CPG 5 Coding and Programming

Introduction to Game Programming

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

Table of Contents

Week 8

3rd Generation Languages

This week's subject areas are:

- Virtual machines
- 3rd Generation Languages

Virtual machines

Although with assembly language, the process of creating programs to run on our simple chip design is much easier, we still have to spend a lot of time working around the limitations of our minimal instruction set. One way around this is to imagine a processor with more registers and more instructions.

As we've already established, machines with more resources are not actually more capable but they can be more efficient in terms of the amount of memory required to describe the desired programs.

Fortunately, we do not need to design any new hardware to make use of such a machine. Instead we can create a VM (virtual machine) specification and write an assembler to create programs for it.

The advantages of doing it this way are that it further abstracts the source code from the underlying hardware. To have your program execute on different hardware, you only need to rewrite the virtual machine itself. The virtual machine code programs will then run on any hardware that can fully implement the virtual machine specification.

Example 1. A virtual machine

To extend the instruction set of our simple processor we could create a virtual machine specification that allowed instructions to operate on four registers A, B, C, and D rather than two.

Such a virtual machine could be implemented on our simplified processor by mapping these registers to the first four RAM locations. So if we wanted to load the value 640 into register A and 480 into register B we could write the assembler instructions for our virtual machine as:

```
; virtual machine assembler code
.code
    mov ax, 640
    mov bx, 480
```

This could then be transformed into native assembler by the virtual machine or a JIT (just in time) compiler:

```
; actual machine assembler code
.data
    ; The first 4 words of RAM are reserved for the VM...
    AX word
    BX word
    CX word
    DX word

.code
    ; VM write 640 to AX
    mov ax, 640
    mov dx, ax
    lea ax, AX
    mov [ax] dx

    ; VM write 480 to BX
    mov ax, 480
    mov dx, ax
    lea ax, BX
    mov [ax], dx
```

Unity scripting

In Unity, all of the programming languages are first compiled to CIL (common intermediate language) before being converted to machine specific instructions by a virtual machine or just in time compiler. Together this is known as the Mono platform. It is an open source version of the Microsoft .NET framework.

3rd Generation Languages

But even extending the limited instruction set of our chip, we still run into many common assembly language problems:

- manual management of registers
- expressions cannot be composed
- code must often be duplicated
- the flow can be hard to follow

3rd generation languages take programming languages a step further creating a whole new layer of programming abstraction. The 3rd generation languages we'll be looking at are the C family of languages.

These languages adopted the structured programming paradigm which is generally considered to make programs easier to read and correctly interpret.

Broadly speaking, the C family are:

- C
- C++
- Objective C
- Java
- C#
- Javascript
- UnityScript

These languages share most of their syntax and punctuation. For example, all of the languages in this family use curly brackets {} to define the block structures of a structured program.

Expressions

Most 3rd level languages allow you to compose expressions into a single statement of code. This allows you to express your idea more naturally and concisely than assembler syntax allows.

Example 2. Expressions and assignments in a C family syntax

```
if (ballPosX - ballRadius > screenWidth - paddleWidth) {

    p1Score = p1Score + 1;

} else if (ballPosX + ballRadius < paddleWidth) {

    p2Score = p2Score + 1;
}
```

A program language compiler would then turn this into native or virtual machine assembler code.

Example 3. Expressions and assignments converted to a virtual machine language

```
.data
    ballPosX word
    ballRadius word
    screenWidth word
    paddleWidth word
    p1Score word
    p2Score word

.code
checkRight:
    mov ax, [ballPosX]
    sub ax, [ballRadius]
    mov bx, [screenWidth]
    sub bx, [paddleWidth]
    cmp ax, bx
    jg [scoreP1]
    jmp [checkLeft]
scoreP1:
    inc [p1Score]
    jmp done
checkLeft:
    mov ax, [ballPosX]
    add ax, [ballRadius]
    cmp ax, [paddleWidth]
    jl [scoreP2]
    jmp [done]
scoreP2:
    inc [p2Score]
done:
    ...
```

The virtual machine or just in time compiler can then go through and convert this into native machine code.

Can you see how to do this?

Variables

The most immediately obvious difference between UnityScript and the rest of the C style languages is the syntax for declaring variables. In UnityScript you declare a variable with the *var* keyword:

Example 4. Declaring variables in UnityScript

The following shows how you might declare some variables for use in your game rules:

```
    var ballPosX: int;
    var ballRadius: int;
    var screenWidth: int = 640;
    var paddleWidth: int;
    var p1Score: int = 0;
    var p2Score: int = 0;
    var gameOver: boolean = false;
```

Example 5. Declaring variables in other C type languages

Other C style languages omit the *var* keyword, reverse the order of the type and the name, and must not be separated with a colon:

```
    int ballPosX;
    int ballRadius;
    int screenWidth = 640;
    int paddleWidth;
    int p1Score = 0;
    int p2Score = 0;
    bool gameOver = false;
```

Whichever language syntax you are using, the compiler will convert your program into a series of assembler data declarations.

Example 6. Variable declarations converted into virtual machine assembler language

The following shows how variable declarations in a 3rd generation language might translate to assembler language:

```
    false = 0
    true = !false

.data
    ballPosX word
    ballRadius word
    screenWidth word 640
    paddleWidth word
    p1Score word 0
    p2Score word 0
    gameOver word false
```

Structured programming

The block structure of conditional and loop statements simplify comprehension of the program for the human reader.

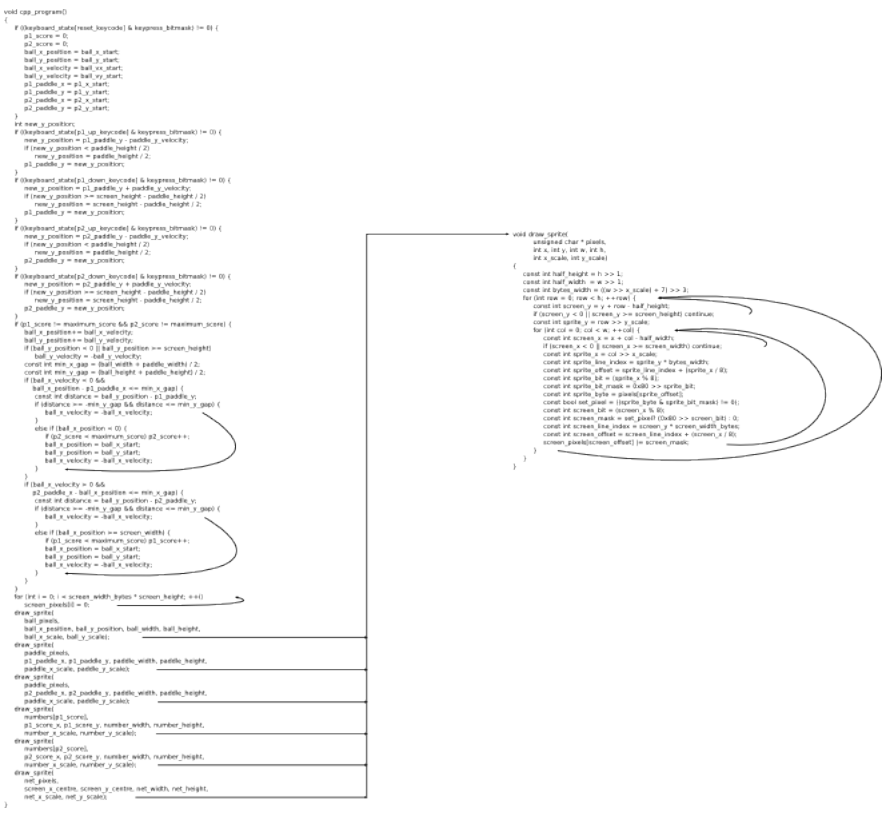
Why does it matter how easy it is for a human to read?

Figure 1. spaghetti code

The following program code is written in C++, but without using the structured programming features of the language. This is not much easier to follow than a machine language program. If we look at the large scale structure (ignore the actual code just look at the overall shape and flow) it looks a bit like a pile of spaghetti:

Figure 2. structured C++

By using the structured programming blocks of the C languages, this version avoids the apparent complexity of spaghetti code:



Exercise

For this week's exercise I'd like you write out the rules to your game using UnityScript. By now you should have some idea of the game code you want to write for your assessment. So for this exercise, I'd like you to switch to describing the rules you want to use in that game if you have not done so already.

I'd like you to use the *if* and *else* keywords of the C languages to help you to structure your rules.

Pay extra attention to make sure you get all of the brackets matched an in the correct places. Use the examples above as a guide.

Write out your program code as a .js text file. Submit your files as usual to source control before Sunday. Don't attempt to submit a Unity project as that will attempt to upload many extraneous files. And don't forget to include your work log with your submission.