Programmer's Quick Start Manual

First steps in game audio programming with

Contents

Introduction

About this manual

This manual will guide you through your first steps using ADX2 to program the audio system of a game. Because you will need some sounds to play, it will also quickly cover the data preparation and export from the authoring tool. However, please refer to the companion “Quick Start Manual” for the tool or its documentation to learn more about creating audio assets for ADX2.

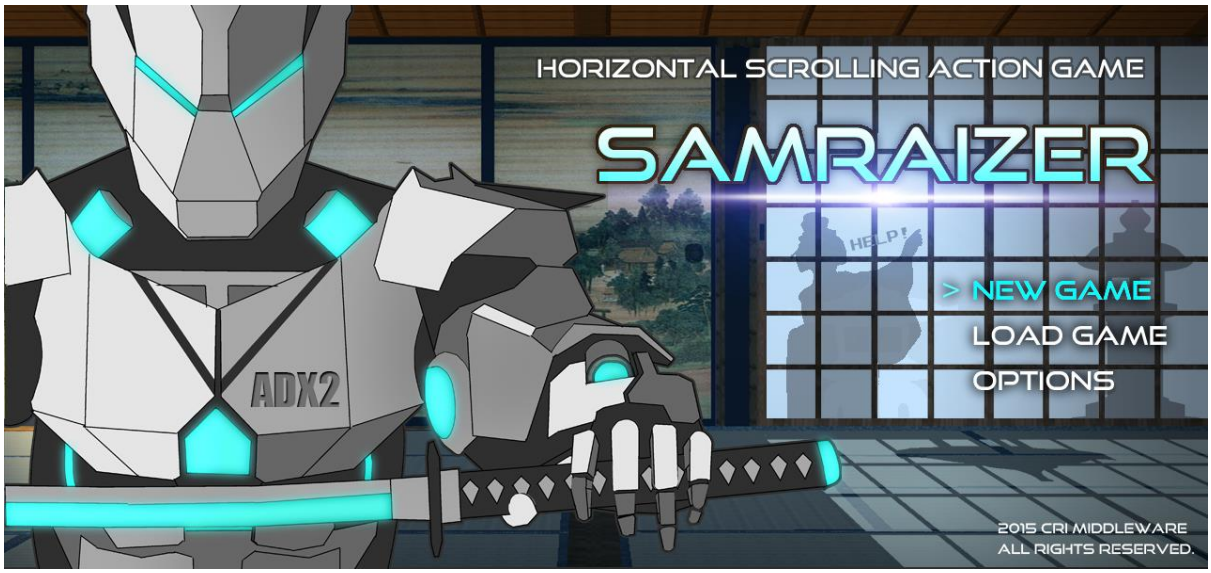
Through this Quick Start manual the following pictograms will be used to indicate sections of interest:

Key Term
A “key term” section will describe an important word from the ADX2 terminology.

Note
A “note” section will add some extra information about an ADX2 feature.

Tip
A “Tip” section will give an idea about how to use an ADX2 feature.

To illustrate how we program the various audio features needed in a game, we will take the example of a fictive game called Samraizer.



Samraizer is a samurai droid equipped with a super-sharp laser blade and a powerful ray gun. He is fighting the infamous NinjaBots across the galaxy. In this manual, we will attempt to build the soundtrack of his adventures by using ADX2!

About ADX2

ADX2 is the leading game audio middleware in Japan, used in more than 2800 games in all genres and on all platforms.

ADX2 offers a user-friendly, DAW-like, authoring tool and a full-featured audio engine including a high-performance proprietary codec.

ADX2 is available for consoles: PlayStation®4 , Xbox One®, Wii U ™, PSP®, PlayStation® Vita, Nintendo 3DS™ , PlayStation®3 , Windows® and for smartphones: iPhone / iPad and Android devices.

About CRI Middleware

CRI Middleware is a Japanese company headquartered in Tokyo, with offices in San Francisco. Founded in 2001, it specializes in audio and video software solutions for the digital entertainment industry.

Popular products include CRI Movie, a high-quality, multi-platform, movie playback system and CRI ADX2, an all-in-one audio solution.

What is a game audio middleware?

First, let's define what a game audio middleware is and what we can expect from it. A game audio middleware is composed of an authoring tool and a run-time component (game audio engine), in this case "CRI Atom Craft" and the "CRI Atom library".

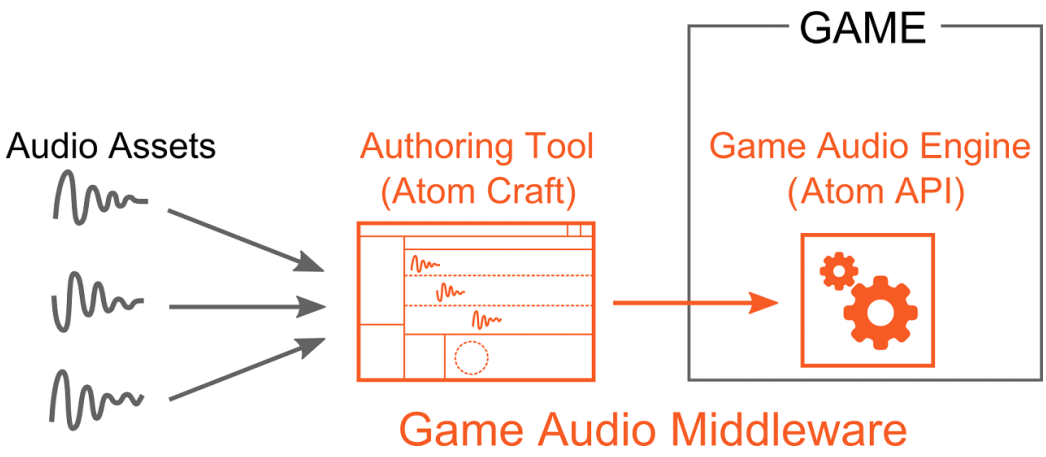
The authoring tool allows the sound designer to import audio assets (wave files), transform them into dynamic events that can be triggered and controlled, organize them into sound banks and export them to the game.

A game audio middleware provides specific features for sound effects (e.g. randomization and 3D positioning), interactive music (e.g. beat synchronization) and dialog (e.g. support for multiple languages).

A game audio middleware also offers mechanisms for the game and the audio engine to communicate (e.g. through game audio variables), so that the audio can truly be dynamic and react to what is happening in the game.

Last but not least, a game audio middleware usually provides game audio previewing from within the authoring tool, lowering the time required to debug and iterate on the sound design.

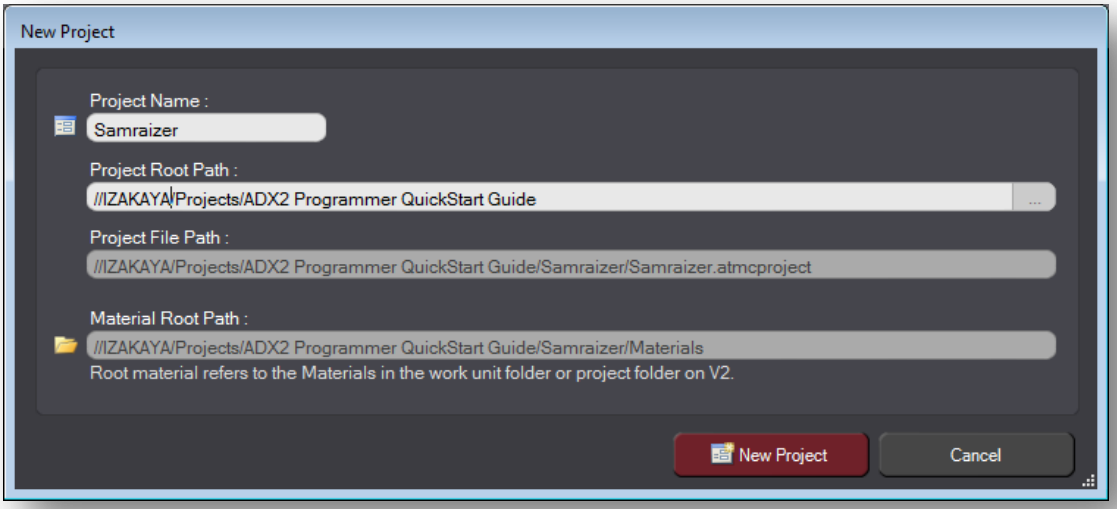
All this – and much more – can be done with Atom Craft and the Atom library, in a very easy and intuitive way. ADX2 truly empowers sound designers while offering rock-solid run-time performance.



Presentation of the ADX2 audio pipeline

Creating a project

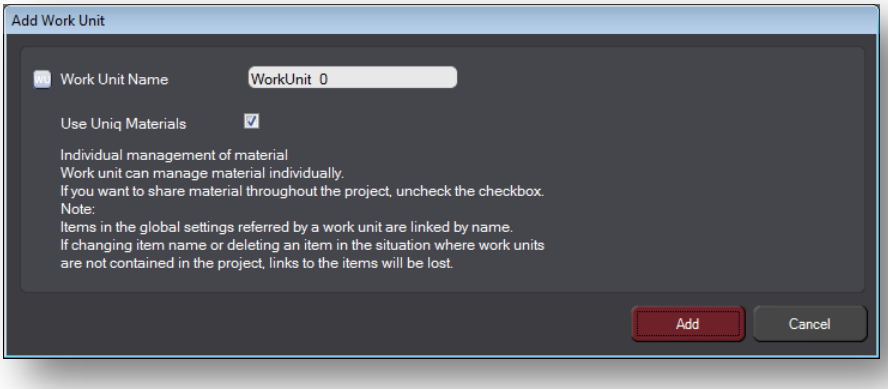
In order to test the Atom library, we will need to create some audio assets. Fortunately, this can be done very easily, and quickly. Let's start by creating a new project. Start Atom Craft, the ADX2 authoring tool. In the "File" menu, select "New Project…". The following window will appear:



Choose the name of your project (here "Samraizer") and where you want to create it (i.e. in what folder). From this information, Atom Craft will automatically deduce the path to your Atom Craft project file (it will have an .atmcproject extension) and the folder where the materials used by your project will be copied.

> Key Term
> *Materials* are audio assets - i.e. sound files - that your project will use. The *Material Root Path* is simply the parent folder in which all of these audio assets will be contained.

Press the "'New Project" button and the next window appears:



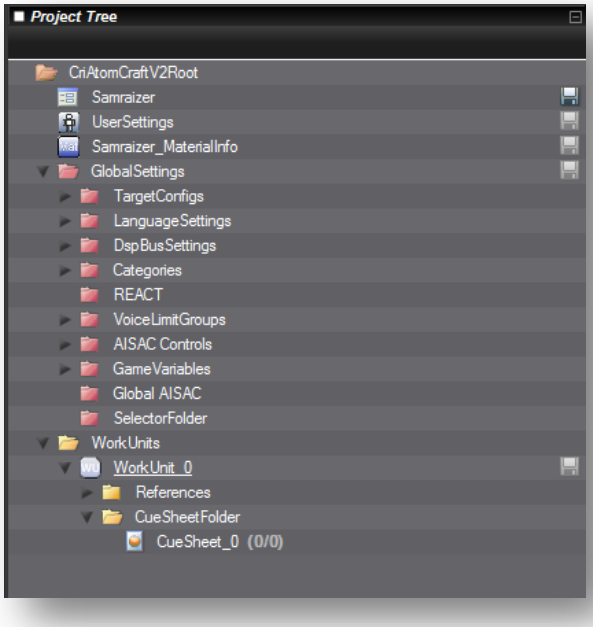
This window lets you create the first work unit in your project.

Key Term

Work Units are components of your project that can be checked-in / checked-out independently. This is a very convenient system that allows several sound designers to work on the same project simultaneously. For example a sound designer could be in charge of the weapon sound effects, another one of the dialogue etc…

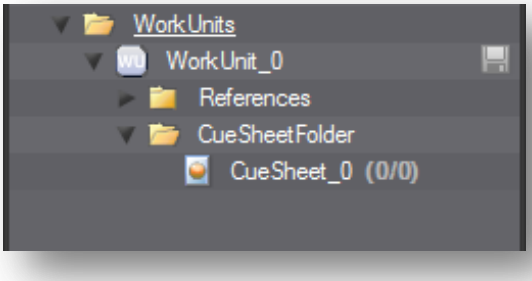
The window also allows you to select if the materials used by this work unit should be local or can be used globally (and therefore put in the root material folder). Let's keep all default settings and press the "Add" button.

Your project is created, and the first work unit with it. In this Quick Start document, we will only use this work unit as it is a small project.



Adding sounds

The project tree already displays many folders and items, but we don't need to worry about most of them at this point. What is of interest to us is located under the Work Units folder. In the newly created work unit, Atom Craft has already added a cue sheet folder and a cue sheet.



Key Term

A *Cue* is a sound object that the game can trigger. A set of cues is called a *Cue Sheet*. It is possible to organize cues and cue sheets even more with folders. A cue sheet in ADX2 corresponds to a sound bank in other systems.

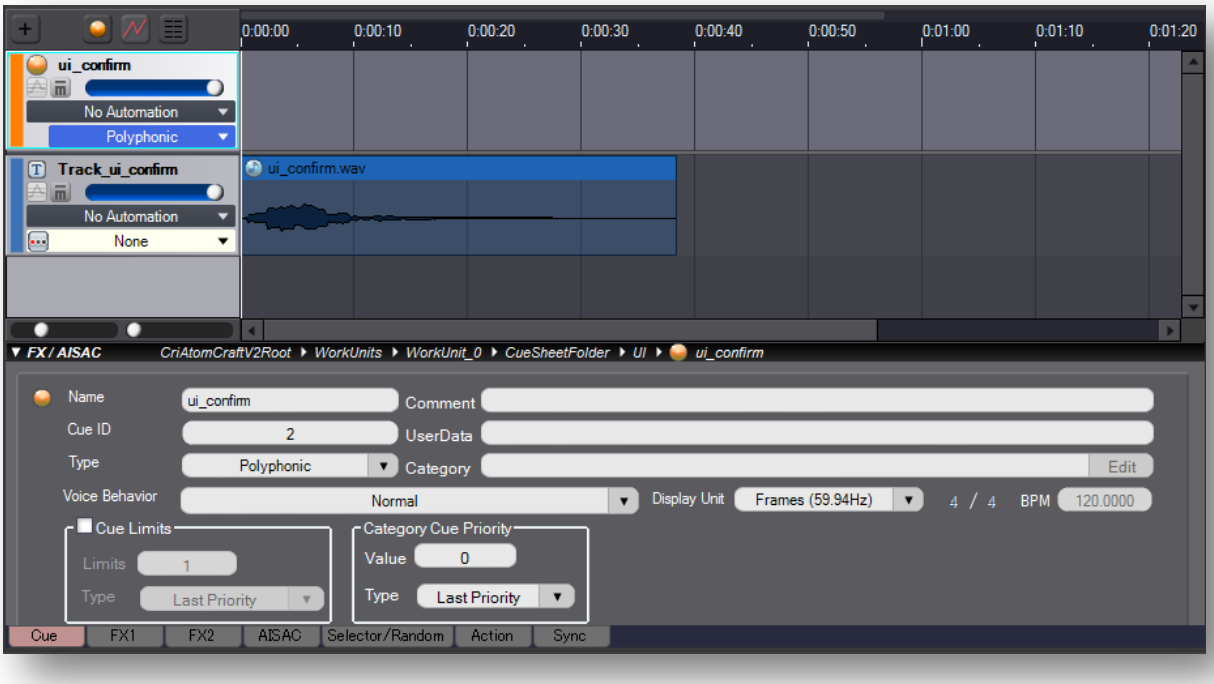
Let's rename our cue sheet "UI": it will be the sound bank where we put all the interface sounds of the game. It's time to add our first sounds! For now, we will just add three: a hover sound, a confirm sound, and a cancel sound.

There are several ways to add materials and cues. Here we will use the fastest one, which consists in dragging our wave files from the Windows Explorer and dropping them directly onto the cue sheet. This will automatically add the samples to the material folder, create a cue per sample and add it on the track.

Key Term

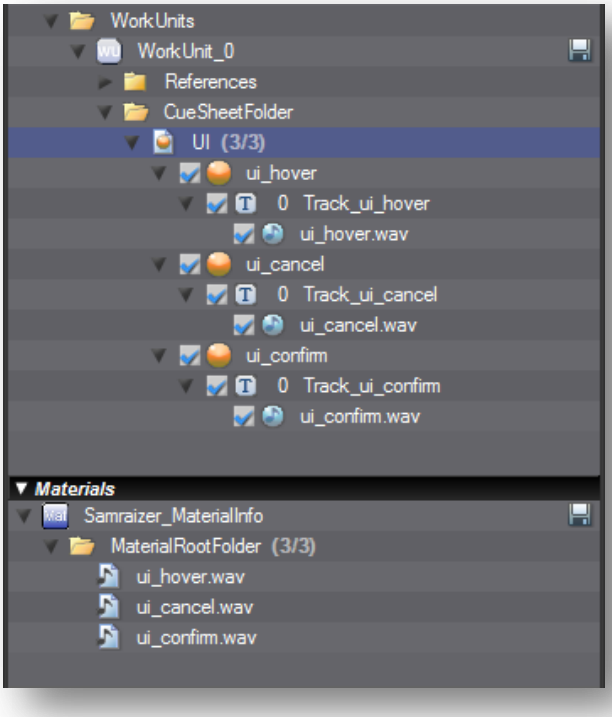
A *Track* in Atom Craft is similar to a track in your DAW. It is a timeline on which you can place one or more waveforms. A cue can have one or more tracks.

If we click on one of our cues, it should now look like this:



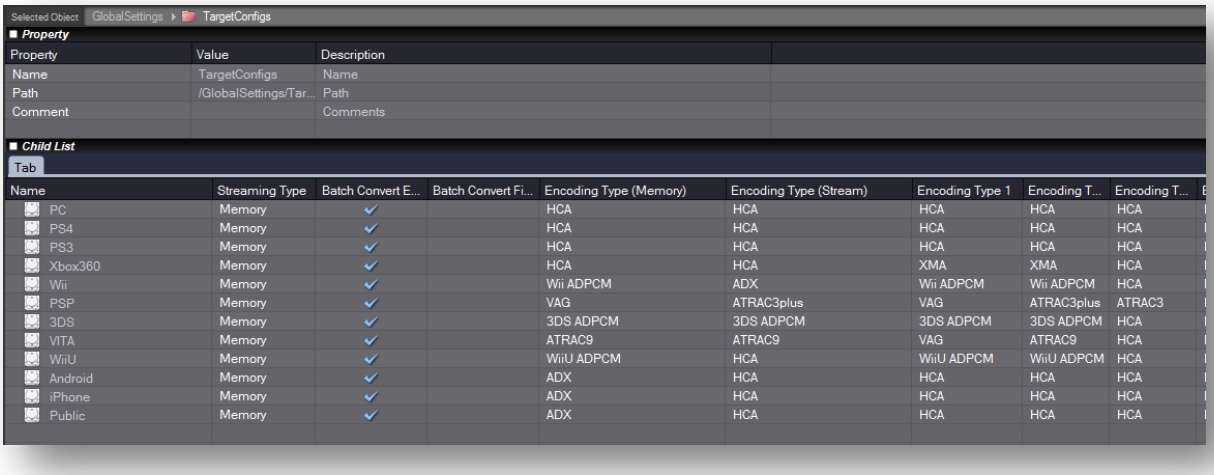
We can see the cue, with a unique track, hosting a single waveform. This is all we need for now, but keep in mind that cues can have many tracks, and each track can play a sequence of waveforms. The cue type determines its behavior at run-time. For example a Polyphonic cue - this is the default, like on the picture above - will play all its tracks simultaneously, while a Random cue will play a different track each time.

Finally, in the project tree, we can verify that the cues and tracks have indeed been added to the project, and that the tracks are referencing wave files which have been copied into the Materials root folder.

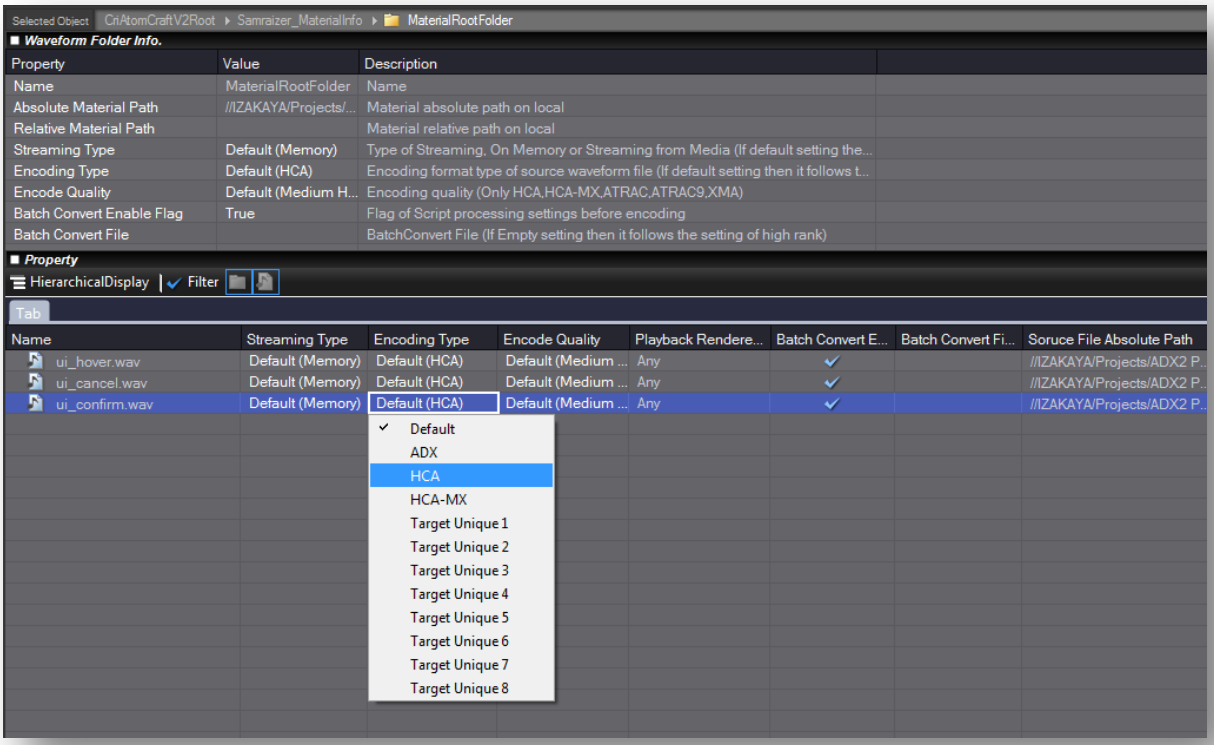


Encoding

We now have our three user interface sounds in a cue sheet. When we will export them to the game, the corresponding wave files will first be encoded to a format suitable to the target platform. We can check and modify the target settings by clicking on *TargetConfigs* in the project tree:



We can also override the encoding type for each waveform in the material folder:



The encoding formats listed here probably need a bit of an explanation:

ADX is a high-fidelity, high compression, low CPU-load codec running on all game platforms, mobile devices, and embedded systems. With ADX, 48 kHz stereo sounds can be compressed up to 1/16.

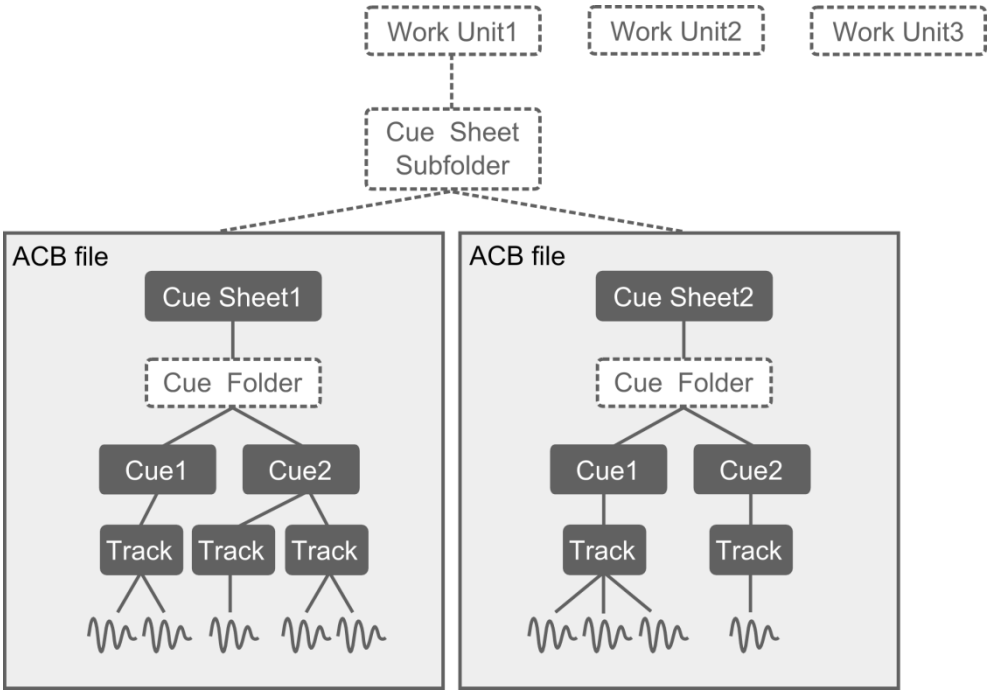
The HCA codec (for High Compression Audio) offers a compression ratio from 1/6 to 1/12. This codec has been tuned for games, the CPU load is very low (compared to MP3 and AAC) and constant during decoding. HCA also supports seeking.

The HCA-MX codec is a variation of the HCA which reduces the CPU load when playing back a large number of sounds simultaneously. Thanks to this, the engine can play hundreds of sounds on consoles, and tens of sounds on mobile platforms.

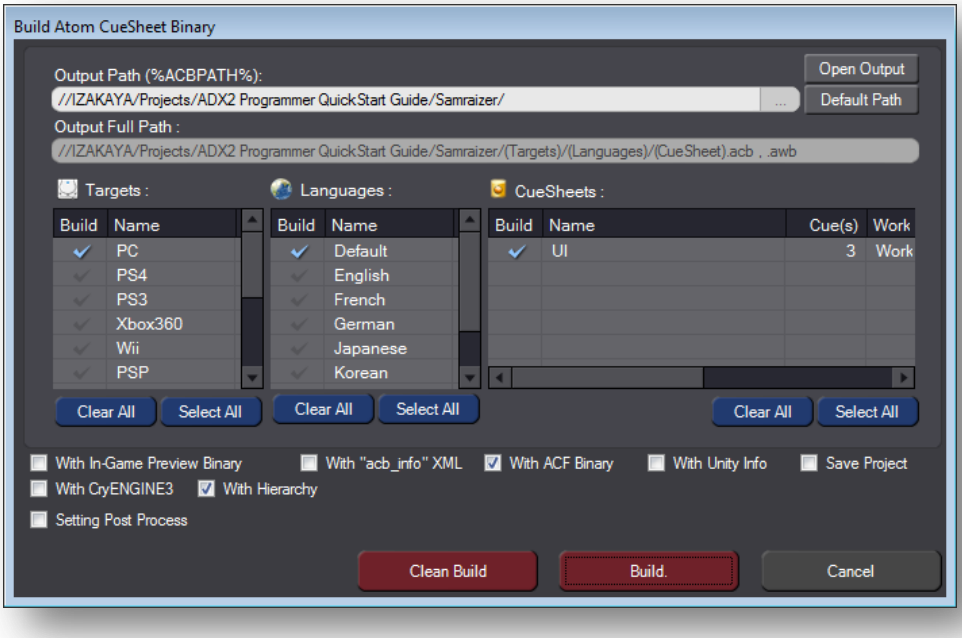
Exporting

Everything is now ready; it's time to export our data to the game! The hierarchy of objects and folders in the project tree can be a bit intimidating but a lot of them are here for convenience. For example, work units have been introduced to enable several sound designers to work on the same project while cue sheet sub-folders and cue folders offer a convenient way to organize your data in the tool.

Therefore, the run-time does not really need them. The following graph shows what part of a work unit is actually exported and used by the Atom library. In dotted lines are the objects present for convenience but which are not packaged for the run-time.

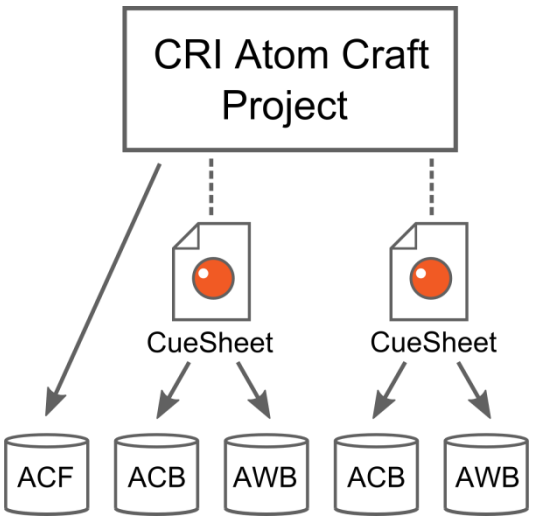


The export itself mainly consists in generating a binary representation of the cue sheets, as well as the general information about the project. To trigger the export, call the *Build Atom CueSheet Binary* command from the *Build* menu or from the toolbar. The following window will appear:



You will be able to select the target platform(s), the target language(s), and of course what cue sheets you want to export.

When Atom Craft exports a project, several files are generated: one ACF file for the whole project, and one ACB and one AWB file (if needed) per cue sheet exported. In addition to these files, C/C++ headers will be generated. Other files specific to game middleware (for example for Unity or Cry Engine) may also be exported if needed, as well as XML files that can be processed by third party tools.



The ACF file (for *Atom Configuration File*) contains the general information about the project and what cue sheets are in it. There is only one ACF file exported per project.

An ACB file (for *Atom Cue sheet Binary*) contains all the parameters of a given cue sheet as well as any sample data destined to be played in memory by that cue sheet. If the audio data is expected to be streamed, it is stored in the corresponding AWB file.

An AWB file (for *Atom Wave Bank*) contains the encoded data for the streams referenced by the cue sheet (the parameters for the stream playback are in the ACB file).

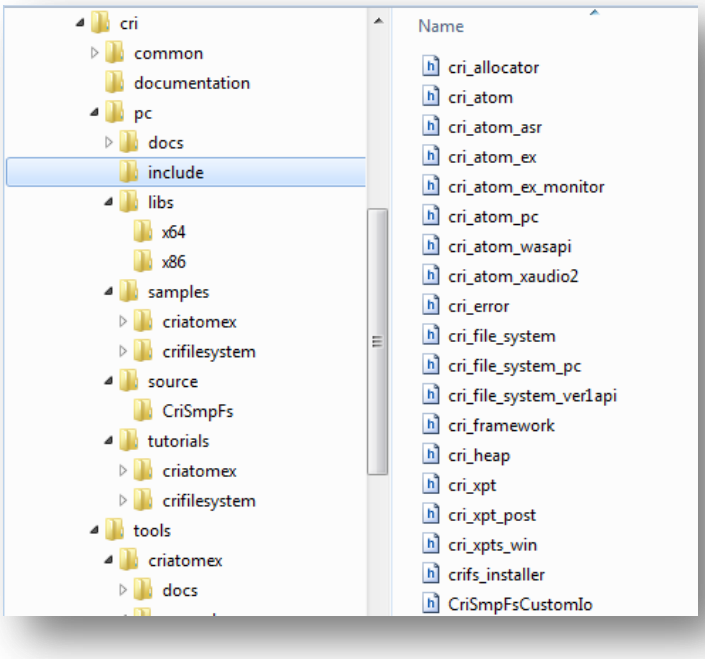
In our case, we will simply select PC as a target and keep the default language. For the moment, we only have the UI cue sheet to export. Press “Build”, and the test data is ready in less than one second! Atom Craft displays the details of the operation in its log window at the bottom of the screen. Now it’s time to get serious and launch the IDE!

```
10:50:09 :  Building Atom CueSheet Binary ... [PC Default] "UI".
10:50:09 :  ACB built using the waveform language setting of[Default].Encoded Done.  [0] (8,621 bytes) "ui_cancel_Enc_44100HCAHigh_a7d8632fd5600cbfa508bca843b90c70.hca"
Encoded Done.  [1] (9,644 bytes) "ui_confirm_Enc_44100HCAHigh_1011a3b527466f4151b482237d00612b.hca"
Encoded Done.  [2] (7,257 bytes) "ui_hover_Enc_44100HCAHigh_2de2cf193209023d900b5c9ff7ac1405.hca"

10:50:09 :  Make Awb Directory "C:/Samraizer/Samraizer_Cache/PreviewCache/PC"Done. ("UI.awbTmp" AWB Size 25,581 bytes)
10:50:09 :  Building ACB file [PC Default] "C:/Samraizer/PC/UI.acb" ... Done. (ACB Size 28,928 bytes)
10:50:09 :  Completed the build of Atom CueSheet Binary. [PC Default] "UI"
10:50:09 :  Complete Building Atom CueSheet Binary.1/1 Build Time"0:0:0.603"
```

Setting up the C/C++ project

Including headers

Like most middleware, ADX2 redefines basic types such as int and float, so that there is no confusion, especially about their size. This is done in the `cri_xpt.h` header file, which can be found in the \cri\pc\include subfolder of the SDK root folder. Therefore this file must be included first. It is common to all CRI technologies.



The other files you will need to include (and which are located in the same folder) are:

`cri_atom_ex.h`, which corresponds to the API declarations for the CRI Atom library
`cri_atom_pc.h`, which includes the PC-specific API declarations

If you are using Visual Studio, select your project, open its properties, and add the path to CRI's include folder to "*Configuration Properties -> C/C++ -> General -> Additional Include Directories*" After that, it should look like this in your code:

```
/* CRI SDK header file */
#include <cri_xpt.h>

/* CRI ADX2 headers files */
#include <cri_atom_ex.h>
#include <cri_atom_pc.h>
```

Of course, you will also usually want to include the header files generated during the export of your audio data by the Atom Craft tool. For example, when we exported our nascent Samraizer project earlier, two .h files were created in the same folder.

The first one, `Samraizer_acf.h`, corresponds to the ACF file and includes information about project-wide settings, such as sound categories, DSP Settings etc… Unsurprisingly, there is not much in that file yet.

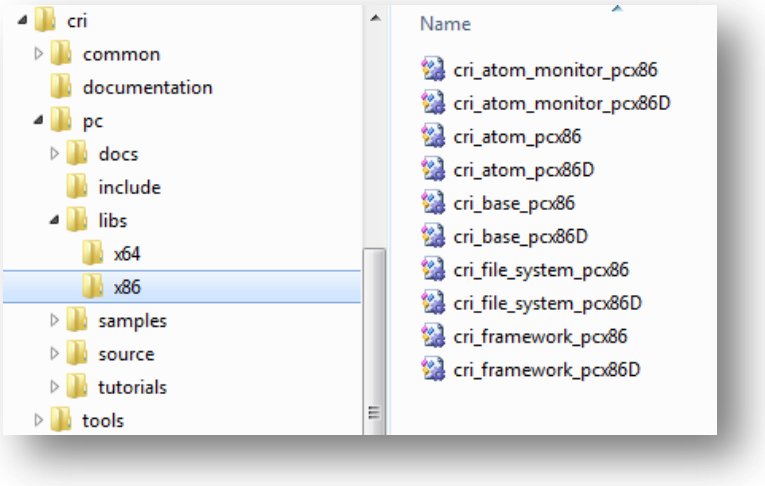
The second, `UI.h`, corresponds to the ACB file generated for the UI cue sheet and contains the IDs of the cues. Later, when we are using AISAC to control cues from game variables, it will also contain the AISAC control IDs. Here is how this header looks like for the moment. You can recognize the definitions for our three UI sounds:

```
/*===========================================================================*
 *  Header file for Atom CueSheet Binary
 *  Project          : Samraizer
 *  Tool Ver.        : CRI Atom Craft Ver.2.07.05
 *  File Path        : C:/Samraizer/PC/UI.acb
 *  File Name        : UI.acb
 *  File Size        : 28,928 bytes
 *  Date Time        : 28/03/2015 10:50
 *  Target           : PC
 *  Cues             : 3
 *  CueSheet Comment :
 *===========================================================================*/

#define CRI_UI_CUENUM (3)

/* AISAC Control List (AISAC Control ID) */
// No AISAC Control

/* Cue List (Cue ID) */
#define CRI_UI_UI_CONFIRM  ( 0) /*   */
#define CRI_UI_UI_HOVER    ( 1) /*   */
#define CRI_UI_UI_CANCEL   ( 2) /*   */

/* Block List (Block Index) */

/* end of file */
```

Adding libraries

While we are setting up our project, we also need to tell the linker where to find the CRI libraries.

Again, if you are using Visual Studio, select your project, open its properties, and add the path to the CRI's library folder to "*Configuration Properties -> Linker -> General -> Additional Library Directories*". Add the folder which corresponds to your target, for example x86:



Then, in "*Configuration Properties -> Linker -> Input -> Additional Dependencies*", add the libraries. For example, if you are building the debug version targeting x86 on PC, you should add the following libraries:

```
cri_atom_pcx86D.lib
cri_base_pcx86D.lib
cri_file_system_pcx86D.lib
cri_framework_pcx86D.lib
```

> Note
> You can also add `cri_atom_monitor_pcx86D.lib` if you are planning to use the in-game preview features of ADX2.

ADX2 engine flow

Preparation

Before even thinking about initializing the audio, there are a couple of things we should be doing. First, we should register the error callback function. This will allow us to get a notification when an error occurs, even if it as early as during the initialization of ADX2.

This is done simply with:

```
criErr_SetCallback(samraizer_error_callback_func);
```

Where the signature of the callback is as follows:

```
static void samraizer_error_callback_func(const CriChar8 *errid, CriUint32 p1,
CriUint32 p2, CriUint32 *parray);
```

As you can see, you get the error ID and a couple of parameters. You can use the `criErr_ConvertIdToMessage` function to generate the corresponding error message based on the ID and two parameters. You can then display this information in the console window or in the in-game debug menu if you have one.

The second important step before the initialization itself is to define the memory allocator our game will use for audio. This is done by calling `criAtomEx_SetUserAllocator`, which is defined as:

```
criAtomEx_SetUserAllocator(p_malloc_func, p_free_func, p_obj)
```

It takes a couple of functions for allocating and freeing memory as parameters, as well as an optional user object. In their simplest form, these functions could just call the standard malloc() and free() functions like this:

```
void *samraizer_alloc(void *obj, CriUint32 size)
{
	void *ptr;
	UNUSED(obj);
	ptr = malloc(size);
	return ptr;
}

void samraizer_free(void *obj, void *ptr)
{
	UNUSED(obj);
	free(ptr);
}
```

However, in a real game, you would usually not do that in order to avoid memory fragmentation and you would rather define memory pools from which you can return pointers on already allocated memory.

Initializing and terminating

With the error notification and the memory allocator taken care of, we are now ready to initialize the Atom library. This can be done very easily by calling:

```
criAtomEx_Initialize_PC(NULL, NULL, 0);
```

You can see that this function is specific to the PC platform - on which Samraizer is supposed to run – and takes 3 parameters, which we all kept at NULL or 0.

Note
There are many initialization or object creation functions in the Atom library. They usually accept configuration parameters. When you pass NULL for these parameters, default values will be used.

These configuration parameters are quite important though and although a detailed explanation goes far beyond the scope of this Quick Start manual, it is a good idea to familiarize yourself with them to optimize your game.

`criAtomEx_Initialize_PC` is a convenience function that internally calls a couple of other initialization functions so that you don't have to deal with all the details. In reality, the Atom library is composed of a series of modules responsible for very specific tasks, like loading ACB files, managing voices or playing sound back. We will encounter some of them later.

Tip
It is useful to note that the names of the functions you will call are based on the following naming scheme:

cri + Atom or AtomEx + module name + "_" + name of the function
The name of a function itself is of the form verb + object.

For example, here is a typical function name that we will see later: `criAtomExPlayer_SetCueId`

This naming convention makes it very easy to both find the right function when you know what you want to do and to read code calling the Atom library functions.

But let's come back to the engine flow: terminating is straightforward and can be done by calling the following function:

```
criAtomEx_Finalize_PC();
```

Again, this is specific to the PC platform and calls several other finalizing functions behind the scenes.

Loading and unloading project data

Once ADX2 initialized, the first thing we will have to do is register the ACF file which corresponds to our project, in our case `Samraizer.acf`. The AtomEx module provides us with a function for that and we can just call:

```
criAtomEx_RegisterAcfFile(NULL, "C:\\Samraizer\\PC\\Samraizer.acf", NULL, 0);
```

As mentioned earlier, the ACF file contains global information that can be referred to by the ACB files, such as the DSP settings, sound categories and more… It only needs to be registered once and can be unregistered when exiting the game (before the call to `criAtomEx_Finalize_PC`) with

```
criAtomEx_UnregisterAcf();
```

With the ACF file registered, it is now possible to load ACB files, i.e. cue sheets.

> Key Term
> `criAtomExAcb` is the module responsible for accessing the data stored in ACB (Atom CueSheet Binary) files. This data contains the cue parameters and - for the cues played from memory - the waveform data as well.

You can use `criAtomExAcb_LoadAcbFile` to load an ACB file. For example, let's load our UI cue sheet at the beginning of the program, as it will always have to be present:

```
CriAtomExAcbHn acbUIHn = criAtomExAcb_LoadAcbFile(NULL, "C:\\Samraizer\\PC\\UI.acb",
 NULL, NULL, NULL, 0);
```

> Note
> You can of course load as many cue sheets as you want (as long as you have enough memory).
> If some cues are streamed, then the path to a corresponding AWB file must also be passed when loading the ACB file.

When you load an ACB file successfully, you will receive an ACB handle (of type `CriAtomExAcbHn`, like `acbUIHn` above) which will allow you to reference that cue sheet later. When you don't need the cue sheet anymore, you can use this handle to release it as follows:

```
criAtomExAcb_Release(acbUIHn);
```

Typically, you will load banks at the beginning of a level or section of the game and unload them at the end. Some banks may be always loaded, for example the interface sounds as you can usually pause the game and go in the menus at any moment.

> Tip
> Smart separation of cues in banks allows you to save memory. For instance, we could have a big bank with all our sound data for each level of Samraizer, but it is often preferable to have one bank per type of sounds, one for Samraizer, one bank for the NinjaBots, one for the ambiences of level 2 etc… so that we can load only what is needed.

Updating ADX2

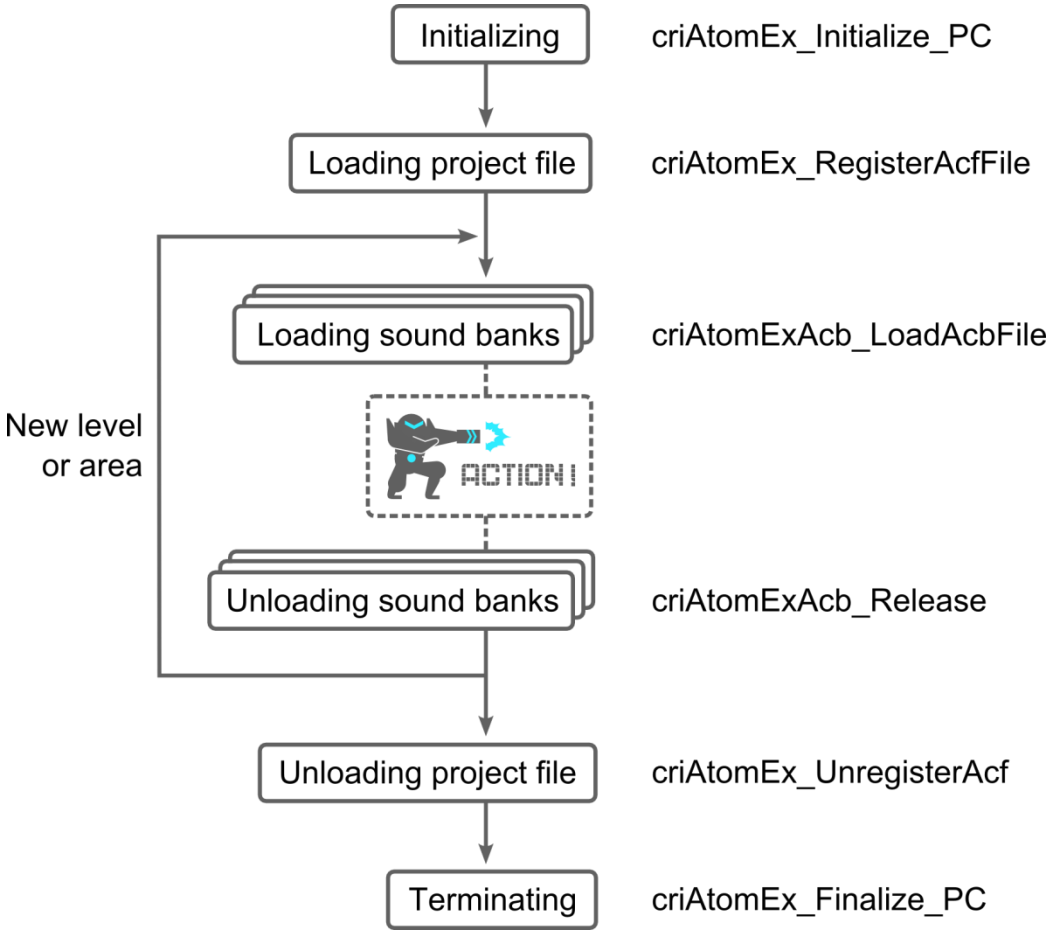
Like any other middleware running concurrently with the game, you must make sure that the CRI Atom library's core function is executed at regular intervals, as it is responsible for decoding the audio data, actually starting the voices, updating all the internal parameters etc… This is done by calling:

```
criAtomEx_ExecuteMain()
```

It will typically be called from the game loop, which is executed for each frame. If this function is not called frequently enough, audio playback may not be updated in real-time, which will cause symptoms such as stuttering.

> Note
> Although we are not going into the details in this Quick Start guide, please note that the frequency at which this function must be called depends on the settings passed when initializing the library. The behavior is also different based on the threading model chosen.

Let's pause for a bit and review what we know. Here is how the initialization / termination and loading / unloading happens in the game:



In code, the initialization up to loading our first cue sheet would look like this:

```
CriAtomExAcbHn acbUIHn;

/* Registration of error callback function */
criErr_SetCallback(samraizer_error_callback_func);

/* Registration of memory allocator */
criAtomEx_SetUserAllocator(samraizer_alloc, samraizer_free, NULL);

/* Library initialization */
criAtomEx_Initialize_PC(NULL, NULL, 0);

/* Load and register an ACF file */
result =  criAtomEx_RegisterAcfFile(NULL,"C:\\Samraizer\\PC\\Samraizer.acf",NULL,0);

/* Load an ACB file and create an ACB handle */
acbUIHn = criAtomExAcb_LoadAcbFile(NULL,"C:\\Samraizer\\PC\\UI.acb",NULL,NULL,NULL,0);
```

And the termination would look like this:

```
/* Destroy ACB handle */
criAtomExAcb_Release(acbUIHn);

/* Unregister ACF */
criAtomEx_UnregisterAcf();

/* Finalize the library */
criAtomEx_Finalize_PC();
```

Simple enough, isn't it? Now it is time for some action and to see how we can actually trigger some sounds!

Basic playback

Voice Pools

Playback of sounds is assured by the combined work of two types of objects: the voice pool and the AtomEx player. First, let's talk about the voice pool and voices. A voice is the most basic object that can be used to play back sound from memory or to stream sound files.

> Key Term
> `CriAtomExVoicePool` is the module responsible for managing voices. It deals both with instance limiting based on priorities and with the decoding of the audio data. Therefore, a "voice pool" of the right type must be created before playing back cues with the AtomEx player.

You can create a standard voice pool (which supports the playback of both ADX and HCA data) with the function `criAtomExVoicePool_AllocateStandardVoicePool`.It will return a voice pool handler (of type `CriAtomExVoicePoolHn`) that you can use for subsequent operations or to release the voice pool with the `criAtomExVoicePool_Free` function.

There are also functions to create voice pools specific to data with a given type of encoding. For example, `criAtomExVoicePool_AllocateAdxVoicePool` can be used to create a voice pool for ADX playback only, while `criAtomExVoicePool_AllocateHcaMxVoicePool` can be used to create a voice pool for HCA-MX playback only.

A special note about HCA-MX data: it cannot be played back at a rate other than the sampling rate specified when the data was created. That is because the mixing is performed before the decoding. When the HCA-MX module is initialized - or when a voice pool is created - you have to make sure to specify a sampling rate that is the same as that of the audio data.

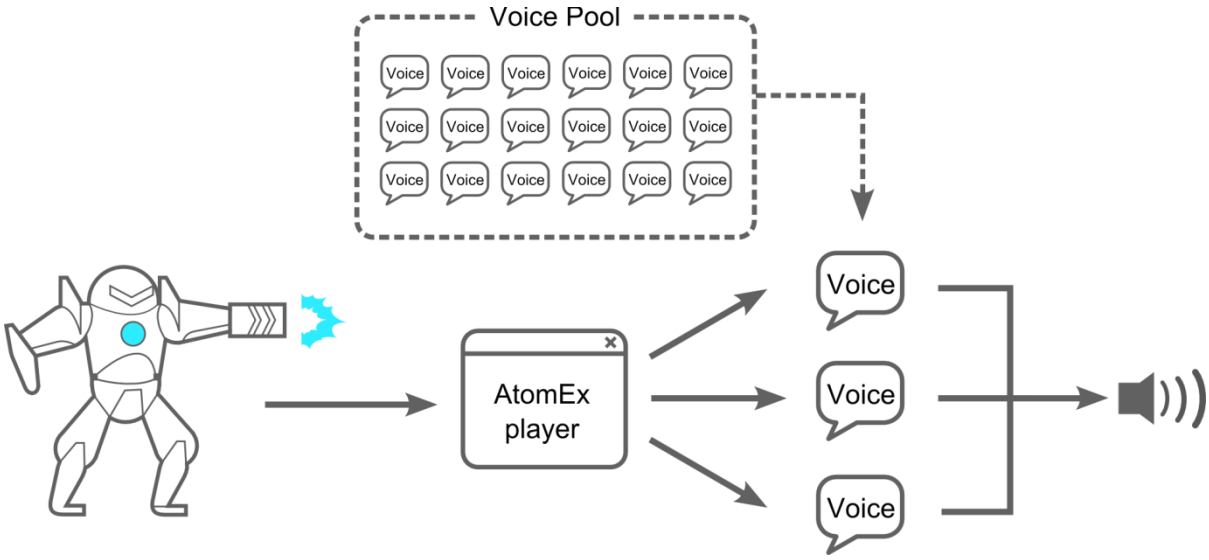
> Key Term
> CriAtomHcaMix is the module responsible for playing back HCA-MX data. It mixes all HCA-MX data and outputs it, while ensuring a low CPU load.

Finally, please note that creating a voice pool is a blocking function and therefore you should do it at initialization time or when switching between levels for example.

The AtomEx player

> Key Term
> CriAtomExPlayer is the module responsible for the playback of the audio data. Playback is assured very simply by creating an AtomEx player, setting the audio data you want to play, and starting the playback.

The graphic below shows the relationship between the voice pool and the AtomEx player for the playback of the cues.



When Samraizer shots his gun, the AtomEx player will require a voice from the voice pool to play the sound. If Samraizer continues to shoot and the first sound has not finished playing, the AtomEx player will require another voice. The voice pool decides when the AtomEx player can get a new voice (or steal an existing one) based on priorities and voice limit groups. This is called dynamic voice assignment. As we will see later, on its side, the AtomEx player is responsible for the updating of the playback parameters such a volume, pitch etc…

An AtomEx player can be created by calling the function:

```
CriAtomExPlayerHn player = criAtomExPlayer_Create(NULL, NULL, 0);
```

This function returns a handle on the player which can be used later to update the playback parameters, or when freeing the AtomEx player if it is not needed anymore:

```
criAtomExPlayer_Destroy(player);
```

> Tip
> It is always better to limit the amount of creation / destruction of objects and to keep a couple of players around during the game. This is even more important in this case as the `criAtomExPlayer_Create` function is blocking.

Now that the AtomEx player is created, all we need to do is associate the cue we want to play with it. There are several ways we can specify the cue to play: by ID, by name, and by index. This is done respectively by the following functions:

```
void CRIAPI criAtomExPlayer_SetCueId(CriAtomExPlayerHn player, CriAtomExAcbHn acb_hn,
CriAtomExCueId id);

void CRIAPI criAtomExPlayer_SetCueName(CriAtomExPlayerHn player, CriAtomExAcbHn acb_hn,
const CriChar8 *cue_name);

void CRIAPI criAtomExPlayer_SetCueIndex(CriAtomExPlayerHn player, CriAtomExAcbHn
acb_hn, CriAtomExCueIndex index);
```

For example, if we wanted to assign the confirm button cue of our previously loaded UI bank to the player, we could write the following code:

```
criAtomExPlayer_SetCueId(player, acbUIHn, CRI_UI_UI_CONFIRM);
```

> Tip
> Even if you stop the sound, the cue information is kept, which means that you can repeatedly play that cue by stating the AtomEx player without setting it again. You only need to set the cue again if you want to play a different cue.

Please note that the AtomEx player can also play an audio file (such as a .hca) directly with the function `criAtomExPlayer_SetFile` or a waveform from an AWB file with `criAtomExPlayer_SetWaveId`. In both cases, the format, number of channels and sample rate can be defined by the following functions:

```
void criAtomExPlayer_SetFormat(CriAtomExPlayerHn player, CriAtomExFormat format);
void criAtomExPlayer_SetNumChannels(CriAtomExPlayerHn player, CriSint32 numChannels);
void criAtomExPlayer_SetSamplingRate(CriAtomExPlayerHn player, CriSint32 srate);
```

Playing and stopping a sound

Now that the player is created and a cue is assigned to it, all we need to do is to start the playback:

```
criAtomExPlayer_Start(player);
```

Pausing and resuming the sound is equally straightforward.

```
criAtomExPlayer_Pause(player, CRI_TRUE);        /* Pause */
criAtomExPlayer_Pause (player, CRI_FALSE);      /* Resume */
```

Stopping it is also a one-line affair:

```
criAtomExPlayer_Stop(player);
```

This is the most “natural” way to stop a sound as it will let the release envelope of the cue finish. However, if the sound must be stopped immediately, then you can call:

```
criAtomExPlayer_StopWithoutReleaseTime(player);
```

Getting the playback status

There are some cases where you will want to know if the sound has actually finished playing, for example to start a second one, or to release the AtomEx player. The player offers a method to get the playback status:

```
CriAtomExPlayerStatus criAtomExPlayer_GetStatus (CriAtomExPlayerHn player)
```

The `CriAtomExPlayerStatus` enum can have 5 values:

```
CRIATOMEXPLAYER_STATUS_STOP
CRIATOMEXPLAYER_STATUS_PREP
CRIATOMEXPLAYER_STATUS_PLAYING
CRIATOMEXPLAYER_STATUS_PLAYEND
CRIATOMEXPLAYER_STATUS_ERROR
```

For instance, if we wanted to do something when the playback of a cue has finished, we could do this at periodic intervals:

```
CriAtomExPlayerStatus explayer_status;

/* Get AtomEx player status */
explayer_status = criAtomExPlayer_GetStatus(player);

/* If playback of the cue has finished */
if (explayer_status == CRIATOMEXPLAYER_STATUS_PLAYEND){
       /* Do something special here */
}
```

In other cases, we may want to know how long a cue has been playing in order to synchronize it with another one or to trigger a non-audio game action (for example making Samraizer move his head when he is talking).

The `criAtomExPlayer_GetTime` function can be used for that purpose:

```
CriSint64 criAtomExPlayer_GetTime(CriAtomExPlayerHn player)
```

It returns the time elapsed since the start of the last playback, in milliseconds.

Let's review now what we have learned in this chapter through a small bit of code. Imagine we already initialized the Atom library and loaded the right ACF and ACB files, like we did at the end of the previous chapter. The next page shows how easy at that point it is to play a cue and to wait for it to finish.

First we create the voice pool, then the AtomEx player. We assign the correct cue to the player, and start the playback. We then periodically check the status of the player to know if the playback has ended, at which point we destroy the player and release the voice pool. That's all!

```
CriAtomExPlayerHn player;
CriAtomExVoicePoolHn voicePool;
CriAtomExAcbHn acbUIHn;

/* Initializing and loading of ACF and ACB here */
/* … */

/* Create a voice pool */
voicePool = criAtomExVoicePool_AllocateStandardVoicePool(NULL, NULL, 0);

/* Create a player */
player = criAtomExPlayer_Create(NULL, NULL, 0);

/* Specify a Cue ID */
criAtomExPlayer_SetCueId(player, acbUIHn, CRI_UI_UI_CONFIRM);

/* Start playback */
criAtomExPlayer_Start(player);

for(;;) {
	CriAtomExPlayerStatus explayer_status;
	samraizer_sleep(10);

	/* Execute the server process */
	criAtomEx_ExecuteMain();

	/* Get AtomEx player status */
	explayer_status = criAtomExPlayer_GetStatus(player);

	 /* Exit the playback loop if the status is PLAYEND */
	if (explayer_status == CRIATOMEXPLAYER_STATUS_PLAYEND) {
		break;
	}
}

/* Destroy player handle */
criAtomExPlayer_Destroy(player);

/* Destroy voice pool */
criAtomExVoicePool_Free(voicePool);

/* Unloading of ACF and ACB and finalizing here */
/* … */
```

Note

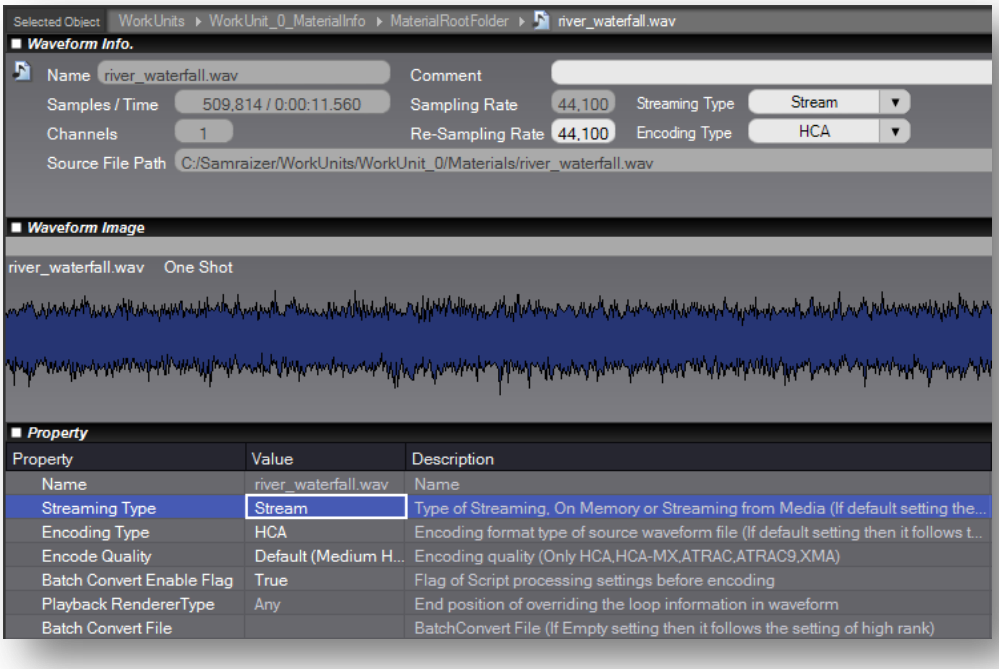
In this example, we played a cue which consists of a single sample, played each time. In a game, this could indeed correspond to a UI sound. However, in many other cases, you will want to have a more complex behavior. For instance, you may want to play sounds picked up randomly from a set of footsteps, to avoid repetitiveness when Samraizer walks. This can be done by the sound designer within the Atom Craft tool itself, without the programmer having to code all the logic. Please refer to the “ADX2 Quick Start Manual” to learn more about all the types of cues available, such as polyphonic, shuffle, switch, random, sequential, etc...

Streaming

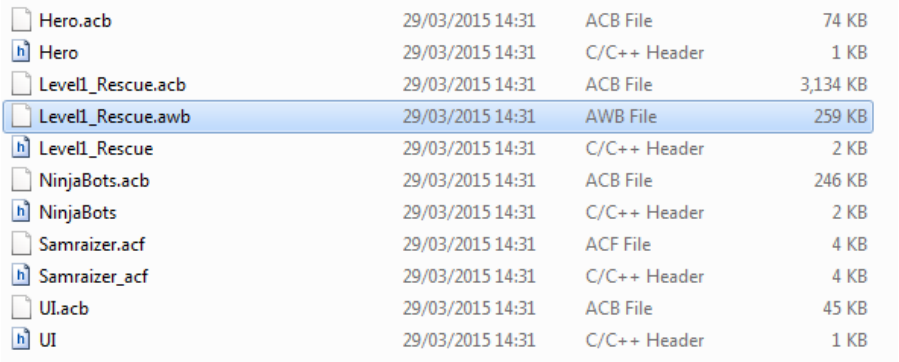
So far, we have been playing cues loaded in memory. But what about streaming from disk?

Fortunately, the process is fairly similar and with only a couple more lines we can stream a sound. What is really interesting, as you will see, is that the part of the code to play the cue does not have to be modified, which makes it very easy to switch audio assets from memory playback to streaming playback depending on your needs and the evolution of the game development.

First, the corresponding audio material must be set to streaming in the Atom Craft tool. You can do this by clicking on the file in the material folder and changing its streaming type (you will most likely want to change the encoding type as well).



For example, in our “Level1_Rescue” cue sheet in Samraizer, we have a couple of ambiences and long water sounds that could probably benefit from being streamed. Let’s switch the river_waterfall.wav sound to streaming and re-export the cue sheet. As expected, in addition to the Leve1_Rescue.acb file and its associated .h header, a Level1_Rescue.awb file has been created, which contains the encoded audio data for the river waterfall:



On the code side, we need to create a D-BAS object, just like this:

```
CriAtomDbasId dbas_id = criAtomDbas_Create(NULL, NULL, 0);
```

This should be done fairly early, before loading the ACF and ACB files. It will also need to be destroyed before finalizing the Atom library, with:

```
criAtomDbas_Destroy(dbas_id);
```

> 🔧 Key Term
> D-BAS is the system responsible for managing streaming buffers for players. The streaming buffer is divided into blocks, which D-BAS assigns as needed to the various players.

The second step on the code side is to actually load the AWB file we exported from the tool.

> 🔧 Key Term
> Although we are not using its functions directly here, please note that CriAtomAwb is the module responsible for accessing the audio data stored in AWB (Atom Wave Bank) files. Once the TOC (table of content) of the AWB file is loaded, it is possible to select a cue with an Atom Player and to start the playback.

This is done very easily, just by passing the path to the AWB file as a parameter when we load the corresponding ACB.

Therefore, if we want to load our Level1_Rescue sound bank, we should now write:

```
acbLevel1Hn = criAtomExAcb_LoadAcbFile(NULL, "C:\\Samraizer\\PC\\Level1_Rescue.acb",
NULL, "C:\\Samraizer\\PC\\Level1_Rescue.awb", NULL, 0);
```

The final step is to make sure that the voice pool will support streams. An easy way to do this is to copy the default configuration of a standard voice pool, to set the streaming flag and allocate the pool, like this:

```
CriAtomExStandardVoicePoolConfig voice_pool_config;
CriAtomExVoicePoolHn voice_pool;

/* Create a voice pool */
criAtomExVoicePool_SetDefaultConfigForStandardVoicePool(&voice_pool_config);
voice_pool_config.player_config.streaming_flag = CRI_TRUE;
voice_pool = criAtomExVoicePool_AllocateStandardVoicePool(&voice_pool_config, NULL, 0);
```

And that's all! The cue playback can be started as usual:

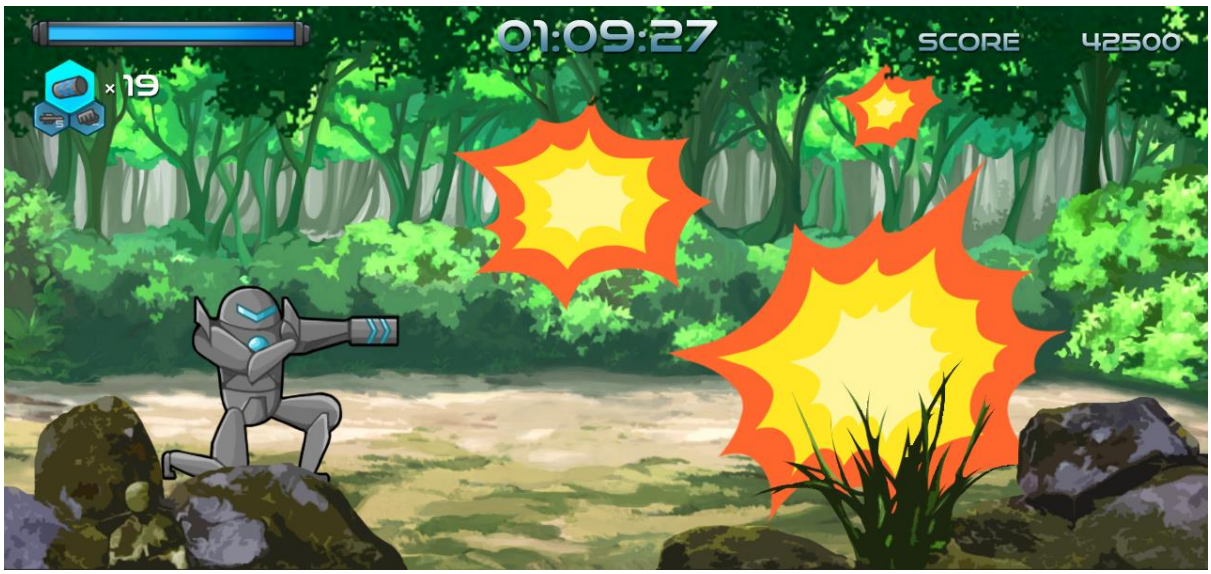
```
criAtomExPlayer_SetCueId(player, acbHnLevel1, CRI_LEVEL1_RESCUE_RIVER_WATERFALL);

criAtomExPlayer_Start(player);
```

Changing parameters

It is very easy to change the parameters of a cue programmatically before or while it is being played back by an AtomEx player.

Changing the volume

Imagine that Samraizer is under attack and many explosions of various sizes are triggered next to him. Although there are many other ways to achieve this, we could decide to use only one explosion sound file and to change its volume.



To change the volume of a cue, we call the `criAtomExPlayer_SetVolume` function on the AtomEx player with a volume value between 0.0f (silent) and 1.0 (full volume):

```
criAtomExPlayer_SetVolume(player,volume);
```

If we want to assign a volume which is in dB, we can use the following formula to convert it back to right range:

```
volume = powf(10.0f, db_vol / 20.0f);
```

The `criAtomExPlayer_SetVolume` function only transmits the new volume parameter to the AtomEx player. For the AtomEx player to actually change the value, it must either be started or - if it is already playing - the `criAtomExPlayer_Update` function must be called with the playback ID returned when starting the player:

```
/* Setting the volume before starting the playback */
criAtomExPlayer_SetVolume(player, 0.5f );
criAtomExPlayer_Start(player);

/* Updating the volume after playback has started */
CriAtomExPlaybackId playbackId = criAtomExPlayer_Start(player);
Sleep(100);
criAtomExPlayer_SetVolume(player, 0.5f );
criAtomExPlayer_Update(player,playbackId);
```

Note

You can also update all the cues played by an AtomEx player by using the `criAtomExPlayer_UpdateAll` function, in which case you don't need to pass a playback ID.

This separation between sending the new parameter value and actually taking it into account is interesting for at least two reasons. Firstly, the function to call to pass the value (`criAtomExPlayer_SetVolume`) stays the same in both cases (otherwise, we may have to pass volume parameters to `criAtomExPlayer_Start` for example). Secondly, if for some reason the volume is changed several times before the actual update needs to be done, we avoid recalculating internal parameters again and again for nothing.

To come back to Samraizer and the explosions, here is a function that plays 10 small explosions with increasing volumes.

```
PlayCueWithIncreasingVolumes(player, acbNinjaBotsHn, CRI_NINJABOTS_EXPLOSION_SHORT);


void PlayCueWithIncreasingVolumes(CriAtomExPlayerHn player,CriAtomExAcbHn
acbHn,CriAtomExCueId id)
{
	criAtomExPlayer_SetCueId(player, acbHn, id);

	for(int i = 1; i <= 10 ; ++i) {
		criAtomExPlayer_SetVolume(player, i / 10.0f );
		criAtomExPlayer_Start(player);
		WaitUntilEndOfCue(player);
	}
}
```

A couple of things are worth noting. First, see how the function setting the ID of the cue to play is outside of the loop. As we mentioned earlier, a cue is kept associated with the player unless we change it, so there is no need to set it each time. Also, the `WaitUntilEndOfCue` function is our own utility function which basically checks the status of the player and waits until it becomes `CRIATOMEXPLAYER_STATUS_PLAYEND`. We will reuse it later.

Note

Since in the next pages we will learn how to change the AtomEx player parameters in many ways, it should be noted that they can be reset at any time by calling `criAtomExPlayer_ResetParameters`.

Changing the pitch

At some point in the game, you may also want to change the pitch of a sound. For example: when a bomb is dropping, or if the player is running out of time and the clock's ticking is becoming higher and higher in pitch to highlight the danger. [Arguably, this could also be done with a game variable and AISIC as we will see later].

Changing the pitch is done very much in the same way than changing the volume. You can call the `criAtomExPlayer_SetPitch` function before starting the AtomEx player or during the playback (in which case you will have to call the `criAtomExPlayer_Update` function as well):

```
criAtomExPlayer_SetPitch(player,pitch);
```

The pitch value is expressed in cents. A cent is 1/1200th of an octave (therefore a half-tone corresponds to 100 cents). This value is relative, which means that it is added to the pitch coming from the Atom Craft tool.

For example, here is the code that plays a falling bomb sound effect. First without pitch change and then again with pitch change, for a much more dramatic effect!

We could call it with:

```
PlayCueWithDecreasingPitch(player, acbNinjaBotsHn, CRI_NINJABOTS_ENNEMY_BOMB_FALL);
```

As you can see in the code below, this time we are changing the parameter of a cue which is already being played back, lowering progressively its pitch.

```
void PlayCueWithDecreasingPitch(CriAtomExPlayerHn player,CriAtomExAcbHn
acbHn,CriAtomExCueId id)
{
	criAtomExPlayer_SetCueId(player, acbHn, id);
	criAtomExPlayer_Start(player);
	WaitUntilEndOfCue(player);

	CriAtomExPlaybackId playbackId = criAtomExPlayer_Start(player);
	CriFloat32 pitchOffset = 0;
	for(;;) {
		/* wait 10 ms */
		samraizer_sleep(10);

		/* update the pitch */
		criAtomExPlayer_SetPitch(player, pitchOffset);
		criAtomExPlayer_Update(player,playbackId);
		pitchOffset -= 5;

		/* Execute the server process */
		criAtomEx_ExecuteMain();

		/* Check if we are done */
		CriAtomExPlayerStatus explayer_status;
		explayer_status =criAtomExPlayer_GetStatus(player);
		if (explayer_status == CRIATOMEXPLAYER_STATUS_PLAYEND) {
			break;
		}
	}
}
```

Note

Pitch changes are implemented by modifying the voice playback rate. While it is straightforward when lowering the pitch, if we want to increase it, we will need to ensure that the sample rate of the voice pool is high enough beforehand.

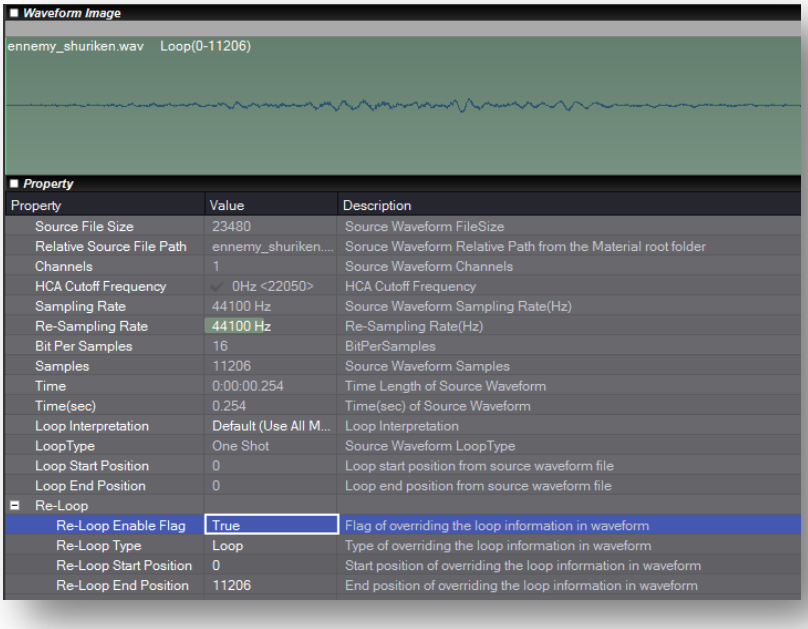
For example, if we have a voice sampled at 48 kHz and we decide to raise its pitch from one octave (i.e. doubling it), then the voice pool sampling rate must be set to 96 kHz in advance. This can be done through the voice pool configuration. In this case:

```
voice_pool_config.player_config.max_sampling_rate = 96000;
```

Changing the pan

As the fearsome NinjaBots attack Samraizer, they may launch shurikens towards him. To add realism to this action sequence, we probably should change the pan of the shuriken cue progressively while it is flying towards our hero.

First in this example, we need to make sure that our shuriken sound is looping so that it won't stop playing mid-course! This is a good opportunity to remind ourselves than by clicking on a waveform in the material folder in Atom Craft, we can access - and edit - all its information, from streaming type to encoding format, looping parameters and so on… Below, we can see that our shuriken wave file does not have a loop initially, so we set one in the Re-Loop section.



Next, let's see how we can change the panning.

Although we are talking about 2D pan in that case, changing it can be done with a combination of these pan 3D functions:

```
criAtomExPlayer_SetPan3dAngle
criAtomExPlayer_SetPan3dInteriorDistance
```

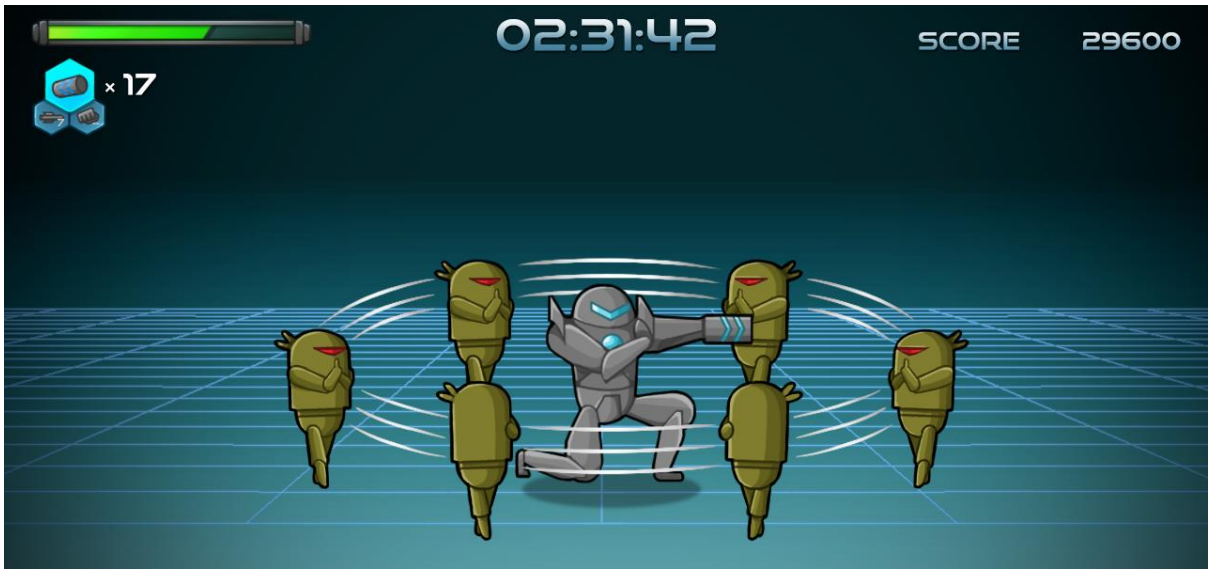
There are many other functions available when dealing with panning in 3D (which involves calculations based on the positions of the sound sources and of the listener) and we will learn about some of them in the next section.

Here, we set the panning angle to 90 degrees and the interior distance is between -1.0f (far left) and +1.0f (far right), 0.0f representing the center. Let's throw that shuriken!

```
PlayCueWithPanChange(player, acbNinjaBotsHn, CRI_NINJABOTS_ENNEMY_SHURIKEN);

void PlayCueWithPanChange(CriAtomExPlayerHn player,CriAtomExAcbHn acbHn,CriAtomExCueId
id)
{
	criAtomExPlayer_SetCueId(player, acbHn, id);
	CriFloat32 angle = 90;
	criAtomExPlayer_SetPan3dAngle(player,angle);
	CriAtomExPlaybackId playbackId = criAtomExPlayer_Start(player);

	CriFloat32 pan = 1.0f;
	for(;;) {
		/* wait 10 ms */
		samraizer_sleep(10);

		/* update the pan */
		pan -=0.01f;
		criAtomExPlayer_SetPan3dInteriorDistance(player,pan);
		criAtomExPlayer_Update(player,playbackId);

		/* Execute the server process */
		criAtomEx_ExecuteMain();

		/* Check if we are done */
		CriSint64 time = criAtomExPlayer_GetTime(player);
		if (time > 3000)
		{
			criAtomExPlayer_Stop(player);
			break;
		}
	}
}
```

Applying 3D positioning

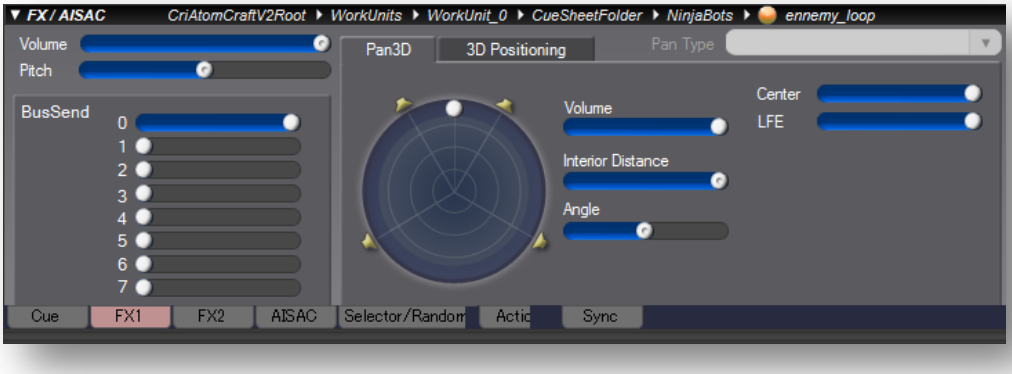
To get a more realistic feeling about where the sound comes from, a game audio middleware uses 3D positioning. For example, let's imagine Samraizer reached the end-level boss and that this NinjaBot is using his super powers to rotate quickly around him while delivering punches. How can we simulate the sound of this scene?

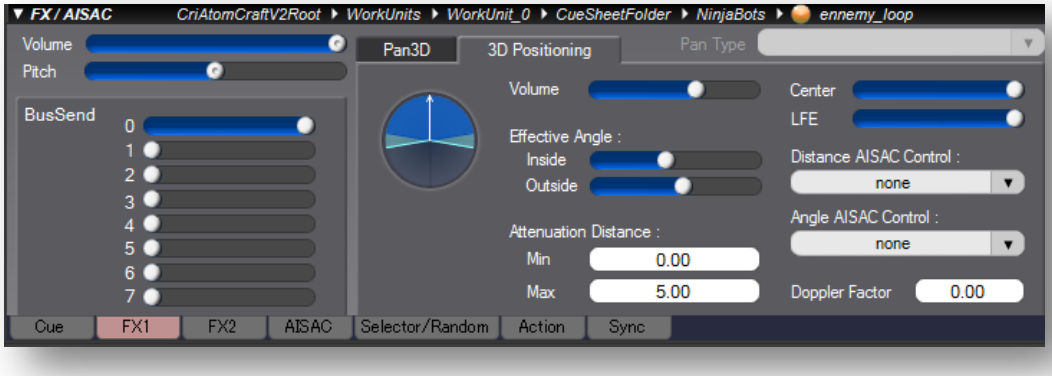


3D audio positioning relies on 2 types of objects: listeners and audio emitters (sound sources). 3D audio positioning calculates the distribution of the sound between all the channels based on the position and the direction of these objects.

Usually, there will be only one listener object, attached either to the player's character (if this is a third person view) or to the position of the camera (in a first person shooter for example). However, it is sometimes possible to have several listeners, for example in the case of split-screen games. The Atom library supports this as well.

There are often many sound sources, each one with its own emitting cone, distance range etc… Again the Atom library supports all these features and most of them can be set either by code or in the Atom Craft tool:





To be able to use all these features, we first need to make sure that the player is using the 3D positioning panning type:

```
criAtomExPlayer_SetPanType(player, CRIATOMEX_PAN_TYPE_3D_POS);
```

Then, we have to create a listener object and at least one sound source. This is done by calling the following functions:

```
CriAtomEx3dListenerHn listener = criAtomEx3dListener_Create(NULL, NULL, 0);
CriAtomEx3dSourceHn source = criAtomEx3dSource_Create(NULL, NULL, 0);
```

Once these objects are created, we want to assign them to the AtomEx player, like this:

```
criAtomExPlayer_Set3dListenerHn(player, listener);
criAtomExPlayer_Set3dSourceHn(player, source);
```

Now, every time we will update the sound source or the listener position, our player will know about it. To represent 3D coordinates, the CRI Atom library uses its own class: `CriAtomExVector` . For example, we can set the position of the listener like this:

```
CriAtomExVector pos;
pos.x = 0.0f;
pos.y = 0.0f;
pos.z = 0.0f;
criAtomEx3dListener_SetPosition(listener, &pos);
```

There is a similar function to set the position of a sound source: `criAtomEx3dSource_SetPosition`. This is the function that we will use to periodically update the position of the source. Changes of positions of both the listener and the source will only be taken into account if we call `criAtomEx3dSource_Update` and `criAtomEx3dListener_Update` respectively.

It is also possible to set the velocities of both listener and sound sources, which will be used to calculate the Doppler Effect. Of course, when you don't need the listener and the sound source objects anymore, don't forget to release them with:

```
criAtomEx3dListener_Destroy(listener);
criAtomEx3dSource_Destroy(source);
```

There are many other 3D positioning-related functions in the Atom library, but for this Quick Start guide we will stop here. Let's review what we need to do for the NinjaBot boss to rotate around Samraizer (the listener):

```
void PlayCueWith3DPositioning(CriAtomExPlayerHn player,CriAtomExAcbHn
abHn,CriAtomExCueId id)
{
    /* Set 3d positoning pan type */
    criAtomExPlayer_SetPanType(player, CRIATOMEX_PAN_TYPE_3D_POS);

    /* Create a listener and a source objects */
    CriAtomEx3dListenerHn listener = criAtomEx3dListener_Create(NULL, NULL, 0);
    CriAtomEx3dSourceHn source = criAtomEx3dSource_Create(NULL, NULL, 0);

    /* Assign them to the AtomEx player */
    criAtomExPlayer_Set3dListenerHn(player, listener);
    criAtomExPlayer_Set3dSourceHn(player, source);

    /* Set listener initial position */
    CriAtomExVector pos;
    pos.x = 0.0f;
    pos.y = 0.0f;
    pos.z = 0.0f;
    criAtomEx3dListener_SetPosition(listener, &pos);
    criAtomEx3dListener_Update(listener);

    /* Set source initial position and distance range*/
    CriFloat32 distance = 10.0f;
    pos.x = distance;
    pos.z = 0.0f;
    criAtomEx3dSource_SetPosition(source, &pos);
    criAtomEx3dSource_SetMinMaxDistance(source,0,100);
    criAtomEx3dSource_Update(source);

    /* Start playback */
    criAtomExPlayer_SetCueId(player, acbHn, id);
    id = criAtomExPlayer_Start(player);

    /* Make the source rotate around the listener for 10 seconds*/
    for (CriUint16  i = 0; i <  1000; ++i) {

       /* Wait 10 ms */
       Sleep(10);

       /* Execute the server process */
       criAtomEx_ExecuteMain();

       /* Update the position of the source */
       pos.x = distance * cos(6.28f * (CriFloat32)i / 1000.0f);
       pos.z = distance * sin(6.28f * (CriFloat32)i / 1000.0f);
       criAtomEx3dSource_SetPosition(source, &pos);
       criAtomEx3dSource_Update(source);
       }

    criAtomExPlayer_Stop(player);

    /* Free sound source and listener objects */
    criAtomEx3dListener_Destroy(listener);
    criAtomEx3dSource_Destroy(source);
}
```

Applying a filter

In some cases you will want to use filters, for example to simulate occlusion (when a sound source is located behind a wall).

Imagine that Samraizer is trying to rescue a princess who has been captured by the NinjaBots. She is kept imprisoned in a box (hopefully there are some holes so she can breathe!). She is screaming for help. Since her voice comes from inside the box, it will be muffled, something that can be advantageously simulated by a low-pass filter.

The function `criAtomExPlayer_SetBiquadFilterParameters` can configure different types of filters: low pass, high pass, notch, low-shelf, high-shelf and peaking. By specifying the type of the filter, but also its parameters such as cutoff frequency, gain and resonance, the Atom library will calculate the correct filter coefficients for the player. Please note that the slope of the filter will be 12 dB maximum as it is a biquad filter. In our case, we want to use a lowpass filter with a relatively low cutoff frequency so we could use the following code:

```
void PlayCueWithLowpass(CriAtomExPlayerHn player,CriAtomExAcbHn acbHn,CriAtomExCueId
id)
{
	/* normal play */
	criAtomExPlayer_SetCueId(player, acbHn, id);
	criAtomExPlayer_Start(player);
	WaitUntilEndOfCue(player);

	/* setup lowpass filter and play again */
	CriAtomExBiquadFilterType type = CRIATOMEX_BIQUAD_FILTER_TYPE_LOWPASS;
	CriFloat32 frequency = 0.35f;
	CriFloat32 gain = 1.0f;
	CriFloat32 q = 3.0f;
	criAtomExPlayer_SetBiquadFilterParameters(player, type, frequency, gain, q);
	criAtomExPlayer_Start(player);
	WaitUntilEndOfCue(player);
}
```

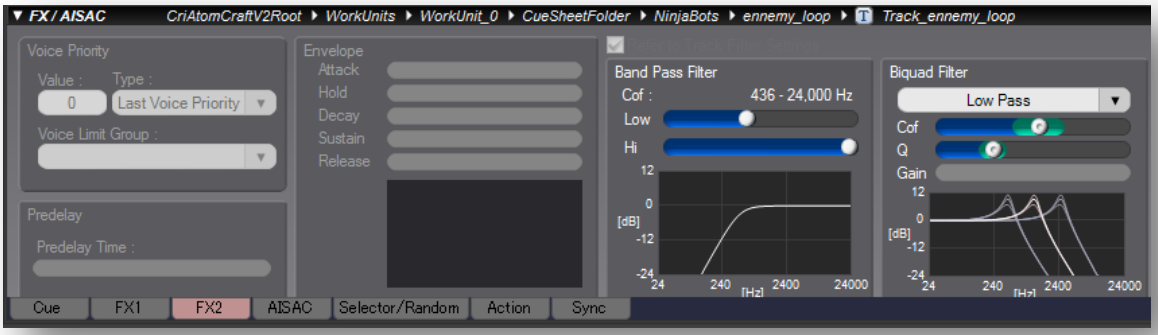
As you can see from the code above, the frequency is in the range [0…1], which means it is normalized ([0..1] is mapped to 24Hz to 24000Hz on a logarithmic scale). The gain is expressed in dB and the Q factor has no unit.

The AtomEx player also supports an extra bandpass filter, which can be set with the `criAtomExPlayer_SetBandpassFilterParameter` function, like this:

```
criAtomExPlayer_SetBandpassFilterParameter(player, lowCutoffFrequency,
highCutoffFrequency);
```

By specifying 0 for the low cutoff frequency or 1.0 for the high cutoff frequency, it is also possible to simulate a lowpass filter or a highpass filter in addition to a bandpass.

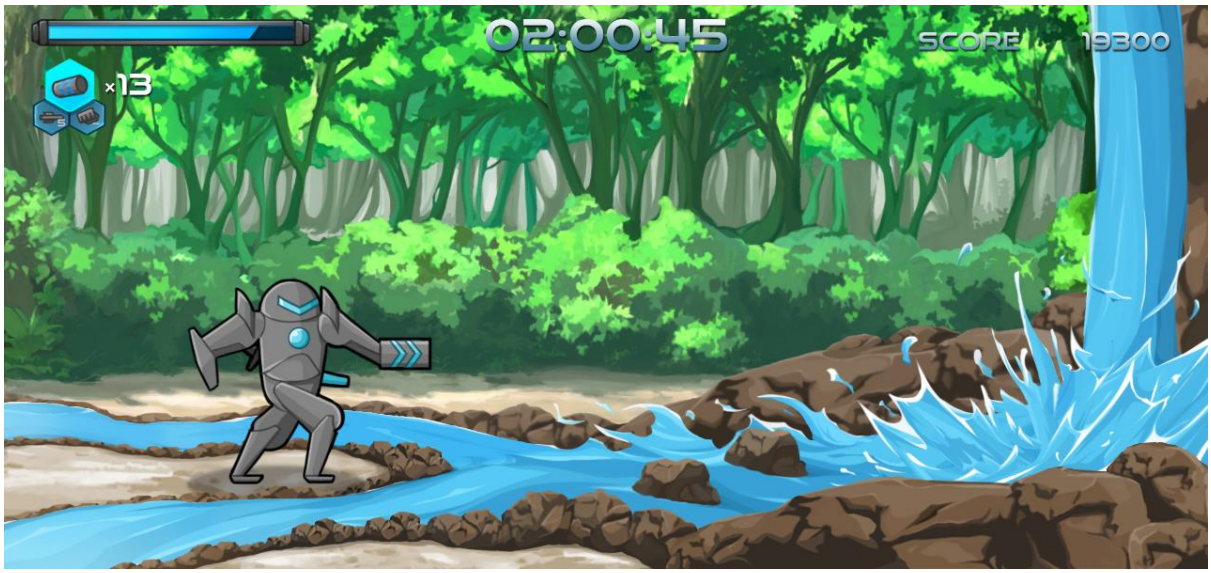
Don't forget that the Atom Craft tool offers a way to set both filters with a graphic user interface, and even to set random ranges for some of their parameters. So unless you have a very specific use of the filter, a lot of things can be done without even typing a line of code!



> **Note**
> We have seen how to update the volume, pitch, pan and filter values dynamically. Please note that if you just want to add random variations to these parameters, you can do it in the Atom Craft tool by selecting random ranges, so it is usually not necessary to program them.

Controlling sounds from game parameters

Sometimes it is necessary to make the sound more responsive to what is happening in the game. For example, on the screen below, we can see Samraizer walking towards a waterfall. Where he is standing, the river is just a small quiet stream. At the other extremity of the screen though, the waterfall looks very powerful and noisy. And in-between, the water looks agitated and waves are crashing against the rocks.

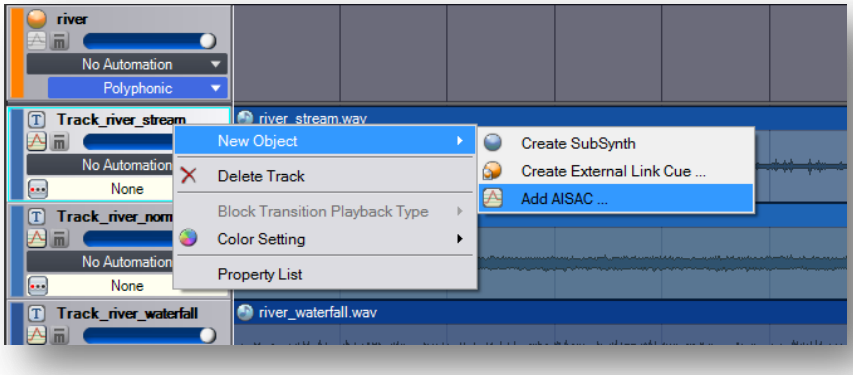


It would be great if, as Samraizer approaches the waterfall, the sound of the water was gradually becoming more powerful. That's exactly the type of behavior we can achieve with AISAC.

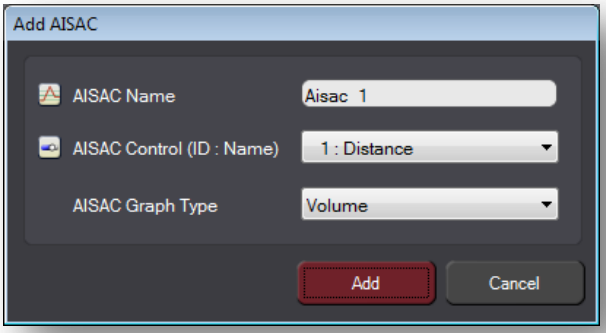
> Key Term
> *AISAC* is a system allowing the control of audio parameters by values coming from the game.

In our Level1_Rescue cue sheet, we created a polyphonic cue with 3 tracks playing water sounds with different levels of movement (small stream, normal and waterfall). We also made them loop.

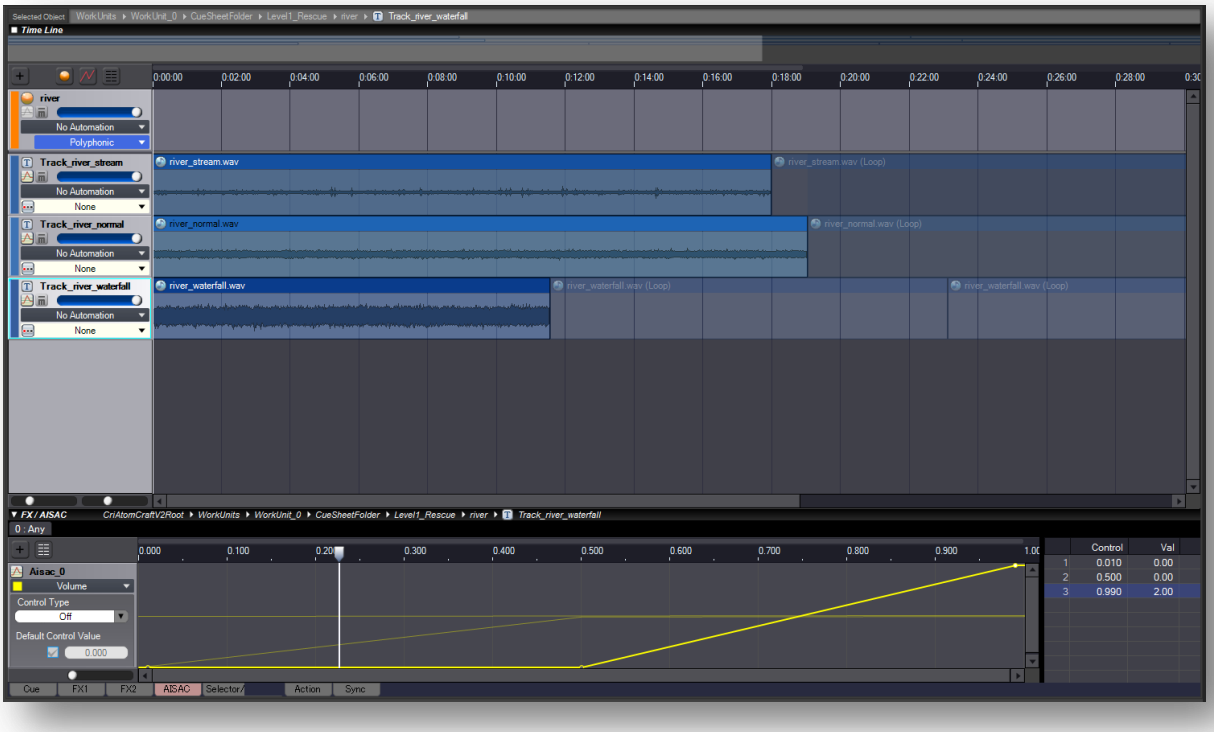
By right-clicking on the tracks we can add an ASAIC control:



When we do it, the following window appears, allowing us to associate a parameter with the new AISAC control. Many parameters can be controlled by AISAC, such as volume, pitch, pan, filter and so on. The idea here is to make the three water sounds vary in volume based on a game parameter.



For each track, we can now draw how we want to map the volume based on the game parameter value. This parameter will actually be the distance from the waterfall, and like all AISAC parameters it will go from 0 (left of the graph) to 1 (right of the graph). As you can see below, we will always be playing the stream sound, while the normal river sound will appear progressively and will reach its maximum volume around the middle of the screen. The waterfall sound also appears progressively but only in the second part of the screen, and it becomes overpowering when we reach the right-most position.



Most of the work is already done in the tool. On the code side, we simply need to call the `criAtomExPlayer_SetAisacById` function to trigger a quite complex audio behavior:

```
criAtomExPlayer_SetAisacById(player, aisacID, parameterValue);
```

`aisacID` is of course the ID of the AISAC control while the `parameterValue` will be between 0 and 1. There is also a `criAtomExPlayer_SetAisacByName` function to reference the AISAC control by name.

Here is how the code will look:

```
void PlayCueWithAISAC(CriAtomExPlayerHn player,CriAtomExAcbHn acbHn,CriAtomExCueId id)
{
    /* Starts the sound */
    criAtomExPlayer_SetCueId(player, acbHn, id);
    criAtomExPlayer_Start(player);

    /* Plays back while controlling AISAC control value */
    for(CriFloat32 distance = 0.0f; distance <= 1.0f; distance += 0.0005f) {

       /* Wait 10 ms */
       Sleep(10);

       /* Execute the server process */
       criAtomEx_ExecuteMain();

       /* Change the zeroth AISAC control value */
       criAtomExPlayer_SetAisacById(player, 0, distance);

       /* Update */
       criAtomExPlayer_UpdateAll(player);
       }

    /* Stops player */
    criAtomExPlayer_Stop(player);
}
```

Managing many sounds

Until now, we have been playing back single cues. But when many things are happening at the same time in the game - for instance if Samraizer is in the heat of the battle and projectiles are coming from everywhere - new techniques will be needed. It is possible that there might not be enough voices, or that the level of a group of voices should be lowered, or that sounds should be synchronized together etc… This section is about managing more than a single cue!

Playing multiple voices

You can load as many ACB files as you want at the same time (or at least as many as your memory budget allows you!) and you can play cues from different ones simultaneously. There are several ways to playback multiple voices with the Atom library and the best choice will depend on the context. It will sometimes be simpler to prepare as much as possible in the tool and to play a single cue on the code side, which may trigger many tracks, each including a lot of waveforms. While this basically puts everything in the hands of the sound designer rather than the programmer, it does not prevent the audio to be dynamically changed by the game, especially with a system such as AISAC. In other cases, we may want to send multiple playback requests to a single player, like this:

```
/* Create a player */
CriAtomExPlayerHn player = criAtomExPlayer_Create(NULL, NULL, 0);

/* Set a Cue ID and start playback */
criAtomExPlayer_SetCueId(player, acbNinjaBotsHn,CRI_NINJABOTS_EXPLOSION_SHORT);
criAtomExPlayer_Start(player);

/* Set a Cue ID and start playback */
criAtomExPlayer_SetCueId(player, acbNinjaBotsHn,CRI_NINJABOTS_EXPLOSION_MEDIUM);
criAtomExPlayer_Start(player);

/* Set a Cue ID and start playback */
criAtomExPlayer_SetCueId(player, acbNinjaBotsHn,CRI_NINJABOTS_EXPLOSION_LONG);
criAtomExPlayer_Start(player);
```

However, since only one player is used, it is not possible to control each voice separately during playback. Another option in that case is to create multiple players and send a playback request to each (which of course is more cumbersome to manage and uses more resources).

```
/* Start a cue on player 1 */
CriAtomExPlayerHn player1 = criAtomExPlayer_Create(NULL, NULL, 0);
criAtomExPlayer_SetCueId(player1, acbNinjaBotsHn, CRI_NINJABOTS_EXPLOSION_SHORT);
criAtomExPlayer_Start(player1);

/* Start a cue on player 2 */
CriAtomExPlayerHn player2 = criAtomExPlayer_Create(NULL, NULL, 0);
criAtomExPlayer_SetCueId(player2, acbNinjaBotsHn, CRI_NINJABOTS_EXPLOSION_MEDIUM);
criAtomExPlayer_Start(player2);

/* Start a cue on player 3 */
CriAtomExPlayerHn player3 = criAtomExPlayer_Create(NULL, NULL, 0);
criAtomExPlayer_SetCueId(player3, acbNinjaBotsHn, CRI_NINJABOTS_EXPLOSION_LONG);
criAtomExPlayer_Start(player3);
```

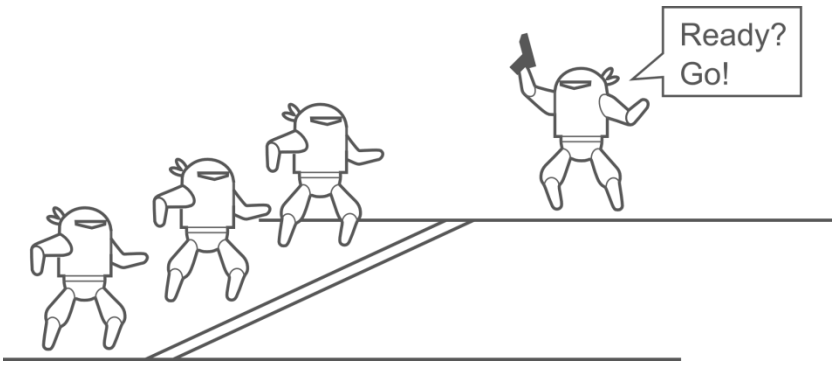
Synchronizing voices

If we are starting a large number of voices, in some instances we will also want them to be synchronized. A function of the AtomEx player is especially useful in that case: `criAtomExPlayer_Prepare`.

This function starts the sound playback in paused mode. When the function is called, resource allocation and buffering are initiated but the playback itself will not start, even when the buffering has completed, since it is in paused mode.

It is possible to determine if the playback is ready or not by checking the status of the playback ID returned by `criAtomExPlayer_Prepare`. If the status is `CRIATOMEXPLAYBACK_STATUS_PLAYING` the playback is ready, if it is still `CRIATOMEXPLAYBACK_STATUS_PREP` it is not ready yet.

By starting several cues that way and waiting for the playback status to become `CRIATOMEXPLAYBACK_STATUS_PLAYING`, we can actually start playing the cues synchronized. All we need to do is to call the `criAtomExPlayer_Pause` function to resume playback when we are ready.



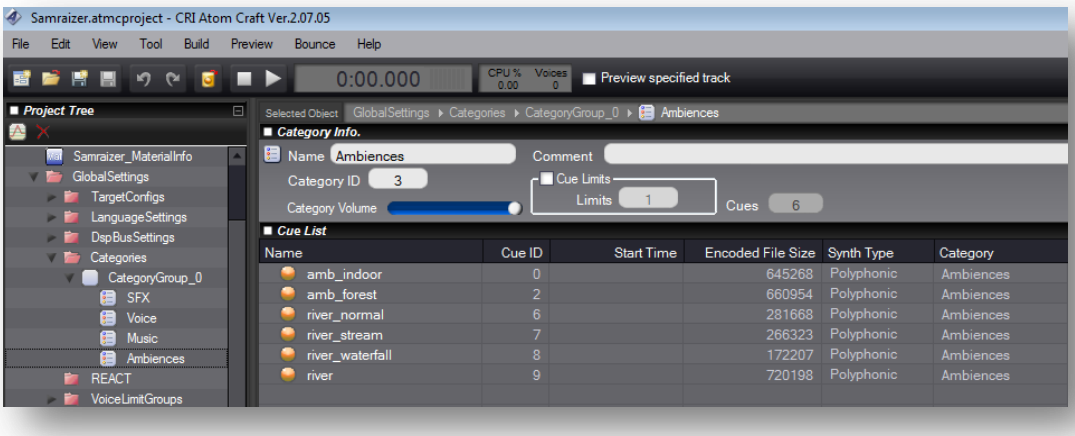
As an example, here is a function that starts the 3 river sounds we met earlier at the same time:

```
PlayCuesSynchronized(player,  acbLevel1Hn, CRI_LEVEL1_RESCUE_RIVER_STREAM,
CRI_LEVEL1_RESCUE_RIVER_NORMAL, CRI_LEVEL1_RESCUE_RIVER_WATERFALL);
```

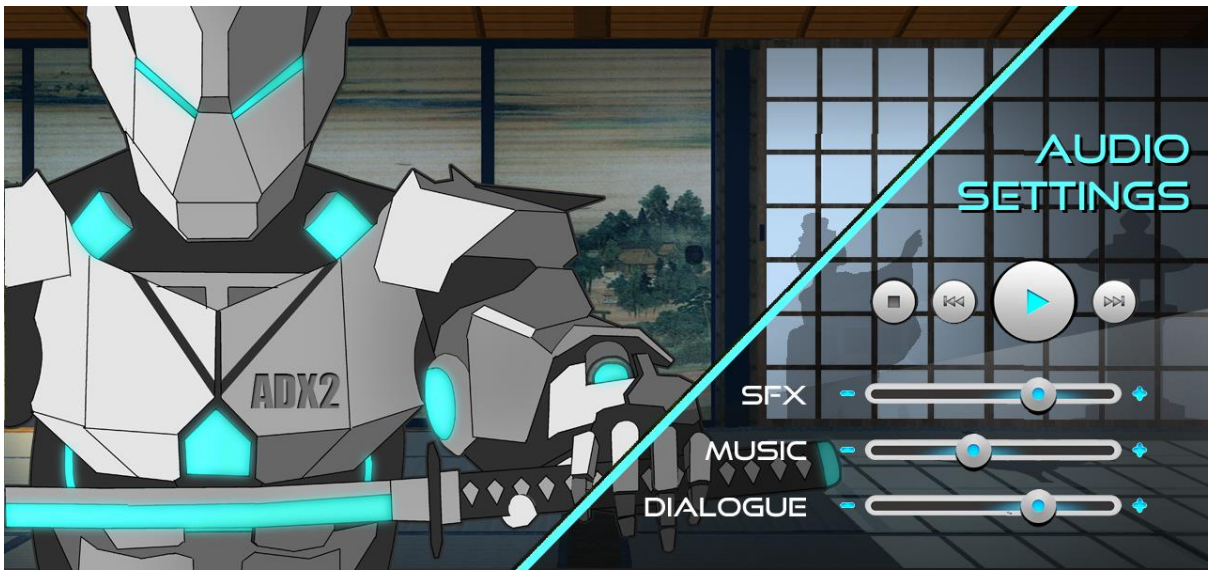
```
void PlayCuesSynchronized(CriAtomExPlayerHn player,CriAtomExAcbHn acbHn,CriAtomExCueId
id1,CriAtomExCueId id2,CriAtomExCueId id3)
{
    /* Prepare playback */
    criAtomExPlayer_SetCueId(player, acbHn, id1);
    criAtomExPlayer_Prepare(player);
    criAtomExPlayer_SetCueId(player, acbHn, id2);
    criAtomExPlayer_Prepare(player);
    criAtomExPlayer_SetCueId(player, acbHn, id3);
    criAtomExPlayer_Prepare(player);

    for(;;) {
        CriAtomExPlayerStatus player_status;
        Sleep(1);
        criAtomEx_ExecuteMain();

        /* Check if playback is ready */
        player_status = criAtomExPlayer_GetStatus(player);
        if (player_status == CRIATOMEXPLAYBACK_STATUS_PLAYING) {
            break;
        }
    }

    /* Start synchronized playback */
    criAtomExPlayer_Pause(player, CRI_FALSE);
}
```

Categories

Categories are a convenient way to manipulate several cues at once. Several categories (up to 4) can be assigned to a cue. These categories (and sub-categories) must be created and assigned by the sound designer in the Atom craft tool. A quick way to assign them is to simply drag cues and drop them onto the chosen category. The picture below shows the list of categories in our Samraizer project on the left, as well as the cues in the “Ambiences” category on the right:



An example of usage of categories could be the “Audio Settings” screen of Samraizer. Indeed, it would be great if the player was able to adjust the level of the sound effects, music and dialogue in the game as he or she wants.



Of course, it would be far too cumbersome to do that individually for each cue. However, if all the sound effects belong to the “sfx” category, all the songs have been tagged with the “music” category and the all voice lines are part of the “dialogue” category, it becomes straightforward.

In that case, we can very simply change the volume of all the cues in the “Voice” category like this:

```
criAtomExCategory_SetVolumeByName("Voice",0.5f);
```

We could also mute all the sound effects with a single line:

```
criAtomExCategory_MuteByName("SFX",CRI_TRUE);
```

Or solo the sounds of a given category, like the ambiences:

```
criAtomExCategory_SoloByName("Ambiences",CRI_TRUE,0.0f);
```

> Tip
> This can be very useful in debug code too, to isolate certain types of sounds in order to better listen to them.

Many other functions have category-specific versions. For example it is possible to pause / resume all the sounds of a given category, set their fading times, to attach an AISAC controller to them etc…

Mixing with Fader

Fader is a system that allows the Atom library to automatically make fade-in, fade-out and crossfades between sounds.

For example, if the player decides to pause Samraizer, the screen should switch to a pause menu and the music should gently crossfade from the high-adrenaline action theme to the softer pause music.

The first step is to attach a Fader to the AtomEx player with the following call:

```
criAtomExPlayer_AttachFader(player, NULL, NULL, 0);
```

To set up the fade-in time and the fade-out time, call the `criAtomExPlayer_SetFadeInTime` and `criAtomExPlayer_SetFadeOutTime` functions respectively, with the time expressed in milliseconds. These same values will be used if you want to do a crossfade between two sounds. For example, here we set both times at one second:

```
criAtomExPlayer_SetFadeInTime(player, 1000);
criAtomExPlayer_SetFadeOutTime(player, 1000);
```

From now on, whenever we start a sound on this player it will fade in for one second, and whenever we stop a sound it will take one second to fade out. If we keep calling `criAtomExPlayer_Start` with different cues, then a crossfade will happen between these cues.

Here is the function we could use for our pause menu:

```
PlayCuesWithFading(player,  acbLevel1Hn, CRI_LEVEL1_RESCUE_ACTIONMUSIC, acbUIHn,
CRI_UI_PAUSEMUSIC);
```

It will make the action music fade-in, wait for a bit, then crossfade between the action music and the pause music, wait a bit more, and finally fade out the pause music.

```
void PlayCuesWithFading(CriAtomExPlayerHn player,CriAtomExAcbHn acbHn1,CriAtomExCueId
id1,CriAtomExAcbHn acbHn2,CriAtomExCueId id2)
{
    /* Attach a Fader and initialize the in and out fading times to 1 second */
    criAtomExPlayer_AttachFader(player, NULL, NULL, 0);
    criAtomExPlayer_SetFadeInTime(player, 1000);
    criAtomExPlayer_SetFadeOutTime(player, 1000);

    /* start the first song (with fade-in)*/
    criAtomExPlayer_SetCueId(player, acbHn1, id1);
    criAtomExPlayer_Start(player);

    /* wait for 5 seconds */
    Sleep(5000);

    /* start the second song (with crossfade) */
    criAtomExPlayer_SetCueId(player, acbHn2, id2);
    criAtomExPlayer_Start(player);

    /* wait for 5 seconds */
    Sleep(5000);

    /* stop the second song */
    criAtomExPlayer_Stop(player);

    /* wait for fade-out to finish */
    Sleep(1000);

    /* detach Fader */
    criAtomExPlayer_DetachFader(player);
}
```

Tip

By combining the categories and the Fader systems (i.e. using the `criAtomExCategory_SetFadeInTimeByName` and `criAtomExCategory_SetFadeOutTimeByName` functions), it becomes possible to create complex mixing behaviors.

DSP busses and effects

To conclude, let's imagine that Samraizer is entering a cave. Most of the sounds will be reflected against the rocky walls again and again, creating a reverberation effect.

Effects such as reverberation are usually assigned to several sounds at the same time (every sound in that cave will reverberate), so we are using busses to simulate them. Also it would be too CPU-intensive for the Atom library to add one full reverberation effect on each AtomEx player!

Busses can be defined in the DSP Settings in Atom Craft. The default DSP Settings already includes a "Reverb" effect on bus 1 and this is what we will be using. The screen below shows how it looks in Atom Craft.

We can see the "Reverb" effect as well as all its parameters on the right. Of course, effects are not limited to reverberation and several effects can be chained on a same bus. The Atom library includes

effects such as compressors, pitch-shifters, flangers, distortions, delays etc... However, be aware that the effects available and their implementation may vary from one target platform to another.

On the code side, DSP Settings must first be "attached". This is done by calling the `criAtomEx_AttachDspBusSetting` function, for example in the case of the default DSP Setting:

```
criAtomEx_AttachDspBusSetting("DspBusSetting_0", NULL, 0);
```

Attaching DSP settings is a blocking operation which should be done during the game initialization or when switching between levels. The ACF file must have been already loaded as it contains the definition of the DSP Settings. Of course, DSP Settings must also be "detached" before unregistering the ACF file with:

```
criAtomEx_DetachDspBusSetting();
```

Once the DSP Settings are attached, using them is very straightforward. The `criAtomExPlayer_SetBusSendLevel` function allows us to specify to which bus and at which level the sound should be sent. The level passed is a value between 0.0f and 1.0f.

The following code plays the sound of the hero's laser twice: first without reverberation and then with a bit of reverberation.

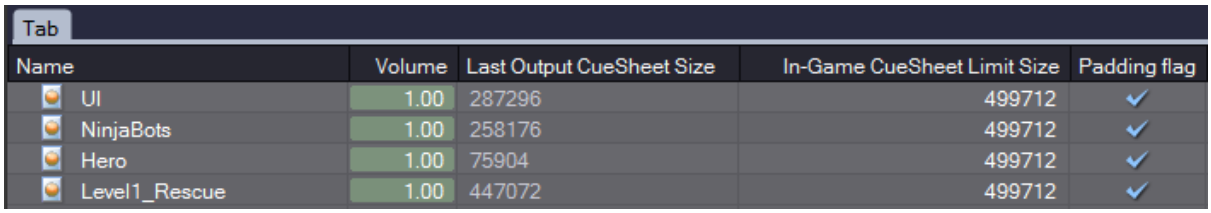
```
PlayCueWithReverb(player, acbHeroHn, CRI_HERO_LASER_01);

void PlayCueWithReverb(CriAtomExPlayerHn player,CriAtomExAcbHn acbHn,CriAtomExCueId
id)
{
	/* normal play */
	criAtomExPlayer_SetCueId(player, acbHn, id);
	criAtomExPlayer_Start(player);
	WaitUntilEndOfCue(player);

	/* play with reverb */
	criAtomExPlayer_SetCueId(player, acbHn, id);
	criAtomExPlayer_SetBusSendLevel(player,1,0.3f);
	criAtomExPlayer_Start(player);
	WaitUntilEndOfCue(player);
}
```

In-game preview and profiling

When developing a game, there is no better way to edit, test, debug or profile than in real-time, within the game itself. This dramatically reduces iteration times. The ADX2 “in-game preview” feature allows you to do just that.

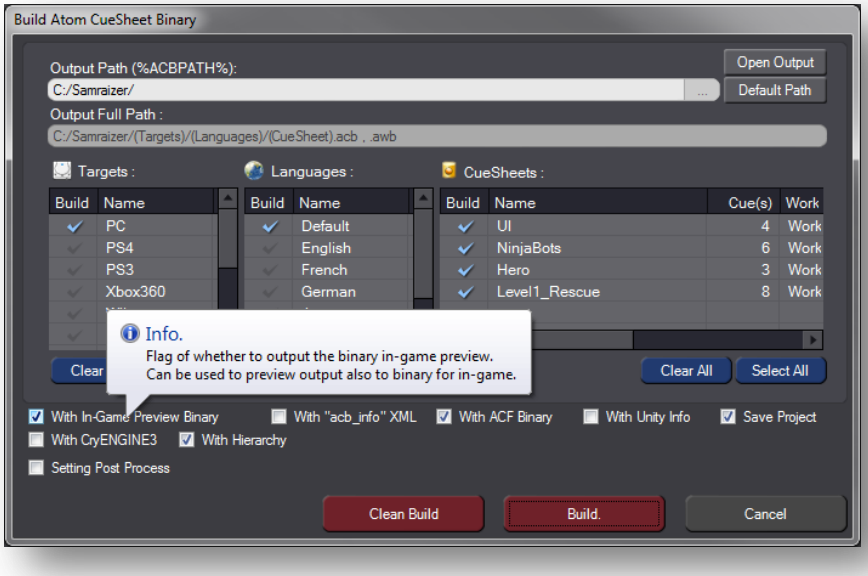
In order to be able to run an in-game preview, special versions of the project and cue sheet files (i.e. ACF and ACBs) must be built. This is because they need a bit more information, like the default values of the parameters for example. They also need to have a bigger size to accommodate potential changes in the cue data or in the waveforms.

It is possible to see and edit the maximum size allocated for the various cue sheets in the tool, by clicking on their parent cue sheet folder.



The size limit for the ACF file can be set after selecting the “Global Settings” item in the project tree.

To create these special ACF and ACB files, we have to select the “With In-Game Preview Binary” option when we export the project:



They will be exported in a special “inGamePreview” subfolder and will indeed be bigger than their regular counterparts. All other files such as C/C++ headers are also generated in that subfolder, so it is easy to just switch your main path in the code.

For the game and the Atom Craft tool to communicate, the run-time must use the monitor library, AtomExMonitor. First we have to make sure that cri_atom_monitor_pcx86D.lib is specified in the linker inputs and that we included the monitor library header:

```
#include "cri_atom_ex_monitor.h"
```

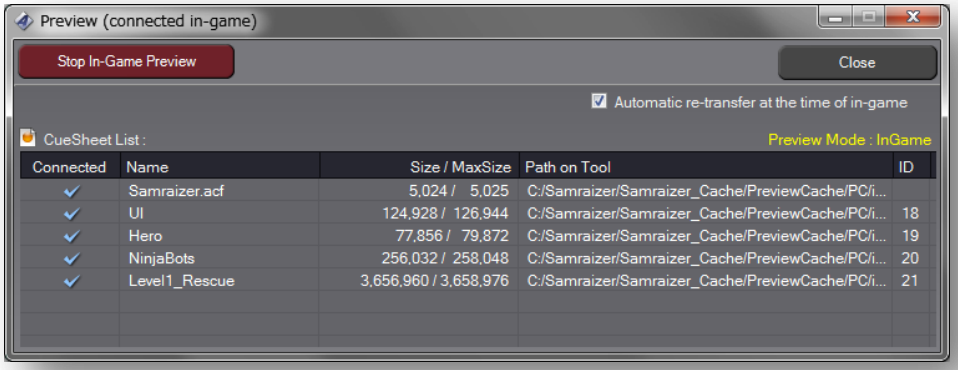
Then we simply need to initialize the library, right after we initialize the Atom library itself:

```
criAtomExMonitor_Initialize(NULL, NULL, 0);
```

Later we can finalize it by calling:

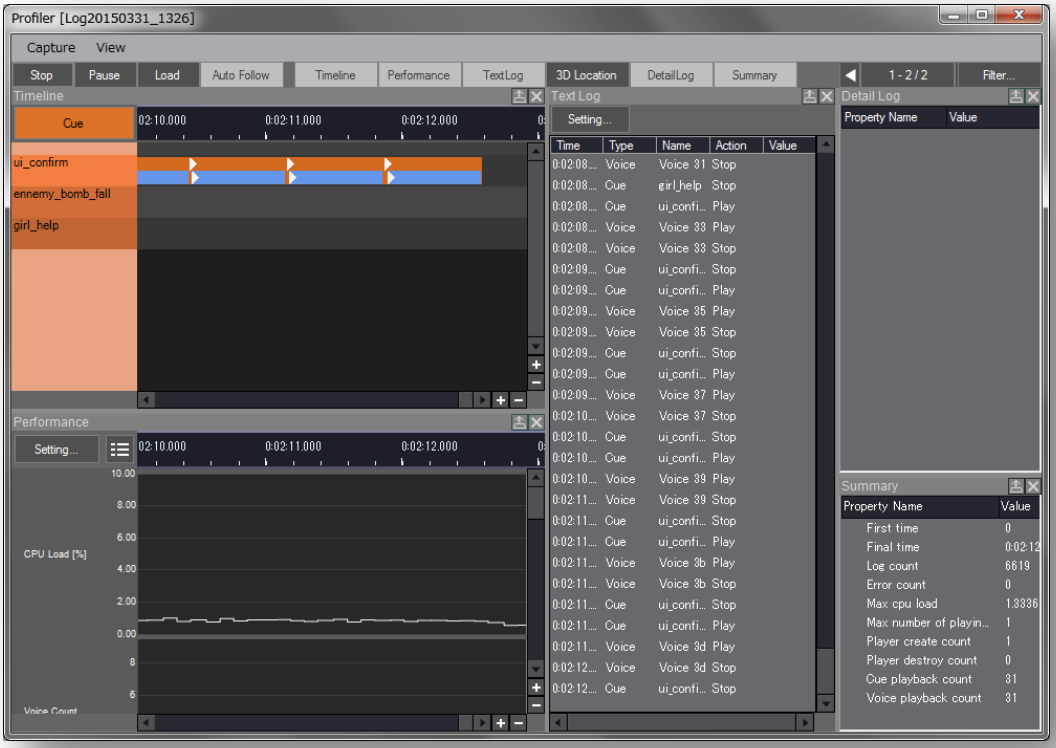
```
criAtomExMonitor_Finalize();
```

To start the game preview, we first run the game (which acts as the server) and then initiate the in-game preview by pressing F10 in the tool (which acts as the client). The connection window will show the ACF and ACB files and their current and maximum sizes:



At that point, it is possible to edit the contents of the cue sheets with the game running. If the modification only concerns basic parameters and does not incur a change in the size of the data, it can be done in real-time without stopping the sound. However if the editing will trigger a change in the size of the data, the entire ACB file will have to be updated. In that case, subsequent updates will not be heard until the ACB is actually updated by clicking on the connection item in the preview window.

Once the game is connected to the tool, it is also possible to launch the profiler. This is an invaluable tool that will allow us to get statistics such as the amount of AtomEx players created, how many cues or even voices were played back etc… The profiler's timeline shows us all this data in real-time, including the CPU load.



Finally, the monitoring library offers a logging function that will send text back to the tool to be displayed. For example:

```
criAtomExMonitor_OutputUserLog("Starting river stream cue");
PlayCue(player,acbLevel1Hn, CRI_LEVEL1_RESCUE_RIVER_STREAM);
```

Stage cleared!

Congratulations! You reached the end of the first level. You know how to initialize the CRI Atom library, load the cue sheets you created in Atom Craft, play the cues and update them either directly or through game parameters, as well as many other useful things.

But this Quick Start manual is only the beginning. Very complex audio behaviors can be achieved by combining the sound designer-friendly features of Atom Craft and the power of the CRI Atom API. The Atom library includes so many features and configuration options that it is impossible to cover everything in a Quick Start manual. However, we hope that through this introduction, you were able to learn some of the most important concepts and to realize how easy it is to implement game audio with ADX2.

Both the authoring tool and the run-time library have been developed over many years and have been used in a multitude of games. Now it's your turn to create engaging interactive audio content for your game with ADX2. See you in the next level!