UNIVERSIDAD AUTÓNOMA METROPOLITANA UNIDAD AZCAPOTZALCO

DIVISIÓN DE CIENCIAS BÁSICAS E INGENIERÍA

INGENIERÍA EN COMPUTACIÓN

PROYECTO TERMINAL:

Plataforma de juego programable para Quoridor

DESARROLLADO POR:

Hernández Hernández Fabrizio Alonso 205301796

Hernández Piña Hugo César 205301958

ASESOR DE PROYECTO:

Dr. Francisco Javier Zaragoza Martínez (20197)

Departamento de Sistemas

UAM Azcapotzalco

MÉXICO D.F

SEPTIEMBRE DEL 2010

Tabla de contenido

Introducción.

Objetivos Generales.

Crear una plataforma visual en 3d para que alumnos que deseen practicar su programación sean capaces de desarrollar Agentes Inteligentes (AI) que puedan llevar a cabo partidas de Quoridor en contra de otros Agentes desarrollados y ver dicha partida en acción.

Desarrollar e implementar un algoritmo capaz de jugar partidas de Quoridor y de aprender de sus partidas perdidas, para ser más difícil de vencer cada vez.

Objetivos Particulares.

- Desarrollar el módulo responsable de verificar las reglas del juego.
- Desarrollar el módulo responsable de cargar los AI.
- Desarrollar un módulo encargado de mostrar visualmente (con gráficos tridimensionales) los movimientos realizados en el juego por los Agentes.
- Poner en funcionamiento cooperativo los módulos anteriores de modo que el último muestre los juegos verificados por el primero.
- Elegir (o diseñar en caso de no encontrar uno que nos satisfaga) un AI para jugar Quoridor.
- Implementar dicho AI dentro de nuestra plataforma.

Historia y Reglas de Quoridor.

Quoridor es un juego de dos o cuatro jugadores, que se enfrentan en un tablero de 9 por 9, en donde cada jugador cuenta con un peón y diez barreras (o cinco en el modo de cuatro jugadores). Cada jugador inicia en un extremo del tablero, y para ganar deberá llevar su peón al extremo opuesto del tablero antes que su contrincante.

Viendo la ilustración **¡Error! No se encuentra el origen de la referencia.** 1, el jugador rosa ganará si llega a la fila A antes que el jugador azul llegue a la fila I, sin importar en qué columna.

En cada turno, el jugador puede ya sea o avanzar una celda a la vez ya sea hacia adelante, atrás, derecha o izquierda, o colocar una de sus barreras disponibles. Las barreras pueden colocarse solamente de modo que bloqueen dos y no más ni menos celdas a la vez, y tampoco pueden éstas

cruzarse, ni salirse del tablero. La Figura 2 nos muestra ejemplos de barreras colocadas rompiendo las reglas.

De igual manera, si el paso a una celda está bloqueado por una barrera, es contra las reglas moverse hacia esa celda.

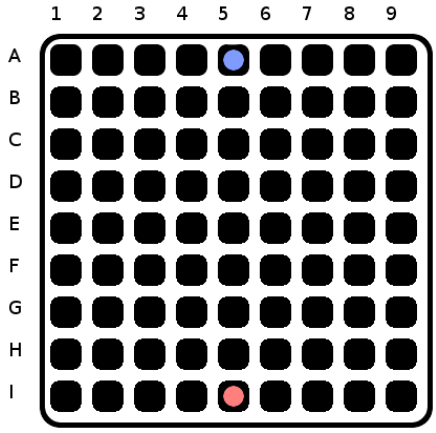


Figura 1. Posición inicial.

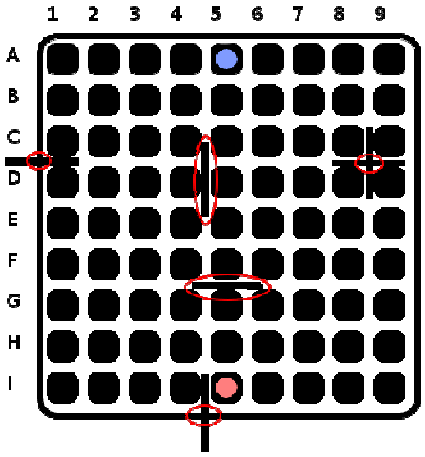


Figura 2. Barreras mal colocadas

Si un peón se encuentra adyacente a otro, puede saltarlo, avanzando así dos celdas (en lugar de una sola como es normalmente permitido). Si un peón se encuentra adyacente el otro, pero hay una barrera bloqueando su salto, éste puede saltar hacia los lados del peón contrincante, como se muestra en la Figura 3.

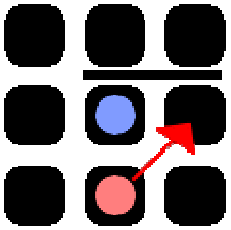


Figura 3: Saltar peón contiguo.

Quoridor fue diseñado por Mirko Marchesi y publicado por Gigamic Games en 1997. El mismo año recibió el premio *Mind Game* por parte de Mensa[1] y también el premio *Game Of The Year* en E.U.A., Francia, Canadá y Bélgica. [1]

En mayo del 2005, Lisa Glendenning de The University of New Mexico Albuquerque, New Mexico, desarrolló una tesis llamada Mastering Quoridor[2] en la cual se propone un algoritmo genético capaz de jugar Quoridor y de aprender de sus errores, aunque (según su propia tesis) dicha implementación no plantea un desafío para un jugador humano.

En junio del 2006, P.J.C. Mertens escribió un documento llamado A Quoridor Playing Agent[3] que plantea estudiar los algoritmos y las heurísticas que pueden ser usadas para desarrollar un agente capaz de jugar Quoridor.

Antecedentes.

Nuestro proyecto terminal tiene relación con los últimos dos trabajos mencionados en la sección Historia y Reglas de Quoridor., pues nosotros desarrollaremos un algoritmo (red neuronal) capaz de jugar al Quoridor y aprender de sus fracasos de modo que cada vez sea más difícil vencerle.[2]

Existen varias implementaciones del juego que están disponibles para su descarga en internet, algunas son libres, mientras otras tienen implementaciones cerradas. A continuación las listamos:

- *Quoridor* es una implementación de Quoridor escrita en Java, presenta una interfaz gráfica bidimensional y, de manera similar a nuestro proyecto, permite al usuario desarrollar su propio *brain* (como lo llama el autor) para que juegue en contra de otro usuario [4].

- *Corridor* [5] es una implementación de Quoridor bajo la licencia GNU GPL escrita en C++ y haciendo uso de la biblioteca de gráficos bidimensionales SDL. No presenta ninguna inteligencia artificial, tan sólo permite a dos jugadores humanos enfrentarse.

- Existe otra implementación de Quoridor sin nombre. Tiene una interfaz de sólo texto y únicamente permite a contrincantes humanos enfrentarse. El código no está disponible. [6]

- El sitio *quoridor.net* [7] ofrece una implementación de Quoridor en tres dimensiones hecha en Adobe Flash que permite jugar contra un oponente artificial o contra oponentes humanos. Soporta la modalidad de dos o cuatro jugadores. El código no está disponible.

[1] La más antigua y grande sociedad para coeficientes intelectuales altos. Sin fines de lucro y abierta a cualquiera que califique más de 98 puntos de coeficiente intelectual en alguna prueba estandarizada.

[2] Como veremos más adelante, nuestro propósito de crear un algoritmo de aprendizaje para nuestra implementación de Quoridor cambió, por recomendación de un par de profesores de la U.A.M. Azcapotzalco.

- *Quoridor Digital* [8], es una implementación de luego en tres dimensiones realizada por Francisco Alpalhão de Matos Bandeira dos Santos y Nuno José Pinto Bessa de Melo Cerqueira , de la Facultad de Ingeniería de la Universidad de Porto, en Portugal. Sólo permite a dos usuarios enfrentarse. Ni el código, ni más información está disponible.

Por parte de la Universidad Autónoma Metropolitana unidad Azcapotzalco encontramos el proyecto terminal de Guillermo Augusto Sánchez Sánchez llamado *Máquina de aprendizaje en un juego de ajedrez*, realizado en el 2008. Éste tiene relación al nuestro pues aunque él trabajó en su proyecto sobre el juego del ajedrez, al igual que nosotros realiza un agente capaz de aprender (ver pie de página 2).

Justificación.

Como se puede ver, no pudimos encontrar un proyecto que reuniera todas las siguientes características:

- Permitir al usuario desarrollar sus propios AIs.
- Tener gráficas tridimensionales.
- Ser de código abierto.

Todas las implementaciones encontradas están concentradas más bien en poner a competir al menos a algún jugador humano. A diferencia de éstas, nuestra implementación está enfocada en que los usuarios escriban su propio agente inteligente que será después puesto a competir contra otros.

De esta manera, daremos la capacidad a quien desee, por ejemplo a un profesor de Diseño de Algoritmos, de realizar competencias de manera automática, y mostrando una interfaz de usuario 3d llamativa, en la que los alumnos tendrán que escribir sus propios agentes de juego de Quoridor, para practicar y demostrar sus habilidades en dicha asignatura.

Así contribuiremos a hacer más didáctica la enseñanza de distintas áreas de la Computación, ya que se motiva al alumno a través de la competencia, haciendo interesante la participación de su AI al verle jugar en la pantalla en contra de otro.

Un estudiante de Ingeniería en Computación tiene todas las habilidades y destrezas necesarias para llevar a cabo este proyecto. Cuenta con los conocimientos acerca de Análisis y Diseño de Algoritmos y de Inteligencia Artificial, para poder crear al agente inteligente que juegue Quoridor. Cuenta también con conocimientos de Graficación, necesarios para el desarrollo del módulo que presenta el juego en 3d.

Cambios Realizados a la Propuesta Original.

La justificación mostrada, al igual que los objetivos particulares y generales, corresponden a los de la propuesta de este mismo proyecto. Sin embargo, durante el curso del desarrollo del proyecto, se tomaron decisiones que cambiaron un poco estos objetivos. Los cambios son:

- El módulo encargado de cargar a los agentes ya no recibirá agentes escritos en C++. Esta decisión se tomó considerando que escribir en C++ no es demasiado intuitivo, o bien, porque la necesidad de manejar apuntadores y memoria manualmente a la que nos expone C++ podría distraer al escritor de AIs de su propósito principal. Así, se decidió escribir un módulo que recibiera agentes escritos en el lenguaje de programación Python, haciendo más sencilla la escritura de los AIs.

- No se utilizó un algoritmo de aprendizaje de ningún tipo. Ya avanzados en el ciclo de desarrollo del proyecto, surgió la necesidad de analizar e implementar un algoritmo para jugar Quoridor; hablando con la profesora Ana Lilia Concepción Laureano Cruces (UAM Azcapotzalco, CBI) se nos dijo que no había mucho que se le pudiera poner a aprender a un AI por sí mismo en este caso. Así que la idea de usar algoritmos genéticos para el aprendizaje se descartó, y se implementó el algoritmo Minimax (descrito en el próximo capítulo).

Método de Desarrollo.

Descripción Técnica:

El desarrollo del proyecto consiste en el diseño, análisis e implementación de tres módulos principales:

- Módulo responsable de mostrar la partida en 3d (nombrado “Grafico”).
- Módulo responsable de cargar los agentes inteligentes (nombrado “Scripting”, por ser un módulo que media la interacción entre nuestro código en C++ con el lenguaje Python).
- Módulo responsable de la verificación de las reglas (nombrado “Reglas”).

Cada módulo se desarrolló independientemente en forma de una biblioteca dinámica. Esto nos ayuda a llevar el desarrollo de manera más independiente, además que facilita tener un acoplamiento bajo entre las clases de nuestra aplicación. Por este mismo motivo también, cada módulo contiene sus clases bajo su propio espacio de nombres, además de que, donde fue posible, se manejó toda la interacción externa con los módulos a través de una sola clase, y con una interfaz simple.

También se cuenta con otros dos módulos secundarios: “*Opciones*”, que se encarga de recibir las opciones de la aplicación (ya sea por línea de comando o por el archivo de configuración) y el módulo “*AgenteWrapper*”, que se encarga de exponer las clases y objetos globales de nuestra aplicación a los AIs escritos en Python.

Simplificando, podemos ver la interacción de los tres módulos principales de la siguiente manera:

- El módulo de carga de agentes crea a dos jugadores para ser usados por el módulo de verificación de reglas.
- El módulo de verificación está en constante comunicación con los jugadores y con el módulo de presentación gráfica. Informará a los jugadores de cuándo es su turno, además de entregarles más información esencial para poder jugar (posición, posición del enemigo, movimientos disponibles, etc.). Al informar a los jugadores sobre cuándo es su turno, también revisa que no se rompa ninguna regla. Al módulo de presentación de partidas se le informa cuándo se ha hecho un movimiento y cuál fue.
- El módulo de presentación pedirá al módulo de reglas las jugadas correctas de cada jugador, y muestra la animación de dicha jugada en pantalla.

La Figura 4 muestra esta interacción.

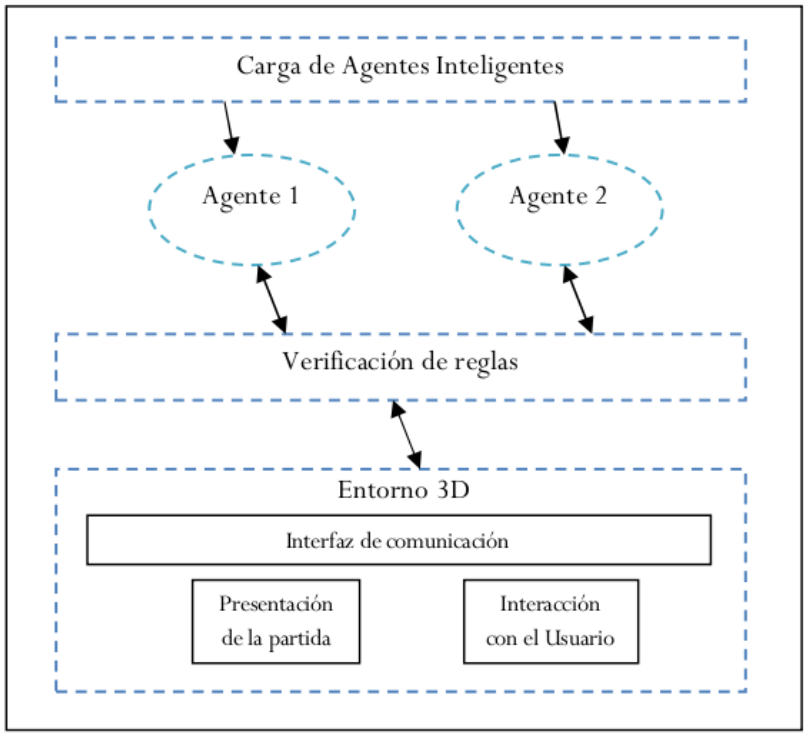


Figura 4: Diagrama de componentes del sistema.

A continuación serán descritos cada uno de los tres módulos principales con un poco más de profundidad.

Módulo "Scripting".

Este es el módulo encargado de leer los AIs para que puedan ser usados por la aplicación. Como ya se mencionó, los AIs están escritos en Python, mientras la aplicación está escrita en C++, lo que nos deja con el problema de comunicarnos hacia atrás y hacia adelante con Python. Para esto, hicimos uso de la biblioteca boost::python, parte de boost[3], que nos permite anidar un intérprete de Python dentro de nuestra aplicación, exponer clases, interfaces, y objetos a los scripts y de igual manera, recibir valores de retorno desde métodos en los scripts, u objetos creados desde implementaciones hechas en Python de interfaces definidas en C++; siendo esta última característica vital para nuestra arquitectura.

Dentro del módulo Reglas, existe una interfaz (clase puramente abstracta) que define el comportamiento que debe tener un agente inteligente. Presenta tres métodos puramente abstractos:

- virtual void iniciar(int id) = 0;
    - Este método se manda a llamar para cada AI antes de que comience la partida. El parámetro id le informa al AI de su identificador, que es necesario para poder pedir información acerca de su posición o barreras disponibles, por ejemplo.
- virtual Reglas::Jugada siguienteJugada() = 0;
    - Este método es llamado por el módulo Reglas cada vez que desea saber la jugada a ejecutar por un agente. En este método, el AI debe armar su jugada con la información que se le proporciona, y regresarla como un objeto tipo Reglas::Jugada.
- Virtual void terminar() = 0;
    - Este método se llama al terminar la partida. Puede ser usado para liberar recursos.

El módulo secundario "AgenteWrapper" expone esta interfaz a los scripts en Python, para que puedan heredar de ella, creando efectivamente sus AIs.

Toda la interacción con el módulo Scripting, se lleva a cabo a través de la clase Scripting::Manejador. Esta clase contiene a los posibles intérpretes necesarios para ejecutar los scripts (representados por la clase Scripting::Interprete), y a su vez, estos intérpretes contienen a todos los scripts que están ejecutando (representados por la clase Scripting::Modulo).

Para ejecutar un script, y recibir una instancia de una clase (de nombre desconocido) heredada de Reglas::Agente, se llama al método Scripting::Manejador::getAgente, que recibe la ruta al script a ejecutar. El manejador busca entre sus intérpretes si es que alguno maneja ese tipo de archivo. El manejador entonces pide al intérprete un Agente a partir de ese archivo. El interprete crea un módulo (y lo deja abierto para no tener que volver a procesar el archivo de ser que se le pidan más

[3] Boost http://www.boost.org/ es un conjunto de bibliotecas de primer nivel, cuyo propósito (entre otros) es el de crear una comprensiva y extensa biblioteca estándar para C++.

agentes desde el mismo archivo) y busca en él una clase que herede de Reglas::Agente, para poder crear una instancia de ella, y regresarla. Toda esta interacción se puede ver de forma más o menos simplificada en el siguiente diagrama de interacción (Figura 5).

Una vez que comprendemos este módulo, podemos explicar el siguiente: Reglas, que necesitará de los Agentes creados por Scripting.

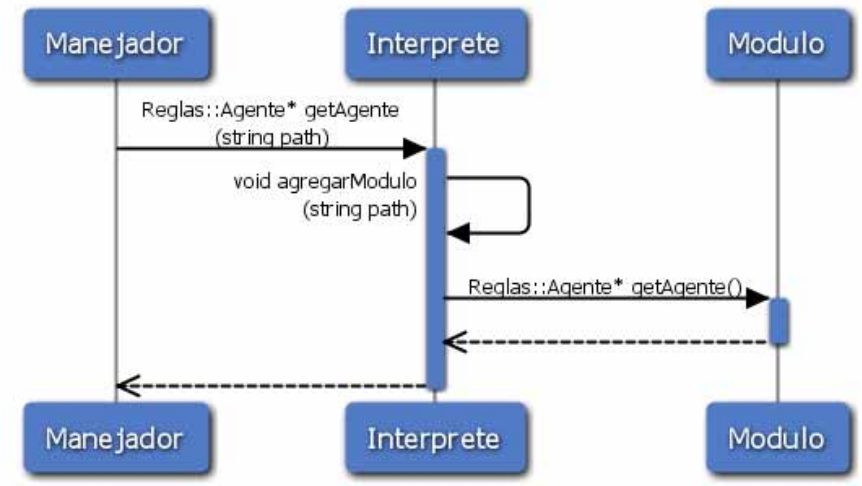


Figura 5: Diagrama de interacción del módulo Scripting.

Módulo "Reglas".

Este módulo mantiene toda la funcionalidad central de la aplicación. Contiene la lógica de las partidas del juego, y también todas las clases necesarias para la representación de las mismas.

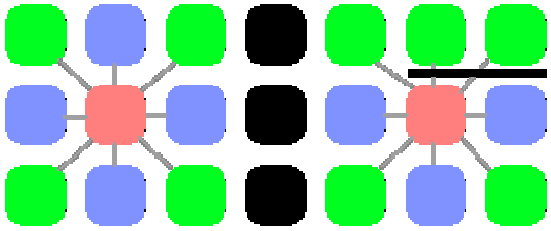
Las clases más importantes que define o implementa son las siguientes:

- *Agente*. Esta es una interfaz que define el comportamiento de un Agente Inteligente; se debe heredar de ella para garantizar que la clase se comporta como uno.
- *AyudanteDeAgente*. Una clase meramente de apoyo. Mantiene un apuntador al Tablero del juego, y de esta manera es capaz de otorgar a los AIs en cada movimiento información vital para realizar las jugadas. Los métodos más importantes que implementa son:
    - *std::list<Jugada> getMovimientosPosibles(int numJugador)*: Que regresa una lista de Jugadas, correspondientes a los movimientos que el Jugador con identificador numJugador puede realizar a continuación sin romper las reglas.
    - *std::list<Jugada> getBarrerasPosibles(int numJugador)*: Que regresa una lista de Jugadas, correspondientes a las todas las Barreras que puede colocar el AI sin romper las reglas.
    - *const Celda& getCelda(int idJugador)*: Que regresa la Celda en la que se encuentra el Jugador con identificador idJugador.

- *Tablero*. Representa al tablero del juego. Tiene una matriz de celdas, y contiene también todo el comportamiento asociado al tablero (es decir, se pueden colocar piezas sobre él, y obtener información de la posición y cantidad de piezas). Los métodos más importantes que implementa son:
    - *void setJugadores(const std::vector< Jugador* > &jugadores)*: Coloca a los Jugadores sobre el tablero, les da un identificador y una posición.
    - *void setBarrera(int idJugador, const Barrera &b), y sobrecargas*: Coloca una Barrera sobre el Tablero pues es necesario para actualizar la información acerca de las barreras disponibles para cada Jugador (Tablero también se hace cargo de esta información).
    - *void moverJugador(int jugador, int x, int y) y sobrecargas*: Mueve al Jugador con identificador "jugador" desde su posición actual sobre el Tablero, hasta la posición (x, y). Donde "x" es la posición sobre las columnas y "y" es la posición sobre las filas.
    - *const std::vector<int> &getPosicion(int idJugador) const*: Igual a AyudanteDeAgente::getPosicion.
    - *const Celda &getCelda(int x, int y) const*: Nos regresa al objeto Celda colocado en la matriz de Celdas en la posición (x, y).
- *Pieza*: Esta clase abstracta define la interfaz, características y comportamiento que comparten todos los elementos colocables sobre el Tablero (Celda, Barrera y Jugador).
    - *virtual bool estaColocado() const*: Nos indica si la Pieza ya ha sido colocada sobre el Tablero.
    - *virtual void colocar(int x, int y) y sobrecargas*: Le da a la Pieza una posición inicial. Sólo puede ser llamada una vez.
    - *virtual const std::vector<int> &getPosicion() const*: Nos indica la posición actual de la pieza.
- *Celda*: Clase que sirve para representar a cada Celda de las ochenta y una que contiene el Tablero. Hereda de Pieza, y por lo tanto todo su comportamiento. Tiene además otras propiedades, como la de saber si está siendo ocupada por un Jugador, y apuntadores a sus Celdas vecinas (o hijos, como les llamamos, característica necesaria para poder implementar la funcionalidad de grafo sobre las celdas del Tablero), explicado más adelante. Los métodos más importantes que implementa son:
    - *bool estaLibre() const*: Nos indica si la Celda está ocupada por un Jugador.
    - *void bloquear(bool bloqueo = true)*: Sirve para indicarle a la Celda si ha sido ocupada o desocupada por un Jugador.
    - *void setHijo(Direccion d, const Celda &c)*: Establece a la Celda "c" como hija de la celda actual, en la Dirección "d". Hay cuatro direcciones establecidas: NORTE, ESTE, SUR y OESTE.
    - *Celda& getHijo(Direccion d) const*: Nos entrega al hijo de la Celda actual en la dirección "d".

- *bool tieneHijo(const Celda& hijo) const*: Una Celda no siempre tiene 4 hijos. Por ejemplo, si una Celda se encuentra en la fila que está más al SUR del Tablero, ésta no tendrá hijo en la dirección SUR. De igual manera si hay una Barrera colocada, se bloqueará el paso a algunas Celdas, y mientras esté bloqueado el paso de una Celda a otra, no se le considera como hija. La siguiente figura explica la relación entre las Celdas y sus hijos. Las Celdas verdes y azules nos indican a las Celdas que rodean a las Celdas rosas, pero mientras las azules son consideradas hijas de las rosas, las verdes no.

  Figura 6: Relación entre celdas contiguas.

- *void bloquearDireccion(Direccion d)*: Bloquea la dirección "d" de una Celda, de modo que se reporte que no tiene hijo en dicha dirección.
- *void desbloquearDireccion(Direccion d, Celda &celdaHijo)*; Pone al apuntador de hijo de la Celda actual en dirección "d" apuntando a la Celda "celdaHijo".
- *bool estaLibreDireccion(Direccion d) const*: Nos indica si la dirección "d" de la Celda actual está apuntando hacia otra Celda o está bloqueada.

- *Jugador*: Representa a un peón dentro del Tablero. Al heredar de Pieza, también comparte todo su comportamiento. Esta clase conserva a un Agente dentro (y por lo tanto también comparte los mismos métodos que Agente), quien puede decidir los movimientos del Jugador en una partida. Sus métodos más importantes son:
  - *void mover(const std::vector<int> &nuevaPos)*: Mueve al Jugador de su posición actual, a la provista como parámetro (un vector de dos enteros).
  - *int getBarrerasDisponibles() const*: Nos indica el número de Barreras que le quedan al Jugador.
  - *int getIdentificador() const*: Nos indica el identificador asignado al Jugador actual.
- *Barrera*: Representa a cada una de las barreras que pueden ser colocadas en el Tablero por los Jugadores. Esta clase también hereda de Pieza, pero agrega otras propiedades, tales como una posición de punto medio, una posición de punta (pues una Barrera ocupa el lugar de 2 Celdas) y una dirección (hacia donde la punta está dirigiéndose). Debido a estas propiedades, el comportamiento de Barrera queda definido con los siguientes métodos:
  - *void colocar(int x, int y, Direccion d)*: "colocar" ha sido sobrescrita, pues ahora requiere saber la dirección hacia la que se dirige la punta de la Barrera.

    - *Direccion getDireccion() const*: Nos indica la dirección hacia la que se dirige la punta de la Barrera.
    - *const std::vector<int> &getPuntoMedio() const y const std::vector<int> &getPunta() const*: Nos indican la posición en la que se encuentran el punto medio y la punta de la Barrera, respectivamente.
- *Grafo*: Esta clase envuelve a un Tablero, y haciendo uso de sus Celdas y de las conexiones que crea entre ellas, lo hace lucir como un grafo. El único método importante que implementa es:
    - *bool hayCaminoMeta(int idJugador) const*: Nos indica si existe al menos un camino entre la Celda donde se encuentra el Jugador con identificador idJugador, hacia su meta.
- *Juez*: Es la clase que regula las reglas del juego. También es el punto principal de interacción desde afuera hacia el módulo Reglas. Para controlar las reglas del juego, mantiene un apuntador al Tablero del juego, y es quien se encarga de pedir las Jugadas a los jugadores durante su turno, y verificarlas antes de regresarlas al que las solicitó. Sus métodos principales son:
    - *Jugada siguienteJugada(int idJugador)*: Pide la siguiente Jugada al jugador con identificador idJugador. Revisa que cumpla con las reglas antes de regresarla.
    - *int hayGanador()*: Nos indica si hay un ganador, y en caso de haberlo nos indica su identificador.
- *Jugada*: Esta clase representa la Jugada que un Jugador quiere realizar. Tiene tres propiedades: "tipo" (MOVIMIENTO o BARRERA), "posicion" y (en caso de ser tipo BARRERA) "direccion". Contiene solamente *setters* y *getters* para cada una de sus propiedades.

El funcionamiento del proceso de pedir una nueva jugada y verificarla puede verse de manera simplificada con el siguiente diagrama de interacción:

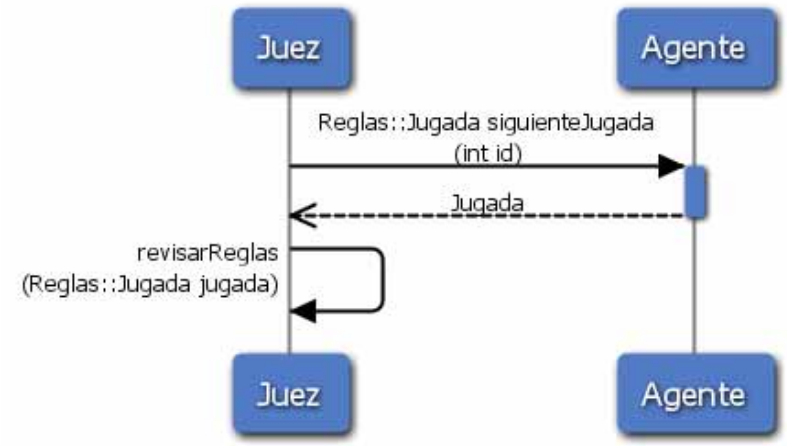


Figura 7: Diagrama de interacción con el módulo de Reglas.

Agente Inteligente.

Como ya se comentó, se decidió no implementar ningún algoritmo de aprendizaje para el Agente Inteligente, sino más bien usar el algoritmo minimax, para explorar el árbol del juego (es decir, el árbol que se crea al generar todos los posibles estados del tablero desde el estado actual, recursivamente para cada estado resultante hasta alcanzar un estado terminal) en búsqueda del camino que nos lleve a la victoria.

Minimax:

Minimax es un algoritmo recursivo que nos sirve para escoger el próximo movimiento en un juego basado en turnos. El algoritmo explora el árbol de juego, asignándoles un valor numérico a los nodos mediante una función de evaluación, empezando por los nodos terminales y propagándose hacia arriba hasta la raíz. La función de evaluación califica qué tan buena es la posición para un jugador cuando la alcanza.

Ya que en este caso es usado en un juego de dos jugadores, al empezar una búsqueda recursiva con este algoritmo, los hijos de la raíz son sólo movimientos hechos por el jugador (por ejemplo) 1. A todos los nodos que se encuentren en este nivel les llamamos “min”, pues en el caso de todos ellos, es turno de realizar un movimiento al jugador oponente, de quien queremos minimizar su éxito. De manera análoga, a los hijos de un nodo tipo min, les llamamos nodos “max” pues son movimientos de nuestro peón, y queremos maximizar el éxito en esos movimientos.

El algoritmo básicamente sigue los siguientes pasos:

- Generar el árbol de juego hasta llegar a un estado terminal.
- Calcular los valores de la función de evaluación para cada nodo terminal.
- Propagar hacia arriba los valores de los nodos. Si los nodos hijo son de tipo min, se propaga hacia arriba el valor mínimo de todos, pero si son de tipo max, se propaga el valor máximo.
- Al final el nodo raíz (de tipo max) ha recibido un valor propagado desde un nodo terminal, sólo falta moverse hacia el nodo que nos propagó ese valor, efectivamente avanzando sobre el camino descubierto.

[9]

Sin embargo, este algoritmo nos es imposible de usar: teniendo en cuenta que un peón tiene la posibilidad de moverse a cinco celdas distintas en el mejor de los casos, o de poner una barrera en 128 formas distintas, cada nodo tiene aproximadamente 133 hijos. Esto se vuelve un problema al tratar de generar el árbol de juego, operación que tiene una complejidad exponencial: para un árbol con profundidad 10, se habrían generado 133^{10} nodos terminales (sin contar a todos los demás nodos no terminales), y si suponemos que usar la función de evaluación sobre cada nodo nos toma 0.001 segundos, nos llevaría aproximadamente $5.568 * 10^{10}$ años evaluarlos a todos.

Es por esto que decidimos usar una condición de paro, permitiéndonos detener la búsqueda a la profundidad que decidiéramos, aunque tal vez arriesgando posibles descubrimientos a más profundidad en el árbol.

El siguiente paso fue encontrar una función de evaluación para cada nodo terminal del árbol de juego. Nuestra función de evaluación recibe el identificador del jugador que realizó la jugada que lleva a dicho nodo, así que si dicho nodo le es favorable al jugador en cuestión, la función debería tener un valor alto, mientras que si le es menos favorable, debería tener un valor más bajo. El único valor significativo que pudimos encontrar para hacer dicha evaluación fue la magnitud (el número de celdas) del camino más corto entre la posición actual del peón, hasta su meta más próxima. Pero si usamos este valor (llamémosle c1) como calificación, se queda un problema: si la evaluación es buena (su mejor camino a la meta es corto), nos da un número más bajo, lo cual podría considerarse como una calificación desfavorable. Nuestra solución entonces es que la calificación sea el número total de celdas (81) restándole c1; pues nunca habrá un camino que tenga más de 81 celdas, y si c1 es bueno, la calificación será más alta en relación a la calificación si c1 es no tan bueno(es decir, un camino más largo). A esta calificación le llamaremos v1.

Dado que el juego también se trata sobre bloquear el paso al otro jugador, también es importante considerar en nuestra función de evaluación qué tanto se empeora el camino más corto hacia la meta para el oponente (c2). Si para un movimiento realizado c2 es bajo, es considerado desfavorable pues significa que hicimos más corto el camino del adversario, mientras que si c2 es alto, es considerado favorable. En este caso también sufrimos del mismo problema que con c1, así que usaremos un valor v2, que es igual a 81 menos c2.

Así, tenemos una función con dos parámetros a considerar.

Parámetro	Valor	Signo	Peso
V1	81 – longitud delcamino más corto a la meta para el jugador	+	W1
V2	81 – longitud delcamino más corto a la meta para el oponente	-	W2

Dejamos abierta la posibilidad de que cada uno de estos parámetros tuviera una importancia distinta, así que le colocamos “pesos” (w1 y w2) a cada calificación.

Con estos valores, podemos ir armando un prototipo de nuestra función de evaluación:

```
evaluación () => int:
      regresa W1 * V1 + W2 *V2;
```

Sin embargo, con pruebas manuales, se obtuvieron mejores resultados si cambiábamos los pesos para el caso de que quisiéramos hacer un movimiento de tipo BARRERA o uno de tipo MOVIMIENTO. El pseudocódigo final es:

```
evaluación () => int:
      si tipo de jugada es MOVIMIENTO:
            W1 <- 1.0, W2 <- 1.5;
      si tipo de jugada es BARRERA:
            W1 <- 2.5, W2 <- 0.5;

      regresa W1 * V1 + W2 *V2;
```

A*:

Para resolver el problema de encontrar el camino más corto a la meta para cada jugador, decidimos usar el algoritmo A* (o A estrella o astar), que es un algoritmo para buscar el camino más corto dentro de un grafo. Es una extensión del algoritmo de Dijkstra, pero usa heurísticas para lograr un mejor rendimiento en cuanto a tiempo. Al igual que el algoritmo de Dijkstra, nos asegura que de haber un camino a la meta, encontrará el más corto.

Usa una función heurística (f(x)) para determinar el orden en que se visitan los nodos del grafo, esta heurística es la suma de dos funciones:

- Una función que estima el costo del nodo de inicio al actual (g(x)).
- Una función que estima el costo de la distancia del nodo actual al nodo meta.

Mientras A* atraviesa el grafo, sigue un camino desde el camino más barato conocido, manteniendo una cola de prioridad ordenada según los valores f(x) de las celdas de un camino alternativo. Si en cualquier punto un segmento del camino que está siendo atravesado tiene un costo más alto que otro segmento encontrado, se abandona el segmento de costo más alto y en su lugar se atraviesa el segmento de costo más bajo. Este proceso continúa hasta que se alcanza la meta.

(Russel & Norvig, 2003)

El pseudocódigo del algoritmo básico es simple:

```
ABIERTOS = cola de prioridad que contiene a INICIO
CERRADOS = conjunto vacío
mientras la prioridad más baja en ABIERTOS no es META:
  actual = pop el elemento de prioridad más baja en ABIERTOS
  agregar actual a CERRADOS
  por cada vecino de actual:
    costo = g(actual) + costo_de_movimiento(actual, vecino)
    si vecino está en ABIERTOS y su costo es menor a g(vecino):
      remover vecino de ABIERTOS, pues el nuevo camino es mejor
    si vecino está en ABIERTOS y su costo es menor a g(vecino):
```

```
      remover a vecino de CERRADOS
    si vecino no está en ABIERTOS y vecino no está en CERRADOS:
      asignar g(vecino) a costo
      agregar vecino a ABIERTOS
      poner su prioridad igual a g(vecino) + h(vecino)
      poner a actual como padre de vecino
reconstruir el camino en reversa desde la meta al inicio usando los
apuntadores a los padres.
```

Pero la implementación se vuelve complicada pues hay mucho con lo que lidiar (las estructuras de datos y todo el manejo de errores). Por eso inicialmente se tomó la decisión de implementar el algoritmo en Python, y dicha implementación aún puede verse en http://code.google.com/p/quassimodo/source/browse/bin/astar_algoritmo.py?spec=svn279ee2eaf0f18e651cd27f15ae21cd7f752f4502&r=279ee2eaf0f18e651cd27f15ae21cd7f752f4502. Pero la implementación resultó bastante lenta, principalmente por la naturaleza interpretada de Python, tanto así que cada búsqueda llevaba varios segundos.

Como no podíamos permitir tanto tiempo para una sola búsqueda (pues necesitamos generar miles de ellas para Minimax) recurrimos a usar una implementación ya hecha de A*, encontrada en http://www.grinninglizard.com/MicroPather/ . De esta manera podemos realizar miles de búsquedas en unos cuantos segundos.

Módulo "Grafico"

Este módulo es el encargado de todo lo relacionado con cuestiones de interfaz de usuario y modelos en 3D. Cada clase de este módulo, tiene la responsabilidad de dibujarse a sí mismo, así como también el moverse, girar, escalarse, etc.

En este módulo se utiliza el motor Irrlicht [4] el cual permitirá la creación de los objetos que serán dibujados en pantalla, la manipulación y eliminación de éstos.

Para explicar el módulo Grafico lo dividiremos en dos partes: interfaz de usuario y modelos 3D.

Interfaz de Usuario

Esta parte, se encarga del diseño de la GUI presentada al usuario, se arman los cuadros de diálogo, el tamaño y colocación de los botones, imágenes y textos. También esta parte de interfaz se encarga de cargar el skin[5] que tendrá la GUI, (El skin utilizado en el programa fue obtenido de la comunidad de desarrolladores de Irrlicht) así como el menú principal de la aplicación y los elementos que éste mostrará.

[4] Irrlicht http://irrlicht.sourceforge.net/ es un motor 3d gratuito y de código abierto.

[5] skin también llamado theme o tema, es una serie de elementos gráficos que, al aplicarse sobre un determinado software, modifican su apariencia externa.

Las clases más importantes que define o implementa son las siguientes:

- *GUI*: Es la clase que se encarga de cargar el skin que tendrá y de dibujar todo lo relacionado con la GUI: ventanas, textos, cajas de mensajes, colocación de los botones de la partida, el cuadro de créditos etc.

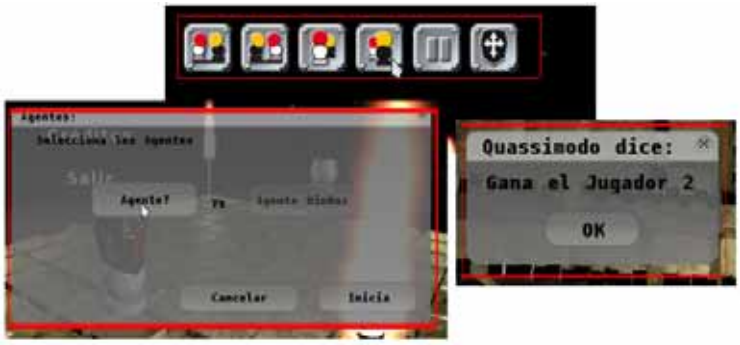


Figura 8: Imágenes de algunas GUI.

- *Menu* : Clase encargada solamente de dibujar y eliminar el menú principal, véase Figura 9.

Figura 9: Imagen del menú inicial.

- *ManejadorGUI*: Esta clase se encarga de manejar las dos clases anteriores para que se dibujen y sean eliminadas en el momento indicado. La mayoría de los métodos aquí implementados son del tipo: dibuja<elemento>, drop<elemento> que solamente llaman a los métodos de las dos clases descritas anteriormente.

Modelos 3D

En modelos 3D, se realizaron envoltorios de las clases Jugador, Tablero, Celda, y Barrera, encontrados en el módulo de Reglas. Se envolvieron sólo estas clases, pues son las únicas de ese módulo que se mostrarán en pantalla con modelos en 3D.

Explicaremos un poco más algunas clases:

- *Jugador*: Clase que como su nombre lo indica se encarga del jugador, dependiendo del número del jugador que sea se cargará el modelo correspondiente. Algunos métodos importantes son:
    - *Mover*: se encarga de realizar la animación del movimiento para el jugador y actualizar la posición de éste.
    - *Gira*: rota el jugador sobre su eje los grados que uno le especifique.

Figura 10. Imagen del Jugador 1

Figura 11. Imagen del Jugador 2

- *Barrera*: Clase que como su nombre lo indica se encarga de la Barrera. Algunos métodos importantes son:
    - *Colocar*: Realiza la animación de colocar la barrera y actualiza su posición.
    - *GiraNorte*: gira la barrera en dirección norte.
    - *GiraEste*: gira la barrera en dirección este.

Figura 12. Imagen de la Barrera

- *Antorcha*: Clase que se encarga de dibujar una antorcha con fuego que funcionará como la fuente de luz en nuestra escena mostrada en pantalla.

Figura 13.Imagen de la Antorcha

- *Celda:* Clase que se encarga de dibujar la celda del tablero.

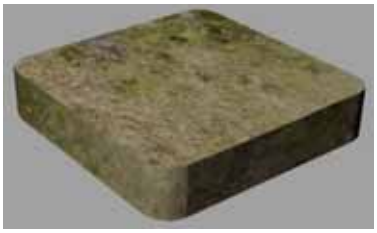


Figura 14. . Imagen de la Celda

- *Tablero:* Clase que será la encargada de dibujar el tablero, así como el mandar a dibujar a las celdas que se encontrarán en el tablero.

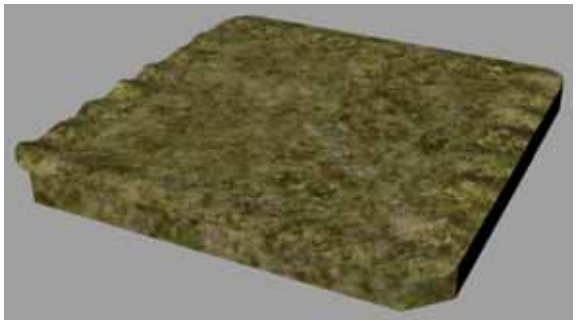


Figura 15. Imagen del Tablero

Se creó además una cámara que se utilizará en el transcurso de la partida, y un animador, que es con el que se moverán los jugadores de la partida en la escena.

A estas dos partes del módulo, se le agrega una clase más: la clase Video. Esta clase se encarga de detectar la resolución de la pantalla, así como también el tipo de *driver* soportado para la parte de 3D (OpenGL, Directx8, DirectX9).

"Aplicacion".

Este paquete no es un módulo como los anteriores, sino la parte del proyecto que une todas las piezas descritas hasta el momento en un ejecutable. Es decir, es el punto de entrada que pone todos los paquetes anteriores a funcionar juntos.

Aquí tenemos las clases principales:

- *EventReceiver*: El Recibidor de eventos, que es como su nombre lo dice el que obtiene todos los eventos del teclado, y del ratón, los maneja y realiza la acción correspondiente para cada uno de estos.

- *ManejadorJuego*: Esta clase es la que controla todo el juego, inicia y termina una partida, carga a los agentes que tendrán los jugadores, coloca la GUI cuando es necesario, etc.

- *Partida*: Esta clase es la clase padre de los demás tipos de partida que se pueden ejecutar. Aquí tenemos los métodos principales con los que se puede realizar una partida:

    - *iniciarPartida*: método que avisa a los jugadores que la partida está por iniciar.

    - *siguienteJugada*: pide una jugada al jugador que le toca realizar una. De ser esta jugada valida, actualiza el tablero.

    - *actualizarTablero*: se encarga de realizar la jugada en el tablero de la partida para que se vean reflejados los cambios.

- *PartidaGrafica*: Clase que hereda de la clase Partida, se encarga de realizar la partida del juego en modo gráfico. Cuando es necesario, la clase manda a los objetos a que se actualicen (esto es, que se muevan de lugar, giren, etc.) para que se muestre el cambio en pantalla.

- *PartidaConsola*: Clase que hereda de la clase Partida, se encarga de realizar la partida del juego en modo consola. Aquí cada vez que se actualiza la posición de algún objeto, se tiene que mandar a imprimir el tablero para que los cambios hechos puedan ser mostrados.

- *Aplicacion*: Esta clase se encarga de manejar toda la aplicación, inicializa los motores gráficos, el recibidor de eventos, selecciona el driver de video y contiene el ciclo principal de juego, algunos de los métodos más importantes son:

- *run*: inicia el ciclo principal de la aplicación, dependiendo del cómo se ejecutará el programa, si en modo consola o en modo gráfico.
- *loopGrafico*: ciclo principal de la aplicación para ejecutarse en modo gráfico.
- *loopConsola*: ciclo principal de la aplicación para ejecutarse en modo consola.
- *nuevoJuego*: reinicializa el manejador del juego. Limpia la pantalla de las barreras colocadas en la aplicación e inicia el menú principal.

Aunque el programa no tiene demasiada interacción con el usuario (pues el usuario como tal no juega una partida), sí es necesario saber cómo el usuario se comunicará con el programa y que éste realice lo que pida. Ésta interacción se muestra en el siguiente diagrama:

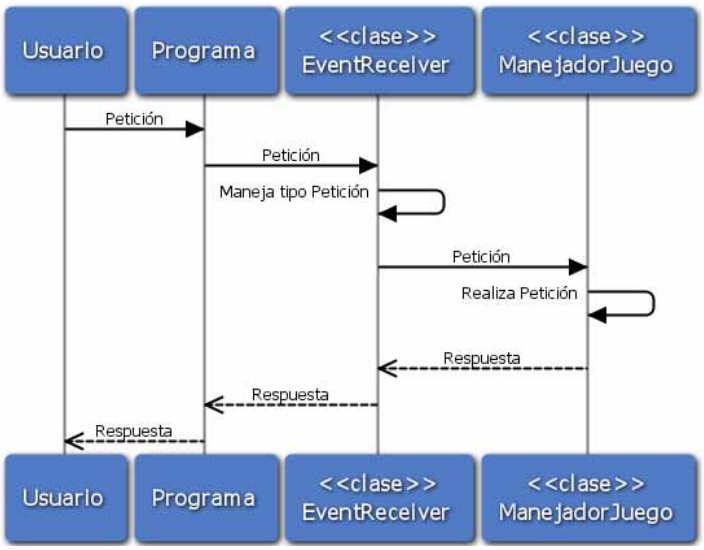


Figura 16: Diagrama de interacción del usuario con el módulo "Grafico".

Conclusiones.

El propósito de este proyecto fue el de crear una plataforma de programación de Agentes Inteligentes y poder ver sus partidas con gráficas tridimensionales para ayudar al proceso de enseñanza-aprendizaje de varios temas relacionados a la programación.

La aplicación fue diseñada y construida con el uso exclusivo de herramientas y bibliotecas libres, tales como:

- NetBeans. El entorno integrado de desarrollo que usamos para construir toda la aplicación.
- gcc: colección de compiladores libres, incluye un compilador de C++: g++.
- Python: Lenguaje (e intérprete del mismo) de naturaleza interpretada y sintaxis simple, leída casi como seudocódigo.
- Irrlicht: motor gráfico 3d escrito en C++.
- boost: colección de bibliotecas multiplataforma para C++.

Desde antes de iniciar el proyecto se tuvo contemplada la implementación de AIs como parte del mismo para tener contrincantes por defecto dentro de la aplicación. Al principio se tenía pensado usar algún algoritmo de aprendizaje para que el AI mejorara poco a poco con cada partida jugada, pero la idea se descartó una vez iniciado el desarrollo de dicho AI.

Después de que se nos hizo saber que tratar de implementar algún algoritmo de aprendizaje estaba fuera de los alcances de un proyecto terminal, se decidió usar el algoritmo Minimax, un algoritmo de exploración del árbol del juego.

Para implementar Minimax se hizo uso de otro algoritmo de apoyo, llamado A*: un algoritmo de búsqueda de caminos dentro de un grafo que nos asegura encontrar la ruta más corta entre dos puntos de éste en caso de que exista. Minimax hace uso de A* para poder calificar los nodos del árbol del juego y con esta ayuda decidir qué jugada realizar maximizando el beneficio propio (acercándose a la meta) mientras se minimiza el beneficio del enemigo (evitando acortar el camino más corto del enemigo, hacer más largo el mismo).

Se desarrollaron tres módulos principales, cada uno separado como una biblioteca de carga dinámica para propiciar el bajo acoplamiento entre nuestras clases. Los módulos son:

- "Reglas"
- "Scripting"
- "Grafico"

El módulo *"Reglas"* define las clases y comportamiento más básico del sistema: clases como Tablero, Celda, Jugador, Jugada, Agente, etc. que representan el comportamiento de cada objeto del juego. También, como su nombre lo indica, este módulo funge como motor de reglas con el uso de la clase Juez, que se encarga de decidir si una jugada es válida para cierto jugador.

El módulo *"Scripting"* realiza toda la comunicación entre Python y C++ necesaria para poder definir nuevas clases en Python derivadas de Agente, y usarlas como clases comunes en C++; esto nos permite crear los AIs en Python y usarlos dentro del código de C++ (sin necesidad de recompilar la aplicación).

El módulo *"Grafico"* se encarga de todo lo que se mostrará en pantalla, como son los modelos en 3D de las piezas tales como: los jugadores, las barreras, las celdas, etc.; así como también de la parte de la interfaz gráfica de usuario, ventanas y el menú principal. Aquí también se encuentra la clase que permitirá al programa saber en qué resolución se deberá mostrar la aplicación. La resolución de pantalla por default es 1024x 768. En caso de que la pantalla de la computadora en donde se ejecute el programa tenga una menor resolución, ésta será detectada y utilizada en su lugar.

Posibles Mejoras y Adiciones.

Aunque cumplidos todos los objetivos (con sus respectivas adaptaciones), el desarrollo del proyecto no estuvo libre de problemas. También hubieron varios aspectos que, aunque fuera de los objetivos del proyecto, quisimos incluir y no se pudieron terminar.

- Uno de los problemas más importantes fueron las fugas de memoria, que se causan al no liberar la memoria usada por un objeto antes de perder las referencias a su apuntador en la memoria. La forma más fácil de eliminar dichas fugas, sería usando solamente las clases *auto_ptr* (dentro de la biblioteca estándar de C++) y *smart_ptr* (parte de boost) en lugar de apuntadores simples (o tipo C); con estas herramientas podemos deshacernos de la mayoría de las fugas de memoria, pues estas clases se encargan de tareas tales como liberar los recursos cuando el apuntador sale de ámbito o llevar conteos de referencias a los objetos, respectivamente.
- En cierto punto del proceso de desarrollo, se quisieron implementar hilos, para que los AIs usaran su propio hilo en la ejecución de sus rutinas, evitando que la pantalla gráfica se congelara. No fue posible por cuestiones de tiempo, y queda pendiente para próximas iteraciones del proyecto.
- Igual que en el punto anterior, a pesar de que se implementó la funcionalidad de skins, éstas tienen que ser configuradas a mano en el archivo de configuración de la aplicación. La mejora propuesta es permitir la configuración de las skins desde la propia aplicación.
- Proponemos la creación de varias skins para ser usadas dentro del juego.
- Otro elemento que quedó descartado por falta de tiempo fue la inclusión de música para acompañar las partidas. Se propone la implementación de dicha funcionalidad haciendo uso de alguna biblioteca de alto nivel para facilitar la tarea.
- También sería una interesante mejora agregar más lenguajes de *scripting* para poder escribir los AIs en ellos. La arquitectura del módulo "Scripting" es bastante sencilla y está hecha especialmente con el propósito de agregar más entornos de *scripting* rápidamente implementando tres interfaces.

Bibliografía.

[1]Artículo sobre Quoridor, Wikipedia. (7 de Agosto de 2010). Recuperado el 27 de Agosto de 2010, de http://en.wikipedia.org/wiki/Quoridor

[2]Glendenning, L. (Mayo de 2005). *Mastering Quoridor.* Recuperado el 27 de Agosto de 2010, de http://hyperion.cs.washington.edu/attachments/15/glendenning_ugrad_thesis.pdf

[3]A Quoridor Playing Agent. (21 de Junio de 2006). Recuperado el 27 de Agosto de 2010, de http://www.unimaas.nl/games/files/bsc/Mertens_BSc-paper.pdf

[4]Quoridor. (5 de Abril de 2007). *Martijn van Steenbergen*. Recuperado el 27 de Agosto de 2010, de http://martijn.van.steenbergen.nl/projects/quoridor/

[5]Dvory, O. (12 de Julio de 2007). *Corridor en Sourceforge*. Recuperado el 27 de Agosto de 2010, de http://sourceforge.net/projects/corridor/

[6]Gamerz network for Quoridor. (s.f.). Obtenido de http://www.gamerz.net/pbmserv/quoridor.html

[7]Gigamic. (2010). *Quoridor The Amazing Maze*. Recuperado el 27 de Agosto de 2010, de http://www.quoridor.net/

[8]Bandeira, F. A. (s.f.). *Quoridor Digital*. Obtenido de http://paginas.fe.up.pt/~ei01036/quoridor

[9]Russel, S. J., & Norvig, P. (2003). *Artificial Intelligence, A Modern Approach.* New Jersey: Pearson Education. Páginas 97 a 101 y 162 a 167.

[10]S.A., Gigamic. (2010). *Quoridor - The Amazing Maze*. Recuperado el 27 de Agosto de 2010, de http://www.quoridor.net/