Universidad San Jorge

Escuela de Arquitectura y Tecnología

Grado en Ingeniería Informática

Proyecto Final

Generación de contenido procedural en videojuegos para casos reales

Autor del proyecto: Javier Verón Mérida

Director del proyecto: Carlos Cetina Englada

Zaragoza, 09 de junio de 2021

Este trabajo constituye parte de mi candidatura para la obtención del título de Graduado en Ingeniería Informática por la Universidad San Jorge y no ha sido entregado previamente (o simultáneamente) para la obtención de cualquier otro título.

Este documento es el resultado de mi propio trabajo, excepto donde de otra manera esté indicado y referido.

Doy mi consentimiento para que se archive este trabajo en la biblioteca universitaria de Universidad San Jorge, donde se puede facilitar su consulta.

Firma Fecha

09/06/2021

Dedicatoria y Agradecimiento

Gracias a mi familia por todo su apoyo. Sin ellos, nada de lo que he conseguido sería posible.

Gracias a Jorge Chueca, Rodrigo Casamayor, Pablo Artiga, Enrique Martínez, Daniel Blasco, y Jaime Font por la ayuda que me han ofrecido en la realización de este trabajo. Han hecho que todo sea mucho más accesible y viera el TFG como algo abarcable desde el primer momento.

Gracias a la Universidad San Jorge por ayudarme en todo lo que he solicitado, tanto para este trabajo como durante estos últimos 5 años; en especial a mi tutor del TFG, Carlos Cetina; y a mi tutora a lo largo de mi estancia en la Universidad, África Domingo.

Gracias, por último, a todos los amigos que he hecho entre mis compañeros del grado por haber estado ahí para ayudarnos entre todos desde el primer curso hasta el último.

Tabla de contenido

Resumen

Diseño, desarrollo, y análisis de un visualizador de modelos realizados conforme al metamodelo *SDML* creado por el Grupo de Investigación SVIT utilizando el motor de videojuegos comercial *Unity*.

Los principales problemas enfrentados son la lectura del archivo con el modelo, las representaciones lógica y visual de las diferentes partes en el motor, y la exportación del proyecto tanto para la plataforma PC como para la plataforma WebGL.

Para comprobar el correcto funcionamiento del visualizador, 4 compañeros han realizado un experimento con 32 personas para probar que puede mostrar diversos modelos. Los resultados son tan satisfactorios que actualmente el visualizador se encuentra disponible en la página web del Grupo de Investigación SVIT para mostrar modelos empleados en futuros trabajos de investigación.

Abstract

Design, development, and analysis of a visualizer for models made according to the *SDML* metamodel created by SVIT Research Group using the commercial videogames engine *Unity*.

The main problems faced are the reading of the model file, the logical and visual representations of the different parts in the engine, and the build of the project for both PC and WebGL platforms.

In order to make sure that the visualizer works properly, 4 partners have performed an experiment with 32 people for evidencing that it can show various models. The results are so satisfactory that the visualizer is currently available on the SVIT Research Group website to show models used in future research work.

1.1. Palabras clave

Metamodelo, modelado, visualizador, jefe (ámbito de los videojuegos), cascos, enlaces, armas, análisis, lector, lenguaje de etiquetas

2. Introducción

Kromaia (Kraken Empire, 2014) es un videojuego en el que controlas una nave espacial capaz de volar y disparar en tres dimensiones. Se lanzó para PC y PlayStation 4 (Sony Interactive Entertainment, 2013) y fue traducido a 8 idiomas diferentes.

Ilustración 1: Pelea contra el jefe "Vermis" en el juego Kromaia. Fuente: Propia.

En este juego te enfrentas a diferentes jefes[1], formados por: cascos, enlaces entre los cascos, diferentes armas, puntos débiles, y módulos de IA[2].

Según podemos ver en el vídeo "*Model Driven Engineering and Video Game Content*" (SVIT Research Group, 2020), la tecnología de *Kromaia* (Kraken Empire, 2014) permite elegir cómo llevar a cabo la creación de los jefes entre crearlos con un lenguaje de modelado o con un lenguaje de programación. Si se crean con el lenguaje de modelado se verán luego sometidos a una interpretación que los transforme al lenguaje de programación; a objetos en tiempo de ejecución. Hasta cierto punto, esa interpretación es lo que será abordado en este proyecto.

Los modelos son el artefacto principal en ingeniería[3], y son capaces de conformar metamodelos[4]. Estos metamodelos son los que se emplean en *Kromaia* (Kraken Empire, 2014) para su contenido.

1 En el contexto de los videojuegos, un jefe se refiere a un desafío de dificultad superior que se presenta al jugador antes de un cambio de escenario/fase del juego.

2 Inteligencia Artificial.

3 Principales de la ingeniería basada en modelos.

4 Los modelos son conforme a metamodelos.

En concreto, utiliza *SDML*: "*SDML* es un lenguaje de modelado que tiene conceptos como cascos (*hulls*), enlaces (*links*), puntos débiles (*weak points*), armas (*weapons*), y componentes de IA" (Blasco, Font, Zamorano, & Cetina, 2021). Los cascos en el metamodelo representan la base para todos los cascos que el jefe tiene en el juego. Los enlaces entre los cascos se representan también en el metamodelo, habiendo varios tipos como fijos, cuerdas, movibles, o telas. De igual forma, todos los demás componentes que definan al jefe se ven representados en el metamodelo (armas, IAs, etc.).

En cada elemento que forma un modelo *SDML* se tienen todos los parámetros que se puedan considerar luego para la generación de cada parte en el juego, como; en el caso de los cascos; el modelo de colisiones, la masa, la dirección, la posición, la escala, etc.

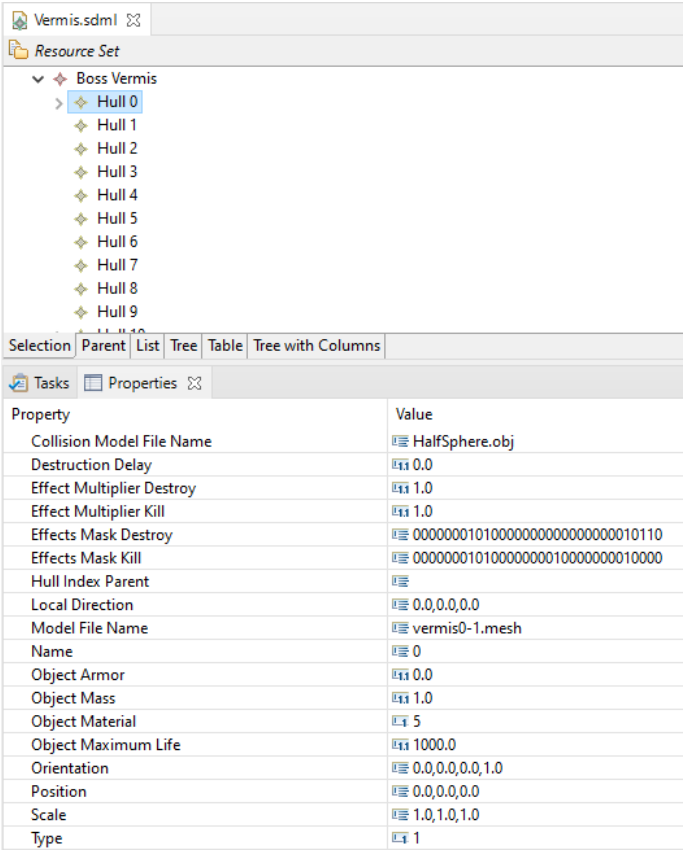

Ilustración 2: Ejemplo de SDML en una herramienta desarrollada por SVIT en Eclipse. Fuente: Propia.

Esto permite alterar cualquier parámetro de cualquier parte de los jefes de una forma muy sencilla. Si, por ejemplo, quisiéramos aumentar el tamaño de la cabeza de un jefe al triple, sería tan sencillo como modificar su escala al triple en el archivo *SDML*.

Sin embargo, con todas las ventajas que supone el empleo de la ingeniería basada en modelos, no existe nada que permita ver modelos de ingeniería basada en modelos en un motor de videojuegos.

Kromaia (Kraken Empire, 2014) utiliza un motor propietario y este trabajo se encarga de dar el salto a un motor popular, por lo que se encarga de leer los modelos de los jefes en lenguaje *SDML* e interpretarlos en el motor *Unity* (Unity Technologies, 2005-2021), para poder visualizarlos desde fuera del propio juego *Kromaia* (Kraken Empire, 2014). Será mucho más cómodo ver el modelo aislado, además de permitirnos, de este modo, procurar un acceso más fácil mediante la web.

En las ilustraciones 3 y 4 podemos ver un ejemplo de esto y de cómo, modificando un parámetro en el modelo *SDML*, se puede ver afectada la representación gráfica:

Ilustración 3: Representación del jefe "Vermis" en el programa desarrollado en este trabajo. Fuente: Propia.

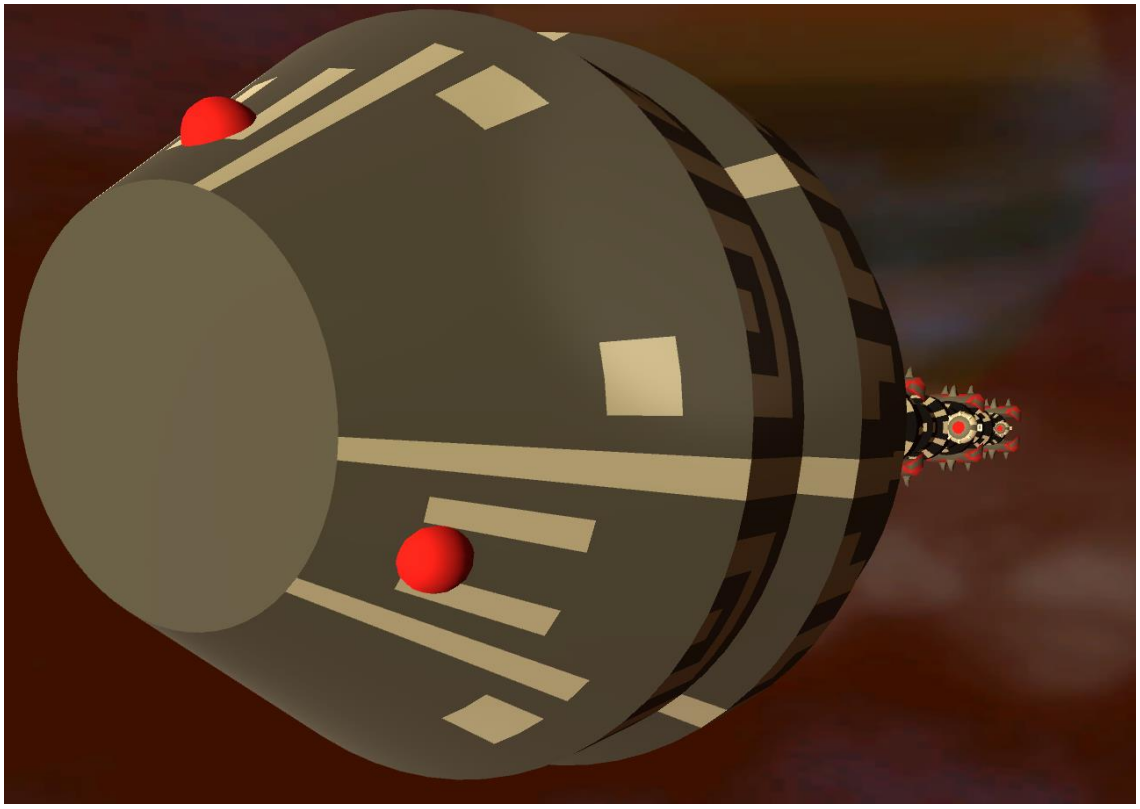

Ilustración 4: Representación del jefe "Vermis" con la escala de la cabeza modificada al triple en el programa desarrollado en este trabajo. Fuente: Propia.

3. Estado del arte

3.1. Trabajo Previo

En el desarrollo de videojuegos, la diversidad de plataformas disponibles es un problema con el que hay que lidiar. Esto es lo que ha motivado a la mayor parte de los trabajos de investigación que combinan modelos software con el mundo de los videojuegos, ya que uno de los beneficios potenciales de emplear modelos como el principal artefacto para desarrollo de software es la independencia de plataforma.

Ampatzoglou y Stamelos, en su encuesta de 2010 *Software Engineering Research for Computer Games* (Ampatzoglou & Stamelos, 2010) identificaron sólo un trabajo que aplicara desarrollo impulsado por modelos a los videojuegos (Montero Reyno & Á Carsí Cubel, 2009). Este trabajo acuñó el término "*Model-Driven Game Development*"; desarrollo de juegos impulsados por modelos; y presentó un primer enfoque para un juego de plataformas 2D[5] prototipando a través de desarrollo de juegos impulsados por modelos; en concreto, empleando clases UML y diagramas de estado que se extendieron con estereotipos[6], y una transformación de modelo a código para generar código en lenguaje C++.

La investigación de Núñez et al. (Núñez Valdez, García Díaz, Cueva Lovelle, Sáez, & González Crespo, 2017), (Núñez Valdéz, Sanjuán Martínez, Pelayo García-Bustelo, Cueva Lovelle, & Infante Hernández, 2013) presenta aproximaciones impulsadas por modelos destinadas a minimizar errores, tiempo, y coste en el desarrollo de un video juego multi plataforma. El trabajo en *A model-driven approach to generate and deploy videogames on multiple platforms* (Núñez Valdez, García Díaz, Cueva Lovelle, Sáez, & González Crespo, 2017) propone un DSL[7], llamado Gade4all, y se centra en juegos orientados a móvil y tabletas.

Solís-Martínez et al. (Solís-Martínez, Pascual Espada, García-Menéndez, Pelayo G-Bustelo, & Cueva Lovelle, 2015) proponen el uso de modelos de proceso de negocio como lenguaje de modelado para videojuegos. Más en concreto, se centran en la lógica tras los bucles de juego en juegos móviles.

5 Los juegos de plataformas son un género de videojuegos en los que el personaje principal salta entre plataformas suspendidas mientras evita enemigos u obstáculos.

6 En UML, se conoce como estereotipo a un elemento de texto que, al ser aplicado a otro elemento, define su categoría.

7 Un DSL o Lenguaje Específico de Dominio es "un lenguaje de programación o especificación dedicado a resolver un problema en particular, representar un problema específico, y proveer una técnica para solucionar una situación particular" (Mernik, Heering, & M. Sloane, 2005).

En otro trabajo, Usman et al. (Usman, Zohaib Iqbal, & Uzair Khan, 2017) propone una SPL[8] impulsada por modelos que se centra en el desarrollo de juegos móviles multiplataforma (Android y Windows Phone) y su mantenimiento. Emplean un modelo de características[9] para configurar los casos de uso de UML, y los diagramas de máquina de estados. Aunque los detalles sean diferentes, los trabajos arriba nombrados comparten una aserción común: en el dominio de los videojuegos, la generación automatizada de código desde modelos de software tiene el potencial para reducir significativamente el esfuerzo y coste del desarrollo. Paradójicamente, la independencia de plataforma es un problema que ha sido abordado por tecnologías ampliamente utilizadas como *Unity* (Unity Technologies, 2005-2021) y *Unreal Engine* (Epic Games, 1998-2021) y está llevando a los desarrolladores a tener una menor preocupación por este problema en este apartado en particular.

En la intersección entre modelos de software y la computación evolutiva[10], Williams et al. (Williams, Poulding, Rose, Paige, & Polack, 2011) emplean un algoritmo evolutivo para buscar comportamientos deseados en el personaje de un video juego basado en texto en el que juega combates sin vigilancia y genera un resultado final. El comportamiento del jugador se define utilizando un lenguaje específico de dominio. Los combates se gestionan internamente y sólo son dirigidos por parámetros de comportamiento, sin tener en cuenta el entorno espacial, representación a tiempo real, o retroalimentación visual (el cuál sí tiene en consideración la interacción física de los personajes, variación en las propiedades, etc.). De todos modos, el caso de estudio es un juego de texto simplificado. Además, *Identifying desirable game character behaviours through the application of evolutionary algorithms to model-driven engineering metamodels* (Williams, Poulding, Rose, Paige, & Polack, 2011) lidia con el ajuste de parámetros del juego, lo que quiere decir que el trabajo no se dirige a la generación de modelos de software.

Para *Kromaia* (Kraken Empire, 2014), cuentan Daniel Blasco et al. (Blasco, Font, Zamorano, & Cetina, 2021) en el *paper An Evolutionary Approach for Generating Software Models: The case of Kromaia in Game Software Engineering*, que los jefes finales se especifican con un modelo DSL para el dominio de los videojuegos llamado *Shooter Definition Model Language*, o *SDML*.

8 "Se definen las líneas del producto de software como un conjunto de sistemas software, que comparten un conjunto común de características (*features*), las cuales satisfacen las necesidades específicas de un dominio o segmento particular de mercado, y que se desarrollan a partir de un sistema común de activos base (*core assets*) de una manera preestablecida" (Clements & Northrop, 2001).

9 "Los modelos de características especifican el conjunto de productos válidos que pueden obtenerse de una SPL. Estos modelos se diseñan en la primera fase de la línea de productos y por lo tanto juegan un papel central en todas las fases del desarrollo de la SPL" (Muñoz Riascos & Rincón, 2014).

10 "La computación evolutiva es una rama de la inteligencia artificial que involucra problemas de optimización combinatoria. Se inspira en los mecanismos de la evolución biológica" (Carmona Suárez & Fernández Galán, 2019).

Específicamente, *SDML* define aspectos incluidos en las entidades de un videojuego tales como la estructura anatómica (incluyendo las partes, propiedades físicas, y cómo se conectan entre ellas), la cantidad y distribución de las partes vulnerables, armas, y defensas en la estructura del personaje, o los comportamientos de movimiento asociados al cuerpo entero o sus partes.

3.2. La aportación de este trabajo

Hemos visto que, en el dominio de los videojuegos, la generación automatizada de código desde modelos de software tiene el potencial para reducir significativamente el esfuerzo y coste del desarrollo, y en algunos de los casos vistos se ha demostrado el funcionamiento de los modelos en pruebas limitadas. Sin embargo, también hemos visto que la independencia de plataforma es un problema en este dominio bastante importante y por abordar, al que pueden ayudar tecnologías ya existentes y válidas como las, ya nombradas, *Unity* (Unity Technologies, 2005-2021) y *Unreal Engine* (Epic Games, 1998-2021).

Es por esto, que he querido poner en práctica esa afirmación y leer desde el motor *Unity* (Unity Technologies, 2005-2021) modelos *SDML* para su correcta visualización tal y como los parámetros del modelo indiquen utilizando el lenguaje C#.

Para ello, se han empleado tanto como modelos generados a partir de los originales del juego *Kromaia* (Kraken Empire, 2014), como modelos generados por 32 voluntarios en un experimento llevado a cabo en las instalaciones de la Universidad San Jorge (Universidad San Jorge, 2005-2021).

Ilustración 5: SDML del modelo "Model02" diseñado y generado por un voluntario en el experimento del día 19/05/2021 cargado en el visualizador de este trabajo. Fuente: Propia.

4. Objetivos

Los objetivos para este proyecto fueron desde el principio concluir resultados sobre la investigación y generar recursos para la divulgación de estos mediante la visualización de jefes de *Kromaia* (Kraken Empire, 2014) en el motor *Unity* (Unity Technologies, 2005-2021) a través de la lectura de modelos *SDML* y la posibilidad de multiplataforma (en concreto, disponibilidad para PC y web).

Esto engloba:

- Lectura y comprensión del modelo *SDML* por parte del programa.
- Procesamiento de los datos correcto para generar el jefe en tiempo de ejecución.
- Lectura posible tanto desde PC como desde web.
- Experiencia de usuario capaz de permitir la exploración del modelo generado e interfaz de usuario capaz de permitir el cambio de modelo.
- Diseño modular para poder ampliarlo cuando sea necesario.

5. Metodología

5.1. Metodología empleada y por qué ha sido elegida

Al haber sido el desarrollo de este proyecto individual, en vez de utilizar una metodología más común en el desarrollo de software como SCRUM u otras metodologías ágiles, he decidido utilizar *Getting Things Done* (Allen, 2001), más comúnmente conocida como *GTD*.

GTD es un sistema de productividad personal desarrollado por David Allen que se centra en el hecho de que si una tarea está en tu mente, la ocupará por completo y no podrás llevar a cabo correctamente ninguna otra, cometiendo errores en ambas tareas y sufriendo estrés.

La solución que propone esta metodología es apuntar todos los elementos de información, tareas y proyectos en un elemento externo con el fin de sacar de tu cabeza todas estas tareas y olvidarte de ellas, para más tarde procesarlas en diferentes tipos de listas, siendo la primera de estas listas la *Inbox,* o Bandeja de entrada, donde apuntar todas las tareas según vayan surgiendo para después organizarlas en recordatorios, proyectos, y tareas procesables que se puedan realizar directamente de principio a fin.

Es un método muy eficaz para poder realizar proyectos individuales a la vez que otros asuntos y proyectos personales, teniendo siempre una mente clara y sin estrés, por lo que he decidido que sería la opción apropiada para mí en la actualidad, al ser mi situación actual personal bastante ajetreada.

5.2. Planificación inicial

Mi planificación inicial conformaba 4 bloques:

Fase	Tareas	Horas esperadas
Comprensión del SDML	Aprender a utilizar el editor de jefes creado por SVIT	10
	Entender la traducción a *SDML* y las diferencias con XML	20
	Subtotal	**30**
Integración en el motor	Hacer un programa capaz de leer y entender el metamodelo *SDML*	100
	Representar cada uno de los conceptos en el motor	30
	Generar visualmente los modelos en el proyecto	10
	Conseguir enlaces con físicas funcionales siguiendo los parámetros del *SDML*	20
	Integración para WebGL[11] y petición de archivos *SDML* en tiempo real	20
	Subtotal	**180**
Experiencia de usuario	Movimiento en mundo para la correcta visualización del modelo en 3D	20
	Menú de pausa con diversas opciones que permita cambiar de modelo	10
	Optimización y minimización de tiempos de carga y espera	30
	Creación de entorno apropiado (integración de música y fondo de *Kromaia*, cuadro de controles, etc.)	10
	Subtotal	**70**
Análisis de resultados	Reuniones y revisiones con Carlos Cetina (tutor de este TFG)	10
	Experimento con voluntarios	10
	Subtotal	**20**
	Total	**300**

Tabla 1: Tabla de planificación inicial de este proyecto. Fuente: Propia.

11 "WebGL trae gráficos en 3D para la Web mediante la introducción de una API que cumple estrictamente la OpenGL ES 2.0 que se puede utilizar en elementos canvas HTML5" (Mozilla and individual contributors, 2005-2021).

- "Comprensión del *SDML*": En esta fase, mi intención era entrar en contexto para la realización del trabajo entendiendo el medio y el material con el que iba a trabajar. Tengo la suerte de conocer y saber utilizar correctamente *Unity* (Unity Technologies, 2005-2021), ya que es el motor que empleo desde 2018, y esto me permitió evitar la fase de familiarización con el motor empleado.
- "Integración en el motor": El grueso del trabajo, donde planeaba tener el programa ya funcionando en multiplataforma (PC y WebGL).
- "Experiencia de usuario": Todo lo necesario para poder utilizar el programa correcta y fácilmente.
- "Análisis de resultados": Reuniones con mi tutor cada vez que avanzara y, finalmente, un experimento con 32 voluntarios desarrollado en las instalaciones de la USJ (Universidad San Jorge, 2005-2021) donde pondría a prueba el funcionamiento de mi herramienta en modelos *SDML* para los que no estaba originalmente diseñada.

5.3. Planificación final

Finalmente, he acabado empleando mucho más tiempo del esperado en la integración en el motor, lo que se ve contrarrestado con los demás apartados llevándome menos tiempo del esperado; en especial el bloque de Experiencia de usuario, como se puede ver en la tabla siguiente:

Fase	Tareas	Horas esperadas	Coste en horas
Comprensión del SDML	Aprender a utilizar el editor de jefes creado por SVIT	10	8
	Entender la traducción a *SDML* y las diferencias con XML	20	20
	Subtotal	**30**	**28**
Integración en el motor	Hacer un programa capaz de leer y entender el metamodelo *SDML*	100	85
	Representar cada uno de los conceptos en el motor	30	40
	Generar visualmente los modelos en el proyecto	10	5
	Conseguir enlaces con físicas funcionales siguiendo los parámetros del *SDML*	20	60
	Integración para WebGL y petición de archivos *SDML* en tiempo real	20	40
	Subtotal	**180**	**230**
Experiencia de usuario	Movimiento en mundo para la correcta visualización del modelo en 3D	20	10
	Menú de pausa con diversas opciones que permita cambiar de modelo	10	12
	Optimización y minimización de tiempos de carga y espera	30	20
	Creación de entorno apropiado (integración de música y fondo de *Kromaia*, cuadro de controles, etc.)	10	6
	Subtotal	**70**	**48**
Análisis de resultados	Reuniones y revisiones con Carlos Cetina (tutor de este TFG)	10	6
	Experimento con voluntarios	10	13
	Subtotal	**20**	**19**
	Total	**300**	**325**

Tabla 2: Tabla de planificación final de este proyecto. Fuente: Propia.

Los principales problemas han aparecido en dos apartados del bloque "Integración en el motor":

- Al aplicar físicas al modelo generado en el apartado "Conseguir enlaces con físicas funcionales siguiendo los parámetros del *SDML*", debido a que las físicas en el motor Unity funcionan de forma diferente a la que lo hacían en el motor original de *Kromaia* (Kraken Empire, 2014).
- En la petición de los modelos *SDML* en la versión para WebGL, ya que la seguridad y protocolado para enviar y recibir ficheros mediante Web es mayor que para hacerlo en un mismo sistema como es el caso de la versión de PC.

Finalmente, el objetivo inicial de "Conseguir enlaces con físicas funcionales siguiendo los parámetros del *SDML*" no pudo completarse, ya que este problema acabó siendo bastante más complejo de lo planeado originalmente y, tras emplear el triple del tiempo esperado en este punto, me reuní con mi tutor y concluimos que no era una parte esencial del proyecto y que, por tanto, sería mejor dejarlo para centrarme en partes más importantes como la integración para WebGL.

6. Estudio económico

Este trabajo forma parte de un trabajo de investigación de SVIT (SVIT Research Group, 2011-2021), y por tanto, no hay una forma correcta de realizar un estudio de viabilidad del proyecto al no poder estimar los posibles beneficios ni conocer el trabajo exacto que han realizado otros compañeros. Sin embargo, se pueden recopilar todos los costes aproximados requeridos para diseñar, desarrollar, e integrar este trabajo; tanto recursos humanos como software y otros servicios empleados.

6.1. Recursos humanos

Aunque haya más personas desarrollando otros apartados del mismo proyecto, como la herramienta de diseño o de validación, este apartado se va a centrar únicamente en el visualizador que se ha desarrollado en este trabajo, por lo que las personas involucradas en este proyecto en particular somos los siguientes:

- El tutor, Carlos Cetina, que ha actuado como cliente y provisto de retroalimentación constante en mis avances por un total aproximado de 6 horas de reuniones, como se puede ver en el anexo. Para la estimación del salario, se considerará como programador analista.
- Daniel Blasco, que como desarrollador original de *Kromaia* (Kraken Empire, 2014), ha provisto de información, materiales, y recursos del videojuego original. Ya que los materiales y recursos ya existían de anteriores trabajos y proyectos, y que este apartado es para recursos humanos, sólo se considerarán las horas de reunión conmigo ayudándome a enfocar la implementación, por un total de 5 horas como se puede ver en el anexo, y las horas de búsqueda de recursos: aproximadamente 3. Para la estimación de salario, se considerará, también, como programador analista.
- Los tres compañeros que me han acompañado en los experimentos, 13 horas cada uno; Jorge Chueca, Rodrigo Casamayor, y Jaime Font; a los que consideraré programadores Junior y programador analista respectivamente.
- El autor de este proyecto, Javier Verón, que ha diseñado y desarrollado esta herramienta de visualización, por un total de 325 horas. Para la estimación de salario, se considerará como programador Junior, al no tener más de 2 años de experiencia laboral (Campana, 2018).

Para estimar los costes, se ha empleado un motor de búsqueda de empleo y media de salarios llamado Indeed (Indeed, 2004-2021), especificando por hora y como país España, al pertenecer a este país SVIT (SVIT Research Group, 2011-2021) y yo residir en el mismo. Los sueldos que se han encontrado en este motor son en bruto, por lo que ha habido que añadirles el coste de la Seguridad Social aparte (Camacho, 2021).

Puesto de trabajo	Precio por hora	Horas	Sueldo bruto	Coste SS	Coste total
Programador Analista 1	14,47 €	6	86,82 €	26,05 €	112,87 €
Programador Analista 2	14,47 €	8	115,76 €	34,73 €	150,49 €
Programador Junior 1	9,56 €	13	124,28 €	37,28 €	161,56 €
Programador Junior 2	9,56 €	13	124,28 €	37,28 €	161,56 €
Programador Analista 3	14,47 €	13	188,11 €	56,43 €	244,54 €
Programador Junior 3	9,56 €	325	3.107,00 €	932,10 €	4.039,10 €
				Total	**4.870,13 €**

Tabla 3: Estimación de costes de recursos humanos. Fuente: Propia.

6.2. Recursos materiales

Los recursos materiales y de software han sido diversos programas que se ven reflejados en la siguiente tabla y, para el experimento, dos aulas (para cuyo precio he cogido el ofrecido por BSSC (Centro de negocios en Zaragoza, 2021) como referencia) y 16 ordenadores portátiles que, aunque ambos préstamos de aulas y ordenadores hayan sido cedidos por la USJ (Universidad San Jorge, 2005-2021), tendré también en cuenta.

Para servicios mensuales he contado únicamente un mes, ya que en algo más de un mes a jornada completa se habrían finalizado las 325 horas.

Software			
Recurso	**Precio**	**Unidades**	**Coste total**
Windows 10 (Microsoft Corporation, 2015-2021)	139,00 €	1	139,00 €
Microsoft 365 (Microsoft Corporation, 2011-2021)	7,00 €	1	7,00 €
JetBrains Rider (JetBrains s.r.o, 2000-2021)	13,90 €	1	13,90 €
Unity (Personal) (Unity Technologies, 2005-2021)	0,00 €	1	0,00 €
Adobe Photoshop (Adobe Inc., 1990-2021)	31,49 €	1	31,49 €
		Total	**191,39 €**
Hardware			
Recurso	**Precio**	**Unidades**	**Coste total**
Asus ROG Strix GL502VM-FY213T (ASUS, 2017)	1.752,00 €	16	28.032,00 €
		Total	**28.032,00 €**
Servicios			
Recurso	**Precio**	**Unidades**	**Coste total**
Sala foro de 75 m² (Centro de negocios en Zaragoza, 2021)	180,00 €	2	360,00 €
Agua (Ferro, 2021)	15,64 €	1	15,64 €
Electricidad (Cisa, 2021)	70,95 €	0,5	35,48 €
Internet (O2 España, 2018-2021)	38,00 €	1	38,00 €
		Total	**449,12 €**

Tabla 4: Estimación de costes de recursos materiales. Fuente: Propia.

6.3. Coste total

Contando con todos los gastos en recursos juntos, humanos y materiales, el presupuesto requerido del proyecto tal y como aparece encima quedaría en 33.542,63 €.

Sin embargo, si descontamos de este precio los 16 ordenadores y las dos aulas cedidas para el experimento por la USJ (Universidad San Jorge, 2005-2021), ya que ha sido todo esto prestado por la misma Universidad, el presupuesto baja drásticamente a unos, mucho más realistas, 5.150,63 €.

7. Desarrollo e Implementación

En este apartado veremos cómo durante el desarrollo del visualizador ha habido cambios en el diseño debido a que las primeras referencias se tomaron, como se explicará a continuación, utilizando *XML*[12].

También veremos cómo se han afrontado las decisiones tanto de diseño como de implementación para un resultado final multiplataforma para PC y para WebGL.

7.1. Análisis de la estructura de un jefe y comienzos utilizando XML

Para poder empezar a trabajar en mi proyecto, era necesario trabajar primero con la herramienta de diseño de jefes y creación de archivos *SDML* que SVIT (SVIT Research Group, 2011-2021) tenía desarrollada en el momento.

En esta herramienta, como se puede ver en la ilustración 6, se pueden insertar las diversas partes que forman a un jefe; como *hulls* (cascos), *links* (enlaces), *weapons* (armas), etcétera; e introducir los atributos necesarios y deseados para cada una de estas.

En el ejemplo que vemos a la derecha vemos únicamente los parámetros del casco '0', ya que es este el que tenemos seleccionado; pero si seleccionamos otro elemento, los parámetros que aparecen en la parte inferior del editor cambiarían a los propios de esa parte.

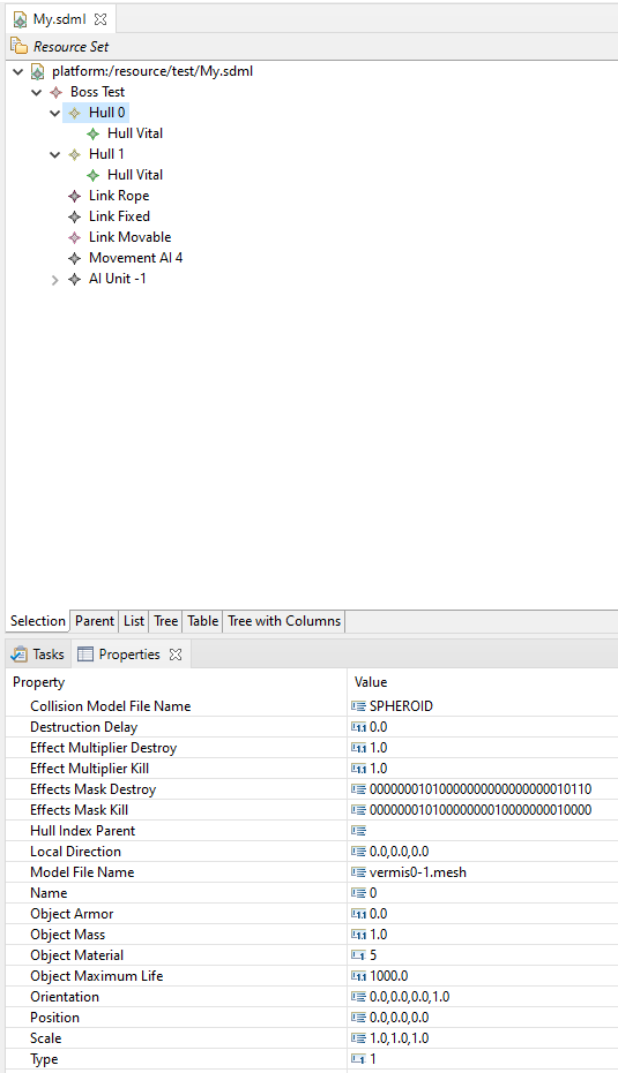

Ilustración 6: Ejemplo de diseño de un metamodelo SDML de un jefe en la herramienta de diseño de SVIT. Fuente: Propia.

12 *XML* significa "*Extensible Markup Language"* y es "un lenguaje de etiquetas, es decir, cada paquete de información está delimitado por dos etiquetas" (Sagástegui Lescano, Consultado en 2021). Se utiliza "para estructurar información en cualquier documento que contenga texto, como por ejemplo documentos de configuración de un programa o una tabla de datos" (RI5, Consultado en 2021).

Así, podemos ir rellenando los campos con los valores deseados hasta tener el jefe que buscamos.

Sin embargo, para poder crear el visualizador desde un ejemplo práctico completo, es más seguro hacerlo con un modelo creado por un modelador profesional que con uno creado por mí, el cual en ese momento del desarrollo no tenía aún. De lo que sí disponía a esas alturas, sin embargo, es del modelo en *XML* del jefe *Vermis* (Ilustración 7).

Ilustración 7: jefe Vermis de Kromaia. Fuente: Videojuego Kromaia.

Con este archivo *XML* pude empezar a analizar cómo abordar el proyecto y luego traducir el trabajo que llevara de lectura de *XML* a lectura de *SDML*.

7.1.1. Análisis de XML:

Cuando abrí el archivo *XML* por primera vez, lo primero en lo que me fijé es en que era enorme: 1070 líneas de texto. Por ello, decidí empezar por el principio y ver cómo estaba distribuido.

El jefe entero está dividido en un árbol de etiquetas, todas ellas partiendo de una raíz llamada "compuesto" *(<COMPOUND>)*, el cual contiene todos los elementos del jefe por separado.

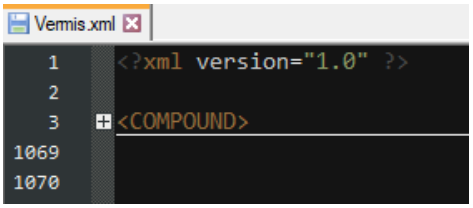

Ilustración 8: Etiqueta COMPOUND en el archivo XML del jefe Vermis. Fuente: Propia.

Tras esto, encontramos la primera etiqueta con información: *<GENERAL>*. En esta etiqueta se encuentra toda la información relevante al jefe en general, y no a una parte en específico, como la escala inicial, la pérdida de velocidad lineal y angular por segundo, etc.

Después de la etiqueta *<GENERAL>*, hay otra etiqueta llamada *<HULL>*, que únicamente tiene como parámetros el número de cascos que vamos a encontrar en su interior como parámetro.

Como el número del parámetro indicaba, la etiqueta *<HULL>* contiene; en el caso de "*Vermis*", 34 cascos con la etiqueta *<Hull>*. Cada uno de ellos tiene el tipo de casco al que pertenece como parámetro en forma de número entero y contiene en su interior una serie de elementos con las siguientes etiquetas:

- *<ObjectData>*: Contiene el nombre del archivo del modelo 3D que se asociará al casco al que pertenece esta etiqueta), la escala en el espacio tridimensional, la posición en el espacio tridimensional, y la orientación como cuaternión.
- *<PhysicalObjectData>*: Tiene el nombre del archivo del modelo de colisiones 3D (que a menudo es uno menos detallado que el visible para tener colisiones más sencillas de calcular), la masa del objeto, y el material del objeto.
- *<AliveObjectData>*: Esta etiqueta contendrá la vida máxima que tiene el casco al que pertenece (en caso de tener vida, ya que en caso negativo el valor será -1), la armadura de la que dispone, y otros parámetros relacionados con la destrucción del casco y los efectos que aparecen cuando esto ocurre.
- *<ComponentData>*: Contiene la dirección local actual de este casco en particular y dentro de qué casco se encuentra este, si es que estuviera dentro de un casco. Cuando no

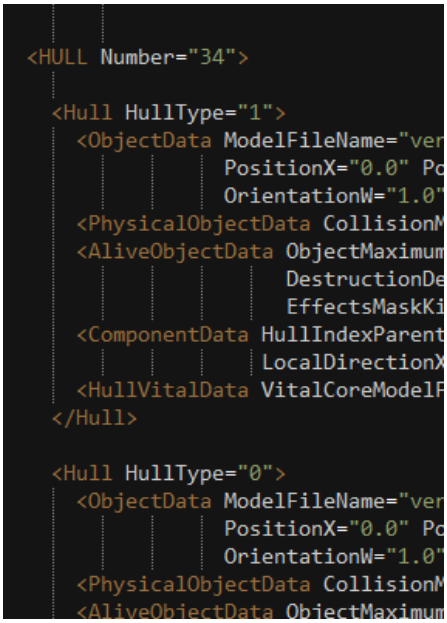

Ilustración 9: Etiquetas HULL y Hull en el archivo XML del jefe Vermis. Fuente: Propia.

pertenece a ningún casco, el valor es -1, pero este es un atributo interesante ya que nos permite insertar cascos dentro de cascos para hacer una ramificación real dentro del juego.

Finalmente, si pertenece al tipo '1', será un casco de tipo vital, por lo que también contendrá otra etiqueta llamada *<HullVitalData>*. Esta etiqueta contiene el nombre del archivo del modelo 3D del núcleo vital, y la capa en la que se encuentra en el juego (lo que habitualmente se emplea en colisiones).

Tras 34 cascos, y aún dentro de la etiqueta superior *<HULL>*, encontramos otra etiqueta que nos marca el comienzo de otra colección con la etiqueta *<LINKS>*, cuyo único parámetro vuelve a ser el número de elementos que comprende y tiene en su interior ese mismo número de elementos con la etiqueta *<Link>*.

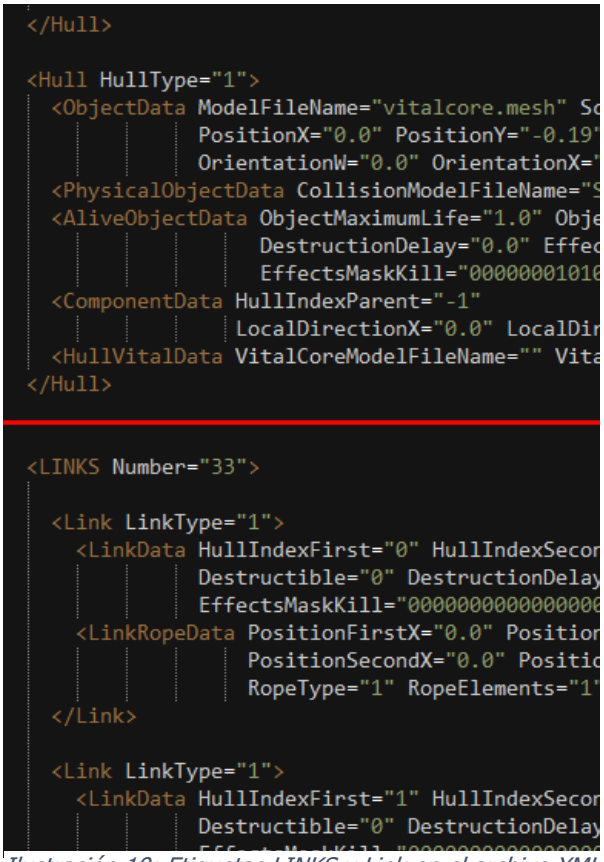

Ilustración 10: Etiquetas LINKS y Link en el archivo XML del jefe "Vermis" tras la última etiqueta Hull. Fuente: Propia.

Cada *<Link>* representa un enlace entre dos cascos, y tiene; al igual que la etiqueta *<Hull>*; un solo atributo en forma de número entero que indique el tipo de enlace al que pertenece, para luego contener más etiquetas en su interior. La única etiqueta interior que comparten los diferentes tipos de enlace es *<LinkData>*, que contiene el identificador de los dos cascos que une, si es o no destruible, y los efectos y demás atributos sobre qué ocurre durante su destrucción. Las etiquetas que sigan a esta antes de cerrar la etiqueta *<Link>* serán propias de cada tipo de enlace. Por ejemplo, el enlace de tipo movible tendrá una etiqueta llamada *<LinkMovableData>* con información sobre cómo se va a poder mover.

Como podemos ver en la ilustración 11, cuando acabe el último elemento con la etiqueta *<Link>* se cierra también la etiqueta *<LINKS>* y la etiqueta *<HULL>*, para dar paso a la siguiente etiqueta contenedora de una colección: *<WEAPONRYWEAPONSAI>*.

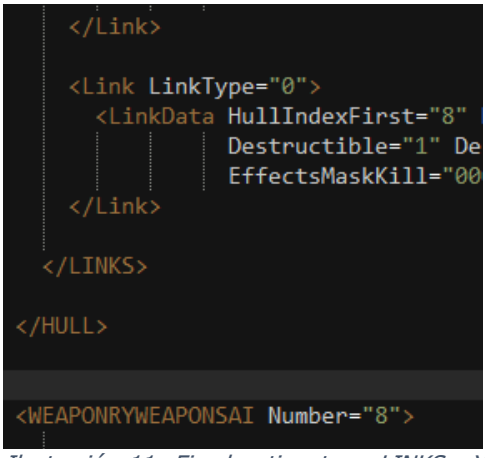

Ilustración 11: Fin de etiquetas <LINKS> Y <HULLS> y comienzo de etiqueta <WEAPONRYWEAPONSAI> en el archivo XML del jefe Vermis. Fuente: Propia.

Como ha ocurrido previamente, *<WEAPONRYWEAPONSAI>* nos vuelve a presentar como único atributo el número de elementos que contendrá; en este caso, 8 elementos con la etiqueta *<Weapon>*, que serán las armas del jefe Vermis.

La etiqueta *<Weapon>* viene, de nuevo, acompañada del atributo que indica el tipo de arma.

Las armas tienen, también, una enorme cantidad de elementos en su interior, ya que en algunos casos también tienen que contar con los datos de los proyectiles que disparen y los cañones con los que los disparan.

Las etiquetas que contiene cada elemento con etiqueta *<Weapon>* son las mismas que las etiquetas que contienen los elementos comunes con etiqueta *<Hull>*, más los siguientes añadidos:

- *<MobileObjects>*: Esta etiqueta es un nuevo conjunto de elementos dentro de cada arma, conteniendo un número de elementos fijado en un atributo y un tipo concreto de movilidad definido con un número entero, al igual que los tipos de armas, enlaces, y cascos. Los elementos contenidos tienen la etiqueta *<MobileObject>* y como parámetros una referencia al nombre del archivo del modelo 3D del objeto movible y la posición en la que se encontrará.
- *<WeaponData>*: Tiene atributos para el comportamiento del arma como: prioridad, tiempo de recarga, etc.
- *<WeaponWithCannonsData>*: Contiene datos como: el máximo ángulo de la torreta, su aceleración angular, su máxima velocidad angular, etc.
- *<WeaponCompoundObjectData>*: Tiene información sobre la munición y los proyectiles del arma como: la munición máxima, la dispersión, la escala, velocidad lineal y angular, etc.

- *<WeaponProjectileData>*: Especifica el tipo de proyectil y el poder de este.
- *<Cannons>*: Similarmente a *<MobileObjects>*, indica el número de elementos con la etiqueta *<WeaponCannon>* que va a contener y el tiempo máximo de turno de disparo que tendrá cada uno. Dentro de *<Cannons>*, cada *<WeaponCannon>* contiene a su vez otro elemento con la etiqueta *<WeaponCannonData>* que tiene los atributos necesarios para saber la posición y dirección de este elemento e información sobre el comienzo y final de su turno de disparo.

```
<Weapon WeaponType="3">
  <ObjectData ModelFileName="" ScaleX="1.0" ScaleY="1.0" Scale
              PositionX="0.0" PositionY="0.0" PositionZ="0.0"
              OrientationW="1.0" OrientationX="0.0" Orientatio
  <PhysicalObjectData CollisionModelFileName="" ObjectMaterial
  <AliveObjectData ObjectMaximumLife="100.0" ObjectArmor="0.0"
                   DestructionDelay="0.0" EffectMultiplierKill
                   EffectsMaskKill="00000001010000000000000000
  <ComponentData HullIndexParent="8"
                 LocalDirectionX="0.0" LocalDirectionY="0.0" L
  <WeaponData WeaponPriority="2" WeaponBehaviour="2" ReloadTim
              AffectorType="-1"
              AffectorScaleFactor="1.0" AffectorPowerFactor="1
              AffectorPositionX="0.0" AffectorPositionY="0.0"
  <WeaponWithCannonsData MaximumTurretAngle="0.0" Acceleration
                         FiringOffsetLocalMultiplier="0.0" Fir
                         ReachDestinationAtFullAngularSpeed="f
  <WeaponCompoundObjectData MaximumAmmo="-1" Dispersion="0.0"
                            ProjectileScaleTime="0.25"
                            ProjectileInitialScaleX="0.1" Proj
                            ProjectileFinalScaleX="0.1" Projec
                            ProjectileInitialLinearVelocityX="
                            ProjectileInitialAngularVelocityX=
  <WeaponProjectileData ProjectileType="0" ProjectilePower="10
  <WeaponProjectileLoadedData ProjectileLoadedType="18" Projec
                              MaximumRollingRadiansPerSecond="
                              ProjectileLoadInitialScaleX="0.1
                              ProjectileLoadFinalScaleX="0.1"
  <Cannons Number="3" FiringTurnMaximum="0.0" >
    <WeaponCannon>
      <WeaponCannonData FiringTurnStart="0" FiringTurnEnd="0"
                        FiringPositionX="0.0" FiringPositionY=
                        FiringDirectionX="0.0" FiringDirection
    </WeaponCannon>

    <WeaponCannon>
      <WeaponCannonData FiringTurnStart="0" FiringTurnEnd="0"
                        FiringPositionX="0.3" FiringPositionY=
                        FiringDirectionX="0.0" FiringDirection
    </WeaponCannon>

    <WeaponCannon>
      <WeaponCannonData FiringTurnStart="0" FiringTurnEnd="0"
                        FiringPositionX="-0.3" FiringPositionY
                        FiringDirectionX="0.0" FiringDirection
    </WeaponCannon>
  </Cannons>
</Weapon>

<Weapon WeaponType="0">
  <ObjectData ModelFileName="" ScaleX="1.0" ScaleY="1.0" Scale
```

Ilustración 12: Etiquetas Weapon en el archivo XML del jefe Vermis. Fuente: Propia.

De nuevo, dependiendo del tipo de arma al que pertenezca cada elemento, tendrá contenidos más elementos con etiquetas como *<WeaponProjectileLoadedData>*, que especificarían datos y atributos especiales para ese tipo de arma en concreto.

Finalmente, tenemos los elementos con las etiquetas *<MOVEMENTAI>*, con sus contenidos *<AIModuleMovementDirectData>* y *<AIModuleMovementOrientedData>*; y *<AIUNIT>*, que de nuevo actúa como contenedor de los elementos con la etiqueta *<BehaviourUnitAIData>* que hay dentro.

Estas etiquetas ofrecen información sobre la IA del jefe. No hablaré de ellas, ya que en este proyecto no se ha utilizado nada de esto al ser sólo un visualizador sin modo de juego real.

```
<MOVEMENTAI AIModuleMovementType="4">
  <AIModuleMovementDirectData Acceleratio
  <AIModuleMovementOrientedData Accelerat
</MOVEMENTAI>

<AIUNIT Number="3" UnitValue="-1">
    <BehaviourUnitAIData AIFormationTyp
                         AIGroupBehavio
                         AIBehaviourTyp

    <BehaviourUnitAIData AIFormationTyp
                         AIGroupBehavio
                         AIBehaviourTyp

    <BehaviourUnitAIData AIFormationTyp
                         AIGroupBehavio
                         AIBehaviourTyp
</AIUNIT>
```

Ilustración 13: Etiquetas <MOVEMENTAI> y <AIUNIT> con sus contenidos en el archivo XML del jefe Vermis. Fuente: Propia.

A continuación, se puede ver en la ilustración 14 un ejemplo condensado de la estructura del modelo de un jefe en lenguaje *XML* tal y como hemos visto hasta ahora.

```
<COMPOUND>
	<GENERAL/>
	<HULL>
		<Hull>
			<ObjectData/>
			<PhysicalObjectData/>
			<AliveObjectData/>
			<ComponentData/>
			<HullVitalData/> (Dependiendo del tipo, está o no)
		</Hull>
		<LINKS>
			<Link>
				<LinkData/>
				<LinkRope/> (Dependiendo del tipo, está o no)
				<LinkMovableData/> (Dependiendo del tipo, está o no)
			</Link>
		</LINKS>
	</HULL>
	<WEAPONRYWEAPONSAI>
		<Weapon>
			<ObjectData/>
			<PhysicalObjectData/>
			<AliveObjectData/>
			<ComponentData/>
			<MobileObjects>
				<MobileObject/>
			</MobileObjects>
			<WeaponData/>
			<WeaponWithCannonsData/>
			<WeaponCompoundObjectData/>
			<WeaponProjectileData/> (Dependiendo del tipo, está o no)
			<WeaponProjectileData/> (Dependiendo del tipo, está o no)
			<WeaponProjectileLoadedData/> (Dependiendo del tipo, está o no)
			<WeaponProjectileData/>
			<Cannons>
				<WeaponCannon>
					<WeaponCannonData/>
				</WeaponCannon>
			</Cannons>
		</Weapon>
	</WEAPONRYWEAPONSAI>
	<MOVEMENTAI>
		<AIModuleMovementDirectData/>
		<AIModuleMovementOrientedData/>
	</MOVEMENTAI>
	<AIUNIT>
		<BehaviourUnitAIData/>
	</AIUNIT>
</COMPOUND>
```

Ilustración 14: Condensación de la estructura del metamodelo de un jefe en lenguaje XML. Fuente: Propia.

Con la estructura que acabamos de ver, pude comenzar a diseñar y prototipar una primera versión de lo que sería el visualizador, capaz de leer archivos *XML* y mostrar por pantalla diferentes figuras geométricas dependiendo de qué se estuviera representando en cada momento.

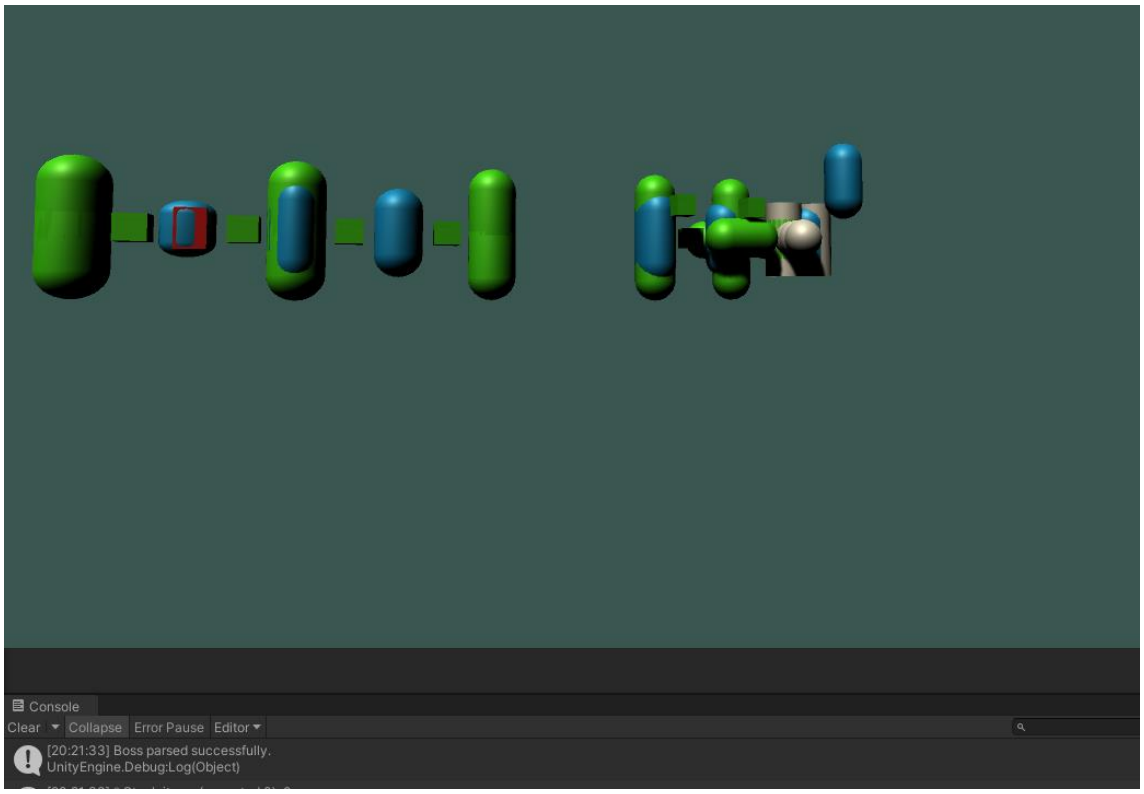

Ilustración 15: Resultado de la versión más temprana del visualizador, leyendo desde lenguaje XML el jefe "Vermis". Fuente: Propia.

7.1.2. Diseño e implementación tempranos del visualizador para leer desde XML

Para el diseño de la primera versión, cuyo resultado podemos ver en la ilustración 15, me reuní, como puede verse en anexos, con Daniel Blasco; desarrollador del videojuego original de *Kromaia* (Kraken Empire, 2014); que me facilitó la ilustración 16, en la que se puede ver perfectamente la herencia de las clases que al leer y procesar el modelo se deberán instanciar.

Con esta base, pude comenzar a diseñar las clases indicadas en la imagen y, en cada una, diversos métodos para inicializar cada una de las partes de las que hemos visto en el apartado anterior.

A continuación necesitaba una forma de crear todo en tiempo real y teniendo referencias a lo que se indicara por encima en la jerarquía del *XML*. Tras considerar varias opciones, escogí como la estructura de datos necesaria para resolver este problema de forma óptima la pila[13].

En la ilustración 17 se puede leer el método encargado de leer de principio a fin el archivo *XML* mientras se va creando el jefe en tiempo real. Para ello, utilizo una

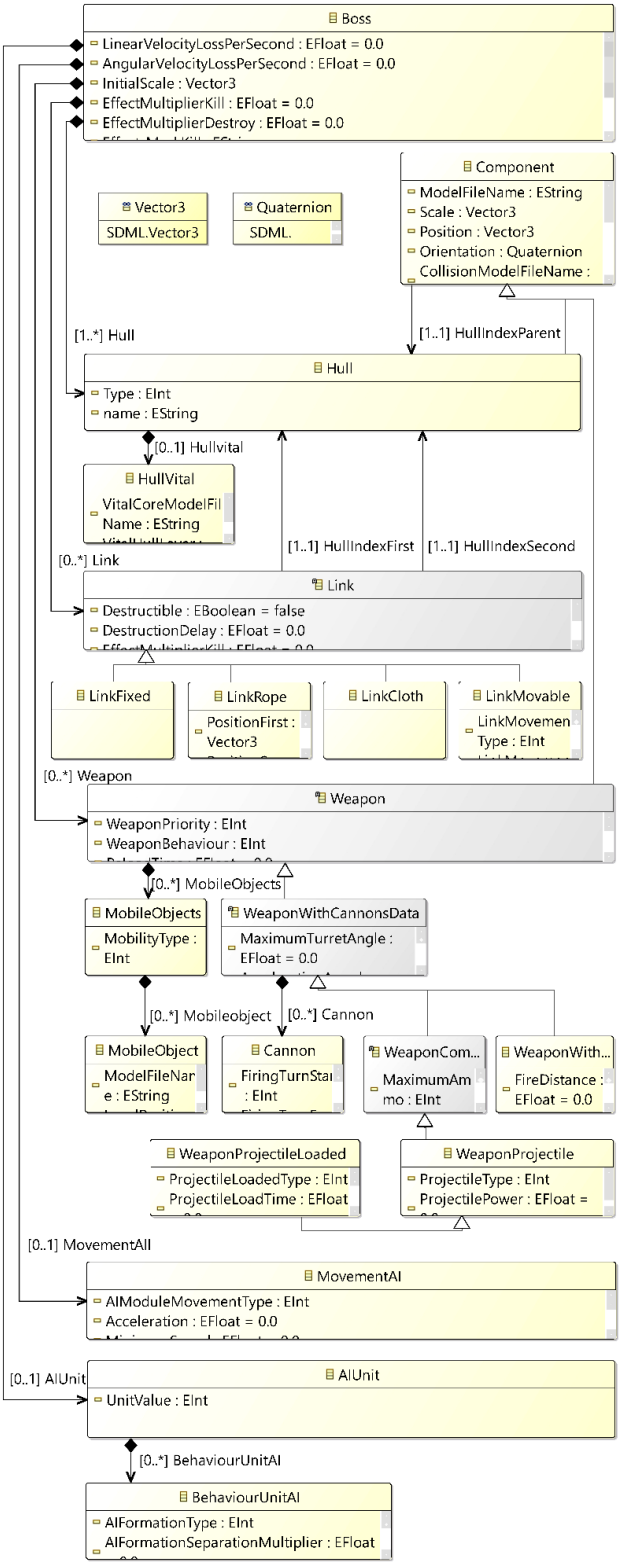

Ilustración 16: Esquema de clases para jefes de Kromaia. Fuente: Daniel Blasco.

13 "Una pila es una colección o lista estructurada de llamadas a funciones y parámetros utilizada en programación computacional moderna y arquitectura de unidades centrales de procesamiento. Igual que una pila de platos en un restaurante o cafetería, los elementos en la pila son añadidos o eliminados de la misma desde la parte alta de la pila, en un orden *LIFO* (último en entrar, primero en salir)" (Bolton, 2019).

pila en la que voy insertando elementos cuando comienza una etiqueta y sacándolos cuando acaba.

Con esto, tengo acceso a las anteriores referencias leídas y puedo llamar a las funciones de inicialización del jefe pertinentes a los datos existentes para cada elemento del archivo *XML*.

```
private void ReadXmlFile(string path)
{
    XmlTextReader reader       = new XmlTextReader(path);
    Stack<object> stack        = new Stack<object>();
    Stack<int>    placeInArray = new Stack<int>();

    while (reader.Read())
    {
        switch (reader.NodeType)
        {
            case XmlNodeType.Element:
                PushElement(reader, stack, placeInArray);
                break;
            case XmlNodeType.EndElement:
                PopElement(reader, stack, placeInArray);
                break;
            default:
                if (reader.NodeType != XmlNodeType.Whitespace &&
reader.NodeType != XmlNodeType.Comment &&
                    !reader.Name.Contains("xml"))
                {
                    Debug.Log($"EXCEPTION: Node {reader.NodeType} ->
\"Name\": {reader.Name}");
                }

                break;
        }
    }
```

Ilustración 17: Retazo del código de lectura de XML que muestra cómo los elementos se insertan y sacan de una pila. Fuente: Propia.

En cada uno de los métodos "*PushElement()*" y "*PopElement()*" se hace un *switch case*[14] en el que, dependiendo del nombre de la etiqueta actual, se llamará a unas funciones u otras esperando una referencia u otra en la cima de la pila.

```
private void PushElement(XmlTextReader reader, Stack<object> stack,
{
    switch (reader.Name)
    {
        case "COMPOUND": // Root, which will be the boss | EXPECTED
        case "GENERAL": // Data for root | EXPECTED STACK TOP: BOSS
        case "HULL": // Hulls array | EXPECTED STACK TOP: BOSS SCRIP
        case "Hull": // Hull | EXPECTED STACK TOP: ARRAY OF HULLS...
        case "ObjectData": // Component Object Data | EXPECTED STACK
        case "PhysicalObjectData": // Component Physical Object Data
        case "AliveObjectData": // Component Alive Object Data | EXP
        case "ComponentData": // Component Data | EXPECTED STACK TOP
        case "HullVitalData": // Hull Vital Data | EXPECTED STACK TO
        case "LINKS": // Links array | EXPECTED STACK TOP: ARRAY OF
        case "Link": // Link | EXPECTED STACK TOP: ARRAY OF LINKS...
        case "LinkData": // Link data | EXPECTED STACK TOP: LINK...
        case "LinkRopeData": // Link data | EXPECTED STACK TOP: LINK
```

14 "*switch* es una instrucción de selección que elige una sola sección *switch* para ejecutarla desde una lista de candidatos en función de una coincidencia de patrones con la expresión de coincidencia" (Microsoft Corporation, 2019).

El caso de elementos con etiquetas tales como *<HULL>*, *<LINKS>*, o *<WEAPONRYWEAPONSAI>* es llamativo ya que lo que se inserta en la parte alta de la pila no será una referencia a un objeto en particular como sí es el caso de cada *<Hull>*, *<Link>*, y *<Weapon>*; sino que será una colección de tipo *array*[15] con el tamaño indicado en la etiqueta. Por tanto, la parte alta de la pila esperada en cada uno de los elementos individuales será ese mismo *array*, en el que se insertarán. Para colocarlos en el sitio correcto, creé otra pila de números enteros que se encargue de contener las posiciones actuales para cada *array* que lo necesite. Decidí seguir estas cuentas utilizando otra pila, en lugar de un único número entero, ya

Ilustración 18: Captura de pantalla de un retazo de código del método PushElement() de la lectura de XML. Fuente: Propia.

15 "Un *array* es la estructura de datos que almacena un número fijo de valores literales (elementos) del mismo tipo de dato" (TutorialsTeacher, 2020).

que hay situaciones en las que elementos que forman parte de un *array* a la vez que contienen otro; por ejemplo, los elementos de etiqueta *<Weapon>* con *<Cannons>*.

7.2. El salto a SDML: Análisis

Todo parecía funcionar bien e ir en la dirección correcta. Sin embargo, poco después de llegar a estos resultados, tuve acceso al primer archivo *SDML* profesional del jefe Vermis, con lo que pude pivotar mi trabajo y comenzar a ver qué podía aprovechar de esta primera versión que había diseñado y comenzado a implementar.

Cuando empecé a analizar el metamodelo *SDML*, me di cuenta de que este lenguaje tenía varias

```
case "HULL": // Hulls array | EXPECTED STACK TOP: BOSS SCRIPT
{
   Boss script = (Boss) stack.Peek();

   script.m_Hulls = new Hull[int.Parse(reader.GetAttribute("Number"),
CultureInfo.InvariantCulture.NumberFormat)];

   stack.Push(script.m_Hulls);
   placeInArray.Push(0);
   break;
}
case "Hull": // Hull | EXPECTED STACK TOP: ARRAY OF HULLS
{
   Hull[] array = (Hull[]) stack.Peek();

   switch ((Hull.Type) int.Parse(reader.GetAttribute("HullType"),
CultureInfo.InvariantCulture.NumberFormat))
   {
      case Hull.Type.HULL_TYPE_NORMAL:
         array[placeInArray.Peek()] =
GameObject.Instantiate(m_HullPrefab).GetComponent<Hull>();
         break;
      case Hull.Type.HULL_TYPE_VITAL:
         array[placeInArray.Peek()] =
GameObject.Instantiate(m_HullVitalPrefab).GetComponent<HullVital>();
         break;
      default:
         Debug.LogError($"EXCEPTION IN GenerateElement: New Hull
found: \"Type\": {int.Parse(reader.GetAttribute(0),
CultureInfo.InvariantCulture.NumberFormat)}");
         break;
   }
   array[placeInArray.Peek()].InitSpecific();

   stack.Push(array[placeInArray.Peek()]);
   placeInArray.Push(placeInArray.Pop() + 1);
```

Ilustración 19: Retazo del código de la primera versión del método PushElement() que muestra el caso de HULL y Hull. Fuente: Propia.

diferencias importantes con *XML*. Lo primero que vi fue la espectacular diferencia de tamaño y líneas que había para el mismo jefe: el archivo *SDML* pesa 35,9 KB y tiene sólo 175 líneas,

mientras el archivo *XML* pesa 72,2 KB con 1070 líneas: el archivo *XML* pesa más del doble y tiene más de 6 veces más líneas de código conteniendo la misma información. Con esto deduje que habría que hacer algunos cambios importantes.

A continuación comencé a analizar las diferencias entre ambos archivos. El archivo *SDML* contiene la misma información que podemos observar al leerlo con el programa de diseño y edición de modelos *SDML* desarrollado por SVIT (SVIT Research Group, 2011-2021). Como hemos podido observar en el *SDML* ejemplificativo de la ilustración 6, todos los cascos, enlaces, y armas están contenido en el jefe y a la misma altura, sin ser a su vez contenidos por otra etiqueta *<HULL>*, *<LINKS>*, ni *<WEAPONRYWEAPONSAI>* que indique cuántos elementos vamos a encontrarnos; como era el caso de *XML*; y por tanto, sabía que tendría que cambiar la forma en la que se creaban los *arrays*.

La siguiente gran diferencia es que en *SDML* no existen los elementos pertenecientes a los cascos (etiquetas *<Hull>*) con las etiquetas *<ObjectData>, <PhysicalObjectData>, <AliveObjectData>,* ni *<ComponentData>*, así como sí existe ahora; contenido por únicamente los cascos de tipo vital; un nuevo elemento con la etiqueta *<Hullvital>*, que nos indica la capa en la que se encuentra (la cual se encontraba indicada en la etiqueta *<HullVitalData>* en la versión de *XML*).

Igual que en el caso de los cascos, en el caso de los enlaces (etiquetas *<Link>*) tampoco encontramos *<LinkData>* ni ninguno de los elementos específicos de cada tipo de enlace.

De la misma forma, las armas (etiquetas *<Weapon>*) pasan de contener los hasta 13 elementos que encontrábamos en *XML* a contener únicamente todos los cañones de los que disponga; estando todos al mismo nivel; y el elemento con la etiqueta *<MobileObjects>*, el cual ahora sólo indica el tipo de movilidad y contiene los diferentes elementos con la etiqueta *<Mobileobject>*.

Además, en todos los elementos no están escritos los atributos que tengan su valor por defecto.

Tras esto, sólo están los elementos etiquetados como *<MovementAII>* y *<AIUnit>*, los cuales; de nuevo; no voy a analizar ya que este trabajo se centra sólo en la visualización del jefe sin tener en cuenta nada relacionado con la IA.

Con estos cambios advertidos, comencé a modificar el diseño del lector del primer apartado para adaptarlo a la nueva estructura del lenguaje *SDML*.

7.3. Diseño

Queriendo aprovechar la máxima cantidad posible del diseño previo, decidí conservar las clases de cada parte del jefe con sus métodos y atributos tal y como estaban en la primera versión, teniendo que modificar únicamente el diseño del lector.

7.3.1. Generación de arrays de cascos, enlaces, armas, objetos móviles y cañones

Al dejar de disponer de las etiquetas *<HULL>, <LINKS>,* y *< WEAPONRYWEAPONSAI>* presentes en la versión en *XML*, las cuales tenían en sus atributos directamente el número de elementos en cada *array*, tuve que pensar qué otra forma de guardar los elementos me ofrecía *SDML*.

Probando diferentes acercamientos con la herramienta de diseño de modelos, observé que cuando creas un elemento siempre se coloca en un lugar en particular; aunque haya otros elementos; para seguir el siguiente orden: *Hulls, Links, Weapons, MovementAI, AIUnit*.

Esto significa que, en *SDML*, todos los cascos estarán seguidos, con debajo todos los enlaces seguidos, con debajo todas las armas seguidas, para acabar con los dos elementos de IA; los cuales no son de interés para este trabajo.

El diseño final para la generación de los *arrays* funciona de la siguiente forma:

Partiendo del diseño anterior, los elementos ya no se sacan de la pila al acabar de formarse, sino que se siguen introduciendo, hasta que se llegue a un elemento con una etiqueta diferente a la del elemento anterior y de igual o menor profundidad. Esta profundidad se evaluará basada en la posición del elemento con respecto a la base (la cual en *XML* era *<COMPOUND>* y en *SDML* es *<SDML:Boss>*).

Cuando se cumplan ambas condiciones, se realizará un *switch case* para; dependiendo de la etiqueta del anterior elemento (y siempre sólo en caso de que sea un casco, enlace, objeto móvil o cañón); sacar de la pila elementos uno a uno mientras se guardan en una lista[16], hasta que la clase del siguiente objeto de la pila no sea igual. Entonces, formaremos el *array* deseado a partir de la lista.

Ahora, usaré como ejemplo la ilustración 20, que condensa la estructura de los cascos en *SDML*:

```
<SDML:Boss>
        <Hull/>
        <Hull>
                <Hullvital/>
        </Hull>
</SDML:Boss>
```

Ilustración 20: Condensación de la estructura de los cascos en SDML. Fuente: Propia.

16 Representa una lista de objetos fuertemente tipados a la que se puede obtener acceso por índice. (Microsoft Corporation, Consultado en 2021)

Según la estructura de la ilustración 20, lo primero que se leerá; con profundidad = 0; será la etiqueta *<SDML:Boss>*. Esto hará que se genere un jefe y se inicialice con los atributos que se incluyan en la etiqueta.

Una vez creado el jefe, se añadirá a la pila y se pasará a leer el siguiente elemento etiquetado: *<Hull>*, que tendrá profundidad = 1. Antes de generarlo, se comprobará si este elemento tiene una etiqueta diferente a la del elemento anterior y es de igual o menor profundidad. Sí que tiene una etiqueta diferente, pero es de mayor profundidad, así que continuaremos con la generación de este: el elemento etiquetado como *<Hull>* seguirá un proceso parecido al de *<SDML:Boss>*, generando un casco u otro dependiendo de su tipo para después llamar a todas las inicializaciones necesarias, igual que lo harán los enlaces y las armas.

Cuando se haya creado el casco, se añadirá también a la pila y se pasará a leer el siguiente elemento, que de nuevo es *<Hull>* y tiene la misma profundidad, 1. De nuevo, se realizará la comprobación de etiqueta y profundidad para ver que sí tiene el mismo nombre y profundidad, por lo que se continúa el proceso de generación sin hacer nada previamente.

Tras hacer exactamente lo mismo que en el caso anterior, pasamos al siguiente elemento, con profundidad = 2 y etiqueta *<Hullvital>*. Igual que en el caso del primer casco, veremos en la comprobación que tiene un nombre diferente, pero mayor profundidad, así que; de nuevo; no hacemos más que procesar el elemento etiquetado. Este elemento en particular sólo aportará información al objeto que está justo encima, que será un casco de tipo vital.

Al acabar y leer la siguiente etiqueta, que será el cierre de la etiqueta *<Hull>*, se hará la comprobación de nuevo, devolviendo en este caso verdadero al tener una etiqueta diferente y una profundidad menor. Sin embargo, en el *switch case* no se entrará en ningún caso al no ser la etiqueta anterior (*<Hullvital>*) igual a *<Hull>*, *<Link>*, *<Weapon>*, *<Mobileobject>*, ni *<Cannon>*, por lo que la lectura continuará sin más.

Al cerrar la etiqueta *<Hull>* y pasar a la siguiente, se volverá a entrar al *switch case*, ya que la etiquete que cierra *<SDML:Boss>* es diferente a *<Hull>* y tiene una menor profundidad.

Esta vez, sí que se entra en un caso del *switch case*: el caso de *<Hull>*:

Primero crearé una lista de objetos de la clase que tengan los cascos. Después, miraré la clase del siguiente objeto en salir de la pila y, a partir de ahí, se irán añadiendo a la lista objetos que se saquen de la pila hasta que la clase del siguiente sea diferente a las clases admitidas (en este caso, las pertenecientes a casco y a casco vital).

Cuando se acabe este bucle, se dará la vuelta a la lista para ponerla en el mismo orden que en el archivo *SDML*, y se generará el *array* con ella.

Este proceso será similar para los *arrays* de los enlaces, las armas, los objetos móviles y los cañones.

7.3.2. Cascos (Hulls)

Los cascos pueden tener dos tipos: vitales o normales, por lo que cuando se lea una etiqueta *<Hull>* y comience a procesarse, lo primero será mirar el tipo para crear una instancia de la clase pertinente a los cascos o una instancia de la clase pertinente a los cascos vitales, que heredará del casco. Una vez sepamos el tipo, el proceso siguiente será el mismo para ambos: la inicialización de todos los atributos que haya en la etiqueta.

A los cascos se les aplicará el modelo 3D indicado por el *SDML* y se colocarán en la posición indicada, con la escala y rotación también indicadas. También se harán hijos de otro casco si así estaba especificado en el *SDML* y los demás atributos se guardarán sin más repercusión.

7.3.3. Enlaces (Links)

Con los enlaces, de nuevo, se instanciará una clase u otra (heredando todas ellas de una clase base) dependiendo del tipo especificado en el atributo de la etiqueta. Cuando tengamos la instancia, se llamará a las inicializaciones propias de cada tipo, como en el caso de *<LinkRope>* y *<LinkMovable>*, los cuales tienen atributos extra.

Finalmente, se inicializarán todos los demás atributos, comunes en su existencia entre todos los enlaces.

Los enlaces tendrán una referencia a los dos cascos que unan y, en el caso del *LinkRope*, tendrá además una línea trazada entre ambos cascos.

7.3.4. Armas (Weapons)

El caso de las armas es como mezclar los cascos y los enlaces. Por un lado, se escogerá; dependiendo del tipo de arma especificado; una clase para instanciar u otra (todas heredando de una clase base, de nuevo) y se llamará a las inicializaciones propias de cada tipo. Una vez hecho esto, se llamará a todos los inicializadores comunes y se dará valor a todas las variables de las que disponen las armas.

Las armas en sí tendrán la referencia al casco al que pertenezcan, pero los modelos 3D y posiciones locales serán individuales en cada objeto móvil que pertenezca al arma. También los datos sobre turnos y posición y dirección de disparo estarán en cada cañón individual.

7.3.5. Unión de todo el sistema: jefe

La instancia de la clase del jefe tendrá referencias a todo lo demás, además de todos los parámetros que en la versión de *XML* contenía la etiqueta <GENERAL> y ahora contiene la propia etiqueta <SDML:Boss>.

7.3.6. Avatar del usuario

Una vez cubierto el diseño de todas las partes del jefe, falta hacer que el visualizador sea interactivo para poder ver apropiadamente todos los detalles del jefe.

Mi primer acercamiento fue simplemente poder rotar con el ratón el jefe y acercarlo o alejarlo, pero al reunirme con mi tutor y hablar de este diseño, llegamos a la conclusión de que la mejor forma de visualizar los jefes de un videojuego de naves espaciales es desde una nave espacial.

Tras diseñar la nave para que se moviera aplicando aceleración sobre la velocidad y la velocidad sobre la posición, pensé que quizás no estaba muy claro hacia dónde se movía el usuario al tener como única referencia el jefe. Por ello, decidí que la mejor opción era acompañar el movimiento de partículas que quedaran en el espacio al mover la nave, para saber la dirección de la aceleración al ver de dónde salían y la dirección de la velocidad al ver dónde quedaban con respecto a la nave.

También necesitaba una forma de decelerar la nave. Pensé en tres opciones para afrontar esto: la más común, decelerar con el tiempo; decelerar mediante la pulsación de un botón que te acelera en la dirección opuesta a la velocidad; o bajar la velocidad a cero en un tiempo al pulsar un botón. Como no quería complicar los controles ni las mecánicas al ser sólo el medio para utilizar el visualizador, me decanté por la última opción: siempre bajar la velocidad a 0 en un tiempo dado al pulsar un botón.

Finalmente quedaba el giro de la nave, para el cual también tenía varias opciones: hacerlo mediante teclas; pulsando el ratón y girándolo; o, el más común, manteniendo el ratón en el centro de la pantalla y girando conforme el ratón se moviera. Esta última opción es la más obvia para el usuario, por lo que me decanté por ella. También quería dejar claro que el jugador estaba girando cuando esto ocurriera, por lo que decidí también que la cámara tendría un leve efecto de muelle; lo cual potenciaría a su vez la sensación de movimiento ya ofrecida por las partículas.

7.3.7. Menú

Era necesaria una interfaz para ofrecer la posibilidad de modificar cualquier parámetro de control, poder cambiar el modelo que se está viendo a otro, o poder salir de la aplicación en la versión de PC.

Decidí dividir la pantalla de menú en 3 franjas: La primera con acciones que modifiquen la ejecución de la aplicación (reiniciándola o acabándola), la segunda con acciones que repercutan sobre la ejecución actual de la aplicación (continuar con la ejecución quitando la pausa o cambiando el jefe que se estaba visualizando), y la tercera con opciones para modificar los parámetros de control para el usuario.

En la primera franja incluí un botón para reiniciar la aplicación y el botón de salir (en la versión de PC).

En la segunda franja el botón de continuar y dos botones para selección del modelo de jefe actual.

En la tercera franja, un botón deslizable para modificar la máxima velocidad de la nave-cámara del usuario, otro para la aceleración, otro para la sensibilidad del ratón (que influirá al girar la cámara), y un botón normal para devolver esos valores a los predeterminados. También en esta franja estaría incluido un botón para silenciar el audio (el cual es, únicamente, música de *Kromaia* (Kraken Empire, 2014)).

Todo esto deberá estar encabezado por un título que especifique que estamos en el menú de pausa. Estando todo el texto del menú en inglés, el diseño sería algo parecido a lo que podemos ver en la ilustración 21.

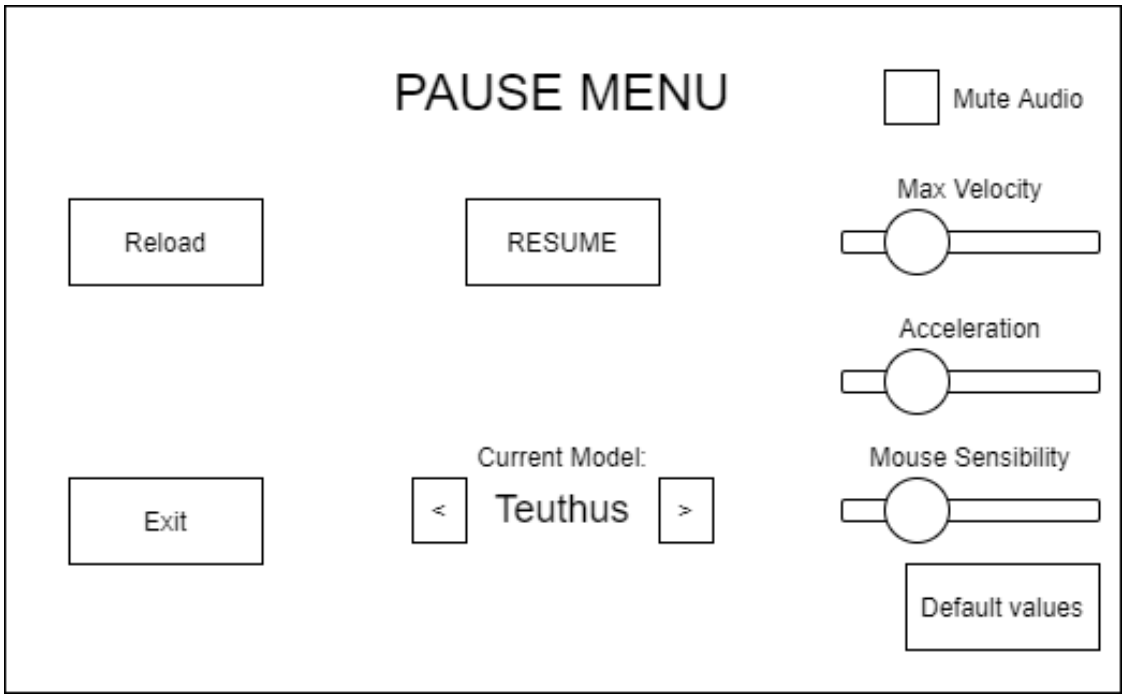

Ilustración 21: Diseño del menú de pausa del visualizador de modelos de jefes de Kromaia. Fuente: Propia.

7.4. Implementación

Una vez todo el diseño e idea de cómo empezar quedan planteados, llega la hora de empezar a implementar todas las ideas poco a poco y comprobar si todo funciona como estaba previsto y, en caso de que no lo haga, pensar cómo modificar el diseño para que sea correcto.

7.4.1. Lectura de valores correcta y segura

En *SDML*, cuando un atributo está en su valor por defecto, no se añade la información al archivo. Véase, por ejemplo, la posición { 0, 0, 0 }; que significa que el objeto está en el origen de coordenadas del espacio, el punto en X = 0, Y = 0, y Z = 0. En ese caso, la posición no se refleja en el archivo *SDML* y eso es un problema cuando se espera que todos los valores estén al leer el archivo, como sí era el caso de *XML*. Por ello, decidí hacer una serie de métodos que me ayudaran a leer los valores de forma segura y devolviendo siempre un valor, con la posibilidad de ser un valor por defecto elegible.

Habrá un método para cada clase de atributo que pueda leerse del archivo. Todos seguirán la siguiente estructura: cada método tendrá una entrada para el valor leído y un valor por defecto que se podrá dar de forma opcional. Este valor por defecto, al ser opcional, tendrá también a su vez un valor por defecto definido por mí en la implementación. Al llamar al método, primero se comprobará si el valor leído es válido o no se ha podido leer nada. En caso de tener un valor no nulo, se transformará de texto a la clase pedida y se devolverá. Por otro lado, en caso de ser nulo, se devolverá el valor por defecto indicado o, si no lo hubiera, el valor por defecto especificado en la declaración del método como valor por defecto del valor por defecto.

El primero de estos métodos, que podemos ver en la ilustración 22, se utiliza para leer valores booleanos. El valor por defecto de su valor por defecto será "falso". Como podemos ver, únicamente comprueba si el valor leído del texto es nulo o no para, después, devolver el valor booleano conseguido al analizar gramaticalmente la cadena de texto introducida o, si esta fuera nula, el valor por defecto.

```
private bool ParseBool(string value, bool def = false)
{
    return value != null ? bool.Parse(value) : def;
}
```

Ilustración 22: Método ParseBool para leer un valor booleano de forma segura desde la información leída de un texto. Fuente: Propia.

El segundo de estos métodos, que podemos ver en la ilustración 23, se utiliza para leer valores que sean números enteros. El valor por defecto de su valor por defecto será "0". Sigue el mismo procedimiento que el anterior método para, de nuevo, devolver el valor conseguido al analizar gramaticalmente la cadena de texto introducida o, si esta fuera nula, el valor por defecto.

```
private int ParseInt(string value, int def = 0)
{
   return value != null ? int.Parse(value,
CultureInfo.InvariantCulture.NumberFormat) : def;
}
```

Ilustración 23: Método ParseInt para leer un valor entero de forma segura desde la información leída de un texto. Fuente: Propia.

El siguiente método, visible en la ilustración 24, es algo diferente, ya que lo utilizaremos cuando tengamos que leer una máscara de bits[17], que devolveremos en forma de número entero sin signo. El valor por defecto para el valor por defecto de este método será la máscara de bits con todos sus bits en "falso", es decir, todo ceros.

Su modus operandi es exactamente el mismo que hasta ahora, con la diferencia de que, en vez de analizar gramaticalmente la cadena de texto introducida, se llama a otro método: "*StringToUint()*", el cual tengo creado en una clase estática para utilidades misceláneas a la que he llamado "*MiscUtilities*".

```
private uint ParseUint(string value, uint def =
00000000000000000000000000000000)
{
   return value != null ? MiscUtilities.StringToUint(value) : def;
}
```

Ilustración 24: Método ParseUint para leer un valor entero sin signo de forma segura desde la información leída de un texto. Fuente: Propia.

Este método "*StringToUint()*", que se encuentra en la ilustración 25, recorre todos los caracteres del texto introducido añadiendo un 1 a una máscara de bits; que más tarde se devolverá como número entero sin signo; cada vez que el carácter leído en la iteración actual no sea igual a '0'.

17 Cuando un programa tiene un estado basado en múltiples valores booleanos, que pueden ser 1 (verdadero) o 0 (falso), una forma de guardar estos valores es mediante una máscara de bits, capaz de emplear cada uno de los 32 bits de un número entero para representar diferentes valores booleanos. Utilizar máscaras de bits permite usar operaciones que trabajen a nivel de bits para modificar bits particulares de la máscara o comprobar si un valor en particular es verdadero o falso (Janssens, 2015).

```
public static uint StringToUint(string s)
{
    char[] charArray   = s.ToCharArray();
    int    accumulated = 0;
    uint   value       = 0;
    for (int i = charArray.Length - 1; i >= 0 ; --i)
    {
        if (charArray[i] != '0')
        {
            value |= (uint) (1 << accumulated);
        }
        ++accumulated;
    }

    return value;
}
```

Ilustración 25: Método StringToUint para transformar un valor en forma de máscara de bits de texto a entero sin signo. Fuente: Propia.

El cuarto método es para leer los números en representación de coma o punto flotante y, como se puede ver en la ilustración 26, seguirá exactamente el mismo patrón, siendo muy similar al de los enteros ya que el valor por defecto de su valor por defecto también será 0; o, más específicamente para este caso, "0.0".

```
private float ParseFloat(string value, float def = 0.0f)
{
   return value != null ? float.Parse(value,
CultureInfo.InvariantCulture.NumberFormat) : def;
}
```

Ilustración 26: Método ParseFloat para leer un valor flotante de forma segura desde la información leída de un texto. Fuente: Propia.

El penúltimo método de esta lista cumple con el mismo procedimiento que hemos visto hasta ahora, pero; al ser para leer vectores tridimensionales; dividiendo el texto en tres partes. Se puede ver en la ilustración 27.

Por último, también necesito un método para leer cuaterniones, ya que es la forma en la que se representan las orientaciones (o como se denominan en *Unity* (Unity Technologies, 2005-2021), rotaciones). Este método puede verse en la ilustración 28 y seguirá el mismo procedimiento que el anterior método, pero dividiendo el texto en 4 para generar un cuaternión, en vez de en 3 para generar un vector tridimensional.

```
private Vector3 ParseVector3(string value, Vector3 def =
default(Vector3))
{
   if (value != null)
   {
      string[] scaleComponents = value.Split(',');
      if (scaleComponents.Length == 3)
      {
         return new Vector3(
                        float.Parse(scaleComponents[0],

CultureInfo.InvariantCulture.NumberFormat),
                        float.Parse(scaleComponents[1],

CultureInfo.InvariantCulture.NumberFormat),
                        float.Parse(scaleComponents[2],

CultureInfo.InvariantCulture.NumberFormat));
      }
   }

   return def;
}
```

Ilustración 27: Método ParseVector3 para leer un vector tridimensional de forma segura desde la información leída de un texto. Fuente: Propia.

```
private Quaternion ParseQuaternion(string value, Quaternion def =
default(Quaternion))
{
   if (value != null)
   {
      string[] scaleComponents = value.Split(',');
      if (scaleComponents.Length == 4)
      {
         return new Quaternion(
                        float.Parse(scaleComponents[0],

CultureInfo.InvariantCulture.NumberFormat),
                        float.Parse(scaleComponents[1],

CultureInfo.InvariantCulture.NumberFormat),
                        float.Parse(scaleComponents[2],

CultureInfo.InvariantCulture.NumberFormat),
                        float.Parse(scaleComponents[3],

CultureInfo.InvariantCulture.NumberFormat));
      }
   }

   return def;
}
```

Ilustración 28: Método ParseQuaternion para leer un cuaternión de forma segura desde la información leída de un texto. Fuente: Propia.

7.4.2. Adición de modelos 3D

Si el objetivo es visualizar los jefes que se modelen en *SDML* igual que se verían en el juego original *Kromaia* (Kraken Empire, 2014), hará falta que tengan los modelos 3D originales en vez de las formas geométricas que se habían empleado hasta el momento, como se puede ver en la ilustración 15.

Sabiendo esto y llegado a este punto, procedí a enviar un mensaje a Daniel Blasco solicitando los modelos originales de los jefes *Vermis* y *Teuthus*, los cuales enseguida me hizo llegar.

Estos modelos estaban en formato *wings*. Este formato se usa principalmente para el programa de modelado 3D de código abierto[18] *Wings 3D* (Gustavsson, Gudmundsson, & others, 2006-2021), pero no es soportado por Unity. Un formato habitual para Unity sería *fbx* (Autodesk, Inc., 2006-2021). Por ello, utilizando el programa *Wings 3D* (Gustavsson, Gudmundsson, & others, 2006-2021), intenté exportar los modelos en formato *fbx* (Autodesk, Inc., 2006-2021), pero no estaba disponible. En otros programas de modelado 3D de código abierto, como *Blender* (Roosendaal & others, 1998-2021), sí se puede exportar en formato *fbx* (Autodesk, Inc., 2006-2021), pero no se puede leer desde formato *wings*.

Para poder conseguir los modelos en el formato que necesitaba, primero convertí todos los modelos a formato *obj* (Wavefront Technologies, Consultado en 2021) en *Wings 3D* (Gustavsson, Gudmundsson, & others, 2006-2021) para, después, leer esos archivos ya en formato *obj* (Wavefront Technologies, Consultado en 2021) en *Blender* (Roosendaal & others, 1998-2021) y exportarlos en formato *fbx* (Autodesk, Inc., 2006-2021).

Una vez los tenía en este formato, los importé en *Unity* (Unity Technologies, 2005-2021) y, para poder acceder a ellos desde código de una forma más cómoda, creé *Prefabs*[19] a partir de cada uno de ellos. Estos *Prefabs* serían lo que más tarde buscaría desde código para poder visualizar los modelos necesarios para cada parte del jefe.

A parte de todos estos modelos, los cuales pueden verse en la ilustración 29, añadí también un *Prefab* por defecto; el cual sólo es una esfera; para los casos en los que se intente buscar un modelo no existente en la versión actual del visualizador, y un *Prefab* vacío para los casos en los que no haya modelo especificado. Este último *Prefab* lo hice debido a que, en principio, si no se escoge un modelo para una parte del jefe será porque no tiene modelo, pero durante el desarrollo

18 "El software open source es un código diseñado de manera que sea accesible al público: todos pueden ver, modificar y distribuir el código de la forma que consideren conveniente" (Red Hat, 1993-2021).

19 El Sistema de *Prefab* de *Unity* permite crear, configurar, y almacenar un *GameObject*; que es el objeto de representación fundamental en las escenas de *Unity*; completo con todos los componentes, propiedades, e hijos *GameObjects* para poder reutilizarlo, actuando como plantilla para cada vez que se quieran crear instancias del *Prefab* en la escena (Unity Technologies, 2018).

sí hice que en este *Prefab* hubiera un modelo de un cubo, con fin de saber que se estaba leyendo y generando todo correctamente.

Ilustración 29: Modelos necesarios para la visualización del jefe "Vermis" y el jefe "Teuthus" del juego Kromaia. Fuente: Propia.

7.4.3. Generación de los arrays de cascos, armas, y enlaces

En la versión de *SDML* es importante que cada elemento tenga el renderizado indicado: el modelo 3D que se especifica en el atributo "*ModelFileName*". Hay muchas formas de proceder para realizar esto, pero yo decidí hacer un método estático en la clase *Component*.

Como podemos ver en la ilustración 16, tanto casco como arma heredan de componente (*Component*), y después casco vital hereda de casco. Por ello decidí que los métodos para inicializar los datos del objeto (modelo 3D, escala, posición, y orientación) deberían estar en la clase abstracta[20] *Component*, excepto el método especial para el caso del casco vital y todos los métodos de los casos especiales de las armas.

Así pues, en la clase *Component* se encuentran la referencia al jefe, una referencia a una instancia de la clase *Rigidbody*, que pertenece al espacio de nombres *UnityEngine*, y los atributos que

20 "El modificador abstract indica que lo que se modifica carece de implementación o tiene una implementación incompleta. […] Use el modificador abstract en una declaración de clase para indicar que una clase está diseñada como clase base de otras clases, no para crear instancias por sí misma" (Microsoft Corporation, 2015).

tengan que ver con la vida del componente y sus colisiones; como, por ejemplo, la vida máxima, la armadura, el material, etc.

En cuanto a los métodos de *Component,* encontramos "*public void InitPhysicalObjectData()"* y *"public void InitAliveObjectData()",* que únicamente darán valores a diversos parámetros como la masa de *Rigidbody*, la vida, o la armadura. También está el método *"public void InitComponentData()",* que pide el nombre del casco padre (si lo hay) y la dirección local. En este método, se llamará en caso de recibir el nombre del padre el método de *MiscUtilities* "*public Hull ParseHullFromScene()"*, que se puede ver en la ilustración 30. En este método se busca en la escena activa un objeto con el nombre especificado y, cuando se encuentre, se devolverá su referencia de clase *Hull*. Una vez conseguida la referencia a *Hull*, el objeto al que pertenezca el componente actual en la escena se hará hijo del objeto al que pertenezca la referencia a *Hull* en la escena. Hecho esto, se recolocará la posición, la escala, y la dirección a su nuevo espacio local.

Después está el único método abstracto de la clase: "*public abstract void InitSpecific()"*. Este método servirá para inicializar el tipo del componente (en el caso de *Hull*, tipo normal o tipo vital; en el caso de *Weapon*, tipo proyectil, proyectil auto apuntado, proyectil singularidad, proyectil cargado, láser, etc.).

Para acabar, el método más importante de la clase *Component* es el método "*public T InitObjectData<T>()",* donde 'T' hereda de *Component*. Es el más importante ya que todas las clases que hereden de *Component* se crearán con este método desde el lector. Primero buscará entre los modelos que hay guardados un modelo con el nombre del modelo que se haya dado. Si no lo encuentra (porque el nombre sea erróneo o porque no esté incluido ese modelo en la versión del visualizador actual), se cargará el modelo por defecto (que es una esfera). Una vez se instancie un objeto con ese modelo y siendo hijo del jefe, se le dará una referencia a una nueva instancia de la clase *T* que se haya indicado al llamar al método. Sólo entonces, se guardará la referencia al jefe y se colocará en la posición indicada con la orientación y escala indicados. Finalmente se guardará una referencia a una nueva instancia de la clase *Rigidbody* y se devolverá la referencia de la instancia de la clase indicada *T*.

Entrando ya en casos particulares, *Hull* tendrá como añadido el tipo de casco y *HullVital* tendrá también un entero que represente la capa de casco vital. *Weapon*, por su lado, será otra clase abstracta que tendrá bastantes más añadidos: todos los valores *float*, *int,* y *Vector3* indicados en el *SDML,* una referencia al *array* de todos los cañones, y una referencia al *array* de todos los objetos móviles. En sus métodos de inicialización sólo se llenarán los parámetros, dejando los dos *arrays* para más adelante; como se ha explicado en el apartado 8.3.1.

Las clases herederas de *Weapon* tendrán cada una, igual que *Hull* y *HullVital*, sus propias versiones de "*public override void InitSpecific()"* y nada nuevo, a excepción de

WeaponProjectileLoaded y *WeaponLaser*, que tendrán también varios parámetros extra que habrán de añadirse con los métodos "*public void InitWeaponProjectileLoadedData()*" y "*public void InitWeaponWithLasers()*", respectivamente.

```
public static Hull ParseHullFromScene(string hullInfoParsed)
{
    // Receives something like @Hull.1
    string[] firstHullInfo = hullInfoParsed.Split('@');
    firstHullInfo = firstHullInfo[1].Split('.');
    System.Type type                    =
System.Type.GetType(firstHullInfo[0]);
    GameObject  firstHullGameObject =
GameObject.Find(firstHullInfo[1]);
    if (firstHullGameObject == null)
    {
        string error =
            $"Error while parsing hull type and name. Result = [type =
{firstHullInfo[0]}] [name = {firstHullInfo[1]}] - GameObject with that
name does not exist in the scene.";
        return null;
    }
    else
    {
        Hull hull = (Hull) firstHullGameObject.GetComponent(type);
        if (hull == null)
        {
            string error =
                $"Error while parsing hull type and name. Result =
[type = {firstHullInfo[0]}] [name = {firstHullInfo[1]}] - GameObject
does not contain that type.";
            Debug.LogError(error);

            return null;
        }
        return hull;
    }
}
```

Ilustración 30: Método ParseHullFromScene para devolver un objeto de clase Hull que haya en la escena. Fuente: Propia.

Los cañones, en su clase *WeaponCannon* tendrán un método "*public void SetParent()*", que hará al objeto en el que esté esta instancia en la escena hijo del arma a la que pertenezca; y un método "*public void InitWeaponCannonData()*", que únicamente inicializará parámetros.

Por su parte, los objetos móviles, en su clase *MobileObject* tendrán un método "*public void SetParent()*" muy similar al de *WeaponCannon* y un método estático "*public static MobileObject InitMobileObject()*" que funcionará de forma muy similar a "*public T InitObjectData<T>()*" en la

clase *Component*, buscando el modelo 3D para luego al objeto instanciado darle una referencia a una nueva instancia de *MobileObject*, que será la que se utilice.

Finalmente, los enlaces parten de la clase abstracta *Link*, que tiene todos los parámetros relativos a su destrucción y a los cascos que une, su tipo de enlace, una referencia al jefe, otra a una instancia de la clase *Rigidbody*, y una referencia a un *array* de *Joints;* los cuales son otra clase del espacio de nombres *UnityEngine;* que unirán los cascos al enlace. En lo relativo a métodos, comienza con "*public virtual void InitData()"*, la cual guarda la referencia al jefe y hace que este sea padre del objeto que contenga la referencia al enlace actual en la escena, busca los cascos que tenga que unir en la escena volviendo a emplear "*ParseHullFromScene()"*, que podemos ver en la ilustración 29, los conecta llamando a su otro método "*protected void ConnectToHulls()",* y acaba aplicando los valores leídos del archivo *SDML* a los parámetros de la clase.

En "*protected void ConnectToHulls()"* se conectan las instancias que estén referenciadas de *Rigidbody* en las instancias de *Hull* a unir a las instancias de *Joint* que hay en la clase, para así poder aplicar las fuerzas correctamente con el sistema de físicas que utiliza *Unity* (Unity Technologies, 2005-2021).

Finalmente, *Link* acaba con el método abstracto "*public abstract void InitSpecific()"*, que; igual que en el caso anterior de *Component*, se deberá completar en las clases herederas de *Link*: *LinkFixed, LinkMovable, y LinkRope.* En este método se crearán también las instancias de *Joint* para el array de *Joint*s.

Al igual que en el caso de algunas armas, *LinkMovable* tiene un método extra, "*public void InitMovableData()",* que dará el valor a todos los parámetros que este tipo de enlace tiene añadidos y *LinkRope* tiene un método similar, "*public void InitRopeData()"*. *LinkRope* también tiene la peculiaridad de tener una referencia a una instancia de la clase *LineRenderer* del espacio de nombres *UnityEngine* y un *override* de "*InitData()"* que, tras llamar al método base, aplicará a su referencia de *Rigidbody* la masa que deberá tener la hipotética cuerda y dará valor a los diferentes apartados de *LineRenderer* para poder visualizarla correctamente.

7.4.4. Complicación con enlaces y físicas

Llegados a este punto, el jefe leído desde el archivo *SDML* se podía visualizar correctamente, pero si quería que tuviera colisiones o algún tipo de movimiento simple, tendría que hacer que los enlaces funcionaran con físicas siguiendo las bases indicadas en los parámetros del *SDML*.

Cuando probé por primera vez a hacer que funcionaran con los parámetros indicados, las físicas no funcionaban correctamente, haciendo que las piezas se movieran solas por las propias fuerzas que ejercían unos enlaces sobre otros a cada vez mayor velocidad, hasta hacer que se perdiera de vista el jefe.

Decidí reunirme con Daniel Blasco para ver cómo afrontar este problema, como se puede ver en anexos, quien me recibió enseguida y me explicó que probablemente este fuera un problema que me llevaría demasiado tiempo solucionar. Aun así, decidimos que lo intentaría con una serie de pruebas antes de darme por vencido en este apartado. Estas pruebas consistieron en intentar aplicar la masa de diferentes formas y en modificar los parámetros de cada tipo de *Joint* para intentar ajustarlo a algo controlable.

Finalmente, tras invertir en este apartado tres veces el tiempo que previ sin llegar ni acercarme a ningún resultado satisfactorio, decidí junto a mi tutor que; siendo este un apartado tangencial en el trabajo; me centrara en los siguientes apartados; los cuales sí eran bastante importantes para el visualizador.

7.4.5. Movimiento para el usuario

Siguiendo el diseño decidido, el movimiento se compondrá de dos archivos de código diferentes: *ShipMovement.cs* y *SpringCamera.cs*.

ShipMovement.cs se encargará de controlar el cambio de posición del usuario y las partículas de aceleración. Para el cambio de posición, tiene una serie de valores que le harán calcularla: tres números flotantes que indiquen la aceleración, velocidad máxima, y tiempo para estabilizar (frenar), dos vectores tridimensionales que indican la velocidad actual y la velocidad con la que se ha comenzado a estabilizar, un *array* con las referencias a las 6 instancias de los sistemas de partículas, y un booleano que indica si se está utilizando algún sistema de partículas.

Para controlar el movimiento, el método "*private void Move()*" comprobará que la velocidad no sea superior a la máxima y moverá la posición de la nave que controla el usuario en función de la velocidad actual.

Para controlar a estabilización, si se presiona la barra espaciadora se llamará al método "*private void Stabilize()*", que, siempre que la velocidad sea mayor que 0, comenzará a estabilizar la nave decrementando la velocidad hasta 0 en el tiempo dado.

Cuando no estemos pulsando la barra espaciadora, se comprobarán el resto de las entradas (que serán las teclas 'W', 'A', 'S', 'D', 'Q', y 'E') en "*private Vector3 GetInput()*" para aumentar la velocidad dependiendo de la dirección que se haya dado con la entrada y de la aceleración indicada en "*private void Accelerate()*".

Finalmente, el funcionamiento de las partículas se controlará con el método "*private void UpdateParticles()*".

Por otro lado, "*SpringCamera.cs*" se encargará de la rotación de la nave y el movimiento de la cámara. Esta clase dispone de una serie de parámetros para cumplir con ambas funciones.

En lo relativo al movimiento de la cámara tendremos dos vectores tridimensionales a parametrizar que serán la compensación de la posición base (dónde estará la cámara respecto a la nave) y la compensación del punto hacia al que mirará la cámara con respecto a la nave, para los cuales también necesitamos; y tenemos; una referencia a la nave. También tiene tres números flotantes que servirán para el cálculo del movimiento de muelle: la fuerza del muelle, la amortiguación, y la distancia máxima desde la posición ideal.

En lo relativo a la rotación de la nave tendremos la cantidad que giraremos la cámara al mover el ratón, que podría entenderse como sensibilidad del ratón, y una referencia a la cámara.

La rotación de la nave se controla con el método "*private void RotateTarget()*". Este método captura la dirección en que estemos moviendo el ratón y rota la nave dependiendo de esta y de la sensibilidad actual.

El movimiento de muelle de la cámara se controla desde "*private void SpringMovement()*". Aquí, lo primero que se hace es calcular la posición del punto hacia el que queremos que la cámara mire y la posición ideal en espacio de mundo. Después se calcula la amortiguación con respecto a la velocidad que tenía la cámara en este momento y se calcula la fuerza del muelle siguiendo la siguiente ecuación de la ley de Hooke (Fernández & Coronado, Consultado en 2021):

Ecuación 1: Fuerza que se aplica sobre el muelle según la Ley de Hooke. Fuente: FisicaLab

$$F = k \cdot (x - x_0)$$

Sumando la amortiguación y la nueva aceleración del muelle a la velocidad, podemos calcular la posición a la que se moverá la cámara. Entonces se comprueba que no se supere la máxima distancia desde la posición ideal y, en caso de hacerlo, se restringe a esa distancia máxima. Una vez hecho esto, se cambia la posición de la cámara a la nueva posición calculada y se rota para mirar al punto con respecto a la nave al que se haya especificado que mire.

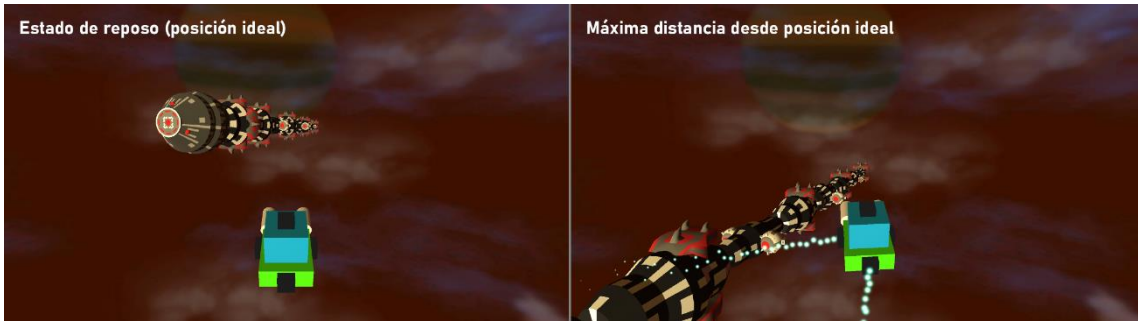

Ilustración 31: Estados de la cámara con muelle del visualizador. Fuente: Propia.

7.4.6. Menú

Al darse la situación de ser un proyecto relativamente pequeño con sólo un menú de pausa, este menú será controlado por una clase *GameManager* que controlará otros aspectos del estado de la aplicación como la escala del tiempo (al pausar y continuar), qué archivo *SDML* se está visualizando en este momento o qué botones se mostrarán en la versión de PC y de WebGL.

Esta clase es un *singleton*[21], por lo que tiene una referencia estática a una instancia de sí misma y lo primero que hará al instanciarse será comprobar si esta referencia está instanciada ya. En caso de no estarlo, se guardará en ella a sí misma, pero en caso de estarlo ya, la instancia actual se destruirá.

Una vez realizada esta comprobación, generará las rutas a los archivos *SDML*. Esto lo hará en su método "*private void GeneratePaths()*", que veremos más adelante ya que realizará unas acciones u otras dependiendo de si estamos en la versión para PC o para WebGL.

Esta clase tiene también una referencia al lector para la representación del jefe actual, por lo que después de esas acciones inicializará dicho lector y llamará a su método "*public override void ProcessFile()*", el cual también veremos más adelante, debido a que también realizará unas acciones u otras dependiendo de si estamos en la versión para PC o para WebGL.

El estado de la pausa se manejará con un valor booleano llamado "*m_Paused*", que será verdadero cuando el estado del programa esté pausado y falso cuando no. Este valor empezará la ejecución en falso, y eso conllevará ciertas acciones: el cursor del ratón se bloqueará en el centro de la pantalla, la escala de tiempo será igual a 1 (lo que significa que el tiempo del juego pasará normalmente), y el menú se desactivará.

Finalmente, desactivará unos botones u otros del menú dependiendo de si estamos en la versión de PC o de WebGL.

A partir de aquí, no realizará ninguna acción a no ser que se pulse la tecla de pause (que será por defecto 'P', con la posibilidad de funcionar también con "Escape"), ya que entonces llamará a su método "*public void PauseUnpause()*", el cual invertirá el estado actual de "*m_Paused*" y realizará ciertas acciones dependiendo de su nuevo valor: si es falso, desactivará el menú, dará valor a la escala de tiempo igual a 1, y bloqueará el cursor en el centro de la pantalla. Por otro lado, si fuera verdadero, activará el menú, dará valor igual a 0 a la escala de tiempo (con lo que el tiempo del juego se detendrá), desbloqueará el cursor, y realizará los ajustes necesarios para que el menú funcione correctamente.

21 S*ingleton* o instancia única es un patrón de diseño que permite restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella (Gupta, 2012).

Aparte de esto tendrá la mayor parte de las acciones de las botones repartidas en diferentes métodos:

El botón "*Reload*" llamará al método "*public void Reload()*", el cual simplemente recargará la escena de nuevo con un método estático de *SceneManager*, una clase del espacio de nombres *UnityEngine.SceneManagement*.

El botón "*RESUME*" llamará al método "*public void PauseUnpause()*", igual que las teclas 'P' y 'Escape', como hemos visto previamente.

El botón "*Mute Audio*" llamará al método "*public void MuteAudio()*", el cual cambiará a su opuesto el valor booleano que controla si el audio está silenciado en la fuente de audio, de la que tiene una referencia la clase *GameManager*.

Los botones de cambio de modelo actual llamarán al método "*public void ChangeModels()*". Este método pondrá en funcionamiento la animación que cambiará el nombre del modelo por el del modelo localizado siguiente, pero lo importante en ella serán sus otras acciones: destruirá el jefe actual e instanciará otro jefe vacío. Inicializará el lector y pondrá a funcionar un contador que, cuando acabe, llamará a "*ProcessFile()*" en el lector para procesar el nuevo archivo *SDML*. Esta acción se hace tras esperar a que el contador acabe debido a que en caso contrario puede haber conflictos con el jefe que se acaba de destruir al buscar cascos u otros elementos en la escena y no haber dado suficiente tiempo para que se procese la destrucción del objeto correctamente.

Finalmente, el botón "*Default values*" y los tres botones deslizantes "*Max Velocity*", "*Acceleration*", y "*Mouse Sensibility*" llamarán a los métodos pertinentes de cambio de velocidad, aceleración, y sensibilidad del ratón en las clases *ShipMovement* y *SpringCamera*.

7.4.7. Carga de distintos modelos en PC

En el caso de la versión de PC, todos los archivos *SDML* estarán en una carpeta con nombre "SDML", la cual se encontrará en el mismo lugar que la aplicación del visualizador.

Para tener las rutas a todos los archivos *SDML* hemos visto que se llama al método "*private void GeneratePaths()*" en la clase *GameManager*.

En este método, la primera acción que se hará para tener la referencia a todos los archivos *SDML* que haya en la carpeta será comprobar si dicha carpeta existe. En caso positivo, se cargará el nombre de todos los archivos que haya en la carpeta para, después, quedarnos sólo con los que tengan la extensión "*.sdml*".

Podemos ver el proceso en el retazo de código de la ilustración 32.

```
// Desktop
string startingPath = "." + Path.DirectorySeparatorChar + "SDML";
if (Directory.Exists(startingPath))
{
   string[]     allFiles = Directory.GetFiles(startingPath);
   List<string> validFiles = new List<string>();
   foreach (string file in allFiles)
   {
      if (file.EndsWith(".sdml"))
      {
         validFiles.Add(file);
      }
   }

   m_AllFilePaths = validFiles.ToArray();
}
```

Ilustración 32: Retazo de código para generar las rutas de los archivos SDML. Fuente: Propia.

Una vez tengamos las rutas generadas, cuando se llame al método del lector; cuya clase es *SDMLReader*; "*public override void ProcessFile()*" esta función llamará al método de la misma clase *SDMLReader* "*private void ReadSdmlFile()*", a la cual habrá que pasarle la ruta al archivo *SDML* actual. Esta ruta la podremos coger de *GameManager*, que ya deberá haber generado las rutas al instanciarse y tiene también guardado qué archivo es el actual que estamos leyendo.

En "*private void ReadSdmlFile()*", se utilizará una instancia de la clase *XmlTextReader* del espacio de nombres *System.Xml* para leer después en el bucle que podemos ver en la ilustración 33 (que recorrerá el archivo *SDML* organizando las clases que se vayan instanciando con ayuda de una pila) el archivo *SDML* con el modelo que queremos visualizar. Para saber qué texto debe leer, deberemos especificar en el constructor de *XmlTextReader* la ruta al archivo o una instancia de la clase *Stream* del espacio de nombres *System.IO* que tenga la información. Dado que ya tenemos la ruta, será lo que se use.

En la ilustración 33 también podemos observar que hay directivas de compilación condicional[22] que harán que ciertas partes del código se compilen en la versión para PC y otras en la versión para WebGL. Estas directivas se repetirán en cada parte de código en la que se dé la situación de diferentes necesidades según la plataforma.

```
#if UNITY_STANDALONE_WIN
   private void ReadSdmlFile(string path)
   {
      XmlTextReader reader = new XmlTextReader(path);
#else
   private void ReadSdmlFile(byte[] bytes)
   {
      // Web
      Stream        stream = new MemoryStream(bytes);
      XmlTextReader reader = new XmlTextReader(stream);
#endif

      m_LastElementName = "";
      m_LastDepth       = 0;

      Stack<object> stack = new Stack<object>();

      float startingTime = Time.realtimeSinceStartup;

      while (reader.Read())
      {
         switch (reader.NodeType)
         {
            case XmlNodeType.Element:
               PushElement(reader, stack);
               break;
            case XmlNodeType.EndElement:
               PopElement(reader, stack);
               break;
            default:
               if (reader.NodeType != XmlNodeType.Whitespace &&
reader.NodeType != XmlNodeType.Comment && reader.Name != "xml")
               {
                  Debug.Log($"EXCEPTION: Node {reader.NodeType} ->
\"Name\": {reader.Name}");
               }
               break;
         }
      }

      Debug.Log("Read and built in game in " +
(Time.realtimeSinceStartup - startingTime) + " seconds.");
   }
```

Ilustración 33: Retazo de código del método ReadSdmlFile() de la clase SDMLReader. Fuente: Propia.

22 "Cuando el compilador de C# encuentra una directiva #if, seguida en última instancia por una directiva #endif, compila el código entre las directivas solo si se ha definido el símbolo especificado" (Wagner & "olprod", 2021).

7.4.8. Carga de distintos modelos en WebGL

Para la versión de WebGL, todos los archivos *SDML* estarán subidos en un servidor accesible y en la misma ruta que el código *PHP* que se puede ver en la ilustración 34. Para este proyecto, dicha ruta será "https://svit.usj.es/kromaia/visualizer/SDML/".

El *PHP* de la ilustración 34 mostrará al entrar una lista con los nombres de todos los archivos que se encuentren en la carpeta separados por comas. Cuando queramos recibir el texto con el modelo contenido en uno de estos archivos, deberemos acudir a la ruta con un método *HTTP* "*GET*"[23] en el que especifiquemos el modelo que buscamos.

Como hemos visto antes, para tener las rutas a todos los archivos *SDML* se llama al método "*private void GeneratePaths()*" en la clase *GameManager*.

En este método las acciones serán diferentes ahora que se trata de compilación para WebGL. Esta vez, lo único que se hará en este método es empezar una corrutina[24] con el método "*public Coroutine StartCoroutine()*" de la clase *MonoBehaviour* de *Unity* (Unity Technologies, 2005-2021), al que hay que especificar qué método se desea ejecutar en la corrutina. Este método, será el método de *GameManager* "*private IEnumerator LoadPathsFromWeb()*".

El método "*private IEnumerator LoadPathsFromWeb()*" crea una petición web con el método *HTTP* "*GET*" a la ruta "https://svit.usj.es/kromaia/visualizer/SDML/" y lo envía. Cuando recibe el resultado, al no indicar ningún modelo, este será el nombre de todos los archivos que se encuentran en esa ruta separados por comas, por lo que se comprueba la extensión de cada uno y se guardan sólo los archivos *SDML* (con extensión "*.sdml*"). Los nombres de estos archivos serán lo que se guarde como "rutas" en la versión para WebGL, ya que habrá que especificar el nombre del archivo de los modelos a la misma ruta que ya hemos llamado para recibir los modelos.

También, al final de este método, se creará un *array* de *arrays* de *bytes*, en el que guardaremos la información de cada modelo una vez lo hayamos buscado una vez, para acortar esperas y evitar más llamadas web de las necesarias.

Mientras tanto, en la clase *SDMLReader*, el método "*public override void ProcessFile()*" también realizará una acción diferente a la que realiza en la versión de PC. Esta vez llamará también al método "*public Coroutine StartCoroutine()*" para ejecutar el nuevo método de *SDMLReader* "*private IEnumerator LoadModelFromweb()*". Este método esperará sin actuar hasta que el

23 "HTTP define un conjunto de métodos de petición para indicar la acción que se desea realizar para un recurso determinado. […] El método GET solicita una representación de un recurso específico. Las peticiones que usan el método GET sólo deben recuperar datos" (Mozilla and individual contributors, 2005-2021).

24 "Una corrutina es una función que tiene la habilidad de pausar su ejecución y devolver el control a Unity para luego continuar donde lo dejó en el siguiente *frame*" (Unity Technologies, 2016).

singleton de *GameManager* tenga generados todas las rutas (en este caso, los nombres de los archivos con los modelos). Entonces, comprobará si tiene ya guardado el modelo que quiere cargar. En caso positivo, se procederá a llamar a la versión de WebGL de "*private void ReadSdmlFile()*", que se puede también ver en la ilustración 32. En caso negativo, se creará la petición con el método *HTTP* "*GET*" especificando el modelo que queremos recibir. Una vez el servidor responda, se guardarán los *bytes* recibidos a no ser que haya habido un error (en cuyo caso se mostrará el error por consola).

```
<?php
	header('Access-Control-Allow-Origin: *');
	header('Access-Control-Allow-Headers: *');
	 header('Access-Control-Allow-Methods: GET');
	$path	= './';
	if (isset($_GET['model']))
	{
		echo file_get_contents($path.$_GET['model']);
	}
	else
	{
		$files = array_diff(scandir($path), array('.', '..', 'index.php'));
		$html = '';
		foreach ($files as $file)
		{
			$html.= $file.',';
		}
		echo $html;
	 }
?>
```

Ilustración 34: Código PHP para generar la información y rutas de los archivos SDML para la versión de WebGL. Fuente: Propia.

Finalmente, se llamará también a la versión de WebGL de "*private void ReadSdmlFile()*", que recibirá el *array* de *bytes* guardados y creará una instancia de *Stream*, la cual a su vez recibirá el *array* de *bytes* recibido en su constructor y se utilizará también en el constructor de una nueva instancia de la clase *XmlTextReader* para, de igual forma a antes, leer después en el bucle; que podemos ver, de nuevo, en la ilustración 33; el archivo *SDML* con el modelo que queremos visualizar.

7.4.9. Últimos detalles

Teniendo ya la aplicación funcionando en ambas plataformas; PC y WebGL (para el cual hay una versión funcionando en https://svit.usj.es/kromaia/visualizer/); necesitaba una forma de presentarlo para el uso público ya que recordemos que este proyecto es un visualizador para el grupo de investigación SVIT (SVIT Research Group, 2011-2021) con el fin de poder visualizar los modelos creados en *SDML* de forma sencilla.

Debido a esto, creé una imagen utilizando el programa Adobe Photoshop (Adobe Inc., 1990-2021) en la que explico los controles. Dicha imagen se puede ver en la ilustración 35 y especifica con qué teclas la nave podrá moverse, estabilizarse, y rotar; y con qué tecla abrir el menú y liberar el ratón.

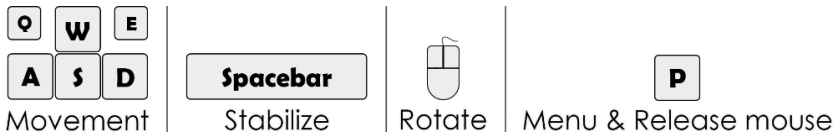

Ilustración 35: Controles del visualizador. Fuente: Propia.

La página web con el visualizador se puede apreciar en la ilustración 36.

Ilustración 36: Página web con la versión WebGL del visualizador. Fuente: Propia.

8. Resultados

El visualizador, en principio, parecía funcionar perfectamente en relación con lo que se buscaba, pero sólo se había probado el jefe "*Vermis*" en él, por lo que ahora habría que ponerlo a prueba con el archivo *SDML* del jefe "*Teuthus*".

De todas formas, aunque funcionara también correctamente con este jefe, serían dos modelos diseñados por un profesional. Si de verdad quería ver los resultados de mi visualizador, habría que probar con modelos hechos por gente novata en el modelado *SDML*, para poder llevar al límite mi visualizador y comprobar que siguiera funcionando correctamente.

Por ello, este capítulo de resultados se verá compuesto por dos apartados: "modelos reales" y "modelos de experimento".

8.1. Modelos reales

Tras el éxito de la visualización del jefe "*Vermis*", que se puede ver en la ilustración 3, probé a cargar el modelo del jefe "*Teuthus*". El resultado, que se puede ver en la ilustración 37, fue exitoso. Este modelo también se había cargado perfectamente y tal y como estaba previsto.

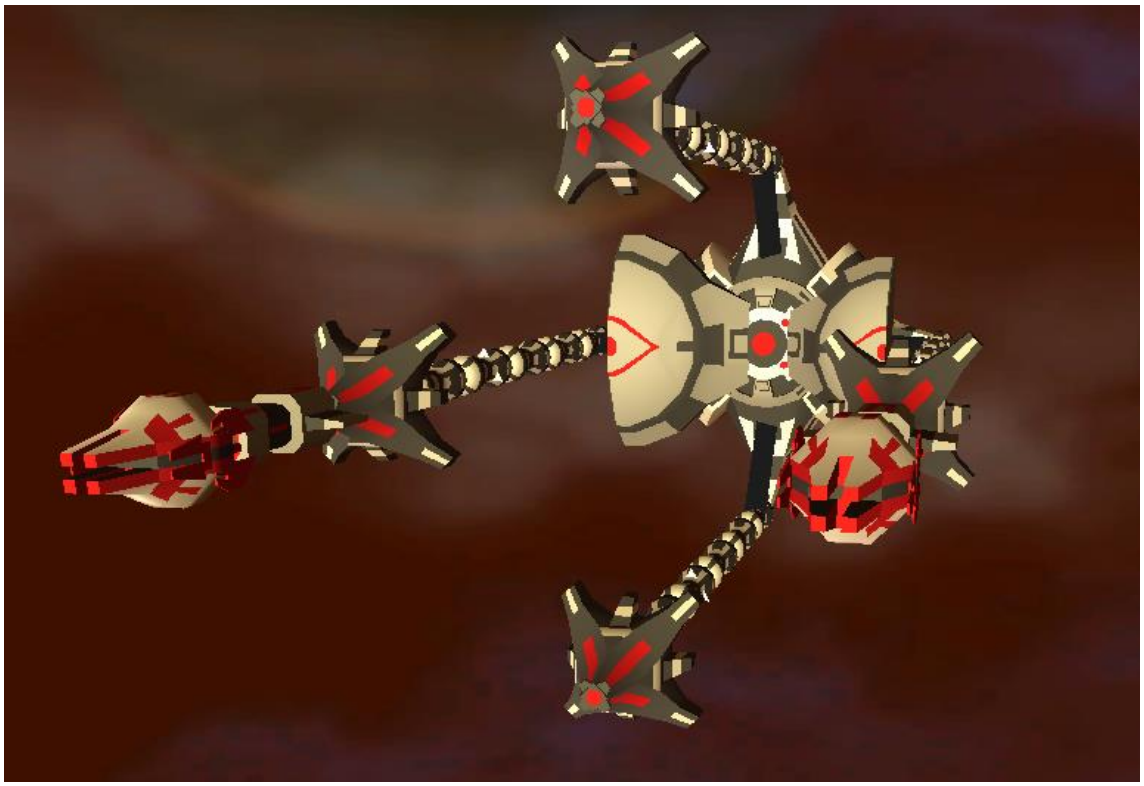

Ilustración 37: Representación del jefe "Teuthus" en el programa desarrollado en este trabajo. Fuente: Propia.

8.2. Modelos de experimento

Con fin de descubrir las debilidades del visualizador, realicé un experimento junto a Rodrigo Casamayor, Jorge Chueca, y Jaime Font en el que 32 alumnos de la Universidad San Jorge divididos en dos grupos para dos días diferentes se presentaron como voluntarios para utilizar la herramienta de diseño y crear el modelo de un jefe para *Kromaia* (Kraken Empire, 2014) inventado por ellos en el momento. Se pueden ver imágenes de este experimento en anexos.

El experimento consistía en dos partes:

En la primera parte, los voluntarios tenían que seguir unas indicaciones para generar un jefe con las especificaciones que les pedíamos para que comenzaran a desenvolverse con el programa. Durante la segunda parte, tenían que utilizar la base que habían creado en el anterior ejercicio para diseñar un jefe con total libertad.

Para el apartado del visualizador, el experimento fue un éxito, dando un único problema en los elementos *Hull*: si no se especificaba el tipo de casco, o se especificaba como casco normal, y luego ese casco tenía elementos únicos de casco vital, el lector fallaba. Este problema se solucionó cuando se encontró haciendo que si el casco se identifica como normal, no se lean los elementos de casco vital aunque existan.

El resultado puede considerarse fiable ya que los 15 primeros de los 32 alumnos pertenecían a 2º curso de Ingeniería Informática, mientras los otros 17 alumnos pertenecían a 3er curso de Diseño y Desarrollo de Videojuegos, que son los dos ámbitos en los que se mueve el trabajo.

Los resultados son visualmente variados, y se pueden ver todos en la copia del visualizador que hay subida en https://svit.usj.es/kromaia/visualizer-experiment/, donde se leen los modelos diseñados y generados por los voluntarios.

De todos estos modelos, Rodrigo Casamayor, Jorge Chueca, y yo hemos concluido que los 9 con más sentido visual y conceptual son los siguientes: "*Model02*", "*Model06*", "*Model15*", "*Model16*", "*Model17*", "*Model22*", "*Model23*", "*Model26*" y "*Model29*". Estos modelos se pueden ver en la ilustración 38.

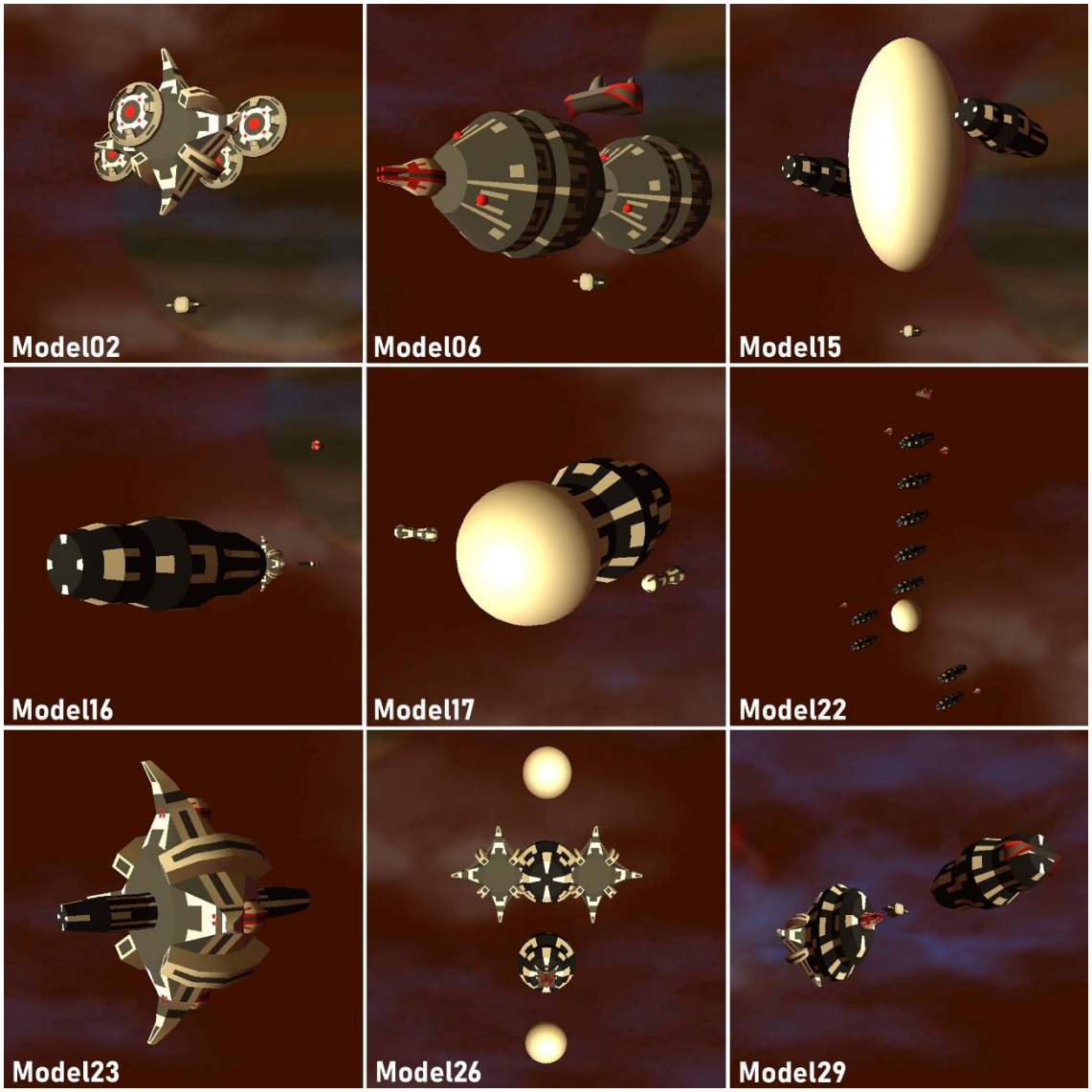

Ilustración 38: Visualización de modelos correctos visualmente, de acuerdo con la opinión de R. Casamayor, J. Chueca, y J. Verón, del experimento llevado a cabo entre el día 19/05/2021 y el día 24/05/2021. Fuente: Propia.

9. Conclusiones

Me habría gustado tener un visualizador capaz de poder funcionar también de acuerdo con las físicas de *Kromaia* (Kraken Empire, 2014), con los enlaces uniendo correctamente los cascos, y haciendo al jefe capaz de moverse en el espacio de la escena. Sin embargo, el tiempo para el desarrollo de este proyecto es un recurso limitado y, por ello, he debido centrarme en lo realmente importante para el proyecto: la visualización correcta de los modelos 3D en el espacio y la multiplataforma ofrecida al emplear un motor comercial como *Unity* (Unity Technologies, 2005-2021).

La principal consecuencia que veo sobre este proyecto es la enorme ventaja que va a suponer en la creación de modelos *SDML* tanto para para SVIT (SVIT Research Group, 2011-2021), como para futuros desarrolladores de este tipo de juegos, ya que hasta ahora no había una forma sencilla de ver cómo había quedado un modelo o cómo iba progresando, y con esta herramienta se podrá ver de forma casi inmediata, tanto a través de la aplicación para PC como a través de la aplicación para WebGL, con la comodidad que este último implica al permitir visualizar desde el navegador sin la necesidad de instalar ni descargar nada.

10. Bibliografía

Adobe Inc. (1990-2021). *Adobe Photoshop pricing and membership plans | Adobe*. Obtenido de Adobe Photoshop: https://www.adobe.com/products/photoshop/pricing-info.html

Allen, D. (2001). *Getting Things Done.* Ojai, California, Estados Unidos: Penguin Books. Obtenido de https://gettingthingsdone.com

Ampatzoglou, A., & Stamelos, I. (septiembre de 2010). Software engineering research for computer games: A systematic review. *Information and Software Technology*, págs. 888-901.

ASUS. (20 de abril de 2017). *Portátil Gaming | Asus ROG Strix GL502VM-FY213T, 15.6", FHD, i7-7700HQ, 16GB, 1TB + 128GB, GTX1060*. Obtenido de MediaMarkt: https://www.mediamarkt.es/es/product/_portátil-gaming-asus-rog-strix-gl502vm-fy213t-15-6-fhd-i7-7700hq-16gb-1tb-128gb-gtx1060-1383337.html

Autodesk, Inc. (2006-2021). *FBX | Adaptable File Formats for 3D Animation Software | Autodesk*. Obtenido de Autodesk: FBX: https://www.autodesk.com/products/fbx/overview

Blasco, D., Font, J., Zamorano, M., & Cetina, C. (2021). An evolutionary approach for generating software models: The case of Kromaia in Game Software Engineering. *Journal of Systems and Software, Volume 171*.

Bolton, D. (23 de junio de 2019). *Definition of Stack in Programming*. Obtenido de ThoughtCo.: Definition of Stack in Programming: https://www.thoughtco.com/definition-of-stack-in-programming-958162#:~:text=A%20stack%20is%20an%20array,first%20out"%20or%20LIFO%20order.

Camacho, M. (2021). *El coste de un trabajador para la empresa [+ fórmula + vídeo] - Factorial*. Obtenido de Factorial Blog: https://factorialhr.es/blog/coste-empresa-trabajador/

Campana, N. (23 de mayo de 2018). *¿Qué distingue a un programador junior de un programador senior?* Obtenido de freelancermap: https://www.freelancermap.com/blog/es/diferencias-programador-junior-y-senior/

Carmona Suárez, E. J., & Fernández Galán, S. (2019). *Fundamentos de la computación evolutiva.* Marcombo.

Centro de negocios en Zaragoza. (2021). *Alquiler aulas formacion zaragoza*. Obtenido de BSSC alquiler aulas de formación: http://bssc.es/espacios-bssc/alquiler-aulas-de-formacion

Cisa, N. (2021). *¿Cuántos kw consume una casa? Consumo medio | Podo*. Obtenido de Podo: Factura media de luz en un hogar de España ¿Cuánto se gasta al mes?: https://www.mipodo.com/blog/informacion/factura-media-luz-hogar-espana/

Clements, P., & Northrop, L. (2001). *Software Product Lines.* Addison-Wesley Professional.

Epic Games. (1998-2021). *Unreal Engine*. Obtenido de The most powerful real-time 3D creation platform - Unreal Engine: https://www.unrealengine.com/

Fernández, J. L., & Coronado, G. (Consultado en 2021). *Ley de Hooke*. Obtenido de FisicaLab: Ley de Hooke: https://www.fisicalab.com/apartado/ley-hooke

Ferro, M. (23 de abril de 2021). *▷¿Cuál es el precio del agua en España y quién lo regula?* Obtenido de Tarifasdeagua by Selectra: Precio de agua en España: Toda la información: https://tarifasdeagua.es/info/precio

Goldsby, H. J., & Cheng, B. H. (2008). Automatically Generating Behavioral Models of Adaptive Systems to Address Uncertainty. En K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, & M. Völter, *Model Driven Engineering Languages and Systems* (págs. 568-583). Toulouse, France: Springer Berlin Heidelberg.

Gupta, L. (22 de octubre de 2012). *Java Singleton Pattern Explained - HowToDoInJava*. Obtenido de HowToDoInJava: Java Singleton Pattern Explained: https://howtodoinjava.com/design-patterns/creational/singleton-design-pattern-in-java/

Gustavsson, B., Gudmundsson, D., & others. (2006-2021). *Wings 3D*. Obtenido de Wings 3D: A Polygon Modeler: http://www.wings3d.com

Indeed. (2004-2021). *Ofertas de trabajo, bolsa de trabajo | Buscar empleo en Indeed España*. Obtenido de indeed: https://es.indeed.com/

Janssens, D. (23 de julio de 2015). *What is a bitmask and a mask? - Stack Overflow*. Obtenido de Stack Overflow: What is a bitmask and a mask?: https://stackoverflow.com/questions/31575691/what-is-a-bitmask-and-a-mask

JetBrains s.r.o. (2000-2021). *Comprar Rider: Precios y licencias, Descuentos - Suscripción a JetBrains Toolbox*. Obtenido de JetBrains Rider: https://www.jetbrains.com/es-es/rider/buy/#personal?billing=monthly

Kraken Empire. (23 de octubre de 2014). Kromaia. *Kromaia*. Zaragoza, Zaragoza, Spain: Kraken Empire.

Mernik, M., Heering, J., & M. Sloane, A. (2005). When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, págs. 316-344.

Microsoft Corporation. (2011-2021). *Comparar todos los planes de Microsoft 365 (anteriormente Office 365): Microsoft Store*. Obtenido de Microsoft Office 365: https://www.microsoft.com/es-es/microsoft-365/buy/compare-all-microsoft-365-products?rtc=1

Microsoft Corporation. (20 de julio de 2015). *abstract: Referencia de C# | Microsoft Docs*. Obtenido de Microsoft | Docs: abstract (Referencia de C#): https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/abstract

Microsoft Corporation. (2015-2021). *Buy & Download Windows 10 - Microsoft Store*. Obtenido de Microsoft Windows 10: https://www.microsoft.com/en-us/store/b/windows?activetab=tab%3ashopwindows10

Microsoft Corporation. (9 de abril de 2019). *Instrucción switch de C# | Microsoft Docs*. Obtenido de Microsoft | Docs: switch (referencia de C#): https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/switch

Microsoft Corporation. (Consultado en 2021). *List<T> Clase (System.Collections.Generic) | Microsoft Docs*. Obtenido de Microsoft | Docs: List<T> Clase: https://docs.microsoft.com/es-es/dotnet/api/system.collections.generic.list-1?view=net-5.0

Montero Reyno, E., & Á Carsí Cubel, J. (2009). Automatic prototyping in model-driven game development. *SPECIAL ISSUE: Media Arts and Games (Part II)*, págs. Vol. 7, No. 2.

Mozilla and individual contributors. (2005-2021). *Métodos de petición HTTP - HTTP | MDN*. Obtenido de MDN Web Docs: Métodos de petición HTTP: https://developer.mozilla.org/es/docs/Web/HTTP/Methods

Mozilla and individual contributors. (2005-2021). *WebGL - Referencia de la API Web | MDN*. Obtenido de MDN Web Docs: WebGL: https://developer.mozilla.org/es/docs/Web/API/WebGL_API

Muñoz Riascos, E. J., & Rincón, L. (2014). *Modelo de características de una línea de productos de software para aplicaciones que usan dispositivos de Reconocimiento Biométrico.* Cali, Colombia.

Núñez Valdez, E. R., García Díaz, V., Cueva Lovelle, J. M., Sáez, Y., & González Crespo, R. (Junio de 2017). A model-driven approach to generate and deploy videogames on multiple platforms. *Journal of Ambient Intelligence and Humanized Computing 8(3)*, págs. 435-447.

Núñez Valdéz, E. R., Sanjuán Martínez, Ó., Pelayo García-Bustelo, B. C., Cueva Lovelle, J. M., & Infante Hernández, G. (2013). Gade4all: developing multi-platform videogames based on

domain specific languages and model driven engineering. *The International Journal of Interactive Multimedia and Artificial Intelligence (IJIMAI), 2(2)*, págs. 33-42.

O2 España. (2018-2021). *O2 | 300Mb de fibra y 10GB para tu móvil con la mejor cobertura.* Obtenido de O2: Tarifa Fibra 300Mb y Móvil 10GB: https://o2online.es/fibra-y-movil-300/

Red Hat. (1993-2021). *¿Qué es el open source?* Obtenido de Red Hat: ¿Qué es el open source?: https://www.redhat.com/es/topics/open-source/what-is-open-source

RI5. (Consultado en 2021). *RI5.com.ar Informatica - Que es XML y para que sirve*. Obtenido de RI5: https://www.ri5.com.ar/wiki/Que_es_XML.php

Roosendaal, T., & others. (1998-2021). *blender.org - Home of the Blender project - Free and Open 3D Creation Software*. Obtenido de blender: https://www.blender.org

Sagástegui Lescano, W. (Consultado en 2021). *¿Qué es y para qué sirve el lenguaje de etiquetas XML (Extensible Markup Language)?* Obtenido de aprenderaprogramar: ¿Qué es y para qué sirve el lenguaje de etiquetas XML (Extensible Markup Language)?: https://www.aprenderaprogramar.es/index.php?option=com_content&view=article&id=102:ique-es-y-para-que-sirve-el-lenguaje-de-etiquetas-xml-extensible-markup-language&catid=46&Itemid=163

Solís-Martínez, J., Pascual Espada, J., García-Menéndez, N., Pelayo G-Bustelo, B. C., & Cueva Lovelle, J. M. (2015). VGPM: Using business process modeling for videogame modeling and code generation in multiple platforms. *Computer Standards & Interfaces*, págs. 42-52.

Sony Interactive Entertainment. (15 de noviembre de 2013). *PS4*. Obtenido de PS4 | Juegos increíbles, entretenimiento sin fin | PlayStation: https://www.playstation.com/es-es/ps4/

SVIT Research Group. (2011-2021). *SVIT Research Group /*. Obtenido de SVIT Research Group: https://svit.usj.es

SVIT Research Group. (21 de marzo de 2020). Model Driven Engineering and Video Game Content. Villanueva de Gállego, Zaragoza, España. Obtenido de https://youtu.be/Vp3Zt4qXkoY

TutorialsTeacher. (10 de mayo de 2020). *C# Arrays (With Easy Examples)*. Obtenido de TutorialsTeacher C# Arrays: https://www.tutorialsteacher.com/csharp/array-csharp#:~:text=An%20array%20is%20the%20data,%2C%20multidimensional%2C%20and%20jagged%20array.

Unity Technologies. (2005-2021). *Unity*. Obtenido de Unity Real-Time Development Platform | 3D, 2D VR & AR Engine: https://unity.com

Unity Technologies. (2016). *Unity - Manual: Corrutinas*. Obtenido de Unity | Documentation: Corrutinas: https://docs.unity3d.com/es/530/Manual/Coroutines.html

Unity Technologies. (31 de julio de 2018). *Unity - Manual: Prefabs*. Obtenido de Unity | Documentation: Prefabs: https://docs.unity3d.com/Manual/Prefabs.html

Universidad San Jorge. (2005-2021). *UNIVERSIDAD SAN JORGE*. Obtenido de Universidad San Jorge - Zaragoza: https://www.usj.es

Usman, M., Zohaib Iqbal, M., & Uzair Khan, M. (2017). A product-line model-driven engineering approach for generating feature-based mobile applications. *Journal of Systems and Software, Volume 123*, 1-32.

Wagner, B., & "olprod". (17 de marzo de 2021). *Directivas de preprocesador de C# | Microsoft Docs*. Obtenido de Microsoft | Docs: Directivas de preprocesador de C#: https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/preprocessor-directives

Wavefront Technologies. (Consultado en 2021). *obj-spec.pdf*. Obtenido de B1. Object Files (.obj): http://fegemo.github.io/cefet-cg/attachments/obj-spec.pdf

Williams, J. R., Poulding, S., Rose, L. M., Paige, R. F., & Polack, F. A. (2011). Identifying Desirable Game Character Behaviours through the Application of Evolutionary Algorithms to Model-Driven Engineering Metamodels. En M. B. Cohen, & M. Ó Cinnéide, *Search Based*

Software Engineering, Third International Symposium (págs. 112-126). Springer Berlin Heidelberg.

11. Tablas de figuras

11.1. Tabla de ilustraciones

11.2. Tabla de tablas

12. Anexos

12.1. Propuesta de proyecto final

Nombre alumno: Javier Verón Mérida
Titulación: Grado en Ingeniería Informática
Curso académico: 5º (Doble grado)

1. TÍTULO DEL PROYECTO

Generación de contenido procedural en videojuegos para casos reales.

2. DESCRIPCIÓN Y JUSTIFICACIÓN DEL TEMA A TRATAR

Investigación con la beca por colaboración en investigación en diseño y desarrollo de videojuegos, en la que potencialmente se colaborará con Tequila Works, Genera Games y Kraken Empire.

3. OBJETIVOS DEL PROYECTO

Concluir resultados sobre la investigación y generar recursos para la divulgación de estos.

4. METODOLOGÍA

Se definirá cuando empiece el proyecto.

5. PLANIFICACIÓN DE TAREAS

Se definirá cuando empiece el proyecto.

6. OBSERVACIONES ADICIONALES

El proyecto será dirigido por Carlos Cetina.

12.2. Reuniones

1. **Elabora acta:** Javier Verón
   Convocados: Carlos Cetina
   Fecha: 19 de noviembre de 2020.
   Modo: online.
   Duración: 2h.
   Programa: Microsoft Teams.
   Contenido: Se han sentado las bases de lo que será mi trabajo y cuáles son los objetivos finales. Cuando tenga un proyecto funcional se realizará la próxima reunión.

2. **Elabora acta:** Javier Verón
   Convocados: Carlos Cetina
   Fecha: 2 de abril de 2021.
   Modo: online.
   Duración: 2h.
   Programa: Microsoft Teams.
   Contenido: Evaluación de problemas encontrados hasta la fecha con las físicas. Se ha decidido pulir el proyecto y empezar ya a integrar la versión para WebGL, abandonando por el momento este apartado ya que no es prioritario.

3. **Elabora acta:** Javier Verón
   Convocados: Carlos Cetina
   Fecha: 17 de mayo de 2021.
   Modo: online.
   Duración: 2h.
   Programa: Microsoft Teams.
   Contenido: El proyecto está funcionando bien. A partir de ahora será pulir los diferentes apartados y mejorar opciones del menú. A partir de aquí, en principio la comunicación será vía email.

4. **Elabora acta:** Javier Verón
   Convocados: Daniel Blasco
   Fecha: 08 de enero de 2021.
   Modo: online.
   Duración: 3h.
   Programa: Discord.

Contenido: Se ha hablado de cómo representar las diferentes partes tras leerlas del SDML. Daniel me ha enviado una imagen con las clases para basarme en ella.

5. **Elabora acta:** Javier Verón
   Convocados: Daniel Blasco
   Fecha: 16 de abril de 2021.
   Modo: online.
   Duración: 2h.
   Programa: Discord.
   Contenido: Se ha hablado sobre el problema de físicas con los Joints de Unity, llegando a la conclusión de que llevaría demasiado tiempo aunque haya merecido la pena probar ciertas acciones que podrían haber funcionado.

12.3. Imágenes del experimento

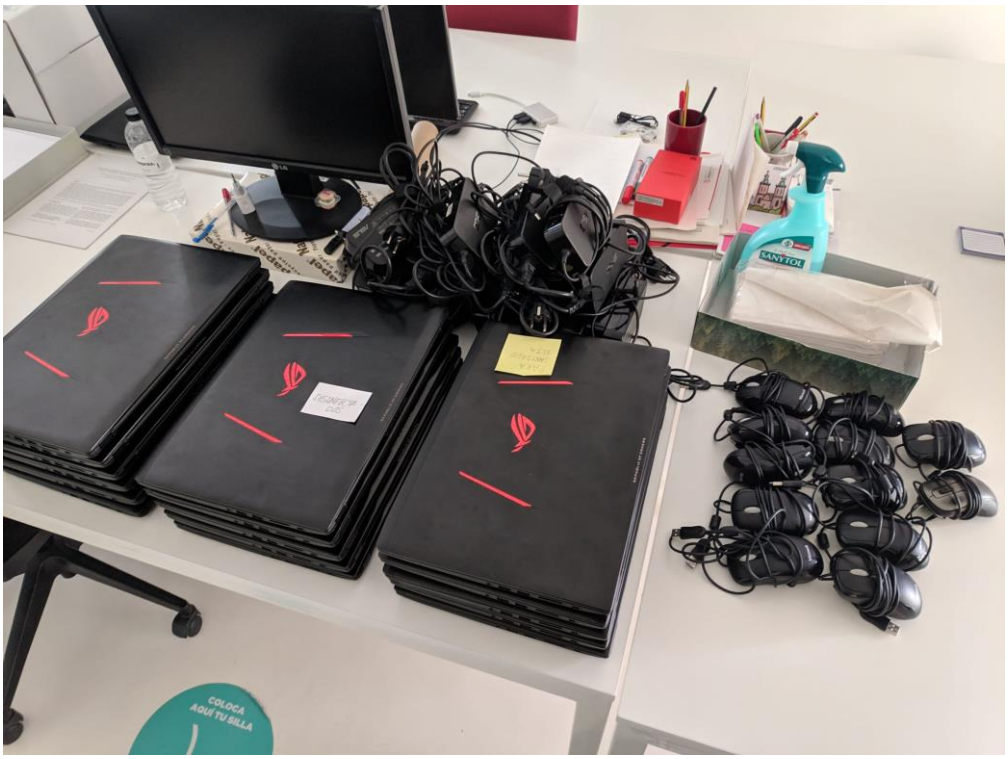

Ilustración 39: Equipos informáticos prestados por la Universidad San Jorge para el desarrollo del experimento. Fuente: Propia.

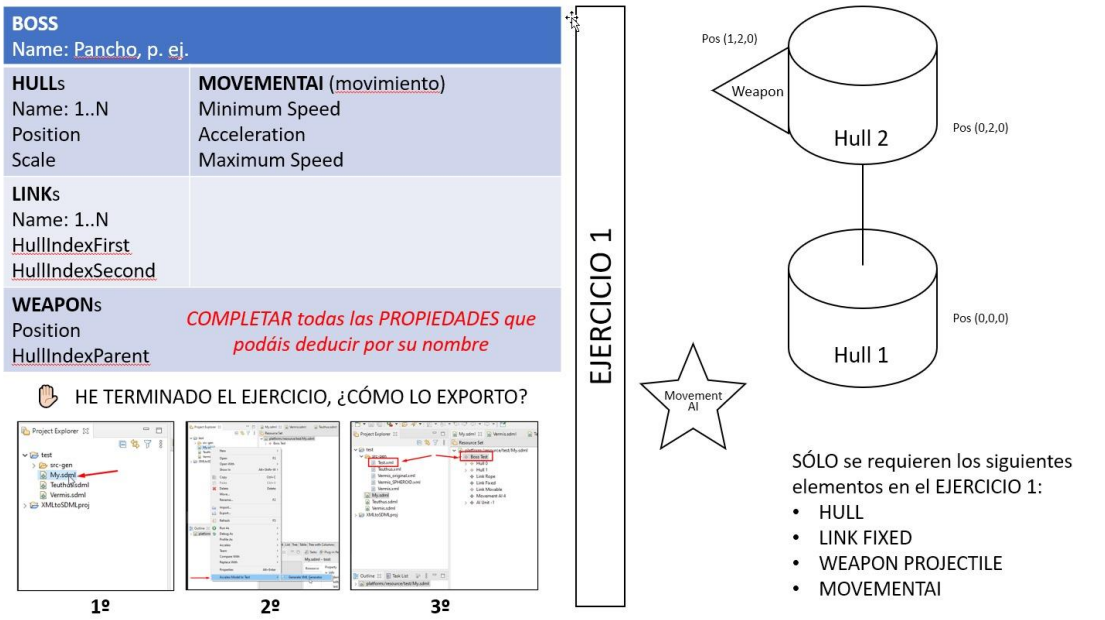

Ilustración 40: Primer ejercicio del experimento. Fuente: Jorge Chueca.

Ilustración 41: Fotografía durante la primera parte del experimento del día 19/05/2021. Fuente: Rodrigo Casamayor.

Ilustración 42: Fotografía durante la segunda parte del experimento del día 24/05/2021. Fuente: Propia.