34ª Jornadas Argentinas de Informática e Investigación Operativa

"Algoritmos de búsqueda heurística en tiempo real. Aplicación a la navegación en los juegos de vídeo"

Autor: **Fernández, Martín Osvaldo**
e-mail: *martinosvaldofernandez@gmail.com*

Director: **Felice, Laura**
e-mail: *lfelice@exa.unicen.edu.ar*

Categoría: **Trabajos de Cátedra**
Área: **Algoritmos, Lenguajes y Programación**

Asignatura:
Análisis y diseño de algoritmos II
2º año de la carrera de Ingeniería de Sistemas
Facultad de Ciencias Exactas
Universidad Nacional del Centro de la Provincia de Buenos Aires
Tandil

Algoritmos de búsqueda heurística en tiempo real. Aplicación a la navegación en los juegos de vídeo.

Resumen

Este trabajo se inició en el marco de un curso de introducción al análisis y diseño de algoritmos, dictado en el 2º año de una carrera de informática. La motivación fue estudiar cómo se extienden los conceptos básicos de búsqueda para adaptarlos a los requerimientos de los sistemas de tiempo real. Como caso de estudio se eligió la navegación en los juegos de vídeo, por ser un problema que se resuelve naturalmente mediante la búsqueda. Se consideran los juegos en tiempo real en los que el entorno es demasiado grande o el terreno es dinámico, de modo que las estrategias básicas de búsqueda resultan inadecuadas. En este contexto, se analizan e implementan algoritmos de búsqueda heurística en tiempo real los cuales fueron concebidos para resolver problemas con estas características.

1. Introducción

La constante evolución de los juegos de vídeo ha llevado a que la inteligencia artificial constituya uno de los aspectos más importantes; es fundamental que los agentes (entidades autónomas) controlados por la computadora se comporten en forma inteligente. Un problema característico es la *navegación*, que consiste en determinar el camino más conveniente entre una posición inicial y una posición de destino. Si bien el planteo del problema es sencillo, el mismo está lejos de ser trivial debido a la creciente complejidad de los entornos simulados y los requerimientos de tiempo real de los juegos modernos [*Nareyek- 04*].

La búsqueda es una de las técnicas más utilizadas para resolver los problemas de *pathfinding* [1] o planificación que se presentan en la inteligencia artificial en los juegos de vídeo. En particular, la búsqueda es utilizada para resolver el problema de la navegación. De los distintos tipos de algoritmos de búsqueda (*figura 1*), los algoritmos búsqueda heurística completa se encuentran ampliamente difundidos. Sin dudas, el algoritmo A* es el algoritmo de búsqueda heurística más popular [*Stout- 96*].

1 Es importante mencionar una ambigüedad en la utilización del término pathfinding en la bibliografía referenciada. Pathfinding se utiliza para hacer referencia tanto al problema general de encontrar una serie de pasos para llegar a un estado objetivo partiendo de un estado inicial, en el contexto de un problema cualquiera, como al problema específico de la navegación.

Los algoritmos heurísticos tradicionales muestran limitaciones importantes cuando el espacio de búsqueda es demasiado grande o existen factores dinámicos. En la navegación, de los juegos de vídeo en tiempo real, este problema aparece cuando las rutas a determinar son muy largas, el terreno es modificable o existen muchos objetos móviles. Bajo esas condiciones, los algoritmos de búsqueda básicos no pueden responder en el tiempo requerido y resultan inadecuados. De esta forma, surge la necesidad de desarrollar nuevas estrategias de búsqueda, que se adapten a los requerimientos de tiempo real de los juegos de vídeo y resuelvan adecuadamente caminos en condiciones de incertidumbre sobre terrenos de gran extensión [*Laird,Pottinger- 00*].

Actualmente existen dos clases de algoritmos de búsqueda que se adecuan a la resolución de problemas con las características mencionadas: los algoritmos de *búsqueda heurística incrementales* y los algoritmos de *búsqueda heurística en tiempo real*. Los algoritmos incrementales utilizan información de búsquedas previas para encontrar soluciones a problemas similares posiblemente más rápido que realizando cada búsqueda partiendo de cero [*Koenig- 04*]. Por otra parte, los algoritmos de búsqueda en tiempo real alternan planificación y ejecución del plan y restringen la planificación a la parte del dominio inmediata al estado actual del agente [*Koenig- 01*].

Aunque ambas técnicas se adaptan a los requerimientos de los juegos en tiempo real, la idea de dividir la planificación en etapas de duración limitada parece ser la más difundida. Esto se debe a que los algoritmos de tiempo real se basan en las estrategias de búsqueda de profundidad limitada utilizadas en los problemas de juegos de dos jugadores como el ajedrez, las damas y el Othello [*Brockington- 00*].

Este trabajo se desarrolla considerando los algoritmos de búsqueda heurística en tiempo real. En la sección 2 se presentan los fundamentos de la búsqueda en tiempo real. Esta discusión es carácter general y no se habla de ningún algoritmo en particular. En la sección 3 se describen los algoritmos implementados; además, se presentan los resultados de las pruebas realizadas para analizar experimentalmente el comportamiento de estos algoritmos, en un entorno similar al de los juegos de vídeo en tiempo real. Una parte esencial del proyecto consistió en el desarrollo de la aplicación que proporcionó la plataforma de prueba para los algoritmos. Todas las imágenes de los ejemplos presentados en las figuras fueron generadas con la misma. Los detalles del diseño y la implementación de esta aplicación se encuentran en el apéndice. Finalmente en la sección 4 se encuentran las conclusiones y comentarios finales.

2. Fundamentos de la búsqueda en tiempo real

La búsqueda es una técnica para resolver problemas cuya solución consiste en una serie de pasos que frecuentemente deben determinarse mediante la prueba sistemática de las alternativas. Desde los inicios de la Inteligencia Artificial, la búsqueda se ha aplicado en diversas clases de problemas como juegos de dos jugadores, problemas de satisfacción de restricciones y problemas de pathfinding de un único agente [*Korf- 00*].

En la *figura 1*, se presenta una clasificación de los algoritmos de búsqueda, haciendo hincapié en su modo de operación, y se incluyen los algoritmos más representativos de cada clase [*Bender- 96*].

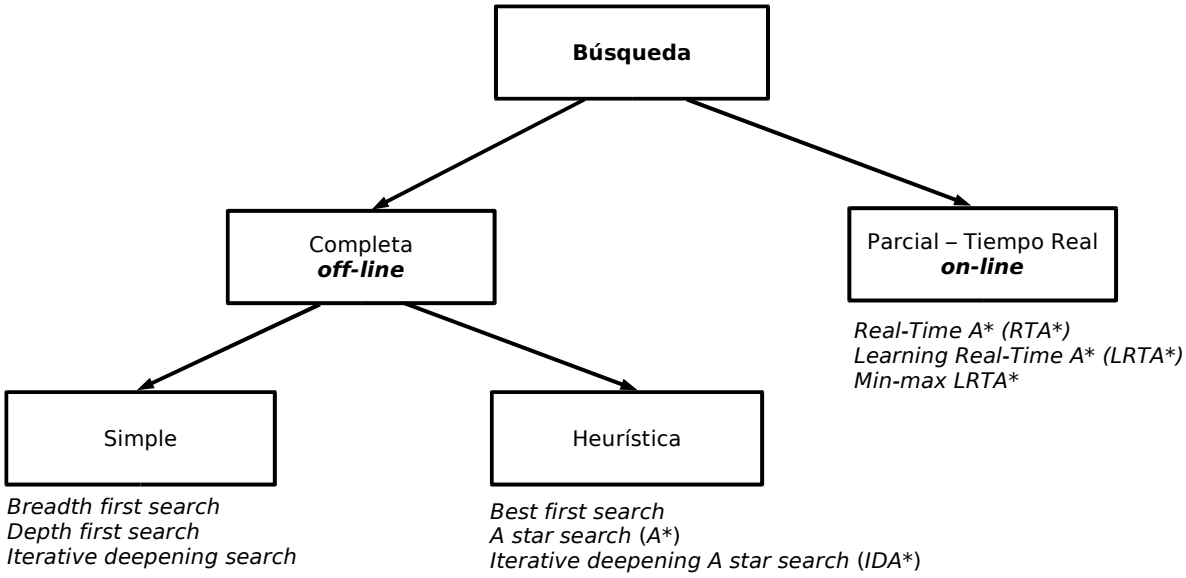


Figura 1: *Clasificación de los algoritmos de búsqueda.*

Los algoritmos de búsqueda completa[2] tradicionales se caracterizan por su modo de operación *off- line*, que determina que debe encontrarse la solución entera en una única etapa de planificación, antes de comenzar la ejecución de los pasos o acciones que la componen (*figura 2*). Es posible utilizar este esquema de búsqueda si se cuenta con la información suficiente sobre el problema y el tamaño del espacio de búsqueda permite el cálculo de la solución con los recursos computacionales disponibles.

Figura 2: *Modo de operación off- line.*

2 La búsqueda completa puede realizarse sin información o utilizando conocimiento del dominio del problema, lo que da lugar a los algoritmos de búsqueda simple y búsqueda heurística respectivamente.

Hacia fines de los 80' se comenzó a utilizar conceptos de la búsqueda aplicada a juegos de dos jugadores en el desarrollo de estrategias para resolver problemas de pathfinding de un único agente. La idea fundamental que se tomó fue la de intercalar etapas de planificación (donde se realizan búsquedas de profundidad limitada) con etapas de ejecución (*figura 3*). Este modo de operación se denomina *on-line* y da lugar los algoritmos de búsqueda de tiempo real[3]. Estos algoritmos de búsqueda general son capaces de responder a los requerimientos de las aplicaciones de tiempo real sobre espacios de búsqueda grandes y en condiciones de incertidumbre. Algunos algoritmos desarrollados con estas características son el *Real-Time A**, el *Learning Real-Time A** y el *Min-Max Learning Real-Time A**.

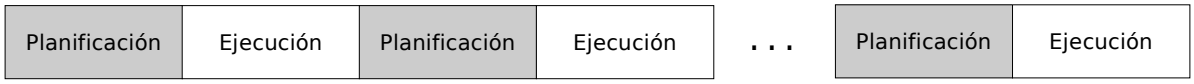


Figura 3: *Modo de operación on-line.*

2.1. Descripción de las fases de la búsqueda heurística en tiempo real

De la discusión anterior se ve que el proceso de búsqueda en tiempo real se lleva a cabo alternando dos modos de operación bastante distintos. El primer modo es de planificación, donde se simulan los movimientos del agente para evaluar los resultados de las acciones inmediatas. Luego de terminar la búsqueda, en este espacio acotado de posibilidades, el agente pasa al modo de ejecución y realiza efectivamente un movimiento. El proceso continúa alternando fases de planificación y ejecución hasta que se alcanza el estado objetivo [*Korf-90*].

2.1.1. La fase de planificación

Durante la etapa de planificación, es fundamental restringir la profundidad de búsqueda para poder responder dentro de los límites de tiempo impuestos. Si bien el tiempo de ejecución de los algoritmos de búsqueda es exponencial respecto a la frontera de búsqueda, la misma está acotada por una constante y por lo tanto también lo está el tiempo de ejecución. Dado que la profundidad de búsqueda es limitada, es necesario contar con un método para evaluar los estados no-objetivos. Es esencial para el funcionamiento de esta estrategia la utilización de heurísticas que permitan estimar el costo de alcanzar un objetivo desde un estado intermedio.

3 Los algoritmos de búsqueda en tiempo real también se conocen como algoritmos de búsqueda de profundidad limitada, algoritmos de búsqueda parcial o local. No existe aún un término común para denominar a estos algoritmos, aunque recientemente se ha propuesto el de “búsqueda centrada en el agente” [*Koenig-01*] (que abarca a todos los algoritmos que intercalan búsqueda y ejecución, incluyendo aquellos que no responden en tiempo constante que es el requerimiento de los algoritmos de tiempo real).

Utilizando una función heurística, extendida mediante búsqueda, es posible construir una función de evaluación de estados que incluya el costo para alcanzar un estado determinado así como una aproximación sobre el costo para encontrar el objetivo desde ese estado (*figura 4*). Se definen los siguientes costos:

- ***f(n)*** es el costo heurístico asociado a un estado *n*
- ***g(n)*** es el costo real para alcanzar un estado *n* a partir del estado actual [4]
- ***h(n)*** es el costo heurístico para alcanzar un objetivo desde el estado *n*

La relación que existe entre los costos es:

$$f(n) = g(n) + h(n)$$

Una función heurística *h* es admisible si las estimaciones arrojadas nunca sobrestiman los valores reales de los estados evaluados. La admisibilidad es una propiedad importante de la función heurística, y frecuentemente es requerida por los algoritmos de control para garantizar un funcionamiento correcto.

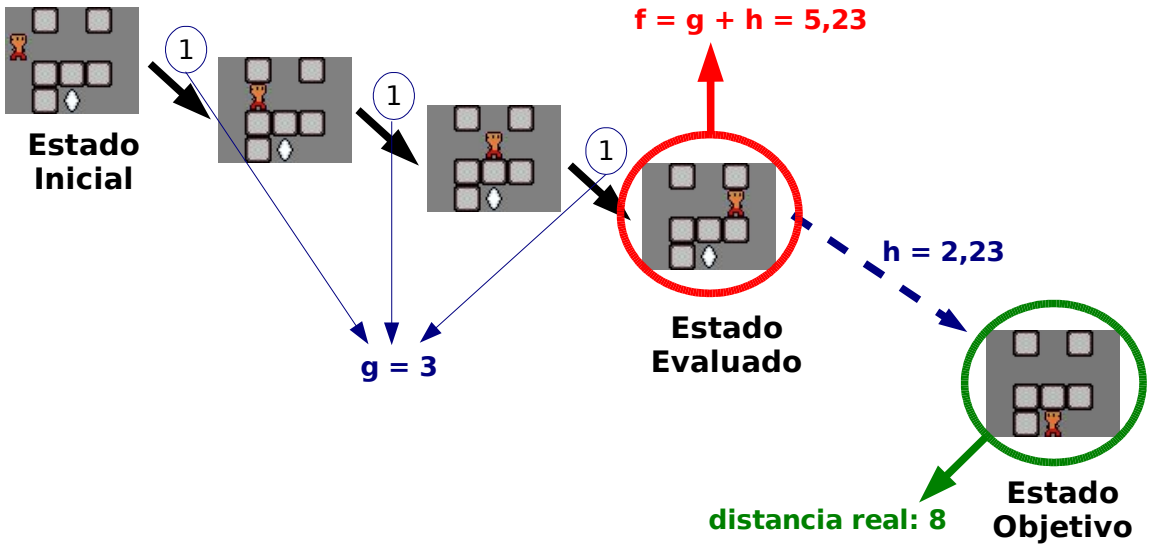


Figura 4: *Los componentes de la función de evaluación de estados para un ejemplo sencillo Los gráficos corresponden a los estados alcanzados mediante los movimentos simulados. La heurística utilizada es la distancia euclídea y la profundidad de búsqueda es 3.*

Al diseñar el algoritmo de planificación se deben realizar concesiones entre los recursos computacionales invertidos y la precisión de las evaluaciones devueltas. La función heurística *h* en conjunto con la búsqueda de profundidad limitada se consideran una sola función heurística *f*. La precisión de los valores obtenidos durante la planificación aumenta al incrementar la profundidad búsqueda, pero al mismo tiempo el costo de ejecución aumenta exponencialmente. De esta forma, se obtiene toda una gama de funciones de evaluación heurísticas que intercambian precisión por costo [*Korf- 00*].

4 En general, el costo entre dos estados vecinos puede ser fijado arbitrariamente. En adelante se supondrá un costo de una unidad para pasar de un estado a cualquier estado vecino.

2.1.2. La fase de ejecución

Uno de los principios más importantes de la búsqueda en tiempo real es la utilización de la estrategia del menor compromiso para los movimientos. Es decir, la información obtenida en cada etapa de planificación sólo se utilizará para determinar el próximo movimiento. La razón es que luego de ejecutar la acción, se supone que la frontera de búsqueda se expandirá, lo cual puede llevar a una elección para el segundo movimiento diferente a que la que arrojó la primera búsqueda.

Sin embargo, para realizar una secuencia de decisiones no es suficiente con la información devuelta por el algoritmo de planificación. La estrategia básica de repetir el algoritmo de planificación para cada decisión resulta inadecuada al ignorar la información relacionada con los estados anteriores. El problema radica en que, al volver a un estado previamente visitado, se entrará en un ciclo infinito. Esto es algo que sucederá con frecuencia, debido a que las decisiones se basan en información limitada y por lo tanto direcciones que al principio parecían favorables pueden resultar equivocadas al reunir más información durante la exploración. En general, se desea evitar entrar en ciclos infinitos y a la vez permitir volver a estados ya visitados cuando parezca favorable.

El principio para resolver el problema del control de la etapa de ejecución es: *sólo se debería regresar a un estado visitado cuando la estimación de resolver el problema desde ese estado más el costo de volver al mismo es menor que el costo estimado de seguir hacia adelante desde el estado actual*. Los primeros algoritmos en implementar esta solución fueron el *Real-Time A** y una variante llamada *Learning Real-Time A** [Korf-90].

3. Análisis e implementación de los algoritmos RTA* y LRTA*

El estudio se centró en el análisis e implementación de los dos algoritmos más conocidos para el control búsqueda de tiempo real, el *Real-Time A** (*RTA**) y el *Learning Real-Time A** (*LRTA**); como la función de evaluación o algoritmo de planificación se implementó el algoritmo *Minimin*. Con este fin, se desarrolló una aplicación que proporciona un entorno similar al de un juego en tiempo real. El entorno de navegación consiste en un espacio discreto de dos dimensiones en forma de grilla, denominado *gridworld*. Al utilizar un entorno tan simple se evita el *análisis de terreno*, que consiste en la extracción de información útil y manejable de un modelo complejo de terreno [*Pottinger-00*].

3.1. Los algoritmos utilizados

En esta sección se describen y presentan las características principales de los algoritmos que se utilizaron para la planificación y el control de la navegación.

3.1.1. El algoritmo de planificación: *Minimin*

Minimin [*Korf- 90*] es un algoritmo búsqueda de profundidad limitada que se emplea durante la etapa de planificación. La búsqueda se hace a partir del estado actual hasta una profundidad determinada y en los nodos de la frontera se aplica la función de evaluación *f*. El valor de cada nodo interno es el mínimo de los valores de los nodos de la frontera del subárbol debajo del nodo (*figura 5*).

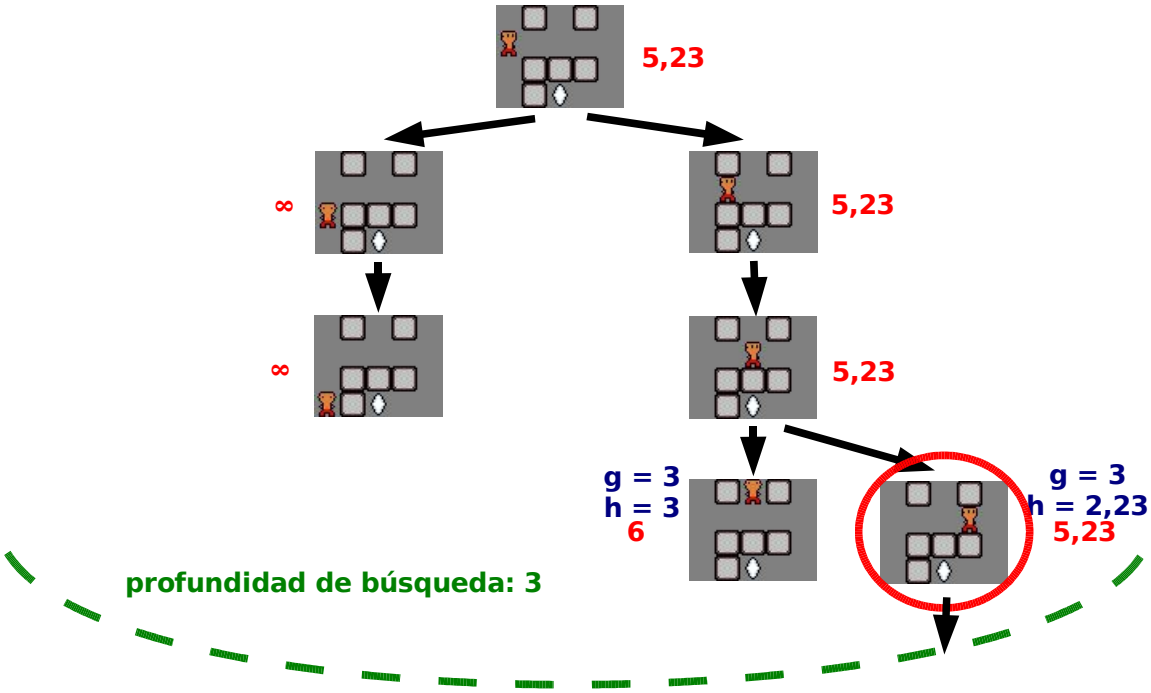


Figura 5: *Ejemplo de la asignación de valores a los estados intermedios realizada por el algoritmo Minimin. Los gráficos corresponden a los estados alcanzados mediante los movimentos simulados. La heurística utilizada y el estado objetivo son los mismos que los del ejemplo de la figura 4.*

Si la función heurística *h* es monótona no decreciente, es posible aplicar *branch- and-bound*. El algoritmo resultante se denomina *poda alfa* por analogía con el algoritmo *alfa-beta*. De esta forma, se añade un parámetro *alfa* que será el mínimo valor *f* de los nodos de la frontera evaluados hasta el momento. Cuando se genera cada nodo interno se calcula su valor *f*, si el mismo iguala o supera el valor de alfa se termina la rama de búsqueda correspondiente. Es más, si se ordenan los estados expandidos de modo que los mejores nodos de la frontera sean evaluados primero, será posible reducir considerablemente la búsqueda al mejorar la eficiencia de la poda.

3.1.2 Los algoritmos de control de la ejecución

- ***Real- Time A****

*RTA** [*Korf- 90*] es un algoritmo de para controlar la fase de ejecución de la búsqueda en tiempo real, y es independiente del algoritmo de planificación. El algoritmo guarda el valor de cada estado visitado durante la etapa de ejecución en una tabla de hash. A medida que la búsqueda avanza se actualizan estos valores utilizando técnicas derivadas de la programación dinámica [*Koenig- 01*].

El proceso de búsqueda que realiza el algoritmo es el siguiente:

- *A los estados que no fueron visitados se les aplica la función de evaluación heurística, posiblemente extendida mediante una búsqueda.*
- *Para los estados que se encuentran en la tabla se utiliza el valor heurístico h guardado en ella.*
- *El vecino con el menor valor f es elegido para ser el nuevo estado actual y la acción para alcanzar ese estado es ejecutada.*
- *El anterior estado actual se guarda en la tabla y se le asocia el segundo mejor valor f de los vecinos más el costo para regresar al mismo desde la nueva posición.*

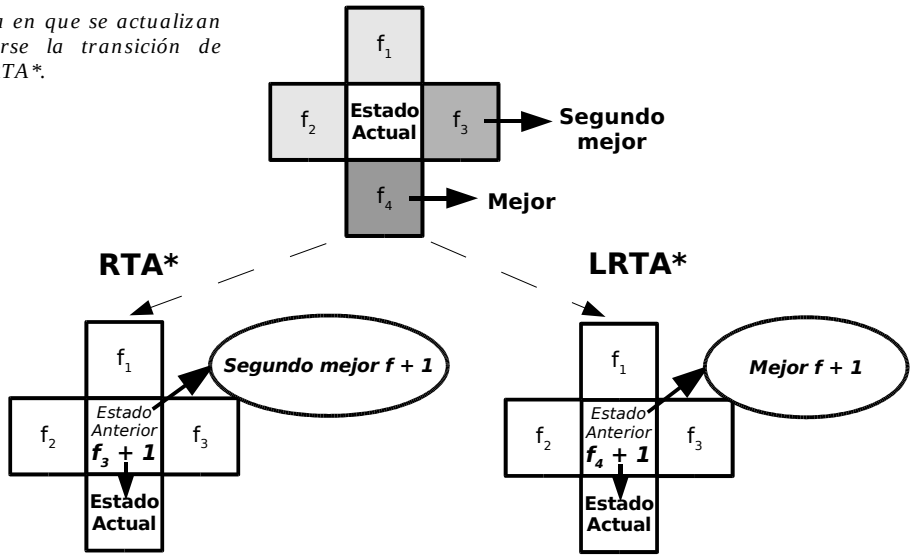
La propiedad más importante de RTA* es que, en espacios finitos y si el estado objetivo se puede alcanzar, se garantiza que se encontrará una solución. Además, el espacio requerido es lineal respecto al número de movimientos realizados, ya que sólo lleva una lista de los estados previamente visitados. El tiempo de ejecución también es lineal respecto a los movimientos ejecutados. Esto se debe a que, aunque el tiempo de planificación es exponencial respecto a la profundidad de búsqueda, el tiempo esta acotado por una constante al limitar la profundidad de búsqueda.

- ***Learning Real- Time A****

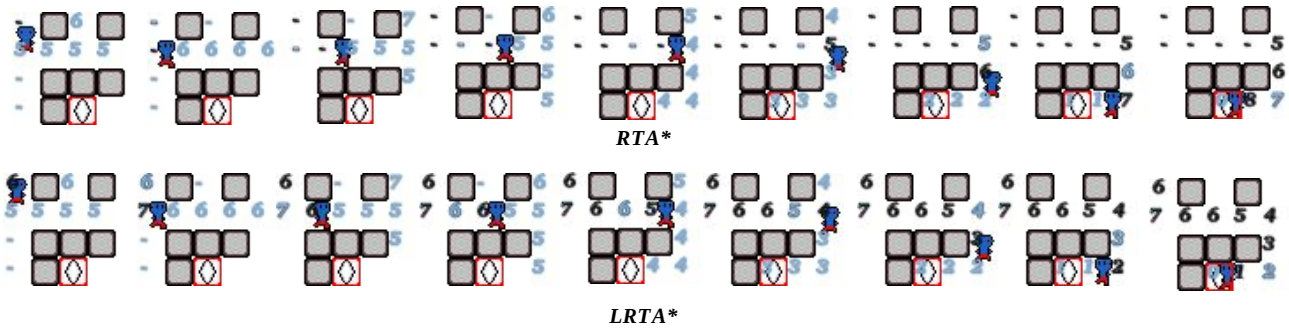
*LRTA** [*Korf- 90*] es una versión del RTA* con aprendizaje. LRTA* es un algoritmo eficiente para resolver problemas de búsqueda donde el estado objetivo es el mismo. El algoritmo tiene la propiedad de que con las sucesivas resoluciones de un problema los valores heurísticos convergiran a los valores exactos de los caminos óptimos.

LRTA* se comporta en forma similar que RTA*, sólo cambia la forma en la que se guardan los valores en la tabla (*figura 6*). En vez de guardar el segundo mejor valor *f*, la tabla se actualiza con el mejor valor *f*. Entonces, cuando un problema es resuelto, se guardan los valores heurísticos y estos se convierten en los valores iniciales para la próxima instancia del problema

Figura 6: *Comparación de la forma en que se actualizan los valores heurísticos, al realizarse la transición de estados, en los algoritmos RTA* y LRTA*.*



Aquí se muestra como funcionan los algoritmos RTA y LRTA* para la resolver del camino del ejemplo que se viene desarrollando. Los números celestes son los valores heurísticos asignados a las posiciones por el algoritmo Minimin durante la etapa de planificación. Los números negros son los valores heurísticos guardados en la tabla para las posiciones visitadas durante la ejecución de los movimientos.*



RTA*

LRTA*

3.2. Detalles de la implementación del control de la navegación

Al diseñar la aplicación, se decidió modelar en forma separada los componentes lógicos y los componentes que los controlan (*ver apéndice*). La clase *Simple_Token* describe los elementos lógicos con la capacidad de moverse y la clase *Simple_Controller* define un tipo de controlador, que le permite a la computadora dirigir el proceso de navegación de los elementos asociados (*figura 7*).

La clase *Simple_Controller* toma las decisiones de más alto nivel en el proceso de navegación: determina una posición de destino y le ordena al elemento asociado que se mueva en una dirección determinada. Este proceso se implementa en el método *update* que será llamado periódicamente (*figura 8*). La elección del movimiento se realiza a través de un *pathfinder* o *buscador*, independizando al controlador de la estrategia de búsqueda elegida.

El proceso de búsqueda en tiempo real está implementado en la clase *Pathfinder*. Los buscadores mantienen las estructuras de los algoritmos de ejecución y realizan la planificación. El método *get_move* implementa los algoritmos RTA* y LRTA* (*figura 9*). El

método *minimin* implementa el algoritmo de búsqueda Minimin con poda alfa, que se utiliza para la evaluación de estados (*figura 10*). Un aspecto relevante de la implementación de estos métodos es la forma en la se utilizan los valores heurísticos guardados, que difiere un poco a lo descripto en la sección anterior. La tabla de hash *h_values* es actualizada en el método *get_move*, pero es en el método *minimin* que se utilizan sus valores en reemplazo de la función heurística siempre que sea posible.

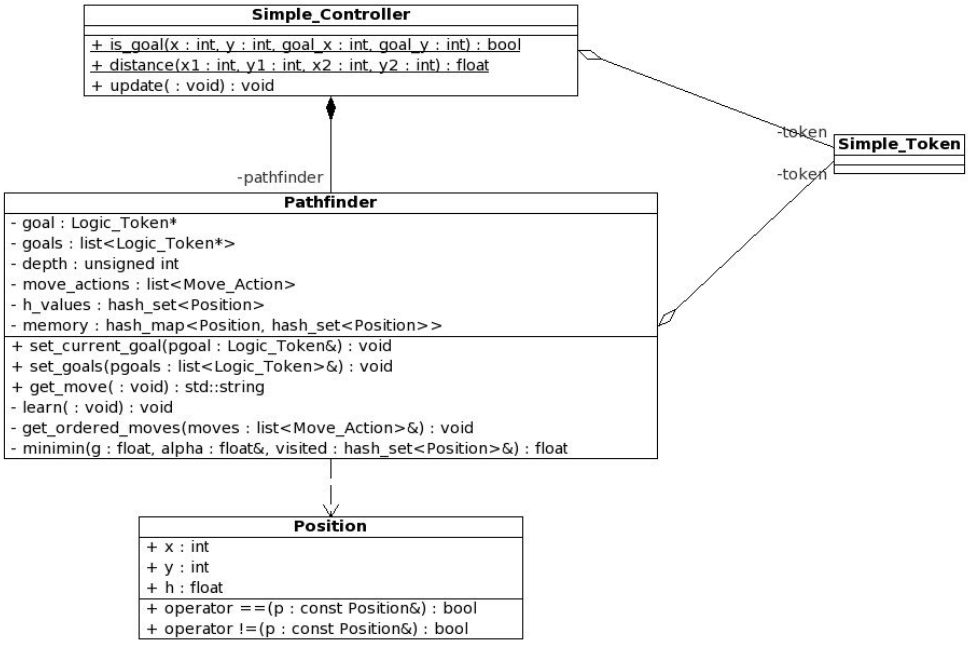


Figura 7: *Diagrama de clases de los componentes de la aplicación que implementan el control de la navegación*

```
void Simple_Controller::update (void) {
    // si no hay un objetivo, elegir como nuevo objetivo aquel que se encuentre más cerca
    if current_goal == 0
        min = ∞
        for each: goal in: goals
            d = distance (token->get_x(), token->get_y(), goal->get_x (), goal->get_y ())
            if d < min
                min = d
                current_goal = goal
        if current_goal != 0
            pathfinder.set_current_goal (current_goal)

    if current_goal != 0
        // verificar si se alcanzó la posición de algún objetivo
        isgoal = false
        for each: goal in: goals
            if is_goal (token->get_x(), token->get_y(), goal->get_x(), goal->get_y())
                isgoal = true
                current_goal = goal
        if isgoal
            // si se alcanzó un objetivo se destruye el mismo
            current_goal->destroy ()
            current_goal = 0
         else
            // sino, se ejecuta un nuevo movimiento en la dirección del objetivo actual
            move = pathfinder.get_move ()
            token->perform_action (move)
}
```

Figura 8: *Pseudocódigo del método Simple_Controller::update que implementa el control global del proceso de navegación.*

```
string Pathfinder::get_move (void){
    action = ""
    best = second_best = alpha = ∞

    position = new Position (token->get_x(), token->get_y())
    visited.insert (position)

    get_ordered_moves (moves)
    for each: move in: moves
        token->perform_action (move)
        f = minimin (0, alpha, visited)
        token->undo_action (move)
        if f < best
            second_best = best
            best = f
            action = move
        else
            if f < second_best
                second_best = f

    if RTA*
        p.h = second_best + 1
    else // LRTA*
        p.h = best + 1

    h_values.erase (p)
    h_values.insert (p)

    return action
}
```

Figura 9: *Pseudocódigo del método Pathfinder:: get_move que implementa los algoritmos de control RTA* y LRTA**

```
float Pathfinder::minimin (float g, float & alpha, hash_set<Position> & visited){
    if Simple_Controller::is_goal (token->get_x(), token->get_y(), goal->get_x(), goal->get_x())
        if g < alpha
            alpha = g
        return g
    else
        position = new Position (token->get_x(), token->get_y())
        pos = h_values.find (position)
        if pos != h_values.end ()
            if pos->h + g < alpha
                alpha = pos->h + g
            return pos->h + g
        else
            h=Simple_Controller::distance(token->get_x(),token->get_y(),goal->get_x(),goal->get_y())
            if (g >= depth) || (h + g >= alpha)
                if h + g < alpha
                    alpha = h + g
                return h + g
            else
                minimum = ∞
                visited.insert (position)
                get_ordered_moves (moves)
                for each: move in: moves
                    token->perform_action (move)
                    neighbor = new Position (token->get_x (), token->get_y ())
                    if visited.find (neighbor) == visited.end ()
                        f = minimin (g + 1, alpha, visited)
                        if f < minimum
                            minimum = f
                    token->undo_action (move)
                visited.erase (position)
                return minimum
}
```

Figura 10: *Pseudocódigo del método Pathfinder::minimin que implementa el algoritmo de planificación Minimin*

3.3. Análisis del comportamiento de los algoritmos

El análisis consistió en la observación del comportamiento de los algoritmos en laberintos generados mediante la aplicación. Los laberintos son de distintos tamaños e incluyen obstáculos estáticos y móviles, y pueden ser modificados dinámicamente. Tanto el RTA* como el LRTA* fueron probados con distintos límites de profundidad para el algoritmo Minimin y utilizando la distancia euclídea como función heurística.

Se realizó una comparación entre el RTA* y la primera instancia de resolución del LRTA* en varios laberintos. En general se vió que el RTA* tiene un desempeño mucho mejor. Esto se debe a que el LRTA* guarda valores menores para los estados visitados y por lo tanto tiende a volver a ellos con más frecuencia.

Por otra parte, la calidad de las soluciones de LRTA* tienden a mejorar notablemente en pocas repeticiones de la resolución del mismo camino. Esto no se da siempre ya que, cuando la frontera de búsqueda es pequeña y los laberintos son relativamente complicados, el costo de las soluciones tarda varios ciclos en estabilizarse cerca del óptimo.

Respecto a la variación de la profundidad de búsqueda se tuvieron resultados que van en contra de lo que se esperaba. Al aumentar la profundidad de búsqueda y contar con evaluaciones más precisas mejoran algunos aspectos del comportamiento de la navegación, pero no siempre aumenta la calidad de la solución. En general, dependiendo del laberinto y el camino a resolver, contar con una función de evaluación más precisa no lleva a que el RTA* o una primera instancia del LRTA* ejecute menos movimientos.

El problema es que el agente vuelve a posiciones ya visitadas con mucha frecuencia, aunque la dirección que llevaba era la correcta. Un primer análisis permitió observar que los valores heurísticos guardados son parte de la función evaluación de estados, y que la única forma en la que los algoritmos básicos pueden actualizar estos valores es visitándolos otra vez. Cuando las evaluaciones iniciales están por debajo de los valores exactos el proceso de búsqueda se vuelve propenso a regresar a esos estados. Así se recorrerán zonas ya visitadas sólo para subir los valores heurísticos asignados y ajustar la diferencia. Como ya se señaló, el algoritmo LRTA* guarda valores heurísticos menores, en comparación con el RTA*, y por lo tanto sufre mucho más de este problema.

Después de una investigación más profunda del problema, se encontró un trabajo en el que se aplica la búsqueda en tiempo real a la navegación y búsqueda de objetivos móviles [*Ishida- Korf- 95*]. Allí se discute el problema expuesto y se lo denomina “depresión heurística”. El problema de la depresión heurística consiste en la formación de regiones que son asignadas valores heurísticos menores que los exactos y quedan limitadas por regiones de valores más elevados. Cuando se forma una depresión

heurística, el proceso de búsqueda queda estancado momentáneamente en las posiciones de esta región, hasta que los valores asignados a las mismas iguala a los valores del límite de la depresión. Una forma de solucionar el problema es identificar la formación de una depresión y actualizar los valores de la región mediante una búsqueda off- line.

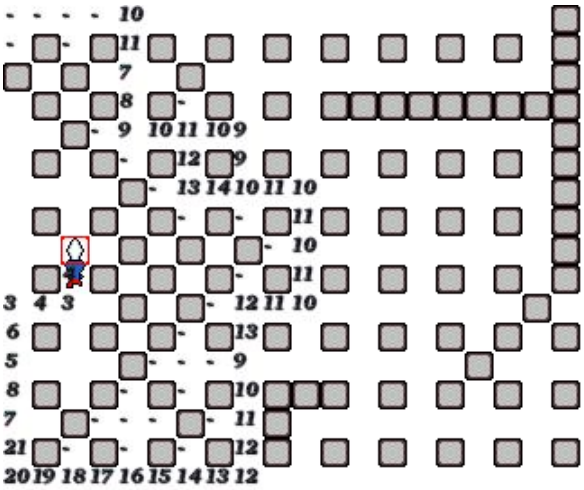
Por otra parte, se comprobó una mejora muy importante en la performance del algoritmo Minimin con poda alfa, respecto a la versión sin poda (*figura 11*). Esta optimización, en conjunto con el ordenamiento de los nodos, es de extrema importancia para tener buenos tiempos de respuesta, sin sacrificar demasiada precisión en las evaluaciones.

Sin poda ***Con poda alfa***

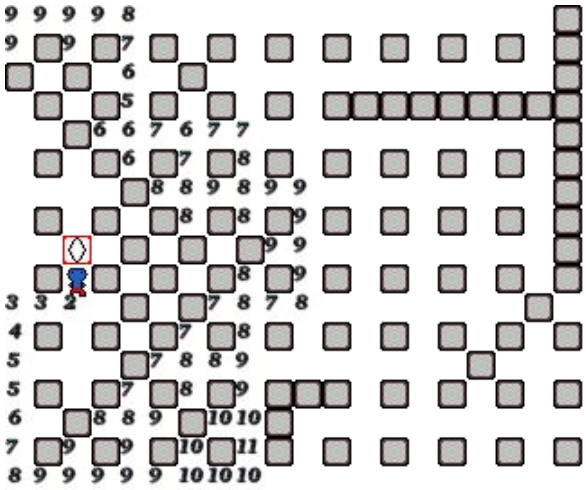
Figura 11 *: Comparación de los estados expandidos durante la planificación por el algoritmo Minimin, sin realizar podas y utilizando branch- and- bound o poda alfa y ordenamiento de nodos. La frontera de búsqueda es 10.*

Las pruebas en entornos dinámicos fueron bastante satisfactorias con ambos algoritmos. La estrategia de tener en cuenta sólo los cambios más cercanos al estado actual parece ser bastante buena, sobre todo en entornos donde hay una cantidad elevada de elementos dinámicos. En general, si los obstáculos móviles no son extremadamente difíciles de sortear, los movimientos realizados eventualmente se coordinan para superarlos.

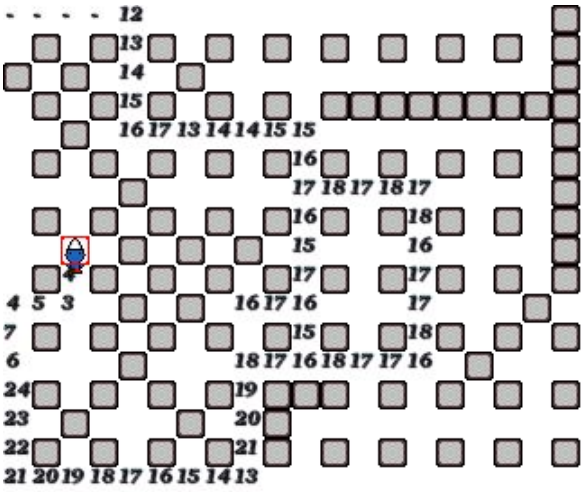
En la figura 12 se comparan los algoritmos RTA* y LRTA* utilizando distintas profundidades. En este caso particular, es posible ver algunos de los resultados presentados. En este ejemplo, se muestran las posiciones que visita cada algoritmo junto con los valores heurísticas asignados, y se indica aproximadamente la máxima extensión que alcanza la zona de depresión heurística en los casos donde es posible observar el fenómeno con claridad.



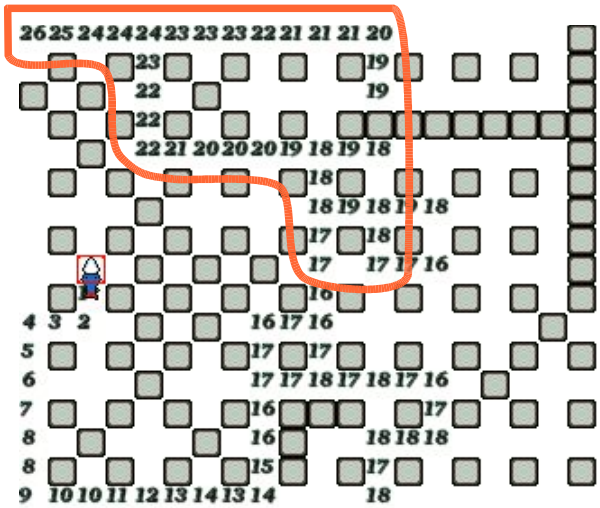
Profundidad 0 – Movimientos 102



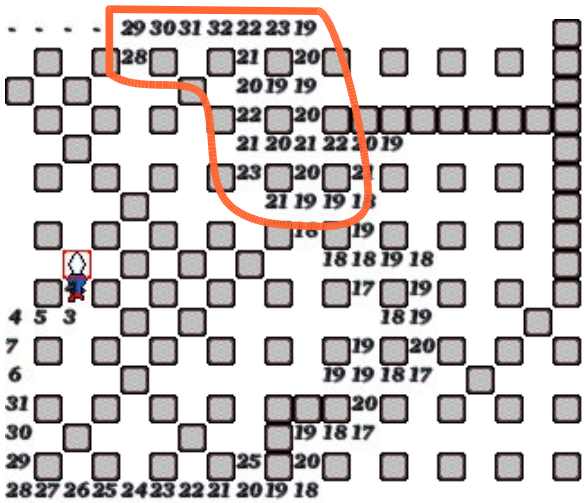
Profundidad 0 – Movimientos 104



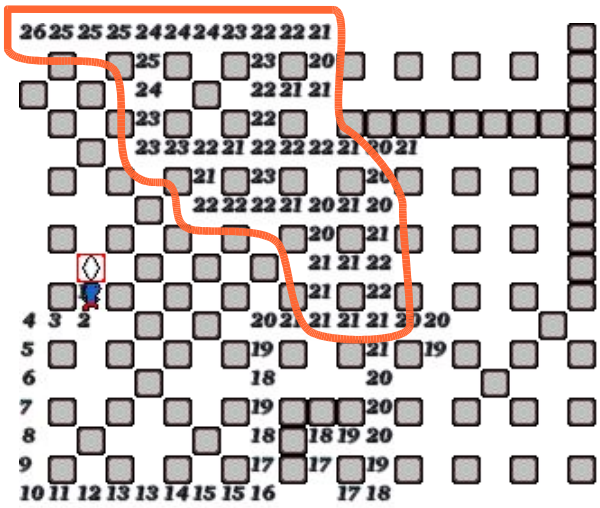
Profundidad 5 – Movimientos 74



Profundidad 5 – Movimientos 410



Profundidad 10 – Movimientos 116



Profundidad 10 – Movimientos 398

RTA*

LRTA*

Figura 12 *: Valores asignados a las posiciones visitadas para la misma búsqueda utilizando la profundidad de búsqueda y realizando el número de movimientos indicados. El número de movimientos óptimo es 46. La posición de salida es siempre la esquina superior izquierda y el destino es el casillero con el marco rojo. La heurística utilizada es la distancia euclídea. En los casos que corresponde, se indica aproximadamente la máxima extensión de la zona de depresión heurística.*

4. Conclusiones

La experiencia de implementar los algoritmos y el análisis realizado permitió ganar una comprensión mucho mayor del tema de búsqueda en general y, por supuesto, de los algoritmos de búsqueda en tiempo real. Con la información recopilada durante el estudio de las técnicas de búsqueda y los juegos de vídeo, y junto con los resultados obtenidos durante los experimentos realizados, se puede concluir que los algoritmos de búsqueda en tiempo real constituyen una alternativa viable para la navegación en el contexto de los juegos de vídeo en tiempo real. Sin embargo, se ve son necesarias técnicas y estrategias adicionales para poder obtener soluciones de la calidad requerida por este tipo de aplicaciones.

En muchos casos el comportamiento de los algoritmos implementados fue aceptable. En particular, la navegación en entornos dinámicos resultó satisfactoria. Sin embargo, se observó que los algoritmos presentan algunos inconvenientes. El problema más importante lo constituye la formación de regiones de depresión heurística. Este fenómeno lleva a que la búsqueda se estanque en una región durante un tiempo considerable. Lo peor es que esto sucede con frecuencia, sobre todo cuando se utiliza el algoritmo LRTA*. Una alternativa para de solución es la utilización de búsquedas off-line, que eviten la necesidad de regresar innecesariamente a regiones ya visitadas. En el contexto de la navegación en los juegos de vídeo, el comportamiento de los algoritmos puede ser mejorado considerablemente. En este sentido, muchas de las técnicas utilizadas para mejorar el desempeño de los algoritmos tradicionales como el A* [*Pinter-01*] podrían ser adaptadas para que funcionen con el RTA* y el LRTA*.

Es importe mencionar que la aplicación desarrollada, en la cual se encuentran implementados los algoritmos estudiados, se convirtió en un proyecto open source. En realidad, esta fue una idea que surgió al comienzo del proyecto y se decidió llevar adelante debido a los resultados resultados satisfactorios obtenidos con la misma para demostrar el funcionamiento de la búsqueda en tiempo real. El proyecto se encuentra en SourceForge.net bajo el nombre de *"Simple Real Time Search for Pathfinding"*. El enlace es: *sourceforge.net/projects/rts-simple*. Además de subir el código, se planea incluir documentación relacionada y hacer disponibles todas las extensiones que se realicen sobre este trabajo.

Apéndice – Diseño e implementación de la aplicación

A.1. La arquitectura

La aplicación fue desarrollada teniendo como premisas principales de diseño la flexibilidad y expansibilidad. Para conseguirlo se crearon una serie de clases que abstraen las características y comportamiento común de los juegos en tiempo real que se desarrollan en laberintos de dos dimensiones. Este conjunto de clases describen una arquitectura para la aplicación muy simple que se describe a continuación.

Lo primero que se identificó fue la existencia de componentes que conceptualmente son independientes. Los componentes principales que se consideran son los elementos que constituyen la lógica (por ejemplo el jugador, los enemigos, las paredes del laberinto, etc) y los elementos visuales (como las animaciones y efectos). Además, como parte de la lógica se distinguen los elementos lógicos propiamente dichos y los componentes que determinan su comportamiento y los controlan (como los módulos de la inteligencia artificial). Todos estos componentes estan relacionados ya que deben trabajar en conjunto; sin embargo es deseable que exista cierta independencia entre ellos. Esto se consigue modelando la lógica, el control y la visualización en forma separada. La clase *Base_Token* modela los atributos y comportamiento de estado común que poseen estos componentes, e implementa un mecanismo para comunicarlos a través de los métodos del patrón observador [*Gamma - 94*]. La intención es definir a partir de esta clase a el resto de las clases que modelan la lógica, control y visualización.

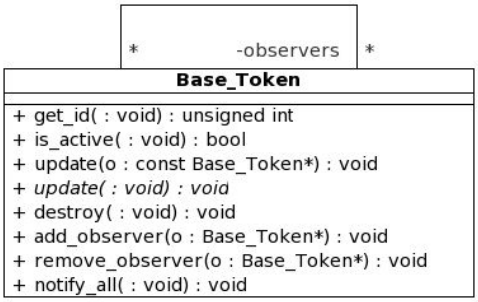


Figura A.1: *La clase base los elementos lógicos, los controladores y los elementos visuales*

La arquitectura incluye los componentes fundamentales que constituyen la lógica del juego. A través de las clases Logic_Token y Action se logra desacoplar de los elementos lógicos los comandos o acciones mediante los cuales modifican su estado o el estado de su entorno (este mecanismo constituye el patrón comando [*Gamma - 94*]). La clase Space representa a una estructura que permite el acceso a los elementos lógicos a través de su posición.

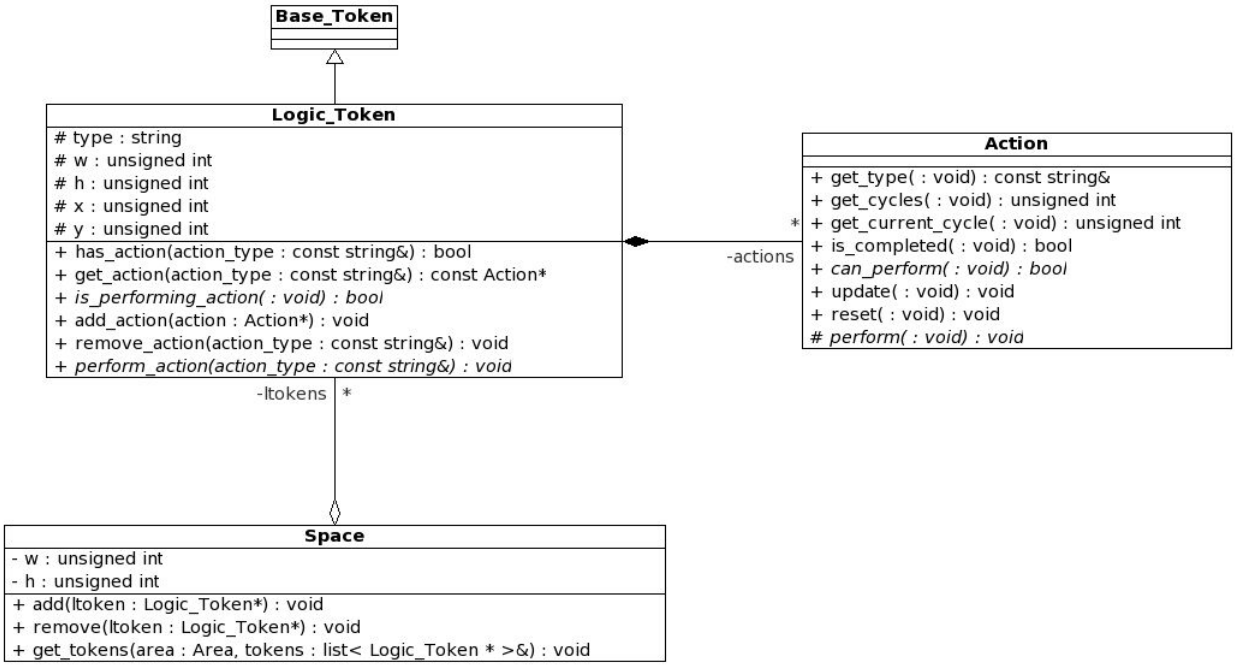


Figura A.2: *Los componentes de la lógica modelados en el framework*

La visualización fue un aspecto secundario y esto llevó a que sólo se añadiera un soporte mínimo para la visualización de animaciones a través de la clase *Animation*. Sin embargo, extender esta parte del sistema debería ser fácil porque el resto de los componentes son independientes de la representación gráfica.

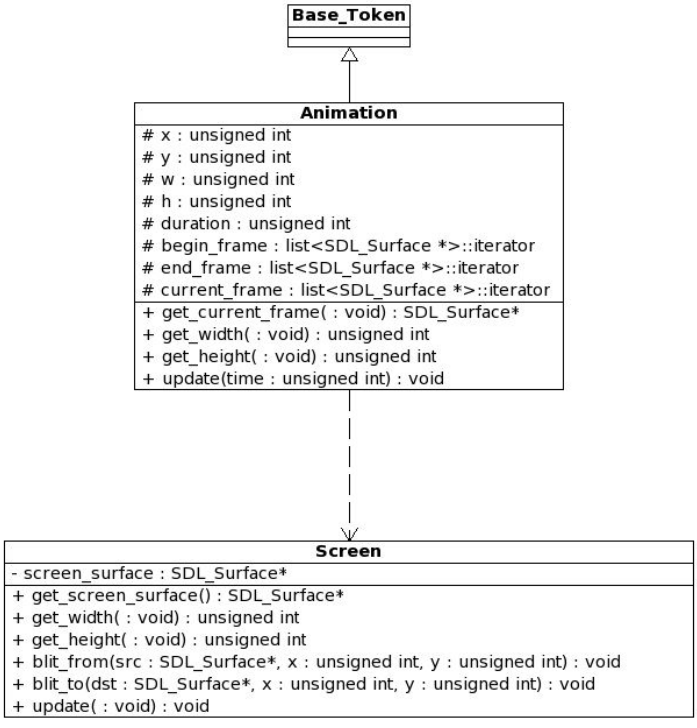


Figura A.3: *Los componentes de la visualización modelados en el framework*

A.2. La aplicación

El desarrollo de la aplicación consistió fundamentalmente en completar la definición de las clases abstractas introducidas en la arquitectura general descripta. De esta forma se introdujeron varias clases que definen el comportamiento y atributos de los elementos lógicos (Simple_Token), de control (Automove/Player/Simple_Controller) y visualización (Simple_Animation) específicos.

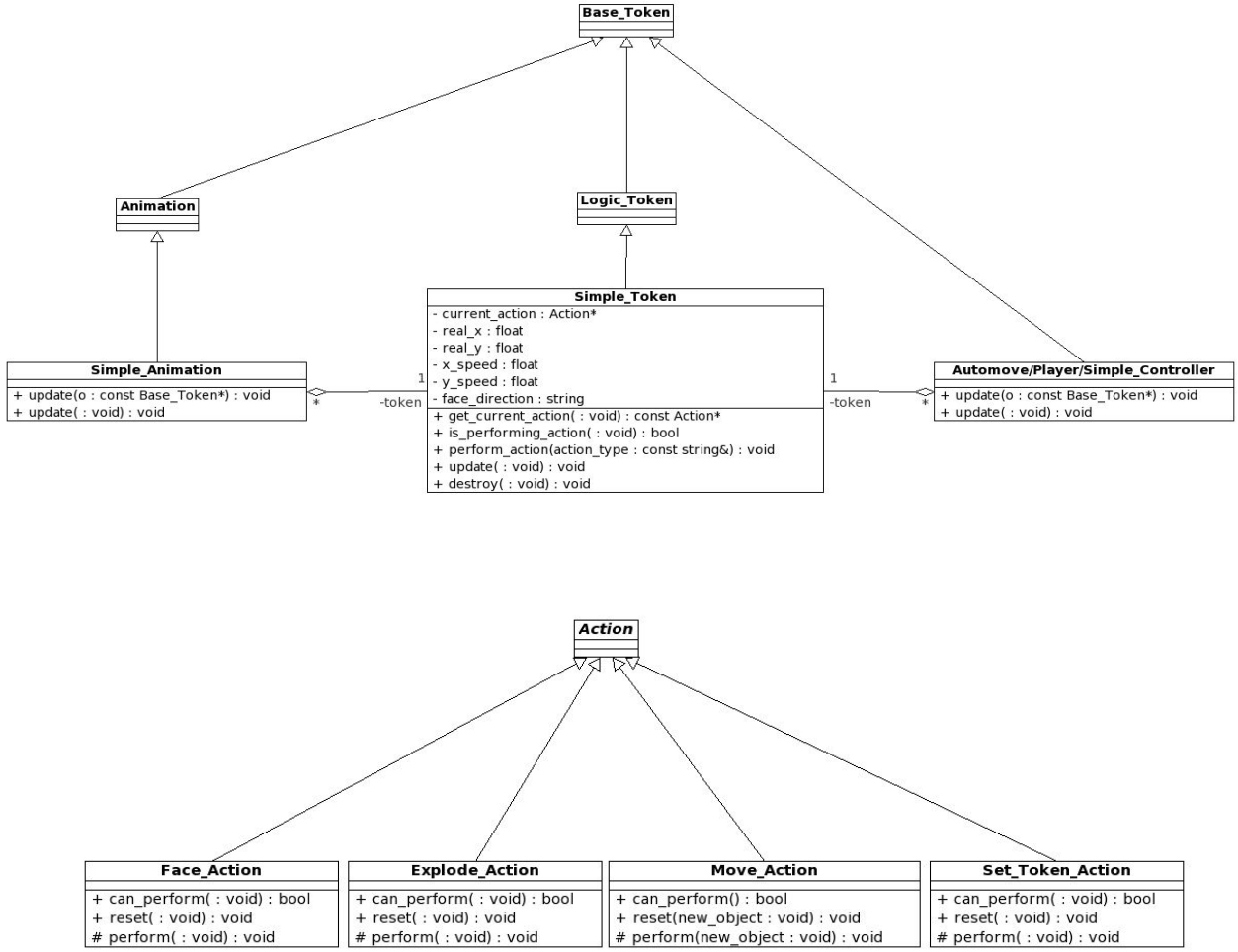


Figura A.4: *Los clases introducidas por la aplicación para instanciar el framework*

En lo que respecta a la lógica es importante remarcar que objetos lógicos tan diversos como los enemigos, las paredes y diamantes son todos instancias de la clase *Simple_Token*. Esto se debe a que los atributos estáticos (como la posición) son esencialmente los mismos para todos. Lo que cambia es el comportamiento y esto se modela mediante la asociación dinámica de las acciones y los controladores.

Hay muchas ventajas al trabajar con el bajo nivel de acoplamiento presente entre los componentes lógicos, el control y la visualización. En la aplicación esto permitió implementar un sistema de configuración mediante scripts muy flexible.

A.3. Detalles de la implementación

El lenguaje de programación en el que se desarrolló la aplicación es C++ standard. La idea es que la aplicación corra en varias plataformas, por eso todas las librerías utilizadas son multi-plataforma. El compilador utilizado es el de la Gcc (versión 3.x.x), pero deberían poder utilizarse otros sin necesidad de realizar modificaciones al código. Las plataformas probadas hasta el momento son Linux (que es la plataforma de desarrollo principal) y todas la versiones de Windows (9x y NT) (utilizando el port de la Gcc provisto por la MinGW).

Las librerías que se utilizan son:

- **Lua**: Motor de scripting.

  www.lua.org

- **SDL** (Simple Media Layer): Multimedia (video, sonido, entrada, eventos, threads).

  www.libsdl.org

- **STLport**: Implementación portable de la STL (Standard Template Library) de C++.

  www.stlport.org

Bibliografía

[Bender- 96] *"Mathematical Methods in Artificial Intelligence"*
Edward A. Bender (IEEE Computer Society Press and John Wiley – Enero 1996)

[Brockinton- 00] *"Pawns Captures Wyvern: How Computer Chess Can Improve Your Pathfinding"*
Mark Brockington (Game Developer Conference – 2000)

[Gamma- 94] *"Design Patterns. Elements of Reusable Object- Oriented Software"*
Erich Gamma, Richad Helm, Ralph Johnson, John Vlissides (Adisson- Wesley – 1994)

[Ishida,Korf- 95] *"Moving- Target Search: A Real- Time Search for Changing goals"*
Toru Ishida, Richard E. Korf (IEEE Transactions on Pattern Analisys and Machine Intelligence, 17 – Junio 1995)

[Koenig- 01] *"Agent- Centered Search"*
Sven Koenig (Artificial Intelligence Magazine, 22 – 2001)

[Koenig- 04] *"Incremental Heuristic Search in Artificial Intelligence"*
Sven Koenig, Maxim Likhachev, Yaxin Liu, David Furci (Artificial Intelligence Magazine – 2004)

[Korf- 90] *"Real- Time Heuristic Search"*
Richard E. Korf (Artificial Intelligence, 42 – Marzo 1990)

[Korf- 00] *"Artificial Intelligence Search Algorithms"*
Richard E. Korf (Agosto 2000)

[Laird,Pottinger- 00] *"Game AI: The State of the Industry, Part Two"*
John E. Laird, Dave C. Pottinger (Gamasutra – Noviembre 2000)

[Nareyek- 04] *"AI in Computer Games"*
Alexander Nareyek (Game Development Vol 1, 10 - Febrero 2004)

[Pinter- 01] *"Toward More Realistic Pathfinding"*
Marco Pinter (Gamasutra – Marzo 2001)

[Pottinger- 00] *"Terrain Analysis in Realtime Games"*
Dave C. Pottinger (Game Developer Conference – 2000)

[Stout- 96] *"Smart Moves: Intelligent Pathfinding"*
Bryan Stout (Game Developer Magazine – Octubre 1996)

[Underger- 01] *"Real- Time Edge Follow: A New Paradigm to Real- Time Search"*
Cagatay Underger, Faruk Polat, Ziya Ipekkan (2001)

Links

www.gamasutra.com

www.gamedev.net

www- cs- students.stanford.edu/~amitp/gameprog.html