Project 9: Maze Generator

CS 200 • 20 Points Total
Due Friday, April 21, 2017

Objectives

- Write a recursive maze generator in assembly.
- Practice using procedural calling, general assembly programming, and using a high-level language implementation as a reference for writing assembly.

Overview

This is the first part of a two-part project. For this project, you will only write the helper functions (everything except Visit) and modify the main program to test your work. If you don't show me that you tested your functions, then you won't get credit. Also, my skeleton code cheats in how it creates the maze array to make it easier to display. Ignore the header comments for the display routine and pay attention to the comments inside the routine, instead.

Recursion can be used to easily create simple tile-based mazes. The idea is you start somewhere in "solid rock" and tunnel your way two steps in a random direction, then recursively repeat. When your twisty little passage runs out of room to grow (when it's about to cross itself), you fall back to the previous level of recursion and see if you can go in another direction.

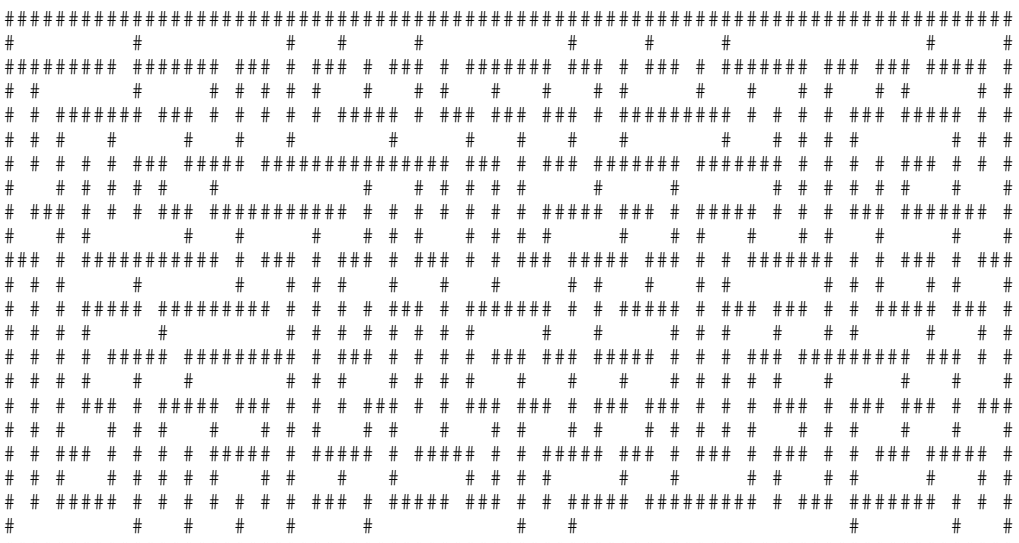
Here is a C++ implementation of such a maze generator:

```
//============================================================================
//  maze.cpp
//
//  C++ implementation of a recursive maze-generating program.
//
//  History:
//    2006.03.30 / Abe Pralle - Created
//    2010.04.02 / Abe Pralle - Converted to C++
//============================================================================

#include <iostream>
using namespace std;

//----CONSTANTS--------------------------------------------------------
#define GRID_WIDTH  79
#define GRID_HEIGHT 23

#define NORTH 0
#define EAST  1
#define SOUTH 2
#define WEST  3


//----GLOBAL VARIABLES-------------------------------------------------
char grid[GRID_WIDTH*GRID_HEIGHT];
```

```
//----FUNCTION PROTOTYPES--------------------------------------------
void ResetGrid();
int  XYToIndex( int x, int y );
int  IsInBounds( int x, int y );
void Visit( int x, int y );
void PrintGrid();



//----FUNCTIONS-----------------------------------------------------
int main()
{
  // Starting point and top-level control.

  srand( time(0) );  // seed random number generator.
  ResetGrid();
  Visit(1,1);
  PrintGrid();

  return 0;
}



void ResetGrid()
{
  // Fills the grid with walls ('#' characters).

  for (int i=0; i<GRID_WIDTH*GRID_HEIGHT; ++i)
  {
    grid[i] = '#';
  }
}



int XYToIndex( int x, int y )
{
  // Converts the two-dimensional index pair (x,y) into a
  // single-dimensional index.  The result is y * ROW_WIDTH + x.
  return y * GRID_WIDTH + x;
}



int IsInBounds( int x, int y )
{
  // Returns "true" if x and y are both in-bounds.
  if (x < 0 || x >= GRID_WIDTH)   return false;
  if (y < 0 || y >= GRID_HEIGHT)  return false;
  return true;
}
```

```
// This is the recursive function we will code in the next project
void Visit( int x, int y )
{
  // Starting at the given index, recursively visits every direction in a
  // randomized order.

  // Set my current location to be an empty passage.
  grid[ XYToIndex(x,y) ] = ' ';

  // Create an local array containing the 4 directions and shuffle their order.
  int dirs[4];
  dirs[0] = NORTH;
  dirs[1] = EAST;
  dirs[2] = SOUTH;
  dirs[3] = WEST;

  for (int i=0; i<4; ++i)
  {
    int r = rand() & 3;
    int temp = dirs[r];
    dirs[r]  = dirs[i];
    dirs[i]  = temp;
  }

  // Loop through every direction and attempt to Visit that direction.
  for (int i=0; i<4; ++i)
  {
    // dx,dy are offsets from current location.  Set them based
    // on the next direction I wish to try.
    int dx=0, dy=0;
    switch (dirs[i])
    {
      case NORTH: dy = -1; break;
      case SOUTH: dy =  1; break;
      case EAST:  dx =  1; break;
      case WEST:  dx = -1; break;
    }

    // Find the (x,y) coordinates of the grid cell 2 spots
    // away in the given direction.
    int x2 = x + (dx<<1);
    int y2 = y + (dy<<1);

    if (IsInBounds(x2,y2))
    {
      if (grid[ XYToIndex(x2,y2) ] == '#')
      {
        // (x2,y2) has not been visited yet... knock down the
        // wall between my current position and that position
        grid[ XYToIndex(x2-dx,y2-dy) ] = ' ';

        // Recursively Visit (x2,y2)
        Visit(x2,y2);
      }
    }
  }
}


void PrintGrid()
{
  // Displays the finished maze to the screen.
  for (int y=0; y<GRID_HEIGHT; ++y)
  {
    for (int x=0; x<GRID_WIDTH; ++x)
    {
      cout << grid[XYToIndex(x,y)];
    }
    cout << endl;
  }
}
```

Here is an example of a maze generated by the program:



Requirements

Convert the C++ program into an MIPS assembly language program using the following guidelines:

- I've included a skeleton code file (Project9.s) that will get you started. It has some TO DO comments; look for them to see what you have to add. In this project, you **DO NOT** have to implement Visit (we'll do that in Project 10), so your finished program should only print out a 79 x 23 grid of '#' characters, plus any testing output you included for the functions you coded.
- Use stack-based parameter passing. Additionally, use the stack to preserve registers. For example, to call the "XYToIndex" procedure with (4,2) the following code might be used:

```
      li      $t0, 4          # set x in $t0
      li      $t1, 2          # set y in $t1
              # remember, reverse order of parameters for C convention
      sw      $t0, -8($sp)    # push first param (x)
      sw      $t1, -4($sp)    # push second param (y)
      jal     XYToIndex       # call the procedure
      lw      $t0, 0($sp)     # get the returned value into $t0

        ...

XYToIndex:

      # save any registers we need to use.
      sw      $s0, -12($sp)   # will hold current x
      sw      $s1, -16($sp)   # will hold current y
      sw      $s2, -20($sp)   # loop counter
      sw      $s3, -24($sp)   # running index total
        ...
```

```
# and now we can grab the input parameters
lw      $s0, -8($sp)    # load x into $s0
lw      $s1, -4($sp)    # load y into $s1
  ...
        --- the rest of the procedure's code goes here ---
  ...
# save the return value
sw      $s3, 0($sp)     # return value was running total of index

# when done restore the registers
lw      $s0, -12($sp)   # put old $s0 back
lw      $s1, -16($sp)   # put old $s1 back
lw      $s2, -20($sp)   # put old $s2 back
lw      $s3, -24($sp)   # put old $s3 back

# then return to the caller
jr      $ra             # and return
```

- I've given you the code to set the grid in memory and to prompt for a random number seed. The code runs and if you stop it with a breakpoint before it exits, you can view the grid in memory.
- Notice that the procedures I gave you don't follow stack frame layout religiously (I've put some comments in discussing this); you don't need to either. Only Visit, which we aren't implementing in this project, requires the reentrant capabilities of a stack frame. But be sure your stack doesn't get messed up by your procedures or Visit will never work in the next project.
- Implement and test your program in stages to avoid getting overwhelmed by having to debug too much assembly at once. For example, start by defining the PrintGrid procedure and make sure you can print out a "maze" that's completely full of pound signs. Then tackle the remaining procedures one by one. Since they are all called by Visit, you will have to add some test calls to the main procedure to make sure they work correctly.
- **I need to see your testing from above.** So be sure your main code not only calls the procedures but also prints out the calling and return values. Then you can include a copy of your output in your report as proof of testing.
- You don't need to implement Visit in this project. Instead, just show me some of your testing for the other procedures; the way they should work is detailed in the function headers and you can also use the C++ code as a guide.

Project Report

The final step of this assignment is to create a report consisting of a cover page, an overview of the project, sample output, and the source code. See Assignment Policies on either the class website or Bb Learn.