DISEÑO Y VISITA VIRTUAL DE INVERNADEROS 3D

Alumno
Antonio Moisés Espínola Pérez

Directores
José Antonio Torres Arriaza
Luis Iribarne Martínez

Ingeniería Informática
- Universidad de Almería -

2 de febrero de 2005

A mi tío Carlos, porque siempre confiaste en mí.
Gracias estés donde estés

Agradecimientos

- Luis Iribarne Martínez y José Antonio Torres Arriaza, mis directores de proyecto.

- Mi amigo Acedo, ingeniero técnico agrícola, por su conocimiento sobre la agricultura.

- Andros y Oliver, guardianes indiscutibles del laboratorio de robótica, por todas las horas que me han dejado trabajar en él.

- Mi compañero y amigo Harlock, por sus sabios consejos.

- Departamento de Lenguajes y Computación de la universidad de Almería.

Prólogo

El mundo de la informática gráfica 3D es apasionante, sólo hay que observar las estadísticas para comprobar que la industria del videojuego está en continua expansión. Gracias a la evolución producida dentro de este mundo se pueden realizar en la actualidad simulaciones de cualquier tipo y visitas virtuales a lugares tridimensionales que existen en el mundo real, o a otros que sólo son fruto de la imaginación de algún diseñador. Esta aplicación del mundo de la informática gráfica se puede enfocar para poder visualizar y recorrer, por ejemplo, construcciones artificiales antes de ser creadas.

Con el desarrollo de este proyecto se pretende cubrir un área casi inexplorada dentro del mundo de la agricultura almeriense: la visita virtual a un invernadero 3D construido por el usuario. El objetivo es poder crear una estructura de invernadero que se adapte a un terreno determinado y que se pueda observar dicha estructura interna y externamente con total libertad de movimiento, para que se permita visualizar dicho invernadero antes de su construcción.

Nadie discute la enorme relevancia que posee el cultivo en invernadero dentro de la provincia de Almería y el crecimiento exponencial acontecido en las últimas décadas. Actualmente existen en la zona bastantes empresas dedicadas a la construcción de estas estructuras, y por lo tanto es predecible un aumento en la expansión de la zona invernada. Debido a la gran competencia en el sector, sería deseable poseer una herramienta capaz de mejorar si cabe la profesionalidad del servicio ofrecido por estas empresas constructoras, de tal modo que el cliente pueda ver su futura inversión antes incluso de comenzar a construirla. Y es aquí donde entra en juego mi proyecto, que como he comentado permite realizar un diseño ajustado a un terreno irregular y una posterior visita virtual al invernadero 3D creado. Gracias a esta herramienta aumentará la confianza del cliente hacia la empresa que lo adquiera, ya que se ofrecerá un trato de mayor calidad y mucho más profesional.

Índice general

Parte I

Memoria

Capítulo 1

Introducción y planteamiento

En este primer capítulo se incluyen los estudios iniciales que realicé antes de desarrollar el proyecto. Está dividido a su vez en las siguientes secciones principales:

1. *Anteproyecto: 'Diseño y visita virtual de invernaderos 3D'.* En primer lugar incluyo el anteproyecto completo, tal y como fue entregado para su aprobación. En él se detallan los requisitos que debe cumplir el proyecto para su correcta finalización.

2. *Estudio de la estructura del invernadero.* A continuación hago una pequeña introducción al mundo del cultivo en invernadero explicando sus ventajas, evolución y algunos tipos principales: invernadero tipo parral y multitúnel.

3. *Estudio de las tecnologías 3D actuales.* En tercer lugar he realizado un estudio completo de las tecnologías que actualmente están en vigor para la creación de visitas virtuales: Blender, OpenGL, Direct3D, engines 3D descargados de Internet, VRML y Blitz3D. He comparado todas estas tecnologías y finalmente he elegido la que más conviene para el desarrollo de este proyecto.

4. *Plan de proyecto.* Por último he llevado a cabo una estimación del coste en tiempo y personas necesarios para completar el trabajo, así como una planificación temporal que me permite establecer fechas límite de entrega. Para ello he realizado un estudio del ámbito del software, una descomposición funcional, una estimación basada en líneas de código y puntos de función, he aplicado los modelos COCOMO básico e intermedio y Putnam, y finalmente un diagrama de Gantt.

1.1. Anteproyecto: 'Diseño y visita virtual de invernaderos 3D'

1.1.1. Objetivos

Con este proyecto se pretenden cubrir dos objetivos bien diferenciados: por un lado elaborar una aplicación que permita diseñar invernaderos 3D y por otro una que permita realizar una visita virtual a la estructura anteriormente creada. Las dos aplicaciones se podrán crear de forma separada (dos programas independientes) o bien crear un solo programa que realice toda la funcionalidad necesaria, como el alumno lo estime oportuno. Los invernaderos creados serán de tipo modular, teniendo la parte superior de los mismos forma de arco.

1.1.2. Metodología y métodos

A) Diseño del invernadero

La primera aplicación consistirá en una interfaz gráfica constituida, entre otros elementos, por una estructura rectangular (a partir de ahora llamada plano) dividida en pequeños cuadrados (a partir de ahora cuadrículas) que se corresponderá con una vista aérea o superior de la estructura del invernadero. El plano estará por tanto formado por una maya de cuadrículas, que compartirán vértices y aristas con sus vecinas adyacentes. El tamaño de la arista de cada cuadrícula se corresponderá con una medida en metros determinada.

El usuario (diseñador de invernaderos) podrá, sobre el plano, delimitar los límites del invernadero, estableciendo de este modo incluso formas irregulares no correspondientes a rectángulos (forma habitual de un invernadero). Para ello podrá marcar los puntos origen y destino de cada una de las paredes externas del invernadero, correspondiéndose cada uno de esos puntos con un vértice de una cuadrícula.

Para construir el invernadero en tres dimensiones, el programa creado por el alumno usará una serie de elementos básicos previamente construidos con alguna herramienta de diseño 3D, como pueden ser Autocad (formato de exportación .dxf), 3D Studio (.3ds) o alguna freeware como puede ser Blender (.blend). Dichas estructuras 3D, independientemente del formato que posean, podrán ser transformadas a cualquier otro formato más sencillo para facilitar la carga de estructuras 3D por parte de la aplicación a construir. Si el alumno así lo desea podrá crear o usar alguna aplicación de exportación de formatos, e incluso crear un programa que modifique la salida de otro en caso de no realizar bien la exportación, para completarla y así adaptarla mejor a sus necesidades.

En dicha interfaz también se podrán modificar una serie de parámetros correspondientes con medidas de las estructuras básicas del invernadero, como por ejemplo la distancia entre los pilares internos, y establecer si la envoltura del invernadero será de plástico o de policarbonato.

A la hora de almacenar la estructura 3D creada (el invernadero completo) existen dos opciones: guardarlas en disco o en memoria. Si el alumno crea dos aplicaciones independientes para elaborar este proyecto, como es evidente deberá guardar la estructura 3D en disco (con el formato que desee) para que la otra aplicación pueda leerla. Si crea una sola aplicación, puede optar por almacenar la estructura del invernadero en memoria si lo desea, y evitar guardarla en disco.

B)Visita virtual

La segunda parte va a consistir en desarrollar un miniengine 3D mediante el cual se podrá realizar una visita virtual del invernadero tridimensional. Para ello el alumno usará alguna de las librerías o herramientas actualmente en vigor, como OpenGL, Direct3D, Blender, etc... ayudado de todas las APIs que necesite. En dicha visita el usuario podrá desplazarse horizontalmente por el interior del invernadero mediante el teclado, usando el ratón para poder dirigir el objetivo de la cámara al punto deseado (al purísimo estilo del videojuego Quake).

Además se deberá implementar algún algoritmo para detectar colisiones, en principio, con las paredes que delimitan los límites del invernadero, para que el visitante quede encerrado en su interior. Aparte de las paredes, el alumno tendrá libertad de implementar la detección de colisiones con los elementos que desee: todo irá en función de la influencia que tengan las colisiones sobre la fluidez de la visita virtual (puede que, si añade elementos como tubos u objetos situados a baja altura se entorpezca la visita virtual). Por lo tanto, la cámara nunca podrá salir de la estructura que delimita el invernadero, pero si el alumno lo desea podrá atravesar objetos en su interior. Para implementar la detección de colisiones se usará alguna técnica que contenga una base matemática justificada.

La iluminación deberá ser la correcta para poder apreciar los objetos a una cierta distancia y las texturas o materiales usados también deberán apreciarse. También pueden incluirse otros efectos si el alumnos así lo desea, como por ejemplo niebla.

1.1.3. Planificación a seguir

1. En primer lugar, el alumno deberá documentarse debidamente y elegir las tecnologías 3D apropiadas, comparando algunas de las existentes.

2. Seguidamente deberá realizar una descomposición funcional de cada una de las partes en las que se divide en proyecto.

3. A continuación desarrollará dichas funciones usando las tecnologías elegidas, y las integrará sucesivamente al proyecto.

4. Finalmente plasmará en la documentación los resultados obtenidos.

1.1.4. Bibliografía básica

3D Game Engine Design: Approach to real-time Computer Graphics, David H. Eberly
OpenGL programming Guide, Jackie Neider (2000)
OpenGL Programming for Windows, Ron Fosner (1997)
OpenGL Programming Guide, Neider & Davis & Woo (1996)
OpenGL SuperBible 2nd Edition, Richard S. Wright Jr & Michael R. Sweet
Interactive Computer Graphics - A top-down approach with OpenGL, Edward Angel
OpenGL Game Programming, Kevin Hawkins & Dave Astle
OpenGL 2.0 Specification (.pdf desde Internet)

1.2. Estudio de la estructura del invernadero

Un invernadero se considera una estructura con las medidas requeridas y cubiertas con un determinado material translúcido o trasparente, que permite tanto el crecimiento óptimo de las plantas como el acceso a las personas para laborar en el cultivo.

Es muy importante hacer una buena selección del plástico para reducir los riesgos de la inversión, no solamente en el material, sino también en toda la plantación. Para escoger la cubierta adecuada es necesario tener en cuenta la situación geográfica, las temperaturas máxima, mínima y media, las posibilidades de heladas, el régimen de vientos, la humedad relativa, el régimen de lluvias, la radiación solar, la especie que se va a sembrar.

La cubierta requiere de bloqueador de la radiación ultravioleta por lo menos hasta los 315 nanómetros. En función de los requerimientos puede incrementarse el bloqueo a costes gradualmente más elevados, que no siempre alcanzan a justificarse. La cubierta ideal debe entonces bloquear la radiación ultravioleta propuesta, pero ser permeable a la radiación solar del resto de la banda hasta 3000 nanómetros; retener la energía calorífica generada por las radiaciones IR que emanan del suelo y de las plantas; minimizar los problemas que se derivan de la condensación de agua; tener larga duración y coste balanceado con los beneficios.

1.2.1. Ventajas del uso del invernadero

Son muchos los factores que contribuyen a beneficiar una plantación protegida bajo invernadero. Entre ellos se destacan los siguientes:

1. Difusión de luz: propiedad que tienen las cubiertas de cambiar la dirección de los rayos solares distribuyéndola equitativamente por toda el área para beneficiar a todo el invernadero en su conjunto y a la vez impedir que lleguen directamente a la planta. Este factor permite el desarrollo armónico del cultivo y ayuda a obtener frutos más homogéneos y sanos.

2. Fotosíntesis: el proceso fotosintético se ve favorecido dentro del invernadero, debido en gran medida a la forma en que es difundida la luz y a la conservación de temperaturas homogéneas, que deben ser en términos generales, las óptimas.

3. Microclima: manejar un microclima que permite controlar y mantener las temperaturas óptimas aporta cosechas más abundantes y de mejor calidad, reconocidas en el mercado por mejores precios. Adicionalmente permite programar las cosechas para épocas de escasez.

4. Luminosidad: dentro de un invernadero se puede obtener mayor o menor luminosidad, dependiendo de su diseño y de su cubierta.

1.2.2. Evolución del invernadero

Actualmente es evidente relacionar la provincia de Almería con una producción agrícola muy intensa, lo que ha impulsado enormemente la economía de esta provincia en las cuatro últimas décadas. Todos estos logros está íntimamente ligados al desarrollo de las técnicas de producción en invernadero, capaces de convertir unas condiciones edafológicas y climáticas adversas para la práctica de la agricultura, en grandes recursos para diferentes tipos de cultivos promover cultivos. Almería es una de las zonas con mayor concentración del mundo destinada al cultivo intensivo bajo plástico.

Han pasado más de 40 años desde la construcción del primer invernadero con cubierta de plástico en la provincia de Almería. La expansión de este sistema de cultivo, muy lenta en su comienzo, ha adquirido en las dos últimas décadas un desarrollo muy importante. Existen varias miles de decenas de hectáreas de superficie invernada en la provincia de Almería.

El primer gran impulso de la horticultura almeriense comenzó con el sistema de cultivo enarenado, que tuvo su origen en 1880 cuando un agricultor observó un notable incremento de la producción de tomate en aquellas zonas que habían sido cubiertas con arena extraída por las hormigas de un hormiguero contiguo. Seguidamente aplicó arena a otras plantas, y también se observó la mejoría. A partir del año 1941 tuvo lugar la expansión del cultivo enarenado en la zona costera.

El motivo definitivo de la hegemonía almeriense en el mundo de la agricultura fue consecuencia del empleo de invernaderos. El primer invernadero se construyó en 1963 con cañas y madera de eucalipto. Para sostener el filme de plástico se empleó alambre galvanizado. La cubierta era de polietileno.

Los datos de producción en kg/ha hablan por sí solos:

Cultivo	Aire libre	Plástico doble	Plástico sencillo
Judías	476	9.520	7.020
Pimiento asociado	1.660	7.770	2.870
Pepino asociado	9.440	16.800	12.770
Tomate	16.780	36.250	33.390

Estos resultados tan positivos animaron a muchos agricultores, y la perspectiva de la agricultura cambió en Almería, en la que el enarenado seguía teniendo bastante protagonismo para potenciar el suelo y el plástico para optimizar el clima y protector de inclemencias. Durante toda la década de los 70 se extendió la superficie invernada.

En 1980 se introducen las semillas híbridas, y en 1981 el riego por goteo localizado. Diez años después comenzó a haber una serie de adelantos sin precedentes: control informático de riego, nutrición con máquinas de fertirrigación, polinización natural del tomate mediante abejorros, control integrado de plagas, la hidroponía (técnica de cultivar sin tierra), etc...

La construcción de invernaderos, que comenzó siendo un ensayo, terminó convirtiéndose en una poderosa técnica que transformó en poco tiempo muchos aspectos de la agricultura almeriense: paisaje, calidad, productividad, etc...

Por lo tanto debido al gran auge y al futuro prometedor que se espera del uso del invernadero, el desarrollo de este proyecto puede beneficiar enormemente a la empresa constructora de inver-

naderos que posea la herramienta resultante, ya que se ofrecerá al cliente una profesionalidad que será recompensada con su fidelidad. No es lo mismo mostrar a un cliente en una hoja de papel un plano donde se describa la forma del invernadero, que diseñárselo directamente sobre el programa con una forma que se pueda adaptar lo mejor posible a su parcela de terreno, y posteriormente se pueda realizar una visita virtual sobre dicho invernadero 3D, es decir, poder ver el invernadero antes de ser construido.

1.2.3. Invernadero tipo parral

Otro aspecto importante ha sido el modo en el que han evolucionado los invernaderos. En general se desarrollan en estructuras sencillas y carentes de sistemas para el control climático, por lo que el crecimiento y desarrollo de los cultivos están sujetos a merced de la evolución del clima local, lo que induce acumulaciones productivas y amplias variaciones en la calidad y cantidad de las cosechas.

En la zona del sudeste de Almería predominan las estructuras de 'tipo parral', llamadas así en referencia a las estructuras que servían de soporte al cultivo de uva de mesa. Estos invernaderos sencillos de estructura de madera o metálica están anclados al terreno mediante alambres tensados. Las principales ventajas de estos invernaderos son:

1. Su economía de construcción.
2. Su gran adaptabilidad a la geometría del terreno.
3. Mayor resistencia al viento.
4. Aprovecha el agua de lluvia en períodos secos.
5. Presenta una gran uniformidad luminosa.

Sin embargo, son muchos los inconvenientes de este tipo de invernadero:

1. Poco volumen de aire.
2. Mala ventilación.
3. La instalación de ventanas cenitales es bastante difícil.
4. Demasiada especialización en su construcción y conservación.
5. Rápido envejecimiento de la instalación.

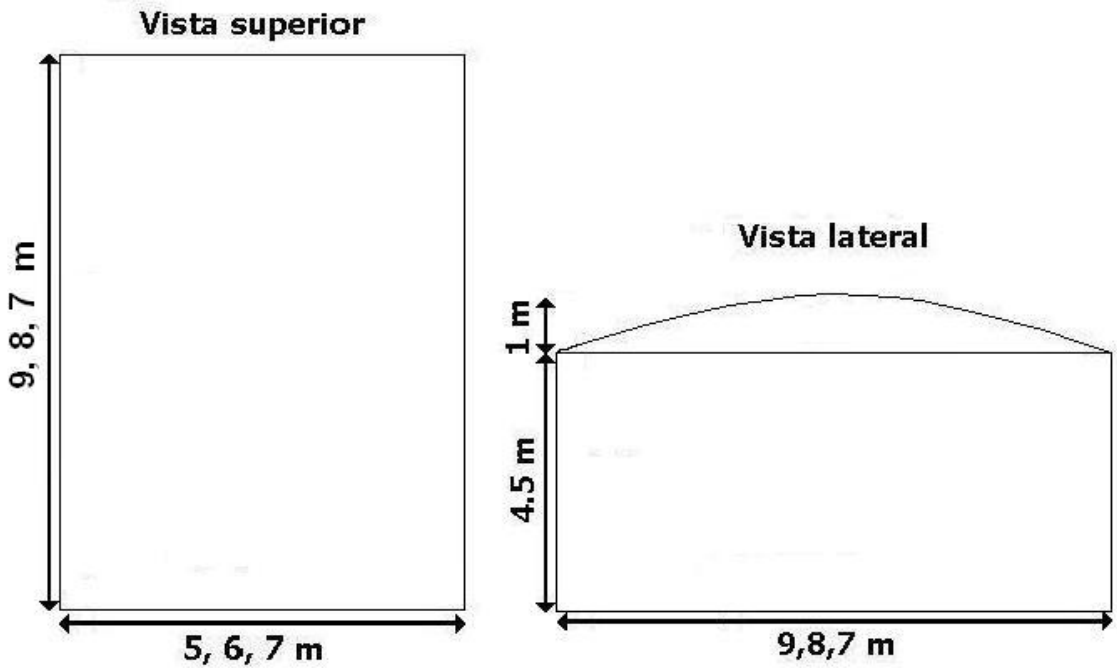
6. Peligro de hundimiento por las bolsas de agua de lluvia que se forman en la lámina de plástico.

7. Peligro de destrucción del plástico y de la instalación por su vulnerabilidad al viento.

1.2.4. Invernadero multitúnel

A finales de la década de los 90 se produjeron avances en nuevas estructuras de invernaderos: parral mejorado y multitúnel. Es en este último tipo de estructura en la que me voy a centrar en mi proyecto, en el invernadero multitúnel, y sus características principales son las siguientes:

1. Las medidas del invernadero que yo voy a construir son las siguientes:



2. El techo es de tipo arco, no ojival.

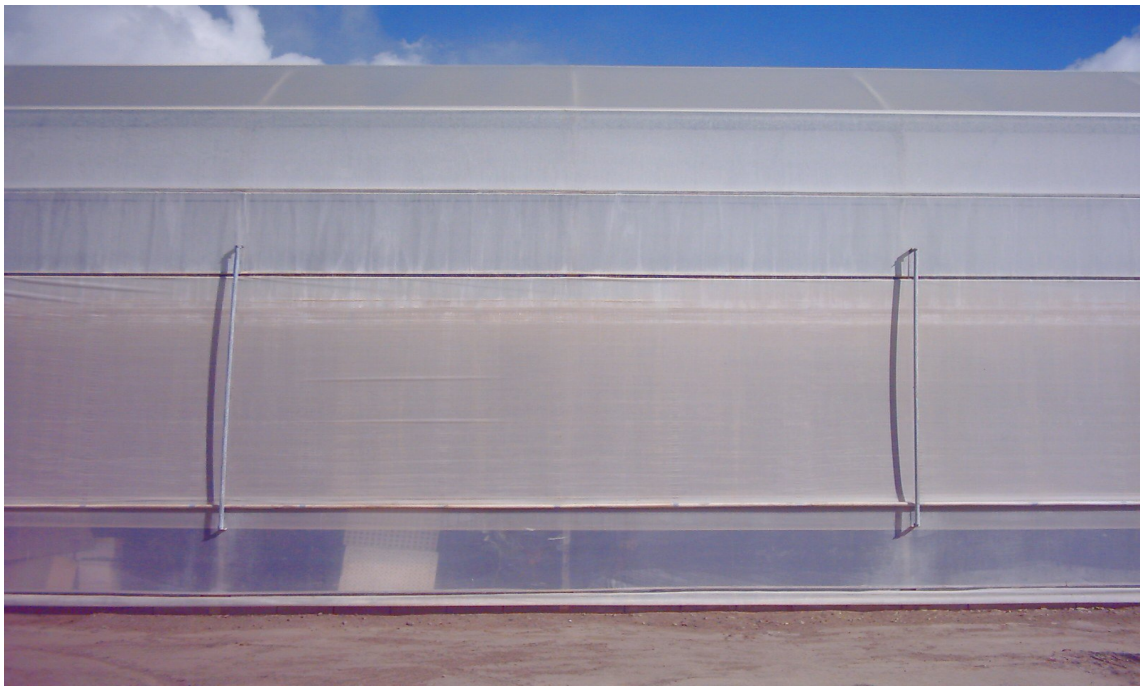
3. La cubierta exterior puede ser de plástico o policarbonato.

4. Posee ventilación cenital.

5. En lo que respecta a la calidad ofrecida, este tipo de invernadero proporciona mucho más rendimiento que el de tipo parral. Pero su coste de construcción es también mucho mayor.

A continuación muestro fotografías del invernadero que voy a simular con sus tipos:

Frontal del multitúnel tipo plástico



Lateral del multitúnel tipo plástico

Frontal del multitúnel tipo policarbonato (éste es ojival, yo simularé el tipo arco)

Lateral del multitúnel tipo policarbonato

1.3. Estudio de las tecnologías 3D actuales

Cuando comencé el proyecto, todo era un inmenso mar de dudas. Conocía algunas de las tecnologías 3D vigentes que podrían serme útil para desarrollar el proyecto, pero a priori no me decantaba por ninguna. Todas tenían muchas ventajas, así como muchos inconvenientes. Además, aunque algunas a primera vista parecían ser las adecuadas, no hacían nada más que llevarme a callejones sin salida porque llegaba a un punto en donde no podía continuar con el proyecto, ya que no ofrecían la suficiente funcionalidad.

Las principales opciones que barajé fueron las siguientes:

1. Blender: herramienta muy completa que incluye, aparte de la funcionalidad necesaria para poder construir modelos tridimensionales, un game-engine incorporado muy fácil de utilizar. En principio parecía la mejor opción, ya que me resolvía muchos de mis grandes problemas (detección de colisiones, engine 3D, etc).

2. Engines 3D: existe en Internet una gran cantidad de engines 3D con su código fuente completamente gratuito para poder descargarlos. También parecía una opción muy interesante porque me ahorraría mucho trabajo.

3. OpenGL: era una opción también muy buena, ya que se ha convertido casi en un estándar en la creación de videojuegos 3D. Además es multiplataforma y se puede programar con C++, lenguaje que creo dominar.

4. Direct3D: no parecía una mala opción, aunque siempre me he decantado más por OpenGL.

5. Blitz3D: lenguaje de programación que también se ha puesto de moda en la creación de videojuegos 3D.

6. VRML: la tecnología que tuve menos en cuenta, ya que es perfecta para hacer visitas virtuales estáticas en Web, pero no para poder modificar dinámicamente propiedades de los objetos. Quedaba muy limitada para mi proyecto.

Además tuve en cuenta otras posibilidades, aunque pronto quedaron descartadas por no resultar adecuadas. A este análisis inicial le dediqué casi una quinta parte del tiempo total dedicado al proyecto, porque fue quizá la decisión más difícil que tuve que tomar, ya que muchas herramientas ofrecían diversas ventajas y no sabía por cuál decantarme.

Con algunas herramientas malgasté semanas intentando realizar el proyecto con la tecnología inadecuada. Pero poco a poco fui descartando la mayoría, como explico en los siguientes apartados.

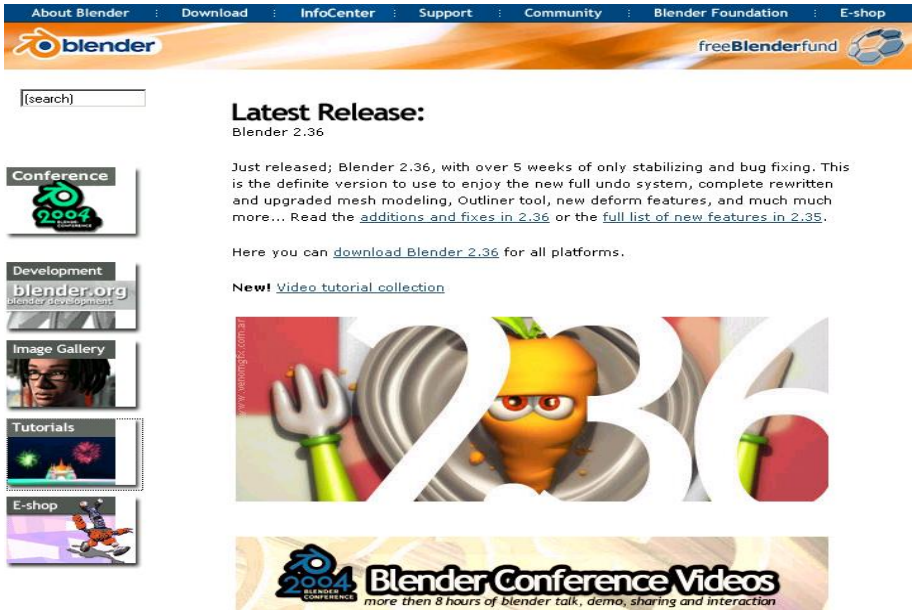
1.3.1. Blender

Blender es una herramienta de creación de gráficos 3D que integra opciones de modelado, animación, renderizado, post-producción, interactividad 3D en tiempo real y creación de videojuegos.

Ventajas

La característica que realmente hace interesante a esta herramienta consiste en que es de código abierto, totalmente gratuita, y además ofrece tanta o más funcionalidad que otras herramientas de pago como 3D Studio o Autocad. Se puede adquirir accediendo a su página principal:

http://www.blender3d.org/
http://www.blender.org/



Además soporta una gran cantidad de plataformas, por lo que si desarrollo el proyecto con Blender se ampliaría enormemente el número de usuarios o futuros clientes:

1. Windows 95, 98, 2000, XP, ME, NT (i386)
2. Mac OS X
3. Linux i386
4. Linux PPC
5. FreeBSD 4.2 (i386)

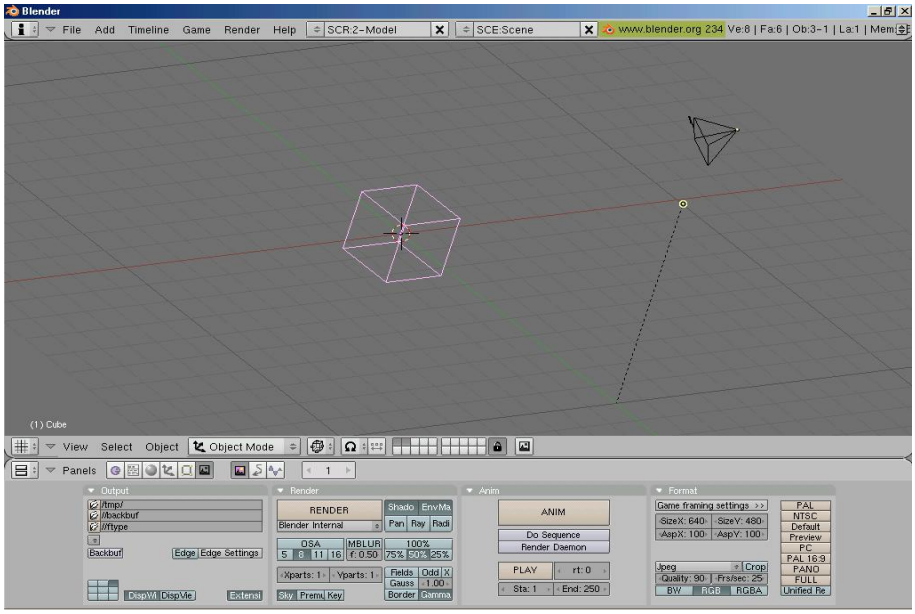
6. SGI Irix 6.5
7. Sun Solaris 2.8 (sparc)

Pero sin duda la funcionalidad que más me interesa de Blender para el desarrollo de mi proyecto es la que ofrece su game-engine, incorporado en el propio Blender, del que destacan las siguientes características:

1. Editor gráfico para definir el comportamiento interactivo sin tener que programarlo.
2. Detección de colisiones y simulaciones.
3. API para escribir scripts en Phyton para un control más avanzado.
4. Soporta todos los modos de iluminación de OpenGL, incluyendo transparencias, texturas animadas, etc..
5. Ejecución del videojuego o contenido 3D interactivo sin compilar o preprocesar.

Gracias a su game-engine, Blender ha sido usado en multitud de aplicaciones en algunos aspectos parecidas a mi proyecto. De entre todas ellas pero caben destacar las dos más importantes:

1. Aplicaciones industriales y arquitectónicas: es ideal para realizar visitas virtuales a mundos tridimensionales y además es muy fácil de utilizar. Se puede además interaccionar con los objetos del mundo, realizar transformaciones en él, etc...
2. Diseño web: gracias a la tecnología de Blender se puede publicar en una web el contenido de una visita virtual, y sólo es necesario descargar de su web el '3D web pluggin' para poder visualizarlo, disponible actualmente para los navegadores Internet Explorer y Netscape (ambos bajo una plataforma Windows).



Además de la gran calidad que ofrece Blender, existen muchos tutoriales en la página oficial donde se explican algunas de las posibilidades de esta potente herramienta. En ellos me basé para aprender a manejar Blender, y además había manuales para muchas categorías:

1. Comienzo: desde cómo utilizar la interfaz de Blender hasta cómo construir y texturizar un castillo y un delfín.

2. Modelado: hay muchos objetos 3D cuya construcción se explica paso a paso, como animales, caras, etc...

3. Animación: también explica cómo se pueden realizar animaciones con los objetos creados, añadirles sonidos y música.

4. Game-engine: la categoría más importante, donde se explica cómo realizar una visita virtual, interaccionar con los objetos de la escena, asignarles propiedades físicas, etc...

Realmente he dedicado bastante tiempo intentando desarrollar el proyecto con Blender, sobre todo porque así hubiera evitado todo el trabajo de construir un engine 3D, ya que el que incorpora esta herramienta está muy optimizado y ofrece muchísimas prestaciones.

De hecho al principio estaba convencido de que iba a ser la herramienta que iba a utilizar. En primer lugar descargué la guía oficial de Blender, distribuida en dos volúmenes, que se puede descargar en pdf de la web oficial. También descargué algunos tutoriales para aprender a usar el lenguaje de script Phyton, usado para el game-engine. Además estuve estudiando la mayoría de los ejemplos que encontré por Internet que tocaban una gran cantidad de materias: creación de objetos 3D, aplicación de texturas, y sobre todo los relacionados con el motor de videojuego que incorpora Blender.

En seguida descubrí que realizar una visita virtual a un modelo estático era bastante sencillo, pues solamente se tenía que importar dicho modelo del archivo correspondiente y seguir paso a paso uno de los tutoriales que posee la web oficial de Blender, en el que se explica detalladamente cómo crear y configurar una cámara para que pueda moverse y rotar por un mundo tridimensional, controlando detalles físicos como la detección de colisiones. Descargué de Internet un modelo tridimensional del famoso TITANIC y conseguí hacer una visita virtual. Pude comprobar que el game-engine que incorpora Blender es fantástico, ya que la detección de colisiones es muy buena así como las propiedades físicas que se les pueden asignar a los objetos (como por ejemplo conseguir que una esfera con una textura determinada rodara por el suelo al empujarla con la cámara, fue un detalle que me llamó mucho la atención). Para ello había que asignar a cada objeto una serie de propiedades físicas, como por ejemplo el peso.

Otro aspecto importante era que Blender permite trabajar con una gran cantidad de formatos diferentes, luego no tendría que realizar conversiones entre formatos ni nada por el estilo. Además como los modelos los hubiera creado con el propio Blender, no habría tenido ningún tipo de problema.

Inconvenientes

Sin embargo uno de los problemas con los que primero tropecé fue con la generación de un archivo ejecutable de la visita virtual ya que, aunque realizar un recorrido a través de un mundo virtual cargado previamente era muy sencillo, había que abrir el programa Blender y ejecutar el game-engine cada vez que se quería realizar dicha visita. Finalmente conseguí solucionar este primer inconveniente.

Una vez controlada la visita virtual a un modelo estático, comencé a reflexionar sobre otras cuestiones. Por ejemplo, antes de dicha visita virtual se debería mostrar una interfaz de usuario donde éste pudiera configurar algunas de las características del invernadero, así como su forma vista desde arriba. Y fue entonces cuando, tras un largo recorrido por foros y chats, descubrí que en Blender existen dos tipos de scripts en Phyton, unos que se pueden ejecutar mientras se utiliza el game-engine, y otros como por ejemplo las interfaces que tienen que ejecutarse independientemente.

Es decir, tenía que separar las dos partes principales del proyecto: por un lado construir una interfaz mediante un script Phyton con la que el usuario definiera las características del invernadero así como su forma, y seguidamente ejecutar el game-engine sobre el invernadero creado.

A partir de aquí, por cada una de las dudas que conseguía resolver, aparecían otras tantas más complejas e inquietantes. Algunas de ellas fueron:

1. Si el usuario crea el invernadero usando la interfaz escrita en Phyton (ejecutada a través de Blender) y guarda dicha estructura 3D en disco, ¿se podría programar otro script que tomara dicho modelo de disco, le asignara automáticamente las propiedades necesarias para realizar correctamente la visita virtual (necesarias para la detección de colisiones entre otras cosas) y seguidamente comenzara dicha visita? ¿O tendría el usuario que configurar la visita virtual cada vez que creara un modelo de invernadero, realizando siempre un conjunto de pasos en Blender que le ocuparían un tiempo de varios minutos? Para que se realice correctamente la visita virtual, como he comentado, hay que asignarle al objeto varias características, y esto se tiene que hacer a través de Blender, editando las propiedades del objeto. Y la interfaz de Blender no es precisamente intuitiva ni inmediata, sino re-

torcidamente compleja, mostrando todos mis respetos hacia sus creadores... ¿Realmente era necesario que el usuario, eterno inculto en el campo de la informática, aprendiera a configurar la visita virtual para cada invernadero que creara? ¿Dónde estaba entonces el automatismo?

2. Suponiendo que el problema anterior tuviera solución, ¿sería realmente capaz de crear a través de la interfaz en Phyton una estructura de invernadero 3D? No es una tarea sencilla ni muchísimo menos, ya que puede tener formas irregulares, y necesito tener un control total sobre la programación, estar al nivel más bajo posible, ya que tenía que trabajar no sólo con objetos, sino a nivel de polígono, incluso de vértices. Por ejemplo, las partes más conflictivas del invernadero son las intersecciones de las paredes externas que no poseen las esquinas con ángulo recto, ya que esas zonas no corresponden con una cuadrícula estándar, sino que se debería deformar de alguna manera posible (también hay que deformar los objetos cuando se varía la distancia entre los pilares o alguna otra medida). Una posible solución la encontré en las llamadas 'curvas IPO' que permitían deformar un objeto determinado en cualquiera de los ejes de coordenadas, usando también scripts Phyton, pero tras realizar varias pruebas los resultados obtenidos eran lamentables, y terminé por abandonar cuando me di cuenta de que en ningún ejemplo que tenía se deformaba una malla en tiempo de ejecución del game-engine.

3. Suponiendo que el problema anterior también tuviera solución, los ejemplos que circulan por la web de visitas virtuales no son realmente demasiado sorprendentes, y lo que sí sorprende es la experiencia que poseen los creadores, verdaderos gurús en Blender. Simplemente permiten alguna interacción con algún que otro objeto, pero no ofrece información sobre el estado del mundo, ni tiene otras ventanas ni opciones. Luego si verdaderos profesionales en el campo de la informática gráfica, usando esta herramienta, no han terminado de deslumbrar con sus ejemplos, no esperaba hacerlo yo.

4. Además, aunque realizar una visita virtual era relativamente sencillo, existen en Blender varios métodos para aplicar texturas a los objetos, y para la visita virtual hay que elegir unos concretos que no terminaban de convencerme, porque se empleaba demasiado tiempo en el proceso.

5. Y ya ni hablar del cambio dinámico de características del mundo virtual, como por ejemplo poder añadir niebla durante la visita, cambiar a modo alambre, cambiar texturas dinámicamente, etc...

Conclusión

Al principio puse muchísima esperanza e ilusión en Blender, pero terminó por desanimarme. Blender es excelente para realizar visitas virtuales a modelos estáticos ya creados, e interaccionar

con ellos, pero no me parece tan bueno para crear modelos 3D y hacer cambios dinámicamente. Necesitaba alguna tecnología con la que trabajar a más bajo nivel.

1.3.2. Engines 3D

Existen en Internet muchos engines 3D totalmente gratuitos, de los que se puede descargar incluso el código fuente completo. Esta fue una de las opciones que más me llamaba la atención, ya que con un engine ya programado, evitaría muchas complicaciones y ahorraría gran parte del trabajo.

Y la verdad es que lo más complejo de todo el proyecto ha sido precisamente la creación desde cero del engine 3D, y es algo que yo sabía antes de programarlo desde cero, por eso busqué la alternativa de usar un engine ya desarrollado para ahorrar tiempo, cuando comprobé que con Blender no tenía nada que hacer.

Ventajas

Basta con ir al motor de búsqueda *Google* y escribir 'engines 3d', y aparecen una serie de páginas que contienen listas completas de motores gráficos 3D. Sólo en la primera página que aparece tras la búsqueda hay en el momento que escribo esta documentación 643 engines 3D disponibles, de los que 266 ofrecen su código fuente para poder descargar de manera totalmente gratuita.

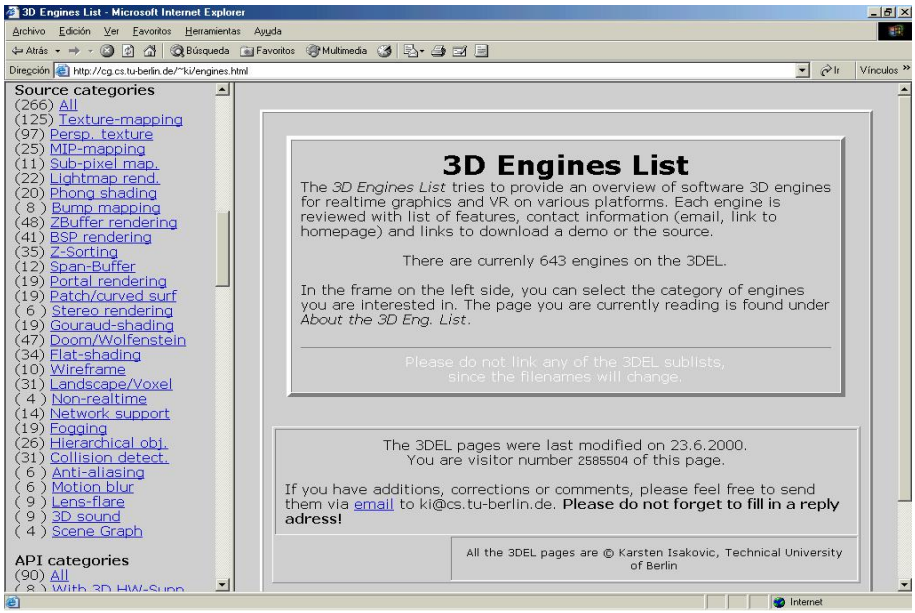
Además, como se puede observar en la imagen, dichos engines aparecen clasificados según el número de características que incorporan: detección de colisiones, árboles BSP, fogging, anti-aliasing, motion blur, cartoon, texture-mappint, etc...

En esta misma web se pueden encontrar también los engines clasificados por APIs, es decir, distinguen los engines en los que se usa OpenGL de los que usan Direct3D o QuickDraw3D. También hace distinción entre los engines que incorporan soporte para aceleración 3D hardware, como por ejemplo se puede hacer gracias a las extensiones de OpenGL.

No sólo existen engines para el sistema operativo Microsoft Windows, sino para muchas otras plataformas: Linux, Mac, Solaris, etc... Y además en multitud de lenguajes de programación diferentes: C, C++, Java, Pascal, etc..

Para cada uno de los engines registrados, se ofrecen todas las características que ofrece una a una, con lo que el usuario puede elegir aquél que más le convenga para descargar. No obstante

existen otras muchas webs desde las que se pueden descargar otros engines 3D diferentes.



También encontré los engines de algunos videojuegos clásicos como el Wolfenstein, el Doom o más recientemente, el Quake. Los primeros eran tan antiguos que ni los tuve en cuenta, pero el del Quake sí. Pero era demasiado complejo, al fin y al cabo era el engine de un videojuego profesional que estuvo líder en el mercado durante mucho tiempo. Para la visita virtual al invernadero no hacía falta algo tan complicado.

Inconvenientes

Por lo tanto aunque a priori esta opción también parecía muy buena, pero terminó por desilusionarme ya que tras descargar y examinar diversos códigos comprobé que no sería tan fácil acoplar dicho engine a mi proyecto, debido a que los códigos eran extensos y en muchas ocasiones sin comentar, por lo que muchas de las funciones (C) o métodos (C++) no sabía exactamente lo que realizaban.

Cada vez me agradaba menos la idea de comenzar a utilizar un engine 3D y que luego me diera problemas, o la idea de dedicar más tiempo en entender lo que otra persona ha hecho a crear mi propio engine 3D. Además, la mayoría de los motores gráficos ofrecían demasiada funcionalidad, mucha más de la que yo necesitaba.

Yo necesitaba sólo una pequeña parte de la funcionalidad como puede ser la detección de colisiones, el mapeado de texturas, la carga de objetos 3D desde archivo, la rotación de la cámara y movimiento a través del mundo virtual, etc.. por lo que me sobraban muchísimas funciones. Y aunque había engines más sencillos que otros, los sencillos precisamente eran los que peor

funcionaban.

Pero también necesitaba tener un control total y absoluto sobre el engine, ya que no quería quedarme atrancado en algún punto concreto del desarrollo del proyecto. Los cambios dinámicos del tamaño de los objetos (distancia entre pilares) no eran tan sencillos con algunos de los engines que encontré. Y cada vez estaba más convencido de que el engine lo tendría que programar yo mismo.

Conclusión

Finalmente terminé por abandonar esta idea, sobre todo después del tiempo perdido con Blender, que al principio también me pareció la opción adecuada.

1.3.3. OpenGL

OpenGL es sin lugar a dudas la API que prevalece en la industria para desarrollar aplicaciones gráficas 2D y 3D. Se le puede considerar el sucesor a la formidable IRIS GL-library de Silicon Graphics que hizo tan popular las estaciones de trabajo SGI como plataforma predilecta para desarrollo científico, de ingeniería y de efectos especiales. SGI puso en OpenGL una buena parte de su pericia para hacer una API para el futuro fácil de usar, intuitiva, portable y perceptiva a las redes.

Al mismo tiempo podemos acreditar a SGI por darse cuenta de la importancia de los estándares abiertos. Varios fabricantes de software y hardware tomaron parte en la especificación de OpenGL y permanecen detrás suyo. Gracias a esto podemos decir que las aplicaciones OpenGL pueden ser fácilmente portadas virtualmente a cualquier plataforma del mercado, desde Windows a Linux, pasando por estaciones Unix de alto nivel y mainframes.

Por encima de todo, OpenGL es una biblioteca estilizada de trazado de gráficos de alto rendimiento, hay varias tarjetas gráficas aceleradoras y especializadas en 3D que implementan primitivas OpenGL a nivel hardware. Hasta hace poco estas avanzadas bibliotecas gráficas solían ser muy caras y solo estaban disponibles para estaciones SGI u otras estaciones de trabajo UNIX. Las cosas están cambiando muy deprisa y gracias a las generosas licencias y el kit de desarrollo de controladores de SGI vamos a ver más y más hardware OpenGL para usuarios de PCs.

Ventajas

Para conseguir la independencia del hardware de OpenGL no se incluyeron comandos para tareas de ventanas ni para obtener entrada del usuario. Esto suena como un serio contratiempo

pero tiene un simple arreglo. De hecho, para mantener el estilo y alto rendimiento de OpenGL, no se provee de comandos para describir modelos complejos, tales como moléculas, aeroplanos, casas, pájaros, etc. En OpenGL solo hay primitivas de objetos geométricos: puntos, líneas y polígonos. El desarrollador tiene que construir sus propios modelos basándose en unas pocas y simples primitivas. Hay bibliotecas relacionadas que proveen de modelos más complejos, y cualquier usuario se puede construir las suyas.

Algunas de las características que OpenGL implementa:

1. Primitivas geométricas permiten construir descripciones matemáticas de objetos. Las actuales primitivas son: puntos, líneas, polígonos, imágenes y bitmaps. Es decir, con OpenGL se puede trabajar con los objetos a un nivel muy bajo, pudiendo incluso modificar sus vértices. Es justo lo que necesito.

2. Codificación del color en modos RGBA (Rojo-Verde-Azul-Alfa) o de color indexado. Con esta opción puedo aplicar texturas semitransparentes a algunas zonas del invernadero (posteriormente lo usaría como método de reducción del número de polígonos cargados en escena).

3. Visualización y modelado que permite disponer objetos en una escena tridimensional, mover nuestra cámara por el espacio y seleccionar la posición ventajosa deseada para visualizar la escena de composición. El inconveniente es que es el propio programador el que tiene que desarrollar los movimientos y rotaciones de la cámara a través de un eje arbitrario cualquiera.

4. Mapeado de texturas que ayuda a traer realismo a los modelos por medio del dibujo de superficies realistas en las caras de nuestros modelos poligonales.

5. La iluminación de materiales es una parte indispensable de cualquier gráfico 3D. OpenGL provee de comandos para calcular el color de cualquier punto dadas las propiedades del material y las fuentes de luz en la habitación.

6. El doble buffering ayuda a eliminar el parpadeo de las animaciones. Cada fotograma consecutivo en una animación se construye en un buffer separado de memoria y mostrado solo cuando está completo.

7. El anti-aliasing reduce los bordes escalonados en las líneas dibujadas sobre una pantalla de ordenador. Los bordes escalonados aparecen a menudo cuando las líneas se dibujan con baja resolución. Es una técnica común en gráficos de ordenador que modifica el color y la intensidad de los píxeles cercanos a la línea para reducir el zig-zag artificial.

8. El Z-buffering mantiene registros de la coordenada Z de un objeto 3D. El Z-buffer se usa para registrar la proximidad de un objeto al observador, y es también crucial para el eliminado de superficies ocultas.

9. Efectos atmosféricos como la niebla, el humo y las neblinas hacen que las imágenes producidas por ordenador sean más realistas. Sin efectos atmosféricos las imágenes aparecen a veces irrealmente nítidas y bien definidas. Niebla es un término que en realidad describe un algoritmo que simula neblinas, brumas, humo o polución o simplemente el efecto del aire, añadiendo profundidad a nuestras imágenes.

10. El 'alpha blending' usa el valor Alfa (valor de material difuso) del código RGBA, y permite combinar el color del fragmento que se procesa con el del píxel que ya está en el buffer.

11. Las 'listas de display' permiten almacenar comandos de dibujo en una lista para un trazado posterior, cuando las listas de display se usan apropiadamente puedan mejorar mucho el rendimiento de nuestras aplicaciones.

12. Transformaciones: rotación, escalado, perspectivas en 3D

Como ya he comentado para hacer OpenGL verdaderamente portable e independiente de la plataforma fue necesario sacrificar todos los comandos que interactuaban con el sistema de ventanas, por ejemplo: abrir una ventana, cerrar una ventana, escalar una ventana, dar forma a una ventana, leer la posición del cursor; y también con los dispositivos de entrada como la lectura del teclado etc.. Todas estas acciones son altamente dependientes del sistema operativo.

Originalmente, la biblioteca GL tenía su propio conjunto de comandos para manejo de ventanas y periféricos pero eran específicos de IRIX (el SO UNIX de SGI). Se deja al desarrollador de OpenGL conocer su propia plataforma y tomar medidas para manejar ventanas en la plataforma nativa.

Gracias a Mark J. Kilgard de SGI hay una biblioteca adicional que evita este problema. Mark escribió la GLUT-library, un conjunto de herramientas y utilidades que sustituyen a la antigua biblioteca AUX. La biblioteca GLUT es libremente disponible y como Mesa (OpenGL) se puede encontrar código fuente para ella así como versiones binarias para Linux. La biblioteca GLUT es dependiente de la plataforma, y ofrece un paradigma común para el sistema de ventanas y dispositivos periféricos.

Así pues cuando una aplicación OpenGL quiere abrir una ventana para una animación gráfica usa el conjunto de comandos GLUT y éste toma el control del sistema de ventanas subyacente. De algún modo GLUT oculta al desarrollador los detalles 'sucios' de los sistemas de ventanas

específicos (X11, Windows, etc.) y le deja concentrarse en la tarea principal: el código OpenGL. Otra ventaja de usar GLUT es que hace el código independiente de la plataforma.

Inconvenientes

Uno de los pocos inconvenientes que existen en OpenGL es precisamente una ventaja para mi proyecto: hay que trabajar forzosamente a un nivel de abstracción muy bajo, incluso a nivel de vértice y coordenada de objeto, por lo que facilita poco el trabajo con objetos 3D, pero a su vez es lo que yo necesito ya que esta característica me ofrece una total libertad.

Conclusión

Estaba convencido de que sería una opción muy buena, y cuanto más profundizaba en conocimiento de OpenGL, más convencido estaba.

1.3.4. Direct3D

Direct3D (llamada DirectGraphics desde la versión 8 de DirectX) es una API que permite programar videojuegos 3D en Windows. Inicialmente Windows no ofrecía el rendimiento suficiente en el manejo de gráficos y sonido como para poder desarrollar videojuegos. En 1995 Microsoft sorprendió al mundo con DirectX, un conjunto de herramientas que permitía a los programadores manejar gráficos y sonidos con el mismo rendimiento que MS-DOS, y con la ventaja de poder despreocuparse de los drivers del sistema (puesto que cada fabricante provee sus propios drivers a Windows).

DirectX también tiene características muy interesantes que puedo usar en mi proyecto. Se compone de:

1. DirectGraphics: cálculo de graficos 2D y 3D.
2. DirectMusic: reproducción de sonido, música, mp3, etc...
3. DirectInput: manejo del teclado, ratón, joystick, volantes, etc...
4. DirectPlay: funciones de red para Internet, network, modem, etc...
5. DirectShow: permite reproducir videos en general, dvd, mpg, etc...

Ventajas

Direct3D, al igual que su rival OpenGL, tiene muchas ventajas que han hecho que esta API tenga un gran número de seguidores entre los programadores de videojuegos. Es una API muy potente:

1. Para empezar, una aplicación desarrollada con Direct3D va a poder ejecutarse en cualquier PC aunque no se disponga del último modelo de tarjeta gráfica, ya que Direct3D automáticamente emula esas características mediante software de forma que al menos el videojuego funcione.

2. Aunque existen otras APIs como OpenGL que también son muy buenas para programar aplicaciones 3D (como ya he comentado OpenGL es portable incluso a Linux), Direct3D ofrece más posibilidades porque incluye DirectMusic y DirectInput en la misma API. Por lo que Direct3D se convierte así en una API perfecta para programadores del entorno Windows que deseen realizar algún videojuego 3D.

3. Direct3D está basado totalmente en tecnología COM (al igual que otras muchas APIS que Microsoft ha lanzado). Básicamente COM es un componente software que provee servicios a través de interfaces, y una interfaz es un grupo de métodos relacionados. Un componente COM es generalmente un fichero .DLL que puede ser accedido de varias maneras definidas por las interfaces. Gracias a que COM es independiente del lenguaje, se pueden programar aplicaciones 3D desde Visual C++, Visual Basic o Delphi. Para comenzar a programar se usa el SDK de DirectX, que se puede descargar desde la página oficial de Microsoft y contiene las librerías listas para usar por el programador.

Inconvenientes

Sin embargo su principal inconveniente es la portabilidad. El hecho de que OpenGL sea multiplataforma, y que además iguale la potencia de Direct3D gracias al uso de algunas librerías adicionales (GLUT, SDL, etc) hace de OpenGL una opción mucho más atractiva. Y aunque mi proyecto lo voy a desarrollar para la plataforma Windows, puede que el día de mañana me interese comercializarlo, y por lo tanto me gustaría abarcar el máximo número de clientes posibles, y actualmente muchas empresas están emigrando a Linux por ser gratuito y ofrecer las mismas posibilidades o más que el sistema operativo de Microsoft.

Conclusión

En caso de tener que programar por mí mismo el engine 3D, elegiría OpenGL antes que Direct3D, por el aspecto de la portabilidad.

1.3.5. Blitz3D

Blitz3D es un lenguaje de programación (a diferencia de OpenGL y DirectX que son APIs) que nació con la idea de llevar el entorno de programación del Blitz Basic 2D original al campo de las tres dimensiones sin dejar en ningún momento la filosofía de sencillez y potencia que siempre

ha caracterizado a esta herramienta de desarrollo multimedia.

Algunas de sus características principales que pueden interesarme para el desarrollo de mi proyecto son las siguientes:

1. Se han conservado todas las funciones 2D y se le ha añadido un grupo, bastante generoso, de instrucciones para el manejo de las 3D. Con toda esta funcionalidad ahorraría mucho trabajo, ya que partiría de una base muy buena y no completamente desde cero.

2. Al igual que su antecesor, el corazón de Blitz3D es un entorno de desarrollo integrado (IDE) donde se puede escribir, testear y ejecutar programas. Dispone de una extensa gama de posibilidades, entre las que se encuentra un gran depurador de código y un visualizador de variables que ayuda al programador, en cualquier momento, a obtener información del proceso de ejecución y poder corregir rápidamente errores en el código. Todos estos detalles son importantes a la hora de enfrentarse a un proyecto de programación como el mío.

3. Gracias a que el compilador produce 100 por 100 código máquina, se obtiene un rendimiento de velocidad más que satisfactorio en los ejecutables. También los videojuegos pueden ejecutarse a pantalla completa o en una ventana de escritorio configurable. Es muy deseable que la visita virtual al invernadero 3D sea lo más fluida posible, por lo que cualquier mejora en el rendimiento de la aplicación es bienvenida.

4. En lo que respecta al formato de los archivos, ofrece una amplia gama de posibilidades, desde el formato de DirectX (.X) al de 3D Studio (.3DS), con la posibilidad de animación por deformación de malla (.MD2). Usando Blitz3D ahorraría bastante trabajo porque no tendría que implementar un cargador de objetos, ya está implementado.

5. Pero quizá el punto fuerte de Blitz3D es la gran potencia de su motor gráfico (es lo que más me interesa) ya que es capaz de:

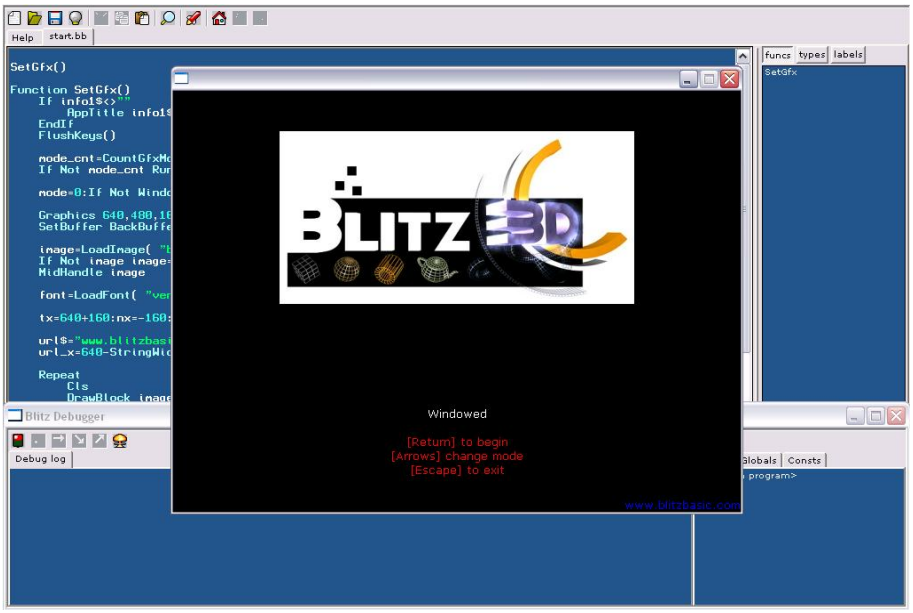
   a) Transformar y mover objetos en 3D y 2D.

   b) Detectar colisiones entre objetos.

   c) Aplicarles texturas y multitud de efectos de visualización.

   d) Puede manejar y transformar terrenos, manipular luces en tiempo real, añadir efectos especiales como niebla o transparencias.

   e) Incluye la posibilidad de optimizar el rendimiento con la utilización del nivel de detalle o LOD, que básicamente consiste en reducir los polígonos de los objetos a medida que se alejan de la cámara.



Ventajas

Son muchos los aspectos positivos que posee este lenguaje de programación. Los principales que he encontrado relacionados con mi proyecto son los siguientes:

1. El engine 3D ya está implementado, y aspectos como la detección de colisiones y las transformaciones de objetos 3D me serían muy útiles en la visita virtual a los invernaderos.

2. La calidad gráfica que ofrecen las creaciones de Blitz3D son espectaculares.

3. La carga de objetos 3D también está implementada.

El trabajo que ahorraría usando Blitz3D sería para tenerlo en consideración, ya que tiene implementada al igual que ocurría con Blender, muchísimos detalles que necesito para mi proyecto.

Inconvenientes

Como se puede ver este lenguaje de programación ofrece muchas posibilidades, y la verdad me hubiera ahorrado mucho trabajo a la hora de realizar el proyecto. Sin embargo existen algunos inconvenientes que me han hecho renunciar a Blitz3D:

1. En primer lugar hay comprarlo, y aunque el precio no es muy elevado (145 euros), sería una lástima tener que comprar una herramienta para usarla sólo una vez en el departamento, ya que no creo que se oferten muchos proyectos fin de carrera relacionados con la construcción o manejo de un engine 3D. Además hay que tener en cuenta que mi proyecto sí se puede desarrollar con otras herramientas que el departamento ha comprado (Microsoft Visual C++ por ejemplo).

2. Además hay cientos de engines 3D por Internet que ofrecen la misma o incluso más funcionalidad, y son freeware, por lo que sería un desperdicio de dinero pudiendo recurrir a los recursos gratuitos de la red.

3. Aunque es un lenguaje muy bueno y potente, no ha llegado a utilizarse tanto como por ejemplo las APIs OpenGL y Direct3D.

4. Como yo necesito trabajar al nivel más bajo posible (para la transformación y creación dinámica del invernadero), no estaba muy convencido de con Blitz3D pudiera acceder a los objetos tridimensionales a nivel de polígono, incluso de vértice, y poder modificar coordenadas a mi antojo.

5. Tenía miedo de que una herramienta tan buena en principio como Blender, pudiera volver a dejarme a medias con mi proyecto.

Conclusión

Las opciones anteriores me convencían mucho más que ésta.

1.3.6. VRML

El VRML (Virtual Reality Modeling Language) es un lenguaje de modelado de mundos virtuales en tres dimensiones. Sirve para crear mundos 3D a los que se pueden acceder utilizando el propio navegador, igual que una página web cualquiera (con la ayuda de un pluggin). Además las visitas no se limitan a ver un simple texto y fotografías, sino que nos permite ver todo tipo de objetos y construcciones en 3D por los que podemos pasear o interactuar.

Ventajas

1. El engine 3D ya está implementado, con muchas características deseables: detección de colisiones, movimiento de la cámara a través del mundo 3D, etc...

2. El movimiento se puede realizar en todas las direcciones, no solo izquierda y derecha sino también hacia delante, atrás, arriba y abajo.

3. Se puede interaccionar con los objetos como en la vida misma, tocarlos, arrastrarlos, etc. Además los escenarios parecen muy reales.

4. Además para su desarrollo son necesarios muy pocos recursos:

   a) Un editor de textos en modo ASCII sencillo, como por ejemplo el Block de notas de Windows. Aunque existen también otros editores más especializados, como el VRML PAD.

   b) Visualizador VRML, que se instalar en el navegador como cualquier otro pluggin. Uno de los más importantes es el famoso Cosmo Player.

Como en el caso de Blender y Blitz3D, con VRML me ahorraría mucho trabajo, ya que la parte más compleja del proyecto (el engine 3D) ya está implementado en el pluggin que se aplica al navegador.

Inconvenientes

Sin embargo existen algunos inconvenientes demasiado negativos como para no tenerlos en cuenta:

1. Los objetos tienen un formato diferente y específico para VRML, y además no se puede trabajar con ellos tan bien como por ejemplo con OpenGL o Direct3D. Eso era muy negativo para el desarrollo de mi aplicación.

2. Su funcionalidad queda muy corta para realizar mi proyecto.

Conclusión

VRML es genial para realizar visitas virtuales con interacción, pero a modelos estáticos, y yo necesito poder trabajar incluso a nivel de vértice con los objetos para poder modificarlos dinámicamente.

1.3.7. Decisión final

Las características principales de cada una de las tecnologías 3D que he estudiado las he resumido en la siguiente tabla:

	Blender	Engines 3D	OpenGL	Direct3D	Blitz3D	VRML
Detecta colisiones	Si	Si/No	No	No	Si	Si
Multiplataforma	Si	Si/No	Si	No	No	Si
Carga de objetos	Si	Si/No	No	Si	Si	Si
Gratuita	Si	Si	Si	Si	No	Si
Visita implementada	Si	Si	No	No	Si	Si
Interacción objetos	Si	No	No	No	Si	Si
Trabaja a bajo nivel	No	No	Si	Si	No	No
Cambios dinámicos	No	No	Si	Si	No	No
Multilenguaje	No	Si	Si	Si	No	No

Como se puede observar las tecnologías que a priori ofrecen mayores prestaciones son Blender y VRML. Sin embargo con Blender ya intenté desarrollar el proyecto y descubrí que no me permitía trabajar a un nivel de abstracción bajo (nivel de polígono e incluso de vértice) para poder realizar los cambios dinámicos de la forma del invernadero. Y es una verdadera lástima porque realmente hubiera ahorrado mucho trabajo al tener ya implementado el engine 3D que detecta colisiones, mueve y rota la cámara, y la carga de objetos 3D. Aunque el lenguaje de scripts Python ofrece cierta libertad, no terminaba de convencerme completamente. Yo necesitaba mucha más libertad. Por lo tanto, aunque Blender es una herramienta muy completa (al igual que el lenguaje VRML) no es la adecuada para el desarrollo de este proyecto.

Utilizar algún engine 3D puede que me hubiera dado más trabajo que realizarlo yo desde cero, ya que la mayoría de ellos ofrecían muchísima más funcionalidad de la que yo realmente necesito, y además los códigos que descargué no estaban debidamente comentados, luego no estoy seguro de haber controlado completamente el engine 3D (requisito indispensable para el desarrollo de este proyecto).

En lo que respecta al lenguaje Blitz3D ocurría algo similar a Blender, ofrece mucha funcionalidad (mucha más de la que necesito) pero no permite trabajar a un nivel de abstracción bajo. Además no es una herramienta gratuita, y hubiera sido una lástima realizar el proyecto con Blitz3D habiendo tantas herramientas 3D o APIs de libre distribución que ofrecen las mismas o incluso más prestaciones.

Finalmente tuve que elegir entre Direct3D y OpenGL. Me incliné hacia OpenGL por ser multiplataforma y por permitir trabajar a un nivel de abstracción muy básico. Esto tiene sus

ventajas y sus inconvenientes. La ventaja es que se puede hacer con los objetos prácticamente lo que uno quiera. El inconveniente es que se debe programar todo desde cero: la carga de objetos 3D en el formato deseado, la modificación de la forma del objeto, etc... Pero es necesario para adaptar el invernadero al polígono creado por el usuario en la interfaz inicial.

Por lo tanto después de este estudio de las tecnologías 3D actuales acabé decantándome por usar OpenGL. Otro inconveniente de mi decisión ha sido el tener que programar yo desde cero el miniengine 3D para la visita virtual al invernadero tridimensional. Me ha llevado mucho tiempo, pero también hay otro motivo que me llevó a crear mi propio engine: siempre he tenido mucha curiosidad por saber cómo funciona realmente un videojuego, así que profundizando en la materia conseguiría además cumplir una satisfacción personal.

Además de usar OpenGL también tenía que elegir qué librerías auxiliares me servirían de ayuda. En mi caso he usado la librería SDL, ya que ofrece muy buen rendimiento sobre todo en frames por segundo. Con esta librería capturaré los eventos del teclado y ratón, etc..

Una vez me había decidido por OpenGL aparecieron nuevos problemas. Tenía que centrarme en la construcción del engine 3D desde cero y algunos requisitos que debía contemplar eran los siguientes:

1. Carga de objetos 3D (elección del formato adecuado).
2. Detección de colisiones de la cámara contra las paredes del invernadero.
3. Movimiento de cámara a través del mundo virtual con el teclado.
4. Rotación de la cámara con el ratón.

La solución a estos problemas la documento en la primera sección del capítulo 2. En esa sección describo detalladamente las adversidades a las que me enfrenté y también las soluciones que ofrezco. El movimiento de la cámara y su rotación, así como la detección de colisiones son las funciones que más he intentado documentar a través de su justificación matemática, porque son funciones bastantes complejas desde el punto de vista algorítmico.

Pero antes de comenzar con el desarrollo del proyecto en sí tenía que realizar una planificación temporal del mismo, para poder trabajar siguiendo unas fechas aproximadas de finalización, comprobar las estimaciones de esfuerzo iniciales y finalmente comparar la estimación con el trabajo realizado.

En la siguiente sección me centraré en el plan del proyecto y aplicaré toda una metodología existente para poder estudiar aspectos tan importantes como el ámbito del software, descomposición funcional del sistema, etc.

Aunque todas estas estimaciones no son obligatorias para el desarrollo del proyecto, sí son muy recomendables ya que permiten por ejemplo establecer una posible fecha de finalización del mismo.

1.4. Plan de proyecto

Antes de desarrollar un proyecto software ambicioso es necesario realizar una serie de estudios para poder estimar algunas métricas importantes tales como las siguientes:

- Líneas de código esperadas.
- Esfuerzo en personas-mes necesario.
- Duración del proyecto.
- Planificación temporal del proyecto.

Todas estas métricas no son totalmente objetivas, sino que se cuantifican de una manera aproximada, pero sirven para hacernos una idea de: el número de personas que deben trabajar en el proyecto, la duración del mismo, las fechas tope de entrega intermedias de las que disponemos, etc. De esta manera el jefe de proyecto puede realizar valoraciones del estado actual del trabajo y puede comprobar si todo se está realizando como se esperaba. Para planificar mi proyecto voy a seguir los siguientes pasos:

1. Configurar el método de trabajo.
2. Detectar los elementos de información básicos que hacen falta para desarrollar el proyecto.
3. Fijar las fuentes de información a utilizar.
4. Delimitar el ámbito del software.
5. Realizar la descomposición funcional del sistema y el diagrama WBS.
6. Obtener las primeras estimaciones: con el método de líneas de código y puntos de función.
7. Aplicar COCOMO básico, COCOMO intermedio y Putnam.
8. Realizar una planificación temporal del proyecto.

Todo este trabajo además ayuda a introducirse en el ámbito del proyecto que se desea desarrollar, y a tener un control más preciso y mayor conocimiento de las tareas que se llevarán a cabo.

1.4.1. Configurar el método de trabajo

En el caso de mi proyecto el grupo de trabajo estará constituido sólo por mí, ya que voy a desarrollar todo el trabajo sin ayuda de otros programadores. Para el desarrollo del software voy a seguir el modelo espiral, propuesto por Boehm, que es a su vez una mezcla del modelo lineal secuencial y el modelo de construcción de prototipos. Elijo este modelo porque es uno de los que ofrecen resultados más rápidamente a través de los primeros prototipos del programa, resultados que pueden mejorarse con las sucesivas versiones incrementales del mismo, de tal modo que cada versión será más completa que la anterior. Además, en el desarrollo de todo proyecto de fin de carrera siempre quedan aspectos o ideas en el tintero que pueden ser añadidas en trabajos posteriores, y por lo tanto la versión final de mi proyecto podría considerarse como mi última versión, pero usando este modelo de desarrollo de software siempre se podría incrementar su funcionalidad en posteriores trabajos. Es decir, el proceso no termina cuando se entrega el programa, sino que puede aplicarse y adaptarse a lo largo de toda la vida del software.

Aunque en mi proyecto voy a modelar únicamente un tipo de invernadero, una ampliación obvia sería por ejemplo añadir más tipos nuevos y por lo tanto el modelo espiral de desarrollo de software ofrece un enfoque muy realista de la evolución de mi aplicación.

1.4.2. Detectar los elementos de información básicos que hacen falta para desarrollar el proyecto

En este apartado he analizado qué información me hace falta para desarrollar el proyecto. Dicha información la he clasificado en varios grupos:

1. Información relacionada con los invernaderos:
   - *a*) Tipos de invernaderos que existen.
   - *b*) Elementos básicos que constituyen un invernadero.
   - *c*) Variedades en los materiales de construcción.
   - *d*) Estándares usados en la construcción.
   - *e*) Métodos de construcción de un invernadero.
   - *f*) Adaptabilidad de un invernadero a un terreno concreto.
2. Información relacionada con la informática gráfica:
   - *a*) Formatos estándar de archivos 3D.
   - *b*) Modificación de objetos 3D a nivel de polígono.
   - *c*) Aplicaciones para creación de modelos 3D (Autocad y 3D Studio).

d) Conocimiento necesario para realizar visitas virtuales con OpenGL.

e) Rendimiento ofrecido por la tecnología OpenGL.

f) Conocimiento para realizar la detección de colisiones.

1.4.3. Fijar las fuentes de información a utilizar

Para obtener toda la información necesaria voy a tener que consultar una serie de fuentes, que pueden ser de varios tipos:

1. Personas:

   a) Directores de mi proyecto

   b) Ingenieros agrónomos

   c) Ingenieros informáticos

2. Otros medios:

   a) Bibliografía: construcción de invernaderos y de programación gráfica 3D, etc.

   b) Páginas web: documentos digitales, foros, chats, etc.

1.4.4. Delimitar el ámbito del software a desarrollar

Antes de poder comenzar con la planificación de un proyecto deben establecerse el ámbito y los objetivos, deben considerarse soluciones alternativas y deben identificarse restricciones técnicas y de gestión. Sin ésta información es imposible obtener estimaciones de coste razonables y precisas, una identificación realista de las tareas del proyecto o un plan de trabajo adecuado. El desarrollador de software y el cliente deben ponerse de acuerdo para definir el ámbito y los objetivos. Los objetivos son identificar los fines globales del proyecto sin considerar como se llegará a esos fines.

El ámbito identifica las funciones primordiales que debe llevar a cabo el software y, lo que es más importante, intenta delimitar esas funciones de manera cuantitativa. El ámbito describe la función, el rendimiento, las restricciones, las interfaces y la fiabilidad.

Descripción funcional

El producto que voy a desarrollar se trata de una aplicación software que construye una estructura de invernadero 3D a partir de una serie de parámetros introducidos por el usuario así como de la forma que tendrá dicho invernadero, y posteriormente permite realizar una visita virtual a la estructura previamente creada. Como se puede observar a primera vista, mi proyecto tiene dos funcionalidades bien diferenciadas: por un lado la construcción del invernadero y por

otra la visita virtual al mismo.

En primer lugar deberé crear una interfaz de usuario para poder introducir los datos del invernadero a construir, y una vez insertados se deberá crear una estructura tridimensional ajustándose a dichos parámetros.

En segundo lugar crearé una versión reducida de un engine 3D para poder visitar virtualmente dicho invernadero, y se podrá recorrer completamente, incluso se podrán realizar rotaciones con la cámara para 'mirar' hacia arriba, abajo, izquierda y derecha.

Y además habrá otra serie de funcionalidades secundarias, como por ejemplo cambios dinámicos en la visita virtual a través de una consola de texto para poder modificar algunas distancias entre objetos o cambiar las texturas de algunos polígonos.

Rendimiento

En este ámbito el rendimiento está determinado sobre todo por el número de polígonos que se van a cargar en escena en la visita virtual, ya que si éste es excesivo se podría ralentizar el movimiento de la cámara y la visita no sería fluida sino que iría dando saltos. Por lo tanto gran parte del esfuerzo deberé emplearlo en minimizar lo máximo dicho número de polígonos, así como establecer restricciones para no poder cargar más de un número determinado en escena.

Existe una cierta complejidad algorítmica, sobre todo en la parte de la visita virtual para el movimiento de la cámara a través del mundo tridimensional. En este caso deberé esforzarme para intentar ofrecer el mayor rendimiento posible, medido en fotogramas por segundo.

Restricciones

En este caso puede haber restricciones de rendimiento si no se posee una CPU con una frecuencia de reloj elevada o una tarjeta gráfica adecuada. También será importante la memoria del PC, ya que se deberá almacenar en memoria dinámica la estructura del invernadero durante la visita virtual.

Interfaces

En mi caso voy a tener dos interfaces, una para el empresario (dueño de la empresa de construcción de invernaderos) y otra para el cliente (que quiere que le construyan el invernadero). El primero se encargará de crear mediante el programa el invernadero 3D, introduciendo los parámetros que le ofrezca el cliente así como la forma deseada. El segundo podrá visitar virtualmente la estructura creada, usando para ello el teclado y el ratón.

1. Interfaz del empresario

   a) Hardware: el software es ejecutado por la CPU del PC. El dispositivo controlado indirectamente por el software es la pantalla.

   b) Software: en este caso no hay software ya existente y que pueda ser integrado en mi sistema, sino que lo desarrollare yo mismo.

   c) Persona: el empresario hará uso de esta interfaz a través del ratón y del teclado (dispositivos de entrada/salida).

   d) Procedimientos: el empresario marcará con ratón las opciones deseadas para ir creando la forma del invernadero.

2. Interfaz del cliente

   a) Hardware: el software es ejecutado por la CPU del PC. El dispositivo controlado por el software es la pantalla.

   b) Software: lo desarrollaré también yo.

   c) Persona: el cliente a través del teclado y ratón.

   d) Procedimientos: el cliente se desplazará por el mundo virtual a través del ratón y el teclado.

Fiabilidad

La fiabilidad que se espera tener de mi sistema no es baja, pero al no tratarse de un software del que dependan vidas humanas, tampoco se exige una fiabilidad demasiado elevada. Cuanto mayor seguridad aplique a los procedimiento, mayor esfuerzo necesitaré para conseguirla y por tanto aumentará el coste total del proyecto.

1.4.5. Descomposición funcional del sistema

Antes de empezar a diseñar y codificar una aplicación hay que realizar una descomposición de toda la funcionalidad del mismo, para ir dividiendo poco a poco las distintas tareas que se deberán realizar. En mi descomposición he detectado dos funciones principales: al interfaz y la visita virtual.

Por un lado la interfaz debe dibujarse y crear la forma del invernadero. Además también deberá realizar la transformación del polígono 2D dibujado por el usuario a la estructura 3D final para la visita virtual. Estas funciones las he subdividido, y en el capítulo 3 explico qué hace cada una de las funciones básicas. Por otro lado la visita virtual se encargará de dibujar los objetos 3D en escena y del movimiento de la cámara a través del mundo creado, así como la modificación de parámetros a través de la consola de texto.

1. Interfaz:
   - *a*) Dibujar interfaz:
     1) Inicializar entorno gráfico en OpenGL.
     2) Dibujar elementos estáticos de la interfaz.
     3) Escribir texto en la interfaz.
     4) Dibujar plano del invernadero en la interfaz (las cuadrículas).
   - *b*) Crear la forma del invernadero:
     1) Limitar el cursor del ratón a la cuadrícula.
     2) Capturar eventos de teclado y ratón.
     3) Aproximar posición del cursor del ratón a punto de cuadrícula.
     4) Comprobar cruces entre líneas.
     5) Marcar punto de la cuadrícula.
     6) Dibujar líneas que muestran la forma del invernadero.
   - *c*) Transformación 2D a 3D:
     1) Buscar cuadrículas completas
        - a' Comprobar las cuadrículas internas al polígono.
        - b' Comprobar si las cuadrícula internas están atravesadas.
        - c' Añadir modelo básico de Autocad por cada cuadrícula completa.
     2) Buscar cuadrículas incompletas
        - a' Creación de techos
        - b' Buscar los 4 puntos de los techos.
        - c' Algoritmo de Bresenham para trazado de líneas.
        - d' Transformación de la forma de los techos.
        - e' Creación de paredes.
        - f' Completar invernadero
        - g' Añadir pilares en las zonas incompletas.
        - h' Añadir hierros horizontales en zonas incompletas.
        - i' Añadir barras curvas en zonas incompletas.

2. Visita virtual:
   - *a*) Visualizar elementos del mundo 3D:
     1) Utilizar objeto 3D:
        - a' Cargar objetos ASC de disco.

b' Representar los objetos de la visita virtual.

2) Utilizar texturas:

a' Cargar imagen BMP desde disco.

b' Añadir canal de transparencia Alpha a la textura en BMP.

c' Mapear las texturas sobre los polígonos.

3) Dibujar elementos adicionales:

a' Dibujar la consola de texto para capturar comandos.

b' Calcular el número de fotogramas por segundo.

c' Dibujar el logotipo de la UAL.

b) Movimiento de la cámara:

1) Posicionar cámara en un punto concreto.
2) Desplazar cámara hacia delante y hacia atrás.
3) Desplazar cámara lateralmente.
4) Desplazar cámara hacia arriba y abajo.
5) Rotar la cámara con el movimiento del ratón.
6) Implementar operaciones con vectores.
7) Detección de colisiones.

c) Trabajo con la consola de texto:

1) Modificación dinámica de la distancia entre pilares.
2) Modificación de las texturas de la cubierta.
3) Añadir efecto de niebla.
4) Visualizar objetos en modo alambre.

Con la descomposición funcional del sistema he comenzado realmente a diseñar el aspecto de lo que será la aplicación definitiva. He tomado cada función y la he dividido hasta que he creído oportuno, hasta obtener las funciones que creo que serán básicas. Sin embargo esta es una primera aproximación, ya que algunas funciones que creía básicas luego tendré que seguir descomponiéndolas en otras. Esto ocurre como veremos en capítulos posteriores con la detección de colisiones, que no es una tarea tan sencilla como estimé inicialmente.

En el capítulo 3 además incluyo un diagrama de clases donde se muestra visualmente el funcionamiento del sistema. A continuación muestro un diagrama WBS con las funciones obtenidas:

Proyecto
Interfaz
Dibujar interfaz
Limitar cursor
Crear la forma del invernadero
Capturar eventos
Aproximar cursor
Comprobar cruces
Marcar punto
Dibujar líneas
Inicializar OpenGL
Dibujar estáticos
Escribir texto
Dibujar plano
Visita virtual
Transformación 2D a 3D
Cuadrícula completa
Cuadrícula incompleta
Cuadrículas internas
Cuadrículas atravesadas
Añadir modelo básico
Creación de techos
Buscar 4 puntos techo
Bresenham para líneas
Transformación techo
Creación de paredes
Completar invernadero
Añadir pilares
Añadir hierros
Añadir barras
Visualizar elementos 3D
Cargar objeto ASC
Dibujar consola
Calcular FPS
Dibujar logo UAL
Dibujar objetos
Cargar imagen
Añadir alpha
Mapear textura
Movimiento de cámara
Posicionar cámara
Desplazar cámara
Ladear cámara
Volar cámara
Rotar cámara
Operación vectores
Detección colisiones
Consola de texto
Distancia pilares
Texturas cubierta
Efecto de niebla
Modo alambre

1.4.6. Estimaciones basadas en puntos de función y líneas de código

En primer lugar realizo las estimaciones por puntos de función y líneas de código para posteriormente utilizar algunos modelos de estimación basándome en los datos obtenidos. Las líneas de código (LDC) y puntos de función (PF) son medidas básicas a partir de las que se pueden calcular métricas de productividad, pero son técnicas de estimación totalmente distintas. Cada métrica tiene sus ventajas y sus inconvenientes.

En el caso de las líneas de código, la gran desventaja es que es totalmente dependiente del lenguaje que uso actualmente, luego si cambio de lenguaje de programación no servirán los datos históricos de proyectos anteriores, a no ser que haga una posible equivalencia entre el número de líneas de código necesarias de uno y otro lenguaje y ajuste los valores del pasado a los del nuevo lenguaje. Además la persona que realiza la estimación debe tener bastante experiencia en el lenguaje elegido precisamente para poder hacer una planificación del número de líneas de código aproximadas que serán necesarias para desarrollar cada una de las funciones básicas que se han obtenido en la descomposición funcional.

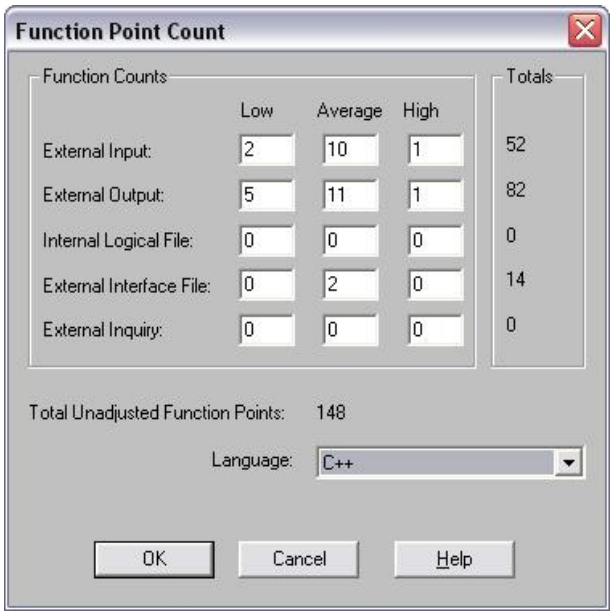
Los puntos de función en cambio dependen más del programa en sí, es decir, de la funcionalidad propiamente dicha, luego es independiente del lenguaje de programación usado, y por tanto más útil de cara a usar como dato de estimación de proyecto histórico. Para realizar la estimación mediante puntos de función también usaré la descomposición funcional del apartado anterior, y para cada tarea en principio básica iré asignando los valores oportunos.

Puntos de función

Los puntos de función miden la aplicación desde una perspectiva del usuario, dejando de lado los detalles de codificación. Es una técnica totalmente independiente de todas las que se consideran técnicas de lenguaje.

Para realizar una estimación en PF se estiman cada una de las características del dominio de la información: entradas, salidas, archivos de datos, peticiones e interfaces externas, y los catorce valores de complejidad asociados.

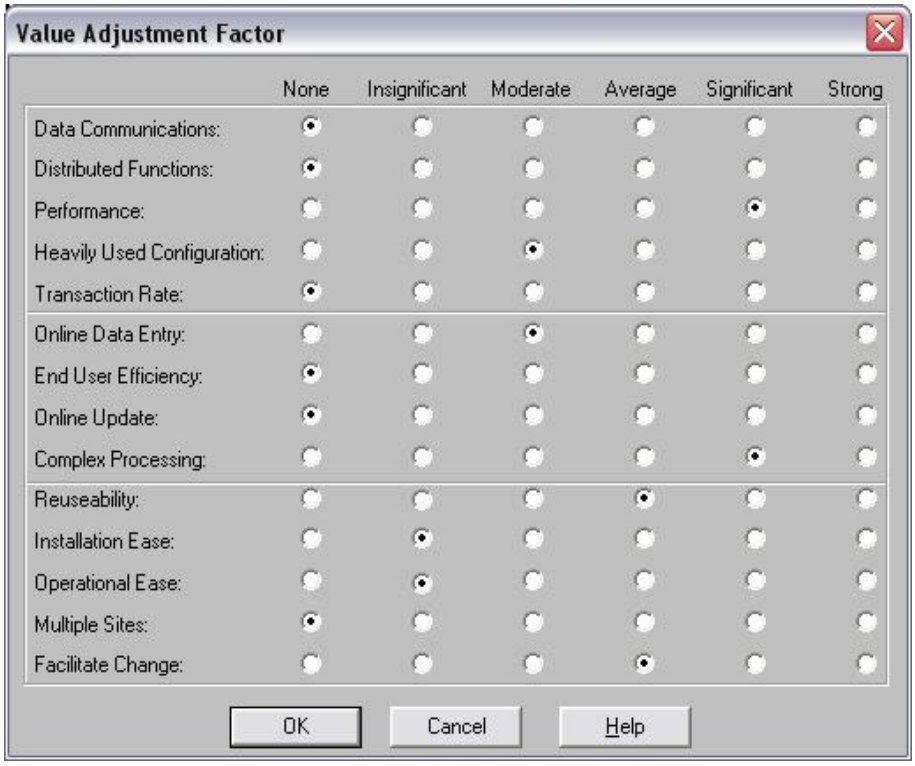
Para estimar los PF lo que hacemos en primer lugar es construir la tabla de valores correspondiente al dominio de la información, y estimarla. La siguiente imagen muestra la estimación con el programa COSMOS:



Para realizar esta primera estimación he ido asignando valores a cada una de las tareas básicas de la descomposición funcional y luego los he sumado. Como se puede observar lo que más predomina en mi proyecto son las entradas y salidas. Algunas entradas por ejemplo son sencillas ya que sólo requiere que el usuario pulse alguna tecla, otras en cambio son más complejas como por ejemplo la rotación de la cámara con el ratón. Lo mismo ocurre con las salidas hacia el usuario.

En mi caso al no tener una base de datos asigno un valor nulo a la tercera y quinta características del dominio de la información. Por lo tanto a algunas de la tareas básicas de la descomposición funcional no se les ha asignado ningún valor para las características del dominio de la información. Aunque a priori parezca extraño, posteriormente se ajustarán los resultados obtenidos gracias a los valores de complejidad de software.

El número de puntos de función obtenidos es 148, y el número de líneas de código en lenguaje C++ es de 5098. Sin embargo todavía se pueden ajustar más parámetros, para poder realizar una estimación de una manera mucho más precisa. Estos parámetros son los 14 valores de complejidad del software, y yo los he configurado como muestro a continuación:



En mi caso no se requiere comunicación de datos ni tampoco existen funciones de procesamiento distribuido, por lo tanto los dos primeros valores de complejidad están descartados. Sin embargo sí se requiere que tenga un buen rendimiento (la parte de la visita virtual para que sea lo más fluida posible), por lo que le he asignado un gran al tercero. Mi aplicación se va a ejecutar en un entorno operativo existente y fuertemente utilizado como es Microsoft Windows, por lo que el cuarto valor lo establezco como moderado (ya que se podrá ejecutar en Windows ME, XP, NT, etc.. y pueden surgir problemas). Además existe entrada interactiva de datos, y la complejidad del procesamiento interno del código fuente va a ser bastante elevada. También lo voy a diseñar para que pueda ser reutilizado, para que sea fácil de instalar y de utilizar, e intentaré que sea sencillo realizar cambios en la aplicación. Finalmente los resultados obtenidos tras el ajuste son:

Title:	**Diseño y visita virtual de invernaderos 3D**
Prepared By:	**Moisés**
Description:	**Proyecto fin de carrera**
Unadj. Function Points:	**148.0**
Value Adjustment Factor:	**0.85**
Adj. Function Points:	**125.8**
Language:	**C++ (53 SLOC/FP)**
Source Lines of Code:	**6667.4**

Sale un total de 6667 líneas de código, bastantes más que antes de ajustar los factores de

complejidad.

Líneas de código

Este tipo de estimación es si cabe más objetiva que la anterior, ya que si el analista posee experiencia en el lenguaje de programación, puede realizar una estimación mucho más precisa que usando los puntos de función.

La tabla siguiente muestra la estimación realizada para mi proyecto basada en LDC. Para cada una de las funciones básicas estimo el número de líneas de código mínimo, el medio y el máximo, y luego hago la media para finalmente sumar los resultados.

Como se puede observar existen algunas funciones básicas que poseen una estimación muy pequeña (20 líneas de código), mientras que existen otras que poseen bastantes más (por ejemplo la 'detección de colisiones' y la 'representación de los objetos tridimensionales en escena' tienen unas 500). Esto significa que seguramente esas funciones, en un principio estimadas como básicas, serán divididas en el futuro en otras más, ya que no es elegante tener un método en C++ con 500 líneas de código. Por lo tanto conforme vaya avanzando en el diseño del proyecto seguramente iré descomponiendo esas funciones en otras más básicas.

Hay que tener en cuenta que ahora simplemente estoy realizando una estimación, que no es otra cosa que una aproximación del resultado final. Por lo tanto estos datos son sólo orientativos, ya que podré obtener muchas más líneas de código, o incluso muchas menos.

A continuación muestro la tabla de estimación basada en líneas de código, clasificada por las funciones básicas:

Función	Baja	Media	Alta	LDC
Inicializar entorno gráfico en OpenGL	50	80	120	84
Dibujar elementos estáticos de la interfaz	150	200	230	194
Escribir texto en la interfaz	20	40	50	36
Dibujar plano del invernadero en la interfaz (las cuadrículas)	40	50	60	50
Limitar el cursor del ratón a la cuadrícula	10	15	25	16
Capturar eventos de teclado y ratón	150	210	240	200
Aproximar posición del cursor del ratón a punto de cuadrícula	20	30	40	30
Comprobar cruces entre líneas	60	80	100	80
Marcar punto de la cuadrícula	20	30	40	30
Dibujar líneas que muestran la forma del invernadero	30	50	70	50
Comprobar las cuadrículas internas al polígono	200	250	300	250
Comprobar si las cuadrícula internas están atravesadas	80	90	120	96
Añadir modelo básico de Autocad por cada cuadrícula completa	20	30	40	30
Creación de techo	200	250	300	250
Buscar los 4 puntos de los techos	270	300	350	306
Algoritmo de Bresenham para trazado de líneas	70	90	120	90
Transformación de la forma de los techos	300	400	500	400
Creación de paredes	100	150	200	150
Añadir pilares en las zonas incompletas	100	125	150	125
Añadir hierros horizontales en zonas incompletas	100	125	150	125
Añadir barras curvas en zonas incompletas	100	125	150	125
Cargar objetos ASC de disco	50	80	100	76
Representar los objetos de la visita virtual	450	500	550	500
Cargar imagen BMP desde disco	60	70	100	80
Añadir canal de transparencia Alpha a la textura en BMP	15	20	30	21
Mapear las texturas sobre los polígonos	10	20	30	20
Dibujar la consola de texto para capturar comandos	30	50	70	50
Calcular el número de fotogramas por segundo	200	250	300	225
Dibujar el logotipo de la UAL	10	20	30	20
Posicionar cámara en un punto concreto	20	30	40	30
Desplazar cámara hacia delante y hacia atrás	40	50	60	50
Desplazar cámara lateralmente	100	125	150	125
Desplazar cámara hacia arriba y abajo	30	40	50	40
Rotar la cámara con el movimiento del ratón	200	250	300	250
Implementar operaciones con vectores	200	300	400	300
Detección de colisiones	400	450	500	450
Modificación dinámica de la distancia entre pilares	200	300	400	300
Modificación de las texturas de la cubierta	200	250	300	250
Añadir efecto de niebla	50	75	100	75
Visualizar objetos en modo alambre	10	20	30	20

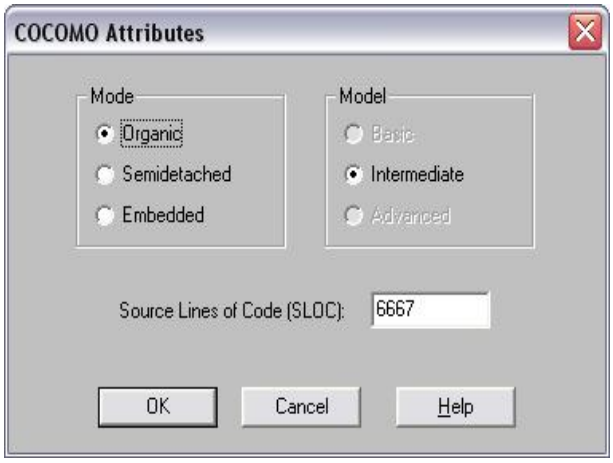
El número total de líneas de código fuente es de 5599, unas mil líneas de código fuente menos que con la estimación basada en puntos de función.

1.4.7. Modelo COCOMO básico

El 'modelo constructivo de costes' (COnstructive COst MOdel) fue desarrollado por B. W. Boehm a finales de los 70 y comienzos de los 80. COCOMO es una jerarquía de modelos de estimación de costes software que incluye submodelos básico, intermedio y detallado. En este primer caso usaré el COCOMO básico. Este modelo intenta estimar de una manera rápida y más o menos burda la mayoría de proyectos pequeños y medianos.

En primer lugar debo realizar una clasificación de mi proyecto en alguno de los tres tipos que se ofrecen: modo orgánico, modo semi-acoplado y modo empotrado. Mi proyecto se incluye en el primer tipo, modo orgánico, ya que se trata de un proyecto dentro de lo que cabe sencillo con un solo programador, y además el número de LDC es muy inferior a 50000 (límite que a veces diferencia los proyectos orgánicos de los semi-acoplados).

De las estimaciones del número de líneas de código realizadas en el apartado anterior me quedaré con el valor más elevado: el obtenido mediante puntos de función (6667 líneas de código), ya que prefiero estimar por exceso que por defecto.



Los resultados obtenidos en COSMOS son los siguientes:

Title:	**Diseño y visita virtual de invernaderos 3D**
Prepared By:	**Moisés**
Description:	**Proyecto fin de carrera**
Source Lines of Code:	**6667.0**
Nominal Effort:	**23.5 person months**
Adjusted Effort:	**23.5 person months**
Time to Develop:	**8.3 calendar months**

Donde los resultados se han obtenido de la siguiente manera:

$$E = 2{,}4 * 6667^{1{,}12} = 23{,}5 personas - mes$$

$$D = 2{,}5 * 23{,}5^{0{,}38} = 8{,}3 meses$$

El número de personas para completar el desarrollo en este tiempo:

$$N = \frac{E}{D} = \frac{23{,}5}{8{,}3} = 2{,}8 personas$$

Se pueden realizar algunos comentarios sobre los datos obtenidos:

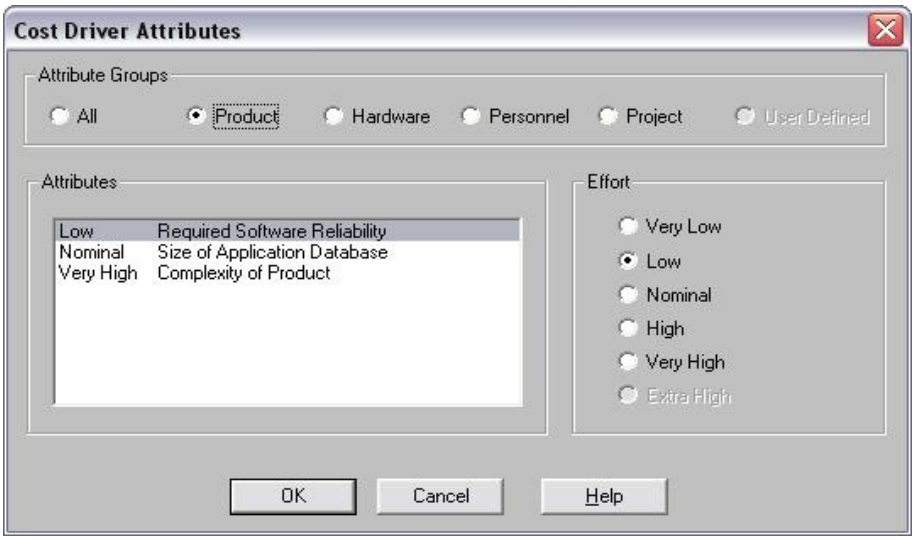
- La productividad por persona es de $\frac{6667}{23{,}5} = 283{,}7 LDC/persona - mes$
- La mínima duración del proyecto depende del esfuerzo en lugar del número de personas que trabajan en el proyecto. Esto recuerda con la experiencia que añadir personal a un proyecto que sobrepasa la planificación no ayuda a acabarlo a tiempo.
- El modelo no predice el resultado de trabajar un período largo de tiempo con menos personal de la cuenta.
- La estimación del esfuerzo incluye un valor promedio del tiempo de vacaciones, formación y faltas por fuerza mayor.
- Asume que los requisitos no cambian significativamente durante el desarrollo.

1.4.8. Modelo COCOMO intermedio

Este modelo es una extensión del modelo anterior donde se tiene en cuenta el entorno de desarrollo usando una serie de atributos del software que incrementan la precisión de la estimación. A continuación ofrezco una breve explicación de cómo he elegido los distintos valores disponibles de los 15 atributos guía:

1. Producto
    - Fiabilidad del software requerida: el fallo del software provoca inconvenientes menores, ya que no se trata de ninguna aplicación crítica de cuya funcionalidad dependan vidas humanas. Si ocurre algún tipo de error, puede se que se representen mal los polígonos en escena o que se construya erróneamente alguna parte del invernadero 3D, pero no posee una especial relevancia. Por eso le he asignado a este atributo el segundo valor más bajo.
    - Tamaño de la base de datos de la aplicación: en mi caso no tengo una base de datos, luego evito este atributo (le asigno el valor por defecto y así no influye en la decisión final).

- Complejidad del producto: este atributo sí va a ser decisivo en mi proyecto ya que tiene bastante complejidad a la hora de programar algunas funciones concretas (rotaciones de cámaras, movimiento a través del mundo 3D, detección de colisiones, etc). Por lo tanto le he asignado un valor elevado para que se tenga bastante en cuenta.



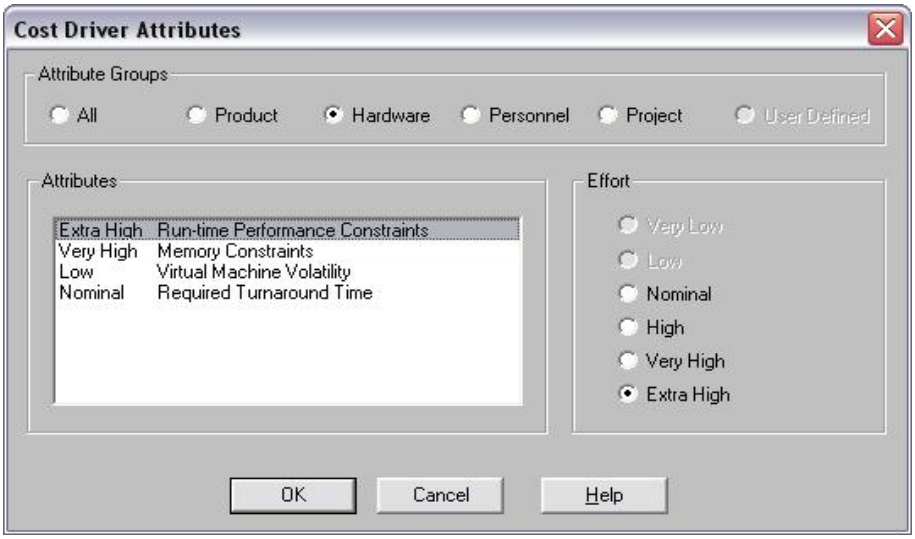
2. Hardware

- Restricciones de rendimiento en tiempo de ejecución: es deseable que la visita virtual sea lo más fluida posible, luego este atributo tiene mucha importancia. En este aspecto no sólo tendré que centrarme en el engine 3D para que éste me ofrezca un gran rendimiento, sino que procuraré que los elementos básicos que constituyen el invernadero tengan el menor número de polígonos posible.

- Restricciones de memoria: se intentará que el programa ocupe la menor cantidad de memoria posible en funcionamiento. Para ello barajaré diferentes opciones y me quedaré con la más óptima.

- Volatilidad del entorno de la máquina virtual: le he asignado el valor más bajo, ya que cambiará muy pocas veces al año.

- Tiempo de vuelta requerido: como va a ser un sistema interactivo, le he asignado un valor normal.



3. Personal

- Capacidad de análisis: le he asignado un valor alto, el esperado de una persona que está a punto de acabar una ingeniería.
- Experiencia con las aplicaciones: no poseo demasiada experiencia con las tecnologías que voy a usar, así que he puesto un valor intermedio.
- Capacidad de programación: se da por supuesto que existe un gran dominio en el lenguaje de programación.
- Experiencia con la máquina virtual: un valor bajo.
- Experiencia con el lenguaje de programación: como voy a utilizar un lenguaje conocido (C++), no tendré demasiados problemas.

4. Proyecto

   - Aplicación de metodologías (ingeniería del software): he intentado aplicar la metodología en un aspecto muy riguroso, luego le asigno el máximo valor.
   - Utilización de herramientas de software: le he asignado un valor intermedio para que no influya en el factor de ajuste del esfuerzo.
   - Planificación temporal del desarrollo requerida: también le hemos asignado un valor intermedio para que tampoco influya en el factor de ajuste del esfuerzo.

Los resultados obtenidos los muestro a continuación:

Title:	**Diseño y visita virtual de invernaderos 3D**
Prepared By:	**Moisés**
Description:	**Proyecto fin de carrera**
Source Lines of Code:	**6667.0**
Nominal Effort:	**23.5 person months**
Adjusted Effort:	**20.0 person months**
Time to Develop:	**7.8 calendar months**

El valor del esfuerzo y el tiempo necesarios han variado levemente con respecto al modelo básico de COCOMO, debido a que el intermedio es más preciso y permite ajustar mejor nuestro entorno real. Con el básico habíamos sobreestimado el tiempo necesario, así como los recursos humanos que hacen falta para el desarrollo. Ahora el número de personas para completar el desarrollo en este tiempo serán:

$$N = \frac{E}{D} = \frac{20}{7{,}8} = 2{,}5\text{personas (frente a 2.8 con COCOMO básico)}$$

1.4.9. Modelo Putnam

Este modelo es un modelo de estimación teórico. Se trata de un modelo multivariable dinámico que asume una distribución específica del esfuerzo a lo largo de la vida de un proyecto de desarrollo de software.

Ofrece una ecuación del software que relaciona el número de líneas de código esperadas con el es fuerzo y el tiempo de desarrollo:

$$K = \frac{L^3}{C_k^3 * t_d^4}$$

donde:

- K es el esfuerzo de desarrollo (ofrecido en personas/año)
- L es el número de líneas de código (calculado en apartados anteriores)
- Ck es la constante de la tecnología, que refleja las restricciones intrínsecas que frenan el progreso del programador. Se obtiene de datos históricos sobre esfuerzos de desarrollo (yo al no tener datos históricos, estimaré este valor para mi proyecto).
- Td es el tiempo de desarrollo en años (lo obtengo de modelos anteriores, como por ejemplo COCOMO intermedio, porque ya he calculado el tiempo estimado necesario para realizar el proyecto).

Nota: debido a que aparece con potencia 4 inversamente proporcional al tiempo de desarrollo, se deduce que un ligero incremento en la fecha de finalización del proyecto supone un ahorro sustancial en el esfuerzo humano de desarrollo.

En el caso de mi proyecto tengo los siguientes valores:

- L = 10.666 LDC
- Ck =15.000 (entorno muy bueno de desarrollo de software)
- Td = 0,66 años (8 meses del COCOMO intermedio)

$$K = \frac{6667^3}{15000k^3 * 0{,}66^4} = 0{,}5\text{personas-año}$$

El coste en personas-año sale de 0.5, esto significa que es necesario que una persona trabaje durante medio año (6 meses) para desarrollar el proyecto, valor más o menos lógico ya que no se trata de un proyecto de gran envergadura. Sin embargo como se puede ver existe una cierta diferencia con respecto a la estimación realizada con COCOMO, ya que como he comentado el modelo de Putnam supone un ahorro en el esfuerzo humano.

1.4.10. Planificación temporal

En esta última sección he realizado una planificación temporal, para poder establecer aproximadamente el tiempo de finalización del proyecto. Para ello he utilizado Microsoft Project y he realizado un diagrama de Gantt. En primer lugar, y usando los datos obtenidos en apartados anteriores sobre la duración del proyecto, he planificado una serie de tareas que se deben realizar, así como la duración de cada una y la interdependencia entra las mismas. He tomado el mínimo valor obtenido (fue de medio año para terminar el proyecto obtenido mediante Putnam) porque pretendo realizar el proyecto lo más rápido posible, y por lo tanto deseo ponerme fechas límite para comprobar si puedo cumplirlas.

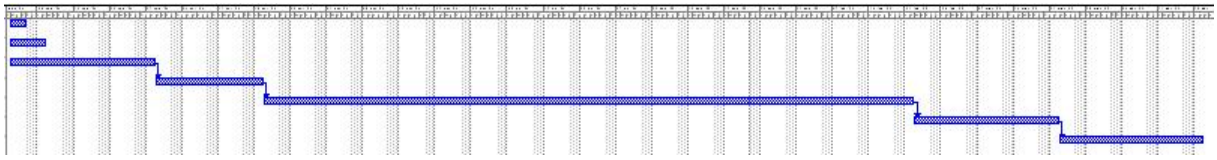
	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1	Entrevistas con directores de proyecto	3 días	mié 01/09/04	vie 03/09/04	
2	Búsqueda bibliográfica	5 días	mié 01/09/04	mar 07/09/04	
3	Aprendizaje de las tecnologías elegidas	20 días	mié 01/09/04	mar 28/09/04	
4	Diseño del proyecto	15 días	mié 29/09/04	mar 19/10/04	3
5	Codificación del proyecto	90 días	mié 20/10/04	mar 22/02/05	4
6	Prueba	20 días	mié 23/02/05	mar 22/03/05	5
7	Documentación	20 días	mié 23/03/05	mar 19/04/05	6

La primera tarea tiene una corta duración, y son las distintas entrevistas iniciales con los directores de mi proyecto para ir explicándole mis inquietudes. Además también tengo que realizar una búsqueda bibliográfica exhaustiva. Y a continuación deberé aprender a utilizar las tecnologías elegidas, en mi caso: Visual C++, OpenGL, SDL, Autocad, etc...

Una vez haya realizado dichas tareas, y sólo cuando haya aprendido a usar las tecnologías, comenzará la fase de diseño del proyecto, donde iré modelando las distintas partes y visualizando las distintas clases y métodos. A continuación llegará una de las fases más largas, la de codificación, que ocupará unos 3 meses aproximadamente. En teoría la fase de codificación de un proyecto no debe durar demasiado, sin embargo yo soy realista y estimo que al enfrentarme a OpenGL tendré bastantes problemas porque no es sencillo enfrentarse a la creación de un engine 3D desde cero.

Después llegará la fase de prueba, donde iré mejorando el proyecto y eliminando los posibles errores que vayan apareciendo. Y finalmente realizaré una documentación explicando en la medida de lo posible mi trabajo. Dicha documentación la presentaré en LaTeX, exportada en formato PDF.

El diagrama de Gantt que he obtenido es el siguiente:



He supuesto que comienzo a trabajar el día 1 de septiembre del año 2004 y la fecha de finalización se establece el día 23 de marzo del 2005. No obstante intentaré presentar el proyecto en febrero del 2005, por lo que deberé echar bastantes horas extra...

Capítulo 2

Metodología y primeros pasos

En este capítulo explico cuáles fueron mis primeros pasos una vez elegido OpenGL como tecnología de desarrollo y realizada la planificación del proyecto. Lo he dividido en cuatro secciones principales:

1. *Solución de problemas iniciales.* En esta primera sección explico cuáles fueron los motivos que me llevaron a elegir el formato ASC para la representación de los objetos 3D, así como la justificación matemática de la detección de colisiones y de las rotaciones de la cámara con el ratón.

2. *Reducción del número de polígonos del modelo 3D.* Seguidamente he comentado cómo era el modelo 3D básico inicial y las diferentes transformaciones por las que fue pasando con el objetivo de ir reduciendo polígonos en escena.

3. *Creación de los modelos 3D con Autocad.* A continuación documento mi trabajo con Autocad para crear las formas básicas por las que está compuesto el invernadero, así como el mapeado de texturas aplicado.

4. *Creación de las texturas.* Aquí comento algunos aspectos relacionados con las texturas que he creado para los modelos 3D.

5. *Integración de Visual C++ con OpenGL y SDL.* Y por último explico cómo se puede utilizar OpenGL y SDL con el entorno Visual C++ de Microsoft, detallando toda la configuración necesaria.

2.1. Solución de problemas iniciales

En esta sección describo algunos problemas con los que tuve que enfrentarme una vez elegido OpenGL como tecnología de desarrollo. Como comenté anteriormente al realizar un engine 3D completamente desde cero tenía que preocuparme de aspectos tales como:

1. Carga de objetos 3D desde disco (elección del formato).
2. Detección de colisiones de la cámara con las paredes del invernadero.
3. Rotación de la cámara con el ratón.
4. Movimiento de la cámara a través del mundo virtual.

El último aspecto (movimiento de la cámara) no es algo demasiado complejo como se verá en el capítulo 3, por lo que su explicación correspondiente la realizo más adelante. Aquí me centraré en los 3 primeros aspectos.

2.1.1. Formatos de archivos 3D

Existen en la actualidad una gran multitud de formatos de archivos para modelos de objetos tridimensionales: 3DS, MD2 (Quake2), MD3 (Quake3), OBJ, POV, ASE, ASC, DXF, DWG, VRML, etc... Los más utilizados son los siguientes:

VRML: acrónimo del inglés Virtual Reality Modeling Language. Es un formato de archivo normalizado que tiene como objetivo la representación de gráficos interactivos tridimensionales. Está diseñado particularmente para su empleo en la web. Consiste en un formato de fichero de texto en el que se especifican los vértices y las aristas de cada polígono tridimensional, además del color de su superficie. Es posible asociar direcciones web a los componentes gráficos así definidos, de manera que el usuario pueda acceder a una página web o a otro fichero VRML de Internet cada vez que pinche en el componente gráfico en cuestión.

DXF: el formato DXF (Drawing Interchange file) fue desarrollado para auxiliar la comunicación entre el Autocad y otros programas. Estos archivos pueden ser transformados para otros sistemas CAD u otros programas. Los archivos DXF son archivos de texto en formato ASCII. La organización de un archivo DXF tiene el siguiente formato:

1. 'Header': informaciones generales sobre el archivo.
2. Sección de tablas: presenta la información en forma de tablas del tipo de líneas, texto, coordenadas, planos de información, o sea, formas de visualización.

3. Sección de bloques: presenta la definición de grupos de entidades.

4. Sección de entidades: sección que muestra los datos propiamente dichos, a través del uso de puntos, líneas, círculos, arcos, etc...

5. Sección de final de archivo: presenta solamente las informaciones de secciones de entidades y de final de archivo. La sección de entidades solo muestra referencia de los objetos seleccionados para salida.

ASC: Los ficheros **ASC** de 3D Studio fueron diseñados para facilitar el intercambio de modelos con otros modeladores y por ello son fáciles de comprender. En los ficheros ASC hay ciertas palabras concretas que se pueden emplear para identificar el comienzo de los datos:

1. 'Named object': marca el comienzo de los datos de un objeto.

2. 'Vertices': le sigue un valor con el número de vértices del objeto.

3. 'Faces': otro con el número de caras.

4. 'Vertex list': le sigue la lista de vértices, y para cada vértice están sus tres coordenadas de posición (tras 'X:', 'Y:' y 'Z:') y además las coordenadas de mapeado de texturas U y V.

5. 'Face list': le sigue la lista de caras del objeto, donde se usan las cadenas 'A:', 'B:' y 'C:' para determinar los índices de los 3 vértices del polígono triangular.

Ejemplo de archivo ASC

Ambient light color: Red=0.0 Green=0.0 Blue=0.0
Named object: 'Object'
Tri-mesh, Vertices: 8 Faces: 4
Vertex list:
Vertex 0: x: 0.000000 y: 450.000000 z: 5.000000 u: 0.000000 v: 0.000000
Vertex 1: x: 900.000000 y: 450.000000 z: 5.000000 u: 1.000000 v: 0.000000
Vertex 2: x: 900.000000 y: 550.000000 z: 5.000000 u: 1.000000 v: 1.000000
Vertex 3: x: 0.000000 y: 550.000000 z: 5.000000 u: 0.000000 v: 1.000000
Vertex 4: x: 0.000000 y: 450.000000 z: 250.000000 u: 0.000000 v: 0.000000
Vertex 5: x: 900.000000 y: 450.000000 z: 250.000000 u: 1.000000 v: 0.000000
Vertex 6: x: 900.000000 y: 550.000000 z: 250.000000 u: 1.000000 v: 1.000000
Vertex 7: x: 0.000000 y: 550.000000 z: 250.000000 u: 0.000000 v: 1.000000
Face list:
Face 0: A: 1 B: 2 C: 0

Face 1: A: 3 B: 0 C: 2
Face 2: A: 5 B: 6 C: 4
Face 3: A: 7 B: 4 C: 6

Además también estudié la especificación del archivo 3DS, bastante extensa por cierto, y hubiera sido algo complejo elaborar un cargador de objetos 3DS, así como de objetos DXF. Un dato curioso que me sorprendió al revisar los formatos 3D es el hecho de que, en casi todos por no decir todos, se usan polígonos triangulares siempre, es decir, todos los objetos tridimensionales están formados por un conjunto finito de triángulos que pueden formar polígonos mucho más complejos. Así, si por ejemplo estamos almacenando un simple plano rectángulo, éste se dividirá en dos triángulos, nunca en un polígono rectangular.

Tras revisar la especificación de estos y varios formatos 3D más (como por ejemplo el del formato VRML que difiere mucho al resto de formatos) me decanté finalmente por el formato **ASC**, ya que me ofrece una serie de ventajas sobre los otros que a continuación enumero:

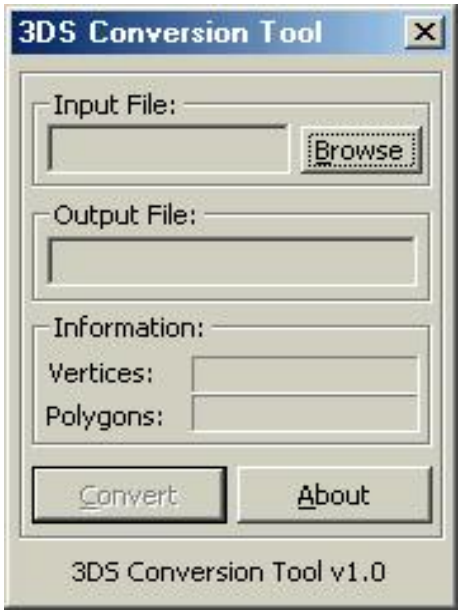
1. Es un formato muy sencillo y por lo tanto intuitivo a la hora de trabajar con él. Además tiene toda la información que necesito para mi proyecto: coordenadas de posición de los vértices, de mapeado de texturas, y polígonos triangulares con sus tres vértices cada uno. Es decir, ni me falta ni me sobra información para trabajar perfectamente. Es justo lo que necesito.

2. Además por el tamaño reducido de un archivo ASC aumento la eficiencia a la hora de cargar los objetos de disco, ya que hay otros formatos, como por ejemplo el 3DS y DXF, que son mucho más extensos, y ofrecen demasiada información irrelevante.

Pero como ya iba siendo bastante común desde que comencé este proyecto, el camino elegido no iba a ser tan fácil como parecía a priori... Como ya he comentado, **ASC** es un formato de intercambio de 3D Studio, y dado que yo iba a usar Autocad para la creación de los modelos y que, aunque en el departamento de Lenguajes y Computación tienen una licencia de 3D Studio pero por lo que pude comprobar no exportaba a **ASC**, tuve que buscar por Internet conversores de formatos de modelos tridimensionales. De tal modo que desde Autocad exporto el archivo en formato 3DS (explicado en la sección de Autocad de esta documentación) y a continuación convierto dicho archivo a ASC con un conversor gratuito que encontré por Internet en la siguiente dirección:

http://kurtm.flipcode.com/grain/downloads.htm

El nombre del conversor es: *3DS Conversion Tool*, y por supuesto es completamente *freeware*:



Con este sencillo conversor ya podía comenzar a trabajar en Autocad, creando los modelos con la precisión matemática que éste ofrece, y posteriormente exportarlos a 3DS y convertirlos a ASC para trabajar cómodamente con los modelos 3D.

2.1.2. Primeras ideas para detectar colisiones

Existen muchos métodos para implementar la detección de colisiones. Los más usuales consisten en rodear al objeto en cuestión con el que se desea detectar las colisiones (normalmente la cámara, es decir, la posición del usuario a través del mundo virtual) de una esfera, cilindro, cubo, etc... o cualquier otra forma tridimensional y comprobar si dicho sólido colisiona con los elementos deseados.

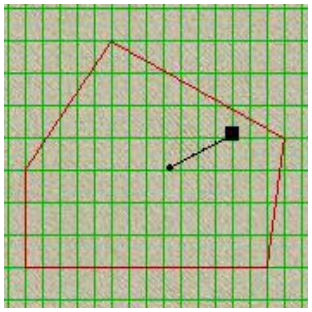
A veces no es deseable que el usuario choque con todos los objetos de la escena, como ocurre por ejemplo en mi caso. Yo sólo voy a aplicar la detección de colisiones a las paredes externas del invernadero, ya que si la aplicara también a otros elementos, como por ejemplo los pilares internos, se entorpecería demasiado la visita virtual porque el usuario se chocaría muy a menudo. Por lo tanto en mi engine 3D la cámara atravesará cualquier elemento interno del invernadero como si fuera un fantasma, y se chocará contra las paredes.

Tal y como he programado mi aplicación no sería complicado aplicarlo a todos los objetos de la escena. Sin embargo no lo haré ya que esto ralentizaría demasiado la visita virtual, y uno de los objetivos que he seguido desde el principio es intentar conseguir una animación lo más fluida posible.

Como en mi caso sólo se iba a realizar la detección con las paredes, y como dichas paredes se encuentran almacenadas como líneas entre vértices de la cuadrícula inicial, en un principio pensé en reducir mucho el problema simplificándolo a una detección de colisiones en 2 dimensiones. Los pasos que seguí fueron los siguientes:

1. En primer lugar, como en la interfaz del comienzo del programa guardo todos los vértices de los extremos de las paredes, hubiera trabajado con dichos vértices para trazar líneas entre ellos (cada línea sería una pared). Esas líneas las hubiera almacenado en memoria dinámica (usando el algoritmo de trazado de líneas de Bresenham), guardando todos y cada uno de los puntos de las líneas. Este paso se tendría que realizar una sola vez, al comienzo de la visita virtual.

2. Después sólo tendría en cuenta dos coordenadas de la posición de la cámara, así como también dos coordenadas del vector de vista de la cámara (trabajar en 2D). ¿Por qué iba a usar las coordenadas de dichos puntos? Muy sencillo, si la cámara está mirando (vector vista) hacia una pared y la posición (vector posición) fuera muy próxima a dicha pared, se produciría una colisión. Pero para ello, repito, es necesario que la cámara esté mirando hacia la pared. Porque si mira hacia otro punto, puede avanzar perfectamente ya que hacia ese lugar no hay pared. Por lo tanto, la idea consistía básicamente en rodear la posición de la cámara de un círculo (trabajaba en 2D) cuyo radio fuera la distancia entre el vector posición y el vector vista de la misma. De este modo trazaría una línea entre ambos puntos también con Bresenham, y comprobaría punto a punto si se produciría alguna coincidencia con los calculados anteriormente. Este paso se tendría que realizar para cada uno de los frames de la animación.



Dibujo: el punto grueso es la posición de la cámara y el más pequeño es el punto hacia donde mira la cámara. Las líneas rojas las paredes del invernadero. En este momento concreto, el usuario sí hubiera podido avanzar. Sin embargo, si la vista hubiera estado dirigida hacia la derecha, la línea entre la posición y la vista hubiera atravesado una pared, y por lo tanto se

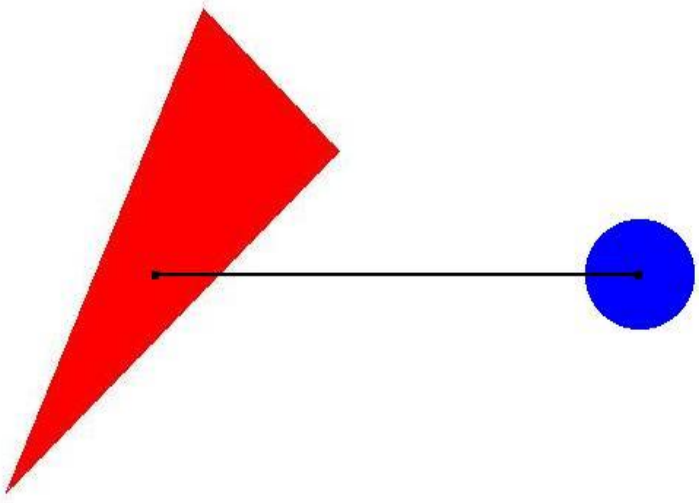
hubiera producido una colisión y no hubiera avanzado.

Sin embargo me encontré con una serie de inconvenientes como ya he adelantado:

1. El algoritmo suponía un gasto de tiempo enorme, sobre todo cuando el invernadero creado fuera grande, ya que las líneas creadas por Bresenham tendrían muchísimos puntos, y habría que comprobar la detección de colisiones para cada uno de ellos muchísimas veces, tantas como puntos creara Bresenham para la línea posición-vista.
2. Además en mi visita virtual no sólo puedo desplazarme hacia delante, sino también hacia atrás y hacia los lados, lo que complicaba enormemente el tratamiento de la detección de colisiones con este algoritmo en principio tan simple. Y ni que decir tiene cómo solucionar la detección de colisiones que se produciría cuando la cámara 'vuela', al trabajar sólo en 2D no hubiera tenido en cuenta la coordenada Y (altura) y hubiera colisionado aun estando a varios 'metros' de altura de la pared...

Por lo tanto, opté por buscar otro método alternativo, y finalmente no tuve más remedio que trabajar con las 3 dimensiones.

2.1.3. Colisión esfera-triángulo



Aunque existen muchos algoritmos de detección de colisiones, yo he elegido el algoritmo que detecta la colisión entre una esfera 'imaginaria' y un polígono, en mi caso un triángulo, ya que las paredes del invernadero, aunque son rectangulares, están formadas por dos triángulos cada una. Existen varias razones por la que he elegido este algoritmo:

1. Es ideal para comprobar si un personaje (vista en tercera persona) o una cámara (vista en primera persona, como es mi caso) colisiona con los elementos de un mundo. En mi visita virtual ningún objeto estará en movimiento excepto la cámara que se va desplazando por el mundo 3D, luego el punto que corresponde a la posición de la cámara será el centro de dicha esfera imaginaria (imaginaria porque no se ve, el usuario no sabrá que existe), y comprobaré si dicha esfera colisiona con alguno de los polígonos pertenecientes a las paredes del invernadero.

2. Además es uno de los algoritmos de detección de colisiones más rápidos que existen, ya que existen otros que usan por ejemplo un cilindro (como el algoritmo MDK2), elipses o cubos, y se obtiene un menor rendimiento a lo largo de la animación.

Por lo tanto este algoritmo, aunque sencillo, es muy eficiente. Los pasos que sigue son los siguientes:

1. **Colisión esfera-plano infinito**. En primer lugar se comprueba si la esfera colisiona con el plano del triángulo. Hay que tener en cuenta un detalle: aunque el polígono triangular es finito, el plano del mismo es infinito, luego en este apartado se va a realizar una detección de colisiones entre la esfera y el plano infinito del triángulo. Pueden obtenerse dos resultados posibles: que haya o que no haya colisión. Si la esfera no colisiona con dicho plano infinito, no lo hará con el polígono (que es una pequeña parte de dicho plano), luego el algoritmo finalizará ofreciendo el resultado de 'no colisión'. Si la esfera sí colisiona con dicho plano, entonces tendremos que ir al siguiente paso.

   Para comprobar dicha pseudocolisión hay que realizar una serie de pasos:

   a) En primer lugar obtengo un vector unitario perpendicular al plano del triángulo. Para ello obtengo dos vectores paralelos al plano del polígono. Cada vector lo obtengo restando dos vértices del triángulo, y teniendo en cuenta que ambos vectores deben tener su origen en el mismo vértice, la operación debe ser la siguiente:

   $$Vector1 = Vertice2 - Vertice1$$

   $$Vector2 = Vertice3 - Vertice1$$

   De este modo obtenemos dos vectores paralelos al plano y con mismo origen. Ahora para conseguir un vector perpendicular al plano, simplemente hago el producto vectorial de ambos vectores. Como estoy trabajando en cartesianas, se realiza de la siguiente manera:

   $$vNormal.x = Vector1.y * Vector2.z - Vector1.z * Vector2.y;$$

$$vNormal.y = Vector1.z * Vector2.x - Vector1.x * Vector2.z;$$

$$vNormal.z = Vector1.x * Vector2.y - Vector1.y * Vector2.x;$$

Por lo tanto ya tenemos un vector perpendicular al plano. Ahora sólo nos falta normalizarlo para hacerlo unitario. Para ello, se calcula el módulo del vector:

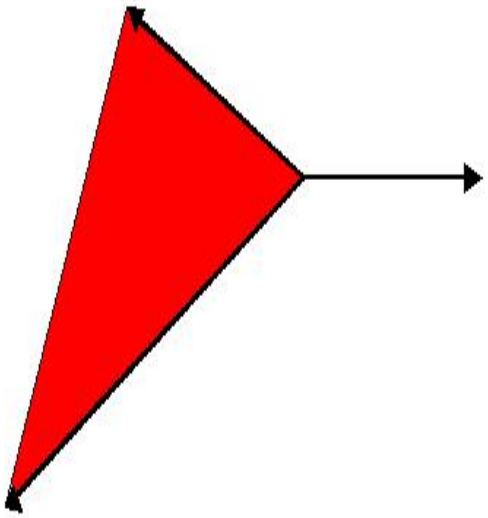
$$modulo = \sqrt{(vNormal.x * vNormal.x) + (vNormal.y * vNormal.y) + (vNormal.z * vNormal.z)}$$

Y a continuación se dividen todas sus coordenadas entre dicho módulo:

$$vNormal.x = vNormal.x/modulo$$

$$vNormal.y = vNormal.y/modulo$$

$$vNormal.z = vNormal.z/modulo$$



b) El próximo paso es encontrar la distancia del plano al origen, que se usará posteriormente en la ecuación para hallar la distancia entre el centro de la esfera y el plano:

$$distancia = Ax + By + Cz + D$$

Donde 'distancia', como su nombre indica, es la distancia desde el punto **(x,y,z)** al plano. Si el punto está sobre el plano 'distancia' es 0, si está detrás del plano es un número negativo y positivo si está en frente. Los valores **A, B y C** son las coordenadas **(x,y,z)** del vector normal al plano calculado anteriormente. Y el valor **D** es la distancia del origen al plano, y por tanto también deberemos calcularlo. Por lo tanto, para encontrar dicho valor **D** se hará lo siguiente:

$$-D = Ax + By + Cz - distancia$$

$$D = -(Ax + By + Cz) + distancia$$

Como sabemos que un punto que pertenezca al plano tiene como 'distancia' el valor 0, podemos sustituir por ejemplo un vértice del triángulo (que pertenece al plano) y por lo tanto eliminamos esa incógnita. Luego quedaría la fórmula así:

$$D = -(Ax + By + Cz)$$

Siendo **A, B y C** las coordenadas **(x,y,z)** del vector normal al plano y **x,y,z** las coordenadas de un vértice del triángulo:

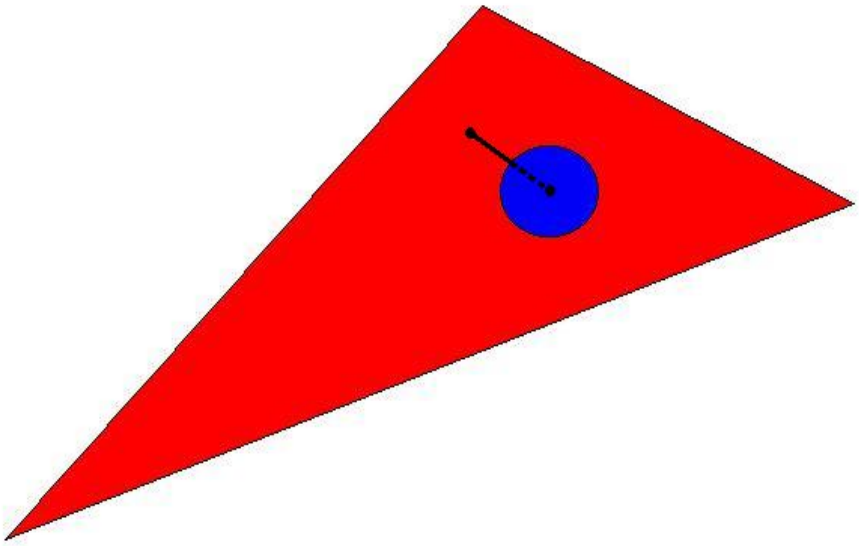
$$D = -(vNormal.x * Vertice1.x + vNormal.y * Vertice1.y + vNormal.z * Vertice1.z)$$

c) Con esto ya hemos obtenido el valor de **D**, y ahora sí que podemos sustituir las coordenadas del centro de nuestra 'esfera imaginaria' en la ecuación del plano para obtener la distancia entre ambos.

$$distancia = Ax + By + Cz + D =$$
$$vNormal.x * vCentro.x + vNormal.y * vCentro.y + vNormal.z * vCentro.z + D$$

Si el valor absoluto de esta distancia es menor que el radio que le hemos asignado a la esfera, entonces hay colisión con el plano del triángulo. Si no, finaliza el algoritmo.

```
if (abs(distancia)<=RADIO)
        colisionPlano=true;
else
        colisionPlano=false;
```



2. **Punto de proyección esfera-plano infinito**. Una vez que sabemos que la esfera colisiona con el plano falta saber si colisiona con el triángulo en sí. Para ello el siguiente paso consiste en obtener el punto de intersección de la esfera con el plano infinito (el siguiente paso será comprobar si dicho punto de intersección está dentro del triángulo). Para ello vamos a necesitar tanto el vector normal al triángulo calculado anteriormente como la distancia que hay desde el centro de la esfera al plano infinito. Y simplemente debemos multiplicar el vector normal por dicha distancia, para obtener así un desplazamiento (offset). El siguiente paso es restar este desplazamiento al centro de la esfera, y así hemos conseguido obtener la posición del centro de la esfera dentro del plano infinito, en la dirección de su normal.

$$vOffset = vNormal * distancia;$$

$$vProyeccion = vCentro - vOffset;$$

3. **Colisión esfera-triángulo**. Este es el último paso de todos, donde realmente se decide si existe o no colisión. Para ello se usa un procedimiento muy básico que consiste en calcular 3 ángulos, los formados por los 3 pares de vectores que parten del punto de proyección obtenido en el paso anterior y con los otros vértices. Los tres pares de vectores son los siguientes:

   Primer par:
   Vector1 = Vertice1 - PuntoProyección
   Vector2 = Vertice2 - PuntoProyección

   Segundo par:
   Vector3 = Vertice2 - PuntoProyección
   Vector4 = Vertice3 - PuntoProyección

   Tercer par:
   Vector5 = Vertice3 - PuntoProyección
   Vector6 = Vertice1 - PuntoProyección

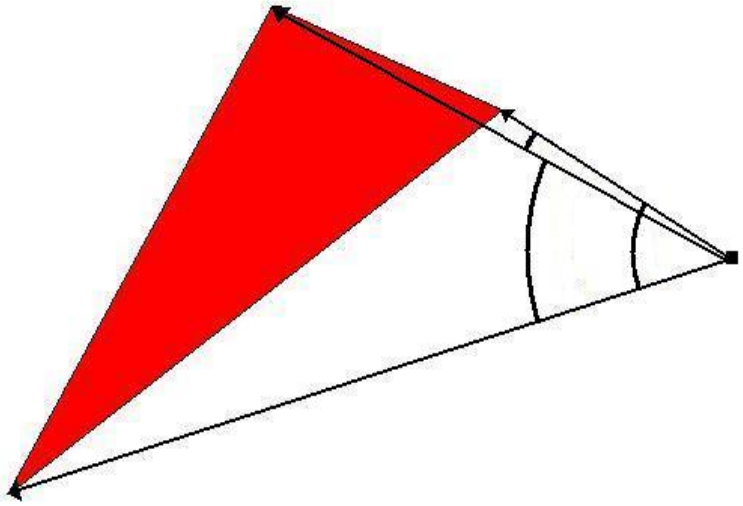
   A continuación se calcula el ángulo α formado por cada par de vectores. Para ello se usa la siguiente fórmula:

$$\cos\alpha = \frac{VectorX.VectorY}{|VectorX|.|VectorY|}$$

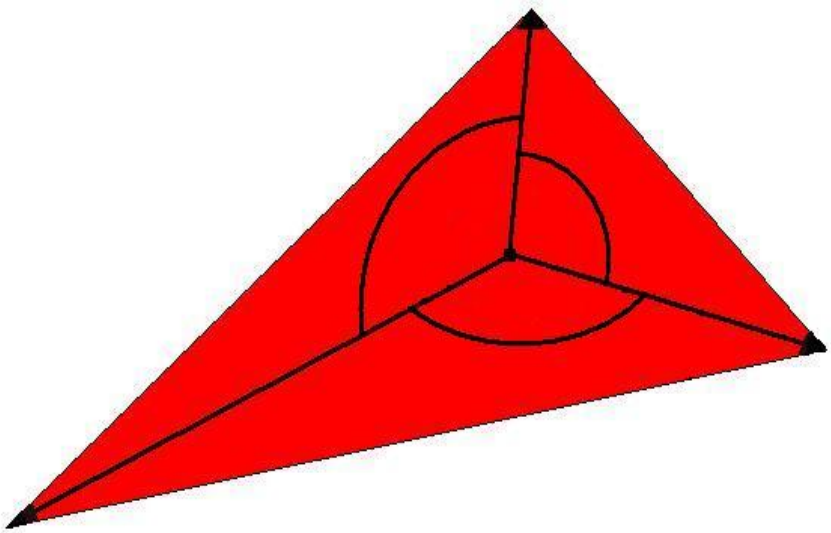
   Finalmente se suman todos los ángulos. Por pura matemática, si la suma de los 3 ángulos es mayor o igual a 360º (o lo que es lo mismo, 2*Π) , el punto de proyección anteriormente

calculado realmente pertenece al triángulo, y si es menor, dicho punto sólo intersecta con el plano infinito del triángulo pero no con el polígono en sí.

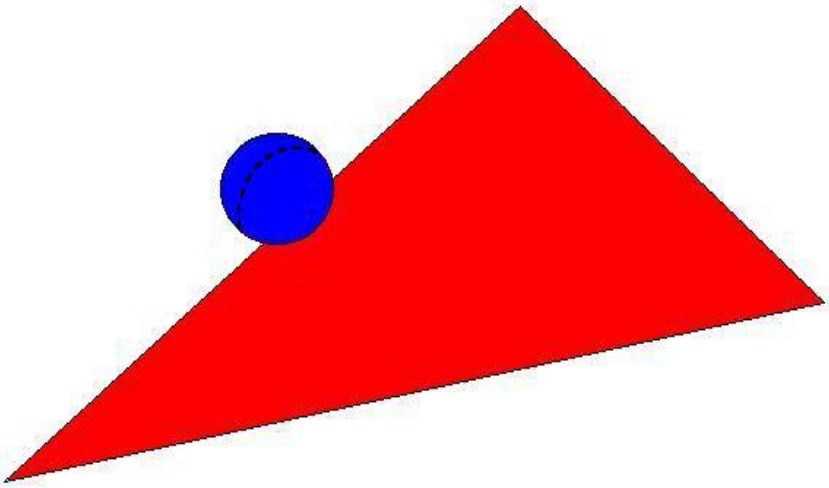
if (sumaAngulos>=2*PI) colisionTriangulo=true; else colisionTriangulo=false;



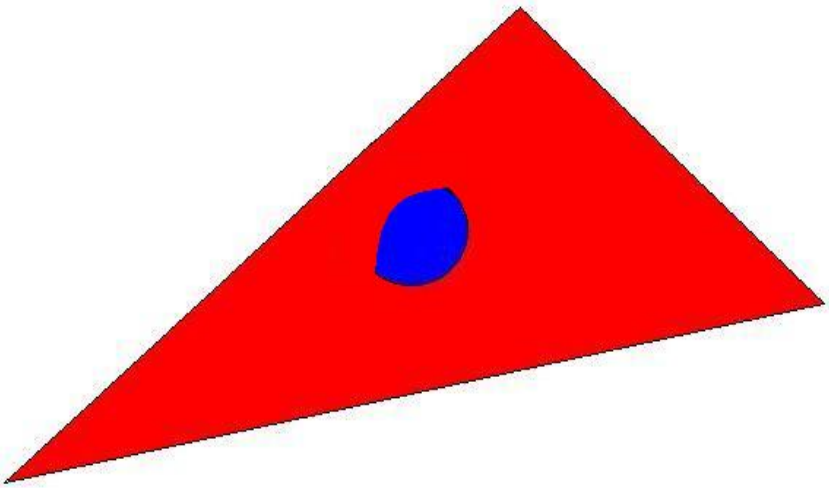
Ejemplo de punto de proyección fuera del triángulo. La suma de los ángulos es mucho menor que 360º



Ejemplo de punto de proyección dentro del triángulo. La suma de los ángulos es igual a 360º



Ejemplo de colisión sólo con el plano



Ejemplo de colisión con el triángulo

2.1.4. Rotaciones de la cámara - Rotación sobre un eje arbitrario

El fundamento matemático necesario para rotar la cámara con el movimiento del ratón es bastante más complejo. Aunque a priori simplemente se trata de realizar una sencilla rotación, la complejidad viene dada por el hecho de que es casi imposible que el eje de rotación que necesitamos coincida con alguno de los 3 ejes de coordenadas. Es más, también es muy difícil que

dicho eje de rotación sea paralelo a alguno de los ejes de coordenadas. Por lo tanto, la rotación en este caso se complica bastante.

Para poder rotar los objetos con respecto a un eje cualquiera es necesario realizar una transformación compuesta que va a implicar una combinación de traslaciones y rotaciones con respecto a los ejes de coordenadas. La rotación necesaria la vamos a llevar a cabo en los siguientes 5 pasos:

1. Se traslada objeto de tal modo que el eje de rotación pase a través del origen de las coordenadas.
2. Se rota el objeto para que el eje de rotación coincida con alguno de los ejes de coordenadas (en la explicación usaré el eje Z).
3. Se lleva a cabo la rotación que se desea.
4. Se realiza la rotación inversa a la del paso 2, de tal modo que se regresa el eje de rotación a su orientación original.
5. Se ejecuta la traslación inversa al paso 1, para llevar el eje de rotación a su posición inicial.

En lo que respecta al eje de rotación, lo voy a definir mediante 2 puntos, *P1* y *P2*, y por lo tanto:

$$V = P2 - P1 = (x_2 - x_1,\ \ y_2 - y_1,\ \ z_2 - z_1)$$

Vamos a definir un vector unitario u a lo largo del eje de rotación como:

$$U = \frac{V}{|V|} = (a, b, c)$$

Donde los componentes a, b y c del vector unitario u son los cosenos de dirección para el eje de rotación:

$$a = \frac{x_2 - x_1}{|V|} \quad b = \frac{y_2 - y_1}{|V|} \quad c = \frac{z_2 - z_1}{|V|}$$

Paso 1 - Traslación del objeto

Durante este primer paso, aplica al eje de rotación una matriz de traslación que cambia su posición, para que pase por el origen de las coordenadas. Dicha matriz de traslación es la siguiente:

$$T = \begin{pmatrix} 1 & 0 & 0 & -x_1 \\ 0 & 1 & 0 & -y_1 \\ 0 & 0 & 1 & -z_1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Paso 2 - Rotación hacia un eje

En este paso vamos a colocar el eje de rotación en el eje *Z*. Este paso es quizá el más complejo, no sólo porque requiere algunas nociones de trigonometría sino también porque se divide a su vez en otros dos pasos:

1. Se realiza una rotación con respecto al eje X para colocar el vector u en el plano xz.
2. Se rota alrededor del eje Y para que coincida con el eje de coordenadas *Z*.

Paso 2.1

Para conseguir los elementos de las dos matrices de rotación podemos usar las operaciones de vectores estándar, ya que dichas operaciones requieren funciones de seno y coseno.

La matriz de rotación con respecto al eje X la obtenemos al determinar los valores para el seno y coseno del ángulo de rotación necesario para situar u en el plano xz. Dicho ángulo de rotación es el mismo ángulo entre la proyección de u en el plano yz y el eje *Z*. Vamos a designar la proyección de u en el plano de yz como el vector *u'=(0, b, c)*, y entonces se puede obtener el coseno del ángulo de rotación α con el producto punto de u y el vector unitario u_z que está en el eje *Z*.

$$cos\alpha = \frac{u'.u_z}{|u'||u_z|} = \frac{c}{d}$$

donde d se puede calcular de la siguiente manera:

$$d = \sqrt{b^2 + c^2}$$

También se puede obtener el seno de α con el producto cruz de *u'* y u_z de la siguiente manera:

$$u' \times u_z = u_x|u'||u_z|sen\alpha$$

Y la forma cartesiana del producto cruz nos da:

$$u' \times u_= u_x.b$$

Al balancear el lado derecho de las ecuaciones anteriores y anotar que $u_z = 1$ y $|u'| = d$ tenemos que

$$dsen\alpha = b$$

Y

$$sen\alpha = \frac{b}{d}$$

Ahora, se puede establecer la matriz de rotación de u alrededor de las X, que gira el vector unitario con respecto al eje de las X en el plano xz.

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{c}{d} & \frac{-b}{d} & 0 \\ 0 & \frac{b}{d} & \frac{c}{d} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Paso 2.2

Seguidamente deberemos conseguir la matriz de rotación que gira el eje de rotación en el plano xz en sentido opuesto al de las manecillas del reloj, con respecto al eje Y. Este vector se llamará *u"*, y tiene como componentes *(a, 0, d)*. Ahora, para conseguir el coseno y seno de β:

$$cos\beta = \frac{u''.u_z}{|u''||u_z|} = d$$

$$u'' \times u_z = u_y.(-a)$$

encontramos que

$$sen\beta = -a$$

La matriz de rotación queda por tanto de la siguiente manera:

$$R_y(\beta) = \begin{pmatrix} d & 0 & -a & 0 \\ 0 & 1 & 0 & 0 \\ a & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Paso 3 - Rotación deseada

Una vez realizados todos estos pasos, es posible de una vez realizar la rotación con el ángulo inicial, con respecto al eje de coordenadas Z.

$$R_z(\theta) = \begin{pmatrix} cos\theta & -sen\theta & 0 & 0 \\ sen\theta & cos\theta & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Ahora necesitamos volver a situar el eje de rotación en su posición original. Para ello se deben realizar los pasos 2 y 1 de manera inversa.

Paso 4 - Inversa del paso 2

Este paso es el inverso del paso 2, y consiste en girar el eje de rotación de nuevo con respecto al eje Y y X, para que siga con su orientación original. Por lo tanto deberemos aplicar las matrices inversas del paso 2:

$$R_y(\beta) = \begin{pmatrix} d & 0 & a & 0 \\ 0 & 1 & 0 & 0 \\ -a & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{c}{d} & \frac{b}{d} & 0 \\ 0 & \frac{-b}{d} & \frac{c}{d} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Paso 5 - Inversa del paso 1

En este paso se va a devolver al eje de rotación al su posición original, y por lo tanto se debe aplicar la matriz de traslación del paso 1, pero invertida.

$$T = \begin{pmatrix} 1 & 0 & 0 & x_1 \\ 0 & 1 & 0 & y_1 \\ 0 & 0 & 1 & z_1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Todos estos pasos se pueden agrupar en una sola matriz, que sea resultado de la composición de todas las anteriores de la siguiente manera:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = T^{-1}R_x^{-1}(\alpha)R_y^{-1}(\beta)R_z(\theta)R_y(\beta)R_x(\alpha)T \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

2.2. Reducción del número de polígonos del modelo 3D

Elegí Autocad para la realización de los modelos tridimensionales básicos porque se ha convertido en la herramienta de diseño por excelencia para crear estructuras arquitectónicas. Además, el departamento de Lenguajes y Computación posee licencias de Autocad, luego puedo usarlo para el proyecto.

En la siguiente sección explicaré cuál fue mi trabajo con esta herramienta. En esta sección me voy a centrar en los pasos que seguí para reducir el número de polígonos del invernadero. Para ello hay que tener en cuenta que, a la hora de crear el invernadero dinámicamente, lo concebí como una sucesión de pequeños módulos unidos entre sí.

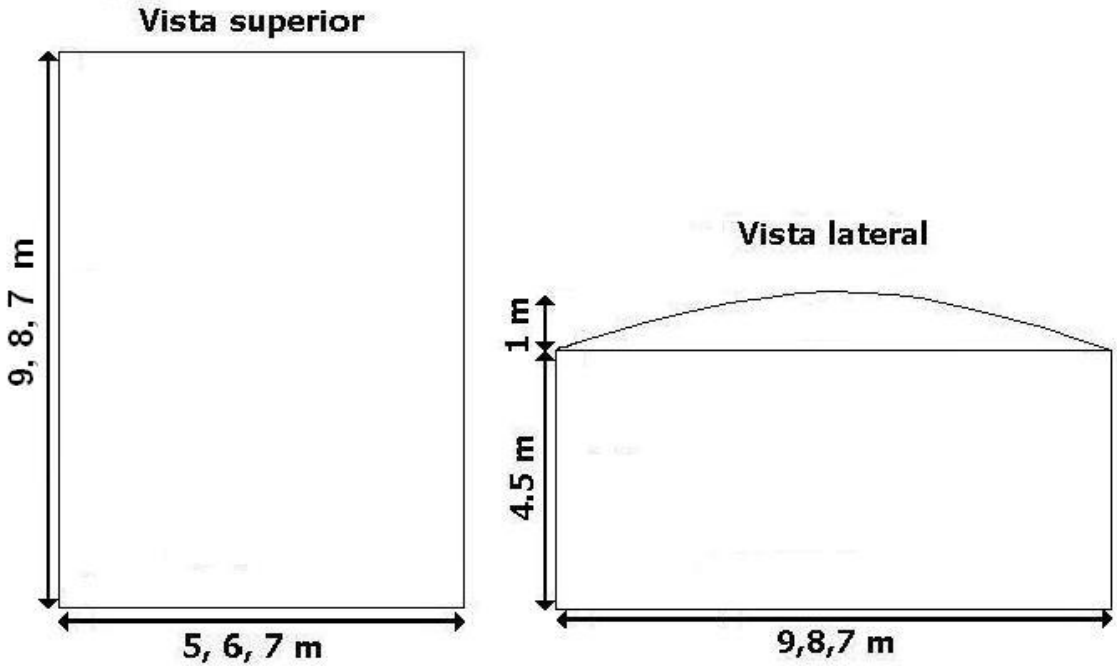
A partir de ahora llamaré 'rectángulo completo' a cada uno de los módulos pequeños por los que está formado un invernadero. También lo llamaré indistintamente 'módulo', y es el conjunto básico de objetos con los que voy a trabajar a la hora de construir el invernadero. Es decir, un 'rectángulo completo' se corresponde a una cuadrícula del plano de la interfaz inicial. Así, los iré colocando consecutivamente hasta formar toda la extensión del invernadero.

Posteriormente, ya con el programa, iré añadiendo otros elementos como por ejemplo las paredes del invernadero, el suelo e incluso algunos pilares o barras que falten en las superficies irregulares. Y además deformaré los módulos que sean necesario para crear formas irregulares de invernadero. Esta ha sido uno de los puntos más complejos de la elaboración del proyecto, como más tarde comentaré.

Es la mejor forma que se me ha ocurrido de hacer un invernadero dinámico irregular que se adapte a un polígono concreto. De este modo se pueden ir añadiendo módulos mientras sea necesario, incluso modificar la forma de algunos de ellos que formen parte de los extremos irregulares

del invernadero.

Las medidas de un módulo básico de invernadero son las siguientes:



Hay que tener en cuenta que estas medidas las he tomado de una empresa de construcción de invernaderos concreta. Cuando pregunté por las distancias entre pilares y alturas y medidas en varias empresas constructoras de invernaderos de la 'Expoagro' celebrada recientemente en Aguadulce, pude comprobar que cada una tenía unas medidas diferentes. Además, a la hora de crear superficies irregulares (paredes que no forman ángulos de 90º) no existen estándares.

Como ya he comentado anteriormente, aunque voy a construir los modelos básicos con Autocad, posteriormente los exportaré al formato nativo de 3D Studio y a continuación los convertiré a su formato ASCII, el formato ASC. En dicho formato, como en casi todos los que he estudiado antes de elegirlo, los objetos están formados por triángulos (como ya comenté). Por lo tanto un rectángulo estará formado por dos polígonos triangulares.

A la hora de construir los modelos siempre he tenido en cuenta sobre todo el detalle de no sobrecargar demasiado la escena con polígonos innecesarios. De este modo, los objetos básicos que he ido construyendo han ido evolucionando conforme he ido avanzando en el proyecto, de tal forma que los últimos que he creado tienen muchísimos menos polígonos, sin haber reducido el realismo de la escena.

Dicho módulo básico a su vez está formado por una serie de elementos:

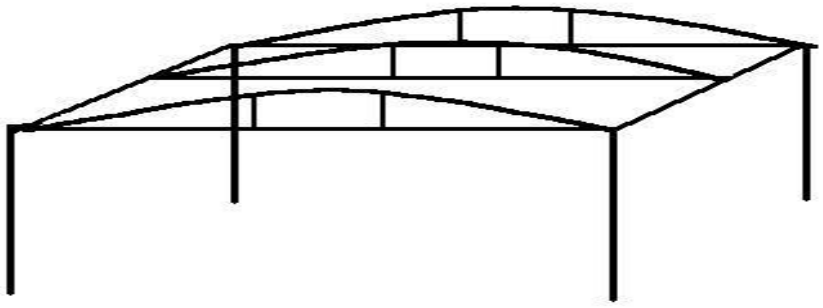
1. En primer lugar, cada módulo tiene 4 pilares que sostienen la estructura superior de medio arco. Cada pilar está formado por 4 caras, para que pueda contemplarse desde todas las perspectivas posibles.

2. Sobre dichos pilares descansan 5 barras horizontales, y también estará formada cada barra por 4 caras.

3. Además hay 3 barras con forma de arco, que descansan sobre las barras anteriores y sobre dos barras verticales cada uno. Cada una de estas 3 barras tiene en principio muchos polígonos, ya que son curvas. Era uno de los puntos que tenía que controlar, ya que 3 simples barras tenían casi el mismo número de polígonos que el resto del modelo.

4. Un elemento que no aparece en el dibujo anterior es la bóveda que cubre el invernadero. Haciendo cálculos, para que dicho semiarco tenga una forma semicircular, debería usar unos 7 polígonos rectangulares (hay que tener en cuenta que todos los polígonos son planos, luego si se quiere obtener una apariencia curva, hay que colocar varios polígonos planos consecutivamente inclinados).

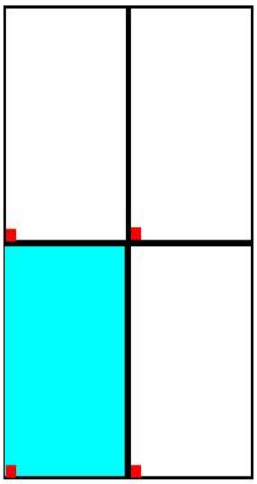
5. También hay que tener en cuenta el suelo. El suelo es un rectángulo con una textura de tierra.



2.2.1. Primer modelo tridimensional

Los primeros objetos básicos que cree fueron los siguientes:

1. En lugar de dibujar 4 pilares para cada módulo básico, sólo añadí 1. Como posteriormente en el propio programa se irían colocando los módulos de manera consecutiva, cada módulo estaría a su vez rodeado por otros módulos: si por ejemplo dejo sólo el pilar inferior izquierdo, cada módulo del invernadero tomará el inferior derecho del módulo situado a su derecha, y los superiores de otros dos módulos como se muestra en la siguiente imagen:



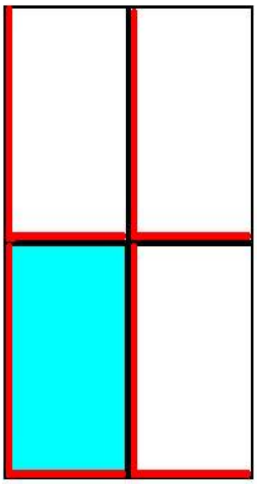
Luego simplemente, a través de mi propio programa, iría colocando los pilares que faltaban. Este ha sido uno de los detalles que por ejemplo no creo que hubiera podido implementar con Blender, Blitz3D ni por supuesto con VRML.

Con este simple detalle, había conseguido eliminar 3 pilares de cada módulo básico, cada pilar formado por 4 caras y cada cara rectangular como ya he comentado, formada por 2 polígonos triangulares. Es decir, había reducido el número de polígonos de cada módulo en 24 unidades.

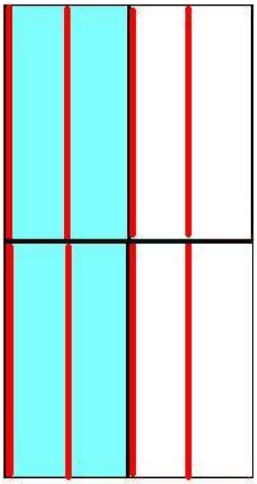
Aunque el número de polígonos reducido parece una cifra insignificante en un sólo módulo, no lo es cuando el invernadero es muy extenso, como veremos a continuación, ya que el número de polígonos aumenta de manera alarmante.

2. Además, también eliminé la cara superior de las barras horizontales, por el sencillo hecho de que no lo va a ver el usuario durante la visita virtual ya que desde su posición no se ve la parte superior de hierro.

3. Y reduje el número de barras horizontales de 4 a 2, al igual que el número de pilares, siguiendo la misma filosofía. De este modo cada uno de los módulos tendría por sí mismo la barra izquierda e inferior, y la derecha y superior la tomaría de los módulos consecutivos. Por lo tanto la mayoría de los módulos tendrían todas sus barras, y como en el caso de los pilares, por software se añadirían las barras que falten a la derecha y arriba de los módulos de los extremos:



4. Por último también eliminé una de las tres barras curvas, y gracias a la colocación consecutiva de los módulos del invernadero también quedarían completados:



Sin embargo el número de polígonos total que obtuve en esta primera ocasión fue realmente desalentador. Por unas horas me vi encerrado en otro callejón sin salida, como me había ocurrido antes con Blender. Con casi 250 polígonos triangulares por módulo, y haciendo una sencilla cuenta, teniendo un invernadero 8x8, es decir, 64 'rectángulos completos', me salían 16.000 polígonos en escena. Y eso teniendo en cuenta que ese invernadero es muy pequeño. Con un tamaño de invernadero un poco mayor la visita virtual directamente se hubiera hecho inviable. No hubiera alcanzado la fluidez que yo deseaba tener.

2.2.2. Modelo 3D definitivo

A continuación seguí reduciendo el número de polígonos de cada 'rectángulo completo', una y otra vez hasta un total de 4 veces. Finalmente obtuve un número inferior a 50, lo que me permitía visualizar invernaderos 5 veces más grandes que con el primer modelo creado. Para dicha reducción fueron decisivos:

1. Sustitución de las barras curvas por una textura semitransparente, que además aportaba más realismo a la escena con una imagen real. Para ello, en lugar de los casi 100 polígonos que ocupaban las 2 barras curvas (en otra optimización había eliminado la tercera, al igual que hice antes con los pilares), ahora ocupaban 4 polígonos las 2. Simplemente eran dos rectángulos, y luego en mi programa emplearía una textura de barras con fondos negros, para que sólo se dibujaran las barras y el resto (negro) fuera completamente transparente.

2. Además, pensé en sustituir el suelo de cada módulo por un enorme polígono rectangular que abarcara toda la extensión, y así no tendría tantos rectángulos como módulos, sino sólo uno enorme. Con esto había conseguido reducir el número de polígonos muchísimo, sobre todo cuando los invernaderos eran grandes. Luego simplemente tenía que aplicar las texturas teniendo en cuenta el tamaño total del rectángulo y el tamaño de la imagen de la textura.

3. Otro aspecto importante que también redujo considerablemente el número de polígonos fue una de las últimas decisiones que tomé: como había ocurrido con el suelo, en lugar de tener una bóveda semicircular para cada uno de los módulos, lo que hice fue transformar dicha estructura, alargándola, para que ocupara toda una fila completa de módulos, y así también evité muchísimos polígonos, sobre todo teniendo en cuenta que dicha estructura posee muchos polígonos por ser curvilínea.

4. Finalmente apliqué esa misma filosofía a otros elementos del módulo, de tal forma que tuve que crear métodos en C++ que permitirán hacer transformaciones en los objetos 3D para ajustar su tamaño y forma a los límites del invernadero. Gracias a esta idea el número de polígonos como ya he comentado, se redujo considerablemente. Posteriormente hablaré más de este tema.

5. Y como última nota, comentar que si el invernadero era gigantesco, seguía habiendo demasiados polígonos en escena. En este caso la solución me vino dada gracias a la API de OpenGL: especifiqué una distancia máxima de visualización respecto a la cámara, es decir, los objetos que estuvieran a mayor distancia de la determinada, no se dibujarían. También probé aplicando efectos de niebla, para así evitar aumentar el rendimiento.

   Esta técnica ha sido ampliamente utilizada por la industria de los videojuegos. Por ejemplo, la poco exitosa pero famosa Nintendo 64 utilizaba mucho esta técnica. Aunque para su

época tenía unos videojuegos de muchísima calidad (no olvidar que fue la primero consola en aceptar juegos totalmente en 3D, sin usar otras técnicas que lo simulaban como el llamado 'modo-7').

Sin embargo, como consecuencia de la enorme calidad de los gráficos y la gran extensión de los mundos virtuales, en casi todos los videojuegos de la N-64 se hacía un criticable uso de la niebla, de tal modo que el jugador sólo podía contemplar los objetos que se encontraban a un cierto radio de distancia. De este modo los objetos más alejados directamente no se dibujaban, y el engine 3D podía centrarse en mejorar la calidad de los que se encontraban más cercanos y así conseguían dos objetivos: aumentar la fluidez del videojuego y también la calidad de los gráficos.

No obstante esta técnica también fue ampliamente criticada, ya que a veces se hizo un uso abusivo de ella. Por ejemplo citar el 'Turok: Dinosaur Hunter', título de la N-64 que fue muy rechazado porque el jugador apenas podía ver a unos pocos 'metros' desde su posición, aunque eso sí, sus gráficos eran espectaculares.

2.3. Creación de los modelos 3D con Autocad

En primer lugar me documenté sobre el manejo de Autocad. Para ello consulté varios libros especializados en esta herramienta, y descargué algunos tutoriales de Internet. Creo que Autocad es más sencillo de manejar que Blender, por lo menos esa fue mi primera impresión, aunque no llegué a profundizar tanto en Autocad como para averiguarlo, ya que sólo tenía que crear unos objetos muy básicos.

Para la construcción de los objetos tridimensionales no he usado ninguna primitiva de Autocad para generar alguna forma determinada, ni ningún método de modificación de la malla de los sólidos (extrusión, giro, unión, etc.), ya que desde el principio tuve algunos problemas entre el mapeado de texturas de Autocad y el formato de archivo 3D que yo había elegido previamente. Este asunto lo explico en la siguiente sección.

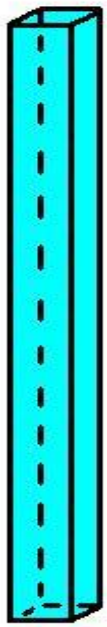
Por lo tanto sólo usé un comando en Autocad: *3dcara*. Este comando es muy sencillo de manejar: simplemente hay que ir introduciendo mediante teclado o ratón las coordenadas de cada uno de los vértices de la cara 3D. En mi caso los datos los ofrezco mediante teclado, para obtener una precisión perfecta. Se van introduciendo vértices hasta que el usuario lo desee, y cuando no se quieren más vértices el polígono se cierra.

Un último comentario respecto a la creación de objetos: todas las caras 3D que he creado en Autocad tienen forma rectangular, ya que por ejemplo el pilar del módulo está formado por 4 caras rectangulares. Sin embargo, dichos rectángulos serán transformados a 2 polígonos triangulares (como ya he comentado) mediante la transformación del formato del archivo.

2.3.1. Descripción de los objetos 3D

Las medidas de los objetos finales que he introducido en Autocad las he tomado en centímetros, y son las siguientes:

Pilar:
Cara frontal: (0,0,0), (15,0,0), (15,450,0), (0,450,0)
Cara derecha: (15,0,0), (15,0,10), (15,450,10), (15,450,0)
Cara izquierda: (0,0,0), (0,0,10), (0,450,10), (0,450,0)
Cara trasera: (0,0,10), (15,0,10), (15,450,10), (0,450,10)

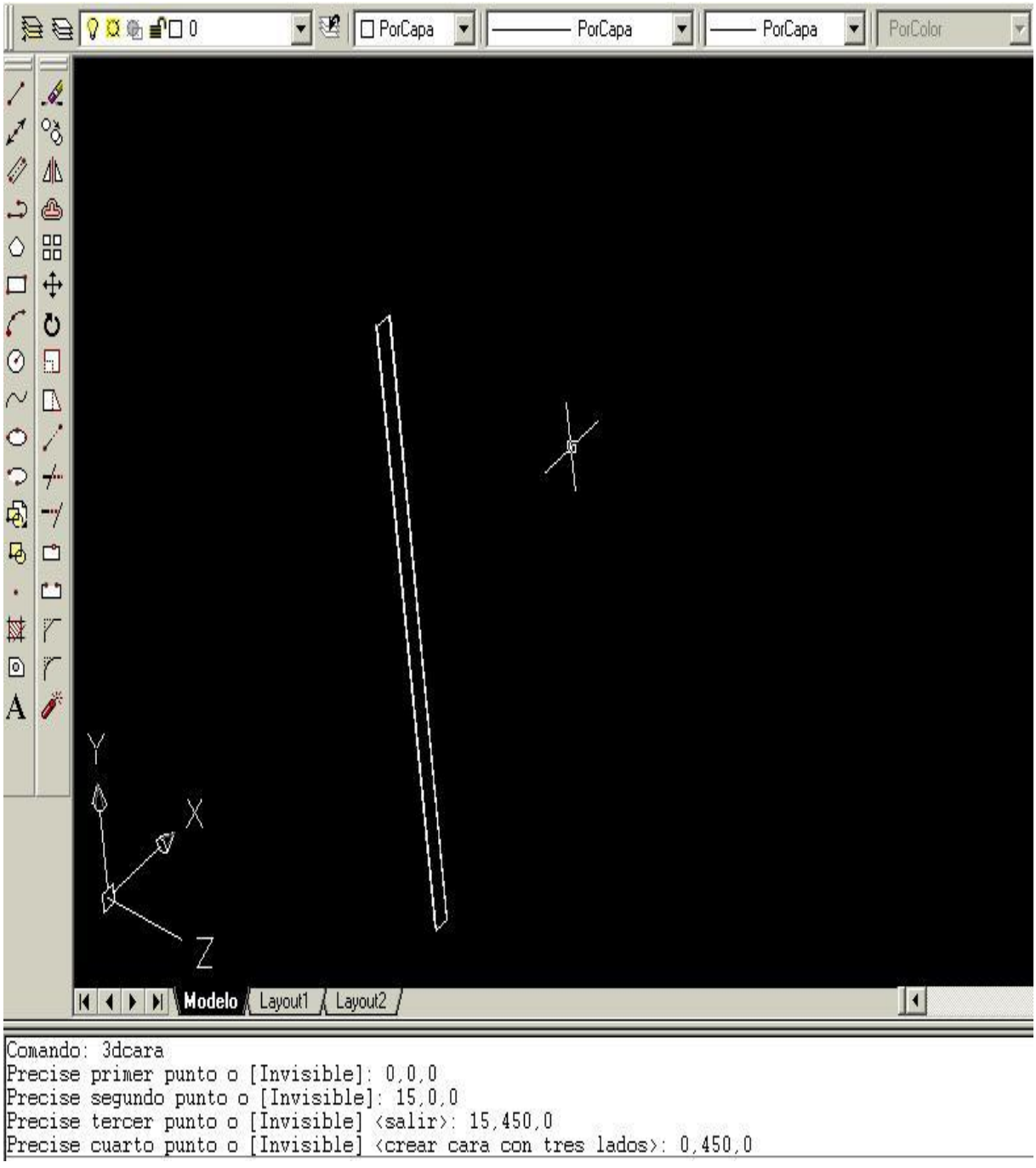


PILAR

Como se puede ver en el dibujo, el pilar está formado sólo por 4 caras. No dibujo las caras superior e inferior del pilar porque la inferior no se vería con el suelo y la superior con las barras horizontales. De este modo también ahorro polígonos, 4 en este caso, 2 por cada rectángulo.

Por lo tanto un pilar va a tener un total de 8 polígonos. Además la base del pilar es de 10x15 cm, y la altura es de 4.5 metros desde el suelo (hasta donde se encuentran las barras horizontales).

Para generar las caras del pilar como he comentado, he usado el comando *3dcara*, como muestro a continuación con la primera de las caras. Cuando he introducido los 4 vértices del rectángulo, simplemente pulso la tecla ESC para cerrar el polígono. Este mismo proceso habría que repetirlo para todas las caras:

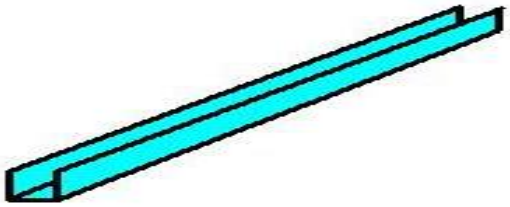


Primera barra horizontal:

Cara central: (15,450,10), (0,450,10), (0,450,500), (15,450,500)

Cara izquierda: (0,450,10), (0,455,10), (0,455,500), (0,450,500)

Cara derecha: (15,450,10), (15,455,10), (15,455,500), (15,450,500)



PRIMERA BARRA

Esta es la barra horizontal que mide 5 metros, que descansa sobre el pilar anterior junto con la otra barra horizontal. El tamaño podrá variar durante la visita virtual a través del teclado (usando la consola de texto).

Tan sólo he dibujado 3 de las cuatro caras que tendría la barra, entre otras cosas porque el usuario desde su posición poco elevada no podrá ver la cuarta cara que debería estar en la parte superior.

Un aspecto muy positivo que tiene el uso del comando *3dcara* es que permite especificar con total precisión las coordenadas de los vértices, y por lo tanto los objetos son invulnerables a rotaciones y a cambios de perspectivas durante la creación de los mismo.

Segunda barra horizontal:
Cara central: (15,450,10), (0,450,0), (900,450,0), (900,450,10)
Cara izquierda: (15,450,10), (15,455,10), (900,455,10), (900,450,10)
Cara derecha: (15,450,0), (15,455,0), (900,455,0), (900,450,0)

SEGUNDA BARRRA

Esta barra es la que mide 9 metros (posteriormente este tamaño también se podrá modificar dinámicamente durante la ejecución de la visita virtual). Tampoco dibujo la cara superior de la barra.

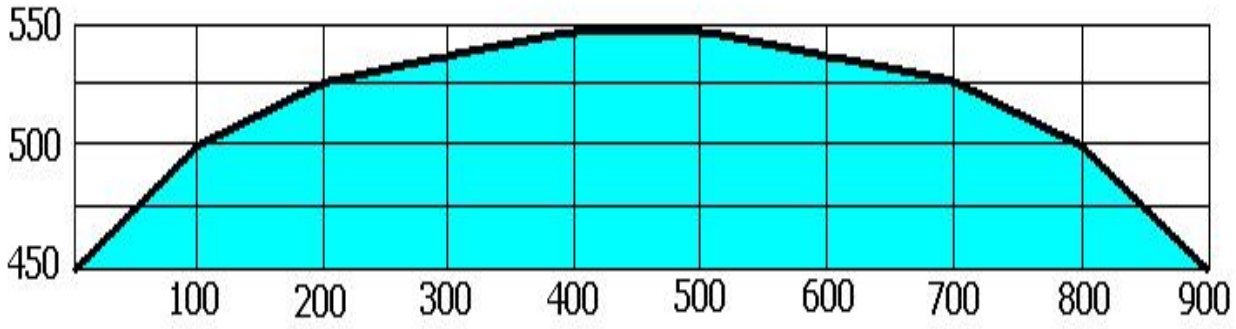
Primera barra curva:
Cara: (0,450,5), (0,550,5), (900,550,5), (900,450,5)

Segunda barra curva:
Cara: (0,450,250), (0,550,250), (900,550,250), (900,450,250)

A este par de rectángulos posteriormente les aplicaré una textura que represente unas barras curvas apoyadas en otras verticales. Gracias a esta idea he conseguido disminuir mucho el número de polígonos, ya que estas barras curvas eran las más complejas.

Bóveda:
Cara 1: (0,450,0), (100,500,0), (100,500,500), (0,450,500)
Cara 2: (100,500,0), (200,525,0), (200,525,500), (100,500,500)
Cara 3: (200,525,0), (300,538,0), (300,538,500), (200,525,500)
Cara 4: (300,538,0), (400,550,0), (400,550,500), (300,538,500)
Cara 5: (400,550,0), (500,550,0), (500,550,500), (400,550,500)
Cara 6: (500,550,0), (600,538,0), (600,538,500), (500,550,500)
Cara 7: (600,538,0), (700,525,0), (700,525,500), (600,538,500)
Cara 8: (700,525,0), (800,500,0), (800,500,500), (700,525,500)
Cara 9: (800,500,0), (900,450,0), (900,450,500), (800,500,500)

Las caras están contadas de izquierda a derecha. A continuación muestro una vista lateral de la bóveda, las unidades están medidas en centímetros y la coordenada Y comienza en 4.5 metros porque es en esa altura donde terminan las barras horizontales, donde se apoyan.



VISTA LATERAL DE LA BÓVEDA

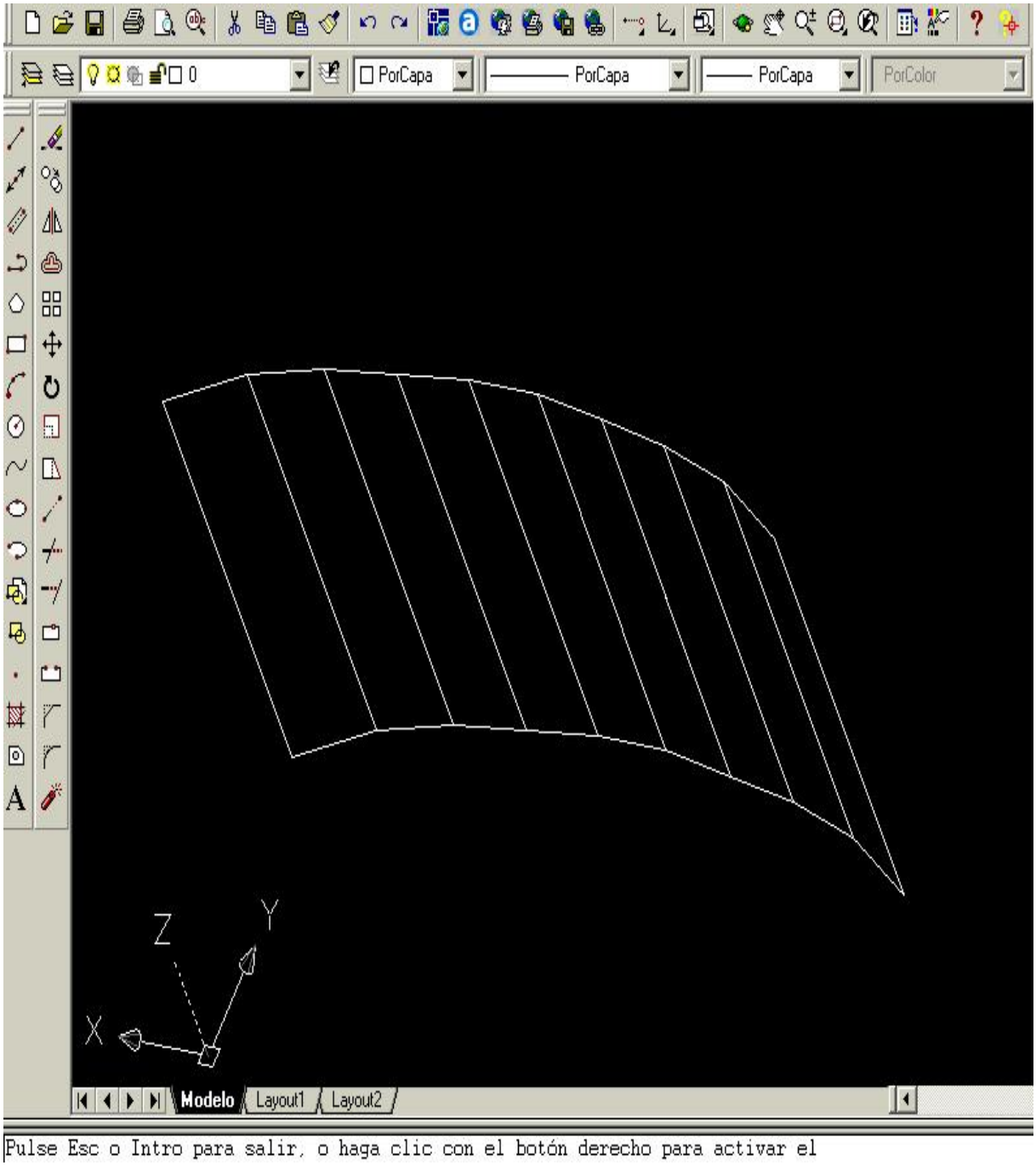
Resumen

Como se puede observar, el diseño de un módulo ha quedado muy simplificado, pero ello no implica una reducción en la calidad ni en el realismo de la escena, como se podrá comprobar. Lo único que he conseguido ha sido mejorar el rendimiento de la visita virtual al reducir enormemente el número de polígonos.

Este proceso de reducción del número de polígonos no ha sido inmediato ya que, como he ido explicando, el modelo inicial fue pasando por diferentes filtros hasta que acabó convenciéndome.

Era uno de los objetivos que me planteé desde el principio, y aunque se corre el riesgo de reducir el realismo de la visita, se deben usar las texturas adecuadas para que compensen la falta de polígonos.

El objeto más complejo es el de la bóveda, que a continuación muestro finalizado en Autocad:

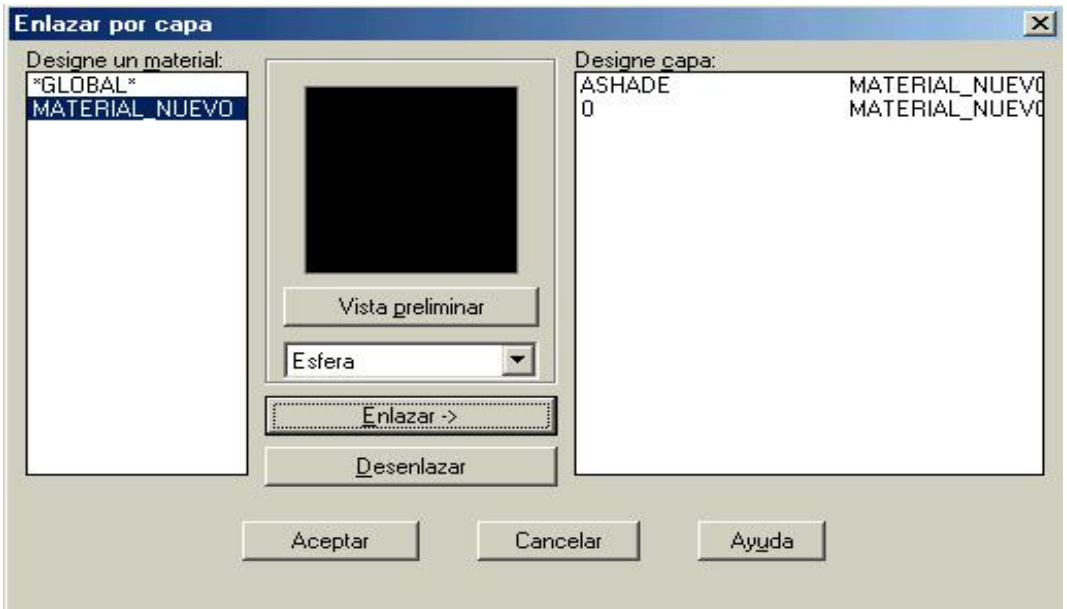


2.3.2. Aplicación de texturas con Autocad

Para aplicar texturas en Autocad, en primer lugar hay que crear un material nuevo (que es el que se va a aplicar). Para ello se elige la siguiente opción:

Ver ->Render ->Materiales ->Nuevo...

Se deben especificar las propiedades del material, así como la imagen que utilizará (en mi caso la textura en sí del polígono). Además poner un nombre al material para poder guardarlo en la biblioteca de materiales y usarlo posteriormente. A continuación, elegir la designación por capa del material, y asignarlo a las dos capas que aparecen en la parte de la derecha eligiéndolos y pulsando 'Enlazar':



Una vez se ha creado el material se debe asignar al polígono mediante el mapeado de texturas, eligiendo todos los polígonos de la escena con CTRL+E:

Ver ->Render ->Mapeado...

Como se puede observar en la imagen anterior, existen en Autocad 4 tipos de mapeados o proyección diferentes:

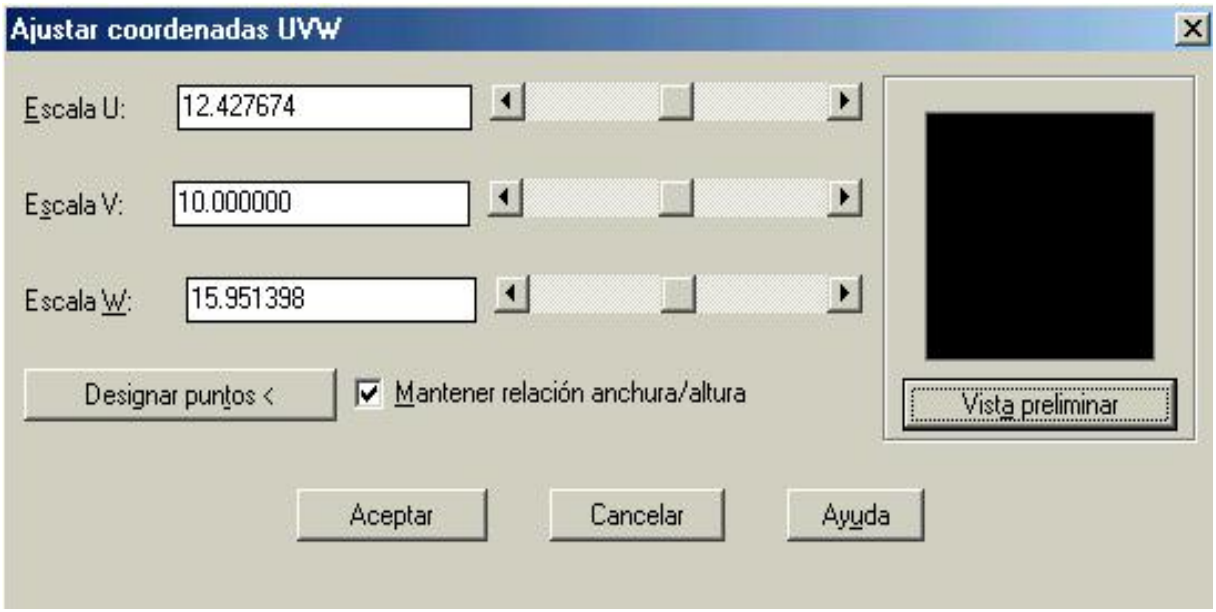
1. Plana: se asigna a objetos que están formados por un solo polígono, como su nombre mismo indica.

2. Cilíndrica: para aquellos objetos que poseen una forma parecida a un cilindro, esta proyección es la adecuada, ya que la textura va a envolver al objeto por todas sus caras laterales y por las dos caras restantes: superior e inferior.

3. Esférica: se emplea en los objetos con forma redondeada, de tal modo que la textura lo envuelve completamente.

4. Sólida: usada para el resto de los objetos. Con este mapeado se utilizan las llamadas coordenadas de mapeado U y V, que posteriormente son tan valiosas en OpenGL para mapear la textura. Estas coordenadas son parte de la información que ofrece el formato ASC de cada uno de los vértices del objeto tridimensional. Es por esta razón por que he elegido este tipo de mapeado, y no el mapeado plano que a primera vista podría ser el más lógico.

Para ajustar dichas coordenadas de mapeado hay que elegir la opción *Ajustar coordenadas...* y dentro del cuadro de diálogo siguiente la opción *Designar puntos <*:



Y a continuación, para dicho polígono rectangular, elegir las 4 esquinas del mismo en el siguiente orden:

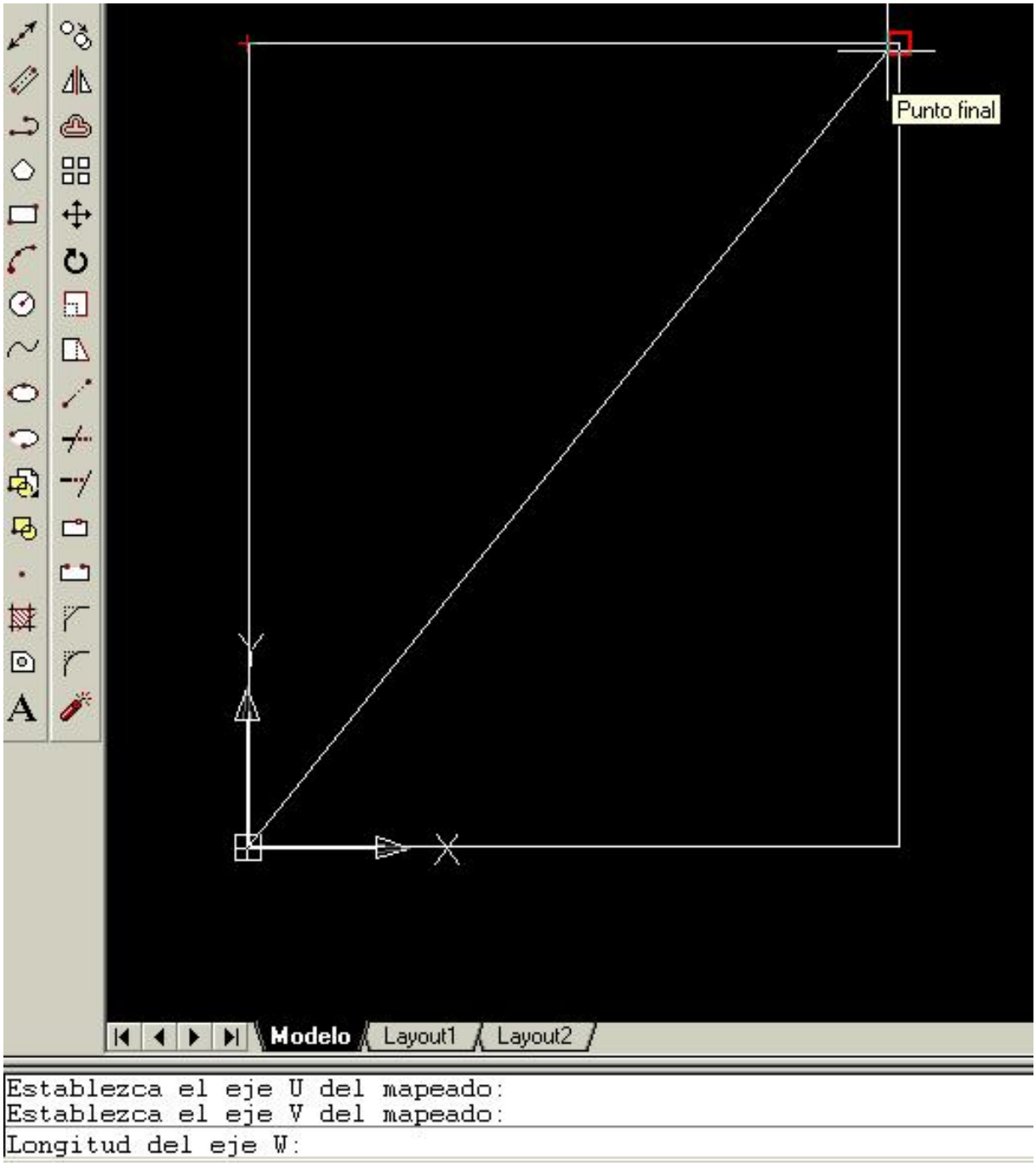
1. Primero la esquina inferior izquierda, que es el vértice desde donde comenzará a dibujarse la textura.

2. En segundo lugar la esquina inferior derecha, que será la coordenada U de la textura, es decir, la anchura de la imagen se adaptará a la distancia entre ambos vértices del objeto, alargándose o estrechándose.

3. Tercero elegir la esquina superior izquierda para la coordenada V, es decir, la altura de la imagen de la textura.

4. Por último elegir una nueva coordenada: la W, que marca la dirección de la textura. En este caso se elige el vértice que falta.

A continuación muestro el proceso de mapeado sólido, en el momento de elección de la coordenada W:



Finalmente, si se desea observar el objeto 3D con el material aplicado, elegir la siguiente opción:

Ver ->Render ->Render...

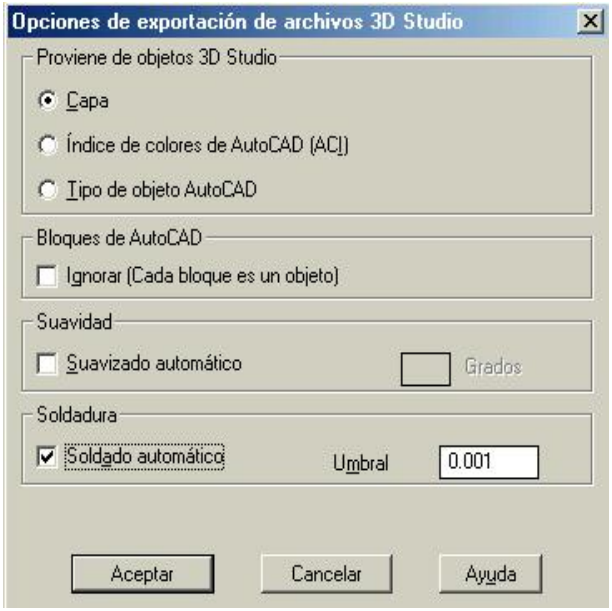
En la opción 'Tipo modelizado' elegir 'Fotorrealístico', para que también dibuje las texturas del polígono. Y finalmente pulsar el botón 'Modelizar'.

2.3.3. Exportación a 3DS y conversión a ASC

Una vez construí todos los objetos tridimensionales, los exporté al formato 3D Studio de la siguiente manera:

Archivo ->Exportar... ->3D Studio (.3DS)*

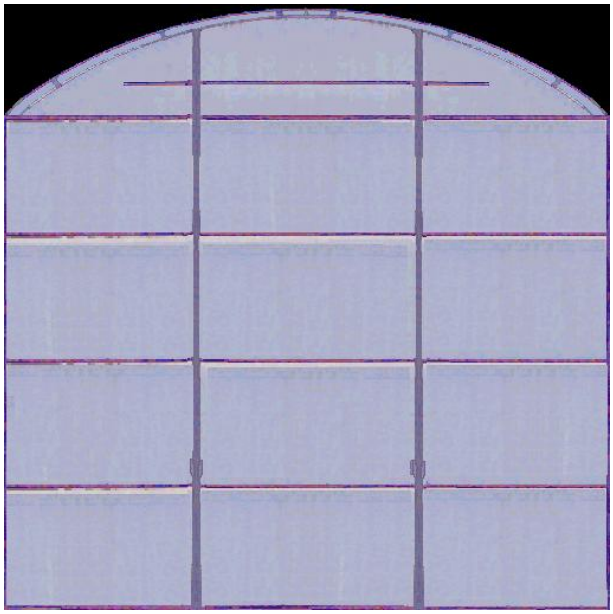
Para ello anteriormente había que seleccionar todos los vértices de los polígonos (Control + E). A continuación aparece el siguiente cuadro de diálogo donde se pueden modificar algunos parámetros de la exportación a 3DS:



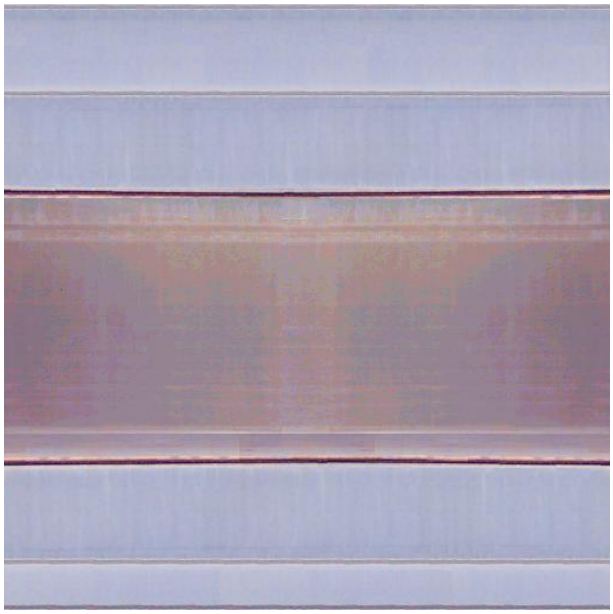
Y para finalizar, como ya comenté, utilizo el programa '3DS Conversion Tool', versión 1.0, que convierte el archivo de formato 3DS a ASC.

2.4. Creación de texturas

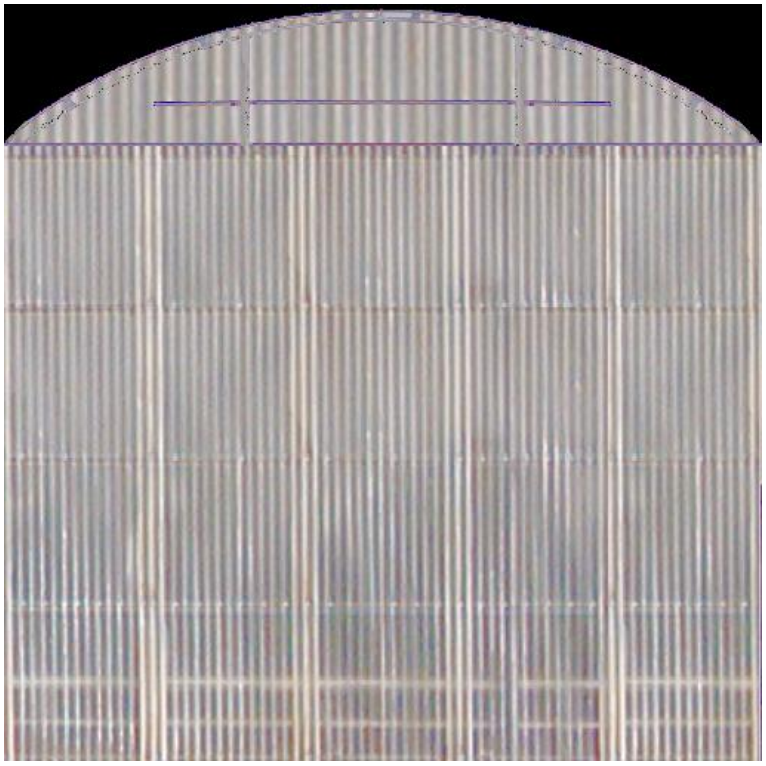
Las texturas que he utilizado para mis modelos son fotografías que he realizado sobre invernaderos reales, y que luego he retocado digitalmente con el programa de edición gráfica GIMP. He tenido que ir reduciendo la resolución de dichas texturas, ya que al principio tenían una demasiado grande y disminuía el rendimiento. A continuación muestro las más importantes, teniendo en cuenta que el color negro será transparente en mi engine 3D:



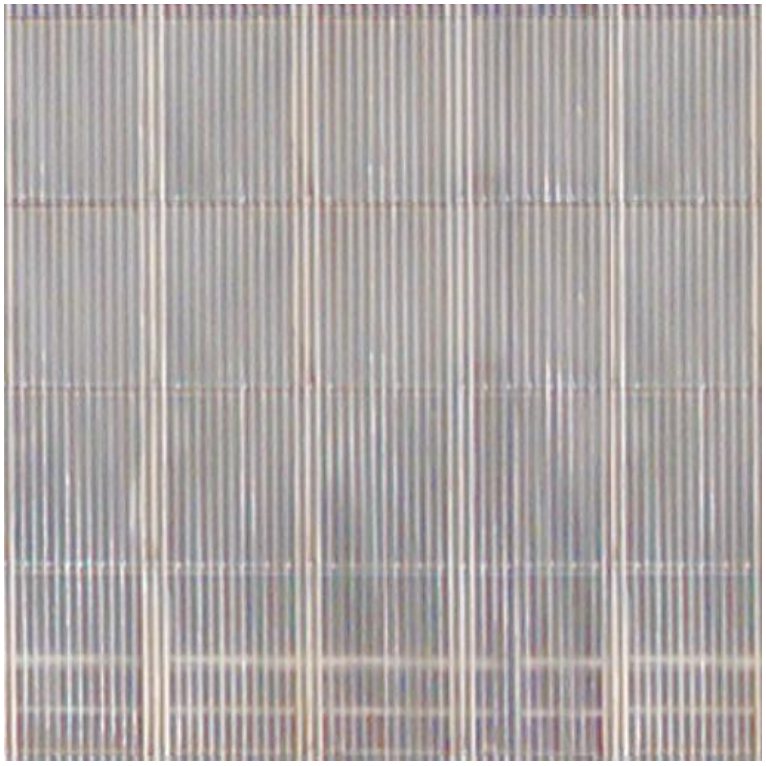
Frontal del invernadero tipo plástico



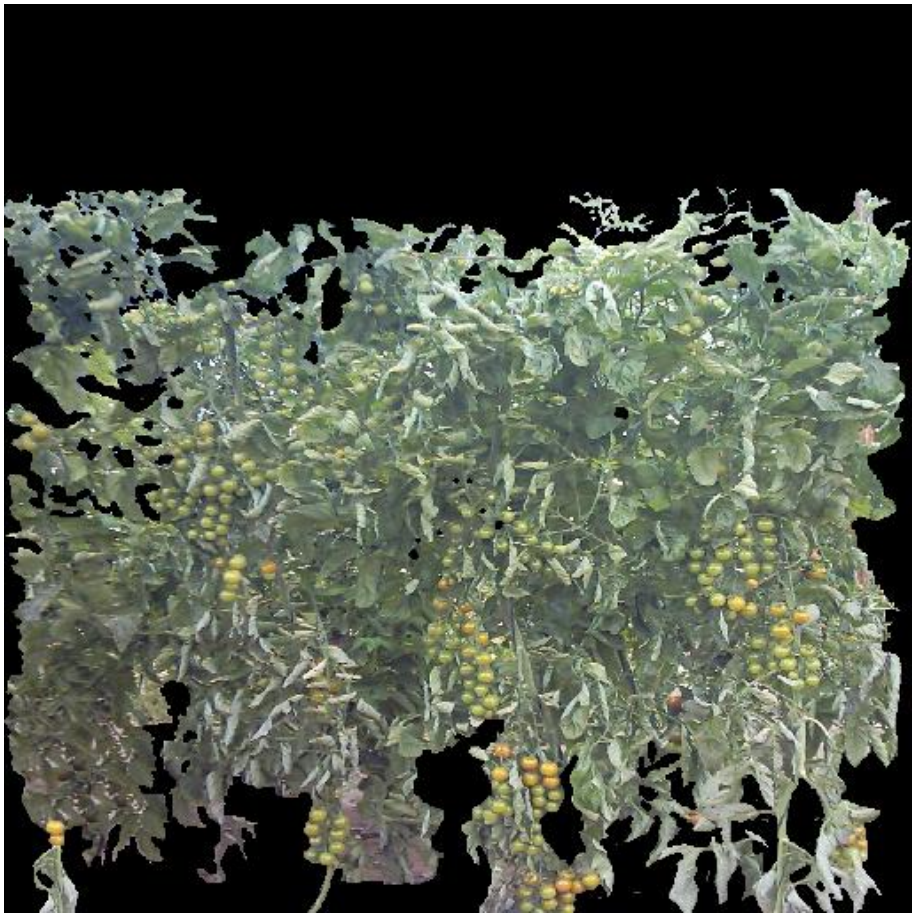
Lateral del invernadero tipo plástico



Frontal del invernadero tipo policarbonato



Lateral del invernadero tipo policarbonato



Textura de las plantas de tomate

Logotipo de la universidad de Almería

2.5. Integración de Visual C++ con OpenGL y SDL

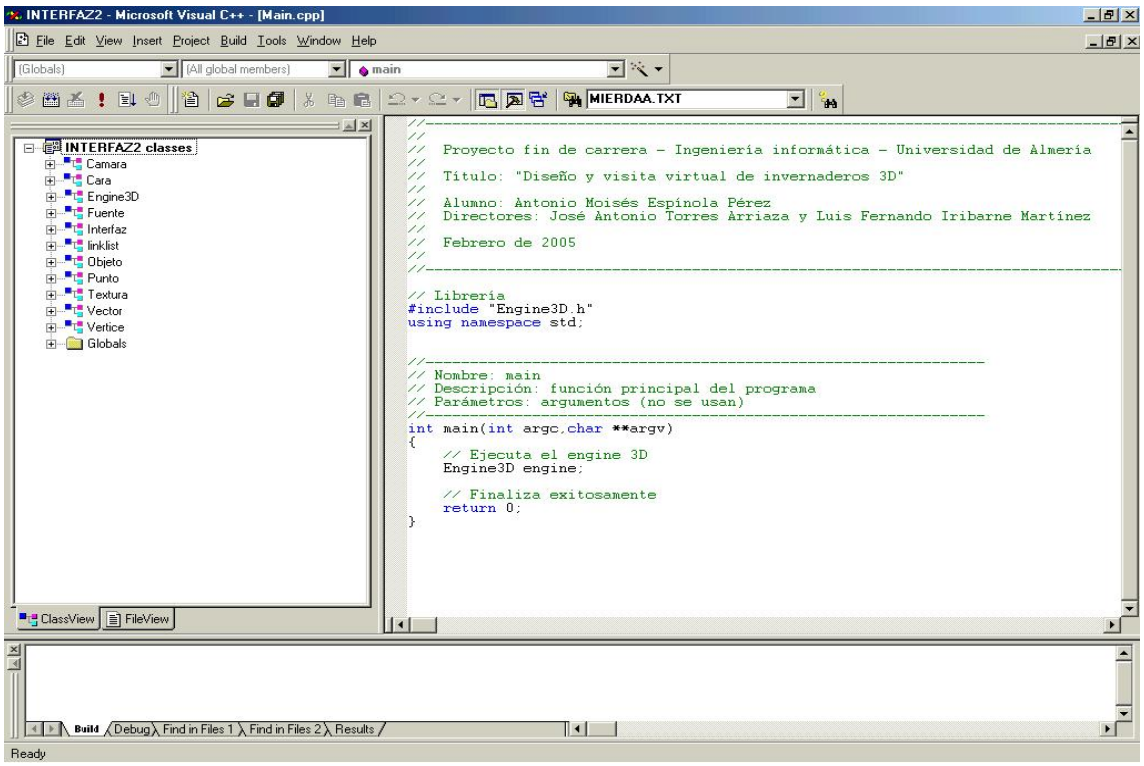
En este capítulo voy a explicar los aspectos que conciernen a la programación usando Visual C++ 6.0 con OpenGL y SDL, detalladamente explícitamente y paso a paso cómo hay que instalar las librerías necesarias para empezar a trabajar.

Todo esto es fundamental para ponerlo todo en funcionamiento y para poder compilar correctamente. En un capítulo posterior iré centrándome en todas y cada una de los clases de las que se compone mi código fuente, destacando no el código fuente sino la funcionalidad.

Aunque OpenGL puede utilizarse con otros muchos lenguajes de programación (C, C++, Visual Basic, Delphi, etc) yo he decidido utilizar quizá el lenguaje más estándar: C++, en mi caso Visual C++. No obstante, como OpenGL es una API multiplataforma, no sería demasiado complicado realizar una adaptación del código fuente para que pueda ejecutarse también en Linux. Por ello elegí la librería SDL como auxiliar y no otra.

2.5.1. Trabajando con Visual C++ 6.0 y OpenGL

En primer lugar es necesario instalar Visual Studio 6.0 que incluye Visual C++ junto con otros compiladores.



A continuación debemos añadir las librerías que hacen falta para programar con OpenGL. En mi caso he utilizado la versión 1.1 de OpenGL, y lo he bajado del centro de descarga de Microsoft. El nombre del archivo es *Opengl95.exe*.

Seguidamente descomprimir el contenido en un directorio temporal y copiar los archivos generados en los siguientes directorios:

1. *.h en *C:\Archivos de programa \Microsoft VisualStudio \VC98 \Include \GL*
2. *.lib en *C:\Archivos de programa\Microsoft VisualStudio\VC98 \Lib*
3. *.dll en *C:\Windows\System* para Windows 95, 98 y ME, o *C:\Windows\System32* en Windows NT, 2000 y XP.

2.5.2. Usando la librería adicional SDL

SDL (Simple Directmedia Layer) es una librería multimedia multiplataforma diseñada para proporcionar acceso a bajo nivel al audio, teclado, ratón, joystick, hardware 3D a través de OpenGL y al framebuffer del video 2D. Se utiliza para software de lectura de MPEG, emuladores, y muchos videojuegos populares.

Como ya he comentado es multiplataforma, y soporta: Linux, Windows, BeOS, MacOS clásico, MacOS X, FreeBSD, OpenBSD, BSD/OS, Solaris, IRIX y QNX. También hay código, aunque no es oficial, para Windows CE, AmigaOS, Dreamcast, Atari, NetBSD, AIX, OSF/Tru64, RISC OS y SymbianOS. Esta enorme portabilidad es la que me ha inclinado a usar SDL en lugar de otras librerías, como por ejemplo GLUT.

SDL está escrito en C, pero puede trabajar con C++ nativamente, y puede usarse con otros muchos lenguajes de programación, como Ada, Eiffel, Java, Lua, ML, Perl, PHP, Pike, Python y Ruby.

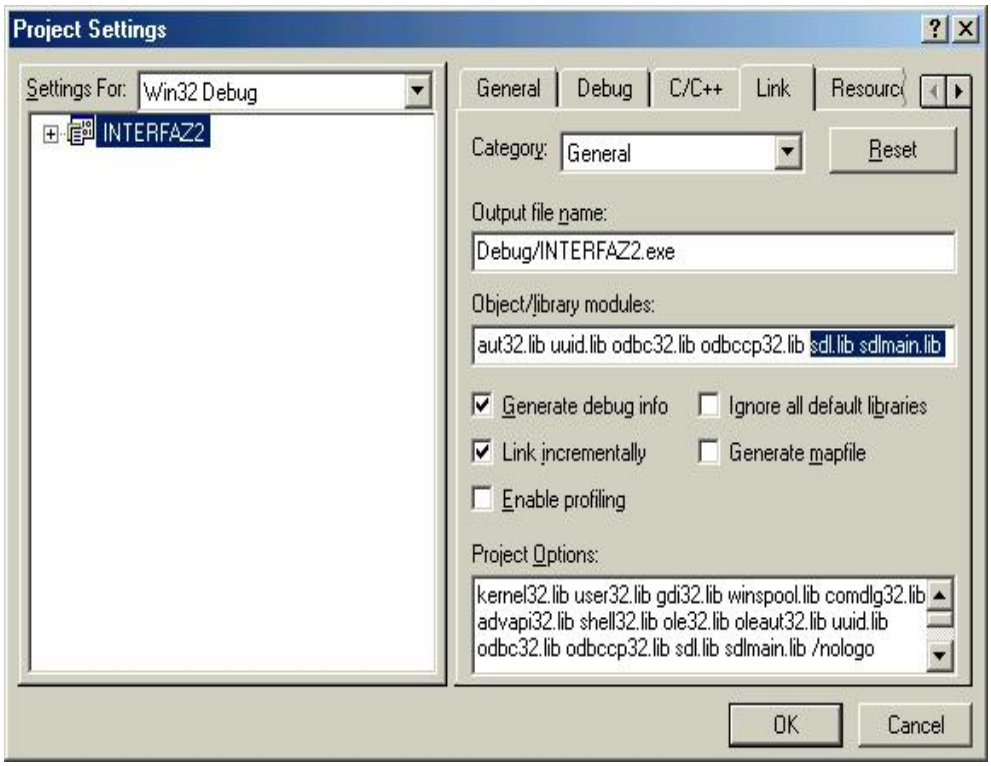
Para trabajar con SDL, en primer lugar debemos descargar de la página oficial de esta librería, que es *www.libsdl.org* la última versión estable de la librería de desarrollo para nuestro sistema operativo y compilador elegidos, en mi caso la versión 1.2.7 para Win32 y Visual C++ 6.0. El nombre del archivo descargado es en mi caso *SDL-devel-1.2.7-VC6.zip*. En esta página además contamos con muchísimas secciones para poder aprender a usar SDL, con tutoriales y ejemplos.

Una vez descargado el archivo, descomprimirlo. Aparecerán dos directorios principales que son los que nos interesan: *include* y *lib*. Se debe copiar el contenido del directorio *lib* dentro

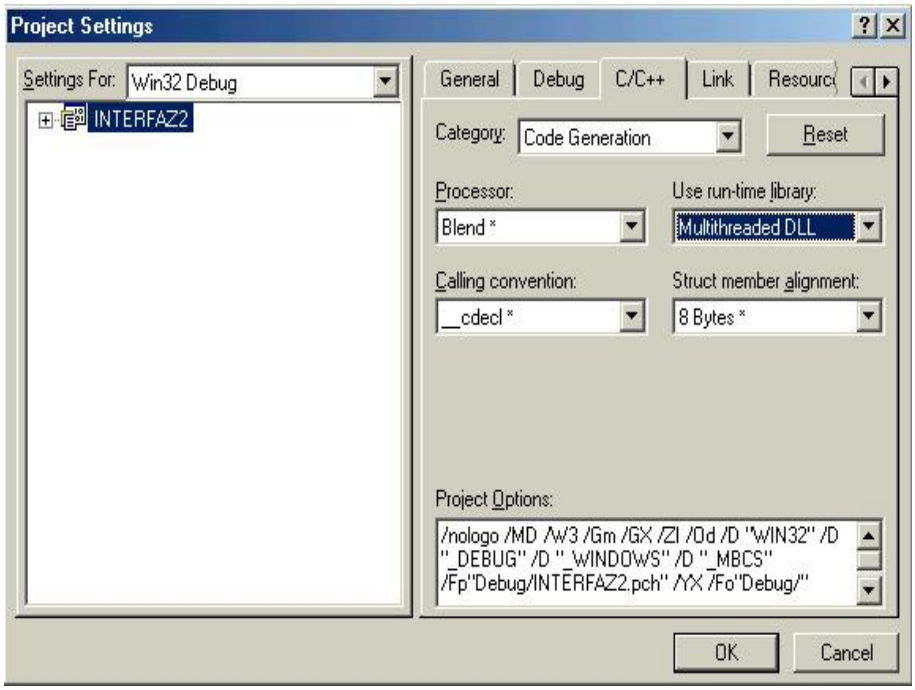
del directorio *lib* de MSVC6, en mi caso *c:\archivos de programa\Microsoft Visual Studio\VC98 \Lib*. Ahora debemos crear un directorio llamado SDL dentro del directorio *include* de nuestro MSVC6, en mi caso *c:\archivos de programa\Microsoft Visual Studio\VC98 \include\SDL*, y copiar todos los archivos *.h* del directorio *include* de SDL, y también copiarlos dentro del directorio include propiamente dicho *c:\archivos de programa \Microsoft Visual Studio \VC98 \include*.

Ahora para comenzar se ejecuta Visual C++ y se crea un nuevo proyecto, eligiendo *WIN32 Application*, y seguidamente elegir la opción 'an empty project' en la próxima ventana. A continuación añadir todos los archivos *.cpp* y *.h* del código fuente con la opción *Project/Add to project/Files*.

Y por último falta realizar algunos ajustes en la configuración del proyecto en *Project/Settings*. Pulsar en la pestaña *Link* y añadir 'sdl.lib' y 'sdlmain.lib' al final de la larga línea de archivos *.lib*.



Y finalmente pinchar en la pestaña *C/C++*, y del menú desplegable elegir la opción *Code generation* y marcar *Multithreaded DLL* del menú desplegable llamado *Use run-time library*.



Un aspecto a destacar es el archivo *SDL.dll* que se incluye con la distribución de la librería. Este archivo es necesario para ejecutar programas que usen SDL, y por tanto es necesario copiarlo en los directorios *c:\windows\system* en el caso de Windows 95, 98 o ME, o en *c:\windows\system32* para Windows NT, 2000 y XP. Otra opción es la de copiar dicho archivo en el mismo directorio del ejecutable.

Capítulo 3

Engine 3D: diseño e implementación

En este capítulo realizo una descripción del engine 3D que he creado, que el fin y al cabo es el gran grueso del proyecto. Todo este trabajo lo realicé una vez tuve resueltos los problemas que explico en el capítulo anterior. Fue entonces cuando diseñé el engine 3D y comencé la codificación, que me ocuparía casi 3 meses. Este capítulo también lo he dividido en varias secciones como los anteriores:

1. En la primera sección muestro el diseño del código fuente del engine 3D utilizando un diagrama de clases en UML. Con este diagrama intento facilitar la comprensión del programa en sí.

2. A continuación realizo una descripción preliminar de cada una de las clases que he creado en C++, así como del formato que he utilizado a la hora de codificar y documentar el código fuente.

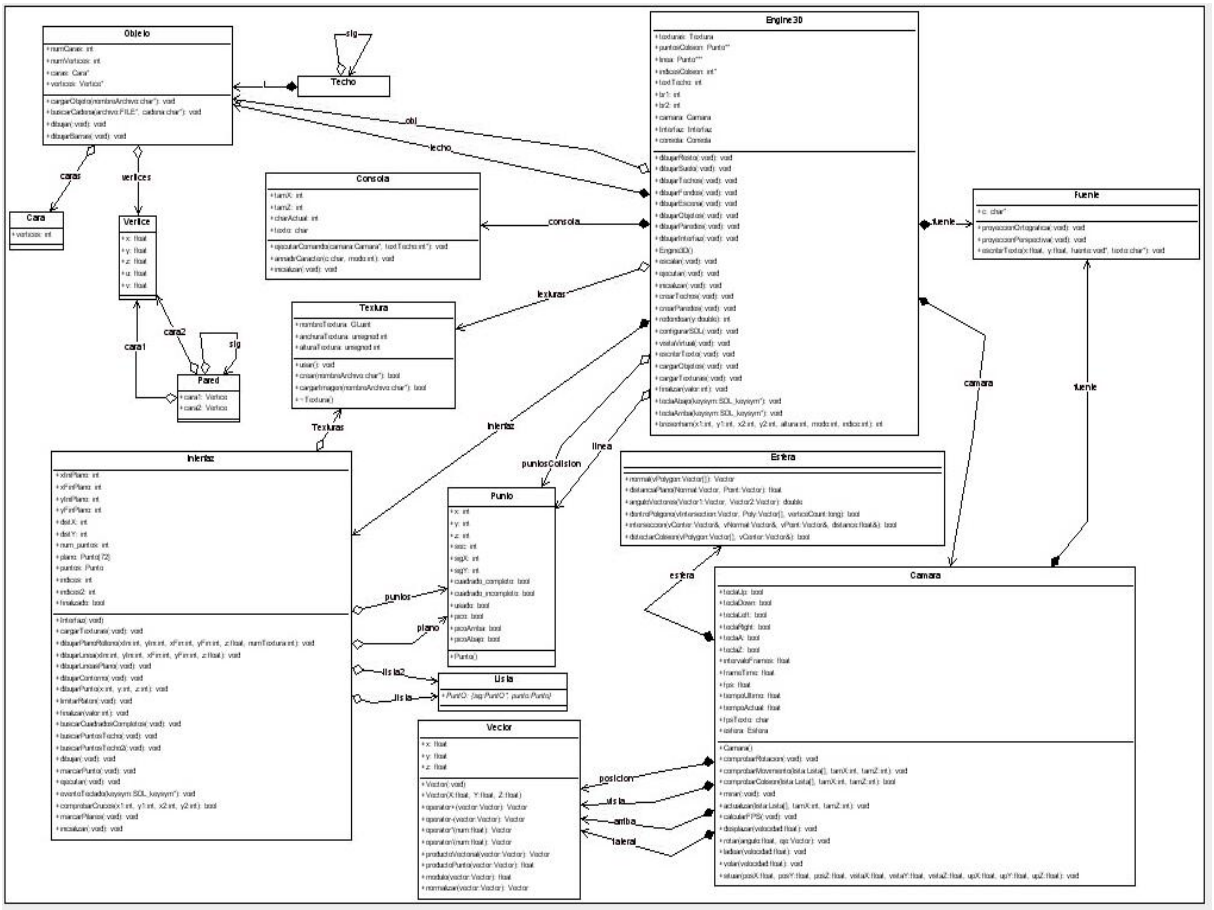
3. Y finalmente en el resto de las secciones del capítulo voy describiendo las clases principales del engine 3D, pero no desde el punto de vista del código fuente, sino desde el punto de vista de la funcionalidad en sí que posee cada una, explicando cuál es su papel dentro del programa principal y las funciones principales que desempeña. En estas secciones es al fin y al cabo donde explico las tareas o funciones básicas que obtuve tras realizar la descomposición funcional del sistema para el plan de proyecto.

3.1. Diagrama de clases en UML

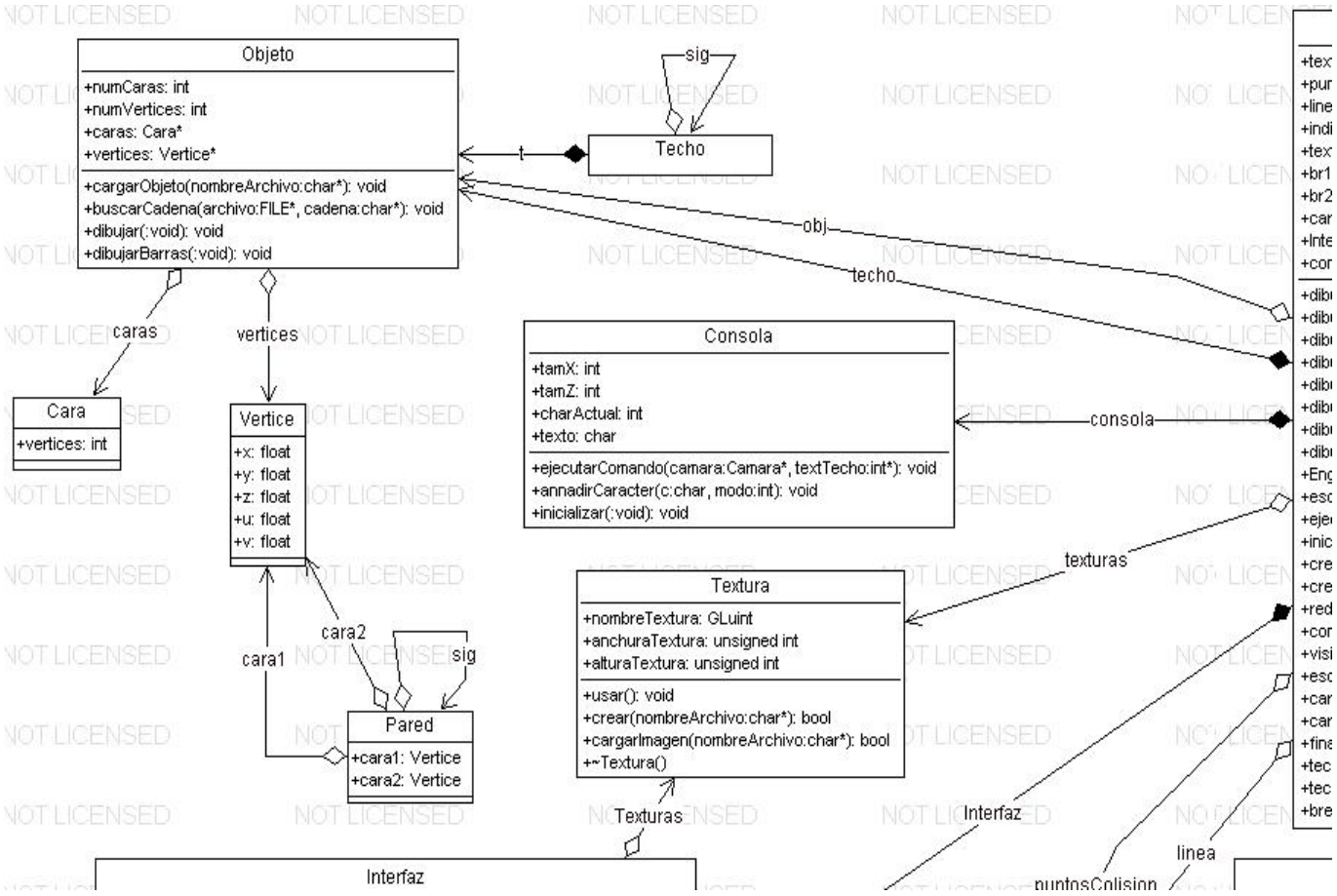
Para crear este diagrama de clases he usado la herramienta UMLStudio, que permite realizar ingeniería inversa para crear el diagrama UML a partir del código fuente. Gracias a este diagrama se pueden visualizar todas las clases que están incluidas en el sistema con sus métodos y atributos, así como las relaciones que se establecen entre ellas, que en el caso de mi proyecto son de tipo asociativo, de uso y contenimiento.

Al principio quise usar Rational Rose para generar dicho diagrama, sin embargo encontré en la red de redes esta herramienta que, siendo gratuita y además ocupando menos de 3 MB, tiene una funcionalidad enorme y además de permitir la generación del diagrama de clases mediante ingeniería inversa, también tiene la opción de guardar el resultado en disco como una imagen BMP.

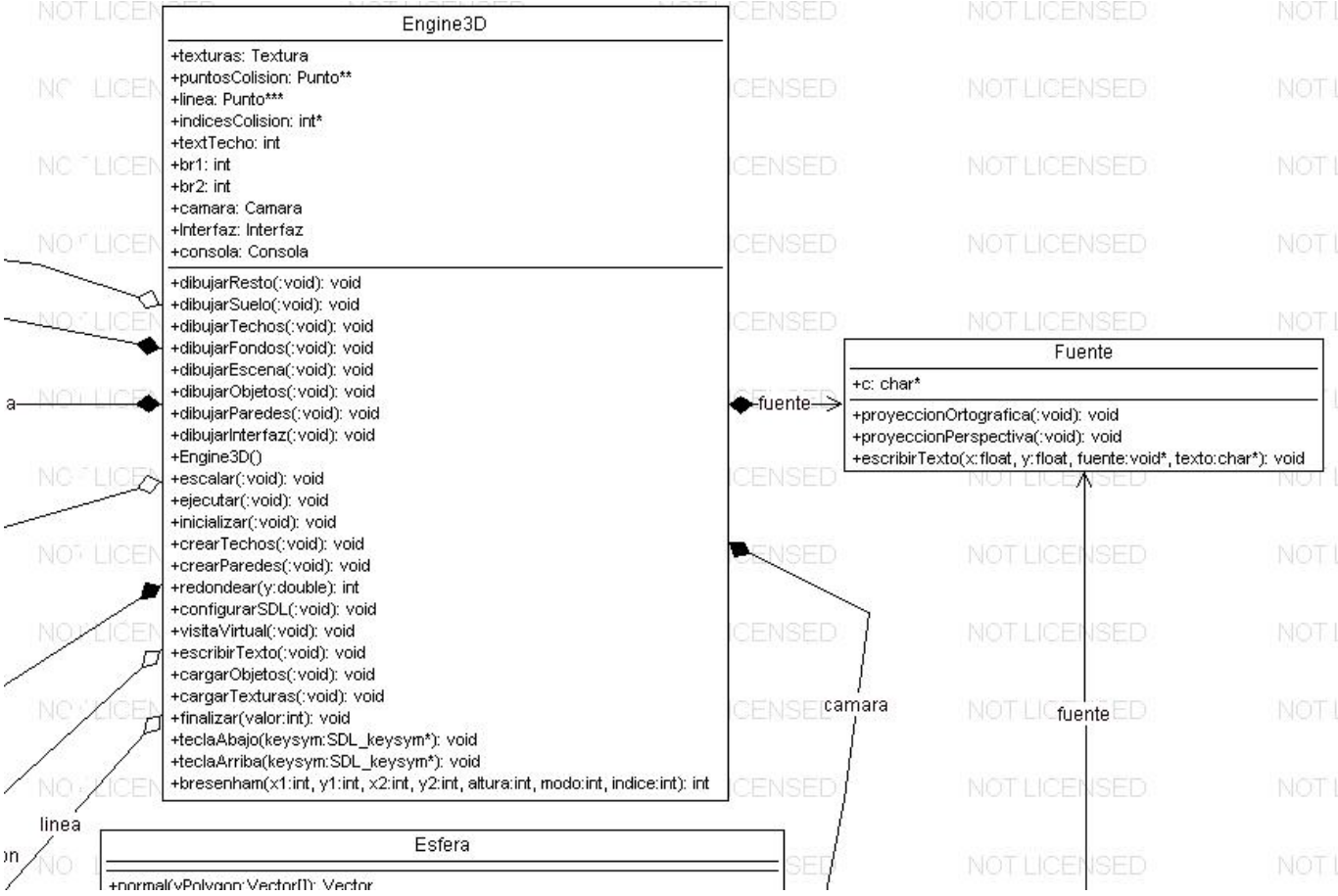
A continuación muestro el diagrama completo, para hacerse una idea de la forma y posición de las clases dentro del diagrama:



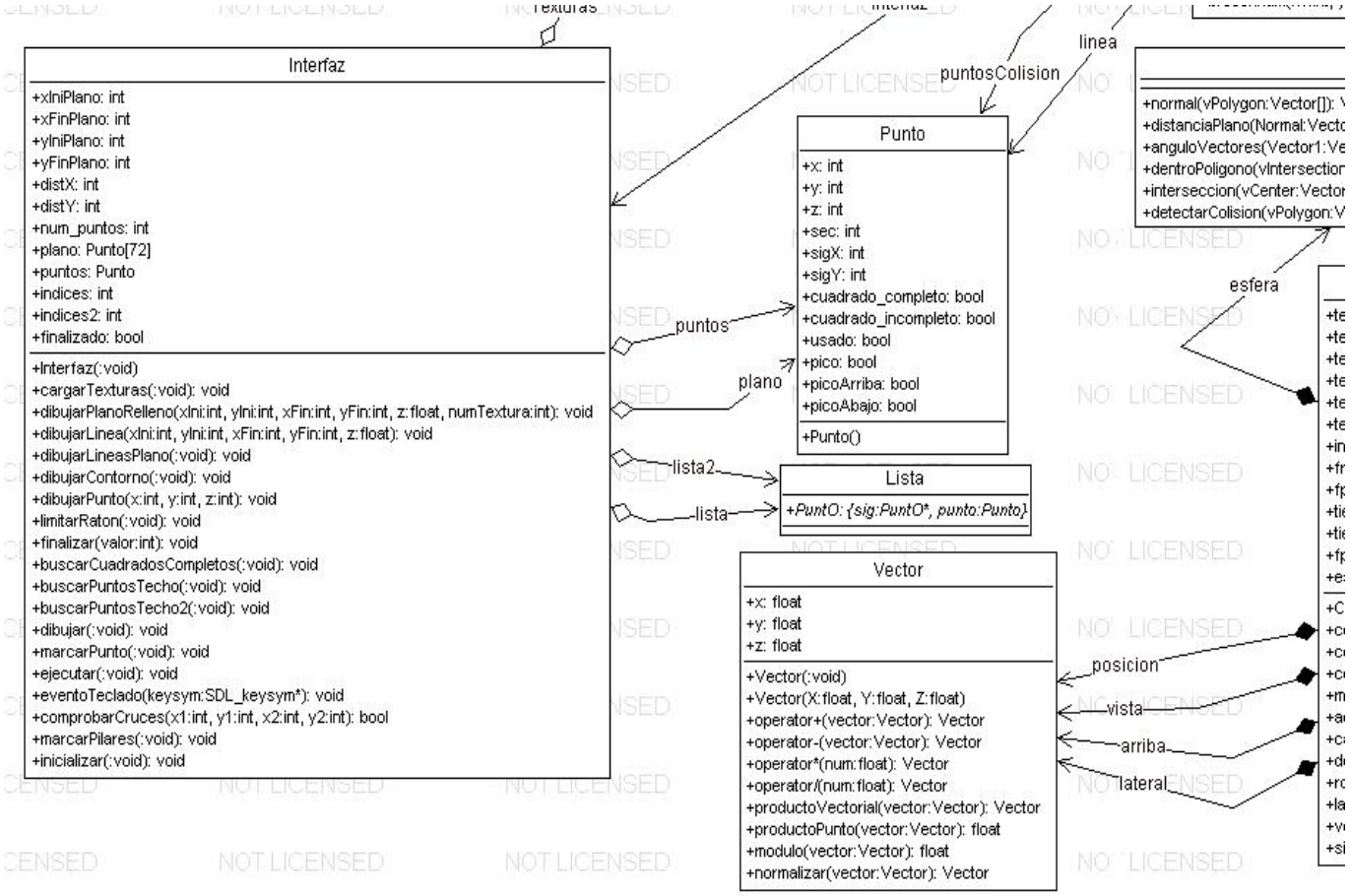
Para poder verlo con más claridad y más de cerca lo he dividido en 4 partes que muestro en las 4 siguientes páginas:



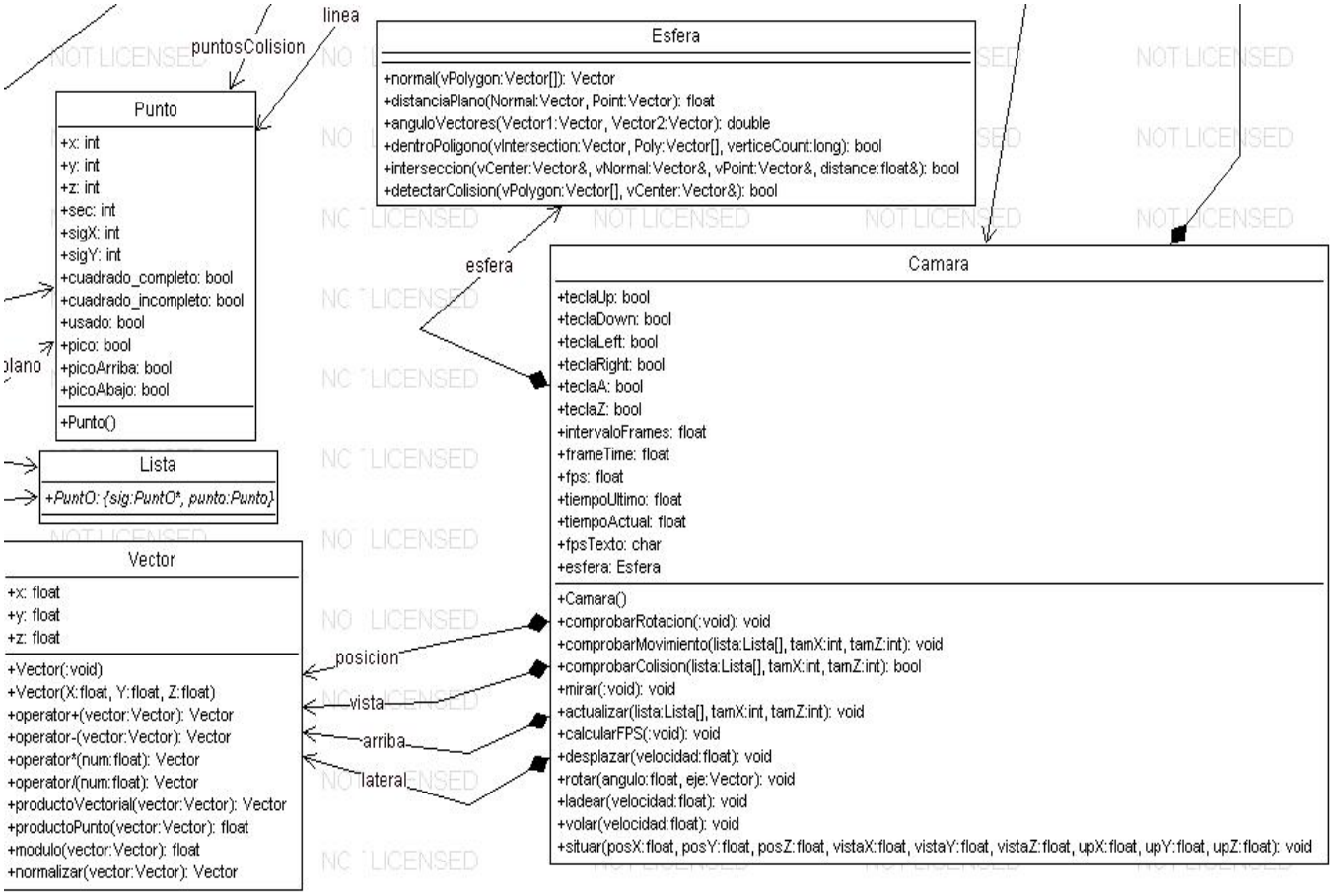
Parte superior izquierda



Parte superior derecha



Parte inferior izquierda



Parte inferior derecha

3.2. Descripción general del código

En mi código fuente escrito en Visual C++ 6.0 he creado un total de 13 clases, y el número de líneas de código resultante se aproxima a las 6000. En esta documentación no voy a explicar el código fuente, porque ese no es el objetivo del proyecto y además porque el propio código está lo suficientemente comentado. Pero sí voy a comentar la funcionalidad de las clases más importantes a grandes rasgos, así como la importancia dentro del programa. Las clases de mi código fuente son las siguientes:

- *Engine 3D*: clase principal que contiene el engine 3D.
- *Interfaz*: se encarga de dibujar y gestionar la cuadrícula del inicio del programa.
- *Camara*: movimiento y rotación de la cámara a través del mundo virtual.
- *Esfera*: implementación de la detección de colisiones de la cámara.
- *Vector*: clase auxiliar para cálculo de operaciones con vectores.
- *Consola*: modifica algunas propiedades del mundo virtual durante la visita.
- *Lista*: posee algunos objetos 3D adicionales que completan el modelo.
- *Punto*: para la cuadrícula de la interfaz inicial.
- *Textura*: carga y mapeado de texturas.
- *Fuente*: escribir en pantalla texto con OpenGL.
- *Objeto*: carga y maneja el modelo 3D desde disco en formato ASC.
- *Vertice*: los que posee el objeto 3D en formato ASC.
- *Cara*: las que posee el objeto 3D en formato ASC.

Además también está el método principal 'main', que es el que llama a la función principal del engine 3D para que arranque. Las librerías que he utilizado relacionadas con OpenGL han sido las siguientes:

- *glaux.h*
- *glut.h*
- *sdl.h*

El formato que he seguido a la hora de programar todas las clases ha sido el mismo, y he intentado comentar de manera adecuada el código fuente para que se pueda entender lo mejor posible:

1. Archivos **.h*:

   - En primer lugar hago una pequeña introducción de la clase, como la que se indica a continuación:

     ```
     //-------------------------------------------------------------
     // Clase: Engine 3D
     //-------------------------------------------------------------
     // Descripción: clase principal que contiene el ngine 3D
     //-------------------------------------------------------------
     ```

   - A continuación empleo la directiva 'ifndef', que es una condicional especializada para comprobar si un macro-identificador está definido o no. Así, en lenguaje C++ es muy común que se cometan cierto tipo de errores al trabajar con ficheros de cabecera. Por ejemplo: en mi caso tengo un archivo denominado 'constantes.h', donde he agrupado una serie de constantes que utilizo en muchas clases distintas. Como algunas de esas clases incluyen a otras (con 'include'), seguramente ocurrirían errores de compilación del tipo *... Multiple declaration for 'FILE'* indicándonos que estamos redefiniendo los símbolos de 'constantes.h' la segunda vez que la cabecera es incluida. La solución es disponer el código como indico a continuación:

     ```
     #ifndef _CLASEX_H
     #define _CLASEX_H
     #include 'Constantes.h'
     ...
     #endif
     ```

     Y seguidamente escribo las diferentes librerías que incluye la clase (con 'include'), ya sin problemas gracias a 'ifndef'.

   - Y por último defino la clase con sus atributos y métodos, siempre en este orden. Para la mayoría de los atributos añado una pequeña descripción de los mismos dentro de la clase. De los métodos no comento nada porque lo hago posteriormente antes de su implementación.

2. Archivos **.cpp*:

- Comienzan estos archivos de la misma manera que su equivalente .h con una descripción de la clase.
- A continuación incluyo todas las librerías necesarias para la implementación de los métodos de la clase.
- Y seguidamente implemento todos los métodos uno a uno, con una pequeña introducción para cada uno donde hago una descripción y explico sus parámetros.

```
//————————————————————————————————
// Nombre: Engine3D
// Descripción: constructor de la clase
// Parámetros: ninguno
//————————————————————————————————
```

En los próximos apartados voy a explicar cómo he realizado toda la funcionalidad de mi programa. Para que sea más fácil de entender el funcionamiento, iré comentando las diferentes clases en el mismo orden que se utilizarían en el programa, y así el lector podrá seguir el control lógico del mismo.

3.3. Clase Interfaz

Comienzo con esta clase porque el programa empieza mostrando una interfaz con la cuadrícula del invernadero, donde el usuario introduce los vértices que delimitan las paredes exteriores del mismo. Esta es una de las clases más importantes, ya que se encarga de la transformación 2D a 3D, es decir, convierte el polígono bidimensional dibujado por el usuario mediante el ratón en una estructura 3D correspondiente al invernadero equivalente.

3.3.1. Descripción inicial de la interfaz

Esta clase posee una serie de atributos, donde el más importante es el atributo 'plano', que es una matriz que se corresponde con los distintos puntos sobre los que puede pulsar el usuario con el ratón sobre la cuadrícula. Dicha matriz tiene objetos de la clase 'puntos', que explicaré posteriormente. Cuatro atributos guardan los límites X e Y superior, inferior, izquierdo y derecho de la cuadrícula del invernadero, y también hay dos atributos que especifican la distancia en X e Y entre cada dos puntos consecutivos.

El resto de atributos guardan las texturas utilizadas como imágenes en la interfaz, el número de puntos que el usuario ha marcado para delimitar la forma del invernadero o listas dinámicas de objetos necesarios para la construcción del invernadero.

Llegados a este punto debo hacer un par de comentarios sobre esta interfaz inicial:

1. Uno de los requisitos que debe cumplir el proyecto es que se puedan configurar algunos parámetros del invernadero, como por ejemplo la distancia entre pilares y la cubierta del mismo (policarbonato o plástico). En un principio pensé incluir dichos parámetros en esta interfaz inicial, para que el usuario pudiera elegir la configuración deseada. Pero tras meditarlo llegué a la conclusión de que sería muchísimo más interesante incluir dicha configuración en la propia visita virtual en forma de 'consola de texto', de tal modo que el usuario introdujera una serie de comandos que permitieran modificar en tiempo real y de manera dinámica varios parámetros del invernadero, y pudiera así contemplar directamente la diferencia entre una configuración y otra. Aunque esta segunda opción que elegí es mas compleja de implementar que la primera, los resultados obtenidos han valido la pena.

2. Otro de los eternos dilemas con los que tuve que luchar fue el hecho de diseñar una interfaz que fuera muy sencilla de manejar, y que el usuario pudiera darle forma al invernadero de la manera más simple posible. Para ello pensé en todo tipo de métodos. Los primeros eran sencillos, sí, pero ineficaces.

a) Uno de los primeros que trabajé consistía en pinchar sobre cada una de las cuadrículas para hacerlas 'parte del invernadero'. De esta forma cada cuadrícula se comportaba como un interruptor: ON u OFF, es decir, pertenece o no al invernadero. Sin embargo con esta técnica no podía construir invernaderos irregulares, que era lo deseado, ya que el ángulo entre cada dos paredes consecutivas siempre sería de 90º. Además podía crear muchos invernaderos independientes, y no era uno de los objetivos del proyecto. El verdadero objetivo es crear un solo invernadero, pero que pueda poseer una forma irregular.

b) También pensé en incluir una serie de 'elementos constitutivos básicos', como por ejemplo pilares, arcos, etc... y que fuera el propio usuario el que construyera el invernadero, como por ejemplo ocurre en muchos programas de diseño 3D. Sin Pero esta propuesta era cuanto menos descabellada, ya que el trabajo por parte del usuario sería tan complejo que éste rechazaría la herramienta. Por lo tanto necesitaba una método muy sencillo de generación automática del invernadero 3D.

c) Fue entonces cuando comencé a perfilar la técnica que he acabado utilizando: el usuario pincha con el ratón sobre los vértices deseados, de tal forma que esos vértices son los extremos de las paredes del mismo. Como se puede observar, es un método muy simple, el más simple que he podido crear, para que cualquier usuario, por muy inexperto que sea, pueda crear su propio invernadero 3D. Tan sólo hay que tener en consideración unos pocos detalles:

- Se puede crear un solo invernadero, es decir, aunque el polígono sea grande, la región interna del mismo deberá estar toda conectada.
- Como las paredes del invernadero no pueden cruzarse entre sí, las líneas dibujadas por el usuario tampoco podrán hacerlo. Para evitar cruces de líneas utilizo un algoritmo que explicaré en breve.
- Para finalizar la construcción del invernadero, el usuario simplemente tendrá que unir el último vértice (puntero del ratón) con el primero que marcó. De este modo se cierra el polígono y comienza la generación del invernadero 3D.
- El mínimo número de vértices pinchados por el usuario deberá ser igual a 4, es decir, un invernadero de forma triangular (teniendo en cuenta que el primer y último vértice serán el mismo, tenemos un triángulo).

3.3.2. Funcionalidad principal de la interfaz

El método principal es el 'ejecutar', y es el más importante porque en torno a él giran el resto de métodos. Su código básicamente es el siguiente:

```
// Bucle principal
while(!finalizado)
{
   // Captura eventos
   while(SDL_PollEvent(&event))
   {
      // Determina tipo de evento
      switch (event.type)
      {
         // Finaliza
         case SDL_QUIT:
              finalizado=true;
              break;
         // Pulsación de tecla
         case SDL_KEYDOWN:
              eventoTeclado(&event.key.keysym);
              break;
         // Movimiento de ratón
         case SDL_MOUSEMOTION:
              limitarRaton();
              break;
         // Pulsación de botón de ratón
         case SDL_MOUSEBUTTONDOWN:
              marcarPunto();
              break;
      }
   }
   // Dibuja la interfaz
   dibujar();
}

// Acciones para finalizar
buscarCuadradosCompletos();
buscarPuntosTecho();
```

Como se puede comprobar en el código fuente, la interfaz se dibuja en cada iteración. Además antes se comprueba si ha ocurrido algún evento de teclado o ratón (gracias a la función

'SDL_PollEvent' de la librería SDL). En este caso los eventos importantes son: pulsación de tecla (SDL_KEYDOWN), movimiento del cursor del ratón (SDL_MOUSEMOTION) y pulsación de un botón del ratón (SDL_MOUSEBUTTONDOWN).

Gracias a esta librería auxiliar de OpenGL se consigue una programación orientada a eventos, que captura eventos de teclado y ratón y permite realizar una u otra opción dependiendo del tipo de evento capturado. Esto es una de las muestras de la potencia de la librería 'SDL'.

Desde este método principal 'ejecutar' por lo tanto se llaman al resto de métodos de la clase, que los he clasificado en tres grandes grupos:

- Los métodos que se ejecutan cuando ocurre un evento de ratón o teclado: 'eventoTeclado', 'limitarRaton' y 'marcarPunto'.
- El método 'dibujar' que muestra la interfaz en pantalla.
- Aquellos métodos que se ejecutan una vez el usuario ha cerrado el invernadero, y por lo tanto realizan un postprocesamiento para continuar la conversión de 2D a 3D: 'buscarCuadradosCompletos' y 'buscarPuntosTecho'.

Es decir, en la interfaz inicial el programa está continuamente esperando a que ocurra un evento por parte del usuario a la vez que está dibujando dicha interfaz. Una vez se termina la creación del invernadero, se pasa a la segunda fase que es la construcción del invernadero 3D.

3.3.3. Eventos del teclado

Cuando el usuario pulsa una tecla, ésta es procesada. De tal modo que en esta interfaz inicial existen dos posibilidades desde teclado: que el usuario pulse la tecla ESC, con lo que finaliza la ejecución del programa, o que pulse la tecla A, que hace que comience de nuevo el proceso de creación del invernadero, es decir, borra todas las líneas dibujadas y además inicializa todos los puntos de la cuadrícula.

3.3.4. Limitación del ratón

El cursor del ratón no podrá salir de los límites establecidos por la cuadrícula del invernadero, de tal forma que queda atrapado para que el usuario no pueda salir de él. Esto lo he conseguido gracias a dos funciones de la API SDL, 'SDL_GetMouseState' que toma la posición actual del ratón y 'SDL_WarpMouse' que coloca el cursor en una posición determinada.

3.3.5. Dibujando la forma del invernadero

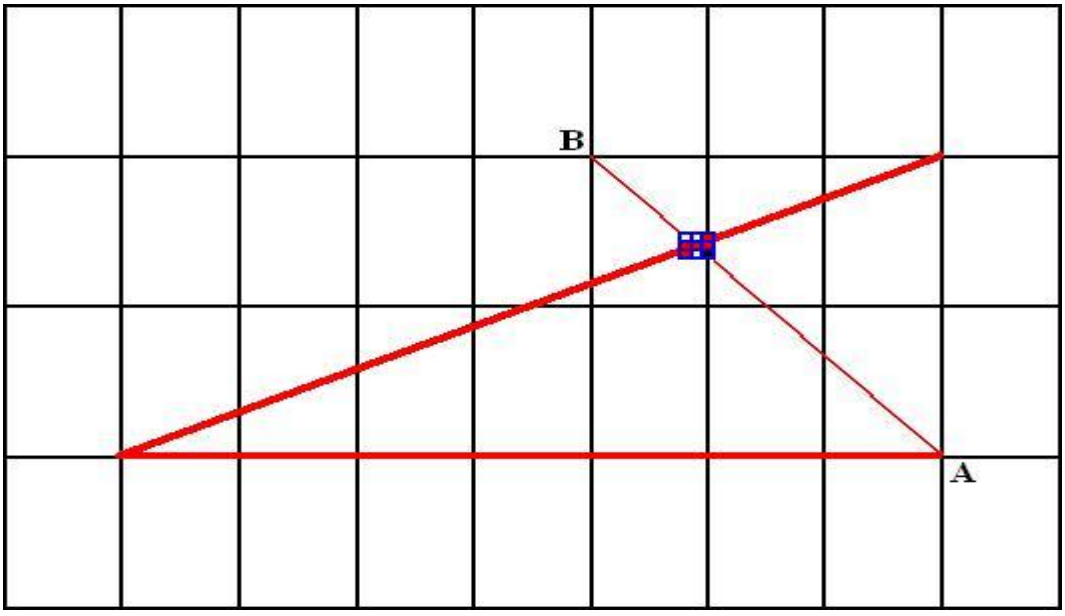
Cuando el usuario pulsa la tecla del ratón se realizan una serie de pasos:

- En primer lugar, la posición del ratón se aproxima al vértice más cercano, para que resulte mucho más fácil el proceso de diseño del invernadero para el usuario. Para ello redondeo la posición del ratón y le asigno la del vértice más cercano.

- En segundo lugar se comprueba si la nueva línea (correspondiente a una pared del futuro invernadero tridimensional) colisiona con otra línea ya dibujada (recordar que las paredes de un invernadero no colisionan entre sí).

  Para ello utilizo el algoritmo de trazado de líneas de Bresenham entre el punto inicial y final de la nueva línea, con una modificación añadida por mí: gracias a la función 'glReadPixels', leo un cuadrado 3x3 por cada punto generado por el algoritmo de Bresenham. Como el color de las líneas dibujadas es el rojo (en RGBA sería 255,0,0,255), compruebo si en dichos cuadrados existen píxeles rojos, si existen entonces hay colisión entre ambas líneas.

  Por ejemplo, en la siguiente situación no se podría realizar una línea entre los puntos A y B, ya que existe un cruce con otra línea anterior. El cuadrado azul muestra el lugar donde se detectaría la colisión de ambas líneas:

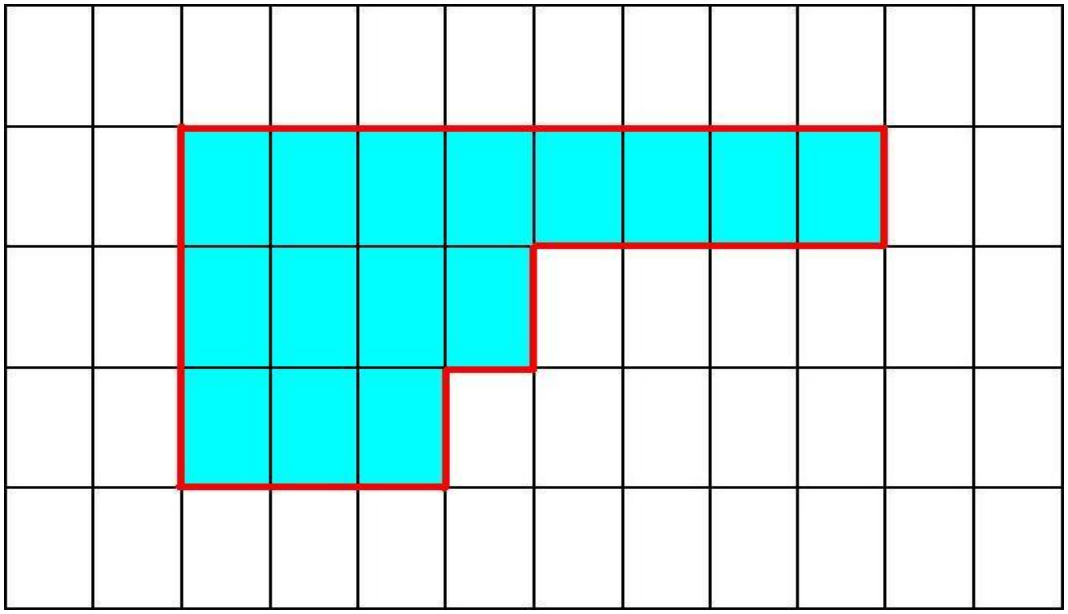


- En tercer lugar, si no existe cruce entre líneas marco definitivamente el punto como 'usado', y lo enlazo con el inmediatamente anterior marcado. Si se ha cerrado el invernadero, finaliza la ejecución de la interfaz y comienza la transformación de 2D a 3D.

3.3.6. Transformación del polígono 2D al invernadero 3D - Cuadrículas completas

Esta es una de las partes más importantes de mi programa, y realmente es la parte innovadora de mi proyecto: cómo convertir un simple polígono irregular en un invernadero. Este proceso comienza en el momento en que se cierra el polígono que representa la forma del invernadero, y está dividido a su vez en varias partes.

Lo primero de todo es localizar aquellas cuadrículas que pertenecen íntegramente al invernadero, o lo que es lo mismo, que se encuentran dentro del polígono del invernadero y además no están atravesadas por ninguna línea. Cada uno de estas cuadrículas corresponde con uno de los módulos básicos que cree con Autocad, de modo que posteriormente sólo tendré que dibujar el modelo 3D de Autocad en la posición de cada cuadrícula 'completa'. Esta es la parte más sencilla. Pero si sólo hubiera tenido en cuenta estas cuadrículas, las formas creadas sólo podrían ser 'regulares', es decir, no podría haber 'picos' en las paredes que formaran un ángulo distinto de 90º. En la siguiente imagen destaco las 'cuadrículas completas':

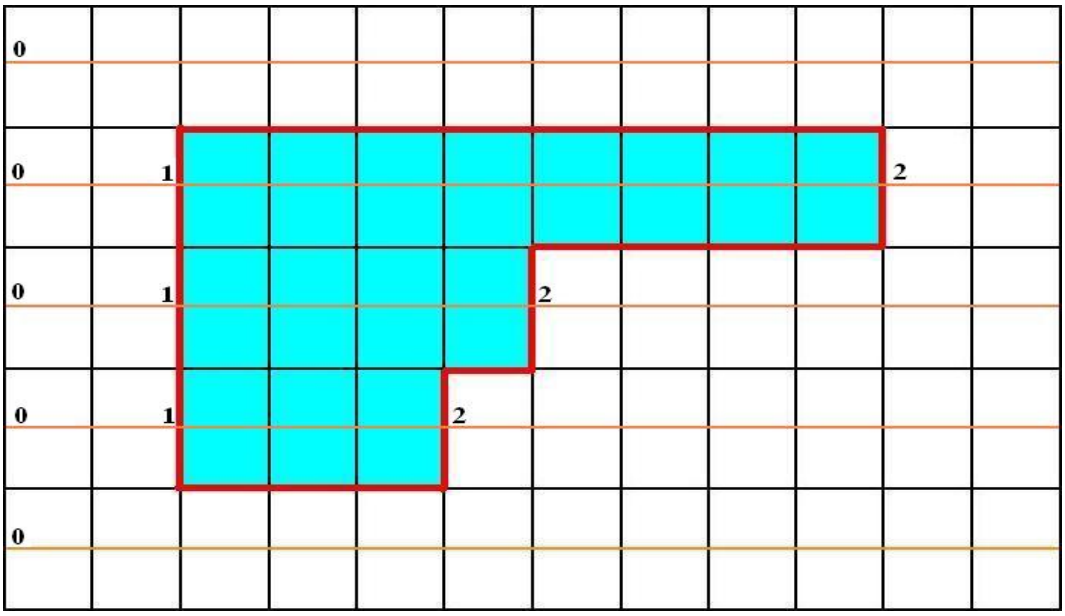


Como se puede ver el ángulo formado por cada dos paredes consecutivas es siempre de 90º ó 270º, pero nunca hay picos. Por eso denomino a estos invernaderos 'regulares'.

En realidad, aunque sea sólo de esta manera, se pueden crear muchos tipos de invernaderos diferentes, aunque todavía no se permite una total adaptación del invernadero al terreno del propietario. Para aumentar la precisión, permitiré también ángulos no rectos entre paredes, es decir con forma de picos, como explicaré posteriormente.

Para detectar las 'cuadrículas completas', he usado el siguiente algoritmo:

- Para cada una de las filas de rectángulos, captura una línea horizontal de 1 píxel de grosor que recorre el plano de izquierda a derecha a una altura de medio rectángulo, como muestro en la siguiente imagen:

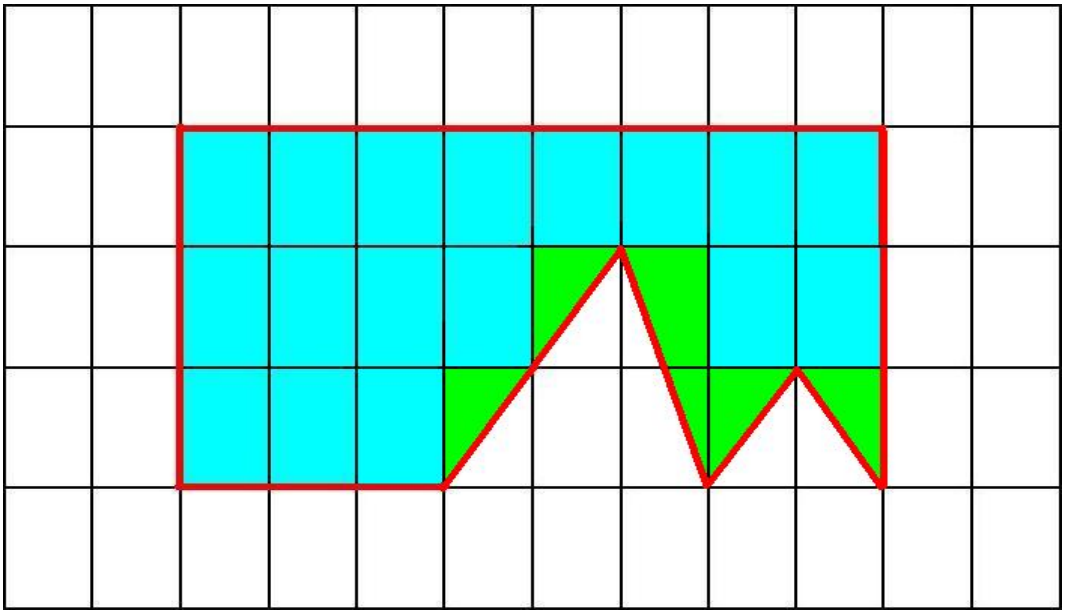


- Recorre la línea de píxeles y cada vez que encuentra 1 píxel rojo aumenta el contador de píxeles rojos para esa línea. De esta forma el número de píxeles rojos será siempre impar cuando estamos dentro del polígono, y par cuando estamos fuera.

  Cada vez que se encuentra al comienzo de una cuadrícula (línea izquierda de la misma), si el número de píxeles rojos es impar significa que esa cuadrícula pertenece al invernadero. Si el número de píxeles rojos es 0 ó par, las cuadrículas no pertenecen al invernadero.

  Como se puede apreciar en la imagen anterior, la cuenta de número de píxeles rojos comienza de izquierda a derecha, de tal modo que al comenzar tienen cuenta igual a 0. Seguidamente aumenta en una unidad cuando se encuentra con la primera línea roja, por lo tanto las cuadriculas que vengan a continuación están dentro del polígono (forman parte del invernadero). Y finalmente se aumenta en otra unidad cuando encuentra otra línea roja, por lo que como ya tenemos un número par de píxeles rojos encontrados, las cuadrículas que vengan a continuación no pertenecen al polígono.

  Sin embargo surgen ciertos problemas, cuando por ejemplo se intentan crear formas irregulares. En la siguiente imagen se aprecian los problemas:



3.3.7. Transformación del polígono 2D al invernadero 3D - Cuadrículas incompletas

En la imagen anterior, las cuadrículas coloreadas de verde son incompletas, ya que, aunque pertenecen al polígono del invernadero, están atravesadas por líneas rojas (paredes del invernadero), y por lo tanto su forma no es completa. A este tipo de invernaderos les llamo 'irregulares' porque existen picos (intersecciones entre líneas consecutivas) que forman ángulos distintos a los 90^o y 270^o.

Por lo tanto, al algoritmo anterior habría que añadir que una cuadrícula es clasificada como 'completa' si el número de píxeles rojos es impar y además no está atravesada por una línea roja. Para comprobar esto, cuando encuentro una cuadrícula candidata a pertenecer al polígono (número de píxeles rojos impar) capturo todos los píxeles que hay en su interior, y a continuación los recorro para comprobar si existe algún píxel rojo. En cuanto encuentro alguno, dicha cuadrícula está atravesada por una línea, y por lo tanto el tratamiento es diferente.

En este caso, al tener una forma irregular, estas partes del invernaderos las tengo que construir yo mismo a través de mi programa, ya que no ocurre como con las cuadrículas completas, que se puede crear un modelo predefinido en Autocad. La construcción de estas zonas irregulares la he dividido en varias etapas, que comienzan con la detección de la forma irregular de los techos, para finalmente transformar un techo básico construido en Autocad a dicha forma. Para ello hay que comenzar teniendo en cuenta que cada techo, mirado desde la parte superior, tiene 4 puntos que delimitan su forma (ya que es rectangular). Por lo tanto la primera tarea que tenía que realizar era buscar estos cuatro puntos. Sin embargo esta tarea se me presentó algo más

compleja de lo que en un principio imaginé, ya que pueden darse muchos casos diferentes como muestro a continuación:

En principio, un techo debería ocupar todas la cuadrículas completas de una misma línea. Las cuadrículas tienen forma rectangular debido a que un módulo básico construido en Autocad, como ya vimos, tiene por defecto el tamaño de 9x5 metros. Por lo tanto el arco con las barras curvas coinciden con la parte más alargada de la cuadrícula, como vimos en el diseño básico en Autocad. Como consecuencia para ahorrar polígonos mi idea es 'alargar' ese techo básico creado que se ajusta en principio a una cuadrícula para ajustarlo a las cuadrículas que sea necesario. Pero para ello lo primero que necesito es encontrar los cuatro puntos de los techos de cada fila.

En el ejemplo anterior la fila superior de cuadrículas posee un solo techo. Además dicho techo, aunque abarca un total de 10 cuadrículas, es 'regular', ya que las líneas laterales son totalmente verticales. Si observamos la segunda fila de cuadrículas se puede comprobar que también existe un solo techo, pero éste en cambio es 'irregular', ya que la parte derecha está inclinada. Si se observa la fila inferior se pueden contar un total de 3 techos y además 'irregulares'. Y además estos techos poseen un inconveniente añadido: a diferencia de los techos de las otras dos filas, éstos son triangulares, por lo que sólo tengo 3 de los 4 vértices de cada uno de los techos. Este problema tiene solución, como veremos a continuación.

Para la creación de cada uno de los techos he realizado una serie de pasos:

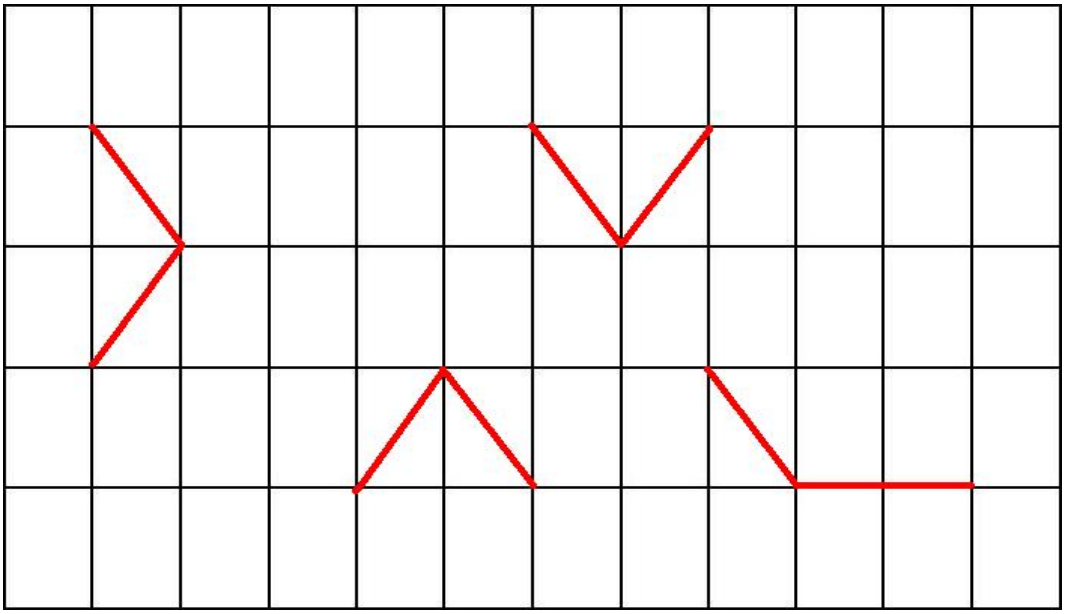
1. En primer lugar consigo los 4 puntos de los extremos del techo. Es importante coger los 4 y no 3 como ocurriría en los techos triangulares, que en realidad tienen también 4 puntos

pero hay dos que coinciden en un vértice del triángulo.

Como se ve, cuando hay un pico hacia arriba o hacia abajo, en ese mismo punto hay dos vértices. Esto lo tengo que tener en cuenta. Para conseguir los 4 puntos de los techos, para cada una de las filas hago los siguientes pasos:

a) Capturo 3 franjas horizontales de píxeles del grosor de 1 píxel. La primera franja está justo a la altura de los vértices de dicha línea, otra a 1 píxel más de altura y la última a 1 píxel menos de altura.

b) Voy buscando píxeles rojos en la franja central. Cuando encuentro uno busco también en las líneas superior e inferior, en una posición cercana al píxel encontrado, si también hay algún píxel rojo. Si sólo hay píxeles arriba o abajo, se trata de un pico. Si hay a ambos lados no hay pico; esto en principio. Con este método puedo detectar los picos que hacen que un techo tenga 3 puntos en lugar de 4. En la siguiente imagen se ven algunos ejemplos:



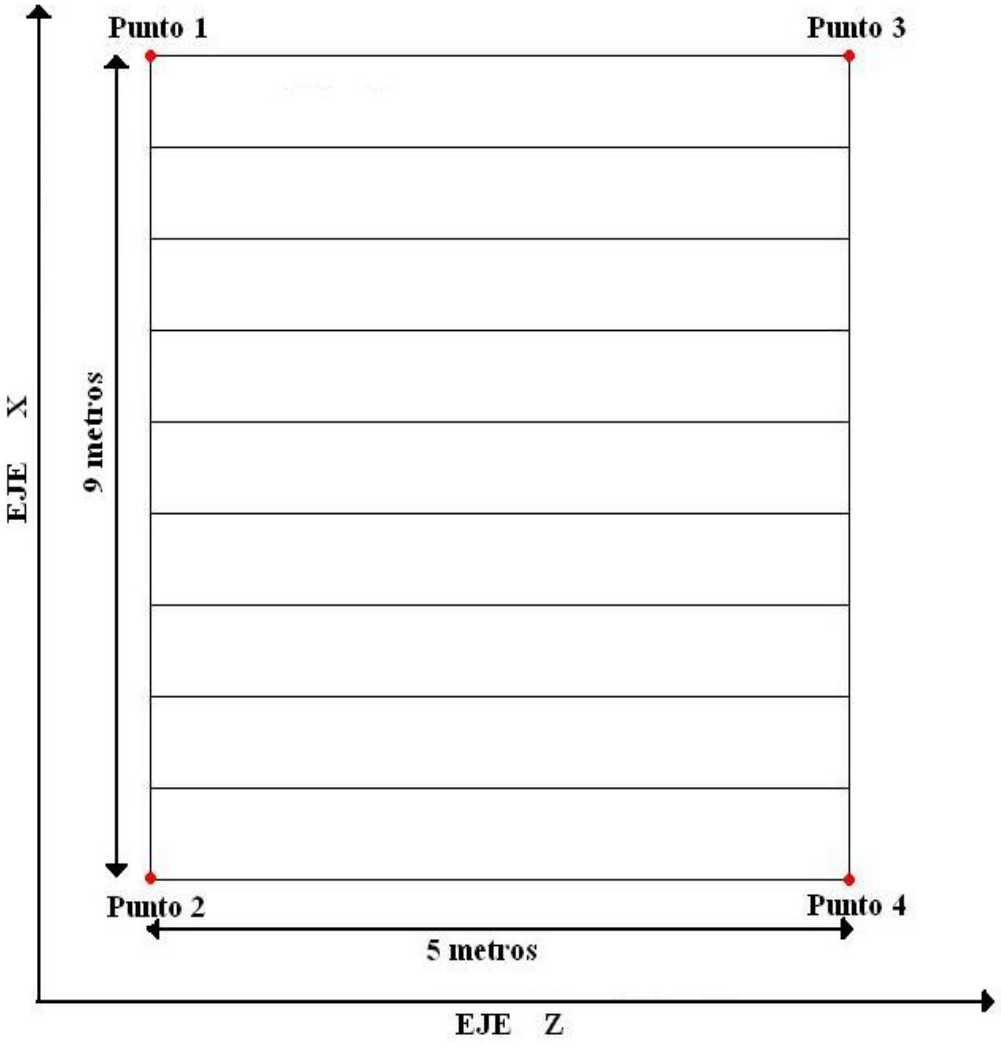
En el ejemplo situado más a la izquierda se tomaría la intersección de las dos líneas como un solo punto de un techo. Sin embargo, en los dos siguientes que son picos (uno hacia arriba y otro hacia abajo), se tomaría esa intersección entre ambas líneas como 2 puntos cada una, para que la pared contenga 4 puntos en total. En el ejemplo situado más a la derecha también se tomaría como 1 solo punto del techo.

El método que he utilizado por lo tanto es que, si encuentro píxeles rojos arriba y abajo de cada píxel rojo de la línea central lo tomo como un solo punto. Si sólo

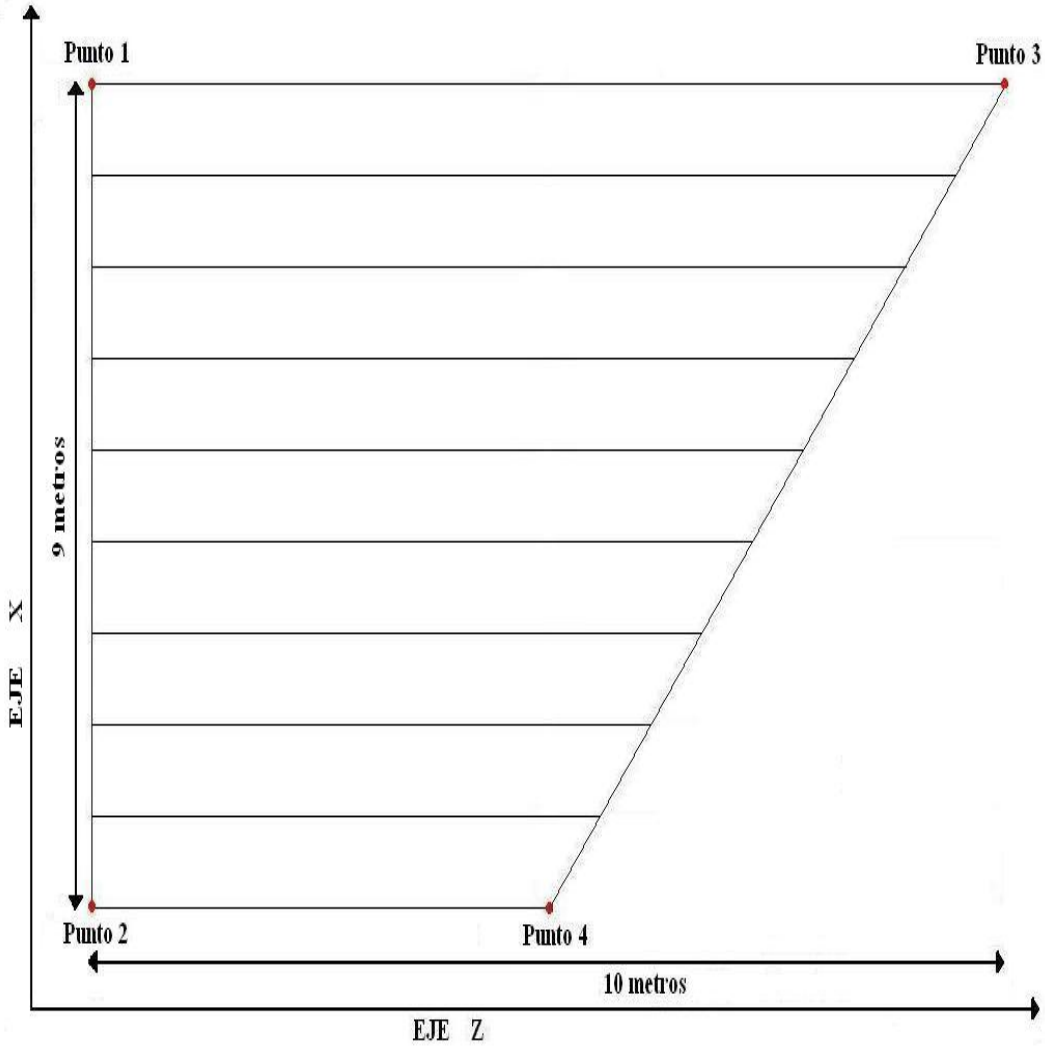
encuentro píxeles arriba o sólo abajo pues se trata de un pico y por lo tanto lo anoto como dos puntos del techo. Aquí también existe una excepción: como se puede ver en la imagen anterior, tal y como lo he planteado, el ejemplo situado más a la derecha lo tomaría como pico, cuando en realidad no es pico, sino que tiene que tomarlo como 1 solo punto del techo. Estos casos también los he tenido en cuenta, comprobando simplemente si una de las líneas que intersecta es horizontal o no.

3.3.8. Creación de techos

Una vez que se tienen los 4 puntos o extremos de cada uno de los techos, el siguiente paso consiste en adaptar la forma del techo básico construido en Autocad a esos puntos. Esta funcionalidad en realidad lo he incluido en la clase 'Engine3D', pero la explico aquí porque es la continuación de la sección anterior. En la siguiente imagen se pueden observar el módulo básico de techo visto desde arriba con los 4 puntos de los extremos resaltados:

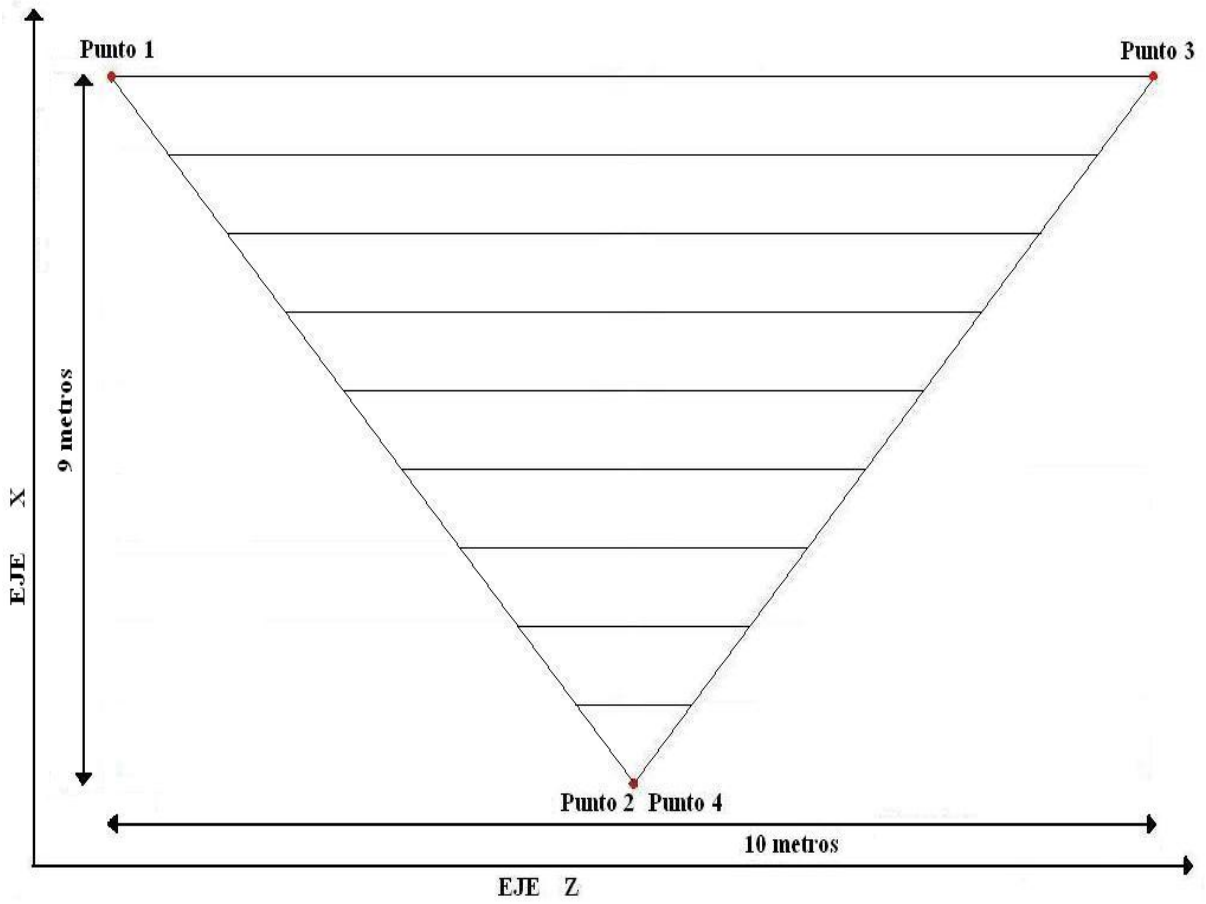


Hay un total de 9 polígonos que forman la bóveda semicircular del techo como ya comenté en la sección de Autocad. Si el techo estuviera formado por un único polígono rectangular, ajustar la forma a los 4 puntos nuevos de los extremos sería sencillo: sólo tendría que sustituir la coordenada Z del nuevo punto en el correspondiente. Notar que la coordenada X no varía nunca ni la Y tampoco, sólo la coordenada Z. En el siguiente ejemplo se puede ver cómo el 'punto 3' del techo ha sido modificado, variando así su forma y tamaño en el eje Z:



El tamaño en X del techo nunca varía, sólo el tamaño en Z, y en este caso hay una parte que tiene 5 metros y otra que tiene 10. Como ya he comentado, si el techo fuera un solo polígono tan sólo tendría que desplazar a través del eje Z los nuevos puntos. Sin embargo un techo está formado por 9 polígonos, luego existen vértices intermedios cuya coordenada Z debe ser modificada. En la imagen anterior estos puntos son todos los vértices que hay entre el punto 3 y 4 trazando una línea recta, que como se puede ver se han desplazado hacia la derecha respecto a la imagen original.

Ahora para modificar las coordenadas Z de todos esos puntos intermedios utilizo el algoritmo de trazado de líneas de Bresenham entre los puntos entre los puntos 3 y 4. De esta forma, y comenzando la línea desde el punto 4 hacia arriba, como los vértices están separados a una distancia de 1 metro, cada vez que 'recorro' esta distancia en vertical con Bresenham guardo el valor en el eje Z de ese punto, para asignarlo al punto intermedio correspondiente.



De esta forma consigo realizar techos de cualquier forma posible, pueden tener los dos laterales inclinados, y en ese caso también tengo que trazar una línea de Bresenham entre los puntos 1 y 2 para calcular sus puntos intermedios. En el ejemplo anterior además el punto 2 y 4 coinciden, ya que el techo tiene forma triangular. De todas formas el procedimiento para calcular su forma es el mismo: algoritmo de Bresenham entre los puntos 1 y 2, y entre los puntos 3 y 4 (aunque en este caso el punto 2 y el punto 4 son el mismo). Gracias a este método he realizado una transformación dinámica de objetos en tiempo de ejecución del programa, ha sido uno de los aspectos del proyecto a los que más vueltas le he dado y creo que le he dado una solución sencilla y eficaz.

No obstante esta tarea se complica un poco, ya que los 4 puntos de los techos los adquiero de la interfaz inicial en 2D donde el tamaño de las cuadrículas es pequeño, y luego tengo que transformar dicho punto a su equivalente en el mundo 3D y con distancias mayores (para poder modificar el techo virtual). Es una transformación inmediata cuando está controlada pero al principio me dio bastantes problemas.

Este es uno de los puntos que creo no hubiera sido capaz de desarrollar en Blender, ya que aquí he tenido que trabaja no sólo a nivel de objeto ni de cara ni de vértice, sino al nivel más bajo que existe: las coordenadas de un vértice. Esta modificación la planteé en Blender con las llamadas curvas IPO, que modificaban la forma de un objeto, pero sin embargo no permitía mucho control sobre dicha modificación, no tanto como me ha ofrecido OpenGL con Visual C++.

La determinación de la forma de los techos se realiza una sola vez, justo cuando se cierra el polígono de la interfaz y antes de comenzar la visita virtual.

3.4. Clase objeto

Esta es la clase que se encarga de la carga de objetos 3D en formato ASC desde el disco duro, así como de dibujar todos sus polígonos. Entre sus atributos tenemos el número de caras del objeto, así como el número de vértices. Además también hay un conjunto de caras y de vértices (ambos arrays dinámicos en C++), donde se guardan los datos.

Un vértice va a estar formado no sólo por las coordenadas X, Y y Z de la posición del mismo, sino también por las coordenadas U y V de mapeado para las texturas. He creado una clase que se llama 'Vertice' que contiene estos atributos. Un objeto 3D tendrá por tanto un número limitado de vértices.

Además, esos vértices se relacionan entre sí de 3 en 3 para formar las caras triangulares del objeto 3D. Por lo tanto una cara tan sólo debe poseer la referencia a cada uno de los 3 vértices de sus esquinas, nada más. De este modo tan sencillo evito la repetición de información, ya que los vértices pertenecen a más de una cara, y si hubiera estructurado los datos de tal modo que cada cara almacenara las coordenadas de sus 3 vértices, habría mucha redundancia de información. Este aspecto es importante de cara sobre todo al rendimiento y a la memoria usada por el programa.

3.4.1. Carga de objetos ASC y dibujo del mismo

En el formato ASC los vértices del objeto se encuentran al principio del archivo, y las caras después. Teniendo esto en cuenta, y como sé el número de vértices y caras que hay en total, voy buscando las cadenas de texto oportunas y cargando todos los datos del objeto tridimensional.

Para representar el objeto en escena, simplemente voy recorriendo todas sus caras una a una y en OpenGL dibujo un triángulo con sus tres vértices, mapeando la textura con las coordenadas U y V.

3.5. Clase Vector

Esta la considero como una especie de 'clase auxiliar', ya que es usada por otras clases para el trabajo con vectores. Incluye las operaciones básicas de vectores:

- Suma y resta de vectores, desde el punto de vista algebraico (sumando o restando sus componentes).
- Producto de un vector por un escalar: se multiplica cada uno de los componentes del vector por el número.
- Producto vectorial de dos vectores: yo lo calculo desde el punto de vista algebraico, que se obtiene de la siguiente manera:

$$xNuevo = (yA * zB) - (zA * yB)$$

$$yNuevo = (zA * xB) - (xA * zB)$$

$$zNuevo = (xA * yB) - (yA * xB)$$

- Normalización de un vector: se divide cada uno de sus componentes entre el módulo del vector, calculando el módulo con la siguiente fórmula:

$$modulo = \sqrt{x * x + y * y + z * z}$$

3.6. Clase textura

Esta clase se encarga de cargar texturas desde disco y guardarlas para poder utilizarlas posteriormente sobre algún polígono, mapeando su posición sobre el mismo. Un aspecto importante a comentar es que las texturas que tienen que cumplir una serie de requisitos:

- Aunque no tienen por qué ser cuadradas, tanto la altura como la anchura de la imagen deben ser potencia de 2. De tal modo que puedo tener imágenes de tamaño por ejemplo: 512x256, pero nunca 512x100 (100 no es potencia de 2).

- El formato que he elegido para las texturas es el BMP, y el motivo es muy simple: es quizá el formato de imágenes más sencillo que existe. Aunque con JPEG las imágenes ocupan mucho menos al estar comprimidas, tendría que haber elaborado un algoritmo de lectura de este formato, y como este no es uno de los objetivos de este proyecto, he usado un formato que no me diera problemas.

- El formato BMP de por sí no tiene transparencias, pero sin embargo he diseñado un método para poder utilizarlas. Por lo tanto en mi programa puedo usar texturas semitransparentes: unos píxeles son opacos y otros puede que transparentes (no se dibujan).

3.6.1. Crear texturas semitransparentes a partir de un BMP

Para conseguir el efecto de transparencia en algunos píxeles he elegido el color completamente negro para ello. Es decir, los píxeles de color negro absoluto no los dibujo. Los pasos que sigo para transformar una imagen BMP normal con 3 componentes (rojo, verde y azul) en una imagen con 4 componentes (los 3 anteriores más el cuarto que indica el nivel de transparencia del píxel) son los siguientes, y además lo aplico a cada uno de los píxeles:

- En primer lugar leo y compruebo el valor de sus tres componentes de color: rojo, verde y azul (RGB).

- A continuación añado un cuarto componente (el canal Alpha) que me indica el grado de transparencia del píxel. En mi caso, o son completamente transparentes (color negro) o completamente opacos (el resto). Por lo tanto si entre los 3 valores anteriormente leídos obtengo un color negro para el píxel, añado al canal alpha el valor transparente. En cualquier otro caso, añado el valor opaco.

3.6.2. Mapeado de texturas

Una textura se puede definir como un array de una o dos dimensiones de píxeles. A cada píxel se le denomina texel. En el caso de las texturas 1D, se trata de imágenes con un solo píxel de anchura por ejemplo, y varios píxeles de altura. Este tipo de imágenes no las usaré, pero pueden ser útiles para definir los 7 colores del arco iris, por ejemplo. En mi proyecto usaré texturas 2D, que poseen más de un píxel de ancho y alto, y en mi caso van a ser imágenes en formato BMP.

El mapeado de texturas es una técnica gráfica que consiste en aplicar uno o varios dibujos a las distintas caras de un objeto tridimensional, para conseguir así un efecto muchísimo

más realista. Además, gracias al uso de texturas se consigue reducir enormemente el número de polígonos mostrados en escena, ya que por ejemplo si queremos dibujar una puerta, tan sólo deberemos aplicar una foto de una puerta a una cara rectangular, y como resultado tendremos una puerta tan real como lo sea la foto que hemos usado. En este aspecto me he preocupado bastante por el uso de texturas en el desarrollo de este proyecto, ya que uno de los principales objetivos que me propuse desde un principio fue que la visita virtual fuera lo más fluida posible, y para ello es necesario sobre todo reducir mucho el número de polígonos en escena, intentando por otro lado no quitar demasiado realismo a la misma. Y la mejor manera de conseguirlo es usando las texturas adecuadas. Por lo tanto deberé conseguir un compromiso entre el número de polígonos y el realismo, intentando minimizar el primero maximizando todo lo posible el segundo.

Hay que tener en cuenta que el mapeado de texturas es una ilusión visual, ya que si el usuario se acerca demasiado al objeto descubrirá que se trata de una textura plana y no de un objeto geométrico 3D, pero a cierta distancia del objeto este aspecto apenas se aprecia.

Para utilizar texturas en OpenGL hay que seguir los siguientes pasos:

1. Habilitar el mapeado de texturas. Para ello se llama a la siguiente función:

   glEnable(GL_ TEXTURE_ 2D).

   Con el parámetro *GL_ TEXTURE_ 2D* dicho parámetro le indicamos que queremos utilizar texturas bidimensionales.

2. Especificar la imagen que se va a usar. Esto se consigue con la siguiente función:

   *void glTexImage2D(GLenum target, GLint level, GLint components, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid *pixels)*

   En los parámetros se debe especificar que es una textura 2D, el nivel de detalle de dicha textura, el número de componentes del color (en mi caso voy a utilizar texturas semitransparentes, luego tengo 4 componentes o lo que es lo mismo, RGBA), la anchura y altura de la imagen, el formato del valor de cada píxel (GL_RGBA), el tipo de datos usado para cada componente de valor de un píxel y un puntero al mapa de valores de los píxeles (la imagen en sí misma).

3. Mapear la textura. Hay que indicar, para cada vértice de un objeto, qué posición de la textura le corresponde. Para ello se usa la siguiente función:

   void glTexCoord2f(GLfloat s, GLfloat t)

Los parámetros 's' y 't' se corresponden con las coordenadas 'u' y 'v' e indican una posición sobre el mapa de la imagen. Esta función se debe anteponer a la que indica uno de los vértices del polígono, por ejemplo de la siguiente manera:

glTexCoord2f(1.0,1.0);

glVertex3f(1.0,1.0,0.0);

4. Indicar cómo se aplicará la textura a cada píxel

   En este apartado se indica qué hacer cuando se indica en las coordenadas de textura 's' y 't' un valor mayor que 1 o menor que 0. Hay dos posibilidades, la primera es repetir los píxeles de los bordes de la textura cuando se referencie fuera de ella, lo cual no parece que tenga mucha utilidad. La otra posibilidad es la de repetir la textura. Esto es, en lugar de tener un mapa con solo una imagen, se tiene un mapa donde la imagen de la textura está repetida infinitas veces, unas contiguas a las otras. Yo me inclinaré como es obvio por la segunda opción.

   Para indicar si se quiere repetir el borde de la textura, o se quiere repetir la textura completa se utiliza la siguiente función:

   void glTexParameterf(GLenum target, GLenum pname, GLfloat param)

   El primer parámetro debe valer *GL_ TEXTURE_ 2D*, el segundo puede ser *GL_ TEXTURE_ WRAP_ S* o *GL_ TEXTURE_ WRAP_ T* para indicar las coordenadas X e Y de la textura, y el tercer parámetro indica si queremos que se repita el borde (GL_CLAMP) o la textura completa (GL_REPEAT).

5. Otro aspecto a tener en cuenta es el filtrado de las texturas. Cuando la cámara se acerca demasiado a un objeto que posee texturas, se pueden notar la diferencia entre los píxeles contiguos de la textura, viéndose cuadros grandes. Cuando la cámara está lejos del objeto también pueden ocurrir efectos no deseados, como que aparezcan formas extrañas en las texturas debido a que sólo se dibujan algunos píxeles del dibujo. Para evitar de algún modo estos efectos se puede usar la siguiente función:

   void glTexParameterf(GLenum target, GLenum pname, GLfloat param)

   Donde el primer parámetro debe ser *GL_ TEXTURE_ 2D*, el segundo *GL_ TEXTURE_ MIN_ FILTER* o *GL_ TEXTURE_ MAG_ FILTER* para cuando la cámara está cerca o lejos, y el tercero puede valer *GL_ NEAREST* o *GL_ LINEAR*. Con el primero se indica que no se van a filtrar las texturas, y con el segundo que se va a hacer un filtrado lineal. Aplicar el filtrado es costoso en tiempo, pero evita el efecto de anti-aliasing.

En mi caso voy a utilizar imágenes BMP semitransparentes, en las que las zonas completamente negras no las visualizaré en escena, recortando así las partes que no quiera ver de la textura. Todo esto se consigue gracias al 'canal alpha'.

Así, a la hora de especificar el color de un píxel, se puede hacer con el modo RGB (glColor3f) o con el modo RGBA (glColor4f). Este cuarto componente (A) es el denominado 'canal alpha', y se le puede considerar como el grado de opacidad y el resto, RGB, el color en sí. De tal modo que *glColor4f(R,G,B,0.0f)* es completamente opaco y *glColor4f(R,G,B,1.0f)* completamente transparente. En mi caso voy a utilizar la técnica llamada *masking con test Alpha*. Este método se basa en un test que se hace sobre el valor de la componente alpha que estamos evaluando. Para ello elegimos una función y un valor de referencia y según el resultado de esta evaluación, aceptamos o rechazamos el fragmento que estamos procesando. Para activar este test se usa la siguiente instrucción:

glEnable(GL_ALPHA_TEST)

Seguidamente se debe especificar la función que se va a usar. Para ello utilizo la siguiente función:

void glAlphaFunc(GLenum funcion, GLclampf referencia)

Esta función establece el valor de referencia y la función de comparación para el test alpha. El valor de referencia está comprendido entre 0 y 1. La función puede tomar una serie de valores, y en mi caso he usado *GL_EQUAL*, para indicar que sólo voy a representar aquellos píxeles que tengan el valor alpha igual a 1.0.

Sin embargo, para usar esta función es necesario que la imagen posea un canal Alpha, y el formato BMP no lo posee. Por lo tanto se puede o bien usar otro formato, como el TGA, o añadirle a una imagen BMP cargada un canal Alpha. Yo lo haré de la segunda manera. De tal modo que he usado una función que carga los datos de un fichero BMP y va comparando si el color que está leyendo es negro (transparente en mi caso), y si sí es negro le añade a ese píxel un valor Alpha = 0 (transparente), y si no es negro le asigna Alpha = 1 (opaco), como ya he explicado.

Estas dos funciones (*glEnable(GL_ALPHA_TEST) y glAlphaFunc()*) sólo hay que llamarlas una vez, y en mi caso lo hago justo después de habilitar el mapeado de texturas con *glEnable(GL_TEXTURE_2D)*.

Existen además otras funciones que también son muy útiles para el tratamiento de texturas, como *glBindTexture*. y *glGenTextures*. Son útiles para gestionar texturas cuando se van a utilizar

un gran número de ellas, ya que permite asignar un 'nombre' a la textura cuando se carga, y en el momento de aplicarla evitamos llamar de nuevo a la función *glTexImage2D*, y con sólo llamar a *glBindTexture* y pasarle el nombre de la textura a activar será suficiente.

Como último apunte sobre el manejo de texturas en OpenGL, comentar que las dimensiones de la imagen debe ser siempre potencia de 2 tanto en anchura como en altura, pudiendo ser de forma rectangular.

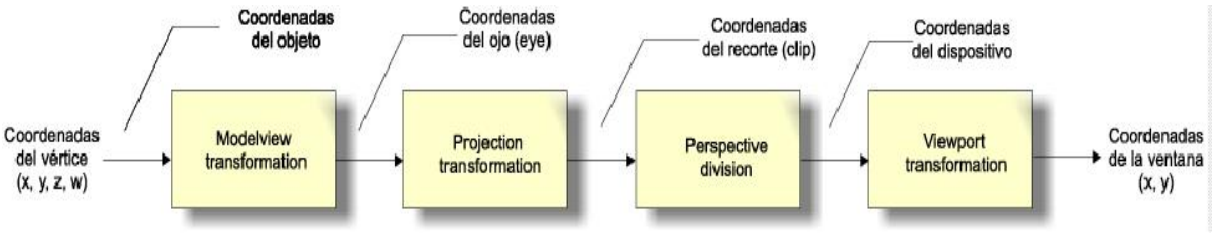
3.7. Clase fuente

Esta clase se encarga de escribir texto en pantalla. Pero una tarea que en principio parece sencilla no lo tanto ya que en mi caso yo utilizo esta clase para escribir texto durante la visita virtual, y como la cámara se puede desplazar y rotar, surge el problema de escribir de tal manera que el texto aparezca en una zona concreta de la pantalla, aunque se produzcan los citados movimientos de cámara.

Para que esto sea posible se pueden utilizar varios métodos. Yo en mi caso voy a jugar con las matrices de modelado y proyección que ofrece OpenGL. A continuación explico en qué consisten estas matrices, y qué hago con ellas.

3.7.1. Matrices en OpenGL: modelado y proyección

Desde que le damos a OpenGL unas coordenadas 3D hasta que aparecen en la pantalla, estas coordenadas son modificadas de la siguiente manera:



Las coordenadas 3D del vértice son coordenadas homogéneas, esto es, (x, y, z, w). De todas maneras, en la mayoría de los casos pondremos simplemente (x, y, z), puesto que, excepto en casos muy especiales, w=1. A estas coordenadas las llamaremos coordenadas del objeto, puesto que es el sistema de coordenadas en que se definen los objetos.

Ahora bien, hay que montar una escena, y podríamos querer situar el objeto en diferentes posiciones, ya sea para poner varios objetos del mismo tipo en la misma escena, como para realizar animaciones. Por otro lado, dependiendo de donde situemos la cámara, veremos los objetos

dispuestos de una u otra manera. Es por ello que existen las coordenadas del ojo. Para obtener las coordenadas del ojo a partir de las coordenadas del objeto, se realiza una transformación de modelo y vista, que consiste en la siguiente operación:

$$(x, y, z, w)^T_{ojo} = (x, y, z, w)^T_{objeto}.M$$

donde M recibe el nombre de matriz Modelview. En realidad, esta transformación aglutina dos: por un lado, la obtención de las coordenadas de la escena, (o coordenadas mundo), y por otro, la transformación que se realiza para situar el punto de vista. Por tanto, podríamos separar la matriz M en dos:

$$M = M_{escena}.M_{punto\_de\_vista}$$

Una vez tenemos las coordenadas de la escena definidas respecto a la cámara, se realiza la 'foto', que consiste en multiplicar las coordenadas por una matriz de proyección, que denominaremos P. Esta matriz realiza una transformación afín del espacio de la escena, de tal manera que el volumen de visualización (es decir, la parte de la escena que quedará 'fotografiada') se deforma hasta convertirse en un cubo que va de (-1, -1, -1) a (1, 1, 1). Por tanto, para seleccionar la parte de la escena que caerá en el volumen de visualización, bastará con ver si queda dentro de estos límites. Es por esto que a estas coordenadas se las denomina coordenadas de recorte o clipping:

$$(x, y, z, w)^T_{clipping} = (x, y, z, w)^T_{ojo}.P = (x, y, z, w)^T_{objeto}.M.P$$

Después, una vez realizado el recorte, se realiza la transformación de 'perspective division', que da como resultado coordenadas del dispositivo (coordenadas de ventana-mundo). Finalmente, se pasa de coordenadas de ventana-mundo a coordenadas de viewport (coordenadas de ventana-vista), mediante la llamada transformación del viewport. Resumiendo, para obtener una vista de una escena en nuestra pantalla tenemos que definir:

1. La matriz del Modelview M, que define la colocación de los objetos en la escena y el punto de vista.

2. La matriz de Proyección P, que define el tipo de proyección.

3. La posición del viewport en coordenadas de pantalla.

Las matrices M y P por defecto son iguales a la identidad, mientras que la posición del viewport es en la esquina superior izquierda y ocupando toda la pantalla (en este caso, la ventana de ejecución).

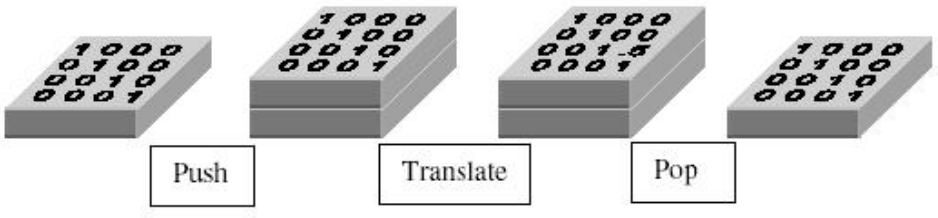
Como resumen a lo anterior se puede decir que OpenGL guarda la transformación de los objetos en una matriz. A esta matriz se le denomina matriz de modelado (MODELVIEW), Además de esta transformación, OpenGL posee otra matriz muy importante, que es la matriz de proyección, en la que se guarda la información relativa a la 'cámara' a través de la cual vamos a visualizar el mundo virtual.

Al realizar operaciones que modifiquen alguna de estas dos matrices, tendremos que cambiar el 'modo de matriz', para que las operaciones afecten a la matriz que nos interesa. Para ello utilizaremos las funciones glMatrixMode(GL_MODELVIEW) o glMatrixMode(GL_PROJECTION).

Además, existen dos funciones que permiten guardar y restaurar los valores de la matriz activa en una pila. La función glPushMatrix() guarda una matriz en la cima de la pila, y glPopMatrix() la extrae y la restaura. Esto se puede utilizar para dibujar un objeto y, antes de dibujar el siguiente, se restaura la transformación inicial. Por ejemplo:

```
<transformación común para la escena>

glPushMatrix();
<trasformación propia del elemento 1>
<dibujado del elemento 1>

glPopMatrix(); // volvemos a la transformación común

glPushMatrix();
<transformación propia del elemento 2>
<dibujado del elemento 2>

glPopMatrix(); // volvemos a la transformación común
...
```

Por ejemplo en la siguiente imagen muestro el estado de la pila de matrices:



Como se puede ver en esta imagen, el efecto de la traslación sólo afectará a los objetos comprendidos entre el 'push' y el 'pop'.

3.7.2. Trabajando con la matriz de proyección

En mi caso, para mostrar el texto de tal modo que siempre esté orientado hacia la cámara, antes de escribirlo realizo lo siguiente:

```
glMatrixMode(GL_PROJECTION);

glPushMatrix();

glLoadIdentity();

glScalef(1, -1, 1);

glMatrixMode(GL_MODELVIEW);
```

Con este código en primer lugar elijo la matriz de proyección para trabajar con ella. Luego la guardo en la pila y cargo la matriz por defecto (identidad). A continuación invierto el eje Y y elijo la matriz de modelado, para escribir el texto. Con esto he evitado usar la matriz de proyección que estaba utilizando, que estaba desplazada y rotada por el movimiento de la cámara a través del mundo 3D. Para escribir el texto hago lo siguiente:

```
glRasterPos2f(x, y);

for (c=texto; *c!='\0'; c++)
   glutBitmapCharacter(fuente, *c);
```

Con la primera función sitúo el primer carácter donde deseo, y con la segunda voy 'dibujando' carácter a carácter en pantalla. Luego debo restaurar la matriz de proyección que había al principio (con la posición y rotación de la cámara), y elegir de nuevo la matriz de modelado para continuar trabajando sobre ella:

```
glMatrixMode(GL_PROJECTION);
```

```
glPopMatrix();

glMatrixMode(GL_MODELVIEW);
```

3.8. Clase cámara

Esta clase contiene toda lo necesario para poder desplazarse a través del mundo virtual: movimiento de la cámara hacia delante y hacia atrás, también hacia los lados, rotación de la misma con el ratón e incluso se puede volar por el mundo 3D. Es una de las partes más importantes, ya que es lo que proporciona la 'jugabilidad' a la visita. A continuación explico algunos de los aspectos más importantes.

3.8.1. Posicionando la cámara

Para trabajar con una cámara en OpenGL tenemos que definir no sólo su posición, sino también hacia dónde mira y con qué orientación. Para hacer esto, basta con modificar la matriz 'ModelView' para mover toda la escena de manera que parezca que hemos movido la cámara. El problema de este sistema es que tenemos que pensar bastante las transformaciones a aplicar. Es por ello que la librería GLU posee la función 'gluLookAt'. Su sintaxis es la siguiente:

void gluLookAt(posicionX, posicionY, posicionZ, vistaX, vistaY, vistaZ, orientacionX, orientacionY, orientacionZ)

donde 'posicion' corresponde a la posición de la cámara, 'vista' corresponde al punto hacia donde mira la cámara y 'orientacion' es un vector que define la orientación de la cámara. El vector 'orientacion' no puede ser paralelo al vector formado por 'posicion' y 'vista', es más, debería ser perpendicular. Si no, el resultado es impredecible.

Estos tres vectores (posición, vista y orientación) son atributos de la clase. De tal modo que para desplazarme por el mundo o rotar la cámara tendré que modificarlos como indico en los siguientes apartados.

3.8.2. Movimiento hacia delante y hacia atrás

Para desplazar la cámara hacia delante en primer lugar tengo que calcular el 'vector de vista', es decir, el vector hacia donde mira la cámara, para que el movimiento se realice en esa misma dirección y no en otra. Esto lo consigo restando al vector 'vista' el vector 'posición'.

A continuación sumo a las coordenadas de la posición y la vista el producto del vector calculado anteriormente por la velocidad de desplazamiento que he establecido. Si el movimiento es hacia delante la velocidad es positiva. Si no, la velocidad es negativa. En el caso de este movimiento, como se hace en horizontal, sólo se modifican las coordenadas X y Z, la coordenada Y la modifica cuando 'vuela' la cámara.

3.8.3. Movimiento lateral

La base es la misma que en el movimiento anterior, excepto la dirección de movimiento. En este caso el movimiento se realiza hacia los lados, por lo tanto se tiene que obtener el vector perpendicular al 'vector de vista'. Como es sabido que el producto vectorial de dos vectores nos da uno normal al plano definido por ambos, calculo el producto vectorial del 'vector de vista' (que lo calculo como antes dije) por el de 'orientación'. El primero está dirigido hacia donde mira la cámara, y el segundo hacia arriba en la coordenada Y, por lo tanto el vector normal saldrá lateralmente y perpendicular al plano definido por ambos.

3.8.4. Vuelo de la cámara

En este caso se modifica la coordenada Y de la posición y la vista: aumenta cuando desea elevarse y decrece cuando quiere descender. Aquí no hace falta calcular ningún vector de dirección, ya que el desplazamiento siempre es totalmente vertical.

3.8.5. Rotación de la cámara

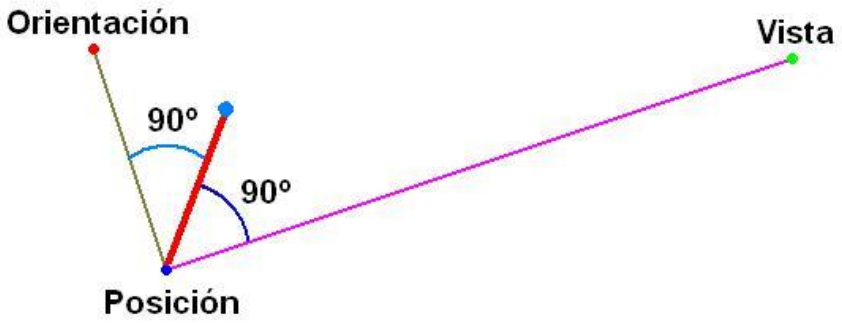
Como ya comenté en secciones anteriores, realizar una rotación respecto a un eje de coordenadas es relativamente sencillo. La cosa se complica cuando dicha rotación se realiza respecto a un eje arbitrario cualquiera. El fundamento matemático ya lo expliqué en su sección correspondiente. Aquí simplemente voy a comentar que lo que debo rotar, en mi caso, es justo el vector 'vista', es decir, hacia donde está mirando la cámara, ya que en una rotación de cámara la posición de la misma no cambia, ni tampoco la orientación, sólo el lugar hacia donde mira.

Por lo tanto todas las transformaciones se las tengo que aplicar al punto hacia donde mira la cámara. Pero como siempre esto no es tan sencillo como parece, ya que con el ratón se pueden realizar dos tipos de rotaciones:

1. La primera rotación se realizará si el ratón de mueve a la derecha y a la izquierda. Esta rotación se hace respecto al vector 'orientación', de derecha a izquierda. A priori puede parecer que este eje coincidirá con el eje Y, ya que al principio está totalmente vertical. Sin embargo, como se puede observar en la imagen posterior, mediante las rotaciones del ratón se puede inclinar, y una vez se haya modificado, será casi imposible que vuelva a coincidir

de nuevo con el eje Y. Por lo tanto, en este caso debo hacer una rotación respecto a un eje que no coincide con uno de los 3 ejes de coordenadas.

2. La segunda rotación es algo más compleja, se produce cuando el ratón se mueve arriba o abajo. El eje de rotación será uno perpendicular al plano definido por el vector 'orientación' y el 'vector de vista' (calculado como expliqué anteriormente). Por tanto se deberá rotar respecto a un vector que se obtenga con el producto vectorial de ambos vectores. En la siguiente imagen se aprecia este eje de rotación: como se puede observar, forma 90º con respecto a los ejes de 'orientación' y de 'vista', es decir, es perpendicular al plano que forman ambos vectores (desde el punto de vista de la perspectiva, se 'aleja' en el eje Z de la imagen). Ese eje de rotación era el mismo que calculaba a la hora de realizar los desplazamientos laterales.



Antes de realizar las rotaciones debo comprobar si realmente se ha desplazado el ratón. Para ello siempre guardo la posición del cursor en el fotograma anterior, de tal modo que si ésta varía se realiza una rotación.

En cada frame de la animación de la visita virtual, se comprueba si hay movimiento de la cámara (es decir si se ha pulsado alguna de las teclas de movimiento) o si hay rotación (se ha desplazado el ratón), y a continuación calculo el número de frames por segundo, como muestro en la sección siguiente.

3.8.6. Frames por segundo

El número de frames por segundo es muy importante a la hora de determinar el rendimiento de una aplicación 3D como es este caso. Es por ello por lo que lo que lo muestro en pantalla. Para realizar dicho cálculo, voy calculando el tiempo en milisegundos transcurrido desde que comenzó la visita virtual con una función de SDL llamada 'SDL_GetTicks' y en cada fotograma aumento el número de frames en una unidad. Cuando ha transcurrido 1 segundo completo, actualizo el número de frames por segundo e inicializo el contador.

3.8.7. Detección de colisiones

Como ya comenté anteriormente, sólo voy a realizar la detección de colisiones con las paredes del invernadero, ya que si tuviera en cuenta otros objetos 3D como por ejemplo los pilares, se entorpecería la visita virtual. Los pasos que sigo son los siguientes:

1. La detección de colisiones sólo la realizo cuando la cámara se desplaza, bien sea hacia delante, hacia atrás, o lateralmente, y no la tengo en cuenta cuando vuela o cuando rota (que no se desplaza). Por lo tanto en primer lugar desplazo la cámara hacia el lugar correspondiente, sin tener en cuenta si colisiona o no. Aunque desplazo la cámara, no dibujo todavía la escena, de tal modo que si hubiera colisión, regresaría la cámara en su posición original y luego dibujaría, de tal modo que parezca que no se ha desplazado.

2. Para detectar la colisión voy comprobando, como ya expliqué en capítulos anteriores, si la esfera imaginaria de la cámara colisiona con cada uno de los 2 triángulos por los que está formado cada una de las paredes.

3. Si existe colisión con alguna pared, simplemente deshago el movimiento que había realizado de tal modo que, como aún no lo había dibujado en escena, pues parece que la cámara no se ha desplazado. Si no hay colisión simplemente dejo la cámara en su nueva posición, ya que al no chocar con ninguna pared ese movimiento sí está permitido.

3.9. Clase esfera

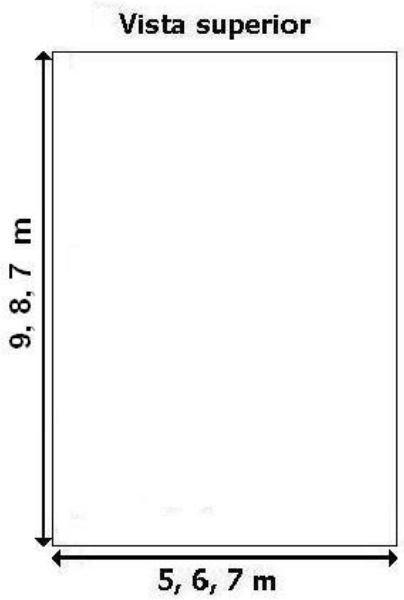
Esta clase es la que implementa la detección de colisiones propiamente dicha. Todo el fundamente matemático ya lo expliqué en su sección correspondiente, aquí simplemente la implemento.

3.10. Clase consola

Se encarga de almacenar los comandos que se introducen por teclado durante la visita virtual, y ejecutarlos. A través de estos comandos se pueden modificar algunas características del mundo 3D, y es realmente el elemento que hace más dinámica la aplicación.

Algunos aspectos que puede modificarse son:

- Distancia entre pilares: esta distancia se puede modificar en los dos lados de cada módulo principal de invernadero, como se puede ver en la siguiente imagen:



El lateral más ancho tiene por defecto una distancia entre pilares de 9 metros, mientras que el más corto tiene una distancia por defecto de 5 metros. El lateral más largo puede 'encogerse' a 8 y 7 metros, y el lateral más corto puede 'alargarse' a 6 y 7 metros, por lo que se puede dar lugar a un cuadrado, cuando ambos laterales tengan 7 metros de longitud. Existen por tanto 6 comandos que modifican dichas distancias: 3 comandos para cada uno de los 2 laterales.

- Cambiar la cubierta: se puede cambiar el exterior que cubre el invernadero (desde el punto de vista del programa es cambiar la textura que cubre los polígonos externos). Es decir, la cubierta puede ser de policarbonato o de plástico.

- Activar o desactivar la niebla: también se puede activar efectos de niebla, para embellecer la escena o dar un poco más de realismo.

- Y muchos más comandos que comentaré en el capítulo 4, en la sección del manual de usuario.

3.11. Clase Engine3D

Esta clase es la principal, y es la que contiene y utiliza al resto de clases del proyecto instanciadas como objetos. Un engine 3D es un motor gráfico tridimensional capaz de dibujar objetos virtuales representando sus polígonos y texturas, y que además incorpora una serie de características que intentan asemejar ese mundo virtual al propio mundo real: detección de colisiones, iluminación, movimiento de una cámara a través del mundo simulando la visión de una persona,

etc...

Mi engine 3D incorpora estas características, y aunque podría haber incorporado muchas más, he añadido sólo las que me interesaban para hacer de la visita virtual algo atractivo. Por ejemplo, como bien se sabe en el mundo real una persona no puede volar, y aunque en mi caso he podido 'ofender' a las leyes de la gravedad por su ausencia en mi engine, he visto oportuno incorporar un movimiento de vuelo de la cámara a través del mundo, para poder apreciar desde cualquier ángulo, altura y perspectiva posible el invernadero creado. Es un aspecto que me pensé que sería interesante añadir, y de hecho lo he añadido.

Aunque el propósito principal de esta documentación no ni mucho menos centrarme en el código o las funciones de OpenGL y SDL, sí voy a dedicar un par de apartados a explicar lo que considero más importante. En las secciones siguientes comento algunos de estos aspectos.

3.11.1. Inicializando OpenGL y SDL

La primera función a la que debemos llamar en el *main* es *SDL_Init*, que se encarga de inicializar SDL, y se debe llamar antes de ejecutar cualquier función SDL. Esta función posee un parámetro, como se muestra a continuación:

int SDL_Init(Uint32 flags)

Dicho parámetro indica qué parte de SDL se desea inicializar. En mi caso, como quiero inicializar el vídeo, el parámetro será *SDL_INIT_VIDEO*.

En segundo lugar se configuran las propiedades del formato del píxel de la superficie de dibujo de OpenGL. Dicha superficie es la zona dibujable de una ventana SDL. Tenemos que establecer las propiedades para dicha superficie. En mi caso he usado las siguientes características: *SDL_OPENGL* (ya que es una ventana de OpenGL), *SDL_HWPALETTE* (acceso exclusivo a la paleta de colores hardware) y *SDL_FULLSCREEN* (la ventana se abre a pantalla completa). Además con la función *SDL_GetVideoInfo* compruebo si se pueden añadir otras características, como la posibilidad de *blitting* del hardware que permite la copia y movimiento rápidos de secciones contiguas de memoria.

En tercer lugar uso la función *SDL_GL_SetAttribute* que posee dos parámetros:

int SDL_GL_SetAttribute(SDL_GLattr attr, int value)

Esta función asigna al atributo de OpenGL *attr* el valor *value*. Estos atributos sólo tendrán efecto cuando se llame a la función *SDL_SetVideoMode*, que configura un modo de vídeo con la

anchura, altura y bits por píxel especificados como parámetros, y además se le pasan todas las características del formato del píxel anteriores:

*SDL_Surface *SDL_SetVideoMode(int width, int height, int bpp, Uint32 flags)*

En cuarto lugar inicializamos OpenGL, y para ello llamo a las dos funciones siguientes:

glEnable(GL_DEPTH_TEST): activar el test de profundidad *SizeOpenGLScreen(int width, int height)*: configura el tamaño del viewport para OpenGL e inicializa la ventana.

En quinto lugar llamamos a una serie de funciones relacionadas con el manejo del teclado y el ratón:

SDL_WM_GrabInput(SDL_GRAB_ON): enlaza el ratón con la aplicación actual, y además todas las entradas de teclado se pasan directamente a la aplicación.

SDL_ShowCursor(SDL_DISABLE): oculta el cursor del ratón, para que no se muestre en pantalla.

SDL_EnableKeyRepeat(int delay, int interval): controla la repetición del teclado, el parámetro 'delay' establece cuánto tiempo debe estar presionada la tecla antes de que empiece a repetirse, y entonces se repite a la velocidad especificada en el parámetro 'interval'.

3.11.2. Manejo de eventos de teclado

Una vez se ha inicializado OpenGL y SDL, el programa se introduce en un bucle que se repite hasta la salida del mismo. En dicho bucle uso la función *SDL_PollEvent* para comprobar si se ha producido un evento. Los eventos que voy a tener en cuenta son los siguientes:

1. SDL_KEYDOWN: el usuario ha presionado una tecla.
2. SDL_KEYUP: el usuario ha soltado una tecla.
3. SDL_QUIT: el usuario desea salir del programa.

Para cada evento he creado una función que lo maneja. Así, cuando el usuario presiona una tecla, se comprueba qué tecla fue la que pulsó, si fue la tecla escape (SDLK_ESCAPE), alguna de movimiento (SDLK_UP, SDLK_DOWN, etc). Cuando suelta dicha tecla también se comprueba qué tecla ha sido la que ha producido el evento, del mismo modo que en la función anterior.

Además, en cada iteración del bucle se llama a la función que he programado para renderizar la escena actual. Esa escena puede ser o bien la interfaz gráfica del principio, o bien la visita virtual.

3.11.3. Visita virtual

En el siguiente código se ve la parte principal que implementa la visita virtual:

```
// Mientras no se pulse \textquoteleft Escape'
while(!finalizado)
{
    // Captura evento
    while(SDL_PollEvent(&event))
    {
        // Comprueba tipo de evento
        switch(event.type)
        {
            // Finaliza
            case SDL_QUIT:
              finalizado=true;
              break;

            // Pulsa tecla
            case SDL_KEYDOWN:
              teclaAbajo(&event.key.keysym);
              break;

            // Suelta tecla
            case SDL_KEYUP:
              teclaArriba (&event.key.keysym);
              break;
        }
    }

    // Actualiza la cámara
    camara.actualizar(Interfaz.lista,consola.tamX,consola.tamZ);

    // Dibuja la escena
    dibujarEscena();
}
```

Como se puede ver la visita finaliza cuando se pulsa la tecla 'ESC'. Este código está continuamente capturando eventos de teclado con la función 'SDL_PollEvent' y luego realiza diversas opciones dependiendo de evento recogido. Los eventos de teclado pueden ser de dos tipos: cuando se pulsa una tecla y cuando se suelta. Yo distingo ambos eventos:

1. Pulsa la tecla: existen unas cuantas teclas predefinidas que realizan una acción determinada con sólo pulsarlas:

   - Las teclas direccionales (arriba, abajo, izquierda y derecha) provocan que la cámara se desplace por el mundo 3D.
   - La tecla 'q' eleva la posición de la cámara (asciende en el vuelo).
   - La tecla 'w' disminuye la posición de la cámara (desciende en el vuelo).
   - Cualquier otra tecla es insertada como parte del comando de la consola. Para borrarlas simplemente pulsar la tecla RETROCESO. Cuando se pulsa ENTER se comparan los caracteres existentes con la lista de comandos permitidos. Si se trata de un comando se ejecuta la acción correspondiente, y si no se borran todos los caracteres de la consola de texto.

2. Suelta la tecla: este evento sólo lo tengo en cuenta con la teclas especiales de movimiento de cámara explicadas anteriormente. Como comentario añadir que, como dicho movimiento se realiza por la cámara en sí, los atributos que guardan el estado actual de las teclas especiales pertenecen a la clase cámara, porque lo he visto conveniente ya que influyen en su comportamiento, en lugar de pertenecer a la clase 'Engine3D'.

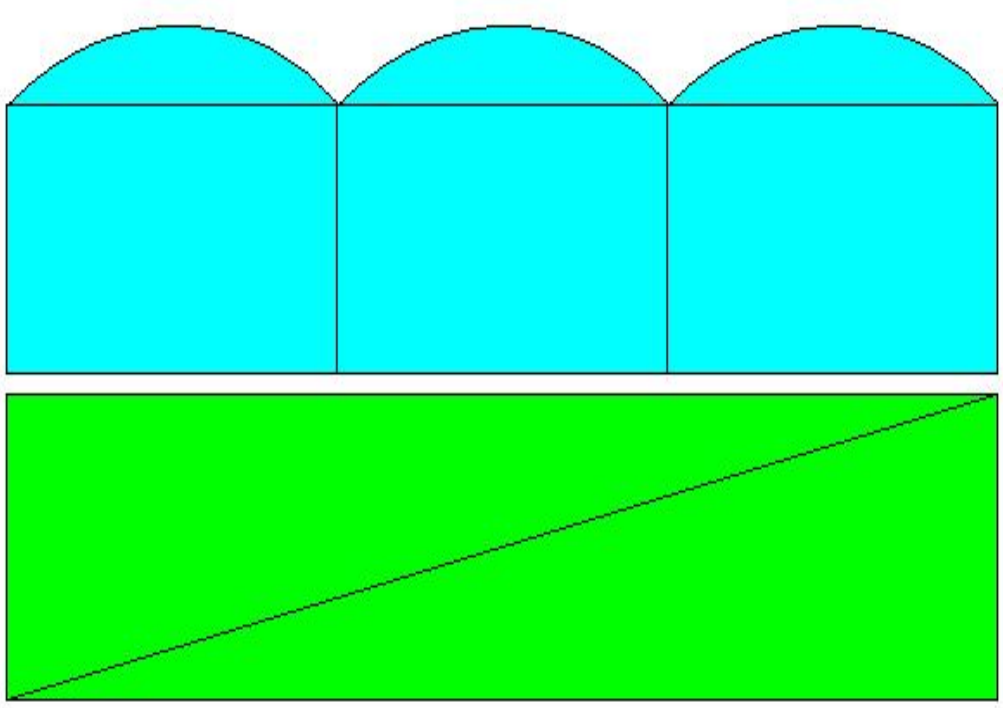
A continuación, como se puede ver en el código fuente, se actualiza la cámara, es decir, realizo en la cámara las acciones pertinenetes de acuerdo con los eventos capturados. Además los eventos de ratón los capturo en la propia clase 'Camara', para comprobar las rotaciones de la misma.

Y por último dibujo la escena, en ese método incluyo todo los elementos que constituyen el mundo virtual: objetos básicos, suelo, paredes, techos, consola de texto, logotipo de la universidad de Almería, etc...

3.11.4. Creación de paredes

La creación de paredes es otra de las funcionalidades de esta clase. Gracias al empleo de las texturas adecuadas conseguí disminuir enormemente el número de polígonos que poseían al principio las paredes. De tal modo que una pared simplemente es un rectángulo (2 triángulos en realidad), y posteriormente usando la coordenada de textura U (número de veces que la textura se repite horizontalmente) establezco tantas imágenes como sean necesarias.

La creación de paredes la realizo utilizando los puntos que marcó el usuario en la interfaz, de tal modo que creo planos rectángulos entre cada dos puntos (haciendo la correspondiente transformación de tamaño). Posteriormente a la hora de aplicar la textura es cuando determino cuántos módulos abarca dicha pared, o como dije en el párrafo anterior, el número de imágenes que debo poner consecutivas para que parezca que hay varios polígonos.



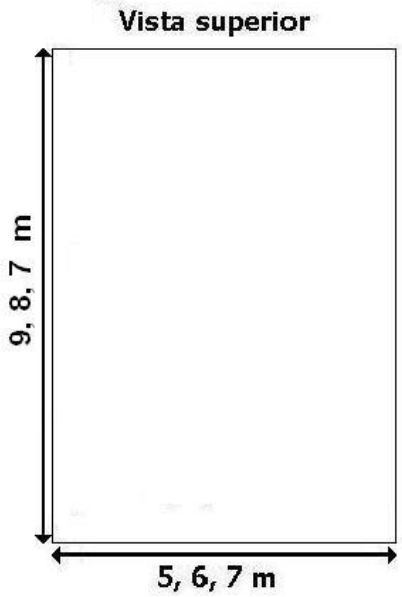
Por ejemplo en la imagen anterior se puede ver una pared creada por mí que abarca 3 filas de módulos básicos. Por lo tanto la coordenada de mapeado U en este caso deberá se 3, ya que necesito que se dibujen 3 texturas de pared consecutivas que den apariencia de continuidad en la imagen.

Para finalizar hago un comentario acerca de la forma curva que poseen las paredes en su parte superior. Esta forma, siguiendo mi filosofía de ahorra polígonos, la consigo gracias a una textura semitransparente, de tal forma que la parte superior la dibujo de negro (color transparente para mi programa) y así consigo una forma curva.

3.11.5. Modificación dinámica de la distancia entre pilares

Para modificar dinámicamente la distancia entre los pilares he utilizado una técnica basada en las matrices de escalado, es decir, antes de dibujar los objetos 3D en escena aplico una transformación de los mismos para variar sus tamaños en los ejes que me convenga. Como se

puede ver en la siguiente imagen ya utilizada con anterioridad, existen diferentes distancias posibles entre los pilares:



Como he tenido el cuidado de dibujar los módulos del invernadero siguiendo los ejes X y Z, para modificar por ejemplo la distancia del lado mayor (9, 8 y 7 metros) tendré que aplicar una matriz de escalado a todos los objetos de la escena en el eje X, de tal modo que parezca que dichos objetos se alargan o contraen.

Igualmente si deseo modificar la distancia del lado mayor (5,6 y 7 metros) tendré que aplicar una matriz de escalado al eje Z. El escalado consiste en multiplicar las coordenadas del objeto por una constante (puede que distinta para cada componente) para modificar el tamaño del mismo. La matriz de escalado se define como la siguiente:

$$M_e = \begin{pmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

No obstante la API de OpenGL incorpora entre sus múltiples funciones ya la de escalado. Para escalar toda una escena, antes de dibujarla y tras elegir la matriz de modelado, aplico la función 'glScaled' e introduzco en sus parámetros el valor por los que quiero multiplicar cada coordenada (es decir los valores 'sx', 'sy', 'sz' de la matriz de escalado).

Para cambiar la distancia entre los pilares durante la visita virtual por tanto sólo tengo que ver el tamaño actual y a continuación aplicar dicha función de tal modo que alargue o encoja los

objetos en el eje deseado.

3.11.6. Cambio de las cubiertas del invernadero

Este era otro de los requisitos del proyecto: que se pudiera modificar la cubierta que cubre la estructura del invernadero. En mi caso las dos cubiertas posibles son policarbonato y plástico, y para cambiar de una a otra hay que introducir y ejecutar el comando adecuado a través de la consola de texto.

Para modificar la cubierta lo que hago es cambiar la textura mapeada sobre los polígonos exteriores (techo y paredes), de tal modo que guardo variables internas de tipo 'booleano' que almacenan el estado actual del mundo, es decir, la textura que se está utilizando.

Estas variables son modificadas a través de la consola de texto, cuando el usuario elige una u otra cubierta. Cuando mi programa se dispone a dibujar estas estructuras externas comprueba el estado actual de dichas variables. Y dependiendo de su estado se mapeará una u otra textura, cargadas todas al inicio de la visita virtual.

Esta es otra de las partes del proyecto que creo que hubiera sido imposible desarrollar con la herramienta Blender, ya que no se tiene un control total de los objetos y texturas durante la visita virtual, sino que se representan los polígonos con la textura que se les asignó con anterioridad sin posibilidad de cambio dinámico.

Además gracias a la gran libertad que me han ofrecido Visual C++ con OpenGL he podido resolver esta función de una manera bastante sencilla y eficiente.

3.12. Clase listas

Esta clase es la que hace posible el dinamismo en la creación del invernadero. He agrupado en esta clase todos aquellos elementos dinámicos que posee el invernadero, como por ejemplo las paredes (no sabemos a priori el número de paredes que tendrá), los techos (tampoco se sabe), los pilares que quedan por rellenar, etc...

Por tanto en esta clase voy almacenando todos estos objetos para posteriormente dibujarlos en escena a través de la visita virtual. Para ello he usado listas enlazadas de objetos. Dichas listas enlazadas no voy a explicarlas porque no es el objetivo de esta documentación, es algo intrínseco a la programación.

3.13. Resumen

Aunque no he explicado toda la funcionalidad de mi código con detalle, he intentado abarcar lo más importante y obviar algunos aspectos menos relevantes relacionados con OpenGL o con el lenguaje de programación en sí. No obstante se puede inspeccionar el código fuente que adjunto como anexo digital ante cualquier duda.

Capítulo 4

Resultados y conclusiones

En este último capítulo me centro en los resultados que he obtenido de mi proyecto y los futuros trabajos que se pueden realizar. Lo he dividido en varias secciones:

1. *Objetivos cumplidos.* Detallo explícitamente los diferentes objetivos del anteproyecto y concluyo que los he cumplido todos.

2. *Funcionamiento del sistema y requisitos.* En esta sección me centro en los sistemas operativos que pueden ejecutar mi proyecto y los requisitos básicos.

3. *Experiencia aportada a nivel profesional.* He comentado los conocimientos que he adquirido con el desarrollo de este proyecto y la implicación a nivel profesional.

4. *Programas utilizados.* Aquí hago mención a los distintos programas que he usado para el desarrollo de este proyecto.

5. *Experimentos y resultados.* Visualizo unos cuantos ejemplos, incluyendo capturas de pantalla.

6. *Trabajos futuros y conclusiones.* Por último explico las diferentes conclusiones y el trabajo que se podría añadir el día de mañana.

4.1. Objetivos cumplidos

Para comenzar he integrado en una sola aplicación las dos funcionalidades básicas que se pedían: el diseño y la visita virtual del invernadero 3D, ya que he creído conveniente hacerlo así para evitar al usuario tener que estar utilizando varias aplicaciones a la vez. Como consecuencia no guardo el invernadero en disco, ya que no es necesario por estar almacenado en la memoria por parte del propio programa.

Además no sólo se pueden crear invernaderos de tipo 'regular', sino que también se pueden construir invernaderos 'irregulares' con ángulos entre sus paredes contiguas distintos a 90º y 270º. Por otro lado, en mi caso los elementos básicos los he creado con la herramienta Autocad y los he convertido al formato ASC.

Aunque en el anteproyecto indicaba que era en la propia interfaz donde se debían introducir los parámetros de modificación del invernadero (tipo de cubierta, distancias entre pilares, etc) lo he acabado incluyendo en la propia visita virtual, ya que es muchísimo más espectacular que el usuario pueda observar directamente los cambios que se producen para poder comparar visualmente. Esto me ha acarreado muchísimo más trabajo que si lo hubiera integrado simplemente en la interfaz inicial.

Para el desarrollo del engine 3D he utilizado OpenGL y además he incluido toda la funcionalidad que se pedía: movimiento de la cámara a través del mundo virtual, rotación con el ratón al purísimo estilo de los videojuegos tipo Quake, he implementado la detección de colisiones y además he proporcionado la justificación matemática correspondiente, y los objetos 3D pueden apreciarse con claridad y realismo gracias al uso de las texturas adecuadas.

En lo que respecta a la planificación a seguir, comentar que he seguido las pautas básicas que se indican. En primer lugar me he documentado debidamente sobre las tecnologías 3D, y una vez elegí la adecuada realicé la descomposición funcional del sistema (estas dos partes están incluidas en el primer capítulo de esta documentación). En los capítulos 2 y 3 he realizado el tercer paso: desarrollar las funciones obtenidas e integrarlas. Y finalmente he escrito esta documentación.

Por último quiero hacer una reflexión sobre la planificación temporal que realicé en el plan de proyecto del capítulo 1:

- El número de líneas de código que estimé (6667 con los puntos de función y 5599 con la estimación de líneas de código) resultó aproximado a lo que he acabado programando: aproximadamente 6000.

- La planificación temporal mostrada en el diagrama Gantt la he cumplido correctamente, es decir, he tardado medio año en realizar el proyecto desde el principio hasta el final. No obstante algunos problemas que inicialmente se presentaban complicados después resultaron ser menos complejos realmente, e inversamente surgieron muchos problemas que no contemplé al principio y que me han hecho perder mucho tiempo en resolverlos.

4.2. Funcionamiento del sistema y requisitos

Esta aplicación funciona correctamente en el sistema operativo Microsoft Windows, en todas las versiones que he probado: Windows 98 SE, Windows ME, Windows 2000, Windows NT y Windows XP.

Es necesario que se incluya los archivos 'glut32.dll' y 'sdl.dll' en el mismo directorio donde se encuentra el ejecutable del proyecto, en por defecto incluirlo en alguno de los directorios de Windows siguientes: SYSTEM o SYSTEM32.

En lo que respecta a la tarjeta gráfica, como es lógico cuanto mayor número de polígonos sea capaz de mover mayor rendimiento dará. En mi caso he desarrollado el proyecto con una tarjeta gráfica dentro de lo que cabe antigua: NVIDIA RIVA TNT2 Modelo 64, de 32 MB. Por eso el número de frames por segundo es reducido. Pero con una tarjeta con mayores prestaciones no tiene que haber problema alguno. De hecho gracias al esfuerzo dedicado a la reducción del número de polígonos, el rendimiento ofrecido por el programa es muy alto ya que no se ralentiza durante la visita virtual utilizando una tarjeta gráfica en condiciones.

4.3. Experiencia aportada a nivel profesional

Gracias a la realización de este proyecto he conseguido muchos objetivos profesionales, además de otros más personales:

- En primer lugar me he enfrentado a un problema de gran envergadura y he tenido que trabajar desde todos los puntos de vista del software: desde analista hasta programador. Y quizá lo más importante, he tenido que solucionar por mí mismo muchísimos problemas con los que me he enfrentado, y elegir tanto las tecnologías que he usado como los procedimientos.

- También he tenido que planificar mi propio trabajo, algo a lo que no estaba tan acostumbrado, ya que normalmente el trabajo viene determinado por las exigencias por parte de los profesores. En este caso he tenido que ser bastante exigente conmigo mismo.

- He aprendido a manejar a bastante profundidad herramientas tales como Visual C++ y OpenGL, punteras actualmente en el mundo de la informática gráfica. Con este proyecto he cubierto un vacío que creo que existe en la Ingeniería Informática, relacionado con el mundo gráfico 3D, ya que sólo hay una asignatura relacionada con este tema y tiene muy pocos créditos. Un tema tan interesante debería ofertarse en más asignaturas.

- Me ha permitido introducirme un poco en el mundo de la agricultura, mucho más complejo de lo que en un principio creía, y muy interesante. Me ha hecho ver que la informática hoy en día puede aplicarse a prácticamente cualquier área de estudio.

- Además desde el punto de vista personal este proyecto me ha aportado mucho conocimiento sobre el funcionamiento interno de un videojuego 3D, tema que siempre me ha interesado muchísimo y uno de los motivos que me llevó a estudiar esta carrera (me encantan los videojuegos, no lo puedo evitar).

4.4. Programas utilizados

En esta sección voy a mencionar los distintos programas y tecnologías que he usado para desarrollar el proyecto. Por un lado hay están los que he utilizado directamente:

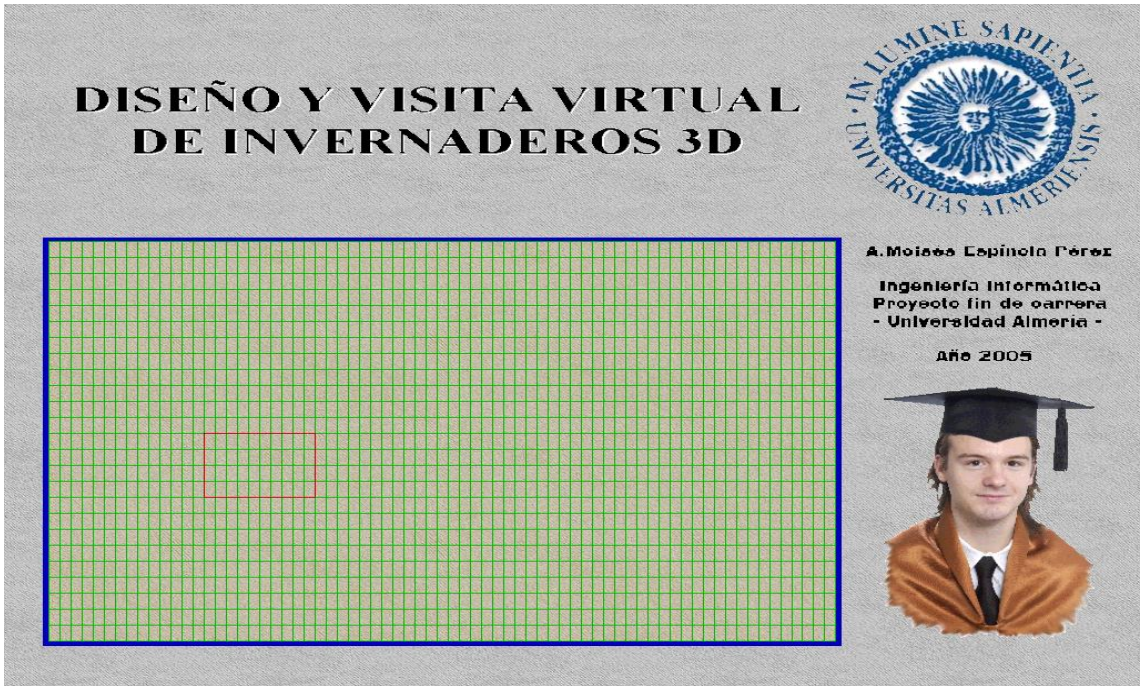
- *Microsoft Visual Studio 6.0*: en mi caso he utilizado el Visual C++ como lenguaje de programación básico.
- *OpenGL 1.4*: es la API que me ha ofrecido toda la funcionalidad necesaria para trabajar con gráficos 3D.
- *SDL*: librería auxiliar de ayuda a OpenGL que permite, entre otras cosas, la interacción con el teclado y ratón.
- *Autocad 2002*: para realizar los modelos 3D básicos que luego he utilizado en el engine 3D.
- *3DS Conversion Tool 1.0*: programa que convierte un modelo 3DS a formato ASC.
- *GIMP 2.2*: programa de edición gráfica con el que he construido las texturas.
- *COSMOS 4.1*: para hacer las estimaciones mediante puntos de función, COCOMO básico y COCOMO intermedio.
- *UML Studio 7.1*: con él he generado los diagramas en UML de mi código fuente.
- *Microsoft Project Professional 2002*: para realizar la planificación temporal del proyecto (el diagrama Gantt).
- *Miktex*: sistema TeX para Microsoft Windows de distribución gratuita, que incluye el programa 'pdflatex' con el que he elaborado esta documentación.
- Muchos *programas conversores* de formatos 3D, así como programas de visualización de objetos tridimensionales.

Otros programas y tecnologías que he estudiado, aunque finalmente no han participado en el proyecto, son los siguientes:

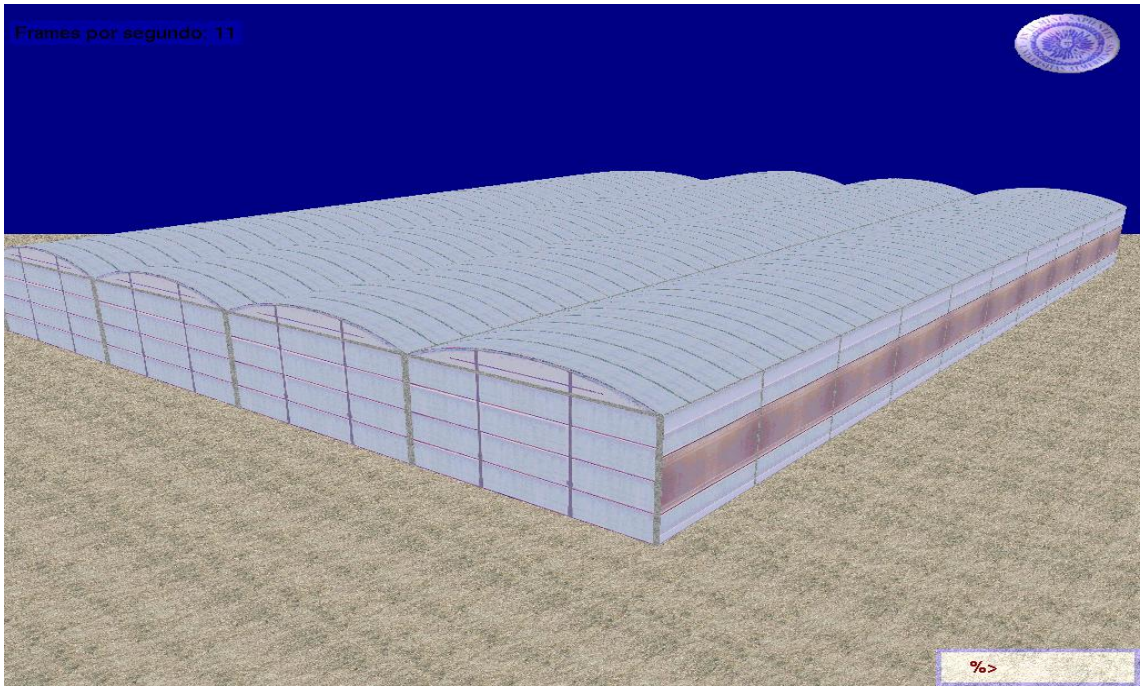
- Blender (y Phyton).
- 3D Studio.
- Blitz3D.
- Direct3D.
- VRML.

4.5. Experimentos y resultados

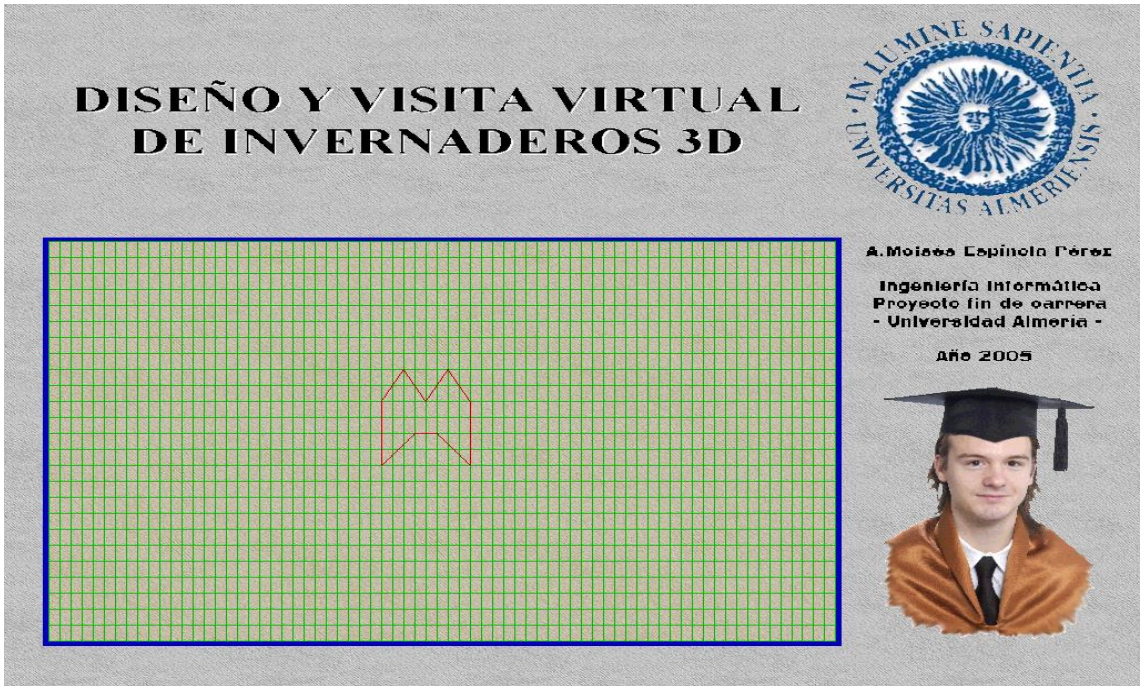
En esta sección voy a mostrar algunas capturas de pantalla que he visto importantes.



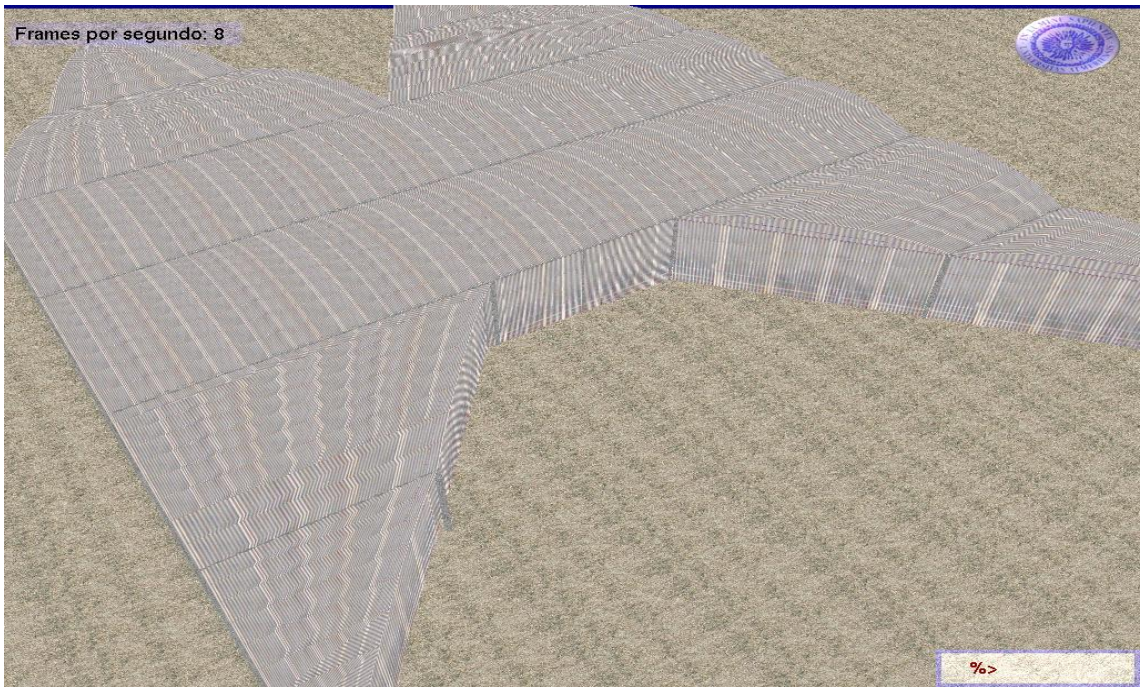
Dibujando un invernadero regular



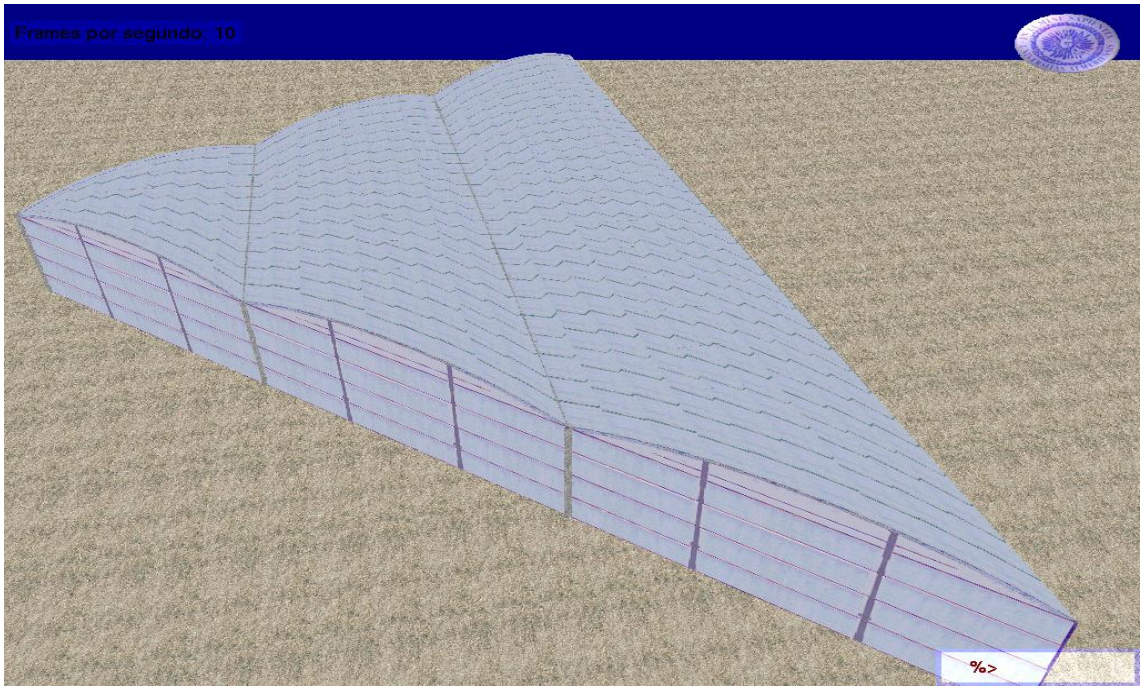
Invernadero regular



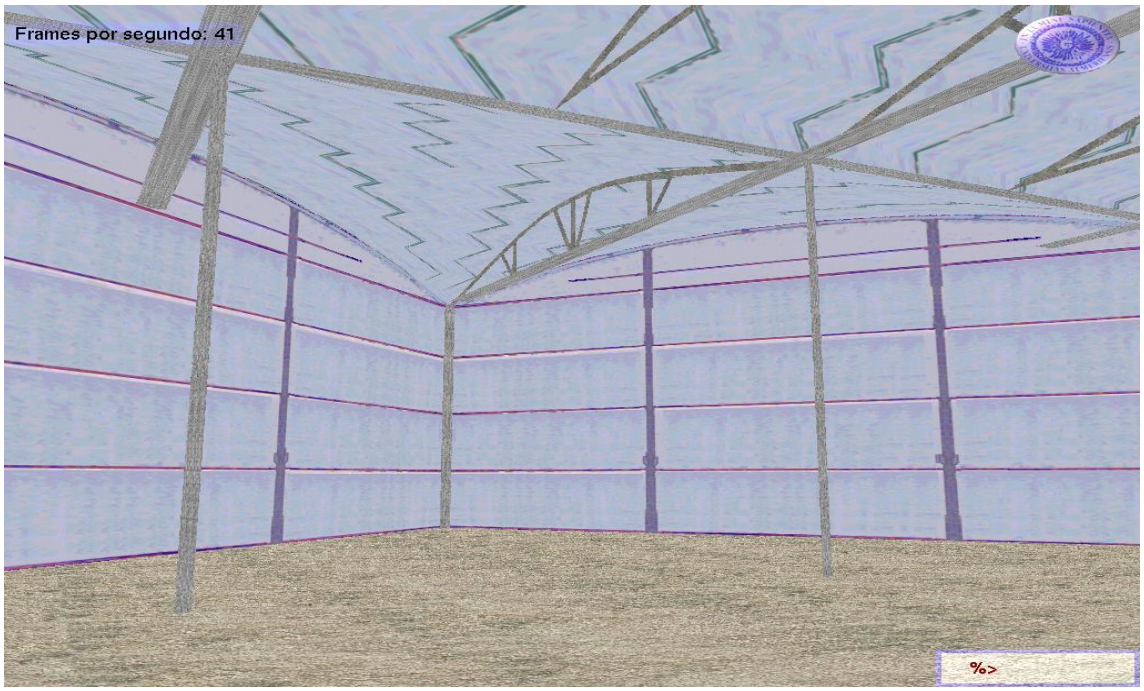
Dibujando un invernadero irregular



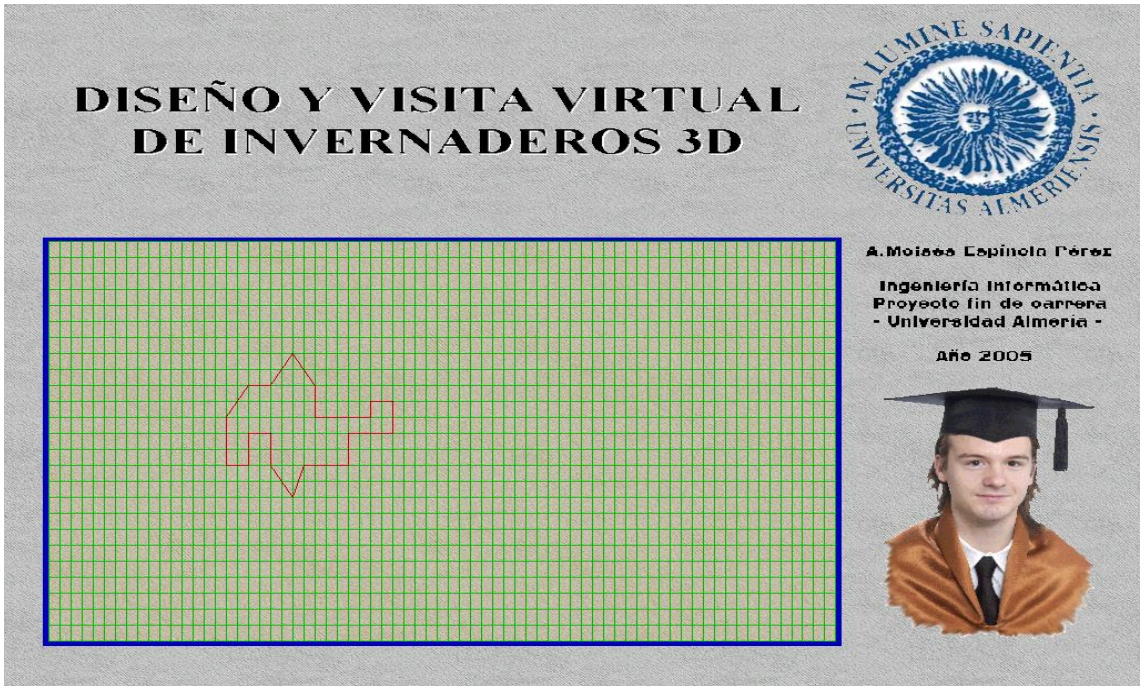
Invernadero irregular



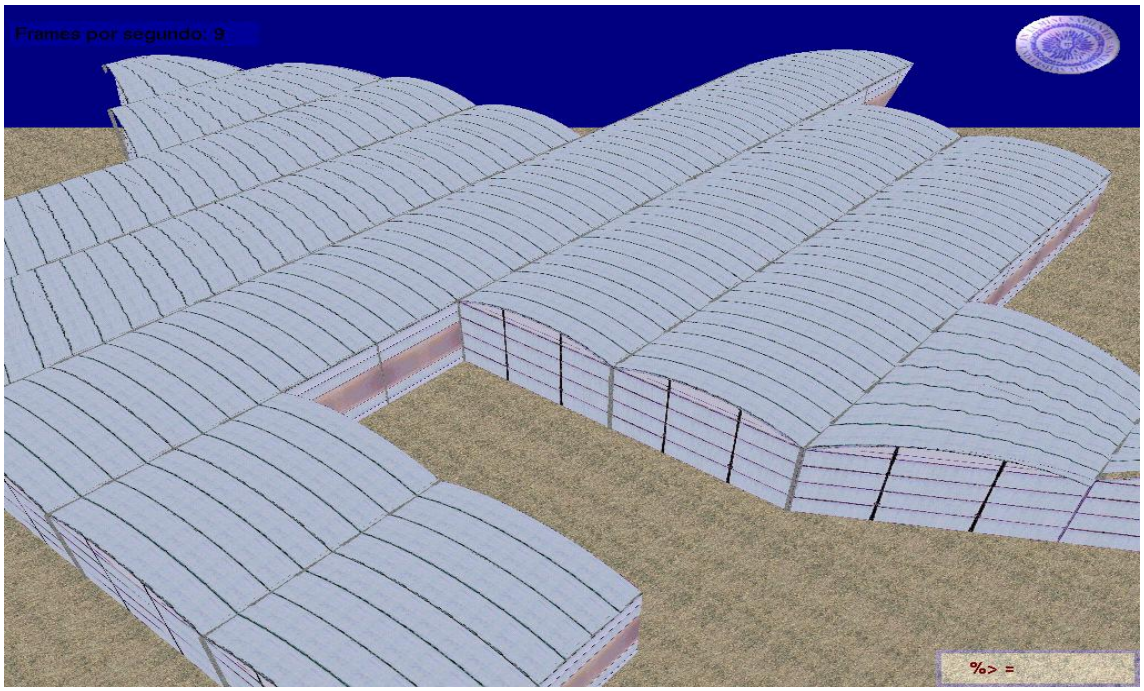
Invernadero con forma triangular



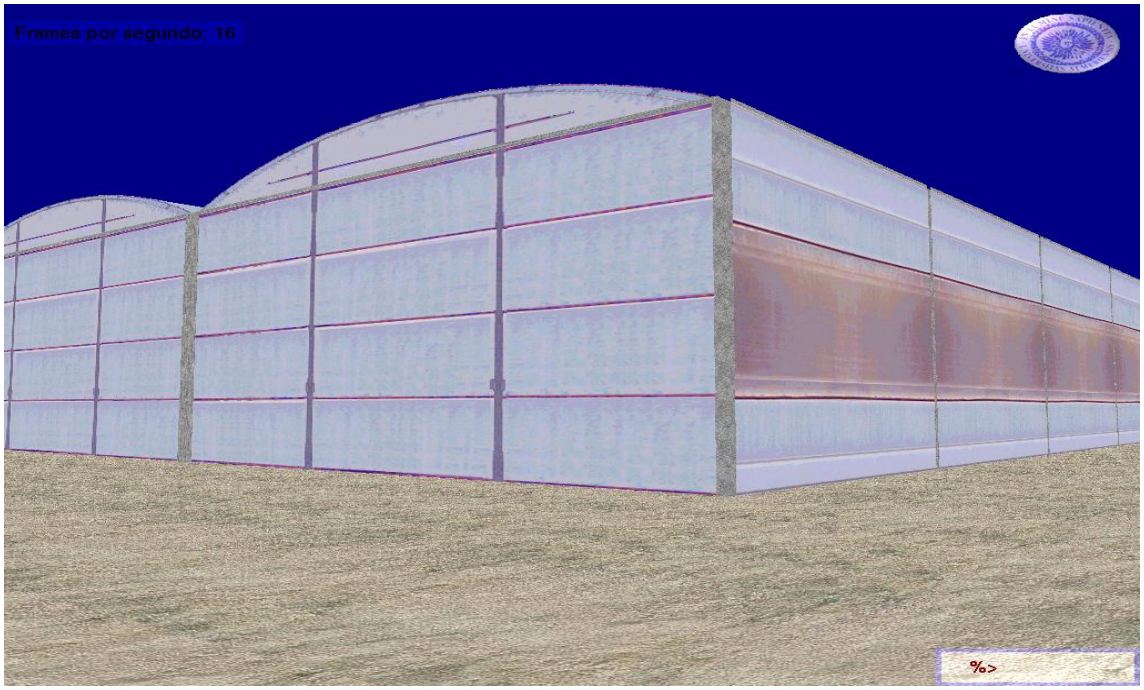
Esquina irregular del invernadero superior



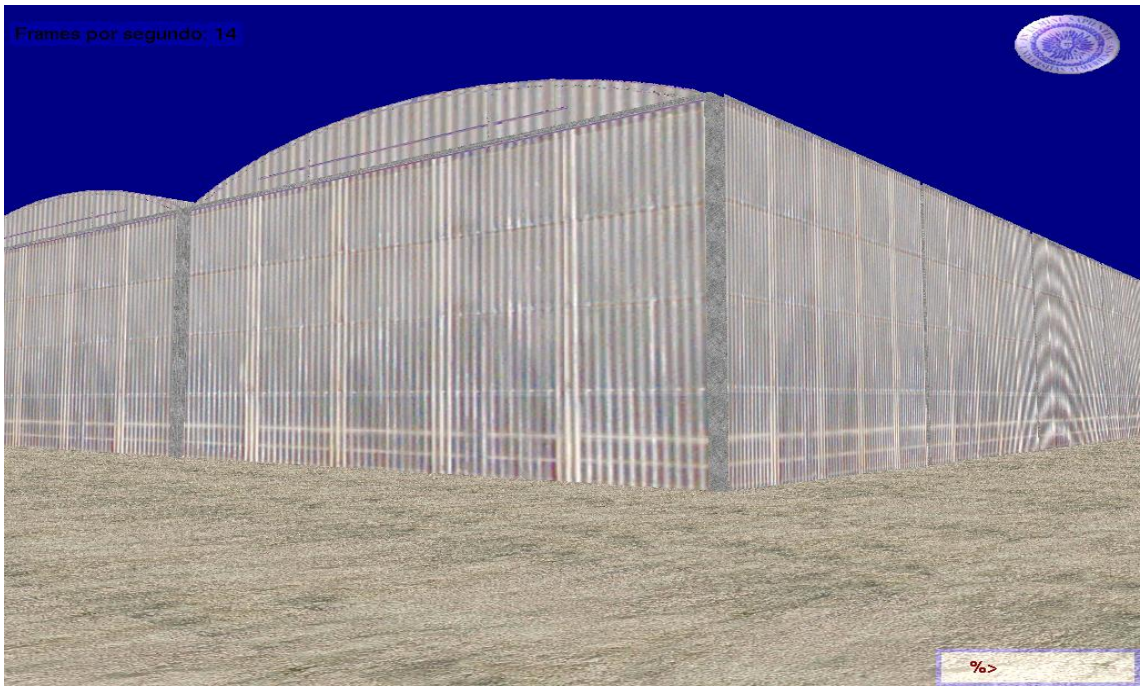
Dibujando formas muy extrañas



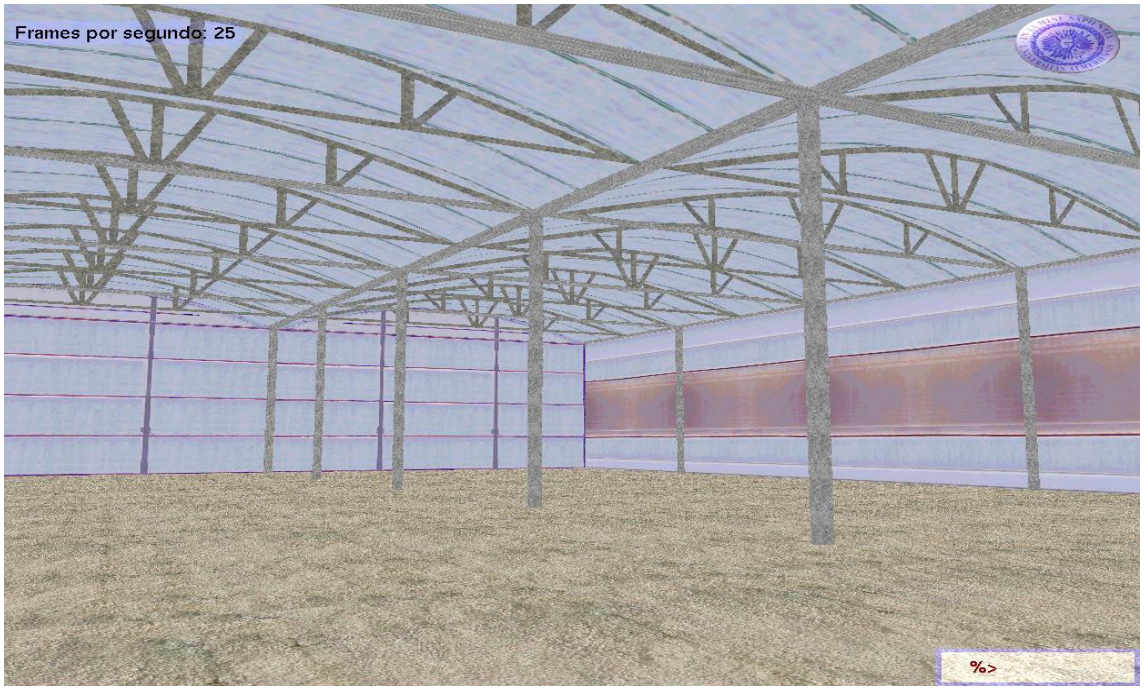
Invernadero muy irregular



Invernadero multitúnel de plástico (exterior)

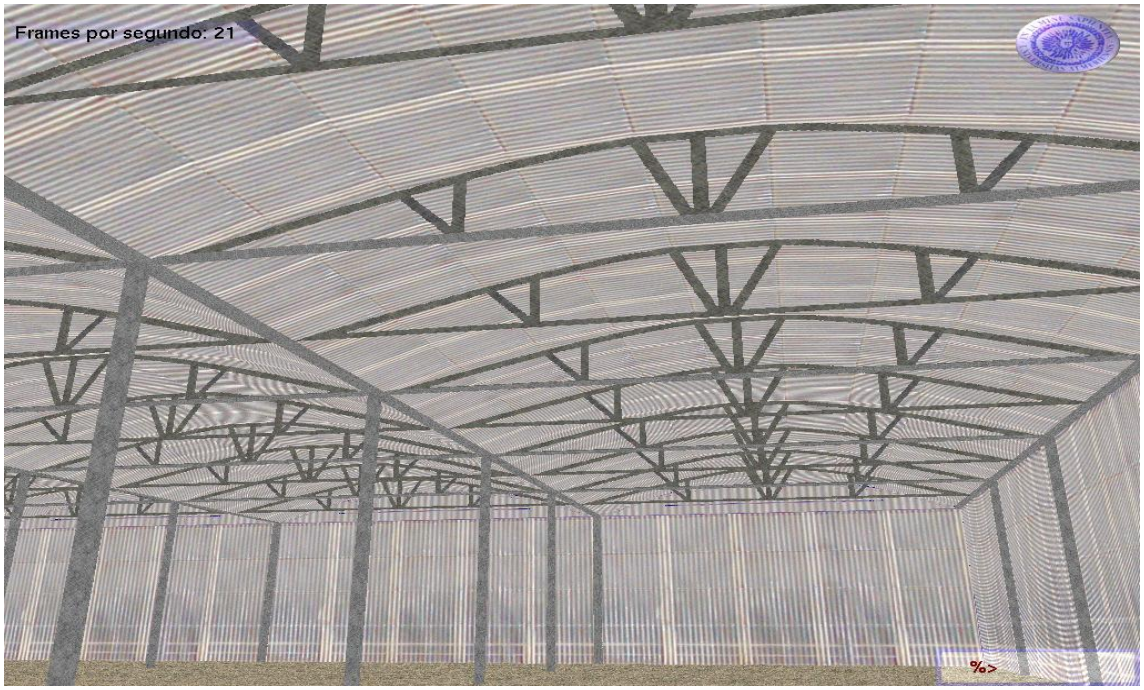


Invernadero multitúnel de policarbonato (exterior)

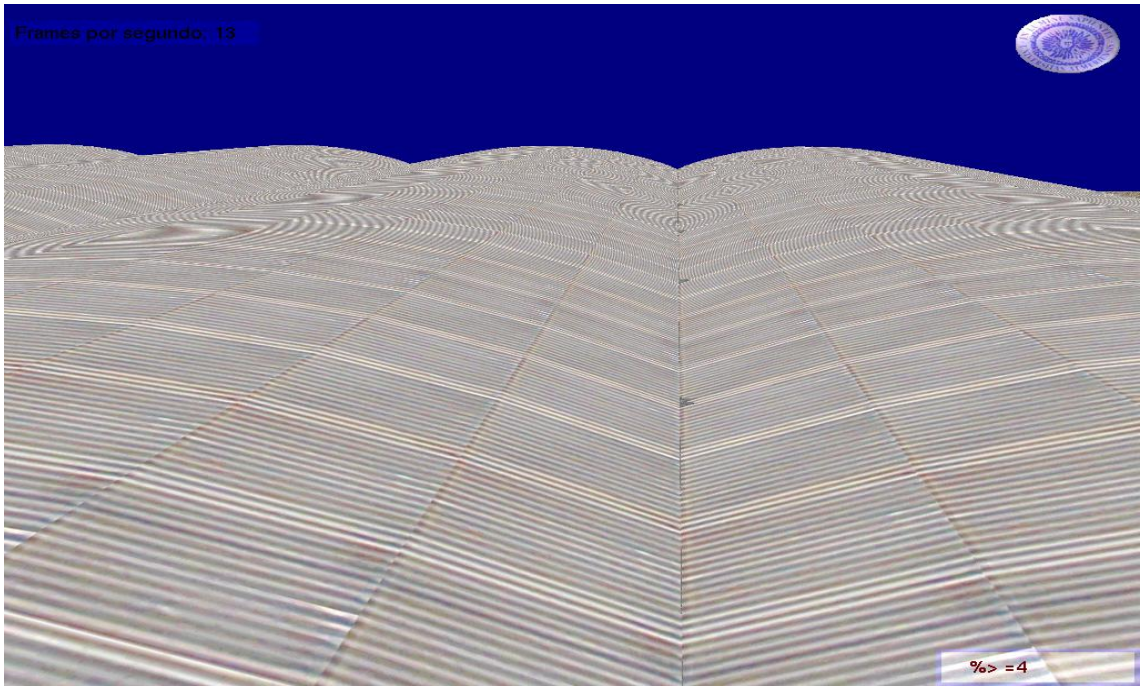


Invernadero multitúnel de plástico (interior)

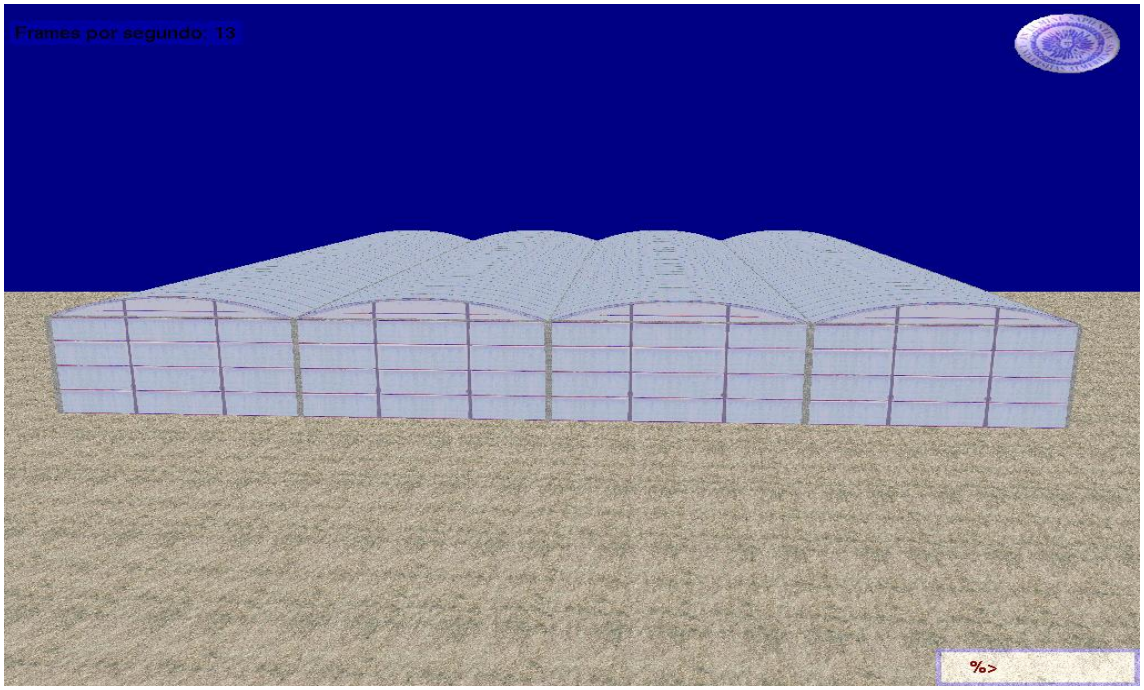
Invernadero multitúnel de policarbonato (interior)



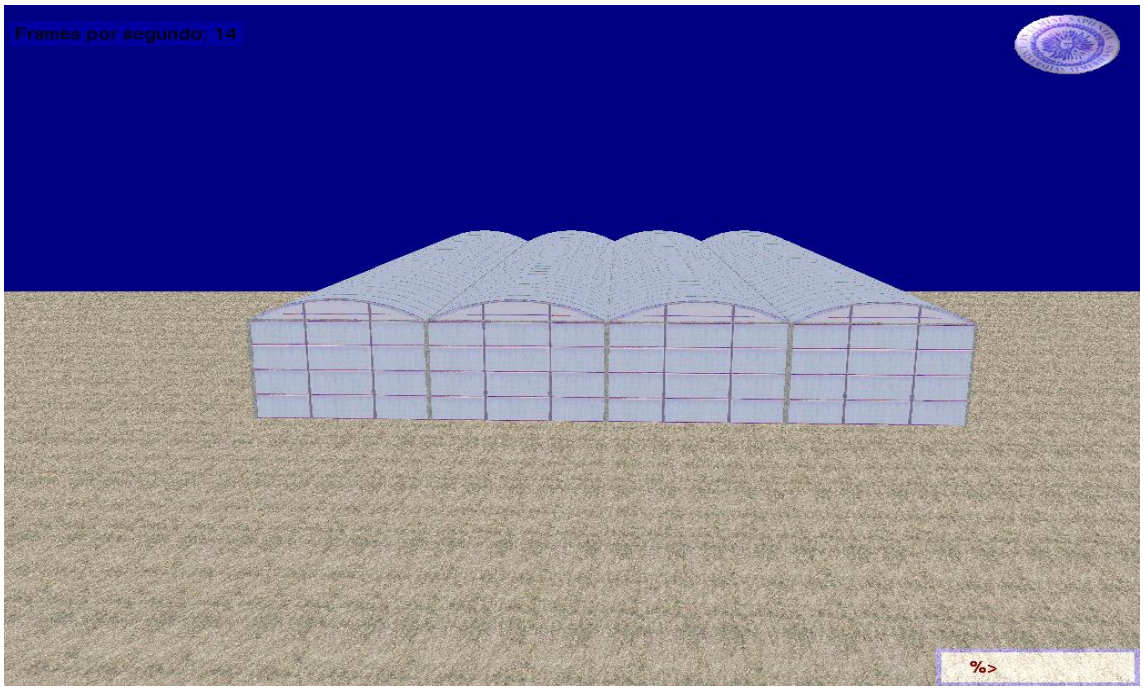
Techos desde el interior



Techos desde el exterior

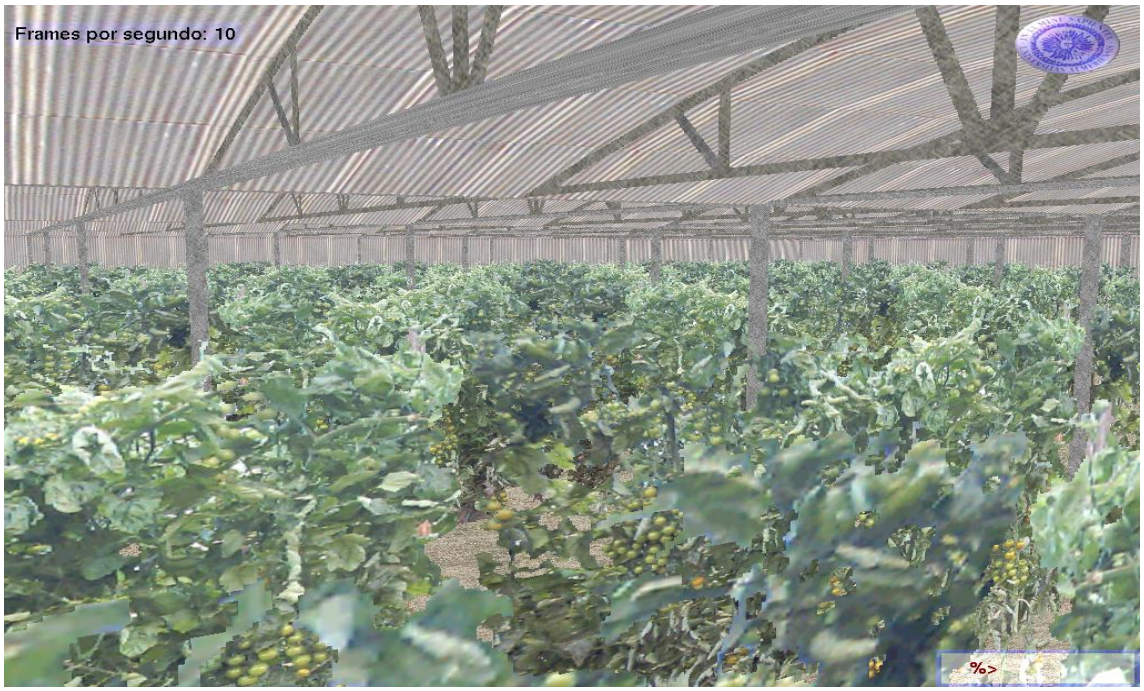


Distancia entre pilares de 9 metros (eje Z)

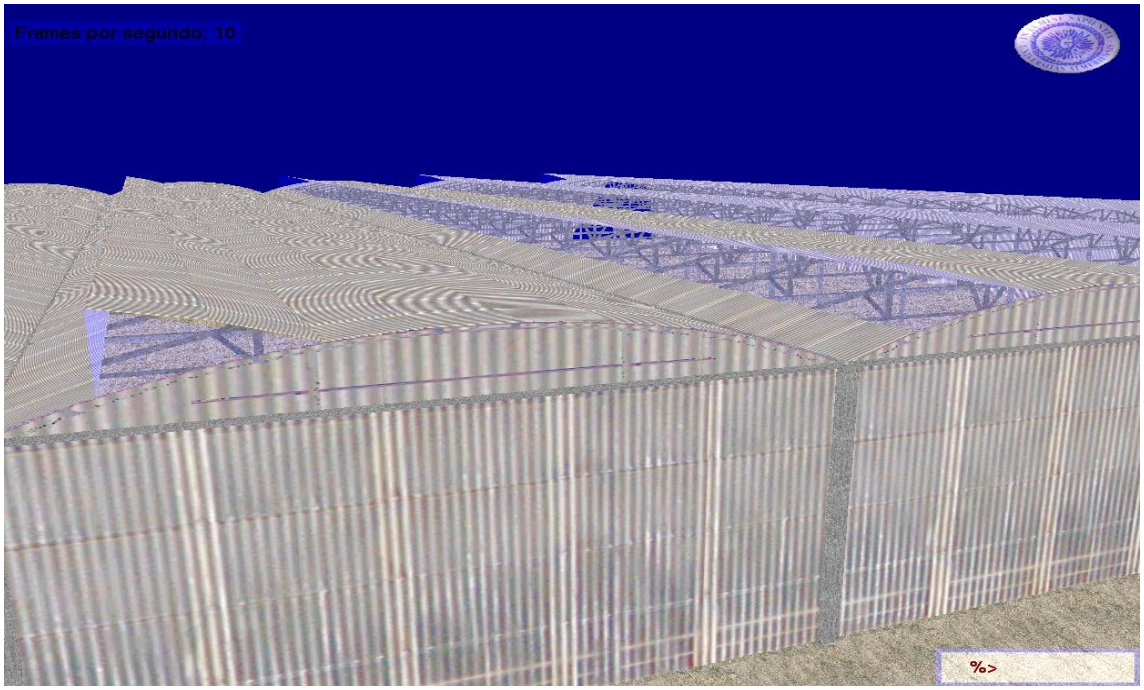


Distancia entre pilares de 7 metros (eje Z)

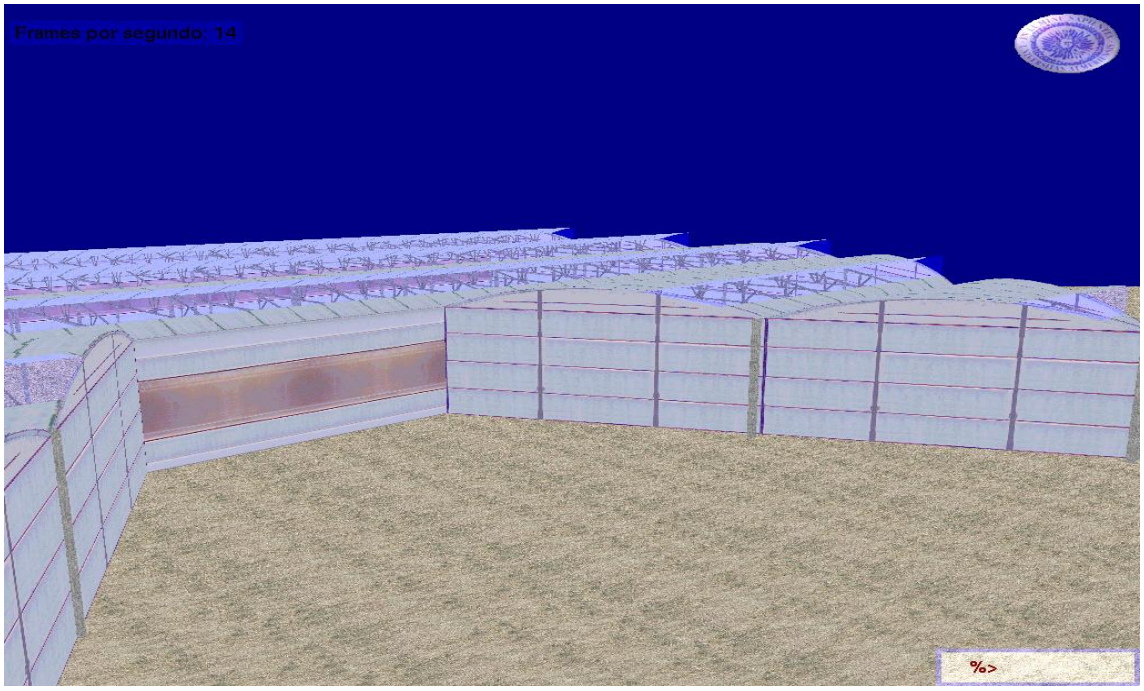
Invernadero con plantas



Invernadero con plantas

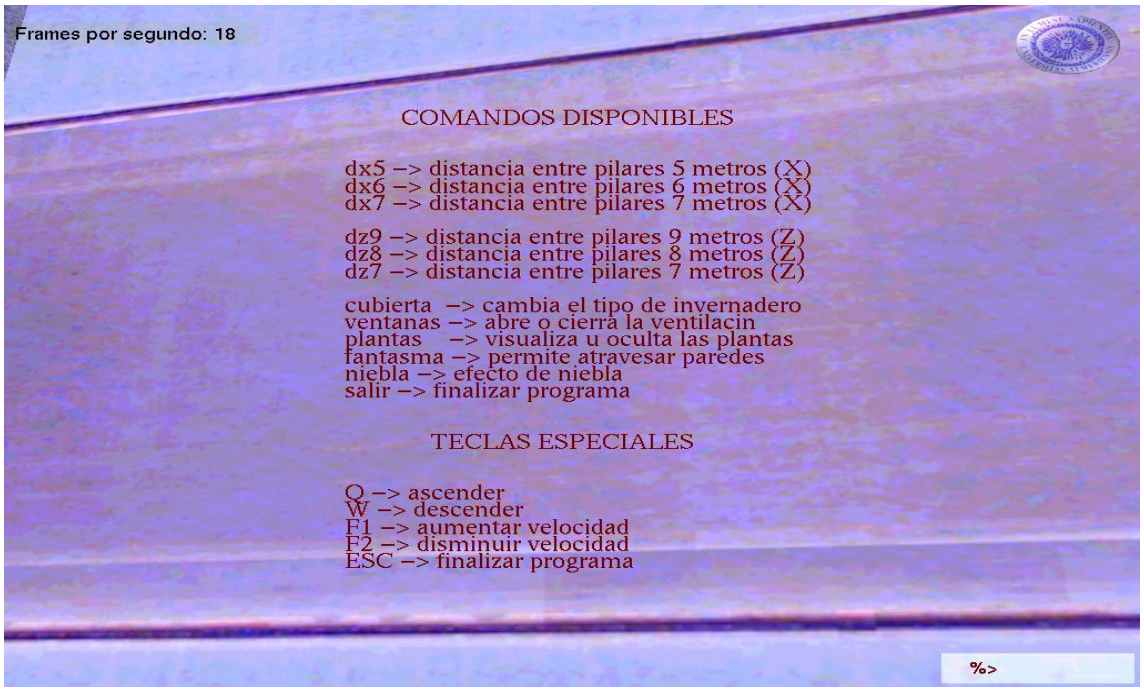


Ventanas de ventilación abiertas



Ventanas de ventilación abiertas

Efecto de niebla



Mensaje de ayuda

4.6. Trabajos futuros y conclusiones

Existen muchas posibles ampliaciones a este proyecto, algunas que mejoran el rendimiento y otras que permiten una mayor funcionalidad:

- Mejorar la rotación de la cámara sobre un eje arbitrario mediante el uso de cuaternios. Las matrices de rotación y de traslación ofrecen buenos resultados, pero también existen otros métodos alternativos para mejorar el rendimiento de la aplicación. Es el caso de los cuaternios, que son números que poseen 3 partes imaginarias, muy usados para realizar rotaciones. Su funcionamiento es algo complejo de entender.

- Ampliar el número de invernaderos posibles a simular. Por ejemplo, añadir el invernadero tipo parral o el ojival. Además dar una descripción de cada tipo de invernadero.

- Aumentar el número de elementos internos al invernadero, incluir por ejemplo los sistemas de regadío o añadir imágenes de plantas diferentes a la del tomate.

- Ya que los resultados de rendimiento obtenidos han sido satisfactorios, se podrían crear objetos más complejos desde el punto de vista del número de polígonos, para añadir más realismo aún a la visita virtual.

- Optimizar la precisión del diseño situando cada vértice de la cuadrícula no a 9 y 5 metros de distancia, sino a un valor menor (por ejemplo 1 metro).

- Permitir la exportación del invernadero (guardar el modelo en disco en formato ASC).

- El engine 3D podría completarse con una infinidad de características, para que su comportamiento sea lo más parecido posible al mundo real.

- Realizar una compilación para el sistema operativo Linux. Para ello es necesario instalar primero la API OpenGL y la librería auxiliar SDL. Es lo realmente bueno de mi código fuente, con muy pocas modificaciones si es que las hay se puede cambiar de plataforma.

- Realizar animaciones de objetos 3D dentro o fuera del invernadero para poder interaccionar con ellos (como por ejemplo ocurre con las ventanas de ventilación, pero con otros objetos distintos). Esto se podría realizar utilizando algún formato que permita la animación, como puede ser el 3DS, o incluso los formatos nativos del Quake para animar a los personajes (MD1, MD2 y MD3).

- Añadir la posibilidad de poder almacenar imágenes de capturas de pantalla en disco o incluso animaciones completas de la visita virtual en formato de vídeo, para poder enviarle dicha información al cliente en caso de no poder asistir personalmente a la empresa constructora de invernaderos.

- Programar visitas guiadas a través de una trayectoria predefinida. Esta funcionalidad sería válida por ejemplo si se desea exponer el programa sin que un usuario esté utilizándolo, para atraer clientes. De este modo la cámara se desplazaría a través del mundo virtual siguiendo una ruta establecida por el usuario de antemano. Esa ruta sería circular, si finaliza comienza de nuevo.

- Simular el comportamiento de las plantas del interior, añadiendo algunos factores externos que podrían interactuar con las plantas de manera beneficiosa o perjudicial. El programa se podría convertir en un sistema experto que no sólo permitiera realizar una visita virtual al invernadero ya creado, sino también peritar dicho invernadero, y de este modo los peritos más inexpertos podrían aprender esta profesión sin causar ningún daño a plantaciones reales.

Y las conclusiones más importantes desde mi punto de vista son las siguientes:

- He conseguido desarrollar una herramienta con una clara aplicación práctica, ya que existía un cierto vacío en este aspecto por no haber programas con estas prestaciones: el usuario puede ver su invernadero antes de ser construido, caminar por su interior e interaccionar con su entorno.

- Uno de mis objetivos desde el comienzo fue optimizar lo máximo posible la visita virtual para que se viera de un modo fluido. Para ello reduje considerablemente el número de polígonos en escena. Y aunque en un principio pensé que disminuiría el realismo de la escena, no ha sido así.

- Se pueden construir invernaderos de formas muy irregulares y extrañas, tan irregulares que jamás serían construidos en la realidad. Es decir, no sólo construye invernaderos comunes, sino que va mucho más allá. Serán la empresa y el cliente los que decidan la forma deseada final.

- Esta herramienta es multiplataforma, basta con modificar unas cuantas líneas de código para compilar en Linux.

Parte II

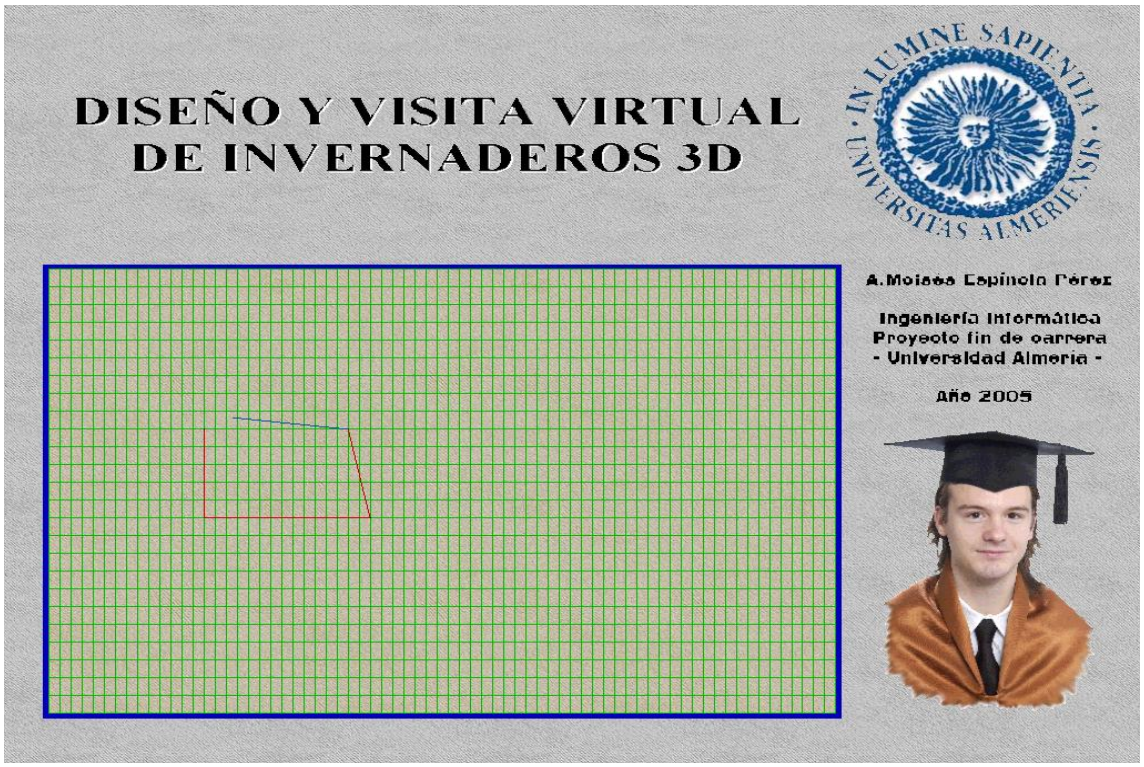
Anejos

Capítulo 1

Manual de usuario

Este programa está dividido en dos partes: por un lado el diseño del invernadero y por otro la visita virtual al modelo creado.

Inicialmente aparece una interfaz en la que se muestra información sobre el proyecto y un plano de cuadrículas. El usuario deberá ir pinchando con el ratón sobre los vértices de las cuadrículas para realizar la forma del invernadero. No se permiten los cruces de líneas. Para finalizar la creación del invernadero el usuario simplemente debe cerrarlo, es decir, volver a pinchar sobre el vértice inicial.



A continuación comienza de manera automática la visita virtual. En dicha visita virtual el

usuario puede moverse a través del mundo 3D mediante las teclas: ARRIBA, ABAJO, IZQUIERDA y DERECHA. Además puede dirigir la cámara hacia el objetivo deseado simplemente moviendo el ratón. También puede volar a través del mundo virtual, utilizando las teclas: Q para elevarse y W para descender.

Como se puede observar, en la esquina superior izquierda se puede ver el número de fotogramas por segundo. Es un factor que muestra el rendimiento del sistema.

En la esquina inferior derecha está la consola de comandos, desde donde el usuario puede introducir el comando deseado. Para ello simplemente debe teclear y las letras aparecerán directamente en la consola. Se puede pedir ayuda con estos comandos: *ayuda*, *info* y *help*. Entonces aparecerán los comandos disponibles:

- dx5: distancia entre pilares 5 metros (en el eje X).

- dx6: distancia entre pilares 6 metros (en el eje X).

- dx7: distancia entre pilares 7 metros (en el eje X).

- dz9: distancia entre pilares 9 metros (en el eje Z).

- dz8: distancia entre pilares 8 metros (en el eje Z).

- dz7: distancia entre pilares 7 metros (en el eje Z).

- cubierta: cambia el tipo de invernadero.

- ventanas: abre o cierra las ventanas de ventilación.

- plantas: muestra u oculta las plantas.

- fantasma: puede atravesar paredes.

- niebla: efecto de niebla.

- salir: finaliza el programa (también con tecla ESC).

Además existen otras dos teclas especiales (F1 y F2) que sirven para aumentar o disminuir la velocidad de la cámara a través del mundo 3D. Esto lo he realizado para poder ajustar más aún el programa a la máquina sobre la que se ejecuta, ya que cuando el invernadero creado es muy grande el número de polígonos mostrados en escena aumenta y conviene aumentar también la velocidad de la visita virtual.

Capítulo 2

Glosario de términos

A

Alpha channel: representación de imágenes dividiéndolas en los colores primarios: Rojo, Verde y Azul (canales RGB), y se añade un cuarto canal (alpha) el cual indica la cantidad de transparencia que tiene la imagen respecto al fondo utilizado para mostrarla.

ASC: formato de archivos tridimensionales del 3D Studio en modo texto. En mi caso he usado este formato, tras exportarlo con un conversor.

AutoCAD: es el estándar universal de diseño asistido por ordenador (CAD). AutoCAD se utiliza mucho más que cualquier otro software de CAD.

B

Blender: software 3D para modelado, texturizado, animación, render y videojuegos. Blender soporta curvas, mallas poligonales, texto, NURBS y metaballs.

Blitz3D: lenguaje de programación (a diferencia de OpenGL y DirectX que son APIs) que nació con la idea de llevar el entorno de programación del Blitz Basic 2D original al campo de las tres dimensiones sin dejar en ningún momento la filosofía de sencillez y potencia que siempre ha caracterizado a esta herramienta de desarrollo multimedia.

Bresenham: algoritmo de trazado de líneas que he utilizado para la deformación de los techos. También existe un algoritmo de trazado de circunferencias de Bresenham.

Bump-mapping: textura o imagen utilizada para modificar la normal de reflexión de la fuente de luz sobre un objeto, dando la apariencia de rugosidades.

C

COSMOS: herramienta software que permite realizar estimaciones para proyectos informáticos utilizando los modelos de estimación basados en puntos de función y COCOMO entre otros.

Curva bezier: tipo de curva de enésimo grado que se genera a partir de lineas guia. Esta curva tiene la particularidad de ser tangente a las rectas guía en el punto de referencia perteneciente a ambas.

D

Direct3D: librería que permite programar videojuegos 3D en Windows. Es el claro rival de OpenGL, pero tiene el inconveniente de no ser multiplataforma como OpenGL. No obstante en funcionalidad y prestaciones no tiene nada que envidiarle.

DWF: acrónimo inglés de 'Design Web Format'. Se trata de un formato de archivo abierto y seguro que Autodesk ha desarrollado específicamente para compartir datos de diseño de ingeniería.

DXF: tipo de archivo nativo de Autocad. Yo creo los objetos en este formato, aunque luego los exporto a otro.

E

Engine 3D: motor gráfico que permite realizar una visita virtual a un mundo 3D, desplazándose a través de él y pudiendo enfocar con el ratón el objetivo deseado. Además puede tener algunas características del mundo real (como la detección de colisiones, transformaciones de objetos, leyes físicas, etc).

Environment map: tipo de textura utilizada para simular reflexiones sobre los objetos. Consiste en calcular las vistas de la escena desde el punto de vista del objeto y luego realizar una proyección sobre la superficie del mismo.

Extrusión: proceso mediante el cual parte de un volumen es trasladado en una dirección dejando un 'rastro' volumétrico. Por ejemplo si se extruye un circulo en dirección perpendicular al plano que lo contiene se obtiene un cilindro.

F

Frame: cuadro o imagen individual de las que componen una animación. La sucesión de frames consecutivos con pequeñas diferencias producen la ilusión de movimiento. La velocidad de la animación se mide en Frames por segundo (PFS).

Frames por segundo: medida de rendimiento que equivale a la velocidad de la animación, es decir, el número de frames o imágenes que se muestran en un segundo por pantalla.

G

GIMP: programa de edición gráfica que permite trabajar de manera profesional con todo tipo de imágenes. Lo he usado para la creación de texturas.

I

Invernadero multitúnel: estructura con las medidas requeridas y cubiertas con determinado material translúcido o trasparente, que permite tanto el crecimiento óptimo de las plantas, como el acceso a las personas para laborar en el cultivo. En mi caso la cubierta es de plástico o policarbonato.

M

Mesh: término que se refiere a una figura en 3D en general que esté formada por polígonos.

Metaball: objeto 3D constituido por un cuerpo volumétrico similar a la gota de un líquido. Varios objetos metaball tienden a fundirse cuando se colocan en coordenadas muy cercanas o solapadas.

Microsoft Project: herramienta software que permite realizar planificaciones para proyectos. Entre otras funciones importantes puede realizar red de tareas Pert y diagramas Gantt.

Microsoft Visual Studio: paquete de programación que integra entre otros lenguajes el Visual C++, que yo he utilizado como base para mi proyecto.

N

Nurb: objeto 3D que es generado mediante curvas bezier. Existen superficies nurb, así como curvas nurb que es algo así como una curva bezier 3D.

O

OpenGL: librería que posee una gran funcionalidad relacionada con la programación de videojuegos 3D. Existen primitivas básicas, muchas funciones para poder operar con ellas, etc. Actualmente es puntera en el mercado del mundo virtual.

P

Phyton: lenguaje de script de Blender que permite programar algunas características del gameengine que incorpora.

R

Render: término que se refiere a la generación de un frame o imagen individual. El render consume potencia de cómputo debido a los cálculos necesarios según el número de objetos, luces y efectos en la escena y según el punto de vista de la cámara.

S

SDL: librería auxiliar de OpenGL que ayuda a gestionar aspectos como los eventos producidos por el teclado o ratón, y añade mucha más funcionalidad a OpenGL. Ofrece mayor rendimiento que otras librerías en principio equivalentes.

T

Textura: imagen proyectada sobre parte o la totalidad de la superficie de un objeto para darle apariencia fotorrealística. Existen texturas procedimentales (calculadas) o texturas de mapa de imagen.

U

UMLStudio: programa que permite realizar ingeniería inversa para obtener el diagrama de clases en UML a partir de un código fuente existente.

V

VRML: es un lenguaje de modelado de mundos virtuales en tres dimensiones. Está muy ligado a Internet, ya que existen muchos pluggins para navegadores que permiten visualizar mundos en VRML.

W

WBS: diagrama de descomposición donde se visualizan las diferentes tareas o procesos que se tienen que realizar.

Wireframe: forma de presentación de los objetos 3D en pantalla por medio de 'alambres' o líneas que representan el volumen.

Capítulo 3

Fuentes de información

3.1. Referencias bibliográficas

3.1.1. OpenGL

- *OpenGL programming Guide*, Jackie Neider (2000)

- *OpenGL programming for Windows*, Ron Fosner (1997)

- *OpenGL programming guide*, Neider & Davis & Woo (1996)

- *OpenGL superBible 2nd edition*, Richard S. Wright Jr & Michael R. Sweet

- *Interactive computer graphics - A top-down approach with OpenGL*, Edward Angel

- *OpenGL game programming*, Kevin Hawkins & Dave Astle

- *OpenGL 1.4 reference Manual*, Dave Shreiner

- *The OpenGL extensions guide*, Eric Lengyel

- *OpenGL shading language*, Randi J. Rost

- *OpenGL 2.0 specification* (.pdf desde Internet)

- *OpenGL programming for the X Window system*, Mark J. Kilgard

- *Open Geometry: OpenGL + Advanced Geometry*, George Glaeser & Hellmuth Stachel

- *3D graphics programming with OpenGL*, Clayton Walnum

- *Computer graphics using OpenGL*, F.S. Hill

3.1.2. Engines 3D

- *Gráficas por computadora*, Donald Hearn & M. Pauline Baker
- *3D game engine design: approach to real-time computer graphics*, David H. Eberly
- *3D games: real-time rendering and software technology*, Alan Watt & Fabio Policarpo
- *Game architecture and design: a new edition*, Andrew Rollings & Dave Morris
- *Designing virtual worlds*, Richard Bartle
- *Game design: theory and practice*, Richard Rouse, Steve Ogden & Mark Louis Rybczyk
- *Game art: the graphic art of computer games*, Leo Hartas & Dave Morris
- *Tricks of the 3D game programming gurus - advanced 3D graphics and rasterization*, André LaMothe
- *3D game engine programming*, Stefan Zerbst
- *3D game engine architecture: engineering real-time applications with wild magic*, David H. Eberly
- *Building a 3d game engine in C++*, Brian Hook

3.1.3. Autocad

- *AutoCAD avanzado 2002*, José Antonio Tajadura Zapirain & Javier López
- *AutoCAD 2000 avanzado*, Javier López Fernández & José Antonio Tajadura
- *La biblia de AutoCAD 2002*, George Omura
- *AutoCAD 2004 bible*, Ellen Finkelstein
- *Engineering graphics with AutoCAD 2004*, James D. Bethune
- *Engineering design graphics*, James H. Earle

3.1.4. Visual C++

- *Visual C++, Aplicaciones para Win32*, Fco. Javier Ceballos
- *Beginning Visual C++ 6.0*, Ivor Horton
- *Professional MFC With Visual C++ 6.0*, Mike Blaszczak
- *Programming Visual C++*, David J. Kruglinski

- *Microsoft Visual C++ 6.0 programmer's guide*, Beck Zaratian
- *Microsoft Visual C++ 6.0 reference library*, Microsoft corporation
- *Microsoft Visual C++ 6.0*, Don Gosselin
- *Learn Microsoft Visual C++ 6.0 now*, Chuck Sphar

3.1.5. Blender

- *The official blender 2.3 Guide: free 3D creation suite for modelling, animation, and rendering*, Ton Roosendaal & Stefano Selleri
- *The official blender gameKit: interactive 3D for artists*, Carsten Wartmann & Ton Roosendaal

3.1.6. Blitz3D

- *Curso de programación y diseño de videojuegos, coleccionable de kiosco*, diversos autores

3.1.7. Direct3D

- Real-time 3D terrain engines using C++ and DirectX 9, Gregory Snook
- Beginning Direct 3D game programming, Wolfgang F. Engel
- Direct 3D programming kick start, Clayton Walnum

3.1.8. VRML

- *The VRML 2.0 handbook*, Jed Hartman
- *VRML: browsing and building cyberspace*, M. Pesce
- *The VRML sourcebook*, Andrea L. Ames & Moreland
- *The annotated VRML 2.0 reference manual*, Rikk Carey & Gavin Bell

3.1.9. Invernaderos

- *Construcción de invernaderos*, Zoilo Serrano Cermeño
- *Tecnología de invernaderos II, Curso superior de especialización*, Jerónimo Pérez Parra & Isabel Mª Cuadrado
- *Invernaderos: diseño, construcción y ambientación*, Antonio Matallana Gonzalez & Juan Ignacio Montero Camacho
- *Invernaderos: planificación y construcción*, Jorn Pinske

3.2. Páginas web

3.2.1. OpenGL

- *http://www.opengl.org/*
- *http://www.sgi.com/products/software/opengl/*
- *http://www.eecs.tulane.edu/www/Terry/OpenGL/Introduction.html*
- *http://www.gametutorials.com/Tutorials/OpenGL/OpenGL_ Pg1.htm*
- *http://www.ncrg.aston.ac.uk/ cornfosd/graphics/opengl/openglman.html*
- *http://rush3d.com/reference/opengl-bluebook-1.0/*
- *http://www.cs.man.ac.uk/applhax/OpenGL/*
- *http://openglenfichas.uji.es/*

3.2.2. Engines 3D - Programación de videojuegos

- *http://cg.cs.tu-berlin.de/ ki/engines.html*
- *http://www.3dengines.net/*
- *http://www.gametutorials.com/*
- *http://nehe.gamedev.net/*
- *http://www.codepixel.com/*

3.2.3. Autocad

- *http://www.autodesk.com/*
- *http://www.arquitectuba.com.ar/bloques.asp*
- *http://www.galiciacad.com/manuales/*
- *http://www.todoarquitectura.com/*

3.2.4. Visual C++

- *http://www.dcp.com.ar/*
- *http://www.ftponline.com/vsm/*
- *http://msdn.microsoft.com/visualc/*
- *http://www.codeguru.com/*
- *http://www.mvps.org/vcfaq/*
- *http://www.microsoft.com/latam/visualc/*

3.2.5. Blitz3D

- *http://www.blitzbasic.com/*
- *http://portalxuri.dyndns.org/blitzbasico/*
- *http://www.freewebs.com/sweenie/*

3.2.6. Blender

- *http://www.blender3d.com/*
- *http://www.blender.org/*
- *http://www.elysiun.com/*
- *http://mipagina.cantv.net/planetablender/default.htm*
- *http://www.3dup.com/*

3.2.7. VRML

- *http://www.vrml.org/*
- *http://cosmo.sgi.com/*
- *http://www.vrmlsite.com/*
- *http://www.embl-heidelberg.de/vrml*
- *http://vrml.sgi.com/intro.html*
- *http://hiwaay.net/ crispen/vrml*
- *http://tom.di.uminho.pt/vrml*

- *http://www.sdsc.edu/siggraph96vrml*

- *http://vrml.wired.com/*

- *http://wmaestro.com/*

- *http://www.ctv.es/USERS/palancar*

3.2.8. Invernaderos

- *http://www.infoagro.com/industria_auxiliar/tipo_invernaderos.asp*

- *http://agrarias.tripod.com/invernaderos.htm*