Pontificia Universidad Católica de Valparaíso
Facultad de Ingeniería
Escuela de Ingeniería Informática

DESARROLLO DE VIDEOJUEGO DE ESTRATEGIA EN TIEMPO REAL CON INTELIGENCIA ARTIFICIAL BASADA EN MODELO BELIEF-DESIRE-INTENTION

FELIPE ANDRÉS HERNÁNDEZ LAGOS
CHOON-HO YOON BUSTAMANTE

Profesor Guía: **Silvana Roncagliolo De La Horra**

Profesor Co-referente: **Rodolfo Villarroel Acevedo**

Carrera: **Ingeniería Civil en Informática**

Diciembre, 2012

Resumen

En este estudio se diseña e implementa un videojuego de estrategia en tiempo real en el que la inteligencia del jugador controlado por el computador se basa en un modelo Belief-Desire-Intention. Para esto se elabora una representación del problema utilizando este modelo. Se determinan qué parámetros son necesarios para ser incluidos en la representación, de modo que la lógica de resolución de problemas sea consistente. La implementación del modelo se integra como un módulo dentro del prototipo de videojuego. Además, se toma en consideración la carga adicional de procesamiento generada por el módulo, permitiendo una ejecución fluida y manteniendo un comportamiento inteligente del jugador controlado por el computador.

Palabras-claves: inteligencia artificial, modelo belief-desire-intention, videojuegos, estrategia en tiempo real.

Abstract

This research designs and implements a real-time strategy video game in which the computer-controlled player's intelligence is based on a Belief-Desire-Intention model. To achieve this, a representation of the problem is made using this model. The parameters to be included in the representation are determined so that the problem solving logic is consistent. The model implementation is integrated as a module into the prototype of the video game. Also considers the additional processing load generated by the module, allowing a fluent execution and maintaining an intelligent behavior of the computer-controlled player.

Keywords: artificial intelligence, belief-desire-intention model, video games, real-time strategy.

Índice

1 Introducción 1

2 Definición de Objetivos 2

2.1 Objetivo General 2

2.2 Objetivos Específicos 2

3 Plan de Trabajo 3

4 Marco Referencial 4

4.1 Videojuegos 4

4.1.1 Perspectiva Global 4

4.1.2 Perspectiva Particular 5

4.1.3 Género Estrategia 5

4.2 Inteligencia Artificial 6

4.3 Algoritmos Genéticos 6

4.3.1 Evolución en la Naturaleza 6

4.3.2 Evolución en los Videojuegos 7

4.4 Razonamiento Basado en Casos 8

4.5 Modelo Belief-Desire-Intention 10

5 Estado del Arte 12

5.1 Algoritmos Genéticos 12

5.2 Razonamiento Basado en Casos 13

5.3 Modelo Belief-Desire-Intention 15

6 Ambiente de Desarrollo 16

6.1 Sistema de Control de Versiones (VCS) 16

6.1.1 Sistemas Locales de Control de Versiones 16

6.1.2 Sistemas de Control de Versiones Centralizados 17

6.1.3 Sistemas de Control de Versiones Distribuidos 18

6.2 Modelado y Animación en 3D 20

6.2.1 Blender 20

6.2.2 Modelado 3D 20

6.2.3 Creación de Material 21

6.2.4 Creación de Textura 22

6.3 Motor de Videojuego 23

6.4 Entorno de Desarrollo Integrado (IDE) 26

6.4.1 Eclipse IDE 26

6.4.2 Kit de Desarrollo de Software jMonkey (jMonkey SDK) 26

7 Implementación del Videojuego 28

7.1 Datos Generales **28**
7.2 Descripción General **28**
7.3 Reglas Específicas del Videojuego **28**
7.4 Arquitectura de la Implementación **30**

8 Implementación de la Inteligencia Artificial **50**
8.1 Representación del Estado del Juego **50**
8.1.1 Representación Preliminar 50
8.1.2 Representación Formal 51
8.2 Modelo de la Arquitectura **53**
8.3 Elaboración de Plan y Estrategia **55**
8.4 Detalles de la Implementación **56**
8.4.1 Interfaz de Comunicación Preliminar 57
8.4.2 Interfaz de Comunicación Formal 57
8.4.3 Controladores 58
8.4.4 Implementación de Controladores Preliminar 58

9 Propiedades de Unidades **60**

10 Pruebas Modelo BDI (Belief-Desire-Intention) **64**
10.1 Prueba Nº1: *Acciones Iniciales* **65**
10.2 Prueba Nº2: *Acciones Finales* **65**
10.3 Prueba Nº3: *Acciones de Defensa Lateral* **66**
10.4 Prueba Nº4: *Acciones de Defensa Bilateral* **67**
10.5 Prueba Nº5: *Acciones de Reconexión* **67**
10.6 Prueba Nº6: *Manejo de Indicadores de Área* **68**
10.7 Prueba Nº7: *Tiempo en Mapa Pequeño* **69**
10.8 Prueba Nº8: *Tiempo en Mapa Mediano* **69**
10.9 Prueba Nº9: *Tiempo en Mapa Grande* **70**
10.10 Prueba Nº10: *Tiempo en Mapa Gigante* **70**

11 Conclusiones **71**

12 Referencias **73**

Lista de Figuras

Figura 4.1 Modelo de Razonamiento Basado en Casos 9
Figura 6.1 Esquema de Sistemas Locales de Control de Versiones 17
Figura 6.2 Esquema de Sistemas de Control de Versiones Centralizados......................18
Figura 6.3 Esquema de Sistemas de Control de Versiones Distribuidos 19
Figura 6.4 Vista de edición (izquierda) y Vista objetual (derecha) 21
Figura 6.5 Modelo con material de color amarillo..22
Figura 6.6 Mapeo UV del modelo de ejemplo. 23
Figura 6.7 Captura de pantalla de Doom. 24
Figura 6.8 Vista oblicua desde arriba hacia abajo..25
Figura 6.9 Modelo de trabajo de jMonkey. 27
Figura 7.1 Esquema de arquitectura de la implementación. 30
Figura 7.2 Modelo 3D de la Unidad Centro de Control..41
Figura 7.3 Modelo 3D de la Unidad Recolectora. 42
Figura 7.4 Modelo 3D de la Unidad de Ataque. 42
Figura 7.5 Modelo 3D de la Unidad de Defensa...43
Figura 7.6 Representación del escenario de batalla. 43
Figura 8.1 Representación Preliminar del Estado del Juego...51
Figura 8.2 Representación del Terreno e Indicador de Defensa 53
Figura 8.3 Modelo Arquitectura de la Inteligencia Artificial 54
Figura 8.4 Representación Gráfica del Árbol de Estrategia...56
Figura 9.1 Contenido del recurso *unitProperties.properties*. 63

Glosario de Términos

Conceptos de Modelado en Tres Dimensiones (McConnell, 2006)

Arista: segmento de línea que une dos vértices adyacentes. Para un polígono también puede ser denominado lado y para un poliedro puede ser denominado borde.

Cara: polígono que forma y delimita un poliedro. Está delimitado por aristas e incluye la superficie plana que se genera en el espacio intermedio.

Escalado: consiste en aumentar o reducir el tamaño de un modelo a partir de un factor de escala. Si el modelo escalado es N-dimensional, el factor de escala puede ser distinto en cada eje. Con ello se obtiene un máximo de N factores de escala distintos. Cuando todos los ejes son escalados utilizando un mismo factor, el modelo variará su tamaño manteniendo las proporciones de su forma original.

Malla Poligonal: malla no estructurada que reúne una colección de vértices, aristas, y caras definiendo una figura de objeto poliédrico en gráficas tridimensionales para computador.

Poliedro: figura geométrica sólida en tres dimensiones, contiene vértices, aristas y caras.

Polígono: figura plana conformada de segmentos de líneas rectas unidas entre sí para formar un circuito cerrado.

Reflectividad: para simular el efecto real de reflexión en los objetos, se les incluyen propiedades reflectoras que permiten replicar una luz de una dirección a otra, según el ángulo de proyección y con distintas grados de luminosidad.

Rotación: es el movimiento de cambio de orientación que sufre una forma en base a un eje fijo. Este eje es llamado eje de rotación.

Sombreado Difuso: es un efecto visual que representa las variaciones microscópicas de una superficie que está siendo impactada por un rayo de luz dado un ángulo.

Sombreado Especular: es un efecto visual que representa las reflexiones de un objeto brillante sobre una superficie. Estas reflexiones no son necesariamente provocadas por una luz.

Transformación: se dice que una forma ha sido transformada si y sólo si ha sido desplazada desde un punto de referencia a otro, ha cambiado su tamaño o fue rotada a una nueva dirección.

Translación: consiste en mover una forma completa desde un punto a otro, donde cada punto de dicha forma se mueve la misma distancia especificada. Este desplazamiento puede ser en cualquier coordenada de un espacio tridimensional.

Translucidez: es un efecto de los objetos que pueden permitir que la luz los traspase pero es demasiado difusa como para ser transparente.

Transparencia: es la capacidad o posibilidad de ver a través de un objeto. Este efecto se puede generar disminuyendo la opacidad del objeto, pero para generar un efecto real se necesitan otro tipo de técnicas que causen refracción de la luz que pasa a través del objeto.

Vértice: describe una posición en el espacio y representa una esquina o borde de una figura geométrica. No tiene dimensiones.

Conceptos del Videojuego

Costo de Construcción: cantidad de recursos necesaria para construir una unidad en el terreno de juego.

Energía: todas las estructuras requieren de energía para poder funcionar. La única fuente de energía es Centro de Control y la propaga a través de la red energética formada por todas las estructuras Recolectoras de Recursos, es decir, cada estructura Recolectora de Recursos construida extenderá la cobertura de la red energética teniendo como origen el Centro de Control.

Unidades: Entidades que pueden construirse sobre el terreno de juego y que poseen diversos atributos que las diferencian.

Producción de Recursos: cantidad de recursos que genera una unidad cada cierto intervalo de tiempo. Estos recursos serán utilizados para la construcción de otras unidades.

Puntos de Ataque: capacidad de las unidades de disminuir los puntos de vida de las unidades enemigas. No todas las unidades tienen la capacidad de atacar.

Puntos de Vida: cantidad de daño que puede resistir una unidad antes de ser destruida. Cada ataque reduce los puntos de vida de la unidad.

Recurso: unidad mínima de flujo económico del juego. Sirve para pagar el costo de construcción de unidades y es producido por las unidades Centro de Control y Recolector de Recursos.

Terreno de Juego: espacio físico donde se lleva a cabo el combate entre los contrincantes.

Zona Energética: las unidades sólo pueden ser construidas dentro del radio de energía que proporcionan las estructuras Centro de Control y Recolector de Recursos.

Lista de Abreviaturas o Siglas

2D : Dos dimensiones.
3D : Tres dimensiones.
AI : Artificial Intelligence o Inteligencia Artificial.
BDI : Belief-Desire-Intention o Convicción-Deseo-Intención.
CVCS : Centralized Version Control System o Sistema de Control de Versiones Centralizado.
DVCS : Distributed Version Control System o Sistema de Control de Versiones Distribuido.
GA : Genetic Algorithm o Algoritmo Genético.
HUD : Head-Up Display o Pantalla de Visualización Frontal.
JME3 : jMonkey Engine 3.
RTS : Real Time Strategy o Estrategia en Tiempo Real.
UV : Denota los ejes de una textura en dos dimensiones.
VCS : Version Control System o Sistema de Control de Versiones.

1 Introducción

En una búsqueda del inicio de los videojuegos, se puede observar que están presentes desde la creación de las primeras plataformas de tecnologías de la información. En la actualidad existen distintos géneros y clasificaciones según sus características principales de jugabilidad. Entre los géneros más populares se encuentran: Acción, Aventura, Deportes, Disparos, Estrategia, Peleas y Carreras.

El desarrollo de videojuegos ha sido impulsado y popularizado en gran manera por los avances tecnológicos en el área de la informática durante los últimos 40 años. Esto ha generado oportunidades para que personas, que no necesariamente dominan las disciplinas requeridas para la creación de videojuegos, puedan poner en marcha el desarrollo de un videojuego.

Esta ayuda proviene de un gran número de alternativas de aplicaciones disponibles para el desarrollo de videojuegos. Entre ellas se encuentran los motores de videojuego, que proveen funcionalidades como el dibujado de gráficos en 2D y 3D, motor de física, sonidos, scripting, animación, inteligencia artificial, trabajo en red, streaming, administración de memoria, entre otras. Además, existen entornos de desarrollo integrado que incorporan un conjunto de herramientas que facilitan la manipulación de los recursos del juego y su integración.

Para videojuegos de género de estrategia, específicamente estrategia en tiempo real, es necesario, y muy importante, generar un algoritmo que controle el comportamiento del o los enemigos creando la ilusión de inteligencia artificial. Existe una gran variedad de algoritmos capaces de cumplir este objetivo, dentro de los cuales destacan Máquinas de Estado Finito, Máquinas de Estado Difuso, Redes Neuronales, Algoritmos Genéticos, Razonamiento Basado en Casos, Scripting y Modelo Belief-Desire-Intention.

El presente documento comienza con la descripción de los objetivos del proyecto y el plan de trabajo. Dentro del marco referencial, se expone la base teórica sobre la cual se desarrollará el proyecto, explicando conceptos básicos, alternativas y avances en el desarrollo de inteligencia artificial en el área de videojuegos. En la siguiente sección se mencionan las herramientas utilizadas para la implementación. Posteriormente, se explica el concepto del videojuego, las reglas que lo rigen y su arquitectura de implementación. Finalmente, se exponen las conclusiones derivadas del estudio.

2 Definición de Objetivos

2.1 Objetivo General

Diseñar e implementar un videojuego de estrategia en tiempo real en el que la inteligencia del jugador controlado por el computador se base en un modelo Belief-Desire-Intention.

2.2 Objetivos Específicos

- ✓ Elaborar conceptos y las reglas del videojuego.
- ✓ Implementar un prototipo del videojuego.
- ✓ Elaborar modelo Belief-Desire-Intention que permita dirigir la inteligencia del jugador controlado por el computador.
- ✓ Implementar e integrar el modelo elaborado en el prototipo del videojuego.
- ✓ Realizar pruebas de sensibilidad sobre el prototipo del videojuego, con el objetivo de estudiar el comportamiento del modelo ante variaciones en sus parámetros.
- ✓ Documentar pruebas de sensibilidad del modelo.

3 Plan de Trabajo

El desarrollo del proyecto se dividió en cuatro etapas generales. Durante la primera etapa, denominada *Diseño de Prototipo de Videojuego*, se elaboraron de forma preliminar los conceptos y reglas. Luego, se construyeron modelos en tres dimensiones de las estructuras y escenario de juego. En esta etapa también se consideró la preparación del entorno de desarrollo, definición de la estructura del proyecto para la implementación del prototipo y su versionamiento en GitHub. La etapa concluyó con la entrega del informe de avance de proyecto 1.

En la segunda etapa, llamada *Implementación de Prototipo*, se implementó un videojuego para obtener un prototipo. Actividades de esta etapa incluyeron la implementación del escenario de juego, HUD, interacción del usuario con el videojuego e interacción entre componentes del videojuego. Además, se integraron los modelos en tres dimensiones preliminares diseñados en la primera etapa y se definieron los controles de juego del usuario. Los modelos preliminares se utilizaron al comienzo de la implementación del prototipo, pero más tarde se refinaron para obtener los definitivos. Dentro de esta etapa se comenzó con la elaboración e implementación de un prototipo del módulo BDI de inteligencia artificial. La etapa terminó con la entrega del informe final de proyecto 1 y un prototipo del videojuego que incluyó el módulo de inteligencia artificial. Cabe mencionar que existe un espacio de tiempo, correspondiente al *Receso Interperíodo*, donde se continuaron aplicando correcciones sobre el prototipo entregado.

La tercera etapa, especificada como *Elaboración de Modelo BDI*, corresponde al inicio de proyecto 2. Las actividades de esta etapa involucraron el refinamiento del modelo BDI y su implementación (a partir del modelo obtenido en la etapa anterior). Las actividades de esta etapa incluyeron la identificación de variables relevantes del problema y la elaboración de la representación del modelo BDI. La etapa concluyó con la entrega del informe de avance de proyecto 2.

La cuarta y última etapa, con nombre *Integración de Modelo BDI en Prototipo*, incluyó actividades de implementación del modelo BDI y su integración en el prototipo. Se concluyó con la entrega del informe final de proyecto 2 y la entrega definitiva del prototipo de videojuego.

4 Marco Referencial

4.1 Videojuegos

4.1.1 Perspectiva Global

El concepto de juego es definido por Chris Crawford, desarrollador de videojuegos desde 1980, como un sistema cerrado formal que subjetivamente representa un subconjunto de la realidad (Crawford, 1984).

Que el sistema sea cerrado se refiere a que un juego es completo en sí mismo y en estructura, el modelo del mundo creado es entendible por sí mismo y no debería ser necesario buscar elementos externos al juego en sí para entenderlo. Sus reglas cubren todas las contingencias que pudieran ocurrir dentro de él. Por formal se entiende que las reglas están formuladas explícitamente y no existen ambigüedades. Normalmente, si algún problema no puede ser solucionado con las reglas vigentes del juego, entonces, es indicio de que el juego está mal diseñado.

Como se trata de un sistema, posee varios módulos y elementos que interactúan entre sí, coordinándose por un mismo objetivo.

La subjetividad de la representación varía según las percepciones del usuario y su interpretación de la realidad. Es por esto que muchas veces se habla de realidad fantástica y depende directamente del usuario que esté jugando. La diferencia entre un simulador y un juego es que los simuladores intentan en todo momento replicar exactamente, o lo mayor posible, la realidad, mientras que un juego no posee este concepto como objetivo principal.

El subconjunto de realidad que el juego representa debe ser claramente acotado, ya que difícilmente podría representar toda la realidad. Este subconjunto debe mantener un tamaño adecuado que no agobie ni sobrepase la comprensión del jugador, por lo que una extensión mayor podría ser considerada inadecuada.

Para que un juego sea considerado como tal, debe contener tres conceptos fundamentales: interacción, conflicto y seguridad.

La interacción está relacionada con la forma en que cambia la realidad representada. En medios como pinturas y esculturas la realidad se representa de forma estática, para otros medios como la música o las películas esta realidad se muestra de forma dinámica. Para los juegos esta realidad, siendo dinámica, tiene un factor extra denominado interacción, y es la capacidad que tiene el usuario de intervenir en el acontecer de los eventos que ocurren en el juego. Esto se genera a través de la exploración, la intervención y la observación de las causas y efectos generados. Tiene la capacidad de transformar el desafío de un puzzle en una situación interpersonal o social, desde un reto técnico a un reto personal. Tiene la función de cambiar los desafíos de un aspecto pasivo a uno activo, de pensar una solución a un acertijo

que una vez se encuentra ya pierde su diversión, a interactuar con un ente vivo que se adecua a los cambios que genera el mismo interlocutor.

El conflicto nace intrínsecamente con la interacción, el jugador está constantemente en búsqueda de cumplir una meta, con obstáculos y dificultades que no facilitan el cumplimiento de este objetivo. Si existe un agente o entidad que activamente está intentando obstaculizar el avance del jugador para el cumplimiento de su meta, entonces se genera un conflicto entre el jugador y esa entidad.

La seguridad que debe facilitar un juego para prevenir que el jugador no reciba un daño que posiblemente dicho conflicto generaría en la realidad. Es por esto que los juegos deben ser los artífices de brindar la experiencia psicológica de interactuar y entrar en conflicto para cumplir una meta que no está sucediendo en la realidad, como una forma segura de experimentar dicha realidad modelada.

4.1.2 Perspectiva Particular

Los videojuegos son una clasificación especial de los juegos. Para que un videojuego sea considerado como tal debe tener dos características fundamentales: un dispositivo electrónico de video que despliegue la información de retroalimentación por una pantalla y un dispositivo electrónico para manejar la interacción humana denominado control.

El hardware/software o conjunto de dispositivos electrónicos que permiten que el videojuego funcione es denominado plataforma. Entre los casos más comunes de plataformas se encuentran: computadores personales, consolas de hogar, consolas portátiles y en la actualidad cualquier ambiente que permita su reproducción, como Facebook o Android.

Por lo general los dispositivos de control utilizados en la actualidad para la interacción con el usuario abarcan desde un teclado y mouse hasta los denominados controles de movimiento, con un fuerte énfasis en los controles tradicionales o controles de mando.

Por motivos prácticos los videojuegos son divididos en géneros dependiendo de varios factores, como sus metas o formas de jugabilidad. Esta división se genera debido a que los videojuegos son completamente dependientes de sus contenidos y a la vez evolucionan para inventar nuevos tipos de géneros o combinaciones de géneros antiguos. Entre los géneros más conocidos se encuentran los de disparo, carreras, peleas, estrategia y rol.

4.1.3 Género Estrategia

Los videojuegos de estrategia enfatizan la utilización del pensamiento y la planeación para el cumplimiento de la meta u objetivo. Los conceptos fundamentales en el modo de juego contienen principios de estrategias, principios de tácticas y desafíos lógicos (Rollings & Adams, 2003).

Existen dos clasificaciones generales para los videojuegos de estrategia: Estrategia en Tiempo Real y Estrategia por Turnos. Esta clasificación está hecha en base a la modalidad de juego que prevalezca en el transcurso normal del videojuego.

La Estrategia por Turnos se caracteriza por permitir a los jugadores un tiempo de análisis para la ejecución de la variedad de acciones que acontecerán, es decir, el juego se traduce en una sucesión continua de turnos.

En Estrategia en Tiempo Real el tiempo de juego es continuo y las acciones realizadas por los jugadores cambian el estado del juego constantemente, generalmente requiere un manejo de recursos y planificación de tácticas para las unidades en juego.

4.2 Inteligencia Artificial

La inteligencia artificial se define como la capacidad que tienen los artefactos de tener comportamiento inteligente. El comportamiento inteligente a su vez requiere percepción, razonamiento, aprendizaje, comunicación, y actuar en ambientes de alta complejidad. Como uno de los objetivos a muy largo plazo, la inteligencia artificial espera generar máquinas que puedan actuar inteligentemente como los seres humanos o incluso mejor (Nilsson, 1998).

Existen diversas posturas con respecto a que puede considerarse inteligencia artificial para videojuego. "Con respecto a la inteligencia artificial para videojuegos, estoy completamente de acuerdo con la opinión que si el jugador cree que el agente contra el cual está jugando es inteligente, entonces *es* inteligente" (Buckland, 2005), por otro lado existen desarrolladores para los cuales el concepto es mucho más estructurado y debe ser modelado, no necesariamente con rigidez, pero si con un tipo de orden y pasos a seguir (Millington & Funge, 2009).

4.3 Algoritmos Genéticos

4.3.1 Evolución en la Naturaleza

Los algoritmos genéticos permiten buscar soluciones a problemas algorítmicos y se basan en el principio de evolución observable en los sistemas naturales. Los procesos evolutivos de dichos sistemas cuentan con las siguientes características (Schwab, 2004):

- ✓ Para sobrevivir como especie, todas las criaturas vivientes necesitan reproducirse. La reproducción se lleva a cabo, de forma muy simplificada, ejecutando las reglas codificadas que son necesarias para construir un organismo. Estas reglas son almacenadas en cadenas de ADN (constituidas por proteínas) llamadas cromosomas, encontrados en cada una de las células que conforman a un ser vivo.
- ✓ Los cromosomas, a su vez, se componen de pequeñas secuencias modulares llamadas genes, que consisten en varias permutaciones de las cuatro proteínas básicas: timina, adenina, citosina y guanina. Cada gen almacena información sobre

los “ajustes”, o alelos, de un rasgo en particular (también puede ser un número de rasgos dado que cada gen usualmente se encuentra ligado a más de un rasgo dentro de un cuerpo).

- ✓ Cuando dos padres se reproducen su ADN se divide. En el hijo la mitad del ADN proviene de uno de los padres y, la otra mitad, del otro. Este proceso se llama cruce (crossover) o recombinación genética.
- ✓ El cruce genético resulta en una nueva mezcla de rasgos genéticos que son traspasados a la descendencia. Si esta nueva mezcla es buena, el hijo tendrá una vida plena y podrá reproducirse también. Por el contrario, si el hijo hereda rasgos más débiles, podría tener una vida corta, limitaciones para reproducirse o no ser un compañero deseable. Tras varias generaciones, la tendencia de los organismos que cuentan con una mejor mezcla de rasgos apuntará a la generación de sociedades de criaturas genéticamente superiores dentro de su especie. La medida de la calidad de la mezcla de rasgos se denomina fitness (aptitud o ajuste). Mientras más elevado es el valor fitness, la criatura será más capaz de introducir sus rasgos en la población.
- ✓ A veces, una falla ocurre dentro del sistema (es tema de discusión si esta falla tiene implicancias positivas o negativas dentro del sistema). Cuando esto ocurre, un gen dentro de un organismo hijo cambia creando uno completamente nuevo, de forma que no es posible realizar un trazado del gen y atribuirlo a uno de sus padres. El gen es replicado erróneamente. Este evento se denomina mutación, resultando en que el alelo de un gen se vuelve aleatorio (con efectos aleatorios dentro del organismo). En la mayoría de los casos, se obtienen rasgos negativos. Por ejemplo, un ave puede nacer con alas demasiado pequeñas para volar.
- ✓ A veces, una mutación resulta en que el hijo puede desempeñarse mejor dentro de su entorno o, de alguna manera, hacerlo más propenso a reproducirse. Cuando esto ocurre, el gen positivo puede pasar a formar parte de las siguientes generaciones.
- ✓ Este paradigma de supervivencia de los más aptos gradualmente cambia el conjunto de rasgos (llamado genoma) que las especies contienen en promedio, llevándolas en dirección del conjunto ideal, que representa los genes mejor adaptados para una criatura en particular para las condiciones actuales de su entorno.

4.3.2 Evolución en los Videojuegos

Una correcta implementación de un algoritmo evolutivo puede dotar de inteligencia artificial a un videojuego. En este proceso de búsqueda evolutiva, dirigido por el ajuste genético, se recorre un abanico de posibles soluciones para un problema dado.

Este método se divide en dos partes bastante inequitativas: evolucionar una solución y utilizar la solución. Generalmente, el uso de la información obtenida con un algoritmo genético en el juego final es una operación que puede verse como una caja negra. El proceso de evolucionar una solución representa la mayor parte del trabajo y, usualmente, se lleva a cabo durante el desarrollo del juego. Son pocos los juegos que salen al mercado con sus partes evolutivas aún activas y sus componentes de aprendizaje son monitoreados muy de cerca. Los rasgos y comportamientos que se ven influenciados por los elementos de aprendizaje son construidos de forma tal que restringen el aprendizaje o elementos genéticos, por lo que el

aprendizaje o la evolución están altamente controlados. La naturaleza de estos juegos otorga imprevisibilidad y, a su vez, libertad de acción para comportamientos torpes o inapropiados.

Los algoritmos genéticos pertenecen a la clase de métodos de búsqueda estocásticos, lo que significa que la aleatoriedad es un factor relevante en el direccionamiento de la búsqueda. Es por ello que son requeridas muchas iteraciones para alcanzar una mejor solución. Si bien este factor de aleatoriedad podría alejar del camino a la mejor solución, es menos probable quedarse atrapado en una solución particular, es decir, puede llevar la búsqueda desde un conjunto de máximos locales a uno de máximos globales.

Se debe considerar que los algoritmos genéticos no garantizan desempeño o éxito. De hecho, podrían llegar a comportarse de la peor forma. Este es el precio de integrar elementos de aleatoriedad en el algoritmo y, probablemente, será necesario ajustar la estructura de los genes, e incluso la configuración del algoritmo y operadores si no se obtiene el comportamiento esperado. En el peor de los casos, se podría llegar a determinar que los algoritmos genéticos no se ajustan al problema planteado.

4.4 Razonamiento Basado en Casos

El razonamiento basado en casos es un paradigma para resolución de problemas que a través de la utilización de conocimiento previamente adquirido sobre un problema en específico, llamado caso, intenta solucionar un problema actual similar. Permite independizarse del conocimiento general del dominio del problema como también de generar asociaciones de relaciones entre las descripciones y conclusiones del problema.

También genera un aprendizaje incremental, ya que las nuevas experiencias son retenidas cada vez que un problema es resuelto, transformándose en un caso, y habilitado para ser utilizado en el futuro. Un nuevo problema es solucionado encontrando un caso anterior similar y reutilizando la forma en la que se solucionó para esta nueva situación.

Este tipo de procedimiento o paradigma de resolución de problemas, utilizando casos exitosos del pasado, es una forma frecuentemente aplicada por los seres humanos para resolver problemas de la vida real (Ross, 1989). Un médico al momento de diagnosticar a un paciente recuerda pacientes anteriores con síntomas similares que le permitan entender cuáles serían las posibles causales de dichos síntomas y poder deducir de mejor manera cual es el problema o enfermedad que presenta el paciente.

Un caso es definido como una situación problemática solucionada, cuya experiencia ha sido capturada y aprendida de forma que pueda ser utilizada en un futuro para solucionar problemas similares, también conocido como casos pasados, casos anteriores o casos retenidos. Un nuevo caso es aquel que no ha sido encontrado con anterioridad o simplemente ya no puede ser solucionado con la experiencia anterior.

Por lo tanto se genera un proceso cíclico integrado de resolver un problema, aprender de dicha experiencia, resolver nuevos problemas y volver a repetir el ciclo. Esta dinámica genera

varios desafíos con lo que respecta a: justificar la solución propuesta, interpretar los problemas, generar posibles soluciones y generar expectativas de estados futuros.

Generalmente, si un caso nuevo se intenta solucionar con un caso antiguo y se encuentran deficiencias en las que el caso antiguo no permite resolver de la mejor forma el caso nuevo, entonces se intenta modificar la forma en que se solucionó el problema anterior haciendo los ajustes necesarios e intentando nuevas posibilidades, si el resultado es positivo se procede a actualizar el caso antiguo para que considere la nueva experiencia adquirida (Aamodt & Plaza, 1994).

El modelo que explica de forma global el funcionamiento del algoritmo es dividido por (Aamodt & Plaza, 1994) en cuatro procesos: Recuperar casos similares, Reutilizar la información y experiencia para resolver el nuevo caso, Revisar la propuesta de solución y Retener las partes de experiencia que puedan ser utilizadas en el futuro. Este modelo se muestra en la Figura 4.1.

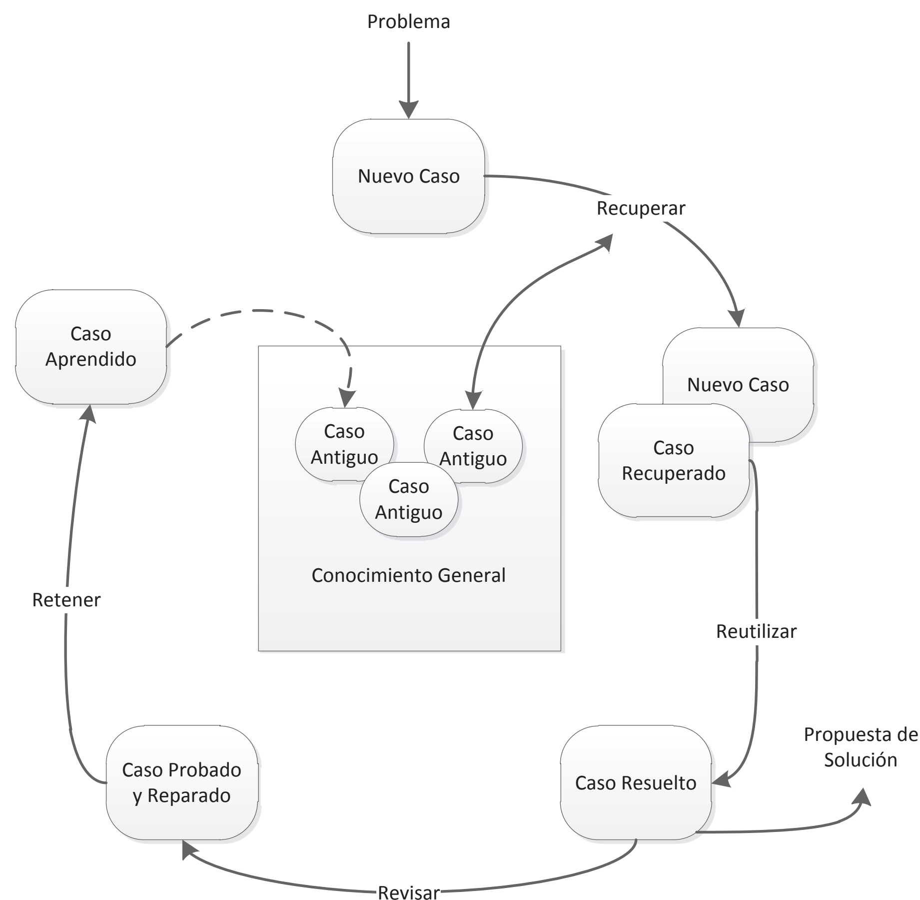

Figura 4.1 Modelo de Razonamiento Basado en Casos

4.5 Modelo Belief-Desire-Intention

El modelo de Convicción, Deseo e Intención (BDI por sus siglas en inglés) provee un mecanismo para separar la actividad de seleccionar un plan desde una base de planes y la actividad de ejecutar activamente dichos planes. Permite generar un equilibrio entre el tiempo que se destina a seleccionar un plan y el tiempo que se usa ejecutándolos. Cabe destacar que no es parte del modelo la creación de estos planes, por lo que esta actividad queda a cargo del desarrollador.

Este modelo posee como clave el razonamiento limitado por recursos. Estos límites de recursos surgen debido a que los sistemas, donde posiblemente se ejecuten dichos procesos simbolizados por el modelo, poseen un poder computacional limitado o nivel de memoria limitado. Esto puede afectar el nivel de razonamiento que se les da a los procesos que intentan encontrar una solución a un problema determinado.

Por otro lado, tiene como ventaja el soportar ambientes dinámicos de trabajo, en el que en cada momento el estado actual puede variar por cambios externos, y es deber de la implementación del modelo adaptarse de la mejor forma a estos cambios. Las implicaciones que poseen los ambientes dinámicos para el modelo Belief-Desire-Intention son:

- ✓ Los procesos no se pueden detener para permitir, en la ejecución de la implementación del modelo, razonar sobre cómo resolver el problema, sino que debe hacerlo en tiempo real.
- ✓ No se puede volver a calcular los planes de acción cada vez que existan cambios en el ambiente de trabajo, ya que esto generaría un continuo momento de planeación y que nunca llegaría a la etapa de ejecución.

En general, para poder solucionar estos dos problemas es que se generan implementaciones del modelo de forma más reactiva y con menor razonamiento. Pero esto requeriría una cantidad de trabajo, a nivel de codificación, mucho mayor al de generar una implementación basada, en mayor proporción, en planes y razonamiento.

Este modelo es una implementación de los aspectos principales presentados por el modelo de Bratman, (1999). Este modelo considera a la intención y el deseo como actitudes proactivas de la mente humana, y pone en énfasis a la intención, como la responsable de controlar la conducta del deseo. El compromiso, que es un factor fundamental entre deseo e intención, conlleva a una persistencia temporal a la ejecución de planes, así como también a la creación de nuevos planes en relación a los compromisos ya establecidos. El modelo aplicado a la computación, por otro lado, no considera la persistencia temporal, simplificando la ejecución.

La arquitectura del modelo contempla cuatro elementos fundamentales (Rao & Georgeff, 1995):

- ✓ **Convicción**: representa el estado informacional, como su convicción del mundo donde se encuentra situado, incluyéndose a sí mismo. Las convicciones pueden

también incluir reglas inferidas que permitan un encadenamiento de deducciones para obtener nuevas convicciones del estado informacional.

  - Set de Convicciones: son almacenados para futuro uso, aunque esto es una decisión de implementación.

- ✓ **Deseo**: representa el estado motivacional, considera los objetivos o situaciones que se desean cumplir. Ejemplos de deseos pueden ser “aumentar recursos” y “disminuir fuerzas del contrincante”.
  - Objetivos: un objetivo es un deseo que fue escogido para ser ejecutado. Los objetivos deben ser consistentes, esto quiere decir que no deben contradecirse entre ellos. Por ejemplo, “aumentar el ataque” y “disminuir el ataque”. Esto no quiere decir que aquellos no serían deseos válidos, pero no pueden pasar a ser objetivos de forma simultánea.
- ✓ **Intención**: representa el estado deliberado, son las acciones que ya se decidieron que se realizarían. Esto quiere decir que estas acciones ya comenzaron a ejecutarse o están en un estado de terminación.
  - Planes: son una secuencia de acciones que se ejecutan para cumplir una determinada intención. Algunos planes pueden incluir otros planes. Un plan para “construir estructuras de ataque” puede incluir el plan “aumentar recursos”.
- ✓ **Evento**: representa activadores de una reacción por medio de un estímulo que ocurre durante la ejecución.

5 Estado del Arte

En años recientes, los avances en hardware han permitido grandes mejoras en gráficos y sonido. Si bien esto ha resultado beneficioso para los jugadores, también les ha facilitado la posibilidad de detectar comportamientos sin sentido o cuestionables dentro de los videojuegos. Por lo tanto, se ha vuelto importante que estos reflejen un nivel apropiado de inteligencia, con el fin de desarrollar productos más atractivos en un mercado cada vez más competitivo (Johnson & Wiles, 2001).

La aplicación de inteligencia artificial a los videojuegos de estrategia en tiempo real ha sido un desafío que se ha intentado solucionar desde los inicios del desarrollo masivo de entretenimiento electrónico de este tipo (Firclough, et al., 2002). Con el surgimiento en la popularidad de este género, se ha comprobado que los jugadores prefieren jugar en contra de seres humanos en oposición a jugar contra el computador. Esto debido a que la mayoría de implementaciones para la inteligencia del jugador controlado por el computador se ha desarrollado en base a máquinas de estados finitos que generan un comportamiento altamente predecible y, por lo tanto, disminuyendo la capacidad que posee el videojuego de sorprender y entretener a largo plazo (Cheng & Thawonmas, 2005).

Uno de los desafíos de implementar inteligencia a videojuegos de estrategia en tiempo real es que normalmente poseen un gran número de objetos, información incompleta, micro acciones, y reacción de forma inmediata, por lo que para diseñar un modelo adecuado debe considerarse gran parte de estas condiciones (Buro & Furtak, 2003). Una alternativa de aproximación a la implementación utilizada por ciertos desarrolladores es codificar la inteligencia para cada situación, por supuesto que esto conlleva a que en cada momento del transcurso del juego se pueda saber exactamente que determinación tomará el jugador manejado por el computador. Por otro lado, el tiempo necesario para desarrollar este tipo de inteligencia crece considerablemente entre más condiciones y elementos contenga el videojuego (Ontañón, et al., 2008).

A continuación se presentan alternativas que se utilizan en la implementación de videojuegos. Algunas son más eficientes que otras debido a la cantidad de procesamiento requerida durante su ejecución.

5.1 Algoritmos Genéticos

En particular, entre las técnicas de inteligencia artificial, los algoritmos genéticos suelen ser bastante costosos en términos de procesamiento, pues, para obtener soluciones cercanas al óptimo es necesario recorrer muchas generaciones en una gran población de posibilidades. Es por esta razón que, en el pasado, el uso de estos algoritmos resultaba una opción bastante costosa (a esto se debe que la mayor parte del trabajo evolutivo se realice dentro del ambiente de desarrollo). Dado el incremento generalizado en el poder de procesamiento de los computadores, el uso de estos métodos se ha vuelto cada vez más común (Schwab, 2004).

Existen trabajos que demuestran el interés por investigar el tema. Por ejemplo, se utilizó un algoritmo genético para ajustar el comportamiento de los *bots* del conocido videojuego de disparos en primera persona Counter Strike, donde se evolucionó un conjunto de parámetros que otorga a un *bot* un comportamiento equivalente al de uno con parámetros establecidos por un humano con conocimientos avanzados en la estrategia del videojuego (Cole, et al., 2002). En otro trabajo (Watson, et al., 2004), se optimiza el desarrollo de una ciudad en Freeciv, un videojuego OpenSource de estrategia por turnos. Se implementó un algoritmo genético que ajusta factores de desarrollo de la ciudad que optimizan el desarrollo, permitiendo que el jugador humano preste atención a otros factores del juego.

Entre los géneros de videojuegos, casi todos pueden utilizar técnicas de algoritmos genéticos en alguno de sus aspectos. En los videojuegos de estrategia en tiempo real, por ejemplo, se pueden resolver problemas difíciles como determinar el orden de construcción o defenderse de una táctica en particular como la denominada *rushing*, que consiste en una técnica común de los seres humanos que involucra la creación de unidades en masa en etapas tempranas del juego y luego atacar rápidamente para acabar con el oponente mientras se encuentra realizando sus primeras construcciones. En el género de carreras, algunas compañías utilizan algoritmos genéticos en etapas de desarrollo (offline) para calibrar el comportamiento de los automóviles. Los resultados obtenidos se almacenan y se utilizan directamente durante la ejecución del videojuego (Schwab, 2004).

Dentro del género de estrategia en tiempo real (RTS), algoritmos genéticos se utilizaron con éxito en el juego NERO (Neuro Evolving Robotic Operatives)[1]. En este videojuego los soldados pueden ser entrenados para la guerra dejándolos ejecutar tareas una gran cantidad de veces. Continuamente se producen nuevos soldados basándose en aquellos que mejor se desempeñaron en la generación anterior (Walther, 2006).

5.2 Razonamiento Basado en Casos

Existen diversos trabajos aplicados a videojuegos de estrategia en tiempo real con la utilización de razonamiento basado en casos. En un trabajo particular, (Cheng & Thawonmas, 2005), pudieron emplear un modelo para reconocer la planificación y razonamiento de las unidades del juego para disminuir su predictibilidad, tomando en consideración que la representación debía minimizar los recursos necesarios y así poder ser desplegada en una gran variedad de plataformas.

Un ejemplo de razonamiento basado en casos, desarrollado por (Ontañón, et al., 2008), permitió que el comportamiento aplicado por la inteligencia del computador contrincante pudiera ser enseñado por medio de demostración, y no tener que ser escrita a base de líneas de código. Esto permitiría que los desarrolladores de videojuegos de estrategia pasen menos tiempo desarrollando la inteligencia artificial y más tiempo desarrollando las características del videojuego. Si el sistema presentara un comportamiento incorrecto en una situación dada,

[1] http://www.nerogame.org/ - Revisada última vez 18 de Abril de 2012

aprendería de este error, y en un caso futuro utilizaría una mejor implementación para dicha situación.

Otra contribución realizada por (Aha, et al., 2005) presenta una arquitectura integrada para razonamiento basado en casos aplicado a videojuego de estrategias. En esta arquitectura, el diseño de la planificación, la composición, la adaptación y la ejecución se encuentran intercaladas. El planificador mantiene y coordina todas las metas del plan actual, y para cada plan en desarrollo, busca el mejor comportamiento en la base de casos dependiendo del estado del videojuego. Este comportamiento luego se suma al plan actual, adaptándose de ser necesario, para ser ejecutado a la brevedad.

La mayoría de estas aplicaciones prácticas del modelo han adoptado un enfoque con la particularidad de que cada caso contendrá un comportamiento que especificará la información relevante para que el algoritmo pueda integrarse con el funcionamiento en tiempo real del videojuego.

Un comportamiento estará dividido en dos partes, una declarativa y una procedimental. La parte declarativa tiene como objetivo entregar la información necesaria al sistema acerca del uso que se le dará al comportamiento. La parte procedimental es el conjunto de pasos que se ejecutarán para realizar la declaración.

La parte declarativa de un comportamiento está constituida por tres elementos:

- Un objetivo, que es la representación de la intención del comportamiento. Cada dominio debe poseer un lenguaje para expresar los objetivos. Para un juego de estrategia, como el que se presenta en este documento, un objetivo puede ser "aumentar la producción de recursos".
- Un conjunto de precondiciones que deben estar presentes antes de que el comportamiento pueda ser ejecutado. Un ejemplo de precondición, para el caso del juego de estrategia y el objetivo de "aumentar la producción de recursos", es el de contar con recursos para la construcción de unidades recolectoras.
- Un conjunto de condiciones de existencia que representan las condiciones necesarias para que un comportamiento en ejecución tenga posibilidades de éxito. Si en algún momento, durante la ejecución del comportamiento, las condiciones de existencia no están presentes, dicho comportamiento debe ser puesto en estado pendiente, ya que será imposible que pueda cumplir su objetivo. Si en el videojuego de estrategia se tiene por objetivo construir una infraestructura de unidades recolectoras, y en algún momento no quedan recursos suficientes para construir más recolectores, entonces el comportamiento se detiene y queda en espera hasta que se cumplan nuevamente las condiciones de existencia, o simplemente se rechaza el comportamiento y vuelve a su estado anterior.

La parte procedimental de un comportamiento está conformado por código ejecutable que determina tres elementos:

- Un conjunto de acciones que pueden ser usadas en el dominio. Son representaciones básicas de funcionalidades de la aplicación. El videojuego cuenta con acciones como “construir” y “destruir”.
- Uno o más sensores que permitan obtener la información actualizada del estado del videojuego. Estos sensores deberían entregar información relevante para la toma de decisiones sobre los comportamientos. En el videojuego, alguno de los elementos principales que conforman la información relevante son “número de unidades de ataque en el terreno”, “cantidad de recursos disponibles” y “unidades desconectadas del suministro de energía”.
- Un subconjunto de objetivos que deben ser ordenados jerárquicamente según las relaciones que existan entre ellos. Si el comportamiento tiene una estructura compleja que requiera de otros comportamientos para poder cumplir los objetivos de más bajo nivel, entonces, dado este orden jerárquico, se podrá determinar cuáles otros comportamientos deberán activarse primero.

5.3 Modelo Belief-Desire-Intention

El desarrollo del videojuego *Black & White* fue fuertemente influenciado por la visión de inteligencia artificial de su creador. Ésta utilizaba el modelo BDI para permitir que al personaje principal se le pudieran enseñar acciones, y éste las pudiera utilizar adaptándose a diversas situaciones, dependiendo del estado donde quisiera ejecutarlas (Molyneux, et al., 2002).

Un trabajo importante en el área de videojuegos de estrategia en tiempo real fue llevado a cabo por (Kok, 2009) quien utilizó la tecnología de agentes inteligentes, más específicamente agentes inteligentes basados en objetivos, y adaptados a la arquitectura del modelo BDI. Los agentes desarrollados razonan en un alto nivel basados en la arquitectura BDI operando muchas veces en terreno desconocido. Tienen el deseo de resolver un problema, y adoptan un plan para ello. Para resolver cómo se solucionará el problema debe utilizar la convicción que tiene sobre el estado del juego como también su propio estado. Finalmente es el plan quien dicta cuáles serán las acciones a considerar para cumplir el objetivo.

Esta implementación utilizó la plataforma 2APL para el desarrollo de los agentes y la arquitectura BDI. Esto permitió que se pudieran diseñar comportamientos de adaptación y desprendimiento de objetivos durante la ejecución del juego, elaboración de planes abstractos con reglas procedimentales e interacción con el ambiente del videojuego.

6 Ambiente de Desarrollo

En este apartado se mencionarán y describirán de forma general las herramientas y frameworks de desarrollo utilizados para apoyar el proceso de implementación del videojuego.

6.1 Sistema de Control de Versiones (VCS)

Un sistema de control de versiones (VCS) permite registrar los cambios realizados sobre un archivo o un conjunto de ellos, de forma tal que es posible recuperar una versión específica más tarde. Hoy en día es bastante común que las empresas orientadas al desarrollo de software incluyan dentro de sus políticas de desarrollo de software estos sistemas para el control de las versiones/revisiones del producto. Esta práctica permite al equipo desarrollador revertir archivos a un estado previo, revertir el proyecto completo a un estado previo, comparar los cambios de archivos/componentes/sistema a través del tiempo, determinar el individuo y el momento en que se modificó algo que podría estar causando conflictos, obtener estadísticas de actividad en el proyecto, etc.

Los VCS se pueden clasificar en tres categorías (Chacon, 2009): Sistemas Locales de Control de Versiones, Sistemas de Control de Versiones Centralizados y Sistemas de Control de Versiones Distribuidos.

6.1.1 Sistemas Locales de Control de Versiones

El método de control de versiones utilizado por muchas personas consiste en copiar archivos en diferentes directorios con nombres que describan la versión de los archivos que contiene. Este acercamiento es bastante común debido a su simpleza. Sin embargo, es un método demasiado propenso a errores. Es fácil olvidar el directorio de trabajo y modificar accidentalmente otros archivos.

Para hacer frente a este problema, algunos programadores desarrollaron VCS locales que contaban con una base de datos simple que guardaba todos los cambios de los archivos que se mantenían bajo control de versiones (ver Figura 6.1).

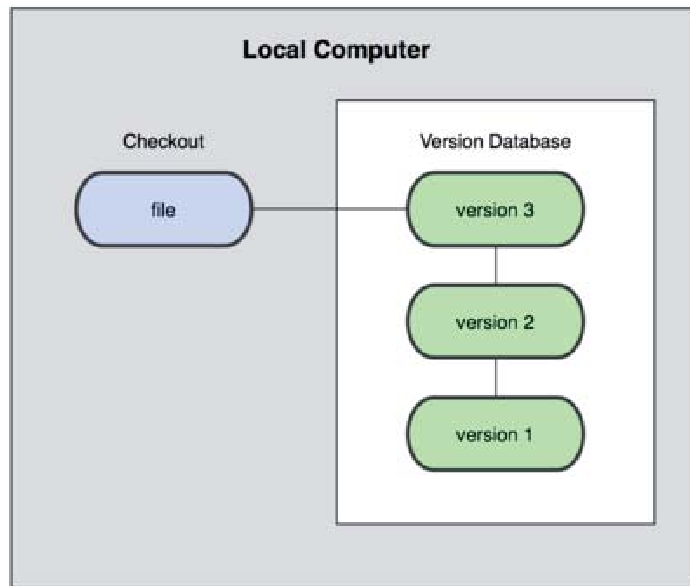

Figura 6.1 Esquema de Sistemas Locales de Control de Versiones

Uno de los VCS más populares era un sistema llamado *rcs* (por el nombre del comando UNIX), el cual se distribuye con muchos sistemas operativos hasta el día de hoy (por ejemplo, distribuciones de Linux y Mac OS X).

6.1.2 Sistemas de Control de Versiones Centralizados

El siguiente gran problema que se presentó era la necesidad de trabajar de modo colaborativo con otras personas. En este punto se desarrollaron los Sistemas de Control de Versiones Centralizados (CVCS). Entre estos sistemas se encuentran, por ejemplo, CVS, Subversion y Perforce. En estos sistemas es sólo un servidor el que contiene todos los archivos bajo control de versión. Además, se cuenta con un conjunto de usuarios remotos que obtienen mediante un *checkout* versiones o revisiones específicas de los archivos en dicho servidor. Por muchos años, este ha sido el estándar como método de control de versiones (ver Figura 6.2).

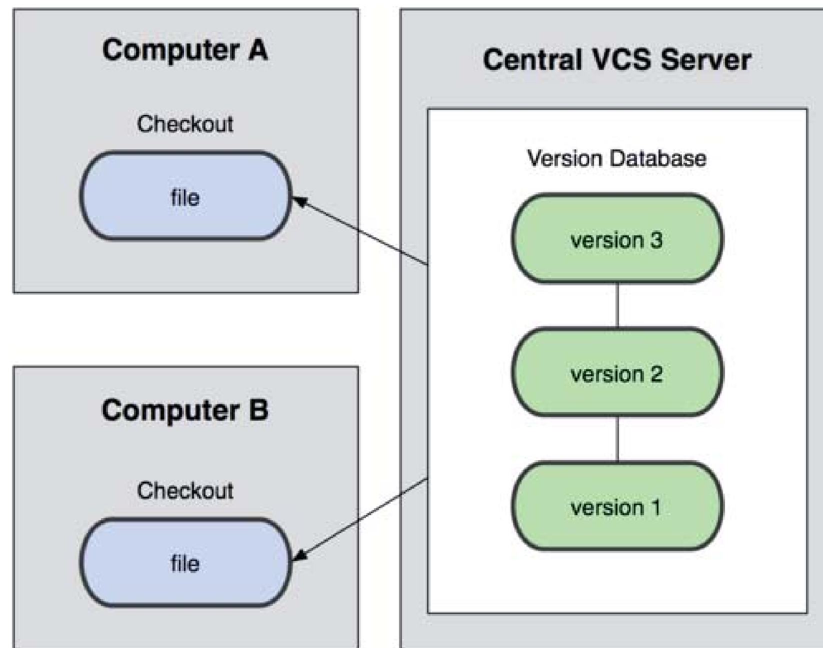

Figura 6.2 Esquema de Sistemas de Control de Versiones Centralizados

Esta configuración cuenta con varias ventajas, entre ellas por ejemplo, todo el conjunto de usuarios sabe lo que otros hacen sobre el proyecto. Los administradores tienen control absoluto sobre quién hace qué. Además, es más sencillo administrar un CVCS que administrar bases de datos locales ubicadas en cada cliente.

Sin embargo, esta configuración también cuenta con serios problemas. El principal está relacionado con el escenario en que falle el servidor central. Durante el tiempo que el servidor esté caído, nadie podrá colaborar con otros, ni guardar cambios versionados. Peor aún, si el disco duro del servidor se corrompe y no existen respaldos apropiados, se perderá absolutamente toda la historia del proyecto. Lo único que quedará serán snapshots del proyecto distribuidos entre los clientes. Hay que resaltar que los CVS Locales también sufren de este inconveniente. Siempre que la historia completa del proyecto exista en un sólo lugar, se corre el riesgo de perder todo.

6.1.3 Sistemas de Control de Versiones Distribuidos

En un sistema de control de versiones distribuido (DVCS) los usuarios no sólo recuperan una versión/revisión determinada desde el repositorio, sino que se convierten en un espejo de éste, guardando en la copia de cada usuario la historia de todo el proyecto. Por lo tanto, si los datos en el servidor se corrompen, cualquiera de los repositorios cliente puede copiarse al servidor para restablecer los datos con toda su historia. En el fondo, cada checkout es en realidad una copia de respaldo completa de todos los datos (ver Figura 6.3). Ejemplos de sistemas dentro de esta categoría son: Git, Mercurial, Bazaar y Darcs.

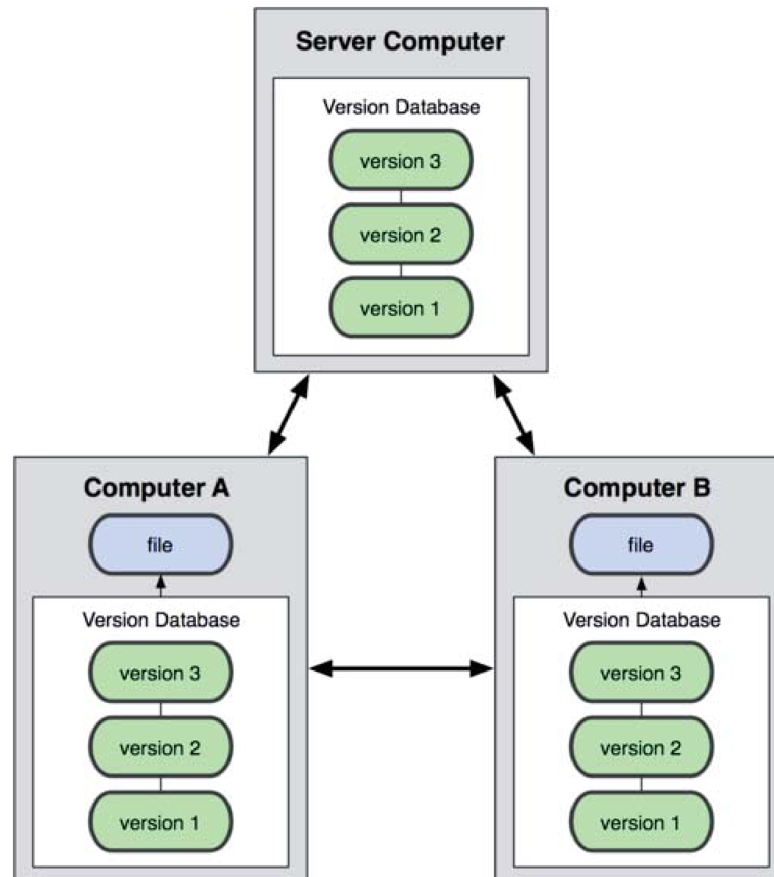

Figura 6.3 Esquema de Sistemas de Control de Versiones Distribuidos

Tomando en consideración las características de las categorías de los CVS, el control de versiones del prototipo del videojuego se llevará a cabo utilizando Git, principalmente por el hecho de mantener copias completas dentro de cada repositorio cliente, disminuyendo el riesgo de pérdida total de datos.

Para evitar gastar tiempo de instalación y configuración del servidor Git, se utilizará el servicio que ofrece GitHub[2]. El servicio de hospedaje de repositorios Git en GitHub es gratuito, pero, por términos de uso, todos los repositorios gratuitos son públicos. Por razones de privacidad, el código fuente del prototipo debe ser privado, es por ello que se optó por contratar un plan que permita crear repositorios privados (plan micro).

[2] https://github.com - Revisada última vez 18 de Abril de 2012

6.2 Modelado y Animación en 3D

El desarrollo de aplicaciones 3D, como los videojuegos, requiere del uso y conocimiento de herramientas especializadas para el modelamiento y animaciones de mallas poligonales. Estos modelos serán incluidos en la aplicación final y, posteriormente, reproducidos en tiempo real por el motor del videojuego.

Entre las herramientas de desarrollo de gráficos 3D más conocidas se encuentran Google SketchUp, Blender y 3D Studio Max. Cada una posee ventajas y desventajas con respecto a las otras y queda en manos del diseñador escoger cuál es la que más le acomoda por su estilo de trabajo.

La herramienta que se utilizará para el diseño de los modelos en tres dimensiones será Blender[3], en su versión 2.5 estable. Si bien esta herramienta ha sido criticada por una interfaz de usuario poco intuitiva, la cantidad de tutoriales disponibles, completa compatibilidad con el motor de videojuegos jMonkey y su calidad de software gratuito la convierten en una alternativa conveniente para apoyar el desarrollo del videojuego. Incluye también la posibilidad de generación de materiales, texturas, luminosidad y animación.

6.2.1 Blender

Blender es una aplicación integrada que permite la creación de una gran variedad de contenido en dos y tres dimensiones. Posee una arquitectura de código abierto que provee interoperabilidad de plataforma, extensibilidad y flujo integrado de trabajo.

Contiene una suite de creación completamente integrada con herramientas para el desarrollo de contenido 3D, modelado, traspaso de modelos 3D a representación 2D, creación de texturas, animación de modelos, utilización de partículas, simulación, scripting, composición e incluso un motor de videojuegos.

Para el diseño de los modelos en tres dimensiones se siguen tres pasos básicos: modelado en tres dimensiones, creación de material y creación de texturas.

6.2.2 Modelado 3D

Esta etapa requiere la utilización de técnicas de translación, rotación y escalado sobre la malla poligonal que se está creando. Se debe poseer una imagen clara de lo que se desea crear, ya sea mentalmente o en un dibujo previo, y transformando los vértices, caras y aristas para concretar la visión generada. Para esta etapa es importante tener en cuenta dos tipos de vistas incluidas en la herramienta de desarrollo, la vista objetual y la vista de edición (ver Figura 6.4).

[3] www.blender.org - Revisada última vez 22 de Septiembre de 2012

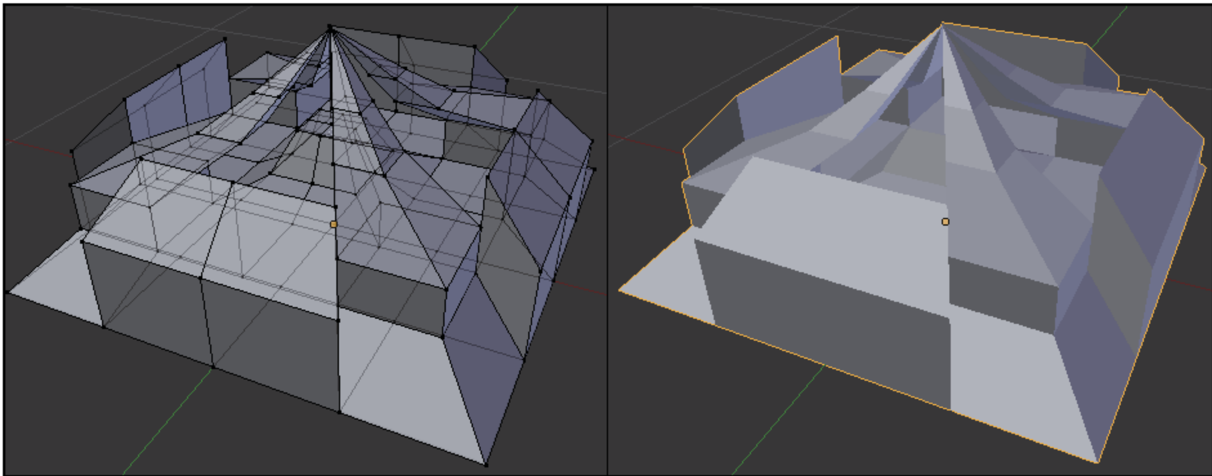

Figura 6.4 Vista de edición (izquierda) y Vista objetual (derecha)

Una técnica importante incluida en Blender es la posibilidad de extrudir un vértice, arista o cara, generando un nuevo polígono cuyo inicio es el final del polígono extruido y que en forma es idéntico al polígono inicial. Esta técnica permite ahorrar tiempo en la generación y unión de nuevos polígonos o mallas poligonales.

6.2.3 Creación de Material

En esta etapa se le incluye un material al objeto creado. Este material define las propiedades visuales que contendrá el objeto y como deberá ser reproducido por el motor del videojuego. Las propiedades que influyen en un objeto pueden ser sombreado difuso, sombreado especular, reflectividad, transparencia y translucidez.

Cada parámetro puede ser modificado en la herramienta y queda en manos del desarrollador brindarle un efecto realista al objeto que desea modelar. El objeto anterior puede ser modificado para tener color, propiedades para las reflexiones y refracciones de luz y transparencia. En este caso sólo será pintado de color amarillo (ver Figura 6.5).

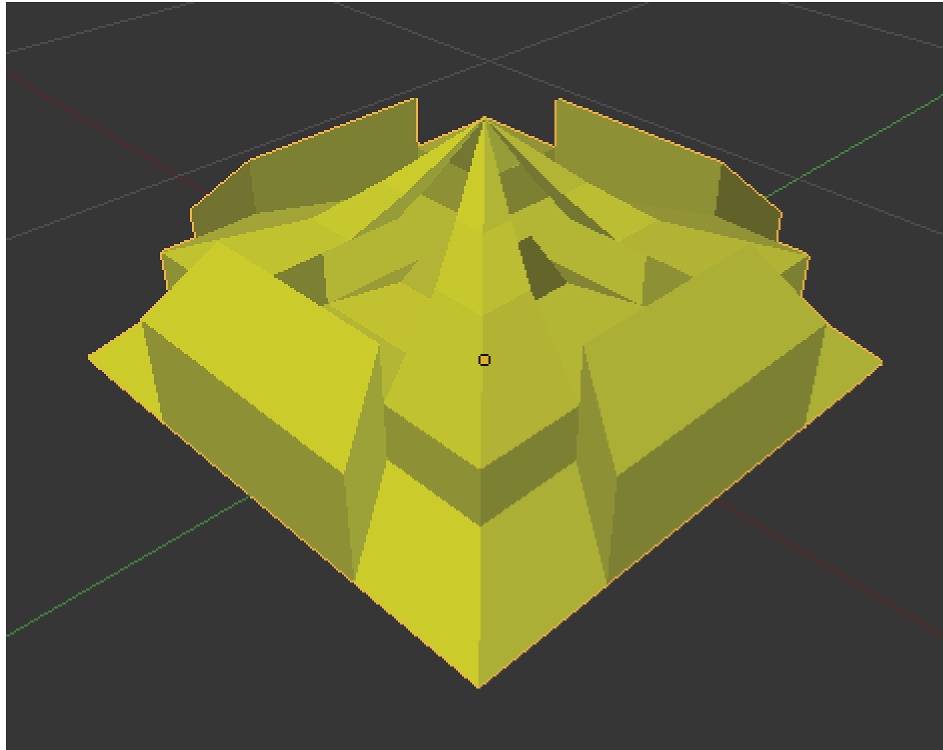

Figura 6.5 Modelo con material de color amarillo.

6.2.4 Creación de Textura

Teniendo el modelo y el material completo es posible incluir la textura final que cubrirá el modelo completo y permitirá añadir detalles a la visualización del objeto. Esta textura puede ser una imagen o un patrón prediseñado que transforme los colores del material en formas dictadas por la figura de la textura. También tienen la capacidad de cambiar otros aspectos del material cuyas propiedades fueron mencionadas anteriormente.

Muchas veces se utilizan varias capas de texturas para crear efectos realistas y que remuevan la uniformidad que producen los materiales. Con estas técnicas y el uso adecuado del nivel donde se aplicará la textura es posible crear objetos metálicos, tejidos, de piel, madera, entre otros.

Al utilizar imágenes como texturas es importante haber mapeado el modelo 3D en coordenadas UV (debido a que las letras X, Y y Z ya son utilizadas para las coordenadas en 3D) y generar un modelo plano que luego puede ser editado con una herramienta de diseño en dos dimensiones para un acabado final (ver Figura 6.6).

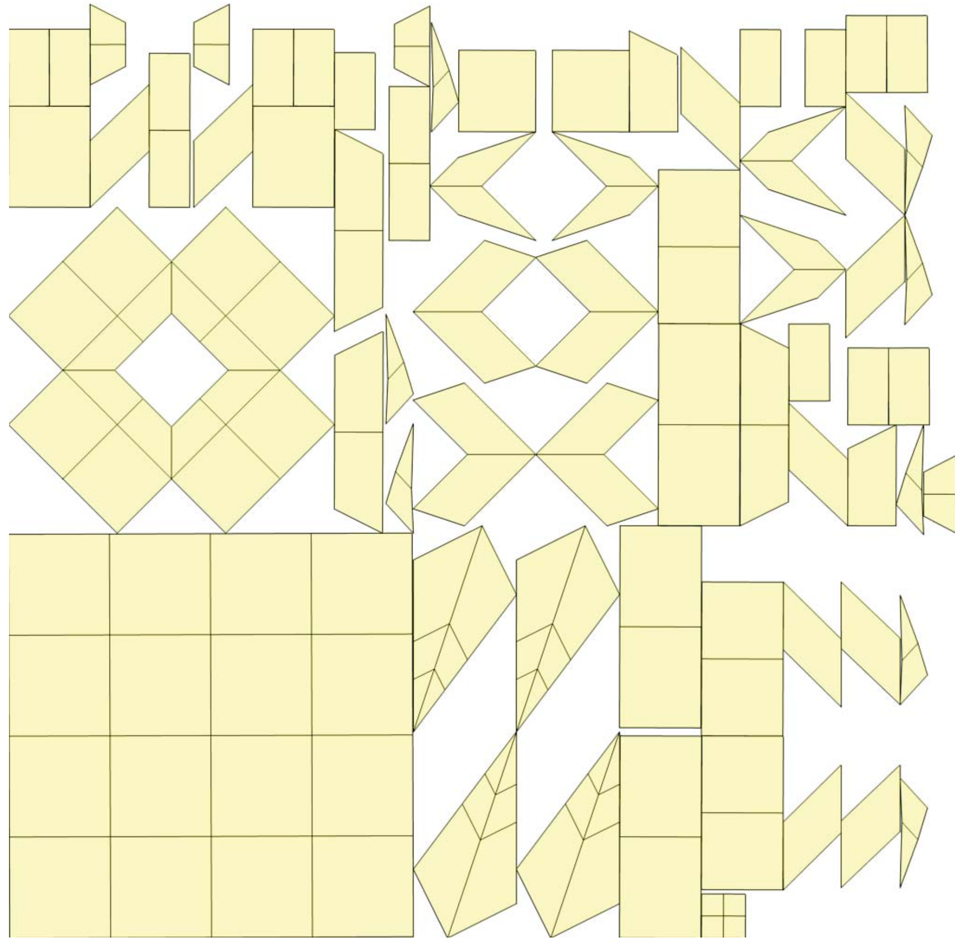

Figura 6.6 Mapeo UV del modelo de ejemplo.

6.3 Motor de Videojuego

El concepto de motor de videojuego se establece a mediados de 1990 en referencia a los videojuegos de disparos en primera persona (first-person shooter) como el popular Doom (Figura 6.7) de la compañía id Software. La arquitectura de Doom contaba con una separación bien definida entre sus componentes base (sistema de dibujado de gráficos en 3D, sistema de detección de colisiones o el sistema de audio), modelos gráficos y texturas, modelos/mapas de mundos y las reglas de juego que conforman la experiencia de juego. El valor obtenido con esta separación se hizo evidente cuando los desarrolladores comenzaron a reutilizar estos componentes base, requiriendo aplicar pequeñas modificaciones al motor. Esto dio inicio a una comunidad compuesta por jugadores y pequeños estudios independientes que desarrollaron nuevos videojuegos modificando algunos ya existentes, utilizando herramientas gratuitas distribuidas por los desarrolladores originales (Gregory, 2009).

Figura 6.7 Captura de pantalla de Doom.

En la práctica, en ningún videojuego se ha logrado una separación completa entre el videojuego y su motor. Como tendencia general, los motores de videojuegos se ajustan mejor a un género específico para el cual fue desarrollado. Esto es sólo una tendencia general, y no impide que un motor pueda ajustarse a varios géneros de videojuegos. Como regla general puede decirse que mientras más general sea el propósito del motor, será menos óptimo para ejecutar un videojuego específico en una plataforma específica. Dicho de otro modo, es probablemente imposible construir un motor que permita desarrollar cualquier videojuego imaginable capaz de correr óptimamente en múltiples plataformas. Sin embargo, esto no debe tomarse como una regla, pues, la llegada de hardware cada vez más potente en términos de procesamiento, parece degradar dichas restricciones relacionadas al género-plataforma.

Para el caso en estudio de este proyecto cabe tomar en cuenta consideraciones asociadas al género de videojuegos de estrategia en tiempo real. En este género, el jugador debe desplegar las unidades y estructuras que tiene a su disposición, de forma estratégica, en un escenario generalmente de gran tamaño. La estrategia empleada por el jugador deberá permitir vencer a uno o más oponentes dentro del mismo escenario.

El escenario de juego generalmente se le presenta al jugador con una vista oblicua desde arriba hacia abajo (ver Figura 6.8) y con libertad bastante restringida de modificar el ángulo la cámara. Esto permite limitar la cantidad de gráficos dibujados en pantalla manteniendo una ejecución fluida del videojuego.

Figura 6.8 Vista oblicua desde arriba hacia abajo.

Cada unidad es relativamente de baja resolución. De este modo, el videojuego permitirá desplegar un gran número de unidades/estructuras en la pantalla al mismo tiempo.

Una de las prácticas que apuntan a acelerar el desarrollo de software, así como también disminuir la cantidad de defectos en el código, es la reutilización de componentes. Es por este motivo que se utilizará el motor de videojuegos jMonkeyEngine 3.0[4] en la implementación del prototipo.

jMonkeyEngine 3.0 es un motor que permite desarrollar videojuegos con gráficos en 3D en Java. Se rescribió completamente para acomodarse a los modernos estándares y mejores prácticas en el desarrollo de videojuegos.

Este motor se escogió para utilizarlo en el prototipo, principalmente, por las siguientes razones:

- ✓ Por preferencia de los desarrolladores del prototipo de videojuego, se escogió el lenguaje Java. El lenguaje es completamente viable para el desarrollo serio de videojuegos. Si bien siempre se ha criticado al lenguaje por su lento desempeño en comparación a lenguajes como C/C++, las últimas versiones han demostrado que la eficiencia es casi igual y, en algunos casos, es incluso más rápido (Davison, 2005)[5].
- ✓ Es gratuito. Se encuentra bajo la licencia New BSD[6].

[4] http://jmonkeyengine.com - Revisada última vez 18 de Abril de 2012
[5] http://keithlea.com/javabench/ - Revisada última vez 18 de Abril de 2012
[6] http://www.opensource.org/licenses/BSD-3-Clause - Revisada última vez 18 de Abril de 2012

- ✓ Implementa modernos estándares y las mejores prácticas en el desarrollo de videojuegos.
- ✓ Entre sus características se encuentran: uso de shaders, efectos de iluminación, motor de físicas, efectos de partículas, compatibilidad con varios formatos multimedia, integración con Nifty GUI para interfaces gráficas de usuario, efectos post-procesado (bloom, fog, cartoon), etc.

6.4 Entorno de Desarrollo Integrado (IDE)

Un entorno de desarrollo integrado es un sistema informático compuesto de una serie de herramientas habilitadas para la programación. En general posee un editor de código, un compilador, un intérprete y un depurador. Otras características incluidas pueden ser la posibilidad de detectar errores de sintaxis mientras se escribe el código, resaltado de sintaxis, automatización de tareas repetitivas e incluso ver la documentación de las estructuras de programación.

Para el desarrollo del videojuego será necesaria la utilización de dos entornos de desarrollo: Eclipse, por la familiaridad con los comandos, accesos directos y uso en general, y Kit de Desarrollo de Software jMonkey, por su integración completa con el motor de videojuegos jMonkeyEngine.

6.4.1 Eclipse IDE

Es una plataforma de desarrollo compuesta de marcos de trabajo extensibles, herramientas para la construcción, despliegue y manejo del software a través de todo su ciclo de vida.

Permite el uso de su tecnología de código abierto para el desarrollo de aplicaciones comerciales y sus servicios anidados. Esto es posible debido a que todos los proyectos de Eclipse están licenciados bajo la Licencia Pública de Eclipse[7].

El equipo de desarrollo del videojuego posee experiencia superior a dos años en el desarrollo de aplicaciones en el entorno Eclipse. Esta experiencia facilita el uso de dicho entorno para el manejo de proyectos, desplazamiento por clases y métodos, manejo de errores y sus soluciones, utilización de marcos de trabajos integrados, accesos directos y versionamiento.

6.4.2 Kit de Desarrollo de Software jMonkey (jMonkey SDK)

Es un entorno de desarrollo pre configurado diseñado específicamente para desarrolladores de videojuegos en Java. Contiene todas las librerías del motor de videojuegos

[7] http://www.eclipse.org/org/documents/epl-v10.php - Revisada última vez 18 de Abril de 2012

jMonkeyEngine y herramientas específicas para el diseño y desarrollo de elementos a incluirse en el videojuego final.

Está basado en la plataforma de desarrollo NetBeans por lo que tiene acceso a todas las herramientas de desarrollo ya existentes para dicha plataforma. Entre sus características principales contiene creación de proyectos y edición de código, despliegue de aplicaciones, manejo de errores y pruebas de código, importación, vista y conversión de modelos, exploración y composición de escenas y edición de terrenos y materiales.

Sigue un modelo de trabajo como muestra la Figura 6.9, con gran apoyo de la comunidad y siguiendo tres capas de desarrollo, la capa de contenido visual como materiales, texturas y modelos 3D, la capa de control, donde se codifican los flujos y acciones de los objetos junto con su comportamiento, y la capa del motor, encargada de reproducir el contenido visual y unirlo a la capa de control, teniendo en cuenta las características físicas, luminosidad, efectos especiales, entre otros.

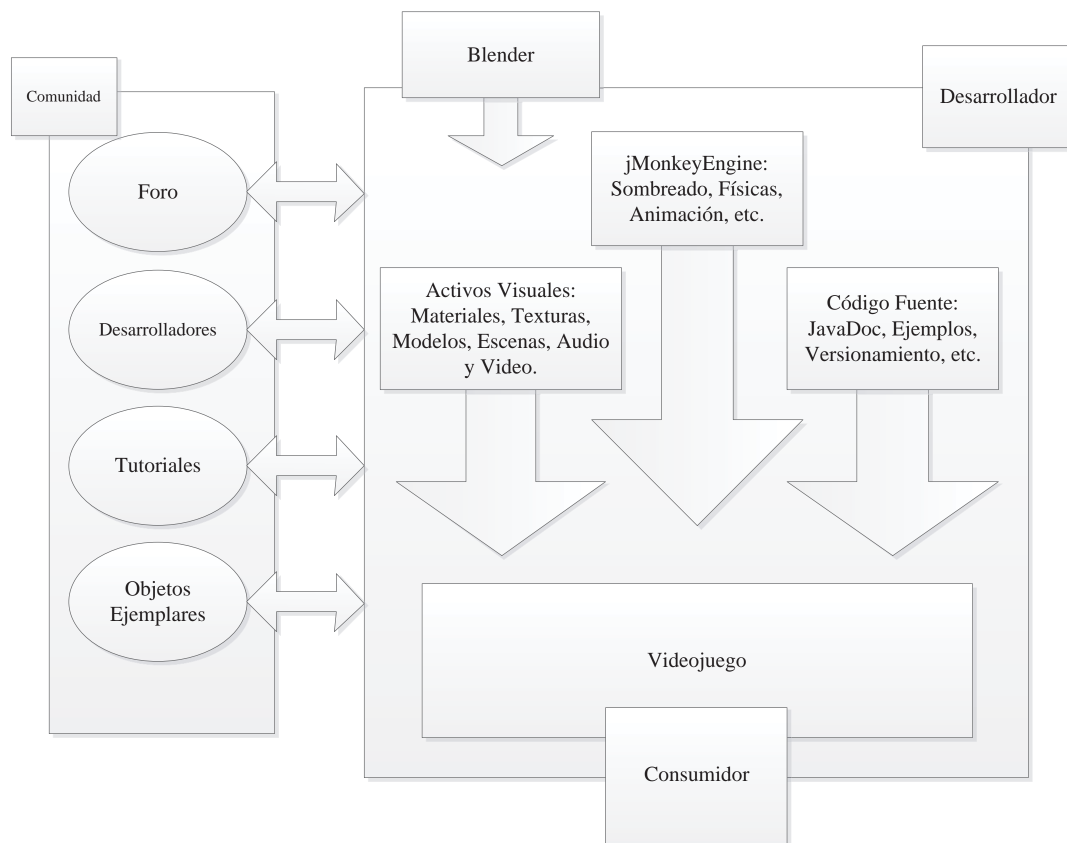

Figura 6.9 Modelo de trabajo de jMonkey.

7 Implementación del Videojuego

7.1 Datos Generales

- ✓ Nombre prototipo: *Evo*
- ✓ Nombre definitivo: *PegasusWars*
- ✓ Género: Estrategia en Tiempo Real (RTS)

7.2 Descripción General

El juego consiste en que dos rivales se enfrentan dentro de un escenario. Uno de los contrincantes es controlado por el computador y el otro por el jugador humano.

Cada competidor podrá construir una variedad de estructuras que le permitirán defenderse de ataques de las estructuras de su oponente, atacar a las estructuras de su oponente y recolectar recursos necesarios para construir más estructuras.

Al inicio de cada partida, los jugadores contarán con una estructura inicial denominada Centro de Control. Esta estructura es la más importante, ya que su principal objetivo es proveer de energía al resto de las estructuras construidas. Por lo tanto, si esta última es destruida, las estructuras restantes no recibirán energía. Si uno de los jugadores pierde esta unidad, pierde el juego.

Cada contrincante deberá definir la estrategia que le permita administrar sus recursos eficientemente para la construcción de estructuras en el terreno de juego. El juego transcurre en tiempo real, donde las decisiones se toman segundo a segundo, por lo que cada elección repercutirá en el desenlace de la partida. Cada jugador comenzará en igualdad de condiciones, por lo que la ventaja la obtendrá el jugador que adopte la estrategia más efectiva.

El juego será controlado principalmente a través del mouse, con apoyo de accesos rápidos del teclado. El terreno de juego tendrá una vista isométrica, que permitirá tener una percepción global de lo que ocurre en todo momento.

7.3 Reglas Específicas del Videojuego

A continuación se enuncian las reglas específicas que regirán el juego:

- ✓ Dos jugadores se enfrentan en un terreno de juego. Uno es controlado por el computador y el otro por el jugador humano.
- ✓ El terreno de juego se divide en bloques, de forma similar al tablero de ajedrez.
- ✓ Cada jugador cuenta con un *Centro de Control* como unidad inicial. El *Centro de Control* es la unidad que otorgará energía al resto de las unidades. Entrega un radio de construcción inicial de dos bloques. Además, genera recursos lentamente en

comparación a la unidad *Recolector de Recursos*. Si un jugador pierde esta unidad, pierde el juego.

- ✓ Existen tres unidades disponibles para la construcción: *Recolector de Recursos*, *Unidad de Ataque* y *Unidad de Defensa*. Para construir una unidad, el jugador deberá pagar su costo en recursos.
- ✓ El *Recolector de Recursos* cumple con dos funciones: primero, es el encargado de transmitir energía desde el *Centro de Control* hacia otros sectores del terreno de juego. Esto permite construir otras unidades dentro de un radio de dos bloques y, de esta forma, cubrir un área mayor de terreno. En segundo lugar, extrae recursos para cubrir el costo de construcción de otras unidades.
- ✓ La *Unidad de Ataque* es la encargada de atacar a las unidades del enemigo. El ataque es automático, se ejecuta constantemente en un intervalo de tiempo y tiene un radio de alcance máximo de dos bloques.
- ✓ La *Unidad de Defensa* tiene una resistencia superior a las demás unidades. Permite defender a otras unidades posicionadas dentro de un radio de un bloque. Si alguna de las unidades posicionadas dentro del radio de alcance es atacada, el daño lo recibirá la *Unidad de Defensa*.
- ✓ Para construir un *Recolector de Recursos*, una *Unidad de Ataque* o una *Unidad de Defensa* deben cumplirse dos requisitos: contar con los recursos que la unidad requiere para su construcción y la ubicación destino debe ser un bloque habilitado en términos de energía por el *Centro de Control*, o bien, por un *Recolector de Recursos*.
- ✓ Si alguna de las estructuras deja de recibir energía proveniente del *Centro de Control* o de algún *Recolector de Recursos*, no es destruida, pero queda automáticamente deshabilitada hasta que se restablezca el suministro.

7.4 Arquitectura de la Implementación

De forma general, la arquitectura de la implementación del prototipo puede representarse a través del esquema de la Figura 7.1.

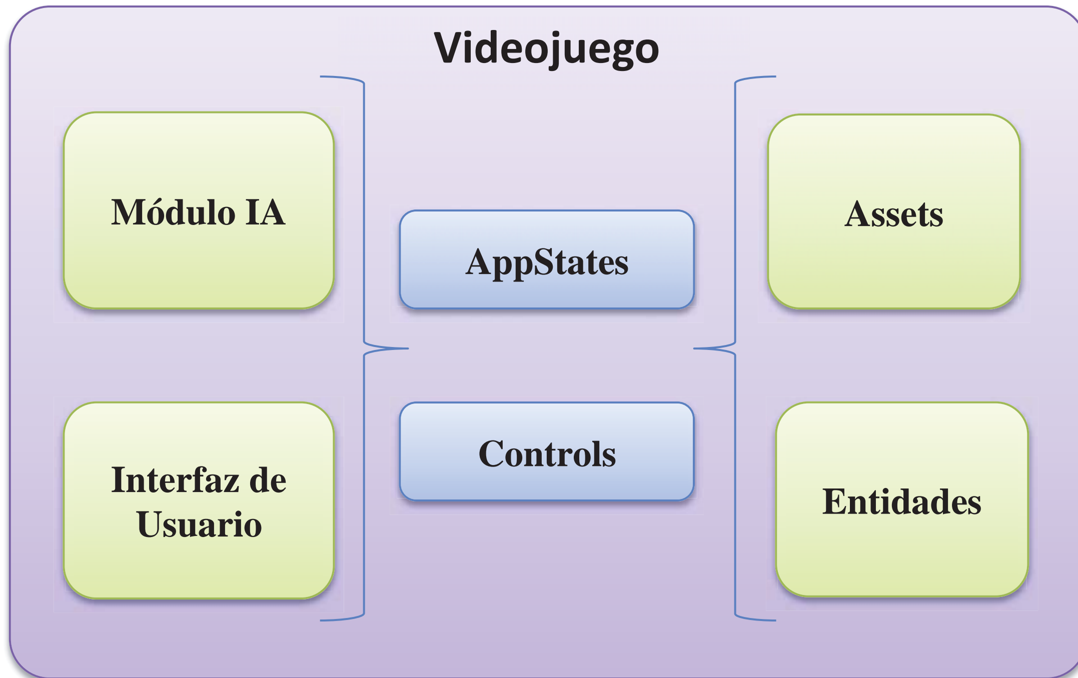

Figura 7.1 Esquema de arquitectura de la implementación.

Los componentes del esquema se detallan a continuación:

AppStates

En jMonkey se pueden representar los estados de la aplicación a través de AppStates. Para esto se crea una clase que implementa la interfaz *com.jme3.app.state.AppState*, que permite controlar la lógica global del videojuego.

Cada AppState permite definir lo que sucede cuando ocurre alguno de los siguientes eventos: el AppState es inicializado, el AppState es activado, el AppState es desactivado y el AppState es removido.

Un AppState puede ser activado en cualquier momento y, con ello, cargar un conjunto de modelos 3D, cargar sonidos, establecer controles de juego determinados, ubicar luces en el escenario, iniciar animaciones, etc. Todo esto en un paso, que consiste en la inicialización del estado. Además, pueden inicializarse más de un estado simultáneamente, permitiendo interacción entre varios estados potenciando la característica modular del videojuego.

A continuación se mencionan y describen los AppSates que se encuentran en la implementación del prototipo.

BattleSceneAppState

Este AppState se encarga de inicializar el cursor del mouse, configurar la cámara, cargar el escenario de batalla, crear las fuentes de luz, crear los mapeos de entrada (mouse y teclado), inicializar la matriz de estados de bloques del escenario, inicializar el cielo de fondo y cargar efectos gráficos adicionales (efecto bloom).

Tanto el videojuego como el módulo de inteligencia artificial necesitan manejar un estado del escenario de batalla. Para reducir el acoplamiento entre ambos componentes, cada uno maneja una forma independiente de representar dicho estado.

El AppState BattleSceneAppState es el encargado de mantener el estado del escenario de batalla para el videojuego. Para ello mantiene una propiedad de tipo *Map<PlayerType, BattleSceneBlockState[][]>*, un mapa que tiene como clave el tipo de jugador (*PlayerType*), que indica si se trata del jugador humano o del jugador controlado por el computador y, como valor asociado, una matriz de objetos de tipo *BattleSceneBlockState*.

PlayerType es una enumeración con dos posibles elementos: *HUMAN_PLAYER* y *COMPUTER_PLAYER*. Por lo tanto el mapa *Map<PlayerType, BattleSceneBlockState[][]>* mantendrá una matriz *BattleSceneBlockState[][]* por cada jugador.

Como se mencionó, cada elemento de la matriz será un objeto del tipo *BattleSceneBlockState*. Esta clase cuenta con las siguientes propiedades definidas para representar el estado de un bloque:

- Una lista *List<BattleSceneBlockState> poweredBy* que almacena una referencia a todos los bloques que proveen energía a este bloque, si corresponde.
- Una lista *List<BattleSceneBlockState> givesPowerTo* que almacena una referencia a todos los bloques a los que este bloque entrega energía, si corresponde.
- Una lista *List<BattleSceneBlockState> protectedBy* que almacena una referencia a todos los bloques que entregan protección a este bloque, si corresponde.
- Una lista *List<BattleSceneBlockState> protectsTo* que almacena una referencia a todos los bloques a los que este bloque brinda protección, si corresponde.
- Una propiedad *Node unitNodeInBlock* que almacena una referencia a la unidad que ocupa este bloque, si corresponde.

Por otro lado, cada uno de estos objetos de tipo *BattleSceneBlockState* que componen la matriz de estados de bloques cuenta con los siguientes métodos relevantes:

- *boolean isBlockUsed()*: Retorna *true* si en el bloque se encuentra posicionada una unidad, de lo contrario retorna *false*.
- *boolean isBlockPowered()*: Retorna *true* si el bloque se encuentra energizado, de lo contrario retorna *false*.

- *boolean isBlockProtected()*: Retorna *true* si el bloque se encuentra protegido, de lo contrario retorna *false*.

Volviendo a la especificación del AppState BattleSceneAppState, los métodos que incluye esta clase son:

public BattleSceneAppState(): Constructor vacío de la clase.

private void bloomEffect(): Crea el efecto bloom sobre el escenario. Este efecto consiste en agregar un resplandor sobre los objetos que se indiquen dentro del escenario.

private void createSky(): Crea efecto de espacio con estrellas que rodean al escenario.

private void lightSources(): Crea las fuentes de luz que iluminan el escenario.

private void setCamera(): Inicializa la cámara.

private void loadTerrain(): Carga y dibuja el escenario de batalla en la pantalla.

private void initKeyMappings(): Inicializa las entradas por mouse y teclados correspondientes a este estado: rotar la cámara alrededor del escenario con teclas *A* y *D*, seleccionar objetos con *Click Izquierdo* y, con tecla *Escape* cerrar menús y salir del videojuego.

public void seleccionarUnidad(Node unitNode): Establece como seleccionada la unidad recibida como parámetro.

public void deseleccionarUnidad(): Anula la selección de unidad.

public Node getBattleSceneNode(): Retorna el nodo raíz que contiene al resto de nodos del escenario.

public void placeUnit(Node unitNode, int x, int y, boolean updateGameState): Posiciona una unidad en el escenario de batalla, en la posición (x, y) y, si el parámetro updateGameState es *true*, se actualiza el estado de la matriz de bloques de estado del escenario. Este último parámetro es útil cuando se requiere posicionar una unidad no definitiva en el escenario, como cuando se está seleccionando posición definitiva de unidad tras presionar el botón de construcción de una unidad específica.

public void removeUnit(Node unitNode, boolean updateGameState): Quita una unidad del escenario de batalla. A partir del parámetro updateGameState se determina si es necesario actualizar el estado de la matriz de bloques de estado del escenario.

public void destroyUnit(Node unitNode): Destruye una unidad del escenario de batalla. Este método muestra el efecto de destrucción sobre la unidad destruida. Además, este método siempre actualiza el estado de la matriz de bloques de estado del escenario.

public Node getUnitInPosition(PlayerType playerType, int x, int y): Retorna la unidad ubicada en la posición especificada para un jugador en particular (computador o humano). Si no existe unidad en la posición indicada, se retorna *null*.

public Vector2f getGridPositionFromPoint(Vector3f terrainPoint): Retorna la posición ocupada dentro de la matriz del escenario a partir de un punto especificado por el parámetro terrainPoint. Cabe destacar que el tipo *Vector3f* representa un vector de dimensión tres. El tipo *Vector2f* representa un vector de dimensión dos. Por lo tanto, se está convirtiendo un punto en tres dimensiones en una representación de dos dimensiones, correspondiente a la matriz del escenario de batalla.

private Vector3f getTerrainOrigin(): Retorna la posición en el espacio de tres dimensiones en la que comienza el escenario de batalla (esquina superior izquierda). Este método es de uso interno para realizar cálculos de posicionamiento.

public void showTerrainGrid(boolean show): Muestra u oculta la malla desplegada sobre el escenario para resaltar separación entre bloques. El parámetro show indica si se debe mostrar u ocultar la malla.

public boolean isBuildAllowed(UnitType unitType, Vector2f gridPosition, PlayerType playerType): Determina si es posible construir un tipo de unidad en la posición indicada para un tipo de jugador.

public Node getSelectedNode(): Retorna el nodo que contiene a una unidad seleccionada por el jugador humano.

public void setSelectedNode(Node selectedNode): Establece como seleccionado al nodo que contiene a la unidad entregada como parámetro.

public Node getCameraTargeNode(): Retorna el nodo utilizado como referencia para dirigir la posición de la cámara.

public Map<PlayerType, BattleSceneBlockState[][]> getBlocksStates(): Retorna un mapa que contiene las matrices de estado de bloques para el jugador humano y el jugador controlado por el computador.

public void attackTarget(BattleSceneBlockState attackerBlockState, BattleSceneBlockState targetBlockState, int attackPower): Ejecuta un ataque desde la unidad que se encuentra en el bloque del atacante hacia la unidad que se encuentra en el bloque objetivo. El parámetro attackPower especifica el poder de ataque que determinará el número de puntos de vida que perderá quien reciba el ataque.

private void damageUnit(Node unit, int attackPower): Aplica el daño a una unidad especificada.

public void addUnitControl(Node node, int x, int y): Asigna el control que corresponda a la unidad recibida como parámetro. Los parámetros x e y representan la posición de la unidad dentro del escenario de batalla. El uso de controles se tratará más adelante.

public SimpleAplicacion getApp(): Retorna referencia a objeto que inicializa la aplicación y contiene métodos de uso general para controlar el comportamiento de la aplicación.

BattleSceneScreenController

Este es un AppState que representa una capa intermedia entre la interfaz de usuario y los demás AppStates. Captura las acciones del jugador a través de la interfaz de usuario e interactúa con el resto de los AppStates. Además, se encarga de mantener informado al jugador de los cambios de estado del videojuego. Antes de inicializar este AppState se requiere que existan los AppStates de tipo BattleSceneAppState y PlayersAppState inicializados, es decir, BattleSceneScreenController utiliza métodos existentes en BattleSceneAppState y PlayersAppState.

Los métodos que incluye esta clase son:

public BattleSceneScreenController(): Constructor vacío de la clase.

private void updateAvailableResources(): Actualiza el estado de los recursos disponibles, para el jugador humano, que se muestra por pantalla. Utiliza el método *getPlayerResources(PlayerType playerType)* del AppState PlayersAppState para obtener los recursos del jugador.

private void createUnitBuildPlaceControls(): Crea los controles que se muestran por pantalla mientras el jugador humano selecciona la posición definitiva donde se construirá la unidad. Por ejemplo, se despliega el botón Cancelar que anula la construcción de la unidad.

private void createControlPanel(): Construye el panel de control que se despliega al jugador humano. Este panel es el contenedor de los botones para construir y destruir unidades.

private void addButtonToControlPanel(String imagePath, String imageId, String methodToCall): Agrega un botón al panel de control. Por ejemplo, puede agregarse un botón para destruir una unidad. Entre los parámetros, imagePath es la ruta a la imagen que se mostrará como ícono, imageId es un identificador que se asigna a la imagen y methodToCall es el nombre del método al que se llamará al hacer click sobre el botón. El método debe existir dentro de este AppState.

private void updateControlPanelButtonState(): Actualiza el estado del panel de control. Esto permite mostrar un botón como deshabilitado si no es posible realizar la acción. Por ejemplo, si no se cuenta con los recursos suficientes para construir una unidad, debe deshabilitarse el botón.

public void showUnitBuildScreen(UnitType unitType): Muestra pantalla de selección de ubicación de unidad a construir definitiva. Este método no crea los controles como el método *createUnitBuildPlaceControls()*, sólo los hace visibles para el jugador.

public void hideUnitBuildScreen(): Oculta la pantalla de selección de ubicación de unidad a construir.

public void addScreenMessage(String message, GameMessageSeverity severity): Agrega un mensaje a la pantalla. El parámetro message especifica el mensaje y severity es una enumeración que determina la severidad del mensaje. Es decir, la severidad puede ser *INFO* para un mensaje de información, *WARN* para un mensaje de advertencia y *ERROR* para un mensaje de error. La severidad también determina el color del texto del mensaje. Los mensajes de información se muestran en color verde, los de advertencia en color amarillo y los de error en color rojo.

private void clearGameMessage(): Limpia los mensajes mostrados por pantalla. Este método es llamado automáticamente transcurridos cinco segundos desde que se agregó un mensaje con el método *addScreenMessage(String message, GameMessageSeverity severity)*.

public void buildCollectorUnit(): Este método es llamado por el botón de construcción de Unidad Recolectora. Inicializa el proceso de construcción de dicha unidad.

public void buildAttackUnit(): Este método es llamado por el botón de construcción de Unidad de Ataque. Inicializa el proceso de construcción de dicha unidad.

public void buildDefenseUnit(): Este método es llamado por el botón de construcción de Unidad de Defensa. Inicializa el proceso de construcción de dicha unidad.

public void destroySelectedUnit(): Destruye una unidad seleccionada. Este método depende del método *destroyUnit(UnitNode unitNode)* del AppState BattleSceneAppState. Sólo se pueden destruir unidades pertenecientes al jugador siempre y cuando no sea la el Centro de Control.

public void showControlPanel(): Hace visible el panel de control.

public void hideControlPanel(): Oculta el panel de control.

public void showUnitPanel(Node unitNode): Hace visible el panel de unidad seleccionada.

public void hideUnitPanel(): Oculta el panel de unidad seleccionada.

private void addDestroyUnitButton(UnitProperties unitProperties): Agrega el botón de destrucción de unidad al panel de control. Esto ocurre sólo si la unidad seleccionada pertenece al jugador y no es el centro de control.

private void removeDestroyUnitButton(): Quita el botón de destrucción de unidad del panel de control.

private void setUnitAvatar(UnitType unitType): Establece la imagen de avatar de la unidad seleccionada.

private void setUnitDescription(UnitProperties unitProperties): Establece la descripción de la unidad seleccionada.

UnitBuildAppState

Este AppState representa el estado de construcción de unidad para el jugador. Esto ocurre cuando el jugador decide construir una unidad y debe seleccionar la ubicación definitiva. Antes de inicializar este AppState se requiere que exista un AppState de tipo BattleSceneAppState inicializado, es decir, UnitBuildAppState utiliza métodos existentes en BattleSceneAppState.

Los métodos que incluye esta clase son:

public UnitBuild(UnitType unitType): Único constructor de la clase. Requiere como parámetro el tipo de unidad a construir.

private void initKeyMappings(): Inicializa las entradas por mouse y teclados correspondientes a este estado: *Click Izquierdo* del mouse para confirmar construcción en una posición específica del escenario de batalla y *Escape* para cancelar construcción de la unidad.

private void placeGhostUnit(): Ubica una unidad "fantasma" en el escenario de batalla, utilizando el método *placeUnit(Node unitNode, int x, int y, boolean updateGameState)* del AppState BattleSceneAppState, pero con el parámetro updateGameState en *false*. Esto porque como no se ha confirmado la construcción, no se debe actualizar el estado de la matriz de estado de bloques del escenario. La posición de la unidad está determinada por la posición del cursor del mouse, es decir, la unidad sigue la posición del cursor.

public UnitType getUnitTypeToBuild(): Retorna el tipo de unidad a construir. El tipo se especificó al construir el objeto de este AppState.

public Node getUnitToBuild(): Retorna el nodo que contiene a la unidad que se está construyendo.

public void confirmUnitBuild(int x, int y): Confirma la construcción de la unidad en el lugar especificado. Esto implica que la unidad deberá fijarse en la posición indicada, actualizar el estado de la matriz de estado de bloques del escenario de batalla y terminar la ejecución de este AppState UnitBuildAppState.

PlayersAppState

En este AppState se inicializan las propiedades tanto del jugador humano, como del jugador controlado por el computador. Antes de inicializar este AppState se requiere que exista un AppState de tipo BattleSceneAppState inicializado, es decir, PlayersAppState utiliza métodos existentes en BattleSceneAppState.

Los métodos que incluye esta clase son:

private void initHumanPlayer(): Este método inicializa todo lo relacionado al jugador humano. Eso contempla establecer los recursos iniciales, el color del jugador y la construcción de la unidad inicial de ambos jugadores, el Centro de Control.

private void initComputerPlayer(): De forma análoga al método *initHumanPlayer()*, se inicializa todo lo relacionado al jugador controlado por el computador: recursos iniciales, color de jugador y la construcción de la unidad inicial, el Centro de Control.

public int getPlayerResources(PlayerType playerType): Retorna la cantidad de recursos que tiene el jugador especificado por el parámetro playerType en el momento en que se llama a este método.

public void spendPlayerResources(PlyerType playerType, int resourcesAmount): Permite disminuir los recursos del jugador especificado por el parámetro playerType en la cantidad indicada por el parámetro resourcesAmount. Este método es llamado cuando un jugador confirma la construcción de alguna unidad.

public void addResourcesToPlayer(PlayerType playerType, int resources): Permite incrementar los recursos con los que cuenta el jugador especificado por el parámetro playerType en la cantidad indicada por el parámetro resources. Este método es llamado cuando las unidades Centro de Control y Recolectora generan recursos.

public boolean isUnitAvailableToBuild(PlayerType playerType, UnitType unitType): Determina si el jugador especificado por el parámetro playerType puede construir una unidad del tipo indicado por el parámetro unitType. Para esto se verifica si la cantidad de recursos con la que cuenta el jugador son suficientes para cubrir el costo de la unidad.

public Player getHumanPlayer(): Retorna la instancia de la clase *Player* que contiene los datos del jugador humano.

public Player getComputerPlayer(): Retorna la instancia de la clase *Player* que contiene los datos del jugador controlado por el computador.

AIAppState

En este AppState se inicializa el módulo de inteligencia artificial y se encarga de integrarlo al juego a través de la implementación de la interfaz *JuegoInterface* definida por dicho módulo. Además, ejecuta la secuencia de acciones que entrega el módulo de inteligencia artificial. Antes de inicializar este AppState se requiere que existan los AppStates de tipo BattleSceneAppState y PlayersAppState inicializados, es decir, AIAppState utiliza métodos existentes en BattleSceneAppState y PlayersAppState.

Los métodos que incluye esta clase son:

private void generateRandomSecondsBetweenActions(): Calcula el número de segundos que toma la ejecución de una acción. El número es calculado entre acción y acción, generando la ilusión de que el computador está pensando. El número de segundos varía aleatoriamente entre un mínimo y un máximo, especificado a través de las constantes de esta clase *MIN_SECONDS_BETWEEN_ACTIONS* y *MAX_SECONDS_BETWEEN_ACTIONS*, respectivamente.

private void wait(float tpf): Cuando el módulo de inteligencia artificial entrega una acción de espera, este método efectúa una pausa antes de realizar la siguiente acción. Esta espera no es aleatoria como en el caso de los segundos entre acciones generada por el método *generateRandomSecondsBetweenActions()* de este mismo AppState. El tiempo de espera se especifica en la acción entregada por el módulo de inteligencia artificial. El parámetro tpf (time per frame) indica el tiempo que transcurre entre cuadro y cuadro. Permite, entre otras cosas, realizar el cálculo del tiempo transcurrido sin importar la velocidad de ejecución.

private void destroy(float tpf): Cuando el módulo de inteligencia artificial entrega una acción que contempla la destrucción de una unidad, este método ejecuta dicha destrucción. El tiempo de espera antes de ejecutar la acción es obtenido a partir del tiempo aleatorio generado por el método *generateRandomSecondsBetweenActions()* de este mismo AppState. El parámetro tpf (time per frame) indica el tiempo que transcurre entre cuadro y cuadro. Permite, entre otras cosas, realizar el cálculo del tiempo transcurrido sin importar la velocidad de ejecución.

private void build(float tpf): Cuando el módulo de inteligencia artificial entrega una acción que contempla la construcción de una unidad, este método ejecuta dicha construcción. El tiempo de espera antes de ejecutar la acción es obtenido a partir del tiempo aleatorio generado por el método *generateRandomSecondsBetweenActions()* de este mismo AppState. El parámetro tpf (time per frame) indica el tiempo que transcurre entre cuadro y cuadro. Permite, entre otras cosas, realizar el cálculo del tiempo transcurrido sin importar la velocidad de ejecución.

private void removeCompletedAction(Accion accion): Quita la acción, especificada en el parámetro acción, de la lista de acciones que entrega el módulo de inteligencia artificial. Este método es llamado tras ejecutar correctamente una acción.

private void generateCurrentGameState(): Genera el estado actual del escenario de batalla. El estado será utilizado por el método implementado de la interfaz *JuegoInterface* declarado como *EstadoJuego obtenerEstadoJuegoActual()*.

public EstadoJuego obtenerEstadoJuegoActual(): Este método es la implementación del único método especificado por la interfaz del módulo de inteligencia artificial *JuegoInterface*. El método retorna el estado generado por el método *generateCurrentGameState()* de este mismo AppState. Mediante este método, el módulo de inteligencia artificial puede conocer el estado actual del escenario en un instante de tiempo.

private Unidad UnitTypeToUnidad(UnitType unitType): Convierte un objeto de tipo *UnitType*, utilizado por el videojuego, a un objeto de tipo *Unidad*, utilizado por el módulo de inteligencia artificial.

private UnitType TipoEstructuraToUnitType(TipoEstructura tipoEstructura): Convierte un objeto de tipo *TipoEstructura*, utilizado por el módulo de inteligencia artificial, a un objeto de tipo *UnitType*, utilizado por el videojuego.

Controles

Un control es una clase que implementa a la interfaz *com.jme3.scene.control.Control*. Permite controlar el comportamiento de alguna entidad en particular (como una unidad de ataque) que requiera interactuar con otras entidades.

En el prototipo de videojuego, existen los siguientes controles definidos:

AttackUnitControl

Este control es el encargado de controlar el comportamiento de una unidad de ataque una vez construida sobre el escenario de batalla.

Las funcionalidades que otorga el control a la entidad a la que se le asigna son:

- Lleva el registro de los puntos de vida de la unidad.
- Quita puntos de vida a la unidad si se recibe un ataque.
- Si los puntos de vida de la unidad llegan a cero, destruye la unidad.
- Realiza ataques a unidades dentro del alcance a un intervalo de tiempo constante, siempre y cuando la unidad reciba energía de la red de suministro energético proveniente desde el centro de control o a través de unidades recolectoras.
- Si la unidad no tiene un objetivo para atacar establecido, realiza un escaneo antes de realizar un ataque. Si la unidad que tenía como objetivo es destruida, comienza nuevamente el proceso de escaneo hasta que encuentra otro objetivo dentro del alcance.

BaseUnitControl

Este control es el encargado de controlar el comportamiento del centro de control una vez construido sobre el escenario de batalla.

Las funcionalidades que otorga el control a la entidad a la que se le asigna son:

- Lleva el registro de los puntos de vida de la unidad.
- Quita puntos de vida a la unidad si se recibe un ataque.
- Si los puntos de vida de la unidad llegan a cero, destruye la unidad.

- Entrega recursos al jugador al que le pertenece la unidad a un intervalo de tiempo constante. La cantidad de recursos que son generados siempre serán menores a los que entrega una unidad recolectora.

CollectorUnitControl

Este control es el encargado de controlar el comportamiento de una unidad recolectora una vez construida sobre el escenario de batalla.

Las funcionalidades que otorga el control a la entidad a la que se le asigna son:

- Lleva el registro de los puntos de vida de la unidad.
- Quita puntos de vida a la unidad si se recibe un ataque.
- Si los puntos de vida de la unidad llegan a cero, destruye la unidad.
- Entrega recursos al jugador al que le pertenece la unidad a un intervalo de tiempo constante. La cantidad de recursos que son generados siempre serán mayores a los que entrega el centro de control.

DefenseUnitControl

Este control es el encargado de controlar el comportamiento de una unidad de defensa una vez construida sobre el escenario de batalla.

Las funcionalidades que otorga el control a la entidad a la que se le asigna son:

- Lleva el registro de los puntos de vida de la unidad.
- Quita puntos de vida a la unidad si se recibe un ataque.
- Si los puntos de vida de la unidad llegan a cero, destruye la unidad.

SpatialRemoverControl

Este control es el encargado de quitar de la escena un spatial transcurridos un número especificado de segundos. Esto permite, por ejemplo, crear un efecto de partículas (como una explosión, que es un spatial) y programarlo para que sea quitado de la escena después de un determinado tiempo.

Las funcionalidades que otorga el control a la entidad a la que se le asigna son:

- Una vez que se cumple el tiempo definido al crear la instancia del control, la entidad es removida de la escena.

Cada uno de los AppStates y Controles puede interactuar otros componentes del videojuego. Estos componentes son: AppStates, controles, entidades, assets, interfaz de usuario y el módulo de inteligencia artificial.

Entidades

En jMonkey una entidad (también conocida como spatial) puede ser una geometría o un nodo. Las entidades se organizan en una jerarquía de árbol. Toda la escena dibujada en pantalla nace a partir de un único nodo raíz denominado *rootNode*. Luego, cada spatial (nodo o geometría) es hijo de sólo un padre. Pero sólo los nodos pueden agrupar a varios spatials hijos. Por lo tanto, una geometría representaría un nodo hoja que no puede tener hijos.

Otra forma de entender los spatials es en términos de visibilidad. Una geometría representa un elemento visible de la escena. Por el contrario, un nodo no es visible, pero si puede agrupar uno a más elementos visibles.

Desde el punto de vista del propósito de los spatials, una geometría representa cómo se ve una entidad. El nodo apunta a especificar como se agrupan. Es buena práctica contar con un nodo que agrupe a una o más geometrías, y luego aplicar transformaciones sobre el nodo. Cualquier transformación que se aplique sobre un nodo, afectará a todos sus hijos. Un caso práctico es el siguiente: se tiene un personaje y un arma, cada uno en una geometría distinta. Es muchísimo más sencillo manipular al nodo que agrupe a ambas geometrías, que manipular cada una por separado, sobre todo si se trata de transformaciones geométricas como traslaciones o rotaciones.

A continuación se describen las principales entidades presentes en la implementación del prototipo del videojuego.

Centro de Control

Esta entidad permite representar a la Unidad Centro de Control en el escenario de batalla. La Figura 7.2 muestra el modelo que representa a dicha unidad.

Figura 7.2 Modelo 3D de la Unidad Centro de Control.

Unidad Recolectora

Esta entidad permite representar a la Unidad Recolectora en el escenario de batalla. La Figura 7.3 muestra el modelo que representa a dicha unidad.

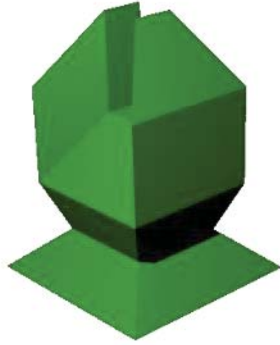

Figura 7.3 Modelo 3D de la Unidad Recolectora.

Unidad de Ataque

Esta entidad permite representar a la Unidad de Ataque en el escenario de batalla. La Figura 7.4 muestra el modelo que representa a dicha unidad.

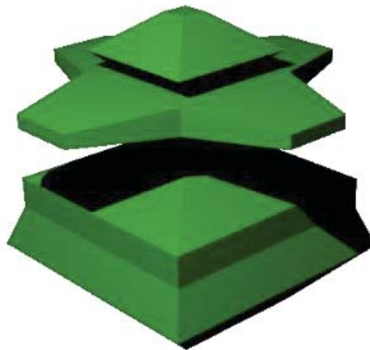

Figura 7.4 Modelo 3D de la Unidad de Ataque.

Unidad de Defensa

Esta entidad permite representar a la Unidad Centro de Defensa en el escenario de batalla. La Figura 7.5 muestra el modelo que representa a dicha unidad.

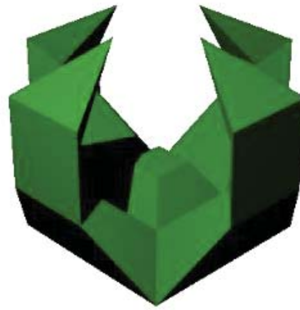

Figura 7.5 Modelo 3D de la Unidad de Defensa.

Escenario de Batalla

Esta entidad permite representar el escenario de batalla. La Figura 7.6 muestra el modelo que representa dicho escenario. Además, todas las entidades que se ubiquen en este escenario de batalla serán descendientes del nodo padre del escenario, de modo que cualquier transformación geométrica aplicada a dicho nodo, sea heredada tanto por el escenario como por el resto de las entidades.

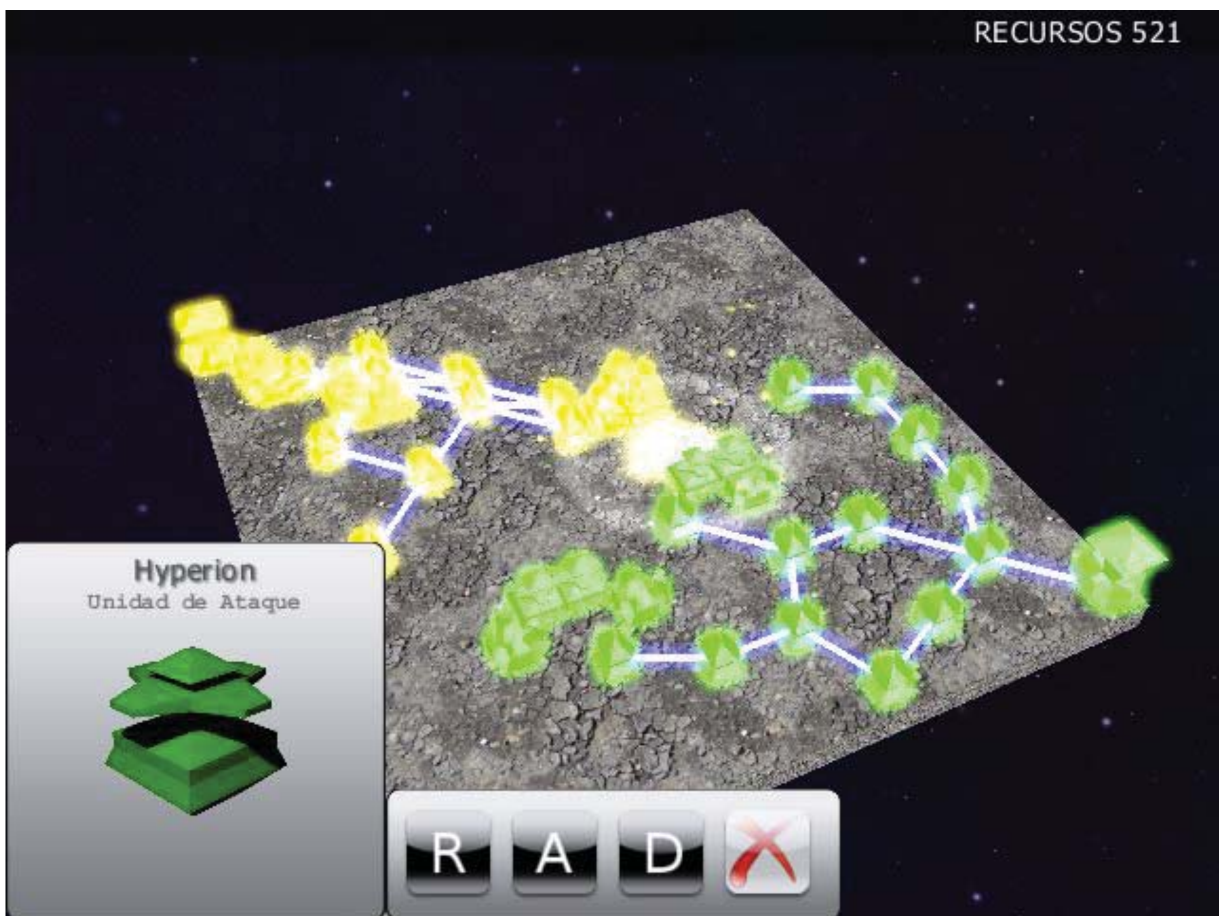

Figura 7.6 Representación del escenario de batalla.

Assets

Los assets son todos los recursos con los que cuenta el videojuego. Estos recursos pueden ser elementos de la interfaz de usuario, materiales, modelos, escenas, sonidos, texturas, etc. Se puede pensar en ellos como un repositorio de recursos.

Los assets pueden ser accedidos en cualquier momento desde alguno de los AppStates que se encuentran inicializados.

La documentación de jMonkey recomienda seguir el siguiente patrón de jerarquía de directorios para los assets:

- ✓ assets/Interface
- ✓ assets/MatDefs
- ✓ assets/Materials
- ✓ assets/Models
- ✓ assets/Scenes
- ✓ assets/Shaders
- ✓ assets/Sounds
- ✓ assets/Textures

En la implementación del prototipo de videojuego se cuenta con el siguiente conjunto y estructura de assets externos a los que entrega jMonkey:

- ❖ **assets/Interface/icons/**
  - **attack-unit-a.png**
    Ícono para construir Unidad de Ataque habilitado.
  - **attack-unit-b.png**
    Ícono para construir Unidad de Ataque deshabilitado.
  - **collector-unit-a.png**
    Ícono para construir Unidad Recolectora habilitado.
  - **collector-unit-b.png**
    Ícono para construir Unidad Recolectora deshabilitado.
  - **defense-unit-a.png**
    Ícono para construir Unidad de Defensa habilitado.
  - **defense-unit-b.png**
    Ícono para construir Unidad de Defensa deshabilitado.
  - **destroy-unit-a.png**
    Ícono para destruir unidad.

- ❖ **Assets/Interface/**
  - **screen.xml**
    Archivo XML que define la estructura general de las pantallas de interfaz de usuario creadas con Nifty GUI.

- **assets/Interface/portraits/**
  - **attack-unit.png**
    Avatar para Unidad de Ataque.
  - **base-unit.png**
    Avatar para Unidad Centro de Control.
  - **collector-unit.png**
    Avatar para Unidad Recolectora.
  - **defense-unit.png**
    Avatar para Unidad de Defensa.

- **assets/Textures/**
  - **ground01.jpg**
    Textura del suelo del escenario de batalla.

- **assets/Textures/Sky/Galaxy/**
  - **galaxy+x.png**
    Textura de parte Este de galaxia de fondo.
  - **galaxy-x.png**
    Textura de parte Oeste de galaxia de fondo.
  - **galaxy+y.png**
    Textura de parte Superior de galaxia de fondo.
  - **galaxy-y.png**
    Textura de parte Inferior de galaxia de fondo.
  - **galaxy+z.png**
    Textura de parte Sur de galaxia de fondo.
  - **galaxy-z.png**
    Textura de parte Norte de galaxia de fondo.

Interfaz de Usuario

Para permitir la interacción del jugador humano con los elementos de la pantalla es necesario construir una interfaz de usuario. Para esto se utiliza la librería Nifty GUI, que permite la construcción de interfaces de usuario interactivas para videojuegos o aplicaciones similares.

La definición de una interfaz en Nifty GUI puede realizarse de dos formas. La primera consiste en definir la estructura general de la interfaz utilizando documentos XML. La segunda forma se basa en construir programáticamente desde Java en tiempo de ejecución.

En general, todo elemento puede definirse tanto desde el documento XML como programáticamente desde Java. Sin embargo, es una buena práctica definir el esquema general en el documento XML y controlarlos dinámicamente desde Java.

Una interfaz de Nifty se compone de los siguientes elementos:

- Una interfaz Nifty contiene una o más *Pantallas*.
    - Sólo una *Pantalla* es visible a la vez.
    - Cada *Pantalla* es controlada por una clase controladora en Java.
- Una *Pantalla* contiene una o más *Capas*.
    - Las *Capas* son contenedores que imponen alineación sobre su contenido (vertical, horizontal o centrado).
    - Las *Capas* se pueden superponer (orden-z), pero no se pueden anidar.
- Una *Capa* contiene *Paneles*.
    - Los *Paneles* son contenedores que imponen alineación sobre su contenido (vertical, horizontal o centrado).
    - Los *Paneles* pueden anidarse, pero no se pueden superponer.
- Un *Panel* contiene imágenes, texto o controles (botones, listas, checkboxes, componentes de ingreso de texto, entre otros).

Módulo de Inteligencia Artificial

El módulo de inteligencia artificial está orientado a controlar el comportamiento del jugador controlado por el computador. Esto implica que debe conocer en todo momento el estado del terreno de juego para poder entregar soluciones coherentes en respuesta a las acciones del jugador humano.

Este módulo se integra como una librería completamente independiente desde el punto de vista de la implementación del prototipo. Se define una interfaz que establece el contrato entre Videojuego-Módulo IA, de modo que se disminuye el acoplamiento. Así, las refactorizaciones de código no afectan a la interacción entre los componentes, siempre y cuando no se modifiquen las interfaces.

La interfaz declara sólo un método: *EstadoJuego obtenerEstadoJuegoActual()*. A través de este método, el módulo puede conocer el estado del escenario de batalla en un instante de tiempo. El método debe ser implementado por el videojuego para permitir la integración.

Por lo tanto, desde el punto de vista del módulo de inteligencia artificial, la forma que tiene de interactuar con el videojuego es mediante llamadas al único método *obtenerEstadoJuegoActual()* definido por la interfaz *rts.JuegoInterface*. El objeto de la clase que implementa dicha interfaz es entregado por el videojuego al momento de construir la instancia de la clase *rts.controller.PlayerAI*. Esta es la clase principal del módulo de inteligencia artificial, y es la que expone el método *List<Accion> listarAccionesAI()* al videojuego. Es de este modo que el videojuego recibe la lista de acciones a ejecutar.

En resumen, los pasos a seguir para integrar el módulo de inteligencia artificial al videojuego se enuncian a continuación:

1. Crear clase que implemente la interfaz *rts.JuegoInterface* definida por el módulo de inteligencia artificial.

2. Iniciar el módulo de inteligencia artificial creando una instancia de la clase *rts.controller.PlayerAI*. Esta clase cuenta con un único constructor: *PlayerAI(JuegoInterface juegoInterface)*. Al constructor debe entregársele una instancia de la clase que implementa *rts.JuegoInterface*.

En este punto, el módulo de inteligencia artificial ya se encuentra listo para funcionar. En adelante, será el videojuego quien llame al método *List<Accion> listarAccionesAI()* para conocer las acciones de debe ejecutar y, dentro de cada llamada, el módulo de inteligencia conocerá el estado del escenario de batalla a través de la implementación de *rts.JuegoInterface* proporcionada por el videojuego.

Otros Elementos Relevantes

Dentro de la implementación del prototipo de videojuego, se cuenta con otros elementos que resultan relevantes. Entre ellos se encuentran las clases con métodos útiles utilizados a través de todo el componente videojuego. A continuación se describen estas clases, que agrupan métodos estáticos que se utilizan con mayor frecuencia.

UnitUtils

Contiene métodos relacionados a unidades.

public int getUnitBuildCost(UnitType unitType): Retorna la cantidad de recursos requerida para la construcción de una unidad.

public int getUnitResourcesProductionInterval(UnitType unitType): Retorna el tamaño del intervalo de tiempo en segundos para la producción de recursos de una unidad específica.

public int getUnitResourcesProductionAmount(UnitType unitType): Retorna la cantidad de recursos que una unidad genera cuando se completa un intervalo de tiempo de generación de recursos.

public int getUnitProtectRadius(UnitType unitType): Retorna el radio de protección que genera la unidad especificada.

public int getUnitEnergyRadius(UnitType unitType): Retorna el radio de energía que genera la unidad especificada.

public int getUnitAttackRadius(UnitType unitType): Retorna el radio de ataque de una unidad específica.

public int getUnitAttackAmount(UnitType unitType): Retorna la cantidad de daño que genera el ataque de una unidad específica.

public int getUnitAttackInterval(UnitType unitType): Retorna el tamaño del intervalo, en segundos, entre ataques de una unidad específica.

public int getUnitHitPoints(UnitType unitType): Retorna los puntos de vida de una unidad específica.

public void overlayUnitWithColor(Node unit, ColorRGBA color): Cubre la unidad con el color especificado.

public void resetUnitColors(Node unit): Reestablece el color original de una unidad.

public void removeUnitGlow(Node unit): Quita el efecto de resplandor (glow) de la unidad.

public void activateUnitGlow(Node unit): Activa el efecto de resplandor (glow) de la unidad.

public ColorRGBA getUnitPlayerColor(Node unit): Retorna el color de la unidad a partir del jugador al que pertenece.

public String resolveUnitTypeFriendlyName(UnitType unitType): Retorna el nombre amigable (que se mostrará por pantalla) del tipo de unidad especificado.

public String resolveUnitFriendlyName(UnitType unitType): Retorna el nombre amigable (que se mostrará por pantalla) de la unidad especificada.

public UnitProperties extractUnitProperties(Node unitNode): Extrae el objeto de tipo *UnitProperties* desde una unidad. Este objeto almacena propiedades de la unidad que lo contiene.

public boolean isSelectableUnitNode(Node node): Determina si un nodo es seleccionable mediante click del mouse. Un ejemplo de nodos seleccionables son las unidades. El escenario de batalla es un nodo no seleccionable.

SceneUtils

Contiene métodos relacionados al escenario de batalla.

public void initializeBlocksStates(Map<PlayerType, BattleSceneBlockState[][]> blocksStates, int xSize, int ySize): Inicializa la matriz de estados de bloques.

public boolean isBuildAllowed(UnitType unitType, int x, int y, Map<PlayerType, BattleSceneBlockState[][]> blocksStates, PlayerType player): Determina si es posible construir una unidad del tipo y en el bloque especificado, para un jugador.

public void updateBlocksStatesOnUnitPlacement(Node unitNode, int x, int y, Map<PlayerType, BattleSceneBlockState[][]> blocksStates, PlayerType player): Actualiza los bloques que representan el estado del escenario de batalla al construir una unidad.

public void updateBlocksStatesOnUnitRemoval(Node unitNode, Map<PlayerType, BattleSceneBlockState[][]> blocksStates): Actualiza los bloques que representan el estado del escenario de batalla al quitar una unidad.

public BattleSceneBlockState searchTarget(BattleSceneBlockState[][] blocksStates, Vector2f position, int attackRadius): Busca un bloque que contenga una unidad objetivo dentro de los bloques de estado con la posición y radio especificado. Si se encuentra más de una unidad objetivo, se entrega el bloque de la más cercana. Si no se encuentra una unidad, se retorna *null*.

GameUtils

Contiene métodos de uso general a través del juego.

public String getGameText(String key): Recupera texto del juego desde un archivo *properties* a partir del identificador *key*.

EffectsUtils

Contiene métodos que crean efectos sobre una entidad.

public void createUnitPlacementEffect(Node unit, AssetManager assetManager): Crea efecto de construir una unidad en el escenario de batalla.

public void createUnitDamageEffect(Node unit, AssetManager assetManager): Crea efecto de una unidad que recibe daño a partir de un ataque.

public void createUnitReflectedDamageEffect(Node unit, AssetManager assetManager): Crea efecto de una unidad que refleja el daño recibido a otra unidad.

public void createAttackerEffect(Node unit, AssetManager assetManager): Crea efecto de una unidad que ataca.

public void createUnitDestroyEffect(Node unit, Node sceneNode, AssetManager assetManager): Crea efecto de una unidad destruida.

8 Implementación de la Inteligencia Artificial

Para la inteligencia del videojuego se decidió emplear el modelo de Belief-Desire-Intention, esto debido a la similitud que posee en la interpretación de la estrategia y la planificación utilizada en los videojuegos RTS. Este modelo es principalmente utilizado en aplicaciones basadas en agentes pero eso no implica que no pueda ser utilizada de otra forma.

Las etapas del desarrollo de la inteligencia artificial fueron abordadas según el modelo presentado por (Kok, 2009) que consta de: representación del estado del juego, arquitectura de la inteligencia artificial, elaboración de plan y estrategia e implementación.

8.1 Representación del Estado del Juego

A continuación, se analiza la representación del estado del videojuego en el módulo de inteligencia artificial. Para obtener la mejor representación fue necesario tener en consideración los siguientes requisitos:

- ✓ El estado debe ser completo. Esto requiere que el estado debe representar en un momento dado la información de todo el terreno del videojuego, tanto para las unidades del contrincante como las propias, considerando la posición de las unidades, lugares disponibles de construcción y nivel de producción de recursos.
- ✓ El estado debe ser compacto. Debe existir información que resuma datos, de menor nivel, en conocimiento, de alto nivel, del terreno de juego. Esta información resumida facilitará la ejecución estratégica y táctica del algoritmo.
- ✓ El estado debe ser consistente. No debe existir información del terreno del juego que contradiga a los resúmenes o viceversa y el estado siempre debe encontrarse en una instancia coherente, esto quiere decir que, dado un terreno cualquiera, nunca se estarán vulnerando las reglas del videojuego especificadas con anterioridad.

8.1.1 Representación Preliminar

Para esta etapa se revisaron las reglas y conceptos del videojuego. El objetivo principal es generar una representación compacta, que contenga la menor cantidad de elementos y que, a su vez, entregue la mayor cantidad de información, ya sea explícita o fácilmente deducible.

El terreno del juego será representado como una matriz de tamaño N por M de enteros para simplificar el cálculo y el posicionamiento de las estructuras. Esta simplificación también se reflejará en la interfaz gráfica del juego, ya que el jugador sólo podrá posicionar las estructuras en las celdas determinadas por la matriz. Cada lugar de la matriz podrá contener seis posibles alternativas:

- ✓ Cero: Indica que en dicha posición se encuentra situado el centro de control.
- ✓ Uno: Indica que en dicha posición se encuentra situado un recolector de recursos.
- ✓ Dos: Indica que en dicha posición se encuentra situada una unidad de ataque.

- ✓ Tres: Indica que en dicha posición se encuentra situada una unidad de defensa.
- ✓ Cuatro: Indica que la posición se encuentra disponible para construir, debido a que se encuentra energizada por un recolector de recursos o por el centro de control. Esta alternativa se decidió para facilitar el cálculo de posicionamiento posterior.
- ✓ Valor Nulo (*null*): Indica que en dicha posición no existe ninguna unidad y que tampoco es posible construir ahí.

El terreno puede ser graficado en cualquier momento del transcurso del juego y puede permitir tomar determinaciones de cómo posicionar las estructuras. Un ejemplo de un estado de ejemplo del juego se puede ver en la Figura 8.1.

```
[4][4][3][3][N][N][N][N][N][N][N]
[4][0][1][4][4][N][N][4][4][4][N]
[3][2][4][1][4][N][N][4][1][3][N]
[N][4][1][4][4][N][N][4][3][1][2]
[N][4][4][4][N][N][N][N][N][2][0]
```

Figura 8.1 Representación Preliminar del Estado del Juego

Otros elementos importantes en la representación del estado del juego son los indicadores, que permiten tener información resumida acerca de varios aspectos del juego, como son el nivel de ataque, nivel de defensa o producción de recursos. Estos indicadores serán calculados al momento de que el algoritmo solicite ver el estado del juego para evaluar las acciones a seguir. El cálculo de estos índices será optimizado con el tiempo al observar que tan bien representan la información resumida y su veracidad con respecto al estado real del videojuego.

Actualmente, el terreno de juego se divide en 9 zonas, tanto para los indicadores de ataque como los de defensa, y se calcula sumando cada estructura asociada al indicador y dividiéndola por el número total de posiciones de dicho sector o también denominado área de la zona.

8.1.2 Representación Formal

Una vez avanzado el desarrollo del algoritmo de inteligencia artificial, por motivos de una mejor implementación y mayor conocimiento del avance real del motor y ejecución del videojuego, es que se tomaron las consideraciones o modificaciones a la representación preliminar.

El estado del videojuego contendrá los siguientes elementos: representación del terreno, indicadores de ataque y defensa, promedios de ataque y defensa, producción de recursos y cantidad de recursos totales.

La representación del terreno será implementada a través de un arreglo bidimensional de *UnidadTerreno*. Esta clase contiene una enumeración y un booleano. La enumeración, llamada *Unidad*, contiene todos los elementos posibles que pueden existir dentro del terreno del

videojuego, para evitar cualquier tipo de errores por inconsistencias y validaciones, facilitando la integración con el motor de videojuego.

Estas posibles unidades son: *ATAQUE, DEFENSA, RECOLECTOR, CENTRO_CONTROL, DISPONIBLE, NO_DISPONIBLE*, los primeros cuatro elementos de la enumeración son las estructuras definidas de las reglas del videojuego y las últimas dos indican si es o no posible construir alguna estructura en dicha posición.

Por último, la variable booleana, denominado *enemigo*, indica si la estructura en cuestión es perteneciente al enemigo o es una estructura propia.

Los indicadores serán implementados con un arreglo bidimensional de tamaño tres por tres de números flotantes, y existirá uno para el ataque propio y enemigo, y uno para la defensa propia y enemiga. Estos indicadores contendrán información resumida de cada sector del terreno del videojuego y serán de utilidad al momento de elegir el curso de acción y el posicionamiento local de las estructuras.

El cálculo de cada indicador se realiza al momento de solicitar una actualización del estado actual del terreno del videojuego por parte del módulo de inteligencia y se calcula como un promedio del número de dicha estructura en el área del sector que se esta considerando.

Los promedios globales de ataque y defensa tanto para el contrincante como para el jugador primario, sirven de referencia para el estado global del terreno del videojuego y son calculados por sobre todo el tablero.

El nivel de producción de recursos y la cantidad de recursos totales ayudarán al momento de enviar la acción de construir o no construir una estructura cualquiera o si es mejor esperar a la generación natural de recursos o la construcción de más estructuras de recolección de recursos.

Una representación gráfica del estado del juego se muestra en la Figura 8.2.

[C1]	[R1]	[E1]	[A1]	[D1]	[N1]	[A2]	[A2]	[E2]
[D1]	[E1]	[E1]	[D1]	[D1]	[N1]	[A2]	[A2]	[E2]
[E1]	[E1]	[R1]	[D1]	[D1]	[N1]	[E2]	[E2]	[R2]
[E1]	[E1]	[E1]	[E1]	[D1]	[N1]	[E2]	[E2]	[E2]
[E1]	[E1]	[E1]	[E1]	[D1]	[N1]	[E2]	[E2]	[R2]
[N1]	[N1]	[N1]	[N1]	[N1]	[N1]	[E2]	[E2]	[E2]
[N1]	[N1]	[N1]	[N1]	[N1]	[N1]	[E2]	[E2]	[R2]
[N1]	[N1]	[N1]	[N1]	[N1]	[N1]	[E2]	[E2]	[E2]
[N1]	[N1]	[N1]	[N1]	[N1]	[N1]	[E2]	[E2]	[C2]

Representación del Estado del Terreno de Juego

1,1	5,5	0
0	2,2	0
0	0	0

Indicador de Defensa Propia

Figura 8.2 Representación del Terreno e Indicador de Defensa

En la representación del terreno cada letra representa una unidad (no necesariamente una estructura) y el número representa el atributo booleano de pertenencia de la unidad, enemigo o propia.

8.2 Modelo de la Arquitectura

La arquitectura presentada para la interacción del videojuego con su inteligencia, como se explicó en el punto 7.4, será a través de una interfaz de comunicación. Esta interfaz contiene los métodos en común que permitirán al motor del videojuego solicitar a la inteligencia artificial un listado de acciones a ejecutar por el contrincante computador.

El modelo de la arquitectura se muestra en la Figura 8.3, con la consideración de que la interfaz de comunicación queda implícita por medio de las solicitudes y respuestas de ambos módulos.

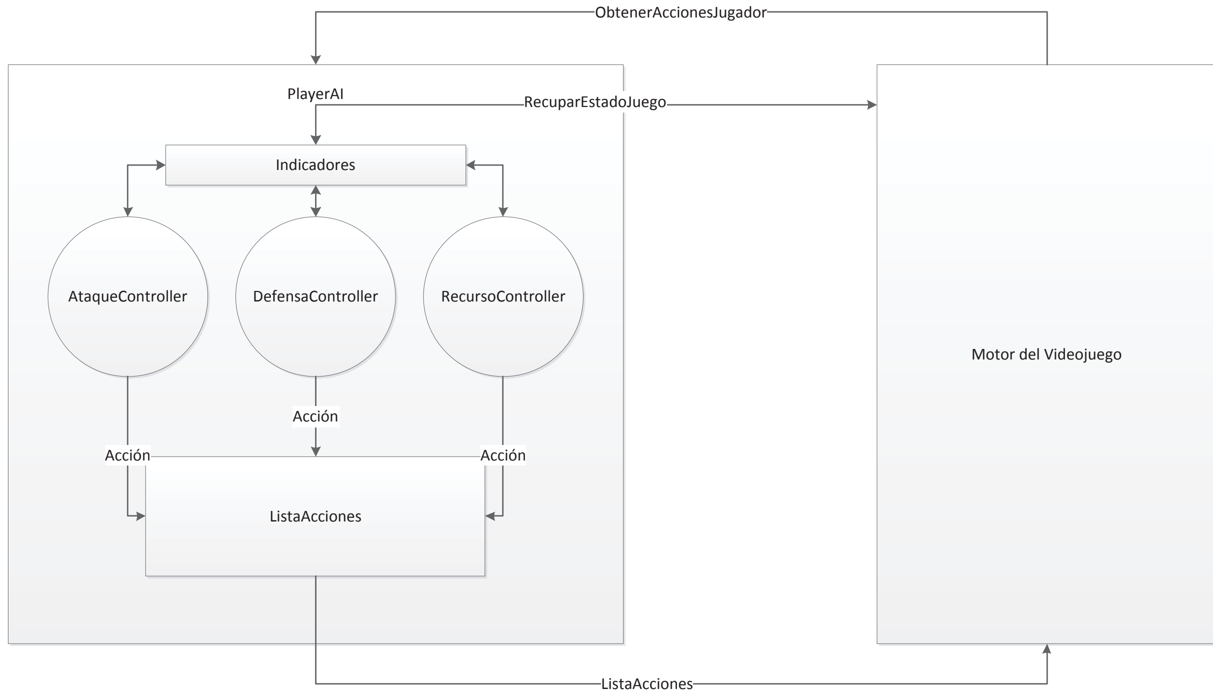

Figura 8.3 Modelo Arquitectura de la Inteligencia Artificial

Esta lista de acciones será decidida por el módulo de inteligencia artificial por medio de una implementación del modelo Belief-Desire-Intention y considerando el estado del juego. Cada acción contiene tres elementos, que le permitirán al motor saber que paso realizar:

- ✓ Posición: es una coordenada válida del terreno con el lugar relativo a la esquina superior-izquierda como punto inicial [1, 1]. Por otro lado la esquina inferior-derecha será el punto final [N, M].
- ✓ Tipo de Estructura: definirá cual será la estructura utilizada en la acción. Existen cuatro estructuras disponibles: Recolector de Recursos, Estructura de Ataque, Estructura de Defensa y Centro de Control, aunque esta última no puede ser seleccionada para realizar acciones.
- ✓ Tipo de Acción: define la acción en sí que se ejercerá sobre la estructura. Los tipos de acciones disponibles para el juego son construir, destruir y esperar.

Este módulo cuenta con tres controladores para cada concepto clave del videojuego, ataque, defensa y recurso. Estos controladores se encargarán de tomar una decisión en base a los indicadores presentes o su convicción del estado actual de juego y sus deseos de acción que tienen para resolver la problemática presente. Esto posteriormente se materializa en la intención, que se representa como una acción en la lista de acciones que aceptará el motor de videojuego para ejecutarse en el contrincante computador.

8.3 Elaboración de Plan y Estrategia

El objetivo principal del videojuego es ganarle al contrincante, ya que es un juego de dos rivales que deben luchar hasta que uno le gane a otro. Este objetivo es el más general de todos pero al que ambos contrincantes deben anhelar. Para lograr dicho objetivo se necesita cumplir otros dos objetivos secundarios, aumentar las fuerzas propias y disminuir las fuerzas del enemigo.

La fuerza de cada contrincante será un factor dependiente de la cantidad de estructuras de cada tipo que posea, esto quiere decir que se necesita aumentar la cantidad de estructuras de ataque, defensa y recolectores de recursos para aumentar la fuerza general interna. Otro factor importante es que las estructuras desconectadas del flujo de energía no serán funcionales por lo que se tiene que considerar el hecho de reparar las conexiones rotas y con eso también aumentar o recuperar las fuerzas perdidas.

Por otro lado, la misión de debilitar al enemigo se puede realizar cumpliendo dos objetivos de menor grado: cortar el flujo energético del enemigo y destruir sus estructuras. Es labor de los controladores determinar en qué parte cortarán el flujo energético, o qué estructuras destruir.

Una representación gráfica del árbol estratégico puede ser vista en la Figura 8.4.

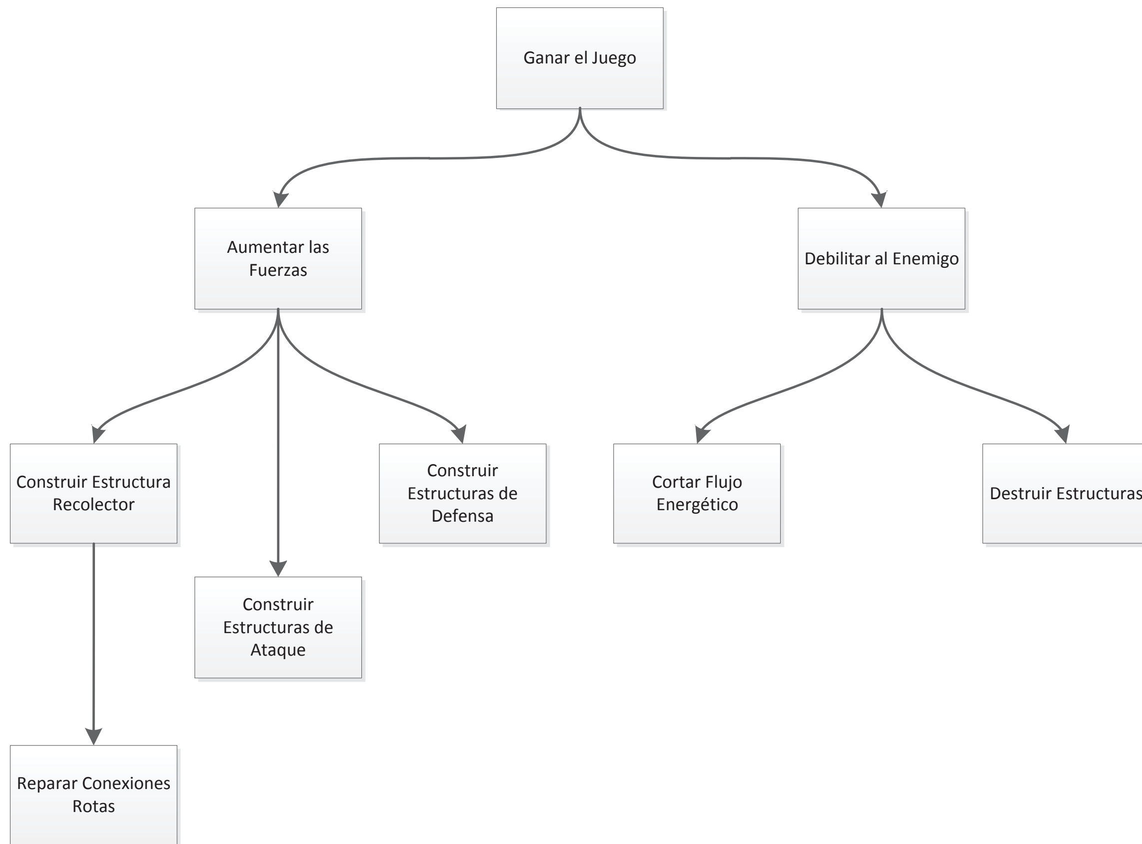

Figura 8.4 Representación Gráfica del Árbol de Estrategia

8.4 Detalles de la Implementación

A continuación se presentan elementos aplicados para la implementación del algoritmo. Se tomaron diversas consideraciones para poder abordar el problema y en algunos casos simplificar la resolución.

8.4.1 Interfaz de Comunicación Preliminar

Esta será la interfaz que utilizará tanto el módulo de inteligencia artificial como el motor de videojuegos para comunicarse entre sí. La interacción que existe entre estos dos componentes está basada en solicitud y respuesta para independizar el funcionamiento de ambos artefactos.

Los métodos que debe implementar el motor de videojuegos son:

- ✓ ***obtenerAccionesJugador(PlayerAI playerAI)***: solicita al módulo de inteligencia artificial que retorne una lista de acciones, según el estado actual del juego.
- ✓ ***construirEstructura(Estructura estructura, Integer x, Integer y)***: construye una unidad en la posición [x, y] del terreno. Esta posición siempre es válida ya que fue calculada por los controladores.
- ✓ ***destruirEstructura(Integer x, Integer y)***: destruye una unidad propia ubicada en la posición [x, y] no importando cuál sea, aunque es imposible destruir el centro de control o una unidad inexistente. El controlador se encarga de determinar la unidad y la posición según sus deseos y convicciones, siempre eligiendo una alternativa válida.
- ✓ ***tiempoEspera(Integer segundos)***: es el tiempo en segundos que debe esperar el motor antes de volver a solicitar o ejecutar una acción. Generalmente se debe a que se está esperando que se incrementen los recursos o que se realice otra acción.

8.4.2 Interfaz de Comunicación Formal

Tras consideraciones de implementación se decidió que la interfaz de comunicación sería sólo a través de la actualización del estado del terreno del videojuego, eliminando la interacción directa del componente de inteligencia artificial por sobre el terreno del videojuego y dejando a interpretación del motor del videojuego las acciones a tomar en construcción o destrucción de unidades, de esta forma se eliminan los riesgos asociados a un mal comportamiento generado por el algoritmo.

El método a implementar por el motor del videojuego es:

- ✓ ***EstadoJuego obtenerEstadoJuevoActual()***: este método debe retornar el objeto *EstadoJuego* con toda la información especificada con anterioridad en el punto 8.1.2. Debe encargarse de indicar la posición y estructura de cada punto del terreno y a quien pertenece. Una vez completada esta información base se procede a calcular la información resumida generando una llamada al método *calcularIndicadores()*.

8.4.3 Controladores

Los controladores son módulos internos de la inteligencia del jugador computador que se encargan de decidir las acciones a ejecutar por el contrincante. Estas acciones deben ser determinadas según el estado del juego y los deseos de cada controlador.

Todos los controladores son heredados de la clase *Controlador* que posee el método de ***calcularAccionSiguiente(EstadoJuego estadoJuego)***, este método considera la convicción que tiene el controlador sobre el ambiente de juego, y dependiendo de sus deseos genera una acción o intención de ejecución.

Existen tres controladores:

- ✓ ***AtaqueController***: se encarga de velar por la estrategia de ataque, aumentar las fuerzas de ataque, posicionarlas de manera coherente en el terreno y encontrar debilidades del contrincante donde poder disminuirlo.
- ✓ ***DefensaController***: análogo al controlador de ataque, este controlador busca aumentar las defensas del jugador, encontrar puntos débiles que necesiten refuerzos y proteger al Centro de Control.
- ✓ ***RecursoController***: tiene dos objetivos, conseguir la mayor cantidad de recursos y reparar las conexiones rotas en el terreno de juego.

8.4.4 Implementación de Controladores Preliminar

A continuación se muestra como cada controlador implementó el método *calcularAccionSiguiente(EstadoJuego estadoJuego)* y sus consideraciones:

AtaqueController

El controlador de ataque toma en consideración tres elementos del estado actual del videojuego, el indicador de ataque propio, el indicador de defensa del enemigo y el promedio de ataque propio.

En una primera ocasión se busca el primer sector, de noroeste a sureste, que contenga un valor menor al promedio de ataque global. Cuando se encuentre el sector débil se intenta encontrar un lugar disponible de construcción dentro del mismo sector, y al momento de encontrar el primer lugar disponible, se guarda la posición y se genera la acción de construcción de estructura de ataque, anidándola a la lista de acciones que se enviará al motor de videojuegos.

Para buscar una posición disponible, primero se encuentra el sector más fortificado del enemigo, esto quiere decir, el sector con mayor índice de defensa. Dependiendo de la ubicación de dicho sector es que se busca la disponibilidad de construcción más cercana a la orientación y ubicación encontrada.

En caso de no existir un lugar disponible de construcción, se evalúa la posibilidad de construir una estructura de recolección que habilite una posición en el sector débil o bien destruir una estructura sobrepoblada que ya no sea de utilidad.

DefensaController

Análogo al controlador de ataque, se intenta encontrar el sector más débil en términos de defensa y es en ese sector que se intenta construir una unidad de defensa como prioridad. Esto se hace en dirección al sector más fuerte del enemigo y requiere que exista a lo menos una posición disponible de construcción.

Si el sector está completamente inhabilitado para la construcción, entonces se intenta de generar un recorrido de construcción, ya sea destruyendo una estructura o construyendo estructuras recolectores.

RecursoController

Este controlador se encargará de establecer las conexiones de energía en todo el terreno del juego y reparar las ramas no conectadas, como también mantener un nivel adecuado de generación de recursos.

Si existe un exceso de estructuras de ataque y defensa en el terreno, y a la vez un déficit de estructuras recolectoras de recursos, entonces se intenta de posicionar un recolector con preferencia a reconectar una matriz sin energía.

ComunController

Existe también un controlador auxiliar a los otros controladores, denominado *ComunController*, que posee métodos de ayuda neutrales para el posicionamiento o determinación de los mejores lugares de construcción de estructuras en el terreno.

El método *obtenerPosicionDisponible* se encarga de encontrar el mejor lugar de construcción dado un sector y la orientación preferible de construcción. Esto quiere decir que dado un sector cualquiera y una dirección fija, intenta de encontrar la posición disponible más cercana a dicha dirección.

Esta operación se realiza por medio de recorridos matriciales diagonales, dependiendo de la dirección especificada, comienza a recorrer concéntricamente a dicha esquina del sector, aumentando el radio de búsqueda cada vez que no se encuentre una posición disponible en el radio de búsqueda.

9 Propiedades de Unidades

En esta sección se enuncian y describen las propiedades que determinan el comportamiento y capacidades de las unidades. Estas características pueden ser, por ejemplo, puntos de ataque, puntos de defensa, puntos de vida y costo de construcción, entre otros.

El proceso de definir estas características no es sencillo. Está dirigido por una fuerte componente de prueba y error debido, principalmente, a la cantidad de características con las que cuentan las unidades. Se debe tener en cuenta que se debe lograr un equilibrio entre los valores asignados a las propiedades, de otro modo, un jugador podría encontrar una estrategia que lo vuelve prácticamente invencible. Por lo tanto, dentro del dominio de estrategias posibles que un jugador puede llevar a cabo en el juego, dichas propiedades deben evitar que se produzcan estas estrategias que permiten obtener la victoria siempre.

Como se mencionó, el proceso de calibración está basado principalmente en la prueba y el error. Por esto, se debe recurrir un poco al sentido común para definir un conjunto de propiedades base y, luego, comenzar a modificar según sea requerido. Se puede resumir el proceso de la siguiente forma:

a) Definir un conjunto de valores base para las propiedades basándose en el sentido común. Por ejemplo, evitar utilizar valores de ataque demasiado elevados en comparación a los puntos de vida de las unidades.
b) Correr el juego y evaluar cómo se comporta en base a las propiedades que se definieron.
c) Determinar si el juego se comporta de forma esperada. Si se está satisfecho, la calibración termina aquí. De lo contrario, determinar que propiedades son las causantes de las anomalías detectadas, corregirlas y volver al paso b).

A continuación se mencionan las propiedades con las que cuenta cada unidad. En la implementación del prototipo, estas propiedades se encuentran definidas en el recurso *unitProperties.properties*.

- **Unidad Central**
    - **Costo de Construcción (*baseUnitBuildCost*)**
      Representa el costo en recursos que se deben pagar para construir una Unidad Central.
    - **Intervalo de Producción de Recursos (*baseUnitResourcesProductionInterval*)**
      Representa el número de segundos que transcurren entre cada cantidad de recursos que produce una Unidad Central.
    - **Cantidad de Producción de Recursos (*baseUnitResourcesProductionAmount*)**
      Representa la cantidad de recursos que una Unidad Central genera en cada Intervalo de Producción de Recursos.
    - **Radio de Energía (*baseUnitEnergyRadius*)**

Representa el radio (en bloques) que una Unidad Central abarca para proveer de energía a otras unidades dentro del terreno de juego.

- **Puntos de Vida (*baseUnitHitPoints*)**
  Representan los puntos de vida de la Unidad Central.

- **Unidad Recolectora de Recursos**
  - **Costo de Construcción (*collectorUnitBuildCost*)**
    Representa el costo en recursos que se deben pagar para construir una Unidad Recolectora de Recursos.
  - **Intervalio de Producción de Recursos (*collectorUnitResourcesProductionInterval*)**
    Representa el número de segundos que transcurren entre cada cantidad de recursos que produce una Unidad Recolectora de Recursos.
  - **Cantidad de Producción de Recursos (collectorUnitResourcesProductionAmount)**
    Representa la cantidad de recursos que una Unidad Recolectora de Recursos genera en cada Intervalo de Producción de Recursos.
  - **Radio de Energía (*collectorUnitEnergyRadius*)**
    Representa el radio (en bloques) que una Unidad Recolectora de Recursos abarca para proveer de energía a otras unidades dentro del terreno de juego.
  - **Puntos de Vida (*collectorUnitHitPoints*)**
    Representan los puntos de vida de la Unidad Recolectora de Recursos.

- **Unidad de Defensa**
  - **Costo de Construcción (*defenseUnitBuildCost*)**
    Representa el costo en recursos que se deben pagar para construir una Unidad de Defensa.
  - **Radio de Protección (*defenseUnitProtectRadius*)**
    Representa el radio (en bloques) que una Unidad de Defensa abarca para defender a otras unidades dentro del terreno de juego.
  - **Puntos de Vida (*defenseUnitHitPoints*)**
    Representan los puntos de vida de la Unidad de Defensa.

- **Unidad de Ataque**
  - **Costo de Construcción (*attackUnitBuildCost*)**
    Representa el costo en recursos que se deben pagar para construir una Unidad de Ataque.
  - **Radio de Ataque (*attackUnitRadius*)**
    Representa el radio (en bloques) que una Unidad de Ataque abarca para atacar a otras unidades dentro del terreno de juego.
  - **Cantidad de Ataque (*attackUnitAttackAmount*)**
    Representa la cantidad de daño que produce una Unidad de Ataque cada vez que ataca a una unidad enemiga.
  - **Intervalo de Ataque (*attackUnitAttackInterval*)**
    Representa el número de segundos que transcurren entre cada cantidad de ataque que produce una Unidad de Ataque.
  - **Puntos de Vida (*attackUnitHitPoints*)**
    Representan los puntos de vida de la Unidad de Ataque.

Tras definir un conjunto inicial de valores para las propiedades mencionadas, se realizaron pruebas para evaluar el comportamiento de las unidades. Estas pruebas consistieron en ejecutar el prototipo y jugar una partida contra el enemigo controlado por el módulo de IA. A partir del comportamiento observado se identificaron los comportamientos anómalos y se determinaron las causas posibles. Una vez establecidas las posibles causas, se tomó la decisión de aplicar correcciones sobre los valores de las propiedades, con el objetivo de corregir dicho comportamiento indeseado. Finalmente, tras aplicar las correcciones se volvió a ejecutar el prototipo para evaluar nuevamente el comportamiento de las unidades y determinar si se obtuvo el efecto deseado. Este procedimiento se repitió hasta alcanzar un comportamiento estable de las unidades.

En la Figura 9.1 se presenta el contenido del recurso *unitProperties.properties* con los valores de las propiedades obtenidos al aplicar el proceso mencionado. El significado de cada propiedad se describió anteriormente para cada unidad.

file | 27 lines (23 sloc) | 0.614 kb   Edit Raw Blame History

```
# Unit build cost
attackUnitBuildCost=2
defenseUnitBuildCost=3
baseUnitBuildCost=0
collectorUnitBuildCost=1

# Base unit properties
baseUnitResourcesProductionInterval=5
baseUnitResourcesProductionAmount=1
baseUnitEnergyRadius=2
baseUnitHitPoints=10

# Collector unit properties
collectorUnitResourcesProductionInterval=3
collectorUnitResourcesProductionAmount=2
collectorUnitEnergyRadius=2
collectorUnitHitPoints=10

# Defense unit properties
defenseUnitProtectRadius=1
defenseUnitHitPoints=10

# Attack unit properties
attackUnitRadius=2
attackUnitAttackAmount=2
attackUnitAttackInterval=3
attackUnitHitPoints=10
```

Figura 9.1 Contenido del recurso *unitProperties.properties*.

A partir de la experiencia de calibrar los valores de las propiedades, se puede afirmar que, debido a la naturaleza del procedimiento, resulta prácticamente imposible determinar por adelantado el número de iteraciones que serán requeridas para lograr un comportamiento adecuado.

10 Pruebas Modelo BDI (Belief-Desire-Intention)

Para continuar, se presentará un breve análisis basado en una serie de pruebas realizadas sobre el modelo BDI (Belief-Desire-Intention), utilizando los elementos fundamentales que contempla la arquitectura de dicho modelo, tales como; convicción, deseo, intención y evento. Lo anterior se aplicará a la inteligencia artificial del prototipo de videojuego de estrategia en tiempo real. En general, estas pruebas tienen por objetivo medir el rendimiento del modelo, mencionado con anterioridad, ante diversos escenarios y situaciones, así como también, evaluar los resultados y determinar el tiempo que demora, el modelo aplicado, en encontrar una acción factible para realizar.

Las pruebas, relativas al modelo, se llevaron a cabo en un computador portátil, que consta de las siguientes especificaciones técnicas:

- ✓ Sistema Operativo: *Ubuntu 12.04 - Distribución Linux*
- ✓ Procesador: *Intel® Core™ i5-3210M*
- ✓ Placa Madre: Mobile Intel® HM76 Express
- ✓ Memoria RAM: 4 GB 1333 MHz DDR3 SDRAM

En su totalidad, las pruebas realizadas al modelo BDI, corrieron bajo el framework de JUnit, para pruebas unitarias de Java. Es muy importante destacar que las acciones se representarán a través de tres parámetros definidos. Estos son: *Posición*, *Tipo de Estructura* y *Tipo de Acción*. Cada uno de los elementos mencionados, se explicaron de forma detallada en secciones anteriores, correspondientes a este informe.

Primero que todo, las pruebas aplicadas al modelo BDI, comienzan con la lectura de un archivo de texto plano, en el cual se encuentra la especificación de un terreno de juego. El terreno de juego, para esta ocasión se considera parcialmente válido y será utilizado como una simulación del estado actual del juego de estrategia en prueba. En base a dicho estado resultante, es que se decidirá qué acción generar y, al mismo tiempo, realizarla de manera oportuna.

10.1 Prueba Nº1: *Acciones Iniciales*

- **Nombre de Prueba:** *accionesInicialesTest*
- **Objetivo:** Registrar y evaluar las primeras acciones opcionales que despliega el modelo por defecto, para optar posteriormente por una de ellas.
- **Tiempo de Ejecución:** 0,027 segundos.
- **Mapa de Prueba:** *mapaTestInicial.txt – Anexo 1*
- **Resultado:**

  - ✓ Posición : [(7,8)]
  - ✓ Tipo de Estructura : [*RECOLECTOR*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

  - ✓ Posición : [(9,10)]
  - ✓ Tipo de Estructura : [*RECOLECTOR*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

Al dar comienzo a un nuevo juego, y asumiendo de forma hipotética que el enemigo no cuenta con unidades existentes, la primera decisión que escoge el modelo es acercarse al *Centro de Control* contrario, para más tarde iniciar el enfrentamiento. La confrontación la realiza a través de dos acciones, las cuales corresponden a la construcción de *Recolectores*, por medio de éstas se logra generar el inicio de la cadena de energía, apuntada en dirección sureste del terreno de juego, hacia donde está ubicado el *Centro de Control* enemigo.

10.2 Prueba Nº2: *Acciones Finales*

- **Nombre de Prueba:** *accionesFinalizacionTest*
- **Objetivo:** Evaluar decisiones generadas por el modelo, traducidas correctamente en acciones, ante una situación de término del conflicto, la cual sólo ocurre en el momento que el *Centro de Control* enemigo se encuentra completamente desprotegido, y en total vulnerabilidad a cualquier ataque en su contra.
- **Tiempo de Ejecución:** 0,013 segundos.
- **Mapa de Prueba:** *mapaTestCentroSolo.txt – Anexo 2*
- **Resultado:**

  - ✓ Posición : [(15,12)]
  - ✓ Tipo de Estructura : [*ATAQUE*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

  - ✓ Posición : [(14,13)]
  - ✓ Tipo de Estructura : [*ATAQUE*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

  - ✓ Posición : [(12,13)]
  - ✓ Tipo de Estructura : [*RECOLECTOR*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

En el preciso momento en que el *Centro de Control* enemigo se encuentre sin protección, la cual es brindada por estructuras de *Ataque* o *Defensa*, la única acción más recomendable a realizar, es rodearlo de estructuras de *Ataque* para finalizar el juego lo antes posible. También, al realizar la última acción antes mencionada, no permitirá que el jugador contrario logre expandirse nuevamente, ya que las estructuras de *Ataque* rodearán la base.

10.3 Prueba Nº3: *Acciones de Defensa Lateral*

- **Nombre de Prueba:** *accionesAtaqueLateralTest*
- **Objetivo:** Comprobar las decisiones que escogerá el modelo al enfrentarse a una situación de ataque lateral. Cabe destacar que en el caso de las pruebas, cada *Centro de Control* se encuentra en la misma dirección de la diagonal principal del terreno de juego, por lo tanto, por “lateral” se refiere a las diagonales secundarias aledañas a la diagonal principal.
- **Tiempo de Ejecución:** 0,015 segundos.
- **Mapa de Prueba:** *mapaTestAtaqueIzquierdo.txt – Anexo 3*
- **Resultado:**

  - ✓ Posición : [(6,13)]
  - ✓ Tipo de Estructura : [*ATAQUE*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

  - ✓ Posición : [(6,17)]
  - ✓ Tipo de Estructura : [*ATAQUE*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

  - ✓ Posición : [(0,14)]
  - ✓ Tipo de Estructura : [*RECOLECTOR*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

Una vez que el modelo alcanza y comprueba la zona en la cual existe mayor concentración de *Ataque* o *Defensa* por parte del jugador contrario, intenta construir estructuras de *Ataque* y *Defensa* con rapidez en dicha zona, para poder prevenir, amortiguar o incluso eliminar la inminente amenaza que podría significar no realizar esa acción, ya que las unidades enemigas, con alta probabilidad, se hubiesen acercado al *Centro de Control* propio y hubiesen ocasionado un daño mayor, quizás irreparable.

10.4 Prueba Nº4: *Acciones de Defensa Bilateral*

- **Nombre de Prueba:** *accionesAtaqueBilateralTest*
- **Objetivo:** Comprobar las decisiones que escogerá el modelo al enfrentarse a una situación de ataque bilateral. Cabe destacar que en el caso de las pruebas, cada *Centro de Control* se encuentra en la misma dirección de la diagonal principal del terreno de juego, por lo tanto, por "lateral" se refiere a las diagonales secundarias aledañas a la diagonal principal. En este caso, el ataque es por ambos laterales de forma simultánea.
- **Tiempo de Ejecución:** 0,015 segundos.
- **Mapa de Prueba:** *mapaTestAtaqueBilateral.txt – Anexo 4*
- **Resultado:**

  - ✓ Posición : [(10,16)]
  - ✓ Tipo de Estructura : [*RECOLECTOR*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

  - ✓ Posición : [(3,13)]
  - ✓ Tipo de Estructura : [*DEFENSA*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

  - ✓ Posición : [(4,13)]
  - ✓ Tipo de Estructura : [*RECOLECTOR*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

De forma similar a la prueba anterior, una vez que el modelo comprueba la zona donde existe una mayor concentración de ataque y defensa por parte del enemigo, el propio modelo comienza a defender la zona de la izquierda, pero no se acerca a la zona de la derecha. Todo esto se debe a que sólo se considera el indicador de mayor riesgo en el terreno de juego.

10.5 Prueba Nº5: *Acciones de Reconexión*

- **Nombre de Prueba:** *accionesReconexionRecolectoresTest*
- **Objetivo:** Evaluar, y al mismo tiempo, entender las decisiones que tomará el modelo, cuando se encuentre en la situación de que un *Recolector* o un conjunto de estos, estén desconectados del *Centro de Control*.
- **Tiempo de Ejecución:** 0,018 segundos.
- **Mapa de Prueba:** *mapaTestReconectorEnergía.txt – Anexo 5*
- **Resultado:**

  - ✓ Posición : [(10,9)]
  - ✓ Tipo de Estructura : [*RECOLECTOR*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

  - ✓ Posición : [(11,9)]
  - ✓ Tipo de Estructura : [*DEFENSA*]

- ✓ Tipo de Acción : [*CONSTRUIR*]

- ✓ Posición : [(8,7)]
- ✓ Tipo de Estructura : [*RECOLECTOR*]
- ✓ Tipo de Acción : [*CONSTRUIR*]

Por una parte, el modelo verifica cada una de las unidades existentes, verifica la conexión de los recolectores, si encuentra alguno desconectado, analiza la zona y busca el más cercano. Luego, de encontrar el más cercano, realiza la acción más conveniente, para así posteriormente, volver a conectar todos los recolectores, y no perder recursos.

10.6 Prueba Nº6: *Manejo de Indicadores de Área*

- **Nombre de Prueba:** *manejoIndicadoresAreaTest*
- **Objetivo:** Observar de qué manera se comporta el modelo frente al manejo de áreas, y cómo calcula dichos indicadores cuando las unidades del enemigo están dispersas por el mapa. Además, verificar las prioridades de ataque-defensa que posee el modelo.
- **Tiempo de Ejecución:** 0,012 segundos.
- **Mapa de Prueba:** *mapaTestIndicadoresDiferenciados.txt – Anexo 6*
- **Resultado:**

  - ✓ Posición : [(16,15)]
  - ✓ Tipo de Estructura : [*ATAQUE*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

  - ✓ Posición : [(4,2)]
  - ✓ Tipo de Estructura : [*ATAQUE*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

  - ✓ Posición : [(2,3)]
  - ✓ Tipo de Estructura : [*RECOLECTOR*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

En esta prueba, fue posible demostrar que al existir unidades cercanas al *Centro de Control*, independiente de que fuesen menos unidades respecto a otras áreas del mapa, el modelo decide defender, o atacar en su defecto, a las unidades enemigas más cercanas independiente del número, ya que prioriza la cercanía de una unidad contraria al *Centro de Control.* Por el contrario, mientras más lejanas se encuentren las unidades enemigas, menor será la prioridad que le entregará el modelo a dichas unidades.

10.7 Prueba Nº7: *Tiempo en Mapa Pequeño*

- **Nombre de Prueba:** *tiempoMapaPequeñoTest*
- **Objetivo:** Comprobar el comportamiento del modelo respecto a un mapa de tamaño pequeño, de dimensiones 10x10.
- **Tiempo de Ejecución:** 0,010 segundos.
- **Mapa de Prueba:** *mapaTestPequeño.txt – Anexo 7*
- **Resultado:**

  - ✓ Posición : [(4,5)]
  - ✓ Tipo de Estructura : [*RECOLECTOR*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

  - ✓ Posición : [(4,6)]
  - ✓ Tipo de Estructura : [*RECOLECTOR*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

Es posible apreciar, a través de la ejecución, que el tiempo que tarda es mínimo.

10.8 Prueba Nº8: *Tiempo en Mapa Mediano*

- **Nombre de Prueba:** *tiempoMapaMedianoTest*
- **Objetivo:** Comprobar el comportamiento del modelo respecto a un mapa de tamaño mediano, de dimensiones 100x100.
- **Tiempo de Ejecución:** 0,013 segundos.
- **Mapa de Prueba:** *mapaTestMediano.txt – (debido a su tamaño, no se incluye en anexo)*
- **Resultado:**

  - ✓ Posición : [(80,83)]
  - ✓ Tipo de Estructura : [*ATAQUE*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

  - ✓ Posición : [(84,82)]
  - ✓ Tipo de Estructura : [*DEFENSA*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

  - ✓ Posición : [(80,81)]
  - ✓ Tipo de Estructura : [*RECOLECTOR*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

Se puede observar que el tiempo de ejecución varía levemente, a pesar que el tamaño del mapa de juego aumentó diez veces su tamaño anterior.

10.9 Prueba Nº9: *Tiempo en Mapa Grande*

- **Nombre de Prueba:** *tiempoMapaGrandeTest*
- **Objetivo:** Comprobar el comportamiento del modelo respecto a un mapa de tamaño grande, de dimensiones 500x500.
- **Tiempo de Ejecución:** 0,3 segundos.
- **Mapa de Prueba:** *mapaTestGrande.txt – (debido a su tamaño, no se incluye en anexo)*
- **Resultado:**

  - ✓ Posición : [(419,413)]
  - ✓ Tipo de Estructura : [*RECOLECTOR*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

  - ✓ Posición : [(417,413)]
  - ✓ Tipo de Estructura : [*RECOLECTOR*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

Se puede apreciar que el tiempo de ejecución varía de mayor forma (30 veces más), considerando que la dimensión de la matriz del mapa de juego aumentó 5 veces.

10.10 Prueba Nº10: *Tiempo en Mapa Gigante*

- **Nombre de Prueba:** *tiempoMapaGiganteTest*
- **Objetivo:** Comprobar el comportamiento del modelo respecto a un mapa de tamaño gigante, de dimensiones 5000x100.
- **Tiempo de Ejecución:** 0,8 segundos.
- **Mapa de Prueba:** *mapaTestGigante.txt – (debido a su tamaño, no se incluye en anexo)*
- **Resultado:**

  - ✓ Posición : [(4164,80)]
  - ✓ Tipo de Estructura : [*RECOLECTOR*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

  - ✓ Posición : [(4164,81)]
  - ✓ Tipo de Estructura : [*RECOLECTOR*]
  - ✓ Tipo de Acción : [*CONSTRUIR*]

Por último, es posible comprobar que a medida que aumenta la dimensión de la matriz del mapa de juego, aumenta de forma proporcional el tiempo de ejecución. Sin embargo, en ningún caso esto afecta o provoca un gasto desmesurado de memoria de la plataforma en uso.

11 Conclusiones

En la industria de los videojuegos, dentro del abanico de géneros disponibles, el de estrategia es uno de los que más requiere de la utilización de inteligencia artificial para dirigir el comportamiento del agente obstaculizador del protagonista. Esto se debe a que el género en cuestión se basa, principalmente, en el pensamiento y la planificación, motivo principal por el que se desarrolló una inteligencia artificial.

Existen muchos enfoques para abordar el problema de dotar con inteligencia artificial a entes que, en el caso de los videojuegos, son virtuales. Una de las razones más importantes por las cuales se descartaron alternativas de algoritmos de inteligencia artificial fue la eficiencia, desde el punto de vista de intensidad en cómputo. Es por esto que se utilizó un modelo Belief-Desire-Intention para la elaboración e implementación del videojuego del género de estrategia en tiempo real.

Se establecieron las reglas que rigen el videojuego, donde se contextualiza a los jugadores, se describieron las características de cada unidad y las condiciones que se deben cumplir para ganar la partida. Se prestó especial atención a esta etapa, porque las reglas definen las bases del videojuego. Esto fue un acierto, que entregó un concepto estable, ya que evitó desviar el enfoque a medida que progresaba el proyecto.

Dado que el desarrollo de videojuegos es una actividad multidisciplinaria, se requiere de la utilización de diversas tecnologías que apoyen el diseño, integración e interacción de recursos que el videojuego contiene. Las herramientas utilizadas en la implementación del videojuego se clasifican en tres áreas generales: herramientas de edición de recursos (como Blender), herramientas del entorno de programación (como Eclipse) y frameworks específicos para el desarrollo de videojuegos (como jMonkeyEngine).

El proceso de implementación de un videojuego puede resultar bastante complejo. Es por ello que se vuelve necesario contar con desarrolladores disciplinados. Es decir, se deben establecer estándares y directrices basadas en las buenas prácticas, además de recurrir a patrones de diseño cuando sea posible. Con esto se produce código de calidad y fácil de mantener, con un impacto positivo dentro de cualquier proyecto de software.

Considerando la complejidad del módulo de inteligencia artificial, se optó por implementarlo e integrarlo como un módulo dentro del prototipo del videojuego. De esta forma, se tienen dos módulos complejos (videojuego y módulo de inteligencia artificial) que se comunican mediante interfaces establecidas que representan el contrato de interacción entre ambos.

Las pruebas de integración son fundamentales en el desarrollo de cualquier software, por lo que fue necesario aplicarlas utilizando los elementos principales que contempla la arquitectura del modelo BDI (Belief-Desire-Intention). Se aplicó cada uno de los elementos (captura del ambiente, procesos dirigidos por deseos y la materialización de la acción) a la inteligencia artificial del prototipo de videojuego, para medir el rendimiento del modelo ante

distintos escenarios y diversas situaciones. De tal manera, fue posible evaluar resultados y determinar tiempos de demora en encontrar una acción posible para realizar.

A modo de trabajo futuro, se proponen mejoras que permitirían refinar ciertos aspectos del prototipo. Desde un punto de vista cosmético, es posible mejorar los modelos 3D, con lo que se obtendría una apariencia más atractiva. En cuanto al comportamiento de las unidades, realizar más iteraciones de calibración de sus propiedades permitiría la elaboración de más estrategias alternativas, previniendo la utilización de una única estrategia que siempre lleve a la victoria. Desde la perspectiva de la inteligencia artificial, aplicar mejoras en el modelo BDI permitiría al jugador controlado por el computador elaborar estrategias de juego más avanzadas y efectivas.

Finalmente, se concluye que el uso de un modelo BDI para implementar la inteligencia artificial de un videojuego resulta conveniente principalmente por dos razones. En primer lugar, cuando se busca minimizar la carga producto del cómputo intensivo de una aplicación y, en segundo lugar, cuando se busca flexibilidad en el modelamiento del problema, así como también en su implementación.

12 Referencias

Aamodt, A. & Plaza, E., 1994. *Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches.* s.l.:Artificial Intelligence Communications.

Aha, D., Molineaux, M. & Ponsen, M., 2005. *Learning to win: Case-Based Plan Selection in a Real-Time Strategy Game.* s.l.:ICCBR'2005.

Bratman, M., 1999. *Intentions, Plans, and Practical Reason.* s.l.:CSLI Publications, Ventura Hall.

Buckland, M., 2005. *Programming Game AI by Example*. Texas: Wordware Publishing.

Buro, M. & Furtak, T., 2003. *RTS Games as Test-Bed for Real-Time Research.* s.l.:Invited Paper at the Workshop on Game AI JCIS.

Chacon, S., 2009. *Pro Git*. s.l.:Apress.

Cole, N., Louis, S. J. & Miles, C., 2002. *Using a Genetic Algorithm to Tune First-Person Shooter Bots*.

Cheng, D. & Thawonmas, R., 2005. *Case-Based Plan Recognition for Real-Time Strategy Games*. s.l.:Department of Human and Computer Intelligence Ritsumeikan.

Crawford, C., 1984. *The Art of Computer Game Design*. s.l.:McGraw-Hill/Osborne Media.

Davison, A., 2005. *Killer Game Programming in Java*. s.l.:O'Reilly Media.

Fairclough, C., Fagan, M., Namee, B. & Cunnigham, P., 2002. *Research Directions for AI in Computer Games*. s.l.:Department of Computer Science Trinity College Dublin.

Gregory, J., 2009. *Game Engine Architecture*. s.l.:AK Peters.

Johnson, D. & Wiles, J., 2001. *Computer Games with Intelligence.* Queensland: 10th IEEE Int'l Conf. on Fuzzy Systems.

Kok, E., 2009. *Adaptive Reinforcement Learning Agents in RTS Games.* s.l.:Intelligent Systems Group.

McConnell, J., 2006. *Computers Graphics: Theory Into Practice*. Sudbury: Jones and Barlett Publishers, Inc.

Millington, I. & Funge, J., 2009. *Artificial Intelligence for Games*. Segunda ed. Burlington: Morgan Kaufmann Publishers.

Molyneux, P., 2002. *The Future of AI Games: A Personal View.* s.l.:Game Developer.

Nilsson, N., 1998. *Artificial Intelligence: A New Synthesis*. New York: Morgan Kaufmann.

Ontañón, S., Mishra, K., Sugandh, N. & Ram, A., 2008. *Case-Based Planning and Execution for Real-Time Strategy Games.* s.l.:Cognitive Computing Lab.

Rao, A. & Georgeff, M., 1995. *BDI Agents: From Theory to Practice.* s.l.:Proceedings of the First International Conference on Multiagent Systems.

Rollings, A. & Adams, E., 2003. *Andrew Rollings and Ernest Adams on Game Design.* s.l.:New Riders Publishing.

Ross, B., 1989. *Some Psychological Results on Case-Based Reasoning.* s.l.:Morgan Kaufmann.

Schwab, B., 2004. *AI Game Engine Programming in Java*. s.l.:Charles River Media.

Walther, A., 2006. *AI for Real-Time Strategy Games,* Copenhagen: Communication and Media IT-University.

Watson, I., 2004. *Optimization in Strategy Games: Using Genetic Algorithms to Optimize City Development in FreeCiv*.