Práctica 7:

Visualizador de elementos gráficos

Objetivo de la práctica

Los objetivos de esta práctica son los siguientes:

- Presentar técnicas de informática gráfica usadas en un visualizador de objetos.
- Gestión de aplicaciones Java complejas en el entorno Eclipse.
- Identificación de los distintos elementos que aparecen en el visualizador.

1. El visualizador

1.1. Importando el visualizador

☞ Descarga el archivo *visualizador.zip* y descomprímelo en la carpeta del *workspace* de Eclipse. Vimos cómo hacerlo en la práctica 3: utilizamos la opción *"File: Import"* y a continuación seleccionamos *"General: Existing projects into workspace"* (ver Figura 1) y posteriormente la carpeta *PracticaVisualizador* , que contiene el proyecto a importar. Otra opción es crear un nuevo proyecto a partir de clases Java existentes. Para ello seleccionamos *"File: New: Java project"*, elegimos el nombre del proyecto, seleccionamos *"Create project from existing source*", pinchamos sobre *"Browse"* y seleccionamos la carpeta que contiene el código de las clases Java que queremos importar (ver Figura 2).

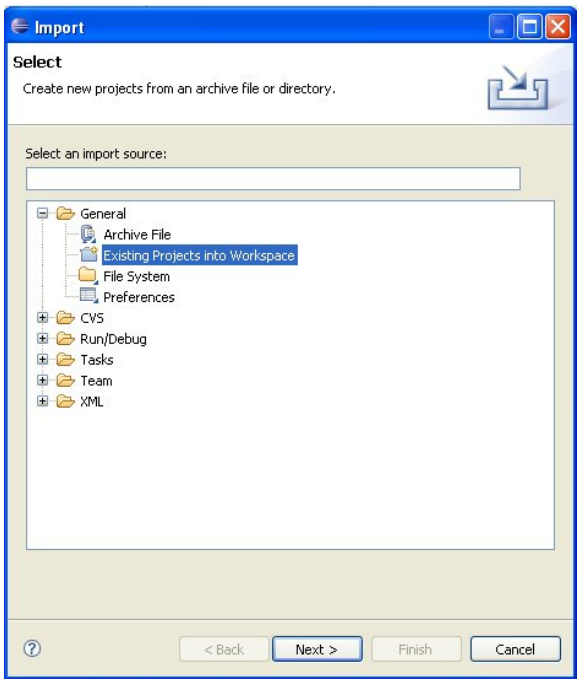

Fig. 1. Importar proyectos del workspace

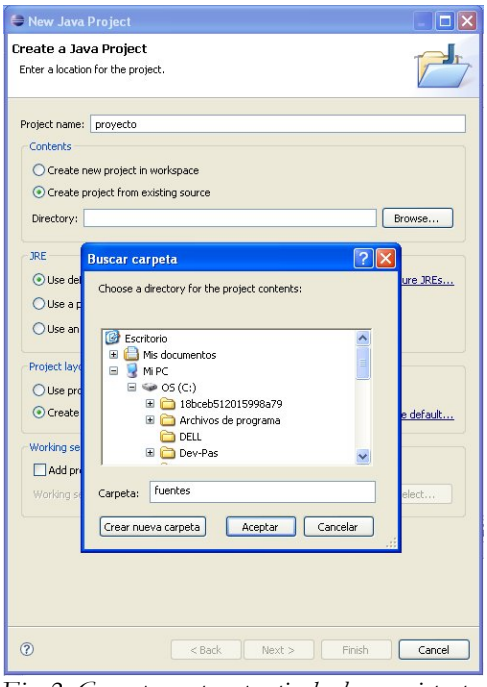

Fig. 2. Crear proyecto a partir de clases existentes

1.2. El código fuente: organización por paquetes

El visualizador es una aplicación de un tamaño suficientemente grande como para que el manejo de tantos ficheros comience ser un problema. Cuando eso ocurre, es recomendable agrupar las clases (representada por sus respectivos fichero fuente .java) en paquetes. Un paquete no es más que un conjunto de clases que representan conceptos similares.

El código fuente del visualizador está dividido en 3 paquetes

- **`geometria`:** clases que representan elementos geométricos.
- **`escena`:** clases que representan elementos de la escena, como colores, materiales, luces o la cámara.
- **`interfaz`:** clases que representan el interfaz gráfico.

Puedes ver estos paquetes en Eclipse en el explorador de paquetes / clases (parte izquierda), como se ve en la Figura 3. Estos paquetes son en los que se distribuye el código fuente de la aplicación. Ve a la carpeta del proyecto y verás que los paquetes se corresponden también con carpetas.

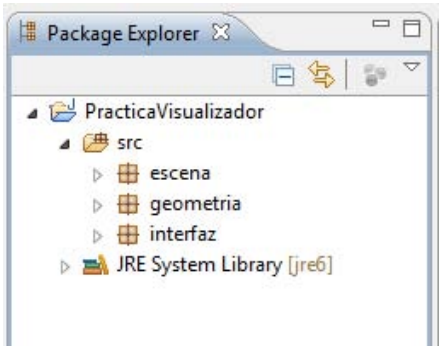

Fig. 3. Paquetes del código fuente del visualizador

☞ Tú también puedes crear un paquete para crear incluir nuevo código fuente. Haz click derecho sobre el proyecto, y ve a *New: Package*. Aparecerá una ventana (figura 4) en la que puedes ponerle nombre a dicho paquete. Ponle el nombre que quieras, y pulsa *Finish*.

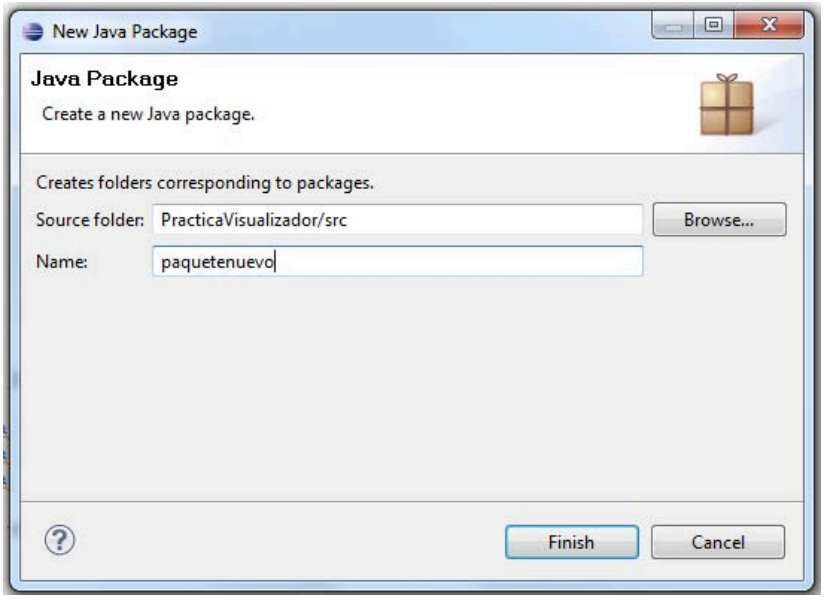

Fig. 4. Paquete nuevo

☞ Para incluir un archivo dentro de un paquete, basta con arrastrar y soltar. Arrastra y suelta el archivo **`Luz.java`**, que está en el paquete **escena**, en el paquete que acabas de crear. Te aparecerá un cuadro de diálogo. Asegúrate de que está pulsada la pestaña de *Update references to the moved elements*, y pulsa el botón *Preview*. Esto te llevará a un nuevo cuadro de diálogo (figura 5), en el que aparecen todos los cambios que se van a realizar en todo el código de tu proyecto. Revísalos. Esto es una de las ventajas de trabajar con una herramienta como Eclipse.

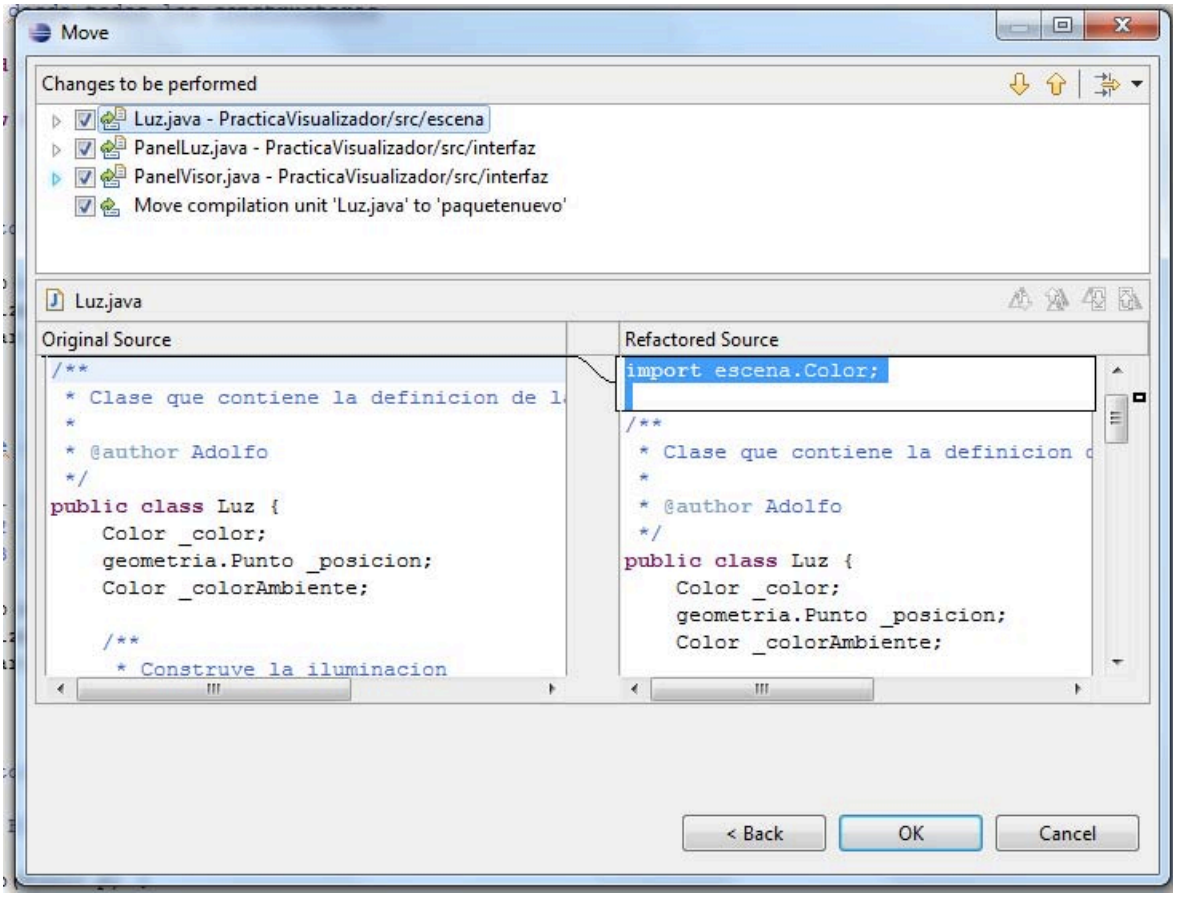

Fig. 5. Preview: arrastrar y soltar una clase produce una serie de cambios en el propio código fuente

☞ Para terminar, si quieres, vuelve a dejar el código como estaba, volviendo a poner el archivo **`Luz.java`** en su paquete original **escena**, y borra el paquete que has creado, pulsando el botón derecho sobre el paquete y seleccionando *Delete*. Date cuenta que, en realidad, el código fuente funciona exactamente igual aunque no restaures la distribución inicial de clases.

¿Por qué esos cambios de código? En realidad, cuando desde un archivo quieres acceder a una clase de otro archivo, tienes dos opciones: o escribir **<paquete>.<clase>** o bien poner al principio **import <paquete>.<clase>** y luego escribir **<clase>** cada vez que la quieras utilizar.

1.3. El código fuente: una clase, un archivo, un concepto

Para distribuir el código a lo largo de diferentes archivos, lo que suele (y debe) hacerse es organizarlo de tal forma que cada archivo contenga una clase. Desde el punto de vista conceptual, cada una de dichas clases representará un concepto concreto.

Por ejemplo, para trabajar con geometría (en el paquete **`geometria`**) hay 3 clases básicas:

- **`Punto`:** Punto en el espacio (vector de 3 componentes)
- **`Direccion`:** Dirección en el espacio (vector de 3 componentes)
- **`Normal`:** Normal en el espacio, dirección de longitud 1.

Un objeto (representado por la clase **`Objeto`**) no es más que una serie de caras y vértices. Un vértice del objeto es un punto junto con su normal, y una cara (clase **`Cara`**) de un objeto es una lista de vértices (indicando que existen aristas entre dos vértices consecutivos y entre el último vértice de la lista y el primero).

☞ Abre los archivos que corresponden con cada una de esas clases: **`Punto`**, **`Direccion`** y **`Normal`**. Repasa el código de cada una de ellas, y estudia su relación con el *Outline* que aparecerá en la ventana de la derecha (Figura 6).

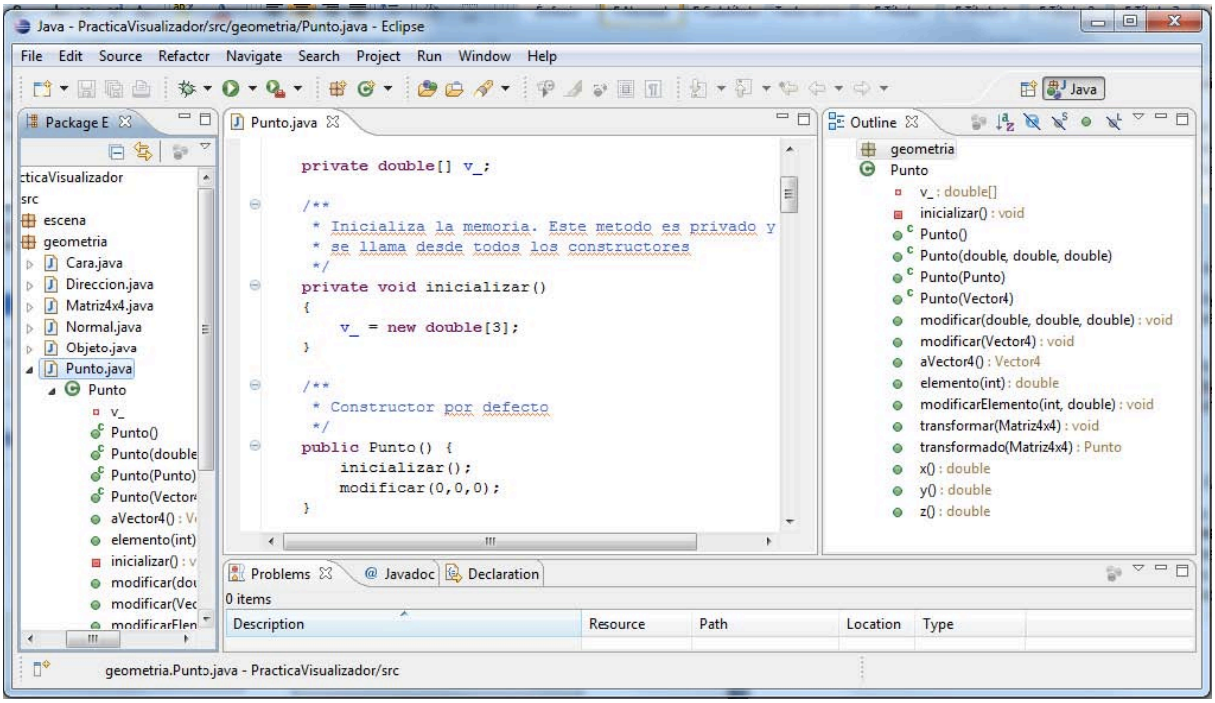

Fig. 6. Código y outline de la clase Punto

☞ Aunque todavía no entiendas el código, intenta relacionar lo que ves en el código con lo que ves en la parte derecha del interfaz gráfico de Eclipse, en el *Outline*. Deberías ver cierta relación entre ciertas líneas de código y los diferentes elementos del *Outline* de la clase.

Y, con tantas clases, ¿por dónde empieza a ejecutar la aplicación? Recuerda que toda aplicación empieza a ejecutarse a partir del método `public static void main(String s[])`

☞ ¿En qué clase está? Sólo puede haber uno por cada aplicación; búscalo (pista: está en una de las clases del paquete **`interfaz`**).

1.4. Ejecutando el visualizador

☞ Ejecuta el visualizador. El interfaz del visualizador es bastante intuitivo, pero vamos a probarlo.

☞ Lo primero y más importante es cargar un objeto tridimensional. Nuestro visualizador permite cargar objetos en formato *.obj*. Te proporcionamos una serie de objetos en este formato, que se pueden encontrar en el archivo *objetos3d.zip*. Para cargar un objeto, dentro del visualizador ve a *Archivo:Abrir*. Luego busca un archivo *.obj* (te recomendamos probar con **teapot.obj**, que es uno de los que te proporcionamos). El archivo tarda un tiempo en cargarse, pero cuando lo haga, podrás verlo en tu aplicación. Además, si cambias el modo de visualización de *malla de alambre* a *raster*, puedes ver el objeto sólido (Figura 7).

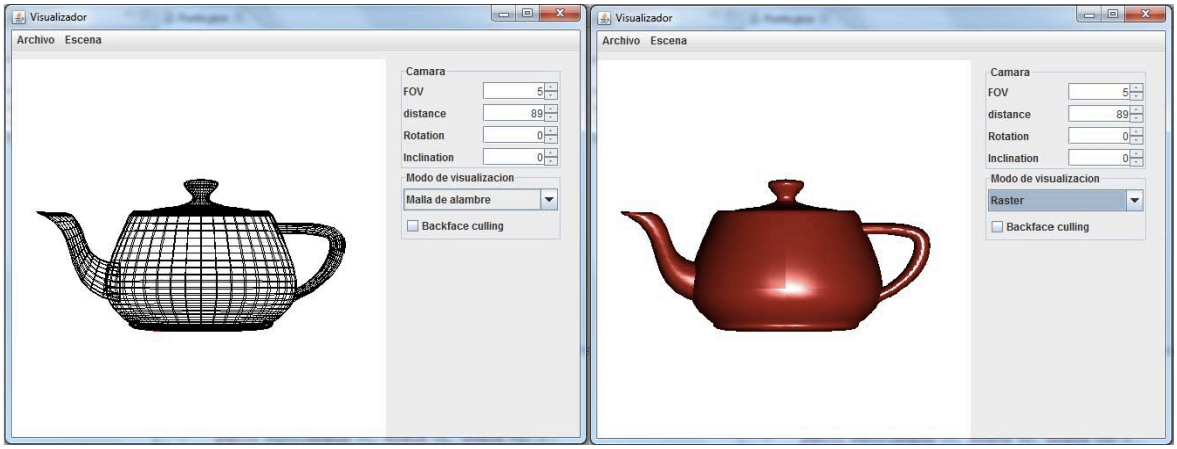

Fig. 7. Pantalla principal del visualizador. Izquierda: modo malla de alambre. Derecha: modo raster

Ahora vas a experimentar con la aplicación.

☞ Prueba qué ocurre cuando cambias los diferentes parámetros de la cámara, o cuando pulsas *backface culling*.

☞ Después, asegúrate de que estás en modo *raster* y prueba a cambiar las propiedades del material (menú *Escena: Material*). Intenta conseguir que tu objeto sea de color amarillo.

☞ Luego juega también con las propiedades de la luz (menú *Escena: Luz*).

1.5. Editando código

Habrás notado que el código que estás ejecutando hace que el texto de los parámetros de la cámara esté en inglés en el interfaz gráfico. Vamos a cambiarlo a castellano. Todo el interfaz gráfico relacionado con la cámara está en la clase **`PanelCamara`**.

☞ Busca las cadenas que representan el texto a traducir, y cámbialas para que pongan lo que tú quieras.

Pistas: Las cadenas en Java siempre vienen entre comillas dobles. Por ejemplo, “FOV”. Además, las cadenas del interfaz gráfico suelen meterse en etiquetas de Java, definidas por la clase **`JLabel`.** Las cadenas que tienes que cambiar siempre estarán en la misma línea de código que la palabra **`JLabel`**.

Ojo: si cambias algo que no sea una cadena puede que el código deje de funcionar bien o haya errores de compilación. Tranquilo, siempre podrás volverlo a dejar como estaba.

1.6. Depuración de errores

En una aplicación de este tamaño, empieza a tener sentido la depuración de errores. Como primera aproximación, vamos a probar a depurar el método **`matrizDeProyeccion()`** de la clase **`PanelVisor`** del paquete **`interfaz`**.

☞ Busca la línea:

```
geometria.Matriz4x4 proyeccion = new geometria.Matriz4x4();
```

y pon un punto de ruptura (botón derecho: *Toggle breakpoint*). Ejecuta tu aplicación en modo depuración de errores. Ve pulsando F6 para ir viendo cómo se va modificando la matriz de proyección en cada uno de los diferentes pasos.

Notarás que cuando hayas acabado, se dibujarán las ventanas de la aplicación pero volverá a la misma línea en la que estabas. ¿Por qué ocurre eso? Porque llega a esa línea cada vez que tiene que dibujar el objeto por pantalla, es decir, cada vez que la ventana se oculta tras otra ventana o se mueve. Como estás a la vez manejando Eclipse, la aplicación vuelve una y otra vez a la misma línea.

☞ Para evitar esto, botón derecho: *Disable breakpoint* sobre el punto de ruptura que has creado antes.

2. El visualizador por dentro

2.1. Proyectando 3 dimensiones en 2 dimensiones

¿Cómo funciona el visualizador? En realidad para visualizar un objeto conociendo sus puntos en un espacio tridimensional, lo primero que hay que hacer es transformar las tres coordenadas *(x,y,z)* del punto a coordenadas *(x,y)* de pantalla. La clase encargada de pintar el objeto por pantalla esta en el paquete **`interfaz`** y se llama **`PanelVisor`**.

Todas las transformaciones se realizan mediante matrices de cuatro dimensiones (representada por la clase **`Matriz4x4`**), para lo cual al punto hay que añadirle una cuarta coordenada, llamada coordenada homogénea:

$(x,y,z) \rightarrow (x,y,z,1)$.

La clase que representa un vector de 4 dimensiones se llama **`Vector4`**. Para transformar la clase **`Punto`** a la clase **`Vector4`** la clase **`Punto`** tiene un método llamado **`aVector4`**. La transformación de un vector con respecto a una matriz es un simple producto:

$$Vt = M \cdot V$$

- *Vt* es el vector transformado (es también un vector de cuatro componentes),
- *M* es la matriz,
- *V* el vector original.

Para volverlo a transformar en un **`Punto`** (3 componentes) hay que quitar su coordenada homogénea como sigue:

(x,y,z,w) → (x/w, y/w, z/w)

☞ Busca qué dos métodos de la clase **`Punto`** encapsulan este proceso y estudia su código.

La matriz de transformación que transforma las coordenadas de cada punto en coordenadas de pantalla se denomina *matriz de proyección*. Esta matriz se aplica a todos los puntos: la *x* y la *y* del resultado serán las coordenadas de pantalla en las que pintar. La *z* de ese punto nos servirá para el *z-buffer*, que guarda las posiciones *z* de todos los píxeles para evitar pintar un píxel que esté por delante de otro. Esta matriz de proyección es calculada por el método **`matrizDeProyeccion()`** de la clase **`PanelVisor`**.

☞ En el método **`matrizDeProyeccion()`** de **`PanelVisor`**, busca la siguiente línea:

```
proyeccion.rotacionX(camara().inclinacion() * Math.PI /180.0);
```

Coméntala (poniendo al principio de la línea una doble barra //, y se pondrá verde, como el resto de los comentarios del código) y vuelve a ejecutar la aplicación. ¿Qué ha cambiado?

☞ Prueba a comentar otras líneas diferentes del mismo método y comprueba qué cambia.

2.2. Matrices de transformación

Las transformaciones están representadas por matrices, y la clase **`Matriz4x4`** contiene todos los métodos que generan dichas matrices:

`traslacion(x,y,z)` $\rightarrow \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$

`escalado(x,y,z)` $\rightarrow \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

`rotacionX(a)` $\rightarrow \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(a) & \sin(a) & 0 \\ 0 & -\sin(a) & \cos(a) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

`rotacionY(a)` $\rightarrow \begin{pmatrix} \cos(a) & 0 & -\sin(a) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(a) & 0 & \cos(a) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

`rotacionZ(a)` $\rightarrow \begin{pmatrix} \cos(a) & \sin(a) & 0 & 0 \\ -\sin(a) & \cos(a) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

☞ Calcula numéricamente el resultado de rotar 90º con respecto al eje Z el punto (4, 3, 2). Para ello, habrá que multiplicar la matriz

$$\begin{pmatrix} 4 & 3 & 2 & 1 \end{pmatrix}$$

por la matriz de rotación correspondiente.

☞ Ejecuta el código en modo depuración para comprobar cómo se realiza una rotación.

La clase **`Matriz4x4`** también incluye un método para hacer cambios de perspectiva, siendo la matriz necesaria:

`perspectiva(fov,ar,n,f)` $\rightarrow \begin{pmatrix} (ar \cdot \tan(fov/2))^{-1} & 0 & 0 & 0 \\ 0 & \tan(fov/2)^{-1} & 0 & 0 \\ 0 & 0 & f/(f-n) & 1 \\ 0 & 0 & 0 & -n/(f-n) \end{pmatrix}$

En nuestro sistema:

- $n = 1$,
- $f = 100$,
- *ar* es el "*aspect ratio*", equivalente a la altura dividida por la anchura de la imagen,
- *fov* es el "*field of view*" o ángulo de cámara (uno de los parámetros de la clase **`Escena.Camara`**).

La siguiente matriz transforma coordenadas unitarias a coordenadas de píxeles de pantalla:

`pantalla(w,h)` $\rightarrow \begin{pmatrix} w/2 & 0 & 0 & -w/2 \\ 0 & h/2 & 0 & -w/2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

La matriz de transformación está ya en la clase **`PanelVisor`**, y la da el método **`matrizDeProyeccion`**:

$$M_p = pantalla(anchura, altura) \cdot perspectiva(fov, altura/anchura, 1, 100) \cdot traslacion(0,0,distancia) \cdot rotacionX(inclinacion) \cdot rotacionY(rotacion) \cdot traslacion(x_c, y_c, z_c)$$

donde:

- *Mp* es la matriz de proyeccion final, que se aplicará a todos los puntos y normales de todos los vértices de todas las caras del objeto,
- *anchura* y *altura* se obtienen del **`PanelVisor`**,
- *fov*, *distancia*, *inclinacion* y *rotacion* son parámetros de la **`Camara`**,
- *xc*, *yc* y *zc* son las coordenadas del centro del **`Objeto`**.

☞ Revisa el código de estos métodos para identificar dónde se implementa cada una de las transformaciones anteriores.

2.3. Modos de visualización

`PanelVisor` es la clase que se encarga de la visualización del **`Objeto`**, utilizando java2D. Hay dos modos de funcionamiento: *malla de alambre* y *raster*. Además, para acelerar los dos modos de pintado se puede activar la opción de *backface culling*. Esto hace que las caras cuya normal apunte en la dirección contraria a la dirección de la cámara (o sea, hacia atrás) no se consideren a la hora de pintar (método **`considerarCara`**).

☞ Revisa el código de este método y estúdialo.

En el modo *malla de alambre* se gestiona a través del método **`pintarMallaAlambre`**: se recorren todas las **`Caras`** del objeto, transformando los **`Puntos`** de dicha cara y pintando líneas (utilizando java2D) entre dos puntos adyacentes de la cara (método **`pintarCaraMallaAlambre`**).

☞ Revisa el código de estos métodos y estúdialo.

En el *modo raster* se utiliza el método de iluminación de Phong:

$$C_p = k_a C_a + k_d C_l \left| n_p d_l \right| + k_s C_l \left| r_p d_l^{e_s} \right|$$

donde las propiedades del material son:

- k_a es el coeficiente ambiental,
- k_d el coeficiente difuso,
- k_s el coeficiente especular,
- e_s el exponente especular (conforme mayor es, más fino e intenso es el reflejo especular),

las propiedades de la luz son:

- C_a el color ambiental,
- C_l el color de la luz,
- d_l la dirección desde el punto hasta la posición de la luz,

y las propiedades geométricas son:

- n_p la normal en el punto,
- r_p el vector reflejado de la cámara en ese punto.
- C_p representa el color de un píxel.

Los colores están en espacio de color RGB (rojo-verde-azul), y tanto los coeficientes del material como el color de la luz están en ese espacio.

☞ Ejecuta el visualizador, carga un objeto y ponlo en modo *raster*. Ahora, pon una luz completamente roja (con los canales verde y azul a 0) y el material completamente verde (con los canales rojo y azul a 0). La definición de luces y materiales la encontrarás en el visualizador, en el menú *Escena*. ¿De qué color te queda el objeto?

☞ Prueba con otras combinaciones de colores.

El algoritmo para pintar cara por cara del objeto en modo *raster* es un poco complicado. Para utilizar este método de iluminación, necesitamos conocer la normal de un píxel, y la posición tridimensional (punto) de un píxel. Además, hay que evitar pintar un píxel que está por detrás de otro. Esto se soluciona guardando un *z-buffer*, que guarda las posiciones *z* de cada píxel, de tal forma que al pintar un píxel guardamos su *z* en su posición del *z-buffer*, y si la *z* del píxel es mayor que la *z* que ya hay guardada en el z-buffer, entonces ese píxel estará por detrás de otro y no habrá que pintarlo. El algoritmo es el siguiente:

- Para cada Cara
  - Para cada Arista de la Cara (con los puntos transformados con la matriz de proyección)
    - Se recorren todos los píxeles de la arista, obteniendo para cada uno de esos píxeles su *z,* su posición y su normal interpolando los de los extremos de la arista (a esas tres cosas lo llamaremos “datos del píxel”).
    - Para cada fila (píxeles con la misma *y*) se buscan los datos del píxel de menor *x* y de mayor *x*. En esa fila se recorren todos los píxeles entre el menor y el mayor. Para cada uno de esos píxeles:
      - Se obtiene su *z* interpolando entre el píxel de menor *x* y el píxel de mayor *x*.
      - Si esa *z* es mayor que la *z* guardada en el z-buffer en esa posición, no se hace nada.
      - En caso contrario, se obtienen también el punto y la normal por interpolación, y se aplica la ecuación del modelo de Phong para obtener y guardar el color del píxel en esa posición.

☞ Busca qué 3 métodos implementan las acciones anteriores.

☞ Compara el código de los 2 métodos de visualización.

3. Conceptos de programación orientada a objetos

Crea un archivo de texto o un archivo de Word y responde a las siguientes preguntas:

☞ ¿En qué paquetes está dividido el código del visualizador?

☞ ¿Qué clases contiene cada paquete?

☞ Enumera los atributos de objeto, atributos de clase, métodos de objeto y métodos de clase de las siguientes clases:

- **`interfaz.VentanaPrincipal`**
- **`geometria.Matriz4x4`**
- **`geometria.Objeto`**

☞ Enumera las variables y sus tipos de los siguientes métodos:

- **`construirNormales`,** de la clase **`geometria.Objeto`**
- **`cargarObj`**, de la clase **`geometria.Objeto`**

☞ Escribe el tipo de los siguientes métodos:

- **`modificar`**, de la clase **`geometria.Punto`**
- **`interpretarComoVertice`**, de la clase **`geometria.Objeto`**

☞ Enumera los parámetros y sus tipos de los siguientes métodos:

- los dos métodos **`modificar`** de la clase **`geometria.Punto`**
- **`cargarObj`**, de la clase **`geometria.Objeto`**

☞ El método **`modificar`** (el que tiene tres parámetros) de la clase **`geometria.Punto`** es llamado durante la ejecución del programa. ¿Qué valores toman sus tres parámetros la primera vez que se llama a ese método? ¿Y la segunda vez?