UNIVERSIDAD AUTÓNOMA METROPOLITANA
Unidad Azcapotzalco

GENERACIÓN Y SOLUCIÓN DE ROMPECABEZAS RUSH HOUR

Asesor:
Dr. Francisco Javier Zaragoza Martínez

Alumnos:
Daniel López Rubio
Francisco Hermosillo García

México D.F Trimestre 2010 P

Contenido

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Algoritmos

El ser humano desde sus inicios se ha inclinado a crear mecanismos que resuelvan problemas de cierto tipo creando máquinas que han ido evolucionando, desde la simplicidad pero efectividad de una rueda de piedra hasta las complejas computadoras que nos ayudan a satisfacer las necesidades actuales igual de complicadas. Así mismo ha creado ciertas metodologías para llevar a cabo tareas de una manera cada vez más sencilla y estandarizada.

Un algoritmo es un conjunto de reglas o pasos bien definidos, ordenados y finitos que permita realizar una actividad o resolver un problema. Parte de un estado inicial y mediante esta secuencia de pasos se debe de llegar a un estado final, obteniendo así una solución. En la vida cotidiana se utilizan algoritmos para resolver diversas tareas, como el seguir una receta para elaborar un pastel o generar rutas de entrega de mercancías que involucre el menor costo posible.

En la actualidad la facilidad y rapidez con la que los equipos de cómputo resuelven los problemas humanos no es gratuita, éstos se deben de programar para que puedan realizar su labor de manera satisfactoria, esta programación en ocasiones se enfoca a la implementación de algoritmos dedicados a un problema en particular, siendo necesario en ocasiones aplicar diversas técnicas para el diseño de algoritmos que ofrezcan una solución viable al problema. Existe una rama del estudio encargada de encontrar y perfeccionar estas técnicas: La algoritmia.

La algoritmia se enfoca principalmente en el diseño, creación y análisis de los algoritmos. Tratando de definir ciertas pautas para logralo se han desarrollado técnicas para generar algoritmos que resuelvan un problema. Las técnicas más comunes son las siguientes:

- **Divide y Vencerás:** Implica resolver un problema difícil, dividiéndolo en pequeños problemas similares en partes más simples tantas veces como sea necesario, hasta que la solución de las partes es obvia. Al final la solución del problema original se construye a partir de las soluciones de las partes más pequeñas.

  Un ejemplo es el de la búsqueda binaria: Se trata de encontrar un elemento dentro de una lista, la opción de buscar elemento por elemento (búsqueda lineal) es poco efectiva, dado que en el peor de los casos el elemento se puede encontrar al final. La solución de divide y vencerás, separa la lista en dos, recursivamente, se divide cada una de estas listas en otras dos, parando cuando sólo nos quede un elemento en cada subconjunto.

  En general, los algoritmos de divide y vencerás suelen parecer más complicados, pero la ejecución es menos costosa, en cuanto a tiempo de ejecución se refiere.

- **Búsqueda con Retroceso:** Se basa en resolver problemas recorriendo el espacio completo de las soluciones al problema planteado. Generalmente estos algoritmos no aplican ningún tipo de optimización dado que recorren todo el árbol de soluciones. Sin embargo se puede aplicar una operación de poda que evita que se recorra por estados que de antemano se sabe que no conducen a la solución.

  El funcionamiento básico de los algoritmos de búsqueda con retroceso son una aplicación directa del método de búsqueda en profundidad en un árbol. Sus pasos son: Tomar una posible opción, para cada elección considerar recursivamente cada opción posible, devolver la mejor solución posible.

  La desventaja de estos algoritmos es que su ejecución puede resultar de tiempo exponencial, además que su análisis es muy complicado.

- **Programación dinámica:** Es una optimización a los algoritmos de busqueda y retroceso, pero con un costo de memoria, dado a que utiliza estructuras auxiliares para almacenar soluciones de subproblemas que son repetidos hasta llegar a la solución del problema original. De tal forma que cada subsolución se calcule sólo una vez y no se explore todo el conjunto de soluciones completo.

Los algoritmos pueden arrojar distintos tipos de resultados, dependiendo del tipo de interés en el área de trabajo en los que se empleen, por ejemplo, dar la solución óptima o simplemente proponer una solución que podría no ser la mejor, pero si factible al problema. Entre los de segundo tipo se encuentran los algoritmos genéticos que nos brindan una solución que resuelve un problema, que cabe destacar que puede ser óptima o no.

Un algoritmo genético consta de los siguientes pasos:

1. **Inicialización:** Se define de manera aleatoria un conjunto de individuos, los cuales son posibles soluciones.
2. **Evaluación:** A cada individuo se le aplica una función de aptitud, con el fin para tener un parámetro que nos indique que tan buena es la solución.
3. **Selección:** Se eligen a los mejores individuos provenientes de la evaluación.
4. **Cruzamiento:** Se combinan en pares las soluciones seleccionadas para generar una nueva generación de posibles soluciones.
5. **Mutación:** Se altera de manera aleatoria alguna parte del individuo, asegurando el explorar otras zonas del espacio de búsqueda.
6. **Reemplazo:** Se reemplaza la población inicial con la nueva generación de soluciones.

Este algoritmo se repite hasta que se encuentre una solución que se ajuste a las restricciones dadas, dando así una solución factible a un problema.

El proyecto que se desarrolló se enfoca en diseñar algoritmos que se especialicen en solucionar los siguiente problemas:

- Encontrar la solución óptima a un escenario de juego de tipo combinatorio y dar la secuencia de pasos que llegan a ella mediante técnicas como búsqueda con retroceso y programación dinámica.
- Proponer una solución factible a la creación de escenarios de juego dadas ciertos parámetros, tales como el número de pasos requeridos para darle solución. Esto se puede lograr utilizando algoritmos genéticos

1.2. Graficación e interfaces

1.2.1. Computación gráfica

La Computación Gráfica se encarga del estudio, diseño y trabajo del despliegue de imágenes en la pantalla de un computador a través de las herramientas proporcionadas por la física, la óptica, la térmica, la geometría, etc. El principal objetivo es lograr desarrollar un modelo que se asemeje al mundo real lo más fielmente posible. Para lograr esto se deben considerar distintos factores que se encuentran en la naturaleza, tales como luz, sombra y textura.

Lograr una representación fiel de un objeto del mundo real en una gráfica de computadora requiere del diseño e implementación de diversos algoritmos que están enfoncados a simular un hecho o fenómeno que ocurre en la realidad, por ejemplo, la refracción de la luz que produce reflejos sobre un tipo de material brilloso o la proyección de la sombra sobre una superficie, etc.

Todos estos algoritmos son herramientas útiles para la creación de *mundos virtuales*, los cuales son muy utilizados hoy en día porque muchas de sus aplicaciones trascienden en varios campos de la ciencia generando simulaciones de lo que ocurriría en el mundo real, dadas cierta condiciones. Además la industria de los videojuegos está en auge gracias a los avances logrados en el campo de la Computación Gráfica.

1.2.2. Interfaces

La interfaz gráfica de usuario o GUI por sus siglas en inglés, es el medio con que el usuario puede comunicarse con una máquina, un equipo, una computadora o un sistema y comprende todos los puntos de contacto entre el usuario y el equipo, normalmente suelen ser fáciles de entender y fáciles de accionar.

El principal objetivo de una interfaz de usuario es que éste se pueda comunicar a través de ella con algún tipo de dispositivo como el *mouse*, el teclado de la computadora o en el caso de los equipos táctiles modernos el toque de la pantalla; conseguida esta comunicación, el segundo objetivo que se debería perseguir es el de que dicha comunicación se pueda desarrollar de la forma más fácil y cómoda posible para el usuario y que éste pueda visualizar los cambios que se realicen en el sistema de una manera igual de sencilla.

Uno de los objetivos del proyecto es diseñar mediante la graficación por computadora un modelo virtual del tablero de juego que ofrezca una interfaz de usuario amigable y que permita la interacción mediante la generación y captura de eventos.

1.3. Definición del juego Rush Hour

El juego Rush Hour es un puzzle compuesto de una cuadrícula de tamaño $m \times n$ cerrada de los bordes por un marco, excepto por un agujero en el borde derecho (Figura 1.1). Sobre la cuadrícula van colocadas piezas, específicamente de tres tipos: aquellas que ocupan dos espacios de la rejilla; otras que ocupan tres de estos espacios y una pieza especial la cual es única y de un color diferente a todas las demás. Las piezas se colocan de forma vertical u horizontal según se desee.

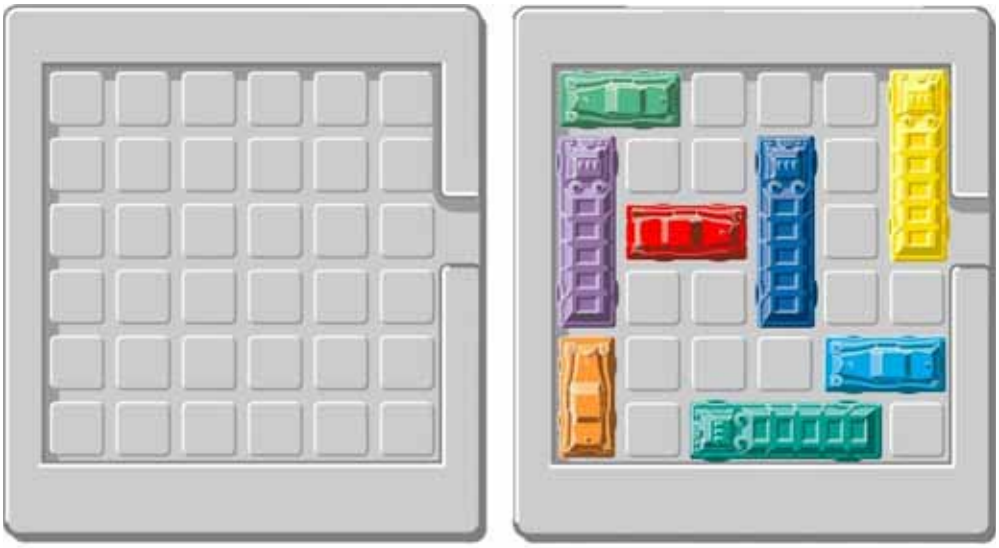

Figura 1.1: *Tablero de juego*

El objetivo del juego es sacar de la cuadrícula el auto especial por el orificio en el marco mediante reglas sencillas:

- Las piezas solo se pueden mover hacia adelante o hacia atrás las casillas que sea posibles avanzar. A esto se le considera un movimiento.
- Una pieza no puede salirse del marco que rodea la cuadrícula.
- La pieza especial debe estar en donde se encuentra el agujero por el cual deberá salir.
- Pueden existir muchas soluciones a un mismo tablero, pero lo interesante de cada uno sería hacerlo en el mínimo número de movimientos, esto es, en el menor número de desplazamientos hacia atrás o adelante de cada pieza (Figura 1.2).

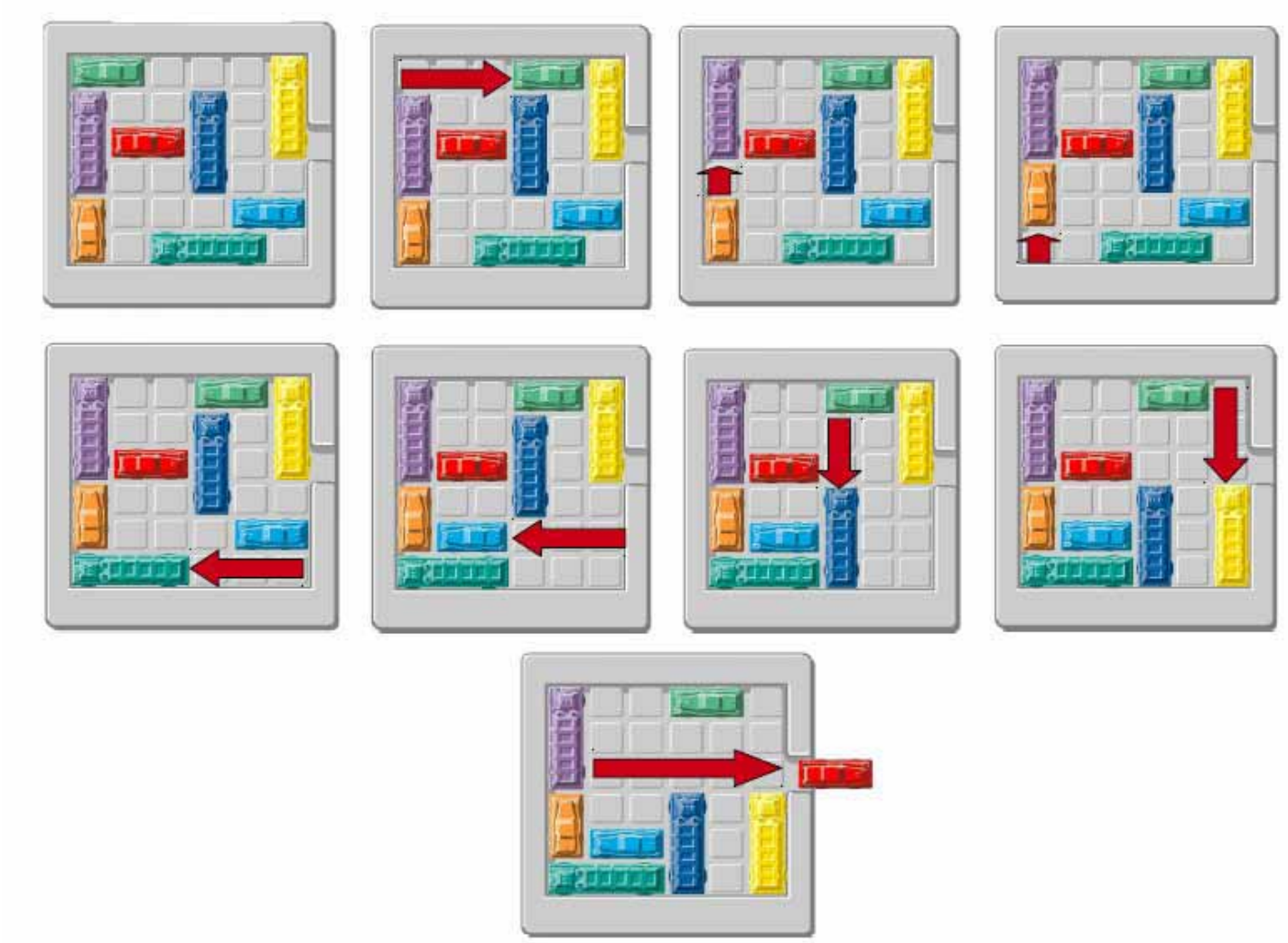

Figura 1.2: *Solución con el mínimo número de pasos*

Capítulo 2

Análisis

2.1. Alcance del sistema

Se tiene contemplado realizar un sistema completo de juego del rompecabezas Rush Hour, el cual permita al usuario mover las diferentes piezas para lograr el objetivo del rompecabezas, el cual es sacar por un orificio el objeto especial.

Los movimientos que el usuario podrá hacer se harán mediante una interfaz gráfica de usuario sencilla que permitirá obtener una visualización completa del tablero de juego y que mediante las funciones del ratón sea posible realizar todas las opciones de juego.

También se debe diseñar un algoritmo que sea capaz de resolver el rompecabezas de manera automática y dé el conjunto de pasos necesarios para ello, el cual debe de interactuar con la interfaz gráfica y mostrar una animación de cada uno de estos pasos. Además gracias al algoritmo de solución es posible ofrecer una pista sobre el siguiente movimiento en caso de que el usuario se atore en el rompecabezas.

Otro de los puntos importantes es el diseño de un algoritmo que sea capaz de generar tableros dados ciertos parámetros de entrada, como la cantidad de piezas y el número de pasos deseado para la solución del tablero.

2.2. Objetivo general

Desarrollar un sistema que contenga algoritmos capaces de resolver y generar de manera automática tableros de prueba para el juego Rush Hour. Además de crear un motor visual que permita al usuario interactuar y resolver los tableros generados. Este sistema se llamará de ahora en adelante RHS (Rush Hour Solver).

2.3. Objetivos particulares

- Diseñar un algoritmo eficiente que resuelva el problema del juego en el mínimo número de pasos posible.
- Crear un algoritmo que evalúe y compruebe que los tableros de entrada cumplan con los requisitos del juego.
- Generar mediante algoritmos genéticos tableros para el juego.
- Definir un formato de representación de tableros que sea fácilmente interpretado como entrada o salida de los algoritmos antes mencionados.
- Diseñar e implementar una interfaz gráfica del juego y la interacción de éste con los algoritmos anteriores.

2.4. Arquitectura

El sistema en general está formado por dos módulos principales y un módulo de integración que permite su interacción. Éstos serán detallados posteriormente en el capítulo de desarrollo. A continuación se da una descripción general de cada módulo y un diagrama general del sistema (Figura 2.1).

- Motor de juego: Es el encargado de manejar lógicamente el juego y la posición de las piezas dentro de una matriz de caracteres. Contendrá las funciones de resolver de manera automática un tablero y de generar ejemplos de juego.
- Motor gráfico: Es el encargado de dibujar en pantalla las piezas y el tablero, procesar los eventos producidos por el ratón o el teclado. Es la parte del sistema que estará en contacto directo con el usuario.
- Modulo de Integración: Se trata de un sistema de archivos cuyo contenido será utilizado por ambos motores para lograr una intercomunicación y poder lograr el objetivo del sistema

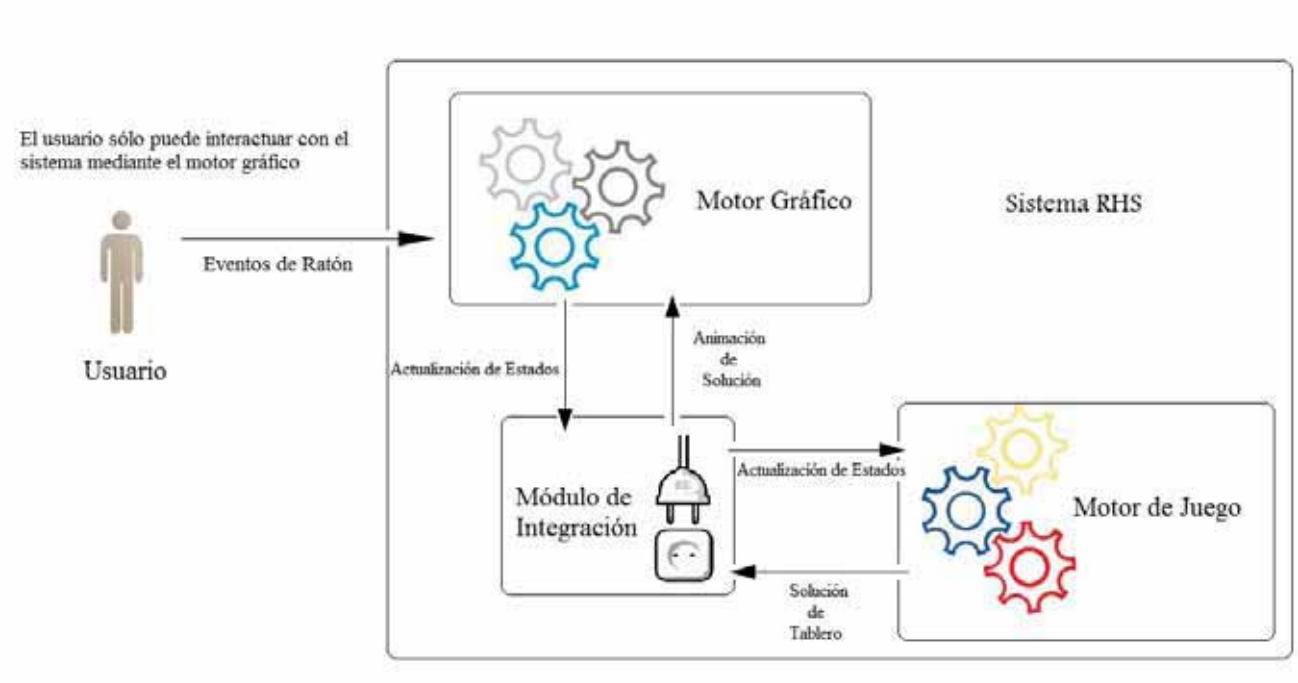

Figura 2.1: *Arquitectura del Sistema RHS*

2.5. Requerimientos de hardware y software

- Memoria RAM 512 MB mínimo.
- Procesador a 2 Ghz o superior.
- Resolución de pantalla mínima de 600 x 600.
- OpenGL 2.1 o superior.
- Sistema Operativo Windows XP o superior.

Capítulo 3

Desarrollo

3.1. Módulo de integración

Es el encargado de comunicar los 2 módulos principales, éste consiste en un pequeño sistema de archivos que contienen información de intercambio entre el Motor Gráfico y el Motor de Juego. Los archivos contenidos tienen ciertas características y dentro de ellos se encuentran las descripciones del tablero y del Conjunto de Soluciones encontradas para cada Estado de Juego.

Antes de entrar en detalle de los archivos dentro de éste módulo, fue necesario diseñar una representación de un tablero de juego que pueda ser entendida tanto por el Motor Gráfico y el Motor de Juego (Figura 3.1).

3.1.1. Representación estándar de un tablero de juego

La representación que se escogió para un tablero es mediante una matriz de caracteres de 6 por 6 elementos, la cual tiene las siguientes características:

- Cada pieza del juego es representada por una letra, desde la 'a' en adelante.
- Se reserva la letra 'x' para la pieza especial que será el objetivo del juego.
- Por defecto el agujero estará en la misma linea que el auto especial y no se representa.
- Los espacios en blanco son representados por el caracter '.'

A continuación se muestra un ejemplo de tablero en su representación de caracteres y su equivalente:

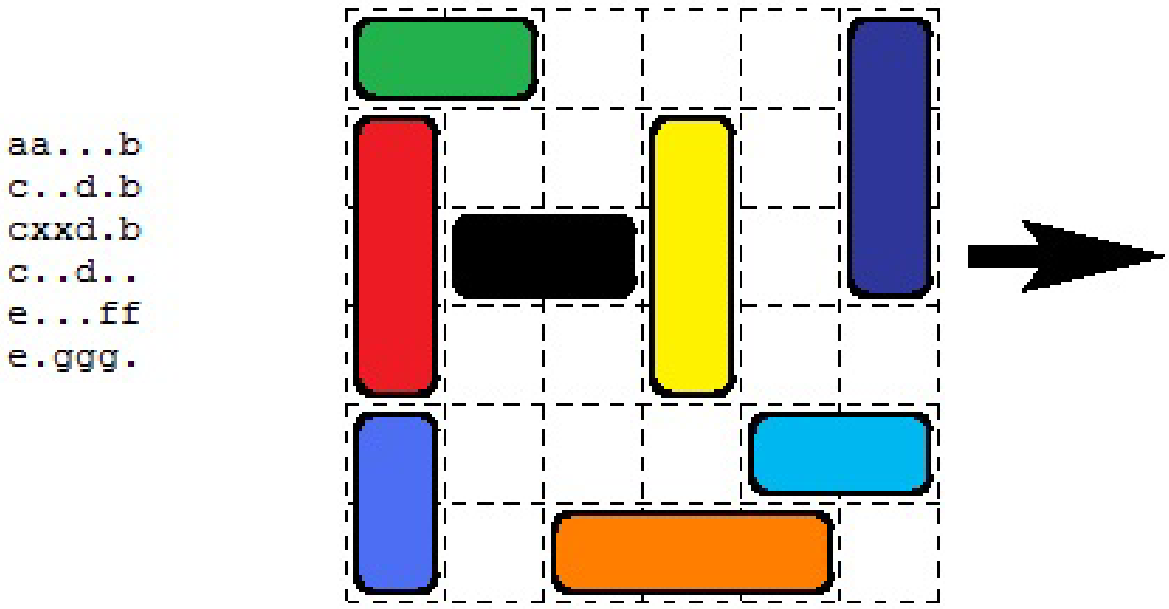

Figura 3.1: *Representación en caracteres de un tablero*

3.1.2. Archivo de definición del tablero original

En este archivo está contenido la definición del tablero inicial del juego, el cual deberá ser interpretado posteriormente por el Motor Gráfico para generar la visualización correspondiente. El archivo tiene como nombre tablero.tab y sólo contendrá una única matriz de caracteres. En este archivo el usuario podrá definir sus propios tableros manualmente.

3.1.3. Archivo de soluciones

Este archivo contiene un número entero positivo s, que es el número de pasos necesarios para resolver el tablero, a continuación se encuentran s matrices de caracteres, cada una de ellas representando cada movimiento que se debe seguir para solucionar el juego. Un ejemplo se puede observar en la Figura 3.2

```
17

bb...c
daae.c
d..e..
f...gg
f.hhh.

.bb..c
d..e.c
daae.c
d..e..
f...gg
f.hhh.

.bb...
d..e.c
daae.c
d..e.c
f...gg
f.hhh.

dbb...
d..e.c
daae.c
...e.c
f...gg
f.hhh.
```

Figura 3.2: *Representación de un Conjunto de Soluciones*

Este archivo tiene como nombre solucion.tab y es generado por el Motor de Juego a cada movimiento realizado por el usuario, actualizando el conjunto de soluciones.

3.1.4. Archivo de estado de juego

Este archivo contiene una matriz que es el Estado de Juego actual, inicialmente contiene la misma información que tablero.tab, pero a cada movimiento de una pieza el Motor Gráfico sobreescribe en este archivo el Estado Actual que representa la posición de las piezas en el tablero después de efectuado el movimiento, éste después será leído por el Motor de Juego para generar una solución a el Estado de Juego actualizado.

El nombre de este archivo es estado.tab y contiene una única matriz de caracteres.

3.2. Motor gráfico

Este módulo está encargado de la interfaz entre el usuario y el Sistema RHS, esta capa del sistema es la única forma por la cual el usuario puede interactuar con el sistema completo; tiene las siguiente funciones principales:

- Generar una visualización amigable al usuario que permita jugar y resolver un tablero de juego Rush Hour.
- Escuchar y recoger eventos del *mouse*, procesarlos mediante funciones específicas.
- Actualizar el Estado de Juego, dado un movimiento de cualquiera de las piezas en el tablero.
- Interpretar el conjunto de soluciones del tablero actual, dibujando una animación que muestre los pasos a seguir.
- Dar sugerencias sobre posibles movimientos del tablero en caso de que el usuario lo decida.

3.2.1. Descriptor de Tablero de Juego

Anteriormente se mostró como se representaba un tablero mediante una matriz de caracteres y los archivos que las incluyen. Sin embargo estas matrices sólo son utilizadas como datos de entrada y salida de los motores principales del juego.

Internamente cada motor principal no maneja de manera directa estas matrices, sino que con un procesamiento explota de ellas cierta información útil en la implementación de las distintas funciones que manipulan el Estado de Juego y el Conjunto de Soluciones del mismo. Por lo tanto se creó una interpretación lógica de un tablero de juego.

El Descriptor de Tablero es un número de tipo entero y positivo, el cual se calcula a partir de una matriz de caracteres. El proceso por el cual se obtiene este número es el siguiente:

1. El Descriptor de Tablero T, se inicializa en 0.
2. Dada un matriz $M_{6,6}$, para cada $M_{i,j}$:
   - Si $M_{i,j}$ es '.' se ignora y se continua.
   - Si $M_{i,j}$ es 'x' entonces $M_{i,j}$ = 'a'.
   - Si $M_{i,j}$ es cualquier otro caracter, $M_{i,j} = M_{i,j} + 1$.
   - Estos dos ultimos pasos se realizan para poder manejar una secuencia de caracteres continua sin preocuparnos por manejar el caracter 'x' por separado.

   A este proceso se le llama Cambio de Tablero (Figura 3.3)

Ejemplo:

```
aa...b                                  bb...c
c..d.b                                  d..e.c
cxxd.b  Proceso de Cambio de Tablero    daae.c
c..d..                                  d..e..
e...ff                                  f...gg
e.ggg.                                  f.hhh.
```

Figura 3.3: *Proceso de Cambio de Tablero*

3. Se construye una tabla que relacione un identificador p con la pieza representada por el caracter:

pieza	$\boldsymbol{p}$
a	0
b	1
c	2
d	3
e	4
...	...

Tabla 3.1: *Tabla de Identificadores de Piezas*

4. Dada la matriz cambiada M, para cada $M_{i,j}$:

   - Si $M_{i,j}$ es '.' se ignora y se continua.
   - Si no, se busca la primer aparición de cada pieza(caracter) dentro de la matriz y se averigua si tiene orientación vertical u horizontal de la siguiente manera:
     - Si $i+1 < 6$ y $M_{i,j} = M_{i+1,j}$ Entonces es una pieza en posición vertical

       $$T = T + (i5^p)$$

       donde p es el identificador de la pieza descrito en la Tabla 3.1

     - Si $j+1 < 6$ y $M_{i,j} = M_{i,j+1}$ Entonces es una pieza en posición horizontal

       $$T = T + (j5^p)$$

       donde p es el identificador de la pieza descrito en la Tabla 3.1

Al final se obtiene el Descriptor de Tablero.

Ejemplo del cálculo de un Descriptor de Tablero:

```
bb...c
d..e.c
daae.c
d..e..
f...gg
f.hhh.
```

Se guardan en una estructura ciertos datos:

pieza	***p***	**orientación**	**tamaño**	***i***	***j***	**posicion**
a	0	horizontal	2	N/A	2	2
b	1	horizontal	2	N/A	0	0
c	2	vertical	3	5	N/A	5
d	3	vertical	3	0	N/A	0
e	4	vertical	3	3	N/A	3
f	5	vertical	2	0	N/A	0
g	6	horizontal	2	N/A	4	4
h	7	horizontal	3	N/A	5	5

Tabla 3.2: *Tabla de datos sobre las piezas*

$$T = 2(5^0) + 0(5^1) + 5(5^2) + 0(5^3) + 3(5^4) + 2(5^5) + 4(5^6) + 5(5^7)$$

$$T = 2 + 0 + 125 + 0 + 1875 + 6250 + 62500 + 390625$$

$$T = 461375$$

Es preciso mencionar que este proceso es reverible, esto es, dado un Descriptor de Tablero y la Tabla con lo Datos sobre las piezas (Ver Tabla 3.2) T, es posible determinar a que matriz de caracteres corresponde, este proceso inverso se explica a continuación:

1. Para cada $M_{i,j}$

$$M_{i,j} = \text{'.'}$$

Para cada pieza P_k para $k = 0, 1, 2, ...$

- Si la orientación de P_k es vertical.

$$j = \text{posicion de } P_k.$$

$$i = T\,\%5$$

$$M_{i,j} = k+ \text{ 'a'}$$

$$M_{i+1,j} = k+ \text{ 'a'}$$

  Si tamaño de la pieza P_k es 3

$$M_{i+2,j} = k+ \text{ 'a'}$$

- Si la orientación de P_k es horizontal.

$$i = \text{posicion de } P_k.$$

$$j = T\,\%5$$

$$M_{i,j} = k+ \text{ 'a'}$$

$$M_{i,j+1} = k+ \text{ 'a'}$$

  Si tamaño de la pieza P_k es 3

$$M_{i,j+2} = k+ \text{ 'a'}$$

- $T = T/5$

De esta manera se vuelve a construir la Matriz $M_{i,j}$, a partir de su Descriptor de Tablero y la Tabla de Datos de la piezas.

3.2.2. Visualizacion, inicialización y renderización

Mediante el lenguaje C++ y la ayuda de la API de programación OpenGL es posible generar un modelo capaz de representar de manera amigable un tablero de juego. Para esto se definieron distintas clases para cada los elementos del juego. Esto nos permite asemejar nuestro modelo virtual a un modelo de la realidad. Las principales clases definidas son:

- **tablero**: La clase principal, define el tablero de juego y sobre de esta se hará la instancia de las demás clases del sistema.

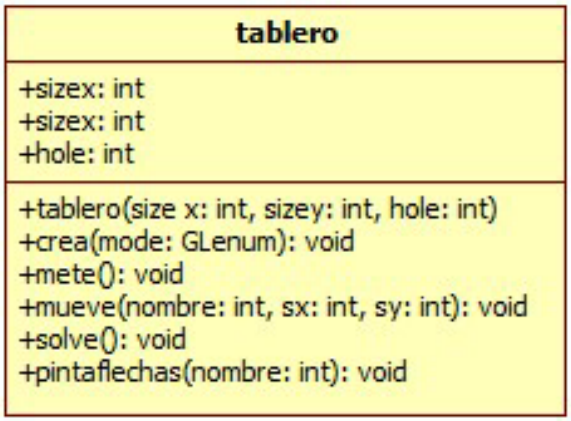

Figura 3.4: *Clase* ***tablero***

- **special**: Esta clase representa el objeto especial, el cual es el objetivo del juego.

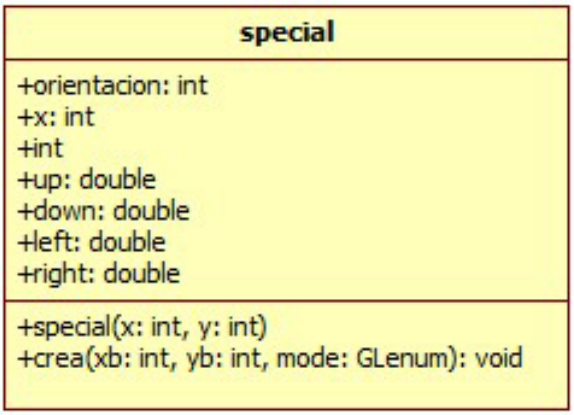

Figura 3.5: *Clase* ***special***

- **other**: Es una clase similar a la anterior, pero esta define las demás piezas, escencialmente la diferencia es el modo de construcción y dibujo de la pieza, la cual se detallará en capítulos posteriores.

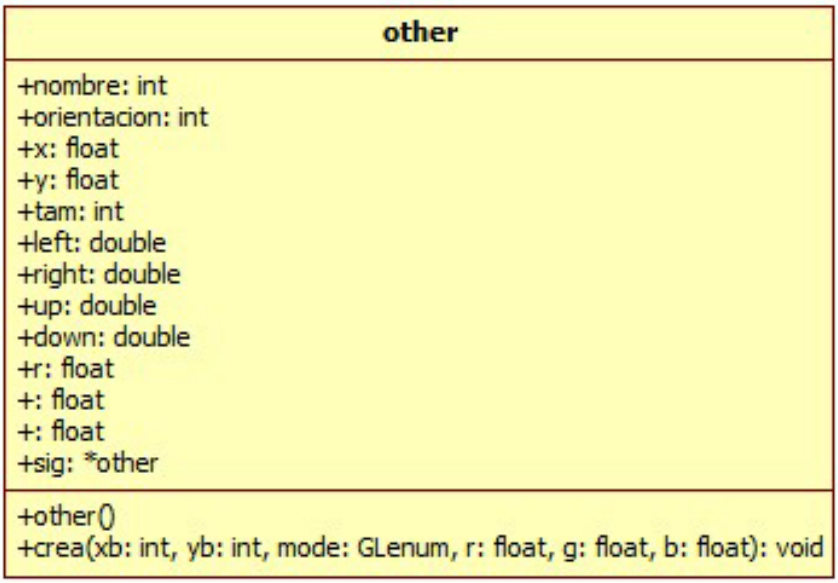

Figura 3.6: *Clase* ***other***

- **list**: Es una lista ligada que almacena a todos los objetos de la clase **other** para poder tener un mejor acceso a ella, esta clase es la abstracción de la Tabla de Datos sobre cada una de las piezas (Tabla 3.2) y cada entrada de esta tabla es un objeto de la clase **other**.

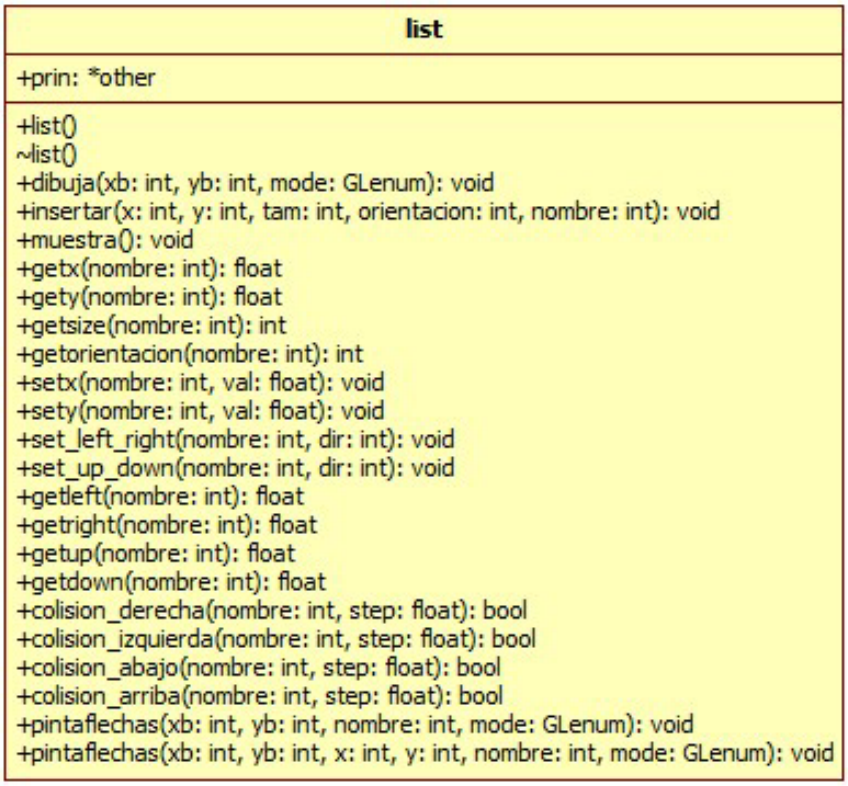

Figura 3.7: *Clase* ***list***

La clase principal **tablero** es la encargada de dibuja el tablero de juego, leer el Archivo de Definición del Tablero Original e interpretarlo, instancia la clase **list** e introduce en todas las piezas que estarán definidas por la clase **other**. La lista de objetos será utilizada para dibuja cada una de las piezas. El procedimiento se detalla en la Figura 3.8.

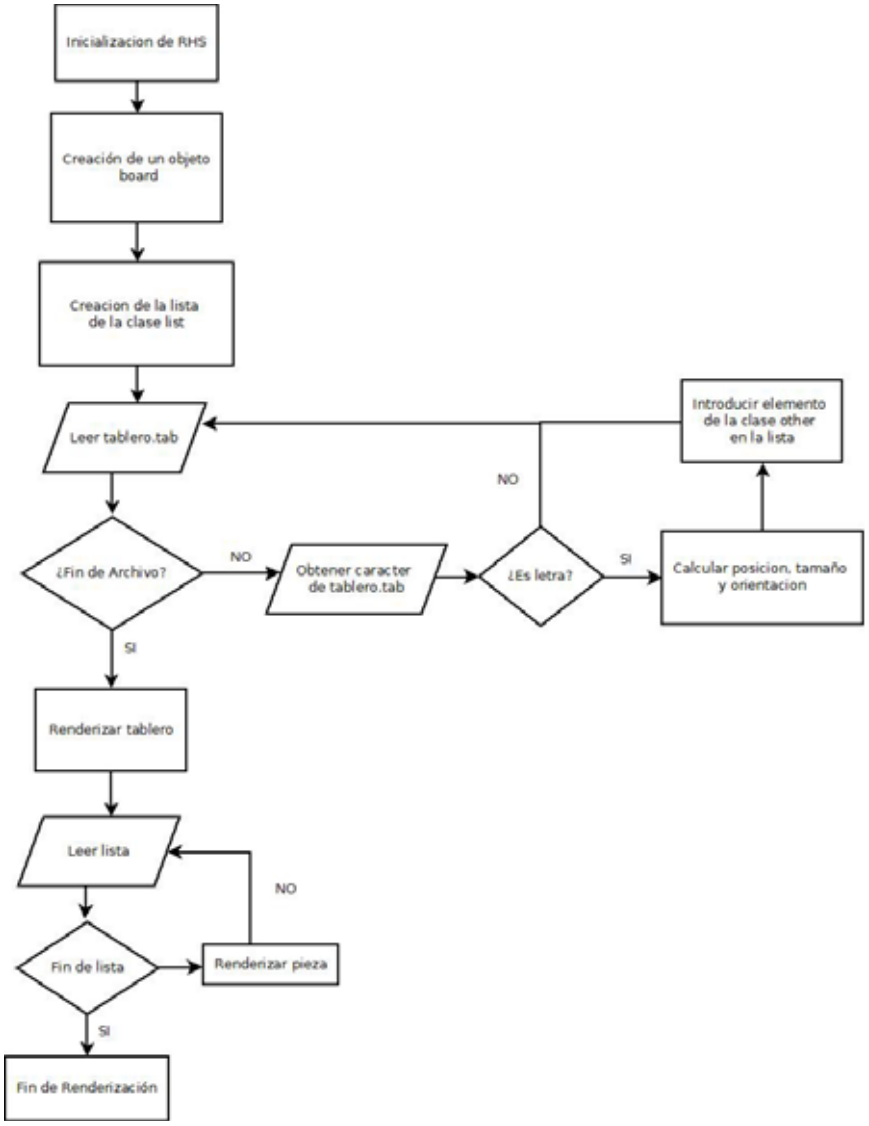

Figura 3.8: *Procedimiento de inicialización y renderización*

3.2.3. Actualización del estado de juego

Otra de las funciones que cumple el Motor Gráfico es el de escuchar y atender a los eventos del *mouse* y del teclado. Los eventos con el *mouse* son el movimiento de las piezas dentro del tablero. El movimiento de una pieza es el cambio en su posición vertical u horizontal, según sea el caso.

Cuando se produce un movimiento en el tablero, el Motor Gráfico tiene la tarea de actualizar la estructura definida por la clase **list**, introduciendo los nuevos valores en el objeto **other** correspondiente a la pieza a la que se le haya modificado su posición. Posteriormente el Estado de Juego descrito en el archivo estado.tab debe actualizarse con el fin de que en un futuro sea interpetado por el Motor de Juego.

La Actualización del Estado de Juego se logra de la siguiente manera:

1. Determinar el Descriptor de Tablero a a partir de el contenido de la estructura **list**, recorriendo cada uno de los elementos.
2. Construir la matriz de caracteres, a partir del Descriptor de Tablero obtenido y del contenido de la misma estructura **list**.
3. Sobreescribir el contenido de estado.tab con la nueva matriz de caracteres.

3.2.4. Interpetación del conjunto de soluciones

Cuando sea requerido, el sistema deberá leer el conjunto de datos contenidos en el archivo solucion.tab, interpretando cada matriz de caracteres como un paso para llegar a la solución, generando un animación de dicha solución:

1. A cada matriz $M_{i,j}$ leida en solucion.tab
    - Recalcular las posiciones de las piezas dado el nuevo tablero de juego y actualizar la estructura **list**.
    - Volver a dibuja las piezas a partir de los nuevos datos contenidos en dicha estructura.

Al final se deberá observar el Estado Final de Juego, El procedimiento se ilustra en el siguiente diagrama de flujo (Figura 3.9).

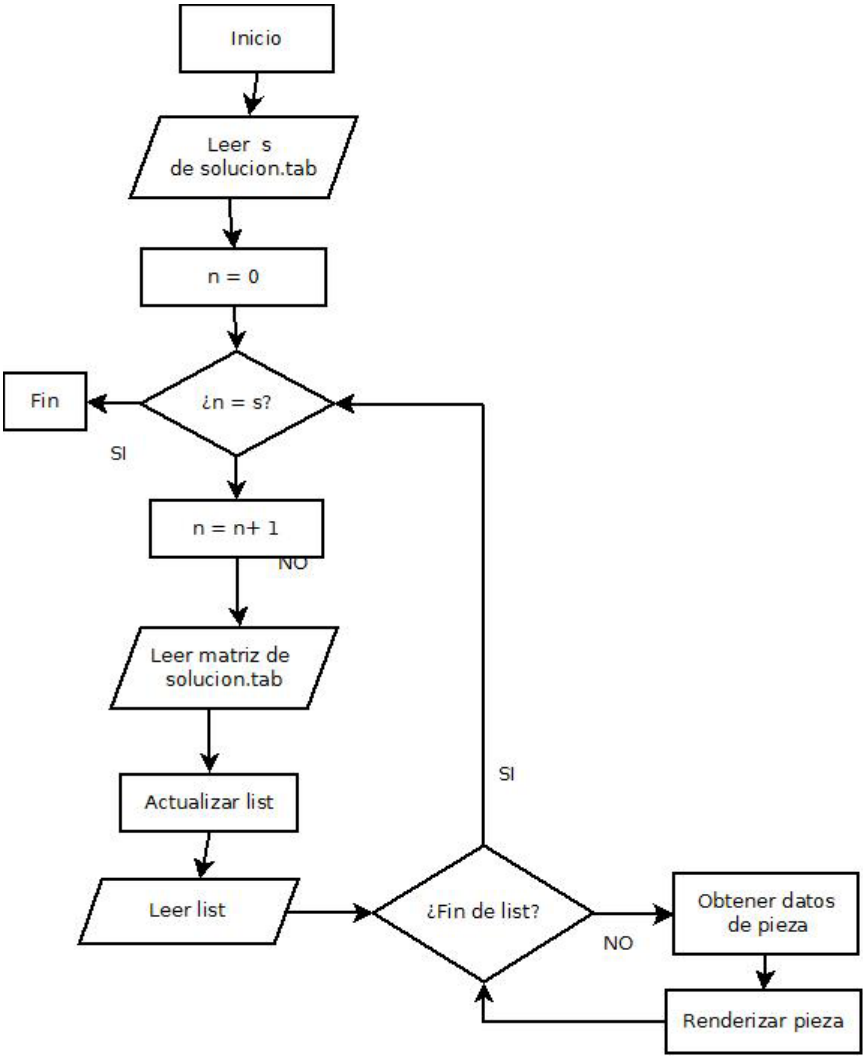

Figura 3.9: *Generación de la animación de solución*

3.2.5. Sugerencias de Solución

Esta opción consiste en mostrar una pequeña animación al usuario sobre el siguiente paso que debe seguir a efecto de solucionar el tablero, esto se logra leyendo la primer matriz del archivo solucion.tab, interpretarlo y mostrar el movimiento y volver al Estado de Juego actual. Este paso no debe actualizar el Estado de Juego Actual ni la estructura **list**.

3.3. Motor de Juego

Este módulo es el que lleva la algorítmica del sistema y es transparente al usuario, la única manera de interactuar con él es a través del Motor Gráfico como se detalla en la Arqutectura del Sistema explicada en la Figura 2.1, sus principales funciones son:

- Solucionar el tablero de juego a partir del Estado Actual de Juego y especificar los pasos a seguir en el Conjunto de Soluciones.
- Generar Tableros de Juego a través de un algoritmo genético que reciba como parámetros el número de movimientos necesarios para resolver el rompecabezas.

3.3.1. Solución de Tableros

Consiste en un algoritmo que resuelve el tablero, éste utiliza las funciones encargadas de obtener el Descriptor de Tablero y su función Inversa. Utiliza ciertas técnicas de Programación Dinámica y de Búsqueda a lo Ancho en el árbol que representa el espacio de soluciones del problema.

El algoritmo de Solucion de Tableros utiliza una estructura de datos temporal de nombre TD, la cual almacena información sobre las piezas, esta estructura es diferente a la definida por la clase **list**, debido a que TD, sólo se utiliza de manera temporal mientras se ejecuta el algoritmo, mientras que **list** es persistente durante todo el ciclo de vida del sistema.

TD almacena las siguientes características de las piezas del tablero:

- Posición.
- Longitud.
- Orientación.

El algoritmo escencialmente consiste en una búsqueda a lo ancho en el árbol del conjunto de soluciones y utiliza ciertos registros especiales que nos ayudan a ejercer la acción de poda en el árbol para evitar repetir posibles soluciones o incluso quedar atrapados en un ciclo. Estas estructuras son:

- Un buffer **power**: Almacena las potencias precalculadas de 5 para evitar realizarlas a cada paso y perder rendimiento del algoritmo:

1	5	25	125	625	3125	15625	78125	390625	1953125	...

Tabla 3.3: *Buffer de Potencias de 5*

- Un buffer **visit** : Cada posición i del buffer almacenará el número de pasos necesarios para llegar al Estado con el Descriptor de Tablero de valor i. Este buffer será el que nos ayudará a determinar el número mínimo de pasos a seguir para la solución.
- Un buffer **padre**: Almacenará en cada posicion i el Descriptor de Tablero Padre por el cual se llegó al Estado Actual con el Descriptor de Tablero de valor i, éste buffer nos será de utilidad para dar el Conjunto de Soluciones del Tablero.
- Un buffer **road**: Contendrá los Descriptores de Tablero que conforman el camino a la solución, escencialmente es el Conjunto de Soluciones.
- Un buffer **Q**: Esta es la cola de datos que nos ayudará a visitar cada Estado de Juego.
- Un buffer **board**: Es una matriz $B_{6,6}$ que almacena temporalmente el tablero de juego. Inicialmente su contenido será aquel que sea leído al inicio del algoritmo del archivo tablero.tab. Este tablero será cambiente durante la ejecución del Algoritmo de Soluciones, representando en cada paso el tablero correspondiente al Estado de Juego Actual.

Una vez inicializados todas estas estructuras de manera adecuada, el algoritmo comienza llenando la estructura TD, con el siguiente proceso:

Por cada $B_{i,j}$.

- Si $B_{i,j}$ = '.' ignorar y continuar.
- Si $B_{i,j}$ = 'x' , $p = 0$.
- Si $B_{i,j}$ es otro caracter, $p = B_{i,j}$ - 'a' +1.
- Si $i + 1 < 6$ y $B_{i,j} = B_{i+1,j}$
  - $orientacion_p = Vertical$
  - $posicion_p = j$
  - $longitud_p = 2$
  - Si $i + 2 < 6$ y $B_{i,j} = B_{i+2,j}$
    - $longitud_p = 3$

- Si $j+1<6$ y $B_{i,j}=B_{i,j+1}$
  - $orientacion_p = Horizontal$
  - $posicion_p = i$
  - $longitud_p = 2$
  - Si $j+2<6$ y $B_{i,j}=B_{i,j+2}$
    - $longitud_p = 3$

De esta manera se tiene llena la estructura TD, que será la que contendrá durante la ejecución del algoritmo la información necesaria de todas las piezas.

Ejemplo de una estructura TD:

```
aa...b
c..d.b
cxxd.b
c..d..
e...ff
e.ggg.
```

Se guardan en una estructura ciertos datos:

pieza	***p***	**orientación**	**tamaño**	**posición**
x	0	horizontal	2	2
a	1	horizontal	2	0
b	2	vertical	3	5
c	3	vertical	3	0
d	4	vertical	3	3
e	5	vertical	2	0
f	6	horizontal	2	4
g	7	horizontal	3	5

Tabla 3.4: *Estructura TD*

Realizado este paso se hace un Cambio de Tablero para poder trabajar con caracteres consecutivos y no preocuparnos por tratar el caracter 'x' por separado, tal y como se explicó en la Figura 3.3.

Todo lo explicado anteriormente corresponde sólo a la inicialización de todas las estructuras que se necesitan para poder ejecutar el algortimo, a continuación se explicarán todos los pasos para encontrar la solución y determinar el número total de movimientos necesarios para ello:

1. Para el primer Estado de Juego, que es el Tablero Original
   - Calcular el Descriptor de Tablero T del primer tablero

- Introducir en la cola **Q** T
- $\mathbf{visit}_T = 1$.

2. Mientras **Q**, no esté vacío.

   - Extraer valor V de la cola **Q**
   - Para cada elemento k en la estructura TD.
     - Convertir el Valor V a Tablero sobreescribiendo B con ayuda de la estructura TD.
     - $T = V$, guardamos el valor original de V en T.
     - Si $orientacion_k = Vertical$
       - $j = posicion_k$
       - $i = T\,\%5$
       - Mientras $i > 0$ y $B_{i-1,j} =$ '.', esto es realizar todos los movimientos posibles de la pieza hacia arriba
         - $B_{i-1,j} = Bi, j$
         - $B_{i+1,j} =$ '.'
         - Si $longitud_k = 3$

           $B_{i+2,j} =$ '.'

         - $A = T$, guardamos el Descriptor de Tablero dado a que va a ser modificado.
         - Obtener Descriptor de Tablero T a partir del nuevo tablero.
         - Si $\mathbf{visit}_T = 0$.

           $\mathbf{visit}_T = \mathbf{visit}_A + 1$.
           Metemos T a la cola **Q**
           $\mathbf{padre}_T = A$

         - $i = i$ - 1.

- Mientras $i+longitud_k < 6$ y $B_{i+longitud_k,j} =$ '.', esto es realizar todos los movimientos posibles de la pieza hacia abajo.
  - $B_{i+longitud_k,j} = Bi, j$
  - $B_{i,j} =$ '.'
  - $A = T$, guardamos el Descriptor de Tablero dado a que va a ser modificado
  - Obtener Descriptor de Tablero T a partir del nuevo tablero.
  - Si $\mathbf{visit}_T = 0$.

    $\mathbf{visit}_T = \mathbf{visit}_A + 1$.
    Metemos T a la cola **Q**
    $\mathbf{padre}_T = A$

  - $i = i + 1$.
- Si $orientacion_k = Vertical$
  - $i = posicion_k$
  - $j = T\,\%5$
  - Mientras $j > 0$ y $B_{i,j-1} =$ '.', esto es realizar todos los movimientos posibles de la pieza hacia la izquierda.
    - $B_{i,j-1} = Bi, j$
    - $B_{i,j+1} =$ '.'
    - Si $longitud_k = 3$

      $B_{i,j+2} =$ '.'

    - $A = T$, guardamos el Descriptor de Tablero dado a que va a ser modificado.
    - Obtener Descriptor de Tablero T a partir del nuevo tablero.
    - Si $\mathbf{visit}_T = 0$.

      $\mathbf{visit}_T = \mathbf{visit}_A + 1$.
      Metemos T a la cola **Q**
      $\mathbf{padre}_T = A$

    - $j = j$ - 1.

- Mientras $j+longitud_k < 6$ y $B_{i,j+longitud_k} =$ '.', esto es realizar todos los movimientos posibles de la pieza hacia la derecha
  - $B_{i,j+longitud_k} = Bi, j$
  - $B_{i,j} =$ '.'
  - $A = T$, guardamos el Descriptor de Tablero dado a que va a ser modificado
  - Obtener Descriptor de Tablero T a partir del nuevo tablero.
  - Si $\mathbf{visit}_T = 0$.

    $\mathbf{visit}_T = \mathbf{visit}_A + 1$.
    Metemos T a la cola **Q**
    $\mathbf{padre}_T = A$
    Si $k = 0$ **y** $j + longitud_k = 5$. **Fin del algoritmo** $sol = \mathbf{visit}_T$
  - $j = j + 1$.
- $V = V/5$.

- Si la cola **Q**, queda vacía, entonces el tablero no tiene solución.

Si el algoritmo termina satisfactoriamente, entonces la variable *sol* contiene el número de pasos necesarios para resolver el tablero y T contiene el Descriptor del Tablero en donde la pieza especial puede salir libremente. Además la estructura padre está llena con la información de los Estados Padre, lo cual nos ayudará a construir el camino que nos dió la solución al problema

El camino lo construiremos en el buffer **road** de la siguiente manera:

a) Inicializamos $p = 0$;

b) Creamos una variable *paso* que inicialmente tiene el Descriptor del primer tablero.

c) Mientras $paso \neq T$
- $\mathbf{road}_p = paso$
- $paso = \mathbf{padre}_{paso}$
- $p = p + 1$.

Este pequeño algoritmo nos introduce el camino en orden inverso en el buffer **road**, además de que sólo contiene Descriptores de tablero. Así que para escribirlo en el archivo solucion.tab, se ejecuta lo siguiente:

a) Escribe *sol* en solucion.tab.

b) Mientras $p \geq 0$
- Convierte el valor $\mathbf{road}_p$ a Tablero
- Escribe el Tablero en solucion.tab
- $p = p$ - 1.

3.3.2. Generación de Tableros

Para esta parte el objetivo era generar tableros a partir de un algoritmo genético, sin embargo no se nos ocurrió como realizar las mutaciones adecuadamente para poder hacer que se pudiera generar un tablero más complejo que el que se genera aleatoriamente al inicio del algoritmo. Por lo anterior optamos por utilizar tableros ya predeterminados y utilizar estos para nuestra aplicación.

Se parte de estos tableros predeterminados y se elabora un camino en retorceso para dar un tablero nuevo.

Capítulo 4

Implementación

4.1. Definición de Clases

A continuación se muestran las definiciones en codigo C++ de la clases principales que se utilizan para que el sistema pueda funcionar correctamente:

La clase principal **tablero**, sus atributos contienen información como el tamaño de la cuadrícula y la posición del agujero por donde deberá salir el objetivo.

```
1 class tablero
2 {
3         public:
4                 int sizex, sizey;
5                 int hole;
6
7                 tablero(int sizex, int sizey, int hole);
8                 void crea(GLenum mode);
9                 void mete();
10                void mueve(int nombre, int sx, int sy);
11                void solve();
12                void pintaflechas(int nombre);
13 };
```

La clase **special** define la pieza especial, la cual es el objetivo de juego.

```
1 class special
2 {
3         public:
4                 int orientacion;
5                 int x, y;
6                 double up, down, left , right;
7
8                 special(int x, int y);
9                 void crea(int xb, int yb, GLenum mode);
10 };
```

La clase **other** define a las demás piezas del juego:

```
1 class other
2 {
3         public:
4                 int nombre;
5                 int orientacion;
6                 float x, y;
7                 int tam;
8                 double left , right , up, down;
9                 other *sig;
10                float r, g, b;
11
12                other(int x, int y, int tam, int orientacion , int nombre);
13                void crea(int xb, int yb, GLenum mode, float r, float g, float
14 };
```

La clase **list** es la lista que contiene a todos lo objetos **other**.

```
1 class list
2 {
3         public:
4                 other *prin;
5
6                 list();
7                 ~list();
8                 void dibuja(int xb, int yb, GLenum mode);
9                 void insertar(int x, int y, int tam, int orientacion, int
10                float getx(int nombre);
11                float gety(int nombre);
12                int getsize(int nombre);
13                int getorientacion(int nombre);
14                void setx(int nombre, float val);
15                void sety(int nombre, float val);
16                void set_left_right(int nombre, int dir);
17                void set_up_down(int nombre, int dir);
18                float getleft(int nombre);
19                float getright(int nombre);
20                float getup(int nombre);
21                float getdown(int nombre);
22                bool colision_derecha(int nombre, float step);
23                bool colision_izquierda(int nombre, float step);
24                bool colision_abajo(int nombre, float step);
25                bool colision_arriba(int nombre, float step);
26                void pintaflechas(int xb, int yb, int nombre, GLenum mode)
27                void pintaflechas(int xb, int yb, int x, int y, GLenum mod
28                void actualiza();
29 };
```

El funcionamiento debido de la inicialización de estas clases y de la generación de la visualización se explica en la Figura 3.8.

4.2. Algoritmos del motor de juego

4.2.1. Solución automática de tablero

A continuación se mostrarán los códigos de todas las funciones involucradas en el algoritmo.

- Instanciación y creación de los búfferes y estructuras necesarias.

```
#define HOR 1
#define VER 0
struct TD
{
        int pos;
        int len;
        int type;

}Car[10];

char board[6][7];
char visit[10000000];
int padre[10000000];
int road[1000];
int Q[10000000];
int N;
int res;
int target, origen;
int power[] = {1, 5, 25, 125, 625, 3125, 15625, 78125, 390625, 1953125};
```

- La función Guarda_Posicion mostrada a continuación, llena la estructura TD con la matriz de caracteres leída con anterioridad.

```
1 void Guarda_Posicion()
2 {
3         int i, j, car_name;
4         for(i = 0; i < 6; i++)
5         {
6                 for(j = 0; j < 6; j++)
7                 {
8                         if(board[i][j] == '.') continue;
9                         if(j > 0 && board[i][j - 1] == board[i][j]) continue;
10                        if(i > 0 && board[i - 1][j] == board[i][j]) continue;
11                        if(board[i][j] == 'x') car_name = 0;
12                        else  car_name = board[i][j] - 'a' + 1;
13                        N++;
14                        if(i + 1 < 6 && board[i][j] == board[i + 1][j])
15                        {
16                                Car[car_name].pos = j;
17                                Car[car_name].len = 2;
18                                Car[car_name].type = VER;
19                                if(i + 2 < 6 && board[i][j] == board[i + 2][j])
20                                {
21                                        Car[car_name].len = 3;
22                                }
23                        }
24                        else if(j + 1 < 6 && board[i][j] == board[i][j + 1])
25                        {
26                                Car[car_name].pos = i;
27                                Car[car_name].len = 2;
28                                Car[car_name].type = HOR;
29                                if(j + 2 < 6 && board[i][j] == board[i][j + 2])
30                                {
31                                        Car[car_name].len = 3;
32                                }
33                        }
34                }
35        }
36 }
```

- Las funciones de Tablero_a_Valor y su función inversa.

```
1 int Tablero_a_Valor()
2 {
3         int i, j, car_name, val = 0;
4         for(i = 0; i < 6; i++)
5         {
6                 for(j = 0; j < 6; j++)
7                 {
8                         if(board[i][j] == '.') continue;
9                         if(j > 0 && board[i][j - 1] == board[i][j]) continue;
10                        if(i > 0 && board[i - 1][j] == board[i][j]) continue;
11                        car_name = board[i][j] - 'a';
12                        if(i + 1 < 6 && board[i][j] == board[i + 1][j])
13                        {
14                                val += power[car_name] * i;
15                        }
16                        else if(j + 1 < 6 && board[i][j] == board[i][j + 1])
17                        {
18                                val += power[car_name] * j;
19                        }
20                }
21        }
22        return val;
23
```

```
24 }
25
26 void Valor_a_Tablero(int val)
27 {
28         int i, j, k;
29         for(i = 0; i < 6; i++)
30         {
31                 for(j = 0; j < 6; j++)
32                         board[i][j] = '.';
33         }
34         for(k = 0; k < N; k++)
35         {
36                 if(Car[k].type == VER)
37                 {
38                         j = Car[k].pos;
39                         i = val % 5;
40                         board[i][j] = k + 'a';
41                         board[i + 1][j] = k + 'a';
42                         if(Car[k].len == 3) board[i + 2][j] = k + 'a';
43                 }
44                 else if(Car[k].type == HOR)
45                 {
46                         i = Car[k].pos;
47                         j = val % 5;
48                         board[i][j] = k + 'a';
49                         board[i][j + 1] = k + 'a';
50                         if(Car[k].len == 3) board[i][j + 2] = k +'a';
51                 }
52                 val /= 5;
53         }
54 }
```

La función Tablero_a_Valor calcula el Descriptor de Tablero a partir del buffer **board** y del contenido de la estructura TD, mientras que la función Valor_a_Tablero hace el proceso inverso, esto es, dado un Descriptor de tablero y la estructura TD, reconstruye el buffer **board**.

- La parte central del algoritmo se encuentra en la implementación de la función que recorre el árbol de soluciones del tablero.

```
1  int Resuelve()
2  {
3          int rear = 0;
4          int front = 0;
5          int val, new_val, org_val;
6          int temp;
7          int i, j, k;
8          Q[rear] = Tablero_a_Valor();
9          val = Q[rear];
10         origen = val;
11         visit[Q[rear++]] = 1;
12         padre[0] = origen;
13         j = val % 5;
14
15         if(j + Car[0].len == 6)
16                 return 1;
17
18         while(front <= rear)
19         {
20                 org_val = Q[front++];
21                 val = org_val;
22
23                 for(k = 0; k < N; k++)
24                 {
25                         Valor_a_Tablero(org_val);
26                         temp = org_val;
27                         if(Car[k].type == VER)
28                         {
29                                 j = Car[k].pos;
30                                 i = val % 5;
31                                 while(i > 0 && board[i - 1][j] == '.')
32                                 {
33                                         board[i - 1][j] = board[i][j];
34                                         if(Car[k].len == 3)board[i + 2][j] = '.';
35                                         else board[i + 1][j] = '.';
36                                         new_val = Tablero_a_Valor();
37                                         if(visit[new_val] == 0)
38                                         {
39                                                 visit[new_val] = visit[org_val] + 1;
40                                                 Q[rear++] = new_val;
41                                                 padre[new_val] = temp;
42                                         }
43                                         i--;
44                                         temp = new_val;
45                                 }
46                                 while(i+Car[k].len<6 && board[i+Car[k].len][j]=='.')
47                                 {
48                                         board[i + Car[k].len][j] = board[i][j];
49                                         board[i][j] = '.';
50                                         new_val = Tablero_a_Valor();
51                                         if(visit[new_val] == 0)
52                                         {
53                                                 visit[new_val] = visit[org_val]+1;
54                                                 Q[rear++] = new_val;
55                                                 padre[new_val] = temp;
56                                         }
57                                         i++;
58                                         temp = new_val;
59                                 }
60                         }
61
62
63
```

```
64                          else if(Car[k].type == HOR)
65                          {
66                                  i = Car[k].pos;
67                                  j = val % 5;
68                                  while(j > 0 && board[i][j - 1] == '.')
69                                  {
70                                          board[i][j - 1] = board[i][j];
71                                          if(Car[k].len == 3)board[i][j + 2] = '.';
72                                          else board[i][j + 1] = '.';
73                                          new_val = Tablero_a_Valor();
74                                          if(visit[new_val] == 0)
75                                          {
76                                                  visit[new_val] = visit[org_val] + 1;
77                                                  Q[rear++] = new_val;
78                                                  padre[new_val] = temp;
79                                          }
80                                          temp = new_val;
81                                          j--;
82                                  }
83                                  while(j + Car[k].len < 6 && board[i][j + Car[k].len] == '.')
84                                  {
85                                          board[i][j + Car[k].len] = board[i][j];
86                                          board[i][j] = '.';
87                                          new_val = Tablero_a_Valor();
88                                          if(visit[new_val] == 0)
89                                          {
90                                                  visit[new_val] = visit[org_val] + 1;
91                                                  Q[rear++] = new_val;
92                                                  padre[new_val] = temp;
93                                                  if(k == 0 && j + Car[k].len == 5)
94                                                  {
95                                                          target = new_val;
96                                                          return visit[new_val];
97                                                  }
98                                          }
99                                          temp = new_val;
100                                         j++;
101                                 }
102                         }
103                         val /= 5;
104         }
105         return -1;
106 }
```

- Por último en la siguiente implementación se explica como es que se construye el camino que conforma el Conjunto de Soluciones del Tablero de Juego Original.

```
1  void Crea_Camino()
2  {
3          int step = target, p = 0, i;
4          while(step != origen)
5          {
6                  road[p++] = step;
7                  step = padre[step];
8          }
9          road[p] = step;
10         fprintf(sal, "%d\n\n", p + 1);
11         for(i = p; i >= 0; i--)
12         {
13                 Valor_a_Tablero(road[i]);
14                 Imprime_Tablero();
15         }
16 }
```

Capítulo 5

Pruebas y resultados

5.1. Configuración inicial

La prueba de funcionamiento se realizó en un equipo con las siguientes características de software y hardware:

- Memoria RAM 2 GB
- Memoría de Vídeo 256 MB
- Sistema Operativo Windows 7
- Framework .NET
- Visual C++ 2008 Express Edition
- Biblioteca OpenGL 2.1

La configuración inicial es verificar que se cuenta con todos estos requerimientos e incluir los siguientes archivos en sus correspondientes rutas:

- glut.h C:/Program Files/Microsoft SDKs/Windows/v6.0A/Include/gl
- glut32.lib C:/Program Files/Microsoft SDKs/Windows/v6.0A/Lib
- glut32.lib C:/Program Files/Microsoft SDKs/Windows/v6.0A/Lib/x64
- glut32.dll C:/WINDOWS/system o C:/WINDOWS/system32 dependiendo del sistema operativo y el procesador de 64 o 32 bits

5.2. Pruebas

Se probó en 2 equipos distintos con las mismas características, un equipo era una computadora de escritorio y el segundo una portátil.

Se inició la interfaz gráfica, leyendo de manera correcta el archivo tablero.tab, después se comenzó probando cada una de las opciones que ésta ofrece de manera repetida y verificando que se actualicen los archivos correspondientes.

A continuación se muestran detalles de cada una de las opciones del sistema:

a) Opción de Juego.

- Al posicionar el puntero en cualquiera de las piezas, se muestran los posibles movimientos que se pueden realizar con la pieza seleccionada.
- Se debe hacer *click* en una de las direcciones y en el número de espacios que se quiera avanzar, sólo entonces se cambia la posición de la pieza y se actualiza el Estado de Juego de manera satisfactoria en el archivo estado.tab
- El juego termina cuando se logra sacar la pieza especial del tablero.

b) Opción de Solución del Tablero

- Al seleccionar esta opción se puso en marcha el algoritmo de solución del tablero, el cual se ejecutó con un buen rendimiento y de manera correcta escribio los datos correspondientes al Conjunto de Soluciones en el archivo solucion.tab.
- Se mostró por medio de la interfaz gráfica la animación de la solución dada y que estaba descrita en el archivo solucion.tab.

c) Opción de Sugerencia

- Al seleccionar esta opción, la interfaz gráfica muestra sobre la pieza el proximo movimiento a seguir dentro del Conjunto de Soluciones.

5.3. Resultados

En este apartado se describió el uso del sistema RHS. En general los resultados obtenidos fueron satisfactorios, no hubo problemas en cuanto al tiempo de ejecución de los algoritmos, y el rendimiento general del sistema es bueno, de igual manera se puede hablar de la correctitud de los algoritmos empleados en el Motor de Juego.

Sin embargo al inicio del desarrollo del proyecto se planteaba la opción de que las dimensiones del tablero de juego podían ser de diferentes tamaños a las estándares de 6×6, esto no se logró debido a que se utilizan potencias relacionadas a dichas dimensiones, en el caso del tablero de 6×6, se utilizan potencias de 5, el peor de los casos es 5^k, donde k es el número de piezas en juego, esto implica para un tablero de $n \times n$, en el peor caso la mayor potencia es $(n-1)^{n^2/2}$.

Si consideramos un caso concreto en donde el tablero sea de dimensiones 20×20 como el máximo, entonces el peor de los casos se daría si el tablero está lleno de piezas de longitud 2, o sea existen 200 piezas en juego, esto implica el cálculo del número 19^{200}, lo cual es un número lo suficientemente grande como para reducir el tiempo de ejecución del algoritmo en gran medida. Esto reduce considerablemente el número de piezas que se pueden incluir en el tablero y aumenta la cantidad de estados en el espacio de soluciones que se deben explorar, esto implica un gasto enorme de memoria y una reducción en el rendimiento de los algoritmos involucrados en la solución del tablero. Por tal razón se optó por mantener fija la dimensión del tablero a su estándar de 6×6.

Además el algoritmo genético que se mencionaba en la propuesta cambió por un algoritmo en el cual se generan tableros a partir de otros ya preestablecidos.

Con excepción del detalle antes mencionado se alcanzaron los objetivos que se buscaban completar con el desarrollo del proyecto.

Capítulo 6

Conclusiones y trabajos futuros

6.1. Conclusiones

El uso de distintas técnicas de diseño y análisis de algoritmos fueron de gran ayuda para el desarrollo del proyecto, ya que gracias a ello se logró encontrar una manera factible de encontrar la solución óptima al problema original del juego. Sin embargo sabemos que la naturaleza del juego es de tipo combinatorio y eso implica que al aumentar el tamaño del problema también lo hace su espacio de soluciones, lo cual puede generar complicaciones para el algoritmo diseñado en solucionar el juego, reduciendo entre otras cosas su rendimiento.

También es merecido destacar que el uso de una interfaz gráfica para poder visualizar e influir de manera directa sobre los resultados de un procedimiento es hoy en día muy necesario y este proyecto no es la excepción, ya que gracias al uso de un modelo gráfico pudimos generar nuevos Estados de Juego, los cuales el algoritmo de solución toma como datos de entrada, además se pudo observar el Conjunto de Soluciones de una manera más amigable.

Por este hecho, se pueden realizar simulaciones de todo tipo, desde la simplicidad de un Tablero de Juego a una representación de un sistema complejo, lo cual es de gran ayuda para la investigación de cualquier ramo y herramientas como OpenGL nos ofrecen un conjunto de soluciones para lograr este objetivo, reduciendo el tiempo de desarrollo al brindarnos funciones especializadas en optimizar el rendimiento del proceso de generar modelos que representen fenómenos o hechos que sirvan como caso de estudio.

En conjunto, el uso de una interfaz gráfica y el diseño de algoritmos nos brindan una herramienta poderosa para la investigación sobre la búsqueda de métodos para la solución de problemas específicos, ya que nos permite visualizar de una mejor forma el comportamiento de éstos y el resultado de los mismos.

6.2. Trabajos futuros

Un trabajo posterior a este proyecto podría ser la mejora del algoritmo de solución, esto es, desarrollar una adaptación a éste para que sea posible resolver tableros de juego más grandes y que puedan incluir un gran número de piezas que se encuentren dentro. De igual manera se pueden incluir nuevas reglas de juego, tales como involucrar más piezas objetivo, piezas de mayor o menor longitud, aumentando el nivel de complejidad de los tableros.

Como se mencionaba en la propuesta del proyecto también es posible diseñar un tablero de juego tridimensional como un cubo, en donde las piezas se muevan también en profundidad teniendo distintos niveles, aumentando así el espacio de soluciones del problema del rompecabezas y haciendo más desafiante el desarrollo de una visualización que permita un juego amigable.

Dentro a lo que se refiere a la graficación, es posible aumentar el rendimiento de ésta, además buscar un grado mayor de realismo en la construcción del modelo virtual y agregarle más características que hagan una mejor experiencia de juego, como incluir sonido con bibliotecas como OpenAL o agregar nuevas opciones al juego y buscar crear una mejor interfaz gráfica de usuario.

Apéndice A

Archivos fuente y bibliotecas

A.1. Archivos fuente

Arbol de directorios de los archivos fuente.

```
RUSH
|--stdafx.h
|--librush.h
|--tga.h
|--targetver.h
|--targetver.cpp
|--rush.cpp
|--board.h
|--board.cpp
|--special.h
|--special.cpp
|--other.h
|--other.cpp
|--Juego
|--|--solver.cpp
|--|--generator.cpp
|--Interaccion
|--|--tablero.tab
|--|--estado.tab
|--|--solucion.tab
```

A.2. Bibliotecas utilizadas

```
windows.h
stdio.h
tchar.h
stdlib.h
time.h
math.h
string.h
GL/glut.h
GL/gl.h
GL/glu.h
```

Bibliografía

[1] Página oficial de OpenGL *http://www.opengl.org/* consultada el 30 de Agosto del 2010.

[2] Página oficial del juego Rush Hour. *http://www.puzzles.com/products/rushhour.htm* consultada el 2 de Agosto del 2010.

[3] *Algoritmos en C++*. Robert Sedgewick Editorial Pearson.

[4] *An introduction to genetic algorithms for scientists and engineers*.David A. Coley Editorial World Scientific Publishing Company