Piano Video

Release 0.4.0

Patrick Huang

Sep 18, 2021

GENERAL

CHAPTER

ONE

ABOUT

Piano Video is a completely free piano visualization software, which the GNU GPL v3 guarentees.

The official repository and distribution on GitHub also provides all downloads gratis.

CHAPTER

TWO

ACKNOWLEDGEMENTS

Thanks to the developers

Thanks to Blender, a free 3D creation software. Many design ideas were based off of Blender, though no code was directly copied.

CHAPTER

THREE

RELEASE SCHEDULE

There is currently no release schedule. Generally, Piano Video is released every 3 to 14 days.

Piano Video follows Semantic Versioning.

CHAPTER

FOUR

CHANGELOG

0.4.0 (development)

0.3.2 (current release)

- Minor improvements to smoke and particles.
- Improved block glow.
- Option for octave lines.
- New glare look with rainbow effects.

0.3.1

- Fade out bottom of keyboard.
- Streaks on bright particles.
- Minor improvements to smoke and particles.

0.3.0

- Add your video to the final render.

CHAPTER

FIVE

INSTALLATION

5.1 Requirements

- The GNU Compiler Collection
- GNU Make
- Python 3.8
- Python packages

5.2 GNU/Linux

5.2.1 Pre-Built

The latest release can be found here.

All releases (starting from `v0.2.0`) have pre-built binaries. There will be a Debian package for Ubuntu and Debian, and two Python wheel files.

If you are unable to install one or more of those, please see the "From Source" section below.

To install the Debian package:

```
sudo dpkg -i pvid_x.x.x.deb
```

To install the wheel files:

```
pip install pv_...whl
pip install pvkernel_...whl
```

File names vary.

5.2.2 From Source

First build the files. Instructions are here

Then, install the files as above.

5.3 Windows and Mac

If you are using Windows or MacOS, switch to GNU/Linux and give yourself some freedom

Piano Video is developed and tested on GNU/Linux and may or may not work on other operating systems.

5.4 Old Versions

Piano Video has had two previous versions before `v0.2.0`.

The first was a CLI version which was difficult to use. Unfortunately, the documentation does not exist anymore. You can find the program on the `cli-old` branch.

The second is an unfinished GUI version, similar in design to the current version. It can be found on the branch `gui-old` and sphinx documentation is in the `/docs` folder.

CHAPTER

SIX

SUPPORT

Please open a discussion or issue on GitHub for support.

CHAPTER

SEVEN

USER MANUAL

7.1 Recording

Set up a camera and a piano keyboard capable of MIDI recording. Record a video and MIDI file. This tutorial assumes they are titled `midi.mid` and `video.mp4`.

You might want to save these in a new folder dedicated to this piano recording.

7.2 Your First Video

First, install Piano Video.

Import the `pvkernel` library, which will give us access to the `Video` class. The default resolution and fps are `(1920, 1080)`, and `30` respectively.

```
import pvkernel

resolution = (1920, 1080)
fps = 30
video = pvkernel.Video(resolution, fps)
```

Now that we have our video initialized, we can start adding MIDIs. We add them by modifying the *property* (explained later) `midi.paths`. To add MIDIs, run:

```
video.props.midi.paths = "your/midi/path.mid"
```

This adds your MIDI file into the paths that the kernel will render. To seperate multiple MIDIs, use just a colon with no spaces. There are example MIDIs all ready inside the examples folder.

We can also add the video recording to play underneath the blocks. Along with the video, we need to specify the start time in seconds and the crop.

The start time is the time you press the first note in the video. So, if you press the first note at `1.23` seconds, put `1.23` into the property.

The crop is a list of four lists (see below). They specify the four corners of the keyboard keys in your video, starting from the top left and going clockwise. The X coordinate starts at 0 from the left and increases going to the right. The Y coordinate starts at 0 from the top and increases going down.

```
video.props.keyboard.video_path = "your/video/path.mp4"
video.props.keyboard.video_start = 1.23   # seconds
video.props.keyboard.crop = [[x1,y1], [x2,y2], [x3,y3], [x4,y4]]  # put in actual values␣
↪here
```

(continues on next page)

(continued from previous page)

Almost done! We just have to export the video using the code

```
video.export("video.mp4")
```

Hooray! You have your first video exported! With the default settings, this can take a few minutes. Feel free to stop the export (`Ctrl+C`) at any time, and you will be able to view the rendered frames in the video.

7.3 Properties

Each video class instance has it's own collection of property values. The available properties are defined by *add-ons* (covered below). Detailed property documentation can be found here.

Let's change the block color to blue (add this line before the export):

```
video.props.blocks_solid.color = (100, 100, 200)
```

Now, export again, and you should see the blocks are now blue.

CHAPTER

EIGHT

BUILT-IN OPTIONS

This page provides reference to the built-in options you can use.

TODO Some of this is old.

8.1 General

- `core.pause_start`: Number of seconds before the first note starts.
- `core.pause_end`: Number of seconds after the last note ends.
- `keyboard.left_offset`: Pixel offset between the left of the screen and the left of the piano.
- `keyboard.right_offset`: Pixel offset between the right of the screen and the right of the piano.
- `keyboard.black_width_fac`: Black key width factor respective to white key.
- `keyboard.video_path`: Path to video file.
- `keyboard.video_start`: Time you start playing the first note in seconds.
- `keyboard.crop`: List of locations of the corner of the keys starting from top left and going clockwise.
- `keyboard.height_fac`: Keyboard height multiplier for rendered image.
- `keyboard.mask`: Amount of space to show under the keyboard in pixels.
- `keyboard.sub_dim`: Subtractive dimming from 0 to 255.
- `keyboard.mult_dim`: Multiplicative dimming.
- `keyboard.rgb_mod`: `[R, G, B]` intensity factors.
- `lighting.on`: Whether to use CG lighting.
- `lighting.piano_width`: Width of all of the piano keys combined in meters.
- `lighting.lights`: List of lights. See lighting for more info.
- `midi.paths`: MIDI file paths. Separate multiple with pathsep (`:`).
- `midi.min_len`: Minimum note length in seconds.
- `midi.reverse`: If True, notes go up from the keyboard.
- `blocks.speed`: Speed in screens per second.
- `blocks.rounding`: Corner rounding radius in pixels.
- `blocks.border`: Border thickness in pixels.
- `blocks.color`: RGB color of the center of the block.

- `blocks.border_color`: RGB color of the border of the block.
- `glare.intensity`: Brightness multiplier.
- `glare.radius`: Glare radius in pixels.
- `glare.jitter`: Amount to randomize intensity by.
- `ptcls.intensity`: Particle brightness multiplier.
- `ptcls.pps`: Particles per second per note.
- `smoke.intensity`: Brightness multiplier
- `smoke.pps`: Particles per second per note.

CHAPTER

NINE

API

The API is a python module named `pv` that allows users to modify the kernel's behavior.

9.1 Properties

Properties are just variables, but they can be interpreted by the GUI and displayed properly.

class pv.**Property**
Base property class.

Inherit and define:

- `type`: Type.
- `set(value)`: Set the property's value. Default just sets it, but you may need to check requirements.

Call `super().__init__()` **AFTER initializing** `self.default`

set(*value: Any*) → None
Sets the property's value.

Parameters **value** – Any value. Will be casted with `self.type`.

Returns None

class pv.**BoolProp**(*name: str = ''*, *description: str = ''*, *default: bool = False*)
Boolean property.

class pv.**IntProp**(*name: str = ''*, *description: str = ''*, *default: int = 0*, *min: int = Ellipsis*, *max: int = Ellipsis*)
Integer property.

class pv.**FloatProp**(*name: str = ''*, *description: str = ''*, *default: float = 0*, *min: float = Ellipsis*, *max: float = Ellipsis*)
Float property.

class pv.**StrProp**(*name: str = ''*, *description: str = ''*, *default: str = ''*, *max_len: int = 1000*, *choices: Sequence[str] = []*)
String property.

class pv.**ListProp**(*name: str = ''*, *description: str = ''*, *default: List[Any] = [0, 0, 0, 0]*)
List property. Use this for color. May contain a list of lists.

A PropertyGroup is a collection of properties.

class pv.**PropertyGroup**
A collection of Properties.

When creating your own PropertyGroup, you will inherit a class from this base class. Then, define:

- `idname`: The unique idname of this property group.
- properties: Define each property as a static attribute (shown below).

```
class MyProps(pv.PropertyGroup):
    prop1 = pv.props.BoolProp(name="hi")
```

_get_prop(*name: str*) → *pv.props.Property*
You can use this to bypass `__getattribute__` and get the actual Property object.

9.2 Data and Cache

The API has a few classes for storing and accessing data.

class pv.**DataGroup**
A group of data pointers.

When creating your own DataGroup, inherit and define the idname.

Then, you can run `video.data_idname.value = x` or `video.data_idname.value2` to access and set values.

The values can be any type. Value names cannot be `idname` or `items`, as they will overwrite internal variables.

class pv.**Cache**(*video: None*)
Cache managing for a video.

You can read and write specific file names with `cache.fp(name, mode)`, or automatically set the name to the current frame with `cache.fp_frame`.

To add a cache, inherit and define:

- `idname`: Cache idname. Will also be the cache folder name.
- `depends`: Tuple of property idnames this cache depends on. If any of them change, the cache will be cleared. Default `()`.

9.3 Operators

Operators are functions that operate on a video and can be displayed in the GUI.

class pv.**Operator**(*video: None*)
A function that is positioned at `pv.ops.group.idname`. It can be displayed in the GUI.

The return value will always be None.

To create your own operator, inherit and define:

- `group`: Operator group.
- `idname`: Unique operator idname.
- `label`: The text that will show on the GUI (as a button).
- `description`: What this operator does.
- `execute(video)`: This will be run when the operator is called. The first parameter is the video class (`pvkernel.Video`)

9.4 Jobs

Jobs modify the rendering process.

class pv.**Job**

Create a job to modify the final video.

See https://piano-video.rtfd.io/en/latest/blog/render_jobs.html for more info.

Inherit and define:

- `idname`: Job idname.
- `ops`: List of operator idnames (`"group.idname"`) to run.
- `execute`: This function will run before running the operators. Default does nothing.

9.5 Utilities

pv.utils.**register_class**(*cls: Type*) → None

Register a class to "apply" it to the kernel.

pv.utils.**add_callback**(*func: Callable*, *classes: Sequence[str]*) → None

Add a callback function when a class is registered. The function will be passed one argument, the class.

Parameters

- **func** – Function to call.
- **classes** – A list of strings indicating which types of classes to listen for. Valid values:
  - "cache": Cache
  - "dgroup": DataGroup
  - "pgroup": PropertyGroup
  - "ogroup": Operator

pv.utils.**get**(*objs: Sequence[Any]*, *idname: str*) → Any

Return the object in `objs` with idname `idname`.

pv.utils.**get_index**(*objs: Sequence[Any]*, *idname: str*) → int

Return the index of the object in `objs` with idname `idname`.

pv.utils.**get_exists**(*objs: Sequence[Any]*, *idname: str*) → bool

Check whether there is an object with idname `idname`.

CHAPTER

TEN

INTRODUCTION

Thanks for considering contributing to Piano Video!

As outlined in the plan, Piano Video comes in three sections:

- GUI
- API
- Kernel

Documentation on the sections can be found in their respective pages.

10.1 General

Most typo or bug fixes are accepted. If you propose an incompatible API change or a major feature, please discuss with me first.

Do not make patches for Windows or MacOS support. I will not support malware in this project. If you would like Windows or MacOS support, please fork the project and edit your own copy.

To start working on a task, please either look through the projects or issues on GitHub and choose one that interests you.

10.2 File Structure

The Piano Video repository contains these folders:

Table 1: File Structure

Path	Description
`/.github`	GitHub workflows for testing.
`/build`	Build scripts for `deb` and `whl`.
`/docs`	Documentation (mainly Sphinx).
`/examples`	Example MIDI and videos. They are licensed as CC0.
`/src`	Source code. Contains a few subdirectories.
`/src/pv`	API source code.
`/src/pvgui`	GUI source code.
`/src/pvkernel`	Kernel source code.
`/tests`	Testing scripts.

CHAPTER

ELEVEN

BUILDING

As mentioned in the plan, Piano Video comes in three parts. Each part is built to separate binary files.

Build instructions:

```
git clone https://github.com/phuang1024/piano_video.git
cd ./piano_video
make dist
```

This will generate files in the `build` directory.

The distributable binaries are:

- `build/pvid_x.x.x.deb`: The GUI, as a Debian package.
- `build/dist/pv-...whl`: The API, as a Python library.
- `build/dist/pvkernel-...whl`: The Kernel, as a Python library.

To install, see Installation.

CHAPTER

TWELVE

RUN FROM SOURCE

While developing, it may be faster to run the scripts directly from source, instead of building and installing for every test. To do this, place a Python file in the *src* directory, and it will be able to import *pv* and *pvkernel*.

CHAPTER

THIRTEEN

GUI

This is the interface for most end users. Currently, it does nothing, and I will likely not accept contributions to this area for now.

It will probably be written in Python, with GUI libraries like Tkinter or Pygame.

The GUI source can be found in `/src/gui`

CHAPTER

FOURTEEN

API

For API documentation, see API

This is a pure Python library that doesn't do much computationally, but must be easy and intuitive for add-on developers.

The API **cannot** be changed without prior notice in the docs, so if you intend to improve this area, you will most likely be doing docstrings, type hinting, etc.

The API source can be found in `/src/pv`

CHAPTER

FIFTEEN

KERNEL

This is the core rendering engine. The kernel can be further split into two parts: Rendering System and Built-in Add-ons.

The rendering system is relatively simple. All it does is call the rendering functions in order and put together the final video. This is written in Python.

The built-in add-ons are complex. They are the actual implementations of the effects. They use Python to access the API, but may optionally call C++ libraries for a speed improvement.

The rendering system can be found in `/src/pvkernel`. The add-ons are at `/src/pvkernel/addons`.

Most optimization or minor improvements are welcome. If your patch greatly changes the appearance, please receive feedback from some people about it.

If you would like to create a new add-on bundled with the kernel, please discuss with me first. Thanks!

However, you can develop your own add-ons and install them through the GUI.

CHAPTER

SIXTEEN

PIANOSYNTH

This is a plan for an idea I have, which is a software synthesizer specifically for piano.

It is a command line tool.

You can track the progress in `piano_video/src/pianosynth`.

More updates coming!

CHAPTER

SEVENTEEN

PLAN

Written on August 4, 2021

Updated August 5, 2021

Currently, I have the plan to develop Piano Video in three sections:

- `pvkernel`: This is a Python library (probably with C++) that handles all rendering, effects, MIDI parsing, etc. The kernel also contains built-in add-ons, which use the API (below) to add functionality to the kernel. They will be considered part of the kernel.
- `pv`: This is a Python library which provides an API to the kernel.
- `pvgui`: This is a Python GUI application (with Tkinter) that most end users will launch. It provides a graphical interface to the API and kernel. **Invoked through the command** `pvid`.

End users can develop their own add-ons and install them. `pvgui` manages user add-on installations.

CHAPTER

EIGHTEEN

RENDER JOBS

Written August 6, 2021

Updated August 10, 2021

In order to be easily extensible, I will develop the rendering process with *jobs*: Jobs will be included in *Job Slots*, which are specific to each Video class instance.

- `init`: Prepare for rendering
- `intro`: Text introduction
- `frame_init`: Initialize variables specific to a frame.
- `frame`: Rendering the actual video (individual jobs can do a part of it)
- `frame_deinit`: Free memory (if applicable) for each frame.
- `outro`: Text outroduction
- `modifiers`: Modifies the whole image
- `deinit`: Free memory (if applicable), close files, etc.

18.1 API

Jobs are a collection of operator idnames to run.

Some job slots can have multiple jobs (e.g. effects), and some can only have one job (e.g. piano).

The graphical job operators should modify the input image at `video.render_img`

ALL INPUT AND OUTPUT IMAGES WILL HAVE THREE CHANNELS, RGB.

CHAPTER

NINETEEN

GPU ACCELERATION

Written August 9, 2021

For people with cuda GPUs, good news! I will plan cuda libraries for computationally intensive jobs, like smoke effects. In my tests, the GPU performed 600x faster than the CPU on math operations like float modulo and power.

CHAPTER

TWENTY

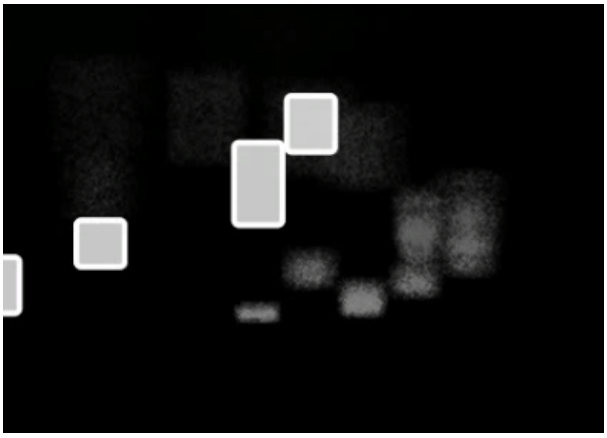
SMOKE SIMULATION

Written August 10, 2021

Updated August 12, 2021

When a note is pressed, smoke will emit out of it. This is different to "Particles" which are small dots of light that also emit out of the note.

20.1 Screenshot



This is the smoke simulation algorithm on August 12. The default setting uses 20000 new smoke particles per second, which exports at about 20 fps on my computer.

The smoke starts rectangular, and then disperses into an amorphous shape. This kind of "echoes" the notes in the smoke, which I think is pretty cool.

20.2 Method 1

The first method is fast but not very accurate. It keeps track of a collection of verticies which represent the outer boundary of the smoke area.

This presents a few problems:

- Currently, there is no polygon draw function implemented.
- If verticies cross (e.g. a vertex previously on top is now lower than another), we need to figure out what order to draw the mesh in.

- Real smoke does not have a surface mesh, and is instead many particles in it's volume.

20.3 Method 2

This is similar to Brownian Motion. The program will simulate possibly millions of particles moving around, and use the density to determine the smoke intensity.

This will be computationally costly.

We need to implement a few rules and parameters:

- Diffusion: Particles repel each other (will also cause the smoke to expand outwards).
- Inertia: How fast particles are affected by outer forces. Parameter called mass.
- Air Resistance: Particles slow down over time.

We can ignore gravity because smoke particles are light.

CHAPTER

TWENTYONE

PARTICLES

Written August 12, 2021

Updated August 20, 2021

For simulating particles, we need to do something different than smoke. Otherwise, the particles will just spread out evenly, which looks boring.

21.1 Update

I didn't implement the algorithm described below, and instead took a different approach, which is just simulating particles without any outer forces.

21.2 Algorithm

First, we emit a invisible element while the note is pressed. It is used for reference in intenral computations, but not rendered. Every frame or so, an (invisible) dot emits at around the same speed as smoke. These dots join to form a 1D squiggly chain.

At the same time, we emit particles, like in smoke. However, these particles will be bigger and there will be less of them. These particles have their own velocity, but are also attracted to the chain. This will cause them to clump up in a fuzzy line.

CHAPTER

TWENTYTWO

NEW API

After developing effects for a few weeks with the API of v0.3.x, it became clear that some changes were needed.

22.1 Problems

- The `pv.Cache` API is hard to use.
- Unable to access C++ utility functions from external add-ons.
- Render Jobs are weird and difficult to use
- No good way of rendering a chunk of frames (this will be useful for multicore export).

22.2 Solutions

- The effects should not exist in the folder `pvkernel/addons`. They should be in their own folder, but may be copied to *pvkernel/addons* ` when building the wheel.
- Instead of render jobs, we will have props and operators under the namespace `render`. These will handle the rendering.
- In each release, `pvutils.hpp` file is provided. This is the header for utility functions.

A free piano visualizer.

INDEX