3D Maze Screenshot (Stefan Haubenthal)

```
#include <stdio.h>
#include <stdlib.h>         // random
#define random(MAX) rand()%MAX
//#include <time.h>           // randomize
#include <tgi.h>
#include <conio.h>          // cgetc
#if 0
#include <dos.h>            // sound, nosound, delay
#else
#define sound(FREQ)
#define nosound()
#define delay(TIME)
#endif

enum {UP,RIGHT,DOWN,LEFT};  // equally NORTH,EAST,SOUTH,WEST

#define RIGHT  1            // walls and borders
#define BOTTOM 2


typedef struct
  {
  unsigned char  Walls;
  unsigned char  Border;
  unsigned short Path;
  } CELL;


struct
  {
  int   Width, Height, NumCells;
  int   Dir[ 3 ];
  CELL *Maze;
  int   Entry;
  int   CenterX;
```

```
  int   CenterY;
  } g;


#define Cell( Row, Col )       ((Row) * g.Width + (Col))
#define IsConnected( C1, C2 )  (g.Maze[ (C1) ].Path == g.Maze[ (C2) ].Path)
#define GoodDir( Cell, Dir )   ((g.Maze[ (Cell) ].Border & (Dir)) == 0)

#define ROTATE_LEFT( Direction )   ((Direction + 3) & 0x3)
#define ROTATE_RIGHT( Direction )  ((Direction + 1) & 0x3)

enum { CENTER_WALL,LEFT_WALL,RIGHT_WALL,RIGHT_EDGE,LEFT_EDGE};
enum { UNMOVED, MOVED,BLOCKED,EXITED };


// Draws a wall shape

void DrawShape(int shape,int Size)
  {
  int OldSize=Size*2;

  // vertical lines shared across all shapes (closes the shapes)
  tgi_line( g.CenterX - Size, g.CenterY - Size,
        g.CenterX - Size, g.CenterY + Size);
  tgi_line( g.CenterX + Size, g.CenterY - Size,
        g.CenterX + Size, g.CenterY + Size);

  switch (shape)
    {
    case CENTER_WALL:
      tgi_line( g.CenterX - Size, g.CenterY - Size,
            g.CenterX + Size, g.CenterY - Size);
      tgi_line( g.CenterX - Size, g.CenterY + Size,
            g.CenterX + Size, g.CenterY + Size);
      break;

    case LEFT_WALL:
      tgi_line( g.CenterX - OldSize, g.CenterY - OldSize,
            g.CenterX - Size,    g.CenterY - Size);
      tgi_line( g.CenterX - OldSize, g.CenterY + OldSize,
            g.CenterX - Size,    g.CenterY + Size);
      break;

    case RIGHT_WALL:
      tgi_line( g.CenterX + OldSize, g.CenterY - OldSize,
            g.CenterX + Size,    g.CenterY - Size);
      tgi_line( g.CenterX + OldSize, g.CenterY + OldSize,
            g.CenterX + Size,    g.CenterY + Size);
      break;

    case RIGHT_EDGE:
      tgi_line( g.CenterX + OldSize, g.CenterY - Size,
            g.CenterX + Size,    g.CenterY - Size);
      tgi_line( g.CenterX + OldSize, g.CenterY + Size,
            g.CenterX + Size,    g.CenterY + Size);
      break;

    case LEFT_EDGE:
      tgi_line( g.CenterX - OldSize, g.CenterY - Size,
```

```
            g.CenterX - Size,    g.CenterY - Size);
      tgi_line( g.CenterX - OldSize, g.CenterY + Size,
            g.CenterX - Size,    g.CenterY + Size);
      break;
    }
  }


// moves one cell in direction 'Direction' starting from cell *x,*y
//   assumes *x,*y is within the maze and Direction is a valid direction.

int MoveXY(int Direction,int *x,int *y)
  {
  switch (Direction)
    {
    case (UP):
      if( *y==0 )  return( BLOCKED );
      if( g.Maze[ Cell( (*y)-1, *x ) ].Walls & BOTTOM )  return( BLOCKED );
      (*y)--;
      return( MOVED );

    case (RIGHT):
      if( g.Maze[ Cell( *y, *x ) ].Walls & RIGHT )  return( BLOCKED );
      (*x)++;
      return( MOVED );

    case (DOWN):
      if( g.Maze[ Cell( *y, *x ) ].Walls & BOTTOM )
        {
        if( *x != g.Entry)  return( BLOCKED );
        else                return( EXITED );
        }
      (*y)++;
      return( MOVED );

    case (LEFT):
      if( *x==0 )  return( BLOCKED );
      if( g.Maze[ Cell( *y, (*x)-1 ) ].Walls & RIGHT )  return( BLOCKED );
      (*x)--;
      return( MOVED );
    }

  return( BLOCKED );
  }


//  Calculate which shapes are needed for a given cell

void DrawBlock(int Direction, int x, int y,int Size)
  {
  int tx, ty;             // temporary position for movement tests
  int movement;           // result of movement tests

  // Look to the left - draw nessary shapes
  tx = x;  ty = y;

  movement = MoveXY( ROTATE_LEFT( Direction ), &tx, &ty );

  if( movement == BLOCKED )
```

```
    {
    DrawShape( LEFT_WALL, Size );
    }
  else
    {
    if( movement == MOVED )
      {
      if( MoveXY( ROTATE_RIGHT( ROTATE_LEFT( Direction ) ), &tx, &ty) == BLOCKED )
        {
        DrawShape(LEFT_EDGE,Size);
        }
      else
        {
        } // Draw nothing - there is just empty space to the left.
      }
    else
      {
      } // Draw nothing - the Exit is to the left
    }

  // Look to the right - draw nessary shapes
  tx = x;  ty = y;

  movement = MoveXY( ROTATE_RIGHT( Direction ), &tx, &ty );

  if( movement == BLOCKED )
    {
    DrawShape( RIGHT_WALL, Size );
    }
  else
    {
    if( movement == MOVED )
      {
      if( MoveXY( ROTATE_LEFT( ROTATE_RIGHT( Direction ) ), &tx, &ty ) == BLOCKED )
        {
        DrawShape( RIGHT_EDGE, Size );
        }
      else
        {
        } // Draw nothing - there is just empty space to the right.
      }
    else
      {
      } // Draw nothing - the Exit is to the right
    }

  // Look straight ahead
  if( MoveXY( Direction, &x, &y ) == BLOCKED )
    {
    DrawShape( CENTER_WALL, Size );
    }
  }


// Draws the maze as seen from cell x,y facing direction Direction

void Draw3d( int Direction, int x, int y )
  {
  int Size = g.CenterX / 2;
```

```
  while( 1 )
    {
    DrawBlock( Direction, x, y, Size );

    if( MoveXY( Direction, &x, &y ) != MOVED )  break;

    // reduce shape size
    Size /= 2;
    }
  }


// Interactive exploration of the maze in 3d

void Explore( void )
  {
  unsigned char GraphicsMode = TGI_MODE_320_200_2;     // use VGA 640x480 16 color mod

  int c;                        // character input
  int x, y;                     // "position" of viewpoint in maze
  int Direction;                // direction of view (enumerated)
  int Status;                   // results of attempted move.

  tgi_load( GraphicsMode );
  tgi_init( );
  g.CenterX = tgi_getxres()/2;
  g.CenterY = tgi_getyres()/2;
  tgi_setcolor( COLOR_WHITE );

  // random starting position and direction
  x = random( g.Width );
  y = random( g.Height );
  Direction = random( 4 );
  Draw3d( Direction, x, y );

  c = 0;

  while( 1 )
    {
    Status = UNMOVED;

    c = cgetc();

    if( c=='q' || c=='Q')  break;

    switch( c )
      {
      case 'J': // Rotate Left
      case 'j':
      case '4':
        Direction = ROTATE_LEFT( Direction );
        break;

      case 'L': // Rotate Right
      case 'l':
      case '6':
        Direction = ROTATE_RIGHT( Direction );
        break;
```

```
      case 'I': // Move Forward
      case 'i':
      case '8':
        Status = MoveXY( Direction, &x, &y );
        break;

      case 'K':
      case 'k':
      case '2':
        Status = MoveXY( (Direction + 2 ) % 4 ,&x, &y );
        break;

      default:
        break;
      }

    if( Status==BLOCKED)
      {
      // beep the speaker
      sound(550);
      delay(10);
      nosound();
      }
    else
      {
      if( Status == EXITED )
        {
        break;
        }
      else
        {
        tgi_clear();
        Draw3d( Direction, x, y );
        }
      }
    }

  tgi_done();

  if( Status == EXITED )
    {
    printf("Congratulations, you found your way out!\n");
    }
  }


// Joins two cells together into the same path.

void Join( int CellNum, int Dir )
  {
  int i, OldPath;

  g.Maze[ CellNum ].Walls &= ~Dir;   // clear the wall between them

  // now merge the paths

  OldPath = g.Maze[ CellNum + g.Dir[ Dir ] ].Path;
```

```
  for( i = g.NumCells; i >= 0; i-- )
    {
    if( g.Maze[ i ].Path == OldPath )
      {
      g.Maze[ i ].Path = g.Maze[ CellNum ].Path;
      }
    }
  }


// Randomly connects a cell with either its bottom or right neighbor.

int Connect( int CellNum )
  {
  int Neighbor, Offset, Dir;

  Dir = random( 2 ) + 1;   // random direction, 1 or 2

  Neighbor = CellNum + g.Dir[ Dir ];

  if( !GoodDir( CellNum, Dir ) || IsConnected( CellNum, Neighbor ) )
    {
    Dir = 3 - Dir;     // get the OTHER neighbor

    Neighbor = CellNum + g.Dir[ Dir ];

    if( !GoodDir( CellNum, Dir ) || IsConnected( CellNum, Neighbor ) )
      {
      return( 0 );
      }
    }

  Join( CellNum, Dir );

  return( 1 );
  }


// Generates the maze.

void Generate( void )
  {
  int ACell, Cnt;

  // Connect a random cell in the maze to another path.
  // Continue doing this until there is only one path in the maze.

  do
    {
    for( ACell = random( g.NumCells ), Cnt = g.NumCells;
         Cnt;
         ACell++, Cnt-- )
      {
      if( ACell == g.NumCells )  ACell = 0;

      if( Connect( ACell ) )  break;
      }
    }
  while( Cnt );
```

```
  }


// Initialization code.

void Init( void )
  {
  int i, Row, Col;

  _randomize();

  g.Dir[ 1 ] = 1;                // right neighbor
  g.Dir[ 2 ] = g.Width;          // bottom neighbor

  if( (g.Maze = calloc( g.NumCells, sizeof( CELL ) )) == NULL )
    {
    puts( "ERROR: Out of memory." );
    exit( -1 );
    }

  for( Row = 0; Row < g.Height; Row++ )
    {
    g.Maze[ Cell( Row, g.Width - 1 ) ].Border |= RIGHT;
    }

  for( Col = 0; Col < g.Width; Col++ )
   {
   g.Maze[ Cell( g.Height - 1, Col ) ].Border |= BOTTOM;
   }

  for( i = 0; i < g.NumCells; i++ )
    {
    g.Maze[ i ].Path  = i;
    g.Maze[ i ].Walls = RIGHT + BOTTOM;
    }
  }


void main( int Argc, char *Argv[] )
  {
  if( Argc != 3 )
    {
    puts( "Needs dimensions, width and height." );
    }
  else
    {
    g.Width  = atoi( Argv[ 1 ] );
    g.Height = atoi( Argv[ 2 ] );
    g.NumCells = g.Width * g.Height;

    if( g.Width < 2 || g.Width > 50 || g.Height < 2 || g.Height > 50 )
      {
      puts( "Bad dimensions." );
      }
    else
      {
      Init();

      // open up a random column on the bottom
```

```
    g.Entry = random( g.Width );

    Generate();

    Explore();
    }
  }
}
```