PART 13: “Data Structures”

Introduction

You reached episode 13 of the FASTTRACK tutorial series. If you got here in good shape, you learned a lot: you went though the basics of C programming, and got a taste of object oriented programming as well. In the meantime, you experimented with quite a few game related concepts. In the upcoming episodes you will further expand your knowledge, with more info on bit magic, file I/O, graphics programming and game development in general. But first: let’s get acquainted with the wonderful world of *data structures*.

The problem with arrays

Article 10 introduced you to arrays. These allow you to store a list of values:

```
int x[10], y[10];
float speed[10];
```

Obviously, you can conquer the world with arrays alone. However, sometimes they are not the most efficient solution to a problem. To illustrate this, let’s start with a small game of snake.

```
int x[8] = { 5, 6, 7, 8, 9, 10, 11, 12 };
int y[8] = { 8, 8, 8, 8, 8, 8, 8, 8 };
int heading = 0, delta[8] = { 1, 0, 0, 1, -1, 0, 0, -1 };
void Game::Tick( float deltaTime )
{
    screen->Clear( 0 );
    for ( int i = 0; i < 7; i++ )
    {
        x[i] = x[i + 1], y[i] = y[i + 1];
        screen->Box( x[i] * 8, y[i] * 8, x[i] * 8 + 6, y[i] * 8 + 6, 0xffffff );
    }
    x[7] += delta[heading * 2], y[7] += delta[heading * 2 + 1];
    if (GetAsyncKeyState( VK_LEFT )) heading = (heading + 3) % 4;
    if (GetAsyncKeyState( VK_RIGHT )) heading = (heading + 1) % 4;
    Sleep( 100 );
}
```

That’s pretty short, isn’t it? I do think it requires some explanation though.

The snake is represented by 8 squares, stored in arrays. Element 0 is the tail; 7 is the head of the snake. To move the snake forward, a heading is used: 0 is east, 1 is south, 2 is west and 3 is north. The snake thus starts by moving east.

All snake segments are supposed to follow the lead of the ‘head’ (element 7), so we replace the position of each segment by the position of the next segment. This happens in the `for` loop, which also draws the snake.

Next, the head is placed on a new position, based on the current heading. A little trick is used here: array `delta` contains eight numbers: each pair is a movement in x and y. So, looking at the first two elements, we conclude that `x[7]` is increased by '1' for 'east', and `y[7]` is increased by '0'.

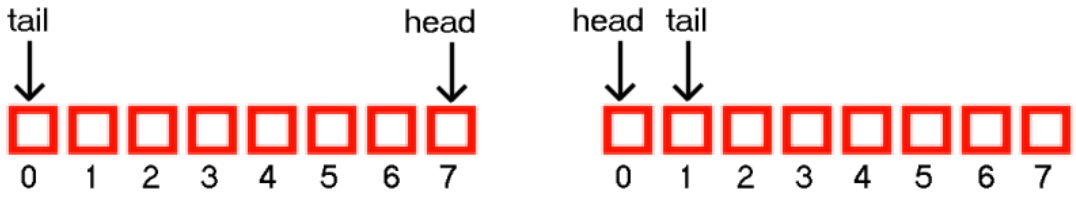
And finally, the heading is controlled by the player, using two `GetAsyncKeyState` commands.

This code is using arrays to store the snake. At first this appears to be a logical choice, until you start thinking big: what happens when the snake is hundreds of blocks long? Are you really going to copy every element to the next? That appears to be inefficient, because most of the snake is stationary: Only the head and the tail moves. The rest of the sliding snake body is merely suggested.

So what if…

Imagine this: rather than starting our array at 0 (tail) and ending it at 7 (head), we start at a random position. When the tail is at 0, the head is at 7. But when the tail is at 1, the head is at 0. And when the tail is at 2, the head is at 1. So, to get to the tail, you take the head, and proceed one element. If that takes you beyond the length of the array, you wrap around to 0.

Now *why* would you want to do that? Well… It saves you work! Or rather, it saves *your computer* work. Because, if your snake's tail *was* located at 0, and the head at 7, you could move the whole thing by making it start at 1. Like this:



All you need to do after this is setting he head at the correct new location, because this is now the only element with a wrong position: it's located where the tail was in the previous situation.

The code, with the modification:

```
int x[8] = { 5, 6, 7, 8, 9, 10, 11, 12 };
int y[8] = { 8, 8, 8, 8, 8, 8, 8, 8 };
int heading = 0, delta[8] = { 1, 0, 0, 1, -1, 0, 0, -1 };
int head = 7, tail = 0;
void Game::Tick( float deltaTime )
{
    screen->Clear( 0 );
    for ( int i = 0; i < 8; i++ )
        screen->Box( x[i]*8, y[i]*8, x[i]*8 + 6, y[i]*8 + 6, 0xffffff );
    int headx = x[head] + delta[heading * 2];
    int heady = y[head] + delta[heading * 2 + 1];
    tail = (tail + 1) % 8;
    head = (head + 1) % 8;
```

```
    x[head] = headx;
    y[head] = heady;
    if (GetAsyncKeyState( VK_LEFT )) heading = (heading + 3) % 4;
    if (GetAsyncKeyState( VK_RIGHT )) heading = (heading + 1) % 4;
    Sleep( 100 );
}
```

So is this better? The code grew a few lines, is incomprehensible, and not noticeably faster. Well, the answer is: yes. Instead of copying all the elements of the array to the next element, we do nothing. The only reason that for loop is still there is the fact that we still need to draw the snake. Or… do we?

Data structures

This was a small example of a smart trick to reduce the amount of work that your computer has to do. There are data structures for many things: for huge arrays that have just a few useful values, for searching though endless piles of values, for stacking values so that you can retrieve them in the opposite order, and so on. Often, a data structure is the key to game performance. You've just added the first of many to your arsenal.

Assignment

BASIC

1. Get rid of the for loop altogether. Just paint the tail in black, and the head at the new position. Note: you will not want to clear the screen for every frame.

2. Add food (blue square) at position (20, 15). When the snake 'eats' it, let the snake grow, and put new food at position (4, 4).

INTERMEDIATE

3. Add a boundary around the screen and check for collisions with this boundary. Also check for collisions between the head and the body of the snake. Hint: you can do this with a tile map, but it's perhaps easier to just read the color of a pixel at the new position of the head.

ADVANCED

4. Complete the game: print the current score, add new food at a random position whenever it has been eaten, restart the game when the snake dies.

END OF PART 13