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

Carlos Ureña

Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

2018-19

Teoría. Tema 4. Interacción. Animación.
Índice.

1 Introducción
2 Eventos en GLFW
3 Posicionamiento
4 Control de cámaras
5 Selección
6 Animación

Sección 1
Introducción

Sistemas interactivos

Un sistema gráfico interactivo es un sistema que responde de forma directa a las acciones del usuario.

El sistema funciona según el siguiente ciclo:

- El usuario realiza una acción
- El programa recibe información
- Procesa la acción
- Redibuja el modelo

Interactividad

La incoporación de interactividad permite realizar aplicaciones que respondan a las acciones de los usuarios.

La mayor parte de los sistemas gráficos son interactivos.

La interactividad es esencial en

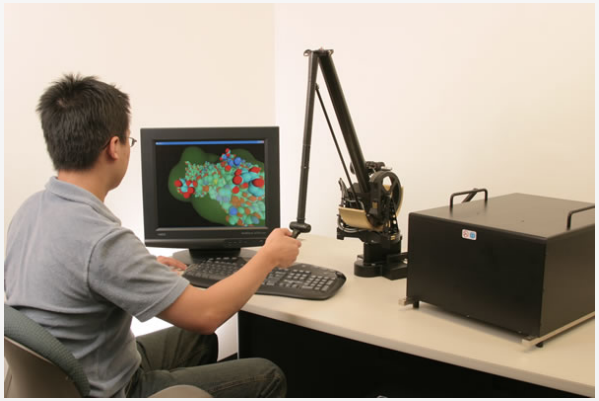
- sistemas de diseño
- juegos
- modeladores
- simuladores

Interactividad

En un sistema gráfico interactivo el usuario debe disponer de al menos un dispositivo de entrada y un dispositivo de visualización. Como mínimo téclado y monitor. Normalmente se dispone además de un ratón.

En sistemas profesionales se suelen utilizar además dispositivos de entrada especiales como:

- Tableta digitalizadora
- Sistemas de posicionamiento
- Palpadores



Interactividad

La interactividad requiere que el retardo (latencia) entre la acción del usuario y la respuesta del sistema sea suficientemente pequeño para que el usuario perciba una relacción de causa efecto.

Las restricciones de un sistema de tiempo real son mayores, el retardo debe ser menor que un tiempo prefijado.

Realimentación

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

Técnicas

Con frecuencia, el usuario debe introducir una posición que satisfaga una determinada condición, por ejemplo tener la misma coordenada X que la posición previa.

En estas situaciones, la aplicación puede ayudar al usuario a realizar su tarea proporcionándole técnicas especificas que simplifiquen el proceso de interacción, usando técnicas de interacción.

Por ejemplo puede hacer que el cursor solo se mueva en horizontal.

Técnicas:

- Líneas elásticas
- Guías
- Gravedad

Funciones

Un sistema gráfico interactivo necesita normalmente funciones de entrada para leer texto y valores númericos como cualquier sistema gráfico, pero necesita además poder

- leer posiciones
- seleccionar componentes del modelo geométrico

La lectura de posiciones (posicionamiento) permite generar posiciones en coordenadas del mundo.

La selección permite identificar interactivamente elementos del modelo geométrico.

Dispositivos

El usuario introduce la información por medio de dispositivos de entrada. Estos pueden ser de propósito general (teclado), o específicos para la generación de datos geométricos (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**: palpador, digitalizador, tracking
- **desplazamientos 2D**: ratón, trackball, joystick

Dispositivos lógicos

La dependencia de la aplicación con el hardware de entrada se resuelve, en parte, trabajando con dispositivos abstractos, denominados **dispositivos lógicos**.

Cada dispositivo lógico representa un dispositivo físico ideal, que genera un único tipo de información de entrada. Cada dispositivo físico puede ser tratado como una realización de uno, o varios, dispositivos lógicos.

Los sistemas gráficos necesitan dos tipos de dispsitivos específicos:

- **Locator**: lee posiciones
- **Pick**: lee identidicadores de componentes del modelo geométrico

Procesamiento

Otro de los problemas relacionados con las operaciones de entrada es el de sincronizar las acciones del usuario con la aplicación.

La sincronización debe realizarse de forma distinta dependiendo de la función a realizar.

La sincronización se resuelve en los sistemas gráficos, haciendo uso de mecanismos de abstracción.

A las abstracciones del comportamiento se les denomina **modos de entrada**.

En cada momento, laaplicación puede asociar un modo a cada dispositivo lógico.

Hay tres modos de funcionamiento: **muestreo, petición y cola de eventos**.

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 asociado:
  - 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_BotonRaton )
```

El callback debe estar declarado con estos parámetros y tipo devuelto:

```
void FGE_BotonRaton( 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_BotonRaton( 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 )
         RatonClickIzq( int(x), int(y) );
      else if ( button == GLFW_MOUSE_BUTTON_RIGHT )
      {  xclick_der = int(x) ;
         yclick_der = int(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_RatonMovido )
```

El callback debe estar declarado con tres parámetros enteros:

```
void FGE_RatonMovido( 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*, podemo, 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_RatonMovido( 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.

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 )
   {  VisualizarFrame();     // visualizar la ventana si es necesario
      redibujar = false ;    // no volver a visualizar si no es necesario
   }
   glfwWaitEvents();         // procesar eventos pendientes o esperar uno y procesarlo.
   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`** true si el usuario ha realizado alguna acción de cierre de ventana en el gestor de ventanas.

El bucle de gestión de eventos con animaciones

Para animaciones, se ejecuta una **función desocupado** cuando no hay eventos:

```
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 )
   {  VisualizarFrame();     // visualizar la ventana si es necesario
      redibujar = false ;    // no volver a visualizar si no es necesario
   }
   glfwPollEvents();         // procesar eventos pendientes
   terminar = terminar || glfwWindowShouldClose( glfw_window ) ;
   if ( ! terminar && ! redibujar )
      FuncionDesocupado();
}
```

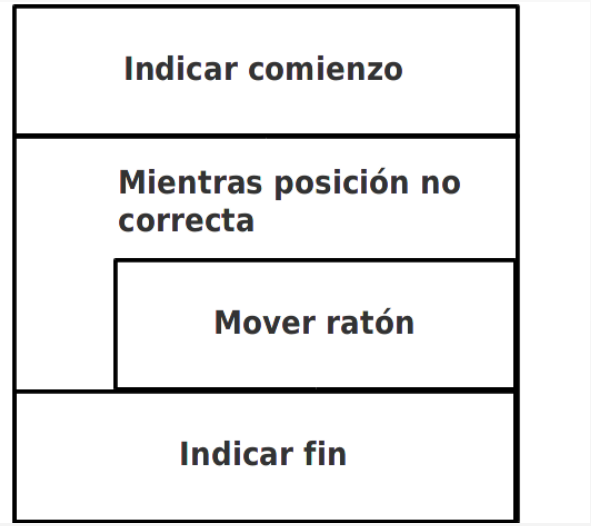
- **`glfwPollEvents`** si hay eventos pendientes, los procesa, en otro caso no hace nada.
- **`FuncionDesocupado`**: función de la aplicación que se encarga de ejecutar un código cuando no hay eventos pendientes ni hay que redibujar la ventana.

Sección 3
Posicionamiento

Posicionamiento

El objetivo de las operaciones de posicionamiento es permitir la introducción de un punto.

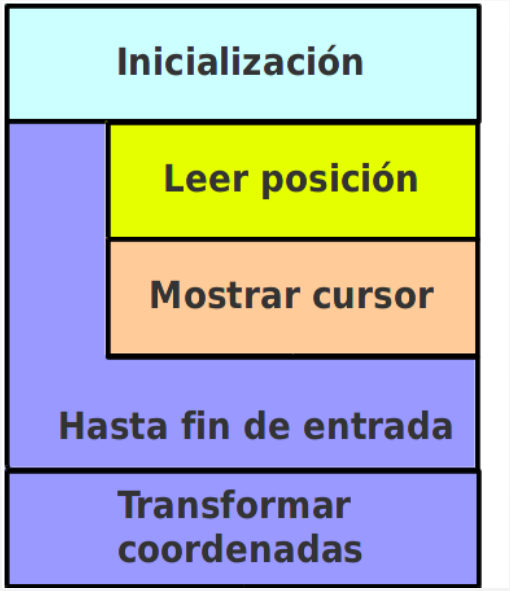
Desde el punto de vista del usuario, que utiliza un dispositivo que controla un cursor, el proceso de entrada es el que se muestra en la figura



Posicionamiento

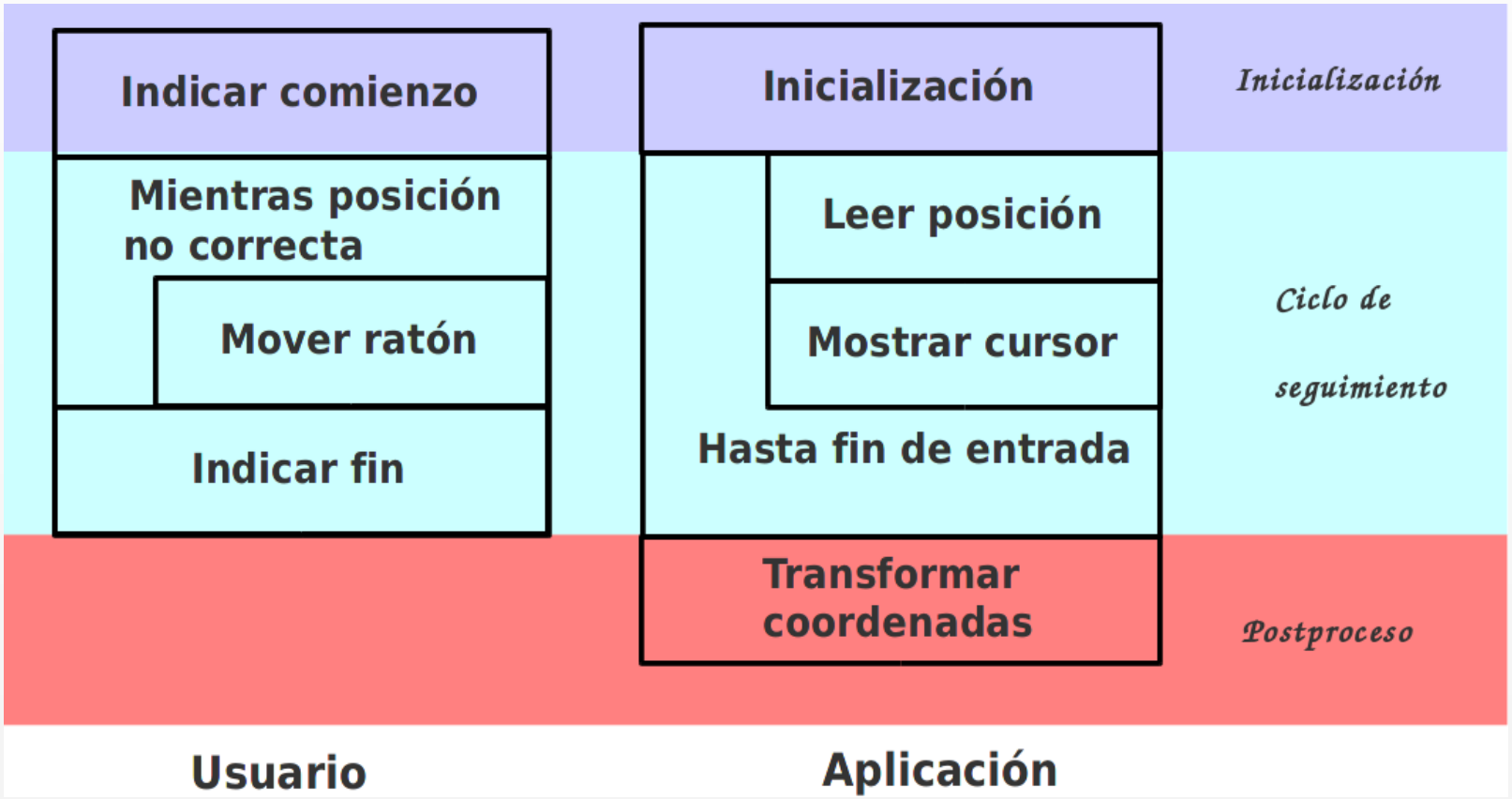
El proceso en el software de tratamiento se realiza en una estructura cíclica, que se corresponde con el ciclo de búsqueda que realiza el usuario. En este proceso podemos distinguir tres etapas:

- **inicialización**
- **ciclo de seguimiento**
- **postproceso**



Posicionamiento

Correspondencia entre operaciones del usuario y del sistema

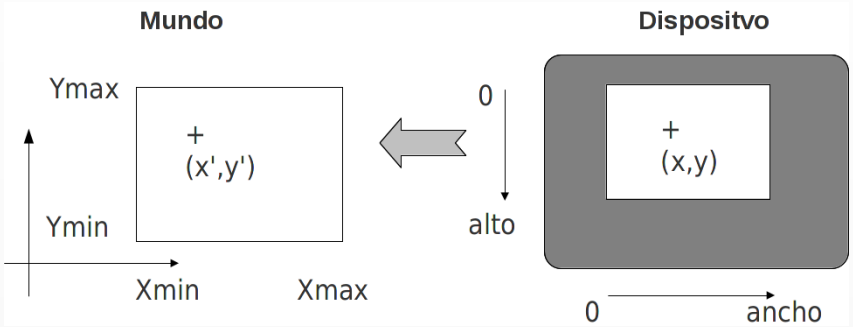


Conversión a coordenadas del mundo

La posición obtenida en la lectura está dada en coordenadas de dispositivo. Es necesario pasarla a coordenadas del mundo. Para ello se puede:

- Invertir la transformacin de visualización (si el modelo es 2D)
- Restringir el posicionamiento a un plano (calculando la posición sobre el plano)
- Utilizar un cursor 3D
- Utilizar vistas con tres proyecciones paralelas
- Invertir la transformacin de visualización (si el dispositivo de entrada es 3D)

Posicionamiento 2D



$$x' = X_{min} + x * (X_{max} - X_{min}) / ancho \quad (1)$$

$$y' = Y_{max} - y * (Y_{max} - Y_{min}) / alto \quad (2)$$

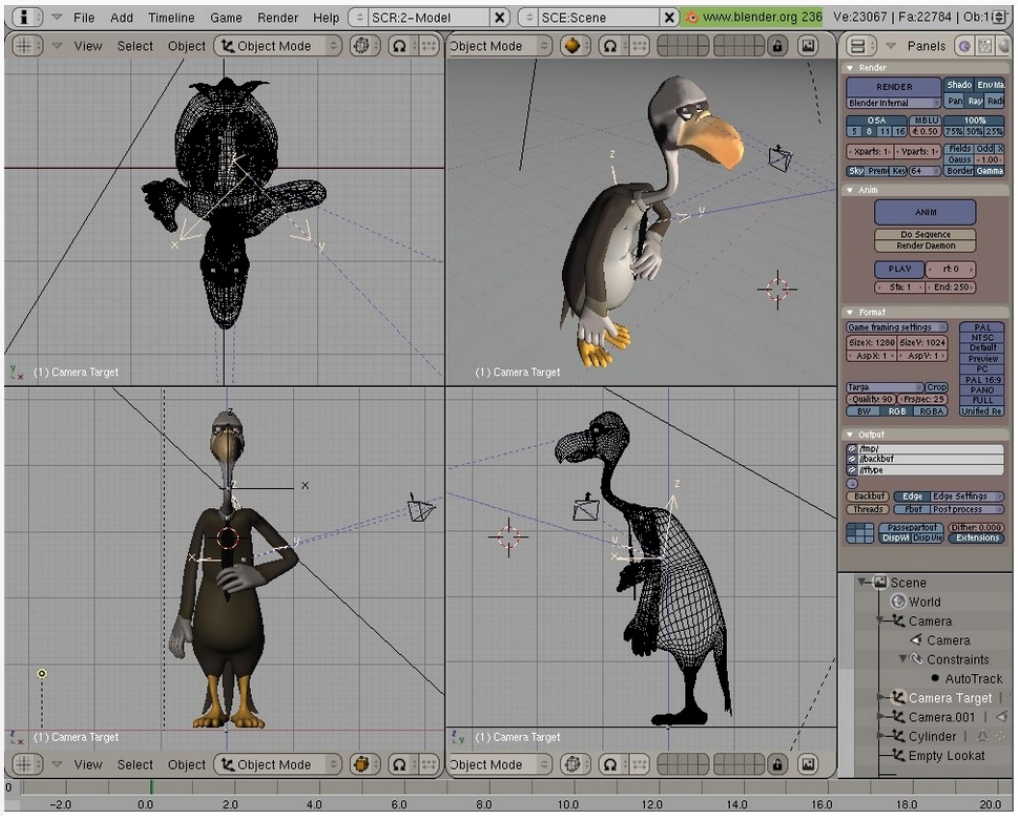
Los parámetros son los indicados a las funciones **glOrtho** y **glViewport**:

```
glOrtho( Xmin, Xmax, Ymin, Ymax, Zmin, Zmax );
glViewport( x0, y0, ancho, alto );
```

Posicionamiento 3D

Vistas con tres proyecciones paralelas

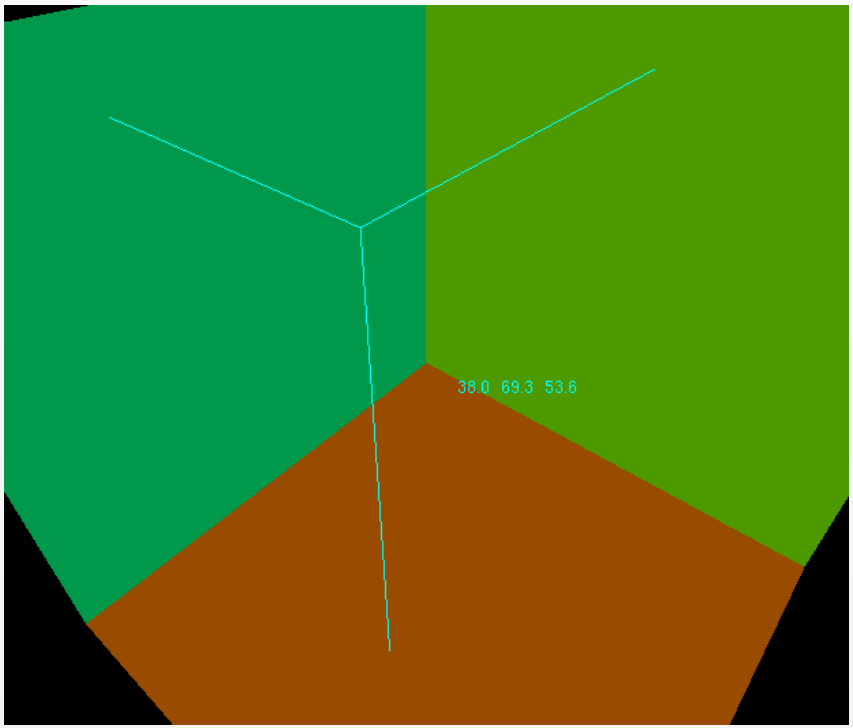
En cada vista se fijan dos coordenadas



Posicionamiento 3D: Cursor 3D

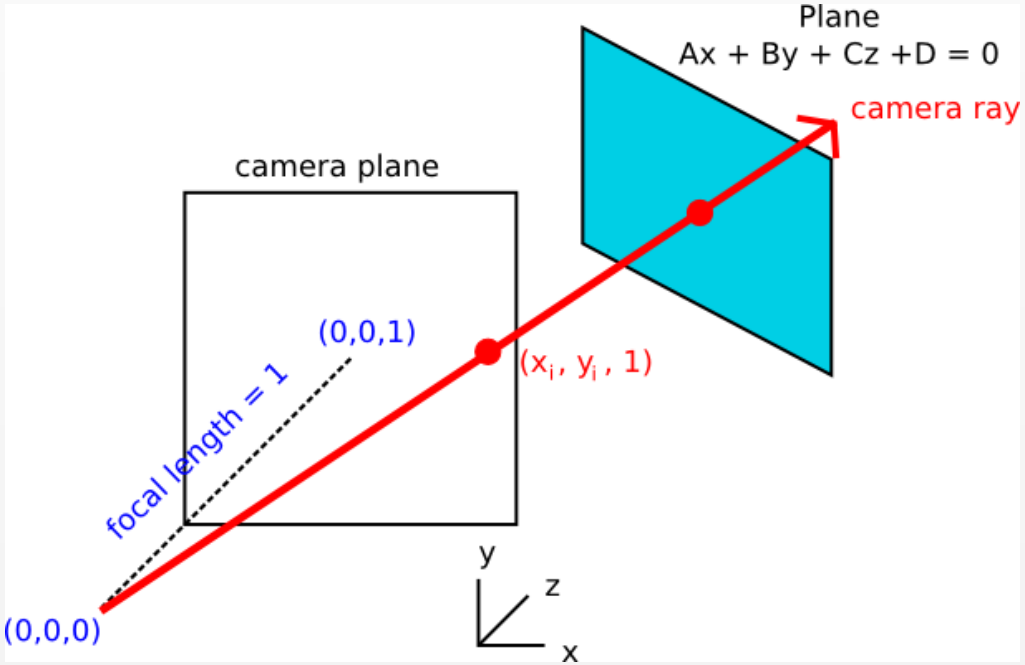
La realimentación es un cursor 3D

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

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. Cámaras en modo primera persona.
4.2. Cámara orbital.

Control interactivo de la cámara

En aplicaciones interactivas la cámara virtual muchas veces puede ser cambiada por el usuario (con retroalimentación inmediata), para obtener la vista que desee de la escena. Esto se puede hacer de varias formas:

- Para **visualización de objetos**: se usa cámara en modo **orbital** (o **examinar**): El observador se situa fuera del objeto, y lo puede observar desde distintas direcciones o a distintas distancias del mismo. El objeto se mantiene centrado en pantalla.
- Para **exploración de escenarios**: se usa cámara en modo **primera persona**: el usuario/observador modifica interactivamente su
  - **posición**: se puede desplazar VRP hacia arriba, abajo, izquierda, derecha, adelante o detrás (en direcciones relativas al marco de coordenadas de la vista, es decir, relativas a la cámara virtual).
  - **orientación**: se puede rotar la dirección de la vista (VPN y VUP) entorno a los tres ejes del marco de coordenadas de la cámara virtual.

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

Representación de la cámara y cálculo de V.

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

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

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 $\mathcal{C}$, es decir:

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

Subsección 4.1
Cámaras en modo primera persona.

Cámara primera persona: operaciones

Con la cámara en modo **primera persona**, el usuario modifica o el marco de vista $\mathcal{C} = [\mathbf{x}_c, \mathbf{y}_c, \mathbf{z}_c, \mathbf{o}_c]$

- **Rotaciones:** se rotan los vectores $\vec{x}_c$,$\vec{y}_c$ y $\vec{z}_c$ entorno a $\dot{o}_c$, pero no se modifica $\dot{o}_c$, de tres tipos:
  - (a.1) Rotar $\vec{x}_c$ y $\vec{z}_c$ entorno a $\vec{y}_c$ (hacia derecha o izquierda)
  - (a.2) Rotar $\vec{z}_c$ y $\vec{y}_c$ entorno a $\vec{x}_c$ (hacia arriba o hacia abajo)
  - (a.3) Rotar $\vec{x}_c$ y $\vec{y}_c$ entorno a $\vec{z}_c$ (la imagen rota entorno a su centro)
- **Traslaciones:** no se modifica la orientación del marco de vista, únicamente se desplaza el origen $\dot{o}_c$, de tres formas:
  - (b.1) Hacia derecha o hacia izquierda (en la dirección de $\pm\vec{x}_c$).
  - (b.2) Hacia arriba o hacia abajo (en la dirección de $\pm\vec{y}_c$).
  - (b.3) Hacia adelante o hacia atrás (en la dirección de $\pm\vec{z}_c$).

Implementación de operaciones.

Se necesita modificar las tuplas que representan el marco $(\vec{x}_c, \vec{y}_c, \vec{z}_c, \vec{o}_c)$:

- **Rotaciones:** se aplica una matriz de rotación R (de eje arbitrario), por un ángulo θ, a los tres vectores:

$$\mathbf{x}_c := R\,\mathbf{x}_c \qquad \mathbf{y}_c := R\,\mathbf{y}_c \qquad \mathbf{z}_c := R\,\mathbf{z}_c$$

(a.1) rotar en horizontal:	$R \equiv \mathrm{R}[\theta, \mathbf{y}_c]$
(a.2) rotar en vertical:	$R \equiv \mathrm{R}[\theta, \mathbf{x}_c]$
(a.3) rotación de la imagen	$R \equiv \mathrm{R}[\theta, \mathbf{z}_c]$

- **Traslaciones:** se aplica una traslación T a $\mathbf{o}_c$, de una distancia d:

$$\mathbf{o}_c := T\,\mathbf{o}_c$$

(b.1) mover $\mathbf{o}_c$ a la izquierda o la derecha:	$T \equiv \mathrm{D}[d\vec{x}_c]$
(b.2) mover $\mathbf{o}_c$ hacia arriba o abajo:	$T \equiv \mathrm{D}[d\vec{y}_c]$
(b.3) mover $\mathbf{o}_c$ hacia adelante o atrás	$T \equiv \mathrm{D}[d\vec{z}_c]$

Subsección 4.2
Cámara orbital.

Cámara en modo orbital o examinar.

En este modo (modo **orbital**, o modo **examinar**), el usuario manipula la cámara virtual, pero siempre se mantiene un determinado **punto de atención** proyectado en el centro de la imagen:

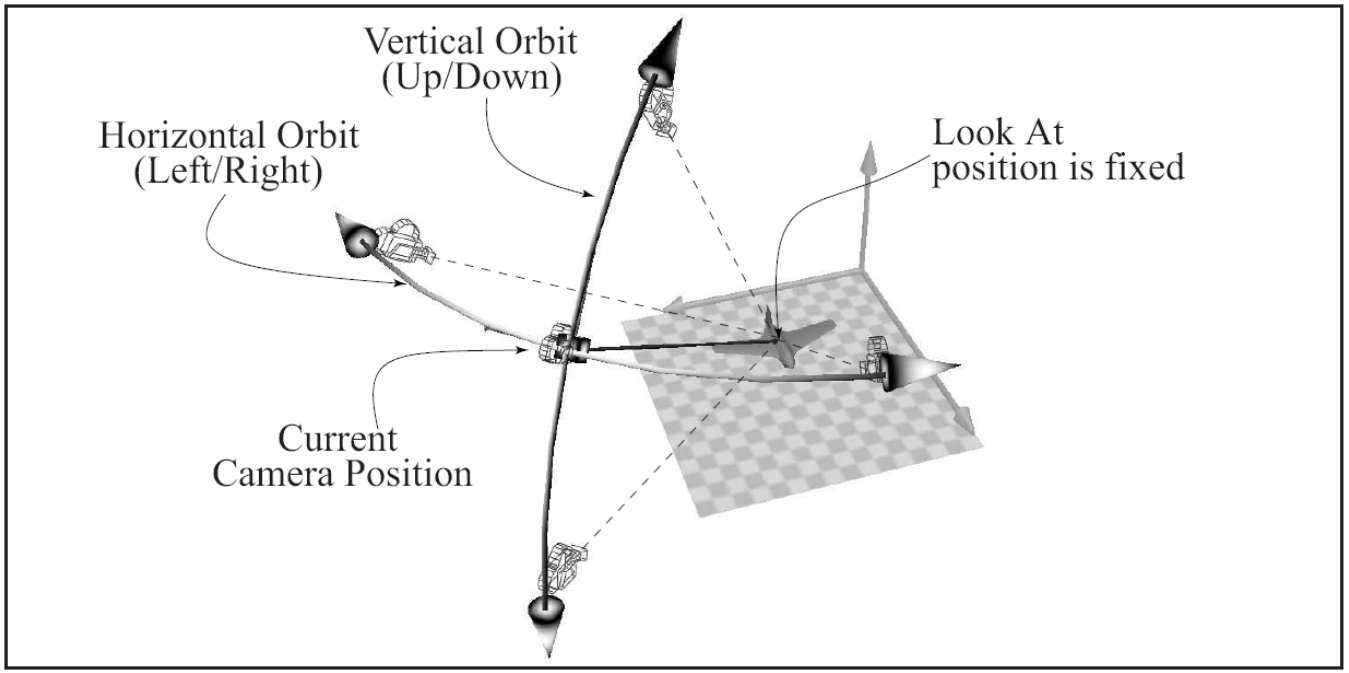


Figura obtenida de: *K.Sung, P.Shirley, S.Baer* **Essentials of Interactive Computer Graphics: Concepts and Implementation.**

Representación de la cámara orbital

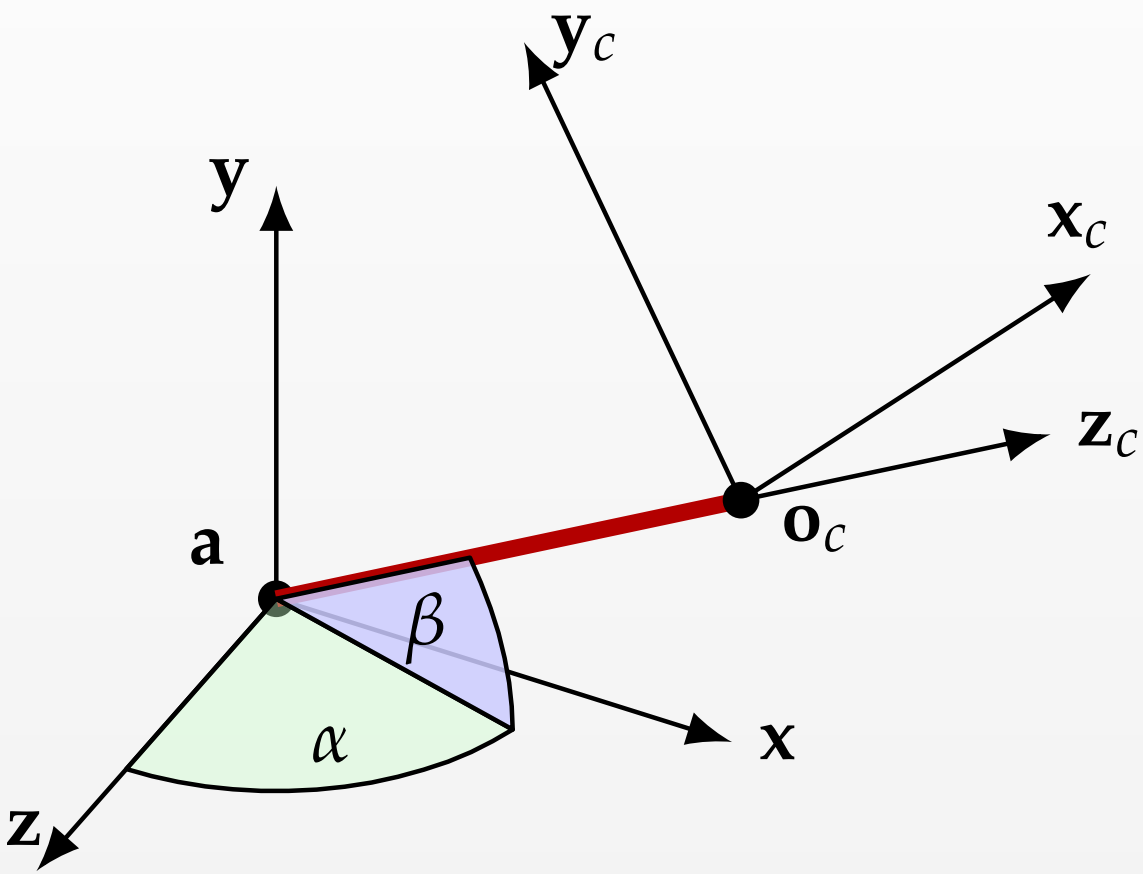
La posición y orientación del marco de coordenadas de la cámara viene determinada por las coordenadas esféricas (α, β, r) del origen del marco respecto del punto de atención, y por el propio punto de atención.

- α es un ángulo que representa la *longitud* (rotación entorno al eje Y).
- β es un ángulo que representa la *latitud* (rotación entorno al eje X)
- r es el radio ($r > 0$), es decir, la distancia entre el punto de atención y el origen del marco de la cámara (la posición del observador).
- $\mathbf{a} = (l_x, l_y, l_z, 0)^t$ es un vector con las coordenadas de mundo de $\vec{a}$, el punto de atención o *look-at* .

Para representar en memoria una cámara orbital, podemos almacenar el marco de vista (como cualquier cámara), pero si se quiere manipular interactivamente, debemos, adicionalmente, de guardar α,β,r y $\mathbf{a}$

Elementos de la cámara orbital

En esta figura se observan el marco de la cámara orbital, junto con el marco del mundo (trasladado su origen al punto de atención).



Operaciones sobre cámaras orbitales

Las posibles modificaciones que un usuario puede hacer sobre una cámara en modo orbital son las siguientes:

- Rotación en **horizontal** (cambiar la longitud), se hace: $\alpha := \alpha + \Delta$. El valor Δ es el ángulo de rotación (> 0 para la derecha, < 0 a la izquierda).
- Rotación en **vertical** (cambiar la latitud), se hace: $\beta := \beta + \Delta$. El valor Δ es > 0 para rotar la cámara hacia abajo, y es < 0 para rotarla hacia arriba.
- **Acercamiento o alejamiento** al punto de atención (cambiar la distancia r), se hace:

$$r := d_{min} + (1 - f)(r - d_{min})$$

El valor real f, con $\|f\| < 1$ es el ratio de cambio del exceso de distancia respecto de una distancia mínima d_{min} (siempre $d_{min} \leq r$). Si $f < 0$, supone un alejamiento (r crece), si $f > 0$ entonces supone acercamiento (r disminuye).

Cálculo del marco de vista

Queremos escribir las coordenadas del marco de vista $\mathbf{x}_c, \mathbf{y}_c, \mathbf{z}_c$ y $\mathbf{o}_c$ a partir de α, β, r y $\mathbf{a} = (l_x, l_y, l_z, 1)^t$.

Esto se puede lograr aplicando una matriz A a las tuplas de coordenadas del marco de coordenadas del mundo. La matriz es la composición de estas matrices (todas en coordenadas de mundo):

1. Traslación a lo largo del eje Z, una distancia r.
2. Rotación entorno al eje X, un ángulo $-\beta$.
3. Rotación entorno al eje Y, un ángulo α.
4. Traslación por un vector igual al punto de atención $(l_x, l_y, l_z, 0)$.

Por tanto, podemos calcular la matriz A haciendo composición de traslaciones y rotaciones:

$$A = \mathrm{D}[l_x, l_y, l_z] \cdot \mathrm{R_y}[\alpha] \cdot \mathrm{R_x}[-\beta] \cdot \mathrm{D}[0, 0, r]$$

Coordenadas de $\mathcal{C}$ y matriz de vista

Debemos de obtener $\mathbf{x}_c, \mathbf{y}_c, \mathbf{z}_c$ y $\mathbf{o}_c$ aplicando la matriz A a las coordenadas de los componentes del marco de coordenadas del mundo:

$$\begin{aligned} \mathbf{x}_c &= A(1,0,0,0)^t \\ \mathbf{y}_c &= A(0,1,0,0)^t \\ \mathbf{z}_c &= A(0,0,1,0)^t \\ \mathbf{o}_c &= A(0,0,0,1)^t \end{aligned}$$

La matriz de vista V puede obtenerse directamente a partir de las tuplas, o bien, si se desea escribir en términos de matrices de transformación elementales, se puede tener en cuenta que $V = A^{-1}$, es decir, haríamos:

$$V = \mathrm{D}[0,0,-r] \cdot \mathrm{R_x}[\beta] \cdot \mathrm{R_y}[-\alpha] \cdot \mathrm{D}\left[-l_x,-l_y,-l_z\right]$$

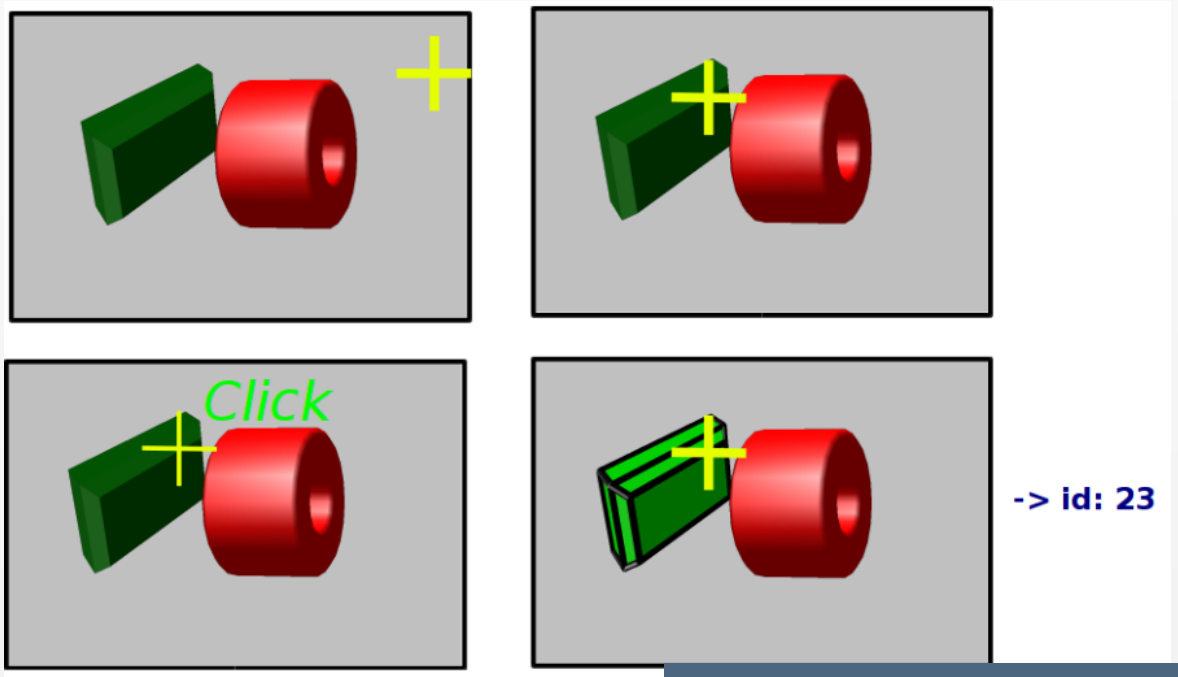
Sección 5
Selección

5.1. Modo de *selección* de OpenGL
5.2. Selección con *frame-buffer* invisible.

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

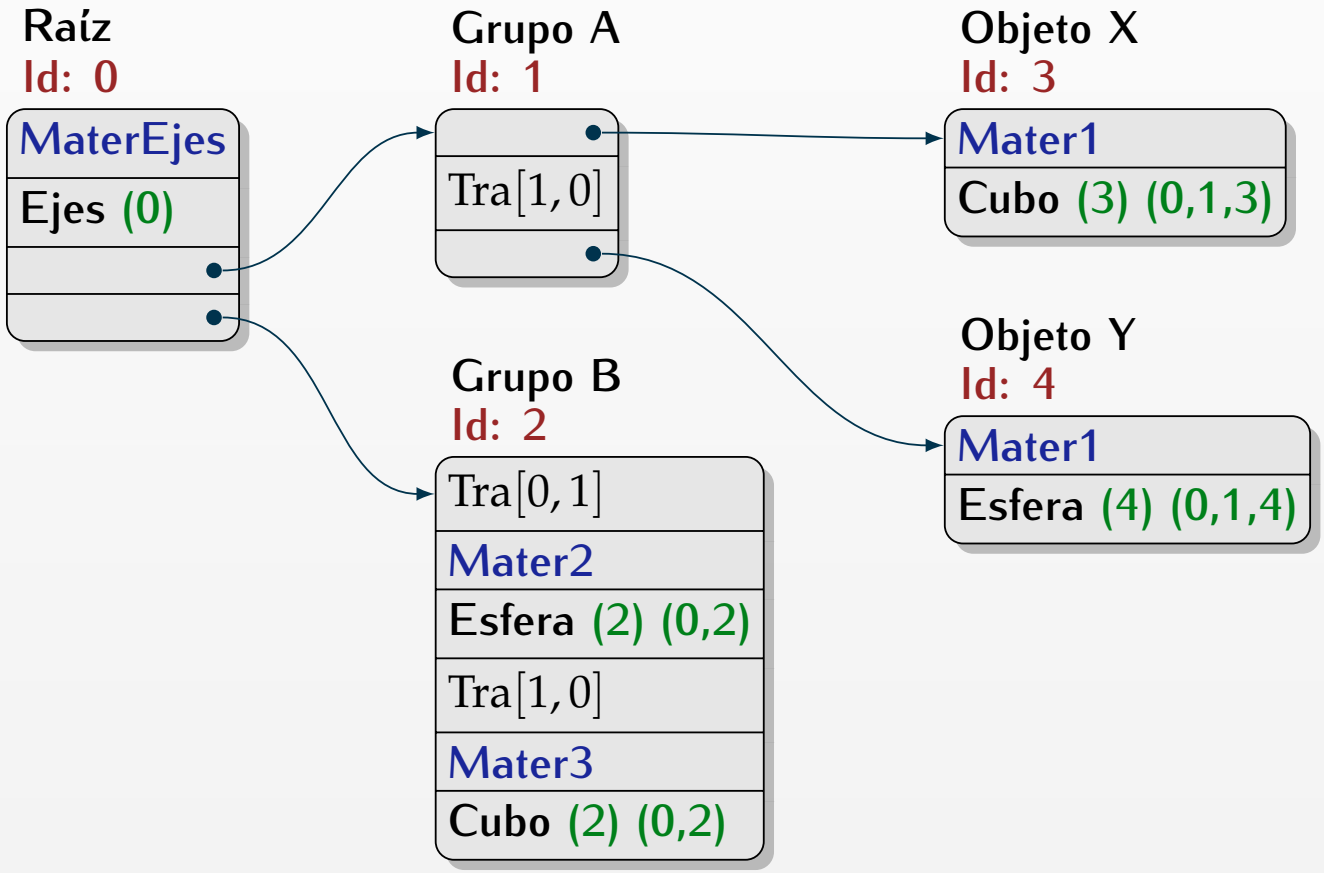
Para poder realizar la selección los componentes de la escena deben tener asociados identificadores numéricos (enteros), a distintos niveles:

- **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

Cada nodo del grafo guarda un identificador entero (no negativo). Cada primitiva u objeto terminal tendrá siempre asociado un identificador (o bien una lista de identificadores), en la figura se han anotado en verde:



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

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

- **Modo selección de OpenGL:** se usa una funcionalidad de OpenGL, específica para este fin:
  - 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.

  (funcionalidad obsoleta desde OpenGL 3.0).
- **Frame-buffer no visible**. Se usa algún **frame buffer object** (array de colores de pixels en la memoria de la GPU) distinto del que se está visualizando en la ventana:
  - Se debe desactivar iluminación y texturas, y visualizar con colores planos (colores obtenidos de los identificadores de los objetos).
  - Al final se leen los colores (identificadores de objetos proyectados) en el pixel y alrededores.

Subsección 5.1
Modo de *selección* de OpenGL

Modo selección de OpenGL

OpenGL contempla un modo de visualización llamado **modo de selección**.

- Se puede activar con `glRenderMode( GL_SELECT )` (se debe usar la constante `GL_RENDER` para volver al modo habitual).
- Con este modo activo, al enviar primitivas no se visualizan en pantalla.
- OpenGL mantiene una variable interna con una lista de enteros, llamada **pila de nombre**, manipulable por la aplicación.
- Cuando se envia una primitiva desde la aplicación, se le asocia la lista de enteros en la pila de nombre actual de OpenGL. A la lista se le llama **nombre** de la primitiva.
- OpenGL registra cada nombre disinto de cada primitiva que se visualiza en cualquier pixel. Para cada nombre se almacenenan en el **buffer de selección**:
  - Lista de enteros que forman el nombre.
  - Rango de profundidades en Z para todas las apariciones de ese nombre.

Manipulación de la pila del nombre

La aplicación debe asegurarse que la pila (LIFO) del nombre contiene los identificadores adecuados antes de visualizar cada objeto, para ello se pueden usar estas operaciones sobre dicha pila:

- **`glLoadName`**`( i )`
  sustituye entero en el tope de la pila por `i`
- **`glPushName`**`( i )`
  apila el entero `i`
- **`glPopName`**`()`
  desapila un entero (si la lista no está vacía)
- **`glInitNames`**`()`
  vacia la pila de nombres

en todos los casos, `i` es de tipo **`GLenum`** (normalmente equivalente a **`unsigned int`**)

Ejemplo de manipulación de la pila del nombre

Dibujo de un tablero de ajedrez, el nombre cada casilla es el par (número de fila, número de columna).

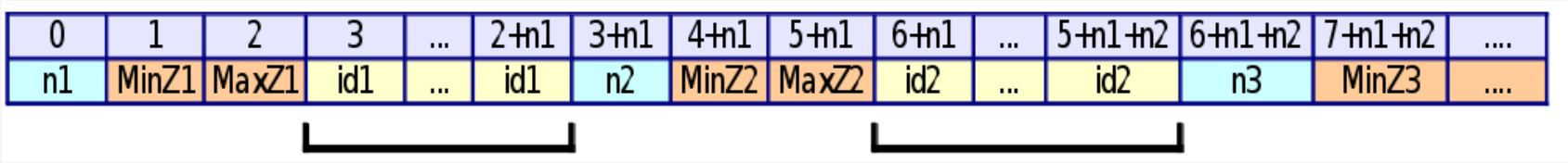
```
for( num_fila = 0 ; num_fila < 8 ; num_fila++ )
{
  glTranslatef( 1.0, 0.0, 0.0 );
  glPushMatrix();
  glPushName( num_fila );

  for( num_colu = 0 ; num_colu < 8 ; num_colu++ )
  {
     glTranslatef( 0.0, 1.0, 0.0 );
     glPushName( num_colu );
     DibujaCasilla(); // aquí, la pila contiene: ( num_fila, num_colu )
     glPopName() ;
  }
  glPopName();
  glPopMatrix();
}
```

Suponemos que **DibujaCasilla** dibuja un cuadrado de lado unidad en el plano XY ($z = 0$), con centro en $(0.5, 0.5)$.

Contenido del buffer de selección

El buffer de selección es un vector de entradas de tipo **`GLsizei`** (normalmente equivalente a **`int`**). Las entradas se dividen en bloques de tamaño variable, cada bloque corresponde a un nombre distinto aparecido el visualizar. En cada bloque se incluye esta información del nombre:

- El número de enteros que forman el nombre.
- El intervalo de profundidades del nombre: mínimo seguido de máximo (dos entradas).
- La lista de enteros que forman el nombre (uno por entrada).



Durante la rasterización, OpenGL registra nombres (y profundidades) en el buffer de selección, mientras haya memoria suficiente para ello.

Buffer de selección: implementación

Podemos usar una clase (singleton) para representar el buffer de selección, de esta forma:

```
class BufferSeleccion
{
  public:
  const GLsizei tamB = 1024 ;  // tamaño del buffer (== 4 Kb, p.ej.)
  int   vec[ tamB ];      // vector de entradas enteras (buffer)
  // variables calculadas en finModoSel:
  int   numNombres ;      // número de nombres (bloques) en el buffer
  int * nombreMin ;       // puntero al nombre más cercano (dentro de vec)
  int   longNombreMin ;   // longitud de nombreMin
  // métodos:
  BufferSeleccion() {}    // constructor por defecto (no hace nada)
  void inicioModoSel();   // fijar buffer en OpenGL, pasar al modo selección
  void finModoSel();      // analizar buffer, pasa a modo render
} ;
```

La constante **tamB** determina la cantidad de memoria disponible para el buffer, de forma que OpenGL nunca produce desbordamiento al escribir durante la rasterización en modo selección.

Procesamiento del buffer de selección

El número total de bloques se obtiene al llamar a **glRenderMode** estando en modo **GL_SELECT**. Se puede hacer al volver al modo *render*.

```
void BufferSeleccion::inicioModoSel()  // inicio del modo de selección
{
  glSelectBuffer( vec, tamB );   // decirle a OpenGL donde está el buffer de selección
  glRenderMode( GL_SELECT );     // entrar en modo selección
  glInitNames() ;                // vacía la pila de nombres
}
void BufferSeleccion::finModoSel( )   // fin del modo de selección
{
  int   profZmin;       // mínima profundidad encontrada hasta el momento
  int * bloque = vec;  // puntero a la base del bloque actual en el buffer
  numNombres = glRenderMode( GL_RENDER );// leer núm. de bloques, volver modo render
  for( i = 0 ; i < numNombres ; i++ )     // para cada nombre (bloque) en el buffer:
  { if ( i == 0 || bloque[1] < profZmin )   // si la profundidad es menor:
    {
      profZmin      = bloque[1] ;          // guardar nueva prof. mínima
      longNombreMin = bloque[0] ;          // registrar longitud del nombre
      nombreMin     = &(bloque[3]) ;       // registrar puntero a nombre mínimo.
    }
    bloque += bloque[0]+3 ;   // avanzar puntero a la base del bloque siguiente
  }
}
```

El view-frustum centrado en el pixl

Esta función fija un view-frustum de 5x5 pixels centrado en el pixel cuyas coordenadas se pasan como parámetro:

```
void FijarViewFrustumSel( int xpix, int ypix )
{
  GLint viewport[4];
  glGetIntegerv( GL_VIEWPORT, viewport );
  glMatrixMode( GL_PROJECTION );
  glLoadIdentity();
  gluPickMatrix( xpix, viewport[3]-ypix, 5.0, 5.0, viewport[0] );
  glFrustum( left, right, bottom, top, near, far ); // fijar view-frustum
}
```

- El efecto neto es fijar la matriz de proyección de OpenGL.
- Se deben usar los mismos parámetros del view-frustum usados para la visualización de la escena (**left**,**right**,**bottom**, etc....).
- Se hace uso de la librería GLU, para multiplicar por una matriz que restringe el tamaño del view-frustum

Gestión de eventos de click

Si queremos (por ejemplo) hacer selección con el botón izquierdo del ratón, podemos definir la función **Seleccion** que aparece abajo, e invocarla desde la función gestora de evento de botón de ratón:

```
BufferSeleccion buffer ;

void Seleccion( int xpix, int ypix )
{
  buffer.inicioModoSel();              // entrar en modo selección
  FijarViewFrustumSel( xpix, ypix ); // fijar view-frustum centrado en pixel
  VisualizarEscena();                  // visualizar la escena
  buffer.finModoSel() ;                // analizar buffer de selección, volver modo render
  // imprimir nombre seleccionado (lista de enteros)
  cout << "nombre seleccionado: "  ;
  for( i = 0 ; i < buffer.longNombreMin ; i++ )
    cout << " " << buffer.nombreMin[i] ;
}
void FGE_BotonRaton( GLFWwindow* window, int button, int action, int mod)
{
   if ( button == GLFW_MOUSE_BUTTON_RIGHT && action == GLFW_PRESS )
   {  double xf,yf ; glfwGetCursorPos( window, &xf, &yf );
      Seleccion( int(xf), int(yf) );
   }
}
```

Subsección 5.2
Selección con *frame-buffer* invisible.

Visualización sobre un frame-buffer no visible

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.
  - Se hace rendering sobre el buffer trasero (ocurre por defecto), y al final no se invoca a **`glfwSwapBuffers`**, pero se lee el identificador (color) del pixel.

Usaremos la segunda opción por su mayor simplicidad, al no requerir gestión de objetos de tipo FBO de OpenGL.

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

El cambio de color de OpenGL debe hacerse evitando errores numéricos que podrían darse si se usan ternas RGB en coma flotante.

- Para evitar errores de precisión, se usa una variante de **glColor** que acepta 3 datos de tipo **GLubyte** (≡ **unsigned char**).
- Los identificadores son enteros sin signo (tipo **unsigned** de C/C++, usualmente 4 bytes), con valores entre 0 y $2^{24}-1$ (el byte más significativo es 0, se usan los tres menos significativos).

Se puede hacer el cambio de color con una función como esta:

```
void FijarColorIdent( 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

  glColor3ub( byteR, byteG, byteB );            // cambio de color en OpenGL.
}
```

Lectura de colores

Para leer los colores se puede usar la función **glReadPixels**, que lee los colores de un bloque de pixels en el frame-buffer activo para escritura (en el modo de doble buffer, ese buffer siempre es el buffer trasero actual).

- 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.

La lectura y reconstrucción se puede hacer con una función como esta:

```
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] ) ;
}
```

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.

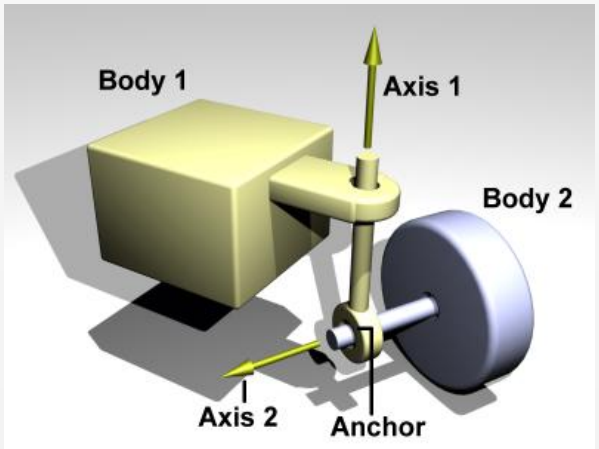
(c) Walt Disney Company, from "The Illusion of Life"

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:

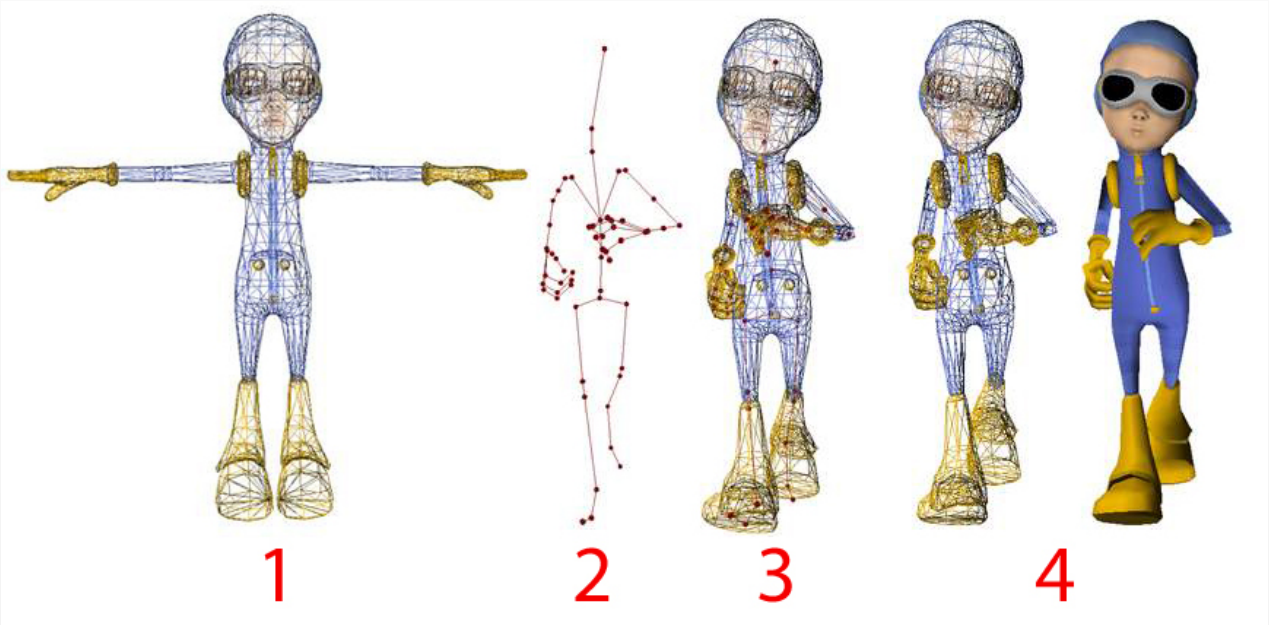
- ODE: Open Dynamic Engine.
- Newton: Newton Physics Engine
- Bullet: Physics Library



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.