Audio Visualizer – Documentation

Introduction

This documentation describes classes and their functionalities in a project called Audio Visualizer that was made as a semester project to the beginner class of C++.

Using the application

To compile this application, one must use Visual Studio 2019 or newer on Windows. User documentation is described in README.md file.

Classes overview

The application is divided into four main parts. First part is menu. This part takes care of the main menu screen and processing input from the user. Second part are modes. In this part all the visualization logic is implemented. Next part is called visualizer. This part connects menu, modes and song database. Which is the fourth part.

Each part implements its drawing to the screen. Part called visualizer only decides what part will be drawn next.

Menu Class

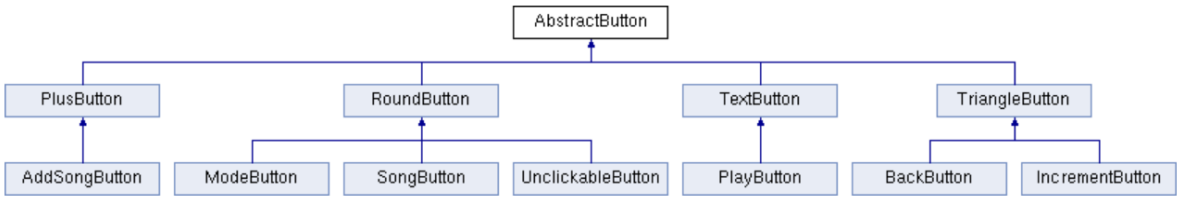
This is the first part of the application. Currently active buttons, loaded songs and modes are stored in this section. This class has its own song database instance where songs are stored.

Each button from the screen is saved and drawn separately. However, buttons are grouped to three categories. Song buttons contain only buttons that represent songs that can be played. Mode buttons represent modes that can be visualized. And lastly all other buttons such as play, next page, add or previous page button. Each song and mode have its own button whose properties are set on initialization of the class Menu.

Function	Description
void set_mode()	Sets mode in a field *chosen_mode* as currently active, so the corresponding button of this mode is drawn highlighted. Field *chosen_mode* is selected by Vizualizer section on the command of the user.
string get_mode()	Returns currently active mode in a field *chosen mode*.
void set_song()	Sets song in a field *chosen_song* as currently active, so the corresponding button of this song is drawn highlighted. Field *chosen_song* is selected by Vizualizer section on the command of the user.
string get_song()	Returns currently active song in a field *chosen_song*.
void add_song (**string** song)	Adds a new song with a name *song* to the database. This is done by first adding the song name to the *song_database.txt*, then by reloading all of the song buttons completely.
void load_song_buttons()	Deletes all current song buttons, loads names of all the songs from *song_database.txt* file and sets correct positioning of new buttons. Those buttons can be then displayed on the menu screen. This is used when adding a new song and at the initialization of the menu.
void change_page(**int** i)	Moves page counter called *song_page* for *i* positions. This is used when clicking buttons to cycle through pages of songs.
void update_buttons()	Updates song buttons based on the current *song_page* field.

Void draw(**RenderWindow** window)	Draws all the buttons and sprites to the *window* using sfml draw function.
void restart()	Reopens the menu. Redraws all menu buttons, starts playing the menu song. This is used after the user comes back from the mode visualization.
void quit()	Quits the menu screen. The menu song stops playing and any of the menu buttons is drawn. This is used when the user starts the song visualization.

For the purpose of nicer UI whole button hierarchy was implemented.



Each button consists of few drawable objects such as circle, rectangle, triangle or just plain text. Each button has unique functionality. Buttons hierarchy has three levels. First level implements common functionality every button must have. Second layer determines the shape. Third layer determines behaviour after being clicked.

First layer

bool is_focused(**Vector2i** vector)	Returns true, if *vector* is at the position of the button.
void set_text_color(**color** color)	Sets the draw colour of the text of the button.
string get_text()	Returns text that is displayed on the button.
void set_font(**font** font)	Sets the font of the text of the button.
void set_text_size(**int** size)	Sets size of the text.
void set_text_string(**string** text)	Sets the text content of the button.
void set_button_color(**color** color)	Sets the draw colour of the button.
void draw(**renderwindow** window)	Draws the button to the position of *x* and *y* properties of the button.
void set_position(**float** x, **float** y)	Sets the *x* and *y* properties of the button that it will be drawn at.
void activate(**menu** menu)	Runs the functionality of the button. This is characteristic for each buton.

Second layer

This layer only implements what drawable objects is the button composed of and how is it positioned relatively to its *x* and *y* fields.

Button classes

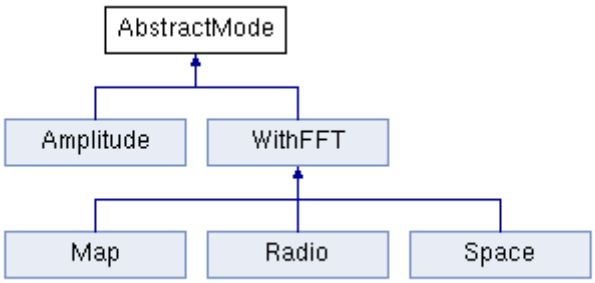
Last layer determines the behaviour when clicked.

AddSongButton	Implements *add_song* function from the Menu class.
ModeButton	Sets the *chosen_mode* field in menu class to clicked mode. Each mode button represents different mode.

SongButton	Sets the *chosen_song* field in menu class to clicked song. Each song button represents different song.
UnclickableButton	This button does nothing when clicked. Purpose of it is just to simplify the UI. “Song names” or “Mode names” buttons are implemented by this type of button.
PlayButton	This button initializes and runs the visualization of chosen song and mode.
BackButton	This button terminates visualization and restarts menu with its restart function.
IncrementButton	This button either increments or decrements *song_page* field in menu class using its *change_page* function.

Modes class

Modes have unique hierarchy.



First level implements common properties of every mode. Second level divides modes to those who need fast Fourier transformation for their visualization and those who don’t. Leaves represent modes that can be used to visualize a song.

AbstractMode class

This class contains fields needed for proper sound processing such as buffer, sample rate and sample count.

void draw(**RenderWindow** window)	This function draws all the drawables of a specific mode to the window.
void update()	This function is ran every tick and updates properties of all drawables so it changes over time.

WithFFT class

This class implements all fast Fourier transformation function, creation of a hamming window and helper functions for modes that use fast Fourier transformation.

This function implements a lot of functions so that creation of new modes is easy, and those functions can be reused in many modes.

All functions (with exception of *frequency_spectrum_lr*) work on logarithmic scale of frequencies. That means that more the lower the frequency the larger gap between each sample that is drawn.

Because human ears are used to this logarithmic scale, it feels more natural to show more lower frequencies than higher.

In the visualizing functions data from the song buffer after the fast Fourier transformation are divided by very large numbers. That is so the amplitudes are not as high, because this data is directly converted to the amplitudes of frequencies that are being visualized. Those constants were computed by manual testing, so the visualization provides perfect results.

void fft(**ComplAr** data)	Runs fast Fourier transformation on *data*. Classic algorithm.
void create_hamming_window()	Creates hamming window for smoother visualization after Fourier transformation. This window is stored and is used at each update call on song buffer before applying fast Fourier transformation on the song buffer.
void generate_bars_lr_up(**VertexArray** VA, **Vector2f** starting_position)	This function sets positions of vertices in *VA* so that they create bar-like formation representing loudness of each frequency. Bars are made from one point at the base and one point that is higher. The relative height of second point corresponds to the dominance of specific frequency. Each frequency has its own pair of points (frequency count is limited by the size of samples and sampling rate). Those points are connected forming a line. Those bars go from left to right (lower frequencies on the left, higher on the right) and are bouncing upwards. That explains the name of the function. Logarithmic distribution makes it look like there are bars, not just lines. Because the same frequency is visualised more times. The lower frequencies might have around tens of pairs of points whilst higher frequencies only have one pair.
void generate_bars_lr_down(**VertexArray** VA, **Vector2f** starting_position)	This function is the very same copy of the *generate_bars_lr_up*. The only difference is that the bars are bouncing downwards.
void generate_line_lr_up(**VertexArray** VA, **Vector2f** starting_position)	This function is very similar to the *generate_bars_lr_up* one. The only difference is that there is always only one point per frequency. This point is the same as the one that is higher in the *generate_bars_lr_up* function. Those points are then connected forming a line. This line copies the silhouette of bars generated by *generate_bars_lr_up*. The line is from left to right (lower frequencies on the left, higher on the right) bouncing upwards. That explains the name again.

void generate_line_lr_down(**VertexArray** VA, **Vector2f** starting_position)	This function is the very same copy of the *generate_lines_lr_up*. The only difference is that the line is bouncing downwards.
void frequency_spectrum_lr(**VertexArray** VA, **Vector2f** starting_position)	This is function is raw visualization of frequencies without being scaled logarithmically. Otherwise it works on the same principle as *generate_bars_lr_up*.
void frequency_spectrum_round(**vector<VertexArray>** VAs,**vector<Color>** colors ,**vector<float>** heights, **Vector2f** center, **float** radius, **int** from, **in**t to)	This function generates area of frequencies forming a circle with the centre in *center* of a radius of *radius*. This circle has two identical sides. Frequencies visualized are starting at the value of *from* and end in the value of *to*. This area consists of *n* layers depending on the number of *heights*, and *colours* put in a vector as a parameter. Each part of the circle represents part of the frequency spectrum set in parameters. The frequency spectrum is further from the centre the more dominance is put to that specific frequency in data in the song buffer. Each layer has its own colour. The bottom layer is at index 0, the top layer is at the last position at the vectors. Array of vertices at vector at index *i* corresponds to the colour at vector of colours at index *i*. Simply put this function only generates two bar lines curving them around a circle. With common base for all the vertices in the centre. This is done for all the layers only with different maximum height the amplitude can go to.
void generate_map(**VertexArray** VA, **Vector2f** starting_position)	This function operates on the same principle as *generate_line_lr_up* function. The only difference is that the points are not connected so they don't create a line. They are moved slightly closer to the right top corner as the time passes. It creates map-like scenery. This is done by calculating amplitudes of frequencies that are logarithmically scaled. After each update the previous points are moved few pixels up and to the right and redrawn. This is repeated each update.

Amplitude

This mode draws raw song buffer values to the screen. Song buffer is using the sfml library function. Position for each vertex is set to the raw amplitude that is read from the buffer and draw. Vertices are connected forming a line.

Radio

This function uses *generate_bars_lr_up* and *generate_bars_lr_down* to create radio-like scene.

Amplitude of the bars correspond to the dominance of each frequency scaled logarithmically. Each bar section (the one facing up and the on facing down) has its own array of vertices that are update each tick.

Map

Function map uses *generate_lines_lr_up* function to generate current representation of frequency dominance each tick. Then it is moved closer to the top right corner and the colour is dimmed. The map is stored at single array of vertices. After the size of the array is full, the space of the last set of vertices is removed and replaced by the currently new one.

Space

This is the most complex mode.

It consists of stars shooting from the centre to the edges. That is done by generating vertices at random positions on the screen and updating their position each tick. These positions are moving away from the centre. Each tick the last position of a star is remembered and connected with its current position. Creating a tail of the shooting star. The speed of the stars is a constant multiplied by a value linked to frequencies from 0 to 60. Simply the sum of amplitudes of those frequencies is computed and divided by constant, that puts it into acceptable speed limits. Around 1.

Then there is a planet in the centre. Its radius is again linked to the frequencies from 90 to 280. This radius change is again computed by sum of amplitudes of frequencies in the range divided by a constant. Again, this constant was chosen by manual testing. Rotation speed of this planet is linked to frequencies 500 to 3000.

Finally, the hallo around the planet is represents amplitudes of frequencies from 20 to 80. The halo consists of three layers. Each has its own colour and maximum height they can reach. This maximum is a part of a constant that divides data from the sound buffer that affects the height of amplitudes. The computation of the amplitudes works the same here as it does in the Radio mode. The only difference is that the positions are curved here.

Vizualizer class

This is the neural system equivalent of the application.

Visualizer has its menu class instance It calls functions on this instance based on input from the user. It initializes mode when requested by the user, it changes the *chosen_song* field in the menu when requested.

All the user input is processed in the *run()* function. This function is basically one while loop that terminates after closing the application. It orders the window to render every loop.

SongDatabase class

This is simply helper class for easier manipulation with the songs in the application.

It protects and processes all queries on the songs stored in the *song_database.txt* file.