DESARROLLO E INTEGRACIÓN DE HERRAMIENTAS DE VISUALIZACIÓN DE OBJETOS 3-D REALES MODELADOS POR RECUBRIMIENTOS SUPERFICIALES APROXIMADOS

Autor: Francisco Morán Burgos
Tutor: Narciso García Santos

Grupo de Tratamiento de Imágenes
Depto. Señales, Sistemas y Radiocomunicaciones
E.T.S. Ingenieros de Telecomunicación
Universidad Politécnica de Madrid

Junio de 1992

TÍTULO: Desarrollo e integración de herramientas de visualización de objetos 3-D reales modelados por recubrimientos superficiales aproximados.

AUTOR: Francisco Morán Burgos.

TUTOR: Narciso García Santos.

DEPTO.: Señales, Sistemas y Radiocomunicaciones - UPM.

TRIBUNAL:

Presidente:

Vocal:

Vocal Secretario:

FECHA DE LECTURA:

CALIFICACIÓN:

RESUMEN DEL PROYECTO:

Muestreando la superficie de un objeto real mediante técnicas láser se obtiene un conjunto de puntos independientes excesivamente denso para ser posteriormente procesado con comodidad. Eligiendo algunos de ellos, y reagrupándolos en subconjuntos para definir facetas planas o parches alabeados, es posible recubrir de manera aproximada la superficie original con un volumen de datos notablemente inferior. Este modelo aproximado puede ser luego visualizado con cierto foto-realismo gracias a técnicas que no sólo lo proyectan en una imagen bidimensional simulando correctamente los parámetros de cámara (posición respecto al objeto, distancia focal del objetivo, etc.), sino que permiten también controlar la iluminación de la escena sintética. Se presentan las mejoras obtenidas en un programa de *rendering* concebido originalmente como herramienta de ayuda al diseño por ordenador, y los resultados teóricos y prácticos sobre el problema de la facetización adaptativa de superficies recubiertas por parches de Gregory.

PALABRAS CLAVE:

Modelado de escenas 3-D, recubrimientos superficiales, facetas planas vs. parches alabeados, subdivisión adaptativa; visualización (*rendering*), *z-buffer*, pulido del color; calidad subjetiva, foto-realismo.

A mi tío, por si no tengo ocasión de dedicarle una película.

Preámbulo. MARCO DEL PROYECTO

El Proyecto de Fin de Carrera que aquí se presenta fue realizado durante el segundo semestre de 1990 por Francisco Morán Burgos en el Laboratorio del Depto. IMA*(ges)* de la *Ecole Nationale Supérieure des Télécommunications* (ENST) de París, y supervisado por Francis Schmitt, responsable del Grupo *Image* de este Departamento. Los resultados obtenidos fueron presentados en la ENST de París ante los miembros de IMA en diciembre de 1990, y quedaron recogidos en una breve memoria redactada en francés.

Posteriormente se confeccionó el presente informe definitivo en castellano durante el curso 1991-92, en el Grupo de Tratamiento de Imágenes de la Universidad Politécnica de Madrid. Este informe fue expuesto en la ETSIT de Madrid en junio de 1992, ante un Tribunal presidido por Narciso García Santos, responsable del GTI y tutor de este Proyecto.

Agradecimientos

He tenido la suerte de realizar mi Proyecto de Fin de Carrera sobre un tema apasionante, que ahora conozco algo mejor: la imagen sintética. Además, esto ha sido posible gracias a haber formado parte de la primera iniciativa oficial de intercambio de estudiantes entre la ETSIT de Madrid y la ENST de París, razón de por sí suficiente para considerarme afortunado. Por ello quiero comenzar agradeciendo su ayuda a las personas que en una y otra Escuela nos ayudaron a rodear los obstáculos administrativos; esperando que haya cada vez menos en el futuro, es obligado felicitar a todas ellas por el interés que nos mostraron, y muy especialmente a las que en París nos acogieron tan calurosamente.

Pasando de lo general a lo particular —en la más pura tradición francesa—, agradezco a Francis Schmitt su apoyo y su buen humor, garantizados hasta horas a las que suelen escasear, a lo largo de los casi seis meses que estuve en *Télécom*. Deseo también recordar a Alain Clainchard, por todo lo que llega a deber un novato a su *system manager*, y a todos los doctorandos y proyectandos de nombres impronunciables que confieren al *Labo Image* su "color" característico. Y mencionar especialmente, de entre los primeros, a Isabelle Bloch-Boulanger y a Xin Chen, que aportaron a mi trabajo la crítica irreemplazable del usuario, y a los que debo varias fotografías y muchas horas.

Agradezco a Narciso García Santos el riesgo que tomó al aceptarme como proyectando "remoto" del GTI, cuando me fui a París, y sobre todo su paciencia durante la redacción de esta versión en castellano de mi Proyecto, una vez yo en Madrid. Y a todos los miembros del GTI el haberme ayudado a pesar del poco tiempo que he compartido con ellos.

Con especial placer, agradezco a los pocos *telecos* que recordaré, los años que hemos pasado juntos en esta Escuela, y fuera de ella: a José María Martínez Sánchez (pronúnciese *Shema*), a Juan José Martínez Madrid, y a Luisa Molina Fernández (que en realidad no es *teleca*), el haberse paseado conmigo por "el país de las Maravillas"; a Gregorio Mayoral (*Goyo*), el enseñarnos a pronunciar *Shema*, y el suspender más veces que yo la Química; a Roberto Sancho Villa, el haber compartido conmigo muchas horas de laboratorios y de residencia en la *Maisel*; y a Esteban Pérez Castrejón, el completar el peculiar cuarteto de madrileños en París, y el ayudarme a enseñar algo de mus a unos chiquitos de Barcelona (Jesús y Gerardo: eso va por vosotros).

Para terminar con lo más importante, quiero que conste por escrito —ya que se me da tan mal decir estas cosas— que sin mis padres, mis tíos y el resto de mi familia, y la educación que entre todos me han dado, ni habría aquí una sección de Agradecimientos, ni yo hubiese podido realizar este Proyecto, ni sería quien soy. Probablemente esto parezca obvio, pero estoy seguro de que a mí nunca me lo parecerá demasiado.

Francisco Morán Burgos.

Contenido

Lista de Figuras e Imágenes

Capítulo 1. INTRODUCCIÓN

1.1. La imagen sintética

De entre las imágenes que se pueden crear mediante ordenador, tienen especial interés aquellas que intentan representar objetos tridimensionales reales, por el reto que supone el hacerlo con un cierto **foto-realismo**. Se trata de conseguir un resultado que pueda pasar por una fotografía de una escena verdadera, es decir, de lograr una síntesis convincente del mundo 3-D, y de proyectarla sobre un espacio 2-D acotado, con propiedades ópticas correctas para que el ojo humano pueda interpretarla. La gran diferencia entre una imagen de este tipo y un gráfico de barras, por ejemplo, reside en que a nadie se le ocurre pedir al gráfico de barras que se parezca a nada, porque se sobreentiende que refleja un concepto abstracto; sin embargo, a la imagen llamada "de síntesis" se le exige siempre que represente con la mayor verosimilitud posible la escena que sintetiza, y más aún cuando esa escena está compuesta por objetos de la vida real.

El proceso de representación gráfica de objetos tridimensionales por ordenador se compone de dos fases claramente separadas: el **modelado** del objeto, y la **visualización** del modelo creado previamente. A lo largo de la corta historia de los gráficos sintéticos, las técnicas aplicadas a una y otra fase del proceso de creación han evolucionado rápidamente en pos de ese ideal de foto-realismo, gracias en gran parte al desarrollo de la tecnología de circuitos integrados especializados, pero también a las cuantiosas investigaciones teóricas realizadas en este campo. Esta fulgurante evolución fue motivada inicialmente por las necesidades industriales de las ya "clásicas" aplicaciones de tipo CAD (*Computer-Aided Design*: diseño asistido por ordenador), y más recientemente, por un número creciente de nuevos usos comerciales de la infografía: carátulas de programas y *spots* publicitarios en televisión, vídeo-juegos, efectos especiales de animación en cine...

1.1.1. Modelado de objetos 3-D

Esta es la fase de **síntesis de la escena**. A diferencia de lo que ocurre en el dibujo técnico tradicional, en el que el dibujante parte de un modelo (como poco, mental) del objeto que desea representar, cuando se trabaja con un ordenador se parte siempre de la nada más absoluta —el ordenador es una criatura sin duda voluntariosa, pero exasperantemente necia, y desprovista por completo de imaginación creativa: así pues, hay que describirle todo con sumo detalle.

La primera aproximación de un objeto 3-D suele ser una *malla de alambres* (*wire-frame*): el modelo consiste en un conjunto de poliedros de los que sólo intesesan las *aris-*

tas. No hay ningún tipo de *información superficial*, por lo que la fase de visualización se reduce a trazar segmentos rectilíneos en el plano de la imagen, como simple proyección de las aristas tridimensionales. Este modelado permite extraer, rápida y sencillamente, una primera idea visual acerca de la corrección de lo creado, y no hay que olvidar que era el único existente en los tiempos no tan lejanos de los monitores monocromos —*wire-frame* se suele asociar aún con fósforo verde.

Sin embargo, en cuanto se dispone de un monitor en color, es posible añadir y tratar esa información superficial. Los poliedros que forman el objeto pasan de ser conjuntos de aristas a serlo de *facetas planas*, a las que es posible asignar distintos colores. Con esto se permite al algoritmo de visualización introducir en la imagen una información capital para el ojo humano: la que proporcionan las *partes vistas y ocultas*. Esta mejora aumenta implícitamente el tamaño y la complejidad del modelo, y sobre todo, como se verá más adelante, introduce nuevos problemas en la fase posterior.

Aun siendo muy utilizadas por su sencillez matemática, las superficies facetizadas son, demasiadas veces, sólo tristes aproximaciones de las que delimitan verdaderamente los objetos reales. Por ello, a partir de mediados de los años '70 se empezó a desarrollar el modelado con *parches alabeados*, que tienen sobre las facetas planas la ventaja de permitir generar superficies *suaves*[1]. Además, aproximan más fielmente la realidad, y producen modelos notablemente más compactos: típicamente, es fácil poder describir un objeto real con menos de la veinteava parte de parches que de facetas, obteniendo una aproximación equivalente. Y esto, siendo el volumen de información relativa a un parche sólo cuatro o cinco veces superior al correspondiente a una faceta. Por ejemplo, la figura 1.1 muestra un "cuadrado" bicúbico de Bézier, uno de los tipos de parches más estudiados y empleados junto con los *B-splines* y los triángulos de Gregory (cf. [FAUX-79], [BÉZIER-86]): está definido por dieciséis *puntos de control*, no coplanarios en general, de los que sólo los cuatro de las esquinas pertenecen a la superficie, mientras que los otros doce tiran del parche hacia sí, moldeándolo según la rejilla tridimensional que forman, lo que confiere una gran libertad al creador del modelo, que no tendría con cuatro cuadriláteros planos (también dieciséis puntos 3-D si cada faceta se describe independientemente del resto).

La explicación de esta aparente paradoja está en que hay información predefinida implícita: tanto los parches espaciales como las facetas planas pertenecen a familias de superficies *primitivas* conocidas de antemano. En ambos casos se restringe el número de grados de libertad del resultado, que se rige por una ecuación determinada, y con el que no se puede modelar sin error una superficie arbitraria, pero en el caso de los parches esta información oculta es mucho más potente por ser tridimensional. Al introducir un *a priori* se logra, a fin de cuentas, una *transformación de complejidad* (de cuantitativa a cualitativa), que es beneficiosa porque la parte del sistema que se complica es la predefinida, la que no varía de un modelo a otro.

---

[1] Con este término vago pero intuitivo se pretende traducir *smooth*, evitando el uso inadecuado de otros cargados de significado matemático más preciso y riguroso, como "continuas" o "derivables".

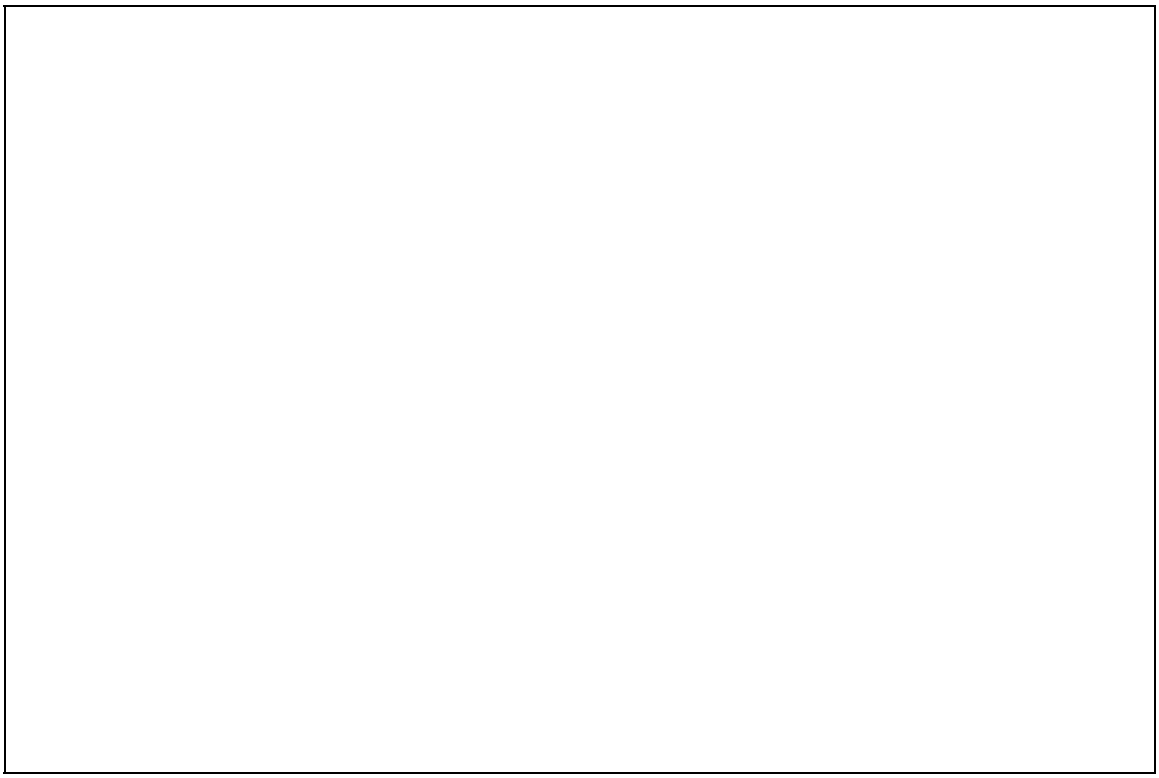

figura 1.1 - Ejemplo de parche alabeado: cuadrado bicúbico de Bézier.

El precio a pagar por este ahorro en volumen de información es alto, sin embargo. Por una parte surgen problemas en la fase de modelado: así como el tratamiento matemático posterior de las facetas planas es muy sencillo, por ser todas las ecuaciones lineales, el de las superficies de grado superior se complica. Por ejemplo, interconectar varios parches garantizando un resultado globalmente suave, no es tarea trivial; y modificar parcialmente el modelo una vez acabado, tampoco. Y por otra parte, tal vez más importante, sólo los algoritmos de visualización más sofisticados saben tratar directamente objetos descritos por superficies curvas. En definitiva, como todo artículo informático "de lujo", compensa siempre que se disponga de métodos de cálculo suficientemente potentes.

Como etapa final, dentro de la fase de modelado cabe incluir la especificación del *punto de vista*. Llega finalmente la hora en que el creador se calza la boina de director para situar la "cámara" respecto a la escena (posición y punto de mira), y precisa sus *parámetros ópticos*, que serán más o menos completos según el grado de foto-realismo que se espere de la imagen. Uno de los más usuales es la distancia focal del objetivo utilizado, porque al ser variada, permite acercar o alejar aparentemente la escena sin mover la cámara, logrando lo que se conoce como efecto de *zoom*, o crear interesantes distorsiones ópticas como las producidas por un objetivo de "ojo de pez".

Eventualmente, también tendrá que ser fijada la *iluminación*, que puede ser blanca o coloreada; difusa (luz ambiente) o compuesta por una o más fuentes puntuales (focos). Con un nivel suficiente de sofisticación en el tratamiento de la luz, es posible atribuir a los objetos valores en una escala de "calor" (clasificación según índice de refracción o "brillo metá-

lico"). Lo que implica, en el caso de que haya focos, dotar al programa de visualización de un mecanismo adecuado para el tratamiento de los reflejos especulares sobre las superficies brillantes.

1.1.2. Visualización (*Rendering*)

Esta es la fase de **producción de la imagen**. Se trata de tomar como entrada el modelo tridimensional creado en la fase anterior para calcular una matriz de píxeles (castellanización de la contracción inglesa "*pixels*": *picture elements*). La resolución de la imagen, que es doblemente discreta, depende tanto de sus dimensiones como de su riqueza cromática, que viene dada por el rango de valores posibles del color, y se suele expresar en "número de píxeles (horizontal x vertical) x número de bits por píxel (bpp) para el color". Cabe señalar también que el conjunto de colores que pueden estar simultáneamente presentes en una imagen, llamado *paleta*, es generalmente un subconjunto del espectro total disponible.

Actualmente, incluso los ordenadores personales más económicos vienen ya equipados mayoritariamente con tarjetas gráficas y monitores que permiten al menos trabajar con imágenes de 640 x 480 píxeles y 8 bpp; y en las *workstations* (estaciones de trabajo) se puede disponer de 1280 x 1024 x 24 bits a precios razonables, aunque siguen siendo más frecuentes —léase "sensiblemente más baratas"— las que ofrecen sólo 8 bpp para esa misma resolución. Cuando se codifica el color en un solo octeto (8 bits), se logra una paleta de 2^8=256 colores, y es necesario almacenar en una tabla la intensidad de cada uno de los canales o colores básicos (RGB, de *Red, Green, Blue*: rojo, verde, azul) correspondiente a cada color compuesto de la paleta. A esto se llama usualmente trabajar con *pseudo-color*, frente al *color real* que proporcionan 24 bpp, i.e. un octeto por canal. De hecho, estos nombres están plenamente justificados dado que en el primer caso el espectro percibido por el ojo humano es claramente discreto, mientras que en el segundo es virtualmente continuo: casi 17 millones de colores simultáneos posibles (2^{24} exactamente).

El primer obstáculo real con el que se enfrentaron los diseñadores de algoritmos de visualización fue el de las superficies ocultas. Cuando el modelo del objeto es simplemente una lista de segmentos 3-D, no hay más que proyectarlos en 2-D de acuerdo con el punto de vista y los parámetros de cámara especificados. El único problema en este caso es el del *clipping* (recorte de la escena causado por el encuadre): hay que trazar sólo las partes de los segmentos 2-D que se hallen dentro del marco de la imagen. La figura 1.2 ilustra esta proyección, y las convenciones adoptadas usualmente respecto a la orientación de los ejes cartesianos y al ángulo de visión.

Sin embargo, cuando se trata de dibujar una lista de polígonos, introduciendo la información superficial —que no vana— antes mencionada, hay que calcular primero sus eventuales intersecciones y ocultaciones. Otro aspecto delicado es el del tratamiento correcto de la luz: cuando se trabaja con objetos facetizados, el resultado de colorear uniformemente el interior de cada faceta es una discontinuidad molesta (por inverosímil) en el color, como la

que muestra la imagen 1.1. Durante los años '70, investigadores como Catmull, Gouraud y Phong produjeron, en la Universidad de Utah, importantes resultados en ambos campos.

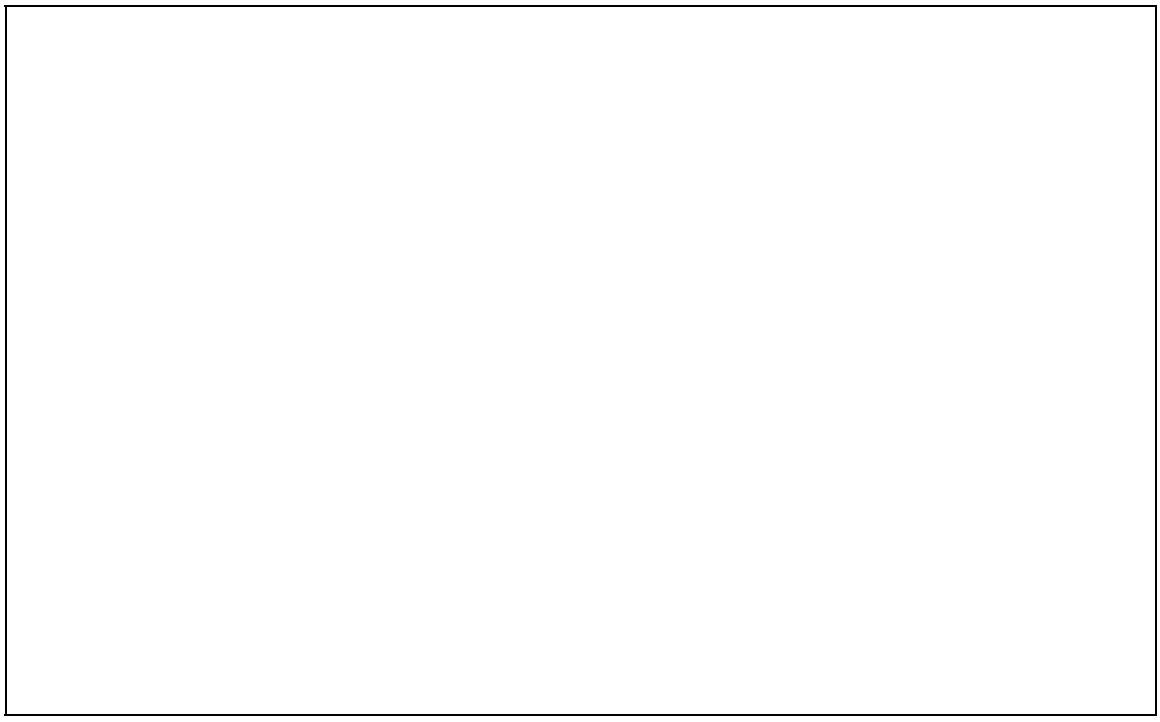

figura 1.2 - Proyección de la escena 3-D sobre el plano de la imagen.

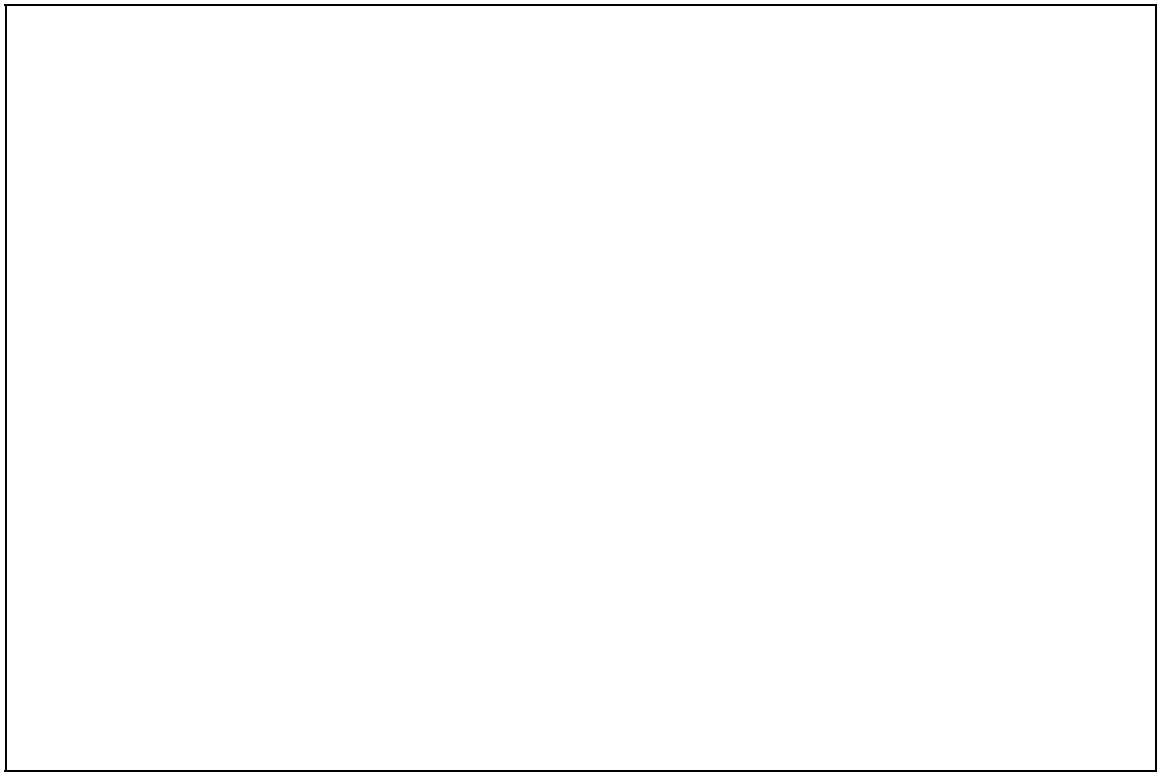

imagen 1.1 - Coloreado uniforme del interior de cada faceta.

Ed Catmull, en 1974, publicó un algoritmo para resolver el problema de las superficies ocultas que se ha convertido en un clásico por su sencillez y generalidad: el *z-buffer*. Este algoritmo rompió con el método inicial de ordenar la lista de facetas según su profundidad antes de dibujarlas, para garantizar que las más alejadas del espectador no tapen a las más cercanas. Cuando no se dispone de *hardware* especializado para esta ordenación, es el programa de aplicación quien debe encargarse de realizarla, y esto puede ser muy costoso en tiempo de cálculo cuando la lista de polígonos es grande.

La idea de Catmull consiste en almacenar el valor de la *z* correspondiente a cada píxel en un *buffer* (zona matricial de memoria) conocido por el inevitable nombre de *z-buffer*. Así, cuando un punto 3-D es proyectado sobre un píxel que ya estaba ocupado, se compara la *z* del punto actual con la almacenada en memoria: si esta última es menor, es que el punto actual está detrás del ya existente (obsérvese la orientación del eje *z* en la figura 1.2), con lo que no es necesario alterar el correspondiente píxel (simplemente, no se hace nada). Si, por el contrario, la *z* actual es menor que la de la memoria, es que el punto actual está delante del anterior, y por lo tanto lo oculta: en este segundo caso, hay que reemplazar en la imagen el color del píxel antiguo por el del nuevo punto, y actualizar en el *z-buffer* el correspondiente valor de *z*.

De esta manera, las facetas no necesitan estar ordenadas, sino que se dibujan todas según la secuencia arbitraria en que sean entregadas al algoritmo de visualización. También se resuelve con elegancia el problema de las intersecciones de distintas facetas. Y además, es posible crear imágenes "paso a paso": en el caso de que se quieran añadir objetos para completar una escena, no hay más que guardar el estado final del *z-buffer*, y aplicar el mecanismo descrito a los puntos de los nuevos polígonos, que se "barajarán" con toda naturalidad con los ya existentes.

En los años '80, el *z-buffer* tradicional fue mejorado por el *a-buffer*, también llamado a veces *alpha-buffer*, que permite tratar objetos transparentes y corregir defectos de *aliasing* (fenómenos de borde debidos a escasez de resolución gráfica, conocidos como "dientes de sierra"). Como se explica en [CARPENTER-84], gracias a este algoritmo se logra un notable incremento de la resolución efectiva de la imagen con un pequeño aumento del coste computacional y de la memoria necesaria.

De la misma manera que en el *z-buffering* se asocia a cada píxel de la imagen una medida de profundidad, en el *a-buffering* se le asigna, además, un valor de opacidad *a*, normalmente codificado en un octeto (rango: 0..255). Suponiendo que varios objetos de la escena compitan, al ser proyectados, por un píxel con $a = 255$ (totalmente opaco), el color del píxel estará únicamente determinado por el del objeto más cercano, como en el caso del *z-buffer* normal; pero si $a = 204$ (transparencia del 51/255 = 20%), el píxel obtendrá su color en un 80% del objeto más cercano, y en un 20% de los objetos tapados por él. Esta mezcla (*blending*) de colores permite también que, en el caso de que un píxel sólo esté parcialmente tapado por un objeto, el fondo contribuya al color final, consiguiéndose así un filtrado corrector de los dientes de sierra en las siluetas.

Sin embargo, con este método tampoco se consigue simular efectos especiales como, por ejemplo, la refracción de la luz a través de una lupa; ni cosas aparentemente sencillas,

como las sombras proyectadas por unos objetos sobre otros o sobre sí mismos. Para ello es necesario aplicar algoritmos notablemente más sofisticados como el *ray-tracing*, creado en 1980 por Turner Whitted, que simula con precisión la trayectoria que siguen los rayos luminosos hasta llegar a la cámara, teniendo en cuenta las propiedades ópticas de los objetos atravesados. El *ray-tracing* es sin duda el mecanismo de visualización más lujoso, pero también, y con mucho, el más caro en tiempo de cálculo.

En cuanto al tratamiento del color, Henri Gouraud desarrolló en 1971 un algoritmo para suavizar las bruscas transiciones de color entre polígonos adyacentes de un mismo objeto. Consiste en interpolar bilinealmente el color en el interior de cada polígono a partir del calculado para sus vértices para hacer más continuo el sombreado debido a la iluminación. Cuatro años más tarde, Bui-Tong Phong publicó los resultados de sus mejoras sobre el trabajo de Gouraud. Logró un mecanismo más convincente para la interpolación del color dentro de cada polígono, y un modelado más exacto de las leyes físicas de la luz, sobre todo en lo que toca a la reflexión especular en objetos brillantes, que les confiere un toque plástico característico (cf. imagen 1.2). Es de notar que aún ahora, a principios de los '90, la mayor parte del *hardware* especializado en gráficos no implementa algoritmos de "pulido" del color más sofisticados que los arriba mencionados de Gouraud y Phong.

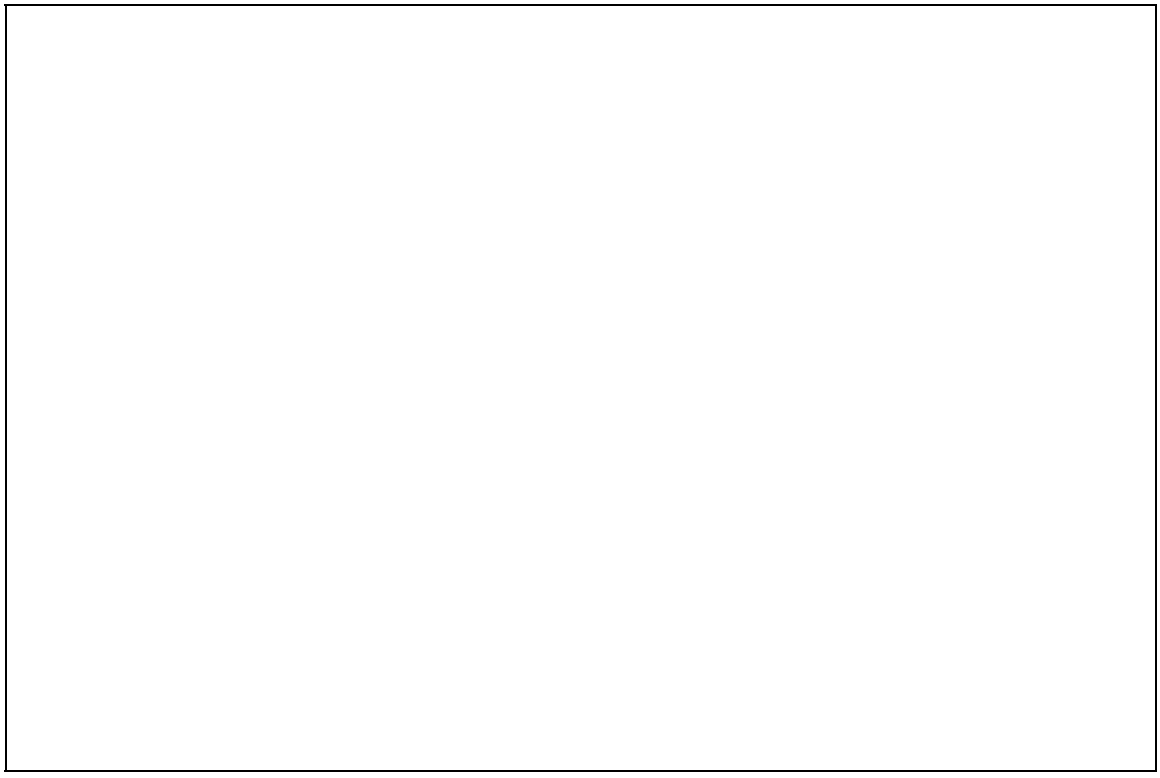

imagen 1.2 - Interpolación del color según el método de Phong.

Otra técnica interesante ideada por Catmull a mediados de los '70, y muy desarrollada desde entonces, es la del *texture-mapping*: en lugar de colorear los objetos uniformemente, se trata de simular que estuvieran recubiertos por una tela estampada, representada por un

mapa de bits (una imagen bidimensional convencional). Así, es posible modelar con algún realismo, a la vez que económicamente, cosas que aun siendo estructuralmente sencillas, pueden ser cromáticamente complejas (una naranja, un mosaico, la Tierra, una piel de leopardo...).

Los años '80 han sido probablemente los que más frutos han dado hasta ahora en esta búsqueda del foto-realismo: a medida que se ha ido disponiendo de la tecnología necesaria para abordarlos, se han tratado complejos efectos de cámara, como el enfoque y la profundidad de campo (no sólo se fija la distancia focal del objetivo, sino también la apertura del diafragma); o las imágenes movidas, muy útiles en secuencias animadas para dar impresión de movimiento (en este caso, se selecciona además la velocidad de obturación).

También, en un esfuerzo de agrupar y unificar todas las técnicas descritas, y otras más, la compañía californiana P·I·X·A·R publicó en 1989 el interfaz "RenderMan", que describe un formato común como base de diálogo entre los algoritmos de modelado y los de visualización. Y que define, al mismo tiempo, el objetivo de diseño para el *hardware* gráfico de la próxima generación, al representar el estado del arte de la imagen sintética a principios de la década de los '90, que parece llamada a ser la de los primeros pasos de la realidad virtual.

1.2. Descripción del Proyecto

1.2.1. Objetivos

El Grupo *Image* trabaja desde el principio de la década de los '80 en el desarrollo de un **tratamiento de objetos tridimensionales** comparable al tratamiento de imagen tradicional. Se trata de conseguir un conjunto de herramientas teóricas, metodológicas, y materiales, que permitan manejar la información correspondiente a objetos reales con comodidad y eficacia. Entre otros temas relacionados con el dominio 3-D (reconocimiento de formas, *shape from shading*, simulaciones de fenómenos físicos...), el Laboratorio del Grupo *Image* estudia con especial interés el muestreo de superficies espaciales, con el fin de establecer un puente entre la realidad y la imagen sintética. En particular, su sistema *3D-Vidéolaser* realiza, mediante técnicas láser, un muestreo denso y topológicamente regular de la superficie de un objeto cualquiera, produciendo bases de datos muy voluminosas que contienen las coordenadas de cientos de miles de puntos, y a partir de las cuales se puede aproximar la superficie original por un modelo manejable por ordenador.

El Laboratorio del Grupo *Image* contemplaba, cuando se realizó este proyecto en 1990, tanto los modelos de recubrimiento de superficies clásicos en síntesis de imagen, como otros más prometedores, aún en fase de desarrollo teórico y experimental. Así, se trabajaba con recubrimientos de la superficie considerada por medio de facetas planas poligonales, y por parches alabeados. De entre los parches alabeados, había ya abundante literatura refe-

rente a los cuadrados polinomiales bicúbicos de Bézier, adoptados por algunas aplicaciones comerciales de CAD. Los triángulos cuárticos de Gregory eran más desconocidos, pero presentan sobre los primeros ventajas interesantes como son la ausencia de ciertas restricciones geométricas de la superficie a modelar, y una mayor adaptabilidad a ésta en casos "patológicos" (agujeros, esquinas, picos...).

Para avanzar en la investigación de este último tipo de parches, era imperativo disponer de métodos gráficos de verificación del modelado. Y para poder visualizar los resultados producidos por los diferentes tipos de modelos, era necesario desarrollar una herramienta versátil, capaz de manejar cualquiera de esos formatos de superficies, ofreciendo al mismo tiempo al usuario una total libertad a la hora de elegir los parámetros de cámara. Además, se deseaba obtener un alto nivel de calidad en el tratamiento del color y de la iluminación. Finalmente, esta herramienta debería ser capaz de trabajar con superficies arbitrariamente complejas, compuestas por un número de facetas variable en el rango de decenas a cientos de miles.

1.2.2. Antecedentes

Se disponía de un programa llamado "Ripolin", desarrollado originalmente para correr sobre IBM-PC / MS-DOS en la ENSAD (Escuela Nacional Superior de Artes Decorativas), y posteriormente modificado en Télécom [NOSMAS-89], que permitía entonces representar objetos descritos por un conjunto de facetas planas poligonales. Se basaba en un algoritmo de *z-buffer* en *modo línea* (la imagen se produce línea por línea, y no globalmente) que realizaba el tratamiento de superficies ocultas, y opcionalmente, la interpolación del color según los métodos de Gouraud y Phong. Sin embargo, admitía un número pequeño de facetas, y trabajaba sólo con imágenes de baja resolución (512 x 512 píxeles con 8 bpp), aproximando con cálculos muy laboriosos los valores del color.

También existía un programa traductor de modelos superficiales, desarrollado en Télécom [BURGSTAHLER-89], para pasar de parches alabeados a facetas planas, único formato de entrada admitido por el algoritmo de *z-buffer*. Desgraciadamente, sólo estaba contemplado el caso de los parches cuadrados bicúbicos de Bézier, y no el de los triángulos de Gregory.

1.2.3. Lenguajes y material utilizados

El programa Ripolin estaba escrito en FORTRAN-77 por motivos históricos —las primeras aplicaciones de CAD desarrolladas en la ENSAD datan de la época en que los ordenadores no hablaban nada más razonable—, por lo que las modificaciones y ampliaciones correspondientes se escribieron en ese lenguaje. El traductor de modelos de superficies, "Métamorphose", estaba escrito en PASCAL. Como se consideró que la subdivisión de los

triángulos de Gregory presentaba suficientes analogías con la de los parches de Bézier, se conservó alguna parte del programa y, por ello, su lenguaje.

Los equipos sobre los que se realizó el presente Proyecto fueron principalmente un DEC-VAX 8550 del Departamento CAL (centro de cálculo), conectado a dos DEC-µVAX de IMA, y a dos terminales gráficos de 1280 x 1024 x 24 bits (monitores SONY, tarjeta GOULD); y varios ordenadores personales compatibles con IBM-PC, dotados de tarjetas gráficas con salida en 512 x 512 píxeles y 256 colores sobre monitores SONY.

En cualquier caso, dada la alta portabilidad de los lenguajes utilizados, la implantación del software desarrollado sobre cualquier otro sistema que no sea VAX/VMS no debería plantear ningún problema mayor.

1.2.4. Contenido de esta Memoria

A este capítulo de **INTRODUCCIÓN** siguen los dos principales, en los que se detalla el trabajo de programación realizado. El primero de ellos explica el funcionamiento general de **RIPOLIN**, así como las mutaciones y mejoras que sufrió; el segundo analiza los problemas involucrados en la subdivisión adaptativa de los triángulos de Gregory que lleva a cabo **TRIANGLE**.

Estos dos capítulos tienen una estructura similar: un primer apartado describe el punto de partida, en lo referente a los resultados, tanto teóricos como algorítmicos, con los que contaba el Labo. IMA al inicio de este Proyecto; después, un segundo apartado presenta brevemente los resultados comparativos obtenidos, y un tercero, el funcionamiento del programa, desde la óptica del usuario (aspectos externos: diálogos, entradas y salidas), y desde la de la máquina (aspectos internos: descripción cualitativa de estructuras de datos, procedimientos y funciones). Además, cada uno de estos dos capítulos tiene dos anejos: uno referente a la teoría matemática involucrada en cada caso, y otro que lista exhaustivamente los módulos escritos, describiendo sus funcionalidades individuales y las interrelaciones del conjunto.

Luego se presentan, en un capítulo aparte, los **RESULTADOS GRÁFICOS** obtenidos en uno y otro caso, y se discute su grado de adecuación a los objetivos iniciales para evaluar el trabajo realizado. También se sugieren posibles mejoras para futuras versiones de los programas.

Finalmente, cierran la Memoria los **anejos** mencionados, que figuran en el siguiente orden: **A. RIPOLIN: PROYECCIÓN Y COLOR**, **B. RIPOLIN: DESCRIPCIÓN DE FUNCIONES**, **C. TRIANGLE: PARCHES ALABEADOS**, y **D. TRIANGLE: DESCRIPCIÓN DE FUNCIONES**.

Capítulo 2. RIPOLIN

2.1. Punto de partida

Patrick Nosmas realizó su PFC para *Télécom-Paris* durante un *stage* en la ENSAD (Escuela Nacional Superior de Artes Decorativas), donde reestructuró un programa de síntesis de objetos 3-D mediante *z-buffer* llamado *Ripolin*, que permitía visualizar escenas sencillas modeladas mediante polígonos, e interpolar el color en el interior de las facetas según los métodos de Gouraud y Phong[2]. La versión desarrollada por Nosmas fue concebida para correr sobre ordenadores personales compatibles con IBM-PC administrados por MS-DOS, por lo cual tenía serias limitaciones.

La más aparente era sin duda la que afectaba a la complejidad de los objetos tratables por el programa: los números máximos de facetas, vértices y aristas siendo respectivamente 1000, 2000 y 3000, los modelos debían ceñirse prácticamente a lo mínimo imprescindible. Además, no estaba implementada la posibilidad de completar imágenes previamente calculadas introduciendo en ellas nuevos objetos. Esta composición gradual de la escena, que es una ventaja inherente al mecanismo del *z-buffer*, permite tratar escenas complejas (i.e. con muchos objetos sencillos, o equivalentemente, pocos objetos voluminosos) aun cuando existan restricciones sobre el tamaño del modelo en cada ejecución individual del programa, como se explicó en el primer apartado del capítulo de introducción. Estas dos primeras limitaciones venían inevitablemente impuestas por la escasez de memoria típica de las n primeras versiones de MS-DOS, incapaces de gestionar más de los consabidos 640 Koctetos de memoria de base.

Por otra parte, la calidad de la imagen también se veía reducida al tener que trabajar obligatoriamente con pseudo-color (8 bpp). Esta restricción, determinada por la sobriedad cromática de tarjetas y monitores gráficos disponibles a finales de los '80 para ordenadores personales de estas características, exigía además realizar el cálculo de la paleta más representativa para la escena considerada. Las dimensiones de las imágenes tampoco se ajustaban a lo que hoy se considera como calidad profesional (no menos de 1024 x 1024 píxeles), puesto que usualmente eran de 512 x 512 píxeles, si bien es cierto que esta limitación era, como la anterior, atribuible a la resolución de los dispositivos gráficos, y no al programa en su esencia.

Finalmente, la determinación de los vectores normales a la superficie en sus vértices, necesaria para los cálculos de luminancia en los casos de interpolación del color de Gou-

[2] *Ripolin* es una marca de pintura laqueada, registrada por el holandés Riep en 1888. El creador del programa original la eligió seguramente por el aspecto plástico y brillante de los objetos tratados con el método de Phong.

raud y Phong, se realizaba de una manera aproximada y muy costosa en tiempo de cálculo. Y no se contemplaba la posibilidad de que el creador del modelo suministrara, en caso de conocerlos, esos vectores normales junto con el resto de datos de descripción de la superficie. Sobre este último punto es justo añadir que, en el marco de la ENSAD, era perfectamente lógico suponer desconocidas en principio las normales, dado que los modelos serían creados "a mano", y no matemáticamente, por diseñadores preocupados antes por la apariencia estética del resultado que por su rigor geométrico—artistas, al fin y al cabo...

2.2. Resultados obtenidos

Se realizaron dos versiones distintas de Ripolin: *RipLigne* produce la imagen, junto con su *z-buffer* asociado, en *modo línea*, como la antigua de Nosmas; y *RipPage* funciona en *modo página*. Es decir, que la primera versión calcula por separado cada línea de la imagen, y la escribe en el fichero de salida antes de pasar a la siguiente, mientras que la segunda mantiene en memoria en todo momento la imagen completa, y realiza una única operación de escritura al acabar la proyeccción de la escena. Esta última solución es lógicamente más exigente en cuanto a necesidades de memoria, pero también más rápida. Además, como se verá más adelante en el apartado 2.3 que describe el funcionamiento del programa, el modo página permite iterar de manera más eficiente sobre las facetas del objeto, lográndose una mejora sobre el tiempo de cálculo que se añade a la debida a la reducción del número de operaciones de E/S. Sin embargo, varios proyectos del Grupo *Image* necesitaban con urgencia una adaptación de Ripolin al nuevo entorno para verificar la corrección de sus resultados teóricos, por lo que se prefirió hacer mayor hincapié en la versión línea, que es la que más se desarrolló y probó, quedando RipPage como prototipo destinado a futuros proyectos de mejora. En todo caso, ambas versiones ofrecen las siguientes posibilidades.

El formato de descripción del modelo de la ENSAD era único y muy generalista: los polígonos podían tener un número arbitrario de lados, de manera que siempre había que especificar, para cada faceta, cuántas aristas la constituían. Esta solución, aunque flexible, es innecesaria si se sabe con certeza que la superficie está enteramente recubierta por una malla topológicamente regular, ya que en ese caso el número de aristas y vértices por faceta es constante. Así, se conservó el aparato de lectura de polígonos cualesquiera, pero se añadió además un módulo que lee n-gonos, pensando sobre todo en triángulos y cuadriláteros, dada la importancia de los parches de Bézier.

También se incluyó un mecanismo que permite al creador del modelo especificar, en caso de conocerlas, las normales exactas a la superficie en sus vértices. De esta manera, lo único que tiene que hacer el programa es leerlas de un fichero anejo al de descripción del objeto y almacenarlas en memoria, en lugar de calcularlas una y otra vez haciendo uso de aproximaciones.

Para poder aprovechar totalmente las ventajas que presenta el algoritmo de Catmull, se dotó al programa de la facultad de salvar el *z-buffer* final en un fichero junto con el de la imagen. Por supuesto, también se puede leer un *z-buffer* inicial para completarlo con nuevos objetos. Y para no obtener incoherencias cubistas indeseadas en la perspectiva, se guarda siempre en otro fichero aparte la información relativa a la cámara y a la iluminación. De hecho, si el usuario no proporciona un fichero de entrada de descripción de punto de vista e iluminación, el propio programa sitúa al observador de manera conveniente respecto a la escena, una vez leída ésta, y salva los parámetros escogidos para que puedan ser posteriormente modificados o reutilizados. El listado 2.1 corresponde a un posible fichero de este tipo (formato ASCII).

```
Paramètres de la caméra et de l'éclairage (angles en degrés):

  Point visé (centre repère): x, y, z =  0.000E+00  0.000E+00  6.403E+02
  Position caméra:     D, cap, azimut =  6.403E+02  0.000E+00 -9.000E+01
  Angles caméra:    ouverture, roulis =  2.000E+00 -9.000E+01
  Plans de "clipping": avant, arrière =  1.000E+00  2.000E+04
  Lumière ambiance:        amb1, amb2 =  5.000E-01  3.000E-01
  Source ponctuelle:      cap, azimut = -1.800E+02 -3.000E+01

Paramètres de la couleur:

  Couleur du fond:            R, V, B =    0    0    0
  Couleur de l'objet:         R, V, B =  150  150  150
  Reflet spéculaire:          R, V, B =  200  200  200
  "Chaleur" de l'objet:      exposant =   30
```

listado 2.1 - Ejemplo de fichero de parámetros de observación.

Por otra parte, el usuario elige entre crear una imagen en color real o en pseudo-color. En el primer caso, la salida del programa es un fichero que contiene el RGB de cada píxel; en el segundo, se producen dos ficheros: la imagen propiamente dicha, que en este caso no asocia a cada píxel un trío (R,G,B), sino un número de color de la paleta que mejor representa el cromatismo de la escena, y una *Look-Up Table* (LUT) que define el RGB de cada uno de los 256 colores de esa paleta. Es claro que, gracias al mecanismo de indirección, una imagen en pseudo-color ocupa prácticamente la tercera parte que una en color real (despreciando los 768 octetos que ocupa la LUT independientemente del tamaño de la imagen); pero, aunque esas 256 tonalidades puedan ser suficientes en algún caso, no se puede pretender completar escenas en pseudo-color dado que el programa calcularía dos paletas óptimas distintas, y los colores de la una no se corresponderían con los de la otra.

En cuanto al tratamiento del color, lo que desde luego se ha conservado es la posibilidad de realizar un *z-buffer* simple, o una interpolación de Gouraud o de Phong. En el listado 2.1 se puede apreciar que la iluminación se compone de una fuente puntual (*source ponctuelle*) que por una parte genera luz ambiente (*lumière ambiance*), y por otra hace que los objetos

brillen más o menos según el "calor" del material con que están hechos, cuando se emplea el tratamiento de Phong. De hecho, además del color natural del objeto, hay que precisar el del reflejo especular en su superficie, que no tiene por qué ser blanco. Normalmente, se suele pedir al programa que efectúe primero una síntesis rápida de la escena mediante un *z-buffer* simple, y luego interpole el color sin tener que repetir los cálculos de perspectiva y *clipping*.

Como se ha dicho más arriba, todas estas características son compartidas por RipLigne y RipPage, que únicamente se diferencian en el modo de producción de la imagen, y en sus limitaciones de memoria. Así, tras la adaptación, RipLigne es capaz de visualizar objetos de 20 Kfacetas, 20 Kvértices y 30 Karistas[3], superando en más de un orden de magnitud la anterior limitación de tamaño del modelo, y permitiendo tratar superficies complejas de manera satisfactoria. Esto es posible, además, con resoluciones de 1024 píxeles por línea en color real, y almacenando la profundidad de cada píxel (el valor de su *z*) en variables de tipo *real* de 4 octetos. Por su parte, RipPage debe mantener constantemente en memoria principal tanto la imagen como el *z-buffer*, lo que representa (24 + 32) bpp, i.e. 7 Moctetos para una imagen de 1024 x 1024 píxeles. En esta resolución, no tolera objetos de más de 1000 facetas, aunque puede llegar a 2000 para imágenes de 512 x 512. De todas maneras, para escenas de complejidad media como éstas (unos miles de polígonos), el tiempo debido a operaciones de E/S representa una fracción no despreciable del total, incluso en la versión página, debido a que el cálculo de la imagen es rápido. Y por ello, la ventaja en rapidez de ejecución de RipPage sobre RipLigne (entre el 15 y el 20% en este caso) no compensa siempre satisfactoriamente la limitación de talla del modelo.

2.3. Descripción del programa

En este apartado se explica cómo se desarrolla la ejecución de RipLigne desde dos puntos de vista: primero según lo que percibe el usuario, y después según lo que ocurre de verdad internamente. En esta segunda parte se presentan además las estructuras de datos y variables del programa, se describen funcionalmente sus módulos principales, y se discuten las diferencias existentes entre las dos versiones (RipLigne y RipPage). La matemática involucrada en transformaciones de perspectiva, modelado de la luz, e interpolación del color, se encuentra en el anejo A; el anejo B, por su parte, describe extensivamente los procedimientos y funciones del programa.

[3] Estos valores están en la proporción 2:2:3 como solución de compromiso entre 2:1:3 (mallado infinito regular del plano con triángulos) y 2:2:4 (id. con rectángulos), porque se supuso que los objetos estarían modelados mayoritariamente con polígonos sencillos, y no con hexágonos (2:4:6), por ejemplo.

2.3.1. Interfaz con el usuario

El usuario no invoca directamente al ejecutable `RipLigne.EXE`, sino a un fichero de comandos llamado `RipLigne.COM`. Éste es quien lo invoca realmente, tras mantener con el usuario un diálogo que tiene por fin preparar los ficheros de entrada necesarios para la ejecución de `RipLigne.EXE`.

Lo primero que pide `RipLigne.COM` es el nombre del fichero que contiene la descripción del objeto. El tipo (la extensión) de este fichero determina el formato del modelo. Si es `.BIN`, se entiende que la superficie está descrita según el formato antiguo de polígonos cualesquiera (ENSAD), en un único fichero que contiene las coordenadas de los vértices y la información topológica (`.NRM` indica que se suministran además las normales). Sin embargo, en el caso de los nuevos formatos de polígonos de orden constante, se utilizan dos ficheros asociados, de mismo nombre y distinta extensión: uno numera los vértices, listando sus coordenadas 3-D, y otro relaciona cada faceta con los vértices que la componen. En el caso de los triángulos, las extensiones respectivas de estos dos ficheros son `.T_S`[4] (`.T_N` si figuran las normales), y `.T_F`; en el de los cuadriláteros, `.C_S` (`.C_N`) y `.C_F`. Después de verificar la existencia del (o de los) fichero(s) del modelo, `RipLigne.COM` intenta hallar un fichero con el mismo nombre y extensión `.OBS`, como el del listado 2.1, que contendría los parámetros asociados al observador, salvados tras alguna visualización anterior de la misma escena. Por ejemplo, si se responde que el fichero del objeto es `cubo.t_s`, el programa se encarga de buscar también `cubo.t_f` y `cubo.obs`.

Hecho esto, `RipLigne.COM` pregunta si se desea trabajar en pseudo-color o color real. En el primer caso, informa sobre los nombres de los ficheros que serán producidos (en nuestro ejemplo, `cubo.IMA` para la imagen, y `cubo.3LT` para la LUT); en el segundo, también pregunta si se trata de completar una escena antigua o de generar una nueva. Si entonces se le da un nombre de fichero, él verifica la existencia de la imagen (`.RVB`, por *Rouge, Vert, Bleu*) y del *z-buffer* correspondiente (`.ZBF`). A continuación, y siempre dentro del caso de color real, pregunta si será preciso salvar el *z-buffer* final además de la imagen, y comunica después los nombres de los ficheros de salida (continuando con el ejemplo, `cubo.RVB` y, eventualmente, `cubo.ZBF`).

Afortunadamente, es posible pasar parámetros a `RipLigne.COM` al invocarlo para evitar todo (o al menos parte de) este pesado diálogo. La sintaxis, que él mismo resume cuando es llamado sin argumento ninguno, es:

```
ripligne data [bits [zbfin [zbfout]]]

parámetro obligatorio: data        (fichero del objeto)
parámetros opcionales: bits        (por defecto: 24 bpp)
                       zbfin, out  ( "     "     ninguno)
```

Una vez recopilada esta información, `RipLigne.COM` la salva en un fichero temporal de opciones, y llama a `RipLigne.EXE`, que la lee y pide al usuario, además, detalles so-

[4] La elección de la letra "S" para el primer fichero se debe a que en francés, vértice se traduce por *sommet*.

bre el tamaño de la imagen, en el caso de que sea nueva, y sobre el tratamiento del color. Estas preguntas no se incluyen en la fase previa para permitir realizar, sobre un mismo objeto y dentro de una misma sesión, una prueba rápida de baja calidad antes de producir un resultado definitivo. Y finalmente, tras la lectura de los parámetros de cámara e iluminación (si existía un fichero `.OBS`), o su elección juiciosa a tenor del tamaño y la posición del objeto (en caso contrario), comienza verdaderamente la síntesis de la escena.

Si el programa es lanzado en modo interactivo, mantiene informado de sus progresos al paciente usuario, que va leyendo esperanzado cómo se completa, línea a línea, la ansiada imagen. Pero si, por falta de tiempo que perder, se desea ejecutar `RipLigne.EXE` en *batch*, es posible hacerlo indicándoselo a `RipLigne.COM`, que entonces pregunta la talla de la imagen y el tratamiento del color, y añade las respuestas al fichero temporal de opciones. En este caso, la información sobre el desarrollo de la síntesis, junto con los tiempos parciales de cálculo y escritura, queda registrada en un fichero (`RipLigne.LOG`) que crea automáticamente el sistema operativo al encolar la tarea.

2.3.2. Funcionamiento interno

2.3.2.1. Lectura de Datos

Como ya se ha explicado, `RipLigne.COM` lanza la ejecución de `RipLigne.EXE` tras haber preparado la información que éste necesita para la síntesis de la escena en un fichero de opciones, cuya existencia es temporal y transparente al usuario, dado que una vez terminada la ejecución de `RipLigne.EXE`, `RipLigne.COM` lo destruye. Este fichero contiene la siguiente información cuando se corre el programa en modo interactivo:

- ⋆ nº de aristas por faceta (0/3/4; 0 para formato ENSAD de polígonos cualesquiera);
- ⋆ especificación de normales (TRUE / FALSE);
- ⋆ color real (TRUE / FALSE);
- ⋆ *z-buffer* de entrada (TRUE / FALSE);
- ⋆ *z-buffer* de salida (TRUE / FALSE);
- ⋆ fichero de parámetros de observación (TRUE / FALSE).

Cuando se decide lanzar el ejecutable en *batch*, debe contener además las respuestas a las preguntas que tendrían lugar en modo interactivo, es decir:

- ⋆ anchura de la imagen (píxeles por línea);
- ⋆ altura de la imagen (líneas);
- ⋆ tratamiento deseado para el objeto (Z/G/P según sea Z-*buffer*, Gouraud o Phong).

En el listado 2.2 se muestra el fichero de opciones, de formato ASCII, que resultaría de la siguiente situación: el usuario, que ha producido previamente una imagen de una esfera (`esfera.rvb`) salvando el *z-buffer* final (`esfera.zbf`) y los parámetros de observación (`esfera.obs`), quiere añadir un cubo a su escena, que tiene modelado mediante triángulos sin normales en `cubo.t_s` y `cubo.t_f`, para generar `esf_cubo.rvb` y

esf_cubo.zbf. Para ello, copia primero esfera.obs en cubo.obs y luego invoca a RipLigne.COM tecleando "ripligne cubo.t_s 24 esfera esf_cubo". Tras verificar la existencia de cubo.t_s, cubo.t_f, cubo.obs, esfera.rvb, y esfera.zbf, RipLigne.COM pregunta al usuario si desea correr RipLigne.EXE interactivamente, y éste —que está a punto de agotar su cuota de CPU en modo interactivo— responde que prefiere lanzarlo en *batch*, y se somete al cuestionario sobre tamaño de la imagen y "pulido" del color, diciendo que quiere 1024 x 768 píxeles[5], y "pulido" de Phong.

```
Nombre d'arêtes par face            =  0
norm, VraieCouleur, zin, zout, obs  =  FALSE  TRUE  TRUE  TRUE  TRUE
1024
768
P
```

listado 2.2 - Ejemplo de fichero temporal de opciones.

Otra función muy importante de pre-procesamiento que realiza RipLigne.COM es la de asignar nombres de ficheros físicos (los que maneja el usuario) a los nombres lógicos que utiliza RipLigne.EXE (FORxxx): es costumbre en el Laboratorio IMA que sea un fichero de comandos, y no el propio ejecutable, quien realice las tareas de gestión de ficheros, porque ello facilita el acceso a las potentes primitivas que provee el sistema operativo VMS. Así, antes de lanzar la ejecución de RipLigne.EXE, RipLigne.COM define una serie de equivalencias de acuerdo con los valores internos manejados por el programa RipLigne, que son los siguientes:

FichOptions	=	FOR010	(fichero temporal de opciones);
FichObj	=	FOR011	(descripción objeto formato ENSAD);
FichSommets	=	FOR012	(id. polígonos fijos - info. vértices [y normales]);
FichFaces	=	FOR013	(id. id. - info. facetas);
FichParamObs	=	FOR014	(fichero de parámetros de observación);
FichImaIn	=	FOR015	(imagen de entrada para completar);
FichZbuffIn	=	FOR016	(*z-buffer* de entrada para completar);
FichImaOut	=	FOR020	(imagen de salida);
FichZbuffOut	=	FOR021	(*z-buffer* de salida);
FichLUT	=	FOR022	(*Look-Up Table*);

[5] En el caso de que las dimensiones pedidas para la imagen de salida no coincidan con las de la que debe ser completada, prevalecen las de esta última. De hecho, en modo interactivo, y cuando hay que completar una imagen, RipLigne.EXE ni siquiera pregunta las dimensiones de la imagen de salida.

`RipLigne.COM` no asigna siempre nombres de ficheros físicos a todos y cada uno de los nombres lógicos `FORxxx` mencionados más arriba (de hecho, algunos son mutuamente excluyentes). En el ejemplo anterior, `RipLigne.COM` hubiera definido:

```
FOR010 = ripligne.tmp;          FOR015 = esfera.rvb  ;
FOR012 = cubo.t_s    ;          FOR016 = esfera.zbf  ;
FOR013 = cubo.t_f    ;          FOR020 = esf_cubo.rvb;
FOR014 = cubo.obs    ;          FOR021 = esf_cubo.zbf;
```

Una vez que `RipLigne.EXE` ha leído el fichero de opciones, procede a la lectura del objeto, llamando al módulo que corresponda al formato del modelo especificado mediante el número de aristas por faceta (`i` en el programa): `Lec_Obj_ENSAD (FichObj)` para `i=0`; o `Lec_Obj_Ngones (i, FichSommets, FichFaces)` para `i=3` ó `4`. En ambos casos se trata de rellenar las tres matrices principales que almacenan la descripción del objeto (cuyos números totales de vértices, facetas, y aristas son respectivamente `ns`, `nf`, y `na`):

`xyz (coord, som)`	coordenadas del vértice nº `som`:	`coord` = `1/2/3` (*x/y/z*); $1 \le$ `som` $\le$ `ns`;
`nbaf (face)`	nº aristas de la faceta nº `face`:	$1 \le$ `face` $\le$ `nf`;
`ias (arete)`	nº del primer vértice de la arista nº `arete`:	$1 \le$ `arete` $\le$ `na`.

Si se desea obtener la coordenada *z* del segundo vértice de la faceta nº 42, por ejemplo, hay que proceder así:

1. `arete = nbaf (1) + nbaf (2) + ... + nbaf (41) + 2;`
2. `som = ias (arete);`
3. `z = xyz (3, som).`

Esta estructura de datos, un tanto enrevesada, no está sólo impuesta por el formato libre de polígonos cualesquiera: como se explica más abajo, en el apartado referente al *clipping*, es una estructura plenamente dinámica dado que el número de vértices de una faceta puede variar a lo largo de la ejecución del programa. Sin embargo, sí es cierto que, en el caso de que se esté trabajando con un objeto modelado únicamente con triángulos, todas las facetas tienen inicialmente tres aristas, i.e. `nbaf (face) = 3` $\forall$ `face`, con lo que el paso 1 se simplifica notablemente, convirtiéndose en `arete = 3*41 + 2` para nuestro ejemplo.

El fichero único de descripción del objeto de formato ENSAD, que es binario, contiene, además de la información correspondiente a esas tres matrices, los colores individuales de cada faceta (básico y de reflejo especular), así como el del fondo, y eventualmente las coordenadas de las normales en cada uno de los vértices—aunque, como ya se explicó en el apartado 2.1, la inclusión de las normales no cae dentro de la filosofía de este tipo de modelado. Por su parte, los ficheros de vértices y facetas del formato de polígonos fijos, también binarios, contienen única y respectivamente la información correspondiente a `xyz` (y en su caso, a las normales) e `ias`. El color de todo el objeto se supone único, y se precisa en el fichero de parámetros de observación junto con el del fondo.

A continuación `Lec_Param_Obs` lee el fichero de parámetros de observación, si existe (`RipLigne.COM` pasó `obs=TRUE`); si no existe, coloca la cámara y la luz de manera que la escena esté bien iluminada y encuadrada, y salva los parámetros escogidos para que el usuario los modifique si no son de su agrado. El formato de este último fichero, de texto ASCII, es el ya mostrado en el listado 2.1.

2.3.2.2. PREPARACIÓN MATEMÁTICA DE LA ESCENA

Después de leer todos los ficheros de entrada, el programa realiza, para cada uno de los vértices del objeto, los cálculos de luminancia y perspectiva. Los últimos consisten en la transformación de las coordenadas proporcionadas por el creador del modelo, que están referidas a un origen absoluto arbitrario, en coordenadas ligadas a la cámara, para poder proyectarlas después más cómodamente sobre el plano de la imagen, según queda descrito con detalle en el anejo A.

A continuación, dentro de este bloque de preparación matemática de la escena, tiene lugar el *clipping*, fase que realiza el módulo `CalClip`, y en la que se recortan aquellos segmentos que se salen del cuadro. Este es un proceso que puede consumir mucho tiempo de cálculo cuando hay que recortar efectivamente un número grande de segmentos, dado que en ese caso la gestión de las tres matrices que describen el objeto es compleja: aparecen nuevos vértices, desaparecen otros, crecen o menguan los números de aristas de algunas facetas, etc. En concreto, para poder cambiar los parámetros de la cámara (posición y ángulos de visión y alabeo), sin que ello obligue a releer los ficheros de entrada, lo que se hace es duplicar el objeto al proyectarlo: se almacenan en `xyz2`, `nbaf2`, e `ias2` los valores resultantes de los cálculos de la perspectiva en curso, y se realizan sobre estos duplicados las modificaciones introducidas por el encuadre y los cálculos posteriores; si al finalizar la síntesis de la escena, el usuario solicita modificar algún parámetro de observación, sólo hay que retomar los valores leídos originalmente en `xyz`, `nbaf`, e `ias` para proyectar de nuevo y volver a calcular el *clipping*. Un caso aún más sencillo es aquél en que el usuario sólo desea repetir la síntesis con otro método de interpolación del color: entonces, se pueden conservar los valores duplicados, y únicamente se deben repetir los cálculos de luminancia.

A todo lo anterior, sigue la ordenación de las facetas según su profundidad (*z-sorting*) por `AffIni` para que las más cercanas al observador sean procesadas después que las más lejanas, ocultándolas. Este también es un proceso relativamente lento dado que no se aplica un algoritmo especialmente eficiente en la ordenación (tipo *quicksort* o similar).

2.3.2.3. PRODUCCIÓN DE LA IMAGEN

Finalmente, comienza la producción de la imagen propiamente dicha, que es la fase que diferencia realmente los algoritmos de los modos *línea* y *página*: no sólo son distintas sus

maneras de escribir las líneas en el fichero de salida (de una en una o todas de una vez), sino, sobre todo, sus filosofías internas. RipLigne itera sobre las líneas, comenzando por la más alta y siguiendo hacia abajo, mirando qué facetas son intersectadas por la línea en curso (llamadas *facetas activas*), en un segundo bucle interior al primero; mientras tanto, RipPage itera más bien en el orden inverso: primero sobre las facetas, activando las lejanas antes que las cercanas, y luego sobre las líneas correspondientes a la faceta activa, de arriba abajo. Esta segunda manera de anidar los bucles, que es la tradicional en el *z-buffer*, es mucho más eficiente porque una línea sólo se procesa dos veces si dos facetas la tocan[6], pero sólo es posible cuando toda la imagen está simultáneamente en memoria. En la primera, una vez que se ha dado por buena (y añadido al fichero de salida) una línea, ya no hay marcha atrás: por eso es necesario recorrer la lista completa de facetas para cada línea, aunque ello consuma un tiempo extra que, en la mayor parte de los casos, es comparable al que ya se pierde por realizar tantas operaciones secuenciales de E/S como líneas tenga la imagen.

El trabajo de coordinación de funciones en esta fase lo lleva el módulo `jZbuff`, que llama a `FacActi` para activar las facetas cuando proceda, y a `jGouraud` o `jPhong` si el usuario seleccionó alguno de los métodos de interpolación del color. Por otra parte, si se debe completar una imagen, también llama a `LecIma`[7] para leer imagen y *z-buffer* (línea a línea en RipLigne, antes de procesar cada nueva línea; de una vez en RipPage, antes de procesar la primera faceta). Y, evidentemente, al acabar de producir una línea (RipLigne), o la imagen completa (RipPage), el programa la salva en disco llamando a `EcrIma`[7], junto con el *z-buffer* correspondiente, si es el caso.

---

[6] En realidad, ni siquiera en ese caso tiene por qué ser reprocesada la totalidad de la línea, dado que, para cada faceta activa, se barren únicamente los segmentos intersectados por ella.

[7] `LecIma` y `EcrIma` son rutinas de lectura y escritura de imágenes de la librería del Labo. IMA.

Capítulo 3. TRIANGLE

Como se explicó en el capítulo de introducción, los parches alabeados han cobrado mucha importancia en el modelado de objetos 3-D por su poder descriptivo, que permite una gran adaptabilidad a superficies complejas, a la vez que introduce una economía considerable en el volumen de información necesario para obtener una calidad de síntesis elevada, respecto al modelado tradicional con facetas planas. Sin embargo, cuando el programa de *rendering* del que se dispone sólo acepta como entrada polígonos planos, como es el caso de Ripolin, es preciso construir un puente entre ambos formatos, que "facetice" la superficie obtenida con los parches alabeados. La forma más sencilla de hacerlo, que es la utilizada por los *algoritmos de subdivisión*, consiste en fragmentar recursivamente cada uno de los parches originales en varios subparches hasta que pueda ser aproximado por un polígono plano, momento en el que puede ser entragado al programa visualizador. El problema es entonces elegir acertadamente los criterios según los cuales se tomará la decisión de subdividir o no, e implementarlos mediante pruebas que determinen, de manera efectiva, la bondad de la eventual aproximación del parche considerado por un polígono. Es decir, subdividir según una *estrategia adaptativa*, y no de forma sistemática.

Para entender cómo funcionan estos algoritmos, es necesario hacer primero una breve presentación geométrica de los tipos de parches más empleados, y de las dificultades que conlleva su subdivisión. La figura 3.1 muestra el aspecto general de una de estas familias de superficies: los cuadrados bicúbicos Bézier. Como se explica más detalladamente en el anejo C, tanto estos cuadrados como sus primos hermanos los triángulos de Gregory (cf. figura 3.2) tienen por bordes curvas de Bézier de tercer grado, y presentan propiedades geométricas muy interesantes. Los parches se suelen definir mediante una ecuación superficial paramétrica, que da las tres coordenadas espaciales (x, y, z) de todo punto de la superficie en función de sus dos *coordenadas paramétricas normalizadas* (u, v)[8] y del *grafo de control 3-D* del parche, un conjunto de puntos de (x, y, z) conocidas y prefijadas. Es muy importante observar cómo, en ambos casos, los puntos de control de las esquinas son los únicos que necesariamente pertenecen a la superficie, mientras que los demás "tiran" de ella hacia sí, moldeando el parche según la rejilla tridimensional que forman.

[8] En el caso de los triángulos, resulta más cómodo trabajar con tres coordenadas paramétricas (u, v, w) que, obviamente, no son independientes—cualquier superficie que se precie tiene sólo dos grados de libertad—sino que verifican la identidad « u+v+w ≡ 1 ».

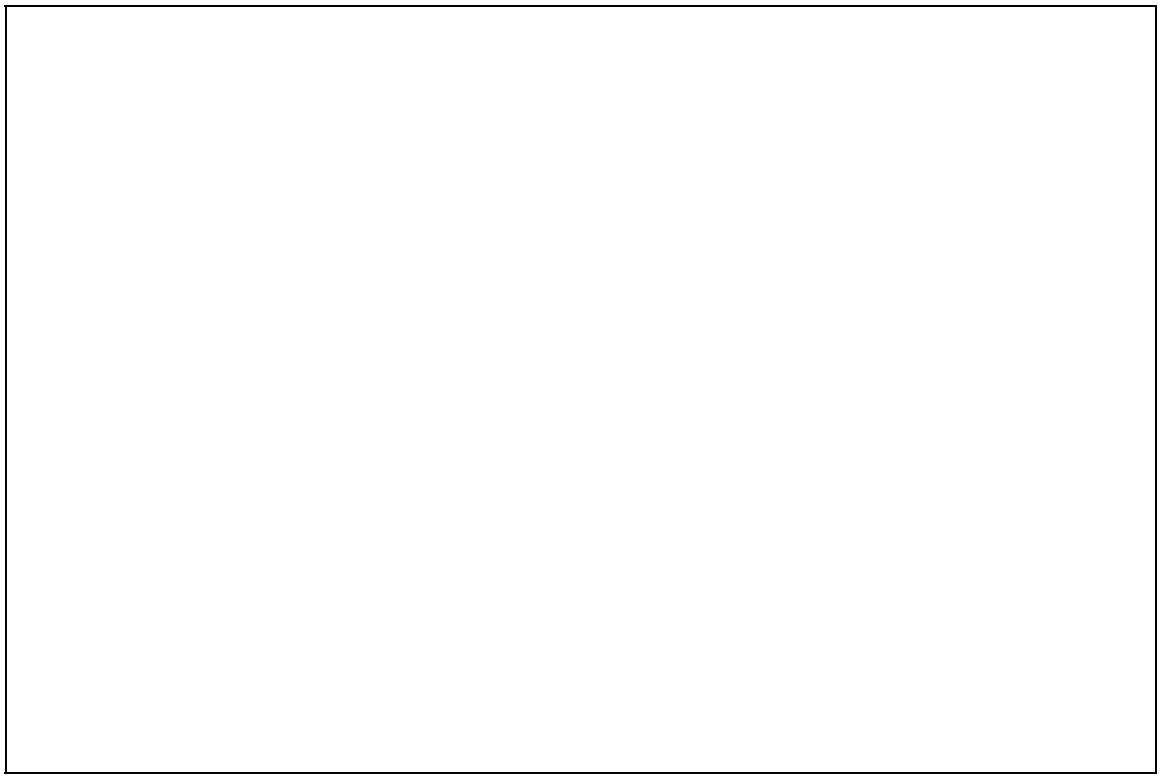

figura 3.1 - Cuadrado bicúbico de Bézier: dominio paramétrico y grafo de control.

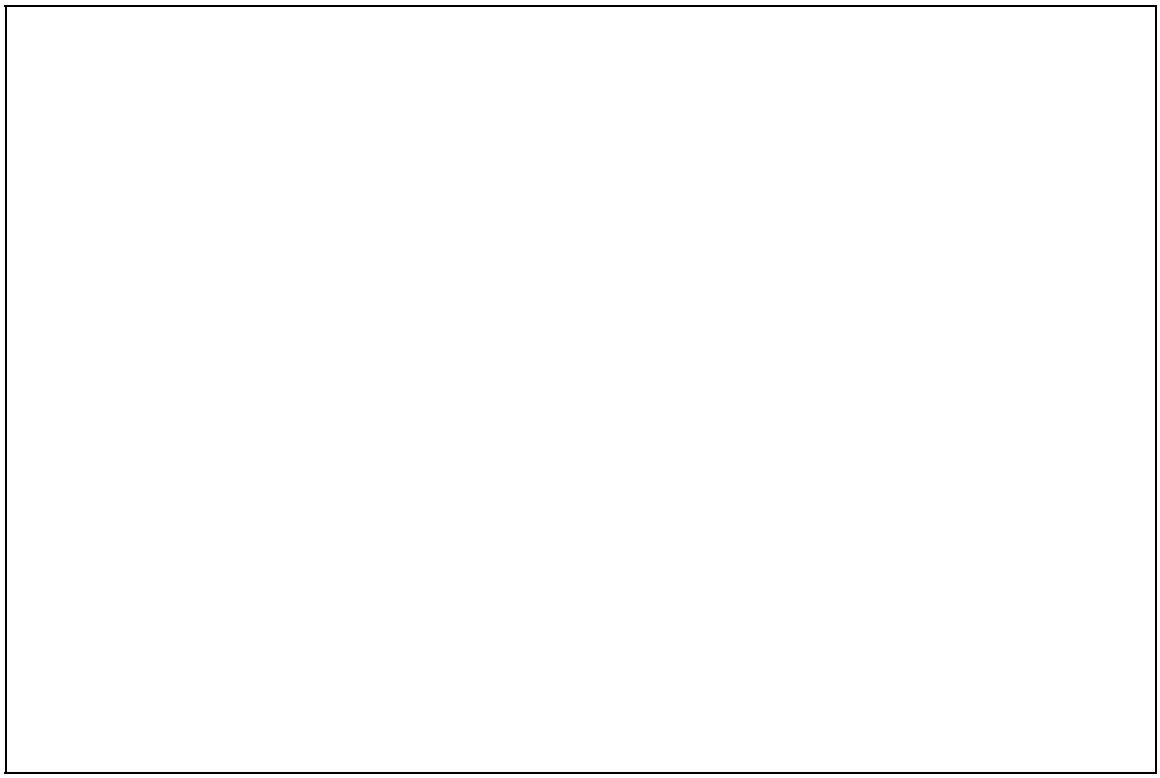

figura 3.2 - Triángulo cuártico de Gregory: dominio paramétrico y grafo de control.

Por eso, cuando se decide finalmente aproximar el parche por una faceta plana, se hace precisamente con el polígono formado por los puntos de control de las esquinas (un cuadrilátero en el caso de los cuadrados de Bézier; un triángulo en el de los parches de Gregory). Sin embargo, cuando el parche examinado no ha satisfecho los requisitos exigidos para convertirse en faceta, porque no es suficientemente *plano* según alguna medida dictada por los criterios establecidos de antemano, debe ser dividido en varios subparches, cada uno de los cuales será más *plano* que el original. Esto último es lo que garantiza la convergencia del algoritmo aun cuando éste sea recurrente. En el caso de los cuadrados de Bézier, la subdivisión puede ser, por ejemplo, dicotómica: una vez determinada la dirección óptima de división, se toma como frontera de separación la curva que une los puntos medios (en el dominio paramétrico) de los bordes divididos; en el caso de los triángulos de Gregory, lo usual es fragmentar cada parche en cuatro, uniendo también los puntos paramétricos medios de los lados originales. Las figuras 3.3.a y 3.3.b muestran cómo se aproximaría cada uno de esos parches (polígonos sombreados), y cómo se dividiría (líneas y puntos gruesos).

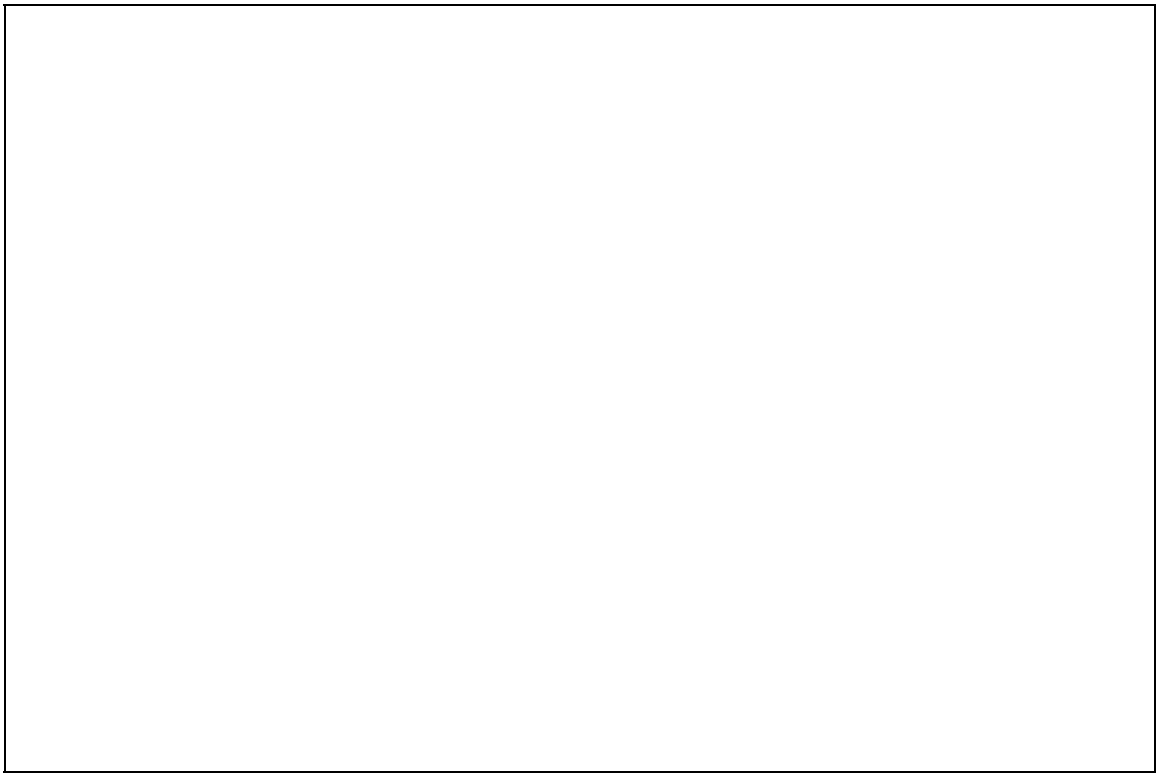

figura 3.3 - "Facetización" o subdivisión de:
a) un cuadrado de Bézier; b) un triángulo de Gregory.

Cuando la subdivisión de cada parche original no se realiza sistemáticamente (un número de veces prefijado, típicamente), sino de manera adaptativa allí donde más se necesita, surge un problema importante: la posible aparición de **grietas** (*crackings*) en la superficie facetizada resultante, que se produce cuando dos parches colindantes no son fragmentados por igual. La figura 3.4 ilustra el fenómeno de agrietamiento en el caso de un par de cua-

drados de Bézier de los cuales uno puede ser directamente facetizado, mientras que su vecino necesita al menos una subdivisión suplementaria. En el apartado 3.3, que describe el funcionamiento del programa, se presentan y comparan dos posibles soluciones genéricas (independientes del tipo de parches utilizados en el modelado) a este problema, y se detalla la implementación de la adoptada para los triángulos de Gregory.

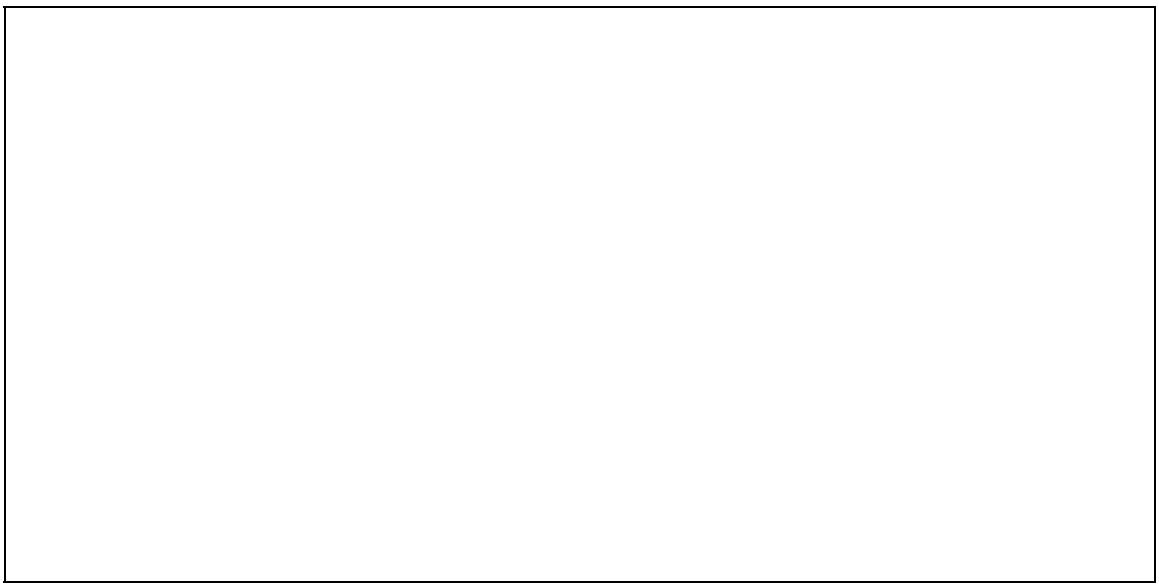

figura 3.4 - El problema del agrietamiento (*cracking*) de la superficie.

3.1. Punto de partida

Éric Burgstahler había desarrollado en 1989, para el Depto. IMA de *Télécom-Paris*, un algoritmo de subdivisión adaptativa de cuadrados bicúbicos de Bézier inspirado en el excelente artículo de Reed D. Clay y Henry P. Moreton ([CLAY-88]), en el que explican cómo solucionaron sus necesidades de visualizar superficies modeladas con parches de este tipo a velocidades de animación cinematográfica[9]. Aunque los resultados del programa de Burgstahler, *Métamorphose*, eran satisfactorios en la mayoría de los casos, sólo se había utilizado realmente para verificar la pertinencia de los distintos criterios de subdivisión citados en dicho artículo, pero nunca para facetizar una superficie compleja compuesta por un número importante de parches. De cualquier modo, Métamorphose no consideraba más que los cuadrados bicúbicos de Bézier, y dado que el interés del Labo. IMA había pasado, en 1990, a centrarse en los triángulos cuárticos de Gregory, era imperativo diseñar y es-

[9] i.e. 24 imágenes por segundo—claro que, en justicia, hay que precisar que Clay y Moreton trabajaban con una *workstation* IRIS 4D GT de 10 Mips, diseñada específicamente para proceso de gráficos en tiempo real.

cribir una versión específica para estos últimos. La tarea exigía un estudio comparativo detallado de ambos tipos de parches que permitiera determinar qué partes del algoritmo existente eran reciclables y cuáles no[10].

3.2. Resultados obtenidos

El programa *Triangle* desarrollado como parte de este Proyecto sigue también de cerca las directrices del artículo de Clay y Moreton: se adoptaron todas las posibles razones de subdivisión de un parche que allí se describen, aunque las pruebas correspondientes a los distintos criterios, así como la estrategia misma de la fragmentación, hubieron de ser redefinidas para adaptar mejor los resultados a las circunstancias específicas del Proyecto. Y es que Triangle se diseñó pensando en una futura integración en Ripolin, como un módulo de pre-procesamiento de los objetos modelados con triángulos de Gregory. Por ello, el programa propone valores para los límites de tolerancia que favorizan aquellos motivos de subdivisión (como los que tienen lugar en la silueta del objeto) que optimizan el comportamiento global del tándem Triangle-Ripolin.

Sin embargo, esto no afecta a la validez general del algoritmo en sí, ya que es el usuario quien fija en definitiva los límites de tolerancia para cada una de las razones de subdivisión, imponiendo en todo momento su propio orden de prioridades. Incluso, gracias al esfuerzo de modularidad llevado a cabo durante la escritura del programa, nada impide reemplazar los *tests* matemáticos implementados por otros hechos "a medida", como queda explicado en el apartado 3.3.2.3. En él se describe con mayor detalle la estrategia de la fragmentación de cada triángulo original, que en resumidas cuentas, consiste en someterlo a las siguientes pruebas (un resultado positivo en cualquiera de ellas provoca la subdivisión del parche):

1. **bordes curvos**: alguno de los tres bordes no es suficientemente plano;
2. **bordes silueta**: alguno de los tres bordes cruza la silueta del parche;
3. **bordes frontera**[11]: alguno de los tres bordes es un segmento de uno de los bordes del parche original;
4. **interior alabeado**: aun siendo sus tres bordes satisfactorios, el interior del parche no es suficientemente plano.

---

[10] Dado que, sin que hasta la fecha se sepa por qué, una primera aproximación al problema, consistente en cambiar todos los `4`'s del programa por `3`'s, no dio los resultados esperados.

[11] Aunque este criterio de subdivisión parezca arbitrario, puede ser deseable aplicarlo cuando el parche original no tiene vecinos sobre la superficie del objeto porque ésta es abierta.

3.3. Descripción del programa

En este apartado se explica cómo se desarrolla la ejecución de Triangle desde dos puntos de vista: primero según lo que percibe el usuario, y después según lo que ocurre de verdad internamente. En esta segunda parte se presenta la solución al problema del agrietamiento de la superficie, y la estrategia del algoritmo de subdivisión. Además, se describen las estructuras de datos del programa, y se detalla la implementación de los *tests* correspondientes a cada una de las razones por las que se puede fragmentar un parche. Así como en el caso de Ripolin el usuario no necesita saber nada de los manejos internos del programa, en el de Triangle es conveniente que conozca al menos la estrategia de subdivisión, y el significado de los distintos límites de tolerancia, para poder controlar con mayor eficacia el resultado obtenido al modificarlos. Por otra parte, las precisiones geométricas sobre los parches alabeados pueden encontrarse en el anejo C; y la descripción exhaustiva de los procedimientos y funciones de Triangle, en el anejo D.

3.3.1. Interfaz con el usuario

Como se explicó en el apartado 2.3, es costumbre en el Labo. IMA que el usuario invoque al ejecutable a través de un fichero de comandos, que realiza antes de lanzar al programa principal las tareas de comprobación de existencia de ficheros y de asignación de nombres lógicos. Así, no se debe llamar a `Triangle.EXE`, sino a `Tri.COM`, según la sintaxis siguiente:

```
tri patchs [plot]

parámetro obligatorio: patchs     (fichero del objeto)
parámetro opcional:    plot (sic) (imprime las facetas
                                   resultantes en LPS40)
```

El nombre del fichero de los triángulos de Gregory que modelan el objeto se debe teclear sin extensión, porque se le supone "`.T_G`"; la opción de salida por impresora sirve para volcar a papel el *wire-frame* que opcionalmente muestra Triangle una vez facetizada la superficie del objeto. Suponiendo que se le haya invocado con "`tri victor`", y de manera análoga a lo que hace `RipLigne.COM`, `Tri.COM` busca "`victor.T_G`", que será el fichero de entrada para `Triangle.EXE`, y definirá las equivalencias lógicas pertinentes para que el ejecutable grabe los resultados en "`victor.T_N`" (coordenadas de vértices y componentes de normales), "`victor.T_F`" (información relativa a las facetas), y "`victor.OBS`" (parámetros de observación). Estos tres ficheros de salida serán los de entrada para Ripolin. Nótese que, al ser necesarios los cálculos de las normales para subdividir los parches de Gregory, el creador del modelo que se entrega a Ripolin (en este caso, Triangle) puede facilitárselas, por lo que se genera un fichero del tipo "`.T_N`", y no "`.T_S`".

Hecho todo esto, toma control `Triangle.EXE`, que presenta una pantalla de diálogo con los valores por defecto de los parámetros de observación, y de los límites de tolerancia para cada uno de los criterios de subdivisión. El usuario puede entonces modificar estos

valores a su antojo y solicitar información en tiempo real sobre el desarrollo de la conversión, y finalmente ordenar el comienzo de la facetización de la superficie. Una vez completada ésta, el programa presenta un resumen (números totales de vértices y facetas; número de subdivisiones por cada razón), y ofrece la posibilidad de visualizar en *wire-frame*, y siempre que se disponga de una pantalla gráfica, los resultados obtenidos.

3.3.2. Funcionamiento interno

3.3.2.1. Solución al Problema del *Cracking*

La rotura de la superficie se produce cuando dos parches vecinos son subdivididos un número distinto de veces: entonces, al ser facetizados los subparches resultantes, el borde curvilíneo que compartían los parches originales no es aproximado de la misma manera de un lado y de otro, y aparece la discontinuidad indeseada. Para detectar la posibilidad de aparición de grietas de este tipo, y así poder prevenirla, sería necesario saber si los parches colindantes con el que se considera en un momento dado van a ser subdivididos. Es decir, hacer una primera inspección de toda la superficie sin facetizar ni subdividir ningún parche, sino almacenando los resultados de las pruebas efectuadas sobre cada parche para crear una especie de *árbol lógico* de la subdivisión, que permitiera posteriormente tomar las acciones adecuadas. Pero esta solución, además de ser inherentemente lenta, por exigir dos pasadas sobre el conjunto de parches originales, es nefastamente cara en requisitos de memoria, ya que esa información de vecindad puede alcanzar un volumen prohibitivo: en el artículo de Clay y Moreton, por ejemplo, la famosa tetera de SIGGRAPH se aproxima con unas 2500 facetas, partiendo de tan sólo 32 cuadrados de Bézier.

La alternativa es basarse únicamente en las características propias de cada parche a la hora de decidir qué hacer con él, economizando tiempo y memoria. Además, dado que el objetivo último de este Proyecto era lograr que Ripolin pudiera leer los objetos parche a parche (y facetizarlos él mismo), para sobrepasar cualquier limitación en la talla del modelo, ésta parecía la solución idónea. Ahora bien, en este caso, la única información disponible sobre los parches vecinos del considerado es la suministrada por el borde frontera que comparten. Así que, inicialmente, se toma la decisión de subdividir un parche según los resultados de pruebas realizadas sólo sobre sus bordes. Claro que esto no impide por sí solo que aparezcan grietas a lo largo de un borde cuando no es él el causante de las fragmentaciones de los parches que separa: hay que introducir una información adicional que permita que las decisiones que conciernen a un borde se repitan independientemente del lado del que es considerado. Lo que hace Triangle es añadir a la información espacial del grafo de control del parche, variables lógicas que marcan a cada uno de sus bordes como "satisfactorio" o no, según sus resultados en las distintas pruebas. Cuando un borde es finalmente dado por bueno (y es importante insistir en que esto sólo depende de él mismo), toda eventual fragmentación que deba sufrir (por causas ajenas a él) lo tomará por un segmento rectilíneo. Obviamente, mientras no haya alcanzado aún ese *status*, es considerado como el segmento curvilíneo que en realidad es, para que sus sucesivas subdivisiones produzcan bordes cada vez más "satisfactorios". La figura 3.5 muestra lo que ocurriría en el

caso de dos parches vecinos que comparten una frontera suficientemente recta, que debe ser fragmentada en el parche de la izquierda (por culpa de los otros bordes), y no en el de la derecha (que es todo él aceptable).

figura 3.5 - Solución al problema del agrietamiento (*cracking*) de la superficie.

Pero esto no basta, porque una vez que todos los bordes del parche han sido validados, todavía puede ser que su interior no sea lo suficientemente plano como para ser aproximado por una faceta. Consecuentemente, a los *tests* sobre los bordes sigue uno sobre el interior: si el parche lo pasa, se convierte por fin en faceta; si no, debe ser subdividido, pero recordando que pasó los anteriores (es decir, tomando sus bordes como segmentos rectilíneos). Podría parecer que con una subdivisión así no se logra nada, pero no hay que olvidar que los nuevos bordes que aparecen en el interior del parche no son rectos (precisamente porque el interior del parche no es plano), con lo cual serán fragmentados más tarde. En el apartado 4.1, que presenta los resultados de Triangle, se puede ver cómo actúa el programa sobre un triángulo de Gregory que tiene lados perfectamente rectos a pesar de ser abombado.

Es importante notar que esta solución adoptada para los triángulos de Gregory tiene una validez general —de hecho, es conceptualmente muy parecida a la descrita por Clay y Moreton para los cuadrados de Bézier, de la que está inspirada. Por supuesto, lo que sí varía de un tipo de parches a otro son las pruebas realizadas, y la topología de la subdivisión, pero no la esencia de la solución al problema del *cracking*, que se presenta sea cual sea la familia de parches considerada.

3.3.2.2. ESTRUCTURAS DE DATOS

El listado D.1, que aquí se reproduce parcialmente por comodidad como 3.1, contiene un resumen de los tipos y variables globales del programa que afectan más directamente al algoritmo de subdivisión. En particular, interesa observar la definición de las estructuras `Structure_Sommet` y `Structure_Triangle`, que son las que permiten realizar lo descrito en el apartado anterior.

```
Type Vecteur = Array [1..3] of Real; {-∞ < x,y,z < +∞}
     Param   = Array [0..2] of Real; {0 ≤ u,v,w ≤ 1}

     Sommet = ^Structure_sommet;
     Structure_sommet = Record
                          Precedent,          {Lista de vértices    }
                          Suivant:   Sommet;  {doblemente encadenada}
                          Place:     Int;     {Número en la lista}
                          Coord,              {x,y,z}
                          Normale:   Vecteur;
                          Norm_OK:   Bool;
                          Bary:      Param;   {u,v,w}
                          Scalaire:  Real;    {Coord.Normale}
                        End;

     Triangle = ^Structure_triangle;
     Structure_triangle = Record
                            Suivant:    Triangle;
                            Point:      Array [1..3] of Sommet;
                            Tordu,
                            Silhouette,
                            Frontiere:  Array [0..2] of Bool;
                            Compteur_S,
                            Compteur_F: Array [0..2] of Int;
                          End;

Var Pile_des_Triangles: Triangle;
    Liste_des_Sommets:  Sommet;

    P0, P1, P2, P01, P02, P11, P12, P21, P22,
    P211v, P211w, P121w, P121u, P112u, P112v: Vecteur;

    Tolerance_Rectitude,
    Tolerance_Platitude,
    Supplement_Silhouette,
    Max_Frontiere:         Int;  {valores pedidos por el usuario}
```

listado 3.1 - Principales tipos y variables globales de Triangle.

De los vértices interesa, sobre todo, el hecho de que se almacenan tanto sus coordenadas espaciales en `Coord` (x, y, z) como las paramétricas (o baricéntricas) en `Bary` (u, v, w). Y

de los triángulos, el que, además de guardar punteros a sus tres esquinas en `Point`, se tiene información de cada uno de sus bordes en `Tordu`, `Silhouette`, `Frontiere`, `Compteur_S` y `Compteur_F`, cuyo significado se explica más adelante.

Por otra parte, cabe también señalar la manera de la que se crean y gestionan estas dos estructuras, de las que una es conceptualmente una pila (a cuyo elemento superior apunta siempre `Pile_des_Triangles`), y la otra, una lista doblemente encadenada (cuyo primer elemento es `Liste_des_Sommets`). Al término de la fase de diálogo con el usuario, toma control el procedimiento `Metamorphose` que, tras llamar a `Determine` para averiguar el número de parches que componen la superficie, hace lo siguiente para cada uno de ellos: primero lee, mediante `Lit_Patch`, las coordenadas espaciales de sus quince puntos de control, y las pasa al referencial ligado a la cámara, llamando a `Passage_au_Repere_de_l_Oeil`; los resultados quedan almacenados en `P0`, `P1`, ..., `P112v`. Luego, `Initialisation_des_Donnees` constituye la pila de los triángulos con el parche inicial recién leído, y la lista de vértices con sus tres esquinas, ordenadas[12] según su situación espacial. Entonces, y mientras haya triángulos empilados, saca el primero de ellos y lo somete a una serie de pruebas: si las pasa todas, lo convierte en faceta con una llamada a `Transfert_Triangle_Facette`; si no, `Subdivise` lo fragmenta en cuatro nuevos parches, que son empilados; y en cualquiera de los dos casos, vuelve a sacar el primer triángulo de la pila y repite el proceso. Los vértices que aparecen en las distintas subdivisiones son a su vez añadidos, en el lugar que corresponda, a la lista de vértices. Esta lista no hace sino crecer, a diferencia de la pila de triángulos, que sube y baja hasta que eventualmente se vacía por completo, momento en el que `Metamorphose` escribe en los ficheros de resultados los correspondientes al parche recién facetizado (procedimiento `Ecrit_Resultats`), lee el siguiente parche de la superficie original, y todo vuelve a empezar. Y una vez facetizada la superficie completa, se salvan los parámetros de observación que eligió el usuario, y que han provocado esta facetización concreta: no se debe olvidar que la subdivisión se adapta al punto de vista.

3.3.2.3. ESTRATEGIA DE SUBDIVISIÓN Y PRUEBAS IMPLEMENTADAS

Aprovechando la experiencia descrita en [CLAY-88], se adoptaron como posibles razones de subdivisión de un triángulo de Gregory las cuatro que en ese artículo se mencionan. Primero se somete al parche considerado a tres pruebas que determinan si sus bordes son curvos, si pertenecen a la silueta del objeto en la imagen, o si son parte de la frontera original. Luego, si no se da ninguno de los tres casos anteriores para ninguno de los tres bordes del parche, se verifica si su interior es suficientemente plano antes de convertirlo en faceta.

Antes de describir en detalle la implementación de cada una de estas pruebas, es crucial aclarar que no se debe subdividir un parche sin disponer de información completa sobre

[12] Ver anejo D para una descripción de esta ordenación.

cada uno de sus bordes: es decir, que aunque el primer borde sea claramente curvo, por ejemplo, hay que esperar a saber cómo son los demás para tener siempre actualizados los valores de sus respectivos contadores y variables lógicas, porque estos valores constituyen la *memoria* del parche, que transmitirá como *herencia* a sus hijos al ser subdividido. En concreto, esta transmisión hereditaria funciona así: de los cuatro hijos (cf. fig. 3.3), tres son *legítimos*, y tienen dos bordes antiguos y uno nuevo, mientras que el restante es *ilegítimo* —además de *invertido*—, y tiene tres bordes nuevos. Los bordes antiguos ceden a cada uno de sus dos fragmentos los valores de sus variables lógicas y contadores: `Tordu`, `Silhouette` y `Frontiere`, indican respectivamente si el borde en cuestión debe pasar cada uno de los tres primeros *tests* o si ya ha sido eximido de ellos; `Compteur_S` registra el número de veces que el borde ha sido subdividido por cruzar una silueta, una vez fue dado por recto; y `Compteur_F` el número total de veces que un borde ha sido subdividido, por cualquier razón. La memoria de los bordes nuevos, por su parte, es inicializada como sigue: evidentemente, no pertenecen a la frontera original (`Frontiere:=False`), pero sí tendrán que sufrir los *tests* de rectitud y silueta (`Tordu`, `Silhouette:=True`); y los contadores se ajustan, para no provocar un número excesivo de subdivisiones, al máximo valor de los tres del parche padre.

En el artículo de Clay y Moreton, se considera que la pertenencia de un borde a la frontera del parche original justifica un número elevado de subdivisiones, y puede que esto sea cierto en el caso de parches aislados o superficies abiertas. Pero en el marco de este Proyecto, en donde se trataba finalmente de tratar superficies grandes (y por lo general, cerradas) parche a parche, por limitaciones de memoria, parecía más recomendable la solución adoptada que la descrita en [CLAY-88], que consiste en aplicar a los bordes de la frontera el mismo tratamiento que a los de la silueta (i.e. subdividirlos un número suplementario, y no total, de veces). De todas maneras, el apartado 4.1 recoge los resultados de una y otra táctica a un parche aislado, por lo que el lector podrá apreciar cómo, incluso en un caso desfavorable, la solución escogida demuestra ser eficaz.

A continuación se presenta la implementación de cada una de las pruebas a las que se somete todo parche, que están aisladas en cuatro funciones *booleanas* (`Cotes_Tordus`, `Cotes_Silhouette`, `Cotes_Frontiere` y `Triangle_Gauche`), como se puede ver en el anejo D. Por ello, son muy fácilmente sustituibles (siempre que se disponga del código fuente de Triangle) por otras que midan de distinta manera la rectitud de los bordes, por ejemplo. Para seguirlas, es esencial tener presente el mecanismo *hereditario* descrito más arriba, y conveniente ayudarse de la figura 3.6.

1. **bordes curvos**: si un borde *b* estaba previamente marcado como *torcido* (`Tordu[b]=True`), se mide la distancia en pantalla entre *Exacto* (punto paramétrico medio, sobre el segmento curvilíneo) y *Aprox* (punto medio del segmento rectilíneo). Si es mayor de `Tolerance_Rectitude` píxeles, se deja a *b* como *torcido*, y la función `Cotes_Tordus` devolverá `True`, lo que causará una subdivisión del parche en la que *b* será partido por *Exacto*; si es menor (en ese caso se hace `Tordu[b]:=False`), o si *b* no debía sufrir este *test*, porque no estaba marcado como *torcido*, toda eventual subdivisión del parche (causada por otro motivo) partiría a *b* por *Aprox*,

conservando así el segmento rectilíneo para evitar la aparición de grietas en la superficie;

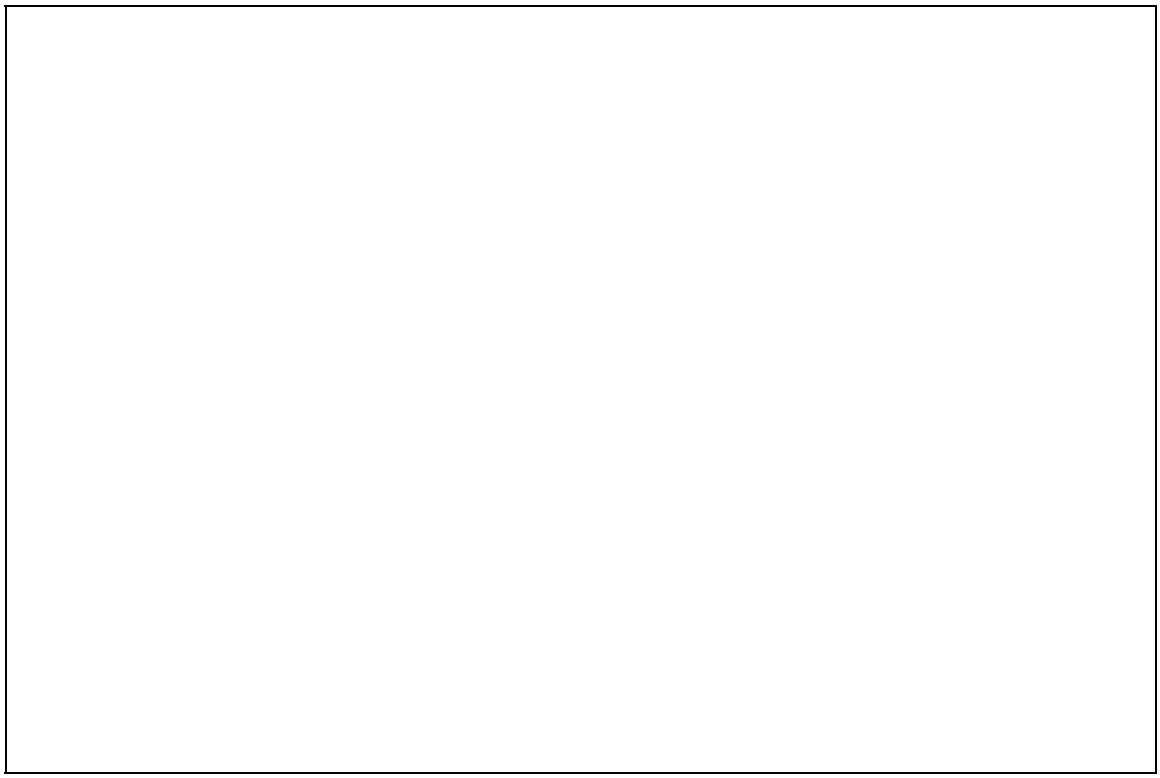

figura 3.6 - Posibles razones de subdivisión de un triángulo de Gregory:
a) bordes curvos; b) bordes silueta;
c) bordes frontera; d) interior alabeado.

2. **bordes silueta**: si un borde *b* estaba previamente marcado como *silueta* (`Silhouette[b]=True`), se comparan los signos de `Scalaire` de los dos puntos extremos de *b*. Si son distintos, y si *b* no ha sido todavía fragmentado tantas veces como pidió el usuario (suplementarias, por esta razón, i.e. `Compteur_S[b] < Supplement_Silhouette`), se deja a *b* como *silueta*, y la función `Cotes_Silhouette` devolverá `True`, lo que causará una subdivisión del parche en la que *b* será partido por *Exacto*; si no se dan ambas condiciones a la vez (en ese caso se hace `Silhouette[b]:=False`), o si *b* no debía sufrir este *test*, porque no estaba marcado como *silueta*, toda eventual subdivisión del parche partiría a *b* por *Aprox*;
3. **bordes frontera**: si un borde *b* estaba previamente marcado como *frontera* (`Frontiere[b]=True`), y si *b* no ha sido todavía fragmentado tantas veces como pidió el usuario (en total, y por cualquier razón, es decir: `Compteur_F[b] < Max_Frontiere`), se deja a *b* como *frontera*, y la función `Cotes_Frontiere` devolverá `True`, lo que causará una

subdivisión del parche en la que *b* será partido por *Exacto*; si ya se ha alcanzado el número de subdivisiones pedido (en ese caso se hace `Frontiere[b]:=False`), o si *b* no debía sufrir este *test*, porque no estaba marcado como *frontera*, toda eventual subdivisión del parche partiría a *b* por *Aprox*;

4. **interior alabeado**: si `Tolerance_Platitude > 0`, se mide la distancia entre el baricentro del parche y el de la faceta plana, en relación al tamaño del parche. Si es mayor que `Tolerance_Platitude` %, la función `Triangle_Gauche` devolverá `True`, y se producirá una subdivisión del parche en la que todos sus bordes serán asimilados a segmentos rectilíneos; si es menor, o si el usuario prohibió las subdivisiones por esta razón con `Tolerance_Platitude = 0`, `Triangle_Gauche` devolverá `False`, y el parche será finalmente aproximado por una faceta plana.

Capítulo 4. RESULTADOS GRÁFICOS Y CONCLUSIONES

Así como en los dos capítulos anteriores se daba la "ficha técnica" de los dos programas desarrollados, en éste se presentan los resultados gráficos obtenidos por cada uno de ellos individualmente, y por el tándem que forman: no se debe olvidar que Triangle y Ripolin se concibieron como eslabones de una cadena. Por eso, aunque hasta ahora se haya preferido seguir un orden de exposición determinado por la cronología del Proyecto —las modificaciones de Ripolin tuvieron lugar antes que la escritura de Triangle—, aquí ha parecido más lógico exponer antes los productos del algoritmo de subdivisión, puesto que son luego procesados por el de visualización. Estos resultados gráficos son, en el caso de Triangle, trazados del *wire-frame* del conjunto de facetas planas de salida, volcados a impresora; y en el de Ripolin, imágenes de 1024 x 1024 píxeles en color real, fotografiadas directamente de un monitor de alta resolución.

4.1. Subdivisiones producidas por Triangle

Para comprobar el funcionamiento del algoritmo de subdivisión en sí, se diseñaron dos parches de Gregory: a continuación se listan sus grafos de control, y se muestran sus formas mediante los resultados de subdivisiones *sistemáticas* de cuarto orden. Conviene recordar que la subdivisión sistemática es la más sencilla posible, puesto que la estrategia de fragmentación es inexistente: se toma el triángulo alabeado original y se fragmenta en cuatro subparches sin someterlo a ninguna prueba; con cada uno de ellos se repite el proceso, n veces en total (n es el orden de la subdivisión), para obtener 4^n=256 subparches que son finalmente asimilados a facetas planas. Ahora bien, como se espera demostrar con los resultados de Triangle sobre estos parches de prueba, la subdivisión sistemática produce un número excesivo de facetas, lo que resulta caro en espacio de almacenamiento y tiempo de proceso posterior. Por eso es más deseable una estrategia *adaptativa,* que sea capaz de generar muchas facetas allí donde sea necesario en la imagen (según el criterio subjetivo del espectador humano) y pocas donde no lo sea, para lograr en definitiva una reducción importante del número total.

Al final de este apartado se muestra, además, el resultado producido por el programa para una superficie real de 3000 parches: un busto de Victor Hugo digitalizado mediante el sistema *3D-Vidéolaser* del Laboratorio IMA.

El primero de los dos triángulos cuárticos de Gregory de prueba, llamado *Chaise* (silla), sirvió para poner a punto las pruebas realizadas sobre los bordes, y es como una sábana triangular elástica, sobre cuyo centro se hubiera puesto un peso, y de cuyos tres bordes se hubiera tirado hacia arriba (ver listado y figura 4.1).

```
Coordonnées des points de contrôle du patch "CHAISE":

              X       Y       Z

  P0       -100       0       0
  P01       -50      29     100
  P02         0      58     100
  P1         50      87       0
  P11        50      29     100
  P12        50     -29     100
  P2         50     -87       0
  P21         0     -58     100
  P22       -50     -29     100
  P211v     -43       0    -100
  P211w     -43       0    -100
  P121w      11      34    -100
  P121u      11      34    -100
  P112u      11      -5    -100
  P112v      11      -5    -100
```

listado 4.1 - Coordenadas de los puntos de control de *Chaise*.

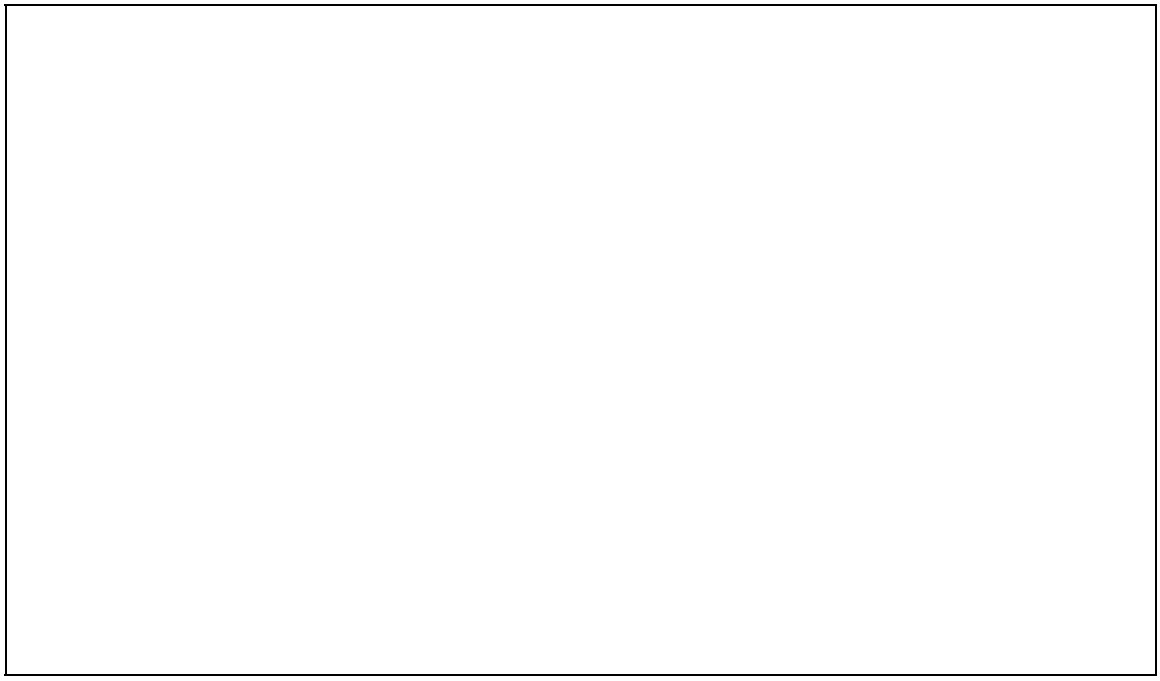

figura 4.1 - Subdivisión sistemática de *Chaise* en 4^4=256 facetas.

El segundo parche, *Bosse* (joroba), es una colina cuyos bordes forman un triángulo equilátero plano (cf. listado y figura 4.2), por lo que fue utilizado para observar el comportamiento del mecanismo ideado para las subdivisiones debidas al interior de un parche de bordes correctos.

```
Coordonnées des points de contrôle du patch "BOSSE":

              X       Y       Z

  P0       -100       0       0
  P01       -50      29       0
  P02         0      58       0
  P1         50      87       0
  P11        50      29       0
  P12        50     -29       0
  P2         50     -87       0
  P21         0     -58       0
  P22       -50     -29       0
  P211v     -43       0     100
  P211w     -43       0     100
  P121w      11      34     100
  P121u      11      34     100
  P112u      11      -5     100
  P112v      11      -5     100
```

listado 4.2 - Coordenadas de los puntos de control de *Bosse*.

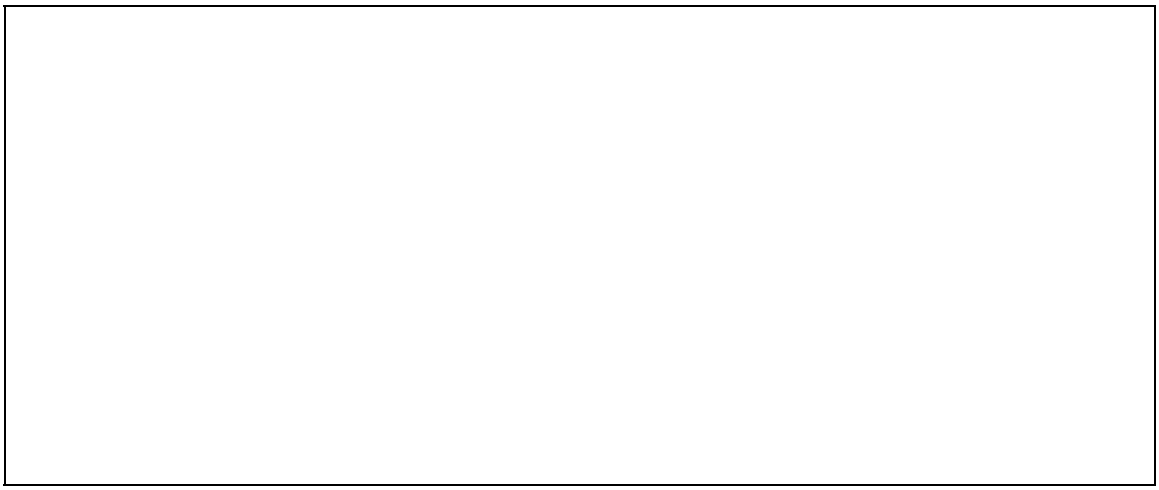

figura 4.2 - Subdivisión sistemática de *Bosse* en 4^4=256 facetas.

En cada una de las figuras que siguen, se dan el número total de facetas y los valores de los límites de tolerancia que produjeron las subdivisiones correspondientes. Estos últimos deben ser interpretados así:

R = tolerancia de Rectitud de los bordes (en píxeles de una imagen de 1024 x 1024);

S_S = número de subdivisiones **Suplementarias** para bordes Silueta;

F_S = id. id. **Suplementarias** para bordes Frontera (según sugiere [CLAY-88]);

F_T = id. id. **Total** para bordes Frontera;

A = tolerancia de Alabeo del interior (% sobre el tamaño del parche).

NB: X = 0 inhibe las subdivisiones por la razón correspondiente.

Además se detalla, en cada caso, el número de subdivisiones que tuvieron lugar por cada una de las cuatro razones posibles. Por ejemplo, para obtener el resultado mostrado en la figura 4.3, se ejecutó Triangle exigiendo (R, S_S, F_S, A) = (20, 0, 0, 0), lo que causó únicamente 47 fragmentaciones por bordes excesivamente torcidos: el resto de razones no se tomaron en cuenta, al estar sus límites fijados a 0. Obsérvese cómo, en los dos bordes más cercanos a la cámara, se produjeron más divisiones que en el de detrás, precisamente por el carácter adaptativo del algoritmo.

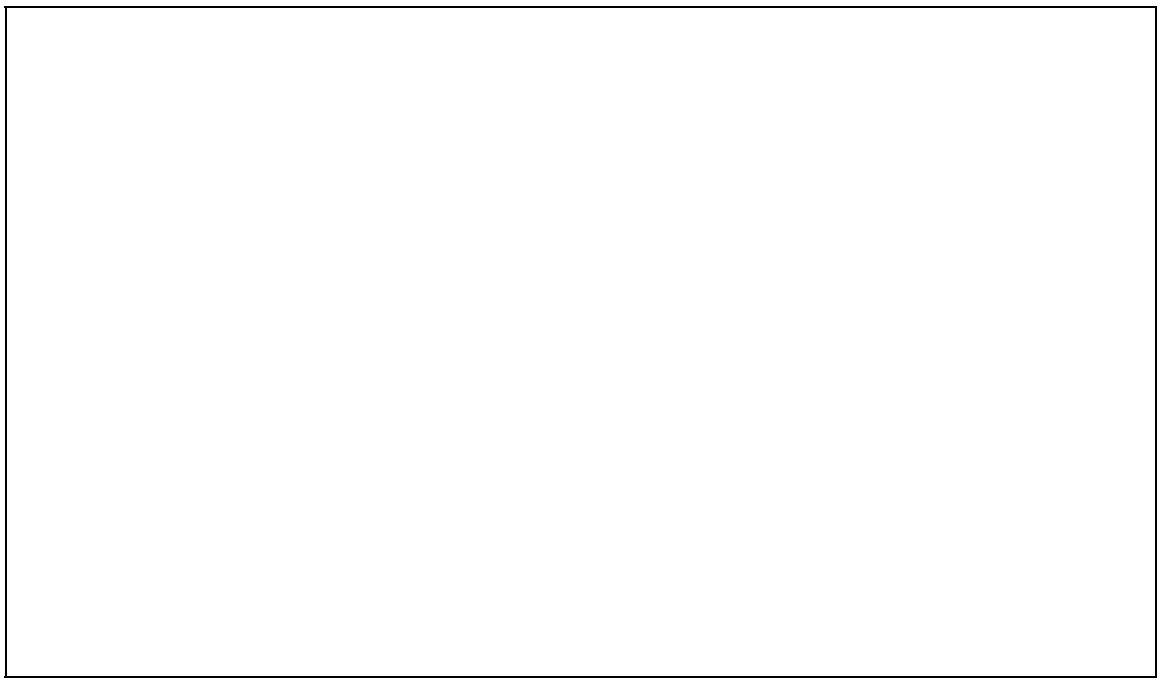

figura 4.3 - Subdivisión adaptativa de *Chaise* en 142 facetas.
Límites: (R, S_S, F_S, A) = (20, 0, 0, 0); motivos: (47, 0, 0, 0).

En la figura 4.4 se introducen las subdivisiones en la silueta del objeto: una vez han sido dados por buenos como rectos, hay 34 bordes que son fragmentados por cruzar la silueta, lo que hace que aparezcan otros 34 x 3 = 102 nuevos parches, que son facetizados inmediatamente por estar inhibidas las otras razones de subdivisión. Compárese este resultado con el anterior para ver lo importantes que son las siluetas en la impresión de calidad producida sobre el ojo humano, y lo cara que resulta la mejora: casi un 72% más de facetas.

figura 4.4 - Subdivisión adaptativa de *Chaise* en 244 facetas.
Límites: (R, S_S, F_S, A) = (20, +1, 0, 0); motivos: (47, 34, 0, 0).

En las figuras 4.5 a 4.10, que han de ser consideradas por pares, se compara la técnica aplicada por Clay y Moreton a la frontera del parche de partida con la utilizada en este Proyecto. En el primer ejemplo (primer par de figuras), se ilustran las consecuencias de la elección "F_S vs. F_T" cuando las subdivisiones suplementarias para los bordes silueta están prohibidas: en el primer caso (fig. 4.5), se originan 39 fragmentaciones, de las que una inmensa mayoría son innecesarias, incluso para un parche sin vecinos como *Chaise*; en el segundo (fig. 4.6), tan sólo 6 bordes pertenecientes a la frontera original no han sido divididos ya 4 veces cuando le llega el turno a este *test*, y son fragmentados por ello (lo que hace aparecer otros 10 bordes insuficientemente rectos). Obsérvese que los resultados obtenidos en los bordes de partida son indistinguibles.

figura 4.5 - Subdivisión adaptativa de *Chaise* en 289 facetas.
Límites: (R, S_S, F_S, A) = (20, 0, +1, 0); motivos: (47, 0, 39, 0).

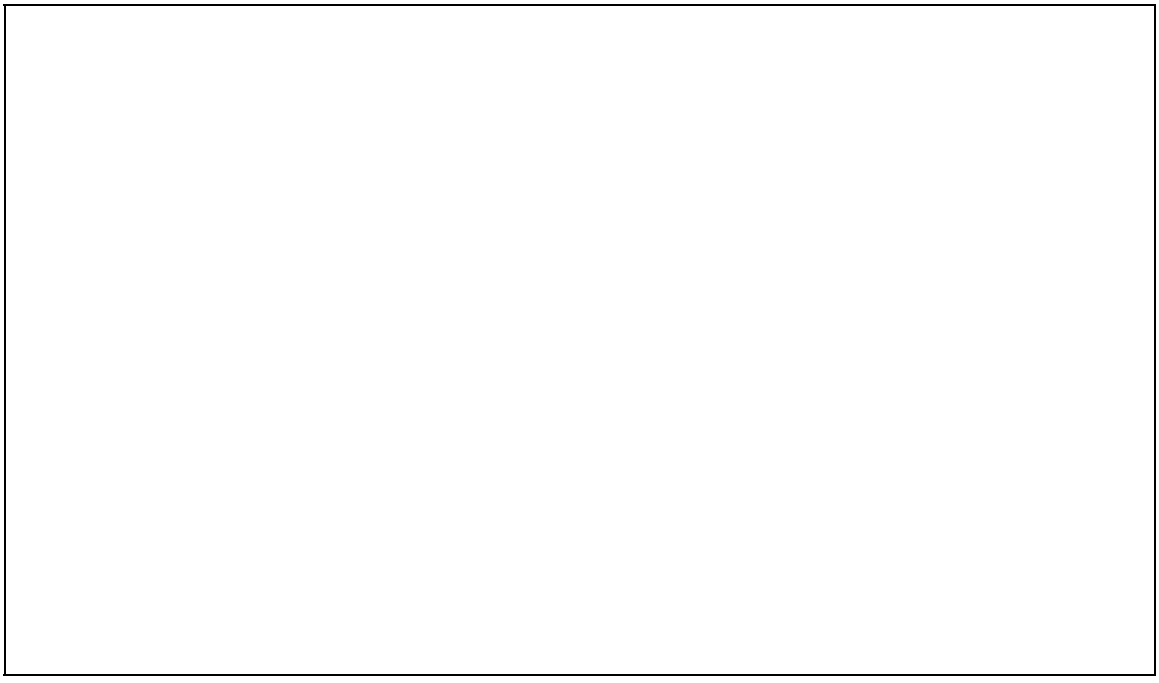

figura 4.6 - Subdivisión adaptativa de *Chaise* en 190 facetas.
Límites: (R, S_S, F_T, A) = (20, 0, 4, 0); motivos: (57, 0, 6, 0).

En el segundo ejemplo (figuras 4.7 y 4.8), se introducen además las fragmentaciones en los bordes silueta, que tienen lugar antes que las de la frontera (también a diferencia de lo propuesto por Clay y Moreton), para hacer más patente la superioridad del método adoptado: 310 vs. 463 facetas para un resultado muy similar.

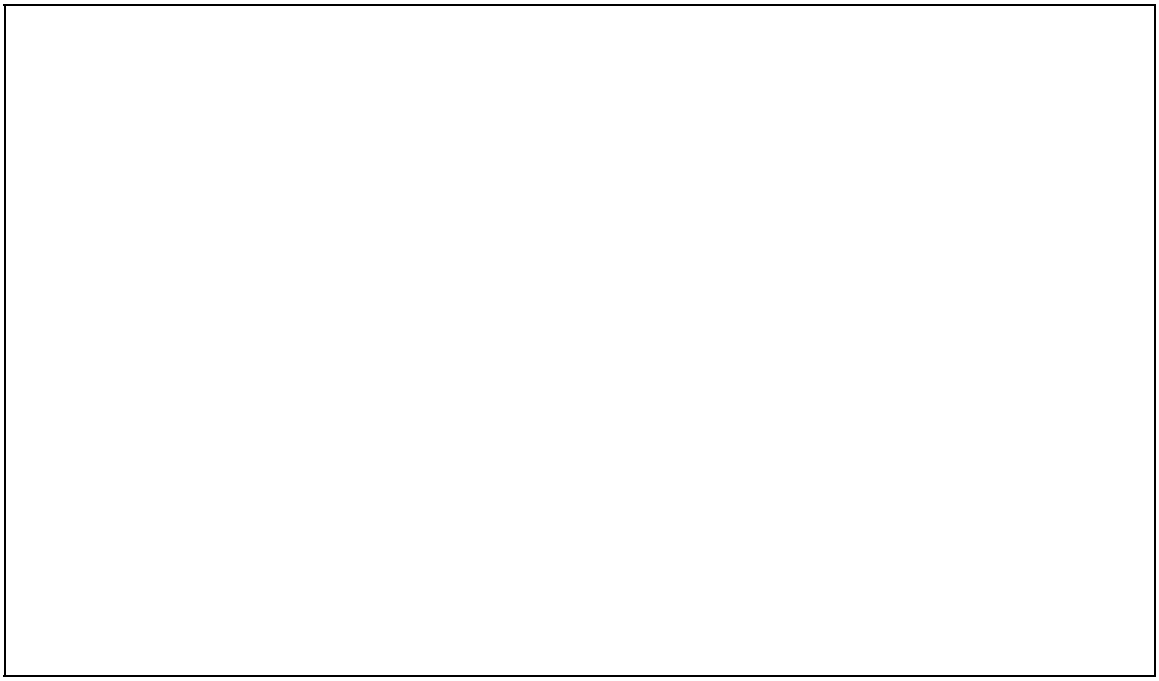

figura 4.7 - Subdivisión adaptativa de *Chaise* en 463 facetas.
Límites: (R, S_S, F_S, A) = (20, +1, +1, 0); motivos: (57, 51, 46, 0).

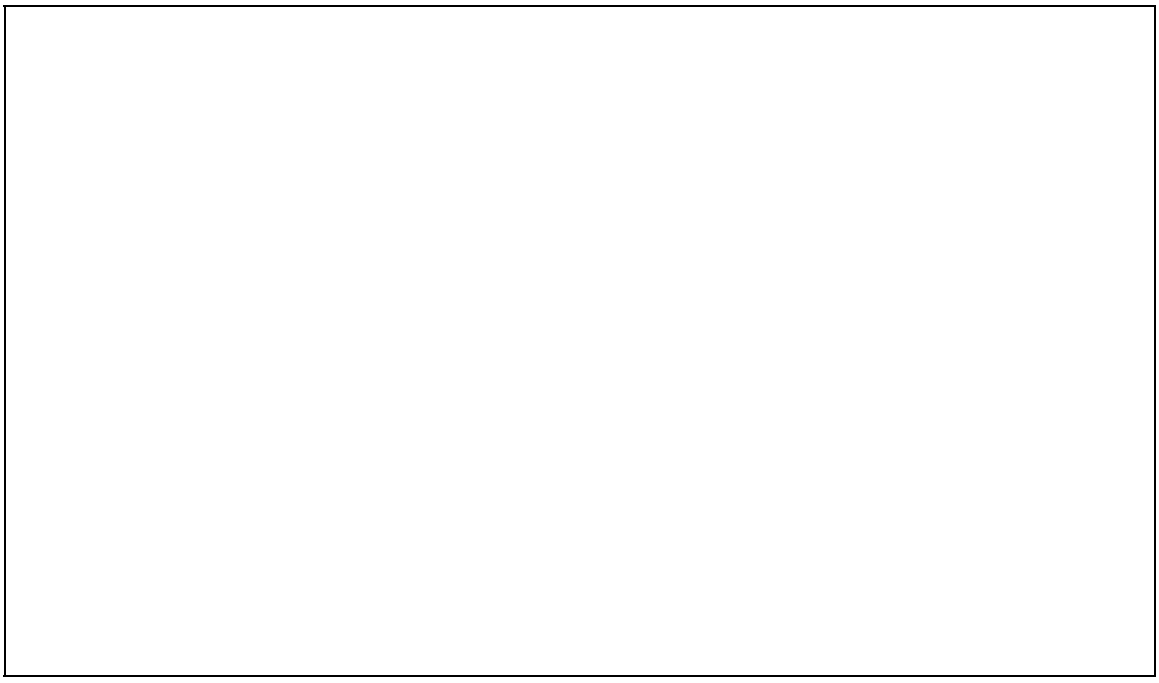

figura 4.8 - Subdivisión adaptativa de *Chaise* en 310 facetas.
Límites: (R, S_S, F_T, A) = (20, +1, 4, 0); motivos: (57, 42, 4, 0).

En el tercer ejemplo (figuras 4.9 y 4.10), se disminuye la tolerancia de rectitud para los bordes, con lo que se aumenta la probabilidad de que un borde haya sido subdividido ya varias veces por ser torcido antes que por ser frontera: de hecho, se puede ver que en el segundo caso no se produce ni una sola fragmentación de las 84 que ocurren en el primero.

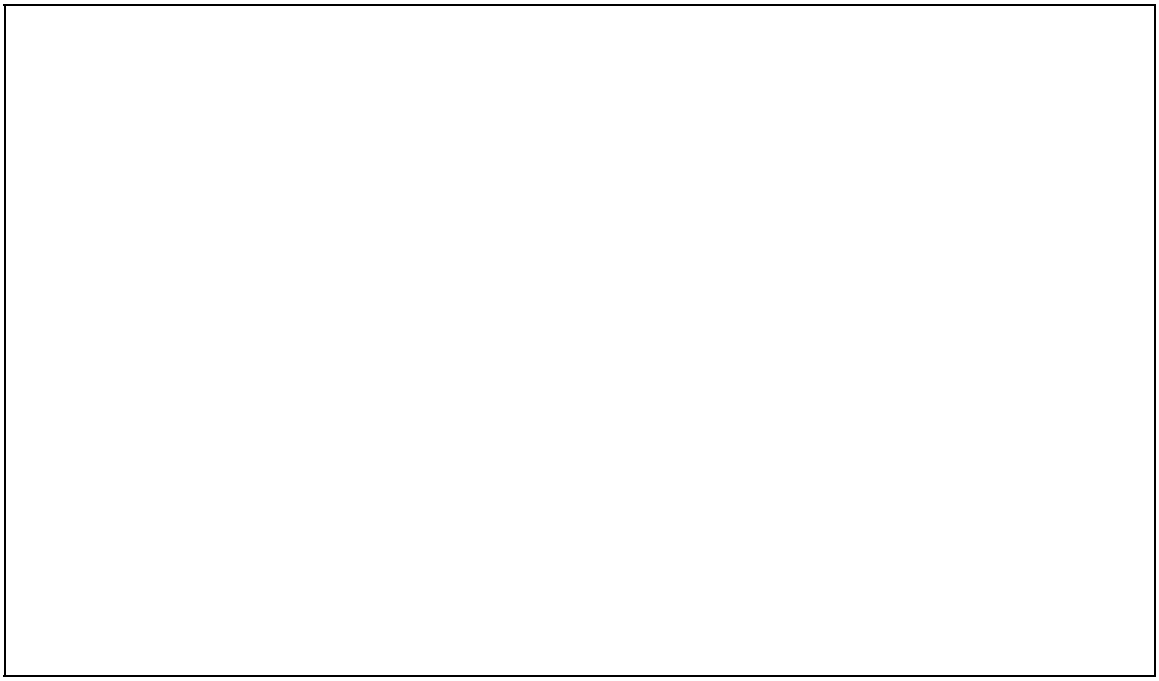

figura 4.9 - Subdivisión adaptativa de *Chaise* en 970 facetas.
Límites: (R, S_S, F_S, A) = (5, 0, +1, 0); motivos: (239, 0, 84, 0).

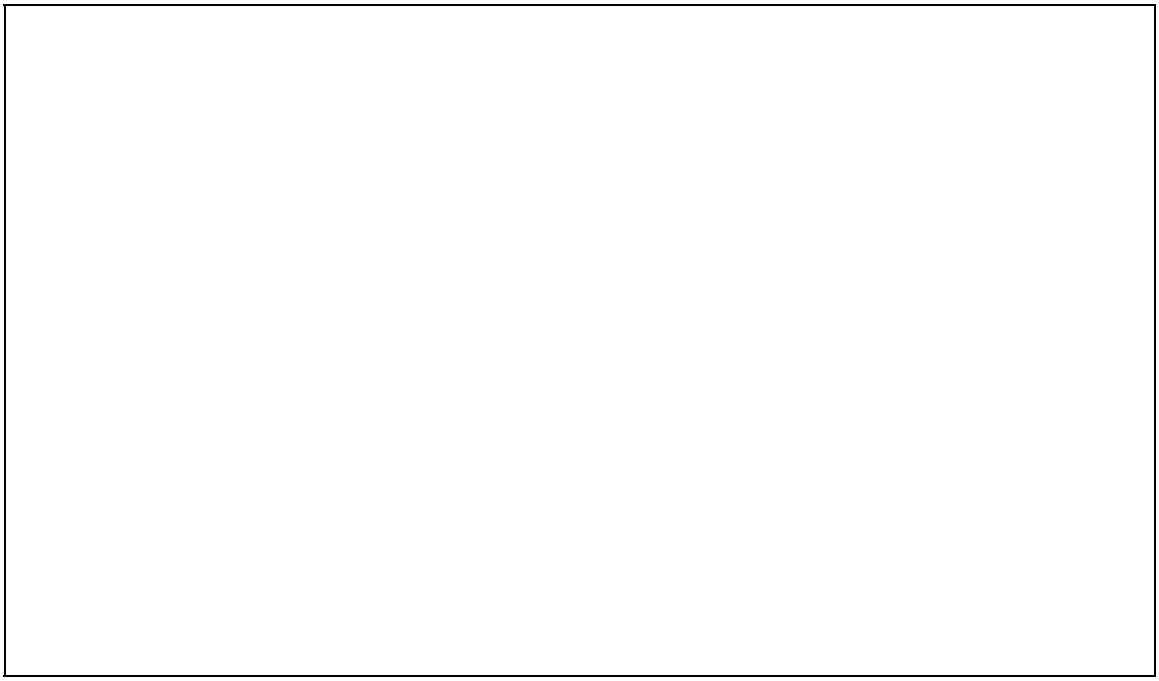

figura 4.10 - Subdivisión adaptativa de *Chaise* en 718 facetas.
Límites: (R, S_S, F_T, A) = (5, 0, 4, 0); motivos: (239, 0, 0, 0).

Para lograr un resultado visualmente parecido al de la figura 4.10, en lo que se refiere a la suavidad de la superficie, habría que realizar una subdivisión sistemática de quinto orden como la de la figura 4.11, que produce un número de facetas casi 50% mayor.

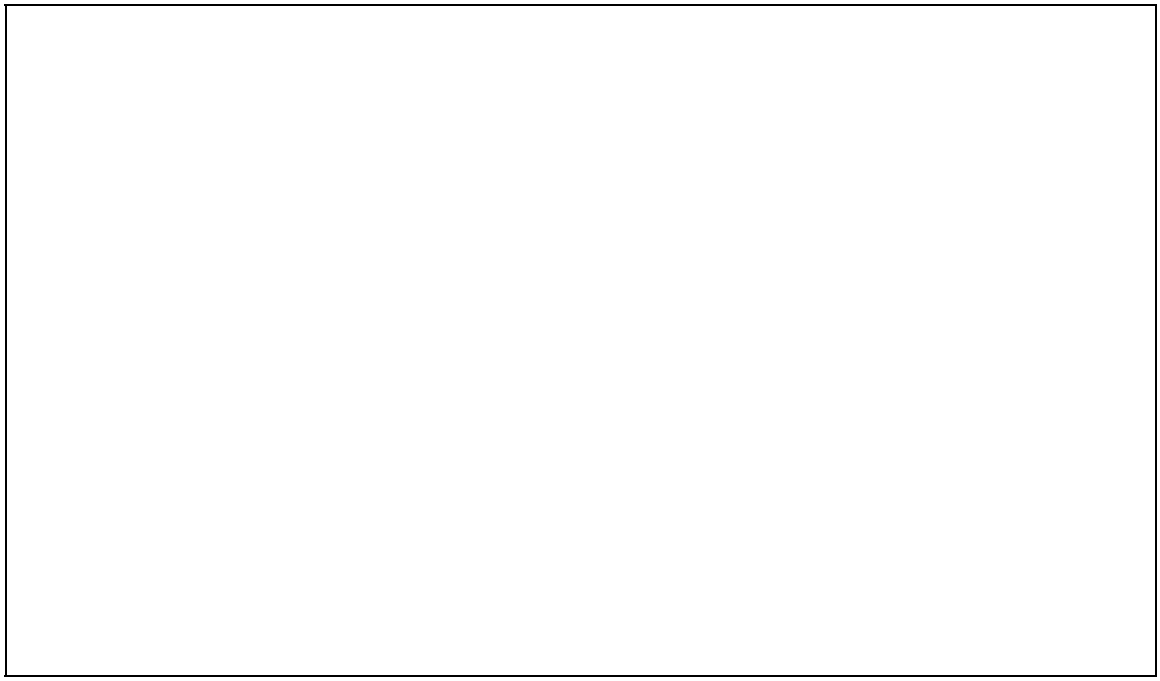

figura 4.11 - Subdivisión sistemática de *Chaise* en 4^5=1024 facetas.

Antes de presentar el comportamiento de Triangle sobre una superficie real compleja, queda por describir cómo funciona el mecanismo de las subdivisiones debidas al interior del parche. Según se explicó en el capítulo dedicado a este programa, para evitar la aparición de grietas en la superficie facetizada existen dos soluciones: la primera consiste en almacenar todo el árbol lógico de subdivisión para disponer de la información completa de vecindad de cada parche antes de fragmentarlo; y la segunda, en basarse únicamente en las características individuales de cada parche para tomar las decisiones.

Además de ser más económica en espacio de almacenamiento y tiempo de cálculo, esta segunda opción era la única viable en el marco de este Proyecto, puesto que se deseaba que, en definitiva, fuera el propio Ripolin quien leyera los parches del modelo uno a uno y los facetizara antes de visualizarlos. Pero presenta un inconveniente: para que un borde juzgado recto en un parche pueda ser fragmentado en el vecino (por culpa de los otros bordes o del interior del parche) sin que aparezca una grieta entre esos dos parches, debe introducirse información, en la descripción de cada borde, que permita saber, basándose únicamente en las características propias del borde, si debe ser considerado como curva o como segmento rectilíneo en el curso de una eventual fragmentación.

Un parche como *Bosse*, cuyos tres bordes son perfectamente rectos, pero cuyo interior es inaceptablemente abombado, es un reto difícil de superar, porque los tres bordes originales deben ser dados por buenos desde el principio, y divididos desde el principio por

puntos coplanarios con las esquinas, que no parecen ayudar a "diluir" el abombamiento. Sin embargo, no hay que olvidar que los bordes que aparecen en el interior por efecto de la subdivisión debida a ese abombamiento sí son curvos, y son marcados como tales al nacer, por lo que a la primera (y única) subdivisión por "interior insuficientemente plano" siguen muchas por "bordes insuficientemente rectos" o "bordes silueta" (cf. fig. 4.12).

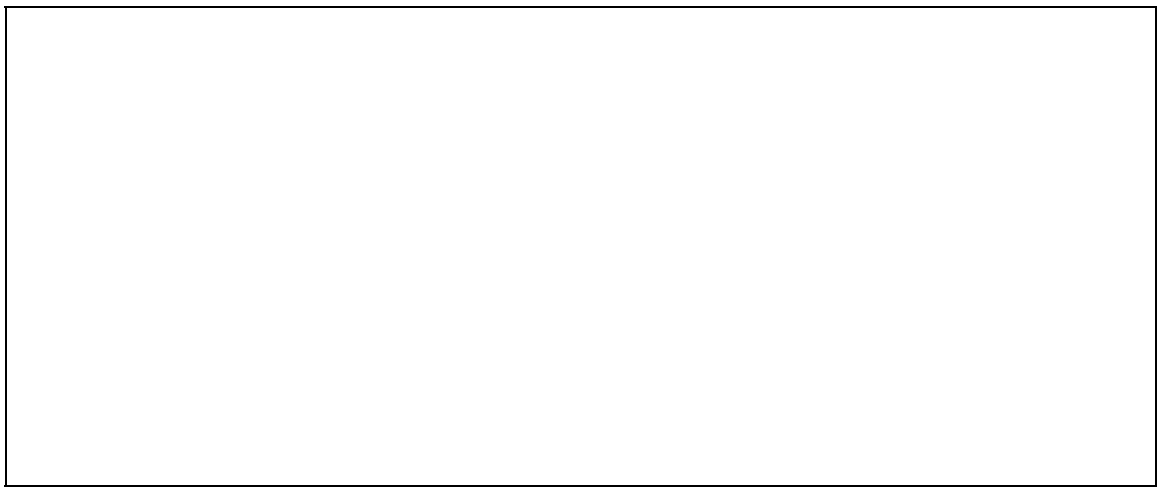

figura 4.12 - Subdivisión adaptativa de *Bosse* en 121 facetas.
Límites: (R, S_S, F_T, A) = (17, +1, 0, 40); motivos: (22, 17, 0, 1).

Una vez analizadas exhaustivamente las características de Triangle con estos parches de prueba, diseñados "a medida", se enfrentó al algoritmo con una superficie real de más de tres mil triángulos de Gregory (*Vic_3K*, modelo de un busto de Victor Hugo muestreado mediante técnicas láser), especificando diversas combinaciones para los límites de tolerancia. De los resultados obtenidos, se presenta a continuación el correspondiente a una sola subdivisión de los bordes silueta, que es el único en el que se puede apreciar cómodamente el trabajo de Triangle, dado que el elevado número de parches de partida procuraba ya de por sí un detalle algo excesivo para las dimensiones de la figura 4.13. Nótese que sólo se han trazado las facetas vistas por la cámara para hacerla inteligible, y que las siluetas sobre las que opera el algoritmo no son sólo los recortes del objeto sobre el fondo, sino también los de partes prominentes del objeto sobre sí mismo (la ceja, el bigote, ...).

En el siguiente apartado se muestran fotografías de imágenes obtenidas suministrando a Ripolin conjuntos de facetas resultantes de distintas subdivisiones de *Vic_3K*.

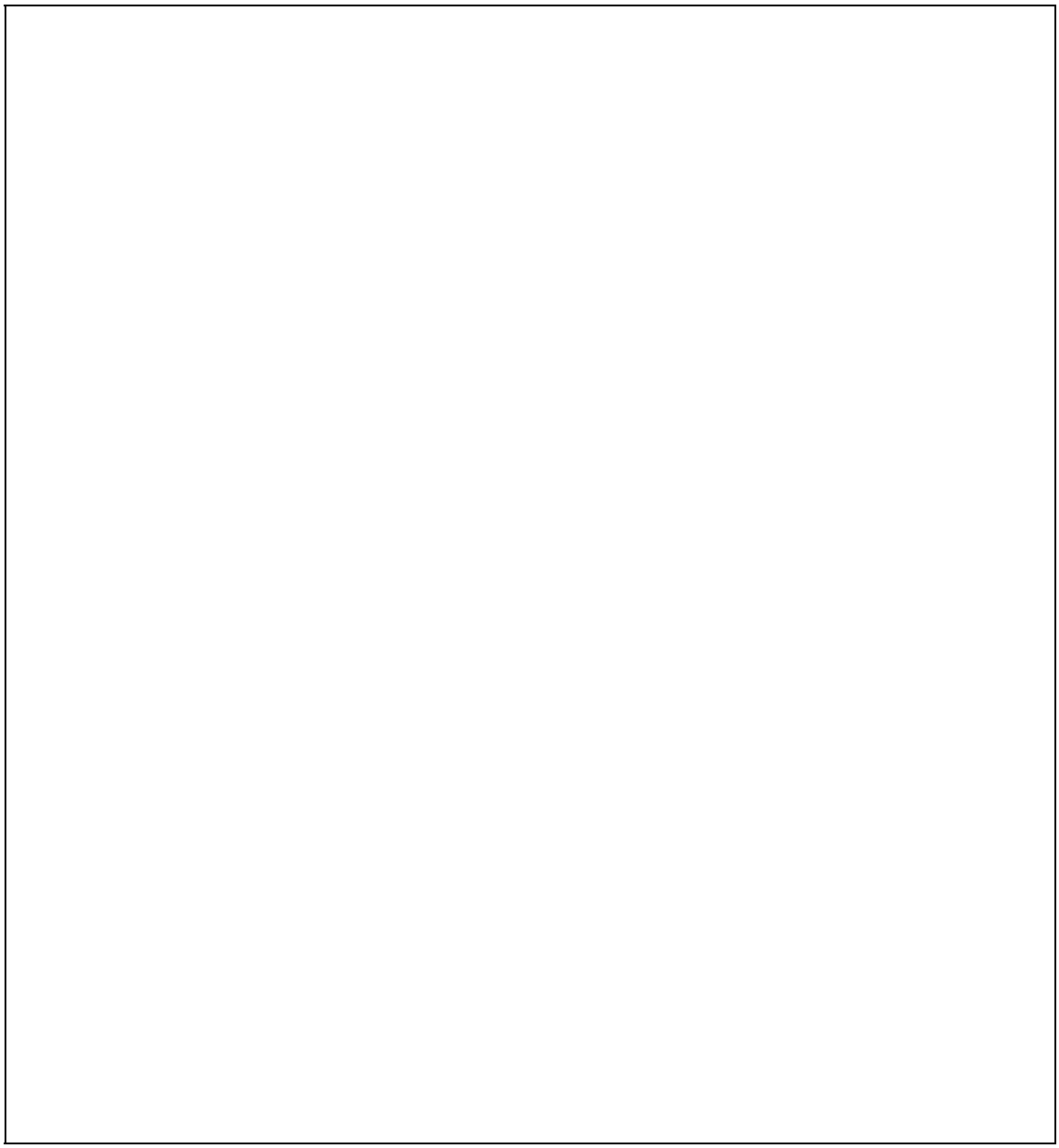

figura 4.13 - Subdivisión adaptativa de *Vic_3K* en 3641 facetas (sólo partes vistas).
Límites: (R, S_S, F_T, A) = (0, +1, 0, 0); motivos: (0, 213, 0, 0)[13].

[13] NB: 213 subdivisiones dan lugar a la aparición de 639 nuevos parches que, sumados a los 3002 que en realidad componen *Vic_3K*, justifican el total de 3641 facetas.

4.2. Imágenes producidas por Ripolin

En este apartado se presentan varias imágenes obtenidas con los programas realizados, a partir de distintos tipos de modelos superficiales. Algunos objetos estaban descritos originalmente por facetas planas (formato ENSAD), y pudieron ser entregados directamente a Ripolin: es el caso del cubo que se muestra a continuación, o de la molécula retratada en las fotografías del capítulo de introducción. Sin embargo, otros estaban recubiertos por triángulos de Gregory, por lo que hubo que facetizarlos con Triangle antes de poder visualizarlos con Ripolin: es el caso de los bustos de Victor Hugo (*Vic_1K* y *Vic_3K*) mostrados más adelante.

La imagen 4.1 ilustra por una parte la diferencia entre los distintos tipos de tratamiento del color ofrecidos por Ripolin (*z-buffer* simple vs. pulidos de Gouraud y de Phong), y por otra la posibilidad de realizar cortes con los planos de *clipping* (en este caso, con el plano frontal, que deja ver el interior del objeto). Obsérvese que sólo en los dos últimos casos, para los que se ha utilizado el mecanismo de interpolación del color de Phong, se ve en la imagen el reflejo especular del foco luminoso sobre la superficie del cubo, definida como "brillante" (*iexp*=30; cf. anejo A).

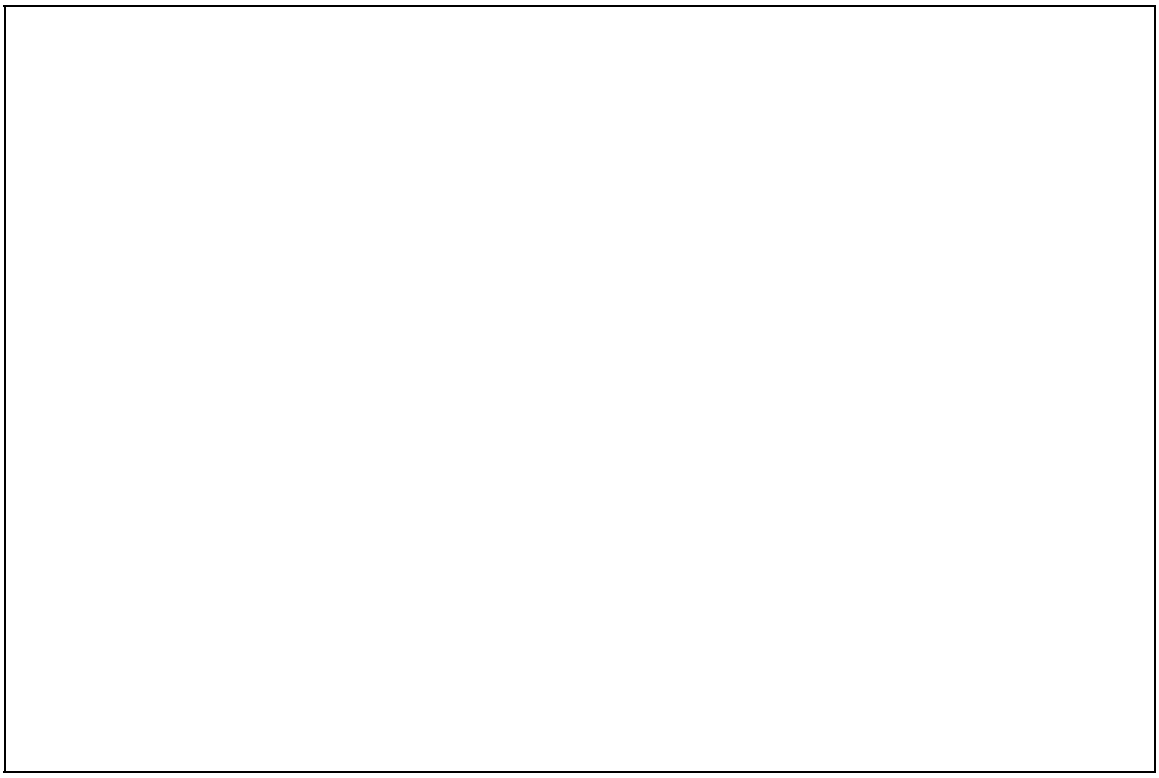

imagen 4.1 - Tratamiento del color y *clipping*:
a) *z-buffer* simple; b) pulido de Gouraud;
c) pulido de Phong; d) Phong + *clipping* frontal.

La imagen 4.2 muestra con mayor detalle el resultado de la 4.1.c.

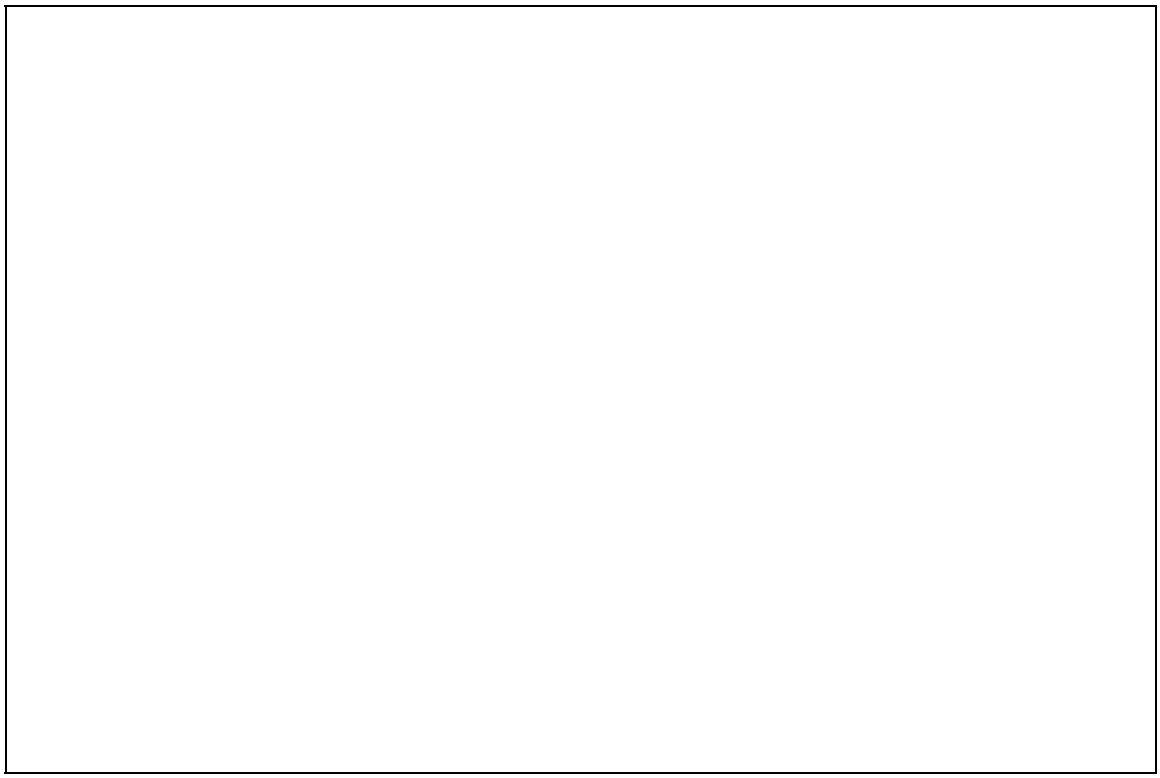

imagen 4.2 - Cubo "brillante" (*iexp*=30) con pulido de Phong.

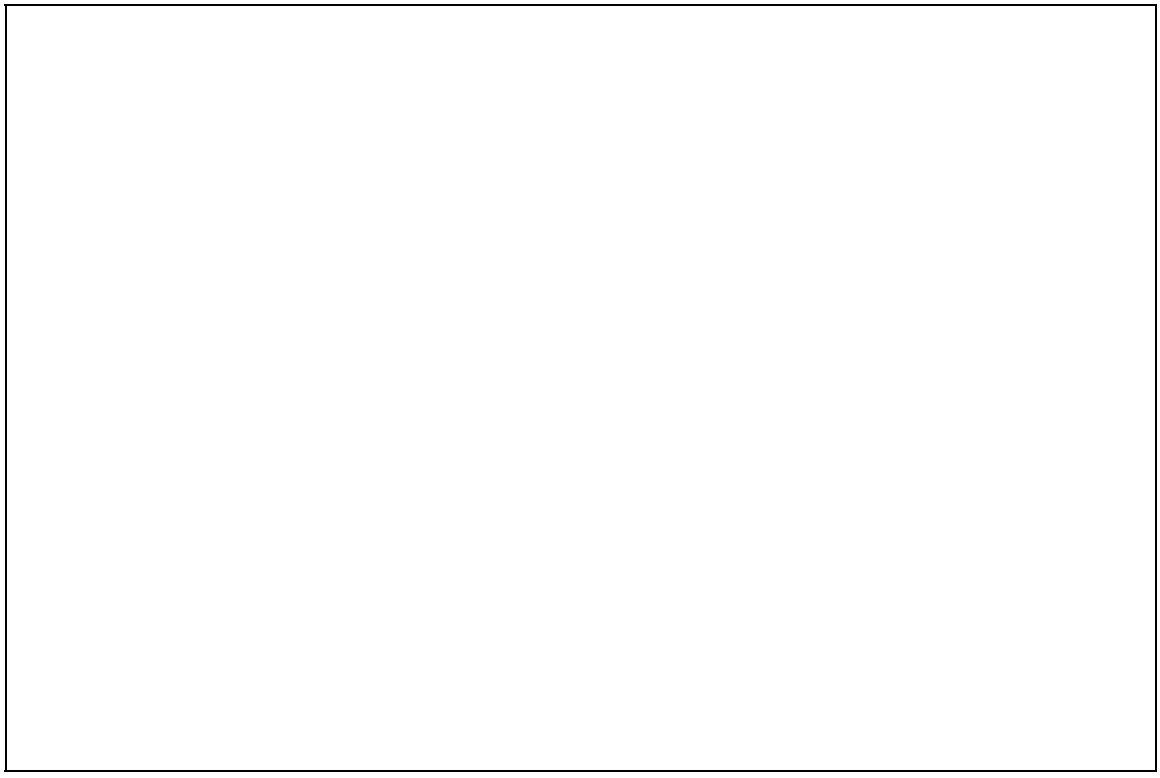

imagen 4.3 - Tratamiento de partes vistas y ocultas mediante la técnica del *z-buffer*.

La imagen 4.3 ilustra el funcionamiento de la técnica del *z-buffer* descrito en el capítulo de introducción. El proceso para obtenerla fue el siguiente: se disponía de dos moléculas, modeladas por facetas planas, cuyas formas debían ser comparadas; se produjo primero una imagen con una de ellas (la amarilla, aunque poco importa el orden), pidiendo a Ripolin que salvara el *z-buffer* final; luego, se completó la imagen partiendo del *z-buffer* salvado, con la otra molécula (la roja), y teniendo cuidado de utilizar los mismos parámetros de cámara. El mecanismo de Catmull solucionó elegantemente el problema de partes vistas y ocultas, dejando que una molécula se imbricara en otra con total naturalidad. Tanto para una molécula como para otra se empleó el pulido de Phong: nótese que con ello se logra suavizar aparentemente el interior de las superficies, pero no sus contornos, puesto que sólo se interpola el color dentro de cada faceta, sin alterar la forma de los objetos ni, por lo tanto, sus siluetas (por este mismo motivo, las intersecciones entre las superficies tampoco son suaves).

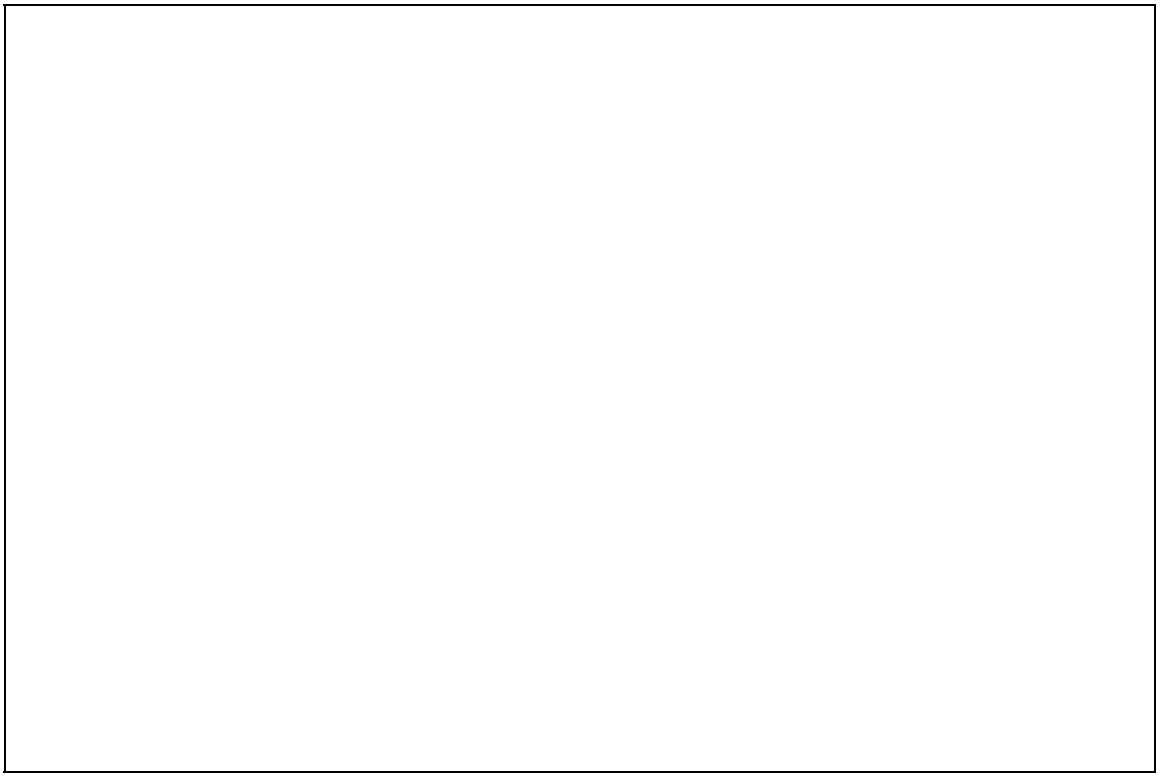

imagen 4.4 - Resultados de Triangle+Ripolin sobre *Vic_1K* (1002 parches):
a) 1002 facetas; b) 2148 facetas;
c) 1002 facetas; d) 2223 facetas.

La imagen 4.4 da una primera idea acerca de la mejora de calidad subjetiva que puede suponer el aumento de resolución en el modelado de las siluetas: a la izquierda se muestran dos perspectivas distintas de un busto de Victor Hugo descrito por unos mil triángulos de Gregory, sin subdividir; a la derecha, los resultados correspondientes a sendas subdivisiones adaptativas con límites de tolerancia (R, S_S, F_T, A) = (10, +1, 0, 0).

La imagen 4.5 muestra el efecto combinado de la subdivisión adaptativa y la interpolación del color sobre *Vic_1K* y *Vic_3K*: arriba, el resultado de la imagen 4.4.d en *z-buffer* simple y con pulido de Gouraud; abajo, el producto de la subdivisión para el busto de tres mil parches con los mismos límites de tolerancia, también en *z-buffer* simple y con pulido de Gouraud.

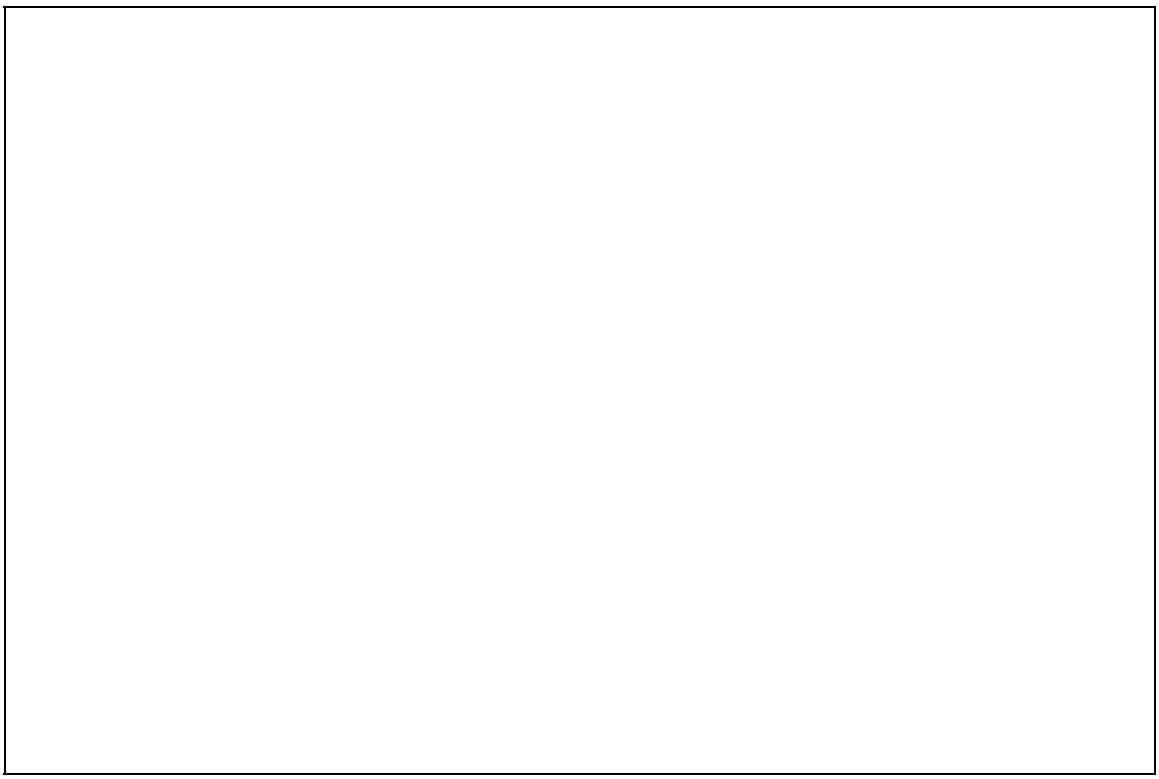

imagen 4.5 - Resultados de Triangle+Ripolin sobre:
a) *Vic_1K* (1002 parches; 2223 facetas); b) = a) + pulido de Gouraud;
c) *Vic_3K* (3002 parches; 3756 facetas); d) = c) + pulido de Gouraud.

La imagen 4.6 muestra, con la máxima resolución de Ripolin (1024 x 1024 píxeles), el resultado de una subdivisión adaptativa más exigente que la de la imagen 4.5.c, en la que Triangle generó 5 Kfacetas, y que se debe comparar al producto de la subdivisión sistemática de primer orden (12 Kfacetas) reflejado en la imagen 4.7: las siluetas de ambas superficies son indistinguibles, y el exceso de detalle de la segunda en las zonas interiores no justifica una diferencia del 140% en el número total de facetas.

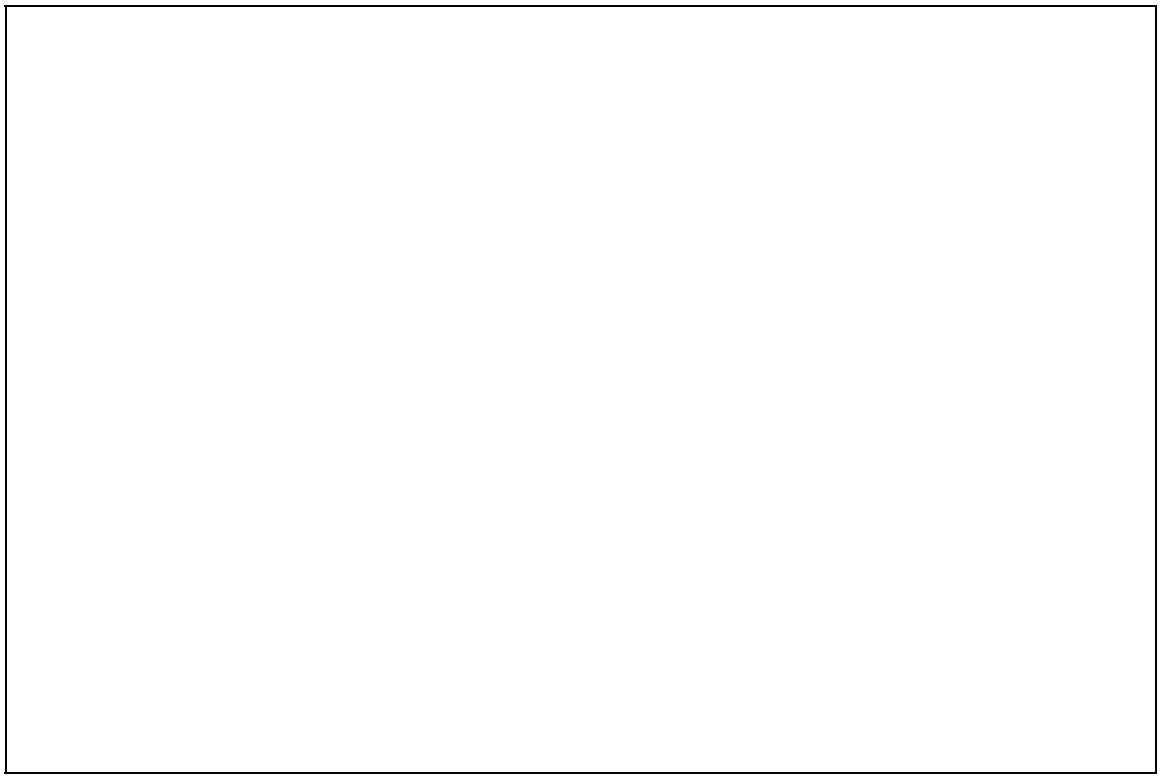

imagen 4.6 - Subdivisión adaptativa de *Vic_3K* en 5 Kfacetas.

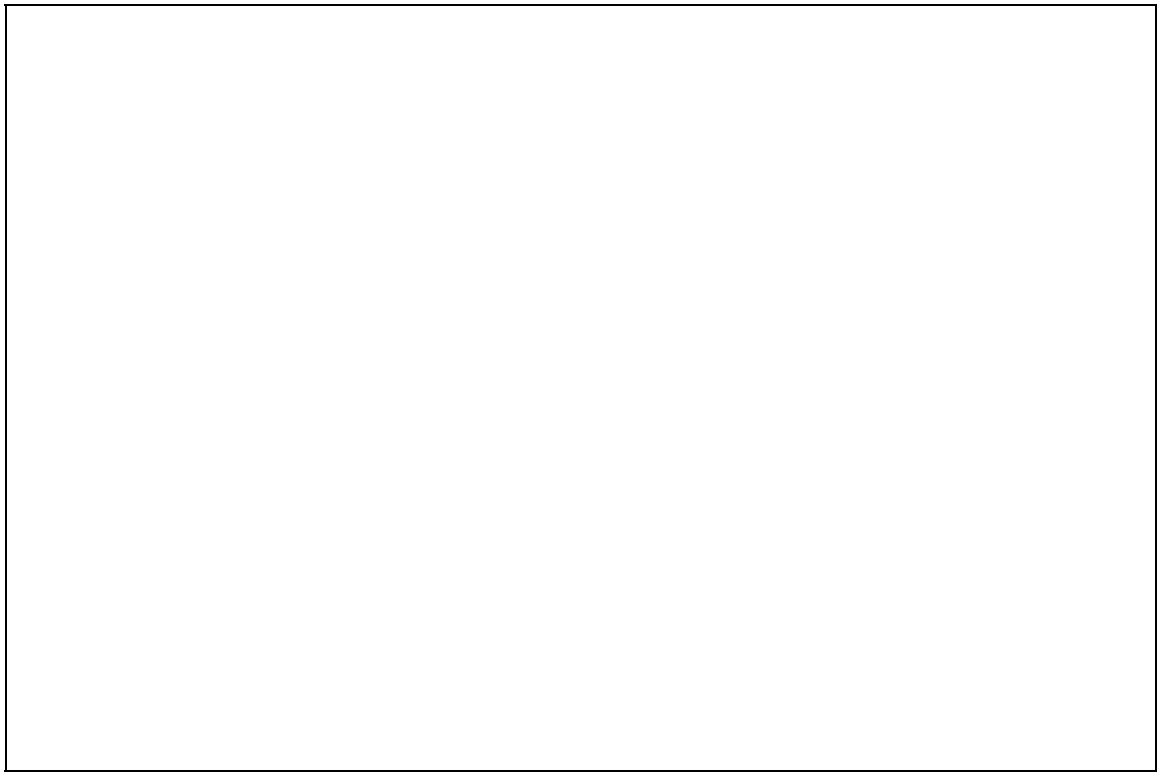

imagen 4.7 - Subdivisión sistemática de *Vic_3K* en 12 Kfacetas.

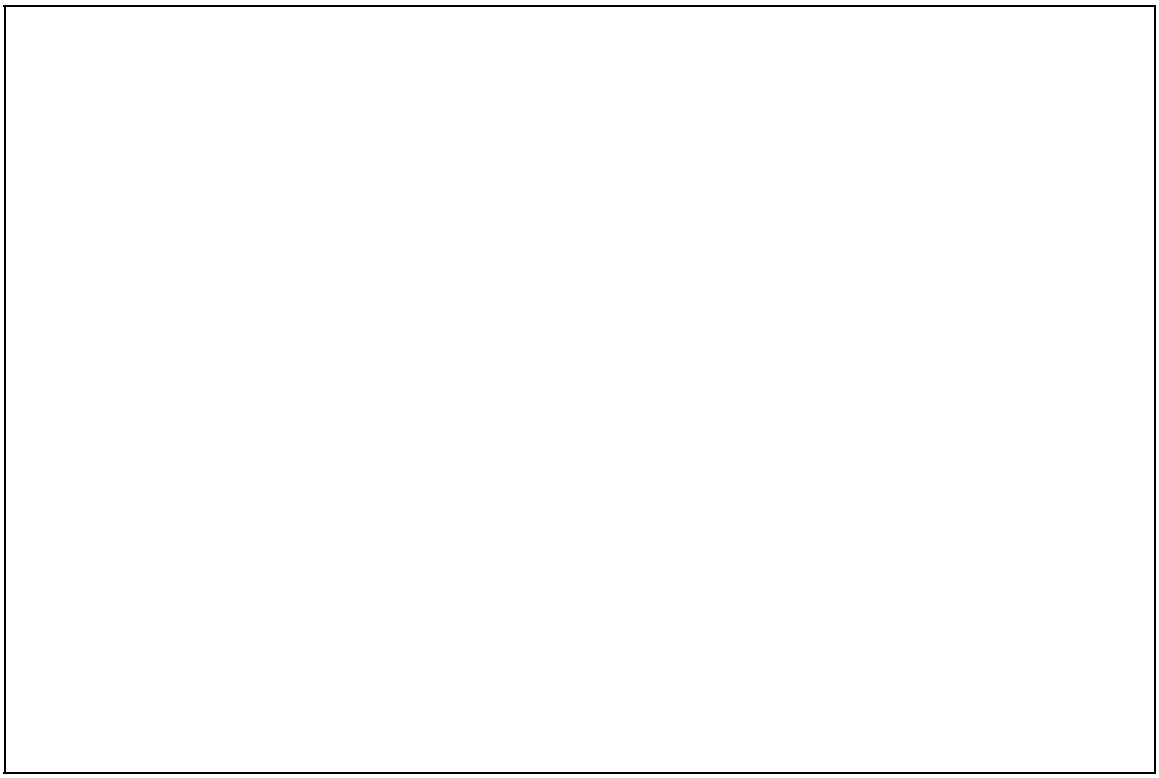

imagen 4.8 - Subdivisión sistemática de *Vic_3K* en 12 Kfacetas con pulido de Phong.

Finalmente, la imagen 4.8 corresponde a la subdivisión sistemática anterior de *Vic_3K* en 12 Kfacetas, tratada con pulido de Phong (*iexp*=1: superficie mate), y sirvió como experimento para comprobar la capacidad de Ripolin, y para comparar la eficiencia de dos tipos de modelos: aquellos en que el creador suministra las normales a la superficie (los producidos por Triangle, y aquellos en que no (formato ENSAD).

4.3. Conclusiones

A finales del mes de diciembre de 1990 quedaron a disposición de los miembros del Laboratorio IMA dos programas totalmente operativos, Triangle y Ripolin. El primero permite realizar subdivisiones adaptativas de superficies complejas modeladas por triángulos cuárticos de Gregory, para lograr recubrimientos por facetas planas. Es capaz de funcionar en solitario, aunque fue concebido como un módulo de pre-tratamiento para trabajar en equipo con el segundo, dado que sus habilidades son complementarias: Triangle facetiza de manera inteligente la superficie parcheada, generando más detalle allí donde a Ripolin le será más difícil producir resultados satisfactorios (generalmente, en las siluetas de los objetos en la imagen). La primera aplicación práctica que se le dio a Triangle fue la verificación de los modelos de superficies reconstruidas a partir de los datos resultantes del

muestreo láser de objetos 3-D reales, como el busto de Victor Hugo mostrado a lo largo de este capítulo.

Por su parte, Ripolin fue completamente reestructurado para sobrepasar sus antiguas limitaciones de tamaño de las escenas y de resolución de las imágenes. Se desarrollaron dos versiones distintas: RipLigne, que genera la imagen en *modo línea*, y RipPage, que lo hace en *modo página*. La primera, que conserva la filosofía del Ripolin original de la ENSAD, era con mucho la más utilizada en el Laboratorio (por ejemplo, las fotografías de las moléculas fueron producidas con RipLigne para un proyecto de reconocimiento de formas). Es abrumadoramente más capaz que RipPage: el nuevo límite para la talla de cada modelo de entrada quedó, para RipLigne, en 20 Kfacetas y 20 Kvértices, pudiendo obtenerse imágenes en color real de 1024 x 1024 píxeles, y codificando los valores de profundidad en el *z-buffer* asociado con una precisión de 4 octetos. RipPage toleraba sólo 1000 facetas en esta resolución, pero se emprendió su escritura para avanzar hacia el objetivo final de fusión de Triangle y Ripolin.

La integración completa de ambos permitiría olvidar definitivamente las limitaciones sobre el tamaño de las superficies de entrada si el programa resultante las leyera parche a parche, tratando cada uno individualmente, y conservando siempre la totalidad de la imagen en memoria, es decir, funcionando en *modo página*. Esto no reduciría en todos los casos el número de operaciones de E/S (para objetos verdaderamente grandes, más bien lo aumentaría, y mucho), pero presentaría además la ventaja que supone poder aprovechar los cálculos exactos de las normales a la superficie en sus vértices, frente a tener que rehacerlos de manera aproximada, lo que le ocurre a RipLigne con los modelos de tipo ENSAD. Esta ventaja se mide, sobre todo, en tiempo de cálculo, y llega a ser realmente importante: con objetos de 5 Kfacetas, RipLigne tarda un 30% menos en producir la imagen si se le dan las normales que si debe calcularlas él; pero con objetos de 12 Kfacetas, puede tardar un 70% menos. Por supuesto, este efecto no convertiría por sí solo a Ripolin en un ejemplo de interactividad —el último porcentaje mencionado se refiere a un total de casi 45' sobre un DEC-VAX 8550, suficientes para desanimar a muchos usuarios—, pero no es en absoluto despreciable cuando se añade a lo ya dicho sobre la talla virtualmente ilimitada de las escenas tratables.

Por eso parecía claro que la dirección inmediata para mejorar de los resultados de este Proyecto, sería la de continuar con la integración de Triangle y RipPage. Paralelamente a ello, durante el curso 1990-91, un equipo de alumnos de *Télécom* se encargaba de implementar en Ripolin un mecanismo de *a-buffer* para filtrar las siluetas en las imágenes (y así corregir los característicos *dientes de sierra*, mejorando la resolución aparente) y para poder abordar modelos de luz más sofisticados que permitieran tratar, por ejemplo, objetos transparentes.

Anejo A. RIPOLIN: PROYECCIÓN Y COLOR

A.1. Cálculos de perspectiva

Lo habitual, en aplicaciones de CAD, es que el creador del modelo describa la escena respecto a un origen absoluto, O, que él elige de forma arbitraria; y que exprese la posición de todos los puntos en coordenadas cartesianas, salvo la de la cámara, C, de la que da las coordenadas esféricas (r, θ, φ) respecto al punto de mira (normalmente, O: cf. figura A.1). Esto último facilita los desplazamientos del observador *alrededor* del punto de mira de una imagen a otra, que son los que más interesan, dado que en este tipo de aplicaciones suelen ser más frecuentes los movimientos giratorios de la cámara que los *travellings*.

Sin embargo, para poder proyectar cómodamente la escena 3-D sobre el plano de la imagen, es muy conveniente expresar las coordenadas de los puntos en un referencial ligado a la cámara, como se verá más adelante. Por ello, de no disponer de *hardware* especializado que lo haga, casi todo programa visualizador comienza efectuando el cambio de referencial descrito en la figura A.1, que pasa de uno ligado al origen arbitrario suministrado por el creador, como {O; **x**, **y**, **z**}, a uno ligado a la cámara, como {C; **x'**, **y'**, **z'**}.

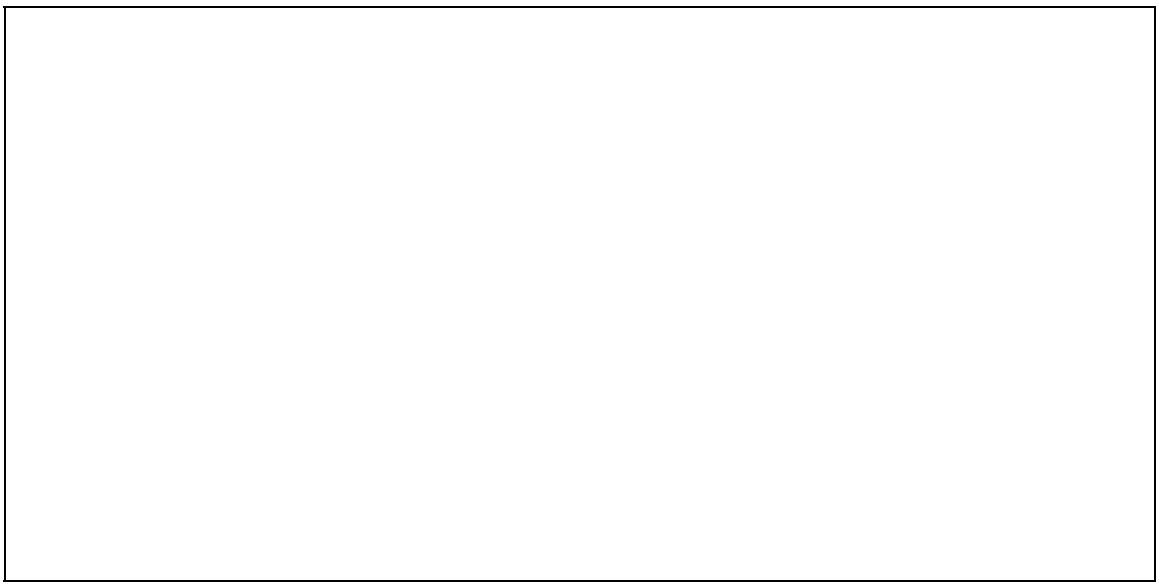

figura A.1 - Cambio de referencial (origen y base).

Así, se trata de calcular, para todo punto P, de coordenadas antiguas (x, y, z), las nuevas (x', y', z'). Para ello, basta con escribir **CP** en los dos referenciales tras definir:

$$\mathbf{OC} = x_c\mathbf{x} + y_c\mathbf{y} + z_c\mathbf{z} \text{ , con } (x_c, y_c, z_c) = r\,(\cos\varphi\cos\theta, \cos\varphi\,\mathrm{sen}\theta, \mathrm{sen}\varphi).$$

Es inmediato que **CP** = **OP** - **OC** = $(x\text{-}x_c)\mathbf{x} + (y\text{-}y_c)\mathbf{y} + (z\text{-}z_c)\mathbf{z}$, y como por otra parte es también **CP** = $x'\mathbf{x'} + y'\mathbf{y'} + z'\mathbf{z'}$, el problema se reduce a hallar la expresión de (**x**, **y**, **z**) en función de (**x**, **y**, **z**). Pero con ayuda del triedro de Frenet {**r**, **θ**, **φ**}, resulta claro que:

$$\begin{bmatrix} \mathbf{x'} \\ \mathbf{y'} \\ \mathbf{z'} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\theta} \\ -\boldsymbol{\varphi} \\ -\mathbf{r} \end{bmatrix} = \begin{bmatrix} -\operatorname{sen}\theta & \cos\theta & 0 \\ \operatorname{sen}\varphi\cos\theta & \operatorname{sen}\varphi\operatorname{sen}\theta & -\cos\varphi \\ -\cos\varphi\cos\theta & -\cos\varphi\operatorname{sen}\theta & -\operatorname{sen}\varphi \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \\ \mathbf{z} \end{bmatrix} = [M_{\theta\varphi}] \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \\ \mathbf{z} \end{bmatrix},$$

y como $[M_{\theta\varphi}]$ es una matriz de cambio de base, su inversa es su traspuesta, por lo que:

$$\begin{bmatrix} \mathbf{x} \\ \mathbf{y} \\ \mathbf{z} \end{bmatrix} = [M_{\theta\varphi}]^{-1} \begin{bmatrix} \mathbf{x'} \\ \mathbf{y'} \\ \mathbf{z'} \end{bmatrix} = [M_{\theta\varphi}]^{T} \begin{bmatrix} \mathbf{x'} \\ \mathbf{y'} \\ \mathbf{z'} \end{bmatrix}.$$

Ahora ya es posible identificar:

$$[x'\ y'\ z'] \begin{bmatrix} \mathbf{x'} \\ \mathbf{y'} \\ \mathbf{z'} \end{bmatrix} = [x\text{-}x_c\ \ y\text{-}y_c\ \ z\text{-}z_c] \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \\ \mathbf{z} \end{bmatrix} = [x\text{-}x_c\ \ y\text{-}y_c\ \ z\text{-}z_c]\ \left([M_{\theta\varphi}]^{T} \begin{bmatrix} \mathbf{x'} \\ \mathbf{y'} \\ \mathbf{z'} \end{bmatrix} \right), \text{ i.e.}$$

$$[x'\ y'\ z'] = [x\text{-}x_c\ \ y\text{-}y_c\ \ z\text{-}z_c]\ [M_{\theta\varphi}]^{T}, \text{ con } (x_c, y_c, z_c) = r\ (\cos\varphi\cos\theta, \cos\varphi\operatorname{sen}\theta, \operatorname{sen}\varphi).$$

En el caso más general de que la cámara no apunte al origen O, sino a un punto V, de coordenadas (x_v, y_v, z_v) en {O, **x**, **y**, **z**}, basta escribir **OC** como (**OV**+**VC**), es decir, tomar en la expresión anterior $(x_c, y_c, z_c) = (x_v, y_v, z_v) + r\ (\cos\varphi\cos\theta, \cos\varphi\operatorname{sen}\theta, \operatorname{sen}\varphi)$. Si además se desea introducir un ángulo α de alabeo (giro alrededor de **z'**, que afecta a la horizontalidad de la cámara, cf. fig. A.2), es necesario hacer:

$$[x'\ y'\ z'] = [x\text{-}x_c\ \ y\text{-}y_c\ \ z\text{-}z_c]\ [M_{\theta\varphi}]^{T}\ [M_{\alpha}]^{T}, \text{ con } [M_{\alpha}]^{T} = \begin{bmatrix} \cos\alpha & -\operatorname{sen}\alpha & 0 \\ \operatorname{sen}\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

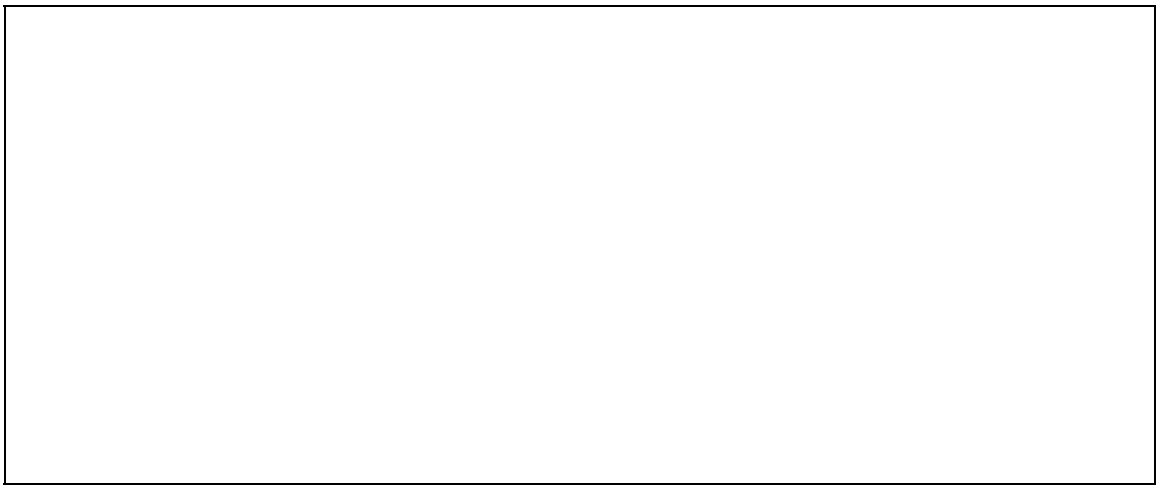

figura A.2 - Ejes de navegación de la cámara.

En definitiva, el cambio de base de {**x**, **y**, **z**} a {**x'**, **y'**, **z'**} se puede considerar como el resultado de tres rotaciones elementales encadenadas (es fácil verlo con paciencia y ayuda de la figura A.1):

1. alrededor del eje **z** original, de ángulo $(\pi/2 + \theta)$;
2. alrededor del nuevo eje **x**, de ángulo $-(\pi/2 + \varphi)$;
3. alrededor del nuevo eje **z**, de ángulo α.

Y así, se puede escribir inmediatamente:

$$\begin{bmatrix} \mathbf{x'} \\ \mathbf{y'} \\ \mathbf{z'} \end{bmatrix} = \begin{bmatrix} \cos\alpha & \text{sen}\alpha & 0 \\ -\text{sen}\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & -\text{sen}\varphi & -\cos\varphi \\ 0 & \cos\varphi & -\text{sen}\varphi \end{bmatrix} \begin{bmatrix} -\text{sen}\theta & \cos\theta & 0 \\ -\cos\theta & -\text{sen}\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \\ \mathbf{z} \end{bmatrix} =$$

$$= \quad [M_\alpha] \quad \times \quad [M_\varphi] \quad \times \quad [M_\theta] \quad \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \\ \mathbf{z} \end{bmatrix}, \text{ i.e.}$$

$$\begin{bmatrix} \mathbf{x} \\ \mathbf{y} \\ \mathbf{z} \end{bmatrix} = ([M_\alpha] [M_\varphi] [M_\theta])^{-1} \begin{bmatrix} \mathbf{x'} \\ \mathbf{y'} \\ \mathbf{z'} \end{bmatrix} = [M_\theta]^T [M_\varphi]^T [M_\alpha]^T \begin{bmatrix} \mathbf{x'} \\ \mathbf{y'} \\ \mathbf{z'} \end{bmatrix} .$$

Después de este cambio de referencial, el paso de las coordenadas 3-D a las de la imagen es trivial (en la figura A.3, se ilustra el principio del procedimiento para x):

1. se define z=1 para el plano de la imagen, que es un cuadrado de lado tg(s), siendo s el semiángulo de visión elegido por el usuario[14];
2. de cada punto espacial P, se proyectan las coordenadas x_{3D} e y_{3D} sin más que dividirlas por z_{3D}, obteniéndose x_{2D} e y_{2D};

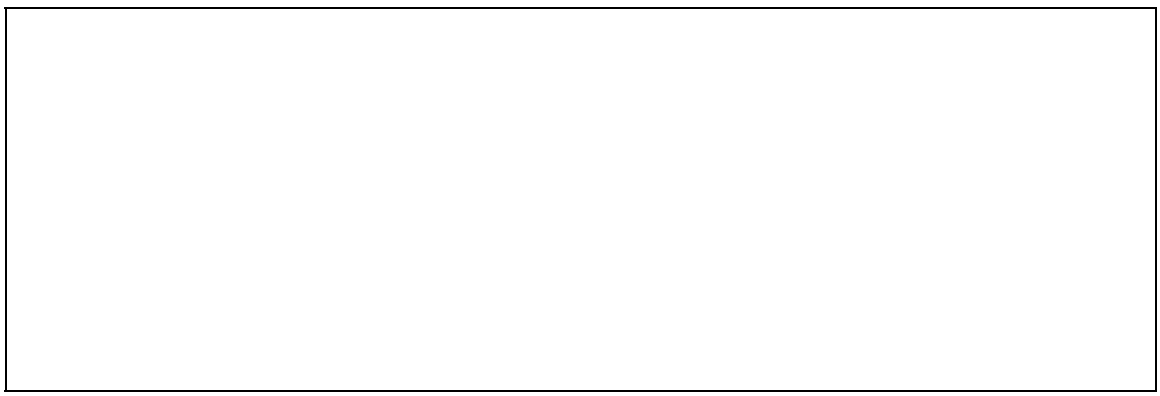

figura A.3 - Proyección de x_{3D} sobre x_{2D}.

[14] NB: tg(s) es (aproximadamente) inversamente proporcional a la distancia focal del objetivo.

3. por comodidad, se normalizan x_{2D} e y_{2D}, dividiéndolas por tg(s): se obtienen así x_{2DN} e y_{2DN} que verifican, para los puntos que caen dentro del cuadro, $-1 \le x_{2DN}, y_{2DN} \le 1$;
4. finalmente, se calculan las coordenadas en la imagen, que mide *ppl* x *lpi* (píxeles por línea x líneas por imagen) píxeles:

   $x_i = 1 + \text{int}\,[(ppl\text{-}1)(x_{2DN}+1)/2]$ ➡ $1 \le x_i \le ppl$;

   $y_i = 1 + \text{int}\,[(lpi\text{-}1)(y_{2DN}+1)/2]$ ➡ $1 \le y_i \le lpi$;
5. si el formato del píxel en la pantalla no es cuadrado, sino rectangular, con relación de aspecto ρ (ancho/alto), se puede introducir una corrección en el cálculo de y_i:

   $y_i = 1 + \text{int}\,[\rho(lpi\text{-}1)(y_{2DN}+1)/2]$ ➡ $1 \le y_i \le \rho(lpi\text{-}1) \approx \rho \cdot lpi$.

A.2. Modelado de la luz

En Ripolin se considera un modelo de iluminación simplificado consistente en una única fuente puntual infinitamente alejada de la escena en una dirección elegida por el usuario. Este "sol" genera, por una parte, una luz ambiente descrita por medio de los parámetros *amb1* y *amb2*, y por otra, produce reflejos especulares en la superficie de los objetos definidos como brillantes mediante el parámetro *exp*, que se aprecian en la imagen únicamente cuando se utiliza el mecanismo de interpolación del color de Phong. En cualquier caso, todos los objetos son completamente opacos a la luz, y no producen sombras.

En los casos de *z-buffer* simple y Gouraud, sólo se considera la luz ambiente, y la luminancia *lum* en un punto se determina independientemente de la posición del observador: únicamente se tiene en cuenta si el ángulo *a* que forman el vector normal a la superficie ($\mathbf{V_N}$) y el "vector luz" ($\mathbf{V_L}$), que se dirige desde ese punto hacia la fuente luminosa (véase la figura A.4.a). Concretamente:

si $a \le 90°$, $lum = amb1 + \cos(a)(1\text{-}amb1)$ ➡ $amb1 \le lum \le 1$;

si $a > 90°$, $lum = amb1 - \cos(a)(amb2\text{-}amb1)$ ➡ $amb2 \le lum \le amb1$, con $0 \le amb2 \le amb1 \le 1$.

En el ejemplo de la figura A.4.a, la esfera está dividida en dos mitades: aquellos puntos como P_1 en los que el vector normal se dirige hacia la luz tienen luminancias comprendidas entre 1 (punto más cercano al foco) y *amb1* (puntos de la frontera); para los puntos de la otra semiesfera, como P_2, la luminancia desciende progresivamente hasta *amb2* (punto más lejano del foco). Valores típicos son *amb1* = 0,5 y *amb2* = 0,3.

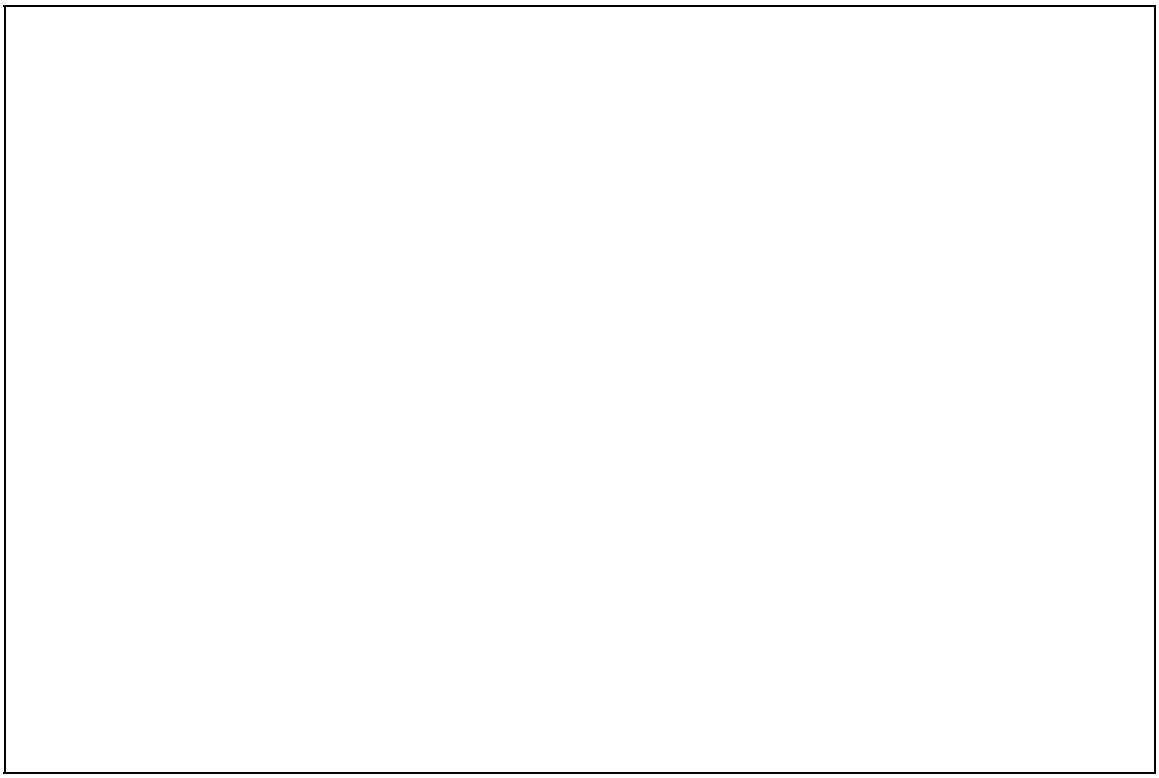

figura A.4 - Modelo de iluminación: a) luz ambiente; b) reflejo especular.

En el caso de Phong, se considera además el posible reflejo del foco sobre la superficie de objetos brillantes, que evidentemente sólo tiene interés si la cámara puede ver la zona donde se produce. La intensidad de este reflejo, que se añade a la luminancia debida a la luz ambiente, se calcula como sigue a partir de *exp*, parámetro con el que el usuario controla el "calor" del objeto (cf. fig. A.4.b): siendo b el ángulo formado por $\mathbf{V_O}$ y el "vector reflejo" $\mathbf{V_R}$, simétrico de $\mathbf{V_L}$ respecto a $\mathbf{V_N}$,

si $\cos(b) > 10^{-3,8/exp}$, $lum' = [\cos(b)]^{exp}$;

si $\cos(b) \leq 10^{-3,8/exp}$, $lum' = 0$.

Como se ve en la expresión anterior, *exp* gobierna el tamaño de la zona del reflejo, y sobre todo, la rapidez con que su intensidad decrece allí donde existe. Así, un valor de 30, típico para objetos brillantes, hace que, a partir de $\cos(b) > 0,75$ ($b < 41°$) exista reflejo, aunque sólo para una zona mucho más pequeña sea importante, ya que es $lum' = [\cos(b)]^{30}$ ($lum' > 0,5$ para $b < 12°$); sin embargo, $exp = 5$, valor aplicable a objetos mate, proporciona un decrecimiento más lento, en un área mayor: $lum' > 0$ para $b < 80°$, y $lum' > 0,5$ para $b < 30°$.

Queda por explicar la constante "-3,8": se trata de un valor puramente empírico que gobierna la discontinuidad en la función *lum'*, que vale respectivamente 0 y $10^{-3,8}$ (en este caso) a un lado y a otro del límite para $\cos(b)$. Cuando el salto es pequeño (ligeramente mayor de una diezmilésima para $10^{-3,8} \approx 10^{-4}$), no es apreciable; y si lo fuera, el reflejo tendría un contorno más marcado, lo que no es en principio indeseable: sugeriría una fuente luminosa redonda, en lugar de puntual.

A.3. Interpolación del color

El "pulido" del color se realiza, tanto en el método de Gouraud como en el de Phong, mediante una interpolación bilineal. La diferencia estriba en aquello que se interpola: en el caso de Gouraud, dado un punto interior de una faceta, se halla su luminancia interpolando las luminancias de dos pares de vértices de la faceta, previamente calculadas a partir de los respectivos vectores normales; mientras que en el caso de Phong, se interpola el vector normal en el punto a partir de esos cuatro vectores normales, y sólo al final se calcula la luminancia.

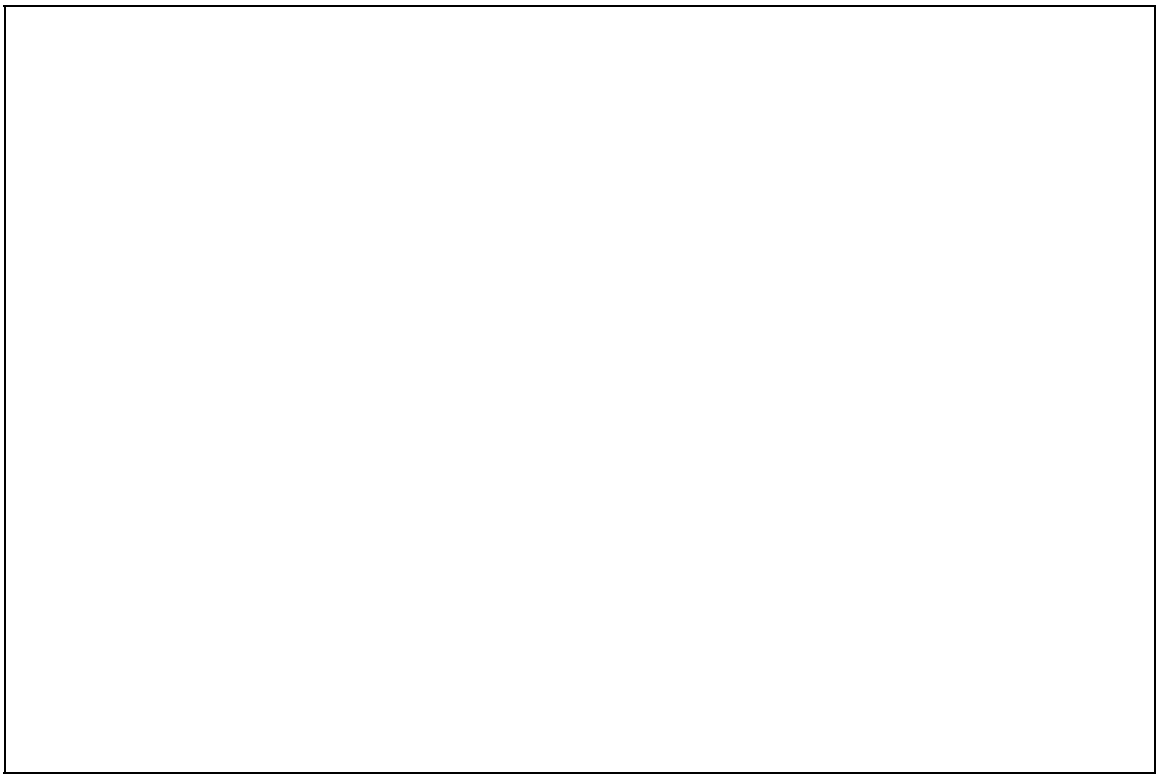

figura A.5 - Interpolación del color:
a) método de Gouraud; b) id. de Phong;
c) diferencia.

Las figuras A.5.a y A.5.b ilustran ambos mecanismos que, tomando dos pares de puntos (tres puntos son suficientes aunque se hayan dibujado cuatro por claridad), calculan primero mediante interpolación lineal, las luminancias o las normales en los puntos "14" y "23", que tienen por coordenada y_i la del punto objetivo; después, mediante la segunda interpolación lineal sobre x_i, se halla el valor deseado. La figura A.5.c ilustra la superioridad del método de Phong en un caso concreto en el que la superficie es una chapa ondulada (vista de perfil) que ha debido ser modelada con facetas planas. Por simetría, *lum1* = *lum2*, con lo que, interpolando luminancias, se obtendría un color constante en todo el segmento, y se perdería la impresión de ondulación; sin embargo, interpolando vectores normales primero y calculando la luminancia después, se logra la variación deseada del color a lo largo del segmento.

Anejo B. RIPOLIN: DESCRIPCIÓN DE FUNCIONES

En este anejo se describen las funciones de todos los módulos del programa RipLigne, en su versión del 2 de diciembre de 1990 (última correspondiente a este Proyecto). El árbol de ejecución del programa que sigue a este párrafo no pretende ser un organigrama exhaustivo, sino servir de referencia para entender la jerarquía de las subrutinas. En concreto, se han omitido en él, por claridad, todas las llamadas a las funciones matemáticas y a los procedimientos de E/S de pantalla y teclado (diálogos interactivos con el usuario).

RipLigne

(polígonos cualesquiera)
Lec_Obj_ENSAD
  Test_Max
  read FORTRAN

(polígonos fijos)
Lec_Obj_Ngones
  Test_Max
  LecImaD

(en todo caso)
Param_Obs
  [read FORTRAN][o]

(*z-buffer* simple)
CoulfFac
  CaLumi
CalClip
  jClip
jOrgFac
(jZbuff8|jZbuff24)[v]
  AffIni
  [LecIma][z]
  FacActi
  EcrIma
  write FORTRAN

("pulido" del color)
Lissages
  (CalNorm|CalNorm2)[n]
  [CouLis][v]
  CaLumi
  CalClip
    jClip
  jOrgFac
  (jZbuff8|jZbuff24)[v]
    AffIni
    [LecIma][z]
    FacActi
    (jGouraud|jPhong)[g]
    EcrIma

---

[o] La lectura del fichero de parámetros de observación tiene lugar si `obs=TRUE`.

[n] Si `norm=TRUE`, `CalNorm2`; si no, `CalNorm`.

[v] Si `VraieCouleur=TRUE`, `jzbuff24`; si no, `jzbuff8`.

[z] La lectura de la imagen y el *z-buffer* para completar se efectúa si `zin=TRUE`.

[g] Si el usuario eligió el método de Gouraud, `jGouraud`; si eligió el de Phong, `jPhong`.

B.1. RIPLIGNE.FOR

B.1.1. `Program RipLigne`

Programa principal: dialoga con el usuario sobre las dimensiones de la imagen y el tipo de interpolación del color deseado, e invoca a las subrutinas que proceda, según lo establecido en el árbol jerárquico de ejecución presentado más arriba.

En concreto: si el número de aristas por cara leído del fichero de opciones (`i`) es 0, llama a `Lec_Obj_ENSAD`, y si no, a `Lec_Obj_Ngones`; si `zin=TRUE`, no tiene lugar el diálogo sobre dimensiones de la imagen producto (serán las de la imagen de entrada); si el usuario elige alguno de los mecanismos de "pulido" del color, invoca a `Lissages`, y si elige el *z-buffer* simple, es el propio programa principal quien orquesta la síntesis.

B.1.2. `Subroutine Lec_Obj_ENSAD (numobj)`

Lee el objeto contenido en el fichero lógico nº `numobj`, descrito mediante polígonos de número de aristas variable, y con especificación completa de colores de todas las facetas y del fondo (formato ENSAD). Si `norm=TRUE`, lee también las normales exactas a la superficie en sus vértices, que figuran entonces al final del fichero.

Realiza un chequeo de las dimensiones del objeto llamando a `Test_Max` tras leer los números totales de vértices, aristas y caras (`ns`, `na`, `nf`). Todas las lecturas se realizan con el comando `read` de FORTRAN.

B.1.3. `Subroutine Lec_Obj_Ngones (i, sommets, faces)`

Lee un objeto modelado con polígonos de `i` aristas. La información relativa a los vértices (y, si `norm=TRUE`, a las normales) figura en el fichero lógico nº `sommets`, y la que toca a las facetas, en el nº `faces`.

Realiza un chequeo de las dimensiones del objeto llamando a `Test_Max` tras averiguar los números totales de vértices, aristas y caras: `ns` y `nf` se obtienen consultando el formato de `sommets` y `faces` con la subrutina `Frmt` del Labo. IMA; `na=nf*i`. Todas las lecturas se realizan con la rutina `LecImaD` del Labo. IMA.

B.1.4. `Subroutine Test_Max`

Imprime `ns`, `na` y `nf`, y verifica que son tolerables para la versión del programa en curso (i.e. inferiores respectivamente a `sommax`, `aretemax`, `facemax`). Si lo son, devuelve el control; si no, es el `Fin`.

B.1.5. `Subroutine Param_Obs (couleurs)`

Lee (si `obs=TRUE`) o escoge (si no) valores para los parámetros de observación: posición de la cámara y del punto de mira, ángulos de visión y alabeo, planos de *clipping* delantero y trasero, dirección e intensidad (`amb1`, `amb2`) de la fuente luminosa.

Si `couleurs=TRUE` (el objeto no está descrito según el formato ENSAD en el que se precisa el color de cada faceta), lee (si `obs=TRUE`) o escoge (si no) el color único para todo el objeto (básico y de reflejo especular; "calor" de la superficie), y el del fondo.

Finalmente, presenta todos esos parámetros en una pantalla de resumen al usuario. Todas las lecturas se realizan mediante el comando `read` de FORTRAN.

B.2. AFFINI.FOR

B.2.1. `Subroutine AffIni`

Determina la secuencia de proceso de las facetas, cargando la tabla `ifacaux` con los números de las facetas ordenadas según su *z*, en sentido decreciente.

B.3. CALCLIP.FOR

B.3.1. `Subroutine CalClip`

Realiza los cálculos de *clipping*: pasa las coordenadas de los vértices al referencial ligado a la cámara, desplaza los planos de *clipping* delantero y trasero hasta los puntos extremos del objeto en *z* para mejorar la precisión de los cálculos, forma la matriz de proyección conforme al ángulo de visión, llama a `jClip` (que gestiona los cambios en la estructura del objeto debidos al encuadre), y finalmente, calcula las coordenadas en pantalla de los vértices que han logrado sobrevivir a todo el proceso.

B.4. CALNORM.FOR

B.4.1. `Subroutine CalNorm (js, nbvoisin)`

Calcula el vector normal unitario a la superficie en su vértice nº `js`, que tiene `nbvoisin` vecinos (i.e. está conectado por aristas con otros `nbvoisin` vértices), y la luminancia correspondiente. Se utiliza cuando el creador del modelo no suministra las normales, con lo que la determinación del vector normal en el vértice implica realizar un promedio de los productos vectoriales de las aristas que concurren en él (y que dan la dirección de los vectores normales a las facetas que lo comparten).

Almacena los resultados en la fila `js` de la matriz global `tablis`.

B.4.2. `Subroutine CalNorm2`

Calcula los vectores normales unitarios a la superficie en todos sus vértices, y las luminancias correspondientes, a partir de las normales (no necesariamente unitarias) suministradas por el creador del modelo.

Almacena los resultados en la matriz global `tablis`.

B.5. CALUMI.FOR

B.5.1. `Subroutine CaLumi (jf, nsom1, nombsom, ical)`

Calcula la luminancia (`ical=0`) o el vector normal unitario (`ical=1`) de la faceta nº `jf`, que tiene `nombsom` vértices, siendo el primero el nº `nsom1`. Para ello, realiza el producto vectorial de dos aristas no paralelas de la faceta.

Devuelve el resultado en la fila `jf` de la matriz global `faclum`.

B.6. COULFAC.FOR

B.6.1. `Subroutine CoulFac`

Utilizada cuando se sintetiza la escena por medio de un *z-buffer* simple. Determina el color de cada faceta llamando a `CaLumi` con `ical=0` (cálculo de luminancia). Luego, si no se

está trabajando en color real (`VraieCouleur=FALSE`), calcula la paleta de 256 colores más representativa para la escena.

B.7. COULIS.FOR

B.7.1. `Subroutine CouLis`

Utilizada cuando se sintetiza la escena con alguno de los métodos de "pulido" del color, y en pseudo-color (`VraieCouleur=FALSE`). Determina, a partir de los datos de la matriz global `tablis`, las luminosidades máxima y mínima en cada faceta, y luego calcula las tonalidades de cada color básico que mejor representan la escena.

B.8. FACACTI.FOR

B.8.1. `Subroutine FacActi (jf, iy, ism, is1, is2)`

Activa o *desactiva* la faceta nº `jf`, teniendo en cuenta que la línea en curso es `iy`. Si la faceta es cóncava (vértices incorrectamente ordenados), o no ha sido alcanzada aún, o ya ha sido tratada, no se hace nada; en el resto de los casos, la faceta es *activada*, y se ponen en `ip1`, `ip2`, `ip3` e `ip4` (variables globales) los números de los 4 vértices utilizados para la interpolación del color (cf. anejo A), hallados a partir de los vértices `ism`, `is1` e `is2`.

B.9. FIN.FOR

B.9.1. `Subroutine Fin (message)`

Imprime el mensaje de error `message` que ha provocado la muerte en esta ocasión, y termina la ejecución del programa.

B.10. INOUT.FOR

B.10.1. Subroutine IO_Init

Inicializa el terminal, crea la ventana de diálogo con el usuario y coloca el cursor lógico en la esquina superior izquierda.

B.10.2. Subroutine IO_Reset

Cierra la ventana de diálogo y "resetea" el terminal.

B.10.3. Subroutine Clear_Screen

Borra la ventana de diálogo y coloca el cursor lógico en la esquina superior izquierda.

B.10.4. Subroutine Write_Ln (String)

Imprime `String` en la posición del cursor, cuya fila incrementa y cuya columna pone a 1.

B.10.5. Subroutine Write_Str (String)

Imprime `String` en la posición del cursor, cuya columna incrementa tantas unidades como caracteres tenga `String` (la fila no varía).

B.10.6. Subroutine Write_Int (Num, Length)

Imprime `Num`, en un campo de `Length` caracteres, a partir de la posición del cursor, cuya columna incrementa `Length` unidades (la fila no varía).

B.10.7. Subroutine Start_Timer

Pone en marcha el cronómetro.

B.10.8. `Subroutine Write_Time`

Imprime el tiempo de CPU transcurrido desde la puesta en marcha del cronómetro (la primera vez que es invocada), o desde la última invocación (las restantes), en un campo justificado a la derecha en la fila del cursor, que es incrementada (la columna vuelve a 1).

B.10.9. `Subroutine Write_Total_Time`

Imprime el tiempo de CPU transcurrido desde la puesta en marcha del cronómetro, en un campo justificado a la derecha en la fila del cursor, que es incrementada (la columna vuelve a 1).

B.10.10. `Subroutine Time_To_String (Time, String)`

Convierte `Time` (expresado en centésimas de segundo) en la cadena `String`, que tiene el formato "mm:ss.cc" o ">= 1h. ".

B.10.11. `Function Get_Int (IMin, IMax)`

Pide al usuario un entero entre `Imin` e `Imax` y devuelve ese valor después de verificar que está entre los límites especificados (ambos comprendidos). Incrementa la fila del cursor, y no altera la columna.

B.10.12. `Function Get_Key (Options)`

Pide al usuario que pulse una tecla de las contenidas en la cadena `Options`, y devuelve ese valor. No establece diferencias entre mayúsculas y minúsculas, y acepta `[ENTER]` o `[SPACE]` como elección de la opción por defecto, que es la primera letra de `Options`. Incrementa la fila del cursor, y no altera la columna.

B.10.13. `Function Upper (c)`

Retorna la mayúscula correspondiente a `c`.

B.11. JCLIP.FOR

B.11.1. `Subroutine jClip (xyzw)`

Gestiona la estructura del objeto durante el *clipping*, borrando aquellos vértices, aristas y facetas que sean truncados por el encuadre, y añadiendo los que aparezcan.

B.12. JGOURAUD.FOR

B.12.1. `Subroutine jGouraud (intensi, iy, jf, jx, jc)`

Cálcula `intensi`, luminancia interpolada por el método de Gouraud, y a partir del color básico nº `jc`, del píxel que se encuentra en la columna `jx` de la fila `iy`, y corresponde a la faceta nº `jf`.

B.13. JORGFAC.FOR

B.13.1. `Subroutine jOrgFac`

Verifica que las facetas son convexas (vértices correctamente ordenados), y halla las dos líneas extremas (superior e inferior) correpondientes en la pantalla a cada faceta.

B.14. JPHONG.FOR

B.14.1. `Subroutine jPhong (intensi1, intensi2, iy, jf, jx, jc)`

Cálcula `intensi1` e `intensi2`, luminancias ambiente y especular interpoladas por el método de Phong, y a partir del color básico nº `jc`, del píxel que se encuentra en la columna `jx` de la fila `iy`, y corresponde a la faceta nº `jf`.

B.15. JZBUFF.FOR

B.15.1. `Subroutine jZbuff`

Si `VraieCouleur=TRUE` (color real), llama a `jZbuff24`; si no, a `jZbuff8`.

B.15.2. `Subroutine jZbuff8`

Produce la imagen en modo línea, iterando primero sobre las líneas (de arriba abajo) y luego sobre las facetas (de atrás adelante, sacándolas de `ifacaux` en el orden marcado previamente por `AffIni`). Antes de comenzar los cálculos para cada línea, la inicializa con el color del fondo. Antes de tratar cada faceta, mira su *status* llamando a `FacActi`. Llama a `jGouraud` o `jPhong` para determinar el color de cada píxel si se está utilizando alguno de los métodos de pulido del color.

Las operaciones secuenciales de escritura de las líneas de la imagen las realiza la rutina `EcrIma` del Labo. IMA. Los números de los colores asignados a los píxeles (0..255) se pueden traducir al RGB correspondiente por medio de una LUT que se graba, mediante el mandato `write` de FORTRAN, en un fichero aparte una vez terminada la producción de la imagen.

B.15.3. `Subroutine jZbuff24`

Como `jZbuff8` con las siguientes salvedades:

1. existe la posibilidad de completar una imagen anterior: en ese caso, antes de procesar cada línea, los píxeles se inicializan conforme a la imagen antigua, y se lee la línea correspondiente del *z-buffer* (todo ello mediante la rutina `LecIma` del Labo. IMA);
2. también cabe la posibilidad de salvar el estado final del *z-buffer*, para poder completar después la escena: si el usuario eligió esta opción (`zout=TRUE`), al final del proceso de cada línea se escribe, además de la imagen, el contenido del *z-buffer*;
3. al trabajar con color real, no existe LUT: la imagen es auto-suficiente.

B.16. LISSAGES.FOR

B.16.1. Subroutine Lissages

Utilizada cuando se interpola el color. Si `norm=TRUE`, llama a `CalNorm2`, que realiza todo el trabajo de determinación de los vectores normales en los vértices; si no, llama a `CalNorm`, pero no antes de averiguar cuántos vecinos tiene cada vértice. Si se trabaja con pseudo-color, llama a `CouLis`. Después, halla el vector normal a cada faceta invocando a `CaLumi` con `ical=1` (cálculo de vector normal). Luego llama a `CalClip` para que se encargue del *clipping*, y finalmente a `jOrgFac` y a `jZbuff`.

B.17. MATH.FOR

B.17.1. Subroutine MatPer

Forma la matriz de perspectiva, almacenando el resultado en la matriz global `perma`.

B.17.2. Subroutine MatAxe (a, angle, pt, vec)

Calcula la matriz de transformación `a` correspondiente a una rotación de `angle` grados alrededor del eje de dirección `vec` que pasa por `pt`.

B.17.3. Subroutine MatFab (a, it, angle, xt, yt, zt)

Fabrica la matriz `a` según el valor de `it`:

1. rotación de `angle` grados alrededor del eje de alabeo;
2. id. id. cabeceo;
3. id. id. guiñada;
4. translación de vector (`xt`, `yt`, `zt`);
5. afinidad;
6. rotación de `angle` grados alrededor del eje de dirección (`xt`, `yt`, `zt`).

B.17.4. `Subroutine MatPro (a, b, c)`

Calcula el producto matricial `a=b*c`. `a` puede ser la misma variable que `b` ó `c`, y todas ellas son matrices de 4 x 4.

B.17.5. `Subroutine jProVec (v1, v2, v3)`

Calcula el producto vectorial `v3=v1^v2`. Todos ellos son vectores de 3 x 3.

B.17.6. `Subroutine MatVec (a, b, t)`

Calcula el producto del vector `b` (3) por la matriz `t` (4 x 4), y almacena el resultado en `a`, que puede ser la misma variable que `b`. Para ello, añade un 1 en la cuarta columna al vector `b`, y descarta la cuarta columna del resultado.

B.17.7. `Subroutine MatVec4 (a, b, t)`

Calcula el producto del vector `b` (4) por la matriz `t` (4 x 4), y almacena el resultado en `a`, que puede ser la misma variable que `b`.

Anejo C. TRIANGLE: PARCHES ALABEADOS

En este anejo se define la expresión matemática, se comentan las principales propiedades geométricas, y se dan ejemplos, de tres familias de parches alabeados de especial importancia: los cuadrados de Bézier (cB en adelante), los triángulos de Bernstein-Bézier (tBB), y los triángulos de Gregory (tG). En [DU-88] se realiza un estudio en profundidad de todos ellos, orientado a la reconstrucción de superficies complejas muestreadas. Por otra parte, descripciones más extensas de estos y otros tipos de parches pueden encontrarse en [BARNHILL-77], [BÉZIER-86], y [FAUX-79].

C.1. Cuadrados de Bézier

C.1.1. Definición

Un cB de grado $m \times n$ se define con un *grafo de control* (o malla característica) formado por $(m+1) \times (n+1)$ puntos del espacio 3-D, repartidos en una cuadrícula topológicamente regular, y se expresa como producto tensorial de dos curvas de Bézier de grados m y n:

$$\mathbf{Q}(u, v) = \sum_{i=0}^{m} \sum_{j=0}^{n} \mathbf{P}_{ij} \, B_i^m(u) \, B_j^n(v), \quad \text{donde } B_k^L(t) = \frac{L!}{k!(L-k)!} t^k (1-t)^{L-k} \tag{C.1}$$

[$B_k^L(t)$ es el k-ésimo término del polinomio de Bernstein de grado L de una variable, en el que $0 \le k \le L$].

En la ecuación (C.1), u y v ($0 \le u, v \le 1$) son las coordenadas paramétricas de un punto de la superficie del parche, que dan sus coordenadas espaciales (las de $\mathbf{Q}(u, v)$) en función de las de los puntos de control $\mathbf{P}_{ij}$. De hecho, el cB definido por (C.1) puede imaginarse como una proyección en el espacio del cuadrado plano de lado unidad, que se denomina habitualmente *dominio paramétrico asociado*. La figura C.1 muestra un cuadrado bicúbico de Bézier (c3B), y la relación entre el dominio paramétrico y el grafo de control asociados: nótese que los cuatro puntos de control de las esquinas son los únicos que necesariamente pertenecen al parche, por lo que son los únicos de los que cabe dar las coordenadas paramétricas (aunque guarden una gran similitud con los parches de tipo *spline*, los de Bézier no interpolan todos los puntos de control).

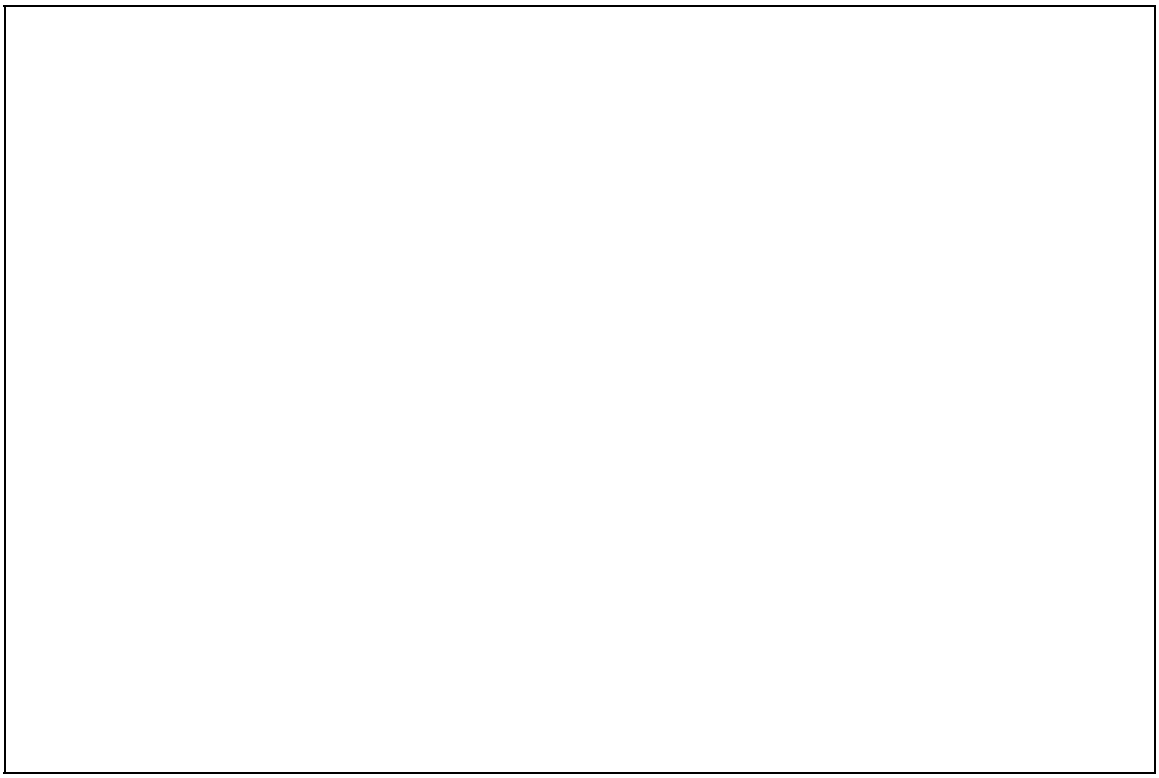

figura C.1 - C3B: dominio paramétrico y grafo de control.

C.1.2. Propiedades geométricas

De entre las propiedades de los cB, las más interesantes en el ámbito del diseño asistido por ordenador son las que atañen a su forma espacial. Por ejemplo, la superficie del parche es moldeada por la malla característica, como si los puntos de control "tirasen elásticamente" de ella sin tocarla (salvo los de las esquinas), lo que es de gran utilidad cuando se conciben objetos modelados por este tipo de parches. También es provechoso saber que la totalidad del parche queda dentro de la *envoltura convexa* (*convex hull*) del grafo de control. Y que la expresión de un borde del parche, que es una curva de Bézier (de grado m o n según la orientación del segmento paramétrico asociado), se obtiene de (C.1) sin más que reemplazar el parámetro elegido por 0 ó 1: por ejemplo, tomando u=0 se obtiene

$$\mathbf{Q}\,(0, v) = \mathbf{\Gamma}\,(v) = \sum_{i=0}^{m}\sum_{j=0}^{n}\ \mathbf{P}_{ij}\,\mathrm{B}_i^m(0)\,\mathrm{B}_j^n(v) \;=\; \sum_{j=0}^{n}\ \mathbf{P}_{0j}\,\mathrm{B}_j^n(v), \quad \text{con } 0 \le v \le 1, \qquad \text{(C.2)}$$

que es la expresión del borde vertical izquierdo (únicamente determinada, como se ve en (C.2), por los puntos de control situados sobre ese borde en el grafo de control).

Además, como es lógico, el grado del cB utilizado determina la máxima complejidad que puede tener la superficie que se desea modelar, para un error de aproximación dado: por lo general, los cuadrados bicúbicos como el de la figura C.1, suelen ser suficientes, y son con mucho los más empleados en aplicaciones de CAD.

C.2. Triángulos de Bernstein-Bézier

C.2.1. Definición

Un tBB de grado m se define con un *grafo de control* formado por $(m+1)(m+2)/2$ puntos del espacio 3-D, repartidos en una retícula triangular topológicamente regular, y se expresa como:

$$\mathbf{Q}(u, v, w) = \sum_{i+j+k=m} \mathbf{P}_{ijk}\, B^m_{ijk}(u, v, w), \quad \text{donde } B^m_{ijk}(u, v, w) = \frac{m!}{i!j!k!} u^i v^j w^k \qquad \text{(C.3)}$$

[$B^m_{ijk}(u, v, w)$ son los polinomios de Bernstein de grado m de dos variables independientes en un dominio triangular, en los que $0 \le i, j, k \le m$, y $i+j+k=m$].

En la ecuación (C.3), u, v y w ($0 \le u, v, w \le 1$) son las coordenadas paramétricas de un punto de la superficie del parche, que dan sus coordenadas espaciales en función de las de los puntos de control $\mathbf{P}_{ijk}$. No son independientes, sino que verifican $u+v+w = 1$. De manera análoga a lo que ocurre con los cB, el tBB definido por (C.3) puede imaginarse como una proyección en el espacio del triángulo equilátero plano de lado unidad, que es el dominio paramétrico asociado al tBB. La figura C.2 muestra un triángulo cuártico de Bernstein-Bézier (t4BB), y la relación entre el dominio paramétrico y el grafo de control asociados: nótese que, como antes, sólo los puntos de control de las esquinas están sobre el parche, por lo que son los únicos que tienen coordenadas paramétricas.

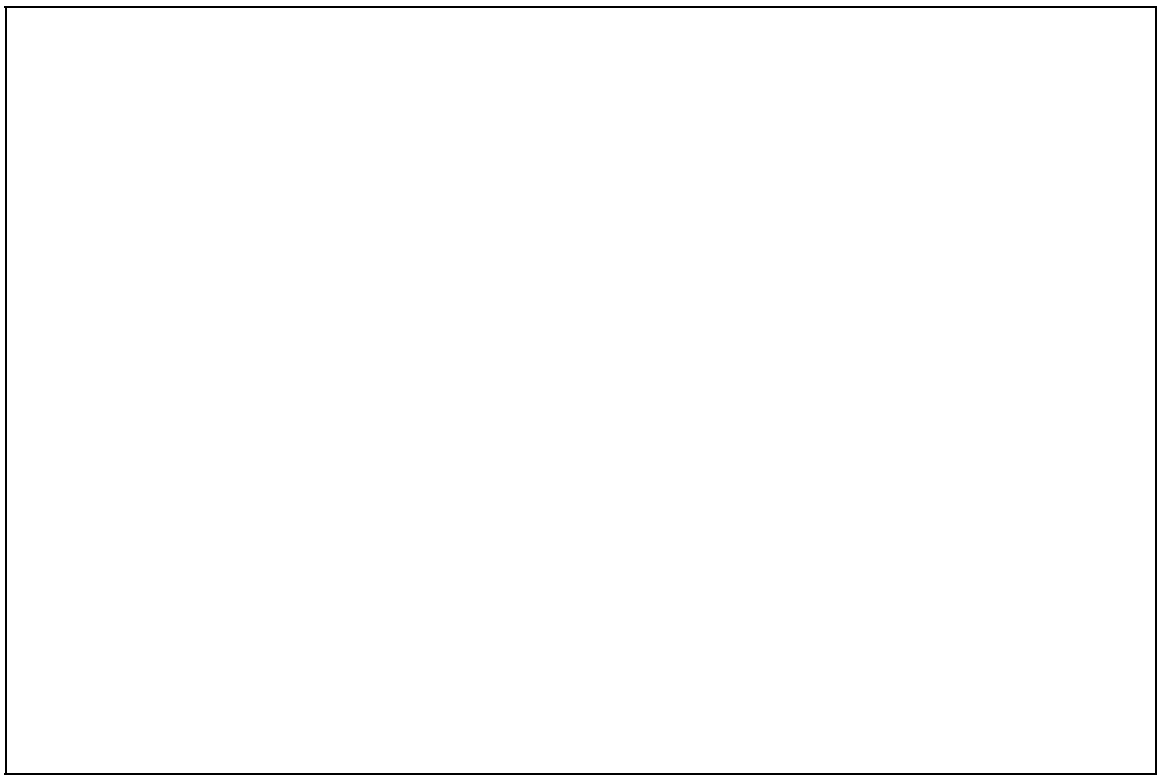

figura C.2 - T4BB: dominio paramétrico y grafo de control.

C.2.2. Degeneración de los bordes

Los tBB poseen todas las propiedades geométricas mencionadas para los cB. En particular, tomando u=0 en (C.3) se obtiene:

$$\mathbf{Q}\,(0, v, 1\text{-}v) = \mathbf{\Gamma}\,(v) = \sum_{j=0}^{m} \mathbf{P}_{0\,j\,m\text{-}j}\, \mathrm{B}_j^m(v), \quad \text{con } 0 \le v \le 1, \tag{C.4}$$

que es la expresión del borde inferior (únicamente determinada, como se ve en (C.4), por los puntos de control situados sobre ese borde en el grafo de control).

Pero además los parches triangulares tienen la ventaja, sobre los cuadrados, de adaptarse mejor a un mayor número de superficies, por requerir un mallado triangular para los grafos de control, en lugar de imponer uno rectangular. El mallado rectangular, que es más rígido ya de por sí, no permite lo que se conoce por *degeneración de los bordes* del parche, que los tBB toleran de manera natural, y que consiste en reducir el grado de uno o más bordes. Usualmente, se degeneran todos ellos en un grado, lo que da lugar a grafos de control con bordes de m puntos, siendo m el grado del parche original: en este aspecto, la degeneración equivale a una pérdida de tres puntos de control, o (más gráficamente) a la sustitución de los dos primeros puntos de cada borde por uno solo. La figura C.3 muestra un ejemplo de triángulo cuártico de Bernstein-Bézier degenerado (t4BBd), que resulta de forzar la pérdida de un grado sobre cada uno de los bordes del de la figura C.2.

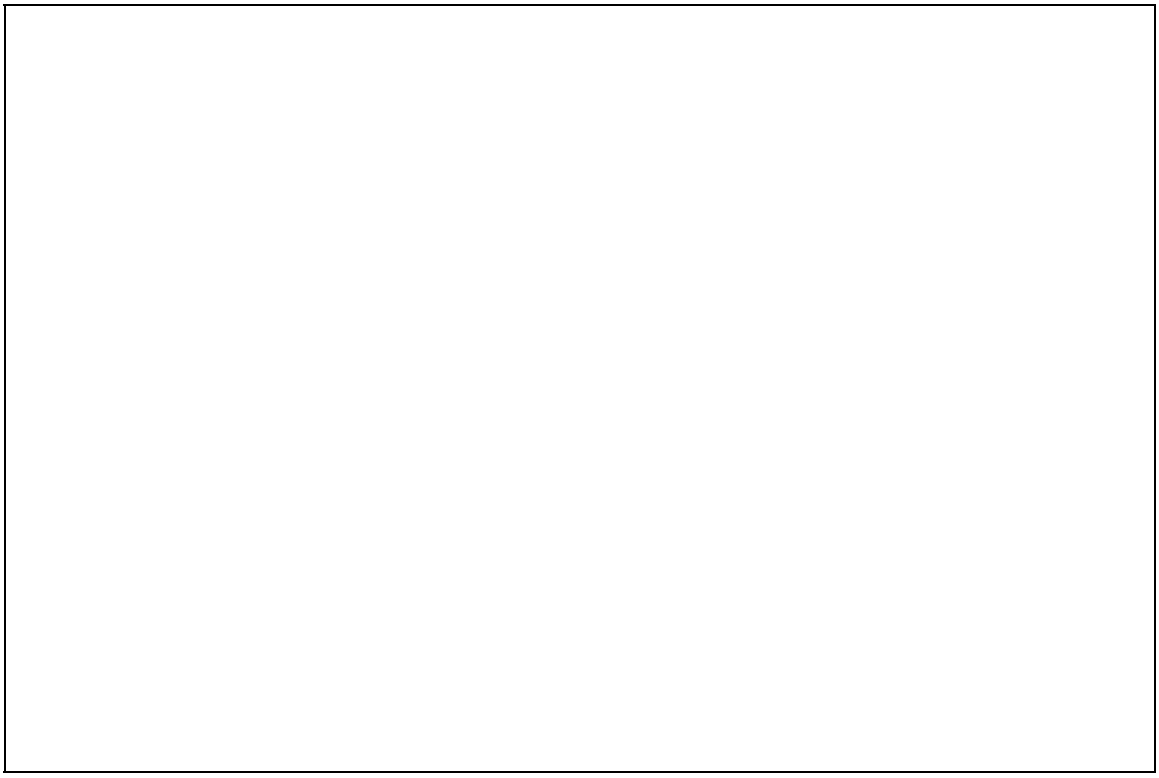

figura C.3 - T4BBd: dominio paramétrico y grafo de control.
(compárese con el t4BB de la figura C.2)

Esta degeneración de los bordes introduce un mayor grado de libertad a la hora de empalmar unos parches con otros para modelar una superficie compleja proporcionando un resultado globalmente continuo. Aunque una discusión sobre los diferentes tipos de continuidad definidos para las superficies paramétricas cae fuera del alcance de este anejo[\], sí cabe precisar que lo más comunmente exigido en las aplicaciones de CAD es que la interconexión de los parches respete la continuidad geométrica de primer orden, conocida como *G1-continuidad*. Es decir, que el plano tangente (y el vector normal) a la superficie evolucione de manera continua al moverse por ella. Y para lograrlo, una vez fijado el borde que separará a dos parches adyacentes, es necesario mover sus respectivos puntos de control interiores. Como se puede comprobar comparando las figuras C.2 y C.3, en el caso del t4BB hay sólo dos puntos de control interiores, en la fila más próxima a un borde, para los tres del borde (excluyendo del razonamiento a los dos de las esquinas); sin embargo, en el caso del t4BBd, hay dos para dos, lo que facilita la resolución de las restricciones impuestas por la G1-continuidad (aunque, eso sí, se pierde un grado en el orden de los bordes).

C.3. Triángulos de Gregory

C.3.1. Definición

Llegando más lejos en la búsqueda de flexibilidad en cuanto al empalme de los parches para garantizar un resultado globalmente continuo, se definen los triángulos de Gregory, de los que, en el caso de este Proyecto, interesan en particular los cuárticos (t4G). Su grafo de control tiene quince puntos que, como se ve en la figura C.4, resultan de duplicar cada uno de los puntos interiores de un t4BBd para poder asociar puntos interiores a cada borde de manera independiente al resto. El dominio paramétrico proyectado en el espacio 3-D sigue siendo un triángulo definido por $0 \le u, v, w \le 1$ y $u+v+w = 1$, y la expresión de un punto de la superficie del parche en función de sus coordenadas paramétricas (o baricéntricas) y de los puntos de control es la siguiente:

$$\begin{aligned}\mathbf{QG}\,(u, v, w) = {} & u^3\,\mathbf{P}_0 + v^3\,\mathbf{P}_1 + w^3\,\mathbf{P}_2 + 3\,[u^2\,v\,(1\text{-}w)\,\mathbf{P}_{01} + u\,v^2\,(1\text{-}w)\,\mathbf{P}_{02}] + \\ & + 3\,[v^2\,w\,(1\text{-}u)\,\mathbf{P}_{11} + v\,w^2\,(1\text{-}u)\,\mathbf{P}_{12} + w^2\,u\,(1\text{-}v)\,\mathbf{P}_{21} + w\,u^2\,(1\text{-}v)\,\mathbf{P}_{22}] + \\ & + 12\,[u^2\,v\,w\,\mathbf{P}_{211} + u\,v^2\,w\,\mathbf{P}_{121} + u\,v\,w^2\,\mathbf{P}_{112}],\end{aligned} \qquad \text{(C.5)}$$

donde $\mathbf{P}_{211} = \dfrac{w\,\mathbf{P}^{v}_{211} + v\,\mathbf{P}^{w}_{211}}{w+v}$, $\mathbf{P}_{121} = \dfrac{u\,\mathbf{P}^{w}_{121} + w\,\mathbf{P}^{u}_{121}}{u+w}$, y $\mathbf{P}_{112} = \dfrac{v\,\mathbf{P}^{u}_{112} + u\,\mathbf{P}^{v}_{112}}{v+u}$.

[\] Un estudio exhaustivo sobre las restricciones impuestas por los distintos tipos de continuidad al problema de la conexión de parches alabeados de cada una de las tres familias mencionadas aquí, para reconstruir superficies muestreadas, puede encontrarse en [DU-88].

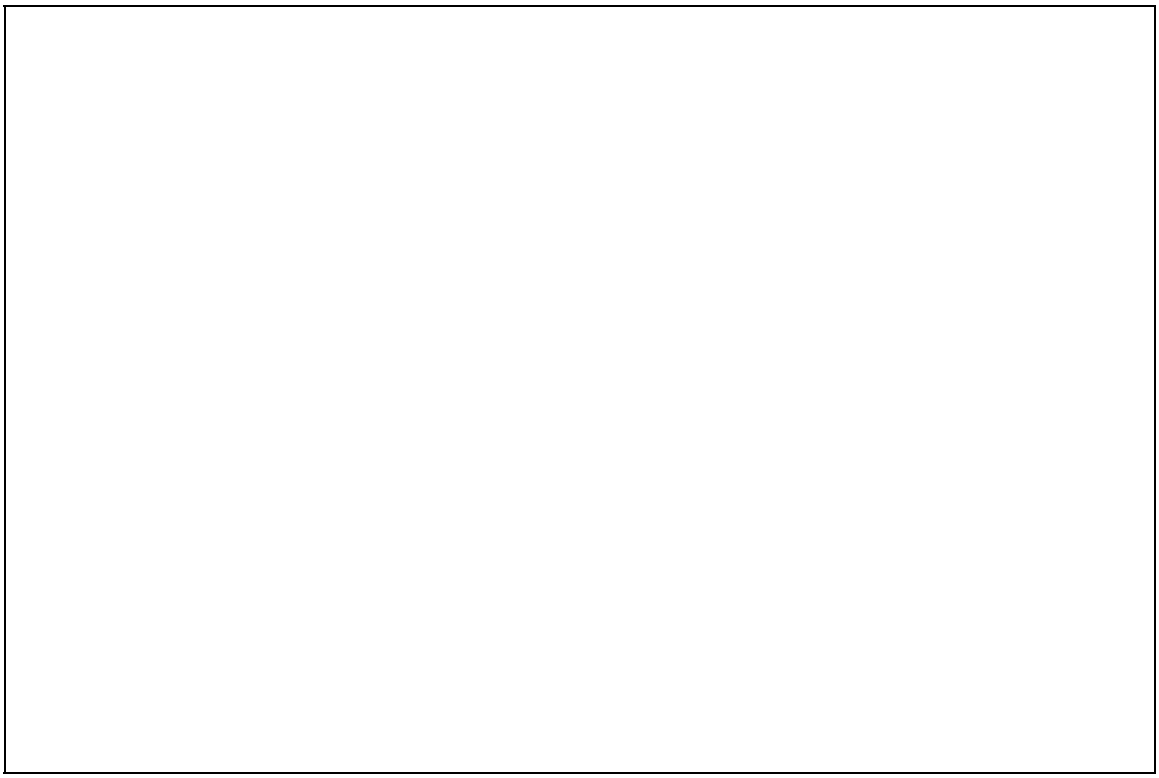

figura C.4 - T4G: dominio paramétrico y grafo de control.

C.3.2. Propiedades geométricas

Los t4G presentan las propiedades descritas para los parches anteriores: en concreto, la de la *envoltura convexa*, y la de que un borde está definido únicamente por los puntos del borde correspondiente en el grafo de control. Por ejemplo, de tomar u=0 en (C.5), resulta:

$$\mathbf{QG}\,(0, v, 1\text{-}v) = \mathbf{\Gamma}\,(v) = v^3\,\mathbf{P}_1 + 3\,v^2\,(1\text{-}v)\,\mathbf{P}_{11} + 3\,v\,(1\text{-}v)^2\,\mathbf{P}_{12} + (1\text{-}v)^3\,\mathbf{P}_2\,, \qquad \text{(C.6)}$$

que es la expresión de la curva de Bézier de tercer grado correspondiente al borde inferior, y que no depende más que de $\mathbf{P}_1$, $\mathbf{P}_{11}$, $\mathbf{P}_{12}$, y $\mathbf{P}_2$ (y v, claro).

Pero es que además, al estar "dislocadas" las articulaciones del grafo de control en los puntos de control interiores $\mathbf{P}_{112}$, $\mathbf{P}_{121}$, y $\mathbf{P}_{211}$ (que se definen por comodidad en la notación en (C.5), pero que no son fijos como en el caso de un tBB o un tBBd, sino que dependen de las coordenadas baricéntricas), resulta mucho más fácil aproximar una superficie dada al mismo tiempo que se produce un conjunto de parches globalmente *suave*.

Anejo D. TRIANGLE: DESCRIPCIÓN DE FUNCIONES

En este anejo se describen todos los procedimientos y funciones del programa Triangle, en su versión del 20 de diciembre de 1990 (última correspondiente a este Proyecto).

D.1. Programa principal: `Program Convertisseur (Input, Output)`

Presenta la pantalla de diálogo, asignando a las distintas variables (parámetros de cámara y límites de tolerancia para la subdivisión de los parches) sus valores por defecto. Para ello, llama a `Ecran_Dialogue` y `Initialisation_Dialogue`. Luego entra en la fase de diálogo, en la que el usuario puede modificar esos valores, invocando a la función `Commande`, y si ésta devuelve `Executer`, comienza la facetización de la superficie: se calcula la matriz de cambio de base (`Calcule_Matrice_de_Passage`) y se llama a `Metamorphose`. Después, se ofrece al usuario la posibilidad de visualizar el resultado en *wire-frame*, siempre que disponga de una pantalla gráfica adecuada (`Affiche`).

En todas las cabeceras de funciones y procedimientos listados a continuación, que son de ámbito global salvo cuando se indica lo contrario, se han abreviado como sigue los siguientes tipos estándar de Pascal: `Int(eger)`; `Str(ing)`; `Bool(ean)`. Además, se alude a los tipos y variables globales relacionados en el listado D.1.

```
Type Vecteur = Array [1..3] of Real; {−∞ < x,y,z < +∞}
     Param   = Array [0..2] of Real; {0 ≤ u,v,w ≤ 1}
     Matrice = Array [1..3] of Vecteur;

     Sommet = ^Structure_sommet;
     Structure_sommet = Record
                          Precedent,          {Lista de vértices    }
                          Suivant:   Sommet;  {doblemente encadenada}
                          Place:     Int;     {Número en la lista}
                          Coord,              {x,y,z}
                          Normale:   Vecteur;
                          Norm_OK:   Bool;
                          Bary:      Param;   {u,v,w}
                          Scalaire:  Real;    {Coord.Normale}
                        End;

     Triangle = ^Structure_triangle;
     Structure_triangle = Record
                            Suivant:    Triangle;
                            Point:      Array [1..3] of Sommet;
                            Tordu,
                            Silhouette,
                            Frontiere:  Array [0..2] of Bool;
                            Compteur_S,
                            Compteur_F: Array [0..2] of Int;
                          End;

     Facette = ^Structure_facette;
     Structure_facette = Record
                           Suivante: Facette;
                           Coin:     Array [1..3] of Sommet;
                         End;

     Action = (Executer, Visualiser, Continuer, Quitter);

Var Pile_des_Triangles: Triangle;
    Pile_des_Facettes:  Facette;
    Liste_des_Sommets:  Sommet;

    P0, P1, P2, P01, P02, P11, P12, P21, P22,
    P211v, P211w, P121w, P121u, P112u, P112v: Vecteur;

    Passage_Oeil: Matrice;

    Tolerance_Rectitude,
    Tolerance_Platitude,
    Supplement_Silhouette,
    Max_Frontiere:         Int;   {valores pedidos por el usuario}
    Epsilon_Rectitude,
    Epsilon_Platitude:     Real;  {valores manejados internamente}
```

listado D.1 - Resumen de tipos y variables globales de Triangle.

D.2. Gestión de la pantalla de diálogo con el usuario

D.2.1. `Function Sans_Espace (texte:Str): Str`

Devuelve `texte` sin los espacios que la rodean.

D.2.2. `Function Blancs (n:Int;attribut:Str): Str`

Imprime una cadena de `n` blancos en el modo `attribut`, y posiciona el cursor al principio de la cadena.

D.2.3. `Function GoToXY (x,y:Int): Str`

Posiciona el cursor en las coordenadas (`x`, `y`), siendo (1, 1) la esquina superior izda.

D.2.4. `Function Centre (x,y:Int;texte,attribut:Str): Str`

Imprime `texte` en el modo `attribut`, y centrado alrededor de (`x`, `y`).

D.2.5. `Function Droite (x,y:Int;texte,attribut:Str): Str`

Imprime `texte` en el modo `attribut`, y justificado a la derecha sobre (`x`, `y`).

D.2.6. `Function Aff_I (i:Int): Str`

Devuelve una cadena de caracteres con el valor de `i` (formato entero).

D.2.7. `Function Aff_R (r:Real): Str`

Devuelve una cadena de caracteres con el valor de `r` (formato real).

D.2.8. `Function Commande: Action`

Pide datos al usuario y los lee en la línea de comandos reservada al efecto en la pantalla de diálogo. Si se trata de un `[ENTER]`, hace que el campo siguiente al actual se convierta en el activo; si es un valor numérico, llama a `Actualise_Valeur_Courante`, que se encarga de verificar su corrección y de modificar el campo activo; si es una "`a`" (*avec*) o una "`s`" (*sans*), activa o desactiva el mecanismo de información en tiempo real sobre la conversión. En todos los casos anteriores, sigue esperando más datos. Sin embargo, si el usuario introduce una "`e`" (*executer*) o una "`q`" (*quitter*), devuelve el control, pasando como resultado de la invocación la acción correspondiente.

D.2.9. `Procedure Ecran_Dialogue`

Presenta la pantalla de diálogo con el usuario.

D.2.10. `Procedure Affiche_Dialogue (n:Int;attribut:Str)`

Imprime el valor del campo nº `n` en el modo `attribut`.

D.2.11. `Procedure Initialisation_Dialogue`

Inicializa las variables del diálogo a sus valores por defecto.

D.2.12. `Procedure Nouvelle_Information (n:Int)`

Imprime el nuevo valor del campo nº `n`.

D.2.13. `Procedure Dialogue_Suivant`

Pasa al siguiente campo del diálogo, convirtiéndolo en el activo para que el usuario pueda modificar su valor.

D.2.14. `Procedure Erreur (texte:Str)`

Imprime el mensaje de error `texte` o borra el anterior (`texte=' '`).

D.2.15. `Procedure Actualise_Valeur_Courante (valeur:Str)`

Reemplaza el valor actual del campo activo por `valeur` (el tecleado por el usuario) tras comprobar su corrección.

D.2.16. `Procedure Information_Conversion`

Imprime los valores variables durante la conversión (números de subdivisiones por las distintas razones, y de triángulos y facetas empilados).

D.3. Funciones matemáticas

D.3.1. `Function Produit_Vectoriel (v1,v2:Vecteur): Vecteur`

Devuelve `v1^v2`.

D.3.2. `Function Produit_Scalaire (v1,v2:Vecteur): Real`

Devuelve `v1.v2`.

D.3.3. `Function Unitaire (v:Vecteur): Vecteur`

Devuelve `v` normalizado.

D.3.4. `Function Signe (r:Real): Int`

Devuelve el signo de `r` (-1, 0, ó +1).

D.3.5. `Function Vecteurs_Egaux (v1,v2:Vecteur): Bool`

Devuelve `True` si `v1` y `v2` tienen todas sus componentes iguales.

D.3.6. `Function Vecteurs_Ordonnes (v1,v2:Vecteur): Bool`

Devuelve `True` si `v1` es estrictamente inferior a `v2` en el orden lexicográfico "x, y, z", i.e. si `v1[x]<v2[x]`, o si (`v1[x]=v2[x] and v1[y]<v2[y]`), o si (`v1[x]=v2[x] and v1[y]=v2[y] and v1[z]<v2[z]`).

D.3.7. `Function Produit_des_Matrices (a,b:Matrice): Matrice`

Devuelve el producto de las matrices `a` y `b` (ambas de 3 x 3).

D.3.8. `Function Produit_Matrice_Colonne (a:Matrice; b:Vecteur): Vecteur`

Devuelve el producto de la matriz `a` (3 x 3) por el vector columna `b` (3).

D.3.9. `Function QG (triplet:Param): Vecteur`

Devuelve las coordenadas espaciales (x, y, z) correspondientes a las paramétricas que contiene `triplet`.

D.3.10. `Function dQG (dir:Int;triplet:Param): Vecteur`

Devuelve el vector derivada parcial respecto a `dir` de QG, en el punto de coordenadas paramétricas `triplet`. El cálculo es aproximado (derivada incremental).

D.3.11. `Procedure Calcule_Normales (t:Triangle)`

Calcula las normales normalizadas en los tres vértices del triángulo apuntado por `t`. Si alguna no es definible, le asigna el vector nulo y lo señala con `Norm_OK:=False`.

También almacena el valor del producto escalar "`Coord.Norm`" en cada vértice, con el fin de saber si la normal apunta o no hacia la cámara, sin necesidad de rehacer los cálculos.

D.4. Cálculo de perspectiva y visualización

D.4.1. `Procedure Calcule_Matrice_de_Passage`

Calcula la matriz `Passage_Oeil`, que permite pasar las coordenadas originales de los puntos de control al referencial del "ojo" descrito en Ripolin:

- origen (O) = cámara;
- eje (Oz) apuntando hacia el punto de mira;
- eje (Ox) paralelo al plano (xOy) original (salvo si Roulis <> 0);
- eje (Oy) definido de manera que {**x**, **y**, **z**} sea un triedro directo.

También calcula los límites de tolerancia manejados internamente por el programa (`Epsilon_Rectitude` y `Epsilon_Platitude`), a partir de los especificados por el usuario (`Tolerance_Rectitude` y `Tolerance_Platitude`) durante la fase previa de diálogo.

D.4.2. `Procedure Passage_au_Repere_de_l_Oeil`

Pasa las coordenadas originales de los puntos de control del parche inicial al referencial ligado a la cámara, utilizando la matriz `Passage_Oeil`.

D.4.3. `Procedure Affiche (signes:Bool)`

Imprime en una pantalla gráfica las facetas obtenidas tras completarse la subdivisión, sin efectuar *clipping* en los planos delantero y trasero. Si `signes=True`, muestra junto a cada vértice el signo de su `Scalaire` asociado (i.e. "-" si la normal en ese vértice apunta hacia la cámara, y "+" en caso contrario). NB: Al proyectar los puntos, destruye sus coordenadas x e y.

D.5. Gestión de E/S de ficheros

D.5.1. `Procedure Determine (Var nb_lignes:Int)`

Determina, llamando a la rutina `FRMT` del Labo. IMA, el número de líneas (o sea, de parches) del fichero de entrada que contiene los triángulos de Gregory que deben ser facetizados.

D.5.2. `Procedure Lit_Patch`

Carga las coordenadas espaciales de los 15 puntos de control del parche inicial.

D.5.3. `Procedure Ecrit_Resultats`

Escribe en los dos ficheros de resultados correspondientes al objeto (de coordenadas de vértices y componentes de normales, y de facetas). Para ello, primero numera los vértices de la lista a la que apunta `Liste_des_Sommets`, que se supone completa para el parche en curso. Si es la primera llamada, crea estos ficheros; si no, añade la información al final de cada uno.

D.5.4. `Procedure Ecrit_Obs`

Salva los parámetros de observación. NB: todas las coordenadas y ángulos expresados en el referencial ligado a la cámara.

D.6. Gestión de la memoria

D.6.1. `Function Cherche_ou_Cree_Sommet (pt:Vecteur; triplet:Param): Sommet`

Busca el punto de coordenadas espaciales `pt` en `Liste_des_Sommets`, que está ordenada según lo explicado en `Vecteurs_Ordonnes`. Si lo encuentra, devuelve un puntero sobre él; si no, lo inserta en la lista en el lugar que corresponda, inicializa su normal al vector nulo, y devuelve un puntero sobre él.

D.6.2. `Procedure Initialisation_des_Donnees`

Inicializa las pilas de triángulos y facetas, y la lista de vértices antes de comenzar el tratamiento de cada parche del fichero de entrada: vacía la pila de facetas (`Nil`), mete el parche inicial en la de triángulos, y forma la lista de vértices con los tres de las esquinas del parche inicial recién leído, llamando a `Cherche_Ou_Cree_Sommet` (para que queden ordenados).

D.6.3. `Procedure Push_Triangle (t:Triangle)`

Mete el triángulo apuntado por `t` en lo alto de `Pile_des_Triangles`.

D.6.4. `Procedure Pop_Triangle (Var t:Triangle)`

Saca el triángulo superior de `Pile_des_Triangles` y lo devuelve, apuntado por `t`.

D.6.5. `Procedure Transfert_Triangle_Facette (t:Triangle)`

Empila en lo alto de `Pile_des_Facettes` la faceta plana cuyos vértices coinciden con las esquinas del parche apuntado por `t`.

D.6.6. `Procedure Libere_Espace`

Vacía la lista de vértices y la pila de facetas.

D.7. Subdivisión: *tests* y acción

D.7.1. `Procedure Metamorphose`

Llama a `Determine` para averiguar el número de parches que componen la superficie y, para cada uno de ellos, hace lo siguiente:

1. lee las coordenadas de los 15 puntos de control, mediante `Lit_Patch`;

2. las pasa al referencial ligado a la cámara, llamando a `Passage_au_Repere_de_l_Oeil`;
3. inicializa las pilas (`Initialisation_des_Donnees`);
4. mientras haya triángulos empilados,
   a) saca el primero de la pila (`Pop_Triangle`);
   b) calcula las coordenadas de los puntos medios `Exact[c]` (sobre el segmento curvilíneo) y `Approx[c]` (sobre el rectilíneo) de cada borde `c`;
   c) somete al triángulo, por orden, a los cuatro *tests*: `Cotes_Tordus`, `Cotes_Silhouette`, `Cotes_Frontiere`, `Triangle_Gauche`;
   d) si el triángulo los pasa todos, se puede aproximar por una faceta plana con `Transfert_Triangle_Facette`; si no, es fragmentado por `Subdivise`; en todo caso, se puede liberar el espacio que ocupaba en memoria;
   e) si el usuario había pedido información en tiempo real sobre la conversión, se le facilita en este momento;
5. cuando se vacía la pila de triángulos, añade los resultados correspondientes al último parche leído en los ficheros de vértices y normales, y de facetas, llamando a `Ecrit_Resultats`.

Una vez facetizados todos los parches de la superficie original, se salvan los parámetros de observación. Los tres ficheros de resultados son directamente legibles por Ripolin.

Las funciones que se listan a continuación (apartados D.7.3 a D.7.5) son todas locales a `Metamorphose`, como el procedimiento `Subdivise` (D.7.6), y el tipo:

```
Type Raison_Subdiv = (Aucune, Rect, Silh, Front, Plat);
```

D.7.2. `Function Cotes_Tordus (t:Triangle): Bool`

Devuelve `True` si alguno de los bordes del triángulo `t`, debiendo pasar el *test* de rectitud, no lo satisface. Almacena en `Tordu[c]` el resultado de ese *test* para cada borde `c` que deba pasarlo.

D.7.3. `Function Cotes_Silhouette (t:Triangle): Bool`

Devuelve `True` si alguno de los bordes del triángulo `t` cruza la silueta del objeto en la pantalla (las normales en sus dos extremos no están orientadas de la misma manera respecto a la cámara), y no ha sido aún subdividido `Supplement_Silhouette` veces suplementarias por esta razón. Almacena en `Silhouette[c]` el resultado de ese *test* para cada borde `c` que deba pasarlo.

D.7.4. `Function Cotes_Frontiere (t:Triangle): Bool`

Devuelve `True` si alguno de los bordes del triángulo `t` pertenece a la frontera del parche inicial, y no ha sido aún subdividido `Max_Frontiere` veces en total (por una razón cualquiera). Almacena en `Frontiere[c]` el resultado de ese *test* para cada borde `c` que deba pasarlo.

D.7.5. `Function Triangle_Gauche (t:Triangle): Bool`

Devuelve `True` si el triángulo `t` no es suficientemente plano.

D.7.6. `Procedure Subdivise (t:Triangle; raison:Raison_Subdiv)`

Subdivide el triángulo padre `t` en cuatro hijos: tres *legítimos* y uno *ilegítimo* (el que aparece en el interior, invertido), que son empilados en `Pile_des_Triangles`. Para los hijos legítimos, la memoria de los dos bordes antiguos es heredada del padre, y la del borde nuevo inicializada; para el hijo ilegítimo, todo es inicializado puesto que sus tres bordes son nuevos. En concreto, los valores iniciales para la memoria de los bordes nuevos son los siguientes:

```
Tordu := (Tolerance_Rectitude>0)
         Or (raison=Plat);

Silhouette := (Supplement_Silhouette>0);

Compteur_S := <máximo de los Compteur_S de los bordes del padre>;

Frontiere := False;

Compteur_F := 0;   {en realidad, no importa si Frontiere=False}
```

El punto medio de cada borde `c` se toma sobre el segmento curvilíneo (`Exact[c]`) si (`Tordu[c] Or Silhouette[c] Or Frontiere[c]`), es decir, siempre que no haya sido otro borde, o el interior del triángulo (`raison=Plat`), el causante de la subdivisión. En cualquiera de estos dos últimos casos, el punto medio se toma sobre el segmento rectilíneo (`Approx[c]`), puesto que el borde `c` había sido ya dado por bueno.

Bibliografía

[BARNHILL-77] R. E. Barnhill: *Representation and Approximation of Surfaces*.
Academic Press, 1977.

[BÉZIER-86] P. E. Bézier: *Courbes et Surfaces*.
Mathématiques et CAO, vol. 4. Hermès Publishing, 1986.

[BURGSTAHLER-89] Éric Burgstahler: *Subdivision adaptative de carreaux de Bézier en facettes planes*.
Informe de Proyecto realizado en el Depto. IMA de la ENST de París, 1989.

[CARPENTER-84] Loren Carpenter: *The A-buffer, an Antialiased Hidden Surface Method*.
Computer Graphics, vol. 18, nr. 3, 1984.

[CLAY-88] Reed D. Clay and Henry P. Moreton: *Efficient Adaptive Subdivision of Bézier Surfaces*.
Elsevier Science Publishers B.V., 1988.

[DU-88] Wen-Hui Du: *Étude sur la représentation de surfaces complexes: application à la reconstruction de surfaces échantillonnées*.
Tesis Doctoral presentada en la ENST de París, 1988.

[DU-88] Wen-Hui Du: *On the G1 Smooth Connection between Triangular Bernstein-Bézier Patches*.
ENST de París, 1989.

[FAUX-79] I. D. Faux and M. J. Pratt: *Computational Geometry for Design and Manufacture*.

Ellis Horwood Ltd., 1979.

[NOSMAS-89] Patrick Nosmas: *Restructuration d'un logiciel de synthèse d'image 3D:* Ripolin.

Informe de Proyecto realizado en la ENSAD para el Depto. IMA de la ENST de París, 1989.

[SAIR-88] F. Sair: *Triangulation adaptative de surfaces échantillonnées*.

Informe de PFC realizado en Paris VI, 1988.

[SHIRMANN-89] Leon A. Shirmann and Carlo H. Séquin: *Local Surface Interpolation with Shape Parameters between Adjoining Gregory Patches*.

Computer Aided Geometric Design , nr. 7, 1989.

Pliego de condiciones

Este documento contiene las condiciones legales que guiarán la realización, en este Proyecto, de procedimientos de visualización de objetos reales, modelados por recubrimientos superficiales aproximados.

En lo que sigue, se supondrá que el Proyecto ha sido encargado por una empresa cliente a una empresa consultora con la finalidad de realizar dichos procedimientos de visualización. Dicha empresa ha debido desarrollar una línea de investigación para elaborar el Proyecto. Esta línea de investigación, junto con el posterior desarrollo de los programas está amparada por las condiciones particulares del siguiente Pliego. Supuesto que la utilización industrial de los métodos recogidos en el presente Proyecto ha sido decidida por parte de la empresa cliente o de otras, la obra a realizar se regulará por las siguientes

- Condiciones generales:

1. La modalidad de contratación será el concurso. La adjudicación se hará, por tanto, a la proposición más favorable sin atender exclusivamente al valor económico, dependiendo de las mayores garantías ofrecidas. La empresa que somete el Proyecto a concurso se reserva el derecho a declararlo desierto.
2. El montaje y mecanización completa de los equipos que intervengan será realizado totalmente por la empresa licitadora.
3. En la oferta, se hará constar el precio total por el que se compromete a realizar la obra y el tanto por ciento de baja que supone este precio en relación con un importe límite si éste se hubiera fijado.
4. La obra se realizará bajo la dirección técnica de un Ingeniero Superior de Telecomunicación, auxiliado por el número de Ingenieros Técnicos y Programadores que se estime preciso para el desarrollo de la misma.
5. Aparte del Ingeniero Director, el contratista tendrá derecho a contratar al resto del personal, pudiendo ceder esta prerrogativa a favor del Ingeniero Director, quien no estará obligado a aceptarla.
6. El contratista tiene derecho a sacar copias a su costa de los Planos, Pliego de condiciones y Presupuestos. El Ingeniero autor del Proyecto autorizará con su firma las copias solicitadas por el contratista después de confrontarlas.
7. Se abonará al contratista la obra que realmente ejecute con sujeción al Proyecto que sirvió de base para la contratación, a las modificaciones autorizadas por la superioridad o a las órdenes que con arreglo a sus facultades le hayan comunicado por escrito al Ingeniero Director de obras siempre que dicha obra

se haya ajustado a los preceptos de los Pliegos de Condiciones, con arreglo a los cuales, se harán las modificaciones y la valoración de las diversas unidades sin que el importe total pueda exceder de los presupuestos aprobados. Por consiguiente, el número de unidades que se consignan en el Proyecto o en el Presupuesto, no podrá servirle de fundamento para entablar reclamaciones de ninguna clase, salvo en los casos de rescisión.

8. Tanto en las certificaciones de obras como en la liquidación final, se abonarán los trabajos realizados por el contratista a los precios de ejecución material que figuran en el Presupuesto para cada unidad de la obra.

9. Si excepcionalmente se hubiera ejecutado algún trabajo que no se ajustase a las condiciones de la contrata pero que sin embargo es admisible a juicio del Ingeniero Director de obras, se dará conocimiento a la Dirección, proponiendo a la vez la rebaja de precios que el Ingeniero estime justa y si la Dirección resolviera aceptar la obra, quedará el contratista obligado a conformarse con la rebaja acordada.

10. Cuando se juzgue necesario emplear materiales o ejecutar obras que no figuren en el presupuesto de la contrata, se evaluará su importe a los precios asignados a otras obras o materiales análogos si los hubiere y cuando no, se discutirán entre el Ingeniero Director y el contratista, sometiéndolos a la aprobación de la Dirección. Los nuevos precios convenidos por uno u otro procedimiento, se sujetarán siempre al establecido en el punto anterior.

11. Cuando el contratista, con autorización del Ingeniero Director de obras, emplee materiales de calidad más elevada o de mayores dimensiones de lo estipulado en el Proyecto, o sustituya una clase de fabricación por otra que tenga asignado mayor precio o ejecute con mayores dimensiones cualquier otra parte de las obras, o en general, introduzca en ellas cualquier modificación que sea beneficiosa a juicio del Ingeniero Director de obras, no tendrá derecho, sin embargo, sino a lo que le correspondería si hubiera realizado la obra con estricta sujeción a lo proyectado y contratado.

12. Las cantidades calculadas para obras accesorias, aunque figuren por partida alzada en el Presupuesto final (general), no serán abonadas sino a los precios de la contrata, según las condiciones de la misma y los Proyectos particulares que para ellas se formen, o en su defecto, por lo que resulte de su medición final.

13. El contratista queda obligado a abonar al Ingeniero autor del Proyecto y director de obras así como a los Ingenieros Técnicos, el importe de sus respectivos honorarios facultativos por formación del Proyecto, dirección técnica y administración en su caso, con arreglo a las tarifas y honorarios vigentes.

14. Concluida la ejecución de la obra, será reconocida por el Ingeniero Director que a tal efecto designe la empresa.

15. La garantía definitiva será del 4% del presupuesto y la provisional del 2%.

16. La forma de pago será por certificaciones mensuales de la obra ejecutada, de acuerdo con los precios del presupuesto, deducida la baja si la hubiera.
17. La fecha de comienzo de las obras será a partir de los 15 días naturales del replanteo oficial de las mismas y la definitiva, al año de haber ejecutado la provisional, procediéndose si no existe reclamación alguna, a la reclamación de la fianza.
18. Si el contratista al efectuar el replanteo, observase algún error en el Proyecto, deberá comunicarlo en el plazo de quince días al Ingeniero Director de obras, pues transcurrido ese plazo será responsable de la exactitud del Proyecto.
19. El contratista está obligado a designar una persona responsable que se entenderá con el Ingeniero Director de obras, o con el delegado que éste designe, para todo relacionado con ella. Al ser el Ingeniero Director de obras el que interpreta el Proyecto, el contratista deberá consultarle cualquier duda que surja en su realización.
20. Durante la realización de la obra, se girarán visitas de inspección por personal facultativo de la empresa cliente, para hacer las comprobaciones que se crean oportunas. Es obligación del contratista, la conservación de la obra ya ejecutada hasta la recepción de la misma, por lo que el deterioro parcial o total de ella, aunque sea por agentes atmosféricos u otras causas, deberá ser reparado o reconstruido por su cuenta.
21. El contratista, deberá realizar la obra en el plazo mencionado a partir de la fecha del contrato, incurriendo en multa por retraso de la ejecución, siempre que éste no sea debido a causas de fuerza mayor. A la terminación de la obra, se hará una recepción provisional previo reconocimiento y examen por la dirección técnica, el depositario de efectos, el interventor y el jefe de servicio o un representante, estampando su conformidad el contratista.
22. Hecha la recepción provisional, se certificará al contratista el resto de la obra, reservándose la administración el importe de los gastos de conservación de la misma hasta su recepción definitiva y la fianza durante el tiempo señalado como plazo de garantía. La recepción definitiva se hará en las mismas condiciones que la provisional, extendiéndose el acta correspondiente. El Director Técnico propondrá a la Junta Económica la devolución de la fianza al contratista de acuerdo con las condiciones económicas legales establecidas.
23. Las tarifas para la determinación de honorarios, reguladas por orden de la Presidencia del Gobierno el 19 de Octubre de 1961, se aplicarán sobre el denominado en la actualidad "Presupuesto de Ejecución de Contrata" y anteriormente llamado "Presupuesto de Ejecución Material", que hoy designa otro concepto.

• Condiciones particulares

La empresa consultora, que ha desarrollado el presente Proyecto, lo entregará a la empresa cliente bajo las condiciones generales ya formuladas, debiéndo añadirse las siguientes condiciones particulares:

1. La propiedad intelectual de los procesos descritos y analizados en el presente trabajo pertenece por entero a la empresa consultora representada por el Ingeniero Director del Proyecto.
2. La empresa consultora se reserva el derecho a la utilización total o parcial de los resultados de la investigación realizada para desarrollar el siguiente Proyecto, bien para su publicación, o bien para su uso en trabajos o Proyectos posteriores, para la misma empresa cliente o para otra.
3. Cualquier tipo de reproducción aparte de las reseñadas en las condiciones generales, bien sea para uso particular de la empresa cliente, o para cualquier otra aplicación, contará con autorización expresa y por escrito del Ingeniero Director del Proyecto, que actuará en representación de la empresa consultora.
4. En la autorización se ha de hacer constar la aplicación a que se destinan sus reproducciones así como su cantidad.
5. En todas las reproducciones se indicará su procedencia, explicitando el nombre del Proyecto, nombre del Ingeniero Director y de la empresa consultora.
6. Si el Proyecto pasa la etapa de desarrollo, cualquier modificación que se realice sobre él, deberá ser notificada al Ingeniero Director del Proyecto y, a criterio de éste, la empresa consultora decidirá aceptar o no la modificación propuesta.
7. Si la modificación se acepta, la empresa consultora se hará responsable al mismo nivel que el Proyecto inicial del que resulta el añadirla.
8. Si la modificación no es aceptada, por el contrario, la empresa consultora declinará toda responsabilidad que se derive de la aplicación o influencia de la misma.
9. Si la empresa cliente decide desarrollar industrialmente uno o varios productos en los que resulte parcial o totalmente aplicable el estudio de este Proyecto, deberá comunicarlo a la empresa consultora.
10. La empresa consultora no se responsabiliza de los efectos laterales que se puedan producir en el momento en que se utilice la herramienta objeto del presente Proyecto para la realización de otras aplicaciones.
11. La empresa consultora tendrá prioridad respecto a otras en la elaboración de los Proyectos auxiliares que fuese necesario desarrollar para dicha aplicación industrial, siempre que no haga explícita renuncia a este hecho. En este caso, deberá autorizar expresamente los Proyectos presentados por otros.

12. El Ingeniero Director del presente Proyecto será el responsable de la dirección de la aplicación industrial siempre que la empresa consultora lo estime oportuno. En caso contrario, la persona designada deberá contar con la autorización del mismo, quien delegará en él las responsabilidades que ostente.

Presupuesto

- Ejecución Material
  - Alquiler de terminal y tiempo de CPU de DEC-VAX (5 meses) ..... 750.000 ptas.
  - Alquiler de monitor gráfico de alta resolución (60 horas) ..... 90.000 ptas.
  - Total Ejecución Material..... 840.000 ptas.
- Gastos Generales
  - 16 % sobre Ejecución Material ..... 134.400 ptas.
- Beneficio Industrial
  - 6 % sobre Ejecución Material ..... 50.400 ptas.
- Honorarios Proyecto
  - 1.200 horas a 4.000 ptas/hora..... 4.800.000 ptas.
- Material Fungible
  - Gastos de impresión..... 5.000 ptas.
  - Encuadernación..... 12.000 ptas.
  - Fotografías ..... 10.000 ptas.
  - Soporte magnético..... 12.000 ptas.
  - Total Material Fungible ..... 39.000 ptas.
- **Subtotal Presupuesto**..... **5.863.800 ptas.**
- IVA aplicable
  - 13 % sobre Subtotal Presupuesto..... 762.294 ptas.
- **Total Presupuesto**..... **6.626.094 ptas.**

Madrid, a 13 de junio de 1992.

El Ingeniero Jefe del Proyecto,

Fdo.: Francisco Morán Burgos
Ingeniero Superior de Telecomunicación.