CPG 5 Coding and Programming

Introduction to Game Programming

Contents

1 Week 10

Managing larger programs This week's subject areas are:

- loops
- switches
- functions

1.1 Loops

As our game concepts increase in complexity so do our programs. 3rd generation languages have a number of features designed to manage that complexity.

A loop is the structured programming way to perform the same action continuously, or while certain conditions hold true.

The C languages have several variations on the structured loop:

- while
- do while
- for
- for each

1.1.1 The *while* loop

The *while* loop is commonly used in situations where you want to continue to loop while a given condition remains true:

Example 1.1 while

```
    var currentPlayer: int = 0;
    while (currentPlayer < numberOfPlayers) {

        // move the current player
        currentPlayer++;
    }
```

The *while* loop works by first evaluating the condition to see if its true. If it's true the program continues into the block and executes until the end of the block. At the end it jumps back to the top and retests the condition. This continues while the condition remains true. But when the condition becomes false, the program jumps to the end of the following block and so exits the loop.

1.1.2 The *do while* loop

This variation on the standard *while* loop is normally used when you always want to loop at least once, and then continue while the condition remains true:

Example 1.2 do while

```
    var gameOver: boolean = false;
    do {
        // update the ball and paddle positions ...
        // check for scoring a point ...
        if (p1Score == winScore || p2Score == winScore) {

            gameOver = true;
        }
    } while (!gameOver);
```

The *do while* loop first executes the contents of the loop. It then checks the condition. If the condition is true, the program counter jumps back to the top of the loop. Otherwise the program continues, exiting the loop.

Can you see the difference between the two forms of *while* loop?

1.1.3 The *for* loop

This type of loop is most commonly used to create a loop that executes a specific number of times. It recognises that a common way to achieve this is through a variable that changes on each loop and stops the loop when it reaches some limit:

Example 1.3 for

```
    for (var index: int = 0; index < numberOfParticles; ++index) {

        // create a random particle ...
    }
```

The *for* loop definition contains 3 parts: the variable initialisation, the conditional, and the variable update. Each part is separated by a semi-colon.
First it initialises the loop variable. Then it tests the condition. If the condition is true, it executes the block, otherwise it exits the loop. If it executed the block, it updates the variable for the next loop and then jumps back to re-test the condition.
On the first run through the loop, the loop variable holds the initial value. When the loop exits, the variable will be one beyond the last valid value of the loop condition.
Note, that if the loop variable is declared inside the loop as in the example above, the variable is not accessible outside of the loop. The loop itself provides the scope for the loop variable.

1.1.4 The *for each* loop

This is a variant of the standard *for* loop. This variation is commonly used to iterate through the contents of an array. This version also specifies a variable which it automatically assigns and updates incrementally through the contents of an array of data:

Example 1.4 for each

```
    for (var enemy: Enemy in enemyList) {

        // do something to each enemy ... e.g.
        enemy.hitPoints -= damage;
    }
```

The *for each* loop initialises and updates the loop variable in the same order as the standard version, but the initial value and update statements are implied.
The machine code produced is usually identical to:

```
    for (var index: int = 0; index < enemyList.Length; ++index) {

        var enemy: Enemy = enemyList[index];
        // do something to each enemy ... e.g.
        enemy.hitPoints -= damage;
    }
```

1.2 Switch cases

Often when you are writing looping code you'll find yourself wanting to do different things in different cases. You could achieve this through a chain of *if* conditions:

Example 1.5 *else if* chain

```
    for (var enemy: Enemy in enemyList) {

        if (enemy.type == 0) {

            // do something to enemies of type 0 ...

        } else if (enemy.type == 1) {

            // do something to enemies of type 1 ...

        } else if (enemy.type == 2) {

            // do something to enemies of type 2 ...

        } else {

            // do something to other types of enemy ...
        }
    }
```

Code like this has two problems. Can you see what they are?

The *enum* type allows us to define a sequential list of values to use in such a chain. By default it assigns an incremental value to each in turn:

Example 1.6 *enum*

```
enum enemyType { fighter, bomber, battleship, freighter };

// ...

    for (var enemy: Enemy in enemyList) {

        if (enemy.type == enemyType.fighter) {

            // do something to enemies of type fighter ...

        } else if (enemy.type == enemyType.bomber) {

            // do something to enemies of type bomber ...

        } else if (enemy.type == enemyType.battleship) {

            // do something to enemies of type battleship ...

        } else {

            // do something to other types of enemy ...
        }
    }
```

The *switch* statement allows us to define a jump table to avoid performing needless tests - particularly when the tests are simple sequential values:

Example 1.7 *switch*

```
enum enemyType { fighter, bomber, battleship, freighter };

// ...

    for (var enemy: Enemy in enemyList) {

        switch (enemy.type) {

            case enemyType.fighter:
                // do something to enemies of type fighter ...
                break;

            case enemyType.bomber:
                // do something to enemies of type bomber ...
                break;

            case enemyType.battleship:
                // do something to enemies of type battleship ...
                break;

            default:
                // do something to other types of enemy ...
        }
    }
```

The *switch* works by first evaluating the expression and then matching it against the listed cases. If it finds a match it jumps directly to that case, otherwise it jumps to the default case. In each case, the code executes and then falls through to the following case unless you provide a break statement. The *break* statement causes the program counter to jump out of the block.

1.3 Functions

Another tool that helps us to scale up programs is the *function* also known as a *subroutine* or *procedure*.

Functions enable us to extract common code and avoid the need to repeat code. Like most 3rd level language constructions they can be nested and combined with other language features. Functions can call other functions, and can even call themselves. These are known as recursive functions.

If you've ever written any Unity script code before, you will be aware that Unity itself provides you with two empty common function definitions when you create a new script: *Start* and *Update*. These functions take nothing as input and return nothing as output. They are called purely for their side-effects.

Example 1.8 Pure function

A function is known as a pure function if it only uses its inputs without modification and does nothing but return an output. The standard mathematical functions would be examples of pure function:

```
    function square(number: int): int {

        return number * number;
    }
```

Pure functions are generally easy to interpret, combine with other functions, re-use, and debug. Generally speaking, the purer you can make your functions, the better.

From a higher level design perspective, you can think of functions as like the black box circuit abstraction techniques we used to design our CPU.

Functions work by passing information through the stack.

Do you remember how stacks work from last week?

When you call a function from your program, the interpreter copies your function arguments and the address of the next instruction onto the stack. It then sets the program counter to beginning of the function code.

The function itself starts by taking the references and values passed to it from the stack. When the interpreter reaches the end of the function it pulls the return address from the stack and pushes the return value onto the stack. Finally it jumps back to point given it by the caller.

The calling code then continues by taking the return value from the top of the stack and using it as the result of evaluating the function.

Example 1.9 evaluating a function

```
var x: int = 9;
var sqX: int = square(x);
```

1.4 Exercise

Your exercise for this week is to review your existing code or game design and look for elements of duplication that can be removed through the use of loops and functions.

Submit your work in UnityScript .js files. You can check your work before submitting by attaching the scripts to Game Objects in Unity first. But only submit the .js files to source control. Don’t forget to keep track of your time and also submit your log files as well. The deadline for this submission is Sunday.