CAPÍTULO 4
Diseño e Implementación

El presente capítulo tiene como principal finalidad, mostrar las principales consideraciones de diseño e implementación que se tomaron en cuenta para la implementación del software del modelo de vértices extremos.

El diseño general del sistema implementado se encuentra enfocado hacia la programación orientada a objetos y, como el nombre de la tesis lo indica, el lenguaje de programación seleccionado para la implantación del sistema fue Java, por lo que en algunas ocasiones se hará referencia al mismo.

Primeramente se presentan algunos aspectos relacionados con el diseño general del sistema, para pasar a algunos aspectos que se consideraron durante el proceso de visualización. Finalmente, se comentan algunos de los aspectos relacionados al editor de pseudo poliedros ortogonales que se integró a la aplicación.

4.1 Consideraciones de diseño.

4.1.1 Clases.

Existen diversas maneras de enfrentarse con algún problema y proponer alguna solución basada en un enfoque de software. Uno de los enfoques más ampliamente utilizados en la actualidad para la solución de problemas que involucren software, es el enfoque que proporciona el paradigma la programación orientada a objetos.

Se puede decir de manera general que, en la programación orientada a objetos, el dominio del problema está caracterizado como un conjunto de objetos, los cuales poseen un estado y comportamiento determinados, por lo que gran parte del éxito de este paradigma se centrará en la adecuada identificación e interacción de los objetos involucrados. Para ello, y siguiendo los lineamientos clásicos de la ingeniería del software, a partir de la planeación inicial del sistema, se estará incurriendo continuamente en el análisis y diseño del mismo para ir realizando un refinamiento progresivo del sistema hasta obtener los resultados esperados.

El identificar las clases involucradas, observar si algunas de ellas existen ya en las bibliotecas del lenguaje a utilizar, la implementación de las clases no disponibles así como de las clases de soporte, son algunos de los aspectos intermedios a considerar para la implementación, aspectos todos ellos tomados en cuenta para el desarrollo del sistema propuesto.

Siguiendo los lineamientos propuestos en [Pressman 2001], en la Figura 4-1 se presentan las principales clases involucradas en el sistema. La figura presenta un diagrama general del análisis del modelo del software implementado, ilustrando el diagrama de

generalización/especialización de las clases, así como también los diagramas de composición/relación de las clases, considerando que una relación existe entre cualesquiera dos clases que estén conectadas.

El diseño global de las clases involucra un paquete: el paquete *evm*. Un paquete absorbe, por decirlo de alguna manera, la representación general de lo que se modela, es el nivel de representación más abstracto y general del modelo de software. En Java, una agrupación de clases relacionadas recibe también el nombre de paquete y, dado que la implementación del sistema se realizó en este lenguaje, la totalidad de las clases desarrolladas con excepción de la clase que constituye el vínculo inicial y de la cual se hablará un poco más adelante, se encuentran incluidas dentro del paquete *evm*[1].

La clase principal y una de las más importantes del modelo es precisamente esa que lleva su nombre, la clase *EVM*. Esta clase contiene las estructuras de datos y los algoritmos propuestos en [Aguilera 1998] para el modelo de vértices extremos, dicho de otra forma, implementa la funcionalidad del modelo *EVM*.

La clase *OUoDBM* implementa las características necesarias del modelo de cajas disjuntas mencionado en el capítulo anterior. La principal utilidad de dicho modelo se hace patente en la visualización de pseudo poliedros ortogonales a través de *EVM* ya que éste, al ser un modelo que sólo almacena un subconjunto de la totalidad de los vértices (los vértices extremos) y no otras características topológicas útiles en la descripción total del pseudo poliedro como podrían ser las adyacencias de aristas definidas por los vértices, carece por sí mismo de la información útil y/o necesaria para la visualización. Los detalles de la visualización así como otras características relacionadas a la misma se discutirán más adelante en este mismo capítulo.

[1] Vale la pena mencionar que en realidad el nombre final del paquete es *mx.udlap.evm*. Esto debido a la forma en que se firman los paquetes en Java, pero en esencia el concepto sigue siendo el mismo.

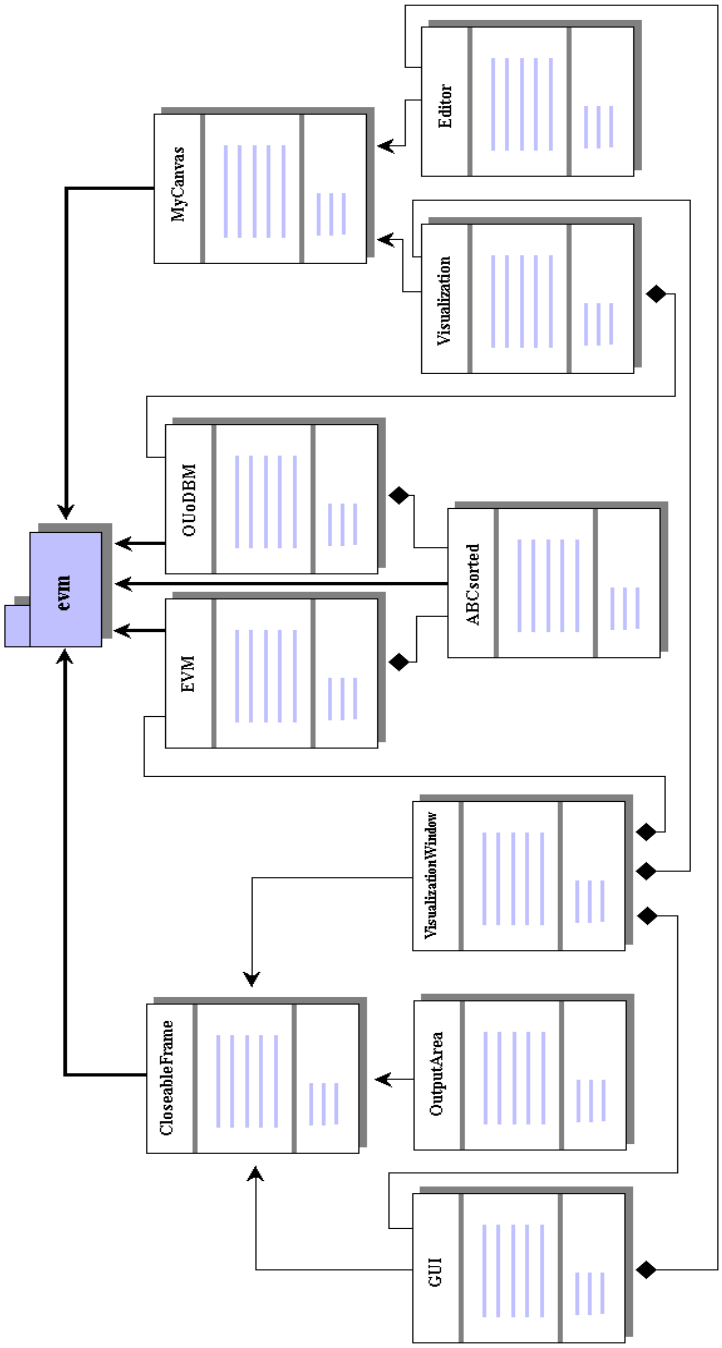

Figura 4-1Análisis del modelo, presentando los diagramas de generalización/especialización de las clases, así como el diagrama de composición de las mismas y la referencia al paquete en el cual se encuentran definidas.

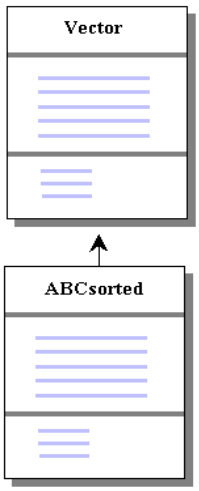

Figura 4-2 Diagrama de generalización/especialización de la clase ABCsorted. La clase Vector pertenece al conjunto de clases de la biblioteca de Java: java.util.Vector.

La clase *ABCsorted* constituye la estructura de datos esencial para el modelo EVM. La estructura de datos que esta clase implementa es básicamente un arreglo de objetos que representan a los elementos a almacenar en la estructura de datos. Dichos elementos son objetos de la clase *Point3D* (no presentada en la Figura 4-1), la cual es una abstracción de un punto tridimensional en base a las coordenadas A, B y C.

Desde el punto de vista de eficiencia, es importante que ésta estructura de datos se comporte de manera dinámica, por lo que una representación de un vector de objetos dinámico parece el candidato adecuado para esta tarea. Una clase en Java que proporciona estas características es la clase *Vector*, la cual es utilizada para la implementación de ésta estructura de datos, de hecho la clase *ABCsorted* es en realidad una especialización de la clase *Vector* proporcionada por la biblioteca de clases de Java. En la Figura 4-2 se presenta el diagrama de generalización/especialización de clases para la clase *ABCsorted*.

Otra clase es *MyCanvas* la cual es también una especialización de la clase *Canvas* perteneciente a la biblioteca de clases de Java y es, como su nombre lo indica, un lienzo sobre el cual dibujar, por lo que como lo ilustra la Figura 4-3 la clase es especialmente útil

en operaciones que involucren el dibujo, como lo es el caso de la clases *Editor* y *Visualization*.

Las clases *Editor* y *Visualization* constituyen respectivamente un apartado especial, ya que involucran aspectos que deben ser mencionados con mayor amplitud que las clases hasta el momento descritas, por lo que se tratarán más adelante cada una de ellas en secciones separadas.

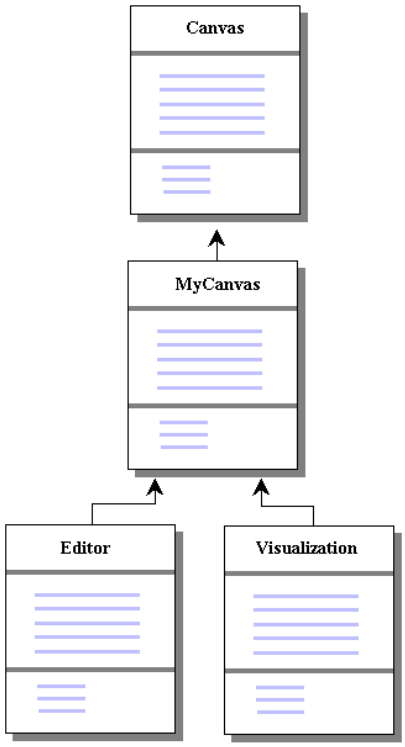

Figura 4-3 Diagrama de generalización/especialización de la clase MyCanvas y las subclases Editor y Visualization. La clase Canvas pertenece al conjunto de clases de biblioteca de Java: java.awt.Canvas.

La clase *CloseableFrame* constituye el comportamiento y el marco de referencia general para las clases que de ella derivan: *GUI*, *OutputArea* y *VisualizationWindow*. Como puede observarse en la Figura 4-4, la clase *CloseableFrame* es también una derivación de la clase *Frame* de la biblioteca de clases de Java.

OuputArea es una clase que integra los elementos necesarios para el despliegue de información textual dentro de una ventana por parte de la aplicación desarrollada. Esta información es por ejemplo, la representación interna del objeto en el modelo EVM, en el OUoDBM o en BRep. En el capítulo referente a los resultados se detallará un poco más este aspecto.

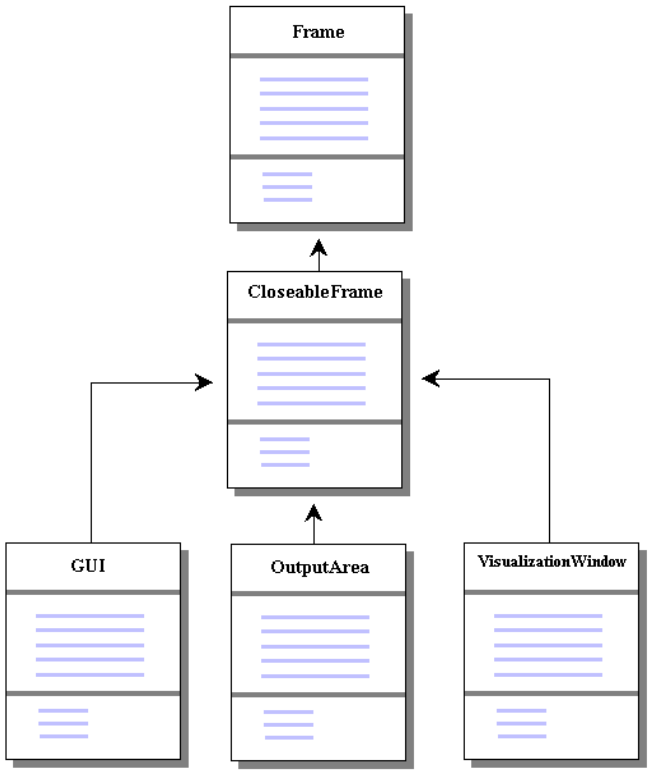

Figura 4-4 Diagrama de generalización/especialización de la clase CloseableFrame y las subclases GUI, OutputArea y VisualizationWindow. La clase Frame pertenece al conjunto de clases de biblioteca de Java: java.awt.Frame.

La clase *VisualizationWindow* está íntimamente relacionada con la clase *GUI* ya que es en ella en donde se capturan y procesan los eventos y las operaciones a ser efectuadas sobre el objeto *evm* en curso. Como su nombre lo sugiere, la clase *VisualizationWindow* engloba y administra los elementos necesarios e indispensables para la manipulación de una ventana que permita la visualización de objetos *evm* a través de la inclusión de un objeto de

la clase *Visualization*. Es importante mencionar que por cada objeto *evm* procesado, se genera una nueva instancia de la clase *VisualizationWindow*.

Finalmente, la clase *GUI* está compuesta de los elementos que integran la interfaz principal del sistema. En ella se capturan y procesan los eventos de apertura (carga) de archivos válidos *evm*, *BRep* y *Octrees,* así como las operaciones booleanas a ser realizadas entre estos objetos. De cierta manera puede decirse que la clase *GUI* constituye el punto de partida de interacción con el resto del sistema, ya que en realidad la clase *EVMapp* (véase la Figura 4-5), es sólo una clase ancla que tiene como único objetivo proporcionar un mecanismo de instanciación externo al paquete *evm*. Esto último fue pensado así con la finalidad de proporcionar una aplicación completamente funcional que integrara los elementos y características del modelo EVM implementado, sin embargo, la clase *EVM* que es como ya se mencionó la que implementa las características esenciales del modelo homónimo, puede ser utilizada de manera independiente de la aplicación, con el objetivo de proporcionar un esquema de software con clases reutilizables, de hecho, la clase *GUI* en combinación con *VisualizationWindow* llevan a cabo sus tareas en base al intercambio de mensajes hacia los objetos *EVM* que contienen.

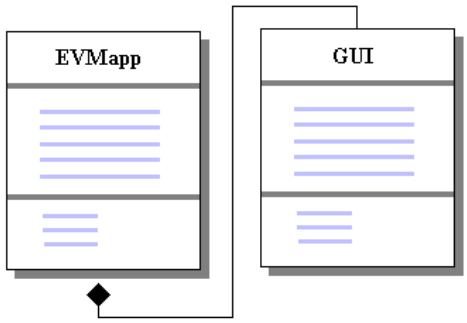

Figura 4-5 Diagrama de composición de la clase principal del sistema implementado. La clase EVMapp, constituye el punto de partida del sistema.

4.1.2 Formato de archivos.

En esta sección se describen algunos de los formatos de archivos soportados por la aplicación. Todos los archivos de entrada para la aplicación son archivos de texto plano, esto es, sin formato. Cada uno tiene características especiales y deben cumplir ciertas condiciones. El formato de los archivos, así como sus características más importantes son descritos a continuación.

4.1.2.1 Formato de archivos EVM.

Se ha optado en este tipo de archivos la convención de agregarles la extensión "evm", sin embargo esto no es condicionante, ya que mientras el contenido del archivo sea válido, no se tendrá ningún problema para su lectura independientemente de su extensión.

En la Tabla 4-1 se muestra una descripción comparativa de las especificaciones para un archivo válido dentro del modelo EVM implementado, así como un sencillo ejemplo de un archivo que define un cubo centrado en el origen.

Tabla 4-1 Formato de los archivos de texto del modelo EVM implementado.

Especificaciones	**Ejemplo**
Ordenación	XYZ
Dimensión	3
Vértice extremo 1	-100 -100 100
Vértice extremo 2	100 -100 100
Vértice extremo 3	100 -100 -100
Vértice extremo 4	-100 -100 -100
.	-100 100 -100
.	-100 100 100
.	100 100 100
Vértice extremo *n*	100 100 -100

Lo que en esencia identifica y distingue a un archivo del modelo EVM es el primer renglón: el renglón de ordenación. En base a las especificaciones mismas del modelo, éste renglón puede contener alguno de los seis tipos de ordenamiento válidos en EVM (XYZ, XZY, YXZ, YZX, ZXY, ZYX), aunque usualmente es XYZ para el caso 3D y ZXY para el caso 2D, esto último sólo para mantener una homogeneidad en el modelo.

El segundo renglón determina la dimensión el objeto *evm* y debe ser un número entero positivo entre 1 y 3. Los siguientes renglones definen los vértices extremos del objeto *evm* representado en el archivo y pueden ser números enteros o de punto flotante, aunque la representación interna de los vértices es en números de punto flotante.

Por último, obsérvese que la única información contenida en los archivos del modelo EVM es la hasta ahora descrita, la cual consiste sólo de tres partes: ***tipo de ordenación***, ***dimensión del objeto*** y ***vértices extremos***, por lo que será responsabilidad del usuario del sistema respetar el formato de los archivos y seguir los lineamientos descritos. Al respecto, se ha integrado en la aplicación un editor de poliedros ortogonales con la particularidad de generar archivos *evm* válidos, pero no es la única forma de crearlos. Este editor junto con las características que contiene se describe más adelante en este mismo capítulo.

4.1.2.2 Formato de archivos BRep.

La aplicación permite leer archivos diferentes a los especificados en el apartado anterior. Esto quiere decir que es posible abrir archivos que contengan información de objetos representados en BRep, siempre y cuando cumplan con las características del modelo EVM y que los objetos representados en los archivos sean pseudo poliedros ortogonales.

Para los archivos en formato BRep se establece la convención de agregarles la extensión "brep". Nuevamente vale la pena hacer la mención de que mientras el contenido del archivo sea válido, no se tendrá ningún problema para su lectura independientemente de su extensión.

Tabla 4-2 Formato de los archivos de texto BRep.

Especificaciones	Ejemplo
Tipo	BRep
Dimensión	3
Cara 1	V_1
	V_2
.	.
.	.
.	.
.	V_n
	. . .
Cara *n*	V_1
	.
	.
	.
	V_n

En la Tabla 4-2 se muestra una descripción comparativa de las especificaciones para un archivo válido dentro del modelo EVM en formato BRep.

La representación BRep que se utiliza es la representación jerárquica en la que un objeto está definido en término de las caras que lo componen, esto es, la secuencia de vértices que constituyen alguna de las fronteras del objeto.

En este tipo de representaciones, los vértices que componen el contorno de una de las caras del objeto, están ordenados en sentido contrario a las manecillas del reloj, mientras que aquellos que definen un agujero (si lo hubiera) dentro del objeto, están definidos en el sentido de las manecillas del reloj.

4.1.2.3 Formato de archivos Octree.

También es posible abrir archivos que representen pseudo poliedros ortogonales definidos en una representación lineal de *octree* clásico.

La estructura del archivo es una secuencia lineal de caracteres representando los nodos del *octree* en donde, siguiendo la convención establecida para los *octree* clásicos, se tiene la siguiente clasificación para las letras:

G: representa un nodo gris.

B: representa un nodo negro.

W: representa un nodo blanco.

La forma en que este tipo de estructura es representada en el archivo puede ser en una estructura lineal agrupada por paréntesis, sin embargo esto no es una condición necesaria, ya que la misma estructura puede carecer de ellos y representar el mismo objeto.

Tabla 4-3 Formato de los archivos de texto Octree.

Especificaciones	Ejemplo
Tipo	Oct3
Estructura de representación lineal	G(BBBWBWWW)

En la Tabla 4-3 se muestra una descripción del contenido del archivo para un objeto EVM representado en una estructura lineal de un *octree* clásico.

Por último cabe mencionar que, al igual que en los casos anteriores, se ha tomado la convención de establecer la extensión “oc3” para este tipo de archivos, sin embargo, lo que realmente identifica a archivos de este tipo como válidos es el tipo indicado en la Tabla 4-3.

4.2 Visualización.

La determinación de superficies visibles o eliminación de caras ocultas es un problema intrínsecamente complejo, ya que los requerimientos y los cálculos involucrados en dicha determinación son generalmente laboriosos y costosos desde el punto de vista computacional, dado que los algoritmos desarrollados consumen considerable tiempo de cómputo y/o memoria.

El grueso de las características de visualización incluidas en el modelo de software desarrollado se encuentra en la clase *Visualization*.

La principal dificultad en la visualización de objetos *evm* para que parezcan sólidos, es la limitada información que se tiene del objeto a representar debido a que esto es lo que constituye precisamente una de las características más importantes del modelo EVM. Así, se cuenta con el tipo de ordenamiento, la dimensión y los vértices extremos, los cuales son un subconjunto (generalmente menor) de la totalidad de los vértices del objeto. Esto en sí mismo representa una dificultad, ya que no se cuenta con algún otro tipo de información topológica útil en la visualización, de hecho, y como ya se mencionó, en la mayoría de las veces ni siquiera se cuenta con todos los vértices, sólo se tienen los vértices extremos.

Presentar el objeto EVM a través de un modelo de alambres resulta relativamente sencillo, ya que a partir de ordenaciones de los vértices, se pueden obtener los *brinks* paralelos a los ejes x, y y z, éstas ordenaciones son, respectivamente *YZX*, *ZXY* y *XYZ*. Pero si lo que se desea visualizar es una representación de un objeto definido en EVM para que parezca un sólido, las cosas se complican.

El modelo de cajas disjuntas (OUoDBM) resultó particularmente útil en la representación del objeto *evm* como un sólido, ya que permite la representación de éste en forma de cajas, mismas que en conjunto determinan al sólido representado en EVM.

El siguiente aspecto de consideración para la visualización tiene que ver con las superficies a dibujar, ya que si no se tiene cuidado con ello, el resultado en pantalla será un

objeto incoherente o fuera de lógica. En este sentido, se implementó una combinación de dos mecanismos clásicos dentro del campo de la determinación de superficies visibles:

1. Eliminación de caras posteriores (*back face culling*).
2. Memoria de profundidad (*z-buffer*).

Aunque por sí misma la técnica que utiliza el algoritmo de memoria de profundidad hubiera solucionado el problema de visualización dado que las caras anteriores cubrían a las posteriores, se consideró la manera de volverlo más eficiente debido a que como ya se mencionó, al ser este tipo de algoritmos ambiciosos en sus cálculos, el desempeño global del sistema podría verse afectado. Por esto último se decidió integrar también la técnica de eliminación de caras posteriores, la cual determina cuales de las seis caras de una caja son, en principio, visibles.

Es importante volver a mencionar que para la visualización del objeto como un sólido, se está empleando el modelo de cajas disjuntas, y en un momento dado sólo son visibles a lo más tres caras de las seis que determinan a una caja (Figura 4-6).

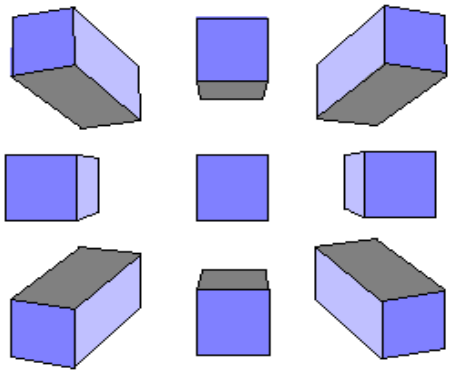

Figura 4-6 Nueve posibles vistas de una caja. Dependiendo del ángulo de visión, sólo es posible observar a lo más tres caras de las seis que componen a una caja.

Dependiendo del tipo de proyección (paralela o perspectiva) y de la opción del modelo (alambres o sólido) establecido para la visualización, será la elección del algoritmo

apropiado. Para el caso del modelo de alambres se dibujan como ya se dijo, los *brinks* correspondientes a *x*, *y*, *z* del objeto *evm*.

Para el caso de visualización del objeto *evm* como un sólido, de manera muy general, el algoritmo implementado es el siguiente:

```
Por cada caja contenida en el modelo de cajas disjuntas
    Obtener las seis caras de la caja
    Por cada cara de la caja
        Aplicarle las transformaciones respectivas (rotación, escalamiento, etc.)
            Si la cara es visible, entonces dibujarla
```

La sentencia dibujarla implica diversos tipos de procesamiento entre los que se encuentra la técnica de memoria de profundidad y el calculo del color, entre otros.

Se implementaron dos modelos de iluminación: *Lambert* y *Gouraud*.

En el modelo de iluminación de *Lambert*, las caras de cada una de las cajas se iluminan con diferente intensidad, pero dicha intensidad es la misma para toda la cara. Cabe hacer mención de que para determinar si una cara dada es o no visible se hizo uso de la normal de dicha cara y, en el modelo de iluminación implementado se hace también uso de esta normal, por lo que el cálculo de la misma es doblemente aprovechado tanto por el algoritmo de eliminación de caras posteriores como por el del modelo de iluminación.

Por otro lado, el modelo de iluminación de *Gouraud*, pinta cada una de las caras interpolando en cada píxel el color, por lo que la iluminación será más suave pero también más costosa. Cada una de las caras a dibujar es pintada en base a dos triángulos. En este modelo de iluminación es necesario calcular el color en cada uno de los vértices de los triángulos, considerando la forma en que la fuente de luz incide en dicho vértice, para poder hacer esto posible, se calcula una "normal" en cada vértice del triángulo definida como un vector que va de las coordenadas del centro de la caja a la cual pertenece dicha cara, a las coordenadas del vértice del triángulo. De esta manera se realiza el cálculo del color en cada vértice del triángulo para obtener un modelo de iluminación mucho más suavizado.

Finalmente debe decirse que el algoritmo de memoria en profundidad tiene un desempeño constante [Foley 1991] ya que, al aumentar el número de polígonos en un escena, se reduce el número de pixeles cubiertos por un solo polígono, además éste algoritmo no requiere de ningún tipo de ordenamiento como podría ser el caso de los algoritmos de ordenamiento por profundidad y de línea de barrido por mencionar algunos. En ocasiones se menciona que una desventaja del algoritmo de memoria en profundidad es su elevado requisito de memoria, este punto podría ser relativo, dado que algunos de los otros esquemas de visualización pueden tener estructuras de datos tan grandes como los *octrees* por ejemplo, que la memoria que las almacena podría tener requerimientos iguales o mayores a los del esquema de memoria en profundidad. Por otro lado la implementación realizada del algoritmo de memoria de profundidad es tal, que libera de manera explícita la memoria[2] cuando no se considera necesaria tenerla ocupada, esto ocurre al minimizar alguna de las ventanas de visualización o al conmutar de modelo de sólido al modelo de alambres por ejemplo.

4.3 Editor de pseudo poliedros ortogonales.

Dentro de la aplicación desarrollada del software que implementa al modelo EVM, se incluye un editor de pseudo poliedros ortogonales (PPO) que es, como su nombre lo indica, un editor que permite generar de manera relativamente sencilla, pseudo poliedros ortogonales. El editor de PPO se encuentra contenido dentro de la interfaz principal de la aplicación.

En la Figura 4-3 se ilustra la clase *Editor* y su posición dentro de la jerarquía de clases involucradas. La funcionalidad proporcionada por el editor de pseudo poliedros ortogonales se encuentra definida dentro de esta clase.

[2] La liberación explícita de memoria se realiza a través de una llamada a la máquina virtual de Java para "sugerirle" que ejecute su mecanismo de recolección de basura siguiendo los mecanismos y técnicas sugeridas en [Bloch 2001].

El editor cuenta con una malla (cuadrícula) de fondo, misma que además de servir como guía, facilita la generación de líneas ortogonales. La malla es generada dinámicamente independientemente de la dimensión de la ventana de trabajo, así mismo, aunque el espacio de trabajo cambie en dimensiones físicas, la malla siempre conserva las dimensiones lógicas para los ejes *x* - *y* cuyos rangos de trabajo se encuentran comprendidos entre -10 y 10. Esto último quiere decir que en realidad el espacio de trabajo del editor es una discretización del espacio euclidiano bidimensional ($\Re^2$) delimitado por el rango antes mencionado.

Las coordenadas lógicas de trabajo del editor, así como los cuadrantes y el rango sobre el que trabaja se ilustran en la Figura 4-7.

Dentro de las características con las que cuenta el editor, se tiene un panel de opciones para controlar la transformación que se realizará sobre el objeto generado. Estas transformaciones son las básicas para poder generar un objeto *evm* y poder situarlo en alguna parte del espacio euclidiano bidimensional.

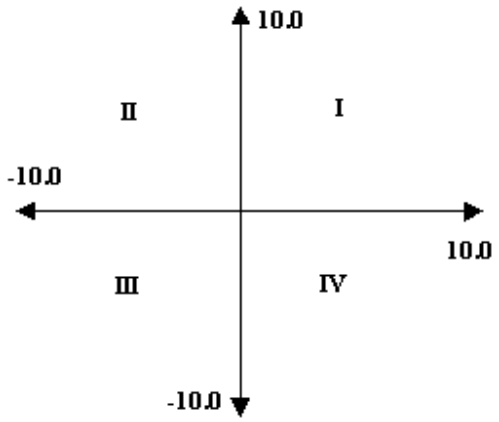

Figura 4-7 Espacio de trabajo, cuadrantes (números romanos) y coordenadas lógicas sobre las que trabaja el editor de pseudo poliedros ortogonales.

La principal idea subyacente del editor, es la de poder crear pseudo poliedros ortogonales más complejos a partir de otros más sencillos generados en el editor, para después si así se desea, combinarlos utilizando las operaciones booleanas regularizadas

sobre de éstos y así obtener otros poliedros más elaborados. Todo esto es posible gracias a que el objeto generado puede ser escalado y trasladado para poder ser "acomodado" en donde mejor convenga.

La forma de generar líneas en el área del editor es a través del uso del ratón. La idea aquí será generar sólo líneas ortogonales, esto es, a través de una corrección dinámica de las coordenadas del ratón dibujar sólo líneas válidas. Cada vez que el usuario intenta representar en el modelo EVM el PPO dibujado en el editor, éste es validado para asegurar que satisface las propiedades necesarias del modelo EVM.

El pseudo poliedro ortogonal irá siendo formado a través de una serie de planos perpendiculares al eje *z*, de tal manera que en cada plano de dibujo lo que en realidad se forma es una extrusión del PPO dibujado en el editor. Cada uno de estos planos es dibujado sobre una representación bidimensional del plano anterior, con la finalidad de proporcionar una guía de dibujo entre lo que constituye el plano actual y el plano anterior. Cada vez que se intenta pasar de un plano de dibujo hacia otro, el nuevo PPO es validado y, si es un objeto EVM válido, entonces se fusiona con el objeto EVM definido en el plano anterior, eliminando si es necesario a los vértices repetidos, conservando así únicamente a los vértices extremos. La idea aquí es entonces, ir construyendo un objeto EVM a partir de una secuencia de objetos EVM que van siendo definidos sobre el eje *z*. La forma de definición sobre el eje *z* será entonces en un sentido de atrás hacia adelante.

Dentro de las características con las que también cuenta el editor, están las de hacer algunas correcciones sobre los objetos definidos. La Figura 4-8 ilustra los posibles casos de error que pudieran ocurrir al definir un objeto en el editor. Las flechas en la figura ilustran la secuencia en la que el objeto es definido, y los puntos muestran todos los vértices marcados, aún los que no son vértices extremos.

Uno de los errores es una inadecuada definición inicial/final del objeto, ya que no empieza en un vértice extremo sino en un vértice intermedio. También puede observarse que entre cada par de vértices extremos se definen otros que no lo son.

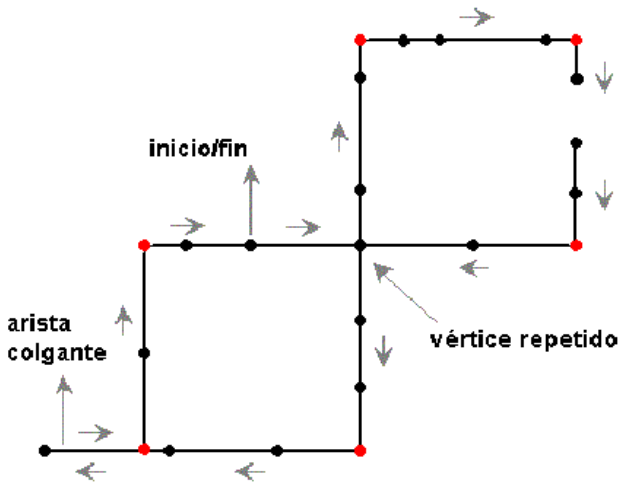

Figura 4-8 Posibles casos de error en la definición de un objeto EVM utilizando el editor.

El mecanismo de corrección aplicado aquí es el siguiente: para cada secuencia de tres vértices, verificar que no tengan la misma coordenada en *x* o en *y*, si es así, se elimina el vértice intermedio y el proceso se repite. Es importante mencionar que esta validación es hecha de manera circular sobre la lista de puntos que van definiendo al objeto. Para el caso de vértices repetidos que no estén contiguos, se realiza un ordenamiento de los vértices y se eliminan, si lo hubiera, cada par de vértices repetidos.

Finalmente debe mencionarse que el mecanismo de corrección entra en acción sólo si el objeto en proceso no es un objeto EVM válido, en este caso se realiza la corrección tratando de interpretar lo que el usuario probablemente quiso decir y, si el objeto generado no es el esperado, se puede deshacer la última edición, este y otros detalles se describen con más amplitud en el capítulo de resultados, además de ilustrar con imágenes algunas de las características más importantes.

4.4 Notas del capítulo.

El presente capítulo ha cubierto de manera general los principales aspectos involucrados en el análisis y diseño del sistema implementado: el software EVM. Dicho

sistema, implementa y ejecuta los algoritmos del modelo de vértices extremos descritos y definidos por el modelo del mismo nombre.

El análisis y diseño general del sistema estuvo enfocado hacia el paradigma de la programación orientada a objetos.

En cuanto a la implementación, el lenguaje seleccionado para la programación fue Java. La elección del lenguaje de programación Java para la implementación del sistema tuvo que ver con su independencia de plataforma, el atractivo inherente al lenguaje, y por qué no decirlo, también a su auge y popularidad crecientes. Con todo, se consideró que se podría tener una aplicación amistosa, funcional y completamente portable sin asirse a los compromisos de "casarse" con algún compilador o arquitectura específica.

Si bien es cierto que Java, al ser un lenguaje que se vale de la interpretación de los *byte codes* tendrá un desempeño menor al de algún otro lenguaje que a través de un compilador genere instrucciones propias de la máquina en la que se ejecuta, los beneficios que ofrece bien podrían compensar ese detalle. Además, aunque en su concepción original Java es un lenguaje interpretado por un máquina virtual, actualmente existen programas capaces de transformar los *bytes codes* de Java en instrucciones de nivel de máquina de la computadora en donde se ejecute, por lo que el desempeño final debería ser tan bueno como el de cualquier otro lenguaje que brinde soporte al paradigma de la programación orientada a objetos, sin perder, claro está, las bondades y satisfacciones que proporciona el programar en un lenguaje como Java.