Nonlinear Music in Direct X

Bachelor Project Computer Science

06/07/2006

Free University Amsterdam

Teunis van Wijngaarden

- Student Computer Science (2003)
- Student number: 1397214
- E-mail: teunis@few.vu.nl

1. Contents

2. Introduction

This document contains all textual deliverables for the bachelor project about DirectX Audio. It contains programming examples that are explained in a tutorial-like style. The final product, a Nonlinear Audio player, is also explained in this document. An evaluation and resources can be found in the back.

Together with the document come some programming examples and the final program. The programs can be found as Visual C++ project[1] and c++ '.cpp' files. The executables are added to run the programs immediately.

[1] The project files might not work on every computer. Please contact the author, or use only the cpp files if you have problems running the project-files.

3. The DirectMusic API

Direct Audio takes away a lot of work from the programmer. A programmer that uses this API, especially DirectMusic, does not need to know about audio-buffers. There is no need to 'manually' manipulate bytes. The API contains a lot of features that are easy to use compared to buffer manipulation. For example, using 3D audio with DirectMusic is done in only a few statements. Especially for game programmers this is very useful. For advanced audio editing software, the loss of buffers might not be so useful. If you want you can still get the buffers and play with them using DirectSound.

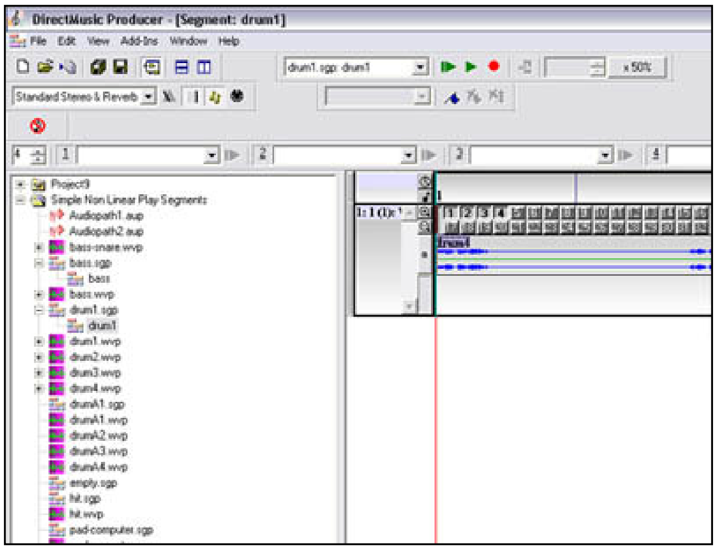
DirectSound is created in 1995. It is quick and efficient. Later on DirectMusic was produced, which is 'higher level'. Direct Audio can be seen as DirectMusic together with DirectSound. For more information of the history and layers in the Direct Audio model, I refer to the book "DirectX 9 Audio Exposed" in section 7.1.

3.1 Useful software: DirectMusic Producer

At the DirectX pages on the Microsoft website[2] DirectMusic Producer is described as:

> *"DirectMusic® Producer is the authoring tool that allows composers and sound designers to create content for DirectX audio."*

The tool can be used to create and edit segments. Segments are the playable unit in DirectX Audio. Segments have certain characteristics that can be added during runtime also, but using DirectMusic Producer is easier. Segments can be created from wav files.



Figuur 1 DirectMusic Producer

In DirectMusic Producer segment variations can be added. Every time the segment is played, another variation is chosen. More on this in section 5.2.2.

3.2 Building blocks

This subsection gives an overview of the part of the API that we use. The coherence of the classes will become clearer as you try to use them. The DirectMusic API consists of a number of classes of which a few are needed for our purposes. This document discusses those classes. The complete list of DirectMusic Interfaces can be found in the MSDN Library Archive[3]. This document is easier to understand if you read literature on how the DirectX audio system

[2] http://www.microsoft.com/downloads/details.aspx?FamilyID=07f29ce2-1c03-4aef-b5b0-cbdaf07af08d&DisplayLang=en

[3] http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/directx9_c/directx/htm/directmusicinterfaces.asp

works. You can use DirectX with little knowledge on the system, but I do not advise you to do this.

DirectMusic's playable unit is a segment. Everything that has to be played by your application should be in a segment or should be loaded in a segment at runtime. You can create segments from wave files or produce MIDI segments with DirectMusic Producer. There is a special segment file-type ('.sgt') that can be loaded into programs. Segments are of type **IDirectMusicSegment8**. A playing instance of a segment is represented by a segment-state of type **IDirectMusicSegmentState8**. It can be used to retrieve information about the playing segment. A segment-state can be used to start a new segment directly after the previous finished.

Segments are loaded by your application by using an object of type **IDirectMusicLoader8**. The loader holds a list with loaded files and their location. It has a mechanism to avoid loading files twice. We will not discuss this, because its focus is on MIDI usage. The functionalities that we use are setting a source directory and loading segment-files (.sgt) and wave-files (.wav) into segments.

The performance object is the heart of a player that uses DirectMusic. This object manages playback. The **IDirectMusicPerformance8** interface contains functions for playback (play, stop) and creating audiopaths. Segments can be played as primary or as secondary segments. Only one primary segment can play at once. The primary segment determines tempo and other overall parameters of the musical performance. Secondary segments play on top of the primary segments. Multiple secondary segments can play at once. Even the same segment can be played multiple times at once. The performance interface contains many more functions that we won't explain nor use.

Audiopaths are of type **IDirectMusicAudioPath8**. According to the MSDN description[4] an audiopath manages the stages of data flow from the performance to the final mixer. We will use this interface only when it comes to 3D audio.

With these interfaces we have the buildings blocks to build an audio application.

[4] http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/directx9_c/directx/htm/idirectmusicaudiopath8interface.asp

4. Using the API

This section explains how to use the DirectMusic API. After reading this you will be able to play sounds, change 3D parameters of sound and consecutively play segments to create a ‘song’. This section contains some sample code. To keep the examples simple, we put all the code in the main section of the program. The programs have no GUI, but use minimal console output.

All DirectX objects are COM objects. This document does not discuss COM. If you are not familiar with COM, you might want to read the MSDN Library page about the model[5]. You might skip this and take the statements that have to do with COM for granted.

4.1 Creating the Loader and Performance

For every application we need the loader and the performance. We create these objects using the COM system. We first need to initialize the COM system.

```
// Initialize COM
CoInitialize(NULL);
```

We can now create the loader and performance object by calling `CoCreateInstance()`. Please read the comments.

```
// Create Loader
CoCreateInstance(
CLSID_DirectMusicLoader, // Class ID
      NULL, // No Aggregate -> Not important to us
      CLSCTX_INPROC, // The context is this program
      IID_IDirectMusicLoader8, // Request the interface
      (void**)&pLoader // Return a pointer to the loader
);

// Create Performance
CoCreateInstance(
      CLSID_DirectMusicPerformance, // Class ID
      NULL, // No aggregate -> Not important to us
      CLSCTX_INPROC, // The context is this program
      IID_IDirectMusicPerformance8, // Request the interface
      (void**)&pPerformance // Return a pointer to the performance
);
```

Most important is the pointer to the object. The pointers are declared at the beginning of the `main` section:

```
IDirectMusicLoader8* pLoader = NULL; // Manages audiofile I/O
IDirectMusicPerformance8* pPerformance = NULL; // The heart
```

These pointers now point to the right objects. These objects are not yet initialized. They are only created by the COM system. When the program finished doing its job, you have to closedown and release the performance, release the loader and uninitialize the COM.

```
pPerformance->CloseDown();
pPerformance->Release();
pLoader->Release();
```

[5] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/com/html/f5f66603-466c-496b-be29-89a8ed9361dd.asp

```
// Uninitialize COM
CoUninitialize();
```

4.2 Simple player

The simple player can play a wav-file. These are the basics of using DirectMusic. The solution can be found in the *'example programs'* directory accompanied with this document.

4.2.1. Initialize performance

Before we can use the `pPerformance` object, we need to initialize it.

```
// Initialize Performance
pPerformance->InitAudio(
      NULL, // Pointer to DirectMusic object, if NULL an object is created
      NULL, // Pointer to DirectSound object, if NULL an object is created
      NULL, // Window handle -> Not important to us
      DMUS_APATH_DYNAMIC_STEREO, // Defines which audiopath to use
      2, // The number of Performance Channels
      DMUS_AUDIOF_ALL, // Flag that speficies wanted features, like buffers
      NULL // Specifies parameters for synthesizer, use defaults if NULL
);
```

The first three pointers are not important for this and the final application. The fourth flag defines the type of the default audiopath to use. There are four possible values:

- `DMUS_APATH_DYNAMIC_3D`,
- `DMUS_APATH_DYNAMIC_MONO`,
- `DMUS_APATH_DYNAMIC_STEREO`,
- `DMUS_APATH_SHARED_STEREOPLUSREVERB`.

It is possible to set the default audio path in a later stadium by using `SetDefaultAudioPath` in the `IDirectMusicPerformance` interface. It is also possible to play a segment on a given audiopath.

If the default audiopath type is non zero (as in my example), we have to define the number of channels. For this wav file we only need two channels[6].

The features-flag is set to all features. We do not need to know what all features are. With this flag set to all, we can do everything we want including using 3D buffers.

The last parameter is also not of interest to us.

4.2.2. Setting up the loader

The next step is loading the files. We use a loader for this, which we point at with `pLoader`. To bring a little structure in our program, we put the samples in a subdirectory. Now, we need to let the loader know in which subdirectory to search for samples.

We use the following declaration at the function start:

```
char searchPath[MAXPATH] = "wavs";
WCHAR wSearchPath[MAXPATH];
```

[6] Please read our resources for more information on channels.

Setting the directory goes like this:

```
// Configure Loader
MultiByteToWideChar(CP_ACP, 0, searchPath, -1, wSearchPath, MAXPATH);
pLoader->SetSearchDirectory(
      GUID_DirectMusicAllTypes, // The file type(s) the directory contains
      wSearchPath, // Path for the directory, can be relative
      FALSE); // Clears the previous entries
```

In the example file some error detection code is added.

The first statement is for converting the pathname parameter to the right format. I will not explain this. The `SetSearchDirectory` function sets the directory. We specify that all filetypes can be found in the directory, this includes: segments, wav files, audiopaths, styles etc. The second parameter defines the directory name. The last parameter clears the previous entries in the search directory table if `TRUE`. The objects that are already loaded remain in the list. The list can be cleared to avoid name clashing.

4.2.3. Loading files into segments

In order to play a segment, we first have to load a wav or segment file into a segment object. This is a task for the loader. The function `LoadObjectFromFile` can be used for loading files into an `IDirectMusicSegment8`. This function can also load filetypes like audiopaths, styles and chordmaps, but we won't use these files.

```
// Load WAV: drums
pLoader->LoadObjectFromFile(
      CLSID_DirectMusicSegment, // Identifier for the class of object
      IID_IDirectMusicSegment8, // Identifier for the interface
      L"drums.wav", // Filename, can be relative to the search directory
      (LPVOID*) &pSegmentDrums); // Variable that receives the pointer
```

In the example file some error detection code is added.

The first two parameters define the type of the class to which the file has to be loaded. We only use the IDirectMusicSegment. The third parameter contains the filename. The fourth parameter receives a pointer. This variable is declared as `IDirectMusicSegment8*`.

The file is now loaded into a segment, but the performance does not know about this segments. The segment has to be downloaded into the performance before it can be played.

```
pSegmentDrums->Download(pPerformance);
```

In the example file some error detection code is added.

When we are done with playing the segment, it should be unloaded and released.

```
pSegmentDrums->Unload(pPerformance);
pSegmentDrums->Release();
```

4.2.4. Playing segments

Only short fragments (samples) will be used for the final Nonlinear Music Application. This requires some samples to be repeated over time. In the simple wav player, we want to repeat a measure of the drums. Therefore we have to set the repeats.

```
// Configure Segments
pSegmentDrums->SetRepeats(4);
```

We are now ready to play the segment, which is a task of the performance.

```
// Play Segments
pPerformance->PlaySegmentEx(
      pSegmentDrums, // Segment to be played
      NULL, // Reserved. Set to NULL
      NULL, // Pointer to template segment, only used for transitions
      0, // Flag that modifies behaviour (primary, secondary, timing etc.)
      0, // Start time, not available for wav files
      NULL, // Segmentstate to write to
      NULL, // Pointer to segmentstate or audiopath to stop
      NULL // Pointer to audiopath on which to play
);
```

The first parameter points to our segment. We won't use the second and third parameter. The fourth parameter is a flag to define if the segment is a primary or secondary segment (default: primary). This flag can also be used to change the start time and handle other timing issues. For instance, a secondary segment can be played on a marker in the primary segment, more on this in section 5.2.2.

The fifth parameter defines the start time. This is not available when playing a wav files. When wav files are loaded to a segment at runtime, the length is unknown. The length of the segment is zero, so the start time cannot be used.

The seventh parameter is a pointer to a segment-state. This segment-state receives all information about the playing segment. This state can for instance be used for playing another segment.

The eighth parameter points to another segment-state or an audiopath. You can define that a segment should start after another segment (-state) or all segments on an audiopath finished playing. So this segment-state is used for reading; it checks if the segment(s) belonging to this state or path finished playing.

The ninth parameter points to the audiopath on which to play. We use the default audiopath.

The `PlaySegmentEx` function is a non-blocking function. This means that the code after this function call is executed immediately, even if the segment is not finished (or even started) playing. This means that we have to sleep or loop while playing to keep the program alive.

```
// Sleep while playing
Sleep(15000);
```

The `IsPlaying` function of the performance does not work properly for wav-files, it returns `FALSE` when the segment is still playing, so we use `Sleep` to 'block' the program.

4.3 Playing 3D sound

3D sound is especially useful in games and movies. In Nonlinear Music, it can be used for playing sound-effects during a 'song'. This example shows how to manipulate the 3D position of the sound of a duck. The 3D manipulation makes it sound as if the duck flies over your head.

Setting up COM, loader and performance is all the same as in the previous example. Loading, unloading and releasing the files and segments is also nothing special with 3D sound. The focus in this subsection is on playing the sound and manipulating the 3D position.

We load a segment file ('.sgt') for the sound of the duck. The reason to play a segment instead of a wav file is that wav files have no specified length. The 3D player runs as long as the duck file plays, so we use a segment file which has the length attribute set. More on segments will follow in the last example.

As a background sound we play recorded rain. Playing this sound is as easy as the previous example:

```
// Configure Segments
pSegmentRain->SetRepeats(16);

// Play Segments
pPerformance->PlaySegmentEx(
      pSegmentRain, NULL, NULL,
      DMUS_SEGF_SECONDARY,
      0,
      NULL, NULL, NULL
);
```

The `DMUS_SEGF_SECONDARY` flag is not important for now. Nothing special happens when this statement is executed, it only starts raining.

Now, we get to the 3D part. In order to play 3D sound, a 3D AudioPath should be used. The performance can create such a path with the method `CreateStandardAudioPath`. It is also possible to create an audiopath in DirectMusic Producer and load it at runtime, but we won't use this method.

```
// Create 3D audio path
pPerformance->CreateStandardAudioPath(
      DMUS_APATH_DYNAMIC_3D, // Type of the path
      2, // Number of Channels
      TRUE, // If TRUE, activite path on creation
      (IDirectMusicAudioPath**) &p3DAudioPath // Receives the pointer
);
```

The possible types of paths and channels are mentioned in section 4.2.1. We want to activate the path immediately and therefore set parameter three to `TRUE`. Traditionally the last parameter receives the pointer to the path. This is the declaration for this pointer:

```
IDirectMusicAudioPath8* p3DAudioPath = NULL; // The 3D audiopath
```

One way to view an audiopath is as a path that leads through several engines and mixers. One of the objects in this path is the 3D buffer. We need this buffer to manipulate the 3D position. The function `GetObjectInPath` retrieves an interface for a given object in the path. Please read the comments

```
// Get 3D buffer
p3DAudioPath->GetObjectInPath(
      0, // Performance channel to search -> the first
      DMUS_PATH_BUFFER, // Stage in the AudioPath
      0, // Index of the buffer -> Not of interest
      GUID_NULL, // Identifier of the object, ignored
      0, // Ignored together with DMUS_PATH_BUFFER
      IID_IDirectSound3DBuffer, // Identifier of the Desired interface
      (void**)&p3DBuffer // Receives the pointer to the requested interface
);
```

The second parameter defines that we want to retrieve the buffer. The identifier of the object (fourth parameter) is set to GUID_NULL because there is only a single class of object in this stage in the audiopath. In the end we have a pointer to the buffer. Note that the buffer is in the DirectSound API.

We can now play the duck-sound via our 3D audiopath.

```
pPerformance->PlaySegmentEx(
      pSegmentEffect, NULL, NULL,
      DMUS_APATH_DYNAMIC_3D, // It is a 3D sound
      0,
      NULL, NULL,
      p3DAudioPath // We use the path from which we have the buffer
);
```

Now it is time to manipulate the location of the sound in 3D space. We put this code between curly brackets so that the 3D manipulating code is all together. The examples include many comments for learning purpose.

```
{
      MUSIC_TIME segmentLength;
      MUSIC_TIME performanceCurrentTime;
      MUSIC_TIME segmentEndTime;

      // Compute the length end time of the duck sample in order
      // to stop changing the 3D position when finished playing
      pSegmentEffect->GetLength(&segmentLength); // Get length of sample
      pPerformance->GetTime(NULL, &performanceCurrentTime); // get time

      // compute the end time
      segmentEndTime = performanceCurrentTime + segmentLength;

      // Set the initial 3D position
      double x = 0;
      double y = -1;
      double z = -1;

      while (performanceCurrentTime < segmentEndTime) {
            // Loop while endtime not reached
            //x = (x + 0.01); Do not change x
            y = (y + 0.1); // Change the y position
            z = (z + 0.01); // Change the x position

            // change position
            p3DBuffer->SetPosition(x,y,z,DS3D_IMMEDIATE);
            Sleep(100); // Sleep some time to change position in time

            // Get current time in order to loop
            pPerformance->GetTime(NULL, &performanceCurrentTime);
      }
}
```

Some `MUSIC_TIME` variables are declared to use later in a computation. The segment-length is retrieved and the end time is computed. This end time is used to loop ‘until the end of the duck sound’. The initial 3D position is set before the while loop starts, we do this because want the duck to start from the initial position and not from the centre of the 3D space. We change the y and z position every 100 milliseconds (`Sleep(100);`). With the `SetPosition` function of the buffer interface the position can be set. This requires a flag, which we set to `DS3D_IMMEDIATE`. This means that the setting are applied immediately. With this flag set to `DS3D_DEFERRED` the settings are not applied until you invoke a special function. This allows you to change multiple settings at a time.

4.4 Parallel and sequential multiple segment player

Up until this point, we only played from wav files. We can also play segments, stored on disk as ‘.sgt’ files. The benefit of this is that we can define the number of repeats in the file. It is also possible to add tracks, markers and variations to a segment. More on this in section 5.2.2. For now we will load wavfiles to ‘.sgt’ files and play those. We can use DirectMusic Producer for creating those segment files. We will use the functionalities of segment-files in the final application.

Playing '.sgt'-files is as simple as playing wav-files. Instead of loading the wav-file we now have to load the segment-file. When it is loaded, every segment can be handled the same way.

The example code is the same as the code used in the application of section 4.2, up until setting the loader directory. This subsection will discuss loading and playing. Releasing and uninitializing goes as stated before.

4.4.1. Loading Files

Now, we have to load multiple files. We use a function for this to avoid repeating code. First, we have to fill two arrays with the filenames of the files we want to use.

```
// Declarations for filename-arrays
int numberOfDrums = 0;
int numberOfPercussion = 0;
string wavFileNameDrums[MAX_SAMPLE_COUNT];
string wavFileNamePercussion[MAX_SAMPLE_COUNT];

// Hard coded filenames
numberOfDrums = 3; // Array size
wavFileNameDrums[0] = "drum1.sgp";
wavFileNameDrums[1] = "drum2.sgp";
wavFileNameDrums[2] = "drum3.sgp";

numberOfPercussion = 1; // Array size
wavFileNamePercussion[0] = "perc.sgp";
```

A special function `loadFiles` now loads the files in the array into segment arrays. We use a function with this header:

```
int loadFiles(string wavFileName[], int wavFileCount, IDirectMusicSegment8*
pSegment[], int segmentCount, IDirectMusicLoader8* pLoader,
IDirectMusicPerformance8* pPerformance);
```

Given a filename-array (and size), a segment-array (and size) and the loader and performance this function loads the files, puts them into the segments in the segment-array and downloads the segments into the performance. After the function call, the program can use the segments in the array. Of course the program has to unload and release the segments in the end. We do not do this in an automated way, because this is not feasible with this array-size.

The code of the `loadFiles` function is not included in this document. It boils down to the code in segment 4.2.3. in a loop.

4.4.2. Playing files

As we want to play segments in parallel and sequential (directly after each other), we need to use secondary segments and segment states. We first discuss the primary segment.

```
pPerformance->PlaySegmentEx(
      pSegmentPercussion[0], // Play the percussion
      NULL, NULL, // Not used
      0, // Play as primary
      0, // Play from the begin
      NULL, // No segmentstate
      NULL, // No segmentstate
      NULL // Use default AudioPath
);
```

The repeats are defined in the segment. We created the segment file with DirectMusic Producer and set the repeats to 7, so it plays 8 times in total No parameters of `PlaySegmentEx` need explanation.

The secondary segments need some more explanation. `PlaySegmentEx` is called four times directly after each other to play the drum segment. We let the drums vary to show how to use the secondary segments.

```
pPerformance->PlaySegmentEx(
      pSegmentDrums[0], // Play the drums
      NULL, NULL, // Not used
      DMUS_SEGF_SECONDARY, // Play as secondary segment
      0, // Play from the begin
      (IDirectMusicSegmentState **) &pSegmentStateDrums,
      // Write to this segmentstate
      NULL, // No segmentstate to read from
      NULL // Use default AudioPath
);
pPerformance->PlaySegmentEx(
      pSegmentDrums[2], NULL, NULL,
      DMUS_SEGF_QUEUE | DMUS_SEGF_SECONDARY,
      // Play as secondary, directly after the segment belonging to the
      // defined segmentstate finished
      0,
      (IDirectMusicSegmentState **) &pSegmentStateDrums,
      (IDirectMusicSegmentState*) pSegmentStateDrums,
      // Segmentstate belonging to the segment after which this segment
      // should be played
      NULL);

pPerformance->PlaySegmentEx(
      pSegmentDrums[1], NULL, NULL, DMUS_SEGF_QUEUE | DMUS_SEGF_SECONDARY,
      0, (IDirectMusicSegmentState **) &pSegmentStateDrums,
      (IDirectMusicSegmentState*) pSegmentStateDrums,NULL);

pPerformance->PlaySegmentEx(
      pSegmentDrums[0], NULL, NULL, DMUS_SEGF_QUEUE | DMUS_SEGF_SECONDARY,
      0, (IDirectMusicSegmentState **) &pSegmentStateDrums,
      (IDirectMusicSegmentState*) pSegmentStateDrums, NULL);
```

All segments play with the `DMUS_SEGF_SECONDARY` flag set. This means that these are secondary segments and play together with the primary segment. The first segment to play starts directly at the invocation of the function and writes to the `pSegmentStateDrums` segment-state. This segments plays for three measures.

The second segment to play does not repeat; it plays for one measure. The `DMUS_SEGF_QUEUE` flag is set. This means that the segment is queued and starts playing after the previous segment with `pSegmentStateDrums` is finished playing. The segment-state to wait for is defined as seventh parameter.

The segments `pSegmentDrums[1]` and `pSegmentDrums[0]` are played on the same way. Together they fill 8 measures of drums, the same count as the percussion.

5. The Nonlinear Music Application

5.1 Nonlinear Music

Nowadays the music that is played on the radio and music that is for sale in the record store is linear. *Linear Music* is just a recording of one performance (live or in studio) that can be played over and over again. The music is exactly the same every time it is played. Nonlinear music is like a live-performance: the same song sounds different every time you play it.

5.1.1. Application of Nonlinear Music

With nonlinear music it is possible to let a song sound slightly different every time it is played, like a live performance. Some people like live performances above the recordings on cd. Nonlinear music makes it possible to repeat a musical score over time without getting bored by the repeating linear track.

Because of the variation a score can be played until infinity without repeating itself even once! This makes nonlinear music suitable for use in elevators or waiting rooms. This is also useful for radio disk jockeys. Radio stations often use background music for news-items, traffic information, the weather and other items. This background music is most of the time a looped instrumental record that repeats itself until the DJ finishes his/her item. Nonlinear music could make the background music less annoying. Of course, the variations must not distract the listener from the news-item.

For this application I use the radio example as a starting point. This requires the music to be instrumental. This use of the application allows us to produce less harmonic music and more dance style, which makes it easier to combine pieces of music. We add some sound-effects that can be used during the talk of the DJ or between items. This can be for example a car horn during the traffic information.

5.1.2. The example program

In short, the application should play recognizable music that is different every time you play it. The music should not contain repeating parts that repeat always on the same way, we need variation. The DJ should be able to play sound effects that fit in the musical background.

5.2 Sound Design

Choosing sounds randomly and playing them in parallel requires the sounds to be designed for this purpose. Choosing the right sounds is really important; this adds another difficulty to composing for the Nonlinear Music Player.

More advanced Nonlinear Music Players should be configurable. It would be nice if the composer could supply rules and information about samples so that the player ‘knows’ what can be played together and what not. I did not include such advanced rules. This makes that the sounds for my application should be chosen and designed very carefully. It also limits the possibilities of my linear player.

5.2.1. Choosing the right sounds

Although Nonlinear Music is defined as non-repeating, my application repeats some samples a number of times. The changing composition of different samples and the slight variation (created with DirectMusic Producer) makes the non linearity. In repeating samples, the loop points should not be notable. The beginning of a sample that has to be looped should closely connect to the end.

As samples are played in parallel, attention should be paid to volume. The best thing to do is first normalize the samples. During compose time volume can be adjusted to create the desired sound. A more advanced player might adjust volume on run time, but I do not include this.

Another consequence of playing sounds in parallel is that attention should be paid to harmonics. The central C on a piano can be played together with the neighbour B, but it sounds awful. Therefore I use non-melodic and non harmonic samples. A dance or R&B style can be chosen to avoid bad harmonics.

For more advanced players it would be useful if the composer of the nonlinear track could state rules like “If sample A is playing, then sample B must not be chosen”. It might make composing more difficult, but then it is possible to use samples that don’t sound nice when played together and tell the program not to play them together. I do not include such rules.

Not only harmonics, but rhythm is also important for good consonance. Of course tempo and half-measure type should be the same for every sample. Also the length should be equal, because the samples directly follow each other. This could be solved for short sounds by adding silence.

Because all instruments will play together by the nonlinear player, we need an ‘empty sound’. This is just recorded silence. See section 5.3.3.

All these rules do not hold for sound effects. Sound effects can be played at any moment the DJ presses the button. They start at a suitable point, for instance on a measure start. Because these are ‘one shot’ sounds the above rules do not hold. Of course attention should be paid to harmonics and rhythm.

5.2.2. Using DirectMusic Producer to design sounds

This subsection only describes the functionality of DirectMusic Producer (DMP) that I used. This is not a manual for this program. If you are interested in *how* to do the things described in this section, please consult the manual.

DMP can be used to create segments from wav files. The chosen files for the player can each be put into a segment. The segment is saved as a ‘.sgt’ file, together with the wav data in a ‘.wvp’ file.

It is possible to put multiple wave files into one segment. One way is creating multiple tracks that play at once. The other way is creating variations. The latter is of interest to us. A segment can hold variations that are played in a random way. Say for example that you have two sounds: (1) bass drum and snare, (2) bass drum, snare and hihat. Now you make one segment and add these two sounds as variation. Every time you play this segment, one of the two variations is chosen. So, some measures are played with hihat and some without hihat. The application does not control this choice, it is chosen by DirectX Audio. Variations can be chosen in random order, in order from first, in order from random, with no repeats and shuffling. This can be configured in DMP. These options will do a great job in Nonlinear Music!

I use the variation functionality only for little variations, because I want my application to decide about great variations. My samples are repeated every four measures and I do not want a totally different sound every measure. So, my variations are for example: “sound the bass drum one time extra this measure”.

A segment has many more options of which we use only a small subset. A segment holds information about the number of repeats. Some wav files that come with my application are only one measure. Because variation is often by four measures, I want to repeat these samples three times. I design the segments to do this.

Segments also hold information about markers. These markers can be placed on a special marker track inside the segment. Such a track can be added with DMP. Markers can be used as a cue point for other segments. In my application the sound effects are started on a cue point of the primary segment. Therefore I added a marker track with markers to the primary segments. It may for instance sound nice to play sound effects only at the start and in the exact middle of a measure. This can be done by placing a marker at those points.

Again: for information about how to create variations, set repeats and add markers please consult a book, manual or DirectMusic Producer help.

5.3 Playing Nonlinear Music with the DirectMusic API

This subsection contains less code than the short examples in section 4. Setting up performance and loader, loading files and playing files is assumed to be known. The code of the Nonlinear Player is provided with vast comments.

5.3.1. Using a player class

The Nonlinear Music Application makes use of a separate `NonlinearPlayer` class. This player needs an already created performance to play nonlinear music. There is a method to configure the player from a file. This is about loading the files. A `NonLinearPlayer` object can be asked to play a number of measures. During play, it can play an effect on the next marker in the primary segment by pressing a key. The list of segments to play is maintained in the `NonlinearPlayer` class.

The performance that the `NonlinearPlayer` class uses has to be created and set up in the main program. By doing it in this way, the main program still has control over volume, audiopaths etc.

A new player object is created in the `main` function of the application. The only thing the main function does is control the player.

5.3.2. The main structure

To make the code easy to understand, most of the work is done in the `main` function. That means the program itself does not contain many functions. This section describes the `main` function.

The first step in setting up audio is creating and setting up the performance and loader. Setting up the loader means setting a directory to find sample files. Setting up the performance is done by calling the `initAudio` function. To keep the `main` function clean, this is put in a separate function. Please read the code if you are interested in these functions.

The peace of code in which the player is really set up and controlled is placed between curly brackets.

```
// Load wavs & segments and play
{
	int measures = 0;
	char configFile[256];
	char keyc;
	ifstream file;
	NonlinearPlayer player(pPerformance);

	cout << "Give in the Configuration File: ";
	cin >> configFile;

	file.open(configFile);
	if (file.is_open() == FALSE) { cout << "Failed to open file.";
	exit(1); }

	cout << "How many measures do you want to play? [1-20] ";
	cin >> measures;
	if (measures  > MAX_PERF_LENGTH || measures < MIN_PERF_LENGTH) {
	cout << "Invalid number of measures."; exit(1); }

	player.configurePlayerFromFile(pLoader, &file);

	player.play(measures);

	while (player.isPlaying() == TRUE) {
		while (player.isPlaying() == TRUE) {
			cout << "-> ";
			cin >> keyc;

			if (keyc == KEY_QUIT) break;
			else if (keyc == KEY_EFFECT) player.playEffect();
			else cout << "Doesn't recognize command.\n";
		}
		if (keyc == KEY_QUIT) break;
		Sleep(100);
	}

	player.closeDown();
}
```

The name of the configuration file and the number of measures to play are asked from the user. The file is opened and the file descriptor is given to the `configurePlayerFromFile` function. The `NonlinearPlayer` object now reads the file and configures the player. This object also closes the configuration file!

The number-of-measures variable is used to call the `play` function. This function does not only start the music, it first ‘composes’ the music. More on this in section 5.3.3. The maximum number of measures is set to 20. This means that this application can not play until infinity. This is not possible because it composes its music before playing. An infinite song will require an infinite array and thus infinite memory. Advanced audio players could solve this by composing during play.

While the player is playing, a sort of command line is provided to the user. Only two commands are accepted. The command ‘q’ quits the program and ‘e’ causes a sound-effect to play. Playing a sound-effect is done by the player not by the main program. The player object has information on the right time to play, namely on a marker in primary segment.

The double while loop with the `player.isPlaying()` invocation needs some explanation. The player plays the music by concatenating segments. The `isPlaying` function just checks if one of the segments is currently playing. The sad thing is that between two segments (measures) the function that checks if a segment is playing returns `FALSE`. In the unlucky case this could cause the while loop to stop. In order to catch this fall-thru, the loop is placed into another loop, what reduces the change of falling thru.

In the end the player is closed down, which means the release of segments. This method doesn't do anything with the performance or loader. Closing down and release of the loader and performance has to be done by the main program.

5.3.3. The NonlinearPlayer class

A special class is created to handle the real playing part. This is the `NonlinearPlayer` class.

```
class NonlinearPlayer {

 public:
   NonlinearPlayer(IDirectMusicPerformance8* pPerformance);
   int playEffect();
   int play(int measures);
   int playNext(int index);
   int setPrimary(IDirectMusicSegment8* pSegment[], int segmentCount);
   int setEffect(IDirectMusicSegment8* pSegment);
   int insertSecondary(IDirectMusicSegment8* pSegment[],
                       int segmentCount);
   int configurePlayerFromFile(IDirectMusicLoader8* pLoader,
                               ifstream *stream);
   bool isPlaying();
   int closeDown();

 private:
   IDirectMusicPerformance8* pPerformance;
   IDirectMusicSegmentState8* pSegmentState[2];
   IDirectMusicSegment8* segmentPrimary[MAX_VARIATION_COUNT];
   IDirectMusicSegment8*
            segmentSecondary[MAX_SECONDARY_INSTR][MAX_VARIATION_COUNT];
   IDirectMusicSegment8* segmentEffect;

   int loadFilesToSegments(string wavFileName[], int wavFileCount,
                           IDirectMusicSegment8* pSegment[],
                           int segmentCount,
                           IDirectMusicLoader8* pLoader);

   int secondarySegmentCount;
   int segmentPrimaryCount;
   int segmentSecondaryCount[MAX_SECONDARY_INSTR];
   int currentSegmentState;

   int primarySequence[MAX_PERF_LENGTH];
   int secondarySequence[MAX_SECONDARY_INSTR][MAX_PERF_LENGTH];

   int fillSequences(int measures);

};
```

Many of the functions have an integer return type in order to build in error correction in a later stadium. A little error correction code in the nonlinear player is already added. Because this is not a consumer product most errors are not caught. If an error occurs the program will probably quit.

Below is a brief explanation of the public functions. For more information on the actual work done in the function, please read the code. The comments will guide you through the code.

- `NonlinearPlayer(IDirectMusicPerformance8* pPerformance);`
  This is the constructor. It just sets the performance (a private variable) and some other private variables to their initial value.

- `int playEffect();`
  This method causes a sound-effect to play at the next marker in the primary segment. This is done by setting a special flag in the `playSegmentEx` class of the performance interface.

- `int play(int measures);`
  This methods calls `fillSequences` to fill an array with segments to play. For every measure it calls `playNext` to actually play the measure. This functions checks the array containing the sequence on what segments it should play. This function uses `Sleep` to wait until the music is certainly started.

- `int playNext(int index);`
  Plays only one measure after the measure that is to be played before.

- `int setPrimary(IDirectMusicSegment8* pSegment[], int segmentCount);`
  Sets an array of primary segments.

- `int setEffect(IDirectMusicSegment8* pSegment);`
  Sets the effect segment.

- `int insertSecondary(IDirectMusicSegment8* pSegment[], int segmentCount);`
  Sets one row in the two dimensional secondary segment array. All segments that belong together (e.g. one instrument) are put into one row. These segments are never played simultaneously.

- `int configurePlayerFromFile(IDirectMusicLoader8* pLoader, ifstream *stream);`
  Given an inputstream and a pointer to the loader, this function reads the input and invokes the private function `loadFilesToSegments` to actually load the files. After loading the file this method calls one of the insert/set functions (primary, secondary, effect) to put the segments into one of the arrays. These insert/set functions are public because a programmer might not want to configure the player with a file, but do this 'by hand'.

- `bool isPlaying();`
  Checks if the nonlinear music is playing by invoking the `isPlaying` function of the performance. It checks every segment. If one or more segments are playing, this function returns `TRUE`.

- `int closeDown();`
  Release all the segments.

The private variables and functions will become clearer while reading the code. For your convenience a few of the private variables and functions are explained on the next page.

- `IDirectMusicSegmentState8* pSegmentState[2];`
  Concatenating the secondary segments is done by using these segment-states.
- `IDirectMusicSegment8* segmentPrimary[MAX_VARIATION_COUNT];`
  This array contains all primary segment pointers. Only one of these can play at a time. The number of primary segments is present in `segmentPrimaryCount`.
- `IDirectMusicSegment8* segmentSecondary[MAX_SECONDARY_INSTR][MAX_VARIATION_COUNT];`
  This 2 dimensional array contains all segment pointers belonging to different secondary segments. You can view a secondary segment as an instrument. There is a maximum number of instruments (`MAX_SECONDARY_INSTR`). There can be a maximum number of segments for one instrument. The variable `secondarySegmentCount` holds the number of instruments. The variable `segmentSecondaryCount[MAX_SECONDARY_INSTR]` holds the number of segments per instrument.
- `IDirectMusicSegment8* segmentEffect;`
  There is only one effect segment. Every time an effect has to played, this segment is used. Remember that with segment variations lots of sound-effects can be played!
- `int fillSequences(int measures);`
  This function fills two arrays. The array `primarySequence[MAX_PERF_LENGTH]` holds a number for every measure. This number defines what segment to play this measure. So, the number is an index for the primary segment array.
  The array `secondarySequence[MAX_SECONDARY_INSTR][MAX_PERF_LENGTH]` holds a number for every instrument in every measure. This also defines the segment to be played.
  When the `playNext` function is invoked these arrays are checked for which segment to play. Segments can be empty (created in DirectMusic Producer), so instruments can be silent.
  This function stipulates for a large part how the music sounds. For now, the segments are chosen more or less on a random basis. To create nice harmonics and give the composer the opportunity to configure the music more, attention should be paid to this method. Because the aim of this project is to learn to use DirectX Audio and not to use input streams and the random generator, this example program chooses the segments randomly.

Please read the code to understand the working of the functions. Also try to use the player with the two provided configuration files. The syntax of the files can be found in comments in the first file (*'exampleconfig1.txt'*).

6. Evaluation

The aim of this project was to explore DirectMusic and Nonlinear Music. Both topics are way too big to fully explore in one month (6 ECTS). Nonetheless I got an idea of the features of the DirectMusic API and I know more about Nonlinear Music. In addition I learnt C++, see section 6.1.

A programmer can do a lot with the DirectMusic API. It is very advanced and takes away a lot of work from the programmer. For instance, I did not know about the variations in segments, so I thought I had to 'shuffle' the samples myself. The existance of variations took away some work. Audiopaths, effects and 3D sounds can also take away a lot of work. I experienced 3D audio, but because of the limited time I did not include 3D audio in the final application.

The characteristics of sounds and the use of audiopaths can add a lot to non-linearity. A sample can sound different by changing audiopath parameters. By doing this 'different samples' can be created without real different samples. This saves disk space and adds to the number of possible changes in music.

Nonlinear Music is a complex subject. Designing sounds and composing music is very difficult if you know that it has to be played more or less at random. Therefore it should be possible for a composer to add some more rules to avoid bad harmonics. With this rules the composer might also bring a certain flow in his performance. In my application the whole performance is played at random, which is not bad for the radio disk jockey example. For real music however it would be nice if the composer/produces could at least define a similar beginning every time the performance is played. I did not create this functionality.

This project took me more time than the 6 ECTS stands for. The reason for this is that I spend more time in unforeseen tasks as designing sounds and working with DirectMusic Producer. I also had to learn the C++ programming language. Because of the time limit I did not include every feature I planned in advance.

6.1 Programming C++ with experience in other languages

In the bachelor Computer Science at the VU Amsterdam the main focus in programming languages is on JAVA. I've had a few courses on object oriented programming in JAVA. For a course of Systems Programming (SP) I had to use C. The assignment in SP was to build an audio streaming server and client. In addition to the programming skills learnt at the VU, I experienced PHP and JavaScript.

The first concern in using C++ is getting the software. The DirectX Audio Exposed book did not mention all the frameworks and platforms, so I had to find out what to install. It turned out that for running a simple C++ program that uses DirectX you need four platforms/frameworks. I had to download a lot of software (see section 6.3). It didn't work after the first installation. I uninstalled everything and installed the software again in another order. After that it worked.

The other thing that needs some attention when C++ is used for the first time is headers, linking and compilation. The first time I tried to compile a program (an example program) it failed. It took me some time decrypting the error messages and searching the internet for people with the same problems. Most of the errors came from directories and libraries that I didn't include.

Although I have programmed in C, there was a lot to learn about C++. In particular the way that C works with pointers is confusing. I learned best by carefully reading example code. I already have knowledge on object oriented programming, which makes C++ easier for me to understand than C.

With a book from the library I succeeded in producing programs. From a C++ programmers point of view my programs are far from perfect. I did not invest too much in this because the emphasis was on using the DirectX API.

7. References

7.1 Books

- Fay, T. M. and Fay, T. J. and Selfon, S.
  ”DirectX 9 Audio Exposed, Interactive Audio Development”
  Wordware Publishing, Inc. 2004.
- Savitch, W.
  “Absolute C++ second edition”
  Pearson Education. 2006.

7.2 Online Resources

- *MSDN Homepage*
  http://msdn.microsoft.com/
- *IAsig Archive*
  http://www.iasig.org/aan/NoOneLivesForever.shtml
- *GameDev.net*
  http://www.gamedev.net/reference/articles/article1689.asp

7.3 Software

- DirectX 9.0 SDK
- Microsoft .NET Framework SDK v2.0
- Microsoft Platform SDK
- Microsoft Visual C++ 2005 Express Edition
- Propellerhead Reason (to create samples)

- Programming examples provided with the book ”DirectX 9 Audio Exposed, Interactive Audio Development”.