UNIVERSIDAD NACIONAL AUTÓNOMA

DE MÉXICO

FACULTAD DE CIENCIAS

"Representación Planar de Vox-sólidos"

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

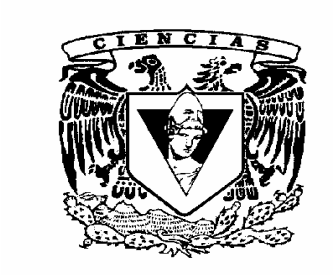
Licenciado en Ciencias de la Computación

P R E S E N T A:

Eliot Peña Rojas

Director de Tesis:

M. en I. María de Luz Gasca Soto



MÉXICO, D.F. **2008**

A mis padres, a mi esposa Yesica y a la vida.

Agradecimientos

Quiero expresar mi gratitud a todas las personas que me apoyaron en la realiación de este trabajo con sus comentarios e ideas, así como a los amigos que en todo momento fueron soporte para continuar en tiempos difíciles.

Por otro lado quiero agradecer a los profesores de la Facultad de Ciencias sin los cuales estre trabajo hubiera sido imposible.

Hoja de Datos del Jurado

1. Datos del alumno
   Peña
   Rojas
   Eliot
   57 83 43 30
   Universidad Nacional Autónoma de México
   Facultad de Ciencias
   Ciencias de la computación
   093111700

2. Datos del tutor
   M en I
   María de Luz
   Gasca
   Soto

3. Datos del sinodal 1
   Dr
   José de Jesús
   Galaviz
   Casas

4. Datos del sinodal 2
   Dra
   Amparo
   López
   Gaona

5. Datos del sinodal 3
   L en CC
   Héctos
   Cuevas
   Vázquez del Mercado

6. Datos del sinodal 4
   M en CC
   Federico
   Juárez
   Almaraz

7. Datos del trabajo escrito
   Representación planar de Vox-sólidos

   162 p
   2008

Índice general

Introducción

Una imagen digital es la representación de un objeto con un número finito de valores digitales. Los elementos que forman la imagen digital, valores digitales, pueden ser dígitos que representan los diferentes colores y tonalidades.

En la actualidad el manejo de imágenes digitales en tres dimensiones ($3D$) ha ido en aumento gracias al incremento en las capacidades de procesamiento de cómputo así como la posibilidad, cada vez más factible, de obtener imágenes digitales por medio de la resonancia magnética y escáners en $3D$.

Mientras que en las imágenes digitales en $2D$ la unidad básica es el pixel (cuadrado unitario), en las imágenes digitales en $3D$ la unidad básica es el voxel (cubo unitario). Las imágenes digitales están conformadas por pixeles o por voxeles según se esté en $2D$ o en $3D$, respectivamente. Al conjunto de voxeles que conforman la imagen, bajo ciertas reglas, se les llama vox-sólido.

Existen diferentes tipos de vox-sólidos, en particular nos interesan los que tienen la forma de un toro o anillo, a este tipo se les conoce como vox-sólidos *toroidales*. Éstos pueden construirse de la siguiente forma: colocar un voxel tras otro, de tal manera que formen una "línea" recta, después doblar esta línea de tal forma que los extremos queden unidos. Si además todas las aristas del vox-sólido están sobre la frontera diremos que es un vox-sólido toroidal *delgado*.

Cuando vemos objetos en $3D$ no es posible observarlos de forma completa desde una sola dirección, por lo cual es necesario hacer varios movimientos para verlo completo; por ejemplo, si se tiene un cubo y lo vemos desde cualquier dirección, a lo más lograremos ver tres de sus caras. Así, para lograr ver todo el vox-sólido es necesario construir su *representación planar*, la cual es una gráfica en $2D$ que nos permite ver todas las caras del vox-sólido y la forma en que éstas se relacionan.

El objetivo de la tesis es desarrollar un método que obtenga la representación planar de vox-sólidos toroidales delgados, para finalmente hacer una implementación didáctica en la cual sea fácilmente visualizado el proceso de construcción de su representación planar.

En el Capítulo 1 se presentan los conceptos básicos sobre la topología digital. En el Capítulo 2 se describe una representación de vox-sólidos. En el Capítulo 3 se desarrollan métodos para obtener la representación planar de diferentes tipos de vox-sólidos toroidales; estudiando con detalle el caso de los vox-sólidos toroidales delgados. Finalmente, presentamos las conclusiones y propuestas para el desarrollo del trabajo futuro.

Además, se anexan un par de apéndices, en los cuales se explican otras posibles representaciones para los vox-sólidos (Apéndice A) y se incluye el código fuente de un programa que implementa los métodos descritos en este trabajo (Apéndice B).

Capítulo 1

Conceptos básicos

1.1. Topología Digital

La topología es un área de las matemáticas que se dedica al estudio de las propiedades de los espacios topológicos y funciones continuas. La topología se interesa por conceptos como: conexidad, conectividad, proximidad, características de los objetos, el tipo de consistencia (o textura) que presenta un objeto, entre otros; también se considera importante la comparación y clasificación de objetos. La topología digital permite la manipulación de las imágenes digitales utilizando sus propiedades topológicas.

Esta área se ha desarrollado extensamente en las imágenes en $2D$ y se han logrado avances significativos en su procesamiento. Con el incremento en el adquisición y procesamiento de imágenes en $3D$, por medio de la resonancia magnética y el uso de programas para el manejo de objetos en $3D$ como los de Diseño Asistido por Computadora (CAD[1]) y Realidad Virtual (VR[2]), se ha incrementado de forma importante el procesamiento de imágenes digitales en $3D$.

Los investigadores que se especializan en las ramas de reconocimiento de patrones, análisis de imágenes o áreas afines han encontrado que los conceptos fundamentales de topología son una herramienta útil. Aunque las definiciones usuales de topología, como adyacencia, número de componentes, grupos fundamentales, métricas, etcétera, no se aplican directamente a las imágenes digitales pero se establecen condiciones para adaptar todos los conceptos a la topología digital.

[1]Por sus siglas en inglés de Computer-aided Design
[2]Por sus siglas en inglés de Virtual Reality

Se considera a A. Rosenfeld el fundador de la topología digital, su trabajo "Digital Topology" [13], es uno de los primeros en este campo.

Habitualmente la topología digital comienza con la noción de objetos conexos basada en elementos adyacentes; dos elementos son adyacentes si están próximos. A partir de la noción de adyacencia se puede construir una analogía de la topología clásica.

En la topología digital se considera a $\mathbb{Z}^N$ como un *espacio digital* que está formado por el conjunto de todos los puntos del espacio $\mathbb{R}^N$ donde sólo se consideran las coordenadas enteras. La topología digital toma las propiedades de los subconjutos del espacio digital, también llamados *conjuntos digitales*.

Definición 1.1.1. Espacio digital de imágenes binarias [2].
Un espacio digital de imágenes binarias es una tripleta (V, β, ω) donde V es el conjunto de puntos generados por una retícula regular en $\mathbb{Z}^N$, con $N = 2, 3$; β y ω son el conjunto de segmentos de líneas rectas que unen pares de puntos en V. A los puntos que son unidos por segmentos de líneas en β se les conoce como puntos *negros*, mientras que a los puntos que son unidos por segmentos de líneas en ω se les conoce como puntos *blancos*.

Dada la retícula regular, los puntos p generados por esta, son el baricentros de cada una de las celdas de la retíclua, en la Figura 1.1 se muestran los puntos generados en cada uno de las celdas de una retícula en $\mathbb{Z}^2$ y $\mathbb{Z}^3$, (a) y (b) respectivamente, las líneas grises muestran como se encontro el baricentro.

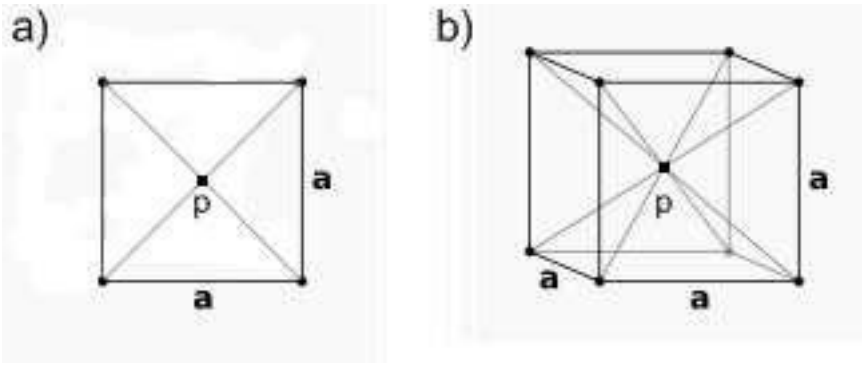


Figura 1.1: Puntos generados por celdas de una retícula en $\mathbb{Z}^2$ y $\mathbb{Z}^3$

En el presente trabajo llamaremos a los *espacios digitales de imágenes binarias* simplemente como *espacios digitales* y nos enfocaremos solamente en el espacio $\mathbb{Z}^3$.

Las celdas generadas por la retícula regular en el espacio $\mathbb{Z}^3$ peden tener diferente forma, por ejemplo, puede ser cubos, octaedros, dodecaedros, etcétera. En particular en este trabajo utilizaremos una retícula regular que genera cubos.

Una noción importante en la topología digital es la de adyacencia entre puntos. Típicamente se utilizan diferentes relaciones de adyacencias para los puntos blancos y para los puntos negros aunque éstas dos pueden ser iguales.

En un espacio digital la relación de adyacencia [2] está definida por los segmentos de líneas rectas en β y ω. El conjunto β contiene todas los segmentos de líneas rectas que unen a pares de puntos en V si los puntos son negros; similarmente, el conjunto ω contiene todas los segmentos de líneas rectas que unen a pares de puntos en V si los puntos son blancos.

El conjunto de segmentos de líneas rectas en β es llamado β-adyacencias, de manera similar, el conjunto de segmentos líneas rectas en ω es llamado ω-adyacencias. En general, si β es el conjuto de m-adyacencias, para algún entero m, y ω es el conjunto de n-adyacencias, para algún entero n; entonces se denota el espacio digital (V, β, ω) como (V, m, n).

En la Figura 1.2 se muestran las configuraciones de las adyacencias 4 y 8 (en los incisos a y b respectivamente) que son las más utilizadas en $\mathbb{Z}^2$; en la figura se muestra la retícula y los puntos adyacentes al punto p que se muestra en el centro.

En $\mathbb{Z}^3$ tenemos que las adyacencias más usuales son 6, 18 y 26. En la Figura 1.3 se muestran las configuraciones de las N-adyacencias en $\mathbb{Z}^3$ para el punto p , en este ejemplo sólo se muestran los puntos y no la retícula.

Definición 1.1.2. Imagen digital binaria. Una imagen digital binaria es la cuarteta $\mathcal{P} = (V, \beta, \omega, B)$ donde (V, β, ω) es un espacio digital y $B \subset V$.

En este trabajo llamaremos a las *imágenes digitales binarias* simplemente como *imágenes digitales* y trabajaremos con imágenes digitales $\mathcal{P} = (\mathbb{Z}^3, 6, 18, \mathbb{N})$, es decir se usa la relación 6-adyacencias para los puntos negros

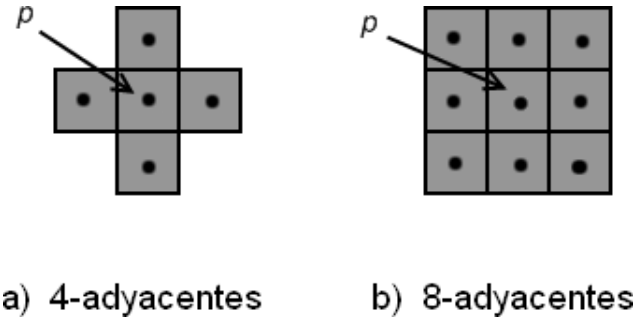


Figura 1.2: Las N-adyacencias más comunes en $2D$

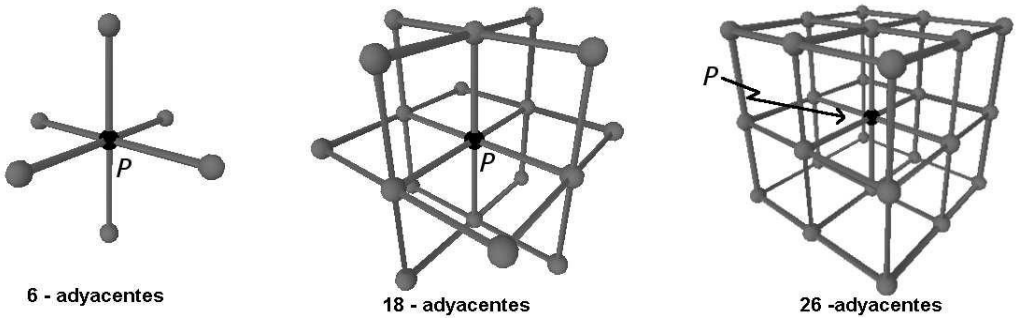


Figura 1.3: Las N-adyacencias más usuales en $3D$

y 18-adyacencias para los puntos blancos.

1.2. Vox-sólido

Definición 1.2.1. Voxel. [11] Sean $\mathcal{P} = (\mathbb{Z}^3, 6, 18, \mathbb{N})$ una imágen digital, un voxel es la región contenida en una celda de la retícual regualar de $\mathcal{P}$ centrada en el punto (x, y, z) con $x, y, z \in \mathbb{N}$.

Con base en esta definición tenemos que un voxel ocupa un *cubo* unitario centrado en el punto (x, y, z), por lo que un voxel tiene 6 caras y 12 aristas. Siendo un voxel el espacio contenido en la retícula de la imágen digital entonces también lo podemos llamar punto. En adelante se utilizará de manera indistinta el concepto de voxel y de punto.

Es posible representar objetos con las *imágenes digitales* haciendo uso de los voxeles, se utilizan voxeles negros para formar el objeto a representar y

voxeles blancos para representar el fondo de la imagen. En la Figura 1.4 se muestra un ejemplo de la representación de objetos en una imagen digital, del lado izquierdo está el objeto a representar y del lado derecho está su imagen digital correspondiente, en la cual se muestra la retícula; en (a) se muestra un objeto en $2D$ mientras que en (b) se muestra un objeto en $3D$.

En adelante nos referiremos a los voxeles negros, que son los que forman el objeto a representar, simplemente como voxeles, mientras que cuando hablemos de voxeles blancos utilziaremos este término completo.

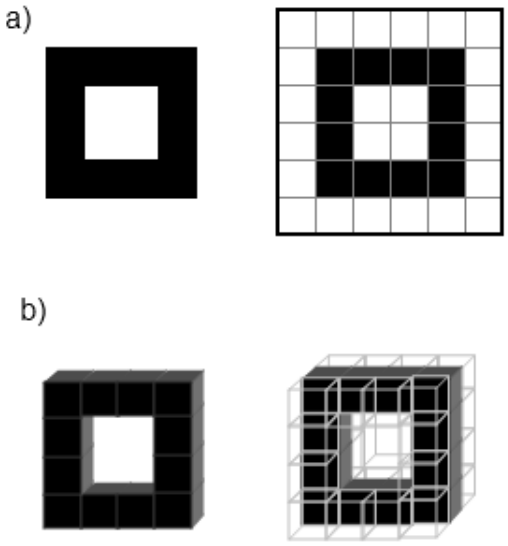


Figura 1.4: Muestra de imágenes digitales en $2D$ y $3D$

Definición 1.2.2. Vox-sólido. Un vox-sólido es un conjunto no vacío y conexo de voxeles cuya frontera es una superficie orientable.

Una superficie es una 2-variedad (objeto geométrico estándar), es decir, un objeto topológico que, intuitivamente hablando, es localmente *parecido* al plano cartesiano $\mathbb{R}^2$. Eso significa que para cada punto P de una superficie hay una vecindad de P que es homeomorfa a un disco abierto de $\mathbb{R}^2$.

Se dice que una superficie es no oritable si contiene al menos una subsuperficie que es homeomorfa a una banda de Möebius cerrada. En caso

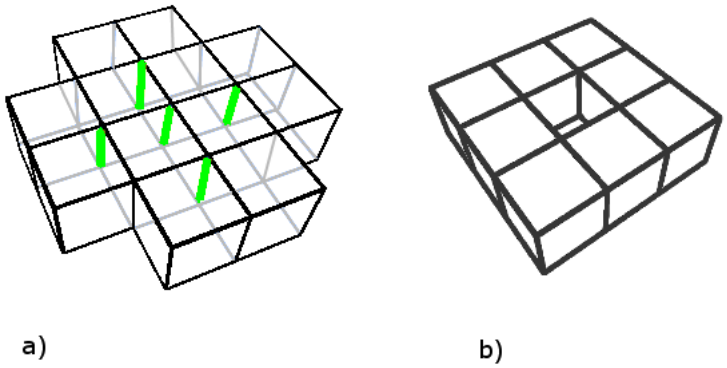


Figura 1.5: Un vox-sólido no-delgado y delgado.

contrario se dice que la superficie es orientable.

Diremos que dos voxeles son adyacentes si están unidos por una cara, es decir, comparten una cara.

Definición 1.2.3. Grado de un voxel. El grado de un voxel v en un vox-sólido $\mathcal{V}$ es el número de voxeles adyacentes a él.

El grado máximo de un voxel es 6, cuando el voxel está aislado su grado es cero.

Definición 1.2.4. Grado de un vox-sólido. Es el máximo grado de los voxeles que componen el vox-sólido.

Definición 1.2.5. Vox-sólido delgado. Se dice que un vox-sólido es delgado si todas las aristas de los voxeles que lo componene están sobre la frontera del vox-sólido. En caso contrario se dira que es $no-delgado$ o $gordo$.

En la Figura 1.5 se muestra en (a) un vox-sólido no-delgado, en este hay cinco aristas internas (las que aparecen más gruesas) que no están en la frontera del vox-sólido; en (b) se muestra un vox-sólido delgado .

Definición 1.2.6. Toro. El Toro es una superficie orientable que resulta de unir las parejas de lados opuestos de un rectángulo.

En la Figura 1.6 *(a)* se construye el toro; primero se unen los extremos del rectángulo etiquetados como a,el resultado es el cilindro *(b)*; posteriormente unimos los extremos marcados como b y tenemos como resultado el toro *(c)*.

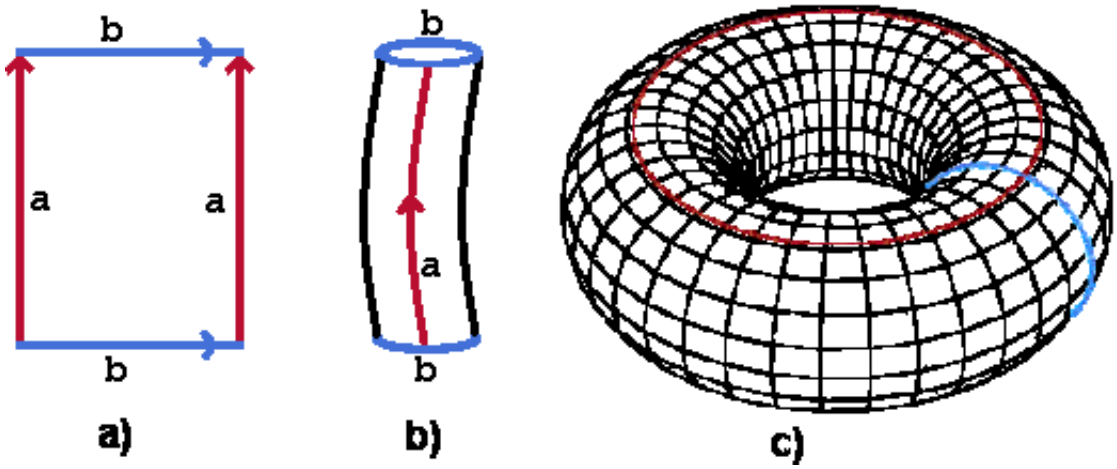


Figura 1.6: Desdoblamiento de un Toro

Al rectángulo que se utiliza para generar el toro se le conoce como *representación planar*. Una de las ventajas de la representación planar es que nos permite observar toda la superficie en una sola vista, mientras que la perspectiva en el espacio esconde, invariablemente, alguna porción de la superficie del toro.

Definición 1.2.7. Vox-sólido toroidal. Un vox-sólido toroidal es aquel que puede ser encajado en un toro de género 1. En la Figura 1.7 se presentan ejemplos de vox-sólidos toroidales.

Definición 1.2.8. Camino [2]**.** Dado un vox-sólido, un camino es una sucesión $p_1, p_2, \ldots, p_n$ de voxeles donde p_i es adyacente a p_{i+1} con $1 \leq i < n$. Una trayectoria $\mathcal{P}$ de voxeles es un camino que no repite voxeles. Si además $\mathcal{P}$ utiliza todos los voxeles del vox-sólido, diremos que es una trayectoria hamiltoniana.

Sea $\mathcal{V}$ un vox-sólido, dos caras, sobre la superficie de $\mathcal{V}$, son adyacentes si tienen una arista en común.

Definición 1.2.9. Gráfica de Adyacencia de caras. Sea $\mathcal{V}$ un vox-sólido y $G = (V, E)$ una gráfica o grafo, se dice que G es la gráfica de adyacencia de caras de $\mathcal{V}$ si los vértices en G están asociadas a cada una de las caras que se encuentran en la superficie de $\mathcal{V}$ y se tiene una arista entre dos vértices si las respectivas caras son adyacentes en $\mathcal{V}$.

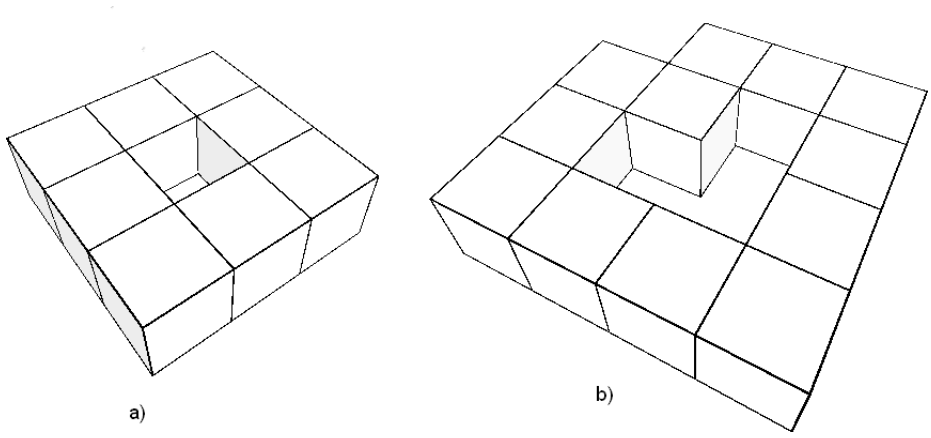


Figura 1.7: Ejemplo de vox-sólidos Toroidales

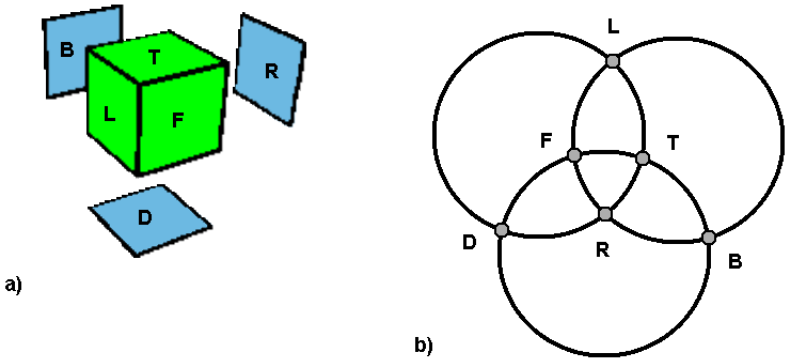


Figura 1.8: Un voxel y su gráfica de adyacencia de caras.

En la Figura 1.8 se muestra la gráfica de adyacencia de caras de un vox-sólido que sólo tiene un voxel. En (a) se muestra el vox-sólido donde aparecen todas sus caras etiquetadas para que se identifiquen fácilmente, las caras en la superficie del vox-sólido que no son visibles desde la perspectiva en que se muestra el objeto aparecen en el fondo de tal forma que sean visibles; en (b) se muestra la gráfica de adyacencia de caras resultantes.

Cuando un vox-sólido tiene más de un voxel la gráfica de adyacencia de caras se hace más compleja, en la Figura 1.9 se muestra la gráfica de adyacencia de caras para un vox-sólido que tiene dos voxeles. En (a) se puede ver el vox-sólido con sus caras etiquetadas y sus caras que no son visibles desde la perspectiva en que se muestra el objeto aparecen en el fondo; en (b) se

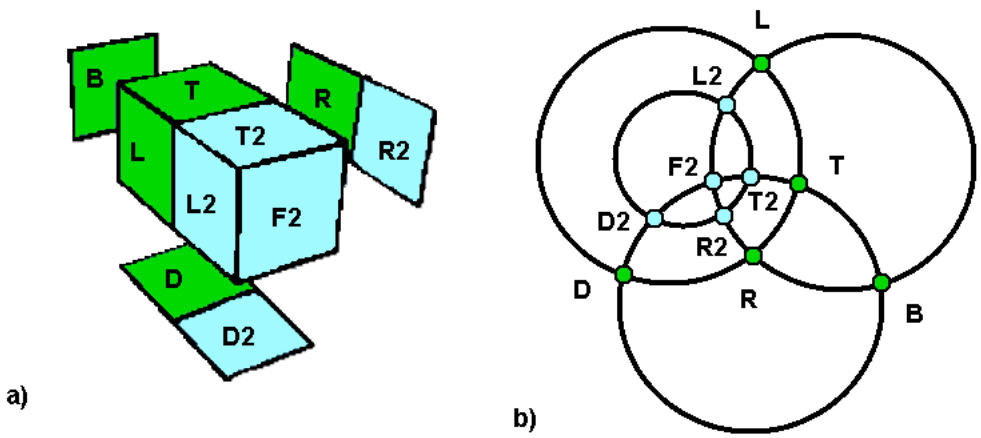


Figura 1.9: Un vox-sólido y su gráfica de adyacencia de caras.

puede ver su gráfica de adyacencia de caras. En la gráfica los vértices tienen el color del voxel al que están asociados.

Para generar la gráfica de adyacencias de dos voxeles tomemos el primer voxel, al que llamaremos v_1 y su gráfica asociada, agreguemos otro voxel, al que llamaremos v_2, en la cara F de v_1, se elimina el vértice asociado a la cara F y en su lugar se pone el vértice $F2$ y sus caras adyacentes de tal forma que se puede poner una arista entre los vértices adyacentes a la cara F de v_1 y los vértices adyacentes a la cara $F2$.

En la Figura 1.9 se puede ver que la cara T y $T2$ son adyacentes y existe un vértice que los une en la gráfica, al igual que para L y $L2$, R y $R2$, D y $D2$, mientras que $F2$ ocupa el lugar que tenía F.

Capítulo 2

Representación matricial de Vox-sólidos

Una representación es la correspondencia entre los elementos de dos conjuntos. Existen varias representaciones de sólidos en $3D$ que se podrían utilizar para vox-sólidos, las más utilizadas son la *representación de bordes (B-Rep*[1]*)* y la *geometría constructiva de sólidos (CSG*[2]*)* [11].

Una representación ideal para vox-sólidos debe mantener información de los voxeles que lo componen y de cómo están relacionadas sus respectivas caras, en particular las caras que se encuentran en la superficie del vox-sólido. Dicha representación también debe ser eficiente en el uso de los recursos de cómputo de los sitemas informáticos.

En este trabajo se propone una *representación matricial* de vox-sólidos cuyo principal objetivo es facilitar la obtención de la representación planar de los vox-sólidos toroidales. Esta representación está diseñada para que se pueda obtener con facilidad las adyacencias de los voxeles y de sus caras. En el Apéndice A se describen otras posibles representaciones de vox-sólidos que podrían utilizarse para este fin sólo que no están enfocadas a obtener su representación planar.

[1]Por sus siglas en inglés de Boundary represerntation
[2]Por sus siglas en inglés de Constructive Solid Geometry

2.1. Representación de una Imagen Digital

La representación clásica, dentro de los sistemas de cómputo, de las *imágenes digitales* en $2D$ es hacer un isomorfismo con una retícula en $\mathbb{Z}^2$, es decir, la imagen digital puede ser representada con una matriz de dos dimensiones $M_{n,m}$, en la cual si una entrada tiene el valor 1 significa que hay un punto negro, mientras que si el valor de la entrada es 0 entonces significa que no hay un punto o es un punto blanco.

La extensión clásica de la representación de las *imágenes digitales* para $3D$ es hacer un isomorfismo con una retícula en $\mathbb{Z}^3$, es decir, se puede representar con una matriz de tres dimensiones $M_{n,m,k}$ donde si una entrada tiene el valor 1 significa la presencia de un voxel, si la entrada tiene el valor 0 significa la ausencia de un voxel blanco.

Definición 2.1.1. Representación de una Imagen Digital. Sea ID una imagen digital en $\mathbb{R}^3$, haciendo un isomorfismocon una retícula en $\mathbb{Z}^3$ representaremos a ID con una matriz $M_{n,m,k}$ de tal forma que cada uno de los puntos de ID están representados por una entrada de $M_{n,m,k}$, si la entrada tiene el valor de 1 significa que es un voxel, mientras que si el valor de la entrada es 0 significa la carencia de este o que es un voxel blanco.

Definición 2.1.2. Semejanza de Imágenes Digitales. Sean ID_1 y ID_2 dos imágenes digitales, diremos que ID_1 es semejante a ID_2 si todos los puntos negros de ID_1 están contenidos en ID_2 y en ID_2 no hay mas puntos negros, ademas todos los puntos negros en ID_2 conservan la misma adyacencia que en ID_1, es decir, que estén representando la misma imagen sólo que la cardinalidad de ambos conjuntos no necesariamente es la misma.

En la Figura 2.1 se muestran imágenes digitales semejantes (que están inmersas en una retícula), en $2D$ y en $3D$, en (a) y (b), respectivamente, nótese que la diferencia de las imágenes es que su fondo es diferente pero ambas están representando el mismo objeto.

Con esta definición podemos encontrar una familia $\mathcal{M} = \{M_{i,j,k}\}\ \forall\, i,j,k \in \mathbb{Z}^+$ de matrices semejantes que sean representación de una ID, por ejemplo, sea la matriz $M_{n,m,k}$ una representación de una imagen digital, podemos generar otra matriz $M_{n,m,k+1}$ que sea otra representación de la imagen digital donde las entradas agregadas sean valores 0 (ausencia de voxel), más aún $M_{n,m,k}$ está contenida en $M_{n,m,k+1}$. De manera similar, se pueden cambiar el

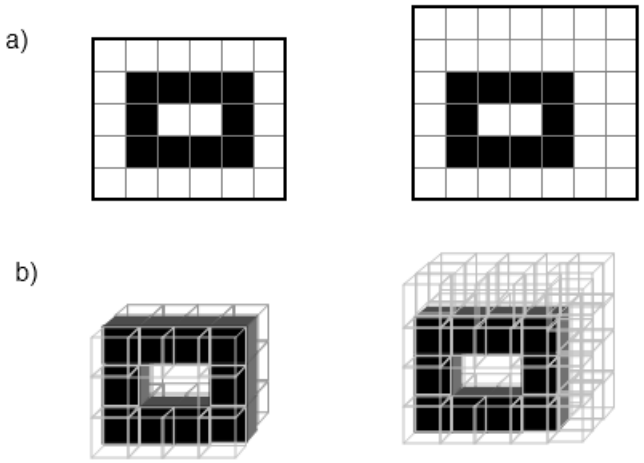


Figura 2.1: Semejanza de imágenes digitales.

resto de las dimensiones de la matriz para obtener toda la familia de representaciónes de la imagen digital.

2.2. Representación simple de Vox-sólidos

Definición 2.2.1. Dimensión de un Vox-sólido. La dimensión de un vox-sólido es el vector (n_1, n_2, n_3) donde n_i es el tamaño del vox-sólido al ser proyectado en el vector canónico i.

Definición 2.2.2. Representación simple de un Vox-sólido.
Sean ID una imagen digital, $\mathcal{V}$ un vox-sólido que tiene dimensiónes (x, y, z) y que está contenido en la imagen ID; $\mathcal{M} = \{M_{i,j,k}\}\ \forall\ i, j, k \in \mathbb{Z}^+$ la familia de matrices de representación de ID y $M_{n,m,k}$ una matriz contenida en $\mathcal{M}$. Diremos que $M_{n,m,k}$ es la representación simple de $\mathcal{V}$ si todos los voxeles que componen $\mathcal{V}$ están contenidos en $M_{n,m,k}$ con $n \leq x$, $m \leq y$ y $k \leq z$.

Con la representación simple de un vox-sólido se tiene toda una familia de matrices de representación del vox-sólido y todas éstas son semejantes, conceptualmente esto no tiene problema pero es mejor si siempre se utiliza la misma matriz para representar al voxel, al final nos facilitará la implementación de un método para obtener la representación planar.

Definición 2.2.3. Mínima representación simple de un Vox-sólido. Sea $\mathcal{V}$ un vox-sólido con dimensión (n_1, n_2, n_3) y $\mathcal{M} = \{M_{i,j,k}\}\ \forall\ i, j, k \in \mathbb{Z}^+$ la familia de las matrices que son representación de $\mathcal{V}$; diremos que $M_{k,k,k} \in \mathcal{M}$ es la mínima representación simple de $\mathcal{V}$ si $n_i = k$ para alguna $i = 1, 2, 3$ y $n_j \leq k$ para $i \neq j, j = 1, 2, 3$.

En otras palabras se elige una matriz, cúbica, con todas sus dimensiones iguales y que sea la más pequeña que contiene al vox-sólido.

2.3. Representación matricial de Vox-sólidos

La mínima representación simple de un vox-sólido es intuitiva y fácil de manejar, pero tiene la desventaja de que la matriz puede ser muy grande y, por lo tanto, requerir mucho espacio para su almacenamiento.

En la Figura 2.2 se presenta un ejemplo de un vox-sólido construido de tal forma que semeje una "línea" entre el punto $(0, 0, 0)$ y $(10, -10, 0)$ y posteriormente otra "línea" al punto $(10, -10, 10)$, la mínima representación simple para éste vox-sólido es una matriz $M_{10,10,10}$ que tendría 1000 entradas para representar un objeto con únicamente 28 voxeles.

En el ejemplo anterior el número de ceros (ausencia de voxel) en la matriz es mayor que el número de unos (presencia de voxel), por lo que utilizamos mucho espacio para almacenar la información que pertenecen al fondo de la imagen. Si de esta matriz tan sólo tomáramos los voxeles, los cuales representan el objeto, entonces almacenaríamos menos información y podríamos recuperar fácilmente los ceros pues son el complemento de los voxeles.

En adelante se hablará de orientaciones dentro de la mínima representación simple de un vox-sólido por lo que para orientarnos fijaremos una de las caras de la matriz, sin importa cual sea, como el frente y sobre estas se orientan las direcciones: derecha, izquierda, superior, inferior y trasera. De igual forma todas las caras de los voxeles que esten en la cara de la matriz etiquetada como el frente se etiqueterán como el frente.

Definición 2.3.1. Enumeración estándar. Sea $\mathcal{V}$ un vox-sólido y M la mínima representación simple de $\mathcal{V}$. Llamamos enumeración estándar a aquella que enumera, comenzando en cero, cada una de las entradas de M en el

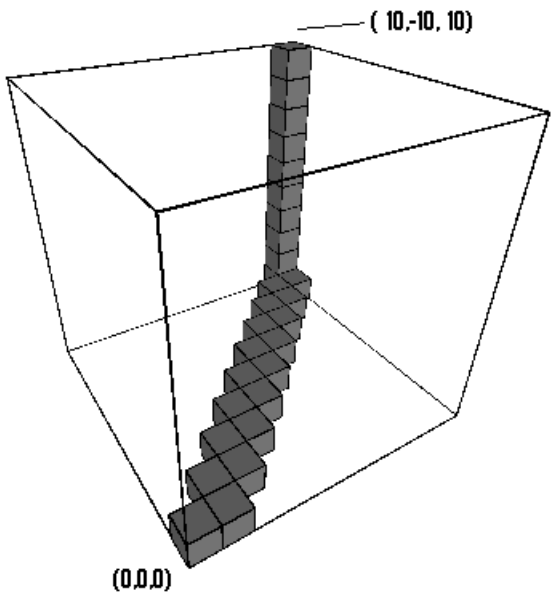


Figura 2.2: Ejemplo de vox-sólido con pocos elementos, si se pone en una matriz $M_{10,10,10}$

siguiente orden, de izquierda a derecha, de adelante hacia atrás y de abajo hacia arriba.

En la Figura 2.3 (a), se muestra la enumeración estándar, la matriz aparece como un cubo, las flechas muestran en órden de enumeración; en (b) se muestra la enumeración de la representación mínima simple, pero cada piso está desfasado, esto es para que se tenga una mejor visión de la enumeración.

Como la representación de los vox-sólidos es mediante una matriz se asocia a cada uno de los voxeles con cada una de las entradas de la matriz, a su vez asociamos cada una de las entradas de la matriz con un número siguiendo la enumeración estándar, entonces estamos asociando cada uno de los voxeles del vox-sólido con el número que corresponde a la entrada de la matriz donde se encuentra. En adelante utilizaremos indistintamente el término voxel y entrada de la mínima matriz de representación del vox-sólido, más aún, llamaremos al voxel con el número asociado, según la enumeración estándar, a la entrada donde se encuentra.

Si tenemos la matriz que representa un vox-sólido y de ésta sólo tomamos el número que representa cada voxel negro, obtenemos una lista de los voxeles

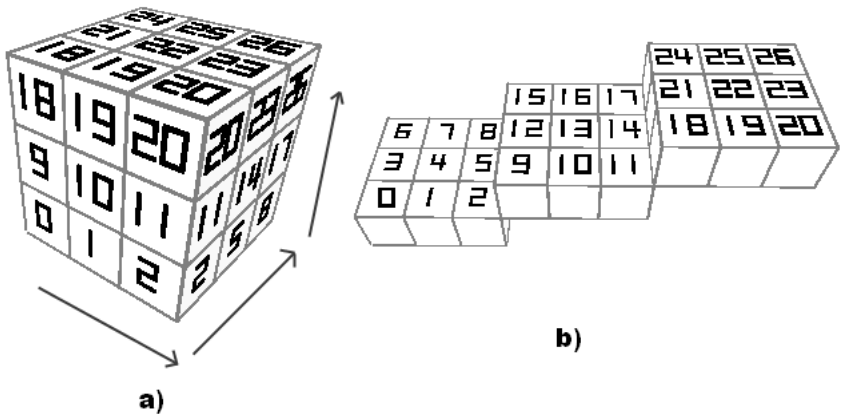


Figura 2.3: Ejemplo de la enumeración estándar.

que conforman el vox-sólido, es decir, generamos una codificación del vox-sólido que tiene como ventaja reducir la cantidad de espacio que se utiliza para almacenarlo, por ejemplo, para representar el vox-sólido de la Figura 2.2 sólo se utilizaría una lista de 28 elementos.

En la Figura 2.4 se muestra la enumeración estándar para un vox-sólido toroidal, el cual está dentro de la matriz mínima que lo contiene; se incluye la enumeración de toda la matriz, los voxel negros aparecen de color gris obscuro. De este ejemplo podemos concluir que el vox-sólido está conformado por los voxeles $\{0, 1, 2, 3, 5, 6, 7, 8\}$, el voxel 4 no forma parte del vox-sólido.

Definición 2.3.2. Representación matricial de un vox-sólido.
La representación matricial de un vox-sólido es la tripleta $\mathcal{V} = \{N, L, M\}$ donde N es la dimensión de la mínima representación simple del vox-sólido; L es la lista de los voxeles que conforman el vox-sólido, utilizando la enumeración estándar y M es la matriz de adyacencias de voxeles, que se describirá más adelante.

En la Figura 2.4 se muestra el vox-sólido toroidal al que llamaremos *tuerca*: $\{3, \{0, 1, 2, 3, 5, 6, 7, 8\}, M\}$; en la Figura 1.7 (b) se presenta otro vox-sólido toroidal: $\{4, \{0, 1, 2, 3, 4, 7, 8, 9, 11, 13, 14, 15\}, M\}$. En la Figura 2.5 (a) se ilustra el vox-sólido *cruceta*: $\{3, \{4, 10, 12, 13, 14, 16, 22\}, M\}$; en la Figura 2.5 (b) se muestra el vox-sólido toroidal $\mathcal{L}_{16}$ [5]:

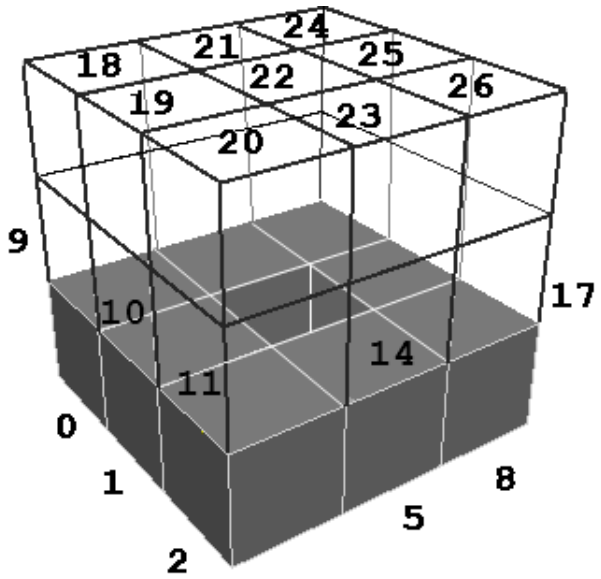


Figura 2.4: Ejemplo de enumeración estardar de un vox-sólido toroidal.

$\{4, \{1, 2, 6, 7, 11, 16, 17, 20, 27, 30, 31, 36, 40, 41, 45, 46\}, M\}$.

Con esta representación de los vox-sólidos podemos identificar para cada uno de los voxeles, por construcción de la enumeración estándar, cuáles son los seis voxeles adyacentes, uno para cada una de sus caras, si éstos existen. Hay que recordar que se está utilizando el concepto de 6-vecinos o 6-adyacencias. Por ejemplo, sabemos que a la izquierda del voxel 1 está el voxel 0 y a su derecha está el voxel 2.

Debido a que la representación matricial de los vox-sólidos se basa en la representación mínima simple y ésta a su vez en una matriz cúbica, podemos saber con precisión donde están cada uno de los voxeles, más aún, podemos encontrar varias fórmulas para saber que voxel está en cierta dirección.

Por ejemplo, la fórmula para saber que voxel está enfrente del voxel x es la siguiente: $x - N$, con $x < N$. En la Figura 2.4 podemos ver que frente al voxel 5 está el voxel 2. En el Método 1 se describe el pseudo-código para encontrar las 6-adyacencias de un voxel. Si la adyacencias es NULL, significa que el voxel no tiene adyacencia en esa dirección.

Si se requiere trabajar con las caras que componen los voxeles dentro del

Método 1 Calcula las adyacencias de un voxel

Precondición: $V = \{N, L, M\}$ un vox-sólido, $x \in L$
Postcondición: Las seis adyacencias del voxel$x$

```
1:
2: // Funciones de uso general
3: Tpiso := N * N
4: funcion col(x){x  mod N}
5: funcion ren(x){parte_entera(x/N)}
6: funcion piso(x){parte_entera(x/Tpiso)}
7:
8: adyFrente := x - N
9: if adyFrente < 0 then
10:    adyFrente := NULL
11: end if
12:
13: adyDerecha := ((ren(x) - 1) * N) + col(x)
14: if adyDerecha < piso(x) * Tpiso then
15:    adyDerecha := NULL
16: end if
17:
18: adyIzquierda := (ren(x) * Tpiso) + (col(x) - 1)
19: if adyIzquierda < (x/N) * N then
20:    adyIzquierda := NULL
21: end if
22:
23: adyAtras := (ren(x) * Tpiso) + (col(x) + 1)
24: if adyAtras ≥ ((x/N) + 1) * N) then
25:    adyAtras := NULL
26: end if
27:
28: adyArriba := (((ren(x) * Tpiso) + 1) * N) + col(x)
29: if adyArriba ≥ (piso(x) + 1) * Tpiso then
30:    adyArriba := NULL
31: end if
32:
33: adyAbajo := x + Tpiso
34: if adyAbajo ≥ Tpiso * N then
35:    adyAbajo := NULL
36: end if
```

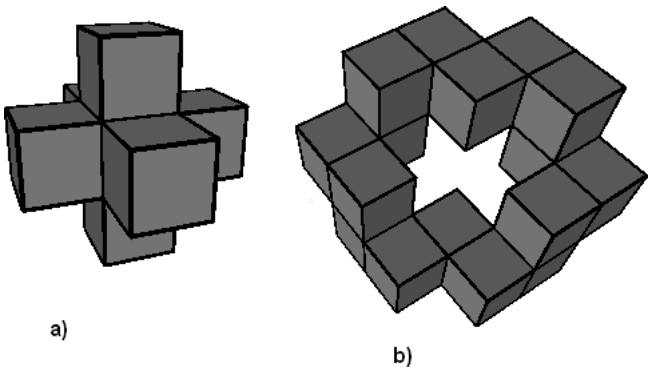


Figura 2.5: Ejemplo de vox-sólidos.

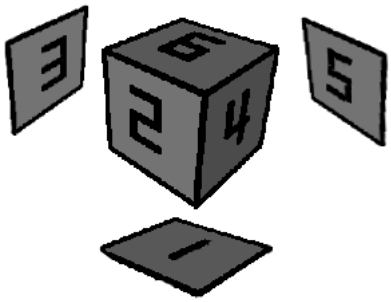


Figura 2.6: Enumeración de las caras del voxel.

vox-sólido utilizamos la matriz de adyacencias que nos permite identificar cuáles de las caras del voxel son adyacentes a otro voxel y cuáles no.

Sin pérdida de generalidad, podemos enumerar las caras del voxel de la siguiente forma, si vemos al voxel de frente la cara de abajo es la 1, la de enfrente es la 2, la cara de la izquierda es la 3, la de la derecha es la 4, la cara de atrás es la 5 y la superior es la 6. A esta enumeración le llamaremos **Enumeración de caras de un voxel**. En la Figura 2.6 se presenta esta enumeración para el voxel.

Sea $\mathcal{V} = \{N, L, M\}$ la representación matricial de un vox-sólido, la matriz de adyacencias M es un arreglo de $6 \times N$, los renglones representan las seis caras del voxel y las columnas cada uno de los voxeles que componen el vox-sólido; en cada entrada de la matriz se almacena el número del voxel con el que la cara es adyacente. Para encontrar las adyacencias se utiliza el Método 1.

		Voxeles							
		0	1	2	3	5	6	7	8
C	1								
a	2				0	2	3		5
r	3		0	1				6	7
a	4	1	2				7	8	
s	5	3		5	6	8			
	6								

Figura 2.7: Matriz de adyacencia de la tuerca.

En la Figura 2.7 se muestra la matriz de adyacencias para la tuerca, Figura 2.4: $\{3, \{0, 1, 2, 3, 5, 6, 7, 8\}, M\}$, las entradas que aparecen vacías tienen el valor de nulo y significa que el voxel no tiene ninguna adyacencia en esa cara.

Nótese que en la matriz de adyacencias sólo hay dos valores utilizados en cada columna, por lo que esta estructura puede ser representada por una lista y así reducir el espacio para almacenar la matriz; para facilitar su explicación en el presente trabajo se referirá a esta estructura como una matriz.

Definición 2.3.3. Caras libres de un voxel. Sea $\mathcal{V} = \{N, L, M\}$ un vox-sólido y v un voxel de $\mathcal{V}$, llamaremos caras libres de v a aquellas que no sean adyacentes a otro voxel, es decir, la cara x es libre si $M[x][v]$ es nulo.

La representación matricial de un vox-sólido es una estructura que contiene la descripción completa del vox-sólido, tanto de los voxeles que lo componen como de las caras que componen los voxeles, también muestra como están relacionadas todas sus partes.

2.3.1. Operaciones con Vox-sólidos

Al tener la representación matricial para los vox-sólidos nos surgen una serie de preguntas relacionadas con la manipulación de dicha estructura, por ejemplo, si tenemos dos vox-sólidos ¿cómo podemos unirlos para generar otro vox-sólido? En general nos preocupan las operaciones básicas como unión, división e intersección que nos permiten manipular la estructura.

Para realizar cualquier operación con vox-sólidos hay que trabajar con los tres elementos que lo componen (dentro de la representación matricial) que son:

N, la dimensión de la mínima matriz de representación
L, la lista de voxeles que conforman el vox-sólido
M, la matriz de adyacencias.

2.3.1.1. Unión

Una de las primeras operaciones necesarias es la de unión de dos vox-sólidos, porque nos permite formar vox-sólidos de forma inductiva, es decir, tener un vox-sólido base e ir agregando voxeles para obtener otro.

En la Figura 2.8 se muestra gráficamente la unión de los vox-sólidos $(a)\mathcal{V}_1 = \{3, \{0, 1, 2, 3, 5, 6, 7, 8\}, M\}$ y $(b)\mathcal{V}_2 = \{4, \{7\}, M\}$ dando como resultado el vox-sólido ; $(c)\mathcal{V} = \{4, \{0, 1, 2, 4, 6, 7, 8, 9, 10\}, M\}$. En la imagen se muestra la matriz de la representación simple mediante el cubo que contiene al voxel. Suponemos que el voxel cero es el que está más cerca del punto $(0, 0, 0)$.

Para obtener la unión de dos vox-sólidos $\mathcal{V}_1$ y $\mathcal{V}_2$ hay que encontrar la dimensión de la mínima matriz de representación (N) que contenga a $\mathcal{V}_1$ y $\mathcal{V}_2$; después hay que crear la lista de los voxeles que conforman el nuevo vox-sólido, es decir hay que crear la lista L y finalmente hay que obtener la matriz de adyacencias M para los elementos de L.

En el Método 2 se describe a detalle el procedimiento para obtener la unión de vox-sólidos. En el pseudo-código estamos utilizando la notación de orientación a objetos y suponemos que la representación matricial es un objeto, N un entero, L es un arreglo y M una matriz; para identificar los miembros de los objetos se utiliza el carácter "." (punto).

En el pseudo-código se utilizan funciones adicionales para su correcto funcionamiento. La función *renumeraVX(L,N_prev,N_nva)* que se describe en el Método 3, calcula la nueva posición de los voxeles de la dimensión de la mínima matriz de representación *N_prev* a la dimensión *N_nva*. La función *adyacentes(v)* regresa un areglo con las seis adyacencias del voxel v, para esto se utiliza el Método 1. La función *tamaño(l)* regresa la longitud de la lista l.

En el método de unión no se verifica que los vox-sólidos no se traslapen,

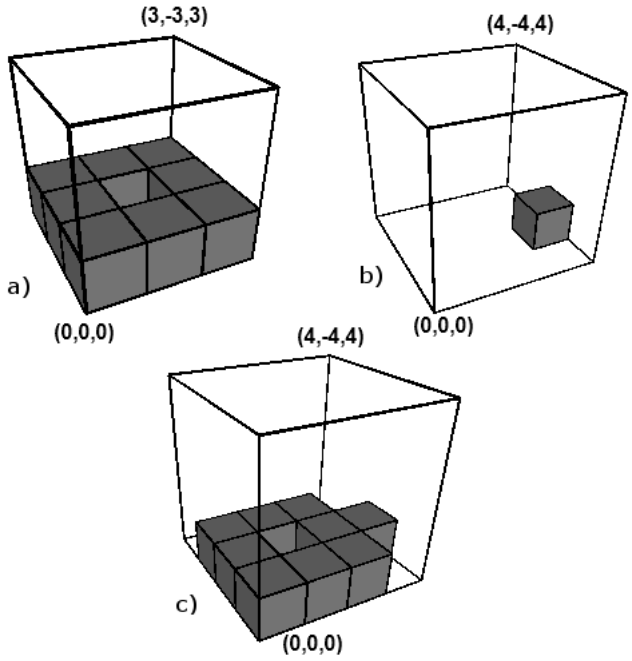


Figura 2.8: Unión de vox-sólidos. (c) es la unión de (a) y (b).

Método 2 Unión de dos Vox-sólidos

Precondición: $V_1 = \{N, L, M\}$, $V_2 = \{N, L, M\}$ dos vox-sólidos.
Postcondición: $U = \{N, L, M\}$ la unión de los vox-sólidos.

1: **if** $V_1.N = V_2.N$ **then**
2: $U.L :=$ unión($V_1.L$, $V_2.L$)
3: **else**
4: $U.N :=$ max($V_1.N$,$V_2.N$)
5: $U.L :=$ renumeraVX($V_1.L$,$V_1.N$,$U.N$)
6: $U.L :=$ unión($U.L$, renumeraVX($V_2.L$,$V_2.N$,$U.N$))
7: **end if**
8: $i := 1$
9: **for** $i \leq$ tamaño($U.L$) **do**
10: $U.M[i] :=$ adyacentes($U.L[i]$,$U.N$)
11: $i := i + 1$
12: **end for**

es decir que tengan voxeles en común, esta consideración no afecta el método de unión, en los pasos 2 o 6, se eliminan elementos duplicados. Por ejemplo, supóngase que se quiere hacer la unión de los vox-sólidos de (b) y (c) de la Figura 2.8, esta unión nos da como resultado el vox-sólido en (c) ya que el método no se ve afectado por esta condición.

Método 3 Renumeración de voxeles.
Precondición: L voxeles a renumerar, N_prev dimensión anterior, N_nva dimensión nueva.
Postcondición: $L2$ la nueva numeración de voxeles.

```
1: if N_prev < N_nva then
2:    d := N_nva - N_prev
3:    diferenciaXpiso = abs((N_nva * N_nva) - (N_prev * N_prev))
4:
5:    i := 1
6:    for i ≤ tamaño(L) do
7:       piso := entera(L[i]/(N_prev * N_prev))
8:       renglon := entera(L[i] / N_prev)
9:       L2[i] := L[i] + (d * renglon) + (diferenciaXpiso * piso)
10:      i := i + 1
11:   end for
12: end if
```

En el Método 3 la función $abs(x)$ regresa el valor absoluto de x. La función $entera(x)$ regresa la parte entera de x.

2.3.1.2. División

En algunos casos es necesario partir un vox-sólido en dos o más partes, para así analizar las características de cada una de las partes, para esto utilizamos la operación de división.

En la Figura 2.9 se muestra gráficamente la división del vox-sólido
$(a)\mathcal{V} = \{4, \{0, 1, 2, 4, 6, 7, 8, 9, 10\}, M\}$ en los vox-sólidos
$(b)\mathcal{V} = \{4, \{0, 1, 2, 4, 6, 8, 9, 10\}, M\}$ y
$(c)\mathcal{V} = \{4, \{7\}, M\}$.

Cuando se desea dividir un vox-sólido se necesita saber cuáles son los voxeles que formarán cada uno de los vox-sólidos en que se separara. Para

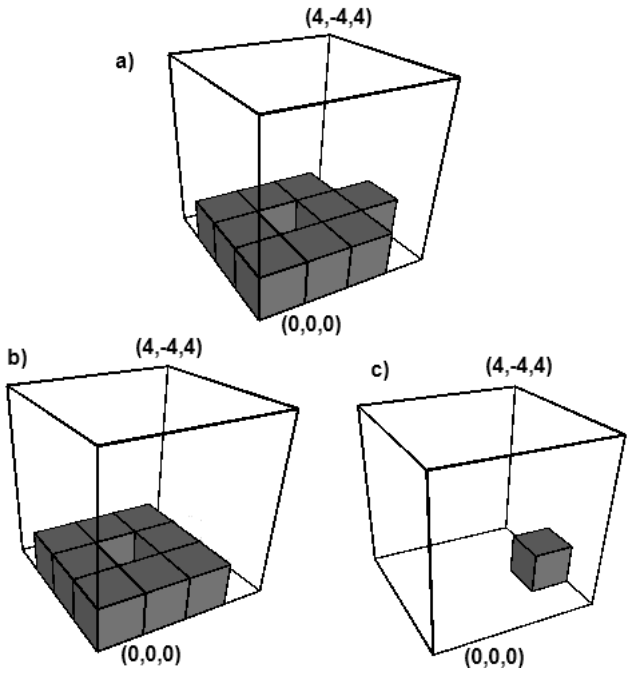


Figura 2.9: División de vox-sólidos. (b) y (c) son de división de (a).

obtener la división de un vox-sólido $\mathcal{V}$ en dos vox-sólidos $\mathcal{V}_1$ y $\mathcal{V}_2$ hay que crear las listas de voxeles L_1 y L_2 correspodientes para v_1 y v_2; la dimensión de la mínima matriz de representación para $\mathcal{V}_1$ y $\mathcal{V}_2$ es la misma que la de $\mathcal{V}$; finalmente hay que obtener la matriz de adyacencias M para los elementos de L de $\mathcal{V}_1$ y $\mathcal{V}_2$. En el Método 4 se describe el pseudo-código de este procedimiento.

Hay que hacer notar que la división genera dos vox-sólidos que tienen la misma dimensión de la matriz mínima de representación de la matriz del vox-sólido inicial.

2.3.1.3. Intersección

Si deseamos saber la intersección de dos vox-sólidos $\mathcal{V}_1$ y $\mathcal{V}_2$, es decir, saber cuáles son los voxeles que tienen en común, basta encontrar la intersección de $\mathcal{V}_1.L$ y $\mathcal{V}_2.L$. En el Método 5 se describe el pseudo-código de este procedimiento.

Método 4 División de un Vox-sólido

Precondición: $V = \{N, L, M\}$ un vox-sólido, L_1 y L_2 los voxeles en que se dividirá V.

Postcondición: $V_1 = \{N, L, M\}$ y $V_2 = \{N, L, M\}$ la división de V.

```
1: V1.N := V.N
2: V1.L := L1
3:
4: V2.N := V.N
5: V2.L := L2
6:
7: i := 1
8: for i ≤ tamaño(V1.L) do
9:    V1.M[i] := adyacentes(V1.L[i],V1.N)
10:   i := i + 1
11: end for
12:
13: i := 1
14: for i ≤ tamaño(V2.L) do
15:   V2.M[i]:= adyacentes(V2.L[i],V2.N)
16:   i := i + 1
17: end for
```

Como se puede ver esta operación es muy sencilla pues sólo hay que verificar la concordancia de números en dos listas.

Método 5 Intersección entre Vox-sólidos

Precondición: $V_1 = \{N, L, M\}$, $V_2 = \{N, L, M\}$ dos vox-sólidos.
Postcondición: L El conjunto de voxeles que tienen en común V_1 y V_2.

1: $L :=$ intersección($V_1.L$,$V_2.L$)

2.3.1.4. Trayectoria Hamiltoniana entre voxeles

Otra de las operaciones que se desearía en los vox-sólidos es que se pueda encontrar una trayectoria hamiltoniana que recorra el vox-sólido, voxel a voxel, en particular, para aquellos que son de tipo toroidal, esta trayectoria hamiltoniana nos ayudará a encontrar una representación planar del mismo. En el Método 6 se muestra la forma de encontrar una trayectoria hamiltoniana.

Hay que hacer notar que una trayectoria inicia y termina en el mismo lugar por lo que el último elemento de la trayectoria debe ser igual al primero, en el presente trabajo eliminaremos el último elemento de la trayectoria para ahorrar espacio.

Método 6 Encontrar una trayectoria hamiltoniana que recorra el vox-sólido

Precondición: $V = \{N, L, M\}$ un vox-sólido toroidal delgado
Postcondición: P la lista que contiene la trayectoria hamiltoniana

1: $P := V.L[0]$
2: **while** tamano(P) $\leq$ $tamano(V.L)$ **do**
3: $t :=$ ultimo-elemento(P)
4: $A :=$ adyacentes(t)
5:
6: **for all** x en A **do**
7: **if** x no esta en P **then**
8: agrega x al final de C
9: break
10: **end if**
11: **end for**
12: **end while**

Por ejemplo, para el vox-sólido que tiene la matriz de adyacencias que aparece en la Figura 2.7 una trayectoria hamiltoniana es
$\mathcal{P} = \{0, 1, 2, 5, 8, 7, 6, 3\}$.

Cabe mencionar que los vox-sólidos también pueden representarse con las estructuras Octree, BSP_tree y B-Rep las cuales se utilizan para representar objetos en $3D$. Aunque estas estructuras están enfocadas a resolver los problemas de sobreposición y composición de objetos en $3D$ también pueden ser adaptadas para la representación de vox-sólidos, el mayor problema que tienen éstas es que se basan en la descripción de objetos en un ambiente $3D$ y no permiten definir de manera simple una relación de adyacencia entre sus componentes.

Capítulo 3

Representación Planar de un Vox-sólido toroidal

El procedimiento para obtener la representación planar de un vox-sólido toroidal, hay que hacer un corte a lo ancho del vox-sólido y extenderlo, después hay que hacer otro corte a lo largo y extenderlo nuevamente.

A manera de ejemplo obtengamos la representación planar de un vox-sólido toroidal simple: *l*a tuerca. En la Figura 3.1 (a) se muestra el vox-sólido toroidal con las caras etiquetadas, sus caras no visibles en la superficie se muestran en la imagen, las aristas remarcadas con blanco son las de corte; en (b) se muestra el el vox-sólido obtenido. Primero se realizó un corte entre las caras {13,12,21,29} y {20,11,28,32} luego se enderezó el toro y se hizo el segundo corte separando las caras {29,30,31,32} de {14,16,18,20}; finalmente, se desdobló.

Hay que hacer notar que el desdoblamiento del toro genera, invariablemente, una deformación de las caras, como se muestra en (b), por lo que en (c) se muestra una esquematización más simple del desdoblamiento de la tuerca.

Obsérvese que las caras de un vox-sólido toroidal pueden ser agrupadas en cuatro conjuntos, que denominaremos *Facetas*, están formados, intuitivamente, de la siguente manera:

Faceta Interior. Son todas las caras que rodean al hoyo.
Faceta Exterior. Son todas las caras opuestas a las del interior y que rodean al vox-sólido.
Faceta Superior. Son todas las caras que se *ven desde arriba.*

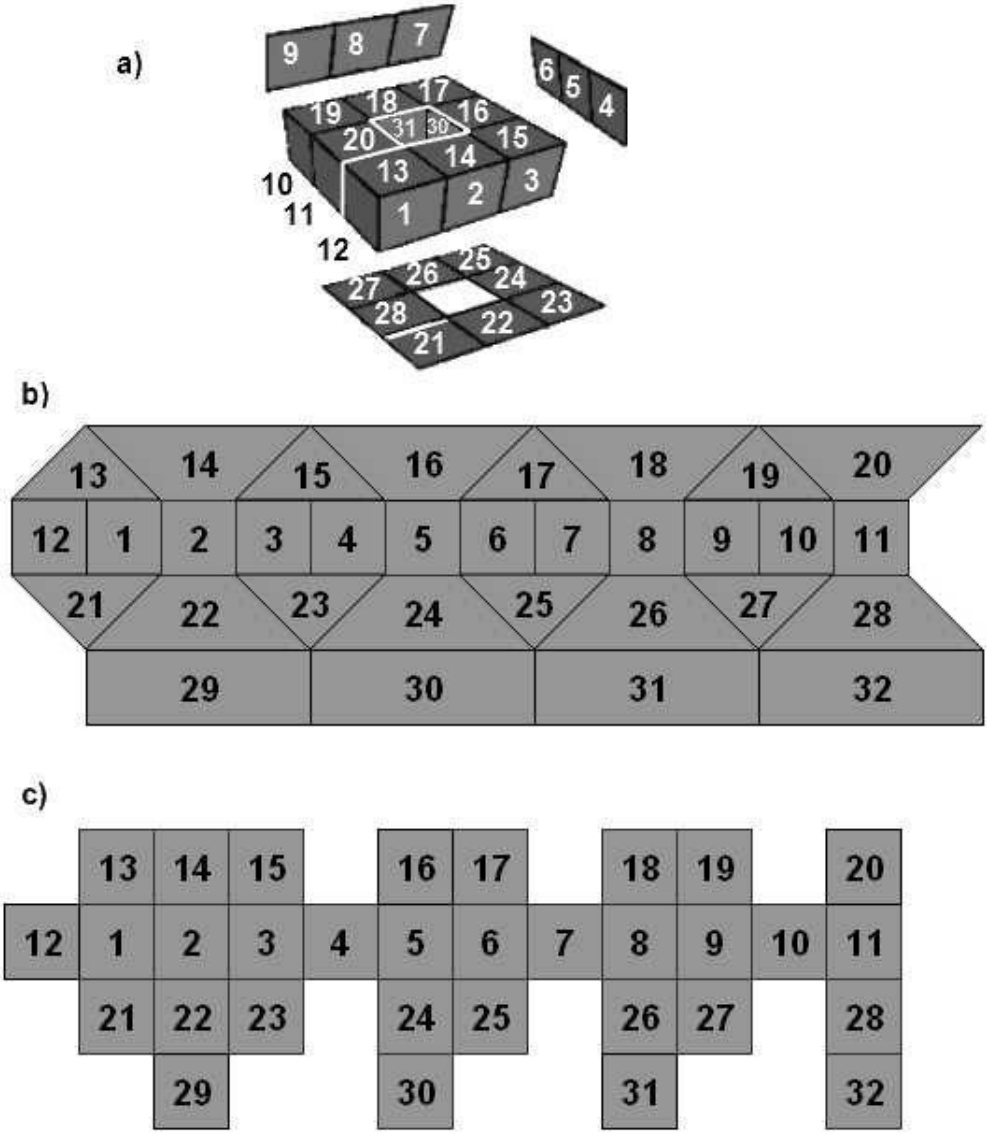


Figura 3.1: Desdoblamiento de un vox-sólido toroidal.

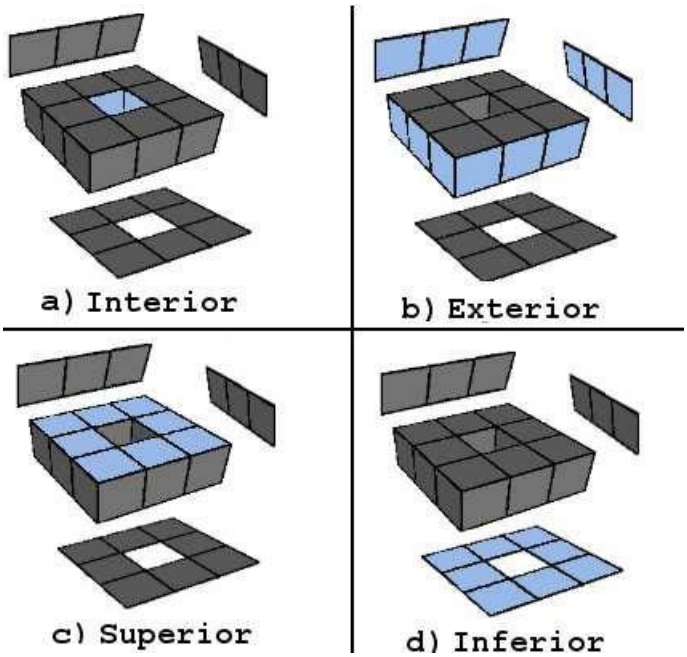


Figura 3.2: Facetas de un vox-sólido toroidal.

Faceta Inferior. Son todas las caras que se *ven desde abajo*, es decir, son opuestas a la Faceta Superior.

En la Figura 3.2 se muestran las cuatro facetas; (a) interior, (b) exterior, (c) superior y (d) inferior; del vox-sólido toroidal. Las facetas se muestran en color claro y las caras no visibles, desde la perspectiva en que se muestra el objeto, aparecen en el fondo de la imagen.

Esta descripción no es exacta por lo que más adelante se re-describirá utilizando la estrategia del laberinto.

Para identificar únicamente cada una de las caras libres de un vox-sólido las enumeramos de la siguiente forma: primero obtenemos una trayectoria hamiltoniana que recorra el vox-sólido y siguiendo cada uno de los voxeles en la trayectoria hamiltoniana enumeramos sus caras libres siguiendo el orden de la enumeración estándar, posteriormente continuamos con los demás voxeles sin reiniciar la cuenta.

En la Figura 3.3 se muestra la enumeración de las caras libres de la

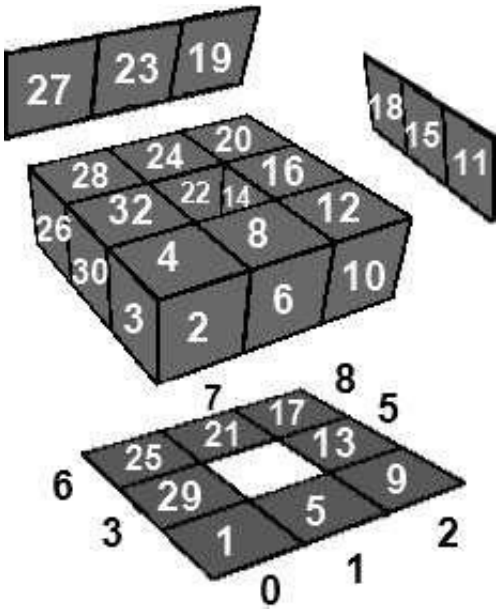


Figura 3.3: Enumeración de las caras libres de un vox-sólido toroidal.

tuerca. La numeración de las caras libres aparece en color blanco, en color negro aparece la enumeración de los voxeles. La trayectoria hamiltoniana de voxeles que se utilizó para la enumeración de las caras libres es $\mathcal{P} = \{0, 1, 2, 5, 8, 7, 6, 3\}$.

Un vox-sólido tiene asociado una gráfica de adyacencias de caras, en particular, la representación planar de un vox-sólido toroidal nos induce una representación planar de su gráfica de adyacencias de caras a la cual llamaremos simplemente *representación planar del vox-sólido.*

Definición 3.0.1. Gráfica Planar. Una gráfica planar es aquella que queda inmersa en un plano sin que sus aristas se intersecten.

Definición 3.0.2. Representación Planar de un Vox-sólido. La representación planar de un vox-sólido toroidal $\mathcal{V}$ es la inmersión de la gráfica de adyacencia de caras de $\mathcal{V}$ en una superficie de género 1, es decir en el toro.

La gráfica de adyacencia de caras utiliza como vértices las caras en la superficie del vox-sólido, éstas coinciden con las *caras libres* del vox-sólido por lo que de aquí en adelante utilizaremos este nombre para referirnos a las caras en la superficie del vox-sólido.

La idea intuitiva de la construcción de la representación planar es obtener las cuatro facetas del vox-sólido y construir la gráfica de adyacencias de caras ordenando cada una de las facetas en diferentes niveles, los vértices se deberán ver ordenados como se muestra en la Figura 3.1 (b).

Existen una amplia variedad de vox-sólidos toroidales los cuales pueden ser clasificados por su delgadez, por el número de pisos que tienen y por su grado. En el presente trabajo se muestra un método para obtener la representación planar de vox-sólidos toroidales delgados con grado dos, aunque también se analizará el comportamiento del método con los otros tipos de voxeles.

Se dice que un vox-sólido toroidal es de un piso si todos los voxeles tienen su cara 6 libre; en general se dice que un vox-sólido $\mathcal{V} = \{N, L, M\}$ es de P pisos si para todo voxel $v \in L$, $v < P * (N * N)$ y existe $v_1 \in L$ tal que $v_1 \geq (P-1) * (N * N)$ para $1 \leq P$.

En las secciones 3.1, 3.2 y 3.3 se trabajará con vox-sólidos toroidales delgados, en la sección 3.4 se analizarán los vox-sólidos no-delgados.

3.1. Vox-sólidos delgados de un piso

Los vox-sólidos toroidales delgados con grado dos de un piso son más simples y nos permiten tener una visión clara de ellos en tan sólo dos vistas, por esta razón analizaremos primero este tipo de vox-sólidos.

3.1.1. Clasificación de voxeles

Con la matriz de adyacencias es posible saber cuáles caras de cada voxel son adyacentes a otro voxel y cuáles no, pero no se puede saber cuáles caras conforman la parte exterior o interior del vox-sólido, es decir, cuáles caras forman cada una de las facetas.

En la Figura 3.1 podemos notar que para los voxeles $0, 1$ y 2 la cara 2 pertenece a la faceta exterior, mientras que para el voxel 7 la cara 2 pertenece a la faceta interior. En general podemos decir que existe un conjunto de voxeles en los que cada una de sus caras pertenece a alguna faceta, en otras palabras, podemos clasificar los voxeles $0, 1$ y 2 con la etiqueta "x" lo que significaría que sus caras $2, 3$ y 4 pertenecen a la faceta exterior; de forma similar podemos clasificar los voxeles 5 y 8 con la etiqueta "z" y a los vo-

Clasificación de Voxeles			Nueva Clasificación
Anterior	Actual	Siguiente	
2	5	1	10
2	4	2	10
4	2	4	9
4	3	4	9
2	5	4	7
2	3	4	7
5	2	5	8
5	4	5	8
4	2	5	5
5	4	2	3

Tabla 3.1: Tabla de adecuación de la clasificación por caras de voxeles.

xeles 6 y 7 con la etiqueta "w" y así sucesivamente para todos los voxeles, las etiquetas asignadas nos indican cuales caras están en que facetas del voxel.

Para obtener la **clasificación por caras** de un voxel dentro del vox-sólido lo hacemos en dos etapas:

independiente. Etiquetamos individualmente cada uno de los voxeles. Utilizamos el número de la primer cara libre según la enumeración de caras del voxel, si éstas se recorren en el orden $\{2, 4, 5, 3, 1, 6\}$.

dependiente. Adaptamos la clasificación independiente de cada voxel según las etiquetas de sus vecinos. Calculamos una trayectoria hamiltoniana para el vox-sólido; recorremos la trayectoria y para cada voxel tomamos los voxeles anterior y siguiente (dentro de la trayectoria hamiltoniana) para adaptar la clasificación del voxel actual según la Tabla 3.1.

En el Método 7 se presenta el pseudo-código para realizar este cálculo. En este procedimiento se considera que la trayectoria es un arreglo y si el índice del arreglo es -1 se refiere al último elemento del arreglo, también se utiliza una función llamada $da\_adecuacion(vx\_Ant, vx\_Act, vx\_Sig)$ que regresa la adecuación de la clasificación del voxel vx_Act según la Tabla 3.1, es decir, modifica la clasificación de la cara de ser necesario.

Método 7 Clasificación de los voxeles.

Precondición: $V = \{N, L, M\}$ un vox-sólido tipo toroidal delgado de un piso.

Postcondición: V con la clasificación de los voxeles de v.

```
1: // Clasificación independiente del voxel.
2: x := 0
3: for x < tamaño(V.L) do
4:    if V.M[2][x] = NULL then
5:       V.L[x].clasificacion := 2
6:       continue
7:    else if V.M[4][x] = NULL then
8:       V.L[x].clasificacion := 4
9:       continue
10:   else if V.M[5][x] = NULL then
11:      V.L[x].clasificacion := 5
12:      continue
13:   else if V.M[3][x] = NULL then
14:      V.L[x].clasificacion := 3
15:      continue
16:   else if V.M[1][x] = NULL then
17:      V.L[x].clasificacion := 1
18:      continue
19:   else if V.M[6][x] = NULL then
20:      V.L[x].clasificacion := 6
21:      continue
22:   end if
23: end for
24:
25: // Calcular en P una trayectoria hamiltoniana para el vox-sólido.
26:
27: // Clasificación dependiente del voxel.
28: i := 0
29: for i < tamaño(P) do
30:    c = da_adecuacion(P[i − 1],P[i],P[i + 1])
31:    if c es diferente de NULL then
32:       V.L[i].clasificacion := c
33:    end if
34: end for
```

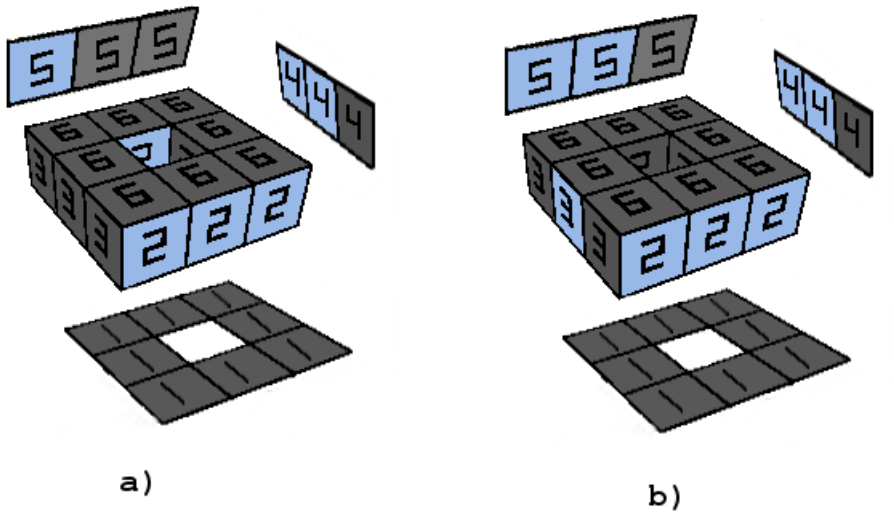


Figura 3.4: Las etapas en la clasificación por caras de voxeles.

Sea $\mathcal{V} = \{N, L, M\}$ un vox-sólido, denotaremos la clasificación por caras con la lista $cv = \{x_1, x_2, \ldots, x_n\}$ donde x_i es la clasificación del voxel que se encuentra en la posición i del arreglo $\mathcal{V}.L$, $1 \leq i \leq n$, $0 \leq n < N * N$.

En la Figura 3.4 tenemos un ejemplo de la clasificación por caras, los voxeles muestran la enumeración de caras, la cara remarcada de un voxel es la cara que lo clasifica. En (a) se muestra la clasificación independiente de los voxeles, sólo el primer paso, que es la siguiente: $cv = \{2, 2, 2, 4, 4, 2, 5, 4\}$; en (b) se muestra el segundo paso, que es la clasificación dependiente, donde se utiliza la trayectoria $\mathcal{P} = \{0, 1, 2, 5, 8, 7, 6, 3\}$, tenemos que la clasificación de los voxeles cambia a la siguiente: $cv = \{2, 2, 2, 4, 4, 5, 5, 3\}$. En el segundo paso se cambia la clasificación de los voxeles 7 y 3, de 2 a 5 y de 4 a 3, respectivamente.

En el ejemplo anterior se puede ver que el cambio en la clasificación se produjo en el voxel 7 porque si tomamos sus voxeles anterior y siguiente (que son el 8 y el 6, respectivamente, según la trayectoria hamiltoniana $\mathcal{P}$) tenemos la tripleta de clasificaciones $(4, 2, 5)$, utilizando la Tabla 3.1 vemos que la clasificación cambia de 2 a 5; lo mismo sucede con el voxel 3 pues la tripleta es $(5, 4, 2)$ que al aplicar la Tabla 3.1 hace el cambio de 4 a 3.

Dado que cada cara del voxel tiene una clasificación podemos saber a

Clasificación de Voxel

Caras	1	2	3	4	5	6	7	8	9	10
1		IF	IF	IF	IF		IF	IF	IF	IF
2		EX	EX	EX	IT			IT	IT	
3		EX	EX	IT	EX		IT		IT	
4		EX	IT	EX	EX			IT		IT
5		IT	EX	EX	EX		IT			IT
6		SU	SU	SU	SU		SU	SU	SU	SU

IF = Inferior, EX= Exterior, IT= Interior, SU = Superior

Tabla 3.2: Facetas a las que pertenecen cada cara del voxel, según la clasificación por caras.

que faceta pertenece cada una de las caras del voxel, por ejemplo, para la clasificación 2 la cara 2 y 3 pertenecen a la faceta exterior, mientras que la cara 6 pertenece a la faceta superior. En la Tabla 3.2 se muestra a que facetas pertenecen cada una de las caras según la clasificación por caras.

3.1.2. Método para la representación planar

Para obtener las facetas de un vox-sólido hay que recorrer una trayectoria hamiltoniana, para cada voxel, tomar su clasificación y utilizar la Tabla 3.2 para saber a que faceta pertenece cada una de las caras del voxel. En el Método 8 se describen los pasos para obtener las facetas.

Las funciones adicionales realizan las acciones descritas por su nombre, *enumera_caras_libres* del vox-sólido; *obten_trayectoria* hamiltoniana que recorre el vox-sólido; *clasifica_voxeles* según el Método 7; $x.caralibre[i]$ regresa la cara i del voxel x sólo si ésta es libre.

Para obtener la representación planar de un vox-sólido hay que identificar las cuatro facetas que componen el vox-sólido, alinearlas verticalmente de arriba hacia abajo en el siguiente orden: superior, exterior, inferior e interior; enumerar las caras libres del vox-sólido; obtener la *gráfica de adyacencia de caras* del vox-sólido donde cada nodo está etiquetado con el número según la enumeración de caras libres del vox-sólido; finalmente para cada nodo trazar una arista a las caras libres que sean adyacentes. En el Método 9 se especi-

Método 8 Obtener las facetas de un vox-sólido.

Precondición: $V = \{N, L, M\}$ un vox-sólido de tipo toroidal delgado. TF la Tabla 3.2.

Postcondición: F un arreglo con las cuatro facetas de V.

```
1: P := obten_trayectoria(V)
2: enumera_caras_libres(V)
3: clasifica_voxeles(V)
4:
5: for all x en P do
6:    i := 1
7:    for i ≤ 6 do
8:       if TF[i][x.clasificacion] = 'IF' then
9:          F[IF].agrega(x.caralibre[i])
10:      else if TF[i][x.clasificacion] = 'EX' then
11:         F[EX].agrega(x.caralibre[i])
12:      else if TF[i][x.clasificacion] = 'IT' then
13:         F[IT].agrega(x.caralibre[i])
14:      else if TF[i][x.clasificacion] = 'SU' then
15:         F[SU].agrega(x.caralibre[i])
16:      end if
17:      i := i + 1
18:   end for
19: end for
```

fica este procedimiento. La función adicional *obtener_facetas* implementa el Método 8.

Método 9 Obtención de la Representación planar

Precondición: $V = \{N, L, M\}$ un vox-sólido tipo toroidal de un piso delgado.

Postcondición: Representación planar de V.

```
 1: P := obten_trayectoria(V)
 2: enumera_caras_libres(V)
 3: clasifica_voxeles(V)
 4: obtener_facetas(V)
 5:
 6: // Alinear horizontalmente todas las caras de cada una de las facetas
    en cuatro niveles verticales, de arriba hacia abajo, en el siguiente orden:
    Superior, Exterior, Inferior e Interior.
 7:
 8: // Agregar a M el primer elemento de la faceta Superior.
 9: while M tenga elementos do
10:    e := pop(M)
11:    e.visitado := true
12:    A := adyacentes(e)
13:    for all x en A do
14:       if x.visitado = false then
15:          M.agrega(x)
16:       end if
17:       Trazar una arista de e a x.
18:    end for
19: end while
```

Nótese que la parte clave de este método está en la identificación de las facetas ya que una vez obtenidas simplemente hay que alinearlas y trazar aristas a cada una de las caras adyacentes. Las facetas son obtenidas mediante la clasificación de los voxeles. Por lo tanto, la parte más importante es obtener una clasificación de los voxeles que nos permita construir las facetas.

3.1.3. Visualización de la representación planar

Si seguimos el método para obtener la representación planar obtendríamos una gráfica como la que se muestra en la Figura 3.5 (a), resulta claro que ésta no es una gráfica planar porque se intersectan las aristas.

Para evitar este problema duplicamos los vértices de la derecha y los ponemos a la izquierda; hacemos algo similar con los vértices de abajo, los duplicamos y los ponemos en la parte superior. Cuando se requiera trazar una arista entre dos vértices v_1, v_2 y además v_2 es uno de los vértices que se duplicaron, siendo v_{2b} su duplicado, entonces dibujamos la arista de v_1 al vértice más cercano; dicho en otras palabras tomamos las distancias $d1 = distancia(v_1, v_2)$ y $d2 = distancia(v_1, v_{2b})$; si $d1 \leq d2$ entonces se traza la arista de v_1 a v_2, de lo contrario se dibuja la arista de v_1 a v_{2b}.

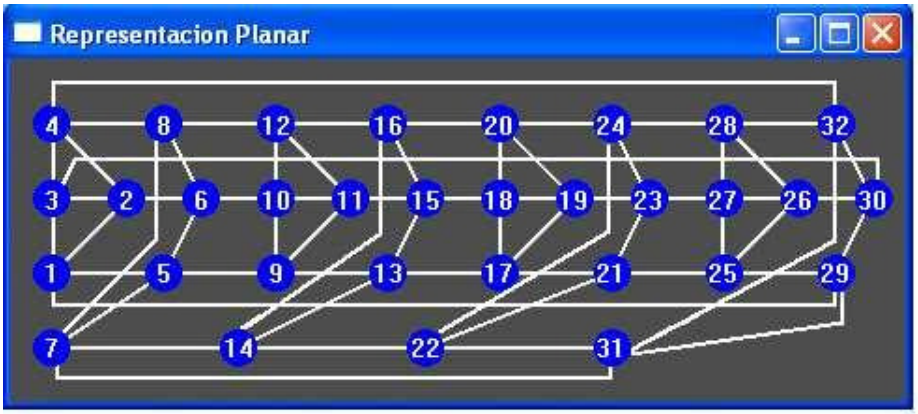
En la Figura 3.5 (b) se muestra la corrección en la visualización de la representación planar, los nodos duplicados en la parte de la izquierda aparecen con la letra b mientras que los nodos duplicados en la parte superior aparecen con la letra a, cuando el nodo duplicado es a y b se etiqueta como c. En adelante la representación planar se muestra con la adecuación de estos nodos.

Para visualizar cómo se relacionan la representación planar y el vox-sólido es necesario ver y recorrer el vox-sólido desde varios ángulos, para esto, y de aquí en adelante, se mostraran dos vistas del vox-sólido la *frontal* y la *trasera*, en la Figura 3.6 se muestrán éstas. Primero se muestran las dos direcciones en que se toman las vistas y después se muestran cada una de ellas. En cada una de las vistas aparece la numeración de las caras libres del vox-sólido.

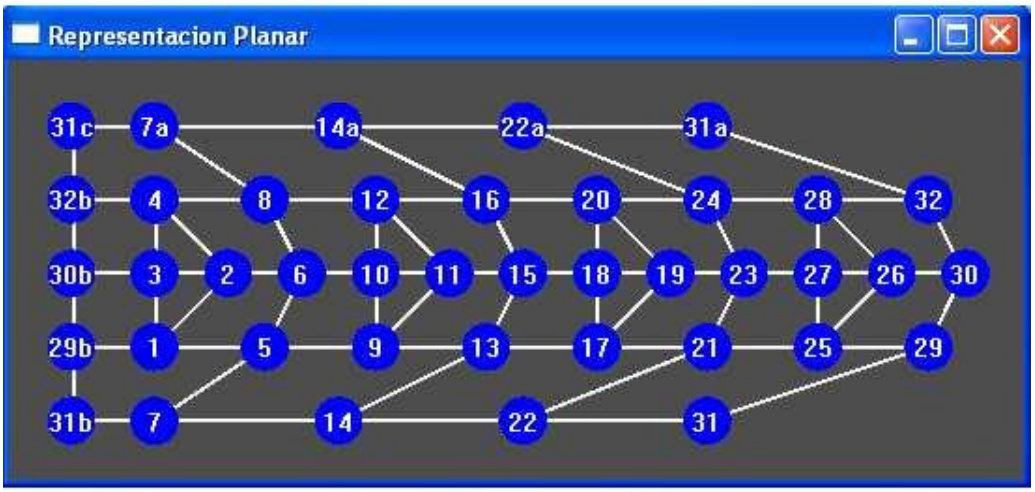
3.1.3.1. Alineación vertical de nodos

En la Figura 3.5 (b) se puede ver que en la gráfica de adyacencias los nodos no están alineados verticalmente, por lo que la visualización se dificulta un poco ya que no se parece mucho al desdoblamiento mostrado en la Figura 3.1, para evitar este desfase se propone cambiar la forma en que se realiza el ordenamiento de los nodos.

La idea principal para ordenar los nodos es tomar el primer elemento en la faceta exterior al cual llamaremos *NE*, colocarlo en la gráfica, encontrar las caras adyacentes de *NE* y de éstas colocar la cara que pertenece a la faceta



a) Representación simple



b) Con adecuación de los nodos

Figura 3.5: Adecuación de la representación planar

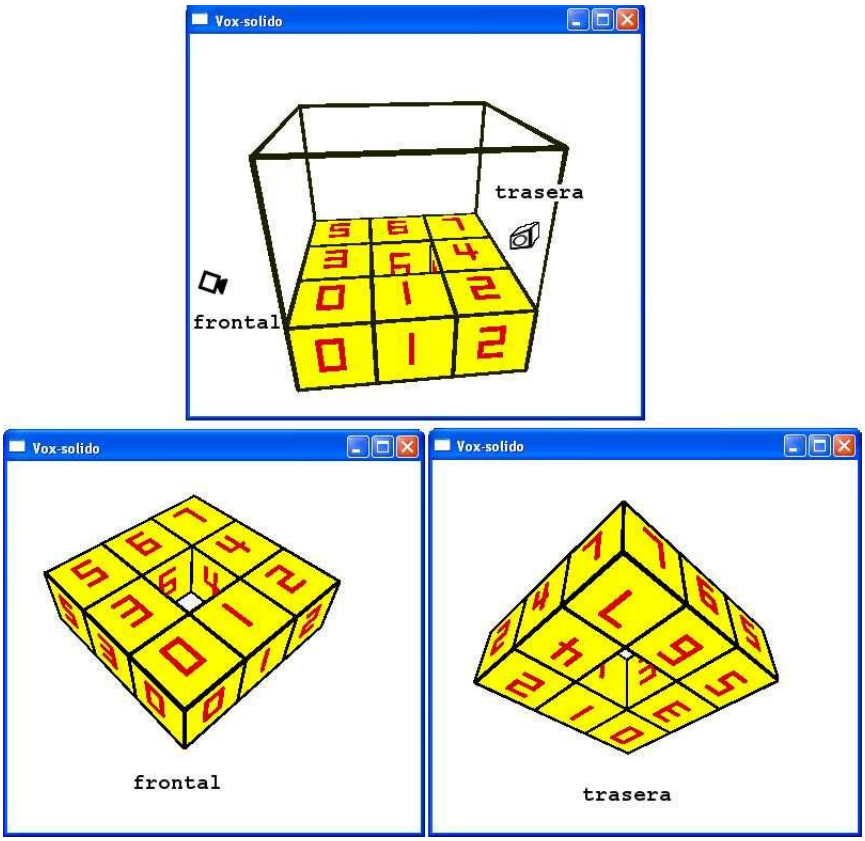


Figura 3.6: Vistas de un vox-sólido

superior arriba de *NE*, poner la cara que pertenece a la faceta inferior exactamente debajo de *NE* y la cara que pertenece a la faceta interior colocarla debajo de la cara que esta en la faceta inferior. El mismo procedimiento se realiza para los demás nodos en la faceta exterior.

En el Método 10 se muestra el pseudo-código para realizar este procedimiento; la función auxiliar $siguiente\_nodo\_no\_visitado(faceta)$ obtiene el siguiente nodo no visitado de la $faceta$ que se da como argumento; la función $caras\_libres(x, fac)$ obtiene las caras adyacentes libres del nodo x en la faceta fac.

Método 10 Alineación vertical de nodos

Precondición: $V = \{N, L, M\}$ un vox-sólido tipo toroidal delgado de un piso, F las facetas de V.

Postcondición: Los nodos alineados vericalmente.

```
1:
2: while No se hayan visitado todos los nodos de la faceta exterior do
3:    e := siguiente_nodo_no_visitado(exterior)
4:    e.visitado := true
5:    alinear e a la derecha del último nodo de la faceta exterior
6:    S := caras_libres(e,superior)
7:    Poner el nodo S arriba de e
8:    I := caras_libres(e,inferior)
9:    Poner el nodo I debajo de e
10:   IF := caras_libres(e,interior)
11:   Poner el nodo IF debajo de I
12: end while
```

Como resultado de esta clasificación tenemos una gráfica que se ve completamente alineada de forma vertical, en la Figura 3.7 se muestra la gráfica planar alineada del *toro*. En el presente trabajo se mostrarán imágenes de la representación planar sin el alineado vertical, pero es posible mediante este método realizar dicha alineación.

3.1.3.2. Lista de adyacencias de la representación planar

Cuando los vox-sólidos son pequeños la representación planar se puede visualizar fácilmente, pero conforme aumenta su tamaño es muy difícil ver claramente la representación planar, en el presente trabajo se mostrará la representación planar gráficamente cuando sea sencillo visualizarla, cuando

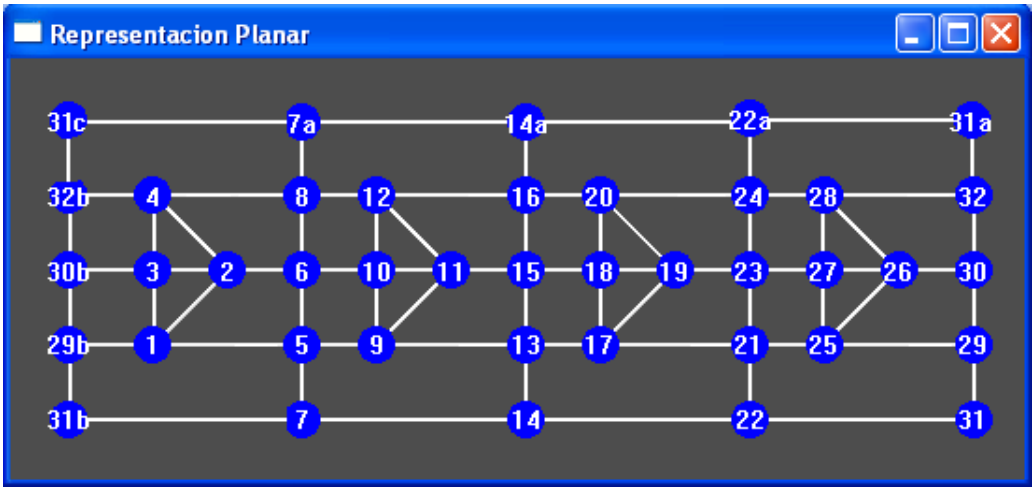


Figura 3.7: Gráfica planar alineada verticalmente del vox-sólido *toro*.

no, se mostrará una lista de adyacencias.

La lista de adyacencias es la lista de todas las aristas que conforman la representación planar del voxel, cada arista está compuesta por la pareja de vértices que une la arista, por ejemplo, la pareja $1, 3$ nos indica que entre los vértices 1 y 3 existe una arista. En la Figura 3.8 se muestra la representación planar de aristas de la tuerca.

3.1.4. Casos de estudio

Primero analizaremos el vox-sólido *tuerca*, en la Figura 3.9 se muestra una vista frontal y trasera. En la Figura 3.10 se muestra la representación planar de la tuerca, las facetas se muestran en el siguiente orden de arriba hacia abajo: superior, exterior, inferior e interior. En la figura aparecen vértices con las letras a y b estos son necesarios para mostrar la representación planar evitando que se crucen las aristas pero no existen en el vox-sólido. En la Figura 3.8 se presenta la lista de aristas para la *tuerca*.

Ahora analicemos una modificación de la tuerca a la que llamaremos *tuerca semi-torcida* y cuya representación es
$\{4, \{0, 1, 2, 3, 4, 7, 8, 9, 11, 13, 14, 15\}, M\}$.

En la Figura 3.11 se muestra la vista frontal y trasera. En la Figura 3.12

4,2	6,10	26,25	15,13
4,3	30,29	26,27	15,18
4,8	30,26	11,9	20,18
4,32	12,10	11,15	20,19
2,1	12,11	16,14	13,17
2,3	12,16	16,15	21,17
2,6	7,5	16,20	21,23
3,1	7,31	14,13	23,19
3,30	7,14	14,22	18,17
8,6	31,29	22,21	18,19
8,12	31,22	22,24	19,17
8,7	28,26	24,22	
32,30	28,24	24,20	
32,31	28,27	24,23	
32,28	5,9	27,25	
1,5	29,25	27,23	
1,29	10,9	9,13	
6,5	10,11	25,21	

Figura 3.8: Lista de adyacencias de la representación planar de la *tuerca*.

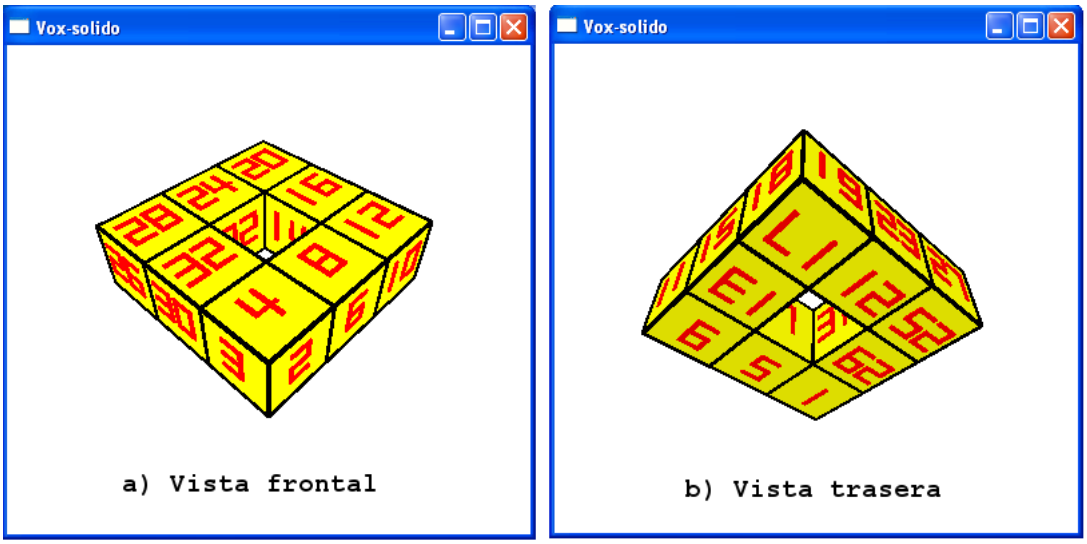


Figura 3.9: Vista frontal y trasera de la *tuerca*.

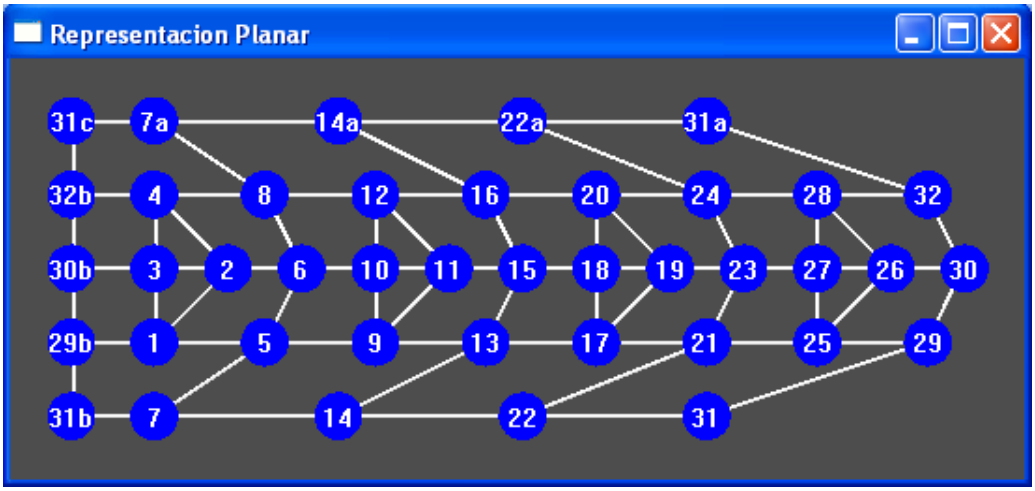


Figura 3.10: Representación planar de la *tuerca*.

se muestra la representación planar de la tuerca semi-torcida, en esta figura aparecen nodos con la letra a y b estos son necesarios para completar la representación planar pero no existen en el vox-sólido.

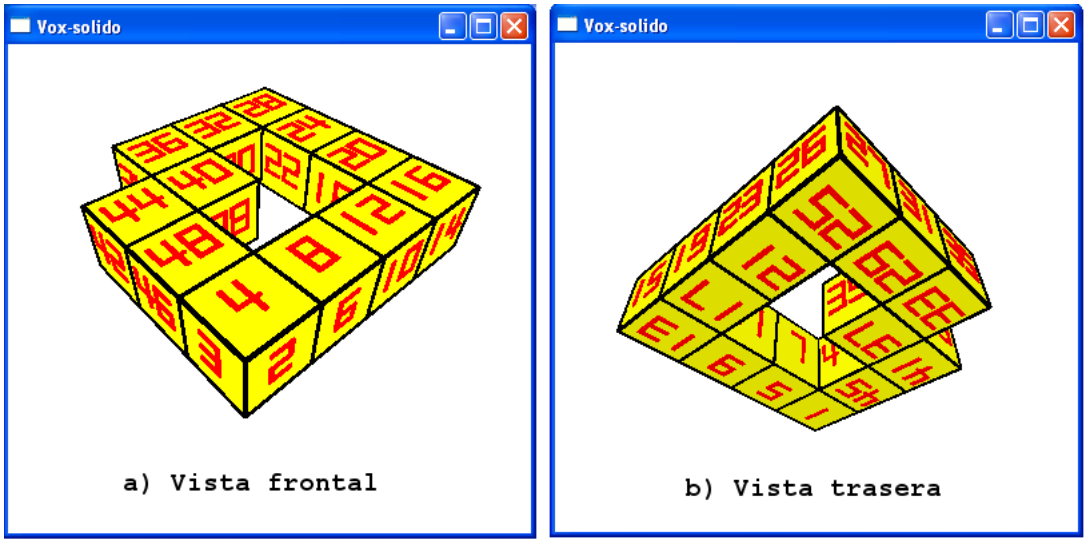


Figura 3.11: Vista frontal y trasera de la *tuerca semi torcida*.

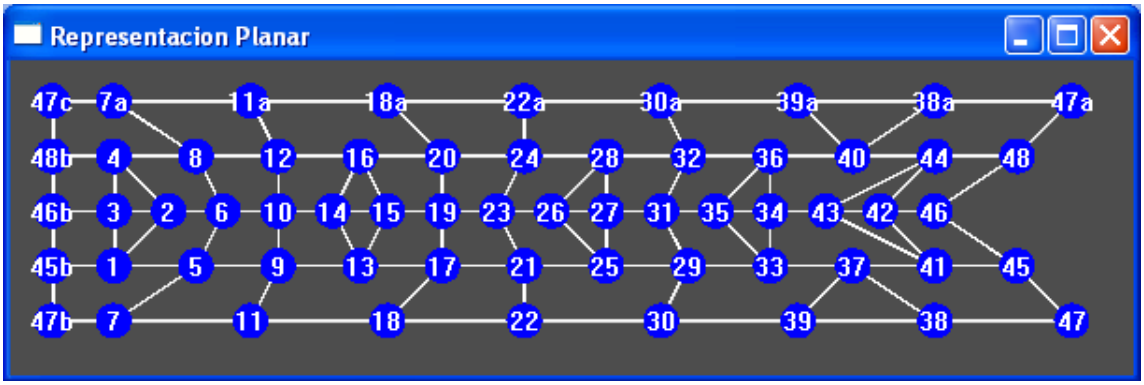


Figura 3.12: Representación planar de la *tuerca semi torcida*.

3.2. Vox-sólidos delgados de más de un piso

Para extender el método a voxeles de más de un piso simplemente tenemos que extender la clasificación de los voxeles (Tabla 3.1). Para hacerlo debemos de analizar cada uno de los casos y encontrar nuevas clasificaciones para los voxeles que permitan encontrar las facetas.

En esta sección se trabajará con vox-sólidos delgados de más de un piso, pero que continuan siendo de género uno.

3.2.1. Clasificación de voxeles

Haciendo un sencillo análisis tenemos que la diferencia de los vox-sólidos de un piso con los de más pisos es que hay voxeles que están sobrepuestos, es decir, que tienen un voxel arriba o uno abajo.

Para clasificar estos voxeles tomemos la clasificación hecha para voxeles de un piso y consideremos el caso cuando un voxel tiene otro arriba o cuando tiene otro abajo. Si un voxel tiene otro arriba le sumaremos 10 a su clasificación anterior y si tienen un voxel debajo le sumamos 20, si un voxel tiene uno arriba y otro abajo le sumaremos 10 y 20; haciendo esto encontramos que la tabla de adecuación de la clasificación de voxeles se extiende con los datos de la Tabla 3.3 y la tabla que indica a que facetas pertenecen cada una de las caras del voxel con la nueva clasificaciones se extiende con los datos de la Tabla 3.4.

Clasificación de Voxeles			Nueva Clasificación
Anterior	Actual	Siguiente	
22	14	22	4
4	22	2	27
4	33	5	28
5	22	14	29
29	14	2	17
2	22	12	30
30	12	2	18
2	12	22	14
19	22	14	31
19	22	4	31
5	22	14	28
28	14	22	17
17	22	12	32
32	12	2	19
3	22	12	32
14	22	4	31
5	4	22	3
3	22	19	32
22	12	2	19

Tabla 3.3: Anexo a la tabla de clasificación de voxeles.

Clasificación de Voxel

Caras	12	14	17	18	19	22	27	28	29	30	31	32	33
1	IF	IF	IF	IF	IF								IF
2	EX			EX	EX	SU	SU	SU	SU	EX	EX	EX	EX
3	IF	IT	EX	IF	EX	IT		EX	EX	SU	SU	EX	EX
4	IF	EX	IT		IF	EX	EX	IT	IT	SU	EX	SU	EX
5	EX	IF	IF	IT	IF		EX		SU	IT			IF
6						SU	SU	SU	SU	SU	SU	SU	SU

IF = Inferior, EX= Exterior, IT= Interior, SU = Superior

Tabla 3.4: Anexo a la tabla de Facetas.

3.2.2. Casos de estudio

Primero obtengamos la representación planar del vox-sólido denominado *dali*[1] definido por $\{3, \{0, 1, 2, 9, 11, 12, 14, 15, 16, 17\}, M\}$, que se muestra en la Figura 3.13. Su representación planar se muestra en la Figura 3.14[2].

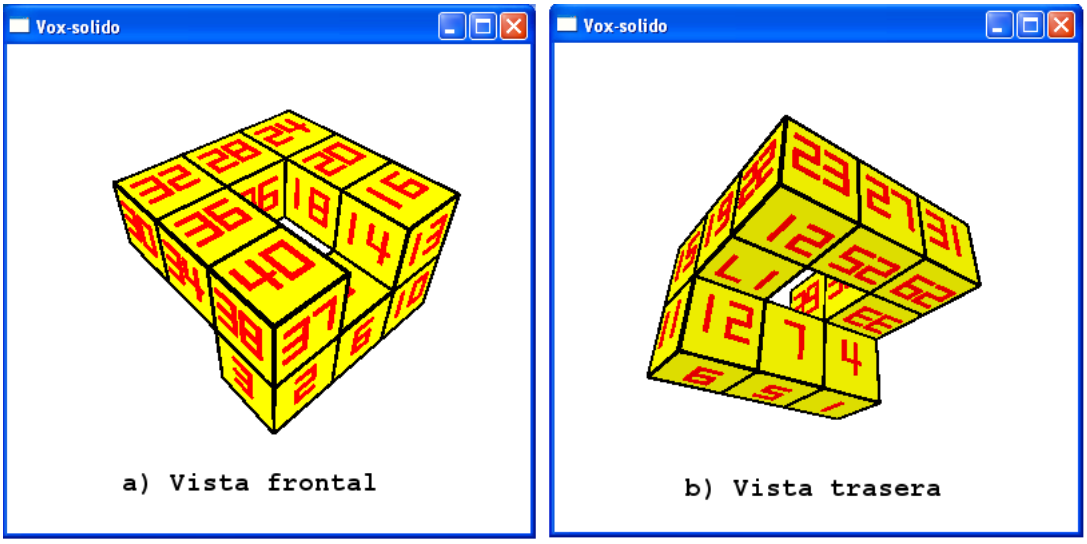


Figura 3.13: Vistas del vox-sólido *dali*.

Como se puede observar la representación planar se obtuvo de forma correcta.

Ahora analicemos vox-sólidos más grandes para ver como se comporta el método. Tomemos un vox-sólido que sea como la *tuerca* pero que en cada lado tenga seis voxeles, su representación es:
$V = \{6, \{0, 1, 2, 3, 4, 5, 6, 11, 12, 17, 18, 23, 24, 29, 30, 31, 32, 33, 34, 35\}, M\}$. En la Figura 3.15 se muestran sus vistas.

Como recordamos, el método para obtener la representación planar nos dice que primero se alinean las caras en las facetas correspondientes y después se trazan las respectivas aristas.

[1] Por similitud a la obra de Salvador Dalí *"La persistencia de la memoria"*, 1931.

[2] Notese que el vertice 7 se puso en la faceta interior y no tiene aristas con otras caras en su misma faceta porque no es adyacente a ninguna de estas. Tambien debe notarse que las caras 29 y 14 estan en la faceta superior.

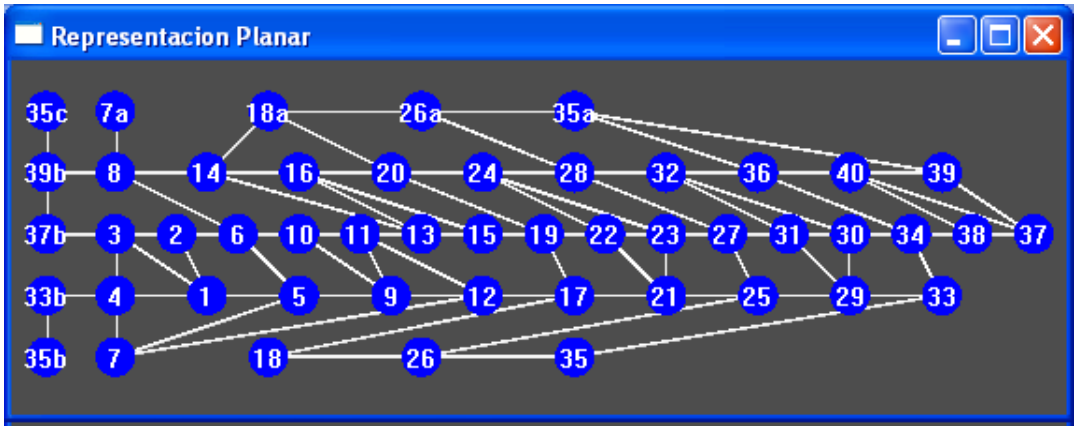


Figura 3.14: Representación planar del vox-sólido *dali*.

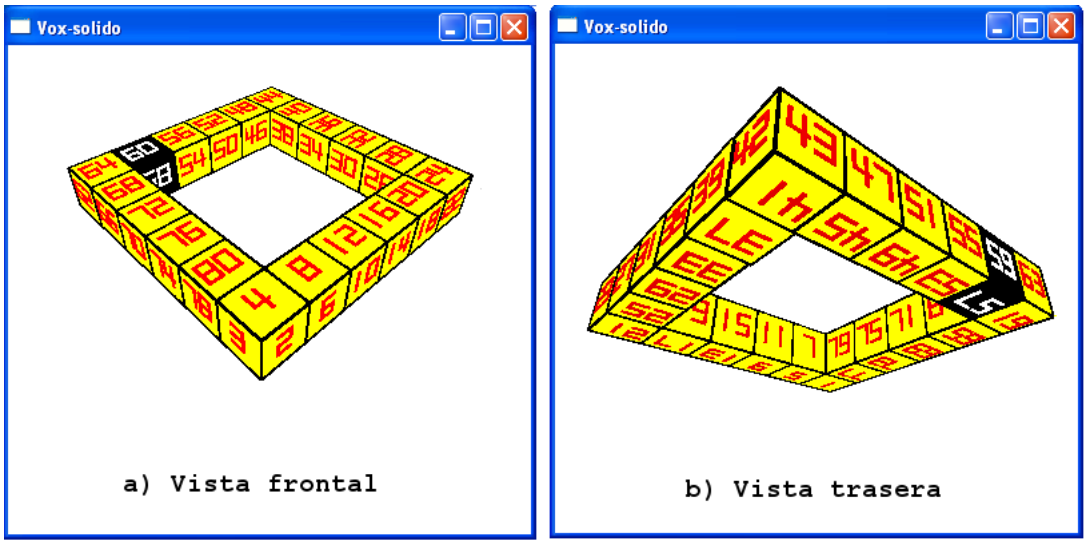


Figura 3.15: Vistas del voxeles *Tuerca Larga*.

Para este ejemplo, tomamos una foto del proceso en el ordenado de las caras en sus facetas, en la Figura 3.16 (a) se muestra dicha foto, pero como no se llega a notar a detalle el número de los nodos se partió por la mitad, en (b) y (c) se muestra un acercamiento a la parte izquierda y derecha, respectivamente.

En este caso se encontró un problema, en la Figura 3.15 está remarcado un voxel que tiene una cara etiquetada con el número 59, dicha cara no aparece

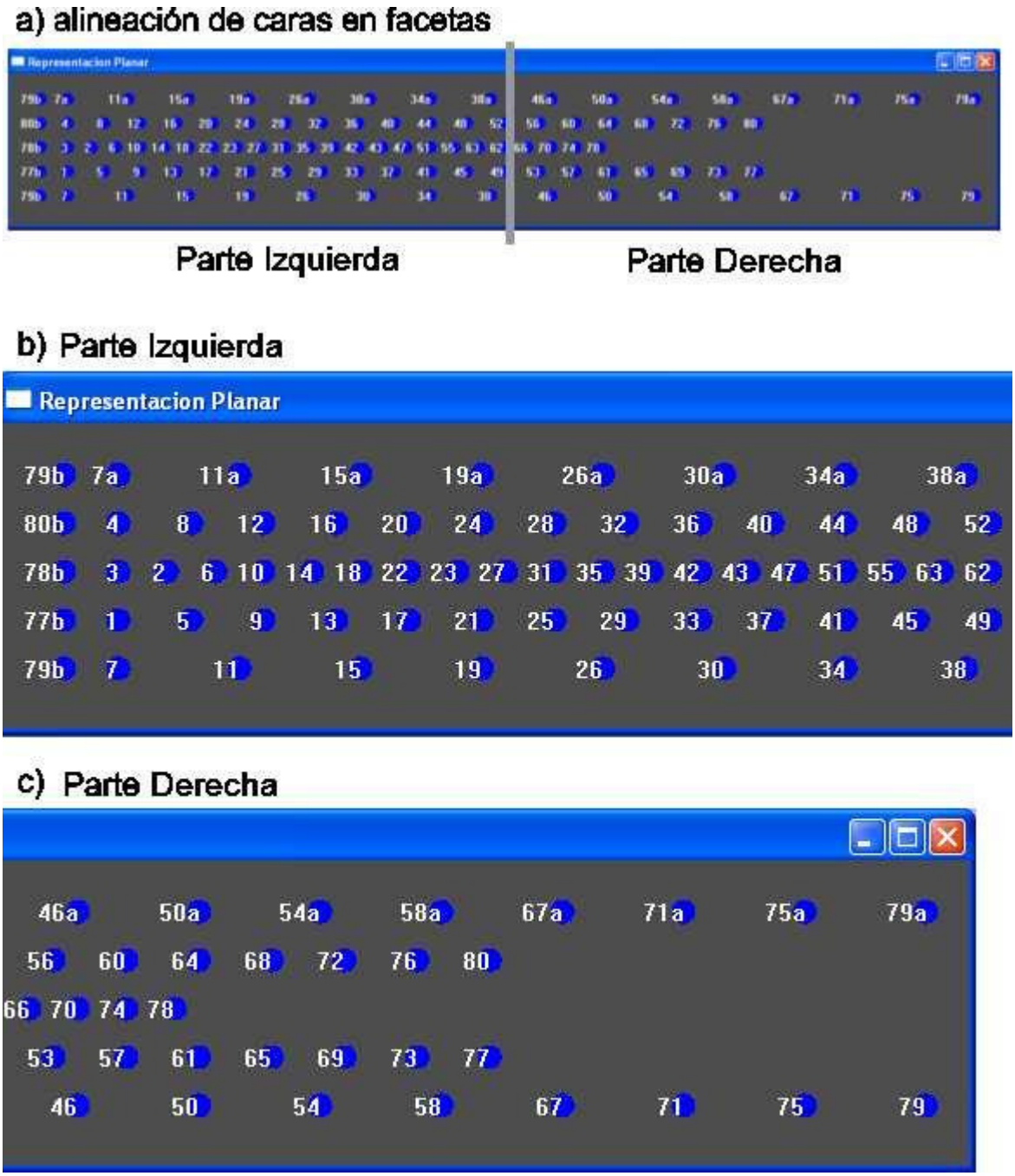


Figura 3.16: Vista de los nodos del voxel *Tuerca Larga*.

en ninguna de las facetas de la Figura 3.16. La causa de este problema es que dicho voxel no fue clasificado de forma adecuada, si analizamos el proceso de la clasificación podemos ver que el voxel en referencia está clasificado con el número 8, si tomamos las facetas a las que pertenecen cada una de las caras del voxel con tipo 8, como se indica en la Tabla 3.2, tenemos que la cara 5 no pertenece a ninguna faceta, por lo que se calcula erróneamente la faceta para esta cara.

El problema en este caso es la clasificación del voxel ya que no obtiene de forma adecuada las facetas, este problema se puede evitar si se agrega una nueva clasificación para este caso en particular. Al analizar diferentes vox-sólidos, sobre todo los de más de un piso, se encontró que este tipo de problemas es muy frencuente y se deben hacerse adecuaciones específicas para cada vox-sólido lo que implica adaptar la clasificación en cada caso.

En resumen, la actual clasificación no se puede aplicar en forma general a todos los vox-sólidos por lo que hay que encontrar otra forma de clasificar los voxeles.

3.3. Método para vox-sólidos delgados en general

Como se vio en la sección anterior el mecanismo de clasificación no es adecuado para aplicarse en forma general a todos los vox-sólidos toroidales delgados, por lo cual se propone otra clasificación, a la que llamaremos **laberinto**, que soluciona este problema.

Hay que hacer notar que el método para encontrar la representación planar es el mismo que se ha estado utilizando, simplemente se cambiará la forma cómo se clasifican los voxeles.

3.3.1. Clasificación de voxeles de laberinto

Supongamos que el vox-sólido es como un laberinto y podemos entrar en él para visitarlo, utilizamos la técnica de la mano derecha para recorrer el vox-sólido, es decir, ponemos la mano derecha en la pared derecha y hacemos el recorrido siempre de frente y sin despegar la mano de la pared.

Durante el recorrido encontraremos que podemos seguir de frente, dar vuelta a la derecha o la izquierda, subir o bajar; en estos dos últimos casos

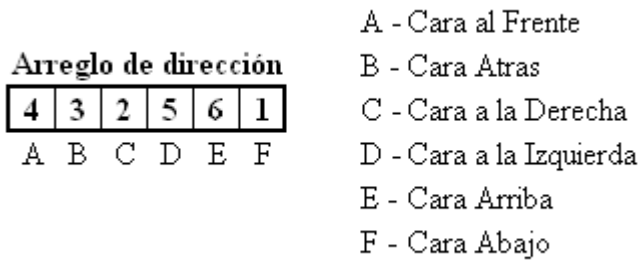


Figura 3.17: Arreglo de dirección de un voxel.

supondremos que existen elevadores lo que nos permite no despegar la mano de la pared de laberinto.

Por las características de los vox-sólidos toroidales delgados tenemos que si recorremos internamente un vox-sólido y del lado derecho se encuentra la parte externa, entonces durante todo el recorrido tendremos del lado derecho siempre la parte externa. De la misma forma podemos hablar la parte interna, superior y exterior de los vox-sólidos se mantienen en la misma dirección durante todo el recorrido.

Durante el recorrido debemos saber exactamente que cara de cada uno de los voxeles está de frente, atrás, a la derecha, a la izquierda, arriba y abajo; para ello almacenaremos por cada voxel un arreglo, al que llamaremos *arreglo de dirección*, con seis posiciones que indicaran el número de la cara que está en esa dirección (siguiendo la enumeración de las caras del voxel). Por ejemplo, en la Figura 3.17 se muestra el *arreglo de dirección* $[4, 3, 2, 5, 6, 1]$ que indica que la cara 4 está al frente, la cara 5 está a la izquierda, la cara 1 está abajo, etcétera.

Durante el recorrido por vox-sólido nos encontramos con los siguientes casos:

0. **De frente** No hay cambio en las condiciones.

1. **Izquierda** En este paso se da vuelta a la izquierda.

2. **Derecha** En este paso se da vuelta a la derecha.

3. **Arriba** En este paso se sube.

4. **Abajo** En este paso se baja.
5. **Previo Bajar** En el siguiente paso se debe bajar.
6. **Previo Subir** En el siguiente paso se debe subir.
7. **Previo Subir-Izquierda** En el siguiente paso se debe subir y luego dar vuelta a la izquierda.
8. **Previo Subir-Derecha** En el siguiente paso se debe subir y luego dar vuelta a la derecha.
9. **Previo Bajar-Izquierda** En el siguiente paso se debe bajar y luego dar vuelta a la izquierda.
10. **Previo Bajar-Derecha** En el siguiente paso se debe bajar y luego dar vuelta a la derecha.
11. **Zig-Zag Derecha** En el siguiente paso se debe dar vuelta a la izquierda y luego a la derecha.
12. **Zig-Zag Izquierda** En el siguiente paso se debe dar vuelta a la derecha y luego a la izquierda.

Entonces clasificaremos a cada uno de los voxeles según se encuentren en cada uno de los casos enumerados anteriormente. El primer elemento de la ruta se clasifica dos veces, una al inicio y otra cuando se llega al final de la trayectoria, en este último caso se re-clasifica según el último elemento de la trayectoria.

Para obtener la faceta a la que pertenece cada una de las caras del voxel utilizamos la clasificación obtenida y el *arreglo de dirección* junto con la Tabla 3.5, la cual sintetiza los casos descritos.

Supondremos que la entrada del vox-sólido para recorrelo internamente es por el voxel cero (o el que tenga el número más pequeño) y por la cara tres.

Como ejemplo, obtengamos la clasificación de los voxeles del vox-sólido *dali* que se muestra en la Figura 3.18; en la parte externa de cada voxel se muestra el número del voxel según la enumeración estándar; en la parte inferior se muestra el arreglo de direcciones y de clasificación del vox-sólido.

Para comenzar a obtener la clasificación entramos por el voxel número 0 y por su cara 3 (en la figura está remarcada la cara donde se entra de frente)

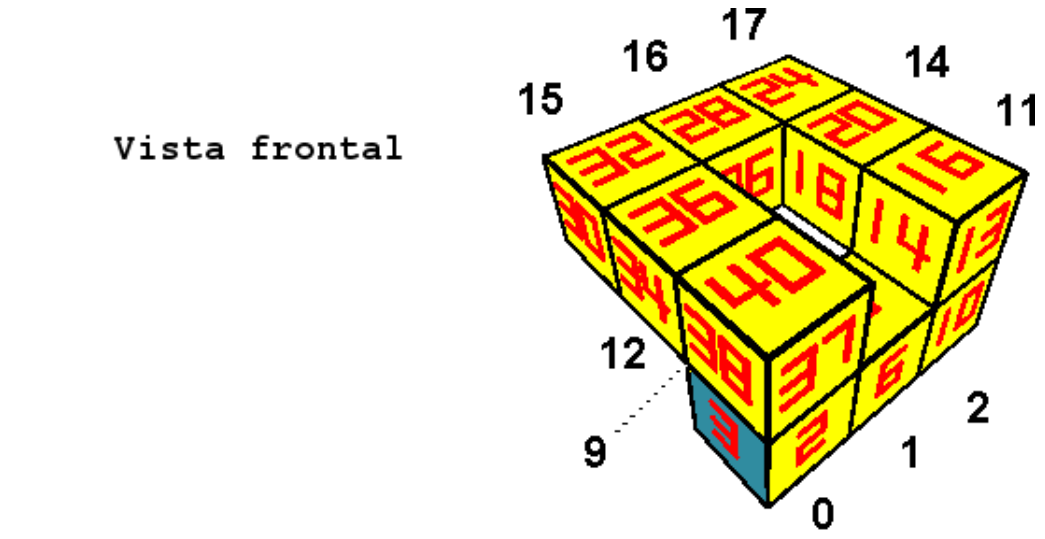
Casos	Facetas de Voxel					
	Atrás	Adelante	Izquierda	Derecha	Arriba	Abajo
0	EX	EX	IT	EX	SU	IF
1	EX	EX	IT	EX	SU	IF
2	EX	EX	IT	EX	SU	IF
3	SU	EX	IT	EX	SU	IF
4	IF	EX	IT	EX	SU	IF
5	EX	SU	IT	EX	SU	IT
6	EX	IF	IT	EX	SU	IF
7	EX	EX	IF	EX	SU	IF
8	EX	EX	EX	IF	SU	IF
9	IF	EX	SU	EX	SU	IF
10	IF	EX	SU	EX	SU	IF
11	EX	IT	IT	EX	SU	IF
12	EX	IT	EX	IT	SU	IF

IF = Inferior, EX= Exterior, IT= Interior, SU = Superior

Tabla 3.5: Faceta a la que pertenece cada voxel con la clasificación de laberinto.

y lo clasificamos como cero; seguimos de frente y al voxel 1 lo clasificamos como 0; cuando llegamos al voxel 2 tenemos que para continuar debemos subir y dar vuelta a la izquierda por lo que lo clasificamos como 7; subimos y al voxel 11 lo clasificamos como 3 porque subimos; seguimos de frente y llegamos al voxel 14 al que clasificamos como 0; al voxel 17 lo clasificamos como 1 pues se da vuelta a la izquierda; a los voxeles 16, 15 y 12 los clasificamos como 0, 1 y 0, respectivamente; al voxel 9 lo clasificamos como 9 porque se debe bajar y dar vuelta a la izquierda, finalmente llegamos al voxel 0 al que re-clasificamos como 4 porque bajamos.

Con esta clasificación y la Tabla 3.5 se puede saber para cada voxel a que facetas pertenecen sus caras por lo que podemos obtener las facetas y en consecuencia la representación planar.



Dirección

Voxel	0	1	2	3	4	5	Clasificación
0	4	3	2	5	6	1	4
1	4	3	2	5	6	1	0
2	4	3	2	5	6	1	7
9	2	5	3	4	6	1	9
11	5	2	4	3	6	1	3
12	2	5	3	4	6	1	0
14	5	2	4	3	6	1	0
15	2	5	3	4	6	1	1
16	3	4	5	2	6	1	0
17	3	4	5	2	6	1	1

Direcciones

0 Cara al Frente
1 Cara Atras
2 Cara a la Derecha
3 Cara a la Izquierda
4 Cara Arriba
5 Cara Abajo

Trayectoria: 0,1,2,11,14,17,16,15,12,9

Figura 3.18: Re-clasificación del vox-sólido *dali* utilizando el método de laberinto.

3.3.2. Casos de estudio

Primero intentemos utilizar la clasificación de *laberinto* para encontrar la representación planar de la *tuerca larga* que se muestra en la Figura 3.15; en la Figura 3.19 se muestra la lista de adyacencias de la representación planar para este vox-sólido obtenida mediante la clasificación de laberinto. Como se puede ver el error de clasificación se corrigió satisfactoriamente, pues la cara 59 tiene correctamente sus adyacencias, las cuales aparecen remarcadas.

4,2	7,11	70,69	64,60	27,31	50,49	44,43
4,3	79,77	70,66	64,63	32,30	50,46	37,41
4,8	79,75	20,18	17,21	32,31	50,52	45,41
4,80	76,74	20,24	65,61	32,36	52,50	45,47
2,1	76,75	20,19	22,21	25,29	52,48	47,43
2,3	76,72	15,13	22,23	30,29	52,51	42,41
2,6	5,9	15,19	62,61	30,34	55,51	42,43
3,1	77,73	71,69	62,63	57,53	35,33	43,41
3,78	10,9	71,67	23,21	57,59	35,39	
8,6	10,14	68,66	23,27	54,53	40,38	
8,12	74,73	68,67	28,26	54,50	40,39	
8,7	74,70	68,64	28,27	54,56	40,44	
80,78	16,14	13,17	28,32	56,54	33,37	
80,79	16,20	69,65	26,25	56,52	38,37	
80,76	16,15	18,17	26,30	56,55	38,46	
1,5	11,9	18,22	58,57	59,55	49,45	
1,77	11,15	66,65	58,54	31,29	49,51	
6,5	75,73	66,62	58,60	31,35	46,45	
6,10	75,71	24,22	60,58	36,34	46,48	
78,77	72,70	24,23	60,56	36,35	48,46	
78,74	72,71	24,28	60,59	36,40	48,44	
12,10	72,68	19,17	63,61	29,33	48,47	
12,16	9,13	19,26	63,59	34,33	51,47	
12,11	73,69	67,65	21,25	34,38	39,37	
7,5	14,13	67,58	61,57	53,49	39,42	
7,79	14,18	64,62	27,25	53,55	44,42	

Figura 3.19: Lista de adyacencias de la representación planar de la *tuerca larga*.

Ahora analicemos un vox-sólido donde el hoyo que tiene en el centro tiene la forma de una cruz, su representación es

$\{5, \{1, 2, 3, 5, 6, 8, 9, 10, 14, 15, 16, 18, 19, 21, 22, 23\}, M\}$. En la Figura 3.20 se muestra la vista frontal y trasera. En la Figura 3.21 se muestra la lista de adyacencias de la representación planar.

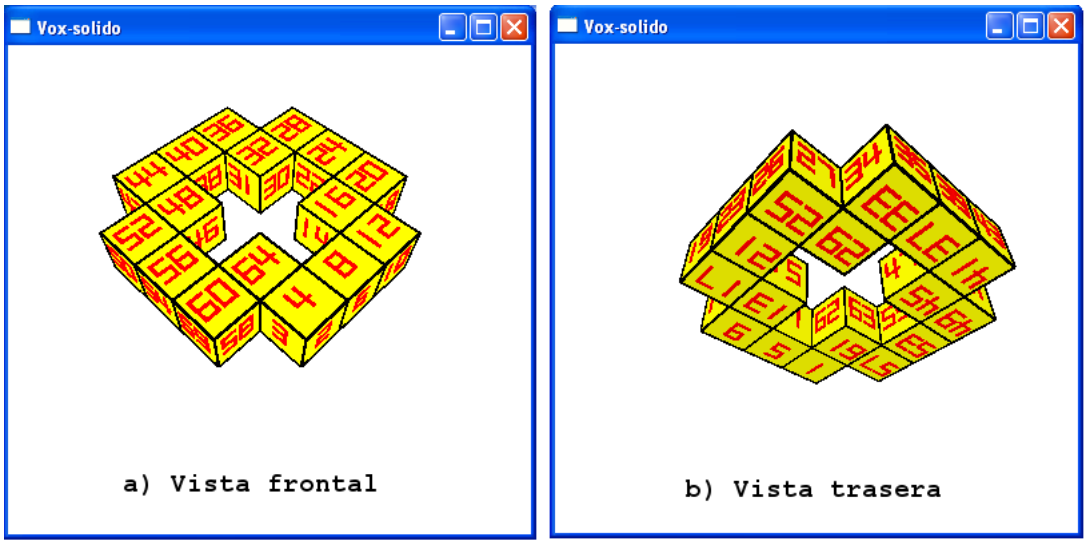


Figura 3.20: Vista frontal y trasera del vox-sólido.

4,2	58,59	59,54	20,19	24,28	44,40	40,36
4,3	58,60	11,9	20,24	22,21	44,43	40,39
4,8	12,10	11,18	15,13	22,30	42,41	43,39
4,64	12,11	16,14	15,22	48,46	42,43	37,33
2,1	12,16	16,20	13,17	48,47	41,37	37,39
2,3	7,5	16,15	52,50	48,44	41,43	36,34
2,6	7,62	14,13	52,48	51,42	38,37	36,35
3,1	7,14	14,15	52,51	45,47	38,31	34,33
3,58	60,59	56,54	46,45	45,41	38,40	34,35
8,6	60,56	56,55	46,47	47,38	25,29	33,35
8,12	62,61	56,52	46,48	21,23	25,26	39,35
8,7	62,63	55,53	49,50	21,25	25,27	
64,60	63,61	55,46	49,45	23,26	26,27	
64,62	63,55	9,13	49,51	28,32	32,30	
64,63	5,9	53,54	50,51	28,26	32,31	
1,5	61,57	53,49	17,19	28,27	32,36	
1,61	10,9	54,50	17,21	30,29	27,34	
6,5	10,11	18,17	19,23	30,31	29,31	
6,10	57,59	18,19	24,22	30,32	29,33	
58,57	57,53	18,20	24,23	44,42	40,38	

Figura 3.21: Lista de adyacencias de la representación planar del vox-sólido.

Como otro ejemplo veamos el vox-sólidos llamado $\mathcal{L}_{16}$ [5]:
$\{4, \{1, 2, 6, 7, 11, 16, 17, 20, 27, 30, 31, 36, 40, 41, 45, 46\}, M\}$, en la Figura 3.22

se muestran sus dos vistas, en la Figura 3.23 se muestra la lista de adyacencias de la representación planar.

Ahora analicemos el vox-sólidos llamado $\mathcal{Q}_{18}$ [5]:
$\{4, \{1, 2, 6, 7, 11, 16, 17, 27, 31, 32, 36, 46, 47, 52, 56, 57, 61, 62\}, M\}$, en la Figura 3.24 se muestran sus dos vistas, en la Figura 3.25 se muestra la lista de adyacencias de la representación planar.

Ahora analicemos el vox-sólidos llamado $\mathcal{F}_{20}$ [5]:
$\{5, \{1, 2, 5, 6, 10, 14, 18, 19, 22, 23, 27, 28, 33, 34, 35, 39, 40, 41, 46, 47\}, M\}$, en la Figura 3.26 se muestran sus dos vistas, en la Figura 3.27 se muestra la lista de adyacencias de la representación planar.

Ahora analicemos el vox-sólidos llamado $\mathcal{V}_{32}$ [5]:
$\{7, \{2, 3, 8, 9, 12, 19, 20, 21, 27, 28, 29, 36, 39, 40, 45, 46, 52, 53, 57, 60, 61, 63,$
$64, 70, 76, 82, 83, 85, 86, 89, 93, 94\}, M\}$, en la Figura 3.28 se muestran sus dos vistas, en la Figura 3.29 se muestra la lista de adyacencias de la representación planar.

Como se puede observar la clasificación de laberinto nos permite encontrar la representación planar de vox-sólidos complejos tanto de un piso como de más pisos, además no se ve afectado por el número de voxeles que tiene el vox-sólido.

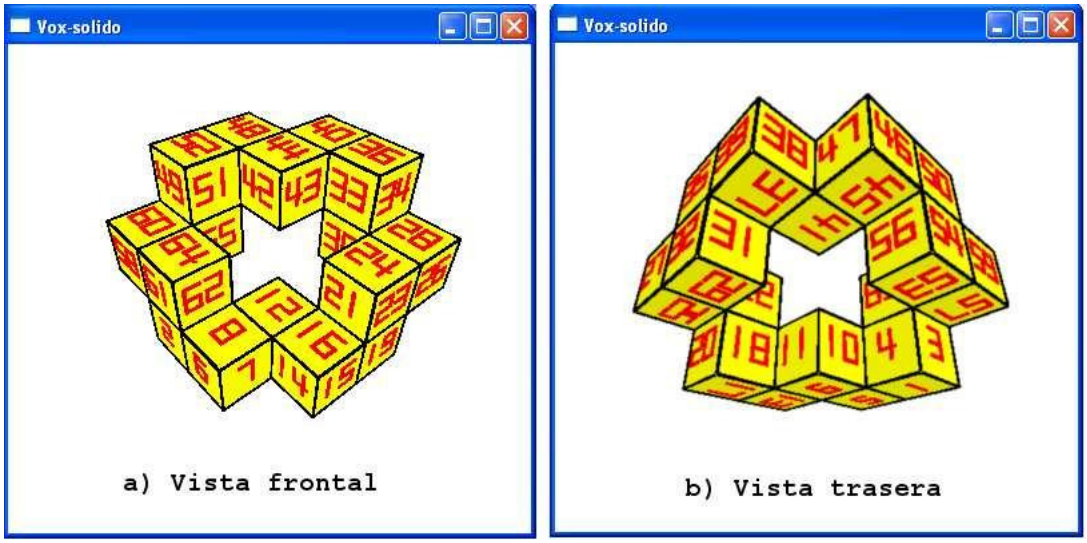


Figura 3.22: El vox-sólido $\mathcal{L}_{16}$.

8,6	64,63	13,17	22,24	25,27	38,40
8,62	14,13	15,19	23,26	46,47	40,36
8,7	14,15	21,22	23,24	46,48	40,39
8,12	14,16	21,23	24,22	41,43	32,35
6,5	10,9	21,24	24,28	41,37	36,33
6,2	10,4	18,17	20,25	47,38	36,35
6,7	10,11	18,20	54,50	47,48	35,39
62,61	16,15	18,22	45,46	43,33	
62,63	16,21	57,59	45,41	43,44	
62,64	11,9	57,53	45,47	44,42	
7,5	11,18	59,54	42,41	44,48	
7,14	1,3	53,54	42,43	44,43	
12,10	1,4	53,56	42,44	44,40	
12,16	9,13	56,54	52,50	29,31	
12,11	3,4	56,45	52,51	29,32	
5,1	3,57	51,49	52,48	31,32	
5,9	58,57	51,42	50,46	31,37	
2,1	58,59	51,52	30,29	33,34	
2,3	58,60	49,50	30,31	33,36	
2,61	4,63	49,51	30,33	27,32	
61,58	55,53	49,52	26,25	34,33	
61,64	55,56	17,19	26,27	34,35	
63,4	55,51	17,20	26,28	34,36	
63,55	60,59	19,20	28,34	37,38	
63,64	60,49	19,23	28,27	37,39	
64,60	13,15	22,30	25,29	38,39	

Figura 3.23: Lista de adyacencias de la representación planar del vox-sólido $\mathcal{L}_{16}$.

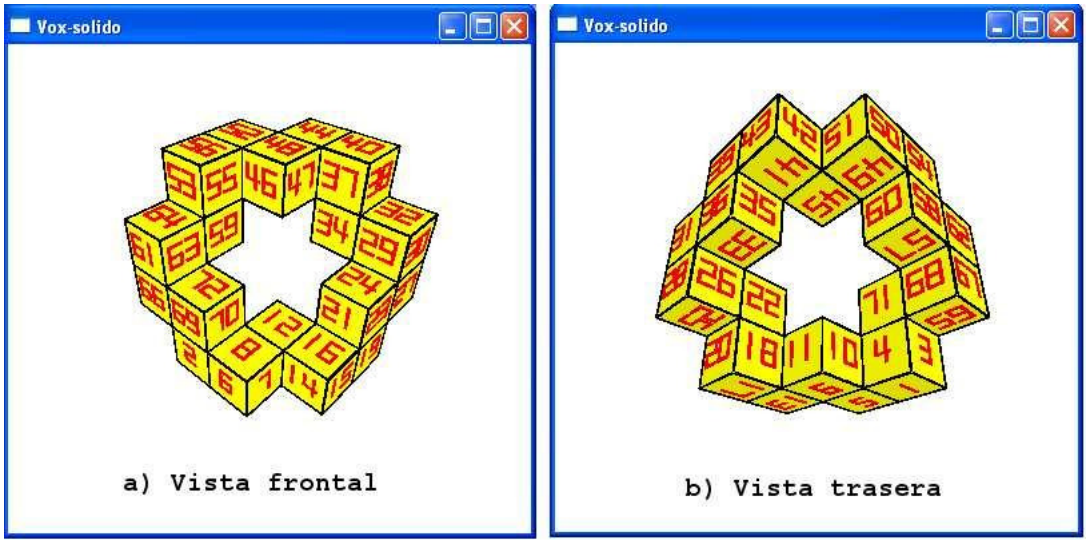


Figura 3.24: El vox-sólido $\mathcal{Q}_{18}$.

8,6	72,71	13,15	23,27	54,56	47,48
8,70	14,13	13,17	23,24	49,50	48,46
8,7	14,15	15,19	24,22	49,45	48,52
8,12	14,16	21,22	24,29	49,51	48,47
6,5	10,9	21,23	20,25	46,45	48,44
6,2	10,4	21,24	62,58	46,47	31,36
6,7	10,11	18,17	58,60	46,48	35,36
70,69	16,15	18,20	58,54	56,55	35,41
70,71	16,21	18,22	60,49	56,52	36,39
70,72	11,9	65,67	55,53	28,31	37,38
7,5	11,18	67,62	55,46	33,34	37,40
7,14	1,3	61,62	55,56	33,35	38,37
12,10	1,4	61,64	53,54	33,36	38,39
12,16	9,13	57,58	53,55	30,31	38,40
12,11	3,4	57,59	53,56	30,32	41,42
5,1	3,65	57,60	26,25	34,35	41,43
5,9	66,65	59,60	26,28	34,37	42,43
2,1	66,67	59,55	26,33	32,38	42,44
2,3	66,61	64,62	27,25	32,31	44,40
2,69	4,71	64,53	27,28	50,51	44,43
69,66	68,65	17,19	27,30	50,52	39,43
69,72	68,67	17,20	29,34	45,47	39,40
71,4	68,57	19,20	29,30	45,41	40,37
71,68	63,61	19,23	29,32	51,42	
71,72	63,59	22,26	25,28	51,52	
72,63	63,64	22,24	54,50	47,37	

Figura 3.25: Lista de adyacencias de la representación planar del vox-sólido $\mathcal{Q}_{18}$.

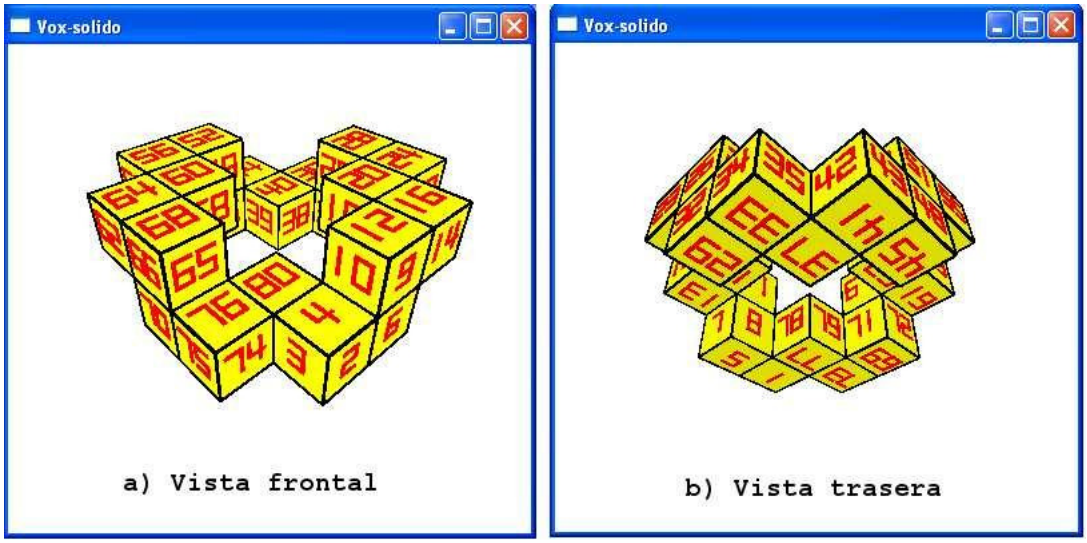


Figura 3.26: El vox-sólido $\mathcal{F}_{20}$.

4,2	11,18	16,15	68,67	30,32	36,35	47,48
4,3	11,12	16,20	68,64	31,29	54,53	46,45
4,10	12,16	65,66	72,61	31,38	54,55	50,44
4,80	12,11	65,67	22,21	27,36	54,56	50,49
2,1	76,75	65,68	22,23	27,25	53,47	50,51
2,3	76,65	71,69	22,24	27,26	53,55	50,52
2,6	78,77	71,72	21,23	27,28	49,46	52,49
3,1	78,8	71,67	21,30	28,25	49,50	52,51
3,74	78,79	13,15	25,31	28,26	49,52	41,45
10,9	79,77	13,17	25,27	63,54	56,52	41,42
10,11	79,71	69,70	25,28	57,59	56,55	41,43
10,12	5,7	69,72	24,23	57,53	33,37	44,42
80,76	5,8	70,72	24,28	59,49	33,34	44,43
80,78	77,73	70,66	62,61	59,60	33,35	42,43
80,79	7,8	15,22	62,63	60,58	34,35	51,48
1,5	7,13	17,21	62,64	60,59	37,39	45,48
1,77	73,75	17,19	58,57	60,56	37,41	48,43
6,5	73,69	19,25	58,59	26,32	39,46	
6,7	75,70	19,20	58,60	29,32	39,40	
6,9	14,13	20,18	64,60	29,33	40,38	
74,73	14,15	20,24	64,63	32,34	40,39	
74,75	14,16	20,19	61,57	38,37	40,44	
74,76	8,78	66,62	61,63	38,39	35,42	
9,14	18,17	66,68	23,26	38,40	55,51	
9,12	18,19	67,58	30,29	36,40	47,45	
11,8	18,20	67,68	30,31	36,34	47,46	

Figura 3.27: Lista de adyacencias de la representación planar del vox-sólido $\mathcal{F}_{20}$.

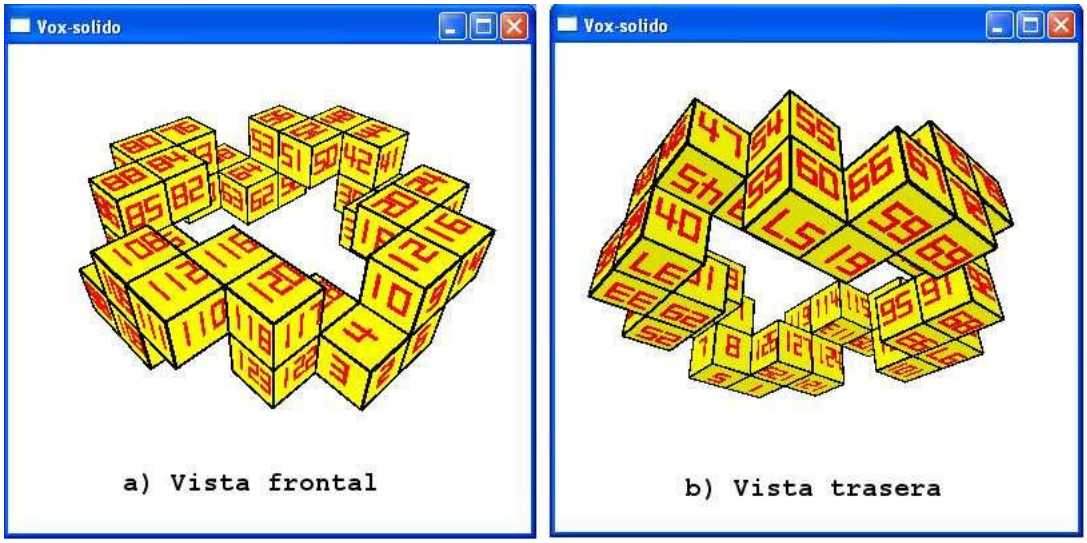


Figura 3.28: El vox-sólido $\mathcal{V}_{32}$.

4,2	125,121	116,114	107,105	95,96	86,87	71,72
4,3	7,8	116,115	107,106	96,94	86,88	79,75
4,10	7,13	110,109	107,108	96,95	82,83	79,80
4,128	121,123	110,111	108,105	96,85	82,84	80,76
2,1	121,124	110,112	108,106	99,90	88,84	57,61
2,3	123,124	21,26	105,103	37,39	88,87	57,59
2,6	123,118	21,22	28,34	37,40	49,51	57,60
3,1	117,118	21,24	29,33	40,39	49,58	62,61
3,122	117,120	27,25	29,31	40,45	47,54	62,63
10,9	14,13	27,26	31,38	42,50	51,53	62,64
10,11	14,15	27,30	36,34	42,44	51,52	59,60
10,12	14,16	23,32	36,35	39,43	52,50	55,60
128,119	8,126	23,22	36,41	43,46	52,51	64,62
128,126	18,17	23,24	101,103	43,44	52,56	64,63
128,127	18,19	24,22	101,97	44,42	87,78	64,68
1,5	18,20	109,111	103,98	44,48	83,73	70,69
1,125	16,15	109,102	94,93	89,90	83,84	70,63
6,5	16,20	106,104	94,95	89,91	77,78	74,68
6,7	114,113	106,107	94,96	89,92	77,71	74,73
6,9	114,115	106,108	100,98	91,92	77,79	74,75
122,121	114,116	112,111	100,96	91,81	84,82	74,76
122,123	120,118	112,108	100,99	85,86	84,83	76,73
122,117	120,116	111,105	34,33	85,82	84,80	76,75
9,14	124,113	26,25	34,35	85,88	58,57	69,65
9,12	13,15	26,28	33,35	90,92	58,62	69,72
11,8	13,17	22,28	33,37	90,86	58,59	72,67
11,18	118,110	25,28	38,37	45,49	54,59	72,75
11,12	15,21	25,29	38,40	45,46	54,55	61,63
12,16	17,27	30,29	38,42	45,47	54,56	61,65
12,11	17,19	30,31	35,39	50,49	53,64	60,66
119,117	19,23	30,32	41,42	50,51	53,55	68,66
119,114	19,20	32,30	41,43	50,52	53,56	68,67
119,120	20,18	32,36	41,44	46,47	56,55	65,66
126,125	20,24	32,31	97,98	46,48	78,79	65,67
126,8	20,19	102,101	97,93	48,52	78,80	67,66
126,127	113,109	102,103	97,99	48,47	73,70	
127,125	113,115	102,104	98,99	92,87	73,74	
127,124	115,106	104,101	93,95	81,82	73,76	
5,7	115,116	104,94	93,89	81,83	71,69	
5,8	116,112	107,100	95,91	81,77	71,70	

Figura 3.29: Lista de adyacencias de la representación planar del vox-sólido $\mathcal{V}_{32}$.

3.4. Vox-sólidos no-delgados.

Hasta este momento hemos trabajado con vox-sólidos toroidales delgados, para encontrar la representación planar de vox-sólidos toroidales no-delgados o gordos utilizamos la misma idea que para los vox-sólidos delgados sólo que se debe modificar la forma en que se realiza la clasificación de los voxeles.

Existen diferentes tipos de vox-sólidos troidales no-delgados de un piso, en la Figura 3.30 se pueden ver dos ejemplos, en (a) se muestra un vox-sólido no-delgado que es la unión de dos vox-sólidos toroidales delgados, uno exteno y otro interno; en contra parte el vox-sólido en (b) no está formado por varios vox-sólidos toroidales delgados.

En la Figura 3.31 se muestran los dos vox-sólidos que conforman el vox-sólido de la Figura 3.30 (a), se puede ver que el vox-sólido interno está siendo rodeado por otro más grande y que se encuentra en la parte externa.

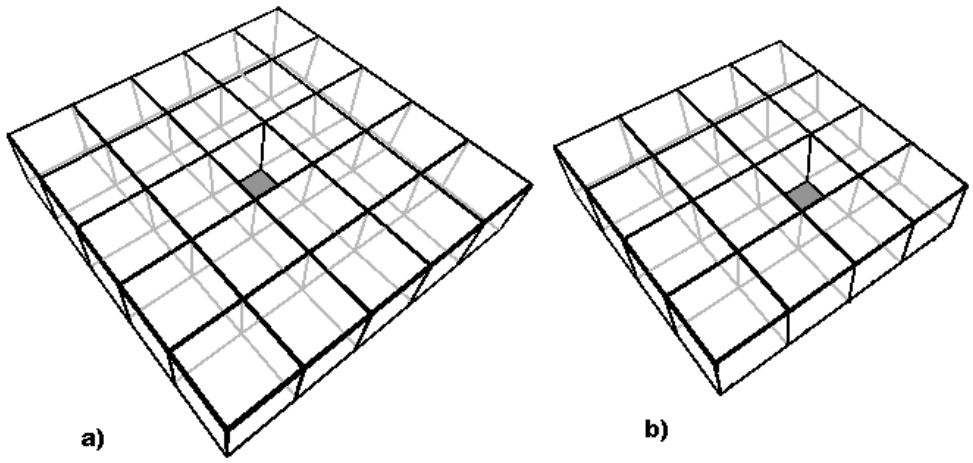


Figura 3.30: Vox-sólidos no-delgados.

Se dice que un vox-sólido rodea al hoyo o a otro vox-sólido si todas las caras de su faceta interna esta en contacto con el hoyo o el otro vox-sólido.

Definición 3.4.1. Vox-sólido envolvente. Sea $\mathcal{VG}$ un vox-sólido no-delgado de un piso, al mínimo conjunto de voxeles que sea un vox-sólido toroidal delgado y que esté rodeando al hoyo se le llamará como vox-sólido *envolvente*, al resto del vox-sólido se le denominará el vox-sólido *externo* de $\mathcal{VG}$.

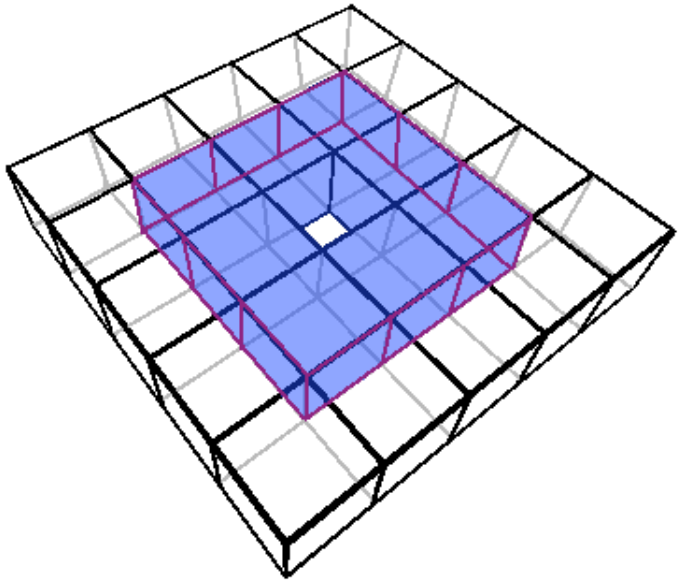


Figura 3.31: Vox-sólido no-delgado como unión de dos vox-sólidos delgados.

En la Figura 3.31 se puede ver la división del vox-sólido en dos partes, el vox-sólido envolvente que aparece remarcado y el vox-sólido externo que aparece en color blanco, este último vox-sólido también resulta ser un vox-sólido toroidal delgado.

Definición 3.4.2. Vox-sólido toroidal completo. Sea $\mathcal{VG}$ un vox-sólido no-delgado de un piso, si $\mathcal{VG}$ está conformado por vox-sólidos envolventes (vox-sólidos toroidales delgados de un piso) entonces diremos que $\mathcal{VG}$ es un vox-sólido no-delgado toroidal completo.

Dividiremos los vox-sólidos no-delgados en dos clases, los que son *toroidales completos* y los que no lo son. Iniciamos trabajando con los primeros y después con el resto.

3.4.1. Vox-sólidos no-delgados de un piso.

Analicemos primero los vox-sólidos no-delgados *toroidales completos* de un piso, en la Figura 3.32 se muestra un vox-sólido no-delgado, en (a) se muestra la vista superior del vox-sólido; en (b) se muestra la parte inferior del vox-sólido; en (c) se muestra el desdoblamiento del vox-sólido si se corta

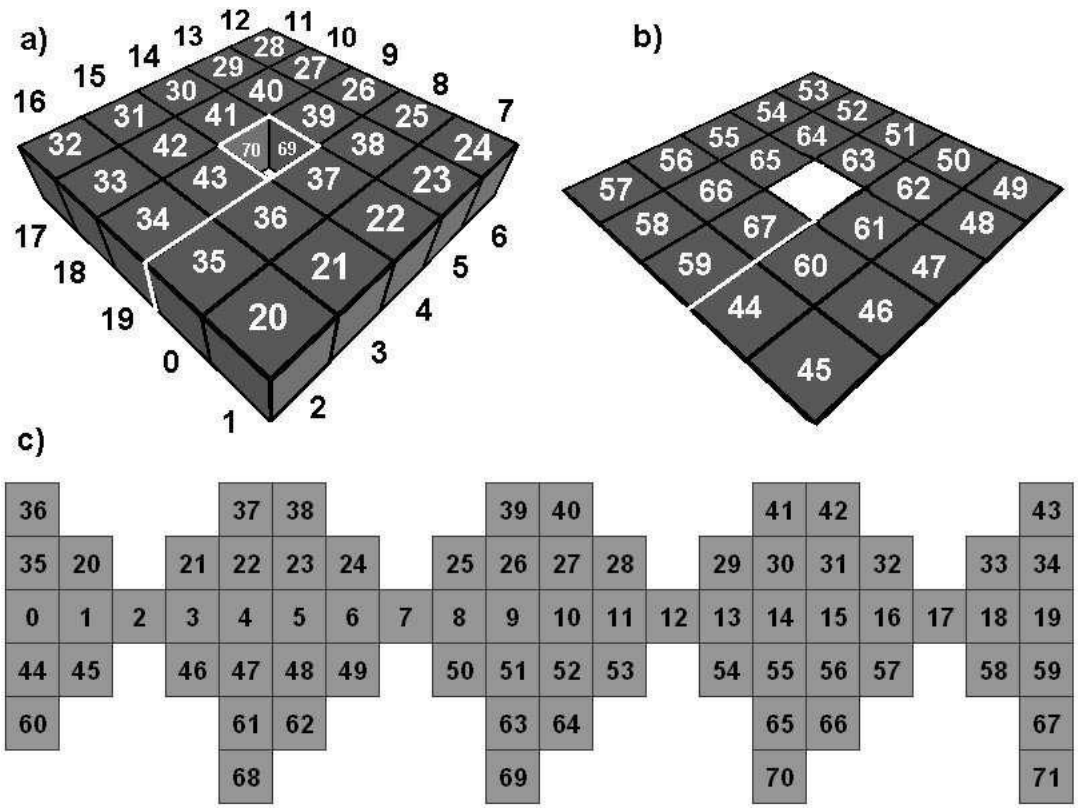


Figura 3.32: Representación planar de un vox-sólido no-delgado.

por las aristas remarcadas, primero se corta entre las caras 34 y 35, después se desdobla el vox-sólido y se corta nuevemente entre las caras 70 y 41 para ser desdoblado nuevamente.

Como se puede ver el desdoblamiento del vox-sólido toroidal no-delgado es muy parecido al del vox-sólido toroidal delgado por lo que intuitivamente podemos esperar que la forma de obtener la representación planar de los vox-sólidos toroidales no-delgados sea muy parecida que la propuesta anteriormente para los vox-sólidos toroidales delgados.

En la Figura 3.31 se puede ver la división del vox-sólido en dos partes, como tenemos dos vox-sólidos toroidales delgados entonces podemos obtener la representación planar a cada uno de ellos y para obtener la representación planar del vox-sólido toroidal no-delgado simplemente unimos las dos representaciones planares de forma conveniente.

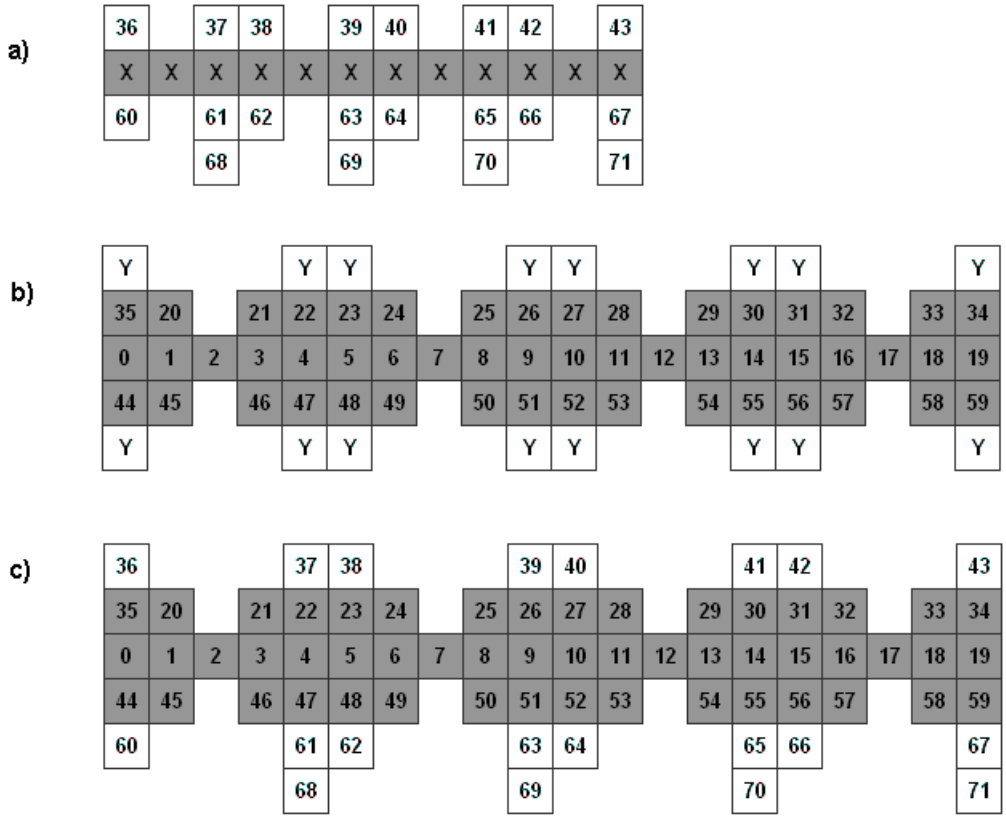


Figura 3.33: Desdoble del vox-sólido toroidal no-delgado.

En la Figura 3.33 (a) se muestra el desdoblamiento del vox-sólido *envolvente* (sus caras están en color blanco), sólo que en la parte donde es adyacente con el vox-sólido *externo* las caras se marcaron con X; en (b) se muestra el desdoblamiento del vox-sólido *externo* (sus caras están en color gris) sólo que en la parte donde es adyacente con el vox-sólido *envolvente* las caras se marcaron como Y; en (c) se muestra la unión de los dos desdoblamientos.

Existen varias formas de hacer la unión conveniente de las representaciones planares, una de ellas es tomar la representación planar del vox-sólido *externo* y en lugar de poner las caras marcadas con Y ponemos las representación planar del vox-sólido *envolvente*; otra forma es tomar la representación planar del vox-sólido *envolvente* y sustituir las caras marcadas con X por la representación planar del vox-sólido *externo*.

Para encontrar las facetas del vox-sólido toroidal no-delgado primero ob-

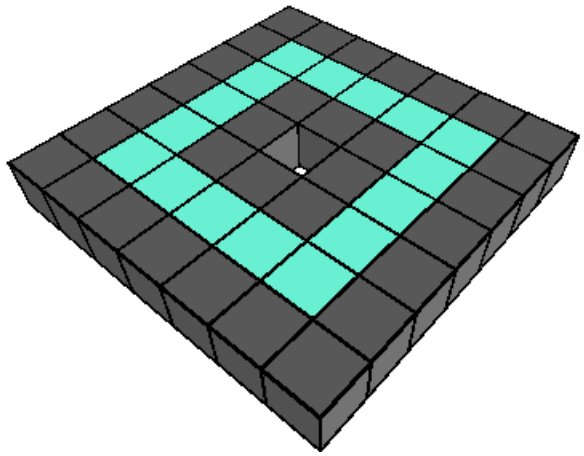


Figura 3.34: Vox-sólido toroidal no-delgado.

tenemos la clasificación, utilizando el método del laberinto, de los vox-sólidos *envolvente* y *externo* por separado y considerando que cuando se analice uno de ellos el otro no existe. Cuando se tengan las facetas de los dos vox-sólidos se procede a poner la facetas del vox-sólido *envolvente* en el orden acostumbrado exceptuando la faceta externa, en su lugar se deben poner las facetas superior, externa e inferior del vox-sólido *externo*.

En el Método 11 se describen los pasos para obtener la representación planar de vox-sólidos toroidales no-delgados completos de un piso. Las funciones $encontrar\_envolvente(vs)$ y $encontrar\_externa(vs)$ determinan los vox-sólidos *envolvente* y *externo*, respectivamente;
$enumera\_caras\_libres(en, ex)$ enumera las caras libres de en y posteriormente (sin reiniciar el contador) las caras libres de ex.

Como se puede ver el proceso de obtención de la representación planar de los vox-sólidos toroidales no-delgados de un piso es casi el mismo que el realizado para los vox-sólidos toroidales delgados de un piso. Sólo nos falta analizar los vox-sólidos que son más gordos que el mostrado en la Figura 3.31; es evidente que el vox-sólido *externo* de un vox-sólido no-delgado puede ser también un vox-sólido no-delgado, en la Figura 3.34 se presenta un ejemplo, del cual también podríamos obtener la *envolvente*.

Para obtener los vox-sólidos envolvente del vox-sólido toroidal no-delgado $\mathcal{VG}$ hay que tomar un voxel que esté rodeando el hoyo, apartir de éste hay

Método 11 Obtener la Representación planar

Precondición: $VG = \{N, L, M\}$ un vox-sólido tipo toroidal no-delgado de un piso.

Postcondición: Representación planar de VG.

```
1: EN := encontrar_envolvente(VG)
2: EX := encontrar_externa(VG)
3: enumera_caras_libres(EN,EX)
4: clasifica_voxeles(EN)
5: clasifica_voxeles(EX)
6: obtener_facetas(EN)
7: obtener_facetas(EX)
8:
9: Alinear horizontalmente las faceta superior de EN; las facetas superior,
   exterior e inferior de EX; las facetas inferior e interior de EN; cada una
   de las facetas en un nivel y alineadas verticalmente de arriba hacia abajo.
10:
11: Agregar a M el primer elemento de la faceta Superior de EN.
12: while M tenga elementos do
13:    e := pop(M)
14:    e.visitado := true
15:    A := adyacentes(e)
16:    for all x en A do
17:       if x.visitado = false then
18:          M.agrega(x)
19:       end if
20:       Trazar una arista de e a x.
21:    end for
22: end while
```

que utilizar el método del laberinto para encontrar el vox-sólido envolvente, para comenzar con el método debemos de colocarnos de forma que la cara que está dando al hoyo nos quede de lado izquierdo y considerar que la cara de la derecha está siempre en la parte externa; de esta manera determinamos la primera envolvente. Para obtener las siguientes hay que repetir estos pasos ignorando la envolvente ya encontrada, informalmente tal envolvente se integra al hoyo topológico, de $\mathcal{VG}$. En el Método 12 se detallan los pasos para encontrar las *envolventes* de un vox-sólido toroidal no-delgado.

Método 12 Obtener las envolventes

Precondición: $VG = \{N, L, M\}$ un vox-sólido tipo toroidal no-delgados de un piso.

Postcondición: Un arreglo con las envolventes de VG.

```
1:
2: envolventes := nulo
3: while No se hayan visitado todos los nodos de VG do
4:    x := un voxel que este rodeando al hoyo considerando los voxeles visitados como parte del hoyo
5:
6:    envolvente := nulo
7:    repeat
8:       envolvente.add(x)
9:       i := voxel adyacente a x del lado izquierdo
10:      f := voxel adyacente a x al frente
11:      if i es no vacio then
12:         y := i
13:      else
14:         y := f
15:      end if
16:      y.visitado := true
17:   until x != y
18:   envolventes.add(envolvente)
19: end while
```

En el Método 13 se muestra la generalización del proceso para encontrar la representación planar para vox-sólidos toroidales no-delgados de un piso, la función $encontrar\_envolventes(vs)$ implemente el Método 12, $ultima\_envolvente(e)$ obtiene la última envolvente del arreglo de envolventes e.

Método 13 Obtener la representación planar

Precondición: $VG = \{N, L, M\}$ un vox-sólido tipo toroidal no-delgados de un piso.

Postcondición: Los representación planar de VG.

1: EN := encontrar_envolventes(VG)
2: enumera_caras_libres(EN)
3: clasifica_voxeles(EN)
4: obtener_facetas(EN)
5:
6: $E := ultima\_envolvente(EN)$
7: Poner en la gráfica planar ordenas horizontalmente las facetas superior, exterior e inferior de E
8: **for** $i := tamano(EN)$ - 1 a 0 **do**
9: $E := EN[i]$
10: Poner sobre la gráfica la faceta superior de E
11: Poner debajo de la gráfica la faceta inferior de E
12: Poner debajo de la gráfica la faceta interior de E, si esta existera
13: **end for**
14:
15: Agregar a M el primer elemento de la faceta Superior de E.
16: **while** M tenga elementos **do**
17: e := pop(M)
18: $e.visitado := true$
19: A := adyacentes(e)
20: **for all** x en A **do**
21: **if** $x.visitado = false$ **then**
22: $M.agrega(x)$
23: **end if**
24: Trazar una arista de e a x.
25: **end for**
26: **end while**

3.4.2. Vox-sólidos no-delgados de más de un piso.

En esta sección trataremos vox-sólidos toroidales no-delgados completos de más de un piso, es decir, cada piso está compuesto por un vox-sólido no-delgado toroidal completo, es decir, cada uno de estos está conformado por vox-sólidos toroidales delgados.

En la Figura 3.35 y Figura 3.36 se muestran vox-sólidos no-delgados de más de un piso, puede verse que los pisos no necesariamente tienen la misma cantidad de envolventes.

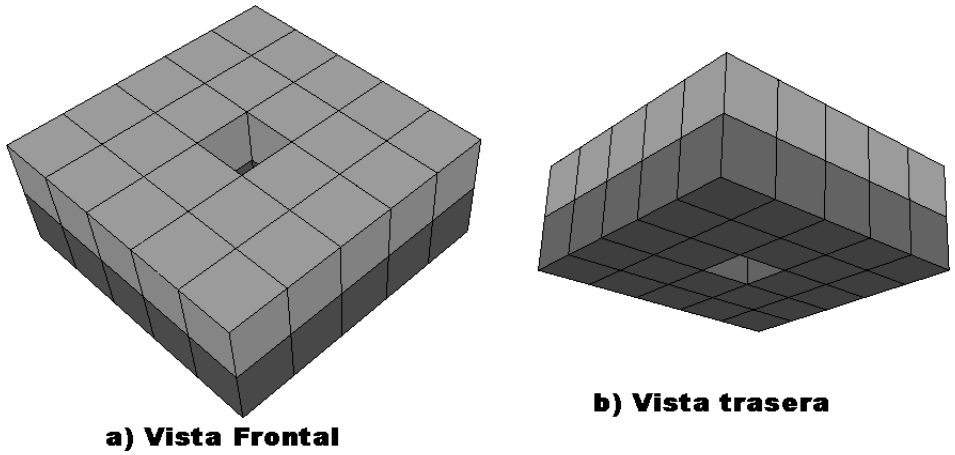


Figura 3.35: Vox-sólido toroidal no-delgado de más de un piso.

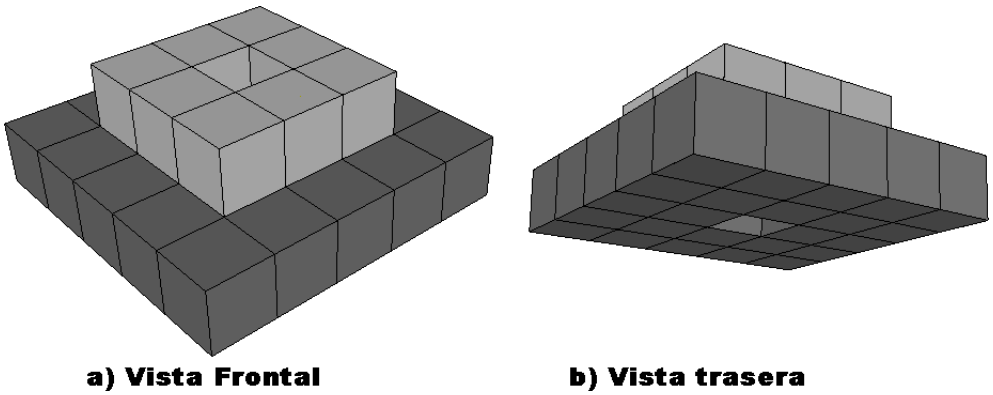


Figura 3.36: Vox-sólido toroidal no-delgado de más de un piso.

Para obtener la representación planar de un vox-sólido no-delgado de más de un piso, primero determinaremos la representación planar de cada uno de los pisos del vox-sólido, los cuales son vox-sólidos toroidales no-delgados completos de un piso, y posteriormente se unen todas las representaciones de forma conveniente para obtener la representación planar del vox-sólido completo.

La forma conveniente de unir las representaciones de los vox-sólidos toroidales no-delgados de un piso es la siguiente, primero se deben obtener todas las envolventes y sus facetas del los vox-sólidos de cada piso; se toma el piso que tenga más envolventes, es decir, el piso que sea más extenso, se toma la envolvente que este más lejos del hoyo y se procede de dos formas:

i) si la envolvente no tiene envolventes en los pisos de arriba y abajo entonces se pone en la gráfica planar la faceta exterior, superior, inferior e interior; al centro, arriba, abajo y mas abajo, respectivamente.

ii) si la envolvente tiene envolventes en los pisos de arriba o abajo entonces se pone en la gráfica la faceta exterior de este piso, las facetas de la envolvente del piso de arriba, las de la envolvente del piso de abajo; al centro, arriba y abajo, respectivamente.

iii) Si algúna de las envonventes tiene una faceta interior, que rodea al hoyo, esta se pone en la parte baja si no hay más pisos de bajo y se pone en la parte mas alta si no hay más pisos arriba de esta.

En la Figura 3.37 (b) se muestra el desdoblamiento del voxel (a), las caras de los voxeles mantienen el color del voxel al cual pertenece. En el Método 14 se muestra el pseudo-codigo para obtener la representación planar de voxeles no-delgados de más de un piso.

Como puede verse el método de la representación planar es lo suficientemente moldeable para adaptarse a los diferentes vox-sólidos que se tienen, tanto delgados como no-delgados e incluso de uno o más pisos.

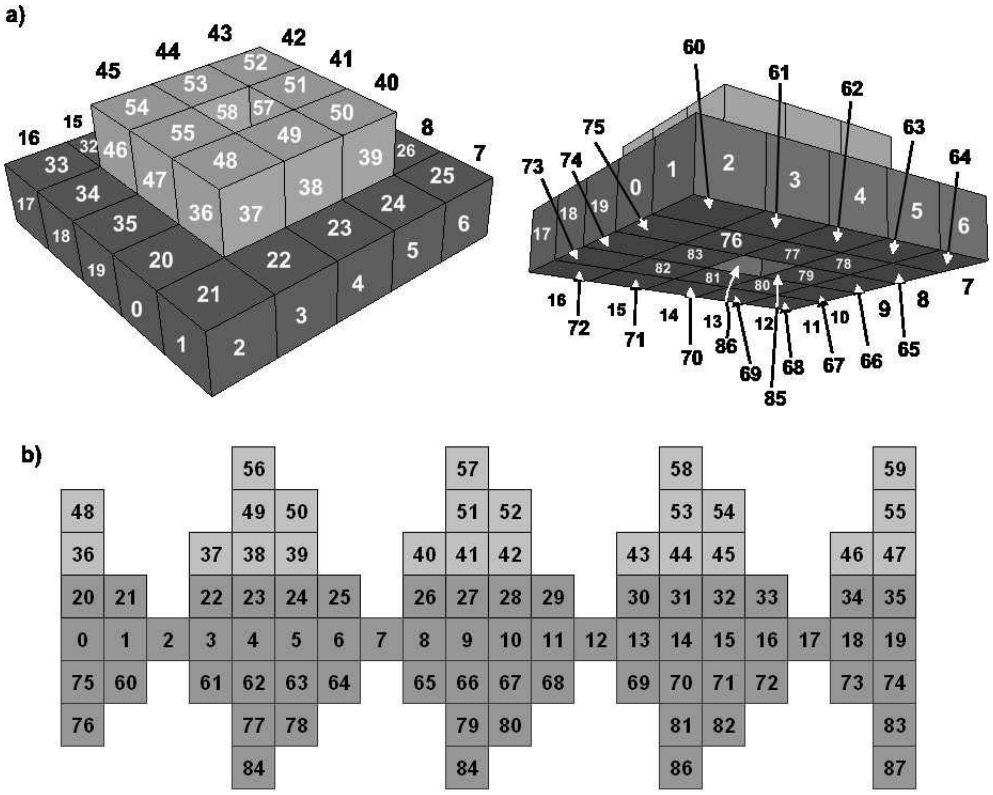


Figura 3.37: Representación planar de un vox-sólido toroidal no-delgado de más de un piso.

Método 14 Obtener la representación planar

Precondición: $VG = \{N, L, M\}$ un vox-sólido tipo toroidal no-delgado de más de un piso.

Postcondición: Los representación planar de VG.

```
1: p := 0
2: max := 0
3: EN := null
4: for p ≤ VG.N do
5:    EN_P := encontrar_envolventes_de_piso(p, VG)
6:    enumera_caras_libres(EN_P)
7:    clasifica_voxeles(EN_P)
8:    obtener_facetas(EN_P)
9:    if max < tamano(EN_P) then
10:      max := p
11:   end if
12:   EN.agregar(EN_P)
13: end for
14: ENM=EN[max]
15: while ENM no este vacia do
16:   E := ultima_envolvente(ENM)
17:   Poner en la gráfica planar ordenas horizontalmente la faceta exterior de E
18:   for i := tamano(EN) - 1 a 0 do
19:      E := EN[i]
20:      Poner sobre la gráfica la faceta superior de E
21:      Poner debajo de la gráfica la faceta inferior de E
22:      if existe faceta interior de E then
23:         Si no hay mas pisos arriba de E, poner la faceta interior arriba de la gráfica
24:         Si no hay mas pisos debajo de E, poner la faceta interior debajo de la gráfica
25:      end if
26:   end for
27: end while
28:
29: Agregar a M el primer elemento de la faceta Superior de E.
30: while M tenga elementos do
31:   e := pop(M)
32:   e.visitado := true
33:   A := adyacentes(e)
34:   for all x en A do
35:      if x.visitado = false then
36:         M.agrega(x)
37:      end if
38:      Trazar una arista de e a x.
39:   end for
40: end while
```

3.5. Otro tipo de vox-sólidos.

En secciones anteriores no se han considerado vox-sólidos con caracteristicas especiales como los llamados *dentados* o que tengan *protuberancias* o los vox-sólidos no-delgados *incompletos*, en está sección enumeraremos las diferencias de estos vox-sólidos especiales y la forma cómo se podría obtener la representación planar de ellos.

3.5.1. Vox-sólidos toroidales delgados

Primero hablemos de los vox-sólidos dentatos que son aquellos vox-sólidos toroidales que en su parte externa o interna tienen alternadamente un voxel o conjunto de voxeles, de tal forma que siga siendo un vox-sólido toroidal delgado. En la Figura 3.38 se muestran variantes del vox-sólido *tuerca* que están dentados, en color claro se muestra la parte dentada. A los voxeles que están en color claro se les llamaremos las *dentaduras* y pueden tener cualquier cantidad de voxeles.

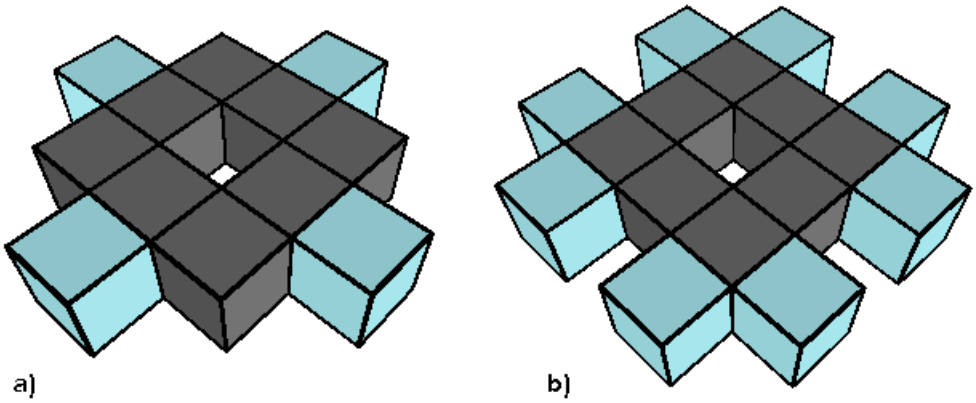


Figura 3.38: Vox-sólidos toroidales delgados dentados.

Estos vox-sólidos presentan una dificultad para encontrar la representación con el método desarrollado, recordemos que la parte importante del método para encontrar la representación planar es clasificar los voxeles para poder especificar las facetas, el método que se utilizó es el de laberinto, la idea principal del método es considerar que el vox-sólido es un laberinto y lo recorremos sin despegar la mano derecha de la pared derecha y además no se puede regresar por un voxel ya visitado.

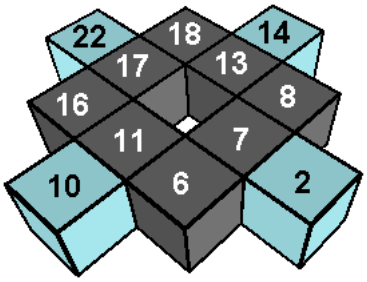


Figura 3.39: Un vox-sólido dentado.

El método del laberinto tiene problemas con los vox-sólidos dentados porque no se puede regresar por un voxel, supongamos que se quiere utilizar este método en el vox-sólido de la Figura 3.39 y entramos al laberinto por el voxel 6, avanzamos de frente hacia el voxel 7, después avanzamos al voxel 2 porque el llevar la mano pegada a la pared derecha nos forza a ir en este sentido, depués ya no podemos avanzar porque entramos a un callejón al no poder regresar por los voxeles que ya visitamos, en este caso el voxel 2 y 7 ya fueron visitados.

Es posible adaptar el método de clasificación de laberinto para funcionar con los voxeles dentados si agregamos una regla que diga que si nos encontramos que no podemos avanzar más entonces podemos dar la vuelta y regresar por donde veniamos sin despegar la mano derecha de la pared, lo que implica que esos voxeles tendrán al menos dos de sus caras del lado derecho del recorrido, esto es posible porque en cualquier dentadura el voxel por el cual se entra es el mismo por el que se sale.

Las dentaduras pueden ser muy largas y estar en cualquier dirección pero no pueden juntarse porque ya no formaría un vox-sólido toroidal delgado o un vox-sólido.

En la Figura 3.40 se muestra un vox-sólido dentado en (a) aparece la vista frontal, mientras que en (b) aparece la vista trasera, los voxeles dentados aparece de color claro; en (c) aparece el desdoble del vox-sólido, nótese que la diferencia con el desdoble del vox-sólido *tuerca* (que aparece en la Figura 3.1) con este vox-sólido es la inserción de las dentaduras en la faceta exterior y entre las facetas superior e inferior.

Para los vox-sólidos toroidales de más de un piso que tengan dentaduras

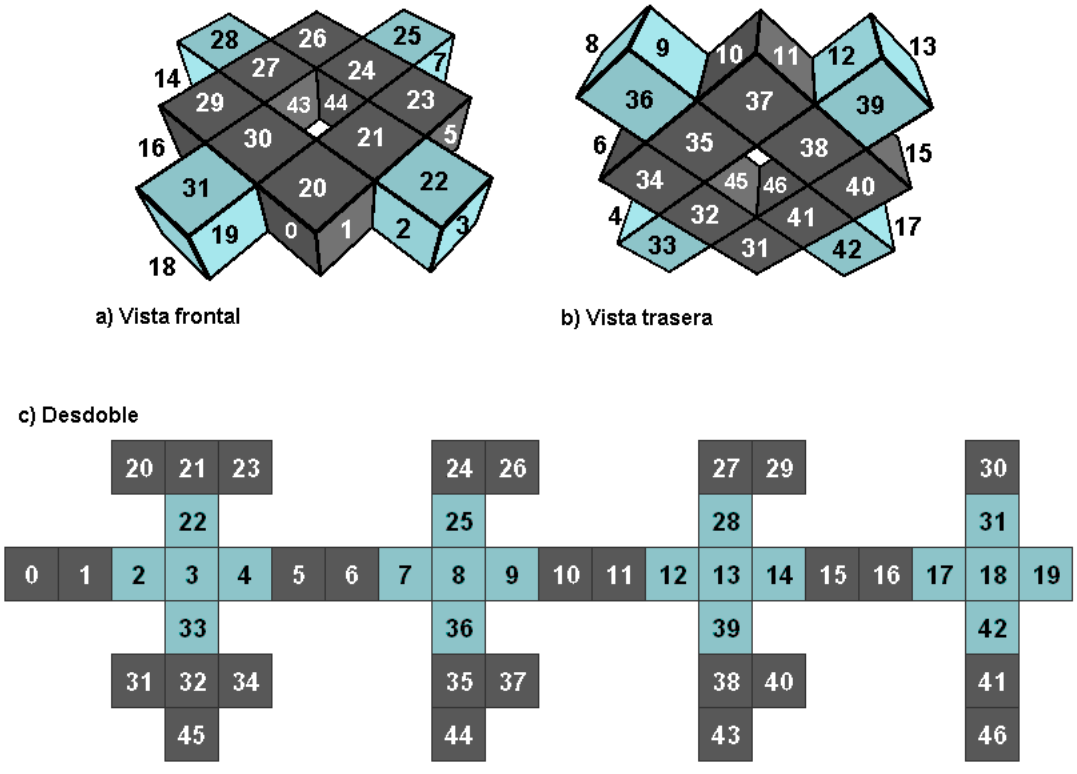


Figura 3.40: Desdoblamiento de un vox-sólido dentado.

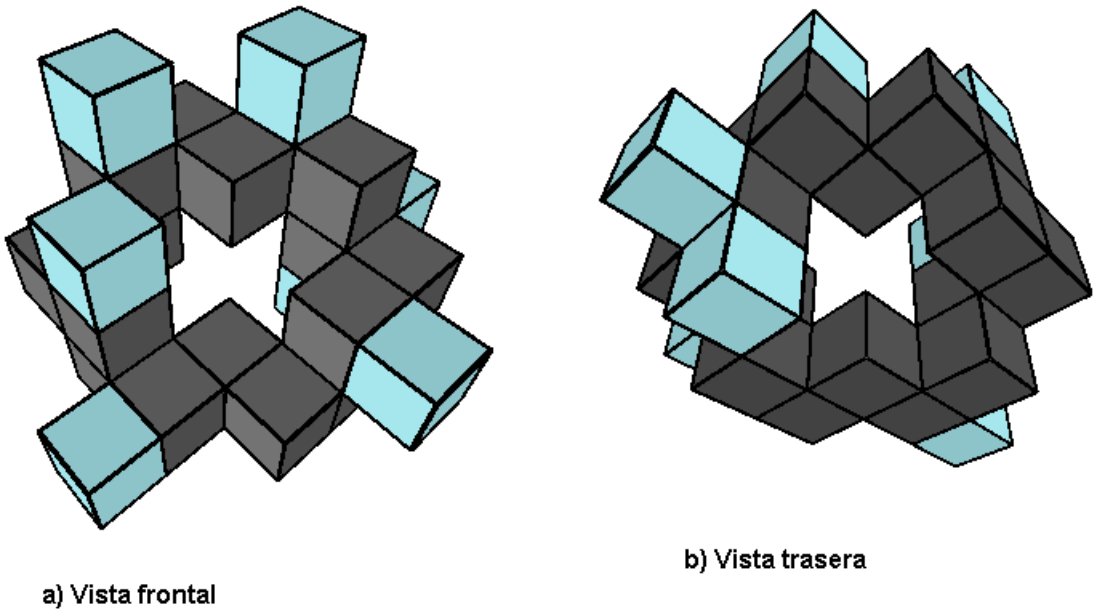


Figura 3.41: Vox-sólido de varios pisos dentado.

en diferentes pisos el proceso sería muy parecido sólo que se tendría que adecuar el recorrido del laberinto para cuando las dentaduras estén en la parte externa, interna, superior o inferior del vox-sólido.

Finalmente en la Figura 3.41 se muestra una alteración del vox-sólido $L16$ que tiene muchas dentaduras en diferentes pisos y direcciones. Las dentaduras aparecen en color claro.

3.5.2. Vox-sólidos toroidales no-delgados

En los vox-sólidos toroidales no-delgados también se presentan las dentaduras pero además existen otros tipos de vox-sólidos como los que son *incompletos* que presentan un problema para encontrar la representación planar.

Para encontrar la representación planar de los vox-sólidos no-delgados completo dentados se utiliza un procedimiento muy parecido al usado en los vox-sólidos toroidales delgados, en términos generales no tiene mayor complicación. En la Figura 3.42 se muestra un vox-sólido no-delgado dentado del

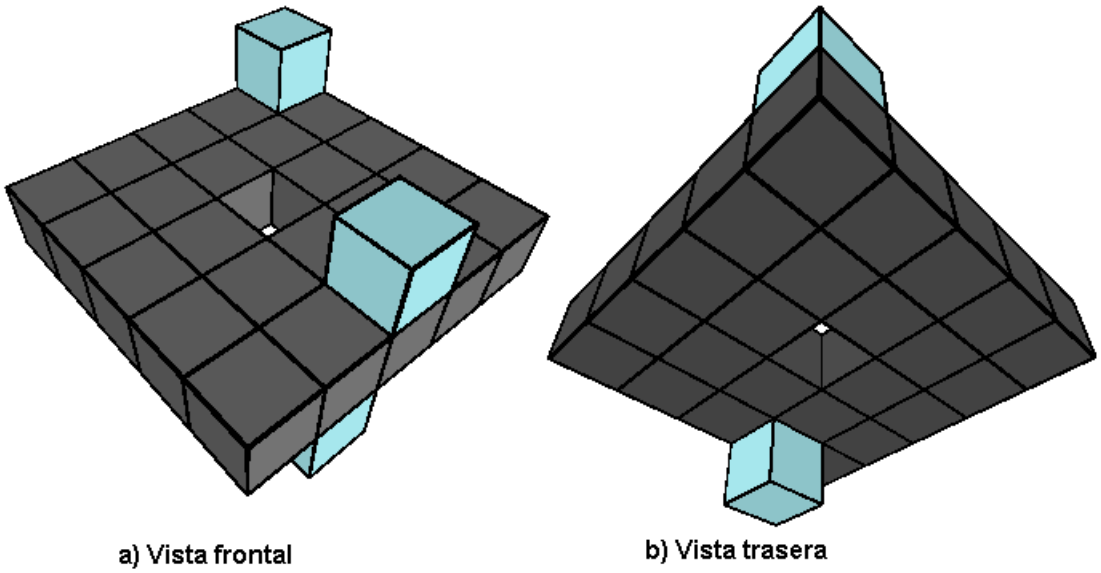


Figura 3.42: Vox-sólido no-delgado completo dentado.

cual se puede ver que su representación planar es relativamente sencilla de encontrar pues simplemente hay que tomar cada una de las envolventes (que son vox-sólidos toroidales delgados dentados) y obtener su representación planar para unirla de forma conveniente.

Llamamos a un vox-sólido incompleto si alguna de sus envolventes son no *completas*. Estos vox-sólidos presentan una complejidad mayor que la vista hasta este momento, en particular presenta un problema para el método de clasificación de laberinto porque existe un momento en el que se puede tener dos caminos a tomar y cualquiera de los dos caminos puede llevarnos a terminar el laberinto (sin pasar por todos los voxeles), esto es problemático porque podría darse el caso que no se clasifiquen todos los voxeles lo que implicaría que no se puedan encontrar de forma correcta las facetas.

En la Figura 3.43 se muestra un vox-sólido toroidal no-delgado que es incompleto en su envolvente externa, la cual se muestra de color claro. Supongamos que se intenta obtener la representación planar de este vox-sólido entonces dividimos el vox-sólido en sus envolventes, la primer envolvente (que está más al centro) es un vox-sólido toroidal delgado completo, por lo que no hay problemas para encontrar su representación planar; el vox-sólido envolvente más externo es incompleto, he aquí donde se encuantran los problemas

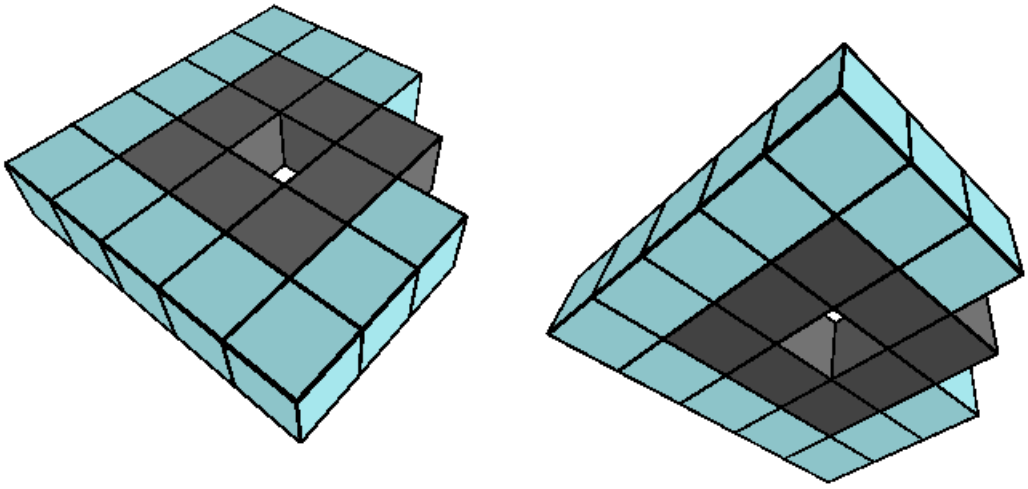


Figura 3.43: Vox-sólido no-delgado incompleto.

porque no terminaríamos el laberinto, nuevamente aquí hay que realizar una adaptación al método del laberinto que consistiría en agregar una nueva regla, que diga si ya no se puede avanzar en el laberinto entonces se da vuelta y se regrese por donde se venía, sin separar la mano derecha de la pared derecha, esto garantiza que se termine el laberinto.

Con la adaptación del método del laberinto es posible encontrar la representación planar de este tipo de vox-sólidos, nuevamente se pueden tener combinaciones de vox-sólidos no-delgados incompletos y dentados además de ser de varios pisos.

Como se puede ver el método del laberinto es sumamente adaptable para facilitar el encontrar la representación planar de la mayoría de los vox-sólidos toroidales de uno o varios pisos incluyendo sus alteraciones y deformaciones.

Conclusiones

En el presente trabajo se describió una representación de vox-sólidos toroidales y métodos para encontrar su representación planar.

Si bien existen otras forma de representar objetos en $3D$, algunas de éstas se describen en el Apéndice A, con la representación propuesta se simplifica la verificación de adyacencias tanto de voxeles como de sus caras y reduce el espacio utilizado para representar los vox-sólidos.

Con la ayuda de la representación de vox-sólidos se propuso un método que permite identificar de forma clara las facetas del vox-sólido, es decir, cuales caras conforman la parte externa, interna, superior e inferior. Al conocer las facetas se puede encontrar la representación planar.

Se realizó la implementación del método para encontrar la representación planar de vox-sólidos toroidales, en el lenguaje de programación Python. Debido a que el tratamiento de los vox-sólidos no siempre es intuitivo se generó una aplicación que muestra paso a paso el método para encontrar la representación planar.

Si bien se logra obtener la representación planar se cree que el método aún puede simplificarse por lo que se propone como trabajo futuro lo siguiente:

- **Representación planar**. Al parecer no debería ser necesario encontra una trayectoria hamiltoniana para recorrer el vox-sólido pues la clasificación de voxeles de laberinto puede realizar el descubrimiento de la trayectoria conforme se va recorriendo el laberinto.

- **Crecimiento hacia arriba**. Si el método se utiliza con un vox-sólido toroidal como el de la Figura 3.44, el método no puede encontrar la representación planar porque el método del laberinto no considera que el vox-sólido se tengan dos voxeles seguidos de subida o bajada, esto es sencillo de corregir, simplemente hay que agregar otra clasificación

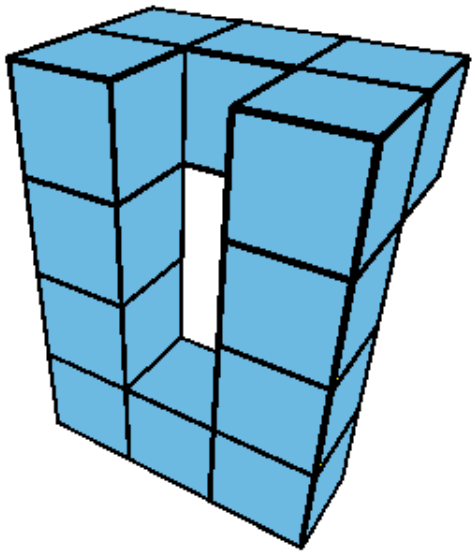


Figura 3.44: vox-sólidos con crecimiento en mas de un voxel hacia arriba.

para estos voxeles.

- **Mejorar operaciones de vox-sólidos**. La representación propuesta para representar los vox-sólidos está enfocada en encontrar la representación planar por lo que no se trabajaron completamente las operaciones que se pueden realizar con esta representación. Por lo tanto, se deberá de mejorar y extender la forma de realizar las operaciones.

- **Vox-sólidos No-delgados**. En el presente trabajo se trabaja con vox-sólidos delgados y se realizó una implementación para obtener su representación planar, también se propone el método para obtener la representación planar de los vox-sólidos no-delgados pero no se realizó una implementación, debe extenderse la implementación para este tipo de vox-sólidos.

- **Impresión de la representación planar**. En la implementación de la representación planar, cuando se dibuja su gráfica, en algunos casos, se muestran cruzadas las aristas, esto se debe corregir haciendo la alineación vertical de los nodos.

Parte I

Apéndices

Apéndice A

Otras Representaciones

En la actualidad existen diferentes representaciones o codificaciones de los vox-sólidos, las cuales están enfocadas a diferentes objetivos, algunas se orientan a la visualización de objetos en $3D$ y otras a preservar las características topológicas o gemétricas. En la presente sección se describirán brevemente las más populares.

A.1. Representación basada en objetos $3D$

Con el incremento en el uso de programas CAD, de sistemas *Realidad Virtual* y de la *Resonancia Magnética Computalizada* ha surgido la necesidad de representar y manipular, en sistemas de cómputo, objetos en $3D$, debido a esto se han desarrollado diversas representaciones que permiten guardar las características de color y forma de los objetos, así como su posicionamiento en un ambiente de $3D$. Estas representaciones son diversas y muy variadas, en el presente trabajo enumeraremos las más utilizadas.

A.1.1. Octree

La estructura Octree es la extesión de los *quadtree*. Aunque la primera definición formal de octree fue publicada en 1980 por Jackins y Tanimoto ya se habia tenido trabajo previo de Hunter y Steiglitz en 1979, quienes habian trabajado con los quadtree en tres dimensiones [8].

Un octree es una estructura jerárquica arbórea que corresponde a la subdivisión del espacio euclidiano en octantes, los cuales puede tener tantos niveles de resolución como se desee, a cada octante de la división se le llama *celda*.

Cada nodo en el árbol representa un cubo en $\mathbb{R}^3$. Cada nodo hijo representa una subdivisión del cubo padre en otro octante.

En el árbol existen diferentes tipos de nodos [6]:

- *Raíz.* La raíz del árbol.
- *Nodo interno.* Un nodo que es padre de otros nodos.
- *Nodos hojas, ocho celdas.* Es un nodo que representa una subdivisión de este volumen padre en otro octeto.
- *Nodos hojas, una celda.* Es un nodo que representa una sóla celda.

En la Figura A.1 se muestra un ejemplo de la formación de un Octree.

Los principales usos de los octree son: la representación de volumenes, detección de colisión, la seleccion de una parte del modelo de un mundo virtual, el cual será mostrado en la pantalla (*viewing frustum*) y para mostrar la relación espacial entre objetos geométricos. Existen diferentes adaptaciones de los octree dependiendo de aplicación en la que se use, también existen diferentes métodos para almacenar la estructura que están enfocadas en eficientar el acceso a los datos.

Las dos principales operaciones con los octree son: *localización de un punto* y *encontar vecinos*, la primera se refiere a encontrar un punto en la estructura y regresar los hijos que contiene que es relativamente fácil; mientras que la segunda se refiere a encontrar los vecinos de una celda, lo cual es un poco más difícil.

Dentro de las diferentes variedades de Octree encontramos los siguentes: *Full*, *Linear* y *Branch-on-need (BONO)* que se utilizan para *volume rendering.*

La información que se almacena en cada uno de los nodos es la que se necesite en la aplicación, por ejemplo, para el trabajo de esta tesis se almacena W (white) si el segmento no contiene parte del objeto, es decir, si es un voxel blanco y B (Black) si el segmento contiene parte del objeto, es decir, si es un voxel negro, esta representación nos sirve mucho para encontrar la representación planar porque nos permite encontrar de forma sencilla las adyacencias a cada voxel. En la Figura A.2 (tomada de [1]) se muestra la representación octree de un objeto con estas características, el cubo de la derecha muestra la numeración de los octetos.

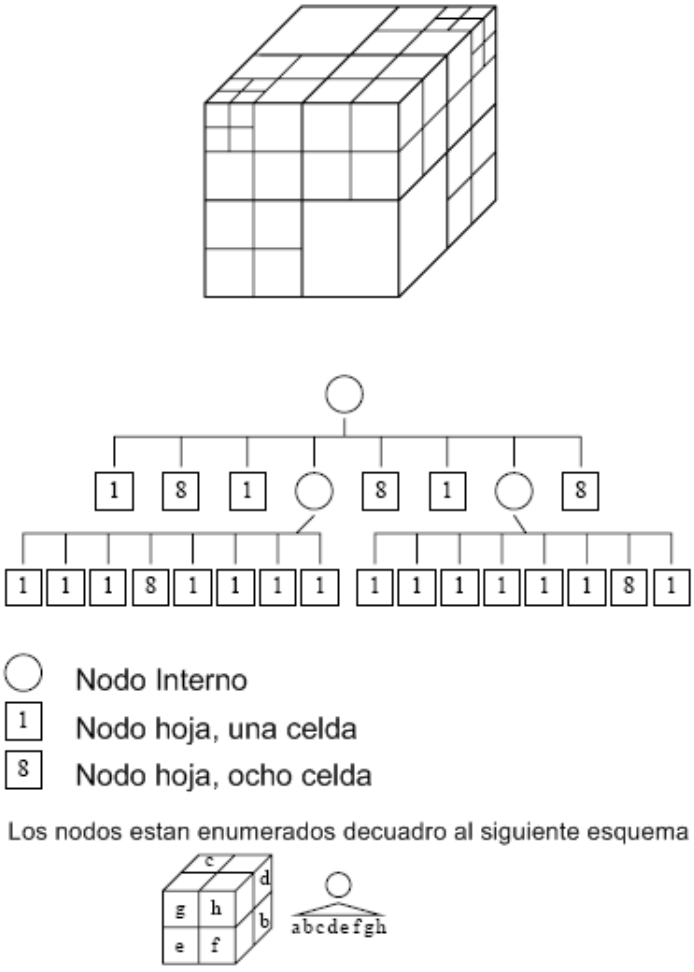


Figura A.1: Contrucción típica de un Octree.

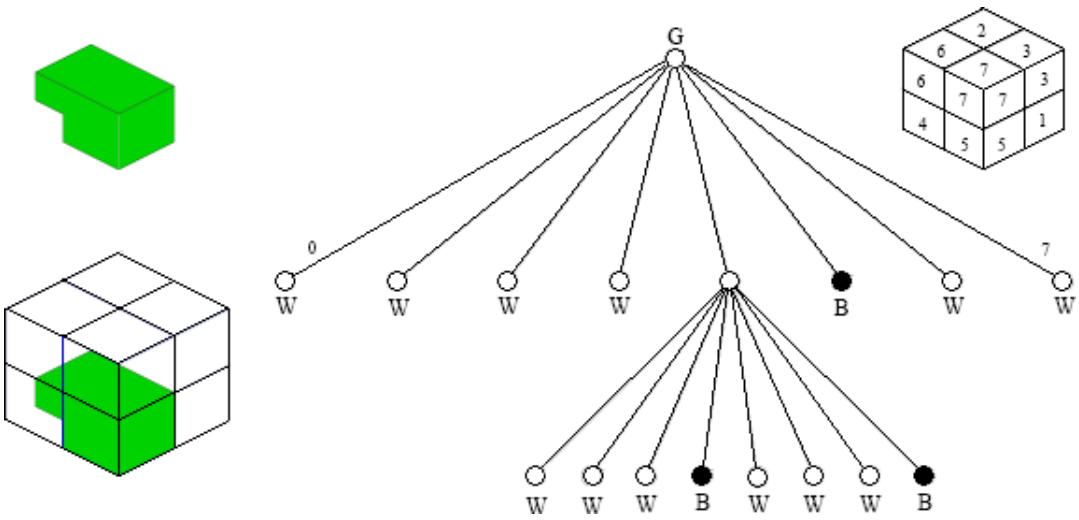


Figura A.2: Contrucción de un Octree.

A.1.2. BSP-tree

La estructura *árbol de particiones espaciales binarias* BSP-tree[1] realiza una partición jerárquica y por niveles sucesivos del espacio en semiespacios utilizando planos de orientación arbitraria. Estos planos se organizan en un BSP-tree de forma que cada nodo representa un plano, y sus dos hijos son los dos semiespacios que lo define. Las hojas del árbol tienen dos valores *interior* o *exterior* que indican si cada región, definida por los sucesivos cortes, está en el interior o el exterior del objeto [7].

En la Figura A.3 se muestra un ejemplo de un BSP-tree para un objeto en $2D$.

Los árboles BSP son usados en varias aplicaciones, especialmente en la graficación por computadora ya que tiene muchas mejoras en cuanto a la eficiencia de los cálculos geométricos tales como: ocultamiento de superficies en algoritmos para dibujar objetos; generación de sombras; operaciones booleanas con poliedros; visualización interactiva según la posición del observador; planeación de movimiento de robots; entre otros.

Dentro de las principales operaciones en un BSP-tree se encuentran la clasificación, la pertenencia de un punto del espacio en el objeto definido,

[1]Por sus siglas en ingles *Binary Space Partition Tree*

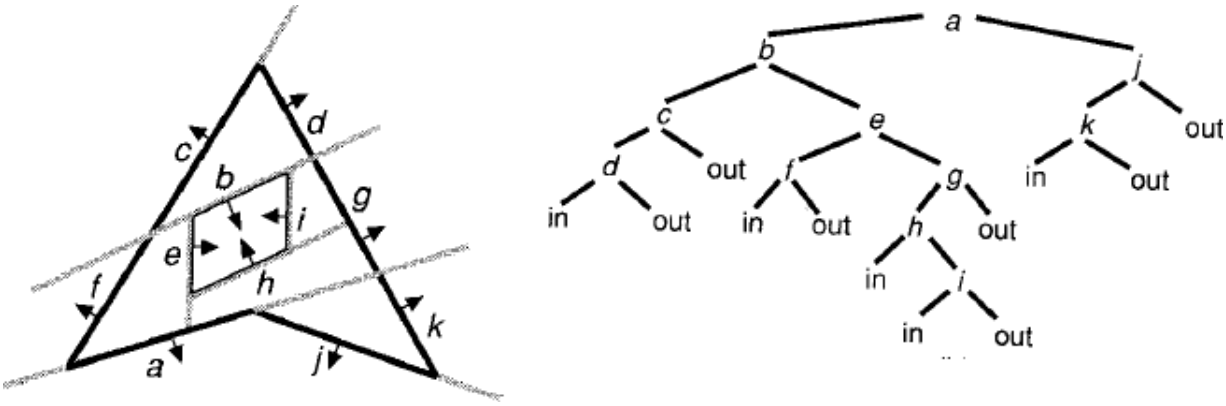


Figura A.3: Representación de una figura en $2D$ con BSP-tree.

esto se hace recorriendo todo el árbol desde la raíz, la decisión de recorrer el árbol por la rama derecha o izquierda se toma conforme a la menor distancia entre el nodo actual y el punto buscado, este procedimiento se sigue hasta llegar a una hoja.

Para cada figura existen diferente representaciones BSP-tree debido a que se pueden utilizar diferentes planos para partir el espacio, en términos generales, se recomienda utilizar la partición que obtenga una estructura más corta para eficientar su tratamiento. Sin embargo en este tipo de representación el cálculo de las vecindades es relativamente complejo.

Existe otra representación de objetos que se llama *kD-tree* que son muy parecidos a los *BSP-tree*, sólo que las particiones se realizan ortogonalmente a los ejes x, y o z [7].

A.2. Representación basada en características topológicas

Los modelos que representan objetos basándose en sus características geométricas o topológicas tienen como principal objetivo la manipulación de los objetos representados, por lo cual es de gran importancia saber exactemente como están formados y que características presentan.

Muchos de estos modelos de representación se utilizan para hacer la voxelización de imágenes. La voxelización está definida como el proceso de

convertir la representación geométrica de un objeto en un conjunto de voxeles.

A.2.1. B-Rep

La representación de bordes *B-Rep*[2] es un modelo que almacena la información de los objetos basándose en su geometría y muchas veces en la topología Euleriana. La topología de Euler representa los vértices, aristas, half-edge, face loop, caras, cubiertas (shell) y los cuerpos, en general se refiere a los principales componentes de los objetos, aunque éstos no son fácilmente identificables. La representación geométrica son los puntos, líneas, superficies (por ejemplo: planos, esferas, conos, toros), sólidos, etcétera.

La representación B-Rep es utilizada por la mayoráa de los sistemas *CAD* tanto para el almacenamiento de sus propios modelos como para hacer intercambio de información con otros sistemas. Una de las principales características de B-Rep es que las operaciones aplicadas al modelo satisfacen la ecuación de Euler [10].

La extensa utilización de esta representación hizo que se generara un estándar de representación llamado STEP y está definido bajo *ISO 10303* el cual se introdujo en los años noventa para reemplezar al anterior llamado IGES.

Aunque este modelo de representación es muy utilizado es difícil relizar la voxelización de objetos debido a que el interior de los objetos no está explícitamente identificado. En la actualidad no existen métodos eficientes que realicen esta transformación, pero con los avances en la paralelización, la velocidad del hardware y el incremento en el poder de cómputo es posible realizar la voxelización casi en tiempo real [11].

Existen otros formatos llamados GWB (Geometric Work Bench) y ACIS's SAT que también se basan en la topología Euleriana.

[2]Por la contracción en ingles de *boundary representation*

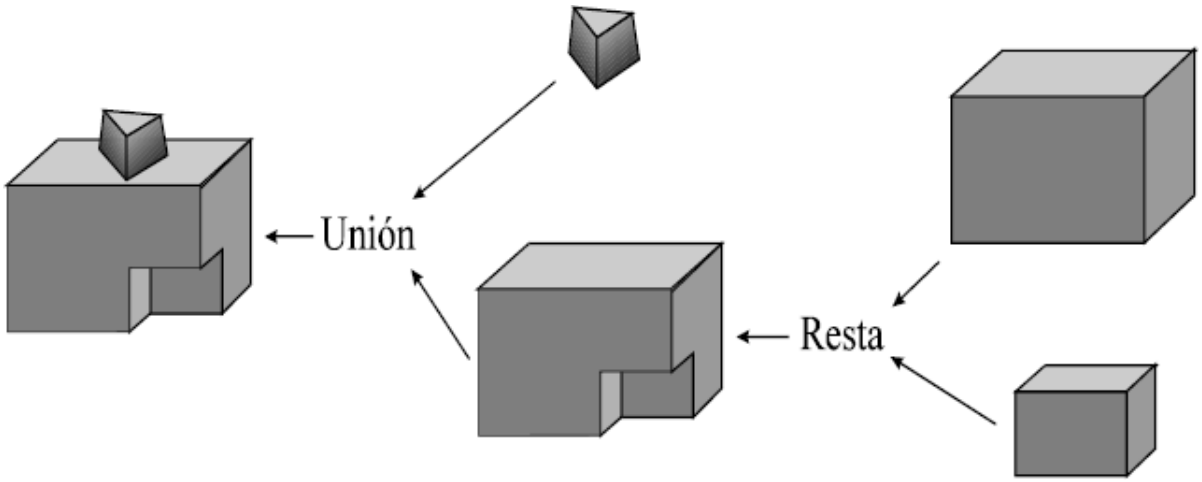


Figura A.4: Contrucción de un objeto mediante *CSG*.

A.2.2. CSG

En la Geometría Sólida Constructiva CSG[3] un sólido es representado como una combinación de primitivas o bloques de construcción, tales como rectángulos, conos, esferas, toros, prismas, entre otros. Las primitivas están combinadas mediante operaciones booleanas lógicas: unión, intersección, resta y complemento; también existen operaciones unarias que realizan la transformación de coordenadas tales como: rotación y traslación.

Los objetos en CSG se representan con un árbol donde se guardan los objetos primitivos y las operaciones que se tienen que realizar para formar el objeto. En la Figura A.4 aparece un ejemplo de como se representa un objeto usando CSG. En la figura se muestra el árbol de representación, la raíz esta a la izquierda y las hojas están a la derecha.

En el árbol CSG se incluyen distintos tipos de nodos :

- **Primitivas**. Son las formas básicas de la representación y las hojas del árbol, en ´éstas se puede almacenar la información de rotaciones o traslaciones.

- **Operaciones**. Describe las operaciones booleanas que combinan los objetos.

[3]Por sus siglas en ingles *Constructive Solid Geometry*

El modelo CSG representa eficientemente objetos que se pueden formar con combiaciones de primitivas básicas sencillas, sin embargo es ineficiente para representar objetos que no se pueden formar por primitivas simples. Las primitivas están definidas por la aplicación pero en general se utilizan las formas geométricas básicas.

La mayoría de los sistemas CAD utilizan esta representación para almacenar la información de los objetos. Para esta representación exiten una amplia variedad de algoritmos que permiten su manipulación de forma eficiente así como su visualización.

A.3. Comparación de Representaciones

Como se ha mostrado existen varias formas de representar objetos en $3D$ que son eficientes y tienen muchas características deseables que permiten trabajar con vox-sólidos complejos.

En el presente trabajo se utilizan vox-sólidos toroidales y para su representación se necesitan tres cosas: poder identificar las adyacencias de los voxeles que los conforman, identificar claramente todas las caras de los voxeles para saber quien esta en cada una de las facetas del vox-sólidos y que la representación utilice el mínimo espacio de almacenamiento ya que los vox--sólidos toroidales más complejos pueden estar formados por una cantidad grande de voxeles.

Si utilizamos alguna de las representaciones descritas en esta sección se necesitaría realizar más trabajo para poder obtener todas las características deseadas, por eso en el capítulo 2 se propone otra forma de representar los vox-sólidos.

Para que sea más clara la diferencia entre utilizar una u otra representación se muestra un ejemplo de como se codificaría el vox-sólidos al que llamamos *tuerca* con el método Octree y con la codificación CSG.

A.3.0.1. *Tuerca* con la estructura Octree

En esta representación se subdivide el espacio en octantes y se construye una estructura jerárquica del espacio, dicho de otra forma, el espacio de subdivididos en ocho cubos, hasta que se tengan objetos indivisibles que en nuestro caso serán los voxeles.

En la Figura A.5 se muestra la forma en que se realiza la división del espacio donde se encuentra el vox-sólido para obtener la división de los octetos. En (a) se muestra la primera división del espacio en octetos, nótese que los voxeles B, C y E quedan en el mismo cubo, al igual que G, H y A, D, pero F queda sólo en un cubo; como todavia hay más de un voxel en un cubo entonces se subdividen nuevamente, pero sólo aquellos cubos que tienen más de un voxel; en (b) se muestra la subdivisión de los cubos que tienen sólo un voxel en cada cubo. En la figura se ven sombreados los cubos que se están dividiendo, las líneas gruesas muestran la división anterior, las líneas delgadas muestran la división actual.

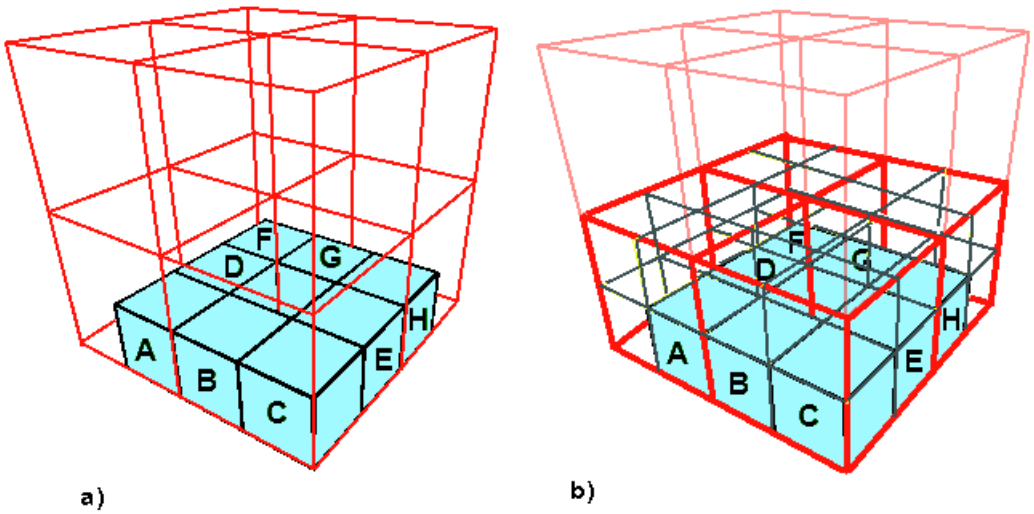


Figura A.5: División de la tuerca para la estructura *Octree*.

En la Figura A.6 se muestra el árbol que se genera con la codificación de la tuerca con la estructura Octree; recordemos que los nodos blancos forman el fondo de la imagen mientras que los nodos negros son los voxeles que forman la tuerca; la enumeración que se utiliza para codificarlos es la que se

muestra en la Figura A.2. En las figuras se nombró cada voxel con una letra para que sea más fácil identificarlos.

Ya que tenemos la representación de la tuerca obtengamos las características deseadas: encontrar las adyacencias y saber cuáles son las caras que lo componenen para obtener las facetas que nos ayudarán a encontrar la representación planar.

- **Encontrar adyacencias.** Esto es relativamente fácil, por ejemplo, es evidente que el voxel G y H son adyacentes porque las dos voxeles son hijos del mismo padre; para ver que E y H (que son hijos de diferentes padres) son adyacentes, basta ver que los nodos en el posición 1 y 5, de cada rama, pueden ser adyacentes, en este caso resulta que sí son adyacentes.

- **Caras que lo componen.** Esta parte es más compleja porque no se almacenan los datos de las caras, por lo cual habría que inferir cuáles son las caras del vox-sólido.

- **Espacio utilizado.** En esta representación se utiliza 40 nodos para representar la tuerca, en nuestra representación sólo se utilizan 8 elementos para guardar la enumeración estandar y 16 elementos para la matriz de adyacencias.

A.3.0.2. *Tuerca* con la estructura CSG

En esta representación se generan los objetos através de primitivas y la mínima unidad con la que nos interesa trabajar son las caras, entonces definimos como primitivas a las caras del voxel.

En al Figura A.7 se muestra la construcción de un voxel através de sus caras; en el inciso *a)* (en la parte superior izquierda) aparecen las caras en que se divide el voxel; en el inciso *b)*, aparece la construcción del árbol CSG para un voxel utilizando las caras. En la Figura A.8 se muestra la contrucción de la tuerca a través de voxeles, entendiéndose que los voxeles están construidos por árboles CSG como el de la figura A.7.

Ya que tenemos la representación de la tuerca obtengamos las características deseadas: encontrar las adyacencias y saber cuáles son las caras que lo componenen para obtener las facetas que nos ayudarán a encontrar la repre-

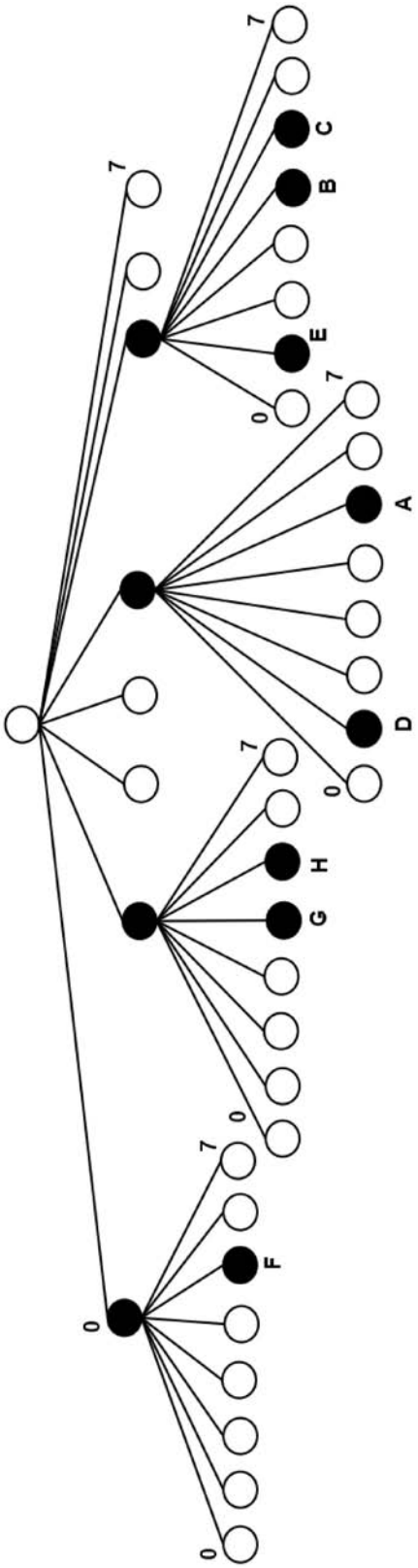


Figura A.6: Codificación de la tuerca con la estructura *Octree*.

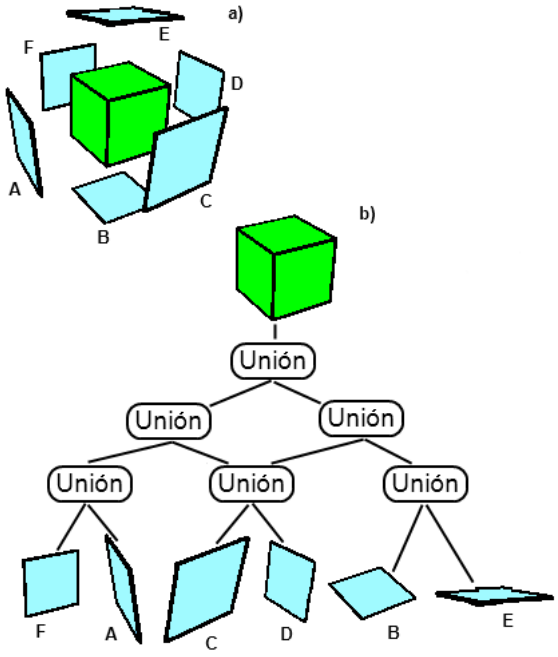


Figura A.7: Codificación de voxeles para la estructura *CSG*.

sentación planar.

- **Encontrar adyacencias.** Esto es relativamente fácil, sólo hay que verificar, en el árbol, las posiciones de las caras de cada voxel, si dos voxeles tienen el mismo padre entonces son adyacentes; si los voxeles no son hermanos entonces hay que verificar si sus caras son adyacentes. Por ejemplo, es evidente que el voxel A y B son adyacentes, para ver que A y E no son adyacentes habría que verificar la adyacencia de todas sus caras.

- **Caras que lo componen.** Esta parte es más sencilla porque se tiene como primitiva a las caras, para saber cuáles son las caras adyacentes habría que verificar que las caras compartan una arista.

- **Espacio utilizado.** En esta representación se utiliza 12 nodos para representar cada uno de los 8 voxeles, más 7 nodos para unir los voxeles que forman el vox-sólido, en resumen se usan $(12*8)+7 = 103$ nodos para representar la tuerca, en nuestra representación sólo se utilizan 8 elementos para guardar la enumeración estandar y 16 elementos para la matriz de adyacencias.

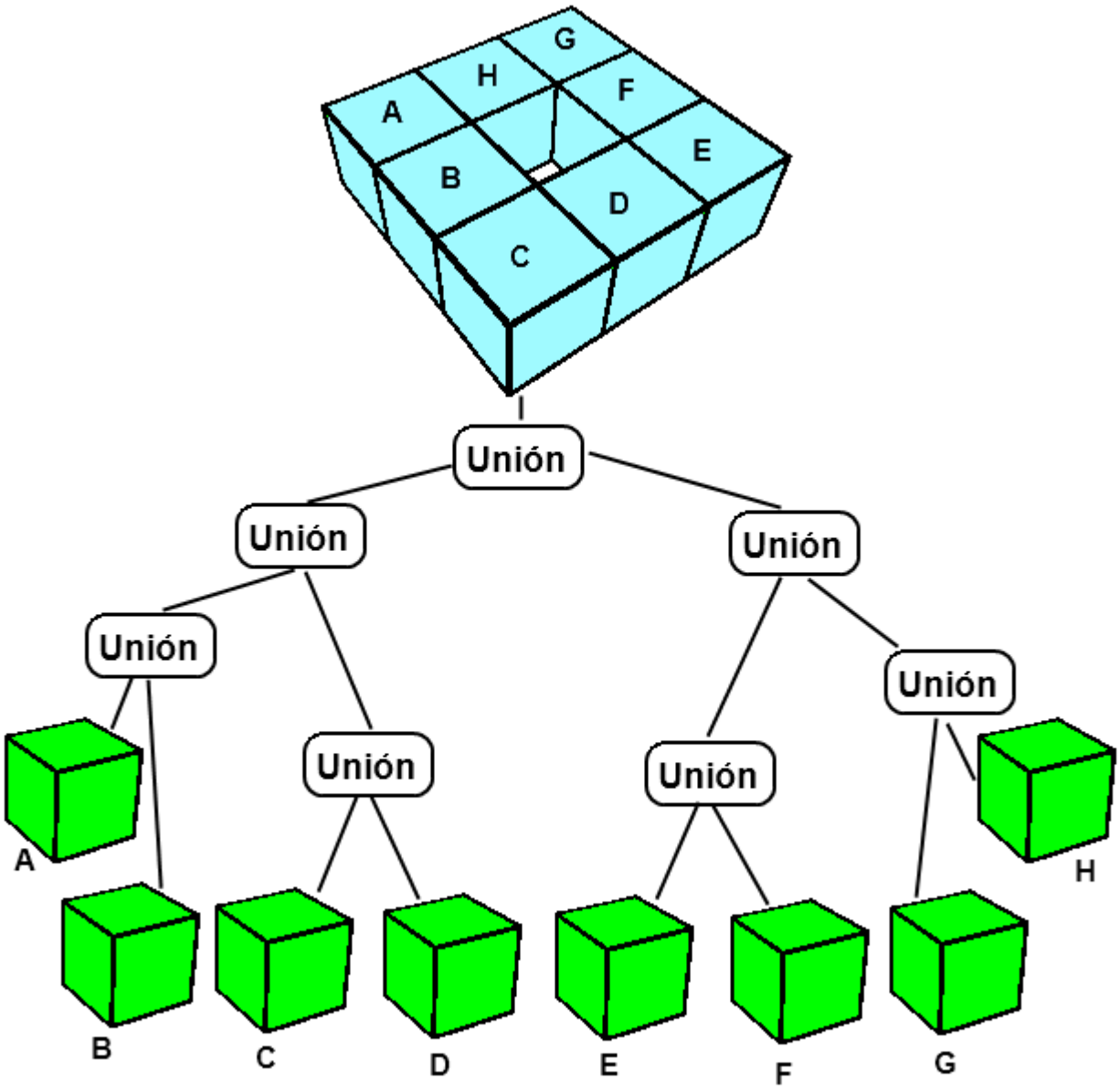


Figura A.8: Codificación de la tuerca con la estructura *CSG*.

En conclusión, utilizar alguna de las estructuras mencionadas en esta sección habría incrementado el tamaño de almacenamiento de los vox-sólidos y se tendría un incremento de las operaciones para obtener las adyacencias de voxeles y de las caras.

La estructura planteada en esta tesis, en el capítulo 2, es simple, utiliza menos espacio para su almacenamiento y permite encontrar de forma sencilla las adyacencias; por lo anterior concluimos que para la obtención de la representación planar de vox-sólidos toroidales la representación propuesta en este trabajo es útil.

En el programa realizado en el presente trabajo se implemento la estructura propuesta y el método para obtener la representación planar, el programa tardo 31.125[4] segundos para obteber la representación planar del vox-sólido $\mathcal{V}_{32}$, que se muestra en la Figura 3.28.

[4]corrida realizada en una computadora con un procesador de 2GHz y 1GB de RAM

Apéndice B

Implementación

Debido a que el trabajo con vox-sólidos no es enteramente intuitivo y en algunos casos es difícil de visualizar ciertos conceptos, se realizó la implementación del método, con fines didácticos para obtener la representación planar para vox-sólidos toroidales delgados sin dentaduras. La idea principal es que el programa pareciera una película donde se va mostrando paso a paso el método.

Se utilizó el lenguaje de programación *Python* [9] debido a que es un lenguaje que facilita hacer prototipos de forma rápida, es portable y posee una gran cantidad de bibliotecas para su uso [4], ente ellas bibliotecas para el manejo de ambientes en $3D$, de ellas se eligió *vPython* [12] por su manejo simple de los objetos en $3D$.

El sistema está programado con la orientación a objetos y se tienen las siguientes clases:

- **textN.py** Tipografia para escribir numeros en la biblioteca *vpython*.
- **voxel.py** Provee la visualización en $3D$ de un voxel.
- **voxsolido.py** Provee la implementación de los vox-sólido, sabe calcular la trayectoria y las facetas.
- **repPlanar.py** Hace el cálculo de la representación planar.
- **animacion.py** Permite ejecutar paso a paso la construcción de la representación planar.
- **voxsolidoDatos.py** Tiene los datos del vox-sólido que se utiliza en animacion.py.

A continuación se explica en términos generales la lógica y las principales características de cada uno de las clases, más adelante se muestra el código completo del programa que está liberado bajo la licencia *GLP2* [3].

voxel.py. Implementa la graficación de un voxel en $3D$. La idea principal de esta clase es que nos permita visualizar un voxel en $3D$. Cada voxel está conformado por el *esqueleto*, que son todas las aristas; las *caras*, la seis caras del voxel y los *textos* que se escriben en cada cara.

Las operaciones que se pueden realizar con el voxel son: mover a una posición específica en el espacio; ocultar o mostrar cada una de sus seis caras así como el esqueleto y el texto; se puede cambiar el texto que se muestra en cada una de las caras; se puede cambiar el color de cada una de las caras.

En cuanto a la implementación de esta clase, se tiene un arreglo de tamaño seis para almacenar las caras, otro arreglo para los vértices y otro arreglo para los textos. Cuando se crea un voxel se crea con sus seis caras, el equeleto y los tex tos, durante la vida del voxel se pueden ocultar o mostrar todo los componentes. Esta clase utiliza la clase *text* que se localiza el el archivo *textN.py* y permite escribir textos en $3D$, sólo permite escribir números.

voxsolido.py. Implementa la representación de un vox-sólido. Esta clase es la principal porque provee el lógica de la representación del vox-sólido, esta clase crea los voxel necesarios para que se dibuje en $3D$ el vox-sólido.

Esta clase recibe como entrada la representación matricial de un vox-sólido, las operaciones que realiza son calcular la matriz de adyacencias, sabe calcular las adyacencias de cada uno de los voxeles y de cada una de las caras del voxel; calcular una ruta sobre el vox-sólido; obtener la etiquetación de los voxeles según el método del laberinto; además calcula las facetas.

La idea de la implementación es tener un arreglo donde se guarda la representación matricial del vox-sólido y otro arreglo donde se tiene la matriz de adyacencias. Utilizando estos dos arreglas se tiene funciones que hacen los demás cálculos.

repPlanar.py. Tiene la implementación del método de la representación planar. Esta clase saber hacer los calculos para encontrar la gráfica de la representación planar paso a paso y dibujar la gráfica planar en $2D$.

Esta clase recibe como entrada un vox-sólido. Las principales operaciones

que realiza son dibujar los nodos y vértices de la representación planar, así como realizar la contrucción de la representación planar paso a paso. Existe una opción para eliminar la representación gráfica y simplemente obtener la representación planar en la lista de vértices.

La idea principal de la representación es tener un arreglo de nodos y vértices que una vez dibujados en la pantalla nos permite remarcarlos conforme vaya avanzando el proceso de encontrar la representación planar.

animacion.py. En este archivo se tiene la función principal que permite ejecutar todo el proceso paso a paso. Aquí se crean los controles para la animación y se permite controlarla.

En este archivo se crean los objetos antes mencionados y las funciones que relizan el control de la animación de la representacion planar, finalmente es el que se encarga de sincronizar la ejecución de la demás clases y funciones.

voxolidoDatos.py. Tiene los datos del vox-sólido que se utiliza en la animación, mas aún este archivo es incluido en el archivo animacion.py.

B.1. Ejecución del programa

En la Figura B.1 se muestra el inicio de la ejecución del programa; en la Figura B.2 se muestra la ejecución después de algunos pasos en la animación, los voxeles remarcados en blancos son los que ya se procesaron y se obtuvo de ellos las aristas de la gráfica planar; en la Figura B.3 se muestra el termino de la ejecución del programa.

El programa puede ejecutarse en forma interactiva, que son las pantallas descritas arriba, o puede ejecutarse en modo no-visual que hace el proceso sin mostrar ventanas y genera un archivo de salida que tiene la lista de las aristas para generar la representación planar.

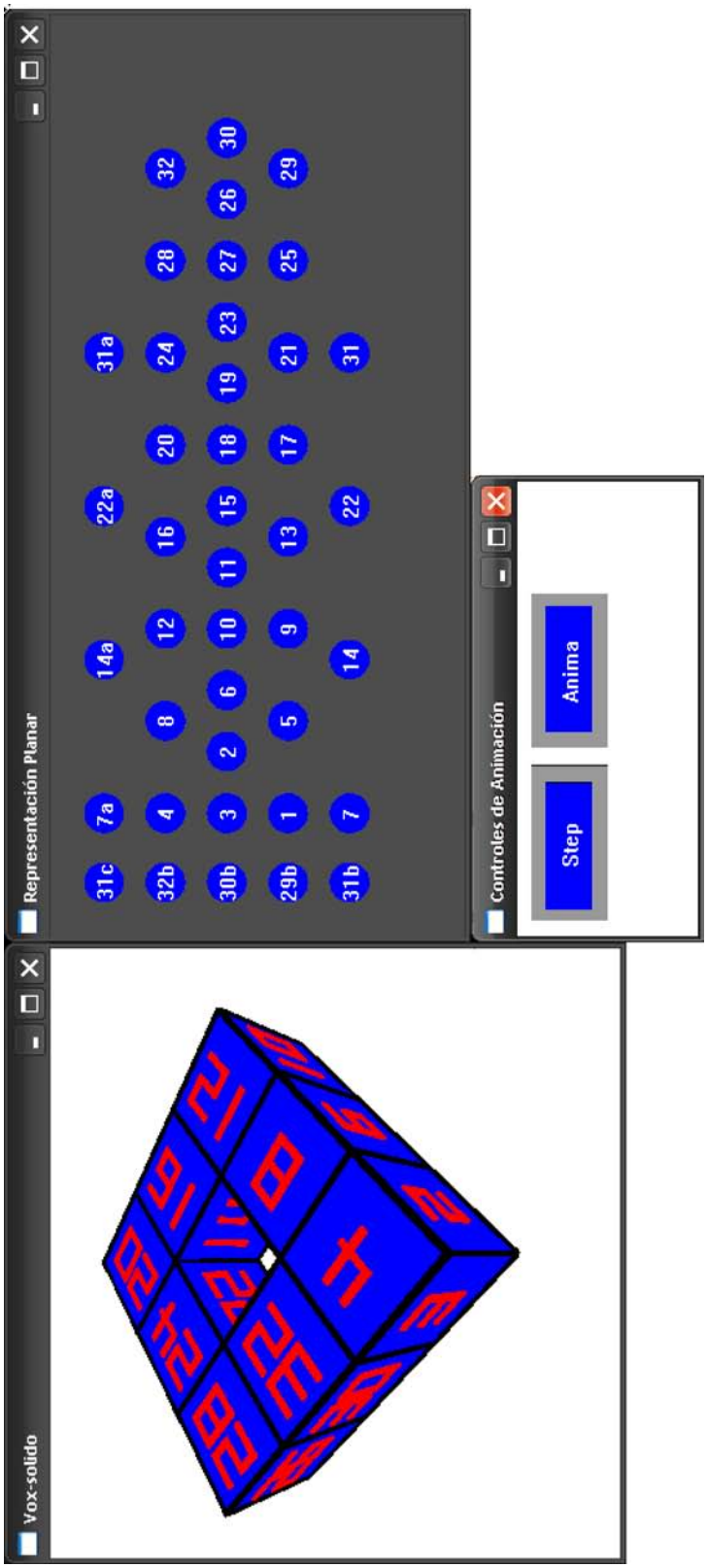


Figura B.1: Inicio de la ejecución.

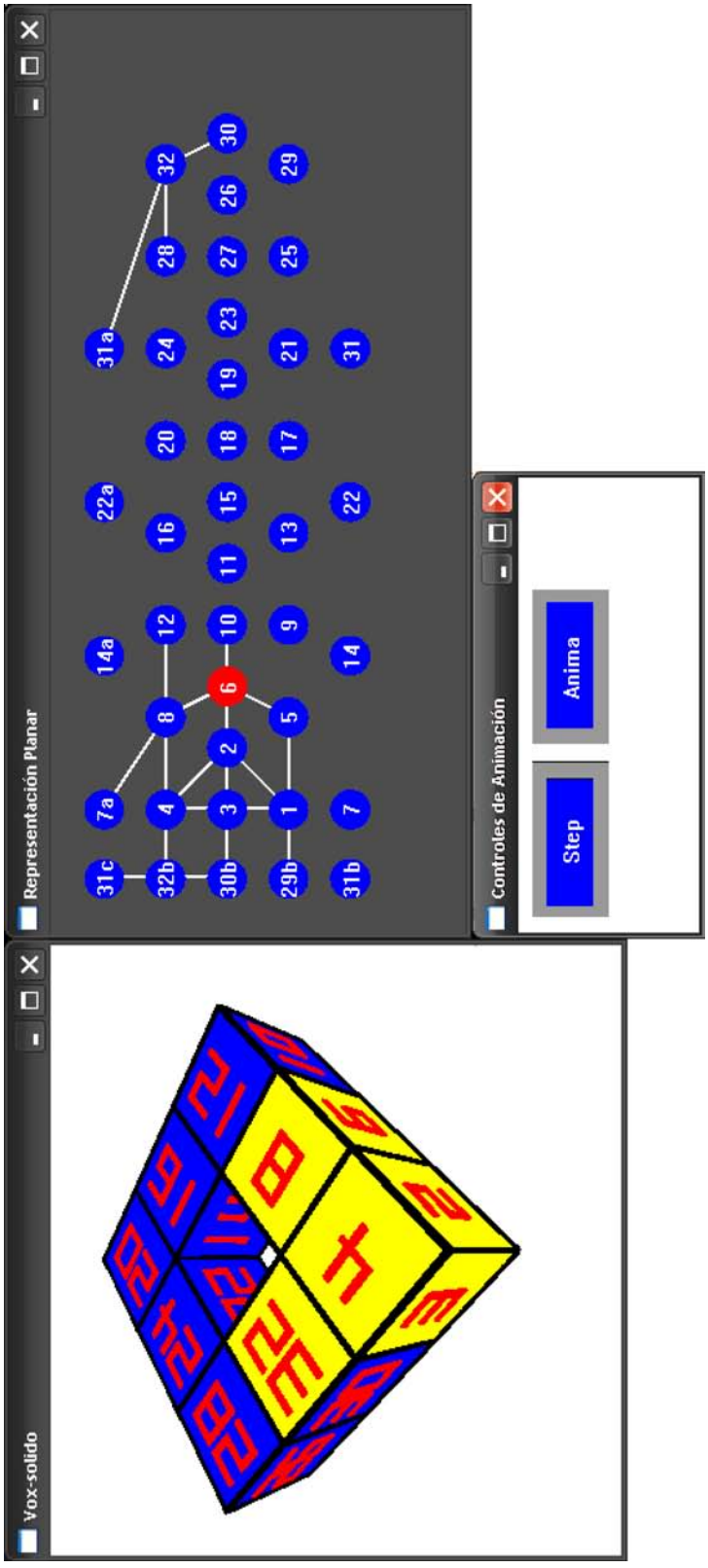


Figura B.2: Parte intermedia de la ejecución.

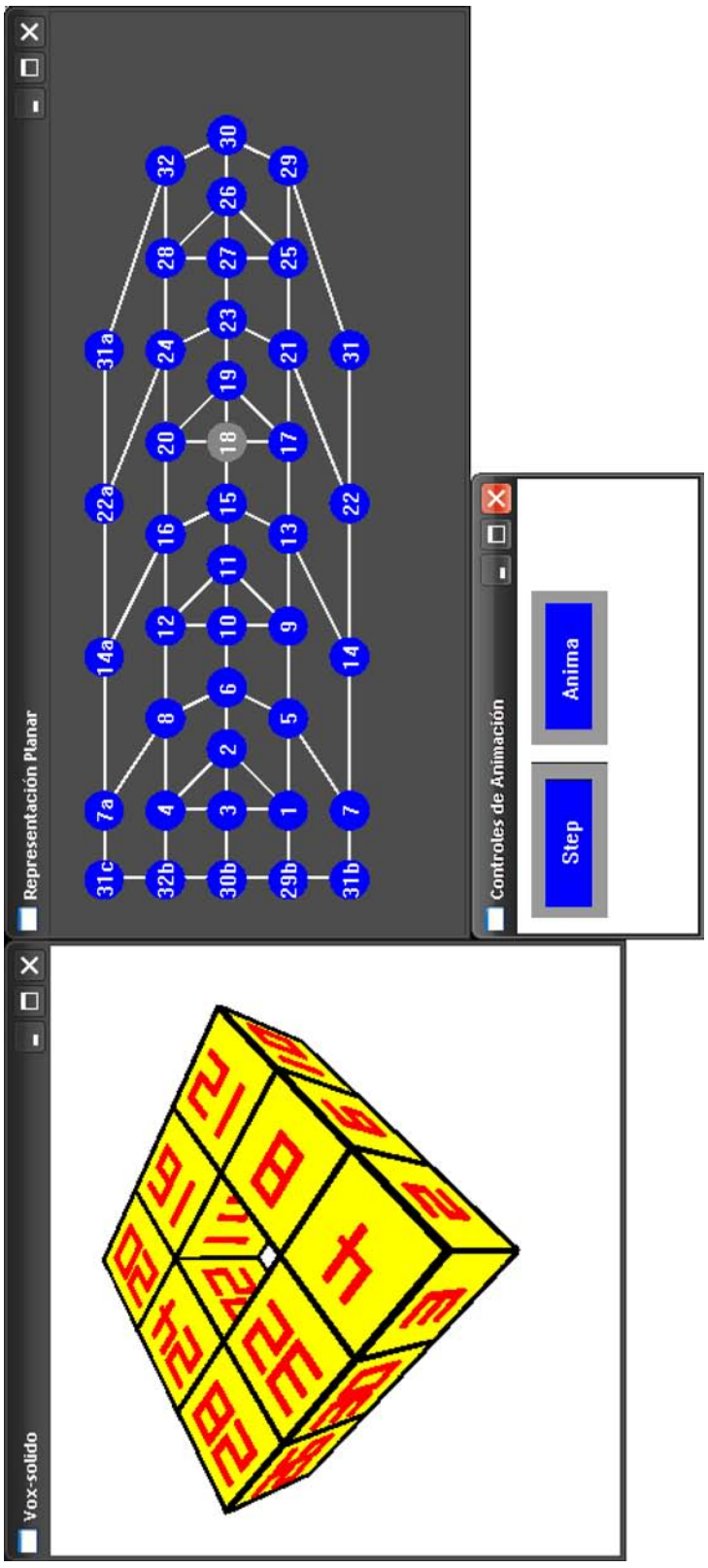


Figura B.3: Fin de la ejecución.

B.2. Código de la implementación

textN.py

```
#-----------------------------------------------------------
# Name:        textN.py
# Purpose:     Modificación de la implementación de Téxtos a sólo números.
#
# Author:      Eliot Peña Rojas
#
# Created:     2006/05/01
# RCS-ID:      $Id: textN.py $
# Copyright:   (c) 2006
# Licence:     GLP 2.0
# Version:     1.0
#-----------------------------------------------------------
#
# Modificado para que solo pueda escribir numeros y en ligar de '*' pone '-'
#

from visual import *


# Display extruded text (uppercase only at present)
# By default, display text along x axis, with letters pointing up parallel to y axis
# Bruce Sherwood, Carnegie Mellon University, begun March 2000

# Example with default values:
#  text(pos=(0,0,0), axis=(1,0,0), string='ABC',
#     height=1, depth=0, width=1,
#     color=currentdisplay.foreground, up=(0,1,0.3))
# axis is direction along which text advances
# if width not specified, it is the same as height
# depth is measured forward from pos
# Only numbers and uppercase letters at present: others display as '*'

defaultdir = vector(1.0,0.,0.)
defaultup = vector(0.,1.0,0.)

letters = {' ': [(0.39,0,0)],
           # los números en formato simple
           '0': [[(0,0.0,0), (0.13,0,0), (0,1,0)],
               [(0.,1.0,0), (0.60,0,0), (0,-0.13,0)],
               [(0.0,0,0), (0.60,0,0), (0,0.13,0)],
               [(0.53,0,0), (0,1,0), (0.13,0,0)],
               (0.78,0,0)],
           '1': [[(0.22,0,0), (0.13,0,0), (0,1.0,0)],
               (0.78,0,0)],
```

```
        '2': [[(0,0,0), (0.66,0,0), (0,0.12,0)],
            [(0,0,0), (0,.53,0), (0.13,0,0)],
            [(0.53,.5,0), (0,.5,0), (0.13,0,0)],
            [(0,1,0), (0.66,0,0), (0,−0.12,0)],
            [(0,.43,0), (0.66,0,0), (0,0.12,0)],
            (0.78,0,0)],
        '3': [[(0,0,0), (0.66,0,0), (0,0.12,0)],
            [(0.53,0,0), (0,1,0), (0.13,0,0)],
            [(0,1,0), (0.66,0,0), (0,−0.12,0)],
            [(0,.43,0), (0.66,0,0), (0,0.12,0)],
            (0.78,0,0)],
        '4': [[(0.42,0,0), (0.13,0,0), (0,1.0,0)],
            [(0,0.34,0), (0.69,0,0), (0,0.10,0)],
            [(0,0.34,0), (0,.65,0), (0.13,0,0)],
            (0.78,0,0)],
        '5': [[(0,0,0), (0.66,0,0), (0,0.12,0)],
            [(0.53,0,0), (0,.53,0), (0.13,0,0)],
            [(0,.5,0), (0,.5,0), (0.13,0,0)],
            [(0,1,0), (0.66,0,0), (0,−0.12,0)],
            [(0,.43,0), (0.66,0,0), (0,0.12,0)],
            (0.78,0,0)],
        '6': [[(0,0,0), (0.66,0,0), (0,0.12,0)],
            [(0,0,0), (0,1,0), (0.13,0,0)],
            [(0.53,0,0), (0,.5,0), (0.13,0,0)],
            [(0,1,0), (0.66,0,0), (0,−0.12,0)],
            [(0,.43,0), (0.66,0,0), (0,0.12,0)],
            (0.78,0,0)],
        '7': [[(0,1.0,0), (0.66,0,0), (0,−0.12,0)],
            [(.52,0,0), (0,1,0), (0.13,0,0)],
            (0.78,0,0)],
        '8': [[(0,0,0), (0.66,0,0), (0,0.12,0)],
            [(0,0,0), (0,1,0), (0.13,0,0)],
            [(0.53,0,0), (0,1,0), (0.13,0,0)],
            [(0,1,0), (0.66,0,0), (0,−0.12,0)],
            [(0,.43,0), (0.66,0,0), (0,0.12,0)],
            (0.78,0,0)],
        '9': [[(0,0,0), (0.66,−.01,0), (0,0.12,0)],
            [(0.53,0,0), (0,1,0), (0.13,0,0)],
            [(0,.5,0), (0,.5,0), (0.13,0,0)],
            [(0,1,0), (0.66,0,0), (0,−0.12,0)],
            [(0,.43,0), (0.66,0,0), (0,0.12,0)],
            (0.78,0,0)],
        '-': [[(0.01,0.34,0), (0,0.13,0), (0.34,0,0)],
            (0.44,0,0)],
            }

def getlength(str):
    dx = 0
    for char in str:
```

```
        if letters.has_key(char):
            data = letters[char]
        else:
            data = letters['-']
        # assume that increment in char description is always of form (dx,0,0)
        dx = dx+data[-1][0]
    return dx

class text:
    def __init__(self, pos=(0,0,0), axis=defaultdir, string='', justify='left',
            height=1.0, width=None, depth=0, color=color.white, up=None, visible=1,
            display=None):
        if display == None:
            display = scene
        self.display = display
        axis = norm(vector(axis))
        self.pos = vector(pos)
        height = float(height)
        if width == None:
            width = height
        width = float(width)
        depth = float(depth)
        container = frame(pos=pos, axis=axis)
        self.string = string
        self.justify = justify
        self.height = height
        self.width = width
        self.depth = depth
        self.color = color
        container.color = color
        container.visible = visible
        if up <> None:
            container.up = up
        self.frame = container
        self.objects = []
        if justify == 'right':
            origin = vector(-width*getlength(string),0.,0.)
        elif justify == 'center':
            origin = vector(-width*getlength(string)/2.,0.,0.)
        else:
            origin = vector(0.,0.,0.)
        for char in string:
            origin = self.showletter(origin, char, width, height,
                                     depth, color, container, visible)

    # agregada por eliot
    def change_text(self, string=None):
        if string <> None:
            del self.string
```

```
        del self.objects
        self.objects=[]
        if self.justify == 'right':
            origin = vector(-self.width*getlength(string),0.,0.)
        elif self.justify == 'center':
            origin = vector(-self.width*getlength(string)/2.,0.,0.)
        else:
            origin = vector(0.,0.,0.)                                    150

        # cuando el número tiene tres digitos esta es la corrección
        if len(string) > 2:
            self.width=0.35
            origin = vector(-self.width*getlength(string)/2.5,0.,0.)

        for char in string:
            origin = self.showletter(origin, char, self.width, self.height,
                                     self.depth, self.color, self.frame, self.frame.visible)
                                                                         160
def makeinvisible(self):
    for obj in self.objects:
        obj.visible = 0

def makevisible(self):
    for obj in self.objects:
        obj.visible = 1


def makeletterbox(self, origin, b, xsize, ysize,                        170
                  thickness, color, container, visible):
    barray = array(b)
    ab = barray*array((xsize,ysize,0.))
    org = origin+ab[0]
    b = convex(display=self.display, color=color, frame=container, visible=visible)
    self.objects.append(b)
    if thickness <> 0:
        for i in range(2):
            for j in range(2):
                for k in range(2):                                       180
                    b.append(pos=org + i*ab[1] + j*ab[2] + \
                                     k*vector(0.,0.,xsize*thickness))
    else:
        for i in range(2):
            for j in range(2):
                    b.append(pos=org + i*ab[1] + j*ab[2])

def showletter(self, origin, char, xsize, ysize,
               thickness, color, container, visible):
    if letters.has_key(char):                                            190
        data = letters[char]
```

```
        else:
            data = letters['-']
        for n in range(len(data)-1):
            self.makeletterbox(origin, data[n], xsize, ysize,
                               thickness, color, container, visible)
        # assume that increment in char description is always of form (dx,0,0)
        dx = data[-1][0]
        return origin+xsize*dx*defaultdir

    def reshape(self, pos=None, height=None, width=None, color=None):
        if pos is not None:
            self.frame.pos = pos
        if height is None:
            height = self.height
        if width is None:
            width = self.width
        if color is None:
            color = self.color
        xratio = width/self.width
        yratio = height/self.height
        #print array((xratio,yratio,0.)) # comentado por eliot
        for obj in self.objects:
            obj.pos = obj.pos*array((xratio,yratio,1.))
            obj.color = color
        self.height = float(height)
        self.width = float(width)
        self.color = color

if __name__ == '__main__':
    scene.title = "3D Text"
    scene.fov = 0.001
    scene.range = 7
    message = text(pos=(0,0,0), string='0123456789', justify='center',
                   color=color.yellow, depth=0.3)
```

voxel.py

```
#-----------------------------------------------------
# Name:        voxel.py
# Purpose:     La implementación de la representacion gr"sfica de un Voxel
#
# Author:      Eliot Peña Rojas
#
# Created:     2006/05/01
# RCS-ID:      $Id: voxel.py $
# Copyright:   (c) 2006
# Licence:     GLP 2.0
# Version:     1.0
#-----------------------------------------------------


"""Implementa la representación gr"sfica de un Voxel

El voxel esta dividido en seis caras y a cada una de ellas se puede:
* Mostrar el esqueleto (aristas)
* Mostrar la cara completa
* Mostrar un Texto (numerico)
* Hacer visible o invisible
* Cambiar color cada uno de los componentes
"""

from visual import *

from textN import *


# los vertices de cada una de las caras del voxel
vertices = [[(0,0,0), (0,0,1), (1,0,1), (1,0,0), (0,0,0)],
            [(0,0,1), (1,0,1), (1,1,1), (0,1,1), (0,0,1)],
            [(0,0,0), (0,0,1), (0,1,1), (0,1,0), (0,0,0)],
            [(1,0,0), (1,1,0), (1,1,1), (1,0,1), (1,0,0)],
            [(0,0,0), (1,0,0), (1,1,0), (0,1,0), (0,0,0)],
            [(0,1,0), (0,1,1), (1,1,1), (1,1,0), (0,1,0)]]

# las posiciones para cada uno de los textos en las caras del voxel
posText= [(.5,0,.3),(.5,.3,1),(0,.3,.5),(1,.3,.5),(.5,.3,-.01),(.5,1,.7)]

# el diametro del esqueleto
skel_radius=0.03

class voxel:
    """ El constructor Voxel

    pos    = La posición donde se pone el voxel
```

```
v_sk   = Si es visile el esqueleto
v_faces = Si son visibles las caras
v_texts = Si son visibles los textos
strings = Los numeros que se mostraran en cada una de las caras
color_f = El color de las caras
color_t = El color de los textos
color_s = El color del esqueleto
display = La ventana donde se dibujara el voxel
"""
def __init__(self,pos=(0,0,0),v_sk=1, v_faces=1, v_texts=0, display=None,
             strings=["1","2","3","4","5","6"], color_f=color.blue,
             color_t=color.red, color_s=color.white):

    if display == None:
        display = scene
    self.display = display
    self.pos = vector(pos)
    self.v_sk=v_sk
    self.v_faces=v_faces
    self.v_texts=v_texts

    # el esqueleto del voxel
    sk1 = curve(pos=[(0,0,0), (0,0,1), (1,0,1), (1,0,0), (0,0,0)],
        radius=skel_radius, visible=v_sk, color=color_s, display=display)
    sk2 = curve(pos=[(0,0,1), (1,0,1), (1,1,1), (0,1,1), (0,0,1)],
        radius=skel_radius, visible=v_sk, color=color_s, display=display)
    sk3 = curve(pos=[(0,0,0), (0,0,1), (0,1,1), (0,1,0), (0,0,0)],
        radius=skel_radius, visible=v_sk, color=color_s, display=display)
    sk4 = curve(pos=[(1,0,0), (1,1,0), (1,1,1), (1,0,1), (1,0,0)],
        radius=skel_radius, visible=v_sk, color=color_s, display=display)
    sk5 = curve(pos=[(0,0,0), (1,0,0), (1,1,0), (0,1,0), (0,0,0)],
        radius=skel_radius, visible=v_sk, color=color_s, display=display)
    sk6 = curve(pos=[(0,1,0), (0,1,1), (1,1,1), (1,1,0), (0,1,0)],
        radius=skel_radius, visible=v_sk, color=color_s, display=display)
    self.skeleton=[sk1,sk2,sk3,sk4,sk5,sk6]

    # las caras del cubo
    f1 = convex(pos=[(0,0,0), (0,0,1), (1,0,1), (1,0,0), (0,0,0)],
        radius=0.1, color=color_f, visible=v_faces, display=display)
    f2 = convex(pos=[(0,0,1), (1,0,1), (1,1,1), (0,1,1), (0,0,1)],
        radius=0.1, color=color_f, visible=v_faces, display=display)
    f3 = convex(pos=[(0,0,0), (0,0,1), (0,1,1), (0,1,0), (0,0,0)],
        radius=0.1, color=color_f, visible=v_faces, display=display)
    f4 = convex(pos=[(1,0,0), (1,1,0), (1,1,1), (1,0,1), (1,0,0)],
        radius=0.1, color=color_f, visible=v_faces, display=display)
    f5 = convex(pos=[(0,0),(0,1),(1,1),(1,0),(0,0)],radius=0.1,
        color=color_f, visible=v_faces, display=display)
    f6 = convex(pos=[(0,1,0), (0,1,1), (1,1,1), (1,1,0), (0,1,0)],
        radius=0.1, color=color_f, visible=v_faces, display=display)
```

```
    self.faces=[f1,f2,f3,f4,f5,f6]

    # el texto en las caras
    t1=text(pos=(.5,0,.3), string='1', color=color_t, depth=0.05, height=.5,
        justify='center',up=(0,0,1), visible=v_texts, display=display)
    t2=text(pos=(.5,.3,1), string='2', color=color_t, depth=0.05, height=.5,
        justify='center', visible=v_texts, display=display)
    t3=text(pos=(0,.3,.5), string='3', color=color_t, depth=0.05, height=.5,
        justify='center',axis=(0,0,1),up=(0,1,0), visible=v_texts,
        display=display)
    t4=text(pos=(1,.3,.5), string='4', color=color_t, depth=0.05, height=.5,
        justify='center',axis=(0,0,-1),up=(0,1,0), visible=v_texts,
        display=display)
    t5=text(pos=(.5,.3,-.01), string='5', color=color_t, depth=0.05,
        height=.5,justify='center',axis=(-1,0,0), visible=v_texts,
        display=display)
    t6=text(pos=(.5,1,.7), string='6', color=color_t, depth=0.05, height=.5,
        justify='center',up=(0,0,-1), visible=v_texts, display=display)
    self.texts=[t1,t2,t3,t4,t5,t6]

    self.move(pos)


""" Mueve a otra posición el Voxel.

pos = es la posición a donde se mueve el voxel
"""
def move(self, pos):     # mueve el voxel a otra posición
    # muevo el esqueleto y las caras
    for i in range(6):
        for j in range(5):
            self.skeleton[i].pos[j]=vector(vertices[i][j])+ pos
            self.faces[i].pos[j]=vector(vertices[i][j])+ pos
        self.texts[i].reshape(pos=vector(posText[i])+pos)

""" Oculta algún elemento (caras, esqueleto o textos) de una cara del voxel.

cara         = Es la cara que se quier modificar.
objectSelected = Indica el elemento a ocultar.
                 0-Todos, 1-Esquelto, 2-Cara, 3-Texto
"""
def hide(self, cara, objectSelected=0):
    if objectSelected==0: # Todos
        self.skeleton[cara-1].visible=0
        self.faces[cara-1].visible=0
        self.texts[cara-1].makeinvisible()

    if objectSelected==1: # el esqueleto
        self.skeleton[cara-1].visible=0
```

```
        if objectSelected==2: # la cara
            self.faces[cara-1].visible=0

        if objectSelected==3: # el texto
            self.texts[cara-1].makeinvisible()

    """ Muestra algún elemento (caras, esqueleto o textos) de una cara del voxel.

    cara        = Es la cara que se quier modificar.
    objectSelected = Indica el elemento a mostrar.
                    0-Todos, 1-Esquelto, 2-Cara, 3-Texto
    """
    def show(self, cara, objectSelected=0):
        if objectSelected==0: # Todos
            self.skeleton[cara-1].visible=1
            self.faces[cara-1].visible=1
            self.texts[cara-1].makevisible()

        if objectSelected==1: # el esqueleto
            self.skeleton[cara-1].visible=1

        if objectSelected==2: # la cara
            self.faces[cara-1].visible=1

        if objectSelected==3: # el texto
            self.texts[cara-1].makevisible()

    """ Oculta algun elemento de todo el voxel.

    objectSelected = Indica el elemento a ocultar.
                    0-Todos, 1-Esquelto, 2-Cara, 3-Texto
    """
    def hideall(self, objSelected=0):
        for cara in range(0,6):
            self.hide(cara+1,objSelected)

    """ Muestra algun elemento de todo el voxel.

    objectSelected = Indica el elemento a mostrar.
                    0-Todos, 1-Esquelto, 2-Cara, 3-Texto
    """
    def showall(self, objSelected=0):
        for cara in range(0,6):
            self.show(cara+1,objSelected)

    def hideshow(self, cara,objectSelected=0):
        if objectSelected==0: # Todos
            if self.skeleton[cara-1].visible==1:
```

```
                self.hide(cara-1,0)
            else:
                self.show(cara-1,0)

    """ Pone los textos a las caras de los voxeles. Se le pasan textos
                                                                        200
    text = Una lista con los seis textos de las caras del voxel
    """
    def setTexts(self, text=['1','2','3','4','5','6']):
        for i in range(0,6):
            self.texts[i].string=text[i]
            self.texts[i].change_text(str(text[i]))

    """ Pone los textos a las caras de los voxeles. Se le pasan números

    valores = Una lista con los seis números de las caras del voxel.    210
              Si el número es positivo se considera como espacio blanco,
              si es negativo se le quita el signo.
    """
    def setTexts2(self, valores=[-1,-2,-3,-4,-5,-6]):
        for i in range(0,6):
            if valores[i]<=0:
                self.texts[i].string= str(-valores[i])
                self.texts[i].change_text(str(-valores[i]))
            else:
                self.texts[i].string=""                                 220
                self.texts[i].change_text("")

    """ Define el color de cada una de las caras

    cara = Es la cara que se quier modificar.
    color = Es el color que se le quiere poner a la cara.
    """
    def setColors(self, cara, color):
        self.faces[cara-1].color=color
                                                                        230
    # Las definicion de ls propiedades
    def get_texts(self): return self.texts
    def set_texts(self, texts):
        setTexts(texts)

    strings = property(get_texts,set_texts)

if __name__ == '__main__':
    v = voxel(pos=(1,1,1),v_texts=1, v_faces=1, v_sk=1)
```

voxsolido.py

```
#-----------------------------------------------------
# Name:       voxsolido.py
# Purpose:    La implementación del objeto Vox-solido
#
# Author:     Eliot Peña Rojas
#
# Created:    2006/05/01
# RCS-ID:     $Id: voxsolido.py $
# Copyright:  (c) 2006
# Licence:    GLP 2.0
# Version:    1.3
#-----------------------------------------------------

from visual import *
from voxel import *

""" Nos dice para cada tipo de voxel a que faceta pertenece cada una de sus
   caras segun su orientación.

Cara uno de los elemento tiene la siguiente forma:
   [no se utiliza ,atras, adelante, izq, der, arriba, abajo]

La numeración significa a que faceta pertenece la cara
0 = Superior
1 = Exterior
2 = Inferior
3 = Interior
"""
TiposFacetas=[[-1,1,1,3,1,0,2], #0
              [-1,1,1,3,1,0,2], #1
              [-1,1,1,3,1,0,2], #2
              [-1,0,1,3,1,0,2], #3
              [-1,2,1,3,1,0,2], #4
              [-1,1,0,3,1,0,2], #5
              [-1,1,2,3,1,0,2], #6
              [-1,1,1,2,1,0,2], #7
              [-1,1,1,1,2,0,2], #8
              [-1,2,1,0,1,0,2], #9
              [-1,2,1,0,1,0,2], #10
              [-1,1,3,3,1,0,2], #11
              [-1,1,3,1,3,0,2]] #12


# me dice cuales son las caras adyacentes a cada cara
CarasAdyacentes=[[],[2,3,4,5],[1,3,4,6],[1,2,5,6],[1,2,5,6],[1,3,4,6],[2,3,4,5]]

"""
```

```
  Me indica cuales es la cara adyacente a la cara X de un voxel v1 a otro
  voxel v2 si la cara X de v2 tiene adyacente otro voxel. La columna es la
  cara donde v2 es adyacente a v1
"""
carasAdyOtroVox=[ [0,0,0,0,0,0], #nada
                  [0,0,5,4,3,2,0], #cara 1
                  [0,6,0,4,3,0,1], #cara 2
                  [0,6,5,0,0,2,1], #cara 3
                  [0,6,5,0,0,2,1], #cara 4
                  [0,6,0,4,3,0,1], #cara 5
                  [0,0,5,4,3,2,0]] #cara 6


""" Regresa una lista donde ninguno de sus elementos se repite

lista = la lista que se desea convertir en conjunto
"""
def conjunto(lista):
    l2=[]
    l2.append(lista[0])
    for i in lista[1:]:
        encontre=0
        for j in l2:
            if i == j:
                encontre=1
                break
        if encontre==0: l2.append(i)
    return l2


""" Clase que implementa un VoxSolido
"""
class voxsolido:

    """ Contructor de la clase

    origen = la posición del voxel cero, segun la representación de voxeles
    datos  = El arreglo con la representación del voxel
    color_f = El color de las caras
    color_t = El color de los textos
    color_s = El color del esqueleto
    display = La ventana donde se dibujara el voxel
    visual = Si el proceso se realiza visualmente o sólo entrega el resultado
    """
    def __init__(self,origen=(0,0,0),display=None, visual=true, datos=None,
                 color_f=color.blue, color_t=color.red, color_s=color.white):

        if visual == true:
            if display == None:
```

```
            display = scene
        self.display = display
        self.origen = vector(origen)

    self.visual=visual
    self.objeto=datos
    self.Lvs=[]                 # la lista donde se almacenena los voxeles
    self.Facetas=[[],[],[],[]]  # Se guarda el número de las cara que
                                # pertenece a cada faceta, en el orden:
                                # [superior, exterior, inferior, interior]
    self.Ruta=[]
    self.MatrizAdy=[]
    self.Orientacion=[]         # la orientacion de los voxeles

    if self.objeto != None:
        # si me pasaron datos del voxsolido
        self.Tpiso= self.objeto[0][0] * self.objeto[0][0]
        for i in range(len(self.objeto[1])):
            # calculo la matriz de adyacencias
            self.MatrizAdy.append(self.adyacentes(self.objeto[1][i]))
            if self.visual==true:
                self.Lvs.append(
                    voxel(pos=origen+
                            vector(self.columna(self.objeto[1][i]),
                                   self.piso(self.objeto[1][i]),
                                   -self.renglon(self.objeto[1][i])-1),
                          v_texts=0, display=display,
                          color_f=color_f, color_t=color_t, color_s=color_s))
            self.Orientacion.append([3,4,5,2,6,1])

""" Regresa en que columna esta un voxel

x = El voxel del cual se quiere sabe en que columna esta
"""
def columna(self,x):
    return x%self.objeto[0][0]

""" Regresa en que renglon esta un voxel

x = El voxel del cual se quiere sabe en que renglon esta
"""
def renglon(self,x):
    return (x/self.objeto[0][0]) - ((x/self.Tpiso)*self.objeto[0][0])

""" Regresa en que piso esta un voxel

x = El voxel del cual se quiere sabe en que piso esta
"""
def piso(self,x):
```

```
    return x/self.Tpiso

"""  Regresa el renglon de la matriz de adyacencias para un voxel

x = El voxel del cual se quiere obtener la matriz de adyacencias
"""
def adyacentes(self, x):
    k=[]
    k.append(0)

    # la cara 1
    k.append(x - self.Tpiso)
    if k[1] < 0: k[1]=-1
    # la cara 2
    k.append(((self.renglon(x)-1)*self.objeto[0][0])+self.columna(x)+ \
             (self.piso(x)*self.Tpiso))
    if k[2] < (self.piso(x) * self.Tpiso): k[2]=-1
    # la cara 3
    k.append((self.renglon(x)*self.objeto[0][0])+(self.columna(x)-1)+ \
             (self.piso(x)*self.Tpiso))
    if k[3] < ((self.renglon(x)*self.objeto[0][0])+(self.piso(x)*self.Tpiso)):
        k[3]=-1
    # la cara 4
    k.append((self.renglon(x)*self.objeto[0][0])+(self.columna(x)+1)+ \
             (self.piso(x)*self.Tpiso))
    if k[4] >= ((self.renglon(x)+1)*self.objeto[0][0])+(self.piso(x)*self.Tpiso):
        k[4]=-1
    # la cara 5
    k.append(((self.renglon(x)+1)*self.objeto[0][0])+self.columna(x)+ \
             (self.piso(x)*self.Tpiso))
    if k[5] >= ((self.piso(x)+1)*self.Tpiso)+(self.piso(x)*self.Tpiso): k[5]=-1
    # la cara 6
    k.append(x + self.Tpiso)
    if k[6] >= (self.Tpiso*self.objeto[0][0]) : k[6]=-1

    # veo cuales voxceles estan en el voxsolido
    for i in range(1,7):
        if k[i] >= 0:
            try:
                indice = self.objeto[1].index(k[i])
            except:
                k[i]=-1

    return k


""" realiza la etiquetación del voxel segun el método del laberinto
"""
```

```
def tipoVoxel3(self):
    elementos=range(0, len(self.Ruta))

    # el primer voxel siempre esta avanzando hacia adelante
    self.MatrizAdy[self.getIndiceVoxel(self.Ruta[0])][0]=0
                                                                    200
    # posicion dentro del voxel [atras, adelante, izq, der, arriba, abajo]
    orden=[3,4,5,2,6,1]
    tipoUltimoVoxel=0;
    for i in elementos:

        vox=self.MatrizAdy[self.getIndiceVoxel(self.Ruta[i])]
        if i+1 == len(self.Ruta):
            next_vox=self.MatrizAdy[self.getIndiceVoxel(self.Ruta[0])]
        else:
            next_vox=self.MatrizAdy[self.getIndiceVoxel(self.Ruta[i+1])]   210

        act_vox=self.MatrizAdy[self.getIndiceVoxel(self.Ruta[i])] # el voxel actual
        # el voxel anterior
        if i>0:
            ant_vox=self.MatrizAdy[self.getIndiceVoxel(self.Ruta[i-1])]
        else:
            ant_vox=-1
        # el previo del anterior
        if i>1:
            ant2_vox=self.MatrizAdy[self.getIndiceVoxel(self.Ruta[i-2])]   220
        else:
            ant2_vox=-1


        # **** los casos normales
        # primero camino hacia enfrente
        orden2=orden
        ordenAct=-1 # en Vt2
        if vox[orden[1]] >= 0: next_vox[0]=0
        elif vox[orden[2]] >= 0:                                        230
            next_vox[0]=1
            orden2=[orden[3],orden[2],orden[0],orden[1],orden[4],orden[5]]
        elif vox[orden[3]] >= 0:
            next_vox[0]=2
            orden2=[orden[2],orden[3],orden[1],orden[0],orden[4],orden[5]]
        elif vox[orden[4]] >= 0:
            next_vox[0]=3
            act_vox[0]=6
        elif vox[orden[5]] >= 0:
            next_vox[0]=4                                               240
            act_vox[0]=5
```

```
# ***** para los casos del Q18 (subidas y bajadas y despues cambio de dirección
if ant_vox != -1:
    #subida
    if act_vox[0]==3: # en el actual subi
        if next_vox[0]==1: # el que sigue es a la izquierda
            if ant_vox[0]==6: # en el anterior era previo de subida
                ant_vox[0]=7                                                    250
        if next_vox[0]==2: # el que sigue es a la derecha
            if ant_vox[0]==6: # en el anterior era previo de subida
                ant_vox[0]=8
    # bajada
    if act_vox[0]==4: # en el actual baje
        if next_vox[0]==1: # el que sigue es a la izquierda
            if ant_vox[0]==5: # en el anterior era previo de bajada
                ant_vox[0]=10
        if next_vox[0]==2: # el que sigue es a la derecha
            if ant_vox[0]==5: # en el anterior era previo de bajada             260
                ant_vox[0]=9

    # **** para los casos del F20 (zigzag)
    if act_vox[0]==1 and ant_vox[0]==0: # si soy izquierda
        if next_vox[0]==2: # el que sigue es derecha
            act_vox[0]=11
    if act_vox[0]==2 and ant_vox[0]==0: # si soy derecha
        if next_vox[0]==1: # el que sigue es izquierda
            act_vox[0]=12
                                                                                270
    # es el previo a bajar Vt2
    if act_vox[0]==5:
        if next_vox[0]==4: # el que sigue es bajada
            # si es bajada e inmediatamente a la izquierda o derecha
            if next_vox[orden[2]] >=0 or next_vox[orden[3]]>=0 :
                act_vox[0]=9
                ordenAct=[5,2,4,3,6,1]

if ant_vox != -1 and ant2_vox != -1:
    # el actual es a la derecha, el anterior es a la izquierda                  280
    if act_vox[0]==2 and ant_vox[0]==1:
        # el anterior es a la izquierda
        if next_vox[0]==1 or next_vox[0]==6:
          # el previo al anterior es de bajada o de frente
            if ant2_vox[0]==0 or ant2_vox[0]==4:
                ant_vox[0]=11 # se parece como zig-zag

orden=orden2

# actualizo la orientacion del siguiente voxel                              290
if i+1 == len(self.Ruta):
    self.Orientacion[self.getIndiceVoxel(self.Ruta[0])]=orden
```

```
        else:
            self.Orientacion[self.getIndiceVoxel(self.Ruta[i+1])]=orden

        #actualizo la orientacion del voxel actual
        if ordenAct != -1:
            self.Orientacion[self.getIndiceVoxel(self.Ruta[i])]=ordenAct


""" Enumera las caras libres del voxolido en una secuencia

Utiliza una ruta que debio de ser creada previamente
"""
def numeraCarasLibres(self):
    # numera las caras libres de cada voxel
    numero=1
    for i in range(len(self.Ruta)):
        indice_en_arreglo=self.getIndiceVoxel(self.Ruta[i])
        # numero la matriz de adyacencias
        for j in range(1,7):
            if self.MatrizAdy[indice_en_arreglo][j]<0:
                self.MatrizAdy[indice_en_arreglo][j]= -(numero)
                numero=numero+1

        # le pongo los numeros al voxel
        if self.visual==true:
            self.Lvs[indice_en_arreglo].setTexts2(self.MatrizAdy[indice_en_arreglo][1:])
            self.Lvs[indice_en_arreglo].showall(3) # muestro los textos


""" Calcula las facetas

Utiliza una ruta que debio de ser creada previamente
Obtiene las facetas utilizando los datos de orientacion (orden) que se
calculo al tipificar los voxeles
"""
def calculaFacetas3(self):
    # obtengo las facetas
    for i in range(len(self.Ruta)):
        # obtengo el tipo de facetas de cada uno de los nodos de la ruta
        fac=TiposFacetas[self.MatrizAdy[self.getIndiceVoxel(self.Ruta[i])][0]]
        tipo_fac=self.MatrizAdy[self.getIndiceVoxel(self.Ruta[i])][0]

        # para que de las caras en orden
        orden=self.Orientacion[self.getIndiceVoxel(self.Ruta[i])]
        #print "voxel:", self.Ruta[i], " - Orden:",orden, " - FAC:", fac
        #print "Matriz:",self.MatrizAdy[self.getIndiceVoxel(self.Ruta[i])]

        # ve de que tipo es y se agrega a la facetas sus caras

        #corrigo el orden en que se tienen que agregar las caras a la faceta,
```

```
        # sirve si un voxel tiene mas de una cara en la misma faceta.
        ordenAgregarFaceta=[1,3,4,2,5,6]
        if tipo_fac==9:
          ordenAgregarFaceta=[6,4,5,1,3,2]
        elif tipo_fac==7:
          ordenAgregarFaceta=[6,3,4,2,1,5]
        elif tipo_fac==5:
          ordenAgregarFaceta=[1,3,4,5,6,2]
        elif tipo_fac==6:
          ordenAgregarFaceta=[1,3,4,6,5,2]

        for j in ordenAgregarFaceta:
            # el número de la cara en el voxsolido
            v=self.MatrizAdy[self.getIndiceVoxel(self.Ruta[i])][orden[j-1]]
            if v <0:
                # si es una cara libre, lo agrego a la faceta que corresponda
                # [número de cara en voxel, voxel, cara del voxel ]
                self.Facetas[fac[j]].append([-v,self.Ruta[i],orden[j-1]])

"""  Regresa las caras adyacentes a una de las caras, segun la ruta

voxel = el voxel al cual pertenece la cara
cara = la cara de la cual se obtendr"sn sus caras adyacentes
"""
def getCarasAdyacentes(self,voxel,cara):
    # da las caras adyacentes del voxel solicitado
    ady=[]
    tipoFacetaCara= TiposFacetas[self.MatrizAdy[self.getIndiceVoxel(voxel)][0]][cara]

    # primer pongo las caras adyacentes del mismo voxel
    cady=CarasAdyacentes[cara] # las caras adyacentes del mismo voxel
    for i in cady:
        v=self.MatrizAdy[self.getIndiceVoxel(voxel)][i]
        if v<0:
            ady.append([-v,voxel,i])
        else:
            #obtengo las caras adyacentes de otro voxel
            encontro=0
            caraAdy=cara
            while encontro == 0:
                v2=self.MatrizAdy[self.getIndiceVoxel(v)][caraAdy]
                if v2<0:
                    # encontre la cara libre
                    # (numero de cara, numero voxel, cara del voxel)
                    ady.append([-v2,v,caraAdy])
                    encontro=1
                else:
                    # esa cara es adyacente a otro voxel, busco en el otro voxel
```

```
                    v=v2
                    # la cara adyacente del otro voxel
                    caraAdy=carasAdyOtroVox[cara][i]

        return conjunto(ady)


    """ Obtiene en que indice del arreglo se encuentra el voxel con numero X.
        Si no se encuentra regresa -1.
                                                                              400
    voxel = el número del voxel buscado
    """
    def getIndiceVoxel(self,voxel):
        #print voxcel
        # regresa el indice de donde se encuentra el voxel en el arreglo de los voxeles
        try:
            pos=self.objeto[1].index(voxel)
            return pos
        except:
            return -1                                                         410

    """ Crea una ruta que recorre el voxolido.

    pos = en que voxel se desea comenzar la ruta
    """
    def creaRuta(self, pos): # crea la ruta para recorrer el voxsolido
        # limpio la lista
        del self.Ruta[:]

        # guardo el primer elemento de la ruta                                420
        self.Ruta.append(pos)

        # hago el recorrido por el voxsolido
        self.recorre(self.getIndiceVoxel(pos))

    """ Recorre el voxolido por us adyacencias.

    pos = el voxel que esta revisando en este momento
    """
    def recorre(self,pos):                                                    430
        # obtengo un camino para el Voxsolido
        # para cada uno de las caras del voxel

        for i in range(1,7):
            # Si tiene una adyacencia en esta cara
            if self.MatrizAdy[pos][i]>=0:
                try:
                    # si ya esta en la ruta no hago nada
                    self.Ruta.index(self.MatrizAdy[pos][i])
```

```
                except:
                    # si no esta en la ruta, lo agrego y recorro el nuevo voxel
                    self.Ruta.append(self.MatrizAdy[pos][i])
                    self.recorre(self.getIndiceVoxel(self.MatrizAdy[pos][i]))
                    #break

    """ Cambia de color la cara de un voxel

    voxel = el voxel a tratar
    cara = la cara del voxel a cambiar de color
    """
    def remarkVx(self, voxel, cara):
        if self.visual == true:
            self.Lvs[self.getIndiceVoxel(voxel)].setColors(cara,color.yellow)

    """ Cambia el texto de las caras del voxel

    vx   = el voxel sobre el que se trabaja
    texts = los textos de cada una de las caras
    """
    def setTexts2(self, vx, texts=[0,0,0,0,0,0]):
        if self.visual == true:
            self.Lvs[vx].setTexts2(texts)

    """ Imprime la matriz de adyacencias

    """
    def printMatrizAdy(self):
        for i in range(len(self.MatrizAdy)):
            print "voxel: " , self.objeto[1][i] , " -> ", self.MatrizAdy[i]


if __name__ == '__main__':

    # genero la vista
    scene1 = display(title='voxsolido',width=400, height=400,
                     center=(0,0,0), background=(0,0,0))

    origen = vector(0,0,0)
    tuerca =[[3],[0,1,2,3,5,6,7,8]]

    vs = voxsolido(origen=origen,datos=tuerca, display=scene1)
    print "Matriz Adyacencias: ", vs.MatrizAdy
    vs.creaRuta(vs.objeto[1][0])
    print "Ruta para recorrer el voxsolido:", vs.Ruta
    vs.tipoVoxel2()
    print "Matriz Adyacencias: ", vs.MatrizAdy
    vs.calculaFacetas()
```

```
    while 1:                                                    490
        true
```

repPlanar.py

```
#-----------------------------------------------------
# Name:       repPlanar.py
# Purpose:    Calcula la representación planar usando un voxsolido
#
# Author:     Eliot Peña Rojas
#
# Created:    2006/05/01
# RCS-ID:     $Id: repPlanar.py $
# Copyright:  (c) 2006
# Licence:    GLP 2.0
# Version:    1.3
#-----------------------------------------------------


from visual import *
from random import *


# la clase de los nodos de una grafica
class Nodo:
    """ Contructor de la clase

    pos    = la posición en que se pondr"s el nodo
    datos  = El arreglo con los datos del nodo
    color  = El color de fondo de los nodos
    display = La ventana donde se dibujara el voxel
    visual = Si el proceso se realiza visualmente o sólo entrega el resultado
    show   = Si se muestra el nodo o no
    """
    def __init__(self,display=None, datos=None, color=color.blue, pos=(0,0,0),
                 show=true, visual=true):

        self.datos=datos # 0-Nombre, 1-Voxel, 2-Cara, 3-nodosAdyacentes
        self.marca=None # la marca del nodo
        self.visual=visual

        self.pos=vector(pos)

        if visual==true:
            if display == None:
                display = scene
            self.display = display
            self.pos = vector(pos)

            # el nodo
            self.n = convex(color=color, display=display, visible=show)
            # la etiqueta del nodo
```

```
            self.t = label(pos=self.pos, text=self.datos[0],line=0, box=0,
                        opacity=0,display=display)

            t = arange(0,2*pi,0.1)


            # nodos pequeños *1.0, para conos grandes *1.5
            self.n.pos = pos+transpose((sin(t), cos(t), 0*t))*1.3

    """ Muestra el nodo
    """
    def show(self):
        if self.visual==true:
            self.n.visible=true
            self.t.visible=true

    """ Oculta el nodo
    """
    def hide(self):
        if self.visual==true:
            self.n.visible=false
            self.t.visible=false

    """ Cambia el color del nodo, de rojo a azul y viceversa
    """
    def blink(self, status):
       if self.visual==true:
            if status == 1:
                self.n.color = color.red
            elif status == 2:
                self.n.color = (0.35,0.35,0.35)
            else:
                self.n.color = color.blue


# incrementos para ajustar la grafica
inc=[1.5,1,1.5,2.5]


# la clase que calcula la representacion planar
class gPlanar:

    """ El contructor de la clase

    display = La ventana donde se dibujara el voxel
    vs      = El voxolido que se utilizara para obtener la representacion
    visual = Si el proceso se realiza visualmente o sólo entrega el resultado
    """
    def __init__(self,display=None, vs=None, visual=true):
```

```
if visual==true:
    if display == None:
        display = scene                                                    100
    self.display = display

self.visual=visual
self.vs=vs      # el voxolido
self.nodos=[]    # los nodos de este objeto
self.aristas=[]  # los vertices de la grafica
self.p_final=[]  # inicio y fin de las facetas para el paso final
self.diccN=dict()
self.aristas2=[] # los vertices de la grafica, en una lista de que vertice a que vertice
                                                                           110
if self.vs != None:
    # si hay datos, creo la representacion
    ld=[]
    x=0
    y=0
    c=0
    for fac in vs.Facetas:
        # para cada una de las facetas
        x=0
        for j in range(len(fac)):                                          120
            # agrego un nodo
            self.nodos.append(Nodo(datos=[str(fac[j][0]),fac[j][1],fac[j][2]],
                                   display=display,
                                   pos=(x,y,0),show=false, visual=visual))

            # Para diccionario, el nombre del nodo, posicion en arreglo
            ld.append((str(fac[j][0]),len(self.nodos)-1))
            x=x+(4*inc[c])
        y=y-4
        c=c+1                                                              130

    # agrego las versiones "a" de los objetos en la parte superior
    x=0
    y=4
    c=3
    for j in range(len(vs.Facetas[3])):
        # agrego un nodo
        self.nodos.append(Nodo(datos=[str(fac[j][0])+"a",fac[j][1],fac[j][2]],
                               display=display,
                               pos=(x,y,0),show=false, visual=visual))     140

        # Para diccionario, el nombre del nodo, posicion en arreglo
        ld.append((str(fac[j][0])+"a",len(self.nodos)-1))
        x=x+(4*inc[c])
    y=y-4
```

```
        c=c+1

        # agrego las versiones "b" de los objetos en la parte de la izquierda
        x=-4.5
        y=4
        c=0
        etiqueta="c" # para que etiquete sólo el primer elemento como "c"
        for fac in [vs.Facetas[3][-1:],vs.Facetas[0][-1:],vs.Facetas[1][-1:], \
                    vs.Facetas[2][-1:],vs.Facetas[3][-1:]]:
            for j in range(len(fac)):
                # agrego un nodo
                self.nodos.append(Nodo(datos=[str(fac[j][0])+etiqueta,fac[j][1],
                                              fac[j][2]],
                                              display=display,pos=(x,y,0),
                                              show=false, visual=visual))

                # Para diccionario, el nombre del nodo, posicion en arreglo
                ld.append((str(fac[j][0])+etiqueta,len(self.nodos)-1))
            y=y-4
            c=c+1
            etiqueta="b"

        # creo el diccionario de nodos
        self.diccN = dict(ld)
        ld=[]

    # para el programa paso a paso
    p_l = None          # la lista sobre la que se itera
    p_nActual = None # el último nodo que trabaje


""" Limpia toda la estructura
"""
def clear(self):
    # borro todos los datos de los nodos
    while len(self.nodos) > 0:
        n = self.nodos.pop(0)
        n.n.visible=0
        n.t.visible=0
        n=None
    self.nodos=None
    self.nodos=[]

    while len(self.aristas)>0 : # los vertices de la grafica
        a = self.aristas.pop(0)
        a.visible=0
        a=None
    self.aristas=None
    self.aristas=[]
```

```
    self.aristas2=None
    self.aristas2=[]

    self.diccN=dict()

""" Guarda la lista de adyacencias en un archivo
Utiliza los datos calculados previamente

archivo = el archivo donde se guardaran los datos
"""
def writeVertices(self,archivo):
    print "---- GUARDANDO DATOS (aristas.data) ----"
    lista=[]
    f=open(archivo, 'w')
    # le quito a la lista las letras

    for n in self.aristas2:
        n2=n.replace("a","")
        n2=n2.replace("b","")
        f.write(n2+"\n")
        lista.append(n2)
    f.close()

    print lista

""" Muestra todos los nodos de la gr"sfica
"""
def drawNodos(self):
    for n in self.nodos: n.show()

""" Realiza cada uno de los pasos para obtener la representación planar

step = El paso del algoritmo, 1=inicio de algoritmo.
"""
def setVerticesSTEPS(self, step):
    # Hace la representación planar paso a paso

    # en el primer paso
    if step == 1:
        self.p_l=[self.nodos[0].datos]   # agrego el primer nodo a la lista
        self.p_nActual=None          # no hay nadie a quien desmarcar


    if len(self.p_l) > 0: # para cada una de los nodos
        n=self.p_l.pop(0) # lo saco

        # desilumino el nodo, si habia uno iluminado
        if self.p_nActual!= None:
            self.blinkNodo(self.p_nActual.datos[0],0)  # desilumino el último nodo visitado
```

```
self.p_nActual=self.nodos[self.diccN[str(n[0])]]  # tomo el nodo

# si no lo he visitado, lo visito
if self.p_nActual.marca == None:
    self.p_nActual.marca='v' # lo marco como visitado
                                                                    250
    #  lo remarco en el vox-solido
    self.blinkNodo(self.p_nActual.datos[0],1)  # ilumino el nodo
    self.vs.remarkVx(n[1],n[2])                # ilumino el vox-solido

    # obtengo las adyacencias
    l2=self.vs.getCarasAdyacentes(n[1],n[2])
    print "Adyacentes de [(",n[0], ") ", n[1], ",", n[2] , "]: ", l2
    for a in l2:
        # el nuevo destino
        try:                                                        260
            k1=self.nodos[self.diccN[str(a[0])]]
        except:
            print "A[0]:",str(a[0])
            print "diccN:",self.diccN

        nSiguiente=k1

        ##### Calculo el destino correcto, segun tenga una "a" o una "b"
        # verifico si hay un nodo destino que sea "a"
        k2=None                                                     270
        try:
            k2=self.nodos[self.diccN[str(a[0])+"a"]]
        except:
            k2=None

        # verifico si hay un nodo destino que sea "a"
        k3=None
        try:
            k3=self.nodos[self.diccN[str(a[0])+"b"]]
        except:                                                     280
            k3=None

        # si tiene tanto un nodo "a" y un nodo "b", tomo el mas cercano
        if k2!=None and k3!=None:
            if(abs(k3.pos - self.p_nActual.pos) < abs(k2.pos - self.p_nActual.pos)):
                k2=k3
        elif k2==None and k3!=None:
            k2=k3

        # veo cual es el nodo mas cercano al nodo origen            290
        if k2!=None:
            if(abs(k2.pos - self.p_nActual.pos) < abs(k1.pos- self.p_nActual.pos)):
```

```
        nSiguiente=k2


####### veo si el origen tenia una a para trazar la arista
nActual=self.p_nActual

k2=None
try:                                                                    300
    k2=self.nodos[self.diccN[str(n[0])+"a"]]
except:
    k2=None

k3=None
try:
    k3=self.nodos[self.diccN[str(n[0])+"b"]]
except:
    k3=None
                                                                        310
# si tiene tanto un nodo "a" y un nodo "b", tomo el mas cercano
if k2!=None and k3!=None:
    if(abs(k3.pos - nSiguiente.pos) < abs(k2.pos - nSiguiente.pos)):
        k2=k3
elif k2==None and k3!=None:
    k2=k3


if k2!=None:
    if(abs(nSiguiente.pos - k2.pos) < abs(nSiguiente.pos - nActual.pos)): 320
        nActual=k2


####### trazo la arista
if nSiguiente.marca == None:
    # pongo los vertices a sus adyacentes
    if self.visual==true:
        a1 = curve(pos=[vector(nActual.pos)-(0,0,0.2), \
                        vector(nSiguiente.pos)-(0,0,0.2)],
                   radius=0.1, visible=true,                            330
                   display=self.display)
        self.aristas.append(a1)

    #if k != None:
    self.p_l.extend([a])   # los agrego a la lista
    #print "Se agrego a la lista :", a
    #print "Vector de ", k2.datos[0], " a ", nSiguiente.datos[0]
    self.aristas2.append(nActual.datos[0] +","+ nSiguiente.datos[0])

                                                                        340
    #### veo si el origen y objetivo tienen componentes "a" o "b" o "c"
```

```
if self.visual==true:

    # si el origen es "a" y el destino es "c"
    try:
        k=self.nodos[self.diccN[str(a[0])+"c"]]
        k2=self.nodos[self.diccN[str(n[0])+"a"]]

        a1 = curve(pos=[vector(k.pos)-(0,0,0.2), \
                        vector(k2.pos)-(0,0,0.2)],
                   radius=0.1, visible=true,
                   display=self.display)
        self.aristas.append(a1)
    except:
        # si el origen es un "c" o "a" y el destino un "a"
        try:
            k=self.nodos[self.diccN[str(n[0])+"c"]]
            k1=self.nodos[self.diccN[str(n[0])+"a"]]

            k2=self.nodos[self.diccN[str(a[0])+"a"]]

            # tomo el de la minima distancia entre el origen y el destino
            if k!=None and k1!=None:
                if(abs(k2.pos - k1.pos) < abs(k2.pos - k.pos)):
                    k=k1

            a1 = curve(pos=[vector(k.pos)-(0,0,0.2), \
                            vector(k2.pos)-(0,0,0.2)],
                       radius=0.1, visible=true,
                       display=self.display)
            self.aristas.append(a1)
        except:


            # si los dos son "a"
            try:
                k=self.nodos[self.diccN[str(a[0])+"a"]]
                k2=self.nodos[self.diccN[str(n[0])+"a"]]
                a1 = curve(pos=[vector(k.pos)-(0,0,0.2), \
                                vector(k2.pos)-(0,0,0.2)],
                           radius=0.1, visible=true,
                           display=self.display)
                self.aristas.append(a1)
            except:
                k=None

            # si el inicio es "b" y el destino tiene un "b" o "c"
            try:
                k=self.nodos[self.diccN[str(n[0])+"b"]]
```

```
                                # veo si tiene un "b"
                                k2=None
                                try: k2=self.nodos[self.diccN[str(a[0])+"b"]]
                                except: k2=None

                                # veo si tiene un "c"
                                k3=None
                                try: k3=self.nodos[self.diccN[str(a[0])+"c"]]
                                except: k3=None
                                                                                400
                                # si tiene los dos, tomo el de minima distancia
                                if k2!=None and k3!=None:
                                    if(abs(k.pos - k3.pos) < abs(k.pos - k2.pos)):
                                        k2=k3
                                elif k2==None and k3!=None:
                                    k2=k3

                                if k2 != None:
                                    a1 = curve(pos=[vector(k.pos)-(0,0,0.2), \
                                                    vector(k2.pos)-(0,0,0.2)],  410
                                               radius=0.1, visible=true,
                                               display=self.display)
                                    self.aristas.append(a1)
                            except:
                                k=None
        else:
            # ya visite este nodo, lo marco de otro color
            # desilumino el último nodo visitado
            self.blinkNodo(self.p_nActual.datos[0],2)
                                                                                420
        return 1
    else:
        return 0


def blinkNodo(self, nombre=None,blink=0):
    if nombre!=None and self.visual==true:

        # marco el que me indicaron
        self.nodos[self.diccN[nombre]].blink(blink)                             430

        # veo si tiene nodos repetidos "a", "b" o "c"
        try:
            self.nodos[self.diccN[nombre+"a"]].blink(blink)
            self.nodos[self.diccN[nombre+"b"]].blink(blink)
            self.nodos[self.diccN[nombre+"c"]].blink(blink)
        except:
            try:
                self.nodos[self.diccN[nombre+"b"]].blink(blink)
```

```
            except:                                                    440
                try:
                    self.nodos[self.diccN[nombre+"a"]].blink(blink)
                except:
                    None


if __name__ == '__main__':
    print "Este programa no funciona aislado, se debe ejecutar el programa 'animación.py'"
    true
```

animacion.py

```
#-----------------------------------------------------------
# Name:        animacion.py
# Purpose:     Hace la animación que muestra como se encuentra la representación planar
#
# Author:      Eliot Peña Rojas
#
# Created:     2006/05/01
# RCS-ID:      $Id: animacion.py $
# Copyright:   (c) 2006
# Licence:     GLP 2.0
# Version:     1.0
#-----------------------------------------------------------

import getopt, sys

from visual import *
from voxsolido import *
from repPlanar import *
from visual.controls import *
from time import *

# creo los objetos del voxolido y de la representación planar


### FUNCIONES REALCIONADAS
def movScene(mov):
    if mov == 'left': return (-1,0,0)
    elif mov == 'right': return (1,0,0)
    elif mov == 'up': return (0,1,0)
    elif mov == 'down': return (0,-1,0)


def animacionStart(valor):
    global animacion, b2
    animacion=valor
    if visual==true:
        b2.text='Animando . . .'

def nextStep():
    global paso, b2, animacion
    global rP, vs

    if paso >= 0:
        if paso == 0:
            if rP == None:
                # cargo el voxolido
                cargaVoxolido('simple')
```

```
            # inicio al representacion planar
            rP = gPlanar(display=scene2,vs=vs,visual=visual)
        else:
            vs=None
            cargaVoxolido('simple')
            rP.clear()
            rP = gPlanar(display=scene2,vs=vs,visual=visual)

        # dibujo los nodos
        rP.drawNodos()
    else:
        # hago el siguiente paso
        if rP.setVerticesSTEPS(paso) == 0 :
            # deshabbilito el boton
            #b1.text='finalizo'
            tiempo2=time();

            print "\n ***** Terminado obtener adyacentes: ", \
                    strftime("%a,%d%b%Y%H:%M:%S", gmtime(tiempo2)), \
                    ". -> Duracion:",tiempo2-tiempo ," segundos \n ****"

            # guardo los datos de los vertices
            rP.writeVertices(archivoSalida)
            paso=-1
            animacion=0

            print "\n\n ***** Terminando proceso: ", \
                  strftime("%a,%d%b%Y%H:%M:%S", gmtime())

            if visual==true:
                b2.text='Anima'
            else:
                sys.exit(1)
    paso=paso+1


def cargaVoxolido(nombre):
    global vs,tiempo, voxsolidoDatos

    print "*****************************************************************************"
    tiempo=time();
    print " ***** Iniciando proceso: ", \
          strftime("%a,%d%b%Y%H:%M:%S", gmtime(tiempo))

    print voxsolidoDatos

    # creo el voxolido.
    vs = voxsolido(origen=vector(0,0,0),datos=voxsolidoDatos, display=scene1,
                   color_s=color.black,visual=visual)
```

```
    # se calcula la ruta del voxolido
    vs.creaRuta(vs.objeto[1][0])
    print "\n -> Ruta para recorrer el voxsolido:", vs.Ruta                    100

    print "\n -> Matriz de adyacencias (sin tipificar los voxeles):"
    # se calcula la matriz de adyacencias
    vs.printMatrizAdy()

    vs.tipoVoxel3()
    print "\n -> Matriz Adyacencias (con los voxeles Tipificados):"
    vs.printMatrizAdy()

    tiempo2=time();                                                           110
    print "\n ***** Termino calculo de Matriz de adyacencias: ", \
             strftime(" %a, %d %b %Y %H:%M:%S", gmtime(tiempo2)), \
             ". -> Duracion:",tiempo2-tiempo ," segundos \n ****"

    # numera todas las caras libre del vox-solido, se basa en una ruta
    vs.numeraCarasLibres()

    # se calculan las facetas
    vs.calculaFacetas3()
    print "\n -> Facetas [Superior, Exterior, Inferior, Interior]:\n", \       120
          vs.Facetas,"\n "

    tiempo2=time();
    print "\n ***** Termino calculo de facetas: ", \
            strftime(" %a, %d %b %Y %H:%M:%S", gmtime(tiempo2)), \
            ". -> Duracion:",tiempo2-tiempo ," segundos \n ****"

    if visual==true:
        print "\n%%%%%%%%%%%%%%%%%%%%%%%%%%%%% "
        print "NOTA. Los tiempo de aqui en adelante pueden variar ", \       130
              "dependiendo del tiempo que el usuario se tarde en animar la grafica"
        print " %%%%%%%%%%%%%%%%%%%%%%%%%%%%% \n"


def ayuda():
    print """ Programa que obtiene la representacion planar de un voxsolido
        Usar:
            animacion.py [opciones]
            Los datos del vox-solido a trabajar se localizan en el
            archivo voxsolidoDatos.py                                         140
        Opciones:
            -h
            --help
            --ayuda     Muestra este mensaje.
            -n
```

```
        --novisual Hace los calculos sin mostrar la grafica.
        --salida   El archivo donde se guardan las aristas de la grafica.
        --velocidad La velocidad de ejecución de la animación, valores de
                   0.1 a 1, 0.1 es lo mas r"spido.
    """

######### main
vs = 0
rP = None
paso=0
animacion=0
velocidad=0.3
visual=true
archivoSalida="aristas.data"
seleccion=None
tiempo=None
voxsolidoDatos=None # el vox-solido con el que se va a trabajar

# tomo los parametros de la línea de comandos
try:
    opts, args = getopt.getopt(sys.argv[1:], "nh", \
                           ["ayuda","novisual","salida=","velocidad="])
except getopt.GetoptError:
    # print help information and exit:
    ayuda()
    sys.exit(2)

for o, a in opts:
    if o in ("-n","--novisual"):
        visual = false
        velocidad = 0.1
        mideTiempo=1
    if o in ("-h", "--help","--ayuda"):
        ayuda()
        sys.exit()
    if o == "--salida":
        archivoSalida = a
    if o == "--velocidad":
        velocidad = a

# importo los datos del vox-solido con el que se va a trabajar
from voxsolidoDatos import voxsolidoDatos

if visual==true:
    ### Animación de representación planar
    # la vista del voxolido
    scene1 = display(x=0, y=0, title='Vox-solido',width=400, height=400,
                     center=(0,0,0), background=(0,0,0))
```

```
    #scene1.foreground=color.black
    scene1.background = color.white
    scene1.ambient=0.8

    # la vista de la representación planar
    scene2 = display(x=400, y=0, title='Representación Planar',width=600, height=300,
                     center=(22,-6,0), background=(0,0,0), pos=(400,1000), userspin = 0)

    #scene2.foreground=color.red
    scene2.background = (.3,.3,.3)
    scene2.ambient=0.8

    # los controles
    c = controls(x=400, y=300, width=300, height=150, range=60, \
                 title='Controles de Animación')
    b1 = button(pos=(-35, 10), height=20, width=40, text='Step', \
                 color=color.blue, action=lambda: nextStep())
    b2 = button(pos=(10, 10), height=20, width=40, text='Anima', \
                 color=color.blue, action=lambda: animacionStart(1))
else:
    b2=None
    scene1=None
    scene2=None
    animacionStart(1)


while 1:
    if visual==true:
        # los eventos de los controles
        c.interact()
        #print scene1.mouse.camera

        # --------
        #los enventos de las representacion planar
        if scene2.kb.keys:
            s = scene2.kb.getkey()
            if s == 'left' or s == 'right' or s == 'up' or s == 'down':
                scene2.center=scene2.center + movScene(s)

        if scene2.mouse.events:
            me = scene2.mouse.getevent()
            if seleccion:
                seleccion.color= color.blue
                seleccion=None
            elif me.pick: # le dio a un objeto
                seleccion = me.pick
                seleccion.color= color.red

        # --------
```

```
        # los envento del voxsolido
        if scene1.kb.keys:
            s = scene1.kb.getkey()
            if s == 'left' or s == 'right' or s == 'up' or s == 'down':
                scene1.center=scene1.center + movScene(s)

    # ---------
    # veo si se tiene que continuar animando o no.
    if animacion == 1:
        sleep(velocidad) # me espero lo que hayan seleccionado
        nextStep()
```

voxolidoDatos.py

```
#-----------------------------------------------------------
# Name:        voxsolido.py
# Purpose:     Los datos que tiene el vox-solido con el que se trabajar"s
#
# Author:      Eliot Peña Rojas
#
# Created:     2006/05/01
# RCS-ID:      $Id: voxsolidoDatos.py $
# Copyright:   (c) 2006
# Licence:     GLP 2.0
# Version:     1.3
#-----------------------------------------------------------

## los datos del vox-solido que se va trabajar
## la variable debe ser voxsolidoDatos, la representacion es casi la misma que la
## representacion planar solo que en no se escribe la matriz de adyacencias porque
## se calcula en el proceso

# los datos del voxolido, torito simple
voxsolidoDatos =[[3],[0,1,2,3,5,6,7,8]]

# la tuerca con semi-torcida
#voxsolidoDatos =[[4],[0,1,2,3,4,7,8,9,11,13,14,15]]

# una tuerca mas grande de 6 de largo
#voxsolidoDatos =[[6],[0,1,2,3,4,5,6,11,12,17,18,23,24,29,30,31,32,33,34,35]]

# tuerca dali
#voxsolidoDatos =[[3],[0,1,2,9,11,12,14,15,16,17]]

# tuerca dali larga
#voxsolidoDatos =[[4],[0,1,2,3,16,19,20,23,24,27,28,29,30,31]]

#L16
#voxsolidoDatos =[[4],[1,2,6,7,11,16,17,20,27,30,31,36,40,41,45,46]]

#Q18
#voxsolidoDatos=[[4],[1,2,6,7,11,16,17,27,31,32,36,46,47,52,56,57,61,62]]

# F20
#voxsolidoDatos=[[5],[1,2,5,6,10,14,18,19,22,23,27,28,33,34,35,39,40,41,46,47]]

# V32
#voxsolidoDatos=[[7],[2,3,8,9,12,19,20,21,27,28,29,36,39,40,45,46,52,53,57,60, \
#                     61,63,64,70,76,82,83,85,86,89,93,94]]

# ejemplo de 1 piso 2
```

```
#voxsolidoDatos=[[5],[1,2,3,5,6,8,9,10,14,15,16,18,19,21,22,23]]
```

Bibliografía

[1] Carlos Andújar. Octree-bases simplification of polyhedral solids. *Tesis Doctoral*, 1999.

[2] T.Y. Kong Azriel Rosenfeld, A.W. Roscoe. Concepts of digital topology. *Topology and its Applications*, 46:219–262, 1992.

[3] GNU GENERAL PUBLIC LICENSE

. http://www.gnu.org/licenses/gpl.txt. 2006.

[4] Python 3D Software Collection. http://www.vrplumber.com/py3d.py. 2007.

[5] Maria de la Luz Gasca Soto. Descomposición hamiltoniana e irreducibilidad en vox-sólidos. *Preprint Tesis de doctorado*, 2007.

[6] Juan Carlos Torres Francisco Velasco. Cells octree: a new data structure for volume modeling and visualization. *VISION, MODELING, AND VISUALIZATION*, 2001.

[7] J. Cagan K. Shimada and S. Yin. Geometric representations for intersection detection in intelligent packaging. *Notas de clase Computer-Aided Design, Carnegie Mellon univerity, Pitssburgh*, 1998.

[8] Aaron Knoll. A survey of octree volume rendering techniques. *GI Lecture Notes in Informatics, Annual IRTG Workshop,Germany*, 2006.

[9] Python Lenguaje. http://www.python.org. 2006.

[10] Ricardo Ramos Montero. Notas de apuntes de asignatura, informática gráfica. tema7. *Universidad de Ovieda, Asturias, España*, 2007-2008.

[11] Kuiyang L. Subramaniam J Natraj I., Yagnanarayanan K. and Karthik R. Reconfigurable 3d engineering shape search system. *ASME 2003 Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, 2003.

[12] Visual Python. http://www.vpython.org. 2006.

[13] Azriel Rosenfeld. Digital topology. *American Mathematical Monthly*, 86:621–630, 1979.