PS5B

GUITAR HERO: GUITARSTRING IMPLEMENTATION AND SFML AUDIO OUTPUT (PART B)

In Part B, we implement the **Karplus-Strong** guitar string simulation, and generate a stream of string samples for audio playback under keyboard control.

See http://www.cs.princeton.edu/courses/archive/spr15/cos126/assignments/guitar.html for the full assignment.

GUITARSTRING IMPLEMENTATION

```
GuitarString(double frequency) // create a guitar string of the given
frequency
                                //   using a sampling rate of 44,100
GuitarString(vector<sf::Int16> init) // create a guitar string with
                               //   size and initial values are given
by the vector
void pluck()                    // pluck the guitar string by replacing
the buffer
                              //   with random values, representing
white noise
void tic()                    // advance the simulation one time step
sf::Int16 sample()            // return the current sample
int time()                    // return number of times tic was called
so far
-------------------------------------------------------------------
-
```

Notes

- In the `GuitarString` private member variables declarations, you must declare a pointer to a `RingBuffer` rather than declaring a `RingBuffer` object itself. Then in the `GuitarString` constructor you must use the `new` operator.
  - This is because you can't allow the ring buffer to be instantiated until the `GuitarString` constructor is called at run time (you don't know how big a ring buffer to make until given the frequency of the string).
  - See http://stackoverflow.com/questions/12927169/how-can-i-initialize-c-object-member-variables-in-the-constructor for an explanation.
  - Because the ring buffer contained in the guitar string class will be a pointer to a

- In the `pluck` method, you must fill the guitar string's ring buffer with random numbers over the **`int16_t`** range. **`int16_t`** is a short integer, which can hold values from -32768 to 32767.

Here is a snippet of code that can be used to generate a random number in this range:

```
(int16_t)(rand() & 0xffff)
```

- Also in **`pluck`**, the guitar string's ring buffer might already be full. So you should either empty it (by dequeuing values until it's empty), or by deleting it and making a new one which you'll then fill up.

Or, you could add a new method to your ring buffer, **`empty()`**, which would set the **`_first`** and **`_last`** index member variables to 0, and the **`_full`** boolean to **`false`**. (This would be the most efficient solution.)

Testing your GuitarString implementation

Before you proceed to generate sound, test that your **`GuitarString`** is implemented correctly!

Do this by compiling it against this test file: GStest.cpp. Build instructions are at the top of the file.

SFML AUDIO OUTPUT

There are two parts of generating audio: (1) getting values out of the **`GuitarString`** object and into SFML audio playback object, and (2) playing the audio objects when key press events occur

This is an elegant solution and allows us to mix together signals from two (or more) **`GuitarString`**s by averaging their values. (This is based on a similar superposition theorem that we used in the 2D physics simulation.)

For SFML, we have to have an existing **`sf::SoundBuffer`** that's created with a vector of sound samples. This SoundBufferis created from a vector of **`sf::Int16`**s.

Then we create an **`sf::Sound`** object from the **`sf::SoundBuffer`**. The **`sf::Sound`** object can then be played.

So the whole sequence is:

Playing SFML Sounds when key presses occur

We'll use SFML to create an electronic keyboard:

- When the “a” key is pressed, a sound corresponding to concert A (440 Hz) should be played.
- When the “c” key is pressed, a C note should be played.

To handle the keypress events, we'll open an SFML window, and look for **`sf::Event::KeyPressed`** events.

When we get one, we'll see if its **`event.key.code`** is equal to **`sf::Keyboard::A`** or **`sf::Keyboard::C`**.

If so, we'll play the appropriate sound.

See the sample code below for how to do this.

GuitarHeroLite.cpp demo file

In the second half of the code, an SFML window and event loop is set up to play the sounds when the "a" or "c" keys are pressed.

This file may be downloaded here: GuitarHeroLite.cpp.

YOUR ASSIGNMENT

Once you have your `GuitarString` class implemented, extend `GuitarHeroLite` starter code per the Princeton assignment.

Follow the instructions that begin with the statement

"Write a program `GuitarHero` that is similar to `GuitarHeroLite`, but supports a total of 37 notes on the chromatic scale from 110Hz to 880Hz."

Notice the statement

"Don't even think of including 37 individual `GuitarString` variables or a 37-way if statement! Instead, create an array of 37 `GuitarString` objects and use `keyboard.indexOf(key)` to figure out which key was typed."

For our implementation, we actually need three parallel arrays (please use vectors):

- a vector of 37 `sf::Int16` vectors. Each individual `sf::Int16` vector holds the audio sample stream generated by one `GuitarString`.
- a vector of 37 `sf::SoundBuffers`. Each `SoundBuffer` object contains a vector of audio samples.
- a vector of 37 `sf::Sounds`. Each Sound object contains a `SoundBuffer`. (It's the Sound object that can finally be played.)

- Your `RingButter.cpp` and associated `RingButter.hpp`
- Your `GuitarString.cpp` and its `GuitarString.hpp`
- Your `GuitarHero.cpp` file
- A `Makefile` that builds an executable named `GuitarHero`.
- A `ps5b-readme.txt`

Submit using the submit utility as follows:

submit schakrab ps5b ps5b

EXTRA CREDIT

For extra credit, make a version of the program that makes a different sound. Modify the algorithm to get a sound that resembles drum, chirp, piano, or anything other than the guitar.

This sound doesn't have to simulate a specific instrument. Here's a couple of ideas:

1. Make your algorithm vary the number of samples on the queue as the sound is being synthesized, producing a frequency chirp. For example, for each 100 times that `tic()` is called, remove 100 samples from the queue, but only re-insert 99 samples. This will produce an up-frequency chirp (make sure to stop removing samples when the queue is almost empty, so that `peek()` and `dequeue()` don't throw exceptions for empty queue.)
2. Change the low-pass filter so it leaves some of the noise in the buffer for longer, resulting in a "noisier" sound - this will sound more like a percussion instrument. One way to do this is to mix 90% of the last sample and 10% of the second-last sample (guitar sound uses 50%/50% mix.)

(evidence that your implementation passes the `GStest.cpp` tests)

GuitarHero player implementation: 4

(transforming the `Lite` version into the full 37-note player per assignment)

Makefile: 1

(Makefile or explicit build/link instructions included)

Readme: 2

(discussion is expected -- at least mentioning something per section.)

Total: 12

Extra credit: 2 : Make a version of the program that makes a different sound. Modify the algorithm to get drum, chirp, piano, or anything other than the guitar