UNIVERSITAT POLITÈCNICA DE VALÈNCIA

CAMPUS D'ALCOI

Diseño y creación de un motor de aplicaciones interactivas en tiempo real usando multithreading

MEMÒRIA PRESENTADA PER:

Sergio Tortosa Benedito

GRAU DE INGENIERÍA INFORMÁTICA

Convocatòria de defensa: *Septembre de 2017.*

Índice

0. Conocimientos previos

0.1. Abreviaciones

AAA*:* pronunciado “triple A” es una designación para aquellos videojuegos desarrollados por grandes compañías que pretenden ofrecer juegos que suelen contar con una gran artística y técnica, en otras palabras AAA es una designación de alta calidad que se le puede dar a un juego, sin embargo, esta calificación (a diferencia de otras) no es oficial, no existe ningún organismo encargado de elegir que juegos pueden ser llamados como AAA y cuáles no.

API: *Application Programming Interface*, interfaz de programación de aplicaciones. La *API* representa la comunicación entre componentes de *software*, referencia al conjunto de llamadas que el programa puede hacer frente a una librería/biblioteca o algún otro agente (mensajes a sistemas remotos también pueden considerarse *API*).

CAS: *Compare and Swap*, compara e intercambia. Esta instrucción forma parte del repertorio de herramientas para el *multi-threading lock-free*, su función consiste en comparar un valor en memoria con otro y si son iguales, realizar un cambio. Esta instrucción es sumamente útil para asegurar que ciertas acciones son correctas (como valores basados en valores anteriores o valores que se esperaban …).

CPU: *Central Processing Unit*, unidad de proceso central, también se le conoce como microprocesador. Se nombra así al chip encargado de ejecutar las órdenes de un ordenador y de comandar al resto de procesadores.

DSL: *Domain Specific Language*, lenguaje de dominio específico. Se refiere a un lenguaje de programación creado para realizar una tarea determinada (construcción de binario, páginas webs …), por lo general, ayudan en la tarea en la que trabajan al permitir al programador expresar sus operaciones en una forma expresiva y declarativa (definiendo cuál es el resultado en lugar de como conseguirlo). Un *DSL* puede presentarse como un lenguaje propio e independiente o como parte de una extensión para un lenguaje de programación. Debe notarse que no todos los lenguajes son apropiados

para esto y es necesario un lenguaje flexible (a través de metaprogramación en tiempo de ejecución o introspección en tiempo de ejecución).

E/S: Entrada/Salida, *I/O* , *Input/Output* en inglés. La E/S representa la comunicación que tiene un programa con el exterior. Aunque E/S suele especificar la entrada salida estándar (siendo esta típicamente la consola) puede usarse para cualquier tipo de comunicación (redes, disco interno, comunicación con el usuario …).

FIFO: *First In First Out*, primero en entrar, primero en salir. Dentro del campo de las estructuras de programación se considera que una estructura como *FIFO* si almacena valores y devuelve estos valores en el mismo orden.

GCC: *GNU Compiler Collection*, colección de compiladores de *GNU*. Se trata de un conjunto de compiladores para distintos lenguajes, desarrollados como software libre por *GNU* bajo la licencia *GPL*, usados en gran cantidad de sistemas operativos de software libre. Aunque el nombre *GCC* se refiere al conjunto de compiladores, corrientemente se usa este acrónimo para referirse al compilador de C/C++.

GNU: Aunque se le reconoce como acrónimo (un acrónimo recursivo de hecho, GNU is Not Unix, GNU No es Unix) su nombre proviene del animal Ñú. Es un proyecto que pretende impulsar el uso de software libre y su difusión. Se encargan de realizar un sinfín de herramientas que son usadas en otros proyectos, entre ellos Linux (recordemos que Linux como tal no es más que un núcleo y como tal no puede funcionar por sí mismo), de ahí que el nombre oficial de los sistemas Linux usados hoy en día sea GNU/Linux.

GPL: GNU Programming License, licencia de programación de GNU. Para impulsar el desarrollo de software libre, se diseñó está licencia de programación que protegía al programador frente a ciertos actos jurídicos al tiempo que asegura que todo código desarrollado bajo esta licencia permanezca libre.

JIT: *Just In Time*, justo a tiempo. En general cuando nos referimos a *JIT* nos referimos a compilación *JIT.* Este tipo de compilación se sitúa entre medias de la compilación tradicional (*AOT, Ahead Of Time*, más allá en el tiempo) que realiza toda la compilación de una vez y ya no realiza más compilaciones y

la interpretación, que no realiza ninguna compilación. El *JIT* interpreta el código y para acelerar dicha ejecución compilará en el momento el código, esto permite una gran flexibilidad puesto que el compilador tiene más información sobre la ejecución en tiempo de compilación, siendo una gran opción para lenguajes dinámicos o entornos que requieran de carga de código arbitrario. Este tipo de ejecución es usada por lenguajes como: *Java*, C#, *Javascript* o *Lua*.

LIFO: *Last In First Out*, último en entrar primero en salir. Cuando se refiere una estructura *LIFO* se refiere a una estructura capaz de almacenar datos y devolverlos en el orden inverso al introducido.

MSC: *Micrososft C++*. Es el compilador de C++ de *Microsoft*, propietario y únicamente disponible para su plataforma.

PBR: *Physically Based Rendering*, renderizado físicamente basado. El *PBR* se define como concepto, en concreto el concepto de observar atentamente como se aprecia el mundo real y trasladarlo al mundo virtual, incluyendo todo tipo de detalles con tal de alcanzar el fotorrealismo.

RAM: *Random Access Memory*, memoria de acceso aleatorio.

SDL[1]: *Simple DirectMedia Layer*, capa *DirectMedia* simple. Librería “por defecto” para la creación de programas multimedia con un sinfín de capacidades: gráficos, sonido, manipulación de ventanas, renderizado de texto, carga de archivos … Usada por un gran número de proyectos incluyendo varios motores de videojuegos comerciales.

SFML[2]: *Simple and Fast Media Library*, librería para multimedia simple y rápida. Es una librería considerada como alternativa a SDL. Se diferencia principalmente por su uso de C++ frente a C que usa *SDL*,haciéndola más sencilla de usar y una opción muy útil para proyectos independientes.

SIMD: *Single Instruction Multiple Data*, una sola instrucción datos múltiples. *SIMD* toma prestado la idea de realizar una operación a un conjunto de datos de forma paralela, permitiendo acelerar la ejecución de este tipo de operaciones. Nótese que aunque *SIMD* se podría aplicar a muchos tipos de operaciones, suele ser usado para referirse al uso de estos sobre números de coma flotante, debido a que suponen un mayor problema para el procesador que resto. En cualquier aplicación que tenga grandes necesidades de cálculo, *SIMD* se considera esencial para maximizar el aprovechamiento del procesador.

WYSIWYG: *What You See Is What You Get*, lo que ves es lo que sacas. Este acrónimo se usa para definir editores visuales, es decir, editores con los modificar , no el código, u otro tipo de representación intermedia, si no, el propio resultado final. Este tipo de editores son muy famosos para la realización de documentos y páginas web, también son ampliamente usados en el diseño de escenarios y otros elementos gráficos dentro de los videojuegos.

0.2. Conceptos

Introspección: La introspección hace referencia a la habilidad de un programa de analizar su propia estructura en tiempo de ejecución e incluso hacer modificaciones sobre ella.

Metaprogramación: Significa literalmente un paso más allá de la programación, aunque se podría traducir como la programación de la programación. Si los programas usan su código para tratar la información que reciben, los metaprogramas usan su código para tratar el código que reciben, en otras palabras, los metaprogrmas perciben el código como los datos sobre los que operar, recibiendo y emitiendo código. Esto permite automatizar partes de la programación y ocultarlos al programador con el fin de liberarlo de dicha tarea haciendo más sencilla y rápida la programación.

1. Introducción

El mercado de los videojuegos ha crecido altamente durante los últimos años, como se trata de un mercado aún en evolución existen una gran cantidad de sistemas orientados al desarrollo de videojuegos. Si bien existe una gran cantidad de este tipo de sistemas, llamados motores de videojuegos, muchos de ellos comparten estrategias, y , en muchos casos, incluso tecnologías.

1.1. Que es un videojuego?

Debido a que es aún un mercado en evolución muchos productos diferentes terminan bajo el paraguas de los videojuegos. Ya que nuestro proyecto es un motor de videojuegos es importante saber con exactitud qué es un videojuego y cuáles son sus características:

- Simulación: Debido a que se ejecutan en computadores (tales como ordenadores personales o consolas) la clasificación básica es de software, además, por lo general un videojuego es un software pensado para ser ejecutado por sí mismo para el usuario final dentro de un sistema operativo, son considerados como una aplicación. En concreto, dado que su intención es simular entornos (tanto reales como ficticios)

- Interactivos: Los videojuegos no son cerrados, no realizan solamente una tarea, si no que esperan entrada del usuario, además esta entrada no es una entrada puntual como podría ser una aplicación cualquiera, se espera que sea una entrada continua y que el programa debe responder a ella cambiando su estado e incluso su contenido.

- De tiempo real: La respuesta de un videojuego se espera que sea instantánea, es decir, que la respuesta que deba generar a la entrada del usuario o a distintos eventos se genere con un nivel muy bajo de retrasos.

- Gráfica: Si bien no todos los videojuegos usan gráficos como tal (algunos usan interfaz por consola) podemos asegurar que la gran mayoría sí lo hace, estos gráficos son , además gráficos ricos, es decir, se diferencian en aplicaciones tradicionales de escritorio en el sentido de que usan muchos dibujos y tienen mucho movimiento.

Por tanto, podemos decir que los videojuegos son simulaciones gráficas e interactivas de tiempo real. Estas clasificaciones le confieren a los videojuegos una series de características tanto técnicos como en referencia a su uso.

1.2. Usos de los videojuegos

Otro aspecto importante de los videojuegos es su uso, puesto que bajo la denominación de videojuegos se encuentran multitudes de productos con intenciones muy diferentes:

- Ocio y entretenimiento: El principal objetivo de los videojuegos es el entretenimiento, esto fue lo que causó el inicio de los videojuegos, el deseo de divertir y entretener a la gente.

- Educación: Los videojuegos son, al fin y al cabo, simulaciones, es decir mundo o entornos artificiales y que no tienen ningún tipo de límite, bajo estas circunstancias es posible crear pequeños mundos cuyas reglas se adapten para el aprendizaje.

- Entrenamiento: Como especialización de la educación se encuentra el entrenamiento y es que las características de los videojuegos los hacen altamente adecuados para entrenar personas para todo tipos de entornos. Este entrenamiento ya es usado de forma satisfactoria para entrenar pilotos de vuelo, abaratando enormemente los cursos de aviación.

- Psicología: La posibilidad de crear cualquier mundo incluye poder crear mundos que ayuden a la recuperación de pacientes con problemas mentales. Un ejemplo es un estudio realizado en mayo del 2012 donde usaban un videojuego creado explícitamente para ayudar en la tarea de curar trastornos mentales[3].

- Medicina: Distintos estudios demuestran que los videojuegos pueden servir de analgésicos[4], además el personal del *Children's National Medical Center* ha puesto en marcha un programa para el tratamiento de niños con enfermedades crónicas[5].

A pesar de la variedad de usos que un videojuego puede tener, sus requerimientos técnicos son similares, haciendo que todas estas variantes usen un mismo tipo de tecnología, el motor de videojuegos.

1.3. Por qué un motor de videojuegos?

Dentro del sector de videojuegos, los motores de videojuegos son una herramienta vital, pues, aunque existen juegos realizados desde cero, pocas veces esto tiene una compensación del algún tipo dentro del juego, siendo mucho más interesante utilizar frameworks para acelerar dicha faena, de otra forma la creación de videojuegos sería un proceso mucho más largo y sería complicado tener juegos de alta calidad técnica.

El autor de este trabajo decidió realizar uno de estos motores por que la importancia que tienen dichos motores junto a su complejidad técnica hace que sean piezas de tecnología de alto interés tecnológico, esto los convierte en un reto muy interesante donde se requieren ciertos conocimientos de alto nivel y durante esta tarea el autor ha sido capaz de aprender una gran cantidad de cosas, en especial, aquello relacionado con el desarrollo de aplicaciones usando multi-threading sin el uso de bloqueos, también se ha aprendido sobre el desarrollo interno de un videojuego, este tipo de conocimientos, si bien aparentemente no suelen ser de gran utilidad marcan una diferencia, ayuda a crear juegos (e incluso otras aplicaciones) con mejores resultados, además el tener un motor de videojuegos propio implica varias ventajas: en un futuro dicho motor se puede utilizar para realizar juegos tanto comerciales como gratuitos, se tiene un control total del código lo que permite añadir nuevas funcionalidades, hacer mejoras o incluso portar a otras plataformas con una menor complicación frente a otro motor desarrollado de forma externa, puesto que se tiene acceso al código y ya se tiene un conocimiento sobre él y por último, es útil tener un motor de videojuegos por que es posible que para otros proyectos partes de él se puedan reaprovechar.

1.4. Antecedentes

Cuando se trata de un motor de videojuegos, existe un estándar: motores escritos en C++, con un lenguaje de scripting incorporado (siendo este lenguaje generalmente Lua, C#, o más recientemente, Javascript), un editor con determinadas funcionalidades (editor de escenas del tipo WYSIWYG, edición de código, prueba de juego dentro del editor…), una estructura de programación basada en orientación a objetos y pensados para un máximo de 4 núcleos. Muchas de las características de este estándar son fruto de la evolución:

- C++: es el lenguaje usado en la gran mayoría de proyectos que necesitan procesar muchos debido a que se requiere un lenguaje de bajo nivel que se ejecute sobre el hardware de la forma más eficiente posible, además, con tal de llegar a esta velocidad se necesita poder tener un gran control sobre el lenguaje. Además, es uno de los lenguajes más extendidos, con una gran cantidad de librerías útiles para el desarrollo de videojuegos y una gran cantidad de documentación.

- Lenguaje de scripting: A pesar de su gran velocidad en la ejecución, la industria crea los videojuegos con lenguajes de scripting incrustados dentro del motor de videojuegos, aunque existe una pérdida de rendimiento el uso de lenguajes de scripting permite a los desarrolladores invertir menos tiempo. A día de hoy existen tres lenguajes de scripting usados:

    - Lua: Fácil de integrar y muy rápido, su implementación en JIT (LuaJIT) es conocida por ser el lenguaje de scripting “tradicional” (sin contar C# o Java) más rápido , llegando a igualar en muchos casos a C# o Java, a pesar de ser menos dinámicos y por tanto, más fáciles de optimizar. Se ha mantenido como un lenguaje de nicho usado principalmente en videojuegos y algunos programas que requieren de extensiones. Ha perdido un poco de popularidad debido a la introducción en el segmento de los videojuegos de Javascript.

    - Javascript: Versátil y omnipresente, Javascript ha sido durante mucho tiempo un lenguaje casi exclusivo de páginas web en navegadores, aunque la aparición de plataformas como

node.js y la gran popularidad de la Web han hecho que crezca y que aparezca en todo tipo de desarrollos: videojuegos, servidores … Fue introducido en el entorno de los videojuegos por qué su velocidad es muy cercana a la de Lua (encontrándose igualados en muchos benchmarks) y la posibilidad de ejecutar su código en el navegador abre nuevas puertas a los juegos, además su popularidad facilita que la gente encuentre documentación sobre este lenguaje , librerías y trozos de código listos para ser usados.

- C#: Es un lenguaje termino medio, es decir, posee capacidades muy parecidas a los lenguajes de scripting tradicionales (código que se puede ejecutar en cualquier plataforma, inferencia de tipos local, orientación a objetos, introspección y *monkey patching.*…) pero cuyas sintaxis y semánticas recuerdan mucho a C++, incluyendo el uso de tipado estático, que a cambio de una menor comodidad y velocidad de desarrollo permite capturar más errores.

- Editor: Con tal de acelerar la tarea de crear videojuegos, los motores actuales incluyen una herramienta para desarrollar tales juegos llamada editor de videojuegos. En términos básicos se les puede considerar entornos de desarrollos integrados preparados para el propio motor de videojuegos. Aunque presentan variaciones en distribuciones y calidad suelen presentar una serie de herramientas parecidas:

  - Editor de código.

  - Editor de escenas WYSIWYG.

  - Árboles de escena y/o árboles de proyecto.

  - Editor de animaciones.

  - Posibilidad de probar el juego.

En el modelo de desarrollo actual de videojuegos este es un punto importante puesto que un buen editor es clave en facilitar la creación de videojuegos.

- Estructura: Tradicionalmente, los videojuegos usaban estructuras basadas en programación orientada objetos. Esta programación se considera ineficiente desde un punto de vista de rendimiento de la máquina, agrupa trozos de datos que no van a ser usados en el mismo momento, esto hace que dichos programas presenten altos niveles de fallos de caché, esto degrada en gran medida el rendimiento, puesto que un fallo de caché implica que la CPU deba acceder a memoria perdiendo una gran cantidad de ciclos. El diseño que se centra en como se usan los datos y que procura minimizar estos fallos de caché se llama diseño orientado a datos[6], muchos motores y videojuegos han realizado una transición parcial a este tipo de diseños, mostrando al usuario una interfaz tradicional orientada a objetos que facilita la programación.

- Multi-threading: Todos los dispositivos de consumo que permitan el uso de videojuegos tienen un mínimo de dos núcleos en sus CPU haciendo que sea imposible pensar en motores de videojuegos profesionales que no hagan uso de multi-threading, sin embargo, el estándar parece haberse quedado en cuatro núcleos hardware, sin escalar más allá, llegando a desaprovechar el resto de núcleos (en caso de haberlos) y el *hyperthreading.*

Por supuesto, cabe recalcar que a pesar de este estándar, vamos a encontrar muchas diferencias entre estos motores, lo que es más, si nos desplazamos hacia motores *open-source* estas diferencias aumentan, puesto que en este caso no existen tan solo motores sino que también encontramos librerías pensadas para el desarrollo de motores de videojuegos: motores de renderizado, librerías de audio, interfaces gráficas, librerías de conexión, motores de físicas…

1.4.1. Motores en el mercado:

1.4.2. Comerciales:

- *Unity*[7]: *Unity* es el referente en cuanto a motores de videojuegos se trata y aunque es más joven que otros motores se ha conseguido posicionar como la herramienta número uno para el desarrollo de videojuegos. Generalmente es usado para proyectos pequeños o medianos o para proyectos que no buscan altas prestaciones y prefieren sencillez y agilidad en el desarrollo.

- *Unreal*[8]: *Unreal* es uno de los motores más veteranos, se caracteriza por procurar una una gran calidad visual mientras que mantiene un grado aceptable en cuanto la eficiencia. *Unreal* es generalmente usado por videojuegos *AAA* o videojuegos que pretenden impresionar con sus gráficos.

- *CryEngine*[9]: *CryEngine* es otro motor veterano, se ha caracterizado por ser el número uno en cuanto a gráficos, sin embargo, ha perdido gran cuota de mercado justamente por qué durante el desarrollo los programadores pusieron demasiado énfasis en esta calidad y dejando de lado ordenadores más modestos, haciendo que su rendimiento en estos dejase bastante que desear. En un punto *Amazon* compró el código de *CryEngine*, creando su propio motor *Lumberyard[10]*, siendo este es gratuito (aunque no software libre) . A día de hoy tanto *Lumberyard* como *CryEngine* existen y ofrecen características diferentes.

1.4.3. Open-source:

- *Ogre*[11]: Dentro del panorama *open-source* un motor altamente conocido es *Ogre*. *Ogre* es un motor, sin embargo, no es de videojuegos, es un motor gráfico. La comunidad de *Ogre* es tan grande que ha creado varios *plug-ins*, extras e incluso construcciones ya hechas alrededor de *Ogre* para su uso. Otro software parecido es *Irrlicht*, tomado generalmente como una alternativa ligera a *Ogre*.

- *Godot*[12]: Un desarrollo altamente interesante que ha estado ocurriendo dentro del software libre es *Godot*, un motor usado por una compañía de videojuegos y que fue liberado en Febrero de 2014 para que todos pudiesen mejorarlo. A día de hoy *Godot* ha lanzado su segunda versión mayor, mientras que la tercera se encuentra en desarrollo, en este tiempo este motor ha crecido enormemente, demostrando la capacidad que la comunidad de programadores tiene y al mismo tiempo ha llenado un vacío que existía en la comunidad de software libre, el de un motor que estuviese destinado a la productividad y que fuese usable.

2. Objetivos

Como hemos dicho el objetivo de este proyecto es el de crear un motor de videojuegos, sin embargo, dado el estándar existente es de gran interés crear proyectos que se desvíen de este, con la intención de crear nuevos productos que aporten nuevas formas y diversidad al conjunto.

Dentro del diseño de este motor se han tenido los siguientes puntos en cuenta:

- Cambio de lenguaje, de C++ a Nim
- Diseño Orientado a Datos
- *Multi-threading*
- Modularidad

2.1. Por qué no C++?

- Una de las elecciones que se han hecho en la creación de este trabajo es el cambio de lenguaje, a pesar de que el estándar es C++ y es el lenguaje más extendido para la implementación de motores de videojuegos, no es un lenguaje que sea rápido de escribir, esto se evidencia al comprobar que muy pocos juegos comerciales en los últimos años usa C++ completamente, todos ellos usan un lenguaje para realizar los scripts del juego, además C++ presenta algunas desventajas que no son evidentes desde simple vista:
    - Facilidad de escritura: Aunque esto ya se ha comentado en el párrafo anterior queda comentar las razones de esto, estas son:

- El tipado estático junto al requerimiento de escribir todos los tipos aumenta el tiempo que un programador tarda en escribir un programa, puesto que debe saber (y esto muchas veces implica revisar la documentación o el código en cuestión) y escribir dicho tipo; además, entorpece, en muchos casos, la lectura debido a que existe información redundante que debe ser entendida. Ejemplo:

  Display gameDisplay = Myengine.GetDisplay();

- Compilación estática: C++ es compilado directamente a código máquina, aunque esto lo hace ideal para la programación a bajo nivel, el código de más alto nivel escrito durante el desarrollo de un videojuego (o para la creación de las herramientas usadas durante el desarrollo) se suele beneficiar de técnicas que benefician a la interpretación y la compilación dinámica: información de tipos en tiempo de ejecución, introspección y *monkey patching* (sustitución de código en tiempo de ejecución). Esto puede ser conseguido en C++ hasta cierto punto, pero un sistema de este tipo no gozaría de la transparencia ni de la integración que existe en estos lenguajes.

- Asignación de memoria manual: La memoria del *heap* que se use en C++ debe ser recolectada manualmente, esta es una de los mayores fuentes de errores y problemas en la programación, por supuesto, en C++ se puede usar conteo de referencias o recolección de basura, sin embargo, esto implica que el código se vuelve bastante verboso y para el caso de la recolección de basura se debe llamar al recolector de forma manual, algo que puede ser muy propenso a errores.

- Lenguaje complicado: C++ es designado como un lenguaje muy complicado tanto de analizar para el ordenador como de entender para el programador: orientación a objetos, punteros, *templates*, preprocesador, además de toda la sintaxis de C.

  Para el programador implica tener que aprender una gran cantidad de información para saber entender el lenguaje con el añadido de que debe conocer un sinfín de técnicas y consejos para suplir las carencias que este lenguaje tiene, aumentando enormemente la curva de aprendizaje.

Para el ordenador esta complejidad deriva en dos efectos: el primero es que aumentan los tiempos de compilación, puesto que requiere de mayor trabajo el analizador y aumenta la dificultad de su optimización, el segundo efecto es debido a esta alta complejidad hay muy pocas herramientas que trabajen sobre el código, a diferencia de otros lenguajes (donde hay gran variedad de sistemas de documentación, *linters*, análisis de estilo…) las herramientas que posee C++ que trabajen directamente sobre código son limitadas, perdiendo oportunidades de facilitar y/o acelerar el trabajo de desarrollo.

- Altos tiempos de compilación: En el punto anterior hemos hablado de como la complejidad de C++ aumenta sus tiempo de compilación. C++ es conocido por tener grandes tiempos de compilación, sin embargo, la complejidad no es el único problema ni el mayor de ellos, el mayor problema que presenta C++ para la compilación son las cabeceras. Debido a la existencia de macros y que estas pueden cambiar de archivo a archivo, pudiendo llegar a cambiar todo el código dentro de la cabecera, es necesaria la recompilicación de todas las cabeceras por cada archivo que se compila, cabe recordar que las cabeceras raramente no incluyen otras cabeceras, aumentando este efecto enormemente. Aunque esto se puede subsanar con el uso de cabeceras precompiladas, en la práctica, raramente es usado, puesto que requiere de una configuración especial. Con tal de que el lector se pueda hacer una idea con el uso de cabeceras precompiladas se suele obtener speed-ups típicos de entre 4x-7x, en el enlace anexado en [13]se presenta una prueba hecha en el compilador GCC, nótese que para este caso se obtiene un speed-up de 5x.

- Macros: Como parte del preprocesador las macros son la forma de metaprogramación de C++, sin embargo, esta metaprogramación es demasiado sencilla realizando un mero cambio del código, de forma directa, sin realizar ningún tipo de chequeo y sin soporte para nada más que esta modificación sencilla del código, supongamos el siguiente ejemplo:

```
#include <stdio.h>

#define CINCO 3 + 2

int main() {
    printf(“%d”, CINCO * 2);
}
```

Suponiendo que no hayan errores de compilación se realiza la siguiente pregunta, cuál es el resultado mostrado en consola? La respuesta es 7, a pesar de que lo lógico sería 10 debido a

que primer se obtiene 5 y luego se multiplica por dos, la realidad es que el código quedaría tal que así:

3 + 2 * 2

Resultando en 7. Es fácil comprobar como el uso de las macros puede volverse muy peligroso, con el añadido de que tiene limitaciones, puesto que es incapaz de usar código C++ que en el el lenguaje del preprocesador no existen bucles. Debido a todo esto un consejo que se le da a los recién llegados a C o C++ es que eviten, tanto como sea posible el uso del preprocesador.

- Fragmentación de herramientas: Cuando se instala las herramientas de desarrollo básicas de C++, estas por lo general incluyen solo un compilador, el resto son desarrolladas de forma independiente, de hecho, la única capacidad que posee este compilador es transformar código C/C++ a código maquina, careciendo de la habilidad de detectar y construir proyectos. Se requiere , por tanto, de muchas herramientas durante el desarrollo del programa: *debuggers*(muchas veces producidos por los desarrolladores del compilador), generadores de documentación, sistemas de configuración y compilación de proyectos (make, cmake, qbs, premake, meson …) y gestores de paquetes para instalar fácilmente librerías, de hecho, a diferencia de las demás plataformas C++ carece de gestor de paquetes oficial y ,aunque existen algunos, raramente son usados.

Como se puede apreciar C++ tiene suficientes carencias como para considerar el uso de un nuevo lenguaje, la pregunta que resta es cual? En este trabajo se hará uso del lenguaje Nim, un nuevo lenguaje independiente que recuerda un tanto a lenguajes de scripting, a pesar de que es un lenguaje compilado. Una alternativa a este lenguaje sería el lenguaje Rust, respaldado por Mozilla, qué se centra en determinar en tiempo de compilación si no hay problemas relacionados con la memoria, asegurando que un programa no tendrá de este tipo de problemas a pesar de no usar gestión de memoria automática.

2.2. Por qué Nim?

Anteriormente se ha comentado que podría haber interés en reemplazar C++ por un nuevo lenguaje, sin embargo, esto representa una mayor dificultad, puesto que debido al uso de C++ existen muchas

herramientas y librerías que alivian parte de la carga pesada de escribir un motor de videojuegos. Por tanto, un lenguaje que pretenda reemplazar a C++ debe presentar una mejora importante para que el esfuerzo de realizar todo este trabajo merezca el esfuerzo y puede considerarse una alternativa.

Nim se define como un lenguaje eficiente, expresivo y elegante. Su intención es proveer un lenguaje que sea capaz de ofrecer un alto rendimiento, que compile a diversos que sea capaz de producir binarios sin dependencias extras, es decir, Nim integra su runtime dentro del ejecutable.

Nim aporta un gran número de características. La más visible de estas es una sintaxis limpia. Nim basa gran parte de su sintaxis en Python y Pascal, sustituyendo los paréntesis , punto y comas y corchetes por elementos que aún con estos elementos son usados y que son mucho más visibles para el ser humano como líneas nuevas o identanción. Al desaparecer símbolos cuya información es redundante y muy poco visual se puede ofrecer una sintaxis más clara y más fácil de entender.

Otra característica que provee Nim es el recolector de basura. El hecho de disponer de un recolector de basura podría hacer parecer que Nim no se encuentra apto para tareas de bajo nivel, sin embargo, esto no supone un problema, primero por que el recolector de basura de Nim es altamente eficiente y se encuentra diseñado para sistemas de tiempo real, segundo, cada thread dispone de su propio recolector de basura eliminando el efecto conocido como *stop the world*, es decir, la necesidad de parar todo el programa para que el recolector pueda realizar su trabajo, tercero, el recolector de basura por defecto puede ser cambiado de forma muy sencilla, dando la oportunidad de usar otros recolectores con otras características o incluso crear uno nuevo ,cuarto, este recolector de basura es opcional, pudiendo usarlo en conjunto con gestión de memoria manual o incluso ser eliminado completamente. Como contrapartida, cabe decir que la memoria común a todos los threads no dispone de recolector de basura (muy probablemente debido a que no es trivial esta tarea sin necesitar del efecto *stop the world*) y que si bien es posible eliminar completamente el recolector, la librería estándar no se encuentra preparada para tal cosa.

Nim es un lenguaje copilado, con la peculiaridad de que primero compila a C, permitiendo aprovechar todas las herramientas y código que existen para este lenguaje. Nim además favorece el ciclo escritura-compilación-testeo usada muy frecuentemente en la programación al proveer tiempos de compilación muy bajos, puesto que su propio compilador posee una gran rapidez y compila a código C que minimiza el uso de cabeceras, el principal factor que ralentiza el código C. Como extra el compilador de Nim puede compilar a cualquier lenguaje para el cuál el compilador tenga un módulo de compilado,

existiendo, a día de hoy además de C, C++ y Javascript, esto aumenta el rango de compatibilidad de Nim y gracias a Javascript permite su ejecución en el navegador.

A nivel general Nim tiene una gran cantidad de cualidades (una API legible y muy extensa) , búsqueda del *0 overhead*,es decir, que todas las mejoras del lenguajes no supongan un impedimento para la eficiencia del programa, utilidades oficiales y muy cohesionadas (generación de documentación, automatización de *bindings* con otros lenguajes, gestión de paquetes …) entre muchas otras, pero dos características que juntas hacen que Nim realmente destaque son: la capacidad que tiene el propia compilador de Nim de entender y ejecutar este lenguaje y las capacidades de metaprogramación que incorpora. Estas características elevan la capacidad de metaprogramación de Nim a un nivel cercano a *Lisp*, lenguaje altamente conocido por permitir tratar de las misma a datos y a código. Esta metaprogramación abre un sinfín de posibilidades: automatización, optimización adicional y optimización automática incluso por parte de librerías, *DSL*s … de hecho esta característica es usada en este trabajo para facilitar la escritura de código para nuestro motor de videojuegos. Un ejemplo de *DSL* para Nim es Rosencrantz[14] basado en el *toolkit Spray* para el lenguaje Scala, en él se puede ver como se puede usar la sintaxis de Nim para crear facilitar de la escritura de determinado tipo de programas, todo ello en tiempo de compilación, permitiendo que el ejecutable sea altamente eficiente.

Parte del interés inicial de este trabajo era entender como el uso de un nuevo lenguaje afectaría al desarrollo de proyectos una envergadura como el que estamos realizando. Las características presentadas pueden hacer pensar que sí puede suponer una mejora importante. Otra razón que movió al autor a realizar este trabajo en este lenguaje es debido a que este lenguaje carece de muchas librerías básicas de interés en el desarrollo de videojuegos, esto obliga al autor a realizar por sí mismo la implementación de estas librerías ayudando en el desarrollo del conocimiento personal del autor, a la par que pueden servir de ayuda a los usuarios de este lenguaje.

Es necesario comentar que Nim no es el único lenguaje con el que se podría haber realizado esta tarea, existe también Rust, desarrollado por la fundación Mozilla. Este lenguaje es de alto interés puesto que pretende ofrecer una gestión de memoria manual segura mediante comprobaciones en tiempo de compilación. Es decir, Rust evita un gran número de fallos de memoria añadiendo cierta semántica al lenguaje que le permite asegurar esto, el principal interés radica en no usar un recolector de basura con el fin de maximizar la eficiencia y permitir que el sistema se comporte de una forma determinista. Además Rust comparte con Nim el interés por el *0 overhead*, es decir, que todas estas mejoras a nivel de lenguaje no repercutan sobre la eficiencia del ejecutable.

Se decidió el uso de Nim frente a Rust debido a que Nim parece tener una mejor economía. Mientras que Nim es comúnmente alabado por ser un lenguaje fácil de leer, Rust es conocido por su gran curva de aprendizaje debido al uso de una sintaxis más tradicional y cercana a C, además la semántica adicional que permite el análisis por parte del compilador dificulta la creación de código, especialmente para los recién llegados a este lenguaje. Debido a que se espera que el posible usuario escriba parte de su juego en el lenguaje que fuera elegido se escogió Nim para facilitar esta tarea.

2.3. Multi-threading

Cualquier dispositivo de consumo general (ordenadores, móviles, consolas…) posee un mínimo de dos núcleos, siendo cuatro el número de núcleos más generalizado. Esto hace que cualquier aplicación que desee sacar el máximo provecho a estos dispositivos necesite del uso de multi-threading.

Por supuesto, los videojuegos actuales toman esto en cuenta y usan esta técnica, sin embargo,suelen escalar hasta, 4 *threads*, desaprovechando generalmente el resto de hardware. Recientemente han estado apareciendo dispositivos para el publico general con un mayor número de *threads*, prueba de ello son los procesadores móviles de 10 núcleos[15] y procesadores de escritorio con 8 núcleos y 16 *threads [16][17]*, se espera que a medio plazo esto sea la norma puesto que un mayor número de *threads* nos permitirían una mayor capacidad de cálculo con una mayor eficiencia a cambio de una mayor complejidad en la programación.

Por tanto, es evidente que un motor de videojuegos, o en general, cualquier programa, que desee adaptarse a estos nuevos procesadores, deberá tener una mejor escalabilidad que hasta estos momentos. Una cifra que podría ser un objetivo a medio plazo sería una escalabilidad de hasta 16 *threads*, un número que se espera que sea la regla general en el futuro.

Otra razón del por qué se ha decidido implementar este motor de videojuegos en multi-threading es la experiencia obtenida con ello por parte del propio autor. Implementar este tipo de software con multi-threading ha permitido extender el conocimiento del autor en cuanto a este tipo de soluciones, tanto mediante el uso de bloqueo como sin él.

2.4. Modularidad

La disposición típica en todo tipo de software es una disposición fija donde los componentes dentro del sistema se encuentran fuertemente acoplados unos con otros, haciendo complicado la modificación o reemplazo de dichas partes. El diseño que se pretende hacer con este trabajo pretende dar la posibilidad de ser usado para cualquier fin, esto requiere de un nuevo camino que se desvíe de los sistemas fijos.

Entramos en la modularidad. Modular significa dividir algo en módulos, con la característica de que estos módulos se encuentran débilmente acoplados y pueden ser reemplazados , eliminados o incluso reutilizados. En este sentido este proyecto pretende alcanzar dos metas.

La primera es que toda la funcionalidad más allá del núcleo se encuentre dividida en módulos, de forma que estos módulos puedan ser modificados o eliminados con gran facilidad, el objetivo es que módulos

de ámbitos completamente distintos no necesiten saber de la existencia de los demás (un ejemplo aquí sería que solamente por que en un momento dado se necesite hacer sonar un audio en cuanto ocurra una colisión, el módulo de físicas no debería estar directamente acoplado al de sonido). Además dicha modularidad no debería ser especial, es decir, no debería usar mecanismos especiales solo disponibles para ciertos tipos de módulos (gráficos, de sonido, de entrada…), los mecanismos usados aquí deberían ser generalizados y usables por cualquier módulo.

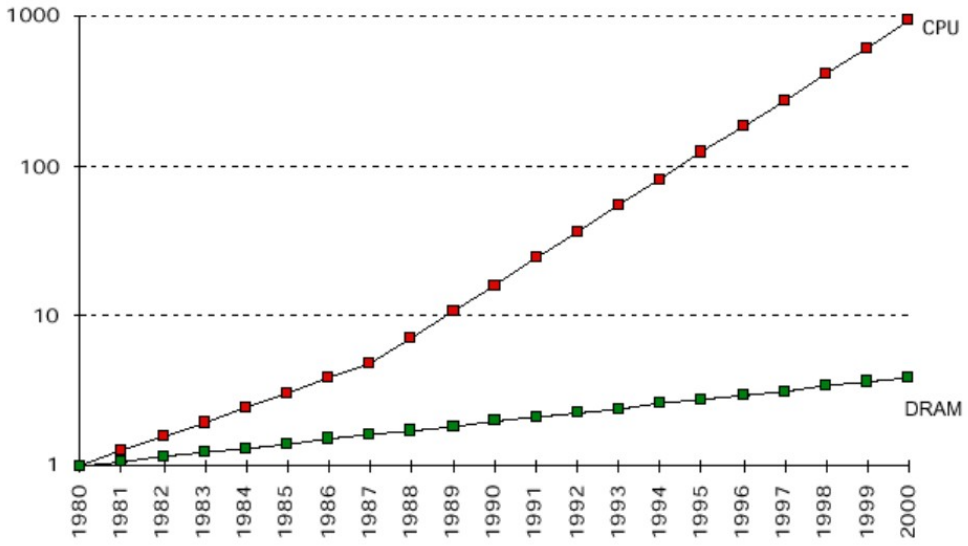
La segunda es que el diseño del núcleo se ha intentado mantener agnóstico del resto de sistema, puesto que este núcleo podría ser útil en software más allá de un motor como el que se realiza en este trabajo. Esto da la ventaja adicional de que reemplazar o modificar dicho núcleo necesitará de un menor trabajo.

2.5. Diseño orientado a datos

El estilo tradicional de escribir aplicaciones es el uso de programación orientada a objetos. La base de este sistema de programación consiste en modelar conceptos como objetos, esto tiene sus raíces en la forma de pensar de los seres humanos, dónde asignamos cosas como objetos y les asignamos que

tienen y qué pueden hacer, de esta forma podríamos modelar un objeto jugador, con diferentes propiedades (posición, dirección, velocidad, imagen a dibujar …) y con acciones que puede realizar (saltar, moverse …).

El problema de este tipo de diseños es que suponen una gran carga en el procesamiento, para entender el por qué de esto, es necesario entender primero las limitaciones de los dispositivos actuales. En los últimos años la unidad de proceso ha recibido grandes mejoras, una CPU moderna es varios órdenes de magnitud superior frente a una antigua, cumpliendo por años la ley de Moore (cada dos años la cantidad de transistores en un microprocesador se duplica). Por otra, parte el acceso a memoria no ha sufrido tal mejora.



Con la gráfica anterior (obtenida del enlace en la referencia [18]) es posible ver como la mejora de la CPU durante años ha sido altamente superior frente a la que ha sufrida la memoria, puesto que mientras

la CPU puede tener mejores diseños, las mejoras que ha sufrido la RAM pasaban por aumentar la velocidad de trabajo.

Para mejorar este escenario se ha implementado una técnica llamada *caché*, que almacena dentro de la CPU un subconjunto de los datos usados en memoria, generalmente los últimos datos usados. Dado que en el transporte de datos una parte importante se debe a la inicialización, y es un valor independiente de la cantidad, cuando se realiza el transporte no se realiza solo del dato en cuestión, además se transportan datos siguientes, con la esperanza de que el programa al continuar los utilice. Este mecanismo se conoce como fallo de caché y un cierto número de estos pueden suponer un problema en la rapidez de ejecución, puesto que el procesador debe esperar cada vez a que se recuperen los datos de memoria, un proceso muy lento.

Por tanto, la pregunta que resta, por que la programación orientada objetos tradicional es tan ineficiente? Como hemos dicho un objeto tradicional incorpora todo tipo de información, muchas veces esta información es tanta y tan variada que un subprograma que deba iterar sobre todos los objetos para realizar una operación (como podría ser el renderizado) generará un fallo de caché por cada iteración del bucle, llegando a provocar grandes ralentizaciones.

La popularización de los lenguajes orientados a objetos y el gran problema que supone el acceso a memoria ha originado una tendencia llamada Diseño orientado a datos (del inglés *Data Oriented Design*). El Diseño orientado a datos pretende estructurar los datos de la misma forma en la que van a ser usados, con el fin de minimizar los fallos de cache. Este tipo de diseños tiene como extra añadido que facilita el uso de instrucciones *SIMD*.

2.6. Allegro

Este trabajo pretende realizar un motor de videojuegos que pueda servir en el futuro, por tanto, el foco aquí dado es principalmente para el núcleo del mismo. Por supuesto, realizar tan solo el desarrollo del núcleo sin tener en cuenta nada más sería un disparate, el motor no tendría ningún tipo de uso hasta que el núcleo no estuviese acabado, y sin uso, su desarrollo disminuiría con el riesgo de terminar.

Para realizar estos módulos se necesita de una librería capaz de abstraer la interfaz del sistema operativo han usado las librerías *Allegro[19]*. Inicialmente se planteó la posibilidad de realizar este trabajo desde cero, esta idea fue rápidamente descartada, un trabajo así superaría con creces el tiempo

del que un trabajo de final de grado dispone, otra opción descartada fue el uso de diversas librerías para cada tarea, elegir e integrar cada librería implica demasiado tiempo, además, el autor de este trabajo no conoce ningún sistema de renderizado 2D para C o *Nim* que funcionen de forma autónoma.

Al final la opción escogida fue el uso de una librería externa con todas las funciones requeridas ya implementadas. Si bien existen un gran número de librerías, hay 3 que destacan: *SDL[1]*, *SFML[2]* y *Allegro*. De entre estas tres librerías, *SFML* fue descartada debido a que es una librería para C++, y si bien *Nim* es capaz de compilar a C++, se prefiere mantener C como lenguaje intermedio. Entre *SDL* y *Allegro* la decisión final fue *Allegro*, puesto que a diferencia de *SDL* se encuentra ideado para su uso en multi-threading.

Puede que en el futuro Allegro sea reemplazado debido a que usa sus propios threads para dar soporte a distintas utilidades, en caso de ser sustituido sería por librerías que se integrasen mejor con el sistema de threads de este motor.

3. Introducción a Nim

En este trabajo el lenguaje usado es Nim, con tal de que el lector pueda comprender los trozos de código que se expondrán más adelante es conveniente entender este lenguaje lo suficiente para ser capaces de llegar a entender estos fragmentos. En esta introducción se presupone que el lector ya conoce C++, otros lenguajes como Java o C# también pueden servir, sin embargo, en las comparaciones entre lenguajes se usara C++ debido a que es el estándar en los motores de videojuegos.

3.1. Primer programa

Veamos un primer ejemplo que nos ayude a entender diferencias básicas de Nim con C++.

Aún con este pequeño ejemplo tenemos una pequeña idea de la sintaxis de Nim. Primero cabe explicar que hace este programa, una vez ejecutado pedirá al usuario mediante consola que inserte su nombre, para luego escribir un saludo.

3.2. Llamadas a funciones

Las llamadas a funciones con Nim se hacen de la misma forma que en C++, se ponen los argumentos entre y paréntesis y se separan entre ellos con comas.

```
readLine(stdin)

myFuncion(argumento1, argumento2, argumento3)
```

También se pueden invocar funciones sin paréntesis, en este caso debe existir un espacio entre el nombre de la función y el primer argumento.

```
readLine stdin
```

Al igual que antes una función con múltiples argumentos debe seguir separando sus argumentos con comas:

```
4  echo "Hola, ", nombre, "!"
```

Hasta ahora se han visto las funciones echo y readLine, son funciones que sirven para la comunicación con la salida estándar, que suele ser la consola. Echo imprimirá en la salida estándar mientras que readLine obtiene caracteres desde la entrada.

3.3. Declaración de datos

En el programa de antes hemos visto lo siguiente:

```
3  var nombre: string = readLine(stdin)
```

En esta línea podemos ver varias cosas, por ahora nos centraremos en la palabra *var. Var* declara una variable, tiene la siguiente sintaxis:

```
var nombreDeVariable: tipoDeVariable
```

La asignación de variables es igual que en C++:

```
nombreDeVariable = valorDeVariable
```

Al igual que en C++, se puede especificar tipo y valor al mismo tiempo, tal y como se ha hecho en el primer ejemplo:

```
3  var nombre: string = readLine(stdin)
```

Nim, incorpora dos variantes de *var: let* y *const*.

```
var variable    : tipo = valor
let soloLectura : tipo = valor
const constante : tipo = valor
```

Let declara una variable de solo lectura, es decir, funciona exactamente de la misma forma que una variable pero no puede ser reescrita. Const declara un valor que es constante y conocido en tiempo de compilación. Estas dos variaciones necesitan ir acompañadas de una asignación, dado que son de solo lectura no tienen ningún sentido no darles un valor inicial.

Cuando asignamos un valor a una declaración (sea *var*, *let*, *const*, o incluso el argumento de una función) no es necesario definir el tipo, esto se conoce como inferencia local y ayuda a mejorar la legibilidad y la facilidad para realizar cambios en los tipos.

```
var variable    = valor
let soloLectura = valor
const constante = valor
```

3.4. Main

En Nim no existe una función *main* que sirva de punto de entrada como en C++, en cambio, Nim llamará a todo lo que está en el primer nivel de cualquier módulo, esto permite añadir código de inicialización en cualquier módulo, además de facilitar la tarea de escribir pequeños programas.

3.5. Funciones

Al igual que cualquier lenguaje que se precie Nim también incluye funciones, con la diferencia de que el lenguaje los llama procedimiento puesto que función en el entorno matemático las funciones no tienen efectos colaterales (es decir, su resultado depende únicamente de sus argumentos). Aquí podemos apreciar una definición de una función:

```nim
proc nombreDeFuncion() =
  echo "Hola, mundo!"
  echo "Hola otra vez"
```

En este caso definimos una función llamada “nombreDeFuncion” que no tiene valor de retorno , que no recibe ningún argumento y que escribe dos líneas en pantalla. Nótese que en casos como este donde no hay argumentos, los paréntesis pueden ser omitidos.

Veamos otra definición:

```nim
proc otraFuncion (argumento1: tipo1): tipoDeRetorno =
  echo "Vamos a devolver un valor"
  result = valorDeRetorno
```

Esta es una función llamada “otraFuncion” que recibe una argumento de tipo “tipo1” y que devuelve un valor del tipo “tipoDeRetorno”, como podemos ver usamos una variable llamada *result* que no ha sido declarada, esta variable es del mismo tipo de retorno y es definida automáticamente por Nim quién retornará esta variable en cuanto termine la función. Esta función retornará valorDeRetorno en cuánto termine. Esta variable es muy útil en funciones donde el resultado sufre varias transformaciones hasta conseguir el resultado final, esto ayuda a evitar *return* en medio de las funciones, algo que entorpece la lectura. En caso de ser necesario se puede usar return para devolver el valor:

```nim
proc nombreDeFuncion (argumento1: tipo1): tipoDeRetorno =
  echo "Vamos a devolver un valor"
  return valorDeRetorno
```

Para funciones sencillas que solo constan de una expresión podemos obviar tanto *return* como *result*:

```nim
proc nombreDeFuncion (argumento1: tipo1): tipoDeRetorno =
  valorDeRetorno
```

Si tenemos argumentos pueden separarse por comas, o por puntos y comas:

```
1  proc nombreDeFuncion (a1: t1, a2: t1; a3: t2, a4: t1): tipoDeRetorno =
2    valorDeRetorno
```

Esta diferenciación es útil por qué si tenemos más de un argumento con tipo repetido de forma consecutiva, es posible escribir el tipo solo del último:

```
1  proc nombreDeFuncion (a1, a2: t1; a3: t2, a4: t1): tipoDeRetorno =
2    valorDeRetorno
```

Nótese que en este caso se ha usado punto y coma para separar los argumentos a1 y a2 de a3, esto es puramente visual y Nim no impone ninguna regla al respecto, la misma función usando solamente comas sería válida.

3.6. Discard

Hemos visto como devolver valores mediante funciones, sin embargo, hay una regla que aún no se ha comentado, en Nim es obligatorio el usar una variable devuelta por una función, el racionamiento tras esto es que si devuelve un valor es por que se espera que será usado,bien por ser el resultado o bien por ser información sobre como ha resultado la función, se espera que sea usada. Si realmente no se desea usar se puede usar la palabra *discard*.

```
1  proc imprimir (): string =
2    echo "Imprime en pantalla"
3    result = "Una cadena"
4
5  # Incorrecto, el compilador emitirá un error
6  imprimir()
7
8  #Correcto
9  var variable = imprimir()
10 discard imprimir()
```

Discard también sirve para aquellos trozos donde el compilador espera código pero no queremos que haga nada (por ejemplo para una función que en ese momento está vacía):

```
1  proc imprimir (): string =
2    discard
```

3.7. Módulos

A diferencia de C++, Nim divide el código en módulos. Cada módulo consta de un fichero. Un módulo puede ser importado mediante la sentencia *import*.

```
import math
```

Debemos diferenciar en que frente a los *include*s de C++ que incluyen todo el archivo en cuestión, import solo deja disponible aquellos símbolos que hayan sido exportados. Para marcar algo (un tipo, una variable, una constante, un let, funciones …) se escribe un asterisco justo luego del nombre:

```
1  var variableNormal: int
2  var variableExportada*: int
3
4  proc funPrivada() =
5    echo "No se puede ver desde fuera"
6
7  proc funPublica*() =
8    echo "Sí se puede ver desde fuera"
```

En este caso otro módulo que importe nuestro archivo solo importaría la variable “variableExportada” y la función “funPublica”.

3.8. Bucles y control de flujo

Las diferencias entre C++ y Nim en cuanto a bucles y control de flujo son pequeñas. Los condicionales funcionan de la misma forma:

```nim
# Condicionales
if valor == 3:
  discard
elif valor == 4:
  discard
else:
  discard
```

En este ejemplo hemos usado *discard* para indicar qué no harán nada ninguna de las ramas. Cuando hay que realizar diversas acciones en función del resultado, se usa el equivalente del *switch* de C++

```nim
# 'Switch' llamado case en Nim
case variable
  of 3:
    variableNueva = 3
  of 4:
    variableNueva = 4
  else:
    variableNueva = 5

```

A diferencia de switch, case no se encuentra limitado solo a enteros y puede funcionar con cualquier cosa, tampoco necesita uso de break justo luego de cada rama. A continuación veremos los bucles:

```nim
# Bucle while
while true:
  break
```

No hay diferencia con los bucles *while* de C++. Por último queda el bucle *for* que sí que tiene cambios:

```nim
# Bucle for numérico
for i in 0..10:
  echo i

# Bucle for iterativo
for elemento in miArray:
  echo elemento
```

En Nim desaparecen los bucles *for* tradicionales y solo quedan los bucles iterativos, por suerte gracias a la sintaxis de Nim realizar un bucle *for* que itere sobre números. Además, en su intención de ser rápido, esto bucles no suponen ningún *overhead* para el programa.

3.9. Orientación a objetos

Igual que C++, también existen clases en Nim. En Nim a diferencia de C++ solo se permite heredar de un padre, además , una variable en un objeto tiene dos niveles de acceso: público (marcado con una estrella en el nombre, igual que los símbolos exportados, todo el mundo puede acceder a dicha variable) y privado (solo el módulo actual puede acceder a la variable).

```
1  type miObjeto = object of RootObj
2    varPrivada: int
3    varPublica*: string
```

En este caso hemos definido un objeto que tiene dos miembros y que hereda de RootObj. En Nim los objetos que no heredan se consideran finales y por tanto no pueden ser padres. Se observa que Nim parece limitar la herencia, y es que Nim favorece la composición (relación: tiene-un) frente a la herencia (relación: es-un).

En Nim no se definen métodos directamente relacionados con clases, en cambio si se realiza una llamada usando la sintaxis de punto (la usada en C++) la expresión justo antes del punto se pasará como primer argumento, si hacemos esto podemos no usar paréntesis :

```
1  # Es lo mismo
2  discard miArray.len()
3  discard miArray.len
4  discard len(miArray)
```

Este tipo de llamadas permiten flexibilidad en la escritura del código, al tiempo que permiten extender cualquier tipo y que en las funciones ofrecen una sintaxis con una menor abstracción y más fácil de ver.

3.10. Funciones y tipos genéricos

En Nim también existen funciones y tipos genéricos:

```
1  proc funcionGenerica[T](a: T) =
2    echo a
3
4  funcionGenerica[int](3)
```

Al igual que en C++, los parámetros de estos genéricos pueden ser tipos o incluso valores, como enteros.

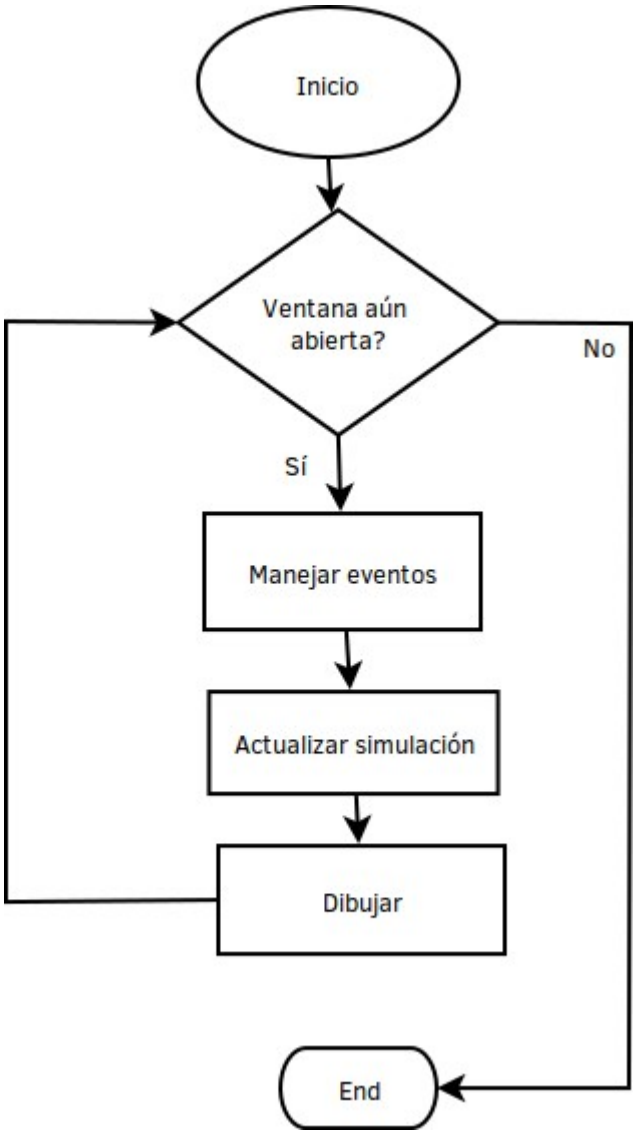
4. Estrategias en el bucle principal

Una de las partes fundamentales de un videojuegos es su bucle principal (*main loop*) en inglés. Es el bucle encargado de mantener todas las operaciones en un orden específico. Es responsable de invocar la actualización, el renderizado y la gestión de entrada ,entre otros.

Generalmente el procedimiento de actualización gestiona la lógica del juego (incluidos scripts y físicas), el procedimiento de renderizado es responsable de dibujar objetos en la pantalla, y el procedimiento de entrada analiza si hay alguna entrada a la que reaccionar.

En videojuegos tradicionales estos procedimientos son fijos y deberían ser modificados para cambiar su comportamiento, mientras que en sistemas orientados a objetos se suelen usar jerarquías de objetos para esto. En este trabajo se ha decidido usar eventos, pues son una forma sencilla de resolver esto mientras que son una buena solución para un sistema en *multi-threading.*

El diagrama base de un bucle sería tal que este:



Durante el bucle existe una característica llamada *timestep*, en español paso del tiempo. Esto se encarga de que el videojuego no esté corriendo de una forma continuada debido a que esto no solo sería un gran desperdicio de recursos, además podría cambiar la experiencia de juego entre diferentes ordenadores. Este *timestep*, sin embargo, puede ser realizado de diferentes formas haciendo que deba ser considerar su implementación.

4.1. Delta de tiempo fijo

La primera aproximación es de mantener un tiempo de paso fijo, es decir que durante la ejecución del programa los pasos que realizará el videojuego serán de un mismo tiempo cada vez (por ejemplo: 16ms para conseguir 60 *FPS).* En sistema con un solso *thread* esto llama a la desigualdad, con ordenadores que serán incapaces de seguir el ritmo y otros que muy probablemente se encuentren desaprovechando potencia, además diversas configuraciones dentro del sistema operativo pueden hacer cambiar este valor (para monitores capaces de mostrar más imágenes por segundo por ejemplo), sin embargo, en nuestro caso seguiría con el valor predeterminado. A pesar de sus inconvenientes, estte tipo de delta puede ser de utilidad en plataformas sencillas (como dispositivos móviles).

4.2. Delta de tiempo variable

La simulación avanza tanto como el tiempo que ha tardado el computador en procesar la iteración, haciendo que la simulación avance el tiempo exacto, independientemente de la velocidad a la que transcurra. La desventaja de este proceso es que un tiempo de paso variable no es adecuado para el motor de físicas puesto que números de paso muy grande puedan provocar errores, debido a que los de físicas analizan la situación tras el movimiento, si un objeto tiene la suficiente velocidad y el tiempo de paso es muy grande este objeto puede terminar en posiciones inválidas, generando comportamientos anómalos en el motor de físicas.

4.3. Delta de tiempo semi fijo

Los dos casos anteriores tenían diferentes ventajas y desventajas, por tanto, este tercer tipo pretende situarse entre medias y sacar lo mejor de ambos.Se tiene un valor fijo y el valor de iteración, se obtiene el menor y este valor es el que será usado. Esto permite que el computador pueda funcionar tan rápido como le sea posible al mismo tiempo que se establece un máximo de tiempo de paso para el motor de físicas.

4.4. Solución en el trabajo desarrollado

Para este trabajo se está implementando un tiempo fijo, sin embargo, esto es un menor problema que en otras plataformas debido a que: por el momento este motor carece de sistemas de físicas, inutilizando buena parte de los argumentos y por otra parte, como la ejecución es concurrente, este paso fijo

implique un menor impacto. Sin embargo, esto es un punto de mejora para el futro, con la intención de que la implementación actual sea reemplazada por una con un delta de tiempo semi fijo.

5. Desarrollo

5.1. Introducción al motor *Tart*

Hasta ahora se ha podido ver cuáles eran los objetivos a cumplir, en está sección se analizará como se han implementado estas ideas y como se ha llegado al diseño actual. Así mismo el autor desea aprovechar para dar a conocer el nombre elegido para el motor: *Tart*.

Por supuesto, se ha tenido en mente la sencillez para recordar este nombre y su facilidad de pronuncia, puesto que a pesar de algo trivial, esto afecta a como los usuarios lo perciben , además por ser un nombre diferente permite que desde el principio destaque como algo nuevo.

Este nombre, sin embargo, no ha sido elegido de manera aleatoria, *Tart* hace referencia a la tarta de frutas, un dulce que consta de una base, un relleno y varios elementos por encima. Este dulce tiene ciertas características con el motor:

- Ligero: Dentro de los dulces, tiende a ser bastante ligero, sin ser muy alto o contener muchos elemento, esto se traduce en el objetivo del motor de minimizar el uso de recursos evitando la saturación (en este caso del ordenador).

- Modular: No existe ninguna regla en que lleva este pastel, pudiendo llevar cualquier tipo de fruta, puede llevar, incluso otros elementos (como chocolate), además el proceso de añadirlas es tan sencillo como dejarlo encima de la base, recordando a la sencillez que pretendemos conseguir con los módulos de este motor.

- Dulce: Como se ha mencionado se trata de un dulce, la dulzura se encuentra relacionada con la facilidad muchas veces, representando la sencillez con la que se pretende que nuestro motor sea extendido.

Por último, cabe mencionar que partes del diseño del motor no han sido realizadas por el alumno, en cambio, han sido usadas a partir de artículos disponibles en internet con permiso de sus respectivos autores, para posterior referencia estos artículos serán añadidos al texto. Cabe mencionar que si bien el diseño de ciertas partes no ha sido realizado por el alumno, la búsqueda y determinación de que partes deben ser usadas así como el diseño global del motor y varias de sus partes sí son originales. La razón de esta utilización de código es el hecho de que muchas de estas estructuras son de una gran complejidad, haciendo que su diseño desde cero requiere de una gran experiencia, además de la depuración de este diseño siendo probado en distintas arquitecturas.

5.2. Multi-threading

Anteriormente, se ha propuesto un sistema de *threads* escalable, es decir, que sea capaz de adaptarse a cualquier número de threads. La forma que se ha propuesto en este trabajo es un sistema de threads que procesan peticiones de trabajo, llamadas *jobs*, debido a que existen distintas necesidades en cuanto a la ejecución también existen distinto tipos de *jobs*:

- *Jobs* síncronos: También llamados solo *Jobs*, son los principales realizadores de acciones. Existe un *thread* de proceso de *jobs* por cada *thread* ofrecido por el hardware, de esta forma se puede maximizar el rendimiento que ofrece el microprocesador. Estos *jobs* son de propósito general, siendo los recomendados para operaciones que se encuentren limitados por la CPU (cálculo, procesamiento …).

- *Jobs* asíncronos: De la misma forma que existen *jobs* pensados para procesos que necesiten de CPU, existen *jobs* para procesos que no lo hagan, por lo general, este tipo de procesos son de E/S. Este tipo de *jobs* están dirigidos a cualquier operación que sin hacer uso de la CPU bloquee el *thread* en el que se encuentra, un ejemplo muy claro de esto sería la carga desde disco. Gracias a este tipo de *jobs* los *jobs* síncronos no se detienen, y , además, se pueden realizar un gran número de peticiones asíncronas debido a que hay más *threads* para uso asíncrono preparados, debido a la limitación de los *theads* síncronos.

- *Jobs* del *thread* principal: Hay funciones que tienen necesidades especiales en cuanto a su ejecución: bien necesitan ser llamadas desde el *thread* principal (con el que se ha iniciado la ejecución del programa) o bien necesitan ser llamadas siempre desde el mismo *thread*, en

cualquiera de los dos casos los *jobs* del *thread* principal (o *main thread jobs*) aseguran que serán ejecutados en el *thread* principal, pudiendo dar control en caso necesario.

En realidad, aunque hayamos definido tres tipos de *jobs*, un *job* no cambia, siempre se mantiene del mismo tipo, lo que cambia es el sistema de planificación que se encarga de su ejecución, este sistema *scheduler* (del inglés planificador). A continuación detallaremos el funcionamiento de los *jobs* y de sus respectivos *schedulers*.

Anteriormente se ha propuesto que un valor muy recomendable de *threads* sobre los que este motor debería trabajar son 16, para llegar a un número tal es necesario minimizar el bloqueo entre *threads* con tal de esto todo el núcleo ha sido diseñado usando herramientas *lock-free*, es decir, libre de bloqueos, estas herramientas garantizan que al menos un *thread* realice avances, pudiendo mejorar la contención (situación dónde un *thread* no puede avanzar por qué otro ya lo está haciendo) y eliminando condiciones de abrazo mortal (situaciones en las que dos *threads* compiten por los mismos *locks* pero cada uno tiene uno haciendo que se esperen infinitamente) .

A pesar de que el objetivo de 16 *threads* es sin duda una meta para el futuro, es una cantidad muy ambiciosa para un proyecto como este, para este caso esperamos que este motor pueda ser capaz de usar 6-8 *threads* eficientemente.

El diseño de la ejecución de *jobs* ha sido basado en el post de *Stefan Reinalter* referenciado en [20].

5.3. AtomicWrapper

Este motor hace un amplio uso de las operaciones atómicas, sin embargo, el soporte de estas es bastante escaso, por tanto para la programación de *Tart* se ha desarrollado un modulo para el uso de este tipos de operaciones. En concreto añade un tipo llamado *Atomic:*

```
type
  Atomic*[T] = distinct T
```

Este tipo lo único que hace es forzar que todo sea llamado mediante las funciones de este módulo.
Nótese que por el momento solo están soportados los compiladores compatible con *GCC* , siendo estos

Clang y *MinGW* (*GCC para Windows*), si bien existe código para *MSC*, este código no está terminado y deberá ser reescrito. A continuación serán vistas las operaciones más importantes para el uso de este módulo.

La primera operación es la obtención de valor:

```
proc `value=`*[T](val: var Atomic[T],  valOrder: ValOrderTuple[T])
```

Suponiendo que la variable “miAtomico” se encuentra definida como una variable atómica, debido a la sintaxis de *Nim* el uso más común será este:

```
miAtomico.value
```

Este es un claro ejemplo donde la sintaxis de *Nim* permite una gran mejora en la legibilidad. Con esta expresión ya se podría hacer uso del valor guardado. Nótese que dado que se trata de una función y existe un parámetro opcional se puede añadir un tipo de orden de memoria en caso de ser necesario:

```
miAtomico.value(MemOrder.Acquire)
```

A continuación la asignación:

```
proc `value=`*[T](val: var Atomic[T],  newVal: T)
proc `value=`*[T](val: var Atomic[T],  valOrder: ValOrderTuple[T])
```

Para la asignación existe dos tipos diferentes de funciones, una para asignar el valor directo y otra para asignar un valor especificando al mismo tiempo con que orden de memoria se realizará:

```
miAtomico.value = nuevoValor
miAtomico.value = (nuevoValor, MemOrder.Release)
```

Otra operación muy usada en *Tart* es el incremento y decremento :

```
proc inc*[T](val: var Atomic[T], order: MemOrder = MemOrder.Relaxed)
proc inc*[T](val: var Atomic[T], value: T, order: MemOrder = MemOrder.Relaxed)
proc incVal*[T](val: var Atomic[T], value: T = 1, order: MemOrder = MemOrder.Relaxed) : T
proc valInc*[T](val: var Atomic[T], value: T = 1, order: MemOrder = MemOrder.Relaxed): T

proc dec*[T](val: var Atomic[T], order: MemOrder = MemOrder.Relaxed)
proc dec*[T](val: var Atomic[T], value: T, order: MemOrder = MemOrder.Relaxed)
proc decVal*[T](val: var Atomic[T], value: T = 1, order: MemOrder = MemOrder.Relaxed): T
proc valDec*[T](val: var Atomic[T], value: T = 1, order: MemOrder = MemOrder.Relaxed): T
```

Ambas operaciones (incremeneto y decremento) vienen de cuatro formas : una simple sin ningún valor (solo se añade uno), la misma pero pudiendo elegir el valor, y luego dos funciones que suman y devuelven un valor, con la diferenciación de que *incVal* y *decVal* devuelven el valor luego de la operación mientras que *valInc* y *decVal* devuelve el valor antes de operar.

La última operación atómica será *compareExchange*, una implementación de *CAS* , que viene con múltiples versiones:

```
proc compareExchange*[T](val: var Atomic[T], expected: T, value: T, order: MemOrder = MemOrder.Relaxed): bool
proc compareExchangeVal*[T](val: var Atomic[T], expected: T, value: T, order: MemOrder = MemOrder.Relaxed): T
proc compareExchangeStrong*[T](val: var Atomic[T], expected: T, value: T, order: MemOrder = MemOrder.Relaxed): bool
proc compareExchangeStrong*[T](val: var Atomic[T], expected: T, value: T, order: MemOrder, orderFail: MemOrder): bool
proc compareExchangeStrongVal*[T](val: var Atomic[T], expected: T, value: T, order: MemOrder = MemOrder.Relaxed): T
```

Las diferencias que existen es que las versiones con *Val* devuelven el valor en lugar de un booleano, y las *Strong* realiza un *CAS* fuerte, la diferencia entre el *CAS* fuerte y el normal (generalmente llamado débil) es que en arquitecturas con un modelo de memoria relajado, se puede producir un falo, en cuyo caso el fuerte hará la revisión por si mismo mientras que el débil requiere que esta revisión sea hecha a mano.

5.4. Job

El objetivo de un Job es ejecutar una porción de código en el procesador, veamos su definición:

```
type
  JobFunction* = proc(self: ptr Job){.thread.}
  Job* = object of RootObj
    function: JobFunction
    parent*: ptr Job
    unfinishedJobs: Atomic[int32]
    data*: array[44,char]


    when defined(cpu32):
      padding: array[8,char]
```

En esta definición se definen dos tipos: primero, un puntero a funciones que acepten un único argumento del tipo puntero a *Job*, este tipo es llamado *JobFunction*, será usado para definir las funciones a las que llamen los *jobs*. Nótese que en esta función de ejecución de *job* el argumento es el propio *Job* esto no es solo por que puede resultar de utilidad, la verdadera razón es que un *job* tiene información muy valiosa para la ejecución del código.

El segundo tipo definido es el del propio *job*. Lo primero que encontramos es un puntero a la función que ejecutará, lo segundo que existe es un puntero a otro *job* llamado *parent (* padre en español), este *job* será ejecutado cuando todos sus *jobs* hijos lo hayan hecho. El tercer elemento (*unfinishedJobs*) es una variable atómica (es atómica puesto que se espera que sea usada desde varios *threads*) que mantiene un registro del número de *jobs* (el propio *job* y los hijos) que quedan por terminar, esto es usado para la ejecución del padre. Por último se puede ver un array de *char*s (caracteres, tamaño de un *byte*) llamado *data*. Dado que su principal misión es el cálculo, se espera de los *jobs* que necesiten datos, en este caso incluimos un pequeño hueco para que junto al *job* se puedan almacenar datos. Esta decisión no es trivial y existe por una buena razón, gracias a esto se pueden reducir las asignaciones de memoria, mejorando el rendimiento. También debe notarse que un *job* está diseñado para tener un

tamaño de 64 *bytes*, este el tamaño de una linea de caché, al ajustarnos a este tamaño podemos evitar problemas como el uso compartido falso (*false sharing*), donde datos diferentes (en este caso *jobs*) se encuentran en la misma linea de caché, aparentando que se encuentran compartiendo, sin embargo, dado que cada núcleo tiene su propia caché deben ser actualizados de forma constante con todos las cambios, aún cuándo estos cambios sean irrelevantes para el resto, al evitar este problema estamos evitando un problema serio de rendimiento, especialmente en una estructura tan central como esta.

Un último apunte es que de hecho en el núcleo se usa exactamente un *lock*, en concreto una variable de condición. Esta variable de condición es usada para hacer que los *threads* que no tengan nada que hacer sean puestos en suspensión por el sistema operativo.

Esta estructura ha sido basada en el post por *Stefen Reinalter* referenciado en [20].

5.5. *Schedulers*(planificadores)

Las estructuras encargadas de ejecutar los *jobs* son los llamados *schedulers*, al ejecutar los *jobs* de forma diferente son los que marcan realmente las diferencias entre los diferentes tipos de *jobs.*

5.6. *Scheduler* síncrono

Anteriormente se ha comentado que los *jobs* síncronos son los principales ejecutores en nuestra tarea por obtener el máximo rendimiento de una CPU. Inicialmente se crean *threads* adicionales para cada uno de los *threads* de *hardware*, exceptuando el *thread* inicial, este *thread* es especial puesto que formará parte de los *threads* síncronos como será el encargado de hacer funcionar su propio *scheduler* especial. Esta estructura se encuentra basada en el diseño propuesto por *Stefen Reinalter*[20][21].

Podemos tener un vistazo al funcionamiento de un *scheduler* síncrono observando sus variables:

```
var
  jobAllocator* {.threadvar.}: CircAlloc[MaxJobCount,Job]
  workerThreadsActive*: RtArray[bool]
  numWorkerThread* {.threadvar.}: int
  workThreadQueues* : RtArray[SharedDeque[MaxJobCount,ptr Job]]
  workerThreads* : RtArray[Thread[ThreadInitInfo]]
  workerThreadsCount* : int
  sleepLine : Cond
  sleepLock : Lock
  initialJob* : JobFunction
```

A continuación procederemos a explicar el uso de estas variables, note se el uso *{.threadvar.}*, en términos básicos se puede decir que es una directiva que indica al compilador que la variable a la que acompaña es una variable de *thread* y que debe existir una copia por cada uno.

Almacenar los *jobs* usando la asignación de memoria estándar sería muy lento, ya que sabemos de antemano que estos *jobs* tienen un tiempo de vida corto podemos crear un *buffer (* una memoria preasignada para guardar un tipo de datos) y devolver estos trozos de memoria de forma circular, es decir, una vez llegan hasta el final vuelven al principio. Este asignador está representado por *jobAllocator*, y es del tipo *CircAlloc (Circular Allocator,* en español “asignador circular”) que es el encargado de realizar dicha asignación de memoria. *CircAlloc* requiere de dos parámetros: el primero es el número de trabajos que almacenará, mientras que el segundo es el tipo de datos a almacenar. Esta variable es única por cada *thread* del *scheduler*, al tener cada *scheduler* su propio asignador es posible realizar ciertas optimizaciones dentro del asignador (en concreto es posible disminuir el uso de variables atómicas a 1) a cambio de usar más memoria, este cambio es razonable puesto que se trata de una cantidad pequeña (8k por *thread*) que muy probablemente fuera a ser usada de cualquier forma, y el aumento de velocidad que se obtiene puede ser de interés en una estructura tan crucial como es esta.

Otra variable interesante es *workThreadQueues* que tiene las colas de los *threads* de proceso, una por *thread*, su tipo es *RtArray[SharedDeque[MaxJobCount, ptr Job]]*, en esencia es un tipo llamado *SharedDeque* que tiene dos parámetros envuelto por otro llamado *RtArray*, este último tipo es un tipo sencillo que representa un array asignado de forma dinámica. *SharedDeque* es un *Deque* pensado para usar de forma concurrente, *Deque* significa *Double Ended Queue*, es decir, Cola con doble final, en otras palabras, una cola de la que se pueden sacar elementos tanto del principio como del final. Se usa este tipo de cola debido a que este tipo de *schedulers* tienen una característica especial llamada *work stealing.*

Con una cola y un *scheduler* estándar cabe el riesgo de que los trabajos está mal balanceados y se amontonen todos en un solo *thread*, para corregir esto de forma automática se permite a los *threads* cuyas colas se encuentren vacías "robar" un *job* de otra cola, de esta forma es posible reducir casos en dónde unos *threads* no tienen nada mientras que los demás se encuentran llenos, esto además tiene la característica de que debido a que los *jobs* van a crear nuevos *jobs* en el mismo *thread* en el que estaban permitirán dar trabajo a estos *threads* sin que el *work stealing* deba estar realizándose de forma continua. A continuación detallaremos el algoritmo que siguen los *schedulers* síncronos:

1. Se inicializa y asigna un *job* inicial.

2. Intenta obtener un *job*

    1. Primero buscarán en su cola y si lo han encontrado pararán la búsqueda

    2. En caso de no encontrarlo seleccionarán un *thread* al azar.

    3. Si el *thread* sacado al azar resulta ser el propio *thread*, ignora la operación e indicará que no ha conseguido obtener un *job*.

    4. Intentará robar un *job* al *thread* obtenido al azar, si lo consigue continuará con su ejecución, en caso contrario indicará que no ha conseguido tal cosa .

3. En caso de haber obtenido obtenido el *job* lo ejecutará, en caso contrario se añadirá a la variable de condición *sleepLine* dónde será quitado del procesador por el sistema operativo, dando más tiempo a los demás *threads*, a otros procesos o simplemente liberando recursos del microprocesador.

4. Existe un *flag* llamado *workerThreadActive* que indica si el *thread* debe continuar con su ejecución. Si este *flag* está activado la ejecución volverá al paso dos. En caso contrario la ejecución del *thread* terminará.

Todos los *threads* durmiendo serán despertados por el *thread* principal cada cierto intervalo de tiempo (en estos momentos se encuentra configurado para usar 16ms, una vez por *frame*, es decir, por redibujado). Un *thread* puede tener su flag *workerThreadActive* desactivada si se activa el fin de la

ejecución o si un cierto número de veces consecutivas el *thread* principal debe recuperarlo de la variable de condición *sleepLine*, indicando que no posee trabajo y que por tanto no está aportando nada por lo que es mejor su eliminación. En esta versión preliminar de *Tart* estos *thread*s no son recuperados, pero en versiones futuras será ideado un mecanismo dónde en situaciones en que haya suficiente trabajo y el número actual de *threads* no sea capaz de hacer frente a él se restaurarán *threads* destruidos anteriormente. Estos mecanismos permiten adaptarse de forma dinámica a la carga de la aplicación.

5.7. *Scheduler* asíncrono

Para situaciones dónde se espera que el código tarde en responder y el *thread* deba ser bloqueado existe el *scheduler* asíncrono.

Frente al síncrono, el *scheduler* asíncrono es mucho más sencillo. Este *scheduler* tiene una serie de *threads* listos para ser construidos en cualquier momento y un *pool* (la misma técnica usada que para el *circularAllocator* del *scheduler* síncrono con una diferente implementación). A diferencia de los *threads* síncronos, los *threads* asíncronos comparten *jobs* debido a que estos trabajos tomaran un tiempo indefinido y harán uso principalmente de otros dispositivos más allá de la CPU, haciendo que no exista ningún tipo de orden en su ejecución y que sea mucho más lógico el uso de una cola común.

```
var
  threadPool: FreeList[Thread[ptr Job]]
  jobPool: FreeList[Job] # Allocate the object and return the pointer
  pendentJobs: WordQueue[maxAsyncJobs - 1, ptr Job]
```

Aquí podemos ver las variables relacionadas con este tipo de planificadores. Existen un *pool* de *threads* y otro de *jobs* dónde se encuentran *threads* y *jobs* listos para ser usados. Con la diferencia de que los *threads* en uso no se encuentran almacenados en ningún sitio, son liberados del *pool* de *threads*, en cuanto terminan su ejecución se añaden a ellos mismos a dicho p*ool.* Es notorio también el uso de otra estructura la *FreeList.*

FreeList es una implementación de una lista libre para *multi-threading* sin bloqueos. Las listas libres son un tipo de asignadores que usan listas para indicar que trozos se encuentran libres. Tienen dos

propiedades: una es que debido a que son listas, no hay problema con el orden, la otra es que frente a otros métodos de asignación mejoran la rapidez de asignación y liberación al tiempo que pueden mejorar el uso de la caché.

Para la implementación de la cola se usa la estructura *WordQueue*. De forma parecida a otras estructuras tiene dos parámetros su tamaño y el tipo de dato que albergará. *WordQueue* es una cola optimizada para el almacenaje de datos cuyo tamaño máximo sea el de una palabra, al establecer esta limitación es posible acelerar su funcionamiento al simplificar el tratado de sus datos puesto que solo necesita de una operación CAS.

A continuación el algoritmo de un *job* que va a ser añadido al planificador asíncrono:

1. Primero, intentará obtener un *thread* sin usar.

    1. Si lo consigue activará dicho *thread* y se ejecutará a si mismo.

    2. En caso contrario, se añadirá a la cola, a la espera de que un *thread* lo realice más adelante.

5.8. *Scheduler* del *thread* principal

Este el *scheduler* más simple. En esencia es una versión a un solo *thread* del *scheduler* asíncrono. Cuando un *job* se añade al *scheduler* del *thread* principal, se mantiene a la espera de que sea ejecutado. Este *scheduler* se reserva para operaciones que requieren de un *thread* específico.

El *thread* principal realiza ejecuta tanto los *jobs* asignados a este *scheduler* como a un *scheduler* síncrono propio, este *scheduler* síncrono propio opera de la misma forma que lo hacen los demás.

Se ha creado esta estructura frente a reusar los *threads* síncronos debido a que estos *threads* sufren de *work stealing* haciéndolo inusable para este tipo de *jobs* al necesitar de forma obligatoria que sean ejecutados en este *thread.*

5.9. Estructuras

Debido a la falta de estructuras *lock-free* para *Nim* y los objetivos de este motor (objetivos que requieren maximizar el uso que se le da al microprocesador) se han implementado una serie de estructuras para su uso en *Tart*

5.9.1. Circular Allocator

Circular allocator o “asignador circular” es una estructura que actúa de *pool*, es decir, almacena instancias de un tipo de forma que su asignación y liberación sean lo más rápidas posibles, al mismo tiempo que se mantienen cerca en memoria, facilitando su reuso en la caché. El diseño original de esta estructura realizado por *Stefen Reinalter* se puede encontrar en[22].

Se llama “asignador circular” debido a que interiormente opera sobre un *array* circular que actuará de *buffer* almacenando las instancias que serán usadas, devolviendo con cada llamada una nueva posición del *array*, cuando se llegue al final el contador será devuelto a 0. Veamos el código:

```
import math

type
  CircAlloc* [Size: static[int] , T]  =  tuple # Better less overhead
    baseArray           : array[Size,T]
    index               : uint16

proc init*[Size: static[int], T](self: var CircAlloc[Size, T]) =
  self.index = 0

proc alloc*[Size: static[int], T](self: var CircAlloc[Size, T]): ptr T =
  inc self.index

  when isPowerOfTwo(Size ):
    let arrayPos : uint = (self.index - 1) and (Size - 1 )
  else:
    let arrayPos : uint = (self.index - 1) mod Size

  result = addr self.baseArray[ arrayPos ]
```

Es un código sencillo, en el que es evidente la falta de operaciones atómicas, esto se debe a que la operación de obtener datos será realizada siempre por un mismo *thread.* La razón de este es una ganancia en la velocidad de asignación, a cambio, cada *thread* debe mantener su propio *CircAlloc* causando un aumento de memoria despreciable y que con probabilidad sea útil igualmente.

Para implementar el reinicio del índice del *array* se usa el módulo , debido a que esta operación devuelve desde 0 hasta tamaño – 1, para los casos en los que el tamaño es potencia de dos se optimiza esto a una operación and.

Nótese que no existe ninguna función para liberar el espacio asignado, este se debe a que la propia estructura del buffer circular no lo necesita. Una solución pasaría por añadir un valor booleano, sin embargo, esto requiere de un mayor análisis, puesto que esto implicaría añadir un condicional, algo que que debe ser evitado en caso de ser posible, un condicional en este punto supondría un grave retroceso en cuanto a velocidad y debe ser evitado. Por el momento, la solución aportada es tener un tamaño lo suficientemente grande como para que esto no pueda suponer un problema.

5.9.2. RtArray

RtArray es una estructura muy simple. Simplemente representa un array creado de forma dinámica sin usar asignadores de memoria propios del recolector de basura. Era necesaria una estructura dinámica para las variables compartidas, Nim incluye un array dinámico llamado *seq* (abreviación de *sequence*, secuencia), sin embargo, este no puede ser usado en memoria global (memoria compartida entre *threads*) debido a que utiliza el recolector de basura (recordemos que el recolector de basura se instancia una vez por *thread dejando* sin recolección de basura a la memoria global), por tanto, era necesaria una estructura que se encargase de esto:

```
const ArrayDummySize* = when defined(cpu16): 10_000 else: 100_000_000
type
  DynArray* {.unchecked.}[T] = array[0..ArrayDummySize, T]
  RtArray*[T] = ptr Dynarray[T]

proc allocSharedDynArray*[T](size: Natural): ptr DynArray[T] =
  let arrayPointer = allocShared(size * sizeof(T) )
  result = cast[ptr DynArray[T]]( arrayPointer )

proc newRtArray*[T]( size: Natural): RtArray[T] {.inline.} =
  result = allocSharedDynArray[T](size)

proc initialized*(self: RtArray): bool =
  self != nil

proc destroy*(self: RtArray) =
  self.deallocShared()

iterator mitems*[T](self: var RtArray[T],maxIndex: Natural): var T {.inline.} =
  for i in 0..maxIndex:
    yield self[i]

proc size*[T]( self: RtArray[T]): Natural {.inline.} =
  sizeof(self) div sizeof(T) # Integer diviision
```

RtArray es en realidad un puntero a un *array*, asignado de forma dinámica. Las funciones de las que dispone se limitan a inicialización, destrucción, y preguntas como el tamaño o si se encuentra inicializado, también incluye un iterador (un tipo de función especial que permite acceder a los elementos del array en un bucle *for*) que permite la edición de los elementos del *array* llamado *mitems* (*modifiable items*, elementos modificables en español).

5.9.3. SharedDeque

Un *deque* significa *double ended queue*, o “cola de doble final” en español. En términos básicos es una cola (recordemos que una cola es una estructura *FIFO* dónde guardamos elementos y podemos coger el primero de los que hemos guardado) con el añadido de que también es posible coger elementos del final de la cola (los últimos elementos añadidos), esto es indispensable para el *work stealing* implementado en los planificadores. Un *deque* es, por tanto, una estructura que permite coger elementos determinados tanto en un estilo *FIFO* como *LIFO*. Se puede acceder al diseño original por *Stefen Reinalter*[21].

En el *scheduler* se usará el lado *LIFO*, para que el mismo *thread* que ha guardado los *jobs* los obtenga, es decir, se usará de forma privada para ese *thread*, mientras que el lado *FIFO* será usado para que los demás *threads* roben de la cola. De esta forma al usar el propio *thread* la parte *LIFO* es posible hacer un mejor provecho de la caché.

Como parte de su nombre indica (*Shared)* es una estructura pensada para ser usada en memoria compartida y como todas las demás estructuras usadas en el núcleo de *Tart* evita el uso de *locks*.

En esta ocasión debido a que el código de *SharedDeque* es más extenso que el resto de de estructuras solo se incluirán los fragmentos de mayor importancia:

```
type SharedDeque*[Size:static[int], T] = tuple
  top,bottom: Atomic[uint16]
  baseArray: array[Size,T]
```

Aquí podemos ver la definición de *SharedDeque*. Es un tipo sencillo, cuenta con un *array* llamado *baseArray* que almacena los valores a usar mientras que dos contadores , *top y bottom*, son indicadores de las posiciones máximas y mínimas ocupadas del *array*.

```
proc push* [Size:static[int] ,T](self: var SharedDeque[Size,T], obj: T) =
    let b = self.bottom.value
    let accIndex = accessIndex(Size,b).uint16
    self.baseArray[ accIndex ] = obj


    threadFence(MemOrder.Acquire)
    inc self.bottom
```

Primero analizaremos la función *push*. Cuándo se trata de colas la función *push* es usado para añadir elementos a la cola. El funcionamiento de esta función es sencillo se obtiene el valor del final de la cola desde memoria, se le asigna el valor y se incrementa. Esta función a pesar de usar variables atómicas no hace ningún esfuerzo para ser usada de forma concurrente, esto se debe a que en el planificador (dónde es usada) no es necesario, y un diseño más complicado que garantizase su uso de esta forma no supondría ningún beneficio mientras que impondría una penalización en la velocidad de ejecución.

La función contraria a *push* es *pop*. Si con *push* añadimos al final de la cola con *pop* quitamos su último elemento, de esta forma *push* es la operación *LIFO* de nuestro *deque*. Puesto que *pop* trata de quitar valores de la lista su funcionamiento es más complicado.

```
proc pop*[Size:static[int], T](self: var SharedDeque[Size,T]): T =
    let b = self.bottom.value - 1
    self.bottom.value = b

    threadFence(MemOrder.AcqRel)
    let t = self.top.value

    if t <= b:
        # non-empty queue
        let accIndex = accessIndex[T](Size,b).uint16
        result = self.baseArray[ accIndex  ]
        if t != b:
          # there's still more than one item left in the queue
          return

        # this is the last item in the queue
        if not self.top.compareExchange(t,t+1):
            # failed race against steal operation
            echo "Failed to race against steal"
            result = universalNil(T)# Here 0 means nil

        self.bottom.value = t+1

    else:
        # deque was already empty
        self.bottom.value = t
        result =  universalNil(T)
```

Algoritmo de *pop:*

1. Comprueba que existan elementos en la cola comparando los valores de inicio y fin

    1. Si no hay nada devuelve el valor nulo para el tipo que se está usando.

2. En caso de que sí existan comprueba que no sea el último elemento de la cola

    1. En caso de no serlo devuelve el elemento obtenido

3. Si resulta ser el último elemento de la cola comprueba que una operación de *steal* no lo ha robado.

La última operación importante que queda por ver es *steal.* Recordemos que en contrapartida a *push steal* exhibe un comportamiento *FIFO* y que debido al comportamiento que se espera que tenga nuestro *scheduler* es la única operación que se espera que sea concurrente.

```
proc steal*[Size: static[int],T] (self: var SharedDeque[Size, T]): T =
    var t = self.top.value(MemOrder.Acquire)
    var b = self.bottom.value
    if t < b:
        # non-empty queue
        result = self.baseArray[ accessIndex[T](Size,t).uint16]

        if not self.top.compareExchange(t+1,t,  MemOrder.Relaxed):
            # a concurrent steal or pop operation removed an element from the
            # deque in the meantime.

            result = universalNil(T)

    else:
        # empty queue
        result = universalNil(T)
```

Algoritmo de *steal*:

1. Verifica que la cola no esté vacía.

1. De estarlo devuelve el valor nulo para el tipo usado.

2. Obtiene el valor más antiguo y comprueba que existe (alguien se podría haber adelantado).

    1. Si no existe devuelve el valor nulo

3. Si este valor existe lo devuelve.

5.9.4. SharedTable

Para el sistema de eventos es necesario una estructura capaz de relacionar cadenas con otros datos, la estructura más frecuentemente usada para este fin es la tabla. Una tabla (también llamada diccionario o mapa) es una estructura que en base a cualquier tipo de valor permite relacionar ese valor con otro tipo, como por ejemplo cadenas de caracteres con números, punteros con letras o incluso objetos con objetos. Esto se consigue al transformar los datos en un valor *hash*, valores numéricos que suponen un identificador único para un valor dentro de un determinado tipo (los valores *hash* pueden repetirse entre tipos), estos valores son asignados en una matriz junto a su valor. Debido a que los valores de los *hash* pueden ser números altamente grandes (del orden de cientos de miles) no se guardan estos valores en el índice representado por su *hash*, en cambio, es reducido hasta un número que pueda ser manejado. Debido a esta reducción existirán colisiones entre diferentes valores, estas colisiones son solucionadas mediante algoritmos de *probbing* que se encargan de que una vez que se obtiene el índice encontrar el valor perteneciente a ese índice.

Para el caso de nuestra tabla, (cuyo diseño original ha sido realizado por *Jeff Preshing*, enlace en la referencia [23], aunque extendido a partir del mismo) podemos decir que es una tabla pensada para el uso concurrente (de ahí el uso del nombre *shared*) con un *probbing* lineal. Un *probbing* lineal consiste en desde el inicio comprobar si el registro actual es el correcto, de no serlo repite el paso con el siguiente, en definitiva lo que está haciendo este algoritmo es una búsqueda lineal sobre los registros empezando por el designado por el *hash*.

```
type
  Entry[V] = tuple[key: Atomic[uint32],evalue: V ]
  SharedTable*[K,V] = object
    data: ptr Dynarray[Entry[V]]
    size: uint32
```

Una tabla es un tipo sencillo, un puntero que apunta a un vector de entradas y un número marcando el tamaño, este número no está marcado como atómico por que , por simplicidad y eficiencia, las funciones de creación y destrucción no están pensadas para ser concurrentes, es decir, no son *thread-safe*. En la imagen anterior se puede apreciar otro tipo, el tipo *Entry* que funciona como el tipo que almacena los registros, en este caso la clave sí es atómica mientras que el valor no lo es, la razón es que se espera usar esta tabla como almacenaje, con tal de minimizar los accesos a memoria, para este caso no se devuelve el valor si no el puntero a la variable. Esta estructura fue diseñada originalmente para que el valor fuese un valor atómico:

```
type
  EntryAtomic[V] = tuple[key: Atomic[uint32],evalue: Atomic[V] ]
  SharedTableAtomic*[K,V] = object
    data: ptr Dynarray[EntryAtomic[V]]
    size: uint32
```

En el código actual ambas versiones se conservan, puesto que tienen diferentes usos. La primera función que será analizada es el *getter* (la función para obtener valores) de la tabla normal.

```
proc `[]`*[K,V](self: SharedTableAtomic[K,V], key: K): Maybe[V] =
  when defined(debug):
    if not self.initialized:
      raise newException(ValueError, "Table not initialized")

  let hashKey = hash(key).uint32

  echo "[" & $key & "]" & ":" & $hashKey

  for idxOrig in hashKey..inf(uint32):
    var idx = idxOrig and ( self.size - 1)

    let probedKey = self.data[idx].key.value
    if probedKey == hashKey:
      return box(self.data[idx].evalue.value)

    if probedKey == 0:
      return Maybe[V](valid: false)
```

Este es el algoritmo de esta función:

1. Obtenemos el valor *hash* de la clave.

2. Y realizamos el probbing iniciando desde esa clave:

    1. Convertimos el *hash* a índice

    2. Buscamos la llave actual, si tiene el valor buscado lo devuelve

    3. En el momento que encontramos 0 es por qué nos hemos encontrado con una zona vacía (en este algoritmo 0 se toma como valor inválido para la clave), debido a que estamos haciendo una búsqueda lineal empezando desde un índice determinado, encontrar este valor inválido es significado de que valor que buscamos no se encuentra.

```
proc `[]=`*[K,V](self: var SharedTableAtomic[K,V], key: K, newValue: V) =
  when defined(debug):
    if self.data == nil:
      raise newException(ValueError, "Table not initialized")

  let hashKey = hash(key)

  for idxOrig in hashKey..inf(uint32):

    var idx = idxOrig and (self.size - 1)

    # Load the key that was there
    let probedKey = self.data[idx].key.value
    if probedKey != key:

      # The entry is either free, or contains another key
      if probedKey != 0:
        continue


      let prevKey = self.data[idx].key.compareExchangeStrongVal( 0'u32, hashKey.uint32)

      if prevKey != 0 and prevKey != key:
        continue
```

La siguiente función a ser escrita será el *setter*, es decir, la función para escribir sobre el valor. Nota: en la imagen del código se puede apreciar la sentencia when, que sirve para compilación condicional, si la expresión que tiene es cierta se añadirá el código al ejecutable, su función es la misma que la llamada de preprocesador ifndef de C++.

1. Obtenemos el valor *hash*

2. Realizamos el *probbing* lineal

    1. Obtenemos el valor del registro actual

    2. Si el registro obtenido es igual al de búsqueda se acepta y se devuelve.

    3. Si este registro no está libre pasa al siguiente valor

4. En caso de estar libre se comprueba que no haya sido tomado por otro *thread* y en caso de estar libre lo asigna y lo devuelve.

Nótese que para estos casos usamos el operador *and* como transformador del valor *hash* al valor de índice.

5.9.5. Tabla de punteros

La tabla de punteros es una estructura diseñada para el sistema de eventos. Su único propósito es el almacenaje de punteros para ser llamados después. A diferencia de otras estructuras esta tabla está completamente diseñada por el autor de este trabajo.

```
type
  TableP* [S: static[int],T] = object
    allJobsFinishedCallback : Atomic[pointer]
    data: array[S,Atomic[pointer]]
    when (0..255).contains(S):
      elements: Atomic[uint8]
    else :
      elements: Atomic[uint16]
```

Aquí tenemos el tipo, que como se puede observar es sencillo. Consta de una matriz donde son almacenados los punteros llamada *data*, un puntero extra que será llamado cuando todos los *callback* terminen y un contador de elementos, el tamaño de este dependerá del número de *callback* a almacenar, si es menor que 256 se usará un número de 8 bits mientras que si es más se usará uno de 16 bits.

```
proc `whenJobsFinished=`*[S,T](self: var TableP[S,T], newVal: T) {.inline.}=
  `value=`(self.allJobsFinishedCallback, cast[pointer](newVal))

proc whenJobsFinished*[S,T](self: var TableP[S,T]): T {.inline.}=
  cast[T](self.allJobsFinishedCallback.value)

iterator items*[S: static[int],T](self: var TableP[S,T]): T =
  if self.elements.value != 0:
    for i in 0 .. self.elements.value - 1:
      yield cast[T](self.data[i])

proc init*[S: static[int] ,T](self: var TableP[S, T]) =
  self.elements.value = 0

proc append*[S: static[int], T](self: var TableP[S, T], newData: T) =
  if self.elements.value == S:
    raise newException(OverflowError, "Can't fit inside the table'")

  `value=`(self.data[self.elements.value],cast[pointer](newData))
  inc self.elements
```

Las funciones que operan sobre el son sencillas, con operaciones para inicializar, añadir y acceder a elementos determinados de la tabla de punteros. Nótese como no se toman en cuenta las operaciones de *acquire* y *release*, esto se debe a que la arquitectura sobre la que está destinada inicialmente *Tart* (arquitectura *Intel*) tiene un modelo de memoria “fuerte”, para cada lectura se realiza un *acquire* mientras que para cada escritura se realiza un *release*. Si en el futuro se decidiese portar a arquitecturas con un modelo más débil esto debería ser tomado en cuenta.

5.9.6. FreeList

En la creación del planificador asíncrono era necesaria una estructura que aceptara y devolviera punteros a bloques de memoria sin ningún orden preestablecido, esta es la *FreeList*. Una *FreeList* es una lista especial que almacena bloques listos para ser usados, al igual que el *Circular Allocator* (asignador circular) este es un tipo de asignador que en el caso de *Tart* ejerce como *pool*. En esta estructura se repite el patrón que hemos visto hasta ahora, uso de variables Atómicas y otros con el fin de evitar *locks*.

```
type
  Node*[T] = tuple [data:T, refs:Atomic[uint16], next:Atomic[ptr Node[T]]]
  FreeList*[T] = tuple [head:Atomic[pointer], memPtr: Atomic[pointer]]
```

Se pueden apreciar dos tipos, el de un nodo y el de la cola. En esencia, es una cola enlazada estándar, la cola tiene un puntero al primer elemento y al inicio de los datos (para que más tarde sean eliminados) mientras que el nodo contiene los datos, un puntero al siguiente nodo y una serie de referencias. Estas referencias del nodo es lo único que parece destacar y es lo que permite solucionar una serie de problemas.

Esta *FreeList* tendrá dos funciones para su uso: *add* que recoge un nodo existente y lo añade a la lista y *tryGet* que intenta obtener desde la lista un valor.

Una *FreeList* concurrente es algo cuya complejidad no parece complicado, se podría usar un bucle CAS para realizar modificaciones:

```
proc add*[T](self:FreeList[T],node: ptr Node) =
  var head = self.head.value
  node.next.value= head

  while not self.head.compareExchangeStrong(head,node):
    node.next.value= head

proc tryGet*[T](self: FreeList[T]): ptr Node =
  var result = self.head.value
  while result != nil and not self.head.compareExchangeStrong(result, head.next.value):
    continue

  #retorno ímplicto de result
```

Sin embargo, al realizar esto estamos incurriendo en un fallo, en el tryGet, una vez obtenido el valor del siguiente a que apunta head , alguien podría mientras tanto eliminar el elemento inicial de la lista , melosilla A, así como el siguiente, B. De esta forma el primer *thread* creería que A es el inicio y que B sigue detrás, puesto que A sigue siendo la cabeza, pero el resto ha cambiado, dejando a B como la cabeza de la lista, cuando B no debería encontrarse en ella.

Este problema es bastante común en el diseño de algoritmos *lock-free* y se le conoce como problema ABA, donde un valor cambia varias veces sin que la parte que está trabajando con el se de cuenta, este valor vuelve al que tenía originalmente pero el resto ha cambiado de cualquier forma, haciendo que las acciones que tome sean erróneas, este problema se encuentra explicado con mayor detalle en la referencia [24], junto al diseño original de esta *free list*.

Para solucionar este problema se realiza un conteo de referencias, es decir, se mantiene en memoria el número de referencias a este nodo, y se espera que sea 0, no añadiéndolo a la cola si no debe estar, aunque marcándolo para que se haga en el futuro.

Antes de ver que contienen las diferentes funciones mostraremos dos constantes que van a ser usadas:

```
const
  RefsMask = 0x7FFF # For 16 bits
  ShouldBeOnFreeList = 0x8000# For 16 bits
```

ShouldBeOnFreeList tiene todos sus bits a 0 menos el del bit más significativo, mientras que *RefsMask* sirve de máscara de bits para aquellos bits que sirvan para contar referencias, siendo el valor negado de *ShouldBeOnFreeList*.

A continuación vamos a hacer la explicación de las funciones, primero analizaremos *add*, la función que toma un nodo desde fuera y lo añade a la lista:

```
proc add*[T](self: var FreeList[T], data: ptr T) {.inline.} =
  # As long as there's no error by part of the programmer, we can assume a ptr to
  # T is the same as a ptr to the Node, becasue is the start of it.
  let node = cast [ptr Node[T]](data)
  # We know that the should-be-on-freelist bit is 0 at this point, so it's safe
  # to set it using a fetch_add


  if valInc( node.refs ,ShouldBeOnFreeList , MemOrder.Release ) == 0:

    # We were the last ones referencing this node, and we know we want to add it
    # to the free list, so let's do it!
    self.addKnowingRefcountIsZero(node)
```

```
proc addKnowingRefcountIsZero[T](self: var FreeList[T], node: ptr Node[T]) {.inline.} =

  var head = self.head.value(MemOrder.Relaxed)
  while true:
    node.next.value = (head, MemOrder.Relaxed)
    node.refs.value = (1, MemOrder.Relaxed)
    if not self.head.compareExchangeStrong(head, node, MemOrder.Relaxed,
     MemOrder.Relaxed):

      # Hmmm, the add failed but we can only try again when the
      # refcount goes back to 1
      if valInc(node.refs ,ShouldBeOnFreeList - 1 , MemOrder.Relaxed ) == 1:
        continue

    return
```

1. Añade al valor de referencia se le suma el de *ShouldBeOnFreeList*, si el valor anterior era diferente a 0 para (en el código aquí es dónde llama a *addKnowingRefCountIsZero*).

2. Obtiene la cabeza.

3. Asigna uno a las referencias del nodo, y asigna al valor siguiente del nuevo nodo la antigua cabeza.

4. Intenta cambiar la cabeza existente por el nodo, si la cabeza no ha cambiado termina.

5. Le añade al valor de referencia el valor del bit más significativo menos uno , dejando todos los bits a 1 menos el más significativo y lo añade al número de referencias, si antes de esta operación este valor era 1 vuelve a intentarlo, de otra forma significará que algún otro lo está usando, por tanto parará la operación y dejará que quién haya vuelto a usar el nodo vuelva a realizar la operación.

5.10. Eventos

Para permitir la ejecución de código en ciertos momentos se ha añadido al núcleo una característica llamada evento. Estos eventos permiten la subscripción a ciertas ocurrencias dentro del sistema, cuando estas ocurran todas las funciones que hayan sido suscritas al evento en cuestión serán llamadas a ejecución permitiendo reaccionar ante estos eventos.

Cualquier módulo puede crear y llamar sus propios eventos, aunque existen eventos que son llamados por el núcleo. A continuación realizaremos un repaso sobre estos eventos, estos eventos se mantendrán en el orden de ejecución permitiendo obtener una vista del sistema de inicio de *Tart*:

- Antes incluso de cualquier evento, en el inicio del programa se realiza una llamada a todas las operaciones fuera de funciones de todos los módulos, esto es realizado por el propio lenguaje.

- *Thread Init:* Este evento marca que un *thread* está siendo iniciado, en el inicio es el primer evento en ser llamado y se realiza una sola vez por *thread*. Este evento es para código de inicialización que debe ser una llamado una vez por *thread* (necesario, por ejemplo, en casos donde se utiliza memoria separada para cada uno de ellos).

- *Game Init*: Este evento se llama justo después de que *thread* haya terminado su inicialización. Este evento desarrolla la inicialización del juego en sí o las librerías de las que depende .

- *Allegro Init*: Este evento está disponible en la implementación actual debido a que es especifico para el uso de *Allegro* y es muy posible que no se encuentre en un futuro cuando se cambie de tecnología. Este evento es llamado justo después de *Game Init* y se utiliza para el inicio de módulos de *Allegro* y otras variables que deben ser ejecutados luego de la base.

- *Game Start*: Una vez todo ha sido inicializado, es momento de iniciar la lógica del juego (carga, primera pantalla, menú …) y este inicio es marcado por *Game Start*.

- *Frame Render*: Cada cierto tiempo la pantalla debe ser redibujada con el contenido actualizado, además de realizar nuevos cálculos (como podrían ser la velocidad y la aceleración).

- *Game End*: Una vez el usuario decide terminar el programa, este evento se activa y avisa de que es momento de empezar a limpiar todo lo que estén usando.

Todos los eventos permiten que cualquier módulo pueda suscribirse a cualquier evento, de hecho, no existe ninguna restricción en este aspecto y un módulo puede suscribirse cuantas veces quiera a un mismo evento (usando diferentes funciones), aunque esto es posible, se recomienda el uso de esto con precaución, puesto que si el uso de esto se determina necesario muy probablemente el módulo sea demasiado grande y deba ser reestructurado en nuevos módulos.

Los eventos serán representados por cadenas de caracteres, se ha elegido este método por ser un método muy flexible frente a otras opciones como la numeración que impondría limitaciones y complicaciones innecesarias en la creación de módulos (como la necesidad de a cada eventos se le asignase un número único).

Debido a que los eventos serán identificados por cadenas de caracteres es necesaria una estructura capaz de relacionar las entradas con los eventos. La estructura generalmente más usada para este tipo de situaciones es la tabla, en este caso se utilizará la tabla *SharedTable* desarrollada como parte de este proyecto.

Se necesita de una estructura lista para el almacenamiento de punteros (en concreto punteros a funciones para que sean ejecutadas), esta estructura parte de un *array* para el almacenamiento de punteros. Sin embargo, nuestras necesidades pasan por saber el número de elementos que se encuentran guardados, para ello, será necesario un contador. Además, un necesidad bastante usual en el sistema de eventos es realizar una acción (esta acción generalmente es una llamada a otro evento) cuando se hayan terminado de procesador todas las funciones del evento, para implementar esto añadiremos a esta estructura un puntero a función además del que ya existe. El resultado será la estructura *WordQueue*.

5.11. Modular

Uno de los objetivos era que Tart tuviese una arquitectura que permitiese el cambio de trozos de videojuegos de una forna sencilla, que se pudiesen reaprovechar y que el núcleo no se encontrase excesivamente ligado a la aplicación en concreto.

Para la parte del núcleo lo que se ha hecho es minimizar la relación que tiene con el sistema mediante el uso de parámetros genéricos y argumentos para sus funciones, esto es especialmente cierto con el planificador.

```
proc init*(threadInit: JobFunction,whenFinished: ptr Job, onFrameRender: Jobfunction) =
  initialJob = threadInit

  var threadNum: int
  for thread in mitems(workerThreads, workerThreadsCount - 1):

    createThread(thread, workerThreadProc, (threadNum,initialJob,whenFinished) )
    pinToCpu(thread, threadNum)
    inc threadNum

  while schedRunning and atLeastOneThreadRunning():
    sleep(16)
    sleepLine.signal()
    loadMainThreadJob( newMainThreadJob(onFrameRender) )
    mainThreadProcess()
```

La función que se puede ver en la imagen es la de inicialización del planificador, un detalle que se ha tenido en cuenta es evitar una dependencia del planificador con el resto del código al mantener las referencias necesarias como argumentos.

En cuánto a los programas *Tart* se encuentra diseñado para permitir que más allá de la funcionalidad que permite ejecutar tareas de forma concurrente (el planificador y el sistema de eventos) todo el resto de funcionalidad (gestión de texturas y otros *assets*, renderizado, gestión de ventanas …) sea realizado de forma independiente por módulos. Cada uno de estos implementará una funcionalidad, por supuesto, dentro del sistema se permite que un módulo importe a otro para usar su funcionalidad, lo que pretende este sistema es desacoplar aquellos módulos aquellos módulos que no tengan una dependencia real entre ellos, para aquellos que tienen una dependencia se ofrecen dos alternativas:

- Compilación condicional: Añadir soporte para que dicho módulo pueda funcionar sin el módulo del que depende mediante la detección del módulo en tiempo de compilación (todo esto es posible gracias al soporte de metaprogramación de Nim).

- Interfaces genéricas: Para módulos que cumplen un propósito general(gráficos, sonido, sistema de ventanas) se propone la creación de interfaces comunes que permitan el intercambio entre distintos módulos de un mismo tipo.

5.11.1. Módulos

La idea de modularidad de *Tart* hace que sus módulos puedan ser descritos en términos simples como conjunto de datos (por lo general no de muy distinto tipo) que pretenden implementar una funcionalidad específica, estos conjuntos de datos tienden a encontrarse en *arrays* y otros conjuntos básicos, puesto, que son la forma de acceso más eficiente y compacta.

Recordemos que este almacenaje de datos está respaldado por el diseño orientado a datos y es que este tipo de almacenaje no solo permite mejorar el uso de caché, si no que además permite el uso directo de instrucciones *SIMD*, facilitando la mejora de la eficiencia en el motor.

5.11.2. Graphics2D

Graphics2D , referenciando a gráficos en dos dimensiones, es el módulo de *Tart* más importante. Se encarga de almacenar información para el dibujado, en concreto la imagen y la posición de cada objeto a dibujar. *Graphics2D* introduce en *Tart* un nuevo concepto: el del *Sprite*, siendo este una representación de un objeto para ser dibujado.

Como ya se ha dicho antes, *Tart* pretende asentar sus bases sobre el diseño orientado a datos, por tanto, queda una pregunta, como relacionar las distintas informaciones sobre un mismo objeto. La respuesta más directa a esta, es el uso de un índice, es decir, almacenar la posición dentro de los *arrays* *(* manteniendo la misma posición en todos ellos), esto tiene el problema de que no es posible mover este *Sprite* una vez creado.

La incapacidad de mover *Sprites* de implica que para la eliminación de estos se deber marcar cada uno en si está siendo usado o no, implicando que se deba revisar en cada iteración este valor, esto va en contra del diseño orientado a datos, pues que se necesitaría de una comprobación para cada uno de los elementos en una iteración, degradando el rendimiento.

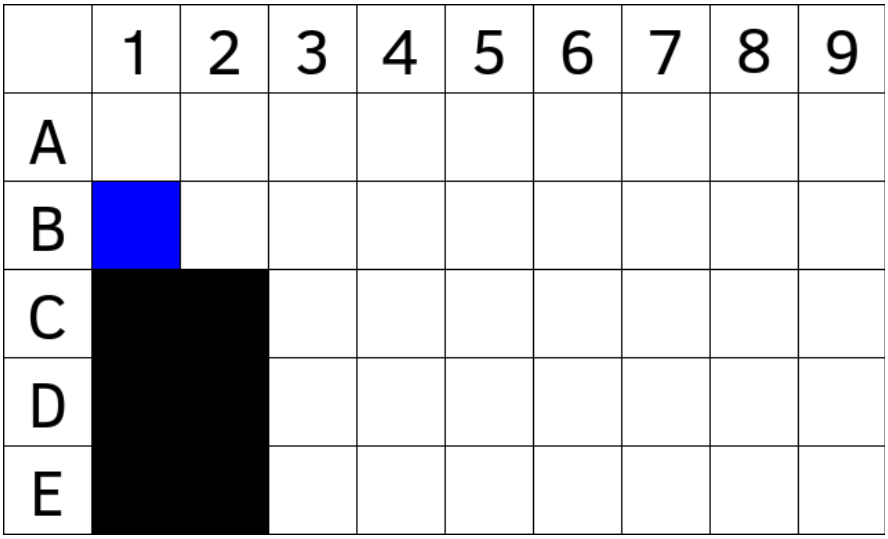
La solución por la que se ha optado es utilizar identificadores para cada *Sprite*, de esta forma todos los accesos aleatorios a estos elementos pasan primero por la tabla de relaciones de cada identificador a su posición real.

Para la implementación de la tabla se usará la tabla desarrollada para este proyecto, la *SharedTable*, elegida por sus capacidades para el acceso y modificación concurrente. La tabla acepta y devuelve enteros, siendo su entrada los identificadores y su salida los índice correspondientes a en los *arrays*. Esta es la definición de dicha tabla:

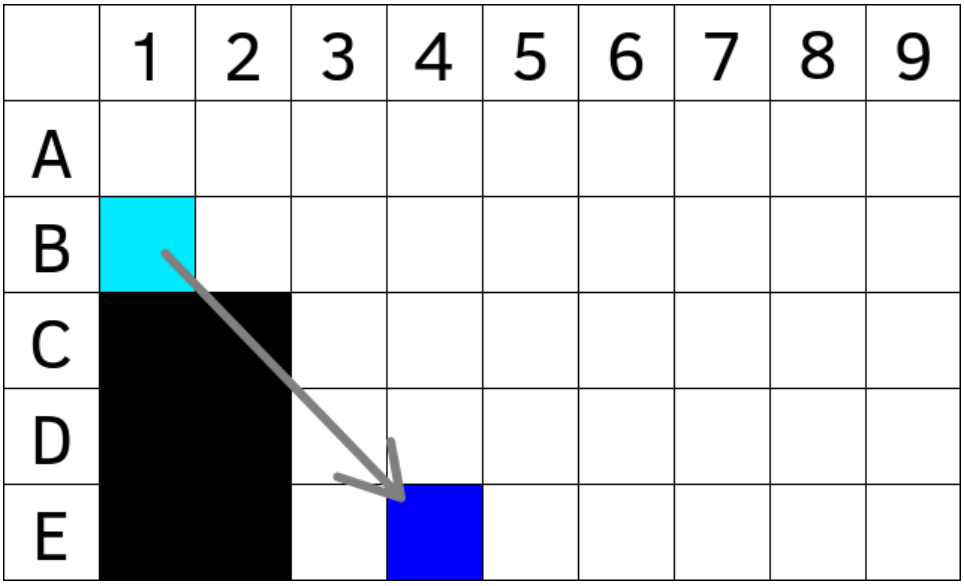
```
var bitmapTable: SharedTableAtomic[uint32,uint]
```

Nótese que *SharedTable* tiene de hecho dos implementaciones y en este caso se usa la versión *SharedTableAtomic* para indicar que ambos valores deben ser tratados de forma atómica.

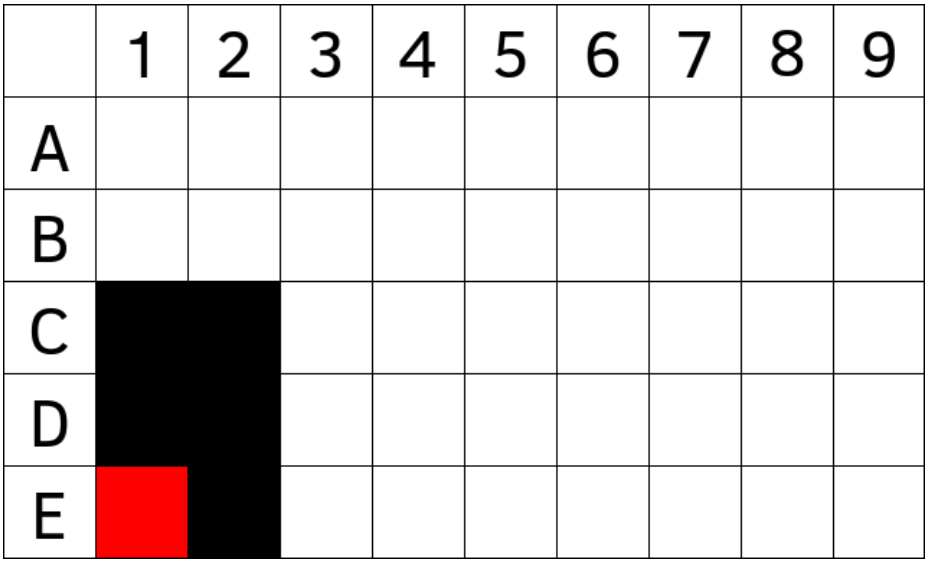
Otro problema que existe es que aunque gracias a las variables atómicas se puede asegurar que no se accede a un número en medio de su actualización (es decir, que al obtener el valor no se va a obtener parte del valor anterior y parte del nuevo) debido a que la posición está formada por más de un número puede ocurrir que se obtenga un valor incorrecto dando lugar a efectos adversos, veamos un ejemplo. Pongamos el siguiente caso:



Para este caso los bloques negros son bloques no accesibles (partes sólidas, fuera del mapa…) y el bloque azul es el jugador que vemos que se encuentra en 1B, pongamos el caso de que durante un *frame* se mueve a 5E :



Este sería el resultado, sin embargo, dado que no existe ningún método de sincronización entre ambas coordenadas, algún módulo podría acabar tomando una posición incorrecta como 1E debido a que se ha actualizado solamente la coordenada Y pero no la X provocando que el personaje se encuentre en un lugar del mapa no accesible:



En esta imagen se ha marcado al jugador en rojo para indicar que se encuentra en una posición en la que no se puede encontrar. Dependiendo del caso esto puede ser más o menos grave, pero pongamos que esto le ocurriese al motor de físicas (el que se encarga de que en los juegos haya interacciones físicas entre elementos: que las cajas se muevan al tocar el personaje, que el personaje se pare al tocar el suelo o incluso la propia gravedad es todo debido al motor de físicas), si se tratase de un bloque sólido generase dentro de este una gran fuerza que mandase al personaje literalmente por los aires, esto es peligroso puesto que puede llegar a romper severamente el *gameplay* (la experiencia de juego que tiene el jugador).

Qué solución se le puede dar a esto? Una solución bastante eficiente podría ser el empaquetamiento de ambas coordenadas en un solo valor de 64 bits para el acceso a este de forma atómica, esto funcionaría en todos los sistemas de 64 bits (siendo estos un alto porcentaje de todos los dispositivos de consumo) , sin embargo, en los sistemas de 32 bits no sería posible hacer esto (dejando de lado muchos dispositivos móviles) al mismo tiempo que sería impracticable para el renderizado en tres dimensiones, por tanto, esta opción es inviable como solución, sin embargo, más adelante puede ser utilizada como optimización para determinados sistemas. Por supuesto, esta misma limitación se puede aplicar a todas las herramientas *lock-free* dado que sufren las mismas limitaciones.

Dado que no es posible implementar este sistema bajo un sistema *lock-free*, es necesario recurrir a

```
proc drawText*(sprite: Sprite, str: string, color: Color = white , fontAlign = FontAlignLeft ) =
  let currentFont = font.value
  let fontBitmap = createBitmap(currentFont.getTextWidth(str) , currentFont.getFontLineHeight())

  withLock drawLock:

    let oldDisplay = getTargetBitmap()
    fontBitmap.setTargetBitmap()

    font.value.drawText color, 0,0, fontAlign, str

    oldDisplay.setTargetBitmap()
    sprite.bitmap = fontBitmap
```

bloqueos. De esta forma se puede mantener la coherencia en el sistema de coordenadas.

Un último requerimiento en los videojuegos (en especial en los de dos dimensiones) es la existencia de múltiples capas. Aunque no se pretende imponer ninguna restricción en este aspecto, *Tart*, por defecto supone que la gran mayoría del dibujado se encuentra en la tercera capa, suponiendo que la primera capa pueda ser usada para la interfaz de usuario y la segunda para efectos visuales. Sin embargo, la implementación actual no puede cambiar el tamaño de estos vectores de forma dinámica, por esta razón tanto el número de capas como los *Sprites* que cada una de ellas puede almacenar se encuentra limitado. La resolución de esto implicaría la introducción de sistemas de recolección de basura, esto queda fuera del alcance de este trabajo.

A continuación la definición de los distintos *arrays* usados en este módulo:

```
var arraysLock: Lock
var layersBitmap {.guard: arraysLocks.}: RtArray[RtArray[PBitmap]]
var layersX {.guard: arraysLocks.}: RtArray[RtArray[float]]
var layersY {.guard: arraysLocks.}: RtArray[RtArray[float]]
var layersNumElements: RtArray[Atomic[uint16]]
```

En la imagen se pueden apreciar cuatro *arrays*, tres de ellos protegidos se encuentran marcados con el *pragma* (un tipo de anotación existente en *Nim*) *guard* que obliga a que el uso de cualquier variable con la que se usa a que sus accesos se realicen dentro del *lock* que tienen asignado. Estos *arrays* con *lock*s almacenan punteros a la imagen y las coordenadas X e Y, el *array* sin *guard* (*layersNumElements*) se usa como contador del número de elementos existentes para los demás *arrays*. El último elemento que queda por nombrar es el *lock* encargado de proteger los *arrays*, *arraysLock.*

Nótese que el tipo *Lock* hace referencia a un mútex. Una limitación de la actual implementación es que no distingue accesos, tratando todos los accesos como iguales, en el caso de que esta implementación se siguiese usando en el futuro, sería recomendable el uso de estructuras de *locks* con un acceso más eficiente.

5.11.3. Tipografía

El modulo *typography* implementa una interfaz para el dibujado de fuentes, aunque basa su funcionalidad en *Allegro* se pretende ocultar este hecho para los demás módulos que dependen de él. Un ejemplo muy claro de está abstracción es *FontAlign* que reutiliza la alineación de que el módulo de tipografía de *Allegro* usa:

```
type FontAlign* {.pure.}= enum
  Left = FontAlignLeft, Center = FontAlignCenter , Right = FontAlignRight,
  LeftPixel   = FontAlignLeft   or FontAlignInteger
  CenterPixel = FontAlignCenter or FontAlignInteger
  RightPixel  = FontAlignRight  or FontAlignInteger
```

Actualmente este es un módulo pequeño, al tratarse solo de una interfaz, aunque se puede apreciar algo interesante, y es el uso de eventos:

```
var font: Atomic[PFont]

const textFont = "/usr/share/fonts/truetype/sourcecode/SourceCodePro-Medium.ttf"
proc typographyInit(self: ptr Job) {.onEvent:"allegro init".} =
  font.value = al.loadFont(textFont, 46, 0)

  if font.value == nil:
      raise newException(Exception,"Failed to load font")
```

Esta función inicializa el sistema de dibujado de texto al cargar la tipografía que se usará, en este caso usará una predeterminada del sistema. Nótese que esta función se encuentra adherida a un evento, en este caso “allegro init” que se produce luego que la base de *Allegro* esté lista y el sistema esté preparado para que el resto sea inicializado.

La función más importante de cara al usuario es *drawText* que dibujará en el *Sprite* que se le pida un texto especificado:

```
proc drawText*(sprite: Sprite, str: string, color: Color = white , fontAlign = FontAlignLeft ) =
  let currentFont = font.value
  let fontBitmap = createBitmap(currentFont.getTextWidth(str) , currentFont.getFontLineHeight())

  withLock drawLock:

    let oldDisplay = getTargetBitmap()
    fontBitmap.setTargetBitmap()

    font.value.drawText color, 0,0, fontAlign, str

    oldDisplay.setTargetBitmap()
    sprite.bitmap = fontBitmap
```

5.11.4. Window

Este modulo se encarga de la inicialización de la ventana. Por el momento también se encarga de la inicialización inicial de *Allegro* debido a que las funciones usadas actualmente requieren de la ventana.

6. Ejemplos

Toda tecnología necesita de ejemplos que suelen tener dos intenciones: la primera demostrar su funcionamiento y la segunda enseñar como és la programación con el sistema propuesto. Para estas demostraciones se han desarrollado dos ejemplos en *Tart*, uno de ejemplo que permita una introducción sencilla y otro más complejo para dar una mejor imagen de como es la programación con *Tart* actualmente.

6.1. Hola mundo

Cuando se inicia en un lenguaje o marco de trabajo nuevo, una tradición existente es crear el ejemplo más simple que simplemente muestre una frase generalmente “Hello world” (“Hola mundo” en español) la razón de esto es que de esta forma se obtiene un ejemplo sencillo que realiza una acción, permitiendo al estudiante entender las acciones así como experimentar desde el inicio, este tipo de ejemplos dado que se ha convertido en un estándar permite a alguien con experiencia avanzar más rápido al ser capaz de ver de una forma cuál es un patrón de uso sencillo.

Para *Tart* se seguirá el mismo procedimiento:

En este caso hemos renderizado por pantalla”Hello from *Tart*” , a continuación el código:

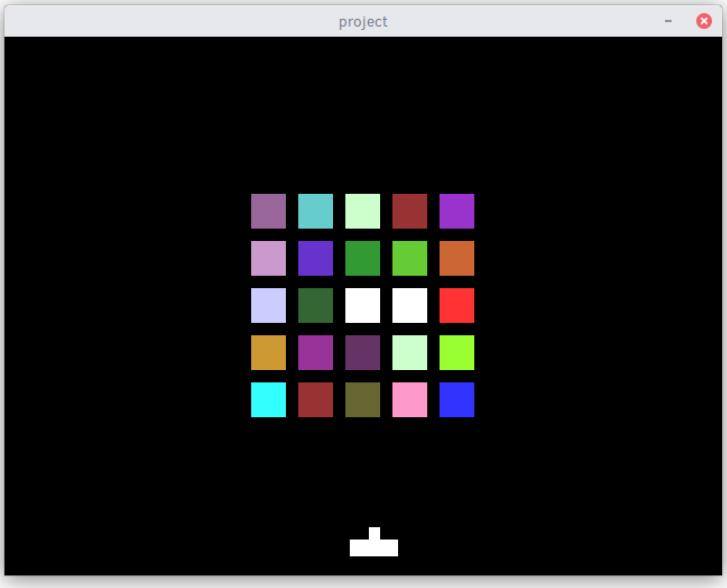
```
import typography
import graphics2d
import render2d
import tartapi

proc testGame*(self: ptr Job){.onEvent: "Game Start".} =
  let text = graphics2d.createSprite(3)
  text.drawText("Hello from Tart!")
  text.x = 50
  text.y = 50
```

El código es simple, una función que se suscribe al evento “*Game Start*” para ser ejecutada en cuanto el juego esté iniciado, las siguientes funciones obtienen un gráfico y cambian su posición, esto solo se realiza una vez y se mantiene así durante toda la ejecución.

6.2. Primer juego

Es momento de realizar un ejemplo que permita ver el potencial que posee *Tart*. Nótese que *Tart* aún se encuentra en una fase de desarrollo muy temprana y por tanto aún existen bordes por pulir.



La idea es que esta nave rompa los cubos mediante disparos. Nótese que todos los elementos son generados de forma procedural. A continuación se proveerá un fragmento del código:

```
proc shipGame*(self: ptr Job){.onEvent: "Game Start".} =

  randomize()

  counter.value = blocksH * blocksV

  ship.value = graphics2d.createSprite(2)

  bullet.value =  graphics2d.createSprite(4)

  let newBulletBitmap = createBitmap(16,16)

  drawAt newBulletBitmap:
    drawFilledRectangle(4,0,12,15,mapRgb(255,255,255))

  bullet.value.exchangeBitmap( newBulletBitmap )

  let newShipBitmap = createBitmap(64,64)

  drawAt newShipBitmap:
    drawFilledRectangle(10,48,53,63, mapRgb(255,255,255)) # Body
    drawFilledRectangle(27,37,37,48, mapRgb(255,255,255)) # Canon

  ship.value.exchangeBitmap( newShipBitmap )

  ship.value.y = 400
  ship.value.x = 298

  # Calculate how much in the horizontalsides is empty
  let blocksSizeH = blocksH * 32
  let blocksSpaceH = (blocksH - 1) * 10
  let occupiedSpaceH = blocksSizeH + blocksSpaceH
  let sideEmptySpaceH = (windowWidth - occupiedSpaceH)/2

  # Same for vertical
  let blocksSizeV = blocksV * 32
  let blocksSpaceV = (blocksV - 1) * 10
  let occupiedSpaceV = blocksSizeV + blocksSpaceV
  let sideEmptySpaceV = (windowHeight - occupiedSpaceV)/2

  for v in 0..blocksV - 1:
    for h in 0..blocksH - 1:
      let newBlock = graphics2d.createSprite(2)
      newBlock.x = sideEmptySpaceH + float(h * (32 + 10))
      newBlock.y = sideEmptySpaceV + float(v * (32 + 10))
      newBlock.bitmap = randomBlock()


      (cast [var Atomic[uint]](addr blocks[v*blocksH + h] )).value = newBlock
```

Este código reutiliza conceptos vistos anteriormente en *Nim* tales como el uso de eventos, o operaciones atómicas. En términos básicos este código inicializa los *Sprites* para que sean usados más tarde en el juego.

Este ejemplo se encuentra dentro del código de *Tart*, hospedado en *GitHub*[25].

7. Trabajo futuro

Un software solo está terminado una vez deja de ser usado, por tanto, cualquier software se encuentra en continua evolución, esto es especialmente en grandes sistemas de software como un motor de videojuegos, debido al sinfín número de herramientas que puede aportar el trabajo que se puede llegar a realizar en él es infinito, sin embargo, a continuación serán comentadas funcionalidades que podrían ser añadidas en un futuro.

7.1. Limpieza y documentación

Tart aún se encuentra en desarrollo inicial, esto implica que aún tiene funciones por implementar, documentación por escribir y tests por programar. Aunque todo esto es menor en sí mismo, es necesario que un sistema que pretende ser expandido se encuentre en las mejores condiciones, especialmente si luego piensa ser software libre. Esto es , por tanto, algo para realizar tan pronto como sea posible.

7.2. Reemplazo de módulos

Los módulos actuales dependen ampliamente de *locks*, algo que contrasta ampliamente con el objetivo de *multi threading* de *Tart*, además, *Allegro* hace uso de algunos *threads* para ciertas tareas, dado que *Tart* ya pretende emplear todos los *threads*, más *threads* pueden empeorar el rendimiento.

Además el reescribir estos módulos puede permitir un mayor control , muy importante para portar *Tart* a otras plataformas.

7.3. Extensión a otras plataformas

Por el momento *Tart* solo ha sido probado en *Linux*, puesto que es el sistema operativo que el autor usa para desarrollo de software, sin embargo, un motor que se centre principalmente en *Linux* como objetivo será muy probablemente de poco interés para el público general, por eso, uno de los objetivos a largo plazo es el soporte de otras plataformas.

Al mismo tiempo, una parte importante de *Tart*, *AtomicWrapper*, solo tiene soporte para *GCC* (y otros compiladores basados en él), por tanto, no es posible el uso de otros compiladores para esto, en concreto, el de mayor interés es *MSC*, el compilador de *Microsoft*.

Una hoja de ruta para esta extensión sería: primero a otros *Unix* puesto que muy probablemente sean los que más sencillos de portar sean (*BSD y macOS*), tras esto el siguiente objetivo sería *Windows* a través de *MinGW* (el compilador de *GCC* para Windows) y más tarde a través de *MSC*, en un objetivo más a medio-largo plazo estarían los objetivos móviles *Android* y *iOS*.

A pesar de la alta popularidad de los dispositivos móviles, *Android* y *iOS* se han dejado para el final debido a que existe un requerimiento previo para funcionar sobre este tipo de dispositivos.

7.4. Extensión a otras arquitecturas

Por el momento *Tart* se encuentra limitado a la arquitectura *Intel (*desarrollada actualmente por *Intel* y *AMD*), esto se debe a que la implementación actual no toma en consideración en varias de sus estructuras ciertos problemas de sincronización que en arquitecturas con un modelo de memoria fuerte como la de *Intel* no ocurren (debido a que todas las lecturas ejercen un *acquire* y todas las escrituras ejercen un *release*, estas semántica permite al desarrollador despreocuparse de errores de reordenación) y que sí ocurren en arquitecturas con un modelo más débil como en la arquitectura *ARM* usada por los dispositivos móviles.

Por tanto, una vez *Tart* empiece a estabilizarse es de gran interés realizar revisiones y pruebas para que pueda funcionar bajo este tipo de dispositivos.

7.5. Tres dimensiones

Debido a que *Tart* carece de renderizado propio y basa el suyo en el de *Allegro* solo puede hacer renderizado en dos dimensiones, aunque es un buen comienzo es altamente limitante, debido a la popularidad y la extensión de productos con soporte para tres dimensiones, especialmente con el auge de las plataformas de realidad virtual.

Por tanto, un gran reto que a *Tart* le espera es la implementación de módulos que den soporte a sistemas de tres dimensiones. Desgraciadamente esta es una gran tarea, puesto que el estándar gráfico actual es , sin duda, altamente complejo pues pretende seguir el realismo más absoluto implementando técnicas muy avanzadas, como *PBR.*

Es irreal pretender que *Tart* gane esta funcionalidad pronto, el objetivo es ,por tanto, la implementación de un renderizador 3D modesto, que permita una visualización de gráficos aceptable, pero basado en principios modernos y pensado para ser extendido fácilmente.

7.6. Realidad virtual

Como se ha mencionado anteriormente se está produciendo un auge en la tecnología de virtualización, además los expertos creen que este auge aumentará en los próximos años. Esta es , sin duda, una gran oportunidad al mismo tiempo que una funcionalidad demandada, con tal de responder a la demanda de posibles usuarios sería conveniente que *Tart* dispusiese de soporte para estos dispositivos de realidad virtual a largo plazo.

7.7. Soporte para Web

Una característica que parece tener un uso aceptable es la capacidad de que el videojuego sea capaz de correr en un navegador Web, siendo una gran opción para videojuegos pequeños. Muy probablemente esta opción será aún más popular en el futuro con la introducción de *WebAssembly* (un lenguaje para la ejecución de código de forma universal con una eficiencia cercana a los lenguajes de programación tradicionales) y una *API* de renderizado de bajo nivel (por el momento la disponible es *WebGL*, una

versión especializada de la *API OpenGL ES*, pensada para dispositivos móviles) que permita obtener un mejor desempeño gráfico.

Esta es una característica que podría ser implementada a largo plazo, especialmente una vez se haya producido la introducción y expansión de las mencionadas tecnologías.

7.8. Editor

Tart pretende diferenciarse centrándose en la facilidad de programación, la eficiencia y el aprovechamiento del microprocesador, sin embargo, para el acercamiento a un público más general es necesario un programa capaz de permitir la edición de videojuegos de la forma más simple posible, esto es un editor.

Los editores que existen a día de hoy exhiben un comportamiento *WYSIWYG* en la edición de escenas, sin embargo, el modelo de testeo que proponen es hacer modificaciones, para luego probarlas y ver queda por mejorar, cambiando entre el videojuego y el videojuego con cada iteración.

Aunque se trata de una mejora frente a los modelos tradicionales (edición de código, compilación, pruebas , muchas veces estas pruebas no están localizadas) puesto que eliminan , o reducen ampliamente, la compilación y facilitan el testeo (algunos incluso permiten el cambio de algunos valores en tiempo real), este modelo puede ser mejorado mediante la programación en vivo.

La programación en vivo es un concepto popularizado por *SmallTalk*, un lenguaje de programación nacido en los años 80 y que aún sigue en desarrollo y con una buena comunidad tras él. Es conocido por su sintaxis fácilmente legible, su filosofía (dinámico y 100% orientado a objetos) y su característica única de ser ejecutado en un entorno completamente escrito en este mismo lenguaje y que puede ser modificado en tiempo real. A día de hoy existen varias implementaciones de *SmallTalk*, aunque la primera en aparecer en una búsqueda sea *Squeak*[26], el autor recomienda el uso de *Pharo*[27] para todo aquel que desee probar este interesante lenguaje.

Basándose en esta característica, un concepto muy interesante sería el de un editor que permitiese la modificación del juego en vivo, de esta forma el desarrollador probaría sus modificaciones sin casi ninguna demora, no solo aumentando el ritmo al que avanzan los ciclos, si no también aumentando su

nivel de interés y reduciendo el nivel de desgaste por el desarrollo debido a que los cambios son instantáneos mejorando significativamente el ciclo de *feedback* que deseamos los seres humanos.

Estas mejoras implicarían una mayor eficiencia y un menor coste de desarrollo al tiempo que se mejoraría la satisfacción del desarrollador, permitiendo a este ofrecer juegos de mayor calidad en un tiempo igual o menor. Al mismo tiempo este desarrollo permitiría el desarrollo de juegos con mecánicas altamente interesantes al proveer bloques para la construcción de videojuegos modificables en tiempo real.

Este es un gran objetivo, desafortunadamente la implementación del mismo en su máximo exponente de forma que permita el aprovechamiento de todas estas características implica la introducción de herramientas de cuya implementación no sería trivial: pausa, retroceder en el tiempo, un sistema de programación en vivo, editor de código … Por tanto, este es un objetivo muy a largo plazo.

7.9. Lenguaje de *scripting*

Para la implementación del editor es indispensable un lenguaje de *scripting* que permita el desarrollo en vivo, aunque esta no es la única razón, el desarrollo de videojuegos es altamente complejo en el día de hoy, realizar videojuegos que se encuentren a la altura de las expectativas es un gran reto, además los consumidores esperan pagar poco por un gran juego que les ofrezca una gran cantidad de contenido, por tanto, es necesario minimizar el tiempo de desarrollo de un videojuego tanto como sea posible.

Aunque este aspecto ha ido a mejor en los últimos años, aún se siguen usando lenguajes que en un principio no estaban pensados para la escena actual, muchos de estos lenguajes han sido elegido en muchos casos debido a su popularidad.

Al mismo tiempo no se debe olvidar que este lenguaje influirá en el rendimiento del juego haciendo que sea imprescindible en el rendimiento que tendrá, cualquiera que sea el lenguaje y la arquitectura alrededor de él va a ser necesario que sean capaces de sacar provecho de la capacidad de procesamiento ofrecida por *Tart*.

Al mismo tiempo, para asegurar la eficiencia de este lenguaje sería de alto interés que este lenguaje permitiese al desarrollador el uso de programación orientada objetos y que internamente transformase esta estructura a una basada en el diseño orientado a datos.

Debido a su uso en un entorno altamente dinámico y lleno de objetos, el autor de este trabajo considera que debería tratarse de un lenguaje con tipos dinámico y que implementase orientación a objetos mediante prototipos. Un lenguaje con este tipo de diseño no tienen clases, en su lugar se usan objetos como plantilla, y estos se copian o referencian para que se hagan uso de ellos, lenguajes como *Lua* o *JavaScript* implementan este tipo de ideas, aunque para la base de este lenguaje el autor recomienda un lenguaje basado en el lenguaje de programación *Io*[28] un lenguaje que combina prototipado ,una simpleza extrema y una alta flexibilidad.

8. Más información

Uno de los objetivos que tenía el autor era que todo el código fuese puesto bajo una licencia de software libre, de esta forma, *Tart* podrá llegar a más gente y tener muchos más usos, además de tener mejores oportunidades de convertirse en un motor que pueda competir con los comerciales.

Debido a esto, todo aquel que desee entender más a fondo *Tart* puede hacerlo visitando la página en *GitHub* de *Tart[25]*, de la misma forma, cualquier comentario constructivo que se desee hacer sobre cualquier aspecto de *Tart* será recibido gratamente.

Bibliografía

[1] https://www.libsdl.org
[2] https://www.sfml-dev.org
[3] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3433177/
[4] https://libres.uncg.edu/ir/asu/listing.aspx?id=6977
[5] http://mashable.com/2013/04/04/video-games-pain/#2vyNs4AaJOqo
[6] http://www.dataorienteddesign.com/dodmain/node3.html
[7] https://unity3d.com/es
[8] https://www.unrealengine.com/
[9] https://www.cryengine.com
[10] https://aws.amazon.com/es/lumberyard/
[11] http://www.ogre3d.org
[12] https://godotengine.org
[13] http://itscompiling.eu/2017/01/12/precompiled-headers-cpp-compilation/
[14] https://github.com/andreaferretti/rosencrantz

[15] https://www.xataka.com/moviles/10-nucleos-en-un-smartphone-asi-es-el-helio-x20-de-mediatek-para-asaltar-la-gama-alta
[16] https://www.amd.com/es/products/cpu/amd-ryzen-7-1800x
[17] https://www.xataka.com/componentes/intel-no-quiere-que-miremos-a-amd-ryzen-y-contraataca-con-nuevos-procesadores-core-x-core-i9-con-18-nucleos
[18] https://pdfs.semanticscholar.org/6ebe/c8701893a6770eb0e19a0d4a732852c86256.pdf
[19] http://liballeg.org
[20] https://blog.molecular-matters.com/2015/08/24/job-system-2-0-lock-free-work-stealing-part-1-basics/
[21] https://blog.molecular-matters.com/2015/09/25/job-system-2-0-lock-free-work-stealing-part-3-going-lock-free/
[22] https://blog.molecular-matters.com/2015/09/08/job-system-2-0-lock-free-work-stealing-part-2-a-specialized-allocator/
[23] http://preshing.com/20130605/the-worlds-simplest-lock-free-hash-table/
[24] http://moodycamel.com/blog/2014/solving-the-aba-problem-for-lock-free-free-lists
[25] https://github.com/sheosi/tart
[26] http://squeak.org
[27] http://pharo.org
[28] http://iolanguage.org