# Audio

Audio is an essential element of all video games. In general, audio effects in games fall into two categories. The first category is background audio. This includes background music or ambient effects and is often used to bring atmosphere or emotion to different portions of the game. The second category is sound effects. Sound effects are useful for all sorts of purposes, from notifying users of game actions to hearing the footfalls of your hero character. Usually, sound effects represent a specific action, triggered either by the user or by the game itself. Such sound effects are often thought of as an audio cue.

One important difference between these two types of audio is how you control them. Sound effects or cues cannot be stopped or have their volume adjusted once they have started; therefore, cues are generally short. On the other hand, background audio can be started and stopped at will. These capabilities are useful for stopping the background track completely and starting another one.

## The Audio Support Project

This project is identical to the previous one with the MyGame and the BlueLevel scenes, you can move the front rectangle left or right with the arrow keys, the intersection with the left boundary triggers the loading of the other scene, and, the Q key quits the game. However, in this version, each scene plays background music and triggers a brief audio cue when the left/right arrow key is pressed. You can also notice the volume variations for each type of the audio clip. The implementation of this project also reinforces the concept of loading and unloading of external resources and the audio clips themselves. You can see an example of this project running in Figure 4-6. The source code to this project is defined in the chapter4/4.6.audio\_support folder.



Figure 4-6. Running the Audio Support project with both scenes

The controls of the project are as follows:

* Left/right arrow key: Moves the front rectangle left and right to increase and decrease the volume of the background music
* Q key: Quits the game

The goals of the project are as follows:

* To add audio support to the resource management system
* To provide an interface to play audio for games
* To optimize and facilitate resource sharing with reference counts of individual resources

You can find the following audio files in the chapter4\4.6.audio\_support\assets\sounds folder:

* bg\_clip.mp3
* blue\_level\_cue.wav
* my\_game\_cue.wav

Notice that the audio files are in two formats, mp3 and wav. While both are supported, audio files of these formats should be used with care. Files in .mp3 format are compressed and are suitable for storing longer durations of audio content, for example, for background music. Files in .wav format are uncompressed and should contain only very short audio snippet, for example, for storing cue effects.

### Define an Audio Resource Module

While audio and text files are completely different, from the perspective of your game engine implementation, there are two important similarities. First, both are external resources and thus will be implemented similarly as engine components in the src/engine/resources folder. Second, both involve standardized file formats with well-defined API utilities. The Web Audio API will be used for the actual retrieving and playing of sound files. Even though this API offers vast capabilities, in the interests of focusing on the rest of the game engine development, only basic supports for background audio and effect cues are discussed.

**Note** Interested readers can learn more about the Web Audio API from www.w3.org/TR/webaudio/.

**Note** The latest policy for some browsers, including Chrome, is that audio will not be allowed to play until first interaction from the user. This means, the context creation will result in initial warning from Chrome (output to runtime browser console). The audio will only be played after user input (e.g., mouse click, or keyboard events).

1. In the src/engine/resources folder, create a new file and name it audio.js. This file will implement the audio component module.
2. Similar to the implementations of text and xml modules, import the core resource management functionality from resource\_map.

"use strict";

import \* as map from "../core/resource\_map.js";

// functions from resource\_map

let unload = map.unload;

let has = map.has;

1. Declare variables to maintain references to the Web Audio context, background music, and to control volumes.

let mAudioContext = null;

let mBackgroundAudio = null;

// volume control support

let mBackgroundGain = null; // background volume

let mCueGain = null; // cue/special effects volume

let mMasterGain = null; // overall/master volume

let kDefaultInitGain = 0.1;

1. Define the init() function to create and store a reference to the Web Audio context in mAudioContext, and initialize the audio volume gain controls for the background, cue, and a master that affects both. In all cases, volume gain of a 0 corresponds to no audio and 1 means maximum strength.

function init() {

try {

let AudioContext = window.AudioContext || window.webkitAudioContext;

mAudioContext = new AudioContext();

// connect Master volume control

mMasterGain = mAudioContext.createGain();

mMasterGain.connect(mAudioContext.destination);

// set default Master volume

mMasterGain.gain.value = kDefaultInitGain;

// connect Background volume control

mBackgroundGain = mAudioContext.createGain();

mBackgroundGain.connect(mMasterGain);

// set default Background volume

mBackgroundGain.gain.value = 1.0;

// connect Cuevolume control

mCueGain = mAudioContext.createGain();

mCueGain.connect(mMasterGain);

// set default Cue volume

mCueGain.gain.value = 1.0;

} catch (e) {

throw new Error("Web Audio is not supported. Engine initialization failed.");

}

}

1. Define the decode and parsing functions, and call the resource\_map loadDecodeParse() function to load an audio file.

function decodeResource(data) { return data.arrayBuffer(); }

function parseResource(data) { return mAudioContext.decodeAudioData(data); }

function load(path) {

return map.loadDecodeParse(path, decodeResource, parseResource);

}

1. Define the playCue() function to play the entire duration of an audio clip with proper volume control. This function uses the audio file path as a resource name to find the loaded asset from the resource\_map and then invokes the Web Audio API to play the audio clip. Notice that no reference to the source variable is kept, and thus once started, there is no way to stop the corresponding audio clip. A game should call this function to play short snippets of audio clips as cues.

function playCue(path, volume) {

let source = mAudioContext.createBufferSource();

source.buffer = map.get(path);

source.start(0);

// volume support for cue

source.connect(mCueGain);

mCueGain.gain.value = volume;

}

1. Define the functionality to play, stop, query, and control the volume of the background music. In this case, the mBackgroundAudio variable keeps a reference to the currently playing audio, and thus it is possible to stop and controls the volume of the clip.

function playBackground(path, volume) {

if (has(path)) {

stopBackground();

mBackgroundAudio = mAudioContext.createBufferSource();

mBackgroundAudio.buffer = map.get(path);

mBackgroundAudio.loop = true;

mBackgroundAudio.start(0);

// connect volume accordingly

mBackgroundAudio.connect(mBackgroundGain);

setBackgroundVolume(volume);

}

}

function setBackgroundVolume(volume) {

if (mBackgroundGain !== null) {

mBackgroundGain.gain.value = volume;

}

}

function incBackgroundVolume(increment) {

if (mBackgroundGain !== null) {

mBackgroundGain.gain.value += increment;

// need this since volume increases when negative

if (mBackgroundGain.gain.value < 0) {

setBackgroundVolume(0);

}

}

}

1. Define functions for controlling the master volume, which in turn, adjusts the volumes of both the cue and the background music.

function setMasterVolume(volume) {

if (mMasterGain !== null) {

mMasterGain.gain.value = volume;

}

}

function incMasterVolume(increment) {

if (mMasterGain !== null) {

mMasterGain.gain.value += increment;

// need this since volume increases when negative

if (mMasterGain.gain.value < 0) {

mMasterGain.gain.value = 0;

}

}

}

1. Define a clearnUp() function to release the allocated HTML5 resources.

function cleanUp() {

mAudioContext.close();

mAudioContext = null;

}

1. Remember to export the functions from this module.

export {init, cleanUp,

has, load, unload,

playCue,

playBackground, stopBackground, isBackgroundPlaying,

setBackgroundVolume, incBackgroundVolume,

setMasterVolume, incMasterVolume

}

### Export the Audio Module to the Client

Edit the index.js file to import from audio.js, initialize and cleanup the module accordingly, and to export to the client.

// … identical to previous code

import \* as aduio from "./resources/audio.js";

// … identical to previous code

function init(htmlCanvasID) {

glSys.init(htmlCanvasID);

vertexBuffer.init();

shaderResources.init();

input.init();

audio.init();

}

function cleanUp() {

loop.cleanUp();

audio.cleanUp();

input.cleanUp();

shaderResources.cleanUp();

vertexBuffer.cleanUp();

glSys.cleanUp();

}

// … identical to previous code

export default {

// resource support

audio, text, xml

// … identical to previous code

}

### Testing the Audio Component

To test the audio component, you must copy the necessary audio files into your game project. Create a new folder in the assets folder and name it sounds. Copy the bg\_clip.mp3, blue\_level\_cue.wav, and my\_game\_cue.wav files into the sounds folder. You will now need to update the MyGame and BlueLevel implementations to load and use these audio resources.

#### Change MyGame.js

Update MyGame scene to load the audio clips, play background audio, and cue the player when the arrow keys are pressed.

1. Declare constant file paths to the audio files in the constructor. Recall that these file paths are used as resource names for loading, storage, and retrieval. Declaring these as constants for later reference is a good software engineering practice.

constructor() {

super();

// audio clips: supports both mp3 and wav formats

this.mBackgroundAudio = "assets/sounds/bg\_clip.mp3";

this.mCue = "assets/sounds/my\_game\_cue.wav";

// … identical to previous code

}

1. Request the loading of audio clips in the load() function, and make sure to define the corresponding unload() function. Notice the unloading of background music is preceded by stopping the music. This is true in general that unloading must be preceded by halting all operations based on a resource.

load() {

// loads the audios

engine.audio.load(this.mBackgroundAudio);

engine.audio.load(this.mCue);

}

unload() {

// Step A: Game loop not running, unload all assets

// stop the background audio

engine.audio.stopBackground();

// unload the scene resources

engine.audio.unload(this.mBackgroundAudio);

engine.audio.unload(this.mCue);

}

1. Start the background audio at the end of the init() function.

init() {

// … identical to previous code …

// now start the Background music ...

engine.audio.playBackground(this.mBackgroundAudio, 1.0);

}

1. In the update() function, cue the players when the right and left arrow keys are pressed, and increases and decreases the background music.

update() {

// … identical to previous code …

// Support hero movements

if (engine.input.isKeyPressed(engine.input.keys.Right)) {

engine.audio.playCue(this.mCue, 0.5);

engine.audio.incBackgroundVolume(0.05);

xform.incXPosBy(deltaX);

if (xform.getXPos() > 30) { // this is the right-bound of the window

xform.setPosition(12, 60);

}

}

if (engine.input.isKeyPressed(engine.input.keys.Left)) {

engine.audio.playCue(this.mCue, 1.5);

engine.audio.incBackgroundVolume(-0.05);

xform.incXPosBy(-deltaX);

if (xform.getXPos() < 11) { // this is the left-bound of the window

this.next();

}

}

// … identical to previous code …

}

#### Change BlueLevel.js

The changes to the BlueLevel scene are similar to those of the MyGame scene but with a different audio cue.

1. In the BlueLevel constructor, add the following path names to the audio resources.

constructor() {

super();

// audio clips: supports both mp3 and wav formats

this.mBackgroundAudio = "assets/sounds/bg\_clip.mp3";

this.mCue = "assets/sounds/blue\_level\_cue.wav";

// … identical to previous code …

}

1. Modify the load() and unload() functions for the audio clips.

load() {

engine.xml.load(this.mSceneFile);

engine.audio.load(this.mBackgroundAudio);

engine.audio.load(this.mCue);

}

unload() {

// stop the background audio

engine.audio.stopBackground();

// unload the scene flie and loaded resources

engine.xml.unload(this.mSceneFile);

engine.audio.unload(this.mBackgroundAudio);

engine.audio.unload(this.mCue);

}

1. In the same manner as MyGame, start the background audio in the init() function and cue the player when the left and right keys are pressed in the update() function. Notice that in this case, the audio cues are played with different volume settings.

init() {

// … identical to previous code …

// now start the Background music ...

engine.audio.playBackground(this.mBackgroundAudio, 0.5);

}

update() {

// … identical to previous code

/// Move right and swap ovre

if (engine.input.isKeyPressed(engine.input.keys.Right)) {

engine.audio.playCue(this.mCue, 0.5);

xform.incXPosBy(deltaX);

if (xform.getXPos() > 30) { // this is the right-bound of the window

xform.setPosition(12, 60);

}

}

// Step A: test for white square movement

if (engine.input.isKeyPressed(engine.input.keys.Left)) {

engine.audio.playCue(this.mCue, 1.0);

xform.incXPosBy(-deltaX);

if (xform.getXPos() < 11) { // this is the left-boundary

this.next(); // go back to my game

}

}

// … identical to previous code

}

You can now run the project and listen to the wonderful audio feedback. If you press and hold the arrow keys, there will be many cues repeatedly played. In fact, there are so many cues echoed that the sound effects are blurred into an annoying blast. This serves as an excellent example illustrating the importance of using audio cues with care and ensuring each individual cue is nice and short. You can try tapping the arrow keys to listen to more distinct and pleasant sounding cues, or you can simply replace the isKeyPressed() function with the isKeyClicked() function and listen to each individual cue.

## Game Design Considerations

In this chapter, we discussed the game loop and the technical foundation contributing to the connection between what the player does and how the game responds. If a player selects a square that’s drawn on the screen and moves it from location A to location B by using the arrow keys (for example), you’d typically want that action to appear as a smooth motion beginning as soon as the arrow key is pressed, without stutters, delays, or noticeable lag. The game loop contributes significantly to what’s known as presence in game design; presence is the player’s ability to feel as if they’re connected to the game world and responsiveness plays a key role in making players feel connected. Presence is reinforced when actions in the real world (such as pressing arrow keys) seamlessly translate to responses in the game world (such as moving objects, flipping switches, jumping, and so on); presence is compromised when actions in the real world suffer translation errors such as delays and lag.

As mentioned in Chapter 1, effective game mechanic design can begin with just a few simple elements. By the time you’ve completed the *Keyboard Support* project in this chapter, for example, many of the pieces will already be in place to begin constructing game levels: you’ve provided players with the ability to manipulate two individual elements on the screen (the red and white squares), and all that remains in order to create a basic game loop is to design a causal chain using those elements that results in a new event when completed. Imagine the *Keyboard Support* project is your game: how might you use what’s available to create a causal chain? You might choose to play with the relationship between the squares, perhaps requiring that the red square be moved completely within the white square in order to unlock the next challenge; once the player successfully placed the red square in the white square the level would complete. This basic mechanic may not be quite enough on its own to create an engaging experience, but by including just a few of the other eight elements of game design (systems design, setting, visual design, music and audio, and the like) it’s possible to turn this one basic interaction into an almost infinite number of engaging experiences and create that sense of presence for players. You’ll add more game design elements to these exercises as you continue through subsequent chapters.

The *Resource Map* and *Shader Loads* project, the *Scene File* project, and the *Scene Objects* project are designed to help you begin thinking about architecting game designs from the ground-up for maximum efficiency so that problems such as asset loading delays that detract from the player’s sense of presence are minimized. As you begin designing games with multiple stages and levels and many assets, a resource management plan becomes essential. Understanding the limits of available memory and how to smartly load and unload assets can mean the difference between a great experience and a frustrating experience.

We experience the world through our senses, and our feeling of presence in games tends to be magnified as we include additional sensory inputs. The *Audio Support* project adds basic audio to our simple state-changing exercise from the *Scene Objects* project in the form of a constant background score to provide ambient mood and includes a distinct movement sound for each of the two areas. Compare the two experiences and consider how different they feel because of the presence of sound cues; although the visual and interaction experience is identical between the two, the *Audio Support* project begins to add some emotional cues because of the beat of the background score and the individual tones the rectangle makes as it moves. Audio is a powerful enhancement to interactive experiences and can dramatically increase a player’s sense of presence in game environments, and as you continue through the chapters you’ll explore how audio contributes to game design in more detail.

# Summary

In this chapter, you learned how several common components of a game engine come together. Starting with the ever-important game loop, you learned how it implements an input, update, and draw pattern in order to surpass human perception or trick our senses into believing that the system is continuous and running in real-time. This pattern is at the heart of any game engine and has everything needed in order to create basic games. You learned how full keyboard support can be implemented with flexibility and reusability to provide the engine with a reliable input component. Furthermore, you saw how a resource manager can be implemented to load files asynchronously and how scenes can be abstracted to support scenes being loaded from a file, which can drastically reduce duplication in the code. Lastly, you learned how audio support supplies the client with an interface to load and play both ambient background audio as well as audio cues.

These components separately have little in common but together make up the core fundamentals of nearly every game. As you implement these common core components into the game engine, the games that are created with the engine will not need to worry about the specifics of each component. Instead, the games programmer can focus on utilizing the functionality to hasten and streamline the development process. In the next chapter, you will learn how to create the illusion of an animation with external images.