# PART 15: “File I/O”

## Introduction

Before we continue with typical game related topics, you need to know a few more general techniques that will help you when you write your games. One of these techniques is *file I/O*. ‘I/O’ stands for ‘input and output’. You are of course already performing file operations, for example when you load an image. However, this is handled by some image library (FreeImage). Sometimes, you simply want to store your own data.

## Text output

The very first article of this series made you write a ‘hello world’ application. This was also the last application that had you print anything to a text window. There’s a reason for this: there are too many books already that make you write text-based applications, and this simply has little to do with game development.

There are however a few areas where `printf` comes in handy. One of these areas is *debugging* (see article 7 – “Debugger”). Try this piece of code:

```
float x = 200, y = 0, vx = 0.1f, vy = 0;
void Game::Tick( float deltaTime )
{
    screen->Clear( 0 );
    screen->Box( x, y, x + 5, y + 5, 0xffffff );
    if ((vy += 0.02f, y += vy) > SCRHEIGHT) vy = -vy;
    if ((x += vx < 0) || (x >= SCRWIDTH)) vx = -vx;
}
```

This is probably the shortest (but also most horribly-written) bouncing cube code possible (*send your code if you can think of something worse!)*. On top of that, it doesn’t work: the bouncing cube leaves the screen at the right side. Let’s do some debugging!

Add the following line to your Tick function:

```
printf( “X-position: %f\nY-position: %f\n”, x, y );
```

What happens?
The output of the `printf` command is directed to the text window that is behind your regular window. Click (and hold) the title bar of the regular window to pause the app, this will let you examine the contents of the text window.

### Printf

The `printf` command used in the previous section has a number of 'interesting codes' in it. There's the '`%f`': apparently, it is replaced by the contents of variable `x`. Then, there is the '`\n`': this sequence emits a 'line break', so printing continues on the next line.

The `printf` command is pretty useful, especially if you know how to use it. Find out about the details at this page: http://www.cplusplus.com/reference/clibrary/cstdio/printf/ . Here, you will learn that you can replace '`%f`' by '`%i`' to print an integer. You can also write '`%.2f`' to print a float with 2 decimals.

### Printing to a string

Sadly, all this formatting goodness is not something we can use with the Surface::Print function, which just expects a plain string to print. We can however *print to a string* to bypass this issue:

```
char text[128];
sprintf( text, "X-position: %f\nY-position: %f\n", x, y );
screen->Print( text, 2, 2, 0xffffff );
```

This time, we get all the benefits of printf, and use this to print the current position of the box to the graphical window.

### Printing to a file

Once you know how to get text to the screen using `printf` and to a string using `sprintf` moving on to files is easy. Try the following:

```
FILE* f = fopen( "positions.txt", "a" );
fprintf( f, "X-position: %f\nY-position: %f\n", x, y );
fclose( f );
```

This time, the position of the cube will be written to a file. Note how 'printf' was replaced by '`fprintf`'. The first parameter of `fprintf` is a '`FILE`', which you first need to open. In this case, the file is opened for *appending* (`"a"`); you can also open a file for *writing* (`"w"`, this will clear the file first) or *reading* (`"r"`). You can also open the file for *binary* reading and writing, using `"rb"` and `"wb"`.

### Retrieving data from a file

Before we continue, you need to make a file for testing purposes. Using notepad, create a file named 'settings.txt' and place it in the directory where your source files are. Put the following info in it:

```
xpos = 100
```

Now, in your init function, add the following code:

```
FILE* f = fopen( "settings.txt", "r" );
fget( f, "xpos = %f", &x );
fclose( f );
```

Also make sure to move the line that defines `x`, `y`, `vx` and `vy` above the `Init` function. When you run the application, you will see that the cube starts bouncing at `x` = 100 now, and you can change the start position using a configuration file, without recompiling your code. This is a very useful feature for games.

**Binary data**

The data that we have written to and read from a file so far has been *text-only.* The advantage of this is that the resulting files are 'human-readable'; the disadvantage is that the files get somewhat large. A float variable, for instance, needs 4 bytes in memory, but in a file, 3.14159265358979 requires 16 bytes.

If you don't need to be able to read the file yourself, you can use *binary file i/o*. This lets you write the contents of any variable, or even better, the contents of any area in memory, directly to a file. Here are some examples:

```
FILE* f = fopen( "bindat.bin", "wb" );
fwrite( &x, 4, 1, f );
fwrite( &y, 4, 1, f );
fclose( f );
```

This code will create a file for binary writing. Then, it writes two blocks of 4 bytes (the size of a float). The memory area that is written by the first line starts at '`&x`' (the *address* where variable `x` is stored).

Note that you need to know how large a variable is to be able to store it to a file. When in doubt, use '`sizeof`':

```
fwrite( &i, sizeof( int ), 1, f );
```

Compared to just writing '4' this will not make your code slower, it's just more typing.

Reading the data from such a file is very similar:

```
FILE* f = fopen( "bindat.bin", "rb" );
fread( &x, 4, 1, f );
fread( &y, 4, 1, f );
fclose( f );
```

This time, the file is opened for binary reading, and `fread` is used instead of `fwrite`.

**Assignment**

Create a function that saves the contents of the screen to a .tga image file that loads in your favorite image editing application.

1. Find one of your previous assignments and add a function `Game::ScreenShot` to it.
2. A *tga file* consists of a 'header' and the actual image data. For the image data, you can simply save lines of pixels. For the header, you need the following data:

```
struct TGAHeader
{
   unsigned char ID, colmap;      // set both to 0
   unsigned char type;            // set to 2
   unsigned char colmap[5];       // set all elements to 0
   unsigned short xorigin, yorigin; // set to 0
   unsigned short width, height; // put image size here
   unsigned char bpp;             // set to 32
   unsigned char idesc;           // set to 0
};
```

   You can now create a variable of type '`TGAHeader`':

```
TGAHeader header;
```

   The size of the header is (and absolutely must be) precisely 18 bytes. You can verify this again using `sizeof( TGAHeader )`.

   Finally, you can set individual fields of this header:

```
header.ID = 0;
```

3. Create the actual function.
4. Add code to your application that saves a screenshot whenever you press a specific button.

OPTIONAL:

5. Use `sprintf` to save your screenshots to a new file each time you press your screenshot key. Use string formatting to produce file names like '`screenshot001.tga`'.

## *END OF PART 15*

*Next part: "Physics"*