Direct Python Audio/Video

Release 0.0.1

Vibrant Labs

May 01, 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 Python Direct Platform.

Functions:

Constructor: __init__()

Functions:

play_sound(Hz, length)

If audio buffer is set: play_sound()

play_sample(string_name_of_wav_file)

Setters: set_audio_buffer(numpyarray) set_audio_device(int) set_waveform(waveform)

Getters: get_bit_number()->int get_sample_rate()->int get_audio_buffer() get_audio_device()->Returns int corresponding to audio device

Misc: list_audio_devices() wait_for_sound_end()

`get_audio_buffer()`

Returns the audio buffer of the Audio class

Description: This will return none if the audio buffer has not been set by the set_audio_buffer method.

audioobject.get_audio_buffer()

Parameters `None` –

Returns numpy array

Return type self._audio_buffer

`get_audio_device()` → int

Gets the current audio device number of the Audio Class

Description: Assuming audioobject.set_audio_device(2) is called, audioobject.get_audio_device() would return 2 [index of audio device in audioobject.list_audio_devices()]

Parameters `None` –

Returns self._audio_device: int value

Notes

Returns the integer value of the device not the device name

`get_bit_number()` → int
Gets the bit rate of the Audio class

Description: Bit rate currently locked to 16 bits

Parameters `None` –

Returns The bit rate of the Audio class - int value

Return type self._bit_number

`get_sample_rate()` → int
Gets the sample rate of the Audio class.

Description: Sample rate is currently locked to 44100

Parameters `None` –

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

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

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

Parameters `None` –

Returns None

`play_sample`(*sample_name: str*) → None
Plays sounds that are wav, ogg or mp3 files.

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

Parameters `sample_name` – String path or name of sound

Returns None

`play_sound`(*input_frequency=0*, *input_duration=0*) → None
Primary sound playing method of the audio class.

Description: 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 play_sound()

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` – int value - input frequency in Hz

- `input_duration` – int value - duration in seconds

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

Returns None

`set_audio_buffer`(*ab*) → None
Sets the audio buffer of the Audio Class.

Description: 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

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)

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

Returns None

`set_audio_device`(*device: int*) → int
Sets the current audio device of the Audio class.

Description: 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

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

Parameters `device` – int value - see all int values for each device by running list_audio_devices()

Returns None

`set_waveform`(*wave*) → None
Sets the expression governing the wave form playing

Description: play_audio uses this in buffer generation

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.

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

Returns None

`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 without a

Description: Placed at the end of python files that do not have loops. Otherwise, sounds would be cut off prematurely.

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

Parameters `None` –

Returns None

Notes:

`class dpav.audio.`**`wave_table`**

Bases: `object`

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

There are 5 waveforms: sin saw square noise triangle

Example

waves = wave_table() sinefunc = waves.sin

`noise`(*input_frequency*, *t*)

Random white noise

Description: Warning: VERY LOUD

Parameters

- **`input_frequency`** – value in Hz at timestep t
- **`t`** – timestep

Returns

Return type random.random() * input_frequency * t

`saw`(*input_frequency*, *t*)

Saw wave

Parameters

- **`input_frequency`** – value in Hz at timestep t
- **`t`** – timestep

Returns

Return type t * input_frequency - math.floor(t * input_frequency)

`sin`(*input_frequency*, *t*)

Sin wave form, default for libary

Parameters

- **`input_frequency`** – value in Hz at timestep t
- **`t`** – timestep

Returns

Return type math.sin(2 * math.pi * input_frequency * t)

`square`(*input_frequency*, *t*)
Square wave form

Parameters

- **input_frequency** – value in Hz at timestep t
- **t** – timestep

Returns

Return type round(math.sin(2 * math.pi * input_frequency * t))

`triangle`(*input_frequency*, *t*)
Triangle wave, similar in sound to saw + sin together

Parameters

- **input_frequency** – value in Hz at timestep t
- **t** – timestep

Returns

Return type 2 * abs((t * input_frequency) / 1 - math.floor(((t * input_frequency) / 1) + 0.5))

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.

Parameters **arg1** (*{(int, int)|np.ndarray(int, int)}*) – Either array dimensions or a 2-dimensional numpy array of integers

If dimensions, will create zeroed-out 2D array of the selected dimensions. Defaults to 800x600.

If numpy array, will set buffer to the contents of that array.

Constructor:
__init__(self, arg1=(800, 600)) -> None

Overloads:
__getitem__(self, idx) -> int __setitem__(self, idx, val) -> None __len__(self) -> int

properties:

getter: dimensions(self) -> (int, int)

setter: dimensions(self, val) -> None

Setter:
write_pixel(self, coords, val) -> None set_buffer(self, buf) -> None clear(self) -> None fill(self, color: int) -> None

Getters:
get_pixel(self, coords) -> int get_dimensions(self) -> (int, int)

File I/O:
save_buffer_to_file(self, filename) -> None load_buffer_from_file(self, filename) -> None

Error Checking:
_check_numpy_arr(self,arg1,arg_name,method_name) -> None _check_coord_type(self, coords, arg_name, method_name) -> None _check_coord_vals(self, x, y, method_name) -> None

`clear()` → None
Set every pixel in buffer to 0 (hex value for black).

property dimensions: tuple
Return dimensions of buffer.

`fill(color: int)` → None
Set every pixel in the buffer to a given color.

Parameters **color** (*Hex color code*) –

`get_dimensions()` → tuple
Return dimensions of visual buffer array.

`get_pixel(coords: tuple)` → int
Return color value of chosen pixel.

Parameters **coords** (*2-tuple or list containing first and second index of pixel*) –

`load_buffer_from_file(filename: str)` → None
Load binary file storing buffer contents, and write it to buffer.

Parameters **filename** (*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 to*) –

`set_buffer(buf: numpy.ndarray)` → None
Set the visual buffer to equal a provided 2D array of pixels.

Parameters **buf** (*A 2-dimensional numpy array of integer color values*) –

`write_pixel(coords: tuple, val: int)` → None
Sets pixel at specified coordinates to specified color.

Sets pixel at coordinates coords in buffer to hex value val

Parameters

- **coords** (*Pixel coordinates (an X and a Y)*) –
- **val** (*The hex value of the desired color to change the pixel with*) –

:raises TypeError : val is not type(int): :raises ValueError : val is negative or greater than 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 Python Direct Platform Functions:

Constructor: __init__()

Setters: set_scale(int/float) set_vbuffer(VBuffer/np.ndarray,optional:int)

Getters: get_mouse_pos()

Misc Methods: open() is_open() close() update()

Private Methods: _update_events(pygame.event) _build_events_dict() _write_to_screen()

Public

vbuffer: active VBuffer object scale: number that scales up/down the size of the screen

(1.0 is unscaled)

events: dictionary of string:bool event pairs,

example: "l_shift": True – left shift is pressed down "l_shift": False – left shift is not pressed

eventq: list of active events that occured since last update cycle

example: ['l_shift', 'mouse', 'a', 'q']

debug_flag: boolean flag if window object should output debug info to log open_flag: boolean flag for if the window is active

Private

_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 enable in-place nparray modification

_screen: PyGame.display object, used for viewing vbuffer attribute

`close()` → None

Closes the active instance of a pygame window

Raises `RuntimeError` – no active pygame window instances exists

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

Returns the current mouse location with respect to the pygame window instance

Raises `Runtime Error` – no active pygame window instances exists

`is_open()` → bool

Updates events on every call, used to abstract out PyGame display calls and event loop

Example

if window.is_open(): # your code here

Returns boolean denoting if the window is currently open

`open()` → None
Creates and runs pygame window in a new thread

`set_scale`(*scale: float*) → None
Sets the window scale

`set_vbuffer`(*arg1:* dpav.vbuffer.VBuffer) → None
Sets the vbuffer/nparray object to display on screen

Parameters `arg1` – VBuffer/np.ndarray

Raises

- `TypeError` – arg1 VBuffer/np.ndarray type check
- `TypeError` – scale int/float type check

`update()` → None
Pygame event abstraction, called at end of pygame loop. Optional function if is_open() is used

Raises `Runtime Error` – 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

CHAPTER SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

T

U

V

W