Direct Python Audio/Video

Release 0.0.1

Vibrant Labs

May 02, 2022

CONTENTS:

CHAPTER

ONE

WHAT IS DIRECT PYTHON AUDIO/VIDEO?

Direct Python Audio/Video is a library wrapping certain functionalities of Pygame that aims to give users a very simple, no-nonsense, direct feeling experience with basic audio and video manipulation. This library features the ability to craft basic waveforms and play them, as well as manipulate pixels in an image using 24-bit hex color codes, using no more than a few calls from our library. We abstract away technical aspects of interfacing with audio and video devices such as the need to maintain an event loop, in favor of straightforward calls that feel intuitive and beginner friendly.

CHAPTER

TWO

INSTALLATION

This library may be installed by:

Cloning the repository:

```
>>> git clone https://github.com/The-krolik/dpav
```

Then navigating to the cloned dpav folder and running:

```
>>> pip install dpav
```

CHAPTER

THREE

AUDIO CLASS

The Audio class is intended to provide basic sound capabilities focused around playing a constant tone for a desired duration in seconds. It supports playing one sound at a time with a waveform: sin, square, noise, saw, or triangle.

To get started, there are three basic steps to play a tone:

1. Create an Audio class object
2. Call the play_sound method with a frequency and duration (in seconds)
3. Use the wait_for_sound_end. This maintains the process

Listing 1: Playing a sound

```
mySound = dpp.Audio()
frequency = 261
duration = 1

mySound.play_sound(frequency, duration)
mySound.wait_for_sound_end()
```

If using audio alongside the Window class or within a while loop, the wait_for_sound_end method is unnecessary.

Listing 2: Using the play_sound inside a while loop

```
mySound = dpp.Audio()
frequency = 261
duration = 1

while window.is_open():
    mySound.play_sound(frequency, duration)
```

The utility function get_note_from_string takes a music note, such as "C", as a string and returns the frequency

Listing 3: Using the utility function: get_note_from_string

```
mySound = dpp.Audio()
frequency = dpp.get_note_from_string("C", 0)
duration = 1

mySound.play_sound(frequency, duration)
mySound.wait_for_sound_end()
```

CHAPTER

FOUR

VBUFFER CLASS

The VBuffer class operates as a 2-dimensional array of hex color values. This is the main data structure used for visualization within the Window class.

4.1 Initialization

Listing 1: VBuffer initialization with dimensions 1920x1080

```
vbuffer = dpp.VBuffer((1920,1080))
```

Listing 2: VBuffer initialization with numpy array

```
arr = np.zeros((1920,1080))
vbuffer = dpp.VBuffer(arr)
```

Listing 3: VBuffer default initializaion provides dimensions 800x600

```
vbuffer = dpp.VBuffer()
```

4.2 Modification

Listing 4: Changing color of pixel to red at location: x=30, y=50

```
red = 0xFF0000
vbuffer[30,50] = red
```

Listing 5: Changing row 30 to red

```
red = 0xFF0000
vbuffer[30,:] = red
```

Listing 6: Fill vbuffer object with color red

```
red = 0xFF0000
vbuffer.fill(red)
```

Listing 7: Clear vbuffer object with color red

```
vbuffer.clear()
```

CHAPTER

FIVE

WINDOW CLASS

The Window class is an abstraction of the PyGame library's display and event handling. It is closely tied to the VBuffer class, using VBuffer objects as the primary data structure to hold the current image to display. An understanding of the VBuffer class may not be required for simple projects, such as those with static displays, but is recommended nonetheless, especially for more complicated use cases. Currently, only one window may be active at a time.

5.1 Initialization

Only one instance of the window class is needed throughout the lifetime of the program. Initialization of the object may be done in one of three ways, based upon the argument passed, or lack thereof. Passing a VBuffer object is the preferred method of initialization, however a 2-dimensional numpy array is also accepted, which will create the VBuffer for you. If neither are provided, the Window will create a default VBuffer with dimensions: (800,600).

Listing 1: VBuffer initialization

```
vbuffer = dpp.VBuffer((1920,1080))
window = dpp.Window(vbuffer)
```

Listing 2: Numpy array initialization

```
vbuffer = numpy.zeros((1920,1080))
window = dpp.Window(arr)
```

Listing 3: Default initialization

```
window = dpp.Window()
```

5.2 Opening the Window

1. Call open function
2. Construct while loop with is_open function

```
window.open()
while window.is_open():
### your code here
```

The open function creates and opens the display. The is_open call maintains and updates the status of all events, as well as the display, on every call. The loop structure is required, as the display will become inactive otherwise.

5.3 Scaling

The window may be scaled up or down in one of three ways:

1. Provide a scale value to Window on initialization
2. Call the set_scale function with the scale value
3. Directly modify the scale member

The default scale value is 1.0. Reducing this value will reduce the size of the display, increasing it will increase the size of the display.

This feature can be useful. Such as: creating a virtual canvas of dimensions (50,50). Scaling this up by a factor of 13 will provide display dimensions of (650,650), making it much easier to visualize any changes made.

5.3.1 Events

Capturing events are the way which users utilize registered mouse clicks and key presses. Users have two ways to interface with these events

5.4 Eventq List

The eventsq list will be most often used, as this structure is best for expressions that only need to register once per key press / mouse click. This list is updated on every iteration of the window loop, removing old events and adding new ones that have been registered. These events may be used by simply checking if a specific event is in the list.

Example of what may be held in the eventq after one iteration:

Listing 4: Held in the eventq after one iteration example:

```
["a", "l_shift"]
```

5.5 Events Dictionary

The events dictionary holds String:Boolean key:value pairs. The key indicates the event to check for, and value is a Boolean indicating if a key or the mouse is currently pressed. It is ideal for continuous expression calls while a key/mouse is held down. It is not recommended to utilize this interface unless incorporated with custom handling when only one expression call is required for an event trigger.

Listing 5: Constantly printing to standard out while left-shift is held down

```
while window.is_open():
    if window.events["l_shift"]:
        print("Left Shift is pressed DOWN!")
```

5.6 Mouse Position

Obtaining the current position of the mouse is done by calling the get_mouse_pos function. This will return a tuple of coordinates: (x , y). These coordinates are with respect to both the window, and the underlying VBuffer data structure.

Listing 6: Setting pixel at mouse location to red

```
if "mouse" in window.eventq:
    red = 0xFF0000
    pos = window.get_mouse_pos() #get mouse position
    window.vbuffer[pos[0], pos[1]] = red # set pixel at mouse (x,y) to red
    print(f"Color at {pos} changed to Red")
```

CHAPTER

SIX

SOURCE DOCUMENTATION

6.1 Audio

class dpav.audio.**Audio**

Bases: `object`

Handles Audio capabilities of Direct Python Audio/Video.

_bit_number

The bit rate of the audio buffer (locked at 16).

Type int

_sample_rate

The sample rate of the audio buffer (locked at 44100).

Type int

_audio_buffer

An array holding sounds to be played.

Type `numpy.ndarray`

_audio_device

The array index of list of audio_devices (see set_audio_devices).

Type int

volume_level

A number between 0 and 1 for the volume.

Type float

waves

A table of built in waveforms.

Type `wave_table`

get_audio_buffer() → numpy.ndarray | None

Returns the audio buffer of the Audio class.

get_audio_device() → int

Gets the current audio device number of the Audio Class.

`get_bit_number()` → int

Gets the bit rate of the Audio class

Note: The bit rate is currently locked to 16 bits.

Returns The bit rate of the Audio class.

`get_sample_rate()` → int

Gets the sample rate of the Audio class.

Note: The sample rate is currently locked to 44100.

Returns The sample rate of the audioClass - int value

Return type self._sample_rate

`list_audio_devices()` → None

Lists the output devices on your system and adds to list self._devices.

Run this function before using set_audio_device() to add devices to the list devices.

Example

audioobject.list_audio_devices() 0 Speakers (Realtek(R) Audio) 1 VGA248 (2-NVIDIA High Def Audio) 2 Speakers (HyperX Cloud II Wireless)

`play_sample`(*sample_name: str*) → None

Plays sounds that are wav, ogg or mp3 files.

Parameters **`sample_name`** – String path or name of sound

Example

audioobject.play_sample(mypath.mp3) would play sounds from the file mypath.mp3

`play_sound`(*input_frequency=0*, *input_duration=0*) → None

Primary sound playing method of the audio class.

- Play sounds directly from this function.
- Need to run set_audio_device() or will default to the default audio device.
- You can use set_waveform to change the type.
- **play_sound is somewhat overloaded to where if you have an audioBuffer set using set_audio_buffer, you can call** and it will play whatever that audio_buffer is e.g. wav files (Example in examples/custombuffer.py). play_sound(440, 1) would play an A note for one second with the sin waveform set.

Parameters

- **`input_frequency`** – Input frequency in Hz.
- **`input_duration`** – Duration in seconds.

Raises `TypeError` – If input_duration not a number, or < 0.

`set_audio_buffer`(*ab: numpy.ndarray*) → None

Sets the audio buffer of the Audio Class.

The audio buffer needs to have two rows so that way stereo works as intended. You can set the audio buffer to wav file data by fetching numpy arrays using wav or scipy, however only 16 bit waves are supported. This process can be seen in custom_buffer.py w/ the utility function sixteenWavtoRawData

Parameters `ab` – An array of shape(samples, channels) e.g. ab[44100][2]

Examples

44100 = sample rate # 32767 is 2 ^ (our bit depth -1)-1 and is essentially the number of samples per time stamp # 260 and 290 are our tones in hz # Below generates a buffer 1 second long of sin wave data-identical to the method used in house data = numpy.zeros((44100, 2), dtype=numpy.int16) for s in range(44100):

t = float(s) / 44100 data[s][0] = int(round(32767 * math.sin(2 * math.pi * 260 * t))) data[s][1] = int(round(32767 * math.sin(2 * math.pi * 290 * t)))

audioobject.set_audio_buffer(data)

`set_audio_device`(*device: int*) → None

Sets the current audio device of the Audio class.

This can only be set ONCE per instance. To change devices, del the current instance set the new device, and continue This needs to be run after list_audio_device() in order to see list of audio devices If not run the device will default to the current device being used by the machine

Parameters `device` – The array index for the selected audio device.

Example

audioobject.set_audio_device(2) Based on example in list_audio_devices() this would change the device to Speakers (HyperX Cloud II Wireless)

`set_waveform`(*wave*) → None

Sets the waveform to be used for audio playback.

play_audio uses this in buffer generation.

Parameters `Wave` – takes a mathematical expression function 'pointer' in the form of f(inputfreq, timestep).

Example

audioobject.set_waveform(object.wave_table.sin) This would change to the waveform sin contained in the wave_table class The wave functions need to take in a input frequency as well as a timestep parameter to solve for a particular frequency at a given time step. See wave_table for an example of this.

`wait_for_sound_end`()

Function call that is placed at the end of scripts without a pygame window instance so sounds play to their full duration.

Example

play_sound(440, 10) wait_for_sound_end() # This prevents the process from closing out before the sound ends.

class dpav.audio.**wave_table**

Bases: `object`

This is a class holding waveforms for usage with the play_sound method.

Example

waves = wave_table() sinefunc = waves.sin

noise(*input_frequency: float*, *t: float*) → float

Random white noise.

This waveform is quite loud.

saw(*input_frequency: float*, *t: float*) → float

Returns the value of a saw wave of a given frequency at a given time.

sin(*input_frequency: float*, *t: float*) → float

Returns the value of a sine wave of a given frequency at a given time.

This is the default waveform for the library.

square(*input_frequency: float*, *t: float*) → float

Returns the value of a square wave of a given frequency at a given time.

triangle(*input_frequency: float*, *t: float*) → float

Returns the value of a triangle wave of a given frequency at a given time.

This wave is similar in sound to a saw and sine wave together.

6.2 VBuffer

class dpav.vbuffer.**VBuffer**(*arg1: tuple = (800, 600)*)

Bases: `object`

Visual buffer for the Python Direct Platform

Holds a 2D array of hex color values. Each element represents a pixel, whose coordinates are its index. VBuffer can be loaded and displayed by the window class.

clear() → None

Set every pixel in the buffer to 0 (hex value for black).

property dimensions: tuple

Returns the dimensions of the buffer.

fill(*color: int*) → None

Sets every pixel in the buffer to a given hex color value.

get_dimensions() → tuple

Returns the dimensions of the visual buffer

get_pixel(*coords: tuple*) → int

Returns the color value of chosen pixel.

load_buffer_from_file(*filename: str*) → None

Load binary file storing buffer contents, and write it to buffer.

Parameters **filename** – The path to a binary file containing numpy array data.

save_buffer_to_file(*filename: str*) → None

Save contents of buffer to a binary file.

Parameters **filename** – The path and name of the file to write.

set_buffer(*buf: numpy.ndarray*) → None

Set the contents of a visual buffer to match those of a given array.

Parameters **buf** – An array of hex color values.

write_pixel(*coords: tuple*, *val: int*) → None

Sets pixel at the specified coordinates to a specified color.

Parameters

- **coords** – Pixel coordinates (x, y).
- **val** – The hex value of the desired color.

Raises

- **TypeError** – If val is not an int.
- **ValueError** – If val is negative or greater than the max color value (0xFFFFFF).

6.3 Window

class dpav.window.**Window**(*arg1: Optional[dpav.vbuffer.VBuffer] = None*, *scale: float = 1.0*)

Bases: `object`

Handles Window capabilites of Direct Python Audio/Video.

vbuffer

The currently active visual buffer.

Type `VBuffer`

scale

A number that scales up/down the size of the screen (1.0 is unscaled).

Type float

events

A dictionary of string:bool event pairs.

eventq

The active events that occured since last update cycle.

Type list

debug_flag

A flag for whether or not the window object should output debug info to log

Type bool

open_flag

A flag for whether or not the window is active.

Type bool

_keydict

int:string PyGame event mapping. PyGame events identifiers are stored as ints. This attribute is used by the public events variable to map from PyGame's integer:boolean pairs to our string:boolean pairs.

_surfaces

Two PyGame Surfaces for swapping to reflect vbuffer changes and to enable in-place nparray modification.

Type list

_screen

A PyGame.display object, used for viewing vbuffer attribute.

close() → None

Closes the active instance of a pygame window.

Raises **RuntimeError** – If no active pygame window exists.

get_mouse_pos() -> (<class 'int'>, <class 'int'>)

Returns the current mouse location.

Returns A tuple containing the coordinates of the current mouse location in the window.

Raises **Runtime Error** – If no active pygame window instances exists.

is_open() → bool

Method that the user can check in a while loop to maintain a window.

This call updates events on every call and is used to abstract out PyGame display calls as well as the event loop.

Example

if window.is_open(): # your code here

Returns A boolean value denoting whether or not the window is currently open.

open() → None

Creates and runs pygame window.

set_scale(*scale: float*) → None

Sets the window scale.

set_vbuffer(*arg1: dpav.vbuffer.VBuffer*) → None

Sets the visual buffer object to display on screen.

Parameters **arg1** – The visual buffer to display.

Raises

- **TypeError** – If arg1 is not either a VBuffer or a np.ndarray.

- **TypeError** – If the scale is not a float.

update() → None

Updates the Pygame window to display changes made to the visual buffer.

Note: This function's use is optional if is_open() is used.

Raises Runtime Error – If there is no active pygame window.

6.4 Utility

The utility.py module defines a variety of utility functions to the dpav library.

This module adds utility functions for line and shape drawing, visual buffer transformations, image parsing, and note conversions.

Examples

$ utility.draw_line(vb, (3, 3), (5, 5), 0x00FF00)

dpav.utility.**convert_wav_to_nparr**(*wavefile: str*) → numpy.ndarray

Takes a string filepath of a wav file and converts it to a numpy array.

dpav.utility.**draw_circle**(*vb: dpav.vbuffer.VBuffer*, *center: list*, *r: float*, *color: int*)

Draws a circle onto a visual buffer of a specified color and radius around a given center point using Bresenham's algorithm.

dpav.utility.**draw_line**(*vb: dpav.vbuffer.VBuffer*, *p0: list*, *p1: list*, *color: int*)

Draws a line of a given color on a visual buffer from p0 to p1 using Bresenham's algorithm.

dpav.utility.**draw_polygon**(*vb: dpav.vbuffer.VBuffer*, *vertices: list*, *color: int*)

Draws lines of a given color connecting a list of given points in the order they are listed

dpav.utility.**draw_rectangle**(*vbuffer: dpav.vbuffer.VBuffer*, *color: int*, *pt1: tuple[int, int]*, *pt2: tuple[int, int]*)

Draws a rectangle into a visual buffer.

Parameters

- **vbuffer** – A visual buffer to write a rectangle into.
- **color** – The color the rectangle should be.
- **pt1** – One corder of the rectangle.
- **pt2** – The opposite corner from pt1 of the rectangle.

Examples

utility.draw_rectangle(vb, 0xFFFFFF, (3, 3), (5, 5))

`dpav.utility.fill`(*vb: dpav.vbuffer.VBuffer*, *color: int*, *vertices*)

Fills a polygon defined by a set of vertices with a color.

`dpav.utility.flip_horizontally`(*vb: dpav.vbuffer.VBuffer*) → dpav.vbuffer.VBuffer

Takes a visual buffer, flips it horizontally about the center, and returns the new visual buffer.

`dpav.utility.flip_vertically`(*vb: dpav.vbuffer.VBuffer*) → dpav.vbuffer.VBuffer

Takes a visual buffer, flips it vertically about the center, and returns the new visual buffer.

`dpav.utility.get_note_from_string`(*note: str*, *octave: int*) → int

Converts a string denoting a note and an octave into a frequency.

Parameters `note` – A musical note denoted with a capital letter and a sharp (#) or a flat (b).

Returns A frequency in hertz.

`dpav.utility.load_image`(*filepath: str*) → numpy.ndarray

Converts an image and returns a numpy array representation of that image in hex.

Parameters `filepath` – The filepath of the image to be loaded

Returns A numpy array filled with the hex color data of the image

`dpav.utility.point_in_polygon`(*x: int*, *y: int*, *vertices*) → bool

Uses the Even-Odd Rule to determien whether or not a given pixel is inside a given set of vertices.

Parameters

- `x` – The x coordinate of the pixel to be checked.
- `y` – The y coordinate of the pixel to be checked.

Returns True if the pixel is within the polygon, False otherwise.

`dpav.utility.replace_color`(*vb: dpav.vbuffer.VBuffer*, *replaced_color: int*, *new_color: int*)

Replaces all pixels in a visual buffer of a chosen color with a new color.

`dpav.utility.rgb_to_hex`(*arr: numpy.ndarray*) → numpy.ndarray

Converts a numpy array with (r, g, b) values into a numpy array with hex color values.

`dpav.utility.translate`(*vb: dpav.vbuffer.VBuffer*, *x_translation: int*, *y_translation: int*) → dpav.vbuffer.VBuffer

Takes a visual buffer, translates every pixel in it by given values, and returns the new visual buffer