UNIVERSIDAD DE MURCIA
FACULTAD DE INFORMÁTICA

ASIGNATURA
PROYECTO INFORMÁTICO

TIERRA INHÓSPITA
DESARROLLO DE UN INTERFACE PERCEPTUAL PARA LA NAVEGACIÓN EN UN MUNDO VIRTUAL 3D.

Autor

Sergio Fructuoso Muñoz
sergiofr@allianz.infonegocio.com

Director

Ginés García Mateos
Departamento de Informática y Sistemas
ginesgm@um.es

Junio 2004

Gracias a mi familia, Ginés y a mi "niña" Ana.

Resumen

En este proyecto abordamos el análisis, diseño y desarrollo de un interfaz perceptual aplicado a la navegación en un entorno virtual tridimensional. La emergente área de los interfaces perceptuales es un ámbito de investigación en el que confluyen dos grandes campos de la informática: la programación gráfica y la visión por computador. El objetivo final de esta área es crear nuevos y más naturales mecanismos de interacción hombre/máquina, que sustituyen o complementan a los métodos tradicionales. En el sistema tratado en este proyecto, la entrada se realiza mediante la detección y seguimiento de una cara, interpretando en cada momento en qué dirección estamos mirando y si nos estamos moviendo. El desarrollo del proyecto ha requerido el estudio, análisis y aplicación de las técnicas y herramientas existentes en la generación de entornos 3D, en la adquisición y procesamiento de imágenes, y en los problemas de detección y seguimiento de las caras humanas.

En concreto, para el desarrollo de la aplicación hemos usado Visual Studio .NET 2003 y el código lo hemos escrito en C++. Hacemos uso de las herramientas DirectX 9.0, OpenCV e Intel® Image Processing Library. Además ha sido necesario investigar en los problemas de extracción de información 3D a partir de imágenes de caras humanas. Se ha realizado una propuesta esencialmente basada en heurísticas, que ha demostrado su viabilidad en el prototipo implementado.

Palabras clave: visión artificial, seguimiento 3D, seguimiento de la cara, interfaces perceptuales y entornos virtuales.

Índice General

1. Introducción y referencias históricas 11
  1.1 Los retos en el campo de la visión artificial 11
  1.2 Entorno virtual en primera persona 12
  1.3 El entorno virtual "Tierra inhóspita" 13
  1.4 Organización del documento 14
2. Análisis de Objetivos 17
  2.1 Objetivos y alcance del proyecto 17
  2.2 Qué no es objetivo del proyecto 17
  2.3 Plan de trabajo 18
3. Tecnologías y herramientas utilizadas 21
  3.1 Técnicas utilizadas 21
    3.1.1 Integrales proyectivas 21
    3.1.2 Motor Gráfico 25
    3.1.3 Colocar objetos en el mundo 30
    3.1.4 Luces 32
    3.1.5 Texturas 33
    3.1.6 Sonidos 33
    3.1.7 Aceleración del renderizado 33
  3.2 Herramientas utilizadas 36
    3.2.1 Elección del lenguaje 36
    3.2.2 Intel® Image Processing Library 37
    3.2.3 OpenCV 37
    3.2.4 3D Studio Max 37
    3.2.5 DirectX 9.0 37
4. Diseño y resolución del proyecto 39
  4.1 Estructura del proyecto 39
  4.2 Detección y seguimiento de caras 39
  4.3 Detección de movimientos 40
    4.3.1 Movimiento en el eje X 40
    4.3.2 Movimiento en el eje Y 41
    4.3.3 Movimiento en el eje Z 42
    4.3.4 Giro en el eje X 43
    4.3.5 Giro en el eje Y 44
    4.3.6 Giro en el eje Z 45
  4.4 Priorizar los movimientos 46
  4.5 Intentando reducir el ruido en las detecciones 47
  4.6 Entorno virtual 3D 47
  4.7 Interacción con el mundo virtual 49
    4.7.1 Moviéndonos por el entorno 49
    4.7.2 Movimientos dentro del plano lógico 54
    4.7.3 Detección de colisiones 55
  4.8 Pintar la escena 58
    4.8.1 ¿Qué vemos? 58
    4.8.2 Efecto de niebla 58
  4.9 Sonidos 59
  4.10 Entrada por teclado 59
5. Conclusiones y trabajos futuros 61
  5.1 Una vista atrás 61
  5.2 Dificultades y limitaciones 61
  5.3 Vías futuras 62
  5.4 Objetivos cumplidos 62
  Bibliografía 63
  APENDICE A. INSTALACIÓN 65

APENDICE B. IMPLEMENTACIÓN 67
APENDICE C. EJEMPLO DE USO...87
APENDICE D. MANUAL DE USUARIO 95

Índice de figuras

Ilustración 1. Juegos en primera persona. A la izquierda Wolfestein y a la derecha Doom................12
Ilustración 2. Duke Nukem. 12
Ilustración 3. Quake. 13
Ilustración 4. Funcionamiento del programa 14
Ilustración 5. Diagrama de Gantt de las fases y tareas para el desarrollo del proyecto.......................19
Ilustración 6. Ejemplo de las proyecciones sobre dos caras. 23
Ilustración 7. Preprocesamiento. 24
Ilustración 8. Alineamiento vertical. 24
Ilustración 9. Alineamiento horizontal. 25
Ilustración 10. Estimación de la orientación. 25
Ilustración 11. Los vectores de la cámara definiendo la posición y la orientación relativa al mundo. 26
Ilustración 12. Rotación sobre el eje X (PITCH). 28
Ilustración 13. Rotación sobre el eje Y (YAW). 29
Ilustración 14. Rotación sobre el eje Z. 30
Ilustración 15. Ejemplo de una traslación. 30
Ilustración 16. Ejemplo de una rotación de 45º 31
Ilustración 17. Ejemplo de escalar. 32
Ilustración 18. Tipos de luz. 33
Ilustración 19. Representación de regiones de un mapa 34
Ilustración 20. El frustum. 34
Ilustración 21. Vista frustum 35
Ilustración 22. A la izquierda un ejemplo de Bounding Sphere y a la derecha un ejemplo de Bounding Box. 36
Ilustración 23. Medición del desplazamiento en X. 41
Ilustración 24. Medición del desplazamiento en Y. 42
Ilustración 25. Medición del desplazamiento en Z. 43
Ilustración 26. Integral proyectiva de los ojos. 44
Ilustración 27. Integral proyectiva de la vertical de la cara. 45
Ilustración 28. Ángulo de inclinación de la cara. 46
Ilustración 29. Ejemplo de un mapa del entorno 3D, con túnel y efecto de niebla. 47
Ilustración 30. Objetos gráficos 3D del entorno de Tierra inhóspita. 48
Ilustración 31. Vista de pájaro del mundo virtual de Tierra inhóspita. 49
Ilustración 32. Trozo de mapa dividido en celdas. 53
Ilustración 33. Puntos que forman la casilla 1. 54
Ilustración 34. Movimiento en el mundo. 55
Ilustración 35. Límites del mapa con paredes y objetos. 56
Ilustración 36. Ejemplo de choque. 57

Índice de ecuaciones

Ecuación 1. Proyecciones verticales y horizontales de una imagen. 21
Ecuación 2. Distancia de una señal a un modelo. 22
Ecuación 3. Posición destino al andar...27
Ecuación 4. Nueva vista al andar. 27
Ecuación 5. Calculo del cambio de altura..27
Ecuación 6. Posición destino al movernos en altura. 27
Ecuación 7. Vista final al desplazarnos en altura...27
Ecuación 8. Vector de desplazamiento lateral 28
Ecuación 9. Posición final cuando nos desplazamos lateralmente. 28
Ecuación 10. Vista final al moverse lateralmente..28
Ecuación 11. Vista final en rotación sobre el eje X. 28
Ecuación 12. Vista destino al rotar sobre el eje Y...29
Ecuación 13. Vista destino al rotar sobre el eje Z...29
Ecuación 14. Traslación de un punto...30
Ecuación 15.Rotación de un punto sobre el eje X...31
Ecuación 16. Rotación de un punto sobre el eje Y...31
Ecuación 17. Rotación de un punto sobre el eje Z...31
Ecuación 18. Escalar un punto. 32
Ecuación 19. Composición de traslación, escala y rotación. 32

Capítulo 1

Introducción y referencias históricas

1.1 Los retos en el campo de la visión artificial

Hay un interés creciente, cada vez mayor, de construir interfaces más naturales, atractivas y con mayor facilidad de uso. Ello ha dado lugar a la aparición de una línea de trabajo en lo que se conoce como **interfaces multimodales**. Esta forma de concebir la comunicación con el ordenador, se caracteriza por la integración de distintos dispositivos (reconocimiento y síntesis de voz, visión por computador, teclado, ratón, lápiz óptico, etc.), con el objetivo de proporcionar un conjunto alternativo y redundante de vías de comunicación. En este sentido, el rostro humano es uno de los medios más prometedores para los nuevos mecanismos de interacción hombre/máquina.

El seguimiento del rostro humano mediante el análisis de una secuencia de imágenes, es el paso previo para numerosas aplicaciones de visión computacional: lectura de labios como apoyo de reconocimiento de voz, codificación de gestos, animación de personajes mediante copia de movimientos, estimación de dirección de la mirada, etc.

Los sistemas de seguimiento de caras que se han construido hasta el día de hoy, pueden agruparse en tres categorías:

- Seguimiento 2D: realizan el seguimiento sólo en posición.
- Seguimiento 2½D: realizan un seguimiento 2D con alguna información de orientación.
- Seguimiento 3D: realizan el seguimiento en los seis grados de libertad de la cabeza.

La mayoría de los sistemas construidos hasta la fecha se caracterizan por usar un único modelo de seguimiento. Los que realizan un seguimiento 3D son menos robustos que los que lo hacen en 2D, pero son mucho más precisos y permiten analizar el movimiento de todas las partes de la cara. Por el contrario, los sistemas que se basan en 2D emplean primitivas muy simples (color, bordes, etc.) y, por ello, son capaces de trabajar en unas condiciones más adversas (mala iluminación, por ejemplo), permitiendo una fácil recuperación ante un fallo de seguimiento.

Para el desarrollo de este proyecto se ha usado un método de seguimiento que podría clasificarse en la categoría de 2D. Este método esta basado en la aplicación de integrales proyectivas y ha sido propuesto y desarrollado por el profesor Ginés García Mateos [García'03]. Utilizando información adicional provista por las integrales proyectivas, se han extendido los resultados del seguimiento para obtener un seguimiento 2½D. A lo largo del capítulo 4 se desarrollan las propuestas realizadas.

1.2 Entorno virtual en primera persona

Los entornos virtuales en primera persona comenzaron con el infinitamente reconocido Castle Wolfenstein. En este juego, tomamos el rol de un personaje que se encuentra prisionero dentro de un castillo lleno de Alemanes Nazis, a los que debemos eliminar con nuestro arsenal de armas modernas. El juego más importante en este género, es el famoso Doom, creado por Id Software. En la Ilustración 1 se muestran los ejemplos de estos juegos.

Ilustración 1. Juegos en primera persona. A la izquierda Wolfestein y a la derecha Doom.

Luego, nos encontramos con el primero de los juegos en primera persona, Duke Nukem, que nos lleva a un ambiente semi-real, y muchos expertos lo denominaron entorno virtual de 2.5 Dimensiones, porque no es totalmente tridimensional ya que usa *sprites*, es decir, mapa de bits planos, para los personajes. La Ilustración 2 muestra un ejemplo, en el que se puede apreciar la mayor complejidad de las escenas respecto a sus predecesores.

Ilustración 2. Duke Nukem.

Realmente, el que marcó un punto de inflexión en los entornos virtuales, es el famoso Quake, nuevamente por Id Software, que nos ofrece por primera vez un ambiente 3D, y gráficos realmente revolucionarios para su época pero permitido por la mayor potencia de las máquinas.

Ilustración 3. Quake.

En consideración tenemos que tener en cuenta que la visualización en primera persona es más atractiva para el usuario porque es mucho más natural y el avance de la informática (mejores procesadores, más memoria, etc.) ha permitido y permite cada vez más la mejora del "realismo" en las imágenes sintéticas.

1.3 El entorno virtual "Tierra inhóspita"

El objetivo del presente proyecto es desarrollar un interface perceptual, en el que el ser humano sea capaz de interactuar en un entorno virtual tridimensional, utilizando para ello los movimientos de su cara. La estructura del sistema que se pretende desarrollar consta de una cámara de vídeo, conectada a un ordenador que realiza dos tipos de tareas: por un lado, el análisis de la secuencia de vídeo para la extracción de la posición de la cara y los elementos faciales (usando técnicas propias de la visión por computador), y por otro lado, la generación de escenas en un mundo virtual partiendo de las posiciones detectadas (usando técnicas propias de síntesis de imágenes 3D).

Como ya hemos comentado, la detección del rostro humano es un problema esencial en el contexto de los interfaces perceptuales y el procesamiento de imágenes, ya que un supuesto de localización fija no es posible en la práctica. En la detección se trata de determinar el número de caras que aparecen en una imagen y, para cada una de ellas, su localización y extensión espacial. Encontrar un método rápido y robusto para detectar caras bajo condiciones no triviales es todavía un reto importante. El otro gran problema es el seguimiento de las caras, es decir, la localización de la cara y sus componentes a lo largo de la secuencia de video.

Entre otros métodos, las integrales proyectivas han sido utilizadas en problemas de estos tipos. La mayoría de las técnicas existentes están basadas en análisis max-min., lógica difusa y aproximaciones heurísticas similares y su uso constituye una mínima parte de los sistemas de visión. En el artículo [García'03] se propone usar estas proyecciones como un instrumento para crear modelos de caras unidimensionales.

Las integrales proyectivas son usadas para modelar la apariencia visual de la cara humana. De esta forma, la detección se basa en la búsqueda de un patrón conocido en las imágenes (patrón formado por integrales proyectivas). El problema clave es la alineación de los patrones de proyección con respecto a un modelo dado de un rostro genérico.

El marco de trabajo de las integrales proyectivas, puede ser también aplicado en problemas como la localización de rasgos faciales, estimación de posturas, expresiones y el seguimiento de caras. En particular, este último problema resulta de especial interés en el presente proyecto, donde se trabaja con secuencias de vídeo, donde aparece el rostro del individuo que pretende interactuar con el sistema. El seguimiento parte de una detección inicial en la primera imagen y debe actualizar la posición de los componentes faciales a lo largo de la secuencia. El proceso se muestra de forma esquemática en la Ilustración 4.

Para la resolución de esta parte del proyecto, se parte del trabajo previo del director antes mencionado, en el que se abordan los problemas de detección y seguimiento de caras. A partir de esta base, ese ha avanzado en la extracción de la información necesaria y su interpretación para posibilitar la navegación por el entorno virtual 3D como veremos más adelante.

Para el entorno virtual se ha construido un mapa con diferentes objetos estáticos en la escena. Se ha creado un motor gráfico que simula los movimientos de un personaje en primera persona y lo hemos dotado de un sistema de detección de colisiones para aumentar el realismo.

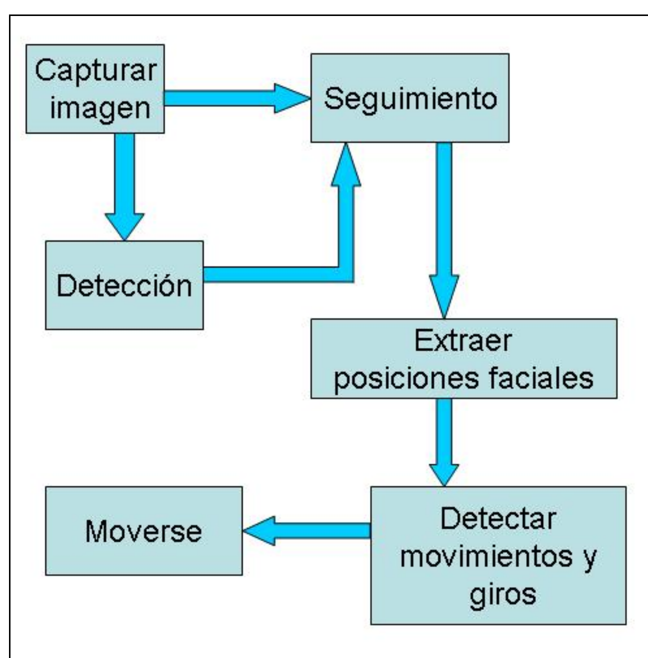


Ilustración 4. Funcionamiento del programa

1.4 Organización del documento

La documentación de este proyecto fin de carrera, está dividida en cuatro secciones principales. La primera de ellas, Introducción y Referencias históricas, tiene por objeto la contextualización de la aplicación dentro del campo de la visión y la informática gráfica y se introducen problemas que surgen de forma breve.

La sección de Análisis de Objetivos comienza con un repaso sobre los elementos que son objetivo de este proyecto y los que no lo son. También define el plan de trabajo, repartiendo en fases el desarrollo de estos objetivos.

En el capítulo de Técnicas y Herramientas se definen los métodos de uso frecuente dentro de todos los campos que tratamos y las herramientas que vamos a utilizar durante el desarrollo del proyecto.

El cuarto capítulo se centra en el diseño y la resolución de la aplicación "Tierra inhóspita". Abarca un completo estudio del diseño e implementación de la interfaz gráfica, del seguimiento de caras y de la detección de movimientos y giros.

Para finalizar, se incluye un capítulo de Conclusiones, resumiendo los objetivos cumplidos y los logros alcanzados.

Capítulo 2

Análisis de Objetivos

2.1 Objetivos y alcance del proyecto

El objetivo de este proyecto es desarrollar un interface perceptual capaz de reconocer e interpretar la posición 3D de un usuario humano, a través del análisis de imágenes faciales, aplicando los resultados del control de movimiento a un entorno virtual tridimensional en primera persona.

Más concretamente los objetivos del proyecto se pueden resumir en los siguientes puntos:

- Investigar los métodos y técnicas usados actualmente en el ámbito de los interfaces perceptuales y los entornos virtuales. En este sentido, el proyecto se mueve entre los ámbitos de la visión por computador y la síntesis de imágenes.

- Diseñar un proceso para el control de un entorno virtual mediante la localización de la cara, usando la técnica de integrales proyectivas. Se deberían extender la técnica existente para la extracción de información sobre posición y giro de las caras.

- Desarrollar un sistema que muestre la viabilidad práctica del acercamiento propuesto.

- Para garantizar la viabilidad, el sistema deberá ser capaz de funcionar en tiempo real. Esto tiene importantes implicaciones sobre la eficiencia de los algoritmos utilizados, tanto en los problemas de visión como en los de generación de imágenes.

2.2 Qué no es objetivo del proyecto

Desarrollar un sistema de detección y seguimiento de caras. Esto viene ya dado por el trabajo previo del director del proyecto.

Se pretende limitar la complejidad del entorno virtual porque no desarrollaremos objetos animados.

Meter personajes en el mundo para que interactúen con el usuario. No se ha podido realizar por la extensión del proyecto

2.3 Plan de trabajo

Partiendo de los objetivos estipulados en el punto 2.1 se han planteado las siguientes tareas, divididas en fases:

FASE 1 (Tiempo estimado: 120 horas)

1. Estudio del modelo teórico.
2. Aprendizaje de DirectX, MFC y OpenCV.
3. Definición de los casos de uso preliminares.

FASE 2 (Tiempo estimado: 400 horas)

1. Definición de la arquitectura del sistema.
2. Implementación del entorno gráfico.
3. Implementación del mapa del entorno virtual.
4. Creación de decorados.
5. Detección de colisiones.
6. Creación de sonidos.

FASE 3 (Tiempo estimado: 150 horas)

1. Adaptación de las librerías existentes de detección y seguimiento de caras.
2. Detectar movimiento y giro 3D e interpretarlo.

FASE 4 (Tiempo estimado: 50 horas)

1. Optimizar la herramienta para tiempo real.
2. Suavizado en la detección de movimientos.

FASE 5 (Tiempo estimado: 50 horas)

1. Evaluación y validación.
2. Documentación.

En la Ilustración 5 se muestra la organización temporal en fases.

ID.	SEP 2003	OCT 2003	NOV 2003	DIC 2003	ENE 2004	FEB 2004	MAR 2004	ABR 2004	MAY 2004
FASE 1									
FASE 2									
FASE 3									
FASE 4									
FASE 5									

Ilustración 5. Diagrama de Gantt de las fases y tareas para el desarrollo del proyecto.

Capítulo 3

Tecnologías y herramientas utilizadas

3.1 Técnicas utilizadas

En este apartado se van a describir las principales metodologías, técnicas y herramientas que han sido probadas con cierto éxito en trabajos previos relacionados, algunas de las cuales, han sido empleadas en el presente proyecto.

3.1.1 Integrales proyectivas

La detección de la cara humana es un problema esencial en el contexto de los interfaces perceptuales. Las integrales proyectivas pueden ser usadas para detectar caras y localizar características faciales, como se explica y desarrolla en [García'03] y [García'02]. A continuación vamos a describir la aplicación de esta herramienta en los problemas de detección y seguimiento de caras, haciendo uso de los mencionados artículos.

3.1.1.1 Definición

Antes de definir las proyecciones verticales y horizontales, deberemos indicar que las proyecciones serán consideradas como señales como señales unidimensionales. Definimos una región de interés de una imagen **i** como el conjunto: **R(i)** = {(x, y) / ∀ (x, y) ∈ Rango(i), PR(x, y) = true}, donde **PR** es una función que nos indica si se cumple una cierta propiedad. Las proyecciones verticales y horizontales vienen definidas de la siguiente forma. Sea **i** una imagen y **R(i)** una región de la misma, se definen las proyecciones vertical y horizontal de **R(i)** como las señales unidimensionales dadas por las siguientes fórmulas:

Proyección vertical de R(i):

$P_{VR(i)}$: Rangoy(R(i)) $\rightarrow \nabla^n$

$$\mathbf{P_{VR(i)}(y)} := \frac{1}{n(y)} \sum_{\forall (x,y) \in R(i)} i(x,y) \quad \text{con } n(y) = \sum_{\forall (x,y) \in R(i)} 1$$

Proyección horizontal de R(i):

$P_{HR(i)}$: Rangox(R(i)) $\rightarrow \nabla^n$

$$P_{HR(i)}(x) := \frac{1}{n(x)} \sum_{\forall (x,y) \in R(i)} i(x,y) \quad \text{con } n(x) = \sum_{\forall (x,y) \in R(i)} 1$$

Ecuación 1. Proyecciones verticales y horizontales de una imagen.

3.1.1.2 Alineamiento

El problema clave en detección de objetos usando proyecciones es el alineamiento. El propósito del alineamiento de las proyecciones es producir nuevas proyecciones derivadas donde la localización de las características faciales es la misma en todas ellas. Después del alineamiento, ojos, nariz y boca aparecen en la misma posición en todos los patrones.

Distinguiremos entre las señales de las integrales proyectivas y los modelos de señal. Una señal es una función discreta S: $\{s_{min}, ..., s_{max}\} \rightarrow R$. Un modelo de proyección describe una variedad de señales, y es normalmente obtenido de las diferentes instancias de un mismo tipo de objeto. En el artículo [García'03] se propone un modelado *gaussiano* de las proyecciones, en el que los modelos correspondientes a un conjunto de señales vienen dados por la media y la varianza de cada punto en el rango de las señales. Así, un modelo de proyección puede ser expresado como el par de funciones:

M: $\{m_{min}, ..., m_{max}\} \rightarrow \mathbf{R}$ Media de cada punto en el rango de la señal.
V: $\{m_{min}, ..., m_{max}\} \rightarrow \mathbf{R}$. Varianza en cada punto.

Como ya se ha comentado, el alineamiento es una pieza clave cuando trabajamos con proyecciones. Tras un alineamiento adecuado, las características en las imágenes deben ser proyectadas en las mismas localizaciones de las señales. El problema del alineamiento de las señales unidimensionales es similar a la localización de patrones en una imagen 2D. Pero usando las proyecciones conseguimos una importante reducción en el número de grados de libertad; en la práctica, sólo dos parámetros tienen que ser considerados en el alineamiento: escala y traslación. Supongamos que una señal S es una instancia de un modelo dado (M,V). El problema puede ser formulado como encontrar los valores de escala d y traslación e, que proyecten puntos equivalentes en las imágenes en las mismas localizaciones. El alineamiento de S, denotado por S' viene dado por:

$$S': \{(s_{min} - e)/d, ..., (s_{max}-e)/d\} \rightarrow R \; ; \; S'(i) := S(d \cdot i + e)$$

Además, si S es una instancia del modelo, la distancia definida en Ecuación 2 puede ser usada como una buena medida de calidad: un valor bajo indicará un buen alineamiento y un valor alto el caso contrario. En definitiva, se trata de encontrar el par de valores *(d, e)* que minimice el resultado de la ecuación de abajo.

$$d(S', (M, V)) = \frac{1}{\|R\|} \sum_{i \in R} \frac{(S'(i) - M(i))^2}{V(i)}$$

Ecuación 2. Distancia de una señal a un modelo.

3.1.1.3 Aplicación de las integrales a las caras

Un ejemplo de aplicación de las integrales proyectivas lo podemos apreciar en la Ilustración 6. Podemos observar en la parte superior de la cara las proyecciones horizontales de los ojos de cada cara respectivamente, en la parte de abajo la proyección horizontal de cada boca y por último la proyección vertical de toda la cara, que está situada a ambos lados de las dos caras.

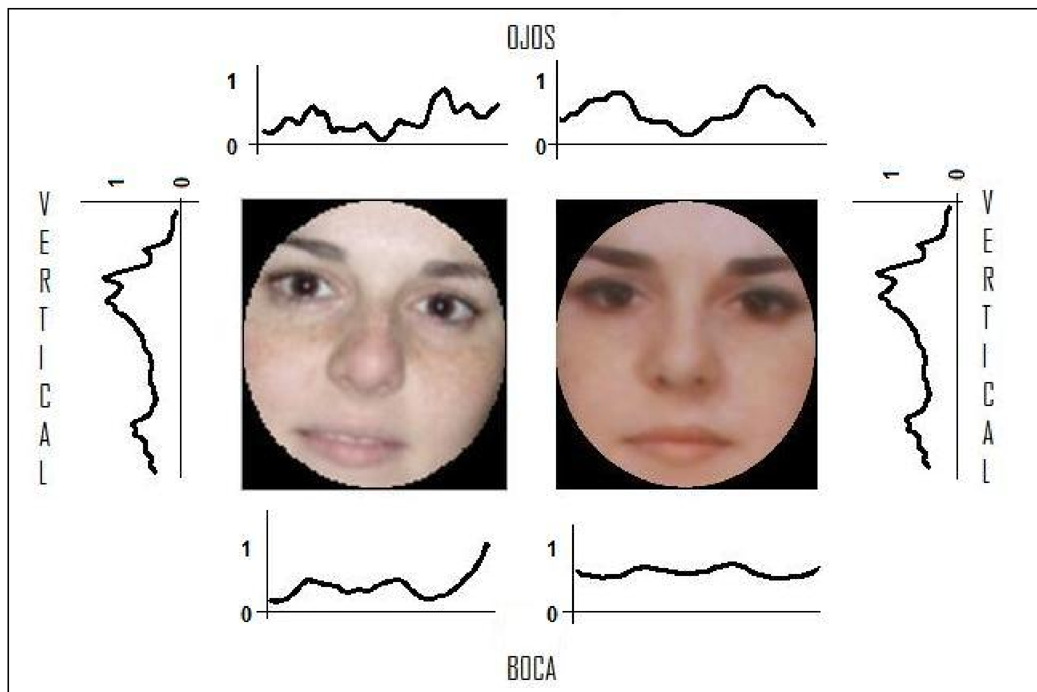


Ilustración 6. Ejemplo de las proyecciones sobre dos caras.

Es importante hacer notar que la base de la técnica de procesamiento de caras usando proyecciones se basa en la uniformidad de los patrones de proyección vertical y horizontal. Independientemente del individuo, de la expresión facial y, hasta cierto grado, de la iluminación y la colocación de la región segmentada, los patrones de proyección vertical y horizontal de las caras tienen estructuras similares. Por ejemplo, en la Ilustración 6 se puede apreciar que la proyección vertical de la cara tiene un patrón típico donde aparece una zona oscura correspondiente a la región de los ojos, una zona más clara a la altura de la nariz, y una zona oscura (pero algo menor) en el lugar donde se proyecta la boca.

3.1.1.4 Seguimiento usando integrales proyectivas

El algoritmo para seguir el movimiento de una cara es parte de un proceso iterativo, en el que se recalcula la localización de la cara y las características faciales en cada imagen de la secuencia. En la inicialización se usa un algoritmo de detección de la cara, que en nuestro caso está disponible en las librerías utilizadas (las librerías OpenCV). Colocamos una elipse en la primera cara que localizamos, que señalamos visualmente con dibujando una cruz en cada ojo y una línea en la boca. La entrada en el seguidor es la cara del modelo, el estado del movimiento en la imagen previa y la nueva imagen. El modelo de la cara consiste en un conjunto de los modelos de la proyección de la cara entera y algunas partes de ella, donde la localización de las características faciales es conocida. Este modelo es obtenido usando la primera imagen de la secuencia, donde los ojos y la localización de la boca han sido dadas por el detector de caras. Básicamente, el algoritmo consiste en tres pasos principales en los que se estiman de forma independiente los parámetros de alineamiento vertical, horizontal y orientación de la nueva imagen. Este proceso es explicado a continuación:

- **Prepocesamiento**: Las localizaciones de la elipse y las características faciales en $i_{t-1}(x,y)$ son tomadas como las localizaciones esperadas en $i_t(x,y)$. En este preproceso, se extrae un rectángulo en $i_t(x,y)$ que contiene la elipse y es rotado respecto a la línea de los ojos. En realidad, el área es más grande que lo que envuelve la elipse. El tamaño de esta área adicional es un porcentaje del tamaño del modelo, denotado como $r_{tolerance}$. La parte interior

de la elipse se segmenta, y se toma como la entrada para el alineamiento vertical. En la Ilustración 7 se puede apreciar la fase de la preprocesamiento:

a) Cara muestreada para computar el modelo de cara.

b) Localización esperada de la elipse y las características faciales en una nueva imagen, usando localizaciones en la imagen previa.

c) Región de la cara extraída del paso "b" usando $r_{tolerance}$ = 25%.

d) Región de la cara envuelta y segmentada según el modelo.

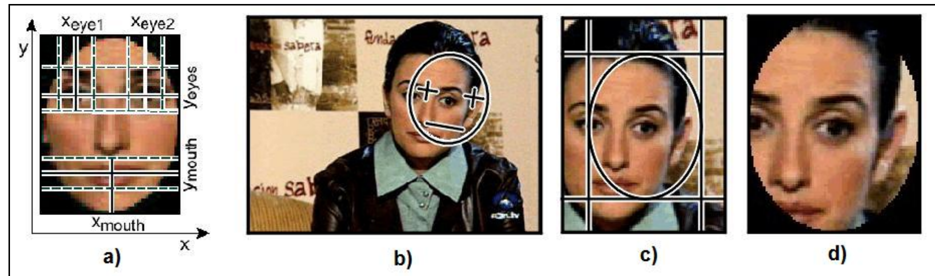


Ilustración 7. Preprocesamiento.

- **Alineamiento vertical**: En este paso, usado la proyección vertical, la traslación vertical y la escala de la cara en la nueva imagen es estimada. Primeramente, se calcula la proyección vertical de la región segmentada de la cara, P_{VCARA}. Esta señal es alineada respecto de la proyección vertical del modelo (M_{VCARA}, V_{VCARA}). Finalmente los parámetros de alineamiento *(d, e)* son usados para alinear la cara verticalmente.

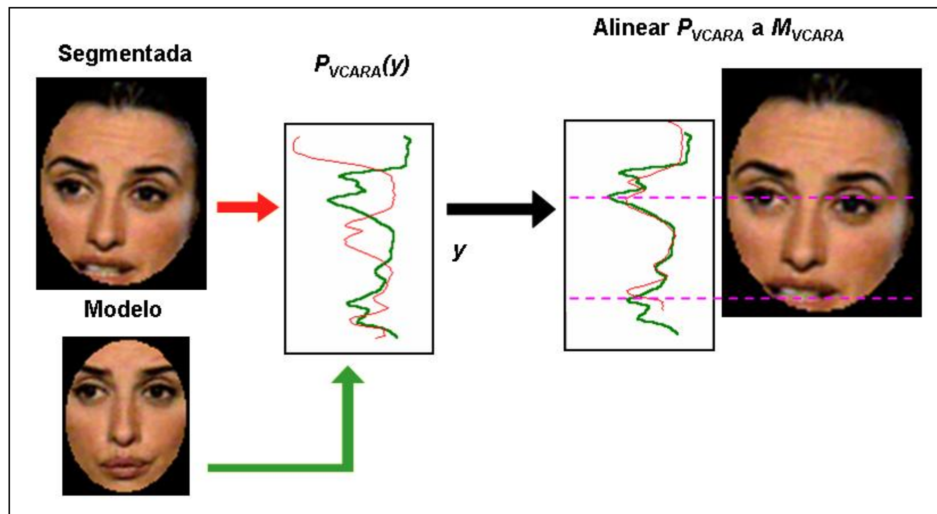


Ilustración 8. Alineamiento vertical.

- **Alineamiento horizontal**: Después del alineamiento vertical, las coordenadas Y de la boca y los ojos son conocidas, así que podemos segmentar las regiones de los ojos y la boca (incluida el área de tolerancia) de acuerdo al modelo. Usando las proyecciones horizontales de esas regiones, el algoritmo estima la traslación horizontal y la escala de la cara. Realmente, solamente la proyección horizontal de la región de los ojos, P_{HOJOS}, es considerada. P_{HOJOS} es normalmente más estable que P_{HBOCA}, produciendo un mejor alineamiento.

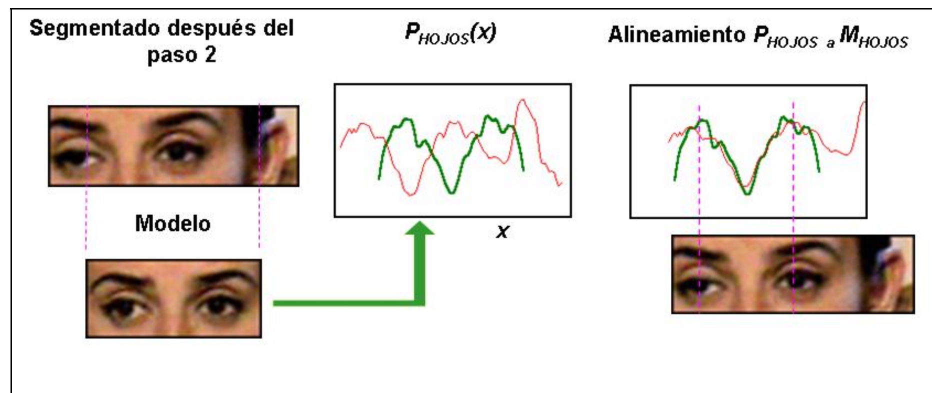


Ilustración 9. Alineamiento horizontal.

- **Estimación de la orientación**: La orientación es calculada a través de la estimación de la línea de los ojos. Mientras que en los pasos previos suponíamos que los ojos estaban localizados en línea recta, aquí las coordenadas Y de los ojos derecho e izquierdo son estimadas independientemente. Después de realizar los dos pasos anteriores, podemos segmentar las regiones que denominamos OJO1 y OJO2. Alinearemos las proyecciones verticales de esas regiones y los valores de las Y de los ojos serán computados.

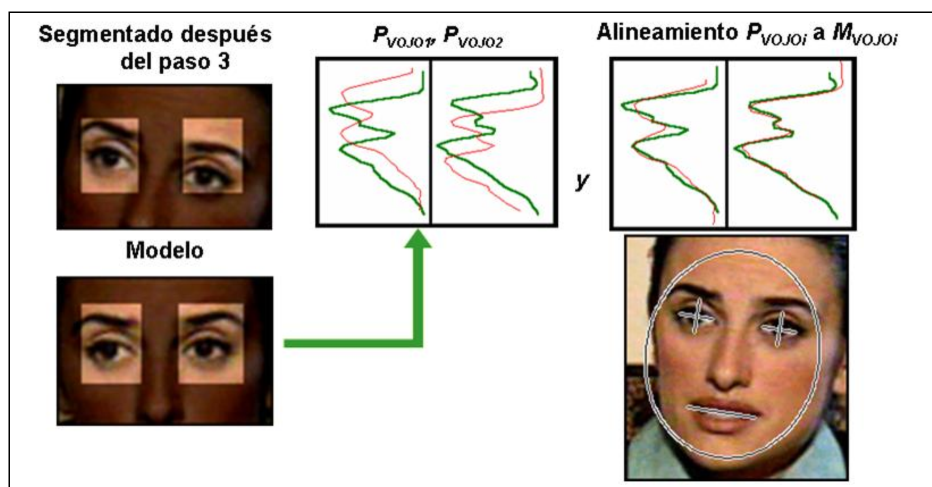


Ilustración 10. Estimación de la orientación.

3.1.2 Motor Gráfico

En este apartado vamos a describir brevemente los fundamentos matemáticos básicos necesarios para el desarrollo de un motor gráfico 3D. En este contexto, la posición de un punto viene dada por un vector de 3 valores: $P = (P_x, P_y, P_z)$. La descripción de una cámara móvil requiere un modelo más complejo. En concreto, definimos la posición y la orientación relativa del personaje, al sistema de coordenadas de nuestro mundo, usando cuatro vectores de cámara: vector desplazamiento, vector arriba, vector vista y vector de posición. Estos vectores se muestran en la Ilustración 11.

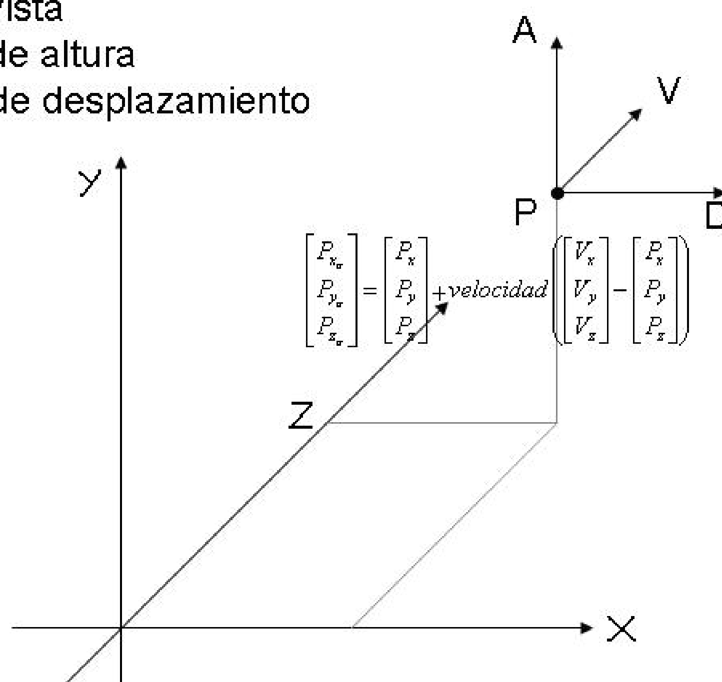


Ilustración 11. Los vectores de la cámara definiendo la posición y la orientación relativa al mundo.

De esta forma, con estos cuatro vectores describimos la cámara, y podemos definir las seis operaciones siguientes:

- Rotar alrededor del vector desplazamiento (PITCH).
- Rotar alrededor del vector arriba (YAW).
- Rotar alrededor del vector vista (ROLL).
- Moverse alrededor del vector desplazamiento.
- Moverse por el vector de altura.
- Moverse a lo largo del vector vista.

Todas estas operaciones consisten en moverse alrededor de los tres ejes y girar alrededor de ellos. A continuación vamos a describir los cálculos necesarios en cada uno de estos tipos de transformaciones.

Moverse a lo largo del vector vista (Avanzar o retroceder)

Quizá esta sea la acción que más veces se producirá durante la ejecución del programa. Lo que hacemos es partir de una posición inicial y calculamos la orientación en la que estamos mirando. Según la velocidad con la que vayamos, se incrementará o disminuirá la posición con mayor rapidez. Una velocidad negativa indicará que estamos retrocediendo y una positiva indicará que vamos caminando hacia delante. Si retrocedemos, la vista no cambiará; se mantendrá, con lo que iremos de espaldas. La fórmula para calcular la nueva posición viene dada por:

$$\begin{bmatrix} P_{x_a} \\ P_{y_a} \\ P_{z_a} \end{bmatrix} = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} + velocidad\left(\begin{bmatrix} V_x \\ V_y \\ V_z \end{bmatrix} - \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}\right)$$

Ecuación 3. Posición destino al andar.

La fórmula para calcular la nueva vista viene determinada por:

$$\begin{bmatrix} V_{x_a} \\ V_{y_a} \\ V_{z_a} \end{bmatrix} = \begin{bmatrix} V_x \\ V_y \\ V_z \end{bmatrix} + velocidad\left(\begin{bmatrix} V_x \\ V_y \\ V_z \end{bmatrix} - \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}\right)$$

Ecuación 4. Nueva vista al andar.

Se puede observar que el movimiento avance/retroceso es siempre en dirección hacia donde se está mirando.

Moverse por el vector de altura (arriba o abajo)

Para cambiar la altura por la que caminamos tendremos que calcular las nuevas posiciones del vector altura. La fórmula para calcular la nueva altura es:

$$\begin{bmatrix} A_x \\ A_y \\ A_z \end{bmatrix} = \left(\begin{bmatrix} V_x & V_y & V_z \end{bmatrix} - \begin{bmatrix} P_x & P_y & P_z \end{bmatrix}\right) * \begin{bmatrix} 0 & -A_z & A_y \\ A_z & 0 & -A_x \\ -A_y & A_x & 0 \end{bmatrix}$$

Ecuación 5. Calculo del cambio de altura.

La fórmula para calcular la nueva posición viene dada por:

$$\begin{bmatrix} P_{x_a} \\ P_{y_a} \\ P_{z_a} \end{bmatrix} = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} + velocidad\begin{bmatrix} A_x \\ A_y \\ A_z \end{bmatrix}$$

Ecuación 6. Posición destino al movernos en altura.

La fórmula para calcular la nueva vista viene determinada por:

$$\begin{bmatrix} V_{x_a} \\ V_{y_a} \\ V_{z_a} \end{bmatrix} = \begin{bmatrix} V_x \\ V_y \\ V_z \end{bmatrix} + velocidad\begin{bmatrix} A_x \\ A_y \\ A_z \end{bmatrix}$$

Ecuación 7. Vista final al desplazarnos en altura.

Moverse por el vector desplazamiento (izquierda o derecha)

Necesitamos poder desplazarnos lateralmente a lo largo del mundo, por lo que tenemos que calcular los nuevos valores para el vector desplazamiento. Lo único que no variará es la altura. Una velocidad positiva indica que nos movemos hacia la derecha, mientras que una negativa, que nos movemos hacia la izquierda. A mayor velocidad, mayor rapidez. El desplazamiento lateral es siempre perpendicular al vector vista. La fórmula para calcular los nuevos valores del vector desplazamiento es la siguiente:

$$\begin{bmatrix} D_x \\ D_y \\ D_z \end{bmatrix} = \left(\begin{bmatrix} V_x & V_y & V_z \end{bmatrix} - \begin{bmatrix} P_x & P_y & P_z \end{bmatrix} \right) * \begin{bmatrix} 0 & -A_z & A_y \\ A_z & 0 & -A_x \\ -A_y & A_x & 0 \end{bmatrix}$$

Ecuación 8. Vector de desplazamiento lateral

La fórmula para calcular la nueva posición viene dada por:

$$\begin{bmatrix} P_{x_a} \\ P_{y_a} \\ P_{z_a} \end{bmatrix} = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} + velocidad \begin{bmatrix} D_x \\ D_y \\ D_z \end{bmatrix}$$

Ecuación 9. Posición final cuando nos desplazamos lateralmente.

La fórmula para calcular la nueva vista viene determinada por:

$$\begin{bmatrix} V_{x_a} \\ V_{y_a} \\ V_{z_a} \end{bmatrix} = \begin{bmatrix} V_x \\ V_y \\ V_z \end{bmatrix} + velocidad \begin{bmatrix} D_x \\ D_y \\ D_z \end{bmatrix}$$

Ecuación 10. Vista final al moverse lateralmente.

Rotar alrededor del vector desplazamiento (PITCH)

Es la rotación respecto al eje X. Aquí lo único que cambia es el vector de la vista. La dirección con la que gira sobre el eje X viene determinada por el signo de la velocidad y la rapidez por el valor absoluto de esta. El signo positivo indica que se mueve hacia arriba y el signo negativo hacia abajo. El ángulo de giro viene determinado por la variable θ. La fórmula para calcular la nueva vista viene determinada por:

$$\begin{bmatrix} V_{x_a} \\ V_{y_a} \\ V_{z_a} \end{bmatrix} = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} + \left(\begin{bmatrix} V_x & V_y & V_z \end{bmatrix} - \begin{bmatrix} P_x & P_y & P_z \end{bmatrix} \right) * velocidad \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & sen(\theta) \\ 0 & -sen(\theta) & \cos(\theta) \end{bmatrix}$$

Ecuación 11. Vista final en rotación sobre el eje X.

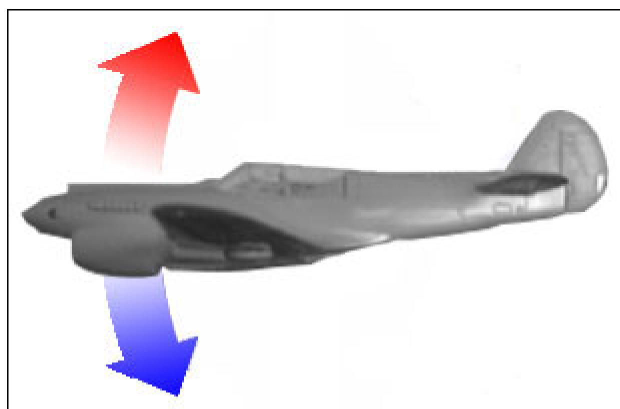


Ilustración 12. Rotación sobre el eje X (PITCH).

Rotar alrededor del vector arriba (YAW).

Es la rotación respecto al eje Y. Los valores de los vectores Posición, Altura y Desplazamiento no cambian. La dirección con la que gira sobre el eje Y viene determinada por el signo de la velocidad y la rapidez por el valor absoluto de esta. El signo positivo indica que se mueve hacia izquierda y el signo negativo hacia la derecha. El ángulo de giro viene determinado por la variable θ. La fórmula para calcular la nueva vista viene determinada por:

$$\begin{bmatrix} V_{x_a} \\ V_{y_a} \\ V_{z_a} \end{bmatrix} = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} + \left(\begin{bmatrix} V_x & V_y & V_z \end{bmatrix} - \begin{bmatrix} P_x & P_y & P_z \end{bmatrix} \right) * velocidad \begin{bmatrix} \cos(\theta) & 0 & -sen(\theta) \\ 0 & 1 & 0 \\ sen(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

Ecuación 12. Vista destino al rotar sobre el eje Y.

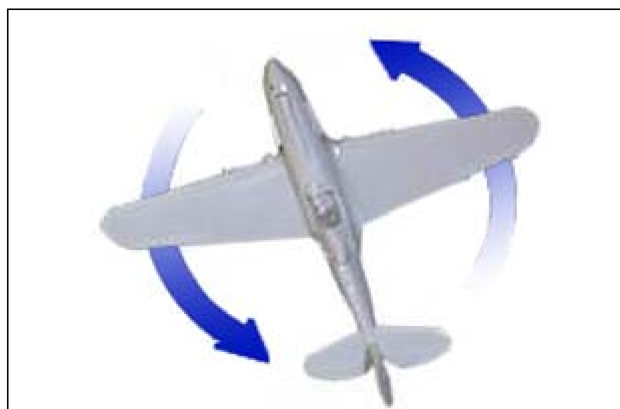


Ilustración 13. Rotación sobre el eje Y (YAW).

Rotar alrededor del vector vista (ROLL)

Es la rotación respecto al eje Z. Al igual que en las otras dos rotaciones lo único que cambia son los valores del vector vista. La dirección con la que gira sobre el eje Z viene determinado por el signo de la velocidad y la rapidez por el valor absoluto de esta. El signo positivo indica que gira hacia la izquierda y el signo negativo hacia la derecha. El ángulo de giro viene determinado por la variable θ. La fórmula para calcular la nueva vista viene determinada por:

$$\begin{bmatrix} V_{x_a} \\ V_{y_a} \\ V_{z_a} \end{bmatrix} = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} + \left(\begin{bmatrix} V_x & V_y & V_z \end{bmatrix} - \begin{bmatrix} P_x & P_y & P_z \end{bmatrix} \right) * velocidad \begin{bmatrix} \cos(\theta) & sen(\theta) & 0 \\ -sen(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Ecuación 13. Vista destino al rotar sobre el eje Z.

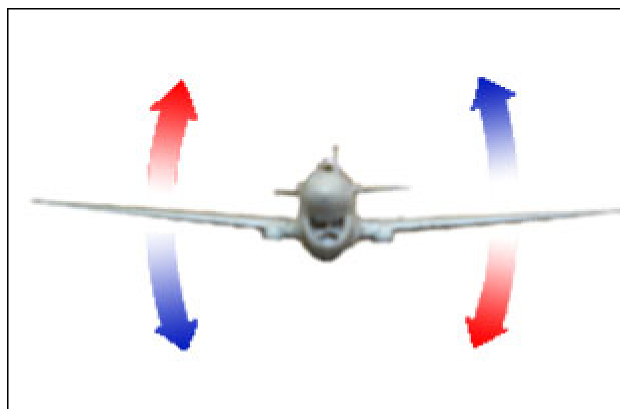


Ilustración 14. Rotación sobre el eje Z.

3.1.3 Colocar objetos en el mundo

En los sistemas de síntesis de imágenes se utilizan matrices de 4 x 4 para representar transformaciones de rotación, escala y posición. Todos los modelos de objetos están formados por puntos. Cuando queremos aplicar transformaciones sobre cada uno de estos objetos haremos operaciones matemáticas para generar los nuevos puntos donde va a ir el nuevo objeto.

Traslación

Consiste en coger la posición de un punto del plano y multiplicarlo por la matriz de traslación T(p). Con esta técnica podemos colocar cualquier objeto en la zona del mapa donde queramos que esté. Un ejemplo de traslación se puede ver en la siguiente figura:

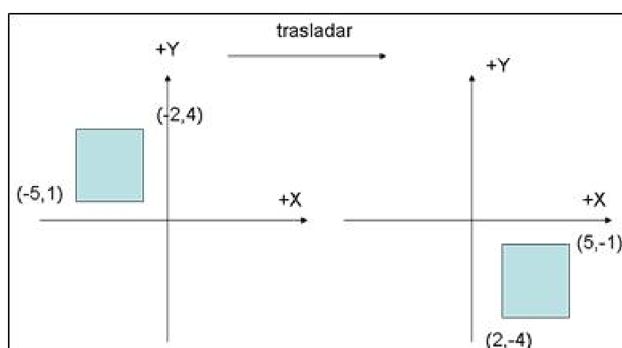


Ilustración 15. Ejemplo de una traslación.

Si representamos un punto en el mundo como (x,y,z,1) la fórmula que determina la nueva posición del punto viene determinada por la siguiente fórmula, siendo P_x, P_y, y P_z, las unidades que me quiero mover en el eje X, Y o Z respectivamente:

$$\begin{bmatrix} P_{x_a} \\ P_{y_a} \\ P_{z_a} \end{bmatrix} = \begin{bmatrix} X & Y & Z & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ P_x & P_y & P_z & 1 \end{bmatrix}$$

Ecuación 14. Traslación de un punto.

Rotación

Podemos rotar cualquier objeto con un ángulo en radianes alrededor de cualquier eje, usando alguna de las fórmulas que se describen a continuación. Un ejemplo de una rotación se puede ver en la Ilustración 16:

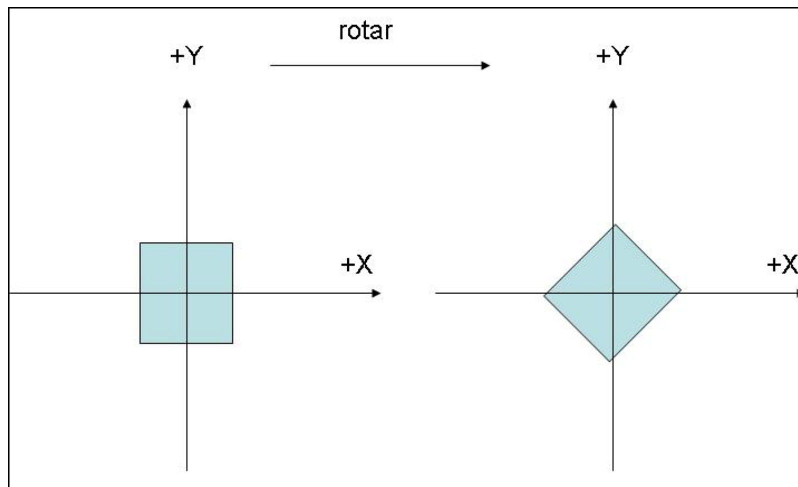


Ilustración 16. Ejemplo de una rotación de 45º

Cogemos un punto y lo multiplicamos por la matriz de rotación que le corresponda según el eje X, Y o Z. En las fórmulas siguientes θ es el ángulo de giro que vendrá en radianes y *X*, *Y* y *Z* representan las coordenadas del punto que queremos girar.

$$\begin{bmatrix} P_{x_a} \\ P_{y_a} \\ P_{z_a} \end{bmatrix} = \begin{bmatrix} X & Y & Z & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & sen(\theta) & 0 \\ 0 & -sen(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ecuación 15.Rotación de un punto sobre el eje X.

$$\begin{bmatrix} P_{x_a} \\ P_{y_a} \\ P_{z_a} \end{bmatrix} = \begin{bmatrix} X & Y & Z & 1 \end{bmatrix} * \begin{bmatrix} \cos(\theta) & 0 & -sen(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ sen(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ecuación 16. Rotación de un punto sobre el eje Y.

$$\begin{bmatrix} P_{x_a} \\ P_{y_a} \\ P_{z_a} \end{bmatrix} = \begin{bmatrix} X & Y & Z & 1 \end{bmatrix} * \begin{bmatrix} \cos(\theta) & sen(\theta) & 0 & 0 \\ -sen(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ecuación 17. Rotación de un punto sobre el eje Z.

Escalar

Además de girar y posicionar una figura, a veces tenemos que escalar un objeto o un escenario para que tenga unas dimensiones acordes con el mundo que tenemos alrededor. Escalar

un objeto es hacerlo más grande o más pequeño. Un ejemplo de escalado se puede ver en la siguiente figura:

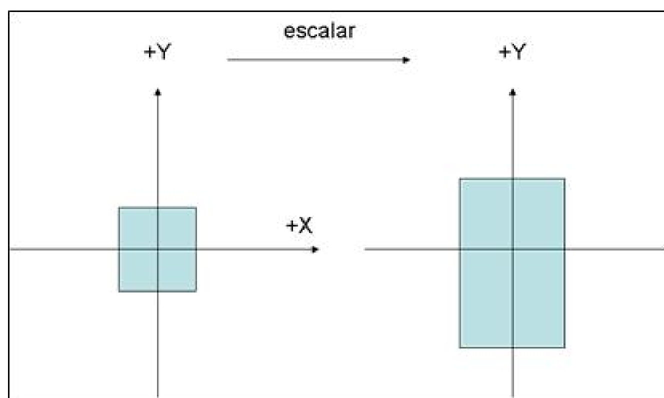


Ilustración 17. Ejemplo de escalar.

Para hallar las coordenadas del nuevo punto después de escalar hay que aplicar la siguiente formula en la que *X*, *Y* y *Z* representan las coordenadas del punto que queremos escalar y q_x, q_y y q_z el valor del escalado. Si el valor es mayor que uno, se hace más grande y si es menor de uno, disminuye.

$$\begin{bmatrix} P_{x_a} \\ P_{y_a} \\ P_{z_a} \end{bmatrix} = \begin{bmatrix} X & Y & Z & 1 \end{bmatrix} * \begin{bmatrix} q_x & 0 & 0 & 0 \\ 0 & q_y & 0 & 0 \\ 0 & 0 & q_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ecuación 18. Escalar un punto.

Combinación de transformaciones

Cuando queremos aplicar una transformación tendremos que multiplicar primero la matriz de traslación por la de rotación y luego por la de escalado. Un ejemplo sería:

$$\begin{bmatrix} P_{x_a} \\ P_{y_a} \\ P_{z_a} \end{bmatrix} = \begin{bmatrix} X & Y & Z & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ P_x & P_y & P_z & 1 \end{bmatrix} * \begin{bmatrix} \cos(\theta) & 0 & -sen(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ sen(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} q_x & 0 & 0 & 0 \\ 0 & q_y & 0 & 0 \\ 0 & 0 & q_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ecuación 19. Composición de traslación, escala y rotación.

3.1.4 Luces

Para generar un escenario es necesario poner focos de luz en la escena para que no nos quedemos en la penumbra. Entre los tipos de luces podemos elegir entre estas cuatro:

Luz ambiental: Es la luz que no viene de ninguna dirección en particular. Tiene una fuente, pero los rayos de luz han rebotado por la escena y han perdido direccionalidad. Los objetos iluminados por la luz ambiente son igualmente luminosos en todas sus superficies y direcciones.

Punto de luz: Esta luz tiene una posición en el espacio y emite luz en todas las direcciones.

Luz direccional: El dispositivo de luz no tiene una posición en el espacio pero emite rayos en la dirección especificada.

Cono de luz: Este tipo de luz es muy similar a una linterna.

Para el desarrollo del entorno gráfico, se han usado la luz ambiental debido a que tiene menos coste computacional, aunque en el código están implementadas las rutinas para poder poner las otras luces.

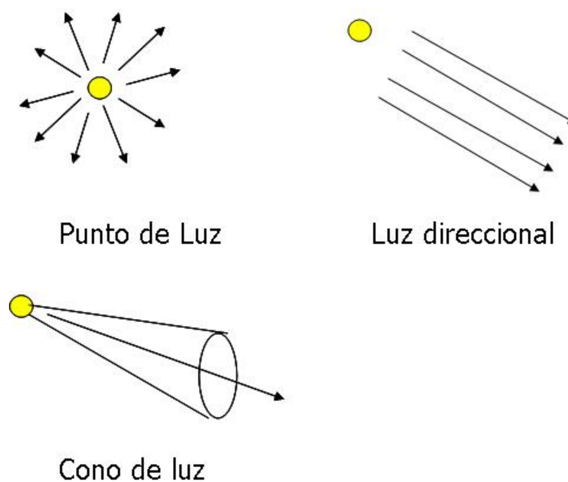


Ilustración 18. Tipos de luz.

3.1.5 Texturas

Las texturas son imprescindibles para conseguir un alto grado de realismo en las imágenes generadas. El uso de ellas para el terreno es el elemento clave a la hora de conseguir unas imágenes hermosas y creíbles. Sin embargo, conseguir una textura que se adapte al terreno con naturalidad no es una tarea sencilla.

Una vez que hemos conseguido obtener una textura convincente, debemos mapearla en el terreno. Para ello, se debe poder cargar desde un fichero.

3.1.6 Sonidos

Para manejar los sonidos, hemos utilizado DirectSound. Los sonidos que utilizamos están en formato RAW. Los hemos convertido de formato wav a raw, mediante el programa Cool Edit Pro. Este formato lo hemos usado porque no usa cabecera y ocupa menos tamaño. Cada sonido tiene un canal para reproducirse. En el caso de que se quiera reproducir el sonido, el programa mirará a que el canal esté libre, si es así, se reproducirá el sonido, en otro caso no hará nada. Esto se hace para no escuchar por ejemplo dobles pisadas. La frecuencia del sonido que utilizamos es 16 KHz.

3.1.7 Aceleración del renderizado

Vamos a usar dos técnicas principales que se van a explicar a continuación: la técnica de portales o casillas y el frustum culling o clipping.

3.1.7.1 Portales

Esta técnica se basa en dividir el mapa del mundo virtual en regiones o casillas y controlar qué ocurre en cada una de ellas. Para el proyecto lo que hacemos es utilizar para las regiones cuadriláteros unidos unos con otros. Esto lo usamos fundamentalmente por dos motivos: el primero es para decir qué objetos son visibles desde donde estamos y así ganamos velocidad al no tener que para no renderizar toda la escena; y el segundo motivo es para controlar cuándo chocamos contra un objeto. En la Ilustración 19 se muestra un espacio de un mapa, en el lado izquierdo se puede ver como se vería solo con las paredes y al lado derecho, cómo quedaría después de subdividirlo en regiones. Cada color representa una región. Hay otras formas de subdividirlo, pero siempre hay que procurar que se haga con el menor número de polígonos posibles ya que de esa forma se mejora el rendimiento.

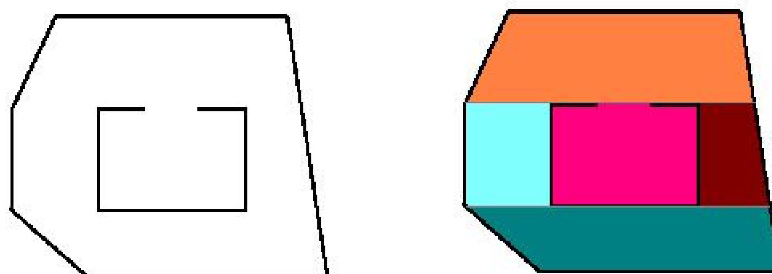


Ilustración 19. Representación de regiones de un mapa

3.1.7.2 Frustum

Cuando se manda un polígono al *pipeline* gráfico, este pasa por una serie de etapas, pudiendo acabar dibujándose en pantalla, o bien terminar en una etapa anterior debido a que no es visible. Aunque el polígono no se pueda ver, debe recorrer varias etapas del *pipeline*, consumiendo recursos del hardware gráfico. Si en una escena tenemos diez millones de polígonos y tan solo son visibles unos cientos, se habrá malgastado mucho tiempo y recursos inútilmente.

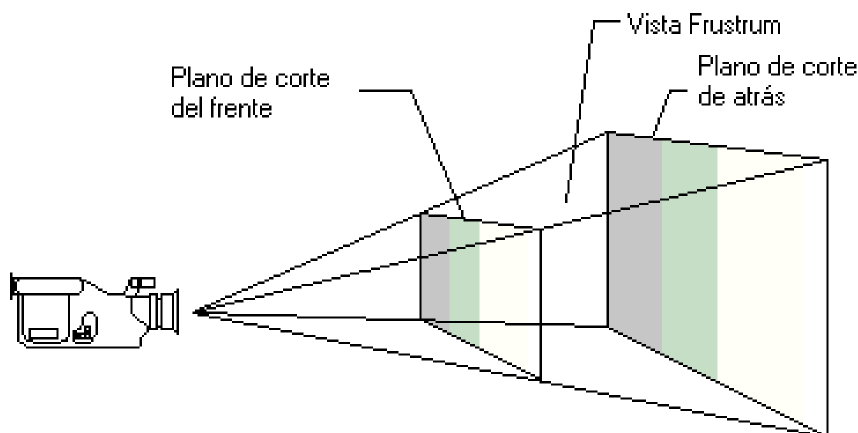


Ilustración 20. El frustum.

La vista frustum está formada por el campo visual de la cámara (en ingles fov, filed of view) que esta comprendido entre dos planos de corte, uno anterior y otro posterior. En las ilustraciones 20 y 21 se muestra esta idea.

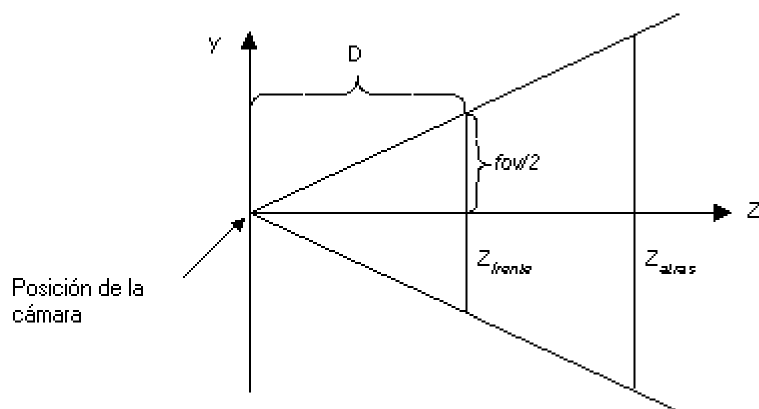


Ilustración 21. Vista frustum

En la Ilustración 21, la variable D es la distancia desde la cámara al origen del espacio que se definió en la última parte de la tubería de geometría -la transformación de vista-. Este es el espacio alrededor del cual disponemos los límites de la vista frustum.

Como nosotros poseemos información de alto nivel sobre los objetos de la escena, podemos realizar optimizaciones de forma que consigamos un aumento de la velocidad sin perder calidad de imagen. Una optimización que da muy buenos resultados es la técnica conocida como *view frustum culling* (recorte del volumen vista), y se ideó para solventar el problema de dibujar todos los polígonos de la escena aunque no sean visibles desde nuestra cámara.

Cada vez que le pasamos un polígono a la tarjeta gráfica, esta comprueba si está dentro del volumen de vista, y no continúa procesando el triangulo de no ser así. Como nosotros sabemos que todos los triángulos de una determinada zona de la escena pertenecen a un objeto (por ejemplo, una torre) podemos calcular la envolvente de dicho objeto y comprobar si dicha envolvente está dentro del volumen de vista. En otro caso, no mandamos los polígonos a la tarjeta gráfica puesto que no serán visibles.

El problema ahora reside en cómo averiguar si una envolvente está dentro del volumen de vista. El volumen de vista está compuesto por seis planos (*front*, *near*, *top*, *far*, *right*, *left*), y dependiendo del tipo de proyección utilizada puede ser o bien una caja, o bien una pirámide truncada, según se aplique una proyección ortográfica o una perspectiva, respectivamente.

Una forma de saber si la envolvente está dentro del volumen vista, es calculando las ecuaciones de los seis planos que forman el volumen vista, y comprobando si la envolvente está dentro de la región de espacio que forma el volumen vista.

Un método rápido para obtener las ecuaciones de los planos que delimitan el volumen de vista en coordenadas del mundo sería el siguiente: Como los objetos de la escena son estáticos, se pueden precalcular las envolventes de todos los objetos en las coordenadas del mundo, y así, en cada fotograma se comprueba si los puntos que forman la envolvente están dentro del volumen de vista. Si no es así, no hace falta mandar la geometría del objeto para que se dibuje en pantalla puesto que no se verá en pantalla.

Normalmente los grupos de polígonos que queremos comprobar se agrupan dentro de envolventes o volúmenes envolventes; como por ejemplo una esfera (*Bounding Sphere*) o una caja alineada con los ejes de coordenadas (*Axis Aligned Bounding Box* o *AABB*). Si la esfera o la caja no están fuera del *frustum*, dibujaremos los polígonos en cuestión. En nuestro caso, se han calculado las envolventes de los modelos usando cajas 3D. La forma de calcular la caja que envuelve a un modelo en coordenadas locales es muy sencilla, ya que simplemente hay que ir recorriendo todos los vértices que forman el modelo y se va actualizando el valor máximo y mínimo en cada dimensión, obteniendo así los dos puntos que definen perfectamente la envolvente en coordenadas de mundo.

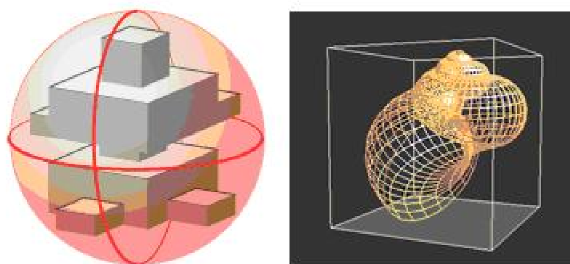


Ilustración 22. A la izquierda un ejemplo de Bounding Sphere y a la derecha un ejemplo de Bounding Box.

3.2 Herramientas utilizadas

En este apartado describimos las herramientas utilizadas en los diversos aspectos de la resolución del proyecto y la justificación de las elecciones realizadas. El proyecto ha sido desarrollado bajo entorno Windows, porque uno de los objetivos era trabajar con las herramientas más habituales en aplicación tipo de programación de videojuegos.

3.2.1 Elección del lenguaje

Entre las características deseables para implementar el proyecto, partimos de la base de que el lenguaje a utilizar debería ser orientado a objetos puesto que es el paradigma más adecuado para la obtención de software correcto, de calidad, reutilizable y extensible. Estos factores son muy importantes puesto que es probable que el programa pueda evolucionar y surjan necesidades o aportaciones futuras al trabajo.

Tiene que ser un lenguaje que permita la programación en tiempo real y además poder utilizar las librerías DirectX, IPL y OpenCV. Esto reduce drásticamente el abanico de posibilidades colocando a C++ como el lenguaje mas adecuado para el desarrollo de videojuegos

Las características más importantes de C++ son:

- Programación orientada a objetos, con lo que podremos agrupar en clases las distintas funcionalidades.
- Velocidad. Este es un motivo en que la mayoría de aplicaciones en tiempo real del mercado se basen en este lenguaje.

3.2.1.1 Visual Studio .NET 2003

Microsoft® Visual Studio® .NET es la herramienta de desarrollo multilenguaje más completa para construir e integrar rápidamente aplicaciones y servicios Web XML.

Hemos utilizado esta herramienta porque incorpora las siguientes características:

- Soporta C++ y DirectX

- Es una de las herramientas más usadas para crear videojuegos.
-
- Constituye un modelo sencillo, flexible y basado en estándares para integrar, ampliar y publicar aplicaciones.
- Incluye buenas herramientas de edición, depuración, etc. y cientos de componentes.
- Aprovecha una seguridad incorporada fiable.

3.2.2 Intel® Image Processing Library

La "Image Processing Library" (IPL) es un conjunto de librerías de Intel que implementan diferentes funciones de procesamiento de imágenes. Está optimizada para procesadores de Intel, aunque funcionan con cualquier procesador compatible con Pentium.

La IPL se implementa en una serie de DLL que se enlazan dinámicamente a nuestra aplicación en tiempo de ejecución.

3.2.3 OpenCV

OpenCV significa Intel® Open Source Computer Vision Library. Es una colección de funciones de C y algunas clases de C++ que implementan algunos algoritmos de procesamiento de imágenes y de Visión por computador. Son las más usadas y extendidas en el campo de la visión artificial. Se pueden descargar las librerías en la dirección Web http://www.sourceforge.net/projects/opencvlibrary.

Hemos usado la librería de Intel para la detección de caras y para la captura de video procedente de la WebCam o de un archivo AVI.

3.2.4 3D Studio Max

3D Studio Max es una aplicación basada en el entorno Windows (9x/NT) que permite crear tanto modelados como animaciones en tres dimensiones (3D) a partir de una serie de vistas o visores (planta y alzados). La utilización de 3D Studio Max permite al usuario la fácil visualización y representación de los modelos, así como su exportación y salvado en otros formatos distintos del que utiliza el propio programa. En particular hemos usado la versión 3D Studio Max 5.0 para la realización del proyecto.

3.2.5 DirectX 9.0

DirectX: es una API creada por Microsoft, para permitir a los desarrolladores de juegos, programar eficientemente bajo Windows, extremo que era improbable antes de la llegada de DirectX, ya que tecnologías como GDI no permitían acceder al hardware tan directamente como lo hace DirectX. Hemos usado esta versión de directX porque gran parte del material teórico utilizaba esta versión y teníamos el inconveniente que muchas funciones cambian su interfaz en anteriores versiones.

Algunas de las principales ventajas de DirectX son: el acceso directo al hardware, la rapidez y la facilidad de uso. Los componentes de DirectX son:

- **Direct3D**: proporciona una librería que permite optimizar la renderización de objetos 3D tomando las ventajas del hardware existente, o mediante software si no se encontrara el hardware adecuado.

- **DirectSound**: Como su nombre indica, la librería DirectSound maneja todo lo relacionado con el sonido, proporcionando tecnologías de *mixing*(mezcla), sonido estéreo y 3D, aprovechando al máximo las capacidades del hardware.

- **DirectPlay**: se encarga de las características multijugador de los juegos.

- **DirectInput**: Proporciona una interfaz para el manejo de entrada y salida, como teclado, mouse, joystick, etc.

Capítulo 4

Diseño y resolución del proyecto

4.1 Estructura del proyecto

El desarrollo del proyecto ha sido descompuesto en cuatro grandes apartados: el control de sonido, la generación gráfica, el tratamiento de caras y el control de las entradas. Cada uno de estos se verá reflejado en el programa en el controlador correspondiente, implementado mediante las clases adecuadas

El controlador de sonido se encarga de cargar todos los sonidos del juego e inicializar DirectSound para poder reproducirlos. Tendremos un canal por cada sonido que queramos reproducir para no repetirlo más de una vez.

El controlador de gráficos, se va a encargar de controlar lo que hay que pintar en la pantalla. También va a tener un sistema de detección de colisiones, así como técnicas para dar mayor rapidez al pipeline de Direct3D. Además se encarga de inicializar Direct3D.

El controlador de tratamiento de caras tendrá la facultad de detectar dónde hay una cara, hacerle un seguimiento, saber encontrarla si se pierde, interpretar los movimientos que están continuamente produciéndose. También tiene que ser capaz de cambiar el modo de entrada de vídeo entre webcam y archivo AVI.

El controlador de entradas se encargará de servir de interfaz entre el teclado y nuestro programa.

4.2 Detección y seguimiento de caras

El punto de partida para la detección y seguimiento de caras es el código escrito por Ginés García Mateos, profesor de Informática de la Universidad de Murcia. Dicho código está basado en OpenCV y en las librerías IPL y consiste en detectar caras en una imagen y posteriormente hacer un seguimiento mediante integrales proyectivas.

A continuación he tenido que adaptarlo a Visual C++ porque el código estaba escrito para Borland y había algunas funciones que no podía utilizar. Una vez concluida esta adaptación, he tenido que hacer alguna pequeña modificación para que funcione en mi entorno, como por ejemplo que solo detecte una cara.

4.3 Detección de movimientos

Aunque las librerías antes referidas resuelven el problema de la detección de caras, queda abierto el problema de la estimación del movimiento y posición 3D de las caras. Entre las principales aportaciones de este proyecto está la propuesta de un método sencillo para la estimación de la posición y giro a partir de los resultados del seguimiento. La propuesta es explicada en este apartado.

Tenemos que detectar todos los movimientos que puede tener la cabeza en las tres dimensiones del espacio. Una vez que tenemos localizada la cara, tendremos tres puntos claramente distinguibles, que son los ojos y el punto intermedio de la boca, y tres integrales proyectivas que corresponden a la proyección de los ojos, la boca y la vertical de la cara. A partir de este punto, con toda la información que tenemos, intentaremos conseguir con la mayor fiabilidad posible todos los movimientos y rotaciones de la cara y al mismo tiempo de forma rápida y robusta.

Como veremos más adelante, algunos movimientos los podemos determinar solo con una imagen, pero otros los tendremos que concretar con referencia a algún valor inicial. A continuación pasaré a describir en pseudocódigo el algoritmo para determinar los valores iniciales.

```
Calcular la distancia entre los ojos y la boca.

Determinar el punto medio de la posición X entre los ojos.

Determinar el punto medio de la posición Y entre los ojos.

Determinar el punto medio entre el punto interocular y la boca, solo en coordenadas de la Y.
```

4.3.1 Movimiento en el eje X

Para intentar detectar este movimiento, se calcula la distancia euclídea entre los ojos y se determina el punto medio. Una vez que lo tenemos, nos valdremos del valor que teníamos almacenado previamente en el modelo inicial, para calcular nuevamente la distancia en píxeles entre estos dos puntos, pero esta vez sólo lo hacemos entre las coordenadas del eje X. Una vez que tenemos calculada esta distancia, tenemos que saber en que sentido se mueve; para ello, lo determinamos mirando el signo de esta distancia. El valor de la coordenada X en la imagen, será mayor cuanto mas a la derecha esté el punto, pero tenemos que tener en cuenta que la imagen de la WebCam está invertida, por lo que cuando esta distancia sea positiva, significará que se mueve hacia la izquierda y si es negativo, hacia la derecha. Además, el movimiento se puede ver afectado por el ruido que tiene la imagen y las posibles variaciones que implica esto. Entonces, establecemos un umbral que tenemos que sobrepasar para saber que se detecta un movimiento. Este umbral se define como un porcentaje de la distancia interocular del modelo inicial. Respecto a la velocidad con la que realizamos este movimiento, será mayor cuanto más grande sea en valor absoluto esta distancia. En la Ilustración 23 se muestra un ejemplo de este movimiento.

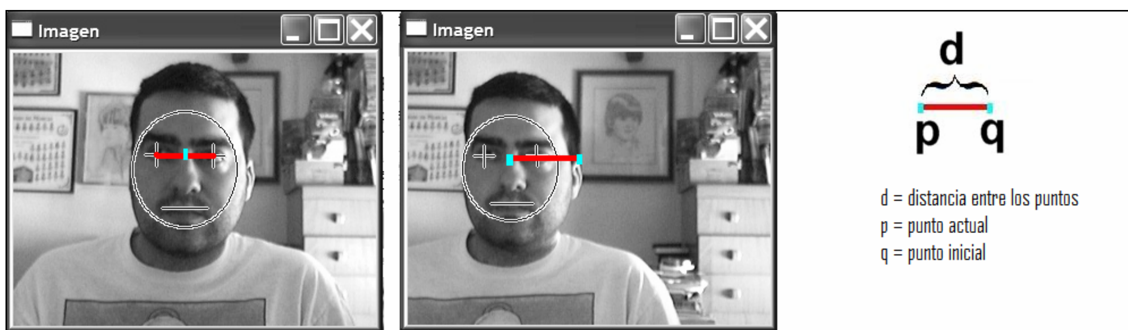


Ilustración 23. Medición del desplazamiento en X.

El pseudocódigo para estimar este movimiento se muestra a continuación:

```
Calcular la diferencia en X entre el punto interocular que había guardado al
inicio y la posición actual.

        SI esta diferencia sobrepasa cierto umbral ENTONCES

                SI esta diferencia es positiva ENTONCES

                        Desplazar a la izquierda.

                SINO

                        Desplazar a la derecha.

                FINSI

        SINO

                No hay movimiento en el eje X.

FINSI
```

4.3.2 Movimiento en el eje Y

El movimiento en el eje X corresponde a los posibles cambios de altura de la cabeza, para lo cual nos valemos de las posiciones de los ojos y la boca. Primero tendremos que calcular el punto intermedio entre los ojos y después calculamos el punto medio entre este punto intermedio y la boca. A continuación, tenemos que saber el espacio que se ha desplazado la cara, con lo que tenemos que calcular la distancia entre este punto actual, con el del modelo que previamente hemos determinado. Esta distancia se hace sólo respecto de la coordenada del eje Y. A diferencia del movimiento anterior, en este caso da igual que la imagen de la WebCam se capture de izquierda a derecha o viceversa. Ahora que sabemos que se produce movimiento en el eje y, lo que tenemos que hacer es determinar si el movimiento es hacia arriba o abajo. Para realizar esta función, lo que tenemos que hacer, es mirar el signo de la distancia entre el punto actual que hemos calculado y el inicial. Si este resultado es positivo, indicará que nos movemos hacia abajo y en caso contrario hacia arriba. La intensidad de este movimiento dependerá del valor absoluto de esta distancia, es decir, a mayor valor, mayor rapidez de movimiento. Aquí, al igual que ocurre en el movimiento anterior, debemos tener en cuenta el ruido de la imagen. Definiremos un umbral que consiste en calcular la distancia entre el punto interlineal de los ojos y la boca y a esta distancia le asignamos un

porcentaje. Si existe un movimiento, tiene que desplazarse como mínimo este porcentaje para que empecemos a movernos. En la Ilustración 24 se muestra un ejemplo de este movimiento.

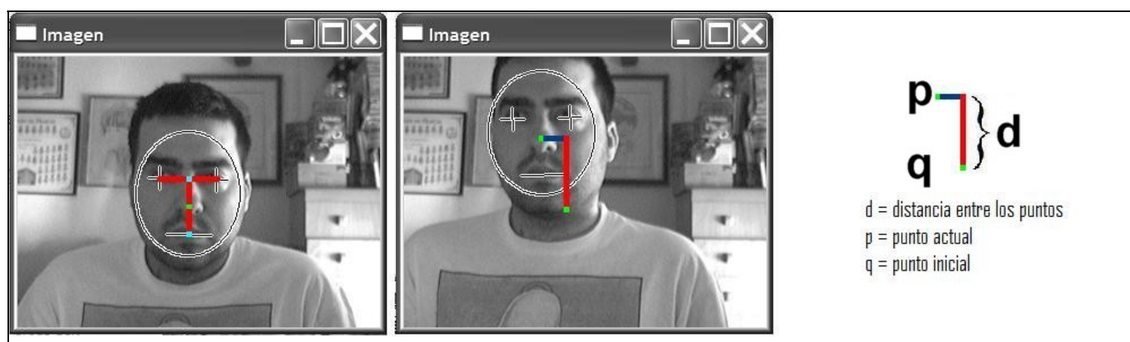


Ilustración 24. Medición del desplazamiento en Y.

A continuación se muestra el pseudocódigo que estima este movimiento:

```
Calcular la distancia entre las coordenadas Y del punto interocular y la boca
de la posición actual y el mismo punto pero que había detectado al iniciar.

SI esta distancia sobrepasa cierto umbral ENTONCES

        SI esta distancia es positiva ENTONCES

                Desplazar hacia abajo.

        SINO

                Desplazar hacia arriba.

        FINSI

SINO

        No hay movimiento en el eje Y.

FINSI
```

4.3.3 Movimiento en el eje Z

La estimación precisa en el eje Z es compleja, puesto que en la imagen no tenemos información de profundidad. Sin embargo, proponemos un método sencillo para aproximarla. Puesto que la cara tiene un tamaño fijo, su tamaño percibido en la imagen depende inversamente de la distancia a la cámara. Por lo tanto, este movimiento es el mas sencillo de calcular porque sólo tendremos que tener en cuenta la distancia interocular, para poder determinar si la cara avanza o retrocede en el espacio. Para comenzar cogemos las coordenadas de los ojos y calculamos la distancia euclídea. Si esta aumenta respecto a la que tenemos previamente calculada del modelo inicial, significará que nos estamos acercando a la cámara y por el contrario, si disminuye, significa que nos alejamos. Para determinar la intensidad, también dependerá del valor de la distancia. Pero aquí, a diferencia de los otros movimientos, cuando retrocedes la distancia no disminuye de la misma forma que cuando te acercas, sino que depende del cociente entre distancias. Un valor de 1 significa que no hay movimiento. Para solucionar el ruido, aquí también utilizaremos la misma idea

que en los movimientos anteriores, de manera que la distancia que hemos recorrido tiene que ser mayor que un umbral que previamente habremos definido.

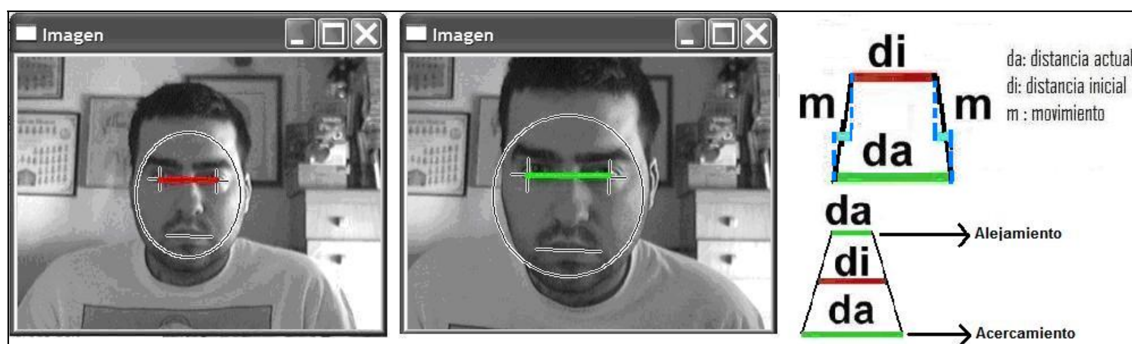


Ilustración 25. Medición del desplazamiento en Z.

El funcionamiento de esa estimación se explica en el siguiente pseudocódigo:

```
Calcular el cociente de la distancia interocular actual y la que habíamos
calculado al inicio.

SI este cociente está por encima de cierto umbral ENTONCES

        SI este cociente es mayor que 1 ENTONCES

                Desplazar hacia delante.

        SINO

                Desplazar hacia atrás.

        FINSI

SINO

        No hay movimiento en el eje Z

FINSI
```

4.3.4 Giro en el eje X

Este es el punto mas problemático en el proceso de las detecciones porque si giramos mucho la cabeza, el seguimiento de la cara se pierde, con lo que no detectamos ningún movimiento. La causa es que tanto el detector de caras como el mecanismo de seguimiento utilizados suponen caras con una posición frontal o casi frontal. Debido a esta razón, es difícil darle intensidad al giro, con lo que he optado por utilizar sólo detección de movimiento y usar una intensidad de giro constante. Suponemos que no hay focos de luz, porque tenemos el problema que podemos producir resultados incorrectos por la aparición de distintos focos de luz. Por este motivo, este movimiento es menos fiable que los movimientos explicados hasta ahora. La detección se hace observando la integral proyectiva de los ojos del modelo con la de la instancia. Buscamos donde está el mínimo, que teóricamente debería corresponder a la posición horizontal de la nariz respecto a los ojos y miramos el desplazamiento que este tiene a la derecha o la izquierda. Esta distancia en píxeles determina la intensidad de giro; a mayor distancia, mayor giro. Aquí también se utiliza un

umbral en número de píxeles para que cuando el movimiento que detectemos sea menor que ese umbral, no produzca ningún giro. Un ejemplo de este giro se muestra en la Ilustración 26.

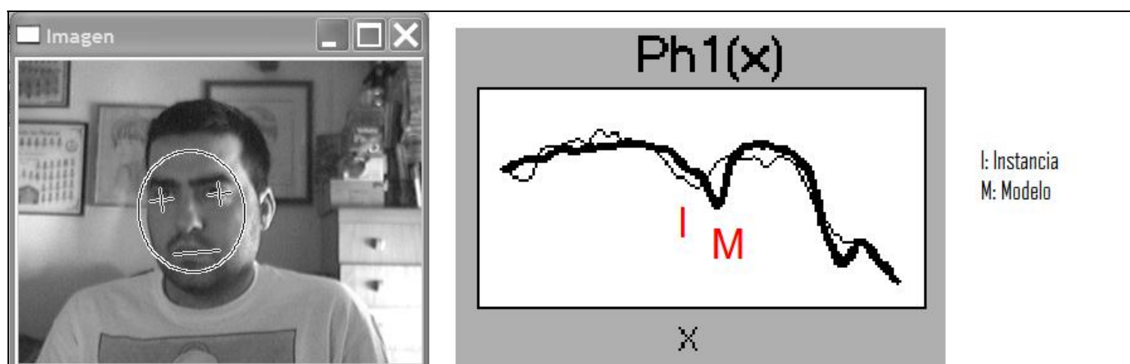


Ilustración 26. Integral proyectiva de los ojos.

El funcionamiento de esto se puede ver con el siguiente pseudocódigo:

```
Rectificar la señal de la instancia.

Buscar el mínimo valor de la señal de la instancia entre las posiciones de los ojos

Calcular la diferencia entre la posición que hemos calculado y la que procede de buscar en la señal rectificada del modelo, entre la posiciones de los ojos, el valor mínimo, que previamente tendremos calculado.

SI la diferencia es menor que cierto umbral ENTONCES

        Girar a la izquierda.

SINO SI la diferencia es mayor que cierto umbral ENTONCES

                Girar a la derecha.

        FINSI

FINSI
```

4.3.5 Giro en el eje Y

Cuando giramos la cabeza hacia arriba o hacia abajo, se produce un cambio en la integral proyectiva vertical de la cara. Para saber si se ha detectado un giro en una dirección, tenemos que mirar la señal del modelo y calcular donde está el mínimo entre la posición de los ojos y la boca en la señal. Luego miramos en la señal de la instancia la variación en esa posición, para saber qué cambio se ha producido. Se sabe si va hacia arriba dependiendo del valor de la diferencia entre las dos señales en ese punto. Se establece un punto para el cual si es menor el valor que esa cota, significa que va hacia abajo y para arriba ocurre exactamente lo mismo pero tiene que ser mayor que una determinada cota establecida. Un ejemplo de este giro lo podemos ver en la Ilustración 27.

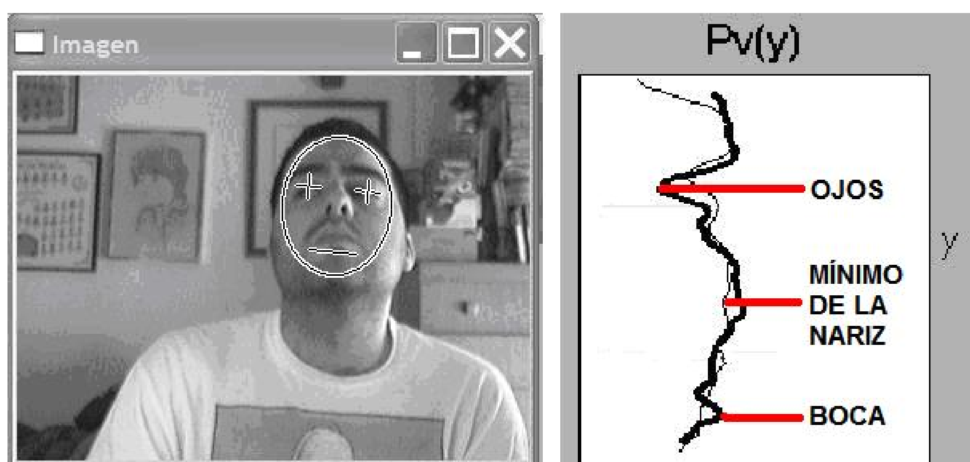


Ilustración 27. Integral proyectiva de la vertical de la cara.

El comportamiento se expresa en el pseudocódigo de abajo:

```
Rectificar la señal de la instancia.

Buscar el mínimo valor de la señal entre la posición de los ojos y la boca.

Calcular la diferencia entre el valor de la señal de la instancia, en la posición
donde hay un mínimo en la señal del modelo rectificado, entre la posición
de los ojos y la boca, con el valor del mínimo en este último punto, que
previamente tendremos calculado.

SI la diferencia es menor que cierto umbral ENTONCES

        Girar hacia abajo.

SINO SI la diferencia es mayor que cierto umbral ENTONCES

                Girar hacia arriba.

        FINSI

FINSI
```

4.3.6 Giro en el eje Z

Para determinar este movimiento nos basamos en la información que tenemos sobre la posición de los ojos. Calculamos el ángulo que forma el vector que componen la unión de los dos puntos de los ojos y la vertical. Después le restaremos 90 grados a este ángulo y con este resultado comprobaremos si es positivo o negativo, con lo que nos moveremos a la derecha o izquierda respectivamente. Lógicamente el movimiento será más intenso el movimiento cuando mayor sea el ángulo en valor absoluto. Para evitar el ruido solamente detectamos movimiento en Z cuando el ángulo que tomamos sea mayor que un ángulo umbral que previamente hayamos definido. Un ejemplo de este giro se muestra en la Ilustración 28.

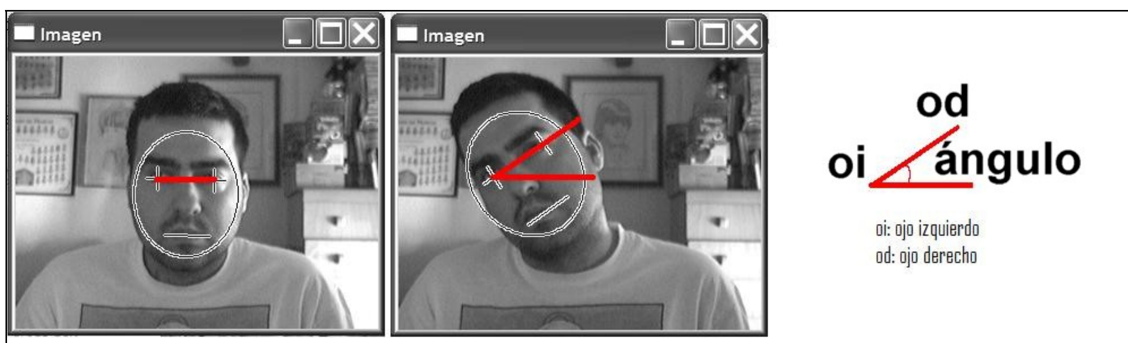


Ilustración 28. Ángulo de inclinación de la cara.

El pseudocódigo de giro en el eje Z es el siguiente:

```
Calculo el ángulo que forman los ojos con la horizontal.

Calcular el arcotangente del cociente entre la diferencia de la Y de los ojos y
la diferencia de las X de estos.

A este ángulo le restamos 90º

 SI esta diferencia sobrepasa cierto umbral ENTONCES

        SI esta distancia es positiva ENTONCES

                Giro hacia la derecha.

        SINO

                Giro hacia la izquierda.

        FINSI
SINO

        No hay giro en el eje Z

FINSI
```

4.4 Priorizar los movimientos

En principio, si detectamos todos los movimientos al mismo tiempo, los procesos que hemos descrito anteriormente deberían funcionar sin problemas. Pero ocurre que si realizamos ciertos movimientos con la cabeza se pueden mal interpretar. Para evitar esta colisión de movimientos, usamos una heurística que hemos desarrollado. Hemos podido comprobar que al realizar movimientos, unos son más críticos y otros tienen menos interacción con los demás. El movimiento que tiene más interacción con los demás, es el giro en el eje Z, con lo que llegamos a la conclusión de no hacer nada cuando detectamos un giro sobre este eje. Además hemos observado que, al ir hacia delante y hacia atrás, sufría una pequeña variación en el eje Y, con lo que deducimos que cuando se detecta un movimiento en el eje X, no debe haber ningún movimiento en Y.

4.5 Intentando reducir el ruido en las detecciones

Para reducir el ruido en las detecciones, se ha aplicado un filtro gaussiano en la escala temporal, lo cual implica y he tenido que retrasar la detección de movimientos para producir un mejor resultado. Utilizamos para esto un buffer de 5 valores. El programa no comienza hasta que se llena este buffer. Ello implica que la imagen que se trata actualmente no sea la misma que la que se está capturando.

4.6 Entorno virtual 3D

Para poder desarrollar un entorno virtual en 3D lo primero que he tenido que hacer es crear un mapa en 2D y subdividirlo en cuadriláteros, lo más grandes posible. De esta manera cambiaremos menos de casilla y reducimos el espacio en la memoria, al tener menos cosas almacenadas. Una vez que obtenemos estos polígonos, tenemos que sacar todos los puntos que forman los cuadriláteros. Con estos puntos nos vamos a un programa de creación de entornos 3D, en nuestro caso 3DStudio, e iremos punto por punto, creando los vértices y uniéndolos entre sí, para crear las paredes del entorno. Una vez finalizado, le añadimos las texturas para que realmente tenga buena calidad visual. Para la creación de los túneles, se han utilizado semicilindros para poder dar una forma redondeada. El suelo ha sido creado mediante planos y le añadimos texturas. Por último, para crear el cielo, existen dos técnicas bastante utilizadas: usar una semiesfera o bien una caja y poner el mapa dentro, por supuesto añadiéndole una textura de cielo. He optado por la caja, porque para conseguir el efecto de niebla era más conveniente, puesto que no se podría distinguir todas las zonas.

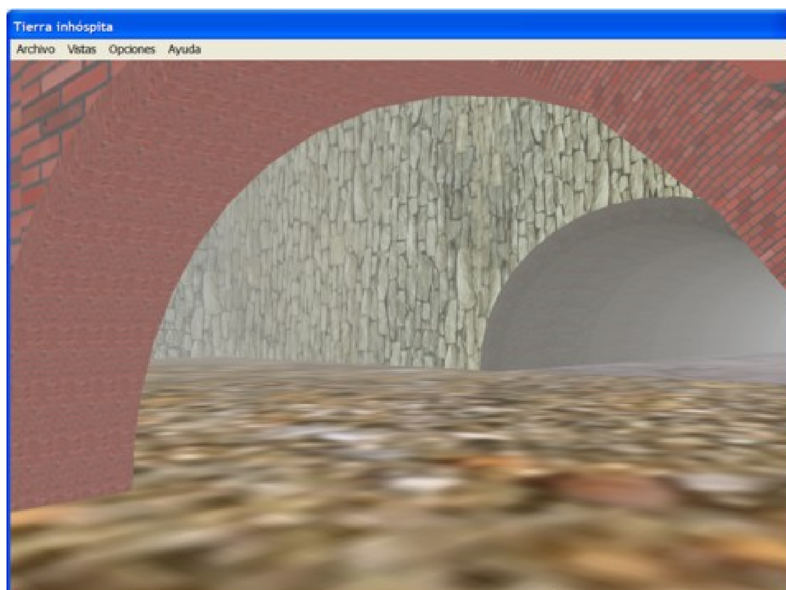


Ilustración 29. Ejemplo de un mapa del entorno 3D, con túnel y efecto de niebla.

Ahora lo que queda es darle realismo colocando objetos tridimensionales en la pantalla. Para esto he utilizado modelos gratuitos que he encontrado en Internet. He procurado que los objetos estuvieran formados por los menos polígonos posibles. Para eso, se ha utilizado un plugin del 3D Studio, para reducirlos al mínimo, pero sin que apenas se note la diferencia gráficamente. A continuación se van a mostrar algunos de los objetos que podemos ver dentro del entorno virtual. En la Ilustración 30 se muestra un ejemplo de estas figuras.

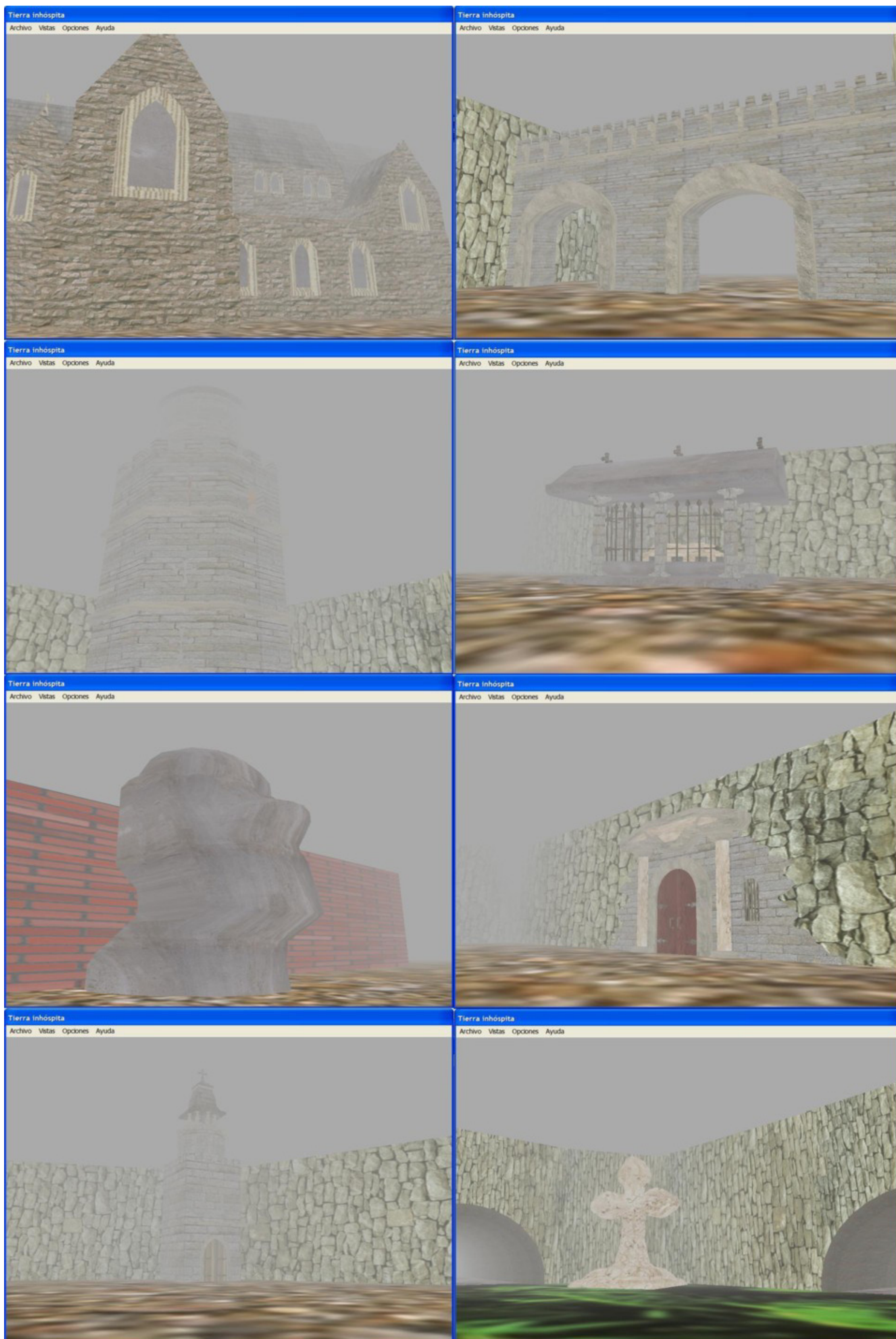


Ilustración 30. Objetos gráficos 3D del entorno de Tierra inhóspita.

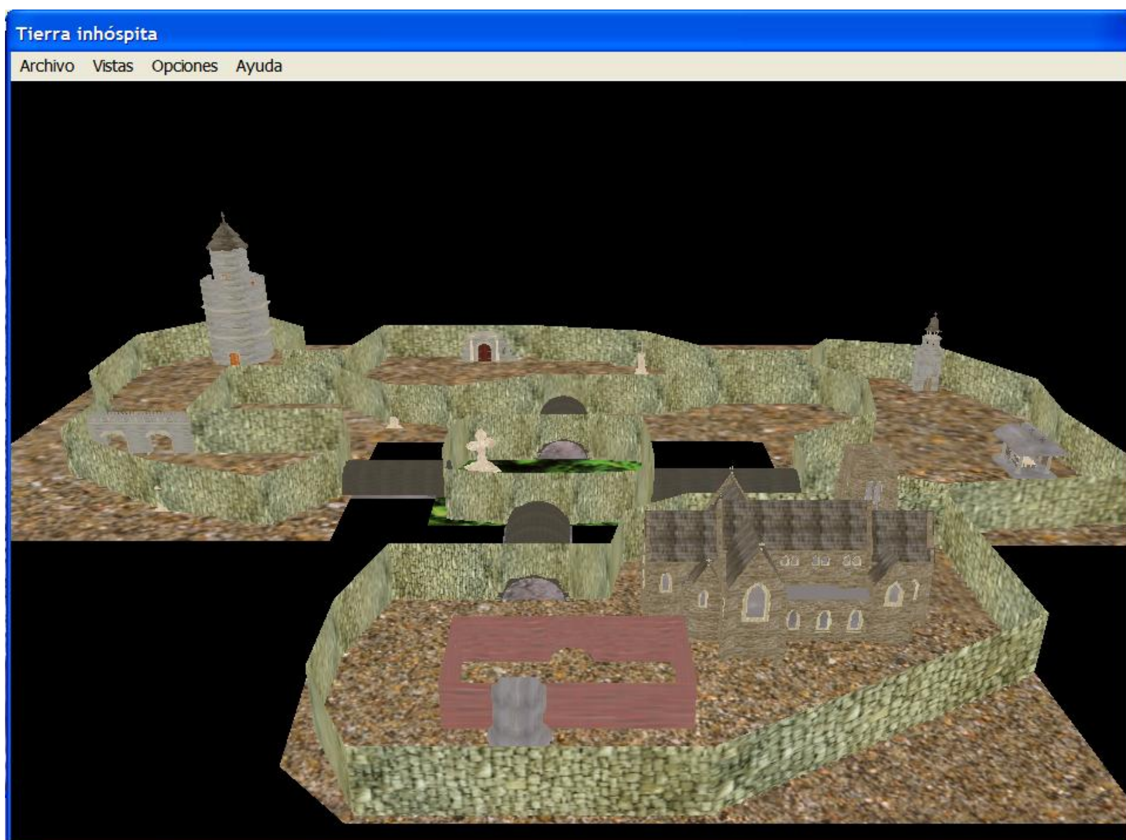


Ilustración 31. Vista de pájaro del mundo virtual de Tierra inhóspita.

4.7 Interacción con el mundo virtual

4.7.1 Moviéndonos por el entorno

Aunque el uso de DirectX permite acelerar la generación gráfica, esté sólo se encarga de pintar los objetos en la posición del espacio donde nosotros le hayamos dicho, pero lógicamente, por sí solo no incorpora un motor y mucho menos un sistema para evitar las colisiones. Tenemos que desarrollar dos procesos que tienen que ir de la mano para poder dotar de realismo al entorno en el que estamos, como es movernos por la pantalla y saber a qué zonas podemos entrar y dónde estamos localizados en el mapa en cada momento.

Principalmente, los programadores gráficos usan dos técnicas para dar sensación de movimiento en los entornos virtuales. La primera es la de mover los objetos del mundo y que permanezca la cámara inmóvil y la segunda es la de mover ésta y que todos los objetos inmóviles del mundo permanezcan siempre en su sitio. Esta técnica es la que hemos usado, debido fundamentalmente a dos razones: la mayor rapidez de cálculo y permitir una visualización en primera persona, totalmente compatible con asociar la cámara a nuestra mirada.

4.7.1.1 Motor Gráfico

Lo que se ha explicado en el capítulo 3 sobre el motor gráfico está pensado para cualquier tipo de entorno, por lo que tenemos que adaptarlo a las características de nuestro entorno. Necesitamos cuatro vectores para tener siempre presente nuestra situación, que son: el vector desplazamiento, vector arriba, vector vista y vector de posición.

El vector de posición nos da nuestra situación exacta en el mapa. El vector vista nos dice hacia dónde estamos mirando. El vector de arriba nos da la altura de nuestro personaje. El vector de posición dice hacia dónde vamos si nos desplazamos hacia los lados.

Desplazamiento hacia delante o atrás.

Para desplazarnos hacia delante o hacia atrás hay que tener en cuenta las posiciones sólo de la X y de la Z y no la de la Y porque no nos movemos en altura. Se trata de un entorno en primera persona y no debemos permitir que se levante del suelo o bien vaya caminando hacia abajo. A continuación se muestra el pseudocódigo de andar hacia delante, para atrás sería exactamente igual pero en lugar de incrementar sería decrementar.

> Obtener la dirección donde miramos (vector vista)
>
> Incrementar la posición de X por la dirección X donde miramos por la intensidad de movimiento.
>
> Incrementar la posición de Z por la dirección Z donde miramos por la intensidad de movimiento.
>
> Incrementar la vista de X por la dirección X donde miramos por la intensidad de movimiento.
>
> Incrementar la vista de Z por la dirección Z donde miramos por la intensidad de movimiento.

Desplazamiento lateral.

En este movimiento ocurre una cosa muy similar que en el caso anterior, y que difiere del general en que no debemos permitir que se mueva del suelo, con lo que tenemos que impedir que se mueva la posición Y. A continuación se muestra el pseudocódigo de desplazarnos hacia la izquierda, para el lado derecho seria exactamente igual pero en lugar de incrementar sería decrementar:

> Obtener la dirección donde miramos (vector vista)
>
> Obtener el producto vectorial de la dirección donde miramos y la dirección de arriba.
>
> Incrementar la posición de X por la coordenada X del producto anterior por la intensidad de movimiento.
>
> Incrementar la posición de Z por la coordenada Z del producto anterior por la intensidad de movimiento.
>
> Incrementar la vista de X por la coordenada X del producto anterior por la intensidad de movimiento.
>
> Incrementar la vista de Z por la coordenada Z del producto anterior por la intensidad de movimiento.

Desplazamiento arriba o abajo.

En este caso queremos simular el movimiento de subir y bajar de altura, igual que haríamos poniéndonos de puntillas o con las rodillas. Lógicamente, habrá que ponerle un límite, puesto que no queremos que nuestro personaje levite, ni atraviese el suelo, con lo que tenemos que comprobar que se encuentre siempre en un intervalo. A continuación se muestra el pseudocódigo de subir y seguidamente el de bajar.

```
SI no se ha superado el limite superior de la altura (posición Y)

        Incrementar la posición de Y por la intensidad de movimiento.

        Incrementar la vista de Y por la intensidad de movimiento.

FINSI
```

```
SI no se ha superado el limite inferior de la altura (posición Y)

        Decrementar la posición de Y por la intensidad de movimiento.

        Decrementar la vista de Y por la intensidad de movimiento.

FINSI
```

Giros.

He implementado todos los giros en una función, a diferencia de lo expuesto en el capítulo 3 para hacer más potente este método, ya que permite girar en varias direcciones al mismo tiempo. Para implementar el giro hacia la izquierda o a la derecha tenemos que llamar a esta función pasándole como valor a las variables *velocidadx* y *velocidadz 0*. Cada variable velocidad representa la intensidad de giro en cada eje. El pseudocódigo del código de giro es el siguiente:

```
Calcular el seno del ángulo que queremos girar.

Calcular el coseno del ángulo que queremos girar.

Obtener la dirección donde miramos (vector vista)

Calcular la nueva posición de la coordenada X de la vista:
xVista = (Coseno + (1 - Coseno) * velocidadx) * XVistaVieja + ((1 - Coseno) *
velocidadx * velocidady - velocidadz * Seno)* YVistaVieja + ((1 - Coseno) *
velocidadx * velocidadz + velocidady * Seno) * ZVistaVieja.

Calcular la nueva posición de la coordenada Y de la vista:
yVista = ((1 - Coseno) * velocidadx * velocidady + velocidadz * Seno) *
XVistaVieja + (Coseno + (1 - Coseno) * velocidady) * YVistaVieja + ((1 - Coseno)
* velocidady * velocidadz - velocidadx * Seno) * ZVistaVieja.

Calculars la nueva posición de la coordenada Z de la vista:
zVista = ((1 - Coseno) * velocidadx * velocidadz - velocidady * Seno) *
XVistaVieja + ((1 - Coseno) * velocidady * velocidadz + velocidadx * Seno) *
YVistaVieja + (Coseno + (1 - Coseno) * velocidadz) * ZVistaVieja.
```

Para el giro de mirar arriba o abajo no he utilizado esta función porque quería poder limitar el movimiento para hacerlo realista. A continuación mostraremos el pseudocódigo del movimiento de subir y el de bajar respectivamente.

```
SI la diferencia entre la coordenada Y de la vista y la posición Y es menor que 1.

        Incrementar la vista por la intensidad del movimiento.

FINSI
```

```
SI la diferencia entre la coordenada Y de la vista y la posición Y es mayor que -1.

        Decrementar la vista por la intensidad del movimiento.

FINSI
```

4.7.1.2 Controlar las posiciones

Con DirectX hemos sido capaces de crearnos un mundo 3D pero no podemos interactuar directamente con él. Para ello hemos creado un sistema lógico que consiste, básicamente, en un plano bidimensional y un personaje que tiene dos coordenadas, que indican la posición de la X y de la Y, dentro del sistema. El mundo es bidimensional porque como no nos movemos en el eje Y suponemos que no vamos a chocar en esta dirección del espacio. Una vez que tenemos este plano, lo tenemos que subdividir en celdas o casillas. Las celdas que vamos a utilizar van a ser cuadriláteros debido a su facilidad de manejo, ya que fácilmente se puede descomponer cualquier mapa con esta figura geométrica. Por motivos de memoria y rendimiento, es preferible que el número de celdas sea lo menor posible. Un ejemplo de cómo funciona lo podemos apreciar en la Ilustración 32.

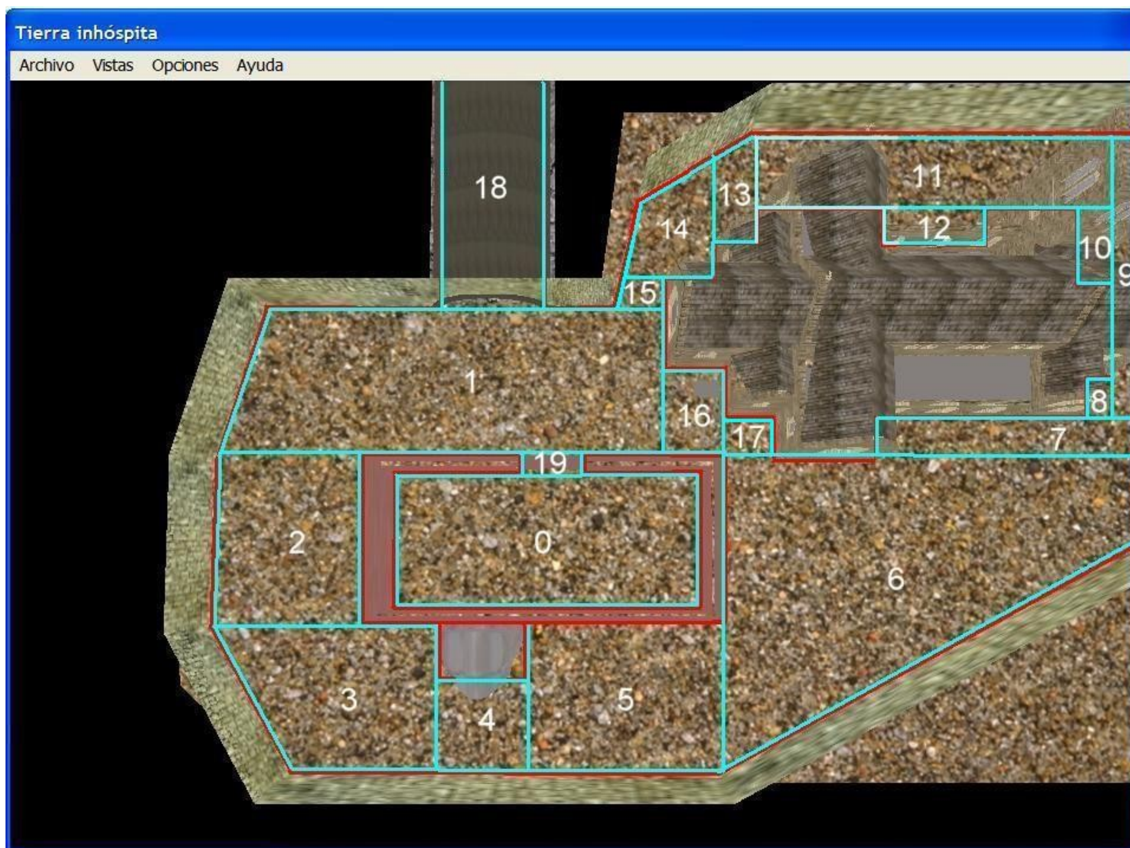


Ilustración 32. Trozo de mapa dividido en celdas.

Como se puede apreciar en la Ilustración 32, cada casilla tiene un número que la identifica y es único para cada una. La línea azul representa los límites de la casilla.

Una vez que tenemos todo el plano dividido en celdas, pasaremos a la siguiente parte que es localizar los puntos de intersección con otras celdas vecinas y los cuatro puntos que delimitan el cuadrilátero. Todos estos puntos se numeran con un identificador, para posteriormente poder tenerlos almacenados como información dentro de la casilla. En la Ilustración 33 se puede ver un ejemplo de cómo sería para la casilla número 1 del ejemplo de la Ilustración 32.

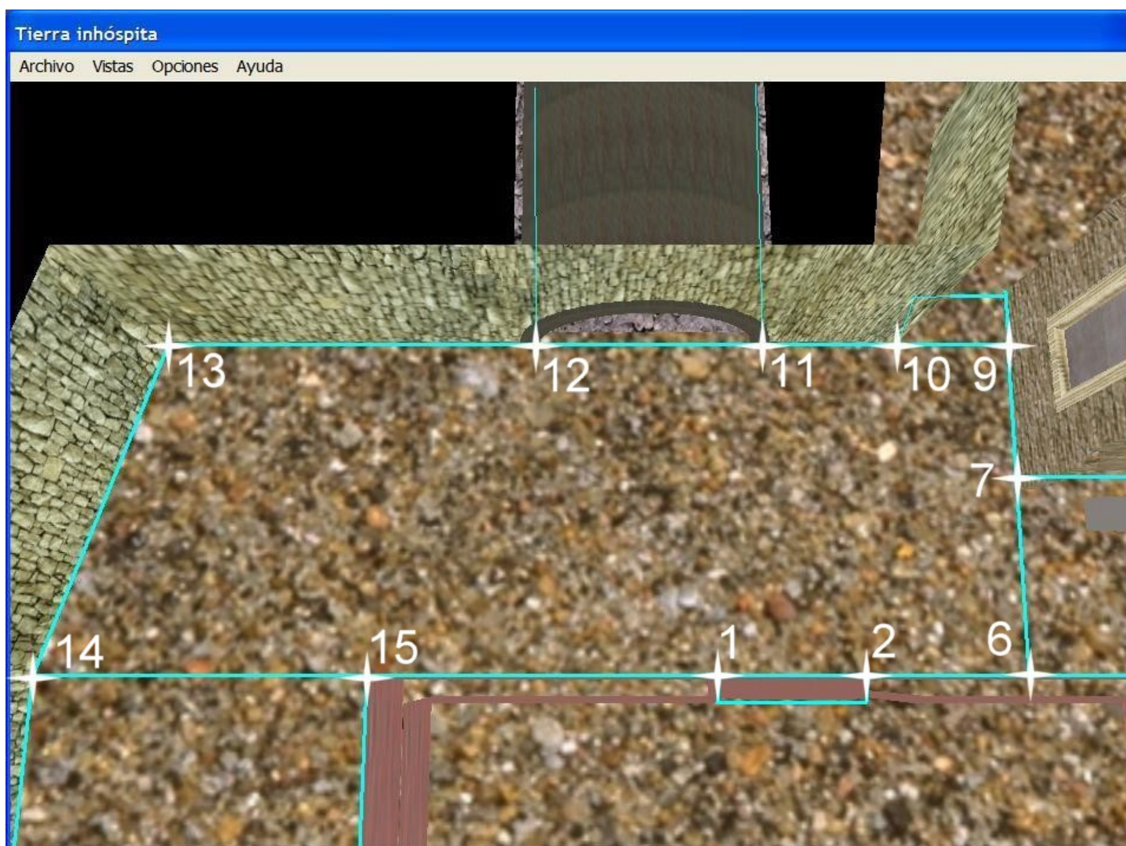


Ilustración 33. Puntos que forman la casilla 1.

Información de cada casilla

Cada casilla contendrá información sobre:

- Un número que la identifique y que debe ser único para cada una.
- Los cuatro puntos que definen las rectas que forman el cuadrilátero de la casilla.
- Información de cada subsegmento de la recta, para saber si es franqueable o bien contiene una pared o un objeto.
- Los objetos que son visibles desde nuestro campo de visión, es decir, desde esa casilla.
- Información sobre dónde están localizados cada uno de sus vecinos.

4.7.2 Movimientos dentro del plano lógico

Pueden ocurrir tres cosas cuando nos movemos por una celda o casilla: que lo hagamos internamente sin que nada ocurra, que choquemos con una pared o un objeto -cosa que explicaremos a continuación; y por último, que cambiemos de casilla.

Lo primero que tenemos que controlar es la manera de ver que nos movemos dentro de la casilla. Para resolver esto, lo que hacemos es considerar las cuatro rectas que forman el cuadrilátero y calcular la distancia de un punto, la posición del personaje, a esas rectas. Esto, teóricamente,

debería funcionar, pero realmente una distancia nunca puede ser negativa y no podemos saber en qué lado estamos, por lo que propondremos otra idea: desplazar las rectas de forma paralela, tantas unidades de longitud como permita la máxima cantidad que andamos en un paso, de manera que la de arriba se desplazará hacia arriba, la derecha hacia la derecha e igual para el resto. A continuación, comprobamos que no nos encontramos dentro de la región que hemos aumentado, con lo que si estamos dentro permanecemos en la casilla original y si hemos cambiado indicará que está en lo aumentado, es decir, que hemos chocado contra la pared.

Ahora tenemos que determinar a qué casilla vamos si cruzamos por un sitio accesible, siempre que estemos en la región aumentada. Elegimos el sitio dependiendo de la distancia menor. Calculamos el punto de corte dentro de la recta original del lado más cercano que delimita la casilla, que en la Ilustración 34 sería el punto donde se entrecorta con la línea azul. Una vez que tenemos esta información, miramos en la información de la celda para determinar si por este lado se puede acceder a otra casilla y, si es así, se cambia la información del personaje y se dice que está en la nueva casilla.

Ilustración 34. Movimiento en el mundo.

4.7.3 Detección de colisiones

Hay que evitar que nos salgamos del mapa o que podamos atravesar los objetos que interactúan en el entorno. Para resolver esto hay varias posibilidades, como puede ser envolver un objeto con una figura geométrica (se suelen usar cilindros, esferas o cajas) o bien utilizar un sistema por polígonos que consiste en que sólo podemos andar dentro del espacio en los polígonos que hayamos definido. Hemos optado por usar el sistema de coordenadas para detectar las colisiones, puesto que ya estaba hecho para definir las zonas del mapa. El mayor problema, surgió a la hora de decidir qué sistema aplicara a los objetos. La técnica de envolver los objetos con figuras geométricas me facilitaba la rapidez a la hora de diseñar el mapa, ya que no tenía que controlar las coordenadas de choque con cada punto, además de permitirme un menor número de casillas que me agiliza el trabajo. Pero tenía el inconveniente, bastante grande, que sólo podía meter figuras que fueran regulares a la hora de envolverlas en un objeto, con lo que vi claramente que era imposible usar este sistema, porque tenía que intentar acercarme a todos los resquicios del entorno y con esta técnica hubiera resultado bastante difícil hacerlo. Esta forma de detectar las colisiones se usa sobre todo en simuladores de coches o aviones. Finalmente la forma que he usado para resolver las colisiones

consiste en ajustar los cuadriláteros a las paredes y objetos. Para mostrar donde están las colisiones, vamos a ver un trozo del mapa, marcando en rojo donde están los objetos y las paredes.

Ilustración 35. Límites del mapa con paredes y objetos.

El funcionamiento de este método consiste en que cuando hacemos un movimiento calculamos donde nos movemos si no existieran colisiones. Entonces si cruzamos la casilla calculamos en que punto hemos cortado al polígono y allí miramos si existe pared. Si es así, impedimos que se cruce. Un ejemplo de esto, lo podemos ver en la Ilustración 36. Se producirá un choque siempre que pasamos por la línea roja.

Ilustración 36. Ejemplo de choque.

A continuación mostramos el pseudocódigo para detectar si chocamos o nos salimos por la pared de la izquierda.

```
Calcular la distancia para la izquierda

SI estamos fuera por la izquierda ENTONCES

        SI el punto de corte por la izquierda esta desbloqueado ENTONCES

                Buscar a qué casilla vamos

                Actualizar la casilla en la que estamos

        SINO

                VOLVER

        FINSI

FINSI
```

Habría que hacer lo mismo para las otras paredes. A continuación mostraremos como quedaría un movimiento, como el de caminar hacia delante, añadiéndole la detección de colisiones.

> Obtener la dirección donde miramos
>
> SI no hay obstáculos en el camino ENTONCES
>
> > Incrementar la posición de X por la dirección X donde miramos por la intensidad de movimiento.
> >
> > Incrementar la posición de Z por la dirección Z donde miramos por la intensidad de movimiento.
> >
> > Incrementar la vista de X por la dirección X donde miramos por la intensidad de movimiento.
> >
> > Incrementar la vista de Z por la dirección Z donde miramos por la intensidad de movimiento.
>
> FINSI

4.8 Pintar la escena

Ahora que ya disponemos de movimientos en el plano lógico pasaremos a explicar el funcionamiento de colocación de los objetos en la escena. Como hemos explicado anteriormente nuestro entorno va a ir cambiando solamente por la posición de la cámara, ya que es un entorno inanimado. Cambiamos los vectores de posición, vista y el de arriba, pero no es necesario el vector desplazamiento, porque éste sólo se usa para movernos lateralmente. Con estos vectores formamos la matriz *vista* (matriz que usa directX para posicionar la cámara), que la usamos para determinar dónde está nuestra cámara y qué es lo que vemos.

4.8.1 ¿Qué vemos?

Al estar nuestro mapa formado por celdas, tenemos todos los objetos y decorados que vemos dentro de la información de cada casilla. Por ejemplo, desde la casilla 1 se ve la iglesia, pero no se ve el tótem. Por consiguiente, si aplicamos esta funcionalidad, tenemos que ir recorriendo uno por uno todos los objetos de la casilla en la que estamos, e ir pintando objeto por objeto. Pero aquí existe el problema que se explicó en la sección 3.1.7.2 sobre el *frustum culling*: es preferible no pintar los objetos que no se ven. Para eso creamos una función frustum que calcula los seis planos para ver si el objeto está dentro de estos y así poder pintarlo. Para aumentar el rendimiento no comprobamos todos los objetos si se ven dentro del *frustum*, ya que hay decorados que se ven en todas las posiciones como puede ser el cielo.

4.8.2 Efecto de niebla

El efecto de niebla se utiliza sobre todo para que no se vea cómo se crean los polígonos e impedir la vista de objetos que están muy alejados y así no tenemos que dibujarlos. El problema radica en que para ver esta niebla es necesario que la tarjeta gráfica soporte toda la funcionalidad de DirectX 9. He utilizado la niebla lineal, que consiste en que la densidad aumenta uniformemente con la distancia.

A continuación mostramos el pseudocódigo de la función que muestra los objetos en la pantalla.

```
Calcular el vector posición.

Calcular el vector vista.

Calcular el vector arriba.

Crear la matriz vista a partir de los tres vectores anteriores.

Mover la cámara al valor de la vista.

Calcular el frustum.

Coger todos los objetos que vemos y calcular cuáles son visibles en nuestro frustum.

Dibujar la niebla.
```

4.9 Sonidos

En todo entorno virtual que se precie debería haber sonido para que el nivel de realismo sea mucho mayor. Aquí voy a utilizar dos sonidos, que van a ser el de las pisadas y el de colisión para cuando se produzca un choque con algún objeto de la pantalla. Para poder integrar los sonidos y coordinarlos con las imágenes, vamos a utilizar el interfaz DirectSound que es un componente de DirectX. Los sonidos han sido trasformados en archivos RAW, porque ocupan menos espacio ya que no utilizan cabecera, como vimos en el apartado 3.1.6. Creamos una clase para cargar estos tipos de archivo. Creamos un buffer primario en el que colocamos los sonidos que vamos a escuchar. Estos representan el audio que realmente escuchará el usuario. Para mezclar los sonidos se usan los buffers secundarios. Para escribir sobre el buffer secundario, primero haremos un *lock*, a un trozo de ese buffer. Este lo creamos de 16 canales aunque realmente sólo voy a utilizar dos, pero lo dejo abierto por si se quieren poner más sonidos. Normalmente lo que se hace es poner un sonido en un canal libre, pero en nuestro caso lo más adecuado es utilizar siempre un mismo canal para reproducir el mismo sonido. La razón de hacerlo así es para saber si se está reproduciendo este sonido, evitar que se duplique y escuchar un efecto eco.

4.10 Entrada por teclado

Además de realizar la detección y control de movimiento con la cara, he querido implementar el control de movimiento mediante el teclado. La diferencia principal, entre usar el teclado y la detección de movimiento de la cara, consiste en que para el primer caso no existe una intensidad variable en el movimiento, mientras que para el segundo sí. El funcionamiento del teclado consiste principalmente en detectar cuándo se ha pulsado una tecla y hacer un tratamiento según el tipo de tecla que hayamos pulsado.

Capítulo 5

Conclusiones y trabajos futuros

5.1 Una vista atrás

El objetivo inicial de este proyecto fin de carrera consistía en crear un mundo virtual controlado mediante un interfaz perceptual, que fuera capaz de captar los movimientos de la cara del usuario y plasmar esta realidad en un movimiento en el mundo virtual. El sistema que he desarrollado funciona con una cámara de vídeo o con un archivo de vídeo AVI. Se ha partido de un sistema de detección y seguimiento de caras creado por el profesor Ginés García Mateos director del presente proyecto, y se han ido añadiendo funcionalidades para poder adaptarlo al mundo virtual que se ha creado.

5.2 Dificultades y limitaciones

El sistema, si se usa en condiciones óptimas y no se hacen movimientos demasiado bruscos, es bastante estable. No obstante, podemos señalar algunas dificultades encontradas.

- Si las condiciones de captura son muy adversas, pueden surgir problemas a la hora de detectar movimientos, por lo que lo ideal es que la cámara esté lo mas aproximada posible a la altura de los ojos.

- En otro caso se pueden producir malas interpretaciones.

- No todos los movimientos son independientes entre sí, por lo que la realización de uno puede producir cambios inesperados en el otro.

- Para evitarlo, se ha usado un proceso de priorizar movimientos.

- También ha habido problemas a la hora de detectar las rotaciones en el eje X y en el Y, debido a que son sensibles a la fuente de iluminación y es conveniente trabajar con luces uniformes.

- Para el entorno gráfico se han utilizado modelos que no estaban expresamente creados para utilizarlos en estos mundos ya que tienen un número excesivo de polígonos incrementa los requisitos del equipo.

5.3 Vías futuras

Propondremos las siguientes vías futuras para continuar con el trabajo desarrollado en este proyecto fin de carrera:

- El principal problema que me he encontrado son los grandes recursos que consume la aplicación, para eso se debería optimizar todo el proceso en busca de liberar memoria y desarrollar algoritmos que den velocidad.

- Sería conveniente investigar nuevos modos de detección de movimientos para las rotaciones en el eje X e Y, para que fueran mas robustas.

- Asimismo sería interesante introducir en el entorno virtual objetos animados y dotarlos de inteligencia para hacer más realista el escenario.

5.4 Objetivos cumplidos

En conclusión, consideramos que los objetivos propuestos para el proyecto han sido cumplidos.

- Se ha construido un mundo virtual 3D totalmente funcional usando las API de DirectX9.

- Se han estudiado y aplicado técnicas de procesamiento de imágenes y captura de vídeo, usando IPL y OpenCV, especialmente adaptado al procesamiento de caras humanas.

- Se han extendido las funcionalidades existentes sobre la detección y seguimiento de caras, abordando el problema de extraer información de giro y movimiento.

- El prototipo implementado funciona de forma satisfactoria en tiempo real.

Bibliografía

[Gong'00] Shaogang Gong, Stephen J McKenna and Alexandra Psarrou. Dynamic Vision. From images to faces recognition. Imperial College Press, 2000

[Walsh'03] Peter Walsh. Advanced 3D GAME Programming with DirectX 9.0.Wordware,2003.

[Luna'03] Frank D. Luna Introduction to 3D GAME Programming with DirectX 9.0. Wordware, 2003.

[Gems'00] Game Programming Gems. Charles River Media, 2000.

[Gems'01] Game Programming Gems II. Charles River Media, 2001.

[Gems'02] Game Programming Gems III. Charles River Media, 2002.

[Gamma'95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software. Addison Wesley, 1995.

[EckelVol1'95] Bruce Eckel. "Thinking in C++ Vol. 1" (Second Edition). Ed. Prentice Hall, 1995.

[EckelVol2'95] Bruce Eckel. "Thinking in C++ Vol. 2" (Second Edition). Ed. MindView, 1995

[Gregory'98] Kate Gregory. "Using Visual C++ 6". Ed. McMillan Computer Publishing, 1998.

[García'97] Paul Yao, Richard C. Leinecker. "Visual C++ 5 Bible". Ed. IDG Booksm, 1997.

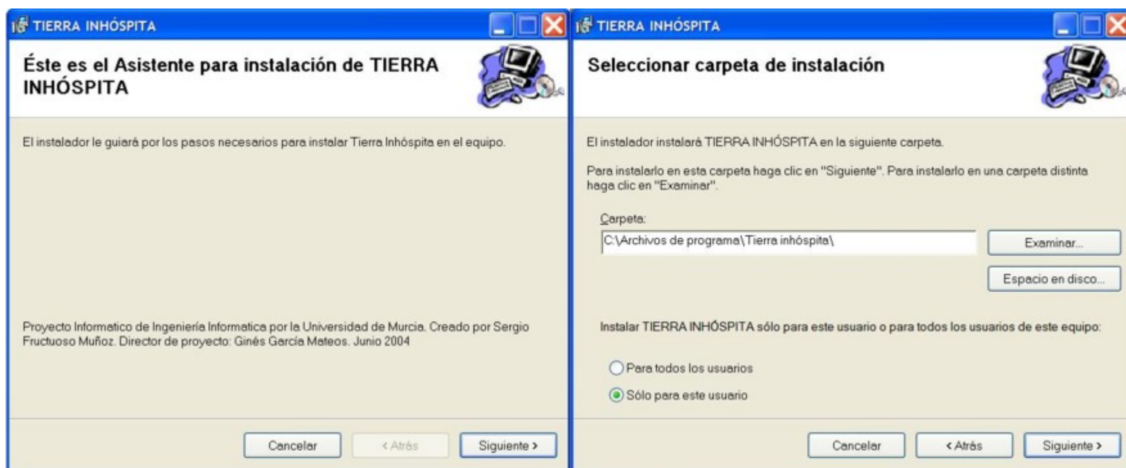
[Ruiz'01] Ruiz Antonio Ruiz. Cómo programar videojuegos en Windows. Ra-Ma, 2001.

[García'03] Ginés García Mateos: "Refining Face Tracking with Integral Projections", 4th International Conference on Audio- and Video-based Biometric Person Authentication, Guildford, UK, LNCS 2688, Springer, junio 2003.

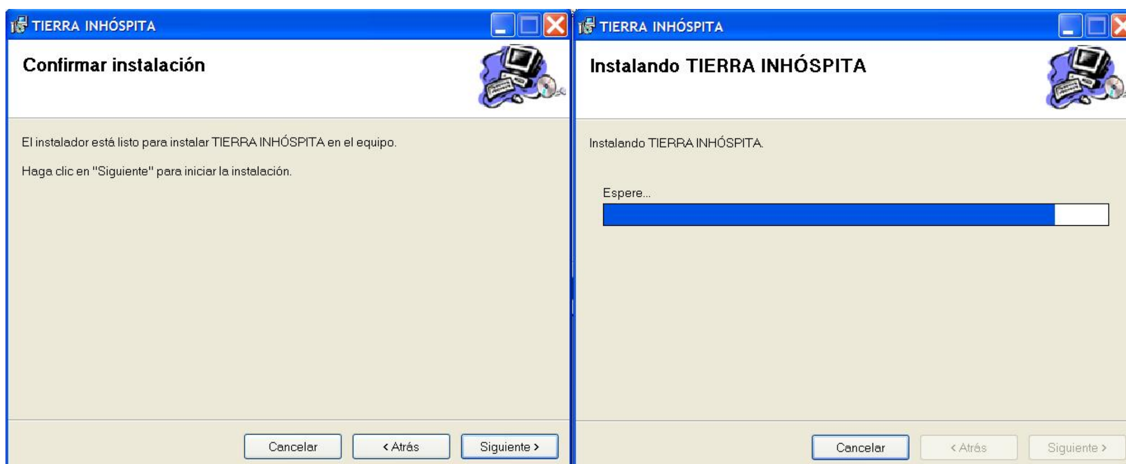
[García'02] Ginés García Mateos, Alberto Ruiz García, Pedro Enrique Lopez-de-Teruel: "Face Detection Using Integral Projection Models", Joint IAPR International Workshops On S+SSPR'2002, Windsor, Canada, LNCS 2396, Springer, agosto 2002.

APENDICE A. INSTALACIÓN

Antes de comenzar la ejecución del programa debemos instalarlo en el equipo destino donde lo queremos ejecutar. Comenzamos instalando el programa y ejecutando el archivo *instalar.msi*. A continuación nos aparecerá la pantalla de bienvenida que se muestra a continuación en la parte izquierda.

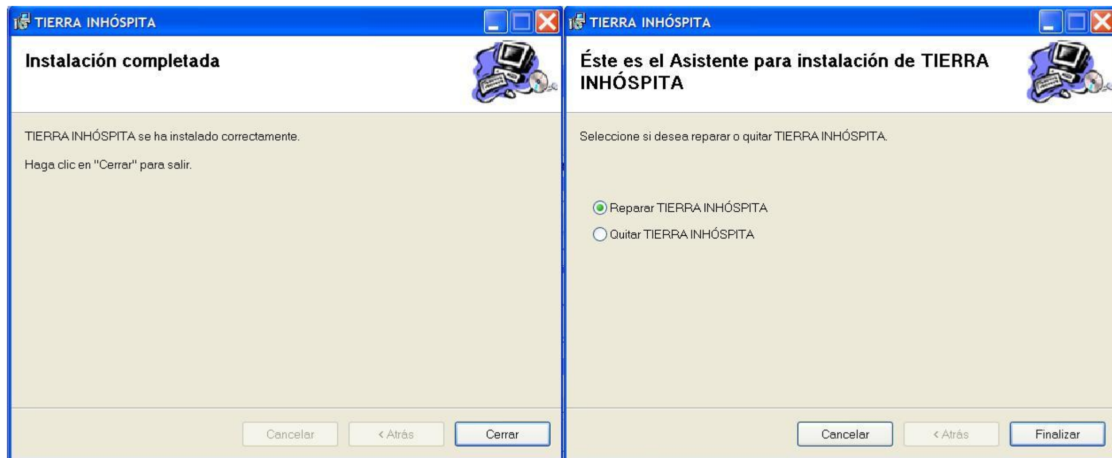


Después de aceptar, debemos indicarle la ruta donde queremos guardar el programa y comenzará la copia de archivos dentro del equipo destino.



Una vez confirmado todo, se creará un acceso directo en el escrito del usuario.

Si se vuelve a ejecutar el archivo de instalación se mostrará la ventana de la parte superior derecha, y podremos reparar o quitar la instalación existente.



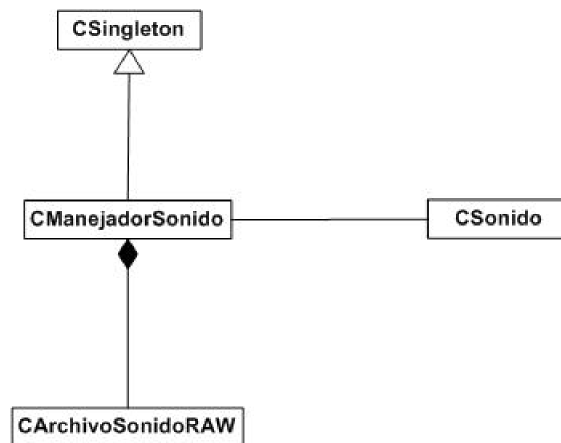
Para ejecutar la aplicación, es necesario bajarse DirectX 9.0 o superior, para que podamos ejecutar nuestra aplicación sin problemas.

APENDICE B. IMPLEMENTACIÓN

Para realizar la implementación he procurado encapsular las funcionalidades en clases para favorecer la reutilización. En este apéndice voy a explicar a grandes rasgos que representa cada atributo y cuales son las funcionalidades de las funciones miembros de las clases principales. Las demás clases las detallaré por encima. A continuación relataré las clases agrupándolos en módulos según tengan funcionalidades parecidas o afines.

Módulo de sonido.

En este módulo se van a agrupar todas las clases que tienen que ver con el sonido dentro de la aplicación.



Esquema 1. Clases pertenecientes al módulo de sonido.

Clase CSonido.

En esta clase fundamentalmente se inicializa el interfaz DirectSound para podernos manejar con sonidos. Crearemos los buffers y cargamos los sonidos que vamos a utilizar.

Atributos.

pDS: Es una estructura LPDIRECTSOUND y es lo que utiliza IDirectSound como interfaz.

bufPrimario: Es una estructura LPDIRECTSOUNDBUFFER y lo usamos como buffer primario.

sonidos: Es un array de estructuras de LPDIRECTSOUNDBUFFER y aquí vamos a contener la lista de sonidos cargados.

canales: Es un array estático de LPDIRECTSOUNDBUFFER y el tamaño viene determinado por el número de canales que vamos a utilizar para reproducir sonidos.

estadoCh: Es un array estático de DWORD cuyo tamaño viene determinado por el número máximo de canales por los cuales podemos reproducir sonidos y nos va a indicar el estado del canal. Esto es útil por ejemplo para saber si por este canal se está reproduciendo algún sonido.

numSounds: Entero que nos indica el numero de sonidos que hay actualmente cargados.

canalLibre: Entero que nos indica cuál es el próximo canal libre que podemos usar.

freq: Entero que nos dice la frecuencia de muestreo.

FPS: Entero que nos dice el número de frames por segundo.

numCanales: Entero que indica según sea mono o estéreo, uno o dos respectivamente.

parado: Variable booleana que indica si esta funcionando o no, para poder parar la reproducción de sonidos en un instante determinado.

appHWnd: Es una estructura HWND que hace referencia a la ventana de la aplicación principal.

DataOK: Variable booleana que indica si el objeto ha sido iniciado con éxito.

Métodos.

CSonido(void): Constructor de clase.

~CSonido(void): Destructor de clase.

init(HWND hWnd, int frec, int fps, int numC, int numSnd): Procedimiento que inicia el plugin de sonido. Le indicaremos el manejador de nuestra ventana de la aplicación principal, la frecuencia de frames por segundo, el número de canales y el número de sonidos que vamos a utilizar.

end(void): Procedimiento que finaliza el plugin de sonido.

pause(bool state): Función que pausa o continúa el sonido que se está emitiendo por cada uno de los canales.

*loadSound(unsigned char *data, long length, int numSnd)*: Función que nos va a cargar los sonidos que posteriormente vamos a reproducir.

playSound(int numSnd, bool loop): Reproduce el número de sonido que le pasamos como parámetro y le indicamos en la variable *loop* si queremos reproducirlo repetidamente.

stopSound(int numCanal): Procedimiento que para la reproducción de un sonido en un determinado canal.

estaSonando(int numerosonido): Función que nos devuelve un booleano que nos indica si se está reproduciendo un sonido por un determinado canal.

initDSound(HWND hWnd): Trata de obtener la interfaz IDirectSound y de fijar el nivel cooperativo. Devuelve en un booleano si se ha inicializado correctamente o no.

*createPrimaryBuffer(WAVEFORMATEX *wf)*: Crea el buffer primario y fija el formato a usar. Devuelve en un booleano si se ha podido crear o no el buffer primario.

createSecondaryBuffer(LPDIRECTSOUNDBUFFER &bufSecundario, int tamBuf): Crea un buffer estático secundario. Devuelve en un booleano el resultado de esta operación.

Clase CarchivoSonidoRaw.

Esta clase se encarga de tener un archivo de sonido tipo RAW para poder reproducir sonidos.

Atributos.

datos: Es un puntero a los datos del archivo Raw que vamos a cargar.

dataOK: Variable booleana que nos indica si se han leído los datos correctamente.

length: Es de tipo long y nos dice el tamaño del archivo que tenemos cargado.

Métodos.

*CArchivoSonidoRAW(const char *fich):* Método creador de clase que nos carga un fichero de sonido que le pasamos por parámetro.

~CArchivoSonidoRAW(void): Destructor de clase.

getDatos(void): Devuelve un array de unsigned char que contiene el archivo de sonido que hemos cargado.

estaOK(void): Método que nos devuelve si el fichero se ha cargado o no con éxito.

getFreq(void): Devuelve un entero que nos indica la frecuencia del archivo de sonido.

getNumBits(void): Retorna un entero que corresponde al número de bits del sonido.

getLength(void): Devuelve la longitud del fichero.

*setDatos(unsigned char *d):* Pone los datos del archivo de sonido que le estamos pasando como parámetros.

liberaDatos(void): Libera el sonido que tenemos cargado.

*carga(const char *fich):* Carga el fichero del archivo que le pasamos como parámetro. Devuelve un booleano que corresponde con el resultado del éxito o fracaso de la carga.

Clase CmanejadorSonido.

Clase que se encarga de interactuar entre la aplicación y el manejo de los sonidos, ya sea para cargar o tocar un determinado sonido. Sólo puede haber una instancia de esta clase ya que utilizamos el patrón Singleton.

Atributos.

freq: Entero que nos indica la frecuencia de muestreo.

FPS: Entero que nos dice los frames por segundo.

numCanales: Entero que nos dice si vamos a utilizar sonidos en mono o en estéreo, siendo su valor uno o dos respectivamente.

sound: Apuntador a la clase de Csonido que se encarga del uso de los sonidos.

Métodos.

CManejadorSonido(): Constructor de clase.

~CManejadorSonido(): Destructor de clase.

init(HWND hWnd, int frec, int fps, int numC, int numSounds): Función que inicia el plugin de sonido. Le indicaremos el manejador de nuestra ventana de la aplicación principal, la frecuencia los frames por segundo, el número de canales y el número de sonidos que vamos a utilizar. Devuelve un booleano que nos indica si se ha inicializado o no el plugin de sonido.

end(void): Finaliza el plugin de sonido.

pause(bool state): Procedimiento que para o permite que continué el sonido.

*loadSounds(CArchivoSonidoRAW **sonidos, int numSonidos)*: Función que carga los sonidos que podremos tocar. Devuelve en un booleano si se ha producido un error en la operación.

playSound(int numSnd, bool loop): Función que toca un sonido el cual le indicamos su número como parámetro y devuelve si ha ocurrido un error.

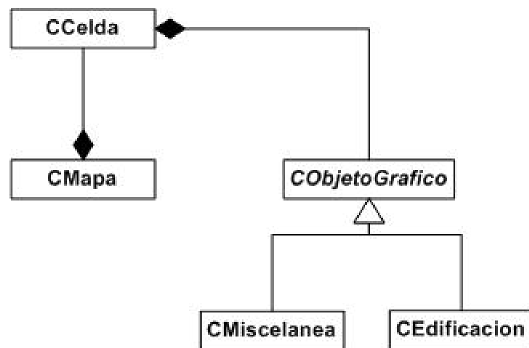
stopSound(int numCanal): Procedimiento que para la reproducción de un sonido en un determinado canal.

changeHWnd(HWND hWnd, int numSounds): Función que se usa para asociar el plugin de sonido a otra ventana. Devuelve el éxito de fracaso de la operación en un booleano.

EstaSonando(int numerosonido): Función que nos devuelve un booleano que nos indica si se está reproduciendo un sonido por un determinado canal.

Módulo de Fase.

Aquí vamos a agrupar las clases que se encargan de contener la información de todo el mapa, agrupados por casillas.



Esquema 2. Clases pertenecientes al módulo de sonido.

Clase CMapa.

Esta clase de encarga de mantener la información de las casillas, los puntos y los objetos que forman el mapa.

Atributos.

celda_actual: Entero que representa la celda en la que estamos actualmente en el mapa.

celdas: Array de estructuras CCelda que contiene todas las casillas en las que se subdivide la fase.

vistaportal: Array de estructura objetocasilla, que contiene todos los objetos que son visibles desde cada uno de las subcasillas.

punto: Array de estructuras Punto que contienen todos los puntos que forman los cuadriláteros de cada casilla dentro de un mapa.

Métodos.

CMapa(void): constructor de clase.

~CMapa(void): destructor de clase.

PuedoAvanzar(float antiguaposx,float antiguaposy,float posx, float posy): Función que dada dos posiciones determina si se puede avanzar o por el contrario se ha producido un choque. Para ello la función retornara un valor booleano.

ObjetosEnLaCasilla(): Procedimiento que consiste en pintar todos los objetos que se encuentran a la vista desde la casilla donde estamos actualmente.

GetCeldaActual(): Función que devuelve un entero que determina la casilla actual.

Clase CCelda.

Clase que representa a una casilla del mapa. Mantiene la información de sus coordenadas, las zonas accesibles como las inaccesibles y donde están situados los vecinos.

Atributos.

Izqarr: Es una estructura de tipo PUNTO que corresponde a la coordenada en el plano del punto en el noroeste de la casilla.

derarr: Es una estructura de tipo PUNTO que corresponde a la coordenada en el plano del punto en el noreste de la casilla.

izqaba: Es una estructura de tipo PUNTO que corresponde a la coordenada en el plano del punto en el suroeste de la casilla.

deraba: Es una estructura de tipo PUNTO que corresponde a la coordenada en el plano del punto en el sureste de la casilla.

celdizq: Es un array de estructuras CELDAS_VECINAS que se utiliza para determinar las celdas a las que se puede acceder por el este de la casilla que estamos actualmente.

celdarr: Es un array de estructuras CELDAS_VECINAS que se utiliza para determinar las celdas a las que se puede acceder por el norte de la casilla que estamos actualmente.

celdder: Es un array de estructuras CELDAS_VECINAS que se utiliza para determinar las celdas a las que se puede acceder por el oeste de la casilla que estamos actualmente.

celdaba: Es un array de estructuras CELDAS_VECINAS que se utiliza para determinar las celdas a las que se puede acceder por el sur de la casilla que estamos actualmente.

Métodos.

CCelda(void): Constructor de clase.

~CCelda(void): Constructor de clase.

DistanciaRecta(float vx1, float vy1,float vx2,float vy2, float posactx,float posacty): Función que dados dos puntos del plano, halla la recta y devuelve la distancia del punto donde estamos actualmente a esa recta.

PuntoCorteXArr(float antiguaposx, float antiguaposy, float posx, float posy): Función que determina si nos hemos salido de la celda actual por el norte a partir de la posición antigua y la posición a la que nos movemos.

PuntoCorteXAba(float antiguaposx, float antiguaposy, float posx, float posy): Función que determina si nos hemos salido de la celda actual por el sur a partir de la posición antigua y la posición a la que nos movemos.

PuntoCorteYIzq(float antiguaposx, float antiguaposy, float posx, float posy): Función que determina si nos hemos salido de la celda actual por el oeste a partir de la posición antigua y la posición a la que nos movemos.

PuntoCorteYDer(float antiguaposx, float antiguaposy, float posx, float posy): Función que determina si nos hemos salido de la celda actual por el este a partir de la posición antigua y la posición a la que nos movemos.

InsertarNumeroVecinos(int izq, int arr, int der, int aba): Procedimiento que inserta el numero de casillas vecinas que tenemos a cada uno de los lados incluyendo también las zonas en las que no podemos entrar.

InsertarVecinosIzq(unsigned short int subdivision, float yarriba, float yabajo, int numcasilla): Procedimiento que dice que casilla vecina es alcanzable por el este a partir de unas coordenadas dadas.

InsertarVecinosIzq(int numcasilla): Este procedimiento dice que casilla es alcanzable por el este. Solo se usa cuando solo se puede alcanzar una única casilla vecina.

InsertarVecinosArr(unsigned short int subdivision, float xizquierda, float xderecha, int numcasilla): Procedimiento que dice que casilla vecina es alcanzable por el norte a partir de unas coordenadas dadas.

InsertarVecinosArr(int numcasilla): Este procedimiento dice que casilla es alcanzable por el norte. Solo se usa cuando solo se puede alcanzar una única casilla vecina.

InsertarVecinosDer(unsigned short int subdivision, float yarriba, float yabajo, int numcasilla): Este procedimiento dice que casilla es alcanzable por el oeste. Solo se usa cuando solo se puede alcanzar una única casilla vecina.

InsertarVecinosDer(int numcasilla): Este procedimiento dice que casilla es alcanzable por el oeste. Solo se usa cuando solo se puede alcanzar una única casilla vecina.

InsertarVecinosAba(unsigned short int subdivision, float xizquierda, float xderecha, int numcasilla): Este procedimiento dice que casilla es alcanzable por el sur. Solo se usa cuando solo se puede alcanzar una única casilla vecina.

InsertarVecinosAba(int numcasilla): Este procedimiento dice que casilla es alcanzable por el sur. Solo se usa cuando solo se puede alcanzar una única casilla vecina.

*AsociarPuntos(Punto *ptoizqarr, Punto *ptoderarr, Punto *ptoderaba ,Punto *ptoizqaba)*: Establece los puntos del cuadrilatero que compone la casilla para establecer los límites.

PuedoAvanzar(float antiguaposx,float antiguaposy,float posx, float posy): A partir de la posición actual y la anterior determinamos si podemos avanzar devolviendo un entero que nos indica si hemos chocado contra una pared o un objeto o bien hemos cambiado a otra casilla.

Clase CObjetoGrafico.

Representa cualquier objeto que haya en la escena.

Atributos.

materiales: Es una estructura de tipo materiales y contiene una lista con todos los materiales que tiene la malla que forma el objeto.
texturas: Es una estructura de tipo Texturas y contiene una lista con todas las texturas de la malla que forman el objeto.

malla: El tipo de esta variables es ID3DXMesh y contiene todos los puntos que forman la malla del objeto.

D3D_Device: Es de tipo LPDIRECT3DDEVICE9 y contiene el dispositivo donde va a dibujarse el objeto.

orientación: Es de tipo D3DXMATRIX y contiene la orientación del objeto en la fase.

Métodos.

CargaModelo(char *nombrefichero): Carga el modelo de un archivo .x pasándole la ruta. Esta función devuelve un booleano con el éxito o fracaso de la operación.

Pinta(void): Método virtual que pinta el objeto en la pantalla.

~CObjetoGrafico(void): Destructor virtual del objeto.

Clase CEdificación.

Representa un objeto de la escena que vamos a dibujar pero depende de si lo tenemos a la vista el objeto o no.

Atributos.

_min: Es un vector D3DXVECTOR3 que contiene las coordenadas mínimas respecto al eje x y z de la caja que envuelve al objeto.

_max: Es un vector D3DXVECTOR3 que contiene las coordenadas máximas respecto al eje x y z de la caja que envuelve al objeto.

Métodos.

Pinta(): Procedimiento que pinta el objeto en la pantalla.

PuntoDentro(D3DXVECTOR3& p): Función que devuelve un booleano que nos dice si un punto está dentro del objeto.

Colisionan(D3DXVECTOR3 vMinA, D3DXVECTOR3 vMaxA,D3DXVECTOR3 vMinB, D3DXVECTOR3 vMaxB): Función que devuelve un booleano que nos dice si estamos dentro del objeto. Esto es útil para detectar colisiones.

Intersecan(void): Función que nos dice si el objeto esta dentro del frustum. Devuelve un booleano con el resultado.

*CEdificacion(LPDIRECT3DDEVICE9 device,char *nombrefichero, D3DXMATRIX T)*: Constructor de clase al que le pasamos el dispositivo donde dibujar, el nombre del fichero y la matriz de posición donde queremos colocar el objeto.

~CEdificacion(void): Destructor de clase.

ConstruirCaja(void): Recorre la malla para construir una caja que envuelva todo. Devuelve el éxito o fracaso de la operación en un booleano.

Situar(void): Sitúan las coordenadas de la caja en la orientación que se le de al objeto. Esto hay que hacerlo siempre después de calcular la caja que envuelve al objeto.

Clase CMiscelanea.

Representa un objeto de la escena que vamos a dibujar siempre sin detectar el plano frustum.

Métodos.

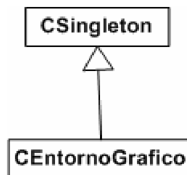
Pinta(): Procedimiento que pinta el objeto en la pantalla.

*CMiscelanea(LPDIRECT3DDEVICE9 device,char *nombrefichero, D3DXMATRIX T)*: Constructor de clase al que le pasamos el dispositivo donde dibujar, el nombre del fichero y la matriz de posición donde queremos colocar el objeto.

~CMiscelanea(void): Destructor de clase.

Módulo Gráfico.

En este módulo se va a incluir la única clase que se encarga de trabajar con el entorno gráfico.



Esquema 3. Clase del módulo gráfico.

Clase CEntornoGrafico.

Clase que se encarga de la inicialización de DIRECT3D y de dibujar por pantalla los objetos y decorados, además de calcular el frustum para optimizar.

Atributos.

Quad: Array de tipo IDirect3DVertexBuffer9 que lo usamos para almacenar vértices.

Tex: Array de tipo IDirect3DTexture y lo usamos para almacenar las texturas de los modelos.

decorado: Lista de la estructura listadecorado que contiene todos los objetos que son visibles desde una casilla.

Font: Es de tipo ID3DXFont y se utiliza para escribir en la pantalla los frames por segundo y las coordenadas por donde nos estamos moviendo en el mapa.

lf: Es de tipo LOGFONT y se utiliza para especificar el tipo y el tamaño de fuente que va a tener la variable font.

DisplayMode: Es una variable de tipo D3DDISPLAYMODE y se utiliza para especificar el modo en que se muestra la pantalla.

Present_Parameters: Es una variable de tipo D3DPRESENT_PARAMETERS y aquí especificamos los parámetros de la pantalla.

D3DCaps: Es de tipo D3DCAPS9 y se usa para obtener las capacidades que dispone nuestro hardware de video.

Direct3D_Object: Es de tipo LPDIRECT3D9 y aquí crearemos el dispositivo donde vamos a renderizar las imágenes.

D3D_Device: Es de tipo LPDIRECT3DDEVICE9 y aquí se especifica el dispositivo donde se va a pintar.

vista: Es de tipo D3DXMATRIX y representa a la matriz vista de la imagen.

proyeccion: Es de tipo D3DXMATRIX y representa a la matriz proyección de la imagen.

Métodos.

CEntornoGrafico(void): Constructor de clase.

~CEntornoGrafico(void): Destructor de clase.

InitD3D(HWND hwnd, int Width, int Height): Función que inicializa la ventana donde aparece el entorno virtual a partir de una altura y un ancho dado. Devuelve en un booleano el éxito o fracaso de la inicialización.

*Display(CMapa *unafase)*: Procedimiento que se encarga de pintar lo que vemos actualmente en la pantalla.

*InsertarTexto(char *texto,int fila,int alto, int ancho)*: Procedimiento que inserta el texto que le pasamos como parámetro por pantalla y en la posición que le indicamos.

EmpiezaEscena(): Prepara la escena para que podamos poner los objetos.

TerminaEscena(): Termina la escena para que podamos dibujarla.

CalculaFrustum(): Calcula los planos del frustum para poder determinar los objetos que hay en el mapa que son visibles.

GetDevice(): Función que nos devuelve el dispositivo actual que es de tipo LPDIRECT3DDEVICE9.

PreparaIntroduccion(void): Función que nos crea la ventana principal de introducción y nos retorna un booleano que nos indica si ha ocurrido un fallo.

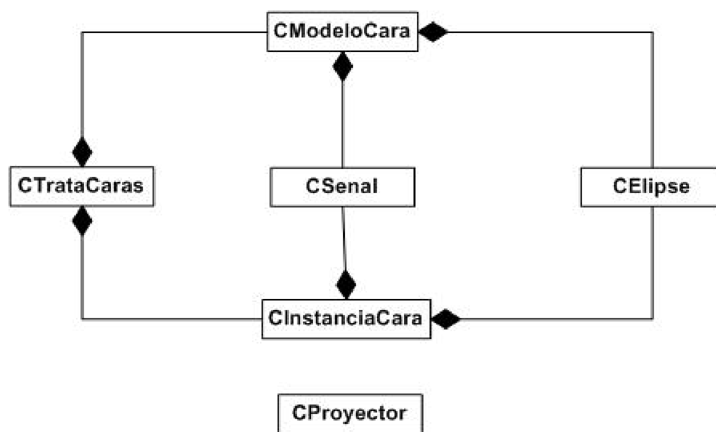
Introduccion(void): Función que muestra la pantalla de introducción y nos devuelve si ha ocurrido o no un fallo.

PonerNiebla(void): Procedimiento que muestra la niebla en el entorno virtual. Podemos elegir que tipo de niebla queremos ya sea lineal o exponencial.

Todas las funciones que se muestran a continuación sirven para colocar objetos en el mundo virtual que hemos creado.
PonerCielo(void).
PonerBusto(void).
PonerIglesia(void).
PonerParedesA(void).
PonerParedesB(void).
PonerParedesC(void).
PonerParedesD(void).
PonerParedesE(void).
PonerParedesF(void).
PonerParedesG(void).
PonerSueloC(void).
PonerSueloD(void).
PonerSueloE(void).
PonerSueloF(void).
PonerPuerta(void).

PonerTumba1(void).
PonerCruz1(void).
PonerCruz2(void).
PonerCruz3(void).
PonerTorre(void).
PonerTorre2(void).
PonerArco(void).
PonerCripta(void).

Módulo de detección y seguimiento de caras.



Esquema 4. Diagrama de clases para la detección de movimientos.

Clase CTrataCaras.

Esta clase es la que tiene la funcionalidad de detectar, seguir e interpretar las caras e interactuar con las clases existentes que ha diseñado Gines García Mateos.

Atributos.

k: Es un entero que nos muestra el número de frames que llevamos capturados.

estado: Estado que determina si la cara se ha localizado o perdido.

img: Variable de tipo IplImage en el que figura la imagen principal, ya sea la capturada por la cámara de video o por el video que le pasamos como parámetro.

img2: Variable de tipo IplImage que utilizamos para guardar la imagen principal, que hemos obtenido desde la cámara.

img3: Variable de tipo IplImage en el que figura la el frame A.

img4: Variable de tipo IplImage en el que figura el Frame B.

cap: Identificador de dispositivo de captura ya sea de video o de la WebCam. Es de tipo CvCapture

med: Variable de tipo double que represente la media.

Var: Variable de tipo double que represente la varianza.

girosCara: Pertenece a la clase CDetectaGiros y sirve para determinar las rotaciones en el eje X y en el eje Y.

frameA: Booleano que indica si se tiene que mostrar el frame A.

frameB: Booleano que indica si se tiene que mostrar el frame B.

preciso: Booleano que indica si se tiene que ser preciso en la detección.

imagen: Booleano que indica si se tiene que mostrar la imagen principal.

depuracion: Booleano que indica si se está en el modo depuración.

modelos: Array de estructuras CModeloCara de los modelos de las caras que hemos detectado.

instans: Array de estructuras CInstanciaCara de las instancias de la cara que hemos detectado.

hid_cascade: Es una estructura CvHidHaarClassifierCascade y aquí es donde se almacena el patrón de clasificación de la detección de caras.

PosicionCara: Pertenece a la clase CEstado y determina los movimientos de la cara

cascade: Variable que contiene una representación interna mas rápida que la precargada en hid_cascade. Es de tipo CvHidHaarClassifierCascade.

storage: Estructura de CVMemStorage de OpenCv para almacenar cosas.

Métodos.

CTrataCaras(void): Creador de clase.

~CTrataCaras(void): Destructor de clase.

Deteccion(void): Detecta las caras que hay en la imagen y nos devuelve un booleano que nos indica si se ha encontrado algo.

SigueCaras(void): Sigue la cara mediante el proceso de integrales proyectivas.

guardaCaras (CvSeq faces, bool usaHeurs,int numCaras)*: Guarda los modelos y las instancias de las caras que encontremos.

borraCaras (void): Borra los modelos y las instancias que tenemos en memoria.

detectaCarasHaar (CvHidHaarClassifierCascade cascade)*: Devuelve en la estructura CvSeq donde se encuentran las regiones en la imagen donde se ha detectado caras.

InicializaWebCam(void): Función que inicializa la WebCam y nos devuelve el existo o fracaso de esta inicialización.

CapturaFrame(void): Captura el próximo frame que vamos a tratar. Devuelve en un booleano el éxito o fracaso de la operación.

DetectaCaras(void): Intenta encontrar una cara en la imagen correspondiente. Devuelve en un booleano el éxito o fracaso de la operación.

CreaVentana(void): Crea la ventana donde se pone la imagen de la WebCam o de video.

MuestraImagen(void): Muestra el frame en la ventana de la WebCam.

*CalculaGiroIzqDer(CInstanciaCara *instancia,CModeloCara *modelo)*: Se le pasa la instancia y el modelo de la cara y nos devuelve en un double el valor del giro en el eje X.

*CalculaGiroArrAba(CInstanciaCara *instancia,CModeloCara *modelo)*: Se le pasa la instancia y el modelo de la cara y nos devuelve en un double el valor del giro en el eje Y.

*InicializaAvi(char *archivo)*: Inicializa un archivo de video que le pasamos como parámetro. Devuelve en un booleano el éxito de esta operación.

ajustaThresholds(void): Ajusta los umbrales por defecto en la detección.

guardaCaras2(CvSeq faces, bool usaHeurs, int numCaras)*: Igual que el otro procedimiento guarda caras salvo que no guarda los modelos y usa el que hemos detectado.

Métodos getters y setters para tratar la imagen
void setFrameA(bool opcion).
void setFrameB(bool opcion).
void setPreciso(bool opcion).
void setImagen(bool opcion).
void setDepuracion(bool opcion).

Métodos getters y setter para ajustar los límites del estado de la cara.
int getGirosCaraDesplX(void).
void setGirosCaraDesplX(int valor).
int getGirosCaraDesplY(void).
void setGirosCaraDesplY(int valor).
int getGirosCaraDesplZ(void).
void setGirosCaraDesplZ(int valor).
int getGirosCaraGiroX(void).
void setGirosCaraGiroX(int valor).
int getGirosCaraGiroYArr(void).
void setGirosCaraGiroYArr(int valor).
int getGirosCaraGiroYAba(void).
void setGirosCaraGiroYAba(int valor).
int getGirosCaraGiroZ(void).
void setGirosCaraGiroZ(int valor).
bool guardarParametros(void).

Métodos getters y setters para dar funcionalidad a todos los movimientos.
bool ObtenerFuncionalidadAdAt(void).
bool ObtenerFuncionalidadArrAba(void).
bool ObtenerFuncionalidadIzqDer(void).
bool ObtenerFuncionalidadGiroX(void).
bool ObtenerFuncionalidadGiroY(void).
bool ObtenerFuncionalidadGiroZ(void).
void PonerFuncionalidadAdAt(bool valor).
void PonerFuncionalidadArrAba(bool valor).
void PonerFuncionalidadIzqDer(bool valor).
void PonerFuncionalidadGiroX(bool valor).

void PonerFuncionalidadGiroY(bool valor).
void PonerFuncionalidadGiroZ(bool valor).

void inercia(): Se mueve con el último movimiento.

Clase CProyector.

Se encarga de la proyección vertical de la cara, ojos y boca.

Clase CModeloCara.

Su principal función es la de contener el modelo de la cara en el que nos vamos a basar en el seguimiento de las caras.

Clase CInstanciaCara.

Esto se utilizar para mantener la localización de la cara actualmente e ir actualizandose a lo largo del tiempo.

Clase CSenal.

Esta clase representa la señal de la integral proyectiva y dispondremos de diversas funcionalidades como es la de ajustar dos señales.

Clase CElipse.

Para especificar donde se ha localizado la cara dentro de la imagen se va a detallar mediante una elipse para lo cual se encarga esta clase.

Clase CDetectaGiros.

La funcionalidad de esta clase es la de deducir la cantidad movimiento que hacemos cuando giramos la cabeza en la izquierda y en la derecha

Atributos.

senalrectificadamodelgiroX: Array de float que representa la señal horizontal de los ojos pasada por un filtro gausiano.

giroxposmodel: Variable entera que indica la posición máxima de la señal horizontal de los ojos.

maxmodelgiroX: Variable de tipo float que representa el máximo de la señal horizontal de los ojos.

giroyposmodel: Entero que muestra la posición máxima de la señal vertical de la cara.

maxmodelgiroY: Float que representa el máximo de la señal vertical de la cara.

Métodos.

CDetectaGiros(void): Constructor de clase.

~CDetectaGiros(void): Destructor de clase.

*giroX(float *datos,CModeloCara *cara,int *pos,int instanciamin)*: Función que devuelve en un entero la intensidad de la rotación en el eje X.

*giroY(float *datos,CModeloCara *cara,int *pos,int instanciamin)*: Función que devuelve en un entero la intensidad de la rotación en el eje Y.

void reiniciar(void): Reinicia los buffers.

*void CalculaModelos(CModeloCara *cara)*: Calcula las señales rectificadas por un filtro gausiano de los modelos de la cara.

Módulo de movimiento.

A este grupo pertenecen las dos clases que se encarga del movimiento por nuestro entorno.

Esquema 5. Clases encargadas de movernos.

<u>Clase CMotorEntradas.</u>

Clase que controla todos los movimientos que hacemos en el mundo.

Atributos.

xPos: Posición en punto flotante en el eje x.

yPos: Posición en punto flotante en el eje y
.
zPos: Posición en punto flotante en el eje z.

xView: Coordenada x en punto flotante del vector vista.

yView: Coordenada y en punto flotante del vector vista.

zView: Coordenada z en punto flotante del vector vista.

xUp: Coordenada x en punto flotante del vector arriba.

yUp: Coordenada y en punto flotante del vector arriba.

zUp: Coordenada z en punto flotante del vector arriba.

xStrafe: Coordenada x en punto flotante del vector desplazamiento.

yStrafe: Coordenada y en punto flotante del vector desplazamiento.

zStrafe: Coordenada z en punto flotante del vector desplazamiento.

Métodos.

CMotorEntradas(void): Creador de clase.

~CMotorEntradas(void): Destructor de clase

GirarXYZ(float angulo,float velocidadx,float velocidady,float velocidadz): Procedimiento que permite girar en cualquier ángulo.

*Caminar(CMapa *unafase,float velocidad)*: Procedimiento para poder caminar en el mapa según la velocidad que le pongamos.

*DesplazamientoLateral(CMapa *unafase,float velocidad)*: Procedimiento para desplazarse lateralmente con la velocidad que le pasemos como parámetro.

GetDirection(float &x, float &y, float &z): Obtiene la dirección donde estamos mirando.

GiroIzquierda(float velocidad): Procedimiento que gira a la izquierda con la velocidad que se le pase como parámetro.

GiroDerecha(float velocidad): Procedimiento que gira a la derecha con la velocidad que se le pase como parámetro.

*DesplazaIzquierda(CMapa *unafase, float velocidad)*: Procedimiento para desplazarnos lateralmente hacia la izquierda con la velocidad que le pasemos como parámetro.

*DesplazaDerecha(CMapa *unafase, float velocidad)*: Procedimiento para desplazarnos lateralmente hacia la izquierda con la velocidad que le pasemos como parámetro.

MiraArriba(float velocidad): Provocamos un giro hacia arriba siempre que no alcancemos un tope.

MiraAbajo(float velocidad): Provocamos un giro hacia abajo siempre que no alcancemos un tope.

*Adelante(CMapa *unafase, float velocidad)*: Procedimiento para caminar hacia delante con la velocidad que le pasemos como parámetro.

*Atras(CMapa *unafase, float velocidad)*: Procedimiento para caminar hacia atrás con la velocidad que le pasemos como parámetro.

RotarIzqZ(float velocidad): Rota en el eje Z con la velocidad que le pasemos como parámetro hacia la izquierda.

RotarDerZ(float velocidad): Rota en el eje Z con la velocidad que le pasemos como parámetro hacia la derecha.

Arriba(float velocidad): Nos movemos hacia arriba siempre que no superemos un tope con la velocidad que se le pasa como parámetro.

Abajo(float velocidad): Nos movemos hacia arriba siempre que no superemos un tope con la velocidad que se le pasa como parámetro.

MiradaAlfrente(void): Compensa el movimiento de rotación en el eje Y para que se estabilice la mirada en el centro.

Clase CEstado

Esta clase se encarga de detectar variaciones en una imagen procedente desde la cámara y transformarla en movimiento.

Atributos.

Parámetros enteros para ajustar los umbrales.
ajustax.
giroizq.
giroder.
giroarr.
giroaba.
giroz.
distocular.
distocularBoca.

Variables booleanas para saber si deseamos detectar movimientos o giros.
funcmovadat.
funcmovizqder.
funcgiroX.
funcgiroY.
funcgiroZ.
funcmovarraba.

inicialEjeX: Float que usamos para representar la coordenada inicial en X del punto interocular.

inicialEjeY: Float que usamos para representar la coordenada inicial en Y del punto interocular.

inicialBocaY: Entero que usamos para representar la coordenada inicial Y de la boca.

distYOjosBoca: Float que usamos para especificar la coordenada inicial en Y del punto interocular y la boca.

girosIzqDer: Variable de tipo float que representa el ángulo de giro en el eje X.

girosArrAba: Variable de tipo float que representa el ángulo de giro en el eje Y.

pintaestadistica: Boolenano que nos dice si está abierta la ventana de depuración.

desplX: Float para especificar el umbral en el movimiento sobre el eje X.

desplY: Float para especificar el umbral en el movimiento sobre el eje Y.

desplZ: Float para especificar el umbral en el movimiento sobre el eje Z.

Métodos.

CEstado(void): Creador de clase.

~CEstado(void): Destructor de clase.

void Incial(float xojoizq, float yojoizq, float xojoder, float yojoder, float xboca, float yboca): Creamos el estado inicial de donde partimos.

void Estado(float xojoizq, float yojoizq, float xojoder, float yojoder, float xboca, float yboca): Extraemos el movimiento del estado actual.

void setGirosIzqDer(double valor): Actualizamos el ángulo de giro en el eje X.

void setGirosArrAba(double angulo): Actualizamos el ángulo de giro en el eje Y.

void CreaVentanaDepuracion(void): Creamos la ventana de depuración.

void DestruyeVentanaDepuracion(void): Cerramos la ventana de depuración.

bool guardaParametros(void): Guarda los parámetros de los umbrales en disco.

void cargaParametros(void): Carga los parámetros desde un fichero.

void ajustarParametros(void): Ajustar parámetros de los umbrales.

void creaArchivosMuestras(void): Procedimiento para crear los ficheros de movimientos.

Procedimientos para escribir un movimiento en un fichero
void escribirArchivoMovX(int movx).
void escribirArchivoMovY(int movy).
void escribirArchivoMovZ(int movz).
void escribirArchivoGiroX(int girox).
void escribirArchivoGiroY(int giroy).
void escribirArchivoGiroZ(int giroz).

void cerrarArchivosMuestras(void): Procedimiento para cerrar todos los archivos de muestras.

void moverse(void): Se mueve según los valores que tengamos en nuestros buffers.

Métodos getters y setters para dar funcionalidad a todos los movimientos.
void setFuncionalidadAdAt(bool valor).
bool getFuncionalidadAdAt(void).
void setFuncionalidadArrAba(bool valor).
bool getFuncionalidadArrAba(void).
void setFuncionalidadIzqDer(bool valor).
bool getFuncionalidadIzqDer(void).
void setFuncionalidadGiroX(bool valor).
bool getFuncionalidadGiroX(void).
void setFuncionalidadGiroY(bool valor).
bool getFuncionalidadGiroY(void).
void setFuncionalidadGiroZ(bool valor).
bool getFuncionalidadGiroZ(void).

Métodos getters y setters para ajustar los parámetros de giros y movimientos.
int getAjustax(void).
void setAjustax(int valor).
int getDistocular(void).
void setDistocular(int valor).

int getDistocularBoca(void).
void setDistocularBoca(int valor).
int getGiroizq(void).
void setGiroizq(int valor).
int getGiroder(void).
void setGiroder(int valor).
int getGiroarr(void).
void setGiroarr(int valor).
int getGiroaba(void).
void setGiroaba(int valor).
int getGiroZ(void).
void setGiroZ(int valor).

Patrones.

Clase Singleton.

Clase que proporciona la funcionalidad necesaria para tener una única instancia de un objeto (singleton). Cuando queremos usar este patrón en una clase heredaremos de esta clase para obtener su funcionabilidad.

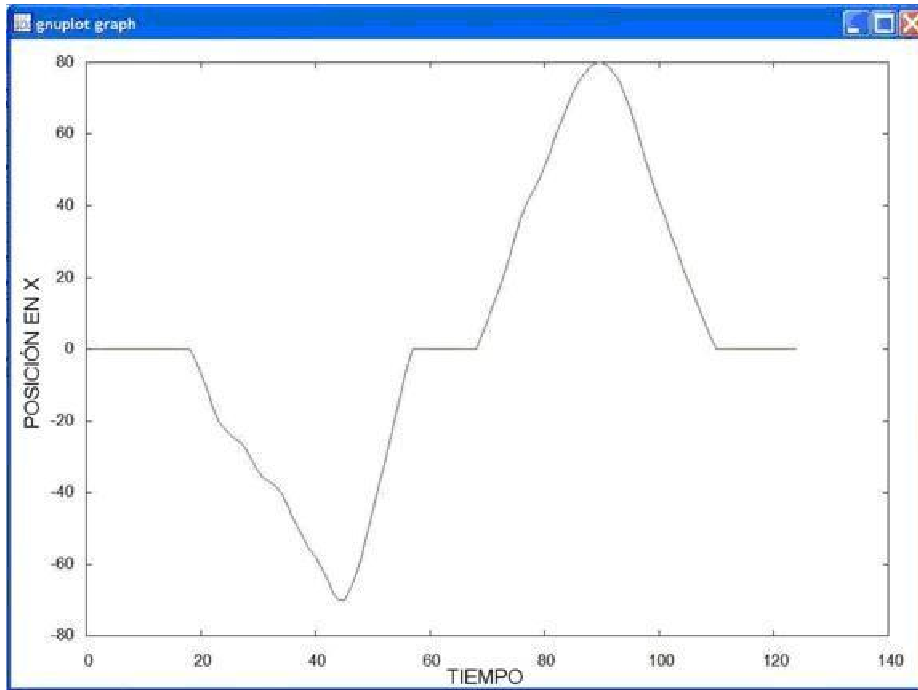
APENDICE C. EJEMPLO DE USO

En este apéndice se mostrará mediante un ejemplo práctico, cada uno de los movimientos y giros existentes, y los resultados de una detección, para poder apreciar la suavidad y el ruido que puede aparecer y los valores en cada instante del tiempo.

Desplazamiento en el eje X

Partimos de una posición inicial centrada y nos desplazamos primero a la izquierda y después hacia la derecha, aunque en la imagen aparezca al revés. Esto es por el efecto espejo que tiene la imagen que procede desde la cámara de vídeo. Por último se volverá al centro.

Esquema 6. Secuencia de un desplazamiento a la izquierda y a la derecha

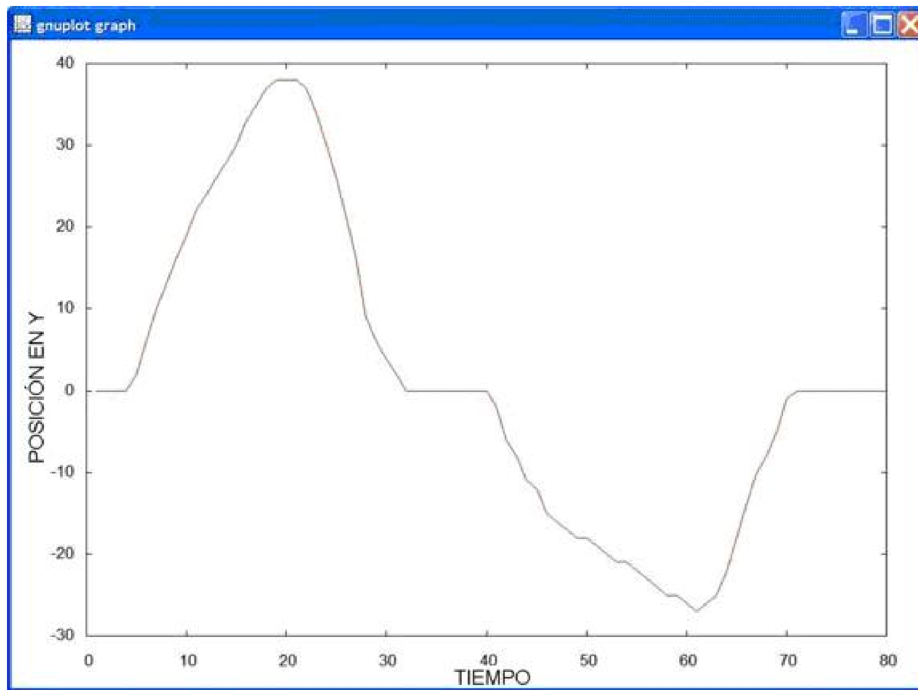


Esquema 7. Representación de valores del movimiento en X de la secuencia superior.

Desplazamiento en el eje Y

Iniciamos el movimiento desde una posición centrada y nos elevamos desde una posición fija para posteriormente agacharnos y volver al lugar inicial.

Esquema 8. Secuencia de un desplazamiento hacia arriba y abajo.

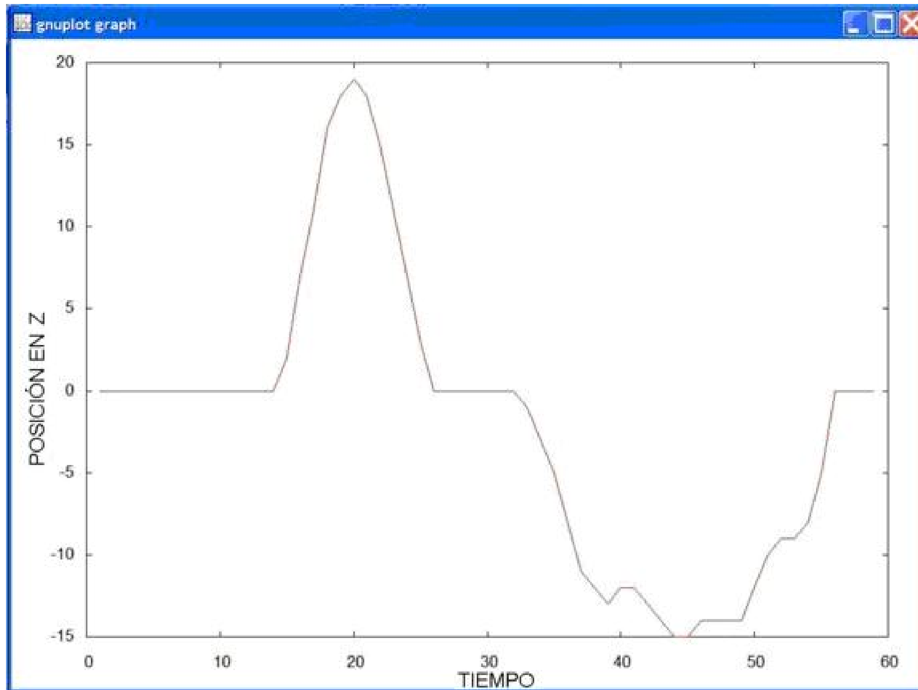


Esquema 9. Representación de valores del movimiento en Y de la secuencia superior.

Desplazamiento en el eje Z

Empezamos por una posición centrada y nos acercamos y alejamos de la cámara para posteriormente volver a la posición inicial.

Esquema 10. Secuencia de un desplazamiento hacia adelante y hacia atrás.

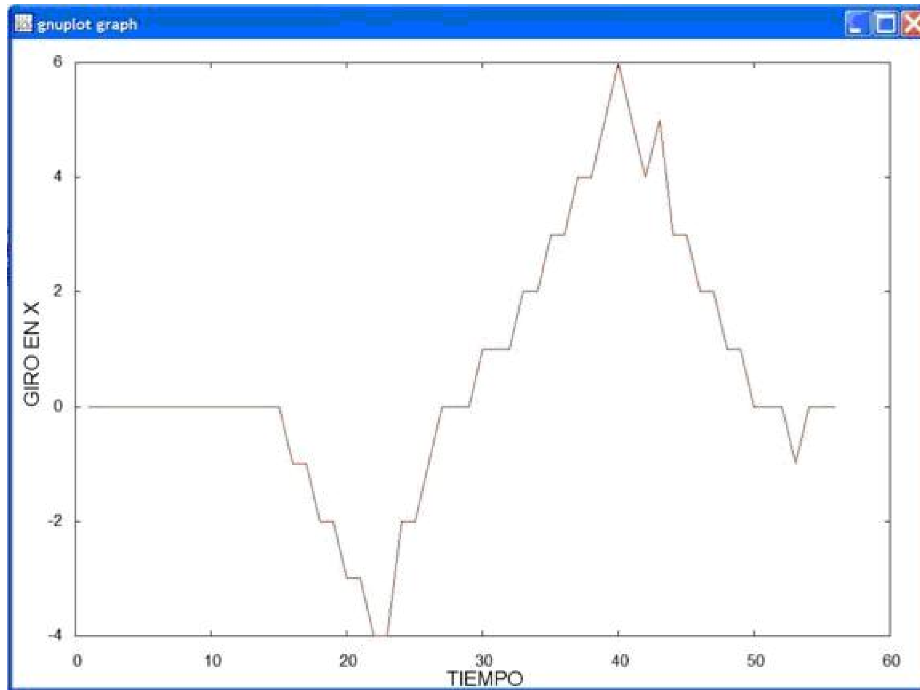


Esquema 11. Representación de valores del movimiento en Z de la secuencia superior.

Giro en el eje X

Aquí pasa igual que en el desplazamiento sobre el eje X, que la cámara capta la imagen al revés como si fuese un espejo. El movimiento consiste en primero girar hacia la derecha y después hacia al centro, volviendo al lugar inicial. Esto se puede apreciar en el Esquema 12 que se muestra a continuación.

Esquema 12. Giro hacia la izquierda y la derecha.

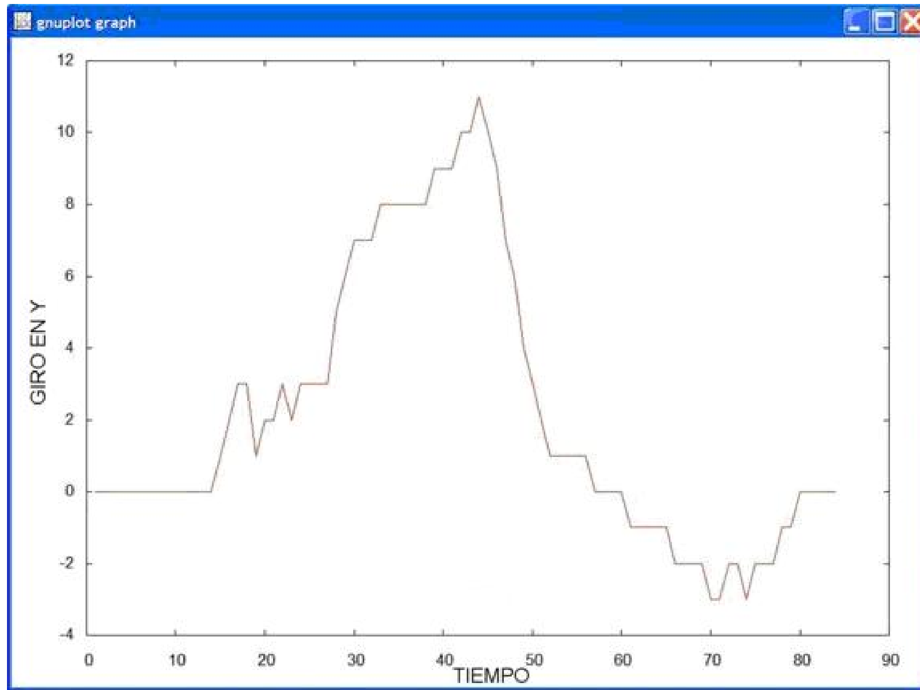


Esquema 13. Representación de valores de Giro en X de la secuencia superior.

Giro en el eje Y

Partiremos de un lugar centrado de la pantalla y giramos la cabeza hacia arriba y posteriormente hacia abajo para volver a la posición inicial. En el Esquema 14 se puede ver cómo es el movimiento.

Esquema 14. Giro hacia arriba y hacia abajo.

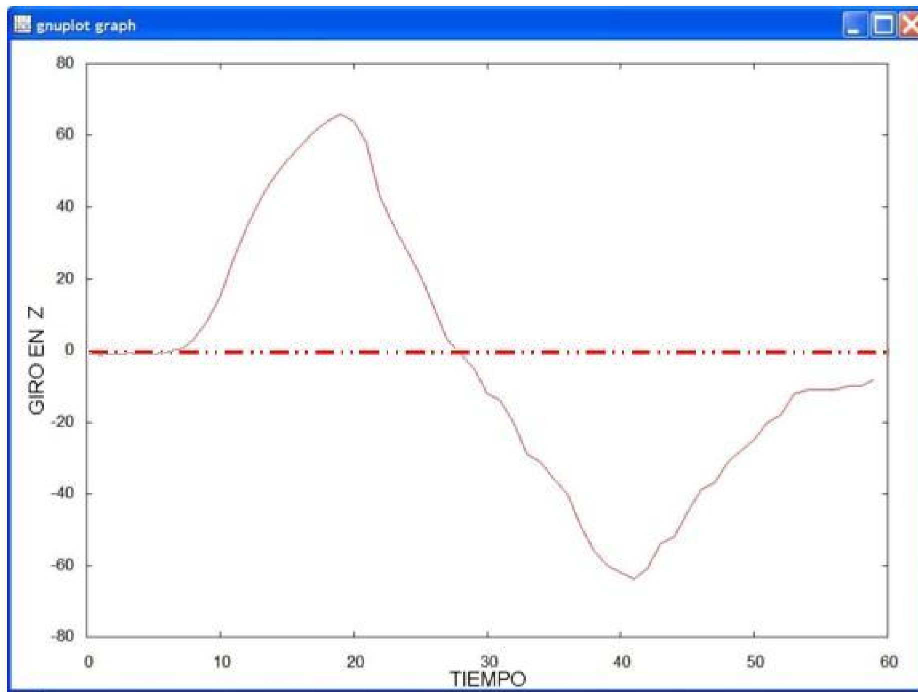


Esquema 15. Representación de valores de Giro en Y de la secuencia superior.

Giro en el eje Z

Aquí también ocurre el efecto espejo, pero no para calcular el ángulo de giro, sino para determinar cuál es el sentido del mismo. Iniciamos desde una posición centrada y giramos hacia la izquierda y posteriormente a la derecha, para terminar en el lugar inicial.

Esquema 16. Rotar hacia la izquierda y la derecha.



Esquema 17. Representación de valores de Giro en X de la secuencia superior.

APENDICE D. MANUAL DE USUARIO

Puesta en marcha

Lanzamos la aplicación bien desde el acceso directo, que se ha creado en el escritorio, o bien directamente sobre la aplicación principal y esperamos a que se nos muestre la ventana de bienvenida que nos pedirá que pulsemos la tecla ENTER para comenzar a funcionar.

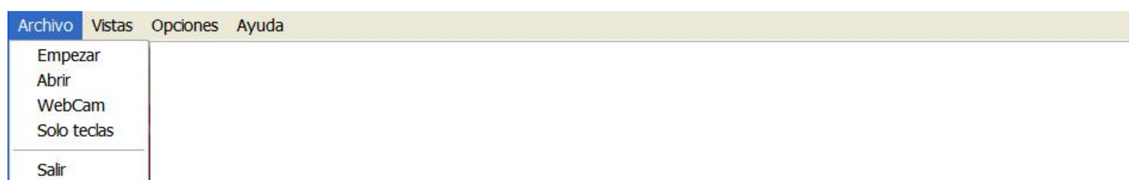
Interfaz de usuario

La aplicación consta de unos menús para configurar los parámetros variables la ventana donde se pintará el entorno virtual y opcionalmente de varias ventanas, donde se muestra la entrada principal de vídeo con la detección y seguimiento de la cara, la localización de la misma, ojos y boca, las tres integrales proyectivas del modelo y de la instancia, y por último, una ventana donde mostrar los valores que representan los movimientos de la cara en cada instante. La activación/desactivación de estos entonces se verá mas adelante.

Menús

Toda nuestra aplicación es configurable desde el menú que se muestra a continuación.

Aparecen cuatro opciones en la que se puede apreciar las siguientes subdivisiones:



- Archivo:
  - *Empezar:* Empezamos el juego si estamos en la pantalla de presentación y si estamos sobre el juego recomenzamos la aplicación.
  - *Abrir*: Esta opción es para especificar un archivo "avi" para detectar los movimientos desde un vídeo grabado.
  - *WebCam*: Sirve para indicar que la entrada de vídeo va a ser la que se captura desde la WebCam de tu ordenador.

- *Solo teclas*: Si señalamos esta opción, la detección de movimientos no se aplicará con lo que únicamente nos basaremos en las teclas para movernos por el mundo.

- Vistas:

  - *FrameA*: Abre la pantalla donde se muestra la cara localizada y la posición de los ojos y la boca.

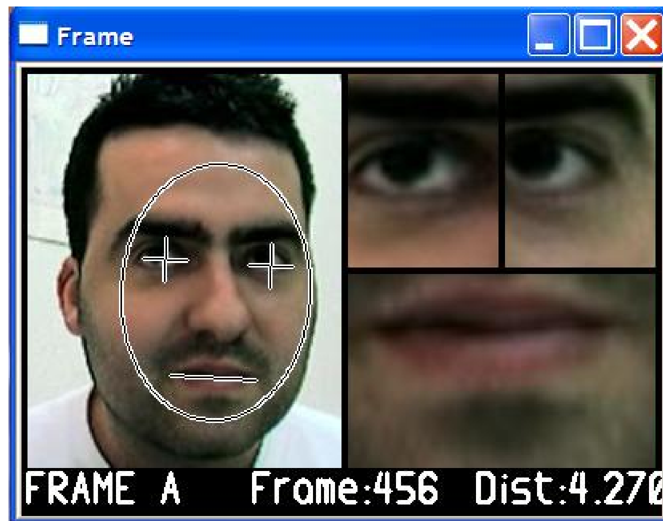


Imagen 1. Frame A.

  - *FrameB*: Muestra una parte de la imagen donde está la cara, con las integrales proyectivas de los modelos y de la instancia actual.

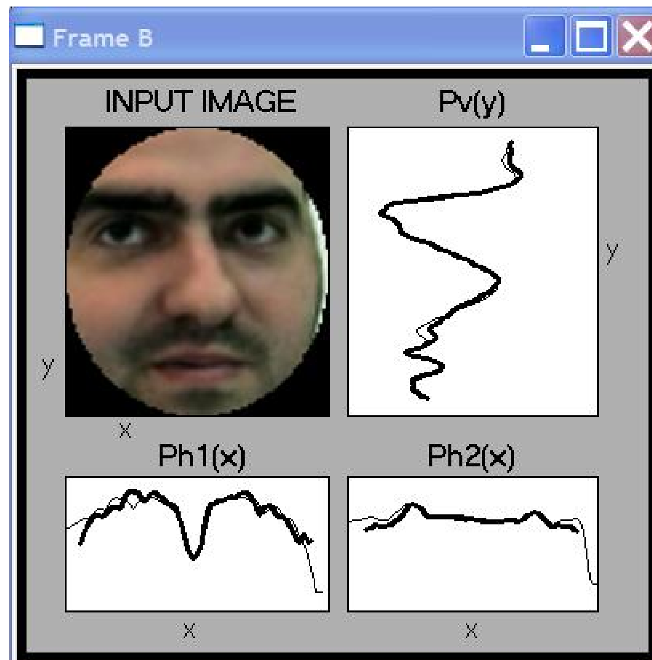
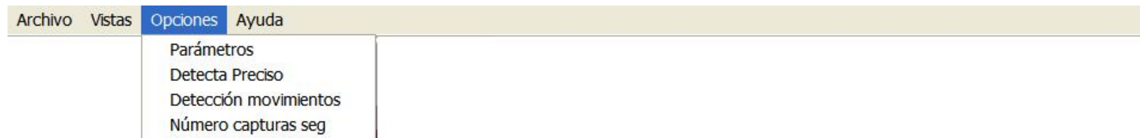


Imagen 2. Frame B.

- *Valores*: Muestra los resultados de la detección de los movimientos de la cara expresados numéricamente.
- *Rendimiento*: Muestra en la pantalla principal el rendimiento de entorno gráfico, así como las posiciones en el mapa.
- *Cámara*: Aquí es donde se nos muestra la imagen capturado por la cámara que nos suministra la imagen para analizarla.

Imagen 3. Imagen principal.



- Opciones:
  - *Parámetros*: Aquí especificamos los umbrales a partir de los cuales podemos decirle al detector que existe movimiento, ya sea en desplazamiento como en rotación, en los tres ejes del espacio.

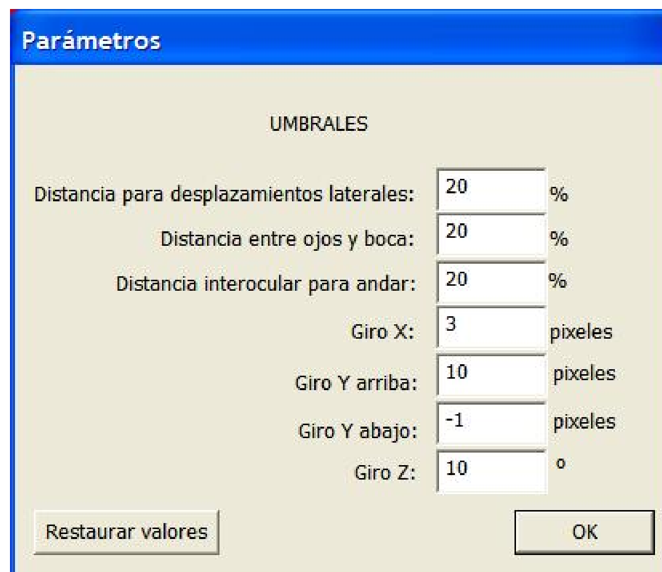


Imagen 4. Ventana de umbrales.

- *Detecta Preciso*: Activando esta opción haremos que el detector de caras sea mucho mas preciso pero ralentizando la aplicación.
- *Detección movimientos*: En esta opción se especifican los desplazamientos y rotaciones que vamos a detectar en las imágenes. Esto nos permite desactivar temporalmente algunas de las detecciones.

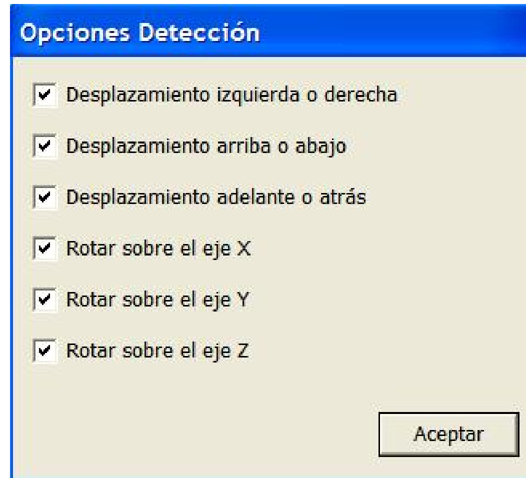


Imagen 5. Opciones de la detección.

- *Número capturas seg.*: Número de imágenes capturadas por segundo que vamos a tratar para la detección de movimientos.

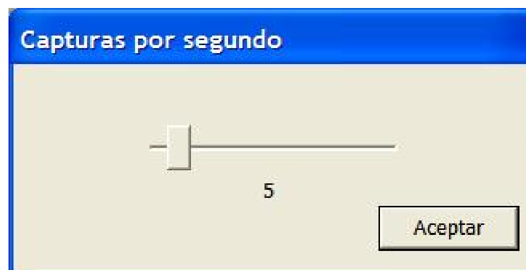


Imagen 6. Ventana de capturas por segundo.

- Ayuda:
  - *Temas de Ayuda*: Si pinchamos sobre esta opción desplegamos la ayuda de la aplicación.

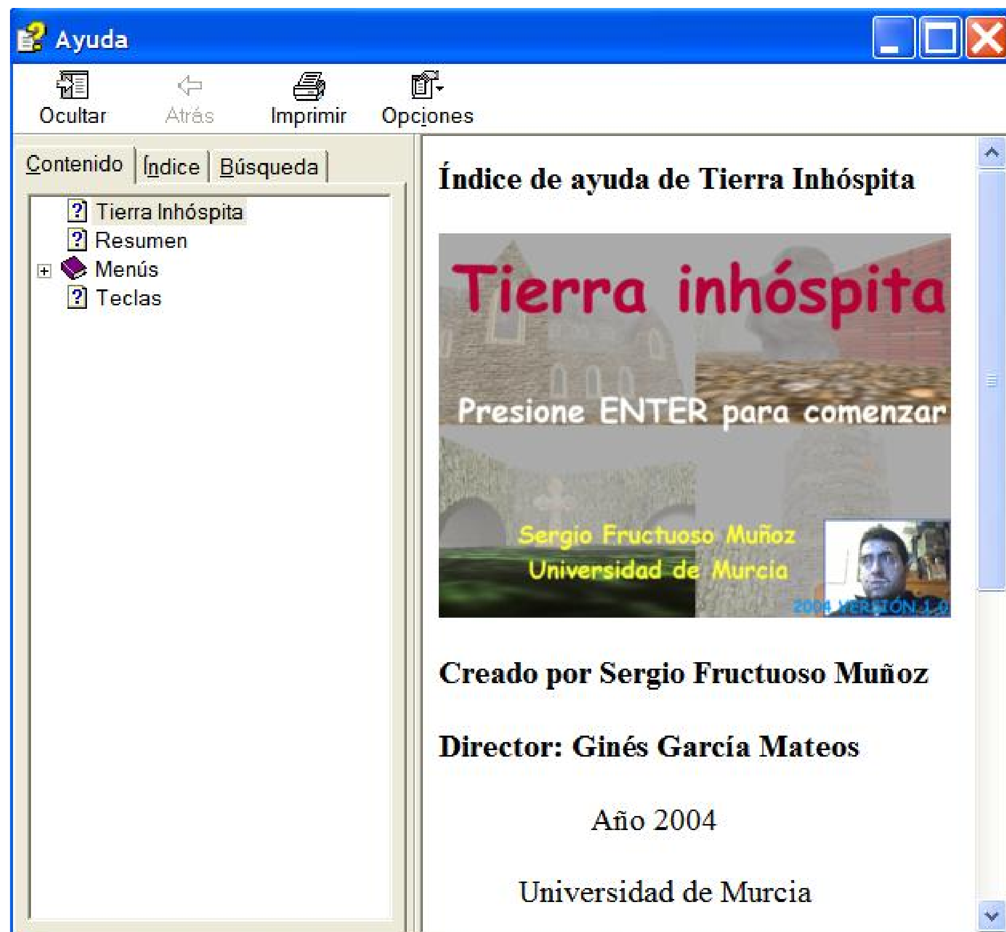


Imagen 7. Ayuda en formato chm.

- *Acerca de Tierra inhóspita…*: Aquí se nos muestra información respecto a la versión de la aplicación.