()*. De forma automática, el mecanismo de construcción de aplicaciones de Visual Objects, toma la librería que necesita.

- Hasta la versión 1.0b no pueden usarse BLOB en archivos DBV independientes, a través de la clase *DBServer*. Esto es algo bastante equívoco, ya que la documentación no hace ninguna referencia a ello y los métodos equivalentes a las funciones para la gestión de BLOB están disponibles (por ejemplo *BLOBRootGet(), BLOBDirectExport()*, etc. Estos métodos no pueden llegar a usarse nunca con un DBV independiente, ya que en el proceso de instanciación obtenemos un error. Dada esta limitación, sólo podemos usar BLOB con la metodología procedural ordinaria basada en las funciones correspondientes.

- Consultada CA-USA respecto a la dificultad anterior, indican que quizá la versión 1.0c, a punto de liberarse, solucione el problema y que de no ser así lo hará el siguiente patch (!!!).

Para aprender cómo trabajar con BLOB en Visual Objects vamos a realizar una pequeña aplicación de ejemplo que nos servirá para catalogar archivos *BMP*. En el área de datos del BLOB vamos a guardar el *BMP* y en el área raíz el puntero a cada uno de los datos junto con un identificador del mismo que el usuario pueda usar para reconocerlo, en este caso usaremos el nombre del *BMP*.

La aplicación se encargará de todas las acciones necesarias para el mantenimiento de este sistema de catálogo de archivos *BMP*: crear catálogo, guardar los *BMP* en ellos y visualizarlos. Veamos en el fuente 4, el método *Start()* de nuestra aplicación, así como otros elementos inciales de la misma: la clase para la ventana de entorno y un variable global que también usaremos más tarde.

```
*----- Fuente 4-----------------------------------------------------
CLASS WENTORNO INHERIT SHELLWINDOW
      EXPORT BlobActive AS STRING

METHOD Start() CLASS App

      LOCAL oWin AS WENTORNO

      Enable3DControls()

      oWin := WEntorno{self}
```

```
      oWin:Caption := "Pruebas con BLOBS | Ningún BLOB activo"
      oWin:Menu := MenuBLOB{Self}
      oWin:Icon := IcoBLOB{}
      oWin:Show(SHOWZOOMED)

      Self:Exec()

GLOBAL ArrayBlob:= {} AS ARRAY
*------------------------------------------------------------
```

Poco que comentar de este fuente. Lo único es aclarar que la *EXPORT BLOBActive* que definimos para nuestra clase ventana de entorno, se encargará de guardar el nombre del BLOB que tenemos activo en cada momento.

El menú será sencillo, las opciones que anotaremos en él, asociadas a sus correspondientes eventos son los de la tabla 1:

Opción	Evento
Crear BLOB	CreaBLOB
Abrir BLOB	OpenBLOB
Llenar BLOB	FillBLOB
Ver BLOB	ViewBLOB
Salir	EndWindow

Tabla 1: Eventos asociadoa a los ítems

El fuente 5 muestra el código de la primera opción.

```
*--- Fuente 5 -------------------------------------------------
METHOD CreaBLOB() CLASS WENTORNO
      LOCAL aDbf := {}
      LOCAL oSD AS SAVEASDIALOG

      oSD := SaveAsDialog{Self, "*.dbv"}
      oSD:Show()

      IF !Empty(oSD:FileName)
            AADD(aDbf, {"NOTAS","M",10,0})
            DbCreate(oSD:FileName,aDbf,"DBFBLOB")
            Self:WarningMessage("Aviso","Archivo creado")
      END IF

RETURN NIL
*------------------------------------------------------------------
```

Lo único digno de mencionarse es el uso de la caja estándar para grabar archivos *SaveAsDialog*, que usamos para que el usuario proporcione al sistema el nombre del archivo, por lo demás el código fuente es totalmente similar al de Clipper.

En el fuente 6 podemos ver la segunda opción que nos sirve para seleccionar un archivo BLOB ya creado y abrirlo. El resto de las opciones del sistema: **Llenar BLOB** y **Ver BLOB** no podrán ejecutarse hasta que no hayamos pasado previamente por la acción de abrir el BLOB.

```
*-- Fuente 6 ------------------------------------------------------
METHOD OpenBlob() CLASS WEntorno
      LOCAL oOD AS OPENDIALOG

      oOD := OpenDialog{Self, "*.DBV"}
      oOD:Show()

      IF !Empty(oOD:FileName)
            Self:BlobActive := oOD:FileName
            USE (Self:BlobActive) VIA "DBFBLOB"
            Self:caption := "Pruebas con BLOB | "+Self:BlobActive
      END IF
*-------------------------------------------------------------------
```

Como podemos ver, lo único que hacemos es usar la caja estándar para apertura de archivos, *OpenDialog* y seleccionar el archivo DBV con el que vamos a trabajar. Se abre dicho archivo con el USE convencional (recuérdese que no podemos usar *DbServer*), se guarda su nombre en la EXPORT *BLOBActive*, anotándose además en el caption de la ventana de entorno.

El fuente 7 nos muestra la opción para guardar los BMP en el archivo BLOB. Para usar esta opción hemos diseñado una pequeña ventana de diálogo no modal con un mensaje que aparecerá en nuestra pantalla mientras el BMP se está guardando en el BLOB. La figura 2 muestra el mencionado diseño.

Figura 2: Ventana de aviso

```
*--- Fuente 7 ------------------------------------------------------------
METHOD FillBLOB() CLASS WENTORNO
      LOCAL oOD AS OPENDIALOG
      LOCAL aRef := {} AS ARRAY
      LOCAL nPunt AS DWORD
      LOCAL oWAviso AS AVISO


      IF !Empty(Self:BlobActive)
            oOD := OpenDialog{Self, "*.BMP"}
            oOD:Show()

            IF !Empty(oOD:FileName)

                  oWAviso := Aviso{Self}
                  oWAviso:Show()

                  nPunt := BLOBDirectImport(0,oOD:FileName)

                  IF ValType(BLOBRootGet()) == "A"
                     aRef := BLOBRootGet()
                     AAdd(aRef, {Right(oOD:FileName,;
                                 Len(oOD:FileName)-;
                                 RAT("\",oOD:FileName));
```

```
                                                ,nPunt})
                  ELSE
                     AAdd(aRef, {Right(oOD:FileName,;
                                 Len(oOD:FileName)-;
                                 RAT("\",oOD:FileName));
                                                ,nPunt})
                  ENDIF

                  BLOBRootPut(aRef)

                  oWAviso:EndDialog()

                  END IF
      ELSE
            MessageBox(0,  "Primero debe abrir un BLOB",;
                                              "Error",MB_ICONSTOP)
      END IF
*-----------------------------------------------------------------------
```

Las operaciones que se hacen son sencillas. Se escoge el BMP a catalogar a través de la caja estándar para abrir archivos, *OpenDialog*. Se muestra el aviso de que está grabando (ventana *Aviso*, ver figura 2) y se graba en el área de datos del BLOB a través de la función *BLOBDirectImport()* que nos devuelve el puntero a donde dicho BMP ha quedado grabado. Más tarde se lee el área raíz (*a través de BLOBRootGet()*) y se añade a la misma (que contiene un array de dos dimensiones) el nombre del BMP y el puntero que el proceso anterior nos ha devuelto. Por último, este array actualizado se vuelve a grabar en el área raíz a través de *BLOBRootPut()*. La instrucción

```
Right(oOD:FileName,          ;
      Len(oOD:FileName)-     ;
      RAT("\",oOD:FileName)  ;
     )
```

tiene como única utilidad la de proporcionarnos un nombre de archivo sin su ruta, ya que el dato *FileName* de *OpenDialog* nos proporciona el nombre del archivo seleccionado con la ruta completa necesaria para acceder al mismo.

El último proceso, y el más complejo, es el que necesitamos para visualizar cada BMP catalogado. Lo primero que necesitamos es una ventana de diálogo modal (a la que llamaremos *View*) que nos permita seleccionarl el BMP a visualizar entre todos los presentes en el BLOB. Dicha ventana necesita también un control sobre el que visualizar el BMP, en este caso, hemos elegido un pushbutton inactivo. La figura 3 nos muestra el diseño de esta ventana.

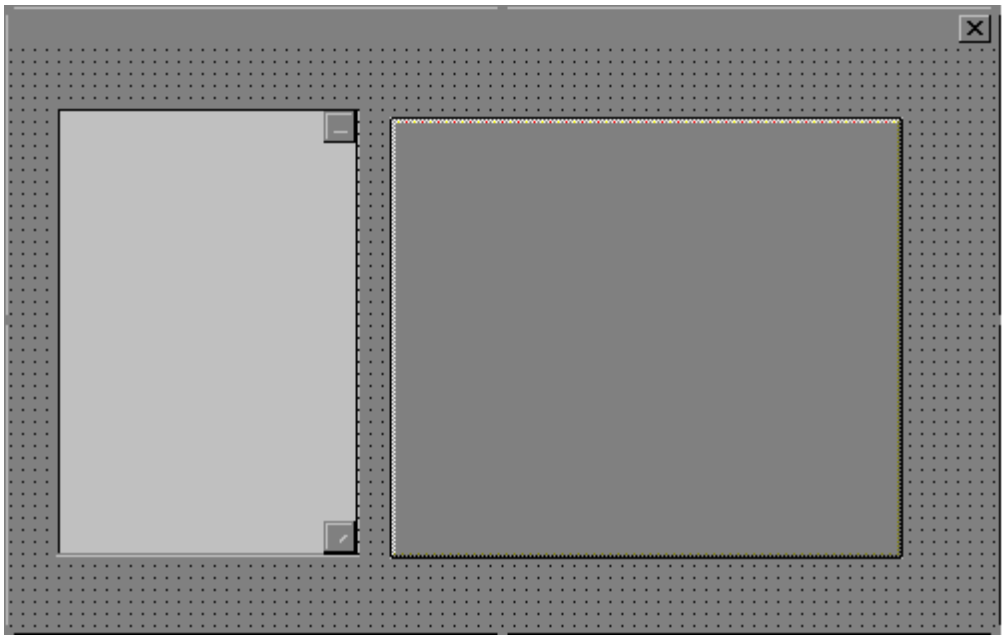

Figura 3: Ventana para visualizar el BMP

Veamos el fuente 8 que nos muestra la opción de visualizar estos BMP.

```
// --- Fuente 8 ---------------------------------------------------------
METHOD ViewBLOB() CLASS WENTORNO
      LOCAL oView AS VIEW

      IF !Empty(Self:BlobActive)

            ArrayBlob := BLOBRootGet()

            IF Len(ArrayBlob) > 0

                  oView := View{Self}
                  oView:caption := Self:BlobActive
                  oView:oDCLista:CurrentItemNo := 1

                  BLOBDirectExport(oView:oDCLista:Value,;
                              "temp.bmp",;
                   BLOB_EXPORT_OVERWRITE )

                  oView:Show()
            ELSE
                  MessageBox(0,  "El BLOB no tiene datos",;
                             "Error",MB_ICONSTOP)
            END IF
      ELSE
            MessageBox(0,  "Primero debe abrir un BLOB",;
                           "Error",MB_ICONSTOP)
      END IF
*------------------------------------------------------------------------
```

El proceso seguido es el siguiente:

- Comprobar que tengamos un BLOB abierto:

```
IF !Empty(Self:BlobActive)
```

y que tenga datos:

```
IF Len(ArrayBlob) > 0
```

- Leer su área raíz y cargarla a la variable global *ArrayBlob*. Emplearemos esta variable global para que el listbox de la caja de diálogo de la figura 3 pueda procesar el array que contiene.

- Ponemos en el *caption* de la ventana el nombre del BMP a visualizar e informamos al listbox que debe seleccionar su primer elemento.

- El pushbutton de la ventana va a mostrar los datos que contenga un archivo denominado *temp.bmp*. Este se empleará como un archivo temporal donde volcamos desde el BLOB cada BMP seleccionado para que pueda ser visualizado. Actuamos así, ya que emplearemos la función del API de Windows *StretchBitMap()* para mostrar el BMP y esta función sólo puede manipular un archivo BMP a través de su nombre. El volcado del BLOB se realiza a través de *BLOBDirectExport()*. Al abrir la ventana por primera vez se vuelca el primer BMP contenido en el BLOB, los siguientes BMP elegidos se volcarán a través del evento *ListBoxSelect()* de la ventana *View*.

- Como puede verse en el fuente 8, no se ha pintado nada hasta ese momento. Esta labor la dejamos en manos del evento *Expose()* de la ventana *View.* El fuente 9 nos lo muestra.

```
// --- Fuente 9 ------------------------------------------------
METHOD Expose( oExposeEvent ) CLASS VIEW
       LOCAL oBoundingBox AS BoundingBox
       oBoundingBox := IIf( oExposeEvent == NULL_OBJECT, NULL_OBJECT,;
                            oExposeEvent:ExposedArea )
       SUPER:Expose( oExposeEvent )

       StretchBitMap( Self:oCCSubWin:Handle(),;
                      "temp.bmp",;
                      "")

RETURN NIL
*------------------------------------------------------------------
```

- La función *StretchBitMap()* que se encarga de pintar el bitmap recibe tres parámetros:

  - El *handle* de la ventana donde se va a pintar el bitmap, en este caso del pushbutton. Recuerde el lector que para Windows, todo son ventanas, incluso los controles, por ello el *handle* de la ventana puede ser el handle de un control de la misma.

  - El nombre del BMP que se va a visualizar.

  - El caption a poner en la ventana.

  El evento *Expose()* se ejecuta la primera vez que se pinta la ventana y cada vez que se necesita repintado de la misma, por ello es el lugar idóneo para usar *StretchBitmap()*.

- Veamos por último, como se produce la selección del bitmap a visualizar a través del evento *ListBoxSelect()*. El fuente 10 nos lo muestra.

```
// --- Fuente 10 ------------------------------------------------
-
METHOD ListBoxSelect( oControlEvent ) CLASS VIEW
   LOCAL oControl AS ListBox
   LOCAL uValue AS USUAL
   oControl := IIf( oControlEvent == NULL_OBJECT, NULL_OBJECT,;
                    oControlEvent:Control )
   SUPER:ListBoxSelect( oControlEvent )
   //Put your changes here

   BLOBDirectExport(Self:oDCLista:Value,;
                     "temp.bmp",;
                     BLOB_EXPORT_OVERWRITE )

   StretchBitMap(Self:oCCSubWin:Handle(),;
                             "temp.bmp",;
                             "")
*------------------------------------------------------------------
```

Este evento que se ejecuta con sólo posicionarnos sobre un evento del listbox, exporta a *temp.bmp* el contenido del bitmap que tenemos seleccionado en la lista. Nótese como usamos un array de dos dimensiones (*ArrayGlobal*) para la gestión del listbox. La primera dimensión contiene el valor a mostrar (el nombre del BMP) y la segunda el puntero para localizar dicho archivo en el BLOB. Este array se obtiene directamente por la lectura del área raíz del BLOB. La última acción que se realiza es pintar el *bitmap* en el pushbutton, ya que *Expose()* no se ejecuta de forma automática cuando seleccionamos un elemento en un listbox.

La figura 4 nos muestra cómo se produce este proceso de visualización.

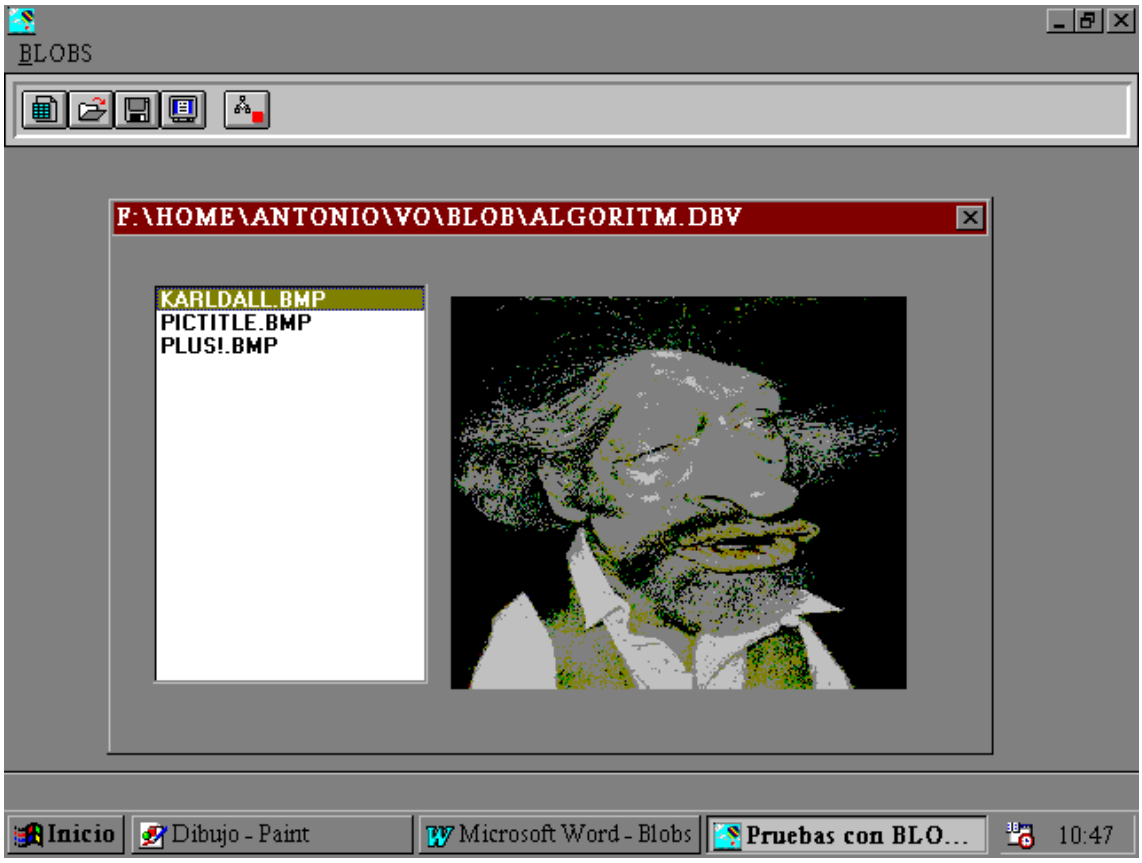

Figura 4: Proceso de visualización

Bien, y esto es todo sobre el tratamiento de BLOB. He intentado mostrar aquí, cómo se produce dicho tratamiento en Clipper y Visual Objects, quizá con dos ejemplos no muy útiles, sin embargo, la potencia que nos proporciona disponer de un sistema de archivo en campos de longitud variable seguro que nos induce a la confección de muchos otros ejemplos de más utilidad que los propuestos.