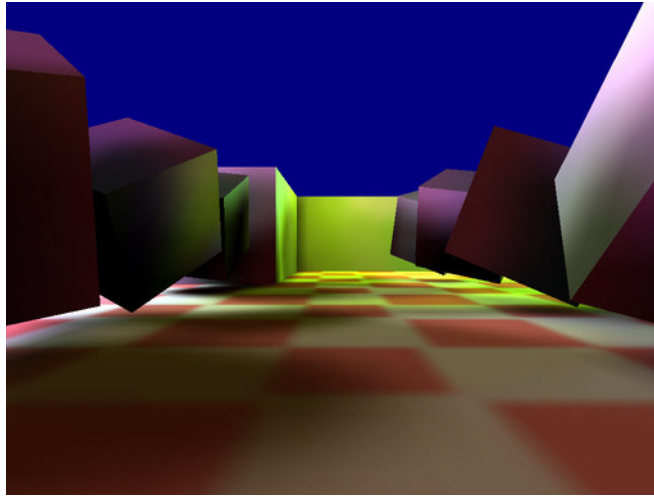


Lighting Scenes – A Simple Approach with Dynamic Results



Sometimes as developers we become impervious to our own creations. We look at the technicalities and miss what is before our very eyes. When I asked my wife to test the sample application for this tutorial one evening, she refused to continue within a minute or two. "Too scary, I'll have nightmares!" came the reply. "Success!" was my retort. This tutorial has no media whatsoever; it consists entirely of DarkBASIC Professional primitive objects, some simple texture generation and the power of Dark Lights to add the moody atmosphere.

This month, to illustrate the simplicity of Dark Lights, a maze will be built from primitive shapes. The resulting maze can be traversed in First Person view, by guiding the camera around with the arrow keys. There is as much, if not more information about creating the maze, as there is about lighting it. We will learn some simple yet useful techniques for creating a grid-based world in the first section. Dark Lights is so powerful that the addition of a small number of commands to this environment is all that is required to get the desired results.

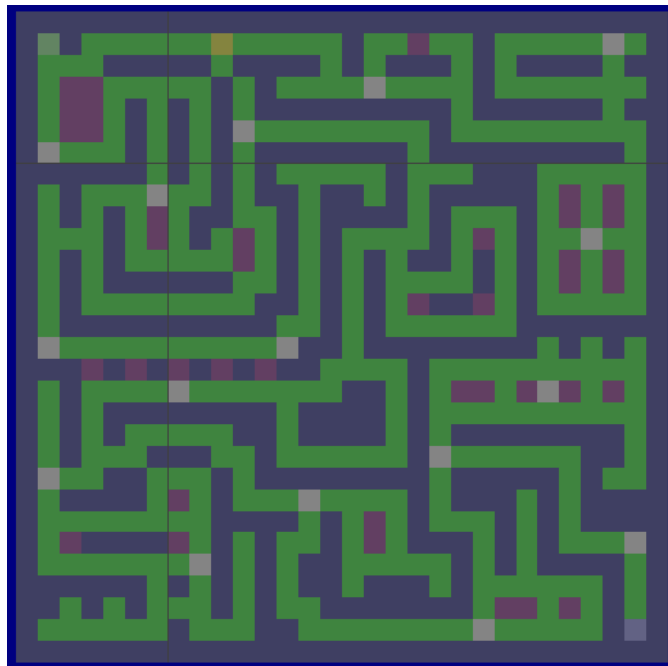
So let's get started.

Generating a Maze

The first step is to build a maze generating utility, or what is more commonly known as a map editor. For our purposes, this is a very simple grid-based system that allows us to:

- plot paths through the maze (spacebar)
- add light positions (L)
- the begin point (B)
- end point (E)
- random debris (D)
- and finally save the map as a text file (S).

You can investigate the source code of this editor, which is included in the download.



Building a maze from primitives

The starting point for our maze is to build the primitive objects as dictated by the map file. An array will hold the information about each square in our maze grid. Each array element will contain our essential data with the use of a type:

- X / Z grid coordinates
- X / Z positions in 3D space
- Whether the square is traversable, blocked or “Debris”
- Whether a light should be positioned at this point.

Once we have the array of information, we can convert it to a visible world. The maze file also holds the X and Z dimensions of the grid, and a very simple nested loop will enable the maze to be built easily:

```
For Z = 0 to (gridSizeZ - 1)
  For X = 0 to (GridSizeX - 1)
    square = (Z * GridSizeX) + X
    ` Create Maze Objects
  Next X
Next Z
```

This structure is the key to converting a grid to array elements and vice-versa. An array is essentially a long string of data, whereas a grid is 2-dimensional. This formula, as illustrated in the small code snippet above, is a highly efficient and simple way to convert from a grid reference to the associated array element. In this example, *square* is our array element.

For each square, we will build a cube if the coordinate is blocked, a random primitive if it is debris, or nothing for traversable squares. Debris squares are simply blocked coordinates, but allow us to generate an environment with more interest when lightmapped.

The final step in populating the maze is to add a floor, using a plain. A texture has also been generated by using coloured boxes drawn on a bitmap and then grabbed to an image. The image is scaled down to repeat many times over.

Moving around the maze

Movement is extremely simple. Every time we press the forward key, we move to the next square, and the down key moves backward. Left and Right keys rotate 90 degrees in the appropriate direction. We check if the next move is valid by moving a hidden sphere first and checking if a collision has occurred. If it does, we do not move the camera. If collision ever seemed like a daunting prospect, take a look at the simple function that does this for us:

```
function checkCollision(direction)

    position object cCOLLIDER, camera position x(), _
                        camera position y(), camera position z()
    yrotate object cCOLLIDER, camera angle y()

    move object cCOLLIDER, 10 * direction

    ret = 0
    if object collision(cCOLLIDER,0) > 0 then ret = 1

endfunction ret
```

The final line in this function simply returns a positive result if the collision between the hidden sphere and *anything* occurs.

Test Run

We already have everything in place to load, build and walk around our maze. Download, compile and run example 1 to see the code in action. Less than 200 lines of code generates a fully-functional maze, which will work with *any* design you create with the simple editor. Now that is incredible in itself, and we haven't explored lightmapping yet!

Let there be Light!

Let's progress from our dull, lifeless environment to something a little more inspiring. The framework of this process is to initiate the lightmapping functionality, add lights and obstacles, generate the texture and finally clear the data from memory. Here are the main commands we will use to build the lighting for the scene:

LM START: Start the Lightmapping functionality

LM Add Collision Object: Define an object as one which will block light

LM Add Light Map Object: Define an object as one which will be affected by the process

LM Add Point Light: Adds a point light to the scene

LM Build Collision: Prepares the object data for the generation process

LM Build Light Maps Thread: Generates the light map

LM reset: Clears the data after generation

The maze data has been used to define the positions of lights in the scene. Every primitive object making the maze has been added as a collision object, and also as a light map object to receive shadows and light. The floor is also added in the same fashion.

Compile and run example 2, to see the results of the lightmapping process. In this tutorial we will be generating the lighting on the fly each time, which is also appropriate because the maze map can be updated at any time. Dark Lights can also be used to create dbo models which can be loaded like any other model, with the previously generated lighting included. This is obviously a much faster way to include lighting in a game. The examples use the threaded technique of generating the map, allowing the progress to be displayed onscreen.

Refining the Results

The resulting scene is rather like an underexposed film. The darkness is just a little too dark, and we would need a great deal more lights to bring it up to an acceptable level. It's not always appropriate to add extra lights to compensate, especially when trying to replicate real lighting scenarios. One solution is to add an ambient light source, which works in a similar way to the ambient settings in DarkBASIC Professional; the greater the ambience, the brighter the scene. Ambience affects all objects:

```
LM Set Ambient Light 0.1, 0.1, 0.2
```

Compile and run example 3, to examine the results. The scene is now lit more brightly. However, it is still a little "flat", with unlit areas looking just like a standard scene with no lightmapping. We can fix this with another global lighting technique, which introduces direction:

```
LM Add Directional Light -45, -45, 10, 0.1, 0.1, 0.2
```

The first 3 parameters specify an X, Y and Z direction to the light, with the final 3 parameters defining the intensity of the colour channels. When you compile and run example 4, you will see a noticeable difference in the realism of the lighting. Walls cast shadows on one side of corridors, whilst the opposite sides are lit. It also adds a sense of direction, the shadows helping the players to orient themselves easily.

Out of the Shadows

In this tutorial we have taken a medialess, flat world and transformed it into a still medialess world but with so much more depth and immersion. The effort involved is minimal, and we have also proved that it is possible to even light unknown scenes, allowing massive potential for random and user-defined arenas. There is one final example included, which delves further into Dark Lights. This example is much more intensive, but includes:

- An optimised maze, with all hidden object faces removed. This leaves more room on the lightmap texture for finer detail where it can be seen.
- Transparent collision objects, with colour-changing light permeating through
- Smoothed sphere debris objects, using shadows to remove the polygon-constructed look.
- Area Lights (multiple point lights)
- Higher blurring and quality settings for a better finish.

Feel free to modify and experiment with the supplied code. The examples can be changed in many different ways, to create different effects and moods, from softly lit realism to hard-edged retro like visuals.

Until next time,
Happy Coding!

Steve Vink.