

**AUTOCONSTRUCCIÓN DE LABERINTOS UNICURSALES  
Y SU RESPECTIVA SIMULACIÓN 3D USANDO VRML**

**ANDRÉS FELIPE PEÑA CASTRO  
JIMMY ALEXANDER OJEDA  
DAVID PARDO VALLEJO**

**FUNDACIÓN UNIVERSITARIA SAN MARTIN  
UNIVERSIDAD ABIERTA, DISTANCIA Y VIRTUAL  
PROGRAMA DE INGENIERIA DE SISTEMAS  
BOGOTÁ  
2012**

# TABLA DE CONTENIDO

1. INTRODUCCIÓN.....	4
2. ESPECIFICACIÓN .....	5
2.1 Requerimiento.....	5
2.2 Tecnología .....	5
3. PLANIFICACIÓN .....	6
3.1 Etapas de construcción .....	6
3.2 Etapas de prueba .....	6
4. MUESTREO .....	7
4.1 Laberinto unicursal .....	7
4.2 Laberinto multicursal.....	8
5. DISEÑO .....	10
5.1 Tecnología .....	10
5.2 Autoconstrucción del laberinto .....	10
5.3 Diagrama de clases.....	12
5.4 Algoritmo de resolución .....	12
6. CONSTRUCCIÓN .....	14
5.1 Algoritmo de autoconstrucción .....	14
5.2 Renderizado 2D.....	20
5.3 ÁRBOL 4-ARIO PARA RESOLVER LABERINTOS.....	25
5.4 Objetivos de búsqueda (Bonus) .....	28
5.5 Inicios de VRML.....	31
5.6 Generación de código con T4 .....	32
5.7 Guardar en disco el mundo virtual.....	42
5.8 Viewpoints .....	43
5.9 Interacción con el usuario .....	45
7. PRUEBAS.....	47
6.1 PRUEBA DE ESCRITORIO 4X4 (CONSOLA) .....	47
6.2 Pruebas de autoconstrucción (Winform).....	48
6.3 Pruebas de resolución con un Árbol 4-Ario.....	53
6.4 Pruebas con búsqueda de objetivos aleatorios .....	57

8.	PUBLICACIÓN.....	60
8.1	Instalador local (Desktop).....	60
8.2	Página Web (ASP) .....	64
9.	VERSIONAMIENTO.....	65
10.	CONCLUSIONES.....	67

# 1. INTRODUCCIÓN

Simular situaciones de la vida diaria a través de modelos es quizás uno de los desafíos más importantes para cualquier ingeniero de sistemas. Las aplicaciones son innumerables cuando consideramos la posibilidad de atender necesidades y mejoras que podríamos hacer en la sociedad por medio de nuestros conocimientos. Sin embargo, llegar a hacer una simulación rigurosa puede llegar a ser una tarea con un esfuerzo importante y resulta necesario iniciarse antes de abordar toda la amplitud de este campo.

Consideremos por ejemplo hablar de matemática. La mayoría de problemas básicos, incluido el laberinto del presente trabajo, se pueden solucionar con una matemática simple y lineal. Por el contrario, los problemas apremiantes y de gran utilidad para el ciudadano de a pie, requiere muchas veces de una matemática más compleja. Desde las ecuaciones cuadráticas utilizadas para simular la caída libre, hasta los crecimientos exponenciales poblacionales, hay problemas que exigen una aguda destreza para una simulación exitosa.

Por esta razón, es importante promover este tipo de problemáticas asociadas a juegos entre los estudiantes que apenas inician en la simulación. Está demostrado que los juegos son una gran forma de mejorar las habilidades para desarrollar software, porque su construcción supone el aprendizaje constante a través de las reglas de ese juego. Si como en este caso, se agrega un modelo para la generación aleatoria de casos a simular, se tienen excelentes ingredientes para un aprendizaje significativo.

Se presenta pues, un trabajo en el cual se siguió la metodología propuesta para modelar mundos de realidad virtual. En este proyecto se avanzó desde la construcción más básica de software, para luego a través de sucesivas mejoras y funcionalidades, se avanza hacia una muy interesante y parametrizable herramienta para la creación de laberintos virtuales.

## **2. ESPECIFICACIÓN**

### **2.1 Requerimiento**

- El objetivo es utilizar la teoría de autoconstrucción de laberintos para construir un mundo virtual que los represente.
- Un avatar debe ubicarse en el mundo virtual y realizar el recorrido que resuelva el laberinto.
- Se deben utilizar texturas en algunos de los elementos del mundo virtual.
- Se debe proveer al menos un punto de vista personalizado.
- El usuario debe por lo menos ser capaz de seleccionar el ancho y el alto del laberinto a autoconstruir.
- No se podrán generar dos laberintos iguales durante las pruebas. Al ser aleatorios, es muy baja la probabilidad de repetición.

### **2.2 Tecnología**

La tecnología para el mundo virtual es VRML. Por no tratarse de un lenguaje en sí mismo, se puede utilizar cualquier otro lenguaje para que al final el resultado sea el mundo virtual.

### 3. PLANIFICACIÓN

Para este proyecto se cuenta con un grupo de tres estudiantes. Todas las etapas descritas en la metodología deben realizarse antes del jueves 29 de noviembre, fecha en la que se hará la sustentación del software funcionando. Dentro de la clasificación por complejidad de proyecto, se considera que el presente es un proyecto simple. Esto es porque el modelo a utilizar será bastante controlado y no ofrece más complejidad que la matemática lineal y del sistema de coordenadas necesario para realizarlo.

#### 3.1 Etapas de construcción

Para no abordar inmediatamente el problema en toda su complejidad usando VRML, se siguieron una serie de etapas en la construcción, desde lo más sencillo como lo es una aplicación de consola, hasta lo más complejo que es un generador de código.



#### 3.2 Etapas de prueba

Para las pruebas, se documentarán los avances y depuración de las etapas del punto anterior de construcción. Al tratarse de laberintos autoconstruidos, será importante probar varias generaciones distintas, con diferentes dimensiones y parámetros para validar que el software se está desarrollando correctamente.

## 4. MUESTREO

Para este caso, las muestras a utilizar durante la simulación serán los laberintos de prueba mostrados durante el diseño. Hay muchos otros tipos de laberinto que podrían tomarse como antecedentes, pero que se salen de la teoría de autoconstrucción que se utilizará.

### 4.1 Laberinto unicursal

Es aquel trazado en el que desde el punto de entrada hasta el centro (o el punto de llegada) no es necesario, ni posible, tomar ninguna elección durante el recorrido. En este tipo de trazado, estemos en el punto en que estemos del mismo, sabemos que las pared

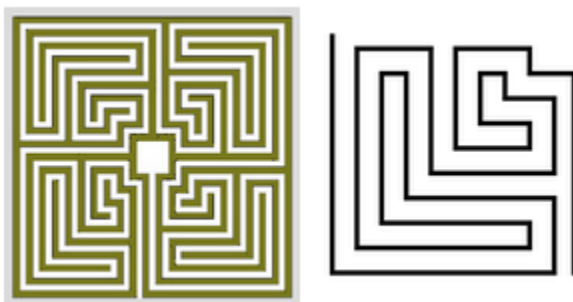
#### 4.1.1 Clásico



#### 4.1.2 Clásico Báltico



#### 4.1.3 Romano



#### 4.1.4 Medieval



#### 4.1.5 Contemporáneos



### 4.2 Laberinto multicursal

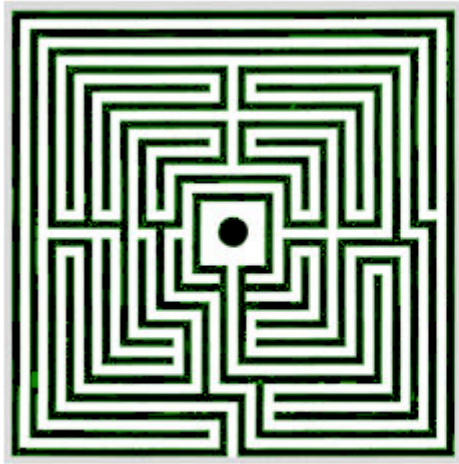
Es el trazado en el que existe la capacidad para elegir entre distintos caminos con la posibilidad de que las elecciones tomadas no nos lleven hacia el destino o incluso nos lleven a calles muertas (callejones sin salida). En algunos de los trazados multicurs.

#### 4.2.1 De conexión simple





#### 4.2.2 De conexión multiple



Trazado de Chevening House



Laberinto del Castillo de Leeds, Condado de Kent

## 5. DISEÑO

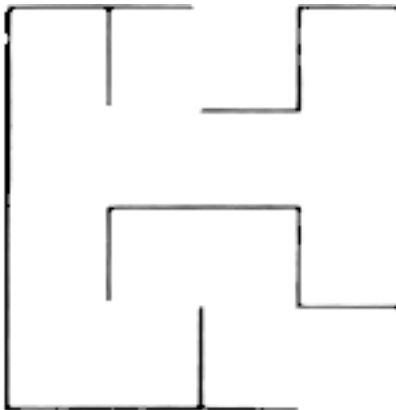
### 5.1 Tecnología

Todo el proyecto fue desarrollado con tecnologías Microsoft. Inicia con una aplicación de consola, luego con un formulario de visualización 2D y finalmente un exportador de código en 3D (VRML). Desde un comienzo se codificó orientado a objetos, esto permitió hacer compatible todo el proceso de desarrollo hasta la última etapa de tecnología VRML.

### 5.2 Autoconstrucción del laberinto

Un laberinto es parecido a la planta de un piso de una casa con una entrada y una salida. Para construir un laberinto, el primer paso es dibujar la casa con las paredes, dividiéndola en habitaciones, pero sin ninguna puerta entre habitaciones, ni en la entrada ni en la salida de la casa. A continuación se añaden puertas hasta tener un camino entre cualquiera de las dos habitaciones de la casa. Finalmente se incluye una entrada y una salida en el sitio que se desee.

Este es un laberinto de 4x4. Verifique usted mismo que sólo hay un camino entre cualquiera de dos habitaciones que escoja. Pruebe a cerrar la entrada y la salida y abra dos nuevas. Seguirá teniendo un laberinto completamente válido ya que su estructura interna siempre permanece igual.

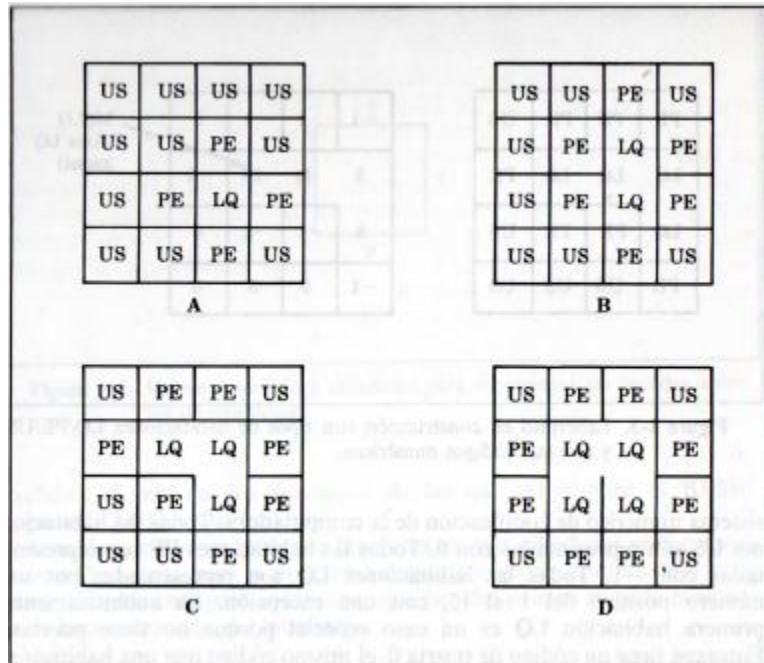


Para que se puedan generar laberintos se deben tener en cuenta los siguientes conceptos:

- Cuartos de estar (LQ): habitaciones que están comunicadas a través de puertas
- Zonas de expansión (PE): habitaciones adyacentes a los cuartos de estar, pero aún no tienen puertas.

- Espacios no utilizados (US): habitaciones que no son adyacentes a los cuartos de estar y no tienen puertas.

Las abreviaturas LQ, PE y US se utilizarán como nombres de variables incluidas en el programa, que mostraremos más adelante en este capítulo. He aquí los pasos para construir un laberinto.



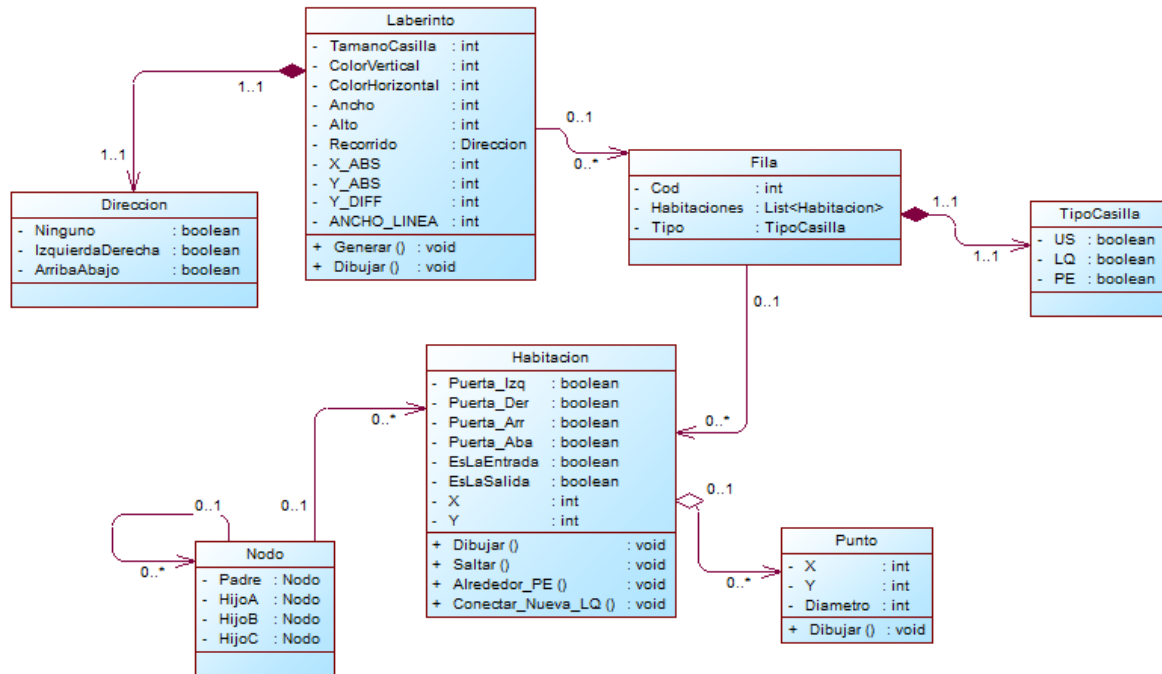
Un algoritmo completamente válido para esta tarea es el que se describe a continuación:

- Dividir el laberinto en habitaciones y marcarlas todas como espacios US.
- Seleccionar aleatoriamente una habitación que haremos de tipo LQ.
- Localizar todas las habitaciones US adyacentes a la LQ y hacerlas de tipo PE
- Si no quedan habitaciones PE, ir al paso 8; en caso contrario, continuar.
- Seleccionar aleatoriamente una habitación de la lista PE. Añadir una puerta de conexión a la LQ (si hay más de una habitación LQ adyacente a la PE, seleccionar aleatoriamente una).
- Marcar la nueva habitación como LQ; marcar todas las habitaciones que se han transformado en PE como resultado de esta adición.
- Volver al paso 3, utilizando la nueva habitación LQ como punto de partida.
- Seleccionar aleatoriamente una entrada en la parte superior y una salida en la parte inferior.

Puede verificar este método al crear sobre un papel y a mano un laberinto 4x4.

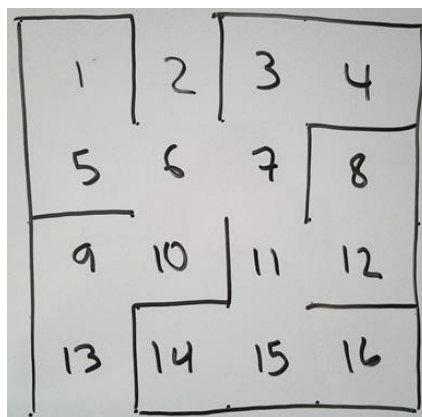
### 5.3 Diagrama de clases

El siguiente diagrama representa lo que se utilizará para dar solución al problema planteado en el proyecto:

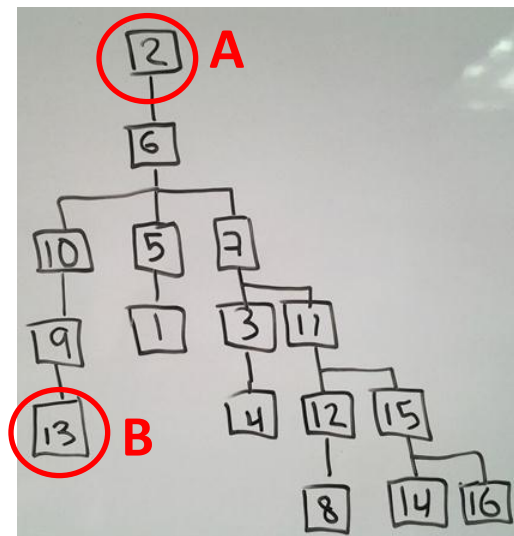


### 5.4 Algoritmo de resolución

Lo siguiente es entender cómo resolver un laberinto de estos generados. Para ello, se va a hacer una prueba de escritorio con un laberinto 4x4:



Se ve entonces que el laberinto también se puede escribir como un árbol:



## 6. CONSTRUCCIÓN

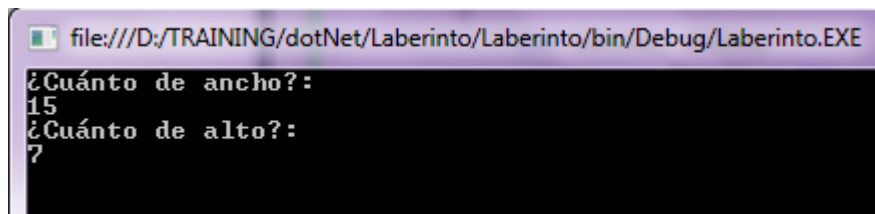
### 5.1 Algoritmo de autoconstrucción

El algoritmo de autoconstrucción se tomará como guía el texto entregado por el tutor. La implementación varia levemente pero es correcta según la explicación del libro. Por tanto, citando el procedimiento de construcción tenemos:

#### 1. Dividir el laberinto en habitaciones y marcarlas todas como espacios US.

- Para empezar, se van a permitir parámetros para construir el laberinto. De momento alto y ancho.

```
//----- i1. Leer tamaño de la matriz
Console.WriteLine("¿Cuánto de ancho?: ");
iAncho = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("¿Cuánto de alto?: ");
iAlto = Convert.ToInt32(Console.ReadLine());
if (iAlto < 4 || iAncho < 4) return; //Debe te
Console.Clear();
```



```
file:///D:/TRAINING/dotNet/Laberinto/Laberinto/bin/Debug/Laberinto.EXE
¿Cuánto de ancho?:
15
¿Cuánto de alto?:
7
```

- La matriz de ejemplo de 15x7 quedó dibujada en términos de US

```
//----- i2. Dibuja toda una matriz con espacios US
for (int j = 0; j < iAlto; j++)
{
    Fila f = new Fila(j + 1); //Crea una nueva fila de habitaciones
    Matriz.Add(f); //Almacena en la matriz cada fila

    for (int i = 0; i < iAncho; i++)
    {
        Habitacion h = new Habitacion(i, j, TipoCasilla.US);
        f.Habitaciones.Add(h); //Almacena la instancia
        h.Dibujar(true);
    }

    Habitacion.Saltar();
}
```

```
file:///D:/TRAINING/dotNet/Laberinto/Laberinto/bin/Debug/Laberinto.EXE
US US US US US US US US US US US US US US US
US US US US US US US US US US US US US US US
US US US US US US US US US US US US US US US
US US US US US US US US US US US US US US US
US US US US US US US US US US US US US US US
US US US US US US US US US US US US US US US
US US US US US US US US US US US US US US US
```

- Una fila es una lista de habitaciones. Es todo lo que tiene esa clase.

```
/// <summary>
/// Cada fila almacena las instancias de sus habitaciones
/// </summary>
public class Fila
{
    public Fila(int iCodigo) //Constructor
    {
        Cod = iCodigo;
        Habitaciones = new List<Habitacion>();
    }

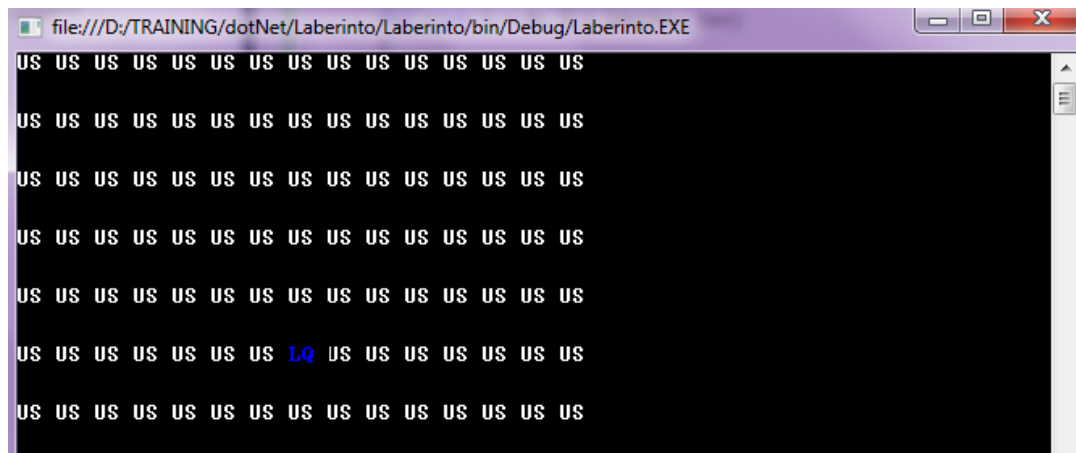
    public int Cod { get; private set; }

    public List<Habitacion> Habitaciones { get; set; }
}
```

## 2. Seleccionar aleatoriamente una habitación que haremos de tipo LQ.

```
//----- i3. Seleccion de una coordenada aleatoria
Random r = new Random(DateTime.Now.Millisecond);
rnd_fila = r.Next(1, iAlto) - 1;
rnd_col = r.Next(1, iAncho) - 1;

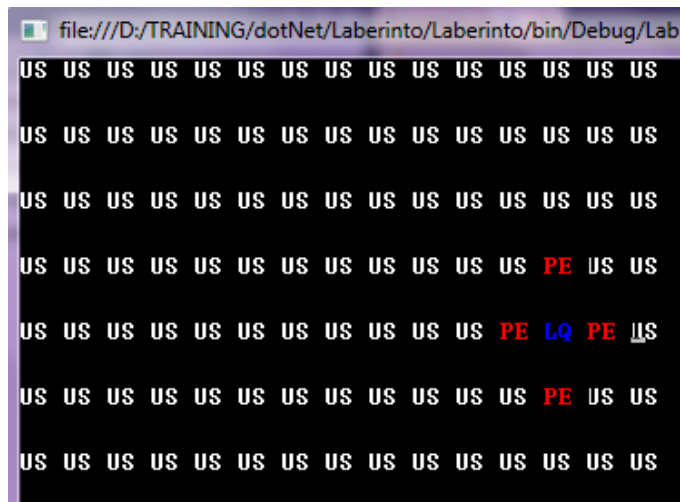
//----- i4. Convierte la celda aleatoria a tipo LQ
Habitacion h_nueva_lq = Matriz[rnd_fila].Habitaciones[rnd_col];
h_nueva_lq.Tipo = TipoCasilla.LQ;
h_nueva_lq.Dibujar(true);
```



### 3. Localizar las habitaciones US adyacentes a la LQ y hacerlas de tipo PE (añadirlas a la lista PE)

- Esta es una funcionalidad que tiene cada habitación. Es decir, cada habitación es capaz de convertir sus celdas adyacentes en PE.

```
//----- i5. Selecciona las habitaciones alrededor y las vuelve PE
h_nueva_lq.Alrededor_PE();
```





- Esto se hará sucesivamente en cada paso, por eso cada habitación es capaz de hacerlo: para reutilizar código. Al interior este código se ve como sigue

```
public void Alrededor_PE()
{
    //Arriba de la celda
    if (Y - 1 >= 0)
        if (Program.Matriz[Y - 1].Habitaciones[X].Tipo.Equals(TipoCasilla.US))
            Program.Matriz[Y - 1].Habitaciones[X].Dibujar(TipoCasilla.PE);
    //Abajo de la celda
    if (Y < Program.Matriz.Count - 1)
        if (Program.Matriz[Y + 1].Habitaciones[X].Tipo.Equals(TipoCasilla.US))
            Program.Matriz[Y + 1].Habitaciones[X].Dibujar(TipoCasilla.PE);
    //A la izquierda de la celda
    if (X - 1 >= 0)
        if (Program.Matriz[Y].Habitaciones[X - 1].Tipo.Equals(TipoCasilla.US))
            Program.Matriz[Y].Habitaciones[X - 1].Dibujar(TipoCasilla.PE);
    //A la derecha de la celda
    if (X < Program.Matriz[0].Habitaciones.Count - 1)
        if (Program.Matriz[Y].Habitaciones[X + 1].Tipo.Equals(TipoCasilla.US))
            Program.Matriz[Y].Habitaciones[X + 1].Dibujar(TipoCasilla.PE);
}
```

- Tomó algo de tiempo construir este método. La dificultad fueron las habitaciones más extremas de la matriz: arriba izquierda, abajo izquierda, arriba derecha, abajo derecha.
- Los if más externos son para validar estos extremos. La idea es evitar que se intente indexar los arreglos fuera del intervalo válido.
- El if interno significa la validación de que la habitación adyacente sea de tipo US. Si no se validara, entonces se sobrescribirían las celdas LQ y PE.

#### 4. Si no quedan habitaciones PE, ir al paso 8; en caso contrario continuar

- Esto requiere de una estructura de control. Elegimos hacerlo así:

```
while (b_quedan_pe) //Mientras hayan habitaciones PE en la matriz
{
    //Escoge una fila al azar
    rnd_fila = r.Next(iAlto);

    //Selecciona todos los PE en esa fila
    List<Habitacion> ls = Matriz[rnd_fila].Habitaciones.FindAll(t => t.Tipo.Equals(TipoCasilla.PE));

    //Si no encontró PE, continua el ciclo para buscar otra
    if (ls == null || ls.Count == 0) continue;

    Habitacion h_pe = ls[r.Next(ls.Count)]; //Escoge una de las PE al azar
    h_pe.Dibujar(TipoCasilla.LQ); //Vuelve la PE de tipo LQ
    h_pe.Alrededor_PE(); //Rodea con PE la nueva LQ. Solo aplica para los US
    h_pe.Conectar_Nueva_LQ(); //Abre las puertas de la habitacion

    for (int i = 0; i < Matriz.Count; i++) //Valida que no queden PE en la matriz
    {
        List<Habitacion> tmp = Matriz[i].Habitaciones.FindAll(h => h.Tipo.Equals(TipoCasilla.PE));
        if (tmp.Count > 0)
        { b_quedan_pe = true; break; }
        else
            b_quedan_pe = false;
    }
}
```

- El bloque seguirá iterando mientras existan habitaciones PE en la matriz. Para controlarlo se usa una bandera. Como se puede ver, hay un ciclo for interno. Se usa para determinar en cada iteración, si aún quedan PE. Se puede ver dentro del ciclo for un bloque if, cuando todas las iteraciones son falsas, entonces el ciclo while entiende que ya no quedan más PE y el programa finaliza.

### 5. Seleccionar aleatoriamente una habitación de la lista PE. Añadir una puerta de conexión a LQ (si hay más de una habitación LQ adyacente a la PE, seleccionar aleatoriamente una).

- Toda esta funcionalidad aleatoria también está contenida dentro de cada instancia de habitación. Es decir, una habitación es capaz de buscar una PE adyacente de forma aleatoria, convertirla a LQ y hacer la conexión de las puertas.

```
Habitacion h_pe = ls[r.Next(ls.Count)]; //Escoge una de las PE al azar
h_pe.Dibujar(TipoCasilla.LQ); //Vuelve la PE de tipo LQ
h_pe.Alrededor_PE(); //Rodea con PE la nueva LQ. Solo aplica para los US
h_pe.Conectar_Nueva_LQ(); //Abre las puertas de la habitacion
```

- Las puertas aunque no se ven en modo consola, realmente existen en memoria. El código se escribió orientado a objetos. La clase con estas propiedades es:

```
/// Esta clase representa a una Habitacion del laberinto
/// </summary>
public class Habitacion
{
    public Habitacion(int iX, int iY, TipoCasilla iTipo) //Constructor
    {
        X = iX;
        Y = iY;
        Tipo = iTipo;
        Puerta_Aba = Puerta_Arr = Puerta_Izq = Puerta_Der = false; //Todas inician cerradas
    }

    public bool Puerta_Izq { get; set; } //Tiene puerta a la izquierda
    public bool Puerta_Der { get; set; } //Tiene puerta a la derecha
    public bool Puerta_Arr { get; set; } //Tiene puerta arriba
    public bool Puerta_Aba { get; set; } //Tiene puerta abajo

    public int X { get; private set; } //Coordenada X
    public int Y { get; private set; } //Coordenada Y
    public TipoCasilla Tipo { get; set; } //Tipo de casilla
```

- En verde se pueden ver las propiedades de las puertas. Estas se modifican en cada instancia de acuerdo a lo explicado en el libro. Como está el algoritmo, ya está listo para migrarse a VRML.
- En rojo se pueden ver las coordenadas de una habitación dentro de la matriz así como el tipo de casilla que representa.
- El algoritmo que permite la aleatoriedad en esta búsqueda está dentro de cada habitación. Aquí:

```

/// Busca y selecciona aleatoriamente una LQ adyacente
/// </summary>
/// <returns></returns>
public void Conectar_Nueva_LQ()
{
    Habitacion h_rnd_lq = null;
    Random r = new Random(DateTime.Now.Millisecond);
    bool b_hay_lq = false;
    while (!b_hay_lq) //Mientras que no encuentre una LQ adyacente, sigue buscando
    {
        int rnd_lq_ady = r.Next(1, 5); //Aleatoriamente elije donde buscar
        switch (rnd_lq_ady) //Solo son 4 las posibles celdas adyacentes a una habitacion
        {
            case 1: //Está arriba de la nueva LQ
                if (Y - 1 >= 0)
                {
                    h_rnd_lq = Program.Matriz[Y - 1].Habitaciones[X];
                    if (!h_rnd_lq.Tipo.Equals(TipoCasilla.LQ)) continue;

                    h_rnd_lq.Puerta_Aba = true;
                    Puerta_Arr = true;

                    b_hay_lq = true; //Encontró LQ, entonces rompe el ciclo
                }
                break;

```

#### 6. Marcar la nueva habitación como LQ; marcar todas las habitaciones que se han transformado en PE como resultado de esta adición.

```

Habitacion h_pe = ls[r.Next(ls.Count)]; //Escoge una de las PE al azar
h_pe.Dibujar(TipoCasilla.LQ); //Vuelve la PE de tipo LQ
h_pe.Alrededor_PE(); //Rodea con PE la nueva LQ. Solo aplica para los US
h_pe.Conectar_Nueva_LQ(); //Abre las puertas de la habitacion

```

#### 7. Volver al paso 3, utilizando la nueva habitación LQ como punto de partida.

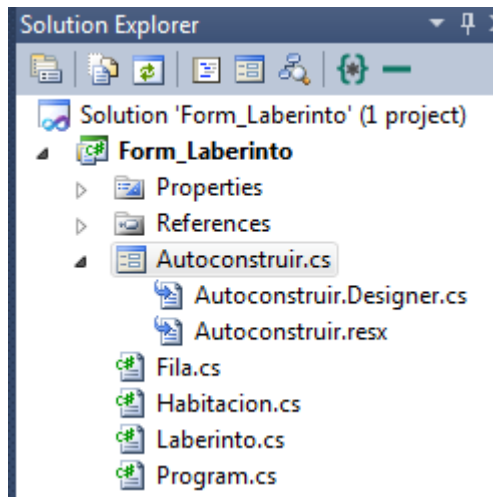
- Esta parte se implementó diferente a la propuesta en el libro. Es decir, en lugar de utilizar la nueva LQ como punto de partida. Busca una PE, las cuales siempre tienen una LQ adyacente y parte de allí. En la siguiente versión del documento se comprobará si este cambio afecta la autoconstrucción. En caso que no funcione, se modificará según dice aquí.

## 8. Seleccionar aleatoriamente una entrada en la parte superior y una salida en la parte inferior.

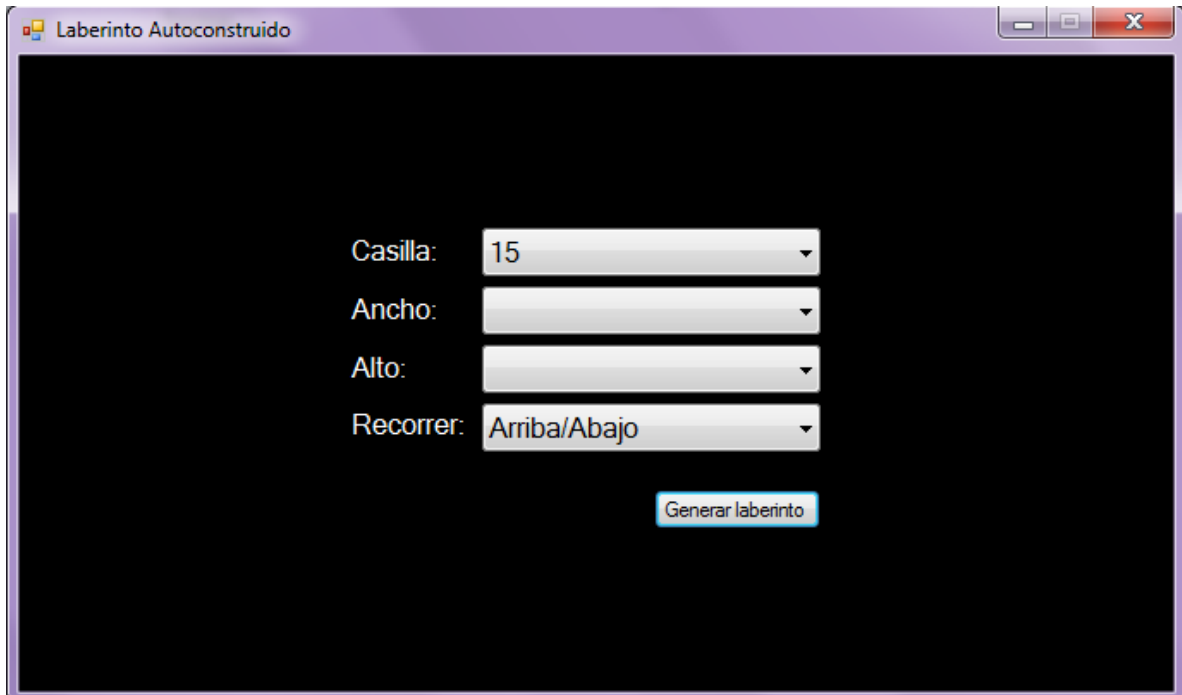
- Este paso está pendiente hasta tanto se hagan las pruebas de escritorio con el algoritmo construido.

## 5.2 Renderizado 2D

- En este punto, el algoritmo está construido y funciona. Lo siguiente es probar en una interfaz gráfica la autoconstrucción. Terminar de parametrizar algunos datos y construir parte de la inteligencia gráfica que deberá llevarse a VRML. Para esto se escogió Windows Forms.
- Las clases del proyecto de consola fueron integradas en un nuevo proyecto. Se ve de esta forma:



- Los datos de entrada son los que se muestran en la imagen.



- Casilla (Tamaño): Representa el tamaño de la casilla en la representación en pantalla. Una casilla pequeña hará que el laberinto sea mucho más complejo y viceversa.
  - # Ancho: número de casillas que tendrá de ancho el laberinto.
  - # Alto: número de casillas que tendrá de alto el laberinto.
  - Recorrer: Se pueden generar las entradas y salidas (solución) de dos formas.
- Durante la migración desde la consola, fue necesario comentar todas las líneas de escritura/lectura de consola, ya que no es un controlador válido de I/O en Windows Forms. Esto era de esperarse, y por tanto no afecta el algoritmo que trajimos. Aquí un ejemplo:

```

public void Dibujar(ConsoleColor iColor)
{
    //Console.ForegroundColor = iColor;
    //Dibujar(false);
}

public void Dibujar(bool iDefaultColor)
{
    //String iCadena;
    //switch (Tipo)
    //{
    //    case TipoCasilla.US: iCadena = "US";
    //        if (iDefaultColor) Console.ForegroundColor = ConsoleColor.White;
    //        break;
    //    case TipoCasilla.LQ: iCadena = "LQ";
    //        if (iDefaultColor) Console.ForegroundColor = ConsoleColor.Blue;
    //        break;
    //    case TipoCasilla.PE: iCadena = "PE";
    //        if (iDefaultColor) Console.ForegroundColor = ConsoleColor.Red;
    //        break;
    //    default: iCadena = "XX"; break;
    //}

    //Console.SetCursorPosition(X * 3, Y * 3); //El 3 es para ajustar lo que se imprime
    //Console.Write(iCadena.Substring(0, 2) + " ");
}

```

- Se deduce de la imagen anterior que es necesario implementar un nuevo y más completo método para dibujar las casillas en pantalla.
- Para ir ajustando el programa a lo que será en VRML, se fueron abstrayendo algunas constantes para representación gráfica.

```

#region Constantes

public const int X_ABS = 10;           //Coordenada superior izquierda donde se pinta el laberinto
public const int Y_ABS = 30;
public const int ANCHO_LINEA = 1;     //Ancho de la línea de cada casilla

#endregion Constantes

```

- Algunos otros datos se abstraieron en propiedades de la clase Laberinto, para facilitar su representación en pantalla.

```

public static int Ancho { get; set; }

public static int Alto { get; set; }

public int Y_DIFF { get { return Y_ABS + TamanoCasilla - ANCHO_LINEA; } } //"Y" con

public static int TamanoCasilla { get; set; } //Tamaño de una casilla o habitacion

public static Color COLOR_VERTICAL { get { return Color.LightGreen; } }

public static Color COLOR_HORIZONTAL { get { return Color.LightSkyBlue; } }

```

- En el programa de consola faltó generar aleatoriamente una entrada y una salida. Eso ya queda resuelto en este paso del informe.

```

//----- i7. Escoge una entrada-salida al azar
switch (iRecorrido)
{
    case Recorrer.IzquierdaDerecha:
        Matriz[r.Next(Alto)].Habitaciones[0].EsLaEntrada = true; //Marca la entrada
        Matriz[r.Next(Alto)].Habitaciones[Ancho - 1].EsLaSalida = true; //Marca la salida

        Matriz[r.Next(Alto)].Habitaciones[0].Puerta_Izq = true;
        Matriz[r.Next(Alto)].Habitaciones[Ancho - 1].Puerta_Der = true;
        break;

    case Recorrer.ArribaAbajo:
        Matriz[0].Habitaciones[r.Next(Ancho)].EsLaEntrada = true; //Marca la entrada
        Matriz[Alto - 1].Habitaciones[r.Next(Ancho)].EsLaSalida = true; //Marca la salida

        Matriz[0].Habitaciones[r.Next(Ancho)].Puerta_Arr = true;
        Matriz[Alto - 1].Habitaciones[r.Next(Ancho)].Puerta_Aba = true;
        break;
}

```

- Ahora existe un nuevo método para dibujar todo el laberinto en pantalla. Básicamente se recorre toda la matriz y le dice a las habitaciones que se dibujen a sí mismas.

```

/// <summary>
/// Dibuja el laberinto en pantalla
/// </summary>
public static void Dibujar(Graphics formGraphics)
{
    for (int i = 0; i < Matriz.Count; i++)
        for (int j = 0; j < Matriz[0].Habitaciones.Count; j++)
            Matriz[i].Habitaciones[j].Dibujar(formGraphics, COLOR_HORIZONTAL, COLOR_VERTICAL);

    formGraphics.Dispose();
}

```

- Mirando ya al interior del método para dibujar una habitación, se observan muchas coordenadas. Ocurre que son dinámicas. Cada habitación sabe exactamente en dónde debe dibujarse basada en sus coordenadas dentro de la matriz, del tamaño de la casilla, del ancho de la línea que la define y de la posición inicial del laberinto.

Aunque es matemática básica, no es fácil comprender a simple vista cómo se dibuja cada una. Por eso se dejó comentado el código.

```
public void Dibujar(Graphics formGraphics, Color iHorizontal, Color iVertical)
{
    //Coordenadas dinámicas rectangulos horizontales
    int iH0 = Laberinto.X_ABS + Laberinto.TamanoCasilla * X;
    int iH1 = Laberinto.Y_ABS + Laberinto.TamanoCasilla * Y;
    int iHD = iH1 + Laberinto.TamanoCasilla;

    //Coordenadas dinámicas rectangulos verticales
    int iV0 = Laberinto.X_ABS + Laberinto.TamanoCasilla * X;
    int iV1 = Laberinto.Y_ABS + Laberinto.TamanoCasilla * Y;
    int iVD = iV0 + Laberinto.TamanoCasilla;

    using (SolidBrush b2 = new SolidBrush(iHorizontal))
    {
        if (!Puerta_Arr) //Horizontal arriba
            formGraphics.FillRectangle(b2, new Rectangle(iH0, iH1, Laberinto.TamanoCasilla, Laberinto.ANCHO_LINEA));
        if (!Puerta_Aba) //Horizontal abajo
            formGraphics.FillRectangle(b2, new Rectangle(iH0, iHD, Laberinto.TamanoCasilla, Laberinto.ANCHO_LINEA));
    }

    using (SolidBrush b1 = new SolidBrush(iVertical))
    {
        if (!Puerta_Izq) //Vertical izquierda
            formGraphics.FillRectangle(b1, new Rectangle(iV0, iV1, Laberinto.ANCHO_LINEA, Laberinto.TamanoCasilla));
        if (!Puerta_Der) //Vertical derecha
            formGraphics.FillRectangle(b1, new Rectangle(iVD, iV1, Laberinto.ANCHO_LINEA, Laberinto.TamanoCasilla));
    }
}
```

- Debido a que el dibujo que se va a hacer es 2D, y además se está permitiendo al usuario escoger el tamaño de las habitaciones del laberinto, es necesario entonces calcular de alguna forma cuánto de ancho y de alto puede tener. Eso se hace aquí en un ciclo de números pares:

```
private void cbxTamanoCasilla_SelectedIndexChanged(object sender, EventArgs e)
{
    int iTamano = Convert.ToInt32(cbxTamanoCasilla.Text);

    cbxAlto.Items.Clear();
    for (int i = 4; i < 330 / iTamano; i += 2) //Valores de ancho (330px en total)
        cbxAlto.Items.Add(i.ToString());

    cbxAncho.Items.Clear();
    for (int i = 4; i < 660 / iTamano; i += 2) //Valores de ancho (660px en total)
        cbxAncho.Items.Add(i.ToString());
}
```



- Finalizando esta parte de dibujo en 2D, este es el método más externo a partir del cual se dibuja el laberinto autoconstruido.

```
private void btnGenerar_Click_1(object sender, EventArgs e)
{
    Validaciones

    //Asigna parámetros
    Recorrer r = Recorrer.Ninguno;
    if (cbxRecorrer.SelectedIndex == 0) r = Recorrer.ArribaAbajo;
    if (cbxRecorrer.SelectedIndex == 1) r = Recorrer.IzquierdaDerecha;
    Laberinto.Ancho = Convert.ToInt32(cbxAncho.Text);
    Laberinto.Alto = Convert.ToInt32(cbxAlto.Text);
    Laberinto.TamanoCasilla = Convert.ToInt32(cbxTamanoCasilla.Text);

    //Autoconstruye un laberinto
    Laberinto.Generar(r);
    //Lo dibuja en pantalla
    Laberinto.Dibujar(this.CreateGraphics());
}
```

### 5.3 ÁRBOL 4-ARIO PARA RESOLVER LABERINTOS

- Es importante entonces almacenar las coordenadas de entrada y salida del laberinto.

```
public static int[,] AB { get; set; }
```

- Las coordenadas AB se asignan cuando se generan aleatoriamente la entrada y salida del laberinto.

```
AB = new int[2, 2]; //Aquí almacenará las coordenadas entrada-salida

switch (Recorrido)
{
    case Direccion.IzquierdaDerecha:
        Matriz[a].Habitaciones[0].EsLaEntrada = true; //Marca la entrada
        Matriz[b].Habitaciones[Ancho - 1].EsLaSalida = true; //Marca la salida

        Matriz[a].Habitaciones[0].Puerta_Izq = true;
        Matriz[b].Habitaciones[Ancho - 1].Puerta_Der = true;

        AB[0, 0] = 0; AB[0, 1] = a; //A: Coordenada entrada
        AB[1, 0] = Ancho - 1; AB[1, 1] = b; //B: Coordenada salida
        break;

    case Direccion.ArribaAbajo:
        Matriz[0].Habitaciones[c].EsLaEntrada = true; //Marca la entrada
        Matriz[Alto - 1].Habitaciones[d].EsLaSalida = true; //Marca la salida

        Matriz[0].Habitaciones[c].Puerta_Arr = true;
        Matriz[Alto - 1].Habitaciones[d].Puerta_Aba = true;

        AB[0, 0] = c; AB[0, 1] = 0; //A: Coordenada entrada
        AB[1, 0] = d; AB[1, 1] = Alto - 1; //B: Coordenada salida
        break;
}
```

- Se adiciona a las habitaciones la capacidad de resaltar si son parte del camino. Esto se hace con un cuadrado de color brillante en el centro. Se observa una bandera que tiene cada casilla para determinar si es parte o no del camino solución. Solo en ese caso se resalta.

```
/// <summary>
/// Resalta el centro de una habitacion para reconocer la ruta que la resuelve
/// </summary>
public void Resaltar(Graphics formGraphics)
{
    if (EsSolucion)
    {
        using (SolidBrush b = new SolidBrush(Color.Yellow))
        {
            int l = Laberinto.TamanoCasilla;
            int X0 = X * l + (int)(l / 8) + Laberinto.X_ABS + 2 * Laberinto.ANCHO_LINEA;
            int Y0 = Y * l + (int)(l / 8) + Laberinto.Y_ABS + Laberinto.ANCHO_LINEA;
            formGraphics.FillRectangle(b, new Rectangle(X0, Y0, (int)(3 * l / 4), (int)(3 * l / 4)));
        }
    }
}
```

- Para resolverlo se extraen las coordenadas de entrada y salida que están en la matriz AB y luego invoca la creación del árbol, que de manera recursiva busca la salida.
- Después, el árbol ya habrá marcado cuáles son las casillas que conforman la solución, por lo que solo resta resaltarlas.

```

/// <summary>
/// Resuelve el laberinto y muestra el resultado
/// </summary>
/// <param name="formGraphics"></param>
public static void Resolver(Graphics formGraphics)
{
    //Crea y resuelve el árbol del laberinto
    Habitacion A = Matriz[AB[0, 1]].Habitaciones[AB[0, 0]];
    Habitacion B = Matriz[AB[1, 1]].Habitaciones[AB[1, 0]];
    Nodo arbol = Crear_Arbol(A, B);

    //Dibuja la solución en pantalla
    for (int i = 0; i < Matriz.Count; i++)
        for (int j = 0; j < Matriz[0].Habitaciones.Count; j++)
            Matriz[i].Habitaciones[j].Resaltar(formGraphics);
}

```

- El método de creación del árbol realmente es una invocación a un algoritmo recursivo. Se creó este método con una sola línea para hacer más fácil comprender que al final se trata de trazar una ruta que lleve de la habitación A a la B.

```

private static Nodo Crear_Arbol(Habitacion A, Habitacion B)
{
    return Recorrer(new Nodo(null, A), B);
}

```

- Este es el algoritmo recursivo para crear el árbol. La primera parte es la validación con la que al final se rompe la pila de llamados. Se valida si el nodo N ya encontró el destino B para romper.

```

private static Nodo Recorrer(Nodo N, Habitacion B)
{
    #region Validación para romper

    //Valida si encontró el destino
    if (N != null)
        if (N.Actual.X == B.X)
            if (N.Actual.Y == B.Y)
            {
                Trazar_Ruta(N);
                return null;
            }

    #endregion Validación para romper

    Busca en habitaciones contiguas

    return N; //Retorna el nodo y sus hijos
}

```

- Ya entrando en el método recursivo que representa el árbol del laberinto. Se pueden destacar 3 partes. Estas 3 partes se hacen en cada una de las 4 posibles direcciones relativas de una habitación (arriba, abajo, izquierda, derecha).

```
private static Nodo Recorrer(Nodo N, Habitacion B)
{
    Validación para romper

    #region Busca en habitaciones contiguas

    //Habitacion a la izquierda de N
    int CoordX = 0, CoordY = 0;

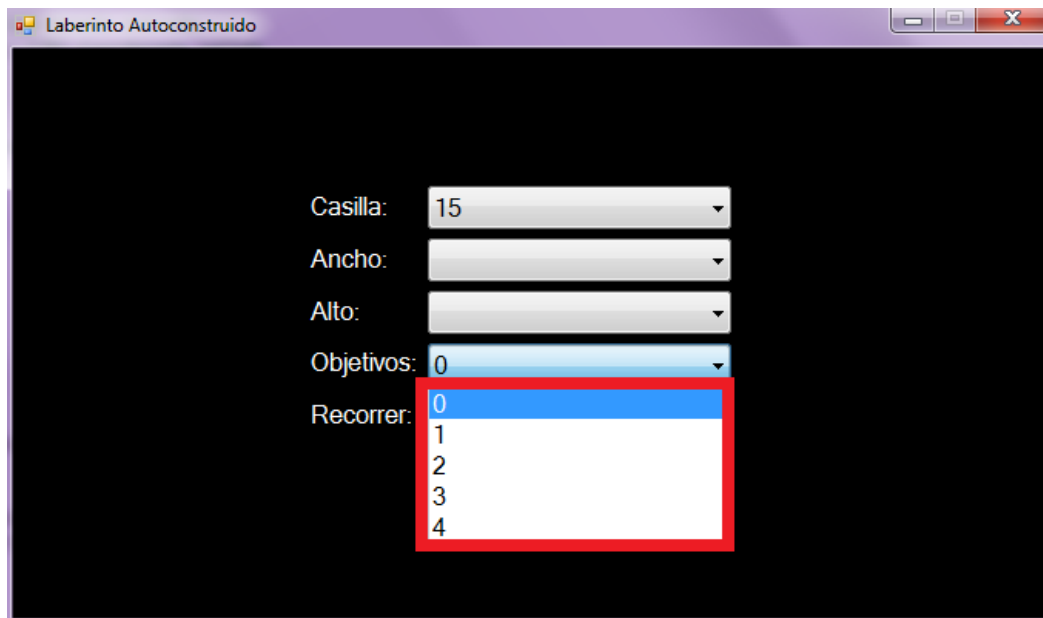
    if (N.Actual.Puerta_Izq && N.Actual.X > 0) 1
    {
        CoordX = N.Actual.X - 1;
        CoordY = N.Actual.Y;
        2 if (!Matriz[CoordY].Habitaciones[CoordX].Mapeada) 3
        {
            N.HijoIzq = Recorrer(new Nodo(N, Matriz[CoordY].Habitaciones[CoordX]), B);
        }
        //Habitacion a la derecha de N
        if (N.Actual.Puerta_Der && N.Actual.X != Matriz[0].Habitaciones.Count - 1)
        {
            CoordX = N.Actual.X + 1;
            CoordY = N.Actual.Y;
            if (!Matriz[CoordY].Habitaciones[CoordX].Mapeada)
            {
                N.HijoDer = Recorrer(new Nodo(N, Matriz[CoordY].Habitaciones[CoordX]), B);
            }
        }
    }
}
```

- La parte 1 valida que la puerta respectiva (arriba, abajo, izquierda, derecha) se encuentre abierta. La segunda validación de la condición es para asegurarse que el árbol no intente indexar la matriz fuera de sus límites.
- La parte 2 muestra que cuando se encuentra que una habitación N tiene una contigua (por ejemplo a la izquierda) porque la puerta está abierta, esta se asigna como un hijo del nodo actual y vuelve y se invoca el método de recorrido (por eso es recursivo), solo que esta vez el padre del nuevo recorrido es ese mismo N actual.
- La parte 3 es una validación muy importante, sin la cual el algoritmo fallaría y se replicaría hasta agotar la pila de ejecución. En palabras sencillas, cada habitación tiene una bandera que indica si ya fue recorrida por el árbol. En caso tal, no se tendría en cuenta la próxima vez.

## 5.4 Objetivos de búsqueda (Bonus)

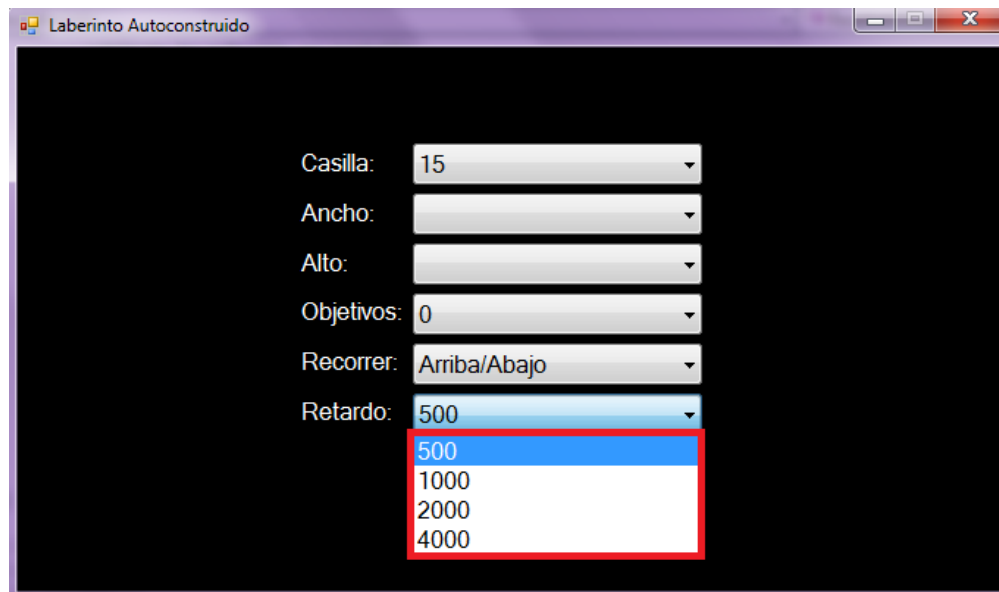
- Esta parte es corta porque reutiliza código. Se pretende adicionar algo más de aleatoriedad al sistema al poner objetivos a buscar antes de salir del laberinto. Un ejemplo sería que haya varias porciones de queso dentro del laberinto y que un ratón (avatar) vaya a recogerlas antes de salir.

- Esto sería relativamente sencillo dado el algoritmo solución que se desarrolló. Este algoritmo permite simplemente ir de una habitación A a una B. Por eso, lo único que se necesita es un nuevo parámetro de entrada desde la pantalla.



The screenshot shows the 'Laberinto Autoconstruido' application window. It contains several input fields: 'Casilla' (set to 15), 'Ancho', 'Alto', 'Objetivos' (set to 0), and 'Recorrer'. The 'Recorrer' dropdown menu is open, showing a list of options: 0, 1, 2, 3, and 4. The option '0' is currently selected and highlighted in blue. A red rectangular box is drawn around the entire dropdown menu area.

- Por tanto, lo siguiente con este parámetro es incluirlo en un ciclo que ubique y resuelva objetivos aleatorios. Se adicionaron colores (en una matriz) y un retardo de tiempo para permitir que se note el efecto.



The screenshot shows the 'Laberinto Autoconstruido' application window with updated settings. The 'Recorrer' dropdown is now set to 'Arriba/Abajo'. The 'Retardo' dropdown menu is open, showing a list of options: 500, 1000, 2000, and 4000. The option '500' is currently selected and highlighted in blue. A red rectangular box is drawn around the entire dropdown menu area.

- Este es el ciclo que controla los caminos y objetivos. Hay 4 partes principales que entender:

```

public static void Resolver(Graphics formGraphics)
{
    //Dibuja tantos objetivos aleatorios como se hayan configurado
    Habitacion A = Matriz[AB[0, 1]].Habitaciones[AB[0, 0]]; //Inicia en la entrada 2
    Habitacion B = null;

    Random r = new Random(DateTime.Now.Millisecond);
    for (int p = Laberinto.Objetivos + 1; p > 0; p--) 1
    {
        if (p == 1)
            B = Matriz[AB[1, 1]].Habitaciones[AB[1, 0]]; //Finaliza con la salida 4
        else
            B = Matriz[r.Next(Alto)].Habitaciones[r.Next(Ancho)]; //Crea un objetivo aleatorio 3

        //Crea y resuelve el árbol del laberinto
        Nodo arbol = Crear_Arbol(A, B);

        //Dibuja la solución en pantalla
        for (int i = 0; i < Matriz.Count; i++)
            for (int j = 0; j < Matriz[0].Habitaciones.Count; j++)
                Matriz[i].Habitaciones[j].Resaltar(formGraphics, ColoresCamino[p - 1]);

        //Restaura las habitaciones recorridas resolviendo el laberinto
        for (int i = 0; i < Matriz.Count; i++)
            for (int j = 0; j < Matriz[0].Habitaciones.Count; j++)
            {
                Matriz[i].Habitaciones[j].Mapeada = false;
                Matriz[i].Habitaciones[j].EsSolucion = false;
            }

        //Empalma y continua
        A = B;
        Thread.Sleep(Retardo_ms);
    }
}

```

- Recordemos que el algoritmo basado en árbol 4-ario crea un camino entre la habitación A y B.
- La parte 1 muestra el ciclo que controla los caminos y objetivos. Se controla con una variable que almacena el número de objetivos que el usuario escogió simular. Van disminuyendo y cuando ya no quedan, ahí si busca la salida del laberinto.
- La parte 2 es importante entenderla: sin importar cuántos objetivos se fijen en la simulación, la entrada siempre será el primer punto A a tenerse en cuenta.
- La parte 3 son todas las habitaciones aleatorias e intermedias que serán simuladas. Esto se puede ver más fácil con las imágenes al final de esta sección.
- La parte 4 es cuando ya no quedan más objetivos por buscar y se llega a la salida.
- Por último, los colores de los caminos en la simulación se controlan en una matriz.

```

//----- i8. Carga la matriz de colores para los caminos
ColoresCamino = new Color[5];
ColoresCamino[0] = Color.Yellow;
ColoresCamino[1] = Color.Salmon;
ColoresCamino[2] = Color.Pink;
ColoresCamino[3] = Color.White;
ColoresCamino[4] = Color.Blue;

```

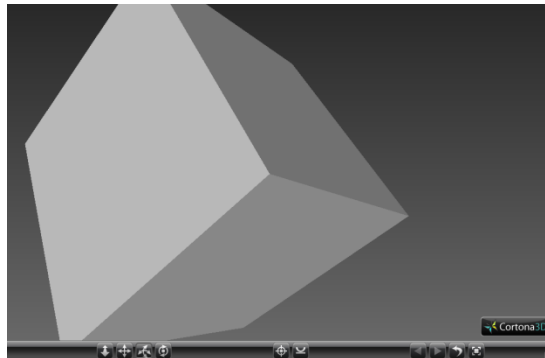
## 5.5 Inicios de VRML

- Antes de generar complejidad en VRML, empecemos por lo básico. La suma de lo básico hará que sea posible que complejos laberintos se muestren.
- Lo siguiente es un cubo en VRML, con base en el cual se van a generar las paredes

```
1 #VRML V2.0 utf8
2 Group
3 {
4     children [
5         DEF box Transform {
6             translation 0 0 0
7             children [
8                 Shape {
9                     appearance Appearance {
10                        material Material { diffuseColor 1 1 1 }
11                    }
12                    geometry Box { }
13                }
14            ]
15        }
16    ]
17 }
```

# open a group to include elements  
# include a first element, a group of 'children'  
# define what kind of object  
# define the coordinates of it : x y z  
# include children  
# define the properties of this object  
# Define appearance  
# Colour\*  
# Close the appearance def.  
# specify the type of object (box)  
# Closing the properties of the included object  
# Closing the second group of children  
# Closing the box properties  
# Closing the first group of children  
# Closing the group

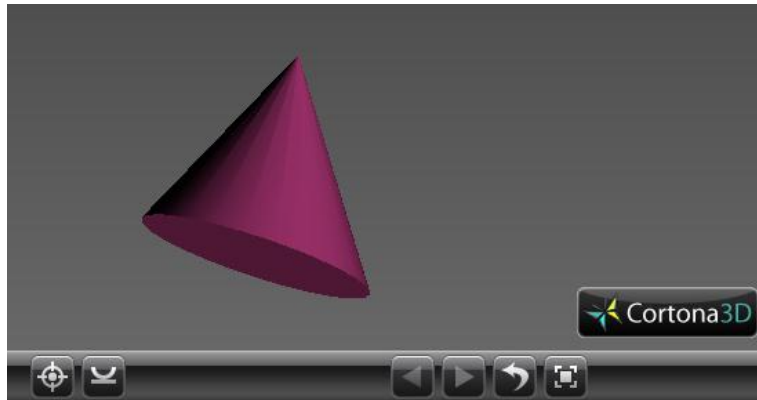
Fuente: [http://www.cse.iitm.ac.in/~vplab/tutorial\\_vrml.pdf](http://www.cse.iitm.ac.in/~vplab/tutorial_vrml.pdf)



- El siguiente es un avatar básico para empezar

```
1 #VRML V2.0 utf8
2 Group
3 {
4     children [
5         DEF box Transform {
6             translation 0 1.5 0
7             children [
8                 Shape {
9                     appearance Appearance {
10                        material Material { diffuseColor 0.87 0.27 0.59 }
11                    }
12                    geometry Cone { }
13                }
14            ]
15        }
16    ]
17 }
```

# open a group to include elements  
# include a first element, a group of 'children'  
# The cone will be placed as follow : x:0 y:1.5 z:0  
# Define the type of object : Cone  
# Closing the first group of children  
# Closing the group



## 5.6 Generación de código con T4

- En adelante, se debe hacer generación de código con T4. Para ello, se crea una plantilla que represente un muro y que pueda ser reutilizada en una principal.
- Dentro de una plantilla T4 es posible iterar de forma fácil para ir generando lo que requiera el laberinto de acuerdo a la matriz que ya se generó en los pasos anteriores.
- Para empezar, vamos a ver una generación para un muro:

```

<#@ template language="C#" hostspecific="True" debug="True" #>
<#@ output extension="wrl" encoding="utf-8" #>
<#@ include file="T4Toolbox.tt" #>
<#@ include file="Habitacion3D.tt" #>
<#
    Habitacion3D template = new Habitacion3D();
    string hab = template.TransformText();
#>
#VRML V2.0 utf8
Background
{
    skyColor
    [
        0.0 0.0 0.0,
        0.02 0.04 0.48,
        0.0 0.0 0.0
    ]
    skyAngle
    [
        1.2,
        1.4
    ]
}
Group
{
    children [
<#= hab1 #>
    ]
}

```

**Encabezado constante del archivo**

**Salida variable desde la plantilla de muro**



- Al ver dentro de la plantilla del muro de habitación, se puede ver cómo se empieza a dividir en archivos la generación de código con T4.

```

<#+
// <copyright file="Habitacion3D.tt" company="">
// Copyright © . All Rights Reserved.
// </copyright>

public class Habitacion3D : Template
{
    public override string TransformText()
    {
        #>
        DEF box Transform {
            translation 0 0 0
            children [
                Shape {
                    appearance Appearance {
                        material Material { diffuseColor 1 1 1 }
                    }
                    geometry Box { size 1,3,1 }
                }
            ]
        }
        #>
        return this.GenerationEnvironment.ToString();
    }
}
#>

```

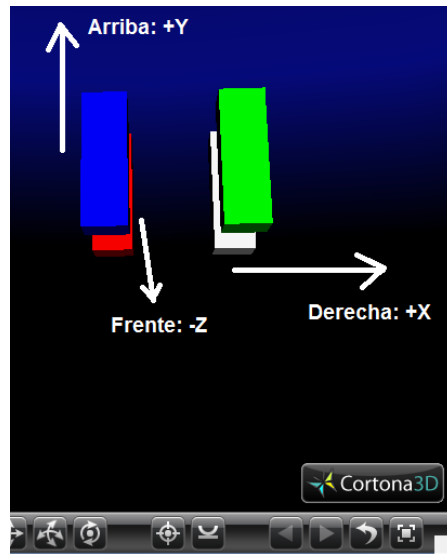
- Pero primero, hay que conocer el sistema de coordenadas a utilizar. Para eso se dibujaron algunas figuras de ejemplo. La orientación de los ViewPoint es la por defecto:

```

StringBuilder c = new StringBuilder();
c.Append(new Habitacion3D().ToVRML(0,0,0,1,0,0)); //Rojo
c.Append(new Habitacion3D().ToVRML(3,0,0,1,1,1)); //Blanco
c.Append(new Habitacion3D().ToVRML(0,0,3,0,0,1)); //Azul
c.Append(new Habitacion3D().ToVRML(3,0,3,0,1,0)); //Verde
#>
#VRML V2.0 utf8
Background {

```

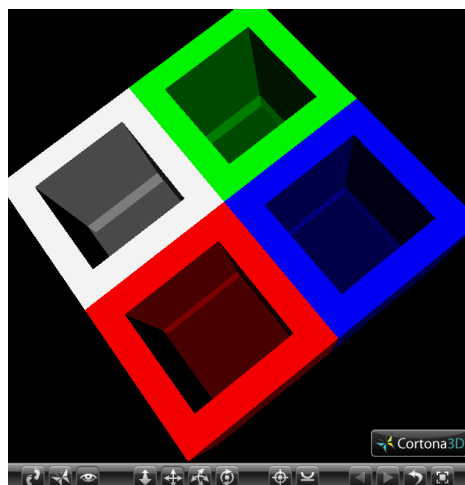
x,y,z,r,g,b



- Se crea un ViewPoint temporal que permita ver las habitaciones generadas. Para eso lo ubicamos justo detrás de la coordenada (0,0,0) que es el objeto rojo. El laberinto ya será mucho más claro de entender cuando se genere:



- Ya es momento de construir cada una de las paredes de una habitación típica. Se Definió un Shape como base y luego se siguió usando, haciendo traslaciones y rotaciones para no repetir tanto código. El piso de la habitación se hizo con algo de transparencia Al final, las 4 se ven de esta forma:



- En código, la generación de estas 4 habitaciones se ve como sigue:

```
<#@ import namespace="System.Text" #>
<#
    StringBuilder c = new StringBuilder();
    c.Append(new Habitación3D().ToVRML(0,0,0,1,0,0)); //Rojo
    c.Append(new Habitación3D().ToVRML(3,0,0,1,1,1)); //Blanco
    c.Append(new Habitación3D().ToVRML(0,0,3,0,0,1)); //Azul
    c.Append(new Habitación3D().ToVRML(3,0,3,0,1,0)); //Verde
#>
#VRML V2.0 utf8
Background {
    skyColor [
        0.0 0.0 0.0,
        0.02 0.04 0.48,
        0.0 0.0 0.0
    ]
    skyAngle [1.2,1.4]
}

Viewpoint {
    fieldOfView 0.785398
    position -15 8 -15
    orientation 0 1 0 -2.3
    description ""
    jump TRUE
}

Group {
    children [
        Shape {geometry DEF Pared Base { size 3,3,0.5 }}
    ]
}

<#= c.ToString() #>
]
```

**Se generan las 4 habitaciones**

**Pared base**

**Habitaciones generadas con T4**

- Lo que sigue entonces será crear un método que recorra la matriz de un laberinto autoconstruido. Para cada habitación se crean las formas en VRML.

```
/// <summary>
/// Genera el código VRML para el laberinto autoconstruido
/// </summary>
/// <param name="iMatriz"></param>
/// <returns></returns>
public static String Crear(List<Fila> iMatriz)
{
    sb.Clear();
    for (int i = 0; i < iMatriz.Count; i++)
        for (int j = 0; j < iMatriz[0].Habitaciones.Count; j++)
            sb.Append(new Habitación3D().ToVRML(iMatriz[i].Habitaciones[j]));

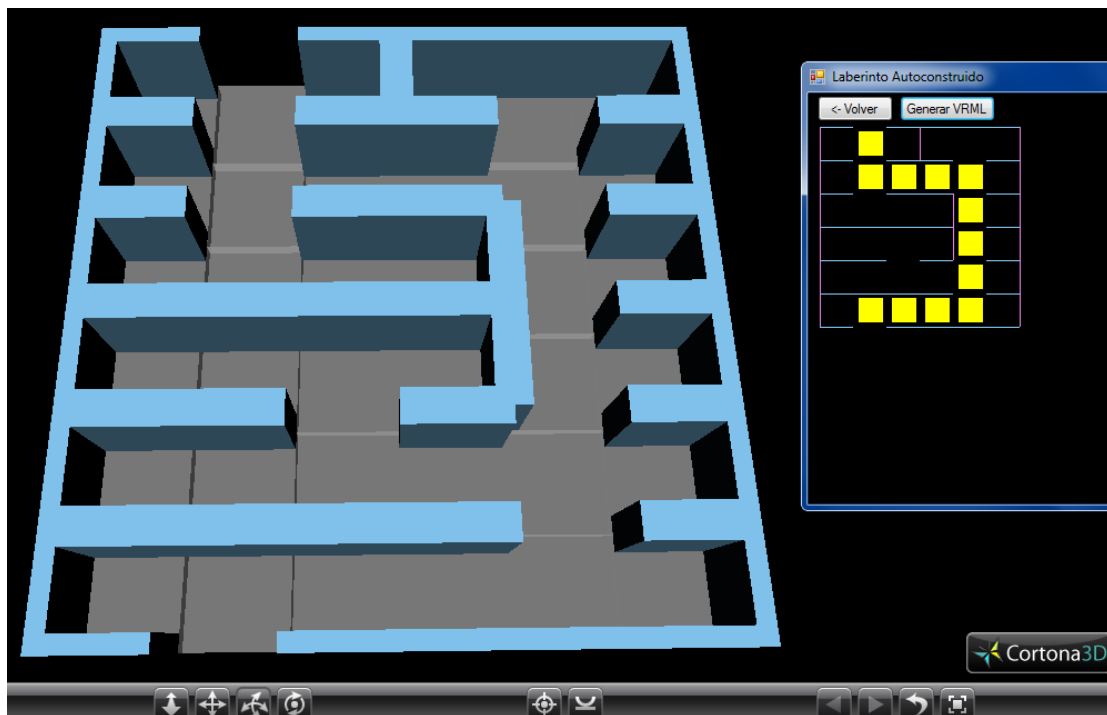
    return new VRML().TransformText();
}
```

- Se ajusta la plantilla de generación de código. Como se ve, ya aquí se valida cuando las diferentes puertas están abiertas y se deja un comentario para poder comprender el código generado.

```
<#@ template language="C#" #>
<#@ assembly name="System.Core" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.Collections.Generic" #>

#----- Habitación (x=<# _x #>, z=<# _z #>)
<# if(!Puerta_Izq){ #>
Transform {
  translation <# _x-1.25 #>,<# _y #>,<# _z+1.25 #>
  rotation 0 1 0 1.57079
  children Shape {
    appearance Appearance {
      material Material { diffuseColor <# _rh #>,<# _gh #>,<# _bh #> } }
    geometry USE ParedP #Puerta Izq +Z <X
  }
}
<# }
if(!Puerta_Der){ #>
Transform {
  translation <# _x+1.25 #>,<# _y #>,<# _z+1.25 #>
  rotation 0 1 0 1.57079
  children Shape {
    appearance Appearance {
      material Material { diffuseColor <# _rh #>,<# _gh #>,<# _bh #> } }
    geometry USE ParedP #Puerta Derecha +Z >X
  }
}
<# }
```

- Un ejemplo de laberinto 6x6 ya en 3D sería:



- Para hacer más liviano el archivo generado, se eliminan paredes redundantes. En lugar de tener dos paredes juntas, se elimina una de ellas. En la imagen anterior se observa que hay partes más anchas que otras, la idea es eliminar eso.

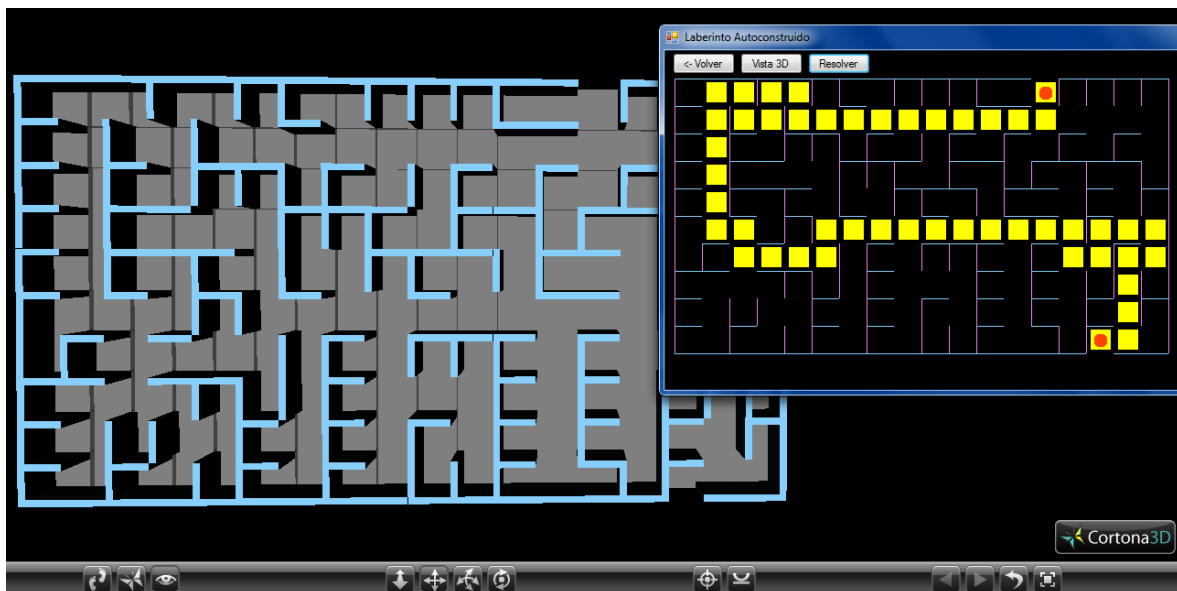
```

/// <summary>
/// Cuando hay dos paredes juntas se deja una sola.
/// Este método corrige las paredes en toda la matriz
/// </summary>
public static void QuitarParedesRedundantes()
{
    for (int i = 0; i < Matriz.Count; i++)
    {
        for (int j = 0; j < Matriz[i].Habitaciones.Count; j++)
        {
            //Puertas redundantes verticales
            if (j + 1 <= Matriz[i].Habitaciones.Count - 1)
                if (!Matriz[i].Habitaciones[j].Puerta_Der)
                    if (!Matriz[i].Habitaciones[j + 1].Puerta_Izq)
                        Matriz[i].Habitaciones[j].Puerta_Der = true;

            //Puertas redundantes horizontales
            if (i + 1 <= Matriz.Count - 1)
                if (!Matriz[i].Habitaciones[j].Puerta_Aba)
                    if (!Matriz[i + 1].Habitaciones[j].Puerta_Arr)
                        Matriz[i].Habitaciones[j].Puerta_Aba = true;
        }
    }
}

```

- Habiendo quitado las redundancias, se prueba con otro laberinto 3D más complejo de 10x18:



- La ruta que seguirá el avatar se construye a partir del camino solución del laberinto (amarillo en la anterior imagen), justo al final de romper el árbol solución:

```

/// <summary> ...
private static void Trazar_Ruta(Nodo iArbol)
{
    //Marca las habitaciones que son solucion
    Nodo N = iArbol;
    while (N != null)
    {
        //Para el sistema de coordenadas 3D, el X,Y se vuelve X,Z
        Laberinto3D.Camino.Add(new Solucion(N.Actual.X, N.Actual.Y));

        N.Actual.EsSolucion = true;
        N = N.Padre;
    }
}

```

- Como avatar se usará ahora una esfera. Se incluye en una plantilla aparte de T4.

Avatar3D.tt X

GeneratedTextTransformation

```

<#@ template language="C#" #>
<#@ assembly name="System.Core" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.Collections.Generic" #>

DEF Avatar Transform {
    children Shape{
        appearance Appearance{
            material Material {}
        }
        geometry Sphere{
            radius 1
        }
    }
}

```

- El siguiente propósito es adicionar recorrido a ese sencillo avatar. Para eso, la plantilla principal de T4 se modifica de esta forma:

```
#VRML V2.0 utf8
Background {
  skyColor [
    0.0 0.0 0.0,
    0.02 0.04 0.48,
    0.0 0.0 0.0
  ]
  skyAngle [1.2,1.4]
}

Viewpoint {
  fieldOfView 0.785398
  position -15 8 -15
  orientation 0 1 0 -2.3
  description ""
  jump TRUE
}

Group {
  children [

    <#= sbAvatar.ToString() #> — Avatar

    Shape {
      geometry DEF Pared Box { size 3,3,0.5 }
    }

    <#= sbHabitaciones.ToString() #> — Entramado de
    habitaciones

    <#= sbRecorrido.ToString() #> — Interpolación de movimiento (Recorrido)
  ]
}
```

- Para el recorrido, se vuelca el camino solución que ya se tenía guardado:

```
<#@ template language="C#" #>
<#@ assembly name="System.Core" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.Collections.Generic" #>
DEF posicioninicial PositionInterpolator
{
  key
  [
    <#= InterpolationKey #>
  ]
  keyValue
  [
    <#foreach (Solucion item in Camino)
    {#>
      <#= item.X * 3 #> 0 <#= (item.Z * 3)+1 #>,
    <#}#>
  ]
}

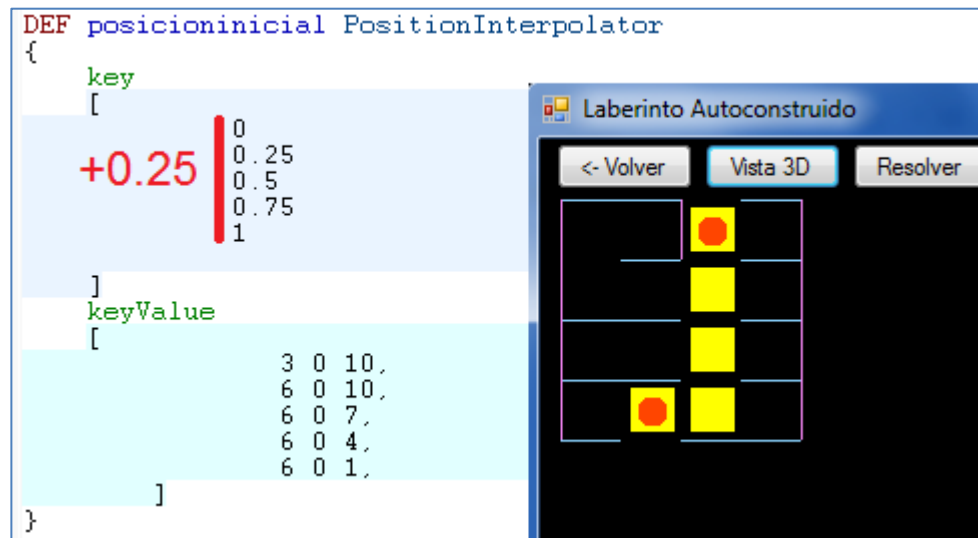
DEF temporizador TimeSensor
{
  cycleInterval <#= 5 #>
  loop TRUE
}

ROUTE temporizador.fraction_changed TO posicioninicial.set_fraction
ROUTE posicioninicial.value_changed TO Avatar.set_translation
```

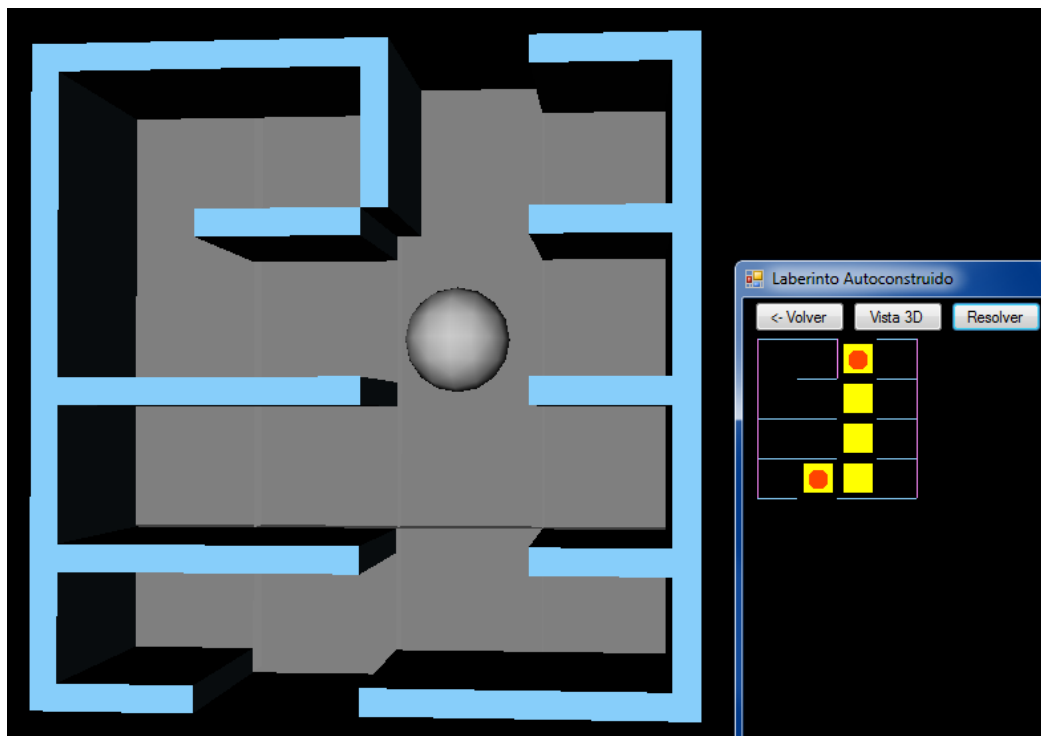
Se crea dinámicamente con intervalos constantes

El camino ya contiene la ruta a seguir, simplemente se genera el código aquí...

- En cuanto a la llave del interpolador, se dejaron intervalos constantes. Es decir, dependiendo de la longitud del camino solución, se hace que cada llave aumente incrementalmente desde 0 a 1. Por ejemplo, la siguiente imagen tiene un camino solución de longitud 5 en intervalos de +0.25:

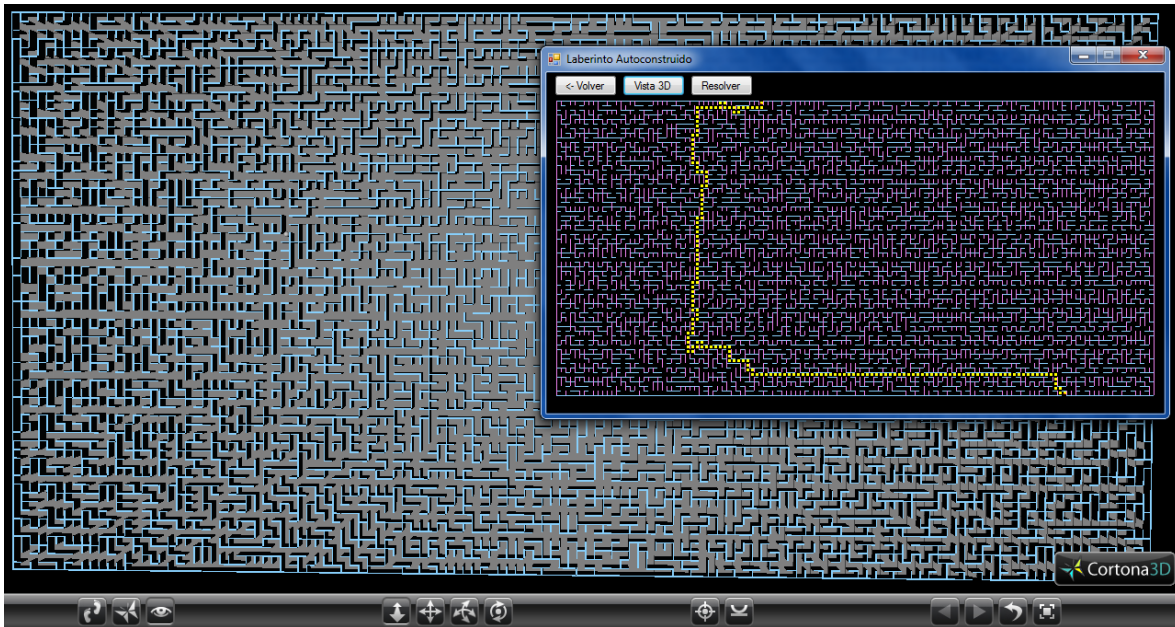


- En el KeyValue están cada una de las coordenadas solución para el laberinto de la imagen anterior.
- En resumen, el ejemplo sencillo de generación de código VRML para un avatar y recorrido es:

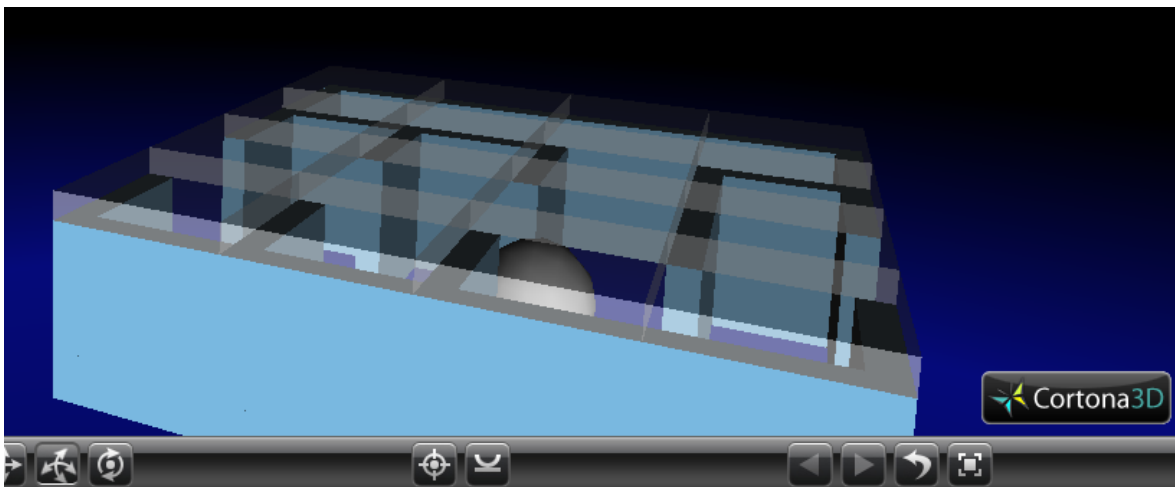




- Desde luego, ya en este punto el generador VRML funciona aún para grandes laberintos. El siguiente es de 130x64:



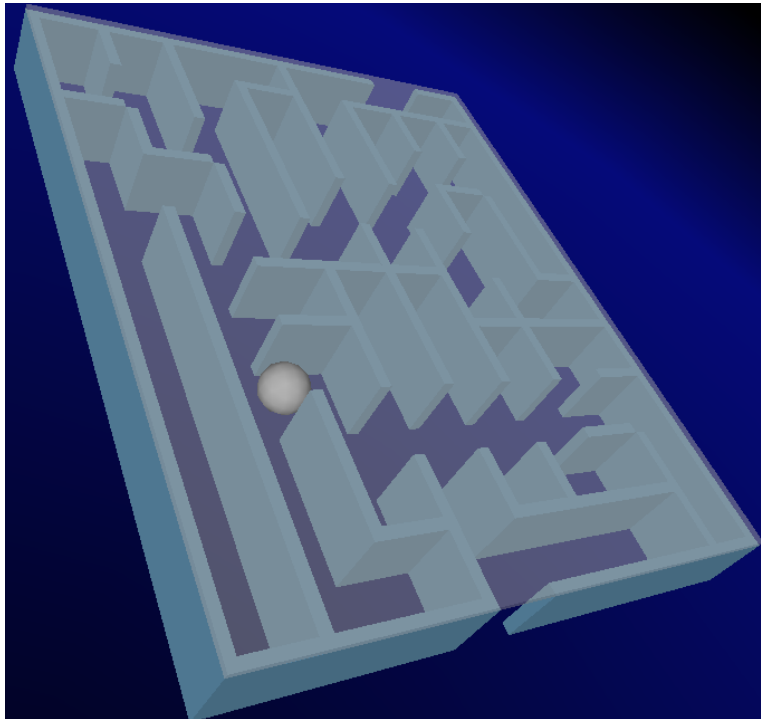
- Para reducir aún más el tamaño del archivo VRML, es necesario que la base del laberinto sea una sola figura y no una por habitación como ocurre ahora:



- Se quitó entonces el piso de las habitaciones y se hizo una sola base independiente

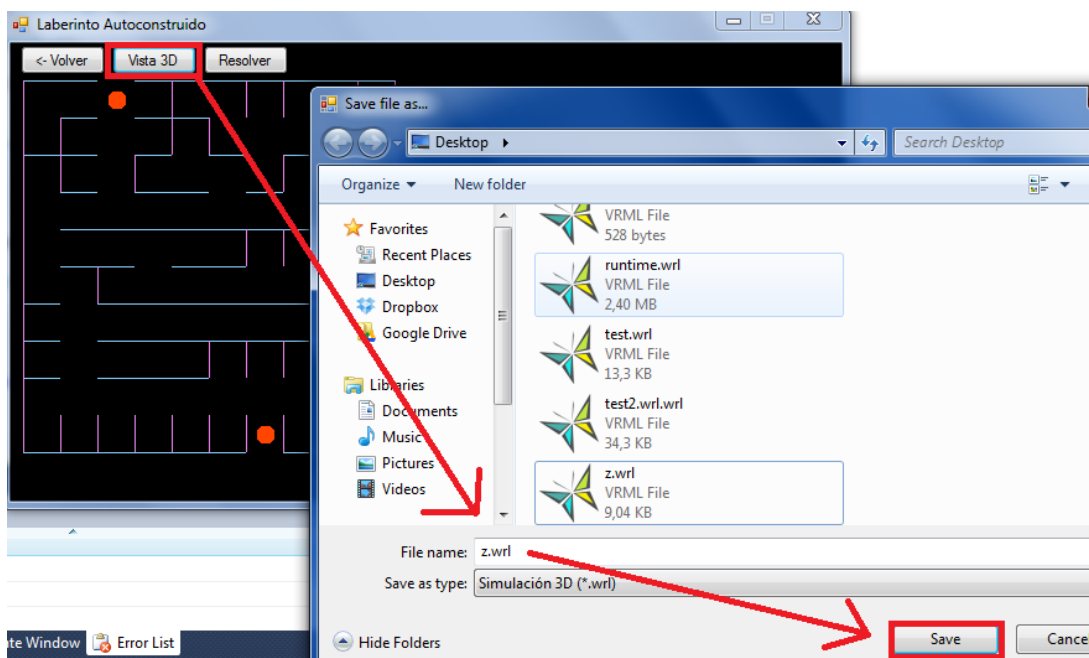
```
<#> sbAvatar.ToString() <#>
  Transform {
    translation <#> (1.5 * Laberinto.Ancho)-1.5 <#>,-1.625,<#> (1.5 * Laberinto.Alto) - 0.25 <#>
    children Shape {
      appearance Appearance { material Material { diffuseColor 1,1,1 transparency 0.5 } }
      geometry DEF Piso Box { size <#> Laberinto.Ancho * 3 <#>,0.25,<#> Laberinto.Alto * 3 <#> }
    }
  }
```

- Aquí se ve el piso del laberinto en una sola pieza.



## 5.7 Guardar en disco el mundo virtual

- Se adicionó la funcionalidad que permite guardar los laberintos 3D generados

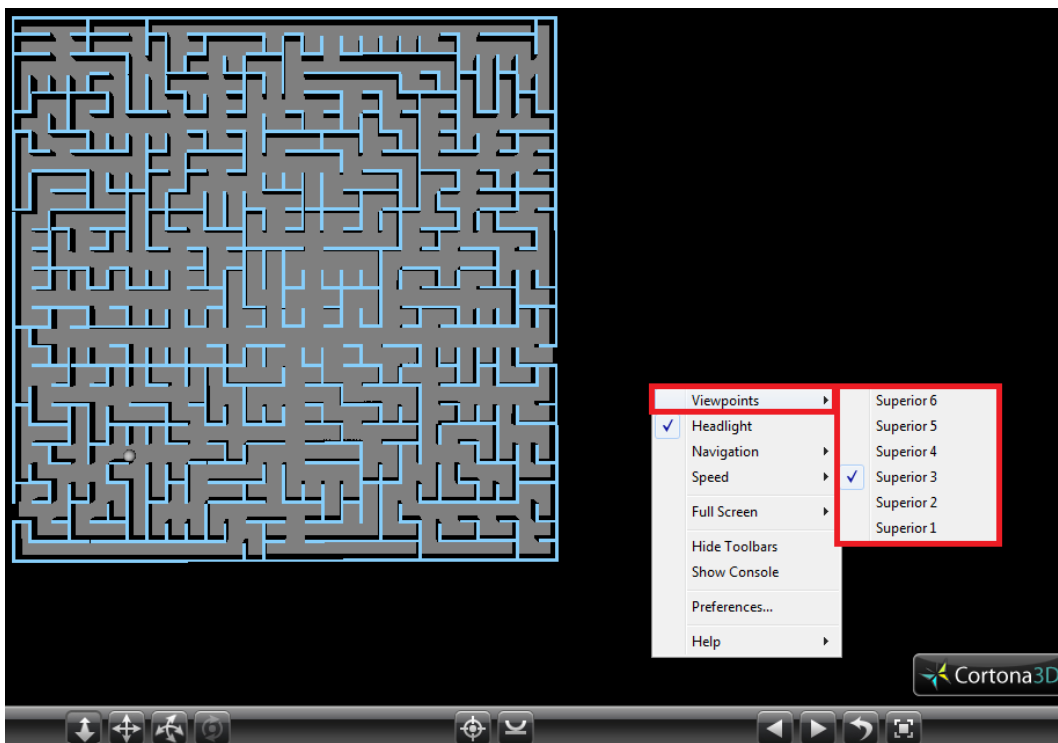


## 5.8 Viewpoints

- Hasta ahora se había utilizado una vista temporal para hacer el desarrollo. El siguiente paso es crear una vista superior del laberinto generado. Para ello se modifica la plantilla T4:

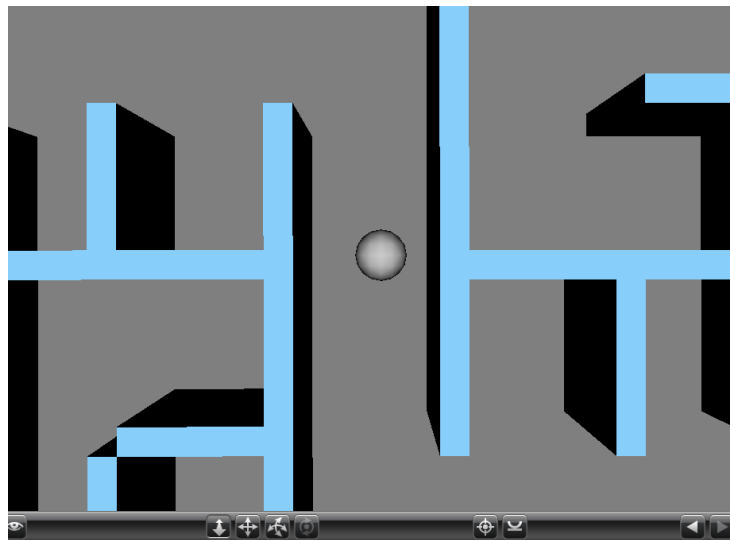
```
<# for (int i = 8; i > 2; i--)  
{#>  
    Viewpoint {  
        position <#= Laberinto.Ancho * 1.25 #> <#= i * Laberinto.Alto #> <#= Laberinto.Alto * 1.25 #>  
        orientation 0 1 1 3.14  
        description "<#= "Superior " + (i-2).ToString() #>"  
        jump TRUE  
    }  
<#>  
#>
```

- Lo que se hizo fue generar varias vistas superiores, unas más cerca del tablero que otras. El usuario puede escoger la que mejor le permita ver la animación. Todas son estáticas:



- Otra vista que se agrega para dar una vista más dinámica es una justo encima del avatar, con eso se puede ver cuando va desde la entrada a la salida del laberinto.

```
DEF Avatar Transform {
  children
  [
    Shape{
      appearance Appearance{
        material Material {}
      }
      geometry Sphere{
        radius 0.5
      }
    }
  ]
  DEF BALLVIEW Viewpoint {
    description "Avatar"
    position 0 12 0
    orientation 0 1 1 3.14
  }
}
```



- Se ajusta mucho mejor el cielo y la tierra del mundo virtual:

```
Background {
  groundAngle [1.57]
  skyAngle [1.57]
  groundColor [0 0 .3, 0 0 0]
  skyColor [0 0 0, .0 0 0]
}
```

## 5.9 Interacción con el usuario

- La interacción con el usuario será básica. Se trata de no iniciar el recorrido sino hasta que la persona haga clic en el avatar. En primer lugar se agrega un sensor a la esfera, se desactiva el temporizador y se coloca la esfera en la posición inicial.

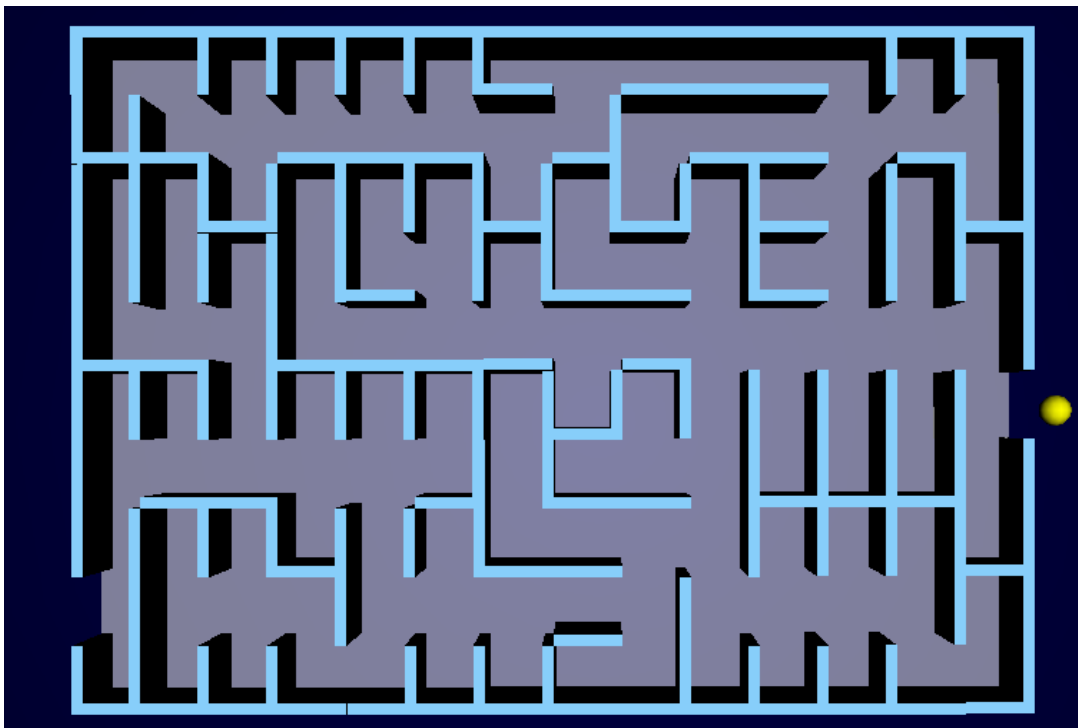
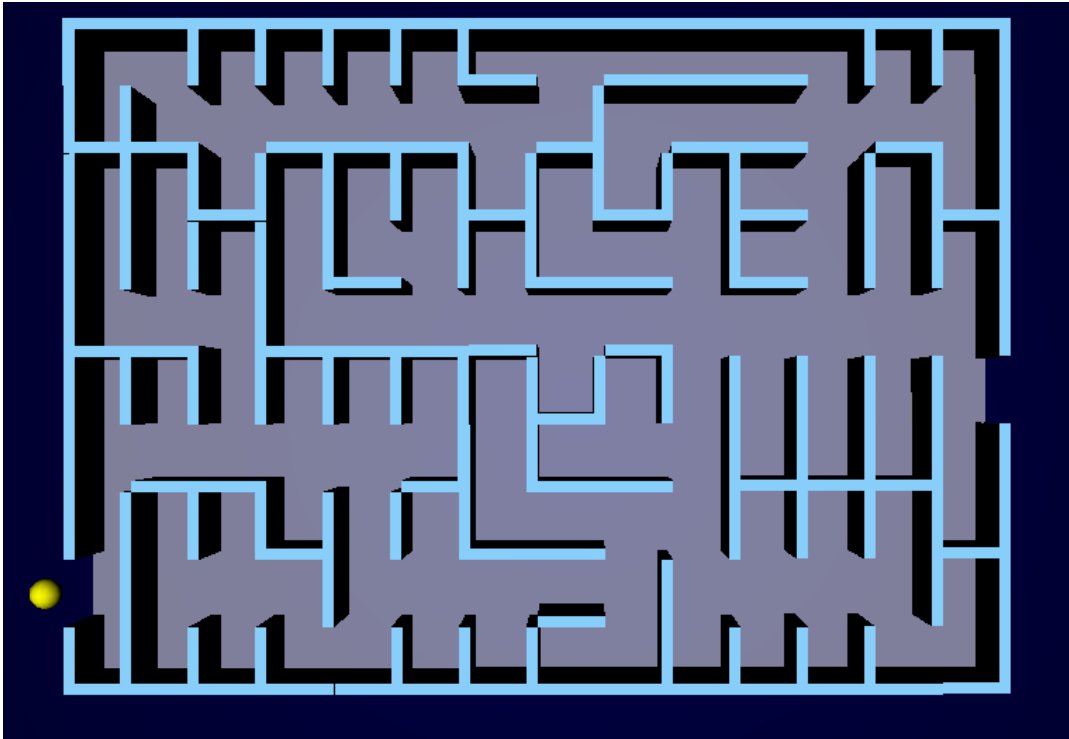
```
DEF Avatar Transform {  
  translation <#= Laberinto3D.Camino[0].X * 3 #>,0,<#= (Laberinto3D.Camino[0].Z * 3)+4 #>  
  children  
  [  
    Shape{  
      appearance Appearance{  
        material Material {diffuseColor 1,1,0}  
      }  
      geometry Sphere{  
        radius 0.7  
      }  
    }  
    DEF sensoravatar TouchSensor {}  
  
    DEF BALLVIEW Viewpoint {  
      description "Avatar"  
      position 0 12 0  
      orientation 0 1 1 3.14  
    }  
  ]  
}
```

```
DEF temporizador TimeSensor  
{  
  cycleInterval <#= 5 #>  
  loop FALSE  
}
```

- El evento de clic hace que la ruta active el temporizador:

```
ROUTE sensoravatar.touchTime TO temporizador.set_startTime
```

- Luego de esta modificación, ya hay una posición inicial en la cual no hay movimiento hasta que el usuario no hace clic. También se detiene en una posición final. Las posiciones inicial y final para un laberinto izquierda/derecha se pueden ver aquí:



## 7. PRUEBAS

Para este proyecto se probaron cada una de las etapas de construcción. La complejidad del software final no se realizó en un solo paso. Es el producto de múltiples pequeñas funcionalidades que se empezaron a desarrollar desde una consola. Se validó desde un algoritmo para que autoconstruyera laberintos hasta la generación de una simulación 3D. En todos los casos se presentan a continuación las respectivas pruebas documentadas:

### 6.1 PRUEBA DE ESCRITORIO 4X4 (CONSOLA)

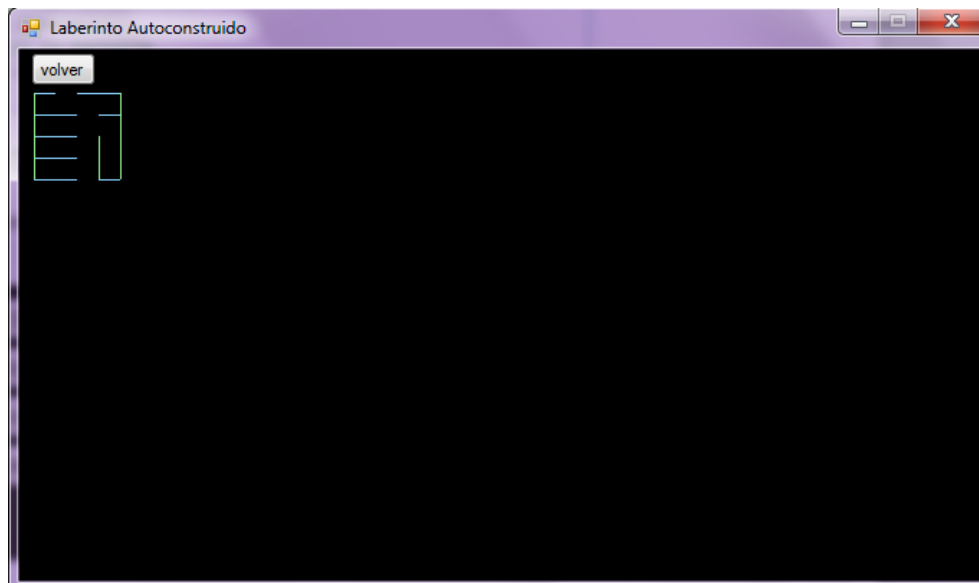
```
file:///D:/TRAINING/dotNe
¿Cuánto de ancho?:
4
¿Cuánto de alto?:
4_
```

Paso 1	Paso 2	Paso 3	Paso 4
<pre>file:///D:/TRAINING/dotNe US US US US US US PE US US PE LQ PE _ US US PE US</pre>	<pre>file:///D:/TRAINING/dotNe US US US US US PE PE US PE LQ LQ PE US PE PE US</pre>	<pre>file:///D:/TRAINING/dotNe US PE US US PE LQ PE US PE LQ LQ PE US PE PE US</pre>	<pre>file:///D:/TRAINING/dotNe US PE US US PE LQ PE PE PE LQ LQ LQ US PE PE PE</pre>
Paso 5	Paso 6	Paso 7	Paso 8
<pre>file:///D:/TRAINING/dotNe US PE US US PE LQ PE PE PE LQ LQ LQ PE LQ PE PE</pre>	<pre>file:///D:/TRAINING/dotNe US PE US US PE LQ PE PE PE LQ LQ LQ PE LQ PE LQ</pre>	<pre>file:///D:/TRAINING/dotNe US PE US US PE LQ PE PE LQ LQ LQ LQ PE LQ PE LQ</pre>	<pre>file:///D:/TRAINING/dotNe US PE US PE PE LQ PE LQ LQ LQ LQ LQ PE LQ PE LQ</pre>
Paso 9	Paso 10	Paso 11	Paso 12
<pre>file:///D:/TRAINING/dotNe PE PE US PE LQ LQ PE LQ LQ LQ LQ LQ PE LQ PE LQ</pre>	<pre>file:///D:/TRAINING/dotNe PE PE US PE LQ LQ PE LQ LQ LQ LQ LQ PE LQ LQ LQ</pre>	<pre>file:///D:/TRAINING/dotNe PE LQ PE PE LQ LQ PE LQ LQ LQ LQ LQ PE LQ LQ LQ</pre>	<pre>file:///D:/TRAINING/dotNe PE LQ PE PE LQ LQ PE LQ LQ LQ LQ LQ LQ LQ LQ LQ</pre>
Paso 13	Paso 14	Paso 15	Paso 16
<pre>file:///D:/TRAINING/dotNe PE LQ LQ PE LQ LQ PE LQ LQ LQ LQ LQ LQ LQ LQ LQ</pre>	<pre>file:///D:/TRAINING/dotNe PE LQ LQ LQ LQ LQ PE LQ LQ LQ LQ LQ LQ LQ LQ LQ</pre>	<pre>file:///D:/TRAINING/dotNe PE LQ LQ LQ LQ LQ LQ LQ LQ LQ LQ LQ LQ LQ LQ LQ</pre>	<pre>file:///D:/TRAINING/dotNe LQ LQ LQ LQ LQ LQ LQ LQ LQ LQ LQ LQ LQ LQ LQ LQ</pre>

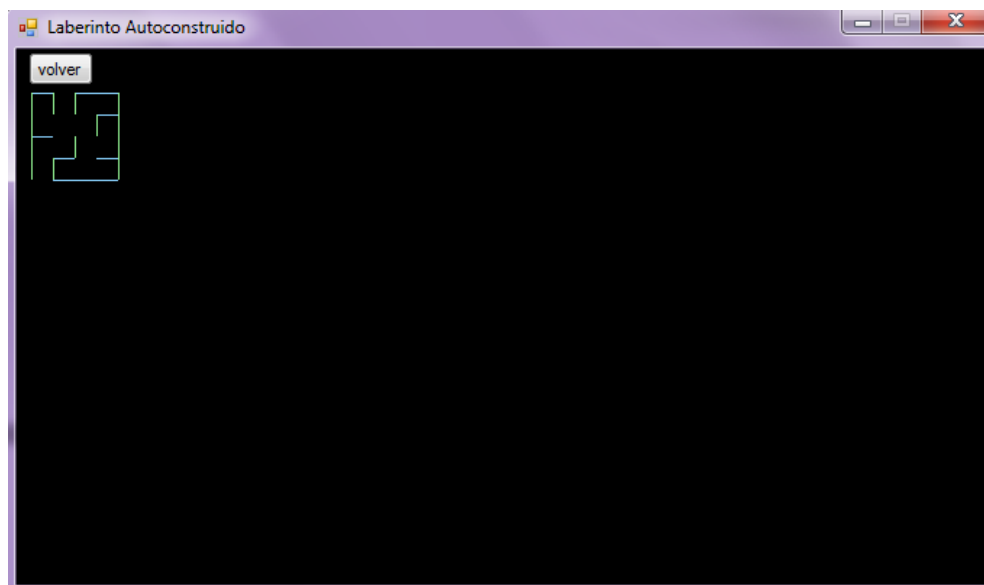
## 6.2 Pruebas de autoconstrucción (Winform)

- Prueba 1: laberinto pequeño con mínima complejidad.

Casilla:	15
Ancho:	4
Alto:	4
Recorrer:	Arriba/Abajo
<input type="button" value="Generar laberinto"/>	



Generado 1.1



Generado 1.2

Es el laberinto más pequeño que se puede construir: 4x4. La casilla es la más grande que se dejó disponible. El recorrido es arriba/abajo, basta con verificar la entrada y la salida.



- Prueba 2: laberinto mediano con mínima complejidad.

Casilla: 15  
Ancho: 24  
Alto: 12  
Recorrer: Arriba/Abajo  
[Generar laberinto](#)



Generado 2.1



Generado 2.2

- Prueba 3: laberinto grande con mínima complejidad.

Casilla: 15  
Ancho: 42  
Alto: 20  
Recorrer: Arriba/Abajo  
Generar laberinto



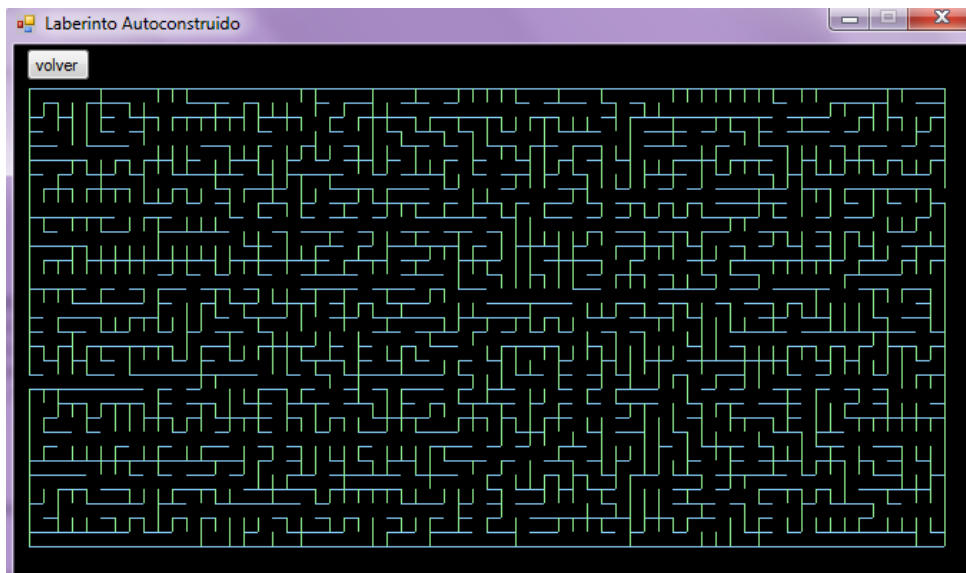
Generado 3.1



Generado 3.2

- Prueba 4: laberinto grande con complejidad media. Salidas a la izquierda y a la derecha.

Casilla: 10  
Ancho: 64  
Alto: 32  
Recorrer: Izquierda/Derecha  
Generar laberinto



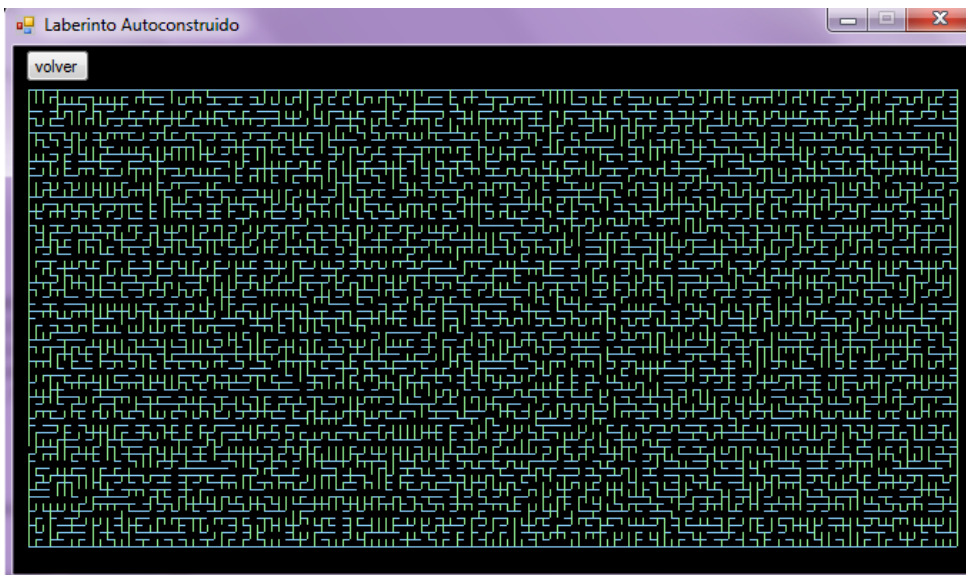
Generado 4.1



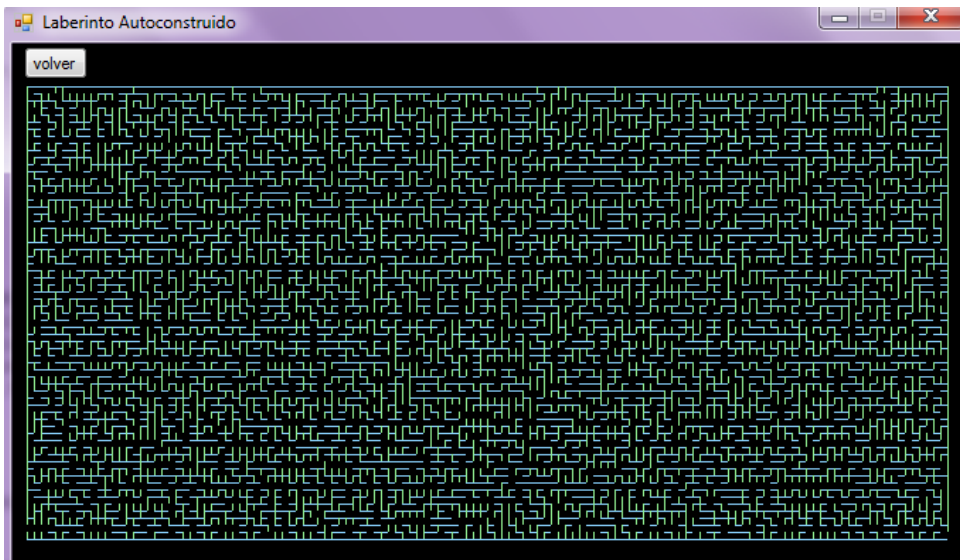
Generado 4.2

- \*Prueba 5: laberinto grande con complejidad alta. Salidas a la izquierda y a la derecha.

Casilla: 5  
Ancho: 130  
Alto: 64  
Recorrer: Izquierda/Derecha  
Generar laberinto



Generado 5.1.

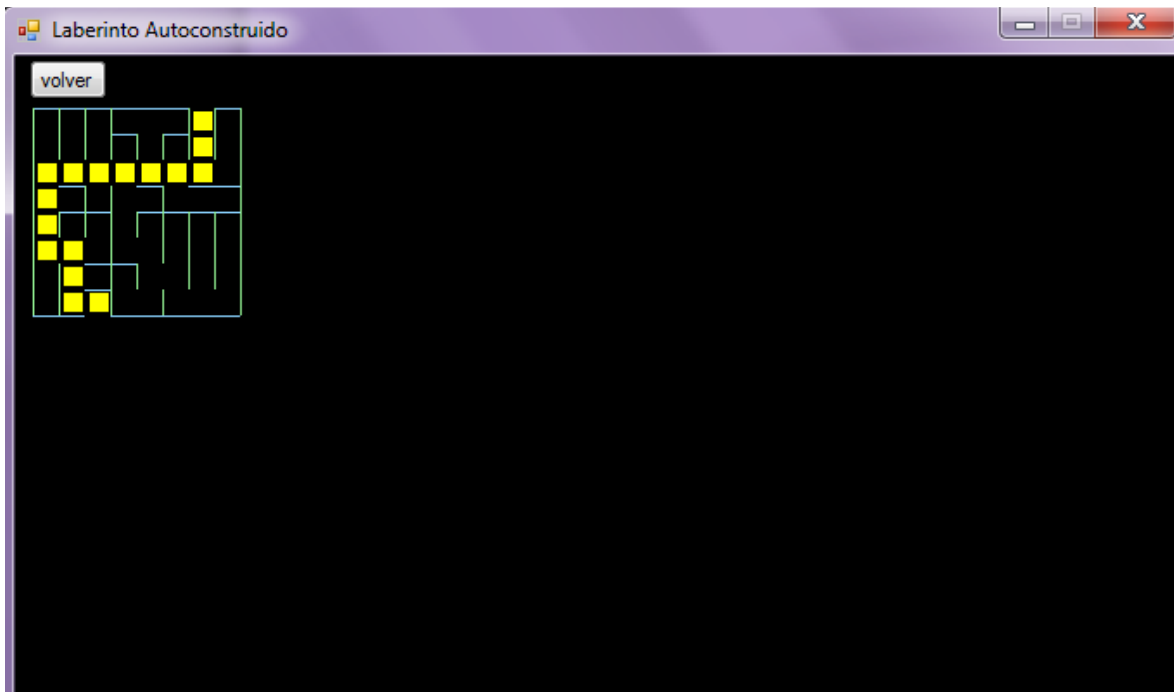


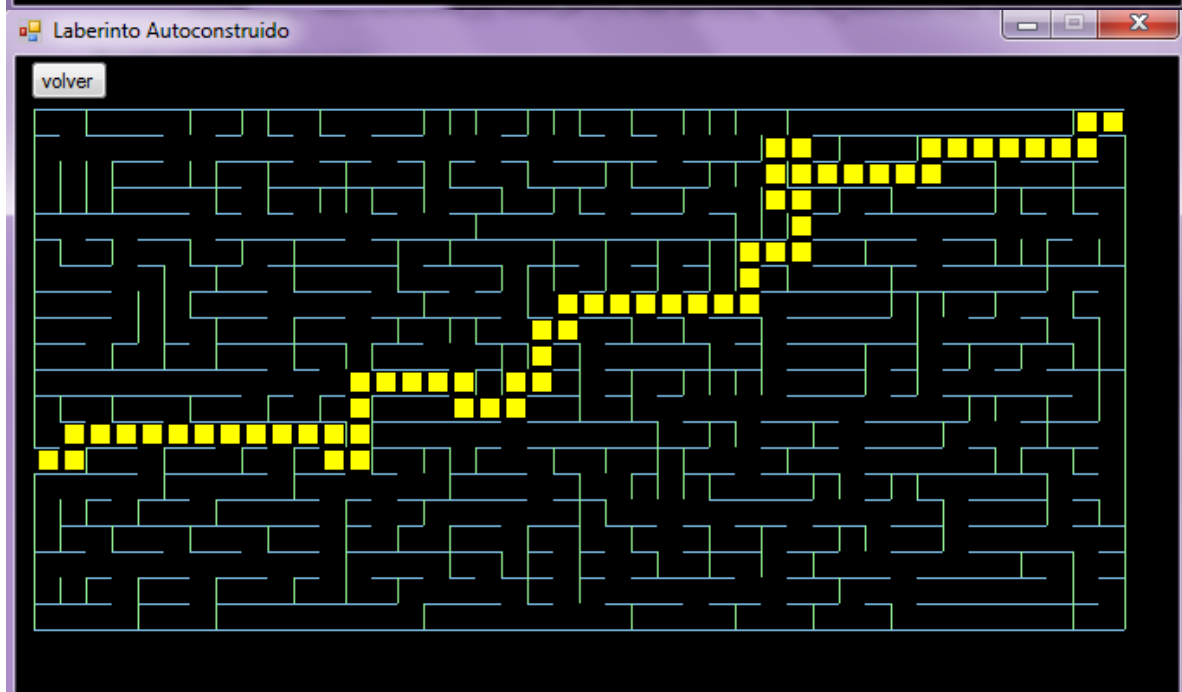
Generado 5.2.

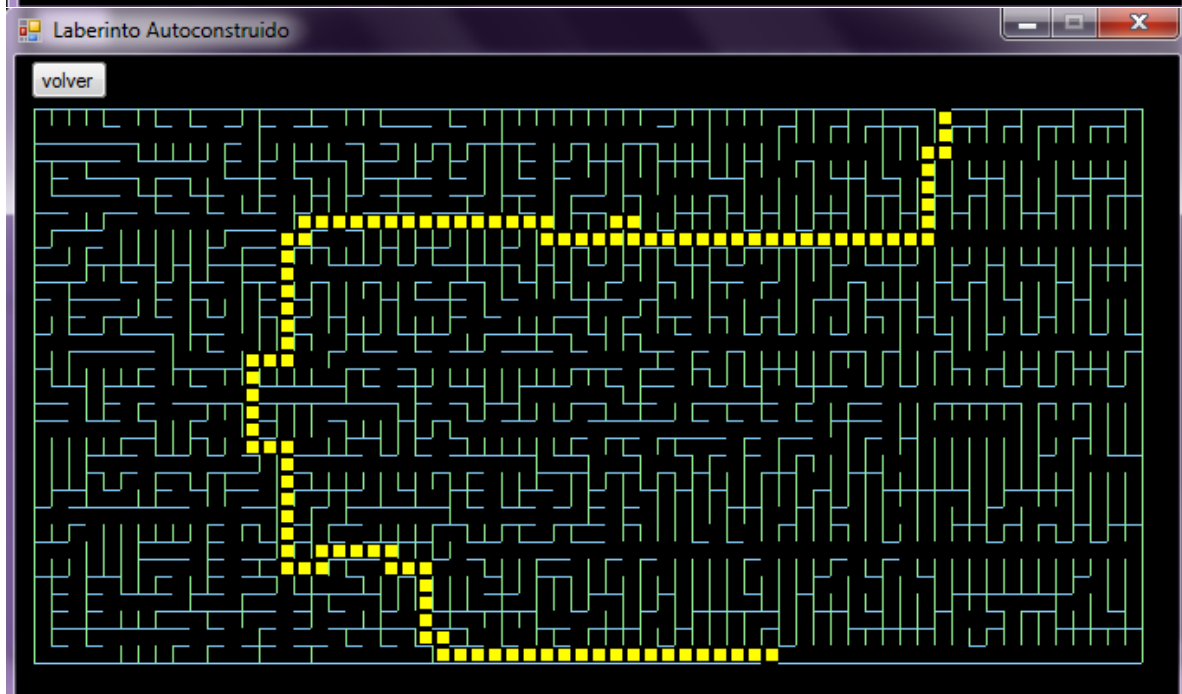
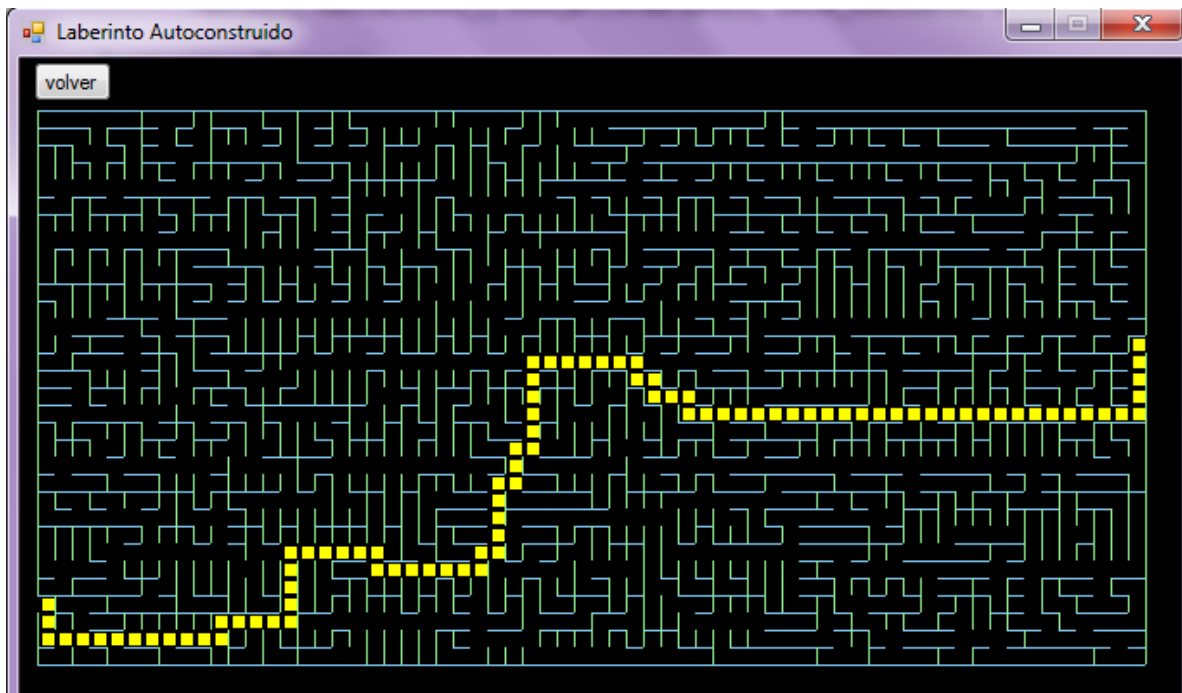
- Se podría seguir generando laberintos de complejidad cada vez más alta. Sin embargo, esto es suficiente para demostrar que el algoritmo de autoconstrucción de laberintos está listo para migrarse a VRML. Sin embargo, sigue pendiente un algoritmo para resolver dichos laberintos.

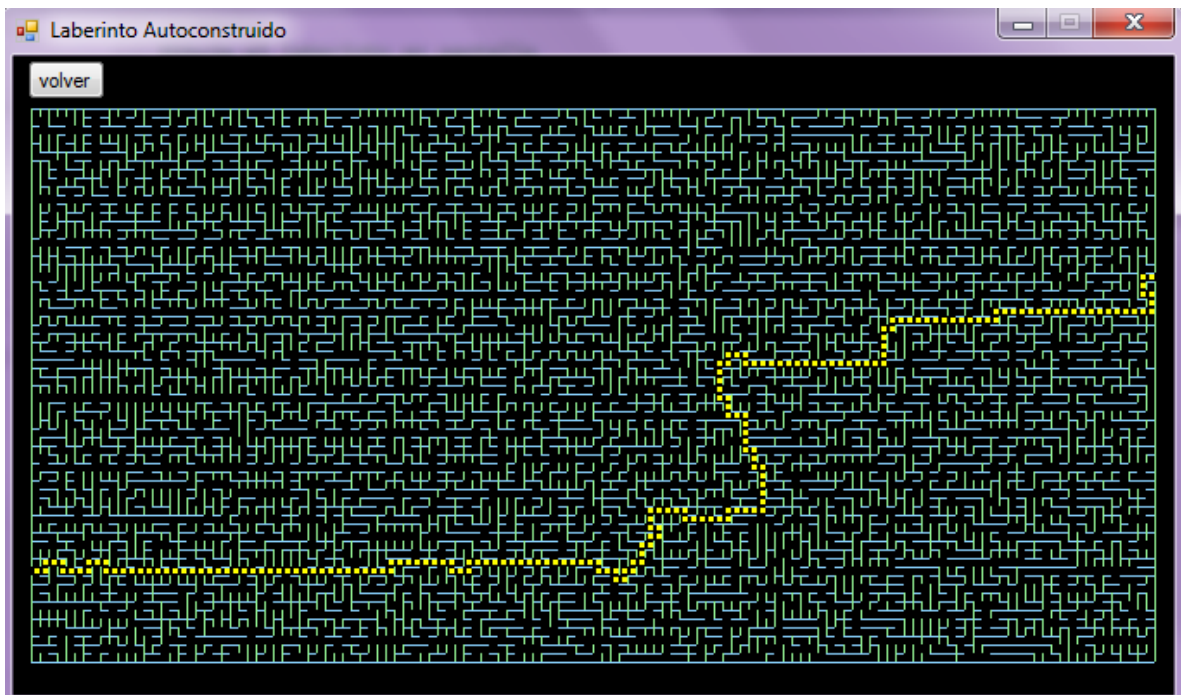
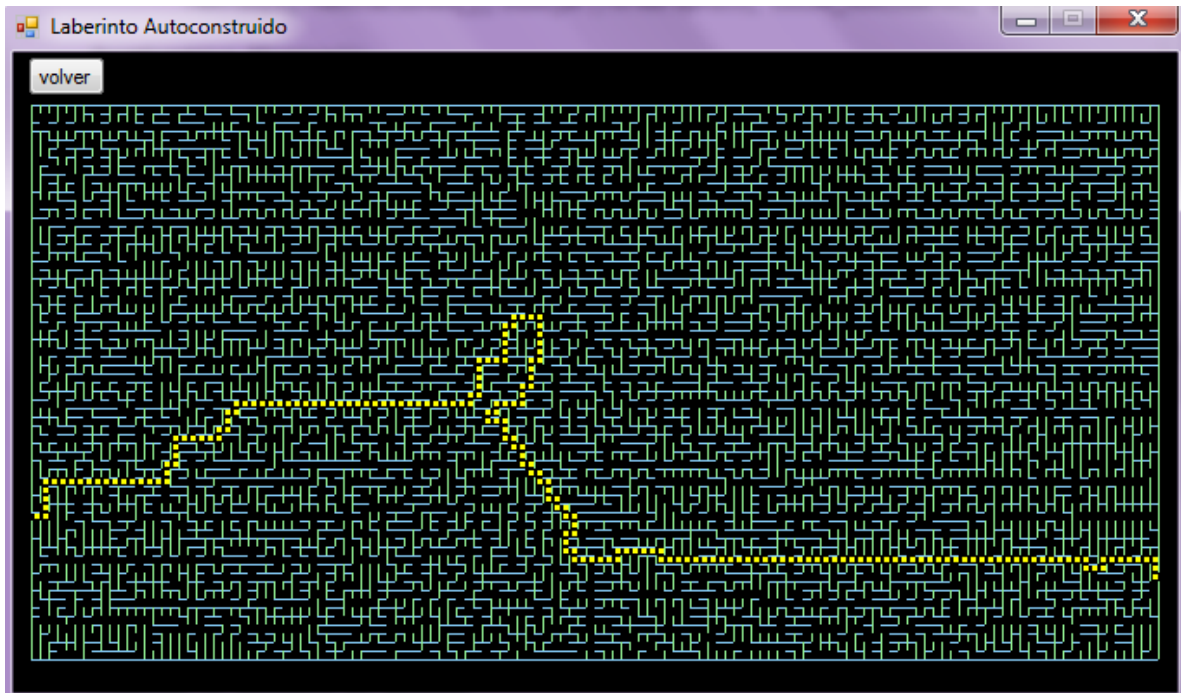
### 6.3 Pruebas de resolución con un Árbol 4-Ario

- Al final, el algoritmo desarrollado es capaz de hallar un camino para ir desde y hacia cualquier habitación dentro del laberinto. Los siguientes ejemplos serán para ir desde la entrada hacia la salida.











## 6.4 Pruebas con búsqueda de objetivos aleatorios

- Se eligió un laberinto de complejidad media-alta y varios objetivos para ilustrar el concepto de búsqueda.

Casilla:	10	▼
Ancho:	64	▼
Alto:	32	▼
Objetivos:	4	▼
Recorrer:	Izquierda/Derecha	▼
Retardo:	4000	▼
<button>Generar laberinto</button>		



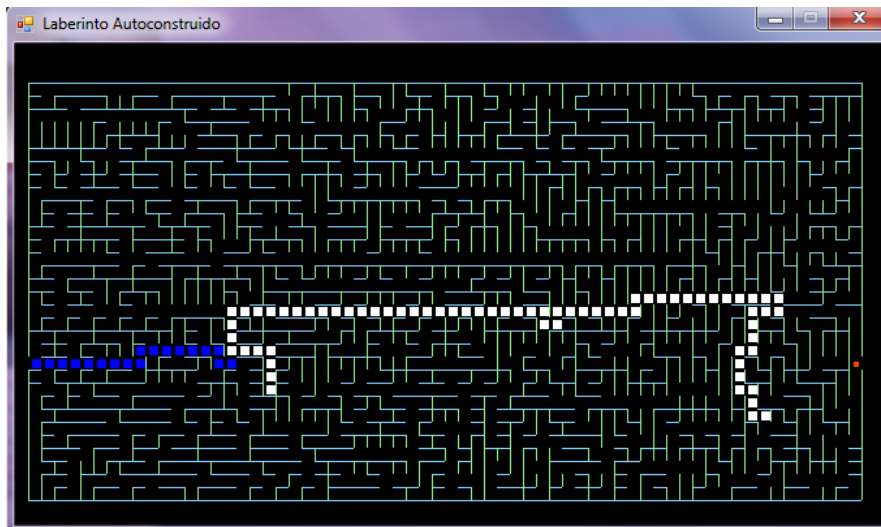
Laberinto inicial

- Se pueden ver los puntos rojos que marcan la entrada y la salida.



Objetivo 1  
(Camino Azul)

- El avatar llega cerca a la salida



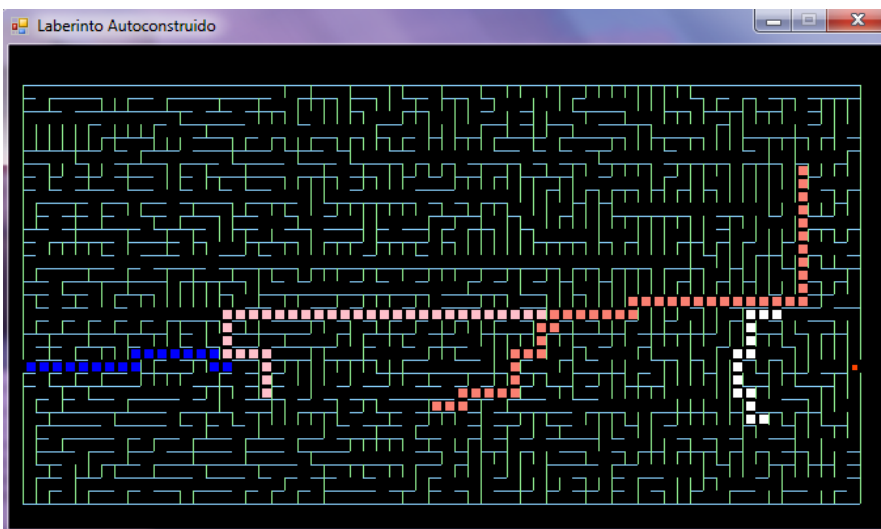
## Objetivo 2 (Camino Blanco)

- El segundo objetivo le hace devolverse bastante. Cerca a la entrada



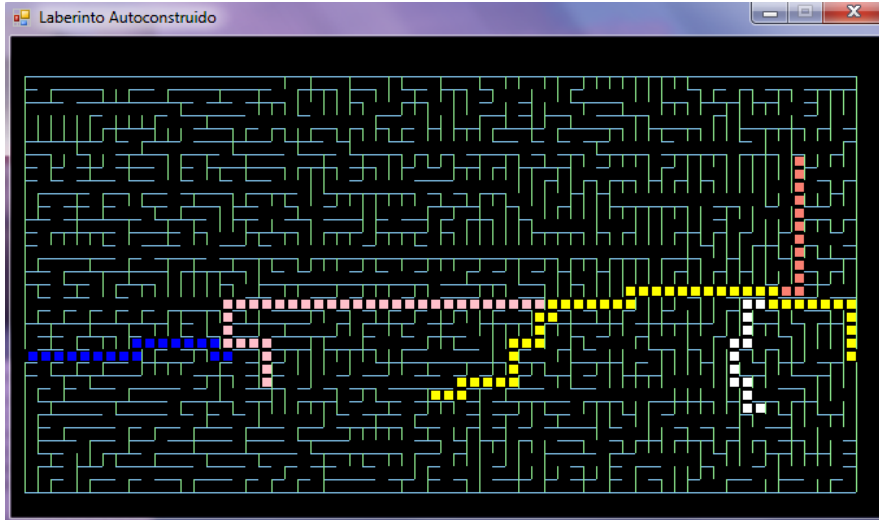
## Objetivo 3 (Camino Rosa)

- El tercer objetivo es muy cerca de la esquina superior derecha.



## Objetivo 4 (Camino Rojo)

- El cuarto objetivo es un poco más abajo del centro del laberinto.



Objetivo final  
(Camino Amarillo)

- Luego de alcanzar los 4 objetivos, el avatar busca el camino para salir del laberinto.

- Esto no sería posible si no se partiera de la premisa que se puede conectar con un **único** camino desde y hacia cualquier habitación. No hay múltiples caminos.

## 8. PUBLICACIÓN

Se desarrolló un software que es capaz de poner en práctica la teoría de construcción de laberintos unicursales y generar modelos de simulación 3D en VRML para resolverlos. La forma de publicar este trabajo es por medio de un instalador o de una página web. En ambos casos se utiliza la misma lógica de generación (Engine), que fue descrita y documentada en las etapas anteriores de este trabajo.

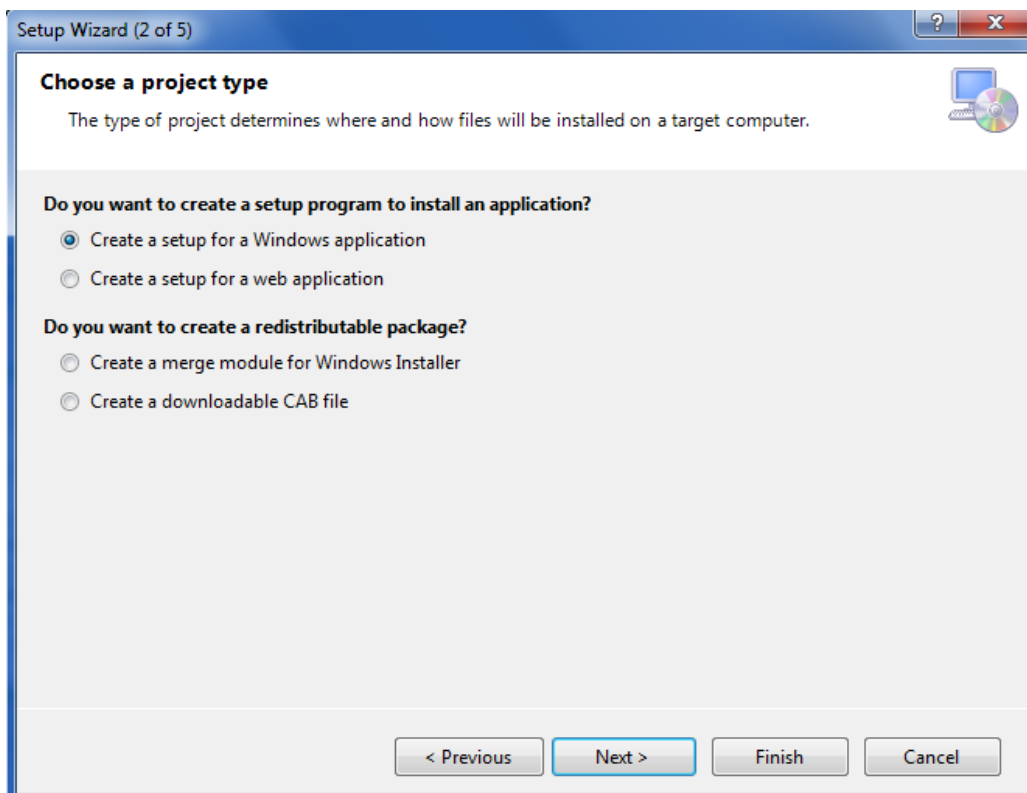
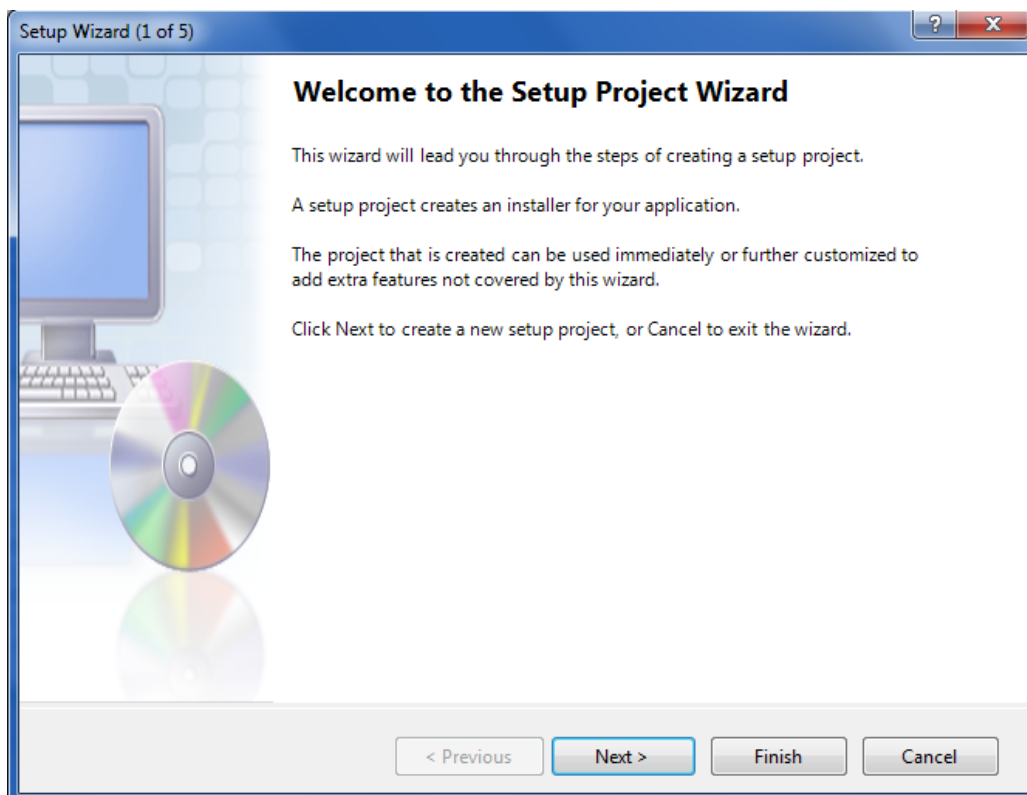
### 8.1 Instalador local (Desktop)

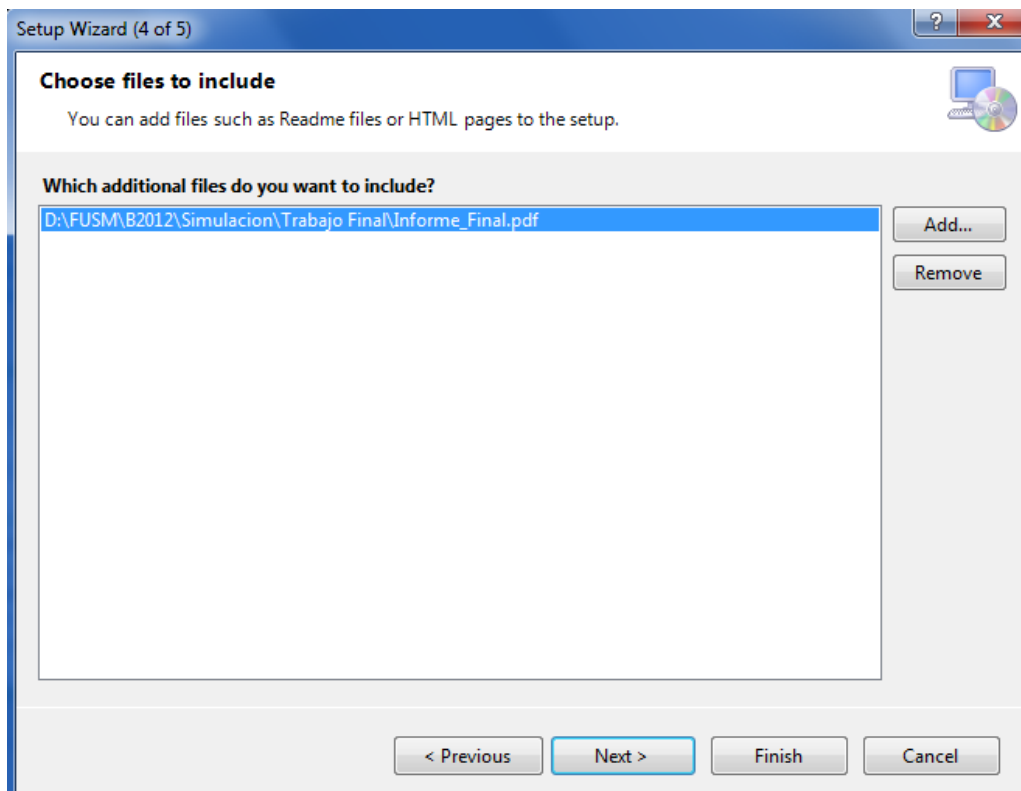
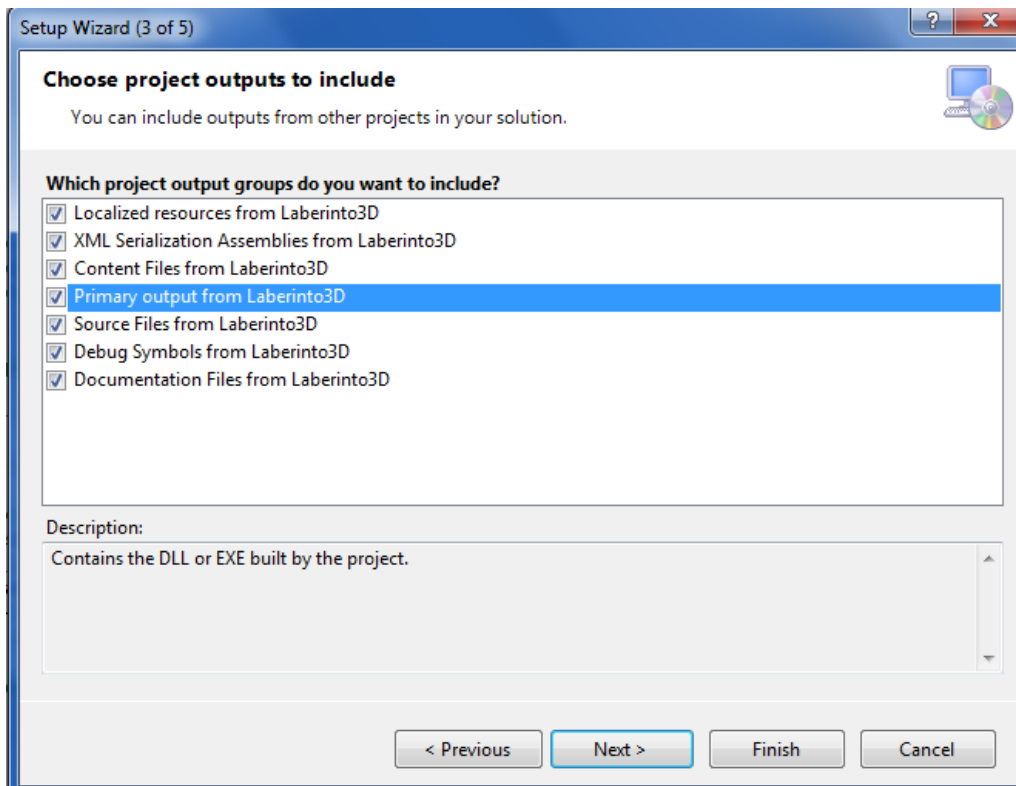
La aplicación local fue la más fácil de construir. Consiste en publicar una serie de campos que permiten parametrizar el motor de generación VRML. Para permitir una fácil distribución de esta aplicación, se suspendió el desarrollo en el proyecto Winform y se generó un instalador. Futuras mejoras y parámetros en pantalla serán implementados sobre la aplicación Web.

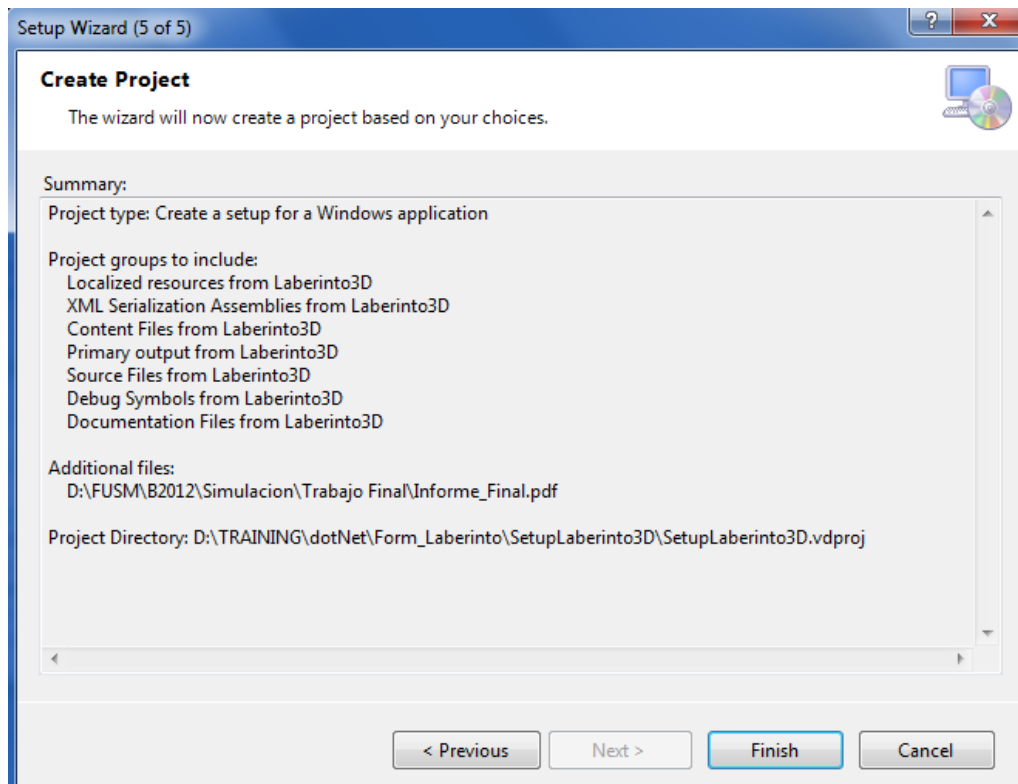
La apariencia de la versión final local es esta:



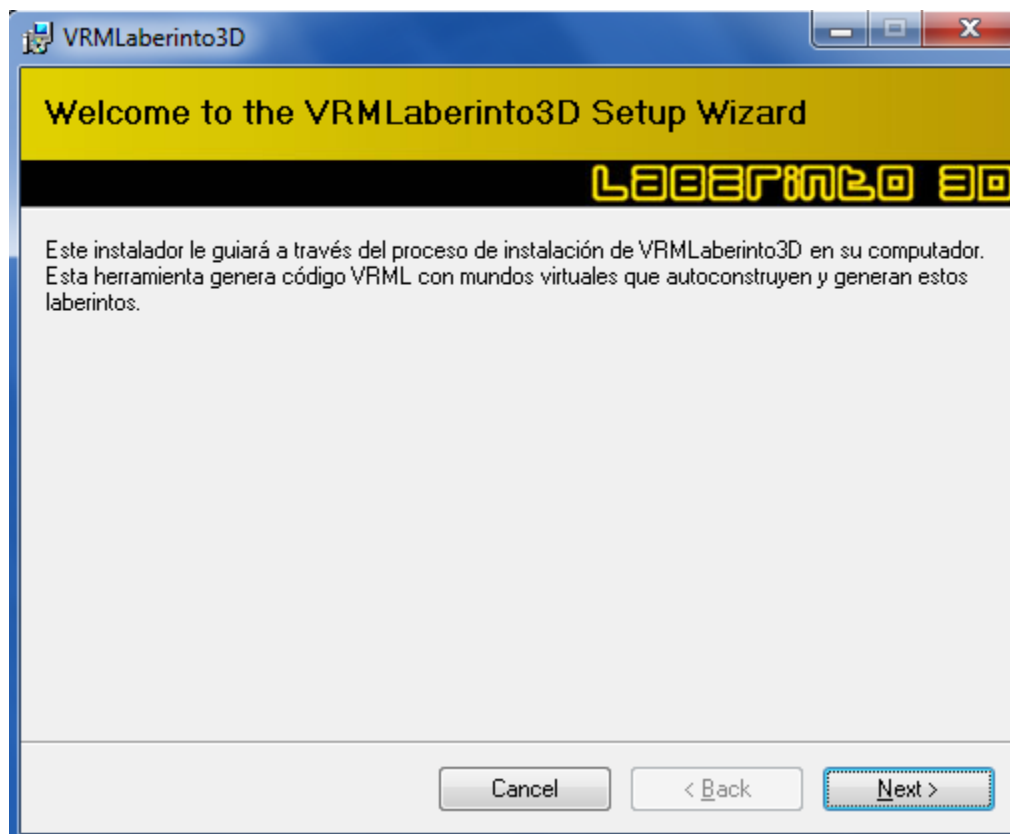
El proceso de creación del instalador es el que sigue:







Al final del proceso de generación del instalador, el asistente luce como sigue:



## 8.2 Página Web (ASP.Net)



## 9. VERSIONAMIENTO

El versionamiento del proceso se desarrolló se implemento con base en este informe. Existen sucesivas iteraciones de funcionalidades que se han venido implementando. Se puede saber exactamente qué fue desarrollado, cuándo y de qué forma, simplemente revisando esa documentación.

La tabla de versiones de este proyecto es:

Documento	Versión software	Descripción	Fecha
<b>ProyectoFinal_SM_v1</b>	0.1	Algoritmo de autoconstrucción en consola (Parte 1)	08 Noviembre 2012
<b>ProyectoFinal_SM_v2</b>	0.1	Algoritmo de autoconstrucción en consola (Parte 2). Primera prueba de escritorio.	09 Noviembre 2012
<b>ProyectoFinal_SM_v3</b>	0.3	Depuración del algoritmo de autoconstrucción. Pruebas con laberintos de mayor tamaño.	10 Noviembre 2012
<b>ProyectoFinal_SM_v4</b>	0.5	Construcción de Renderizado 2D reutilizando las clases de autoconstrucción de laberintos.	12 Noviembre 2012
<b>ProyectoFinal_SM_v5</b>	0.7	Planteamiento de un algoritmo de resolución dinámica basada en un árbol 4-ario.	15 Noviembre 2012
<b>ProyectoFinal_SM_v6</b>	0.8	Reutilización del algoritmo solución para implementar objetivos dentro de la simulación (Bonus)	16 Noviembre 2012
<b>ProyectoFinal_SM_v7</b>	0.9	Generación de código VRML para simulación del mundo virtual con un avatar.	28 Noviembre 2012
<b>ProyectoFinal_SM_v8</b>	1.0	Parametrización del formulario de generación VRML.	29 Noviembre 2012

En cuanto a las clases de la solución .Net, la siguiente tabla muestra sus diferentes versiones y comentarios:

Archivo fuente	Paquete	Tipo	Versión actual	Comentario
<b>Autoconstruir.cs</b>	Form	Formulario	1.0	Se cerró en esta versión por migración a Web.
<b>Program.cs</b>	Form	Clase	1.0	Generado automáticamente
<b>Fila.cs</b>	Engine	Clase	0.3	Estable como estructura de datos.
<b>Habitacion.cs</b>	Engine	Clase	0.8	La estructura básica es estable hace varias versiones. Cuando se actualiza es para adicionar alguna que otra propiedad.
<b>Laberinto.cs</b>	Engine	Clase	0.8	Es la clase central del Engine. La lógica más compleja está aquí.
<b>Nodo.cs</b>	Engine	Clase	0.8	Estructura para la conformación del árbol 4-ario solución.
<b>Punto.cs</b>	Engine	Clase	0.8	Punto que representa la entrada y la salida en el Renderizado 2D.
<b>Avatar3D.tt</b>	T4	Plantilla de generación	0.9	Plantilla para generar el avatar de los laberintos.
<b>Avatar3D.cs</b>	T4	Clase preprocesadora	0.9	Código generado automáticamente.
<b>AvatarCod.cs</b>	T4	Clase parcial	0.9	Clase que extiende la plantilla para lógica adicional
<b>Habitacion3D.tt</b>	T4	Plantilla de generación	0.9	Plantilla para generar cada habitación de los laberintos.
<b>Habitacion3D.cs</b>	T4	Clase preprocesadora	0.9	Código generado automáticamente.
<b>HabitacionCod.cs</b>	T4	Clase parcial	0.9	Clase que extiende la plantilla para lógica adicional
<b>Laberinto3D.tt</b>	T4	Plantilla de generación	0.9	Plantilla para generar el recorrido de los laberintos.
<b>Laberinto3D.cs</b>	T4	Clase preprocesadora	0.9	Código generado automáticamente.
<b>LaberintoCod.cs</b>	T4	Clase parcial	0.9	Clase que extiende la plantilla para lógica adicional
<b>VRML.tt</b>	T4	Plantilla de generación	0.9	Plantilla principal para generar VRML de laberintos.
<b>VRML.cs</b>	T4	Clase preprocesadora	0.9	Código generado automáticamente.
<b>VRMLCod.cs</b>	T4	Clase parcial	0.9	Clase que extiende la plantilla para lógica adicional

## 10. CONCLUSIONES

- VRML, al no ser un lenguaje de programación propiamente dicho, sino de representación de mundos virtuales, requiere indudablemente de la articulación con otros lenguajes para realizar simulaciones modificadas por parámetros de entrada.
- VRML es una herramienta muy poderosa para la representación de mundos virtuales. Ofrece una rica variedad de primitivas que lo hace muy interactivo y muy intuitivo hacia el usuario.
- A diferencia de OpenGL, el VRML tiene el poder de ejecutarse en un navegador. Esto hace que se pueda integrar fácilmente con aplicaciones web y que no haya necesidad de instalar nada más que el plugin de Cortona 3D.
- Para que un proyecto sea realmente utilizado por un usuario, es importante proveer a este una interfaz sencilla para utilizarlo. La publicación es una etapa clave de la metodología utilizada.