Game Programming Genesis Part IX : Adding Characters

by Joseph "Ironblayde" Farrell

Introduction

So you've got the beginnings of a tile engine. You've got your world scrolling across the screen in 16-bit color, complete with animations and anything else you decided to add on your own... but it's a pretty lonely place so far, isn't it? Today we're going to fix that problem by going over an easy way to implement the game's main character, along with some NPCs to keep him company. I'll show you how to get your world scrolling around the character so he stays in the middle of the screen, how to configure NPCs, and how to work them into the function you already have for drawing the map. When all is said and done, we'll have a nice start on an RPG (or other tile-based game) that is easily expanded upon. As usual, I'll have a complete code example for you to play with.

Throughout this article I'll be referring back to the basic tile engine we developed back in article #8, so if you haven't read it, and you haven't created a tile engine of your own before, go back and look over article #8 so you can follow along. Aside from that, all you need is a working knowledge of DirectDraw, the all-powerful Visual C++ (or one of those other ones), and some artwork to use. Ready to breathe a little life into your game world? Let's go!

Keeping Track of Characters

As usual, before we can go about getting something to move around on screen the way we want it to, we have to come up with a logical way to represent it in memory. Naturally, player characters and NPCs (non-player characters) are going to be represented somewhat differently. For instance, NPCs will need some flags or even a script assigned to them that describes how they move and act, whereas player characters do not, because they will be controlled by whoever's playing your game. Similarly, unless you're making your NPCs very complicated, they won't need a lot of the details that go into your player characters. For instance, non-combative NPCs don't need a number of hit points, a list of available spells, or a value for dexterity in a fight.

This leads to a little bit of a problem: what happens when we want to start moving NPCs and player characters around? Are we going to have to write one set of functions for player characters, and another set for NPCs? Obviously we don't want to do that. You C++ people should be thinking about a `Character` base class right now, and child classes called `Player` and `NPC`, or something similar. For those of us using C, we can do much the same thing by nesting a structure. That is, we'll create a `struct` for player characters and a separate one for NPCs, but each one will include a common `struct` that contains movement data. This might be a little unclear right now, so let me show you an example of such a structure, and we'll take it from there.

```
typedef struct CHARMOVE_type
{
  int xTile, yTile;        // character's current location
  int xOffset, yOffset;    // offset from tile location in pixels
  int xMove, yMove;        // number of tiles character is currently moving
  int nFace;               // direction character is facing; takes FACE_ constants
} CHARMOVE, FAR* LPCHARMOVE;

#define FACE_SOUTH 0
#define FACE_NORTH 3
#define FACE_EAST  6
#define FACE_WEST  9
```

All right, so what is all this stuff?

`int xTile, yTile`: As you've probably guessed, this will be the character's current location in tiles. These values will always be at least zero, and less than the size of the map in tiles. Furthermore, when a character is in the process of moving from one tile to another, these variables will be the coordinates of the tile the character is moving towards, not the tile the character is coming from. The reason for this will become clear later.

`int xOffset, yOffset`: Characters won't always be standing directly on top of a tile. When they're in the middle of moving from one to another, we'll need these offsets to track their exact location in pixels.

`int xMove, yMove`: These values are the number of tiles the character is set to move. For player characters these will always be -1, 0, or 1 (depending on direction); but for NPCs, whose every movement is not being controlled by an input device, the program may want to make them move greater distances at once. If xMove and yMove are both 0, the character is standing still. If xMove and yMove are both nonzero, the character is moving diagonally somehow. This will only happen if you allow eight-directional movement in your game. I didn't do that in Terran simply because there's no way in hell I can draw a half-decent character as viewed from a 45-degree angle. :)

`int nFace`: This simply keeps track of which direction the character is facing. Why, you ask, did I use 0, 3, 6, and 9 for the direction values instead of 0, 1, 2, and 3? Well, you'll understand when you see how I've set up the character image files we'll be using.

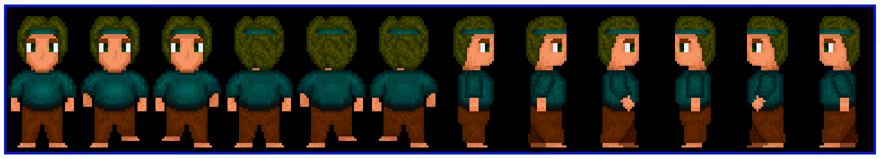
From here we might want to get started on structures for the whole player characters and NPCs, but I'm going to largely leave that out for now because all we'll really be dealing with is movement. The player structure would normally have all sorts of things in it like what level the character is on, what items he's carrying, what spells he can cast, and all kinds of other things. The NPC structure should have flags that determine how NPCs will be controlled. For now, I'm just going to leave those out and we'll use some kind of default values. Once we learn how to manipulate that, I'll show you what you can do to individualize your NPCs a bit. But for now, we'll just use these rather pointless structures:

```
typedef struct PLAYER_type
{
  CHARMOVE move;
} PLAYER, FAR* LPPLAYER;

typedef struct NPC_type
{
  CHARMOVE move;
} NPC, FAR* LPNPC;
```

Now that we've got two structures serving no purpose other than to differentiate between a player and non-player character, let's go on to seeing just how we get them moving around in our game.

Character Image Files

To move characters around, I usually suggest a minimum of 12 frames. There are four cardinal directions, and in each direction, there should be one frame of the character standing still (facing that direction), one of him in mid-stride with left foot forward, and one with the right foot forward. That's what I'm using in Terran, and in each character image file, the frames are arranged in a vertical line like you see below. Notice that the leftmost character (frame #0) is facing south, frame #3 faces north, frame #6 faces east, and frame #9 faces west. Now do you see why I assigned the FACE_ constants the values that I did? To find out which frame to use for a character standing still, you just use the move.nFace variable. To get the frame with his right foot forward, you use move.nFace +1. And to get the frame with his left foot forward, you just use move.nFace +2. Convenient, hey? :)



This character, by the way, is one of my current main character graphics from Terran. It'll probably change before the final game comes out. But my point in bringing this up is that all the graphics included for the demo that come with this article are pulled directly from my game; they are there for you to experiment with until you can come up with your own stuff, but when you do end up making a game and distributing it, please don't use my art, since it may appear in Terran as well. I know there's someone out there saying, "But Blayde, your art is so terrible, why would anyone want to steal it?" Well... you're probably right, but still, I thought I'd mention it.

Anyway, matters of copyright aside, this is a convenient way to set up your image files and I recommend you use it, or something like it. Vertical lines work just as well as horizontal ones do; it's just a matter of preference. For my game and for our demo, each character fits inside a 32x64 box, so each character image should be 384x64, and you can easily locate a RECT containing the correct frame by doing this:

```
    RECT rcChar = {0, nFrame << 5, 64, (nFrame << 5) + 32};
```

The nFrame variable, in this case, is simply the number of the frame you're looking for. We'll do something like this when we get to actually blitting the characters onto the screen. Notice, though, that I have the y-values permanently set to 0 and 64. This only works if you have a separate surface for each character. Generally I don't like to do that; I have one large surface with all my characters on it. So then you'll need some way of computing the y-values. Usually I do this by including a value in the NPC or player character structure that says which "slot" on the surface that character's image is loaded into.

Handling the Camera

Remember back in article #8 when we set up the camera to define what part of the map was being drawn in each frame? Well, now we have to change it just a little. Last time, we set the camera so that it shifted in the appopriate direction anytime the user pressed the arrow keys. But in an actual game, chances are that you're not controlling the camera directly with the arrow keys; rather, you are controlling the character, and the camera follows the character.

The simple method we used in that demo would be fine if the character was always going to be centered on the screen, but generally that is not the case. What happens when the character approaches the very edge of a map? If we centered the camera on him, then there would be an area of the screen with nothing displayed on it, and we don't want that. I'm sure you know what I mean; you've probably played a game that has the character centered on the screen, except when he gets near the edge of a map, at which point the camera remains still and the character moves towards the edge of the screen as well. That way, the whole screen is always occupied by the current map.

So how do we implement this? Actually, it's quite easy. Instead of changing the camera every time the arrow keys are pressed, we'll calculate the camera coordinates during every frame, based on where the character is positioned in the world. Since our screen is 640 pixels wide and our character is 32 pixels wide, we can center the camera horizontally by locating the camera (640 - 32) / 2 = 304 pixels to the left of the character. Similarly, the screen is 480 pixels tall and our character is 64 pixels tall, so the camera should be placed (480 - 64) / 2 = 208 pixels above the character to center it vertically. Then to make sure the camera stops when the character gets near the edge of the screen, all we have to do is make sure the camera's coordinates don't drop below (0, 0), or go above the maximum allowable camera coordinates as defined in the MAPDATA structure we created in article #8. It's that easy! So we can get the camera to follow our character with this simple code:

```
// center camera on main character
mapdata.xCamera = (player.move.xTile<<5) + player.move.xOffset - 304;
mapdata.yCamera = (player.move.yTile<<5) + player.move.yOffset - 240;

// clip camera to map boundaries
if (mapdata.xCamera < 0)
    mapdata.xCamera = 0;
if (mapdata.yCamera < 0)
    mapdata.yCamera = 0;
if (mapdata.xCamera > mapdata.xMaxCamera)
    mapdata.xCamera = mapdata.xMaxCamera;
if (mapdata.yCamera > mapdata.yMaxCamera)
    mapdata.yCamera = mapdata.yMaxCamera;
```

It's worth mentioning here that this code should probably be included somewhere other than the map rendering function, because you may not always want to use it. For instance, in game story scenes, you'll almost certainly want to control the camera using a script instead of simply setting it to follow the character. If you think back to the logic diagram I showed you for Terran in article #7, this camera-control code is only used while the World Map game state is active. Story scenes, on the other hand, occur under the Scripts Only state, and so whatever scripts happen to be running at the time can do what they like with the camera.

Checking For Movement

Now we have everything we need in terms of organization and setup, and we're ready to create some sort of mechanism for controlling the character. Since we haven't covered DirectInput and creating a universal input system from multiple devices (such as keyboard and gamepad), we'll just use the Win32 API function `GetAsyncKeyState()`, which we met back in article #3. As you may recall, we wrapped the function in a macro which masks out the bit we're interested in. Here's the macro again, to refresh your memory:

```
#define KEYSTATE(n) ((GetAsyncKeyState(n) & 0x8000) ? TRUE: FALSE)
```

Now what we need is for the character to start moving in the specified direction as soon as an arrow key is pressed. How do we do that? Well, thinking back to the `CHARMOVE` structure we just set up, there were two member variables called `xMove` and `yMove`. For now, all we'll do is set those to their correct values based on the key being pressed, and we'll see how to handle the data when we write the actual animation code in a bit. So, all we do is check each arrow key. If it's being pressed, and the character is not already moving, then we set `xMove` or `yMove` appropriately. The other thing we need to do is to adjust `xTile` and `xOffset`, or `yTile` and `yOffset`, to reflect the character's new location. We don't want the character's position to change suddenly, so when we increase a tile setting by 1, we decrease the corresponding offset by 32 (our tile size). In this way, the character's position remains constant, but represented in a different way, so that the animation code knows to do something with it. Finally, we update nFace so the character turns to face the right direction. In this code, and for the rest of the article, I'll assume we have a PLAYER structure set up called player.

```
if (KEYSTATE(VK_UP) && (player.move.xMove == 0) &&
                       (player.move.yMove == 0))
{
  player.move.yMove = -1;
  player.move.yTile--;
  player.move.yOffset += 32;
  player.move.nFace = FACE_NORTH;
}
if (KEYSTATE(VK_DOWN) && (player.move.xMove == 0) &&
                         (player.move.yMove == 0))
{
  player.move.yMove = 1;
  player.move.yTile++;
  player.move.yOffset -= 32;
  player.move.nFace = FACE_SOUTH;
}
if (KEYSTATE(VK_LEFT) && (player.move.xMove == 0) &&
                         (player.move.yMove == 0))
{
  player.move.xMove = -1;
  player.move.xTile--;
  player.move.xOffset += 32;
  player.move.nFace = FACE_WEST;
}
if (KEYSTATE(VK_RIGHT) && (player.move.xMove == 0) &&
                          (player.move.yMove == 0))
{
  player.move.xMove = 1;
  player.move.xTile++;
  player.move.xOffset -= 32;
  player.move.nFace = FACE_EAST;
}
```

That's not so bad, right? Just a few simple lines. Consider this for a minute: what would happen if we didn't check whether or not the player was already moving? And what would happen if, when the player pressed up or down, we only checked vertical movement, not horizontal, before assigning a value to yMove (and similarly for when right or left was pressed)?

The answer to the first question is that keystrokes would build up in a sort of buffered fashion... if your game runs at 30 FPS, and the player were to hold down the up key for one second, the player would move 30 spaces to the north. That's obviously not what we want!

The answer to the second question is something more useful: it would yield eight-directional movement instead of our four-directional system. However, it would allow the character to start moving vertically in the middle of moving horizontally, which means the end destination would not be centered on a tile. This means we'd have a pixel-by-pixel scrolling game instead of pixel-by-tile, which is a bit harder to handle; I'll explain the difference once we come to animation.

Before we get any character animations onscreen, there's one more thing we have to consider. Our movement code as it stands right now will work, but it will let the character go anywhere, even off the edge of the map! And what's the point of having a map if the character can just walk through everything? What we need is a function that performs a series of checks on the intended destination, and tell us whether or not it's allowable to move there.

Collision Detection

Collision detection in a tile engine like this one is so simple it hardly deserves such an important-sounding name. All we have to do is look at the map at the place to which we want to move. If there's an object there, movement is not allowed. If the spot is open, movement is OK. How hard can that be? Basically there are three things we must check:

1. Has the character reached the boundary of the map?
2. Is the character attempting to walk into a solid object on the map?
3. Is the character attempting to walk into another character?

The first one is a complete triviality. The second and third and pretty easy as well, as long as we have a way to keep track of that information, which we do! For detecting objects on the map, we have only to look at our tileset, which is represented by the `TILE` structure we set up back in article #8. If you remember, each `TILE` has a value called `bWalkOK`, which is `TRUE` if the tile can be walked on, or `FALSE` for solid objects. And remember that we check layer 1 of the map, not layer 2, because layer 2 is the foreground. If we were employing a three-layer system like I use in Terran, then what I do is to check layer 2 first. If layer 2 has some graphic in it, that tile determines whether or not the space can be walked on. If layer 2 is empty there, I check layer 1.

For checking characters, we look at the main character and all the NPCs, to see if any of them are standing on that tile. This is why, when a character begins moving, we change their tile location immediately instead of waiting until movement is done. This way, walking onto a tile that another character is moving towards is not allowed, whereas walking onto a tile that another character is currently walking away from is all right. Now all we need is an array of NPCs so we can keep track of them:

```
LPNPC lpnpc[100];
```

I've allowed for a lot of NPCs here; change the array size to match what you're going to need. Or you can always use a linked list. Remember that we've got a data member in our map structure to keep track of NPCs and we could use that too; I'm declaring it here separately to keep things a little clearer. Finally, you'll notice that I've declared pointers to structures instead of structures themselves. Why would I do that? Well, it's because we're going to have to sort NPCs later on, and it's faster to swap pointers than to swap entire functions.

Anyway, now we can write a function that checks the validity of a space. We'll pass it an x and y coordinate for the space to check, and have it return `TRUE` or `FALSE`. Here she is:

```
int MoveOK(int x, int y)
{
  int z;

  // first check the map boundaries
  if ((x < 0) || (y < 0) || (x > mapdata.xMax) || (y > mapdata.yMax))
    return(FALSE);

  // now check if the main character is already there
  if ((x == player.move.xTile) && (y == player.move.yTile))
```

```
        return(FALSE);

      // check all the NPCs
      for (z=0; z<mapdata.nNPCCount; z++)
      {
        if ((x == lpnpc[z]->move.xTile) && (y == lpnpc[z]->move.yTile))
          return(FALSE);
      }

      // the tile itself is now the deciding factor, so just return that
      return(tileData[byMap[x][y][0]].bWalkOK);
    }
```

If you're wondering where `mapdata` and `tileData` came from, go back and look over article #8. These variables were defined in the code example that came with that article; we're just continuing the example this time around. Now, the only thing you might be wondering about is, if we're using this function to see if the player can move somewhere, why on earth would we check if the player is already standing there? Well, we'll also be utilizing this function to determine movement for NPCs, so it's necessary.

Now all we have to do is add a call to `MoveOK()` in our previous code that checks for whether the player is moving the character around or not. I won't repeat the whole code block here, but here's what the code for moving in the -y direction would look like:

```
    if (KEYSTATE(VK_UP) && (player.move.xMove == 0) &&
                           (player.move.yMove == 0) &&
                           MoveOK(player.move.xTile, player.move.yTile - 1))
    {
      player.move.yMove = -1;
      player.move.yTile--;
      player.move.yOffset += 32;
    }
```

And we're all set! Now the game responds to keypresses by first checking whether or not a player is allowed to move in that particular direction, then changes the player's location settings if appropriate. The only thing left is the character's visual representation on the screen, and so we move to animation.

Animation

The first thing we should do is to think about what we want to happen when the player presses an arrow key. You may be thinking, "Oh, that's simple: the character should walk from one tile to the next," but that is not always the case. In pixel by tile (PxT) scrolling, this is what happens. As any movement key is pressed, the character walks from one tile to the next. An alternate method is called pixel by pixel (PxP) scrolling, and allows you a bit more freedom, because as soon as the player releases the arrow key, the character stops moving, even if he's between two tiles. The good part about PxP scrolling is that it can take away from the feeling that your character is locked into a grid, especially if you also employ eight-directional movement. The bad part is that it makes collision detection a bit tricky. I'm not going to cover PxP scrolling here, since it's a good idea to get PxT up and running first so you can get a feel for creating a simple game engine, but think about it... it does require a bit more thought, but once you've come up with a good way to handle it, it's easily worth a little extra effort.

All right, so we're going to move one tile at a time. Our animation code checks if animation is necessary by simply looking at the `xMove` or `yMove` member of a character's `CHARMOVE` structure. If it is nonzero, the character is moving. So what do we do from there? Well, remember that at the beginning of an animation, the character is standing on one tile, but his location variables say he's on the next. This is possible because one of the character's offsets is set to 32 or -32. So for each frame of animation, we gradually move the offset back towards zero, and as a result, the character will move towards his correct location. It's really that simple.

The only thing that even deserves any thought is to consider which frames are displayed when. We have three frames of animation for each direction: standing still, right foot forward, and left foot forward. This animation should play in the sequence: 0, 1, 0, 2, 0, 1, 0, 2, etc. So it doesn't look like he's taking ridiculously small steps, we'll say that a movement from one tile to the next is done in four steps: 0 to 1, 1 to 0, 0 to 2, 2 to 0. This will actually appear as two full steps on the screen. To set a reasonable character speed, let's shift the character by four pixels every frame. So a movement from one tile to the next will look like this:

Game Frame	Offset (absolute value)	Animation Frame
0	32	0
1	28	0
2	24	1
3	20	1
4	16	0
5	12	0
6	8	2
7	4	2
8	0	0

Hopefully this will clear things up a bit. The leftmost column is simply a numbering system for game frames -- these are happening at about 30 per second. The middle column represents `player.move.xOffset` for horizontal movement, or `player.move.yOffset` for vertical. This is an absolute value because it may in fact range from -32 to 0, for instance if the character is walking up instead of down. The important thing to note is that this gradually approaches 0. This represents the character's actual movement onscreen, since `xOffset` and `yOffset`, as you should remember, are used in the calculation for where the player appears on the map. When the offset reaches 0, the appropriate movement variable -- that is, `xMove` or `yMove`, is reset to 0, and the character stops. The final column represents the animation frame to display. Note that it goes in the sequence we decided on earlier. We will add this number to `player.move.nFace` to get the actual frame number within the image file. Remember, that's why we used 0, 3, 6, and 9 for the movement directions.

To further demystify things, let's get some code down that implements the table above. What needs to happen? First, the code should check if movement in a given direction is occurring. Second, the offset should be updated. Third, the code should check to see if the character has finished moving from one tile to the next. If so, reset the movement variable. We could write this in two cases, one for horizontal movement and one for vertical, but just in the interest of keeping things easy to understand, I'm going to split it into four instead, one for each direction. I'm also going to replace player.move with simply move, because eventually this will be a function that gets used for players and NPCs. Let's see how it goes.

```
// first check for movement south
if (move.yMove > 0)
{
  move.yOffset += 4;
  if (move.yOffset == 0)
```

```
        move.yMove = 0;
    }

    // now check north
    if (move.yMove < 0)
    {
      move.yOffset -= 4;
      if (move.yOffset == 0)
        move.yMove = 0;
    }

    // check movement east
    if (move.xMove > 0)
    {
      move.xOffset += 4;
      if (move.xOffset == 0)
        move.xMove = 0;
    }

    // check movement west
    if (move.xMove < 0)
    {
      move.xOffset -= 4;
      if (move.xOffset == 0)
        move.xMove = 0;
    }
```

This takes care of moving the player around from step to step, and so there's only one more step remaining: we still have to draw the player on the screen. The only thing that's not obvious about this is how to find the coordinates at which to draw the player. After all, he won't always be at the center of the screen. The answer is simply to calculate his coordinates in terms of world coordinates, and then subtract the camera coordinates from them to get the final rendering position. Then we just get the correct frame like we discussed earlier, and voila, we've finally got what we want! Here's the code:

```
    // set up a default source RECT
    RECT rcSrc = {0, 0, 32, 64}, rcDest;
    int nFrame;

    // first figure out which frame to use
    nFrame = move.nFace;
    if (((move.xMove) && ((abs(move.xOffset) == 4) || (abs(move.xOffset) == 8))) ||
        ((move.yMove) && ((abs(move.yOffset) == 4) || (abs(move.yOffset) == 8))))
        nFrame++;
    if (((move.xMove) && ((abs(move.xOffset) == 20) || (abs(move.xOffset) == 24))) ||
        ((move.yMove) && ((abs(move.yOffset) == 20) || (abs(move.yOffset) == 24))))
        nFrame += 2;

    // update rcSrc to correct frame
    rcSrc.left += (nFrame<<5);
    rcSrc.right += (nFrame<<);

    // set up rcDest
    rcDest.left = (move.xTile<<5) + move.xOffset - mapdata.xCamera;
    rcDest.top =  (move.yTile<<5) + move.yOffset - mapdata.yCamera - 32;
    rcDest.right = rcDest.left + 32;
    rcDest.bottom = rcDest.top + 64;
    // blit sprite
    lpddsBack->Blt(&rcDest, lpddsCharacters, &rcSrc, DDBLT_WAIT | DDBLT_KEYSRC, NULL);
```

The only part of this code that might strike you as odd is that I'm subtracting 32 from the y-value of the character. Remember, that's because we calculated the screen location of the tile on which the character is standing... but the character is two tiles tall, so we need to plot him one tile above his actual location to make sure he's standing in the right place.

Now the character responds to arrow keypresses by walking in the appropriate direction. The animation follows exactly what we set up, and the map scrolls to center the character when he's not near the map's edge. About bloody time, hey? :) He's probably going to get tired of wandering around all by himself though, so let's throw some NPCs in there. We've actually done most of the work already...

NPCs and Random Movement

In a really well-done game, NPCs will probably be doing something suitable to who they are. Old men and women won't move around much, kids will run through the town playing, and thieves will quickly walk away whenever they notice anyone paying too much attention. But to do all that, we either need to hard-code a bunch of behavior for each NPC type (which is not a good idea), or use a script (which we don't know how to do yet). So for now, let's start off our NPCs by implementing the simplest form of behavior that's found in lots of old RPGs -- random movement.

The nice part about implementing NPCs is that we can use a lot of the same code that we used for setting up the player. Our function that checks whether or not a character can move to a certain position can be used for NPCs, and the code we just wrote for actually plotting characters can also be used for NPCs, with one minor change. We'd just have to stick it in a function that takes one argument: the y-value to use for the source RECT. Hence we will have multiple character images loaded onto a single surface, and specify which character to use via this parameter.

So what's left to do? First we'll need to set up some sort of system that determines random movements for the NPC. The simplest way to do it is just to choose a random number, and if it falls in a certain range, or is divisible by a certain number, then the character should move, so we pick a random direction and that's it! The code would look something like this:

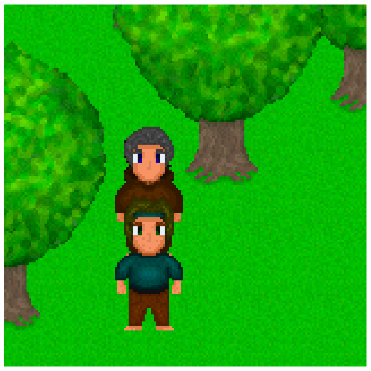
```
    // choose a random number
    int nDir;
    int nRand = rand();

    // move if 128 divides nRand (bits 0-6 used) and NPC is still
    if (((nRand & 0x0000007F) == 0) && (move.xMove == 0) && (move.yMove == 0))
    {
      // use bit 7 to determine positive (south, east) or negative (north, west) movement
      nDir = ((nRand & 0x00000080) >> 6) - 1;

      // use bit 8 to determine horizontal or vertical movement
      if ((nRand & 0x00000100) == 0)
      {
        // move vertically
        if (MoveOK(move.xTile, move.yTile + nDir))
        {
          move.yMove = nDir;
          move.yTile += nDir;
          move.yOffset -= (nDir << 5);  // remember, this is multiplication by 32
          move.nFace = ((nDir == 1) ? 0 : 3);
        }
      }
      else
      {
        // move horizontally
        if (MoveOK(move.xTile + nDir, move.yTile))
        {
          move.xMove = nDir;
          move.xTile += nDir;
          move.xOffset -= (nDir << 5);
          move.nFace = ((nDir == 1) ? 6 : 9);
        }
      }
    }
```

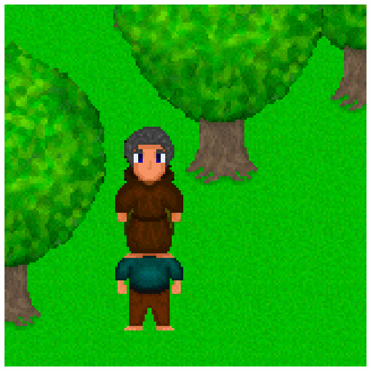
All right, I'm using some pretty odd-looking code in there to people who don't use bitwise operations a lot, so I'll clear it up a little. The bitwise AND operator & can be used to extract bits from a number. That's what I'm doing here -- essentially extracting three random numbers from one. In the case of `nDir`, I mask out a single bit: bit 7. So the result of the & operation is either 2^7 * 1 = 128 or 2^7 * 0 = 0. Shifting by six places is equivalent to dividing by 64, so the result is now either 2 or 0. Then I subtract one to get either 1 or -1. Cool, hey? You can do it more straightforwardly if you want; I use bitwise operators and shifts a lot so that looks pretty natural to me, but do what works for you.

Anyway, this is all it takes to get NPCs moving around, and we already said that we could use our player drawing function to draw NPCs as well. So we're finished, right? Well, not quite.

Ordering Characters

As soon as we introduce a second character into the game, we create a new problem for ourselves: the characters must be drawn in the correct order. When two characters who are two tiles tall come within one tile of each other, the one with the lesser y-value had better be drawn first, or the results will look a little strange!


Right


Wrong

See what I mean? Ordering is the difference between having your characters appearing as they would naturally, or having an old NPC standing on the shoulders of a headless main character. :) What we need to do is make sure that our NPCs are always sorted in order of increasing y-value, so that they get drawn properly. Also, we need to be watching for when the main character should be drawn, because he's part of the order too.

The first thought that comes to mind is a sorting function -- but do you really want to run a full sort every time an NPC moves? If you've got a hundred NPCs, that's going to be a large sort running way too often. What we do instead is to sort the NPCs once, right when they're initialized. Then, anytime an NPC's `yTile` member changes, we move that element only. Since yTile only changes by 1 at a time, we won't need to run a sort on the whole NPC array. We'll just find the NPC with the next smallest (or next largest, as the case may be) `yTile`, and swap those two. Much faster. This is also why we created the NPCs as pointers instead of structures. Swapping pointers is faster than swapping the contents of an entire structure.

For that initial sort that runs right when the NPCs are initialized, you can use whatever you like. If the number of NPCs is going to be small, then something like a bubble sort or insertion sort will serve your purposes just fine. If your vision is slightly larger and involves lots of characters all going at once, then you may want to use something a little more high-powered, like QuickSort. The code to keep the array sorted is no problem. We need two cases: one in which the NPC has moved up, and one for if he's moved down. All we do is find the nearest NPC whose y-value is lesser or greater, respectively, then swap the current NPC to be next to that position. Here it is in action for when NPC number `nIndex` moves upwards:

```
int nPointer = nIndex;
LPNPC lpnpcTemp;

// first check to see if this is equal to the highest NPC
if (lpnpc[nIndex]->move.yTile > lpnpc[0]->move.yTile)
{
  // nIndex is not highest, so a reordering is required.
  // step up the array until we find the next highest NPC
  while (lpnpc[nPointer - 1]->move.yTile == lpnpc[nIndex]->move.yTile)
    nPointer--;

  // now swap NPCs at nIndex and nPointer
  lpnpcTemp = lpnpc[nPointer];
  lpnpc[nPointer] = lpnpc[nIndex];
  lpnpc[nIndex] = lpnpcTemp;
}
else
{
  // swap with highest NPC
  lpnpcTemp = lpnpc[0];
  lpnpc[0] = lpnpc[nIndex];
  lpnpc[nIndex] = lpnpcTemp;
}
```

The code for moving down is analogous so I won't bother showing it here. Note that this code might cause some ugly errors if you run it when the NPC array is not sorted, so be careful to implement it correctly!

Well we're just about done here! Our NPCs now move randomly, and the array of NPCs will always be ordered correctly, with minimal work required to keep it that way. The only thing left to do is to render the characters onto the map. Here's how it's done. Immediately after the map itself has been drawn, start searching through the NPC array for the NPC with the smallest y-value who is in the viewable vertical range. This can be done simply by checking the NPC's location against the camera's y-coordinate. From that point, start going through the NPC array one at a time, drawing each character if he is within the viewable horizontal range. Remember that the player must also be drawn at the correct time, so watch the NPC y-values and how they compare to the player character's y-value. The source code uses some variables that are found in the function for rendering maps, so I'll just let you look at the sample source file for this one. Look at how it's set up in `RenderMap()`, and how it's executed in `RenderCharacters()`.

Closing

This demo of ours is starting to look more and more like a working game, isn't it? Go grab the source code for this article and play with it a little bit. See what you can write on your own without having to look at the source. And then see what you can add to it! Give your NPCs a little bit more freedom by letting them vary from each other a bit more. By adding some data members to the NPC structure, you can make different NPCs walk at different speeds, and more or less often than others. You can allow NPCs to move more than one tile at a time. If you're feeling really ambitious, try giving them some paths to follow. If you visit the GameDev Game Design forum at all, you've probably heard the phrase "NPCs are people too." So don't subject your characters to a lifetime of wandering randomly if you can come up with some better ideas. :)

Well I'm sad to say it, but Game Programming Genesis is just about at an end. I had planned on extending the series to 12 articles and adding a couple more topics, but there are a few reasons for stopping at Part X. For one, my life has gotten a **lot** crazier in the last week or two, so I've gotten very pressed for time, and these articles take a lot of time to come up with, especially with the demos getting larger all the time. I really wanted to get to scripting, but I realized that it's such a huge topic, I'd have to devote an entire series to it. There's no way I could cover everything I want to say about it in one article. So a new series dedicated to building up this scripting engine I'm always talking about is a real possibility for the not-too-distant future if I get the time to do it.

Next time, in the series' final entry, I'll show you all kinds of assorted tips and tricks for you to implement as you start building our demo into a full game. It'll be a little bit of information on a lot of topics, like generating log files of your programs at runtime, protecting your game data from being changed by end users, and whatever else I come up with in the next few weeks. As always, feel free to E-mail me at ironblayde@aeon-software.com, or find me on ICQ at #53210499, with any questions you have, and I'll see you next time!

Discuss this article in the forums