Floating Point versus Fixed Point Numbers

Most of us are familiar with the integer data types used by the Arduboy environment – of various lengths and both signed and unsigned – and we take care to choose appropriate types that consider the range of the data we are holding versus the memory required to store them. Operations involving integer values are incredibly efficient and for many sketches are all that are required.

Sometimes integer values are simply not flexible enough. Imagine trying to calculate the movement of a rocket as it accelerates or calculate the trajectory of a missile in an Arduboy version of Choplifter (now there's a good idea for an Arduboy project!). To make these calculations, you need numbers with an integer and fractional component. These can be represented in two ways – floating point or fixed point

C++ includes two floating point data types of different precisions called 'floats' and 'doubles'. On most platforms would take 4 and 8 bytes each however on the Arduino / Arduboy are actually same precision – 4 bytes and are therefore interchangeable. These data types can store numbers in the range of -3.4028235E x 10^{38} to +-3. 4028235E x 10^{38} (as a comparison there is estimated to be 7.5 x 10^{18} grains of sand in the world) but at a precision of only 6 or 7 decimal digits.

In contrast to integer operations, floating point calculations are incredibly processor intensive on an Arduboy or any other processor for that matter. Including numerous calculations to reposition the player and enemies within the main loop of a program might prevent the game from performing well or even being playable.

An alternative to **Floating Point** numbers and arithmetic are **Fixed Point** numbers. These behave more like the integer values you are familiar with and have a fixed (hence the name) number of integer and decimal points. As such, the processing power required to manipulate fixed point numbers is more aligned with integer manipulations than floating point calculations. Using fixed point numbers, that version of Choplifter might be possible after all!

The trade off with using Fixed Point data types is that you must explicitly nominate the size of the integer and fractional portion of the variable – in much the same way you do when choosing integer data types. In addition to the obvious data range restrictions of the integer component, the resolution or granularity of the fractional component must also be chosen. Although it is practical to have a fixed point variable stored in a single byte with (say) 4 bits devoted to the integer portion and 4 bits devoted to the fractional portion, this will give you a data range for the integer portion of 0 to 15 and the fractional portion 0 to 1 in 1/16th increments - providing an overall range of 0 to 16.

Fixed Points in Action

In the examples below, I am using @Pharap's FixedPoints library which can be found here https://github.com/Pharap/FixedPoints/releases

@Pharap's library defines a number of signed and unsigned data types which are listed at the end of this article. It also includes overrides for the common math and comparison operators (+, -, / , *, <, >, etc) and has some handy functions like ceilFixed(), floorFixed() and roundFixed(). The details of the library can be found in the readme.md file.

Firstly, some simple assignments and math:

```
SQ7x8 myVar = 7.0;
myVar = myVar + 3;
Serial.print("static_cast<float>(myVar) : ");
Serial.println(static_cast<float>(myVar), 4);
```

Output:

```
static_cast<float>(myVar) : 10.0000
```

Simple. Now let's introduce a decimal place:

```
SQ7x8 myVar = 3.7;
Serial.print("myVar.getInteger() : ");
Serial.println(myVar.getInteger());
Serial.print("static_cast<float>(myVar) : ");
Serial.println(static_cast<float>(myVar), 4);
```

Output:

```
myVar.getInteger() : 3
static_cast<float>(myVar) : 3.6992
```

But hang on, we set the variable to 3.7 and it printed out 3.6992. Why so? As mentioned earlier, both **Floating Point** and **Fixed Point** numbers introduce a degree of inaccuracy when storing numbers. With Fixed Point numbers, the accuracy is specified by the programmer. In the sample code above, the variable chosen was a two-byte incarnation and it can store a number between -128 and +128ish. The integer component is stored in 7 bits which allows a range of 0 to 127 and the fractional part is stored in 8 bits and has a range between 0 / 256th to 255 / 256th. The largest fractional portion – 255 / 256ths - converted to decimal is 0.9961 hence the data range cannot quite store either -128 or +128.

As it turns out 179 / 255 is equal to 0.6992. If you could review the bits in the fractional part of variable, it would be stored as 179.

The inaccuracy introduced by a relatively small (two-byte) **Fixed Point** is irrelevant if all you are trying to do is simulate some realistic movements of characters in a game. If you have a need to calculate more accurate values, you can simply increase the size of the **Fixed Point** variable to either an SQ15x16 or SQ31x16. The code below illustrates the accuracy of each data type:

```
SQ7x8 myVar7x8 = 4;

for (uint8_t i = 0; i <= 10; ++i) {
  Serial.print(static_cast<float>(myVar7x8), 4);
  Serial.print(" ");
  myVar7x8 = myVar7x8 + 0.1;
}

SQ15x16 myVar1516 = 4;
```

```
for (uint8_t i = 0; i <= 10; ++i) {
  Serial.print(static_cast<float>(myVar15x16), 4);
  Serial.print(" ");
  myVar15x16 = myVar15x16 + 0.1;
}

SQ31x32 myVar3132 = 4;

for (uint8_t i = 0; i <= 10; ++i) {
  Serial.print(static_cast<float>(myVar31x32), 4);
  Serial.print(" ");
  myVar31x32 = myVar31x32 + 0.1;
}
```

Output:

```
4.0000 4.0977 4.1953 4.2930 4.3906 4.4883 4.5859 4.6836 4.7813 4.8789 4.9766
4.0000 4.1000 4.2000 4.3000 4.4000 4.5000 4.5999 4.6999 4.7999 4.8999 4.9999
4.0000 0.1000 0.2000 0.3000 0.4000 0.5000 0.6000 0.7000 0.8000 0.9000 1.0000
```

One thing to note with the output is the compounding effect that adding two inaccurate numbers results in. This can be seen easily if we repeat the same exercise on the smallest, one-byte data type as shown below. The fractional part of an SQ3x4 number is stored in 1/16th increments or 0.0625 in decimal. When we add the first increment of 0.1 we immediately introduce an inaccuracy as the number gets rounded down to the nearest 1/16th of a number. As we repeat the addition, the error increases. The following code shows that

```
SQ3x4 myVar3x4 = 4;

for (uint8_t i = 0; i <= 10; ++i) {
  Serial.print(static_cast<float>(myVar3x4), 4);
  Serial.print(" ");
  myVar3x4 = myVar3x4 + 0.1;
}
```

Output:

```
4.0000 4.0625 4.1250 4.1875 4.2500 4.3125 4.3750 4.4375 4.5000 4.5625 4.6250
```

Choosing larger **Fixed Point** data types increases the range of the number and accuracy of the fractional component at the expense of memory. Regardless of the data type chosen, calculations involving **Fixed Points** will always out-perform **Floating Points**!

Why would I bother with Fixed Points at all?

Imagine you are building a sideways-scrolling game where aliens (or cars, planes or other moving objects) are moving towards you at various speeds.

With each pass through the program's loop you could decrement the x coordinate of each alien but this would result in all aliens moving at the same speed. You could try to move aliens every second or third time through

the loop by keeping some sort of counter but this may result in the alien's moving too slow. To really allow flexibility in the alien's movement, you could store the x and y in fixed point variables and increment these by a decimal value allowing the aliens to move at similar but slightly different speeds.

Consider the Alien class below. Each alien is described with an x and y coordinate and takes care of its own movement.

```
class Alien {

  public:

    Alien(SQ7x8 x, SQ7x8 y, SQ7x8 incX, SQ7x8 incY, const uint8_t *bitmap) {

      _x = x;
      _y = y;
      _incX = incX;
      _incY = incY;
      _bitmap = bitmap;

    }

    move(uint8_t frame);

    SQ7x8 getX() const { return _x; }
    SQ7x8 getY() const { return _y; }

    void setX(const SQ7x8 value) { _x = value; }
    void setY(const SQ7x8 value) { _y = value; }

  private:

    SQ7x8 _x;
    SQ7x8 _y;
    SQ7x8 _incX;
    SQ7x8 _incY;
    const uint8_t *_bitmap;

);

inline void Alien:move(uint8_t frame) {

  _x = _x - _incX;
  _y = _y - _incY;
  Sprites::drawOverwrite(_x.getInteger(), _y.getInteger(), _bitmap, frame);

}
```

And in our main .ino file ..

```
Alien alien1 = {127, 24, 1.34, alienBitmap );
Alien alien2 = {127, 24, 1.76, alienBitmap );
Alien alien3 = {127, 24, 2.13, alienBitmap );

void loop() {

  alien1.move(frame);
  alien2.move(frame);
  alien3.move(frame);

};
```

With **Fixed Point** data types, we can have our aliens move at slightly different speeds without the overhead of using floating points.

Template Data Types

@Pharap's library includes the following data types. However the library is based on templates and it is a simple matter of declaring your own type - in either of the files named `FixedPointCommon.h` or `UFixedPointCommon.h` - to make it available for use.

Unsigned Data Types:

Data Type	Size	Integer Range	Fractional Resolution	Effective Range (Exclusive)
`UQ4x4`	8 bit (1 Byte)	0 – 15	1 / 16th	0 to 16
`UQ8x8`	16 bit (2 bytes)	0 to 255	1 / 256th	0 to 256
`UQ16x16`	32 bit (4 bytes)	0 to 65,355	1 / 65,536th	0 to 65,356
`UQ32x32`	64 bit (8 bytes)	0 to 4,294,967,295	1 / 4,294,967,296th	0 to 4,294,967,296
`UQ1x7`	8 bit (1 byte)	0 to 1	1 / 128th	0 to 1
`UQ1x15`	16 bit (2 byte)	0 to 1	1 / 32,768th	0 to 1
`UQ1x31`	32 bit (4 byte)	0 to 1	1 / 2,147,483,648th	0 to 1
`UQ1x63`	64-bit (8 byte)	0 to 1	1 / 9,223,372,036,854,775,808th	0 to 1

Signed Data Types

Data Type	Size	Integer Range	Fractional Resolution	Effective Range (Exclusive)
`SQ3x4`	8 bit (1 Byte)	-15 to +15	1 / 16th	-16 to +16
`SQ7x8`	16 bit (2 bytes)	-127 to +127	1 / 256th	-128 to +128
`SQ156x16`	32 bit (4 bytes)	-32,767 to +32,767	1 / 65,536th	-32,768 to +32,768
`SQ31x32`	64 bit (8 bytes)	-2,147,483,647 to +2,147,483,647	1 / 4,294,967,296th	-2,147,483,648 to +2,147,483,648
`SQ1x6`	8 bit (1 byte)	-1 to +1	1 / 128th	-2 to +2
`SQ1x14`	16 bit (2 byte)	-1 to +1	1 / 32,768th	-2 to +2
`SQ1x30`	32 bit (4 byte)	-1 to +1	1 / 2,147,483,648th	-2 to +2
`SQ1x62`	64-bit (8 byte)	-1 to +1	1 / 9,223,372,036,854,775,808th	-2 to +2