Definición de la Arquitectura BICOTI-II

Marcelo Alarcón
Claudio Risso
Carlos Soderguit

3 de Febrero 2000

1 Introducción

El propósito de este documento es establecer en forma precisa, la arquitectura de un sistema que a nuestro entender cumple con los objetivos y consideraciones de diseño ya tratadas en [Sod99]. Presentaremos el documento en forma evolutiva, de acuerdo de hecho a como se fueron tratando las etapas durante el proceso de diseño. Brevemente recordemos que los objetivos propuestos eran: *transparencia*, *flexibilidad*, *alto grado de interacción*, *escalabilidad* y *portabilidad*.

Afinado el análisis anterior, evaluamos como acertada la separación del sistema en dos áreas básicas, de naturaleza claramente distinta, esencialmente independientes en lo que a análisis se refiere. Estas son:

- **Entrada/Salida.** Le concierne todo lo relativo a importación y exportación de imágenes desde y hacia diversos dispositivos. En esta área es la aplicación la que controla el curso de acciones a realizar.
- **Interacción.** Le concierne todo lo relativo a recibir, interpretar y manejar los eventos desencadenados fuera de la aplicación durante su ejecución. En el curso del documento estableceremos el modelo de eventos elegido para cumplir este fin.

Cada uno de estos puntos será tratado en detalle e independientemente en las siguientes secciones para llegar a una unificación al final. El lector debe ser consciente que lo expuesto a continuación es susceptible de sufrir modificaciones en pro de la coherencia del modelo total.

2 Entrada/Salida

Comenzaremos esta sección explicando exactamente que entendemos por entrada/salida (E/S).

El problema de *entrada* consiste en la obtención de una imagen BICOTI-I [1] desde virtualmente cualquier dispositivo(Disco, Scanner, Memoria, Red, etc.) y en cualquier formato(GIF, JPG, TWAIN, BICOTI, etc.).

Entendemos por *salida* la exportación de la imagen BICOTI-I hacia cualquier dispositivo y formato.

Entrada/Salida es en realidad un abuso del lenguaje, sería más preciso hablar de importación y exportación, dado que existe una etapa de lectura/escritura, combinada con otra de conversión de formato.

Basados en la precisión anterior, surgen naturalmente las familias de clases en que nos basaremos para resolver estos problemas. A nuestro entender dichas familias son:

- **Dispositivos.** Los dispositivos*(devices)* son las clases encargadas de encapsular todos los detalles relativos a los dispositivos físicos de E/S. A este nivel nuestro principal interés es leer y/o escribir en los dispositivos físicos en forma estándar, con el fin de independizar la lógica de las aplicaciones de aspectos tales como: fuentes de datos, características particulares de los sistemas operativos, etc.

- **Mapeadores.** Sobre los mapeadores*(mappers)*, recae la responsabilidad de realizar las conversiones de formato desde y hacia BICOTI-I .

[1] Al referirnos a *imagen bicoti* estamos hablando del formato definido en BICOTI-I para el procesamiento de imágenes.

Destacamos que así como ambas familias son ortogonales[2], estas cooperan para realizar todo proceso de E/S que involucre formatos y dispositivos ya definidos.

2.1 Dispositivos

El siguiente esquema propone una jerarquía de clases correspondientes a los dispositivos.

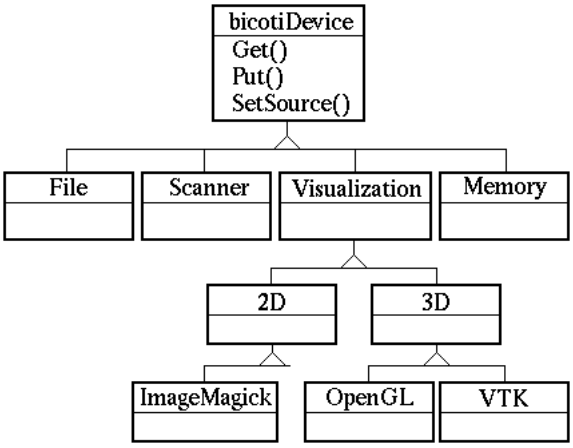

Figura 1: Los Dispositivos

Todos los *devices* comparten una interface básica que esencialmente permite leer *(get)* y escribir *(gut)* en ellos. Aquí aparecen como ejemplos: Archivos *(Files)*, Escáner *(Scanner)*, Memoria *(Memory)* y Visualización *(Visualization)*. En algunos (como *File*) es posible leer y escribir, en otros (como *Scanner*) solo es posible leer, así mismo en Visualization únicamente podremos escribir. Las subclases agregarán métodos específicos para cada tipo de dispositivo. En resumen lo esquematizado muestra una jerarquía de interfaces. Es posible que muchas de las clases concretas involucren detalles tales como: características del sistema operativo o del dispositivo físico usado. Durante la visualización el usuario podría eventualmente capturar el estado del dispositivo y recuperar por ejemplo la posición de la cámara si esta fuera modificada.

[2]Implica independencia de las familias de clases, tanto desde el punto de vista abstracto como práctico. Se traduce en la posibilidad de agregar o quitar implementaciones dentro de una familia en forma absolutamente independiente de la otra.

2.2 Mapeadores

El siguiente esquema propone una jerarquía de clases correspondientes a los mapeadores. El propósito de los mapeadores *(mappers)* es convertir desde y

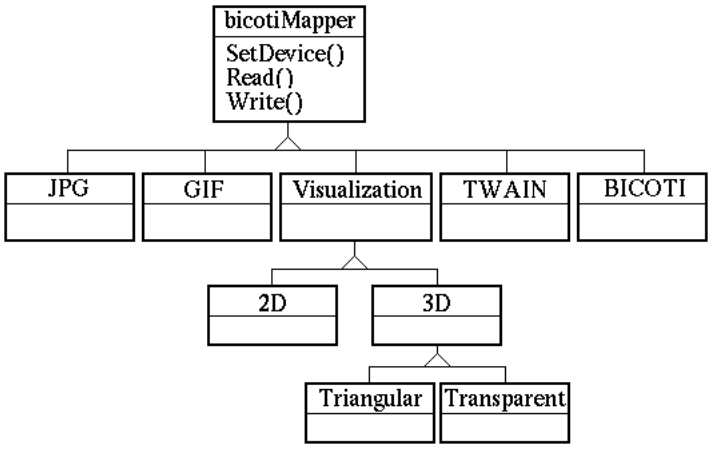

Figura 2: Los Mapeadores

hacia el formato de imagen BICOTI-I , imágenes en cualquier formato gráfico. Se ven como ejemplos JPG, GIF, TWAIN, BICOTI y Visualization. A este nivel se han abstraído las fuentes de datos (encapsuladas en los *devices*), evitando por ende la dependencia de los dispositivos físicos y/o sistemas operativos subyacentes. La idea es que los *mappers* usan a los *devices* como drivers y por lo tanto los detalles específicos de implementación de los métodos en los *mappers*, solo involucran problemas relacionados con el formato en particular.

Los primeros tres ejemplos no merecen mayores comentarios, son tres formatos estándar ampliamente difundidos. La existencia del formato de conversión BICOTI puede parecer algo extraño en principio, sin embargo permite fácilmente extender la ortogonalidad incluso hasta el nivel de las aplicaciones. Como ejemplo concreto pensemos en dos aplicaciones, *aplic1* y *aplic2* independientes, interactuando de la siguiente manera: *aplic1* lee una imagen de disco en formato JPG, realiza un cierto procesamiento y produce como salida una imagen BICOTI-I ; posteriormente *aplic2* recibe esta imagen BICOTI-I y realiza su propio procesamiento. Este esquema es realizable

en forma natural dada la existencia de *bicotiDeviceMemory* y *bicotiMapper-BICOTI*. Ilustramos este ejemplo mediante la **Fig.3**.

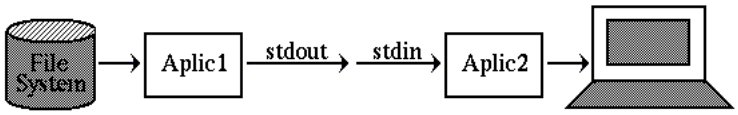

Figura 3: El Mapper Bicoti

Otro caso destacable es el *bicotiMapperVisualization* cuyo cometido es convertir la imagen BICOTI-I a elementos de algún modelo de visualización. Un caso que deja clara la idea es la visualización 3D. Las bibliotecas gráficas en general permiten incrustar elementos en una escena, el propósito de esta subfamilia de *mappers* es generar este tipo de elementos a partir de una imágen digital. Por ejemplo si la imagen BICOTI-I fuera una esfera sólida, un *mapper* de este tipo podría hallar un conjunto de puntos sobre la superficie y agruparlos en triángulos para entregar a la biblioteca de visualización.

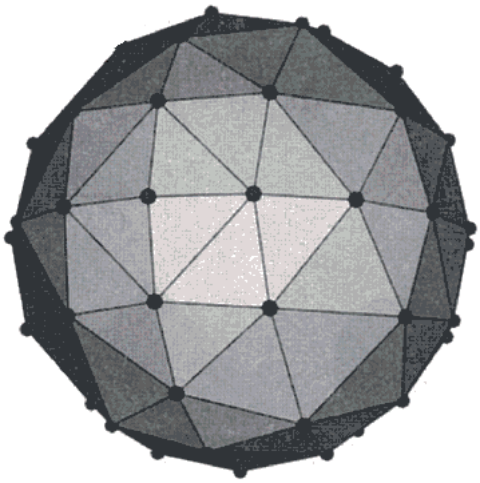

Figura 4: Una representación sólida de la esfera

La anterior no es la única representación espacial posible a partir de los puntos calculados. En este caso hemos elegido un aspecto sólido, pero podríamos haber optado por un *wireframe*[3]. Ver **Fig.5**.

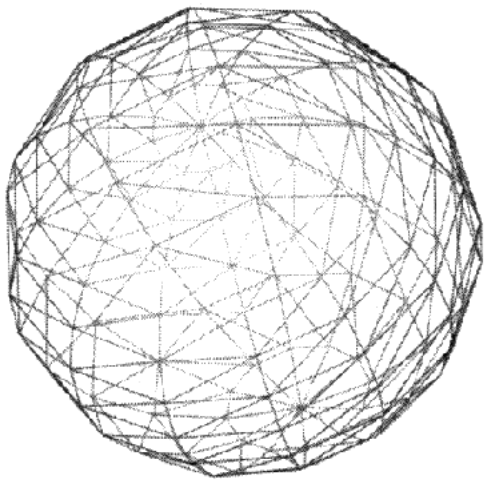

Figura 5: Una representación wireframe de la esfera

Lo que es importante destacar es que al *mapper* solo le competen los detalles de representación de la imagen (en este caso encontrar los puntos); en este ejemplo es el *device* el encargado de interpretar las directivas de iluminación, textura, etc. relacionadas estrictamente con el medio físico. **Notar que en realidad este dispositivo debe implementar métodos de mucho más alto nivel que *Get* o *Put* y es un candidato por ende a encabezar una jerarquía separada**.

[3]Una representación como malla de alambre.

2.3 Ejemplo de I/O

Antes de evaluar la conveniencia del modelo I/O ilustraremos el esquema de funcionamiento propuesto mediante un ejemplo[4].

El ejemplo levanta desde el directorio ~usuario/pictures la imagen formato BMP *heidi.bmp*, la carga en una imagen BICOTI-I , realiza cierto procesamiento y la graba en *heidi2.bmp*.

```
// Inicializacion del dispositivo
bicotiDevice * ptr_device;
ptr_device = new bicotiDeviceFile();
ptr_device->SetSource("file://home/usuario/pictures/heidi.bmp");

// Creacion de la imagen vacia
bicotiImageImplementation2D<char> * ptr_image;
ptr_image = new bicotiImageImplementation2D<char>(0);

// Declaracion e inicializacion del mapper
bicotiMapperBMP<char> mapper;
mapper.SetDevice(ptr_device);
mapper.SetImplementation(ptr_image);

// Lectura, Procesamiento y Escritura de la imagen
mapper.Read();
.........................
...(Procesamiento).......
.........................
ptr_device->SetSource("file://home/usuario/pictures/heidi2.bmp");
mapper.Write();
delete ptr_device;
delete ptr_image;
```

[4]Seguramente las clases finales no coincidan con lo mostrado en estos ejemplos, por lo prematura de los mismos. Aún así son un aporte importante a la claridad.

2.4 Evaluación del Modelo I/O

A la hora de evaluar las ventajas del modelo es bueno refrescar los objetivos de diseño planteados en [Sod99]. Recordemos que lo que buscábamos era lograr:

- Transparencia.
- Flexibilidad.
- Alto grado de Interacción.
- Escalabilidad.
- Portabilidad.

De momento dejaremos de lado Interacción; será tratada en futuras secciones. Así mismo remarcamos que la evaluación siguiente se limitará al modelo I/O presentado anteriormente.

1. **Portabilidad.** ¿Que puede afectar la portabilidad de nuestro sistema I/O? Obviamente los detalles específicos de los sistemas operativos y dispositivos físicos a usar. Se desprende de lo anterior que no deberían existir problemas de portabilidad vinculados a los *mappers*, dado que justamente estos realizan conversiones entre formatos estándar. Donde si existirán problemas relacionados a la portabilidad es en los *devices*. La idea claro está, es confinarlos a estos. ¿Como lograr este cometido? Nuestra solución se basa en ver los dispositivos a través de un conjunto de interfaces bien definidas, que deben ser usadas en el desarrollo de las aplicaciones. De esta forma la aplicación debería funcionar independientemente de la implementación concreta del dispositivo; dígase: "Funcionar en forma independiente de los detalles específicos del ambiente", siempre y cuando existan implementaciones de los dispositivos, compatibles con los detalles del caso.

2. **Flexibilidad.** Un sistema concebido hasta ahora como un par de familias ortogonales y fuertemente jerarquizadas, es sin duda una excelente solución si de crecer estamos hablando. Pensemos en el problema de incluir un nuevo *mapper* para reconocer formato BMP. Esta tarea se limitaría a agregar una hoja en el árbol de herencia, implementando los métodos definidos en la interface correspondiente, sin tener que cuestionarse siquiera la existencia de otros componentes del sistema.

3. **Escalabilidad.** En complemento resulta cómodo extender mediante especializaciones la interface establecida en principio. Pensemos en *bicotiDevice*, la posibilidad de extender la interface para incluir elementos típicos de visualización, o la de sobre-extenderlo para destacar características particulares de la dimensión es un ejemplo del grado de escalabilidad alcanzable.

4. **Transparencia.** La posibilidad de resolver los problemas a través de un pequeño grupo de interfaces abstractas, logra en sí mismo el objetivo de transparencia tal cual fue planteado en [Sod99].

3 Interacción

Frecuentemente en las aplicaciones de tipo gráfico es deseable poder hacer algo más que simplemente desplegar imágenes en la pantalla. Si el usuario deseara ser más que un mero espectador durante la ejecución y pretendiera por ejemplo: cambiar la perspectiva de su escena, mover objetos dentro de esta o simplemente marcar un punto de interés sin tener que recompilar; algún mecanismo que permita este diálogo es indispensable. Actualmente esta necesidad es común a la enorme mayoría de la aplicaciones y claro está Bicoti no es la excepción. El mecanismo unánimemente aceptado (independientemente del modelo usado) para lograr este cometido es el de *eventos*. Antes de pasar a explicar en más detalle el modelo propuesto, ilustraremos el mecanismo de las aplicaciones dirigidas por eventos.

La idea detrás de este mecanismo es que la aplicación este esperando por ciertos sucesos declarados de interés, para ejecutar alguna acción específica a cada uno. Estos sucesos en general pueden ocurrir en cualquier orden y momento ya sea por una acción del usuario u otra fuente. Ejemplos de eventos comunes pueden ser:

- Presionar una tecla, elegir un ítem en un menú o presionar el botón del mouse sobre cierto componente gráfico.

- También pueden existir otros eventos como recibir un mensaje del sistema operativo, indicando que llego una trama a la tarjeta de red o que el sistema se va a re-iniciar.

En bicoti nos concentraremos en el primer tipo de eventos, es decir aquellos que involucren interacción con nuestro modelo de imagen.

3.1 El modelo de eventos

Anteriormente hablamos de la importancia actual de las aplicaciones dirigidas por eventos. En consecuencia no es extraño que como de hecho sucede, existan extensas y exhaustivas aproximaciones propuestas para modelar el problema. Nuestro equipo investigó el estado del arte en el tema y optó por el *modelo de delegación* usado entre otros por las últimas versiones del lenguaje Java^{TM} como modelo general de eventos.

El modelo de delegación presupone la existencia de tres tipos de objetos:

- Un conjunto de eventos*(events)* que pueden suceder y para los cuales es de interés registrar información de estado que detalle para cada tipo de evento los aspectos particulares del caso. Ejemplos de eventos de interés podrían ser: un evento de mouse o uno de teclado. En el primer caso es importante conocer la ubicación del puntero del mouse en la pantalla y/o cual botón se oprimió. En el segundo lo importante saber cual tecla se oprimió o liberó.

- Un conjunto de fuentes*(sources)* de eventos que los disparan, en general frente a una acción del usuario. Fuentes de eventos puede ser: la ventana de diálogo donde se visualiza la imagen, o un actor en esta.

- Un conjunto de observadores*(listeners)* que se subscriben a las fuentes de eventos de su interés y son comunicados oportunamente por estas con el evento adecuado al caso. Es responsabilidad de cada observador implementar las acciones a tomar frente al evento informado.

La jerarquía básica de eventos tendría la siguiente forma.

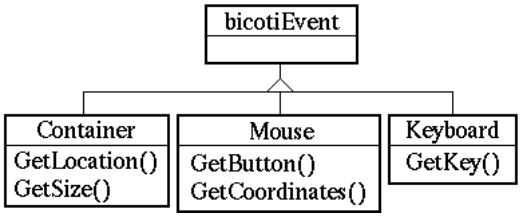

Figura 6: Los Eventos

Los tres son muy claros. En primer lugar el *Container* que no es más que el escenario donde están montados los actores, puede ser movido o cambiarse

su tamaño. Un evento *Mouse* debe informar el estado del mouse en el instante en que se dispara el evento. Un evento *Keyboard* que se dispara si una tecla se oprimió o soltó y debe capturar cual fue.

¿Cuales son nuestras fuentes de eventos? Entendemos que podría ser cualquier actor o la escena misma. Surge en consecuencia la necesidad de diferenciar los distintos elementos de una imagen ya que los requerimientos de interacción de los actores dentro de una misma escena pueden ser muy diversos. En resumen se deben poder individualizar los actores lo que deriva en la existencia de la clase actor.

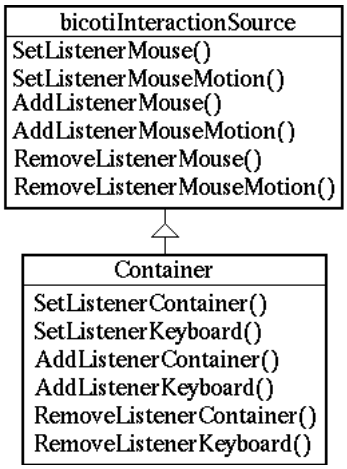

Figura 7: Las Fuentes de Eventos

La figura anterior muestra la jerarquía de estas fuentes de eventos. Un objeto *bicotiInteractionSource* puede ser por lo tanto: la escena *(Container)* o un actor particular *(bicotiActor)*. El *container* puede disparar cualquier tipo de evento de los previstos, mientras que el actor solo dispara eventos del mouse. El actor debe contener la información que le permita al visualizador presentarlo dentro de la escena (una vez fijada su posición) independientemente de la biblioteca gráfica.

El último de los componentes involucrados en el modelo de delegación es el observador *(Listener)* responsable de procesar los eventos disparados por las fuentes ante las que está acreditado.

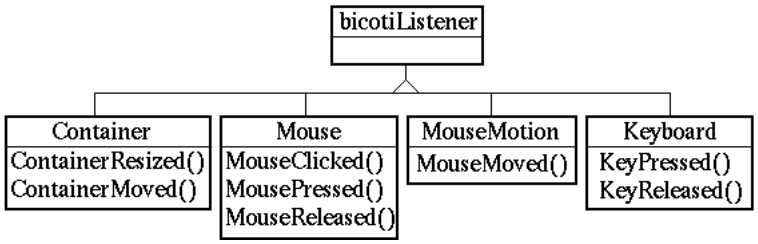

Figura 8: Los Observadores

El modelo de delegación provee un mecanismo: flexible, extensible, fácil de entender, que permite la creación de código de manejo de eventos robusto y poco propenso a error. Un análisis extenso de las virtudes del modelo contrastadas con modelos anteriores se encuentra en [Inc99] y más en detalle en [Ham97].

4 Arquitectura

La arquitectura unificada es la mostrada en la **Fig.9**. La visualización se ha retirado de la jerarquía de dispositivos ya que inspirados en los paquetes estándar de visualización [Lor98], [Ake93], [Phi96] concluimos que era necesaria una interface mucha más rica que la provista por los *devices*. Como los eventos se disparan a través de la visualización se hace heredar a esta última de *bicotiInteractionContainer*.

Además se ha hecho expresa la existencia de los actores*(bicotiActor)* que se construyen mediante el *bicotiConverter* a partir de la *bicotiImageImplementation*. Notar que se ha separado el conversor de los *mappers* porque ya no estamos simplemente escribiendo en un *bicotiDevice*. *bicotiVisualization* nos da una interface para controlar la visualización. Esto es: situar los actores dentro de la escena, la forma de representar estos, las luces, cámaras, texturas, etc.

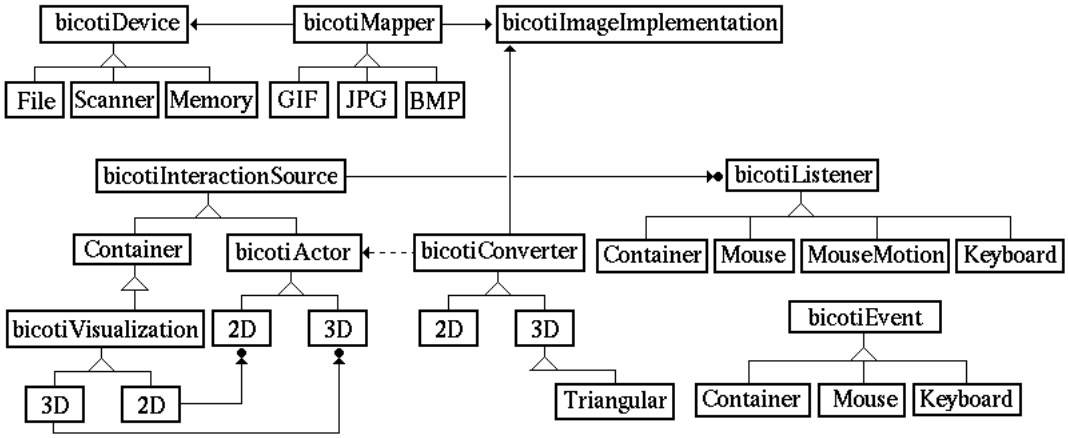

Figura 9: La Arquitectura Final

4.1 Un Ejemplo Final

Como ejemplo final presentaremos lo que puede llegar a ser un caso típico de aplicación, según lo discutido con quienes serán en definitiva los usuarios del sistema. Este consiste en recuperar de disco dos imágenes posiblemente asociadas a cortes de un tomógrafo, con las cuales se desea formar una imagen 3D conjunta. Una idea posible es crear una imagen bicoti de tres dimensiones con cada corte asociando un espesor a los mismos, presentarlos en una escena donde el usuario pueda moverlos, girarlos, etc. hasta hacerlos coincidir para posteriormente recuperar el estado final con miras a unificar los cortes mediante un procesamiento en $B^{I}C_{O}TI$-I .

```
// Leo de disco ambos cortes
bicotiImageImplementation2D<char> * ptr_image1;
bicotiImageImplementation2D<char> * ptr_image2;
ptr_image1 = new bicotiImageImplementation<char>(0);
ptr_image2 = new bicotiImageImplementation<char>(0);
bicotiDevice * ptr_device;
ptr_device = new bicotiDeviceFile();
ptr_device->SetSource("file://home/corte1.bmp");
bicotiMapperBMP<char> mapper;
mapper.SetDevice(ptr_device);
mapper.SetImplementation(ptr_image1);
```

```
mapper.Read();
ptr_device->SetSource("file://home/corte2.bmp");
mapper.SetImplementation(ptr_image2);
mapper.Read();
delete ptr_device;

// Cargo ptr_image3 y ptr_image4 con los cortes y su ancho
bicotiImageImplementation3D<char> * ptr_image3;
bicotiImageImplementation3D<char> * ptr_image4;
.........................
..Convierto a 3D.........
.........................
delete ptr_image1;
delete ptr_image2;

// Creo los actores y cargo en ellos la informacion adecuada
bicotiActor3D * ptr_actor1;
bicotiActor3D * ptr_actor2;
bicotiConverter3DTriangular converter;
converter.SetImplementation(ptr_image3);
ptr_actor1 = converter.CreateActor();
converter.SetImplementation(ptr_image4);
ptr_actor2 = converter.CreateActor();
delete ptr_image3;
delete ptr_image4;

// Creo el visualizador para VTK
bicotiVisualization3DVTK visualizator;
....(Seteos de camara, luz, etc)....
visualizator.AddPickableActor(ptr_actor1,x1,y1,z1,phi1,lambda1,etc1);
visualizator.AddPickableActor(ptr_actor2,x2,y2,z2,phi2,lambda2,etc2);

// Supongamos que el usuario implemento un bicotiMouseListener que se
// encargue de interpretar los eventos del mouse y generar en el
// visualizador los giros o movimientos adecuados
ptr_actor1->SetMouseListener(mouse_listener1);
ptr_actor2->SetMouseListener(mouse_listener2);
```

```
// Comienza la visualizacion e interaccion
visualizator.Render();
.....................................
...Se recupera el estado final........
... y se crea la imagen nueva ........
.....................................
delete ptr_actor1;
delete ptr_actor2;
```

Referencias

[Ake93] Mark Segal; Kurt Akeley. The opengl^{TM} graphics system: A specification (version 1.0). Technical report, Silicon Graphics; URL: http://www.sgi.com, 1993.

[Ham97] Graham Hamilton. Sun microsystems javabeans^{TM}. Technical report, Sun Microsystems, URL: http://java.sun.com/beans, 1997.

[Inc99] Sun Microsystems Inc. Java^{TM} awt: Delegation event model. Technical report, Sun Microsystems Inc., 1999.

[Lor98] Will Schroeder; Ken Martin; Bill Lorensen. *The Visualization Toolkit, An Object-Oriented Approach to 3D Graphics.* Prentice Hall PTR, ISBN 0-13-954694-4., 1998.

[Phi96] Foley; Van Dam; Zeiner; Hughes; Phillips. *Introducción a la Graficación por Computador.* Addison-Wesley Iberoamericana, ISBN 0-201-62599-7., 1996.

[Sod99] Marcelo Alarcón; Claudio Risso; Carlos Soderguit. Análisis primario de requerimientos. bicoti-ii. Technical report, Facultad de Ingeniería, 1999.