CPG 5 Coding and Programming

Introduction to Game Programming

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

Table of Contents

Week 2

Boolean logic

This week's subject areas are:

- Boolean logic
- combining logical operations
- source control functions

Boolean logic

Boolean logic deals only with the values of *true* and *false*. We sometimes refer to these values with different names for the same concept like: 1 and 0, on and off, yes and no, high-voltage and low-voltage, but the meaning is the same in each case.

George Boole

He was a 19th century mathematician and philosopher from Lincoln. He came up with a system of algebra for determining truth values so it was named after him. His work was later used as a basis for the modern digital computer.

Computers deal with things exclusively in terms of Boolean logic. To make your game run on a computer, you need to encode your game's rules with Boolean conditions that resolve to either true or false.

Figure 1. The form of a Boolean rule in a C family language

```
if (condition) { action; }
```

In this example, the program will carry out the given *action* only if the given *condition* is true.

Once you have a complete set of such rules to fully describe your game, you have all of the parts of the program you'll need. However, finding all of the rules is not as easy as it sounds. To help us break things down, we can name our *condition* and *action* identifiers descriptively.

As we work through these examples, I'd like you to adopt the *Javascript* naming convention known as camel case. In this convention you push the words of an identifier together so that it doesn't contain any spaces. This is because C family languages treat spaces as separators between identifiers. Words are still identifiable within this convention through the use of capitalisation. In C family languages, capital and non-capital letters are distinct, so *ball* and *Ball* are considered two different identifiers.

Camel Case

Camel Case is so called because the Capital letters in the words look the humps on a camel's back.

So if we wanted to express the rule in Pong which defines the behaviour of the ball we might do it like this:

Example 1. An example of a rule from the game Pong

```
if (paddleHitsBall) { bounceBallHorizontally; }
```

This is a pretty good expression of the rule. A more accurate expression would be to include another action to emit a beep. We could do this by simply repeating the rule condition. For example:

Example 2. An extended example of a rule from the game Pong

```
if (paddleHitsBall) { bounceBallHorizontally; }
if (paddleHitsBall) { playBeepSound; }
```

However, the convention in C family languages is to simply list multiple actions in order so long as they are between the curly brackets and separated by a semi-colon. For example:

Example 3. A better expression of two actions for one rule

```
if (paddleHitsBall) { bounceBallHorizontally; playBeepSound; }
```

This example and the one above it are equivalent but this form is more usual in the C family of programming languages.

Combining logical operations

We can break down our example of a rule from the game Pong a little further by identifying the left paddle and the right paddle separately. But if we do that, we'll end up with two rules:

Example 4. An extended example of a rule from the game Pong

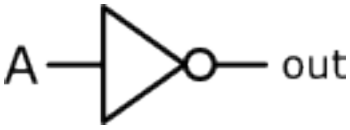
```
if (leftPaddleHitsBall) { bounceBallHorizontally; playBeepSound; }
if (rightPaddleHitsBall) { bounceBallHorizontally; playBeepSound; }
```

As in the previous section we have unwanted repetition of the action we want to describe. To get around this, we can combine the logical operations using an expression that also results in either true or false.

Fortunately there are only 3 basic logical operations that you need to remember: *AND*, *OR*, and *NOT*. It may seem incredible, but all of a computer's functioning is derived from the combination of these operations.

Figure 2. NOT

The *NOT* operation, also known as a logical inverter, reverses the Boolean value of its input. If it has *true* as input, it outputs *false*, and vice versa. When drawing a Boolean logic diagram it is drawn as follows, where *A* is the input and *out* is the output:



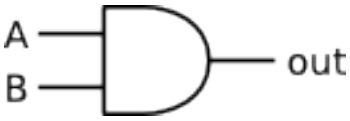
In the C family of programming languages, this operation is written with an exclamation mark:

```
if (!gameOver) { playOneGameFrame; }
```

The above is an extremely common rule that you'll find in the main loop of most game programs. You can read this rule as "if the game is not over then play one frame of the game." This prevents the game from continuing to run after the end condition of the game has been triggered, because the game is only played each frame that the condition `gameOver` results in the value *false*.

Figure 3. AND

The *AND* operation results in a *true* only when both of its inputs are also *true* otherwise it resolves to a *false* value. When drawing a Boolean logic diagram it is drawn as follows, where *A* and *B* are the inputs and *out* is the output:



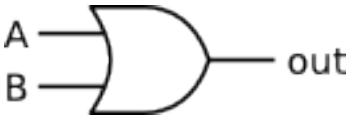
In the C family of programming languages, this operation is written with two ampersands:

```
if (ballInPlay && ballOnScreen) { moveBall; }
```

You can read this rule as "if the ball is in play and the ball is on the screen then update the ball's position." This allows the ball to be on screen but not moving, such as when the game is running but the ball has yet to be served.

Figure 4. OR

The *OR* operation results in a *false* only when both of its inputs are also *false* otherwise it resolves to a *true* value. When drawing a Boolean logic diagram it is drawn as follows, where *A* and *B* are the inputs and *out* is the output:



In the C family of programming languages, this operation is written with two vertical bars:

```
if (leftPaddleHitsBall || rightPaddleHitsBall) {

    bounceBallHorizontally;
    playBeepSound;
}
```

You can read this rule as "if left paddle hit the ball or the right paddle hit the ball, then bounce the ball horizontally, and play a beep.

You can chain these operations together to make other logical operations. In fact with 2 inputs you can make 16 distinct logical operations by combining the 3 basic logical operations. Can you find all 16?

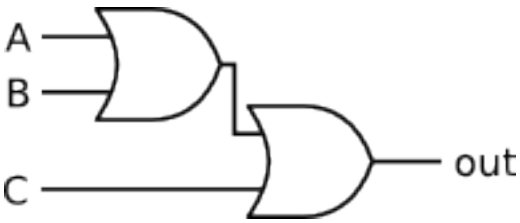
Warning

The equation for working out how many distinct combinations are possible with a given number of inputs is: 2^{2^n} where n is the number of inputs. So for 2 inputs the number of outputs is $2^{2^2} = 2^4 = 16$, for 3 it's $2^{2^3} = 2^8 = 256$, and so on. As you can see, these numbers get big very quickly, so it's important to reason through your logical conditions rather than attempting to guess the right combination. With 4 inputs, your chances of guessing the right combination is 1 in $2^{2^4} = 2^{16} = 65{,}536$!

You can also chain these operations together to make operations with more inputs:

Figure 5. 3 input OR

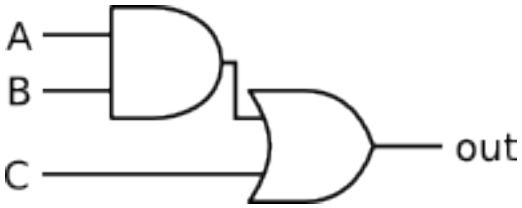
A 3 input *OR* operation can be formed by chaining the output of one 2 input *OR* operation into another:



Here, the output of `(A || B)` is fed into the input of the second *OR* operation. The result can be written `((A || B) || C)`. But more usually we would simplify this to: `(A || B || C)`.

Caution

So long as the operations are of the same type, you can chain as many together as you like. However if you try to combine *OR* with *AND* operations, the order is ambiguous. Does `(A && B || C)` mean:



Or does it mean:

If you have difficulty seeing why these two examples produce different results given the same inputs, try following through the inputs and output at each step. In the first case, if input A is false, the output is the same as the C input. But in the second case, if the input A is false, the output is always false.

In fact, the C language rules specify that && takes precedence and is always first. However, it is good practice not to rely on remembering the C language rule for this because it is so easily mistaken. To force the evaluation one way or the other, use additional brackets to make your intention clear: `((A && B) || C)` for the first example, and `(A && (B || C))` for the second.

Exercise

As an exercise for this module I want you to create a small game that demonstrates your understanding of the concepts that we cover. Each week a small exercise will help you to realise that goal. For this week's exercise, I want you to start thinking about a game design to submit for the course.

Think about the rules that you will need to encode into your program to make your game work. And write down those rules in the form described above:

```
if (condition) { action; }
```

Remember that I want you to create a game design that makes use of the 3 main Boolean operations.

Tip

If you haven't thought of the game design you want to make yet, just pick a simple classic video game design and extract a set of Boolean logical rules from that.

Write out your list of rules in a notepad file, save the result as `week2.txt`, and submit to source control. Don't forget to keep track of your time on the task, and submit your task log along with your list of rules.

Source control functions

As you signed up to source control in week 1, I'd like you to submit your exercise files to the repository. If you don't remember how to do that, we'll go through it again during the class.

Once you have a folder synchronised with our source control you can submit files by saving the file into the folder, using TortoiseSVN to add the file to source control, and then submitting the results.

Caution

Unlike Dropbox, changes to your files are not automatically synchronised with the source control server. When you make any changes locally, they remain local until you choose to submit them. This is a very useful feature in a programming context, because it means you can try out changes locally before committing those changes onto the server. However, it is easy to forget to commit your changes, particularly if you are used to using an automatic update system such as Dropbox.

Once you've added files to source control, TortoiseSVN keeps track of changes you make to the files. You can see this because the green tick icon will change to a red exclamation. But if you delete a file, TortoiseSVN doesn't record the change. And the next time you **Update** from TortoiseSVN the file will be restored from the server.

If you need to make changes to the files that are stored in SVN (as opposed to the contents of the files that are stored) you have to use the TortoiseSVN versions of the file operations rather than the Windows Explorer versions. You can access these from the context menu. There are TortoiseSVN versions of **Delete**, **Rename**, **Copy**, and **Move**. When you've made your changes in this way, they will appear in the box when you're ready to commit your changes to the server.