DIGITLAB

AMBIEMTE Y LENGUAJE PARA PROCESAMIENTO DE DIGITALIZACIONES.

Investigador principal:
OSCAR E. RUIZ SALGUERO.
Coordinador del Centro Interdisciplinario de Investigación (CII).

Asistente de investigación:
R. SEBASTIAN SCHRADER G.

Consultores en áreas específicas:
JUAN GUILLERMO LALINDE (área de Teoría de Compiladores).
CARLOS CADAVID (área de Topología Aplicada).

Trabajo realizado en:
CENTRO INTERDISCIPLINARIO DE INVESTIGACION CAD/CAM/CG.
UNIVERSIDAD EAFIT - (Medellín - Colombia).

DIVISION COGNITIVE COMPUTING & MEDICAL IMAGING
FRAUNHOFER INSTITUTE - (Darmstadt, Alemánia)

UNIVERSIDAD EAFIT

MEDELLÍN

1999

CONTENIDO

LISTA DE TABLAS

LISTA DE FIGURAS

LISTA DE ANEXOS

1. INTRODUCCION

Esta investigación presenta una serie de herramientas asincrónicas[1] para la recuperación de superficies a partir de digitalizaciones que presenten patrones planares (slices) o de rejilla (provenientes de *range pictures*). La decision de crear DigitLAB proviene de una aceptacion de que el estado de arte actual en recuperacion de formas es muy limitado y fragil cuando se intenta atacar todas las posibles gamas de digitalizaciones 3D, las cuales, ademas presentan defectos del muestreo fisico. Tomando un punto de partida eminentemente practico, se limito el alcance a digitalizaciones no aleatorias (ver arriba), y se proveyo un ambiente de preparacion de la digitalizacion, tanto como de ajuste de la superficie. Para lograr este objetivo, se entregaron funciones estadisticas, de filtrado, de soporte informatico y de razonamiento geometrico para las labores de nombrar, accesar y modificar partes selectivas de las digitalizaciones a procesar. Dichas herramientas deben poder aplicarse en cualquier vecindario de la digitalización para permitir un control en el ajuste de la superficie. Aunque en un principio DigitLAB se penso dirigida a digitalizaciones planares, se encontro que muchas de sus capacidades son aplicables a digitalizaciones caracteristicas de muestreos ópticos. Ello hizo que su alcance se ampliara, y es asi como se integró un modulo incipiente de procesamiento e integración de range pictures.

[1] No siguen una secuencia fija de aplicación.

La Figura 1.1 muestra el proceso básico de diseño y realización de un objeto(de izquierda a derecha). El formalismo de Geometría Constructiva de Sólidos, CSG, se usa como un metodo de creación por todos los modeladores geométricos actuales. Después de ello, se prefiere usar como representación el formalismo topológico B-Rep (Boundary Representation). La representacion CSG de un solido permite llegar a una B-Rep, y de ésta a una informacion granulada de su frontera (por medio de un muestreo virtual), la cual se traduce en una coleccion de puntos, vectores normales, tangentes, etc. (ver Fig. 1.1). Esta ultima informacion es requerida para la realizacion del objeto (por ejemplo en maquinado CNC). La aplicacion de digitalizaciones a la actividad de ingenieria inversa (Inverse Engineering) requiere una descripcion matematica compacta del objeto (B-rep), partiendo de un muestreo exhaustivo de puntos de su frontera (Figura 1.1 , flechas punteadas). Este proceso es extremadamente dificil porque (i) varios objetos pueden producir la misma digitalizacion, dado que el proceso de muestreo es discreto mientras que el objeto es continuo, (ii) un objeto puede tener varias digitalizaciones y (iii) la inferencia de ecuaciones analíticas de superficies a partir de un muestreo discreto de ellas es un proceso incierto.

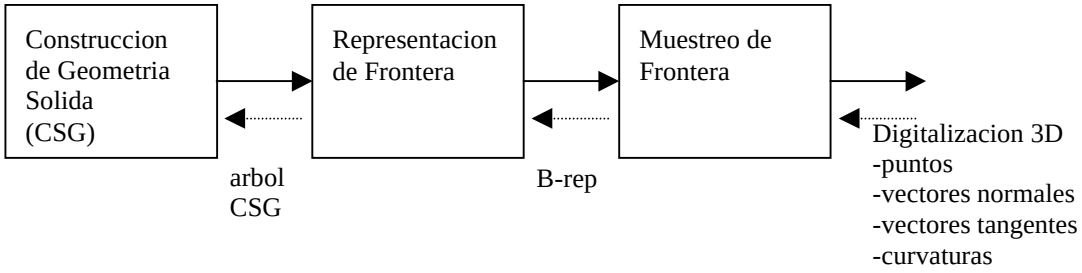


Fig.1.1. Transformacion de Esquemas de Representacion Geometrica.

DigitLAB es una ayuda para la transicion Digitalizacion->B-Rep. Los problemas remanentes, por ejemplo B-Rep->CSG, son parte de un panorama todavia incierto.

2. REVISIÓN BIBLIOGRÁFICA

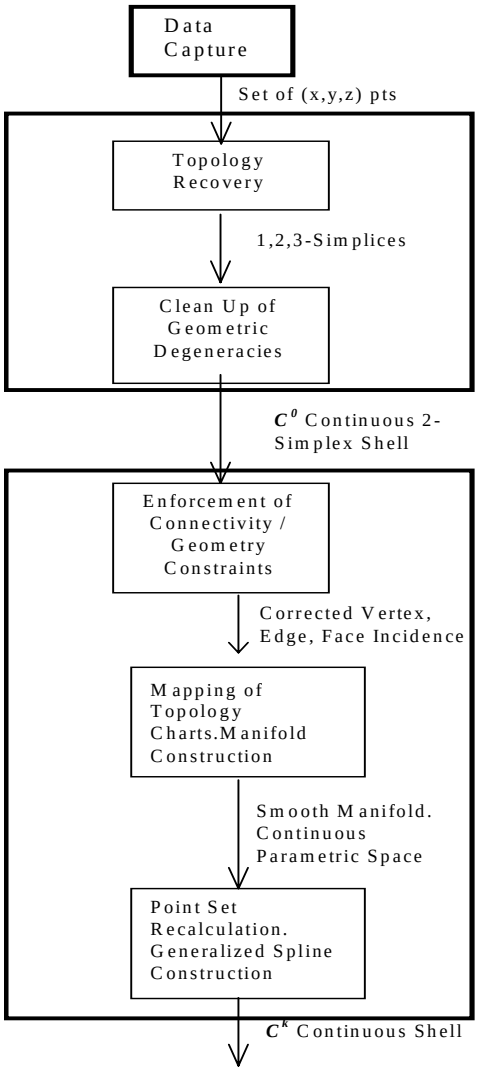


Fig 2.1: Computer Aided Geometric Design from 3D Digitizations.

Tomada de ([RL99])

La reconstrucción de superficies a partir de puntos incluye (Ver Fig. 2.1) la adquisición de datos, reconstrucción topológica, y aseguramiento de la continuidad, especifico para cada aplicación (maquinado CNC, prototipificación rápida, *rendering*, cálculo de las propiedades de masa, etc.). La geometría del objeto influye en la estrategia de muestreo, haciendo que algunos métodos de procesamiento de datos sean más eficientes que otros. De igual manera, la forma de los datos de salida (facetas vs. superficies paramétricas suaves, cascarones abiertos vs. cerrados, etc.) determinan el proceso de reconstrucción de la superficie. Esta revisión intenta resaltar los tópicos más notorios en estos aspectos.

En la literatura de reconstrucción de superficies el termino *topología* es usado con dos relacionados pero diferentes significados. En el contexto de Modelamiento Geométrico, *topología* se refiere a la información de conectividad gráfica que será el esqueleto de un modelo 2D o 3D (caras, ciclos y aristas), opuesto al significado en *geometría*, que provee las posiciones, dimensiones y formas (curvas, superficies y puntos) de las entidades topológicas ([Mor85] [Hof89]). En un contexto mas matemático, *topología* expresa las propiedades de los manifolds o superficies que localmente parecen (o pueden ser mapeados a) espacios Euclidianos (*charts*) mientras que su extensión total presenta arreglos complejos vía funciones de transición entre ellos ([GH95]). En este articulo, el término será usado en ambos sentidos, con el significado preciso dado por el contexto.

2.1. CAPTURA DE DATOS.

En la captura de un conjunto de puntos describiendo la superficie de un objeto (caja superior Fig. 2.1) hay (a) contacto vs. (b) métodos ópticos. Los métodos de contacto han alcanzado un nivel razonable de flexibilidad dada la posibilidad de utilizar una mesa con tres grados de libertad (dof) X-Y-Z o un brazo digitalizador con 5 dof. Aunque las mesas son rápidas y precisas, su aplicabilidad a objetos con formas cóncavas escondidas es limitada. Los brazos digitalizadores son menos precisos y lentos, pero permiten el muestreo de formas escondidas en el objeto dadas las configuraciones del brazo (similar a un brazo de dentista). Los métodos ópticos pueden ser pasivos o activos. Los métodos pasivos integran muchas vistas del objeto mediante al procesamiento de múltiples *range images*, mientras que los métodos activos siguen una luz sobre el objeto para extraer las coordenadas *(x,y,z)* de la superficie del objeto. En el primer caso, una gran cantidad de

fotos son cruzadas y procesadas usando sus traslapes para determinar su posición relativa e inferir las coordenadas *(x,y,z)* de la porción de superficie cubierta por la foto ([Neu97]). Las posiciones relativas de las fotos son determinadas como las que minimicen las diferencias entre los marcos. Aunque este método es caro en términos de recursos computacionales, tiene la capacidad de muestrear objetos muy grandes y cóncavos usando un conjunto suficiente de *range images*. Además permite muestrear objetos vivos aun cuando ellos no permanezcan en una posición rígida por un largo tiempo. Los métodos basados en la luz siguen un rayo que viaja sobre la superficie escaneada. La proyección generada por el rayo sobre la superficie en cada posición es detectada en una o mas vistas y sus coordenadas *(x,y,z)* son extraídas. El cubrimiento depende de la capacidad del rayo para moverse a lo largo de toda la superficie del objeto, y el arreglo de la cámara para grabar tal itinerario. Este método posee una complejidad computacional menor que el pasivo, pero es limitado cuando se muestrean formas cóncavas del objeto. Dado que los sujetos vivos tienen una dificultad en mantener una posición rígida es mas recomendable para objetos inertes ([BFB98] [PTR98]).

Otros métodos en el campo medico (CAT, MRI) implican la detección de tejidos del cuerpo basados en las diferencias que aparecen entre los huesos, músculos, órganos, etc. presentes bajo una excitación magnética o rayos-X. Sin embargo, en estos casos, el elemento básico capturado por el software medico es un *voxel* (pixel de volumen), por lo tanto, representa una información sólida y no de superficie ([Van92][DCH88]) manejada por una numeración espacial o técnicas de *octree* ([Sam95] [YS83]). De todas formas esto se sale del alcance contemplado por este proyecto. Si la información extraída del MRI o del CAT

es convertida a un muestreo de órganos o de contornos de huesos, el problema se convierte en un problema de reconstrucción de superficies, y es relevante a la presente discusión, como se explica mas adelante.

2.2. RECONSTRUCCION TOPOLÓGICA Y DE MANIFOLD.

Una vez que el conjunto de datos es obtenido (caja 1 Fig. 2.1), se procede a la reconstrucción de la superficie (caja 2) produciendo inicialmente un *shell* faceteado con continuidad C^0 ([BRV91]). El post-procesamiento (caja 3) es aplicado en algunos casos para alcanzar una continuidad C^2 suavizada de superficies paramétricas, o en otros casos para obtener superficies optimizadas C^0 para aplicaciones practicas (FEM, visualización, etc.) ([KV95]).

Dado que en muchos casos el patrón de muestreo introduce un orden en los datos, se posee particularmente un caso ventajoso, y los algoritmos generales explícitamente no toman ventaja de esto. Para recuperar la información de conectividad (caja 2 Fig. 2.1) algunas de las tácticas aplicadas son las *Alpha Shapes* ([Ede87][Ede94]), y los *Marching Cubes* ([LC87]). En forma de resumen, las Alpha Shapes establecen un conjunto de puntos cercanos unos de otros dados por un parámetro α. Cuando $\alpha = 0$, las *alpha shapes* son iguales al conjunto original de puntos. Cuando $\alpha = \infty$, las *alpha shapes* son el *convex hull* del conjunto de puntos dado. El usuario selecciona rangos intermedios de valores de α para recuperar la conectividad del conjunto de puntos en la forma de un *complejo simplicial* formado por 0,1,2 y 3 *símplices*. Dado que la salida de este paso es una malla

topológicamente correcta, degeneraciones geométricas (por ejemplo aristas colgadas) pueden estar presentes, y la colección de *símplices* es post-procesada en orden hasta dejar solo 2 *símplices* (dado que la meta intermedia es un *shell*) con una geometría aceptable ([Guo97]). El algoritmo de *Marching Cubes* construye una superficie faceteada que se aproxima a la superficie implícita en $\boldsymbol{R^3}(\boldsymbol{f(p) = 0})$, donde la función $\boldsymbol{f()}$ es inferida desde el muestreo de la digitalización.

La tercera etapa toma un *shell* continuo topológicamente correcto en $\boldsymbol{C^0}$ y lo cubre con *charts* de vértices, aristas y caras para obtener un mapeo completo entre el *shell* $\boldsymbol{C^0}$ y el *manifold* $\boldsymbol{M}$. Este mapeo hace posible definir una parametrización dependiente de los *charts* que produce una superficie continua en $\boldsymbol{C^2}$ excepto las áreas de los huecos (donde la superficie es completamente interrumpida). En la Fig. 2.1 el aseguramiento de la conectividad y las restricciones geométricas implica la re-calculación de los vértices cuyo grado de incidencia es muy alto o caras con muchos lados. Adicionalmente, cualquier manipulación de información de conectividad produce casi inmediatamente caras no planares, una corrección de la ruta ya recorrida de la información del punto es producida para restaurar caras planares con el correcto grado de incidencia ([CC78] [DS78]).

El resultado es una estructura topológica que ofrece condiciones correctas en los vértices, aristas y caras de la construcción de *charts*. La construcción de *charts* resulta en el cubrimiento de la superficie del *manifold* con parches que se traslapan en el espacio paramétrico, que ayudan a asegurar continuidad $\boldsymbol{C^2}$ de superficies B-spline generalizadas ([GH95]). La construcción de Splines generalizadas esta asistida en los cálculos por un conjunto de puntos “corregidos”, la cual es calculada como la menos desviada del modelo

de muestreo propuesto ([Guo97]). En esta referencia, ([Guo97]) se menciona la insuficiencia de forzar una continuidad $\boldsymbol{C^k}$ para garantizar los servicios del ajuste de la superficie, así como la necesidad de una distribución de curvatura y suavización.

2.3. TECNICAS DE ESTADISTICA Y MINIMIZACIÓN EN LOS PROCESOS DE DATOS DE PUNTOS.

La intervención de técnicas estadísticas en el procesamiento de conjuntos de puntos desde digitalizaciones típicamente actúa sobre: (i) formulación y prueba de las hipótesis de planaridad, cilindricidad, perpendicularidad, etc., con un muestreo de puntos *(x,y,z)*. (ii) convergencia de hipótesis verosímiles acerca de los puntos *(x,y,z)* físicos en la superficie desde un muestreo incompleto en información, (por ejemplo $(x_{picture}, y_{picture})$ de imágenes simultaneas), y (iii) creación de un conjunto alternativo de puntos a los *(x,y,z)* físicamente muestreados, satisfaciendo el criterio de representatividad del conjunto de puntos original. En el caso (i) los puntos *(x,y,z)* de la superficie muestreada son ensayados por sus factores de correlación con la geometría deseada ([Ans93] [Car95]). Las ecuaciones que aproximan son el *locus* de puntos minimizado por alguna función (mínimos cuadrados o valor absoluto) de las desviaciones de los puntos muestreados con respecto a las ecuaciones. Debe notarse que estas técnicas son vitales en cuestiones de tolerancias. En el caso (ii), ([Neu97]) se hacen suposiciones acerca de las transformaciones relativas entre las coordenadas $(x_{picture}, y_{picture})$ de las fotos traslapadas para recuperar las coordenadas *(x,y,z)* correspondientes a un pixel particular de las imágenes. El proceso involucra minimización de las desviaciones entre las coordenadas supuestas *(x,y,z)* afinando los mapas de

transformación entre las imágenes. En cada paso, una aproximación por mínimos cuadrados es aplicada para encontrar la mejor transformación entre las fotos ***i*** y ***j***. En el caso (iii), ([Guo97]) se calcula el mejor conjunto de puntos de control para que una superficie B-spline generalizada definida sobre un *manifold* difiera en lo mínimo al *alfa shapes* C^0 continuo del conjunto de puntos muestreado. ([HDD92]) presenta una técnica de faceteo que toma los grupos del conjunto de puntos ***k*** más cercano a la digitalización y encuentra su mejor (estadísticamente hablando) plano. Un proceso de cambio de sentido del vector normal en operaciones consecutivas es seguido a continuación para asegurar una superficie con curvatura consistente. Los planos son usados para aproximar el "conjunto cero" de la superficie ideal del *manifold* en R^3($f(p) = 0$). $Z(f)$ ($Z(f) = \{p \in R^3 \mid f(p)=0\}$).

Una vez que la función $f()$ ha sido aproximada por los planos, un algoritmo de *Marching Cubes* modificado es aplicado para obtener un cascaron C^0 continuo.

2.4. APLICACIÓN DE HEURISTICOS A LA RECONSTRUCCIÓN SUPERFICIAL.

En ([STT93]) una aproximación es presentada, comenzando con la suposición de que muchas superficies son formadas en un proceso natural que representan arreglos que minimizan una función potencial entre partículas formando la superficie. Tomando la derivada de la potencia inter-particulas, las fuerzas y torques actuando son calculadas y usadas para manejar la simulación de las secuencias de estados que un conjunto inicial de partículas seguiría si se dejase libre para evolucionar bajo la acción de los potenciales. El conjunto inicial es aumentado creando nuevas partículas cuando lugares "sin población" son detectados. Cada partícula lleva un sistema coordenado (posición y tres vectores

ortonormales siguiendo la regla de la mano derecha). Algunas partículas del conjunto son "bloqueadas" durante el proceso, significando que su sistema de coordenadas asociado se mantiene durante toda la evolución. El artículo específicamente no se ocupa de estimar los valores de las partículas bloqueadas, y dado que no es una meta en él, no se ocupa de la reconstrucción de superficies que no sean el resultado de procesos naturales (objetos manufacturados, por ejemplo).

2.5. CONCLUSIONES DE LA REVISION BIBLIOGRAFICA.

Para la literatura examinada, es evidente que las técnicas automáticas que son direccionadas para conseguir la superficie global (C^0, C^1 o C^2) no se preocupen de continuidades mezcladas, ruido local, zonas funcionales, o el uso de espacios locales paramétricos para la generación de códigos CNC ([Guo97]). De otra forma, como una suposición subyacente fundamental hecha por todos los autores es que las digitalizaciones tienen una rata característica de muestreo o densidad. Este muestreo característico influencia los parámetros de los algoritmos (α en [Ede94] , k en [HDD92]), los cuales cuantifican la vecindad espacial para recuperar la conectividad de los vértices. Sin embargo, dos puntos p y q pueden yacer en lados opuestos de objetos delgados estando espacialmente muy cerca. En este caso, todos los algoritmos evaluados los conectan. La causa de tal error es una deficiente digitalización dado que la distancia de muestreo no puede ser más grande que la mitad de la característica mas pequeña a ser capturada (frecuencia de Nyquist [Nyq28]). Una vez la digitalización es hecha con este criterio, se continúa con los métodos de corrección.

Es el sentimiento de los autores que al usuario se le debe permitir una aplicación asincrónica de las herramientas para el procesamiento de digitalizaciones. Esta capacidad implica poder efectuar tales operaciones sobre los datos sin seguir una secuencia rígida. Por ejemplo los filtros pueden ser aplicados antes o después de las herramientas para particionar o detectar esquinas. Obviamente los resultados de la aplicación van a depender del orden seguido, pero este orden es dinámicamente definido por el usuario dependiendo de la naturaleza de la digitalización que se tenga a la mano. Las herramientas desarrolladas y el sistema que garantiza la consistencia de los datos sobre su aplicación junto con travesías topológicas, recuperaciones y modificaciones - DigitLAB - son el principal sujeto de esta investigación.

3. GEOMETRÍA COMPUTACIONAL

En esta investigación se acude a herramientas de razonamiento geométrico para procesar y reducir un conjunto de puntos que preceden la aplicación de algoritmos de ajuste de superficies. Los procedimientos en el manejo de digitalizaciones se basan en las diferencias geométricas de los datos, que son esencialmente subjetivos puesto que dependen de la forma física del objeto.

Para la reconstrucción de superficies de digitalizaciones planares en tres dimensiones, la calidad del muestreo físico del objeto es factor indispensable para un buen desempeño en el proceso de la generación de la superficie. El procedimiento para realizar la digitalización planar de un cuerpo físico esta dado por los siguientes pasos: (a) Establecer un plano de digitalización (lo que restringe un grado de libertad al equipo de muestreo) (Ver Fig. 3.1). (b) Determinar una distancia δ entre niveles midiendo el detalle mas fino δ_m en la dirección normal al plano de digitalización (Ver Fig. 3.1) de acuerdo a la regla:

$$\delta \leq \delta_m / 2 \tag{1}$$

llamado periodo de Nyquist ([Nyq28]). (c) Determinar una distancia de digitalización δ' entre puntos consecutivos de un nivel para todos los niveles (Ver Fig. 3.1), así se determina el detalle mas fino para cada nivel asignándole a δ_m el menor de estos valores. La frecuencia mínima a la cual esta distancia debe ser muestreada este dada por la Ec. 1.

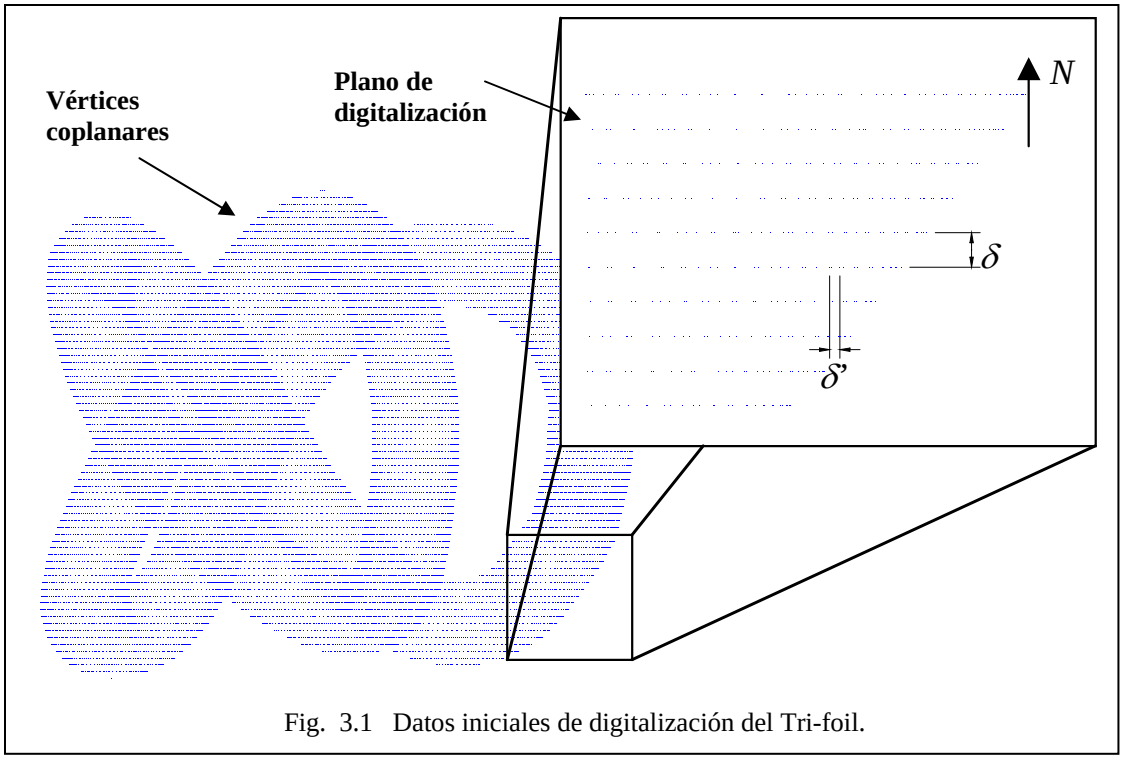


Fig. 3.1 Datos iniciales de digitalización del Tri-foil.

3.1. CLASIFICACIÓN POR NIVELES DE UN CONJUNTO DE PUNTOS.

Las digitalizaciones planares en tres dimensiones se pueden obtener de forma física (restringiendo un grado de libertad en el equipo de muestreo) o virtual (haciendo *scanning* por niveles o desde la representación CAD de un objeto). Por lo anterior el muestreo del modelo se realiza sobre trayectorias predefinidas, lo que permite obtener puntos confinados en planos aproximadamente paralelos (Ver Fig. 3.2). Dependiendo de la forma de la digitalización, la clasificación por niveles presenta dos tipos de aplicación: (i) Clasificación en niveles dados por la digitalización. (ii) Clasificación en niveles dados por el usuario (digitalizaciones aleatorias).

3.1.1. Clasificación en niveles determinados por la digitalización. Dado un conjunto de puntos en $\Re^3$, clasificarlos en una serie de niveles (*pockets*) ordenados consecutivamente en el espacio (Ver Fig. 3.2).

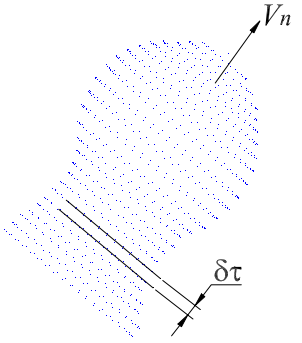


Fig. 3.2: Niveles dados por la digitalización.

Sea $S = \{p_0 = (x_0, y_o, z_0), p_1 = (x_1, y_1, z_1), ..., p_n = (x_n, y_n, z_n)\} \subset \Re^3$ un conjunto finito de puntos con las siguientes características: (i) La distribución de los puntos debe ser aproximadamente planar. (ii) Los planos o niveles deben ser aproximadamente paralelos y equidistantes. Denotemos por δ la distancia entre planos (vecinos) y por V_n el vector normal a los planos dado por:

$$V_n = (n_x, n_y, n_z) \Big/ \left\| (n_x, n_y, n_z) \right\| \qquad (2)$$

donde:

$$n_x = \sum_{i=0}^{i=n} (y_i - y_{i+1})(z_i + z_{i+1}) \;\; ; n_y = \sum_{i=0}^{i=n} (z_i - z_{i+1})(x_i + x_{i+1}) \;\; ; \; n_z = \sum_{i=0}^{i=n} (x_i - x_{i+1})(y_i + y_{i+1})$$

Denotemos por δ_τ la zona de tolerancia de cada nivel (Ver Fig. 3.3) como un espacio en $\Re^3$ por encima y por debajo de cada plano, la cual esta dada por:

$$\delta_\tau = \delta \cdot \tau \qquad \text{con} \quad 0 \le \tau \le 1 \tag{3}$$

donde τ es el grado de error por encima o por debajo de la distancia entre niveles δ.

Sea π_1 el plano que pasa por p_1 y $\perp$ a V_n, y $Q_1 = \{p \in S \mid dist(p, \pi_1) \le \delta_\tau / 2\}$. Ahora, sea $i_2 = min\{i \mid p_i \notin Q_1\}$ y π_2 el plano que pasa por p_{i2} y $\perp$ a V_n, aquí definimos $Q_2 = \{p \in S \mid dist(p, \pi_2) \le \delta_\tau / 2\}$. Sea $i_3 = min\{i \mid p_i \notin Q_1 \cup Q_2\}$ y π_3 el plano que pasa por p_{i3} y $\perp$ a V_n.

En general si ya conocemos $Q_1,...,Q_t$, entonces i_{t+1} será $min\{i \mid p_i \notin Q_1 \cup ... \cup Q_t\}$ y π_{t+1} será el plano que pasa por $p_{i_{t+1}}$ y $\perp$ a V_n.

Este proceso termina por la finitud de S y define conjuntos $Q_j \quad j = 1,...,m$ satisfaciendo:

$$S = \bigcup_{j=0}^{j=m} Q_j \tag{4}$$

Además, si τ fue escogido adecuadamente se debe cumplir que $Q_r \cap Q_s = \phi$ si $r \ne s$.

Así, el tipo de la partición es *{{ puntos }}* (Ver Tabla 3.1) y cada conjunto Q_j está formado por los puntos contenidos en el $j^{ésimo}$ plano de digitalización, con cada plano determinado por su punto origen $Q_{i[0]}$ ([RP97]).

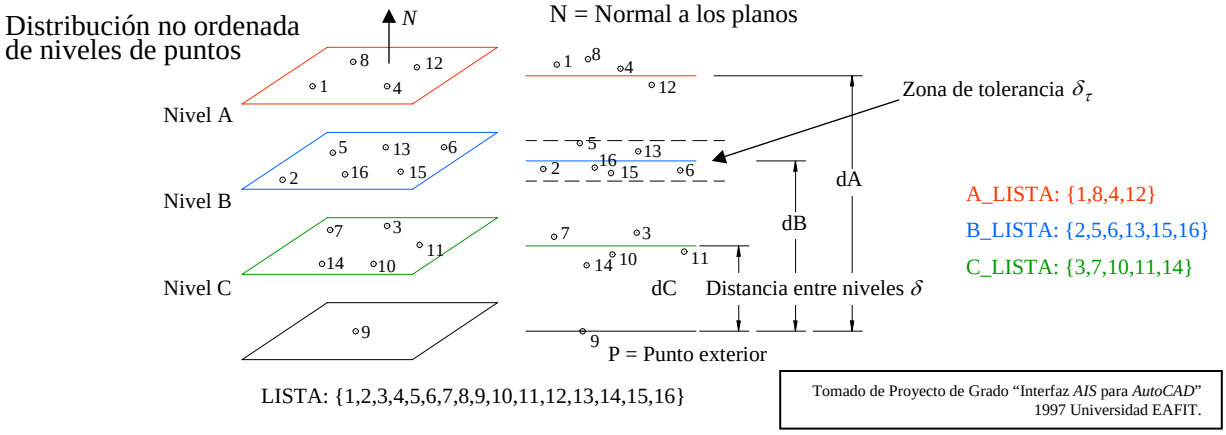


Fig. 3.3 Clasificación por Niveles (*pockets*) según la distancia δ y una tolerancia τ

Tabla 3.1 Funcionalidad de conjuntos de datos para funciones de preprocesamiento

Operación	Operandos		Resultado
Clasificación planar	{puntos}	û	{{puntos}}
Recuperación del polígono	{puntos}	û	[puntos]
Remuestreo por distancia	[puntos]	d	[puntos]
Remuestreo por número de intervalos	[puntos]	N	[puntos]
Filtrado	[puntos]	*f()*	[puntos]
Proyección planar de puntos	{puntos}	plano	{puntos}

Notación: { } representa un conjunto no estructurado de elementos, [ ] corresponde a una lista ordenada y {{ }} denota un conjunto de conjuntos.

3.1.2. Clasificación en niveles determinados por el usuario (digitalizaciones aleatorias).

Dado un conjunto de puntos en $\Re^3$, clasificarlos en una serie de niveles (*pockets* definidos por el usuario) consecutivamente en el espacio (Ver Fig. 3.4).

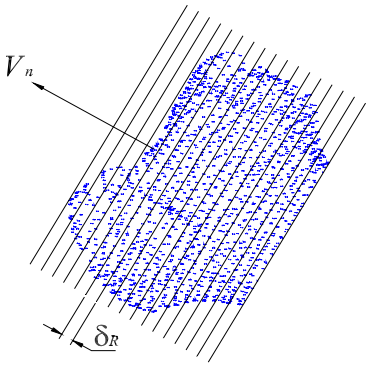


Fig. 3.4: Niveles dados por digitalizaciones aleatorias.

En este proceso, el conjunto de puntos $\boldsymbol{S} = \{ p_0, p_1, p_2, p_3, \ldots, p_n \}$ debe cumplir con las siguientes características: (i) la distribución de puntos no debe seguir ningún patrón. (ii) La

distancia entre puntos debe ser lo más equidistante posible. Esta clasificación se hace mediante la creación de planos paralelos $\Pi = \{ \pi_1, \pi_2, \pi_3, \ldots, \pi_m \}$, los cuales tienen a V_n como vector normal común y una distancia entre niveles δ que es determinada por el usuario.

El proceso comienza determinando dos puntos de S, uno en cada extremo en la dirección del vector normal V_n.

Mas precisamente, tenemos dos puntos $p_{i_{max}}$ y $p_{i_{min}}$ así:

$$p_{i_{max}} \in S \text{ tal que } p_{i_{max}} \cdot V_n \geq p_i \cdot V_n \quad \text{con} \quad i = 0,\ldots,n \tag{5}$$

$$p_{i_{min}} \in S \text{ tal que } p_{i_{min}} \cdot V_n \leq p_i \cdot V_n \quad \text{con } i = 0,\ldots,n. \tag{6}$$

A continuación definimos el número de planos m tal que:

$$m = \left\| \left(p_{i_{max}} - p_{i_{min}} \right) \cdot V_n / \delta \right\| \quad (\text{aquí } \|x\| \text{ con } x \in \Re \text{ denota la parte entera de } x) \tag{7}$$

Sea V_c un vector cuya magnitud es la zona de tolerancia (Ver Fig. 3.3) tal que;

$$V_c = \delta_\tau \cdot V_n \tag{8}$$

Denotamos a α_j como el punto origen del nivel m de tal forma que:

$$\alpha_j = p_{i_{min}} + j \cdot V_c \quad \text{con} \quad j = 0,..,m \tag{9}$$

y π_j el plano que pasa por α_j y $\perp$ a V_n.

Para el proceso de clasificación definimos conjuntos Q_j así:

$$Q_j = \left\{ p \in S \;\middle|\; dist(p, \pi_j) \leq \delta_\tau / 2 \right\} \quad \text{con} \quad 0 \leq j \leq m \tag{10}$$

(Notemos que si $0 \leq \tau < 1$ entonces $S = \bigcup_{j=0}^{j=m} Q_j$ es un subconjunto <u>propio</u> de S).

3.2. RECUPERACIÓN DE SECCIONES.

Partimos de un conjunto de puntos Q_j con $j = 0,...,m$ producto de cualquiera de los procesos descritos en la sección anterior. Recordemos que cada conjunto Q_j esta contenido en la vecindad de un plano y que no tiene un orden que corresponda al contorno que él sugiere.

A continuación daremos un procedimiento que ordena cada Q_j a su continuidad geométrica, donde es importante anotar que un Q_j puede formar más de un contorno.

Para la recuperación de estas secciones es necesario calcular la distancia real de digitalización δ_R por medio de:

$$\delta_R = \delta' \cdot \tau \tag{11}$$

donde τ es la tolerancia de error permisible de la digitalización. Notemos que si la digitalización no cumple con los requerimientos de muestreo los resultados arrojados no serán los esperados produciendo: (i) la inexistencia de las condiciones de continuidad de las secciones, que son dependientes de las características geométricas y (ii) la obligación de un postprocesamiento manual para usos sucesivo de las secciones.

La recuperación de contornos Q_j se hace por medio del criterio de la menor distancia entre puntos, con δ_R como parámetro de comparación, en términos más precisos:

Sea $Q_j = \{q_{j_0}, q_{j_1}, ..., q_{j_{s_j}}\}$, consideremos el elemento q_{j_0} (que por razones que serán evidentes en líneas siguientes denotaremos por q'_{j_0}) y tomamos el punto del conjunto $\{q \in Q_j \mid dist(q, q_{j_0}) \leq dist(q', q'_{j_0}), \forall\, q' \neq q_{j_0}\}$, con índice mas bajo de acuerdo a la

numeración en Q_j y lo llamamos q'_{j_1}. Si $dist(q'_{j_0}, q'_{j_1}) > \delta_R$, $\{q'_{j_0}\}$ es un contorno. Si $dist(q'_{j_0}, q'_{j_1}) \leq \delta_R$ entonces tomamos el elemento con menor índice del conjunto $\{q \in Q_j \mid dist(q, q'_{j_1}) \leq dist(q', q'_{j_1}), \forall\, q' \neq q'_{j_0}, q'_{j_1}\}$ y lo llamamos q'_{j_2}. Si $dist(q'_{j1}, q'_{j_2}) > \delta_R$, $\{q'_{j_0}, q'_{j_1}\}$ es un contorno. En caso contrario tomamos el elemento con menor índice del conjunto $\{q \in Q_j \mid dist(q, q'_{j_2}) \leq dist(q', q'_{j_2}), \forall\, q' \neq q'_{j_0}, q'_{j_1}, q'_{j_2}\}$.

En general, supongamos que ya tenemos $q'_{j_0}, ..., q'_{j_t}$, si $dist(q'_{j_t}, q'_{j_{t-1}}) > \delta_R$, entonces $\{q'_{j_o}, ..., q'_{j_{t-1}}\}$ forman un contorno. Si $dist(q'_{j_t}, q'_{j_{t-1}}) \leq \delta_R$, entonces tomamos el punto con índice menor del siguiente conjunto:

$\{q \in Q_j \mid dist(q, q'_{j_t}) \leq dist(q', q'_{j_t}), \forall\, q' \neq q'_{j_0}, ..., q'_{j_t}\}$, y lo llamaremos $q'_{j_{t+1}}$.

Siempre que se forma un contorno se aplica este proceso arrancando con el punto de índice menor (de acuerdo a la numeración de Q_j) y que no pertenezca a ninguno de los contornos ya formados (Ver Fig. 3.5 y 3.6).

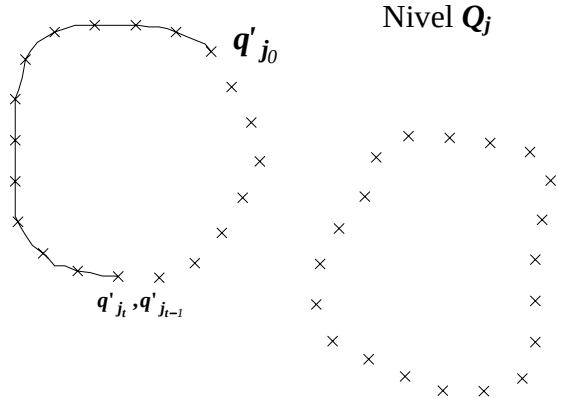


Fig 3.5. Recuperación de contornos

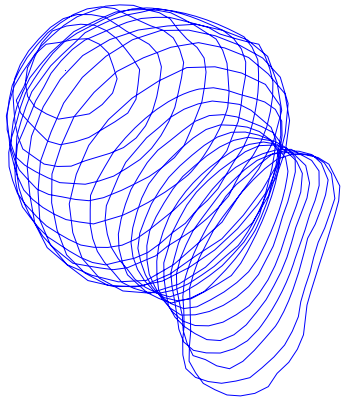


Fig. 3.6: Contornos ya recuperados de la cabeza de un fémur.

3.3. MAPEO INTER – NIVELES.

En esta sección nos proponemos exhibir el método que mapea los contornos de niveles consecutivos.

Sean Q y Q' dos niveles consecutivos y $C_1,...C_n$ los contornos de Q y $C'_1,...C'_n$ los contornos de Q'. Definimos la función de distancia ente dos contornos C y C' en Q y Q' respectivamente por la formula usual:

$$dist(C,C') = min\{dist(p,q) \mid p \in C \wedge q \in C'\} \tag{12}$$

(aquí $dist(p,q)$ es simplemente la distancia Euclidea en $\Re_3$).

Ahora tomamos C_1 y vemos para que índice $j_1 \in \{1,...,n\}$ más bajo tal que :

$$dist(C_0, C'_{j_0}) \leq dist(C_0, C'_{j_k}) \quad \text{con } k = 1,...,n \tag{13}$$

El método unirá a C_0 con C'_{j_0} y de manera análoga se apareará el resto de los contornos. Este apareamiento queda codificado en una remuneración $C'_{j_0},...,C'_{j_n}$ de $C'_0,...,C'_n$.

Pasamos ahora a describir como se mapean los contornos C_0 y C'_{j0}, que por comodidad llamamos C y C' respectivamente. El procedimiento para mapear el resto de los pares es exactamente el mismo.

El proceso de mapeo de niveles presenta dos casos: (i) Mapeo de niveles cuando $n = m$. (ii) Mapeo de niveles cuando $n \neq m$ (m y n son los números de contornos por cada par de niveles consecutivos). Los contornos por recuperar deben poseer (i) el sentido de generación de los contornos igual (vector normal de los contornos de igual sentido y una

dirección aproximadamente igual) y (ii) la distancia entre los puntos iniciales de cada contorno debe ser la menor.

Sean ***p*** y ***p'*** elementos de ***C*** y ***C'*** respectivamente tales que $dist(p, p') = dist(C, C')$. Renumeramos los contornos ***C*** y ***C'*** de tal manera que sus primeros elementos son ***p*** y ***p'*** respectivamente y siguiendo el sentido de rotación escogido al conjunto de todo el proceso de mapeo de niveles.

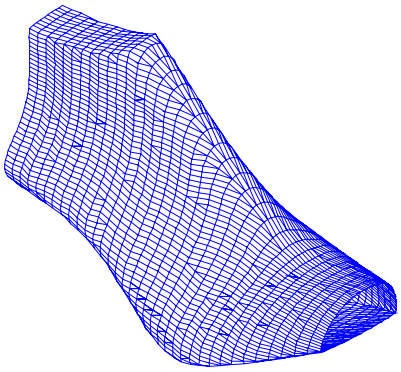


Fig. 3.7: Recuperación de niveles de la horma *m = n.*

3.3.1. Recuperación de niveles cuando *m = n*. Sean un par de niveles consecutivos ***C*** y ***C'***, cada uno en un número diferente de contornos (Ver Fig. 3.7).

Sean $C = \{p_0 = p, p_1, ..., p_r\}$ y $C' = \{p'_0 = p', p'_1, ..., p'_s\}$ los cuales están numerados y definimos una secuencia de parejas $R = \langle (p_{j_0}, p'_{k_0}), (p_{j_1}, p'_{k_1}), ..., (p_{j_t} = p_r, p'_{k_t} = p'_s) \rangle$ en CxC' con $j_0 \le j_1 \le ... \le j_t$ y $k_0 \le k_1 \le ... \le k_t$ por la siguiente regla de recurrencia:

(i) $j_o = 0$ y $k_o = 0$

(ii) Supongamos que ya hemos definido las primeras $(l+1)$ parejas de R. Sea k el numero menor tal que $k \geq l$ y $dist(p'_k, p_{j_{l+1}}) \leq dist(p'_d, p_{j_{l+1}})$ con $d \geq k_l$

Sea j el numero menor tal que $j \geq l$ y $dist(p_j, p'_{k_{l+1}}) \leq dist(p_e, p'_{k_{l+1}})$ con $e \geq k_l$.

(iii) Si $k = j_l + 1$ o $j = k_l + 1$ entonces $(p_{j_l+1}, p'_{k_l+1}) = (p_j, p'_k)$ o

Si $k = j_l$ o $j = k_l$ y además $dist(p_{j_l+1}, p'_k) \leq dist(p'_{k_l+1}, p_j)$ entonces $(p_{j_{l+1}}, p'_{k_{l+1}}) = (p_{j_l+1}, p'_k)$. En caso contrario $(p_{j_{l+1}}, p'_{k_{l+1}}) = (p_j, p'_{k_l+1})$.

(iv) Si $k = j_l$ y $j \neq k_l$ entonces $(p_{j_{l+1}}, p'_{k_{l+1}}) = (p_{j_l+1}, p'_k)$.

Si $k \neq j_l$ y $j = k_l$ entonces $(p_{j_{l+1}}, p'_{k_{l+1}}) = (p_j, p'_{k_l+1})$.

El resultado de esta aplicación será una asociación finita R de parejas de puntos pertenecientes a los vértices de la superficie recuperada.

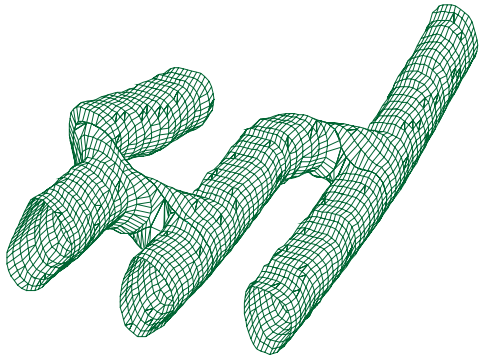


Fig. 3.8: Recuperación de niveles de los túneles $m \neq n$.

3.3.2. Recuperación de niveles cuando m ≠ n. Sean un par de niveles consecutivos P_k y P_{k-1}, cada uno con un número diferente de contornos (Ver Fig. 3.8). Definimos la distancia máxima δ_m permisible con la cual se seleccionan los contornos así:

$$\delta_m = \delta' \cdot \tau \tag{14}$$

En este caso, donde $m \neq n$, el mapeo superficial no es completamente cerrado entre los contornos y la reconstrucción se efectúa desde el contorno P con menor número de vértices n hacia el contorno Q mayor número de vértices m, donde $n > m$, así:

$$P = \{p_0, p_1, p_2, ..., p_n\} \quad \wedge \quad Q = \{q_0, q_1, q_2, ..., q_m\} \tag{15}$$

Sea una asociación finita S de parejas de puntos pertenecientes a los vértices de la superficie recuperada, dada por:

$$S = \text{sucesión finita } [S_d] \text{ donde } S_d = (p_i, q_j) \in P \times Q \qquad \text{siendo } S \subseteq P \times Q \tag{16}$$

El proceso se efectúa consecutivamente vértice a vértice siempre que se cumpla que la distancia entre vértices no supere a δ_m, así:

Sea $p_i \in P \quad \wedge \quad q_j \in Q$ entonces:

$$S_d = (p_i, q_j) \quad iff \tag{17}$$

$$\forall_k \quad \left[0 \leq k \leq m \Rightarrow \exists_j \left((\delta_c = dist(p_i, q_j) \leq dist(p_i, q_k)) \wedge \delta_c \leq \delta_m \right) \right]$$

Si no existe ningún q_j que cumpla con lo anterior, el vértice compañero se crea virtualmente, teniendo como base el último q_j hallado. Este proceso de creación virtual está dado por las siguientes etapas:

(i) Se crea un sistema de coordenadas Si con el último P_i que obtuvo compañero, así:

$$Si = [V_{xi}, V_{yi}, V_{zi}], \quad \text{con } V_{xi} = P_{a(d+1)} - P_{ad} \; ; \; V_{yi} = V_n \; ; \; V_{zi} = V_{xi} \times V_{yi} \tag{18}$$

(ii) Se crea un sistema de coordenadas Sj en el vector que no obtuvo compañero, así:

$$\boldsymbol{Sj}=[V_{xj,}V_{yj},V_{zj}], \quad \text{con } V_{xj} = P_{a(d+2)} - P_{a(d+1)} \; ; \; V_{yj} = V_n \; ; \; V_{zj} = V_{xj} \boldsymbol{x} V_{yj} \tag{19}$$

(iii) Se crea una matriz de traslación y rotación ***M*** entre ***Si*** y ***Sj***.

(iv) Se aplica ***M*** al último punto compañero encontrado en ***Q***, calculando el punto virtual para $\boldsymbol{p_i}$, así:

$$\boldsymbol{q_i = M \cdot q_{bd}} \tag{20}$$

3.4. TRIANGULACIÓN DE POLÍGONOS.

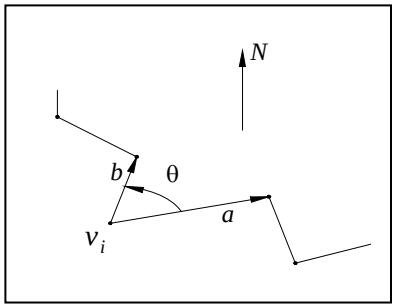
Terminado el proceso de recuperación de niveles, la superficie resultante puede presentar sectores no recuperados (Ver Fig. 3.9 y 3.10). Estos vacíos se presentan en la recuperación superficial de (i) contornos iniciales o finales donde no existe un contorno compañero del siguiente nivel para la recuperación superficial, y de (ii) niveles con diferente número de contornos, donde la recuperación no utiliza todos los vértices de los contornos por unir.

3.4.1. Contornos iniciales o finales. Estos vacíos o "huecos" se presentan en los casos donde la secuencia de recuperación de la superficie se finaliza o interrumpe. Para las caras donde se requiera una superficie totalmente cerrada del modelo (cascarón), estos vacíos deben ser recuperados por medio de una triangularización.

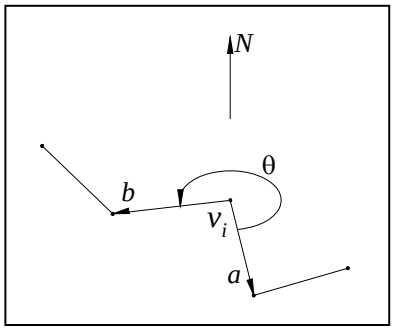
La triangularización de polígonos es una operación fundamental en la geometría computacional y ha obtenido múltiples desarrollos en los últimos años. El procedimiento aquí descrito presenta como resultado final una triangularización topológicamente correcta para un polígono ***P*** dado. Una colección de triángulos se llama *topológicamente correcta* para ***P*** si: (i) Los vértices de ***P*** fueran los vértices de los triángulos. (ii) Cada arista de cada

triángulo hace parte de exactamente una arista de otro triángulo o es una arista de ***P***. (iii) Cada arista de ***P*** pertenece solamente a un triángulo de frontera.

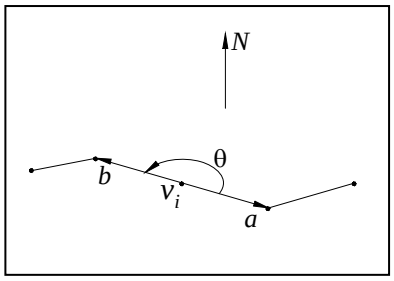
Primero se supone que ***P*** es un polígono en tres dimensiones no autointersectante, cuyos vértices se denotan con $\boldsymbol{V = [v_0, v_1, v_2, ..., v_n]}$ y se describe un vector normal $\boldsymbol{V_n}$ dado por la orientación de ***P*** . De ([Hel99]), un vértice $\boldsymbol{V_i \in P}$ se denomina:



(i). Convexo si: $\theta < 180^\circ$ (21)



(ii). Reflexivo si: $\theta > 180^\circ$ (22)



(iii). Reflexivo Tangencial si: $\theta = 180^\circ$ (23)

Donde $\boldsymbol{a = (V_{i+1} - V_i)}$ y $\boldsymbol{b = (V_{i-1} - V_i)}$

Dos vértices $V_i \neq V_k$ forman una diagonal si el segmento $[V_i, V_k]$ está completamente contenido en el interior de P. Cada vértice convexo V_i está contenido en un triángulo $t_i(V_{i-1}, V_i, V_{i+1})$ tal que:

$$\forall \quad V_i \left(0 \leq i \leq n \quad \Rightarrow \quad \exists\, t_i(V_{i-1}, V_i, V_{i+1})\right) \qquad \textit{iff}$$

(i) $(V_{i-1}) \wedge (V_{i+1})$ sean cóncavos $\vee$

(ii) (V_{i-1}) sea cóncavo $\wedge$ $(\alpha_i > \alpha_2)$ $\vee$

donde $\alpha_1 = angl(V_a, V_b) \quad \wedge \quad \alpha_2 = angl(V_a, V_c)$

con $V_a = (V_i - V_{i-1}), \quad V_b = (V_{i-1}, V_{i-2}) \quad \wedge \quad V_c = (V_{i+1}, V_{i-1})$

(iii) (V_{i+1}) sea cóncavo $\wedge$ $(\alpha_3 > \alpha_4)$ $\vee$ (24)

donde $\alpha_3 = angl(V_a, V_b) \quad \wedge \quad \alpha_4 = angl(V_c, V_b)$

con $V_a = (V_{i+2} - V_{i+1}), \quad V_b = (V_i, V_{i+1}) \quad \wedge \quad V_c = (V_{i-1}, V_{i+1})$

(iv) $(\alpha_1 > \alpha_2) \quad \wedge \quad (\alpha_3 > \alpha_4) \quad \wedge$

(v) El segmento $[V_{i-1}, V_{i+1}]$ forme una diagonal de P.

A los triángulos $t_i(V_{i-1}, V_{i+1})$, que cumplan con la anterior propiedad, se les aplica el factor de forma angular seleccionando el triángulo con mayor factor, así:

$$fact\_ang(t_{i\_max}) \quad \geq \quad fact\_ang(t_i) \tag{25}$$

El triángulo $t_{i\_max}(V_{i\_max-1}, V_{i\_max}, V_{i\_max+1})$ se adiciona a una lista de resultante de triángulos $T = [t_0, t_1, t_2, ..., t_k]$ y el vértice V_{i_max} se sustrae de la lista de vértices $V \in P$.

Este proceso se realiza hasta que la lista V contenga sólo dos elementos, por lo cual la lista T contiene los triángulos correspondientes a la triangularización básica.

Se dice que el resultado obtenido en $T = [t_0, t_1, t_2, ..., t_k]$ es una triangulización básica porque puede existir un triángulo t_m que se tome como polígono P_t y sea triangularizado. Esto se da por medio de las longitudes de las aristas δ_l del polígono original P así:

$$\exists \quad P_t \quad \left\{ \begin{array}{l} |segm[V_0, V_1] > 2 \cdot \delta_l \quad \vee \quad |segm[V_1, V_2] > 2 \cdot \delta_l \quad \vee \\ |segm[V_2, V_0] > 2 \cdot \delta_l \end{array} \right\} \qquad (26)$$

Si P_t existe, se aplica un remuestreo por número de puntos δ_l y se aplica todo el proceso de triangularización hasta obtener el resultado deseado (Ver Fig. 3.9).

3.4.2. Niveles con diferentes número de contornos. La superficie F resultante para este caso se produce desde el contorno $P_i = \{p_0, p_1, p_2, ..., p_n\}$ hacia el contorno $Q_j = \{q_0, q_1, q_2, ..., q_m\}$, donde $m > n$ (Ver 2.3.2.).

Sea $V_i = \{v_0, v_1, v_2, ..., v_k\}$, con $k = 2n$, la lista de vectores F, y $R_i = \{r_0, r_1, r_2, ..., r_n\}$ la lista de vectores compañeros de P_i, donde $R_i = (V_i - P_i)$. La lista de los contornos R_i se une con el contorno Q_j para formar un solo contorno, teniendo como única condición que el sentido de generación de los contornos R_{ij} sea opuesto al sentido generado de Q_j. A este polígono resultante se le hace una detección de islas para obtener los contornos que representen los vacíos o "huecos" de una recuperación de niveles con un número diferente de contornos (Ver Fig. 3.10). Cada isla resultante se toma como un polígono P de n lados, al cual se le aplica el proceso de triangularización aquí descrito. Como resultado de esta aplicación se obtiene una superficie cerrada o "cascarón", la cual puede usarse para diferentes propósitos investigativos; por ejemplo *FEA* (*Finite Element Analysis*).

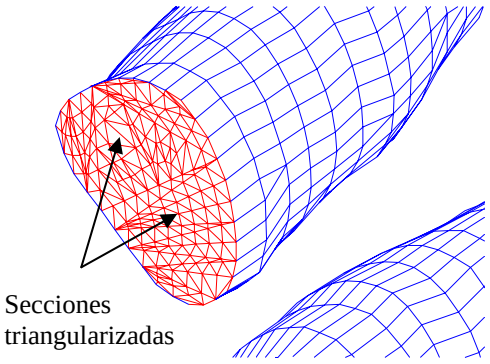


Fig. 3.9: Contornos iniciales o finales

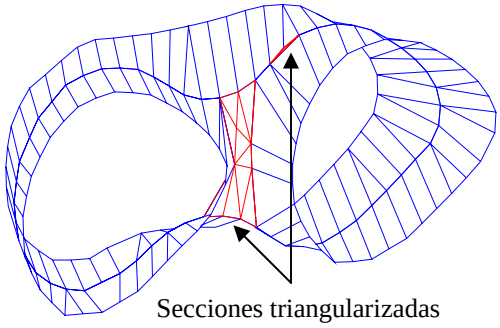


Fig. 3.10: Niveles con diferentes número de contornos

3.5. REMUESTREO DE CONTORNOS.

El remuesteo se usa para generar digitalizaciones artificiales con parámetros de muestreo diferentes de los de la digitalización física original. Los diferentes tipos implementados son: (i) Remuestreo por distancia. (ii) Remuestreo por número de puntos.

3.5.1. Remuestreo por distancia. Dado un polígono ***P*** en E^3, y una distancia ***d,*** se entrega el polígono ***P'*** resultante de remuestrear la frontera de ***P*** con segmentos de longitud ***d*** (Ver Fig. 3.11).

3.5.2. Remuestreo por número de puntos. Dado un polígono ***P*** en E^3, y un entero ***N***, se retorna el polígono ***P'***, formado por los vértices de ***P*** muestreados cada ***N*** puntos (Ver Fig. 3.12).

3.6. FILTRADO DE CONTORNOS.

El filtrado es el proceso por el cual una función ***f( )*** se aplica a una subsecuencia ***S***=[$p_i, p_{i+1}, p_{i+2}, \ldots p_{i+W-1}$] de ***W*** vértices de un polígono ***P***, lo que entrega vértices q_i para formar otra subsecuencia ***S'***=[$q_m, q_{m+1}, q_{m+2}, \ldots q_h$]. Los puntos de ***S'*** no están necesariamente en ***S***, y ***S'*** puede tener menos puntos que ***S***, dependiendo del manejo sobre la ventana ***W***. La función de filtro usada en esta investigación fue:

$$\vec{q}_i = f(\vec{p}_i \ldots \vec{p}_{i+W-1}) = \sum_{j=1}^{j=W} \alpha_j \cdot \vec{p}_{j+i-1} \text{ con } \sum_{j=1}^{j=W} \alpha_j = 1.0 \ , \ \alpha_j \geq 0 \ , \ \mathbf{j=1..W} \qquad (27)$$

con $\alpha_1 = \alpha_2 = \ldots \alpha_N = 1/W$ para un promedio, y $\alpha_1 = 1, \alpha_2 = \alpha_3 = \ldots . \alpha_W = 0$ para re-muestreo (Ver Fig. 3.13).

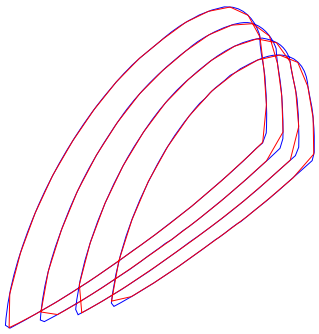


Fig. 3.11: Remuestreo por Distancia. D = 5

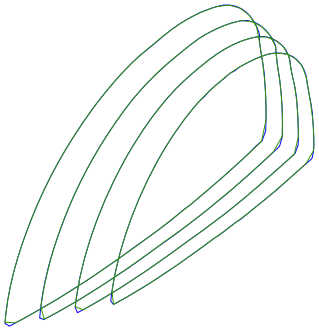


Fig. 3.12: Remuestreo por número de nodos. N = 50

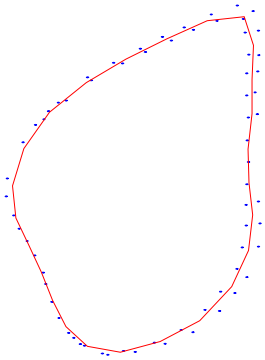


Fig. 3.13: Filtrado de contornos

4. DISEÑO E IMPLEMENTACIÓN DEL LENGUAJE

4.1. INTRODUCCIÓN.

En los lenguajes de programación se habla normalmente de compiladores e interpretadores. Los compiladores toman un código fuente, lo analizan y generan un código objeto. El código objeto, luego de ser encadenado adecuadamente, puede ser ejecutado como un programa independiente. Los interpretadores toman un código fuente, lo analizan y lo ejecutan.

Sin embargo, la distinción entre compilador e interpretador no es muy clara, en primer lugar, las técnicas que se utilizan para construir compiladores e interpretadores son muy similares, y en segundo lugar, los interpretadores modernos pueden ser mirados como compiladores que generan códigos intermedios (en el caso de *java* se denomina *bytecode*) y ese código se ejecuta después sobre una máquina virtual.

En el caso de *DigitLAB* (cuya gramática aparece en el Anexo 3), el lenguaje es interpretado pero utilizando técnicas modernas de compilación, de esta manera se evita el análisis repetitivo de código en los ciclos y se aproxima más al concepto de código intermedio para ser ejecutado en una máquina virtual.

La idea central es ésta: el proceso de compilación genera una lista postfija implementada en *LPOM (Listas Permanentes de Objetos Múltiples)* (Ver Anexo 5). Lo que en otros ambientes se podría asociar con la máquina virtual en *DigitLAB* es la interpretación de dicha lista. Por esta razón el lenguaje y los pasos involucrados en el funcionamiento del lenguaje se describen utilizando las convenciones y estándares del desarrollo de compiladores.

El resultado final de la compilación es una lista de objetos genéricos donde el código intermedio se genera directamente a partir del análisis sintáctico. La detección de los errores semánticos se hace por medio de la verificación que un operador pueda aplicar a un objeto y como el ambiente es interactivo, el usuario recibe notificación del error inmediatamente.

4.2. DISEÑO E IMPLEMENTACIÓN.

DigitLAB esta diseñado con un lenguaje capaz de describir las operaciones básicas aplicadas comúnmente a un proceso de digitalización, permitiendo una acción sintonizada con las condiciones de cada digitalización, y, por lo tanto, es capaz de producir estructuras de datos utilizables con mayor probabilidad de éxito en etapas de manufactura subsecuentes.

Se diseñó un sistema de entrada /salida de datos a través de un lenguaje de programación, con el cual (i) los procedimientos se ejecutan por el llamado de funciones y (ii) las

entidades y variables de la aplicación con su información requerida se aplican como parámetros por valor de esas funciones.

Para la definición del lenguaje de programación para *DigitLAB* se elaboró la descripción de (i) la *sintaxis* (aspecto del programa) y (ii) la *semántica* (significado del programa) del lenguaje. En la especificación del lenguaje se utilizo la notación *BNF*[1] que utiliza una técnica de compilación orientada a la gramática conocida como *traducción dirigida por la sintaxis.* La generación de un compilador está conformada por las siguientes etapas: (i) Tabla de símbolos. (ii) Análisis léxico. (iii) Estructura de la gramática. (iv) Análisis sintáctico. (v) Verificación de tipos.

4.2.1. Implementación de la tabla de símbolos. La tabla de símbolos es la estructura de datos que permite almacenar información sobre las diversas construcciones del lenguaje fuente. Cada entrada de dicha tabla se implementa como un registro compuesto por una secuencia de caracteres consecutivos que corresponde a la declaración de un nombre. Para el análisis léxico deben declararse las palabras claves o reservadas. El analizador léxico buscará entonces, dentro de las secuencias de letras y dígitos contenidos en la tabla de símbolos, dichas palabras claves o reservadas.

4.2.2. Análisis léxico. El análisis léxico (cuyos componentes aparecen en el Anexo 2), recibe como entrada un programa fuente compuesto por caracteres de un alfabeto básico. Su misión consiste en separar las diversas “palabras” que conforman el programa fuente

clasificando cada una de ellas en constantes, identificadores, palabras reservadas, operadores y separadores generados mediante diccionario. A partir de ellos genera una salida, que no es más que una secuencia de componentes léxicos, que utilizará más tarde el analizador sintáctico.

La especificación del analizador léxico de *DigitLAB* se elaboró por medio de un lenguaje de patrón-acción llamado *Flex*[2] ([Rei92]), que es una herramienta generada en *C* para construir analizadores léxicos a partir de notaciones de propósito especial (examinador léxico secuencial), es decir, programas que leen un archivo fuente y van identificando *tokens* (unidades léxicas o palabras), lo cual permite la definición de expresiones regulares y asociar a cada una de ellas una acción que se ejecutará cuando se reciba una secuencia de caracteres que se ajusten a una expresión regular. Las acciones asociadas a cada categoría se responsabilizan de colocar la información adecuada en la tabla de símbolos y pasarle al *parser* (analizador sintáctico) la descripción del *token* que fue identificado para su procesamiento.

Un analizador léxico hecho en *Flex* consta de tres partes (Ver Anexo 2):

(i) Declaraciones, donde se incluyen declaraciones de variables e identificadores que se declaran para representar una constante, y definiciones regulares, que son proposiciones que se utilizan como componentes de las expresiones que aparecen en las reglas de traducción (Ver Tabla 4.1).

---

[1] Forma de Backus-Naur, notación de gramáticas independientes del contexto.

smbl_mtchar	[\t\n]
smbl_number	[0-9]
smbl_letter	[a-zA-Z_]
smbl_character	[-_a-zA-Z]
empty_in	{smbl_mtchar}+
integer_in	{smbl_number}+
real_in	{smbl_number}+(\.{smbl_number}+)?([E|e][+|-]?{smbl_number}+)?+
word_in	({smbl_letter}+{smbl_number}*)+
char_in	(\'{smbl_character}\')
string_in	(\")({smbl_letter}\:\\)?({word_in}(\\)?)+(\.{word_in})?(\")

Tabla 4.1. Declaraciones del analizador léxico (Ver anexo 2).

(ii) Reglas de traducción, donde se incluyen proposiciones de la forma p_n *{ acción* $_n$ *},* aquí p_i es una expresión regular y cada *acción*$_i$ es un fragmento de programa que describe cuál ha de ser la acción del analizador léxico cuando el patrón p_i concuerda con una palabra clave o reservada (Ver Tabla 4.2).

{empty_in}	{ /*Do nothing*/ }
"("	{ strcpy(yylval.union_text,yytext); return TK_LEFT_PAR;}
"]"	{ strcpy(yylval.union_text,yytext); return TK_RIGHT_BRK;}
"}"	{ strcpy(yylval.union_text,yytext); return TK_RIGHT_KEY;}
","	{ strcpy(yylval.union_text,yytext); return TK_COMMA;}

"pocketing"	{ strcpy(yylval.union_text,yytext); return TK_pocketing;}
"build_coplanar"	{ strcpy(yylval.union_text,yytext); return TK_build_coplanar;}

Tabla 4.2: Algunas reglas del analizador léxico (Ver anexo 2).

[2] *Flex*. Reproducción libre de programa *Lex* desarrollado por la *GNU* (*Copyright (C) 1989, 1991 Free Software Foundation, Inc.675 Mass Ave, Cambridge, MA 02139, USA*.)

" build_resample "	{ strcpy(yylval.union_text,yytext); return TK_build_resample;}
" build_levels "	{ strcpy(yylval.union_text,yytext); return TK_build_levels;}
"link_levels"	{ strcpy(yylval.union_text,yytext); return TK_link_levels;}

{char_in}	{ strcpy(yylval.union_text,yytext); return TK_CHAR;}
{string_in}	{ strcpy(yylval.union_text,yytext); return TK_STRING;}
{word_in}	{ strcpy(yylval.union_text,yytext); return TK_NOUN;}
{integer_in}	{ strcpy(yylval.union_text,yytext); return TK_INTEGER;}
{real_in}	{ strcpy(yylval.union_text,yytext); return TK_REAL;}

Tabla 4.2: Algunas reglas del analizador léxico (Cont.).

(iii) Procedimientos auxiliares, donde se implementan todas las funciones adicionales que se puedan necesitar (Ver Tabla 4.3).

Ejecución del *Parser*	int yywrap(void)
Aplicación de los códigos de error	int yyerror(char *messg)

Tabla 4.3: Procedimientos auxiliares del analizador léxico (Ver anexo 2).

4.2.3. Estructura de la gramática. Una gramática es una serie de reglas que el compilador utiliza para reconocer si una entrada es sintácticamente correcta (Ver Anexo 3). Dado que se ha dividido el compilador en dos partes, la primera de ellas el analizador léxico y la segunda el analizador sintáctico (*parser*), es posible establecer los fundamentos de una gramática teniendo en cuenta las limitaciones propias de las entradas de caracteres, así como también la tabla de símbolos predefinida y las salidas del analizador sintáctico.

Para construir una gramática adecuada se deben analizar las relaciones entre asociatividad y precedencia de operadores, y, a partir de éstas, formular las reglas generales de la gramática.

Los componentes que posee una gramática son (Ver Tabla 4.5):

1. Un conjunto de componentes léxicos, denominados símbolos terminales.
2. Un conjunto de símbolos no-terminales.
3. Un conjunto de producciones, donde cada producción consta de: (i) Un no-terminal, llamando lado izquierdo y (ii) Una secuencia de componentes léxicos y no terminales, o ambos, llamado lado derecho de la producción.
4. La denominación de uno de los no terminales como símbolo inicial.

En las reglas formales gramaticales para un lenguaje, cada una de las unidades sintácticas es invocada por un símbolo. Todas aquellas unidades sintácticas, que son hechas por construcciones pequeñas, de acuerdo con las reglas gramaticales se llaman símbolos no-terminales. Todas aquellas las unidades sintácticas que no pueden ser subdivididas se llaman símbolos terminales, como por ejemplo: identificadores, constantes (numéricos y letras) y operadores aritméticos.

Los grupos sintácticos de *DigitLAB* se representan en la gramática por símbolos no-terminales: “expresiones”, ”declaraciones” y “definición de función”. Cada uno de los símbolos no-terminales debe tener una regla gramatical que muestre cómo está construido.

Para especificar la sintaxis del lenguaje utilizado en *DigitLAB* se diseñó una *gramática independiente del contexto*, la cual describe de forma natural la estructura jerárquica de la construcción del lenguaje aquí utilizado. Este diseño se realizó por medio de un generador

de analizadores sintácticos *Bison*[3] ([Rei92]), que es un generador multi-propósito de *parsers*[4] que convierte descripciones de una gramática a un programa en *C*, el cual puede ser utilizado para un gran rango de *parsers* de lenguaje.

El analizador sintáctico de *Bison* lee una secuencia de *tokens* como su entrada y los agrupa usando las reglas gramaticales llamando al analizador léxico cada vez que éste necesita un nuevo *token*. Si la entrada es válida, el resultado final es que el total de secuencias de *tokens* se reduce a una única agrupación cuyo símbolo es el signo gramatical de inicio.

Un analizador sintáctico hecho en *Bison* consta de dos partes:

(i) Declaraciones, donde se colocan declaraciones ordinarias en *C* (delimitadas por %{ y %}) de los componentes léxicos de la gramática (Ver Tabla 4.4).

Nombre de variables o functores	%token <union_text> TK_NOUN
Números enteros	%token <union_text> TK_INTEGER
Números reales	%token <union_text> TK_REAL
Palabras	%token <union_text> TK_STRING
Caracteres o letras	%token <union_text> TK_CHAR

Tabla 4.4: Declaración de componentes léxicos (*tokens*) (Ver Anexo 3).

La proposición %token <union_text> TK_NOUN declara que TK_NOUN es un componente léxico (*token*). Los componentes léxicos declarados en esta sección se utilizan en la sección de las reglas de traducción.

[3] Bison. Reproducción libre de programa Yacc desarrollado por la GNU (Copyright (C) 1989, 1991 Free Software Foundation, Inc.675 Mass Ave, Cambridge, MA 02139, USA.)

[4] Programas que reconocen patrones léxicos en texto por medio de un archivo de datos de entrada o una entrada estándar para una descripción específica.

(ii) Reglas de traducción, donde cada regla (delimitada por %%) consta de una producción de la gramática y la acción semántica asociada con un conjunto de producciones de *DigitLAB* (Ver Tabla 4.5).

LADO IZQUIERDO	LADO DERECHO
seq_of_composed_clauses:	seq_of_composed_clauses composed_clause | composed_clause ;
composed_clause:	line_clause TK_CUOT_MARK { eafit_dgl_process_LineClause( );} | TK_exit TK_CUOT_MARK { return 0;} ;
line_clause:	function_clause | assign_clause ;

Tabla 4.5: Lado izquierdo y lado derecho de la gramática (Ver Anexo 3).

Por notación las producciones con el mismo no-terminal del lado izquierdo pueden tener sus lados derechos agrupados, con los lados derechos alternativos separados por el símbolo |, que se leerá “o”.

4.2.4. Análisis sintáctico. El análisis sintáctico se hace por medio del analizador sintáctico de *Bison* descrito en la sección anterior. En éste proceso la aceptación de una secuencia de palabras es equivalente a la construcción de un árbol sintáctico utilizando una técnica de compilación orientada a la gramática conocida como *traducción dirigida por la sintaxis* con la cual traducen expresiones infijas a la forma postfija.

La forma postfija es una notación en la que los operadores aparecen después de sus operandos, por ejemplo la forma postfija de la expresión: *surf = link_levels(pols,200,close)* es *surf pols 200 close link _levels =* (Ver Fig 4.1).

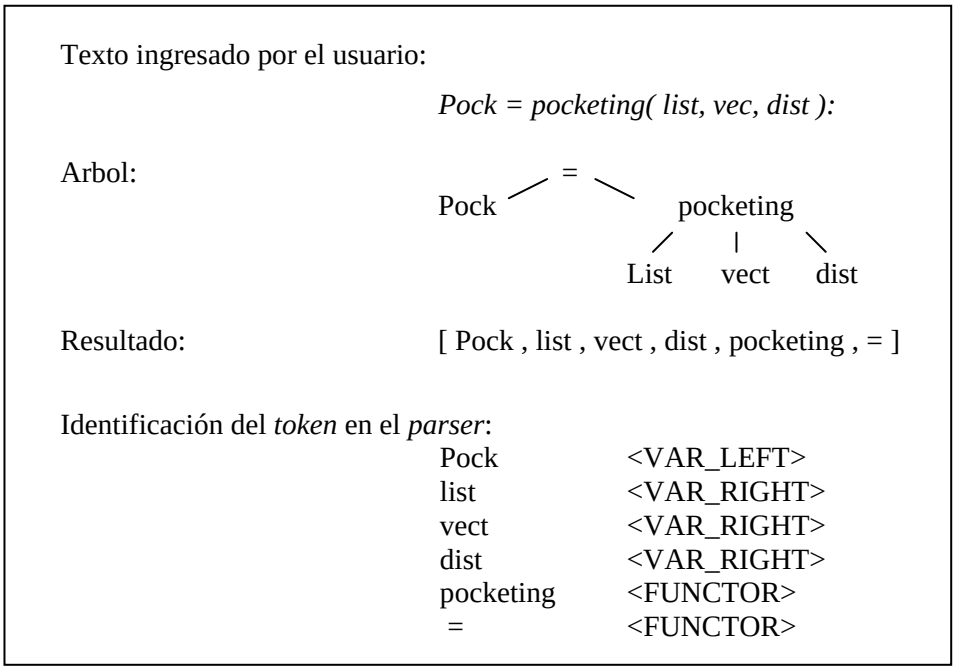


Fig 4.1: Creación de la lista de *tokens* en forma postfija

En la ejecución del *parser* se utilizan funciones que adicionan *tokens* a una lista de tipo *LPOM* (Ver Anexo 5), añadiendo información del tipo de *token* en proceso. Los tipos de *tokens* son:

Tokens tipo VAR_LEFT: Las funciones o procedimientos generan resultados, que se pueden almacenar en una variable a la cual se le asigna un nombre.

Tokens tipo VAR_RIGHT: Para la ejecución de funciones se necesita la existencia de parámetros con los cuales el procedimiento puede generar resultados correctos.

Tokens tipo FUNCTOR: Son las funciones o procedimientos que posee *DigitLAB* para el procesamiento de digitalizaciones (Ver Tabla 4.6).

FUNCTOR	ACCION ASOCIADA
TK_pocketing	{ $$ = constr_Token(K_pocketing,$1,FUNCTOR);}
TK_build_coplanar	{ $$ = constr_Token(TK_build_coplanar,$1,FUNCTOR);}
TK_build_sample	{ $$ = constr_Token(TK_build_coplanar,$1,FUNCTOR);}
TK_see_database	{ $$ = constr_Token(TK_see_database,$1,FUNCTOR);}
TK_save_database	{ $$ = constr_Token(TK_save_database,$1,FUNCTOR);}

Tabla 4.6: Algunas acciones semánticas de la gramática aplicada (Ver Anexo 3).

Para la adición de los *tokens* al *stack* de operaciones se creó una lista de tipo *LPOM* para el almacenamiento de los objetos en forma de pila; esto describe un mejor manejo de la información suministrada por el *parser*.

Procedimiento de adicionar (*push*) y retirar (*pull*) *tokens* del *stack*:

(a) Adicionar (*push*) a la lista del *stack* los *tokens* de tipo Var_Left y Var_Right.

(b) Al encontrar un *token* tipo functor, identificar cuántos parámetros tiene y de qué tipo son.

(c) Retirar (*pull*) de la lista del *stack* el número de parámetros intensificados en el numeral (b) y hacerles una verificación de tipos.

(d) Si la verificación de tipo fue exitosa, proceder a ejecutar la función con los parámetros que fueron retirados del *stack*.

(e) Adicionar el resultado de la ejecución de la función al *stack*.

4.2.5. Verificación de tipos. Un comprobador de tipos asegura que el tipo de una construcción coincida con el previsto en su contexto. El diseño de un comprobador de tipos para un lenguaje se basa en información acerca de las construcciones sintácticas de lenguaje, la noción de los tipos y las reglas para asignar tipos a las construcciones del lenguaje.

Se crearon plantillas para la verificación de tipos y la ejecución de funciones utilizando las siguientes expresiones de tipos: (i) Tipos básicos: integer, long, double, char. (ii) tipos especializados: cip_hlst, cip_ilst, cip_vlst, cit_vect, cit_hndl, cip_vec, etc.

Aquí se especifica un comprobador de tipos para un lenguaje simple en el que se debe declarar el tipo de cada identificador antes de que el identificador se utilice.

Para el proceso de ejecución de la función dada por el usuario se implementaron dos fases:

1. Fase de chequeo CHECK: En esta fase se hace una verificación de tipos, con el fin de detectar errores de forma preventiva.
2. Fase de ejecución EXEC: En esta fase se hace la ejecución de las funciones que el usuario ha dispuesto. Esta parte se ejecuta únicamente si el proceso de CHECK entregó un resultado exitoso.

Para esto se creó una plantilla en *C/C++*, que sirve de base para implementar en forma unificada cualquier función. La base de este procedimiento es la de retirar del *stack* los parámetros que esta función necesita y efectuar las acciones de la fase respectiva para obtener la ejecución de la función.

4.3. IMPLEMENTACIÓN DE LA BASE DE DATOS DE *DIGITLAB*.

En la interacción entre el *stack* de operaciones y las funciones, se requiere la creación de una base de objetos geométricos; esta base de datos es lo más importante para una ejecución asincrónica de rutinas para razonamiento geométrico para modificar los datos de la digitalización. La base de datos de *DigitLAB* es en forma de *container* y fue desarrollada como una lista permanente de objetos múltiples (*LPOM*) en *C* siguiendo una filosofía orientada a objetos.

Ella provee los objetos que la función por ejecutar necesita (por medio de una interacción con el *stack*). La definición de entidades por nombre es un aspecto importante de la implementación, puesto que permite al usuario la ejecución de funciones (comandos) utilizando el lenguaje de programación (ej. *Puntos = clasify_levels (ini_pts, norm_vec, dist_pln):*).

4.4. IMPLEMENTACIÓN DE CÓDIGOS DE ERROR EN TIEMPO DE EJECUCIÓN.

Si los usuarios de compiladores sólo escribieran programas correctos, el diseño e implementación del *parser* se simplificaría mucho. Una de las funciones principales de los *parsers* es la de ayudar al programador a identificar y localizar los errores (Ver Tabla 4.7). Se debe considerar un manejo adecuado de errores para simplificar la estructura del *parser* y mejorar su respuesta.

TIPO	DEFINICION ASOCIADA
Léxico	Secuencia de caracteres no asignada a ningún tipo de *token* valido.
Sintáctico	Secuencia de *tokens* no valida en la gramática, i.e. no existe un árbol de *parsing* asociado con ella.
Semántico	Operador aplicado a un operando incompatible
Lógico	Llamada infinitamente recursiva.

Tabla 4.7: Tipos de errores.

En *DigitLAB* se creó un proceso de detección de errores el cual, al identificar un error, detiene la aplicación y muestra: (i) el tipo de error cometido, (ii) la entidad o variable que incurrió en el error y (iii) el código correspondiente el error por medio de los tipos de error generados para *AIS* por *CAM-I* [5].

[5] *CAM-I. Consorttium for Advanced Manufacturing – International.*

5. RESULTADOS.

5.1. CASOS DE ESTUDIO.

5.1.1. HORMA DE CALZADO: Digitalización de una horma de calzado. Se tiene un archivo de digitalización que contiene puntos en el espacio tridimensional. Los puntos están dispuestos en una serie de niveles aproximadamente paralelos entre sí, formando curvas de nivel. El conjunto inicial de puntos fue de 15.278 en 36 niveles. El mapeo de contornos fue ***m = n = 1*** (*m* y *n* son los números de contornos por cada par de niveles consecutivos).

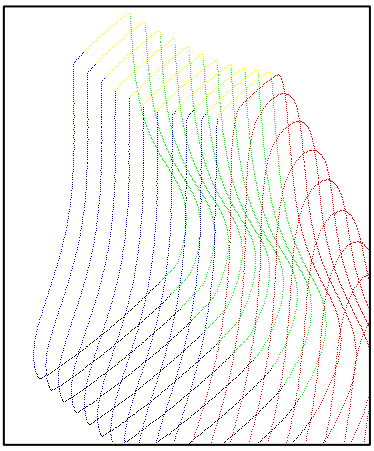


Fig. 5.1 Digitalización de los puntos de la Horma de Calzado.

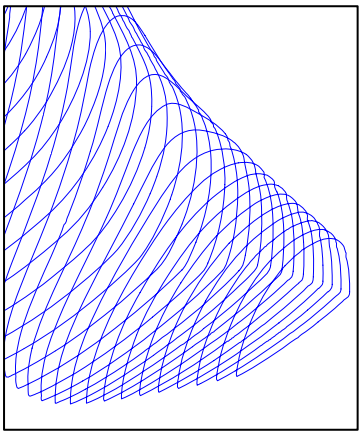


Fig. 5.2 Contornos recuperados y filtrados de la Horma de Calzado.

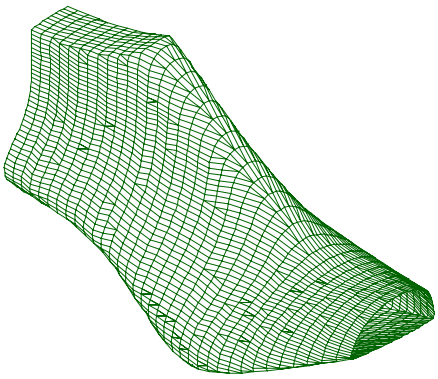


Fig. 5.3 Mapeo inter-niveles de la Horma de Calzado. Resultado final de DigitLAB.

5.1.2. HUESO HUMANO (FEMUR): Digitalización creada manualmente por medio de un brazo digitalizador. La pieza fue dividida en 4 partes dependiendo del nivel de detalle y de la dirección de los planos de digitalización: 3 extremos y un cuerpo (Ver Fig. 5.6). La distancia entre niveles para el cuerpo fue de 4 mm y para los extremos 2 mm, y entre puntos de un mismo nivel para el cuerpo fue de 4 mm y para los extremos fue de 2 mm. El conjunto inicial de puntos fue de 6.553 en 209 niveles. El mapeo de contornos fue ***m* = *n***.

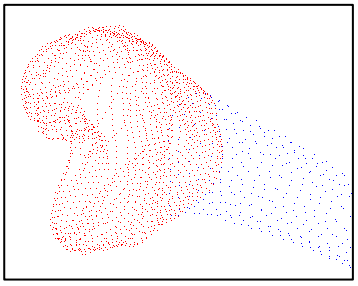


Fig. 5.4 Digitalización de los puntos del femur.

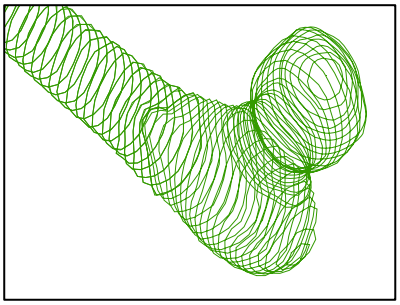


Fig. 5.5 Contornos recuperados y filtrados del femur.

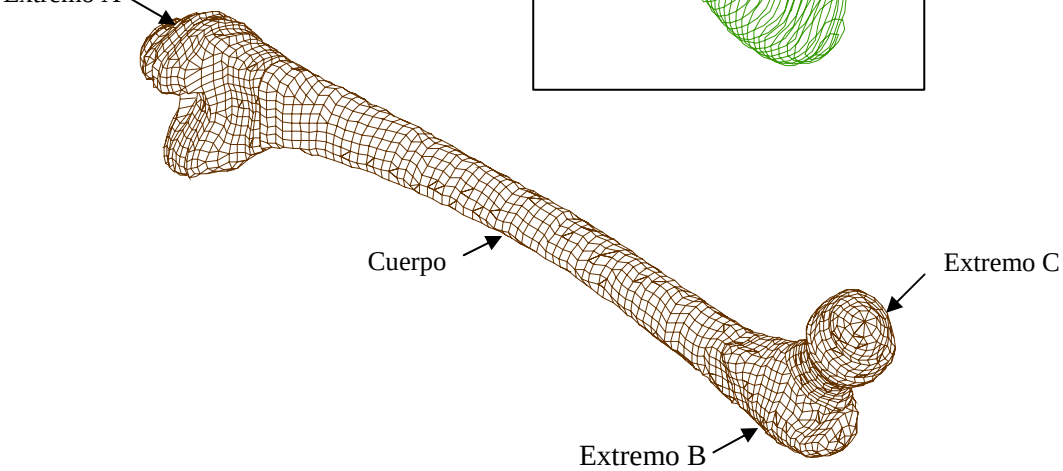


Fig. 5.6 Mapeo inter-niveles del femur. Resultado final de DigitLAB.

5.1.3. LAMPARA: Digitalización virtual de la representación CAD de una lámpara de mesa. Se mostrará solamente uno de los brazos de la lámpara, ya que es la parte del objeto de mayor complejidad. La distancia entre niveles fue de 2 mm, y entre puntos de un mismo nivel fue de 0.5 mm. La normal del plano de digitalización fue el eje Z. El conjunto inicial de puntos virtualmente obtenido fue de 19.853 en 86 niveles. El mapeo de contornos fue $\boldsymbol{m \neq n}$.

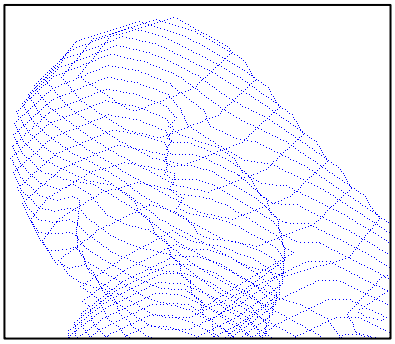


Fig. 5.7 Digitalización de los puntos de la lámpara.

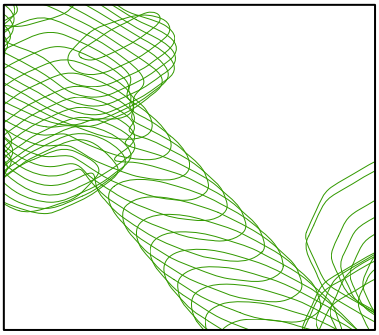


Fig. 5.8 Contornos recuperados y filtrados de la lámpara.

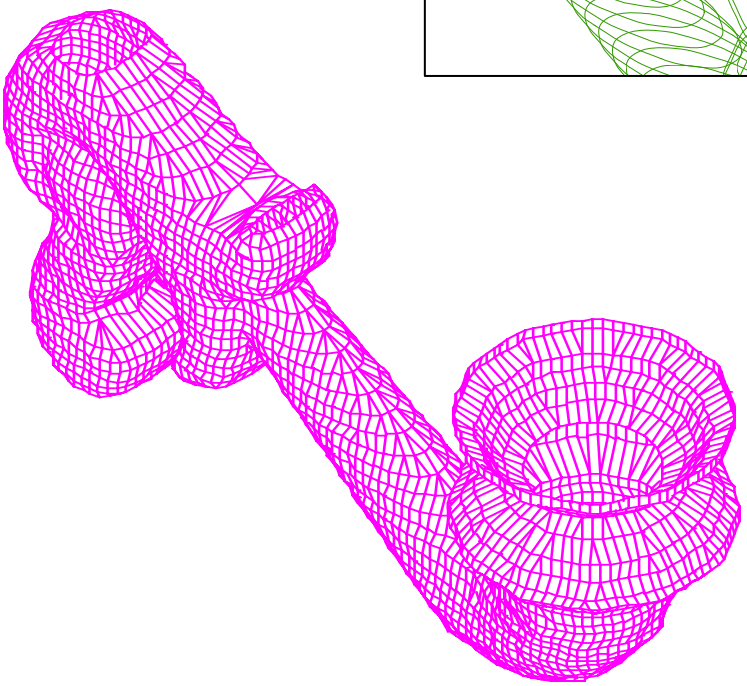


Fig. 5.9 Mapeo inter-niveles de la lámpara. Resultado final de DigitLAB.

5.1.4. PLANO GEOGRAFICO: Se parte de polilineas de nivel de un plano geográfico hecho mediante fotointerpretación. Los planos son paralelos al plano XY, con una distancia ente planos de 4 mts. Las polilineas fueron remuestreadas con una distancia ente puntos de 4 mts. El conjunto inicial fue de 31 niveles. El mapeo de contornos fue ***m = n = 1***.

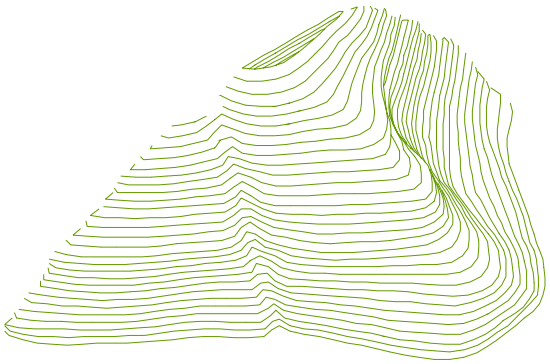


Fig. 5.10 Contornos recuperados y filtrados del plano geográfico.

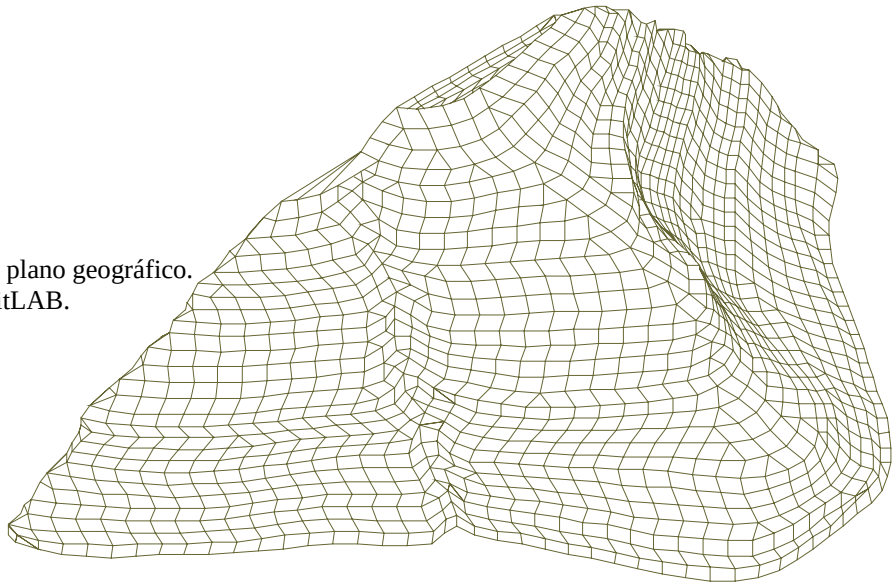


Fig. 5.11 Mapeo inter-niveles del plano geográfico. Resultado final de DigitLAB.

5.1.5. TRI-FOIL: Digitalización virtual de la representación CAD de un Tri-foil en tres dimensiones. Se hizo un muestreo virtual de las facetas con planos paralelos al plano XZ, La normal del plano de digitalización fue el eje Y, el conjunto inicial fue de 5952 facetas tridimensionales. La distancia entre niveles fue de 120 mm y entre puntos de un mismo nivel fue de 80 mm. Se logró una digitalización virtual de 28.835 puntos en 327 niveles. El mapeo de contornos fue $\boldsymbol{m \neq n}$.

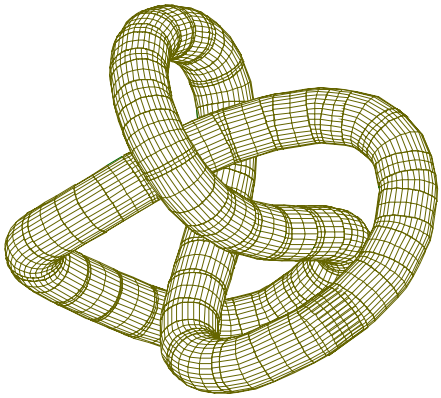


Fig. 5.12 Superficie original de facetas del Tri-foil

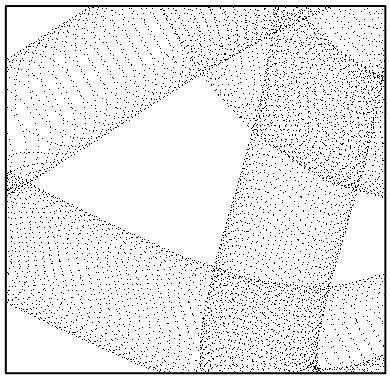


Fig. 5.13 Digitalización virtual del Tri-foil

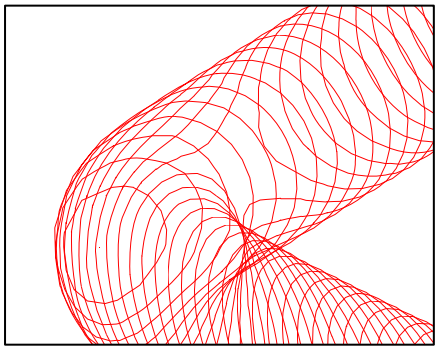


Fig. 5.14 Contornos Recuperados del Tri-foil.

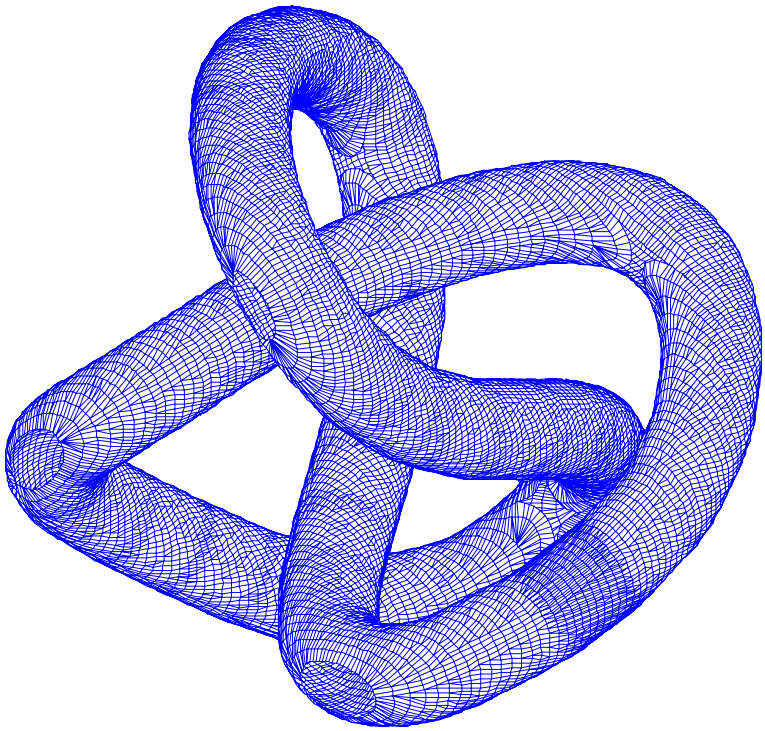


Fig. 5.15 Mapeo inter-niveles del Tri-foil. Resultado final de DigitLAB

5.1.6. TÚNELES: Digitalización creada manualmente por medio de un brazo digitalizador. La distancia entre niveles fue de 4 mm, y entre puntos de un mismo nivel fue de 2 mm. La normal del plano de digitalización fue el eje X. El conjunto inicial de puntos fue de 3.819 en 104 niveles. El mapeo de contornos fue ***m ≠ n.***

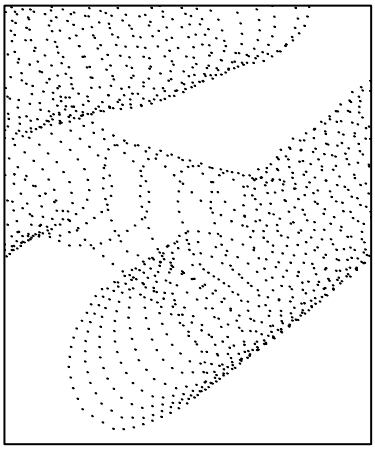


Fig. 5.16 Digitalización de los puntos de los túneles

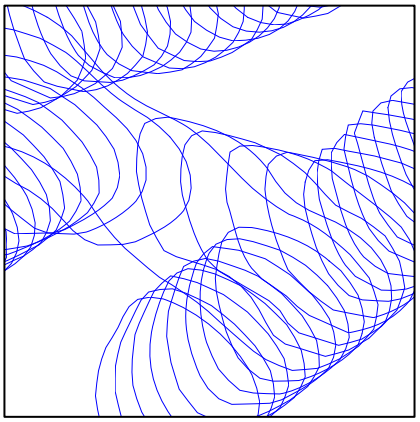


Fig. 5.17 Contornos recuperados y filtrados de los túneles.

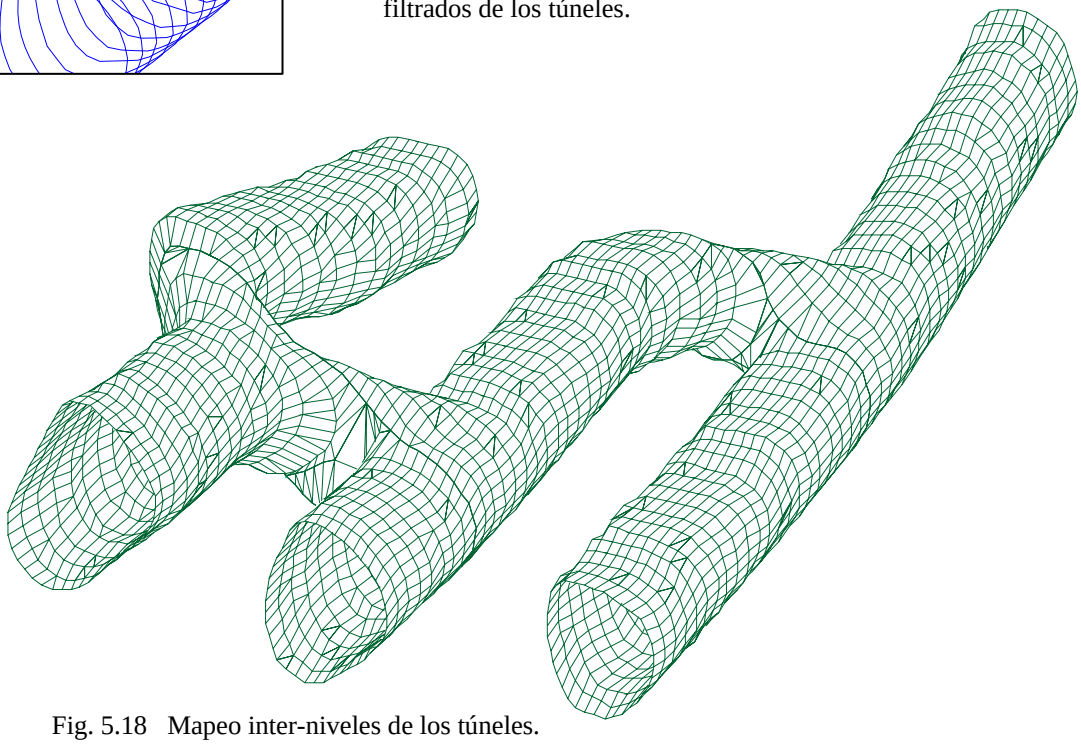
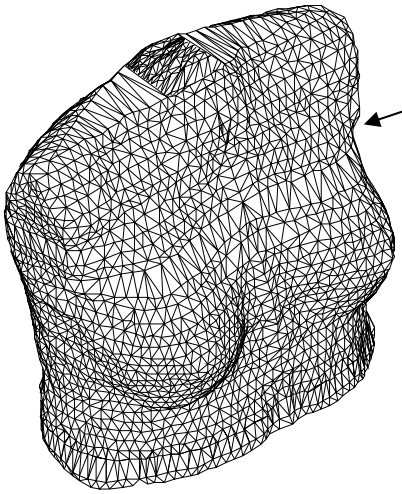


Fig. 5.18 Mapeo inter-niveles de los túneles. Resultado final de DigitLAB.

5.1.7. TORSO FEMENINO: Digitalización virtual de la representación CAD del Torso de una mujer, el conjunto de datos fue suministrado por el Centro Interdisciplinario de Investigación CII-CAD/CAM/CG de la Universidad EAFIT (Digitalización de Contacto). Se hizo un muestreo virtual de las facetas con planos paralelos al plano XY. La normal del plano de digitalización fue el eje Z, el conjunto inicial fue de 6884 facetas tridimensionales. La distancia entre niveles fue de 10 mm y entre puntos de un mismo nivel fue de 3 mm. Se logró una digitalización de 8.020 puntos en 33 niveles.



CONJUNTOS DE DATOS SUMINISTRADOS POR
CII-CAD/CAM/CG EAFIT
(Digitalización de Contacto)
Indeportes – Jesus Camacho (Planeación y Realización de Protocolos Antropométricos)
Johanna Acosta (Tratamiento de Datos)
Oscar Ruiz. (Digitalización y Asesoría)

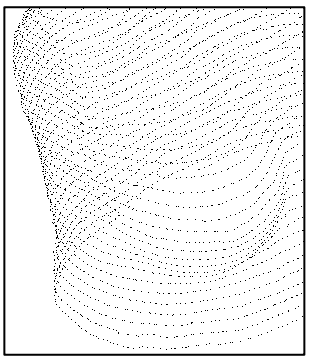


Fig. 5.19 Digitalización virtual del torso femenino.

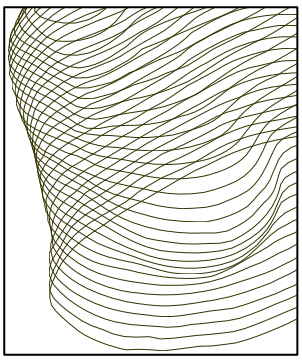


Fig. 5.20 Contornos recuperados y filtrados del torso femenino.

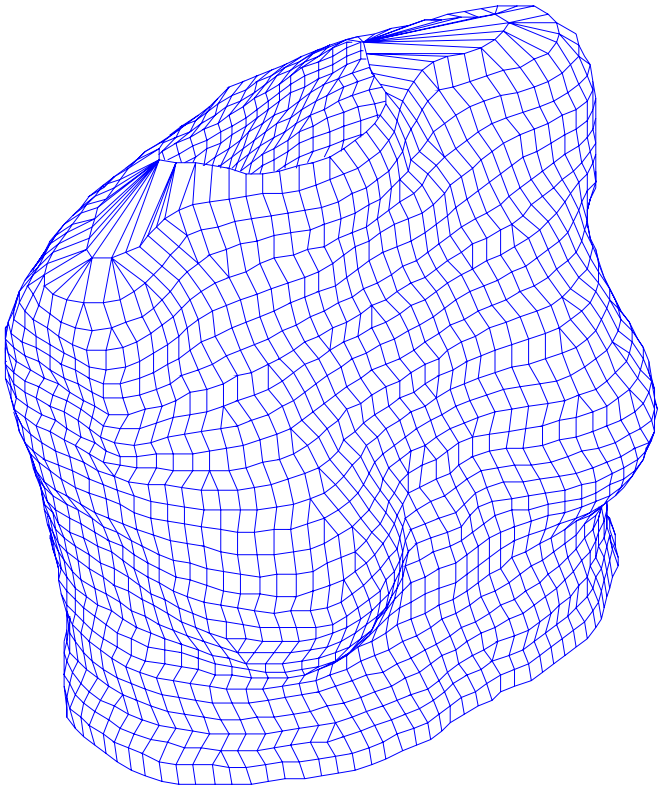


Fig. 5.21 Mapeo inter-niveles del torso femenino.
Resultado final de DigitLAB

5.1.8. TORO: Digitalización virtual de la representación CAD de un Toro en tres dimensiones. Se hizo un muestreo virtual de las facetas con planos paralelos al plano XY, La normal del plano de digitalización fue el eje Z, el conjunto inicial fue de 12.398 facetas tridimensionales. La distancia entre niveles fue de 2 mm y entre puntos de un mismo nivel fue de 0.5 mm. Se logró una digitalización virtual de 166.810 puntos en 142 niveles. El mapeo de contornos fue $\boldsymbol{m \neq n}$.

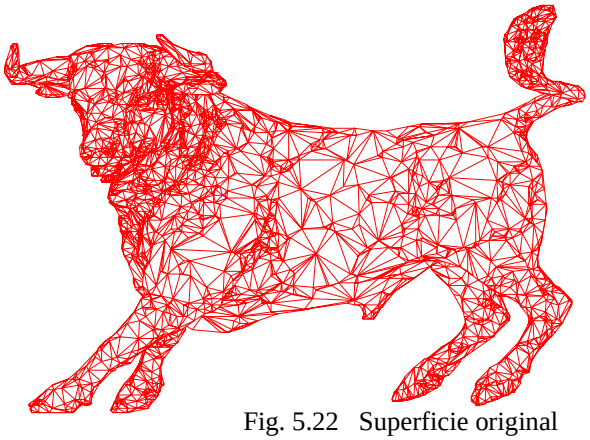
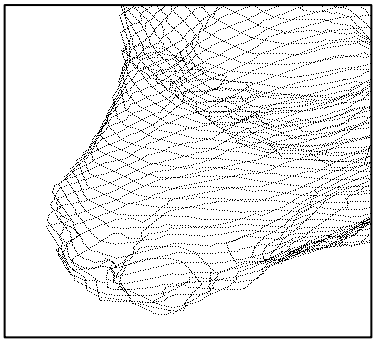
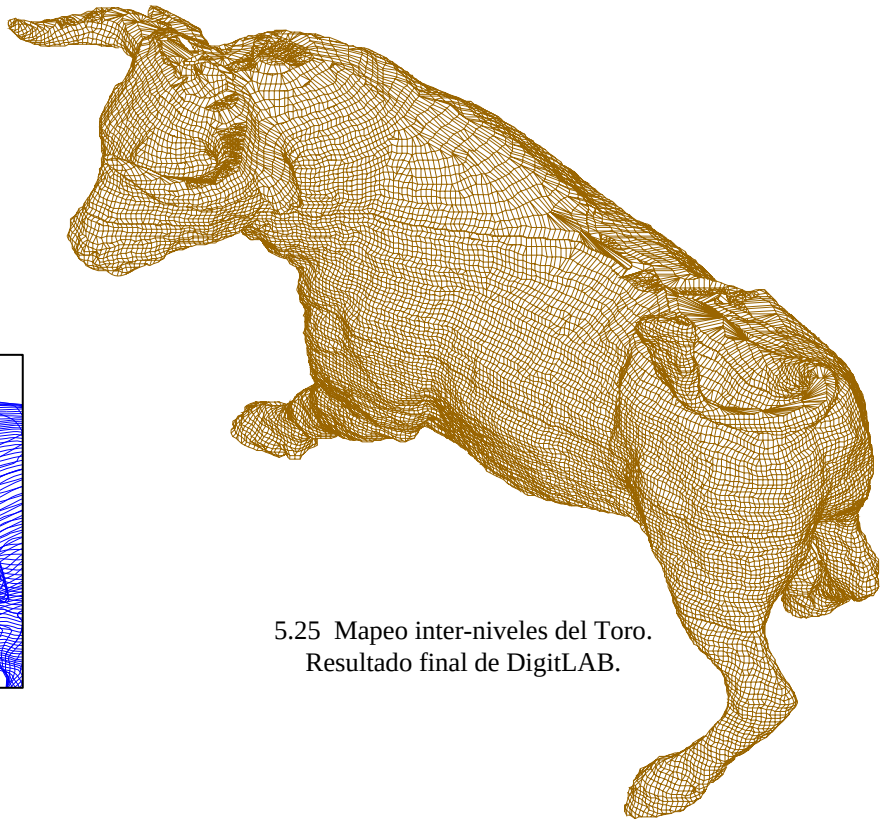


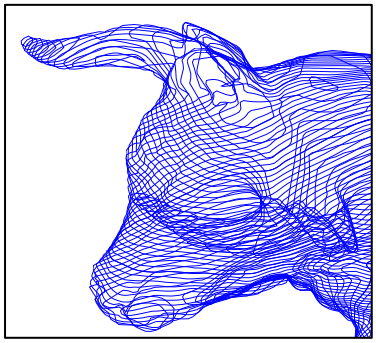
Fig. 5.22 Superficie original de facetas del Toro



5.23 Digitalización virtual de los puntos del Toro.

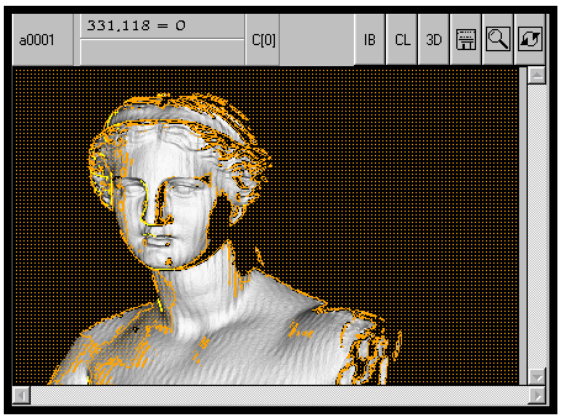


5.25 Mapeo inter-niveles del Toro. Resultado final de DigitLAB.

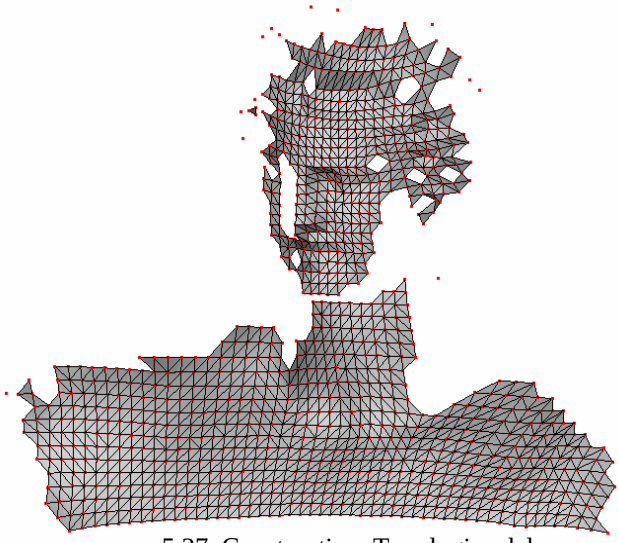


5.24 Contornos recuperados y filtrados del Toro.

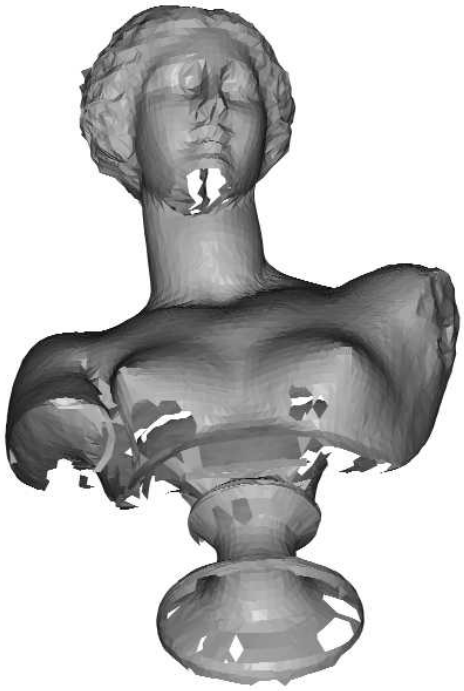
5.1.9. APHRODITE: Reconstrucción de las superficies de una escultura de Afrodita a partir de datos ópticos (Range Pictures). En el muestreo óptico se hicieron 23 Range Pictures. El resultado es del Range picture # a0001 que tiene un tamaño de imagen original de 768 x 484 pixeles, la imagen reducida cada 10 pixeles presenta un tamaño de 76 x 48 pixeles con 958 puntos. Se logró una reconstrucción de 1.522 facetas.



5.26 Muesteo Optico (Range Picture #a0001)



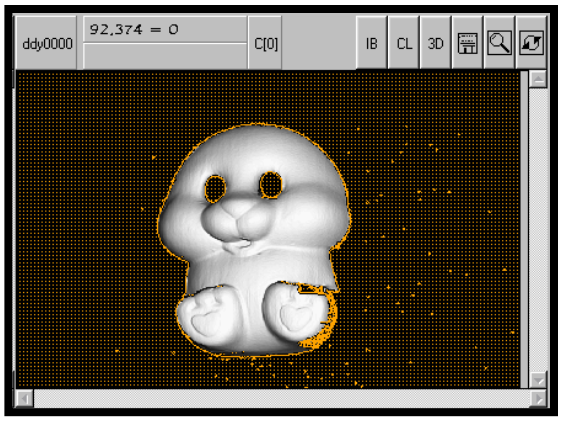
5.27 Construction Topologica del Range picture # a0001



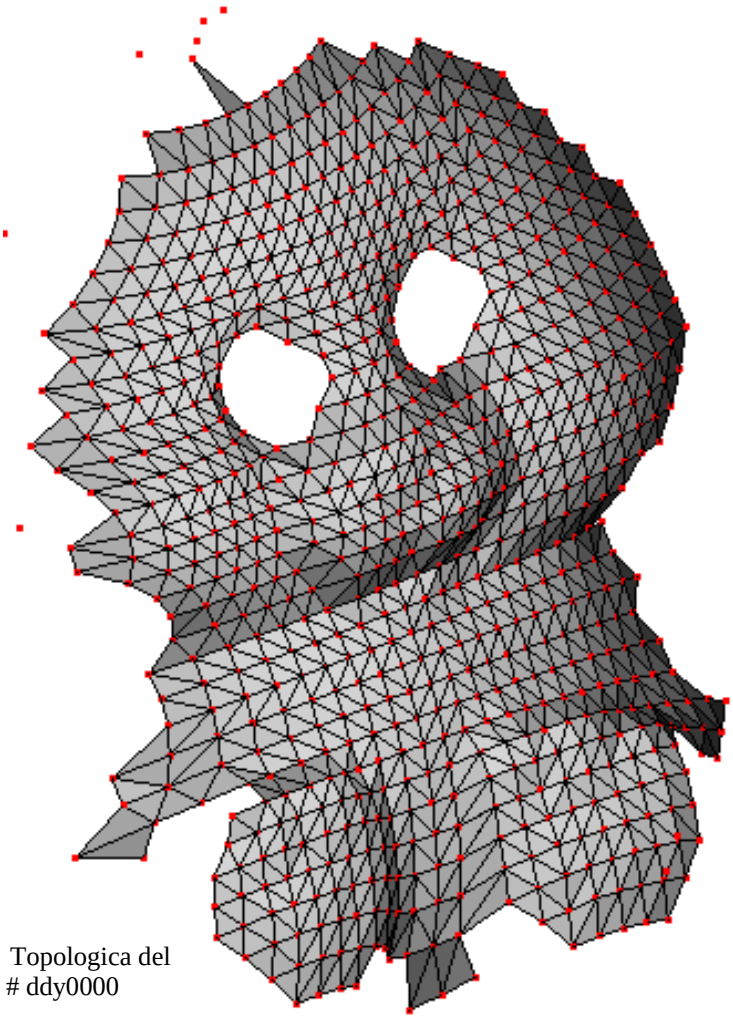
5.28 Conjunto de Máscaras. Resultado intermedio de DigitLAB

5.29 Faceteo integrado de un conjunto de mascaras. Resultado final de DigitLAB.

5.1.10. TEDDY BEAR: Reconstrucción de las superficies de una máscara de un oso de peluche a partir de datos ópticos (Range Pictures). En el muestreo óptico se hicieron 9 Range Pictures. El resultado es del Range picture # ddy0000 que tiene un tamaño de imagen original de 768 x 484 pixeles, la imagen reducida cada 10 pixeles presenta un tamaño de 76 x 48 pixeles con 760 puntos. Se logró una reconstrucción de 1.388 facetas.

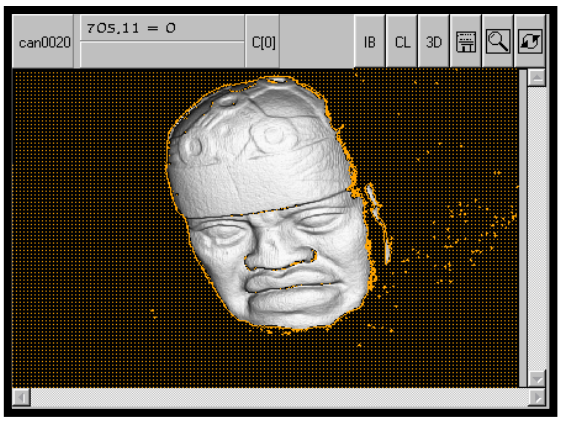


5.30 Muesteo Optico (Range Picture #ddy0000)

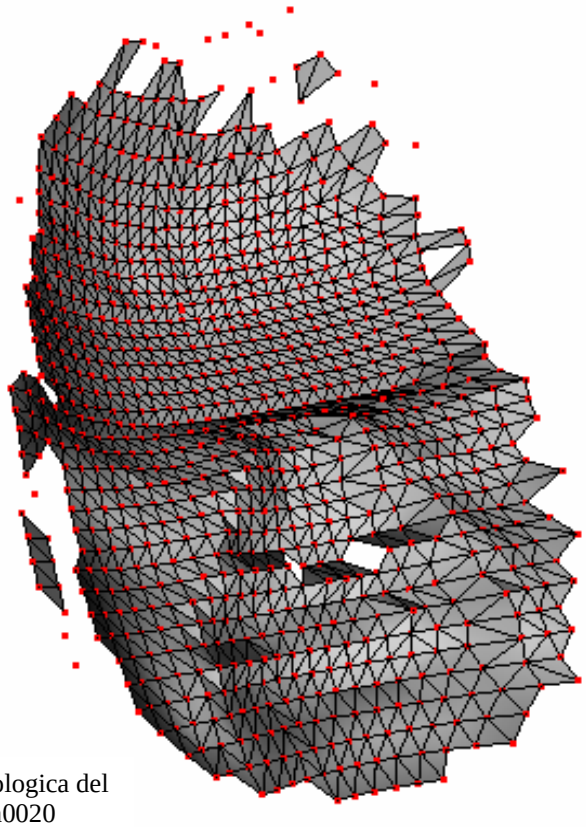


5.31 Construction Topologica del Range picture # ddy0000

5.1.11. STONE HEAD: Reconstrucción de las superficies de una cabeza de piedra a partir de datos ópticos (Range Pictures). En el muestreo óptico se hicieron 23 Range Pictures. El resultado es del Range picture # can0020 que tiene un tamaño de imagen original de 768 x 484 pixeles, la imagen reducida cada 10 pixeles presenta un tamaño de 76 x 48 pixeles con 900 puntos. Se logró una reconstrucción de 1.655 facetas.

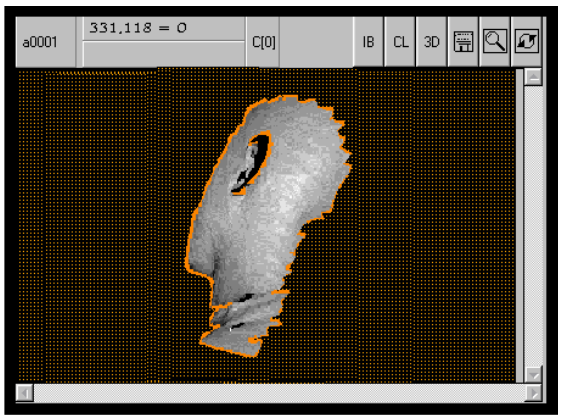


5.32 Muesteo Optico (Range Picture #can0020)

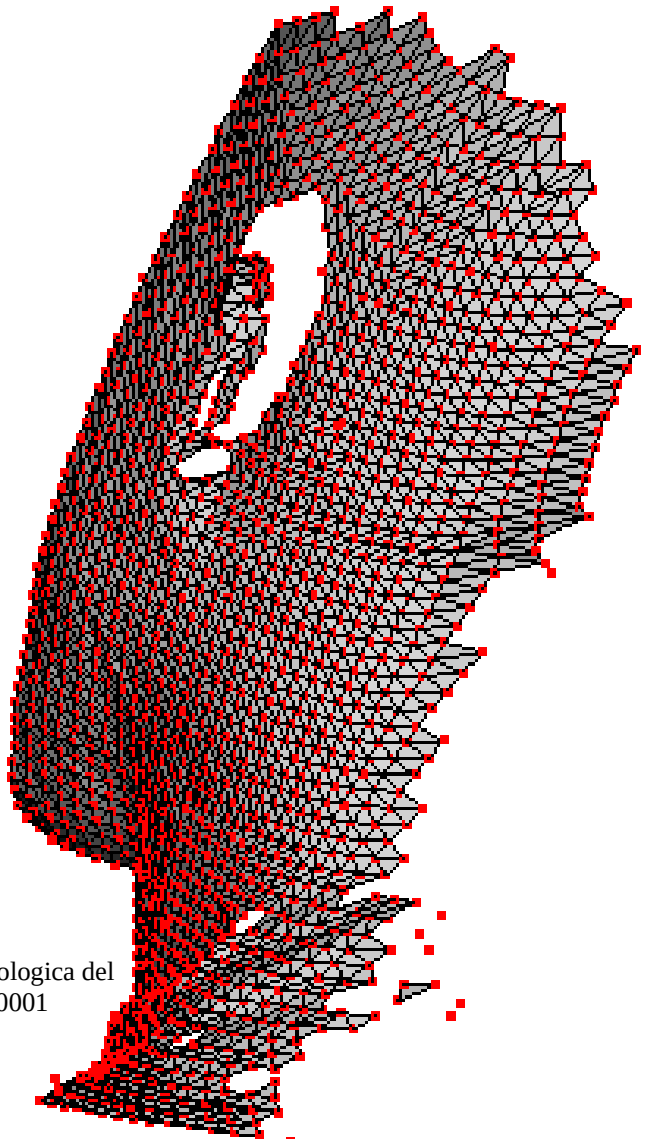


5.33 Construction Topologica del Range picture # can0020

5.1.12. EAR: Reconstrucción de las superficies de la sección de la oreja de una cabeza humana a partir de datos ópticos (Range Pictures). En el muestreo óptico se hizo 1 Range Pictures. El resultado es del Range picture # a0001 que tiene un tamaño de imagen original de 768 x 484 pixeles, la imagen reducida cada 10 pixeles presenta un tamaño de 76 x 48 pixeles con 1.680 puntos. Se logró una reconstrucción de 3.071 facetas.



5.32 Muesteo Optico (Range Picture #a0001)



5.33 Construction Topologica del Range picture # a0001

5.2 DESCRIPCION COMPLETA DEL PROCESO PARA UNA MANDÍBULA.

El presente caso de estudio toma los datos de ([RH98]), de donde se obtienen como entrada las radiografías (*hard copies*) de una *Tomografía axial computarizada* (*CAT*) de la mandíbula inferior de un sujeto. Su objetivo es producir una representación superficial (en facetas) de la mandíbula, para su ulterior análisis mediante elementos finitos. Como se ha acentuado anteriormente, el acceso a la información digital fue imposible, dado el formato propietario en que está registrada, y que sólo es interpretado por el software del fabricante del tomógrafo. Esta circunstancia es una constante en la mayoría de trabajos con digitalizaciones, sean industriales, geográficas o médicas. Romper esa interdependencia constituye una de las motivaciones principales de esta investigación. Por lo tanto, se requirió el paso inicial de convertir la información de las radiografías en información digital de algún tipo.

5.2.1. Traducción a formato digital. El paciente fue sometido a un procedimiento *CAT* sobre un área de exploración de longitud 6.3 cm de la mandíbula inferior y con cortes axiales (perpendiculares a la columna vertebral), con intervalos de 1 mm. Las radiografías fueron pasadas por *scanners*, luego de hacer marcas especiales para registrar los sistemas coordenados (Xi,Yi) locales a cada recuadro. Dichos sistemas permiten el posterior ensamblaje de la información de los cortes individuales bajo un sistema coordenado común. La Fig. 5.36 muestra algunos resultados *raster* (*TIF*) del proceso de *scan*. Obsérvese que coexisten regiones óseas con cortes de dientes y molares (éstos no serán estudiados en esta aplicación). Esta diferencia es significativa para el estudio odontológico y, por lo tanto, se deben ajustar a hueso y dentadura modelos superficiales separados. El hecho de que en las

secciones aparezcan cortes de hueso inconexos dificulta no sólo la recuperación de las secciones sino el tendido de superficies sobre la nube de puntos.

5.2.2. Vectorización. La Fig. 5.37 muestra la vectorización de una sección aproximadamente igual a la mostrada en la Fig. 5.36. Las vectorizaciones pueden, en algunos casos, realizarse automáticamente. En este caso, sin embargo, dado el bajo contraste en áreas importantes de la imagen *raster*, se hizo necesaria la intervención de un usuario. La Fig.5.37 muestra simultáneamente las vectorizaciones de hueso y dientes. Las Figs. 5.36 y 5.37 (que no están en la misma escala) sugieren el considerable ahorro en espacio resultante de la vectorización. En la Fig. 5.37 sólo se conserva, en formato (x,y,z), la secuencia de puntos de interés.

5.2.3. Muestreo de hueso. La vectorización correspondiente al hueso en la Fig.5.37 muestra que las hipótesis de la sección 5.2 no se cumple.

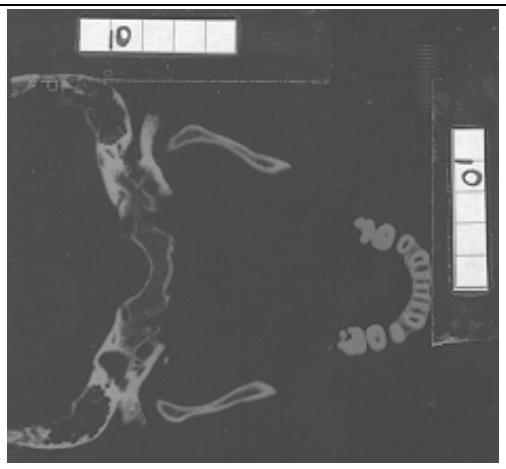


Fig. 5.36: Scan TIF de una sección de la mandíbula.

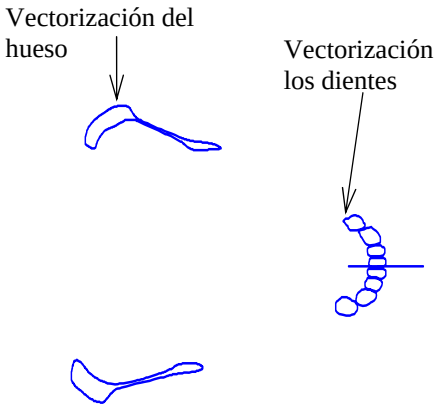


Fig. 5.37: Vectorización de una sección de la mandíbula.

El polígono ***P*** correspondiente al hueso no tiene *kernel*, y su centro de gravedad ***cg(P)*** cae fuera de ***P***. Adicionalmente, las secciones de corte son no conexas. Por estas razones, el algoritmo de reconstrucción de secciones basadas en ordenamiento angular no puede ser aplicado aquí. El diseñador debe rehacer manualmente el polígono ***P*** basándose en los puntos muestreados. Nótese que la hipótesis de que la digitalización está organizada en planos paralelos sí continua cumpliéndose, aunque el algoritmo de partición no fue aplicado, dado que en ningún momento se tuvo acceso a datos (x,y,z) de la digitalización en bruto.

5.2.4. Ajuste de facetas y resultado final. Las facetas fueron dimensionadas (por requerimiento del software *FEM*) a tamaños típicos de 6 mm de lado. La Fig.5.38 muestra la disposición topológica del arreglo de facetas. Las polilíneas deben ser remuestreadas por distancia. En cada nivel las facetas quedan fijas por los puntos remuestreados sobre la polilínea en el nivel anterior. El arreglo de facetas está prácticamente definido, excepto por casos especiales en los cuales el siguiente nivel colapsa o crece en tamaño. En dicho caso se genera un remuestreo por número de puntos de la sección cambiante, y/o se aceptan elementos triangulares además de los rectangulares básicos. Las Figs. 5.38, 5.39 y 5.40 muestran el resultado final como se entrega al software FEM. Nótese que el arreglo de facetas es topológicamente ordenado, excepto por las secciones de transición.

En el proceso de recuperación de la superficie se aplicaron los procesos para el manejo de digitalizaciones de DigitLAB, lo cual significa que fue un proceso automático.

En este proceso de reconstrucción no se reconstruyeron los dientes y molares.

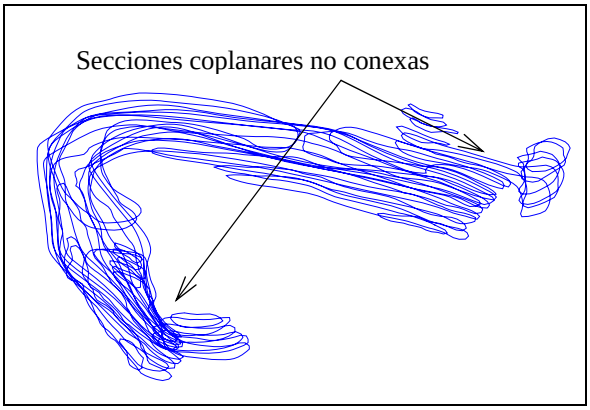


Fig. 5.38: Secuencia completa de polígonos planares.

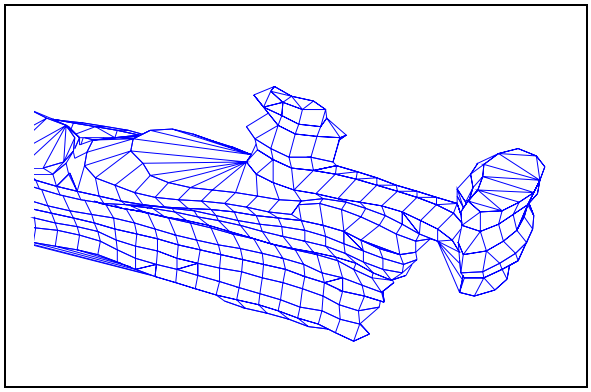


Fig. 5.39: Sección de mandíbula en facetas.

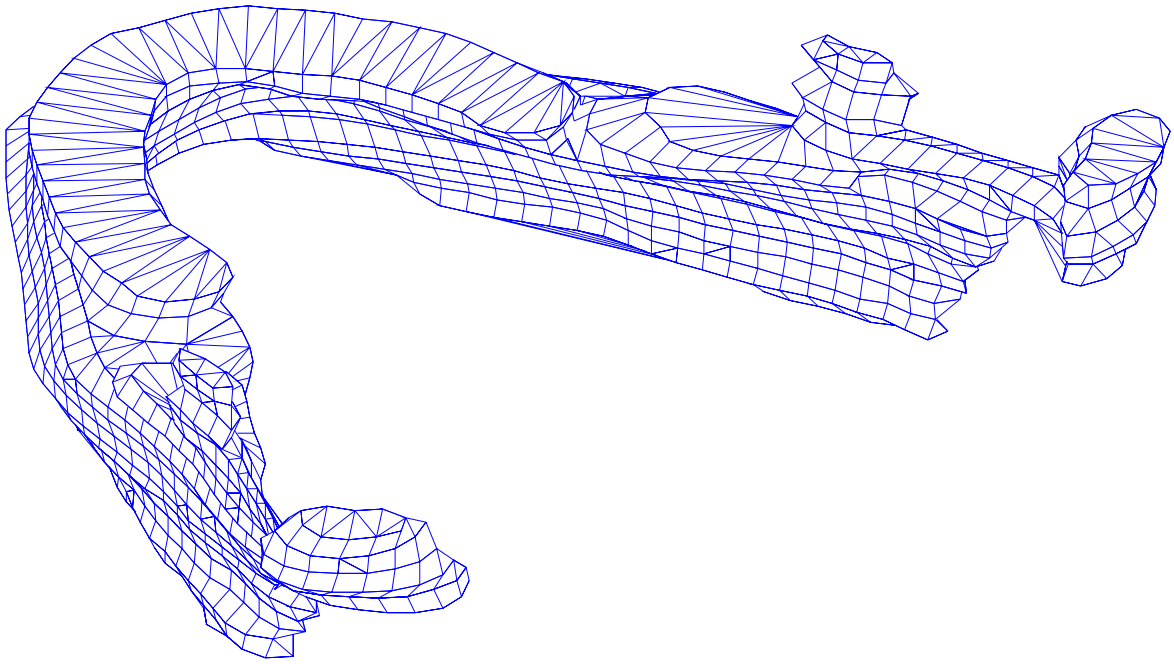


Fig. 5.40: Mandíbula en facetas topológicamente ordenadas para *FEM*. Resultado final de DigitLAB.

6. CONCLUSIONES Y FUTUROS DESARROLLOS.

Este reporte ha descrito la etapa 1999 del proyecto *DigitLAB*. Dicha etapa ha representado una mejora substancial con respecto a los proyectos *HormaCAD* (1997) y *AIS* para Curvas y Superficies Suaves (1998) en razón de que: (i) la capacidad de geometría computacional de la entrega presente se ha ampliado significativamente, y (ii) el ambiente circundante para la aplicación de dichas herramientas se ha reforzado por medio de la implementación del lenguaje *DigitLAB*.

Los puntos anteriores se expanden así:

(i) Geometría Computacional y Razonamiento Geométrico.

DigitLAB se ha concentrado en digitalizaciones que se hacen con patrones de muestreo especifico. Ellas están presentes en la mayoría de muestreos por contacto (*Coordinate Measurement Systems*), los cuales muestrean planarmente (*slices*), y en la rejilla de pixels de *range pictures*. Las utilidades implementadas se pueden dividir en cinco areas:

(a) manipulación estadística, generación de digitalizaciones virtuales,

(b) recuperación de secciones generales,

(c) mapeo general inter – niveles o recuperación topológica,

(d) algoritmos para recuperación de mascaras parciales a partir de *range pictures*

(e) utilización de (a)-(d) para la integración de las mascaras parciales provenientes de varias *range pictures*.

(a) Manipulación estadística, generación de digitalizaciones virtuales.

Las digitalizaciones son realizadas en varios patrones, dictados por el posicionamiento del objeto y el sistema sensor. Esto, unido a los defectos de medición, de muestreo, ruido, oscilación del sistema, etc., producen muestreos defectuosos. Ello se traduce en violación del paralelismo de las secciones de medición, distancia irregular entre ellas, no confinamiento de los puntos medidos en los planos idealmente establecidos, muestreo irregular, o demasiado laxo, etc. Todas estas características afectan la calidad de la superficie recobrada. *DigitLAB* provee herramientas para generar digitalizaciones virtuales, diferentes a las hechas físicamente, corregir los puntos que se desprenden de los planos de muestreo, generación de muestreos uniformes a partir de no uniformes, y, en el caso de secciones despobladas, "préstamo" de puntos entre una y otras. Este ultimo rasgo corresponde a una solución puramente intuitiva, poderosa, pero que requiere de algunas herramientas adicionales de corrección de los efectos generados por el "préstamo" de puntos (programadas también en *DigitLAB*).

(b) Recuperación de secciones generales.

En las versiones anteriores de algoritmos para recuperación de secciones de corte de objetos por planos (*slices*) se limito el alcance a secciones convexas o con *kernel*. Esta suposición se superó en la presente versión, atacando secciones con *kernel* vacío o inconexas (se presentan cuando el plano de muestreo corta al objeto en varias ramas). El algoritmo genera todos los cortes del objeto con el plano de muestreo, asumiendo una digitalización de calidad mínima. En caso de que la digitalización viole el criterio de Nyquist, obviamente ningún algoritmo podría hacer la recuperación de superficie respectiva.

(c) Mapeo general inter – niveles o recuperación topología,

Dados dos planos de sección consecutivos con ***m*** y ***n*** secciones de corte recuperadas respectivamente, las dos acciones básicas consisten en (1) relacionar secciones de un nivel con las del otro (la relación no tiene condiciones –uno a uno, sobre, función, etc.). y (2) mapear puntos entre las secciones relacionadas para producir un *lofting* o piel entre secciones de corte. La operación (1) es no trivial puesto que la geometría y topología del objeto pueden haber cambiado drásticamente entre los planos de muestreo.

Como resultado, secciones de corte pueden haber colapsado, o pueden realizar una evolución en trenza, etc. La operación (2) representa un numero infinito de funciones candidato. El espacio de búsqueda se reduce aplicando el criterio de que las facetas (*facets*) de la piel deben cumplir con los factores de forma usables por el Método de Elementos Finitos. La naturaleza del problema es tal que la efectividad de cualquier heurístico usado estará limitada por la calidad de la digitalización física realizada. Pero el Investigador Principal considera que se ha cubierto un dominio razonable de los objetos industriales, artísticos u orgánicos.

(d) Algoritmos para recuperación de mascaras parciales a partir de *range pictures*

Los Algoritmos para la recuperación de mascaras (*mesh*) parciales a partir de digitalizaciones por *range pictures* fueron escritos por el Investigador Principal en la Pasantia que realizo en el Instituto *Fraunhofer* de Gráficas por Computador (*Fraunhofer IGD*) , División de Imágenes Medicas, en *Darmstadt*, Alemania. Los algoritmos realizados evitan el problema prevalente hasta el momento, de inferir la superficie del objeto que es invisible en una imagen. Hasta el momento, los investigadores hacen hipótesis sobre dicha estructura, obviamente entregando objetos

con artefactos extraños acoplados a ellos. La línea básica del trabajo realizado fue la de no inferir ninguna información no presente en el *range picture*, y en vez de ello expresar por medio de una estructura de datos topológicamente correcta y consistente la incertidumbre en dichas regiones. Ello permite una fácil recuperación en caso de que haya imágenes auxiliares (de hecho, es la premisa del trabajo de *Fraunhofer IGD*).

(e) Utilización de (a)-(d) para la integración de las mascaras parciales provenientes de varias *range pictures*.

DigitLAB permitió, como un bono adicional, inesperado, la integración de las mascaras parciales (un problema que también *Fraunhofer IGD* ataca) en una superficie única del objeto. Ello fue posible gracias a las herramientas de muestreo y corrección estadística de *DigitLAB*. Se generaron digitalizaciones virtuales de las mascaras, produciendo un conjunto de datos planarmente organizado, al cual se aplicaron las demás herramientas *DigitLAB*. Ello permitió hacer el consenso de información entre mascaras de una forma relativamente simple. Este consenso es motivo de investigación en la comunidad científica. Aunque es aventurado decir que *DigitLAB* encontró la solución al problema, ciertamente produjo una solución que podría ser mejorada.

(ii) implementación del lenguaje *DigitLAB*.

DigitLAB fue creado con la premisa básica de entregar herramientas de trabajo al diseñador, que le permitan mejorar el desempeño de los algoritmos de razonamiento geométrico descritos arriba. Por ello se implementaron (a) capacidades para crear, eliminar, nombrar, accesar y manipular subpartes de las nubes de puntos o entidades geométricas presentes. (b) parámetros de control que permiten sintonizar, a nivel de

usuario, los heurísticos que los algoritmos geométricos utilizan. (c) capacidades funcionales en la gramática *DigitLAB* (*LALR*, *parser* implementado con *Flex* y *Bison*) que permiten la descripción de secuencias de acciones desarrolladas sobre los datos, y (d) capacidades de almacenamiento y recuperación de la base de datos *DigitLAB* entre sesiones. Aunque las funciones son invocadas vía botones, ello no debe ocultar el hecho de que ellos representan únicamente macros que representa los *tokens* de las cláusulas que componen en lenguaje. Ello implica que *DigitLAB* tiene una estructura similar a C, *MatLAB™* or *Mathematica™*.

En desarrollos futuros el *CII-CAD /CAM/CG* enfocara

a- El problema de identificación de *shells* dentro de *shells*, presente en casos como el de una esfera de pared finita, o de órganos de organismos animales.

b- Tratamiento de la superficie realizada para eliminar el patrón planar de su parte estética, por medio de técnicas de relajación o re-triangulación.

c- Ampliación de los algoritmos de integración de mascaras parciales.

d- implementación de acceso indexado a recursivo, hasta el nivel permitido por el dato en cuestión.

e- implementación de ciclos de control y funciones como un paso adicional para lograr un lenguaje de programación sobre digitalizaciones.

REFERENCIAS BIBLIOGRÁFICAS

[Ans93] ANSI Y14.5.1M-Draft, “Mathematical Definition of Dimensioning and Tolerancig Principles”, American Society of Mechanical Engineers. New York, 1993.

[Aut97] Autodesk, Inc. AutoCAD Release14 ARX Developer’s Guide. USA, 1997

[BEN97a] Bentley Systems, Inc. Programmer´s Reference Guide. USA, 1997

[BEN98] Bentley Systems, Inc. Introduction to MDL. USA 1998

[BFB98] Borhese, N., Ferrigno, G., Baroni, G., Pedotti, A., Ferrari, S., Savare, E., “AutoScan: A Flexible and Portable 3D Scanner”. IEEE Computer Graphics and Applications, Computer Graphics I/O Devices May 1998, pp. 39-41.

[BRV91] Bolle, M, Rudd, C., Vemuri, 1991, "On Three Dimensional Surface Reconstruction Methods". *IEEE Transactions on Pattern Analysis and Machine Intelligence* Vol3,No1, pp. 1-13.

[Car95] Carr, K., “Modeling and Verification Methods for the Inspection of Geometric Tolerances Using Point Data”, Ph.D. Dissertation, University of Illinois at Urbana-Champaign, 1995.

[CC78] Catmull, E., Clark, J., “Recursively Generated B-Spline Surfaces on Arbitrary Topological Meshes”. Computer Aided Design, Vol 10, No 6, Nov, 1978, pp 350-355.

[DCH88] Drebin, R., Carpenter, L., Hanrahan, P., “Volume Rendering”. Proceedings, ACM SIGGRAPH, 1988, pp. 65-74.

[DS78] Doo, D., Sabin, M., 1978, "Behavior of Recursive Division of Surfaces near Extraordinary Points ". *Computer Aided Design*, Vol 10, No 6, Nov, pp 356-360.

[Ede87] Edelsbrunner, H., “Algorithms in Combinatorial Geometry”, Springer Verlag. New York, 1987.

[Ede93] Edelsbrunner, H., “The Union of Balls and its Dual Space”. Proceedings, 9th Annual Symposium on Computational Geometry, 1993, pp. 218-231.

[Ede94] Edelsbrunner, H., “Three Dimensional Alpha Shapes”. ACM Transactions on Graphics, Vol 13, No 1, Jan, 1994, pp 43-72.

[Far90] Farin, G., “Curves and Surfaces for Computer Aided Geometric Design”. A Practical Guide Academic Press. Boston, 1990.

[GH95] Grimm, C., Hughes, J., “Modeling Surfaces of Arbitrary Topology using Manifolds”. Proceedings, SIGGRAPH - 95 Annual Conference Series, Los Angeles, August 6-11, 1995, n. 29, pp 359-368.

[Guo97] Guo, B., “Surface Reconstruction: From Points to Splines”. Computer Aided Design, Vol 29, No 4, 1997, pp 269-277.

[Hof89] Hoffmann, C., 1989, "Geometric and Solid Modeling", *Morgan Kauffmann Publishers*.

[HDD92] Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., Stuetzle, W., 1992, "Surface Reconstruction from Unorganized Points". *Proceedings, SIGGRAPH -92 Annual Conference Series*, Chicago, July 26-31, n. 26, pp. 71-78.

[Hel99] Held, M., “FIST: Industrial-Strngth Triangulation of Polygons”. Institut für Computerwissenschaften, Universität Salzburg, Austria, 1999.

[KV95] Khan, M., Vance, J., “A Mesh Reduction Approach to Parametric Surface Polygonization”. Proceedings, ASME Design Engineering Technical Conferences. -Advances in Design Automation, 1995, pp. 41-48.

[LC87] Lorensen, W., Cline, H., 1987, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm", *ACM Computer Graphics*, Vol. 21, No. 24, July, pp. 163-169.

[Neu97] Neugebauer P., 1997, "Geometrical Cloning of 3D Objects via Simultaneous Registration of Multiple Range Images". *Proceedings, International Conference on Shape Modeling and Applications*, Aizi-Wakamatsu, Japan, March 3-6.

[Mor85] Mortenson, M., “Geometric Modeling”, John Wiley and Sons. New York, 1985.

[Nyq28] Nyquist, H.,“Certain Topics Telegraph Transmission Theory”, AIEE *Trans*, 47, 1928, 617-644.

[PS85] Preparata, F. and Shamos, M. I., 1985, "Computational Geometry. An Introduction", Springer Verlag. New York.

[PTR98] Petrov, M., Talapov, A., Robertson, T., Lebedev, A., Zhilyiaev A., Polonsky, L., 1998, "Optical 3D Digitizers: Bringing Life to the Virtual World". *IEEE Computer Graphics and Applications, Computer Graphics I/O Devices* May/June 1998, pp. 28-37.

[Ram87] Ramos, J., "Introducción a la Tomografía Axial Computarizada". Memorias, XIII Congreso Latinoamericano de Informática, 1987, pp. 532-544.

[RBP87] Ramos, A., Basso, K., Partichelli, P., Scharcanski, J., Longhi, M., "Um Sistema para Visualizaçao Tridimensional de Tomografía Computarizada". Memorias, XIII Congreso Latinoamericano de Informática, 1987, pp. 545.

[Rei92] O'Reilly & Associates, Inc. Unix Program Tools FLEX & BISON. USA, 1992.

[RH98] Ruiz, O., Henao, C., "Modelaje Geométrico de Estructura Ósea". INGEGRAF-98. X Congreso Internacional de Ingeniería Gráfica, Jun 3-5, 1998 Málaga, España.

[RL99] Ruiz, O., Leiceaga, X., "DigitLAB: A Geometry Sensitive Environment for Digitization Processing". Submitted for publication to the Journal of the Institute of Industrial Engineers (IIE) Transactions. Vigo, Spain, Feb 19, 1999.

[Sam95] Samet, H., 1995, "Application of Spatial Data Structures: Computer Graphics, Image Processing and GIS", Addison Wesley Publishing. Co.

[SI97] Saldarriaga, J. and Isaza Dario, "An Implementation of the AIS Interface on AutoCAD", B.Sc. Thesis, EAFIT University Medellin, Colombia, 1997.

[STT93] Szeliski, R., Tonnesen, D., Terzopoulos, D., 1993, "Modeling Surfaces of Arbitrary Topology with Dynamic Particles". *Proceedings, IEEE Conference on Computer Vision and Pattern Recognition*, pp. 82-87.

[Van92] Vanzandt W. "Scientific Visualization: One Step in lab Analysis Workflow". Advanced Imaging, Feb, 1992, p. 20.

[YS83] Yau, M., Srihari, S.N., 1983, "A hierarchical Data Structure for Multidimensional Digital Images" , *Communications of the ACM*, 66, 7(July 1983), pp. 504-515.

ANEXO 1.

AUTOCAD RUN-TIME EXTENSION (ARX) PROGRAMMING ENVIRONMENT

1.1 EL PRESENTE INFORME DESCRIBE DETALLADAMENTE EL FUNCIONAMIENTO DEL AMBIENTE DE PROGRAMACIÓN *ARX* DE *AUTOCAD*.

Los dos primeros numerales hacen una introducción a *ARX* y su relación con las anteriores *API*'s de *AutoCAD: AutoLISP* y *ADS*. El numeral 3 hace un bosquejo rápido de las librerías suministradas por *ARX*. El resto del capítulo expone en detalle la utilización de cada una de las librerías. Se hace énfasis en el uso de las librerías **`AcDb`** y **`AcGe`** por tratarse de las más utilizadas. De gran importancia es el numeral 5 donde se explica el manejo de la base de datos de *AutoCAD* mediante *ARX* utilizando la librería **`AcDb`**.
Las librerías **`AcEd`** y **`AcGi`** se exponen con brevedad debido al poco uso que se hace de ellas en este proyecto.

1.2 1. GENERALIDADES

El ambiente de programación *ARX* provee una interfaz de programación de aplicaciones *API* para desarrollar aplicaciones ejecutables sobre *AutoCAD R13* que extienden su capacidad. Esta *API*, organizada en una arquitectura orientada a objetos, utiliza el lenguaje de programación C++. Las librerías *ARX* incluyen un conjunto de herramientas que aprovechan la arquitectura abierta de *AutoCAD* para proveer acceso directo a sus estructuras de bases de datos, interfaz gráfica y definición de comandos nativos.
Una aplicación *ARX* es una (*Dynamic Link Library DLL*) que comparte espacio en memoria con *AutoCAD* y hace llamadas directas a sus funciones. Las librerías de *ARX* incluyen macros para facilitar la definición de nuevas clases; además permite aumentar funcionalidad a las clases existentes.

1.3 2. *AUTOLISP, ADS Y ARX*

ARX es incluido a partir de la versión 13 de *AutoCAD* y permite la utilización de las anteriores interfaces de programación *AutoLISP* y *ADS*. *AutoLISP* es un lenguaje interpretado que provee un mecanismo simple para añadir comandos a *AutoCAD*. Aunque hay algunas variaciones dependiendo de la plataforma, *AutoLISP* es lógicamente un proceso separado que comunica con *AutoCAD* a través de una Comunicación Interproceso (*Interprocess Communication IPC*).
Las aplicaciones *ADS* son escritas en C y compiladas. Sin embargo aparecen ante *AutoCAD* iguales a las de *AutoLISP*. Una aplicación *ADS* está escrita como un conjunto de funciones externas que son cargadas y ejecutadas por el interpretador *AutoLISP*. Las aplicaciones *ADS* se comunican con *AutoLISP* vía *IPC*.

ARX difiere de *AutoLISP* y *ADS* en varios aspectos. La diferencia más importante es que una aplicación *ARX* se basa en un enlace dinámico de librerías *DLL* que comparte espacio en memoria con *AutoCAD* y tiene acceso directo a sus objetos.
La figura 1 muestra la relación entre los tres ambientes de programación *AutoCAD*. *ARX* se sirve de gran cantidad de funciones *ADS* para lo cual usa la librería *ADS-Rx*. Esta librería, funcionalmente idéntica a la librería estándar C *ADS* permite la migración de aplicaciones *ADS* hacia *ARX*. Específicamente, *ARX* utiliza *ADS* en el proceso de interacción con el usuario para la selección de entidades, manipulación de selecciones, cuadros de diálogo programables y requerimiento y adquisición de datos.

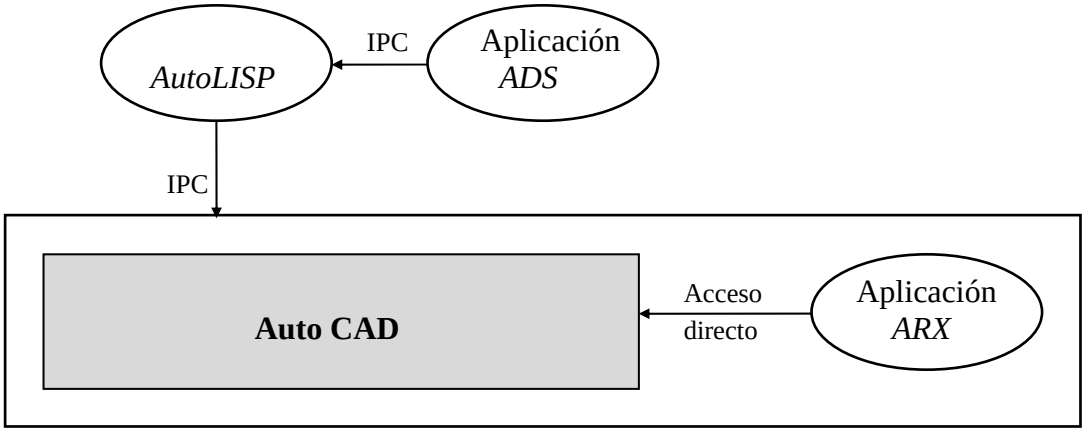


Figura 1. Comparación entre las *API*'s de *AutoCAD*

1.4 3. LIBRERÍAS *ARX*

El ambiente de programación *ARX* consta de las siguientes librerías:

`AcRx` Clases usadas para realizar la inicialización y enlace de la aplicación y para registro e identificación de clases. La clase básica de ésta librería es **`AcRxObject`**, que posee funciones para:
1) Identificación de clases y análisis de herencia de objetos en tiempo de ejecución.
2) Adición de nuevos protocolos a una clase existente en tiempo de ejecución.
3) Ensayos de comparación e igualdad de objetos.
4) Copia de objetos.
Además, la librería **`AcRx`** provee una serie de macros para crear nuevas clases derivadas de `AcRxObject`.

`AcEd` (Edición) Provee clases para definir y registrar nuevos comandos que operan de la misma forma que los comandos *AutoCAD*. De hecho, estos comandos "nativos" residen en la misma estructura interna que los comandos *AutoCAD*. Una clase importante en esta librería es `AcEditorReactor` que monitorea el estado del editor *AutoCAD* y notifica a la aplicación eventos tales como el inicio, la terminación o cancelación de un comando.

`AcDb` (Base de datos) Provee clases que permiten acceder a las estructuras de la base de datos de *AutoCAD*. Esta base de datos almacena toda la información de los objetos

gráficos (entidades) que componen un dibujo *AutoCAD* y los no-gráficos como niveles (*layers*), tipos de línea y estilos de texto. Es posible consultar y manipular instancias existentes de entidades y objetos *AutoCAD*, así como crear nuevas instancias de objetos en la base de datos.
La base de datos *AutoCAD* contiene principalmente:

a) Un grupo de nueve tablas de símbolos que son los símbolos de entrada de los objetos. Estos representan varios objetos `AcDbDatabase` comúnmente usados y sus propiedades.

b) Un objeto diccionario (de la clase `AcDbDictionary`) que provee una tabla de contenido primaria de un dibujo *AutoCAD*. Inicialmente, esta tabla de contenido posee sólo dos ítems: los identificadores (IDs) de otros dos diccionarios usados por *AutoCAD*. Las aplicaciones, sin embargo, son libres de añadir otros objetos al diccionario.

c) Un conjunto variado con cerca de 200 definiciones (#define) contenidas en archivos "*header*" definidas por *AutoCAD* (Estas variables no son objetos de la base de datos).

`AcGi` **(**Interfaz gráfica**)** Provee la interfaz gráfica usada para dibujar las entidades *AutoCAD*. Esta librería es usada por algunas funciones miembro de `AcDbEntity`, tales como `worldDraw(), ViewportDraw(),` y `saveAs()` que son parte del protocolo estándar. Los objetos `AcGiWorldDraw()`proveen el método `AcDbEntity::worlDraw()` que puede producir una representación gráfica para cada "*viewport*".

`AcGe` (Geometría) Esta librería es usada por la librería **`AcDb`** y provee clases utilitarias tales como vectores, puntos y matrices que se usan para realizar operaciones geométricas comunes en 2D y 3D. Además provee objetos geométricos como puntos, curvas y superficies.
La librería consta de dos partes principales: clases para geometría 2D y clases para geometría 3D. Las principales clases abstractas (que no se derivan de ninguna otra) son `AcGeEntity2d` y `AcGeEntity3d`. Existen otras clases abstractas básicas como `AcGePoint2d`, `AcGeVector2d o AcGeMatrix2d`. Estas clases básicas se usan para realizar muchos tipos de operaciones comunes, como sumar un vector a un punto y realizar productos entre vectores o matrices. Las clases de más alto nivel en esta librería fueron implementadas usando estas clases básicas. Estas son las únicas clases en la librería geométrica cuyos datos miembro son públicos.

1.5 4. JERARQUIA DE CLASES *ARX*

La estructura jerárquica de clases *ARX* está compuesta por dos grandes grupos independientes: Clases *AcRx* y Clases *AcGe*. El grupo principal *AcRx* está compuesto por todas las clases necesarias para realizar aplicaciones. El grupo de clases *AcGe* permite la creación y manipulación de instancias geométricas de los objetos.
AcRx tiene una superclase (`AcRxObject`) que abarca todas las demás (Ver figura 2). Suministra una clase básica para el manejo de la base de datos. Esto es, creación y manipulación de Objetos (`AcDbObject`) y creación y manipulación de Entidades

(`AcDbEntity`). El numeral 5 de este capítulo explica en detalle la utilización de estas clases. Las clases restantes permiten el manejo de Edición, Notificación de eventos, Interfaz gráfica y Control en tiempo de ejecución, entre otros.
Las clases *AcGe* están agrupadas en tres bloques (Ver figura 3). El primero permite la creación de entidades primitivas genéricas. Los otros dos para entidades 2D y 3D respectivamente. El numeral 6 de éste capítulo hace una descripción del uso de estas clases geométricas.
En los diagramas mostrados en las figuras 2 y 3 las líneas indican la relación "es un". Por ejemplo un *AcDbPoint* es un *AcDbEntity* el cual es un *AcDbObject* que a su vez es un *AcRxObject*.

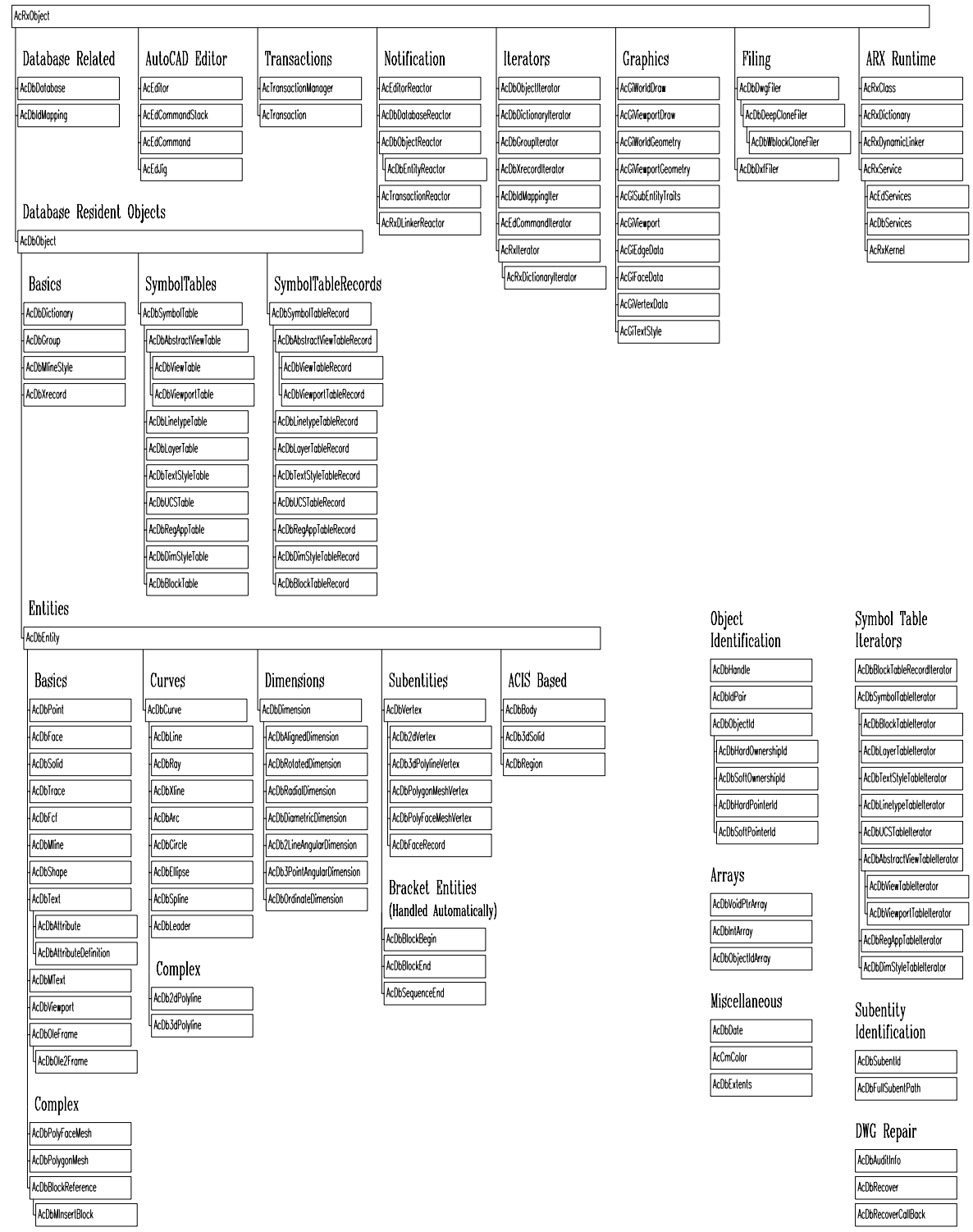


Figura 2. Jerarquía de clases *ARX* (**AcRx**)

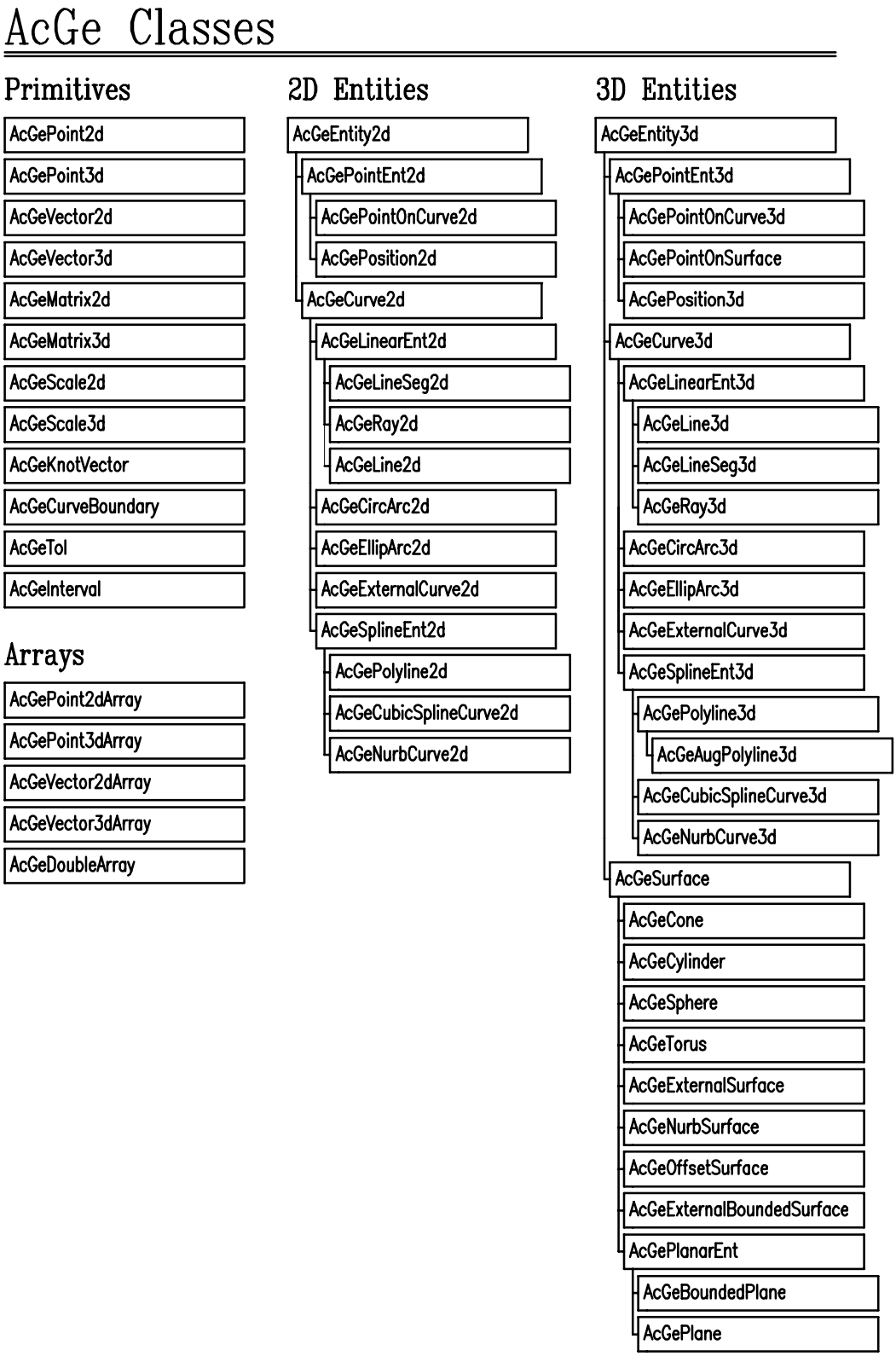


Figura 3. Jerarquía de clases *ARX* (AcGe)

1.6 5. MANEJO DE BASE DE DATOS (LIBRERÍA ACDB)

Un dibujo *AutoCAD* es una colección de objetos **`AcDb`** que son almacenados en una base de datos. Cada objeto en la base de datos tiene un "*handle*" que es un identificador único para tal objeto en el contexto de una sesión. Las Entidades son una clase especial de objetos en la base de datos que tienen una representación gráfica dentro del dibujo. Las entidades pueden ser vistas y manipuladas por el usuario.
Otros objetos importantes en la base de datos son las "tablas de símbolos" y los "diccionarios", objetos que mapean un "*symbol name*" (una cadena de caracteres) al respectivo objeto **`AcDb`**. *AutoCAD* provee un juego variado de "tablas de símbolos" en la base de datos, cada una de las cuales contiene instancias de una clase particular de registro. Ejemplos de esto son las tablas de niveles (*layers*) `AcDbLayerTable` que contienen registros de tablas de niveles y las tablas de bloques `AcDbBlockTable` que contienen registros de las tablas de bloque**s**. Todas las entidades *AutoCAD* son representadas por registros de tablas de bloques.

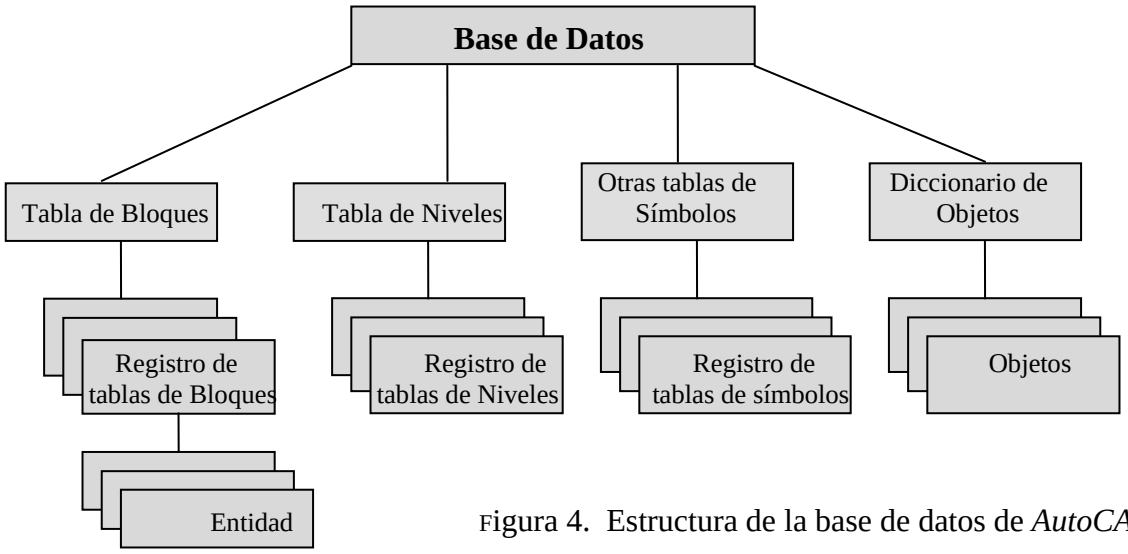


Figura 4. Estructura de la base de datos de *AutoCAD*

Las tablas de símbolos y Diccionarios son contenedores usados para almacenar objetos. Los Diccionarios proveen contenedores más genéricos de almacenamiento. Un Diccionario puede contener objetos `AcDbObject` o sus sub-clases. Se pueden crear nuevos diccionarios para agregar nuevos elementos a la base de datos. No se puede agregar nuevas tablas de símbolos a la base de datos.
Durante las sesiones de edición puede obtenerse la base de datos del dibujo actual utilizando la función global `acdbCurDwg()`.

5.1. IDENTIFICADORES DE OBJETOS (*Object ID*)

Con un *Object ID* se puede obtener el apuntador a un objeto de la base de datos para operar con él. Puede obtenerse un *Object ID* de diferentes maneras:

a) Crear un objeto y agregarlo a la base de datos. La base de datos asigna un ID al objeto.

b) Usar el protocolo de bases de datos (Tablas de símbolos y Diccionarios de Objetos) para obtener los IDs de los objetos que son creados automáticamente cuando ésta se crea.

c) Usar el protocolo de clases específicas para obtener los IDs de los objetos. Ciertos objetos, como las Tablas de símbolos y Diccionarios poseen otros objetos. Las clases que definen estos objetos proveen un protocolo para obtener los IDs de los objetos contenidos (por ejemplo `AcDbBlockTable::getAt`).

d) Usar un Iterador para atravesar la lista o conjunto de objetos. La librería **`AcDb`** provee Iteradores que se pueden usar para recorrer varias clases de objetos portadores de otros objetos (`AcDbDictionaryIterator`, `AcDbObjectIterator`).

e) Consultar una selección. Después de que el usuario haga una selección de objetos, puede consultarse dicha selección para obtener los IDs. (`acdbGetCurrentSelectionSet()`).

5.2. OBJETOS PRINCIPALES DE LA BASE DE DATOS

A medida que se crean objetos, automáticamente se van agregando a sus propios objetos portadores en la base de datos. Las Entidades se agregan a los registros en las tablas de bloques. Los registros de tablas de símbolos se agregan a las correspondientes tablas de símbolos. Los demás objetos se agregan al Diccionario de objetos o a los objetos que los contienen (y, finalmente al Diccionario de objetos).

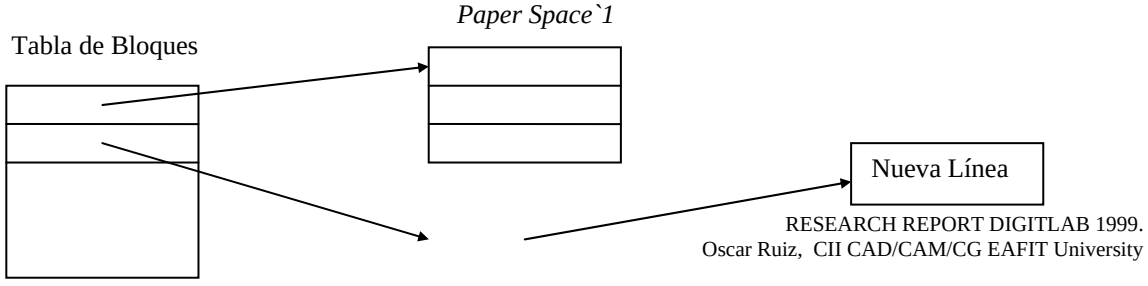
Para ser utilizable, la base de datos debe tener al menos los siguientes objetos:

a) Un grupo de nueve tablas de símbolos que incluye la tabla de bloques y la tabla de niveles. La tabla de bloques contiene por defecto dos registros: *MODEL_SPACE* y *PAPER_SPACE*. La tabla de niveles contiene por defecto un nivel: "*layer*" 0.

b) Un Diccionario de objetos. Al crearse la base de datos, éste Diccionario contiene, además de un grupo disponible, el diccionario de estilo *MLINE*. Dentro del diccionario de estilo *MLINE*, viene incluido el estilo *STANDARD*.

1.7 5.3. CREACIÓN Y MANIPULACIÓN DE OBJETOS

Cuando se invoca *AutoCAD* y la base de datos está en su estado básico, las entidades se almacenan en el espacio de modelo ("*model space*"). Este es el espacio principal en *AutoCAD* usado para los modelos geométricos y gráficos. El espacio de papel ("*paper space*") soporta documentación relacionada con dichos modelos, como hojas de planos y anotaciones textuales. Los comandos de creación de entidades las adicionan a la base de datos, lo mismo que al bloque "*model space*". Puede preguntarse a cualquier Entidad, qué base de datos y qué bloque la contienen. Si durante una sesión de modelado se crean una línea, un círculo y un nivel (*layer*), la apariencia de la base de datos es como se muestra en la figura 5.



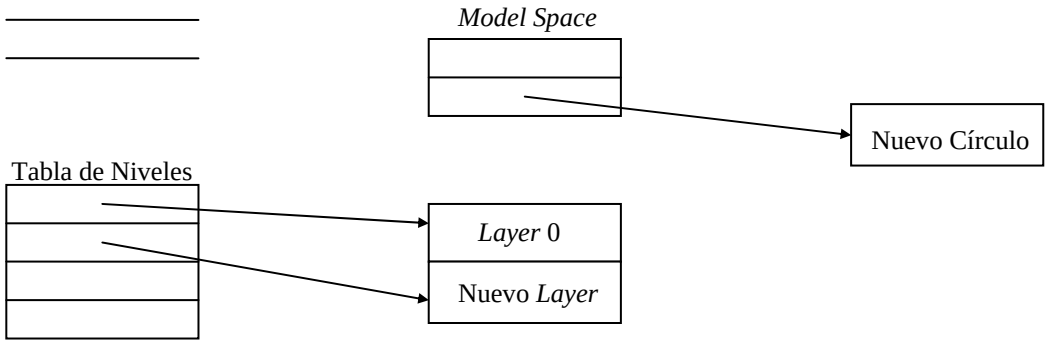


Figura 5. Creación de Objetos en la base de datos de *AutoCAD*

5.4. APERTURA Y CERRADO DE OBJETOS

Cada `AcDbObject` puede ser referenciado de tres formas diferentes: Por medio de su "*handle*", por medio de su *Object ID* y por medio de un apuntador C++.
Cuando *AutoCAD* no está ejecutándose, el dibujo es almacenado en una archivo del sistema. Los objetos contenidos en un archivo *DWG* son identificados por sus "*handles*". Una vez el archivo de dibujo esta abierto y cargado en memoria RAM, la información que contiene es accesible a través de `AcDbDatabase`. Cada Objeto en la base de datos tiene un *object ID* que persiste durante la sesión de edición desde la creación hasta el borrado de la `AcDbDatabase` en la que reside el Objeto. Las funciones de apertura toman el ID como argumento y retornan un apuntador a un `AcDbObject`. Este apuntador es válido hasta que el Objeto sea cerrado como se muestra en la figura 6.

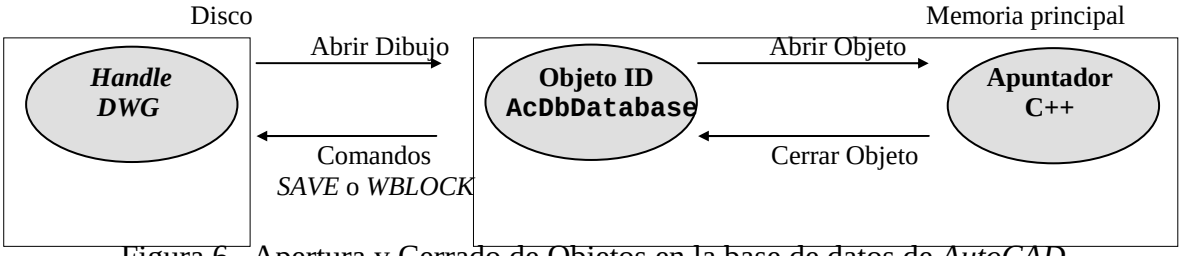


Figura 6. Apertura y Cerrado de Objetos en la base de datos de *AutoCAD*

La función global utilizada para la apertura de objetos de la base de datos es

```
extern Acad::ErrorStatus
acdbOpenObject(AcDbObject*& pObj,
               AcDbObjectId objId,
               AcDb::OpenMode mode,
               Adesk::Boolean openErasedObject = Adesk::kFalse)
```

Un Objeto puede ser abierto en uno de los siguientes modos (`OpenMode`):

a) **`kForRead`**: Un Objeto puede abrirse para lectura siempre y cuando no esté previamente abierto para lectura o notificación.

b) **kForWrite:** Un Objeto puede abrirse para Escritura sólo si no está abierto. De lo contrario, la apertura falla.

c)kForNotify: Un Objeto puede abrirse para notificación cuando el objeto está cerrado, abierto para lectura o abierto para escritura, pero no cuando está previamente abierto para notificación. Pocas veces las aplicaciones necesitan abrir un Objeto para notificación.

Mientras el objeto esté abierto para escritura puede modificarse. Al terminar las modificaciones debe ser cerrado explícitamente. De lo contrario causará errores de memoria en tiempo de ejecución. La función para cerrar estos objetos es `AcDbObject::close()`.

Nuevas instancias de los objetos se consideran abiertas. Un objeto no puede ser cerrado hasta que haya sido añadido a la base de datos.

1.8 5.5. CREACION Y MANIPULACION DE ENTIDADES

Una Entidad es un objeto de la base de datos que tiene representación gráfica. Ejemplos de Entidades incluyen líneas, círculos, arcos, textos, sólidos, regiones, splines y elipses. El usuario puede ver la Entidad en la pantalla y manipularla. Por lo demás, las Entidades son objetos en la base de datos y los conceptos de la sección anterior son aplicables. Las Entidades están agrupadas en una superclase del grupo **`AcRx`** llamada `AcDbEntity`. Sólo los objetos de esta clase pueden representarse gráficamente en la pantalla. Casi siempre las Entidades contienen toda la información necesaria acerca de su geometría. Unas pocas Entidades llamadas complejas contienen otros objetos que almacenan su información geométrica o atributos. Entidades complejas son las siguientes:

a) **`AcDb2dPolyline`** Contiene objetos `AcDb2dPolylineVertex`
b) **`AcDb3dPolyline`** Contiene objetos `AcDb3dPolylineVertex`
c) **`AcDbPolygonMesh`** Contiene objetos `AcDbPolygonMeshVertex`
d) **`AcDbPolyFaceMesh`** Contiene objetos `AcDbPolyFaceMeshVertex` y objetos `AcDbFaceRecords`
e) **`AcDbBlockReference`** Contiene objetos `AcDbAttribute`
f) **`AcDbMInsertBlock`** Contiene objetos `AcDbAttribute`

Todas las Entidades tienen sus propios métodos específicos de creación de instancias. Estos métodos crean la Entidad pero no la agregan a la base de datos. Para la adición de Entidades a la base de datos se usa el método `AcDbObject::new()`. Esto crea una instancia del objeto y lo añade a la base de datos. Si el objeto es creado primero y no se agrega a la base de datos, puede ser borrado. Sin embargo, una vez el objeto está en la base de datos, no puede borrarse.

Las entidades tienen unas propiedades básicas para su representación gráfica:

a) Color. Puede ser asignado y leído como un índice numérico de 0 a 256. Los colores de 8 a 255 son definidos por el dispositivo de *display*. Los demás tienen un significado especial. La función para asignar color es `AcDbEntity::setColorIndex`.

b) Tipo de Línea. Apunta a una tabla de símbolos que especifica una serie de puntos y segmentos usados para dibujar la línea. Cuando se crea una instancia de la entidad, su tipo de línea es `NULL`. Si se añade a la base de datos sin especificar su tipo de línea, este asume

el valor corriente de la base de datos. La función para asignar tipo de línea es `AcDbEntity::setLinetype`.
c) Escala del tipo de línea. Igual que con el tipo de línea, solo que al instanciarse la entidad, la escala tiene un valor inválido. La función para asignar la escala del tipo de línea es `AcDbEntity::setLinetypeScale`.
d) Visibilidad. Especifica si la entidad es visible o invisible. Esta propiedad es independiente del estado del *layer* donde esté la entidad y los valores corrientes de la base de datos. La función para asignar visibilidad es `AcDbEntity::setVisibility`.
e) *Layer.* Todas las entidades están asociadas a un *layer*. La base de datos siempre contiene al menos uno (*layer* 0). Puede especificarse a qué layer asociar una entidad que se crea. De lo contrario quedará en el que la base de datos tenga activo. La función para asignar un *layer* a una Entidad es `AcDbEntity::setLayer`.
Algunas funciones importantes utilizadas en la manipulación de entidades son:
a) **`AcDbEntity::transformBy()`** Usada para pasar una matriz de transformación que mueve, escala o rota puntos en el objeto.
b) **`AcDbEntity::worldDraw()`** Crea una representación geométrica de la entidad, independiente del *viewport.*

c) **`AcDbEntity::viewportDraw()`** Crea una representación geométrica de la entidad, asociada a un *viewport* específico.

d) **`AcDbEntity::draw()`** Coloca la representación geométrica de la entidad en una cola y la envía a la pantalla con todas las que estén en la cola.

1.9 6. USO DE LA LIBRERÍA GEOMÉTRICA ACGE

La librería **`AcGe`** incluye una serie de clases para representaciones geométricas comunes como puntos, líneas, curvas y superficies. Provee representaciones comunes que pueden ser usadas por cualquier aplicación *Autodesk*. Esta librería es puramente matemática. Por lo tanto sus clases no tratan directamente con la base de datos o con representaciones gráficas (*display*). Sin embargo muchas de sus clases son usadas por las librerías **`AcDb`** y **`AcGi`** para sus representaciones internas. La jerarquía de clases fue mostrada en la figura 3 del presente capítulo.
La librería **`AcGe`** provee clases para geometría simple y compleja. Las clases de álgebra lineal simple incluyen punto, vector, matriz, entidades lineales 2D y 3D, y entidades planares. Las clases complejas incluyen curvas spline y superficies *NURBS*. Como se ve en la figura 3, la jerarquía de clases ofrece clases separadas para geometría 2D y 3D. No pueden mezclarse las dos clases en una misma operación.
La librería incluye un número de tipos básicos como `AcGePoint3d`, `AcGeVector3d` y `AcGeMatrix3d` que tienen datos públicos que permiten el acceso rápido y eficiente. Estas clases simples son utilizadas por otras librerías y por las clases de `AcGe` que sean derivadas de `AcGeEntity2d` y `AcGeEntity3d`. (`AcGeEntity2d` y `AcGeEntity3d` no tienen supertipo). El chequeo de tipos de `AcGeEntity2d` y `AcGeEntity3d` se hace mediante las funciones `AcGeEntity2d::type()` o `AcGeEntity3d::type()` que retornan la clase de objeto.

Algunos aspectos relevantes de esta librería son:
a) Tolerancias. Muchos métodos aceptan un valor de tolerancia como uno de sus parámetros. Este valor es de la clase `AcGeTol` y siempre tiene un valor por defecto definido en `AcGeContext::gTol`. Algunas funciones de consulta como `AcDbCurve::sClosed()` o `AcDbCurve::isPlanar()` calculan si los puntos inicial y final están dentro de la tolerancia definida para retornar un valor booleano. La tolerancia puede cambiarse durante una llamada particular de las funciones. También puede cambiarse su valor global.
b) Geometría paramétrica. Las curvas y superficies en la librería **`AcGe`** son paramétricas. Una curva es el resultado de mapear un intervalo de línea *real* dentro de un espacio de modelado 2D o 3D usando una función evaluadora *f(u)*. Cada clase de curva tiene una función `AcGeCurve2d::getInterval()` o `AcGeCurve3d::getInterval()` que retorna dicho intervalo paramétrico.
Las curvas *ARX* tienen las siguientes características:
a) Orientación. Determinada por la dirección en la cual el parámetro incrementa.
b) Periodicidad. La curva es periódica si es cerrada. Además existe un valor *T* tal que un punto sobre la curva en (*u* + *T*) es igual al punto sobre la curva en (*u*) para cualquier parámetro *u.* Ver [3,9]. Puede averiguarse la periodicidad de la curva con las funciones `AcGeCurve2d::isPeriodic()` o `AcGeCurve3d::isPeriodic()`.

c) Cierre. Una curva cerrada es aquella cuyos puntos inicial y final son el mismo. Para saber si es cerrada o abierta se usan las funciones `AcGeCurve2d::isClosed()`o `AcGeCurve3d::isClosed()`.
d) Planitud. Una curva 3D puede ser planar o no planar. Si lo es, todos sus puntos residen en el mismo plano. Las funciones `AcGeCurve2d::isPlanar()` o `AcGeCurve3d::isPlanar()` retornan verdadero si la curva evaluada es planar.
e) Longitud. Puede obtenerse la longitud de la curva entre dos valores dados del parámetro por medio de las funciones `AcGeCurve2d::length()` y `AcGeCurve3d::length()`.
Similarmente una superficie es el mapeo de un dominio 2D dentro del espacio de modelado 3D usando una función evaluadora basada en dos argumentos *f(u, v)*. Los dos parámetros *u* y *v*, definen cada uno la dirección de las líneas paramétricas sobre la superficie.
Las superficies paramétricas definen su orientación por medio del vector normal en cada punto. Si se realiza el producto cruz entre los vectores tangentes *u* y *v* en un punto, se obtiene un vector que es normal a la superficie en dicho punto [3,9].

1.10 7. USO DE LA LIBRERÍA TOPOLOGICA ACBR

Por medio de la librería **`AcBr`** (*AutoCAD* B-rep) puede accederse a los datos geométricos y topológicos contenidos en ciertas entidades *AutoCAD*. Esta librería trata solo con objetos de las clases `AcDb3dSolid`, `AcDbBody` y `AcDbRegion`.
`AcBr` permite crear e interrogar objetos de tipo *traverser* los cuales recorren la estructura topológica de una entidad compuesta por tipos de objetos B-rep como *Faces*, *Edges* y *Vertices*. Estos objetos topológicos contienen información geométrica detallada. Esta información es obtenida de los objetos de la base de datos de *AutoCAD*. Por ejemplo, un

objeto `AcBrFace` tiene el método `AcBrFace::getSurface` que entrega el apuntador a un `AcGeSurface`, superficie portadora de la *Face*.
La librería **`AcBr`** provee acceso de solo lectura a los datos geométricos y topológicos contenidos en el modelo. Los modelos son creados y modificados usando la librería **`AcDb`**. *ARX* está limitado ya que no puede crear sólidos a partir de una definición topológica. Esto quiere decir que no se puede crear un sólido a partir de *Faces*, *Edges*, etc. Dichos objetos son consecuencia de la creación del sólido por otra vía (por ejemplo operaciones booleanas)
La librería **`AcBr`** tiene dos superclases paralelas que heredan directamente de `AcRxObject`: `AcBrEntity` y `AcBrTraverser`. La jerarquía de clases puede verse en el diagrama de la figura 7.

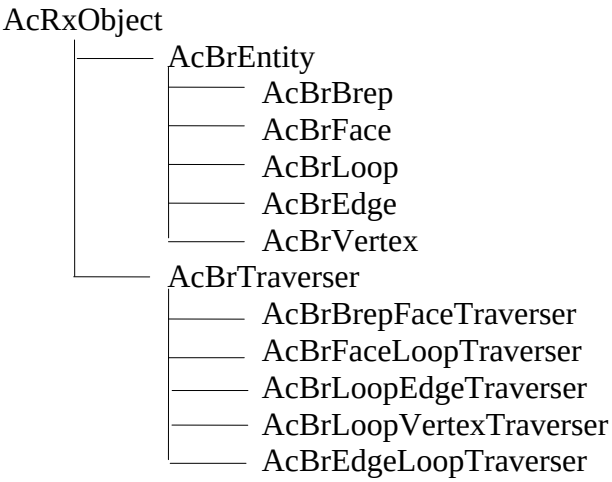


Figura 7. Jerarquía de clases (Librería **`AcBr`**)

Los conceptos de Topología y Geometría son de particular importancia para el análisis de la librería **`AcBr`** :
Un modelo sólido *AutoCAD* es una *boundary representation* (B-rep) que consta de una colección de elementos topológicos y geométricos como se aprecia en la figura 8.

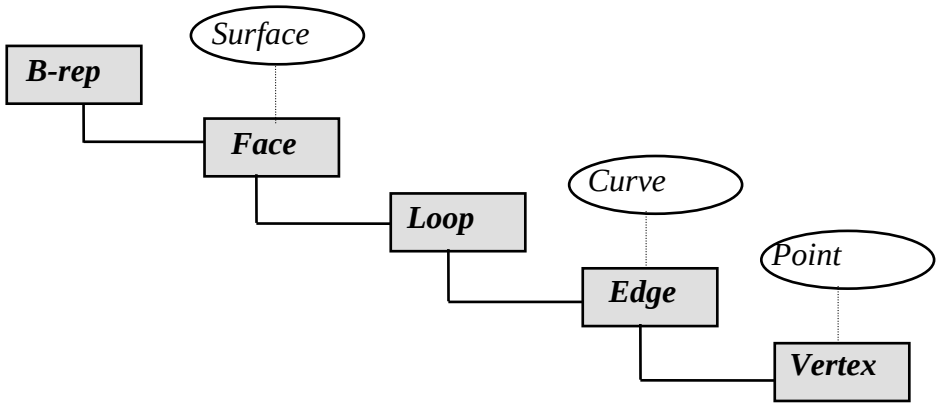


Figura 8. *Boundary representation* (B-rep)

Los elementos topológicos incluyen *Vertices*, *Edges* y *Faces* mientras que los elementos geométricos son especificaciones tales como *Points*, *Curves* o *Surfaces*. Los elementos geométricos dan información geométrica especifica para el elemento topológico. En un modelo B-rep los elementos topológicos son usados para expresar la conectividad del modelo.

Para acceder a los detalles del modelo se usan *traversers* que recorren los elementos topológicos. Los elementos topológicos disponibles son:

Face Una porción de superficie que forma parte de un sólido.
Definida por medio de la clase **`AcBrFace`**

Loop Colección de aristas (*Edges*) que encierran un *Face*.
Definida por medio de la clase **`AcBrLoop`**

Edge Porción de curva que es parte del loop que rodea una cara (*Face*).
Definida por medio de la clase **`AcBrEdge`**

Vertex El punto común de dos o más *Edges*.
Definida por medio de la clase **`AcBrVertex`**

La clase para el sólido completo es **`AcBrBrep`** (Ver figura 7).

Por lo general los *traversers* pueden recorrer los elementos topológicos inmediatamente inferiores que componen un elemento topológico. Otros recorren los elementos adyacentes en la jerarquía topológica. Una lista de los *traversers* **`AcBr`** se muestra a continuación:

`AcBrBrepFaceTraverser` Recorre las caras (*Faces*) de un modelo B-rep
`AcBrFaceLoopTraverser` Recorre los *Loops* de una cara (*Face*)
`AcBrLoopEdgeTraverser` Recorre las aristas (*Edges*) de un *Loop*
`AcBrLoopVertexTraverser` Recorre los Vértices (*Vertex*) de un *Loop*
`AcBrEdgeLoopTraverser` Recorre los *Loops* que contienen esta arista (*Edge*)

1.11 8. LIBRERIAS ACGI Y ACED

Aunque son de carácter muy diferente, estas librerías se agrupan aquí por su poca utilización en este proyecto.

a) La librería **`AcGi`**, como se explicó previamente es utilizada en el manejo de la interfaz gráfica de *AutoCAD* por algunas funciones miembro de la clase `AcDbEntity` para desplegar por pantalla las entidades **`AcDb`** creadas. Realiza operaciones con los *viewports* y manipula las entidades para asignar propiedades de visibilidad y regeneración.

b) La librería **`AcEd`** se utiliza para realizar tareas referentes al monitoreo del editor, con el fin de notificar eventos tales como el inicio, la terminación o cancelación de comandos. Además es útil para agregar nuevos comandos que se comportan igual a los demás comandos *AutoCAD*.

Los comandos *AutoCAD* son almacenados en grupos en el *Command stack*, que es definido por la clase `AcEdCommandStack`. Una instancia del *Command stack* se crea al inicio de cada sesión. Contiene todos los comandos nativos y los que la aplicación va añadiendo. La macro `acedRegCommands()`provee el acceso al *Command stack.*

Cuando se añade un comando también se asigna un nombre al grupo que lo contiene. Los nombres de los comandos dentro de un grupo dado deben ser únicos y los nombres de los grupos deben ser únicos. Sin embargo, diferentes aplicaciones pueden agregar comandos del mismo nombre, ya que el nombre del grupo evita la ambigüedad.
Usualmente los comandos se añaden uno a uno con `AcEdCommandStack::addCommand()`.
Igualmente los comandos se remueven uno a uno con `AcEdCommandStack::removeCmd()` o en grupo con `AcEdCommandStack::removeGroup()`.
En el siguiente numeral puede verse la utilización de los métodos expuestos para la gestión de comandos dentro de una aplicación.

1.12 9. ESTRUCTURA DE LAS APLICACIONES *ARX*

Una aplicación *ARX* es construida a partir de un proyecto C++ 4.0. El producto final es un archivo ejecutable sobre *AutoCAD*. Este archivo tiene extensión *arx*. Los pasos a seguir para construir un proyecto MSVC++ se encuentran en el *Manual del Programador AIS.*
Las aplicaciones *ARX* son dinámicamente enlazadas *(Dynamic Link Library DLL*).
AutoCAD hace llamadas dentro del módulo *ARX* a través de la función `acrxEntryPoint()`. Este reemplaza el main de los programas C o *ADS*. La implementación de `acrxEntryPoint()`es responsabilidad de la aplicación.
La función `acrxEntryPoint()` no solo sirve como punto de entrada a la aplicación desde *AutoCAD* sino también como una forma de pasar mensajes y retornar códigos de estado entre ellas. La función `acrxEntryPoint()`tiene el siguiente formato:

```
extern "C"
AcRx::AppRetCode acrxEntryPoint(AcRx::AppMsgCode msg,
                                void *pkt)
```

donde:

`msg`	Representa el mensaje enviado desde *ARX* a la aplicación.
`Pkt`	Guarda el valor del dato
`AppRetCode`	Contiene el código de estado retornado a *AutoCAD*

Dentro de la definición de `acrxEntryPoint()` se escribe una instrucción *switch* para descifrar los mensajes recibidos de *AutoCAD* y tomar la acción pertinente. De acuerdo a la acción tomada, se retorna un valor entero en el código de estado. Un ejemplo de la estructura del *switch* para la implementación de `acrxEntryPoint()` es como sigue:

```
switch (msg) {
    case AcRx::kInitAppMsg:
        initApp();
        break;
    case AcRx::kUnloadAppMsg:
        unloadApp();
        break;
    case AcRx::kLoadDwgMsg:
        ads_printf("Class1 Rx Sample -- kLoadDwgMsg\n");
        break;
    case AcRx::kUnloadDwgMsg:
        ads_printf("Class1 Rx Sample -- kUnloadDwgMsg\n");
```

```
        break;
    default:
        ads_printf("Class1 Rx Sample -- <unknown message %d>\n", msg);
    }
```

Además de la función básica `acrxEntryPoint()`, el programador puede registrar comandos que sean invocados cuando la aplicación se esté ejecutando. Para ello se utiliza el método
`AcEdCommandStack::addCommand()` cuyo formato es:

```
Acad::ErrorStatus
AcEdCommandStack::addCommand
                            (const char*      cmdGroupName,
                             const char*      cmdGlobalName,
                             const char*      cmdLocalName,
                             Adesk::Int32      commandFlags,
                             AcRxFunctionPtr   functionAddr);
```

donde:

`cmdGroupName`	Grupo al cual se va a añadir el comando. Si el grupo no existe se crea.
`CmdGlobalName`	Nombre global del comando a añadir. No portable a otros lenguajes.
`CmdLocalName`	Nombre local del nuevo comando. Portable a otros lenguajes *AutoCAD*.
`CommandFlags`	Banderas asociadas al comando. Valores posibles son `ACRX_CMD_TRANSPARENT o ACRX_CMD_MODAL`
`functionAddr`	Dirección de la función a ser ejecutada por el comando cuando es invocado por *AutoCAD*

Las aplicaciones *ARX* son cargadas dentro de *AutoCAD* usando diferentes métodos. El más rápido es mediante la utilización de la opción *Load* del comando *ARX*. Este comando también ofrece la opción *Unload*. Por defecto las aplicaciones no son descargables. Para hacer una aplicación descargable hay que asegurarse que ninguna aplicación cliente contenga apuntadores activos a su espacio en memoria.

ANEXO 2.

MANEJO DE LOS COMPONENTES LEXICOS DE DIGITLAB

Flex.l (Archivo para el Análisis Léxico)

ANÁLISIS LÉXICO. El analizador léxico para *DigitLAB* se elaboró por medio de un lenguaje de patrón-acción llamado *Flex*, que recibe como entrada un programa fuente (texto suministrado por el usuario) compuesto por caracteres de un alfabeto básico. La labor del analizador léxico consiste en separar los diversos *tokens* (unidades léxicas o palabras) que conforman el programa fuente clasificando cada uno de ellos en constantes, identificadores, palabras reservadas, operadores y separadores generados mediante diccionario.

Para adicionar un componente léxico en el analizador léxico de DigitLAB (archivo *flex.l*) se deben modoficar tres partes:

A. Declaraciones: Para la declaración de variables e identificadores se debe utilizar la notación de Backus-Naur (notación de gramáticas independientes del contexto). Aquí se declaran las definiciones regulares (simples y compuestas) que son proposiciones que se utilizan como componentes de las expresiones que aparecen en las reglas de traducción así:

```
/*****************************************************************************/
/**********                S I M P L E   D E C L A R A T I O N S             **********/
/*****************************************************************************/
/*  declaration          Backus-Naur Notation  */

    smbl_mtchar          [ \t\n]
    smbl_number          [0-9]
    smbl_letter          [a-zA-Z_]
    smbl_character       [ -_a-zA-Z]

/*****************************************************************************/
/**********              C O M P O S E D   D E C L A R A T I O N S           **********/
/*****************************************************************************/
/*  declaration          Backus-Naur Notation  */

    empty_in             {smbl_mtchar}+
    integer_in           {smbl_number}+
    real_in              {smbl_number}+(\.{smbl_number}+)?([E|e][+|-]?{smbl_number}+)?+
    word_in              ({smbl_letter}+{smbl_number}*)+
    char_in              (\'{smbl_character}\')
    string_in            (\")({smbl_letter}\:\\)?({word_in}(\\)?)+(\.{word_in})?(\")

/*****************************************************************************/
```

B. Reglas de traducción: Son proposiciones de la forma *"p_n" { acción $_n$ }*, donde *"p_i"* es una expresión regular y cada *{ acción $_i$ }* es la acción del analizador léxico. Cuando el patrón *"p_i"* concuerda con una palabra clave o reservada, se deben ejecutar dos acciones: (i) Entregar al *parser* (analizador sintáctico) la descripción del *token* que fue identificado para su procesamiento, copiando la variable *yytext* a la variable *yylval.union_text*. (ii) Colocar la información del *token* que fue identificado en la tabla de símbolos, retornando el identificador del texto patron identificado (en mayúsculas).

```
/*****************************************************************************/
/**********                   T R A D U C T I O N   R U L E S                **********/
/*****************************************************************************/

%%
{empty_in}              { /*Do nothing*/ }

"("                     { strcpy(yylval.union_text,yytext); return TK_LEFT_PAR;}
")"                     { strcpy(yylval.union_text,yytext); return TK_RIGHT_PAR;}
"["                     { strcpy(yylval.union_text,yytext); return TK_LEFT_BRK;}
"]"                     { strcpy(yylval.union_text,yytext); return TK_RIGHT_BRK;}
"{"                     { strcpy(yylval.union_text,yytext); return TK_LEFT_KEY;}
"}"                     { strcpy(yylval.union_text,yytext); return TK_RIGHT_KEY;}
","                     { strcpy(yylval.union_text,yytext); return TK_COMMA;}
"."                     { strcpy(yylval.union_text,yytext); return TK_DOT;}
";"                     { strcpy(yylval.union_text,yytext); return TK_SEMICOLON;}
":"                     { strcpy(yylval.union_text,yytext); return TK_CUOT_MARK;}
".."                    { strcpy(yylval.union_text,yytext); return TK_DOUBLE_DOT;}
"_"                     { strcpy(yylval.union_text,yytext); return TK_UNDER_SCR;}

"="                     { strcpy(yylval.union_text,yytext); return TK_equal;}
"+"                     { strcpy(yylval.union_text,yytext); return TK_plus;}
"-"                     { strcpy(yylval.union_text,yytext); return TK_minus;}
"*"                     { strcpy(yylval.union_text,yytext); return TK_mult;}
"/"                     { strcpy(yylval.union_text,yytext); return TK_div;}

"=="                    { strcpy(yylval.union_text,yytext); return TK_is; }
"!="                    { strcpy(yylval.union_text,yytext); return TK_is_not; }
"&&"                    { strcpy(yylval.union_text,yytext); return TK_and; }
"||"                    { strcpy(yylval.union_text,yytext); return TK_or; }
"!"                     { strcpy(yylval.union_text,yytext); return TK_not; }
">"                     { strcpy(yylval.union_text,yytext); return TK_greater; }
"<"                     { strcpy(yylval.union_text,yytext); return TK_less; }
">="                    { strcpy(yylval.union_text,yytext); return TK_greater_equal; }
"<="                    { strcpy(yylval.union_text,yytext); return TK_less_equal; }

"for"                   { strcpy(yylval.union_text,yytext); return TK_FOR;}
"while"                 { strcpy(yylval.union_text,yytext); return TK_WHILE;}
"if"                    { strcpy(yylval.union_text,yytext); return TK_IF;}
"else"                  { strcpy(yylval.union_text,yytext); return TK_ELSE;}
```

```
"pocketing"              { strcpy(yylval.union_text,yytext); return TK_pocketing;}
"build_coplanar"         { strcpy(yylval.union_text,yytext); return TK_build_coplanar;}
"build_sample"           { strcpy(yylval.union_text,yytext); return TK_build_sample;}
"build_resample"         { strcpy(yylval.union_text,yytext); return TK_build_resample;}
"build_levels"           { strcpy(yylval.union_text,yytext); return TK_build_levels;}
"link_levels"            { strcpy(yylval.union_text,yytext); return TK_link_levels;}
"screen_catch"           { strcpy(yylval.union_text,yytext); return TK_screen_catch;}
"see_database"           { strcpy(yylval.union_text,yytext); return TK_see_database;}
"erase_database"         { strcpy(yylval.union_text,yytext); return TK_erase_database;}
"save_database"          { strcpy(yylval.union_text,yytext); return TK_save_database;}
"load_data"              { strcpy(yylval.union_text,yytext); return TK_load_data;}
"vector_3d"              { strcpy(yylval.union_text,yytext); return TK_vector_3d;}
"best_plane"             { strcpy(yylval.union_text,yytext); return TK_best_plane;}
"load_range_pict"        { strcpy(yylval.union_text,yytext); return TK_load_range_pict;}
"range_to_shellset"      { strcpy(yylval.union_text,yytext); return TK_range_to_shellset;}
"shellset_to_facets"     { strcpy(yylval.union_text,yytext); return TK_shellset_to_facets;}
"shellset_to_vrml"       { strcpy(yylval.union_text,yytext); return TK_shellset_to_vrml;}
"create_list"            { strcpy(yylval.union_text,yytext); return TK_create_list;}
"exit"                   { strcpy(yylval.union_text,yytext); return TK_exit;}

{char_in}                { strcpy(yylval.union_text,yytext); return TK_CHAR;}
{string_in}              { strcpy(yylval.union_text,yytext); return TK_STRING;}
{word_in}                { strcpy(yylval.union_text,yytext); return TK_NOUN;}
{integer_in}             { strcpy(yylval.union_text,yytext); return TK_INTEGER;}
{real_in}                { strcpy(yylval.union_text,yytext); return TK_REAL;}

%%
```

C. Procedimientos auxiliares: Aquí se implementan todas las funciones adicionales que se puedan necesitar.

```
/*****************************************************************************/
/**********              A U X I L I A R   P R O C E D U R E S              **********/
/*****************************************************************************/

int yywrap( void )
{
    return 1;
}

int yyerror( char *messg )
{
    eafit_dgl_output_Error( messg,yytext );
    yyrestart( yyin );
    eafit_dgl_exec_Parser( );
    return 1;
}

/*****************************************************************************/
/*****************************************************************************/
```

ANEXO 3.

MANEJO DE LA GRAMATICA DE DIGITLAB

Bison.y (Archivo para el Análisis Sintactico)

ANÁLISIS SINTACTICO. El analizador sintáctico (*parser*) para *DigitLAB* se elaboró por medio de un generador de analizadores sintácticos llamado *Bison*, que lee una secuencia de *tokens* como su entrada y los agrupa usando las reglas gramaticales llamando al analizador léxico cada vez que éste necesita un nuevo *token*.

Para construir una gramática adecuada se deben analizar las relaciones entre asociatividad y precedencia de operadores, y, a partir de éstas, formular las reglas gramáticales que reconocen si una entrada es sintácticamente correcta. Si la entrada es válida, el resultado final es que el total de secuencias de *tokens* se reduce a una única agrupación de forma postfija, que es una notación en la que los operadores aparecen después de sus operandos.

Los componentes que posee una gramática son:

1. Un conjunto de componentes léxicos, denominados símbolos terminales (deben corresponder a las reglas de traducción del analizador léxico).
2. Un conjunto de símbolos no-terminales.
3. Un conjunto de producciones, donde cada producción consta de: (i) Un no-terminal, llamando lado izquierdo y (ii) Una secuencia de componentes léxicos y no terminales, o ambos, llamado lado derecho de la producción.

Para adicionar un componente de la gramática en el analizador sintáctico de DigitLAB (archivo *bison.y*) se deben modificar tres partes (si alguna de las partes cumple con los requisitos de la nueva adición, no se debe modificar):

1. Definicion del tipo de datos: Aquí se declara un *typedef union* que contiene los tipos de datos que puede tener un *token*.

```
/*****************************************************************************/
/**********   D E F I N I T I O N   O F   D A T A   T Y P E   * U N I O N *   **********/
/*****************************************************************************/

%union
{
    double  union_number;
    char union_text[SPEECH_SIZE];
    struct  fb_token_item *union_token;
}
```

2. Declaraciones: Aquí se declaran los conjuntos de símbolos terminales y no-terminales asi: %token <tipo> símbolo.

```
/*****************************************************************************/
/**********                T E R M I N A L   S I M B O L S                **********/
/*****************************************************************************/

%token <union_text>          TK_FOR
%token <union_text>          TK_WHILE
%token <union_text>          TK_IF
%token <union_text>          TK_ELSE

%token <union_text>          TK_LEFT_PAR
%token <union_text>          TK_RIGHT_PAR
%token <union_text>          TK_LEFT_BRK
%token <union_text>          TK_RIGHT_BRK
%token <union_text>          TK_LEFT_KEY
%token <union_text>          TK_RIGHT_KEY
%token <union_text>          TK_DOT
%token <union_text>          TK_COMMA
%token <union_text>          TK_SEMICOLON
%token <union_text>          TK_CUOT_MARK
%token <union_text>          TK_DOUBLE_DOT
%token <union_text>          TK_UNDER_SCR

%token <union_text>          TK_is
%token <union_text>          TK_is_not
%token <union_text>          TK_and
%token <union_text>          TK_or
%token <union_text>          TK_not
%token <union_text>          TK_greater
%token <union_text>          TK_less
%token <union_text>          TK_greater_equal
%token <union_text>          TK_less_equal

%token <union_text>          TK_equal
%token <union_text>          TK_plus
%token <union_text>          TK_minus
%token <union_text>          TK_mult
%token <union_text>          TK_div
```

```
%token <union_text>          TK_pocketing
%token <union_text>          TK_build_coplanar
%token <union_text>          TK_build_sample
%token <union_text>          TK_build_resample
%token <union_text>          TK_build_levels
%token <union_text>          TK_link_levels
%token <union_text>          TK_screen_catch
%token <union_text>          TK_see_database
%token <union_text>          TK_erase_database
%token <union_text>          TK_save_database
%token <union_text>          TK_load_data
%token <union_text>          TK_vector_3d
%token <union_text>          TK_best_plane
%token <union_text>          TK_load_range_pict
%token <union_text>          TK_range_to_shellset
%token <union_text>          TK_shellset_to_facets
%token <union_text>          TK_shellset_to_vrml
%token <union_text>          TK_create_list
%token <union_text>          TK_exit

%token <union_text>          TK_NOUN
%token <union_text>          TK_INTEGER
%token <union_text>          TK_REAL
%token <union_text>          TK_STRING
%token <union_text>          TK_CHAR
```

```
/*****************************************************************************/
/**********              N O - T E R M I N A L   S I M B O L S          **********/
/*****************************************************************************/

%type <union_token>          seq_of_composed_clauses
%type <union_token>          composed_clause
%type <union_token>          line_clause
%type <union_token>          for_clause
%type <union_token>          while_clause
%type <union_token>          if_clause
%type <union_token>          function_clause
%type <union_token>          assign_clause
%type <union_token>          functor
%type <union_token>          single_assign
%type <union_token>          multi_assign
%type <union_token>          boolean_expression
%type <union_token>          bool_left
%type <union_token>          bool_rigth
%type <union_number>         list_of_left_args
%type <union_number>         list_of_rigth_args
%type <union_token>          left_expression
%type <union_token>          rigth_expression
%type <union_token>          indexed_left_var
%type <union_token>          indexed_right_var
%type <union_token>          list_of_index
%type <union_text>           index_expr
%type <union_text>           costant_var
%type <union_token>          range_var
%type <union_text>           indexes
%type <union_token>          math_expression
%type <union_token>          term_expression
%type <union_token>          factor_expression
```

C. Reglas de traducción: Donde cada regla (delimitada por %%) consta de una producción de la gramática y la acción semántica asociada. Por notación las producciones con el mismo no-terminal del lado izquierdo pueden tener sus lados derechos agrupados, con los lados derechos alternativos separados por el símbolo |, que se leerá “o”.

```
/*****************************************************************************/
/**********                  T R A D U C T I O N   R U L E S                **********/
/*****************************************************************************/

%%

seq_of_composed_clauses  : seq_of_composed_clauses composed_clause
                         | composed_clause
                         ;

composed_clause  : line_clause TK_CUOT_MARK { eafit_dgl_process_LineClause( );}
                 | for_clause
                 | while_clause
                 | if_clause
                 | TK_LEFT_KEY seq_of_composed_clauses TK_RIGHT_KEY
                 | TK_exit TK_CUOT_MARK { return 0;}
                 ;

line_clause  : function_clause
             | assign_clause
             ;

for_clause   : TK_FOR TK_LEFT_PAR single_assign TK_SEMICOLON boolean_expression TK_SEMICOLON
                                                  single_assign TK_RIGHT_PAR composed_clause
             ;

while_clause : TK_WHILE TK_LEFT_PAR boolean_expression TK_RIGHT_PAR composed_clause
                 ;

if_clause    : TK_IF TK_LEFT_PAR boolean_expression TK_RIGHT_PAR composed_clause
                 | TK_IF TK_LEFT_PAR boolean_expression TK_RIGHT_PAR composed_clause TK_ELSE
                                                                        composed_clause
             ;

function_clause  : functor TK_LEFT_PAR list_of_rigth_args TK_RIGHT_PAR {
eafit_dgl_add_Token($1);}
                 | functor TK_LEFT_PAR TK_RIGHT_PAR { eafit_dgl_add_Token($1);}
                 ;

assign_clause    : single_assign
                 | multi_assign
                 ;

functor  : TK_pocketing { $$ = eafit_dgl_constr_Token(
                                      TK_pocketing,$1,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
         | TK_build_coplanar { $$ = eafit_dgl_constr_Token(
                                    TK_build_coplanar,$1,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
         | TK_build_sample { $$ = eafit_dgl_constr_Token(
                                      TK_build_sample,$1,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
         | TK_build_resample { $$ = eafit_dgl_constr_Token(
                                    TK_build_resample,$1,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
```

```
        | TK_build_levels { $$ = eafit_dgl_constr_Token(
                                    TK_build_levels,$1,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
        | TK_link_levels { $$ = eafit_dgl_constr_Token(
                                     TK_link_levels,$1,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
        | TK_screen_catch { $$ = eafit_dgl_constr_Token(
                                    TK_screen_catch,$1,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
        | TK_see_database { $$ = eafit_dgl_constr_Token(
                                    TK_see_database,$1,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
        | TK_erase_database { $$ = eafit_dgl_constr_Token(
                                  TK_erase_database,$1,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
        | TK_save_database { $$ = eafit_dgl_constr_Token(
                                   TK_save_database,$1,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
        | TK_load_data { $$ = eafit_dgl_constr_Token(
                                     TK_load_data,$1,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
        | TK_vector_3d { $$ = eafit_dgl_constr_Token(
                                     TK_vector_3d,$1,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
        | TK_best_plane { $$ = eafit_dgl_constr_Token(
                                    TK_best_plane,$1,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
        | TK_load_range_pict { $$ = eafit_dgl_constr_Token(
                                TK_load_range_pict,$1,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
        | TK_range_to_shellset { $$ = eafit_dgl_constr_Token(
                                 TK_range_to_shellset,$1,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
        | TK_shellset_to_facets { $$ = eafit_dgl_constr_Token(
                               TK_shellset_to_facets,$1,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
        | TK_shellset_to_vrml { $$ = eafit_dgl_constr_Token(
                                  TK_shellset_to_vrml,$1,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
        | TK_create_list { $$ = eafit_dgl_constr_Token(
                                     TK_create_list,$1,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
        | TK_NOUN { $$ = eafit_dgl_constr_Token( TK_NOUN,$1,TRUE,NULL_INT,NULL_REAL,FUNCTOR
                                                                                    );}
        ;

single_assign    : left_expression TK_equal rigth_expression { eafit_dgl_add_TokenByInfo(
TK_equal,$2,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
                 ;

multi_assign : TK_LEFT_BRK list_of_left_args TK_RIGHT_BRK TK_equal function_clause {
eafit_dgl_add_TokenByInfo( TK_equal,$4,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
                 ;

boolean_expression   : bool_left TK_is bool_rigth
                     | bool_left TK_is_not bool_rigth
                     | bool_left TK_and bool_rigth
                     | bool_left TK_or bool_rigth
                     | bool_left TK_not bool_rigth
                     | bool_left TK_greater bool_rigth
                     | bool_left TK_less bool_rigth
                     | bool_left TK_greater_equal bool_rigth
                     | bool_left TK_less_equal bool_rigth
                     | bool_left
                     ;

bool_left    : left_expression
             | function_clause
             ;

bool_rigth   : TK_NOUN
             | TK_REAL
             | TK_INTEGER
             | function_clause
             ;

list_of_left_args    : list_of_left_args TK_COMMA left_expression
                     | left_expression
                     ;
```

```
list_of_rigth_args   : list_of_rigth_args TK_COMMA rigth_expression
                     | rigth_expression
                     ;

left_expression  : TK_NOUN { eafit_dgl_add_TokenByInfo(
                                          TK_NOUN,$1,TRUE,NULL_INT,NULL_REAL,VAR_LEFT );}
                 | indexed_left_var
                 ;

rigth_expression : TK_STRING { eafit_dgl_add_TokenByInfo(
                                       TK_STRING,$1,TRUE,NULL_INT,NULL_REAL,VAR_RIGHT );}
                 | math_expression
                 | indexed_right_var
                 | function_clause
                 | assign_clause
                 ;

indexed_left_var : TK_NOUN TK_LEFT_BRK list_of_index TK_RIGHT_BRK {
            eafit_dgl_add_TokenByInfo( TK_NOUN,$1,TRUE,NULL_INT,NULL_REAL,VAR_LEFT_INDEX );}
                 ;

indexed_right_var    : TK_NOUN TK_LEFT_BRK list_of_index TK_RIGHT_BRK {
           eafit_dgl_add_TokenByInfo( TK_NOUN,$1,TRUE,NULL_INT,NULL_REAL,VAR_RIGHT_INDEX );}
                 ;

list_of_index    : index_expr TK_COMMA list_of_index { eafit_dgl_add_TokenByInfo(
                                      TK_NOUN,$1,TRUE,NULL_INT,NULL_REAL,VAR_LIST_INDEX );}
                 | index_expr { eafit_dgl_add_TokenByInfo(
                                        TK_NOUN,$1,TRUE,NULL_INT,NULL_REAL,VAR_INDEX );}
                 ;

index_expr   : costant_var{ strcpy($$,$1);}
             | range_var
             ;

costant_var  : indexes { strcpy($$,$1);}
             | TK_CUOT_MARK
             ;

range_var    : indexes TK_DOUBLE_DOT indexes
             ;

indexes  : TK_NOUN { strcpy($$,$1);}
         | TK_INTEGER
         | math_expression
         ;

math_expression  : math_expression TK_plus term_expression { eafit_dgl_add_TokenByInfo(
                                           TK_plus,$2,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
                 | math_expression TK_minus term_expression { eafit_dgl_add_TokenByInfo(
                                           TK_minus,$2,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
                 | term_expression
                 ;

term_expression  : term_expression TK_mult factor_expression { eafit_dgl_add_TokenByInfo(
                                            TK_mult,$2,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
                 | term_expression TK_div factor_expression { eafit_dgl_add_TokenByInfo(
                                             TK_div,$2,TRUE,NULL_INT,NULL_REAL,FUNCTOR );}
                 | factor_expression
                 ;
```

```
factor_expression    : TK_LEFT_PAR math_expression TK_RIGHT_PAR
                     | TK_CHAR { eafit_dgl_add_TokenByInfo(
                                        TK_CHAR,$1,TRUE,NULL_INT,NULL_REAL,VAR_RIGHT );}
                     | TK_NOUN { eafit_dgl_add_TokenByInfo(
                                        TK_NOUN,$1,TRUE,NULL_INT,NULL_REAL,VAR_RIGHT );}
                     | TK_INTEGER { eafit_dgl_add_TokenByInfo(
                                      TK_INTEGER,$1,TRUE,NULL_INT,NULL_REAL,VAR_RIGHT );}
                     | TK_REAL { eafit_dgl_add_TokenByInfo(
                                        TK_REAL,$1,TRUE,NULL_INT,NULL_REAL,VAR_RIGHT );}
                     | TK_minus TK_REAL
                     | TK_minus TK_INTEGER
                     | TK_minus TK_NOUN
                     ;

%%
```

ANEXO 4.

PLANTILLA PARA LA ADICION DE FUNCIONES EN DIGITLAB

Functors.cxx

Se creó una plantilla en *C/C++*, que sirve de base para implementar en forma unificada cualquier función. La base de este procedimiento es la de retirar del *stack* los parámetros que esta función necesita y efectuar las acciones de la fase respectiva (fase de chequeo y fase de ejecución) para obtener la ejecución de la función.

Aquí se especifica un comprobador de tipos para un lenguaje simple en el que se debe declarar el tipo de cada identificador antes de que el identificador se utilice.

Para el proceso de ejecución de la función dada por el usuario se implementaron dos fases:

1. Fase de chequeo CHECK: En esta fase se hace una verificación de tipos, con el fin de detectar errores de forma preventiva.
2. Fase de ejecución EXEC: En esta fase se hace la ejecución de las funciones que el usuario ha dispuesto. Esta parte se ejecuta únicamente si el proceso de CHECK entregó un resultado exitoso.

La plantilla aplicada a las funciones de *DigitLAB* esta compuesta por:

1. Declaración de la función:

```
/*****************************************************************************/
/**********                   P A R S E R    F U N C T O R S                 **********/
/*****************************************************************************/
//---------------------------------------------------------------------------//
//  FUNCTION NAME    :   [eafit_dgl_func_######]
//  MADE BY          :   [RSS] R. Sebastian Schrader       DATE: 10/DIC/1998
//  MODIFIED BY      :   [   ] Authors_Name                DATE: --/---/----
//  PLATFORM     :   ARX[ ]     MDL[ ]     M_MDL[ ]     AIS[x]     C/C++[ ]
//  DESCRIPTION      :
//  TYPE OF RETURN   :   long:  return_code
//  OBSERVATIONS :
//---------------------------------------------------------------------------//
long
    eafit_dgl_func_######
    (
        long    process_status              //IN: Process Status
    )
//---------------------------------------------------------------------------//
/*****************************************************************************/
```

2. Declaración de las variables generales:

```
/*****************************************************************************************/
//---------------------------------------------------------------------------------------//
{
// Stack Parameter
    t_list_item* stack_item               = NULL;
    char         stack_item_name[SPEECH_SIZE];
// Wrap to Verify the Object Types
    t_list_item* type_item                = NULL;
//---------------------------------------------------------------------------------------//
/*****************************************************************************************/
```

3. Declaración de las variables particulares:

3.1. (#) Parametro (Selection String):

```
/*****************************************************************************************/
//---------------------------------------------------------------------------------------//
//  (#) Parameter                ->  Selection String
    static int       static_result_#          = FALSE;
    static int       select_str               = NULL;

//---------------------------------------------------------------------------------------//
//  (#) Parameter                ->  Selection String
//---------------------------------------------------------------------------------------//
// Remove the Parameter from Stack:
    return_code = eafit_dgl_remove_ItemHeadDataBaseList ( stack_list, &stack_item );
    if( return_code is_not CI_ER_NONE ) goto return_process;
    strcpy( stack_item_name,(stack_item->str_field) );

// Found the Parameter in DigiLab Data Base:
    if ( process_status is CHECK )
    {
        // Create an Virtual Item with the types of the compatation.
        type_item = eafit_dgl_get_TypeList( NULL, ZERO, NULL, ZERO, OBJECT_NOUN, ONE );

        // Compare the Stack Item with the Virtual Item
        eafit_dgl_compare_TypeItems( stack_item, type_item, &static_result_# );
        if ( static_result_# )
        {
            if ( not strcmpi(stack_item_name,"Selection_Type_1") )
                select_str = TYPE_1;
            else if ( not strcmpi(stack_item_name,"Selection_Type_2") )
                select_str = TYPE_2;
            else
            {
                return_code = CI_ER_EINCMP;
                strcpy( return_str,stack_item_name );
                goto return_process;
            }
        }
```

```
        }
        else if( not static_result_# )
        {
            return_code = CI_ER_EINCMP;
            strcpy( return_str,stack_item_name );
            goto return_process;
        }
    }

//-------------------------------------------------------------------------------------//
/*************************************************************************************/
```

3.2. (#) Parametro (Number <Integer or Real>):

```
/*************************************************************************************/
//-------------------------------------------------------------------------------------//
//  (#) Parameter                    ->  Number ( Integer or Real )
    static int      static_result_#         = FALSE;
    int             result_int_#        = ZERO;
    int             result_real#        = ZERO;
    double          number                  = ZERO;

//-------------------------------------------------------------------------------------//
//  (#) Parameter                    ->  Number ( Integer or Real )
//-------------------------------------------------------------------------------------//
// Remove the Parameter from Stack:
    return_code = eafit_dgl_remove_ItemHeadDataBaseList ( stack_list, &stack_item );
    if( return_code is_not CI_ER_NONE ) goto return_process;
    strcpy( stack_item_name,(stack_item->str_field) );

// Found the Parameter in DigiLab Data Base:
    eafit_dgl_macro_process_CheckExec
    (
     /*Status*/ process_status,
     /*Check*/  type_item = eafit_dgl_get_TypeList( NULL, ZERO, NULL, ZERO, OBJECT_NOUN,
ONE );
                eafit_dgl_compare_TypeItems( stack_item, type_item, &static_result_# );
                if( static_result_# )
                {
                    stack_item = eafit_dgl_get_CheckItemByDigiListItem( stack_item );
                    if( not stack_item )
                    {
                        return_code = CI_ER_NOSCHE;
                        strcpy( return_str,stack_item_name );
                        goto return_process;
                    }
                },
     /*Exec*/   if( static_result_# )
                {
                    stack_item = eafit_dgl_get_ExecItemByDigiListItem( stack_item );
                }
    );
```

```
// Parameter Type Comparation and Initialization:
    eafit_dgl_macro_process_CheckExec
    (
     /*Status*/ process_status,
     /*Check*/  type_item = eafit_dgl_get_TypeList( NULL, ZERO, NULL, ZERO, OBJECT_INT, ONE
);
                eafit_dgl_compare_TypeItems( stack_item, type_item, &result_int_# );
                type_item = eafit_dgl_get_TypeList( NULL, ZERO, NULL, ZERO, OBJECT_REAL,
ONE );
                eafit_dgl_compare_TypeItems( stack_item, type_item, &result_real# );
                if( not ( result_int_# or result_real# ) )
                {
                    return_code = CI_ER_EINCMP;
                    strcpy( return_str,stack_item_name );
                    goto return_process;
                },
     /*Exec*/   if( static_result_# )
                {
                    stack_item = eafit_dgl_get_ExecItemByDigiListItem( stack_item );
                }
    );

// Parameter Type Comparation and Initialization:
    eafit_dgl_macro_process_CheckExec
    (
     /*Status*/ process_status,
     /*Check*/  type_item = eafit_dgl_get_TypeList( NULL, ZERO, NULL, ZERO, OBJECT_INT, ONE
);
                eafit_dgl_compare_TypeItems( stack_item, type_item, &result_int_# );
                type_item = eafit_dgl_get_TypeList( NULL, ZERO, NULL, ZERO, OBJECT_REAL,
ONE );
                eafit_dgl_compare_TypeItems( stack_item, type_item, &result_real# );
                if( not ( result_int_# or result_real# ) )
                {
                    return_code = CI_ER_EINCMP;
                    strcpy( return_str,stack_item_name );
                    goto return_process;
                },
     /*Exec*/   switch ( stack_item->int_field )
                {
                    case OBJECT_INT:
                    {   number = (*((long*)(stack_item->item)));
                        number = (real)number;  break;   }
                    case OBJECT_REAL:
                    {   number = (*((real*)(stack_item->item)));  break;   }
                    default:
                    {   report("eafit_dgl_func_######(): Type not march <INT-REAL>.");
break;   }
                }
    );

//-------------------------------------------------------------------------------------//
/*************************************************************************************/
```

3.3. (#) Parametro (Vector):

```
/*****************************************************************************************/
//---------------------------------------------------------------------------------------//
//  (#) Parameter                    ->  Vector
    static int       static_result_#          = FALSE;
    int              result_vect_#            = ZERO;
    cit_vect         vector;

//---------------------------------------------------------------------------------------//
//  (#) Parameter                    ->  Vector
//---------------------------------------------------------------------------------------//
// Remove the Parameter from Stack:
    return_code = eafit_dgl_remove_ItemHeadDataBaseList ( stack_list, &stack_item );
    if( return_code is_not CI_ER_NONE ) goto return_process;
    strcpy( stack_item_name,(stack_item->str_field) );
// Found the Parameter in DigiLab Data Base:
    eafit_dgl_macro_process_CheckExec
    (
     /*Status*/ process_status,
     /*Check*/  type_item = eafit_dgl_get_TypeList( NULL, ZERO, NULL, ZERO, OBJECT_NOUN,
ONE );
                eafit_dgl_compare_TypeItems( stack_item, type_item, &static_result_# );
                if( static_result_# )
                {
                    stack_item = eafit_dgl_get_CheckItemByDigiListItem( stack_item );
                    if( not stack_item )
                    {
                        return_code = CI_ER_NOSCHE;
                        strcpy( return_str,stack_item_name );
                        goto return_process;
                    }
                },
     /*Exec*/   if( static_result_# )
                {
                    stack_item = eafit_dgl_get_ExecItemByDigiListItem( stack_item );
                }
    );

// Parameter Type Comparation and Initialization:
    eafit_dgl_macro_process_CheckExec
    (
     /*Status*/ process_status,
     /*Check*/   type_item = eafit_dgl_get_TypeList( NULL, ZERO, NULL, ZERO, OBJECT_VEC, ONE
);
                eafit_dgl_compare_TypeItems( stack_item, type_item, &result_vect_# );
                if( not result_vect_# )
                {
                    return_code = CI_ER_EINCMP;
                    strcpy( return_str,stack_item_name );
                    goto return_process;
                },
     /*Exec*/   copy_vector( normal_vector,(*((cit_vect*)(stack_item->item))) )
    );
//---------------------------------------------------------------------------------------//
/*****************************************************************************************/
```

3.4. (#) Parametro (List of Entities):

```
/*****************************************************************************************/
//---------------------------------------------------------------------------------------//
//  (#) Parameter                    ->  List of Entities
    static int      static_result_#          = FALSE;
    int             result_list_#            = ZERO;
    int             entity_type              = ZERO;

    t_list_wrap* wrap_item               = NULL;
    cip_hlst    entities_list            = NULL;
    long        entities_num         = ZERO;
    int             entities_type            = ZERO;

//---------------------------------------------------------------------------------------//
//  (#) Parameter                    ->  List of Entities
//  For a List of Especific Entities Change CI_TYEN for the Entity Code.
//---------------------------------------------------------------------------------------//
// Remove the Parameter from Stack:
    return_code = eafit_dgl_remove_ItemHeadDataBaseList ( stack_list, &stack_item );
    if( return_code is_not CI_ER_NONE ) goto return_process;
    strcpy( stack_item_name,(stack_item->str_field) );


// Found the Parameter in DigiLab Data Base:
    eafit_dgl_macro_process_CheckExec
    (
     /*Status*/ process_status,
     /*Check*/  type_item = eafit_dgl_get_TypeList( NULL, ZERO, NULL, ZERO, OBJECT_NOUN,
ONE );
                eafit_dgl_compare_TypeItems( stack_item, type_item, &static_result_# );
                if( static_result_# )
                {
                    stack_item = eafit_dgl_get_CheckItemByDigiListItem( stack_item );
                    if( not stack_item )
                    {
                        return_code = CI_ER_NOSCHE;
                        strcpy( return_str,stack_item_name );
                        goto return_process;
                    }
                },
     /*Exec*/   if( static_result_# )
                {
                    stack_item = eafit_dgl_get_ExecItemByDigiListItem( stack_item );
                }
    );

// Parameter Type Comparation and Initialization:
//  (for an Especific Entity Change CI_TYEN for the Entity Code)
    eafit_dgl_macro_process_CheckExec
    (
     /*Status*/ process_status,
```

```
     /*Check*/   type_item = eafit_dgl_get_TypeList( NULL, ZERO, OBJECT_LIST_WRAP, ONE,
CI_TYEN, ONE );
                 eafit_dgl_compare_TypeItems( stack_item, type_item, &result_list_# );
                 if( not result_list_# )
                 {
                     return_code = CI_ER_EINCMP;
                     strcpy( return_str,stack_item_name );
                     goto return_process;
                 }
                 eafit_dgl_get_EntityTypeByTypeItem( stack_item, &entity_type ),
     /*Exec*/   wrap_item    = (t_list_wrap*)stack_item->item;
                 entities_list    = (cip_hlst)(wrap_item->e_ptr);
                 entities_num = (long)(wrap_item->size);
                 entities_type    = (long)(wrap_item->type)
    );
//-------------------------------------------------------------------------------------//
/*************************************************************************************/
```

3.5. (#) Parametro (List of Entities - List of lists of Entities):

```
/*************************************************************************************/
//-------------------------------------------------------------------------------------//
//  (#) Parameter                    ->  List of Entities   ->List of lists of Entities
    static int       static_result_#          = FALSE;
    t_list*          wraps_list               = NULL;

    cip_hlst     points_list              = NULL;
    long         num_points               = ZERO;

//-------------------------------------------------------------------------------------//
//  (#) Parameter                    ->  List of Entities   ->List of lists of Entities
//  For a List of Especific Entities Change CI_TYEN for the Entity Code.
//-------------------------------------------------------------------------------------//
// Remove the Parameter from Stack:
    return_code = eafit_dgl_remove_ItemHeadDataBaseList ( stack_list, &stack_item );
    if( return_code is_not CI_ER_NONE ) goto return_process;
    strcpy( stack_item_name,(stack_item->str_field) );

// Found the Parameter in DigiLab Data Base:
    eafit_dgl_macro_process_CheckExec
    (
     /*Status*/ process_status,
```

```
     /*Check*/  type_item = eafit_dgl_get_TypeList( NULL, ZERO, NULL, ZERO, OBJECT_NOUN,
ONE );
                 eafit_dgl_compare_TypeItems( stack_item, type_item, &static_result_# );
                 if( static_result_3 )
                 {
                     stack_item = eafit_dgl_get_CheckItemByDigiListItem( stack_item );
                     if( not stack_item )
                     {
                         return_code = CI_ER_NOSCHE;
                         strcpy( return_str,stack_item_name );
                         goto return_process;
                     }
                 },
     /*Exec*/   if( static_result_3 )
                 {
                     stack_item = eafit_dgl_get_ExecItemByDigiListItem( stack_item );
                 }
    );

// Parameter Type Comparation and Initialization:
    eafit_dgl_macro_process_CheckExec
    (
     /*Status*/ process_status,
     /*Check*/   type_item = eafit_dgl_get_TypeList( NULL, ZERO, OBJECT_LIST_WRAP, ONE,
CI_TYEN, ONE );
                 eafit_dgl_compare_TypeItems( stack_item, type_item, &result_1 );
                 type_item = eafit_dgl_get_TypeList( OBJECT_LIST, ONE, OBJECT_LIST_WRAP,
ONE, CI_TYEN, ONE );
                 eafit_dgl_compare_TypeItems( stack_item, type_item, &result_2 );
                 if( not ( result_1 or result_2 ) )
                 {
                     return_code = CI_ER_EINCMP;
                     strcpy( return_str,stack_item_name );
                     goto return_process;
                 },
     /*Exec*/   switch ( stack_item->int_field )
                 {
                     case OBJECT_LIST:
                     {
                         wraps_list  = (t_list*)stack_item->item;
                         break;
                     }
                     case OBJECT_LIST_WRAP:
                     {
                         wraps_list  = eafit_list_constr_List( NULL_STR );
                         eafit_list_add_ItemTail ( wraps_list, stack_item );
                         break;
                     }
                     default:
                     {   report("eafit_dgl_func_mk_coplanar(): Type not march.");  break;
}
                 }
    );
//-------------------------------------------------------------------------------------//
/*************************************************************************************/
```

4. (#) Proceso de Ejecucion:

```
/*****************************************************************************************/
//---------------------------------------------------------------------------------------//
//  Execution Process
//---------------------------------------------------------------------------------------//
    eafit_dgl_macro_process_CheckExec
    (
     /*Status*/ process_status,

     /*Check*/  //Create an Item with the features of the Resulta Object
                type_item = eafit_dgl_get_TypeList(
                            OBJECT_LIST,         //IN: Object List
                            ONE,                 //IN: Number of Object List
                            OBJECT_LIST_WRAP,    //IN: Object Wrap
                            ONE,                 //IN: Number of Object Wrap
                            CI_TYEN,             //IN: Object General
                                                 //    For an Especific Entity
                                                 //    change CI_TYEN for the Entity Code.
                            ONE                  //IN: Number of Object in General
                                                   );
                eafit_dgl_add_ItemToStackByItem( type_item );
                free( stack_item ),
     /*Exec*/   eafit_ciio_SayMessageInProcess( "Processing... ########" );
                eafit_dgl_add_ItemToStackByItem( eafit_list_constr_Item(
(t_void_ptr)dgl_level_list,
                                                  NULL_STR,OBJECT_LIST, VAR_RIGHT,
PARTICULAR ) );
                free( stack_item )
    );
```

5. Proceso de Retorno:

```
/*****************************************************************************************/
//---------------------------------------------------------------------------------------//
//  Return Process
//---------------------------------------------------------------------------------------//
    return_process:
    {
        eafit_dgl_return ( return_code,return_str );
        return ( return_code );
    }
}
//---------------------------------------------------------------------------------------//
//*****************************************************************************************//
```

ANEXO 5.

CODIGOS DE ERROR CAM-I

Errores del Sistema:

CI_ER_NONE	No error encountered.
CI_ER_MEM	Error allocating memory.
CI_ER_MODELR	Error connecting to modeler.
CI_ER_USER	Error in user ID, password, or priviledges.

Funciones de error y Argumentos:

CI_ER_FUNC	Unknown function error.
CI_ER_DUPENT	Both entities input are same ent.
CI_ER_EMTLST	Empty list.
CI_ER_REQENT	Entity required but null given.
CI_ER_NOTYP	Entity type not supported.
CI_ER_EINCMP	Entity types incompatible.
CI_ER_NOFUNC	Function not supported.
CI_ER_HNDL	Illegal handle given.
CI_ER_INVCHR	Invalid characters in string.
CI_ER_ETYPE	Invalid entity type for function.
CI_ER_LSTYP	Invalid list type.
CI_ER_MNMX	Invalid min/max given.
CI_ER_INDXR	List index out of range.
CI_ER_NOMODL	No model active.
CI_ER_NOSCHE	No such entity.
CI_ER_OPREQ	Optional argument required.
CI_ER_CHARX	Too many characters for string.
CI_ER_UNTYP	Unknown entity type.
CI_ER_VALUE	Value out of range.
CI_ER_ZERVAL	Zero (or negative) value given.

Errores de Modelos de Diseño:

CI_ER_MODEL	Uknown file or model error.
CI_ER_CREMOD	Cannot create model.
CI_ER_REPLACE	Did not replace existing model.
CI_ER_CHKPT	Invalid checkpoint.
CI_ER_FILNAM	Invalid file name.
CI_ER_FILTYP	Invalid file type.
CI_ER_RDONLY	Model is read-only.
CI_ER_NOSAV	No save since last change.
CI_ER_NOSCHF	No such file.
CI_ER_NOSCHM	No such model.
CI_ER_UNDO	Nothing to undo.

Errores de Entidad y Pegado:

CI_ER_ENTITY	Unknown Entity or attached data error.
CI_ER_NOSCHA	Attribute class not found.
CI_ER_CONST	Entity has constructors (not on list).
CI_ER_DEPENT	Entity has dependencies.
CI_ER_DUPASC	Entity is already associated for this class
CI_ER_NOEASC	Entity is not associated
CI_ER_SENSE	Sense does not agree with other input.
CI_ER_UNDEF	Inquired LOGICAL is not defined.
CI_ER_ATCLTY	Invalid attribute class-type combo
CI_ER_ATTTYP	Invalid attribute data type
CI_ER_INVFLD	Invalid field value (outside range).
CI_ER_LABEL	Invalid label.
CI_ER_DUPLAB	Label already exists.
CI_ER_NODATA	No attribute set for that field number.
CI_ER_NODEP	No dependent entity found.
CI_ER_NOASOC	No entities associated for this class.
CI_ER_NOLAB	No such labeled entity.
CI_ER_RESATT	Reserved attribute class

Errores de Estructura de Modelos:

CI_ER_MODST	Unknown model structure error.
CI_ER_MSREF	Cannot find model structure ref.
CI_ER_MSREC	Model Structure recursion.

Errores Geométricos:

CI_ER_GEO	Unknown geometric error.
CI_ER_ZCOEF	Coefficients have inappropriate zero value(s).
CI_ER_ANGLE	Angle out of range.
CI_ER_COEFR	Coefficients out of range.
CI_ER_CONEN	Cone = cylinder (angle too small).
CI_ER_CONEX	Cone = plane (angle too great).
CI_ER_CRVSRF	Curve does not lie on surface.
CI_ER_DIRPRL	Dir parallel to plane (of curve).
CI_ER_DIRPRP	Dir perpendicular to plane.
CI_ER_CRVPT	Curve degenerates to a point.
CI_ER_CRVAX	Curve intersects axis.
CI_ER_NOTRIM	Curve/Surface is not trimmed at this time.
CI_ER_CRVCLO	Curve not closed.
CI_ER_CRVPLN	Curve not in plane.
CI_ER_NOINT	Curves/Surfaces do not intersect.
CI_ER_EPLNE	Entity not planar.
CI_ER_GEOCMP	Geometries are not comprable.
CI_ER_DEGEN	Geometry is already degenerate.
CI_ER_IMGRY	Imaginary value(s) result.
CI_ER_SWEEP	Impermissible sweep.
CI_ER_CURVE	Invalid Curve given.
CI_ER_MATRIX	Invalid transform matrix.
CI_ER_VECTOR	Invalid vector.
CI_ER_WEITS	Invalid weights.
CI_ER_KNOTS	Knot type is not supproted.
CI_ER_NOSUCV	Not a Surface_Curve.
CI_ER_PNTCRV	Point is/is not on curve.
CI_ER_PNTSRF	Point is/is not on surface.
CI_ER_NOPROJ	Point is not a projection.
CI_ER_PNTPLN	Point not in plane.
CI_ER_PRFPRP	Profile curve is perp to axis of rev.
CI_ER_PRJINT	Projection does not intersect base curve/surface.
CI_ER_PRFAXS	Revolved profile crosses axis.
CI_ER_DUPPRM	Start & End param the same.
CI_ER_TOLACH	Tolerance not achievable.
CI_ER_XFRMRG	Transform not rigid.

Errores topológicos:

CI_ER_TOPO	Unknown topology error.
CI_ER_TWEAK	Cannot perform face tweak.
CI_ER_ELINE	Edge not linear.
CI_ER_EDGECON	Edges not properly connected.
CI_ER_EDCRE	Error attaching/creating edge.
CI_ER_GEOM	Geometry is inconsistent.
CI_ER_INISCT	Inner circuits intersect.
CI_ER_INOUT	Inner loop not inside outer.
CI_ER_STRUCT	Invalid connectivity structure.
CI_ER_REFENT	Invalid reference entity given.
CI_ER_VRTLOOP	Not a vertex loop.
CI_ER_NOGEOM	No geometry for this topology.
CI_ER_UNDMAT	No matrix defined.
CI_ER_NOCSG	There is no related CSG class of solid.
CI_ER_OBUNDF	Object not completely defined.
CI_ER_PRFOPN	Profile open.
CI_ER_PFACE	Profile not contained in face.
CI_ER_POPN	Profile not single open path.
CI_ER_SING	The region is already singular.
CI_ER_VERTCON	Vertex not properly connected.

Errores de Modelos Sólidos:

CI_ER_SOLID	Unknown solid operation error.
CI_ER_OPER	Invalid operand for boolean.
CI_ER_NOEXT	No extents have been set.
CI_ER_VECHIT	No hit by vector.
CI_ER_NOSINT	No intersection found.
CI_ER_NSCHND	No such node found in tree or branch.
CI_ER_ROOT	Not valid for root level CSG entity.
CI_ER_OBEXT	Object outside extents.
CI_ER_NOHALF	Surface cannot produce half space.
CI_ER_PYRSID	Too many sides (pyramid or prism).
CI_ER_CLCGEO	Unable to calculate geometry.

ANEXO 6.

LISTAS PERMANENTES DE OBJETOS MÚLTIPLES (*LPOM*)

Las *LPOM* fueron elaboradas por el prof. Oscar Ruiz en *UIUC* 1995. Transcritas y extendidas por Sebastian Schrader en *CII* 1998 para este proyecto. Estas listas doblemente encadenadas presentan las siguientes características: Perdurabilidad de objetos durante operaciones, Accesibilidad de objetos por nombre, aplicación flexible de rutinas sobre objetos, identificador único para objetos, servicios de base de datos: encontrar, eliminar, construir, copiar, consultar objetos creados, creación de objetos geométricos, asignación de nombre, depuración, registro, recuperación, eliminación, asignación automática de sub-partes, consulta, modificación de propiedades (geométricas, atributos, nombre), carga o descarga desde disco o a disco, uso en creación de otros objetos, permitir creación de nuevos objetos a partir de los ya existentes.

LPOM debe proporcionar un entorno que permita básicamente:

1. La aplicación asincrónica de rutinas procesadoras de información de digitalización: La ejecución de las tareas de procesamiento no debe estar limitada por una secuencia única y predefinida de operaciones. Debe permitir el manejo flexible de estas rutinas.
2. El pre-proceso de una nube de puntos anisotrópicamente muestreada. Debe poderse desarrollar herramientas que permitan llevar una nube de puntos genérica a un arreglo topológicamente regular para el ajuste de una superficie paramétrica.
3. Razonamiento geométrico para implementación de servicios de selección, por ejemplo: "`Select surface region with curvature less than _____`".
4. Acceso por nombre a sub-partes de objetos geométricos.

Ej: *P* : polilinea , a = *p[i]*
i : integer **Qué es a = *P[i]* ?**
Si *cv* = convex hull (*S*) , *i* integer
y *S* = conjunto de puntos **Qué es *f = cv[i]* ?**

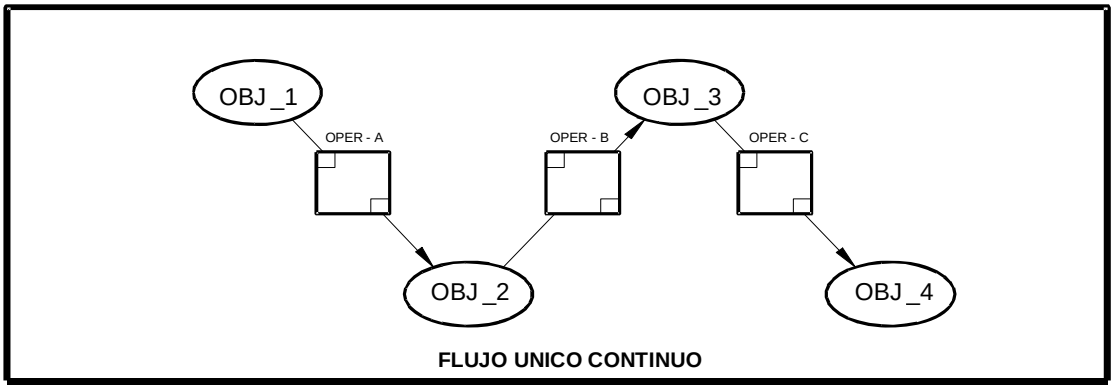
Requerimientos sobre Plataforma

Los requerimientos de la plataforma sobre la cual se construirá DIGI_LAB son:

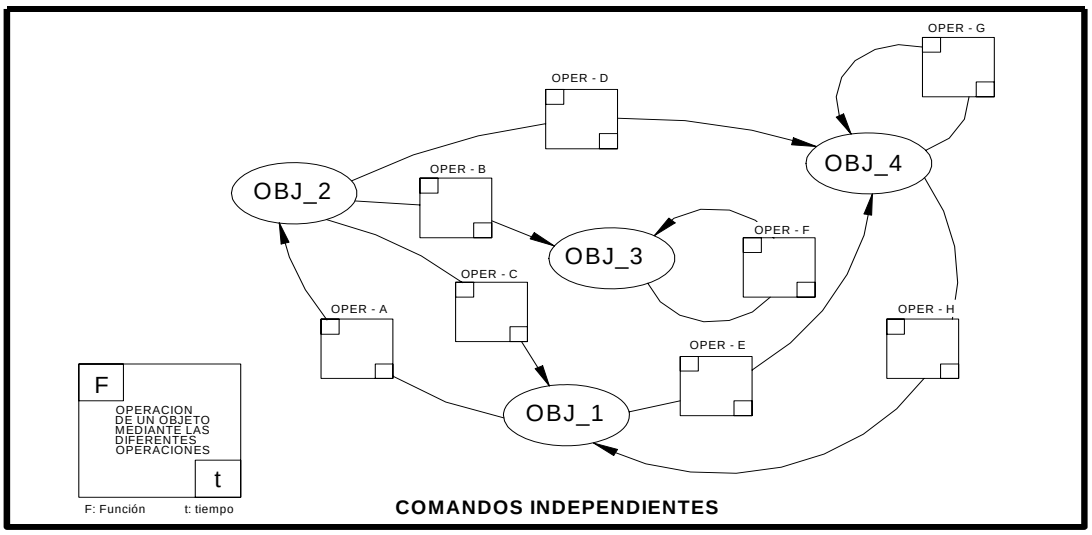
1. Perdurabilidad de objetos durante operaciones. Los objetos creados deben conservarse por fuera del ámbito de las funciones de creación (entidades permanentes)
2. Accesibilidad de objetos por nombre. Cada objeto debe poder ser identificado unívocamente por medio de un identificador (nombre) en cualquier momento y en cualquier ámbito.
3. Aplicación flexible de rutinas sobre objetos. Como varios resultados de los procesos de digitalización no son predecibles –por consideraciones funcionales-, la aplicación de las rutinas debe ser flexible.
4. Necesidad de un identificador único para objetos. Permite su posterior uso para otra rutina.
5. Servicios de base de datos: encontrar, eliminar, construir, copiar, consultar objetos creados en el mundo.
6. Creación de objetos geométricos.
7. Asignación de nombre, depuración, registro Vistazo a propiedades del objeto creado, acceso a su definición y sus atributos.
8. Recuperación (*Retrieve*). Operación que permite recuperar un objeto.
9. Eliminación.
10. Asignación automática de sub-partes. Posibilidad de especificar automáticamente las subpartes de un objeto. Ej: vértices de una polilínea.
11. Consulta.
12. Modificación de propiedades (geométricas, atributos, nombre).
13. Carga o descarga desde disco o a disco.
14. Uso en creación de otros objetos. Permitr creación de nuevos objetos a partir de los ya existentes. Ej: creación de una línea entre dos planos, copia de una entidad, etc.
15. Uso en razonamiento geométrico.

Listas Permanentes de Objetos Múltiples (LPOM)

Una necesidad importante en las aplicaciones para modeladores geométricos es la posibilidad de ejecutar diferentes subtareas con cierta independencia de flujo.
Algunas de las razones principales que explican esta necesidad son: (i) Las aplicaciones de una cierta complejidad pueden tomar un tiempo considerable para ejecutarse. Si en algún proceso intermedio ocurre un error, se requiere iniciar nuevamente la aplicación desde el principio en el caso de un flujo único. (ii) Es frecuente que estados intermedios de la aplicación deban ser estudiados antes de proseguir con la ejecución; también es común que la decisión de qué proceso se debe ejecutar después de otro dependa del juicio humano sobre los resultados del proceso anterior.



Gráfica 1. Ejecución de flujo único

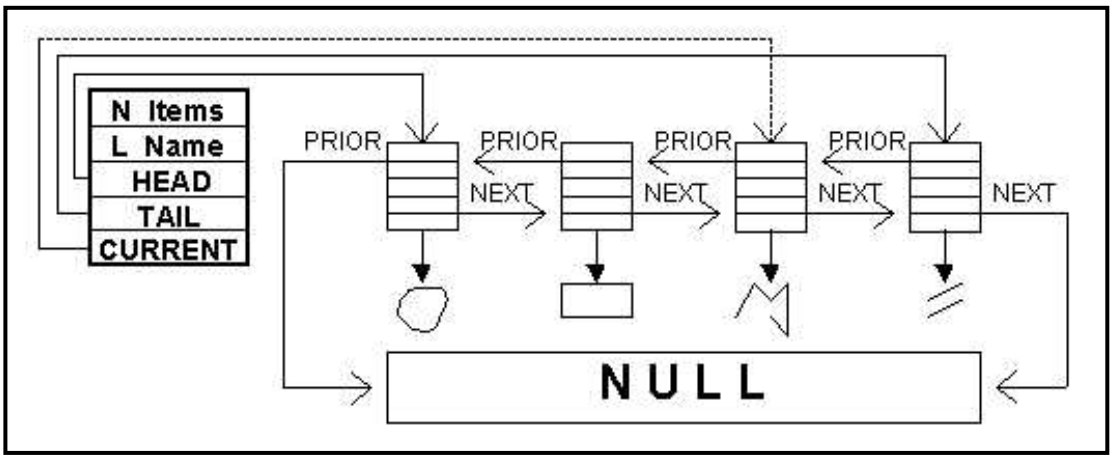


Gráfica 2. Ejecución por comandos independientes

En las gráficas 1 y 2 se ilustra la diferencia entre los modos de ejecución de flujo único (sin división por procesos) y "por comandos" (con independencia de los procesos de la aplicación).

Características Principales de las Listas Permanentes de Objetos Múltiples:

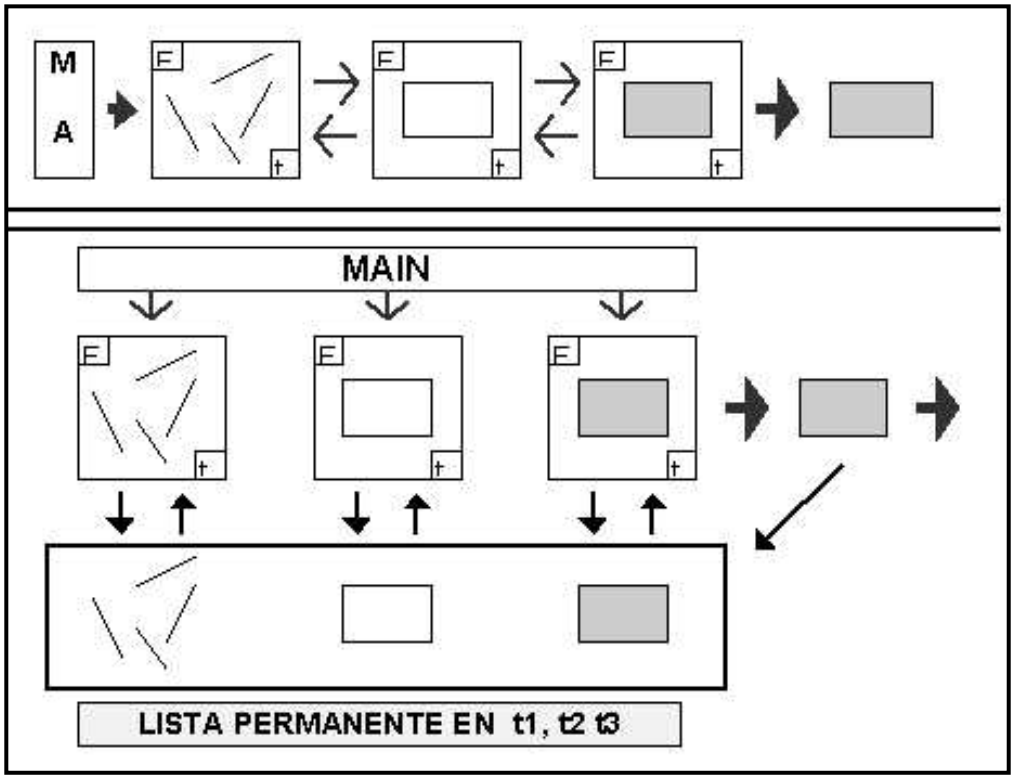
i. Manejo unificado de objetos de cualquier tipo. Operaciones deben ser indiferentes al objeto.
ii. El tipo del objeto debe ser identificable por información externa a él. Esto se debe a que el objeto es representado por un apuntador genérico (void*) que no da ninguna información sobre el objeto apuntado.
iii. Tipos básicos: int, str, real, char y objetos AIS (handles y listas AIS).
iv. Tipos (clases) construidas (listas NO_AIS).
v. Objetos matemáticos (vectores, matrices, transformaciones).
vi. Su permanencia, es decir, su persistencia incluso por fuera del ámbito de los procesos particulares. Tanto la *estructura de la lista* en sí como los *objetos múltiples* que colecciona permanecen después de que se termina algún proceso o función.



Gráfica 3. Lista de objetos múltiples (homogénea)

Listas Permanentes de Objetos Múltiples para Implementar Aplicaciones de Ejecución "Por Comandos" o Subtareas.

Con base en las **LPOM** se puede proporcionar un entorno permanente para almacenar los resultados de una subtarea, de modo que se permite al usuario recurrir a estos resultados desde otra subtarea que puede pertenecer a un ámbito diferente. La ejecución por comandos de una aplicación con ayuda de las **LPOM** se muestra en la gráfica 4.

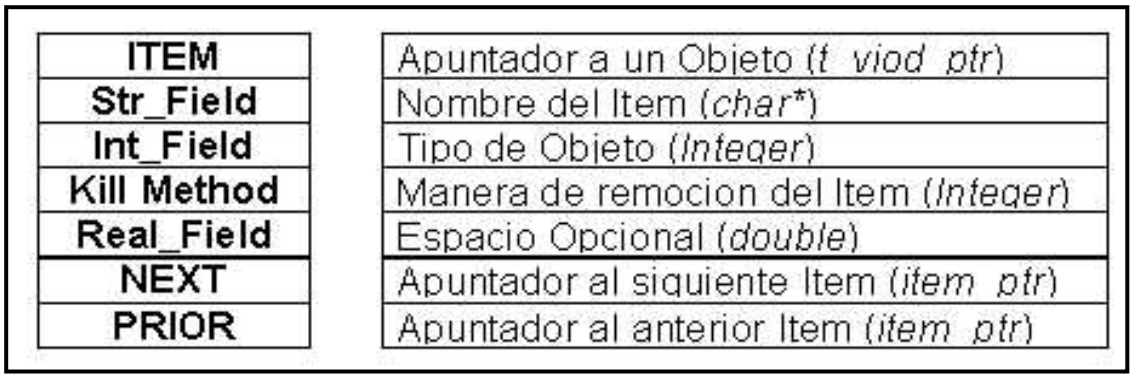


Gráfica 4. Ilustrando una aplicación con ejecución por flujo continuo vs. una aplicación con ejecución por comandos independientes ayudándose de las listas permanentes

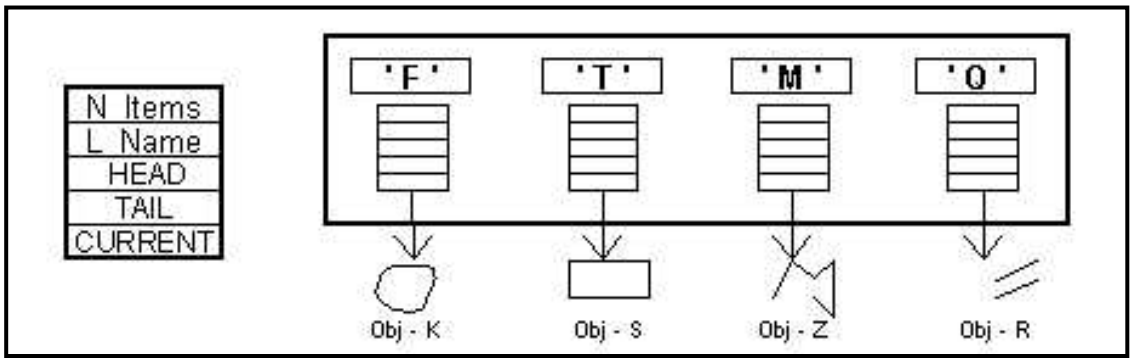
En la ejecución por comandos independientes se facilita la posibilidad de independizar las subtareas en ámbitos diferentes, pues los parámetros relevantes pueden almacenarse en la **LPOM** y son accesibles por lo tanto desde cualquier ámbito.

Estructura de las Listas Permanentes

Como se explicó antes, las **LPOM** pueden almacenar objetos múltiples, pero sólo en última instancia. En primera instancia son homogéneas; es decir, su dominio es únicamente de tipo **ITEM**. Es a través de los **ITEM** que se relacionan los objetos de múltiples tipos con la lista. Esta propiedad permite adaptar posteriormente la lista para contener otros tipos de objetos simplemente definiendo la estructura del ITEM correspondiente.



Gráfica 5. Ilustrando la estructura de un ITEM.



Gráfica 6. Ilustrando que los objetos no se almacenan directamente en la lista, sino a través del ITEM, que actúa como direccionador.

En la lista se pueden insertar los siguientes tipos de objetos:

OBJECT_STRING	<String>	OBJECT_LIST_WRAP:	
OBJECT_CHAR:	<Letter>	OBJECT_LIST:	
OBJECT_REAL	<Real>	CI_TYCP:	<Cartesian_Point>
OBJECT_INT	<Integer>	CI_TYCU:	<Curve>
OBJECT_VEC	<Vector_3>	CI_TYCI:	<Circle>

CI_TYEL: <Ellipse>
CI_TYPN: <Polyline>
CI_TYBC: <B_Spline (curve)>
CI_TYSU: <Surface>