MAP
Assignment 2 Report

Jack Walters

---

1. Introduction

This report accompanies a drum machine written in C++ for the Bela Board for Assignment 2 in the Music and Audio Programming module.

2. render.cpp

2.1. Initialisations

Up until the setup() function in `render.cpp`, all code is concerned with including relevant header files and initialising global variables and data. Arrays being initialised, such as `gReadPointers`, `gDrumBufferForReadPointer` and `movingAverageArray` enable the processing of related information (i.e. X, Y and Z axis acceleration values from the accelerometer) in an efficient manner. These arrays also lend themselves to unambiguous code structures in following audio processing functions. There are also a number of counters being initialised such as `gMetronomeCounter`, which play a crucial role in the timing and synchronicity of strictly time-dependent audio functions inside `render()`. Also present are a number of Boolean-like variables of type integer such as `gPlaysBackwards` which serve as process alteration markers to functions in the project. For sake of legibility of the code, two enumerations are used; both of these enumerations are used in state machines inside `render()`. In the same vein of code comprehensibility, digital and analog pin numbers are also assigned to variables. Just before `setup()`, two high-pass filter objects of class type `Filter` (see *4.2. filter.h*) are instantiated, which are used to high-pass accelerometer Z-axis values for use in triggering the drum fill functionality implemented in `startNextEvent()`.

2.2. setup()

Inside `setup()` audio-dependent assignments of variables and data-structures initialised pre-`setup()` are implemented. 2 Digital pins are assigned as input and output respectively, so that the button can be used to trigger a user-specified drum sound and the LED displays the metronome tempo. The pointer declared pre-`setup()` for audio buffer `audioOutputBuffer`, is declared as dynamic memory to be the same length as the frame size of the Bela Context and then it is initialised to 0, in preparation for its use in `render()`.

Three `for()` loops iterate through and set the indices to 0 of arrays initialised in pre-`setup()`. This is displayed in listing 1:

LISTING 1: Iterative setting of arrays to 0

```
1  for (int i = 0; i < 16; i++) {
2      gReadPointers[i] = 0;
3      gDrumBufferForReadPointer[i] = -10;
4  }
5
6  for (int i = 0; i < movingAverageOrder; i++) {
7      for (int j = 0; j < 3; j++) {
8          movingAverageArray[i][j] = 0;
9      }
10 }
11
12 for (int i = 0; i < 3; i++) {
13     accelerometerPinReference[i] = 0;
14     accelerometerPinMapped[i] = 0;
15     movingAverageResult[i] = 0;
16 }
```

Below these `for()` loops, the sample rate obtained from the Bela Context is passed into the method `setSampleRate` inside the `Filter` class, of which `Filter1` and `Filter2` are a type (see *4.2. filter.h*).

2.3. render()

The entirety of the contents of the `render()` function are encapsulated inside a `for()` loop which iterates from the sample 0 of the buffer to the end of it. Immediately proceeding this `for()` loop are 3 functions that read values from the analog inputs at frame `n/2` (Fs/2); The analog pins on the Bela board are sampled at this rate because the default analog sample rate is half of the audio sample rate. Analog pins 0-4 read the on/off toggle switch, the metronome tempo and the accelerometer X, Y and Z values respectively. The accelerometer pins voltage values are stored in private variables to `render()` and subsequently converted to acceleration values , the `metronomeRead` pin acts as a toggle switch, triggering the Boolean-type integer value `gIsPlaying` if the value is above 0.5. Float variable `tempoRead` is also updated through the `analogRead()` function, however its value is mapped to a millisecond value and subsequently to a sample value, which is used to govern the point at which the `gMetronomeCounter` resets (i.e. the tempo at which the metronome operates and the LED turns on and off).

Between lines 210 and 234, the code is concerned with storing and converting values from the accelerometers. Because of the discrepancy between Bela devices and resistors, after 10 samples the voltage readings of the three axis readings from the accelerometer are

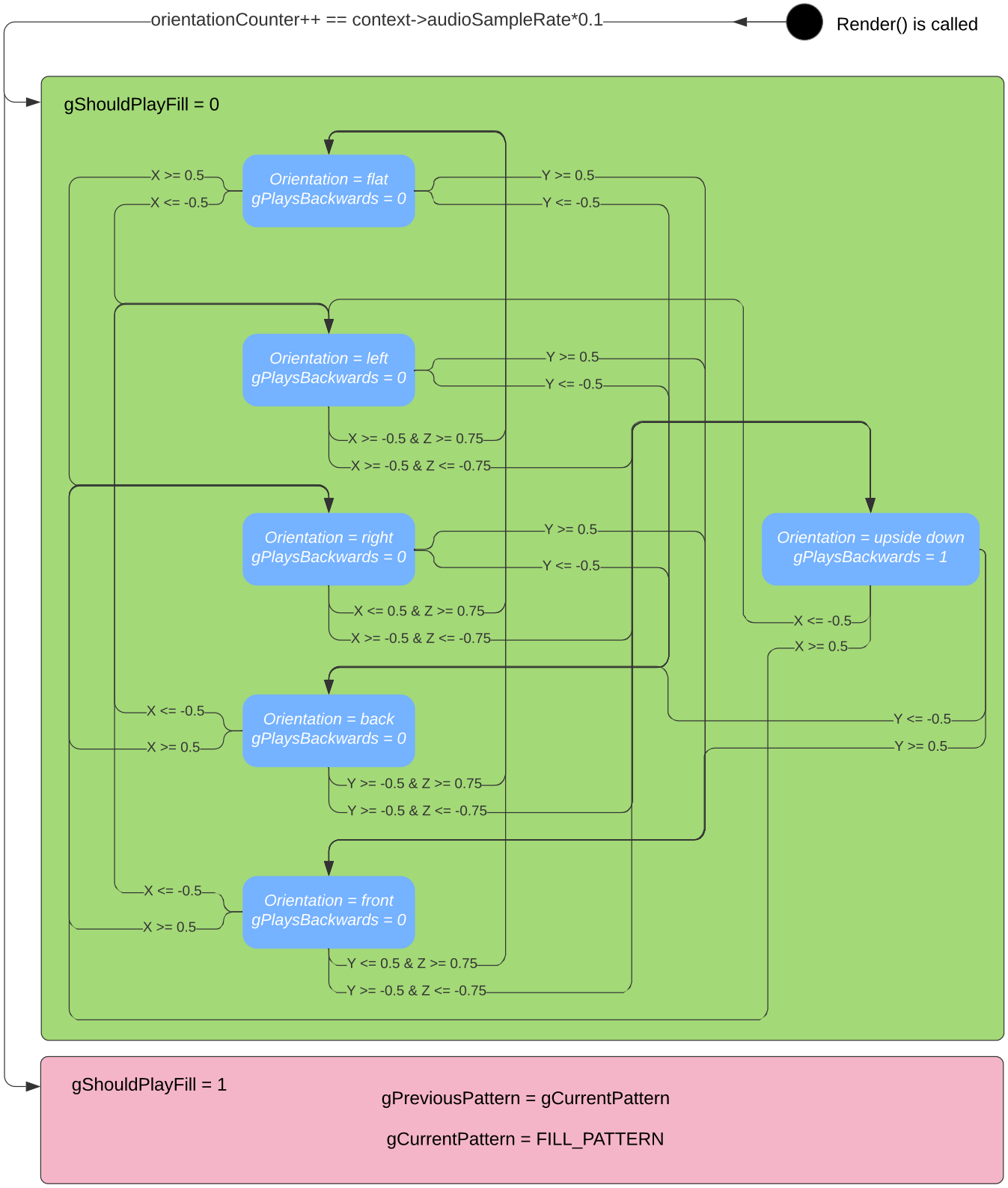

FIGURE 1: Orientation State Machine

stored as references to ensure when the board is laid flat, the X, Y and Z readings are as close to 0, 0 and 1 respectively as possible. These reference values are then used in the calculation of acceleration values: For all axes we will be reading for 90° shifts, which corresponds to a 0.25 change in terms of analog input voltage (normalised between 0 and 1 through the `analagRead()` function). Therefore, for the X and Y readings our `in_min` and `in_max` parameters for the `map()` functions are 0.5* and 1.5* the reference reading (when the board is flat); these values are then mapped between -1 and 1. The Z value is also mapped between -1 and 1, however its reference value is the equivalent of +1g acceleration due to gravity, and a 180° rotation to a acceleration value of -1g is in theory roughly the original reference value minus itself (experimentally I found this to be `accelerometerPinReference[2]`* 0.27). Having to hard code scaling factor of 0.27 into the `in_max` argument of the `map()` function will affect the portability of this code to other Bela boards, but it was unfortunately required for smooth running of the project on my Bela board. The scaled acceleration values from the X, Y and Z axes of the accelerometer are then passed into the `movingAverage` function (see *2.4. Additional Functions*) and stored in the `movingAverageResult` array.

Below this mapping structure the two filter objects instantiated pre-`setup()` are cascaded in series to create a 4th-order biquad filter. `Filter1` takes sample values from the `movingAverageResult` array and outputs 2nd-order high-passed samples, these samples are then passed into `Filter2` and high-passed again, outputting 4th-order high-passed samples. Through an `if()` statement, the code checks whether the high-passed samples are above a certain threshold and if they are, `gShouldPlayFill` is set to 1. Together two counters `tapInitialiseCounter` and `tapCounter` ensure this process happens after 1 period of the sample rate, at Fs/10 intervals. The reason for only enabling tap readings to be taken after 1 second, is that when the code is first run there is a spike in the Z axis due to the initial acceleration gravity exerts on the accelerometer, before it then stabilises. If not accounted for, this would lead to the code consistently falsely identifying drum-fill inducing taps from the user in its first second of being run.

The number of counters in this section do not seem like the most stable design choice, but there was no other way I could think of calling the code that was time-dependent. This project also utilises two state machines: one enables simple on/off state variance for the button, ensuring that the machine is in `stateOn` for a finite interval dictated by `debounceInterval` and calls the `startPlayingDrum()` function while it is in this state. The other state machine depicted in Figure 1 is more complex, and governs the orientation state of the board. Once Fs/10 has passed, the state of the `gShouldPlayFill` global variable either bypasses or enables this state machine and the X, Y and Z values control the switching between states.

Listing 2 exhibits the audio process block. First the primary audio buffer `out` is reset upon every iteration of the initial `for()` loop, and below it another `for()` loop iterates from 0 to 16. This ensures that upon each increment in sample inside `render`, all 16 indexes of `gReadPointers` and `gDrumBufferForReadPointer` are read through. If `gDrumBufferForReadPointer` at index `i` is free (i.e. has a value of -10), the contents of the drum buffer `gDrumSampleBuffers` at this available index of `gDrumBufferForReadPointer` is copied to the secondary audio buffer `audioOutputBufffer`.

Listing 2: Audio Process Block

```
1  float out = 0;
2  for (int i = 0; i < 16; i++) {
3      if (gDrumBufferForReadPointer[i] != -10) {
4          audioOutputBuffer = gDrumSampleBuffers[gDrumBufferForReadPointer[i]];
5          if (gPlaysBackwards == 0) {
6              out += audioOutputBuffer[gReadPointers[i]++];
7              if (gReadPointers[i] >= gDrumSampleBufferLengths
8              [gDrumBufferForReadPointer[i]]) {
9                  gDrumBufferForReadPointer[i] = -10;
10             }
11         } else if (gPlaysBackwards == 1) {
12             out += audioOutputBuffer[gReadPointers[i]--];
13             if (gReadPointers[i] <= 0) {
14                 gDrumBufferForReadPointer[i] = -10;
15             }
16         }
17     }
18 }
19
20 for(unsigned int channel = 0; channel < context->audioOutChannels; channel++) {
21     audioWrite(context, n, channel, out);
22 }
```

Following this, the `if()` statements on lines 5 and 11 checks the global variable `gPlaysBackwards` (which would have been triggered by the orientation pattern selection state machine being `upsideDown`) to see whether the function should be counting positively or negatively through the buffer indices. Primary audio buffer `out` has all 16 indices of `gReadPointers` added to it through compound addition. Depending on whether `gPlaysBackwards` is 0 or 1, two further nested `if()` statements on lines 7 and 13 check whether the end or beginning of the buffer has been reached: if so this `gDrumBufferForReadPointer` at index `i` is reset to -10, after having been set by the `startPlayingDrum()` function. The `for()` loop on line 20 iterates through to the number of channels in the Bela Context, adding audio buffer `out` to all channels at index `n`, incremented by the initial `for()` loop in `render()`.

2.4. Additional Functions

The `startPlayingDrum()` function is passed a drum index from the `startNextEvent()` function, and subsequently iterates through all 16 indices of `gDrumBufferForReadPointer` and `gReadPointers`. In its `for()` loop, when an index of `gDrumBufferForReadPointer` is read that is inactive (i.e. has a value of -10), this index is assigned the value of the drum index. This array structure means that a maximum of 16 drum sounds can be played simultaneously, and when drum sounds are played simultaneously they do not interfere with each other. Depending on the value of `gPlaysBackwards`, the `gReadPointers` at the same index is reset to either 0 (to preempt the buffer playing forwards) to the length of the buffer (to preempt the buffer playing backwards).

The `startNextEvent()` function shown in Listing 3 enables the playing of the next drum sound in the pattern by sending the `drumindex` to `startPlayingDrum`.

Listing 3: startNextEvent()

```
1  if (toggle == 1) {
2      for (int i = 0; i < NUMBER_OF_DRUMS-1; i++) {
3          if (eventContainsDrum(gPatterns[gCurrentPattern]
4          [gCurrentIndexInPattern], i) == 1) {
5              startPlayingDrum(i);
6          }
7      }
8      gCurrentIndexInPattern = (gCurrentIndexInPattern + 1 +
9      gPatternLengths[gCurrentPattern]) % gPatternLengths[gCurrentPattern];
10
11     if (gCurrentIndexInPattern == 0) {
12         if (gCurrentPattern == FILL_PATTERN) {
13             gShouldPlayFill = 0;
14             gCurrentPattern = gPreviousPattern;
15         }
16         gCurrentIndexInPattern = 0;
17     }
18 }
```

If `gIsPlaying` (passed into this function as `toggle`) is 1, then iterate through number of drum buffers. The `eventContainsDrum` function on line 3 has the specific index of specific drum pattern, in relation to drum at index `i` and returns either a 1 or 0 depending on if a drum is present at the aforementioned index. The `if()` statement on line 3 then calls `startPlayingDrum` at the same index if `eventContainsDrum` shows up positive. To ensure the current pattern index wraps around the total length of the pattern, modulo arithmetic is used on line 9. When the modulo equates to 0 (i.e. `gCurrentIndexInPattern` and then length of the current pattern are the same), `gCurrentIndexInPattern` is reset 0, and depending on if the pattern was the `FILL_PATTERN` triggered by the board being held upside down, the pattern playing prior to the fill is reinstated.

Finally, the `movingAverage()` function takes two arguments: the current sample `Xn` and the `arrayIndex`. Depending on `arrayIndex`, the either the X, Y or Z axis from the accelerometer with be being averaged. Global variable `movingAverageOrder` dictates the order of the filter, and hence the size of the array that is summed and averaged at every sample.

3. main.cpp

Inside `main.cpp`, drum samples, sample lengths, patterns and pattern lengths are loaded into buffers. The private variables in the `Filter` class outlined in `filter.h` are also initialised with values through its constructor and `setSampleRate` methods. These methods call the `calculateCoefficients` method which calculate 2nd order high-pass transfer function coefficients that were derived through the bilinear transform. The `Filter` class's method `highPass` outputs high-passed samples based on its input and previous states of input and output values and is invoked through the objects `Filter1` and `Filter2` inside `render()`.

4. Header Files

4.1. drums.h

The `drums.h` header file contains declarations of the methods `startPlayingDrum`, `startNextEvent`, `eventContainsDrum` and `movingAverage` all initialised and called inside `render.cpp`.

4.2. filter.h

This header file contains declarations of both public and private methods initialised in `main.cpp` and called directly and indirectly in `render.cpp`. Private variables for the sample rate, frequency, Q, coefficient values and previous input/output states are also declared in this file. This code was partially inspired by Andrew McPherson. [1]

5. Conclusion

Overall I believe the implementation of this project to be successful. Drums buffers sound simultaneously with no interference, the drum pattern changes responsively depending on the orientation of the board, further drum buffers can be triggered manually by the user, and the user can toggle the pattern on/off as well as adjust its tempo. I believe that a sufficiently minimal amount of values in code have been hard-coded and in general the project would be portable to other Bela Boards provided the correct equipment is available. However, perhaps my biggest issue with the implementation was overcoming board

and resister variance when it came to taking accelerometer readings. Having implemented certain scaling factors that are inevitably due to my specific board means that the project's portability is impacted. This may well lead to the project not performing as well on another user's equipment. Furthermore, having to disable the tap drum-fill functionality for the first second due to the initial spike in the accelerometer Z axis value is a design decision that limits the capacity (albeit for only 1 second) of the board but at present moment seems like a necessary restriction. Having certain initialisations and time-dependent functions rely on their own individual counters does seem somewhat unreliable, but it does ensure I am not oversampling readings and that certain values are not being read when they ought not to be. This project lends itself to further developments whereby more aspects of the pattern can be controlled through buttons and potentiometers on the breadboard and drum fill patterns could vary stochastically dependent on the board's orientation.

References

[1] Andrew McPherson. *filter.h*, 2020 (accessed April 1, 2020).