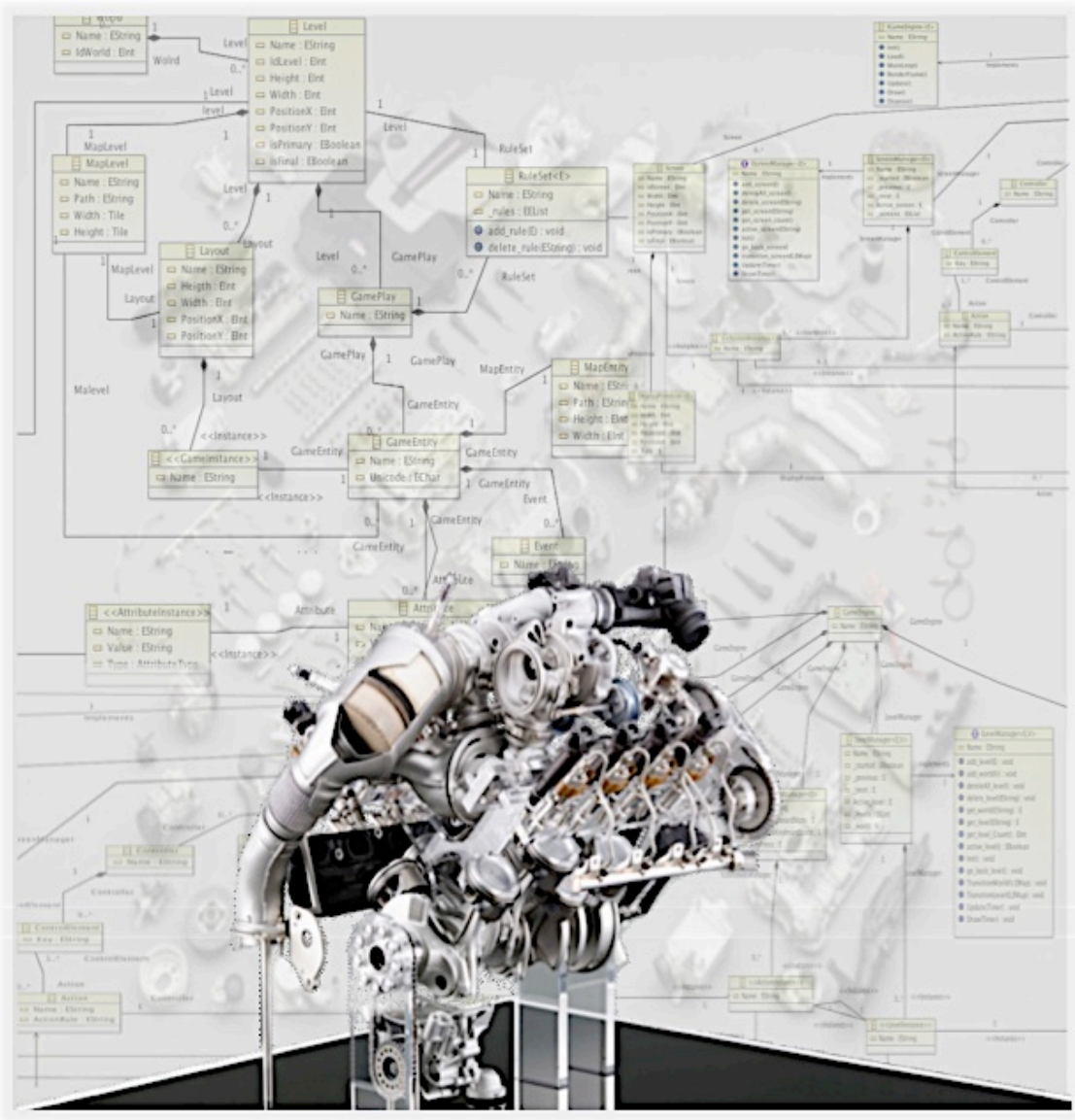
Diseño de Niveles y uso de Motores en el Desarrollo de Videojuegos dirigido por Modelos



Alumno: Víctor M. Bolinches Marín
Director: José A. Carsí Cubel
Grupo de Ingeniería del Software y Sistemas de Información
Departamento de Sistemas Informáticos y Computación
vicboma@ei.upv.es

26 de septiembre de 2012
Máster en Ingeniería del Software, Métodos Formales y Sistemas de Información
Universidad Politécnica de Valencia

Índice

Tabla de Ilustraciones

Resumen

La propuesta del desarrollo de juegos dirigidos por modelos (MDGD) ofrece un multi-modelo para la especificación de videojuegos dividido en varias vistas: jugabilidad, interfaz gráfica y control entre otras. Este concepto de modelado conceptual permite a los diseñadores de juegos especificar juegos a un gran nivel de abstracción independientemente de la plataforma utilizada mediante la aplicación de MDD. En este trabajo se presenta una nueva vista añadida al modelado del gameplay de los videojuegos que permite el diseño de niveles. Ya que es común en el desarrollo de juegos el utilizar motores 'estándares' en su construcción, se muestra cómo es posible integrar un motor de tiles 2D en el proceso de desarrollo definiendo un meta-modelo específico de la plataforma (PSM) genérico que permite definir la estructura y el comportamiento del sistema haciendo uso del motor sin entrar en los detalles técnicos de implementación. De este modo, se pretende agilizar el proceso de software de desarrollo de videojuegos mediante el uso de transformaciones automáticas que especifican el diseño de alto nivel del juego hasta obtener la implementación específica de la plataforma tecnológica. A modo de prueba conceptual, se ha resuelto el desarrollo de videojuegos utilizando un motor de tiles 2D mediante un prototipo de herramienta de desarrollo de juegos dirigido por modelos en el entorno de desarrollo Eclipse Modeling Framework, utilizando transformaciones QVT entre modelos y generación automática de código C# para el middleware específico Microsoft XNA.

***Keywords**: desarrollo de software dirigido por modelos, diseño de niveles, motor de videojuegos 2D, especificación gameplay, diseño de juegos, tiles, PIM, PSM, MDA, MDD.*

1 Introducción

Hace décadas que la industria del videojuego desarrolla software para todos los públicos en una amplia variedad de plataformas tecnológicas. Lejos quedan las sofisticadas recreativas y videoconsolas de los 80 que ejecutaban multitud de juegos en sus *hardwares*. A sus espaldas tenían grandes equipos de desarrollo que más tarde en los 90 catalogarían a estos videojuegos como juegos AAA por ser sus gráficos, jugabilidad y sonido de excelencia. A día de hoy, nuevas tecnologías tales como los *Tablets* y los dispositivos móviles de nueva generación son un foco creciente en el desarrollo de videojuegos. Éstas hacen posible la nueva adaptación de viejos juegos a un entorno con nuevas prestaciones, muchas de ellas táctiles, como muestra la figura 1.1. Permiten que un grupo reducido de personas expertas en el dominio de una plataforma pueda realizar juegos en poco tiempo olvidando las largas etapas que conlleva el desarrollo de grandes juegos y de los motores que acompañan a éstos en tiempo de ejecución. Uno de los problemas en el desarrollo de videojuegos viene dado porque se carece de un lenguaje de especificación para los videojuegos con el que los diseñadores que no saben programar, puedan escribir la documentación del diseño [26]. Los programadores traducen como pueden estas especificaciones a código, que más tarde será compilado, ejecutado y apto para la detección de errores.

Figura 1.1. Ejemplo de un juego 2D adaptado a las nuevas tecnologías.

Con la intención de solventar estos problemas, se propuso en [2] elevar el nivel de abstracción del desarrollo de videojuegos mediante un modelo independiente de la plataforma (PIM) para el diseño de videojuegos. *Model Driven Architecture* (MDA) permite elevar el nivel de abstracción tecnológico puesto que los modelos pueden usarse para favorecer el uso de lenguajes específicos de domino que los diseñadores utilizarán para modelar los artefactos en sus propios conceptos de dominio [6]. Puesto que MDA favorece el proceso de desarrollo de software multi-paradigma y es una arquitectura que proporciona un conjunto de guías para estructurar especificaciones expresadas con modelos, se utiliza un modelo PIM de alto nivel de abstracción a través de lenguajes específicos de dominio (DSLs) para el uso del dominio de videojuegos. Las especificaciones de este modelo PIM son usadas para tener en cuenta detalles de implementación precisando los requerimientos funcionales que lo hace sobrevivir a

los cambios que se produzcan en las tecnologías dependientes y arquitecturas software. En este artículo se presenta una nueva especificación añadida al modelo de interactividad de videojuegos como es el diseño de niveles que junto con las tres perspectivas fundamentales: jugabilidad, control e interfaz gráfica de usuario, definen el *gameplay* del juego.

El término "motor de videojuegos" salió a la luz a principios de los 90 cuando grupos de desarrolladores perfeccionaban sus juegos antes de sacarlos al mercado. Estos se dieron cuenta que separar las partes principales del software les permitía especializarse y crecer en los conceptos básicos de un videojuego. El desarrollo de éste ya no se haría desde una etapa principal partiendo desde cero sino que se reutilizaría este desarrollo para lanzar nuevas secuelas de juegos más rápidos y más fáciles de desarrollar. Gracias a los motores pudiendo ser más competentes en la industria del videojuego como muestra la figura 1.2 donde se presenta la producción iterativa del desarrollo de una secuela del videojuego *Super Mario Bross*.

Se presenta también un motor de *tiles* 2D mediante un meta-modelo PSM para conceptualizar la plataforma destino que especifica la estructura y el comportamiento del sistema (juego) sin entrar en detalles técnicos de implementación para el manejo de objetos en 2 dimensiones inmersos en un ambiente y que interactúan entre sí [7].

Trataremos únicamente el desarrollo de videojuegos 2D puesto que el mercado está en auge, sobre todo en los dispositivos móviles y los *Tablets* que a día de hoy son un punto clave para el desarrollo de videojuegos 2D, donde pequeños equipos de desarrollo proponen juegos casuales o de tipo *indies* en breves periodos de tiempo con sus secuelas o sagas correspondientes.

A modo de prueba conceptual, se ha implementado un prototipo de herramienta de desarrollo de juegos dirigido por modelos en el entorno de desarrollo *Eclipse Modeling Framework*, utilizando transformaciones QVT entre modelos y generación automática de código C# para el *middleware* específico *Microsoft XNA*. Mediante la utilización de esta herramienta se va a utilizar la metodología propuesta de desarrollo de juegos dirigido por modelos para desarrollar un videojuego de lucha en 2D a un elevado nivel de abstracción conceptual. Al mismo tiempo, se ha aumentado la productividad mediante el *middleware* multi-plataforma *Microsoft XNA*, que permite desarrollar de forma sencilla y rápida videojuegos para PC y XBOX 360, dos de las plataformas tecnológicas para el ocio interactivo más extendidas en la actualidad.

Figura 1.2. Game Master, motor de *tiles* para el desarrollo de juegos 2D de *Super Mario Bross*.

2 Estado del Arte

Habitualmente, el desarrollo de videojuegos se ha caracterizado por un bajo nivel de abstracción tecnológica, donde priman las soluciones *adhoc* en pos de un mayor aprovechamiento de los recursos hardware disponibles. En la actualidad, el uso de metodologías de desarrollo de software escasean en el desarrollo de juegos. Se carece de un lenguaje de especificación para juegos donde el diseñador no puede expresar las reglas del juego en un lenguaje natural y se encuentra con un vacío semántico entre el diseño y la implementación del juego. A continuación, se discute sobre algunos trabajos existentes relacionados con nuestro propósito:

Furtador et al [1] abordan el desarrollo de videojuegos mediante un DSL y líneas de productos software, todo el código que generan se apoya en un motor llamado *ArcadEngine* que extiende e implementa dicho código proveniente de los modelos de *ArcadEx* que quita complejidad al código generado. Después un segundo motor llamado *FlatRedBall* es el que consume este código final y lo presenta en el framework destino. Los autores argumentan que no generan el juego en su totalidad sino que pretenden usar un DSL y líneas de productos software para apoyarlas en el motor y obtener una reutilización de software más estructurada, eficaz e intuitiva. El uso de este DSL permite a los diseñadores de juegos elevar el nivel de abstracción y trabajar con conceptos más próximos a su dominio de aplicación, el problema viene dado en que la utilización de un DSL y un motor a bajo nivel les limita la productividad de géneros de videojuegos en una plataforma destino, evitando así el posible uso industrializado donde se desarrollan videojuegos para muchas plataformas.

Hernandez F. et al [25] hacen uso de técnicas de MDD para ayudar a reducir el coste de complejidad del proceso de desarrollo de videojuegos mediante un solo DSL centrado en el modelado de juegos 2D. Esto les permite elevar el nivel de abstracción del desarrollo de juegos mediante modelos y reducir costos de tiempo y esfuerzo en dicho desarrollo. A pesar de que muestran garantías para la generación de videojuegos, ellos sólo se apoyan en un DSL y discuten que el motor que sostiene el juego sólo consume el código generado en vez de apoyarlo mediante clases que manejen esa información para hacerlo más efectivo y estructurado.

Dobbe [8] ofrece el desarrollo de un DSL independiente del género para diseñar juegos, esto permite trasladar el ámbito del juego a un nivel de abstracción mayor donde contempla varios aspectos para la implantación de un diseño de juego, estos aspectos pueden ser considerados como requisitos principales para el funcionamiento y definición del dominio del DSL utilizando los siguientes aspectos: objetos que comparte el mundo del juego, las interacciones con el jugador, las reglas que rigen la mecánica del juego y la historia que nos presenta. Esto puede ser de gran interés para los diseñadores de juegos pero no aborda más etapas del desarrollo ni dice como se transformarán las especificaciones en la implementación.

Altunbay et al [24] ofrecen una aproximación al desarrollo MDA de juegos de tablero compuesto por jugadores y un motor de juego. Los jugadores tienen objetivos que cumplir y controlan elementos de juego. El motor del juego esta constituido de elementos, estados, y un nivel que añade eventos, reglas y acciones. Esta aproximación con MDA es clara y muy útil para diseñar juegos de tablero pero es difícil adaptarla al diseño de videojuegos.

Miroslav et al [10] abordan el desarrollo de videojuegos universitarios educativos trasladados a un ámbito web mediante el uso de un enfoque MDA. La utilización de un modelo PIM les permite abstraer la semántica del problema e identificar las partes más importantes del juego como son el conocimiento, la interacción, el nivel de dificultad y el tipo

de usuario. Mediante una transformación de modelos obtienen hasta 4 tipos de PSM diferentes que finalmente juntan en un lenguaje *HTML y* definen el término EGLO (Educational Game Learning Object) para su desarrollo de videojuegos. Este enfoque tiene el gran inconveniente de que los juegos carecen de la capacidad de presentar diversos conocimientos porque el propio proceso de construcción requiere parte del conocimiento de los creadores.

James et al [14] describre un nuevo enfoque usando MDE con modelos metaheurísticos y se ilustra un problema concreto de ingeniería derivando los personajes de juego de luchas. El comportamiento de los personajes se define mediante un DSL que se interpreta usando técnicas de MDE. El principal problema de este enfoque es que no se define la creación del juego en su totalidad. Se hace incapié en un determinado aspecto y sería un gran problema la posible automatización de este enfoque para generar juegos de lucha y ser competentes en los mercados de videojuegos.

Folmer [27] aplica el desarrollo de juegos basado en la reutilización de componentes, hace referencia a una arquitectura basado en capas que predomina en todos los juegos que desarrollan. El principal problema a este desarrollo es que la reutilización de estos componentes es limitada, gráficos y sonido son los principales perjudicados. Pretenden desarrollar juegos desde la parte más baja de su implementación por lo que no consideran el problema y las tareas relacionadas como dominio de análisis.

2.1 Aplicación de MDA en el desarrollo de videojuegos

La utilización de MDA aplicada al desarrollo de videojuegos permite capturar los conceptos de la jugabilidad de un juego en un modelo PIM. La identificación de estos aspectos permite a los diseñadores expresar cualquier concepto del dominio independiente de la plataforma abstrayéndose de un lenguaje de programación. Los detalles tecnológicos de implementación se ignoran, considerándose únicamente el nivel más bajo de abstracción tecnológico, mediante un modelo específico de la plataforma (PSM) [28]. Un modelo definido a un nivel de abstracción puede transformarse a una o varias plataformas tecnológicas recibiendo el nombre de transformaciones verticales entre modelos PIM-PSM. A su vez, estos modelos específicos podrán transformarse mediante transformaciones PSM a texto para poder obtener finalmente código plano ejecutable.

La Figura 2.1.1 muestra cómo el conocimiento del sistema se define a un elevado nivel de abstracción tecnológica (modelo PIM). Mediante sucesivas transformaciones automáticas entre modelos se deriva la del modelo independiente de la plataforma a las diferentes tecnologías específicas (modelos PSM 1 y 2). Es en el último paso, cuando se compila cada modelo específico de la plataforma obteniéndose el código directamente ejecutable (código 1 y 2).

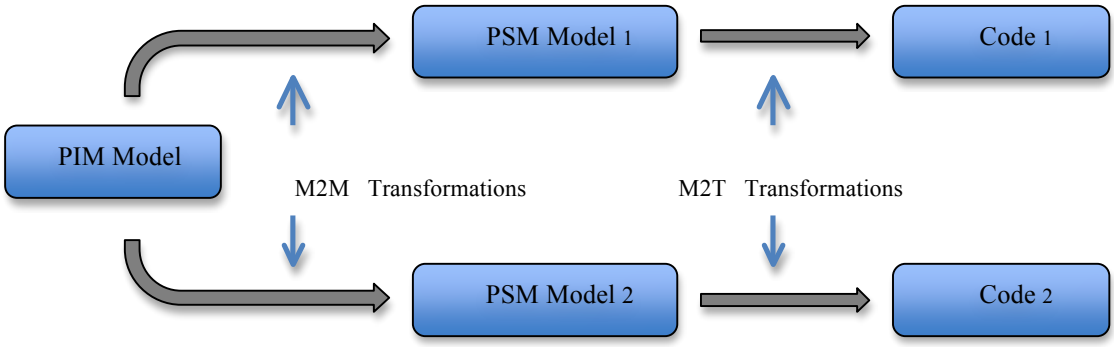


Figura 2.1.1. Desarrollo de software dirigido por modelos.

Los diseñadores de juegos especifican los elementos del juego semánticamente mediante un modelo PIM a diferencia de la manera tradicional. Este artefacto software permite a los diseñadores evitar la difícil tarea de traducir los documentos de texto en lenguaje natural. Las transformaciones automáticas entre modelos se hacen mediante QVT, es un lenguaje standard de transformación que utiliza ciudadanos de primer orden para realizar el *mappings* entre modelos. El PSM es generado a partir del PIM y contempla muchos más detalles específicos de la plataforma. Mediante un compilador de modelos PSM se transforma el modelo PSM a código plano ejecutable. El código fuente es compilado y los binarios generados son utilizados por los *testers* para la realización de pruebas. Los errores detectados en estas fases pueden ser refinados a todos los niveles de abstracción. Los cambios en el diseño del juego son realizados en el modelo PIM, los cambios de implementación sobre el modelo PSM y los cambios a bajo nivel se hacen en el código fuente. Aplicar MDA al desarrollo de videojuegos puede cambiar el ciclo de vida tradicional del juego, haciendo que los diseñadores dirijan todo el proceso de desarrolladores (veasé Figura 2.1.2)

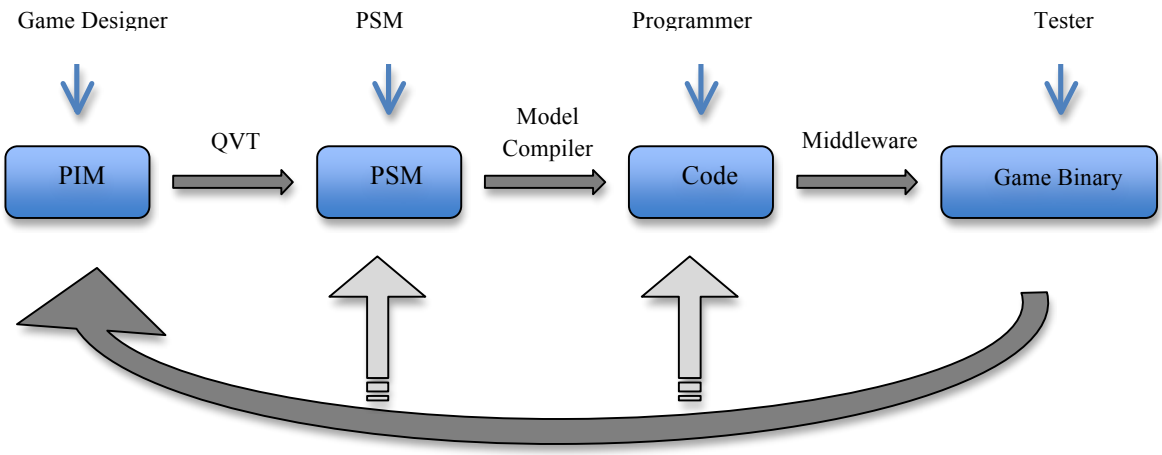


Figura 2.1.2. Proceso de desarrollo de software dirigido por modelos.

Partiendo del ciclo de la jugabilidad de Crawford [5], los videojuegos pueden considerarse sistemas interactivos en el que los jugadores se comunican con el juego a través de acciones de entrada realizadas mediante controladores hardware. Este ciclo se repite continuamente durante la interacción entre los jugadores y el juego.

En la sección 2.1.1 presentamos la perspectiva de la interfaz gráfica de usuario a través de un diagrama de navegación, en la sección 2.1.2 se presenta la perspectiva de control. Por último en la sección 2.1.3 se detalla la perspectiva de las reglas e identidades [11] que forman parte de las vistas que hemos definido en nuestro framework.

2.1.1 Perspectiva de la Interfaz Gráfica de Usuario

Para facilitar ciertos aspectos del modelado visual a los diseñadores de juegos, la perspectiva GUI muestra información a los jugadores de cómo está estructurada la navegación de las pantallas del videojuego y cómo se organiza la información en éstas. La figura 2.1.1.1 muestra el meta-modelo de navegación que representa las principales primitivas de desplazamiento entre pantallas. El principal objetivo de esta perspectiva es que el diseñador pueda hacer transiciones entre pantallas representados como nodos a través de transiciones de una pantalla a otra declarando sus respectivos eventos de entrada o salida, tales como eventos del juego, interacciones de control o tiempo.

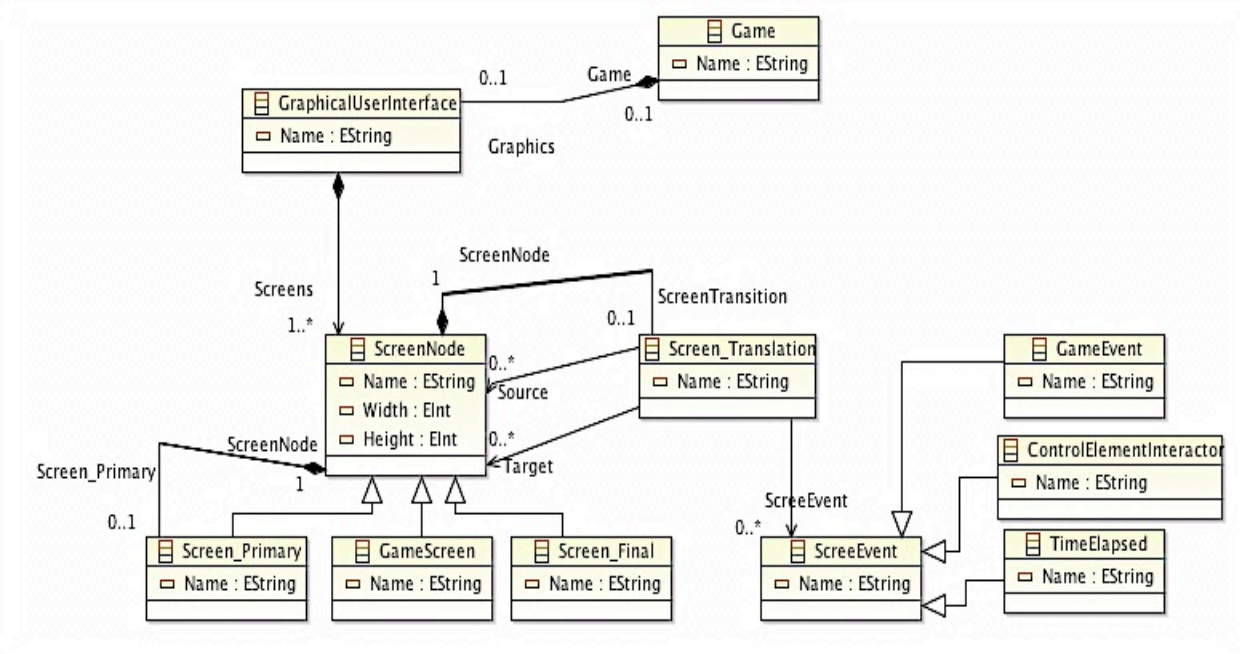


Figura 2.1.1.1. Meta-modelo de navegación entre pantallas.

Tomando un ejemplo, la figura 2.1.1.2 muestra un diagrama de navegación para un videojuego de plataformas en 2D. El diagrama especifica que el juego comienza por una pantalla de presentación de personajes, desde la que se accede al juego, seguidamente se muestra una pantalla donde se visualiza a modo de mapa ciertos niveles del juego. Se observa como cada mundo del mapa tiene unos niveles característicos a la imagen, en nuestro caso observamos como los personajes se posicionan en un mundo lo mas parecido a la transición de una selva. La siguiente imagen muestra el nivel que lo precede donde se observa a los personajes atravesando dicha selva. Finalmente, se llega a una pantalla de derrota. La pantalla de derrota permite volver a la pantalla de presentación de personaje para seguir jugando o bien terminar el juego definitivamente. Para facilitar la especificación a los diseñadores de juegos, se utiliza como notación visual una flecha para representar las transiciones y un rectángulo para representar las pantallas. Un nodo circular representa la pantalla inicial y un nodo en aspa representa la pantalla final.

Figura 2.1.1.2. Ejemplo de un modelo navegacional para un videojuego 2D.

2.1.2 Perspectiva de Control

El control define cómo se comunican los jugadores con el juego a través de dispositivos hardware controladores. Cada plataforma tecnológica ofrece diversos controladores que permiten a los jugadores distintas interacciones. Todos los controladores ofrecen a los jugadores elementos de control que permiten enviar información al sistema de juego. Así, se pueden definir algunos conceptos de control independientes de la tecnología, comunes a todos los controladores y plataformas tecnológicas.

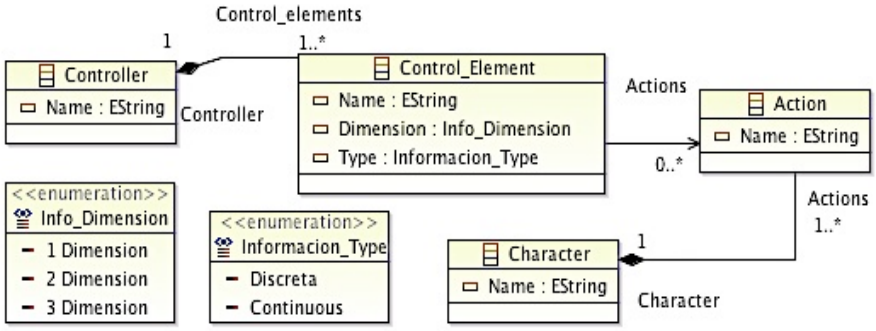


Figura 2.1.2.1. Meta-modelo de control.

La Figura 2.1.2.1 muestra el meta-modelo de control. Un controlador está formado por elementos de control (botones, joysticks, etc..). Los elementos de control envían información al sistema de juego (los botones envían información 0-dimensional, los gatillos 1-dimensional y los joysticks 2-dimensional). Los jugadores interactúan (pulsan, sueltan, mantienen, etc…) los elementos de control para ejecutar reglas de acción del conjunto de reglas .

A modo de ejemplo, los elementos del dispositivo de control de la plataforma tecnológica específica XBOX 360 (figura 2.1.2.2) permite catorce elementos de control mono-dimensionales discretos (un pad 4-direccional y diez botones diferentes), dos elementos de control mono-dimensionales continuos (los gatillos izquierdo y derecho) y dos elementos de control bidimensionales continuos (los thumbsticks izquierdo y derecho). Una complejidad de control que contrasta con la sencillez del teclado y sus cien elementos de control mono-dimensionales discretos (las teclas).

Figura 2.1.2.2. Pad específico de xbox.

2.1.3 Perspectiva de las Entidades y Reglas

En el dominio del desarrollo de juegos, las reglas y las entidades juegan un gran papel para la especificación de la jugabilidad. Especificar qué entidades del juego existen y qué reglas se disparan en el videojuego permite definir de forma precisa la interactividad del juego. A continuación, se detallan las principales primitivas de diseño para la especificación de la jugabilidad mediante reglas y entidades a través de la figura 2.1.3.1.

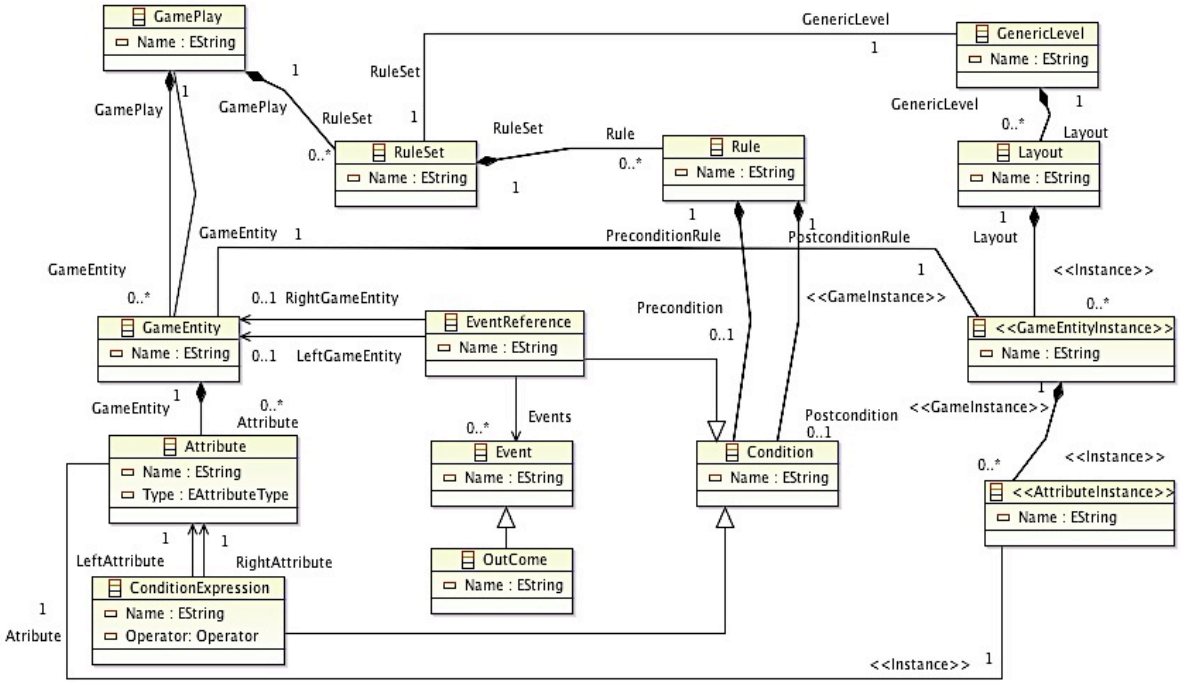


Figura 2.1.3.1. Meta-modelo de las Entidades y Reglas de un nivel.

Una entidad es un elemento que conforma el mundo y que se comporta según las normas del mismo. Un mundo esta compuesto por entidades con atributos y comportamiento, que además interactúan con el mismo según las normas de la lógica de juego. El comportamiento de una entidad viene dado por las normas definidas.

Una regla es una sentencia declarativa con una o más pre- y post-condiciones que establecen qué condiciones deben satisfacerse antes y después de la aplicación de la regla, respectivamente. Del mismo modo, todas las pre- y post-condiciones se expresan utilizando conceptos previamente definidos en el modelo de estructura para la jugabilidad: acciones, eventos y atributos.

Los atributos representan características propias de la entidad de juego. Los eventos representan sucesos disparados por la entidad de juego que cambian el estado del sistema de juego.

La meta-clase *Outcome* define un tipo especial de evento de resultado que establece la consecución de un objetivo de juego.

La meta-clase *GameEntity* define una entidad de juego pasiva, es decir, que no puede actuar en el sistema de juego.

3 Perspectiva del Diseño de Niveles

Cuando se trata del desarrollo videojuegos de alto presupuesto, el diseñador lleva a cabo la construcción de un nivel físicamente en papel [29]. Los equipos de especialistas se reúnen para llevar estos bocetos a la vida. Los programadores de motores gráficos desarrollan algoritmos para detectar cuándo los objetos chocan en el mundo. Los modeladores construyen complejas criaturas y los animadores son quiénes las dotan de movimiento. Los programadores de inteligencia artificial son quiénes dicen cómo se comportan estas criaturas. Los artistas son los encargados de "vestir" con texturas a las criaturas y el mundo en el que viven o interaccionan. Los técnicos de audio son los encargados de aportar los sonidos a los personajes y del hábitat en el que viven. Por último, es el diseñador de niveles es el que toma todas estas piezas desarrolladas por los equipos de especialistas y las une para crear los niveles del juego.

Uno de los objetivos fundamentales de la creación de un videojuego es la forma precisa de cómo se estructura el diseño de los niveles que compone nuestro juego y cómo integrar los aspectos más importantes del comportamiento de la jugabilidad, acción y guión [3]. Cada nivel engloba gran cantidad del contenido de un juego y es una parte que hace depender la evolución de las iteraciones en el desarrollo de un videojuego [20].

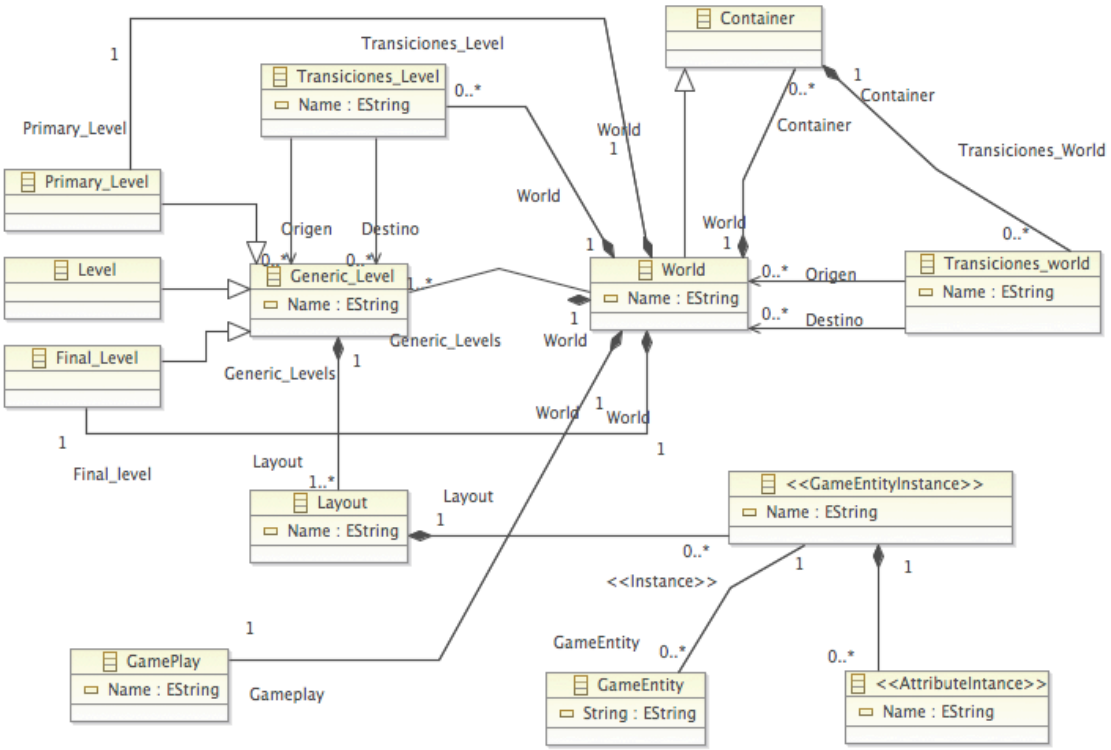


Figura 3.1 Meta-modelo del diseño de niveles.

El diagrama del diseño de niveles es un modelo visual que facilita a los diseñadores especificar la estructura global de los niveles que contiene el juego, abstrayéndolo del comportamiento del sistema.

Nótese que en la figura 3.1, la meta-clase World actúa como principal primitiva para la creación de los niveles. Se ha definido un mundo como un contenedor para utilizar una técnica usada por los guionistas y mapeadores de niveles que permite separar varias historias dentro de la narrativa de un juego, permitiendo así, especificar diferentes *gameplays* en los diferentes

escenarios del juego. Un claro ejemplo de uso es el utilizado en todas las sagas de Super Mario Bross dónde se definen mundos con transiciones entre éstos y se declaran niveles de juego dentro de estos mundos como muestra la figura 3.2.

Figura 3.2. Representación gráfica del diseño de niveles de New Super Mario Bross.

Otra característica que muestra este meta-modelo es la utilización de las capas que forman un nivel para hacer uso de técnicas de animación en el mundo de los videojuegos. Un *layout* es el conjunto de capas de presentación del juego. Esto permite al diseñador la declaración de tantos *layouts* como capas de representación se quiera. Todas estas "cajas" de diseño representan un conjunto que describen la composición final de la presentación del nivel [4].

La figura 3.3 muestra la técnica en uso permitiendo al diseñador diferenciar las capas de presentación de un nivel específico para separar en *layouts* los distintos activos que componen el nivel de un juego como son el background, HUD, *tiles*, etc…

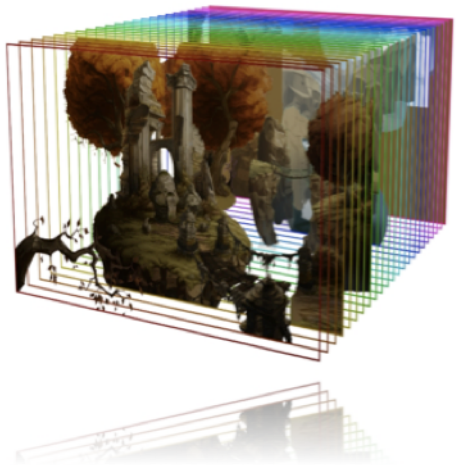


Figura 3.3. Representación gráfica de la composición de los distintos layouts que contiene un nivel del juego "the Whispered World".

Estas entidades se representan mediante la primitiva *GameEntityInstance* dentro de un *layout*. Son instancias que permiten la creación de un objeto perteneciente a una clase. Su representación en el meta-modelo está especificado mediante estereotipos tal y como se describe en [33].

Cada *GameEntityInstance* esta asociada a la meta-clase *GameEntity* que representa una entidad del juego independiente de la plataforma. A su vez, la primitiva *GameEntityInstance* está compuesta de *AttributeInstance*. Estos atributos son definidos tanto a nivel de la perspectiva de la Interfaz Gráfica de Usuario asociados con la meta-clase *DisplayPrimitive* y de la perspectiva de Entidades y Reglas con la meta-clase *Atributte* (veasé Apéndice I).

Para facilitar el modelado visual a los diseñadores de juegos, se ofrece un DSL para especificar un diagrama del diseño de niveles. La figura 3.4 muestra dos mundos que se relacionan entre sí mediante una transición descrita con una línea discontinua. Esta relación vendría descrita por la primitiva *Transiciones_World* del meta-modelo del diseño de niveles. Cada mundo contiene sus niveles correspondientes, éstos son representados mediante la primitiva *Generic_Level*. Se observa cómo en la descripción de los niveles tenemos una condición de obligación que describe el inicio como un punto negro y un final con dos puntos concéntricos, siendo blanco el circulo del interior [22] haciendo referencia a las meta-clases *Primary_Level* y *Final_Level* respectivamente.

Las transiciones entre los niveles se describen con flechas continuas siendo éstas tanto de origen como de destino. En el meta-modelo podemos localizar estas flechas uniformes con la meta-clase *Transiciones_Level*. Se observa cómo el nivel 1 avanza al nivel 2 mediante la transición T1, el nivel 2 apunta al nivel 3 mediante la transición T2, el nivel 3 describe dos posibles transiciones, retroceder al nivel 1 o avanzar al nivel 4. Este último nivel puede bien retroceder al nivel 2 mediante la transición 2 o finalizar el mundo.

La creación de un mundo define un *gameplay* como se observa en el meta-modelo, y con ello la especificación de la jugabilidad. Debido a la dificultad de la representación en la figura 3.1 se ha obviado este punto.

Cada nivel correspondiente a un mundo esta compuesto por un número de *layouts* que definen el comportamiento interno del nivel.

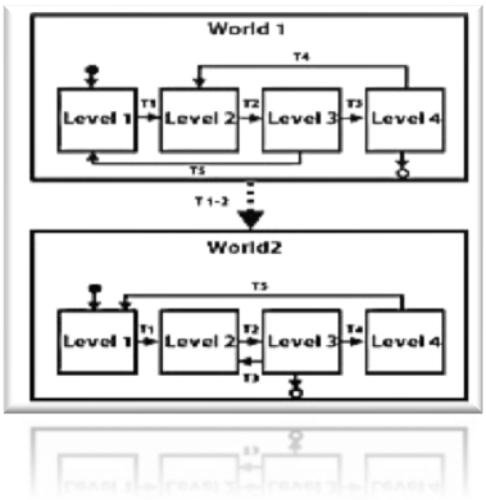


Figura 3.4. Diagrama del diseño de niveles genérico 2D.

4 Más allá del GamePlay de un juego, un Motor de *Tiles*

4.1 Motor de *Tiles* 2D.

Tratar de separar las principales partes que componen un juego permite profundizar y ampliar en el desarrollo de los conceptos más comunes. La elección de usar motor de *tiles* viene regida por el desarrollo iterativo de videojuegos en 2D. Todos los géneros de juegos RPG, plataformas, *scrollers*… toman como referencia el uso de *tiles*. Ésta no es más que una unidad que representa un gráfico en una área constituida por 2 dimensiones definida como una matriz que contiene las referencias de los objetos instanciados en un nivel del juego. Estas entidades del juego deben representar información del tipo de terreno, si es posible o no caminar sobre ella, si causa daño o beneficio al personaje del juego… Este enfoque simple permite a los diseñadores de juegos especificar de forma sencilla la representación de grandes mundos y niveles de juegos basados en *tiles* a través de mapas.

Estas *tiles* generalmente son simples representaciones geométricas como se observa en la figura 4.1.1 pudiendo ser extendidas a formas geométricas más elaboradas como hexágonos muy comunes en los juegos de puzles. Su definición viene dictada por una altura y anchura uniforme en todo el mapa.

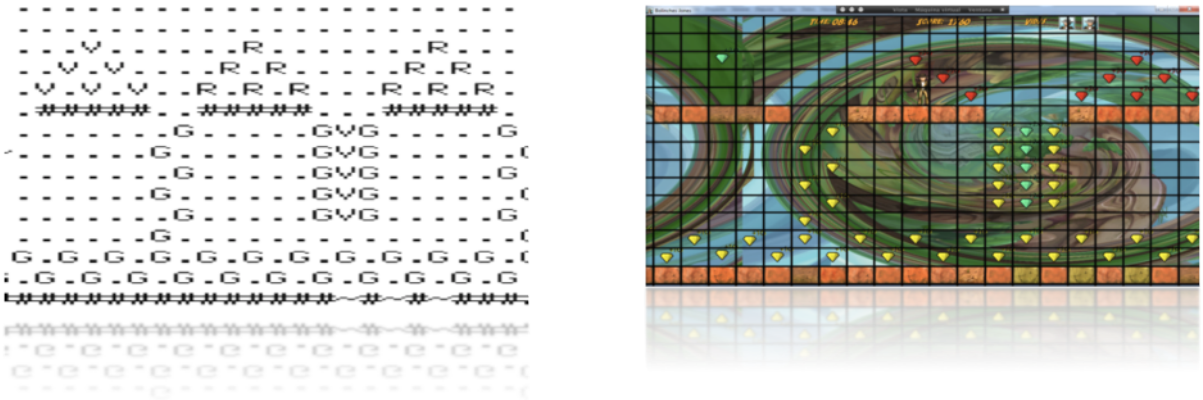


Figura 4.1.1. Representación de un nivel mapeado mediante un *maptiles* (izquierda) siendo un documento de texto y (derecha) gráficamente..

El uso de esta técnica permite separar 2 grandes aspectos en el desarrollo de un videojuego como es la especificación del juego y el motor que lo apoya en tiempo de ejecución, ver figura 4.1.2. Siguiendo un aproximación MDA usando el modelo PIM, cuando el diseñador especifique el gameplay, la GUI, el Control y el Diseño de Niveles del juego es transformado a un PSM llamado Tile Engine obteniendo un modelo específico de la plataforma con la especificación del juego que hace uso de los conceptos del motor.

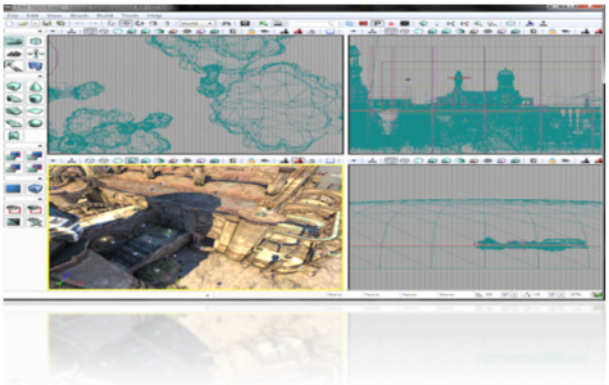


Figura 4.1.2. Esquema del acoplamiento del juego con el motor de *tiles* 2D.

4.1.1 *Tiles* Puras

El desplazamiento del personaje se limita a la posición de las *tiles* y nunca puede estar parado a mitad de un camino entre representaciones de dos tiles.

Cuando el personaje queda parado en el camino y rompe la simetría de la alineación con la *tile* pura, el personaje sufre un ligero movimiento como si de una animación se tratara, pero por lo que la lógica del juego se refiere, el jugador estará siempre en la parte superior derecha de una *tile* específica [19]. Esta es la manera más fácil de poner en práctica la técnica de las *tiles* puras en un juego de plataformas.

Se imponen también fuertes restricciones sobre el control del personaje, por lo que es apto para juegos tradicionales basados en la acción de plataformas estáticos y dinámicos entre otros.

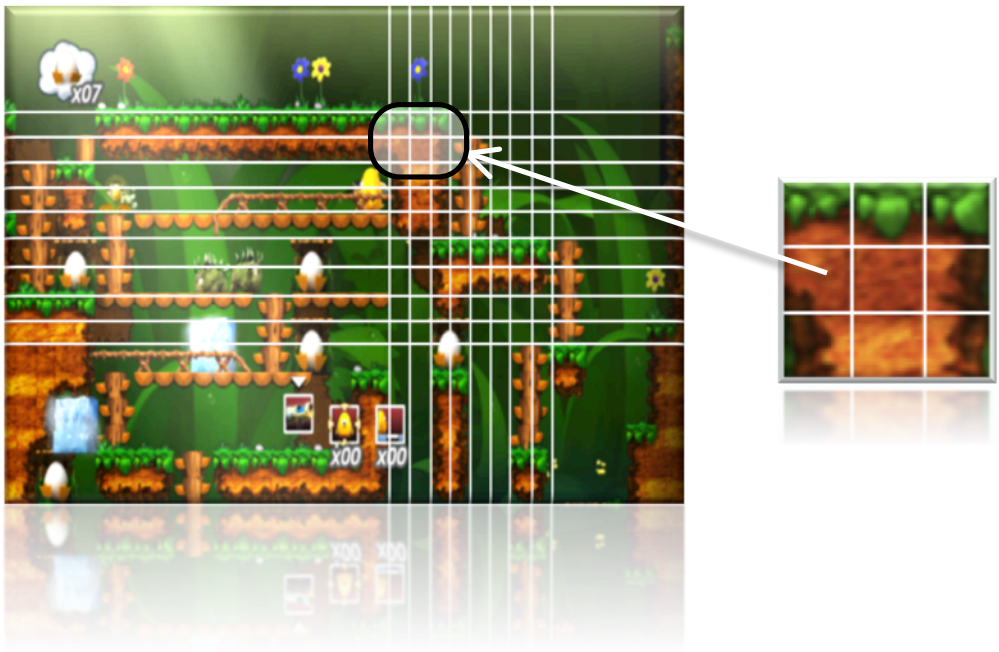


Figura 4.1.1.1. Representación delimitada por *tiles* puras en el juego Toki Tori.

El mapa es una cuadrícula de *tiles* que contiene los caracteres *unicode* que después serán representados gráficamente, cada una tiene almacenado su comportamiento, de esta manera podemos saber si se trata de un obstáculo o no. El jugador y otros caracteres están representados por un conjunto de uno o más *tiles* que se mueven juntas. En la figura 4.1.1.1, el jugador obtiene una representación de 2 × 2 tiles haciendo referencia a un *sprite* en forma de un pollito con patas, en cambio las *tiles* que forman parte del escenario como bloques sólidos obtienen una representación uniforme a lo largo del diseño del nivel.

En este tipo de juego, el jugador rara vez se mueve en diagonal, pero si lo hace, el movimiento se puede descomponer en dos pasos diferentes. Del mismo modo, es probable que sólo se mueva el personaje en un paso y apreciemos una pequeña animación.

Este tipo de movimiento es muy poco adecuado para los juegos que se desarrollan mediante las *tiles* puras, por lo que es muy raro que en juegos de este género se tengan inclinaciones, pendientes o saltos parabólicos entre otros. Lo adecuado en estos casos es dotar a los personajes del juego de saltos verticales u horizontales linealmente (Prince of Persia, Flashback).

Este sistema no permite que los movimientos de los personajes sean de menos de la medida de una *tile*, pero puede ser mitigados de diferentes maneras.

4.1.2 *Tiles* Suaves

Los personajes pueden moverse libremente alrededor de todo el nivel y su comportamiento viene dado por el tipo de colisión al que hace referencia el *tilemap*. Normalmente se usan tiles de 8 x 8, 16 x 16, 32 x 32 y 64 x 64 *pixels* pudiendo alternar el ancho y el alto de varios formatos. Por ejemplo, en el juego de Sonic the Hedgehog, las *tiles* que componen el nivel obtienen un tamaño de 16 x 16 mientras que el personaje principal mide 32 x 16 *pixels*. Esta es la forma más común del desarrollo de juegos de plataformas para consolas de 8 bits y 16 bits en la época de los 90, y sigue siendo las más popular hoy en día, ya que es fácil de implementar y hace que el nivel de edición sea más simple que las técnicas más sofisticadas. Este tipo de representación de *tiles* suaves es una evolución de las *tiles* puras, pudiendo ser éstas las que permiten implementar escalones, pendientes suaves y arcos de salto como veremos más adelante.

Este sistema de representación es muy flexible, relativamente fácil de implementar y da un mayor control en el desarrollo del juego ya que permite el diseño de cualquier tipo de representación de una *tile*. No es de extrañar que la mayoría de los juegos de plataformas de acción de todos los tiempos se basan en este tipo de *tiles*. Un claro ejemplo de ello se muestra en la figura 4.1.2.1.

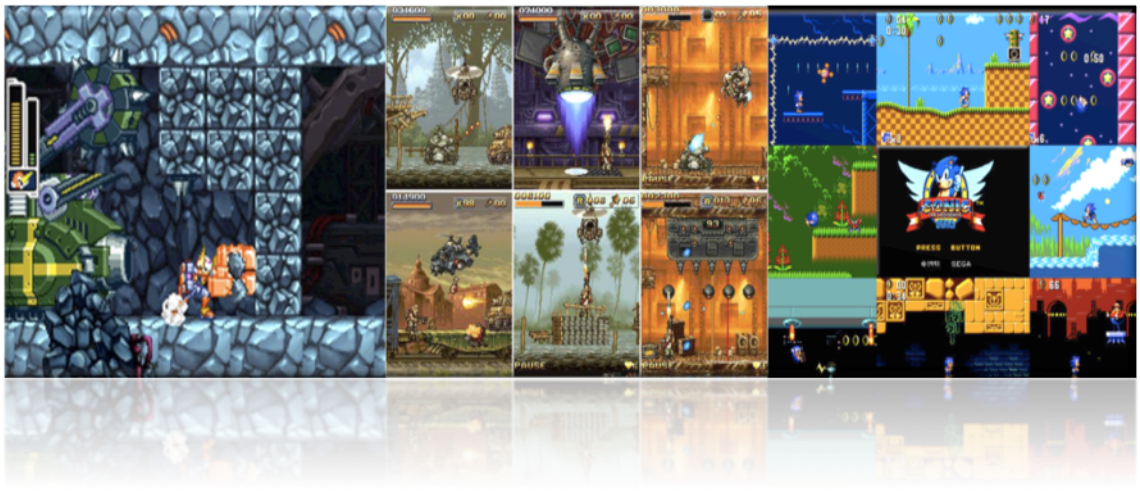


Figura 4.1.2.1. Ejemplo de juegos desarrollados mediante *tiles* suaves, Megaman DS (izquierda), Saga Metal Slug movile (centro) y Sonic the Hedgehog (derecha).

Su representación utiliza un mapa de información que almacena de la misma forma la técnica de las *tiles* puras, la diferencia es meramente en cómo los personajes interactúan con el fondo. El *Hitbox* (rectángulo que envuelve al personaje para la detección de colisiones) es ahora alineado con el eje del cuadro delimitador del *maptile*, es decir, un rectángulo que no se puede girar y que por lo general sigue siendo un múltiplo entero del tamaño de la tile. En muchos casos, el *sprite* del personaje en sí es más grande que el *hitbox* lógico, ya que esto lo convierte en una experiencia visual más agradable y más justa para el juego [9].

En la figura 4.1.2.2, se puede apreciar como el *sprite* mide 2 *tiles* de ancho y 2 *tiles* de largo, esta delimitado por un cuadrado blanco con los bordes rojos que delimita el tamaño, en

cambio su *hitbox* mide poco menos que 2 *tiles* de alto y 1 *tile* de ancho y esta delimitado por un cuadrado verde con los bordes negros. En los juegos no solemos apreciar el *hitbox* porque esta oculto, simplemente se utiliza para cálculos matemáticos de colisiones entre las *tiles* del juego.

Figura 4.1.2.2. Ejemplo de *hitboxs* en el juego Three Wonders.

4.1.2.1 Pendientes

Las pendientes permiten a nuestros personajes ascender y descender tanto hacia la izquierda como a la derecha mediante *tiles* sin hacer uso de saltos. Cuando el personaje de la figura 4.1.2.1.1 quiere moverse hacia la derecha y las *tiles* no están alineadas, por obligación se tiene que saltar pero gracias a las pendientes el personaje puede desplazarse sigilosamente. El único problema viene dado por el movimiento del *sprite* ya que hay rectificar los limites de la colisión del *hitbox* y situarlo en el centro del personaje respecto de las *tiles* inclinadas [21].

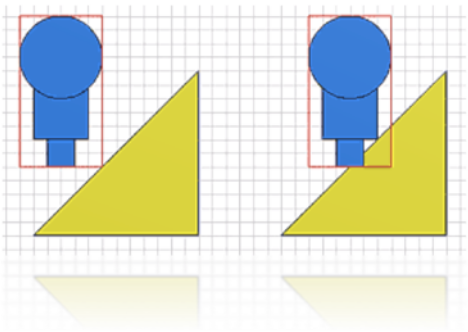


Figura 4.1.2.1.1. Ejemplo del acoplamiento del *hitbox* con la pendiente de *tiles*.

Suponiendo un sistema de coordenadas, donde (0, 0) es en la parte superior izquierda, a continuación, la primera tile que pertenece a la pendiente es la {0, 3} (izquierda, derecha),

luego la *tile* en la que se sitúa el personaje de la figura 4.1.2.1.2 izquierda es la {4, 7}, y seguidamente las {8, 11}, y {12, 15}. Después de esta pendiente se puede apreciar otro ejemplo con una mayor inclinación, esto reside en la diferencia entre la izquierda y derecha de la *tile* inclinada. Para una mejor apreciación se muestra la figura 4.1.2.1.2 derecha.

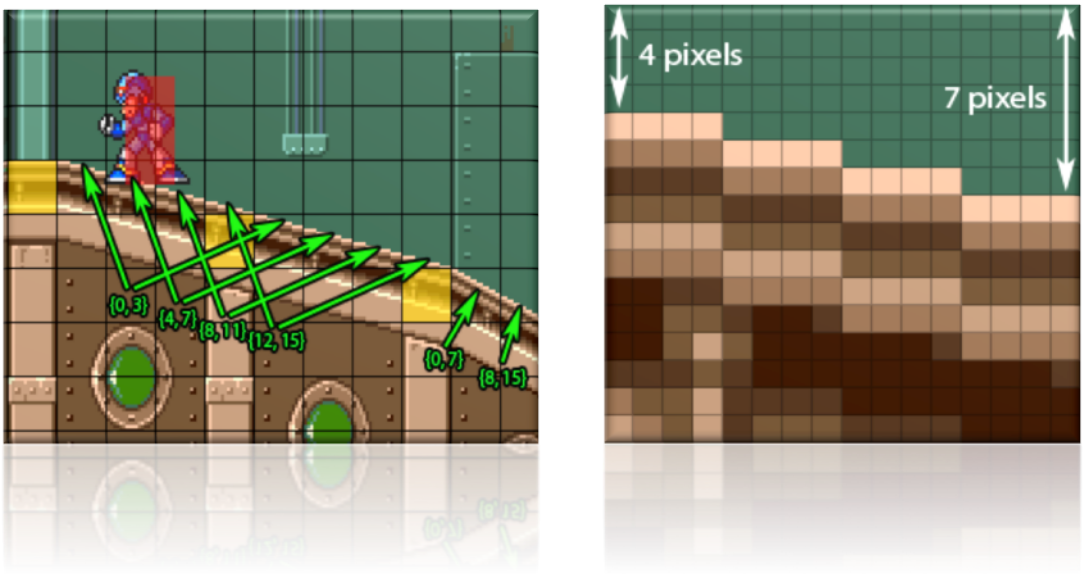


Figura 4.1.2.1.2. Ejemplo de pendientes de *tiles* (derecha) y detalle de la inclinación (izquierda).

4.1.2.2 Unidireccionales

Las *tiles* unidireccionales son las más usadas por los juegos que utilizan las técnicas de representación de *tiles*, no son más que bloques donde el personaje puede situarse por encima de la *tile* actuando cómo un bloque sólido impasable y que permite ser atravesada y transitable por la parte inferior de ésta.

En la figura 4.1.2.2.1 se muestra a Mario atravesando por abajo un *tile* que por la parte de arriba es impasable.

Figura 4.1.2.2.1. Ejemplo de *tile* unidireccional en el juego Super Mario Bross DS.

Algunos juegos permiten que el personaje salte hacia bajo. Hay algunas maneras de hacer esto, pero todas estas son relativamente simples y depende de la capacidad de cómo programarlo. Algunos ejemplos fáciles de citar sería que el personaje este encima de una *tile* impasable y esperar a la acción que desencadena que el personaje traspase la *tile*, una vez detectada esta acción podríamos anular la regla que hace que nuestra *tile* sea impasable. El personaje caería por inercia (puesto que disponemos de gravedad) hacia abajo activando nuevamente la primera regla, comprobado que el tamaño del *hitbox* de nuestro personaje ya no esta haciendo colisión con las coordenadas en negativo del eje Y de la *tile*.

4.1.2.3 Escaleras

Las escaleras son la forma más común de movimiento en los juegos de plataformas, puede parecer complicado la implementación, pero no es más que un estado alternativo, cuando estás en una escalera, se ignora la mayor parte del sistema anticolisión de un set de reglas, y se reemplaza con un nuevo conjunto de reglas. Las escaleras son por lo general de un tamaño proporcional al del *sprite* que conforma el personaje del juego como se aprecia en la figura 4.1.2.3.1.

Son dos las representaciones más usadas en el mundo de los juegos 2D, las escaleras verticales y las horizontales.

Las escaleras verticales son famosas por cambiar el aspecto del *sprite* , normalmente el personaje del juego camina hacia la derecha y el aspecto visual que se obtiene es el de un lateral. Cuando un personaje desciende o asciende en una escalera vertical, su aspecto visual cambia y se obtiene el rostro trasero del *sprite*.

Figura 4.1.2.3.1. Megaman 7, personaje principal descendiendo por las escaleras.

Las escaleras horizontales son una variación de las verticales, se ven en pocos juegos, pero sobre todo en la serie de Castlevania dado que son una característica principal de este juego. La aplicación es muy similar a la de escaleras de la vida real, con pocas excepciones como se aprecia en la figura 4.1.2.3.2.

Cuando el jugador mueve su posición sobre las *tiles*, éstas se dividen en 2 partes y la posición del personaje es un movimiento ascendente o descendente que le hace posicionarse en el centro de la *tile* haciendo un cambio visual del posicionamiento de las piernas en cada *tile* atravesada.

Cada "paso" hace que el jugador se desplace de forma simultánea en las coordenadas X e Y, en un valor preestablecido.

Otros juegos también tienen escaleras que se comportan como las pendientes. En ese caso, son simplemente una función visual.

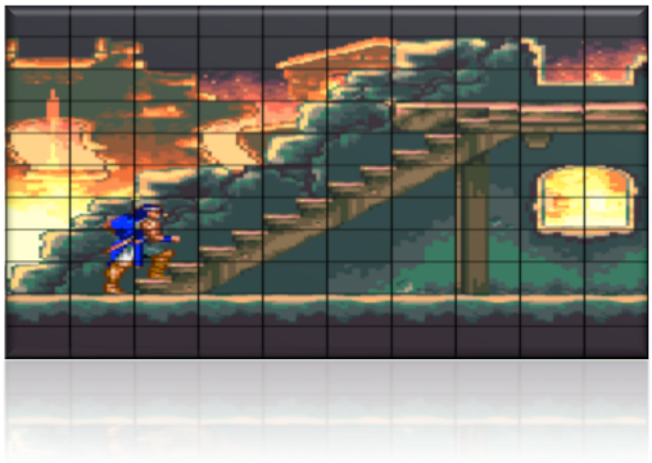


Figura 4.1.2.3.2. Castelvania: Dracula X, escaleras horizontales.

4.1.2.4 Plataformas en movimiento

Las plataformas móviles pueden parecer un poco complicado, pero son en realidad bastante simples. A diferencia de las plataformas normales, no pueden ser representados por las *tiles* fijas (por razones obvias), y en su lugar debe estar representado por una alineación fija sobre el eje delimitador, más conocido como AABB (Axis-Aligned Bounding Box) , es decir, un rectángulo que no puede girar (véase la figura 4.1.2.4.1).

Es un obstáculo habitual para todos los efectos de la colisión, y si el personaje se detiene, se tendrá que implementar un algoritmo para que el personaje se pueda mover a través de ella si se pretende acercar el movimiento a la realidad, sino la plataforma se moverá y el personaje quedará parado en el aire.

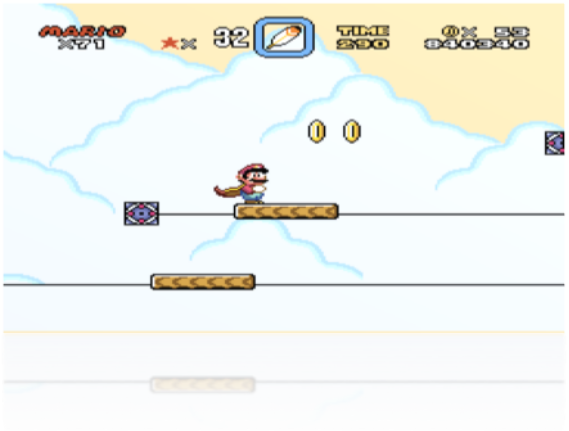


Figura 4.1.2.4.1. Super Mario World

4.1.3 *Tiles* Hexagonales

Las *tiles* hexagonales son también llamadas *Hextiles*. Son la evolución de una combinación de 4 *tiles* que unidas en una forma de 2 x 2 y utilizando el centro concéntrico de esta figura la representamos mediante un hexágono como muestra la figura 4.2.3.1.

Este tipo de *tiles* suelen ser utilizadas en los juegos de puzles y en juegos estratégicos de tableros [23], permitiendo entre otras cosas hasta 3 direcciones de movimientos manteniendo la distancia entre la *tile* pivote y sus vecinas.

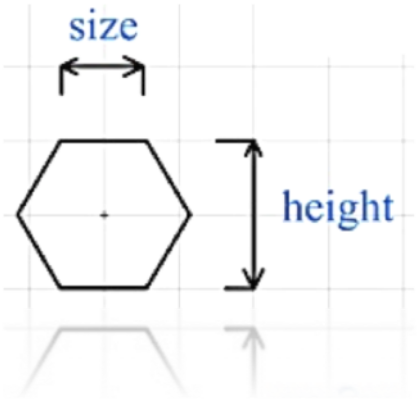


Figura 4.1.2.4.1. Tile hexagonal.

Uno de los factores más importantes a tener en cuenta en el uso de esta técnica son la altura y el ancho que utilizaremos para colocar las *tiles* adecuadamente en nuestro juego.

La representación se basa en una matriz de columnas x filas. De este modo, en una misma fila, cada dos elementos se desplaza la altura en la mitad del tamaño del *height* de la *tile* hexagonal. Por otro lado, la distribución de las columnas se hace de manera ordinaria y sin ningún cambio. La figura 4.1.2.4.2 muestra la distribución de esta técnica.

Figura 4.1.2.4.2. Puzzle Bobble / Bust-A-Move (Set2) de Neo Geo.

4.1.4 *Tiles* Bitmask

Este tipo de representación de *tiles* se asemeja a las *tiles* suaves pero en este caso se utilizan para detectar la colisión en el pixel de cada imagen.

Esto permite detalles más finos, pero aumenta significativamente la complejidad, el uso de memoria, y requiere algo parecido a un editor de imágenes para crear los niveles. También a menudo implica que las *tiles* no se puedan utilizar para efectos visuales, y por lo tanto puede requerir una diseño pesado e individual para cada nivel. Debido a estas cuestiones, esta es una técnica relativamente poco utilizada, pero puede producir resultados de mayor calidad comparándola con las técnicas anteriores. También es adecuado para entornos dinámicos y en escenario donde hayan explosiones, tales como los escenarios destruibles en Worms.

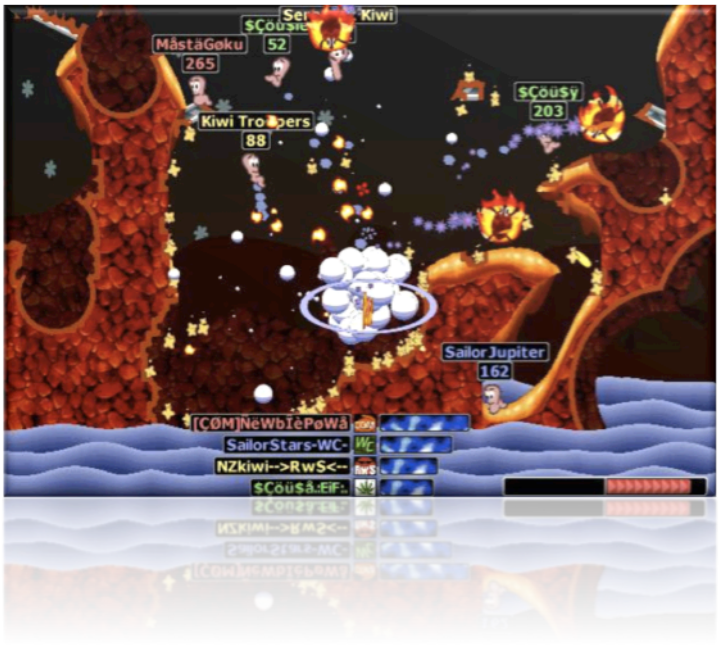


Figura 4.1.4.1. Worms Armageddon: Escenario creado mediante *tiles bitmask*.

4.1.5 *Tiles* Vectoriales

Esta técnica utiliza datos vectoriales (líneas o polígonos) para determinar los límites de las zonas de las colisiones. Es muy difícil de implementar correctamente, sin embargo es cada vez más popular debido a la ubicuidad de los motores de física, tales como Box2D, que son adecuados para la aplicación de esta técnica. Se ofrecen beneficios similares a la técnica de máscara de bits, pero sin sobrecarga de la memoria principal, y el uso de una manera muy diferente de la edición de niveles [13].

La manera de resolver el movimiento y colisiones son similares al método de máscara de bits, pero usando ángulos de polígonos para calcular la desviación y pendientes adecuadas. Normalmente se utiliza un motor de física para estos casos, ya que abordar el problema a mano es casi imposible.

Otra manera de lidiar esta técnica es dividir el personaje en varios polígonos, cada uno con diferentes funciones asociadas: el cuerpo principal en un polígono central, a continuación, un

rectángulo delgado para los pies, y dos rectángulos delgados para los lados, y otro para la cabeza o alguna combinación similar. A veces son cónicos para evitar quedar atrapados en los

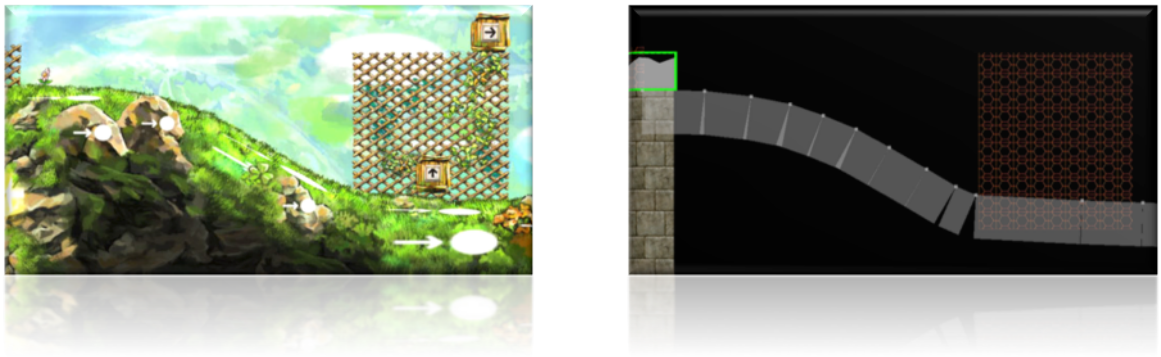


Figura 4.1.5.1. Braid, Graphic Level Editor, capas visibles (izquierda) y colisión de polígonos (derecha).

obstáculos. Cada polígono pueden tener diferentes propiedades físicas y diferentes respuestas a las colisión en los que pueden ser utilizados para determinar el estado de carácter. Esta técnica es muy utilizada en los juegos 3D donde se dota a los jugadores de esta técnica vectorial como muestra la figura 4.1.5.2.

Figura 4.1.5.2. Diseño vectorial de un personaje 3D.

4.2 Modelo Específico de la Plataforma, motor de *tiles* 2D

Normalmente, en el desarrollo de juegos se utiliza un middleware concreto que compila el código para las diferentes partes que componen la plataforma específica. En la utilización de este trabajo se ha utilizado un motor de *tiles* 2D que podrá ser transformado a diferentes lenguajes de programación. Con el objetivo de modelar los principales conceptos tecnológicos que se manejan al desarrollar juegos multiplataforma, se ha implementado un meta-modelo ECORE específico de la plataforma.

A continuación, se muestran las distintas perspectivas de la especificación de la interactividad del juego que han sido capturadas en el PIM y serán transformadas en varias vistas del modelo PSM que representa el motor de *tiles* 2D genérico (Apéndice II). La relación de trazabilidad entre los conceptos del modelo origen y destino no se pierden. Dado que el modelo PSM recoge características que son propias de la plataforma y que no están presentes en el PIM, el diseñador tendrá que introducir manualmente esta información que es específica de la plataforma. Esta información añadida se detalla en cada una de las perspectivas que se muestran a continuación.

Por simplicidad, se han conservado muchos de los nombres de las meta-clases definidas en el modelo PIM para hacer más sencilla la transformación al PSM.

4.2.1 Perspectiva del Manager de la GUI

GraphicDeviceManager es la primitiva de la interfaz gráfica de usuario que define cómo se administra la navegación (cómo se organiza la información de las diferentes pantallas del juego) y la distribución (qué tipo de información se detalla) de todas las pantallas definidas previamente en el modelo independiente de la plataforma. En esta clase se observa cómo se especifican ciertos atributos dependientes de la plataforma que nos permiten establecer los valores predefinidos del buffer tanto del ancho como el alto de la pantalla para que cuando el diseñador genere *ScreenNodes* tengan todos ellos unos valores preestablecidos.

La clase *ScreenManager* permite al modelo PSM que se muestra en la figura 4.2.1.1 capturar todos aquellos detalles que en la perspectiva de la Interfaz Gráfica de Usuario del modelo PIM se han definido previamente.

La clase *ScreenNode* ahora cuenta con atributos de posicionamiento de coordenadas para que el diseñador pueda especificar la ubicación exacta de la pantalla, pudiendo así, definir dos *GameScreens* para que 2 jugadores puedan compartir la pantalla tanto en horizontal como en vertical en caso necesario. Estos nodos siguen relacionándose entre si mediante transiciones declarando sus respectivos eventos de entrada o salida, tales como eventos propios del juego, interacciones de control o tiempo. La primitiva *ScreenNode* está compuesta por las clases *DisplayPrimitive*, *SpriteFont* y *SpriteBatchScreen*.

La clase *DisplayPrimitive* define qué tipo de información se detalla en la pantalla, permitiendo así, la definición de atributos que se posicionan en el *GameScreens* mediante la primitiva *SpriteFont* que es el diccionario de caracteres que forman estos atributos y está asociado al *Batch* de la pantalla.

La primitiva *SpriteBatchScreen* en la clase que permite dibujar en la pantalla. Está compuesta por la clase *SpriteBatchDrawScreen* que facilita el pintado de *texturas2D* tales como *overlays*, *background*... y la clase *SpriteBatchStringScreen* que permite el pintado de los atributos en modo texto ubicados en la pantalla.

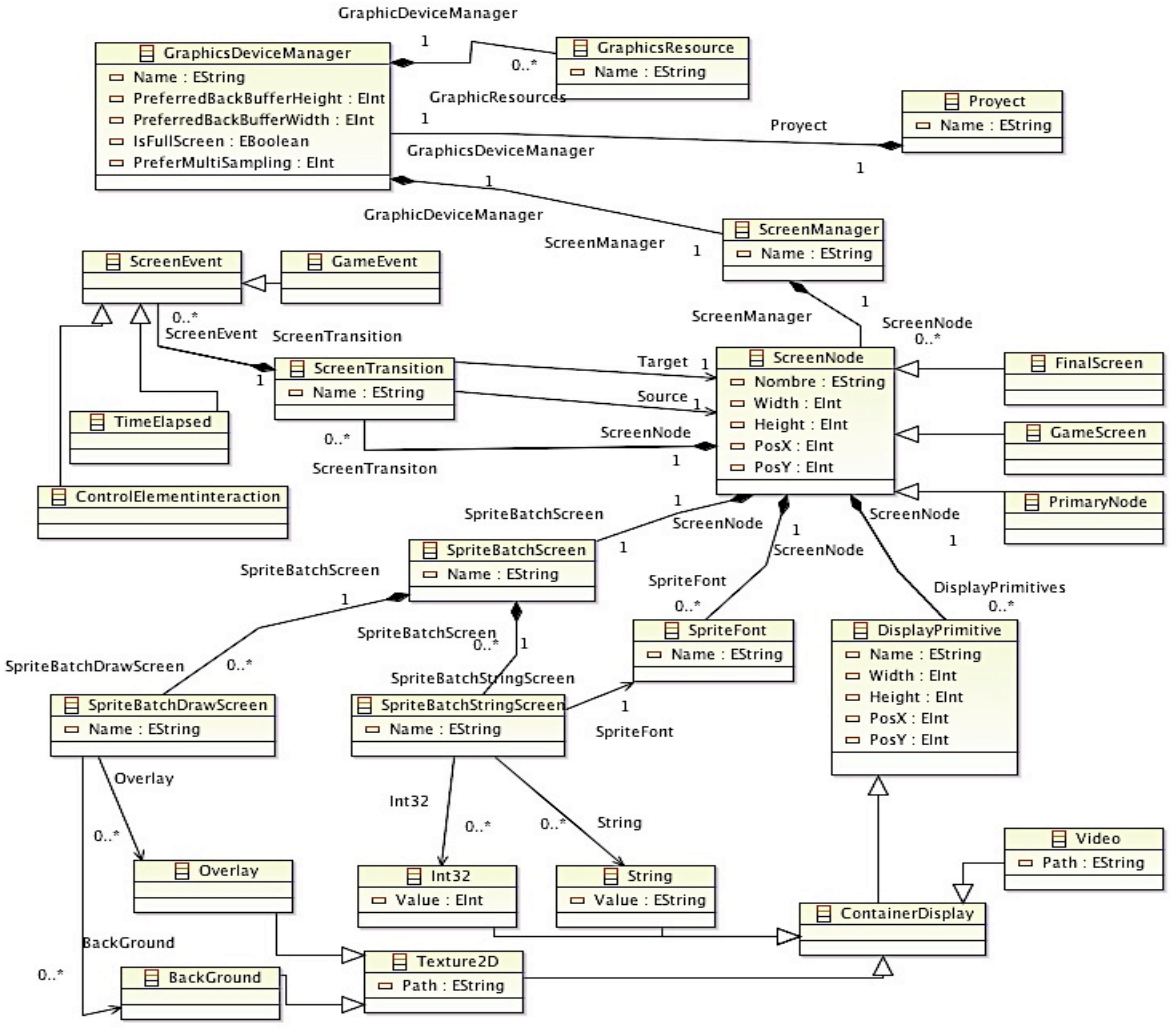


Figura 4.2.1.1. Meta-modelo del Manager de la GUI.

4.2.2 Perspectiva del Manager de Control

El manager de control define cómo se comunican los jugadores con el juego a través de dispositivos hardware. Dada la especificación de los controles en el modelo independiente de la plataforma en el que solamente se declara un *controller* como una clase que interactúa con el juego, ahora se trata de especificar mediante la especialización de un teclado o *Pad's* específicos de la plataforma como se muestra en la figura 4.2.2.1.

La clase *ControllerManager* permite gestionar los controles del juego. Está compuesta por la primitiva *Controller* que transforma la información definida en el modelo PIM en dos posibles tipos de controles específicos.

La clase *GamePadState* es un tipo de control definido como *pad*. En ella el diseñador puede especificar mediante sus atributos si el control pertenece al *player* 1 ó 2. Está compuesta por la clase *ElementButton* que a través de una clase enumeradora llamada *Buttons* especifica todos sus botones para asociarlo a una acción del juego.

La clase *KeyBoardState* en un tipo de control definido como teclado. Está compuesta por la primitiva *ElementKey* que permite especificar qué tipo de tecla va a tener asociado una acción del juego.

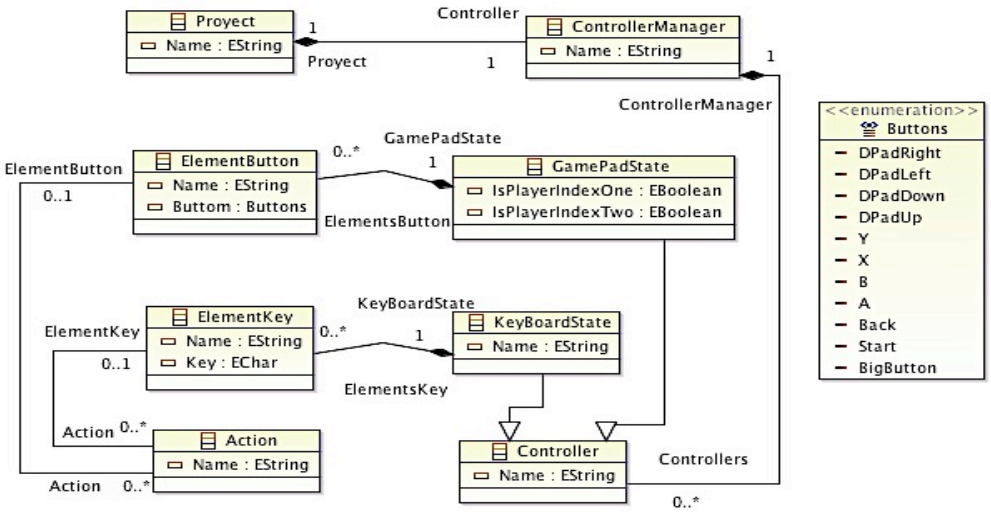


Figura 4.2.2.1. Meta-modelo del Manager de Control.

4.2.3 Perspectiva del Manager de Niveles

La clase *LevelManager* permite gestionar la distribución del diseño del juego procedente de las especificaciones del diseñador en el modelo PIM y está compuesta por las primitivas *World*, *GenericLevel*, *GamePlay* y *TransitionLevel*.

La clase *World* está asociada a transiciones entre mundos que permite la navegación entre éstos pudiendo declarar un mundo como un contenedor de niveles. Todo mundo debe de tener por obligación un nivel primario y final para poder avanzar al siguiente mundo. También esta asociado a la clase *GameScreen* del manejador de pantallas para poder visualizar el mundo con sus niveles. Debido al gran volumen que ocupa el modelo PSM no se ha podido mostrar en la figura 4.2.3.1 el enlace citado, este detalle y muchos otros se pueden apreciar en el Apéndice II.

La clase *GenericLevel* permite transformar los niveles que se han creando en el modelo PIM. Se le añaden atributos específicos tales como el largo y ancho máximo del nivel para delimitar el rango de la longitud de las *tiles* y atributos de posicionamiento. Esta compuesta por la clase *Layout* que transforma las clases procedentes del modelo PIM y por la clase *MapLevel* que el diseñador utiliza para definir el diseño del nivel a través de *tiles*. La especificación de un nivel se hace a través de un pequeño documento de texto que el motor de *tiles* transforma en un vista del *Layout* asociada a ese nivel.

La clase *Layout* está compuesta por *GameTileInstance* que son instancias de *tiles* que representan las entidades en el modelo específico de la plataforma.

La clase *AttributeInstance* permite instanciar los atributos previamente definidos en la perspectiva de la interfaz gráfica de usuario pertenecientes a la primitiva *DisplayPrimitive* así como los atributos que son manipulados en las reglas. Está compuesta por atributos que definen el tipo y valor asociado.

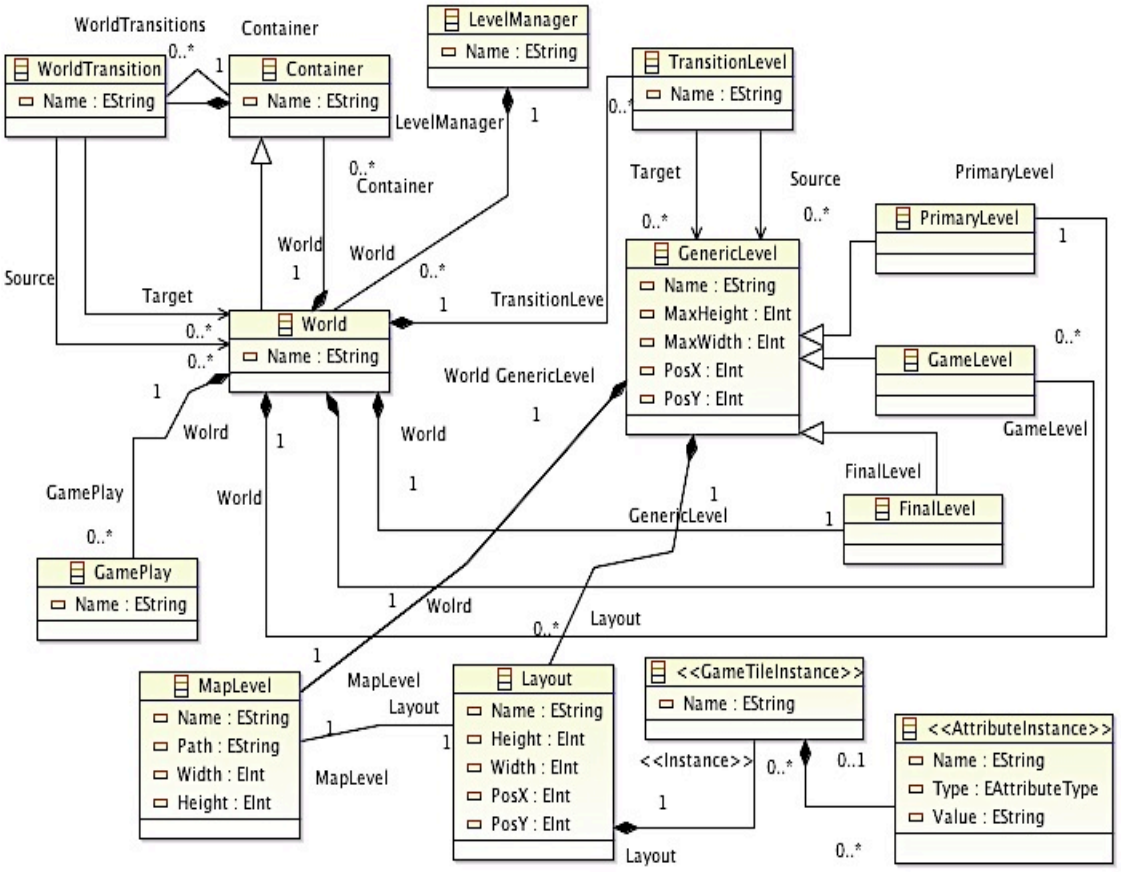


Figura 4.2.3.1. Meta-modelo del Manager de Niveles.

4.2.4 Perspectiva de *Tiles* y Reglas

Todas las entidades declaradas en el modelo PIM se transforman en *tiles* en el modelo específico de la plataforma como muestra la figura 4.2.4.1. Especificar de forma precisa como una entidad del juego se convierte en una *tile* permite al diseñador saber con creces cómo va ser el diseño de los elementos del juego.

Es en la clase *Tile* en la que se define el atributo Unicode que va asociado a la clase *MapTile* en la que se obtiene su representación visual. Se especifican ciertos atributos para caracterizar el comportamiento de una *tile* como es su aceleración y velocidad a través de vectores de dos dimensiones.

Un *tile* puede especializarse en una clase *Character* que a su vez especializa en las clases *PlayerCharacter* para indicar el personaje principal del juego y *NonPlayerCharacter* haciendo referencia a todos los personajes restantes. Es aquí donde la *tile* puede seguir especializándose siendo la representación de un bloque impasable (el jugador no puede atraversarlo), pasable (el jugador solo puede atravesarla por abajo y situarse encima de ella) y transparente (la *tile* es un elemento con el que el personaje no interactúa).

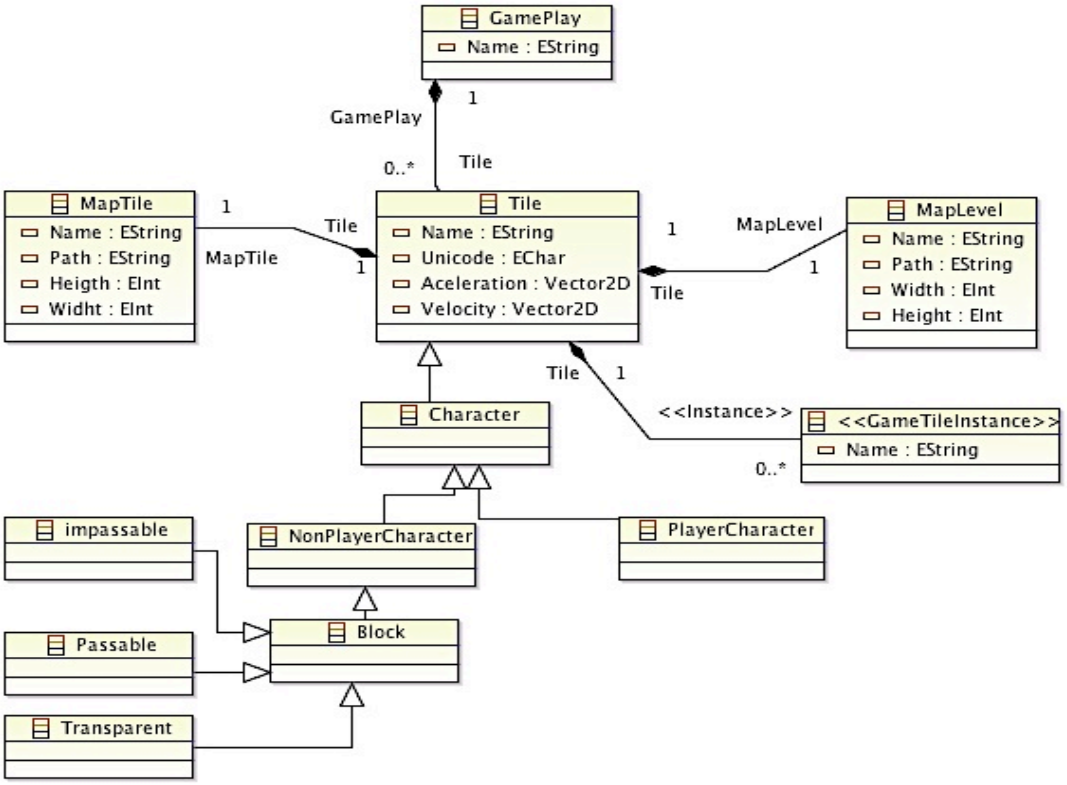


Figura 4.2.4.1. Meta-modelo de *Tiles*.

Las ventajas de este sistema de representación son la simplicidad y precisión. Puesto que los juegos están mejor estructurados, las interferencias son mucho menos probables, y las reglas del juego son más controladas, con menos necesidad de ajustar los valores según las circunstancias del *Gameplay*. La ejecución mecánica de ciertas *tiles* como el manejo de las repisas y las plataformas de una sola vía (ver figura 4.2.4.2), se convierte en proceso fácil, en comparación con los estilos de movimientos más complejos, simplemente se trata de comprobar si los *sprites* de los jugadores y las *tiles* del fondo están alineadas en el camino específico que permite una acción determinada.

A continuación, se detallan las principales primitivas de diseño para la especificación de la jugabilidad mediante reglas. Nótese que, la conceptualización no hace referencia alguna al tipo o género del juego, ni a su finalidad o propósito, ni siquiera se restringe su uso al diseño de videojuegos, pudiéndose especificar del mismo modo juegos tradicionales.

La Figura 4.2.4.3. muestra el meta-modelo que define los conceptos fundamentales para especificar un conjunto de reglas que definan la jugabilidad de un videojuego. Cada meta-clase del meta-modelo representa un concepto claramente definido en el diseño del comportamiento de la jugabilidad y, por tanto, requiere una explicación precisa para ser de utilidad en el diseño de juegos [18].

La primitiva *GamePlay* define el conjunto de reglas asociadas a un mundo y las *tiles* que componen la representación de un nivel.

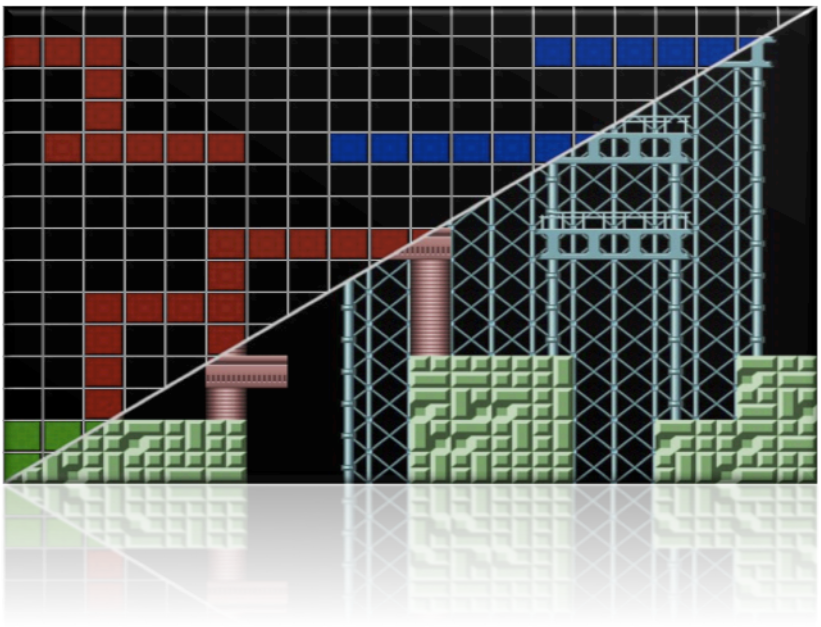


Figura 4.2.4.2. Representación de un sistema de *tiles*.

Se pueden diferenciar tres tipos de reglas, equivalentes a una intersección de los tipos de reglas definidas por Salen y Zimmerman [16] y Frasca [17]: reglas de acción, reglas de resultado y reglas internas.

Las reglas de acción, equivalentes a las reglas operacionales o de manipulación, son aquellas que permiten a los jugadores tomar parte en el juego. De forma más precisa, las reglas de acción definen qué consecuencias (post-condiciones) tiene cada una de las acciones de los personajes jugadores. Así, su pre-condición siempre contendrá al menos una acción.

Las reglas de resultado, equivalentes a las reglas de objetivo, son aquellas que definen las condiciones de victoria, esto es, qué objetivos deben perseguir o evitar los jugadores para ganar en el juego. De forma más precisa, las reglas de resultado definen qué pre-condiciones tienen como post-condición un evento de resultado. Así, las reglas de resultado siempre tienen al menos un evento de resultado como post-condición. Si la post-condición es de victoria, el objetivo será positivo y los jugadores tendrán que perseguir las pre-condiciones para ganar. Si la post-condición es de derrota, el objetivo será negativo y los jugadores tendrán que evitarlo para seguir jugando. Si la post-condición es de empate, los jugadores tendrá que perseguir o evitar las pre-condiciones en función de su estado de juego. O dicho de otro modo, cuando más cerca esté un jugador de alcanzar la pre-condición de derrota con más ahínco deberá perseguir la pre-condición de empate.

Las reglas internas son aquellas que definen el comportamiento del sistema interno de juego. Es decir, definen cómo reaccionan (post-condición) todas las entidades de juego pasivas frente a los diversos estímulos (pre-condiciones). La única restricción para las reglas internas es que no contengan acciones o eventos de resultado, pues pasarían a considerarse reglas de acción o resultado, respectivamente.

Cada regla *Rule* se especifica mediante una pre-condición y una post-condición. Cada una de estas condiciones *Condition* puede ser una condición simple o una expresión compleja. Las condiciones simples referencian un evento o expresan una operación aritmética sobre dos

atributos. Si la condición simple referencia un evento como pre-condición indica que al ser invocado el evento, la regla se ejecutará. Si la condición simple referencia a un evento como post-condición fuerza a que al ejecutarse la regla se invoque al evento referenciado. Si la condición simple evalúa una operación aritmética sobre atributos como pre-condición indica que al cumplirse la expresión la regla se ejecutará.

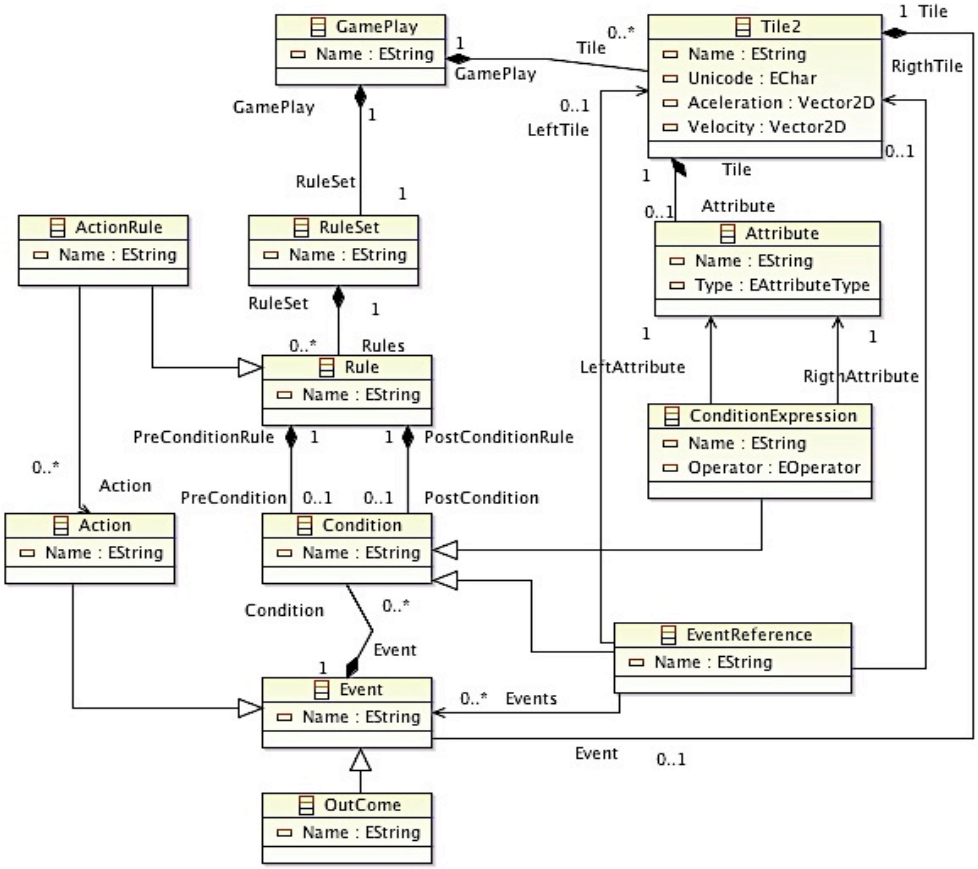


Figura 4.2.4.3. Meta-modelo de Reglas.

Si la condición simple evalúa una operación aritmética sobre atributos como post-condición fuerza a que al ejecutarse la regla se realice la operación sobre dichos atributos. De este modo, las sumas y restas no se permiten como pre-condiciones sino como post-condiciones. Las expresiones complejas relacionan dos condiciones mediante un operador lógico conjuntivo o disyuntivo

La clase *Event* representa sucesos disparados por la entidad de juego que cambia el estado del sistema de juego.

La clase *RuleSet* está compuesta por reglas que especializan en acciones. Éstas están asociadas a la clase *Action* que interactúan con los elementos de un control.

4.3 Transformación de modelos PIM to PSM

Capturar el conocimiento asociado a la codificación de la especificación de los videojuegos entre modelos permite transformar modelos PIM en modelos PSM de la plataforma destino. Las transformaciones QVT declarativas se han definido mediante la herramienta *MediniQVT*,

integrada en el entorno de desarrollo *Eclipse Modeling Tools*. La transformación PIM-PSM parte de una regla de transformación unidireccional en forma de árbol que mapea de padres a hijos todas las perspectivas existentes en la especificación del modelo PIM, creando en el modelo PSM un motor de *tiles* 2D.

De manera más precisa, a la hora de hacer las transformaciones entre modelos, el modelo PIM de origen chequea la existencia de cada una de las perspectivas declaradas en la especificación del multi-modelo que previamente son transformadas en las clases *Managers* del modelo PSM. La relación entre modelos se hacen mediante *mappings* y su cardinalidad es de 1 a 1 como muestra la figura 4.3.1.

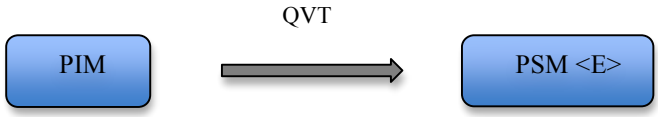


Figura 4.3.1. Relacion 1 a 1 entre modelo PIM y PSM

La transformación se inicia en la clase *Game* del modelo origen siendo la clase padre del meta-modelo y se crea una clase *Project* en el modelo PSM. La primera transformación se inicia en la perspectiva de la interfaz gráfica de usuario, después se mapean las perspectiva de control y del diseño de niveles respecivamente. Finalmente se transforman las perspectivas de las entidades y reglas.

A continuación, se detalla con mucha más precisión la transformación de cada una de las perspectivas del modelo PIM a las perspectivas manejadoras del modelo PSM (Apéndice IV).

4.3.1 Transformación de la Perspectiva de la Intefaz Gráfica de Usuario

El modelo PIM origen comprueba la existencia de la interfaz gráfica de usuario en la especificación de la misma. Una transformación en el modelo PSM genera automaticamente las clases manejadoras de los dispositivos gráficos para poder manipular los *buffers* de la dimensión de la pantalla principal, el tipo de *sampler* y si el juego va a disponer o no de pantalla completa. Seguidamente se crea la clase manejadora del *Screen* del juego en la plataforma específica.

Se recorren las clases *ScreenNode* y *ScreenTransition* en el modelo origen y se transforman en el modelo destino en dos clases con el mismo nombre. Ahora la clase *ScreenNode* del modelo destino obtiene nuevos atributos para que el diseñador pueda especifcar el tamaño de alto y ancho de las pantallas que a modo de menú pueden ser representadas. Las clases *Screen_Primary*, *Game_Screen* y *Screen_Final* son transformadas en las clases *PrimaryNode*, *GameScreen* y *Final Node* respectivamente.

El modelo PIM define una clase de eventos que también es transformada en el modelo origen, esta tranformación no sufre ningún cambio y se conservan los nombres que previamente se habían definido en el modelo origen, *GameEvent*, *ControlElementInteraction* y *TimeElapsed.*

La transformación que más cambios sufre es la que identifica el *display* de la pantalla como muestra la figura 4.3.1.1. El modelo origen no define cómo se van a representar algunos de los

elementos del juegos, tales como el video, *overlays*, *backgrounds*…. Es en el modelo PSM donde se crea un *SpriteBatch* que permite la representación de todo tipo de elementos, diferenciando entre *sprites* planos y *drawSprites*. Los *sprites* planos son fuentes de texto alfanuméricas que reciben el nombre de *SpriteFonts* y se representan en la pantalla del juego gracias a la clase *SpriteBatchStringScreen* que se crea previamente. Los draw*Sprites* son texturas 2D que permiten la representación de todo tipo de imágenes.

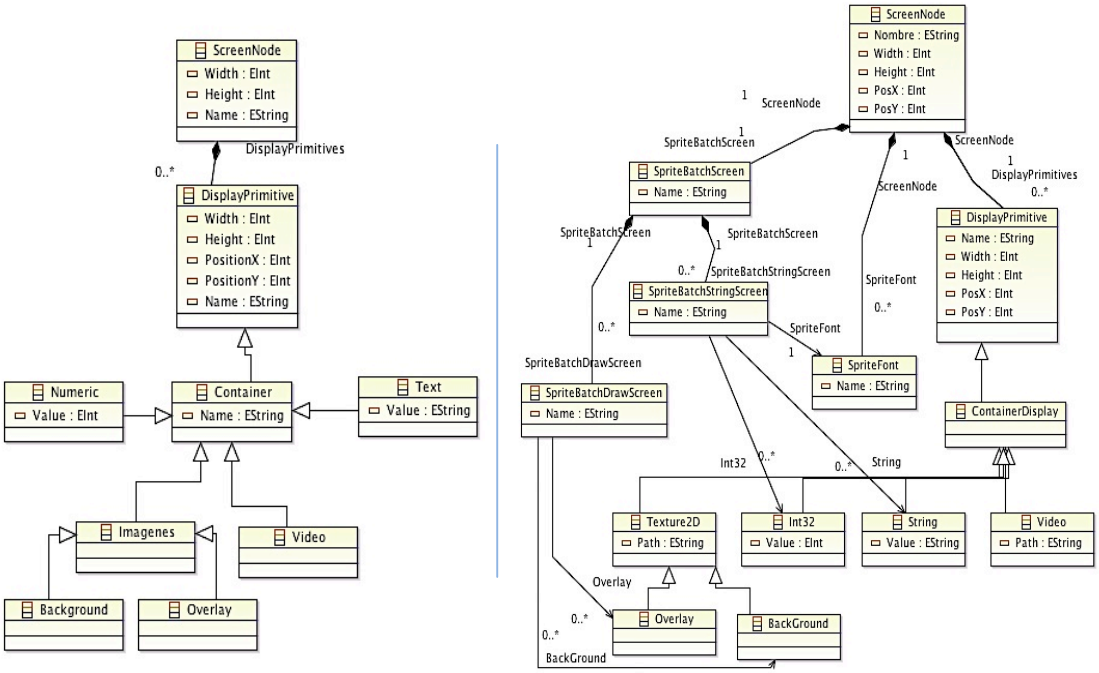


Figura 4.3.1.1. Transformación de la perspectiva de la Interfaz Gráfica de Usuario, (izquierda) modelo PIM y (derecha) modelo PSM..

4.3.2 Transformación de la Perspectiva Control

La transformación de la perspectiva de control del modelo origen empiezan con la comprobación de la clase *Controller* obteniendo la información del tipo y de la dimensión del control. Es transformada en el modelo destino en una clase manejadora llamada *ControllerManager* que es la encargada de gestionar todos los dispositivos de control. Se relaciona con la primitiva *Controller* que especifica el tipo de control que precede el juego según la información detallada en el modelo PIM.

La clase KeyBoardState y GamePadState heredan de la clase *Controller* en el modelo específico de la plataforma. Ambas clases son las encargadas de especificar las acciones de los *Characters* del juego una vez realizada la transformacion.

La clase KeyBoardState se encarga del teclado físico cuando se trata de juegos de ordenador. Mediante el pulso de una tecla se define la clase *ElementKey* que permite las acciones de los personajes. Esta transformación en el modelo PSM necesita de los conocimientos específicos del diseñador dado que se necesita completar ya que esta información no se recoge en el modelo PIM.

La clase *GamePadState* permite definir hasta 2 jugadores y como sucede con la clase *KeyBoardState*, también permite gestionar las acciones de los personajes del juego a través de la clase *Button* que define los botones del *pad*. Esta transformación también necesita de los conocimientos del diseñador.

La figura 4.3.2.1 permite apreciar la transformación de la perspectiva de Control del modelo PIM a la de perspectiva del Manager de Control del modelo PSM.

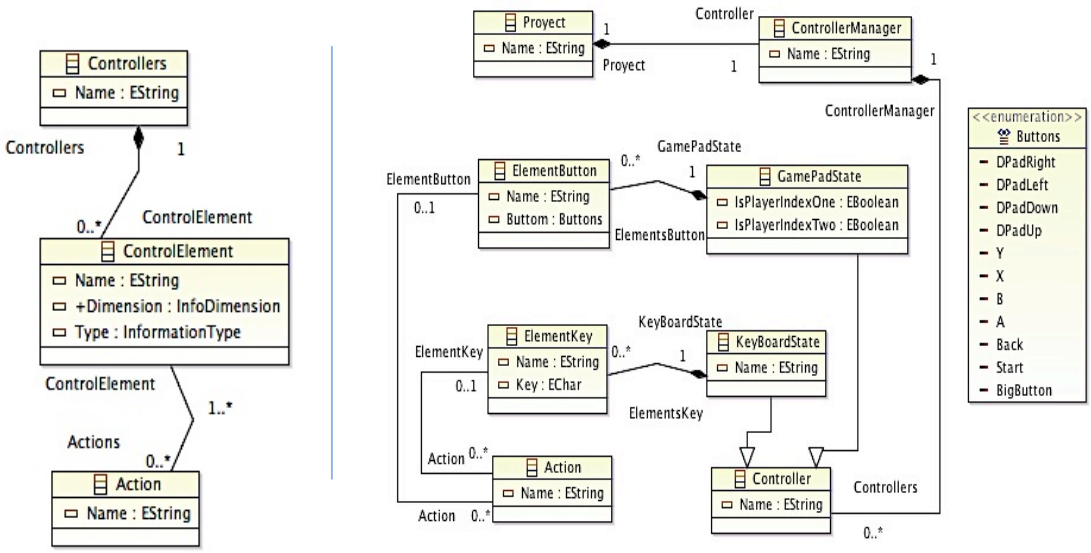


Figura 4.3.2.1. Transformación de la perspectiva de Control, (izquierda) modelo PIM y (derecha) modelo PSM.

4.3.3 Transformación de la Perspectiva del diseño de niveles

El modelo PSM destino obtiene una transformación casi en su totalidad igual a la del modelo PIM origen. Se comprueba la existencia de la clase *World* que es la encargada de crear el diseño del nivel de manera independiente de la plataforma. Es transformada en la clase *LevelManager* que gestiona todos los mundos con sus respectivos niveles y la clase *World* del modelo PSM.

Los clases *TransitionLevel*, *TransitionWorld*,, *Container*, *PrimaryLevel*, *GameLevel*, *SecondLevel y GamePlay* conservan la misma nomenglatura en la transformación del modelo destino que en el de origen para poder ser más sencillo su entendimiento.

Una vez creado todos estas meta-clases, el modelo PIM verifica la clase *Layout* y la transforma al modelo destino añdiendo más información que el diseñador debe completar como es el alto y el ancho del perímetro de la capa y el posicionamiento en el eje de abscisas X e Y. Es en este momento cuando se transforma la clase *GameEntityInstance* del modelo independiente en *GameTileInstance* de modelo específico. Podemos observar como estas instancias en el modelo independiente de la plataforma reciben el nombre de entidades y son transformadas a tiles en el PSM.

De forma similar, los atributos instanciados en el PIM son transformados con el mismo nombre en el modelo PSM. Se añaden atributos específicos tales como el tipo y el valor para especificarlos de manera concreta. De este modo un atributo creado en el PIM haciendo

referencia a las “vidas” de un jugador, en el modelo específico de la plataforma seguirá siendo llamado “vida”, con el añadido del tipo, en este caso entero de tipo *Integer*.

Cuando se ha comprabado la existencia de la clase GenericLevel y Layout en el modelo origen, se crea la clase MapLevel en el modelo destino. Esta clase se asocia a la clase Layout y permite obtener mediante un documento de texto la estrutura externa del nivel junto con las dimensiones de su tamaño.

La figura 4.3.3.1 permite apreciar la transformación de la perspectiva del diseño de niveles del modelo PIM a la de perspectiva del Manager del diseño de niveles del modelo PSM de forma resumida.

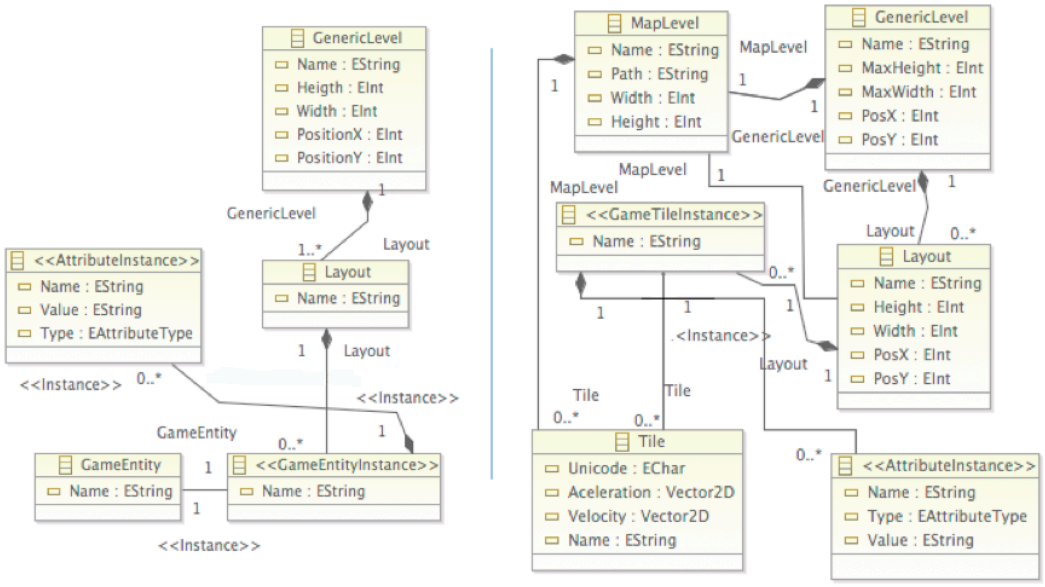


Figura 4.3.3.1. Transformación de la perspectiva del diseño de niveles, (izquierda) modelo PIM y (derecha) modelo PSM.

4.3.4 Transformación de la Perspectiva de Entidades y Reglas

La transformación de la perspectiva de Entidades y Reglas del modelo origen empieza con la comprobación de la clase *GamePlay* que no sufre ninguna modificación en el modelo destino.

Las entidades en el modelo PIM son transformadas en *tiles* en el modelo PSM. Se le añaden atributos tales como el Unicode y vectores 2D para la aceleración y velocidad. Automaticamente se crea la clase Maptile que permite a la *tile* especificar sus dimensiones de alto y ancho, también permite añadir la ruta de la imagen que representa la tile en el juego.

Las clases *Character* y *Block* son tranformadas con los mismos nombres en el modelo específico de plataforma.

La clase *GameEntityInstance* se verifica en el modelo origen y se transforma a la clase *GameTileInstance* que permite prepresentar las instancia de la *tile* en el *layout* de un nivel.

Las perspectiva de las reglas en el modelo destino conserva los mismos nombres que en el modelo origen para poder comprender con mayor facilidad la transformacion. Se verifica la clase *RuleSet* que contiene las reglas del *GamepPlay* de cada mundo, luego se verifican las condiciones y eventos que referencian la *tile*.

La figuras 4.3.4.1 y 4.3.4.2 permiten la visualización de la transformación de la perspectiva del Entidades y Reglas del modelo PIM a la de perspectiva de Tiles y Reglas del modelo PSM.

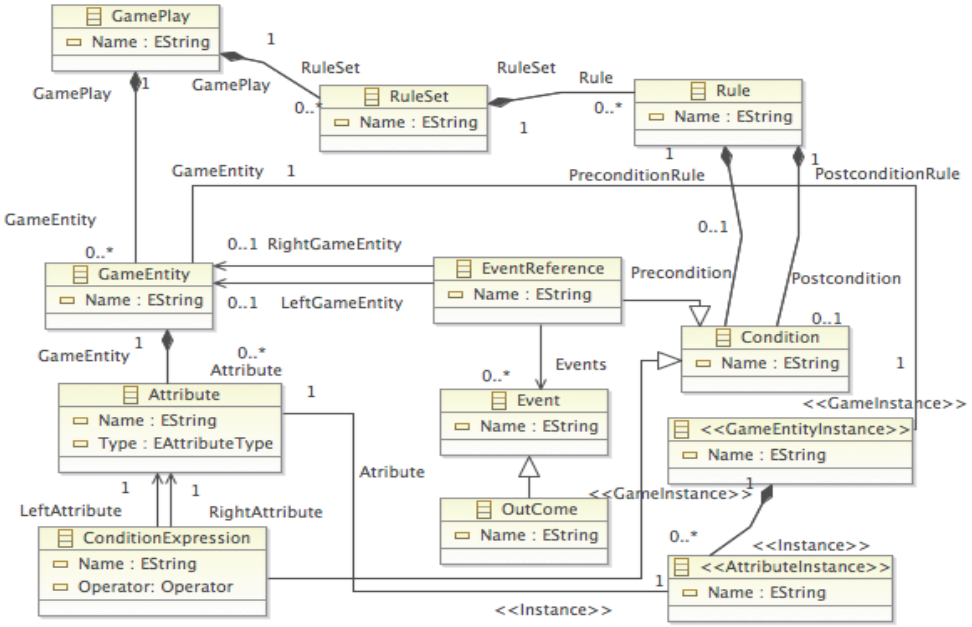


Figura 4.3.4.1 Perspectiva de Entidades y Reglas en el modelo PIM.

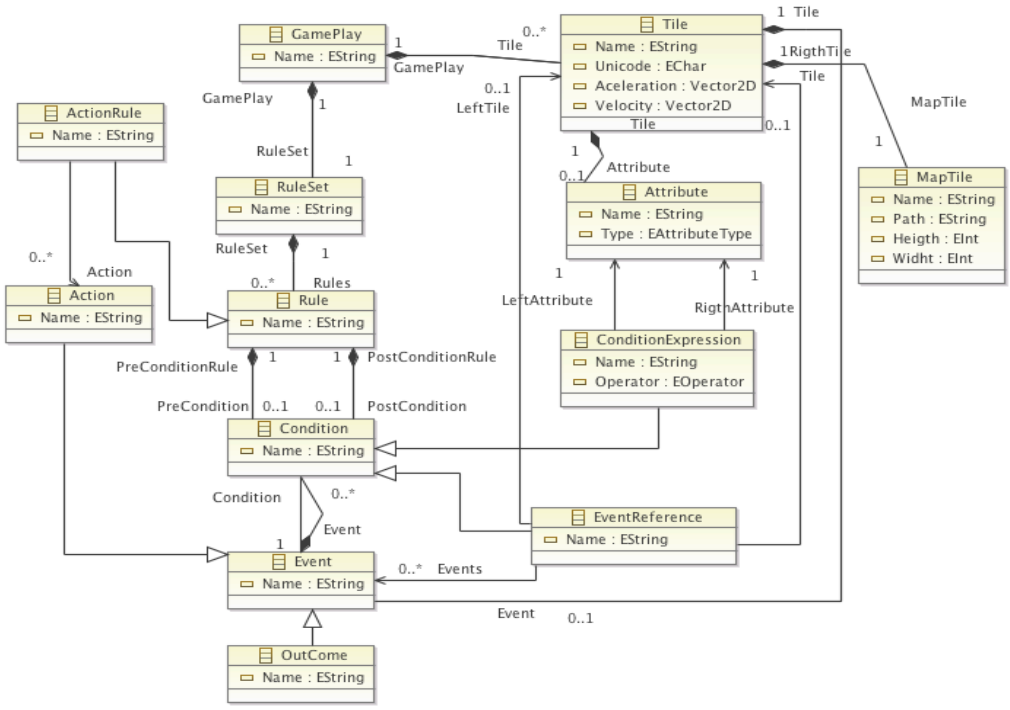


Figura 4.3.4.2. Perspectiva de Tiles y Reglas en el modelo PSM.

4.4 Transformación de modelo PSM to Texto

Los conceptos plasmados en el modelo PSM que representan el juego apoyado en un motor de *tiles* 2D genérico simboliza la descripción del sistema en términos de una plataforma específica. La utilización de un sencillo compilador de modelos PSM nos brinda la oportunidad de generar código para cualquier plataforma como muestra la figura 4.4.1.

En nuestro caso de uso se ha elegido generar código plano en C# correspondiente a la plataforma tecnológica de *Microsoft XNA Game Studio*.

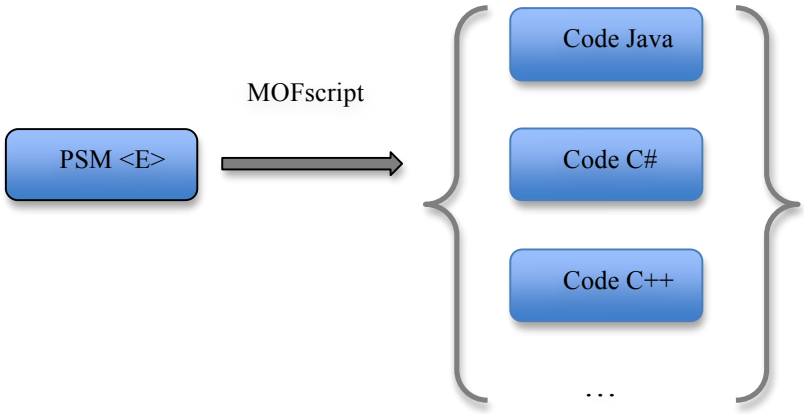


Figura 4.4.1. Relacion 1 a N entre el modelo PSM y la generación de Texto.

Esta transformación modelo-texto genera las estructuras de datos y las clases C# necesarias para poder ser integradas con el motor de *tiles* 2D mediante un lenguaje *MOFscript*. Dado que los conceptos del meta-modelo PSM son muy próximos a los conceptos utilizados, se detallará a continuación la metodología de uso de algunas partes del código de transformación del compilador de modelos PSM a código C# del método principal de los juegos generados. Se muestra en el Apéndice V.

La generación de código de esta clase se basa en la llamada a métodos definidos como *Macros*, éstas no son más que una serie de instrucciones que se almacenan para que se pueda ejecutar con una sola llamada de manera secuencial mediante una sola instrucción. Esta llamada permite la automatización de tareas repetitivas y libera al programador de labores tediosas y complejas a la hora de repetir instrucciones, también permite generar de manera más robusta y eficaz el código que genera el compilador de modelos.

La macro *Espaciado* permite al compilador de modelos invocar a un método que desplaza 'n' tabulaciones el código hacia la derecha que se va a generar de manera automática, esta macro libera al programador de una labor tediosa dado que cuando se genera código se necesita que este bien estructurado y ordenado. El código de la macro se muestra a continuación:

```
psm.Class::Espaciado(Integer numero)
 {
  String str;
  while(numero>0){
        str+="\t";
        numero=numero-1;
    }
  print(str);
    }
```

La macro *Salto* permite al compilador llamar a un método que hace 'n' saltos de líneas del código de las instrucciones que se van a generar automáticamente. Este método permite al programador definir el diseño de las estructuras del código de las clases.

```
psm.Class::Salto(Integer numero)
 {
  String str;
  while(numero>0){
        str+="\n";
        numero=numero-1;
  }
  print(str);
 }
```

La macro *Remarks* sirve para que el compilador pueda añadir anotaciones al código generado de la plataforma *Microsoft Visual Studio .Net.* Permite agregar información sobre un tipo, de modo que completa la información especifica con la *Macro* <*Summary*>. La ventaja de usar la macro *Remarks* es que estas anotaciones pueden servir para generar un documento xml.

```
psm.Class::Remarks( String str)
 {
        println("\n\t\t\t/// <remarks>");
        println("\t\t\t/// "+str+"");
        println("\t\t\t/// </remarks>");

 }
```

La macro *Summary* es muy parecida a la macro *Remarks*, su única diferencia es que permite obtener información sobre el tipo con la herramienta de *Microsoft IntelliSense* para la generación de documentos xml.

```
psm.Class::Summary( String str)
 {
        println("\n\t\t\t/// <summary>");
        println("\t\t\t/// "+str+"");
        println("\t\t\t/// </summary>");

 }
```

La macro *attributeGetterSettersCsharp* permite al compilador de modelos crear las propiedades de modificación y consulta de los atributos de la clase generada de manera automática, esto permite reforzar las clases generadas dado que todos los atributos de la clase cuentan con propiedades.

```
psm.Class::attributeGetterSettersCsharp(a:psm.Attribute) {
    self.Espaciado(3);'public 'a.Type' 'a.Name.firstToUpper() self.Salto(1)
    self.Espaciado(3);'{' self.Salto(1)
    self.Espaciado(4);' get { return this.'a.Name '; }' self.Salto(1)
    self.Espaciado(4);' set { this.'a.Name'= value;  }' self.Salto(1)
    self.Espaciado(3);'}' self.Salto(1)
 }
```

Finalmente, como hemos observado el motor de *tiles 2D* está implementado mediante *Microsoft XNA Game Studio*, puesto que ofrece un *framework* con simples primitivas específicas para el desarrollo de videojuegos para tres plataformas de juegos distintas como: PC, XBOX 360 y WP7.

4.4.1 Estructura Interna del Nucleo del Motor de *Tiles* 2D

Dado nuestro planteamiento de trabajo, se ha utilizado un modelo PIM para plasmar todos los conceptos de la interactividad de un videojuego a través de las perspectivas de jugabilidad, Control, IGU y Diseño de Niveles entre otras. Estos conceptos han sido transformados en un modelo PSM que simboliza un motor de *tiles* 2D que representa la totalidad del videojuego en su fase final.

Por simplicidad, se ha representado un meta-modelo de la estructura interna del motor con las perspectivas más importantes como son la perspectiva del manager de la IGU, Control y Diseño de Niveles dónde se explica con detalle su comportamiento semántico.

La utilización de las clases en el meta-modelo representadas mediante interfaces permite definir métodos abstractos y propiedades que especifican qué debe hacer el método pero sin su implementación. Son las clases definidas en el modelo las que implementen estas interfaces para describir la lógica del comportamiento de los métodos declarados. Su uso atribuye al modelo ventajas como la organización visual de la representación de los conceptos del modelo y la obligación de que las clases que implementen ese interfaz utilicen los mismos métodos y atributos, además pueden ser usadas polimórficamente.

La especificación del uso de la genericidad en el meta-modelo permite desarrollar comoponentes de software que pueden ser reutilizados [30] en diferentes videojuegos, agiliza la producción y mejora la calidad de éstos, proporciona la parametrizacion de tipos, permitiendo definir estructuras de clases que pueden ser instanciadas de varias maneras.

A continuación se muestra las distintas perpectivas del comportamiento interno del nucleo de motor de *tiles* 2D. Para visualizar con más detalle la arquitectura del motor de *tiles* veasé el Apéndice III.

4.4.1.1 Perpectiva del Manager de la IGU

La perspectiva del manager de la interfaz gráfica de usuario define como se administra la navegación (cómo se organiza la información de las diferentes pantallas del juego) y la distribución (qué tipo de información se detalla) de todas las pantallas definidas previamente en el modelo independiente de la plataforma. Para aclarar más en detalle su comportamiento se detalla la descripción de cada una de las clases que la forman.

La primitiva *ScreenManager<E>* permite gestionar todas las pantallas definidas en el juego que se describen en el modelo PIM, implementa los métodos de la interfaz *IScreenManager<E>* que permite a clase *ScreenManager<E>* definir las propiedades y atributos para interactuar con la clase *<<ScreenInstance>>,* esta instancia está asociada a la clase *Screen* que se ha definido en el modelo PIM y se ha transformado en una clase llamada *Screen* especifica de la plataforma.

La clase <<*ScreenInstance*>> se asocia también a la instancia de las acciones definidas en los controles del juego, pudiendo ser una acción el pulsado de un botón del controlador del videojuego para efectuar la transición de una pantalla de Intro a la pantalla de menú.

La clase *Screen* del meta-modelo esta compuesta por la primitiva *DisplayPrimitive*<*E*> que define qué tipo de información se detalla en la pantalla, permitiendo así, la definición de atributos que posicionados en el display mediante coordenadas componen un screen del juego, esta primitiva esta asociada a la instancia de los atributos que asocia los atributos de las entidades del juego, dando lugar al disparo de eventos que componen esta entidad.

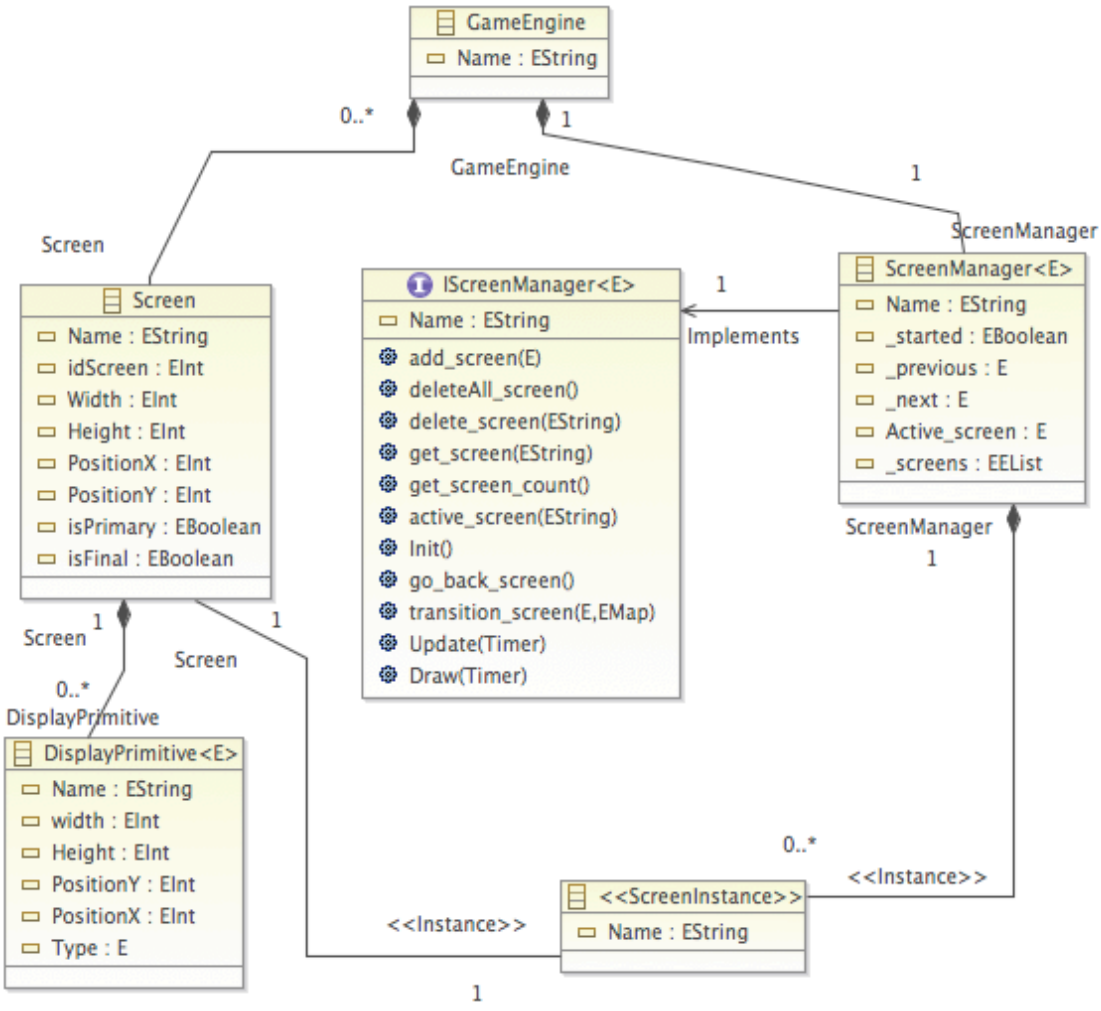


Figura 4.4.1.1.1. Perspectiva del manager de la IGU

4.4.1.2 Perpectiva del Manager de Control

La primitiva de control es la que define cómo se comunican los jugadores con el juego a través de lo controladores asociados al sistema. Estos controladores ofrecen a los jugadores elementos de control que permiten enviar información al sistema del juego, de esta manera es posible definir algunos conceptos de control independiente de la plataforma tecnológica elegida que mediante transformaciones plasman un elemento de control específico. A

continuación se muestra el funcionamiento de la estructura y comportamiento de la perspectiva del manager de controles sin entrar en detalles de implementación que representa la figura 4.4.1.2.1.

La clase *ControllerManager<E>* permite al motor gestionar y administrar todas las entradas y salidas activas del juego a través de dispositivos hardware, implementa la interfaz *IControllerManager<E>* que permite definir los métodos que utiliza la primitiva *ControllerManager<E>* para almacenar las instancias de las acciones del juego (pulsar, mantener, soltar un botón), estas instancias son asociadas a la primitivas *<<LevelInstance>>* de la clase Level, a la clase *<<ScreenInstance>>* que asocia una pantalla del juego y a las acciones que define los elementos de control del dispositivo. Los jugadores interactúan con estos elementos de control para ejecutar estas acciones del juego que se asocian con las reglas que define la jugabilidad del juego dentro de un nivel previamiente definido en la perspectiva de la jugabilidad del modelo.

Esta transformación obtiene toda la información de las acciones asociadas a las reglas del nivel definidas en el modelo independiente de la plataforma que nos permite mediante el administrador de controles, gestionar la información de entrada y salida que comunica el jugador con el sistema del juego.

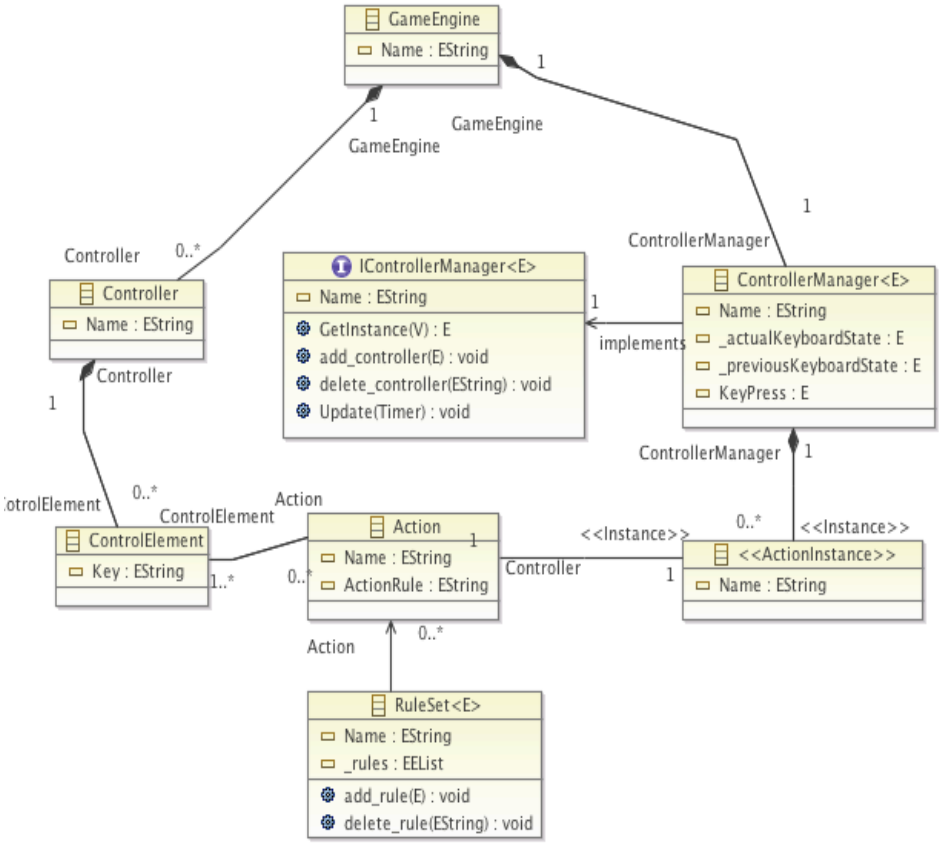


Figura 4.4.1.2.1. Perspectiva del manager de Control.

4.4.1.3 Perpectiva del Manager de Niveles

La clase administradora de niveles permite almacenar niveles y mundos dentro de una colección, esta clase es la que gestiona en tiempo de diseño la distribución y el

comportamiento del diseño de niveles. Para comprender su funcionamiento se detalla la descripción de cada una de las clases que forman esta perspectiva.

La primitiva *LevelManager<E,V>* es una clase contenedor que maneja las clases instanciadas *<<LevelInstance>>* , éstas son asociadas a las clases *Level* del modelo PSM que han sido transformadas previamente del meta-modelo PIM mediante las meta-clases *Level* y *World*. Nótese que la figura 4.4.1.3.1 tiene una clase que posee 2 atributos parametrizados para la especificación de las clases *Level* y *World* del modelo.

LevelManager<E,V> implementa la interfaz *ILevelManager<E,V>* que permite definir los métodos primarios que especifican lo qué debe de hacer la clase administradora de niveles pero sin su implementación, es en esta clase donde se definirán la implementación de los métodos. Se observa también como la clase *GameEngine* del modelo PSM esta compuesto por mundos, éstos a su vez se componen de niveles, esta es la parte del modelo la que tras sufrir la transformación ha tenido menos cambios y ha enriquecido de información que en el modelo PIM no se podía definir, se observa como las meta-clases del modelo PIM, *PrimaryLevel* y *FinalLevel* son transformados en atributos de la clase Level en el meta-modelo PSM , así pues, un nivel ahora esta compuesto por un clase denominada *MapLevel* que es la que el diseñador utiliza para definir el diseño del nivel a través de *Tiles*, la especificación de un nivel se hace a través de un pequeño documento de texto que el motor de tiles transforma en un vista del *Layout* asociada a ese nivel.

Es en la clase *GameEntity* en la que se define el atributo *Unicode* que va asociado a ese *MapLevel* y que obtiene la representación visual a través de la clase *MapEntity*. Finalmente la clase *GameEntity* esta compuesta por atributos del juego que se asocian a las instancias de las primitivas de la pantalla de la interfaz gráfica de usuario como se declaraba en la especificación del modelo PIM.

La clase *GamePlay* ahora esta compuesta por un conjunto de reglas que son capturadas desde el modelo independiente de la plataforma y asociadas a las acciones de los elementos del control de la meta-clase *Level* declarado, todas estas reglas del modelo PIM son transformaran en sentencias condicionales en el modelo PSM.

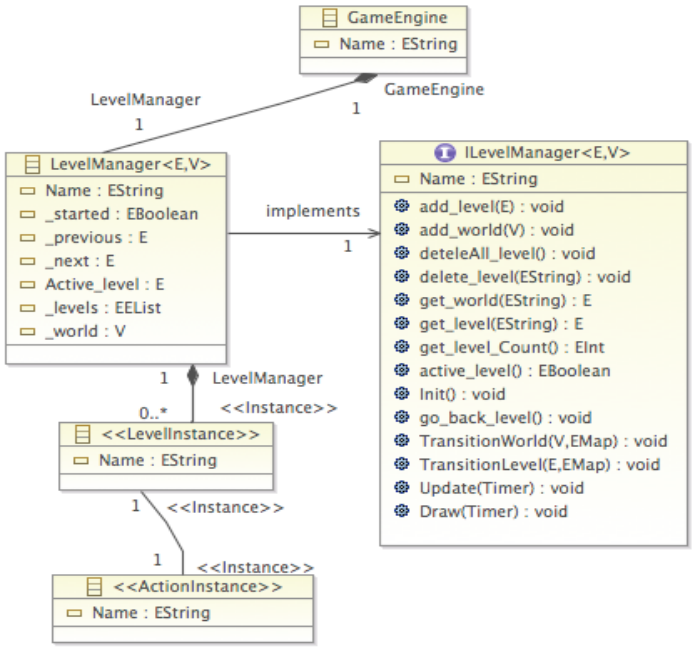


Figura 4.4.1.3.1. Perspectiva del manager del diseño de niveles.

4.5 Depuración y Validación

La depuración de programas es un proceso que sirve para identificar y corregir fallos de programación. A menudo, el error no es evidente y se requiere un tiempo considerable para localizarlo. Muchas veces se requiere incluir en el código fuente instrucciones auxiliares que permitan el seguimiento de la ejecución del programa, presentando los valores de variables, direcciones de memoria y ralentizando la salida de datos.

Mientras se desarrollaba el compilador de modelos, se ha construido y utilizado una técnica de verificación y validación de programas mediante *Echo debugging* [23] por la ventana del explorador de *Eclipse*. De esta manera se cerciora de que el código generado es correcto antes de ejecutarlo en la plaforma destino. El proceso de control asegura que el software cumple con su especificación y satisface las necesidades del usuario.

4.6 *Shell Script*

La creación de un pequeño *Shell script* para su uso en el Terminal de los sistemas operativos *Linux* y *Mac* (Apéndice VI) sirve al diseñador poder trasladar los archivos generados automáticamente por el compilador de modelos, el origen de éstos permanecen en carpetas ocultas y de esta menera se facilita la tarea (véase la imagen 4.4.3.).

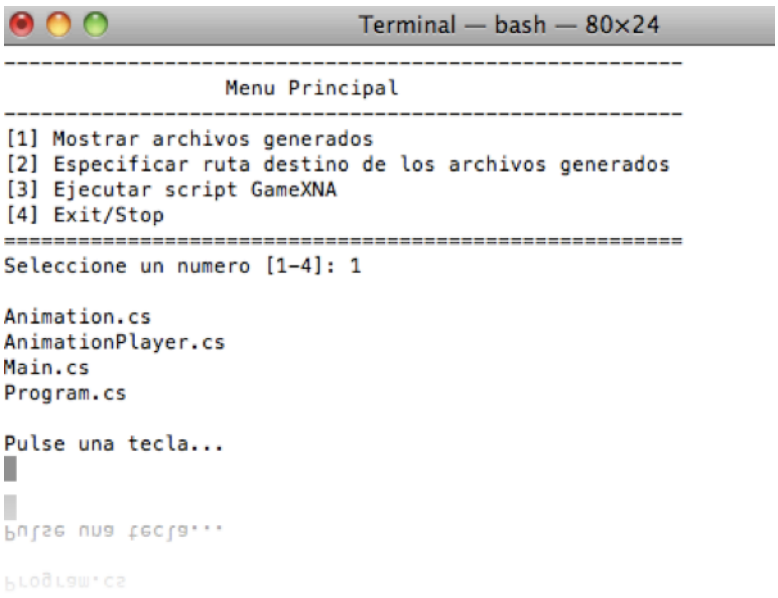


Figura 4.4.3.. Visualización en el terminal de la ejecución del *Shell script*.

5 Implementación de un prototipo

Con el propósito de demostrar las ventajas prácticas que asegura el desarrollo de juegos dirigido por modelos, se ha construido un prototipo de implementación en el entorno de modelado *Eclipse Modeling Framework*. Se han utilizando principalmente las herramientas de generación automática de código que proporciona *Eclipse EMF*. Las transformaciones de modelos se han realizado mediante *MediniQVT* y *MOFscript*, también integradas en el entorno de desarrollo *Eclipse*. Como resultado, la generación de juegos se apoya en la plataforma *Microsoft XNA Game Studio.*

Por simplicidad, en este apartado se han definido ciertas perspectivas de la especificación del multi-modelos en las que se aprecian algunos de los aspectos en el desarrollo del juego. Abordar la totalidad del desarrollo sería demasiado extenso y solamente se pretende enseñar la utilización de la metodología MDA y validar el multi-modelo propuesto.

5.1 Caso de estudio: *Space Invaders Classic*

En este apartado exponemos el caso de estudio de un ejemplo de juego, para ello se ha implementado un Space Invader Classic que utiliza la metodología MDA que se ha definido como propuesta. Permite validar el multi-modelo propuesto y ahondar en los detalles de la utilización del motor de *tiles* 2D, también permite la especificación del desarrollo del juego en cada una de las fases de su creación.

Space Invaders Classic es un juego arcade de marcianos en 2D para un solo jugador. El *player* controla el movimiento de la nave que puede moverse en dos direcciones horizontales (izquierda - deracha) y un botón de disparo. La finalidad del juego es ir destruyendo los marcianos invasores que poco a poco se acercan a la tierra y a medida de que el jugador va eliminandolos, la velocidad de éstos va incrementándose. Si los enemigos llegan a la posición de la nave que controla el *player* o es alcanzada por uno de los disparos de los marcianos invasores, la partida finaliza. El jugador posee de 4 escudos de protección que se van degradando según colisionan los disparos de los enemigos y de la nave terrestre.

5.1.1 Contenido de Artes

Las artes de los videojuegos son un conjunto *asserts* que se guardan en repositorios, constituyen la estructura visual del juego que especifica las diferentes partes que contiene el videojuego cuando se compila. El proceso se inicia con un activo de imagen en su forma original como archivo, y sigue hasta su transformación en datos, los cuales se pueden recuperar y ser utilizados dentro de un juego en la plataforma destino *XNA Game Studio*, a través de la biblioteca de clases de *XNA Framework*.

El contenido de los videojuegos son todas las partes del juego que no ejecutan como código administrado. Incluye todos los activos de imagen, como texturas, sprites, mallas, efectos y fuentes, así como los activos de sonido, como la música o los efectos breves de sonido. Asimismo, puede incluir activos de datos, como tablas de niveles o atributos de caracteres.

El principal motivo del uso del contenido de Artes es ayudar a que el juego se ejecute rápidamente. Sin la canalización de contenido, el juego tendría que compilarse con los activos de imagen en su formato original de archivo. Cuando el juego necesita cargar sus imágenes para dibujarlas en pantalla, tendría que determinar el formato de esas imágenes y convertirlo a un formato que pueda utilizar más directamente. Eso habría que realizarlo en tiempo de ejecución, para cada activo, y el jugador tendría que esperar para poder divertirse.

La canalización de contenido de Artes pone solución a este problema; para ello, lleva a cabo esta tarea donde se invierte un poco más de tiempo en la compilación del juego. En el momento de la compilación, cada activo se importa desde su formato original de archivo y se procesa y convierte en un objeto de código administrado. Esos objetos, a continuación, se serializan en un archivo que se incluye en el ejecutable del juego.

En el momento de la ejecución, el juego puede leer datos serializados directamente desde el archivo y convertirlos en un objeto de código administrado para utilizarlos de inmediato [31].

5.1.1.1 *Sprites*

Los *sprites* son mapas de bits en 2D que se dibujan directamente en un destino de representación sin usar la canalización de transformaciones, iluminación o efectos. Se suelen usar para mostrar información como las barras de estado, el número de vidas o información de texto como puntuaciones entre otras.

Los *sprites* se colocan en la pantalla mediante coordenadas. El eje X representa el ancho de pantalla y el eje Y , el alto. El eje Y se mide desde la parte superior de la pantalla y aumenta a medida que se desplaza *hacia abajo* en la pantalla, mientras que el eje X se mide de izquierda a derecha.

Un *sprite* se basa en un objeto *Texture2D*, esto es un mapa de bits que se usa para dibujar toda la textura o una parte de ésta. Para dibujar parte de una textura, suele utilizarse un rectángulo auxiliar donde se espcifica los *texel* (o píxeles de textura) que se van a dibujar. Una textura de 32×32 tiene 1024 texel especificados como valores X e Y, del mismo modo que se especifican las coordenadas de pantalla.

Los principales *sprites* utilizados en el desarrollo del juego *Space Invader Classic* se muestran en la figura 5.1.1.1.1, notese que se ha intentando realizar un clon del juego *Space Invader* [32].

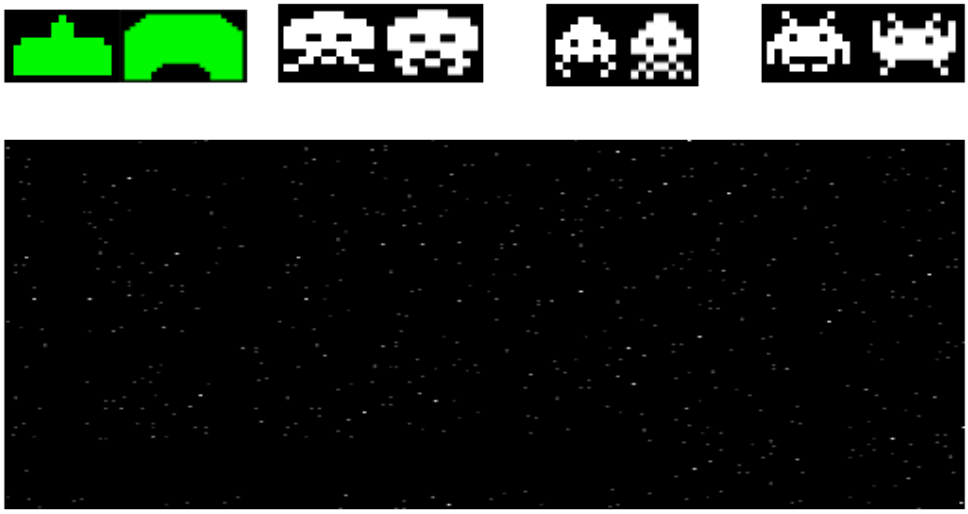


Figura 5.1.1.1.1. Artes del videojuego *Space Invaders Classis.*

La técnica de diseño utilizada mediante *sprites* en el desarrollo del juego por parte del diseñador ha sido la representación mediante viñetas individuales animadas [34]. De esta manera, se puede especificar todos los movimientos en animaciones diferente, agiliza el trabajo al diseñador y permite la representación de tantos *frames* cómo se quiera sin solapamiento de memoria. Antiguamente los diseñadores especificaban todos los *sprites* en una hoja llamada *Sprite Sheet* [12], permitía a los programadores ahorrar memoria a la hora de desarrollar el juego. Su acceso se hacía mediante una matriz especificando el alto y el ancho de la hoja entera donde se contenían todos los *sprites* del juego. Se accedía a ella mediante un rango de coordenadas X e Y que mediante un rectángulo contrendría el *sprite* deseado.

Actualmente, la tecnología emergente nos permite privarnos de esta técnica y poder utilizar otras con mucho mejor resultado. Notese que la representación de los disparos del juego no se ha representado mediante *sprites* sino mediante un rectángulo, siendo de color blanco el disparado de los alienijenas y verde el de la nave protectora.

5.1.2 Modelo Independiente de la Plataforma

Para facilitar la especificación del juego a un elevado nivel de abstracción a los diseñadores de juegos se ha implementado un editor en forma árbol mediante *Eclipse Modeling Tools*. Permite modelar videojuegos mediante los conceptos abstractos utilizados en el diseño de juegos. Se ha modelado un meta-modelo *ECORE* independiente de la plataforma tecnológica que recoge todos los conceptos del desarrollo de juegos.

El Apéndice I muestra el meta-modelo *ECORE* que define el modelo PIM para la especificación del juego a un alto nivel de abstracción que incluye las perspectivas del *GamePlay*, Control, Diseño de Niveles e Interfaz Gráfica entre otras.

La perspectiva *GraphicalUserInterface* define la interfaz gráfica de usuario, la cual indica cómo se representa a los jugadores el sistema interno del juego, véase la figura 5.1.2.1. Se especifica un primer nodo para identificar el comienzo del *Screen*, seguidamente los nodos necesarios para los gráficos de la interfaz gráfica y por último un nodo que identifica el final.

Por simplicidad, se asume que el juego especifica solamente una pantalla *intro* y la correspondiente a la interacción del juego. Las pantallas se especifican mediante una pantalla *ScreenNode*, cuyas dimensiones *Width* y *Height* se indican en píxeles. Ambas pantallas tienen una transición entre ellas con un evento que gestiona el paso o retorno de una pantalla a otra.

El *ScreenNode* que recibe el nombre de *intro* define el inicio del juego, en el se aprecia la pantalla principal del juego dónde se define un evento de control que permite hacer la transición entre pantallas. Las primitivas de *display* se definien mediante coordenadas X e Y junto con el tamaño del contenedor definido, en este caso, se han creado contenedores de texto, numéricos y un *background* de fondo negro.

La clase Juego define el nodo del *screen* que asocia la pantalla de los niveles del mundo que contiene el juego. Posee dos transiciones, una para poder volver a la pantalla principal del juego y poder así inciar una nueva partida. Una segunda transición para salir del juego en el momento que se precise y finalizar la partida. Ambas transiciones disponen de un evento para poder ejecutar el traslado a otra pantalla.

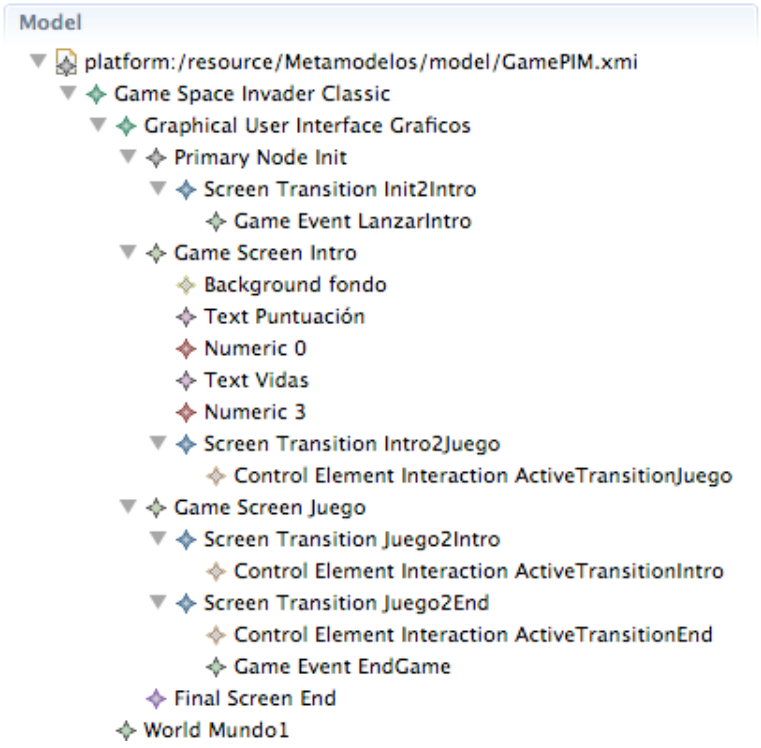


Figura 5.1.2.1. Especificación de los gráficos del juego *Space Invaders Classis.*

La perspectiva del Control define el mapeo de los dispositivos a las acciones que ejecutan los jugadores. Un controlador *Controller* contiene varios elementos de control *ControlElement*. Cada elemento de control se define por su tipo de información, contínua o discreta, y por su dimensionalidad, 1-dimensional o 2-dimensional. Como ejemplo típico de elemento de control 1-dimensional discreto considérese una tecla de un teclado. Como ejemplo típico de elemento de control 2-dimensional contínuo considérese un joystick.

Cada elemento del control puede asociarse a las acciones de un personaje que es controlado por el jugador que interactua con el juego.

La figura 5.1.2.2 ofrece la visión en forma árbol de los controles definidos en el modelo PIM del juego. Se define un mapeo de control que hace referencia al teclado, en el se describen tres elementos del control discretos que hacen referencia a las teclas del teclado previamente definidas. Cada *ControlElement* esta asociado a una acción que se asocia mediante la representación de una flecha.

Figura 5.1.2.2. Especificación del control del juego *Space Invaders Classis.*

La perspectiva del Diseño de Niveles define como gestionar la distribución del diseño del juego procedente de las especificaciones del diseñador en el modelo PIM y está compuesta por las primitivas *World*, *GenericLevel*, *GamePlay* y *TransitionLevel*.

La especificación del diseño de niveles consta de un simple mundo en el que se contiene 4 niveles jugables. Se definen las clases *PrimaryLevel* y *FinalLevel* para denotar el inicio y el final de los niveles del mundo . Para simplificar el desarrollo del juego, todos los niveles tienen el mismo layouts que consta de una <<*GameInstance*>> asociada a un <<*AtributeInstance*>> que hace referencia al *background* que se definió anteriormente en la perspectiva de las primitivas del *display* definida como *background* fondo

La figura 5.1.2.3. muestra el ejemplo del desarrollo donde se define un mundo con niveles y sus transiciones. En la creación de los mundos se permite especificar el alto y ancho que medirán los niveles, así como las coordenadas en las que serán colocadas en el *Game Screener* definido anteriormente en la perspectiva *ScreenNode* definida como Juego.

Todo nivel debe definir un *layout* y por simplicidad, todos los niveles contienen el mismo *layout* que se asocia a una <<GameInstance>>.

Las transciones permiten avanzar por los niveles del juego según las reglas definidas en el nivel. Un nivel puede avanzar y retrocer a más de un nivel del juego según las reglas definidas, de esta manera, si el jugador estuviera situado en el nivel3 y ganara la partida, avanzaría al siguiente nivel, en el caso de que el jugador perdiera, avanzaría a la pantalla principal representada como PrimaryLevel que automáticamente se redirigiría al nivel 1.

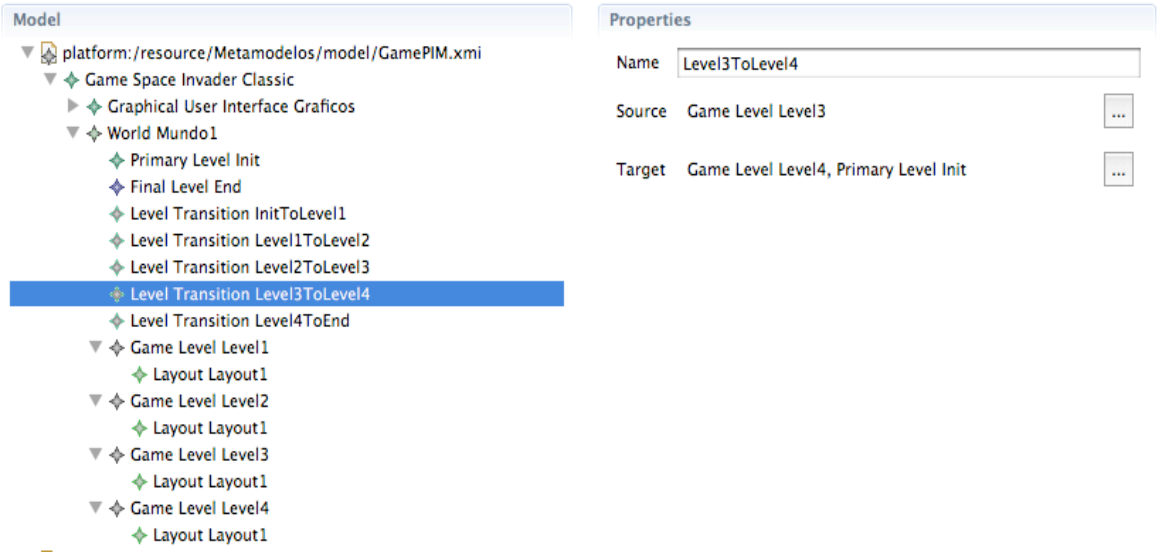


Figura 5.1.2.3. Especificación del Diseño de niveles del juego *Space Invaders Classis.*

El *GamePlay* se especifica en cada mundo creado, es aquí donde se especifican las entidades del juego que se asocian a las <<*GameInstance*>> que se han nombrado anteriormente. También se crean los conjuntos de reglas que se van asociar a cada nivel del juego.

Hay dos tipos de actores que pueden interactuar con un juego, jugadores individuales *Player* y equipos *Team*. Cada equipo está formado por diversos jugadores individuales. Los personajes jugadores son controlados por jugadores individuales. Los personajes no jugadores

son controlados internamente por el sistema de juego. La especificación de los algoritmos de inteligencia artificial que definen el comportamiento de los personajes no jugadores queda fuera del alcance de este trabajo.

Los personajes jugadores como los no jugadores pueden realizar acciones *Action* para interactuar con el sistema de juego. En el caso de los personajes jugadores, los jugadores individuales ejecutarán las acciones mediante un dispositivo controlador. De este modo, los jugadores controlan a sus personajes jugadores que indirectamente interactúan con el sistema de juego

Todas las entidades de juego contienen atributos y eventos para especificar su comportamiento mediante reglas. Cada regla *Rule* se especifica mediante una pre-condición y una post-condición. Cada una de estas condiciones *Condition* puede ser una condición simple o una expresión compleja.

Figura 5.1.2.4. Especificación de las Entidades y Reglas del juego *Space Invaders Classis.*

La figura 5.1.2.4 describe el *GamePlay* de manera resumida asociado al Mundo1 que el diseñador ha especificado en el desarrollo del juego. Se define un conjunto de reglas que permite establecer el comportamiento del juego según las acciones del personaje o de los no jugadores. Se detella la ReglaLevel1-1 que permite identificar la colisión de un disparo alienijena con la nave del personaje, en este caso, se lanza un evento que decrementa en una unidad el numero de vidas. La ReglaLevel1-2 permite saber cuando la nave se queda sin vidas y se tiene que proceder a iniciar el juego nuevamente. La ReglaLevel1-3 identifica la colisión del disparo de la nave con los alienijenas, cuando esto ocurra, se aumenta la velocidad del juego en media unidad de tiempo, apreciando una mayor velocidad en las naves alienijenas. Por último, la ReglaLevel1-4 permite identificar la colisión del disparo de la nave con el escudo protector. A continuación se definen las acciones que son mapeadas por los controles que hemos definido anteriormente.

Finalmente, el diseñador crea las entidades del juego tales como, la nave del personaje principal, el escudo protector, el alien y los correspondientes disparos. Una vez creado el *layout* que es asociado a cada nivel, se enlazan las entidades del juego junto con las clases <<*GameEntityInstance*>> para poder instanciarlas en el *layout* del nivel del juego

Una vez definido en el editor de árbol el meta-modelo utilizando *Eclipse EMF* se procede a la transformación de modelos conocida con las siglas M2M. Para ello se utiliza un lenguaje que nos permite transformar un modelo PIM en un modelo PSM mediante MediniQVT.

La figura 5.1.2.5 muestra el proceso donde el diseñador solo tiene que completar los campos que la ventana de MediniQVT le solicita. Especificamos el archivo de transformaciones, véase el Apendice IV. Seguidamente se especifican las carpetas donde tenemos el modelo PIM origen y el modelo destino PSM, seleccionamos la carpeta del archivo *Traces* donde queremos guardar las trazas a modo de *log* por si hubieran fallos para poder analizarlo en un futuro. Finalmente aplicamos los cambios y ejecutamos la transformación.

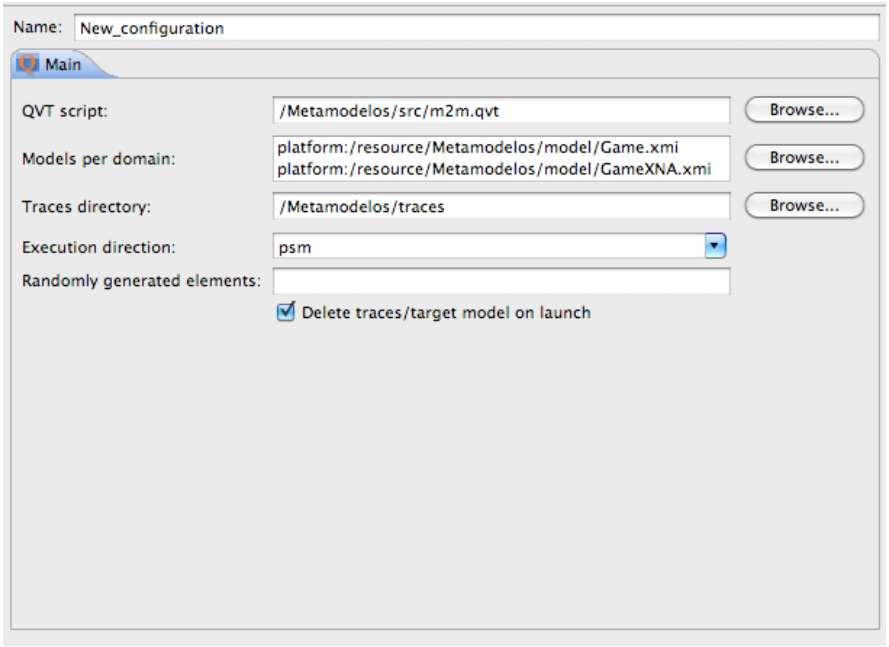


Figura 5.1.2.5. Transformación de modelos con MediniQVT.

5.1.3 Modelo Específico de la Plataforma

Con el objetivo de modelar los principales conceptos tecnológicos que se manejan al desarrollar juegos multiplataforma, se ha implementado un meta-modelo ECORE específico de la plataforma que permite presentar todos los conceptos modelados en el PIM por el diseñador. El middleware concreto compila el código para las diferentes partes que componen la plataforma específica. En la utilización de este trabajo se ha utilizado un motor de *tiles* 2D que podrá ser transformado a diferentes lenguajes de programación.

El Apéndice II muestra el meta-modelo *ECORE* que define el modelo PSM para la especificación del juego que incluye las perspectivas de los manejadores de Control, Diseño de Niveles e Interfaz Gráfica entre otras.

La perspectiva *GraphicalUserInterface* se ha transformado ahora en la perspectiva del *Screen Manager,* se observan considerables cambios que se muestran en la figura 5.1.3.1.

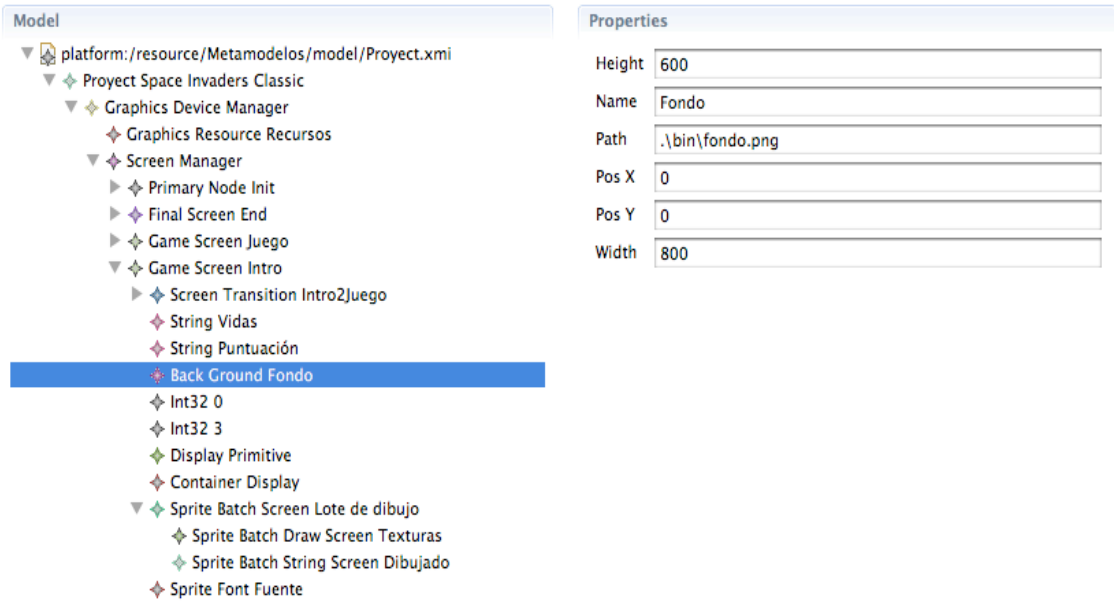


Figura 5.1.3.1. Especificación del *Graphics Device Manager* del juego *Space Invaders Classis.*

Los principales cambios vienen dados por la creación de una clase que gestiona los cambios en el dispositivo gráfico. En ella podemos predefinir los valores de inicio que serán heredados en todos los *Screens* que se hayan creado en el juego. La clase *Graphics Resource* permite almacenar recursos del juego.

Para facilitar la comprensión de la transformación entre modelos, mucho de los nombres del modelo PIM se han dejado igual en el modelo PSM, esto pasa con las clase *ScreenNode* y las que heredan de ella.

La clase *Intro* muestra considerables cambios que analizaremos en detalle. Se observa como la transformación de transición no ha sufrido cambios. Es en la representación de las primitivas del juego donde aparece un lote de dibujo que permite especificar y diferenciar las texturas del juego de la paleta de dibujo. La clase *SpriteBatchDrawScreen* permite presentar todo tipo de texturas tales como *Overlays* y *Backgorund*, es éste ultimo el que se asocia a esta clase y se detallan datos que el modelador del juego debe especificar tales como las dimensiones de la textura (800x600) y las coordenadas en el eje X e Y correspondientes a la

posición de la pantalla. La especificación de la ruta permite a la clase dotarla de una imagen visual en el pantalla. La clase *SpriteBatchStringScreen* representar el contenido numérico y textual mediante las primitivas del *display* declaradas anteriormente. Puesto que estamos especificando un contenido en un lote de dibujado, todas estas primitivas deben tener asociado una fuente de dibujo en la pantalla. Es la clase *SpriteFont* la que dota de aspecto visual a un alfabeto textual y numérico para la representación en la pantalla del juego.

La perspectiva de Control sufre ligeros cambios, se crea una clase que gestiona los controles del juego. La clase *Controllers* ahora se ha transformado en dos clases permitiendo al jugador disponer de dos entradas de interacion de juego como es un *pad* o el teclado.

Las clases *ControlElement* ahora se han transsformado en las clases *Buttons* correspondientes al *pad* y a las clases *ElementKey* definidas por el teclado. Se observa en la figura 5.1.3.2 como ahora el teclado dispone de las mismas interacciones pero específicas de la plataforma, cada ElementKey dispone de una propiedad donde especifica la tecla del teclado a la que hace referencia.

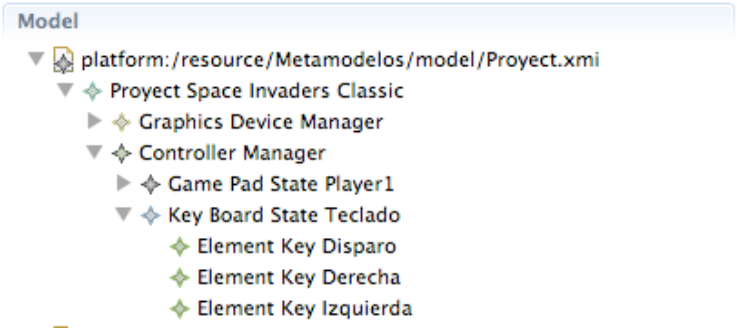


Figura 5.1.3.2. Especificación del *Controller Manager* del juego *Space Invaders Classis.*

La perspectiva de los niveles en el modelo PIM es transformada en una perspectiva manejadora llamada *LevelManager* que permite gestionar los niveles del mundo.

Ahora los niveles declarados tienen la posibilidad de especificar el mapeo de gráficos a través una clase llamada *MapLevel* que permite su representación visual a través de un documento de texto. Como se observa en la figura 5.1.3.3 se le atribuye las propiedades del tamaño de la pantalla (800x600), también la ruta del archivo de texto que mediante las transformación de modelo a texto el motor de *tiles* parseará en la capa gráfica de ese nivel

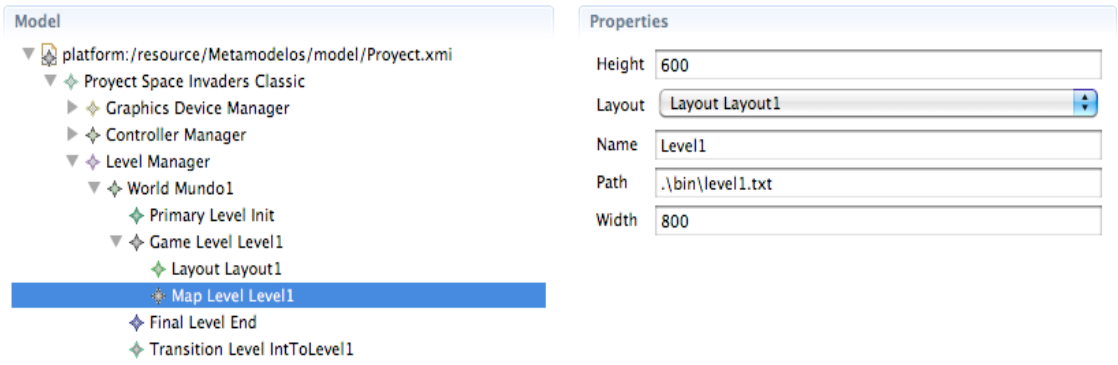


Figura 5.1.3.3. Especificación del *Controller Manager* del juego *Space Invaders Classis.*

correspondiente al *layout* declarado. Notese que por simplicidad, solamente se ha declarado un *layout* en la especificación del juego dónde se declara la estructura externa del juego. Se podría haber definido varios *layout* dotando a éstos últimos de atributos relacionados con las instancias de diferentes primitivas del *display* cómo podrían ser varios *background* para dotar de un mayor aspecto gráfico al juego.

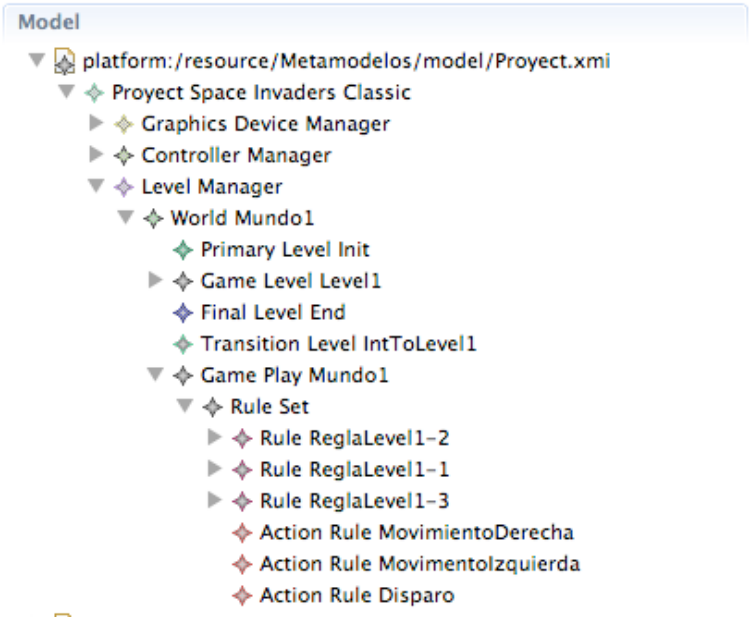


Figura 5.1.3.4. Especificación de las Reglas del juego *Space Invaders Classis.*

La perspectiva de las Reglas del juego no sufren cambio algunos, simplemente son los atributos que manejan estas reglas los que pasan a ser de tipo numérico a entero y de texto a un tipo definido *String*.

Como puede apreciarse en la figura 5.1.3.4 la estructura de las reglas del GamePlay se asimila a la del modeo PIM pudiendo ser de fácil comprensión la tranformación al modelo PSM.

La perspectiva de las Entidades ahora es transformada en *Tiles*. Se especifican atributos tales como la aceleración y la velocidad, también se establece un atributo a través de un valor *Unicode* que permite la representación de la *tile* en el *MapLevel* del nivel del juego. Es en la clase *MapTile* donde se especifica las dimensiones de ancho y lardo de la *tile*. A través del atributo *Path* se le atribuye la imagen visual que se representa en el *layout* del nivel como se puede mostrar en la figura 5.1.3.5.

Por último, con el objetivo de capturar el conocimiento asociado a la codificación de la especificación de videojuegos se ha implementado un compilador de modelos mediante el lenguaje de transformaciones de modelo a texto con *MOFscript*. El compilador lee un modelo PSM y genera automáticamente el código del juego y los archivos XML de descripción de las tipografías del juego para la plataforma *Microsoft XNA*.

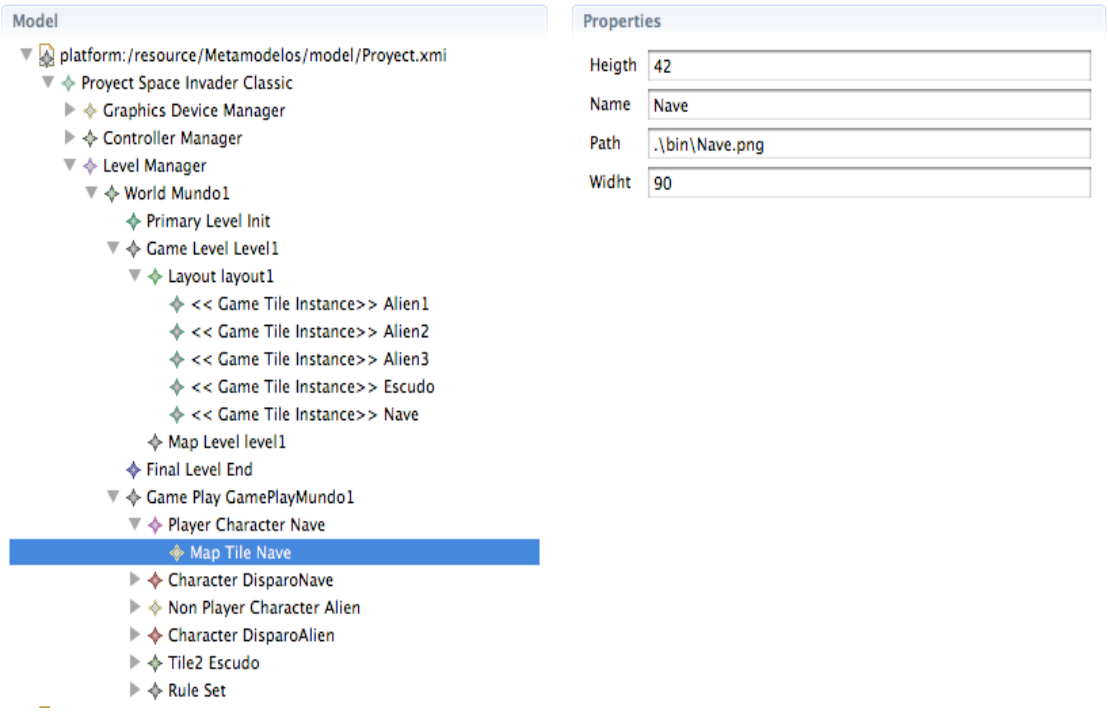


Figura 5.1.3.5. Especificación de las *Tiles* del juego *Space Invaders Classis.*

La figura 5.1.3.6 muestra la ventana de transformación de *MOFscript,* puesto que la naturaleza de este lenguaje no es algo cotidiano, la ventana de interfaz gráfica de usuario para efectuar la transformación carece de un aspecto visual agradable, debemos acceder al archivo de transformación de modelo a texto que hemos creado, véase el Apéndice V, hacer click con el botón derecho y acceder al ítem de *MOFscript*. Se observa como tenemos tres opciones, primero compilamos el archivo, dado que se especifica la ruta en la cabecera de la transformación del modelo PSM, el propio lenguaje mapea el modelo. Finalmente efectuamos la ejecución y realizamos la transformación.

Figura 5.1.3.6. Ejecución de la transformación mediante MOFscript.

5.1.4 Proyecto *Microsoft XNA*

Tras ejecutar MOFscript obtenemos los archivos.cs creados a través de las transformación de modelo a texto obteniendo los *asserts* necesarios para ejecutar el juego en *Microsoft XNA*.

Se debe utilizar el *Shell Script* definido en punto 4.4.2 que anteriormente se ha explicado para poder obtener estos archivos y trasladarlo a una carpeta visible y de fácil acceso. Eclipse define la ruta en una carpeta oculta, se observa como en el editor de texto aparecen unos

iconos con una exclamación amarilla haciendo referencia a que estos archivos solo son de lectura impidiendo así la manipulación de éstos mediante el editor.

Se crea un proyecto XNA y añadimos los archivos generados, así como el resto del contenido artístico del juego al que referencia el código. Para el ejemplo del videojuego Space Invader Classic se ha utilizado gráficos sencillos que se detallan en el apartado de Contenido de Artes.

La Figura 5.1.4.1 muestra el proceso del proyecto del juego generado con los assert añadidos para poder ser ejecutado en un ordenador PC. Al definir dos tipos de *Controllers* en la perspectiva de control, un primero para el teclado y otra segundo para el *pad*, gracias a la portabilidad de *Microsoft XNA* podemos generar el juego para la plataforma Xbox.

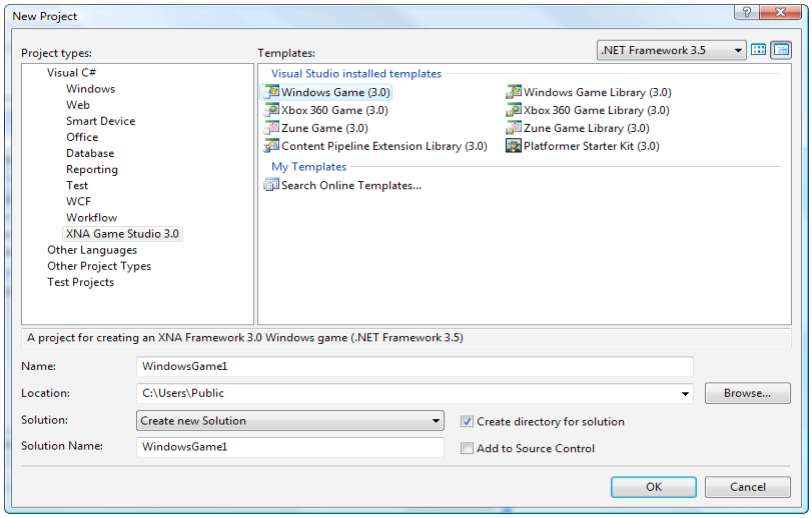


Figura 5.1.4.1. Proyecto *Microsoft XNA*.

Compilando el proyecto XNA resultante el juego se ejecutará correctamente, como muestra la Figura 5.1.4.2 en modo pantalla completa.

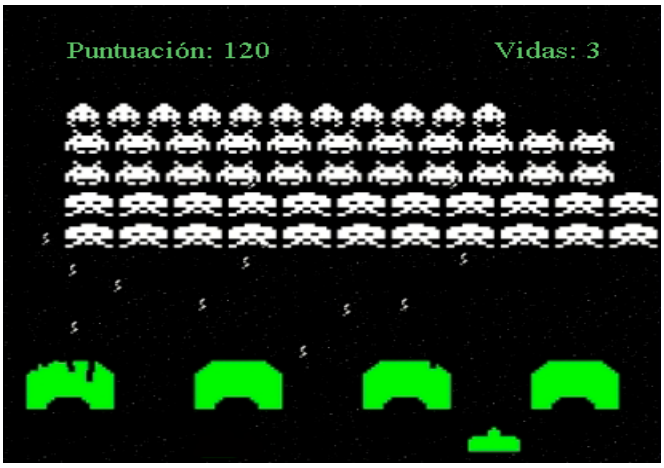


Figura 5.1.4.2. Juego *Space Invader Classic*

6 Conclusiones y trabajos futuros

El objetivo de este desarrollo dirigido por modelos es mostrar a través de un modelo PIM diferentes especificaciones de las perspectivas de los videojuegos tales como la interfaz gráfica de usuario, el control, el diseño de niveles, las entidades y reglas que forman el *gameplay* del juego. También se ofrece la propuesta de un modelo PSM genérico que permite definir la estructura y el comportamiento de un motor de tiles 2D que integra la jugabilidad del juego.

La aportación fundamental de este trabajo es un lenguaje de modelado conceptual que permite a los diseñadores de juegos especificar juegos a un elevado nivel de abstracción de las cambiantes plataformas tecnológicas. Para ello se utilizan varias vistas de interés que conforman un multi-modelo para la especificación de videojuegos mediante jugabilidad, interfaz gráfica de usuario, control y diseño de niveles entre otros.

El editor de niveles ha ayudado a especificar la estructura externa de la organización de los niveles de un juego así como las relaciones que tienen entre éstos. El uso de la técnica de animación representada mediante la primitiva *Layouts* ha servido para especificar también la estructura interna de un nivel dando ventajas al diseñador del juego con las diferentes vistas que contiene la composición final de la presentación de un nivel.

La propuesta para especificar un motor de *tiles* 2D mediante MDA ha favorecido el desarrollo de la propuesta de videojuegos en un escenario 2D reutilizando componentes y separando aspectos de un juego como es la especificación del *gameplay* en distintas perspectivas. Permite a nuestro motor de *tiles* 2D asentar todos estos conocimientos dando soporte a aquellos conceptos que no se alcanzaban en la especificación del modelo PIM.

La definición del motor de tiles 2D genérico en un modelo PSM nos favorece para poder enfocar el desarrollo de videojuegos en el mercado de la industria con el auge de las nuevas tecnologías. De esta forma se puede especificar el *gameplay* de un juego a un nivel muy alto de abstracción y orientarlo a un motor específico de la plataforma para dar soporte a la tecnología que se desee.

Para facilitar la tarea de modelado a los diseñadores de juegos, se han definido diversos lenguajes específicos del dominio para la definición visual de los conceptos del multi-modelo para la especificación de videojuegos. Adicionalmente, se ha implementado un prototipo de herramienta de desarrollo de juegos dirigido por modelos en el entorno Eclipse. Para ello se han utilizado transformaciones automáticas QVT desde los modelos independientes de la plataforma tecnológica a los modelos específicos de la plataforma tecnológica *Microsoft XNA*, así como transformaciones MOFscript que generan automáticamente el código ejecutable.

Como resultado al trabajo realizado se ha publicado un artículo en una conferencia de primer nivel en la investigación y desarrollo de meta-modelos. Diseño de Niveles y uso de Motores en el Desarrollo de Videojuegos dirigido por Modelos ofrece un resumen de las diferentes vistas definidas en el meta-modelo PIM y aporta una vista al diseño de niveles, por otro lado se ha apoyado el *gameplay* con un PSM definiendo el meta-modelo de un motor de *tiles* 2D, publicada en la conferencia JISDB 2012. Por otro lado se esta preparando otro artículo donde se muestran las técnicas de transformación de modelo a modelo y de modelo a texto que se explica en este trabajo.

Como trabajos futuros, se incluye la definición de otras perspectivas del juego como la historia o la inteligencia artificial que permitirán especificar juegos más complejos, la posible implementación del motor de tiles 2D en un lenguaje de desarrollo como Java para abordar prácticamente el mercado de dispositivos móviles de última generación y *Tablets*, también

utilizar el conocimiento existente en el uso de líneas de productos software para poder obtener una reutilización de software más estructurada y eficaz.

Finalmente, se ha de continuar mejorando el prototipo de herramienta para el desarrollo de juegos dirigido por modelos. Actualmente, la herramienta carece de editores gráficos para el modelado de juegos, lo que fuerza a los diseñadores a trabajar con el editor en árbol de Eclipse EMF, lo cual resulta costoso y poco intuitivo.

Resulta, por tanto, fundamental añadir editores gráficos que faciliten la labor de modelado visual a los diseñadores de juegos. Por otro lado, los meta-modelos PIM y PSM que soporta actualmente la herramienta únicamente cubre un subconjunto de la funcionalidad completa del multi-modelo para especificación de juegos y del middleware Microsoft XNA, respectivamente. Como trabajo futuro que ampliar ambos meta-modelos para ofrecer una mayor expresividad a los diseñadores de juegos, aumentando la funcionalidad del prototipo de herramienta de desarrollo de juegos dirigido por modelos.

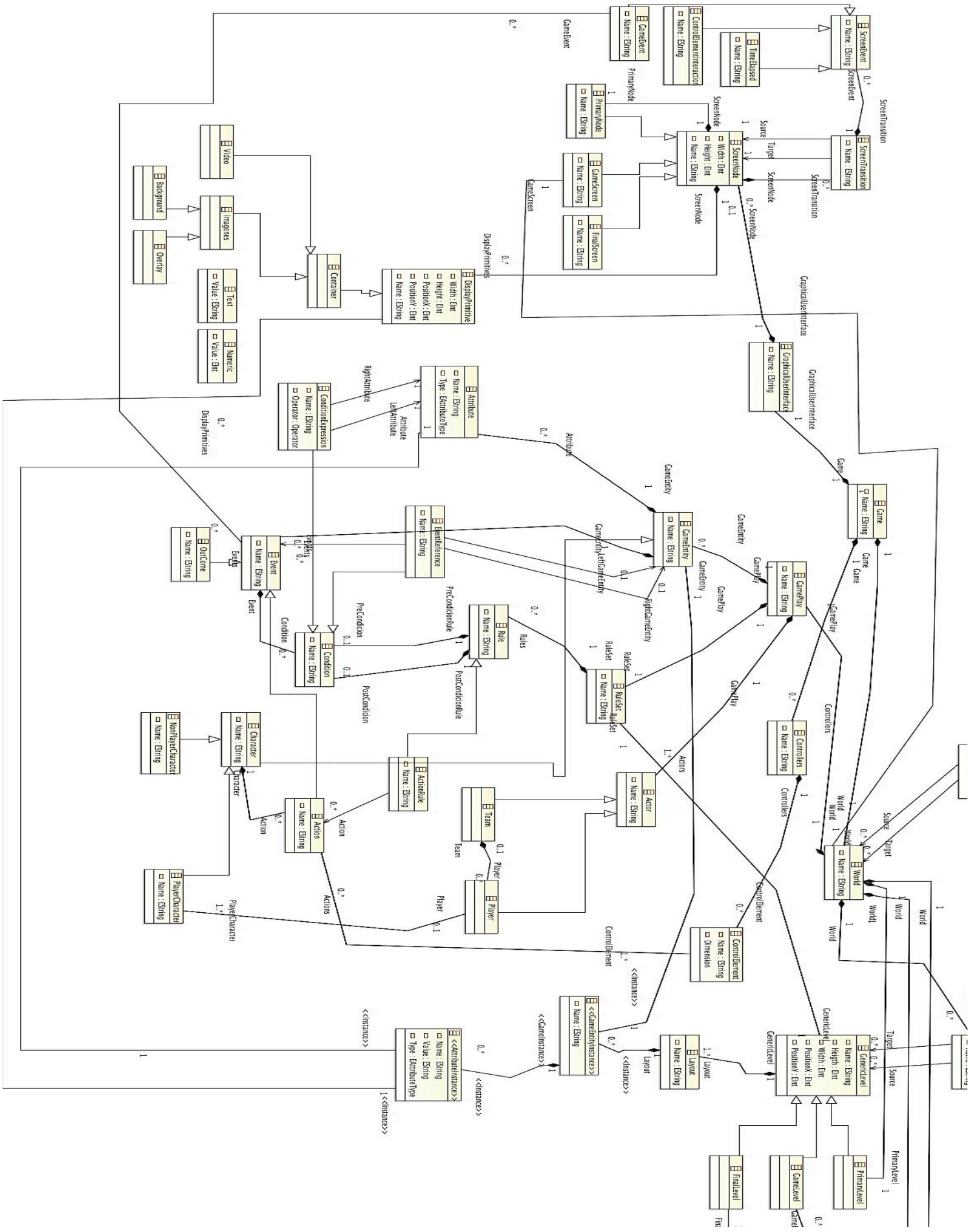
Agradecimientos. Queremos agradecer la asistencia del Ministerio de Ciencia y Tecnología ya que la investigación se ha financiado como parte del proyecto MULTIPLE, con referencia TIN2009-13838.

7 Referencias

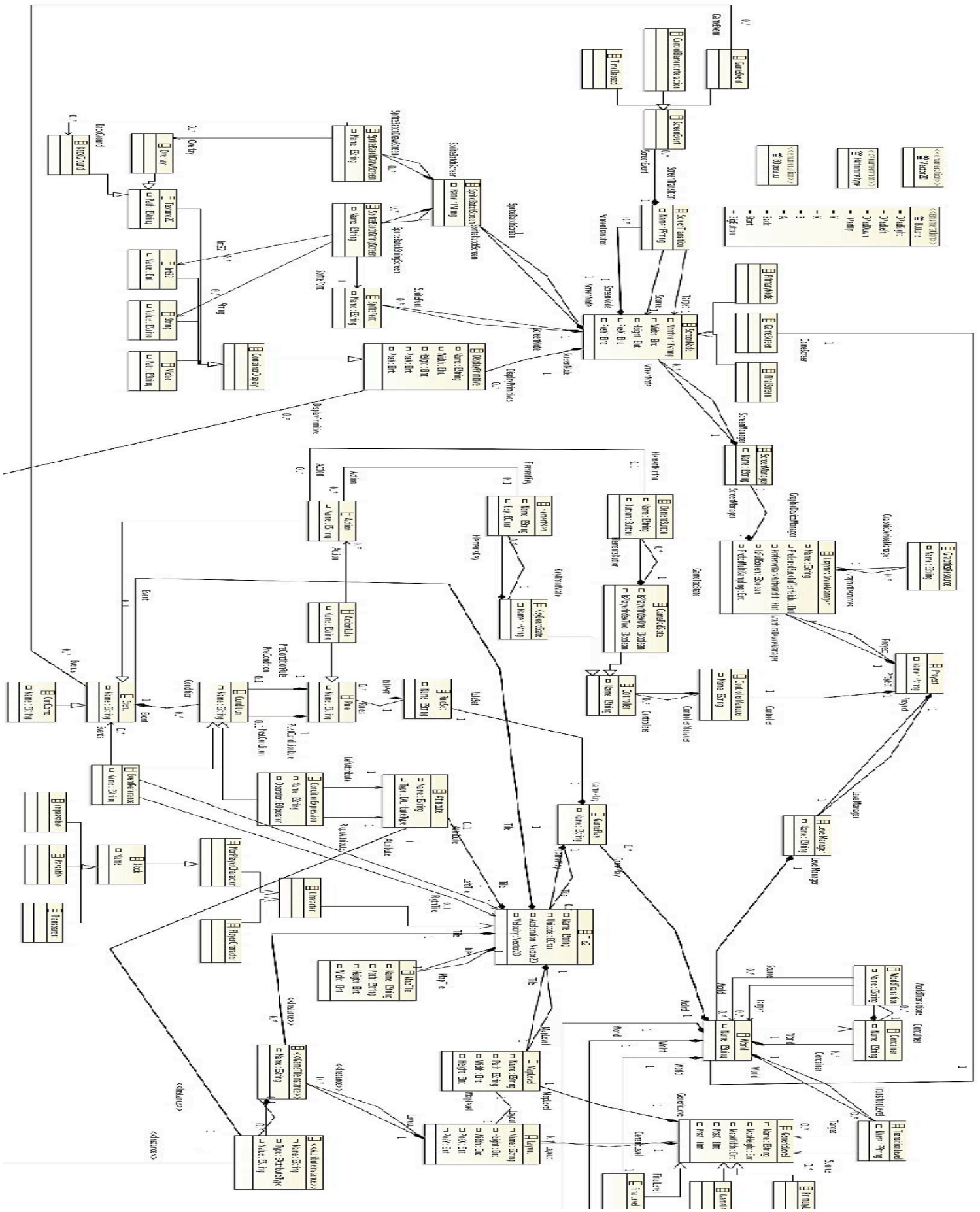
1. Andre W.B. Furtado, Andre L.M. Santos, and Geber L. Ramalho, Federal University of Pernambuco and Eduardo Santana de Almeida, Federal University of Bahia: Improving Digital Games Development with Software Product Lines (2011).
2. Emanuel Montero y José A. Carsí: MDA y Desarrollo de videojuegos, JISDB 2011.
3. Sanchez Crespo, D. : Diseño de niveles, pp, 65-74 .- Hacia la excelencia.
4. Rod tejada . : http://rodtejada.wordpress.com/author/rodtejada/page/2/ - Layout
5. Crawford, C.: On Game Design, pp 76-78, New Riders Publishing (2003).
6. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison Wesley (2003).
7. Gregory, J.:Game Engine Architecture, A K Peters (July 10, 2009).
8. J. Dobbe, "A Domain-Specific Language for Computer Games," MSc dissertation, Dept. of Software Technology, Delft Univ. of Technology, 2007.
9. Rodrigo Monteiro, The guide to implementing 2d platformers, Type #1: Tile-based (Smooth) London-2012.
10. Miroslav M.**,** Milos M.**,** Dušan S. Modelling Knowledge and Game Based Learning: Model Driven Approach, Journal of Universal Computer Science, vol. 17, no. 9 (2011), 1241-1260submitted: 8/1/11, accepted: 28/4/11, appeared: 1/5/11 © J.UCS
11. Montero Reyno E: "Desarrollo de Juegos Dirigido por Modelos: Multi-Modelo para la Especificación de la Jugabilidad, Interfaz Gráfica de Usuario y Control de Videojuegos". Trabajo de Fin de Master, Departamento de Ingeniería de Software y Sistemas de Información. UPV 2008.
12. Loew, Andreas. "SpriteSheets - Essential facts every game developer should know". codeandweb.com. Retrieved 2012-06-21.
13. Rodrigo Monteiro, The guide to implementing 2d platformers, Type #3: Tile-Vectorial. London - 2012.
14. James R. Williams, Simon Poulding, Louis M. Rose, Richard F. Paige ,Fiona A. C. Polack . Identifying desirable game character behaviours through the application of evolutionary algorithms to model-driven engineering metamodels, SSBSE'11 Proceedings of the Third international conference on Search based software engineering Pages 112-126.
15. Sid Meier. : GDC 2012: Sid Meier on how to see games as sets of interesting decisions, More Interesting Decisions.
16. Salen, K. and Zimmerman, E., Rules of Play, The MIT Press, 2004.
17. Frasca, G., Simulation versus narrative: introduction to ludology, In The Videogame Theory Reader, pp. 221–236, Routledge, London, UK, 2003.
18. Emanuel Montero, José A. Carsí, *A Platform-Independent Model for Videogame Gameplay Specification*. Digital Game Research Association Conference (DiGRA '09)
19. Rodrigo Monteiro, The guide to implementing 2d platformers, Type #1: Tile-based (pure) London-2012.
20. E. Byrne. Game Level Design. Charles River Media Boston, 2005
21. Tonypa. Tile Based Games : http://www.tonypa.pri.ee/tbw/tut24.html - Slopes.
22. Dormans, J. (2011c). Level Design as Model Transformation: A Strategy for Automated Content Generation. In Proceedings of the Foundations of Digital Games Conference, Bordeaux France, June 2011.
23. Tonypa. Tile Based Games : http://www.tonypa.pri.ee/tbw/tut25.html - HexTiles.Tip F. (1995) A survey of program slicing techniques. Journal of Programming Languages.
24. Altunbay, D., Metin, M. G., Çetinkaya, M. E.: Model-driven Approach for Board Game Development. In: First Turkish Symposium of Model-Driven Software Development (TMODELS), Ankara, Turkey (2009).
25. F.E. Hernandez and R.R. Ortega, "Eberos GML2D: A Graphical Domain-Specific Language for Modeling 2D Video Games," Proc. 10th SPLASH Workshop on Domain-Specific Modeling, Aalto-Print, 2010;
26. Fullerton, T., Swain, C., Hoffman, S.: Game Design Workshop: Designing, Prototyping, and Playtesting Games, CMP Books (2004).
27. E. Folmer, "Component Based Game Development: A Solution to Escalating Costs and Expanding Deadlines?" Proc. 10th Int'l ACM SIGSOFT Symp. Component-Based Software Eng., Springer, 2007, pp. 66–73.

28. Brown, A., An introduction to Model Driven Architecture, Available at http://www.ibm.com/developerworks/rational/library/3100.html.
29. Bob Bates, Game Desing Second Edition, pp. 107-111. Thomson Course Technology (2008).
30. B. Meyer. Eiffel: the language. Prentice-Hall International, 1989
31. MSDN Library XNA GAME STUDIO - Adding Art, Music, and Other Game Asset , http://msdn.microsoft.com/en-us/library/dd282477.
32. Martin Amis, An addict's guide to battle tactics, big scores and the best machines. Invasion of the space invaders, Hutchinson, 1982.
33. Colin Atkinson, Thomas Kuhne: Model-Driven Development: A Metamodeling Foundation. IEEE Software 20(5): 36-41 (2003).
34. PJ Cabrera, Peter Bakhirev, Ian Marsh, Ben Smith, Eric Wing, Scott Penberthy. Beginning IPhone Games Development. pp, 280- 287. Editorial Apress. (2010).

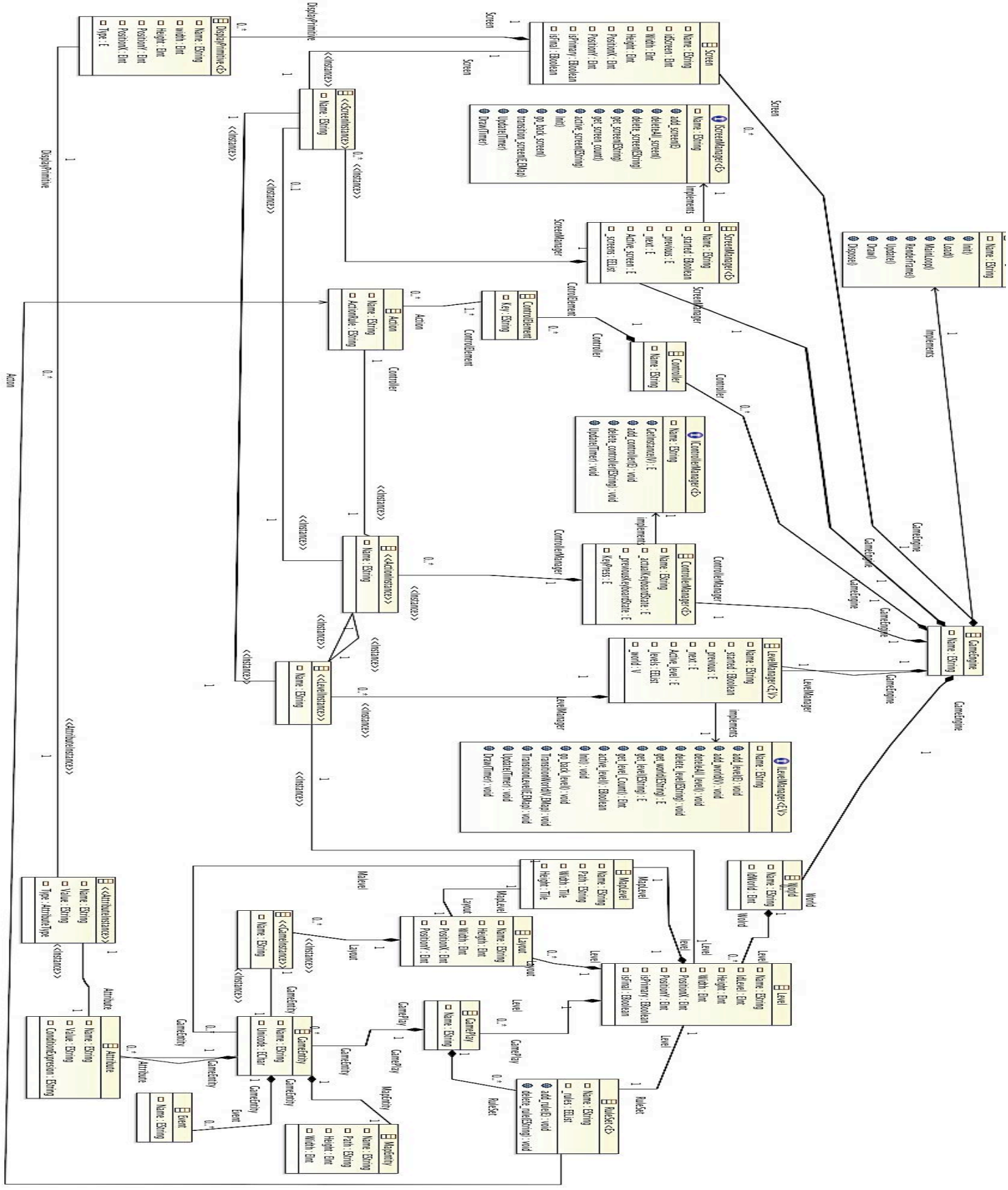
Apéndice I: Modelo PIM para la especificación del GamePlay.



Apéndice II: Modelo PSM para la especificación del Motor *Tiles.*



Apéndice III: Modelo instanciado del PSM genérico.



Apéndice IV: Transformación de modelos PIM to PSM

```
transformation pim2psm( pim : PIM, psm : PSM){

		top relation gameToGameXna{
			StrName : String;
			checkonly domain pim Game:PIM::Game {
		   Name=StrName,
		   /* Subconjunto dedicada a los Graficos del screen del juego */
		   GraphicalUserInterface=graphicalUser:PIM::GraphicalUserInterface{},
		   /*Subconjunto dedicado al gamePlay del juego */
		   GamePlay= gamePlay:PIM::GamePlay{}
		};

		/* Para un proyecto, defino mis clases del juego mas comunes*/
		/* y a éstas, les añado la informacion necesaria del PIM y del PSM */
			enforce domain psm GameXNA:PSM::GameXNA {
			Namespace=StrName,
			/* Me defino un proyecto */
			Project=:PSM::Project{
			Name='Project'
			},
/*A continuacion creo las clases predefinidas que deben de tener la arquitectura del juego de
plataformas

			Clase Main */
			Clases=claseMain:PSM::Class{
			Name='Main',
			Visibility= PSM::Visibilidad::public
			},
			/*Clase Program */
			Clases=claseProgram:PSM::Class{
			Name='Program',
			Visibility= PSM::Visibilidad::public
			},
			/*Clase Animation*/
			Clases=claseAnimation:PSM::Class{
			Name='Animation',
			Visibility= PSM::Visibilidad::public
			},
			/*Clase Animation Player */
			Clases=claseAnimationPlayer:PSM::Class{
			Name='AnimationPlayer',
			Visibility= PSM::Visibilidad::public

			}
		};

			/* Post Condicion */
			where
			{
			classToMain(graphicalUser,claseMain);
			}

		}

/*************************************************************************
*
```

```
/* Creacion de la clase Main */
                /* Esta clase contendra la semántica base de un video juego a partir del Pim y del
Psm*/
                relation classToMain {
                StrGraphics : String;
                IntWidth, IntHeigth : Integer;

               /* Tomo los datos del display grafico*/
                checkonly domain pim Grapichs:PIM::GraphicalUserInterface {
                Name=StrGraphics,
                Game=game:PIM::Game{},
                Screen=screen:PIM::Screen{
                Width=IntWidth,
                Height=IntHeigth
                }

                };

                /* Declaro los metodos necesarios para la estructura de un archivo.cs de XNA */
                /* Me declaro de base un SpriteBatch para que se pueda dibujar en la pantalla*/
                /* Un SpriteFont para que pueda dibujar cualquier tipo de texto */
                /* Un GraphicsDevice para la configuracion de los attr del Screen*/
                /* Transformo cualquier dato contender del PIM a la especifiacion del PSM a traves
de                      DisplayPrimitiveToAttributes(screen,ClassMain);*/
                enforce domain psm ClassMain:PSM::Game{
                Methods=metodoConstructor:PSM::Constructor{},
                Methods=metodoIni:PSM::Initialize{},
                Methods=metodoLoadContent:PSM::LoadContent{},
                Methods=metodoUpdate:PSM::Update{},
                Methods=metodoDraw:PSM::Draw{},
                Methods=metodoUnload:PSM::UnLoadContent{},
                Attributes=GraphicsDevive:PSM::GraphicsDeviceManager
                {
                Name=StrGraphics,
                PreferMultiSampling=false,
                IsFullScreen=true,
                PreferredBackBufferHeight=IntHeigth,
                PreferredBackBufferWidth=IntWidth,
                Type=PSM::Tipo::GraphicsDeviceManager,
                Visibilidad=PSM::Visibilidad::private
                },
                Attributes=spriteBatch:PSM::SpriteBatch{
                Name='spriteBatch',
                Type=PSM::Tipo::SpriteBatch,
                tag=' ',
                Visibilidad=PSM::Visibilidad::private,
                DrawString=DrawStr:PSM::DrawString{ Name='DrawString'},
                Draw_=Draw_:PSM::Draw_{Name='Draw'}
                },
                Attributes=spriteFont:PSM::SpriteFont{
                Name='font',
                Type=PSM::Tipo::SpriteFont,
                Path='********** select here your path **********',
                Visibilidad=PSM::Visibilidad::private
                }

                };

                /*Pre condicion */
```

```
		when
		{
		}
		/*Pos Condicion*/
		where
		{
		/*Método que transforma los datos del contenedor del display a attr especificos de mi
			plataforma*/
		DisplayPrimitiveToAttrNum(screen,ClassMain);
		DisplayPrimitiveToAttrString(screen,ClassMain);
		DisplayPrimitiveToAttrBackground(screen,ClassMain);
		DisplayPrimitiveToAttrOverlay(screen, ClassMain);
		DisplayPrimitiveToAttrVideo(screen,ClassMain);
		}


		}

/*****************************************************************************
*/
/*De la clase Screen voy a DisplayPrimite y luego convierto esta informacion en atributos de la
claseMain*/
	relation DisplayPrimitiveToAttrVideo{

		IntPosX,IntPosY : Integer;
		StrName: String;

		checkonly domain pim screen:PIM::Screen{

		GraphicalUserInterface=graphics:PIM::GraphicalUserInterface{},
		/*Capturo una primitiva de Background*/
		DisplayPrimitives=DisplayText:PIM::Video{
		PositionX=IntPosX,
		PositionY=IntPosY,
		Name=StrName
		}
		};

		enforce domain psm ClaseMain:PSM::Class{
		/*Fuerzo a Textura2D*/
		Attributes=attrString:PSM::Video{
		Name=StrName,
		Type=PSM::Tipo::Video,
		Visibilidad=PSM::Visibilidad::private,
		Path='************** select here your path **************'

		}
		};

		when{ }
		}


/*****************************************************************************
*/
/*De la clase Screen voy a DisplayPrimite y luego convierto esta informacion en atributos de la
claseMain*/
	relation DisplayPrimitiveToAttrOverlay{

		IntWidth, IntHeigth,IntPosX,IntPosY : Integer;
```

```
		StrName: String;

		checkonly domain pim screen:PIM::Screen{

		GraphicalUserInterface=graphics:PIM::GraphicalUserInterface{},
		/*Capturo una primitiva de Background*/
		DisplayPrimitives=DisplayText:PIM::Overlay{
		PositionX=IntPosX,
		PositionY=IntPosY,
		Name=StrName,
		Width=IntWidth,
		Height=IntHeigth
		}
		};

		enforce domain psm ClaseMain:PSM::Class{
		/*Fuerzo a Textura2D*/
		Attributes=attrString:PSM::Overlay{
		Name=StrName,
		Type=PSM::Tipo::Texture2D,
		Visibilidad=PSM::Visibilidad::private,
	X=IntPosX,
		Y=IntPosY,
		Width=IntWidth,
		Heigth=IntHeigth,
		Path='*************** select here your path **************'

		}
		};

		when{  }
		}
/*********************************************************************************
*/
/*De la clase Screen voy a DisplayPrimite y luego convierto esta informacion en atributos de la
claseMain*/
	relation DisplayPrimitiveToAttrBackground{

		IntWidth, IntHeigth,IntPosX,IntPosY : Integer;
		StrName: String;

		checkonly domain pim screen:PIM::Screen{

		GraphicalUserInterface=graphics:PIM::GraphicalUserInterface{},
		/*Capturo una primitiva de Background*/
		DisplayPrimitives=DisplayText:PIM::Background{
		Width=IntWidth,
		Height=IntHeigth,
		PositionX=IntPosX,
		PositionY=IntPosY,
		Name=StrName
		}
		};

		enforce domain psm ClaseMain:PSM::Class{
		/*Fuerzo a Textura2D*/
		Attributes=attrString:PSM::Background{
		Name=StrName,
		Type=PSM::Tipo::Texture2D,
		Visibilidad=PSM::Visibilidad::private,
```

```
                X=IntPosX,
                Y=IntPosY,
                Width=IntWidth,
                Heigth=IntHeigth,
                Path='********** select here your path **********'
                }
                };

                when{  }
                }

/*****************************************************************************
*/
/*De la clase Screen voy a DisplayPrimite y luego convierto esta informacion en atributos de la
claseMain*/
        relation DisplayPrimitiveToAttrString{

                IntPosX,IntPosY : Integer;
                StrName,StrValue: String;

                checkonly domain pim screen:PIM::Screen{

                GraphicalUserInterface=graphics:PIM::GraphicalUserInterface{},
                /*Capturo una primitiva de Texto*/
                DisplayPrimitives=DisplayText:PIM::Text{
                PositionX=IntPosX,
                PositionY=IntPosY,
                Name=StrName,
                Value=StrValue
                }
                };

                enforce domain psm ClaseMain:PSM::Class{
                /*Fuerzo a String*/
                Attributes=attrString:PSM::String{
                Name=StrName,
                Type=PSM::Tipo::String,
                Visibilidad=PSM::Visibilidad::private,
                X=IntPosX,
                Y=IntPosY,
                Value=StrValue
                }
                };

                when{  }
                }

/*****************************************************************************
*/
 /*De la clase Screen voy a DisplayPrimite y luego convierto esta informacion en atributos de la
claseMain*/
        relation DisplayPrimitiveToAttrNum{

                IntPosX,IntPosY,StrValue : Integer;
                StrName: String;

                checkonly domain pim screen:PIM::Screen{

                GraphicalUserInterface=graphics:PIM::GraphicalUserInterface{},
                /*Capturo una primitiva de Texto Numerica*/
```

```
                DisplayPrimitives=DisplayNumeric:PIM::Numeric{
                PositionX=IntPosX,
                PositionY=IntPosY,
                Name=StrName,
                Value=StrValue
                }
                };

                enforce domain psm ClaseMain:PSM::Class{
                /*Fuerzo a N_String*/
                Attributes=attrInt:PSM::int{
                Name=StrName,
                Type=PSM::Tipo::int,
                Visibilidad=PSM::Visibilidad::private,
                X=IntPosX,
                Y=IntPosY,
                Value=StrValue


                };

                when{}

                where{}

                }

}
```

Apéndice V: Compilador de modelos PSM-Texto.

```
//Creación de la Clase Main.cs de nuestro Juego
 psm.Class::ClassMain(d:psm.GameXNA)
 {

  ' // Codigo generado automaticamente

          using System;
          using System.Collections.Generic;
          using System.Linq;
          using Microsoft.Xna.Framework;
          using Microsoft.Xna.Framework.Audio;
          using Microsoft.Xna.Framework.Content;
          using Microsoft.Xna.Framework.GamerServices;
          using Microsoft.Xna.Framework.Graphics;
          using Microsoft.Xna.Framework.Input;
          using Microsoft.Xna.Framework.Media;

          namespace ' d.Namespace
          '{

           'self.Summary("Método Main del VideoJuego.")
            self.Salto(1)
            self.Espaciado(2) self.Visibility ' class ' d.Project.Name' : Microsoft.Xna.Framework.'self.oclGetType()'
              {
                '
          /*************************Atributos*****************************************/
                self.Summary("Atributos")
                self.Salto(1)
                self.Attributes->forEach(att:psm.Attribute) {
                stdout.println("\t\t\t Atributo "+att.Type+' '+att.Name)
                self.Espaciado(3)att.Visibilidad' 'att.Type' 'att.Name';'
                self.Salto(1)
                }
          /*************************Propiedades*****************************************/
                self.Summary("Propiedades Get/Set")
                self.Salto(1)
                self.Attributes->forEach(att:psm.Attribute){
                self.attributeGetterSettersCsharp(att)
                self.Salto(2)
                }
           /*****************************Métodos*****************************************/
                self.Methods->forEach(me:psm.Method){
                 stdout.println("\t\t\tMétodo "+me.oclGetType())

               /*************************Constructor*******************************/
                if(me.oclGetType().equals("Constructor")) {
                 self.Summary("Constructor de la clase")
                 self.Espaciado(3)'public ' d.Project.Name'() { ' self.Salto(2)
                 self.Attributes->forEach(att:psm.Attribute) {
                 if(att.Type.equals("GraphicsDeviceManager"))
                 {
                    self.Espaciado(3) 'this.'att.Name' = new ' att.Type'(this);' self.Salto(1)

                  self.Attributes->forEach(at:psm.GraphicsDeviceManager)
                  {
                     self.Espaciado(3) 'this.'att.Name'.IsFullScreen = ' at.IsFullScreen';' self.Salto(1)
                     self.Espaciado(3) 'this.'att.Name'.PreferredBackBufferHeight =
'at.PreferredBackBufferHeight';'                                    self.Salto(1)
                     self.Espaciado(3) 'this.'att.Name'.PreferredBackBufferWidth = 'at.PreferredBackBufferWidth';'
                                self.Salto(1)
                     self.Espaciado(3) 'this.'att.Name'.PreferMultiSampling = ' at.PreferMultiSampling';'
self.Salto(1)
                     self.Espaciado(3) 'this.'att.Name'.ApplyChanges();' self.Salto(1)
                  }
```

```
                }
                else
                  { }
                }
                self.Espaciado(3)'Content.RootDirectory = "Content";'
                self.Salto(2)
                self.Espaciado(3)'}'
                self.Salto(2)
                }

                /*************************Initialize*****************************/
               else if (me.oclGetType().equals("Initialize")) {
              self.Espaciado(3)'/// <summary>'self.Salto(1)
              self.Espaciado(3)'/// Permite que el juego realice la inicializacion que necesite para empezar a
ejecutarse. '          self.Salto(1)
              self.Espaciado(3)'/// Aqui es donde puede solicitar cualquier servicio que se requiera y cargar todo tipo
de                          contenido' self.Salto(1)
              self.Espaciado(3)'/// no relacionado con los gráficos. Si se llama a base.Initialize, todos los
componentes                                  se enumerarán'self.Salto(1)
             self.Espaciado(3)'/// e inicializarán.'self.Salto(1)
             self.Espaciado(3)'/// </summary>'self.Salto(1)
            self.Espaciado(3)'protected override void 'me.oclGetType()'() { '
            self.Salto(2)

                /********Int********/
                self.Attributes->forEach(att:psm.int){
                    self.Espaciado(3)' 'att.Name' = 'att.Value';' self.Salto(1)
                }
                /******String******/
                self.Attributes->forEach(att:psm.String){
                 self.Espaciado(3)' 'att.Name' = "'att.Value.toString()"';' self.Salto(1)
                }
                self.Salto(1)
                self.Espaciado(3)'base.Initialize();'
                self.Salto(1);
                self.Espaciado(3)'}'
                self.Salto(2)
                }

                /*************************LoadContent*****************************/
               else if (me.oclGetType().equals("LoadContent")) {
                self.Summary("LoadContent se llama una vez inicializa la clase y permite cargar todo el
contenido.")
                self.Espaciado(3)'protected override void 'me.oclGetType()'() { '
                self.Salto(2)
                /*****SpriteBatch*****/
                self.Attributes->forEach(att:psm.SpriteBatch){
                self.Espaciado(3)att.Name' = new SpriteBatch(GraphicsDevice);'self.Salto(1)
                }
                /****Texture2D*****/
                                self.Attributes->forEach(att:psm.Texture2D){
                                self.Espaciado(3)att.Name' = Content.Load<'att.Type'>("'att.Path'");'self.Salto(1)
                                }
                                /****Video*****/
                                self.Attributes->forEach(att:psm.Video){
                                self.Espaciado(3)att.Name' = Content.Load<'att.Type'>("'att.Path'");'self.Salto(1)
                                }
                                /****SpriteFont*****/
                                self.Attributes->forEach(att:psm.SpriteFont){
                                self.Espaciado(3)att.Name' = Content.Load<'att.Type'>("'att.Path'");'self.Salto(1)
                                }
                self.Salto(2)
                self.Espaciado(3)'}'
                self.Salto(2)
                }

        /*************************Update*****************************/
```

```
                    else if (me.oclGetType().equals("Update")) {
                    self.Summary("Permite ejecutar la logica del juego ")
                    self.Espaciado(3)'protected override void 'me.oclGetType()'(GameTime gametime) { '
                    self.Salto(2)
                    self.Espaciado(3)'base.Update(gametime);'
                    self.Salto(1);
                    self.Espaciado(3)'}'
                    self.Salto(2)
                   }

                  /*************************Draw*******************************/
                    else if (me.oclGetType().equals("Draw")) {
                    self.Summary("Maneja los graficos del juego")
                    self.Espaciado(3)'protected override void 'me.oclGetType()'(GameTime gametime) { '
                    self.Salto(2)

                    self.Espaciado(3)'GraphicsDevice.Clear(Color.CornflowerBlue);'
                    self.Attributes->forEach(at:psm.SpriteBatch){

                    /****SpriteBatch****/
                        if(at.Type.equals("SpriteBatch")){
                                    self.Remarks("Empezamos a pintar el/los objetos");
                                    self.Espaciado(3)at.Name'.Begin();'self.Salto(2);

                          /****SpriteBatch.Draw****/
                                                at.Draw_->forEach(D:psm.Draw_){
                                                  stdout.println("\t\t\t\t"+at.Name)
                                stdout.println("\t\t\t\t\t"+D.Name)

                                                /****SpriteBatch.Draw->Background****/
                                                  D.Background->forEach(back:psm.Background){
                                                    stdout.println("\t\t\t\t\t\t"+back.Name)
self.Espaciado(3)at.Name'.Draw('back.Name', new Vector2('back.X','back.Y'), Color.White);'self.Salto(1);
                                                  }
                                                            }

                                /****SpriteBatch.DrawString****/
                                                at.DrawString->forEach(Ds:psm.DrawString){
                                              stdout.println("\t\t\t\t"+at.Name)
                                              stdout.println("\t\t\t\t\t"+Ds.Name)
                                                        /****SpriteBatch.DrawString->Fuente******/
                                                        Ds.SpriteFont->forEach(Sf:psm.SpriteFont){
                                              stdout.println("\t\t\t\t\t\t "+Sf.Name+" "+Sf.Type)
                                                                /****SpriteBatch.DrawString->String******/
                                                  Ds.str->forEach(st:psm.String){
                                                    stdout.println("\t\t\t\t\t\t "+st.Name+" "+st.Type)
self.Espaciado(3)at.Name'.DrawString('Sf.Name','st.Name', new Vector2('st.X','st.Y'), Color.Black);'self.Salto(1);
                                                  }
                                                        }
                                                /****SpriteBatch.DrawString->Int******/
                                                 Ds.SpriteFont->forEach(Sf:psm.SpriteFont){
                                                  stdout.println("\t\t\t\t\t\t "+Sf.Name+" "+Sf.Type)
                                                /****SpriteBatch.DrawString->String******/
                                                    Ds.Int->forEach(st:psm.int){
                                                      stdout.println("\t\t\t\t\t\t "+st.Name+" "+st.Type)
self.Espaciado(3)at.Name'.DrawString('Sf.Name','st.Name'.ToString() , new
Vector2('st.X','st.Y'),Color.Black);'self.Salto(1);
                                                  }
                                                }
                                                  }//DrawString

                    /****SpriteBatch.Draw****/
                                                at.Draw_->forEach(D:psm.Draw_){
                                                  stdout.println("\t\t\t\t"+at.Name)
                                stdout.println("\t\t\t\t\t"+D.Name)

                                                /****SpriteBatch.Draw->Overlay****/
```

```
                                D.Overlay->forEach(over:psm.Overlay){
                                  stdout.println("\t\t\t\t\t\t"+over.Name)
                                  self.Espaciado(3)at.Name'.Draw('over.Name', new Vector2('over.X','over.Y'),
Color.White);'self.Salto(1);
                                }

                                        }
                    self.Salto(1);
                    self.Remarks("Finaliza el pintado");
                    self.Espaciado(3)at.Name'.End();'
                    }
                    }
                    self.Salto(1)
                    self.Espaciado(3)'base.Draw(gametime);'
                    self.Salto(1);
                    self.Espaciado(3)'}'
                    self.Salto(2)
                   }

                     else if (me.oclGetType().equals("UnLoadContent")) {
                    self.Summary("Es llamado cuando la clase de elimina o se ejecuta un Dispose sobre ella")
                    self.Espaciado(3)'protected override void UnloadContent() { '
                    self.Salto(1)
                    self.Espaciado(3)'this.Dispose();'self.Salto(1)
                    self.Espaciado(3)'}'
                    self.Salto(2)
                   }

                   }

                  self.Salto(2)

                self.Espaciado(3)'}' self.Salto(1)

    self.Espaciado(2)'}'self.Espaciado(2)'//'d.Namespace


    } //Class2Main
```

Apéndice VI: *Shell script* en *Mac.*

```
#
# Script que crea un menu simple y da la opcion de seleccionar una tarea.
#

#=================================================================
target_path()       # (c) Victor Bolinches Marin                  [void]
#=================================================================
{
 echo " "
 echo "Ruta destino..." ;
                    read path ;

                    if test -d $path
                              then
                                        echo "La ruta $path es correcta "
                    else
                              echo "La ruta $path no es un directorio"
                    fi
                    echo " "
                    echo "Pulse una tecla..."
}


#=================================================================
show_Fich3()      # (c) Victor Bolinches Marin                    [void]
#=================================================================
{
echo " "
  for i in /private/tmp/*.cs
                              do
                              if test -s $i
                                        then
                                        echo "$i"  | tr "/" " " >> TmpFich
                              else
                                echo "No hay archivos.cs"
                              fi
                              done


          #=====================================================
                              #   En caso de error, como no hay Try/Catch en shell
                              #   script, saco el error por la salida strErr y evito
                              #   posibles visualizaciones de comandos ejecutas por
                              #   el cliente erroneamnete

          #=====================================================

                              awk '/private/{print $3}' TmpFich 2> error.log
                              rm TmpFich 2>>error.log
                              rm error.log

}
```

```
#=================================================================
show_Fich2()      # (c) Victor Bolinches Marin                    [void]
#=================================================================
{
  for i in /private/tmp/*.cs
                                do
                                if test -s $i
                                        then echo "$i"
                                else
                                  echo "No hay archivos.cs"
                                fi
                                done
                        echo " "
                        echo "Pulse una tecla..."
}


awk '/private/{print $3}' fich

#=================================================================
show_Fich()       # (c) Victor Bolinches Marin                    [void]
#=================================================================
{
  for i in /private/tmp/*.cs
                                do
                                if test -s $i
                                        then
                                        str="$i"  | tr "/" " "
                                        set -- $str
                                        pie=$2
                                        shift
                                        cabeza=$@
                                        sudo echo "$cabeza"
                                else
                                  echo "No hay archivos.cs"
                                fi
                                done
                        echo " "
                        echo "Pulse una tecla..."
}

#=================================================================
Create_Leeme()  # (c) Victor Bolinches Marin                     [void]
#=================================================================
{
  echo "Resumen: Cree un nuevo proyecto XNA y sustituya los archivos.cs por los de la carpeta
actual">>$ruta/Leeme.txt
   echo """>>$ruta/Leeme.txt
          echo "Cree un proyecto XNA con el mismo nombre que corresponde al proyecto creado en la
instancia GAMEXNA del metamodelo">>$ruta/Leeme.txt
          echo "En el explorador de Visual Studio y con el proyecto abierto">>$ruta/Leeme.txt
          echo "-> boton derecho">>$ruta/Leeme.txt
          echo "-> Añadir">>$ruta/Leeme.txt
          echo "-> Elemento existente">>$ruta/Leeme.txt
          echo "Seleccionar los archivos.cs de esta carpeta y sobreescribir por los del proyecto
actual">>$ruta/Leeme.txt

}
```

```
#=================================================================
Ejecuta()             # (c) Victor Bolinches Marin                  [void]
#=================================================================
{
if test -d $path
            then
            mkdir $path/JuegoXNA
            ruta=$path/JuegoXNA
            cp /private/tmp/*.cs $ruta
            echo ""
            echo "Archivos :"
            show_Fich3
            echo ""
            echo "Copiados con exito"

            else
            echo "La ruta destino no es correcta"
            fi
                        Create_Leeme

            if test -z $path
            then
            echo " "
            echo "La ruta destino es la raiz debido a que no la ha especificado"
            fi

            echo " "
   echo "Pulse una tecla..."
}
path=' '

while :
 do
   clear
   echo "-----------------------------------------------------"
   echo "                  Menu Principal "
   echo "-----------------------------------------------------"
   echo "[1] Mostrar archivos generados"
   echo "[2] Especificar ruta destino de los archivos generados"
   echo "[3] Ejecutar script GameXNA"
   echo "[4] Exit/Stop"
   echo "======================================================="
   echo -n "Seleccione un numero [1-4]: "

   read opc
   case $opc in

     1) show_Fich3
     echo " "
                        echo "Pulse una tecla..."
                                    read ;
     2) target_path
                          read ;;
     3) Ejecuta
                                    read ;;
     4) exit 0 ;;
     *) echo "Opps!!! Por favor elija correctamente 1,2,3 o 4";
                         echo "Pulse una tecla..." ; read ;;
 esac
done
```