eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Computación

Trabajo de Fin de Grado

Diseño y desarrollo de una herramienta para ayudar en el aprendizaje de lenguajes orientados a objetos

Autor

Miguel Albors Iruretagoyena

2022

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Computación

Trabajo de Fin de Grado

Diseño y desarrollo de una herramienta para ayudar en el aprendizaje de lenguajes orientados a objetos

Autor

Miguel Albors Iruretagoyena

Directora

Begoña Ferrero

Resumen

Al aprender un nuevo lenguaje de programación es muy útil disponer de una batería de proyectos a realizar que permitan afianzar lo aprendido. Durante mi aprendizaje he encontrado que los proyectos audiovisuales suelen resultar motivadores y refrescantes. Desgraciadamente, las herramientas para crear aplicaciones audiovisuales son, por lo general, bastante inaccesibles para alguien que está en pleno proceso de aprendizaje.

Este trabajo diseña una librería que permita tanto a programadores novatos como a experimentados crear todo tipo de aplicaciones audiovisuales; desde la representación de ecuaciones pintando sus valores pixel a pixel, hasta videojuegos completos.

Índice general

Índice de figuras

1

Introducción

Uno de los principales retos que encontramos al aprender a programar es disponer de proyectos en los que se apliquen los conceptos aprendidos para reforzarlos y afianzarlos. Habitualmente se tiende a desarrollar proyectos poco estimulantes que no tienen más utilidad que la meramente académica.

A lo largo de la última década me he encontrado con muchos proyectos muy interesantes que, de haberlos realizado durante mi aprendizaje, lo habrían amenizado y acelerado.

En este trabajo me centraré en las aplicaciones audiovisuales; puesto que éstas presentan toda una serie de retos intrínsecos a sus características propias que en otros ejercicios no brotan con la naturalidad con la que lo hacen en este tipo de proyectos. Incluso la aplicación más sencilla requerirá que el programador aplique sus conocimientos en geometría, busque refinar sus algoritmos para hacerlos más eficientes e, incluso, busque métodos para ahorrar memoria.

Por desgracia, las herramientas gratuitas más extendidas o abstraen al programador de estos retos o son demasiado inaccesibles para un programador novel.

Por eso, en este trabajo se va diseñar e implementar una librería que haga accesible una o más de esas herramientas sin abstraer al programador de estos retos.

2

Antecedentes

Se pueden categorizar las soluciones disponibles en dos tipos: motores y herramientas. Esta categorización se realiza atendiendo a la forma que presentan y, con ello, los problemas que resuelven y los que presentan.

Este trabajo se va a limitar a las soluciones más comunes y extendidas, dejando fuera algunas herramientas que para un futuro trabajo sí podrían ser interesantes como la demotool *a.D.D.i.c.t.2* o el programa *RPG Maker*.

2.0.1. Motores de videojuegos

Un motor de videojuegos es una serie de herramientas que ofrecen todas las funcionalidades requeridas para poder desarrollar videojuegos. Se enfocan en ofrecer una solución completa que permita al desarrollador enfocarse en el diseño de su juego, delegando al motor todas las tareas técnicas [3].

Algunas de las funcionalidades que suelen incluir para poder considerarse motores son:

- **Sistema de scripting:** Todos los motores disponen de la capacidad de crear scripts que proporcionen a los elementos del juego la lógica que necesitan para funcionar. Algunos lo implementan a través de un lenguaje propio que, posteriormente, compilan. Otros, ofrecen una extensa API interna e, incluso, los hay que ofrecen sistemas híbridos. En cualquier caso, todos los motores integran estos scripts dentro de un bucle general (ver figura 2.1).

- **Motor de sonido:** La banda sonora y los efectos de sonido son una parte importante dentro de los videojuegos. Los motores de sonido permiten que los diseñadores de videojuegos puedan controlar no sólo qué suena, sino dónde y cómo, pudiendo hacer que el volumen de cada sonido se ajuste a la distancia con el jugador, simulando el efecto real de alejarse de una fuente de sonido.

- **Motor de físicas:** Aunque no todos los videojuegos hacen uso de ello, muchos no serían posibles sin aplicar aproximaciones eficientes de la mecánica clásica. Aunque la mayoría de motores físicos se ciñen a simular el uso de fuerzas e inercias en el vacío, algunos motores añaden funcionalidades adicionales como el comportamiento de tejidos y fluidos.

- **Motor gráfico:** Todos los motores integran un sistema para representar en pantalla los elementos instanciados. Para ello suelen ofrecer métodos de carga de ficheros gráficos, estructuras de datos destinadas a representar la transformación de los objetos y un sistema de pintado de dichos gráficos, aplicándoles las transformaciones, que permite al desarrollador abstraerse de dicho trabajo. Muchos de ellos suelen incluir sistemas para compilar y ejecutar shaders.

Algunos ejemplos de motores son Unity3D [4], Unreal Engine [5], GameMaker Studio 2 [6] y Godot [7].

2.0.2. Herramientas

El nombre de esta categoría se ha decidido así porque agrupa soluciones muy diversas; las cuales se diferencian de los motores de videojuegos en que no son soluciones tan completas. Algunas, como el lenguaje de programación BennuGD [8], carecen de motor de sonido (limitándose a la funcionalidad básica de reproducir sonido) o motor de físicas[1].

Otras, de hecho la mayoría, son librerías específicas orientadas a solucionar un problema concreto. OpenGL, por ejemplo, aporta una api para renderizar objetos 3D; SDL y Allegro permiten renderizar gráficos 2D y Bass proporciona lo necesario para reproducir la mayoría de los archivos de sonido conocidos [9].

Estas herramientas, si bien, para un programador experimentado son muy útiles para realizar sus proyectos audiovisuales, son demasiado complicadas para que usuarios con poca

[1] En muchos casos, la comunidad de dichas soluciones han aportado módulos propios que solventan estas carencias.

experiencia puedan utilizarlas. Por ello, la solución que se va a diseñar en este trabajo va ser una librería que haga uso de una o más de las herramientas presentadas para ofrecer a los programadores noveles una interfaz más amigable permitiéndoles abstraerse de las partes más engorrosas de su uso.

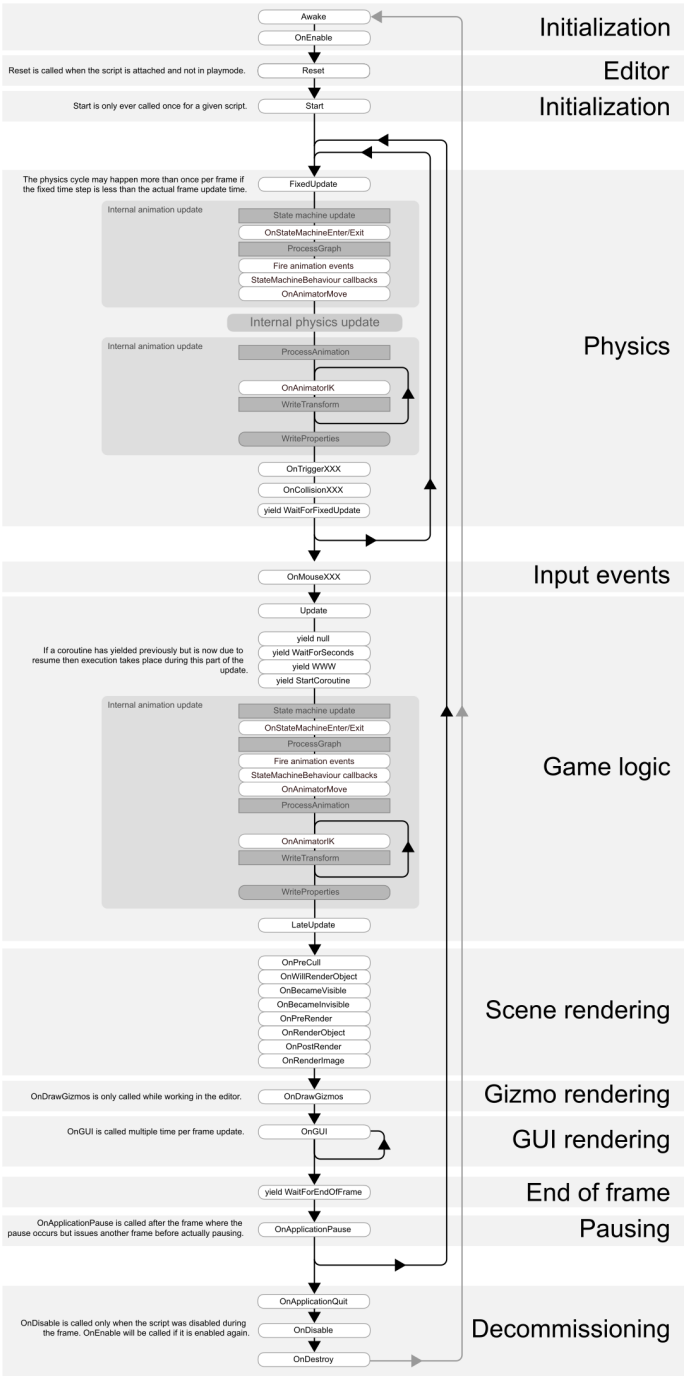

Figura 2.1: Ciclo de ejecución de los scripts de Unity [1]

3

Objetivos y requisitos

El objetivo que se plantea en este trabajo es construir una librería que implemente los recursos necesarios para desarrollar aplicaciones audiovisuales de gráficos 2D, siendo los videojuegos las aplicaciones más complejas que deberían poderse desarrollar.

Como objetivo secundario, se plantea que todas las estructuras de datos de esta librería sean fácilmente ampliables por el usuario, empleando mecanismos de herencia y polimorfismo.

En cuanto a los requisitos que debe cumplir esta librería, tras analizar otras soluciones existentes y las necesidades intrínsecas de este tipo de aplicaciones, se plantean los siguientes:

- *Sistema de scripting*, que permita integrar en un gestor de procesos los diferentes elementos a los que el desarrollador quiera aportar una lógica.
- *Sistema de ventanas y renderizado*, que permita al desarrollador crear ventanas y pintar en ellas tanto formas geométricas primitivas (puntos, líneas, rectángulos y círculos) como gráficos más complejos.
- *Sistema de gestión de assets*, que permita al desarrollador cargar recursos gráficos y de sonido con comodidad.
- *Detección de colisiones* entre dos procesos.

- *Detección de entrada* para permitir que un usuario pueda interactuar con la aplicación.
- *Sistema de reproducción de recursos sonoros*.

4

Desarrollo del proyecto

4.1. Sistema de scripting

Tomando como referencia el modo en el que permiten a un desarrollador crear la lógica que haga funcionar su aplicación, podemos clasificar las herramientas actuales en tres grandes grupos: las que proporcionan su propio lenguaje, las que proporcionan una api y las que ofrecen una clase base integrada con el pipeline.

- *Herramientas con lenguaje propio:* El desarrollador debe diseñar su aplicación como un conjunto de scripts en el lenguaje propio de la herramienta. Posteriormente, la herramienta traducirá dichos scripts a un lenguaje de pila que podrá interpretar durante la ejecución de la aplicación.

  La principal ventaja de estas herramientas es que el archivo de pila que genera es intercambiable entre plataformas sin necesidad de volver a ser compilado.

  Ejemplo: BennuGD [8].

- *Herramientas con api:* Esta es una de las formas más comunes. El ejecutable generado por este motor, durante su inicio, carga scripts que contienen llamadas a la api que proporciona. Usualmente estos scripts se desarrollan en Lua, aunque hay herramientas que emplean otros lenguajes (como javascript o Ruby).

Al igual que la categoría anterior, las aplicaciones hechas con estas herramientas son multiplataforma. Además, ganan en eficiencia puesto que los scripts no son interpretados durante la ejecución, sino solamente durante el inicio.

Por desgracia presenta una desventaja crítica: las aplicaciones desarrolladas con estos motores sólo pueden hacer lo que sus api permiten, muchas veces sin opción a modificarla.

Ejemplo: Minetest [10].

- *Herramientas con clase base integrada en el pipeline:* Esta es la forma más común. Las herramientas ofrecen una clase base de la que deben heredar todos los scripts. Esta clase contiene toda una batería de miembros que dotan al script de integración; desde funciones a las que se irán llamando automáticamente hasta variables donde almacenar datos internos (como la posición, escala, gráfico,...).

  A diferencia de las herramientas descritas anteriormente, estas requieren que las aplicaciones desarrolladas con ellas sean compiladas para cada plataforma de destino específicamente. Y aunque eso es una desventaja, permite que los scripts se vean beneficiados de la optimización que proporcionan los compiladores y de todas las ventajas derivadas de tener todo el ejecutable en el código máquina de la plataforma donde se ejecute.

  Ejemplos: Unity3D [4], Unreal Engine [5].

Tras evaluar estos tres métodos se ha determinado hacer uso de este último método *"Herramientas con clase base integrada en el pipeline"* puesto que es el método que mejor se adapta al objetivo.

4.1.1. Pipeline

Considerando el funcionamiento de las herramientas mencionadas en el apartado anterior se ha observado que en todos los pipelines las clases base proporcionadas suelen ofrecer al menos las siguientes funciones miembro:

- *Una función de inicialización* del objeto. Esta función se invoca al instanciar la clase evitando que el programador haga uso del constructor, dejándolo libre para que sea la herramienta la que haga uso del mismo.
- *Una función de actualización*. Según la herramienta, esta función se ejecuta una vez por fotograma o tantas veces como sea posible entre dos fotogramas.
- *Una función de actualización durante el renderizado*. Algunas herramientas la incluyen antes del renderizado, otras después y otras ofrecen ambas.
- *Una función de eliminación*. Esta función se invoca al liberar el objeto evitando que el programador haga uso del destructor, dejándolo libre para que sea la herramienta la que haga uso del mismo.

De acuerdo a este funcionamiento, el diseño presentará el siguiente pipeline:

1. Al instanciarse la clase se invocará al miembro *Start*.
2. Tantas veces como sea posible entre dos fotogramas se invocará al miembro *Update*.
3. Únicamente una vez por fotograma se invocará al miembro *FixedUpdate*.
4. Justo antes de llamar al sistema de renderizado se invocará al miembro *Draw*.
5. Durante el proceso de renderizado, cuando ya se hayan ordenado los procesos en el orden en el que se van a dibujar en pantalla, se invocará al miembro *LateDraw*. Esta función sólo será invocada si el objeto contiene un gráfico que pueda ser renderizado.
6. Al finalizar el proceso de renderizado se invocará al miembro *AfterDraw*. Esta función sólo será invocada si el objeto contiene un gráfico que pueda ser renderizado.
7. Al eliminar el objeto se invocará el miembro *Dispose*.

En la figura 4.1 se muestra de un modo más visual.

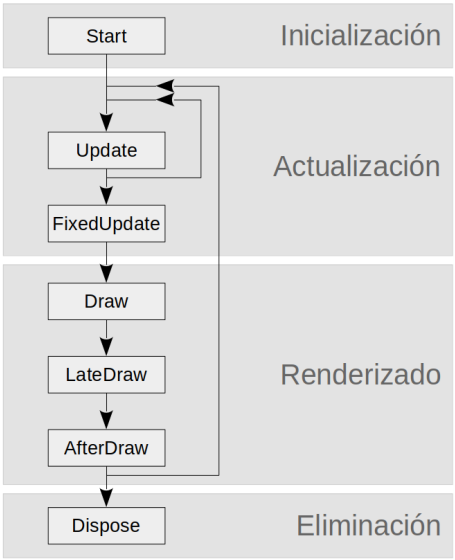

Figura 4.1: Ciclo de ejecución

Algunos miembros, como el trío *Draw*, *LateDraw* y *AfterDraw*, pueden parecer redundantes, sin embargo existen casos en los que disponer de ellos puede ser determinante.

Un ejemplo común es el caso de querer mostrar la figura de un elemento que se encuentra escondido tras otro elemento. Supongamos un juego de disparos en el que los enemigos pueden esconderse tras coberturas. Para facilitar las cosas al jugador se podría querer que, cuando un enemigo se esconda tras una cobertura, la figura de este enemigo se dibuje en un color sólido que contraste sobre la cobertura tras la que se ha escondido.

En esta situación, se necesitaría hacer uso de Draw para colocar los gráficos en relación a la posición de la cámara y que así el sistema pueda ordenarlos correctamente al ir a pintarlos, pero habría que esperar a LateDraw para pintar la figura porque hasta que el sistema no ordene los procesos para pintarlos no se puede saber sobre qué proceso se deberá pintarla. Finalmente, habría que hacer uso de AfterDraw para devolver al proceso su gráfico original.

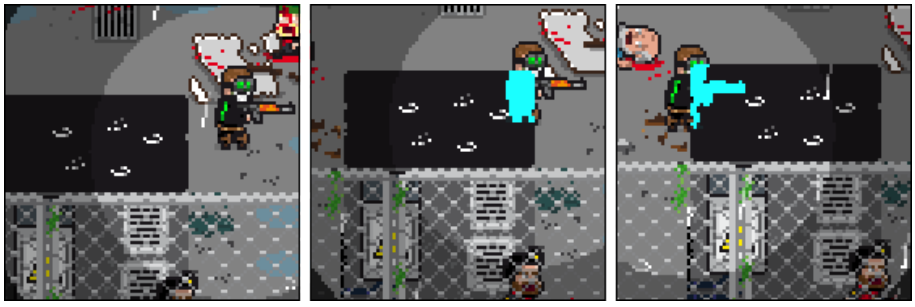

Figura 4.2: Ejemplo de pintado de la figura escondida

4.1.2. Clase *Process*

Esta clase es la destinada a que hereden todos los elementos con lógica, por lo que es la que incluye todas las funciones miembro descritas en el apartado anterior. Pero, además de los miembros a invocar por el sistema, necesitará toda una serie de atributos que garanticen esta integración.

Tras analizar los atributos más comunes en otras herramientas (como los atributos de los GameObject de Unity [11]) la clase *Process* contendrá los siguientes miembros:

- *id*: almacenará el identificador único del proceso.
- *father*: almacenará el identificador del padre del proceso.
- *signal*: almacenará el estado en el que se encuentra el proceso (más adelante se determinará cuáles serán estos).
- *type*: almacenará el tipo al que pertenece el proceso. Ej: enemigo, jugador, checkpoint,...
- *graph*: almacenará la dirección de memoria donde estará el gráfico que le corresponda (en caso de corresponderle uno).
- *transform*: almacenará la posición, rotación y escala del proceso.
- *zLayer*: almacenará el orden de capa del proceso. Este atributo será fundamental para que el sistema pueda determinar qué proceso se pintará encima de qué otro proceso; siendo los procesos con zLayer mayor los que se pintarán por encima.
- *flipType*: almacenará las flags con las que el gráfico asociado (en caso de haberlo) debe ser renderizado.
- *window*: almacenará el puntero a la ventana en la que deberá renderizarse.

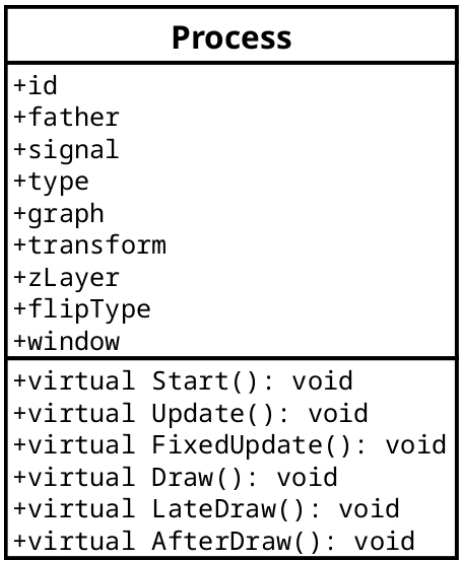

Figura 4.3: Esquema de la clase Process

Además, contará con sobrecargas en los operadores de comparación para poder comparar el zLayer de los procesos con comodidad.

Señales

Como se ha indicado previamente, los procesos tendrán un atributo llamado *signal* que almacenará el estado en el que se encuentren. Además, deberá diseñarse un sistema que permita que los procesos se cambien los estados entre ellos.

Las señales disponibles serán las siguientes:

- S_AWAKE: el proceso está despierto. Este es el estado normal.
- S_AWAKE_TREE: el proceso está despierto y debe despertarse a todos sus descendientes. Al terminar de ejecutarse, esta señal se transformaría en S_AWAKE.
- S_KILL: el proceso está muerto. Al procesar esta señal el sistema eliminará este proceso y a todos sus descendientes.
- S_KILL_CHILD: todos sus descendientes deben ser eliminados. Al procesar esta señal, el sistema eliminará todos sus descendientes. Al terminar cambiará esta señal a S_AWAKE.
- S_SLEEP: el proceso está dormido. Estando en este estado, ni se invocarán las funciones miembro del proceso y ni se enviará al sistema de renderizado.

- S_SLEEP_TREE: el proceso está dormido y se debe dormir a todos sus descendientes. Al procesar esta señal, el sistema dormirá a todos sus descendientes y cambiará este estado a S_SLEEP.

Posición, rotación y escala

Como se ha descrito previamente, los procesos tendrán un atributo *transform* que almacenará la posición, rotación y escala del proceso.

Al ser un motor en 2D, tanto la posición como la escala contienen dos valores: x e y. La rotación, sin embargo, es un único valor: el ángulo de rotación en el plano xy.

Por ello, para la posición y la escala se empleará una estructura llamada *Vector2*[1], que se encargará de almacenar los valores x e y y sobrecargará todos los operadores necesarios para facilitar su uso.

Para agrupar estos tres atributos en un único atributo, se empleará una estructura llamada *Transform2D* que contendrá los dos Vector2 correspondientes a la posición y escala y la variable que contendrá la rotación.

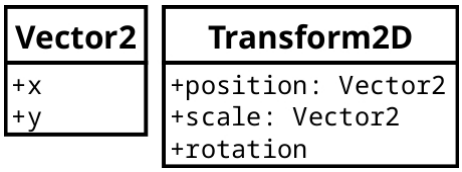

Figura 4.4: Esquema de las estructuras Vector2 y Transform2D

[1] El nombre vector lo recibe por ser una estructura que almacena dos valores, no por el ente matemático.

4.1.3. Gestor de tareas

Para integrar tanto el pipeline como la clase base, se vuelve necesario disponer de un gestor tareas que se encargue tanto de almacenar los procesos y sus relaciones entre ellos como de ir invocando los miembros del pipeline.

Estructuras de datos

Puesto que los procesos se podrán jerarquizar mediante la fórmula *padre/hijo*, se considera adecuado emplear un árbol n-ario para almacenarlos. De este modo, el contenido del atributo *id* que contiene la clase *Process* emerge por sí mismo: *id_del_padre:type*. Como del mismo padre podrían pender más de un proceso con el mismo type, se añadiría un número al final del id.

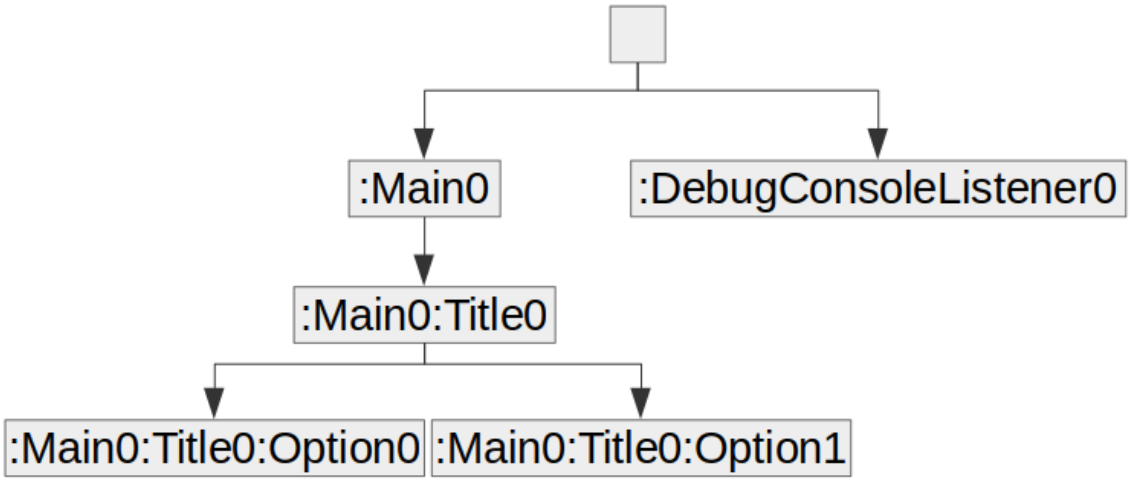

Figura 4.5: Ejemplo de la jerarquía propuesta para el gestor de tareas

Como se prevé que la mayoría de los accesos al gestor de tareas será buscando el proceso que coincida con el identificador indicado y teniendo en cuenta que este gestor de tareas debe responder eficientemente en un escenario como el de un videojuego, en el que se realizan miles de llamadas al gestor de tareas por fotograma, necesitamos hacer uso de alguna estructura más eficiente para este objetivo. Además, es posible que se quieran realizar otros tipo de acceso: por el *type* del proceso. Un ejemplo sería que se quisiera borrar todos los procesos del tipo “enemigo”.

Por ello, se emplearán dos estructuras de datos adicionales:

- Un mapa desordenado: esta estructura es un array asociativo no ordenado que asocia

pares de datos, siendo el primero la clave y el segundo el valor. Se empleará esta estructura para alamacenar punteros a los procesos empleando el par {identificador, puntero} porque, dentro de esta categoría de estructuras, su orden de acceso es el más rápido [12].

- Un multimapa desordenado: este también es un array asociativo que asocia pares de datos. Pero, a diferencia del anterior, permite asociar la misma clave a varios elementos [13]. Lo cual es perfecto para tener un lugar donde poder asociar los tipos de proceso con los punteros a ellos.

Funciones miembro

El gestor de tareas también deberá contar con toda una batería de funciones miembro que permitan invocar al pipeline, crear nuevos procesos, acceder a ellos de diversas maneras, eliminarlos, etc...

Tras analizar los sistemas de gestión de tareas de otras soluciones, este gestor contendrá los siguientes métodos:

- *Constructor* y *destructor*
- *newTask<class>([father,window]*: esta función instanciará la clase señalada (siempre y cuando esta herede de *Process*), le asignará un identificador, lo introducirá tanto en el árbol como en los mapas e invocará a su miembro *Start()*.
- *deleteTask(id)*: esta función eliminará el proceso indicado, liberando el espacio que ocupa en memoria y sacándolo tanto del árbol como de los mapas. Además, antes de eliminarlo, invocará a su miembro *Dispose()*. Si el proceso tuviera hijos, repetiría estas acciones recursivamente para cada uno de ellos. Será una función de uso interno puesto que en la sección anterior se ha planteado que la "vida" de los procesos se gestionará mediante señales.
- *existsTask(id)*: esta función comprobará si existe en el gestor de tareas un proceso con el identificador indicado.
- *getTask(id)*: esta función devolverá un puntero al proceso con el identificador indicado. En caso de no existir, devolverá nulo.

- *getTaskByType(type)*: esta función devolverá una lista con punteros a todos los procesos cuyo tipo coincida con el indicado. En caso de no existir ninguno, devolverá una lista vacía.

- *getTaskByFather(id)*: esta función devolverá una lista con punteros a todos hijos del proceso cuyo identificador coincida con el indicado. En caso de no tener hijos o de no existir el proceso, devolverá una lista vacía.

- *getTaskList()*: esta función devolverá una lista con punteros a todos los procesos contenidos en el gestor de tareas. En caso de no haberlos, devolverá una lista vacía.

- *getTaskIdList()*: esta función devolverá una lista con los identificadores de todos los procesos contenidos en el gestor de tareas. En caso de no haberlos, devolverá una lista vacía.

- *sendSignal(id, signal)*: esta función asignará la señal indicada al proceso con el identificador indicado.

- *manageSignals()*: esta función actualizará la situación de todos los procesos según la señal que tengan asignada. Por ejemplo, eliminará los que tengan la señal *S_KILL*. Será una función de uso interno, puesto que será invocada por una de las funciones responsables del pipeline.

- *Update()*: esta función es la responsable de hacer cumplir la parte del pipeline en la que se gestionan las señales y se invoca al *Update()*. Para ello, invoca a a la función *manageSignals()* y luego al *Update()* de cada uno de los procesos contenidos en el gestor de tareas cuyo signal sea S_AWAKE.

- *FixedUpdate()*: esta función es la responsable de hacer cumplir la parte del pipeline en la que se invoca al *FixedUpdate()* de todos los procesos contenidos en el gestor de tareas. Para ello, invoca al *FixedUpdate()* de cada uno de los procesos contenidos en el gestor de tareas cuyo signal sea S_AWAKE.

- *Draw()*: esta función es la responsable de hacer cumplir la parte de renderizado del pipeline. Para cada proceso contenido en el gestor de tareas invoca a la función Draw() y, si no está dormido, lo inserta en una lista auxiliar ordenada por el zLayer de los procesos. Una vez invocados y ordenados todos los procesos recorre la lista auxiliar realizando las siguientes tareas para cada proceso:

  1. Invoca al miembro *LateDraw()*.

2. Renderiza al proceso en la ventana cuyo puntero contiene su atributo *window*.
3. Invoca al miembro *AfterDraw()*.

- *Clear()*: esta función elimina a todos los procesos contenidos en el gestor de tareas, invocando al miembro *Dispose()* de cada uno.

Además, contendrá un atributo estático llamado *instance* que será un puntero a la única instancia que deberá haber del gestor de tareas, garantizando así su funcionamiento como *Máquina Abstracta de Estados*.

De este modo, para hacer cumplir con el pipeline el desarrollador que emplee esta librería deberá hacer uso del miembro *newTask<class>([father,window])* para instanciar las clases, en el bucle principal deberá invocar a los miembros *Update()* y *Draw()* y deberá emplear el miembro *sendSignal(id,signal)* para marcar los procesos que quiera eliminar.

```
float wait_to_draw = 0, fps = 0.1/60.0;
new TaskManager();

while(!exit)
{
    do
    {
      TaskManager::instance->Update();
      wait_to_draw = wait_to_draw + Clock.deltaTime;
    } while(wait_to_draw < fps);

    TaskManager::instance->FixedUpdate();
    TaskManager::instance->Draw();
}
```

Figura 4.6: Ejemplo en C++ de un bucle de juego que hace uso de este diseño

4.2. Sistema de ventanas y renderizado

4.2.1. Las ventanas

Aunque en la mayoría de motores de videojuegos se considera que la aplicación tendrá una única ventana y, por tanto, se organiza el diseño entorno a ello, resulta interesante poder ofrecer al desarrollador la posibilidad de tener varias ventanas y que sea él quien decida cuántas quiere usar en su aplicación.

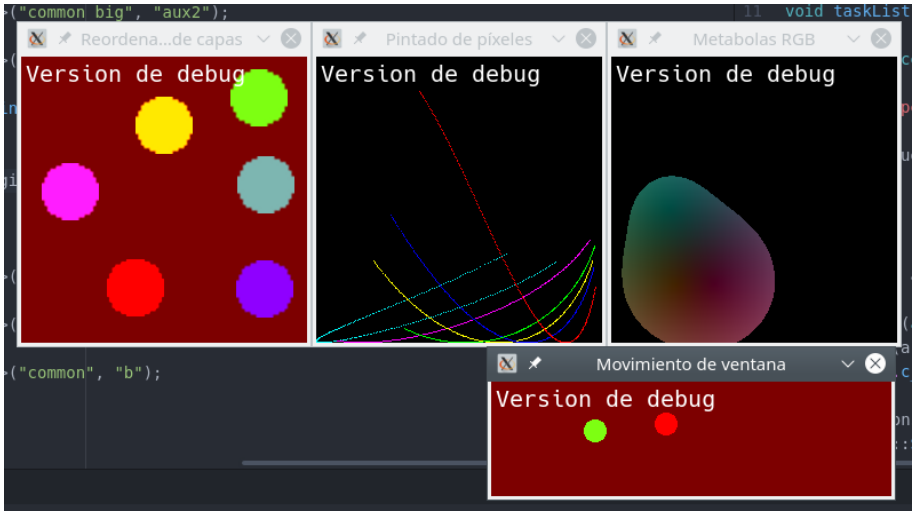

Figura 4.7: Ejemplo de una aplicación con múltiples ventanas [2]

Atributos

La clase *Window* contendrá los siguientes atributos:

- *title*: almacenará el título con el que se muestra la ventana.
- *index*: almacenará el identificador único de la ventana.
- *windowMode*: almacenará el modo de la ventana. Según la librería con la que se implemente puede limitarse al duo pantalla completa/ventana con marco, pero podría no limitarse a ello.
- *rendererMode*: almacenará el modo de renderizado de la ventana. Al igual que con los modos de ventana, cada implementación presentará una gama diferente de modos.

- *position*: almacenará la posición de la ventana respecto a la pantalla del ordenador donde se ejecute.
- *size*: almacenará el tamaño de la ventana.

Adicionalmente, según la librería con la que se implemente, contendrá los atributos que estas librerías requieran para su funcionamiento específico.

En la librería SDL2, por ejemplo, las ventanas deben disponer de los atributos *window:SDL_Window* y *renderer:SDL_Renderer* para poder funcionar.

En cualquier caso, todos los atributos serán privados y sólo podrán modificarse mediante *getters* y *setters*, para garantizar que la modificación de estos atributos se hace siguiendo las indicaciones específicas de cada implementación.

Funciones miembro

Aunque cada implementación tendrá que realizar las tareas de renderizado de diferente manera, seguirán compartiendo las mismas tareas a realizar. Por lo que las funciones miembro que contendrá la clase *Window* serán las siguientes:

- *Constructor* y *destructor*: los cuales se encargarán de preparar la clase tal y como lo requiera la implementación.
- *Getters* y *setters* de los atributos.
- *Rend(graph,transform,flipType)*: esta función se encargará de pintar el gráfico indicado en la posición indicada[2] con la rotación y la escala indicadas aplicando las flags indicadas con *flipType*.
- *doRend()*: esta función finaliza el proceso de renderizado. Todas las librerías consultadas requieren ejecutar una serie de órdenes para hacer efectivo el renderizado. Esta función evita que se realicen en la función *Rend()*, la cual se invocará una vez por cada elemento a renderizar.
- *setBackColor(color)*: esta función asigna a la ventana un color de fondo.
- *move(x,y)*: esta función desplaza la ventana *x* unidades en el eje horizontal e *y* unidades en el eje vertical.
- *hasFocus()*: esta función indica si la ventana tiene el foco o no.

[2]La cuestión del posicionamiento de los gráficos en relación a su posición presenta algunas cuestiones que se discutirán más adelante.

De esta manera, la función *Draw()* del gestor de tareas invocaría al miembro *Rend(graph, transform, flipType)* para cada proceso a renderizar y, una vez terminado, quedaría en manos del bucle principal hacer efectivo ese renderizado invocando al miembro *doRend()*.

Posicionamiento de los gráficos

A la hora de pintar gráficos con un ancho y un alto superiores a un píxel nace un problema importante: ¿las coordenadas *x* e *y* que señalan la posición del elemento a qué coordenada del gráfico representan?

Por un lado, teniendo en cuenta que el origen de coordenadas se sitúa en la esquina superior izquierda de la ventana (creciendo los ejes hacia la derecha y hacia abajo) puede argumentarse que en estos elementos debe hacerse del mismo modo y situar este punto en la esquina superior izquierda del gráfico.

Por otro lado, teniendo en cuenta que cuando vemos un gráfico, de manera natural, tendemos a evaluar su posición en relación a su centro, también puede argumentarse que este punto se sitúe en el centro del gráfico.

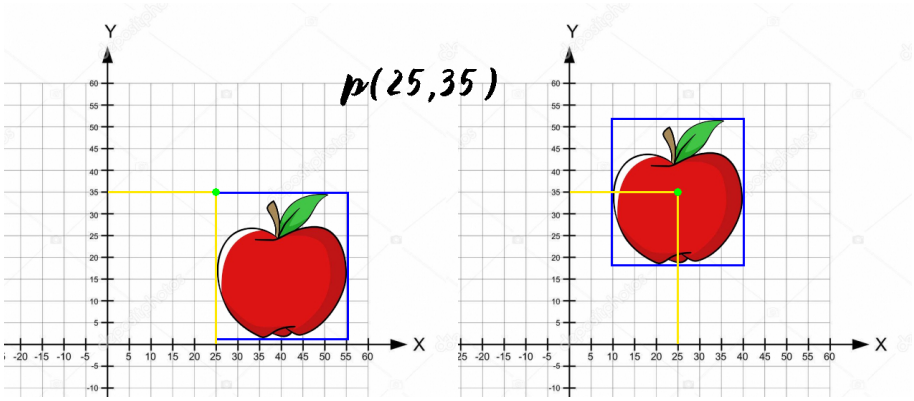

Figura 4.8: Ejemplo visual del problema del posicionamiento

A lo largo de su carrera como programador, el autor de este trabajo ha empleado ambos sistemas y, sin duda, considera que el segundo método resulta más natural y, por tanto, más cómodo para trabajar. Por ello, ese será el método que se emplee para posicionar los elementos.

4.3. Sistema de gestión de assets

En la mayoría de herramientas y motores la gestión de los assets se realiza de manera individualizada para cada tipo. De esta manera, para trabajar con gráficos se proporciona una batería de funciones, para trabajar con audios se proporciona otra batería de funciones y así para cada tipo de recurso [14].

Esto trae consigo algunos conflictos. El más relevante para este trabajo es el hecho de que, si se realiza de ese modo, el desarrollador que haga uso de esta librería no podrá crear sus propios assets integrados con el resto de funcionalidades; el cual es uno de los objetivos planteados.

Afortunadamente, algunos de los motores más relevantes del mercado emplean sistemas de gestión de assets unificados que permiten gestionarlos como *Assets*, en abstracto, de manera que al darles uso cuando sea cuando importe el tipo de asset que sea [15]. Aunque por motivos comerciales limitan mucho la flexibilidad del sistema dificultando poder crear tus propios assets, para tratar de venderte una extensión del producto que incluya tus necesidades concretas.

De este modo, inspirado en ese modo más flexible de gestionar los assets, se ha decidido diseñar un sistema similar basado en una clase base llamada Asset, de la que heredarán las diferentes clases de assets.

4.3.1. Clase *Asset*

Al ser esta clase aquella de la que hereden todos los assets debe contener los atributos y miembros necesarios para dicha tarea.

El problema de la sobrecarga de miembros virtuales

La idea más intuitiva sería proporcionar a esta clase de un miembro virtual que pudiera sobrecargarse en cada clase hijo especificando qué recurso va a devolver. De este modo, el asset de gráfico 2D sobrecargaría la función para devolver un gráfico y el asset tipográfico lo haría para devolver una fuente tipográfico.

Desgraciadamente, algunos lenguajes orientados a objetos, por el modo en el que resuelven las sobrecargas, no permiten hacer esto. En C++, en la mayoría de compiladores, da error de compilación [16].

Almacenamiento de tipos y casteo

La solución al problema presentado previamente se encuentra con la posibilidad que nos dan los lenguajes orientados a objetos de almacenar tipos y nombres de clases y usarlos para castear objetos. De este modo, la clase base *Asset* puede contener un atributo llamado *element_type* donde almacenar el tipo en el que se debe reconvertir el objeto a devolver.

Si a ese atributo se acompaña de una función miembro que se encargue de castear el recurso a lo que se le haya solicitado y de una función miembro abstracta destinada a que las clases hijo puedan devolver el recurso que consideren, podemos lograr sortear el problema de no poder sobrecargar miembros virtuales.

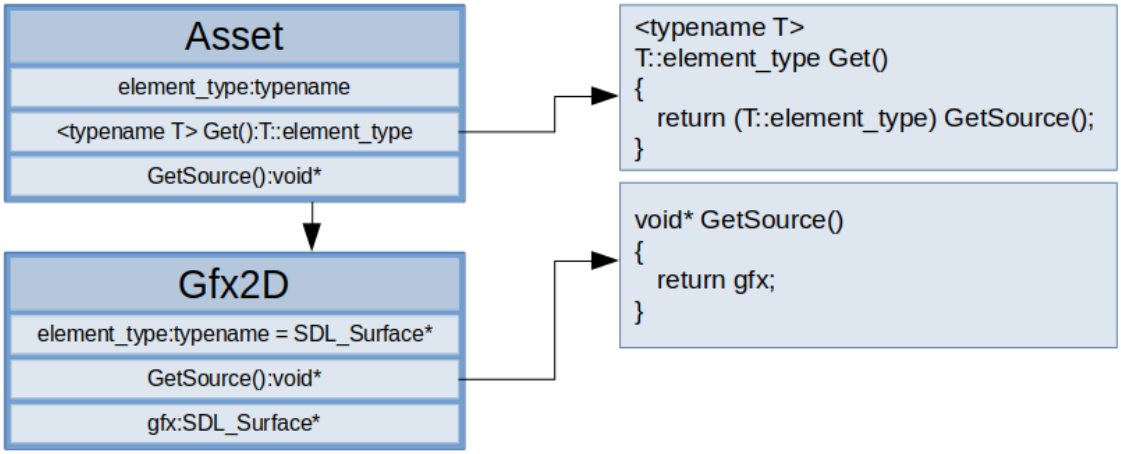

Figura 4.9: Ejemplo del diseño propuesto para Asset

Diseño final de *Asset*

Por lo tanto, y tras todo lo expuesto anteriormente, la clase *Asset* quedaría del siguiente modo:

- *id* [atributo]: identificador único del recurso.
- *element_type* [atributo]: nombre del tipo o clase a la que habrá que reconvertir el recurso devuelto.
- *GetSource()* [función]: función abstracta que todas las clases derivadas deberán usar para devolver el recurso que contengan.
- *Get()* [función]: función a la que se invocará para obtener el recurso que se encargará de invocar a *Getsource()* y recastear lo que esta función devuelva.

4.3.2. Los paquetes de recursos

Para facilitar al usuario el uso de los recursos, se propone un diseño que considera que estos se organizan en paquetes de recursos. Esto evita tener que cargar y descargar uno a uno cada recurso y da la opción a proporcionar formas de configurar los paquetes.

Y es que la configuración de los paquetes de recursos resulta de lo más práctico. Un ejemplo de su practicidad es la siguiente:

Supongamos que el usuario está desarrollando un pequeño juego de disparos y dispone de un paquete de sonidos donde se encuentran los efectos de sonido. Teniendo en cuenta que los efectos de sonido son el mismo tipo de archivo que los de la banda sonora, debe poder indicar al gestor de recursos que son efectos de sonido y no pistas de audio[3]. Introduciendo en ese paquete un fichero de configuración que lo indique bastaría para que este haga esa diferenciación, ahorrando dolores de cabeza al usuario.

Además, al ser el paquete de recursos el responsable de cargar todos los recursos que contenga, deberá disponer de una forma de relacionar las extensiones de los archivos a cargar con el tipo de recurso que son (teniendo en mente que un usuario podrá crear sus propios métodos). De este modo, se propone emplear un mapa desordenado que permita asociar extensiones de ficheros con punteros a funciones especializadas en la carga del recurso que corresponda.

De este modo, las entradas *".png"*, *".jpg"* y *".bmp"* señalarían a una función que cargue recursos gráficos y las entradas *".wav"*, *".mp3"* y *.ogg* señalarían a una función que cargue recursos de sonido, por poner dos ejemplos.

Este paquete también deberá disponer de una función que, dado un identificador, le permita obtener el recurso solicitado.

Así, la clase *AssetList* quedaría de la siguiente manera:

- *id* [atributo]: identificador único del paquete de recursos.
- *assets* [atributo]: mapa desordenado que asocia cada recurso con su identificador.
- *loadFunctions* [atributo estático]: mapa desordenado estático que asocia extensiones de fichero con punteros a funciones con las que cargar los recursos.

[3]Debido al funcionamiento por canales de los sistemas de sonido de los sistemas operativos más comunes, las librerías de sonido suelen tratar de manera diferente a los efectos de sonido y a las pistas de audio. Los primeros tienden a ser pequeños fragmentos de audio cuya calidad de reproducción no suele ser prioritaria mientras que los segundos suelen ser audios largos, muchas veces en bucle, cuya calidad de reproducción es fundamental.

- *Get(id)* [función]: función que invoca a la función *Get()* del elemento solicitado.
- *isLoaded(id)* [función]: función que comprueba si en la lista de recursos existe un recurso con el identificador indicado.
- *Size()* [función]: función que indica cuántos recursos contiene.
- *addResource(extension, function* [función]: función que registra en *loadFunctions* una nueva asociación, asociando la extensión indicada a la función indicada.

4.3.3. El gestor de recursos

Para finalizar con esta sección, se debe incluir una clase más; cuya tarea será la de combinar las clases anteriormente descritas para proporcionar un sistema unificado y monolítico de gestión de recursos.

Si la clase *AssetList* contiene una lista de *Assets*, la clase *AssetManager* contendrá una lista de *AssetLists*. De esta manera, el usuario podrá acceder a cualquier recurso invocando al *AssetManager* indicando el identificador del recurso y el del paquete donde se encuentra.

Debido al diseño de *AssetManager* como una clase que unifica el uso de los recursos, todos sus miembros serán estáticos; siendo estos los siguientes:

- *packages* [atributo]: lista desordenada de paquetes de recursos.
- *loadAssetPackage(path)* [función]: función que carga el paquete indicado.
- *unloadAssetPackage(id)* [función]: función que descarga el paquete indicado.
- *Size()* [función]: función que indica cuántos paquetes de recursos tiene cargados.
- *packagesize(id)* [función]: función que invoca a la función *Size()* del paquete indicado.
- *isLoaded(package, [resource]* [función]: función que indica si el paquete indicado está cargado o, en caso de proporcionar el segundo parámetro, si en el paquete indicado existe el recurso indicado.
- *Get(package, resource)* [función]: función que invoca a la función *Get(id)* del paquete indicado solicitando el recurso indicado.
- *addResource(extension, function)* [función]: función que invoca a la función homónima de *AssetList* para centralizar todas las operaciones con assets en esta clase.

4.4. Detección de colisiones

Existen muchos métodos para detectar colisiones y, en gran medida, es porque dependen de qué es lo que se quiere colisionar. No es lo mismo hacer colisionar dos cuadrados que un cuadrado y un círculo. Tampoco es lo mismo colisionar dos cuadrados alineados que tener rotado alguno de ellos.

Para este trabajo, y entendiendo que podría hacerse un trabajo entero dedicado exclusivamente a este tema, se ha optado por emplear dos tipos de colisión: colisiones AABB y colisiones Pixel-Perfect.

4.4.1. Colisiones AABB

La detección de *colisiones AABB*; o, lo que es lo mismo, la detección de colisión entre dos rectángulos alineados con los ejes x e y [17]. Este método consiste en comprobar la separación de los dos rectángulos en ambos ejes.

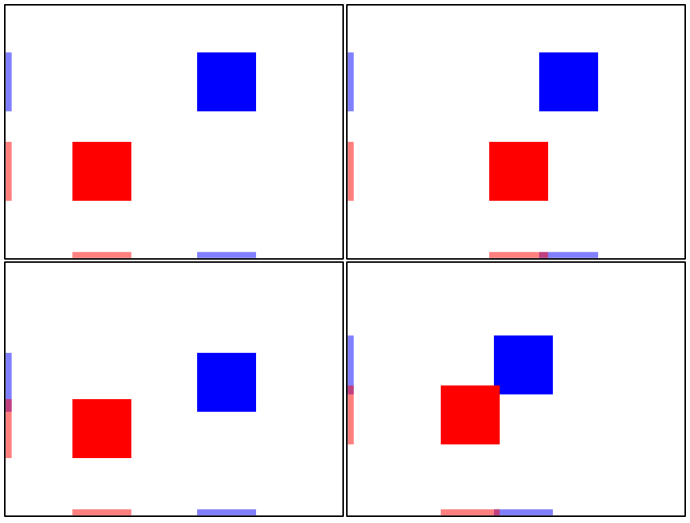

Figura 4.10: Las proyecciones en los ejes x e y revelan si se da la colisión

4.4.2. Colisiones Pixel-Perfect

El método de detección de colisiones AABB es un método sencillo y, dado su bajo costo computacional, muy potente. Para muchas situaciones es más que suficiente. En algunas ocasiones, sin embargo, es necesario hacer uso de un método más preciso.

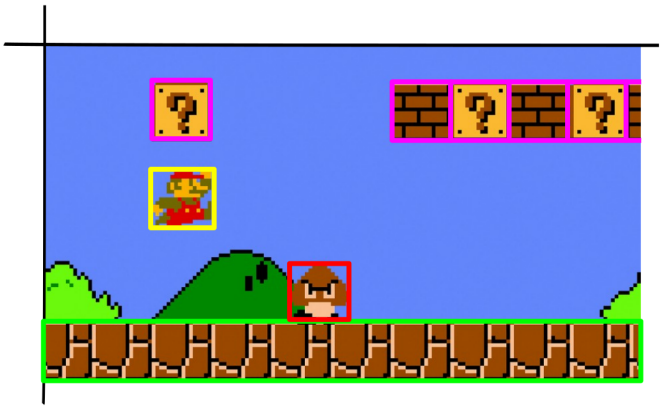

Figura 4.11: Ejemplo de juego que aprovecha las colisiones AABB

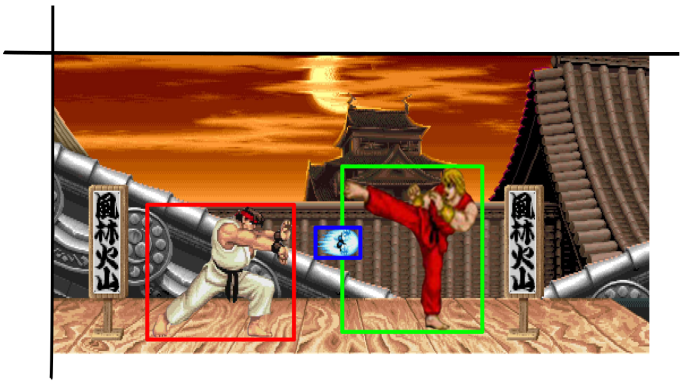

Figura 4.12: Ejemplo de juego al que perjudican las colisiones AABB

El método Pixel-Perfect permite detectar las colisiones píxel a píxel, garantizando una precisión milimétrica. Esto, evidentemente, trae como contraparte un aumento en el coste computacional puesto que, en realidad, consiste en realizar una comprobación de colisión AABB para cada píxel [18].

Precisamente por esta contraparte, se va a desarrollar como método complementario al método AABB. De manera que si se desea comprobar una colisión mediante Pixel-Perfect, primero empleará el método AABB y sólo en caso de dar positivo, y aprovechando algunos de los datos ya obtenidos por AABB, se empleará el método Pixel-Perfect.

4.4.3. Obteniendo las dimensiones del proceso

Con el fin de poder calcular las colisiones entre procesos, primero será necesario obtener las dimensiones de los mismos. Esto implica que deberá añadirse al diseño de *Process* una función miembro llamada *getDimensions()* que realice este cálculo.

Entendiendo que los procesos pueden rotar y cambiar de tamaño, estas dimensiones corresponderán con el rectángulo paralelo al plano *xy* más pequeño que pueda contener al gráfico del proceso tras aplicarle la escala y rotación.

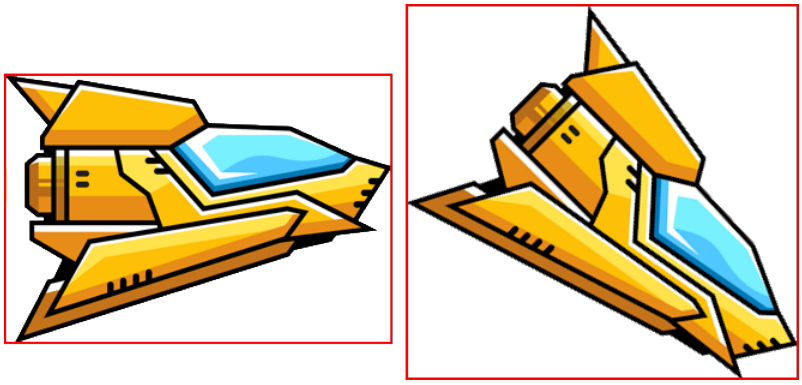

Figura 4.13: El tamaño de la caja se ajusta al tamaño y rotación

Para ello, se calcularán los vértices del proceso tras aplicar la escala y se rotarán en el plano *xy*. Una vez obtenidos los nuevos vértices el ancho se corresponderá con la diferencia entre la *x* más grande y la *x* más pequeña y el alto tendrá la misma correspondencia con la *y*.

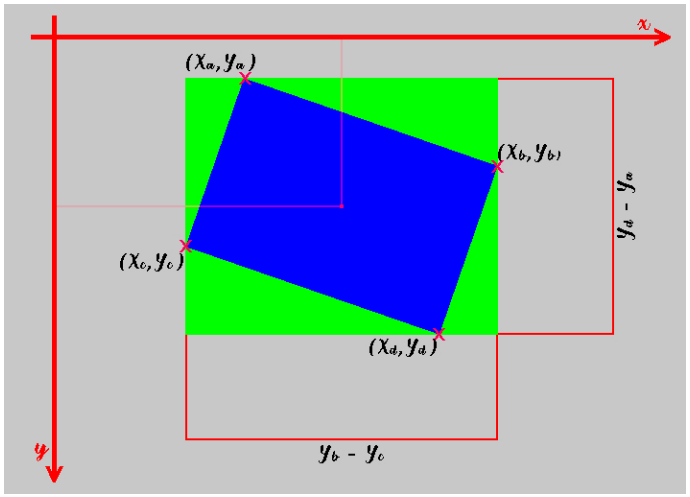

Figura 4.14: Esquema de las dimensiones del proceso

4.4.4. Calculando la intersección entre dos procesos

Tras obtener las dimensiones de los dos *Process* implicados en el cálculo, se obtendrá la proyección de ambas cajas en los ejes *x* e *y*. Si en ambos ejes se solapan, es que se da la colisión entre ambos objetos.

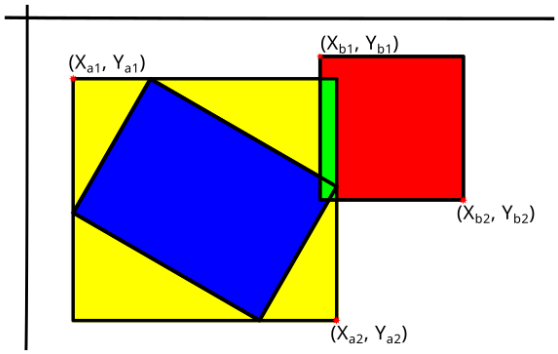

Figura 4.15: Área de colisión entre dos procesos

Si hay colisión, obtener el área de intersección entre ambos procesos es trivial. Atendiendo al ejemplo de la figura anterior, el vértice superior izquierdo del área sería (x_{b1}, y_{a1}), el ancho sería $x_{a2} - x_{b1}$ y el alto sería $y_{b2} - y_{a1}$.

4.4.5. Detección píxel a píxel de la colisión

Si la colisión que se desea hacer es Pixel-Perfect, será necesario continuar con las comprobaciones tras haber obtenido el área de colisión. En el ejemplo anterior podemos ver que el área de colisión es grande, pero en la detección píxel a píxel, el área real de colisión se reduce enormemente.

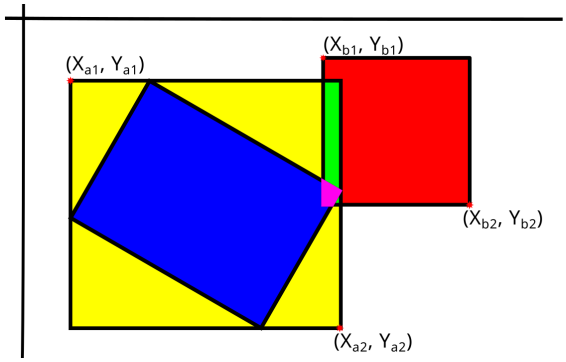

Figura 4.16: Área de colisión real entre dos procesos

Para realizar esta comprobación, recorreremos cada píxel del área solicitando a cada proceso las coordenadas de su gráfico a las que corresponde esa coordenada. Necesitaremos, por tanto, añadir a la clase *Process* una función llamada *checkPoint(x,y)* que se encargue de calcular esta información.

Tras obtener esta información, si en ambos el pixel se encuentra dentro de los límites de

su gráfico y éste no es transparente, se considerará que hay colisión. Si tras recorrer todo el área no se ha encontrado ninguna colisión, se considerará que no la ha habido.

4.4.6. La nueva función *checkPoint(x,y)*

El trabajo de esta función será la de recibir unas coordenadas en el sistema de referencia global y encontrar a qué coordenadas en el sistema de referencia de su gráfico corresponden. Si corresponde a una posición dentro del gráfico, devolverá esas coordenadas; si no, devolverá la posición $(-1,-1)$.

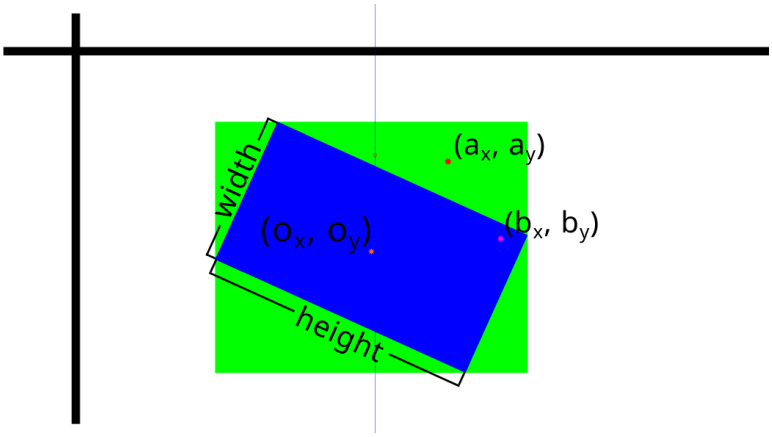

Figura 4.17: Ejemplo de situación en la que usar *ckeckPoint(x,y)*

Para ello, primero resituará las coordenadas considerando que el centro del proceso es el origen de coordenadas. A continuación, aplicará una rotación en sentido contrario al aplicado al proceso, para alinear el sistema de referencia con los ejes x e y globales. Luego, invertirá la escala y, por último, desplazará el punto para que el origen de coordenadas sea la esquina superior izquierda del proceso.

De manera que los dos puntos del ejemplo anterior quedarían de la siguiente manera:

- $a' = (\frac{(a_x-o_x)*cos(\theta)+(a_y-o_y)*sen(\theta)}{scale_x}, \frac{(a_y-o_y)*cos(\theta)-(a_x-o_x)*sen(\theta)}{scale_y})$
- $b' = (\frac{(b_x-o_x)*cos(\theta)+(b_y-o_y)*sen(\theta)}{scale_x}, \frac{(b_y-o_y)*cos(\theta)-(b_x-o_x)*sen(\theta)}{scale_y})$

Es importante recordar que, puesto que la rotación de los procesos es antihoraria[4], la rotación a aplicar es en sentido horario [19].

[4]Desde nuestra perspectiva ante el monitor, parece rotación en sentido horario. Pero se debe recordar que el eje y crece hacia abajo, por lo que la rotación debe ser antihoraria para que en pantalla se vea así

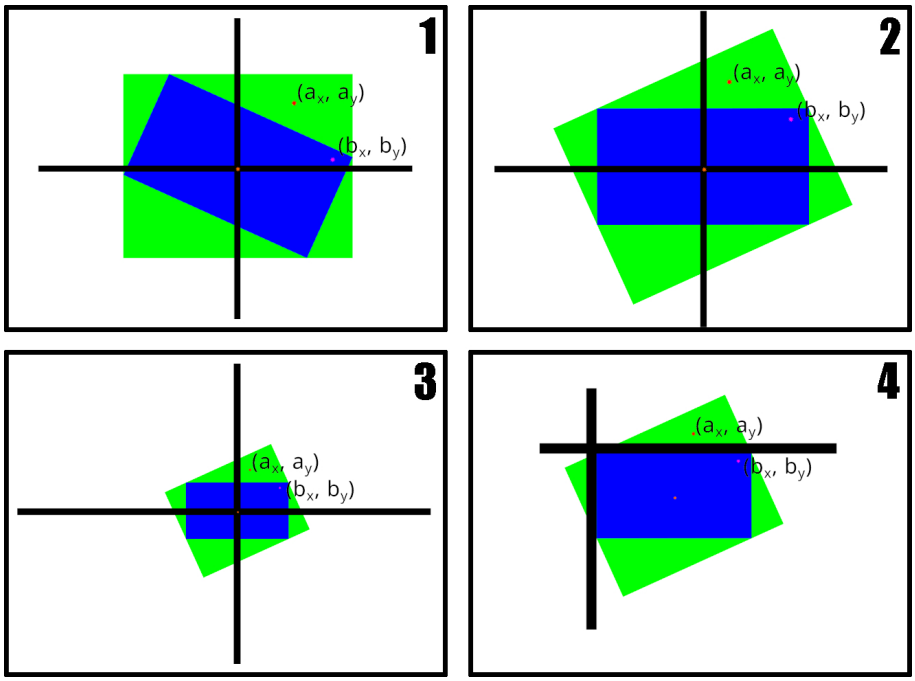

Figura 4.18: Representación visual de los pasos a seguir

4.4.7. El espacio de nombres *Collision*

El diseño del sistema de colisiones se va a realizar mediante un espacio de nombres. Este contará con tres funciones:

- *getIntersection(boundA, boundB)*: esta función comprobará si se da la colisión entre las dos cajas de colisión indicadas. Estos dos parámetros serán estructuras, cuya definición dependerá de cada implementación, que contendrán las coordenadas *x* e *y* de la esquina superior izquierda de la caja y sus dimensiones.

- *checkCollision(entityA:Process, entityB:Process, type)*: esta función se encargará de obtener las dimensiones de los dos *Process* recibidos, invocará a *getIntersection(boundA, boundB)* con las dimensiones obtenidas y comprobará si ambos procesos colisionan de acuerdo al tipo de colisión indicado.

- *checkCollisionByType(entityA:Process, type)*: esta función obtendrá la lista de todos los procesos cuyo tipo sea el indicado y comprobará si el proceso proporcionado colisiona con alguno. Devolverá el puntero al primer proceso con el que detecte colisión.

4.5. Detección de entrada

Todas las librerías suelen proporcionar dos métodos de detección de entrada: por interrupción y por encuesta. El método por interrupción suspende temporalmente la ejecución de un proceso, para pasar a ejecutar una subrutina de servicio de interrupción y, una vez finalizada dicha subrutina, reanuda la ejecución del programa. El método por encuesta, por el contrario, el programa no se ve interrumpido. En vez de eso, integra en su bucle de ejecución la comprobación del estado de las señales y llama a las subrutinas pertinentes.

Aunque ambas opciones son habituales en el desarrollo de videojuegos, el método de encuesta se ajusta mejor al diseño propuesto en este trabajo. Tal y como se ha planteado el pipeline (ver figura 4.1) se busca que sea durante el *Update*/*FixedUpdate* de los procesos donde estos comprueben si la tecla que activa su comportamiento se ha presionado.

Puesto que el objetivo es abstraer a los usuarios de los sistemas dependientes de la implementación, se propone una estructura que almacene el estado de cada entrada aceptada. De este modo, una clase llamada Input se encargaría de actualizar esta información dejándola disponible para su lectura por parte de los procesos.

4.5.1. Miembros de la clase Input

Al ser una clase destinada a que todos los procesos puedan leer su información y al ser esta información única para todo el sistema, la clase será singleton y todos sus miembros serán estáticos.

- *Update()* [función]: esta función, la única de la clase, estará destinada a actualizar el estado de todas las estructuras de la clase.
- *key* [atributo]: array donde se almacenará el estado de cada una de las teclas del teclado, empleando un enumerador para asociar etiquetas con posiciones. De esta manera Input::key[_a] permitirá acceder al estado de la tecla *"a"*.
- *mouse* [atributo]: estructura que almacenará un Vector2 con la posición del ratón y un array donde se almacenará el estado de cada uno de los botones del ratón, empleando un enumerador para asociar etiquetas con posiciones.

4.6. Sistema de sonido

La banda sonora y los efectos de sonido son una parte importante dentro de los videojuegos. Es una sensación habitual sentir que un juego no está completo hasta que se introducen efectos de sonido. Aunque los motores más avanzados suelen ofrecer sistemas de posicionamiento de sonido, haciendo que estos ajusten su volumen a su posición respecto al personaje e, incluso, que apliquen filtros al audio que reproducen para simular las condiciones de escucha, se ha considerado que para este trabajo, y entendiendo que podría desarrollarse un trabajo entero dedicado enteramente a esta sección, se va a desarrollar la funcionalidad más simple posible.

Por ello, se va a diseñar un sistema que pueda reproducir efectos de sonido en diferentes canales así como una banda sonora en un canal dedicado. Además, este sistema permitirá indicar si la reproducción será o no en bucle y permitirá realizar un efecto de transición entre diferentes canciones de la banda sonora.

4.6.1. El gestor de sonidos

El gestor de sonidos será la clase responsable de enviar los audios a reproducir al sistema de reproducción de audio del sistema operativo. Contendrá tres funciones estáticas.

- *playFX(clip, [channel])*: esta función será la encargada de reproducir el efecto de sonido indicado en el canal indicado. En caso de no indicar un canal, se reproducirá en el primer canal libre.
- *playSong(song, [loop])*: esta función será la encargada de reproducir la canción indicada. Si se indica que se desea que se reproduzca en bucle, señalará al sistema de sonido que la reproducción sea en bucle.
- *changeMusic(song)*: la función anterior reproduce música y si había una música ya sonando, la detiene para reproducir la nueva. Esta función, proporciona la funcionalidad de hacer ese cambio más suavemente, reduciendo primero el volumen de la canción activa y luego aumentando poco a poco el de la nueva canción. Para ello, se apoyará en una clase llamada *SoundTransition*.

Para proporcionar una mejor integración con el gestor de recursos, se proporcionará una sobrecarga de estas tres funciones sustituyendo el parámetro *clip/song* por los parámetros

package y *id* para que sean las propias funciones las que accedan al gestor de recursos a buscar el audio concreto.

4.6.2. SoundTransition

Si se quiere que la transición entre dos canciones sea un proceso suave, no es algo que pueda hacerse durante la ejecución de una función concreta. Por eso, se ha decidido emplear una clase que, integrada con el gestor de tareas, su única funcionalidad sea comprobar si la canción que se está reproduciendo es la que corresponde.

Si no es el caso, reducir su volumen poco a poco y, al llegar este a cero, cambiar la canción e ir subiendo el volumen poco a poco hasta restaurarlo.

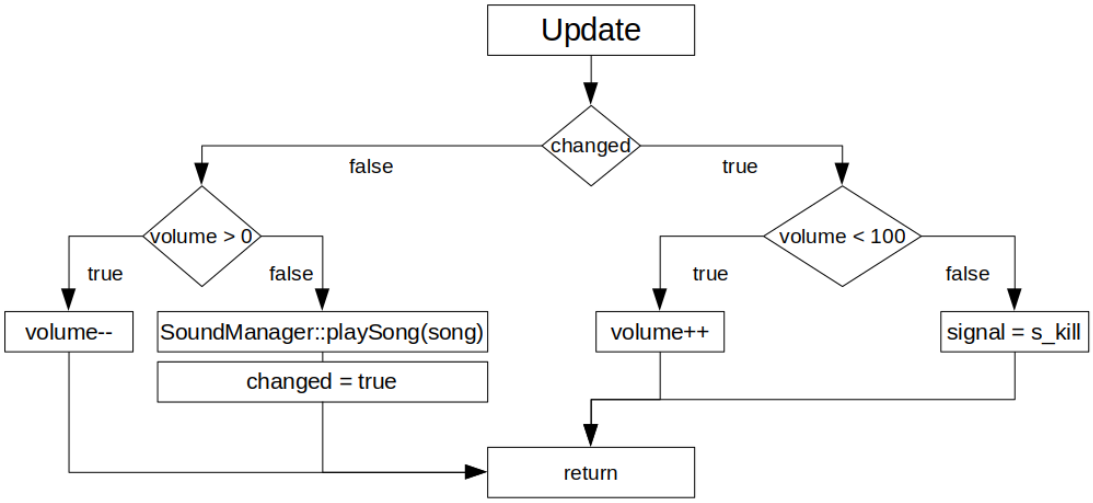

Figura 4.19: Diagrama de flujo de la functión Update de SoundTransition

5

Conclusiones y líneas futuras

La herramienta diseñada en el presente trabajo es una base funcional y suficiente para cubrir los objetivos planteados pero no es, ni de lejos, todo lo que podría ser. El diseño presentado ha sido este en aras de no extender el trabajo más de lo necesario; pero en futuros trabajos podría ampliarse añadiendo más funcionalidades.

Habría aportado un punto muy interesante haber podido contar con alumnos de primero que pudieran hacer uso de esta herramienta para poder aportar resultados de su uso. Desgraciadamente, esta posibilidad se escapa por completo del alcance del proyecto.

De cara a futuros trabajos que amplíen esta herramienta, se proponen algunas funcionalidades que serían interesantes:

- Ampliar la clase Input para detectar flancos, hacer uso de joysticks y mandos e, incluso, permitir la creación de ejes de entrada[20].
- Diseñar un sistema de debug y log que permita al desarrollador interactuar en segundo plano con su aplicación mientras se ejecuta. Esta herramienta es especialmente interesante si se quiere que la aplicación responda a comandos de debug puesto que mientras atienda a la entrada standard no refrescará la pipeline.
- Añadir la posibilidad de mover el origen de los procesos (que en este diseño está en su centro) para permitir al usuario colocarlo donde considere y añadir un sistema de cámara que permita al sistema recolocar a todos los procesos en el lugar que les corresponde.

- Mejora del sistema de renderizado ahondando en la eficiencia. Por ejemplo, estudiando qué elementos van a estar ocultos por otros procesos o fuera de pantalla y, por tanto, no necesitan ser enviados al sistema de dibujado.
- Mejora del sistema de colisiones ahondando en la eficiencia.
- Apoyándose en el sistema de colisiones añadir un sistema de sólidos que permita crear procesos que interactuen entre ellos.
- Diseñar una clase *Clock* que permita obtener información relacionada con el tiempo de ejecución, reloj del sistema y funcionalidades relacionadas.

Bibliografía

[1] Unity Technologies, "Unity Manual: Orden de ejecución de las funciones de eventos." https://docs.unity3d.com/es/current/Manual/ExecutionOrder.html.

[2] M. A. Iruretagoyena, "Ejemplo de aplicación audiovisual con múltiples ventanas." https://video.hardlimit.com/w/opzf2YrTTgrcK33yA6sC1J.

[3] A. C. Carrasco, "¿Qué es un motor de videojuegos?." https://blogs.upm.es/observatoriogate/2018/07/04/que-es-un-motor-de-videojuegos, 2018.

[4] Unity Technologies, "Portal oficial del motor de videojuegos Unity." https://unity.com.

[5] Epic Games, Inc., "Portal oficial del motor de videojuegos Unreal Engine." https://www.unrealengine.com/en-US/.

[6] YoYo Games Ltd., "Portal oficial del motor de videojuegos Unreal Engine." https://gamemaker.io/es.

[7] Juan Linietsky, Ariel Manzur et al., "Portal oficial del motor de videojuegos Godot." https://godotengine.org/.

[8] Juan Jose Ponteprino (SplinterGU), "Portal oficial del lenguaje de videojuegos BennuGD." https://www.bennugd.org.

[9] un4seen developments, "Portal oficial de la librería BASS." https://www.un4seen.com/bass.html.

[10] The Minetest Team, "Wiki oficial para desarrolladores del motor Minetest." https://dev.minetest.net.

[11] Unity Technologies, "Unity Scripting Reference: GameObject." https://docs.unity3d.com/ScriptReference/GameObject.html.

[12] cplusplus.com, "std::unordered_map." https://www.cplusplus.com/reference/unordered_map/unordered_map/.

[13] cplusplus.com, "std::unordered_multimap." https://www.cplusplus.com/reference/unordered_map/unordered_multimap/.

[14] YoYo Games Ltd., "Gestión de activos en GameMaker." https://manual-es.yoyogames.com/#t=GameMaker_Language%2FGML_Reference%2FAsset_Management%2FAsset_Management.htm.

[15] Unity Technologies, "Unity Scripting Reference: Resources." https://docs.unity3d.com/ScriptReference/Resources.html.

[16] Miguel Albors Iruretagoyena, "Ejemplo de lenguaje en el que no es posible sobrecargar un miembro virtual." http://cpp.sh/2grxl.

[17] Lazy Foo' Productions, "Collision detection." https://lazyfoo.net/tutorials/SDL/27_collision_detection/index.php.

[18] Lazy Foo' Productions, "Per-pixel Collision Detection." https://lazyfoo.net/tutorials/SDL/28_per-pixel_collision_detection/index.php.

[19] Prof. Sheila Widnall, "Vectors, Matrices and Coordinate Transformations." https://ocw.mit.edu/courses/16-07-dynamics-fall-2009/resources/mit16_07f09_lec03/.

[20] Unity Technologies, "Unity Scripting Reference: Input Axis." https://docs.unity3d.com/ScriptReference/Input.GetAxis.html.

Anexos

A

Clase Engine

Una clase que podría resultar muy interesante para el uso de esta herramienta sería una que, a modo interfaz, aglutinara todos los módulos y permitiera administrar toda la librería desde ella. El motivo por el que no se incluye es porque se ha considerado que, ya que se busca ayudar a reforzar las habilidades programáticas de los usuarios, aporta más no incluirla y dejar que sea el usuario quien dedique tiempo a diseñarla.

Sin embargo, para poder realizar las pruebas de implementación, se ha llevado a cabo el trabajo de diseño e implementación de la misma.

La implementación completa de la herramienta, así como la implementación de algunas clases adicionales no descritas en este trabajo, se encuentra en el siguiente anexo.

```
class RF_Engine
{
  public:
    static RF_Engine* instance;

    RF_Engine(bool debug=false);
    virtual ~RF_Engine();
   static void Run();

   //Windows
   static RF_Window* addWindow(string i_title = "",
               int i_width = 20, int i_height = 20,
```

```
13              int i_posX = SDL_WINDOWPOS_CENTERED,
14              int i_posY = SDL_WINDOWPOS_CENTERED,
15              int i_windowMode = SDL_WINDOW_OPENGL,
16              int i_rendererMode = SDL_RENDERER_ACCELERATED);
17
18     static RF_Window* getWindow(int id);
19    static int getWindow(RF_Window *window);
20    static RF_Window* MainWindow();
21    static void MainWindow(int id);
22    static void MainWindow(RF_Window *_window);
23
24    static void closeWindow(int id);
25    static void closeWindow(RF_Window *_window);
26
27    //Tasks
28    template<class T = RF_Process>
29    static string newTask(string father="", RF_Window *_window=↩
      MainWindow())
30    {
31      assert(instance != nullptr);
32      return RF_TaskManager::instance->newTask<T>(father, _window);
33    }
34
35     template <class T = RF_Process>
36    static T* getTask(string id)
37    {
38      static_assert(std::is_base_of<RF_Process, T>::value, "T must ↩
      derive from RF_Process");
39      return reinterpret_cast<T*>(RF_TaskManager::instance->getTask(↩
      id));
40    }
41
42    static bool existsTask(string id);
43
44    //Señales
45    static void sendSignal(string taskID, int signal);
46    static void sendSignal(RF_Process* task, int signal);
47    static void sendSignalByType(string type, int signal);
48
49    //FPS
50    static const int Fps();
```

```
51    static void Fps(int _fps);
52
53    //Reloj
54     RF_Time Clock;
55
56    template<typename T>
57    static void Debug(T param)
58    {
59      if(!isDebug){return;}
60       stringstream text;
61       text << "[" << SDL_GetTicks() << "]: " << param;
62       RF_DebugConsoleListener::Log(string(text.str()));
63    }
64
65    static bool& Status(){return isRunning;}
66
67    template<typename T>
68    static void Start(bool debug = false)
69    {
70      new RF_Engine(debug);
71      newTask<T>();
72
73      do
74      {
75        Run();
76      } while(Status());
77
78      delete(instance);
79    }
80
81     private:
82      float fps = 0.1/60.0;
83      float wait_to_draw = 0.0;
84      static bool isDebug, isRunning;
85      int mainWindow = -1, windowCount = 0;
86      unordered_map<int, RF_Window*> windowList;
87 };
```

B

Implementación en C++

Para comprobar que el diseño funcionaba adecuadamente se ha implementado en C++. El código puede encontrarse en el siguiente repositorio: https://gitlab.com/yawin/RosquilleraReforged.git

Aunque el proyecto existe desde antes de comenzar este trabajo, se ha refactorizado para cumplir con el diseño aquí presentado.

Además, se han desarrollado algunos ejemplos de uso, los cuales permiten comprobar que se cumple con la facilidad de uso que se buscaba: https://gitlab.com/yawin/rosquillera-examples

En caso de no poder compilar o ejecutar, esta es la captura de la ejecución del ejemplo "DemoOrDie": https://video.hardlimit.com/w/5Y6GtycVK3z3wwgJ2c5r2H