Notes 21: Tile Games

Organizing the game space into tiles can have many benefits including reducing the number of needed collision checks, easier programming logic, easier maze creation, and enabling path finding algorithms.

Let’s start first with creating a simple grid of tiles. The code below can be found in file:

(a_tile_1.fla)

```actionscript
var depth:Number = 100;
var numCols = 6;
var numRows = 6;
var tileSide = 40;
var tempMC:MovieClip;

for (var i = 0 ; i < numCols ; i++)
    for (var j = 0 ; j < numRows ; j++) {
        tempMC = attachMovie("greenTile","tile_"+i+"_"+j,depth++);
        tempMC._width = tileSide;
        tempMC._height = tileSide;
        tempMC._x = i * tileSide;
        tempMC._y = j * tileSide;
    }
```

Not a MovieClip called named “greenTile” can be found in the library. The doubly nested loop is used to create the grid by attaching tiles. The width/height of the MovieClips are set to the contents of variable “tileSide”.

One obvious use of a board of tiles would be to create a chessboard. The above loop can be modified (a_tile_2.fla) to alternate the tile color:

```actionscript
for (var i = 0 ; i < numCols ; i++)
    for (var j = 0 ; j < numRows ; j++) {
        if ( ((i+j) % 2) == 0 )
            tempMC = attachMovie("greenTile","tile_"+i+"_"+j,depth++);
        else
            tempMC = attachMovie("blackTile","tile_"+i+"_"+j,depth++);
    }
```
The use of tiles can be made easier by creating a tile class and then creating tile objects. One possible example is in the SquareTile class below. The SquareTile class constructor requires the x,y coordinates, side length, and library movie clip name. The class is used by the code in a_tile_3.fla as follows:

```javascript
for (var i = 0 ; i < numCols ; i++)
for (var j = 0 ; j < numRows ; j++) {
  if ( (i+j) % 2 == 0 )
    tempTile = new SquareTile(i*tileSide, j*tileSide, tileSide, "greenTile") ;
  else
    tempTile = new SquareTile(i*tileSide, j*tileSide, tileSide, "blackTile") ;
}
```

The example code is in the two files:

(a_tile_3.fla)
(SquareTile.as)

class SquareTile {
    private var x:Number ;
    private var y:Number ; // current x,y location
    private var width:Number ;
    private var height:Number ;
    private var sideLength:Number ;
    private var tileType:String ;
    private var internalName:String ;
    private var depth:Number ;
    private var mc:MovieClip ;

    // constructor
    function SquareTile(x:Number, y:Number, sideLength:Number, linkName:String) {
        var nextDepth:Number = _root.getNextHighestDepth() ;
        internalName = "mc" + nextDepth ;
        mc = _root.attachMovie(linkName, internalName, nextDepth) ;
        setX(x) ;
        setY(y) ;
        setSideLength(sideLength) ;
        tileType = linkName ;
        trace("SquareTile constructor: " + x + " " + y + " "
                  + sideLength + " "
                  + internalName + " "
                  + nextDepth ) ;
    }

    // set methods
    public function setX(newX:Number) : Void {this.mc._x = newX ; this.x = newX ;}
    public function setY(y:Number) : Void {this.mc._y = y ; this.y = y ;}
    public function setSideLength(y:Number) : Void {this.sideLength = y ;}
    public function setWidth(width:Number) : Void {
        this.width = width ;
```
this.mc._width = width;
}
public function setSideLength(len:Number) : Void {
    this.sideLength = len;
    this.setWidth(len);
    this.setHeight(len);
}
public function setHeight(height:Number) : Void {
    this.height = height;
    this.mc._height = height;
}
public function setMC(newMC:MovieClip) : Void {
    this.mc = newMC;
    this.x = newMC._x;
    this.y = newMC._y;
    this.width = newMC._width;
    this.height = newMC._height;
}

// get methods
public function getX() : Number {return x ;}
public function getY() : Number {return y ;}
public function getSideLength() : Number {return sideLength ;}
public function getTileType() : String {return tileType ;}
public function getWidth() : Number {return width ;}
public function getHeight() : Number {return height ;}
public function getMC() : MovieClip {return mc ;}
public function getInternalName() : String {return internalName ;}
public function getInternalDepth() : Number {return depth ;}

// other methods
public function destroy() : Void { removeMovieClip(mc) ; }

// print method: prints out the object contents
public function print() : Void {
    trace("Inside SquareTile.print(): " + this.x + " " + this.y + " " +
         " " + this.width + " " + this.height) ;
}

The SquareTile class is a step in the right direction, but to make it even friendlier lets create a gameBoard class that uses gameTile objects. The example code is found in:

(a_tile_4.fla)
(GameBoard.as)
(GameBoardTile.as)

First lets look at the code that uses these classes (which is in a_tile_4.fla):
var theGameBoard:GameBoard = new GameBoard(8,6,300,400) ;

for (var i = 0 ; i < theGameBoard.getNumRows() ; i++)
    for (var j = 0 ; j < theGameBoard.getNumCols() ; j++) {
        if ( ((i+j) % 2) == 0 )
            theGameBoard.createTile(i,j,"greenTile") ;
        else
            theGameBoard.createTile(i,j,"blackTile") ;
    }
theGameBoard.changeTileType(0,0,"pinkTile") ;
trace("Tile width = " + theGameBoard.getTileWidth() ) ;
trace("Tile height = " + theGameBoard.getTileHeight() ) ;

The above is much more intuitive and easier to use. We do not need to worry about X,Y coordinates at all and instead abstract the problem. We now just create a board by specifying the number of rows/columns and the dimensions (width,height) of the board. The class allows us to reference tiles as (X,Y) cells within this abstract 2D grid. The coordinates are taken care of for us. The class has a method for changing the tile type and the example above uses it to change tile (0,0) to a “pinkTile”.

Lets look at the class definition code. First, lets look at GameBoard.as:

class GameBoard {
    // A class to hold information about GameBoard.

    // NOTE:  must call destroy() before delete to make sure any
    //    attached MCs are removed
    private var numRows:Number ;
    private var numCols:Number ;
    private var width:Number ;
    private var height:Number ;
    private var cells:Array ;   //  the array that holds arrays of cells
    private var tileWidth:Number ;
    private var tileHeight:Number ;

    // constructor
    function GameBoard(cols,rows,width,height) {
        this.numRows = rows ;
        this.numCols = cols ;
        this.width = width ;
        this.height = height ;

        // create the space for the cells
        this.cells = new Array(this.numRows) ;
        for (var i = 0 ; i < this.numRows ; i++)
            this.cells[i] = new Array(this.numCols) ;
The constructor initializes the rows/cols/height/width members as well as create a 2D array of cells to hold the GameBoardTiles. Initially the cells are empty, we rely on the user of the game board object to specify which tiles they want created. There is a “createTile” method for this purpose. The “createTile” method creates an instance of the GameBoardTile class. The GameBoardTile class looks like:

```typescript
class GameBoardTile {
    private var x:Number ;
    private var y:Number ; // current x,y location
    private var width:Number ;
    private var height:Number ;
```
private var tileType:String;
private var internalName:String;
private var depth:Number;
private var mc:MovieClip;

// constructor
function GameBoardTile(i:Number, j:Number, gameBoard:GameBoard, linkName:String) {
  depth = _root.getNextHighestDepth();
  internalName = "mc" + depth;
  mc = _root.attachMovie(linkName, internalName, depth);
  var x:Number = i * (gameBoard.getWidth() / gameBoard.getNumCols()) ;
  var y:Number = j * (gameBoard.getHeight() / gameBoard.getNumRows()) ;
  setX(x);
  setY(y);
  setWidth( gameBoard.getWidth() / gameBoard.getNumCols() ) ;
  setHeight( gameBoard.getHeight() / gameBoard.getNumRows() ) ;
  tileType = linkName ;
}

// set methods
public function setX(newX:Number) : Void {this.mc._x = newX ; this.x = newX ;}
public function setY(y:Number) : Void {this.mc._y = y ; this.y = y ;}
public function setWidth(width:Number) : Void {
  this.width = width ;
  this.mc._width = width ;
}
public function setHeight(height:Number) : Void {
  this.height = height ;
  this.mc._height = height ;
}
public function setMC(newMC:MovieClip) : Void {
  this.mc = newMC ;
  this.x = newMC._x ;
  this.y = newMC._y ;
  this.width = newMC._width ;
  this.height = newMC._height ;
}

// get methods
public function getX() : Number {return x ;}
public function getY() : Number {return y ;}
public function getTileType() : String {return tileType ;}
public function getWidth() : Number {return width ;}
public function getHeight() : Number {return height ;}
public function getMC() : MovieClip {return mc ;}
public function getInternalName() : String {return internalName ;}
public function getInternalDepth() : Number {return depth ;}

// other methods
public function changeTileType(newType:String) {
  // remove the movieclip and attach again with the same
  // name and depth.  All members stay same
  // but need to call {setX(),setY(),setWidth(),setHeight()}
  // to adjust new movieClip members to be same as old.
}
First consider the GameBoardTile() constructor. The constructor uses the width and height of the GameBoard, along with the (row,col) arguments to calculate the (x,y) coordinates of the tile. It then uses the attachMovie command to create an instance of the specified MovieClip. So, really, tiles are just MovieClips with some useful additional members and methods.

Now look at the changeTileType( ) method. The method first removes the MovieClip associated with that GameBoardTile and then attaches a MovieClip of the type specified in the argument. In order to remove a MovieClip one needs to know the internal flash name of that MovieClip. Where do we get this info? Simple, we just store it in the GameBoardTile object and we get the name when we create the GameBoardTile in the first place. When we attaché the new MovieClip we just reattach at the same depth as the old one. Again, we store the depth in the GameBoardTile object when we create the tile the first time.

In all of the above examples the tileType has also been the name of the library MovieClip to attach. One possibility is to use this type for special purposes. You could have tileTypes of:

- Water
- Rock
- Barrier
- Grass
- HyperPort
- MineShaft

And so on. Then, you could have properties associated with these tile type. For example, perhaps you have a sprite moving about on the board, but the sprite can NOT move onto Water or Barrier tiles and needs to jump to get onto Rock tiles.

Okay, so how do we move a sprite around the tiles? Easy, just attach a movie clip and control it with the keyboard. What is nice is now you can use the GameBoard object to
also tell you where on the board your sprite is located. In the next example we create a black ball that is moved around the board via the keys. Code is in:

(a_tile_5.fla)

Below is the most relevant code. Assume “theGameBoard” was created as above with greenTile and blackTile cells:

```ActionScript
var userIncrement:Number = 5;

var theBall:MovingMC = new MovingMC(tx,ty,0,0, "redBall") ;
theBall.setWidth(theGameBoard.getTileWidth() * 0.7) ;
theBall.setHeight(theGameBoard.getTileHeight() * 0.7) ;

onEnterFrame = function() {
    if (Key.isDown(Key.LEFT))  theBall.setX(theBall.getX() - userIncrement) ;
    if (Key.isDown(Key.RIGHT)) theBall.setX(theBall.getX() + userIncrement) ;
    if (Key.isDown(Key.UP))    theBall.setY(theBall.getY() - userIncrement) ;
    if (Key.isDown(Key.DOWN))  theBall.setY(theBall.getY() + userIncrement) ;

    var ball_i:Number = theGameBoard.get_i_cell(theBall.getX()) ;
    var ball_j:Number = theGameBoard.get_j_cell(theBall.getY()) ;
    // outLocation is a dynamic text box to display the location of theBall
    outLocation = "In cell (" + ball_i + ", " + ball_j + ")" ;

    if (ball_i < ( theGameBoard.getNumCols()- 1) ) {  // then have a cell to my right
        var neighborType:String = theGameBoard.getCellType( (ball_i + 1), ball_j) ;
        outType = "right: " + neighborType ;
    }
    else { outType = "none right" ;  }
}
```

Try out the example. Remember to click on the board to set focus.

The ball has it size set to be a bit smaller than the tiles. As the ball moves around the dynamic text gives the (i,j) coordinates of the cell containing the ball. This is done by calls to GameBoard.get_i_cell( ) and GameBoard_get_j_cell( ). The code also checks to see if there is a cell to the right, and if so prints out the type of that cell. This is done with a call to GameBoard.getCellType( ). This approach is useful for many reasons including:

- you can check out the type of cell that is a neighbor and use that information to allow or disallow movement into that cell
- You can limit collision detection tests to objects in the current (or current plus neighboring cells)

**Exercises**
1. Add a mouse to the a_tile_5 example so that you move the ball by clicking on a cell and the ball moves to that cell.

2. Add multiple balls and then have the mouse move balls by clicking on a ball then clicking on an empty space. The chosen ball is moved to the selected empty square. If a non-empty square is chosen the move does not happen. How can you keep track of which squares are occupied?

3. Create a game where a character (or ball) is moved around a game board via the keyboard but different tiles have different effects. Include “barrier” tiles that are tiles where the character cannot move to and trap door tiles that look like a normal tile but when a character moves on them they are sucked down into a hole (player loses). Have the player start in one corner of the board and time how long it takes them to get to the other corner of the board. Everytime they hit a “trap door” tile they are moved back to the beginning. Hint: use the GameBoardTile.tileType member effectively.

4. Add a “bad” guy to the above game (who knows the location of the trap doors) who tries to catch the players character. If caught the player is restarted at the beginning. The bad guy needs AI to control its movements. Do two types of AI:
   a. Simple move towards the player based on a one step greedy algorithm
   b. Use the A* path finding algorithm