faderboy's Game Sound Workshop

An Introduction to Sound and Music
Using Conitec's 3D Game Studio

by faderboy

a.k.a. Gilberto 'Gil' Morales

Table of Contents

1) Prologue

Over the past few years, as I began to learn the skills of game design, I have been helped by many a generous soul who answered my questions regarding coding and graphics, as I have often answered questions regarding sound. I have also observed that there is so much emphasis on code and graphics that quite often sound and music are but afterthoughts, yet in successful games they are of paramount importance. In gratitude for the help of others and to contribute to the game community I present this workshop in the hope that it will help 'raise the bar' of game sound, in particular those working with Conitec's 3D Game Studio.

This workshop is presented in two parts: First, a 'cookbook' of scripts and 'how-to' instructions for using the sound and music features of 3DGS. For those who desire to know a bit more about how game sound works and for some additional helpful information, there follows a set of 'Fine Manuals' or appendices, including a primer on digital audio, sound design, and mixed-mode CDs.

Premises

This workshop is based on the following premises:

1: The reader has a basic understanding of how the various elements of 3DGS, the A5 engine, WDL coding, and the WED editor work. If not, you should read the tutorials that come with 3DGS and build a couple of practice levels before diving into the material included here. Note that while almost all of the scripts will work with A4, certain functions, such as CD audio, require A5 commercial or better.

2: This workshop is intended for those who have little or no experience with game audio, i.e. beginners. All of the examples in this workshop were created using a computer that is normally used for graphic design, with common software and hardware. YES, there certainly are more advanced ways of doing things than what are represented here, but that would be the topic of a future workshop...

3: It is assumed that as authors of 'intellectual property', we as game designers respect the copyrights of others, including musicians. Please make sure that if you use someone else's music or sound design you have permission.

So let's begin...

2) A5 Sound: First Steps

Conitec's A5 engine, included with 3D Game Studio, uses three types of sound formats, detailed descriptions of which are included in the appendices.

Windows wave (.wav) -
Digital audio files encoded in the Windows native format.

Midi (.mid) -
Music sequencer data that is sent to the computer's sound card.

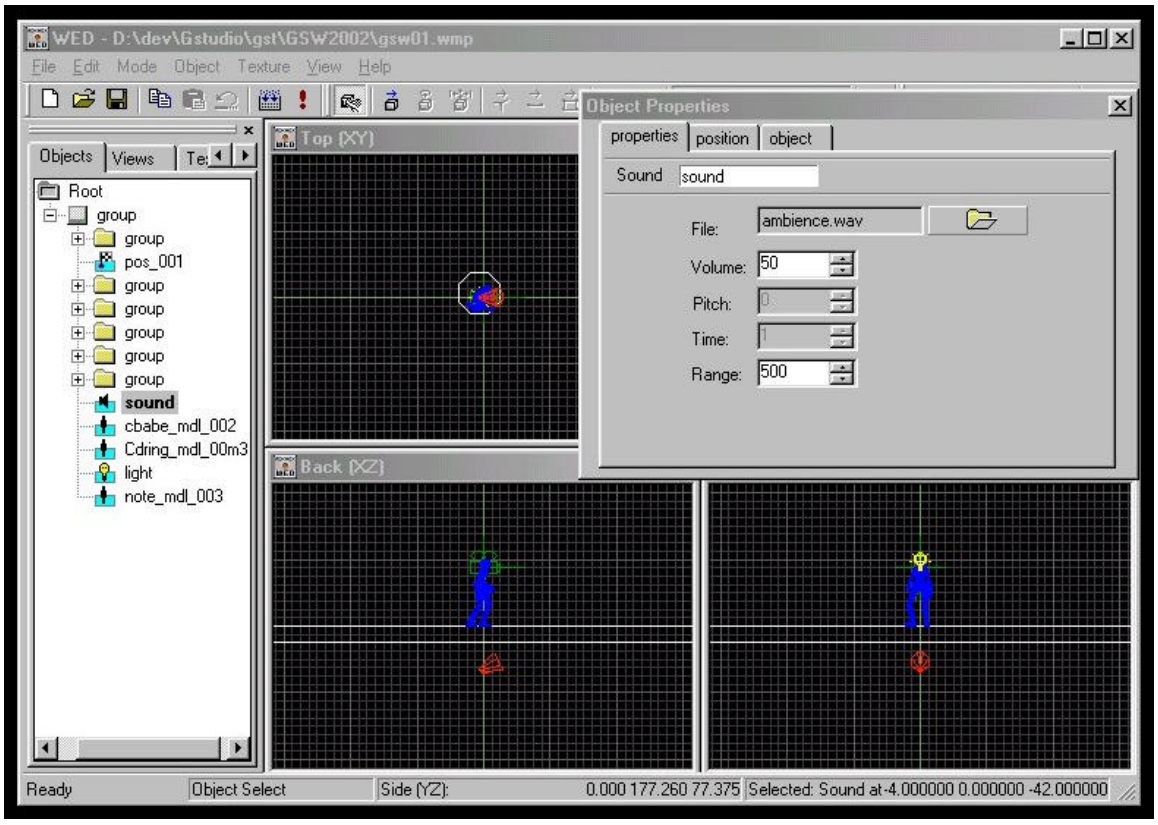
CD
CD audio in the 'Red-book' format, i.e. music Compact Disc.

Test level - gsw01.wmp

Let's begin by unzipping the workshop files into a new directory called /gsw. Next, run WED and open the example level gsw01.wmp, included with this workshop. If you wish to build your own level, build a square room with some lights, an actor, a player start, and a camera position. Save the level as gsw.wmp, then build using the 'test map' option. If you run the level now, it'll be quite dull. Let's begin to enhance it with sound.

Sound in WED Editor

The most basic way of adding sound to a level is to do so using WED. Use the 'Option/Add Sound' menu to select ambience.wav. Remember that the 'Add Sound' menu will only list sounds in the current level's directory. If you wish to add other sounds' use the 'Object/Load Entity' command. Once you've added the sound you will find a gramophone-looking icon for it. Move it so that it is centered in the room a little bit below the floor. Now, right click on the sound and you'll see the 'Sound Properties' dialogue box. The box shows the file name, and values for volume (1-100) and range (in quants). Note that the Pitch, Time, and Ambient values are grayed out - they're not yet implemented. Set the volume to 50, and the range to 1000.



Build the level using the 'update entities' option and run it. You'll hear a general ambient sound that gets quieter as you walk away from the center of the room.

3) Next Step: Using WDL to add sound

Now let's use WDL code to add sound to our level. Create a new file called sound.wdl and include it in your level's main .wdl file after the other includes. You will find an example with this tutorial, and it is included in gsw01.wdl. First we must define the sounds so that the A5 engine knows what they are.

The first line of code we write defines a name for a wave file:

```
sound = wave <fader01a.wav>;//define the sound
```

Next we add a variable that defines a handle for the file, so that we can manipulate it :

```
var_nsave wavehandle;//defines a 'handle' variable for our sound
```

Now we write an action for our sound event:

```
ACTION soundplay//
```

```
{
	wavehandle = snd_play(wave,50);/*plays the sound and assigns the
sound to the variable "wavehandle", which allows further manipulation or
prematurely stopping of that sound.*/
	waitt(250);//plays it for 250 frame cycles
	stop_sound(wavehandle);/*stops the sound that has the handle
'wavehandle'*/
}
```

How it works: Whenever A5 plays a .wav sound, it *'Returns a handle number which allows further manipulation or prematurely stopping the sound. The handle is valid as long as this sound plays.'*

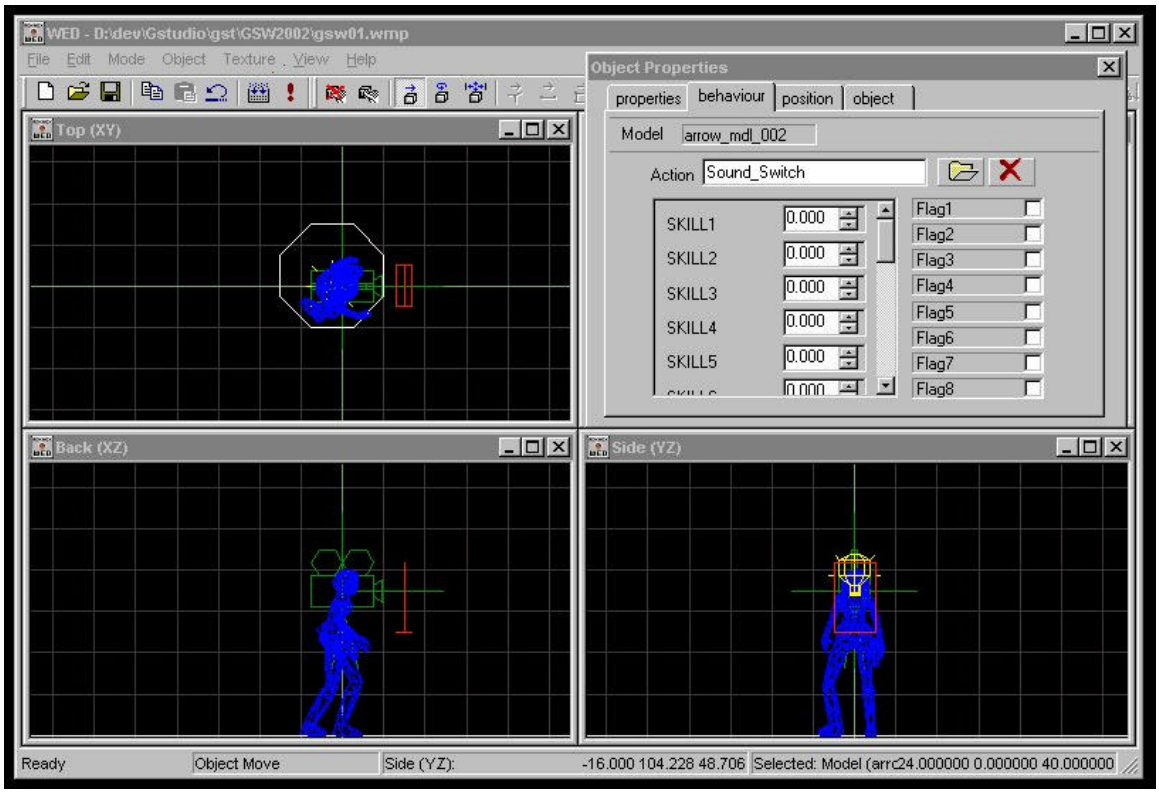
This rudimentary action doesn't add much more functionality than adding a sound in WED except that it stops after a predetermined interval. Let's add a trigger to make it more useful in game. We'll begin by modifying our soundplay action by adding an underscore to its name so that it doesn't show up in WED:

```
ACTION _soundplay//the sound action
{
	wavehandle = snd_play(wave,50,0);/*plays the sound and assigns the
sound to the variable "wavehandle", which allows further manipulation or
prematurely stopping of that sound.*/
	waitt(250);//plays it for 250 frame cycles
	snd_stop(wavehandle);/*stops the sound that has the handle
'wavehandle'*/
}
```

Now we write an action that triggers the sound:

```
ACTION Sound_Switch// assign this action to the switch
{
	MY.ENABLE_SCAN = ON; /*when you walk up to the switch and press
[space], it... */
	MY.EVENT = _soundplay; // ...starts the sound
	wait (1);
}
```

So, insert the arrow.pcx sprite in the level, and assign the Sound_Switch action to it using the 'properties' dialogue box in WED, rebuild the level, and walk your player model up to it. Press the spacebar and the sound will play.



Nice, but boring. Let's use the sound's handle in conjunction with the A5 command snd_tune to make it more interesting.

Begin by defining an additional sound:

```
SOUND boom,<explosin.wav>;//defines the explosion sound
```

As before we add a variable that defines a handle for the file, and three new variables that we can manipulate:

```
var_nsave boom_tune;//defines a 'handle' variable for our sound
var boom_vol = 100;//defines a variable for our sound's volume
var boom_pitch =100;//defines a variable for our sound's pitch
var boom_pan = -100;//defines a variable for our sound's pan
```

These new variables define the *volume* (loudness), *pitch* (frequency) and *balance* (left/right panning) of our sound. We add one more variable which will act as a loop counter:

```
var boomcount = 20;//defines a counter variable
```

Now the fun part:

```
ACTION _boomtune//
{
      my.event = null; //keeps the switch from triggering twice
```

```
      boom_tune = snd_play(wave,100,-100);//plays the sound & assigns
the handle
      While(boomcount > 0)//here's the effect loop
      {
            snd_tune (boom_tune,boom_vol,boom_pitch,boom_pan);/*change
the volume, pitch and pan of the sound*/
            boom_vol -= 5;//for each step of the loop, it gets quieter
            boom_pitch -= 2;//...and higher in pitch...
            boom_pan += 10;//... and pans from left to right
            boomcount -= 1;//next step in the loop
            wait(10);//wait 1, then return

      }
      snd_stop(boom_tune);/*when the loop has run its course it then
stops it*/
      snd_play(boom,100,100);// and plays the explosion sound...
}

ACTION boom_Switch// assign this action to the switch
      {
      MY.ENABLE_SCAN = ON; /*when you walk up to the switch and press
[space], it... */
      MY.EVENT = _boomtune; // ...starts the sound
      wait (1);
      }
```

Assign to the arrow.pcx sprite and you'll hear the sound start in your left speaker, then gradually move to the right as it gets quieter and lower in pitch. When it gets to the left side, you hear an explosion.

How it works: The action boom_Switch works exactly as the previous example - press the spacebar and the _boomtune action is called. This time our action begins with a ***my****.event* = ***null****;* command to keep the switch from double triggering. Then we use **snd_play** as before, this time setting the third variable, pan, to -100 so that it plays only in the left speaker. As before, we then assign a handle to the sound. Next comes a while loop that begins with the A5 command **snd_tune**. This command has four variables: the sound name, its volume, frequency (pitch), and balance (left/right panning). In our action it begins with the values we set beforehand. Note that we set these variables to the same values as we did in the **snd_play** command, so that we don't notice the change the first cycle through the loop. The following three commands change the values of our **snd_tune** variables, and the fourth is used to advance us to the next step of our loop. We let the sound play a little while with a **wait** command and then go to the next step of the loop. When the loop has run its course, the sound is stopped with a **snd_stop** command and then we play an explosion sound. now we can manipulate the sound with actions, but the sound plays in our speakers in the same place regardless of where we are in the level.

Other variations on snd_play

A5 has two commands that are similar to **snd_play**, **snd_loop** and **snd_playfile**. The first, **snd_loop**, plays the file indefinitely until a **snd_stop** command is issued. This is useful for playing music or ambient sounds in a level. The second,

snd_playfile, plays a large mono file once. This would be useful for playing a narration during a cutscene.

4) Attaching Sound To Entities

In order to make sound follow entities in the game, A5 has a command called **ent_playsound**. Let's write an action to use this command. First, we'll define a new handle for this action:

```
var_nsave wavehandle2;//defines a 'handle' variable for our sound
```

Now we'll modify _soundplay to use the ent_playsound command. The three variables define which entity the sound is attached to, which sound it plays, and its volume. Note that the *'range (in quants) of an entity sound is 10 times its volume. The volume may be set to over 100 to give a huge range; the sound itself, of course, is not played louder than with volume 100. The sound will use the sound hardware's stereo and 3D capabilities, and will produce a Doppler shift if the entity was moving towards the camera at the time the instruction was executed. Please note that the entity must already exist at the time the sound is played - that means if it was **create**d,* (it must have) *happened at least 1 frame cycle before.'*

```
ACTION _soundplay2
{
      wavehandle2 = ent_playsound(my,wave,100);/*plays the sound at the
entity's position and assigns the resultant value to the handle
'wavehandle2'*/
      waitt(500);//plays it for 500 frame cycles
      snd_stop(wavehandle2);//then stops it
      }
}
```

Again we add an action that triggers the sound:

```
ACTION Sound_Switch2// assign this action to the switch
{
      MY.ENABLE_SCAN = ON; /*when you walk up to the switch and press
[space],*/
      MY.EVENT = _soundplay2; //it  starts the sound
      wait (1);
}
```

Assign Sound_Switch2 to the arrow.pcx sprite, and rebuild the level. Walk your player up to it, press the spacebar, and then move around. Notice that the sound plays in the stereo spectrum relative to your position and that of the sprite entity. Now our level entities have spatial sounds, which help to add a sense of depth.

Let's try a more complex set of actions using the **snd_tune** command with **ent_playsound** as we did before, this time linking the sound to an object's behavior. Begin by building a rectangular map object in WED, build it as an 'entity', and load it into your level. I've included such an entity called trigger.wmb in the second level of this workshop, named gsw02.wmp. Place the arrow.pcx sprite as you did in the first example, and then place the trigger.wmb somewhere near it. Now in sound.wdl, we'll add a new series of actions.

Start by defining an entity that the sound will be attached to:

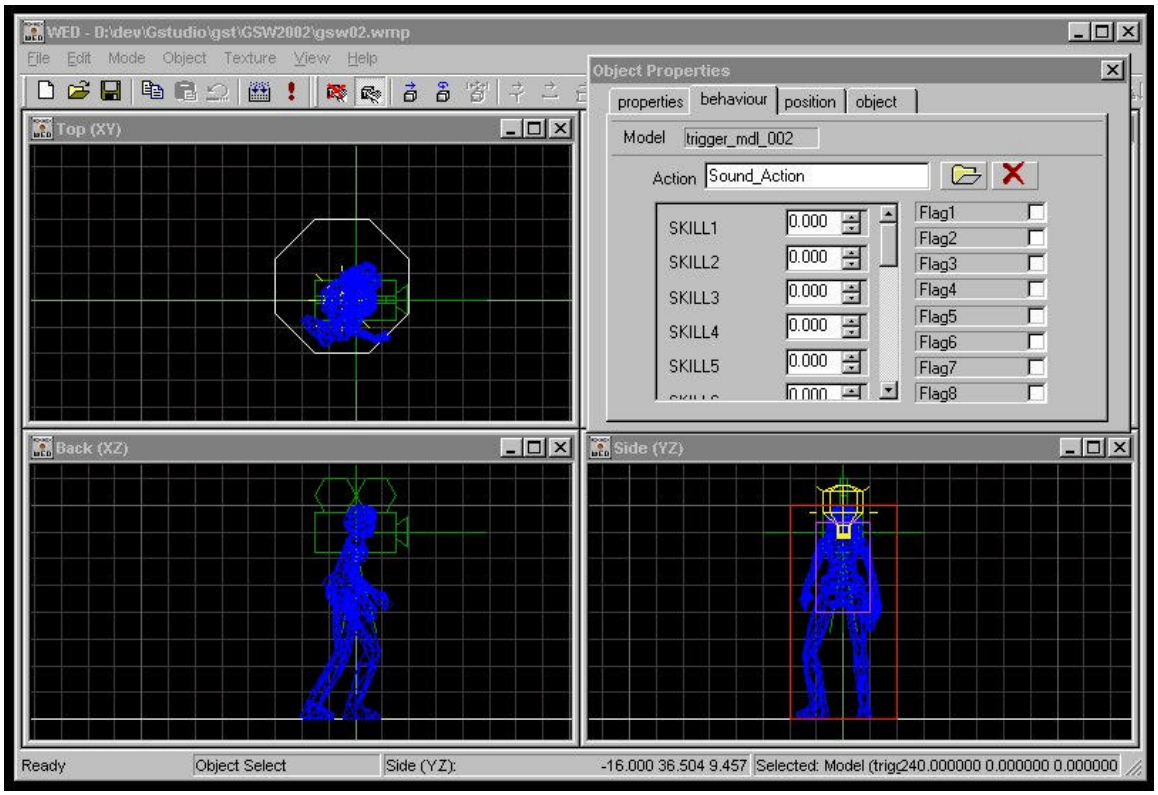
```
entity* sound_ent;
```

As before we add a variable that defines a handle for the file, and variables that we can manipulate:

```
var_nsave wavehandle3;//defines a 'handle' variable for our sound
var wavevol = 100;//defines a variable for our sound's volume
var wavepitch = 0;//defines a variable for our sound's pitch
```

Our first action will be used to define which object will be effected, in this case trigger.wmb:

```
ACTION Sound_Action// assign this action to the object to be effected
{
      sound_ent = me;
}
```

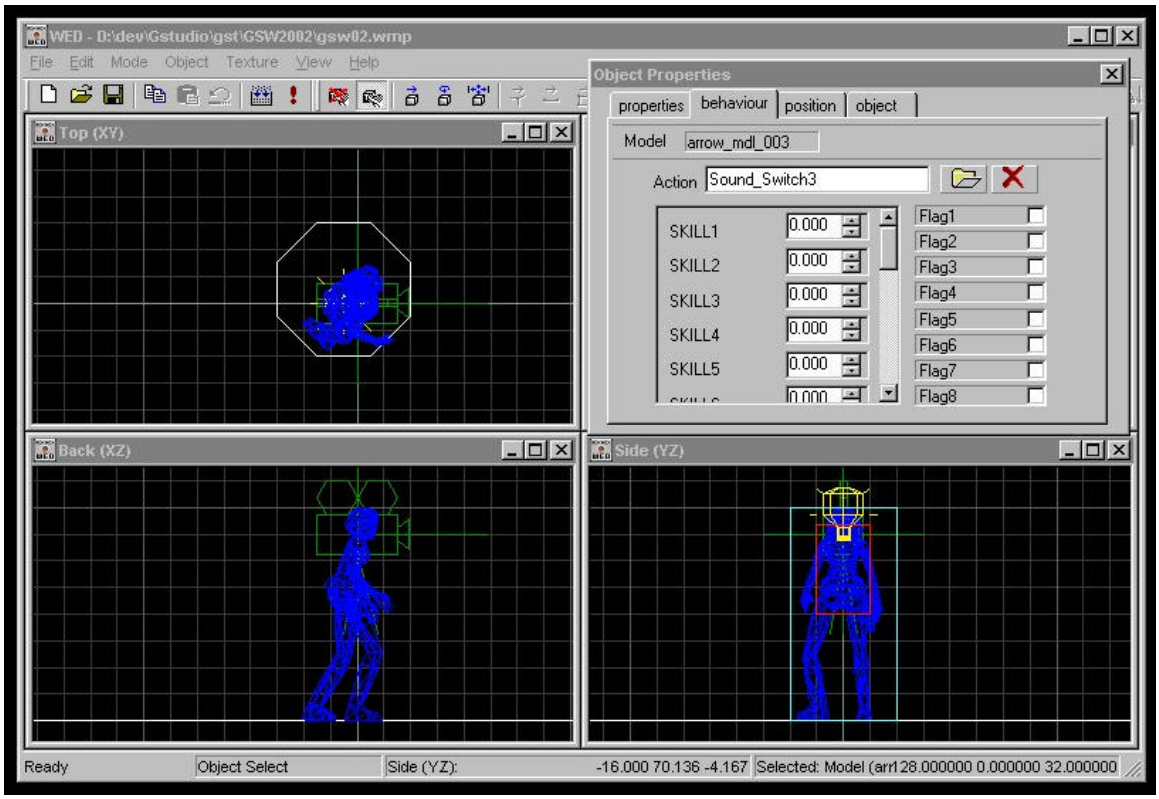


The second action is similar to _boomtune, except that instead of a counter we will change the snd_tune variable according to the position of the entity:

```
ACTION _soundplay3//the animated sound action
      {
      my.event = null; //won't trigger twice
      wavehandle3 = ent_playsound (sound_ent,wave,100);/*plays the sound
at the entity's position and assigns the resultant handle to
wavehandle3*/
      while (sound_ent.z > -20) //while the block is above the floor
      {
              sound_ent.z -= 4*time; // lower lower it
              snd_tune (wavehandle3,wavevol,wavepitch,0);/*change the
volume and pitch of the sound*/
              wavevol -= 1;//as the block descends, it gets quieter
              wavepitch -= 1;//..and higher in pitch
              wait(10);//wait 1, then return
      }
      snd_stop(wavehandle3);/*when the loop has run its course it then
stops it*/
      snd_play(boom,100,100);// and plays the explosion sound...
}
```

Lastly, we add the switch action which will be attached to arrow.pcx:

```
ACTION Sound_Switch3// assign this action to the switch
{
      MY.ENABLE_SCAN = ON; /*when you walk up to the switch and press
[space], it plays*/
      MY.EVENT = _soundplay3; // starts the sound
      wait (1);
}
```



How it works: Like before, once we walk up to the sprite and press the spacebar, the switch becomes inactive, the sound begins to play, and a handle is assigned. Then a loop starts where the z-axis position of trigger.wmb gets lower. With each step of the loop, the sound gets both quieter and higher in pitch. Note that the balance variable is left at 0; This is because we want the sound's balance to depend on our position relative to the entity. Changing the pan using **snd_tune** would override **ent_playsound**'s control over the balance.

5) Adding Music

We can add sound to our level by using three methods: MIDI, WAV, and CD-Audio. Let's review and discuss the pros and cons of each method.

MIDI

Typically music is played by a command in the *Main* function, and starts with the level. The office.wdl included with A5 is an example. A function is called in the *Main* function that starts the midi file playing:

```
// play a midi file
      start_song();
```

Further down in the file we find the function itself:

```
MUSIC testsong = <ribanna.mid>;

// play a song if volume was set at beginning
function start_song()
{
      wait(4);    /* wait until LOAD_INFO has loaded the last volume
setting*/
      if (midi_vol > 0) {
            play_song_once(testsong,66);
      }
}
////////////////////////////////////////////////////////////////////////
```

HOW IT WORKS: When the level is loaded, it calls the function **start_song**. The midi file *ribanna.mid* is first defined by the **MUSIC** command. The **start_song** function begins by allowing any existing information from a previously saved game, such as volume, to be loaded. Then it checks to see if a song is already playing by using an **if** command to determine whether the MIDI volume is 0 or not. If the **midi_vol** is 0, it means that nothing is playing. Then, a **play_song_once** command plays the MIDI file, at the volume given in the second variable, to its end and then stops. By using **play_song**, the song would loop indefinitely until a command sets the MIDI volume to 0.

We could at this point add some more control if we wanted:

```
////////////////////////////////////////////////////////////////////////
// stop the song
function stop_song()
{
      play_song(testsong,0);
}

ON_L = stop_song;
ON_K = start_song;

////////////////////////////////////////////////////////////////////////
//
```

By setting the MIDI volume to 0, it is stopped. By adding a couple of key assignments, we have a way of stopping and starting the music using MIDI.

Attaching midi music to entities

Instead of starting the music as the level starts by using the main function, we can attach music to an entity. Using a switch, we can attach a MIDI file to an entity:

```
////////////////////////////////////////////////////////////////////////
//
```

```
VAR midi_state = 0; // 0 for off, 1 for on
MUSIC mozart = <MozartGm.mid>;

ACTION _midi_event
{
         IF (midi_state == 0)// If Switch is off,
            {
                   play_song (mozart,66);//play the song
                   midi_state = 1;//sets the state of the switch to 'on'

            }
            ELSE //If Switch is on,
            {
                   play_song (mozart,0);/*stops the song - by setting
the volume to 0, we stop it.*/
                   midi_state = 0;//sets the state of the switch to
'off'
            }
}

ACTION midi_switch //starts the music - an' a one, an' a two, an a...
{
      MY.ENABLE_SCAN = ON; /*when you walk up to the switch and press
[space], */
      MY.EVENT = _midi_event; // it starts the midi file
}
////////////////////////////////////////////////////////////////////
//
```

HOW IT WORKS: Here we've used the variable *midi_state* to toggle a sound event on and off. The action first defines the variable **midi_state** and then a midi file (*Note: much like A5, W.A. Mozart was a German who found much favor amongst English speaking people. In the spirit of international collaboration, I have transcribed the opening of Mozart's symphony #5 in G minor as an example...*). The action uses an if/else to check the status of the switch and plays or stops the song accordingly.

PROS: Since a MIDI file is just a set of instructions to the hardware on the sound card, it has the smallest size and uses the least amount of resources, in fact it is negligible in most cases.

CONS: Most onboard synthesizers on sound cards are of only moderate quality, and the sound varies widely from machine to machine.

.WAV

Windows .wav audio files can be used for music in much the same way as MIDI. We can use a function called song_start_wav to use a .wav file in the same way :

First add this where song_start_midi was before:

```
// play a .wav file
      song_start_wav();
```

```
//
```

Next we'll paste this where the song_start_midi function went:

```
////////////////////////////////////////////////////////////////////////
//
//"Start .Wav sound" by faderboy
////////////////////////////////////////////////////////////////////////
//
sound wave = <GST01a.wav>;//defines the sound
var_nsave wavemusic;//defines a 'handle' variable for our sound

function song_start_wav()
{
      snd_loop(jam,60,0);//plays the sound
      wavemusic = result;/*assigns the value 'wave' to the handle
'wavehandle'*/
}

// stop the song
function stop_song_wav()
{
      snd_stop(wavemusic);/*stops the sound that has the handle
'wavehandle' */
}

ON_L = stop_song_wav;
ON_K = song_start_wav;

////////////////////////////////////////////////////////////////////////
//
```

In this example, we've used the *handle* to stop playing the .wav file. By using **snd_loop**, the song loops indefinitely. Were we to have used **snd_play** the music would have only played once.

PROS: A .wav file will sound exactly the same from machine to machine, since it is a native Windows file format. In addition, even on the most basic sound card, .wav files will sound good.

CONS: Wave audio uses large amounts of disc space and system resources, which can slow down game performance, especially if the files are large.

CD-audio

Similar to the way we used MIDI and .wav, now we'll use the CD-ROM to play music, pasting this:

```
// play a CD track
      start_song_cd();
```

and then this:

```
////////////////////////////////////////////////////////////////////////
//
// "Start CD" code by faderboy
function start_song_cd()/*this is the function called in the main
function*/

      {
      play_cd (1, 99);/*starts the CD at track 1, will play it through
track 99*/
      }

function stop_song_cd()// stops the disc

      {
      play_cd (0, 99);/*by setting the first variable to track 0, we
stop the CD player*/
      }
ON_J = stop_song_cd;

////////////////////////////////////////////////////////////////////////
```

HOW IT WORKS: The two variables after the **play_cd** command are the first and last tracks of the CD that will be played. If the first track is 0, the CD is stopped, which is how our switch works. Now we can play CD-audio in our level.

Controlling CD-audio with keystrokes, switches or a panel

When A5 is given a play_cd command, it gives a result variable *track* that corresponds to the currently playing track of the CD. By changing this variable and using the **cd_track** command, we can control which track is playing:

```
////////////////////////////////////////////////////////////////////////
//A5 CD Audio Control, v.01 by faderboy
////////////////////////////////////////////////////////////////////////

var track;/*defines the variable 'track', which is the currently playing
CD track /

ACTION CDplay// Start the CD player

      {
      play_cd (1, 99);/*starts the CD player at the first track, and
runs to track 99*/
      }

////////////////////////////////////////////////////////////////////////

ACTION CDstop// Stop the CD player

      {
      play_cd (0, 99);/*by setting the first variable to 0, we stop the
CD player*/
```

```
      }
////////////////////////////////////////////////////////////////////////

ACTION CDnext// Plays the next CD track
      {
      track = (cd_track + 1);/*makes the value of variable 'track' the
next highest,*/
      play_cd (track, 99);//so it plays the next song
      }
////////////////////////////////////////////////////////////////////////

ACTION CDprev// Plays the previous CD track
      {
      track = (cd_track - 1);/*makes the value of variable 'track' one
less,*/
      play_cd (track, 99);//so it plays the previous song
      }

ON_V = CDstop;
ON_B = CDplay;
ON_N = CDprev;
ON_M = CDnext;
////////////////////////////////////////////////////////////////////////
```

HOW IT WORKS: In addition to using **play_cd** to stop and play code as we did earlier, this code uses the **cd_track** command to enable the previous, next and current track display. At this point we can use the V,B,N, and M keys to control the CD player, or assign the actions to map entities.

We could also use a panel like this:

```
////////////////////////////////////////////////////////////////////////
//CD player Control Panel
//Defines the CD panel graphics and actions
////////////////////////////////////////////////////////////////////////

BMAP stop,<stop.bmp>;//a button for each function, stop
BMAP play,<play.bmp>;//play
BMAP prev,<prev.bmp>;//previous track
BMAP next,<next.bmp>;//next track
BMAP panelb,<panelb.bmp>;//this is the panel itself
FONT Arial24,<Arial24.bmp>,32,36;/*a groovy font to display the track
number  */

/*Define the popup CD Player Panel - Gracias to Panel Builder by
SerpentSoft*/
PANEL CDplayer
{
    BMAP panelb;
    POS_X 8;
    POS_Y 8;
    BUTTON 8, 8, stop, stop, stop, CDstop, NULL, NULL; /*enables the
stop button*/
    BUTTON 64, 8, play, play, play, CDplay, NULL, NULL;/*enables the
```

```
play button*/
    BUTTON 120, 8, prev, prev, prev, CDprev, NULL, NULL;/*enables the
prev button*/
    BUTTON 176, 8, next, next, next, CDnext, NULL, NULL;/*enables the
next button*/
    LAYER 1;
      FLAGS TRANSPARENT, REFRESH;
    DIGITS 230, 16, 2, Arial24 1, cd_track; /*here's where it displays
the current track*/
}

//This is how we display the CD Player Panel
ACTION CD_View {
 IF (CDplayer.VISIBLE == ON){//Is is already being displayed?
      SET CDplayer.VISIBLE,OFF;//If so, turn it off
      } ELSE {
      SET CDplayer.VISIBLE,ON;//If not already on, then display it
      }
}
////////////////////////////////////////////////////////////////////////

ON_C = CD_View;

////////////////////////////////////////////////////////////////////////
```

HOW IT WORKS: Pressing the C key will toggle the display of a CD Player Panel. The panel consists of four bitmap buttons: play, stop, next and previous, and a numerical "current track" display. In addition to using **play_cd** to stop and play code as we did earlier, this code uses the **cd_track** command to enable the previous, next and current track display.

PROS: CD audio is the standard format for recorded music, and is the best sounding format available for A5. In addition, playback of CD audio requires very little system resources.

CONS: Depending on the CD-ROM drive and/or system, there can be a lag when a CD is started, is playing or when a new track is selected. This can be avoided by starting a CD track when the level starts and looping it until the player reaches the next level. This is also highly dependent on the end user having a CD-ROM drive with its audio output connected internally to the soundcard. If it is not, they won't hear music.

6) Sound In Action

By synchronizing and triggering sound effects with other code, we can enhance the overall immersion of a level. Open the demo level biowere.wmp in WED then *biowere.wdl* and *fader_fx.wdl* in your favorite editor. Notice in the *main* function of biowere.wdl, a call is made to a function named **start_song**. Depending how you decide to implement music in your game, this function will start wave audio, MIDI or CD audio, by changing which type is commented out.

Build the level and run it. The first thing you'll notice is the flickering light. This is Wheiraucher's *'flickering light'* code, with sound added. If you observe the code in fader_fx.wdl you'll find that an additional skill is added with which we can define the general range of the sound effect. The variable *zaprange* is a random number multiplied by our predefined skill. Each time the loop cycles, *zaprange* is given a new value and the sound *zap* is played at the light's position with the random range.

As your character makes her way down the hall, she runs into a passable, transparent block that acts as a trigger. This trigger calls the action *_dialogue*, which in turn uses the **snd_play** command to play a dialogue message.

After you pass through the door, the first things you see are two panels, a radar and a scrolling text. These are simple looping sprite animations the first of which, the radar, has an **ent_playsound** instruction attached to it that plays each time the animation loop cycles. On the walls are three doors similar to the one you've just entered through. The one to the right is locked (it's the exit). The one directly in front opens but the hallway is blocked by fires, which will kill you if you walk through them. If you walk into the flames, your collision with them will trigger a 'burning' sound and subtract health.

If you flip the switch on the left door, it opens, revealing a hallway with a switch at the end. This door uses a modified version of Keebo's *'Q2 Door'* code. As the door moves, the sound travels with it, since it uses **ent_playsound**. This simple moving sound code makes the opening of the door more ominous.

After passing through the door, your character triggers another transparent invisible block, which calls the action *Jukebox* from biowere.wdl. *Jukebox* works by first stopping the music that was started in the *Main* function, and then playing a different piece. Using this approach, you can use the music to create a different mood, depending upon where the player is. For example, you could start with a slow, ambient music track, and when the player stumbles into a room full of monsters, play a fast techno track.

When you reach the end of the hall, you trigger the switch. Notice as the handle moves up the sound gets higher in pitch and softer in volume. When the handle has reached its end, the light begins to blink and each time it goes on it plays the *blip* sound. This switch triggers the *'Rain'* code from the tech demo, which in this case is used to simulate a fire sprinkler, which puts out the flames in the middle hallway. Run back to the middle door and you will see the water falling and the flames getting lower. You can also use the 'p' key to trigger this effect as well.

When the flames are gone, you can pass through to the adjacent room. The walkway takes you across a pool of acid, inside of which is the switch to open the exit door. Notice in WED that there is a gurgle sound playing under the block. If you try to swim or walk through it, your collision with the passable block will trigger a burning sound and subtract from your health. Your only way out is to jump from the walkway to the platform and trigger the switch. This switch calls the *_Acid_level* action which not only lowers the block but plays an additional instance of gurgle, which gets higher in pitch and louder as the block descends. Once the block has descended far enough you can jump down, trigger the last switch and run back to the central room. Here you'll find the last door opened, revealing the exit sign.

As you can see, these sound scripts are all rather simple, yet they add greatly to the overall immersion of your game. Feel free to hack and slash them into new effects. If you come up with something cool, please send it to me.

7) Digital Audio Primer

Now that we've explored the implementation of sound in 3D Game Studio, let's take a look at how digital audio works. Sound is converted into and out of electronic media via the use of transducers. Transducers are electronic devices that convert sound waves travelling through air into electronic signals and vice versa. The most common transducers are microphones and speakers.

Microphones

The two most common types of microphones are Dynamic and Condenser.

Dynamic mics are essentially an electromagnet attached to some sort of diaphragm. As sound waves collide with the diaphragm, they move the coil of the magnet, inducing and electronic signal analogous to the waves' amplitude and frequency.

xxx.jpg

In a similar fashion, Condenser mics consist of two parallel, electrically charged plates that, when excited by the movement of air, induce an electronic signal analogous to the waves' amplitude and frequency.

Speakers

Most speaker designs work in the opposite fashion of Dynamic mics: they consist of a diaphragm affixed to an electromagnet. An electronic signal is passed through the magnet, causing the diaphragm to push the air around it, generating a sound analogous to the electronic signal.

Then What?

Once sound is converted into an electronic signal, we can then store it on some form of media. In the past century these media were analog forms such as grooved vinyl discs and oxide coated tape. In this modern age, we store it in digital forms. In order to achieve this we must convert the Analog signal to a Digital one, which is done with an Analog to Digital Converter, or ADC. The ADC *samples* the *frequency* (pitch), *Amplitude* (loudness) and *time* of the incoming electronic signal and *encodes* or *modulates* the sample data into a stream of digital words, which can then be stored and manipulated by computers and other digital devices such as CD players and MiniDiscs (Diagram 4). This process is commonly referred to as *Pulse Code Modulation* (PCM).

The frequency at which the ADC samples the signal is called the *sample rate* or *sampling frequency.* Sample rate is analogous to dots per inch (DPI) in graphics - the higher the frequency, the greater the resolution. Likewise, *Bit Depth* or *Word Length* is analogous to graphics - the more bits in the digital word, the more information can be stored, the better the end result.

As a measure of reference, CD audio is Pulse Code Modulated at a frequency of 44.1k/second, with a 16 bit word length. This equates roughly to about 10 mB of data per minute in stereo.

MIDI

MIDI, or *Music Instrument Digital Interface* is a protocol developed by the now-defunct Sequential Circuits company in the early 1980's to enable music synthesizers to work together. It is a set of data instructions that contains information that a music device such as a synthesizer or a computer's sound card interprets. In its basic form, MIDI data consists of the following:

Note number
Note volume
Amplitude/Frequency modulation
Channel (from 1 to 16)
Patch (preset #)

These values range in each case from 1 to 127. It is important to understand that MIDI is not recorded sound - it is only a set of instructions that a device will interpret to generate a sound. Since they are only a set of data instructions, MIDI files are relatively small in size. Likewise, the sound generated is entirely dependent upon the quality of the hardware that receives it. Having only 128 increments of data also makes the resolution of MIDI files, especially volume, somewhat low in relation to other types of technology. The small size and universal nature of MIDI does, however, make up for it drawbacks.

8) Creating Sound and Music Content

Creating sound and music is an art that one can, as I have done, spend many years studying at a university and is much too broad in scope for a mere tutorial. As a professional music and sound producer, I can only give you, the reader, some advice and hints on how to make your sounds work better in 3D Game Studio.

Tools

Software

You will need certain tools to start. An audio editing package is essential for recording and editing sound effects and music. In addition you will need a music sequencing program with which to compose music and to create MIDI files. Fortunately in this day and age, both of these tools are bundled together. As is the case with graphics tools, every artist has an opinion which is best, and lots of people waste precious time arguing which is better. Ultimately, you as an artist should pick a set of tools that fits your creative style and your budget. Some of the more well known audio/music software packages include:

Cubase by Steinberg
Logic, by Emagic
ProTools by digidesign
Cakewalk, by Twelve Tone Systems

In addition there are some very good audio-only packages, which include:

Sound Forge, Vegas and Acid by Sonic Foundry
Vegas, by Sonic Foundry
Cool Edit, by Syntrillium
GoldWave, by GoldWave Inc.

Over the years, I have owned or operated all of these programs and can verify that each is well suited for game development. Just make sure that whatever package you use has the ability to access DirectX audio plugins, since these are indispensable tools to manipulate the sounds. There are, of course, other packages available for the Mac platform, but as we are discussing a directX game, I will only discuss Wintel here.

For the sake of this discussion I will be using Sonic Foundry's Sound Forge, Vegas and Acid. Sonic Foundry was one of the first developers to release a professional audio tools for the Wintel platform and is still one of the most popular.

Hardware

It seems there are new soundcards out every six months and keeping up with them is a full time job I don't want. Therefore, I am basing this tutorial around that bastion of PC audio, the Creative Labs SoundBlaster. Granted, there are better cards around,

but the Blaster is the most common of all cards and nearly everything is compatible with it. It is also the most common card among end users, so even if you don't use it for audio production, it is wise to have one around to check your work on the lowest common denominator. Other excellent alternatives for audio work are made by Event, Emagic, Stienberg, Turtle Beach, MidiMan, Mark of the Unicorn or Digidesign.

Raw Material

To prove that one doesn't need a professional studio to generate game sounds, I took a Mini-disc player and a cheap computer microphone into a machine shop. One word of advice: always ask permission when recording on someone else's' property. Neither Conitec nor I will be responsible if you are injured or incarcerated while gathering sounds.

After gathering some raw sounds, I plugged the output of the Mini-disc into the Line input of my Sound Blaster and recorded using Sound Forge.

If you prefer, you can also buy sound effects on CD or over the Internet. Always make sure that you comply with copyright laws when using someone else's sounds.

Once you have collected your sounds, there are a number of ways to optimize them for game audio:

Normalization: In layman's terms, when a computer normalizes a sound, it measures the loudest part of the file and then raises everything else in relationship to it by changing the digital values.

Compression/Limiting: Compression reduces the maximum values, and raises the minimum values of a sound according to a set threshold. Similarly, limiting lowers the high values.

Equalization: Equalization changes the frequency or tone of the sound. One important thing to remember while using Equalization is that *what you take out is as important as what you add.*

Reverb/Delay: By using reverb and delay, you can simulate the sound of the environment, i.e. a canyon, a small room, or a hotel lobby.

Just like graphics plugins, the best way to understand how audio plugins works is to experiment with them until you get the effect you desire.

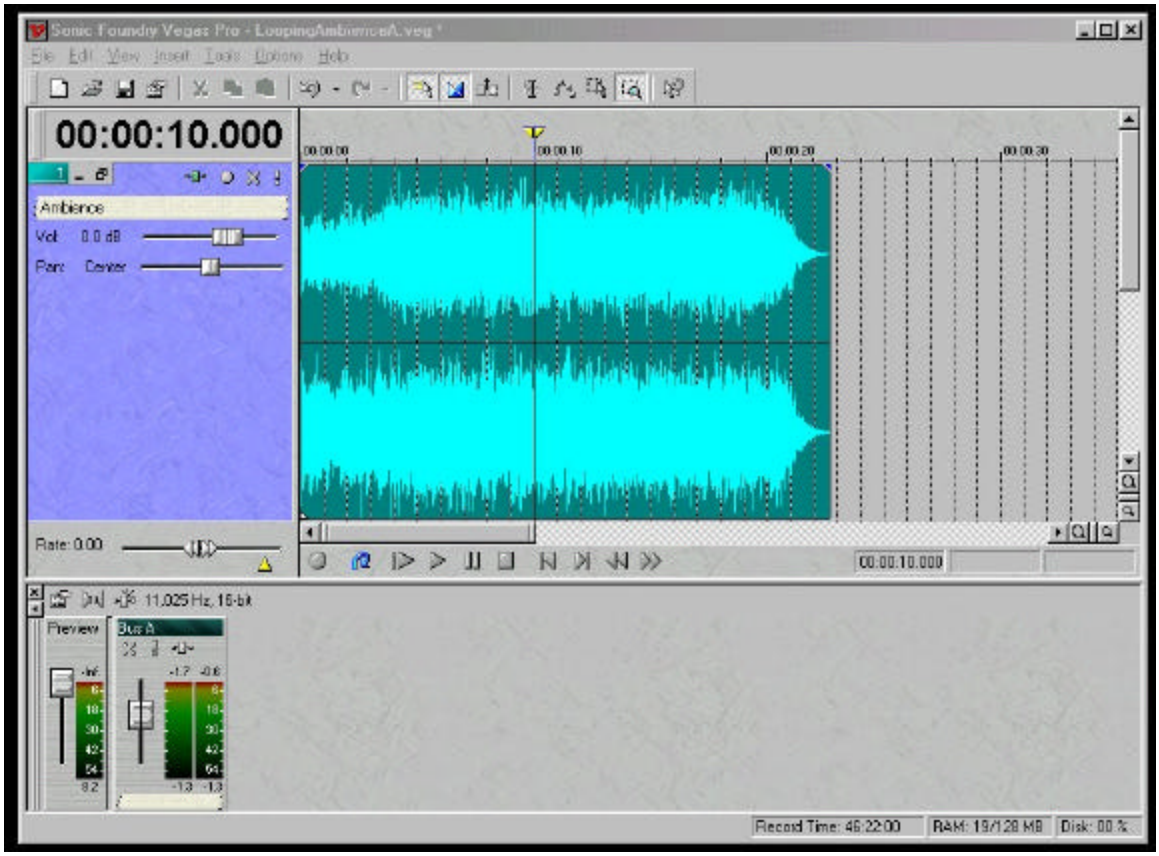
Tricks:

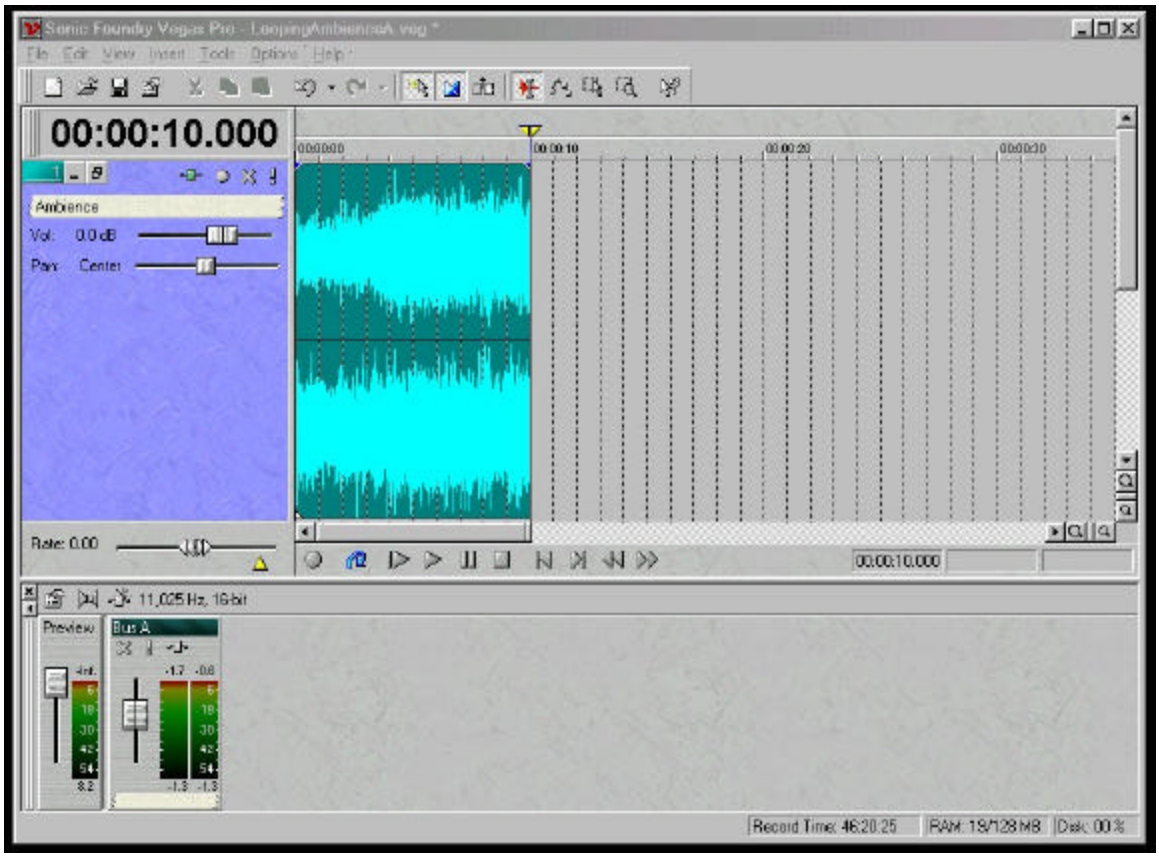
File Size: Always be aware that the higher the bit-depth and resolution of a file, the larger its size and the more it drains resources. Similarly, stereo sounds are twice as large. By using normalization and equalization, you can make your sound effects sound quite good even at 8-bit/11k.

Stereo/Mono: Sounds that will be spatial in nature, especially when used with **`ent_playsound`**, should be mono, since the engine will change their left/right balance.

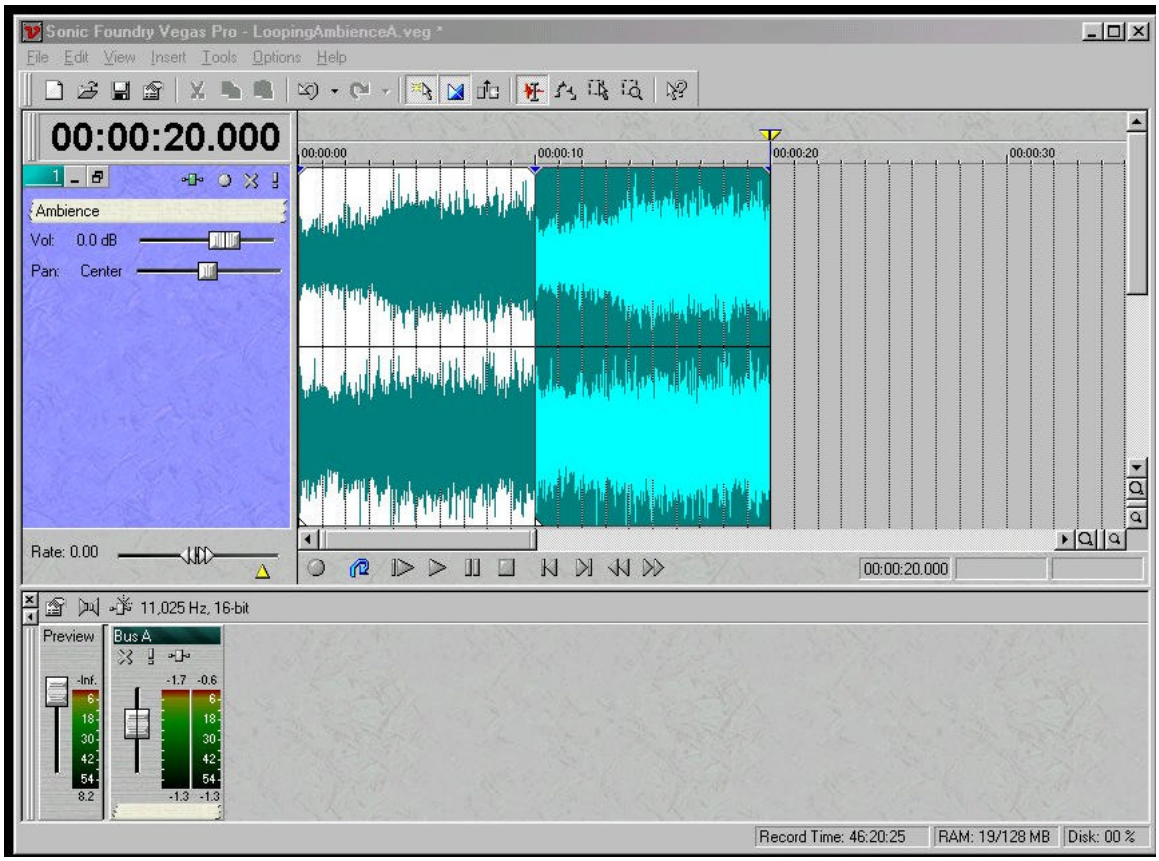
Looping: Just like graphics textures are tiled, sounds can be *looped* to conserve size. Here is a method of creating seamless loops. This example uses sonic foundry Vegas, but the technique works in any software.

First, select a 10-second piece of ambient sound and add it to a track.

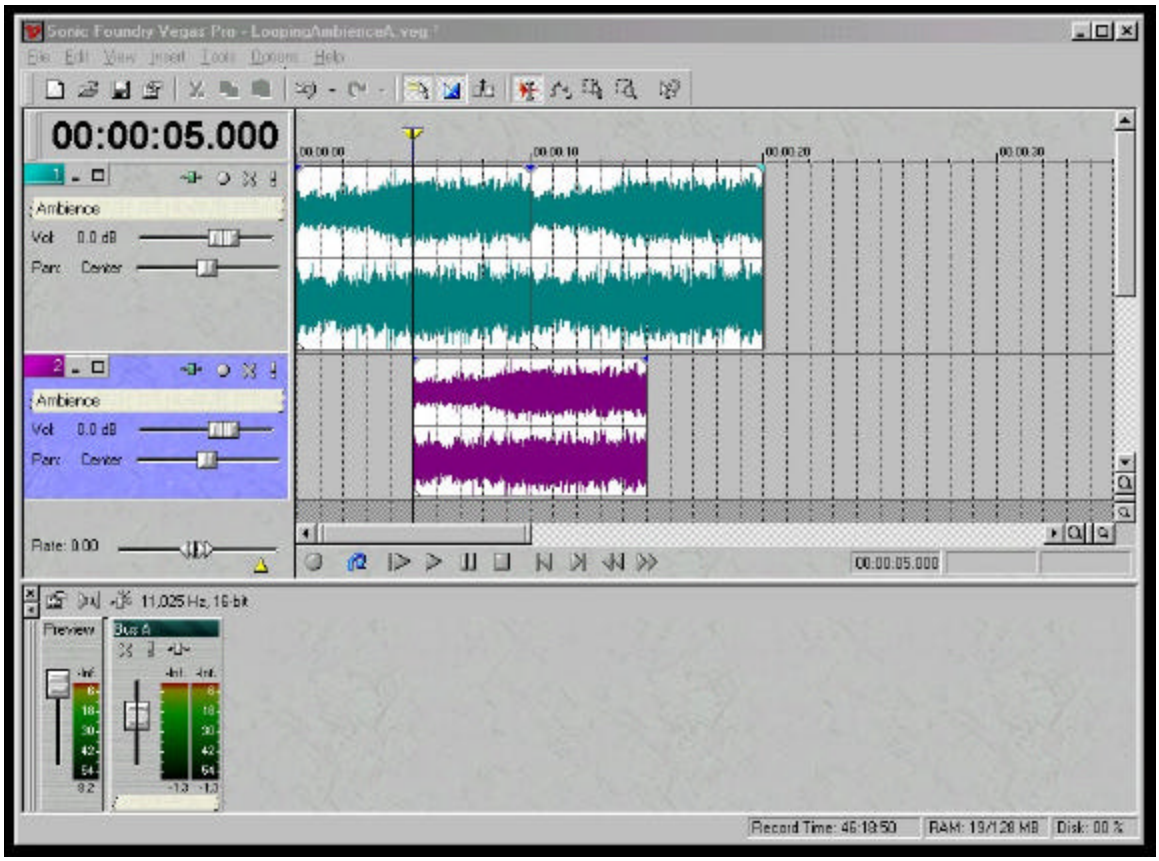




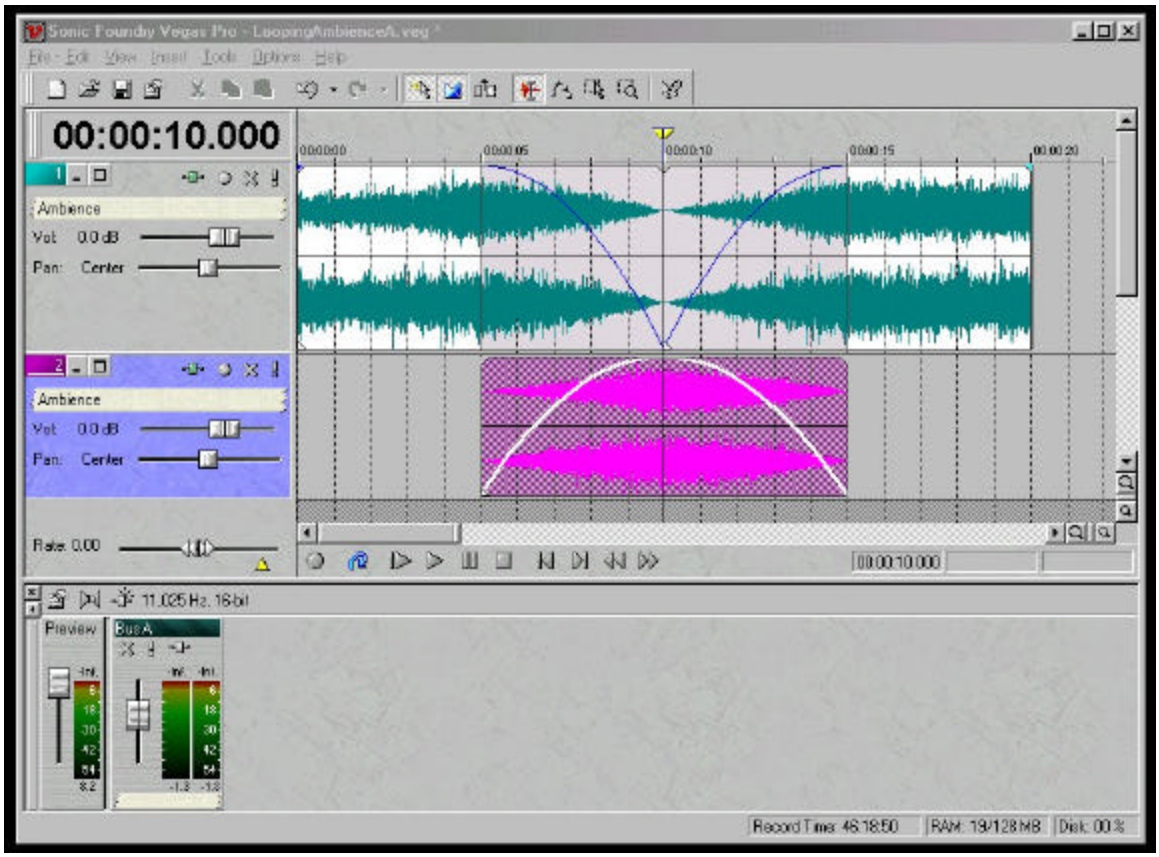
Now, copy it and paste it next to the first piece, giving you a 20-second track



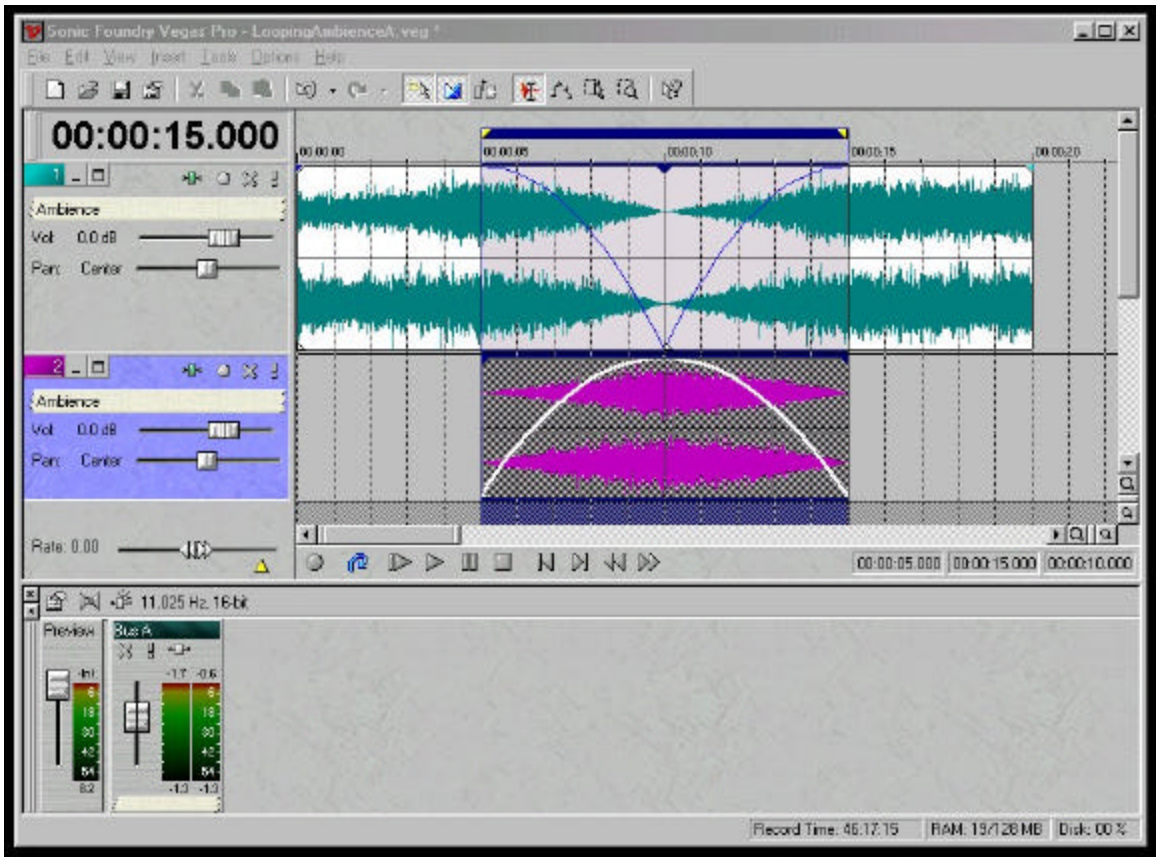
Next, paste the same piece on another track at 5 seconds:



Using the fade tool, fade the first piece out from 5 seconds to 10 seconds, then fade the second piece up from 10 seconds to 15 seconds. Fade the third piece up from 5 seconds to 10 seconds, then down from 10 seconds to 15 seconds

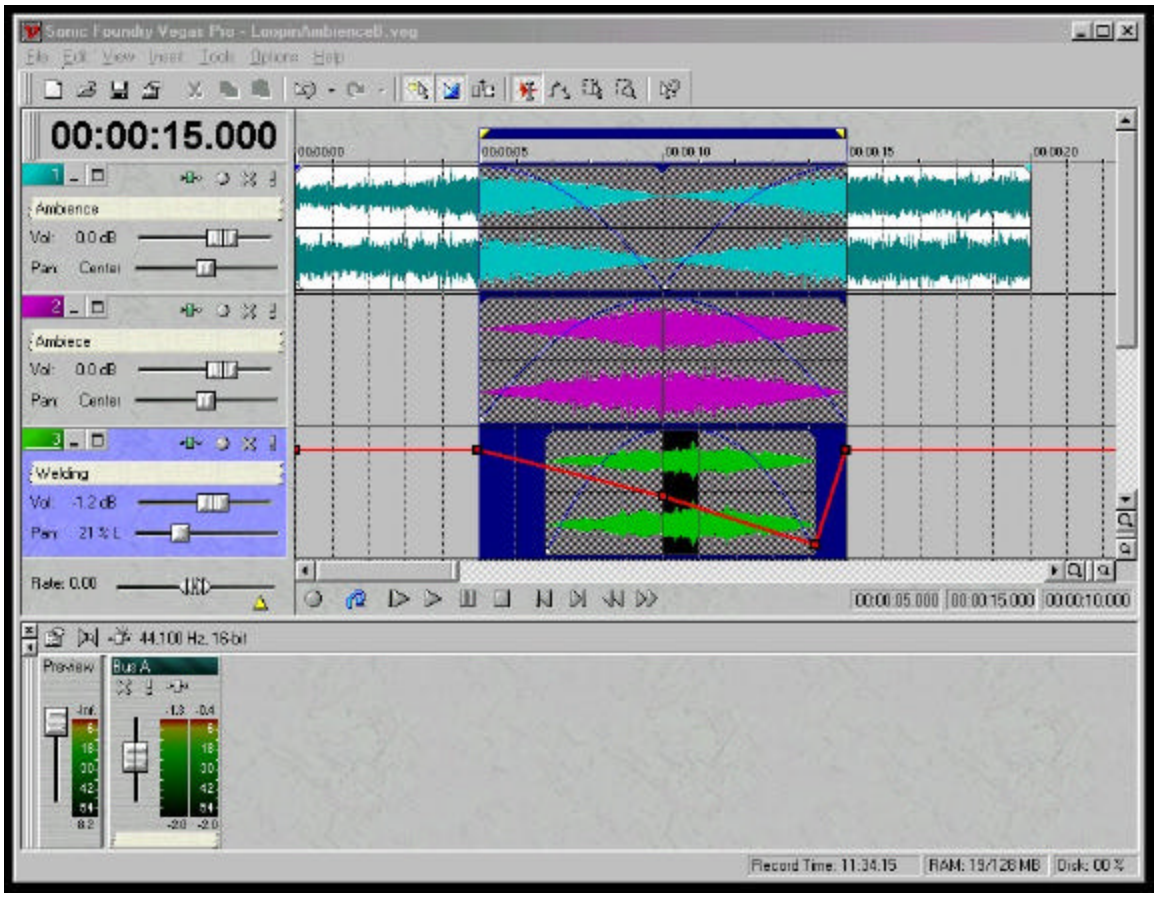


Now select a loop from 5 seconds to 15 seconds and play.

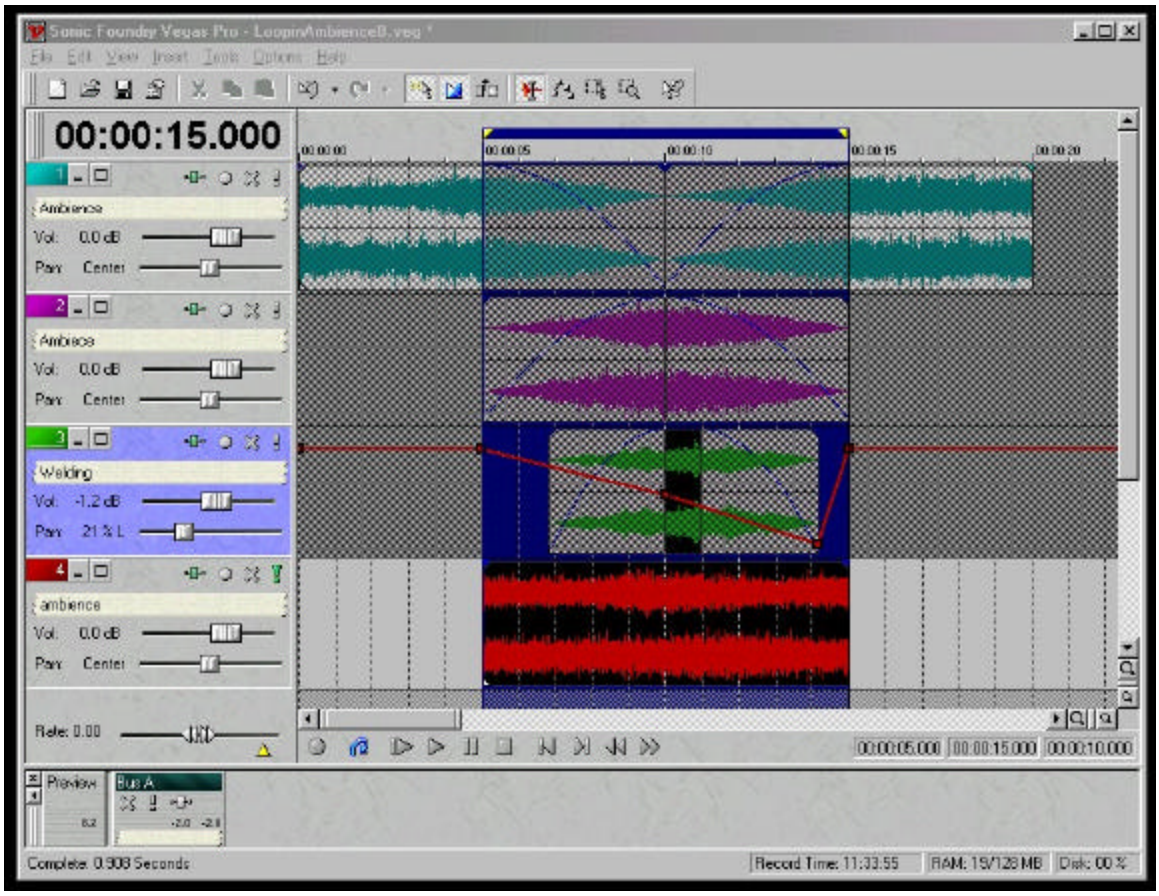


As the first piece fades out, the third piece fades in. At 10 seconds, the second piece fades in as the third piece fades out.

For added realism, you can add other sound effects as well:

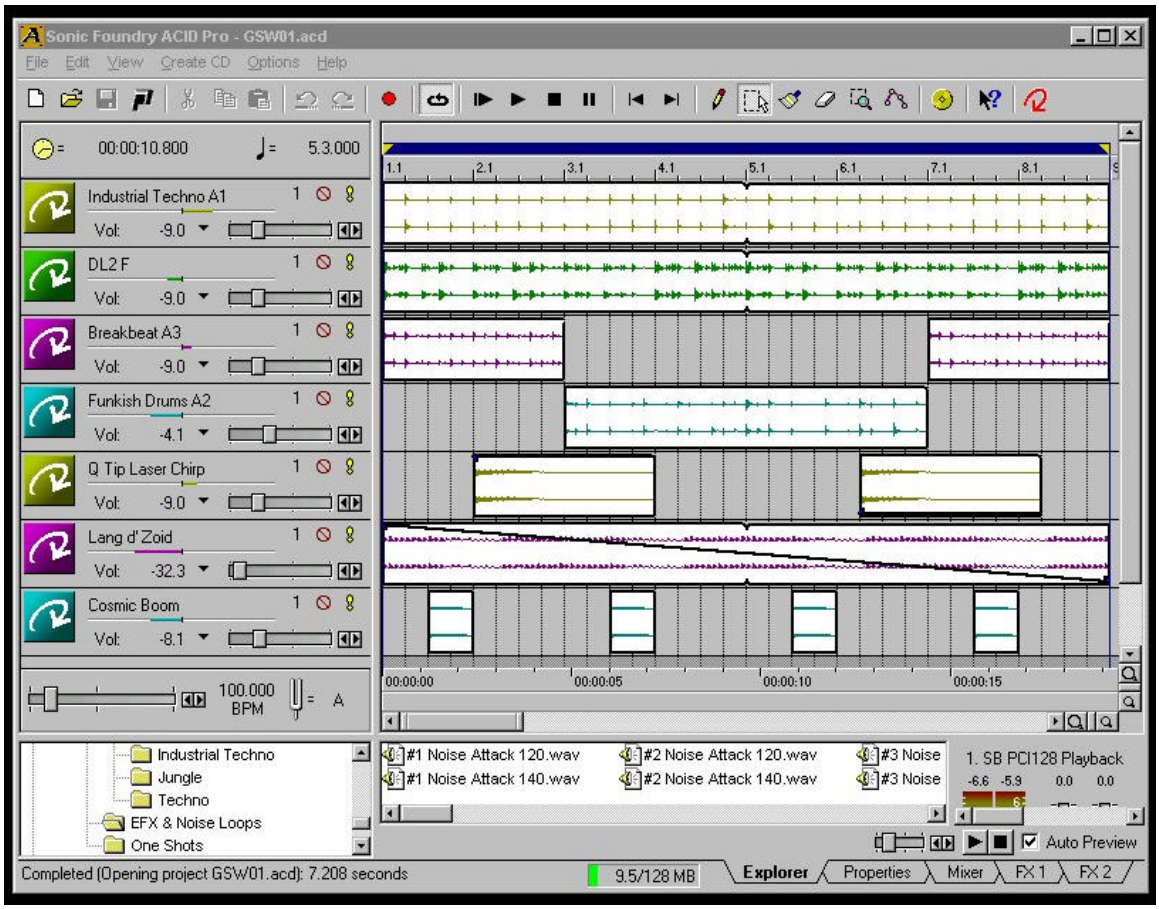


Mix the selection between 5 and 15 seconds to a new file, and you have a 10-second .wav file that loops seamlessly…



For you graphics guys, this technique is analogous to using *displace* to create tiling textures.

Looping Music: You can use Sonic Foundry Acid to create seamless music loops in a similar fashion. Just drag and drop some beats into an 8-bar loop and generate a file.



For more advanced discussion of these techniques, check out Sonic Foundry's website at http://www.sonicfoundry.com

9) Bibliography

GameStudio C-Script / WDL Tutorial & Manual
by J.C. Lotter, Conitec 2002

A Programmer's Guide To Sound
By Tim Kientzle , Addison-Wesley 1998

Game Design Secrets of the Sages
Edited by Marc Saltzman, Brady Publishing 2000

Home Recording For Musicians
By Craig Anderton, Amsco Publications 1996

10) Online Resources

Conitec, makers of 3D Game Studio
http://www.conitec.net/a4info.html

Faderboy's Audio Resources
http://www.faderboy.com

Gamasutra
http://www.gamasutra.com

Audio Engineering Society
http://www.AES.org

Game Music Home Page
http://www.lool.net/Game_Music/links.html

Partners in Rhyme – Royalty Free Music
http://www.partnersinrhyme.com/