QTunes

By: SHAWN MATHEW, VARAN SHARMA, AND AMY WONG

Overview

Purpose

We set out on a journey to design a music player. We wanted graphics such as cover flow and a visualizer along with the basic functions of a music player. There were a few obstacles that we came across, but with the help of a debugger and the teaching assistant, Siavash Zokai, we were able to overcome these obstacles. It was a very worthwhile experience and definitely one that has advanced our skills as computer scientists. This will definitely help us in the future due to the fact that we are learning libraries that are highly used in the industry today.

Libraries

The Qt library was the library we used to create the GUI interface. It had a very integral part in creating the layout, buttons, and other features in our music player. Qt is a cross platform library making our application applicable on most platforms. The library was initially created by Trolltech. The company was acquired by Nokia where Qt continued to be developed. It was then sold to its current owner, Digia. The most current version of Qt is called Qt 5.

The OpenGL library was a major part of our music player. It was the library that we used to create the coverflow and music visualizer. OpenGL is an open source graphics library that has various versions that will run for phones, web browsers, and computers. OpenGL was created by Silicon Graphics Incorporated. It was initially called IRIS GL, but when it became an open standard library it was renamed to OpenGL.

The TagLib library was mainly used to grab and edit the metadata from the various audio files being imported into our music player. In this application, TagLib's ability to read metadata from audio files were useful in getting the audio, song names, and other song information out of the selected files.

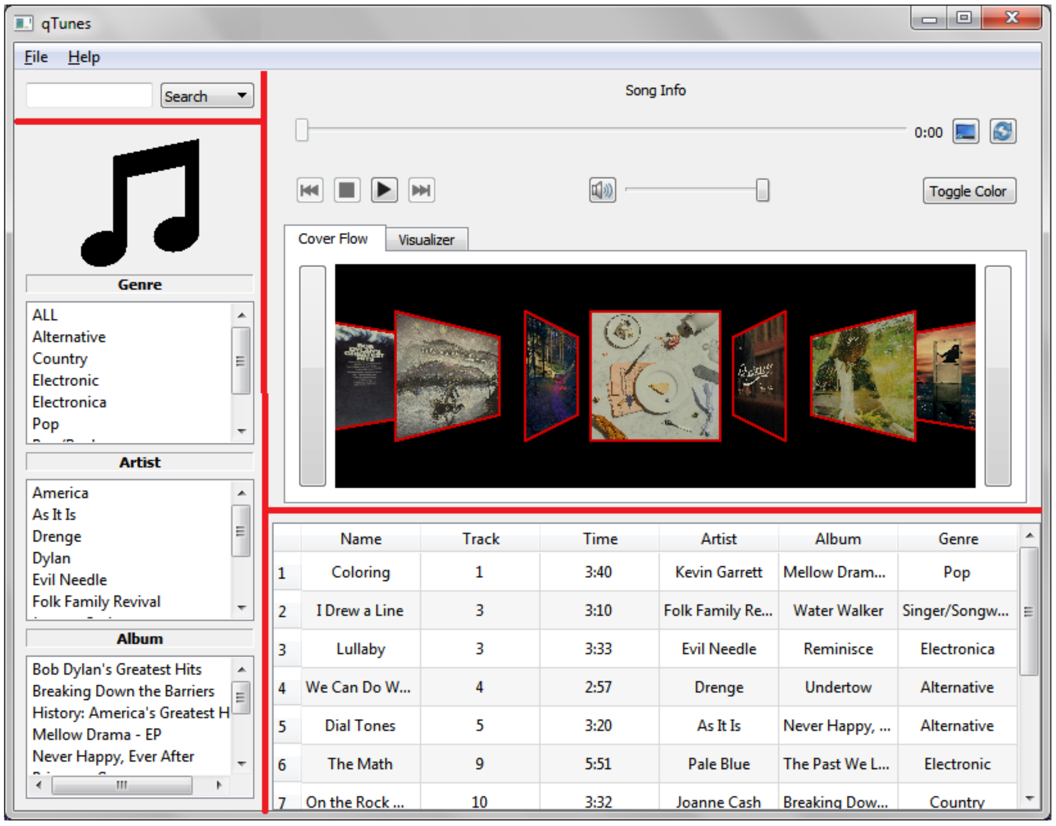
Components

The main features of our music player are the graphics and our interface. We shall begin by describing our interface including the various buttons, widgets, and layout that were incorporated in making our music player. From there we will discuss the two graphics that were included. Cover flow will explain how each album cover was sent into the glWidget class and made displayed onto the squares in cover flow. The way in which these squares were put into place will also be explained in this section. The music visualizer was the second graphic that was included in the application. In this section, the way in which the bars were drawn and the logic behind the falling of the bars will be explained.

MainWindow

Layout

The interface of the music player application allows user interaction. The layout is created in the `MainWindow` class. It contains splitters (represented as red lines in the figure below) that allows the user to stretch or hide some of the objects(also called widgets). For instance, the main splitter denoted by the red vertical line in the figure separates the layout into two sections, left and right. When the splitter is moved, one section is horizontally stretched while the other section is horizontally compressed. Each of the section contains a splitter that will act in a similar fashion except that height of the object in that section will be compressed or expanded instead.

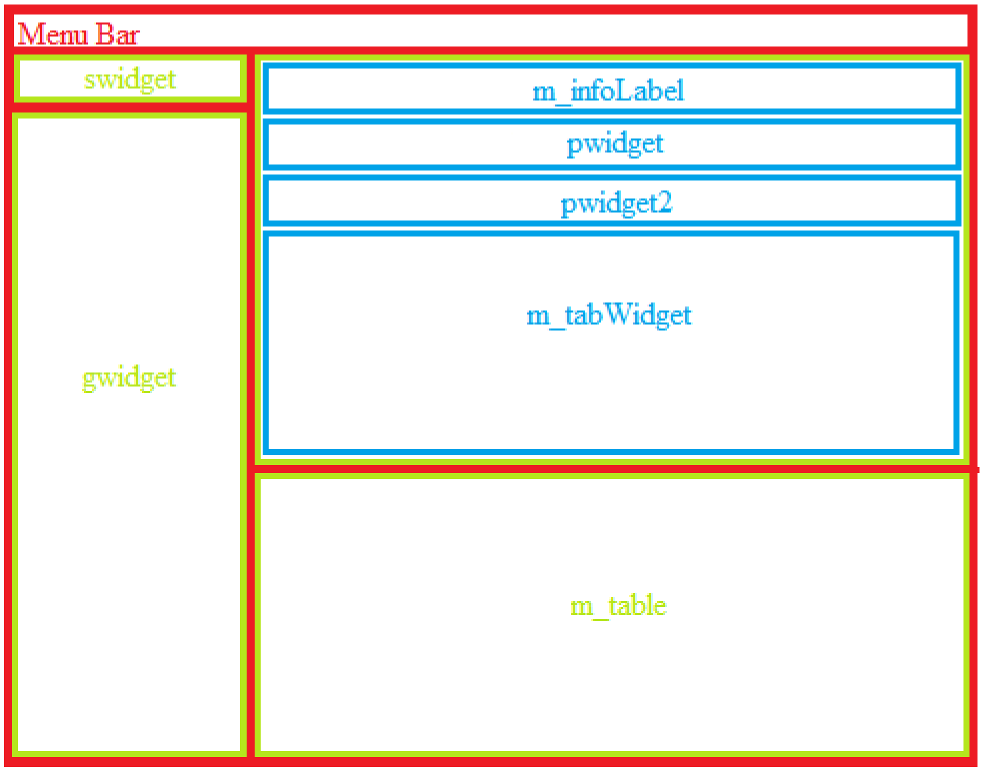


Genre: ALL, Alternative, Country, Electronic, Electronica, Pop

Artist: America, As It Is, Drenge, Dylan, Evil Needle, Folk Family Revival

Album: Bob Dylan's Greatest Hits, Breaking Down the Barriers, History: America's Greatest H, Mellow Drama - EP, Never Happy, Ever After

	Name	Track	Time	Artist	Album	Genre
1	Coloring	1	3:40	Kevin Garrett	Mellow Dram...	Pop
2	I Drew a Line	3	3:10	Folk Family Re...	Water Walker	Singer/Songw...
3	Lullaby	3	3:33	Evil Needle	Reminisce	Electronica
4	We Can Do W...	4	2:57	Drenge	Undertow	Alternative
5	Dial Tones	5	3:20	As It Is	Never Happy, ...	Alternative
6	The Math	9	5:51	Pale Blue	The Past We L...	Electronic
7	On the Rock ...	10	3:32	Joanne Cash	Breaking Dow...	Country

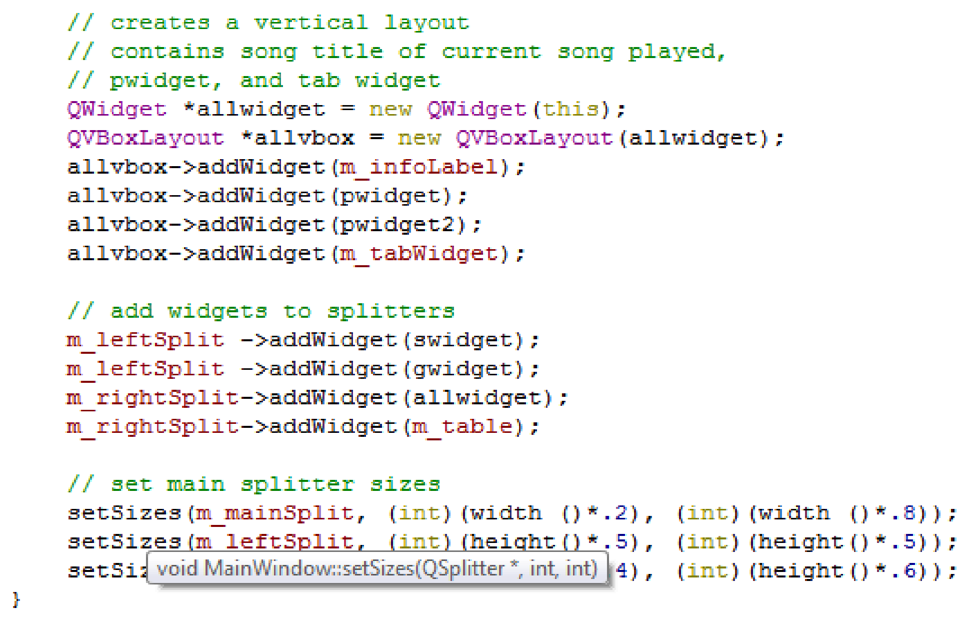


The diagram above shows the basic layout of the application. In the very top of the application, there exists a menu bar that allows the user to load the music into the application through opening the direction the music files are in.

The right section of the main split has two sections. The upper section contains the title of the currently played song in `m_infoLabel`, buttons and sliders for music controls in `pwidget` and `pwidget2`, and a tab widget called `m_tabWidget`. There are two tabs in the tab widget, which holds the graphics. The first tab contains the coverflow and a button is placed to the left and right of it, letting the user animate the cover flow. The user can also animate the cover flow through the left and right arrow keys for convenience sake. The lower section of the right splitter contains a playlist of all the songs available for playing.

On the left side of the main split, the upper section contains a search option in `swidget` that gives the user a choice to find a specific song, album, or artist. The lower section (`gwidget)` contains genre, album, and artist filters to narrow down the number of songs in the playlist. The implementation for the application's layout is written in in the function called `createLayouts()` as shown below:

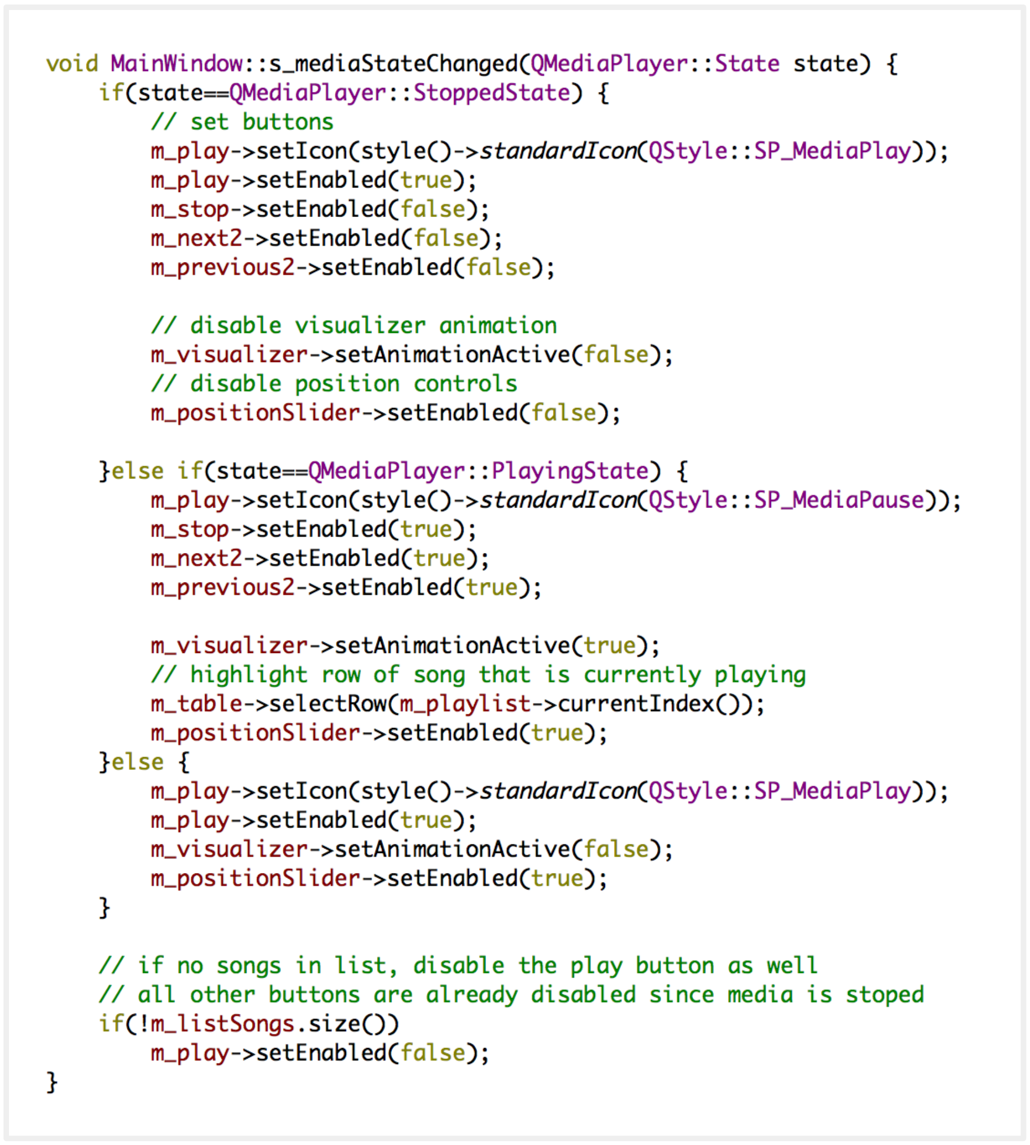
```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// MainWindow::createLayouts:
//
// Create layouts for widgets.
//
void MainWindow::createLayouts() {
    // creates a horizontal layout for coverflow tab
    QWidget *cwidget = new QWidget(this);
    QHBoxLayout *chbox = new QHBoxLayout(cwidget);
    chbox->addWidget(m_previous);
    chbox->addWidget(m_glWidget);
    chbox->addWidget(m_next);

    // initialize tab widget
    // adds tabs to widget of coverflow tab layout and visualizer
    m_tabWidget = new QTabWidget();
    m_tabWidget->addTab(cwidget, "Cover Flow");
    m_tabWidget->addTab(m_visualizer, "Visualizer");
    m_tabWidget->setMinimumSize(QApplication::desktop()->
                                width()/3,QApplication::desktop()->height()/3.5f);

    // creates a horizontal layout for search label and widget
    QWidget *swidget = new QWidget(this);
    QHBoxLayout *shbox = new QHBoxLayout(swidget);
    shbox->addWidget(m_typeSearch);
    shbox->addWidget(m_search);
    swidget->setFixedHeight(40);

    // create a one column grid layout,
    // first row contains the album picture of the currently played song
    // next rows consist of labels and list widgets, respectively.
    QWidget     *gwidget = new QWidget(this);
    QGridLayout *grid   = new QGridLayout(gwidget);
    grid->addWidget(m_albumLabel, 0, 0);
    grid->addWidget(m_label[0], 1, 0);
    grid->addWidget(m_label[1], 3, 0);
    grid->addWidget(m_label[2], 5, 0);
    grid->addWidget(m_panel[0], 2, 0);
    grid->addWidget(m_panel[1], 4, 0);
    grid->addWidget(m_panel[2], 6, 0);

    // create two horizontal layout for labels and widgets associated with playing songs
    QWidget *pwidget = new QWidget(this);
    QHBoxLayout *phbox = new QHBoxLayout(pwidget);
    phbox->addWidget(m_positionSlider);
    phbox->addWidget(m_positionLabel);
    phbox->addWidget(m_shuffle);
    phbox->addWidget(m_repeat);

    QWidget *pwidget2 = new QWidget(this);
    QHBoxLayout *phbox2 = new QHBoxLayout(pwidget2);
    phbox2->addWidget(m_previous2);
    phbox2->addWidget(m_stop);
    phbox2->addWidget(m_play);
    phbox2->addWidget(m_next2);
    phbox2->addStretch();
    phbox2->addWidget(m_muteButton);
    phbox2->addWidget(m_volumeSlider);
    phbox2->addStretch();
    phbox2->addWidget(m_toggleColor);
```



```
    // creates a vertical layout
    // contains song title of current song played,
    // pwidget, and tab widget
    QWidget *allwidget = new QWidget(this);
    QVBoxLayout *allvbox = new QVBoxLayout(allwidget);
    allvbox->addWidget(m_infoLabel);
    allvbox->addWidget(pwidget);
    allvbox->addWidget(pwidget2);
    allvbox->addWidget(m_tabWidget);

    // add widgets to splitters
    m_leftSplit ->addWidget(swidget);
    m_leftSplit ->addWidget(gwidget);
    m_rightSplit->addWidget(allwidget);
    m_rightSplit->addWidget(m_table);

    // set main splitter sizes
    setSizes(m_mainSplit,  (int)(width ()*.2),  (int)(width ()*.8));
    setSizes(m_leftSplit,  (int)(height()*.5),  (int)(height()*.5));
    setSi void MainWindow::setSizes(QSplitter *, int, int) 4),  (int)(height()*.6));
}
```

Music Controls

A set of buttons were provided for music control. This includes a play/pause, stop, previous, and next button. The workings of these buttons are pretty simple. However, the functionality of each button would differ depending on the current state of the media player. For example, if there are no songs in the list all buttons should be disabled. Also, if media is currently playing, the play button should become a pause button. These are just a few of many minor scenarios where the buttons should behave differently. Having `if()` statements at every place in the code where a button may behave differently would be unclean and error-prone. It was then decided that a single function should be made to manage all this. Additionally, `QMediaPlayer` emits a signal whenever the music state changes to another (`void stateChanged(QMediaPayer::State)`). This signal was connected to a slot in `MainWindow` called `s_mediaStateChanged(QMediaPlayer::state).` Based on the new state of the media, the slot would apply any modifications necessary to the buttons. For example, if the music was stopped, all buttons except the play button would be disabled. And if the media is playing, all buttons are enabled and the play button's icon changed to that of a pause button. The implementation is shown below:



```cpp
void MainWindow::s_mediaStateChanged(QMediaPlayer::State state) {
    if(state==QMediaPlayer::StoppedState) {
        // set buttons
        m_play->setIcon(style()->standardIcon(QStyle::SP_MediaPlay));
        m_play->setEnabled(true);
        m_stop->setEnabled(false);
        m_next2->setEnabled(false);
        m_previous2->setEnabled(false);

        // disable visualizer animation
        m_visualizer->setAnimationActive(false);
        // disable position controls
        m_positionSlider->setEnabled(false);

    }else if(state==QMediaPlayer::PlayingState) {
        m_play->setIcon(style()->standardIcon(QStyle::SP_MediaPause));
        m_stop->setEnabled(true);
        m_next2->setEnabled(true);
        m_previous2->setEnabled(true);

        m_visualizer->setAnimationActive(true);
        // highlight row of song that is currently playing
        m_table->selectRow(m_playlist->currentIndex());
        m_positionSlider->setEnabled(true);
    }else {
        m_play->setIcon(style()->standardIcon(QStyle::SP_MediaPlay));
        m_play->setEnabled(true);
        m_visualizer->setAnimationActive(false);
        m_positionSlider->setEnabled(true);
    }

    // if no songs in list, disable the play button as well
    // all other buttons are already disabled since media is stoped
    if(!m_listSongs.size())
        m_play->setEnabled(false);
}
```

There are other things that are being managed by this slot function which will be discussed later on.

It should also be noted that there are two slots for playing music (`s_play()` and `s_play2()`). This is because there are two different ways to play music and both cause separate behavior. The first way is by double clicking a song in the table. This, regardless of the current music state, plays that song. The second is using the play button. Music can be played by highlighting a song in the table and pressing play. Using the play button causes different behavior in the case that play is pressed when a song is paused. If a song is paused, regardless of which song is highlighted in the table, the play button will resume the play of the song. This required two different slots to handle the different ways of playing music.

Along with buttons for basic controls, a mute button and a volume slider are provided to the user. The mute button simply takes advantage of the `setMuted(bool)` from the `QMediaPlayer` class. The mute button's `clicked()` signal is connected to a slot in `MainWindow` that simply toggles the mute status of the media (if media is currently muted, it is unmuted and if it is currently unmuted, it is muted). Additionally, if the media is muted, the icon of the mute button is changed to a mute icon. The volume slider uses the `setVolume(int)` accessor from the `QMediaPlayer` class. The media's volume is connected to the slider. There are also a position slider and a label that corresponds to each other and tells the point of the song that is being played. The slider can be moved by the user to go to a different point of the song. The implementation will not be provided as this was a very trivial task.

Shuffle and repeat buttons were also provided to the user. These buttons were set as checkable buttons since they were to be used as toggles. The functionality of these buttons was quite straight forward thanks to the use of `QMediaPlaylist`. An instance of `QMediaPlaylist` was attached to the `QMediaPlayer`. Rather than setting individual media to `QMediaPlayer` each and every time, all songs were loaded to an instance of `QMediaPlaylist` at once. The playlist was then attached to the `QMediaPlayer` using `setPlaylist(QMediaPlaylist)`. The playlist has different playback modes. The default, for example, was `QMediaPlaylist::Loop`. This plays all songs in the list in order and when the end of the playlist is reached, the music is looped to the front. When the shuffle button is pressed, the playback mode is changed to `QMediaPlaylist::Random`. When the repeat button is pressed, the playback mode is changed to `QMediaPlaylist::CurrentItemInLoop`.

Image

Whenever a new song is being play by the music player, a function, `updateSong()`, will be called to update the labels that holds the title and the album cover to the new song's information.

```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// MainWindow::updateSong:
//
// Displays the album cover and song title of the currently played song.
//
void MainWindow::updateSong() {
    // sets label to the current song's title
    m_infoLabel->setText(m_listSongs[m_playlist->currentIndex()][TITLE] );

    // gets file data
    QString item_title = QString("%1").arg(m_listSongs[m_playlist->currentIndex()][PATH]);
    QByteArray ba_temp = item_title.toLocal8Bit();
    const char* filepath = ba_temp.data();

    // gets the tag that contains the album image
    TagLib::MPEG::File audioFile(filepath);
    TagLib::ID3v2::Tag *tag = audioFile.ID3v2Tag(true);
    QImage coverArt = initImage(tag);

    // scales and positions the image on the label
    m_cover = coverArt.scaled(100, 100, Qt::KeepAspectRatio, Qt::SmoothTransformation);
    m_albumLabel->setAlignment(Qt::AlignHCenter | Qt::AlignVCenter);
    m_albumLabel->setPixmap(QPixmap::fromImage(m_cover));
}
```

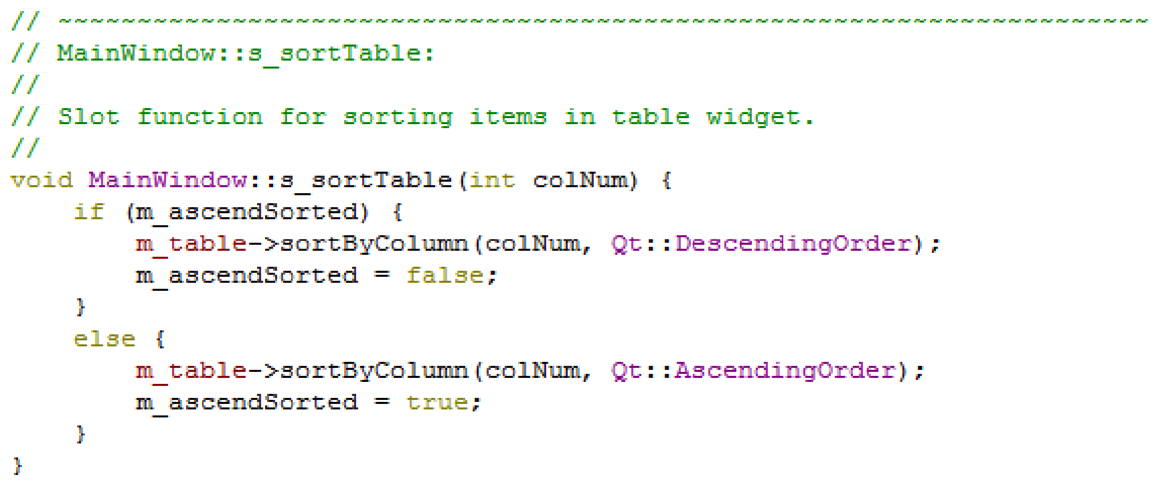
Changing the album cover is more complex than changing the song title as only the text of the label changes. The path of the song's mp3 file is converted into a 8-bit string and then the audioFile data is extracted from it. Using TagLib, a specific tag called ID3v2Tag is located from the audioFile. The data from the tag is used in `initImage()` to load an image for the album cover and is displayed on the label.

```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// MainWindow::initImage:
//
// Returns a qimage loaded from the tag's data
// or a default image if the tag does not contain the data.
//
QImage MainWindow::initImage(TagLib::ID3v2::Tag *tag) {
    // looks for picture frames only
    TagLib::ID3v2::FrameList tag_list = tag->frameList("APIC");
    QImage tag_image;

    // if picture frames do not exists, a default image is used for the album cover image
    // else the first frame's data is coverted to a qimage
    if(tag_list.isEmpty()) {
        QString path = QDir::currentPath();
        path.append("musicnote.png");
        tag_image.load(path);
    }
    else {
        TagLib::ID3v2::AttachedPictureFrame *tag_frame =
                static_cast<TagLib::ID3v2::AttachedPictureFrame *>(tag_list.front());
        tag_image.loadFromData((const uchar *) tag_frame->picture().data(),
                               tag_frame->picture().size());
    }
    return tag_image;
}
```

After locating the ID3v2Tag tag, `initImage()` is called to give an image of the album cover loaded using the tag's data. The data that holds the images in the mp3 file are labeled with APIC and are held in frameList objects. If the tag's data doesn't contain a frameList, then a default image is used for the album cover. If a frameList does exist, then the album cover image is loaded from the data.

Table

The table widget in the music application is a playlist that holds all the songs available for playing. The song's information displayed on the table are the title name, track number, duration of the song, artist, album, and genre. When the user clicks on one of these headers, the items will be sorted in alphabetized or numbered order according to the corresponding column. `m_ascendSorted` is a variable used to keep track of the order of the item are being sorted in. If the user clicks on a header that is already sorted, the the items will be resorted in descending order if the item were previously sorted in ascending order. The opposite will occur if the items were previously sorted in ascending order.



```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// MainWindow::s_sortTable:
//
// Slot function for sorting items in table widget.
//
void MainWindow::s_sortTable(int colNum) {
    if (m_ascendSorted) {
        m_table->sortByColumn(colNum, Qt::DescendingOrder);
        m_ascendSorted = false;
    }
    else {
        m_table->sortByColumn(colNum, Qt::AscendingOrder);
        m_ascendSorted = true;
    }
}
```

Search

The music application contains a searching option. The user can enter some text and search through song titles, albums, or artists already loaded to the application. The items with the matching text will be displayed in the table widget. The user can enter which field to look for by choosing one of the choices in the dropdown menu of the search button. If the search option is chosen from the drop down menu, then all the songs originally loaded will be displayed while the other choices will search through the specific field chosen. The implementation is shown below.

```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// MainWindow::s_search:
//
// Slot function for searching through table widget.
//
void MainWindow::s_search(int in) {
    // matches in (integer selected from m_search) with corresponding field value
    int index = 0;
    bool search = true;
    if(in == 1)
        index = 0;
    else if (in == 2)
        index = 3;
    else if (in == 3)
        index = 4;
    else
        search = false;

    m_searchText = m_typeSearch->text().toLower();
    m_table->setRowCount(0);

    // copy data to table widget
    for (int i=0, row=0; i<m_listSongs.size(); i++) {
        // skip rows whose field doesn't match text
        if (search&&!((m_listSongs[i][index]).toLower()).contains(m_searchText))) continue;

        m_table->insertRow(row);
        QTableWidgetItem *item[COLS];
        for (int j=0; j<COLS; j++) {
            item[j] = new QTableWidgetItem;
            item[j]->setText(m_listSongs[i][j]);
            item[j]->setTextAlignment(Qt::AlignCenter);
            // put item[j] into m_table in proper row and column j
            m_table->setItem(row, j, item[j]);
        }
        // increment table row index (row <= i)
        row++;
    }
}
```

Saving and Loading

One of the big things that music players have is its ability to remember songs that were previously loaded. The only issue with loading music in some applications is that it would take a lot of effort remove these songs from the music application. In order to solve this problem, a prompt window was created asking the user whether they would want to load the previously loaded directory. This made it so that the user had more control of what songs populate the player while also have easier way to load folders.

The functions that made the loading directories possible were saveDir, loadDirs, and s_loadPrev. The saveDir function is called after a folder in loaded and takes the path loaded as a parameter. The function creates a QSetting and maps a value,”dir”, to the path of the loaded directory using the setValue function. This path is saved as a QString elsewhere in the computer so that QTunes can go into the file and pull the data it needs. The path is pulled from this in the loadDirs function which is called in the MainWindow constructor. The function creates a QSettings and pulls the path that is stored as a QString. If this QString is empty the function ends, if not it means there was a path stored, and the function opens the prompt window. This prompt window is connected to the s_loadPrev function which will load the directory and populate the lists.

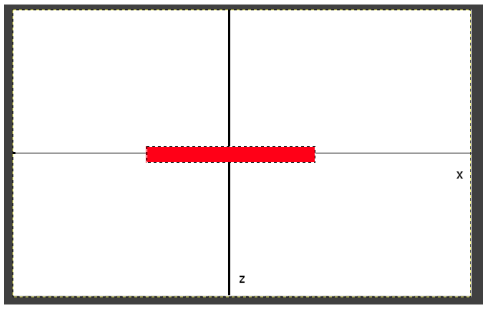
```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// MainWindow::saveDir:
//
// Save loaded directory.
//
void MainWindow::saveDir(QString path) {
    QSettings setting(QSettings::NativeFormat, QSettings::UserScope, "CS221", "qTune");
    setting.setValue("dirs",path);
}



// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// MainWindow::loadDirs:
//
// Pulls previous directory and opens the prompt window.
//
void MainWindow::loadDirs() {
    QSettings setting(QSettings::NativeFormat, QSettings::UserScope, "CS221", "qTune");
    QString p = setting.value("dirs","").toString();
    QString dir = QDir::toNativeSeparators(p);
    if(dir.isEmpty()) return;
    m_directory = dir;
    openPrompt *prompt = new openPrompt;
    connect(prompt, SIGNAL(load()), this, SLOT(s_loadPrev()));
    prompt->exec();
}



// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// MainWindow::s_loadPrev:
//
// Slot function to load previous directories.
//
void MainWindow::s_loadPrev() {
    traverseDirs(m_directory);
    initLists();
    initAlbums();
    m_glWidget->loadImages(m_albumsList);
}
```

Coverflow

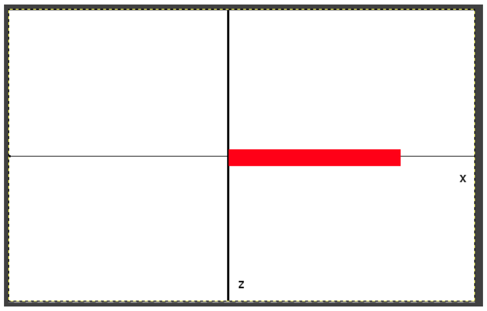
Painting the Pictures

One of the features of our music player is capability to draw the album art form the music that has been downloaded. To do this a new class called glWidget was created. In this class there is a function called `paintGL()` that is called to draw the image that is seen in the widget. The approach that was taken was to draw the images in the opposite direction that they were going to move. This meant that if we were to animate from left to right, the albums would be drawn from right to left. Before the images were drawn, they were categorized into four groups, the image in the middle, the image animating to the middle, images before the animating ones, and the images after the animating ones. The number of albums that were drawn is always one more than is seen due to the fact that one image needs to be animated into view. All of the images that didn't need to change rotation were rotated to either 90 or -90 degrees. This meant that when drawing the albums that were rotating there were extra translations that were needed to be done in order to have it rotate about an edge. The diagram below describes the process for drawing the images that were rotating in and out of the middle position.

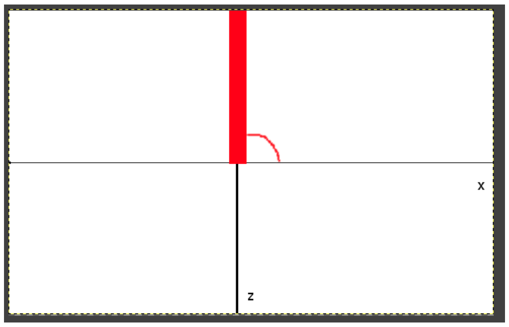
1) initial position:



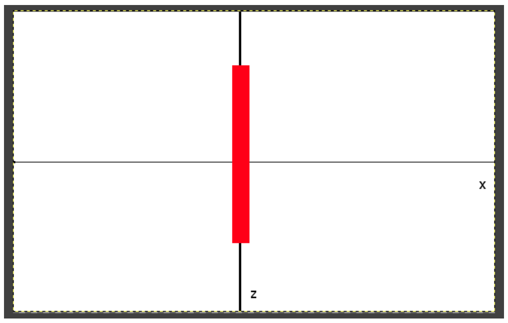
2) translate:



3) rotate:

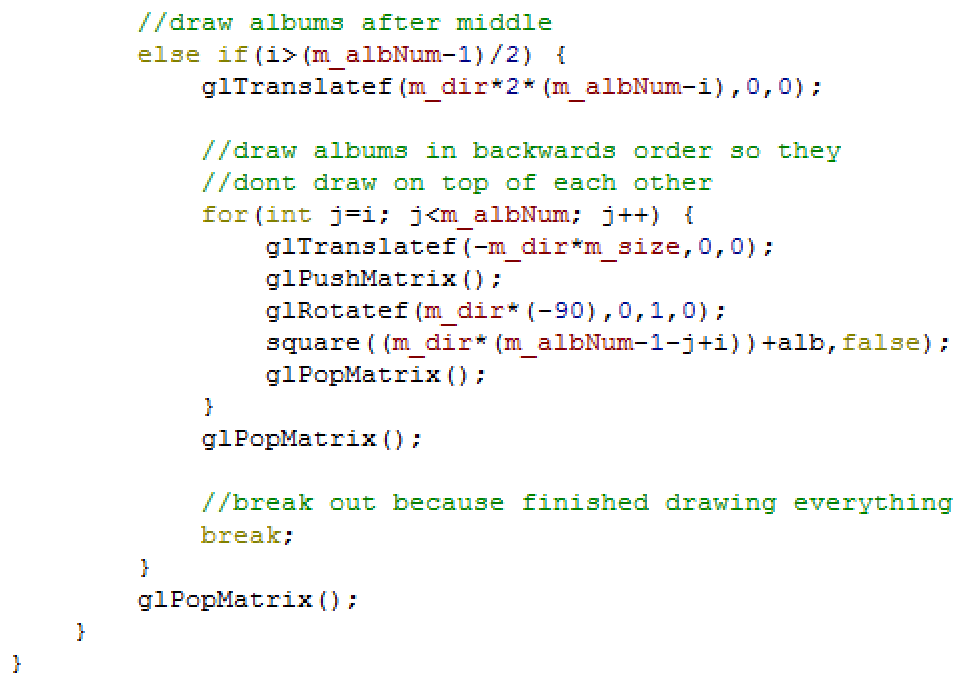


4) translate:



In order to draw each image in the right position a few things needed to be done. First, everything was translated to where the first image would be drawn and as the images were drawn it would translate to the opposite direction. The direction that the animation was occurring is was held by m_dir. This made sure that the albums were going to rotate to the right angle. The m_change kept track of how far into the animation it was, and lastly m_size showed the size of the album so that it translates exactly one image length.

```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// glWidget::paintGL:
//
//draws frames
//
void glWidget::paintGL() {
    glClear(GL_COLOR_BUFFER_BIT);
    glLoadIdentity();
    int alb;

    //which album to display first depending on direction
    if(m_dir==-1)
        alb = m_current+m_albNum;
    else
        alb = m_current;

    //translate to position where the first image will be displayed
    glTranslatef(m_dir*(-(m_albNum+m_change))*m_size/2,0,0);

    //display the album covers
    for(int i=0; i<m_albNum; i++) {

        //translates to spot and stores location
        glTranslatef(m_dir*m_size,0,0);
        glPushMatrix();

        //draws album that will rotate to middle
        if(i==(m_albNum/2)) {
            glTranslatef(m_dir*(m_change/2)*m_size/2,0,-1);
            glRotatef(m_dir*90*((1-m_change/2)),0,1,0);
            glTranslatef(-m_dir*m_size/2,0,0);
            square(alb+(m_dir*i),true);
        }

        //draws album that is in the middle
        else if(i==(m_albNum/2)-1) {
            glTranslatef(-m_dir*(1-m_change/2)*m_size/2,0,-1);
            glRotatef(-m_dir*90*(m_change/2),0,1,0);
            glTranslatef(m_dir*m_size/2,0,0);
            square(alb+(m_dir*i),true);
        }

        //draws albums before middle
        else if(i<(m_albNum-1)/2) {
            glRotatef(m_dir*90,0,1,0);
            square(alb+(m_dir*i),false);
        }
```



```
        //draw albums after middle
        else if(i>(m_albNum-1)/2) {
            glTranslatef(m_dir*2*(m_albNum-i),0,0);

            //draw albums in backwards order so they
            //dont draw on top of each other
            for(int j=i; j<m_albNum; j++) {
                glTranslatef(-m_dir*m_size,0,0);
                glPushMatrix();
                glRotatef(m_dir*(-90),0,1,0);
                square((m_dir*(m_albNum-1-j+i))+alb,false);
                glPopMatrix();
            }
            glPopMatrix();

            //break out because finished drawing everything
            break;
        }
        glPopMatrix();
    }
}
```

Texture Mapping

The display of the album covers in the coverflow is implemented by using texture mapping based on the textures created from a list of the album cover art received from the `MainWindow` class with no repeated album cover images. A unique texture is created based on each image and all the textures are then held in a list called `m_texture` in the function `loadImages()`.

```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// glWidget::loadImages:
//
// Adds images to the list of qimages.
//
void glWidget::loadImages(QList<QImage> imgs) {
    m_loaded =true;
    glEnable(GL_TEXTURE_2D);

    for(int i=0; i<imgs.size(); i++) {

        // loads and binds the texture from a qimage
        m_texture << bindTexture(imgs[i]);
        glBindTexture  (GL_TEXTURE_2D,   m_texture[i]);

        // sets texture parameters
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    }
    glDisable(GL_TEXTURE_2D);
    m_listLength = imgs.size();
    m_current=0;
    updateGL();
}
```

The function `square()` is used bind one of the texture created onto a square. The function takes in two parameters: `index` is used to specify which texture in the list is binded to the square and `flip` determines if the texture needs to be flipped over so that image displayed will be correctly positioned upright. If `index` is not within the range of possible indices for texture list, then it is converted into a number within the range. The texture at the index in the list will draw the album cover image onto the square. A red outline is then drawn surrounding the square to give a clean look. However, if there are no textures in the texture list, then the square will be empty.

```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// glWidget::square:
//
// Draws one album cover.
//
void glWidget::square(int index, bool flip) {

    //handles when index is larger than length of Qimage list
    //and converts to index in the list
    index = index%m_listLength;
    if(index<0) {
        index*=-1;
        index = m_listLength-index;
    }
```

```
// binds a texture to the square
// else a blank square is created instead
if(m_loaded&&index<m_listLength) {

        // enables texture mapping
        glEnable(GL_TEXTURE_2D);

        // selects texture to bind
        glBindTexture(GL_TEXTURE_2D, m_texture[index]);


    // draws filled album cover polygon flipped
    if(flip) {
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
        glColor3f(.8, .8, .7);
        glBegin(GL_QUADS);
            glTexCoord2f(0, 1); glVertex3f(1.0,1.0,0.0);
            glTexCoord2f(1, 1); glVertex3f(-1.0,1.0,0.0);
            glTexCoord2f(1, 0); glVertex3f(-1.0,-1.0,0.0);
            glTexCoord2f(0, 0); glVertex3f(1.0,-1.0,0.0);
        glEnd();
        glDisable(GL_TEXTURE_2D);
    }

    //draws filled album cover not flipped
    else {
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
        glColor3f(.8, .8, .7);
            glBegin(GL_QUADS);
            glTexCoord2f(1, 1); glVertex3f(1.0,1.0,0.0);
            glTexCoord2f(0, 1); glVertex3f(-1.0,1.0,0.0);
            glTexCoord2f(0, 0); glVertex3f(-1.0,-1.0,0.0);
            glTexCoord2f(1, 0); glVertex3f(1.0,-1.0,0.0);
        glEnd();
        glDisable(GL_TEXTURE_2D);
    }

    // draws a red outline on the polygon
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    glEnable(GL_LINE_SMOOTH);
    glColor3f(.8, 0, 0);
    glBegin(GL_QUADS);
         glVertex3f(1.0,1.0,0.0);
         glVertex3f(-1.0,1.0,0.0);
         glVertex3f(-1.0,-1.0,0.0);
         glVertex3f(1.0,-1.0,0.0);
    glEnd();
    glDisable(GL_LINE_SMOOTH);
    glFlush();
}
```

```
    //creates empty white squares for initial start up
    else {
        glBegin(GL_POLYGON);
            glVertex3f(1.0,1.0,0.0);
            glVertex3f(-1.0,1.0,0.0);
            glVertex3f(-1.0,-1.0,0.0);
            glVertex3f(1.0,-1.0,0.0);
        glEnd();
        glFlush();
    }
}
```

Visualizer

A nice additional feature to most music players is a music visualizer. A visualizer attached to music players is a way for the music to be displayed using bars, lines, and many other effects. Usually, the music visualizer corresponds to the bass, mids, and highs of the song. We first attempted to do this with QAudioprobe, but it turns out that it does not work on mac platforms. Instead, random numbers were used to fill the visualizer.

Drawing The Visualizer

`glVisualizer` is a base class of `QGLWidget`. To draw the visualizer, very basic `OpenGL` was used. Additionally, a couple of variables were used to dictate how the visualizer was to be drawn:

```
// glvisualizer.cpp
// constants to draw bars

/* the distance of the first (and last)
 * bar(s) from the side of the canvas */
const float DISTANCE_FROM_SIDE = 0.05f;
/* Y-Coordinate indicating base of bars. */
const float BAR_BASE_POSITION = -0.7;
private:
    // glvisualizer.h

    /* Number of bars in visualizer. */
    static const short NUM_BARS = 100;
    /* Array of bar heights. */
    float m_barHeights[NUM_BARS];
```

The four variables shown above were used to draw each individual bar. Specifically, the width of each bar would be calculated as: `(canvasWidth - (2*DISTANCE_FROM_SIDE))/NUM_BARS`. Note that these numbers could have been hardcoded into the paint function but creating a reusable variable for this task increases readability and lowers the chance of bugs. Additionally, using variables instead of hardcoding number values increases maintainability in the case of future changes. For example, if the number of bars in the visualizer was required to be changed, rather than going through the code and redoing all the calculations, the coder would simply be able to change the value of `NUM_BARS` from 100 to the new number of bars. The code would then do all the calculations for the coder and the visualizer would work in the same exact manner without affecting anything else. This would decrease unanticipated side effects and, therefore, bugs.

Animation

To perform the animation, two different `QTimer`s were used. `m_barDropTimer` was used to perform the drop animation on the bars. `m_barJumpTimer` was used to periodically make the bars jump up. And of course, each timer was connected to a slot function:

```
// connect timer to redraw bars
connect(m_barDropTimer, SIGNAL(timeout()),
        this, SLOT(s_redrawDroppingBars()));
// jump bar timer jumps bars up randomly
connect(m_barJumpTimer, SIGNAL(timeout()),
        this, SLOT(s_resetBarHeights()));
```

`s_redrawDroppingBars()` would simply call `repaint()`. And each time this happened, each bar would be redrawn but with a height of `DROP_RATE` (`0.3f`) less than before. This would create the animation of the dropping bars. It is in `paintGL()` that the heights are lowered. Initially, the heights would be lowered in the slot function and then the paint function would be called. This was the more logical way since the paint function should not be responsible for or know about lowering the heights. It should only draw the bars based on the current heights. However, this would require two separate loops to perform tasks that overlap. So both tasks were placed in the same loop in `paintGL()` which resulted in a more efficient redraw method. The implementation is shown below:

```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// glVisualizer::painGL():
//
// Repaints the bars.  Decreases bar heights every time this is called.
//
void glVisualizer::paintGL() {
    glClear(GL_COLOR_BUFFER_BIT);

    // the left point of the first bar (x-coordinate)
    float leftPoint = DISTANCE_FROM_SIDES - 1.0f;

    // width of each bar is the canvas's length divided by the number of bars
    const float canvasWidth = 2.0f;
    float barWidth = (canvasWidth - (2*DISTANCE_FROM_SIDES))/NUM_BARS;

    // in paintGL()
    // draw [NUM_BARS] bars
    for(int i = 0; i < NUM_BARS; ++i) {
        // every time paintGL() is called, bar heights decrease by DROP_RATE
        m_barHeights[i] -= DROP_RATE;
        // shouldn't go below min height
        m_barHeights[i] = m_barHeights[i] < MIN_HEIGHT ? MIN_HEIGHT : m_barHeights[i];

        glBegin(GL_QUADS);
            glColor3d(m_colorArr[0][0], m_colorArr[0][1], m_colorArr[0][2]);
            glVertex2f(leftPoint, BASE_POSITION);
            glColor3d(m_colorArr[1][0], m_colorArr[1][1], m_colorArr[1][2]);
            glVertex2f(leftPoint, BASE_POSITION + m_barHeights[i]);
            glColor3d(m_colorArr[2][0], m_colorArr[2][1], m_colorArr[2][2]);
            glVertex2f(leftPoint + barWidth, BASE_POSITION + m_barHeights[i]);
            glColor3d(m_colorArr[1][0], m_colorArr[1][1], m_colorArr[1][2]);
            glVertex2f(leftPoint + barWidth, BASE_POSITION);
        glEnd();

        // set new left point for the next bar
        leftPoint += barWidth;
    }
}
```

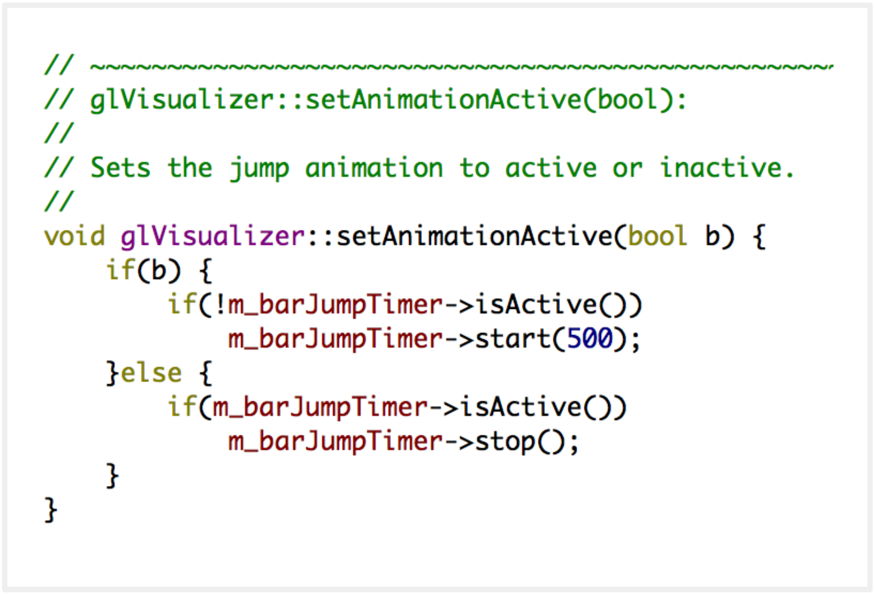
Note that `glColor3d()` is being called and is using `m_colorArr[][]`. This will be discussed in a moment.

The second timer (`m_barJumpTimer`) is used to make the bars jump. To simulate a proper visualizer, a few steps were taken for this. First, a random number generator was used to create a `float` value between the `barMaxHeight` (`1.5f`) and `barMinHeight`(`0.5f`) each time while looping through each bar. If this new height was larger than the current height of the bar stored in `m_barHeights[]`, the height would be updated. Now next time `paintGL()` would be called (which is very soon), the bar would be redrawn higher than it was before. This creates the jump animation. The random number would be discarded if it was lower than the height of the current bar. The implementation of the slot function is shown below:

```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// glVisualizer::s_resetBarHeights():
//
// Creates a new random height for each bar.
// Only applies new height if it is greater than current height.
//
void glVisualizer::s_resetBarHeights() {
    // (maximum height for each bar) * 100 to avoid redundant casting
    int barMaxHeight = 150;
    int barMinHeight = 50;

    for(int i = 0; i < NUM_BARS; ++i) {
            // new height is higher than the current height and less than the maximum height
            float newHeight = (rand() % (barMaxHeight - barMinHeight) + barMinHeight) / 100.0f;
            // if random height is larger than current bar height, set new height
            m_barHeights[i] = newHeight >= m_barHeights[i] ? newHeight : m_barHeights[i];
    }
}
```

The two timers shown largely perform the animations. However, animation management is also required. When implementing the visualizer, it was attempted to keep the class as self contained as possible. This follows the Object Oriented Design principle of encapsulation. This way, the class is much more reusable and much less error-prone. However, the visualizer would have no way of knowing when music is playing or not. So this management was done by the parent class.

The parent class (`MainWindow`) manages the animation activity of the visualizer. A public setter function was offered by the visualizer class for this. `setAnimationActive(bool)` sets the activity of the jump animation:

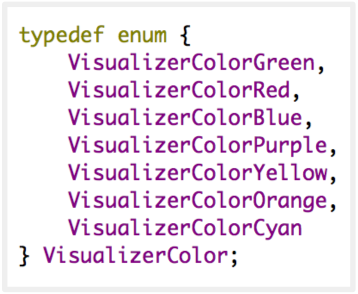


```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// glVisualizer::setAnimationActive(bool):
//
// Sets the jump animation to active or inactive.
//
void glVisualizer::setAnimationActive(bool b) {
    if(b) {
        if(!m_barJumpTimer->isActive())
            m_barJumpTimer->start(500);
    }else {
        if(m_barJumpTimer->isActive())
            m_barJumpTimer->stop();
    }
}
```

The function is used in the `s_mediaStateChanged(QMediaPlayer::State)` slot function residing in `MainWindow` that was mentioned earlier. When music is playing, the jump timer is active and periodically jumps the bars up. Otherwise, the timer is stopped. Note that the bar drop timer is always active regardless of whether music is playing or not. If it were not always active, the bars would just pause at whatever heights they were at when music is stopped or paused. Instead, the bars should keep falling which is accomplished by the drop timer.

Color

The visualizer color is retrieved using the `m_colorArr[3][3]` member variable. It is a two-dimensional array two hold three different floats to make a color, and three different colors to make for a gradient effect. The color of the bar is changed using `setColor(glVisualizer::VisualizerColor)` where `VisualizerColor` is defined as an enumerated type:



```
typedef enum {
    VisualizerColorGreen,
    VisualizerColorRed,
    VisualizerColorBlue,
    VisualizerColorPurple,
    VisualizerColorYellow,
    VisualizerColorOrange,
    VisualizerColorCyan
} VisualizerColor;
```

The function sets the float values for the two-dimensional array. `paintGL()` is then called. Now, when the bars are drawn, a different set of color values will be retrieved from `m_colorArr[3][3]` every time.

The change in colors is managed by the parent class. `glVisualizer` also provides a slot function called `s_toggleVisualizerColor()`:

```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// glVisualizer::s_toggleVisualizerColor():
//
// Slot to be connected to the click signal of a button in an external class.
//
void glVisualizer::s_toggleVisualizerColor() {
    /* increment color and loop around if color limit is exceeded */
    short newColor = (m_color + 1) % NUM_COLORS;
    m_color = (VisualizerColor)newColor;
    // will set m_colorArr values to corresponding glVisualizer::VisualizerColor
    setColor(m_color);
}
```

Taking no arguments, this slot function changes the visualizer to the next color in the sequence. It takes the current enumerated color value, adds one and sets that as the new color. The toggle button in `MainWindow` is connected to this slot. Every time the toggle button is pressed, the next color in the sequence is displayed.

Conclusion

Obstacles

Upon starting the project we came across many obstacles that became very notable. One of the biggest obstacles that we faced was getting TagLibs to work. Trying to get it installed correctly on both macs and windows platforms was a challenge. With the help of the teaching assistant, Siavash Zokai, we got TagLib to work on mac platforms after installing MacPorts and using it to download the TagLib library. MacPorts is a program for mac that helps install libraries on mac platforms. For the Windows platform, Siavash went into the TagLib library code and reconfigured the system at which the library will work for.

Sharing files between the three group members was an issue that needed to be solved on the earlier stages of the project. Initially, the problem became evident when all group members had different versions of the same code. The way files were sent was with email and after discovering the problem, each group member was assigned a issue to solve with a deadline to upload their solution. Once more files were included in the project, the each group member focused on different files and collaborated when there was communication between the files.

What We Learned:

Upon entering the class the different libraries we had to used seem foreign and complex. Learning about these libraries was one of the many rewarding parts about the class. In class we focused on the Qt and OpenGL libraries. Learning about the way in which various Qt widgets were created along with how to draw the items in OpenGL was some of the more crucial thing we learned and implemented in the music player. These libraries are used in various successful and industries and having the knowledge to use them can be very powerful in the future. This project gave our first experience in working on a complex project that required multiple people working on it at the same. In order to work efficiently, the project were divided into parts and worked on separately by each group member. Whenever one group member encountered a problem and wasn't able to resolve it, another member would help often help out.

Future Improvements:

One of the more incomplete parts of our music player is the music visualizer. This is due to the fact that it is not actually connected to any data from the audio that is playing. To get this working, another library is required due to the fact that the one Qt provides only works for Windows platforms. In order for the music player to work with all platforms we would need a library that would pull data from the music on all platforms.

Another incomplete part of the music player is based on the differing platforms. The album covers wouldn't appear in the cover flow when the user chooses to load the previous directories. However, this issue only appeared when the program was running under the Windows platform. In fact, there was no issue with the cover flow, if the program was compiled under the Mac platform. Therefore, we need to pinpoint the solution that can solve this issue that deals with differing platforms.

We also wanted to add extra features into our music player. For example, the user having the ability to alter the audio file metadata would be useful for those who have songs that have inaccurate names. Another feature that would be added would be the ability to play around with the different levels that are involved with music like treble and bass. This way the user has more power over what they are hearing and can change the settings to the way they enjoy listening to their music.