Informática Gráfica.
Sesión 5: Modelos Jerárquicos.

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

Índice

Sección **1.**

Modelos Jerárquicos

1. Grafos de escena.

Subsección **1.1.**

Grafos de escena.

Modelos jerárquicos.

En Informática Gráfica, un **Modelo Jerárquico** es una estructura de datos en forma de grafo que representa las relaciones espaciales entre las ***componentes*** de una aplicación interactiva.

- Es una herramienta fundamental que permite diseñar y gestionar modelos complejos mediante el uso de **componentes complejas constituidas de otras componentes más simples**
- Una ***componente*** es un objeto geométrico 2D o 3D sencillo, como una malla, o un grupo de varias componentes.
    - Permite la **reutilización** de componentes en distintas partes de un proyecto o en distintos proyectos.
- Permite a distintos desarrolladores (o al mismo desarrollador en distintos instantes) trabajar en diferentes componentes de un proyecto sin interferir entre sí, facilitando la colaboración en proyectos grandes.

Grafos de escena.

Un **Grafo de Escena** es un **Grafo Dirigido Acíclico**, cada uno de cuyos nodos contiene un modelo de un objeto. El grafo en sí es también un modelo (jerárquico) que representa un objeto compuesto (llamado **escena**) formado por **réplicas de los objetos cuyos modelos están en los nodos**.

- Un nodo N del grafo puede ser:

  Terminal: si objeto asociado a N no está compuesto de otros objetos, típicamente mallas con vértices en determinadas posiciones en el espacio.

  No terminal: si el objeto está compuesto de otros objetos en otros nodos del grafo, nodos llamados **nodos hijos** de N.
- En el grafo hay un **arco dirigido** desde cada nodo padre a cada uno de sus nodos hijos. Cada arco tiene asociada una **transformación geométrica** (afín).
- Todo nodo del grafo tiene al menos un padre (o más), excepto exactamente un nodo sin padres, que es el **nodo raíz**.

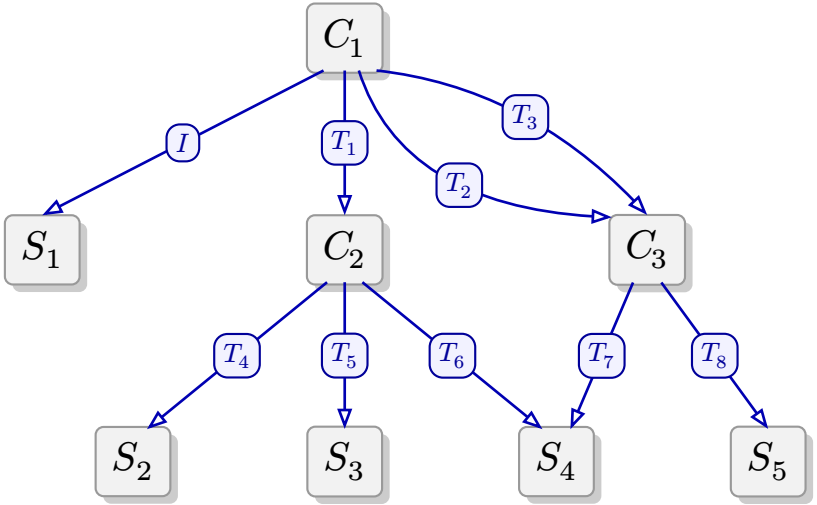
Marcos afines y transformaciones asociados a los nodos.

La escena asociada a un grafo está compuesta de ***objetos instanciados***.

- Un ***objeto instanciado*** es una réplica del objeto asociado a un nodo del grafo, pero con las coordenadas de sus vértices transformadas por una transformación afín propia de esa réplica.
- Un objeto puede aparecer instanciado varias veces en una escena, cada instancia puede tener asociada una transformación afín distinta.
- Las coordenadas de los vértices de un nodo son relativas a un marco afín propio de cada nodo, llamado **marco local** del nodo. A las coordenadas se les llama **coordenadas locales**.
- El marco local del nodo raíz se llama **marco de escena** y sus coordenadas se llaman **coordenadas de escena**.
- La transformación asociada a cada nodo determina como se convierten las coordenadas locales del nodo (relativas al marco del nodo) en coordenadas de escena (relativas al marco de escena).

Ejemplo de grafo de escena

En un grafo de escena, cada arco dirigido tiene asociada una transformación afín:



- cada objeto compuesto de otros es un **nodo no terminal** (C_1, C_2, C_3)
- cada objeto simple (no compuesto) es un **nodo terminal** (S_1, S_2, S_3, S_4, S_5)
- cada arco se etiqueta con una transformación geométrica ($I, T_1, T_2, ..., T_8$)

Instancias de objetos en el grafo de ejemplo

En la escena representada por un grafo **hay una instancia de cada nodo por cada camino posible desde la raíz al nodo**. Esa instancia tiene asociada la transformación que resulta de componer las transformaciones de los arcos del camino (de izquierda a derecha según se va desde la raíz al nodo).

A modo de ejemplo, en el grafo anterior hay las siguientes instancias de objetos terminales (no compuestos):

- Una instancia de S_1, con la transformación I (la transformación identidad).
- Una instancia de S_2, con la transformación $T_1 \circ T_4$.
- Una instancia de S_3, con la transformación $T_1 \circ T_5$.
- Tres instancias de S_4, con las transformaciones $T_1 \circ T_6$, $T_2 \circ T_7$ y $T_3 \circ T_7$.
- Dos instancias de S_5, con las transformaciones $T_2 \circ T_8$ y $T_3 \circ T_8$.

En total hay 8 instancias de objetos terminales. Además, si se permite que los nodos no terminales tengan mallas (además de hijos), habría que añadir 3 instancias de los objetos C_2 y C_3.

Sección **2.**

Grafos de escena en Godot.

1. Escenas y nodos.
2. Transformaciones de los nodos.
3. La transformación de los nodos 2D.
4. La transformación de los nodos 3D.
5. Creación y actualización de árboles en tiempo de ejecución.

Subsección **2.1.**

Escenas y nodos.

Proyectos, escenas y nodos en Godot

El desarrollo de aplicaciones gráficas en Godot se basa en los conceptos de **proyecto**, **escena** y **nodo**.

- Un **proyecto** es un conjunto de archivos que se usan para construir una aplicación. Se guardan en una *carpeta del proyecto* donde, entre otros, habrá un archivo `.godot` con datos del mismo.
- Una **escena** es un árbol de nodos, que se llama **árbol de escena**, con al menos un nodo. Una escena siempre tiene asociado un **nodo raíz** de la misma.
- Un proyecto incluye **una o varias escenas**. Una de las escenas será la **escena principal** del proyecto. En la carpeta del proyecto habrá un archivo `.tscn` por cada escena de dicho proyecto.
- Un **nodo** es un objeto (instancia de alguna clase Godot) que representa un elemento de la aplicación. Un nodo se incluye en una única escena, y aparece una sola vez en el árbol de dicha escena, por tanto un nodo tiene un único padre (excepto el nodo raíz de una escena).
- El nodo raíz de la escena principal se considera el nodo **raíz del proyecto**.

Clasificación de los nodos

Los nodos puede clasificarse en estas dos categorías:

- Nodos **regulares**: son objetos instancias de la clase `Node` o sus derivadas, principalmente:
  - `CanvasItem`: para objetos 2D, puede ser de la clase `Node2D` o derivadas (objetos visibles) o `Control` (objetos de interfaz gráfica).
  - `Node3D`: objetos que contienen, entre otras cosas, mallas 3D, cámaras, fuentes de luz etc...
- Nodos **instancia de escena**: guardan **una referencia a una escena del proyecto** (que no puede ser la escena principal del proyecto). Este nodo es único en el árbol, pero varios nodos de este tipo pueden tener referencias a la misma escena.

Así que la única forma de que un nodo pueda instanciarse más de una vez en un grafo es hacer que ese nodo sea una escena de Godot.

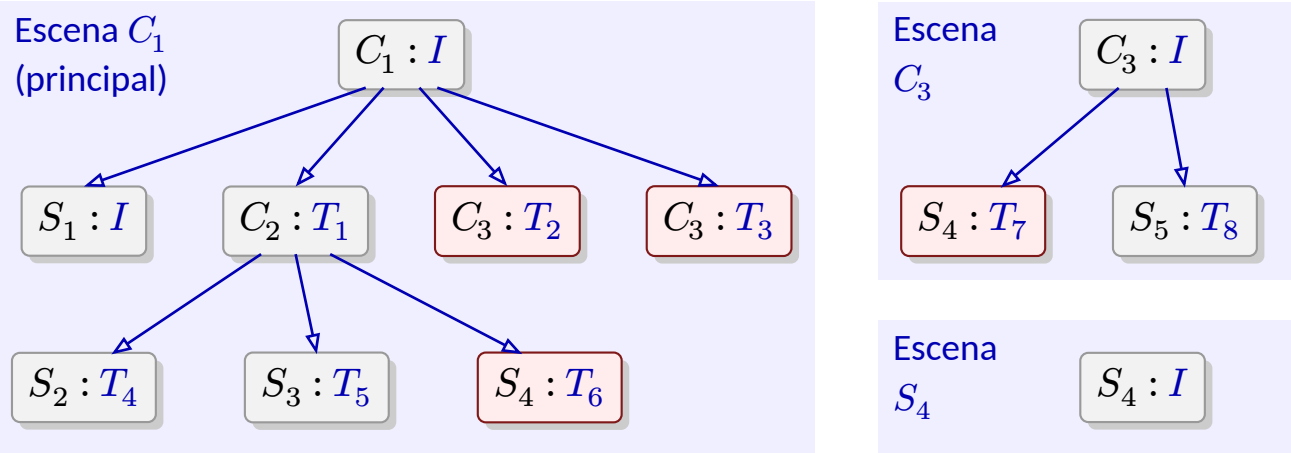
Instancias de nodos y escenas

En un árbol de escena:

- Cada nodo en un árbol de una escena tiene exactamente un padre (excepto el nodo raíz de una escena).
- Para cada escena (distinta de la escena principal del proyecto), puede haber varios nodos de tipo **instancia de escena** que referencien a esa escena, por tanto una escena puede estar:
  - instanciada en varios árboles de escena diferentes, o además
  - instanciada varias veces en un mismo árbol de escena.
- En el panel con el *árbol de escena* (arriba a la izquierda del editor), los nodos sub-escena aparecen como un nodo normal, pero con un icono de una plaqueta. No se pueden expandir.
- En el panel del *sistema de archivos* (abajo a la izquierda del editor), podemos ver un archivo de extensión `.tscn` que guarda información de una escena. Si lo seleccionamos, en el panel *árbol de escena* pasaremos a ver el árbol de escena de esa escena.

Esquema de varias escenas de un proyecto Godot

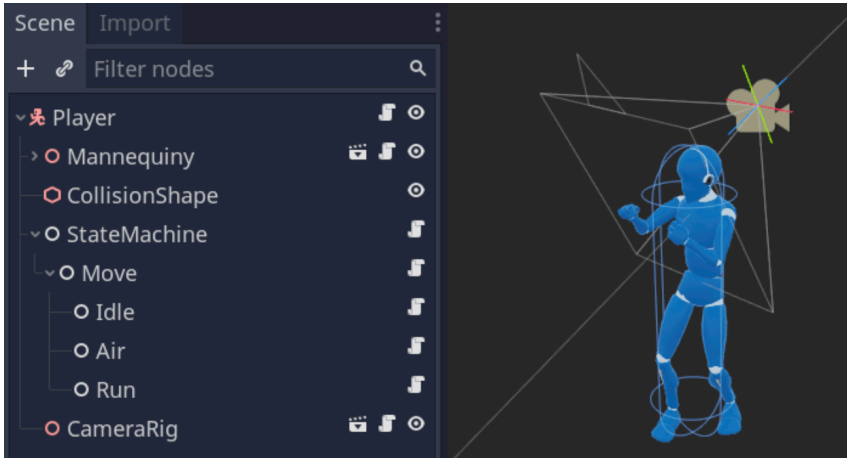
Vemos aquí las escenas equivalentes (en Godot) al grafo de ejemplo que ya vimos, pero ahora cada nodo tiene asociada una transformación (después de :) y los nodos instancia de escena tienen fondo rosa.



Hay tres escenas, cuyas raices son: C_1, C_3 y S_4. Godot ignora las transformaciones de los nodos raíz (ya que se redefinen en las instancias), excepto la transformación del nodo raíz de la escena principal.

Ejemplos de árboles de escena de Godot.

Aquí vemos dos ejemplos de dos árboles de escena de Godot.

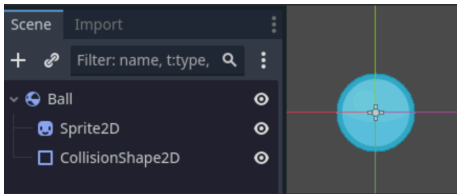


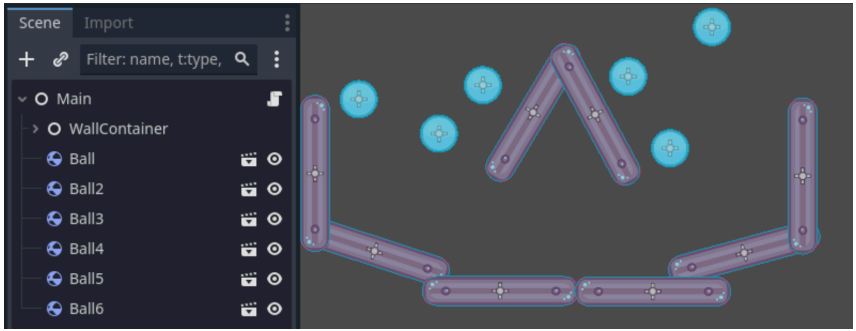
Imágenes obtenidas obtenidas de la documentación oficial de Godot:
docs.godotengine.org/es/4.5/getting_started/step_by_step/nodes_and_scenes.html

Escena instanciada en otra

A la izquierda vemos el disco azul que está modelado en la escena `Ball`, es un árbol formado por tres nodos.

A la derecha, los nodos `Ball`, `Ball2`, `Ball3`, etc.... son instancias de la escena `Ball`, cada una con su propia transformación (en concreto, están trasladadas a distintas posiciones del plano). En el panel aparecen con un icono de plaqueta.





Imágenes obtenidas de la documentación oficial de Godot:
docs.godotengine.org/es/4.5/getting_started/step_by_step/instancing.html

Reutilización de mallas

En las aplicaciones gráficas, los objetos tipo `Mesh` (mallas 2D o 3D) contienen los vértices y las coordenadas de textura, las normales, etc... y por tanto pueden ocupar mucha memoria. Por tanto:

- Se hace necesario evitar la duplicación de mallas en memoria, por ejemplo con dos nodos que incluyan la misma malla.
- Se pueden usar nodos de tipo `MeshInstance2D` o `MeshInstance3D` que contienen una referencia a una malla (propiedad `mesh`), pero no la malla en sí.
- En una aplicación puede haber distintos nodos referenciando la misma malla.
- Puesto que la clase `Mesh` es derivada de `RefCounted` (objetos con cuenta de referencias), en Godot se gestiona automáticamente la memoria de las mallas, siendo esta liberada cuando no hay más referencias a la misma.

Subsección **2.2.**

Transformaciones de los nodos.

El marco y la transformación asociados a un nodo.

Sea N un nodo (Node2D y Node3D) situado en un árbol de escena de Godot:

- El nodo N siempre asociado un marco afín (2D o 3D) que se llama **marco del nodo N**, lo llamamos $\mathcal{N}$
- El nodo N también tendrá asociado un **marco padre**, lo llamamos $\mathcal{P}$:
  - Si N no es el nodo raíz de una escena, $\mathcal{P}$ es el marco del único nodo padre de N en el árbol de esa escena.
  - Si N es nodo raíz de una escena, entonces $\mathcal{P}$ es el llamado **marco *global* de la escena**. Si esa escena es la escena principal de Godot, entonces a $\mathcal{P}$ también se le llama **marco del mundo**.
- La transformación que convierte el marco $\mathcal{P}$ en el marco $\mathcal{N}$ se representa mediante una matriz M_N llamada **transformación (o matriz) del nodo**, tiene en sus **columnas** la base y el origen del marco $\mathcal{N}$, **expresadas en coordenadas relativas al marco $\mathcal{P}$**. Esto implica que

$$\mathcal{P} M_N = \mathcal{N}$$

El espacio de coordenadas local y el espacio padre

En relación a las coordenadas de los vértices en las mallas u objetos guardados en un nodo N:

- Se considera que están siempre expresadas en el marco $\mathcal{N}$ del nodo, y se llaman **coordenadas locales de N**, también se dice que están expresadas en el **espacio de coordenadas local del nodo**.
- Al aplicarseles la transformación M_N, se convierten en coordenadas relativas al marco $\mathcal{P}$, se dice que esas coordenadas están expresadas en el **espacio de coordenadas padre de N**.
- En última instancia, todas las coordenadas de los vértices en una aplicación Godot acaban convertidas en la GPU en coordenadas relativas al marco global de la escena principal o marco del mundo, esas son **coordenadas de mundo**.
- Las coordenadas del mundo se usan por Godot para proyectar los vértices en pantalla y producir la imagen por rasterización en la GPU.

La propiedad *transform* de un nodo en Godot

Para cualquier nodo N, el objeto asociado tiene una propiedad de tipo `Transform2D` o `Transform3D`, llamada `transform`, que guarda la matriz asociada M_N.

La matriz de un nodo N se puede modificar asignando o actualizando su `transform`, se puede hacer de estas formas:

- En el editor, podemos cambiar el valor inicial de `transform` (le asignamos una posición, escalado o rotaciones).
- En tiempo de ejecución del videojuego, podemos modificar N.`transform` directamente en el código GDScript. Se puede hacer de dos formas:
  - Mediante asignaciones a N.`transform` o sus componentes.
  - Mediante asignaciones a propiedades relacionadas con `transform` (`position`, `rotation`, etc..., ver siguiente transparencia).
  - Usando métodos las clases `Node2D` o `Node3D` los cuales, aplicados a un nodo N, modifican N.`transform`.

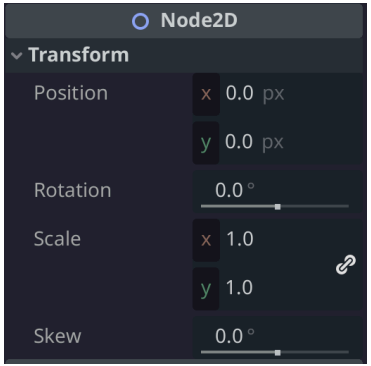
Subsección **2.3.**

La transformación de los nodos 2D.

Atributos de transformación de los nodos 2D

Para cada nodo, en el panel de la derecha del editor podemos ver las propiedades que definen su transformación inicialmente. Godot mantiene los atributos siempre coherentes con la matriz `transform` del nodo. Los atributos son:

`position`: vector 2D (traslación)
`rotation`: ángulo de rotación en radianes
`scale`: vector 2D (factores de escala)
`skew`: real, ángulo de *skew* en grados (tipo de cizalla)



La matriz `transform` se corresponde con la composición de estas transformaciones (en este orden de derecha a izquierda):

1. Escalado por los factores `scale`
2. *Skew* por el ángulo `skew`
3. Rotación de `rotation` radianes
4. Traslación por vector `position`

Actualización de la transformación 2D en un script

Las propiedades de transformación se pueden cambiar en tiempo de ejecución. El siguiente código siempre produce el mismo resultado, independientemente del valor de `asignar_transform` y de las variables declaradas en el código

```
var factores_escala     := Vector2( 2.0, 1.0 )
var angulo_rot_radianes := 1.0
var vector_traslacion   := Vector2( 1.0, 0.0 )

if asignar_transform :
  var esc := Transform2D().scaled( factores_escala )
  var ske := Transform2D() ## identidad (skew nulo)
  var rot := Transform2D().rotated( angulo_rot_radianes )
  var tra := Transform2D().translated( vector_traslacion )
  transform = tra * rot * ske * esc ## el orden es esencial
else:
  scale    = factores_escala     ## (1)
  skew     = 0.0                 ## (2)
  rotation = angulo_rot_radianes ## (3)
  position = vector_traslacion   ## (4)
```

Modificación de la transformación de un nodo 2D.

Estos métodos, aplicados a un nodo N, modifican las propiedades de transformación, y después se recalcula la matriz `transform` del nodo.

- N.`rotate(`θ`)`: añade θ radianes a la propiedad `rotation`, es decir, es equivalente a:

```
rotation += theta
```

- N.`apply_scale(`$\mathbf{s}$`)`: multiplica los factores en la propiedad `scale` por los dos factores en $\mathbf{s}$.

```
scale = Vector2( scale.x * s.x, scale.y * s.y )
```

- N.`translate(`$\mathbf{t}$`)`: suma el vector $\mathbf{t}$ a la propiedad `position`.

```
position += vt
```

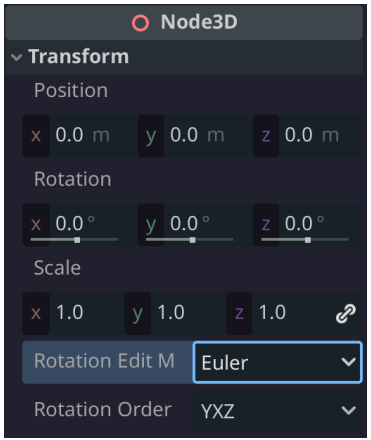
Subsección **2.4.**

La transformación de los nodos 3D.

Atributos de transformación de los nodos 3D

Godot también tiene propiedades que definen la transformación de un nodo 3D. En el caso 3D no hay *skew* y además la rotación es la composición de 3 rotaciones entorno a los ejes en el orden Y, X, Z (aunque se puede cambiar el orden, o usar un cuaternión, o dar directamente los ejes transformados).

`position`: vector 3D (traslación)
`rotation`: vector con tres ángulos de rotación (Y, X, Z)
`scale`: vector 3D (3 factores de escala)



La matriz `transform` se corresponde con la composición de estas transformaciones (en este orden de derecha a izquierda):

1. Escalado por los factores `scale`
2. Rotaciones elementales con los ángulos contenidos en `rotation` (en radianes).
3. Traslación por vector `position`

Actualización de la transformación: rotaciones por la izquierda.

Al igual que en 2D, en 3D existen diversos métodos que permiten modificar las propiedades `rotation`, `scale` de un nodo 3D. En estos casos se componen matrices de rotación por la izquierda de la matriz actual.

- N.`rotate(`$\mathbf{v}$`,`θ`)`: compone la rotación codificada en `rotation` con una rotación entorno al eje $\mathbf{v}$ (del padre). Si `position` es el vector nulo, entonces es equivalente a:

```
var rot := Transform3D().rotated( v, theta )
N.transform = rot * N.transform
```

- N.`rotate_x(`θ`)`: idem, donde $\mathbf{v}$ es el eje X.
- N.`rotate_y(`θ`)`: idem, eje Y.
- N.`rotate_z(`θ`)`: idem, eje Z.

Al usarse composición por la izquierda, el vector 3D $\mathbf{v}$ con el eje de rotación se interpreta como expresado en **el marco de coordenadas del padre.**

Actualización de la transformación: composición por la derecha.

Existen otros métodos que permite componer matrices por la derecha, es decir, la matri que se compone actúa sobre las coordenadas **antes** que la transformación previa, es decir, se aplica a las coordenadas locales en primer lugar. Equivalente a:

```
var M : Transform3D = ...  ## la matriz que se quiere componer
N.transform = N.transform * M
```

- N.`rotate_object_local(`$\mathbf{v}$`,`θ`)`: $M :=$ rotación de θ entorno al eje $\mathbf{v}$
- N.`translate_object_local(`$\mathbf{t}$`)`: $M :=$ traslación por el vector $\mathbf{t}$
- N.`translate(`$\mathbf{t}$`)`: equivalente a la anterior.
- N.`scale_object_local(`$\mathbf{s}$`)`: $M :=$ matriz de escalado con factores $\mathbf{s}$.

Al usarse composición por la derecha, eso implica que las tuplas $\mathbf{v}$,$\mathbf{t}$ y $\mathbf{s}$ (de tipo `Vector3`) que se indican aquí se interpretan como expresadas en el **marco de coordenadas local del nodo** N.

Subsección **2.5.**

Creación y actualización de árboles en tiempo de ejecución.

Creación, inserción y consulta de nodos en un árbol.

Los nodos se pueden crear en tiempo de ejecución con el método de clase `new()`. Si el constructor tuviese parámetros, habría que proporcionarlos como argumentos de `new()`. Por ejemplo, para crear un nodo de tipo `Node2D`:

```
var nodo := Node2D.new() ## al crearlo está "huérfano"
```

Para añadir un nodo (por ejemplo `hijo`) al árbol de escena, habría que usar el método `add_child()` del nodo padre (nodo `padre`). Lo añade como último nodo hijo. Por ejemplo:

```
padre.add_child( hijo )
```

Los hijos directos de un nodo padre p se pueden consultar con estos métodos:

- p`.get_child_count()`: devuelve el número de hijos del nodo.
- p`.get_child(`i`)`: devuelve el hijo número i (entero, empezando en 0).
- p`.get_children()`: devuelve un `Array` con todos los hijos del nodo.

Nombres de los nodos y búsqueda

Todo nodo tiene una propiedad `name` (cadena de caracteres, `String`) que se puede asignar y consultar en el editor o en tiempo de ejecución con scripts. Sirve para **identificar un nodo en el árbol**, y debe ser único entre los nodos hijos de un mismo padre. Los métodos para buscar nodos en un árbol son:

- p`.get_node(`s`)`: devuelve un nodo descendiente (directo o indirecto) del nodo p, siguiendo la ruta (*path*) dada en la cadena s. La ruta tiene nombres de nodos separados por `/`, empezando por un hijo de p. Si no se encuentra da error. Aquí se obtiene el nodo de nombre «*bisnieto*»:

```
var nodo := n.get_node("hijo/nieto/bisnieto")
var nodo := $hijo/nieto/bisnieto ## forma equivalente.
```

- p`.get_node_or_null(`s`)`: igual que el anterior, pero si no encuentra nada devuelve `null` en vez de dar error.
- p`.find_node(`s`)`: igual que el anterior, pero permite caracteres comodín `*` y `?` en s y devuelve el primer nodo que encaja con la cadena, en un recorrido en profundidad del árbol. Si no encuentra nada, devuelve `null`.

Desconexión y destrucción de nodos

Se puede **desconectar un nodo hijo** h de su nodo padre p con el método p`.remove_child(`h`)`. Esto lo elimina del árbol, pero no se destruye el nodo (aunque queda *huérfano* con seguridad, ya que todo nodo tiene un único padre, y no será visible en la escena).

Para **destruir** un nodo huérfano (es decir, liberar la memoria que ocupa ese nodo y todos sus hijos), se usa el método n`.queue_free()`. Esto registra la solicitud de eliminar el nodo, lo cual ocurrirá al final del siguiente frame.

A modo de ejemplo, para desconectar y destruir un nodo hijo (con nombre «*borrar*») de un nodo padre `p`, haríamos:

```
var h := p.get_node("borrar")
p.remove_child( h ) ## queda huérfano
h.queue_free()      ## se puede usar hasta el final del siguiente frame.
```

Objetos de tipo *Mesh* compartidos

Como se ha indicado, en un árbol es conveniente **no replicar objetos `Mesh` complejos**. A modo de ejemplo, este código crea un `ArrayMesh` y dos `MeshInstance3D` que lo comparten. Cada `MeshInstance3D` está instanciado en una posición distinta:

```
var tablas := []
tablas.resize( Mesh.ARRAY_MAX )
GenerarTablas( tablas ) ## función que genera vértices, normales, etc...
var am := ArrayMesh.new()
am.add_surface_from_arrays( Mesh.PRIMITIVE_TRIANGLES, tablas )

var instancia1 := MeshInstance3D.new()
instancia1.mesh     = am
instancia1.position = Vector3( -3.0, 0.0, 0.0 )

var instancia2 := MeshInstance3D.new()
instancia2.mesh     = am
instancia2.position = Vector3( +3.0, 0.0, 0.0 )
```

Sección **3.**

Ejemplo de un árbol 2D

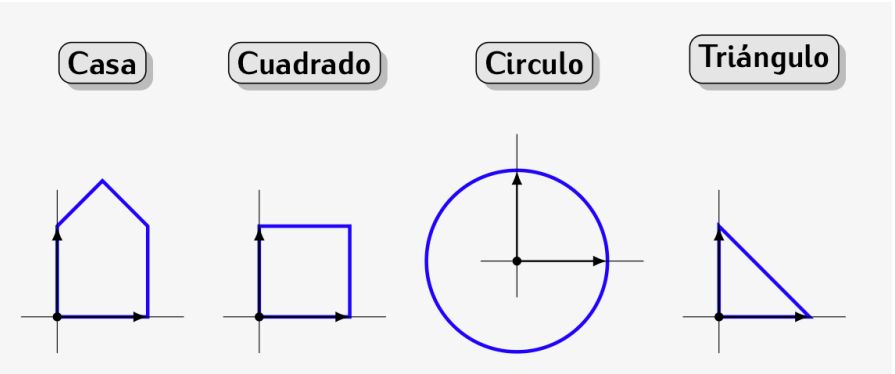
1. Diseño del grafo.
2. Implementación en Godot.
3. Implementación de Grafos parametrizados (y animados).

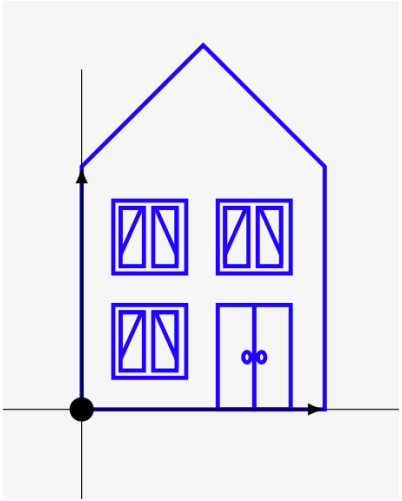
Subsección **3.1.**

Diseño del grafo.

Objetos simples

Supongamos que queremos construir una figura como la de la derecha, usando varios objeto `ArrayMesh` simples (un cuadrado, un triángulo y un círculo), tal y como aparecen a la izquierda.





Cada objeto aparece junto a su marco de referencia local, de forma que podamos entender mejor las transformaciones.

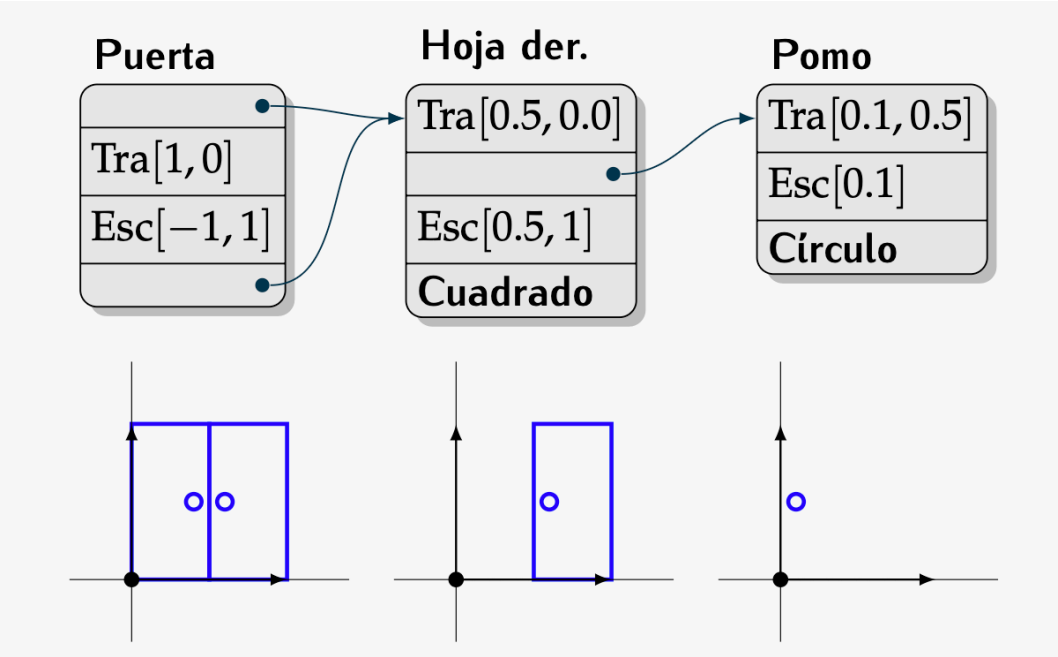
Notación abreviada para grafos de escena

Para este diseño, usaremos una notación más expresiva para especificar el grafo de escena:

- Usaremos un grafo dirigido acíclico, en vez de un simple árbol.
- Los nodos son listas de:
  - Elementos simples (como los de la transparencia anterior), en negrita.
  - Instancias de otros nodos (subárboles), con una flecha a otro nodo.
  - Transformaciones: afectando a todas las entradas de la lista que le siguen en el nodo (y se aplican de abajo hacia arriba).
- La implementación de estos grafos en Godot requiere convertirlos en árboles, bien duplicando nodos, bien usando instancias de escenas.
- En nuestro caso, usaremos nodos duplicados, aunque compartirán objetos `ArrayMesh`.

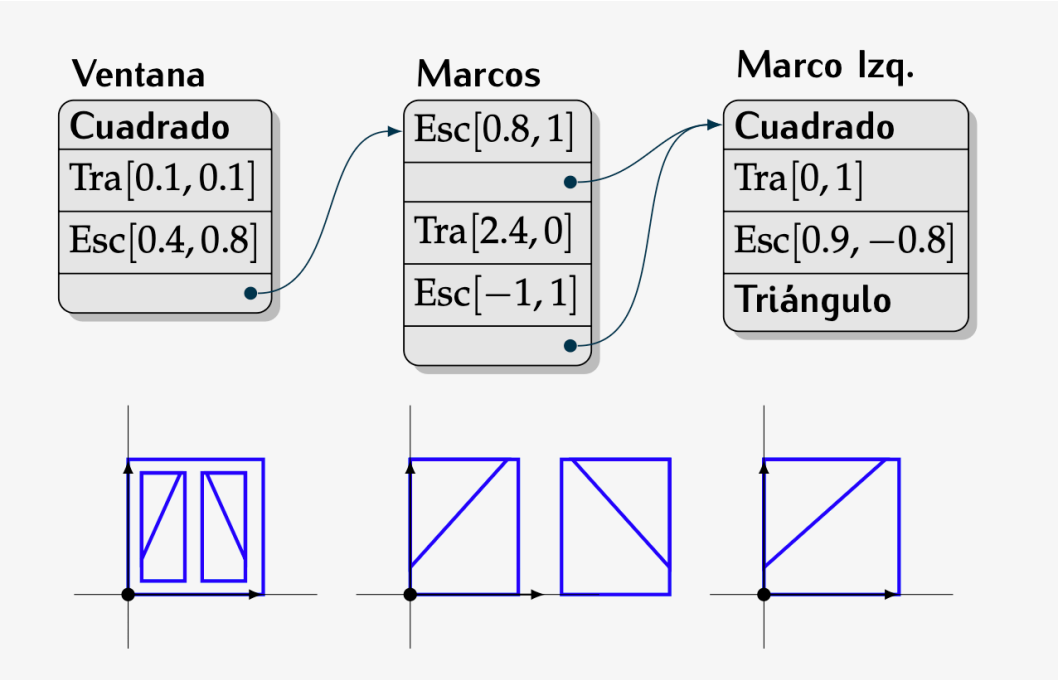
Objetos puerta, hoja derecha y pomo.

El nodo `Puerta`, contiene dos instancias de un nodo llamado `HojaDerecha`, que a su vez tiene un objeto `Pomo`



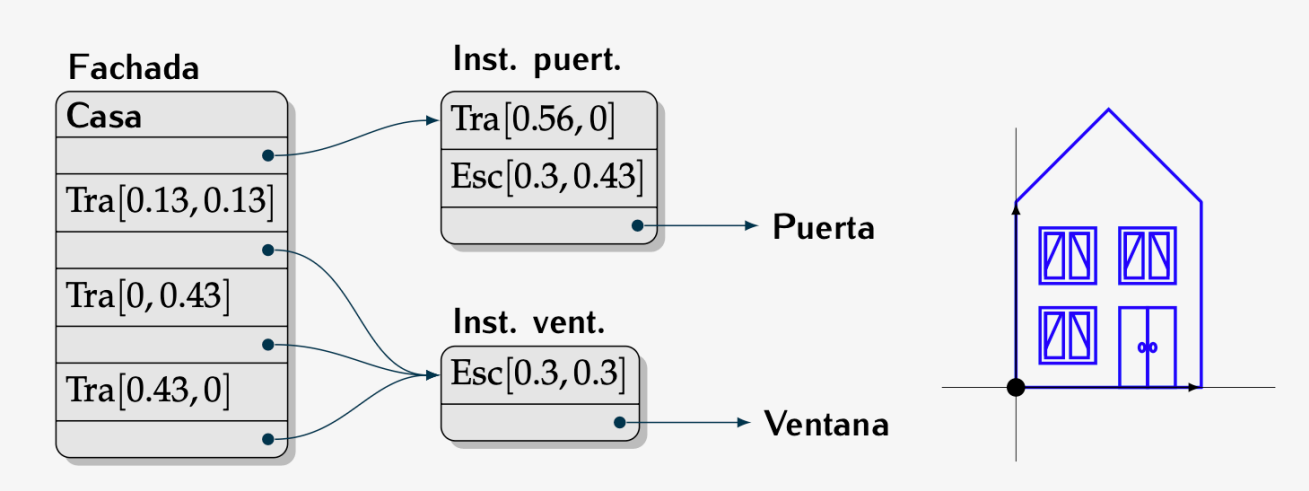
Objetos ventana, marcos y marco izquierdo

Objetos `Ventana`, compuesto de un **Cuadrado** y un objeto `Marcos`, a su vez compuesto de un `MarcoIzquierdo` (que es un **Cuadrado** y un **Triángulo**):



Objeto fachada, instancias de puertas y de ventana

Objeto `Fachada`, que es la raíz del árbol de escena. Contiene los objetos: `Casa`, `InstanciaPuerta` e `InstanciaVentana` (que a su vez incluyen `Puerta` y `Ventana`, respectivamente).



Subsección **3.2.**

Implementación en Godot.

Estrategias de implementación.

El diseño del grafo de escena puede ser implementado en Godot de diversas formas:

- Creando los nodos en el editor, sin scripts, excepto para crear los objetos `ArrayMesh`
- Creación en tiempo de ejecución con uno o varios scripts.
- Combinación de ambos: algunos nodos creados en el editor, y otros en tiempo de ejecución.

En nuestro caso, usaremos un **único script asociado al nodo raíz**. Dicho nodo raíz se puede crear en el editor, y después, en tiempo de ejecución, en la función `_ready()`, se crean todos los nodos descendientes.

Funciones auxiliares.

Esta función crea un objeto **ArrayMesh** a partir de un array de posiciones de vértices (con tipo de primitiva polilínea abierta, **PRIMITIVE_LINE_STRIP**):

```
func CrearArrayMesh( v : PackedVector2Array ) -> ArrayMesh :
    var tablas : Array = [] ; tablas.resize( Mesh.ARRAY_MAX )
    tablas[ Mesh.ARRAY_VERTEX ] = v
    var am : ArrayMesh = ArrayMesh.new()
    am.add_surface_from_arrays( Mesh.PRIMITIVE_LINE_STRIP, tablas )
    return am
```

Esta función crea un nodo **MeshInstance2D** a partir de un **ArrayMesh** dado:

```
func CrearMeshInstance2D( am : ArrayMesh, tr : Transform2D ) ->
                                                    MeshInstance2D:
    var mi := MeshInstance2D.new()
    mi.transform = tr
    mi.modulate  = Color( 0.0, 0.0, 0.7 ) ## azul
    mi.mesh      = am
    return mi
```

Objeto simples: cuadrado, triángulo, casa

Los objetos simples se puede crear como variables únicas, de tipo **ArrayMesh**, de forma que se compartan entre todas las instancias que los usen.

```
var cuadrado := CrearArrayMesh( PackedVector2Array([
  Vector2( 0.0, 0.0 ), Vector2( 1.0, 0.0 ),
  Vector2( 1.0, 1.0 ), Vector2( 0.0, 1.0 ),
  Vector2( 0.0, 0.0 )
]))

var triangulo := CrearArrayMesh( PackedVector2Array([
  Vector2( 0.0, 0.0 ), Vector2( 1.0, 0.0 ), Vector2( 0.0, 1.0 ),
  Vector2( 0.0, 0.0 )
]))

var casa := CrearArrayMesh( PackedVector2Array([
  Vector2( 0.0, 0.0 ), Vector2( 1.0, 0.0 ),
  Vector2( 1.0, 1.0 ), Vector2( 0.5, 1.4), Vector2( 0.0, 1.0 ),
  Vector2( 0.0, 0.0 )
]))
```

Objetos simples: circunferencia

La variable correspondiente a la circunferencia se puede crear así

```
var circunferencia : ArrayMesh  = CrearCircunferencia( 64 )
```

Se usa una función auxiliar que genera los vértices de una circunferencia de radio unidad, con el número de segmentos indicado:

```
func CrearCircunferencia( n : int ) -> ArrayMesh :
    var v := PackedVector2Array()
    for i in range( n+1 ):
      var a : float = (float(i) * 2.0 * PI )/ float(n)
      v.append( Vector2( cos(a), sin(a) ) )
    return CrearArrayMesh( v )
```

Otra función auxiliar útil compone una transformación por la izquierda:

```
func TransformaNode2D( n : Node2D, tr : Transform2D ) -> Node2D :
   n.transform = tr * n.transform
   return n
```

Funciones para el pomo, hoja derecha y puerta

```
func Pomo() -> Node2D :
  var tra = Transform2D().translated( Vector2( 0.1, 0.5 ))
  var esc = Transform2D().scaled( Vector2( 0.06, 0.06 ))
  return CrearMeshInstance2D( circunferencia, tra*esc )

func HojaDer() -> Node2D :
  var tra = Transform2D().translated( Vector2( 0.5, 0.0 ))
  var esc = Transform2D().scaled( Vector2( 0.5, 1.0 ))
  var n = Node2D.new()
  n.add_child( TransformaNode2D( Pomo(), tra ))
  n.add_child( CrearMeshInstance2D( cuadrado, tra*esc ))
  return n

func Puerta() -> Node2D :
  var tra = Transform2D().translated( Vector2( 1.0, 0.0 ))
  var esc = Transform2D().scaled( Vector2( -1.0, 1.0 ))
  var n = Node2D.new()
  n.add_child( HojaDer())
  n.add_child( TransformaNode2D( HojaDer(), tra*esc ))
  return n
```

Marco izquierdo y marcos

```
func MarcoIzq() -> Node2D :
  var tra = Transform2D().translated( Vector2( 0.0, 1.0 ))
  var esc = Transform2D().scaled( Vector2( 0.9, -0.8 ))
  var n = Node2D.new()
  n.add_child( CrearMeshInstance2D( cuadrado, tr_identidad ))
  n.add_child( CrearMeshInstance2D( triangulo, tra*esc))
  return n

func Marcos() -> Node2D :
  var esc1 = Transform2D().scaled( Vector2( 0.8, 1.0 ))
  var tra  = Transform2D().translated( Vector2( 2.4, 0.0 ))
  var esc2 = Transform2D().scaled( Vector2( -1.0, 1.0 ))
  var n = Node2D.new()
  n.add_child( TransformaNode2D( MarcoIzq(), esc1 ))
  n.add_child( TransformaNode2D( MarcoIzq(), esc1*tra*esc2 ))
  return n
```

Ventana, instancia puerta e instancia ventana

```
func Ventana() -> Node2D :
  var tra  = Transform2D().translated( Vector2( 0.1, 0.1 ))
  var esc  = Transform2D().scaled( Vector2( 0.4, 0.8 ))
  var n = Node2D.new()
  n.add_child( CrearMeshInstance2D( cuadrado, tr_identidad ))
  n.add_child( TransformaNode2D( Marcos(), tra*esc ))
  return n

func InstPuerta() -> Node2D :
  var tra = Transform2D().translated( Vector2( 0.56, 0.0 ))
  var esc = Transform2D().scaled( Vector2( 0.3, 0.43 ))
  var n = Node2D.new()
  n.add_child( TransformaNode2D( Puerta(), tra*esc ))
  return n

func InstVentana() -> Node2D :
  var esc = Transform2D().scaled( Vector2( 0.3, 0.3 ))
  var n = Node2D.new()
  n.add_child( TransformaNode2D( Ventana(), esc ))
  return n
```

Fachada

```
func Fachada() -> Node2D :

  var tra1 = Transform2D().translated( Vector2( 0.13, 0.13 ))
  var tra2 = Transform2D().translated( Vector2( 0.00, 0.43 ))
  var tra3 = Transform2D().translated( Vector2( 0.43, 0.00 ))

  var n = Node2D.new()
  n.add_child( CrearMeshInstance2D( casa, tr_identidad ))
  n.add_child( TransformaNode2D( InstPuerta(), tr_identidad ))
  n.add_child( TransformaNode2D( InstVentana(), tra1 ))
  n.add_child( TransformaNode2D( InstVentana(), tra1*tra2 ))
  n.add_child( TransformaNode2D( InstVentana(), tra1*tra2*tra3 ))

  return n
```

Subsección **3.3.**

Implementación de Grafos parametrizados (y animados).

Diseño de grafos parametrizados

A veces los grafos de escena pueden depender de uno o varios **parámetros** o **grados de libertad**.

Son uno o varios valores reales (o vectores), de forma que cada uno de ellos **determina una o varias transformaciones de un modelo jerárquico**. Esto permite:

Animar de forma sencilla un modelo jerárquico: haciendo depender los valores de los parámetros del tiempo real transcurrido durante la ejecución.

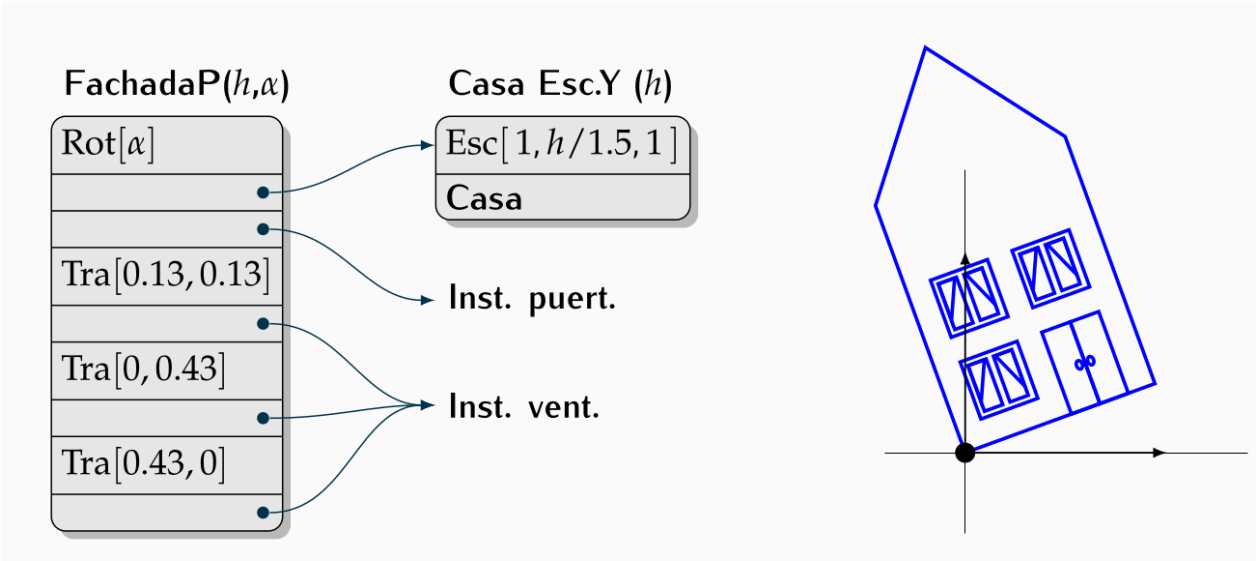
Generar múltiples variantes de un diseño: ya que podemos construir nodos o sub-árboles similares, que solo difieren en los valores de los parámetros.

Modificar interactivamente un modelo en tiempo de ejecución: ya que se puede permitir al usuario usar controles o eventos de entrada para modificar los parámetros.

A modo de ejemplo, en un modelo de un coche, puede existir un parámetro que sea un ángulo de giro de las cuatro ruedas. Es un real en radianes que determina la transformación de rotación.

Ejemplo de diseño de un grafo parametrizado

En el ejemplo anterior, podemos hacer que el grafo dependa de dos parámetros, α (rotación de la figura completa, en radianes) y h (altura de la fachada, real positivo):



Variación lineal u oscilante

Para animar un modelo jerárquico, los valores de los parámetros se pueden variar en cada frame y recalculando las transformaciones de los nodos. Así se consigue animación o interacción. Hay muchas opciones, dos bastante sencillas y útiles son:

- Hacer que un parámetro **varíe linealmente con el tiempo**, por ejemplo, un ángulo de rotación r que aumenta a ritmo constante por cada segundo:

$$r \quad = \quad r_0 + 2\pi \cdot w \cdot t,$$

  donde t es el tiempo transcurrido de animación en segundos, w es el número de vueltas por segundo, y r_0 el valor de r al inicio (cuando $t = 0$).
- Para escalados o desplazamientos queremos evitar que crezcan indefinidamente (no suele tener sentido), así que hacemos que v **oscile entre otros dos valores** a y b, y lo haga w veces por segundo:

$$v \quad = \quad a + (b - a) \cdot \frac{1 + \sin(2\pi \cdot w \cdot t)}{2}$$

Implementación de grafos parametrizados en Godot

En este ejemplo modificamos las transformaciones de dos nodos **n1** y **n2**

```
var a    := .... ## valor mínimo (e incial) del parámetro 2 (oscilante)
var b    := .... ## valor máximo del parámetro 2 (oscilante)
var w1   := .... ## ciclos o vueltas por segundo (parámetro 1)
var w2   := .... ## ciclos o vueltas por segundo (parámetro 2)
var ang2 := 0.0  ## valor acumulado de 2*PI*w2*t (argumento de 'sin')

func _process( delta : float ) :
  if animacion_activada :

    ## actualizar transformación del nodo 'n1':
    n1.rotate( 2*PI*w1*delta ) ## simplemente acumulamos rotación

    ## actualizar transformación del nodo 'n2':
    ## (se calcula un factor de escala 'fe' oscilante)
    ang2 += 2.0 * PI * w2 * delta ## acumulamos en 'ang2'
    var fe : float = a + (b-a)*(1.0 + sin( ang2 ))/2.0
    n2.transform = Transform2D().scaled( Vector2( fe, 1.0) )
```

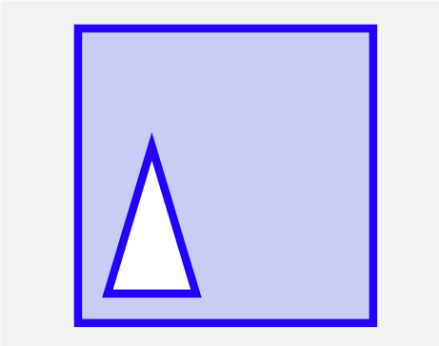
Sección **4.**

Problemas

Escena simple

Problema 5.1:

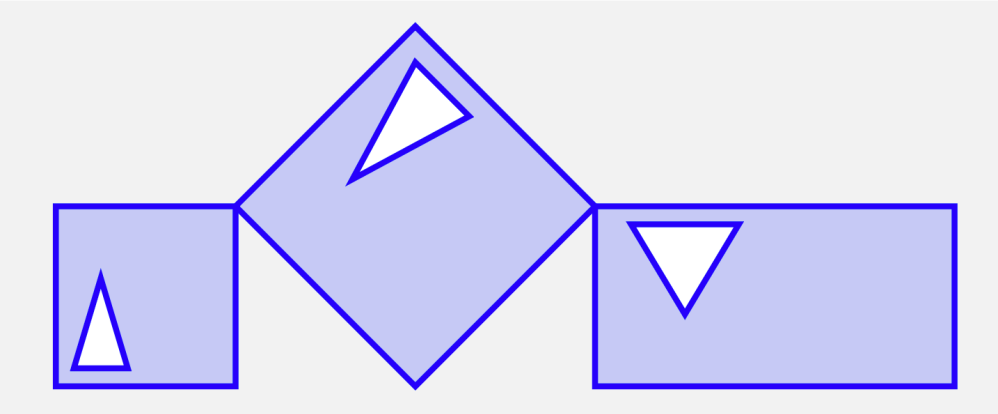
Implementa un proyecto cuya escena principal tenga un de tipo `Node2D` con varios nodos hijos, que formen la figura con un cuadrado de lado 2, centrado en el origen, y con un triángulo inscrito. El cuadrado debe estar relleno de azul claro, el triángulo de blanco, y las aristas deben verse de color azul oscuro.



Proyecto con dos escenas.

Problema 5.2:

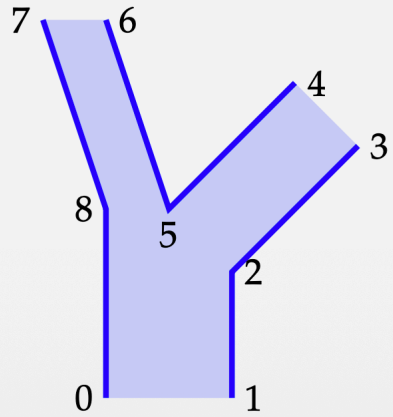
Crea un proyecto Godot con una escena principal con un nodo raíz compuesto. Ese nodo tendrá tres hijos, cada uno es una instancia de la escena del problema anterior, pero con una transformación distinta.



Escena simple

Problema 5.3:

Implementa un proyecto Godot con una función `Tronco`que crea y devuelve un `Node2D` con dos nodos hijos que forman la figura de aquí abajo (uno para el relleno y otro para las aristas).

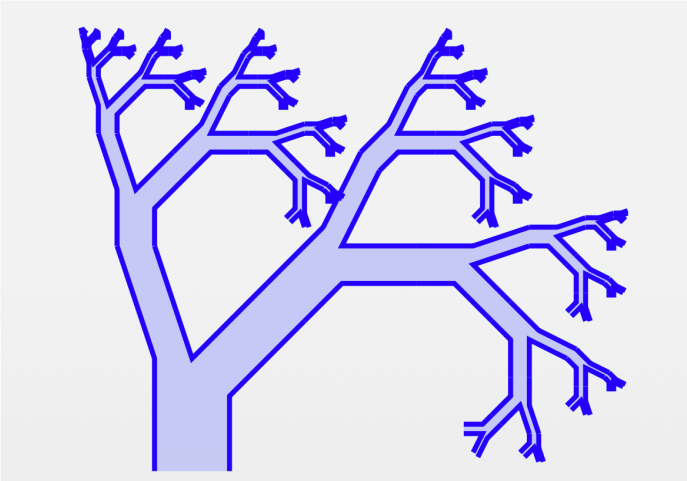


Índice	Coordenadas
0	$(+0.0, +0.0)$
1	$(+1.0, +0.0)$
2	$(+1.0, +1.0)$
3	$(+2.0, +2.0)$
4	$(+1.5, +2.5)$
5	$(+0.5, +1.5)$
6	$(+0.0, +3.0)$
7	$(-0.5, +3.0)$
8	$(+0.0, +1.5)$

Figura recursiva

Problema 5.4:

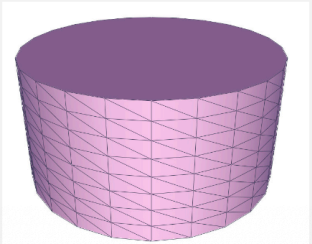
Implementa otro proyecto Godot que use la función del problema anterior para otra función, `Arbol(n)` que genera un árbol de escena con la figura de aquí abajo, que incluye múltiples instancias de `Tronco`, situadas recursivamente unas adyacentes a otras, hasta un nivel de recursividad dado por `n`.



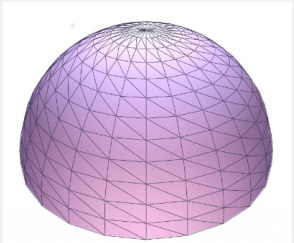
Árbol de escena 3D

Problema 5.5:

En un proyecto Godot 3D (puedes usar la práctica 2) para crear una figura como el logo de Android, usando únicamente dos objetos `ArrayMesh`, uno con un cilindro y otro con una semiesfera.



`Cilindro`



`Semiesfera`

`Android`

Fin de transparencias.