Ilustración automática de elementos arqueológicos

Alumna: **Beatriz Ramos Ontiveros**
Tutor: Francisco Javier Melero Rus
Curso: 5º Ingeniería Informática Superior
E-Mail: ekihan@hotmail.com
DNI: 75146953 – L

Índice de Contenidos

Proyecto Informático

5º Ingeniería Informática

Planificación inicial

Curso 2006 – 2007

I. Introducción

Actualmente, se continúa realizando la documentación de los yacimientos y elementos arqueológicos encontrados a mano, gracias al esfuerzo de los artistas. Estos artistas, se encargan de, mediante distintas técnicas de dibujado, obtener las imágenes 2D pertinentes para poder acotarlas para estudios posteriores. A esto es a lo que se le llama dibujo arqueológico.

El uso de los escáneres 3D y las técnicas de ilustración no realista, pueden facilitar notablemente esta tarea. Mediante este proyecto se pretende automatizar la obtención del dibujo arqueológico permitiendo además el uso de las diferentes técnicas que se pueden aplicar en el caso actual de dibujado a mano. De esta forma, se puede facilitar enormemente el trabajo de los arqueólogos, ya que pueden obtener los mismos resultados en menos tiempo y sin necesidad de la intervención directa de un grupo de artistas especializados. Se permitirá también la acotación del modelo para facilitar posteriores estudios.

Además se pretende que la aplicación resultante sea útil para la visualización en 3D del modelo, así como de ofrecer la posibilidad de cambiar la textura de sombreado, color, etc. de la misma con el objeto de facilitar diferentes pruebas.

II. Declaración del alcance

2.1. Rendimiento

La eficiencia y la velocidad serán factores cruciales en esta aplicación, ya que debemos tener en cuenta que, normalmente, se trabajará con modelos bastante grandes y que, por lo tanto, ralentizarán el programa. Actualmente, aunque los escáneres 3D son muy rápidos obteniendo los modelos, el postprocesamiento de los mismos es bastante lento. Esto es un aliciente más para comprender la necesidad de velocidad y eficiencia en la aplicación, ya que, si computacionalmente vamos a tardar más que si lo hacemos todo manualmente, simplemente no valdría la pena.

Por esta razón, nos será útil en la aplicación hacer un buen uso de los recursos de los que dispongamos.

2.2. Interfaces

2.2.1. Interfaz gráfica de usuario

Obviamente, el programa debe tener una interfaz gráfica de usuario que sea amigable e intuitiva, fácil de usar por cualquier persona con pocos conocimientos informáticos.

2.2.2. Entorno de programación

El lenguaje de programación usado para desarrollar la aplicación será principalmente C++, aunque también se usará C. Será necesario usar, por supuesto, OpenGL, y también se hará uso de las librerías QT para facilitar el desarrollo del entorno gráfico. El uso de QGLViewer será imprescindible, y necesitaremos la librería libtiff para cargar las imágenes de las texturas.

III. Recursos

3.1. Hardware y software

3.1.1. Recursos hardware

Este proyecto se realizará principalmente en los dos siguientes ordenadores personales:

- Procesador *Intel Pentium 4 Mobile* 2800 MHz
- 512 MB de memoria RAM
- Disco duro de 40 GB

- Procesador *AMD XP 2000+* 1.67 GHz
- Un módulo de memoria RAM de 256 MB
- Disco duro de 80 GB

3.1.2. Recursos software

Para el desarrollo de la aplicación se hará uso de las siguientes herramientas software:

- Sistema operativo *Windows XP*® de Microsft®.
- Sistema operativo *Linux*, distribución Red Hat 9.
- Procesador de textos *Microsoft Word*®.
- Procesador de textos *Kwrite*.
- Programa de dibujo Paint.
- Compilador *DevCpp*.
- Compilador *gcc*.
- Biblioteca gráfica *OpenGL*.
- Librerías *QT*.
- *QGLViewer*.
- Librería *libtiff*.
- *Valgrind* (para depuración de memoria).

IV. Planificación Temporal

4.1. Descripción de tareas

Las tareas más importantes definidas para la realización de este proyecto han sido las siguientes:

INICIO

1. Especificación de requisitos

2. Planificación
- 2.1 Obtener planificación inicial
  - *2.1.1 Declaración del alcance*
  - *2.1.2 Identificar recursos*
- 2.2 Actualizar planificación

3. Desarrollar modelado de requisitos
- 3.1 Desarrollar modelado funcional
- 3.2 Identificar subsistemas funcionales
- 3.3 Revisión del modelado

4. Análisis
- 4.1 Identificar clases, atributos y relaciones
- 4.2 Obtener diagrama de clases
- 4.3 Revisión

5. Diseño
- 5.1 Diseño del sistema
  - *5.1.1 Identificar objetivos de diseño*
  - *5.1.2 Identificar la arquitectura software*
  - *5.1.3 Obtener la descomposición en subsistemas*
- 5.2 Revisión

6. Implementación
- 6.1 Búsqueda de información necesaria
- 6.2 Estudio de las estructuras de semiaristas aladas
- 6.3 Estudio de las librerías QT, QGLViewer y demás software.
- 6.4 Estudio de distintos métodos de implementación para cada parte del proyecto.
- 6.4 Transformar el modelo de diseño a implementación

7. Pruebas

FIN

Proyecto Informático

5º Ingeniería Informática

Modelado de Requisitos

Curso 2006 – 2007

I. Introducción

Se tratará ahora de recoger toda la información generada durante el *Modelado de Requisitos,* en el que se pretende capturar el propósito del sistema, tratando de identificar y delimitar el mismo, indicando las características, cualidades y restricciones que éste debe satisfacer.

Será necesario realizar los siguientes procesos:

- Desarrollar el modelo funcional
- Identificar los subsistemas funcionales
- Revisar el modelado de requisitos realizado hasta ahora

Para desarrollar el modelo funcional se hará uso de la identificación de los actores y de los casos de uso, que irán acompañados de una especificación detallada de los mismos. Para la identificación de los subsistemas funcionales se hará uso de los casos de uso obtenidos previamente.

II. Sistema propuesto

2.1. Modelo funcional

2.1.1. Definición de actores

Antes de identificar y explicar los distintos Casos de uso que intervendrán en el sistema haremos una breve pausa para ver cuáles son los Actores que forman parte de la aplicación.

En nuestro caso, no es necesario hacer distinciones entre distintos tipos de actores, ya que cualquier usuario de la aplicación podrá acceder a la totalidad del funcionamiento de la misma, luego sólo tendremos un actor al que llamaremos, simplemente, usuario.

2.1.2. Identificación de los casos de uso

Tras realizar un análisis del problema de partida, se han identificado los siguientes casos de uso:

1. Cargar modelo
2. Guardar imagen
3. Cerrar modelo
4. Salir de la aplicación
5. Rotar modelo
6. Trasladar modelo
7. Cambiar color de fondo
8. Cambiar color de los números de las cotas
9. Cambiar color de la cota seleccionada
10. Activar modo cotas
11. Seleccionar cota
12. Crear cota
13. Cambiar propiedades de una cota
14. Eliminar cota
15. Cambiar color cotas
16. Cambiar grosor cotas
17. Activar ocultación automática de cotas
18. Cambiar color del modelo
19. Cambiar escala
20. Activar modo líneas de forma
21. Cambiar color de las líneas de forma
22. Cambiar grosor de las líneas de forma
23. Cambiar estilo de las líneas de forma
24. Activar modo silueta
25. Cambiar color de la silueta

26. Cambiar grosor de la silueta
27. Cambiar estilo de la silueta
28. Activar silueta estricta
29. Activar modo sombreado
30. Cambiar textura para una intensidad
31. Cambiar texturas para el sombreado
32. Calcular sombreado compuacional
33. Recalcular sombreado

2.1.2.1. Caso de uso *"Cargar modelo"*

Nombre del caso	Cargar modelo
Resumen	Un **usuario** quiere cargar en la **aplicación** un **modelo** con **formato ply**.
Dependencias	
Actores	Usuario
Precondiciones	El modelo que se va a cargar debe tener formato ply (ASCII o binario), y debe existir previamente en el **sistema**.
Postcondición	El modelo queda cargado en el **programa**.
Curso normal	1.- El usuario selecciona la **opción** Abrir **archivo**. 2.- Se muestra un **menú** con los **directorios y archivos del sistema**. 3.- Se selecciona el modelo a cargar. 4.- La aplicación carga el modelo elegido.
Cursos Alternativos	3.- Si el **fichero** elegido no tiene formato ply, o tiene algún problema, se mostrará un **mensaje** al usuario comunicándoselo.
Observaciones	
Requisitos no Funcionales Específicos	

2.1.2.2. Caso de uso *"Guardar imagen"*

Nombre del caso	Guardar imagen
Resumen	Un usuario quiere guardar una **captura de la imagen** que ha obtenido en el **visor** de la aplicación.
Dependencias	
Actores	Usuario
Precondiciones	Debe haber un modelo cargado en la aplicación correctamente.
Postcondición	Queda almacenada en el sistema una captura de la imagen que hay en el visor de la aplicación.
Curso normal	1.- El usuario selecciona la opción guardar imagen. 2.- Se muestra un menú con los directorios y archivos del sistema. 3.- Se selecciona el directorio donde queremos almacenar la imagen y se indica el **nombre** que queremos darle a la misma. 4.- La aplicación guarda la captura.
Cursos Alternativos	3.- Si la **carpeta** indicada o cualquiera de los otros **datos** especificados por el usuario son inválidos, la imagen no se guardará y se dará un error.
Observaciones	
Requisitos no Funcionales Específicos	

2.1.2.3. Caso de uso *"Cerrar modelo"*

Nombre del caso	**Cerrar modelo**
Resumen	Un usuario quiere cerrar el modelo cargado previamente.
Dependencias	
Actores	Usuario
Precondiciones	
Postcondición	La aplicación queda con la configuración inicial.
Curso normal	1.- El usuario selecciona la opción cerrar modelo. 2.- La aplicación pone en blanco el visor y la aplicación vuelve a reinstaurar la **configuración** inicial.
Cursos Alternativos	
Observaciones	Si no hay ningún modelo cargado en el sistema, este caso de uso simplemente no hará nada.
Requisitos no Funcionales Específicos	

2.1.2.4. Caso de uso *"Salir de la aplicación"*

Nombre del caso	**Salir de la aplicación**
Resumen	Un usuario quiere salir de la aplicación.
Dependencias	
Actores	Usuario
Precondiciones	La aplicación debe estar abierta.
Postcondición	La aplicación queda finalizada.
Curso normal	1.- El usuario selecciona la opción salir. 2.- La aplicación se cierra.
Cursos Alternativos	
Observaciones	
Requisitos no Funcionales Específicos	

2.1.2.5. Caso de uso *"Rotar modelo"*

Nombre del caso	Rotar modelo
Resumen	El usuario rota el modelo que hay cargado en el sistema interactuando con él con el **ratón**.
Dependencias	
Actores	Usuario
Precondiciones	Debe haber un modelo cargado en la aplicación correctamente.
Postcondición	El modelo queda rotado el **ángulo** deseado por el usuario e indicado por éste mediante el ratón.
Curso normal	1.- El usuario pincha con el **botón izquierdo** sobre el visor. 2.- Sin soltar, el usuario mueve el ratón hacia derecha/izquierda o arriba/abajo para girar el modelo en el **eje** deseado. 3.- El modelo se va rotando, según el movimiento del ratón.
Cursos Alternativos	
Observaciones	
Requisitos no Funcionales Específicos	

2.1.2.6. Caso de uso *"Trasladar modelo"*

Nombre del caso	Trasladar modelo
Resumen	El usuario traslada el modelo que hay cargado en el sistema interactuando con él con el ratón.
Dependencias	
Actores	Usuario
Precondiciones	Debe haber un modelo cargado en la aplicación correctamente.
Postcondición	El modelo queda trasladado según los **deseos del usuario** y lo indicado por éste mediante el ratón.
Curso normal	1.- El usuario pincha con el botón derecho sobre el modelo. 2.- Sin soltar, el usuario mueve el ratón hacia derecha/izquierda o arriba/abajo para trasladar el modelo en la **dirección** deseada. 3.- El modelo se va trasladando, según el movimiento del ratón.
Cursos Alternativos	
Observaciones	
Requisitos no Funcionales Específicos	

2.1.2.7. Caso de uso *"Cambiar color de fondo"*

Nombre del caso	Cambiar color de fondo
Resumen	El usuario modifica el **color** de fondo del visor.
Dependencias	
Actores	Usuario
Precondiciones	
Postcondición	El color de fondo queda modificado.
Curso normal	1.- El usuario indica a la aplicación que quiere cambiar el color de fondo. 2.- Se muestra un menú con los colores disponibles. 3.- El usuario selecciona el color deseado y le da a aceptar. 4.- El color de fondo del visor queda modificado.
Cursos Alternativos	3.- Si el usuario le da a cancelar, el color de fondo no se modifica y el caso de uso finaliza.
Observaciones	
Requisitos no Funcionales Específicos	

2.1.2.8. Caso de uso *"Cambiar color de los números de las cotas"*

Nombre del caso	Cambiar color de los números de las cotas
Resumen	El usuario modifica el color de los números que indican el tamaño de las **cotas**.
Dependencias	
Actores	Usuario
Precondiciones	
Postcondición	El color de los **números de las cotas** queda modificado.
Curso normal	1.- El usuario indica a la aplicación que quiere cambiar el color de los números de las cotas. 2.- Se muestra un menú con los colores disponibles. 3.- El usuario selecciona el color deseado y le da a aceptar. 4.- El color de los números de las cotas queda modificado.
Cursos Alternativos	3.- Si el usuario le da a cancelar, el color no se modifica y el caso de uso finaliza.
Observaciones	
Requisitos no Funcionales Específicos	

2.1.2.9. Caso de uso *"Cambiar color de la cota seleccionada"*

Nombre del caso	**Cambiar color de la cota seleccionada**
Resumen	El usuario modifica el color que toma una cota cuando ha sido seleccionada.
Dependencias	
Actores	Usuario
Precondiciones	
Postcondición	El color para la cota seleccionada queda modificado.
Curso normal	1.- El usuario indica a la aplicación que quiere cambiar el color de la cota seleccionada. 2.- Se muestra un menú con los colores disponibles. 3.- El usuario selecciona el color deseado y le da a aceptar. 4.- El color de la cota seleccionada queda modificado.
Cursos Alternativos	3.- Si el usuario le da a cancelar, el color no se modifica y el caso de uso finaliza.
Observaciones	
Requisitos no Funcionales Específicos	

2.1.2.10. Caso de uso *"Activar modo cotas"*

Nombre del caso	**Activar modo cotas**
Resumen	El usuario activa el **modo** cotas para indicar al programa que quiere trabajar con éstas.
Dependencias	
Actores	Usuario
Precondiciones	
Postcondición	El modo cotas queda activado.
Curso normal	1.- El usuario le da a activar modo cotas. 2.- El modo cotas se activa y se muestra la **ventana** con todas las opciones disponibles para las cotas.
Cursos Alternativos	
Observaciones	
Requisitos no Funcionales Específicos	

2.1.2.11. Caso de uso *"Seleccionar cota"*

Nombre del caso	Seleccionar cota
Resumen	El usuario selecciona una cota.
Dependencias	Caso de uso *"Activar modo cotas"*.
Actores	Usuario
Precondiciones	El modo cotas debe estar activo y deben existir cotas en la aplicación, al menos una.
Postcondición	La cota elegida queda seleccionada.
Curso normal	1.- El usuario le da a seleccionar cota. 2.- El usuario pincha con el ratón sobre la cota que quiera seleccionar. 3.- La cota queda seleccionada.
Cursos Alternativos	2.- Si no se pincha sobre ninguna cota, no se selecciona nada.
Observaciones	
Requisitos no Funcionales Específicos	

2.1.2.12. Caso de uso *"Crear cota"*

Nombre del caso	Crear cota
Resumen	El usuario coloca una cota sobre el modelo 3D.
Dependencias	Caso de uso *"Activar modo cotas"*.
Actores	Usuario
Precondiciones	Debe haber un modelo cargado en la aplicación correctamente y estar activo el modo cotas.
Postcondición	La cota queda añadida en el modelo 3D.
Curso normal	1.- El usuario selecciona la opción de crear cotas. 2.- El usuario pincha con el botón izquierdo del ratón sobre el punto de inicio y de fin de la cota. 3.- La aplicación calcula el **tamaño de la cota**, la añade a la **lista** de cotas y la dibuja en la posición correcta.
Cursos Alternativos	2.- Si se pincha fuera del modelo, el primer punto de la cota, si ya se había dado, se pierde. Si el **punto de inicio y de fin** coinciden, no se crea la cota.
Observaciones	
Requisitos no Funcionales Específicos	

2.1.2.13. Caso de uso *"Cambiar propiedades de una cota"*

Nombre del caso	Cambiar propiedades de una cota
Resumen	El usuario cambia las **propiedades** de una cota: visible, bloqueada, desplazada en X, desplazada en Y o desplazada en Z.
Dependencias	Caso de uso *"Seleccionar cotas"*.
Actores	Usuario
Precondiciones	Debe estar activo el modo cotas, que indica que queremos trabajar con ellas, debe existir alguna cota y debe haber alguna cota seleccionada.
Postcondición	Las propiedades de la cota quedan modificadas según lo indicado por el usuario.
Curso normal	1.- El usuario selecciona una cota. 2.- El usuario marca los nuevos **valores** de las propiedades de la cota en el menú de las cotas. 3.- El usuario le da a aceptar. 4.- Los valores de las propiedades de la cota se actualizan con los nuevos valores introducidos por el usuario.
Cursos Alternativos	3.- Si no hay seleccionada ninguna cota, no se hace nada.
Observaciones	
Requisitos no Funcionales Específicos	

2.1.2.14. Caso de uso *"Eliminar cota"*

Nombre del caso	Eliminar cota
Resumen	El usuario elimina una cota.
Dependencias	Caso de uso *"Activar modo cotas"*.
Actores	Usuario
Precondiciones	Debe haber un modelo cargadoa en la aplicación correctamente y debe estar activado el modo cotas.
Postcondición	La cota queda eliminada.
Curso normal	1.- El usuario hace **clic** con el **botón derecho** sobre una cota. 2.- El sistema comprueba si se ha hecho clic sobre una cota y sobre cuál de las cotas. 3.- El sistema elimina la cota de la lista de cotas si la propiedad bloqueada de la cota no estaba activa.
Cursos Alternativos	1.- Si no se hace clic derecho sobre una cota, el caso de uso no hace nada.
Observaciones	
Requisitos no Funcionales Específicos	

2.1.2.15. Caso de uso *"Cambiar color cotas"*

Nombre del caso	**Cambiar color cotas**
Resumen	El usuario cambia el color de las cotas.
Dependencias	Caso de uso *"Activar modo cotas"*.
Actores	Usuario
Precondiciones	Debe estar activado el modo cotas.
Postcondición	El color de las cotas queda modificado según la elección del usuario.
Curso normal	1.- El usuario indica a la aplicación que quiere cambiar el color de las cotas.2.- Se muestra un menú con los colores disponibles. 3.- El usuario selecciona el color deseado y le da a aceptar. 4.- El color de las cotas queda modificado.
Cursos Alternativos	3.- Si el usuario le da a cancelar, el color no se modifica y el caso de uso finaliza.
Observaciones	Aunque no haya ningún modelo cargado, éste valor se puede actualizar.
Requisitos no Funcionales Específicos	

2.1.2.16. Caso de uso *"Cambiar grosor cotas"*

Nombre del caso	**Cambiar grosor cotas**
Resumen	El usuario cambia el **grosor** de las cotas.
Dependencias	Caso de uso *"Activar modo cotas"*.
Actores	Usuario
Precondiciones	Debe estar activado el modo cotas.
Postcondición	El grosor de las cotas queda modificado según la elección del usuario.
Curso normal	1.- El usuario indica a la aplicación el nuevo grosor de las cotas en el menú que se obtiene al activar el modo cotas. 2.- El grosor de las cotas se modifica automáticamente y se muestran los resultados.
Cursos Alternativos	
Observaciones	Aunque no haya ningún modelo cargado, éste valor se puede actualizar.
Requisitos no Funcionales Específicos	

2.1.2.17. Caso de uso *"Activar ocultación automática de cotas"*

Nombre del caso	Activar ocultación automática de cotas
Resumen	El usuario elige activar o desactivar la **ocultación automática de cotas**. Si ésta está activada, las cotas pueden quedar ocultas por el modelo según la **perspectiva** de éste, si no, siempre serán visibles.
Dependencias	Caso de uso *"Activar modo cotas"*.
Actores	Usuario
Precondiciones	Debe estar activado el modo cotas.
Postcondición	La ocultación automática de cotas queda activada si antes estaba desactivada o desactivada si antes estaba activada.
Curso normal	1.- El usuario indica a la aplicación el nuevo **estado** de la ocultación automática de cotas. 2.- La aplicación actualiza dicho valor y muestra en el visor lo correspondiente a este nuevo estado.
Cursos Alternativos	
Observaciones	Aunque no haya ningún modelo cargado, éste valor se puede actualizar.
Requisitos no Funcionales Específicos	

2.1.2.18. Caso de uso *"Cambiar color del modelo"*

Nombre del caso	Cambiar color del modelo
Resumen	El usuario cambia el color del modelo, dado según el **modelo RGB**.
Dependencias	
Actores	Usuario
Precondiciones	
Postcondición	El color del modelo queda modificado al nuevo valor RGB dado por el usuario.
Curso normal	1.- El usuario modifica la cantidad de **rojo**, **verde** o **azul** del color del modelo.2.- El nuevo **aspecto del modelo** se va mostrando automáticamente.
Cursos Alternativos	2.- Si no hay ningún modelo cargado, el nuevo color no se apreciará hasta cargar alguno, pero el valor del mismo se irá modificando según los deseos del usuario.
Observaciones	
Requisitos no Funcionales Específicos	

2.1.2.19. Caso de uso *"Cambiar escala"*

Nombre del caso	Cambiar escala
Resumen	El usuario cambia la **escala** del modelo entre las tres posibles: 1:1, 1:10 y 1:20.
Dependencias	
Actores	Usuario
Precondiciones	
Postcondición	La escala del modelo queda modificada al nuevo valor elegido por el usuario, y el modelo se visualiza según esta escala.
Curso normal	1.- El usuario elige una de entre las escalas disponibles para el modelo. 2.- El sistema actualiza la visualización del modelo y el tamaño mostrado de las cotas.
Cursos Alternativos	
Observaciones	
Requisitos no Funcionales Específicos	

2.1.2.20. Caso de uso *"Activar modo líneas de forma"*

Nombre del caso	Activar modo líneas de forma
Resumen	El usuario activa el modo líneas de forma para indicar al programa que quiere que se dibujen y que quiere trabajar con ellas.
Dependencias	
Actores	Usuario
Precondiciones	
Postcondición	El modo líneas de forma queda activado.
Curso normal	1.- El usuario le da a activar modo líneas de forma. 2.- El modo líneas de forma se activa y se muestra la ventana con todas las opciones disponibles para las **líneas de forma**. 3.- Las líneas de forma se muestran automáticamente en el visor en lugar del modelo en 3D.
Cursos Alternativos	
Observaciones	Se pueden mostrar más o menos líneas de forma modificando el valor de un **umbral** en el menú de las líneas de forma.
Requisitos no Funcionales Específicos	

2.1.2.21. Caso de uso *"Cambiar color de las líneas de forma"*

Nombre del caso	**Cambiar color de las líneas de forma**
Resumen	El usuario cambia el color de las líneas de forma.
Dependencias	Caso de uso *"Activar modo líneas de forma"*.
Actores	Usuario
Precondiciones	Debe estar activado el modo líneas de forma.
Postcondición	El color de las líneas de forma queda modificado según la **elección del usuario**.
Curso normal	1.- El usuario indica a la aplicación que quiere cambiar el color de las líneas de forma. 2.- Se muestra un menú con los colores disponibles. 3.- El usuario selecciona el color deseado y le da a aceptar. 4.- El color de las líneas de forma queda modificado.
Cursos Alternativos	3.- Si el usuario le da a cancelar, el color no se modifica y el caso de uso finaliza.
Observaciones	Aunque no haya ningún modelo cargado, éste valor se puede actualizar.
Requisitos no Funcionales Específicos	

2.1.2.22. Caso de uso *"Cambiar grosor líneas de forma"*

Nombre del caso	**Cambiar grosor líneas de forma**
Resumen	El usuario cambia el grosor de las líneas de forma.
Dependencias	Caso de uso *"Activar modo líneas de forma"*.
Actores	Usuario
Precondiciones	Debe estar activado el modo líneas de forma.
Postcondición	El grosor de las líneas de forma queda modificado según la elección del usuario.
Curso normal	1.- El usuario indica a la aplicación el nuevo grosor de las líneas de forma en el menú que se obtiene al activar el modo líneas de forma. 2.- El grosor de las líneas de forma se modifica automáticamente y se muestran los resultados.
Cursos Alternativos	
Observaciones	Aunque no haya ningún modelo cargado, éste valor se puede actualizar.
Requisitos no Funcionales Específicos	

2.1.2.23. Caso de uso *"Cambiar estilo de las líneas de forma"*

Nombre del caso	**Cambiar estilo de las líneas de forma**
Resumen	El usuario cambia el **estilo** de las líneas de forma por otro escogido de una lista.
Dependencias	Caso de uso *"Activar modo líneas de forma"*.
Actores	Usuario
Precondiciones	Debe estar activado el modo líneas de forma.
Postcondición	Las líneas de forma se muestran con el estilo elegido.
Curso normal	1.- El usuario elige uno de los estilos de líneas de forma disponibles de entre los ofrecidos por la aplicación en el menú correspondiente a las líneas de forma. 2.- El usuario elige cuánto quiere exagerar el estilo con la opción **aleatoriedad** del menú correspondiente a las líneas de forma. 3.- La aplicación actualiza la visualización de las líneas de forma según lo elegido.
Cursos Alternativos	
Observaciones	Aunque no haya ningún modelo cargado, éste valor se puede actualizar.
Requisitos no Funcionales Específicos	

2.1.2.24. Caso de uso *"Activar modo silueta"*

Nombre del caso	**Activar modo silueta**
Resumen	El usuario activa el modo silueta para indicar al programa que quiere que se dibujen y que quiere trabajar con ellas.
Dependencias	
Actores	Usuario
Precondiciones	
Postcondición	El modo silueta queda activado.
Curso normal	1.- El usuario le da a activar modo silueta. 2.- El modo silueta se activa y se muestra la ventana con todas las opciones disponibles para la silueta. 3.- La **silueta** se muestra automáticamente en el visor en lugar del modelo en 3D.
Cursos Alternativos	
Observaciones	La silueta se puede mostrar más o menos claramente modificando el valor de un umbral en el menú de la silueta.
Requisitos no Funcionales Específicos	

2.1.2.25. Caso de uso *"Cambiar color de la silueta"*

Nombre del caso	Cambiar color de la silueta
Resumen	El usuario cambia el color de la silueta.
Dependencias	Caso de uso *"Activar modo silueta"*.
Actores	Usuario
Precondiciones	Debe estar activado el modo silueta.
Postcondición	El color de la silueta queda modificado según la elección del usuario.
Curso normal	1.- El usuario indica a la aplicación que quiere cambiar el color de la silueta. 2.- Se muestra un menú con los colores disponibles. 3.- El usuario selecciona el color deseado y le da a aceptar. 4.- El color de la silueta queda modificado.
Cursos Alternativos	3.- Si el usuario le da a cancelar, el color no se modifica y el caso de uso finaliza.
Observaciones	Aunque no haya ningún modelo cargado, éste valor se puede actualizar.
Requisitos no Funcionales Específicos	

2.1.2.26. Caso de uso *"Cambiar grosor silueta"*

Nombre del caso	Cambiar grosor silueta
Resumen	El usuario cambia el grosor de la silueta.
Dependencias	Caso de uso *"Activar modo silueta"*.
Actores	Usuario
Precondiciones	Debe estar activado el modo silueta.
Postcondición	El grosor de la silueta queda modificado según la elección del usuario.
Curso normal	1.- El usuario indica a la aplicación el nuevo grosor de la silueta en el menú que se obtiene al activar el modo silueta. 2.- El grosor de la silueta se modifica automáticamente y se muestran los resultados.
Cursos Alternativos	
Observaciones	Aunque no haya ningún modelo cargado, este valor se puede actualizar.
Requisitos no Funcionales Específicos	

2.1.2.27. Caso de uso *"Cambiar estilo de la silueta"*

Nombre del caso	**Cambiar estilo de la silueta**
Resumen	El usuario cambia el estilo de la silueta por otro escogido de una lista.
Dependencias	Caso de uso *"Activar modo silueta"*.
Actores	Usuario
Precondiciones	Debe estar activado el modo silueta.
Postcondición	La silueta se muestra con el estilo elegido.
Curso normal	1.- El usuario elige uno de los estilos de silueta disponibles de entre los ofrecidos por la aplicación en el menú correspondiente a la silueta. 2.- El usuario elige cuánto quiere exagerar el estilo con la opción aleatoriedad del menú correspondiente a la silueta. 3.- La aplicación actualiza la visualización de la silueta según lo elegido.
Cursos Alternativos	
Observaciones	Aunque no haya ningún modelo cargado, éste valor se puede actualizar.
Requisitos no Funcionales Específicos	

2.1.2.28. Caso de uso *"Activar silueta estricta"*

Nombre del caso	**Activar silueta estricta**
Resumen	El usuario elige activar o desactivar la silueta estricta. Si ésta está activada, sólo se ve la silueta externa de la figura, si no, también se ven las líneas de la silueta del **interior del objeto**.
Dependencias	Caso de uso *"Activar modo silueta"*.
Actores	Usuario
Precondiciones	Debe estar activado el modo silueta.
Postcondición	La silueta estricta queda activada si antes estaba desactivada o desactivada si antes estaba activada.
Curso normal	1.- El usuario indica a la aplicación el nuevo estado de la silueta estricta. 2.- La aplicación actualiza dicho valor y muestra en el visor lo correspondiente a este nuevo estado.
Cursos Alternativos	
Observaciones	Aunque no haya ningún modelo cargado, éste valor se puede actualizar.
Requisitos no Funcionales Específicos	

2.1.2.29. Caso de uso *"Activar modo sombreado"*

Nombre del caso	**Activar modo sombreado**
Resumen	El usuario activa el modo sombreado para indicar al programa que quiere que se dibuje y que quiere trabajar con él.
Dependencias	
Actores	Usuario
Precondiciones	
Postcondición	El modo sombreado queda activado.
Curso normal	1.- El usuario le da a activar modo sombreado. 2.- El modo sombreado se activa y se muestra la ventana con todas las opciones disponibles para el sombreado. 3.- El **sombreado** se muestra automáticamente en el visor en lugar del modelo en 3D.
Cursos Alternativos	
Observaciones	
Requisitos no Funcionales Específicos	

2.1.2.30. Caso de uso *"Cambiar textura para una intensidad"*

Nombre del caso	**Cambiar textura para una intensidad**
Resumen	El usuario cambia la **textura** que está usando el modelo en ese **momento** para una cierta **intensidad** por otra elegida o creada por éste.
Dependencias	Caso de uso *"Activar modo sombreado"*.
Actores	Usuario
Precondiciones	Debe estar activado el modo sombreado.
Postcondición	Se muestra el sombreado del modelo con las **zonas** de esa intensidad sombreadas con la nueva textura.
Curso normal	1.- El usuario selecciona la opción cambiar una textura. 2.- Se muestra un menú con los directorios y archivos del sistema. 3.- Se selecciona el directorio donde se encuentra la textura que queremos usar y se selecciona la textura. 4.- La aplicación carga la nueva textura y, según el número de texturas indicadas por el usuario, asigna la nueva textura cargada a una cierta intensidad. 5.- La aplicación muestra el resultado del sombreado.
Cursos Alternativos	3.- Si la carpeta indicada o cualquiera de los otros datos especificados por el usuario son inválidos, no se carga la textura y se dará un **error** notificando lo ocurrido. 4.- Si la imagen con la textura no cumple lo expuesto en el apartado de *Requisitos no Funcionales Específicos* de este caso de uso, la textura no se cargará y se notificará lo ocurrido.
Observaciones	
Requisitos no Funcionales Específicos	La textura debe ser una imagen de 256 x 256 con **formato tiff**.

2.1.2.31. Caso de uso *"Cambiar texturas para el sombreado"*

Nombre del caso	**Cambiar texturas para el sombreado**
Resumen	El usuario cambia a la vez todas las texturas que está usando el modelo en ese momento por otras elegidas o creadas por éste.
Dependencias	Caso de uso *"Activar modo sombreado"*.
Actores	Usuario
Precondiciones	Debe estar activado el modo sombreado. Todas las texturas deben estar dentro del mismo directorio.
Postcondición	Se muestra el sombreado del modelo con las nuevas texturas.
Curso normal	1.- El usuario selecciona la opción cargar un patrón completo. 2.- Se muestra un menú con los directorios y archivos del sistema. 3.- Se selecciona el directorio donde se encuentran las texturas que queremos usar y se selecciona una cualquiera de las texturas. 4.- La aplicación carga todas las texturas necesarias del directorio. 5.- La aplicación muestra el resultado del sombreado.
Cursos Alternativos	3.- Si la carpeta indicada o cualquiera de los otros datos especificados por el usuario son inválidos, no se cargan las texturas y se dará un error notificando lo ocurrido. 4.- Si las imágenes con las texturas no cumplen lo expuesto en el apartado de *Requisitos no Funcionales Específicos* de este caso de uso, las texturas no se cargarán y se notificará lo ocurrido.
Observaciones	
Requisitos no Funcionales Específicos	Las texturas deben ser imágenes de 256 x 256 con formato tiff.

2.1.2.32. Caso de uso *"Calcular sombreado computacional"*

Nombre del caso	Calcular sombreado computacional
Resumen	El modelo se sombrea mediante el uso de otro tipo de sombreado, que es calculado al completo por la aplicación.
Dependencias	Caso de uso *"Activar modo sombreado"*.
Actores	Usuario
Precondiciones	Debe estar activado el modo sombreado.
Postcondición	Se muestra el modelo con el sombreado computacional.
Curso normal	1.- El usuario selecciona la opción calcular sombreado computacional. 2.- El usuario elige el **número máximo de puntos que habrá por unidad de área** en el modelo. 3.- La aplicación calcula el sombreado. 4.- La aplicación muestra el resultado obtenido.
Cursos Alternativos	
Observaciones	
Requisitos no Funcionales Específicos	

2.1.2.33. Caso de uso *"Recalcular sombreado"*

Nombre del caso	Recalcular sombreado
Resumen	El usuario le indica a la aplicación que quiere recalcular el sombreado y ésta lo recalcula.
Dependencias	Caso de uso *"Activar modo sombreado"*.
Actores	Usuario
Precondiciones	Debe estar activado el modo sombreado.
Postcondición	La aplicación recalcula el sombreado y muestra el nuevo sombreado obtenido con las opciones elegidas para el mismo.
Curso normal	1.- El usuario selecciona la opción recalcular sombreado. 2.- La aplicación calcula el nuevo sombreado para cada cara en función de la intensidad de luz de las mismas, haciendo uso de las texturas activadas en ese momento (las de por defecto, si el usuario no ha elegido ningunas, o las elegidas por el usuario), o calculando de nuevo todo el sombreado, en el caso de tener marcada la opción de sombreado computacional. 3.- Se muestra el nuevo sombreado obtenido para el modelo.
Cursos Alternativos	
Observaciones	
Requisitos no Funcionales Específicos	

2.1.3. Diagrama de casos de uso

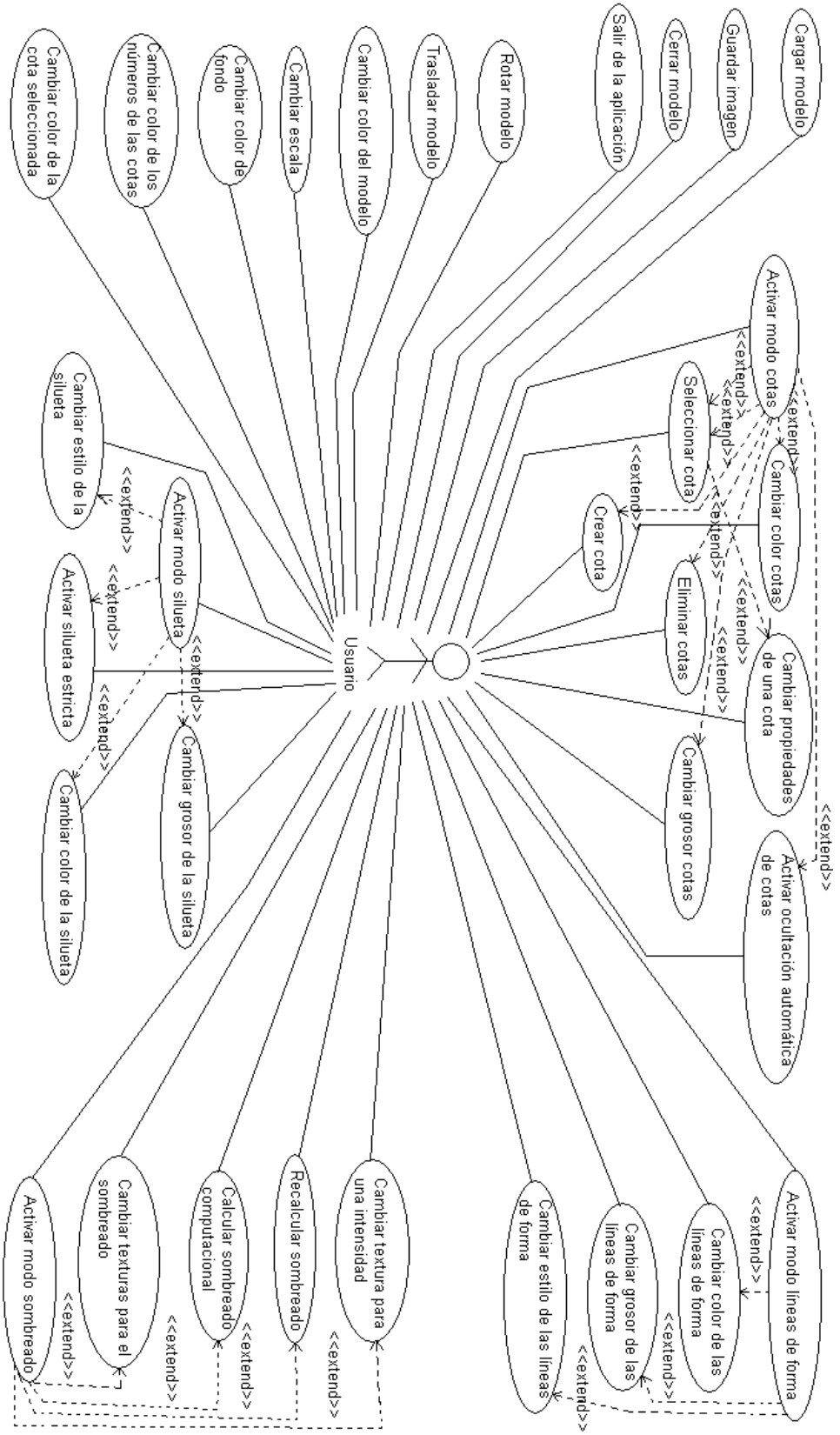


Figura 1. Diagrama de casos de uso resultante

2.2. Subsistemas funcionales

2.2.1. Identificación de los subsistemas funcionales

Una vez hemos obtenido los casos de uso que formarán parte de la aplicación, veamos cómo podemos agruparlos y los distintos paquetes de casos de uso que obtenemos.

- **Paquete *"Opciones de Archivo"***
  - Cargar modelo
  - Guardar imagen
  - Cerrar modelo
  - Salir de la aplicación

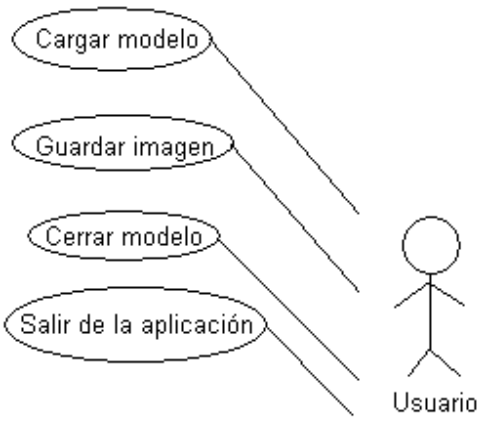


Figura 2. Diagrama de casos de uso para "Opciones de Archivo"

- **Paquete *"Opciones de la Interfaz"***
  - Rotar modelo
  - Trasladar modelo
  - Cambiar color del modelo
  - Cambiar escala
  - Cambiar color de fondo
  - Cambiar color de los números de las cotas
  - Cambiar color de la cota seleccionada

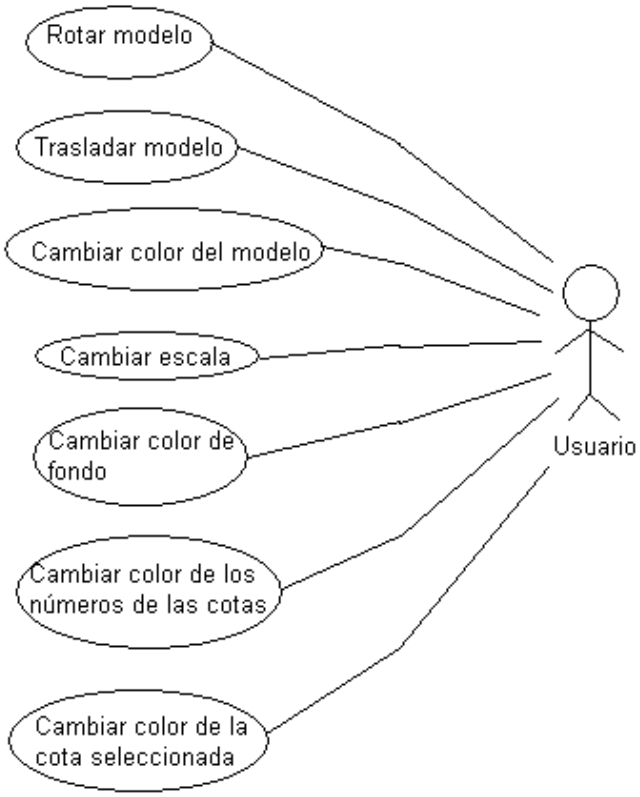


Figura 3. Diagrama de casos de uso para "Opciones de la Interfaz"

- **Paquete *"Cotas"***
  - Activar modo cotas
  - Seleccionar cota
  - Crear cota
  - Cambiar propiedades de una cota
  - Eliminar cota
  - Cambiar color cotas
  - Cambiar grosor cotas
  - Activar ocultación automática de cotas

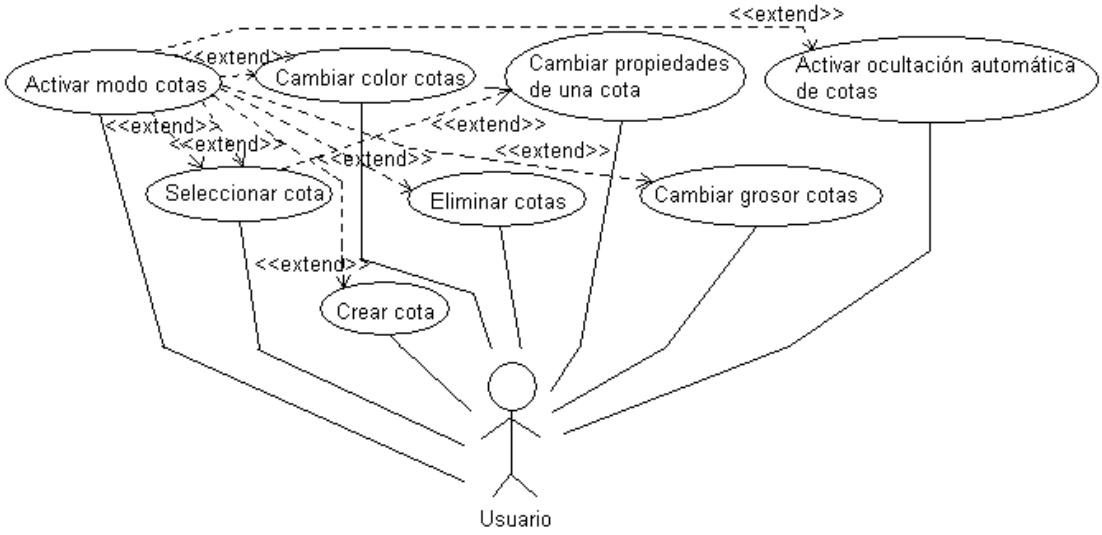


Figura 4. Diagrama de casos de uso para "Cotas"

- **Paquete *"Líneas de forma"***
  - Activar modo líneas de forma
  - Cambiar color de las líneas de forma
  - Cambiar grosor de las líneas de forma
  - Cambiar estilo de las líneas de forma

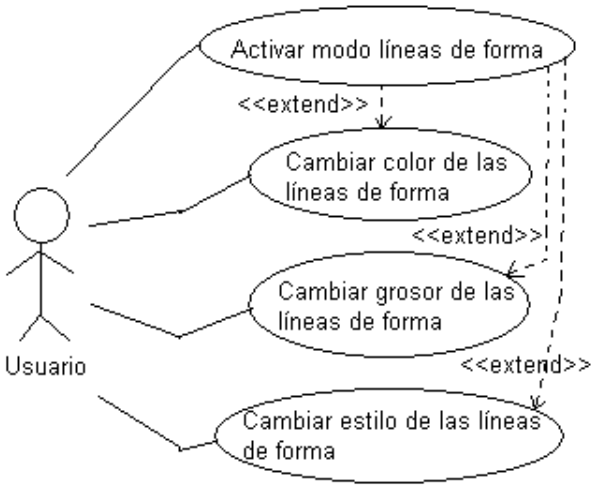


Figura 5. Diagrama de casos de uso para "Líneas de forma"

- **Paquete *"Silueta"***
  - Activar modo silueta
  - Cambiar color de la silueta
  - Cambiar grosor de la silueta
  - Cambiar estilo de la silueta
  - Activar silueta estricta

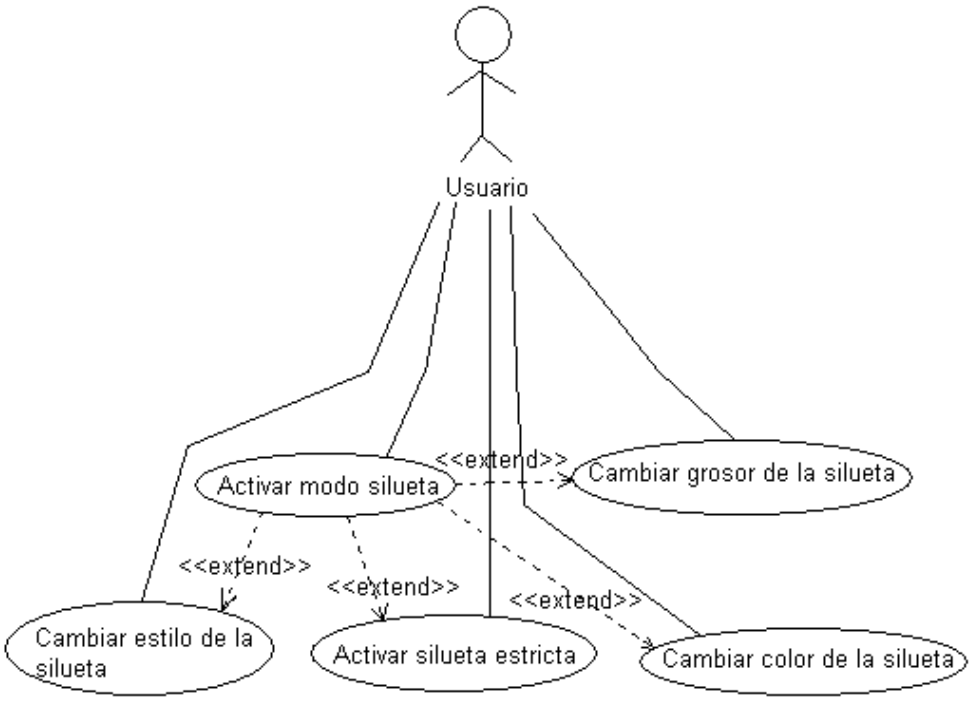


Figura 6. Diagrama de casos de uso para "Silueta"

- **Paquete *"Sombreado"***
  - Activar modo sombreado
  - Cambiar textura para una intensidad
  - Cambiar texturas para el sombreado
  - Calcular sombreado computacional
  - Recalcular sombreado

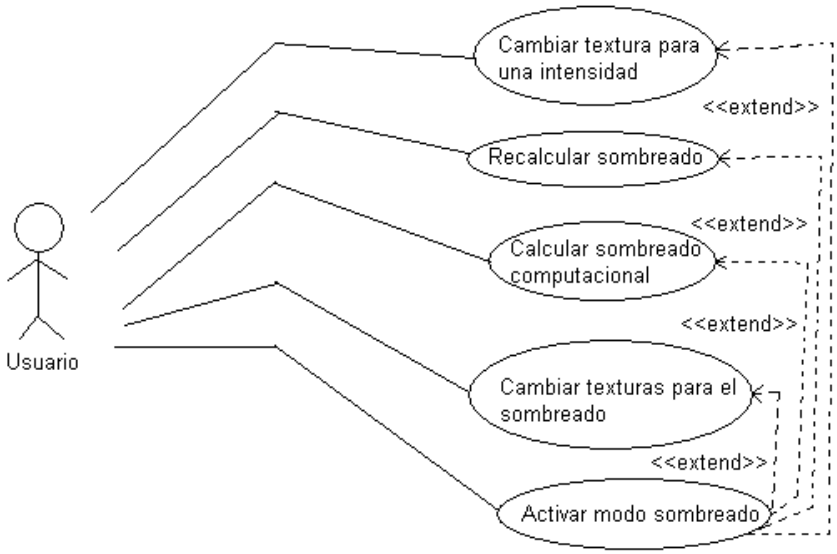


Figura 7. Diagrama de casos de uso para "Sombreado"

Proyecto Informático

5º Ingeniería Informática

Análisis

Curso 2006 – 2007

I. Introducción

Una vez se ha llevado a cabo la fase de *Modelado de Requisitos* de la aplicación software, la siguiente fase del proceso de desarrollo a realizar es el *Análisis,* y en ella se pretende comprender y describir el sistema en el dominio del problema, para lo cual se realizará una estructuración y formalización de los requerimientos obtenidos en el modelado de requisitos. En esta fase se trabajará con abstracciones del mundo real, y el objetivo será obtener objetos, situaciones o procesos de dicho mundo real.

Dentro de la fase de *Análisis* es posible realizar los siguientes procesos:

- Identificar clases, atributos y relaciones
- Realizar el modelado estático

II. Identificación de clases y atributos

2.1. Identificación de clases candidatas

Una de las primeras tareas que debemos realizar a la hora de obtener el análisis de nuestro sistema, consiste en identificar cuáles son las clases que consideraremos en el mismo. Para realizar esta tarea existen distintos métodos, pero, en este caso he optado por aplicar el método de frases nominales, en el que se parte de la descripción detallada de los casos de uso, y se va señalando sobre ella cuáles son los sustantivos y frases nominales que encuentre en ellos. Estos sustantivos se han marcado en **negrita** en la descripción que se dio previamente de cada caso de uso, en la sección 2.1.2 del Modelado de Requisitos. Indicar además, que se ha seleccionado por claridad solamente una vez cada sustantivo en la descripción de los casos de uso.

A continuación se muestra una lista en la que se incluyen todos los sustantivos marcados:

Aleatoriedad, ángulo, aplicación, archivo, aspecto del modelo, botón izquierdo, botón derecho, captura de la imagen, carpeta, color, configuración, cotas, datos, deseos del usuario, dirección en la que se mueve el modelo (posición del modelo), directorios y archivos del sistema, eje, elección del usuario, error, escala, estado, estilo, fichero, formato ply, formato tiff, grosor, intensidad, interior del objeto, líneas de forma, lista, mensaje, menú, modelo, modelo RGB, modo, momento, nombre, números de las cotas, número máximo de puntos por unidad de área, ocultación automática de cotas, opción, perspectiva, programa, propiedades, punto de inicio de una cota, punto de fin de una cota, ratón, color rojo, color verde, color azul, silueta, sistema, sombreado, tamaño de las cotas, textura, umbral, usuario, valor, ventana, visor, zonas.

2.2. Eliminación de clases candidatas

Una vez se ha obtenido la lista con todos los sustantivos que en principio podrían ser candidatos a clases, resulta imprescindible realizar un análisis de la misma con la intención de determinar cuáles de ellos deberán ser eliminados ya que no son clases candidatas.

Para ello se clasificarán dichos sustantivos en función de los criterios que se muestran a continuación:

Redundantes:
Los siguientes sustantivos no son clases candidatas debido a que ya han sido considerados como tales, pero con otro nombre:

- Programa (Aplicación)
- Deseos del usuario (Elección del usuario)
- Archivo (Fichero)

Irrelevantes:
Los siguientes sustantivos no serán considerados como clases debido a que hacen referencia a algo que no va a ser tratado en nuestro problema:

- Ratón
- Carpeta
- Usuario
- Captura de la imagen
- Nombre (de la captura de la imagen)

Imprecisas:
Los siguientes sustantivos no serán considerados como clases debido a que no se puede indicar de forma no ambigua lo que significan:

- Datos
- Opción
- Configuración
- Elección del usuario
- Momento
- Perspectiva
- Valor
- Zonas
- Aspecto del modelo

Fuera del alcance del sistema:
Los siguientes sustantivos no serán considerados como clases debido a que, a pesar de ser relevantes para describir cómo funciona el sistema, no hacen referencia a algo interno al mismo:

- Sistema

Atributo:
Los siguientes sustantivos no serán considerados como clases debido a que hacen referencia a algo sencillo sin un comportamiento interesante, y pueden ser por tanto propiedades o atributos de una clase:

- Aleatoriedad
- Ángulo
- Posición
- Eje
- Escala
- Color
- Estado
- Estilo
- Grosor
- Intensidad
- Modo
- Punto de inicio de la cota
- Punto de fin de la cota
- Color rojo
- Color verde
- Color azul
- Tamaño de las cotas
- Umbral
- Textura
- Propiedades
- Número máximo de puntos por unidad de área

Diseño:
Por último, los siguientes sustantivos no serán considerados como clases debido a que hacen referencia a un componente software o a la solución de un problema:

- Directorios y archivos del sistema
- Error
- Fichero
- Interior del objeto
- Lista
- Mensaje
- Menú
- Modelo RGB
- Números de las cotas
- Ocultación automática de las cotas
- Visor
- Ventana
- Botón izquierdo
- Botón derecho
- Formato (ply y tiff)

2.3. Atributos asociados a clases y/o relaciones

Finalmente, y tras el análisis explicado anteriormente, las clases y los atributos que se deben tener en cuenta son los siguientes:

- **Aplicación**:
  Modo de trabajo, estado ocultación automática de cotas, estado silueta estricta.

- **Modelo**:
  Ángulo de giro en ejes, posición, escala, color (rojo, verde, azul).

- **Cota**:
  Color (rojo, verde, azul), grosor, punto de inicio, punto de fin, tamaño, propiedades.

- **Líneas de forma:**
  Aleatoriedad del estilo, color (rojo, verde, azul), estilo, grosor, umbral.

- **Silueta:**
  Aleatoriedad del estilo, color (rojo, verde, azul), estilo, grosor, umbral.

- **Sombreado:**
  Intensidad, textura, número máximo de puntos por unidad de área.

Entre estas clases especificadas vamos a establecer las siguientes relaciones, explicadas a continuación:

- Una aplicación trabaja con un modelo.

- Un modelo pertenece a una aplicación.

- Un modelo puede tener definidas cero o muchas cotas.

- Una cota pertenece a un modelo.

- Un modelo tiene una silueta.

- Una silueta pertenece a un modelo.

- Un modelo tiene un sombreado.

- Un sombreado corresponde a un modelo.

- Un modelo tiene cero o muchas líneas de forma.

- Una línea de forma pertenece a un modelo.

III. Modelado estático

3.1. Diagrama de clases

Con esta descripción, el diagrama de clases quedaría como sigue:

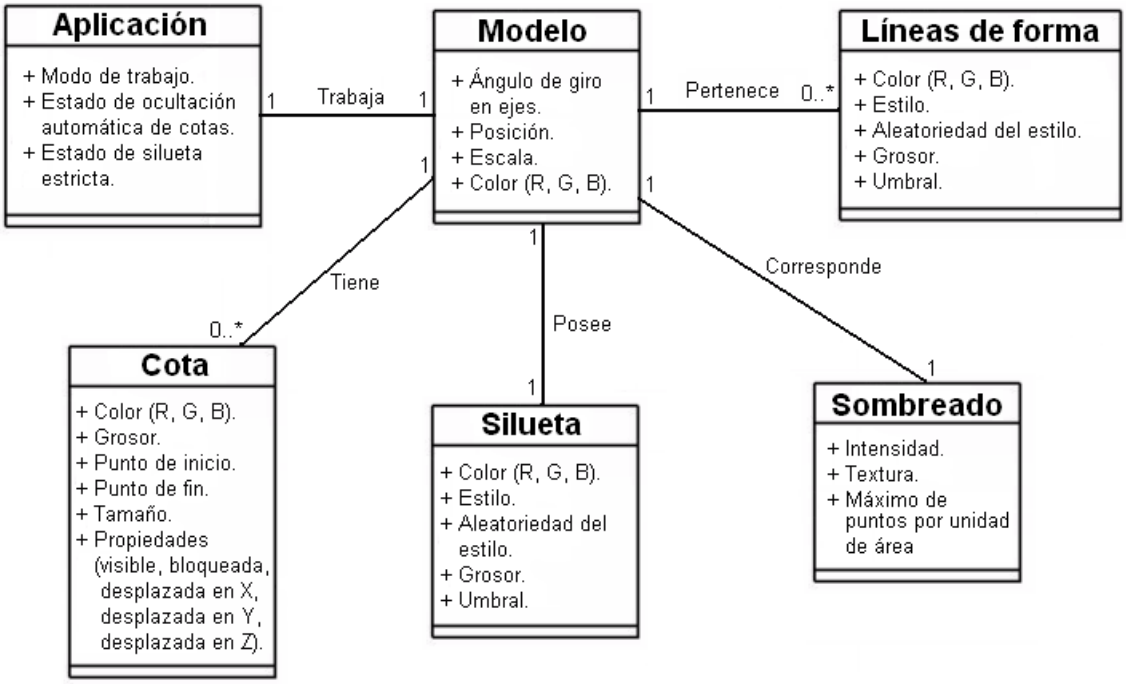


Figura 8. Diagrama de clases (fase de Análisis)

Proyecto Informático

5º Ingeniería Informática

Diseño

Curso 2006 – 2007

I. Diseño del sistema

1.1. Identificación de los objetivos de diseño

La lista de objetivos de diseño se forma en gran medida a partir de los requisitos no funcionales junto con los casos de uso, pero siempre es posible obtener alguno más como consecuencia del desarrollo posterior que se ha realizado sobre el sistema y que en un principio no se incluyeron debido a que la visión sobre el sistema no era tan detallada como en este momento. La siguiente lista muestra los objetivos que se tendrán que satisfacer mediante el diseño de la aplicación:

- Existirá una única aplicación que se podrá ejecutar en un ordenador personal cualquiera.
- La aplicación sólo cargará imágenes con formato .ply (que pueden ser ASCII o binarios), y creará imágenes con formato .tiff y .png.
- No se considerará un acceso concurrente a la aplicación.
- El modelo que carga la aplicación se almacenará en una estructura de tipo semiarista alada.
- La aplicación tendrá una interfaz de usuario atractiva, intuitiva, muy sencilla y será fácil de usar.
- No se considerarán distintos tipos de usuario para la aplicación. Cualquier usuario podrá realizar las mismas operaciones.
- La aplicación debe ser capaz de cargar texturas personalizadas para el sombreado.
- Es imprescindible que las estructuras y el programa en general estén bien optimizados para favorecer la velocidad de la aplicación.

1.2. Estructuras de datos

1.2.1. Semiaristas aladas

Para realizar esta aplicación, será necesario usar una estructura de semiaristas aladas para cargar los datos del modelo 3D y poder trabajar con ellos de forma eficiente. Una estructura de datos de semiaristas aladas es una estructura de datos centrada en las aristas capaz de mantener información de los vértices, aristas y caras. Cada arista se descompone en dos semiaristas con orientaciones opuestas. En cada semiarista se almacenan una cara incidente y un vértice incidente, y se puede almacenar otro tipo de información para acelerar el proceso de dibujado, como el identificador de la semiarista siguiente según la cara incidente. Para cada cara y cada vértice se almacena una semiarista.

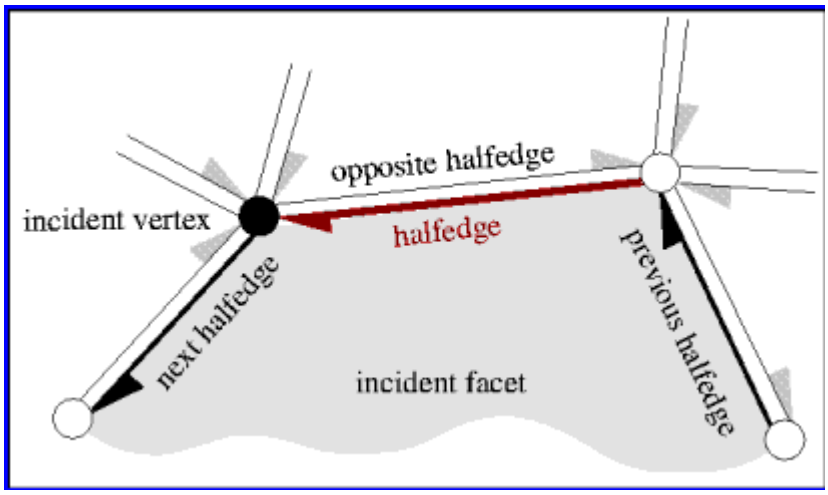


Figura 9. Estructura de datos: semiaristas aladas

En la aplicación, las semiaristas se almacenarán en una lista de tal forma que a cada semiarista con identificador par le siga (con identificador impar) su opuesta.

1.2.2. Punto

Será imprescindible para la aplicación hacer uso de una estructura para controlar los puntos. Ésta no será más que tres flotantes (x, y, z) que definen una coordenada en el espacio tridimensional.

1.2.3. Vértice

Un vértice no es más que un punto en el espacio que forma parte de una cara. Puesto que esta aplicación trabajará con modelos cargados de ficheros ply, necesitaremos tener una estructura para los mismos.

En particular, destacaremos de cada vértice sus coordenadas (un punto), su normal y la intensidad de luz que hay en el mismo.

En la aplicación tendremos una lista con todos los vértices del modelo, y los identificaremos mediante un entero único para cada vértice.

1.2.4. Cara

Una cara es un polígono que forma parte de un modelo tridimensional (en nuestro caso, puesto que los modelos se cargarán desde un fichero ply, pueden ser triángulos o cuadrados, aunque la mayoría de las opciones de la aplicación sólo funcionan correctamente con triángulos).

Necesitaremos conocer de cada cara el número de puntos que la forman, el identificador de la semiarista incidente en la cara (a partir del cual es fácil conocer la opuesta por lo explicado en el apartado 2.3.1.), la normal de la cara, si es una cara visible o no visible (para el cálculo de la silueta), el color único de la cara (para identificar si se ha pinchado sobre ella con el ratón) y el área de la misma. Tendremos también un identificador numérico que nos permitirá referirnos a cada cara fácilmente.

En la aplicación se mantendrá una lista con todas las caras del modelo.

1.3. Identificación de la arquitectura software

1.3.1. Estilo Arquitectónico

La arquitectura MVC (Model-View-Controller) ayuda a separar la capa de interfaz de usuario de otras partes del sistema (separa los datos de una aplicación, la interfaz de usuario y la lógica de control en tres componentes distintos). Los subsistemas se clasifican en tres tipos:

- Subsistema Modelo: Responsable del conocimiento del dominio de aplicación, es decir, es la representación específica de la información con la que el sistema opera. La lógica de datos asegura la integridad de estos y permite derivar nuevos datos.
- Subsistema Vista: Responsable de mostrar los objetos del dominio de aplicación al usuario, es decir, presenta el modelo en un formato adecuado para interactuar (normalmente una interfaz de usuario).
- Subsistema Controlador: Responsable de la secuencia de interacciones entre el usuario, es decir, responde a los eventos (normalmente acciones del usuario) e invoca cambios en el modelo y, probablemente, en la vista.

El flujo seguido normalmente por esta arquitectura es el siguiente:

1. El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, pulsando un botón)
2. El controlador recibe (por parte de los objetos de la interfaz-vista) la notificación de la acción solicitada por el usuario y gestiona el evento que llega.
3. El controlador accede al modelo, actualizándolo y, posiblemente, modificándolo de forma adecuada a la acción solicitada por el usuario.
4. El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se reflejan los cambios en el modelo. El modelo no debe tener conocimiento directo sobre la vista y el controlador no pasa objetos de dominio (el modelo) a la vista aunque puede dar la orden a la vista para que se actualice.
5. La interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente.

1.3.2. Características

Entre las principales características de la arquitectura MVC, se pueden destacar las siguientes:

- Cada uno de los subsistemas puede diseñarse independientemente.

- Se puede aumentar la cohesión de los subsistemas si la vista y el controlador se encuentran unidos en una capa de interfaz de usuario..

- Se reduce el acoplamiento dado que los canales de comunicación entre los subsistemas son mínimos.

- Los subsistemas vista y controlador hacen uso extensivo de componentes reutilizables.

- Es un sistema más flexible dado que se puede modificar la interfaz de usuario cambiando el subsistema vista, controlador o ambos.

- Se puede probar la aplicación separadamente de la interfaz de usuario.

1.4. Diseñar la descomposición en subsistemas

Tras identificar la arquitectura software, veremos la identificación en subsistemas de dicha arquitectura. Definiremos las capas y los subsistemas de cada uno de los elementos de la arquitectura.

Usuarios:

1. **Capa de Interfaz de Usuario:**

   - **Subsistema de Presentación:**
     Este subsistema se encarga de la presentación de la interfaz (activación/desactivación de los distintos modos, cambio del color de fondo, del color de los números de cota…).

2. **Capa de lógica de la aplicación:**

   - **Subsistema del modelo:**
     Subsistema encargado de cargar modelos desde el fichero elegido hasta una estructura de semiaristas aladas.

   - **Subsistema de modificación del modelo:**
     Subsistema encargado de las modificaciones que decide hacer el usuario en el modelo cargado: rotaciones, cambio del color del modelo, cambio de escala…

   - **Subsistema de cotas:**
     Subsistema encargado de la gestión de las cotas en el modelo: añadirlas, modificarlas, eliminarlas...

   - **Subsistema de líneas de forma:**

Subsistema encargado de la gestión de las líneas de forma: cálculo, cambio del grosor, efecto del dibujado…

- **Subsistema de silueta:**
  Subsistema encargado de la gestión de la silueta: cálculo, cambio del grosor, efecto del dibujado…

- **Subsistema de sombreado:**
  Subsistema encargado de la gestión del sombreado: cálculo, cambio de la textura…

3. **Capa de servicios:**

- **Subsistema de imagen:**
  Este subsistema se encargará de almacenar las imágenes que se deseen capturar del modelo en el formato elegido.

El diagrama de paquetes quedaría del siguiente modo:

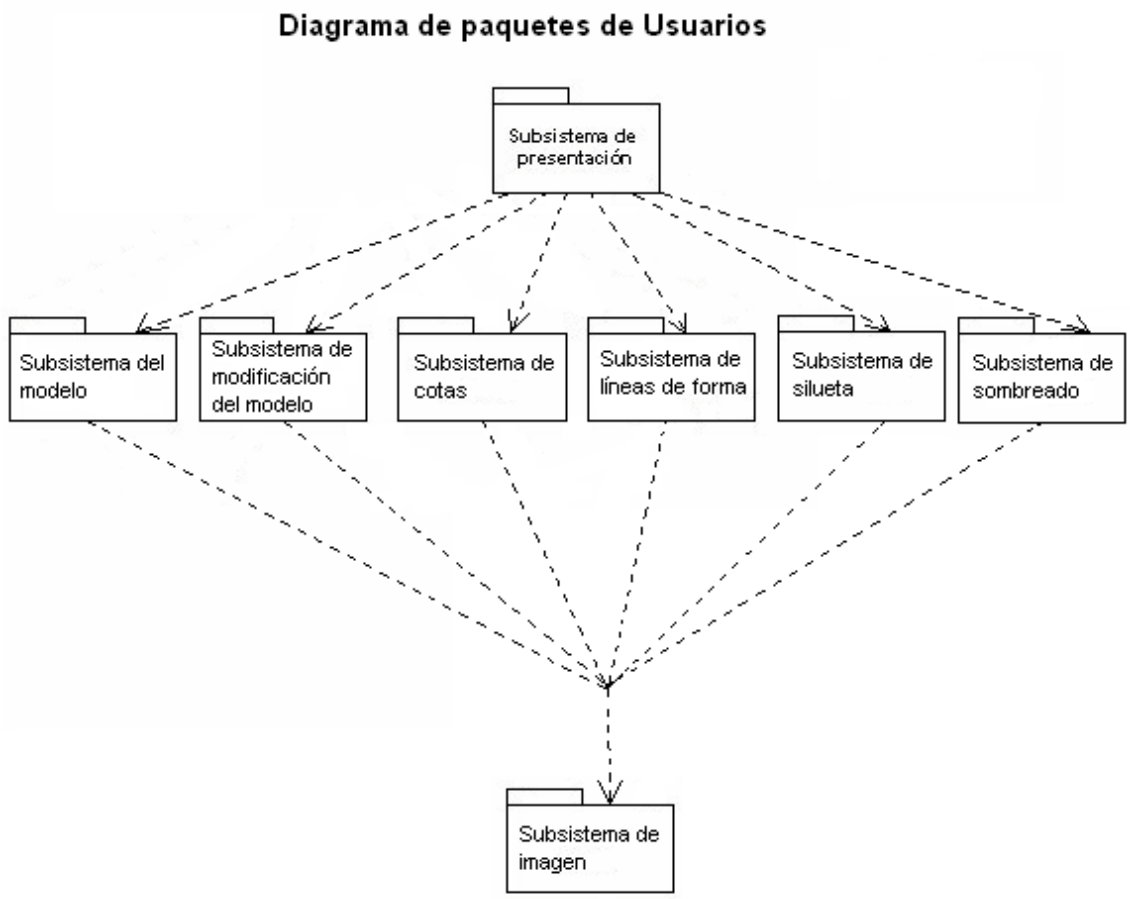


Figura 10. Diagrama de paquetes de Usuarios

1.5. Diagrama de clases del diseño

Tras el estudio llevado a cabo, el diagrama de clases final quedaría del siguiente modo:

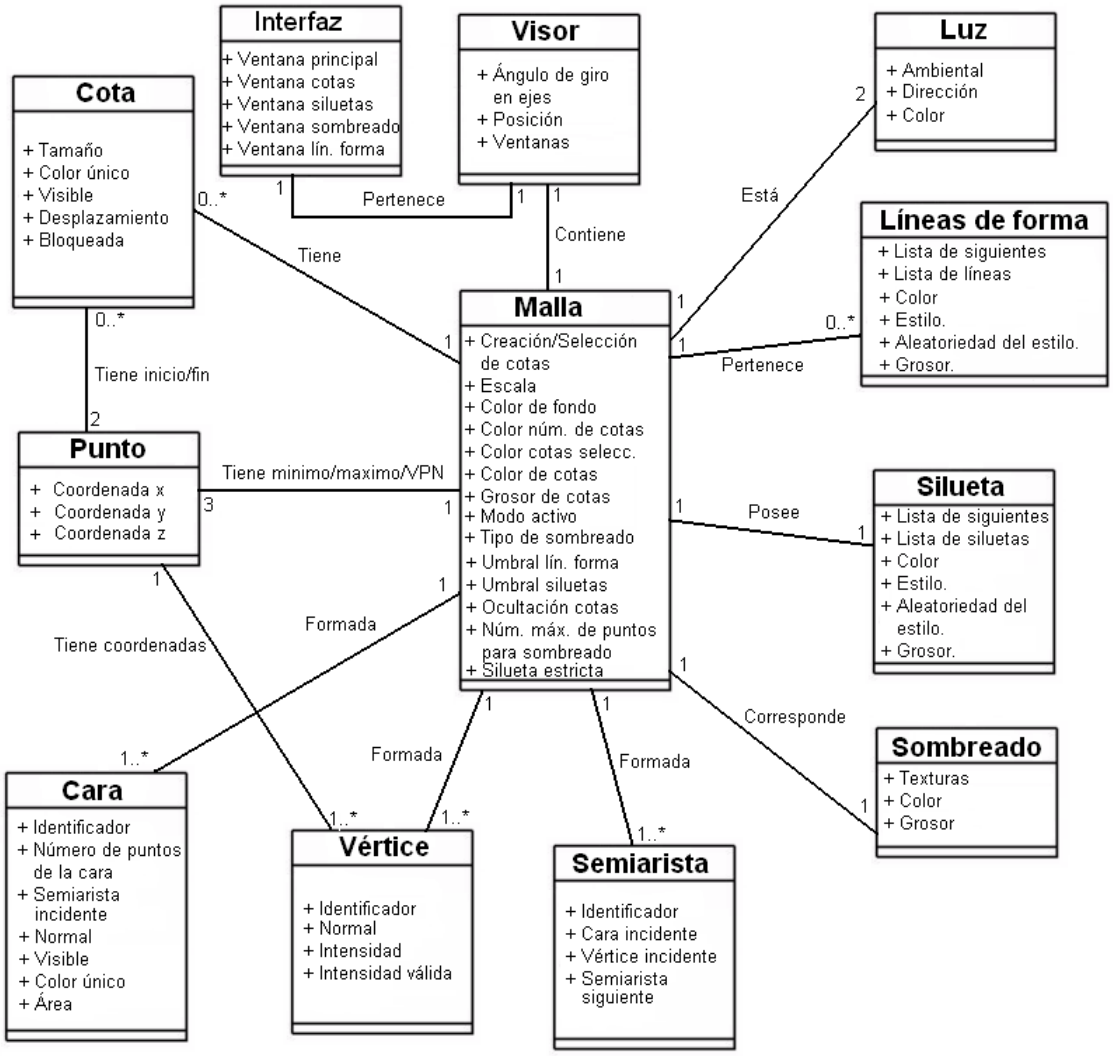


Figura 11. Diagrama de clases (fase de Diseño)

Es posible que haya atributos que no se comprenda aún bien su utilidad, pero será descrita más adelante en otros apartados con total detalle. Diremos simplemente que la luz será necesaria para el tema del sombreado, los colores únicos se usarán para la selección de elementos en el tema de las cotas y las listas de las siluetas y líneas de forma las necesitaremos para saber cuántas líneas de forma o siluetas se han detectado en el sistema y cuál es la siguiente de cada línea de forma o silueta en la estructura.

Respecto a qué clases de las indicadas se podrían englobar dentro de la definición de Vista, Controlador y Modelo, podemos decir lo siguiente:

- **Modelo**: Formado por las clases *Cota, Punto, Cara, Vértice, Semiarista, Sombreado, Silueta, Líneas de Forma* y *Luz*.
- **Vista**: Estará formada por las clases *Interfaz* y *Visor*.
- **Controlador**: Lo formará la clase *Malla*.

Proyecto Informático

5º Ingeniería Informática

Implementación

Curso 2006 – 2007

I. Introducción

Una de las partes más complicadas de este proyecto ha sido la implementación, ya que han surgido durante su desarrollo numerosos problemas que han afectado a la concepción e ideas iniciales que se tenían sobre la realización del trabajo.

En esta sección, pasaremos a explicar las distintas decisiones tomadas sobre el proceso de desarrollo, las técnicas usadas y los problemas surgidos durante el proceso.

II. Desarrollo

2.1. Fases

El desarrollo de este proyecto consta de diversas fases que se comentarán a continuación:

2.1.1. Carga del modelo en el sistema

Lo primero que ha sido necesario hacer para empezar a trabajar con este proyecto ha sido cargar correctamente un modelo de un fichero ply (que puede estar tanto en ASCII como en binario) a una estructura de semiaristas aladas.

2.1.1.1. Ficheros ply

Un fichero ply tiene el siguiente formato:

```
Identificación
Cabecera con descripción de los datos almacenados
Datos (Lista de vértices, lista de caras, elementos adicionales)
```

Para aclararlo más, veamos un ejemplo concreto:

```
ply
format ascii 1.0
element vertex 8
property float x
property float y
property float z
element face 12
property list uchar vertex_indices
end_header
0 0 0
1 0 0
1 1 0
0 1 0
0 0 1
1 0 1
1 1 1
0 1 1
3 1 0 2
3 0 3 2
3 0 4 3
3 4 7 3
```

```
3 4 5 7
3 5 6 7
3 5 1 6
3 1 2 6
3 2 3 6
3 3 7 6
3 0 1 4
3 1 5 4
```

En este ejemplo se define un cubo. La primera línea identifica el tipo de fichero con la palabra ply. A continuación se indica el formato de los datos, que puede ser ascii o binario (binary_big_endian o binary_little_endian), seguido por el número de versión. Luego se informa del número de vértices y, a continuación, de sus propiedades. Tras esto, se indica el número de caras y sus propiedades. En la cabecera también se pueden poner comentarios antecediendo la línea por la palabra comment. Cuando se ha terminado de describir los datos almacenados se pone en una línea a parte end_header, para indicar que hemos terminado de definir la cabecera. Tras esta, ponemos los datos. En el ejemplo, hay 8 vértices, luego los 8 primeros datos corresponden a los vértices (cada línea a un vértice). En este caso se definieron tres propiedades para los vértices (coordenadas x, y, z), luego cada vértice tiene 3 datos, uno por cada coordenada. De este modo, el primero vértice es el (0, 0, 0), el segundo es (1, 0, 0)… Tras los vértices se definen las 12 caras. En este caso, el primer número corresponde al número de vértices que definen cada cara y los otros tres al índice identificador de cada uno de esos vértices. En este ejemplo, las caras son triángulos (tres puntos) y la primera viene definida por los vértices (1, 0, 2). Hay casos más complejos en los que cada vértice puede tener información más general sobre su color, normal, coordenadas para texturas… También las caras pueden tener propiedades adicionales.

Respecto a las propiedades, definen tanto el tipo de dato de la propiedad como el orden en el que la propiedad aparece para cada elemento. Una propiedad puede tener dos tipos de datos: escalares y listas. A continuación se muestra una lista de los tipos de datos escalares que puede tener una propiedad:

Nombre	Tipo	Número de bytes
char	Carácter	1
uchar	Carácter sin signo	1
short	Entero corto	2
ushort	Entero corto sin signo	2
int	Entero	4
uint	Entero sin signo	4
float	Flotante de precisión simple	4
double	Flotante de precisión doble	8

El número de bytes es importante y no debe variar de unas implementaciones a otras para que estos ficheros sean portables. Hay una forma especial de definir propiedades que usa el tipo de datos lista:

property list

Si, por ejemplo, tenemos en un fichero *property list uchar int vertex_index* esto significa que la propiedad vertex_index contiene primero un unsigned char que nos dice cuántos índices contiene la propiedad, seguido por una lista que contiene múltiples enteros, que serán los índices de los vértices para las caras.

Otro ejemplo de definición de un cubo, esta vez un poco más complejo, sería la siguiente:

```
ply
format ascii 1.0
comment author: Greg Turk
comment object: another cube
element vertex 8
property float x
property float y
property float z
property red uchar
property green uchar
property blue uchar
element face 7
property list uchar int vertex_index
element edge 5
property int vertex1
property int vertex2
property uchar red
property uchar green
property uchar blue
end_header
0 0 0 255 0 0
0 0 1 255 0 0
0 1 1 255 0 0
0 1 0 255 0 0
1 0 0 0 0 255
1 0 1 0 0 255
1 1 1 0 0 255
1 1 0 0 0 255
3 0 1 2
3 0 2 3
4 7 6 5 4
4 0 4 5 1
4 1 5 6 2
4 2 6 7 3
4 3 7 4 0
0 1 255 255 255
1 2 255 255 255
2 3 255 255 255
3 0 255 255 255
2 0 0 0 0
```

En este caso, se especifica un valor de rojo, verde y azul para cada vértice. Puesto que estos ficheros permiten que haya caras de distinto número de vértices, podemos observar que las dos primeras caras del objeto son triángulos y el resto cuadriláteros. Luego el número de caras del objeto son 7. Este objeto contiene también una lista de aristas (5). Cada arista contiene dos punteros a los vértices que la delimitan y un color.

A parte de los elementos comentados (caras, vértices y aristas), los ficheros ply permiten definir otros al usuario de la misma forma que se definen los vértices, caras y aristas. [1]

2.1.1.2. Semiaristas aladas y otras estructuras

Las semiaristas aladas son estructuras de datos centradas en las aristas y capaces de mantener información de vértices, aristas y caras. Cada arista se descompone en dos semiaristas con orientaciones opuestas. En cada semiarista se almacenan, al menos, una cara incidente y un vértice incidente, y se puede almacenar otro tipo de información para acelerar el proceso de dibujado, como el identificador de la semiarista siguiente según la cara incidente. Para cada cara y cada vértice se almacena una semiarista. [2]

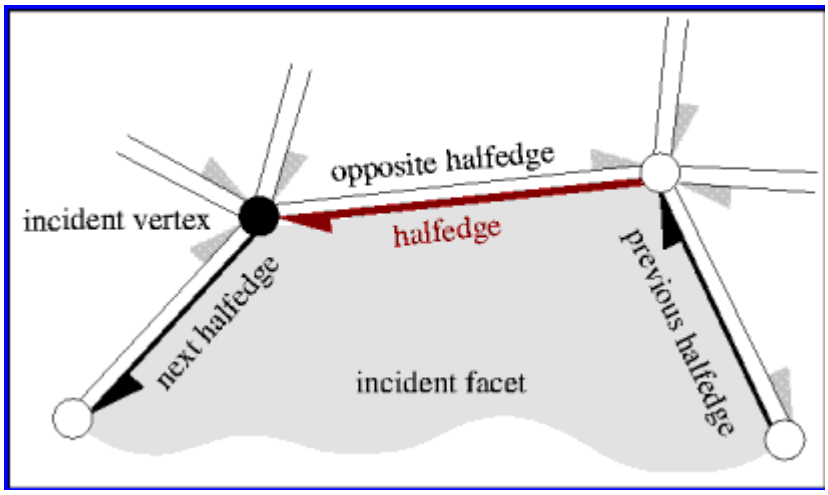


Figura 12. Semiaristas aladas

Para nuestra aplicación tendremos lo siguiente:

- Punto: Esta estructura denota un punto cualquiera dado por sus tres coordenadas.

Datos de un vértice	Tipo de dato	Descripción
x	float	Coordenada x.
y	float	Coordenada y.
z	float	Coordenada z.

- Lista de vértices: Esta estructura será un vector que contiene un vértice en cada índice. Los vértices serán estructuras con el siguiente formato:

Datos de un vértice	Tipo de dato	Descripción
num	int	Número identificador de cada vértice.
vert	punto	Coordenadas del vértice en forma de punto.
normalv	punto	Normal del vértice.
intensidad	float [3]	Intensidad en el vértice (rojo, verde, azul).
inten_valida	int	Indica si la intensidad de ese vértice es válida (se ha calculado) o no.

- Lista de caras: Una lista de caras será un vector que contiene en cada índice una estructura cara. Las caras tienen el siguiente formato:

Datos de una cara	Tipo de dato	Descripción
num	int	Número identificador de cada cara.
puntoscara	int	Número de puntos que forma la cara.
semaris	int	Identificador de la semiarista incidente en la cara (nótese que conociéndolo sé tanto la incidente como la opuesta de ésta).
normalc	punto	Normal de la cara.
Visible	unsigned int	Nos indica si la cara es visible o no (lo necesitaremos para la silueta, como se explicará más adelante).
color_unico	int [3]	Cada cara tendrá un color único (rojo, verde, azul), diferente al del resto de las caras.
area_t	float	Área de una cara.

- Lista de semiaristas: Las semiaristas del modelo se irán almacenando en un vector de semiaristas. Para evitarnos tener que añadir a una semiarista un campo con información de cuál es su opuesta, las almacenaremos en el vector de forma que las aristas con identificador impar sean las opuestas de las aristas con identificador par precedentes, es decir, la opuesta de la arista 0 es la 1 y la opuesta de la 4 es la 5 (por supuesto, la opuesta de la 7 será la 6, ya que la propiedad se cumple en ambas direcciones). El formato de cada semiarista es el siguiente:

Datos de una semiarista	Tipo de dato	Descripción
num	int	Número identificador de cada semiarista.
carinci	int	Cara incidente.
vertinci	int	Identificador del vértice inicial de la semiarista.
siguiente	int	Identificador de la semiarista siguiente según la cara incidente.

2.1.1.3. Implementación: carga del modelo

Algunas notas a destacar sobre la implementación son las siguientes:

- Para leer los ficheros ply se hace uso de dos ficheros ply.h y plyfile.c que se pueden encontrar fácilmente por la red. [8]

- Para ver los valores de colores únicos correspondientes a cada cara, se hace uso de un contador de color. A la primera cara se le asigna el color (1, 0, 0), correspondiendo el primer valor al rojo, el segundo al verde y el tercero al azul. Podemos tener valores en cada color desde el 0 hasta el 255, lo que nos da una gran cantidad de posibilidades. A la segunda cara se le asigna el color (2, 0, 0). Cuando lleguemos al (255, 0, 0), la siguiente cara tendrá el color (0, 1, 0). De este modo, el último color posible será (255, 255, 255), pero ya que un modelo debe tener un número de caras enorme (más de 16 millones) para alcanzar este valor se puede decir que es imposible llegar hasta él. En caso de que un modelo tuviera ese número de caras, el modo de trabajo con las cotas daría problemas, pues está basado en este sistema como se explicará más adelante.

- Se calculan las coordenadas x, y y z mínimas y máximas del modelo, pues las necesitaremos en varias ocasiones más adelante.

- El área de las caras la necesitaremos para el sombreado, al igual que la intensidad de los vértices.

- El color único de cada cara será necesario, como se explicará más adelante, para identificar entre qué caras quiere colocar una cota el usuario sin hacer uso de GLPick, que es mucho más lento. Esto se explicará en el apartado correspondiente a las cotas.

- Los valores de las coordenadas de la estructura de vértices se van leyendo directamente del fichero ply, excepto los identificadores, que se van calculando a partir del último identificador usado (empezando por el 0). La intensidad de un vértice se calcula sólo cuando se va a hacer el sombreado o cuando el usuario está trabajando con éste y lo solicita expresamente, luego se explicará en el apartado correspondiente al sombreado. Sólo notar que el valor de inten_valida será para todos los vértices 0 al principio, ya que no ha sido calculada para ninguno. Respecto a la normal de un vértice, se calcula como la media de las normales de las caras a las que pertenece dicho vértice.

- En principio, para crear las listas de las distintas estructuras según lo que se iba leyendo del fichero ply se usaron listas enlazadas, pero la eficiencia era baja e incluso los modelos sencillos tardaban demasiado en cargarse, por lo que hubo que rehacer esta parte. Para mejorar la eficiencia se han usado los vectores de la librería estándar STL.

- Para rellenar la información de las semiaristas se ha tenido en cuenta que en el fichero ply podemos encontrar directamente la información sobre la cara incidente y el vértice incidente de una semiarista, pero no la información de la semiarista siguiente. Además, al rellenar la lista, se van introduciendo tanto los datos de una semiarista como los de su opuesta (los datos de los que dispongamos, claro está). Es para esta parte, entre otras, para lo que necesitaremos tener un identificador para cada vértice, cara y semiarista. Veamos un ejemplo:

Supongamos que la cara 3 está formada por los vértices 1, 0 y 3. Tendremos ahí tres aristas (1-0, 0-3 y 3-1) y 6 semiaristas (1-0 y 0-1, 0-3 y 3-0, y 3-1 y 1-3). Hay que tener en cuenta que si no son las primeras semiaristas de la lista debemos comprobar que no las hayamos introducido ya. Por ejemplo:

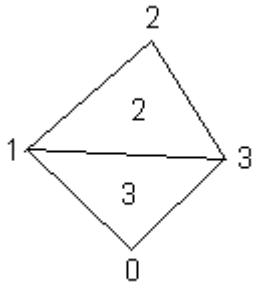


Figura 13. Ejemplo de dos caras (2 y 3) y sus vértices.

Si tenemos las caras 2 y 3 con sus vértices y ya hemos almacenado los datos existentes de la cara 2, las semiaristas 1-3 y la 3-1 ya están almacenadas, así que no hay que hacer nada más con ellas.
Independientemente de esto, si tenemos que almacenar la semiarista 0-3 (y su opuesta 3-0), sabemos lo siguiente:

- Los identificadores son fáciles de obtener: los siguientes que toquen empezando desde el 0.
- La cara incidente también es fácil: para la semiarista directa es la misma cara que estamos examinando. Para la opuesta, es la cara que estemos analizando cuando encontremos la secuencia de vértices 3-0 (por ejemplo, 0-4-3, teniendo en cuenta que el último vértice se une con el primero). Puesto que no tenemos por qué conocer qué cara es la incidente de la opuesta cuando estemos analizando los datos de la directa, metemos esta arista en una lista para saber que nos falta su cara incidente y cuando analicemos una nueva arista comprobamos lo que hay en esta lista por si podemos terminar de rellenar los datos de alguna de las semiaristas.
- El vértice incidente es fácil de obtener: para la semiarista directa es el segundo vértice de los que forman la arista y para la opuesta es el primero.
- La semiarista siguiente es más difícil de conseguir. Una

semiarista será siguiente de otra si tiene la misma cara incidente y el vértice incidente de su opuesta es el vértice incidente de su predecesora. Para aclarar esto:

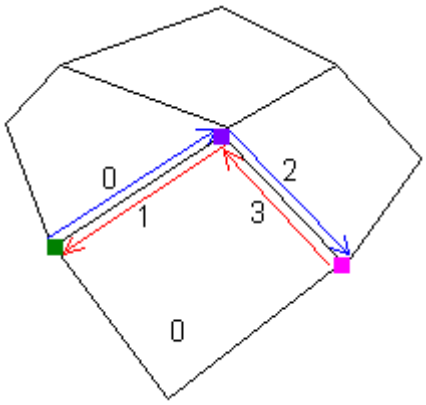


Figura 14. Ejemplo ilustrativo sobre las semiaristas. En azul, las semiaristas directas y en rojo las opuestas.

En el ejemplo, la siguiente de la semiarista 0 es la 2 (ambas en azul), porque la cara incidente de las dos es la misma (la 0) y el vértice incidente de la opuesta de 2 (la semiarista 3) es el vértice morado, que es el mismo que el vértice incidente de la arista 0 (predecesora de 2 puesto que 2 es siguiente de 0). Sin embargo, la semiarista 1 no es siguiente de 3 puesto que la cara incidente de ambas no es la misma.

Para resolver este problema computacionalmente, en un principio rellenaba primero la lista de semiaristas con todos los datos exceptuando el referente a la semiarista siguiente y luego recorría la lista de semiaristas buscando la siguiente (comprobando lo explicado anteriormente). Pero esto ralentizaba mucho el problema según se iba aumentando la complejidad del modelo, ya que los modelos grandes tienen muchas semiaristas y el recorrer toda lista en estos casos es pesado. Para solucionar este problema se usaron dos listas auxiliares teniendo en cuenta que toda semiarista puede ser siguiente de sólo otra semiarista y que toda semiarista debe tener una siguiente (exceptuando si el modelo tiene agujeros, en cuyo caso la siguiente será -1). En una de las listas auxiliares anotaremos las semiaristas sin siguiente (buscan siguiente) y en la otra las semiaristas que no son siguiente de nadie. Para cada semiarista que analizamos, damos dos pasos:

- Se comprueba si puede ser siguiente de alguna de las que buscan siguiente (que estarán en la lista de semiaristas sin siguiente). Si es así, se actualiza la información de esa semiarista en la lista de semiaristas y se elimina de la lista auxiliar puesto que ya no busca siguiente. Si no puede ser siguiente de ninguna, se introduce en la lista auxiliar de semiaristas que no son

siguientes de nadie.
  - Se busca en la lista de las semiaristas que no son siguiente de nadie una que pueda ser siguiente de la que estamos analizando. Si se encuentra, se actualiza la información de la que estamos analizando y se elimina la otra de la lista auxiliar. Si no se encuentra, anotamos la semiarista que estamos analizando en la lista de semiaristas que buscan siguiente.

De este modo, podemos ir rellenando toda la información de cada semiarista sin recorrer la lista completa varias veces.

- Para calcular los datos de una cara, podemos obtener directamente del fichero ply los puntos de una cara. El identificador se obtiene a partir del último identificador usado (empezando por 0). Como semiarista de una cara podemos quedarnos con cualquiera de las semiaristas que tienen a dicha cara como incidente (en nuestro caso, nos quedaremos con la última que identifiquemos como de esa cara mientras cargamos el modelo). Para calcular la normal de una cara, se calcula el producto vectorial de dos de sus aristas. El campo visible se inicializará a 0, y se calculará su valor real cuando estemos trabajando con las siluetas, luego se explicará en dicho apartado como obtenerlo. El color único de cada cara ya se ha explicado cómo calcularlo previamente y, respecto al área de cada cara, se calculará mediante la fórmula de Herón: $A=\sqrt{(s*(s-a)*(s-b)*(s-c))}$, donde $s=\sqrt{a+b+c}$ y donde a, b y c son las longitudes de los lados de un triángulo. Como se puede ver, esto sólo sirve para los triángulos. En caso de un cuadrilátero se tendría que hacer uso de la fórmula correspondiente a los mismos, pero esto último no se ha implementado, por lo que el cálculo del área sólo será válido en el caso de ficheros ply con todas las caras triangulares.

- Se calculan el área total (sumando todas las áreas de cada cara) y el área mayor (comparando cada nueva área calculada con la mayor encontrada hasta el momento) para usarlas si son necesarias en cálculos posteriores.

- La aplicación sólo funciona correctamente para ficheros ply con las caras triangulares. En caso de un fichero con caras cuadrangulares, el modelo se cargará y visualizará, pero el cálculo del área será incorrecto y muchas de las opciones del programa no funcionarán correctamente.

- La carga del modelo se hizo en un principio sin interfaz, por lo que el nombre del fichero ply para cargar se le daba al programa como argumento al ejecutarlo. Posteriormente, al añadir la interfaz se permitió que el nombre se le diera, o bien mediante la interfaz (sin usar ningún argumento) o bien como se hizo inicialmente, como un argumento al ejecutar la aplicación.

2.1.2. Creación de la interfaz

Para la creación de la interfaz se han usado principalmente las librerías Qt, y el visor de OpenGL QGLViewer, aunque previamente fue necesario informarse sobre cómo funcionan. Hay numerosas páginas en Internet con la documentación necesaria de QGLViewer, como es el caso de [3] y sobre las Qt, un interesante libro es el que podemos ver en la referencia [4].

En la interfaz podemos distinguir varias partes y fases principales que se explicarán a continuación.

2.1.2.1. Interfaz principal

La interfaz principal tiene el siguiente aspecto:

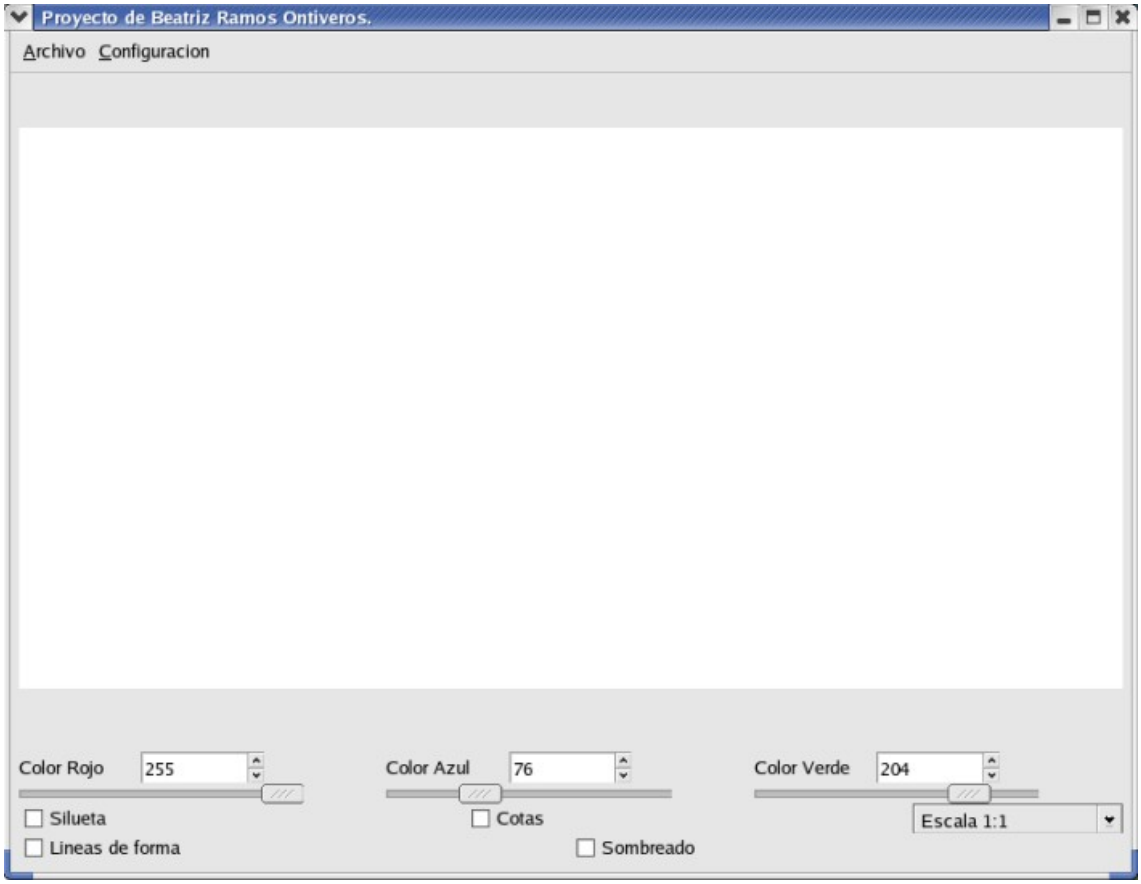


Figura 15. Ventana de la interfaz principal

La parte blanca central es el visor (QGLViewer), donde se irá viendo el modelo al cargarse y los cambios que se le vayan haciendo al mismo.

El menú archivo se encarga de ofrecer al usuario las acciones de abrir un archivo (cargar un modelo), guardar una captura de la imagen que hay en el visor en ese

momento haciendo uso de las funciones que ofrece la clase QGLViewer), cerrar un modelo (inicializando tanto el visor como todas las estructuras) y salir del programa.

El menú configuración permite cambiar el color de fondo del visor, el color de los números que muestran el tamaño de las cotas y el color que toma una cota cuando es seleccionada.

En la parte inferior de la interfaz principal podemos modificar la cantidad de rojo, azul y verde del modelo mostrado en el visor. También podemos activar/desactivar los modos de trabajo con silueta, líneas de forma, cotas y sombreado. Respecto a la escala, hay tres disponibles: 1:1, 1:10 y 1:20, ya que son las más usadas por los arqueólogos.

2.1.2.2. Ventana para el modo silueta

Al activar el modo silueta nos aparecerá una ventana con el siguiente aspecto.

Figura 16. Ventana del modo siluetas

En ella, tal y como se puede observar podemos cambiar el color y grosor de las siluetas, el efecto de dibujado y la aleatoriedad del mismo (que se explicarán en el apartado correspondiente a las siluetas, y por ahora sólo diremos que sirven para aumentar la sensación de que la silueta se ha dibujado a mano) y la nitidez de la silueta (que también se comentará más adelante). También esta ventana podemos activar o desactivar la visualización de una silueta estricta.

2.1.2.3. Ventana para el modo líneas de forma

La ventana que se muestra al activar el modo de líneas de forma es muy similar a la mostrada para las siluetas.

Figura 17. Ventana del modo líneas de forma

En ella, podemos modificar, también el color, el grosor y el efecto de dibujado junto con la aleatoriedad del mismo. También podemos modificar el umbral a partir del cual se considerará que existe una línea de forma (consultar el apartado correspondiente a las líneas de forma para entender a qué se refiere esto exactamente).

2.1.2.4. Ventana para el modo cotas

La ventana correspondiente al modo cotas es algo más compleja.

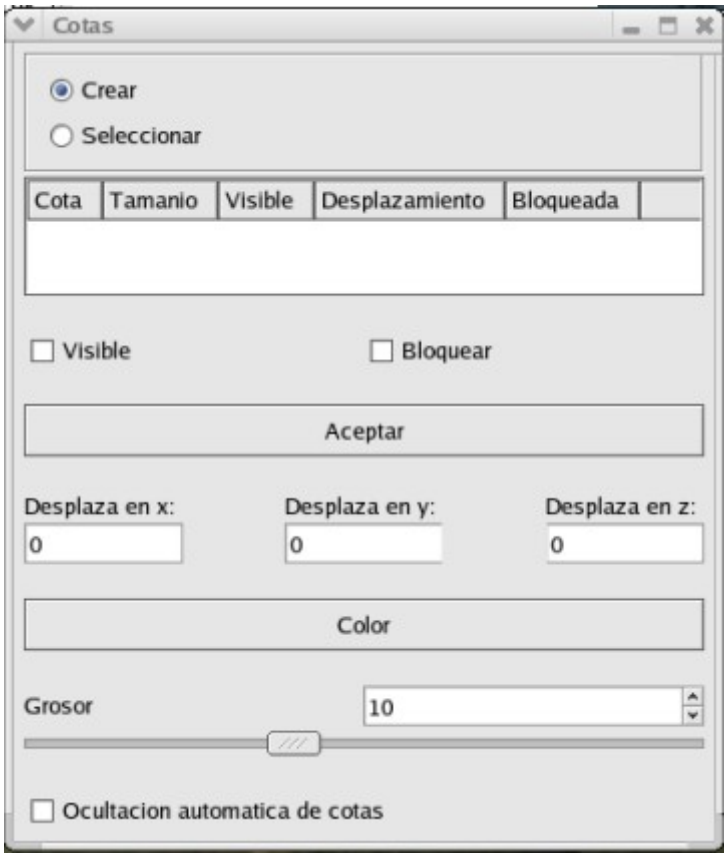


Figura 18. Ventana del modo cotas

Al igual que en las anteriores podemos modificar el color y grosor de las líneas, pero además podemos elegir entre el modo de creación de cotas y de selección de cotas (en la parte superior de la ventana) y podemos decidir si queremos mostrar las cotas siempre o permitir que se puedan ocultar por el modelo según corresponda a la perspectiva con la opción de ocultación automática de cotas. Debajo de los botones radio, se mostrará una lista con todas las cotas que se hayan creado en el sistema y sus características. Y justo debajo de esta lista tenemos lo necesario para modificar las propiedades de cada cota: si es (o no) visible, si va a estar bloqueada para evitar borrarla por error y si se desea que se dibuje con algún desplazamiento en *x*, *y* o *z*. Una vez puestas estas propiedades como el usuario desee, debe pulsarse el botón aceptar para que los cambios se almacenen y afecten realmente a la cota.

2.1.2.5. Ventana para el modo sombreado

La ventana para el sombreado tiene 4 zonas.

Figura 19. Ventana del modo sombreado

La parte superior corresponde a la elección del método para sombreado que vamos a usar: el de texturas o el computacional (en el apartado dedicado al sombreado se explican ambos detenidamente).

En la parte izquierda se pueden marcar las opciones con las que se trabajará en caso de haber elegido el método de sombreado con texturas. Lo primero a decidir en este caso es el número de texturas que vamos a usar: 3, 5, 7 o 9. Según el número de texturas se activarán más o menos botones. Cada botón abre un diálogo en el que indicar al programa dónde está la imagen para usar como esa textura. Además, para

evitar al usuario tener que elegir una a una las texturas, si se marca la opción de cargar un patrón completo, sólo será necesario indicar al programa dónde está una cualquiera de las texturas y él se encargará de cargar el resto (todas deben estar en la misma carpeta, tener formato .tiff o .tif y tener unos nombres con ciertas características; todo esto está explicado en el apartado correspondiente al sombreado).

La parte derecha de la ventana corresponde al sombreado del tipo computacional. Con él se puede modificar el color y grosor del sombreado. Además, se puede hacer un sombreado más o menos intenso modificando el número máximo de puntos por unidad de área. Respecto a los tipos de sombreado ofertados en este caso, hay dos: el de puntos y el de líneas (podemos elegir entre uno y otro en la lista desplegable). Para más información sobre cómo funcionan, se puede leer el apartado dedicado a la implementación del sombreado.

2.1.2.6. Ventana de ayuda

Al ejecutar el programa, se abre siempre automáticamente una ventana con ayuda sobre el manejo del programa con el siguiente aspecto.

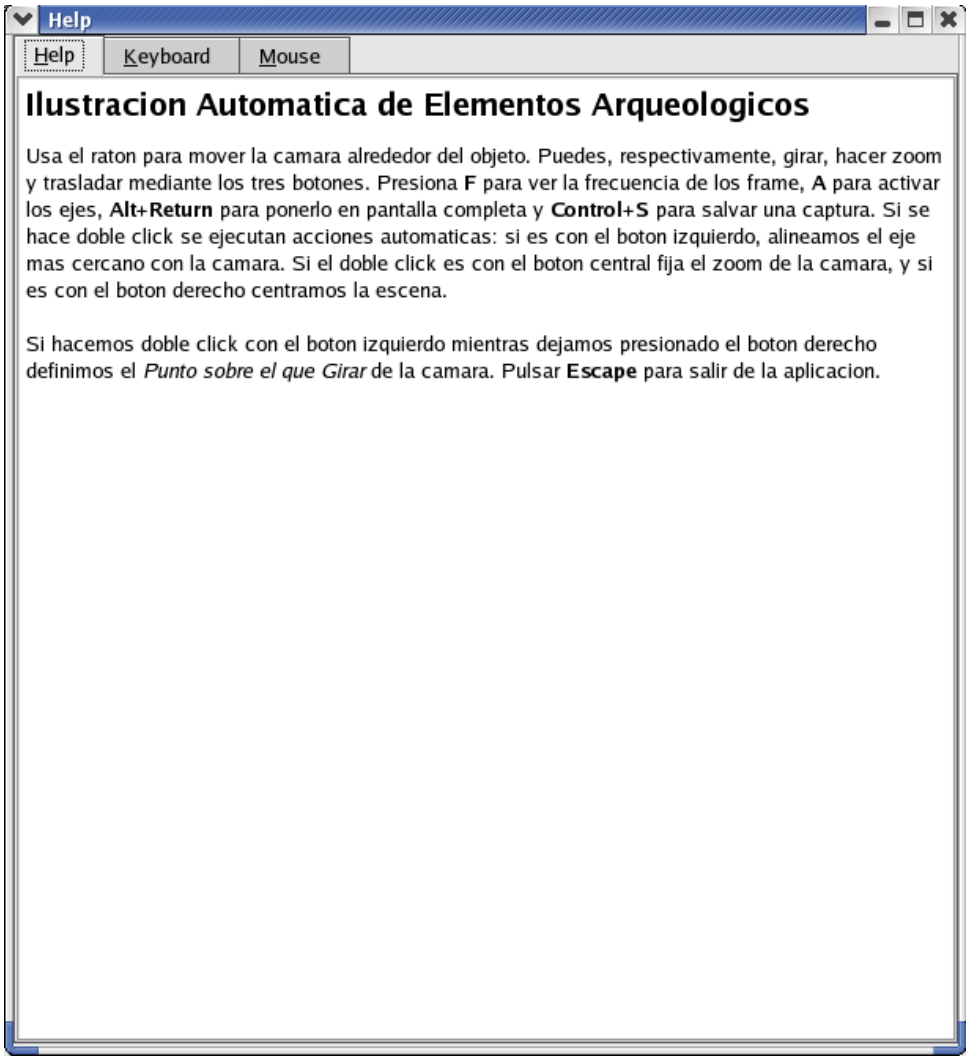


Figura 20. Ventana de ayuda

2.1.3. Silueta

Para comprender la implementación de las siluetas hay que preguntarse primero qué entenderemos por silueta:

Una silueta será una arista que separa una cara visible de una no visible. La cuestión radica ahora en descubrir qué caras son visibles y cuáles no visibles en un determinado momento. Para ello necesitaremos usar el concepto de VPN: El VPN (o View Plane Normal) define la orientación de la cámara, hacia dónde se mira. Su sentido es opuesto hacia dónde se mira.

Una vez aclarado esto, para saber si una cara es visible o no visible, podemos usar el producto escalar del VPN por la normal de la cara en cuestión. Podemos afirmar, entonces, que si el producto escalar es negativo la cara es visible, si es positivo es no visible y si es igual a 0 la cara es perpendicular al VPN (en nuestra aplicación, trataremos esa cara como no visible). En la siguiente imagen, extraída de [9], podemos ver a lo que nos referimos más claramente:

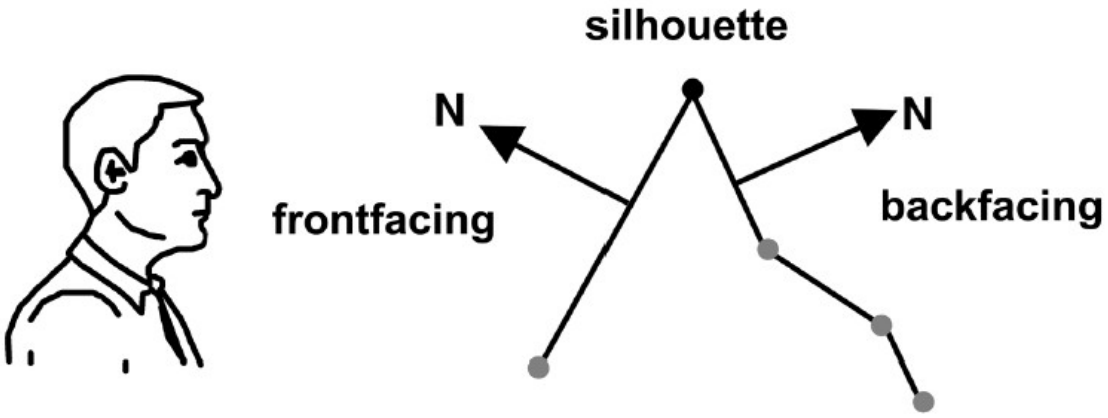


Figura 21. Ejemplo de silueta en un modelo poligonal. N son las normales de las caras, frontfacing las caras visibles y backfacing las caras no visibles.

Una vez que conocemos las caras visibles y las no visibles, es fácil decir qué aristas son parte de la silueta y cuáles no. Lo difícil radica en usar y ordenar la silueta como un único elemento, y no como un conjunto de aristas. El algoritmo, por lo tanto, actúa del siguiente modo:

```
for (todas las semiaristas pares){
        if (es silueta){
                Añadir_información_en_estructura(semiarista, siluetas);
                Añadir_información_en_estructura(opuesta, siluetas);
        }
}
```

Nota:

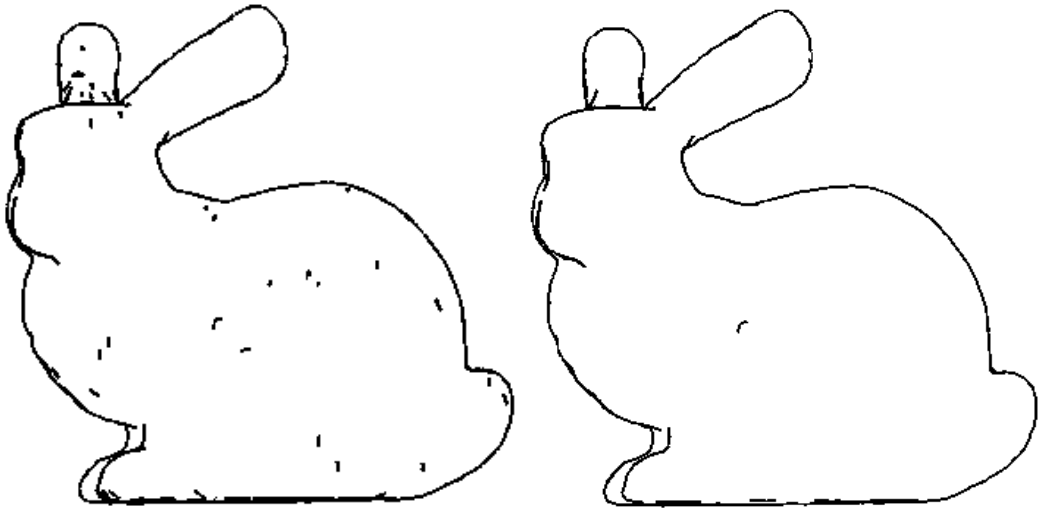
1- Si una semiarista es silueta, su opuesta también, por eso sólo comprobaremos las semiaristas pares.
2- La estructura donde metemos las semiaristas silueta no es más que un vector de semiaristas, y, al igual que antes, metemos cada semiarista directa en las posiciones pares y sus opuestas en las impares.

Tras tener la información de qué semiaristas son silueta, falta ordenarlas para tener de forma rápida la información completa de la silueta. Hay que tener en cuenta que el algoritmo anterior no sólo nos va a dar la silueta exterior de la figura, sino siluetas interiores (una figura en relieve puede tener caras no visibles por la parte central, y ahí marcará también silueta):

(a) **(b)**

Figura 22. En la imagen (b) se ven las siluetas detectadas para el modelo (a). Dentro del objeto puede verse que también hay siluetas.

En figuras complejas, además, tendremos muchas líneas pequeñas de silueta que no nos interesan.



(a) **(b)**

Figura 23. Las pequeñas siluetas que aparecen en el interior del objeto en la imagen (a) son irrelevantes y no debemos dibujarlas. El resultado correcto es el que muestra la imagen (b)

Puesto que sólo nos interesan las siluetas de gran tamaño que son las determinantes, usaremos ahora dos nuevas estructuras: una para ver el tamaño de cada silueta encontrada y localizar la primera semiarista por la que empieza (esta estructura nos localiza cada una de las siluetas existentes), y otra para ver cuál es la siguiente de una semiarista en una silueta.

Vector para identificar todas las siluetas	Tipo de dato	Descripción
p[0]	int (índice 0)	Tamaño de esa silueta.
p[1]	int (índice 1)	Primera semiarista de esa silueta

Vector para crear las siluetas	Tipo de dato	Descripción
p[0]	int (índice 0)	Identificador de esa semiarista.
p[1]	int (índice 1)	Identificador de su siguiente.

Una vez creadas estas dos estructuras, ya podemos identificar las siluetas existentes.

Veamos ahora cómo crear estas estructuras, ya que no es tan sencillo como puede parecer en un principio. Además hay distintas formas de hacerlo, ya que podemos hacerlo por el método de fuerza bruta (tomamos la primera semiarista y recorremos toda la lista hasta encontrar una siguiente, si hay, y así con todas) o podemos hacer uso de la información que nos dan las semiaristas aladas para hacer el programa más eficiente.

Sabemos que, si una semiarista es silueta, las semiaristas adyacentes a ésta también pueden ser siluetas. Si expresamos todas las semiaristas usando como referencia otra a la que llamaremos n:

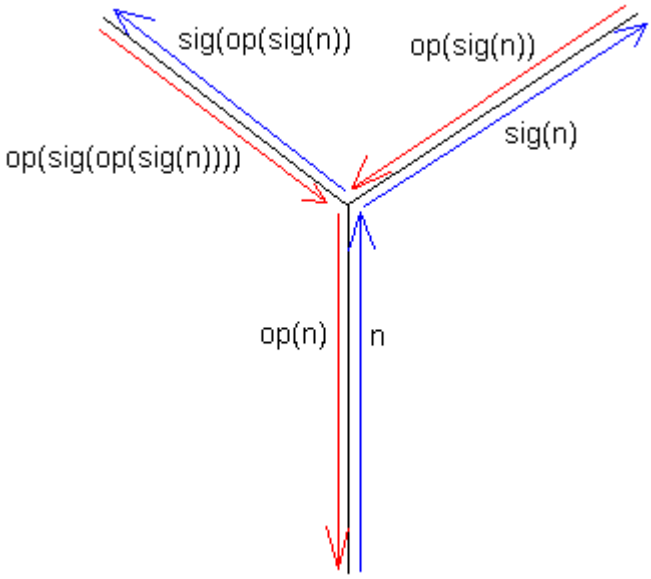


Figura 24. Ejemplo para calcular las semiaristas adyacentes

Hay que tener en cuenta, además, que una semiarista no tiene por qué tener sólo una adyacente, si no que puede tener muchas:

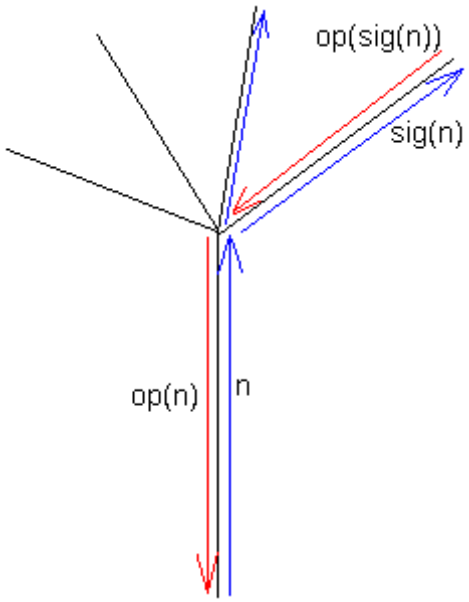


Figura 25. Una semiarista puede tener más de una adyacente

Una vez entendido esto, para ver cuáles son las adyacentes de una semiarista y, por lo tanto, cuáles son las siguientes en la silueta, usaremos el siguiente algoritmo:

```
for (todas las semiaristas de la lista anterior){
        Añadir_en_lista_siluetas(silueta, tamaño 1);
        while(silueta tenga más semiaristas adyacentes){
            while (siguiente(silueta) es silueta) {
                Aumentar_tamaño_en_1(silueta);
```

```
                    Actualizar_lista_siguientes(silueta, siguiente(silueta));
                    silueta=siguiente(silueta);
                }
                if (siguiente(opuesta(silueta)) es silueta) {
                    Aumentar_tamaño_en_1(silueta);
                    Actualizar_lista_siguientes(silueta, siguiente(silueta));
                }
                silueta=siguiente(opuesta(silueta));
            }
        }
```

Nota:

1- Siguiente(opuesta (silueta)) es una adyacente de silueta.
2- Sabemos que no hay más adyacentes cuando la semiarista que vamos a comprobar y la que tomamos inicialmente en el for son la misma.

Usando esta información, el dibujado de la silueta es muy rápido, ya que no es más que ir recorriendo las semiaristas. Algo a tener en cuenta antes de dibujar es, como se comentó antes, el tamaño de las siluetas. En la aplicación, no se dibujarán siluetas cuyo tamaño sea menor de 5.

También debemos tener en cuenta que, al dibujar la silueta, en el visor se verán tanto las líneas de delante como las de detrás, puesto que no hay nada que cubra las líneas traseras. Para evitar esto, dibujaremos el modelo con el mismo color que el fondo del visor. Así dará impresión de estar vacío sin que se aprecien las líneas traseras. El problema que tiene esto es que el modelo, al estar en 3D y dibujado en el mismo lugar que las líneas de la silueta, las cubre parcialmente, por lo que se ven mal. La solución que le podemos dar a esto es sencilla: aumentamos ligeramente el tamaño de la silueta en el sentido de la normal de las caras tal y como se puede apreciar en el siguiente ejemplo:

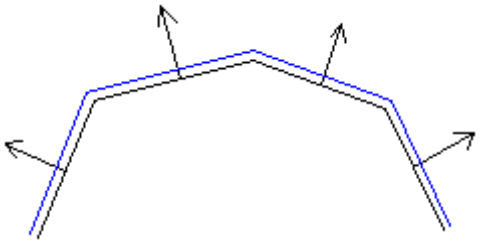


Figura 26. Se desplaza un poco la silueta en el sentido de las normales de las caras para evitar que las cubra el modelo 3D y no se vean.

Puesto que la silueta se verá mejor o peor según la alejemos más o menos del modelo según el caso, se ha provisto al usuario de una variable llamada nitidez que puede ser modificada por el mismo y que será la que aleje más o menos la silueta.

Puede que a alguien le suene extraño el que haya siluetas en la parte trasera del objeto, ya que se supone que allí no hay caras visibles, pero esto no es del todo cierto.

Aunque no son visibles para el ojo humano, el ordenador las puede computar como visibles, ya que esto dependerá del VPN y la normal de la cara. Esto se puede ver más claro en el siguiente ejemplo:

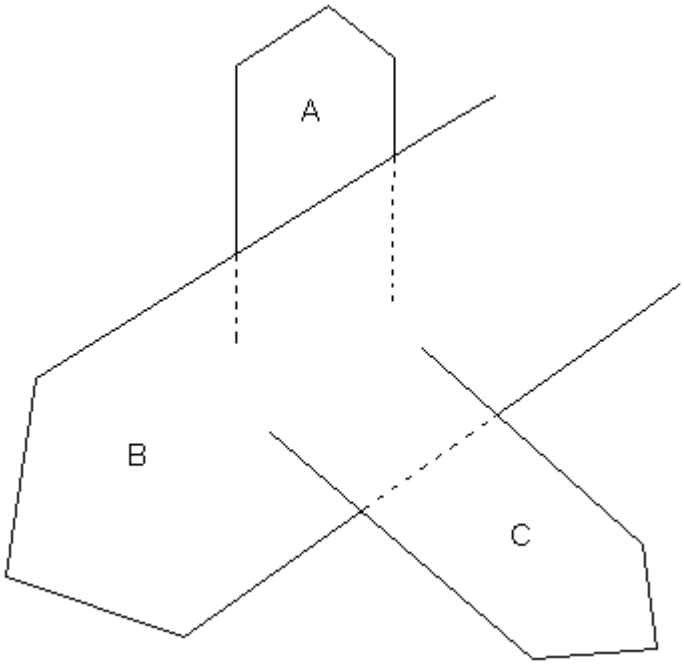


Figura 27. Hay siluetas (o parte de siluetas) que deben permanecer ocultas

Como se puede apreciar, la cara A no es totalmente visible, pero se computará como visible, por lo que las líneas punteadas de la misma serán visibles si no las cubrimos. Pero no sólo esto: aunque la cara B fuera más gruesa y cubriera a la A totalmente, mientras que el VPN no varíe el resultado computacional será el mismo y A seguiría tratándose como cara visible.

Por otro lado, si sólo queremos ver la parte externa de la silueta y no nos interesan las líneas que quedan en el centro del objeto (a lo que nos referiremos como silueta estricta), podemos solucionarlo pintando el modelo con el color de fondo pero en primer plano, es decir, en vez de pintarlo en el mundo en 3D tal y como hacíamos antes, lo pintamos en 2D en coordenadas de pantalla. En el siguiente ejemplo, se señala en azul lo que sería la silueta estricta, para comprender mejor a qué nos referimos:

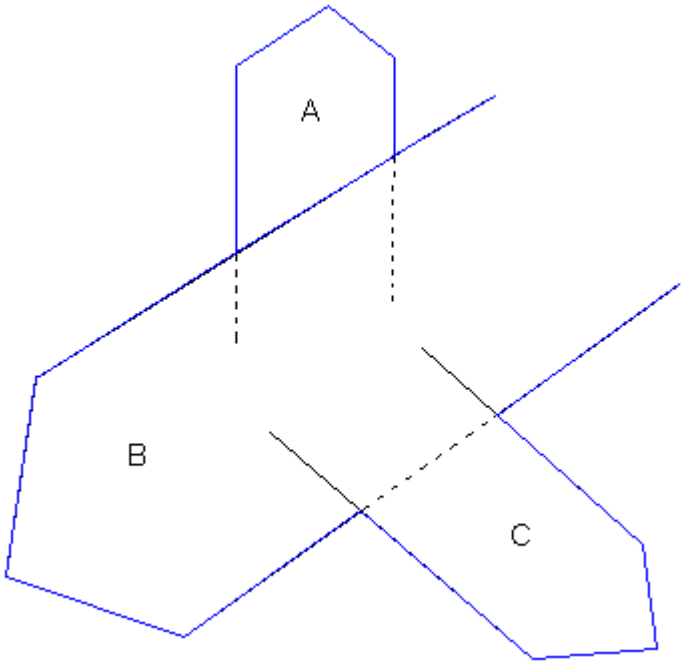
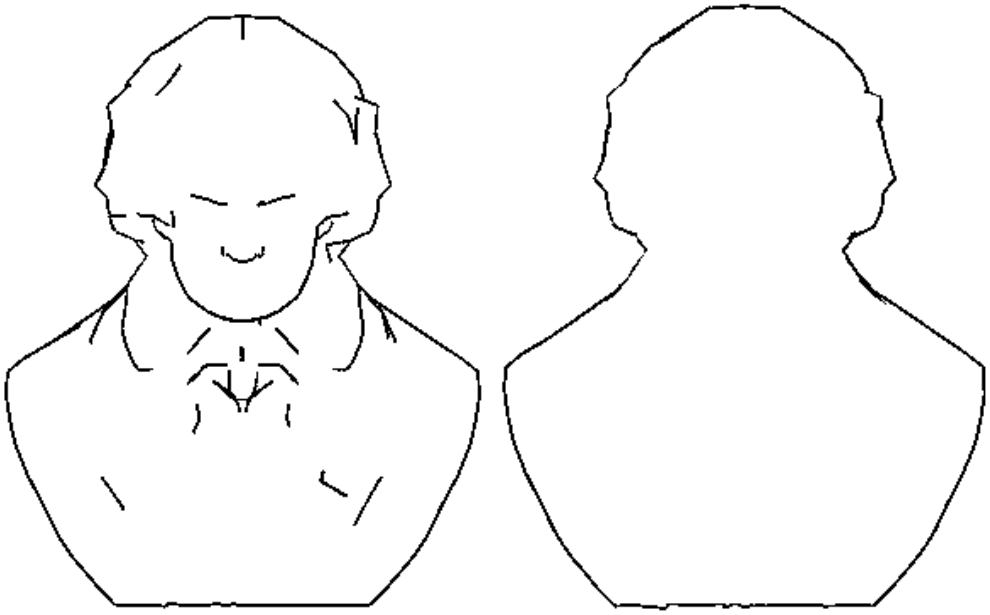


Figura 28. La silueta estricta es sólo la silueta exterior del objeto (marcada en azul)

Y aquí podemos ver un par de ejemplos conseguidos con la aplicación; el primero de ellos muestra la silueta normal y el segundo la silueta estricta:



(a) **(b)**

Figura 29. Diferencia entre la silueta normal (a) y la silueta estricta (b)

Respecto al efecto del dibujado, para dar un mayor realismo, trataremos tres tipos distintos:

- Pen-and-Ink: Con este efecto dibujaremos la silueta como si se hubiera hecho a mano, es decir, las líneas no serán rectas, oscilarán ligeramente. Para conseguir este efecto, usaremos aleatorios: uno para la coordenada *x*, otro

para la *y* y otro para la *z*. Estos aleatorios pueden ser mayores o menores, según los deseos del usuario, ya que se provee a éste de una variable en la que podrá indicar si quiere que la oscilación sea mayor o menor. Estos números los sumaremos o restaremos (de forma aleatoria también) a las coordenadas para obtener la oscilación adecuada que le dé al modelo impresión de dibujado manual. La silueta, en este caso, será continua, por lo que al dibujar dos líneas contiguas de la silueta, tendremos en cuenta que el segundo punto de la primera línea y el primero de la segunda deben coincidir, por lo que en la segunda línea sólo modificaremos el segundo punto (el primero será el mismo que el segundo de la primera línea).

- Lineado: Este efecto será parecido al anterior, excepto que la silueta no será continua, sino que cada línea de la silueta estará separada de la anterior. Para ello, usamos, igual que antes, números aleatorios. La única diferencia es que ahora, para dos líneas contiguas, no hacemos coincidir el primer punto de la segunda línea y el segundo de la primera, sino que modificamos aleatoriamente ambos puntos (si los aleatorios que obtenemos son 0 en ambos casos, entonces sí sería una línea continua).
- Mezclado: Este efecto es una mezcla de los dos anteriores: se elige aleatoriamente si para la semiarista actual vamos a usar el efecto pen-and-ink o el efecto lineado, dándole más posibilidades al primero, ya que así conseguimos mayor realismo, puesto que cuando una persona dibuja una silueta, es normal que lo haga sin levantar continuamente el lápiz del papel (pero también sin tenerlo siempre pegado a la hoja), y normalmente, no lo levantará demasiado.

Por supuesto, también se da la posibilidad de dibujar la silueta sin ningún efecto.

Referencias interesantes para estos temas son [5] y [6].

2.1.4. Líneas de forma

Estudiaremos ahora cómo funcionan las líneas de forma. Las líneas de forma son aquellas aristas que, como su nombre indica, marcan la forma de la figura. Veamos en los siguientes ejemplos a qué nos referimos:

(a) **(b)**

Figura 30. Imagen de las líneas de forma de un modelo (a) y las líneas de forma con la silueta (b)

Estas líneas no dependen del observador, sino del ángulo que formen las caras que las contienen. Si el ángulo es pronunciado, ahí hay una línea de forma.

En la aplicación, se calculan de una forma muy parecida al cálculo de las siluetas:

Lo primero es ver qué semiaristas son líneas de forma y cuáles no, y después se deben organizar del mismo modo que las siluetas para facilitar su tratamiento y dibujado. Veamos primero cómo saber si una semiarista es (o no) línea de forma.

```
for (todas las semiaristas pares){
        c1=cara incidente;
        c2=cara incidente de su opuesta;
        producto=producto_escalar(normal dec1, normal de c2);
        if(producto<umbral){
                Añadir_información_en_estructura(semiarista, líneas de forma);
                Añadir_información_en_estructura(opuesta, líneas de forma);
        }
}
```

Tras haber localizado las semiaristas que son líneas de forma, lo que queda por hacer es ordenarlas en estructuras similares a las de las siluetas:

Vector para identificar todas las líneas de forma	Tipo de dato	Descripción
p[0]	int (índice 0)	Tamaño de esa línea de forma.
p[1]	int (índice 1)	Primera semiarista de esa línea de forma

Vector para crear las líneas de forma	Tipo de dato	Descripción
p[0]	int (índice 0)	Identificador de esa semiarista.
p[1]	int (índice 1)	Identificador de su siguiente.

Al igual que ocurría con las siluetas, una línea de forma puede ser más grande (y estar formada por varias semiaristas) o ser pequeña y estar formada sólo por una, pero, de nuevo, como ocurría con las siluetas, las líneas de forma muy pequeñas no nos interesan, por lo que en el dibujado sólo aparecerán las que tengan un tamaño mayor de 5.

Para crear las dos estructuras de las líneas de forma el algoritmo es totalmente análogo al caso de las siluetas, y lo mismo ocurre con el dibujado de las líneas de forma, luego no se hará más hincapié en ello. Sólo indicar que, en el caso de las líneas de forma sólo hay que tener en cuenta la necesidad de ocultar las líneas de forma de la parte de atrás del modelo (nada parecido a lo que ocurría en el caso de la silueta estricta), por lo que sólo tendremos la necesidad de pintar el modelo en 3D con el color de fondo del visor, y no necesitaremos en ningún caso pintar el modelo en 2D. Además, al contrario de lo que ocurría con la silueta, las líneas de forma son siempre las mismas en el modelo (mientras no se cambie el umbral), no varían con la perspectiva, luego no hay que recalcularlas siempre que movamos el modelo, sino sólo al principio y cuando el usuario aumente o disminuya el umbral.

Respecto a los efectos del dibujado de las líneas de forma, son exactamente los mismos que en el caso de las siluetas.

Referencias interesantes para estos temas son, también [5] y [6].

2.1.5. Cotas

Para tener una visión realista de un objeto necesitamos conocer las medidas del mismo. Para esto, se permitirá al usuario trabajar con cotas: crearlas, modificarlas, etc.

Las cotas en esta aplicación serán líneas (con ciertas propiedades) que van desde un punto del objeto hasta otro. La estructura que se ha creado para las cotas tiene el siguiente formato:

Cota	Tipo de dato	Descripción
p1	punto	Punto origen de la cota.
p2	punto	Punto final de la cota.

Tam	float	Tamaño de la cota.
Color_unico	int[3]	Cada cota se identificará por un color único.
Visible	int	Indica si la cota es visible o no visible.
bloqueada	float[3]	Dice si la cota está no está bloqueada (se puede modificar) o si está bloqueada.
desplazamiento	int	Desplazamiento en *x*, *y* y *z*.

Todas las cotas que se creen en la aplicación estarán almacenadas en un vector de cotas. Para colocar cotas en el modelo, el usuario indicará a la aplicación que desea trabajar con las cotas activando el modo cotas.

Al activarse dicho modelo, cada clic del ratón cobra una gran importancia. Al producirse un clic, captamos las coordenadas sobre las que se ha pinchado y, según el botón que haya sido pulsado (izquierdo o derecho), la aplicación dará una u otra respuesta. Cuando el usuario pincha con el botón izquierdo se indica a la aplicación que se quiere crear una cota (si estamos creando cotas) o que se desea seleccionar una cota para modificarla (si estamos en el modo de selección de cotas). Por el contrario, el botón derecho sólo funciona cuando se están creando cotas e indica que la cota sobre la que se ha pinchado se quiere eliminar. Por lo tanto, en caso de haber pinchado con el botón izquierdo, necesitaremos conocer el color del punto sobre el que se ha hecho clic. Puesto que todas las caras tienen un color único y diferente del de las otras caras, podemos saber sobre cuál de todas las caras se ha pinchado viendo el color del punto (y si se ha pinchado fuera del objeto lo sabremos porque el color del píxel no coincidirá con el de ninguna cara). De este modo, conocemos el primer punto de la cota. Para ver el segundo punto, el método es idéntico. Cuando hemos seleccionado sólo un punto podemos observar una línea que va desde dicho punto hasta la flecha del ratón. Esta línea es una ayuda visual para ver cómo quedará la cota tras dar el segundo punto. Para dibujarla, lo único a tener en cuenta es que debemos conocer en todo momento la posición del ratón. Esto es sencillo de hacer captando el evento de movimiento de ratón que se produce y que nos proporciona las coordenadas del ratón. Tras colocar el segundo punto (o punto destino de la cota), se calcula automáticamente el tamaño de la cota mediante la distancia euclídea entre el punto de inicio y de fin de la misma, y se le asigna a la cota un color único para poder identificarla (el método para hacer esto es el mismo que en el caso de las caras, y debemos tener en cuenta que este color único será distinto del de todas las caras y del de las otras cotas). Por defecto, la cota se almacena en el vector de cotas como visible, no bloqueada y con desplazamiento 0. Si el usuario quiere modificar estas propiedades debe seleccionar la cota en cuestión y modificarlas en la ventana correspondiente a las cotas (se explicará como más adelante en este mismo apartado).

Por otro lado, cuando el usuario pincha con el botón derecho sobre una cota se comprueba el color del píxel sobre el que se ha hecho clic. Si este color coincide con el de alguna cota (y la cota no está marcada como bloqueada), ésta es eliminada tanto del visor como de la lista de cotas. Si no coincide con el de ninguna cota pero teníamos previamente seleccionado el primer punto de una cota, éste punto se pierde. Esto será útil en caso de que el usuario se haya equivocado al marcar el primer punto de una cota.

De este modo, no tendrá que colocar una cota y borrarla, sino que podrá desechar la cota que estaba creando y empezar una nueva.

Una vez visto esto, pasaremos a ver cómo funciona el modo de selección de cotas. Para seleccionar una cota el usuario debe hacer clic izquierdo sobre ella. La aplicación tomará el color del píxel sobre el que se ha pinchado y comparará dicho color con el de las cotas existentes hasta encontrar la seleccionada (o ver que no se ha pinchado sobre ninguna). Tras seleccionar una cota, para modificar los valores de visible, bloqueada y desplazamiento el usuario debe marcar lo que desee en la ventana de las cotas e indicar al programa que aplique los cambios. Si una cota no es visible, el usuario no puede interactuar con ella haciendo clic sobre la misma, ya que no se pinta en el visor y, por lo tanto, no la ve. Para solucionar este problema, en la ventana para las cotas aparece una lista con todas las cotas existentes sobre el modelo. El usuario puede seleccionar ahí la cota haciendo clic izquierdo sobre ella y tratar con la misma del mismo modo que si la hubiera seleccionado como se ha explicado antes.

Puesto que identificamos dónde hemos pinchado haciendo uso de los colores únicos, necesitamos que el modelo se dibuje con estos colores al hacer clic y no con los colores que haya definido el usuario. Por lo tanto, cuando se detecta que se ha hecho clic con el ratón el modelo se dibuja con los colores únicos el tiempo que tarda la aplicación en procesar la información y hacer los cálculos apropiados. En cuanto termina, el modelo vuelve a dibujarse con sus colores normales. En los modelos pequeños y medianos esto es tan rápido que no se aprecia. Con las cotas ocurre lo mismo: no sólo el modelo se dibuja con sus colores únicos, también las cotas. Éstas, además, se dibujan con grosor 5 en lugar de con el grosor definido por el usuario (también momentáneamente) para facilitar la selección de las mismas tanto para interactuar con ellas como para eliminarlas, ya que si las cotas tienen un grosor muy pequeño es difícil hacer clic con el ratón justo sobre un píxel de la misma, y es mucho más cómodo cuando la cota tiene un mayor grosor.

Dependiendo de lo que esté haciendo el usuario en un momento determinado puede interesarle que las cotas queden ocultas por el modelo (según la perspectiva) o que sean visibles siempre. Por esto, se puede elegir en cada momento la visualización de las mismas. La única diferencia en ambos casos es que, si queremos que se vean siempre, las dibujamos en primer plano, delante del modelo, y si queremos que puedan ser ocultadas por el modelo, las dibujaremos en 3D al igual que el resto del modelo (igual que ocurría en el caso de querer una silueta estricta, como se explicó en el apartado 2.1.3).

Para más información sobre el método de identificación de elementos mediante colores únicos se puede consultar [7].

2.1.6. Sombreado

Para el sombreado se usan dos métodos distintos. Los resultados con ambos son igualmente interesantes y, dependiendo del modelo y del gusto del propio usuario, puede que se prefiera usar uno u otro método.

El primero de los métodos oferta un sombreado de puntos y otro de líneas. Sólo requiere que el usuario elija entre uno de los dos, pero no es necesario indicarle ningún patrón o textura a usar. Veamos cómo funciona éste método:

Tanto en el caso de los puntos como en el de las líneas, lo primero será calcular puntos aleatorios sobre cada cara. Una unidad cuadrada del modelo no podrá tener más de cierto número de puntos, variable que puede modificar el usuario a su gusto. Debemos tener en cuenta que un número de puntos alto se corresponde con una cara oscura, es decir, una cara de baja intensidad de luz. Por lo que la intensidad de una cara será determinante para ver cuántos puntos se dibujarán sobre ella. Pero no será esto lo único que afecte, ya que, si tenemos dos caras con la misma intensidad, y una tiene el doble de tamaño que la otra, necesitaremos colocar sobre la misma el doble de puntos para conseguir el mismo efecto.

Para el cálculo de la intensidad de cada vértice usaremos el modelo de iluminación local (MIL) que se describe a continuación:

Tendremos dos luces, una ambiental blanca y otra direccional, también blanca, que vendrá desde arriba a la izquierda, ya que es dónde colocan la luz los arqueólogos. Para cada cara, la intensidad de cada uno de los tres colores (rojo, verde y azul) en un vértice será:

$I = I_{ambiental} + I_{difusa} + I_{especular}$

Donde:

$I_{ambiental} = k_a * rs * fuente\_ambiental.rgb$

siendo k_a el coeficiente de reflexión ambiental, una constante con valor 0.3, rs el color del modelo y fuente_ambiental.rgb el color de la fuente ambiental.

$I_{difusa} = fuente\_direccional.rgb * k_d * rs * (N \cdot L)$

siendo k_d el coeficiente de reflexión difusa, una constante con valor 0.8, rs el color del modelo, fuente_direccional.rgb el color de la fuente direccional y N·L el producto escalar de N y L. Llamaremos N a la normal de la cara y L a la dirección de la luz direccional.

$I_{especular} = fuente\_direccional.rgb * k_s * (V \cdot R)$

siendo k_s el coeficiente de reflexión especular, una constante con valor 0.5, fuente_direccional.rgb el color de la fuente direccional y V·R el producto escalar de V y R. Llamaremos V al vector (0,0,1), que apunta hacia el espectador, y R = (2 * (N·L) * N – L) / modulo (N). R será, por lo tanto, el vector reflejado de L respecto de la normal N. Veamos de dónde sale su valor:

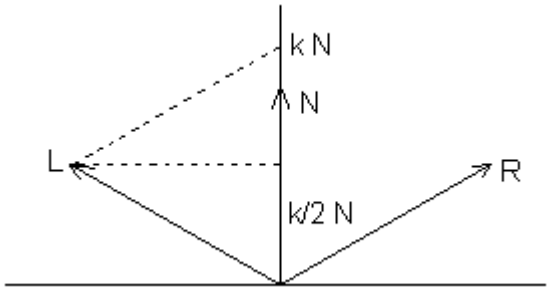
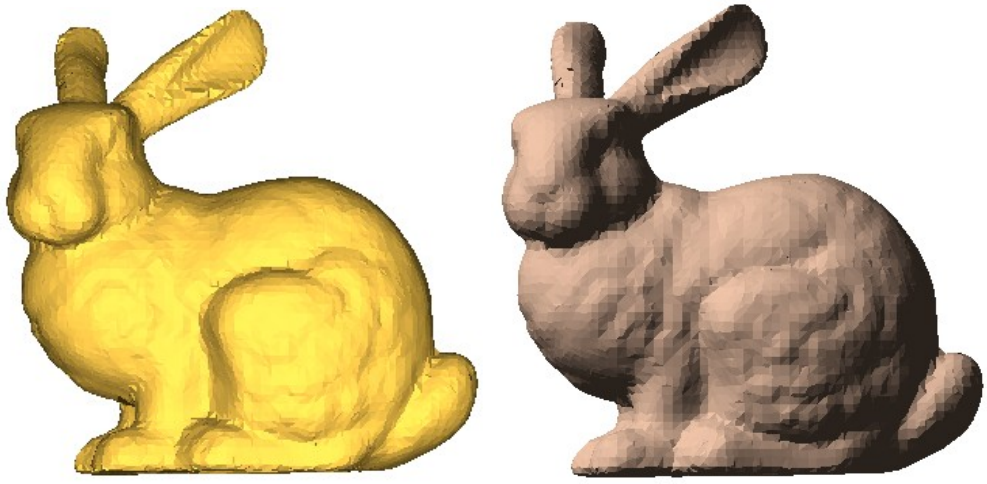


Figura 31. Esquema explicativo sobre la obtención de R

Como N está en la bisectriz de L y R, entonces el vector L + R tiene la misma dirección (aunque distinto módulo), que N. Por esto, L + R = k * N. Además, la proyección de L sobre N es igual a (L·N) * N, y también igual a (k/2) * N. Por lo tanto, k = 2 * (L·N), y así tenemos que L + R = 2 * (L·N) * N, luego: R = 2 * (L·N) * N – L

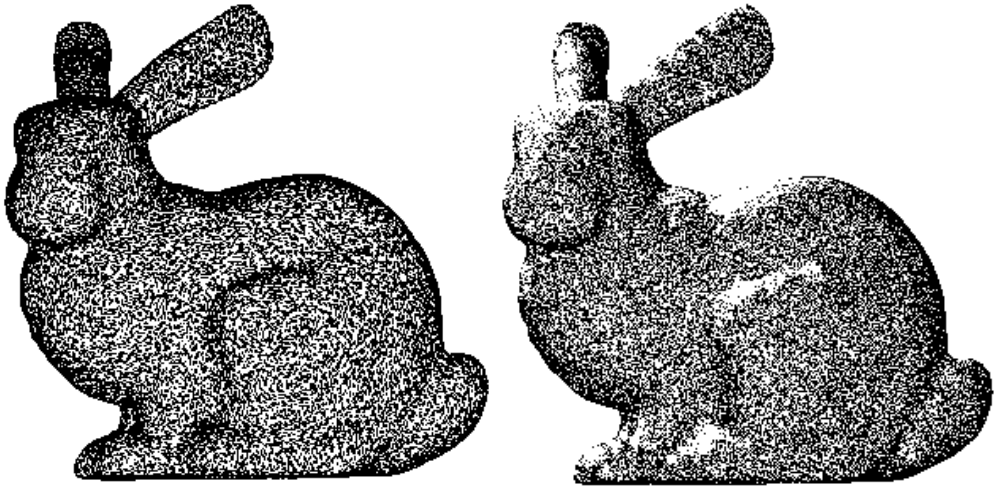
Según vamos calculando la intensidad de cada vértice, se va indicando en la estructura correspondiente que esa intensidad es válida y, más adelante, sólo trabajaremos con las intensidades marcadas como válidas.

Antes de continuar, es necesario aclarar que se hace uso de una luz direccional para resaltar más el sombreado, ya que con una luz posicional el sombreado resultante era muy homogéneo.



(a) **(b)**

Figura 32. Ejemplo de un modelo con luz posicional (a) y el mismo modelo (aunque en otro color) con luz direccional (b).



(a) **(b)**

Figura 33. En estos dos ejemplos tenemos un sombreado a partir de una luz posicional. En cada uno de ellos se trató de exagerar el sombreado de un modo distinto.

Tras haber calculado la intensidad de cada color en cada vértice, pasamos los colores a escala de grises (esto no se dibuja, simplemente son cálculos que hace internamente la aplicación) haciendo la media del rojo, verde y azul (los sumamos y dividimos entre 3). Tras pasarlo todo a escala de grises, interpolamos para asegurarnos de que todas las intensidades están en el rango [0,1] (y no en el, por ejemplo [0.24, 0.82], ya que es importante que las intensidades sean lo más extremas posibles para que el sombreado destaque). Para ello, a cada intensidad le restamos el valor mínimo de todas las intensidades y el resultado de esto lo dividimos por el tamaño del rango (que será la diferencia entre el valor máximo de las intensidades y el valor mínimo).

A continuación, se calculan los puntos para el sombreado de cada cara, pero antes hablaremos de las estructuras que se usan para el sombreado:

Vector para puntos del sombreado	Tipo de dato	Descripción
s	Vector de puntos	Introduciremos aquí los puntos que calculemos para el sombreado según la intensidad.
npsom	int	Número de puntos del vector

Vector para el estado de los puntos del sombreado	Tipo de dato	Descripción
s	Vector de enteros	Nos indica el estado de cada punto del sombreado.

Usando estas dos estructuras, en la aplicación tendremos:

- Un vector con los puntos de cada cara (un vector de la primera estructura explicada). Así cada cara tiene asociado tanto el número de puntos de su sombreado como los puntos del sombreado.
- Un vector con las líneas de todas las caras: cuando se calculan los dos puntos de una línea (más adelante se explicará cómo) ambos puntos se introducen en este vector, primero uno y justo a continuación el otro; luego para dibujar las líneas no hay más que unir cada punto de un índice par con su siguiente de un índice impar.
- Vector con todos los puntos sueltos: al crear las líneas es normal que queden puntos sueltos (bien porque estén impares bien por el método que se usa). Estos puntos se introducen en este vector.
- Vector con el estado del sombreado de cada cara (un vector de la segunda estructura explicada). Así sabemos de forma rápida si un determinado punto de una cara ha sido unido para formar una línea o no. Esto es necesario para el método de cálculo de las líneas.

El algoritmo para calcular el sombreado de las caras es el siguiente (la intensidad de la cara la tomaremos como la intensidad de alguno de los vértices de la misma; otra opción sería tomarla como la media de los tres vértices):

Para cada cara:

1- Vemos cuáles son los tres vértices de cada cara (simplemente, siguiendo la estructura de semiaristas, del mismo modo que se calculan para el dibujado).
2- Calculamos el número de puntos de la cara. Este número (que pasaremos, claro está, a entero), se calcula como:

Num_puntos = (1 – inten_cara) * ((max_puntos * area_t) / area_unid))

Siendo:

- Inten_cara la intensidad de la cara (nos interesa 1 – la intensidad porque la intensidad alta corresponde a pocos puntos y la baja a muchos puntos).
- Max_puntos es el máximo de puntos por unidad de área.
- Area_t es el área del triángulo.
- Area_unid es el área unidad, que se calcula como el área total del modelo dividido entre 100 (el área total del modelo no es más que la suma de las áreas de todas las caras).

Es fácil ver que uno de los problemas de este método es que, si el número de puntos máximo por unidad de área es bajo y las caras son pequeñas, podemos obtener un número de puntos igual a 0 en muchas de las caras (sobre todo con intensidades altas), con lo que tendríamos un sombreado casi nulo. Para solucionarlo, cuando se detecta que esto ha ocurrido (el resultado es 0 puntos pero 1-intensidad es mayor de 0.3) se le pone 0 puntos o 1 punto a la cara de forma aleatoria pero asegurándonos de que sea más posible que tenga un punto si la intensidad es más baja. Para ello, se calcula un aleatorio entre 0 y 1 y se suma a la intensidad. Si el número resultante es menor que 1 se coloca un punto, y si es mayor o igual que 1 no se coloca ningún punto.

Además, para reajustar el sombreado logrando más oscuridad en las zonas más oscuras se suma al número de puntos de la cara 2 más si 1-intensidad es mayor o igual que 0.8, y 1 si está entre 0.7 y 0.8.

3- Una vez calculado el número de puntos de la cara, calculamos tantos puntos aleatorios dentro de la misma como nos indique dicho número. Para ello, hacemos uso de coordenadas baricéntricas:
   - Calculamos dos aleatorios entre 0 y 1 a los que llamaremos p_a y p_b. Si la suma de ambos aleatorios es mayor que 1, modificamos el valor de los mismos por $1-p_a$ y $1-p_b$.
   - Tras esto, calculamos un tercer número p_c como $1-p_a-p_b$.
   - Y las coordenadas del punto aleatorio dentro de la cara vienen dadas por:
     - $x = p_a * v0.x + p_b * v1.x + p_c * v2.x$
     - $y = p_a * v0.y + p_b * v1.y + p_c * v2.y$
     - $z = p_a * v0.z + p_b * v1.z + p_c * v2.z$

     siendo v0, v1 y v2 los tres vértices del triángulo y refiriéndonos a vi.x como coordenada x del vértice i.
4- Con los puntos obtenidos se actualiza la información de las estructuras y variables comentadas anteriormente.

Si sólo queremos dibujar los puntos, el resto a partir de aquí es muy sencillo: simplemente, para cada cara se dibujan los puntos aleatorios que hemos calculado (y para que no se vean los puntos de las caras traseras del modelo, se dibuja el modelo en 3D del color de fondo del visor como se ha comentado en puntos anteriores).

Sin embargo, si queremos dibujar las líneas antes tendremos que calcularlas. Veamos cómo:

Para cada cara:

1- Tomamos un punto de la cara que no haya sido unido aún (para esto necesitábamos el vector que nos indica el estado de los puntos).
2- Un punto se puede unir con otro de su misma cara o de otra cara que está como mucho a una cierta distancia (en caras) predefinida a la que llamaremos profundidad máxima. Esta profundidad se calcula como el número total de caras del modelo entre 4. Por lo tanto, primero habrá que ver de qué profundidad será la cara en la que se encuentra el punto con el que uniremos.
3- Para ello, calculamos un aleatorio entre 0 y 1. Si es menor que 0.5 nos quedamos en esa cara (empezamos partiendo de la misma cara en la que está el primer punto de la línea) y si es mayor o igual que 0.5 pasaremos a alguna de las caras adyacentes (profundizaremos 1 cara). Repetimos el punto 3 hasta que tengamos (según la decisión aleatoria) que dejar de profundizar o alcancemos la profundidad máxima predefinida. Para entender mejor el tema de las profundidades, a continuación se presenta un ejemplo en el que se marcan las profundidades de cada cara:

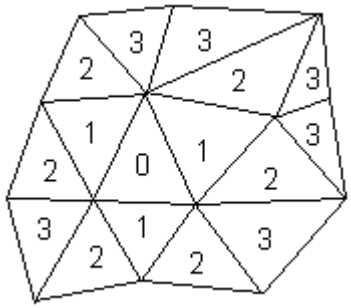


Figura 34. Ejemplo sobre la profundidad. Si partimos de la cara 0, las marcadas como 1 están a profundidad 1 con respecto a ésta; las marcadas como 2, a profundidad 2, etc.

4- Si la profundidad a la que debemos buscar el segundo punto es 0 la cara será la misma en la que estamos y si no, hay que calcular la cara adecuada. Para ello:
   - 4.1. Calculamos todas las adyacentes de la cara actual. Serán adyacentes con una cara las que tengan una arista en común con ella. Luego, para calcular las adyacentes de una cara, vemos cuáles son las tres semiaristas de la cara actual (simplemente, siguiendo la estructura de semiaristas aladas). Y, para ver qué caras comparten la semiarista con nuestra cara, vemos las caras incidentes de las semiaristas opuestas.
   - 4.2. Elegimos una aleatoriamente (distinta de aquella por la que hemos venido) y la tomamos como actual.
   - 4.3. Repetimos el proceso hasta alcanzar la profundidad que se obtuvo en el punto 3.
5- Ahora, buscamos el primer punto no unido aún de la cara que hemos obtenido en el paso 4 y lo unimos con nuestro primer punto.
6- Actualizamos el vector de estado y el de las líneas, y repetimos todo desde el paso 1 hasta haber unido todos los puntos de la cara.

Si nos fijamos en el método, podemos observar que, en el caso de que la cara con la que nos toca unir tenga ya todos los puntos unidos nuestro punto inicial quedará suelto. En estos casos, anotaremos el mismo en el vector de puntos sueltos.

Una vez que ya han sido calculadas las líneas, las dibujamos tomando como primer punto de cada línea una posición par del vector y como segundo par la posición impar inmediatamente posterior.

Puesto que hay puntos sueltos puede dar la impresión de que el sombreado es más claro que en el caso de los puntos. Para evitar esta sensación dibujamos también los puntos sueltos. Puesto que están todos introducidos en un vector los vamos dibujando de uno en uno sin mayor complicación.

Y, al igual que antes, para evitar que se transparente el sombreado de las caras traseras se dibuja el modelo en 3D con el color de fondo del visor.

En modelos en los que sus caras formen ángulos pronunciados el efecto de este sombreado puede resultar extraño. Esto ocurre por que cuando una línea une dos puntos de dos caras el resultado puede quedar por dentro del modelo. Por ejemplo:

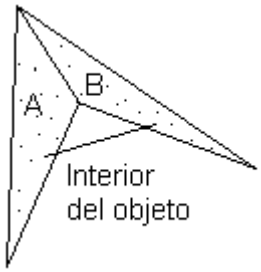


Figura 35. En este ejemplo, por el ángulo que forman las caras A y B, la línea que une dos puntos entre ellas queda dentro del objeto y no se ve.

Veamos un ejemplo real de lo que puede ocurrir en un modelo con estas características:

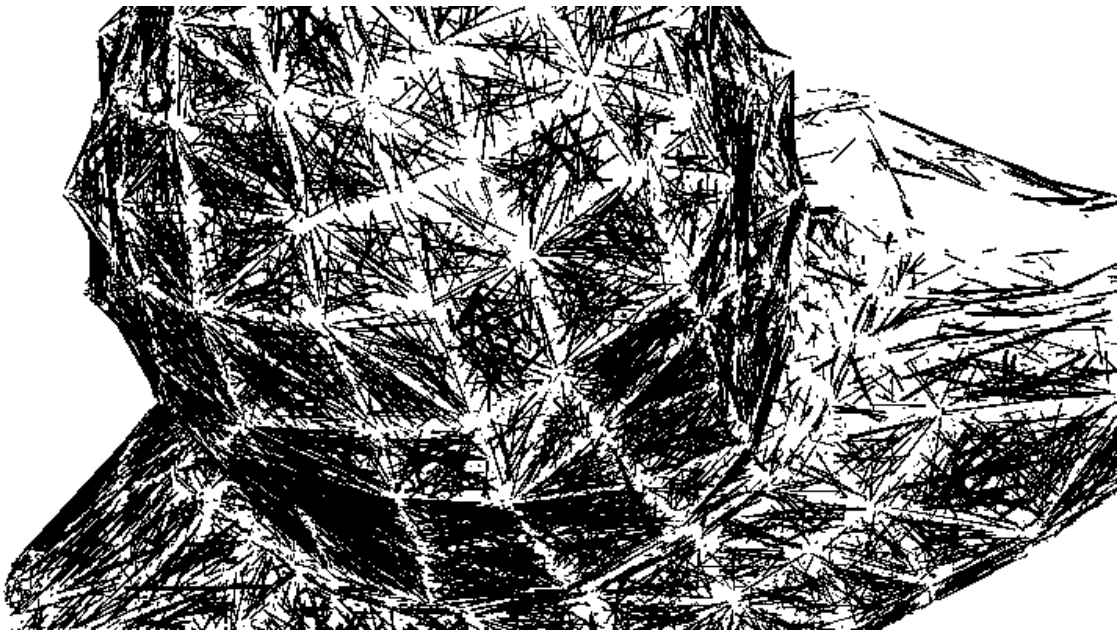


Figura 36. Ejemplo real del problema explicado anteriormente

Por esta razón, el sombreado de líneas mediante este método no es el más indicado para este tipo de objetos.

El segundo método que se usa para el sombreado está basado en texturas. Para sombrear un modelo se pueden usar hasta 9 texturas diferentes a la vez, y un mínimo de 3. Las texturas deben haber sido creadas por el usuario (no las calcula el programa), y son imágenes con extensión .tif o .tiff y con un tamaño de 256 x 256 píxeles. La aplicación ofrece un patrón de texturas por defecto que no debe ser eliminado ni movido de la carpeta en la que está. Un patrón de texturas se refiere a un grupo de texturas que están en la misma carpeta. Por lo general, si queremos usar una textura de, por ejemplo, puntos, lo que debemos hacer es crear, por lo menos, una textura con muy pocos puntos

(para zonas con alta intensidad), otra con muchos puntos (para zonas con baja intensidad) y otra intermedia (para zonas con intensidad media). Si queremos suavizar los cambios, podemos añadir más texturas (5, 7 o 9 en total). Si usamos 5 texturas, además de las 3 comentadas al principio se deben añadir dos: una con una densidad entre la de baja y la de media y otra entre la de media y la de alta. Si usamos 7 se añaden dos entre la de baja y la de media y dos entre la de media y la de alta (una de ellas más clara y la otra más oscura) y si queremos usar 9 se añaden 3 en cada lado.

Tras esta explicación, pasaremos a ver cómo funciona todo computacionalmente.

La primera parte es exactamente igual que para el otro método: se calcula la intensidad de cada cara y se pasa a escala de grises. Luego se hace la interpolación tal y como se explicó antes.

Tras esto, se cargan las texturas que se vayan a usar. Para ello, primero las leemos mediante las librerías libtiff. Hay dos formas de indicarle a la aplicación cuáles son las texturas que vamos a usar:

- La primera es, en la ventana del sombreado, marcar el número de texturas que vamos a usar e indicarle al programa para cada intensidad dónde está la imagen que debe utilizar mediante los diálogos que aparecen al pulsar los botones activos.
- La segunda es, también en la ventana del sombreado y tras marcar el número de texturas deseadas, elegir la opción cargar patrón completo. Luego, se pulsa sobre cualquiera de los botones activos y se marca una de las texturas. Automáticamente, el programa se encargará de cargar todas las demás texturas de la carpeta correctamente. Para poder usar esta opción, es necesario que las imágenes tengan unos nombres determinados. A continuación se muestra un cuadro en el que se indica el tipo de nombre para cada textura (cada textura es, de arriba abajo, más oscura que la siguiente pero menos que la anterior, debe estar en la misma carpeta que la que se le indique al programa y debe tener alguna de las extensiones indicadas anteriormente):

Características que debe cumplir el nombre de la textura	Significado
Contiene la palabra baja en su nombre (sin seguir por ningún número). Ejemplo: textura_baja_defecto.tiff	Textura muy densa (intensidad muy baja).
Contiene la palabra media3 en su nombre. Ejemplo: textura_media3_defecto.tiff	Textura entre densa e intermedia.
Contiene la palabra media2 en su nombre. Ejemplo: textura_media2_defecto.tiff	Textura entre densa e intermedia.
Contiene la palabra media1 en su nombre. Ejemplo: textura_media1_defecto.tiff	Textura entre densa e intermedia.
Contiene la palabra media en su nombre (sin seguir por ningún número). Ejemplo:	Textura intermedia.

textura_media_defecto.tiff	
Contiene la palabra alta3 en su nombre. Ejemplo: textura_alta3_defecto.tiff	Textura entre la intermedia y la muy poco densa.
Contiene la palabra alta2 en su nombre. Ejemplo: textura_alta2_defecto.tiff	Textura entre la intermedia y la muy poco densa.
Contiene la palabra alta1 en su nombre. Ejemplo: textura_alta1_defecto.tiff	Textura entre la intermedia y la muy poco densa.
Contiene la palabra alta en su nombre. Ejemplo: textura_alta_defecto.tiff	Textura muy poco densa (intensidad muy alta).

Inicialmente, el programa sólo carga 3 texturas. Si el usuario le indica que quiere más pero no le dice qué texturas debe usar, el programa usará el número posible más cercano al elegido por el usuario. Es decir, si está marcada la opción de usar 7 texturas pero el usuario sólo le ha indicado 6, el programa podrá usar 3 o 5, según las 6 indicadas. Para entender por qué 3 o 5, debemos saber que las 3 básicas (alta, media y baja) si no las da el usuario las toma el programa de las de por defecto, luego estas 3 siempre estarán. No ocurre lo mismo con las otras texturas. En el caso de querer usar 7 texturas, además de esas 3 se requieren 4 más (las llamaremos, según la tabla anterior, alta1, alta2, media1 y media2). En el caso de 5 texturas, hay que darle al programa, además de las básicas, 2 más (alta1 y media1). Por lo tanto, si vamos a usar 7 texturas pero sólo se le han indicado al programa las texturas alta, baja, media, alta2, media2 y alta1, por ejemplo, la aplicación no puede usar ni 7 texturas ni 5 (le falta la información sobre la textura media1) por lo que dibujará el sombreado con 3 texturas solo. Pero si se dan las texturas alta, baja, media, alta1, media1 y alta2, por ejemplo, sí tiene las texturas que necesita para cubrir un sombreado con 5, luego usará 5 texturas.

Para el dibujado de este sombreado, se calcula cada cara con la textura que le corresponda según su intensidad. La correspondencia textura – intensidad podemos observarla en la siguiente tabla (tener en cuenta que referenciaremos cada textura con la palabra que debe aparecer en su nombre, y que la intensidad siempre estará, por lo explicado previamente, entre 0 y 1):

Número de texturas	Nombre de textura	Intensidad
3	baja	<0.4
	media	0.4<=Intensidad<=0.6
	alta	>0.6
5	baja	<0.2
	media1	0.2<=Intensidad<0.4
	media	0.4<=Intensidad<0.6
	alta1	0.6<=Intensidad<0.8
	alta	>=0.8
7	baja	<0.2
	media2	0.2<=Intensidad<0.3
	media1	0.3<=Intensidad<0.4
	media	0.4<=Intensidad<0.6
	alta2	0.6<=Intensidad<0.7
	alta1	0.7<=Intensidad<0.8

	alta	>=0.8
9	baja	<0.15
	media3	0.15<=Intensidad<0.25
	media2	0.25<=Intensidad<0.35
	media1	0.35<=Intensidad<0.45
	media	0.45<=Intensidad<0.55
	alta3	0.55<=Intensidad<0.65
	alta2	0.65<=Intensidad<0.75
	alta1	0.75<=Intensidad<0.85
	alta	>=0.85

Pasaremos ahora a hablar de las coordenadas de textura usadas. Para poder visualizar correctamente la textura debemos asignar a cada vértice las coordenadas de textura apropiadas. Estas coordenadas determinan qué texel en el mapa de texturas se asigna a cada punto (un texel es la unidad mínima de una textura aplicada a una superficie). El rango para estas coordenadas es desde el 0.0 hasta el 1.0 (siendo el primero el mínimo y el segundo el máximo). Si nuestras caras fueran cuadradas podríamos usar las coordenadas de textura del siguiente modo:

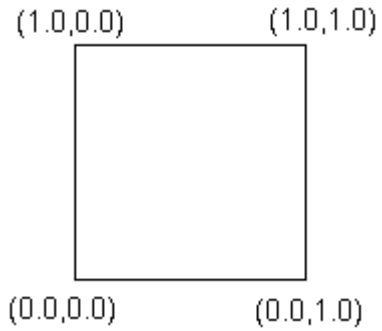


Figura 37. Coordenadas de textura para caras cuadrangulares

Vemos que se aplican como si estuviéramos hablando de coordenadas cartesianas. En nuestro caso, no son cuadrados sino triángulos. Las coordenadas que se han usado son, por lo tanto:

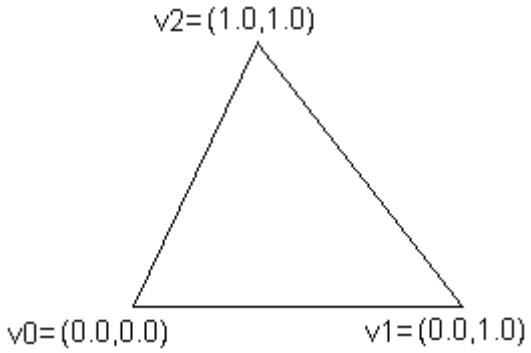


Figura 38. Coordenadas de textura para caras triangulares

v0 se refiere al primer vértice del triángulo, v1 al segundo y v2 al tercero.

Con este método no es necesario dibujar el modelo en 3D con el color del visor de fondo para cubrir el sombreado trasero, ya que las texturas cubren toda la cara y no dejan ver la parte de detrás de la figura.

Una vez explicados ambos métodos sólo queda decir que el sombreado no se recalcula continuamente, sino únicamente cuando lo solicita el usuario pulsando el botón adecuado de la ventana del sombreado.

Proyecto Informático

5º Ingeniería Informática

Ejemplos obtenidos con la aplicación

Curso 2006 – 2007

I. Ejemplos obtenidos con la aplicación

En esta sección vamos a mostrar diversas imágenes obtenidas con la aplicación de distintos modelos. Todos los modelos usados aquí pueden encontrarse sin mucha dificultad en la red.

El primero de estos modelos que veremos se llama rockerArm.ply. Está almacenado en binario (binary_big_endian) y tiene 40177 vértices y 80354 caras.

Aquí podemos ver cómo es la pieza:

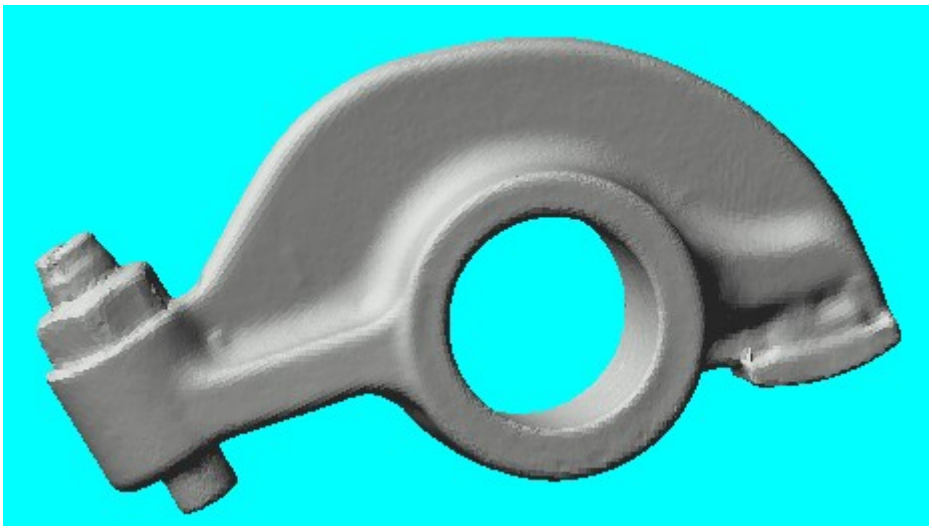


Figura 39. Modelo de rockerArm.ply

Al calcular la silueta, la imagen obtenida es la siguiente:

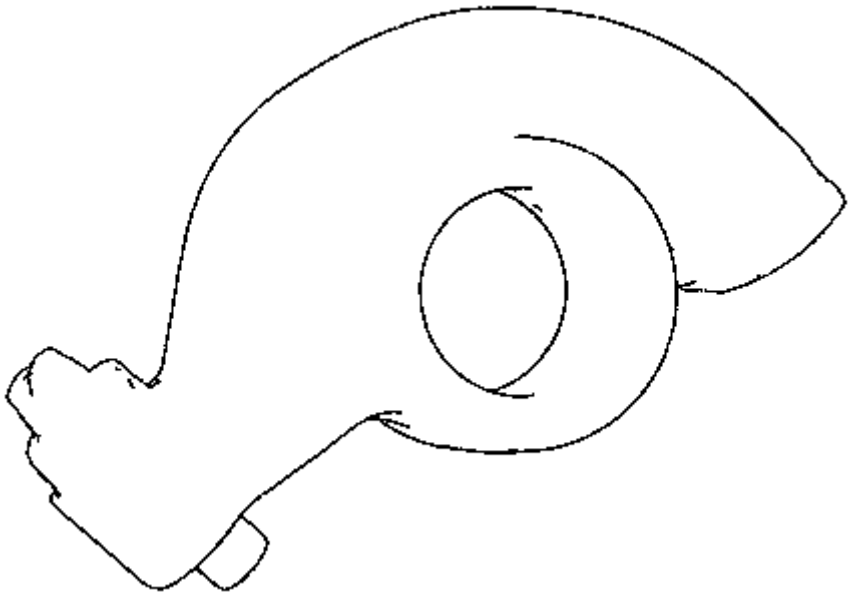


Figura 40. Silueta de rockerArm.ply

Si mostramos tanto la silueta como las líneas de forma:

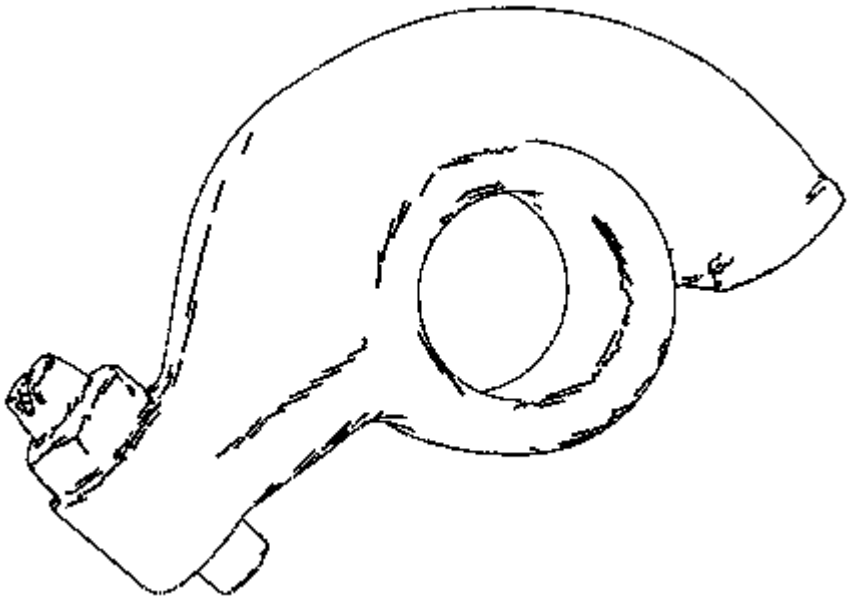


Figura 41. Silueta y líneas de forma de rockerArm.ply

Sombreando con texturas:

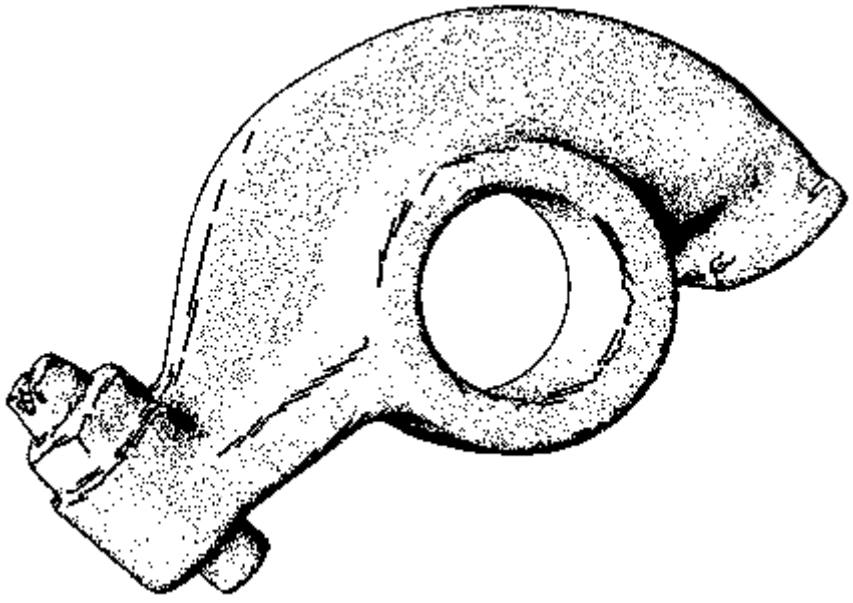


Figura 42. Sombreado con texturas de rockerArm.ply

Si sombreamos con el método computacional:

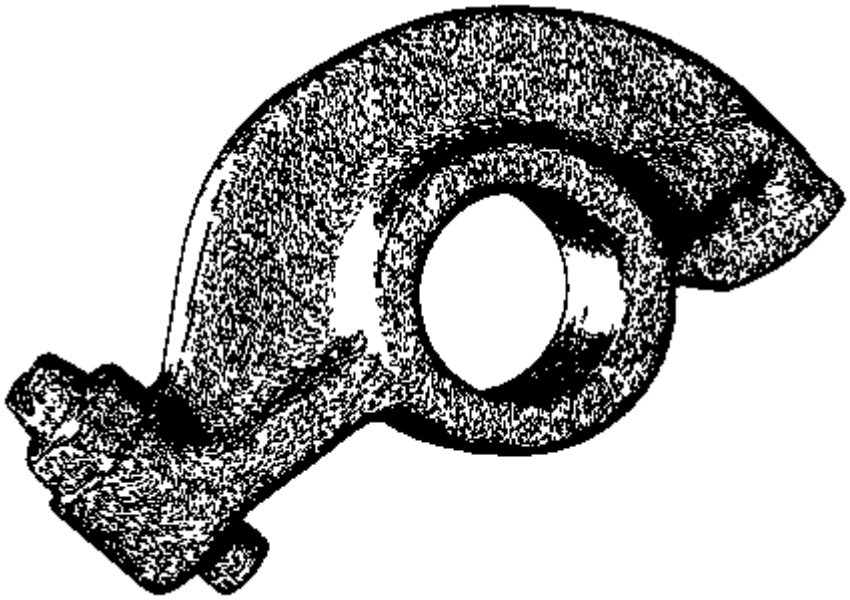


Figura 43. Sombreado computacional de rockerArm.ply

Otra toma con el computacional desde otra posición:

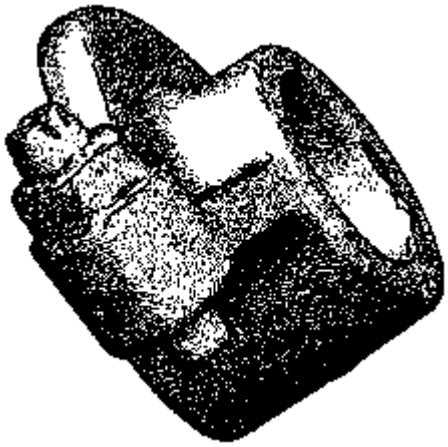


Figura 44. Sombreado computacional de rockerArm.ply

Aquí se han puesto unas cuantas cotas a la imagen con la silueta y las líneas de forma:

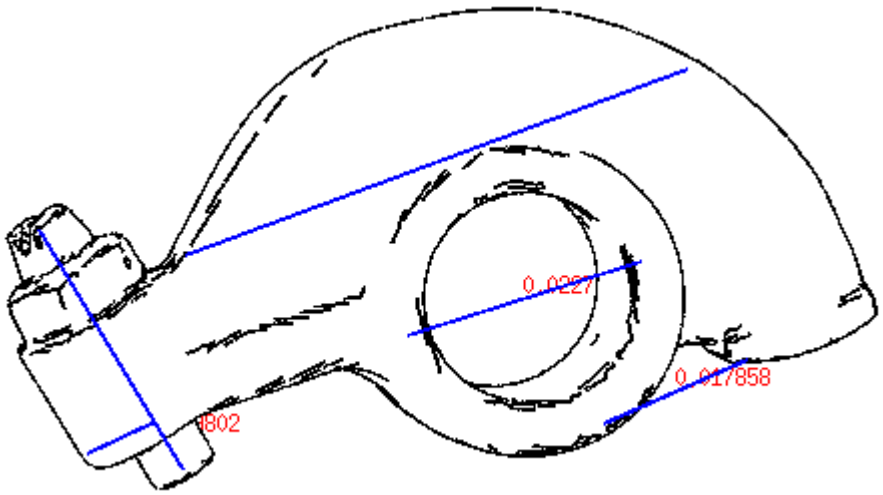


Figura 45. Ejemplo de cotas con siluetas y líneas de forma en rockerArm.ply

En esta otra imagen podemos observar las cotas con el modelo 3D normal. Una de las cotas resalta porque ha sido seleccionada:

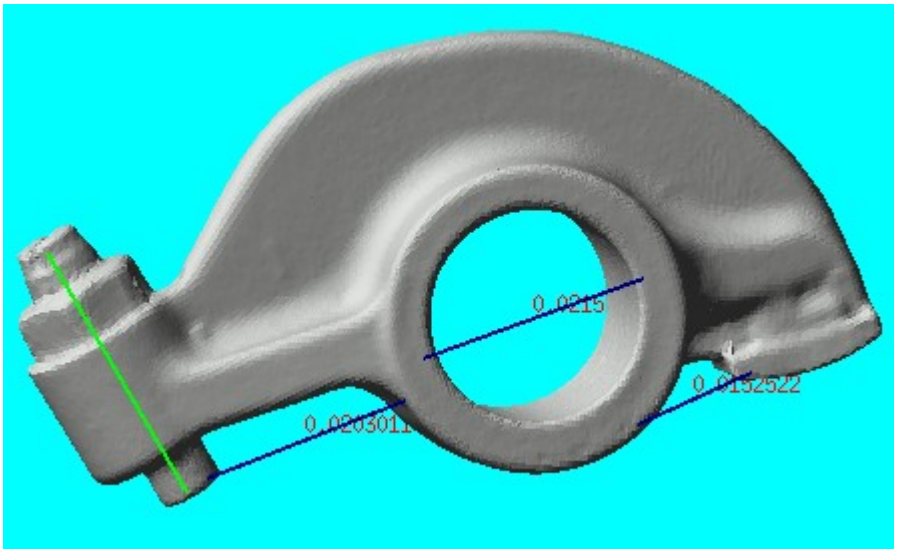


Figura 46. Ejemplo de cotas con cota seleccionada en rockerArm.ply

En la ventana de cotas podemos observar la lista de cotas existentes:

Cota	Tamanio	Visible	Desplazamiento	Bloqueada
0	0.0215727	Si	0,0,0	No
1	0.0203011	Si	0,0,0	No
2	0.0256653	Si	0,0,0	No
3	0.0152522	Si	0,0,0	No

Figura 47. Vista de la ventana de cotas para el ejemplo anterior

El siguiente modelo que mostraremos se llama oberschenkel.ply. Su formato de almacenamiento es ascii y tiene 14174 vértices y 27953 caras.

La pieza en 3D tiene el siguiente aspecto:

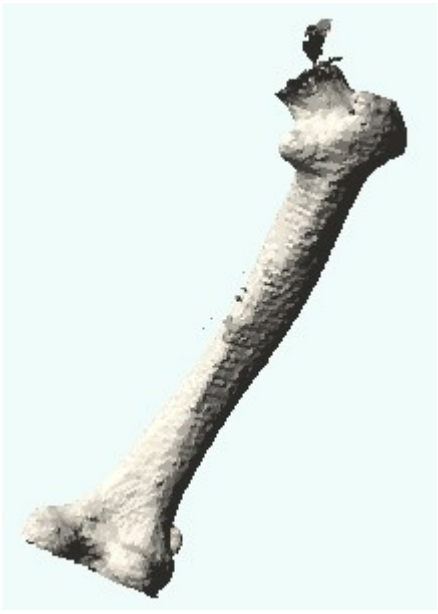


Figura 48. Modelo de oberschenkel.ply

Con las siluetas y las líneas de forma:

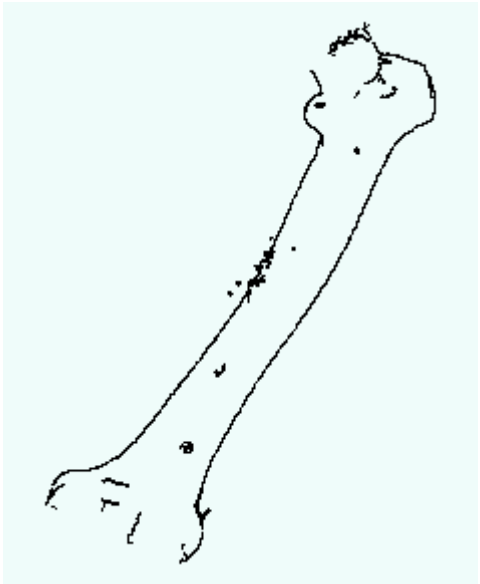
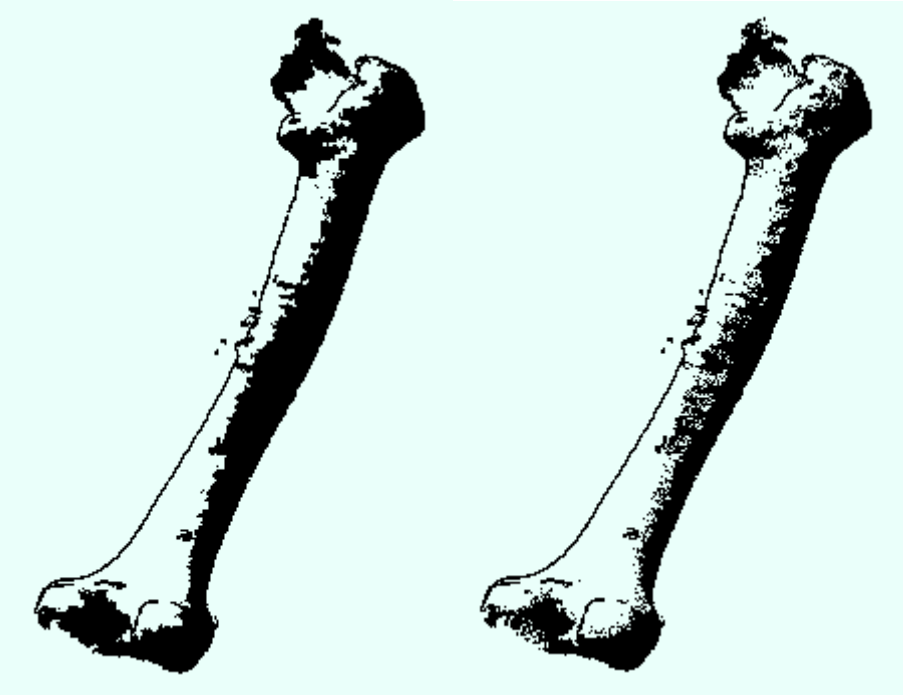


Figura 49. Siluetas y líneas de forma en oberschenkel.ply

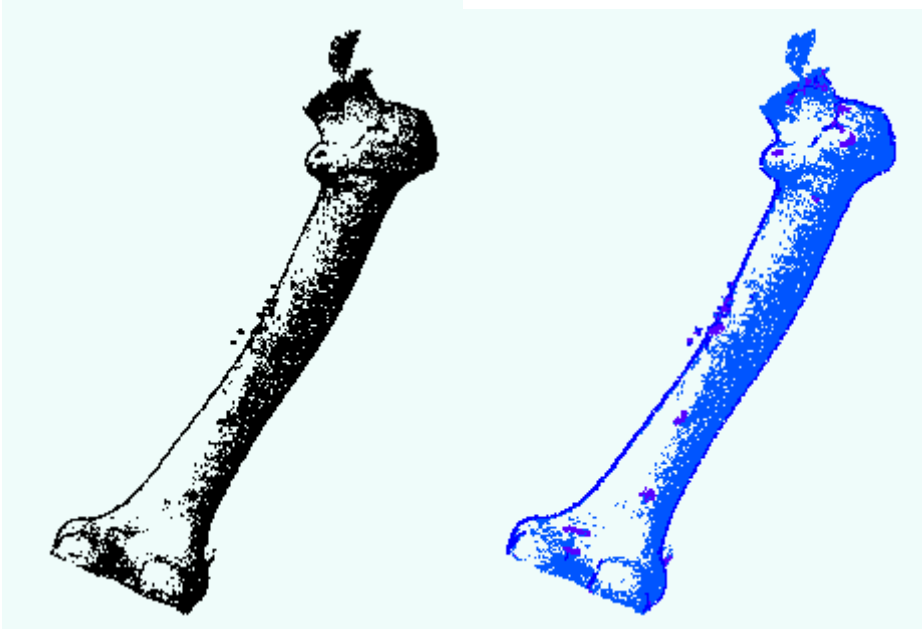
El sombreado de texturas con 3 texturas (Figura 50.a) y con 9 (Figura 50.b):



(a) **(b)**

Figura 50. Sombreado con 3 texturas (a) y con 9 (b) en oberschenkel.ply, incluyendo silueta y líneas de forma

Y el computacional, con distinto color tanto de sombreado como de silueta y líneas de forma:



(a) **(b)**

Figura 51. Sombreado computacionalcon incluyendo siluetas y líneas de forma con distintos colores en oberschenkel.ply

El tercer modelo que vamos a ver se llama aegyptian.ply. Tiene formato binario (binary_little_endian), 82480 vértices y 162056 caras. Se trata de una pieza arqueológica real escaneada con un escáner 3D.

Su modelo 3D tiene el siguiente aspecto:

Figura 52. Modelo de aegyptian.ply

La figura con las siluetas y las líneas de forma:

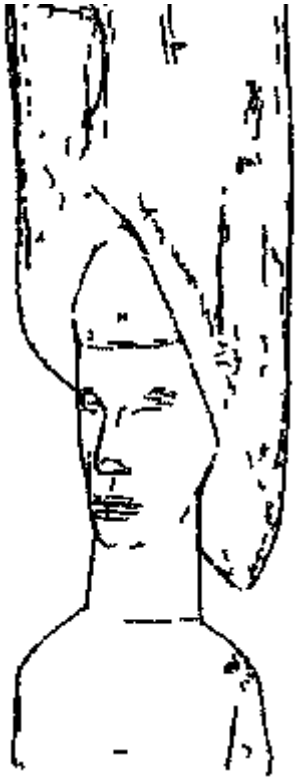


Figura 53. Siluetas y líneas de forma en aegyptian.ply

La visión con los sombreados de 9 texturas (Figura 54.a) y computacional (Figura 54.b):

(a) **(b)**

Figura 54. Sombreado con 9 texturas (a) y computacional (b) en aegyptian.ply, con siluetas y líneas de forma

Con algunas cotas puestas:

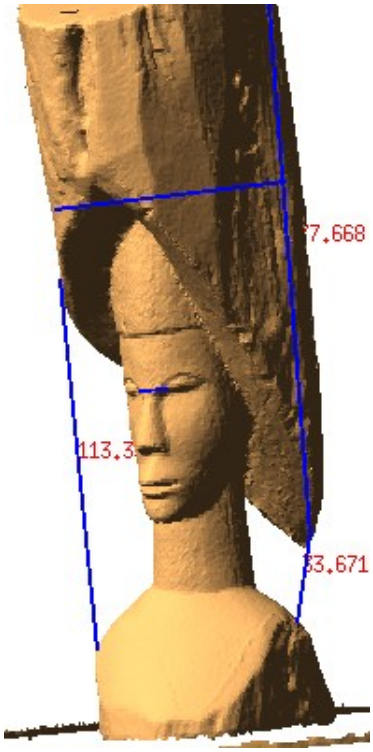


Figura 55. Ejemplo de cotas en aegyptian.ply

Como cuarto modelo veremos fertility.ply. Este modelo también se almacena como binario (binary_little_endian). Tiene 241607 vértices y 483226 caras.

El modelo 3D tiene la forma siguiente:

Figura 56. Modelo de fertility.ply

La silueta del mismo (Figura 57) y con las líneas de forma (Figura 58):

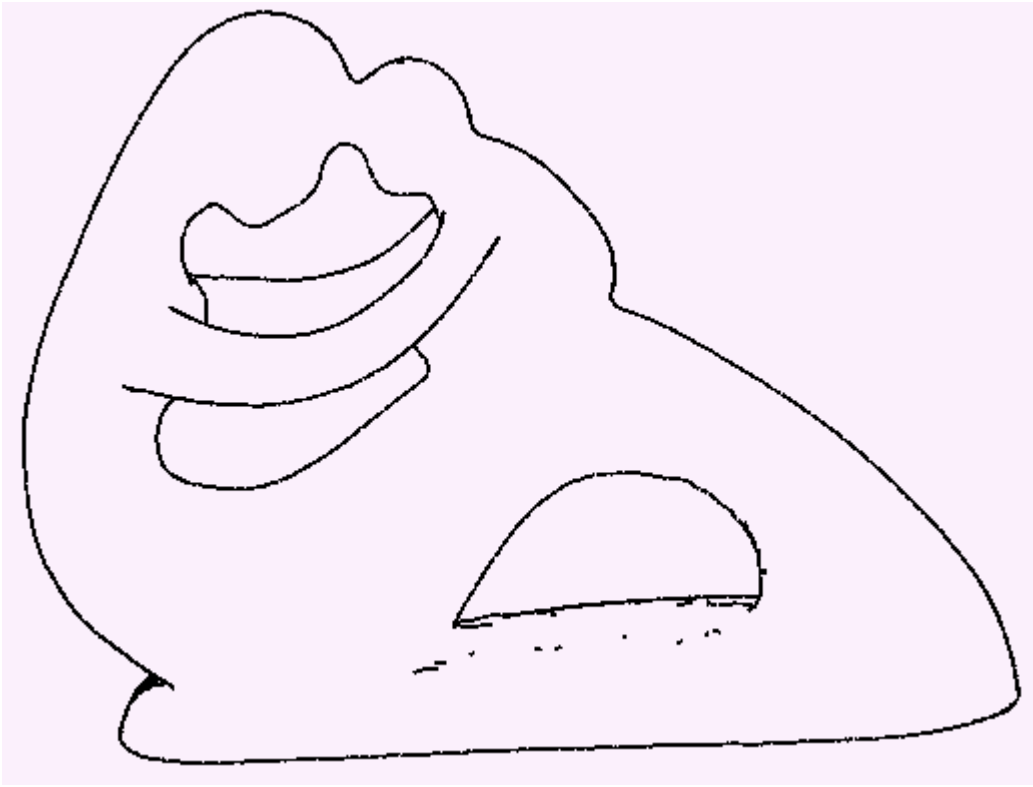


Figura 57. Siluetas de fertility.ply

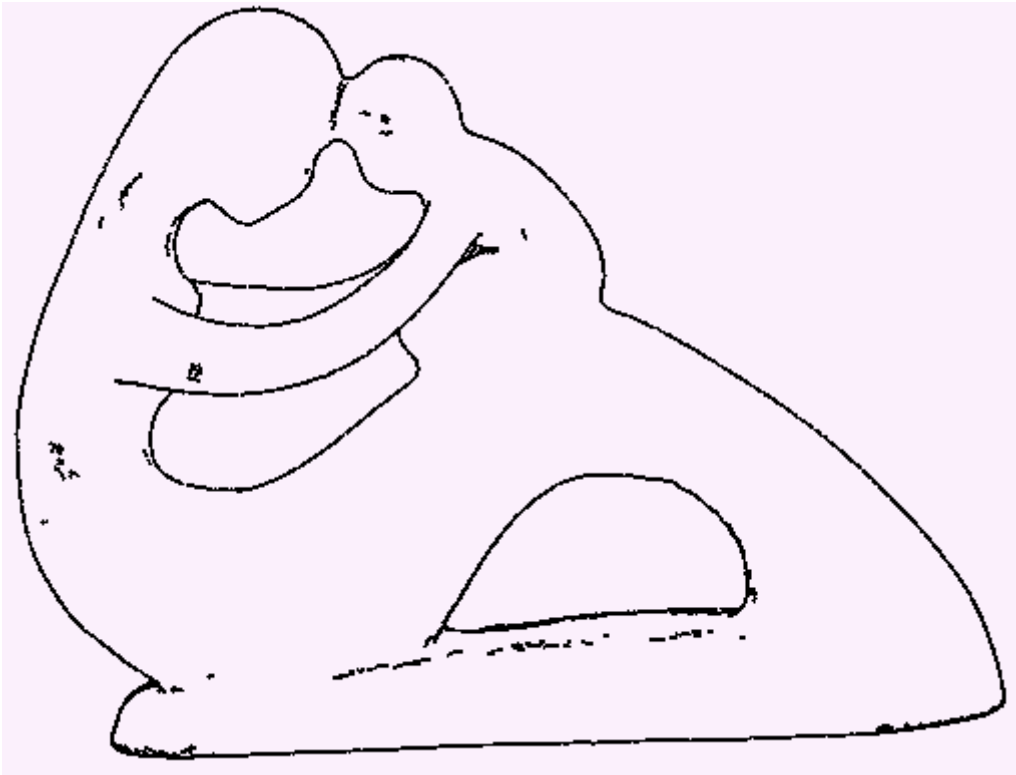


Figura 58. Siluetas y líneas de forma de fertility.ply

Los dos siguientes sombreados están hechos con texturas, pero el patrón usado en ambas es distinto:

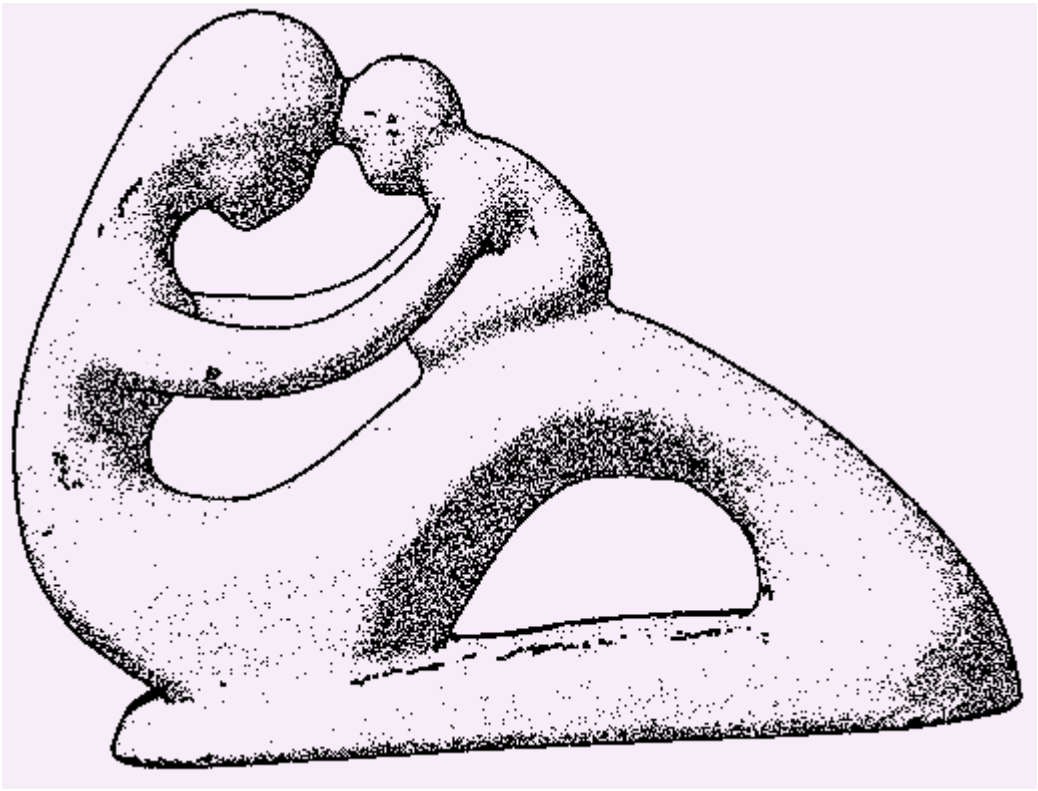


Figura 59. Sombreado con patrón de puntos, siluetas y líneas de forma de fertility.ply

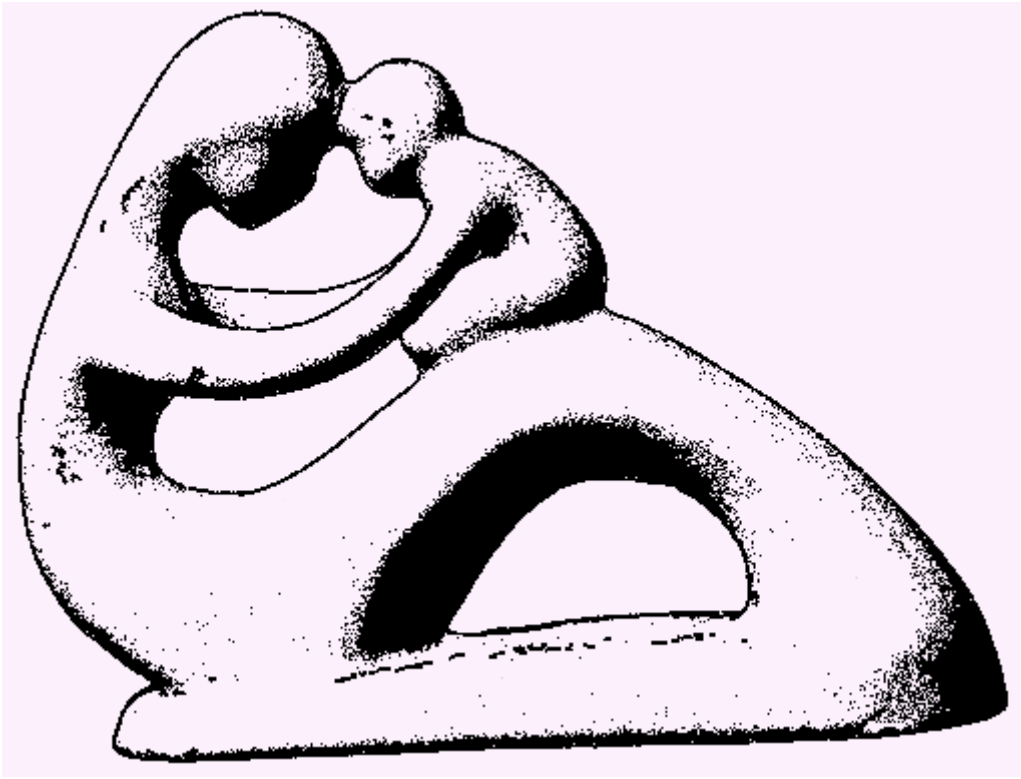


Figura 60. Sombreado con patrón por defecto, siluetas y líneas de forma de fertility.ply

La Figura 61 muestra el sombreado computacional y la Figura 62 una imagen con algunas cotas en el sombreado con texturas:

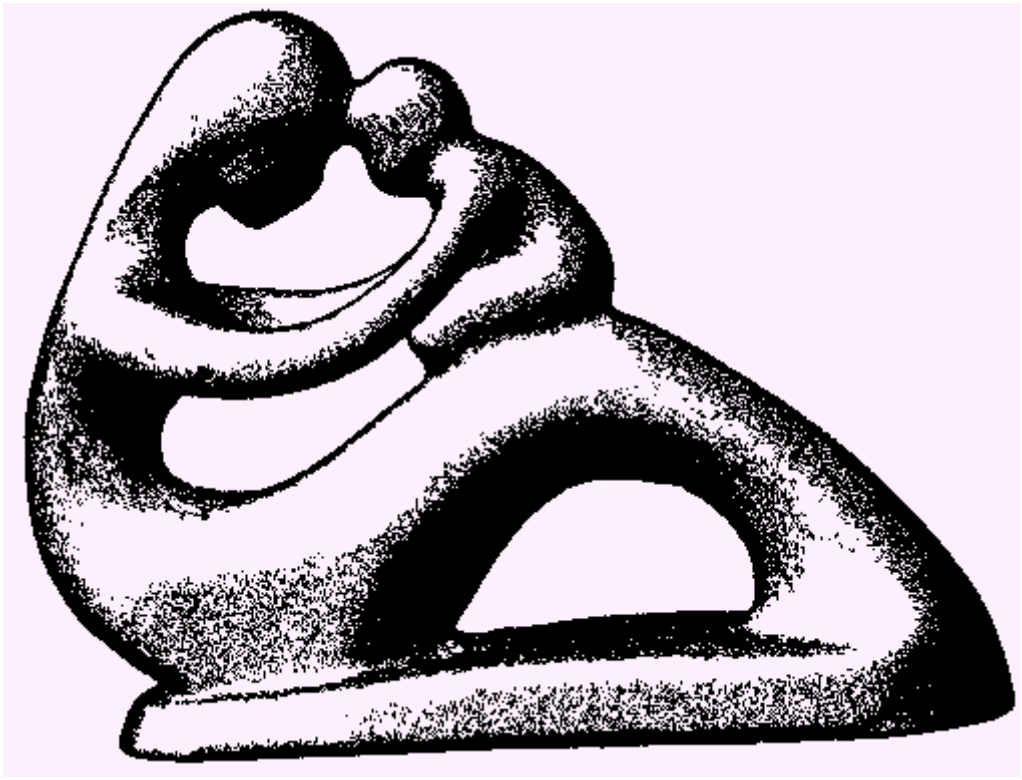


Figura 61. Sombreado computacional con siluetas y líneas de forma de fertility.ply

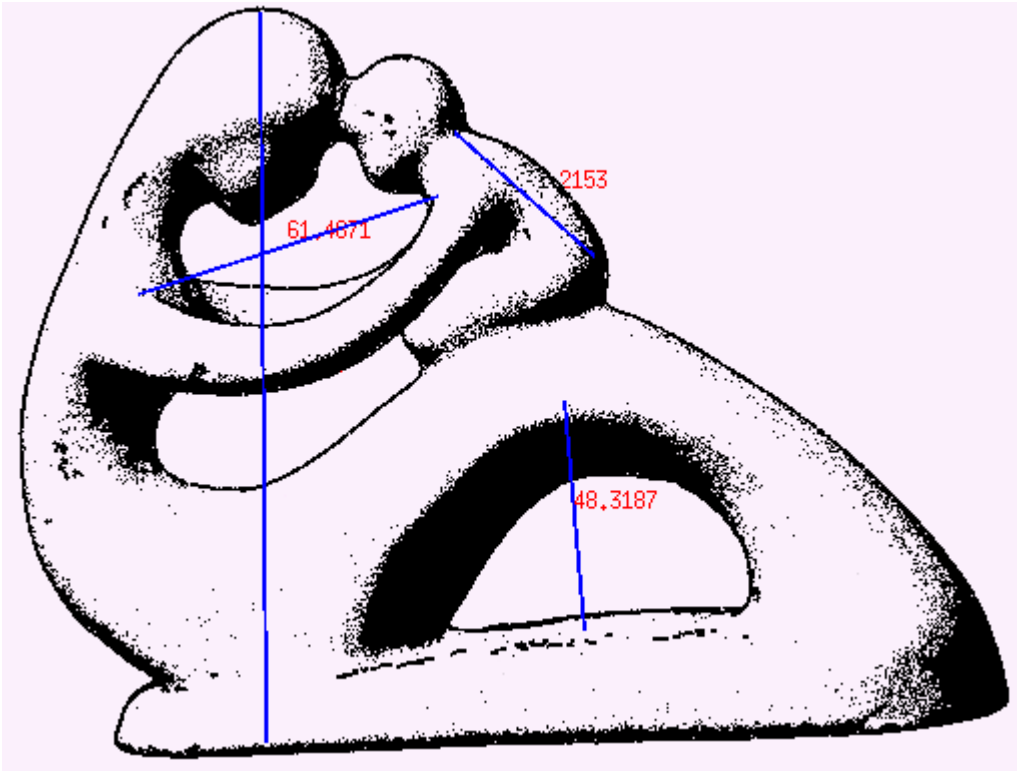


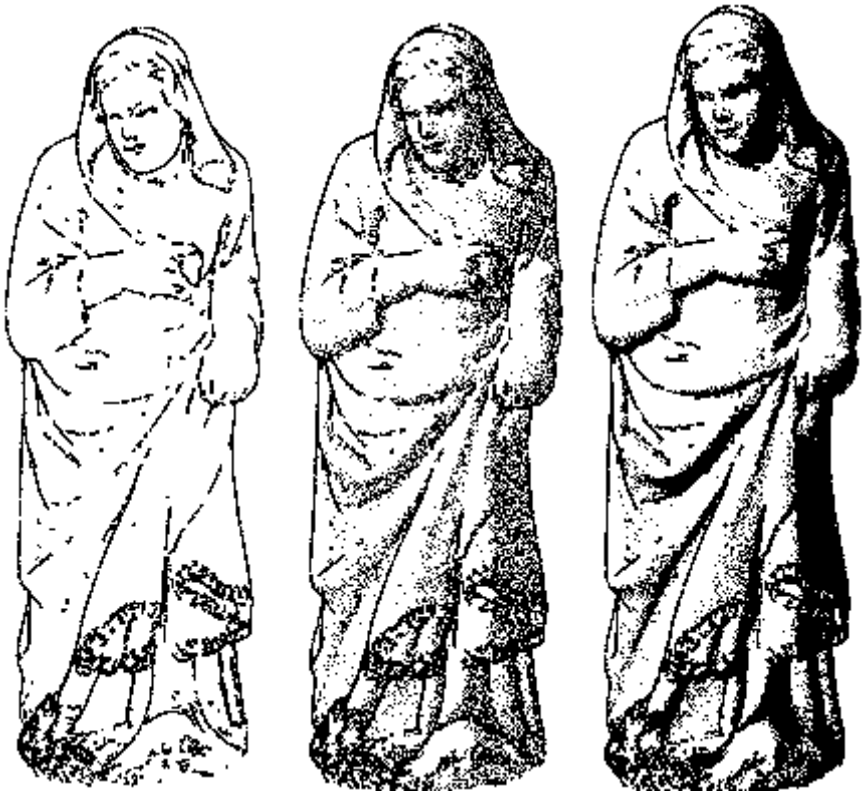
Figura 62. Ejemplo de cotas en sombreado de texturas, con siluetas y líneas de forma de fertility.ply

El último ejemplo que mostraremos se llama virgin_500k.ply. Su formato es binario (binary_little_endian), tiene 252470 vértices y 500000 caras.

En la siguiente imagen podemos observar la figura en 3D:

Figura 63. Modelo de virgin_500k.ply

A continuación tenemos, la silueta con las líneas de forma (Figura 64.a) y el sombreado con texturas (usando un patrón distinto en cada una, Figuras 64.b y 64.c):



(a) **(b)** **(c)**

Figura 64. Silueta con líneas de forma (a), sombreado con texturas con el patrón de puntos, siluetas y líneas de forma (b) y sombreado con texturas con el patrón por defecto, siluetas y líneas de forma de virgin_500k.ply

Podemos observar, a continuación, un detalle en 3D de la cara, y debajo una imagen de las siluetas y las líneas de forma:

Figura 65. Detalle de la cara de virgin_500k.ply

Figura 66. Siluetas y líneas de forma de la cara de virgin_500k.ply

Con la silueta, las líneas de sombra y el sombreado computacional queda la cara del siguiente modo:

Figura 67. Sombreado computacional, líneas de forma y silueta de la cara de virgin_500k.ply

Y, por último, con el mismo sombreado, las líneas de forma y la silueta podemos apreciar en la siguiente imagen un detalle del vestido:

Figura 68. Sombreado computacional, silueta y líneas de forma del vestido de virgin_500k.ply

Proyecto Informático

5º Ingeniería Informática

Mejoras y trabajo futuro

Curso 2006 – 2007

I. Mejoras y trabajo futuro

Hay múltiples mejoras que pueden aplicarse a nuestro programa en un futuro, entre las que comentaremos las siguientes:

- A pesar de que la eficiencia se ha mejorado notablemente desde el comienzo, los modelos de mayor tamaño no se mueven del todo con normalidad, por lo que sería muy recomendable mejorar la eficiencia aún más. Sobre todo, habría que hacer especial hincapié en lo referente al cálculo de la silueta, ya que es con lo que más se ve resentida la eficiencia del programa. Un método útil para el recalculado de la silueta podría ser el siguiente:

  1- Calculamos la silueta inicialmente cuando lo solicita el usuario.
  2- Si se detecta un movimiento en la figura, comprobamos sólo las caras de alrededor de la anterior silueta, ya que normalmente, los movimientos serán pequeños y la silueta no sufrirá grandes variaciones. Vamos, por lo tanto, buscando la nueva silueta sólo en las caras de alrededor de la antigua hasta una cierta profundidad. Si no encontramos por dónde sigue la silueta dentro de dicha profundidad (ha habido cambios bruscos) la recalculamos entera (volvemos al paso 1).

- En el dibujado de las cotas, tal y como se comentó previamente, se muestra, durante los cálculos que se efectúan, el modelo dibujado con los colores únicos. Esto apenas se nota en los modelos pequeños y medianos, pero en los grandes sí puede apreciarse durante un pequeño espacio de tiempo. Aunque no afecta al funcionamiento del programa en sí, visualmente no queda bien, por lo que sería recomendable evitarlo. Para ello, se podría intentar dibujar el modelo con los colores únicos en el búfer trasero (o back buffer) y leer de ahí para hacer los cálculos sin usar en ningún caso swapbuffer para que no se muestre el contenido del búfer trasero.

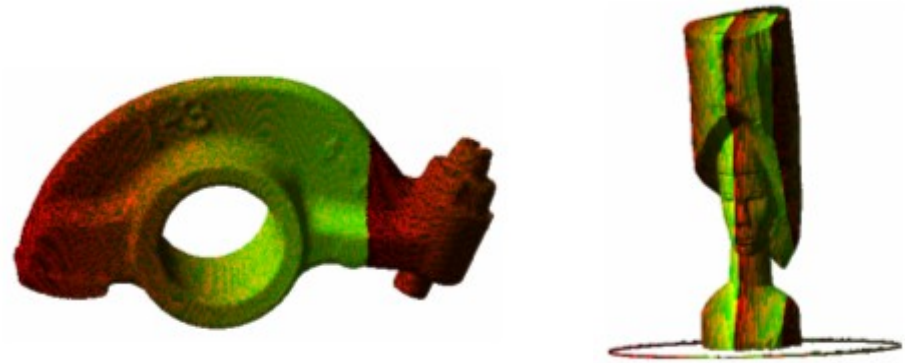


Figura 69. Ejemplos de modelos pintados con sus colores únicos

- En ciertas ocasiones, puede resultar interesante crear cotas fuera del modelo (o bien con los dos puntos fuera del modelo o bien con uno de ellos). Esto no lo permite actualmente la aplicación, pero sería una mejora interesante a añadir (nos puede ayudar, por ejemplo, a ver el ancho y alto de ciertos modelos).

- También sería útil permitir que la aplicación cargue más modelos, a parte de los que podemos encontrar en los ficheros .ply, como por ejemplo, modelos en ficheros .obj. Del mismo modo, podríamos permitir que se lean texturas de imágenes que no tengan formato .tiff o .tif.

- Podría ser útil aumentar el número de estilos ofertados para el dibujado de líneas de forma y siluetas. También podríamos aumentar el número de tipos de sombreado ofertados por el modo de sombreado computacional.

- El sombreado que se muestra tiene la luz fija con respecto al modelo. Podría ser interesante dar al usuario la posibilidad de que la luz permanezca fija en el mundo aunque se gire el modelo.

- Sería útil poder asociar a cada modelo un informe para introducir datos de la pieza arqueológica, datos del yacimiento en el que se encontró, etc.

- En el dibujo arqueológico se divide la imagen muchas veces en dos partes: a la derecha se coloca el desarrollo de la pieza y a la izquierda se pone una imagen de la sección:

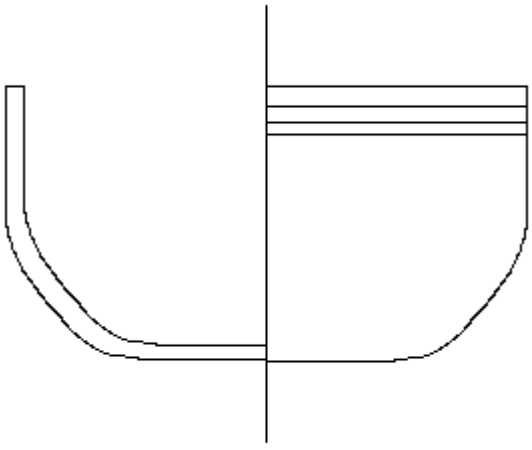


Figura 70. Ejemplo de dibujo arqueológico con sesión de la pieza a la izquierda y el desarrollo de la pieza a la derecha

Esto sería interesante añadirlo a la aplicación, pero no se ha hecho por falta de tiempo, a parte de que quedaba fuera de los límites de la concepción inicial del proyecto.

- También sería interesante permitir al usuario marcar en la silueta determinadas áreas para rellenar con una u otra textura, ya que, por ejemplo, según el material con el que está hecho algo, se dibuja con unas u otras texturas.

- Además, se podría dar la posibilidad de, en el caso de vasijas y piezas simétricas incompletas, crear una reconstrucción aproximada del estado inicial de la pieza (para lo que también sería interesante lo comentado en el punto anterior, ya que, en estos casos, en la parte de pieza que tenemos se usan unas texturas y en el resto otras). Una parte muy importante y necesaria en este proceso, sería el cálculo de la orientación que tenía originalmente la pieza, el cálculo automático del diámetro en el caso de piezas del tipo de vasijas (manualmente, los expertos usan un bordímetro, una serie de círculos concéntricos en papel con los que van comparando la parte de la pieza que tienen), el cálculo automático de la altura de la pieza, la continuación de la decoración que esté fragmentada o se haya perdido, etc.

- En el caso de que nuestro modelo sea una excavación, sería de gran utilidad poder obtener una planimetría con sus medidas, en la que se distinguieran las distintas capas y niveles y con la posibilidad de relacionarlas con un fichero donde informar sobre la composición de la misma (este fichero podría ser el mismo informe que se comentó previamente).

Proyecto Informático

5º Ingeniería Informática

Anexos

Curso 2006 – 2007

I. Material en soporte electrónico

Con esta documentación se adjunta un CD en el que se incluye una versión de este mismo documento en formato *.pdf*.

También se ha incluido el código del programa, debidamente comentado.

II. Bibliografía y referencias

[1] http://astronomy.swin.edu.au/~pbourke/dataformats/ply/

[2] http://vlsicad.ucsd.edu/Resources/SoftwareLinks/documentations/CGAL/ref-manual2/Halfedge_DS/Chapter_hds.html

[3] http://artis.imag.fr/Software/QGLViewer/

[4] "C++ GUI. Programming with Qt 3", Jasmin Blanchette, Mark Summerfield, Prentice Hall.

[5] "Non-Photorealistic Computer Graphics. Modeling, rendering and animation", Thomas Strothotte, Stefan Schlechtwe, Morgan Kaufmann Publishers.

[6] "Non-Photorealistic Rendering", Bruce Gooch, Amy Gooch. University of Utah, School of Computing. A K Peters

[7] http://gpwiki.org/index.php/OpenGL_Selection_Using_Unique_Color_IDs

[8] http://www.cs.ucl.ac.uk/staff/Joao.Oliveira/ply.html

[9] "Silhouette Extraction", Bruce Gooch, Universidad de Utah