CPG 5 Coding and Programming

Introduction to Game Programming

Paul Sinnett, London South Bank University <paul.sinnett@gmail.com>

Table of Contents

Week 9

Data structures

This week's subject areas are:

- code formatting
- local variables
- data structures
- arrays

Code formatting

Why do we need to format code?

The following examples are exactly equivalent. To the compiler they are the same code. Both examples contain the same error in that they are missing a bracket. Can you spot where the bracket is missing?

Example 1. Un-formatted code

The language compiler program does not care about code formatting. It will generate the correct code so long as the punctuation marks are in the correct places.

```
var minYGap:int=(ballHeight+paddleHeight)/2;var distance:int=ballPositionY
-p1PaddleY;if(distance>=-minYGap&&distance<=minYGap)ballVelocityY
=-ballVelocityX;}
```

Example 2. Formatted code

Formatting our code with a standard placement for brackets and consistent indentation helps human readers of the code to spot errors.

```
var minYGap : int = (ballHeight + paddleHeight) / 2;
var distance: int = ballPositionY - p1PaddleY;

if (distance >= -minYGap && distance <= minYGap)
    ballVelocityX = -ballVelocityX;
}
```

Local variables

Sometimes when you're trying to apply some rule of your game, you want to store an intermediate result. You might want this to make the rule easier to read, or to save time by re-using it in multiple locations.

Example 3.

In this example, *ballIsInPlay* is a boolean variable that I'm using to know if I need to check for collisions between the ball and the player paddles. The variables *xMinGap* and *yMinGap* are temporary local variables used to precalculate the total minimum difference in x and y that the ball and paddle would have to be apart to not hit each other. The variable *distance* is used to hold the difference in y between the current ball centre and the player 1 paddle centre.

Local variables are calculated when the program reaches the point they are constructed, but they are thrown away when the program execution leaves the block (marked by the curly brackets) that contains them. This is known as the *scope* of the variable.

A local variable cannot be used in calculations outside of its scope. But the same variable name can be re-used for a new variable in a different scope.

```
    if (ballIsInPlay) {

        var xMinGap: int = (widthBall + widthPaddle) / 2;
        var yMinGap: int = (heightBall + heightPaddle) / 2;

        if (dxBall < 0 && xBall - xP1Paddle <= xMinGap) {

            var distance: int = yBall - yP1Paddle;
            if (distance >= -yMinGap && distance <= yMinGap) {

                dxBall = -dxBall; // bounce the ball in x!
            }
        }

        // .. and the same for the other paddle
    }
```

Can you identify the scope of the local variables in this code?

The compiler needs to allocate space for local variables just as it allocates space for global variables. If it just allocated space for every local variable instance separately, it would require enough RAM to hold every global and local variable in the program simultaneously. But since local variable are only valid within their scope, it can save memory by re-using the memory needed for local variables once they go out of scope.

The way it does this is with a *stack*.

The stack

A stack is the name of a data structure used in computing that is analogous to a pile of stuff. When you need to store more stuff, you put it on the top of the pile, and when you've finished with it, you remove it from the top. It is a feature of a stack that you only remove from the top. So the last thing on is the first thing off. It is sometimes called a last-in first-out (LIFO) collection.

Stacks are generally useful data structures, but the one used by the virtual machine or CPU itself is known as *the stack*. The location and size of the stack can be configured by the program when it starts, but usually defaults to a fixed size at the end of memory.

If your program ever runs out of space to store the stack, the error condition is known as a stack overflow. Due to the nature of program code it is impossible for the compiler to know at compile time if any given program will ever overflow the stack memory so the top of the stack must be monitored by the program at run time. This is known as a run-time error.

Data structures

Last week we looked at how some 3rd generation languages apply types to data e.g.:

- int
- boolean
- float
- String

In some of the C languages (C, C++, C#) you can define your own types using the *struct* keyword:

Example 4. User defined type

This is a user defined type to represent a common set of attributes in a space simulation game:

```
struct ShipAttributes {

    int shields;
    int hull;
    int fuel;
};
```

Once defined you can make variables of these types and pass them around as single units of data:

```
ShipAttributes shipTemplate;
shipTemplate.shields = 100;
shipTemplate.hull = 100;
shipTemplate.fuel = 100;

// ...

ShipAttributes newShip = shipTemplate;
```

This works in the C languages because all value types, even the user defined types are passed by value by default.

Pass by reference

It's possible to pass values by reference in languages that default to pass by value. This is achieved through the use of pointers.

Languages that inherently pass by reference (such as UnityScript) manage the referencing (pointers) and de-referencing automatically.

In this example, the *new* keyword allocates to you enough memory to store your new item of data, the asterisk indicates that the variable is a pointer, and the ampersand takes the address of some data. The use of the asterisk on a pointer de-references the pointer value. The arrow notation allows you to evaluate a member of the data structure through a pointer:

```
ShipAttributes *newShip = new ShipAttributes;

ShipAttributes *shipTemplate = NULL;
if (fighter) {

    shipTemplate = &fighterTemplate;

} else {

    shipTemplate = &freighterTemplate;
}

*newShip = *shipTemplate;

if (hit) {

    newShip->shields -= damage;
}
```

It is important not to get confused between the value and the pointer to the value in these languages.

Getting this wrong could cost you a million dollars. [https://www.youtube.com/watch?v=evKYF1G_uDI]

This saves some typing and variable creation since blocks of duplicated names can be omitted. When you create a variable such a user defined type, the compiler creates all of the component parts automatically and these can be individually referenced using the dot syntax.

Can you see how the C compiler can lay out and identify each component part of a user defined structure in memory?

You can even nest user-defined types inside other user defined types.

As in C, UnityScript treats variables of primitive types (int, float, boolean, and so on) as their values. But it treats user defined types (defined with the *class* keyword in UnityScript) as references to their values. Implicitly, these are always pointers.

When you create such a type in UnityScript, the script interpreter allocates enough memory from RAM to store your object and gives you a reference to that memory.

Worse yet, UnityScript is what is known as a garbage collecting language. What this means is that when the last reference to some data is gone, the run-time system frees up the memory so that it can be re-used. So copying variables does not copy the contents (just the reference) and with the only reference to the original gone, it will then throw away the data as well.

Value types in UnityScript

Although the UnityScript language does not support the *struct* keyword directly, you can approximate this feature by deriving a class from System.ValueType:

```
class ShipAttributes extends System.ValueType {

    var shields: int;
    var hull: int;
    var fuel: int;
}
```

However, value types are rarely used in UnityScript as the object-oriented model is usually preferred to the lower level data structures of C. In object-oriented programming data structures and the code that operates on them are combined into a single unit known as an *object*. We'll come back to objects in a future class.

Fortunately, whenever you write a script in UnityScript, it implicitly generates a data structure for you. And whenever you drag such a script onto a game object in Unity it creates an instance of that data structure that belongs to the game object.

Example 5. Data structures in Unity

Variables at global scope create data structure members that are copied to each game object in Unity to which the script is added.

```
public var shields: int = 100;
public var hull: int = 100;
public var fuel: int = 100;
```

This code therefore encapsulates everything above without the need to explicitly create and copy data structures.

Arrays

Another form of data structure common to the C languages is the concept of arrays of data. You can think of an array as a collection data types of the same type. We've already seen two common uses for arrays in games.

A one dimensional array can be used to store a string of ASCII coded characters. This type of array has a variable length and is terminated by a special end code called *NULL*.

Example 6. C language string as an array

One important thing to remember when dealing with arrays is that the size of the array is number of elements in it, but the index into an array always starts with index 0. This is because the memory is stored contiguously and so to find the memory address of any element you just need to multiply the index by the size of an element.

```
char message[] = "Are you ready?";
char *prompt = "Press space to play!";
char score[4];
score[0] = '1';
score[1] = '0';
score[2] = '1';
score[3] = 0;
```

UnityScript ASCII string

In UnityScript there is a special built in type defined to hold a string. This special syntax allows a lot of programming short-cuts. However, internally, this string is stored as an array of characters just like the C version above.

```
var message: String = "press space to start!";
```

A two dimensional array can be used to store a bitmap or pixel map. The bitmaps we created earlier in the course are an example of data that could be stored in such an array.

Unlike a string, these arrays are usually of a fixed, known size. This allows the programming language to calculate the offset automatically. If you know the width of a two dimensional array you can work out the index given the row and column.

Example 7. C language bitmap as two dimensional array

As with one dimensional arrays, the indexes start at zero. Again, this makes finding the address of the data easy. To find the beginning of any row, multiply the row index by the size of a row. To find the address in the row of any element, multiply the index by the size of an element.

```
char bitmap[8][2] = {
    { 0x0F, 0x00 },
    { 0x7F, 0xE0 },
    { 0xFF, 0xF0 },
    { 0xE6, 0x70 },
    { 0xFF, 0xF0 },
    { 0x39, 0xC0 },
    { 0x66, 0x60 },
    { 0x30, 0xC0 }
};

// animate
bitmap[7][0] = 0xC0;
bitmap[7][1] = 0x30;
```

Three or more dimensional arrays can be creating by extending this concept.

Arrays of different sized things, e.g. an inventory of strings, is achieved by making an array of pointers (since pointers are always the same size, no matter what they point to.)

Arrays can be created from user-defined types. And user-defined types can contains arrays. And each can be nested inside the other.

Exercise

Your exercise for this week is to identify the data structures that you want to use in your game. Name each element and identify what type you want to represent the data.

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.