Control Flow - Part2: Loops

At a Glance

```
// The While-Loop
while(sum <= 100)
{
    // This code will be repeatedly executed as long as sum <= 100,
    // if that condition was satisfied to begin with.
    // If not, the whole code block will be skipped.
    sum = (sum * 2) + 1;
}


// The Do-While Loop:
do
{
    sum = (sum * 2) + 1;    // executed AT LEAST ONCE, since

} while(sum <= 100);        // the condition is evaluated at the end


// The For-Loop:
for(int i = ; i < 10; ++i)
{
    myArray[i] = RandInt(, 100);       // executed 10 times
}

// read (int i = 0; i < 10; ++i) header as:
//     "start at i = 0,                              --> int i = 0
//      loop while i < 10,                           --> i < 10
//      and, after each cycle, increment i by 1"     --> ++i
```

Discussion

Loops are a powerful tool; they allow you to execute peaces of code several times in a row, to iterate over array elements, to easily assign the same callback to several game objects using just one line of code, and more. This is achieved by providing a *condition* which the loop uses to determine how many times, or for how long, it should repeat the execution of it's body (the part between { and }). A loop cycle is also called an *iteration*.

The While-Loop

This is one of the simplest loops, but it is also the one in which you're most likely going to make a mistake and make it loop infinitely. The syntax, in pseudocode, is as follows:

```
while(condition)
{
    // Commands to repeat as long as condition == true
}

// When the loop exits, execution continues here
```

Here, *condition* can be any expression which evaluates to a `bool`. Let's study an example:

```
float currentTime = 0.0;    // in seconds
float maxTime = 60.0;       // in seconds

while(currentTime <= maxTime)
{
    AddTimer("internal.timer.id", currentTime, "OnPlayScaryNoise");
    currentTime += 10.0;    // increments currentTime by 10
}

// OnPlayScaryNoise() callback defined elsewhere...
```

The code above adds several game timers, which in turn make a scary sound play in 10 second intervals. First, two variables are declared: `currentTime` is the time span setting for the timer being currently initialized, and `maxTime` is the maximum time span we want to reach, before we stop looping. Then, the while-loop is started, and it is given the condition `currentTime <= maxTime`. While this condition is true (pun intended), the code in the body of the loop will be repeatedly executed.

In the loop's body, the first line adds the timer for the current iteration. The second line, where `currentTime` is incremented by 10, is very important. Without it, `currentTime` would allways remain 0, and the loop would *never exit*. This is called an *infinite loop*, and is definitively something to be avoided.

The loop above iterates 7 times before it exits. Let's do a quick breakdown to see why.

1. At the beginning, `currentTime == 0`.
   Since the condition is satisfied, we enter the loop. The timer is invoked immediately, and then currentTime is incremented by 10.
2. Now, `currentTime == 10`.
   Condition is still valid, so a 10-second timer is added, and then currentTime is incremented again.
3. `currentTime == 20`,
   which is still <= maxTime. A 20-second timer is added, etc.
4. `currentTime == 30`,
   condition satisfied, 30-second timer added, etc.
5. `currentTime == 40`.
6. `currentTime == 50`.
7. `currentTime == 60`.
   At this point, the test condition passes, so a 60 second timer is added. Finally, currentTime is incremented to 70.

- The condition is checked one more time. As `currentTime == 70`, this causes the loop to exit, skipping it's body and continuing on the first line of code after it.

Note that the *condition* is tested at the start. This means that if the *condition* is not satisfied to begin with, the while-loop is never executed.

The Do-While Loop

The do-while loop is almost exactly the same as the while-loop, except that the condition is located at the bottom, and tested at the end of each operation. For this reason, the body of a do-while loop is executed *at least once*.

Syntax (pseudocode):

```
do
{
    // Executed at least once, and then
    // repeated as long as condition == true.

} while(condition);

// When the loop exits, execution continues here
```

The For-Loop

The for-loop is the most commonly used loop, but it looks somewhat scary when you first look at it. It is not that complicated though. Normally, the for-loop “counts” from some initial value to some other value, executing it's body on each step of the way. This is the syntax:

```
for(counter; condition; step)
{
    // loops as long as condition == true
}
```

The *counter* is a placeholder for a declaration of the loop's counter variable. Traditionally, this variable is named `i` (as it is often used as an *index* to an array), so in place of *counter* you'll usually see something like `int i = 0`. However, it is possible for the loop to use a variable that has been already declared elsewhere in the code, in which case the variable only needs to be initialized, so you might also see something like this: `i = 0`.

The *condition* has the same role as in a while-loop, discussed above. Similarily, it is checked at the begining, so if it is not satisfied from the start, the for-loop never executes. Otherwise it iterates until the condition is broken.

The *step* placeholder stands for an expression which somehow modifies the counter variable (usually incrementing it by 1), thus eventually causing the test to fail, and the loop to stop. Usually you'll see `++i` or `i++` in it's place (both increment i by 1). The *step* expression is executed *after* each iteration.

So, to sum it up, the execution sequence is as follows:

1. The *counter* variable is initialized *once*, at the start.
   (from here on, the loop starts to iterate)
2. The *condition* is checked. If the test fails, the loop's body is skipped, and the execution continues after it (at (5.)).
3. The body of the loop is executed (code inside { and }).
4. The *step* expression is executed. The execution jumps back to (2.)
5. The execution continues on the first line of code after the loop.

A simple example to begin with:

```
int sum = ;

for(int i = ; i < 3; ++i)
{
    sum = sum + 5;
}
```

The loop above iterates 3 times (since the counter `i` takes values 0, 1, and 2). The code has the exact same effect as if this was written instead:

```
int sum = ;

sum = sum + 5;      // sum =  0 + 5 =  5
sum = sum + 5;      // sum =  5 + 5 = 10
sum = sum + 5;      // sum = 10 + 5 = 15
```

In the example below, a for-loop is used to calculate the sum of the first ten integers. Note that the condition states “i <= 10”, and that the initial value is 1. It loops 10 times:

```
int sum = ;

for(int i = 1; i <= 10; ++i)
{
    sum = sum + i;
}
```

What if you wanted to sum up only the even integers in the 1–10 range? Well, here's one way to do it:

```
int sum = ;

for(int i = 2; i <= 10; i = i + 2)
{
    sum = sum + i;
}
```

In the code above, the only difference is in the *step* expression.

Here's a simple example which initializes an array to contain the first 100 even nonnegative numbers.

It executes loops 100 times (since that's the number the length() function returns). However, note that in this case the variable is initialized to `i = 0`, and that the condition checks for (when values are replaced) `i < 100` and not `i <= 100`. This is important, since array indices start at 0. This means that the index of the last element is `length() - 1`, which is 99 in this case, so if the condition `i <= 100` was used, the code would try to assign a value outside the bounds of the evenNumbers array!

```
int[] evenNumbers(100);

for(int i = ; i < evenNumbers.length(); ++i)
{
    evenNumbers[i] = 2 * i;
}
```

Here's another example from the game itself, which executes 30 times (since it starts from 0!); `i` takes values from 0 to 29:

```
// (from 06_distillery.hps)

for(int i = ; i < 30; ++i)
{
    AddTimer("cellar_wood01_5_Body_1", i * 0.1f, "TimerOpenDoor");
}
```

Armed with this knowledge, you can now see that the for-loop is not so different than the while-loop, although it's a bit harder to accidentally create an infinite for-loop. In fact, with some experience, both types of loops can be used to accomplish the same tasks:

```
for(int i = ; i < 10; ++i)
{
    // do something
}

// is the same as

int i = ;
while(i < 10)
{
    // do something

    ++i;
}


// ALSO:


while(condition)
{
    // do something
}
```

```
// is the same as

for(; condition; )
{
    // do something
}
```

To find out more about conditions, comparison operators, and how to create compound conditions using logical operators, see Control Flow - Part1.