Informática Gráfica.
Sesión 9: Interacción.

Carlos Ureña, Sept 2025.
Dept. Lenguajes y Sistemas Informáticos.
Universidad de Granada.

Índice

Introducción 3
Eventos en Godot 17
Interfaz de usuario y señales en Godot. 39
Selección 67

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

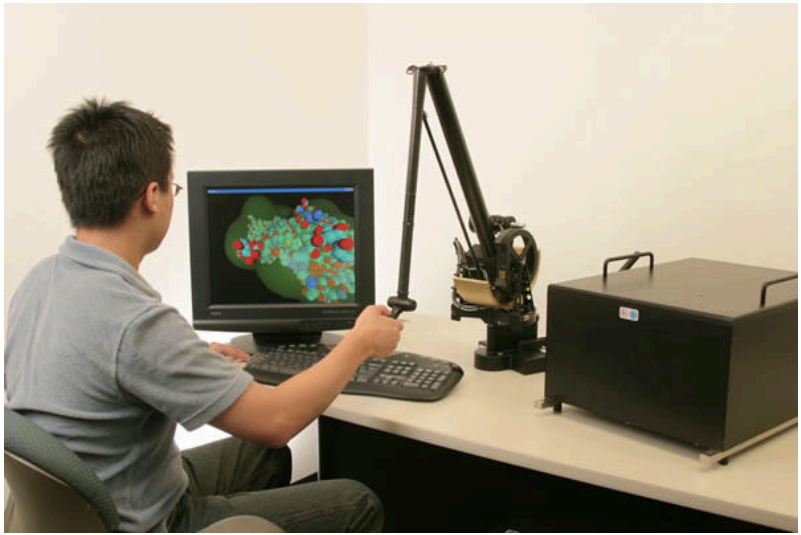
La incoporación de interactividad permite realizar aplicaciones que respondan ágilmente a las acciones de los usuarios y les ofrezcan retroalimentación sobre el efecto de dichas acciones.

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.

Algunas veces, sin embargo, se imponen **requerimientos de tiempo real estrictos**:

- Simuladores, sistemas de realidad virtual, gemelos digitales para vehículos o aviones, donde es necesario reproducir los tiempos de respuesta reales del sistema.
- Videojuegos de alta gama, donde se requiere una latencia mínima para una experiencia de usuario óptima.

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 (directamente o con deslizadores, por ejemplo).
- 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 obtener coordenadas en pantalla, calculadas 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.
- **Detección de gestos** a partir de una secuencia de vídeo en tiempo de real de las manos, se pueden reconocer determinados gestos (previamente definidos) y generar eventos de alto nivel asociados a dichos gestos. Por ejemplo: *mano abierta*, *puño cerrado*, *dedo índice apuntando*.

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 Godot

1. Tipos de eventos en Godot. La clase *InputEvent*.
2. Gestión de eventos.

Subsección **2.1.**

Tipos de eventos en Godot. La clase *InputEvent*.

Eventos en Godot. La clase `InputEvent`

Godot ofrece la clase `InputEvent` (un tipo de `Resource`) para eventos de entrada:

- Un evento es un objeto con datos sobre un cambio de estado en un dispositivo físico o lógico.
- Para cada tipo de evento, existe una clase específica derivada de `InputEvent`
- Muchos eventos se originan en un *viewport* cuya ventana tiene el foco.
- El sistema operativo y/o el sistema de ventanas asocian los eventos a un *viewport* y lo redirigen a la aplicación Godot.
- Cualquier nodo en un árbol de escena puede tener asociados métodos para gestionar eventos de entrada de uno varios tipos.
- Cada uno de esos métodos recibe como parámetro un objeto con los datos del evento, y puede declarar el evento como *consumido* o no.
- Un evento puede ser gestionado por varios nodos, hasta que no hay más nodos que lo gestionen o bien hasta que sea consumido por el último de ellos.

Tipos de eventos en Godot

Las clases derivadas de `InputEvent` son:

- `InputEventFromWindow`: eventos originados en una ventana, un viewport dentro de una ventana, o en una pantalla táctil. Las subclases son:
  - `InputEventWithModifiers` : eventos de teclado, ratón o gestos en pantallas táctiles.
  - `InputEventScreenDrag` : eventos de arrastre de elementos del GUI.
  - `InputEventScreenTouch` : eventos de toque generados en pantallas táctiles
- `InputEventJoypadButton`: eventos de botones de mando de juego (*joypad*)
- `InputEventJoypadMotion`: eventos de movimiento de mando de juego.
- `InputEventAction`: eventos asociados a acciones definidas en el *mapa de entradas* de Godot.
- `InputEventShortcut`: eventos de atajos de teclado
- `InputEventMIDI`: eventos recibidos de dispositivos MIDI.

Eventos de teclado, ratón y gestos

Los eventos más comunes son los de teclado, ratón, o gestos en pantallas táctiles. Se usan las clases que derivan de `InputEventWithModifiers`:

- `InputEventKey` : eventos de pulsar o levantar teclas de un teclado
- `InputEventMouse` : eventos de ratón: desplazamiento del ratón, de su rueda, pulsar o levantar botones del ratón.
- `InputEventGesture` : eventos de gestos producidos en pantallas táctiles.

Estos eventos tienen propiedades con datos sobre el estado de ciertas teclas *modificadoras* en el momento de producirse el evento, a saber:

- `alt_pressed` : si la tecla *Alt* pulsada vale `true`
- `shift_pressed` : tecla *Shift*
- `control_pressed` : tecla *Control*
- `meta_pressed` : tecla *Meta* (*Windows*, *Cmd*, etc.)

Eventos de teclado (*InputEventKey*)

La clase `InputEventKey` tiene las siguientes propiedades importantes:

- `keycode` (tipo `Key`): código de la tecla pulsada o levantada, independiente del idioma del teclado.
- `key_label` (tipo `Key`): código de la tecla, dependiente del idioma del teclado.
- `unicode` (tipo `int`): código Unicode del carácter generado por la tecla (si lo hay)
- `pressed` (tipo `bool`): vale `true` si la tecla se ha pulsado, o `false` si se ha levantado
- `echo` (tipo `bool`): vale `true` si el evento es producido por una repetición automática al mantener la tecla pulsada.
- `location` (tipo `KeyLocation`): para teclas con dos copias en el teclado (como *Shift* o *Control*), indica si la tecla está en la parte izquierda o derecha del teclado.

Eventos de ratón (*InputEventMouse*)

La clase `InputEventMouse` es la clase base para eventos de ratón. Propiedades:

- `position` (de tipo `Vector2`), con la posición del ratón en el viewport (en unidades de pixels) en el momento del evento.
- `global_position` (de tipo `Vector2`), idem, pero para el *viewport raíz*.
- `button_mask` (`MouseButtonMask`): bits con estado de los botones.

Hay dos sub-clases de `InputEventMouse`:

- `InputEventMouseButton`: pulsar o levantar botones del ratón. Propiedades:
  - `button_index` : constante que identifica e botón pulsado o levantado (tipo `MouseButton`)
  - `pressed` : vale `true` si el botón se ha pulsado, o `false` si se ha levantado
- `InputEventMouseMotion` : eventos de movimiento del ratón. Propuedad:
  - `relative` (tipo `Vector2`): desplazamiento en pixels del ratón desde la última posición registrada (el último frame).

Subsección **2.2.**

Gestión de eventos.

Métodos de gestión de eventos

En Godot, el desarrollador puede escribir código que se ejecutará cuando se produzca algún evento de entrada. Para ello, puede asociar a cualquier nodo del árbol de escena (objeto de tipo `Node`) **métodos de gestión de eventos**, que reciben como parámetro un objeto de tipo `InputEvent`. Son los siguientes (listados en orden de prioridad):

(1) `_input`

(2) `_shortcut_input` (únicamente para eventos `InputEventKey`, `InputEventShortcut`, or `InputEventJoypadButton`).

(3) `_unhandled_key_input` (únicamente para eventos `InputEventKey`)

(4) `_unhandled_input`

Cuando se produce un evento se ejecuta el primer método (`_input`) para todos los nodos en un árbol de escena que lo tengan definido, hasta que alguno de ellos lo ***consume***. Si ninguno lo hace, se repite el proceso para el segundo método, y así hasta el cuarto. Entre (1) y (2) un evento **puede ser consumido por un control del *GUI***.

Orden de ejecución de métodos de gestión de eventos

Para cada uno de los 4 métodos de gestión de eventos anteriores, Godot ejecuta el método para todos los nodos del árbol (que lo tengan definido) en un orden **primero en profundidad**, pero **inverso entre hermanos** (*inverso* respecto al orden de los hijos en la lista del padre).

En esta captura de un árbol de escena de Godot, cada nodo tiene un nombre que indica su orden a efectos de gestión de eventos.

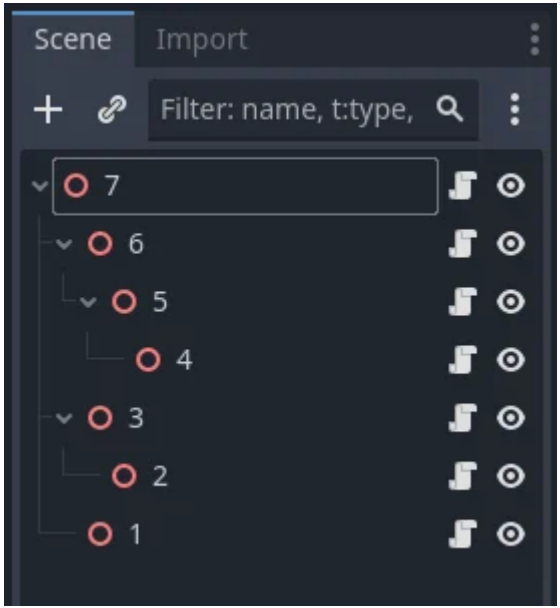


Imagen de la documentación de Godot: Using *InputEvent*

Procesamiento y consumición de eventos

Estos métodos tienen un parámetro `event` de tipo `InputEvent`. En ellos:

- Se puede comprobar de qué clase es el evento, usando `is`. Por ejemplo:

```
if event is InputEventKey:
  # código para gestionar evento de teclado
```

- Cuando ya se sabe el tipo se puede acceder a las propiedades específicas de ese tipo. Por ejemplo:

```
if event is InputEventKey:
  if event.pressed:
    # código para gestionar evento de pulsación de tecla
```

- Se puede invocar el método `set_input_as_handled` de la clase `Viewport`, para **declarar el evento como consumido**. Por ejemplo:

```
if event is InputEventKey:
  if event.pressed and event.keycode == KEY_A:
    get_viewport().set_input_as_handled() ## consume pulsación de 'A'
```

Ejemplo de procesamiento de eventos

Ejemplo tomado de la cámara orbital simple de las prácticas, el método `_input` responde a eventos de teclado y ratón. Para ello actualiza las variables `dxy`, `dz` y `bdrp`. En esta parte inicial se ve el código de gestión de las teclas:

```
func _input( event : InputEvent ):
  var av : bool = true ## actualizar vista sí/no

  if event is InputEventKey and event.pressed: ## pulsaciones de teclas
    match event.keycode:
      KEY_UP:     dxy += Vector2( 0, -at )
      KEY_DOWN:   dxy += Vector2( 0, +at )
      KEY_RIGHT:  dxy += Vector2( -at, 0 )
      KEY_LEFT:   dxy += Vector2( at, 0 )
      KEY_MINUS, KEY_PAGEDOWN, KEY_KP_SUBTRACT: dz *= 1.05
      KEY_PLUS, KEY_PAGEUP, KEY_KP_ADD:         dz = max( dz/1.05, 0.1 )
      _: av = false

  ## ..... continua ...
```

Ejemplo de procesamiento de eventos

En esta segunda parte vemos el código de gestión de eventos de ratón:

```
func _input( event : InputEvent ):

  ## ... código anterior ...

  elif event is InputEventMouseButton: ## botón ratón o rueda
    match event.button_index:
      MOUSE_BUTTON_RIGHT:       bdrp = event.pressed ; av = false
      MOUSE_BUTTON_WHEEL_DOWN: dz *= 1.05
      MOUSE_BUTTON_WHEEL_UP:    dz = max( dz/1.05, 0.1 )
      _: av = false

  elif event is InputEventMouseMotion and bdrp: ## movim. ratón
    dxy += ar * Vector2( -event.relative.x, event.relative.y )

  else: av = false # no actualizar la vista

  if av :  ## actualizar la vista ....
```

El mapa de entrada. Eventos y acciones.

Durante la ejejcución de una aplicación Godot, los eventos de entrada pueden ser mapeados a **acciones** definidas en el **mapa de entrada** (*Input Map*).

- El mapa de entrada es un objeto *singleton* (solo hay una instancia global), de la clase `InputMap`.
- Contiene una lista de acciones, cada una de ellas es una instancia de la clase `InputEventAction` que se identifica con una cadena única llamada **nombre de la acción** (propiedad `action` de la clase, de tipo `StringName`).
- Se puede configurar en el editor, o bien desde código, con los métodos `add_action` y `action_add_event` de la clase `InputMap`
- Cada acción del mapa de entrada tiene asociados una o varias instancias de la clase `InputEvent` (o de cualquier subclase). Son los eventos que **disparan** esa acción.

La **ventaja** del mapa de entrada es que **permite separar** el **tipo de evento** que las dispara del **código que las gestiona**.

Ejemplo de mapa de entrada en el editor

En este ejemplo se asocian pusaciones de teclas y *ejes* del joypad a acciones de movimiento del personaje controlado por el jugador:

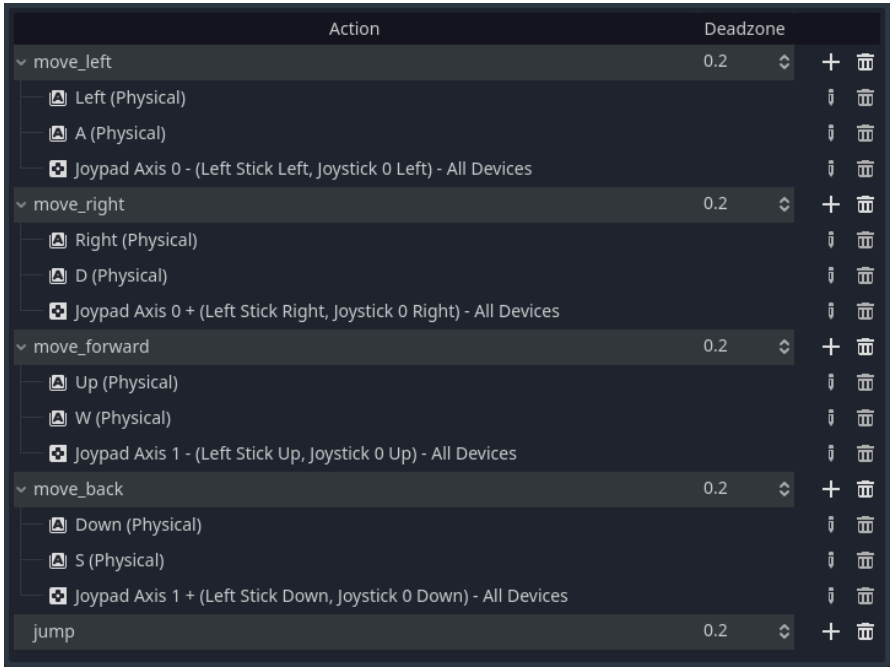


Imagen de la documentación de Godot: *First 3D game: player input*

Acciones predefinidas en el mapa de entrada

Godot incorpora varias acciones predefinidas (*built-in actions*) en el mapa de entrada (se pueden visualizar con el editor si se activa la opción correspondiente):

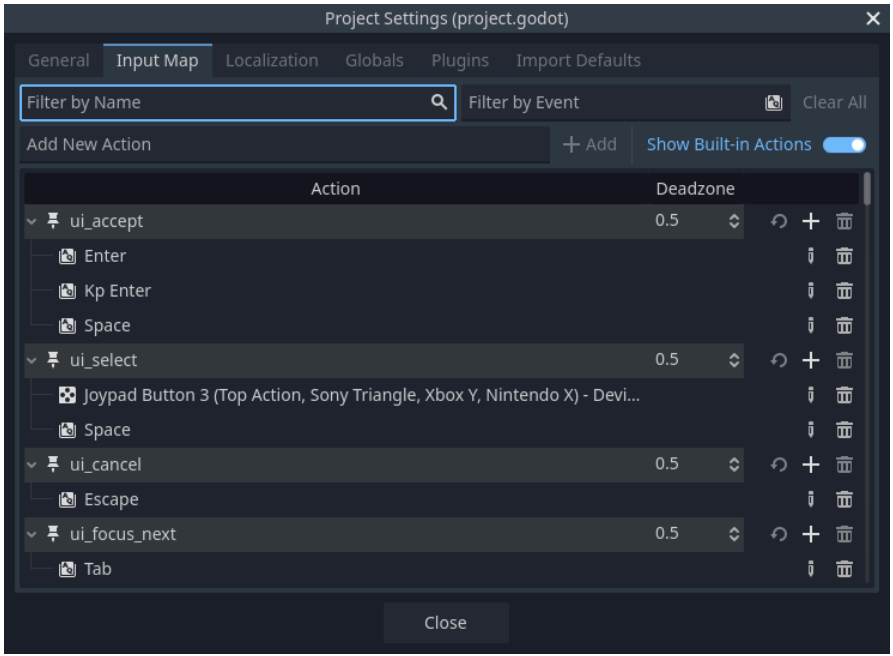


Imagen de la documentación de Godot: *First 3D game: player input*

Definición de acciones usando código

Las acciones se pueden definir y asociar a eventos usando código GDScript, por ejemplo, este código (asociado a un nodo cualquiera) crea una acción que se disparará cuando se pulse o se levante la tecla *A* o la tecla *flecha arriba*:

```
## crear instancia de evento: pulsar o levantar tecla 'A'
var evento_tecla_A := InputEventKey.new()
evento_tecla_A.keycode = KEY_A

## crear instancia de evento: pulsar o levantar tecla 'flecha arriba'
var evento_tecla_arriba := InputEventKey.new()
evento_tecla_arriba.keycode = KEY_UP

## añadir la acción al mapa de entrada
InputMap.add_action( "accion_creada_script" )

## asociar los eventos a la acción
InputMap.action_add_event( "accion_creada_script", evento_tecla_A )
InputMap.action_add_event( "accion_creada_script", evento_tecla_arriba )
```

Respondiendo a acciones en métodos de eventos

Una vez definidas las acciones en el mapa de entrada, se puede escribir código para responder a eventos asociados co ¡n una acción, usando el método `is_action` de la clase `InputEvent`.

Por ejemplo, este código (asociado a un nodo cualquiera) responde a la acción creada en el ejemplo anterior (`accion_creada_script`) y a otra acción llamada `accion_tecla_Q`. En ambos casos el evento es consumido:

```
func _input( event : InputEvent ):

 if event.is_action( "accion_tecla_Q" ) :
    print("Nodo raíz: 'accion_tecla_Q' disparada")
    get_viewport().set_input_as_handled()

 elif event.is_action( "accion_creada_script" ) :
    print("Nodo raíz: 'accion_creada_script' disparada")
    get_viewport().set_input_as_handled()
```

Muestreo de dispositivos. La clase *Input*

En ocasiones puede ser útil usar **muestreo de estado** (***polling***) de los dispositivos de entrada en lugar de las funciones de gestión de eventos.

- Para ello Godot tiene la ***clase singleton*** `Input` con métodos para consultar el estado de teclas, botones de ratón, acciones, ejes o botones de *joypads*, giroscopios, acelerómetros, magnetómetros, etc...

Podemos **consultar el estado de acciones** con estos métodos de `Input`:

- `is_action_pressed`: es `true` si la acción **está pulsada en el momento de la llamada.**
- `is_action_just_pressed`, `is_action_just_released`: son `true` si la acción se ha pulsado o levantado **desde el último frame anterior a la llamada**.
- `is_action_just_pressed_by_event`, `is_action_just_released_by_event`: es `true` si la acción se ha pulsado o levantado **desde el último frame anterior a un evento**.

Ejemplo de muestro de una acción asociada a un tecla

En este ejemplo se usa el método `_input` y además muestro en cada frame (en el método `_process`). Se usa la acción llamada `accion_tecla_Q`, asociada a eventos de pulsar y levantar la tecla *Q*.

```
func _input( event : InputEvent ) :  ## se ejecuta en cada evento
 if event.is_action("accion_tecla_Q" ):
  print("_input: 'accion_tecla_Q' - pressed es ", event.pressed)

var contador : int = 0 ## contador de frames.

func _process( delta : float ):  ## se ejecuta en cada frame
  contador = contador+1
  if Input.is_action_just_pressed("accion_tecla_Q"):
    print("_process ",contador,": 'accion_tecla_Q' pulsada ahora")
  if Input.is_action_just_released("accion_tecla_Q"):
    print("_process ",contador,": 'accion_tecla_Q' levantada ahora")
  if Input.is_action_pressed("accion_tecla_Q"):
    print("_process ",contador,": 'accion_tecla_Q' está pulsada")
```

Ráfaga de mensajes asociados a una pulsación en el ejemplo

Con el código anterior, si el usuario pulsa y levanta la tecla *Q*, se produce una ráfaga de mensajes como los que se muestran a continuación:

```
_input: 'accion_tecla_Q' - pressed es true
_process 161: 'accion_tecla_Q' pulsada ahora
_process 161: 'accion_tecla_Q' está pulsada
_process 162: 'accion_tecla_Q' está pulsada
_process 163: 'accion_tecla_Q' está pulsada
_process 164: 'accion_tecla_Q' está pulsada
_input: 'accion_tecla_Q' - pressed es false
_process 165: 'accion_tecla_Q' levantada ahora
```

El número de frames en los que se repite el mensaje *"accion_tecla Q" está pulsada* depende de la duración de la pulsación de la tecla por parte del usuario (esto puede usarse para comportamientos que puedan depender de dicho tiempo).

Problema

Problema 9.1:

En una aplicación Godot cualquiera, añade código al nodo raíz de forma que cada vez que se pulse y luego se levante una tecla (por ejemplo la tecla *P*), se imprima en pantalla un mensaje con el tiempo total en segundos que dicha tecla ha estado pulsada, en los casos en los que ha permanecido pulsada al menos el tiempo de un frame.

Sección **3.**

Interfaz de usuario y señales en Godot.

1. Interfaz de usuario.
2. Clases de controles.
3. Señales

Subsección **3.1.**

Interfaz de usuario.

Interfaz de usuario (IU)

En general, el término **Interfaz Gráfica de Usuario** (o **GUI**, de **Graphical User Interface**, o simplemente **UI**) se refiere elementos visuales que aparecen en la pantalla durante la ejecución de una aplicación y que

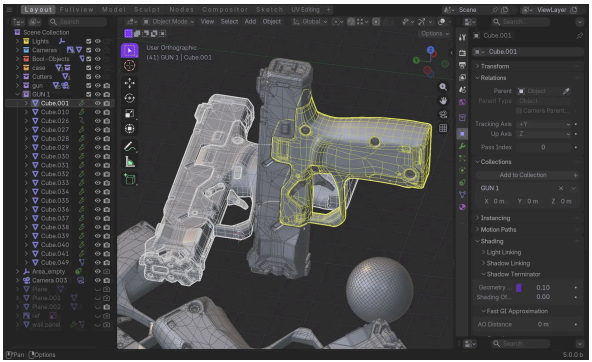
- permiten presentar información al usuario en forma de texto, valores numéricos, colores, iconos, tablas de datos, infografías, etc...
- permiten al usuario introducir información como textos, valores lógicos, cantidades, colores, selecciones de una opción de entre una lista, etc..

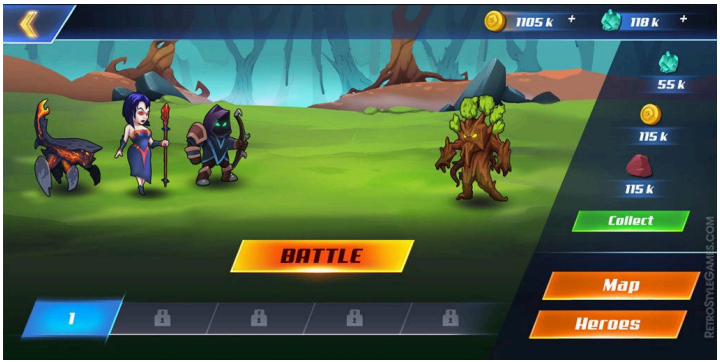
En particular, en el contexto de las **aplicaciones gráficas interactivas 2D o 3D**, el término **interfaz de usuario** (***user interface***) se suele referir a los elementos visuales distintos de las proyecciones 2D o 3D de los objetos de la escena, elementos que sirven únicamente para presentar o pedir información al usuario, pero no son representaciones visibles del modelo 3D o 2D que la aplicación gestiona.

Ejemplos de interfaces de usuario

En estos ejemplos de aplicaciones gráficas interactivas, el **UI** está formado por todos los elementos visuales en pantalla (usualmente en 2D) que se superponen (o rodean) a la imagen 2D o 3D del modelo de escena y objetos.

Estos elementos permiten presentar información de estado al usuario o pedirle que introduzca datos o selecciones.





Imágenes de: *Blender extensions. SHEK:* *Theme Plasticity* (izquierda)
Padel Konstantinov *What is UI in Games ?* (derecha)

Ejemplo de GUI creado con Godot

En esta captura vemos los elementos del GUI rodeando la vista 3D central.

Imágen del editor de mundos virtuales ***RPG in a Box*** zeromatrix.itch.io/rpginabox

El interfaz de usuario en Godot

Godot permite incorporar elementos de interfaz de usuario:

- Se usa la clase base `Control`, derivada de `CanvasItem`, a su vez derivada de `Node`.
- Cada instancia de una clase derivada de `Control` es un elemento visual 2D, que ocupará un **área rectángular** en un *viewport* y cuyo *estilo* es configurable.
- Los objetos `Control` capturan y responden a **eventos de entrada**.
- Algunos controles actúan como **contenedores** de otros, lo cual permite organizarlos jerárquicamente.
- El árbol de escena de la aplicación puede incorporar nodos de clases derivadas de `Control`, donde la relación padre-hijo se interpreta en términos de contenedor-contenido, es decir, **el rectángulo del hijo está incluido dentro del del padre**.
- Al igual que el resto de nodos, los nodos de tipo `Control` pueden crearse y configurarse **mediante scripts y/o con el editor**

Propiedades de la clase `Control`

La forma y posición del rectángulo ocupado por un control en el *viewport* viene determinada por las siguientes propiedades de la clase `Control`:

- `position` (tipo `Vector2`): posición de la esquina superior izquierda del rectángulo, relativo a la posición del control padre (o del *viewport* si no tiene).
- `size` (tipo `Vector2`): ancho y alto del rectángulo, en unidades de pixels.
- `rotation` (tipo `float`): ángulo de rotación (en radianes) alrededor del pivote
- `scale` (tipo `Vector2`): factor de escala en los ejes X e Y, alrededor del pivote.
- `pivot_offset` (tipo `Vector2`): posición relativa del pivote en pixels.

El estilo se puede configurar usando estas propiedades:

- `theme` (tipo `Theme`): permite definir y aplicar estilos coherentes a múltiples controles.
- `theme_type_variation` (tipo `StringName`): permite aplicar variaciones de estilo definidas en el tema.

Subsección **3.2.**

Clases de controles.

Galería de controles de Godot

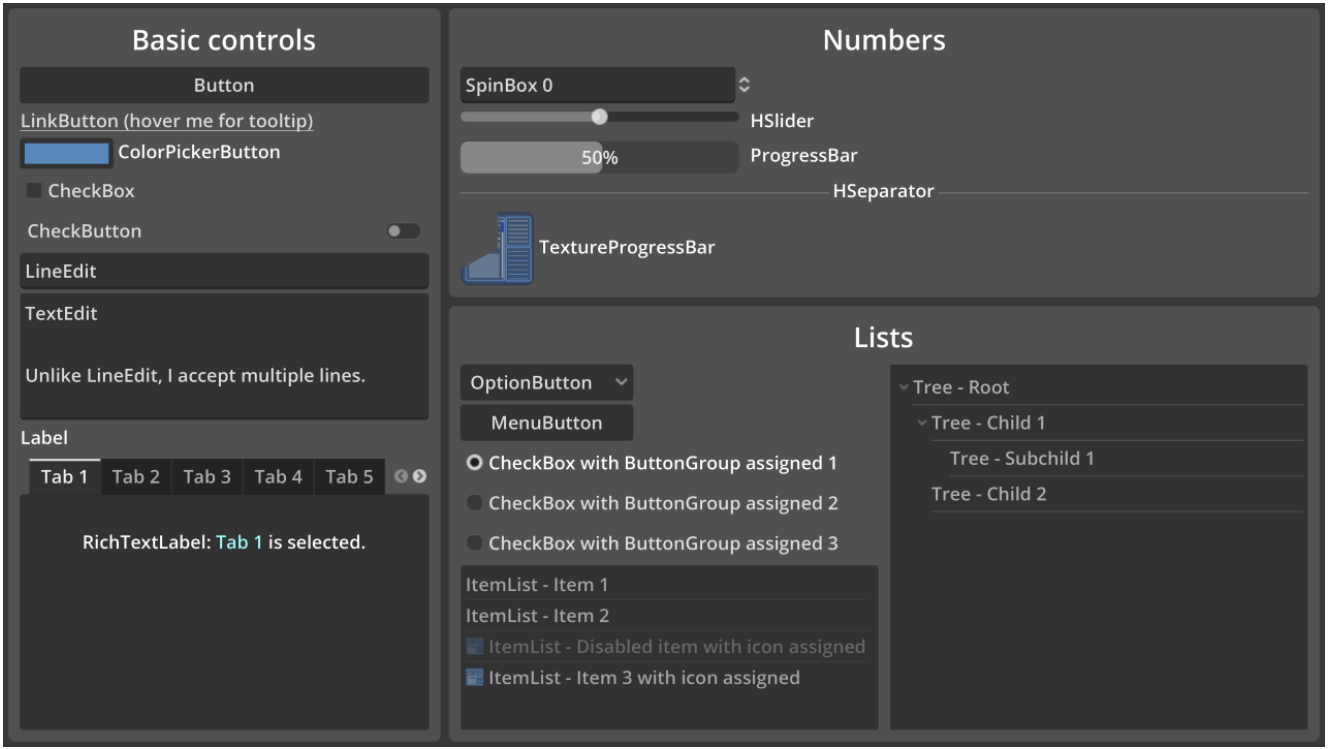


Imagen de *Godot Asset Library: Control Galery Demo*
godotengine.org/asset-library/asset/2766

Algunas clases derivadas de `Control`

- `BaseButton`: clase base para botones.
- `Container`: clase base para contenedores de otros controles.
- `Range` : clase base para valores numéricos en un rango.
- `ItemList`: lista vertical de items seleccionables.
- `Label` : caja con texto plano, puede ser multilínea.
- `LineEdit`: entrada de texto de una línea.
- `MenuBar`: barra de menús.
- `RichTextLabel`: caja con texto enriquecido (con icónos o imágenes).
- `Separator`: línea horizontal separadora.
- `TabBar` : barra de pestañas en horizontal.
- `TextEdit`: editor de textos de múltiples líneas.
- `Tree`: visor de estructuras jerárquicas de items.
- `VideoStreamPlayer`: reproductor de vídeos.

Otras clases derivadas de `Control`

Otras clases derivadas directamente de `Control` son:

- `GraphEdit`: editor gráfico interactivo de grafos de nodos, cada nodo es un objeto `GraphNode`, se usa típicamente para editar interactivammente estructuras de datos que representan diagramas de flujos de datos, o cualquier otra estructura de tipo grafo.
- `ColorRect`: rectángulo de color sólido.
- `NinePatchRect` : rectángulo con una imagen dividida en 9 rectángulos de igual tamaño, de forma que al escalar el control, cada rectángulo se escala en X y/o Y dependiendo de su posición.
- `ReferenceRect`: rectángulo con un borde resaltado.
- `TextureRect`: rectángulo que muestra una imagen (textura).
- `Panel` : rectángulo con estilo configurable (una instancia de `StyleBox`).

Clases para botones (derivadas de `BaseButton`)

Las clases derivadas de `BaseButton` permiten crear botones con diferentes apariencias y comportamientos:

- `Button`: botón estándar con texto o icono. Se puede usar directamente, o bien via algunas de las clases derivadas, a saber:
  - `CheckBox`: entrada de valores lógicos (true/false), o en general selección de una entre dos opciones excluyentes.
  - `CheckButton`: similar a la anterior, pero con apariencia de botón.
  - `ColorPickerButton` : selector de colores, permite seleccionar un color usando uno de entre varios modelos de color (RGB, HSV, etc...)
  - `MenuButton`: despliega un menú flotante (instancia de `PopupMenu`)
  - `OptionButton`: despliega un `PopupMenu` para seleccionar una opción, cuyo texto se muestra en el botón.
- `LinkButton`: botón con una *URI* que se abre con el navegador.
- `TextureButton`: botón con una imagen en lugar de un texto o un tema.

Clases para valores numéricos (derivadas de `Range`)

Las clases que permiten entrada de valores numéricos son:

- `SpinBox` : entrada de valores numéricos, mediante el teclado, y adicionalmente con dos botones para incrementar o decrementar el valor.
- `Slider`: entrada de valores numéricos arrastrando con el ratón un elemento en un segmento o *carril*, hay dos sub-clases `HSlider` (horizontal) y `VSlider` (vertical)
- `ScrollBar`: barra de scroll. También hay dos sub-clases `HScrollBar` (horizontal) y `VScrollBar` (vertical).

Otras clases también derivadas de `Range` permiten mostrar valores numéricos en forma gráfica, son:

- `ProgressBar` barra de progreso, muestra como cambia en el tiempo el porcentaje completado de un proceso
- `TextureProgressBar` igual, pero usando una imagen.

Clases contenedoras (derivadas de `Container`) (1/2)

- `BoxContainer` caja con los nodos hijos dispuestos en horizontalmente en una fila (`HBoxContainer`) o verticalmente en una columna (`VBoxContainer`).
- `GridContainer` caja que dispone los nodos hijos en una cuadrícula con un número fijo de columnas (el número de filas depende de cuantos controles hijos se le añadan).
- `TabContainer` contenedor con múltiples pestañas, cada pestaña contiene un control hijo.
- `MarginContainer` caja contenedora que añade márgenes alrededor del control o controles hijos.
- `SplitContainer` contenedor con dos hijos que permite al usuario mover la línea que los separa, puede ser horizontal (`HSplitContainer`) o vertical (`VSplitContainer`).
- `SubViewportContainer` contenedor cuyo rectángulo constituye un *viewport* (instancia de `SubViewport` en el cual se visualiza una escena 2D o 3D).

Clases contenedoras (derivadas de `Container`) (2/2)

- `AspectRatioContainer`: contenedor que mantiene una relación de aspecto fija (ancho/alto) para el área ocupada por sus hijos.
- `CenterContainer`: contenedor que centra su(s) hijo(s) en el área disponible.
- `FlowContainer`: contenedor que dispone sus hijos en filas (`HFlowContainer`), pasando a la siguiente fila cuando no hay espacio disponible (como se disponen palabras en un párrafo de texto). También pueden configurarse por columnas (`VFlowContainer`).
- `FoldableContainer`: un contenedor que puede plegarse a una sola línea con texto o expandirse y mostrar todos sus nodos hijos.
- `GraphElement` contenedor para elementos de un control `GraphEdit`. Tiene dos subclases: `GraphNode` y `GraphFrame`.
- `ScrollContainer` contenedor que añade barras de scroll automáticvamente a sus nodos hijos, si el área disponible es menor que la requerida.

Creación del GUI

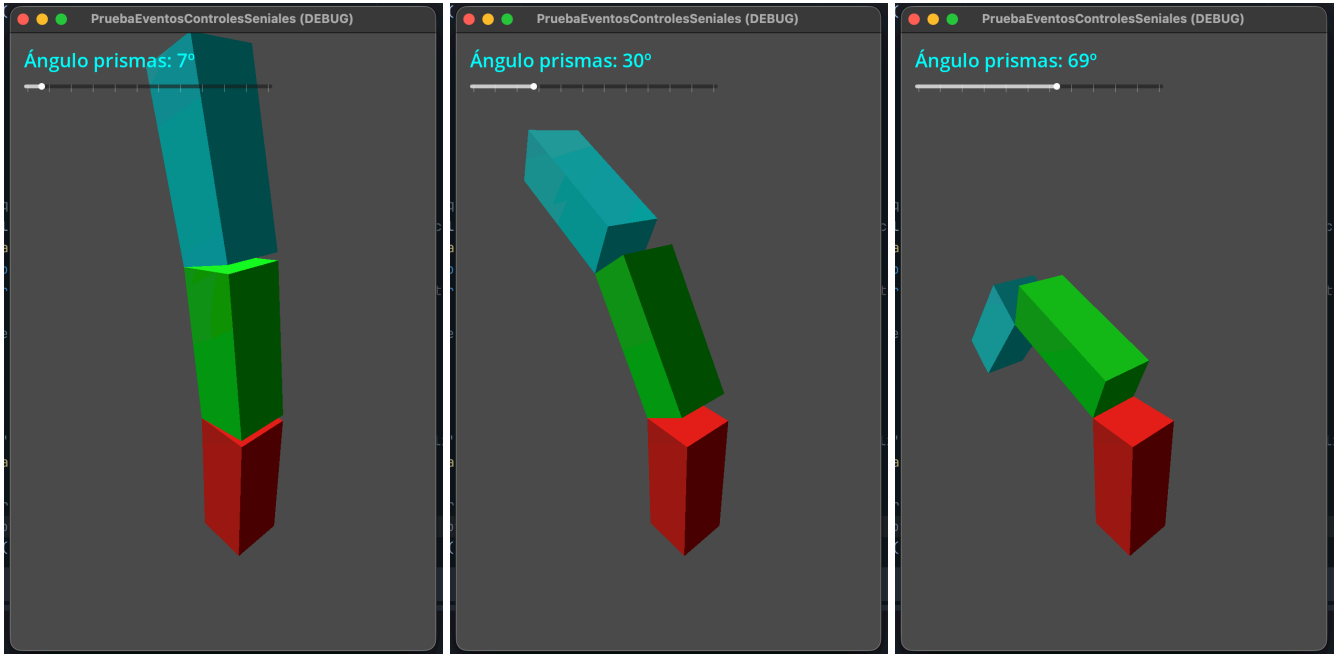
Los controles pueden crearse tanto en el editor como mediante scripts.

- **En el editor**, se añade un árbol de nodos, todos ellos derivados de `Control`, donde los nodos contendores son nodos no terminales que incluyen a sus nodos hijos.
- **Mediante código *GDScript***, creando y configurando cada nodo, y añadiendo los nodos hijos a sus nodos. Se puede hacer, por ejemplo, en el método `_ready()` del nodo raíz de una escena (o en cualquier otro).

En cualquier caso, los controles aparecerán en el viewport asociado al nodo contenedor raiz del GUI.

Ejemplo: deslizador (*slider*) y etiqueta (*label*)

Vemos un ejemplo de creación con GDScript de un control deslizador horizontal `HSlider` y una etiqueta `Label` que muestra el valor actual del deslizador. Controla un grado de libertad de un modelo jerárquico (ángulo entre prismas).



Ejemplo: código GDScript del deslizador y la etiqueta (1/2)

Para crear una caja con el texto y el deslizador podemos añadir código al método `_ready` de cualquier nodo asociado al viewport donde queremos ver el GUI:

```
var etiqueta   : Label = null
var deslizador : HSlider = null

func _ready():  ## es el método `_ready` del nodo raíz (u otro nodo)

  ## crear y confgurar el deslizador
  deslizador = HSlider.new()
  deslizador.size_flags_horizontal = Control.SIZE_EXPAND_FILL
  deslizador.ticks_on_borders = true  # mostrar barras verticales
  deslizador.tick_count = 12      # número de barras verticales
  deslizador.min_value = 0.0      # valor a la izquierda (mínimo)
  deslizador.max_value = 120.0    # valor a la derecha (máximo)
  deslizador.value     = 30.0     # valor inicial
  deslizador.step      = 1        # incremento entre valores posibles

  ## .... continúa
```

Ejemplo: código GDScript del deslizador y la etiqueta (2/2)

```
func _ready():  ## es el método `_ready` del nodo raíz (u otro nodo)
   ## .... código anterior

   ## crear y configurar la etiqueta de texto
   etiqueta = Label.new() ## etiqueta
   etiqueta.size_flags_horizontal = Control.SIZE_EXPAND_FILL
   etiqueta.text = "Ángulo prismas: %.0fº" % deslizador.value
   etiqueta.add_theme_font_size_override( "font_size", 40 )
   etiqueta.add_theme_color_override( "font_color", Color( 0, 1, 1 ) )

   ## crear caja contenedora vertical
   var vbox := VBoxContainer.new()   # crea el objeto con caja vertical
   vbox.position = Vector2( 30, 30 ) # relativa al viewport de este nodo
   vbox.size.x   = 520               # ancho de la caja en pixels
   vbox.add_theme_constant_override( "separation", 20 ) # en pixels

   ## crear sub-árbol de controles
   vbox.add_child( etiqueta )
   vbox.add_child( deslizador )
   add_child( vbox )
```

Subsección **3.3.**

Señales

Señales. Señales en Godot.

En general, en el contexto de la programación, el término **señal** se refiere a un mecanismo de comunicación entre diferentes componentes de un sistema, donde un componente (el emisor) puede emitir una señal para notificar a otros componentes (los receptores) que ha ocurrido un evento o que se ha producido un cambio de estado.

En particular, en Godot, una señal es un objeto, instancia de la clase `Signal`, creado por un **nodo emisor** para notificar la ocurrencia de un cambio de estado, de forma que se ejecuten automáticamente (sin llamada explícita) uno varios métodos en otros nodos (receptores).

Las gran ventaja de usar señales es la **separación entre emisores y receptores**:

- Un nodo emisor detecta el cambio de estado relevante y emite la señal, sin necesidad de conocer qué nodos receptores la van a recibir.
- Un o varios nodos receptores pueden ejecutar código cuando ocurre el cambio de estado, sin necesidad de conocer qué nodo emisor ha emitido la señal.

La clase `Signal`

Un objeto de la clase `Signal` se construye especificando:

- Un objeto (de cualquier clase derivada de `Object`) que podrá actuaar como **emisor** de la señal.
- El nombre único de la señal (una cadena de texto, de tipo `StringName`)

Se pueden usar los siguientes métodos de la clase `Signal`:

- `connect` conecta la señal a un objeto *invocable* (`Callable`) receptor.
- `disconnect` desconecta la señal de un objeto receptor.
- `emit` emite la señal, ejecutando automáticamente los métodos de los objetos receptores conectados (o las funciones conectadas).

Un **objeto invocable** es una instancia de la clase `Callable` y puede ser:

- Una función global (no método de ningún objeto)
- Un método de un objeto concreto.

Señales emitidas por un nodo

Cualquier nodo (objeto de una clase derivada de `Node`) puede emitir un conjunto de señales, que dependen del tipo del nodo. De especial interés son las señales emitidas por los nodos del interfaz (de clases derivadas de `Control`)

En el editor podemos ver la señales emitidas por un nodo seleccionado (en el panel, pestaña *Nodo*, icono *Señales*). A modo de ejemplo, algunas emitidas por un control `Button`:

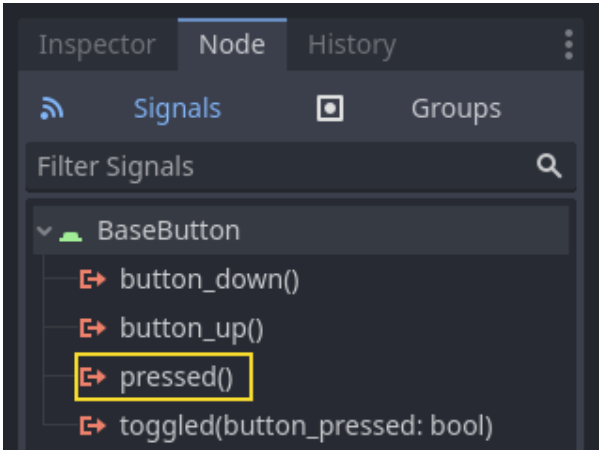


Imagen de la documentación oficial de Godot: *using signals*

Conexión de señales predefinidas en el editor

Usando el editor se puede conectar una señal emitida por un nodo (identificada por su nombre) a un método de otro nodo receptor. En este caso se conecta la señal de nombre `pressed` al método `on_button_pressed` del nodo del árbol con nombre `Sprite2D`

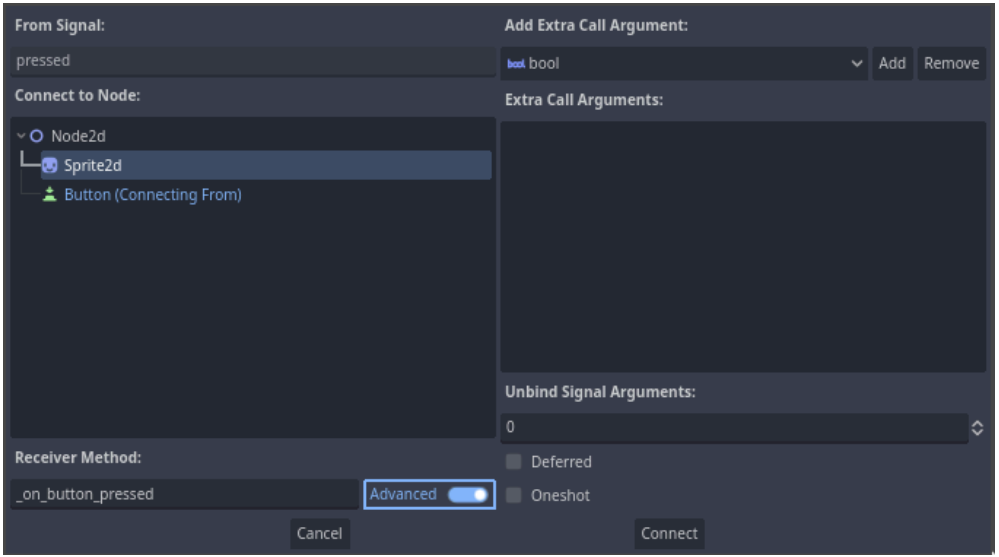


Imagen de la documentación oficial de Godot: *Using signals*

Conexión de señales predefinidas mediante GDScript

La conexión de señales también puede hacerse mediante código GDScript, usando el método `connect` de la clase `Signal`. En el ejemplo anterior, después de crear el deslizador, conectamos su señal `value_changed` al método `_valor_deslizador_actualizado` del nodo actual:

```
var deslizador : HSlider = null
var etiqueta   : Label = null

func _ready():
   ## ... código anterior de creación de los controles
   ## conectar señal y actualizar los prismas la 1a vez.
   deslizador.connect( "value_changed", _valor_deslizador_actualizado )
   $Prismas.fijar_rotacion_nodos( deslizador.value )

func _valor_deslizador_actualizado( valor : float ):
   etiqueta.text = "Ángulo prismas: %.0fº" % valor # actualizar texto
   $Prismas.fijar_rotacion_nodos( valor ) # actualizar los prismas
```

La señal `value_changed` lleva un parámetro con el valor actual.

Acoplamiento entre emisores y receptores de señales

En el script del ejemplo anterior:

- El nodo con el script es el nodo receptor de la señal, y ***conoce*** al nodo emisor (el nodo `deslizador`).
- Asimismo, el objetivo de la señal es ejecutar un método en el nodo de nombre «*Prismas*», así que en el script se ***conoce*** ese nodo de los prismas.

Por tanto:

- Añadir nuevos emisores **requiere extender el script**: será necesario conocer los otros posibles emisores para conectar la señal.
- Añadir nuevos nodos objetivo **requiere extender el script**: será necesario conocer los nuevos nodos objetivo para invocar sus correspondientes métodos.

En definitiva **hay excesivo acoplamiento entre emisores, receptores y nodos objetivo**, lo cual dificulta extender el código. Esto se puede solucionar usando señales definidas por el programador.

Señales definidas por el programador: declaración y emisión

Además de las señales predefinidas en los nodo de tipo `Control`, el programador puede definir sus propias señales (***custom signals***), usando la palabra clave `signal`. Todo esto es **independiente de qué nodos reciban la señal**, para ello usamos un módulo *gdscript* global, por ejemplo en el módulo `Utilidades`:

```
extends Node
## .... otras declaraciones globales en 'Utilidades.gd' ....

## Declaraciones de señales accesibles en todos los scripts:
signal señal1               ## sin parámetros
signal señal2( p : float ) ## un parámetro float
```

Desde cualquier script en un nodo emisor podemos emitir las señales, usamos el método `emit` de `Signal`. Por ejemplo, para emitir las señales anteriores:

```
## emitir las señales (independiente de los posibles receptores):
Utilidades.señal1.emit()          ## sin parámetros
Utilidades.señal2.emit( 34.56 )   ## debemos dar una expresión float
```

Señales definidas por el programador: conexión y recepción

Cualquier nodo interesado en recibir estas señales puede convertirse en receptor de las mismas, conectándose a ellas con el método `connect` de `Signal`. Esto es **independiente de qué nodos emitan la señal**. Por ejemplo, un nodo cualquiera puede suscribirse a las señales anteriores, si se le asigna un script con este código:

```
extends Node

func _ready() -> void:
   ## ... otro código de inicialización ...
   ## conectar las señales con los métodos receptores de este nodo
   Utilidades.señal1.connect( _metodo_señal1 )
   Utilidades.señal2.connect( _metodo_señal2 )

func _metodo_señal1(): # se ejecuta al recibir 'señal1'
 print("Receptor: 'señal1' recibida.")

func _metodo_señal2( v : float ): # se ejecuta al recibir 'señal2'
 print("Receptor: 'señal2' recibida, v == ",v)
```

Sección **4.**

Selección

1. Selección en aplicaciones gráficas interactivas
2. Selección en Godot.
3. Problemas: selección por intersecciones

Subsección **4.1.**

Selección en aplicaciones gráficas interactivas

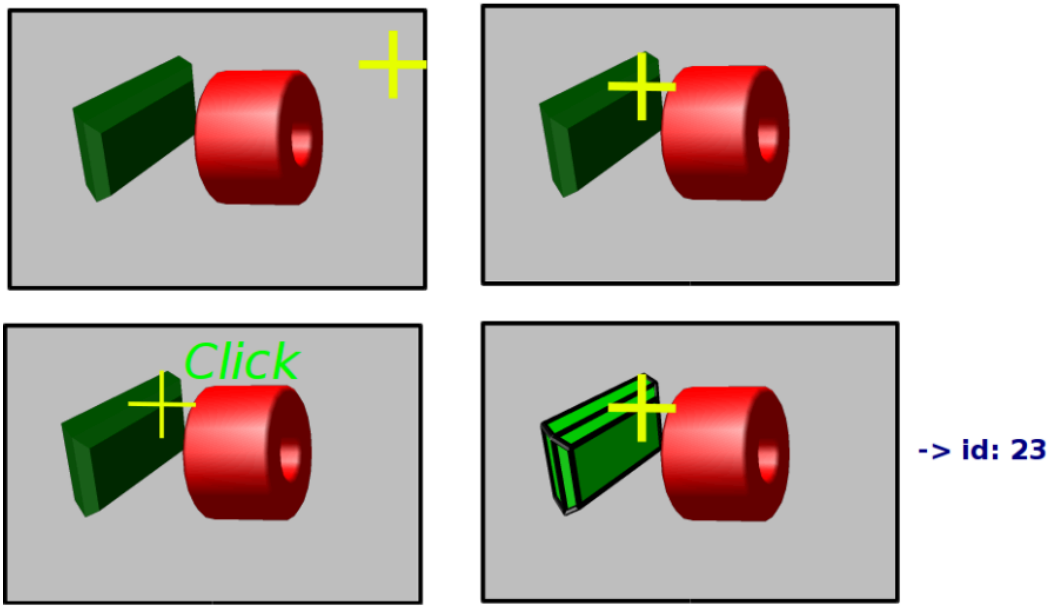
El proceso de selección

En una aplicación gráfica interactiva (2D o 3D), la **selección** es el proceso mediante el cual el usuario usa la aplicación para designar uno o varios objetos o componente de la escena que se está visualizando en pantalla.

- Es una operación fundamental en la interacción usuario-aplicación.
- Permite al usuario indicar uno o varios objetos para: ver o editar sus propiedades, borrarlos, duplicarlos, agruparlos, desagruparlos, moverlos, transformarlos, etc ...
- La selección puede hacerse de diversas formas o con diversos tipos de dispositivos.
- Lo más común es hacer un *click* con el ratón sobre la proyección de un objeto en pantalla (*click* en un pixel donde se proyecta el objeto).
- De esta forma se pueden seleccionar objetos individuales. Para seleccionar varios objetos, se puede hacer uno a uno, añadiendo nuevos objetos a la selección actual en cada paso.

Ejemplo de selección con un *click*

Vemos un ejemplo de selección de un objeto entre varios. Lo usual es que el objeto seleccionado cambie de aspecto de alguna forma (un color resaltado, aristas de contorno resaltadas, etc....).



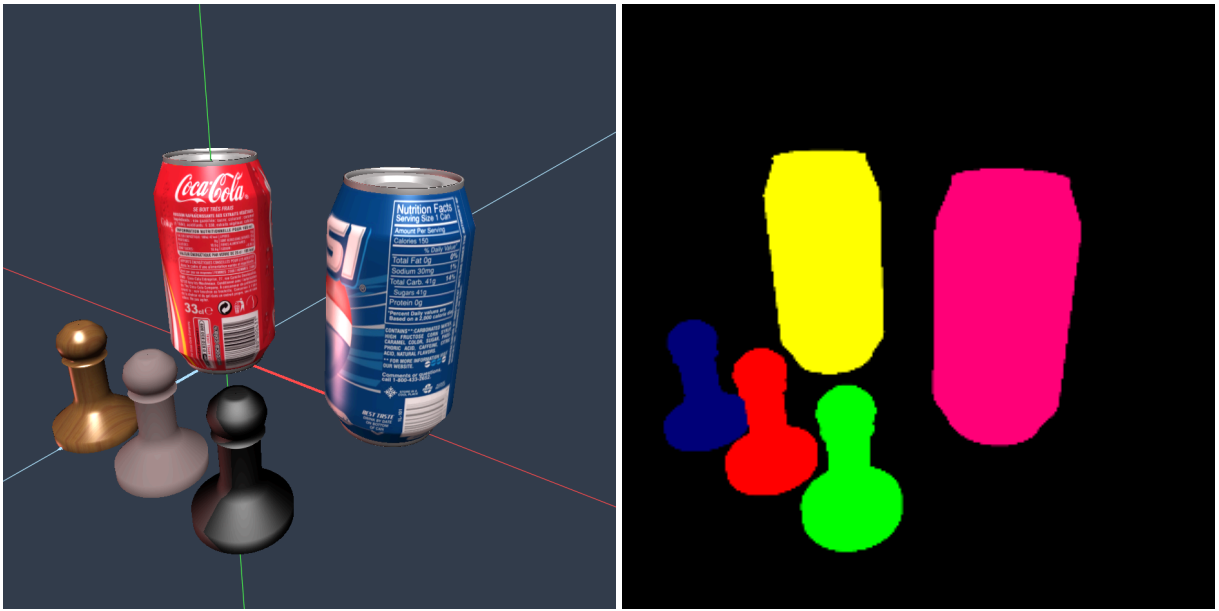
Implementación de la selección

Existen básicamente dos forma distintas de implementar la selección:

- Usando **rasterización** sobre un objeto *framebuffer* oculto:
  - Este *framebuffer* es una zona de memoria en la GPU en la cual se renderiza (por rasterización) la escena. En lugar de usar los colores RGB resultado de texturas e iluminación, a todos los pixels donde se proyecta un objeto se les **asigna un valor entero que identifica al objeto**.
  - Cuando el usuario hace *click* en un pixel, se crea el *famebuffer* y se **lee el identificador de dicho pixel.**
- Usando ***ray casting***, es decir, calculando intersecciones de un rayo:
  - En este caso, se calcula una semirecta (**rayo**) que pasa por todos los puntos de la recta que se proyecta en el punto central a un pixel.
  - Cuando el usuario hace *click* en un pixel, se calcula el correspondiente rayo y se interseca con todos los polígonos o triángulos de la escena. **Se selecciona el más cercano**.

Ejemplo de *framebuffer* de selección

Vemos un ejemplo de selección usando un *framebuffer* oculto. En la imagen de la izquierda se ve la escena renderizada normalmente, y en la imagen de la derecha se ve el *framebuffer* oculto, donde cada pixel tiene un asociado un entero (que aparece como un color RGB en la imagen).



Subsección **4.2.**

Selección en Godot.

Selección en Godot

En Godot, **la selección se puede hacer usando ray-tracing**. Se puede:

- Calcular el rayo que pasa por el centro de un pixel (usando la cámara).
- Calcular intersecciones entre un rayo y los llamados ***nodos colisionadores***. Son objetos con mallas **no visibles** pero **detectables** en el cálculo de intersecciones.
- Este tipo de nodos son de alguna clase derivada de `CollisionObject3D` (clases `StaticBody3D`, `RigidBody3D` entre otras), en los ejemplos usaremos exclusivamente `StaticBody3D`.
- La **forma (geometría) del colisionador** la debe definir otro nodo de clase `CollisionShape3D`, que debe ser **hijo del nodo colisionador**, y que llamamos ***nodo de forma***.
- Todo `CollisionShape3D` tiene la propiedad `shape` de clase `Shape3D`, es el objeto que define la forma, puede ser de varias subclases de `Shape3D`:
  - Objetos simples (`BoxShape3D`, `SphereShape3D`, etc...)
  - Mallas arbitrarias (`ConcavePolygonShape3D` o `ConvexPolygonShape3D`).

Creación de mallas colisionadoras

Los nodos colisionadores se pueden crear a partir de un nodo de la clase `MeshInstance3D`, que contiene la **malla visible** (la que se visualiza). Es necesario:

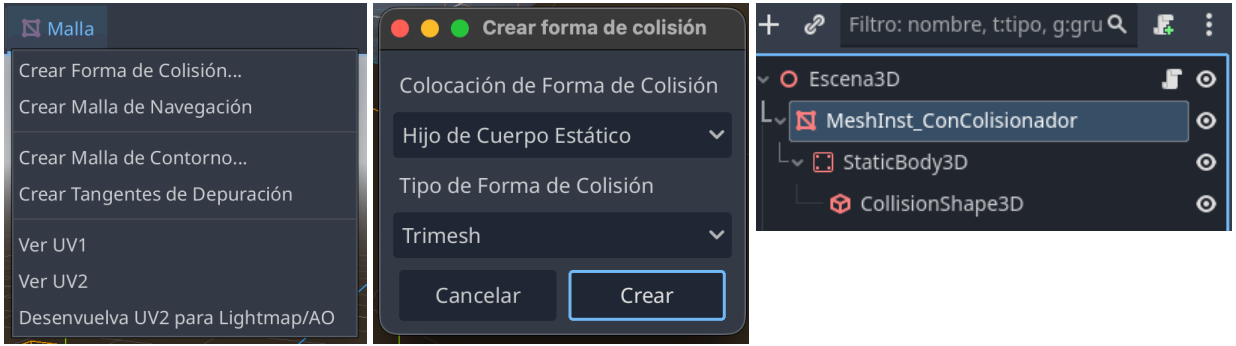
- Crear un nodo `StaticBody3D`, puede ser hijo del `MeshInstance3D`, **para que tenga la misma transformación**.
- Añadir un nodo `CollisionShape3D` como hijo del nodo `StaticBody3D`.
- Asignar un objeto de alguna clase derivada de `Shape3D` en la propiedad `shape` del `CollisionShape3D`.

Esto se puede hacer de dos formas, a partir de un nodo `MeshInstance`

- **Con el editor**: creando el nodo colisionador y el nodo de forma en el editor, o bien usando el menú *Malla/Mesh* y la opción *Crear forma de colisión* (ver siguiente transparencia).
- **En un script**: creando el nodo colisionador y el nodo de forma con código, o bien simplemente **invocando el método** `create_trimesh_collision` del objeto `MeshInstance3D` (ver transparencia).

Creación de mallas colisionadoras en el editor

En el editor, podemos crear un colisionador para un `MeshInstance3D` cualquiera, para ello se selecciona el nodo `MeshInstance3D` y en la parte superior se usa la opción *Crear forma de colisión* en el menú *Malla* (o *Mesh*), creará un hijo `StaticBody3D` y un nieto `CollisionShape3D` con una forma `ConcavePolygonShape3D`.



Vemos: el menú *Malla/Mesh* (izquierda), las opciones para crear el colisionador (centro), y los nodos del árbol con el objeto visible (`MeshInstance3D`), el colisionador (`StaticBody3D`), y su forma (`CollisionShape3D`).

Creación de mallas colisionadoras con un script

En un script, se puede conseguir el efecto equivalente:

- Se parte de un nodo `MeshInstance3D` (con la malla visible), al cual ya se le debe de haber asignado alguna malla en su campo `mesh` (se puede hacer, por ejemplo, en el método `_ready()`, después de la inicialización de `mesh`).
- Se invoca el método `create_trimesh_collision()` del `MeshInstance3D`, que crea automáticamente un hijo `StaticBody3D` y un nieto `CollisionShape3D` con una forma `ConcavePolygonShape3D`.

```
func _ready():
    var tablas : Array = []
    # ..... aquí va el código de inicialización de las tablas.
    mesh = ArrayMesh.new()
    mesh.add_surface_from_arrays( Mesh.PRIMITIVE_TRIANGLES, tablas )

    # crear colisionador automáticamente
    create_trimesh_collision()
```

Intersecciones rayo-malla colisionadora

En Godot, el cálculo de intersecciones rayo-malla se puede hacer usando el método `intersect_ray()` de la clase *singleton* `PhysicsDirectSpaceState`., que calcula la intersección entre un rayo y los colisionadores del árbol de escena.

- El rayo se define por dos puntos: el origen y un punto final (no es una semirrecta, sino un segmento).
- El rayo se puede calcular a partir de las coordenadas del pixel donde se ha hecho click, usando los métodos: `project_ray_origin()` (para el origen) y `project_ray_normal()` (para el vector director, desde el origen hasta el final) ambos de la clase `Camera3D`
- El método `intersect_ray` devuelve información sobre el objeto colisionador (`StaticBody3D`) intersecado más cercano al origen del rayo (si hay intersección), incluyendo una referencia a dicho objeto.
- Para usar este método, es necesario obtener primero el objeto `PhysicsDirectSpaceState` de la escena. Esto se hace con la propiedad `direct_space_state` de la clase *singleton* `World3D`.

Ejemplo de cálculo de selección con un *click* (1/2)

En el siguiente ejemplo, se asocia a un nodo un script que responde a eventos de *click* calculando el objeto que se proyecta en el pixel donde se ha hecho *click*.

```
func _unhandled_input( event : InputEvent ):
   if event is InputEventMouseButton :
      if event.button_index == MOUSE_BUTTON_LEFT and event.pressed :

         ## 1. Buscar nodo de la cámara
         ## ....

         ## 2. Construir objeto con parámetros de intersección
         ## ....

         ## 3. Calcular la intersección
         ## ....

         ## 4. Procesar el resultado
         ## .....
```

Buscar cámara y construir objeto de parámetros

Para buscar el nodo de la cámara, suponemos (en este ejemplo) que dicho nodo es hermano del nodo que tiene el script (está en el mismo nivel del árbol de escena). Se puede hacer de muchas otras formas.

```
## 1. Buscar nodo de la cámara
var cam : Node = get_node_or_null("../CamaraOrbitalSimple")
assert( cam != null , "No encuento 'CamaraOrbitalSimple'")
```

A continuación se usa la posición del *click* (está en `event.position`) para construir un objeto *query* de la clase `PhysicsRayQueryParameters3D`, que contiene el rayo y los parámetros para el cálculo de la intersección:

```
## 2. Construir objeto con parámetros de intersección
var query  := PhysicsRayQueryParameters3D.new()
query.collision_mask = 0xFFFFFFFF  # todas las capas
query.from = cam.project_ray_origin( event.position )
query.to = query.from+100*cam.project_ray_normal( event.position )
```

Calcular las intersecciones y procesar resultado

Para calcular las intersecciones se invoca el método `intersect_ray()` del *space_state*:

```
## 3. Calcular las intersecciones
var space_state = get_world_3d().direct_space_state
var result = space_state.intersect_ray(query)
```

Finalmente, si hay alguna intersección, comprobamos el objeto padre del colisionador, si existe y tiene el método `cuando_click`, lo invocamos:

```
## 4. Procesar el resultado
if result:
   print("SÍ hay objeto en ese punto.")
   var padre : Node = result.collider.get_parent()
   if padre != null :
      print("Objeto padre intersecado: ", padre.name )
      if padre.has_method("cuando_click"): padre.cuando_click()
      else: print("Objeto padre no tiene 'cuando_click'")
   else: print("No hay objeto padre.")
else: print("NO hay objeto en ese punto.")
```

Comportamiento de objetos clickados

En los nodos con hijos con colisionadores se puede definir un método `cuando_click()` para definir el comportamiento al ser seleccionados:

```
var color1 = Color( 1.0, 0.5, 0.2 )
var color2 = Color( 0.2, 0.5, 1.0 )
var mat : StandardMaterial3D = null
var estado : bool = false  ## va alternando al ser clickado.

func _ready():  ## crear un material con 'color1':
   mat = StandardMaterial3D.new()
   mat.albedo_color = color1
   material_override = mat

func cuando_click():   ## cambiar el color al ser clickado:
   print("El DONUT ha sido clickado")
   estado = not estado
   if estado : mat.albedo_color = color2
   else:       mat.albedo_color = color1
   material_override = mat
```

Subsección **4.3.**

Problemas: selección por intersecciones

Problema: intersección rayo-triángulo (1/2)

Problema 9.2:

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 9.2 (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 $\mathbf{n}$ (el producto escalar de ambos es nulo).
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 9.3:

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 $\mathbf{x}_{\text{ec}}$,$\mathbf{y}_{\text{ec}}$ y $\mathbf{z}_{\text{ec}}$ con los versores y la tupla $\mathbf{o}_{\text{ec}}$ 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.

Fin de transparencias.