Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Zentract: Engine con simulación discreta para videojuegos 2D

Proyecto Final de Carrera
Ingeniería Informática

Autor: Pablo de la Ossa Pérez
Director: Ramón Pascual Mollá Vayá
[08/04/2014]

Resumen

Este proyecto aúna el desarrollo de un engine en C++ para juegos en 2D haciendo uso de Box2D, aplicarle una mejora moderna y revolucionaria y finalmente hacer pruebas para conocer el impacto del mismo en una situación real.

Palabras clave: engine, videojuego, C++, Box 2D, RT-DESK, Allegro

Tabla de contenidos

Introducción

Es indiscutible que el ocio interactivo digital (a partir de ahora videojuegos, como comúnmente se les ha llamado) ha tenido una penetración en la vida cotidiana que habría sido imposible de prever.

Por mucho que desde el punto de vista de su usuario un videojuego pueda tener un aspecto alegre, juvenil y desenfadado, por detrás se esconde una obra de ingeniería con una cantidad de trabajo y complejidad de trabajo muy alta. Esto es así para videojuegos de todas las escalas: Desde los videojuegos hechos por un individuo como ejercicio personal, hasta las superproducciones que aúnan el trabajo de cientos de profesionales a lo largo y ancho del mundo.

Dentro de este contexto es donde nos situamos actualmente, y donde se desarrolla el presente proyecto final de carrera.

Motivación

El desarrollo de software es, desde la programación de un algoritmo específico hasta el diseño de la arquitectura de un sistema operativo, uno de los principales pilares de la ciencia de la computación.

De forma reciente el desarrollo de videojuegos se ha visto catapultado a una industria multimillonaria en la que cada pieza de software requiere el trabajo de equipos de decenas de personas. Aún así, el gremio sigue abarcando proyectos de todos los tamaños. Nunca se han dejado de llevar a cabo proyectos individuales (conocido coloquialmente como los *one-man army* o “ejércitos de una persona”) ni el trabajo de equipos pequeños de presupuestos variables, sean ambos pequeños proyectos o trabajos ambiciosos que se alargan durante años.

En estos casos, el desarrollo del engine sobre el que se construyen los juegos ha de estar muy bien enfocado desde el comienzo: Específico para cada proyecto, o reutilizable en varios proyectos distintos.

Por otra parte, que un videojuego sea divertido o no depende de que su diseño (las mecánicas) funcionen correctamente con el usuario. No importa cuán bien programado esté el software: Si el diseño tiene fallos serios, no será divertido y por lo tanto no cumplirá su función final. Además, y al contrario que la programación, el diseño de videojuegos tiene más componentes artísticos. Requiere de inspiración. Tener las

herramientas apropiadas para llevar a cabo un diseño es lo que permite a un profesional poder plasmar sus ideas.

La finalidad de desarrollar un engine como el que se propone en el presente proyecto final de carrera es tener una herramienta sólida, a la vez que versátil, con la cual los profesionales (o el profesional, ejército individual) puedan plasmar el diseño tal y como lo desea, con pocas limitaciones y con todas las ayudas posibles.

Estado del arte

Clasificación de los videojuegos

La clasificación por géneros depende enteramente del diseño. Desde el punto de vista del diseño de software y programación, el enfoque del engine cambia radicalmente dependiendo del diseño que se haya de satisfacer. A pesar de ello, el núcleo central del engine siempre será el llamado bucle principal.

A grandes rasgos, habrán juegos orientados a la acción (donde habrán mapas, entidades móviles, y una carga importante en los componentes que calcularán las interacciones físicas entre las leyes físicas y las entidades y el mapa). Y habrá juegos que no estén orientados a la acción, donde puede que lo importante sea un grafo dirigido representando una serie de mapas (juegos de aventura orientados a historia), o una matriz en la que se enmarca toda la interacción (juegos de puzzle).

Siempre hay experimentos que llevan esta clasificación a extremos inesperados, y además, ya de base, no es una diferenciación exacta. Aún así, es una forma apropiada de clasificar, como ya se dijo, a grandes rasgos.

El ordenador como plataforma

Se puede argüir que las máquinas recreativas y las consolas son computadores, pero se hará una distinción para sincronizar correctamente al lector con la mentalidad del gremio: Una consola solo sirve para reproducir juegos, un computador hará muchas otras cosas además de eso.

Todo comenzó en los computadores, y hoy en día es donde sigue estando todo el peso. A pesar de que el márketing y los intereses económicos hagan creer al ciudadano medio que las consolas son sinónimos de videojuego, no es así. Hay muchas razones para ello: El computador es un formato que en esencia nunca ha cambiado mientras que las consolas son totalmente caducas, todos los juegos que se desarrollen para consola pueden o podrán ser ejecutados en computador, es más fácil (o más bien es posible) publicar y llegar a todo el mundo en el computador mientras que no en las consolas, y

no hay formatos privados y cualquier persona puede realizar su proyecto sin que nadie se interponga. Además, aunque los componentes o ciertos detalles de la arquitectura evolucionen, se mantiene constante. Una vez conocido y estudiado el funcionamiento de un computador, es asumible la actualización constante. Hay más razones, pero estas son las principales.

Colateralmente, y como dato puntual que se asume interesante en este momento: Las nuevas consolas de las grandes empresas actuales, la Play Station 4 de Sony y la Xbox One de Microsoft, finalmente se han decantado por una arquitectura prácticamente igual a la de un computador. De esta forma finalmente convergen las consolas en los computadores, en la conocida como octava generación de consolas. Las diversas razones y consecuencias de estas decisiones no se discuten en el presente documento, pero no le resultará difícil al lector el suponerlas.

Todo esto se presta a considerar el ordenador como el principal objetivo cuando un desarrollador de videojuegos se plantea su entrada en el gremio.

Crítica al estado del arte

C++ sigue siendo el estándar para el desarrollo de videojuegos, y no parece que vaya a cambiar en un futuro cercano. Las modas vienen y van, creando distintas fluctuaciones en los requisitos y en los medios, pero C++ se mantiene la constante.

Tanto los proyectos multimillonarios más ambiciosos (que ya superan, tanto en coste como en recaudación, al cine) como los proyectos personales y experimentales, el hilo conductor es la idea, necesitan un engine y una programación detrás.

Actualmente, la estación de trabajo conocida como Unity (de desarrollo muy ágil, y permite tanto un trabajo muy bien cohesionado de todos los involucrados en el proyecto, así como poder compilar el trabajo para diversos sistemas y consolas) está entrando muy profundamente en el sector. Cubre un puesto que muchos fracasaron intentando cubrir, y de una forma decisivamente acertada. La programación de videojuegos es un proceso lento, difícil y muy poco agradecido, y con Unity se reduce considerablemente esta carga.

Aún así, en ningún momento entra en conflicto con C++, puesto que aunque orientados a un mismo objetivo, satisfacen necesidades distintas.

En el caso de necesitar C++ para un proyecto pequeño, en el que poca gente está involucrada, realizar el engine puede llegar a ser prohibitivo en costes. Y ante esta situación, actualmente, no hay soluciones: Hay que buscarse la vida. Esto fuerza a la gente a dedicar un tiempo muy grande en desarrollar el engine desde 0, buscar engines profesionales con precios muy elevados o ignorar los requisitos y dejar C++ por Unity.

El presente engine pretende solventar este problema, dotando a equipos pequeños y con el objetivo de programar un juego para ordenador de un engine robusto, bien documentado ylo suficientemente abierto, en C++.

Simulación con eventos discreta

Los videojuegos son un tipo de software con ciertas características en común que los definen y diferencian de otros tipos de software. La más importante y omnipresente de ellas, como ya se dijo, es la conocida como el bucle principal.

En todo momento, durante la ejecución de todos los videojuegos que han existido, se han de estar recorriendo cíclicamente a una serie de llamadas que, en esencia, son:

1. Entrada (input).
2. Procesado.
3. Salida (output).
4. Volver al paso 1 a menos que se haya terminado el juego.

Dando un poco más de detalle sobre cada uno de los pasos:

- Entrada: Adquirir datos del ratón, teclado, kinect, micrófono, etcétera. Aquí también se incluye el cálculo de las inteligencias artificiales, puesto que sustituyen a lo que sería la entrada de un jugador activo. Dicho cálculo de inteligencia artificial es la parte más intensa y costosa de este paso.
- Procesado: Calcular cómo afecta la entrada al estado actual del juego. Esto incluye el cálculo de las leyes físicas, la comprobación de la lógica interna del juego, etcétera. Por ejemplo, si tras la entrada de teclado un jugador se ha movido hacia un enemigo, es en este paso en el que se resolverá dicho conflicto. La lógica interna del juego puede decir que el jugador ha perdido la partida. El cálculo de las físicas suele ser la parte más intensa y costosa de esta parte.
- Salida: Devolver al usuario el resultado del procesado, principalmente por el monitor/televisión y los altavoces. También incluye, por ejemplo, los sistemas de vibración de los mandos modernos.

Este bucle principal permite al desarrollador tener un control absoluto sobre el funcionamiento del juego. Pero, a su vez, limita la posibilidad de innovación en el campo de una forma muy restrictiva. En esencia, todos los pasos se ejecutan de una forma constante, ininterrumpida y secuencial. Nunca puede romperse el ciclo. Supongamos el siguiente caso:

1. Entrada: El jugador no se mueve ni realiza ninguna acción. Las inteligencias artificiales han de calcular las decisiones que van a tomar respecto del estado anterior del juego, lo cual como se comentó antes es lo más intenso y costoso de este paso.
2. Procesado: A pesar de no haber ningún cambio, se re-ejecutan los métodos y todas las comprobaciones de la lógica interna del juego. También se re-ejecutan todos los cálculos de físicas en los que se comparan todos los cuerpos con todos que, como se dijo antes, es lo más costoso de esta parte.
3. Output: Se vuelve a calcular todo lo que hay que dibujar en pantalla y se

redibuja, por mucho que no haya ningún cambio. Aunque la música o sonido ambiental siga ejecutándose secuencialmente, los efectos sonoros no, lo cual es otro gasto de recursos innecesario.

Salta a la vista la cantidad de cálculos innecesarios y costosos que se llevan a cabo en cada iteración del bucle. Y más aún si se tiene en cuenta que, debido a la velocidad de reacción humana en contraste con la capacidad de computación de los computadores modernos, la mayor parte de las iteraciones serán como esta. Y, por último, se ha de tener en cuenta lo sumamente exigentes que se han vuelto los cálculos de inteligencia artificial, físicas y gráficos en los últimos años. Son, básicamente, los cuellos de botella.

Romper este bucle y permitir una ejecución desincronizada (bajo demanda) de la colección de elementos necesaria para ejecutar un videojuego resulta, pues, una evolución lógica del problema en cuestión.

Además, colateralmente, dicha ejecución desincronizada (discreta) permitiría ponderar y priorizar unos cálculos ante otros. Hay juegos donde lo más importante es la inteligencia artificial, otros en los que es el cálculo de las físicas, otros donde lo son los gráficos, etcétera. Así pues, es apropiado poder utilizar la mayor cantidad de potencia de cálculo en el más prioritario de ellos en una situación de cuello de botella, dando poca prioridad a los demás.

Ejecución continua (Bucle principal)	**Ejecución discreta (RT-Desk)**
Cada paso habrá de ejecutarse secuencialmente y en su totalidad, aún cuando no sea necesario.	Cada paso se podrá ejecutar bajo demanda, y según sea necesario. Podrán incluso auto-ejecutarse.
Cada cálculo atómico deberá realizarse haciendo uso del total de potencia computacional. Lo máximo que se podrá hacer es utilizar distintos núcleos con distintos hilos de ejecución, pero aún así se seguirá estando coartado por la secuencia continua de pasos.	Se podrá ponderar cada cálculo atómico según la prioridad que cada proyecto requiera. Así se asegurará que las partes realmente importantes recibirán la potencia computacional requerida, relegando otros cálculos no tan importantes.

Dichas ventajas nos conducen a estudiar la ejecución de eventos discreta como una evolución lógica del bucle principal. La importancia de dicho avance tecnológico es muy grande, ya no solo en optimización y rendimientos puros, sino en las posibilidades que ofrece en su consecuencia directa: la ejecución asíncrona. Aquí es donde entra el RT-Desk cuya característica principal, además de la explicada ejecución de eventos discreta, es que asegura la simulación en tiempo real.

Objetivos

Así pues, el siguiente proyecto final de carrera actual tendrá 3 objetivos bien diferenciados:

1. Desarrollar un engine (motor, kernel) para el desarrollo ágil de videojuegos en 2D para Microsoft Windows. En C++ y haciendo uso de Box2D.
2. Una vez implementado dicho engine con la arquitectura clásica de bucle principal del juego, realizar los cambios pertinentes para sustituir dicho núcleo por el moderno RT-Desk del DSIC de la UPV. En este proceso no se modificará la arquitectura, y el código se mantendrá lo más intacto posible. De esta forma podrán realizarse pruebas comparativas en una escala equiparable.
3. Por último, y una vez disponiendo de ambas versiones del engine, construir un videojuego como banco de pruebas con el cual se puedan obtener datos y estadísticas fiables y representativas. Finalmente obtener, arreglar y analizar dichos datos y estadísticas para comparar ambos métodos de ejecución, y encontrar los ventajas e inconvenientes de ambos más allá de lo que es el desarrollo.

Propuesta de trabajo

Estudio y análisis del proyecto

Los tres objetivos de este proyecto establecen una separación perfectamente delimitada del proyecto en tres partes.

El primero conlleva construir un engine desde la base. No se dará por terminado hasta que se quede satisfecho con su robustez, completitud y accesibilidad. Esto es así porque una vez terminado el primer objetivo, el código del engine ya no deberá cambiarse para añadir nuevas funcionalidades, sino que solamente se podrá alterar para amoldarse a las necesidades del segundo objetivo. Para poder estudiar dicho estado de satisfacción se indicará a continuación la lista de características y cualidades requeridas.

El segundo consistirá en primera instancia en aprender a utilizar RT-Desk y todas sus funcionalidades. Una vez bajo control, se harán todos los cambios pertinentes en el engine para integrarlo. Esta integración deberá asegurar que se mantiene una fidelidad lo más absoluta posible con el engine original. De hecho, en todo momento habrá libertad de elegir de qué modo se hará la ejecución, y dentro de lo posible se utilizará el mismo código para ambos.

Finalmente, la realización del banco de pruebas, las pruebas y el análisis de los resultado se hará siguiendo el principio de mínima intromisión. Se diseñará un juego que haga uso lo más realista posible (aunque no sea un juego como tal), y un set de tres mapas con el que podamos permitirnos comprobar el impacto del modo discreto. Para

las pruebas se programará un profiler que, junto al timer exacto del juego, nos de la libertad de tomar todos los datos necesarios con una intromisión mínima. Los datos obtenidos se volcarán en ficheros de tablas para su posterior análisis.

Esquema de la obra

Engine

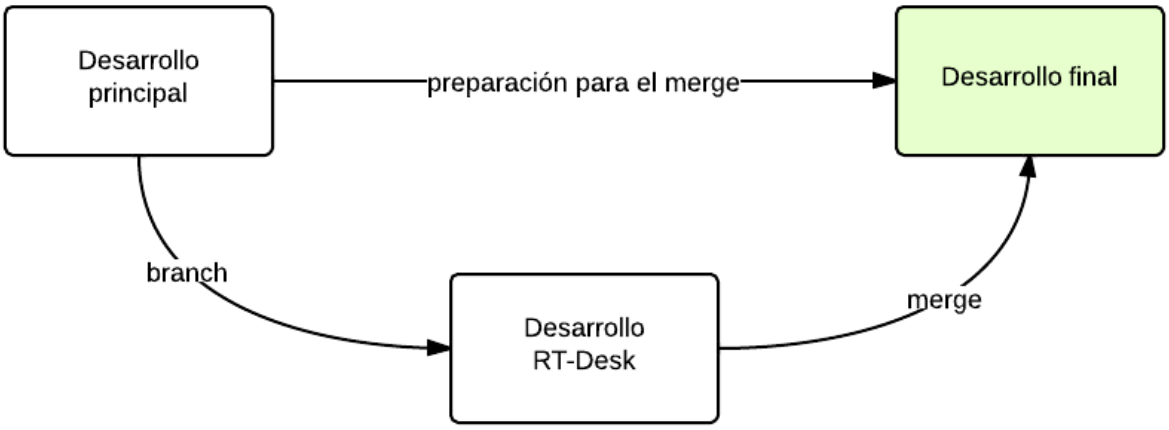
Debido a la envergadura del proyecto, se ha decidido seguir un método de prototipado para el engine.

1. En el **primer prototipo** se será laxo tanto con la arquitectura como con la corrección del código. Con él se decidirá por una arquitectura.
2. En el **segundo prototipo** se diseñará una arquitectura clara, permitiéndose libertad en la adecuación del código mientras se lleve a cabo la arquitectura.
3. Finalmente, y con el segundo prototipo funcional, se realizará una limpieza, corrección y documentación general de código. Ésto se convertirá en el **desarrollo principal** del engine, el cual se utilizará para la segunda fase.

RT-Desk

La segunda parte del proyecto deberá realizarse teniendo en cuenta que las funcionalidades del engine han de mantenerse intactas.

1. El primer paso consistirá en el **aprendizaje del funcionamiento y manejo correcto de RT-Desk**.
2. El segundo la realización de un **desarrollo paralelo** al desarrollo principal donde se modificarán todos los módulos necesarios para el modo discreto con RT-Desk.
3. El tercer paso consistirá en **preparar el desarrollo principal para el merge** con este nuevo desarrollo.
4. Finalmente, **unir ambos desarrollos** en el mismo código y proyecto, y permitir al usuario por medio de macros configurar el proyecto a su deseo.



Banco de pruebas y pruebas

La tercera parte procederá linealmente, como la primera.

1. Primero, el diseño del videojuego banco de pruebas.
2. Segundo, su realización sobre el desarrollo final.
3. Tercero, el desarrollo de las herramientas (principalmente el Profiler) necesarias para realizar las pruebas.
4. A continuación, realizar las pruebas, procesar y analizar los datos y gráficas obtenidos.
5. Finalmente, obtener y redactar las conclusiones.

Diseño y especificación

Tecnología para el desarrollo

El proyecto se va a realizar en el entorno de trabajo Microsoft Visual Studio 2012, usando el compilador por defecto de dicho entorno. El lenguaje elegido es C++. En la sección de "Estado del arte" se comentó el porqué de la elección: Este conjunto de tecnologías fue, es y será el estándar del gremio de desarrollo de videojuegos.

Entrando un poco más en detalle, se ha decidido abandonar el ubicuo patrón de diseño conocido como Singleton en pos del uso de los espacios de nombres (namespaces) del estándar del lenguaje. Por lo demás, y a rasgos superficiales, el uso de C++ será el usado comúnmente en el desarrollo de videojuegos. En el apartado correspondiente se comentará más en detalle el por qué.

También requiere mención la elección de RT-Desk. Es un kernel de simulación discreta en tiempo real desarrollado en la Universidad Politécnica de Valencia (donde también

se desarrolla este proyecto final de carrera). A la altura de desarrollo a la que se encuentra en el momento de realizar este proyecto, dicho kernel está en un estado lo suficientemente avanzado como para que sea estable y no vaya a forzarse la realización de grandes cambios a mitad del desarrollo. Podrá utilizarse de una forma real, funcionando así en una sinergía en que ambos proyectos se benefician uno del otro.

Bibliotecas de terceros

Un engine para videojuegos es una obra muy grande, densa y complicada. Ciertos módulos, más específicamente la parte que trabaja directamente con el sistema operativo en el input y output, el parser de información y el módulo de físicas, se salen fuera del contexto de este proyecto final de carrera. La realización de cada uno de ellos alargaría el proyecto de una forma excesiva y difícil de predecir, además de que no aportaría nada de valor a nuestros objetivos, por lo que se decidió apoyarse en librerías de terceros.

Allegro es una biblioteca multifunción y multisistema que lleva en desarrollo desde principio de los años 90, y ofrece un amplio abanico de herramientas. En el presente proyecto se hará uso de la entrada de input (teclado y ratón), y del manejo de un display (una ventana de Windows, con su correspondiente método de dibujar una imagen en ella). Aún así, se requerirá un posterior de trabajo para poder dar un uso correcto y apropiado a dichas herramientas, que por sí solas no cumplen una función definida.

Box2D es un engine de simulación de físicas bidimensionales, libre y de código abierto, realizado por Erin Catto. Es robusto, eficiente, multisistema y, por esas mismas razones, su uso está muy extendido en la comunidad de juegos 2D que requieran de algún tipo de cálculo de leyes físicas. Ofrece herramientas muy avanzadas, como cuerdas o articulaciones, que aunque queden fuera del presente proyecto, permitiré que se pueda extender para soportarlas en futuros proyectos basados en este.

Boost::Serialize es un módulo de la biblioteca Boost para realizar serialización. No estará en el engine final, pero realizará funciones importantes durante los desarrollos paralelos de bancos de pruebas. La razón de haber escogido parseo ante serialización es lo enormemente intrusiva que es este segundo método.

RapidJSON es la biblioteca de parsing a JSON elegida. Es muy eficiente y rápida, sencilla de usar y muy poco intrusiva.

Metodología

Como repositorio principal se usará el SVN privado otorgado por el DSIC para la ocasión. Como enseña la experiencia, también se dispondrá de un segundo repositorio. Este segundo será usando Hg (mercurial) y será privado. Ambos repositorios tendrán

exactamente los mismos commits. Se seguirán los principios del sentido común a la hora de utilizar los commits:

- Un commit contendrá un cambio atómico. Esto no quiere decir que no modifique más de un módulo, sino que todos los cambios incluidos estarán orientados a solventar, modificar o añadir una sola funcionalidad.
- Un commit siempre conducirá el repositorio de un estado consistente a otro igualmente consistente.
- En el caso de que no sea así y el commit contenga un cambio incompleto o inconsistente, se indicará con la etiqueta WIP (work in progress). A veces es necesario.
- En el caso de que un solo commit realice dos o más cambios atómicos, estos estarán lo suficientemente separados como para que no haya conflicto. De esta forma al revertirse a dicho commit, será fácil reutilizar el código de la segunda funcionalidad de un commit más moderno.

Para la gestión de tiempos y objetivos del proyecto se usará un método de milestones. Al ser un desarrollo individual de un proyecto bastante extenso, se ha de tener claro en todo momento los cambios que se requieren: Ya sean añadir nuevas funcionalidades o modificar o arreglar funcionalidades ya implementadas. Una ventaja del desarrollo individual junto al sistema de milestones es que, al tener perfectamente claro en todo momento qué tareas faltan para completar el desarrollo, se tiene libertad para escoger la siguiente. Aún así, y gracias a las milestones, nunca se entrará en un estado de estancamiento o en el que se han olvidado funcionalidades necesarias.

También cabe remarcar que me se definieron unas directrices para el código (coding guidelines) que se han de seguir en todo momento. Éstas están basadas en guidelines profesionales claramente justificadas, y todo el código del proyecto se deberá ceñir a ellas para mantener la cohesión y coherencia de todo el código

Arquitectura

El segundo prototipo, que se realizará en la primera parte del proyecto, tendrá como objetivo fijar la arquitectura que se proyectará al terminar el primer prototipo. En ella se tratará de cumplir lo máximo posible los principios del diseño de software que aseguran un buen software.

Aún así, surgirá algún que otro problema puesto que la arquitectura estará diseñada con la ejecución de modo continuo en mente, y luego se tendrá que adaptar de forma rígida al modo discreto: Que tiene unos requisitos, procedimientos y herramientas distintas. Esto hará que la versión discreta no siga todos los principios que debería, pero no hay solución a esto: Cambiar la arquitectura para la versión discreta invalidaría totalmente los objetivos del presente proyecto. Algo así se deja sobre la mesa para un futuro proyecto.

Características del engine

Las características más relevantes de la arquitectura, y en las que se podrá basar para saber en qué estado y con qué correctitud se encuentra el proyecto, son las siguientes:

- El proyecto se separa en 2 proyectos perfectamente diferenciados: Engine y Game. Engine incluirá todo aquello que no dependa directamente del proyecto en cuestión. Game, a su vez, incluirá todo lo que se presta a ser modificado de un juego a otro.
- Todo el cálculo de las físicas deberá relegarse a Box2D. Se hará uso de sus métodos y funciones tal y como debe ser hecho, de forma que no puedan introducirse fallos o generar problemas.
- El envoltorio (wrapper de ahora en adelante) para el render de Allegro ha de suplir los posibles usos que de él se puedan dar en Game. Esto incluirán planos de scroll (parallax), tener en cuenta o no la ventana y animaciones basadas en sprites. Así pues, se proporcionará al usuario del engine una serie de métodos claros, concisos y directos con los que trabajar.
- El wrapper para el input de Allegro permitirá el uso del teclado y del ratón.
- Habrá un timer gestionando la ejecución del bucle principal en modo continuo.
- Se creará un gestor de escenas (SceneManager) para gestionar los distintos estados del juego: Menús, distintos estados de juego o mensajes al usuario, etcétera. El bucle principal del juego en el modo continuo simplemente solicitará una actualización de dicho SceneManager cada vez que el timer lo exija.
- A su vez, se proporcionará una escena base vacía (Scene) preparada para que el usuario herede y utilice para sus propias finalidades.
- Se creará un gestor de entidades (EntityManager) para gestionar todos objetos físicos en el juego.
- A su vez, se proporcionará una entidad base vacía (Entity) preparada para que el usuario herede y utilice para sus propias finalidades.
- Habrá un objeto ventana (Window) que permitirá centrar el display en un punto del juego (habitualmente el jugador principal), permitiendo así que el mundo sea de un tamaño decidido por el usuario, y centrar la acción en un sitio específico de él.

Proceso

Primer prototipo

Desarrollo

El desarrollo del proyecto se inicia en Junio. Lo primero a realizar una vez terminado el diseño del proyecto es un primer prototipo con el cual poder modificar dicho diseño para acercarlo más a las futuras necesidades reales, así como para poder aproximar unos objetivos y necesidades mucho más realistas.

Las propias características del primer objetivo impiden que el desarrollo se atasque o que aparezcan problemas de complicada resolución en su desarrollo. Dichas características se resumen en la libertad para hacer todo lo necesario para alcanzar el objetivo final: usar métodos no convencionales o poco apropiados, programar módulos sin suficiente robustez o propensos a fallos, etcétera. Puede resultar poco adulador, pero como ya se detalló previamente, las finalidades ahora son: Obtener un ejemplo significativo del producto final deseado y poder estudiar las necesidades reales del proyecto una vez se tenga que desarrollar de forma correcta.

Por mucho que se trate de prever los requisitos, eso no deja de ser un ejercicio mental e imaginativo hasta que no el proyecto no se lleva a cabo.

Tras una semana de trabajo se obtiene lo que sería el juego banco de pruebas realizado sobre el engine. Aunque se reutilizó algo de código de proyectos pasados, casi todo se hizo partiendo de cero.

El resultado final es satisfactorio. El juego se controla correctamente, hace uso de Box2D de una forma apropiada teniendo en cuenta el objetivo final de ser usado como benchmark entre modo de ejecución discreto y modo continuo. Esto quiere decir que se puede comenzar a diseñar la arquitectura, a preparar los distintos milestones y a tomar decisiones preliminares con tal de que se puedan cumplir todas las características del engine.

Singleton, Namespaces y clases

Es en este momento cuando, en mi estudio personal del lenguaje C++ orientado a este tipo de proyectos, encuentro que el patrón de diseño Singleton ya no sólo no está recomendado, sino que se recomienda evitarlo de forma activa. Las alternativas más correctas son el uso de una clase tal y como dicta el estándar de C++, o el del Namespace de una forma que, aunque el estándar de C++ no define explícitamente, es totalmente correcta con él.

La posibilidad de usar una clase, aunque muy defendida por un gran número de profesionales y expertos, no parece ser una solución apropiada.

Las principales razones para utilizar el patrón Singleton eran:

- Asegurar y forzar que solo haya una instancia del objeto.
- Que dicha instancia ofrezca un acceso global.

Ambas características se pueden ofrecer con una clase básica. Aún así, la primera de las propiedades fuerza al uso de parches complicados y dispersos a lo largo y ancho del código. La segunda, a su vez resulta aún menos apropiada para este proyecto. A continuación un ejemplo:
El singleton Input, que se encargará de ofrecer al código el estado de la entrada de teclado y ratón, debe tener acceso a muchos módulos muy separados de la arquitectura.

Los dos modos de funcionamiento del input se diferencian por el comienzo de la cadena de comando:

1. Cuando el módulo de input recibe un evento de teclado pulsada, lo envía al gestor correspondiente (SceneManager o EntityManager). Este gestor correspondiente debe, a su vez, propagar el evento hacia abajo a lo largo de su árbol hasta que llegue al miembro correspondiente. La cadena de comando comienza por el input, y tenemos una serie de datos recorriendo toda la arquitectura.
2. Cuando el input recibe un evento de tecla pulsada, simplemente actualiza una estructura de datos interna. Cuando un miembro de un gestor (una Scene o un Entity) requiere la comprobación del input, le pide al módulo de input que le diga el estado de dicho input. La cadena de comando comienza por el miembro, y de forma directa realiza la comprobación: Ningún dato pasa por módulos a los que no va dirigido.

En ambos casos se tendrían que incumplir principios del diseño de software. En consecuencia, se decidió estudiar la segunda opción: utilizar namespaces.

Una forma de definir un espacio de nombres (namespace de ahora en adelante) de C++ de una forma aproximada y sencilla es considerarlo como un contenedor abstracto de un ámbito (scope de ahora en adelante) de variables. Como sucede con esta clase de elementos abstractos y complejos, esta definición no es exacta. Aún así, y para el caso, servirá. Con un namespace se define un scope. Por la forma en que el estándar de C++ implementa los namespaces, ambas características del singleton antes mencionadas se ven suplidas.

Por otra parte, los namespaces ofrecen las mismas herramientas que una clase, aunque no sean demasiado conocidas. Se pueden definir variables privadas en un namespace (si se definen en el fichero .cpp pero no en la cabecera). También existen otras herramientas como los namespaces anónimos. La principal desventaja es que los namespaces, al no ser clases, no permiten polimorfismo. Apropiadamente, esto no entra en conflicto con nada de lo que se vaya a hacer uso en un proyecto como el actual.

Además, el namespace no posee las desventajas que, en primera instancia, conducen a evitar el patrón Singleton. Para una lectura más detallada sobre el tema, ver bibliografía.

Como dato curioso, se puede localizar el momento de este descubrimiento entre el 25 de Junio y el 3 de Julio, días en los que el desarrollo se detuvo bruscamente para el tema fuese detenidamente estudiado, y realizar los cambios pertinentes.

Segundo prototipo

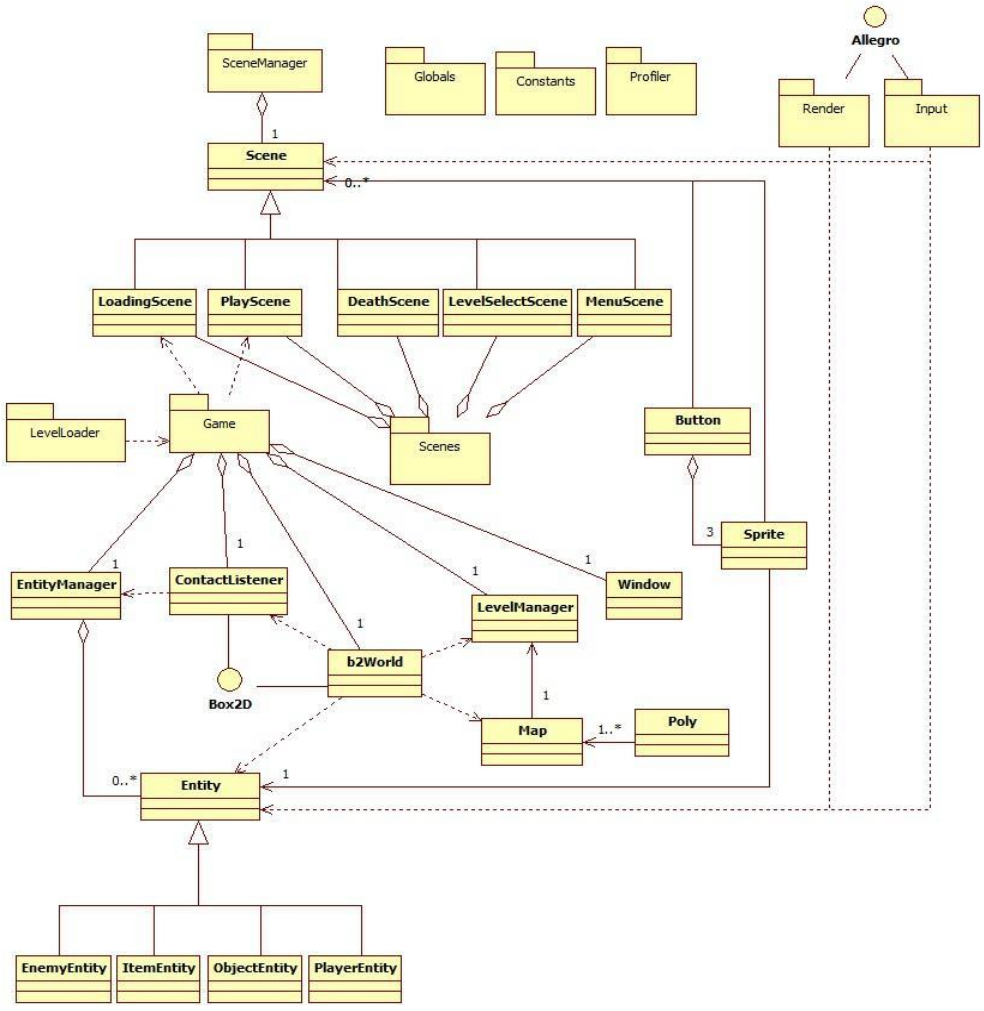
Arquitectura

Teniéndose en cuenta las características objetivo del Engine, lo aprendido en el desarrollo del primer prototipo y, además, los dos grandes cambios que se explicarán en las subsiguientes subsecciones de esta sección (SceneManager y Game), se llevó a cabo la arquitectura final.

Aquí no se incluyen ni los primeros bocetos de la arquitectura ni la primera versión (previa al SceneManager y Game). En el caso de estar interesado en ellos, a pesar de que no solo se encuentran obsoletos sino que además son menos correctos, estos se pueden encontrar tanto en el repositorio como en las carpetas de "deprecated" del código fuente.

A su vez, la arquitectura final del modo discreto se presentará y discutirá en su sección correspondiente.

A continuación, la arquitectura final del modo continuo:



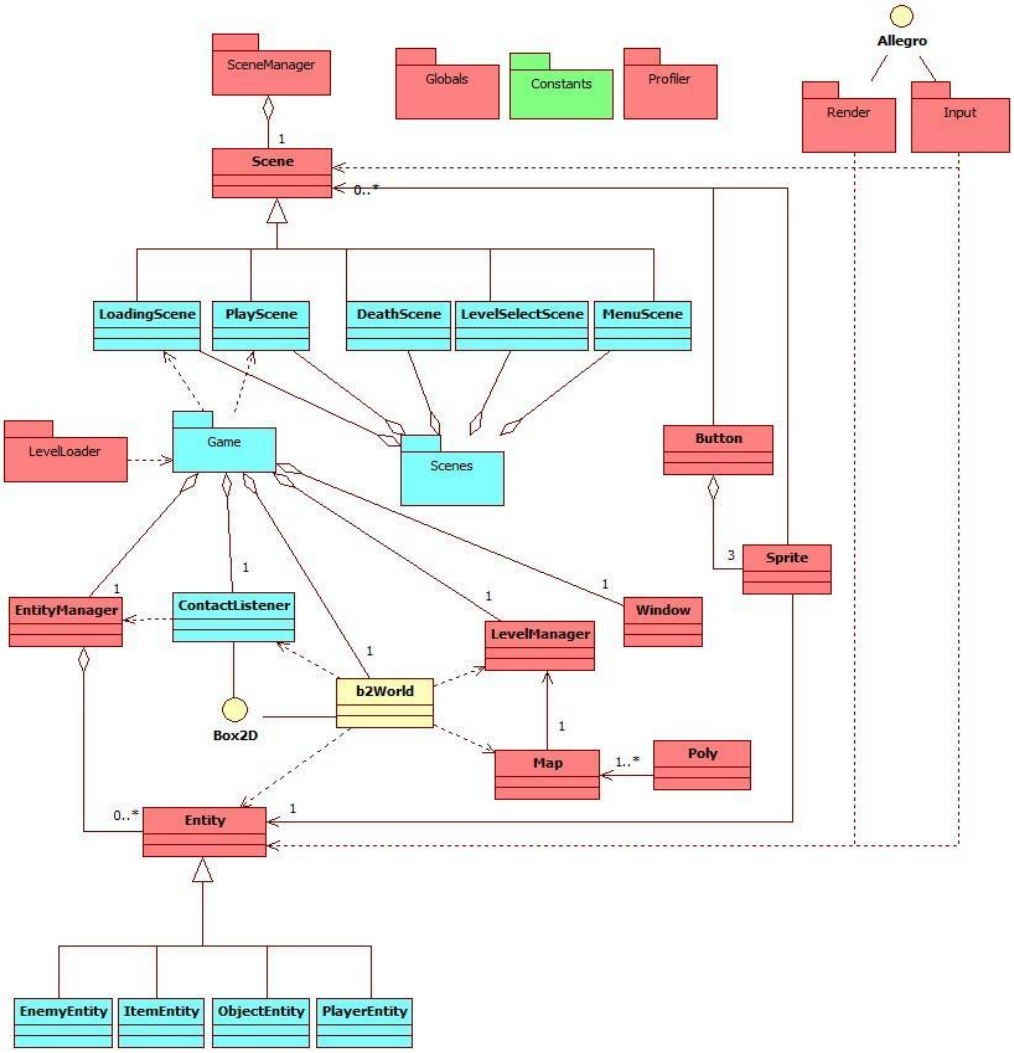
Los puntos más importantes son los siguientes:

- El namespace SceneManager contiene una escena de la clase abstracta Scene.
- De dicha clase Scene heredan las distintas escenas que contiene el juego. Todas ellas estarán alojadas en el namespace Scenes. Las distintas escenas podrán

contener widgets de tipo Button y Sprite.
- Dos escenas, LoadingScene y PlayScene, trabajarán con el namespace Game. El resto no.
- Game trabajará con el namespace LevelLoader para cargar los mapas durante la carga inicial.
- Con la información obtenida del LevelLoader, se inicializarán EntityManager y LevelManager, y junto a ellos el b2World, clase principal para las simulaciones de Box2D.
- El ContactListener es una clase predefinida que incluirá los callbacks para la resolución de colisiones e intersecciones. La clase abstracta viene dada por Box2D, y se ha de heredar para poder definir los comportamientos deseados.
- LevelManager contiene, a parte de los datos del nivel, una instancia de la clase Map, que se conformará de uno o más Poly. Esto es el mapa en sí.
- Por otra parte, el EntityManager contendrá entre 0 y un número máximo predeterminado de clases abstractas Entity.
- De dicha clase Entity heredan las distintas entidades que contiene el juego. Todas ellas estarán alojadas bajo demanda en el EntityManager. Las distintas entidades deberán poseer un Sprite.
- La clase Window define el rectángulo dentro del b2World que se dibujará en la ventana display creada.
- Los namespaces Render e Input, que como ya se ha explicado son wrappers de Allegro, se usarán desde las clases abstractas Scene y Entity.

Una vez decidida la arquitectura, se tuvo que decidir la separación entre Engine y Juego. La definición de dicha diferencia es sencilla: Engine incluye todo lo que no está directamente relacionado con el Juego, y por lo tanto se puede abstraer y reutilizar en distintos Juegos diferentes. Por otra parte, Juego incluye las cosas específicas de cada juego. Aún así, habrá ciertos módulos dentro de Juego que deberán estar en todos los juegos, siendo así comunes y cuyas definiciones serán reutilizables. Estos son los módulos incluidos en esta arquitectura.

A continuación se vuelve a mostrar la arquitectura, pero esta vez con una leyenda de color. Los objetos y namespaces rojos son los situados en el Engine, los azules son los del Juego y el verde es porque es necesario en ambos sitios. El b2World está definido en Box2D.



Métodos comunes

La mayoría de los objetos del proyecto tendrá una serie de métodos comunes. Aunque la definición de estos sea específica para cada objeto, su funcionalidad es la misma.

- **Setup/Create, Shutdown/Destroy:** Crea e inicializa el módulo, y borra y destruye el módulo. Los módulos usan la nomenclatura Setup y Shutdown, y los objetos caducos Create y Destroy. Aunque a efectos prácticos sean idénticos, se hace uso de dicha nomenclatura por el simple hecho de que un módulo no se crea y destruye, sino que se prepara y apaga, mientras que un objeto caduco sí que se crea y destruye.
- **HandleEvents:** Recibe un evento, principalmente de Input aunque puede ser usado para inteligencia artificial, y o bien lo procesa o lo propaga a los nodos inferiores donde será procesado. Si, por ejemplo, se pulsa la tecla J, se crea un evento que se enviará a uno de los dos namespaces, SceneManager o Game, según esté definido en el módulo UserInput. Dentro del HandleEvents de uno

de los dos namespaces, se propagará hacia la escena o la entidad que lo requiera a través de los respectivos HandleEvents.
- **Update:** Simplemente realiza toda la lógica necesaria dentro del módulo u objeto correspondiente. En el objeto Sprite, por ejemplo, consistirá en actualizar la animación. En el objeto Button será actualizar su estado, en el caso de que haya sido pulsado o se encuentre el ratón encima de él. Una entidad solamente se encargará de decirle a su Sprite que se actualice.
- **Draw:** Realizar las llamadas correspondientes a Render, ya sean para los Sprites propios o para los Sprites de los nodos inferiores.

Habrá más métodos comunes y variará el uso de los explicados en el modo discreto pero, de nuevo, se detallarán en su sección correspondiente.

Desarrollo

El segundo prototipo se desarrolló, mayormente, entre el 20 de Julio y el 27 de Agosto.

En la lista de commits (Apéndice E) se puede comprobar que el desarrollo fue fluido y constante excepto en dos puntos bien marcados. Ambos se corresponden con cuestiones de gran importancia que no había podido tener en cuenta en el primer prototipo. Debido a ello, se encontraron comportamientos inapropiados y soluciones incorrectas mientras se estaban desarrollando, o incluso una vez ya desarrolladas y siendo usadas. Esta situación demuestra que incluso con el proceso de desarrollo de prototipos hay situaciones que no se pueden prever.

A continuación se van a analizar estas dos grandes dicotomías, puesto que el resto del desarrollo fue según lo previsto en la arquitectura. El primero de ellos es el namespace SceneManager (del 07/23 al 08/01), y el segundo la creación e integración del Game (del 08/04 al 08/13) separado del Playscene. En conjunto, ambos condujeron a una arquitectura distinta a la diseñada originalmente, y que se ha visto en la subsección previa.

SceneManager y StateManager

Esta modificación consiste en una serie de modificaciones más pequeñas que redundan en un comportamiento mucho más estable y correcto.

Primero se ha de conocer cómo se aborda la situación previo a las modificaciones. Con mucha propiedad, antes llamaba a las escenas States (“estado” en inglés) en vez de Scenes (“escenas” en inglés), y se aprovechará dicha nomenclatura para diferenciar el estado previo y el moderno en el presente texto.

Los distintos States se creaban e inicializaban dinámicamente dentro del StateManager (“gestor de estados”) bajo demanda. En el momento en que debía haber un cambio, se

borraba el State actual, se liberaba su memoria, y se creaba e inicializaba otro, de forma dinámica, usando el mismo puntero. Las desventajas son:

- Se ha de indicar al StateManager el cambio en el momento preciso de la ejecución del bucle principal. En el caso de hacer un cambio a mitad de iteración del bucle, podría darse una situación inconsistente. Dicha indicación puede llegar a complicarse mucho según la lógica interna del juego se complica. Por ejemplo, el hecho de que el jugador principal deba morir antes de que se realice el dibujado en pantalla (algo inevitable) y que, en consecuencia, al morir el jugador se deba cambiar de State a mitad de una iteración y antes del dibujado en pantalla.
- Independiente de la ejecución continua y secuencial de la lógica del juego y los States se encuentra la entrada física (Input como teclado o ratón) o software (por ejemplo cerrar la ventana desde el sistema operativo Windows) que en ambos casos son asíncronas. No tienen en cuenta el bucle principal y llegan en cualquier momento. Hasta cierto punto se pueden sincronizar con la iteración del bucle principal, pero en última instancia nunca se podrá saber cuándo tendrán un efecto directo dentro de la ejecución. De nuevo se producen estados inconsistentes.
- El State que contiene la lógica del juego, el PlayState, contiene un gran número de estructuras de datos con las entidades y mapa y otras cosas de la lógica del juego. Están distribuidos en la memoria según se gestione de forma interna (una posible ampliación del Engine sería un módulo de gestión de memoria). Si se interrumpe su procesado, pueden quedarse bloques de memoria mal distribuidos, inestables,perdidos, y/o se puede acceder a zonas de memoria incorrectas.

Todo ello se solucionaría siguiendo dos principios a la hora de definir el concepto de escena. Estos son:

1. Las escenas se crearán e inicializarán al principio total del juego, y no se destruirán ni borrarán hasta el final de éste. Así pues, lo que podrá hacerse es reinicializar el estado actual de la escena a su estado virgen nada más ser inicializada. Además, se alojarán en un scope (“ámbito” de C++) que permita tenerlas controladas a la vez que permita su uso desde todos los lugares necesarios.
2. El cambio de escena no se efectuará bajo demanda en ninguna situación. El único módulo que sabe cuándo puede realizarse el cambio de forma estable y correcta es el gestor de escenas, y por lo tanto deberá ser él el que, de forma interna, lleve a cabo el cambio. En consecuencia, lo que los distintos módulos harán será **solicitar** (“request”) un cambio de escena al gestor, y seguir con la ejecución normal hasta que el gestor realice el cambio de forma totalmente transparente para todo el resto de la ejecución.

De esta forma se asegura que los cambios de escena no dejen trozos de memoria mal distribuidos, inestables o deslocalizados, puesto que las escenas nunca cambiarán y siempre serán las mismas. Además, se asegura que jamás se entrará en un estado inconsistente, pues el cambio de una escena (ahora llamadas Scene) a otra lo realizará el gestor de escenas (ahora llamado SceneManager) en el momento apropiado y

correcto de la ejecución, y no cuando, de forma asíncrona y totalmente incontrolable, se produzca el evento que lo exija.

Finalmente, se decidió alojar las distintas escenas en un namespace nuevo llamado Scenes (atención a la "s" del final del nombre), otorgándole así un scope propio dentro del ámbito global de Zentract. Por otra parte, su inicialización se hará en el propio Main del juego, definiéndose para ello como external dentro de Scenes aunque su declaración se deje dentro del fichero cpp para mantener el scope. Por otra parte

Game y PlayScene

En el primer prototipo todos los elementos relacionados con el juego en sí (el juego jugable, "Juego" a partir de ahora) se situaban y procesaban en el PlayScene (escena de juego). En ella estaba el EntityManager con las entidades, el MapManager con los polígonos del mapa, todo lo necesario para el cálculo de las físicas de Box2D, la ventana del juego, etcétera. Parece algo lógico cuando se conoce lo que representa el objeto PlayScene. Al fin y al cabo se estudia en las buenas prácticas que un objeto ha de ser auto-contenido y ha de desempeñar sólo una función, aunque en su completitud.

Al contrario de lo que pueda parecer, esto es un error muy grave. La función del objeto PlayScene es mantener y actualizar el Juego según el momento actual, según los timers y la interfaz gráfica. Por otra parte el Juego ignora todas estas cosas, y lo único que hace es mantener las estructuras de datos y los métodos para llevar a cabo la lógica interna del juego. En otras palabras, es totalmente indiferente a la ejecución, a los timers y a la interfaz gráfica.

El hecho de unir estos dos objetos totalmente distintos acarrea una serie de desventajas:

- Cargar los assets es un proceso de una duración indeterminada e impredecible: depende del sistema, del sistema operativo, de los contenidos requeridos y del mapa en cuestión. No se puede indicar de ninguna forma un tiempo fijo para ello. Para poder soportarlo correctamente en PlayScene había que acudir a soluciones poco ortodoxas y que hay que eludir a toda costa (las conocidas como parches, las cuales voy a nombrar más veces durante este documento aunque sea para estudiarlas como malas soluciones).
- Así mismo a la hora de cerrar el juego para, por ejemplo, volver a un menú, el tiempo para la descarga y borrado de los assets no puede determinarse a priori. Siendo cualquier aproximación heurística un parche y por lo tanto un fallo de base.
- Como ya se ha comentado, la propia definición de objeto (o módulo, o clase) implica que proporcionará o contendrá uno o un conjunto de herramientas y datos interrelacionados. PlayScene, los datos/herramientas del Juego y la lógica interna del Juego son conjuntos perfectamente diferenciados, por lo que contenerlos en un mismo módulo va en contra de los principios y buenas prácticas.
- Relacionado con la razón de ser de los principios mencionados en el anterior punto, al vincular dos módulos que, aunque relacionados, no deben estar

juntos, se pierde la posibilidad de hacer con ellos todo lo que debería poder hacerse. Si, por ejemplo, se quiere implementar algo que solo afecte al estado del juego, debemos hacerlo con un parche que haya que no afecte al PlayScene.
- Y, en resumen, forzaba estados inconsistentes e inestables ante los cuales la única solución eran los parches.

Uno solo de estos puntos habría sido suficiente, pero el desarrollo se vió detenido ante una pared insalvable que forzó a refactorizar y modificar la arquitectura para incluir un módulo Game. Así no solo se solucionó todos los problemas anteriormente mencionados, sino que añadió y posibilitó funcionalidades de una forma prácticamente directa.

Las funcionalidades más gráficas y evidentes fueron:
- Ahora pausar el juego en cualquier momento consiste simplemente en una comprobación de un booleano en PlayScene que, de ser verdadero (si está pausado el juego), no se actualiza al Juego.
- Al encontrarse PlayScene y Game separados, puedo mantener Game en cualquier sitio del código y, especialmente importante, manejarlo desde distintas escenas. Lo más importante fue añadir una LoadingScene (escena de cargado) donde el Game puede tomarse todo el tiempo que requiera para cargar los assets, y de una forma totalmente cubierta por el propio comportamiento de las escenas en el engine, pasar a PlayScene una vez los assets estén cargados. Sin parches, ni comportamientos anómalos, ni heurísticas que tratan de adivinar tiempos a priori ni posibilidad de entrar en estados inconsistentes o inestables: Correctamente integrado en el engine.
- Por la misma razón que el punto anterior, ahora puede irse a cualquier otra escena implementada en el engine (aunque no tenga ninguna relación con el módulo Game) y después volver a este y encontrarlo en el mismo estado. Esto puede usarse por ejemplo para un menú de opciones donde se puede cambiar el volumen de los sonidos. O los artistas pueden usarlo para ir cambiando gráficos o sonidos sin necesidad de reiniciar el juego cada vez.
- Colateralmente, y al ser un módulo cuya información es exactamente la información necesaria para el juego (ni más, ni menos), puede trabajarse con ella de forma atómica. Por ejemplo, por medio del patrón de serialización, se puede guardar el estado del juego en un fichero. En cualquier momento. Tras eso se puede cargar dicho fichero con serialización y volcarlo en el objeto Game para cargar la partida anterior y continuar.

En resumen, la mayor parte de ellas giran en torno al concepto de que las Escenas conocen y llevan la ejecución del juego, y el Juego lleva su lógica interna, y la forma en que ambos módulos pueden interactuar.

Serialización y parseo

Aunque no haya sido una dilema grande y problemático, es de importante mención el por qué se decidió cambiar la serialización por un parseo genérico de ficheros de scripting JSON.

En primera instancia, y por si no se conoce la serialización (“serialization” en inglés), ésta consiste en la traducción directa de datos a un formato almacenable. Esto viene a ser, a efectos prácticos, el traducir un área de memoria (definida por el propio área, o por una serie de variables, objetos y estructuras de datos) a un fichero de texto plano con un formato apropiado.

Características de la serialización:

- La traducción se lleva a cabo de forma directa cuando los datos proceden de estructuras sencillas, pero a medida que dichas estructuras aumentan en complejidad, la traducción se torna ilegible a simple vista. Dependiendo del contexto es, pues, un método poco escalable, y que ante cierto nivel de complejidad resulta inmodificable por un ser humano a simple vista. No es una característica inheremente mala, pero puede ser inapropiada.
- Requiere una penetración en el código muy alta. De nuevo, no es una característica inherentemente mala, pero se ha de tener en cuenta y en muchos casos se rechazará. En todas y cada una de las estructuras de datos que se vayan a serializar (incluso las anidadas dentro de otras) se deberá permitir acceso al método de serialización. Esto se llevará a cabo con una definición de amistad y un método sobrecargado, o bien directamente definiendo un método para que sea llamado por el de serialización.

Por otra parte, parsear consiste simplemente en leer un fichero de scripting teniendo en cuenta sus etiquetas. En consecuencia:

- Los ficheros podrán definirse con unas etiquetas ordenadas y nombradas de forma representativa. De esa forma cualquier persona, a simple vista, podrá ser capaz de modificar los valores deseados de la forma deseada.
- A su vez, la lectura y escritura de los ficheros será directa. Requerirá del conocimiento a nivel de código de las etiquetas y su jerarquía en los ficheros. Pero, en cualquier caso, eso es algo lógico y que se debe saber en cualquiera de los casos.
- No hay intrusión alguna. Se puede definir un método que vaya exportando las variables deseadas según las etiquetas nombradas en el anterior punto.

Cabe decir que la elección fue rápida. En el primer prototipo se utilizó la serialización de la biblioteca Boost, pero una vez conocidas y comparadas las características de ambos, se optó por parsear ficheros en el formato definido por JSON, que es el más reciente y utilizado actualmente. Para dicho parseo se decidió utilizar la biblioteca RapidJSON, cuyas principales características son el minimalismo, la simpleza y la velocidad.

Desarrollo principal

Desarrollo

La mayor parte del desarrollo principal consiste en la reutilización de código del segundo prototipo, siendo éste a su vez cuidadosamente arreglado. Hay una serie de principios y buenas prácticas que se deben suplir, y ahora es el momento apropiado para llevarlo a cabo puesto que la mayor parte de los módulos son totalmente funcionales y se encuentran en un estado final. Los principios y buenas prácticas que se repasan y corrigen son:

- Const correctness.
- Asegurar la atomicidad y autocontención de los objetos y módulos: Ya se ha solventado un gran fallo al respecto durante el segundo prototipo, pero ahora es el último momento donde un cambio de cualquier envergadura se puede permitir.
- Fijar los scopes. Es la primera vez que se lleva a cabo un uso tan cercano a clases de los espacios de nombres en C++, así que se encuentran pequeños fallos e inconsistencias repetidos un número considerable de veces. Algunos de ellos son, por ejemplo, la declaración de variables privadas en el header. En las clases basta con utilizar la palabra reservada “private” o “protected” para definir las variables como privadas, pero en los espacios de nombres se ha de declarar la variable fuera del fichero header. Al declararse solamente en el fichero cpp, la variable no será visible desde fuera. En otras situaciones, el uso de variables estáticas contra las externas era inconsistente.
- Comentar y documentar el código: Se adoptó Doxygen como método para generar la documentación, y conforme a él se comentan los ficheros .h del Engine.
- Seguir las directrices de código.
- Asegurar la buena gestión de la memoria. Comprobar que la memoria dinámica se cree y borre correctamente.

El primer gran cambio que se lleva a cabo fue separar, en dos Proyectos de Visual Studio distintos, el Engine del Game. Hasta ahora, y como se ha mostrado en la arquitectura con leyenda de colores del segundo prototipo, se ha diseñado todo con dicha separación. A pesar de ello, el desarrollo se ha llevado a cabo en un solo Proyecto. Se procede, pues, a realizar dicho cambio.

Fin de la primera parte

Como se vió en la sección “Esquema de la obra” donde se detallan los pasos del presente proyecto, se ha llegado al final de la primera parte.

El producto que se tiene entre manos consiste en el Engine que se pretende utilizar durante todo el resto del proyecto. De ahora en adelante sus funcionalidades no cambiarán, desde el punto de vista del usuario del engine. Lo que se cambiará serán partes del kernel (el core, el núcleo), pero eso ya se corresponde a la segunda parte del proyecto.

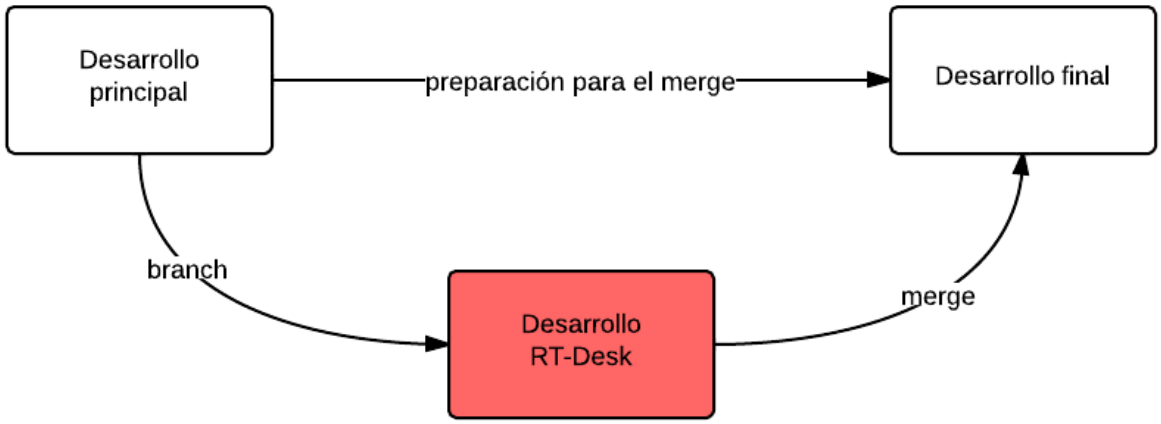
Este es el punto apropiado para que los potenciales interesados lleven a cabo su estudio del engine si su objetivo final es conocer el funcionamiento de un engine en C++ para videojuegos 2D con Box2D.

Desarrollo RT-DESK

Pruebas

Dejándose de lado y reservando el desarrollo principal, el único objetivo en este momento es el de conocer y aprender a usar el RT-DESK. Para afrontar dicha situación, se decide que la mejor solución es ir realizando distintas pruebas con creciente complejidad. La primera de ellas será un simple envío y recepción de mensaje, y la última de las cuales consistirá en el Desarrollo RT-DESK que, se recuerda, es la versión del Desarrollo Principal de la anterior parte con RT-DRT-DESK totalmente integrado.

A continuación se recuerda el esquema de la segunda parte para contextualizar el presente desarrollo:



Durante toda esta sección, y consiguientes, se asume que el lector conoce las características y el funcionamiento básicos de RT-DESK.

A continuación, se indica la lista de pruebas desarrolladas. Se llaman RTDtestX, donde X será el número correspondiente.

1. Inicialización básica y manejo básico: Envío y recibimiento de mensajes.
2. Integración de RT-DESK con los distintos módulos de Allegro que se van a usar: Input, Render y Timer. Uso básico de mensajes personalizados y de mensajes

privados.

3. Integración básica con módulos en clases y con módulos en espacios de nombre. A su vez, creación de Messager.
4. Integración avanzada y completa con módulos en clases y con módulos en espacios de nombre. Definición de los Messager específicos para Render e Input.
5. Con todo lo realizado hasta ahora se desarrolla un Pong.
6. En base al Pong del RTDtest5 se realizan los preparativos para llevar a cabo el Desarrollo RT-DESK. Se mantiene el Pong intacto, aunque a nivel de núcleo el código se modifica en gran medida.
7. El Desarrollo RT-DESK. Como ya se explicó, este consiste en el Desarrollo Principal pero utilizando RT-DESK en vez de el bucle principal y las llamadas secuenciales.

El Desarrollo RT-DESK parte de un branch del Desarrollo Principal, de forma que se comienza el trabajo con una versión funcional.

Nuevos métodos comunes

Además de los explicados en la sección anterior, se deberán añadir dos nuevos métodos comunes. El primero será Begin, y el segundo el que contenga el comportamiento de la recepción de mensajes.

- **Begin:** Para permitir la ejecución desincronizada y discreta del RT-DESK, los módulos que puedan trabajar de forma independiente se autollamarán por medio de mensajes propios. El primer mensaje deberá surgir de dentro, y con tal fin se define este método común.
- **HandleMessage:** Recibe un mensaje básico de RT-DESK, y contiene la lógica para procesarlo. Cada módulo y cada objeto tendrá su propia lógica. Este método se define en la clase Messager de la cual se heredarán todas las clases que lo necesiten. Para los módulos definidos con espacios de nombres se definirán instancias de Messager específicas.
- A su vez, **los métodos comunes previos deberán ser duplicados**: Uno para el modo continuo y uno para el modo discreto.

Desarrollo de la prueba 7

El primer paso que se realiza consiste en, antes de cambiarse el comportamiento de continuo a discreto, llevar a cabo una serie de ajustes ahora que el código está en su estado más completo y arreglado:

- Se define una directiva de preprocesador para centralizar en un solo lugar <Engine\Settings.h> la elección del modo de ejecución. Dicha directiva se llama SETTINGS_DISCRETE, y en base a ella se elegirá qué parte del código se elegirá según el modo deseado.
- Se duplican los métodos comunes y se incluyen en bloques de compilación condicional seleccionados por SETTINGS_DISCRETE, de forma que cuando

este exista solo se compile el código específico para la ejecución discreta. Viceversa con el continuo.
- El main se divide en dos partes, según el modo de ejecución, para poder realizar las inicializaciones específicas.

A continuación se definen los distintos tipos de mensajes personalizados. Se separan los mensajes de Engine y los mensajes de Game de tal forma que el usuario del Engine no pueda modificar los mensajes de Engine.

Mientras se realizan estos cambios, se van trasladando los módulos del Desarrollo Principal a la prueba 7 paulatinamente. Una vez se tiene el Engine completo, y están los ajustes para permitir la utilización de RT-DESK terminados, se procede a realizar el cambio. Dicho cambio se lleva a cabo desde el 11 al 15 de Noviembre, y consiste, a rasgos generales, en:

- Todas las llamadas internas de un módulo u objeto han de realizarse con SelfMsg.
- Todas las llamadas de un módulo u objeto a otro han de realizarse mandando mensajes.
- Se ha de definir el tratamiento que cada módulo u objeto va a dar a los mensajes recibidos en el método común HandleMessage.
- Se ha de asegurar la correcta herencia o uso del Messager.
- En todos estos casos se ha de decidir si el mensaje vale la pena que sea propietario o que se escoja dinámicamente del pool de mensajes cada vez.
- En todos estos casos se ha de decidir la latencia apropiada para el envío del mensaje, si va a haber una. Se hablará más detalladamente de este punto en una subsección posterior.
- En todo estos casos se ha de analizar si realmente se ha de mandar un mensaje o se ha de realizar la llamada directa. Se hablará más detalladamente de este punto en una subsección posterior.

Hasta ahora la decisión por defecto respecto a estos dos últimos puntos ha sido de “siempre que se pueda, mensaje”, y “si el mensaje no ha de ser instantáneo, una latencia predefinida compromiso”. Cabe añadir que esta parte del proyecto (sobre todo susodichos dos últimos puntos) causó una cantidad considerable de problemas durante el Desarrollo Final. Será entonces el momento apropiado para hablar en detalle de ello.

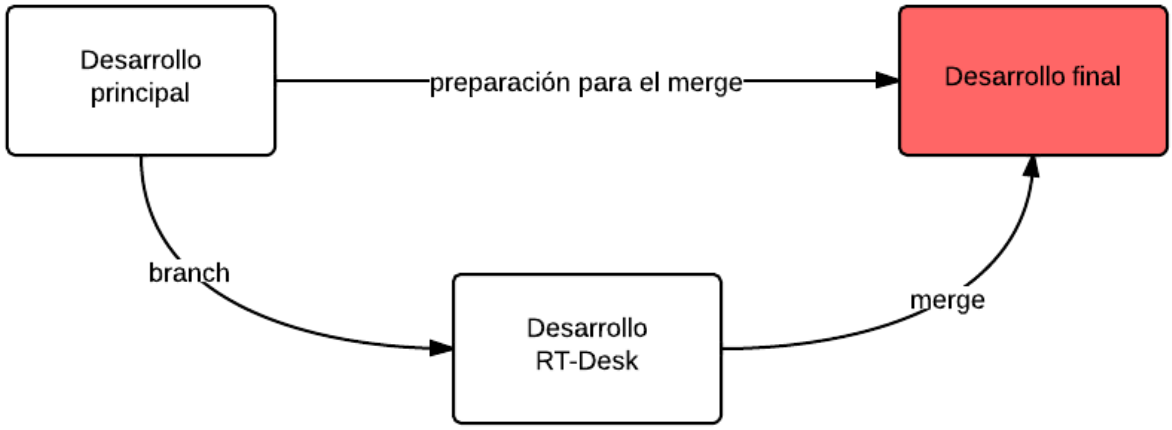
Se da por terminada la unión cuando la ejecución no causa ningún problema, es fluida y en todos los sentidos idéntica a la ejecución continua del Desarrollo Principal. Y, además, que posea todas sus características y estén totalmente integradas. Así es como se da por completo el Desarrollo RT-DESK.

Desarrollo final

Unión de desarrollo principal y la prueba número 7

Este último desarrollo de la segunda parte del proyecto se supuso sencilla, rápida y directa, y al final resultó ser la más costosa y lenta. Un rápido vistazo a la lista de commits, un poco más abajo, lo hará latente.

Se recuerda dónde nos encontramos:



El primer paso es crear el nuevo proyecto y adaptarlo a los requisitos. Una vez el Desarrollo RT-DESK (prueba 7) está en el mismo estado que en el proyecto de la sección previa, se procede a la integración del Desarrollo Principal. Esto toma del 27 de Noviembre al 2 de Diciembre, y después del 8 al 17 de Diciembre.

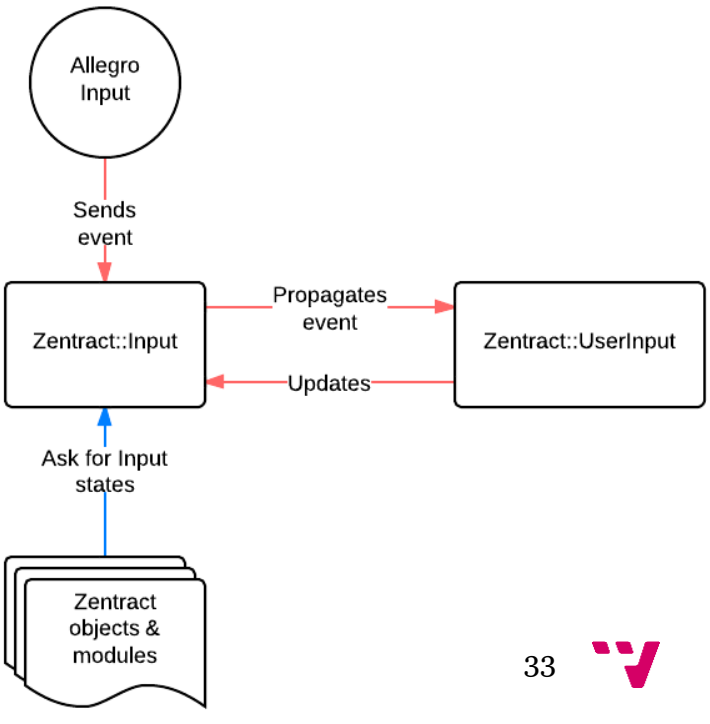
A continuación se van a analizar diversas cuestiones importantes.

Cadena de comando en la entrada

Ante un evento generado por la entrada física (un "Input", a partir de ahora) hay varias formas de proceder. En concreto se han seleccionado las dos más representativas y ventajosas. A continuación se adjuntan dos esquemas y sus explicaciones:

El **modo de acceso bajo demanda**, el primero que se implementó, antes de conocer las especificaciones y necesidades específicas del modo discreto asíncrono de RT-DESK.

- Zentract::Input solo posée una serie de estructuras de datos en las que se mantiene el estado de los distintos posibles Inputs (pulsado o no pulsado). Se recuerda que este módulo está ubicado en la parte de Engine, por lo que no conocerá nada



directamente dependiente del proyecto que se lleve a cabo con el Engine. Para modificar el estado de sus estructuras de datos hace uso de Zentract::UserInput.

- Zentract::UserInput contiene los métodos para procesar el Input y modificar las estructuras de datos. Se encuentra en la parte del Game puesto que el usuario del engine deberá modificarlo bajo necesidades del proyecto. Aquí es donde se encontrará la lista de posibles teclas del juego, de forma independiente al Input real, y se equipararán uno a otro.
- Cuando llega un evento, Zentract::Input lo propaga a UserInput, el cual le devolverá el evento procesado para que pueda aplicar el cambio correspondiente a la estructura de dato correcta.
- De forma independiente y asíncrona, cualquier módulo u objeto dentro de Zentract que tenga acceso al espacio de nombres Zentract::Input y Zentract::UserInput puede solicitar la información de las estructuras de datos de Zentract::Input por medio de una serie de métodos. Éstas peticiones se efectuarán cuando el objeto o módulo interesado se esté ejecutando, y por lo tanto se considera asíncrono y no secuencial, aunque siga estando, lógicamente, enmarcado en el bucle principal.

El **modo de cadena de comando estricta**, en forma de árbol, que se implementó para trabajar con mayor precisión y facilidad con RT-DESK y el modo de ejecución discreto asíncrono:

- El evento que recibe Zentract::Input se propaga tal cual a Zentract::UserInput.
- En Zentact::UserInput se encontrará el código para la bifurcación del envío o bien a Zentract::Game o bien a Zentract::SceneManager. De nuevo no se procesa el evento, sino que se propaga.
- A partir de Zentract::Game y Zentract::SceneManager, y a lo largo y durante sus objetos y módulos hijo, se sigue propagando el evento hasta encontrarse en el lugar correcto. Una vez allí se procesará.
- Todo el proceso se sucede individualmente para cada evento generado. En el modo continuo se producirá todo el recorrido de cada evento durante la actualización del Input dentro el bucle principal. En el modo de ejecución discreto la propagación se realizará asíncrona.

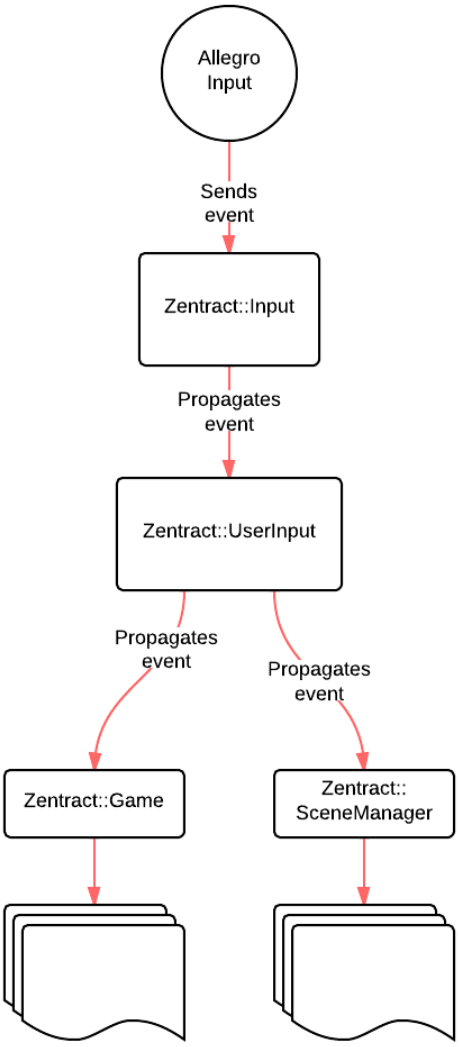


Tabla comparativa de características y consecuencias de ambos modos:

	Modo de acceso bajo demanda	**Modo de cadena de comando estricta**	**Solución más correcta**
Cadena de comando	La cadena de comando comienza en el objeto o módulo pertinente.	La cadena de comando empieza con la generación de un evento de Input.	Indiferente.
Obtención de Input	El objeto o módulo pertinente accede de forma directa al dato que requiere.	El evento de Input deberá propagarse de módulo en módulo a lo largo de todo el árbol que pende del SceneManager o del Game (los dos espacios de nombres jerárquicamente más altos de Engine y Game respectivamente). O lo que es lo mismo: el evento deberá recorrer toda la arquitectura	Bajo demanda. No es apropiado tener un dato recorriendo toda la arquitectura.
Alcance de ámbitos	Todos los objetos o módulos pertinentes deberán tener acceso al ámbito de Input y de UserInput, definidos como espacios de nombres con este fin.	La estructura de árbol jerárquica se mantiene intacta, siendo solamente Input el que deba estar vinculado a SceneManager y Game para comenzar la propagación.	Indiferente. Al estar el Input enmarcado dentro del ámbito del espacio de nombres definido para él, las razones por las que se han de evitar las variables globales no le afectan.
Tolerancia ante retrasos*	Al ser gestionado por una cola de eventos, tan solo el último evento de Input realizado sobre un Input físico durante un frame será tenido en cuenta. Los previos a él se sobreescribiran.	Todos y cada uno de los eventos de Input serán procesados individualmente, independientemente de retrasos.	Cadena de comando estricta. En el caso más excluyente, conviene tener en cuenta todos los eventos generados, independientemente de la fluidez de la ejecución.
Intrusión**	No ha de haber ningún código que no sea específicamente necesario para el procesado del evento de Input.	Todos los módulos y objetos intermedios han de tener código para la propagación, aún cuando no vayan a recibir ningún evento de Input para ellos. Toda la jerarquía debajo de SceneManager y Game	Bajo demanda. Es, en este sentido, la solución menos intrusiva posible para este problema. El otro método, en contraposición, es el más intrusivo puesto que requiere código

		deberá tener, pues, un método común HandleEvent.	sin utilidad primaria en todos los módulos y objetos.
Dispersión de código* **	Por la propia razón de ser del acceso bajo demanda, el código de acceso al Input se encontrará totalmente disperso en los módulos y objetos que lo requieran. No habrá control respecto a él, más allá del llevado a cabo por el usuario.		Cadena de comando estricta. El nivel de dispersión/centralización de código coincide con la dificultad de su refactorización y mantenimiento.
Claridad y facilidad de uso	Fuera de los módulos Input y UserInput, todo el código que hace falta es una simple llamada a uno de los tres métodos (está la tecla pulsada, ha sido pulsada o ha sido soltada). El uso del Input no puede ser más claro y sencillo.	En todos los módulos y objetos habrá un método común HandleEvent donde, por medio de instrucciones condicionales se deberá realizar su propagación o procesado.	Bajo demanda. No puede llegar a ser más sencillo y más claro.

* Se utiliza el término retraso como traducción de "lag". Este suceso consiste en que el tiempo de procesado de un frame es más alto que el tiempo medio de duración de un frame, haciendo así que el juego tenga un retraso.
** Se utiliza el término intrusión la cantidad de código no orientado al fin objetivo que se ha de colocar, aumentada por la localización de este en sitios donde dicho comportamiento no es esperado.
*** Refiriéndose en contraposición a la centralización de código.

De la tabla comparativa se puede inferir que no parece haber una solución superior. Podrían haberse incluido en la comparación otras soluciones más complejas y profesionales, pero se habría llegado a la misma conclusión: Se habrá de ponderar las ventajas e inconvenientes más apropiadas o menos apropiadas para cada proyecto individual, y adoptar una solución que ofrezca el mejor compromiso.

Llegada a esta conclusión se decidió usar el modo de acceso bajo demanda desde el comienzo del proyecto. Las principales razones fueron su claridad y facilidad de uso, así como su nula intrusión. Por otra parte, la desventaja de tolerancia ante retrasos no importaba debido a que el engine posee cálculo de físicas en tiempo real.

Un pequeño inciso para clarificar este último punto, puesto que puede resultar muy poco obvio. Ante un retraso de gran envergadura se pueden producir un número antinaturalmente alto de eventos de Input sobre el mismo Input.

Esto puede ser una desventaja en un juego donde no hay cálculo de físicas constante puesto que los movimientos serán atómicos e instantáneos. Véase:

- Un juego de puzzle basado en una matriz. Cada movimiento se corresponderá a un desplazamiento de una posición a otra dentro de la matriz, y por lo tanto 4 eventos de desplazamientos a la derecha se deberán corresponder, sí o sí, con 4 desplazamientos a la derecha.
- En un juego de acción con cálculo de físicas constantes, 4 pulsaciones hacia la derecha en un mismo frame no pueden equipararse a 4 impulsos lineales con la misma fuerza tras un solo frame, pues no producen un efecto realista. Se deberá realizar solo 1 impulso por frame, que es como el movimiento ha sido previsto. De cualquier otra forma se romperían las reglas en las cuales se basa el diseño del juego.

Volviendo al tema en cuestión, una vez llegados al punto de desarrollo actual, resulta haber una característica que no se tuvo en cuenta en la toma de decisión inicial. Dicha característica es exclusivamente importante para el modo de ejecución discreto del RT-DESK, y por ello se perdona su previa exclusión, puesto que no se conocía tal modo de ejecución. Ahora que ya se conoce de primera mano, ya se puede tener en cuenta.

En el modo discreto, ciertos módulos u objetos se ejecutarán de forma asíncrona. Si se ha utilizado la solución del acceso a Input bajo demanda, no se tiene absolutamente ningún control sobre cuándo leerá de Input cada uno de ellos. Esta situación es, en cualquier tipo de proyecto, totalmente inadmisible.

	Modo de acceso bajo demanda	**Modo de cadena de comando estricta**	**Solución más correcta**
Tolerancia a modo de ejecución discreto	Ninguna. Se carece de control sobre cuándo se realizarán las consultas, y se ha de carecer puesto que es la característica principal de ambas partes. La propia razón de ser de este modo hace nula la tolerancia, por lo que son totalmente incompatibles.	Completa. No quiere decir que sea la solución que ofrece una tolerancia mayor, pero sí que ofrece una tolerancia suficiente.	Cadena de comando estricta es la única solución, siendo imposible el modo de acceso bajo demanda.

En consecuencia, queda totalmente obsoleto el modo de acceso bajo demanda que se ha venido usando hasta ahora, y se procede a implementar el modo de cadena de comando estricta.

Cuellos de botella y limitaciones

La característica y objetivo principal del modo de ejecución discreto es el permitir, por medio del envío de mensajes, un grado de libertad apropiado a la ejecución discreta de los distintos procesos. Por consiguiente, y ante un escenario idóneo, se podría llegar a ejecutar cada módulo de forma totalmente independiente y asíncrona del resto de módulos y objetos, teniendo en cuenta únicamente sus propios requisitos.

En el presente proyecto, puesto que no coincide con un escenario idóneo sino con un escenario real, no se tiene dicha libertad. Hay dos cuellos de botella: la biblioteca Box2D y el módulo de Render. Y, en consecuencia de ellos, se imponen ciertas limitaciones a las entidades. A continuación se analizan los tres casos:

Box2D, al ser una biblioteca de terceros, nos fuerza su metodología. Sería posible poder realizar los cálculos de físicas de una forma menos compacta y ensamblada, aunque en última instancia siempre hiciese de cuello de botella puesto que requiere una comparación par a par de posiciones y velocidades tarde o temprano (lo que fuerza a que dichos pares estén sincronizados para una comparación realista). Aún así, todo lo que no es el cálculo de colisiones se puede realizar de forma asíncrona, permitiendo un grado de libertad muy grande en comparación con el actual. Dicha modificación se estudia en la sección “Trabajos futuros”, pues es perfecta para un posible futuro proyecto: integrar RT-DESK dentro de Box2D. Solamente con ese cambio, e implantando el resultado final en el presente proyecto, se podría eliminar el cuello de botella más grande.

Render. El módulo que dibuja en pantalla todo lo necesario debe ejecutarse ordenada y secuencialmente, tanto respecto al resto del proyecto como respecto al propio orden de renderizado.

- El orden de dibujado debe ser secuencial puesto que se trata de un juego 2d dibujado en buffer. Lo primero que se dibuje se verá tapado por las siguientes cosas que se dibujen, obligando así a seguir un orden secuencial.
- El proyecto debe encontrarse en un estado consistente en cada dibujado, puesto que el Render tan solo recorre el grafo de escena (el namespace Game) ciegamente. Si en él se encuentra una entidad residual en mitad de proceso de eliminarse y trata de dibujarla se entrará en un estado inconsistente y saltará una excepción.

Entidades. Como se ha visto en los dos cuellos de botellas, la lógica de las entidades que no pertenezca al input ni a las físicas se ve delimitada al intervalo que se encuentra entre Box2D y Render. Por suerte, aunque esto nos limite, nos sigue permitiendo una ejecución discreta y asíncrona de los mismos dentro de dicho intervalo. Aún así, es una diferencia latente con respecto al escenario idóneo.

Necesidad y latencia de los mensajes

El último paso una vez está todo desarrollado y funcionando correctamente en el modo discreto es ajustar las latencias de los mensajes de eventos. Ciertas latencias se parametrizarán en el fichero <Zentract\Engine\Constants.h> para su fácil manejo. Estas se escogerán debido a su importancia dentro de la ejecución. Para mayor información consultar el documento <Zentract\Docs\RTDESK Message Delays.txt>.

A las distintas permutaciones de este juego de latencias se les llamará "perfiles de latencias".

Lo primero será escoger uno o varios perfiles de latencia base compromiso que aproximen lo máximo posible la experiencia de juego al diseño del juego. Una vez encontrada o encontradas se podrán realizar variaciones sobre ella con la finalidad de estudiar su impacto. Todo esto se corresponde con la parte final del proyecto, y entonces se tratará con la profundidad requerida.

Documentación

Durante la realización del Desarrollo RT-DESK y Desarrollo Principal se mantuvo constante la costumbre de comentar el código, así como generar documentación al respecto.

Los ficheros de documentación se pueden encontrar en los distintos directorios de desarrollo, siempre dentro de una carpeta llamada "Docs". La única excepción es la documentación sobre el proyecto en sí. Esta se encuentra en la carpeta raíz de cada desarrollo, con el nombre de "Notas.txt".

En él se puede encontrar información sobre la estructura de carpetas, descripción de los contenidos y características del proyecto, notas importantes sobre su ejecución o sobre su uso, y detalles para la compilación y enlazado correcto con las bibliotecas.

Compilación y enlazado

Como detalle conclusivo, se indica a continuación la configuración general para los proyectos, según las distintas bibliotecas:

Respecto a Allegro5:

1. Allegro requiere la biblioteca de enlace dinámico (.dll) en tiempo de ejecución.
2. Se provee el material para poder compilar la biblioteca que se desée. Así pues, se puede compilar tanto para Debug como para Release.

- En conclusión, Allegro permite que el enlazado se realice en /MT y /MD, tanto en Release como Debug.

Respecto a RT-DESK:

1. La biblioteca de RT-DESK que se ha proveído es solo para Release.
2. RT-DESK no requiere de una biblioteca de enlace dinámico en tiempo de ejecución. Bastará con el enlazado con la biblioteca de enlace estático (.lib).

- En conclusión, RT-DESK limita al enlazado en modo "/MT" en modo Release. Aún así, permite su ejecución en modo Debug "/MTd", aunque con una serie de warnings y parches que no deberían permitirse.

Respecto a Box2D:

1. Box2D no requiere de una biblioteca de enlace dinámico en tiempo de ejecución. Bastará con el enlazado con la biblioteca de enlace estático (.lib).
2. Se provee el material para poder compilar la biblioteca que se desée. Así pues, se puede compilar tanto para Debug como para Release.

- En conclusión, Box2D limita al enlazado en modo "/MT", pero se puede ejecutar tanto en modo Debug como en Release.

Conclusión final:

- RT-DESK y Box2D nos limitan al modo de enlazado "/MT".
- RT-DESK nos limita a Release.

También, y para evitar problemas de incompatibilidad, las bibliotecas de terceros deberán ser incluidas en el siguiente orden:

1. RTDeskEngine.lib;
2. Box2D-MT.lib;
3. allegro-5.0.9-monolith-md.lib;

Videojuego tipo para las pruebas y profilers

Requisitos y diseño del videojuego

Se han de tener en cuenta los requisitos del hardware (los sistemas sobre los que se van a realizar las pruebas), los requisitos de software (los requisitos técnicos del engine en cuestión) y los requisitos del banco de pruebas para que el estudio que se va a realizar sea representativo.

Respecto a los requisitos hardware, se plantean como posibles sistemas representativos 3 ordenadores distintos, cada uno de los cuales representará una de las 3 gamas diferenciadas de potencia computacional actual en los sistemas caseros de particulares. Dichas gamas no existen diferenciadas puesto que los fabricantes de hardware, así como la modularidad de los computadores personales, permiten un homogéneo abanico de posibles configuraciones. Es por eso que se escogen tres sistemas como

valores discretos compromiso que representarán a dicho rango continuo de posibilidades actuales.

Se llamarán a estas gamas “alta”, “media” y “baja”, refiriéndose alta a un computador personal capaz de mover los videojuegos modernos más exigentes a una velocidad apropiada para el juego normal. Dicha velocidad apropiada se define en base a la capacidad del ojo humano de ver animación fluida en vez de imágenes separadas. Esto viene a ser aproximadamente a partir de los 20 frames por segundo.

A continuación se decide en base a los requisitos técnicos del engine desarrollado en este proyecto hasta este punto, y que ahora se considera un producto completo. Es importante, sobretodo, que se sepa en todo momento qué parte de la carga no depende del código realizado para este proyecto: El código de terceros. Gracias a la arquitectura llevada a cabo no será necesario realizar ningún trabajo extra para aislar el código de Box2D y Allegro (código de terceros).

Por otra parte, siendo este un juego en 2d sin una inteligencia artificial compleja, la carga computacional recaerá durante la ejecución en el cálculo de físicas y el cálculo de la lógica interna del juego. Lo primero se corresponde con el Box2D (que como ya se ha explicado, ya se encuentra aislado), y lo segundo con el resto de código en cada frame que no haga uso de Allegro (todo lo que no sea render ni input).

Así pues, y en conclusión, los requisitos software del banco de prueba son realizar cargas controladas (de menor a mayor) tanto en el cálculo de físicas como en el cálculo de la lógica interna del juego. Esto se traduce a generar objetos mientras Allegro (el input y el render) se mantienen estáticos: El personaje se mantendrá quieto.

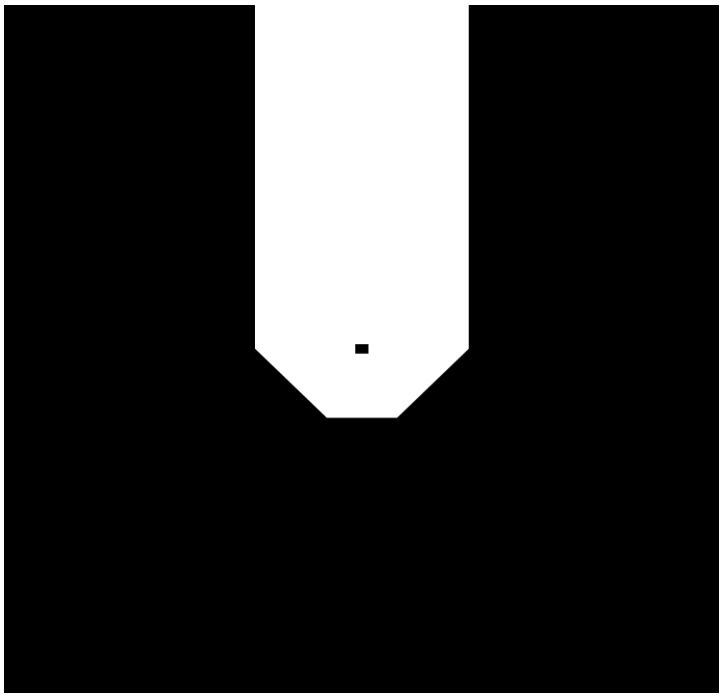
Una vez conocidos los requisitos hardware y software, se procede a cubrirlos con el videojuego banco de pruebas (“juego” a partir de ahora).

Desarrollo del videojuego

Una vez se saben los requisitos y se han analizado con detalle, realizar el videojuego banco de pruebas (“juego” a partir de ahora) es una tarea meramente técnica. El engine se ha realizado para el desarrollo ágil de videojuegos, y como tal brinda al usuario la posibilidad de trabajar en lo que realmente exige el diseño del videojuego, sin la necesidad de trabajar nunca a bajo nivel.

El desarrollo se lleva a cabo sin mayores incidencias, lo cual se considera satisfactorio teniendo en cuenta las características e hitos que se proyectaron para el engine.

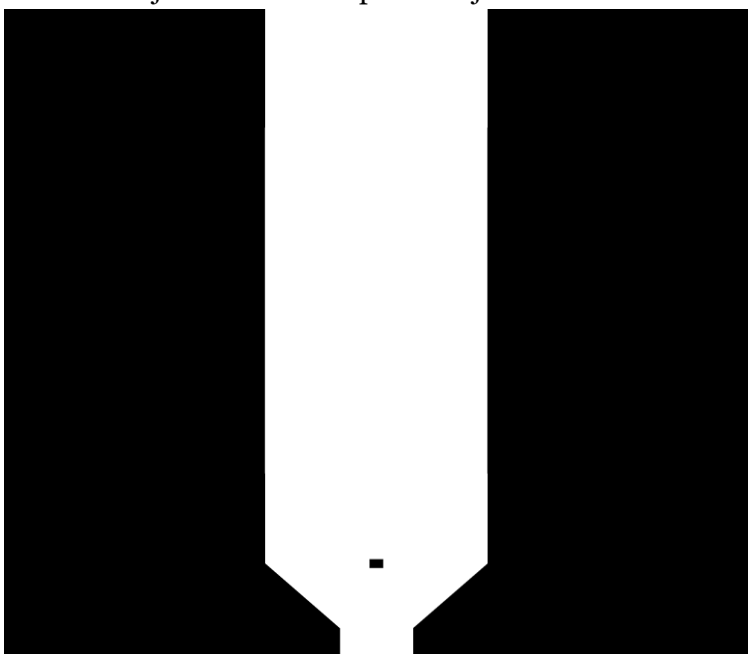
A continuación se incluye el primer mapa de prueba. Se ha de tener en cuenta además de los requisitos anteriormente estudiados el hecho de que el tamaño de cada mapa es de 3000x3000 píxeles y el de la pantalla 1200x800.



Encima del cuadro pequeño se colocará al protagonista, que por el bien de la captura de datos se mantendrá quieto ahí, y junto a él la cámara. En un punto superior de la pantalla se generarán un cierto número de objetos a una velocidad relativa a los frames con tal de lograr encontrar una carga apropiada para las mediciones.

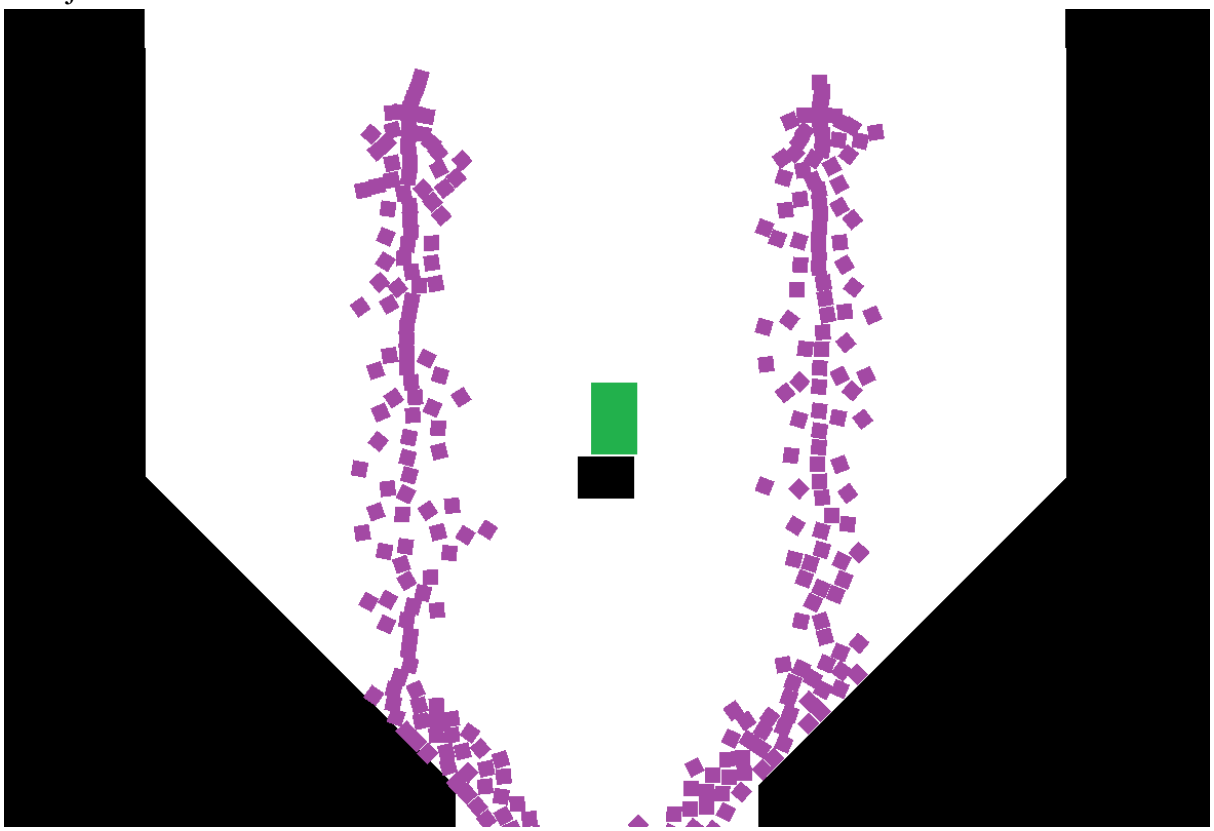
Con 600 objetos en total, generándose a uno por frame, resulta una carga apropiada.

Será necesario un segundo mapa para cargar y descargar de forma controlada la ejecución. Se basará en este primer mapa pero permitiendo a los objetos creados borrarse dejando un hueco por debajo. A continuación se incluye:



Que para permitir al lector la visualización de su funcionamiento, se incluye una captura de pantalla a continuación. En ella se ve al jugador sobre la plataforma central,

y una serie de objetos (violetas) cayendo desde sus fuentes y desapareciendo por debajo.



Profilers

En la adquisición de tiempos se tendrá en cuenta 3 consideraciones de gran importancia:

- Se habrá de lograr la mayor exactitud posible en los tiempos. Para ello se requiere un timer preciso. El timer elegido para el presente proyecto es el High Realtime Timer realizado en la UPV, un timer que asegura una aproximación exacta al tiempo real.
- Un timer no será suficiente para la adquisición de datos que se requiere. Se necesitará un módulo nada intrusivo al cual se le pueda indicar un intervalo. Dado el punto de inicio y el punto de final, dicho módulo tomará el tiempo transcurrido entre uno y otro. De esa forma no se habrá de realizar diferencias ni realizar arreglos: el tiempo adquirido es el tiempo final. A este módulo se le llamará Profiler.
- Para asegurar la mínima intrusión posible, los Profilers guardarán los datos en un array previamente inicializado y no será hasta un momento predefinido que se exportará este array a un fichero. Exportar los datos a un fichero directamente o utilizar alguna estructura de datos más compleja que permita el manejo dinámico de su tamaño puede crear interferencias en los tiempos adquiridos, o incluso generar problemas en ejecuciones muy largas. Así pues, se decide mantener el control sobre una estructura de datos fija, estable y finita, aunque ello limite el tamaño de las adquisiciones a un número prefijado en

tiempo de compilación.

En base a estas consideraciones se desarrolla el módulo Profiler, que se puede conocer con mayor detalle tanto en la documentación interna del código <Src\Engine\Profiler.h> como en la documentación externa <Docs\Profiler.txt>.

El timer ofrecido por la UPV ofrece una serie de herramientas que facilitarán el desarrollo del Profiler. Aún así, y al final, se ha optado por no utilizarlas (el gestor, por ejemplo) puesto que daba problemas no relacionados con el engine ni el Profiler. Sustituciones para dichas herramientas tuvieron que desarrollarse a mano para este proyecto.

Antes de continuar se han de tener claros dos conceptos.

El primero de ellos es el concepto de "frame" (fotograma en castellano). Un frame es una imagen estática en pantalla. Para que una secuencia de frames parezca animada, el ojo humano requiere que dicha secuencia tenga más de 20 frames en cada segundo, a una velocidad continua.

El segundo concepto es el de iteración del bucle principal del juego. Dicho bucle ya se explicó en la sección <Simulación con eventos discreta>. Lo que ahora se viene a explicar e que un frame no se corresponde necesariamente con una iteración de la lógica del juego, o del cálculo de físicas, o de la obtención de input, pero por norma general sí se corresponderá. De esa forma, y aunque no sea estrictamente necesario, se considerará que un frame corresponde con una iteración del bucle principal. En el caso de la ejecución discreta, se corresponderá con una ejecución de Box2D y de Render, siendo ambos los módulos que forzosamente requieren una ejecución continua.

Es importante mencionar que aunque el paradigma planteado por el modo de ejecución discreto pretende destruir el concepto de bucle principal, el presente proyecto no ha adoptado en ningún momento dicho paradigma. En dicho paradigma, el "bucle principal" se puede equiparar, de una forma un tanto vaga, a la "simulación". Como ya se explicó con detalle en secciones previas, este proyecto pretende aplicar la ejecución discreta a un engine realizado en base al paradigma continuo. En consecuencia, en ningún momento representa las virtudes netas del paradigma de ejecución discreto como tal, y en la sección próxima <Trabajos futuros> se instará a llevar a cabo un proyecto similar pero con ese otro objetivo.

Así pues, y aunque no sea correcto, se usará el término de "bucle principal" cuando se hable de ambos modos a la vez, aunque en el modo discreto debiera llamarse "simulación".

Una vez explicado todo esto, y desarrollado el módulo de Profiler, se puede proseguir.

Los Profiler se ha de ubicar en lugares específicos del código para poder adquirir los datos relevantes que se requieren. En primera instancia, los profilers que se requieren son: STEP, IDLE, UPDATE, BOX2D, LOGIC y RENDER. Además, hará falta un reloj: CLOCK. Se explicarán a continuación:

- STEP e IDLE son en esencia complementarios, aunque en la práctica no siempre se complementen. STEP se refiere al tiempo de procesado, e IDLE al tiempo de no procesado. En el modo de ejecución continuo esto viene a significar que, del intervalo de tiempo reservado para cada frame, la fracción que se utilice en recorrer el bucle principal será STEP y el tiempo que no, IDLE. Es por eso que en teoría son complementarios, pues juntos conforman el tiempo de un frame. A efectos prácticos, y teniendo en cuenta que uno de los objetivos de este proyecto es la sobrecarga del sistema, el tiempo de bucle principal tenderá a ser mayor al tiempo de frame, de forma que el tiempo de STEP será mayor que el de un frame, manteniéndose el IDLE en 0 (no se discute el concepto de tiempo negativo, pues no tiene ningún sentido), y sumando entre ambos un tiempo mayor al tiempo del frame.
- UPDATE es el tiempo de STEP (del bucle principal) dedicado a actualizar el juego en sí. No tiene en cuenta retardos del sistema ni input. Así pues, incluye a los siguientes tres Profilers, y es incluído por STEP. A efectos generales, deberá mantenerse muy cercano a STEP en el modo continuo.
- BOX2D y RENDER son dos de los Profilers exigidos en los requisitos software. Abarcan módulos cuyo código son totalmente de terceros (Box2D y Allegro respectivamente), y por lo tanto módulos forzados a la ejecución continua. Es de suma importancia conocer qué parte de la carga de UPDATE forma parte de dichos módulos puesto que el RT-Desk, en un principio, no les afectará de ninguna forma.
- LOGIC se corresponde a toda la parte del UPDATE que no se corresponda ni con BOX2D ni con RENDER. Es el código correspondiente a la lógica interna del juego y, ademas, código que puede beneficiarse de la ejecución discreta.
- Finalmente, e independiente a los profilers, se encuentra CLOCK. Consiste en simplemente un timer que contará el tiempo real transcurrido en la ejecución. Cada vez que la ejecución pase por PlayScene se guardará el valor actual de CLOCK.

Para mayor detalle en los 6 Profilers tanto en modo continuo como en discreto, se puede leer la documentación externa <Docs\Profiler.txt>.

Requisitos especiales de RT-DESK: Saturación

El RT-DESK se ejecuta por medio del método Simulate. Dicho método realiza la simulación hasta el momento presente. Así pues, cuando la carga de la simulación hasta el momento presente es menor que el intervalo de tiempo real para dicha simulación, su ejecución será correcta, y su IDLE será amplio.

En cambio, cuando la carga de la simulación hasta el momento presente es mayor que el intervalo de tiempo real para realizar dicha simulación, antes de poder terminar de procesar dicha carga habrá recibido más carga para procesar. La carga correspondiente al siguiente intervalo. Y así sucesivamente: La carga de cada intervalo requiere del tiempo de procesado de los siguientes intervalos. A este estado se le llamará “estado de saturación”.

De esa forma se entra en un ciclo que, mientra que la carga de los intervalos no disminuya durante un tiempo que permita a la simulación estabilizarse, la ejecución del método Simulate no terminará nunca. Esto afecta de forma directa al profiling, puesto que cuando RT-DESK se encuentra en saturación, los profilers IDLE y STEP no vuelven a actualizarse, quedándose así con los valores "por defecto" en todos los frames siguientes que se ejecutan.

Es de grave importancia entender que el hecho de que el RT-DESK se encuentre saturado no quiere decir que estén dejándose de ejecutar cosas, o que se pierda su orden: RT-DESK asegura que se ejecutará todo, y en el orden en el que se debe ejecutar. La única característica es que al no salir la ejecución del método Simulate, no podrán actualizarse los profilers de STEP e IDLE. El juego, por otra parte, sigue ejecutándose. Y, de salir de la saturación, el valor inmediatamente siguiente del profiler STEP sería equivalente a la duración de la saturación: muy elevado.

En el Desarrollo de las pruebas con CLOCK se mostrará este suceso con datos reales.

Requisitos especiales de RT-DESK: Profiling descentralizado

En el modo de ejecución continuo es fácil delimitar la ejecución de un frame, puesto que todas sus distintas secciones se ejecutan secuencial y ordenadamente. El profiling se realiza de forma centralizada y ordenada.

Por otra parte, en el modo discreto, no todos los módulos se van a ejecutar a la misma velocidad o con la misma latencia. Esto crea un gran problema a la hora de hacer profiling: Un frame de RENDER puede no corresponder con un frame de STEP, puesto que se ejecutan asíncronamente. A su vez, la localización de algunos profilers debe cambiar, puesto que mientras que en modo continuo se hacía una llamada ahora se envía un mensaje. La llamada terminaba cuando se había ejecutado todo su contenido enteramente, por lo que su tardanza era representativa. Por otra parte el envío de un mensaje no incluye la tardanza de su tratamiento.

Para ello se creó, también, el temporizador CLOCK dentro de profiling, puesto que permite un análisis independiente de los sucesos.

Pruebas

Formato y especificaciones

El formato elegido para la obtención y almacenaje de datos es el de valores separados por comas (CSV, del inglés "Comma Separated Values").

Dicho formato no tiene un estándar actualmente, aunque su uso es lo suficientemente general como para poder usarse de forma estable. En este formato se reservan un número X de filas iniciales como etiquetas (o posibles formato de las filas, según el convenio, aunque esto se reservará al sistema encargado del dibujo), y de ahí en adelante cada fila se corresponderá con un valor en las columnas correspondientes. Todo ello en texto plano.

Un ejemplo:
Etiqueta1, Etiqueta2, Etiqueta3
0.374, “Primero”, 4234
0.9999, “Segundo”, 29034

Por otra parte se recuerda que en <Zentract\Docs\RTDESK Message Delays.txt> se podrán encontrar los distintos “perfiles de latencia”, a los cuales a partir de ahora solo se les llamará por el nombre técnico significativo descrito en dicho documento.

Finalmente, el número de muestras y objetos se podrá encontrar y modificar en <Zentract\Engine\Settings.h>. Con él se modifican los datos en los Profilers (número de muestras) y en el PlayScene (número de objetos) directamente desde Settings.

Se procede a comentar el desarrollo de las pruebas.

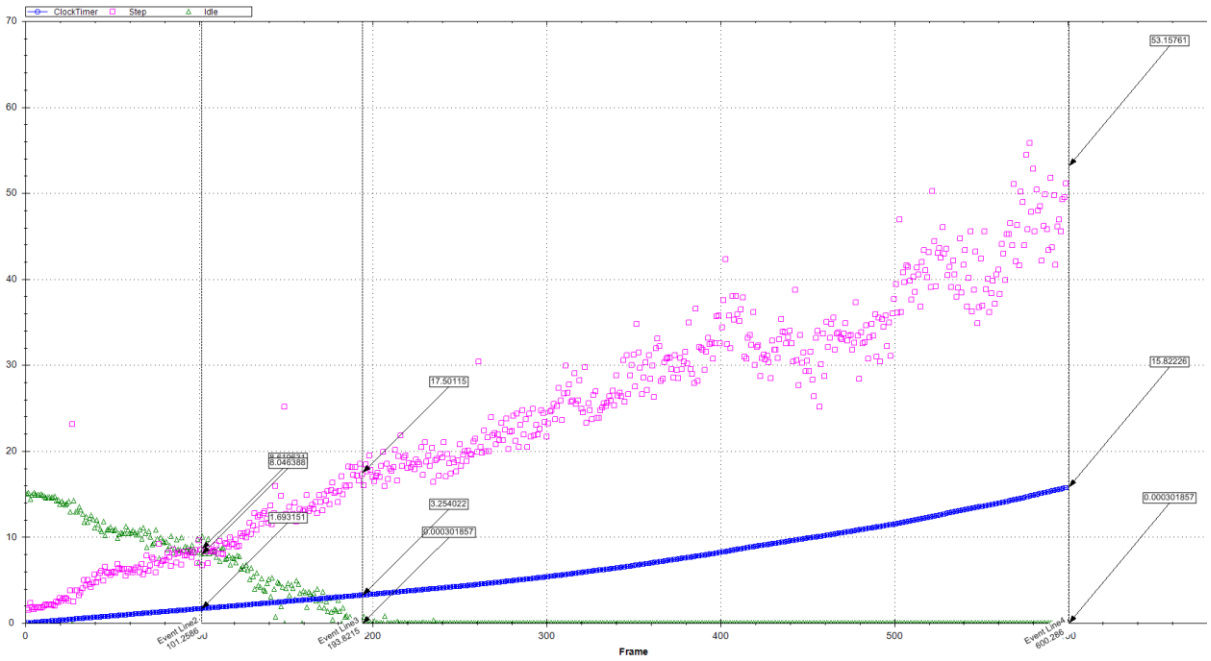
Desarrollo de las pruebas con CLOCK

Para estas pruebas se ha elegido generar 600 entidades, una por frame, durante 600 frames. El perfil de latencia escogido es el “base”, y la velocidad de ejecución 60 frames por segundo (16.67 milisegundos cada frame). Todos los ficheros y datos se encuentran en <Zentract\Test001>.

Se recuerda que CLOCK es un temporizador, medido en segundos, cuyo valor guardaremos cada vez que suceda un frame real del juego. Gracias a él podemos comprobar de forma neta el momento inmediato en el que un frame se ejecuta. Con él se puede medir de una forma clara y concisa la velocidad (y frecuencia) a la que se ha llevado a cabo la ejecución.

Puesto que en el modo discreto los módulos se ejecutan de forma asíncrona, como ya se comentó antes y se explicará con mucho más detalle en el siguiente apartado, con CLOCK se podrán comparar de forma directa y sin lugar a dudas los resultados obtenidos.

Antes de nada, se procede a tomar datos de una ejecución normal en el modo continuo. A continuación la gráfica:

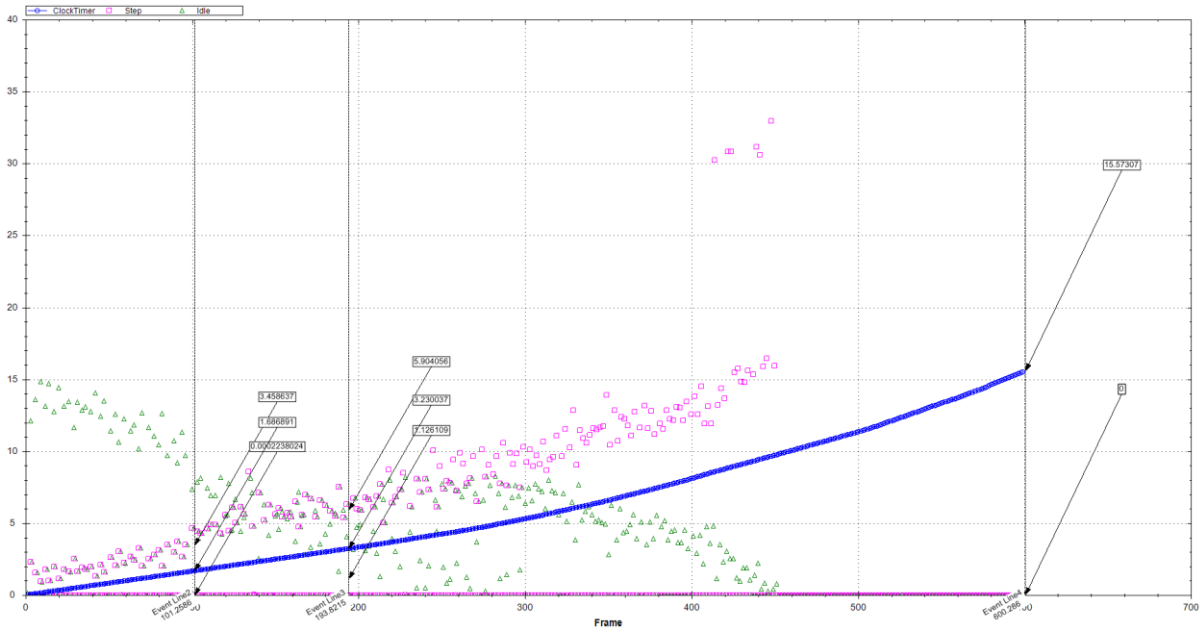


En el eje X se encuentra el número de frame, y en el eje Y la latencia de un profiler en dicho frame. La leyenda para esta gráfica y las siguientes de esta sección es:
Rosa: STEP. Verde: IDLE. Azul: CLOCK.

- Según van apareciendo las entidades, el STEP va creciendo linealmente, así como la dispersión de sus muestras.
- Es cerca del frame 100 donde STEP e IDLE se cruzan, y cerca del frame 200 (por lo tanto con 200 entidades en pantalla) cuando deja de haber tiempo de IDLE.
- Se puede localizar en este momento el comienzo del llamado “lag”: cuando un frame dura más del tiempo esperado (16.67ms), y por lo tanto los frames por segundo bajan de 60.
- Es a partir de aquí cuando el tiempo total de la ejecución (en CLOCK) empieza a no corresponderse con el tiempo real debido a dicho lag.
- Teniendo en cuenta la tendencia de STEP, se encuentra su tiempo en el último frame de aproximadamente 53ms.

Para mayor comodidad se han trazado 2 líneas verticales: una en el punto donde IDLE y STEP se cruzan, y otra en el punto en que IDLE llega a 0.

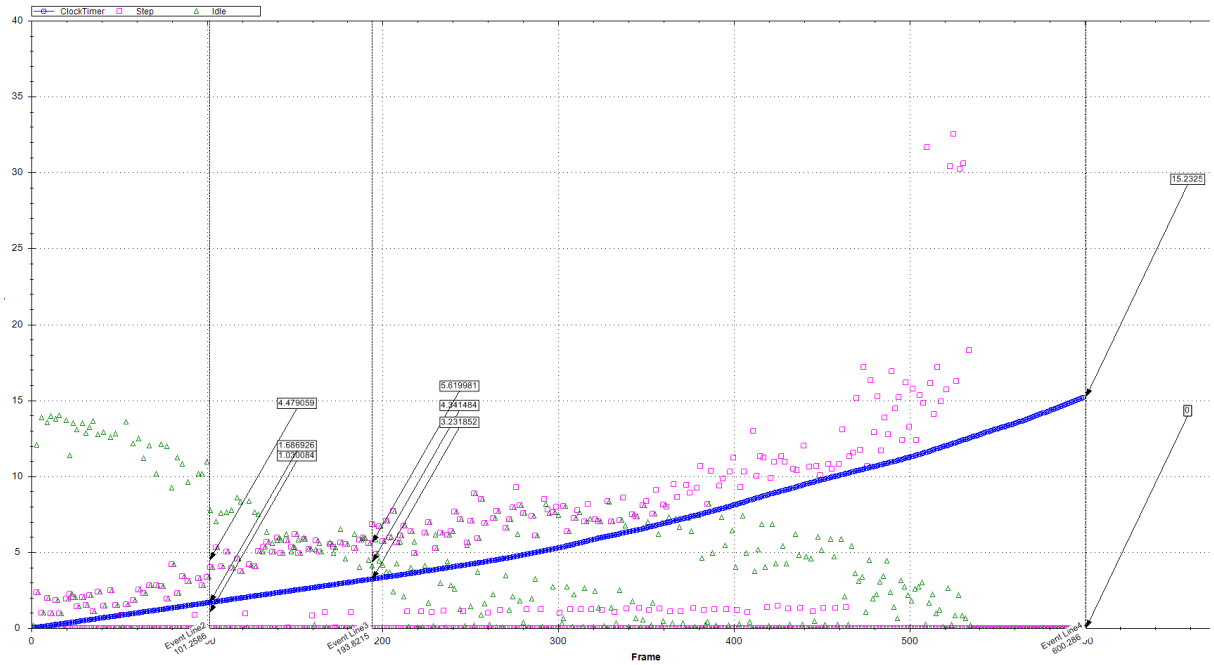
Se procede a continuación a comprobar la simulación RT-DESK. Para ello se ejecuta en el modo discreto con el perfil de latencias base. Usándose las líneas verticales para comparar, se puede ver una clara mejoría en la ejecución. Se obtiene la siguiente gráfica:



- Teniendo en cuenta la tendencia de STEP e IDLE, se cruzan aproximadamente desde el frame 150 hasta el 270.
- Por otra parte, IDLE llega a 0 y por lo tanto la simulación entra en estado de saturación sobre el frame 450.

La mejoría es latente, y se puede comprobar en la ejecución del juego. Visto esto, se procede a comprobar el comportamiento del estado de saturación realizando otra prueba más sobre el perfil de latencias base.

En este nuevo perfil tan solo se modificarán las latencias de actualizado de Input y Render. Ambos módulos hacen uso de Allegro (código de terceros), y este a su vez usa la WINAPI para el Input y OpenGL para el Render. Aún así, no va a producirse ningún Input, y el Render como se comprobará en la siguiente sección, tan solo genera un coste lineal. El actualizado de Input y Render (que, se recuerda, se llaman para actualización a sí mismas) tenían una latencia de 16.67ms (60FPS), y ahora se le pondrá de 33.34ms (30FPS). Esto quiere decir que, mientras que el resto del juego se ejecutará 60 veces cada segundo, el Input y el Render tan solo se ejecutarán 30 veces: La mitad.



- STEP e IDLE se cruzan desde 150 hasta 350, a partir de donde se separan.
- Finalmente, IDLE toca 0 aproximadamente en 530, momento en que se satura.

Ralentizar el Render e Input ha permitido al RT-DESK ir más ligero durante casi 100 frames y por lo tanto 100 entidades más. Dicha mejora es considerable, más aún si se compara con la ejecución continua. En la siguiente tabla se incluye los datos de CLOCK:

Configuración	**IDLE == STEP**	**IDLE == 0 (saturación)**	**CLOCK (seg)**
Continua	150	200	15.8223
Discreta, Input y Render a 60FPS	150 a 270	450	15.5731
Discreta, Input y Render a 30FPS	150 a 350	530	15.2325

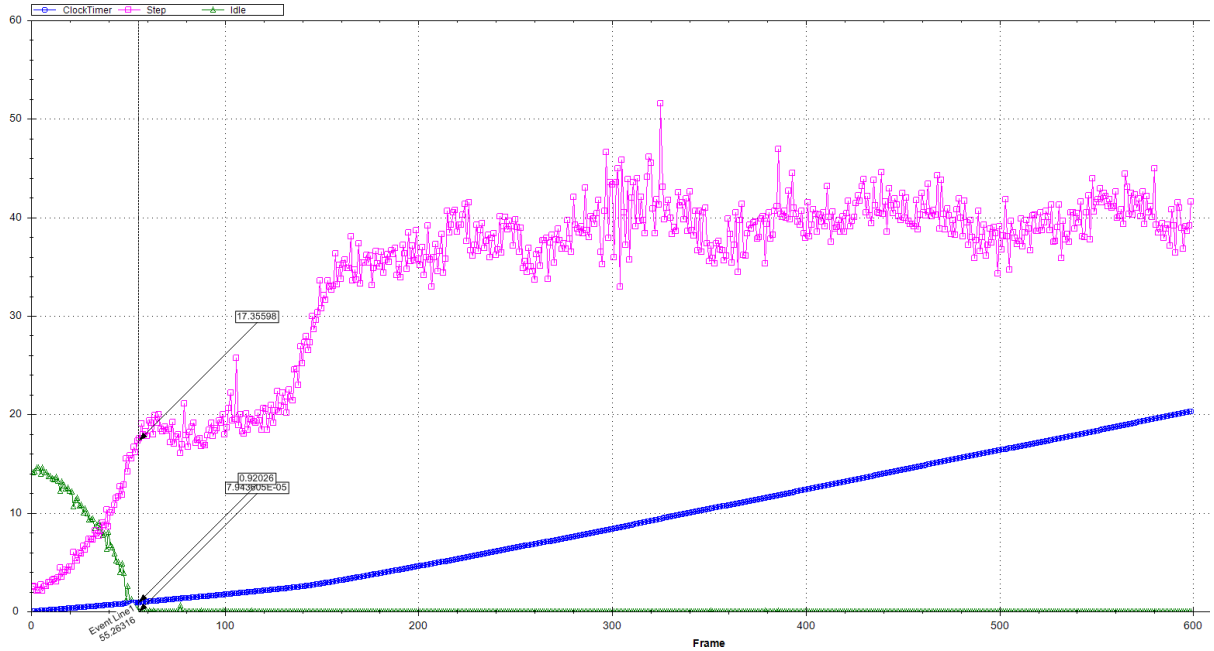
Con configuraciones en las actualizaciones del Input y del Render son menores que 30FPS el juego se torna injugable y por lo tanto no se han tenido en cuenta. Esto sucede porque por debajo de los 30 frames por segundo el ojo humano deja de ver animación, y por otra parte porque el Input a esa velocidad de actualizado es excesivamente lento comparado con los reflejos humanos.

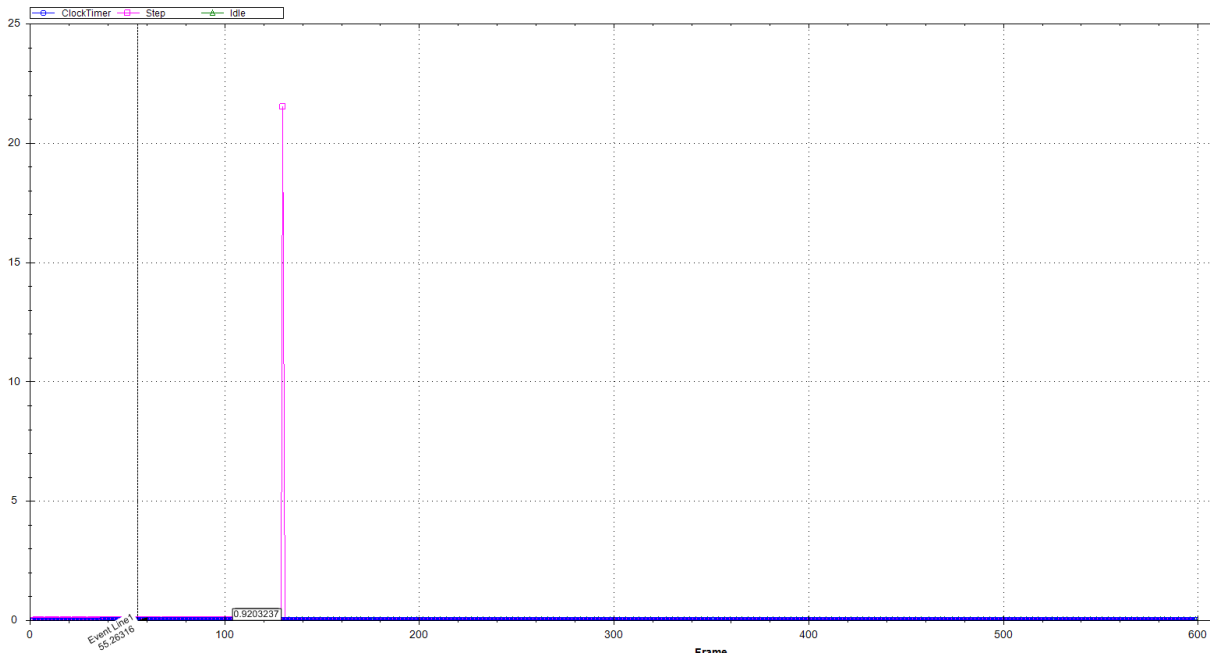
A continuación se van a llevar a cabo un par de pruebas para profundizar un poco en el comportamiento de la saturación del RT-DESK. Más específicamente, en su estabilización tras salir de este.

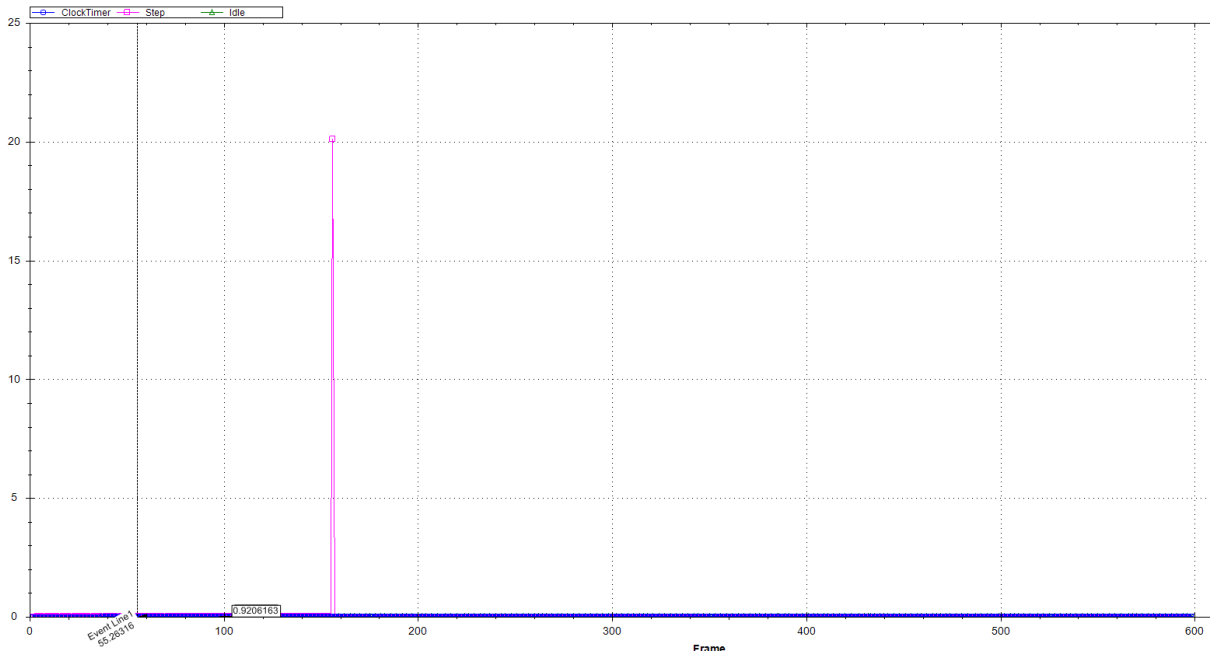
Lo que vamos a buscar en las siguientes pruebas es un frame que tome una cantidad desorbitada de tiempo al RT-DESK, tras el cual volverá a estabilizarse.

La primera de ellas creará una carga justo para saturar el sistema (o lagearlo en el modo continuo) controlada y tras ello se irán eliminando progresivamente, al mismo ritmo que se van generando. Para ello se usará el mapa número 2, dos fuentes de objetos y 600 objetos a generar en total en cada una de ellas, uno por fuente y por frame.

A continuación incluyo las gráficas del modo continuo, discreto con la configuración base, y discreto con la configuración IR030 (input y render a 30FPS, resto como base):

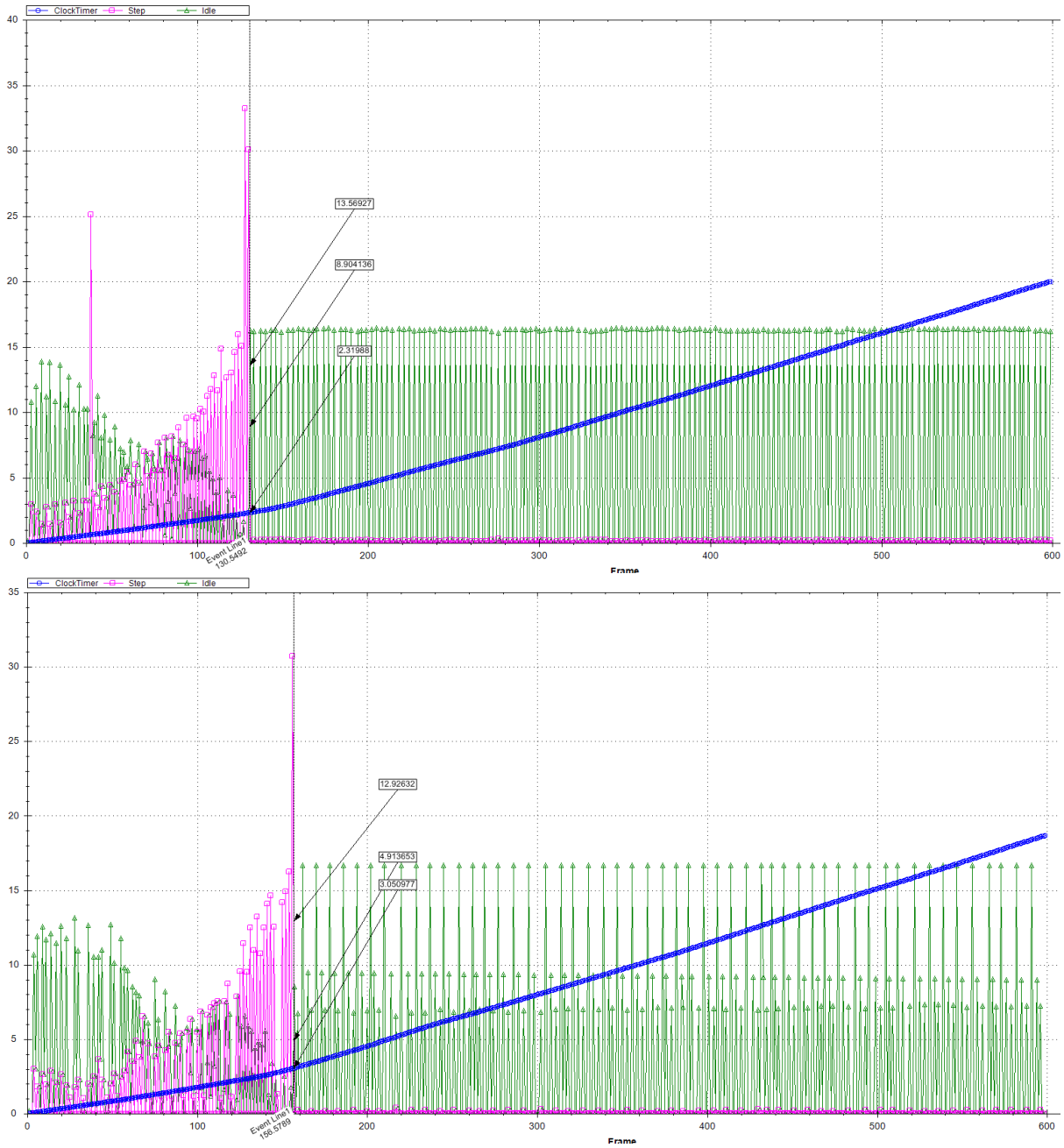






He incluído una linea vertical en el punto en que IDLE==0 en el modo continuo para comparar. Resulta evidente en qué frame se produce la saturación en el modo discreto. En el modo continuo es en el frame 55, en el modo discreto con la configuración base es en el frame 130, y en la configuración IR030 en el frame 155. La mejora del modo discreto se puede comprobar de nuevo que es sustancial. Pero no es eso lo que se venía a comprobar. Los 2 frames de saturación duran, correspondientemente, 21 y 20 segundos, que son valores mayores a los tiempos que indica el CLOCK cuando termina la prueba. Más sobre esto adelante.

Ahora que sabemos que estamos saturando y estabilizando de forma correcta la ejecución, se puede comprobar el comportamiento al salir de esta. Basta con eliminar el pico extremo y comprobar qué sucede a partir del frame en que sucede la saturación. Por razones de facilidad, he añadido una linea vertical en el frame en que se supone que sucede la saturación. BASE e IR030 a continuación:



Queda comprobado, pues, que el comportamiento del RT-DESK al estabilizarse de forma natural tras un estado de saturación es el deseado. Aún así, de estas gráficas se pueden deducir ciertas características:

- El siguiente frame que se ha logrado obtener tras la estabilización tiene una carga prácticamente nula, con un IDLE completo.
- Si se tiene en cuenta el dato sobre CLOCK y la duración de los frames de saturación antes mencionados, se concluye que el tiempo de estabilización es más largo que el tiempo en el que la carga realmente permitiría una ejecución normal. La causa de este comportamiento es lógica: Todos los eventos que se han visto retrasados debido a la propia saturación se ejecutarán en un instante más tardío al correcto. Por mucho que se aplique la ejecución discreta y asíncrona, la potencia real de cálculo del procesador no se puede aumentar. Se concluye pues que la re-estabilización no es instantánea.

- Por otra parte, la ejecución estable continúa de una forma correcta y prevista una vez se ha restablecido.

Como último detalle, y de nuevo indicado como dato colateral, los datos de CLOCK al final de la prueba han sido:
Continuo: 20.3183 seg
Discreto con BASE: 19.9903 seg
Discreto con IR030: 18.6715 seg

Características de las pruebas de profiling en una ejecución discreta

Para estudiar los profilers en el modo discreto, como ya se comentó en su subsección correspondiente, se han de tener en cuenta ciertas consideraciones especiales. En las gráficas en las que el eje X se corresponde con el número de frame:

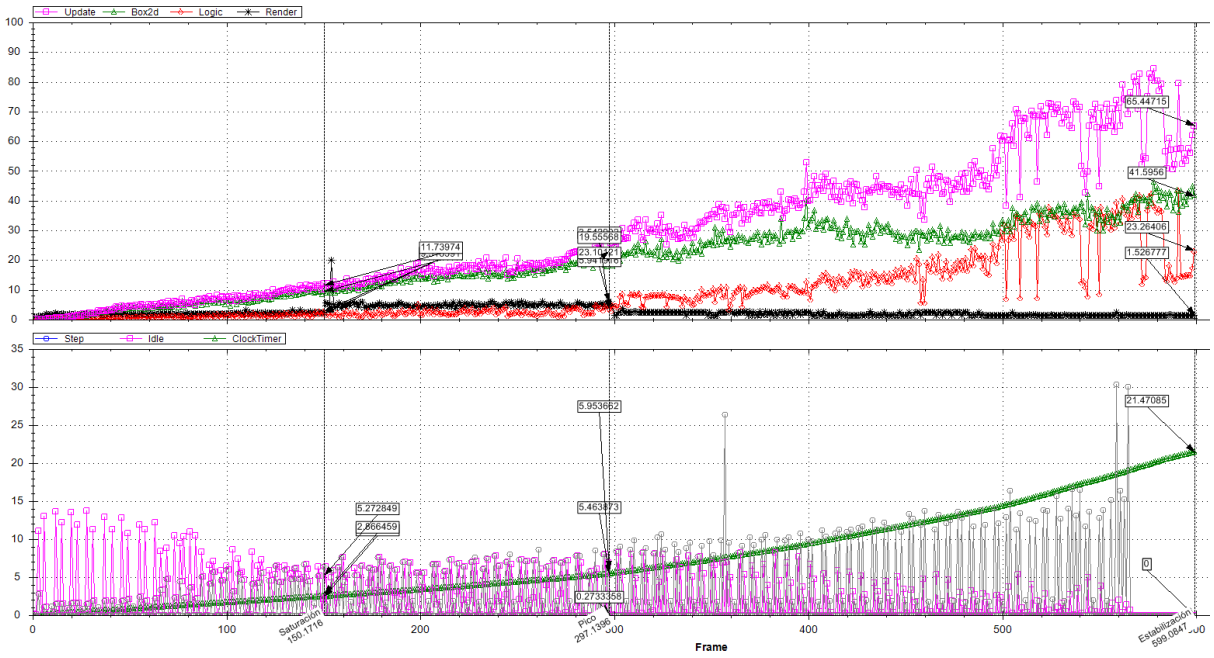
- Un módulo ejecutándose a 60FPS tardará 10 segundos en ejecutar 600 frames.
- Un módulo ejecutándose a 30FPS tardará 20 segundos en ejecutar esos mismos 600 frames.
- Un módulo ejecutándose a 90FPS tardará 5 segundos.

Por ello se propuso la alternativa al profiling por frame: profiling con temporizador. Esta técnica consiste en, puesto que lo que nos interesa es conocer el comportamiento de un módulo durante los 600 frames que tardaría la ejecución a 60FPS, lo que se hará para sincronizar todo el profiling será tomarlo según tiempo y no según frame. En el ejemplo actual, esto sería: 10 segundos. Con esta técnica no se puede tener en cuenta la duración de cada frame individual, sino el número de éstos que se ejecuta durante cada intervalo y compararlo con el número teórico:

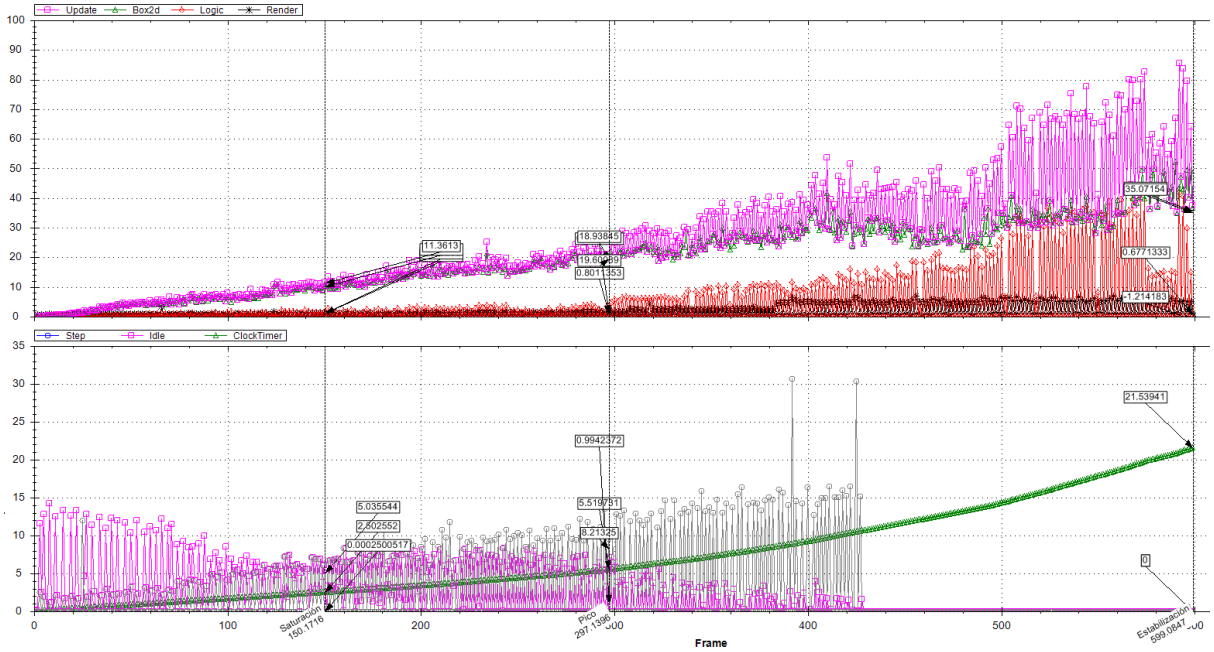
- Un módulo que debería estar ejecutándose a 30FPS se está ejecutando a 28FPS, por lo que hay 2 frames de latencia por segundo.
- Por medio de esta latencia, comprobar qué configuración es más apropiada y medir cuánto se diferencia la ejecución de cada módulo de la versión continua en cada una de dichas configuraciones.

El mayor inconveniente de esta técnica es que no permite tener en cuenta la duración de cada frame individual. Para estudiar el comportamiento de los módulos en una ejecución no saturada puede resultar útil, pero en el momento en que la ejecución se satura lo que más nos interesa es justamente el dato que se está dejando de capturar. En consecuencia, se desecha esta técnica y se buscan métodos alternativos.

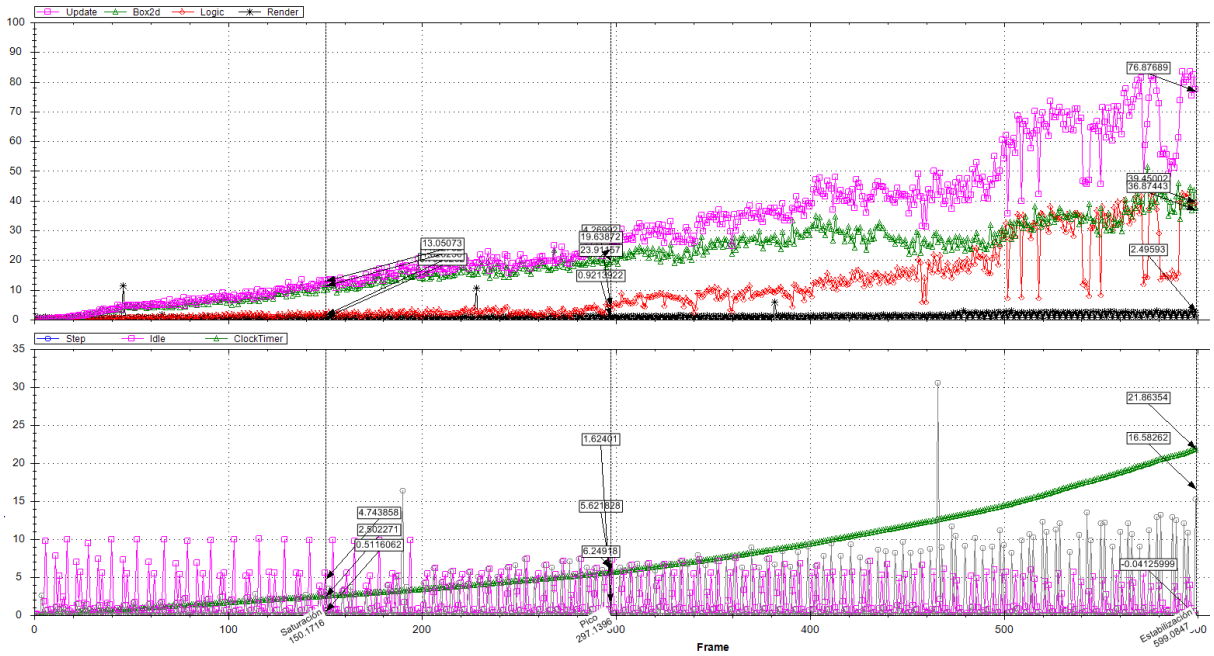
Para comprobar lo que sucede usando el profiling usual del modo continuo, a continuación se incluyen 3 ejecuciones en modo discreto. En ellas se usará la misma configuración que para la primera prueba de CLOCK: se empieza con 0 objetos, y durante 600 frames se genera uno por cada frame. En la primera gráfica Input y Render se actualizan a:
30FPS


Update
Box2d
Logic
Render
65.44715
41.5956
23.26406
1.526777
11.73974
Step
Idle
ClockTimer
21.47085
5.953662
5.463873
5.272849
2.866459
0.2733358
0
Saturación
150.1716
Pico
297.1396
Estabilización
599.0847
Frame

60FPS


Update
Box2d
Logic
Render
35.07154
0.6771333
-1.214183
11.3613
18.93845
0.8011353
Step
Idle
ClockTimer
21.53941
0.9942372
5.519751
5.035544
2.602552
0.0002500517
0
Saturación
150.1716
Pico
297.1396
Estabilización
599.0847
Frame

90FPS



Por cada caso hay 2 gráficas: La de arriba es la de profilers (UPDATE rosa, BOX2D verde, LOGIC rojo y RENDER negra). La de abajo es la de clock (CLOCK verde, STEP gris e IDLE rosa).

Lo importante en este caso es el RENDER en color negro. En 60FPS está ejecutándose a su propio ritmo de forma discreta pero sincronizado de forma general con UPDATE (y por lo tanto con la lógica y box2d). Se puede comprobar que cerca de 375 frames se produce un salto de consumo en el Render que ya dura hasta más allá del final del muestreo.

Este mismo salto al estar en 30FPS sucede aproximadamente en el frame 150. 150 no se corresponde con la mitad de 375, así que han de haber más factores en juego.

Por otra parte, este cambio afecta a la simulación de RT-DESK. Como ya se comprobó al principio de la sección previa, al realizarse menos ejecuciones por segundo la simulación está menos cargada y tarda más en saturarse.

En la prueba de 90FPS se ve como la subida de Render sucede fuera de las 600 muestras. Esto se debe a que el Render va tan rápido que durante los 600 frames no ha llegado a suceder la carga de entidades suficiente para causarse la subida.

Además, y aún más interesante y problemático que lo que se pretendía estudiar con esta prueba, es el hecho de que al verse el RT-DESK forzado a ejecutarse a 90FPS específicamente para el Render e Input, ha tenido que realizar más frames por segundo (no es seguro que sean 90, puesto que aunque sea el módulo más exigente, puede estar desincronizado con el resto de módulos y por lo tanto no marque su ritmo). En consecuencia, la saturación de la simulación también queda fuera del gráfico, ya que se han realizado un número mucho más grande de simulaciones en un intervalo de tiempo más pequeño.

Para poder tener en cuenta todas estas diversidades hay dos opciones:

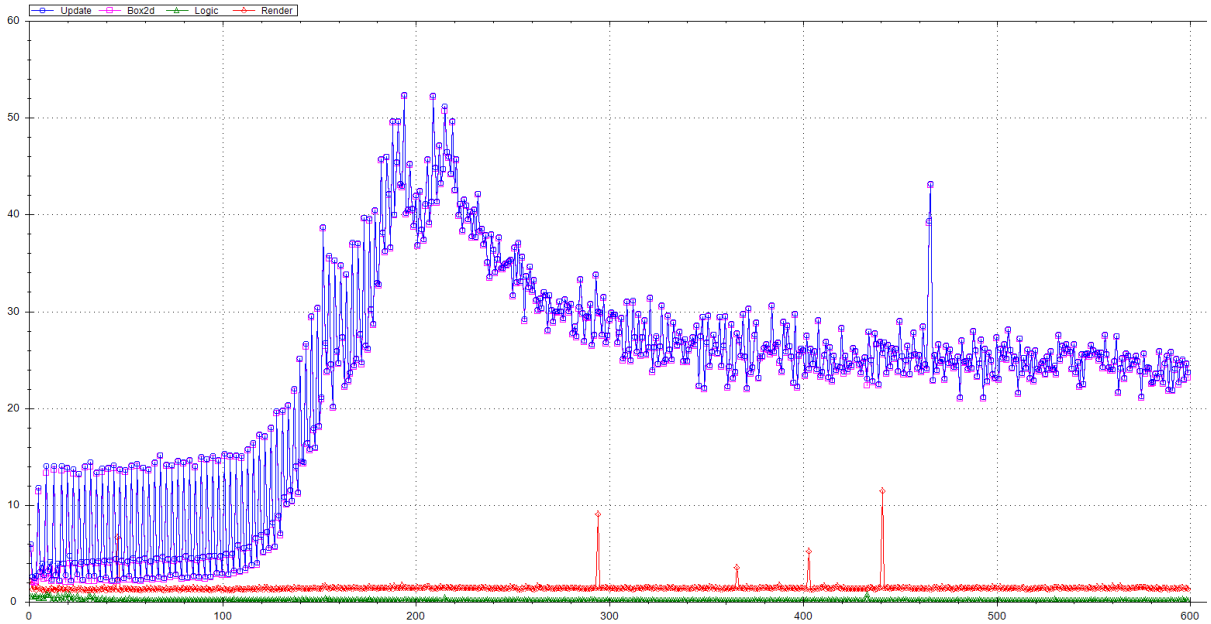
1. Hacer muestreos, gráficas y comparaciones que tengan en cuenta estas variaciones
2. Implementar una nueva técnica de profiling discreto y variable que se “ancle” a un módulo (por ejemplo el Update para el presente juego) y que, respecto a él y sus iteraciones, se ordenen y coloquen temporalmente el resto de ejecuciones. La complejidad de esta técnica conlleva a que se proponga como futuro proyecto, ampliación, o nueva implementación para el propio RT-DESK.

Para este proyecto se ceñirá a la primera opción, puesto que la segunda se sale considerablemente del alcance predispuesto. Así pues, se realizarán las pruebas y los análisis teniendo en cuenta esta característica.

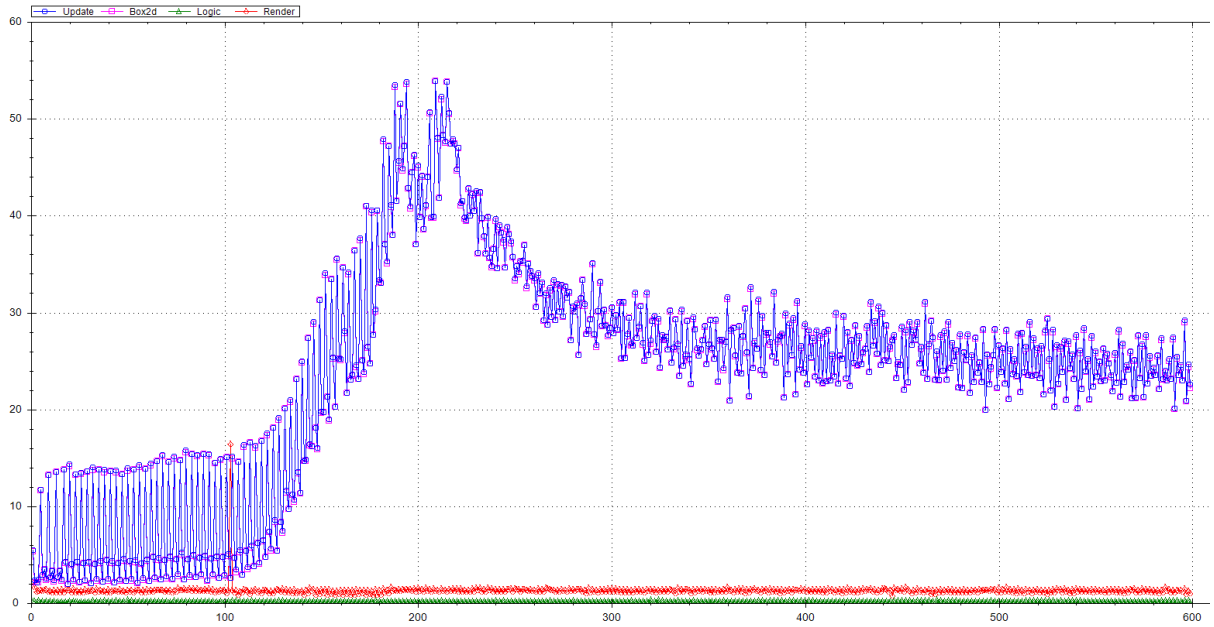
Desarrollo de las pruebas con profilers

Teniendo en cuenta la consideración final de la sección anterior, se procede a realizar una prueba generando 600 objetos encima de la pantalla, y dejándolos caer y amontonarse en la parte visible del mapa 1. Se activan todos los profilers y se prueba con el modo continuo, el modo discreto con la configuración BASE y la configuración IR030 (input y render a 30FPS, el resto a 60FPS se recuerda).

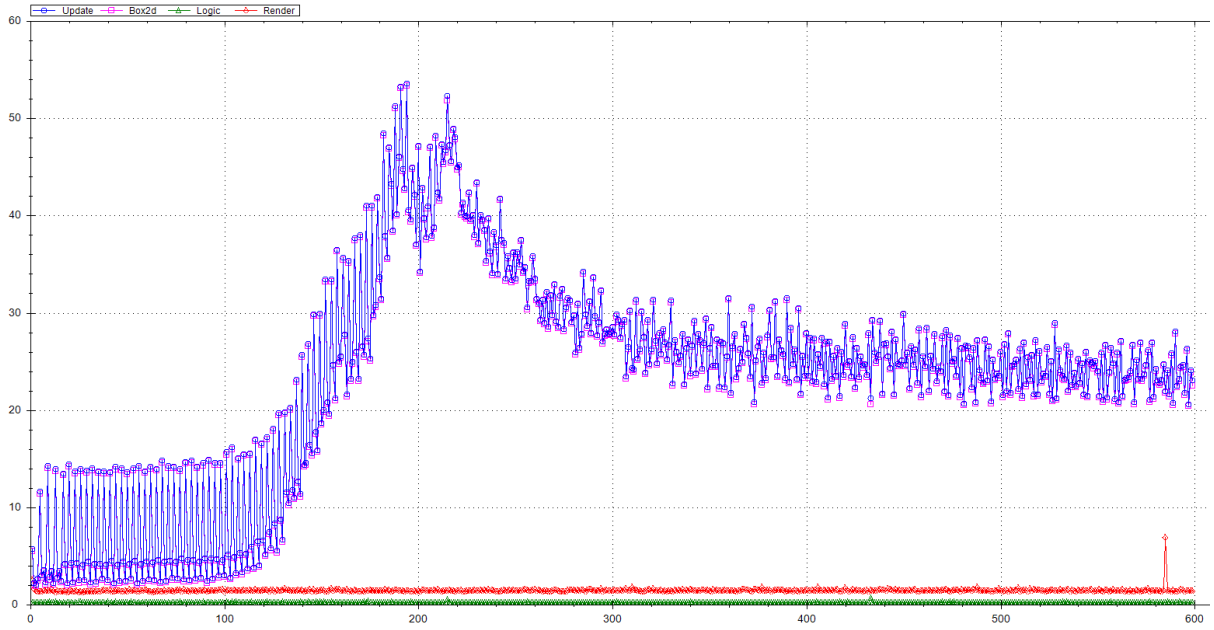
Continuo



Discreto con BASE



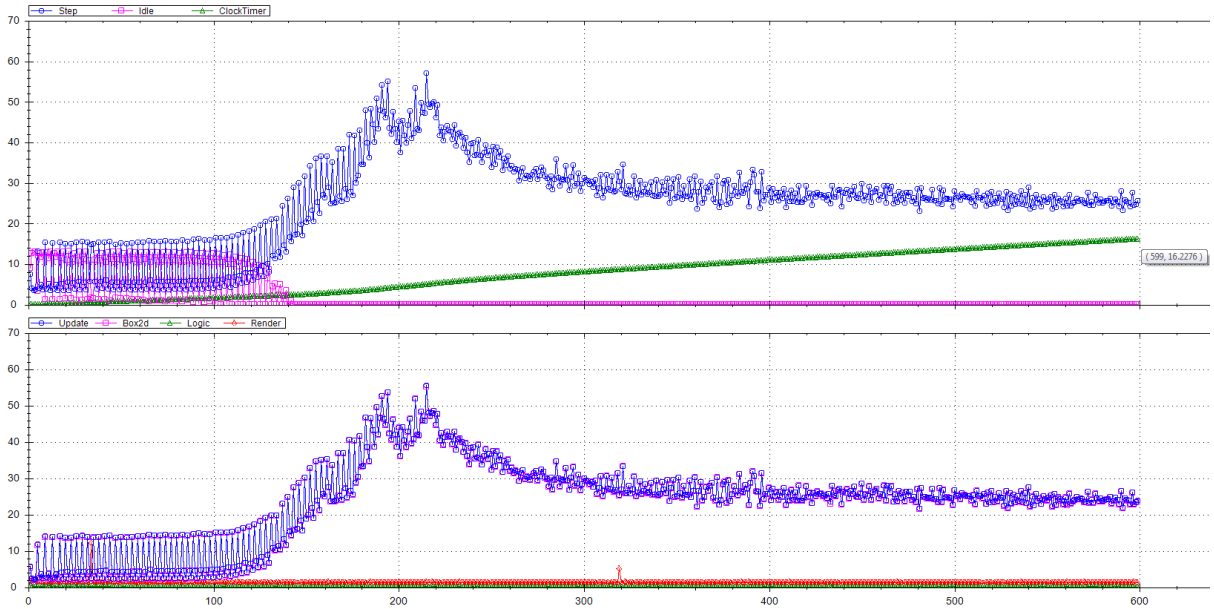
Discreto con IR030



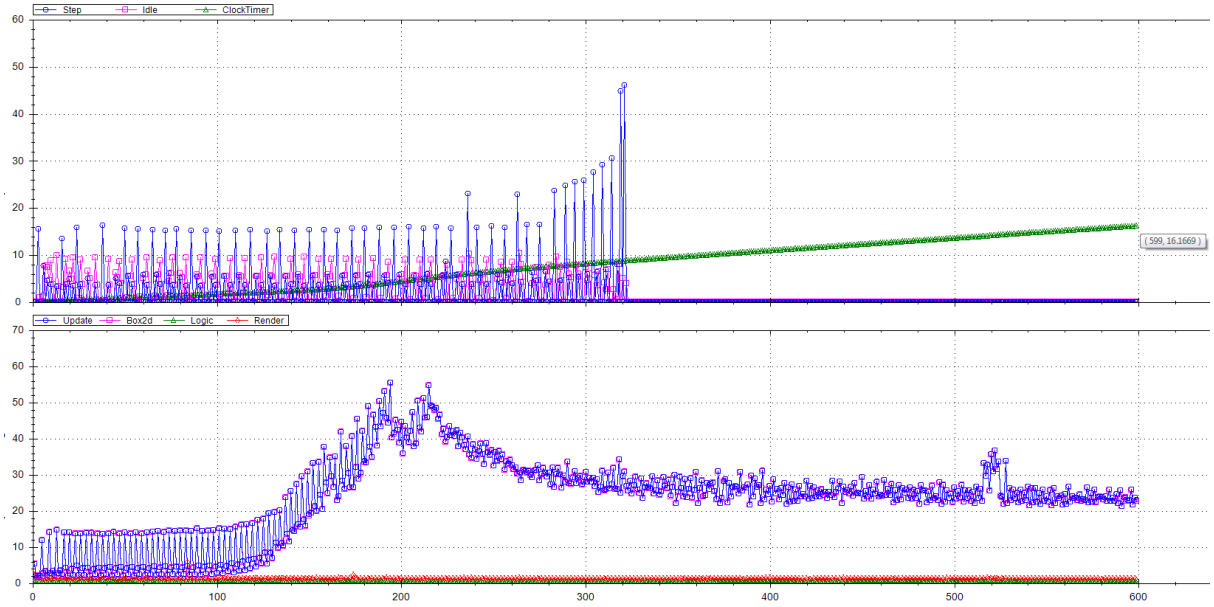
En secciones previas se comprobó que se producían mejoras en la latencia de la ejecución optimizando el tiempo de IDLE y de STEP, pero lo que son las latencias específicas de cada uno de los módulos se mantienen iguales. De la actual prueba se realizaron, como con las demás, varias pasadas. El resultado de la mayoría era similar al que se expone: No hay mejora.

A continuación pongo las gráficas del continuo y del discreto con base mostrando tanto los profilers (como arriba) como el CLOCK, STEP e IDLE, para comprobar si a pesar de no haber mejora en los módulos, sigue existiendo esa mejora en la latencia mientras la ejecución está saturada:

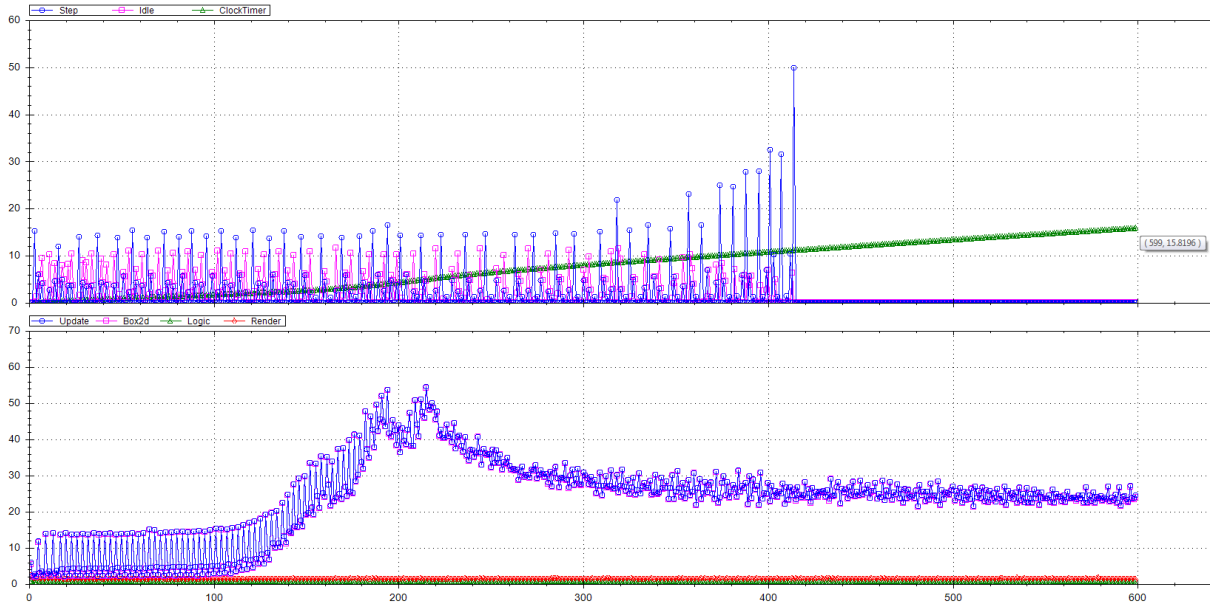
Continuo



Discreto con BASE



Discreto con IR030

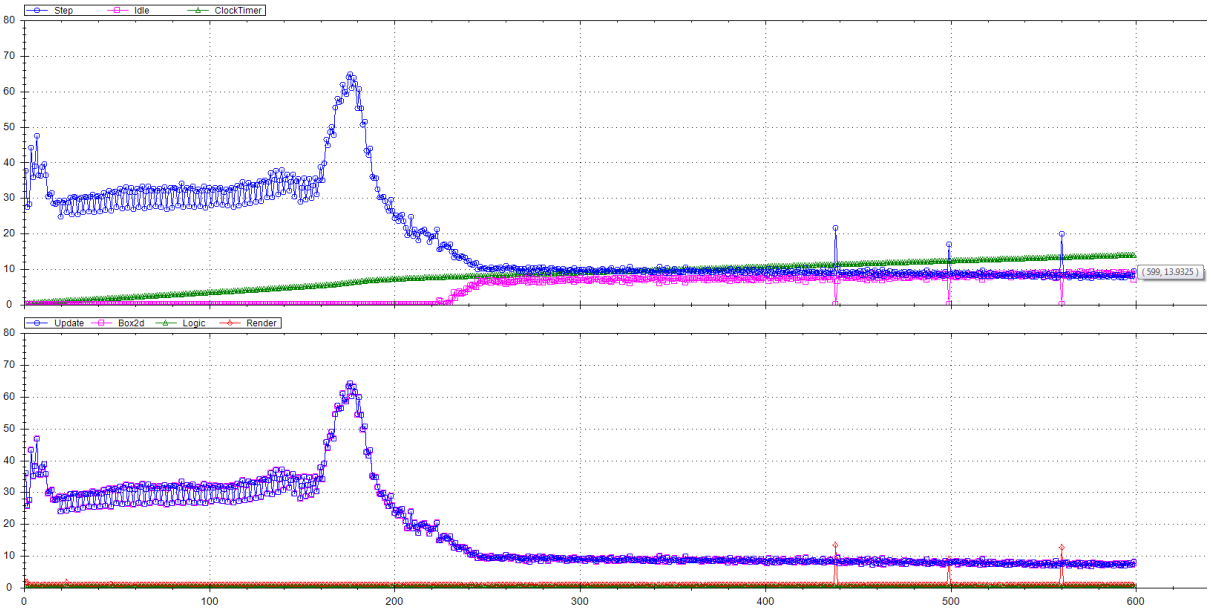


Por si se da el caso de que en la copia del lector no se llega a leer correctamente, los datos de CLOCK al final de la ejecución son los siguientes:
Continuo: 16.2276
Discreto BASE: 16.1669
Discreto IR030: 15.8196
Por lo que se puede comprobar de nuevo que dicha mejora existe, aún cuando los módulos siguen tardando lo mismo.

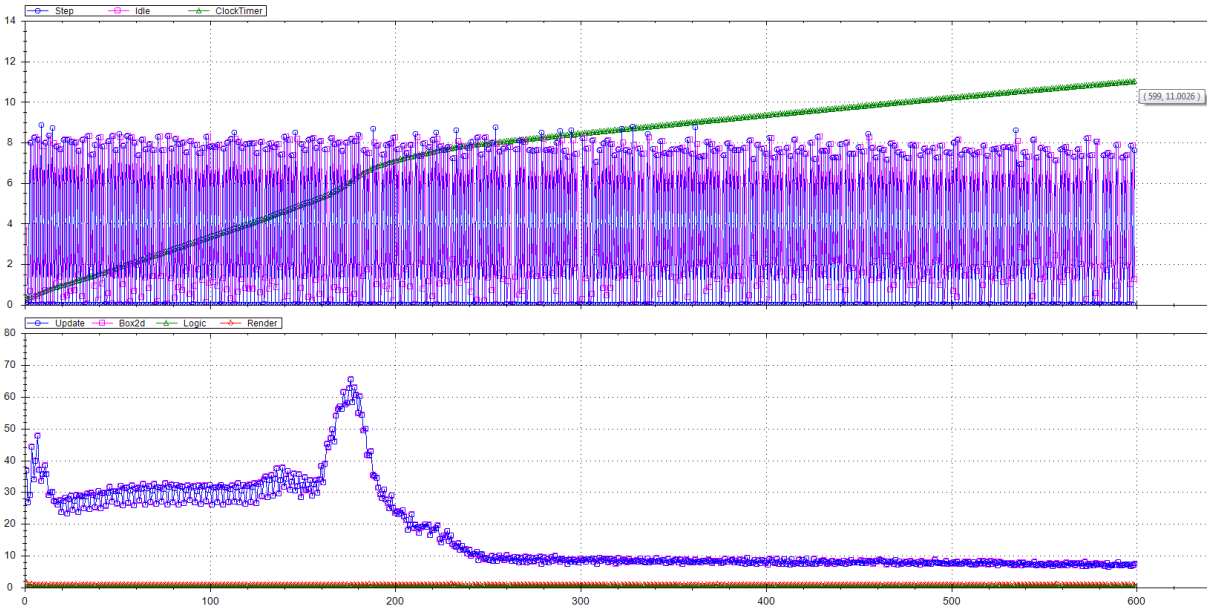
Para proseguir se realizará la misma prueba pero sobre un mapa con comienzo y condiciones distintas. Así se comprobará si en otra situación distinta sí se observan mejoras.

Para esta segunda prueba se ha escogido el mismo mapa pero con tan solo 300 entidades en vez de 600, que además aparecerán todas juntas. En otras palabras: se realizará una carga grande de forma espontánea, en vez de varias cargas medias secuenciales. A continuación las gráficas:

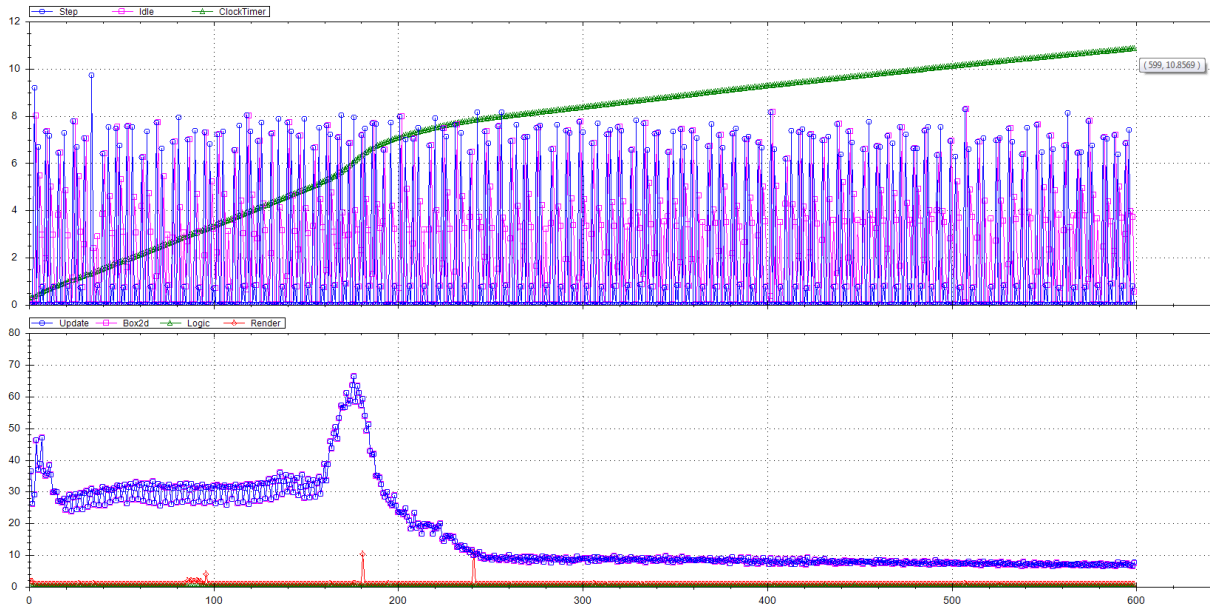
Continuo



Discreto con BASE



Discreto con IR030



De nuevo los resultados de los profilers son iguales en sendas ejecuciones. Por otra parte, el STEP e IDLE del RT-DESK se mantienen apropiadamente estables. Esto es porque ha habido una saturación al comienzo (ignorada en las gráficas para poder visualizar el CLOCK correctamente puesto que cubría la mayor parte de la ejecución, haciendo un pico de aproximadamente 10000 ms).

Por si se da el caso de que en la copia del lector no se llega a leer correctamente, los datos de CLOCK al final de la ejecución son los siguientes:
Continuo: 13.9325
Discreto BASE: 11.0026
Discreto IR030: 10.8569
De nuevo lo mismo. En este caso la mejora es sustancial. Esos 2 segundos, puestos en perspectiva y extrapolándolos a una ejecución de juego continua, constituyen una diferencia enorme. Sobre todo teniendo en cuenta que la ejecución actual dura apenas 15 segundos.

La conclusión definitiva es que, lamentablemente, no se han producido mejoras en la ejecución de los módulos. Habrá que tener

Desarrollo de las pruebas con distintos hardware

Hasta ahora todas las pruebas se han realizado con el computador principal descrito en el Apéndice D: https://docs.google.com/document/d/1Ee8KmKPIUbFZM16AZIz_dqw3-RFa82mCJalERldeVro/edit#heading=h.4g4m2toitnxr

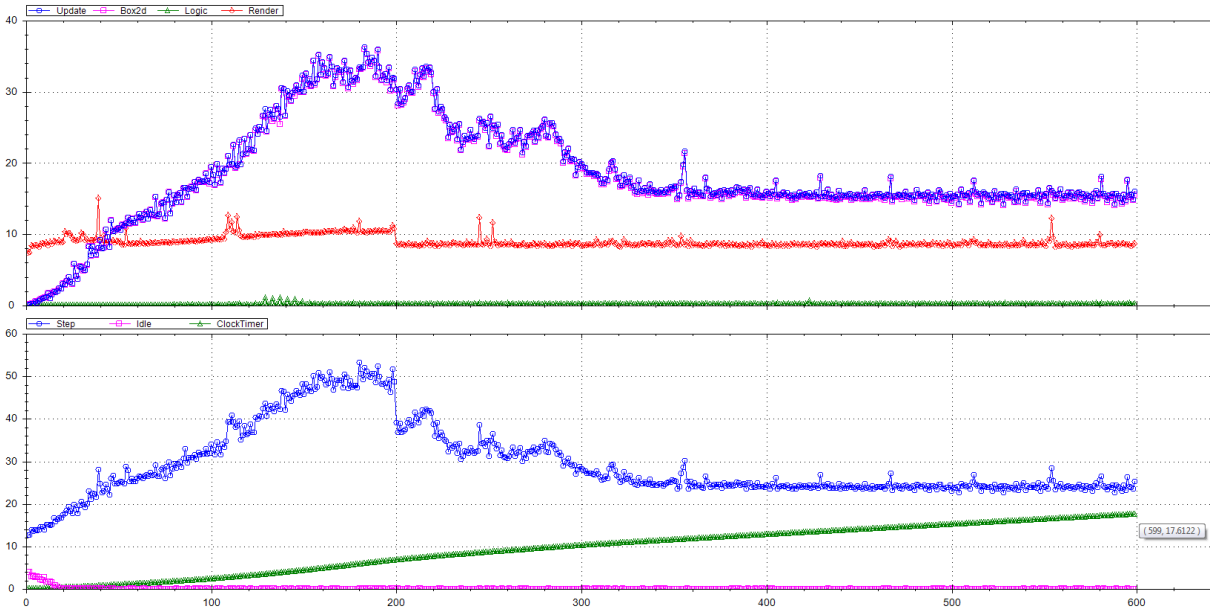
Por razones de exhaustividad, se han dispuesto de otros 3 computadores, de gamas más bajas, con tal de comparar los comportamientos del modo continuo y discreto. En el mismo Apéndice D se pueden encontrar los datos del hardware de cada uno de los computadores. Para facilidad de uso y comunicación, se han denominado

computadores A, B y C de forma ordenada de menor a mayor potencia final. Además, el computador C es menos potente que el computador principal.

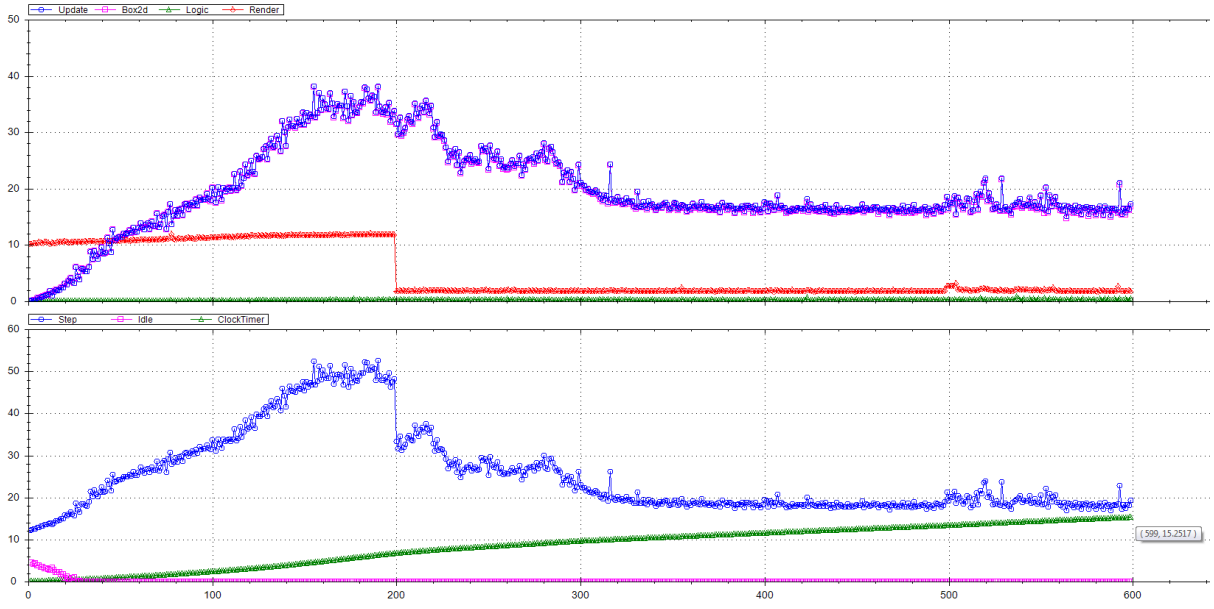
El primer paso es realizar una prueba básica y continua en los 4 computadores para poder estudiar la situación básica.

En esta prueba se dispondrá del jugador en el mapa 1. Durante 100 frames irán apareciendo objetos de 2 en 2, hasta un total de 200. Después se esperará 500 frames para que se estabilicen, y para comprobar la carga una vez estabilizados.

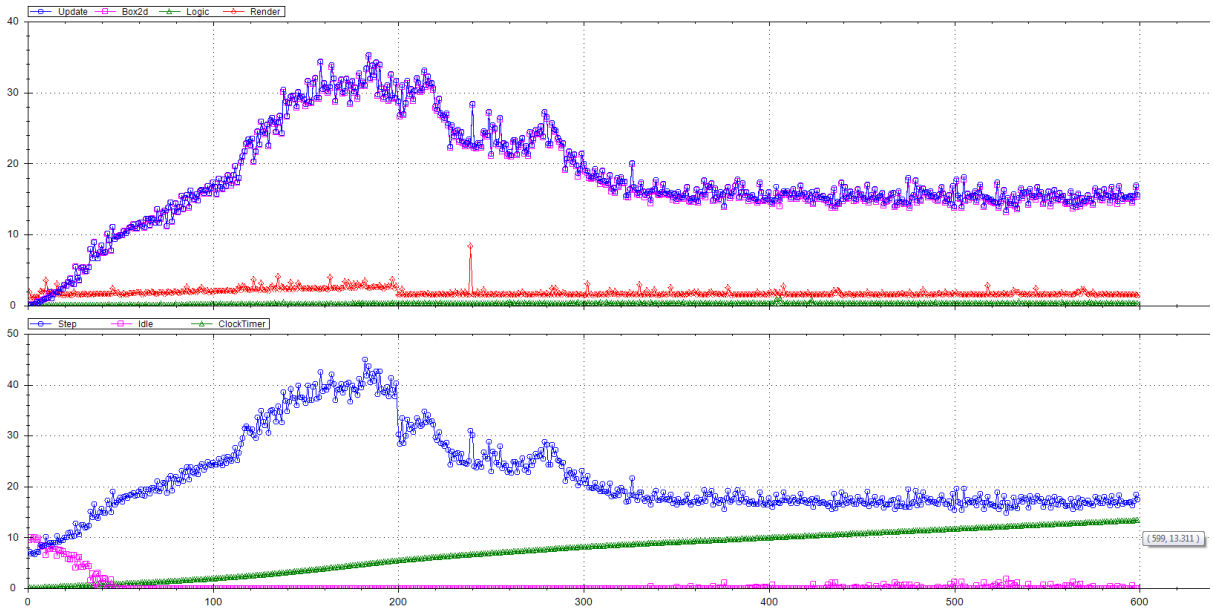
Computador A:



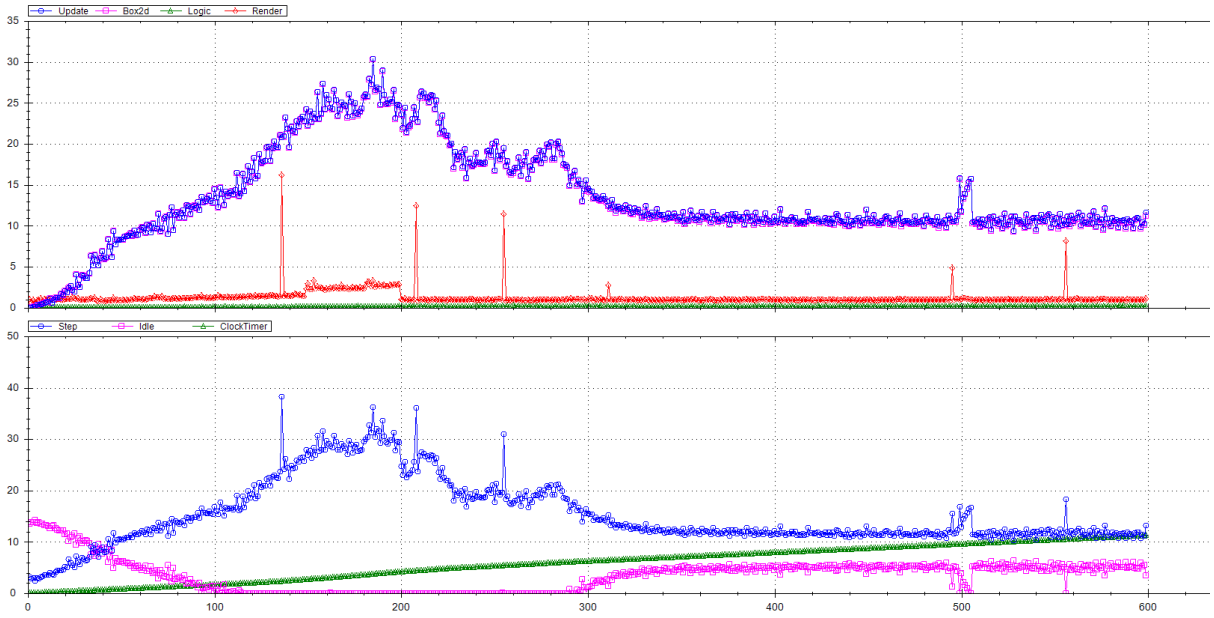
Computador B:



Computador C:

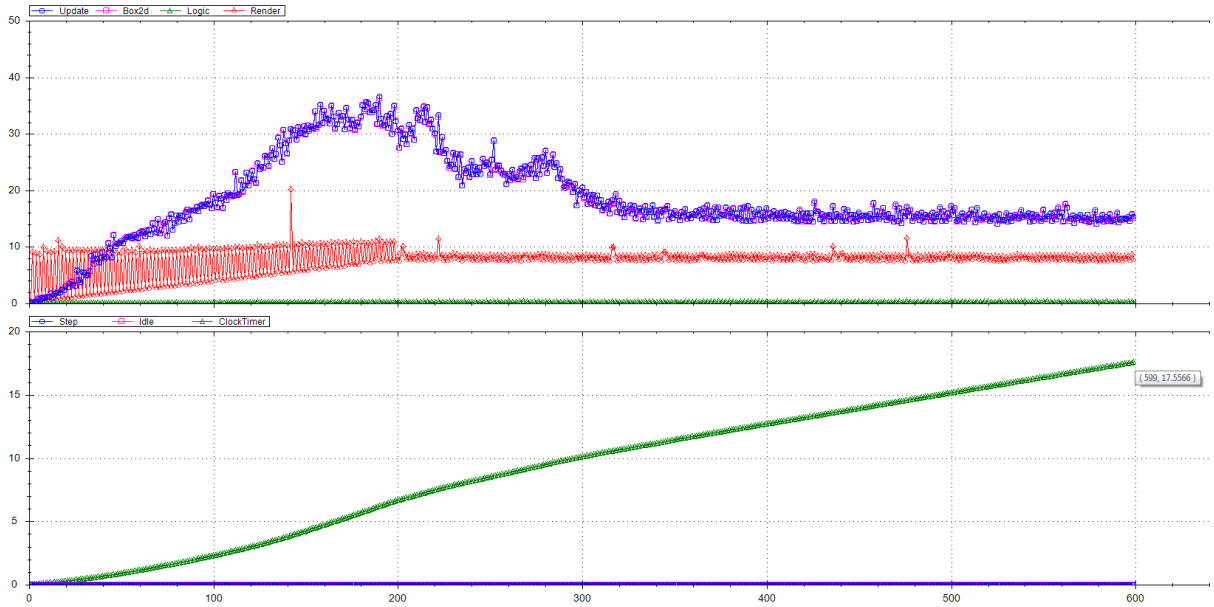


Computador principal:

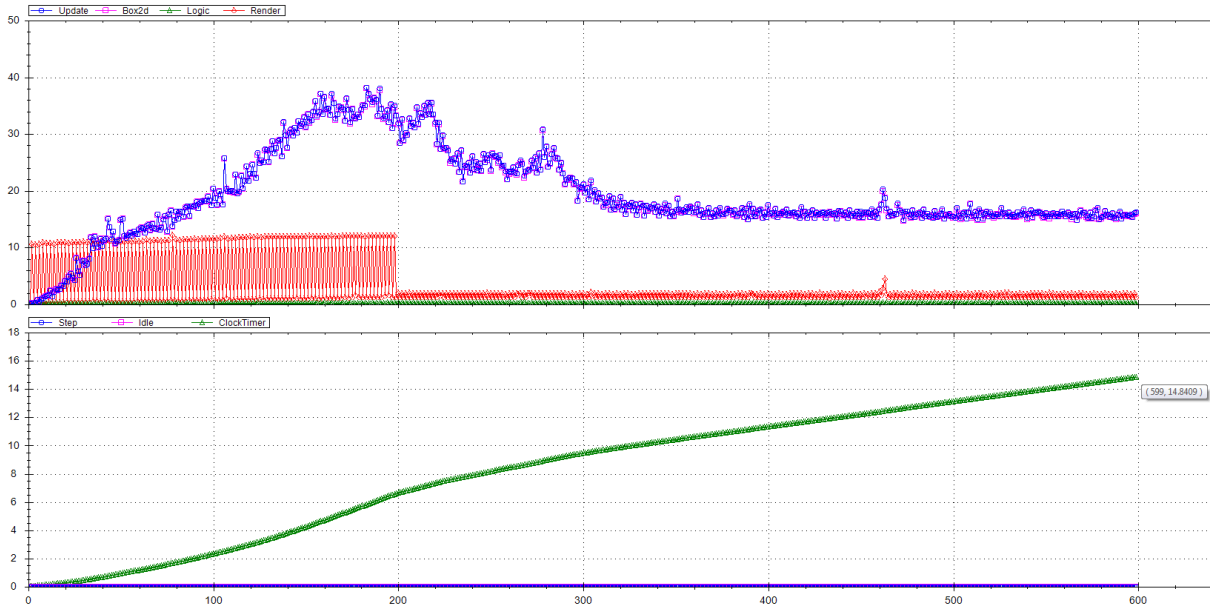


Y ahora esa misma prueba, en los mismos 4 computadores, en modo discreto con la configuración base:

Computador A:

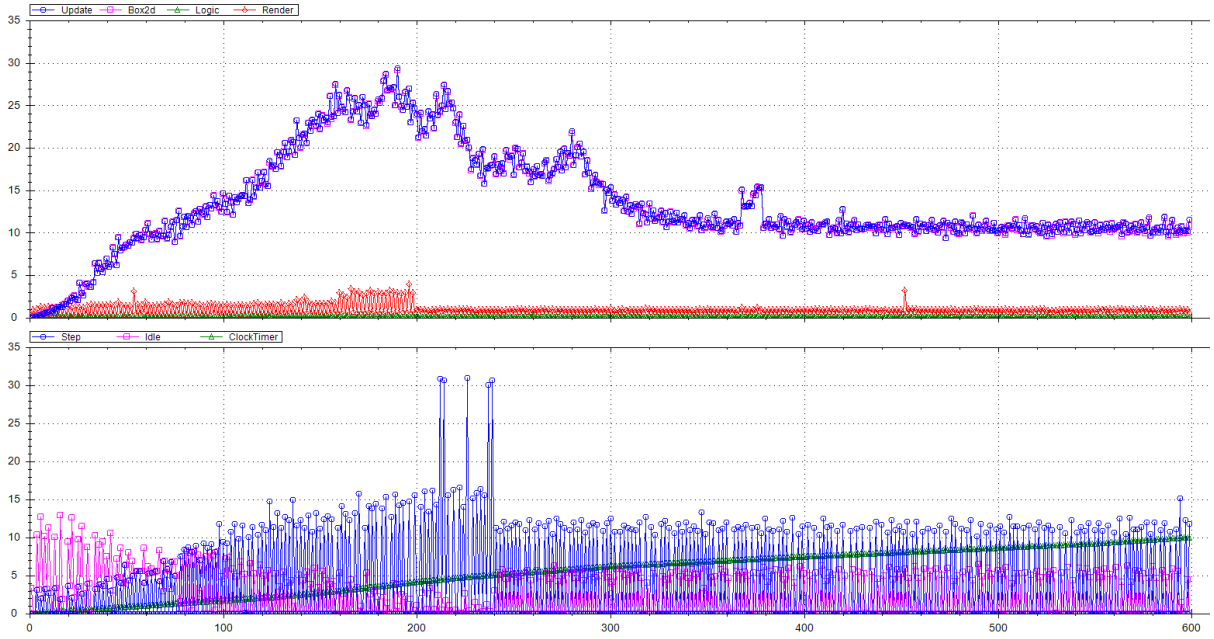


Computador B:



Computador C:

Computador principal:



Primero, el análisis y comparación de CLOCK:

	A	B	C	Principal
Continuo	**17.6122**	**15.2517**	**13.311**	**11.2313**
Discreto	**17.5566**	**14.8409**	**12.9079**	**9.984**
Diferencia	0.0556	0.4108	0.4031	1.2473

La diferencia entre la ejecución continua y discreta parece variar en función de un hardware u otro. Por suerte, la diferencia en los casos B y C es muy similar, mientras

que los valores globales respectivos son distintos, lo que puede ayudarnos a discriminar el factor.

Centrándose en principio en los computadores B y C y su similar factor de mejora en la diferencia, se encuentran las siguientes características destacables:

- Respecto a la tarjeta gráfica
    - B tiene una ATI Radeon HD2400
    - C tiene una NVIDIA GeForce GTS250
- Por lo que C tiene una tarjeta gráfica considerablemente más potente que B. A pesar de ello la diferencia entre modo discreto y continuo es igual. Eso nos transmite que la tarjeta gráfica puede no tener ninguna relación con la diferencia.
    - Aún así, el propio hecho de tener una tarjeta gráfica dedicada puede ser un factor importante. Más sobre esto adelante.
- Por otra parte, sus procesadores son prácticamente iguales.
    - B tiene un Intel Core2 Duo Processor E8200 @ 2.66GHz
    - C tiene un Intel Core2 Duo Processor E8500 @ 3.15GHz
    - En los benchmarks donde se usan ambos núcleos obtienen resultados idénticos, pero cuando solo se usa un núcleo el E8500 obtiene resultados ligeramente mejores.
    - Puede concluirse, pues, que el procesador es el factor más relevante en el factor de mejora del modo discreto sobre el continuo.
- El sistema operativo y la memoria RAM no parecen ser factores muy relevantes.
- Se ha de tener en cuenta que las tarjeta gráfica no ofrecen, por definición, mucha mejora a un juego en 2D.

Si se tienen en cuenta también los datos obtenidos y el hardware de los otros dos computadores:

- El procesador del computador principal es sustancialmente mejor en todos los aspectos, por lo que su factor de mejora en la diferencia tiene sentido que sea tan grande.
    - Por otra parte, puede que la memoria RAM también tenga que ver en la gran diferencia. Mientras que los otros computadores tienen 3 o 4GB, este tiene 8GB.
- A la hora de tener en cuenta los factores que causan una diferencia tan pequeña con el computador A se complican las cosas.
    - El procesador del computador A, un Intel Core2 Quad Q9550, es ligeramente mejor que los E8200 y el E8500 de los computadores B y C. Tanto por el tener dos núcleos más, como por el resultado en los benchmarks más importantes del mercado. En consecuencia, no se puede achacar al procesador en sí.
    - Por otra parte, es el único equipo sin tarjeta gráfica dedicada utilizado en las pruebas. Como se comentó antes, el modelo de tarjeta gráfica tiene un impacto despreciable sobre el factor de mejora. Aún así, y como queda demostrado, el mero hecho de tener una tarjeta gráfica dedicada sí tiene un impacto.

Se puede concluir que el tener una tarjeta gráfica tiene un impacto importante en el factor de mejora, pero no el modelo y su potencia. Por otra parte, el procesador es también importante. Puede que la memoria RAM también lo sea, pero con las pruebas realizadas no se puede aseverar de forma concluyente.

Respecto a los profilers en sí, exceptuando RENDER, la calidad de la ejecución ha sido igualmente equivalente a la calidad de CLOCK. Las mejoras son escalonadas en cada hardware inmediatamente superior. El computador A apenas tiene un poco de IDLE al comienzo, mientras que el computador principal tiene IDLE una vez los objetos se estabilizan. A su vez y aún más llamativo, en el modo discreto y en el computador principal el RT-DESK no llega a saturarse en absoluto.

El comportamiento de RENDER quizá sea el más llamativo, por lo que se menciona a pesar de no estar directamente relacionado con RT-DESK. Allegro tiene un factor de mejora lineal pero escalonado. Mientras que en computador A primero tiene una latencia de 10ms que tras la estabilización baja a 9ms, en el computador B baja de 10ms a 2ms. En el computador C comienza en 4ms y baja a 2ms. En el computador principal se mantiene en 2ms excepto durante el clímax de la carga física y lógica del juego, donde hace un pico de 4ms.

Este comportamiento es curioso e inherente a la implementación de Allegro, por lo que analizarlo no forma parte de los objetivos finales de este proyecto. Aún así, y puesto que el correcto funcionamiento del engine fue el objetivo de la primera parte, se ha considerado interesante resaltarlo.

Análisis de resultados y conclusiones

Visión global

Hay una serie de cuestiones que se buscaba responder con la integración del RT-DESK en el engine 2D presente. Esta serie de preguntas se corresponden al impacto de éste sobre la corrección y la calidad de su ejecución, a la corrección ante distintos sistemas de distinta potencia, y los efectos secundarios de estos, como por ejemplo impacto sobre la vida de la batería o el impacto respecto a los distintos códigos de terceros y los distintos módulos implementados.

En la presente sección se van a exponer las distintas conclusiones de las pruebas y análisis realizados en la sección anterior. Para la exposición se seguirá un orden que permita su entendimiento sin necesidad de haber estudiado en profundidad el desarrollo del proyecto y el desarrollo y análisis de las pruebas.

Se comenzará con el que ha resultado el punto más conflictivo durante el desarrollo del modo discreto: el código de terceros. Tras ello se expondrán las conclusiones del RT-DESK y la ejecución discreta respecto al resto de código. Finalmente, conclusiones generales y efectos secundarios. Como conclusión, se resumirá todo lo dicho hasta ahora. También se hará mención del comportamiento ante distintas configuraciones hardware.

Conclusiones sobre código de terceros: Box2D y Allegro

En el código de terceros no se ha integrado RT-DESK, así que las siguientes conclusiones no afectan directamente a su juicio. Se incluyen, aún así, porque el proyecto también incluye la realización del engine Zentract y porque han afectado al benchmarking de diversas formas.

Una de las características que más salta a la vista en todos los resultados obtenidos es la **linealidad y continuidad de RENDER**. El render realizado en este engine se corresponde directamente, como ya se explicó, a un wrapper de Allegro. Esto viene a significar que es puramente código de Allegro: código de terceros. Allegro utiliza para su módulo de rendering OpenGL, y sumado al hecho de que el 2D es muchísimo menos costoso de llevar a cabo que el 3D, los resultados que obtiene son plenamente satisfactorios. En el ámbito de este proyecto no es excesivamente importante puesto que es código forzosamente continuo y de terceros, pero eso no quiere decir que no se puedan obtener conclusiones de ello.

En la totalidad de las pruebas realizadas el RENDER se ha mantenido siempre estable, aunque con subidas y bajadas. Eso quiere decir que el hecho de tener un número elevado de bitmaps a dibujar por pantalla no le ha afectado ni de forma lineal, así como tampoco la variación de las posiciones de estos. Al aumentar considerablemente el número de entidades en pantalla a dibujar sí se vió afectado pero por escalones.

Por otra parte, la metodología de Box2D ha tenido el mayor impacto en el proyecto, puesto que ha creado un segundo cuello de botella a la hora de permitir la desincronización de la ejecución. Independientemente de este hecho, que una vez estudiado y asumido no resulta en más problemas, se han llegado a obtener una serie de conclusiones interesantes aunque no inherentemente relacionadas con la ejecución discreta. Es por eso que dichas conclusiones se comentan durante el análisis de las pruebas correspondientes, pero no se hará especial énfasis en ellos en la presente sección.

Respecto a su situación como cuello de botella, es esencial puntualizar que se propone como proyecto futuro la conversión de Box2D a la ejecución discreta también usando RT-DESK. Dicho proyecto resulta de un alto interés en el estudio y aplicación del RT-DESK pero se sale de los límites del presente proyecto.

Se podría haber minimizado un poco la limitación impuesta por Box2D delegando la mayor cantidad de funciones posibles de su código al código del engine Zentract. La primera de dichas delegaciones habría sido el cálculo de posición/velocidad/aceleración de cada entidad. Permitiendo a Zentract el cálculo y actualización de sendos parametros habría dotado de cierto grado de libertad a la hora de ejecutar las entidades de forma asíncrona. Aún así, el grado de libertad habría seguido siendo muy bajo puesto que la comparación dos a dos de Box2D sigue siendo continua y secuencial.

La carga de Box2D no siempre se ha visto equivalente y lineal al número de entidades en pantalla. En las distintas simulaciones con profilers se ha comprobado que cuanto más cercana a 0 es la velocidad de los objetos cuyas físicas se han de comprobar, el coste computacional se relaja, de forma que en las pruebas había una carga mayor en la primera parte, cuando las entidades se estaban creando y éstas iban cayendo y colocándose, que en la segunda parte en la cual la mayor parte de las entidades en pantalla ya están colocadas y en una posición estable. Es un dato a tener en cuenta a la hora de decidirse por el uso de esta biblioteca, y muy probablemente sea una de las principales razones por las que es la más usada hoy en día.

Conclusiones sobre código del engine Zentract

Se ha de tener en cuenta las limitaciones creadas por el código de terceros de las bibliotecas de Allegro y de Box2D, sobre todo las de este segundo. Dicho tema se ha estudiado larga y cuidadosamente a lo largo de esta memoria. Ahora lo que interesa es conocer las fronteras de dichas bibliotecas para saber dónde comienza el territorio manejable.

La mayor parte de este territorio, que muy desafortunadamente ha resultado mucho más pequeño de lo que el autor del proyecto había esperado y deseado, se corresponde con la lógica del juego.

La lógica del juego se ha encontrado limitada entre la lógica de las físicas y el renderizado. Todas las entidades debían encontrarse en el mismo punto de actualización para poder dibujarse en orden, así como para poder calcular las leyes físicas a su alrededor.

Ante un tan pequeño margen de maniobrabilidad, se ha tratado de abarcar la mayor cantidad de libertad posible. En esencia, liberando las propias entidades, y los útiles del juego (como la cámara, los botones, la actualización de la lógica de las entidades, los cambios de escena, la actualización del input y del render, entre otros). Aunque resultó ser un buen ejercicio de diseño de software, y ofrece cierto grado de mejora que se comentará más adelante, sigue dependiendo de la física y del renderizado.

Los módulos que sí se permiten tener una ejecución asíncrona son el actualizador de Render y el Input. La ejecución discreta forzó el estudio en profundidad de este segundo, resultando así en un módulo mucho más cohesivo y fiable (modo de cadena

de comando estricta). Aunque sean dos módulos pequeños y con poca carga computacional, el poder variar su configuración de tiempos en ejecución discreta ha permitido realizar un estudio detallado.

En conclusión, el código del engine en sí al terminar la primera parte del proyecto y con los añadidos y modificaciones posteriores, resulta muy satisfactorio. Cumple la lista de características que se marcó al comienzo de este proyecto, y además ha servido como banco de trabajo idóneo tanto a nivel de diseño y arquitectura de software, como para añadir los profilers y diseñar las pruebas deseadas.

Si no hubiese sido por la elección de Box2D, el proyecto se habría alargado considerablemente pues un motor de física, incluso elemental, puede llegar a ser muy costoso. Por otra parte, haber realizado un motor personalizado habría permitido llevar a cabo una implantación del RT-DESK real, completa y concluyente.

Conclusiones sobre código del engine Zentract en ejecución discreta

En resumidas cuentas, el tiempo real de ejecución mejora gracias al RT-DESK, pero las latencias de los distintos módulos en sí se mantienen iguales.

En las pruebas de CLOCK se comprobó que hay un cierto factor de mejora nada despreciable. Dichas pruebas tuvieron una duración de entre 10 y 20 segundos, y aún así dicha mejora era apreciable. En una ejecución normal de un juego, que se pueda suponer de 1 hora o incluso más, la diferencia es considerable. En ese aspecto la ejecución discreta ha sido un éxito.

Esta mejora en el tiempo real se traduce, durante la ejecución real, en que una vez el computador se satura (quedándose sin tiempo para computar dentro del intervalo predefinido para cada frame) su comportamiento no es ralentizar el juego. El RT-DESK mantiene la ejecución lo más ajustada al tiempo real que permite el computador. Así pues, en los hardware que lo permiten, se obtiene una ejecución más escalonada, donde hay frames que no se llegan a ver, pero a una velocidad mucho más cercana a la real.

Esta característica puede no ser necesariamente deseable en ciertos juegos, pero por lo general sí lo sería. En juegos online, donde han de sincronizarse varios jugadores, es sin duda una característica no deseable sino necesaria.

Por otra parte, que los módulos en sí no hayan podido mejorarse gracias a la ejecución discreta es una pérdida. A falta de inteligencia artificial y una alta carga visuales, el peso computacional de un juego desarrollado con este engine recaerá en la física y la lógica. Ambos módulos consisten en esencia en trabajar con un conjunto N de objetos que pueden interactuar o no unos con otros. Por ello, el poder optimizar y ponderar con el método de ejecución discreta estos módulos habría resultado idóneo.

En base a esta consideración se propone el primer posible proyecto futuro: Integración de RT-DESK en Box2D. Otra alternativa sería realizar un motor de física personalizado para sustituir al Box2D. Al estar realizado para el engine, cumpliría todos los requisitos necesarios.

En la sección previa se dijo que los módulos de Input y Render sí pudieron implementarse con ejecución asíncrona. Se ha aprovechado para estudiar el comportamiento de modificar la frecuencia de actualizado de ambos módulos en el resto de la ejecución. La principal conclusión alcanzada es que a mayor frecuencia de actualizado, se produce una mayor carga computacional. Lógicamente, también se produce una respuesta al input y un resultado del output más certera. Se podría ponderar la frecuencia de actualizado de dichos módulos en base al estado de la ejecución.

Se ha de saber que realizar benchmarks con profilers clásicos ha resultado complicado por la propia definición del modo de ejecución discreto. Se ha analizado en detalle este tema en la sección “Características de las pruebas de profiling en una ejecución discreta”. En base a ello se propone un nuevo trabajo futuro consistente en un sistema de profiling adaptado al RT-DESK. El método actual no nos permite saber en qué momento se ha ejecutado cierto frame de cierto módulo (se recuerda que la ejecución no es síncrona), y a la vez conocer el tiempo que ha tardado en ejecutarse.

Conclusiones sobre el hardware

Se pudieron comprobar ciertas características del hardware sobre el engine que vale la pena conocer. Aún así y antes de empezar, un trabajo de benchmark más exhaustivo y completo podría aportar aún más información valiosa.

El factor más importante estudiado en las pruebas de hardware ha sido el factor de mejora de la ejecución discreta respecto a la continua. Esto viene a ser la diferencia obtenida en el tiempo real en una ejecución respecto a la otra.

Primero, respecto a la RAM y al sistema operativo (entre Microsoft Windows 7 y 8) no ha habido ninguna diferencia notable. Por otra parte, sí las han habido con el procesador y la tarjeta gráfica.

El procesador es importante en el factor de mejora de la ejecución discreta respecto a la continua. Esto quiere decir que, a mejor procesador, la mejora entre la ejecución discreta y la continua es mayor. Es importante notar que esto afecta tan solo a un núcleo. La ejecución no tiene hilos ni soporta multinúcleo. Se propone una aplicación de distintos RT-DESK en distintos núcleos como trabajo futuro.

Por otra parte, la tarjeta gráfica es un factor significativo, pero no debido a su potencia sino a su mera disponibilidad. Cuando no se ha dispuesto de tarjeta gráfica (recayendo

el cálculo gráfico en el propio procesador) se ha obtenido una mejora despreciable entre el modo continuo y el discreto. Por otra parte, el propio hecho de poseer tarjeta gráfica independiente de su potencia, ha permitido que existan mejoras en el factor.

Respecto a la correctitud y fiabilidad de la propia ejecución en modo continuo y en modo discreto, la potencia del hardware ha incidido de forma directa. Esto era esperable, y no es en ningún modo una sorpresa: A mayor potencia de cálculo, mejor ejecución. En cualquier caso, y puesto que el coste añadido era nulo (sirven los mismos datos y gráficas obtenidos para comprobar el factor de mejora), se decidió por comprobarlo.

Trabajos futuros

Ampliación o mejora del engine Zentract

Se indicó previamente un número de objetivos para el presente proyecto. Ahora me atrevo a añadir que estaba ocultando información: Durante todo el desarrollo ha habido un objetivo más, subyacente a todo el proceso: Permitir que en un futuro se puedan realizar modificaciones, añadidos o mejoras. El engine realizado por el alumno debe poder ser leído, estudiado, utilizado y ampliado por cualquier futuro alumno de la UPV interesado en el desarrollo de videojuegos.

Si el proyecto en cuestión no va a tener nada que ver con RT-Desk:
El estado del engine al completar el “desarrollo principal” (antes de empezar la segunda parte del proyecto) es el estado idóneo para que se lleve a cabo.

Si el proyecto en cuestión va a estar relacionado con RT-Desk:
Dependiendo de los objetivos, deberá partir o de las pruebas con RT-Desk, o del “desarrollo final” en el que se encuentran ambos modos de ejecución.

Además, el engine es fácilmente ampliable. Al ser modular y al haber seguido las buenas prácticas del diseño de software, una vez comprendido el funcionamiento de un módulo de forma interna y externa, se pueden proyectar cambios sobre él sin la necesidad de conocer como funciona todo el resto del engine.

Es importante también tener en cuenta la posibilidad de cambiar el Box2D por un motor de física personalizado que cumpla todos los requisitos de la ejecución discreta. Como alternativa a esta ampliación, se propone el siguiente trabajo futuro.

Integración de RT-DESK en Box2D

El cuello de botella más significativo a la hora de implantar la ejecución discreta ha sido la biblioteca de terceros Box2D. Al imponer una metodología continua muy estricta, ha obligado a ejecutar secuencialmente dos de los puntos más críticos a la optimización por ejecución discreta del engine: la inteligencia artificial y la lógica de entidades. A ello se suma el propio cálculo de las físicas como un tercer bloque de ejecución que se queda sin optimización.

En consecuencia, la integración del RT-DESK, incluso más allá de como un proyecto de por sí interesante tanto desde el punto de vista del RT-DESK y el del Box2D, resultaría en una serie de mejoras para Zentract que merecerían el estudio.

Dicha integración, al igual que el presente proyecto, no sería directa e instantánea, sino que requiere y permite una integración profunda en la propia metodología del Box2D. Es por eso que se plantea como posible futuro proyecto.

Profiling con anclaje para ejecución asíncrona

Uno de los objetivos del presente proyecto eran las comparativas entre la ejecución continua y discreta. Para ello la principal herramienta han sido profilers basados en temporizadores de tiempo real. Dicha herramienta, aunque ha sido útil y ha permitido alcanzar las conclusiones que se buscan, ha resultado incompleta. El proyecto futuro que se propone sería, justamente, la herramienta de profiling idónea para la toma de datos en el modo de ejecución discreto.

La principal característica del modo de ejecución discreto es que permite y favorece la ejecución asíncrona: Cada módulo u objeto puede llegar a ejecutarse a su propio ritmo. Como ya se ha comentó de forma teórica en la sección “Requisitos especiales de RT-DESK: Profiling descentralizado” y se analizó de forma empírica en “Características de las pruebas de profiling en una ejecución discreta”, trabajar con los profilers típicos en estos términos requiere de un trabajo extra para poder corregir y adaptar los datos obtenidos.

Este nuevo profiler permitiría, por ejemplo, asignar un módulo u objeto como “ancla”, cuya frecuencia de actualización serviría como base para comparar las frecuencias de actualización del resto. De esa forma, se podrían aunar en un solo conjunto de datos: las diferencias relativas entre las frecuencias de actualización y los tiempos de ejecución de cada una de dichas actualizaciones, por módulo y por objeto.

Con estos datos se podrían comparar de una forma directa y cuantitativa los datos obtenidos de los benchmarks, y realizar con ellos estudios igual de profundos que los realizados en el presente proyecto sin la necesidad de partirlos en distintas pruebas muy distintas.

Agradecimientos

A Ramón Mollá tanto por darme la libertad de poder realizar el engine tal y como yo tenía planeado realizarlo, como por conducirme y aconsejarme para dar siempre lo mejor de mi en todos sus aspectos. Además, por permitirme el aprendizaje y utilización del kernel RT-Desk, con el cual he adquirido una serie de conocimientos (tanto respecto a él y a la ejecución discreta, como respecto a la integración de un módulo de terceros tan innovativo en mi engine, como de técnicas de benchmarking y profiling) que sin duda me van a resultar de un valor incalculable.

A mis padres, mi hermano y a Dana por el apoyo moral durante todos estos meses de arduo trabajo.

Apéndice A: Documentación del engine Zentract

Adjunto a esta memoria se incluye una serie de ficheros con la documentación del engine.

A parte de estos existe mucha más documentación específica de los distintos proyectos, así como los comentarios dentro del propio código, que se consideran indispensables para la comprensión profunda de las distintas partes del código. Todo esto se puede encontrar en las partes correspondientes de la estructura de carpetas, donde cualquier interesado lo buscará.

Otros documentos de importancia son la guía de estilo del código (coding guidelines) y dicha documentación específica, ambos repartidos por la estructura de carpetas.

Apéndice B: Resultados de las pruebas

Dentro del código del proyecto adjunto se podrán encontrar los resultados en crudo (raw) de las pruebas realizadas. Estos se encuentran dentro de las carpetas del proyecto ZentractFinal con los nombres serializados de TestXXXX. En cada una de las carpetas de las pruebas se podrá encontrar un fichero “info.txt” el cual detalla tanto la configuración del Engine, como la configuración del Game, así como los objetivos a analizar con los datos que se obtengan.

El formato de los datos CSV se ha detallado en la sección “Formato y especificaciones”, y para realizar las gráficas se ha utilizado el software libre DatPlot (http://www.datplot.com/) por su comodidad, velocidad y adecuación.

Si se desea obtener información sobre cómo configurar el Engine y el Game de Zentract para las pruebas, sobre cómo modificar los mapas o sobre cómo configurar los profilers, se deberá acudir a la documentación correspondiente dentro de la carpeta "ZentractFinal\docs\", así como el fichero "ZentractFinal\src\engine\settings.h".

Apéndice C: Datos de los computadores utilizados en las pruebas

Computador principal:

Procesador: Intel Core i5-3570 @ 3.4GHz
Tarjeta gráfica: NVIDIA GeForce GTX660
RAM: 8GB
Sistema operativo: Windows 7 SP1 x64

Computador A:

Procesador: Intel Core2 Quad Q9550 @ 2.83GHz
Tarjeta gráfica: Intel Q45 Express (integrada)
RAM: 4GB
Sistema operativo: Windows 7 SP1 x64

Computador B:

Procesador: Intel Core2 Duo Processor E8200 @ 2.66GHz
Tarjeta gráfica: ATI Radeon HD2400
RAM: 3GB
Sistema operativo: Windows 7 SP1 x86

Computador C:

Procesador: Intel Core2 Duo Processor E8500 @ 3.15GHz
Tarjeta gráfica: NVIDIA GeForce GTS250
RAM: 4GB
Sistema operativo: Windows 8 SP1 x64

Apéndice E: Listados de commits

Puesto que estos listados se han obtenido de forma directa de la lista de commits, es importante notar que siguen un orden de abajo hacia arriba.

Primera parte ~ Primer prototipo

Cleanup commit.	2013-07-09
Solved problem with contact sensor. Added map.	2013-07-09
Added multiple barrels, both the singleton and the EntityManager support. The foot sensor	2013-07-03
Player gravity implemented. Something is wrong with sensor contact detection in the pointy	2013-06-25
Multiple multi-vertex polygons implemented.	2013-06-25
One single multi-vertex polygon possible.	2013-06-25
Map with one polygon implemented! TODO: Make more than one possible poly per map.	2013-06-21
Prepared to start working on MapLoader.	2013-06-21
Added player and entitymanager and camera and input. Corrected constants.	2013-06-20
Continued porting project from SuperSlidingLeafboy. Updated Box2D to VS2012's toolset.	2013-06-19
Created project	2013-06-18

Primera parte ~ Segundo prototipo

Removed logo and publish.	2013-10-15
Final commit: Removed Boost's dynamic-link library, it is not needed.	2013-10-15
Final commit: Added MapEditor. Updated Notes.txt to include it.	2013-10-15
Final commit: Cleaned folder tree. Changed everything to /MT Release-only. Added	2013-10-15
Added Finish ItemEntity, and integrated. Added Globals->UnlockedLevels, and integrated.	2013-08-27
Cleaned up: Allegro dependencies now only in Render and Sprite. Also, Scenes are	2013-08-22
Cleaned: sensor handling.	2013-08-22
Changed sensors handling to a single array.	2013-08-22
Changed Level to Map. Cleaned up a bit Map and Poly dependencies.	2013-08-20
Templated "ALLEGRO_BITMAP*" to "image" in Sprite.	2013-08-19
Splitted BarrelEntity into ObjectEntity and ItemEntity.	2013-08-18
Cleanup: Constants and Globals cleaned and ordered.	2013-08-18
Cleanup: Includes and dependencies. But still FAR, FAR AWAY from perfect. Lot of bad	2013-08-18
Added LevelSelectScene and InstructionsScene, and integrated the first.	2013-08-18
Applied polymorphism to Entity. Now I have 3 kinds of entity: Object, Player and Enemy.	2013-08-18
Now entities kill the Player. Just testing.	2013-08-16
Fixed TODO list.	2013-08-16
Continued fixing contacts. Cleaned ContactListener.	2013-08-16
Added bottom-diagonal contacts to Entity. Continued fixing sizes and physical variables.	2013-08-15
Continued implementing BitMasks. Fixed Entity and Level Box2D sizes.	2013-08-15
Started implementing BitMasks to identify fixtures. Stable but unfinished!!	2013-08-15
Implemented Entity->JumpTimer. Somehow not working perfectly	2013-08-15
Separated Input and AI calling inside Entity. Moved Move into Entity, and added	2013-08-14
Added serialization to LoadLevel. Levels can finally be loaded from files.	2013-08-14
Fixed and improved level loading. Now the level is loaded with LevelLoader and created	2013-08-13
Implemented reset and resume game. LevelManager is now a class.	2013-08-13
Forgot LoadingScene images.	2013-08-13
Added and integrated Game (but Level still not inside it). Added and integrated	2013-08-13
Created Game. Started implementing it.	2013-08-10
Forgot to commit DeathScene.	2013-08-04

Added world boundary, DeathScene and the functionality to ContactListener.	2013-08-04
Added ContactListener and integrated. Improved conditional animation to be like in Sonic.	2013-08-04
Added Poly class and integrated it into Level. Improved conditional animation (not so	2013-08-04
Added (even more primitive) conditional animation.	2013-08-03
Added (still primitive) animation, with a Sprite class.	2013-08-03
Modularized Level from LevelManager. Added parallax rendering.	2013-08-03
Added Window. Integrated it.	2013-08-02
Added MapLoader and other bunch of stuff. Added first testing executable.	2013-08-02
Added an array of entities, and its management. Included Stack.h although still not used.	2013-08-02
Added EntityManager and Entity, and integrated Box2D to them and GameScene.	2013-08-02
Added Button. Cleaned up everything.	2013-08-02
Added mouse to Input and Button. Button isn't tested.	2013-08-01
Added base Input.	2013-08-01
Added base Rendering.	2013-08-01
Changed singleton State management to proper Scene namespace management.	2013-08-01
Integrated Box2D to the EntityManager and Entity.	2013-07-23
Added basic EntityManager and entities.	2013-07-23
State Manager finished.	2013-07-20
Half assed State Manager. TODO1: Change state needs polish. TODO2: Resolve	2013-07-20

Segunda parte ~ Desarrollo RT-DESK

RTDtest7: Enhanced Input: Finished discrete mode.	2013-11-26
RTDtest7: Enhanced Input, created UserInput for customized key mapping. Refactoring	2013-11-26
RTDtest7: Moved all game-related render stuff to Game.	2013-11-20
RTDtest7: Solved all TODO1.	2013-11-15
RTDtest7: Many TODO1s solved. A lot of multi-breaking little mistakes scattered around	2013-11-15
RTDtest7: Solved all TODO2: Discrete self-messaging that were still made with direct	2013-11-13
RTDtest7: Solved bug. Updating discrete up to continuous. A huge amount of work is done.	2013-11-12
[WIP] RTDtest7:Updating discrete up to continuous. All the code is done, only one hidden	2013-11-11
RTDtest: Updated .hgignore.	2013-11-11
RTDtest7: Added pause to Game. Moved Ball physics to Game. Couldn't move paddle	2013-11-11
RTDtest7: Added AssetManager. Added more assets and content. Added player 1 and 2.	2013-11-10
RTDtest7: Added more scenes. Improved architecture: Main->SceneManager->Game-	2013-11-09
RTDtest7: Added SceneManager. By the way, Discrete mode is HEAVILY OUTDATED. I'm	2013-11-07
RTDtest7: Begun test. Gigantic changes to properly do d/c modes, specially in Entities	2013-11-07
RTDtest6: Begun test. You can now pick Discrete or Continuous in RTD/Settings.h.	2013-11-02
RTDtest5: Added PlayScene. Moved RTDESK and timer management to MessageSystem.	2013-11-02
RTDtest5: Added Rect class. Integrated in Entity. This adds intersection.	2013-11-01
RTDtest5: Differentiated Paddle from Ball from Entity.	2013-10-28
RTDtest5: Begun test. Created MessageSystem to contain RTDESK.	2013-10-28
RTDtest4: Entity from RTDtest2 integrated. Now it works with msgs.	2013-10-26
RTDtest4: Skelleton of Render integrated.	2013-10-26
RTDtest4: Basic Input working.	2013-10-26
Finished RTDtest3, it is working.	2013-10-25
UNSTABLE: Converted RTDtest3 into a stripped down class/namespace test area.	2013-10-18
Cleaning up.	2013-10-15
First commit. RTDtests 1 and 2 finished, and begun with 3.	2013-10-12

Segunda parte ~ Desarrollo Final

Added Engine and RTDESK documentation.	2014-01-08
Finished adding doxygen comments to Engine. Also solved an inconsistency problem with	2014-01-07
Solved a little problem with consts in Engine\Data\Rect.h.	2014-01-05
WIP: Documenting. Added 2 architecture pictures and half of the Engine is doxygenated.	2014-01-05
Solved a few problems related to Input's refactorization.	2014-01-03
Added Input refactorization to discrete mode.	2014-01-02
Input refactorization finished. Only cleanup is needed. Now Input is the one that calls Game	2014-01-02
WIP: Refactorizing Input so the command chain begins in UserInput.	2014-01-02
Updated doxygen conf to new structure. A few other little and non.important changes.	2013-12-18
Solved many more problems.	2013-12-17
Solved many bad modularity issues. Specially important was the EntityManager one: Now	2013-12-10
Many more method calls changed to msgs. Removing entities and their bodies problems	2013-12-10
Sprite is now a Messager, and works in discrete. Yet its update is not working. Many	2013-12-10
Discrete mode ALMOST up to date. It's pretty buggy and messy, but it works. Still many	2013-12-09
Finally, the levels are loaded correctly. There's a bug when I change finish position tho.	2013-12-08
Level loading is now done with my Parser wrapper of rapidjson.	2013-12-07
MapEditor: Outputting levels using my Parser. That's	2013-12-03
Fixed animation handling. Now everything is loaded from data.json, and the info is in	2013-12-02
WIP: Finished integrating ZentractP2's entities and scenes. Only RTDesk mode to go.	2013-12-02
WIP: Integrating ZentractP2's entities, assets, window and scenes. Hopefully second to	2013-12-01
WIP: Integrating ZentractP2's Entity, entities and EntityManager	2013-11-30
WIP: Integrating ZentractP2's game and physics.	2013-11-30
WIP: Integrating ZentractP2's scenes.	2013-11-30
Added Data\Button. Copied code from ZentractP2: MenuScene and PlayScene. Working	2013-11-27
Solved the HUGE modular inconsistency in SceneManager about switching to scenes that	2013-11-27
Two big things: Discrete mode is totally up to date. And now the msgTypes are distributed	2013-11-27
This is a wonderful commit: Added Data structures. Added the whole Scene module from	2013-11-26
Separated base Input from customized UserInput, dependant on the project and not the	2013-11-26
Ignore filefor Mercurial.	2013-11-23
Forgot to commit project.	2013-11-23
Separated Game and Engine. Now Engine builds a static library, and Game uses it.	2013-11-23
Initial commit.	2013-10-15

Tercera parte ~ Desarrollo de los profilers

Added LOGIC profiler info to documentation.	2014-02-17
Render in discrete mode now works asynchronously.	2014-02-17
Added and modified a few documents and arch pictures.	2014-02-12
Added LOGIC Profiler to code. Corrected some profilers, but there's still few mistakes	2014-02-07
Added baseFPS and box2d step dynamic calculation in continuous main's main loop. It	2014-02-05
Box2d now working at 60FPS.	2014-02-05
Now the maps have 600 object entities.	2014-02-05
Modified maps for benchmarking. Added 300 entities. Begun testing benchmarking.	2014-02-04
Added entity creating and loading to MapEditor and LevelLoader.	2014-02-04
Modified map design for proper benchmarking. Entities remain.	2014-02-01
Further improved timers.	2014-02-01
Added the profilers. Also, added preprocessor directives settings.	2014-02-01
Added more preprocessor directives settings, and moved the settings file to <Engine>.	2014-02-01
Switched continuous mode timer to UPV's HRTTimer. Again, without using the manager.	2014-02-01
New UPV's Timer added. Switched MessageSystem's HRTTimerManager for HRTTimer.	2014-02-01
WIP: Added Profiler namespace, and integrated into PlayScene to test.	2014-01-13

Apéndice F: Bibliografía

Referencia y documentación de C++
Para el correcto uso de C++ se ha utilizado de una forma constante, como no, la siguiente web de referencia: http://en.cppreference.com/w/cpp y la siguiente web como documentación: http://www.cplusplus.com/. Otra web que he usado como referencia ha sido la de microsoft http://msdn.microsoft.com/en-us/library/ debido a que el compilador de Visual Studio 2012 fuerza como obsoletos métodos de la biblioteca estándar de C++. Aún así, provée mucho material de valor en general.

Segundo a estas dos páginas webs ha sido: http://www.parashift.com/c++-faq-lite/index.html donde se habla con precisión y exactitud sobre buenas prácticas en C++.

Una de las buenas prácticas llevadas a cabo con cuidado ha sido el const correctness:
http://en.wikipedia.org/wiki/Const-correctness
http://www.parashift.com/c++-faq/const-correctness.html En la web previamente mencionada.

Como referencia para los templates, que he aprendido durante la realización de este engine, ha resultado de enorme ayuda el siguiente artículo:

http://www.codeproject.com/Articles/257589/An-Idiots-Guide-to-Cplusplus-Templates-Part-1

También ha sido de ayuda en ciertas ocasiones el conocer en detalle el funcionamiento de C++ una vez compilado a ensamblador. La mejor fuente de esta información (a parte de la asignatura PDL de la UPV): http://www.altdevblogaday.com/author/alex-darby/

Como base para mis directrices de programación (coding guideline) partí de http://www.yolinux.com/TUTORIALS/LinuxTutorialC++CodingStyle.html y en menor medida de las antiguas directrices de John Carmack, cuando empezó a utilizar C++ ftp://ftp.idsoftware.com/idstuff/doom3/source/CodeStyleConventions.doc.
En ninguno de los dos casos utilicé las directrices de forma directa, sino que me basé en sus argumentos y recomendaciones. En el primero se incluye la forma en la que he utilizado la documentación de doxygen.