En el desarrollo de mundos virtuales es casi imprescindible saber cuando dos objetos se superponen o entran en contacto. Esto es conocido como detección de colisiones. Cuando una colisión es detectada, normalmente es deseable que algo suceda, esto es llamado respuesta a la colisión.

Godot ofrece varios objetos de colisión para proveer tanto detección como respuesta a una colisión. Tratar de decidir cuál utilizar para tu práctica puede ser confuso. Puedes evitar problemas y simplificar el desarrollo si entiendes como funciona cada uno y cuáles son sus ventajas y desventajas.

Vamos a comenzar con los objetos que dependen de la física, y luego seguiremos con los que dependen tan solo de las colisiones y los eventos.

1. Nodos dependientes de la física

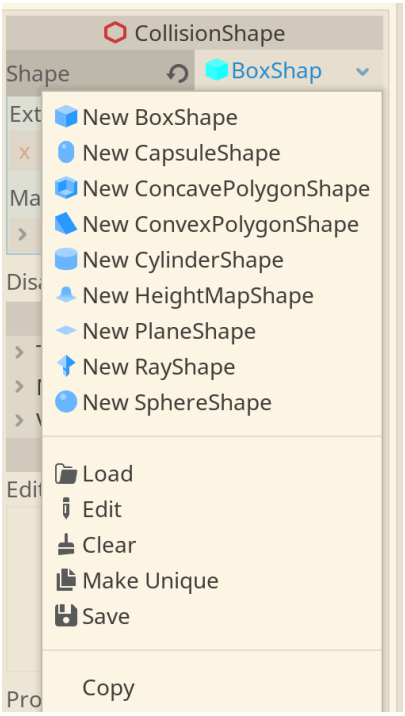
Existen cuatro tipos de objetos físicos que extienden del **CollisionObject**:

- **StaticBody:** Un cuerpo estático es uno que no se mueve por el motor de física. Participa en detección de colisiones pero no se inmuta en respuesta a la colisión. Se usan en objetos que forman parte del entorno y no necesitan ningún comportamiento dinámico.
- **RigidBody:** Este es el nodo que implementa simulación de física. No se puede controlar directamente, sino que puedes aplicar fuerzas (gravedad, impulsos, etc.) y el motor de física calcula el movimiento resultante.
- **KinematicBody:** Es un cuerpo que provee detección de colisiones, pero no de física. Todo movimiento y respuestas a colisiones debe ser implementado en código.
- **PhysicalBone:** Este nodo se usa para la respuesta física de personajes cuando la animación se desactiva. Si has visto lo que pasa con los cadáveres de los *NPC* enemigos en juegos triple A entenderás bien para que sirve.

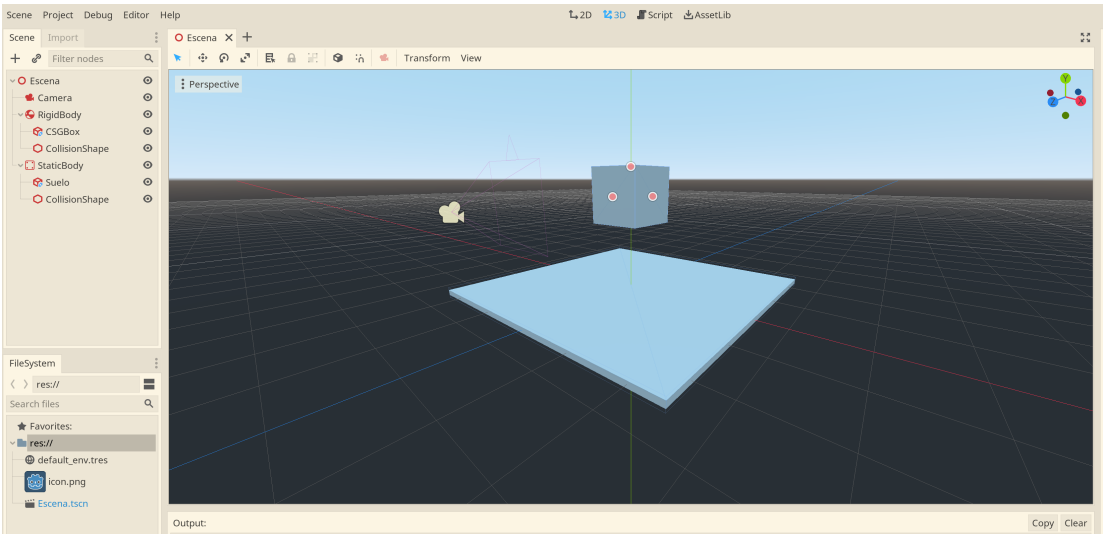
Un cuerpo físico puede contener cuaquier número de objetos **CollisionShape** como hijos. Estas formas son usadas para definir los límites de colisión y detectar contactos con otros objetos físicos.

La mejor forma de entender los **StaticBody** y los **RigidBody** es hacer un ejemplo. Crea un nodo de cada una de estas clases. Añade un **CollisionShape** a cada uno de los nodos como hijos.

Verás que en las opciones de **CollisionShape** puedes poner una figura simple:



Vamos a elegir una caja (**BoxShape**), y cambiar sus parámetros. La del cuerpo estático hará de suelo, mientras que el rígido hará de caja. Puedes añadir también nodos **CSGBoxes** para que se pueda visualizar algo lo más ajustado posible al sistema de colisiones. Fíjate en la imagen.

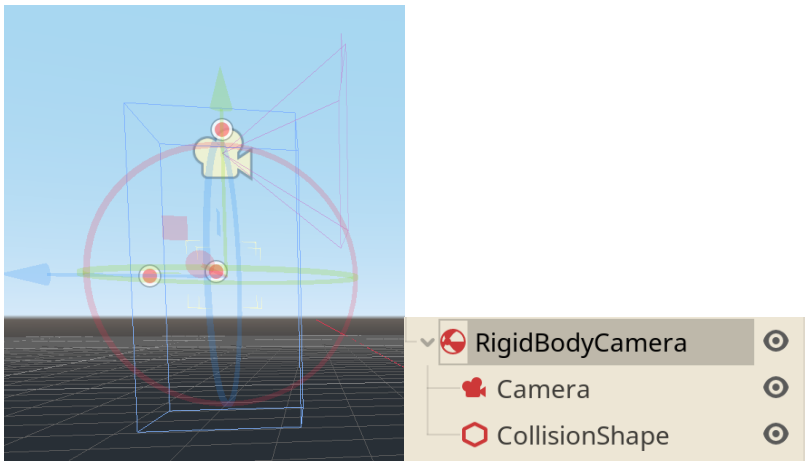


Con respecto a los cuerpos de tipo **Kinematic**, estos están pensados para ser controlados por el usuario. ¡Pero cuidado! No se van a ver afectados por la física (no esperes que tu personaje rebote contra una pared al chocar contra ella), en realidad son muy parecidos a los objetos estáticos.

Por ahora vamos ha crear una cámara basados en rígidos, y luego pasaremos al de tipo **Kinematic**.

Pero primero crea unas paredes al entorno para que puedas probar bien el mismo (recuerda que tengan física **static**).

Ahora vamos a construir la cámara. Añade un objeto rígido y otro de colisión (un cubo), la cámara heredará del rígido. El cubo de colisión tendrá el tamaño de una persona, la cámara hará de cabeza, fíjate en la figura.



Si la colocas a cierta altura verás como caes hacia el suelo, y te detienes al chocar con él.

Añadamos un *script* en el nodo **RigidBodyCamera** (el nodo de tipo **RigidBody** del que hereda la cámara).

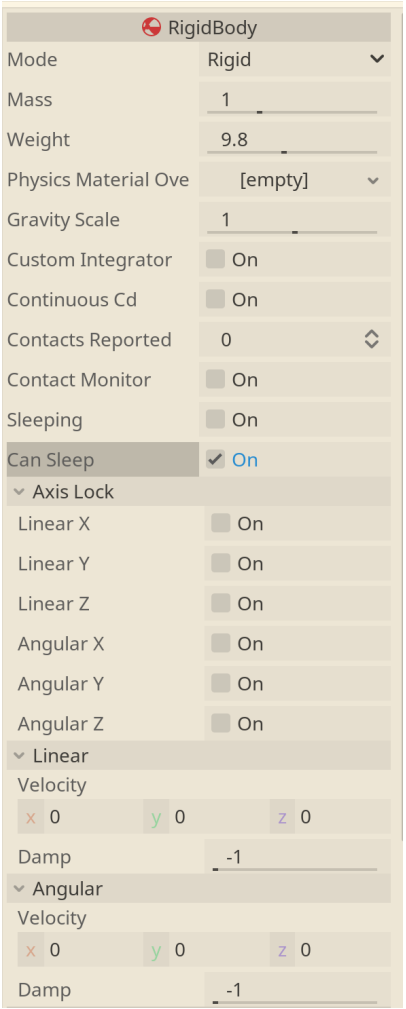
Mira los métodos del objeto **RigidBody**, podemos aplicar fuerzas lineales (add_force) o fuerzas de giro (add_torque).

Aplica una fuerza hacia delante cuando pulses una tecla:

```
add_force(Vector3(0, 0, -10), Vector3(0, 0, 0))
```

¿qué sucede? Mira la ayuda, a ver si puedes evitar que caíga, y a la vez que se mueva.

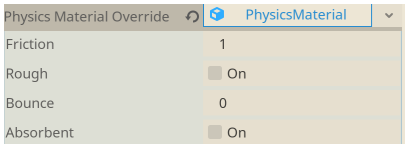
En cualquier caso hay más cosas, los objetos físicos rígidos tienen bastantes parámetros, vamos a verlos en detalle:



- **Mode:** nos permite cambiar el tipo de comportamiento del rígido a otra cosa (**Kinectic**, etc.)
- **Mass:** Es la masa del objeto, no confundir con el peso.
- **Weight:** Es el peso, básicamente gravedad multiplicada por masa.
- **Physics Material Overrive:** Permite modificar los parámetros del material de física, ver más abajo.
- **Gravity Scale:** Un multiplicador de la gravedad.
- **Custom Integrator:** Si se marca permite implementar cualquier función de fuerzas personalizada (recuerda que la fuerza resultante es la sumatoria de las fuerzas combinadas). Tendrás en ese caso que implementarlo en el callback _integrate_forces().
- **Continous Cd:** También llamados balas (*bullets*). Se usa la trayectoria para predecir objetos muy rápidos y pequeños (de ahí lo de *balas*), este cálculo es menos óptimo, pero evita problemas de detección de este tipo de objetos.
- **Contacts Reported:** Es el máximo número de contactos que serán reportados al mismo tiempo. Utilizado por ejemplo, para conocer las fuerzas y los objetos que golpean a este objeto.

- **Contact Monitor:** El objeto emite señales cada vez que colisiona, muy útil para programar acciones activadoras de eventos (*triggers*).
- **Sleeping:** Si el objeto por defecto está en reposo (en clase te explicarán mejor esto).
- **Can Sleep:** Si el objeto puede quedar en reposo o no (en clase te lo explicarán mejor).

Si creamos un material físico, en **Physics Material Overrive:**, veremos que aparecen los siguientes parámetros:



- **Friction:** La resistencia con el suelo.
- **Rough:** Se aplica la rugosidad del objeto que tenga mayor rugosidad si está activo, en otro caso la del de menor rugosidad.
- **Bounce:** El rebote que tiene el objeto al chocar.
- **Absorbent:** Absorbe todos los rebotes del objeto con el que choca.

Más abajo encontramos los movimientos **Linear** y **Angular**, y un parámetro **Damp**, asociado a cada uno de ellos. Este parámetro permite (positivo) o evita (negativo) que haya fuerzas laterales de rozamiento en el objeto.

Podemos cambiar unos pocos parámetros para mejorar algo el movimiento de cámara: primero reduce bastante el tamaño del personaje, luego baja la fricción (**Friction**) a 0.1 o así. Y cambia las fuerzas de movimiento a 2 o -2, según su dirección. Verás que al chocar con paredes todo va bien, y con el cubo podemos caer (según el impacto que provoquemos).

Aumenta algo de masa de la cámara (tendrás que corregir la fuerza al andar) y baja la masa del cubo.

Mejor, pero aún puedes hacer caer al personaje. ¿Qué pasa? Lo que sucede es que el tipo de objeto rígido se comporta como una caja que vamos empujando, necesitamos estabilidad.

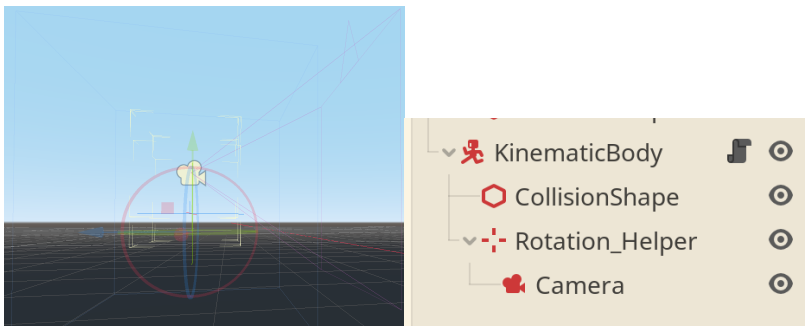
Vamos a cambiar el nodo padre, sustituye dicho nodo por un **KineticBody**, verás que lo primero que sucede es que el objeto no cae al suelo.

Ahora ve a la ayuda del nodo, y podrás comprobar que hay dos funciones interesantes, por un lado tienes move_and_collide, que permite mover el objeto y parar si colisiona, y por otro move_and_slide, que es similar, solo que habrá algo de rebote. Verás que las magnitudes en los movimientos necesitan ser mayores en *slide*, ya que tiene que calcular cierta fricción con los objetos.

Prueba a crear un *script* que use estas funciones en el movimiento del personaje.

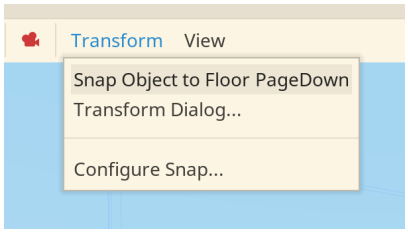
Fíjate que interesante es el parámetro test_only, que permite (si es *true*) que, sin moverse, provea información de si habría o no colisión en caso de que se hubiera movido en esa dirección y magnitud.

Ahora vamos a crear una cámara real, para ello, crea los siguientes nodos:



Recuerda la práctica anterior, el objeto llamado **Rotation_Helper** es un objeto de tipo Position3D, y se usa para la rotación **Yawn**.

Puedes ir al menú para colocar el objeto **kinetic** en el suelo.



Ahora crea un script en el nodo KinematicBody, y copia el código del jugador, disponible en este tutorial, cambia las referencias a las acciones y las referencias que necesites (*camera*, *rotation_helper*, etc.):

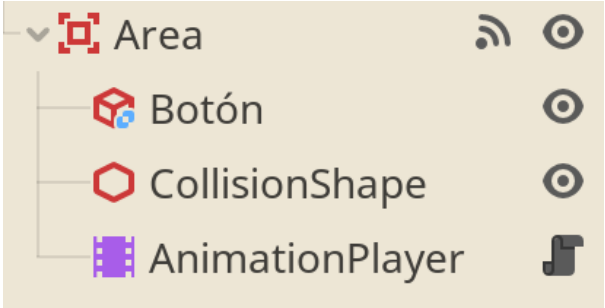
https://docs.godotengine.org/es/stable/tutorials/3d/fps_tutorial/part_one.html

2. Nodos dependientes de eventos y colisiones

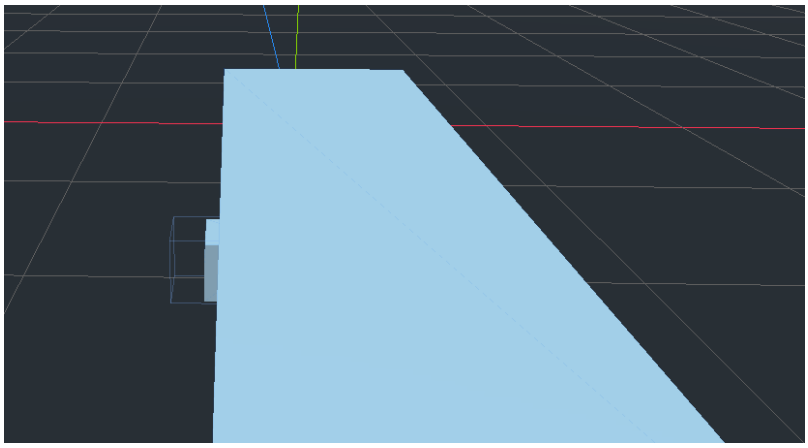
El nodo por defecto para actuar como un sensor es el nodo **Area**. Este nodo permite, no solo detectar si un cuerpo entra o sale de un determinado área, sino también si el ratón pasa por un determinado lugar o hacemos click en un objeto.

Vamos a hacer un ejemplo con el *click* del LBM del ratón. Para ello construye un pequeño cuadro de mandos (una caja con física estática, para que nuestro avatar no lo atraviese, y una caja a modo de botón).

Crea una simple animación de un botón hundiéndose y volviendo a su estado. Ahora crea un área (**Area**) y coloca el botón que has creado en él. El botón no necesita colisiones, así que una simple caja debería bastar, coloca también la animación dentro del objeto **Area** o bien como hijo del botón. Crea una **CollisionShape** que se ajuste al tamaño del botón, y hazla hija del **Area**. Fíjate en las imágenes, la idea es tener un árbol como este:



En el espacio 3D tu panel con botón debería parecerse a algo como esto:

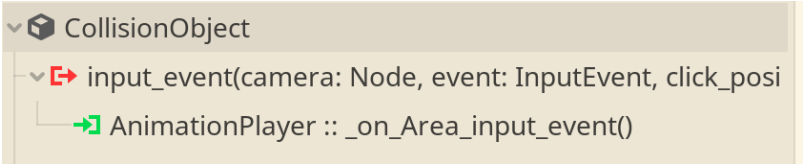


Ignora por ahora la señal que aparece en **Area**, volveremos a ella en un momento. Pero fíjate que el nodo AnimationScript tiene un *script*. Añádelo, por ahora no hay que quitar ni añadir nada al *script* de por defecto.

Ahora necesitamos conectar la señal del área a la animación, de forma que al hacer *click* en el área, se active la animación.

Por defecto, las áreas son sensibles al ratón, pero solamente cuando éste es visible y no se encuentra capturado. Si no hiciste el **Ejercicio de entrenamiento 1** de la práctica anterior, ahora es el momento. Algo muy típico (especialmente en juegos) es asociar el RBM para el cambio de modo de interacción al de visualización y movimiento de la cámara.

Ahora que tienes disponible el cursor, vamos a conectar el área y la animación juntos, vete a las señales y conéctalas como hemos comentado.



Ve al código, y en el *callback* correspondiente reproduce la animación que has hecho:

```
func _on_Area_input_event(camera, event, click_position, click_normal, shape_idx):
    play("Presionar")
```

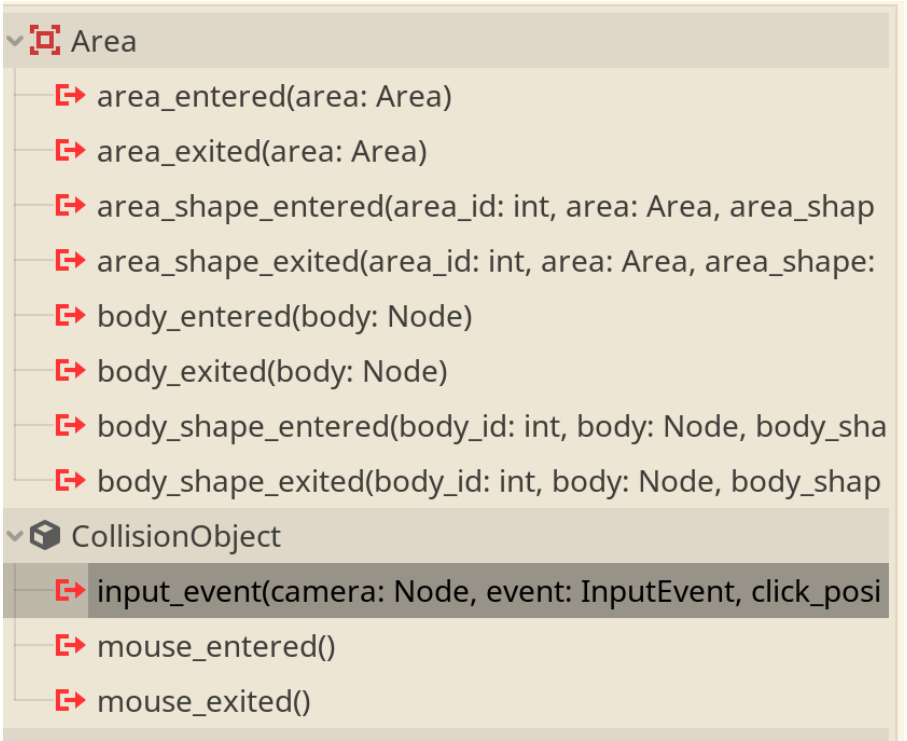
Lo probamos y… ¡No funciona!… no al menos como queremos.

!¿Qué está pasando?!

Pues que no estamos controlando que evento del ratón se está efectuando, así que simplemente por pasar el ratón por encima del botón ya se activa.

Ya sabes como controlar eventos de entrada, crea una acción y asigna el LBM, ahora comprueba en el código si esa acción se efectúa antes hacer *play* en la animación y verás como todo funciona perfectamente.

Volviendo a los eventos del área, comprobarás que existen multitud de situaciones que son interesantes para multitud de efectos. Como por ejemplo, detectar cuando un cuerpo entra o sale del área. Todos estos efectos se pueden ajustar con las opciones **Collision** y las capas (**Layer**) y máscaras (**Mask**). Esto lo verás mejor en clase de teoría, pero basta con comentar que sirve para ver qué objetos detectan a qué objetos, o que objetos pueden colisionar entre sí.



Además, las áreas también permiten sobreescribir los parámetros físicos, vamos a echar un vistazo a sus opciones:

Los más importantes (y que no hemos visto hasta ahora) son:

- **Space Override:** Permite sobreescribir o combinar los parámetros físicos. Es muy típico usar **Replace**. Pero hay que combinarlo en cualquier caso con **Gravity Distance Scale**.
- **Gravity Point:** Por defecto, la gravedad es un vector, pero puede convertirse en un punto, y así tener un punto de atracción radial en su lugar. Permite hacer efectos de imanes (sobre todo si se ajustan las máscaras y capas correctamente).
- **Gravity Distance Scale:** parámetro que indica como se hace la transición desde el área hasta la parte externa, y se usa como peso multiplicativo.
- **Priority:** Las áreas de más alta prioridad se efectúan primero.
- **Monitoring:** El área monitoriza que los objetos entren y salgan. Si vamos a usar un objeto

solamente para interacción con el ratón podemos desactivarlo.
- **Monitorable:** Las áreas pueden detectarse entre sí (muy útil para saber si estamos cerca de un sitio concreto y activar animaciones, sonidos de ambiente, etc.). Si lo desactivamos no puede ser monitorizada por otras áreas.
- **Input Ray Pickable:** Si lanzamos un rayo para detectar este área (por ejemplo al disparar, esto lo verás mejor en clase), si el rayo colisiona o la ignora. También sirve para detectar los eventos de dispositivos de entrada.
- **Input Capture On Drag:** Si los eventos se siguen lanzando cuando estemos arrastrando el ratón o no.

Puedes probar a hacer una zona con menor gravedad, si quieres hacer zonas con ausencia de gravedad deberías poner un poquito de gravedad negativa para compensar la inercia de los objetos. ¿Serviría para realizar una zona de agua?

3. Carga y descarga de escenas

Una de las cosas más relevantes a tener en cuenta cuando realizamos entornos virtuales, es el rendimiento de nuestro mundo. Si tu ordenador no es muy potente verás como introducir unas pocas cajas con física relentece toda la experiencia.

Podemos pensar que cuanto mejor el ordenador de desarrollo, mejor la experiencia creada. Craso error. Uno de los fallos más comunes es no planificar bien nuestros entornos. Ten en cuenta que si el mundo que creamos es muy amplio, habrá zonas que no sean visibles desde todos los lugares. Entonces, ¿para qué tenerlas cargadas en memoria? Piensa la cantidad de recursos desperdiciados en términos de memoria CPU y GPU (texturas, materiales, geometría, etc. etc.).

Uno de los métodos más comunes para adaptar las escenas es la carga y descarga dinámica de las mismas.

En videojuegos esto se hace constantemente, no es casual que en algunos juegos mal depurados se vean personajes (o vehículos) aparecer de la nada tras nuestro avatar.

Otra de las cosas más importantes es la interactividad y la sensación de inmersión (que está relacionada con la causa anterior). Así, tener un escenario dinámico, donde se puedan transportar objetos, y donde pueda conectar piezas, ayuda (y mucho) a la sensación de realismo en el mundo virtual.

Podría sorprenderte como este tipo de elementos dinámicos es mucho más importante que tener materiales fotorrealistas cuando hablamos de experiencia inmersiva.

La forma más sencilla en Godot para hacer todo este tipo de operaciones (cargar escenas, cambiar de escena, eliminar nodos, etc.) es trabajar con el *árbol general* mediante el uso de la función **get_tree()**. Por ejemplo, para cargar un nuevo nodo en la escena simplemente tenemos que usar **change_scene**:

```
get_tree().change_scene(«res://ruta/a/la/escena.tscn»)
```

Aquí tienes un ejemplo de cambio de escena completa que hereda de un nodo cualquiera llamado **tu_nodo**:

- Elimina el nivel actual:

```
var level = tu_nodo.get_node(«Nivel»)
tu_nodo.remove_child(level)
level.call_deferred(«free») # llama al método de liberar recursos
```

- Añadir el siguiente nivel:

```
var next_level_resource = load("res://ruta/a/la/escena.tscn)
var next_level = next_level_resource.instance()
tu_nodo.add_child(next_level)
```

Si lo que quieres es eliminar todos los recursos de un nodo y eliminarlo de la escena, puedes hacer directamente:

```
tu_nodo.queue_free()
```

Si quisieras transportar una caja, por ejemplo, lo único que tendrías que hacer sería eliminar el nodo de donde estuviera ubicado y añadirlo como hijo de la a cámara principal. Si quisieras colocarla, pues sería el proceso contrario.

A veces se utilizan inventarios, en este caso, el objeto puede ser hijo de un nodo que escale todos los objetos y que aparezca en un lateral, o utilizar renderizado por texturas (ver más abajo).

Revisa los métodos de la clase *Node*, para tener claro como buscar nodos, y como duplicarlos, añadirlos, etc. mediante *scripting*.

3. ECS y videojuegos

Es innegable que una gran parte de los entornos virtuales se desarrollan en la industria de los videojuegos. En este tipo de sistemas, y especialmente cuando se desarrollan mundos abiertos, es muy común utilizar un sistema de programación distinto, como el ECS. En teoría verás más como programar sistemas mediante ECS, pero en prácticas solamente te daremos algunas guías.

El ECS se basa en descomponer los objetos en características (una especie de *flags*), que indican como se comportará el objeto. Estas etiquetas pueden realizarse muy fácilmente en Godot mediante los *grupos*.

Sin embargo, estas etiquetas por sí solas no tienen código que las gestionen, en los ECS hay unos gestores que son los *sistemas*. En Godot no hay forma de que un nodo esté constantemente ejecutándose, ya que como ves, todos dependen del árbol. Aunque hay una excepción, los **Singleton**, objetos persistentes que se cargan al inicio y pueden monitorizar todo lo que sucede en el entorno virtual. Típicamente también se usan para mantener variables globales en los videojuegos, u otra información que quedaría extraño que estuviera en el propio nodo. Si quieres saber más acerca de como crear estos objetos de tipo **singleton** (en realidad son escenas), puedes leer el *manual*.

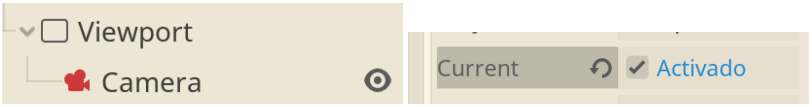
4. Renderizado en texturas

Godot tiene un componente especial (no tiene color) llamado **Viewport**.

Este componente está diseñado prácticamente en exclusiva para capturar una imagen de una cámara. Concretamente para utilizar la técnica de renderizado por textura. Si has programado alguna vez en OpenGL, o has leído foros de NVIDIA, quizás te suene el término equivalente FBO (*Frame Buffer Objects*).

El **Viewport** puede ser de cualquier tamaño, aunque algunas tarjetas gráficas antiguas requieren que el tamaño sea potencia de 2 (256×256, 512×512, 1024×1024, etc.).

Todo **viewport** necesita una cámara que tenga la propiedad **current** activa como hijo. Sí, en la escena puede haber varias cámaras **current** activadas al mismo tiempo, pero como hijo de un **viewport** solo puede haber una. Piensa que el nodo root, en realidad tiene otro padre que es el **viewport**, solamente que Godot lo ha *quitado* de la interfaz para hacer las cosas más simples.

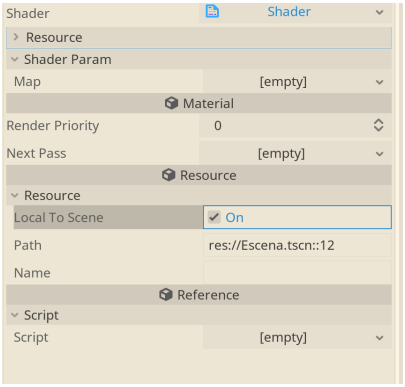


Una vez tienes las cámaras ya se podría crear cualquier objeto y aplicar la vista de la cámara como textura.

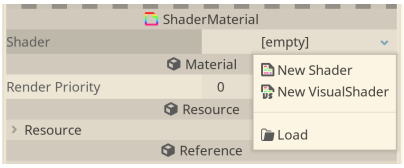
Por ejemplo podemos crear un simple quad (un plano cuadrado) y ponerlo como hijo de nuestra cámara principal (para que nos acompañe siempre), y aplicar un material. Sin embargo, los materiales que hemos visto hasta ahora no permiten hacer efectos complejos. Para ello necesitamos hacer uso de los *shaders*, tu profesor de teoría te hablará más de ellos en clase. Por ahora solamente necesitas saber que son programas en un lenguaje específico (GLSL, HLSL, etc., Godot tiene el suyo propio de más alto nivel), que te permiten decidir todos los detalles de cómo dibujar los elementos gráficos.

Para crear un material de este tipo solamente tenemos que seleccionar un material **ShaderMaterial** en lugar de un **SpatialMaterial**. Además tendremos que activar en **Resources** (abajo en las propiedades

del material), la casilla **Local To Scene**. Si no lo hacemos se quejará más tarde, al asignar la textura del **Viewport**.



Ahora sí, vamos a crear un **Shader** (no marques **Visual Shader**):



Ahora sí, nos dejará escribir código, tu profesor te explicará este código en clase.

```
1  shader_type spatial;
2  render_mode unshaded;
3
4  uniform sampler2D map;
5  void fragment() {
6      ALBEDO = texture(map, vec2(UV.x, UV.y)).rgb;
7  }
```

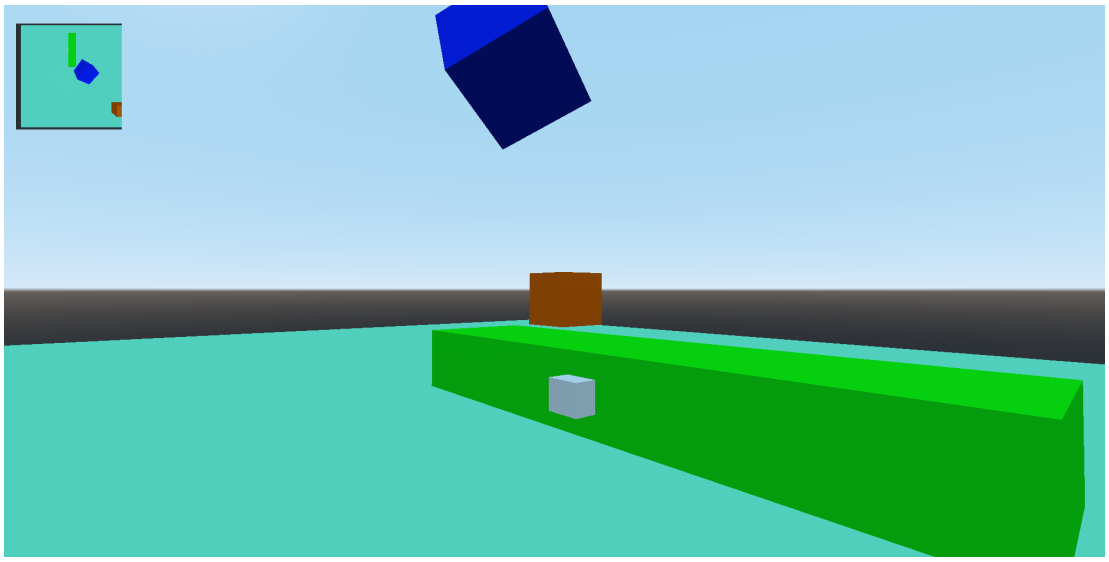
Por ahora solamente necesitas saber que:

- **Líneas 1 y 2:** Se establece el tipo de shader (3D) y los flags del material que queremos activar.
- **Línea 4:** Se pide una textura a la que llamaremos map.
- **Línea 6:** Se cambia el albedo (concretamente el color) y se le asigna los colores dadas por la textura llamada map, usando las coordenadas de textura (UV) del propio objeto.

Finalmente, si miramos los parámetros del shader, aparecerá uno llamado **Map**, es nuestra textura, que la tenemos sin asignar aún.

Ahora podemos elegir la última opción **Viewport Texture** y nos dejará marcar nuestro **Viewport**.

Si colocamos la cámara en una posición superior, dominando el escenario tendremos un minipama que se actualizará en tiempo real según nos desplacemos en la escena.



Con esta técnica podemos simular cámaras de vigilancia y monitores, espejos retrovisores, ¡incluso geometría no euclídea!

Ejercicio calificable P4. A.

> En esta práctica deberás terminar de modelar la física de tu sistema. Añade la cámara FPV indicada en esta ultima parte de la guía, e intégrala en tu mundo.

> Añade elementos interactivos (parecidos al botón que hemos visto, por ejemplo en puertas, etc.) y permite transportar algún objeto con el ratón y soltarlo en algún sitio.

> Añade un área de tal forma que al entrar en él la habitación que no se vea desaparezca del árbol, liberando sus recursos. Haz lo mismo con la otra habitación.

> Plantéate qué sucede con los objetos físicos que creaste, ¿cómo podrías solucionar el problema? No es necesario que lo resuelvas, solamente plantea cuál sería la solución.

Ejercicio final opcional (EXTRA)

> Este ejercicio es extra y tu profesor te comentará más acerca de los beneficios de realizarlo en clase.

> Termina el sistema virtual de dos habitaciones. Para ello, añade técnicas avanzadas, como renderización en texturas, uso de shaders, uso de otros elementos que hayas visto en clase (partículas, quadtrees, etc.).

Deja volar tu imaginación, además de emplear las técnicas correctas. Algunos ejemplos de cosas que se pueden hacer (sin estar límitados a ellas) en esta última parte como extra son:

- **Puertas con sensores:** No dejan pasar a nadie a menos que portes la tarjeta correcta. Cada puerta, o sistema de seguridad tiene su tarjeta específica que tendrás que insertar en una ranura.
- **Sistema de inventario:** Tienes una tecla que muestra un inventario, puedes ir guardando objetos ahí, y puedes sacar objetos del inventario para colocarlos en lugares específicos. Si los sueltas en el suelo se aplicará la física.
- **Sistemas de vigilancia mediante monitores:** puedes ver otras salas mediante monitores en la escena. Podrías tirar un monitor y funcionaría, o podría dejar de funcionar. Podrías mover las cámaras y colocarlas en otros sitios también.
- **Zonas de exploración submarina:** en el agua los objetos que sueltes flotan, cuando te lances al agua todo se verá un poco más azul, y quizás con distorsión.
- **Zonas de magia o efectos especiales:** Fuego, electricidad, chispas, etc. Con sistemas de partículas puedes implementar algunos de estos efectos y más. También podrían afectarte este tipo de sistemas de forma física (pérdida de vida, maná, etc.) y tener consecuencias (no hay más magia, muerte, etc.).
- **Resolver un puzle para salir:** el ascensor no se mueve, las puertas están cerradas, etc. hasta que consigues colocar todas las piezas que están aleatoriamente dispersas en la habitación y la máquina que mueve el engranaje vuelve a funcionar, dejándote avanzar a la otra habitación.