UNIVERSIDAD
DE GRANADA

Informática Gráfica:
Teoría. Tema 4. Interacción. Animación.

Carlos Ureña

2022-23

Grado en Informática y Matemáticas
Grado en Informática y Administración y Dirección de Empresas
Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Teoría. Tema 4. Interacción. Animación.

Índice.

Sección 1.
Introducción.

Sistemas Gráficos Interactivos

Un **Sistema Gráfico Interactivo** (SGI) es un sistema software cuya respuesta a cada acción del usuario

- ocurre (por lo general) en un tiempo corto (del orden de décimas de segundo como mucho) desde dicha acción del usuario.
- se presenta al usuario en forma de visualización gráfica 2D o 3D

Un sistema SGI, por lo general, mantiene en memoria una estructura de datos (un modelo) y ejecuta un ciclo infinito, en cada iteración

1. espera o detecta una acción del usuario.
2. obtiene los datos que caracterizan dicha acción.
3. modifica el estado del modelo según dichos datos.
4. visualiza una nueva imagen obtenida a partir del nuevo estado del modelo

Interactividad

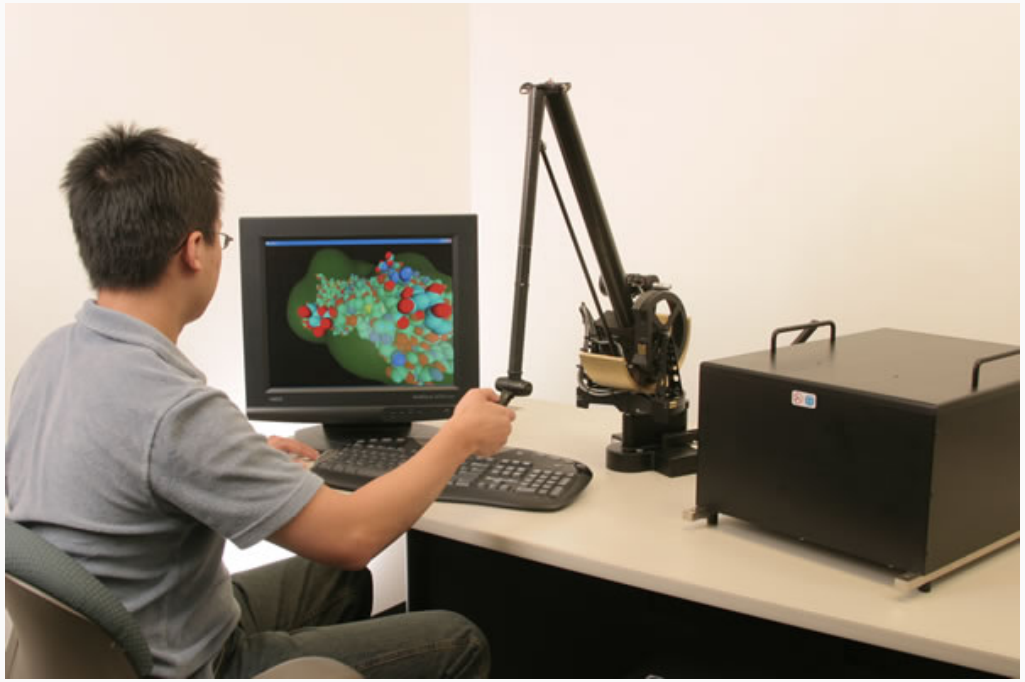
La incoporación de interactividad permite realizar aplicaciones que respondan ágilmente a las acciones de los usuarios y les ofrezcan retroalimentación sobre el efecto de dichas acciones.

La mayor parte de los sistemas gráficos son interactivos. Es esencial en:

- Videojuegos (*Videogames*) y Juego Serios (*Serious Games*).
- Sistemas de Diseño Asistido por Ordendor (CAD: *Computer Aided Design*)
- Sistemas de Realidad Virtual (VR: *Virtual Reality*)
- Sistemas de Realidad Aumentada (AR: *Augmented Reality*)
- Simuladores de aprendizaje (de conducción, de aviones, de barcos, etc...)

Dispositivos de entrada y salida

En un SGI el usuario debe disponer de al menos un dispositivo de entrada (p.ej. teclado, ratón) y un dispositivo de visualización (típicamente un monitor).



Hay otros dispositivos de entrada: tabletas digitalizadoras, sistemas de posicionamiento 3D.

Sistemas interactivos y de tiempo real

Los sistemas gráficos interactivos no siempre son **sistemas de tiempo real**:

- En un sistema interactivo se requiere que el retardo (latencia) entre la acción del usuario y la respuesta del sistema sea suficientemente pequeño como para que el usuario perciba una relacción de causa-efecto.
- No obstante, eventualmente la respuesta puede demorarse algo más.
- En un **sistema de tiempo real** la latencia debe ser menor o igual que un tiempo máximo de respuesta prefijado en las especificaciones del sistema. Un retraso superior a ese límite se considera un fallo del sistema.

Realimentación: utilidad

La realimentación es el mecanismo mediante el cual el sistema da información al usuario útil para permitir al usuario

- conocer más fácilmente el estado interno del sistema.
- ayudar a decidir la siguiente acción a realizar.

La información de realimentación que el sistema genera en cada momento depende lógicamente del estado del sistema y de la información previamente entrada por el usuario. Se puede usar con diferentes fines específicos:

- mostrar el estado del sistema
- como parte de una función de entrada
- para reducir la incertidumbre del usuario

Funciones de entrada en un SGI

Un sistema gráfico interactivo necesita normalmente funciones de entrada usuales en todo tipo de aplicaciones, p.ej:

- Entrada de una cadena de texto.
- Entrada de un valor numérico.
- Selección de un dato en una lista.

Además en un SGI se suelen necesitar otros tipos de entrada más específicos, por ejemplo, en un sistema CAD 3D podemos encontrar, entre otras, estas funciones:

- Lectura de posiciones 3D (selección de unas coordenadas específicas en el espacio de coordenadas del mundo)
- Selección de una componente de un modelo jerárquico 3D
- Entrada de los ángulos de rotación que determinan la orientación de un objeto.

Dispositivos físicos de entrada

El usuario introduce la información por medio de **dispositivos físicos de entrada**. Estos pueden ser

- de propósito general: p.ej. el teclado, o
- específicos para datos geométricos: p.ej. digitalizador.

Podemos clasificar los dispositivos de entrada gráfica atendiendo a la información que generan de forma directa. Esta puede ser:

- **Posiciones 2D**: tableta digitalizadora, lápiz óptico, pantalla táctil.
- **Posiciones 3D**: digitalizador, tracking.
- **Desplazamientos 2D**: ratón, trackball, joystick.
- **Imágenes o vídeos**: cámaras de fotografía o vídeo.

Dispositivos lógicos de entrada

Un **dispositivos lógico de entrada** es una componente software que usa uno o varios dispositivos físicos de entrada para producir información de más alto nivel o mas elaborada, obtenida a partir de los datos recibidos directamente de los dispositivos físicos (o indirectamente de otros dispositivos lógicos). Ejemplos:

- **Puntero del ratón:** permite entrar puntos en pantalla a partir de los desplazamientos físicos del ratón y del estado de sus botones.
- **Selector de componentes** (*Picker*): permite seleccionar un componente de un modelo 3D usando el puntero de ratón.
- **Fotografía 3D:** permite entrar imágenes con información de profundidad, a partir de pares de imágenes obtenidas con dos cámaras de fotografía convencionales.

Lectura de datos dispositivos de entrada: modos de entrada.

Un **modo de entrada** es un método que usa una aplicación para decidir cuando debe consultar los datos relacionados con el estado (y los cambios de estado) de un dispositivo físico o lógico.

- Distintos tipos de dispositivos pueden tener asociados distintos modos de funcionamiento.
- Algunos tipos de dispositivos se pueden usar con más de un modo de funcionamiento.

Veremos los tres modos básicos más frecuentes:

- **modo de muestreo**: la aplicación consulta del estado actual en instantes arbitrarios.
- **modo de petición**: la aplicación espera hasta que se produzca un cambio de estado.
- **modo de cola de eventos**: la aplicación recibe una lista de cambios de estado no procesados aún.

Estado y eventos de un dispositivo

Los cambios de estado que ocurren en un dispositivo de entrada (físico o lógico) se denominan **sucesos** o **eventos**.

- El **estado** de un dispositivo en un instante es el conjunto de valores de las variables gestionadas por el driver del dispositivo, y que representan en memoria su estado físico. P.ej:
  - En un teclado: un vector de valores lógicos que indican, para cada tecla, si dicha tecla está pulsada o no está pulsada.
  - En un ratón: estado de los dos botones (dos lógicos) y posición actual en pantalla del cursor (dos enteros).
- Un evento tiene asociados ciertos datos:
  - Instante de tiempo en el que el cambio ha ocurrido o se ha registrado
  - Información sobre: el estado inmediatamente después del evento, y sobre como ha cambiado el estado respecto al anterior al evento.

Modo de muestreo

Un dispositivo puede usarse **modo de muestreo**: (**sample**):

- El software del dispositivo mantiene en memoria variables que representan el estado actual del dispositivo.
- La aplicación puede consultar dichas variables en cualquier momento, sin espera alguna.

Ventajas/Desventajas

- Es muy eficiente en tiempo y memoria, y simple.
- Requiere a la aplicación emplear tiempo de CPU en muestrear a una frecuencia suficiente como para no perderse posibles cambios de estado relevantes.
- No hay información de cuando ocurrió el último cambio de estado.

Ejemplo: en un teclado, array de valores lógicos que indica, para cada tecla, si está pulsada o levantada.

Modo de petición.

Para evitar perder eventos relevantes, la aplicación puede usar el **modo petición** (**request**):

- La aplicación hace una petición y espera a que se produzca determinado tipo de evento.
- Cuando se produce, la aplicación recibe datos del evento.

Ventaja/Desventajas

- Nunca se perderá el siguiente evento tras hacer una petición.
- Puede perderse un evento si no se hace una petición antes de que ocurra.
- Se puede perder mucho tiempo esperando (no se puede hacer otras cosas).

Ejemplo: en un teclado, esperar hasta que se pulse una tecla alfanumérica, y entonces saber de que tecla se trata.

Modos cola de eventos

En el modo **cola de eventos**

- Cada vez que ocurre un evento, el software del dispositivo lo añade a una cola FIFO de eventos pendientes de procesar.
- La aplicación accede a la cola, extrae cada evento y lo procesa.

Ventajas:

- No se pierde ningún evento.
- La aplicación no está obligada a consultar con cierta frecuencia, ni antes de cada evento.
- La aplicación no pierde tiempo en esperas si es necesario hacer otras cosas (se funciona en modo asíncrono).

Ejemplo: en un teclado, acceder a la lista de pulsaciones de teclas, ocurridas desde la última vez que se consultó.

Sección 2.
Eventos en GLFW.

Modelo de eventos de GLFW

GLFW gestiona los dispositivos de entrada en modo *cola de eventos*:

- Para cada tipo de evento se puede registrar una función de la aplicación que procesará eventos de ese tipo (cada función se llama **callback** o **función gestora de eventos**, FGE).
- Las *callbacks* se ejecutan al producirse un evento del tipo.
- Los eventos cuyo tipo no tiene *callback* asociado se ignoran.
- Los parámetros de un *callback* proporcionan información del evento.
- La aplición debe incluir explícitamente un bucle de gestión de eventos, en cada iteración se espera hasta que se han procesado todos y luego se visualiza un frame o cuadro de imagen, si es necesario (si ha cambiado el estado de la aplicación).

Funciones para registrar *callbacks*

Para cada tipo de evento, GLFW contiene una función para registrar el *callback* asociado a dicho tipo. Las funciones de registro y los tipos de eventos asociados son:

- **glfwSetMouseButtonCallback**: pulsar/levantar de botones del ratón.
- **glfwSetCursorPosCallback**: movimiento del cursor del ratón.
- **glfwSetWindowSizeCallback**: cambio de tamaño de la ventana.
- **glfwSetKeyCallback**: pulsar/levantar una tecla física.
- **glfwSetCharCallback**: pulsar una tecla (o una combinación de ellas) que produce un único carácter unicode.

Eventos de botones del ratón

Son los eventos que ocurren cuando se pulsa o se levanta un botón del ratón. Para registrar el callback asociado, usamos esta llamada:

```
glfwSetMouseButtonCallback( ventana, FGE_PulsarLevantarBotonRaton )
```

El callback debe estar declarado así:

```
void FGE_PulsarLevantarBotonRaton( GLFWwindow* window, int button,
                                   int action, int mods );
```

Los parámetros permiten conocer información del evento:

- **button**: botón afectado (valores: **GLFW_MOUSE_BUTTON_LEFT**, **GLFW_MOUSE_BUTTON_MIDDLE**, **GLFW_MOUSE_BUTTON_RIGHT**)
- **action** : estado posterior del botón afectado, indica si se ha pulsado o levantado (valores: **GLFW_PRESS**, **GLFW_RELEASE**)
- mod: estado de teclas de modificación en el momento de levantar o pulsar (*shift*, *control*, *alt* y *super*).

Ejemplo de *callback* de botón del ratón

Este *callback* (**FGE_BotonRaton**) se encarga de procesar una pulsación del botón izquierdo o derecho del ratón

```
void FGE_PulsarLevantarBotonRaton( GLFWwindow* window, int button,
                                   int action, int mods )
{
   if ( action == GLFW_PRESS ) // si se ha pulsado un botón
   {
      double x,y ;
      glfwGetCursorPos( window, &x, &y ); // leer posición del ratón
      if ( button == GLFW_MOUSE_BUTTON_LEFT ) // si se ha pulsado izquierdo:
         RatonClickIzq( int(x), int(y) );      // hacer acción asociada
      else if ( button == GLFW_MOUSE_BUTTON_RIGHT )
      {  xclick_der = int(x); // registra coord. X de pulsación bot. der.
         yclick_der = int(y); // idem coord. Y
      }
   }
}
```

Usamos **glfwGetCursorPos** para leer la posición. La función **RatonClickIzq** se encarga de procesar el click izquierdo como sea necesario.

Eventos de movimiento del cursor del ratón

Son los eventos que ocurren cada vez que se mueve el ratón. Se pueden registrar con esta llamada:

```
glfwSetCursorPosCallback( ventana, FGE_MovimientoRaton )
```

El callback debe estar declarado con tres parámetros enteros:

```
void FGE_MovimientoRaton( GLFWwindow* window, double xpos, double ypos )
```

Los parámetros permiten conocer información del evento:

- **xpos**,**ypos**: posición del cursor en coordenadas de pantalla, en el momento del evento.

Ejemplo de *callback* de movimiento activo del ratón

Si se registra este *callback*, podemos, por ejemplo, registrar el desplazamiento cada vez que se mueve el ratón estando pulsado el botón derecho (en combinación con **`glfwGetMouseButton`** para saber si está pulsado el botón derecho)

```
void FGE_MovimientoRaton( GLFWwindow* window, double xpos, double ypos )
{
 if ( glfwGetMouseButton( window, GLFW_MOUSE_BUTTON_RIGHT ) == GLFW_PRESS)
 { const int
      dx = int(xpos) - xclick_der ,  // calcular desplazamiento en X desde click
      dy = int(ypos) - yclick_der ;  // calcular desplazamiento en Y desde click
   RatonArrastradoDer( dx, dy );
 }
}
```

La función **`RatonArrastradoDer`** podría ser cualquier función que se ejecuta cuando se mueve el ratón con el botón derecho pulsado. Recibe como parámetros los desplazamientos desde que se pulsó dicho botón derecho.

Eventos de *scroll*. Ejemplo.

Este *callback* sirve para detectar movimientos de la rueda del ratón o de otros mecanismos de scroll (por ejemplo, gestos en un *touchpad*). Se pueden detectar movimientos tanto verticales como horizontales, en función del dispositivo usado.

```
glfwSetScrollCallback( ventana, FGE_Scroll );
```

El callback debe estar declarado con estos parámetros

```
void FGE_Scroll( GLFWwindow* window, double xoffset, double yoffset  )
```

A modo de ejemplo, si queremos detectar únicamente scroll vertical

```
void FGE_Scroll( GLFWwindow* window, double xoffset, double yoffset  )
{
   if ( fabs( yoffset) < 0.05 ) // poco movimiento vertical -> ignorar evento
      return ;
   const int direccion = 0.0 < yoffset ? +1 : -1 ;
   ScrollVertical( direccion );  // función que procesa scroll (+1 o -1)
}
```

El bucle de gestión de eventos

Se encarga de procesar eventos y visualizar:

```
terminar  = false ;  // escrito en los callbacks para señalar que se debe terminar.
redibujar = true ;   // escrito en los callbacks para señalar que hay que redibujar.
while ( ! terminar ) // hasta que no sea necesario terminar...
{  if ( redibujar )
   {  VisualizarEscena(); // visualizar la ventana si es necesario
      redibujar = false; // no volver a visualizar si no es necesario
   }
   glfwWaitEvents();      // esperar y procesar eventos
   terminar = terminar || glfwWindowShouldClose( glfw_window ) ;
}
```

- **`glfwWaitEvents`** procesar todos los eventos pendientes, o bien, si no hay ninguno, espera bloqueada hasta que haya al menos un evento pendiente y entonces procesarlo (llama a los *callbacks* que haya registrados).
- **`glfwWindowShouldClose`** devuelve **`true`** solo si el usuario ha realizado alguna acción de cierre de ventana en el gestor de ventanas.

Bucle de gestión de eventos con animaciones

Para animaciones, se ejecuta una **función de actualización de estado** cuando no hay eventos pendientes de procesar:

```
terminar  = false ;  // escrito en los callbacks para señalar que se debe terminar.
redibujar = true ;   // escrito en los callbacks para señalar que hay que redibujar.
while ( ! terminar ) // hasta que no sea necesario terminar...
{ if ( redibujar )
  {  VisualizarEscena();    // visualizar la ventana si es necesario
      redibujar = false ;   // no volver a visualizar si no es necesario
  }
  glfwPollEvents();         // procesar eventos pendientes (sin esperar)
  if ( ! terminar && ! redibujar ) // si no terminamos ni redibujamos
     ActualizarEscena();           // actualizar el estado del modelo
  terminar = terminar || glfwWindowShouldClose( glfw_window ) ;
}
```

- **`glfwPollEvents`**: si hay eventos pendientes, los procesa todos, en otro caso no hace nada.
- **`ActualizarEscena`**: se invoca continuamente, actualiza el modelo al siguiente estado de la animación.

Bucle de gestión de eventos genérico

En realidad las animaciones pueden activarse o desactivarse:

```
terminar  = false ; // escrito en los callbacks para señalar que se debe terminar.
redibujar = true ;  // escrito en los callbacks para señalar que hay que redibujar.
animacion = false ; // true solo cuando están activadas las animaciones
while ( ! terminar )
{  if ( redibujar )          // si ha cambiado algo:
   {  VisualizarEscena();    // dibujar la escena
      redibujar = false;     // evitar que se redibuje continuamente
   }
   if ( animacion )          // si la animación está activada:
   {  glfwPollEvents();      // procesar eventos pendientes (sin esperar)
      if ( ! terminar && ! redibujar ) // si no terminamos ni redibujamos:
         ActualizarEscena();           // actualizar estado
   }
   else                      // si las animaciones está desactivadas:
      glfwWaitEvents();      // esperar que se produzcan eventos y procesarlos
   terminar = terminar || glfwWindowShouldClose( ventana_glfw ) ;
}
```

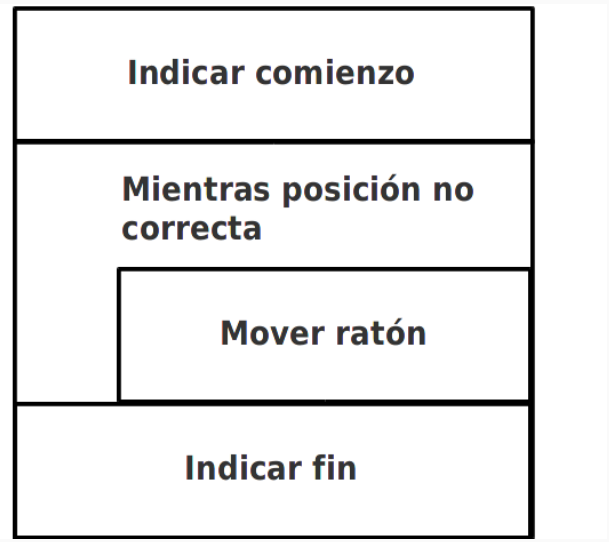
- **animacion**: tiene el estado de las animaciones.

Sección 3.
Posicionamiento.

Posicionamiento: acciones del usuario.

El **posicionamiento** es la operación que permite al usuario seleccionar fácilmente un punto en el espacio de coordenadas del mundo de una aplicación 2D o 3D.

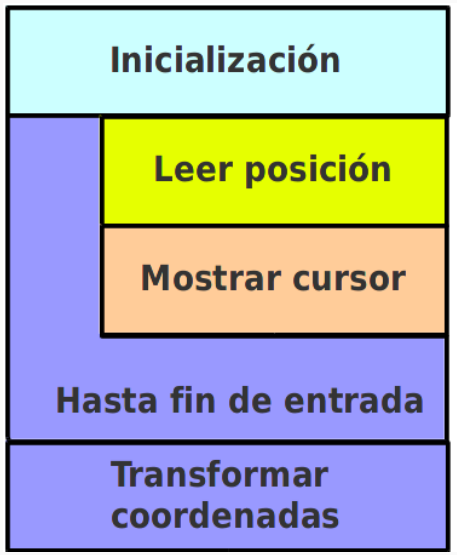
- Esto se lleva a cabo usando un dispositivo lógico (de tipo cursor) que permite seleccionar pixels en pantalla.
- El usuario opera mejorando iterativamente su selección hasta que la juzga correcta. La realimentación es esencial.



Posicionamiento: pasos de la aplicación.

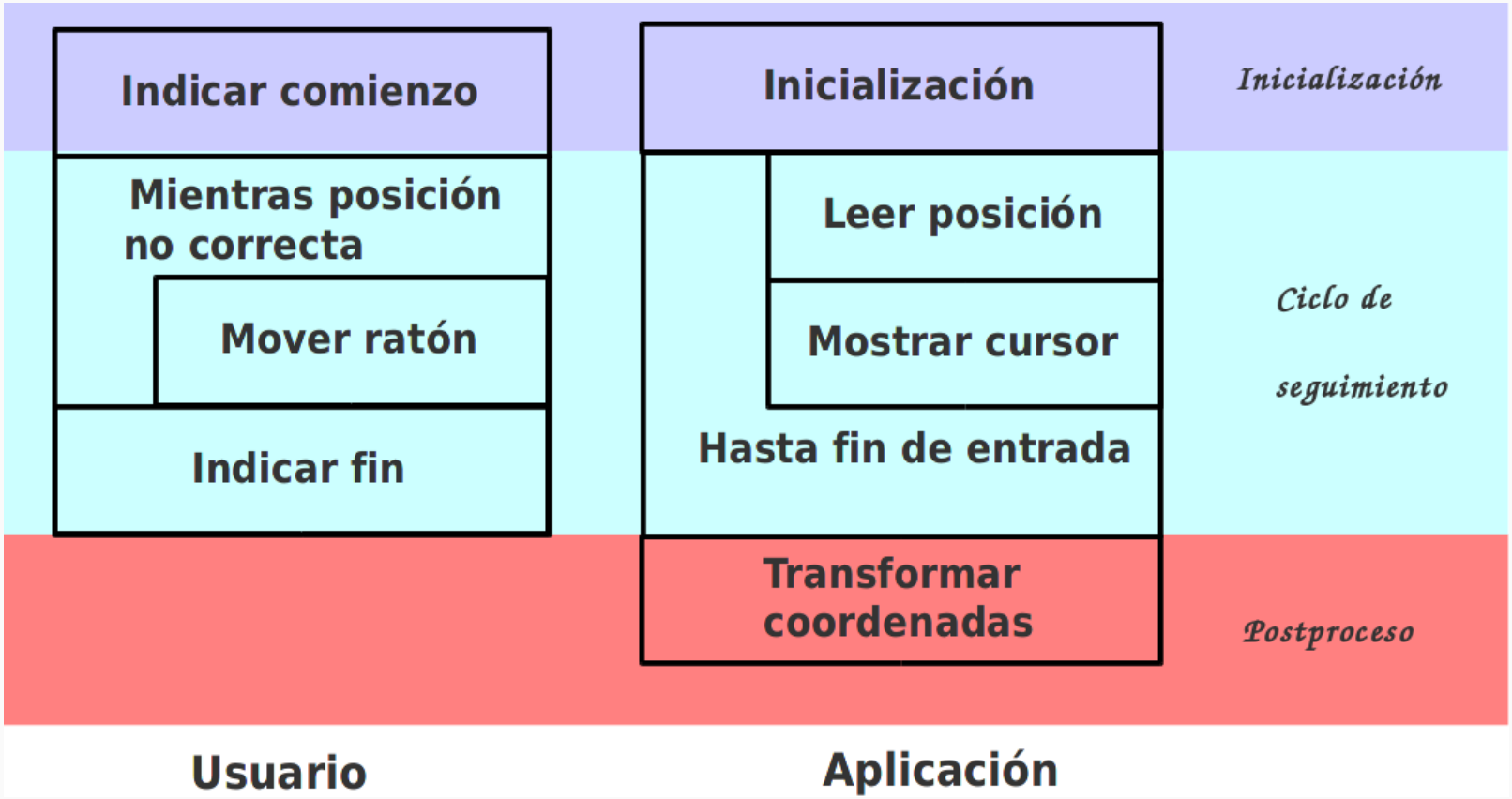
El proceso en el software de tratamiento se realiza en una estructura cíclica, que se corresponde con el ciclo de búsqueda que realiza el usuario. En este proceso podemos distinguir tres etapas:

- **Inicialización:** incialización de estado.
- **Ciclo de seguimiento:** hasta que el usuario no indica que está satisfecho, se ejecuta un bucle en el cual: (a) se procesan los eventos de entrada y (b) se actualiza la información visual de realimentación y la posición seleccionada.
- **Postproceso:** se transforma la posición final.



Acciones del usuario y del sistema.

Correspondencia entre operaciones del usuario y del sistema



Modos de introducción de coordenadas

En visualización 2D de modelos planos (con una matriz de proyección ortogonal O y una matriz de dispositivo D), usamos:

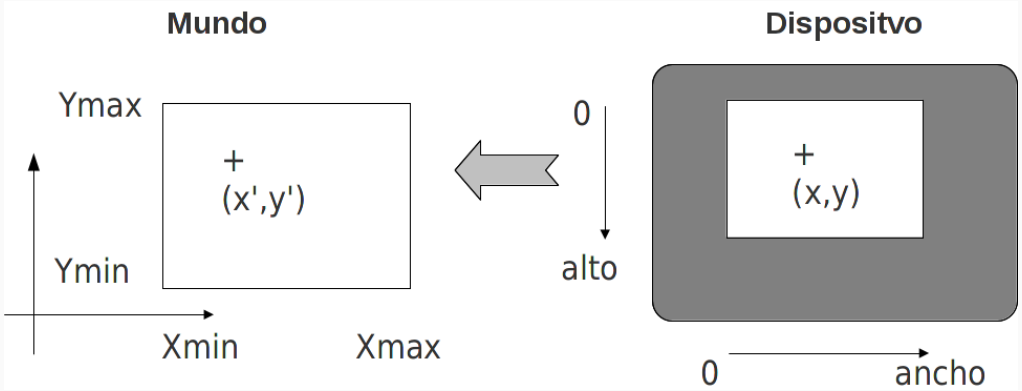
- **Transformación lineal**: se usa la matriz D^{-1} para pasar de DC a NDC, y luego O^{-1} para pasar de NDC a WC (la matriz usada es $M = O^{-1}D^{-1}$).

En visualización de modelos 3D en pantalla, hay varias estrategias

- **Restricción a un plano**: el punto seleccionado se obtiene proyectando un punto 2D (obtenido con un cursor convencional 2D) sobre un plano 3D.
- **Cursor virtual 3D**: se manipulan las tres coordenadas en una vista ortográfica.
- **Tres cursores 2D ligados**: se usan tres vistas ortográficas perpendiculares.

Posicionamiento 2D: transformación lineal

Usamos una transformación lineal para convertir desde (x_d, y_d) (en DC) hacia (x_w, y_w) (en WC) (las coordenadas DC son enteras, proporcionadas por el gestor de ventanas):

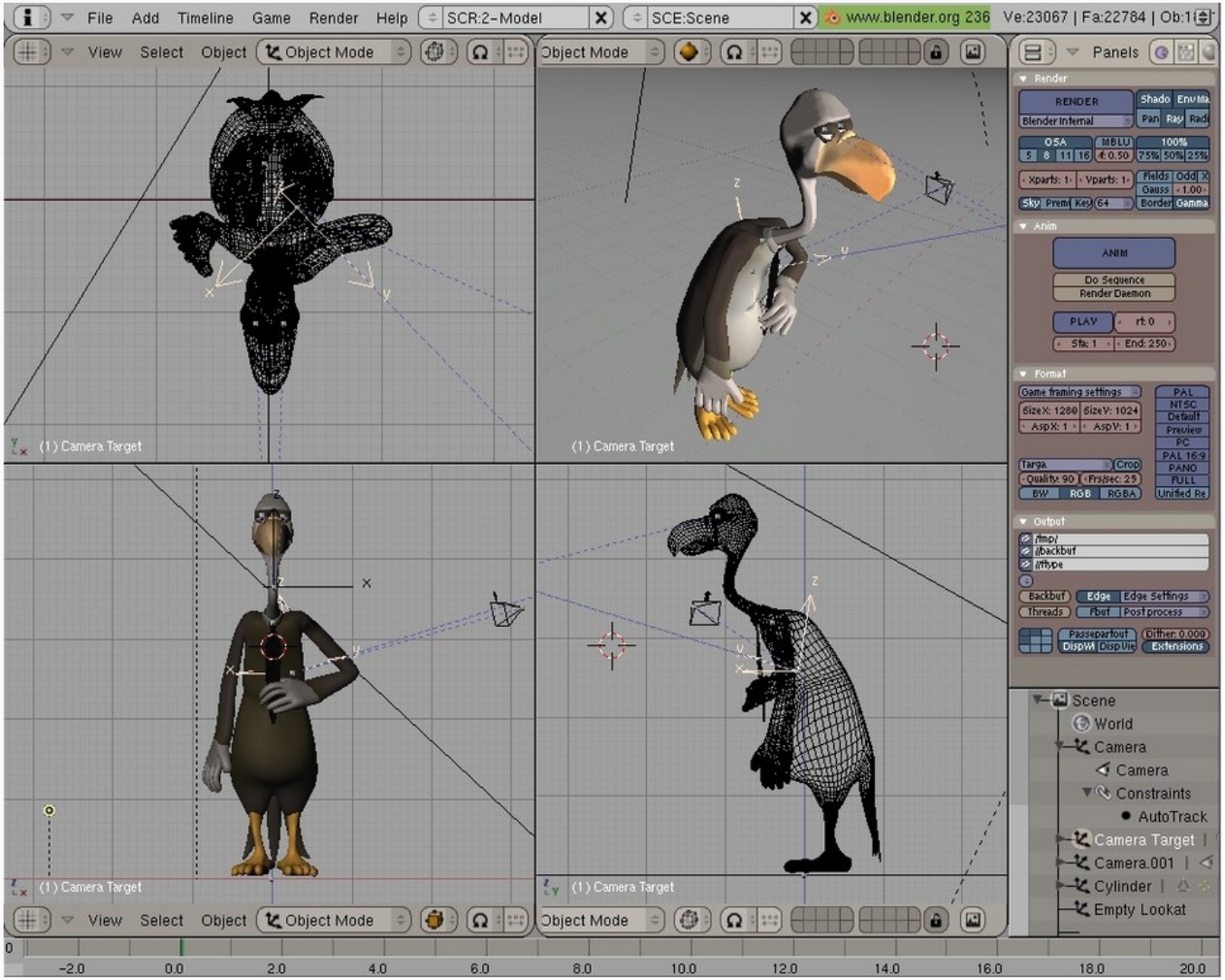


$$x_w = l + (r-l)\left(\frac{x_d + 1/2}{n_x}\right) \qquad y_w = t - (t-b)\left(\frac{y_d + 1/2}{n_y}\right)$$

- l, r son los límites del *view-frustum* 2D en X.
- b, t son los límites del *view-frustum* 2D en Y.
- n_x, n_y son el ancho y el alto (en pixels) del *viewport*.

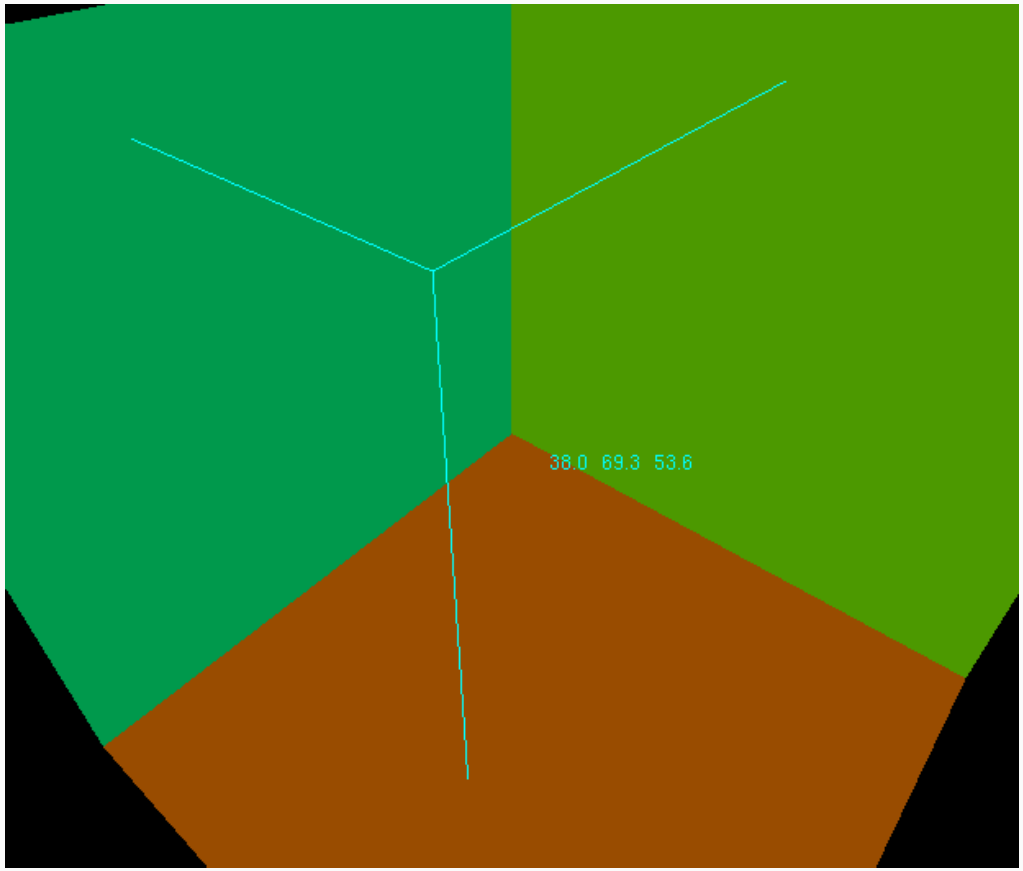
Posicionamiento en 3D: tres cursores 2D ligados

Se usan tres proyecciones ortográficas perpendiculares:



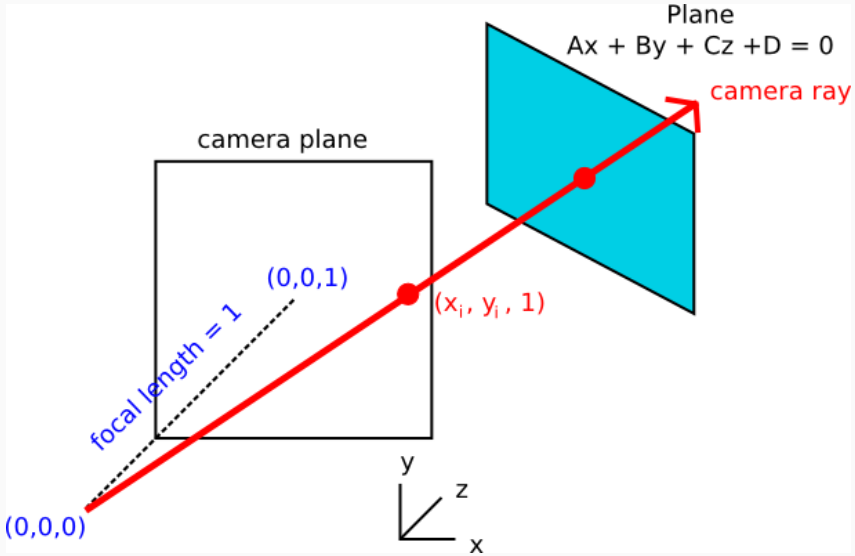
Posicionamiento 3D: Cursor virtual 3D

Se usa un cursor virtual en 3D. El desplazamiento se realiza en el plano X-Z o en el X-Y en función de los botones del ratón que estén pulsados.



Posicionamiento 3D: restricción a un plano

Si se restringe la posición a un plano que no sea perpendicular al de proyección se puede obtener una posición 3D por intersección de la recta que pasa por el punto introducido (en el plano de proyección y el centro de proyección con el plano



Entrada de transformaciones

Las transformaciones geométricas se pueden definir a partir de puntos.

- Una traslación se puede definir por el vector que va de la posición original a la posición nueva
- Una rotación por el ángulo formado por el vector que va del punto de referencia a la posición actual con la horizontal

Sección 4.
Control de cámaras.

4.1. Modelo y operaciones de cámaras
4.2. Modos de cámara. La cámara de 3 modos.

Control interactivo de la cámara: modos

Un **modo de cámara** es una de modificar interactivamente (con retroalimentación) los parámetros de la transformación de vista. Hay varios:

- **Modo orbital** (o **examinar**): útil para visualización de objetos, la cámara mantiene como punto de atención el origen de un objeto, y rota alrededor del mismo.
- **Modo primera persona**: útil para exploración de escenarios, manipulamosla la cámara cambiando su
  - **posición**: se desplaza la posición del observador en el sentido de los ejes de la cámara.
  - **orientación**: se rotan los ejes de la cámara entorno a la posición del observador (que no cambia).

esto da lugar a tres modos distintos de control de cámara.

Cámara en modo orbital o examinar.

En este modo el usuario puede manipular la cámara virtual, pero siempre se mantiene el **punto de atención** proyectado en el centro de la imagen:

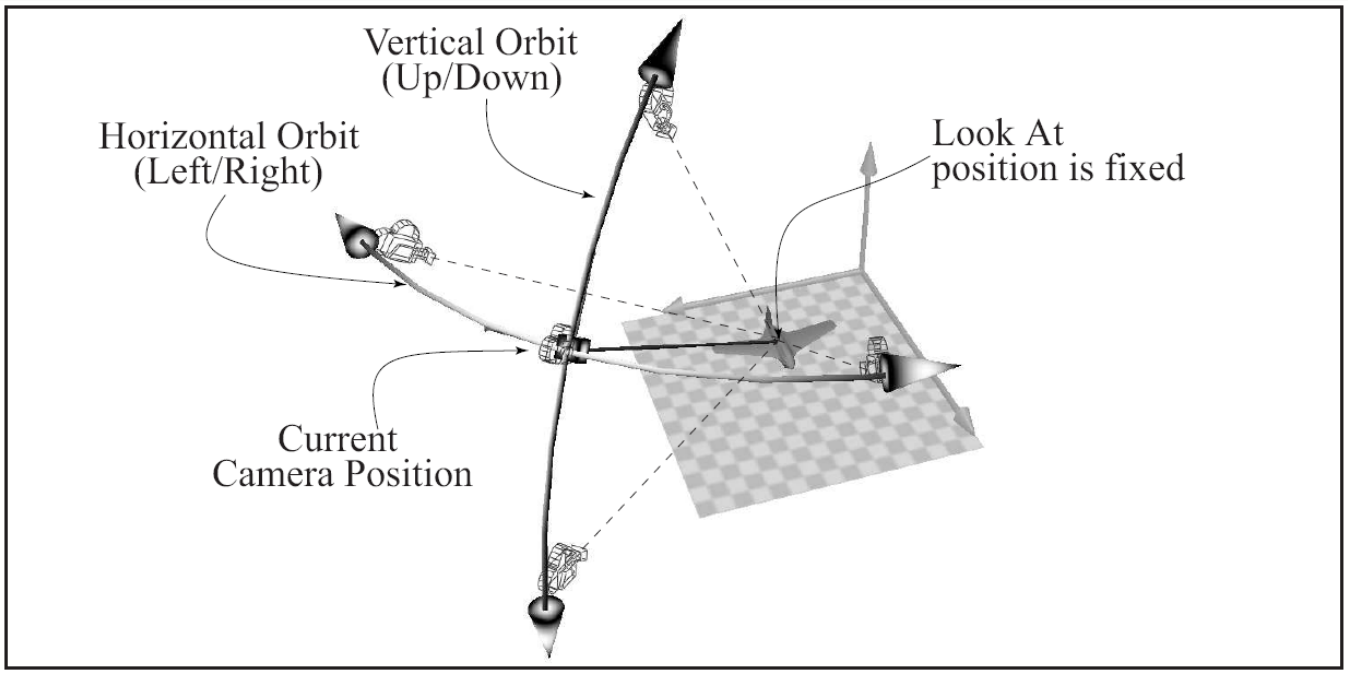


Figura obtenida de: *K.Sung, P.Shirley, S.Baer* **Essentials of Interactive Computer Graphics: Concepts and Implementation**.

Subsección 4.1.
Modelo y operaciones de cámaras.

Parámetros de la cámara: Marco $\mathcal{V}$ y matriz de vista V.

Suponemos que el marco de coordenadas de la vista $\mathcal{V}$ tiene como componentes $[\vec{x}_{\text{ec}}, \vec{y}_{\text{ec}}, \vec{z}_{\text{ec}}, \dot{o}_{\text{ec}}]$, y dichas componentes están representadas en memoria por sus coordenadas homogéneas $\mathbf{x}, \mathbf{y}_{\text{ec}}, \mathbf{z}_{\text{ec}}$ y $\mathbf{o}_{\text{ec}}$ en el marco de coordenadas del mundo $\mathcal{W}$:

$$\begin{array}{lcllcl} \mathbf{x}_{\text{ec}} & = & (a_x, a_y, a_z, 0)^t & \mathbf{y}_{\text{ec}} & = & (b_x, b_y, b_z, 0)^t \\ \mathbf{z}_{\text{ec}} & = & (c_x, c_y, c_z, 0)^t & \mathbf{o}_{\text{ec}} & = & (o_x, o_y, o_z, 1)^t \end{array}$$

La matriz de vista V es la composición de una matriz de traslación por $(-o_x, -o_y, -o_z)$ seguida de una matriz cuyas filas son las coordenadas de los ejes de $\mathcal{V}$, es decir:

$$V = \begin{pmatrix} a_x & a_y & a_z & 0 \\ b_x & b_y & b_z & 0 \\ c_x & c_y & c_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -o_x \\ 0 & 1 & 0 & -o_y \\ 0 & 0 & 1 & -o_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Parámetros de cámara: origen y punto de atención.

Podemos determinar una cámara usando el **punto de atención** y el **vector al origen**

- **Punto de atención** $\dot{a}_t$: es un punto del espacio que se va a proyectar en el centro de la imagen (está en el la rama regativa del eje Z de la cámara).
- **Vector al origen** $\vec{n}$: es el vector que va desde el punto de atención hasta el origen del marco de coordenadas de la camara, es decir $\vec{n} = \dot{o}_{\text{ec}} - \dot{a}_t$

Podemos expresar el marco de cámara en función de $\vec{n}$ y $\dot{a}_t$:

$$\begin{aligned} \vec{z}_{\text{ec}} &= \vec{n} / \|\vec{n}\| & \qquad \vec{y}_{\text{ec}} &= \vec{z}_{\text{ec}} \times \vec{x}_x \\ \vec{x}_{\text{ec}} &= (\vec{y}_w \times \vec{z}_{\text{ec}}) / \|\vec{y}_w \times \vec{z}_{\text{ec}}\| & \qquad \dot{o}_{\text{ec}} &= \dot{a}_t + \vec{n} \end{aligned}$$

($\vec{z}_{\text{ec}}$ es paralelo a $\vec{n}$, y el vector VUP siempre es $\vec{y}_w$).

Modelo de cámaras. Coordenadas esféricas.

Para representar una cámara podemos usar las tuplas $\mathbf{a}_t$, $\mathbf{s}$ y $\mathbf{n}$:

- Coordenadas del punto de atención ($\mathbf{a}_t$), en WC.
- Coordenadas esféricas y cartesianas del vector al origen. Las cordenadas esféricas forman una tupla $\mathbf{s} = (\alpha, \beta, r)$, y las coordenadas cartesianas son una terna $\mathbf{n} = (n_x, n_y, n_z)$, ambas en WC.

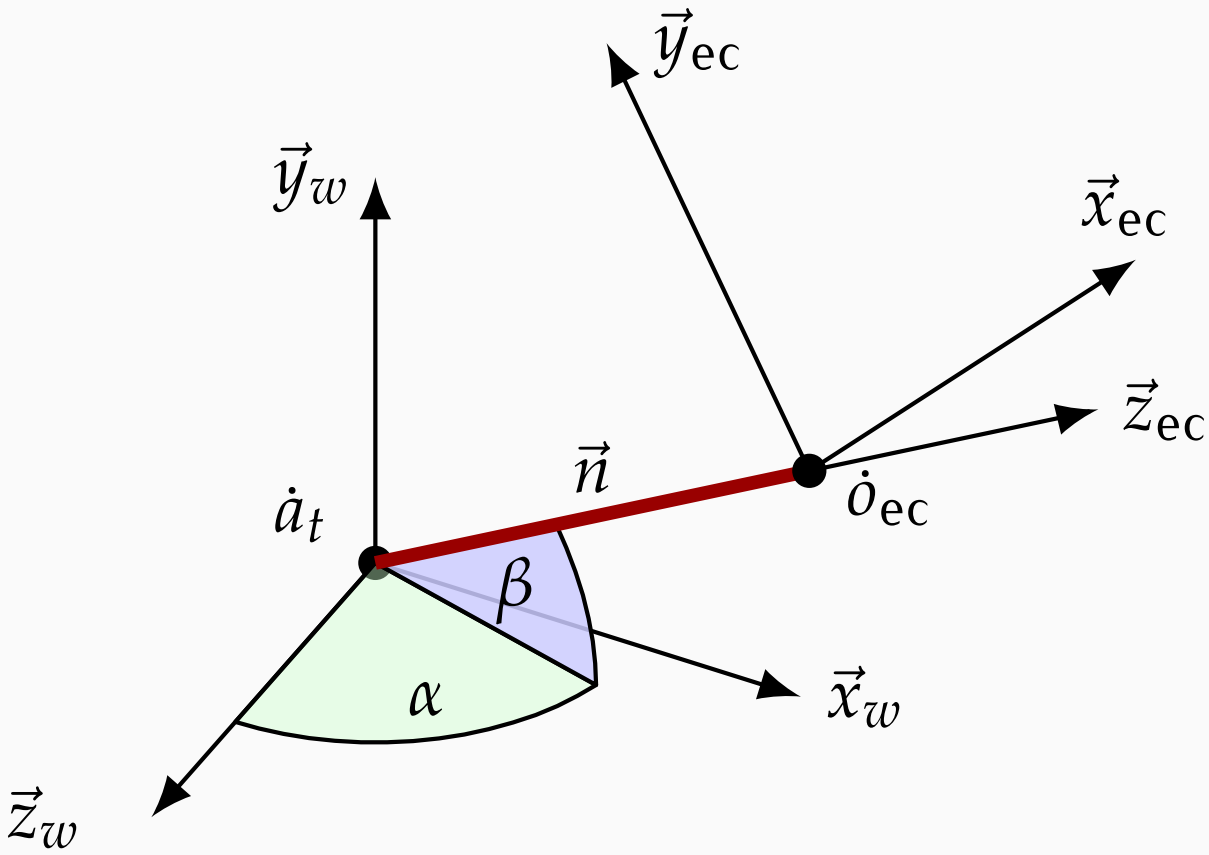
Se usa una representación redundante por comodidad. Se cumple:

$$\begin{aligned} n_x &= r(\sin\alpha)(\cos\beta) \\ n_y &= r(\sin\beta) \\ n_z &= r(\cos\alpha)(\cos\beta) \end{aligned} \qquad \begin{aligned} \alpha &= \mathbf{atan2}(n_x, n_z) \\ \beta &= \mathbf{atan2}(n_y, r_h) \\ r &= \|\mathbf{n}\| \end{aligned}$$

donde: $r_h = \sqrt{n_x^2 + n_z^2}$ y $\mathbf{atan2}(a,b)$ es como $\mathbf{arctan}(a/b)$ pero teniendo en cuenta los signos y con valores en $(-\pi, \pi]$.

Elementos del modelo de cámara

En esta figura se observan el marco de la cámara $\mathcal{V}$ con origen en $\dot{o}_{\text{ec}} = \dot{a}_t + \vec{n}$, junto con el marco del mundo $\mathcal{W}$ (trasladado a $\dot{a}_t$):



Cámaras interactivas de 3 modos

En general podemos hacer tres operaciones de modificación interactiva de una cámara:

- **Izquierda/derecha**: rotaciones en torno a $\vec{y}_w$ o traslaciones paralelas a $\vec{x}_{\text{ec}}$
- **Arriba/abajo**: rotaciones en torno a $\vec{x}_{\text{ec}}$ o traslaciones paralelas a $\vec{y}_{\text{ec}}$
- **Adelante/detrás**: traslaciones paralelas a $\vec{n}$.

Cada vez que se modifica el estado de una cámara:

1. se actualizan las tuplas $\mathbf{a}_t, \mathbf{n}$ y $\mathbf{s}$
2. se recalcula el marco de cámara (tuplas $\mathbf{o}_{\text{ec}}, \mathbf{x}_{\text{ec}}, \mathbf{y}_{\text{ec}}$ y $\mathbf{z}_{\text{ec}}$).

A las cámaras que se pueden actualizar así las llamamos **cámaras interactivas**.

Clase base Camara: declaración

La clase base para cualquier cámara es la siguiente:

```
class Camara
{
   public:
   // fija las matrices model-view y projection en el cauce
   void activar( Cauce & cauce )  ;
   // cambio el valor de 'ratio_vp' (alto/ancho del viewport)
   void fijarRatioViewport( const float nuevo_ratio ) ;
   // lee la descripción de la cámara (y probablemente su estado)
   virtual std::string descripcion() ;

   protected:
   bool     matrices_actualizadas = false;// true si matrices actualizadas
   Matriz4f matriz_vista = MAT_Ident(),   // matriz de vista
            matriz_proye = MAT_Ident();   // matriz de proyección
   float    ratio_vp     = 1.0 ;          // ratio viewport (alto/ancho)

   // actualiza matriz_vista y matriz_proye a partir de los parámetros
   // específicos de cada tipo de cámara
   virtual void actualizarMatrices() ;
} ;
```

Representación de cámaras interactivas

La clase **CamaraInteractiva** es una clase derivada de **Camara** que puede ser manipulada con estas operaciones:

```
class CamaraInteractiva : public Camara
{
   public:
   // operación izq/der (da) y arriba/abajo (db)
   virtual void desplRotarXY( const float da, const float db ) = 0 ;
   // operación acercar/alejar (dz)
   virtual void moverZ( const float dz ) = 0 ;
   // cambiar punto de atención manteniendo origen de cámara
   virtual void mirarHacia( const Tupla3f & paten )  ;
   // cambiar el modo de la camara al siguiente modo o al primero
   virtual void siguienteModo()  ;
} ;
```

Esta clase define un interfaz pero no se puede instanciar. Se definen clases derivadas de ella. Usaremos dos: **CamaraOrbitalSimple** y **Camara3Modos**. Estudiaremos la segunda.

Subsección 4.2.
Modos de cámara. La cámara de 3 modos..

La clase para cámaras de tres modos

La clase **Camara3Modos** implementa los tres modos:

```
class Camara3Modos : public CamaraInteractiva
{
public:
 Camara3Modos();  // cámara perspectiva por defecto
 Camara3Modos( ..... ) ; // cámara con parámetros iniciales específicos
 virtual void desplRotarXY( const float da, const float db )  ;
 virtual void moverZ( const float dz )  ;
 virtual void mirarHacia( const Tupla3f & nuevo_punto_aten );//pasa a m.ex.
 virtual void siguienteModo();      // ir al siguiente modo
 virtual Tupla3f puntoAtencion() ; // devuelve el punto de atencion actual
private:
 virtual void actualizarMatrices(); // actualiza matriz V y P
 void actualizarEjesMCV() ;          // actualiza ejes del MCV
 ModoCam modo_actual     = ModoCam::examinar;  // modo actual
 Tupla3f punto_atencion  = { 0.0,0.0,0.0 };    // punto de atención
 Tupla3f org_polares     = { 0.0,0.0,d_ini };  // vector origen (esféricas)
 Tupla3f org_cartesianas = { 0.0,0.0,d_ini };  // vector origen (cartesianas)
 Tupla3f eje[3] = { {1,0,0},{0,1,0},{0,1,0} }; // ejes del MCV
} ;
```

Aquí **d_ini** es el radio inicial (en las prácticas es 3 unidades).

Cámara en primera persona con traslaciones

En el modo de primera persona con traslaciones, la actualización de la cámara supone simplemente trasladar el origen del marco de cámara $\mathbf{o}_{\text{ec}}$ y el punto de atención $\mathbf{a}_t$ de forma solidaria:

- La operación **`desplRotarXY`**(Δ_a, Δ_b) supone:
  1. $\mathbf{a}_t = \mathbf{a}_t + \Delta_x \mathbf{x}_{\text{ec}} + \Delta_y \mathbf{y}_{\text{ec}}$
  2. $\mathbf{o}_{\text{ec}} = \mathbf{a}_t + \mathbf{n}$
- La operación **`moverZ`**(Δ_z) supone:
  1. $\mathbf{a}_t = \mathbf{a}_t + \Delta_z \mathbf{z}_{\text{ec}}$
  2. $\mathbf{o}_{\text{ec}} = \mathbf{a}_t + \mathbf{n}$

Las tuplas $\mathbf{s}, \mathbf{n}, \mathbf{x}_{\text{ec}}, \mathbf{y}_{\text{ec}}, \mathbf{z}_{\text{ec}}$ no cambian.

Cámara primera persona con rotaciones

En este caso se usan rotaciones entorno al origen de la cámara $\mathbf{o}_{\text{ec}}$. Dichas rotaciones se implementan modificando los ángulos α y β que hay en $\mathbf{s}$. El movimiento en Z es similar al anterior

- La operación **`desplRotarXY`**(Δ_a, Δ_b) supone:
  1. $\mathbf{s} = \mathbf{s} + (\Delta_a, \Delta_b, 0)$ (incrementa o decrementa α y β)
  2. $\mathbf{n}' = \text{Cartesianas}(\mathbf{s})$ (es el nuevo valor de $\mathbf{n}$).
  3. $\mathbf{a}_t = \mathbf{a}_t - (\mathbf{n}' - \mathbf{n})$ (trasl. pto de atención a nueva pos.)
  4. $\mathbf{n} = \mathbf{n}'$
  5. actualizar $\mathbf{x}_{\text{ec}}, \mathbf{y}_{\text{ec}}$ y $\mathbf{z}_{\text{ec}}$ (el origen $\mathbf{o}_{\text{ec}}$ no cambia)
- La operación **`moverZ`**(Δ_z) supone:
  1. $\mathbf{a}_t = \mathbf{a}_t + \Delta_z \mathbf{z}_{\text{ec}}$
  2. $\mathbf{o}_{\text{ec}} = \mathbf{a}_t + \mathbf{n}$ (los vectores $\mathbf{x}_{\text{ec}}, \mathbf{y}_{\text{ec}}$ y $\mathbf{z}_{\text{ec}}$ no cambian)

Este modo es típico en videojuegos FPS (*First Person Shooter*), normalmente sin movimientos de rotación en vertical ($\Delta_b = 0$)

Cámara en modo orbital o examinar

En este caso se usan rotaciones entorno al punto de atención $\mathbf{a}_t$. El movimiento en Z nos acerca o aleja a dicho punto (cambia el valor de r que hay en $\mathbf{s}$):

- La operación **`desplRotarXY`**(Δ_a, Δ_b) supone:
  1. $\mathbf{s} = \mathbf{s} + (\Delta_a, \Delta_b, 0)$ (incrementa o decrementa α y β)
  2. $\mathbf{n} = \mathbf{Cartesianas}(\mathbf{s})$ (es el nuevo valor de $\mathbf{n}$).
  3. $\mathbf{o}_{\mathrm{ec}} = \mathbf{a}_t + \mathbf{n}$, actualizar $\mathbf{x}_{\mathrm{ec}}, \mathbf{y}_{\mathrm{ec}}, \mathbf{z}_{\mathrm{ec}}$

- La operación **`moverZ`**(Δ_z) supone:
  1. $r = r_{min} + (r - r_{min})(1 + \epsilon)^{\Delta_z}$
  2. $\mathbf{n} = \mathbf{Cartesianas}(\mathbf{s})$
  3. $\mathbf{o}_{\mathrm{ec}} = \mathbf{a}_t + \mathbf{n}$ (los vectores $\mathbf{x}_{\mathrm{ec}}, \mathbf{y}_{\mathrm{ec}}$ y $\mathbf{z}_{\mathrm{ec}}$ no cambian)

El radio nunca es inferior a $r_{min} > 0$. Para $\Delta_z \geq 1$ aleja, y para $\Delta_z \leq -1$ acerca (Δ_z nunca debe estar en $(-1, 1)$).

Apuntar la cámara hacia un punto

Esta operación supone hacer que el punto de atención se fije a unas coordenadas de mundo $\mathbf{c}$ dadas, sin modificar el origen de cámara.

La operación `mirarHacia(c)` supone:

1. $\mathbf{n} = \mathbf{n} + \mathbf{a}_t - \mathbf{c}$
2. $\mathbf{s} = \text{Esfericas}(\mathbf{n})$
3. $\mathbf{a}_t = \mathbf{c}$
4. actualizar $\mathbf{x}_{ec}, \mathbf{y}_{ec}$ y $\mathbf{z}_{ec}$ (el origen $\mathbf{o}_{ec}$ no cambia)

La fución Esfericas produce las coordenadas esféricas a partir de las cartesianas (es la inversa de Cartesianas).

Esta operación permite seleccionar un objeto y que pase a ocupar el centro de la imagen (fijando el punto de atención a un punto central de dicho objeto).

Problema: animación de cámaras

Problema 4.1.

Supongamos que queremos visualizar una secuencia de frames, en los cuales la cámara va cambiando. Para ellos queremos escribir el código de una función que fija la matriz de vista en el cauce. La función acepta como parámetro un valor real t, que es el tiempo en segundos transcurrido desde el inicio de la animación. Suponemos que la animación dura s segundos en total.

En ese tiempo el observador de cámara se desplaza con un movimiento uniforme desde un punto de coordenadas de mundo $\mathbf{o}_0$ (para $t = 0$) hasta un punto destino $\mathbf{o}_1$ (para $t = 1$). Además el punto de atención de la cámara también se desplaza desde $\mathbf{a}_0$ hasta $\mathbf{a}_1$. Durante toda la animación, el vector VUP es $(0, 1, 0)$.

Escribe el pseudo-código de la citada función.

Sección 5.
Selección.

5.1. Introducción. Métodos de selección

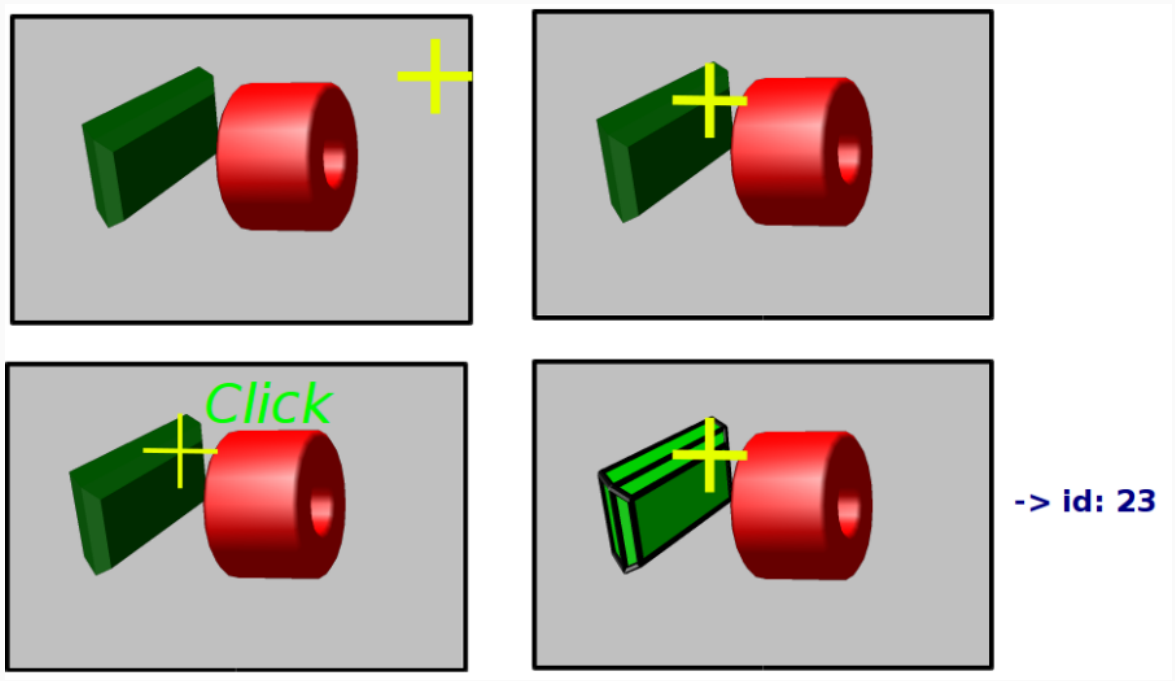
5.2. Selección con *frame-buffer object* invisible.

Subsección 5.1.
Introducción. Métodos de selección.

Selección de objetos

La selección permite al usuario identificar componentes u objetos de la escena:

- Se necesita para identificar el objeto sobre el que actuan las operaciones de edición.
- Suele realizarse como posicionamiento seguido de búsqueda.



Identificadores de objetos

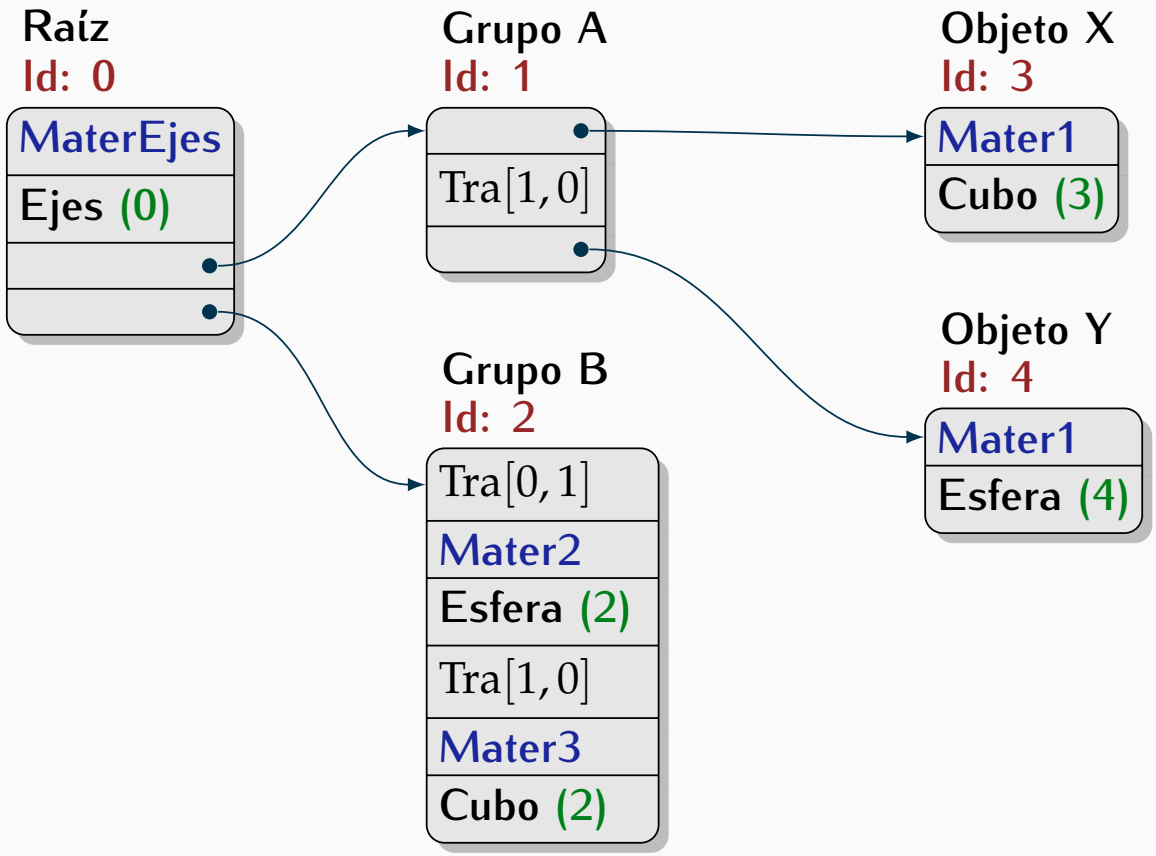
Para poder realizar la selección los componentes de la escena deben tener asociados identificadores numéricos (enteros), a distintos niveles:

- **Triángulos**: cada triángulo o cara tiene asociado un entero, permite seleccionarlos para operaciones de edición de bajo nivel en las mallas.
- **Mallas**: cada malla de la escena puede tener un identificador único.
- **Grupos de objetos**: los grupos de mallas (o, en general, grupos de objetos arbitrarios), pueden tener asociados identificadores, permiten operar con partes complejas de la escena.

Para ediciones de alto nivel en objetos o partes de la escena, lo más simple es **asignar identificadores enteros a los nodos del grafo de escena**.

Grafo de escena con identificadores

Todos los objetos (instancias de **`Objeto3D`**) tienen asociado un valor entero: -1 significa que heredan el identificador del padre, 0 que no es seleccionable, > 0 que es seleccionable con ese identificador.



Procedimiento y métodos de selección

Para hacer la selección se pueden dar estos pasos:

1. El usuario selecciona un pixel en pantalla .
2. Se buscan los identificadores de los elementos (triángulos, mallas u objetos) que se proyectan en el centro de dicho pixel (o de pixels cercanos).

La búsqueda se puede hacer de varias formas:

- **Ray-casting**: calculando intersecciones de un rayo (semirecta con origen en $\mathbf{o}_c$ y pasando por el centro del pixel) con los objetos de la escena.
- **Clipping**: (*recortado*) calculando que objetos están parcial o totalmente dentro de un *view-frustum* pequeño centrado en el pixel.
- **Rasterización**: visualizar la escena por rasterización usando identificadores en el lugar de los colores. Permite obtener el color (un identificador de objeto) del pixel en cuestión.

Selección por rasterización en OpenGL (1/2)

En OpenGL podemos usar visualización (por rasterización), de dos formas:

- **Modo selección de OpenGL:** se usa funcionalidad de OpenGL, específica para este fin, esta funcionalidad es declarada obsoleta desde OpenGL 3.0 y eliminada desde OpenGL 3.1.
- **Frame-buffer no visible**. Se usa algún **frame buffer object**, que es una zona de memoria de la GPU que contiene una imagen sobre la que se puede visualizar.

Modo de selección de OpenGL.

Se usa una funcionalidad de OpenGL, específica para este fin, requiere:

- Visualizar con el **modo de selección** activado, OpenGL usa identificadores en lugar de colores (tomados de la **pila de nombres**).
- Los identificadores visualizados se registran en un **buffer de selección** (en memoria) específicamente destinado a contenerlos.

En nuestro caso, no usaremos esta funcionalidad, al no estar disponible en OpenGL 3.3.

Selección con un *framebuffer* invisible

Se usa algún **frame buffer object** (array de colores de pixels en la memoria de la GPU) , hay dos opciones:

- Usar uno de los dos framebuffers que se suelen existir en OpenGL para la técnica de *double buffering*.
- Usar un *framebuffer object* específicamente creado para esto. No depende de la existencia de *double buffering* y del acceso a sus buffers.

Nosotros usamos la segunda opción. Se debe:

- Crear y activar un *Frame Buffer Objects* (FBOs) del tamaño y características que queramos.
- Visualizar la escena sobre el FBO usando identificadores en lugar de colores.
- Leer identificador de objeto en el pixel.

Problema: intersección rayo-triángulo (1/2)

Problema 4.2.

Una posibilidad para hacer selección en mallas de triángulos es usar cálculo de intersecciones entre un *rayo* (una semirrecta que pasa por el centro de un pixel) y cada uno de los triángulos de la malla.
Diseña un algoritmo en pseudo-código para el cálculo de intersecciones entre un rayo y un triángulo:

- El rayo tiene como origen o extremo el punto cuyas coordenadas del mundo es la tupla $\mathbf{o}$, y como vector de dirección la tupla $\mathbf{d}$ (la suponemos normalizada).
- Las coordenadas del mundo de los vértices del triángulo son $\mathbf{v}_0$,$\mathbf{v}_1$ y $\mathbf{v}_2$.
- El algoritmo debe de indicar si hay intersección o no, y, en caso de que la haya, calcular las coordenadas del mundo del punto de intersección.

(ver la siguiente transparencia).

Problema: intersección rayo-triángulo (2/2)

Problema 4.2. (continuación)

Ten en cuenta que habrá intersección si y solo si se cumplen cada una de estas dos condiciones:

1. El rayo interseca con el plano que contiene al triángulo, es decir, existe $t > 0$ tal que el punto $\mathbf{p}_t \equiv \mathbf{o} + t\mathbf{d}$ está en dicho plano. Equivale a decir que el vector $\mathbf{p}_t - \mathbf{v}_0$ es perpendicular a la normal al plano.
2. El punto $\mathbf{p}_t$ citado arriba está dentro del triángulo. Es decir, hay dos valores reales no negativos a y b (con $0 \leq a + b \leq 1$) tales que el vector $\mathbf{p}_t - \mathbf{v}_0$ es igual a $a(\mathbf{v}_1 - \mathbf{v}_0) + b(\mathbf{v}_2 - \mathbf{v}_0)$.

(a los tres valores a, b y $c \equiv 1 - b - a$ se les llama *coordenadas baricéntricas* de $\mathbf{p}_t$ en el triángulo, se usan en ray-tracing).

Problema: calculo de rayos para selección

Problema 4.3.

Para implementar la selección usando intersecciones es necesario calcular el rayo que tiene como origen el observador y pasa por centro del pixel donde se ha hecho click. Escribe el pseudo-código del algoritmo que calcula el rayo a partir de las coordenadas del pixel donde se ha hecho click:

- Tenemos una vista perspectiva, y conocemos los 6 valores l, r, t, b, n, f usados para construir la matriz de proyección.
- También conocemos el marco de coordenadas de vista, es decir, las tuplas $\mathbf{x}_{\mathrm{ec}}$,$\mathbf{y}_{\mathrm{ec}}$ y $\mathbf{o}_{\mathrm{ec}}$ con los versores y la tupla $\dot{o}_{\mathrm{ec}}$ con el punto origen (todos en coordenadas del mundo).
- El viewport tiene w columnas y f filas de pixels. Se ha hecho click en el pixel de coordenadas enteras x_{p} e y_{p}

El algoritmo debe producir como salida las tuplas $\mathbf{o}$ y $\mathbf{d}$ (normalizado) que definen el rayo.

Subsección 5.2.
Selección con *frame-buffer object* invisible..

Visualización sobre un frame-buffer object invisible

Si no se quiere usar funcionalidad obsoleta, se puede utilizar directamente visualización sobre un frame-buffer distinto del que se está viendo en pantalla. Se puede hacer de dos forma:

- Creando un objeto OpenGL de tipo **frame-buffer object** (FBO), y haciendo rasterización con ese objeto como imagen de destino (*rendering target*).
- Usando el modo de **doble buffer**:
  - En este modo siempre existen dos FBOs creados por OpenGL: un *buffer trasero* (*back buffer*), que es donde se visualizan las primitivas, y un *buffer delantero* (*front buffer*), que es el que se visualiza en pantalla.

Usaremos la primera opción al no depender de la existencia de doble buffer.

Visualización identificadores

La visualización sobre el frame-buffer no visible requiere:

- **Codificación de identificadores**: se necesita codificar los identificadores como colores (R,G,B), y usar esos colores en lugar de los materiales de los objetos.
- **Cambio de colores:** durante la visualización es necesario cambiar el color actual de OpenGL (antes de cada objeto), usando esos colores, con total precisión numérica.
- **Modo de visualización:** es necesario desactivar iluminación y texturas, activar sombreado plano y visualizar con todas las primitivas (triángulos) rellenos.

Esto constituye un nuevo modo de visualización, lo llamaremos **modo de identificadores**.

Codificación y cambio de color

Debemos cambiar el color del cauce usando un identificador:

- Los identificadores son enteros sin signo (tipo **unsigned** de C/C++, de 4 bytes), con valores entre 0 y $2^{24}-1$ (el byte más significativo es 0, se usan los tres menos significativos).
- Cada byte del identificador (entre 0 y 255) se convierte a un **float** (en $[0..1]$) que se usa para cambiar el color actual del cauce.

```
void FijarColorIdent( Cauce & cauce, const unsigned ident ) // 0 ≤ ident < 2^24
{
  const unsigned char
   byteR = ( ident             ) % 0x100U, // rojo = byte menos significativo
   byteG = ( ident /   0x100U  ) % 0x100U, // verde = byte intermedio
   byteB = ( ident / 0x10000U  ) % 0x100U; // azul = byte más significativo

  cauce.fijarColor( float(byteR)/255.0f, float(byteG)/255.0f,
                    float(byteB)/255.0f);
}
```

Lectura de colores

Para leer los colores se puede usar la función **glReadPixels**, que lee los colores de un bloque de pixels en el framebuffer activo para escritura.

- Leeremos un bloque con un único pixel.
- Se leen tres valores **unsigned char** en orden R,G,B.
- Se reconstruye el identificador **unsigned**, conocidos los tres bytes.

Lo podemos hacer así:

```
unsigned LeerIdentEnPixel( int xpix, int ypix )
{
  unsigned char bytes[3] ;  // para guardar los tres bytes
  // leer los 3 bytes del frame-buffer
  glReadPixels( xpix,ypix, 1,1, GL_RGB,GL_UNSIGNED_BYTE, (void *)bytes);
  // reconstruir el indentificador y devolverlo:
  return bytes[0] + ( 0x100U*bytes[1] ) + ( 0x10000U*bytes[2] ) ;
}
```

Frame-buffer objects (FBOs)

OpenGL permite crear y gestionat FBOs alojados en la memoria de la GPU

- Cada FBO tiene asociado un identificador entero único, no negativo.
- El FBO incial (que se visualiza en la ventana de la aplicación) tiene asociado el identificador 0 y está creado al inicio
- Es posible crear un FBO, indicando su tamaño.
- Un FBO puede tener asociado
  - Un array de colores de pixels (*color buffer*): es una textura de OpenGL alojada en la GPU.
  - Un array de profundidades (*render buffer*): contiene la profundidad del objeto proyectado en cada pixel (es el Z-buffer usado para EPO)

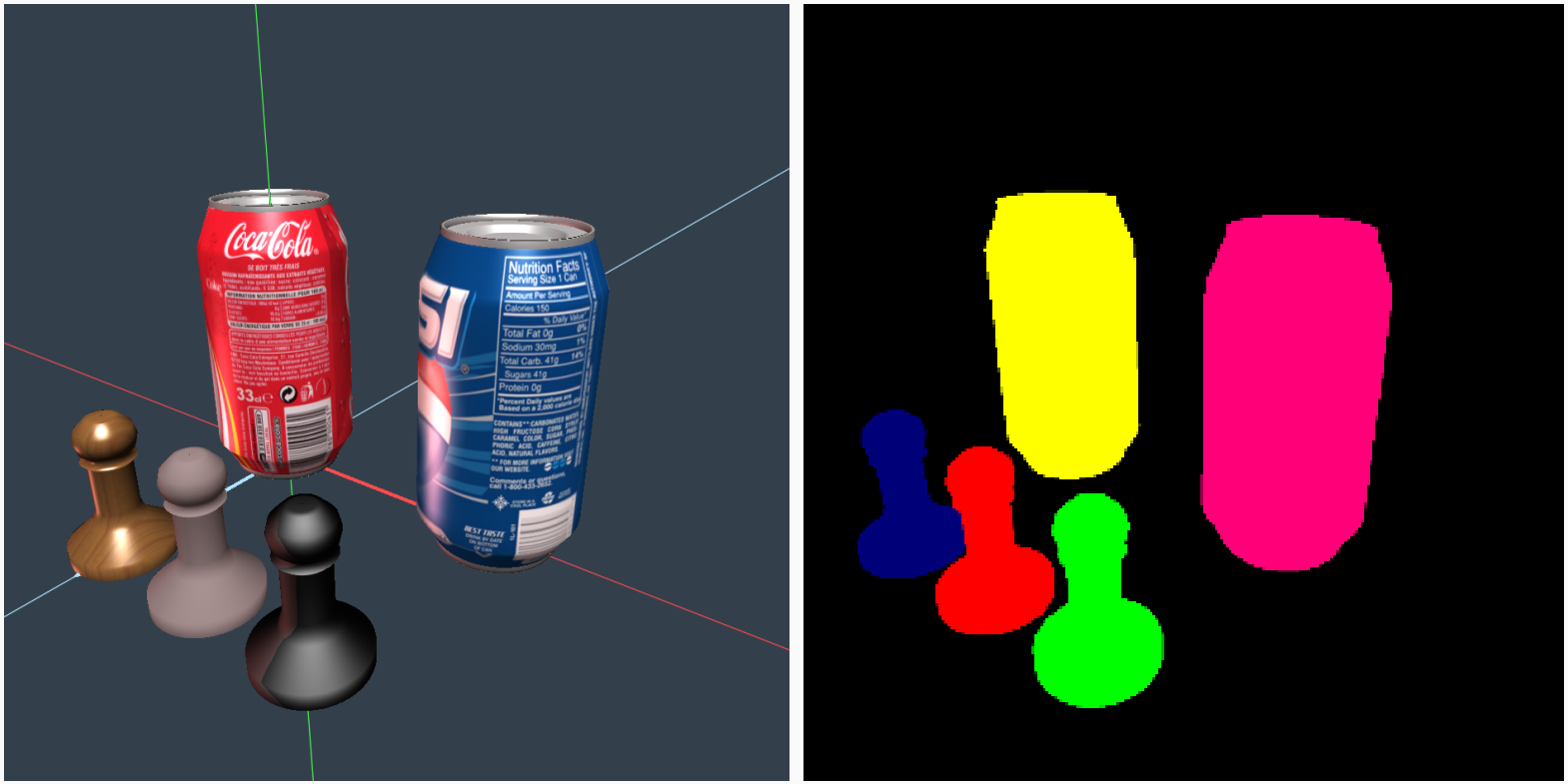
La clase `Framebuffer`

Encapsula un identificador de FBO, y su ancho y alto. Permite activarlo, redimensionarlo y consultar un pixel (**`leerPixel`**). Al activarlo, se rasteriza sobre este FBO.

```
class Framebuffer
{
public:
   // crear un FBO de este tamaño
   Framebuffer( const int pancho, const int palto );
   // activar, recreando el FBO si hay cambio de tamaño
   void activar( const int pancho, const int palto );
   void desactivar(); // desactivar (activa el 0)
   ~Framebuffer(); // llama a 'destruir'
private:
   void inicializar( const int pancho, const int palto );
   void destruir(); // libera memoria en la GPU
   GLuint fboId  = 0,  // identificador del framebuffer (0 si no creado)
          textId = 0,  // identificador de la textura de color
          rbId   = 0;  // identificador del z-buffer
   int    ancho  = 0,  // ancho del framebuffer, en pixels
          alto   = 0;  // alto del framebuffer, en pixels
};
```

Escena y FBO con identificadores

Se visualiza una escena en modo normal (izquierda) y en modo selección (derecha), con la misma cámara. Se han seleccionado los identificadores para que se correspondan con colores RGB distinguibles. Los ejes no son seleccionables.



Sección 6.
Animación.

Animación

Generar animaciones implica:

- Regenerar la imagen periodicamente (al menos 30 veces por segundo)
- Modificar la pose de los objetos de la escena

Keyframe

El animador define dos configuraciones del modelo.

El sistema calcula los fotogramas intermedios (in-betweens) por interpolación.

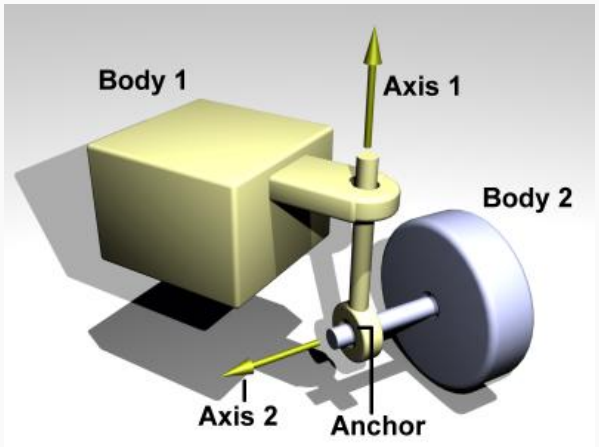
(c) Walt Disney Company, from "The Illusion of Life"

Simulación física

Para escenas con modelos físicos simples se puede calcular la configuración del escenario en cada fotograma usando las leyes de la mecánica clásica.

Existen librerías específicas para realizar esta simulación:

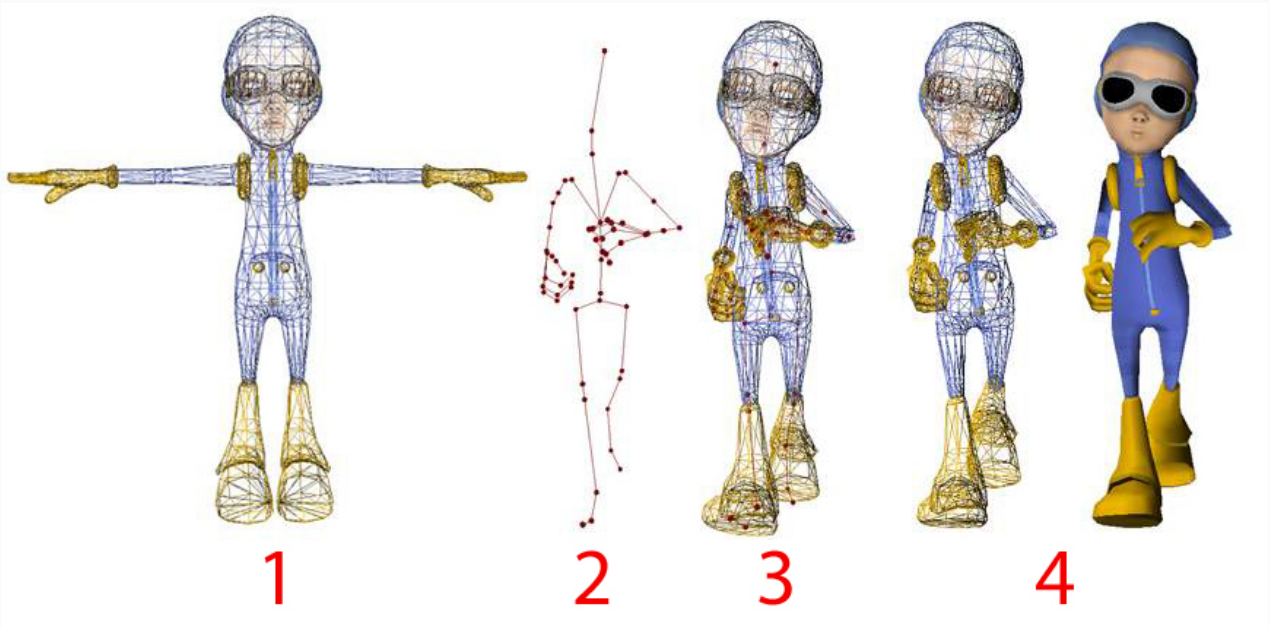
- ODE: Open Dynamic Engine.
- Newton: Newton Physics Engine
- Bullet: Physics Library



Esqueletos

Para conseguir que la animación de personajes resulte plausible se suelen usar esqueletos.

Un esqueleto es un modelo simplificado del personaje, formado por segmentos rígidos unidos por articulaciones.



Animación procedural

El comportamiento del objeto se describe mediante un procedimiento.

Es útil cuando el comportamiento es fácil de generar pero dificil de simular físicamenete (p.e. la rotura de un vidrio).

Fin de la presentación.