QTunes: A Music Player and Library Organizer

Programmed with in C++ with QT and OpenGL

By Svyatoslav Turets and Whitney Drazan
Overview

Our project was to create a music player, with a basic feature set, mimicking the most commonly used features in iTunes. In our program the user selects a directory and all MP3 files in that directory and its subdirectories are imported into the program. The user then has the ability to play this music and filter the imported songs in various ways. The user can also browse the album artwork of their songs or see a visual representation of a playing song. We called our program qTunes since it was created using QT.

QT is an application framework used for developing applications with GUIs. QT has built in classes for many interface elements and implements event handling through signal slot connections. QT will throw events for the most common ways people interact with the different types of GUI elements. This simplifies the process of creating a graphical user interface since we can use the built-in functionality of QT. We can create an interface element with a single line of code and easily specify actions when, for example, a user clicks a button or adjusts the value of a slider.

An additional advantage to using QT is that where we do not specify a property, QT makes a reasonable guess. For example, in a table widget, pressing the down key will change the active row to the one below it, even though we have not defined the behavior. Also, we do not need to manually set the size for each and every widget in our interface. QT automatically sets the size when it is not specified so that the result is reasonable given its place in the interface. Additionally, because QT is cross-platform, much of the code from programs created with it can be ported to other operating systems with minimal changes.

The visualizer and cover flow portions were created using OpenGL. OpenGL stands for Open Graphics Library. It is a multi-platform API for creating computer graphics. While OpenGL is very powerful, it only has support for basic drawing functions. For example, while you can create lines and polygons using OpenGL, there is not built-in way to create many common shapes, such as circles and ellipses. However, like programs created with QT, programs created with OpenGL can often be brought to other platforms without many changes to the code.
Media Player and Metadata

Media Player
Our group reviewed three different libraries for music playback: Audiere, GStreamer and QMediaPlayer. We chose to use QMediaPlayer.

In the beginning stages, our program implemented both media playback and metadata reading with Audiere. Audiere as compared to GStreamer, it has a user-friendly API. However, it came with some major drawbacks. Many of these flaws stem from the age of the library. Development ceased on Audiere in 2006. When Audiere was developed, most compressed, downloaded audio was distributed in MP3 format at bitrates of 128kbps or lower. Media formats have changed greatly in the intervening time. So while the library supports multiple formats, Audiere does not support many of the current popular formats, such as M4A. Additionally, the library only supports MP3s with bitrate 192kbps or lower.

The second option we investigated was GStreamer. GStreamer is quite flexible in file format support. It can play both audio and video. The biggest drawback to GStreamer is that it is cumbersome to use. It uses a pipeline-based approach. A file element connects to a decoder element, which connects to an output hardware element. Although this may offer great control over the media playback, to actually implement the code was quite complicated. Since our project only required us to play media at single speed, with any appropriate codec, the flexibility of the GStreamer API was not utilized and did not justify the increased complexity.

The third option, which we ultimately chose, was QMediaPlayer, a built-in part of QT 5.0. We chose to use QMediaPlayer for ease of use and flexibility. While QMediaPlayer is based on GStreamer, the API is streamlined so that we did not need to worry about choosing pipelines, codecs and hardware. Using this API, we only need to specify a file and the library will load and play the song with the appropriate codec on the default device. So, we get the file format flexibility, without the added complexity. Additionally, since QMediaPlayer is part of QT, it supports signal slot connections. This makes it easy to trigger events based on the status of the media. It also makes the code for our program more uniform, since our event handling for GUI elements was already implemented in QT. In order to determine what occurs when certain actions are taken by the program or user, we need only look at the events handled by QT. We do not need to examine how event handling is implemented by an external library. The disadvantage to QMediaPlayer is that it is currently not possible to use it to get media metadata.

Metadata
Just as with playback, we examined several options to get the metadata for our audio files. We looked at Audiere, GStreamer and TagLib. QMediaPlayer, which we used to play the media, does not have support for metadata.

We first used Audiere to get the metadata. However, the same as with playback, we found the Audiere library to be very limiting. Audiere only has support for tag data from MP3 files, it is read-only and it only supports ID3v1 tags. Since many modern audio files do not meet all of these criteria, it would not have been the best choice.
Similar to its playback functions, we also found GStreamer’s metadata functions to be too complicated to implement for the scope of our project. The pipeline architecture also makes it difficult to read the code and determine what the player is doing. While GStreamer could read metadata for any type of media, which we would want to support with our program, the implementation was too difficult the added complexity.

We ultimately chose TagLib. TagLib allows read-write access to the metadata, supports metadata from multiple formats and the more modern ID3v2. The API is very user friendly and the code is quite easy to read. There are built-in methods for getting common tags. Whereas Audiere required us to read every tag and search the tag type to determine if it was what we wanted, in TagLib a search is not necessary. There are specific member functions for the most common tag types, such as artist, genre and length of song.
Main Window

In our interface, the playback controls and information are always displayed, as is the song table. The user should always know what song is playing and have the ability to control playback. Additionally, the user should always be able to select a different song to play.

Figure 1 qTunes with Lists

Figure 2, qTunes with Visualizer
For other items, it made sense to have them visible only some of the time. The genre, artist and album list filters do not make sense to display at the same time as cover flow. Cover flow applies its own filter and there would be a conflict about which filter to apply if both were displayed simultaneously. Since cover flow and the category filters are both filters, we chose to put them in a tab widget as users may want to easily flip between the two types of filtering.

Below are diagrams of the highest level interface elements.

We chose to contain the visualizer and filtering tab widget in a stacked widget. A stacked widget is similar to tab widget in that only one layout is displayed at a time. However, it differs by not displaying a control to switch between the layouts. We used a stacked widget for two reasons. First is space on the screen. There is not enough room on the screen to display everything at once. Simultaneously displaying the lists or cover flow and visualizer would mean that some or all of the elements were much smaller than desirable. So either the song table would only display several rows or the visualizer and coverflow would be too small to be useful.

The second reason we chose to implement this as a stacked widget is that users generally activate a visualizer when they have finished searching for songs. Since visualizers require the user’s visual attention, the user cannot simultaneous search for a song in the lists and view the visualizer. The user can toggle between the filters tab widget and the visualizer via an option in the menu bar.
Playback Information and Controls

Figure 3, Playback Information and Controls Layout

The Playback information and control layout displays information about the song, which is currently loaded into the player. It also enables the user to toggle pause and play, to adjust the volume of the player or skip to a specified position in the song.

The Pause/Play Button is implemented as a push button. We used sliders for the user to control the volume and for the position of the song. We used labels to display the information regarding the time elapsed, duration of the song and the artist and song title.

Category Lists

Figure 4, Category Lists

We used table widgets to display lists of genres, artists and albums. When a user clicks on an item in one of the lists, the song table and subordinate lists are filtered to reflect the selection. The implementation of this is discussed in more depth in a later section.
Song Table

<table>
<thead>
<tr>
<th>Song</th>
<th>Artist</th>
<th>Album</th>
<th>Genre</th>
<th>Track</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>November Rain</td>
<td>Guns N Roses</td>
<td>Greatest Hits</td>
<td>Pop</td>
<td>5</td>
<td>2:20</td>
</tr>
<tr>
<td>Baby ft. Ludacris</td>
<td>Justin Bieber</td>
<td>My World 2.0</td>
<td>Pop</td>
<td>6</td>
<td>3:45</td>
</tr>
<tr>
<td>I Will Wait</td>
<td>Mumford &amp; Sons</td>
<td>Babel</td>
<td>Rock</td>
<td>3</td>
<td>2:20</td>
</tr>
<tr>
<td>Wannabe</td>
<td>Spice Girls</td>
<td>Greatest Hits</td>
<td>Pop</td>
<td>3</td>
<td>2:20</td>
</tr>
<tr>
<td>Spice Up Your Life</td>
<td>Spice Girls</td>
<td>Greatest Hits</td>
<td>Pop</td>
<td>5</td>
<td>2:20</td>
</tr>
</tbody>
</table>

Figure 5, Song Table

The song table is a table widget. It holds the information about all of the songs. User may double click to play a song or right click to bring up a context menu. When filters are applied, no information is deleted. The visibility property of each item row is set so that the user only sees songs which match the filter.
Filters and Information Storage

Importing Songs
Using a file dialog, the user selects a directory. The program finds all MP3 files in the directory and all subdirectories of the selected directory and gets the metadata to add them into the library.

Song Table
Our table stores the following seven pieces of information about each file in the library: Title, Album, Artist, Genre, Track Number, Duration and Path. The Path entries are hidden from the user to save space on the screen. In general, the path to a song is not information a user considers when selecting a song to play.

The song table is populated during the import process. For each MP3 file found the program, the creates a temporary string list and uses TagLib to get the values of Title, Album, Artist, Genre, Track Number, Duration from the metadata and store them in the string list. The string list is then used to fill a row in our table.

The song table serves two purposes: to both store and display information about songs. It stores the list of all songs in the library, and it displays the songs matching the filter criteria. Equivalently to the latter, it hides entries not matching the filter criteria.

Category Lists
Our implementation of the category lists changed through the course of our project. The first approach was as follows. From the main song table we compiled the unique values of from each of the categories in simple lists without considering dependencies of the different criteria. That is, if we encountered a duplicate album name, we did not consider that the two entries could be unique. Filtering the song table consisted of hiding all songs not matching the filter criteria and repopulating the any dependent criteria lists based on the displayed songs.

A problem we encountered with this simple approach is that it did not allow for the possibility that multiple artists could have an album with the same name. So, when the user chose to filter by the album name, the table would display all songs with album name matching the entry selected in the list. Consider an album named “Greatest Hits”. It is quite conceivable that many artists would have an album with this title. If the user clicked on “Greatest Hits” in the album list, all songs from every album named “Greatest Hits” would be visible in the song table. There was no way for the user to select only a single album because the filtering did not differentiate between these albums. While it’s true the user could first select the artist and then the album to work around this issue, we did not feel that is was proper solution. The list should be in one-to-one correspondence with unique albums, not just with the album names.

So we decided to store information about the categories as shown in the diagram below. Note that only the first field will be visible to the user.
The Genre Table contains only one column, a list of genres. This is because genre is the top level filter criterion. The Artist Table also contains columns for both artist and the genre of the artist. This allows us to filter the artist based on a genre selection. The Album Table is the most complicated. It contains Display Name, Artist, Genre, Original Album Name and Path to Cover Art. Display Name is what will be shown to the user. Most of the time, this will just be the name of the album. Genre is employed when filtering the Album Table. Artist is included for filtering, but also to solve the duplicate album title problem as described below. Album is used to filter because the display name may be different. Path points to a file with cover art, if such a file exists and to any song in the album if none has cover art.

Since the Artist List contains genre information and the Album List contains artist information, we no longer have to generate the lists each time a filter of a higher level item is applied. We can simply filter the dependent tables based on the criteria for the higher level item.

The other advantage of the table based approach is that it allows us to solve the duplicate album name problem. We define an album based on both the name of the album and the artist who recorded it. When we encounter a duplicate album name, the display name of the album will be concatenated with the name of the artist. For example, “Greatest Hits(The Beatles)”.

Having a display name allows us to retain the original information about the album, while simultaneously allowing the user to distinguish between albums of the same title.

**Filtering**

A major function of our program was to create of a library of the user’s songs. In addition to listing the songs, we wanted the user to be able to filter their songs based on three criteria: Genre, Artist and Album.

As you can see in the section explaining the interface, our program displays unique values of each of the three categories in lists above the table displaying the songs. For the purposes of this project, we
considered categories to be hierarchical. Genre is the highest level element. Artists belong to a genre and albums belong to an artist.

When the user clicks on an entry in one of the lists, the song table displays only the items matching item clicked. Additionally, the dependent category lists are filtered. Figure 7 describes the filtering process when an item in a list is clicked.

![Figure 7, Process for filter based on selection in list](image)

**Filtering by Album**

We decided that an album is unique if it is a unique combination of album and artist. So when we filter by album, we check that entries in the table match both the artist and album of the row selected in the list. In the album list, we want the user to be able to differentiate between albums of the same title by different artists. So upon finding a duplicate we append the album name with the artist name. We also need the program to be able to filter based the user’s selection in the Album Table. If there were only one field for title, then appending the title field with the artist name in the Album Table would destroy our ability to filter. Searching for the edited album name would not yield any results. The title of the album in the Album Table would no longer match the title of the album in the Song Table.

To solve this we have a separate display name. We can display one name to the user, but employ the original album title to search. For example, if the user sees “Greatest Hits (The Beatles)” the program will filter based on songs matching “Greatest Hits” in the album field and “The Beatles” in the artist field.

**Generating the Lists**

The category lists are generated after the song table has been populated. Since each of the category tables has a different structure, manner in which they are generated is slightly different. However, the principle is the same. We store the appropriate fields from every entry from the library table in a list. Then we sort the list, remove the duplicates and add and “(All)” option, which when selected, removes the filter. In the case of the album table, if we find that two albums by different artists have the same name, we concatenate the artist name after the album name.

Below is the code to populate the album list.
void MainWindow::populateAlbumList()
{
    //Clear the table in case this is a subsequent population of the list
    m_Lists[2]->clear();

    //Insert items from the library into the TableWidget.
    //There are two entries for album in case there is a duplicate album and
    //the displayed album name needs to be concatenated with the artist to
    //distinguish them in the list.
    for(int x = 0; x < m_Table->rowCount(); x++){
        if(m_Table->isRowHidden(x) == false){
            m_Lists[2]->setRowCount(m_Lists[2]->rowCount()+1);
            m_Lists[2]->setItem(x, 0, new QTableWidgetItem(m_Table->item(x, ALBUM)->text()));
            m_Lists[2]->setItem(x, 1, new QTableWidgetItem(m_Table->item(x, ARTIST)->text()));
            m_Lists[2]->setItem(x, 2, new QTableWidgetItem(m_Table->item(x, GENRE)->text()));
            m_Lists[2]->setItem(x, 3, new QTableWidgetItem(m_Table->item(x, ALBUM)->text()));
            m_Lists[2]->setItem(x, 4, new QTableWidgetItem(m_Table->item(x, PATH)->text()));
        }
    }

    //Sort based on artist
    m_Lists[2]->sortItems(0, Qt::AscendingOrder);

    //Iterate over the rows and check for duplicate where both album and
    //artist are equal.
    //If found remove, but make sure that the entry kept has cover art if
    //possible.
    for(int x = m_Lists[2]->rowCount() - 2; x >= 0; x--){
        if(m_Lists[2]->item(x, 0)->text() ==
            m_Lists[2]->item(x+1, 0)->text() &&
            m_Lists[2]->item(x, 1)->text() ==
            m_Lists[2]->item(x+1, 1)->text()){
            if(!imageInFile(m_Lists[2]->item(x, 4)
                ->text().toUtf8().constData())){
                m_Lists[2]->removeRow(x);
            }
        }

        m_Lists[2]->removeRow(x+1);
    }

    //Iterate over the rows and check if there is a duplicate album name
    //Since we have already removed any duplicates of both artist and album,
    //if there are duplicates, add the artists' names in parentheses after
    //the album
    for(int x = m_Lists[2]->rowCount() - 2; x >= 0; x--){
        if(m_Lists[2]->item(x, 0)->text() == m_Lists[2]->item(x+1, 0)->text()){
            QString album1Text = m_Lists[2]->item(x, 3)->text() + " (" +
            m_Lists[2]->item(x+1, 3)->text() + ") " +
            m_Lists[2]->item(x, 1)->text() + " - " +
            m_Lists[2]->item(x+1, 1)->text() + ") " +
            m_Lists[2]->item(x, 4)->text() + " (" +
            m_Lists[2]->item(x+1, 4)->text() + ") " +
            m_Lists[2]->item(x+1, 0)->text() + ");
        }

        m_Lists[2]->removeRow(x);
    }
}
m_Lists[2]->item(x,1)->text() + "")";
QString album2Text = m_Lists[2]->item(x+1,3)->text() + "(" +
m_Lists[2]->item(x+1,1)->text() + ")";

m_Lists[2]->setItem(x,0,new QTableWidgetItem(album1Text));
m_Lists[2]->setItem(x+1,0,new QTableWidgetItem(album2Text));
}
}

//Add a row to the beginning and insert "(All)"
m_Lists[2]->insertRow(0);
  m_Lists[2]->setItem(0,0,new QTableWidgetItem("(All)"));
  m_Lists[2]->setItem(0,1,new QTableWidgetItem("(All)"));
  m_Lists[2]->setItem(0,2,new QTableWidgetItem("(All)"));
  m_Lists[2]->setItem(0,3,new QTableWidgetItem("(All)"));

//Set the selected row to row 0, corresponding to '(All)'
m_Lists[2]->setCurrentCell(0,0);
Cover Flow

Cover Flow is a visual way to explore a music library. It allows the user to see the album art from various albums in their library. As the user cycles through the artwork, the song table is filtered based on which album is in focus. While this is functionally quite similar to clicking a selection from the album list, images often convey something different from words. Thus, a user may want to explore the artwork rather than clicking through a list.

Figure 8, Cover Flow

Our implementation of coverflow extends QT’s built in QGLWidget class with a class called GLWidget.

The first thing that our widget does is to get the cover art images for each album in our list and convert into a GL texture. As noted in an earlier section, each row in albums table has the path to a file with cover art for the album, if such as file exists. If there is no cover art, we insert a default picture as a placeholder. We iterate over our album list as follows:

```cpp
void GLWidget::LoadGLTexture( const char * path, int i )
{
    //Load the image from file at path
    QImage img = imageForFile(path);

    //If the loaded image is null, load the default picture
    if(img.isNull())
        img.load("/home/administrator/Downloads/stock-photo-4466576-i-don-t-know.jpg");

    //Format the image into data readable by OpenGL
    QImage GL_formatted_image;
    GL_formatted_image = QGLWidget::convertToGLFormat(img);

    //Create the texture from the image
    glGenTextures(1, &m_records[i].texId);
    glBindTexture(GL_TEXTURE_2D, m_records[i].texId);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```
The textures are then mapped to squares, each representing a different album. A single album is rotated to be in the viewing plane. The remaining records are placed perpendicular the active record. Due to perspective, we see a profile of each non-focused record. The artist and album are printed on the screen and a filter is applied to the table so that we show only songs from the active record.
When the user presses the left or right key, the current record rotates out of focus and an adjacent record rotates to front. All other records are moved left or right one place. A filter is applied to reflect the new album.

```cpp
void GLWidget::scrollRecords(int dir)
{
    int middle = m_recordCount / 2 + m_shift - dir;
    // do nothing if we are already scrolling
    if(m_bScrolling) return;

    // init scrolling params
    m_bScrolling = 1;    // flag scrolling condition
    m_scrollDir = dir;   // save scrolling direction

    //Filter the song Table based on newly displayed record
    if(middle < m_parent->rowCount())
        m_parent->coverFlowFilter(middle);

    // register a timer callback (scrollTimer) to be triggered in a
    // specified number of milliseconds
    QTimer::singleShot(m_scrollUpdateInterval, this, SLOT(redraw()));
}
```

The scrollTimer function makes the calls to draw successive frames of the animation. At an interval specified by m_scrollUpdateInterval, scrollTimer updates the offset parameter to tell the draw function where in the process of the animation should be drawn and then calls the paintGL function to draw the records on the screen.

```cpp
void GLWidget::scrollTimer(int value)
{
    static unsigned int counter=0;

    // update velocity and position
    counter++;
    m_scrollOffset = (float)m_scrollDir * counter * m_scrollUpdateInterval / m_scrollTime;

    // last iteration
    if(fabs(m_scrollOffset) >= 1 ||
        counter >= m_scrollTime/(m_scrollUpdateInterval)) {
        m_shift = -m_scrollDir;
        counter = 0;
    }
}
m_scrollDir = 0;
m_scrollOffset = 0;
m_bScrolling = 0;
paintGL();
} else {
    paintGL();
    //Wait for the duration of update interval and draw again.
    QTimer::singleShot(m_scrollUpdateInterval, this, SLOT(redraw()));
}
Visualizer

(Note: In the current implementation, our visualizer is not connected to our waveform data. Amplitude values are calculated randomly.)

The visualizer in our program mimics an equalizer. Each bar corresponds to a frequency band. The size and color of each bar are modulated by the amplitude of the sound in that range.

Amplitudes
While the song is being played, at a specified interval, the amplitude of each frequency band is calculated. An amplitude value of zero represents silence and a value of one represent the maximum possible amplitude. The amplitude value is then input into a normalization function. We normalize the amplitudes to make sure that the values are distributed over a wide range. If the data is clustered on one extreme or another then the visualizer will be almost static. It is more aesthetically pleasing when the movements are more dramatic and dynamic.

While we want the movements to be dramatic, we also want to make the bars to seem animated, rather than flashing instantaneous readout of the sound being played. To accomplish this, we gave each bar “memory”. That is when one of the bands encounters a very high amplitude, we want the bar representing that band to convey the impact of the loud sound by staying large for a short period of time and slowly shrinking. To accomplish this we determine the amplitude of the bar based on the maximum of the new amplitude and the previous amplitude minus 0.005. So, every time we calculate a new amplitude, the bar grows if the new amplitude is above this threshold, or it slowly contracts. In this way the equalizer animates, rather than flashing different values. Because of the “memory”, when playback stops, instead of instantly going to the default inactive shape and color, the bars slowly shrink until the value of the amplitude reaches zero.

Figure 9, Visualizer (Song Playing)
```cpp
void GLVisualizer::setAmplitudes()
{
    float r[m_BandCount];
    for(int i = 0; i<m_BandCount; i++)
    {
        r[i] = (float)rand()/(float)RAND_MAX;
        r[i] = normalize(r[i]);
    }
    for(int i = 1; i<m_BandCount; i++)
    {
        //Linearly decrease the amplitude
        m_Bands[i].amplitude = m_Bands[i].amplitude - .005;
        //If not playing, input amplitude = 0
        if(m_Playing == false)
            r[i] = 0;
        //Set new amplitude to the max of current input and the linear decrease
        m_Bands[i].amplitude = max(r[i], m_Bands[i].amplitude);
        //Set the color based on the amplitude
        setColor(i);
    }
}
```

**Colors**

In addition to amplitude, each bar has a color. The color varies, utilizing green for soft sounds, yellow for medium and red for loud. In OpenGL, the value of a color is represented by a three dimensional vector, each component with values between zero and one. Transitioning from red to yellow to green only involves changing the values of red and green color channels. The value of the blue channel is constant.
at close to zero. To calculate the color, we constructed piecewise functions for the value of each color channel. The red channel is the minimum of 1 and double the amplitude. The green channel is given by the minimum of 1 and two minus double the amplitude.

![Color Channel Values vs Band Amplitude](image)

**Figure 11, Graph of Color Channels**

```c++
void GLVisualizer::setColor(int i)
{
    float color0, color1, color2, amplitude;

    amplitude = m_Bands[i].amplitude;

    //Amplitude = 0 corresponds to green, Amplitude = .5 corresponds to yellow
    //Amplitude = 1 corresponds to red
    //R increase 0 to .5 constant after
    //G constant 0 to .5, decreases after
    //B is constant
    color0 = min(1., amplitude*2.);
    color1 = min(1., 2-2.*amplitude);
    color2 = .05;

    //Set the color to current band
    m_Bands[i].color[0] = color0;
    m_Bands[i].color[1] = color1;
    m_Bands[i].color[2] = color2;
}
```
Size of Bars

Our visualizer draws a series of ten adjacent bars. Before the amplitude reaches a threshold of 0.5, the size and position of the bar remains constant, however the color may still change. If the amplitude reaches a value of 0.5, then the bar grows larger and moves forward toward plane of the screen.

```cpp
void GLVisualizer::drawBar(FrequencyBand *band)
{
    //Calculate the factors of modulation based on the amplitude of the band
    double distanceModulation = max((float)(band->amplitude-.5),(float)(0));
    double heightModulation = distanceModulation * 8;

    //Define the z-direction displacement based on amplitude
    double distance = -4 + distanceModulation;

    // draw filled polygon
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    glColor3f(band->color[0], band->color[1], band->color[2]);
    glBegin(GL_QUADS);
        glVertex3f(0,0,distance);
        glVertex3f(0,band->height + heightModulation,distance);
        glVertex3f(1,band->height + heightModulation,distance);
        glVertex3f(1,0,distance);
    glEnd();
}
```
Tag Edit Dialog

In addition to displaying information about media, most media players also give the user to edit the metadata. Most music purchased and downloaded from an online store will have the correct metadata. However, there are other ways to get media. A user may have added songs which they converted from one of their CDs or downloaded from a small band without a label contract. Many of these files may either have incomplete or incorrect information stored in the tags. Using a dialog, the user can correct these errors and fill in gaps.

![Edit Song Info](image)

*Figure 12, Tag Edit Dialog*

The user can access this dialog through the context menu associated with the song table. The user right clicks on the row of the song of which they would like to edit the information. Then the user clicks “Edit Info” and the dialog appears.

```cpp
void MainWindow::editInfo()
{
    QStringList *myData = new QStringList;
    int currRow = m_Table->currentRow();

    //Place the data into a string list to pass to the dialog
    for(int x = 0;x<COLS;x++){
        myData->append(m_Table->item(currRow,x)->text());
    }

    //Create the dialog, set the data and show the dialog
    ID3Dialog infoDialog(this);
    infoDialog.setData(myData);
    infoDialog.exec();

    //If the dialog made changes, update the song table
    if(infoDialog.madeChanges()){
        updateRow(currRow, myData);
    }
}
```
When the user clicks ‘Save’, the dialog checks if anything was updated. If any changes were made, the main window calls the writeTags function. This function uses methods built into TagLib to write the tags to the file, making the changes permanent. Notice that the items in the QStringList must be converted to a standard string before they can be passed as arguments in the TagLib functions.

```cpp
void ID3Dialog::writeTags()
{
    // Declare and initialize a string containing the path to the file
    // Note: TagLib does not accept standard strings
    TagLib::String myFileLocation = m_SongData->at(PATH).toUtf8().constData();

    // Create a reference to the file to be changed.
    TagLib::FileRef mediaFile(m_SongData->at(PATH).toUtf8().constData());

    // Set the tags to be written
    mediaFile.tag()->setArtist(m_SongData->at(ARTIST).toUtf8().constData());
    mediaFile.tag()->setAlbum(m_SongData->at(ALBUM).toUtf8().constData());
    mediaFile.tag()->setTitle(m_SongData->at(TITLE).toUtf8().constData());
    mediaFile.tag()->setGenre(m_SongData->at(GENRE).toUtf8().constData());
    mediaFile.tag()->setTrack(m_TrackSpinBox->value());

    // Write the tags to the file
    mediaFile.save();
}
```

After the tags are written, the table is updated to reflect the changes. In our current implementation, users can edit only the fields which appear in the song table.
Future Improvements

While our program has a good base of features, there are many improvements we would have to make before anyone would want to use our program as their primary music player. A music player and library organizer is only as good as the user interface. People want easy ways to be able to find, play and arrange their music.

Graphical Improvements

Font Size: While there are users who prefer a larger font, many users would be prefer to see more information on the screen at once. In iTunes, the font size is quite small in the table containing the songs. This allows users to browse their libraries much more quickly.

Icons: Icons not only make a program look more polished, but they save space as compared to using text. In an update our program, we would use icons for playback control.

Features

Playlists: The most important feature to make our program usable would be to implement playlists. Most of the time, users would like to select more than one song to play at a time and save lists of songs to play at a later date.

Searching: One of the easiest ways for a user to find the song they are looking for is to simply type in the name of the song or artist. This is especially important when library become large. The more songs in the library, the more cumbersome it is to scan the library for the desired song. Aearch functionality would greatly improve the usability of our program.
What We Learned

*Using External Libraries:* Any advanced program will need external libraries. While it is not difficult to use them, we had never done it before. Knowing how to install a library and use it to compile a program will be important knowledge going forward.

*QT:* Learning how to use QT is a valuable skill. You can build very advanced programs using QT and it greatly eases the process of designing an interface.

*Divide and Conquer:* The most important thing we learned from creating qTunes was how to break down a large programming task into smaller pieces. When creating the visualizer or filtering, the end result is easy to explain. However, it is not quite so easy to see how to get to that end result. You have to break the process down into steps. Some of these steps may not even be programming. They may be researching the best way to accomplish something or drawing diagrams to explain a process or interface.

The larger the task the more important it is to break it into steps. For example, in our program we mapped album art to GL textures. If you search for an example of how to do this, you may not find anything. However, you can divide this task into parts. The first is how to recover the album artwork. The second is how to create an OpenGL texture from an image file. It is easy to find examples of both of these individual parts. When you combine the parts, you achieve the original goal.

*Flexibility:* When creating a program, it is important to be flexible. The first way you thought to implement an idea may not be the best. For example, at the beginning of our project we used Audiere to for playback and to get metadata. Yet there proved to be superior methods of accomplishing both tasks. If we had continued to use Audiere, we could not have edited the tag data or retrieved the cover art from the music files. Even though we had to reimplement everything we had done before, subsequent parts of our program were easier to complete because we did.