

Playing Sound and Video

pyglet can load and play many audio and video formats, often with support for surround sound and video effects.

WAV and MP3 files are the most commonly supported across platforms. The formats a specific computer can play are determined by which of the following are available:

1. The built-in pyglet WAV file decoder (always available)
2. Platform-specific APIs and libraries
3. PyOgg
4. [FFmpeg](#) version 4, 5, 6, or 7.

Video is played into OpenGL textures, allowing real-time manipulation by applications. Examples include use in 3D environments or shader-based effects. To play video, [FFmpeg](#) must be installed.

Audio is played back with one of the following: OpenAL, XAudio2, DirectSound, or PulseAudio. Hardware-accelerated mixing is available on all of them. 3D positional audio and surround sound features are available on all back-ends other than PulseAudio.

Audio drivers

pyglet can use OpenAL, XAudio2, DirectSound, or PulseAudio to play sound. Only one driver can be used at a time, but the selection can be changed by altering the configuration and restarting the program.

The default driver preference order works well for most users. However, you may override it by setting a different preference sequence before the `pyglet.media` module is loaded. See [Choosing the audio driver](#) to learn more.

The available drivers depend on your operating system:

Windows	Mac OS X	Linux
OpenAL ²	OpenAL	OpenAL ²
DirectSound		
XAudio2		PulseAudio ¹

[1] The [PulseAudio](#) driver has limitations. For audio-intensive programs, consider using [OpenAL](#).

[2] (1,2) OpenAL does not come preinstalled on Windows and some Linux distributions.

Choosing the audio driver

The `'audio'` key of the `pyglet.options` dictionary specifies the audio driver preference order.

On import, the `pyglet.media` will try each entry from first to last until it either finds a working driver or runs out of entries. For example, the default is equivalent to setting the following value:

```
pyglet.options['audio'] = ('xaudio2', 'directsound', 'openal', 'pulse', 'silent')
```

You can also set a custom preference order. For example, we could add this line before importing the media module:

```
pyglet.options['audio'] = ('openal', 'pulse', 'xaudio2', 'directsound', 'silent')
```

It tells pyglet to try using the OpenAL driver first. If is not available, try Pulseaudio, XAudio2, and DirectSound in that order. If all else fails, no driver will be instantiated and the game will run silently.

The value for the `'audio'` key can be a list or tuple which contains one or more of the following strings:

String	Audio driver
'openal'	OpenAL
'directsound'	DirectSound
'xaudio2'	XAudio2
'pulse'	PulseAudio
'silent'	No audio output

You must set any custom `'audio'` preference order before importing `pyglet.media`. This can also be set through an environment variable; see [Environment settings](#).

The following sections describe the requirements and limitations of each audio driver.

XAudio2

XAudio2 is only available on Windows Vista and above and is the replacement of DirectSound. This provides hardware accelerated audio support for newer operating systems.

Note that in some stripped down versions of Windows 10, XAudio2 may not be available until the required DLL's are installed.

DirectSound

DirectSound is available only on Windows, and is installed by default. pyglet uses only DirectX 7 features. On Windows Vista, DirectSound does not support hardware audio mixing or surround sound.

OpenAL

The favored driver for Mac OS X, but also available on other systems.

This driver has the following advantages:

- Either preinstalled or easy to install on supported platforms.
- Implements features which may be absent from other drivers or OS-specific versions of their backing APIs.

Its main downsides are:

- Not guaranteed to be installed on platforms other than Mac OS X

- On recent Windows versions, the [XAudio2](#) and [DirectSound](#) backends may support more features.

Windows users can download an OpenAL implementation from openal.org or their sound device's manufacturer.

On Linux, the following apply:

- It can usually be installed through your distro's package manager.
- It may already be installed as a dependency of other packages.
- It lacks the limitations of the [PulseAudio](#) driver.

The commands below should install OpenAL on the most common Linux distros:

Common Linux Distro	Install Command
Ubuntu, Pop!_OS, Debian	<code>apt install libopenal1</code>
Arch, Manjaro	<code>pacman -S openal</code>
Fedora, Nobara	<code>dnf install openal-soft</code>

You may need to prefix these commands with either `sudo` or another command. Consult your distro's documentation for more information.

PulseAudio

This backend is almost always supported, but it has limited features.

If it fails to initialize, consult your distro's documentation to learn which supported audio backends you can install.

Missing features

Although PulseAudio can theoretically support advanced multi-channel audio, the pyglet driver does not. The following features will not work properly:

1. Positional audio: automatically changing the volume for individual audio channels based on the position of the sound source
2. Integration with surround sound

Switching to [OpenAL](#) should automatically enable them.

Supported media types

pyglet has included support for loading Wave (.wav) files, which are therefore guaranteed to work on all platforms. pyglet will also use various platform libraries and frameworks to support a limited amount of compressed audio types, without the need for FFmpeg. While FFmpeg supports a large array of formats and codecs, it may be an unnecessarily large dependency when only simple audio playback is needed.

These formats are supported natively under the following systems and codecs:

Windows Media Foundation

Supported on Windows operating systems.

The following are supported on **Windows Vista and above**:

- MP3
- WMA
- ASF
- SAMI/SMI

The following are also supported on **Windows 7 and above**:

- AAC/ADTS

The following is undocumented but known to work on **Windows 10**:

- FLAC

GStreamer

Supported on Linux operating systems that have the GStreamer installed. Please note that the associated Python packages for gobject & gst are also required. This varies by distribution, but will often already be installed along with GStreamer.

- MP3
- FLAC
- OGG
- M4A

CoreAudio

Supported on Mac operating systems.

- AAC
- AC3
- AIF
- AU
- CAF
- MP3
- M4A
- SND
- SD2

PyOgg

Supported on Windows, Linux, and Mac operating systems.

PyOgg is a lightweight Python library that provides Python bindings for Opus, Vorbis, and FLAC codecs.

If the PyOgg module is installed in your site packages, pygame will optionally detect and use it. Since not all operating systems can decode the same audio formats natively, it can often be a hassle to choose an audio format that is truly cross platform with a small footprint. This wrapper was created to help with that issue.

Supports the following formats:

- OGG
- FLAC
- OPUS

To install PyOgg, please see their [installation guide on readthedocs.io](#).

FFmpeg

Note

The most recent pygame release can use FFmpeg versions 4.X, 5.X, 6.X, and 7.X

See [FFmpeg installation](#) to learn more.

 [latest](#) ▼

FFmpeg is best when you need to support the maximum number of formats and encodings. It is also worth considering the following:

- Support for many formats and container types means large download size
- FFmpeg's compile options allow it to be built and used under [either the LGPL or GPL license](#)

See the following sections to learn more.

See [FFmpeg & licenses](#) to learn more.

Supported Formats


It is difficult to provide a complete list of FFmpeg's features due to the large number of audio and video codecs, options, and container formats it supports. Refer to [the FFmpeg documentation](#) for more information.

Known supported audio formats include:

- AU
- MP2
- MP3
- OGG/Vorbis
- WAV
- WMA

Known supported video formats include:

- AVI
- DivX
- H.263
- H.264
- MPEG
- MPEG-2
- OGG/Theora
- Xvid
- WMV
- Webm

The easiest way to check whether a file will load through FFmpeg is to try playing it through the `media_player.py` example. New releases of FFmpeg may fix bugs and add support for new formats.  [latest](#) ▼

FFmpeg & licenses

FFmpeg's code uses different licenses for different parts.

The core of the project uses a modified LGPL license. However, the GPL is used for certain optional parts. Using these components, as well as bundling FFmpeg binaries which include them, may require full GPL compliance. As a result, some organizations may restrict some or all use of FFmpeg.

pyglet's FFmpeg bindings do not rely on the optional GPL-licensed parts. Therefore, most projects should be free to use any license they choose for their own code as long as they use one of the following approaches:

- Require users to install FFmpeg themselves using either:
 - The [FFmpeg installation](#) section on this page
 - Custom instructions for a specific FFmpeg version
- Make FFmpeg optional as described at the end of the [FFmpeg installation](#) instructions
- Bundle an LGPL-only build of FFmpeg

See the following to learn more:


- [FFmpeg's license overview](#)
- The license documentation for your specific FFmpeg version:
 - [The FFmpeg 4.4 license breakdown](#)
 - [The FFmpeg 5.1 license breakdown](#)
 - [The FFmpeg 6.0 license breakdown](#)
 - [The FFmpeg 7.0 license breakdown](#)

FFmpeg installation

You can install FFmpeg for your platform by following the instructions found in the [FFmpeg download](#) page. You must choose the shared build for the targeted OS with the architecture similar to the Python interpreter.

All recent pyglet versions support FFmpeg 4.x.

- Support for version 5.X requires at least: 1.5.28.
- Support for version 6.X requires at least: 2.0.8.
- Support for version 7.X requires at least: 2.0.20.

Choose the correct architecture depending on the targeted **Python interpreter**. |  [latest](#) ▼
shipping your project with a 32 bits interpreter, you must download the 32 bits shared binaries.

On Windows, the usual error message when the wrong architecture was downloaded is:

```
WindowsError: [Error 193] %1 is not a valid Win32 application
```

Finally make sure you download the **shared** builds, not the static or the dev builds.

For Mac OS and Linux, the library is usually already installed system-wide. It may be easiest to list FFmpeg as a requirement for your project, and leave it up to the user to ensure that it is installed. For Windows users, it's not recommended to install the library in one of the windows sub-folders.

Instead we recommend to use the `pyglet.options` `search_local_libs`:

```
import pyglet
pyglet.options['search_local_libs'] = True
```

This will allow pyglet to find the FFmpeg binaries in the `lib` sub-folder located in your running script folder.

Another solution is to manipulate the environment variable. On Windows you can add the dll location to the PATH:


```
os.environ["PATH"] += "path/to/ffmpeg"
```

For Linux and Mac OS:

```
os.environ["LD_LIBRARY_PATH"] += ":" + "path/to/ffmpeg"
```

! Tip

Prevent crashes by checking for FFmpeg before loading media!

Call `pyglet.media.have_ffmpeg()` to check whether FFmpeg was detected correctly. If it returns `False`, you can take an appropriate action instead of crashing. Example  [latest](#) ▼

- Showing a helpful error in the GUI or console output

- Exiting gracefully after the the user clicks OK on a dialog
- Limiting the formats your project will attempt to load

If you still have issues with the FFmpeg not being recognized, try enabling the debug flags to see if any relevant information is output: `pyglet.options.debug_lib` and/or

`pyglet.options.debug_media`.

Loading media

Audio and video files are loaded in the same way, using the `pyglet.media.load()` function, providing a filename:

```
source = pyglet.media.load('explosion.wav')
```

If the media file is bundled with the application, consider using the `resource` module (see [Application resources](#)).

The result of loading a media file is a `Source` object. This object provides useful information about the type of media encoded in the file, and serves as an opaque object used for playing back the file (described in the next section).

The `load()` function will raise a `MediaException` if the format is unknown. `IOError` may also be raised if the file could not be read from disk. Future versions of pyglet will also support reading from arbitrary file-like objects, however a valid filename must currently be given.

The length of the media file is given by the `duration` property, which returns the media's length in seconds.

Audio metadata is provided in the source's `audio_format` attribute, which is `None` for silent videos. This metadata is not generally useful to applications. See the `AudioFormat` class documentation for details.

Video metadata is provided in the source's `video_format` attribute, which is `None` for audio files. It is recommended that this attribute is checked before attempting play back a video file – if a movie file has a readable audio track but unknown video format it will appear as an audio file.

You can use the video metadata, described in a `VideoFormat` object, to set up di-

video before beginning playback. The attributes are as follows:

Attribute	Description
<code>width</code> , <code>height</code>	Width and height of the video image, in pixels.
<code>sample_aspect</code>	The aspect ratio of each video pixel.

You must take care to apply the sample aspect ratio to the video image size for display purposes. The following code determines the display size for a given video format:

```
def get_video_size(width, height, sample_aspect):
    if sample_aspect > 1.:
        return width * sample_aspect, height
    elif sample_aspect < 1.:
        return width, height / sample_aspect
    else:
        return width, height
```

Media files are not normally read entirely from disk; instead, they are streamed into the decoder, and then into the audio buffers and video memory only when needed. This reduces the startup time of loading a file and reduces the memory requirements of the application.

However, there are times when it is desirable to completely decode an audio file in memory first. For example, a sound that will be played many times (such as a bullet or explosion) should only be decoded once. You can instruct pygame to completely decode an audio file into memory at load time:

```
explosion = pygame.media.load('explosion.wav', streaming=False)
```

The resulting source is an instance of `StaticSource`, which provides the same interface as a `StreamingSource`. You can also construct a `StaticSource` directly from an already- loaded `Source`:

```
explosion = pygame.media.StaticSource(pygame.media.load('explosion.wav'))
```

Audio Synthesis

In addition to loading audio files, the `pygame.media.synthesis` module is available for simple audio synthesis. There are several basic waveforms available, including:

 [latest](#) ▼

- `Sine`

- `Square`
- `Sawtooth`
- `Triangle`
- `WhiteNoise`
- `Silence`

These waveforms can be constructed by specifying a duration, frequency, and sample rate. At a minimum, a duration is required. For example:

```
sine = pygamelet.media.synthesis.Sine(3.0, frequency=440, sample_rate=44800)
```

For shaping the waveforms, several simple envelopes are available. These envelopes affect the amplitude (volume), and can make for more natural sounding tones. You first create an envelope instance, and then pass it into the constructor of any of the above waveforms. The same envelope instance can be passed to any number of waveforms, reducing duplicate code when creating multiple sounds. If no envelope is used, all waveforms will default to the `FlatEnvelope` of maximum amplitude, which essentially has no effect on the sound. Check the module documentation of each `Envelope` to see which parameters are available.



- `FlatEnvelope`
- `LinearDecayEnvelope`
- `ADSREnvelope`
- `TremoloEnvelope`

An example of creating an envelope and waveforms:

```
adsr = pygamelet.media.synthesis.ADSREnvelope(attack=0.05, decay=0.2, release=0.1)
saw = pygamelet.media.synthesis.Sawtooth(duration=1.0, frequency=220, envelope=adsr)
```

The waveforms you create with the synthesis module can be played like any other loaded sound. See the next sections for more detail on playback.

Simple audio playback

Many applications, especially games, need to play sounds in their entirety without needing to keep track of them. For example, a sound needs to be played when the player's `snare` `shin` explodes, but this sound never needs to have its volume adjusted, or be rewound  [latest](#)  interrupted.

pyglet provides a simple interface for this kind of use-case. Call the `play()` method of any `Source` to play it immediately and completely:

```
explosion = pyglet.media.load('explosion.wav', streaming=False)
explosion.play()
```

You can call `play()` on any `Source`, not just `StaticSource`.

The return value of `play()` is a `Player`, which can either be discarded, or retained to maintain control over the sound's playback.

Controlling playback

You can implement many functions common to a media player using the `Player` class. Use of this class is also necessary for video playback. There are no parameters to its construction:

```
player = pyglet.media.Player()
```

A player will play any source that is *queued* on it. Any number of sources can be queued on a single player, but once queued, a source can never be dequeued (until it is removed automatically once complete). The main use of this queueing mechanism is to facilitate “gapless” transitions between playback of media files.

The `queue()` method is used to queue a media on the player - a `StreamingSource` or a `StaticSource`. Either you pass one instance, or you can also pass an iterable of sources. This provides great flexibility. For instance, you could create a generator which takes care of the logic about what music to play:

```
def my_playlist():
    yield intro
    while game_is_running():
        yield main_theme
    yield ending

player.queue(my_playlist())
```

When the game ends, you will still need to call on the player:

```
player.next_source()
```

The generator will pass the `ending` media to the player.

A `StreamingSource` can only ever be queued on one player, and only once on that player.

`StaticSource` objects can be queued any number of times on any number of players. Recall that a `StaticSource` can be created by passing `streaming=False` to the `pyglet.media.load()` method.

In the following example, two sounds are queued onto a player:

```
player.queue(source1)
player.queue(source2)
```

Playback begins with the player's `play()` method is called:

```
player.play()
```

Standard controls for controlling playback are provided by these methods:

Method	Description
<code>play()</code>	Begin or resume playback of the current source.
<code>pause()</code>	Pause playback of the current source.
<code>next_source()</code>	Dequeue the current source and move to the next one immediately.
<code>seek()</code>	Seek to a specific time within the current source.

Note that there is no *stop* method. If you do not need to resume playback, simply pause playback and discard the player and source objects. Using the `next_source()` method does not guarantee gapless playback.

There are several properties that describe the player's current state:

Property	Description
<code>time</code>	The current playback position within the current source, in seconds. This is read-only (it cannot be set).
<code>playing</code>	True if the player is currently playing, False if there are no sources queued or the player is paused (see the <code>play()</code> and <code>pause()</code> methods).
<code>source</code>	A reference to the current source being played. This is read-only (but see the <code>queue()</code> method).
<code>volume</code>	The audio level, expressed as a float from 0 (mute) to 1 (normal volume). This can be set and read.
<code>loop</code>	<code>True</code> if the current source should be repeated when reaching the end. If set to <code>False</code> , playback will stop when the source ends.

Handling playback events

When a player reaches the end of the current source, an `on_eos()` (on end-of-source) event is dispatched. Players have a default handler for this event, which will either repeat the current source (if the `loop` attribute has been set to `True`), or move to the next queued source immediately. When there are no more queued sources, the `on_player_eos()` event is dispatched, and playback stops until another source is queued.

For loop control you can change the `loop` attribute at any time, but be aware that unless sufficient time is given for the future data to be decoded and buffered there may be a stutter or gap in playback. If set well in advance of the end of the source (say, several seconds), there will be no disruption.

The end-of-source behavior can be further customized by setting your own event handlers; see [Event dispatching & handling](#). You can either replace the default event handlers directly, or add an additional event as described in the reference. For example:

```
my_player.on_eos = my_player.pause
```

Gapless playback

To play back multiple similar sources without any audible gaps, `SourceGroup` is provided. A `SourceGroup` can only contain media sources with identical audio or video format. First create an instance of `SourceGroup`, and then add all desired additional sources with the `add()` method. Afterwards, you can queue the `SourceGroup` on a Player as if it was a single source.

Incorporating video

When a `Player` is playing back a source with video, use the `texture` property to obtain the video frame image. This can be used to display the current video image synchronised with the audio track, for example:

```
@window.event
def on_draw():
    player.texture.blit(0, 0)
```

The texture is an instance of `pyglet.image.Texture`, with an internal format of either `GL_TEXTURE_2D` or `GL_TEXTURE_RECTANGLE_ARB`. While the texture will typically be created only once and subsequently updated each frame, you should make no such assumption in your application – future versions of pyglet may use multiple texture objects.

Positional audio

pyglet includes features for positioning sound within a 3D space. This is particularly effective with a surround-sound setup, but is also applicable to stereo systems.

A `Player` in pyglet has an associated position in 3D space – that is, it is equivalent to an OpenAL “source”. The properties for setting these parameters are described in more detail in the API documentation; see for example `position` and `pitch`.

A “listener” object is provided by the audio driver. To obtain the listener for the current audio driver:

```
pyglet.media.get_audio_driver().get_listener()
```

This provides similar properties such as `position`, `forward_orientation` and `up_orientation` that describe the position of the user in 3D space.

Note that only mono sounds can be positioned. Stereo sounds will play back as normal, and only their volume and pitch properties will affect the sound.

Ticking the clock

If you are using pyglet's media libraries outside of a pyglet app (not using `pyglet.app.run()`) you will need to use some kind of loop to tick the pyglet clock periodically (perhaps every 200ms or so), otherwise you will have unintended side effects. Depending on the backend in use, this could mean only the first small sample of media will be played, or crashes due to internal resource exhaustion. At a minimum you will need to call:

```
pyglet.clock.tick()
```

If you wish to have a media source loop continuously (`player.loop = True`) you will also need to ensure Pyglet's events are dispatched inside your loop:

```
pyglet.app.platform_event_loop.dispatch_posted_events()
```

If you are inside a pyglet app then calling `pyglet.app.run()` takes care of all this for you.

[Watch Our Demo](#) video to see firsthand how to upgrade your site with end-to-end AI Search.

Ads by EthicalAds