PursuedPyBear Documentation

Piper Thunstrom

Sep 24, 2019

Contents:

PursuedPyBear, also known as `ppb`, exists to be an educational resource. Most obviously used to teach computer science, it can be a useful tool for any topic that a simulation can be helpful.

CHAPTER 1

A Game Engine

At its core, `ppb` provides a number of features that make it perfect for video games. The `GameEngine` itself provides a pluggable subsystem architecture where adding new features is as simple as subclassing and extending `System`. Additionally, it contains a state stack of `Scenes` simple containers that let you organize game scenes and UI screens in a simple way.

The entire system uses an event system which is as extensible as the rest of the system. Register new values to existing event types, and even overwrite the defaults. Adding a new event system is as simple as calling `Engine.signal` with a new datatype. Instead of a publisher system the engine knows everything in its own scope and only calls objects with appropriate callbacks. The most basic event is `Update` and your handlers should match the signature `on_update(self, update_event, signal)`.

CHAPTER 2

Guiding Principles

Because ppb started to be a game framework great for learning with, the project has a few longterm goals:

2.1 Education Friendly

Non-technical educators should feel comfortable after very little training. While some programming knowledge is required, the ability to think in objects and responses to events allows educators to only focus on their lessons.

2.2 Idiomatic Python

A project built on ppb should look like idiomatic Python. It also should look like modern Python. As such, we often add new language features as soon as they're available, letting a new user always know ppb runs on the latest Python.

2.3 Object Oriented and Event Driven

ppb games are built out of instances of objects that inherit from `EventMixin`. Each object only has enough information to respond to the event provided, which always includes the current `BaseScene`. Because ppb doesn't have a master list of events, you can provide new ones simply to add more granular control over your game.

2.4 Hardware Library Agnostic

Because ppb strongly tries to be extensible and pluggable, each hardware extension can provide its own hooks to ppb, and you can nearly seamlessly switch between various Python libraries.

2.5 Fun

One of the maintainers put it best:

> If it's not fun to use, we should redo it

ppb is about filing off the rough edges so that the joy of creation and discovery are both emphasized. A new user should be able to build their first game in a few hours, and continue exploring beyond that.

2.5.1 Getting Started

This guide will start by getting you a fresh virtual environment and installing ppb. It will then walk you through building a basic game that will look a lot like our sample game targets.py.

Prerequisites

Before you get started here, you should know the basics of Python. We use classes extensively in ppb, and you should be comfortable with them. Consider the Python.org tutorial or automate the boring stuff to get started.

Additionally, you need to have Python 3.6 or later on your machine. You can install this via Python.org or Anaconda whichever is more comfortable for you.

Installing ppb

Once you have a working Python install, you're going to want to make a new folder. Open your shell (Terminal on Mac, CMD or Powershell on Windows, your favorite tool on Linux) and run:

All Systems:

```
mkdir -p path/to/my_game
cd path/to/my_game
```

`path/to/my_game` can be any path you'd like, and the name can be anything you'd like. We cd into it so we have a place to work.

The next step we're going to do is set up a virtual environment. Python 3.6 comes with a tool to create them, so in your terminal again:

All Systems:

```
python3 -m venv .venv
```

This creates a new python environment that we'll use to make our game. To make the next few steps easier, we'll want to activate our virtual environment. This is different on Windows than anywhere else, so make sure to use the right command.

Windows:

```
.venv/bin/activate.bat
```

Linux and Mac:

```
source .venv/bin/activate
```

After you've done this, your shell prompt should include `(.venv)`. We're ready for installing `ppb`:

All Systems:

```
pip install ppb
```

You should see a few libraries get put together in your terminal, and when you have a prompt again, we're ready to go!

A Basic Game

The next step is to make a new file. If you're using an IDE, open your game folder in that and make a new file called `main.py`. If you're using a plain text editor, you'll want to open a new file and save it as `main.py`.

Note: `main.py` *is just being used as a convention and this file can be named anything. If you change the name you'll want to use the new name in further commands.*

In your code file, add this:

`main.py`:

```
import ppb


ppb.run()
```

Save your file, then run it from your shell:

All Systems:

```
python main.py
```

You should have a window! It will be 800 pixels wide and 600 pixels tall, and if you click the x button (or the red dot on MacOS), it should close.

Now let's add a `Sprite`. Sprites are game objects that can often move and are drawn to the screen. Add the following code after your `import`. Note that `ppb.run` has a new parameter.

`main.py`:

```
import ppb


class Player(ppb.BaseSprite):
    pass


def setup(scene):
    scene.add(Player())


ppb.run(setup=setup)
```

When you run this, you should have the same window with a colored square in the middle.

At this point, if you have a png on your computer, you can move it into your project folder and call it `player.png`. Rerun the file to see your character on screen!

Our sprite is currently static, but let's change that. Inside your `Player` class, we're going to add a function and some class attributes.

`main.py`:

```
class Player(ppb.BaseSprite):
    velocity = ppb.Vector(0, 1)

    def on_update(self, update_event, signal):
        self.position += self.velocity * update_event.time_delta
```

Now, your sprite should fly up off the screen.

Taking Control

This is cool, but most people expect a game to be something you can interact with. Let's use keyboard controls to move our `Player` around. First things first, we have some new things we want to import:

`main.py`:

```
import ppb
from ppb import keycodes
from ppb.events import KeyPressed, KeyReleased
```

These are the classes we'll want in the next section to work.

The next step is we'll need to redo out `Player` class. Go ahead and delete it, and put this in its place:

`main.py`:

```
class Player(ppb.BaseSprite):
    position = ppb.Vector(0, -3)
    direction = ppb.Vector(0, 0)
    speed = 4

    def on_update(self, update_event, signal):
        self.position += self.direction * self.speed * update_event.time_delta
```

This new `Player` moves a certain distance based on time, and a direction vector and its own speed. Right now, our direction is not anything (it's the zero-vector), but we'll change that in a moment. For now, go ahead and run the program a few times, changing the parameters to the `direction` `Vector` and the speed and see what happens. You can also modify `position` to see where you like your ship.

Now that you're comfortable with the base mechanics of our new class, revert your changes to `position`, `speed`, and `direction`. Then we can wire up our controls.

First, we're going to define the four arrow keys as our controls. These can be set as class variables so we can change them later:

`main.py`:

```
class Player(ppb.BaseSprite):
    position = ppb.Vector(0, -3)
    direction = ppb.Vector(0, 0)
    speed = 4
    left = keycodes.Left
    right = keycodes.Right
```

The `keycodes` module contains all of the keys on a US based keyboard. If you want different controls, you can look at the module documentation to find ones you prefer.

Now, under our `on_update` function we're going to add two new event handlers. The snippet below doesn't include the class attributes we just defined, but don't worry, just add the new methods at the end of the class, beneath your `on_update` method.

main.py:

```
class Player(ppb.BaseSprite):


    def on_key_pressed(self, key_event: KeyPressed, signal):
        if key_event.key == self.left:
            self.direction += ppb.Vector(-1, 0)
        elif key_event.key == self.right:
            self.direction += ppb.Vector(1, 0)

    def on_key_released(self, key_event: KeyReleased, signal):
        if key_event.key == self.left:
            self.direction += ppb.Vector(1, 0)
        elif key_event.key == self.right:
            self.direction += ppb.Vector(-1, 0)
```

So now, you should be able to move your player back and forth using the arrow keys.

Reaching Out

The next step will to make our player "shoot". I use shoot loosely here, your character can be throwing things, or blowing kisses, or anything, the only mechanic is we're going to have a new object start at the player, and fly up.

First, we need a new class. We'll put it under `Player`, but above `setup`.

main.py:

```
class Projectile(ppb.BaseSprite):
   size = 0.25
   direction = ppb.Vector(0, 1)
   speed = 6

   def on_update(self, update_event, signal):
       if self.direction:
           direction = self.direction.normalize()
       else:
           direction = self.direction
       self.position += direction * self.speed * update_event.time_delta
```

If we wanted to, we could pull out this `on_update` function into a mixin that we could use with either of these classes, but I'm going to leave that as an exercise to the reader. Just like the player, we can put a square image in the same folder with the name `projectile.png` and it'll get rendered, or we can let the engine make a colored square for us.

Let's go back to our player class. We're going to add a new button to the class attributes, then update the `on_key_pressed` method. Just like before, I've removed some code from the sample, you don't need to delete anything here, just add the new lines: The class attributes `right` and `projector` will go after the line about `speed` and the `new elif` will go inside your `on_key_pressed` handler after the previous `elif`.

main.py:

```
class Player(ppb.BaseSprite):

    right = keycodes.Right
    projector = keycodes.Space

    def on_key_pressed(self, key_event: KeyPressed, signal):
```

(continues on next page)

(continued from previous page)

```
        if key_event.key == self.left:
            self.direction += ppb.Vector(-1, 0)
        elif key_event.key == self.right:
            self.direction += ppb.Vector(1, 0)
        elif key_event.key == self.projector:
            key_event.scene.add(Projectile(position=self.position + ppb.Vector(0, 0.
↪5)))
```

Now, when you press the space bar, projectiles appear. They only appear once each time we press the space bar. Next we need something to hit with our projectiles!

Something to Target

We're going to start with the class like we did before. Below your Projectile class, add

`main.py`:

```
class Target(ppb.BaseSprite):

    def on_update(self, update_event, signal):
        for p in update_event.scene.get(kind=Projectile):
            if (p.position - self.position).length <= self.size:
                update_event.scene.remove(self)
                update_event.scene.remove(p)
                break
```

This code will go through all of the `Projectiles` available, and if one is inside the `Target`, we remove the `Target` and the `Projectile`. We do this by accessing the scene that exists on all events in ppb, and using its `get` method to find the projectiles. We also use a simplified circle collision, but other versions of collision can be more accurate, but left up to your research.

Next, let's instantiate a few of our targets to test this.

`main.py`:

```
def setup(scene):
    scene.add(Player())

    for x in range(-4, 5, 2):
        scene.add(Target(position=ppb.Vector(x, 3)))
```

Now you can run your file and see what happens. You should be able to move back and forth near the bottom of the screen, and shoot toward the top, where your targets will disappear when hit by a bullet.

Congratulations on making your first game.

For next steps, you should explore other *tutorials*. Similarly, you can discover new events in the *event documentation*.

2.5.2 Tutorials

Tutorials live here, except for the basic Quick Start tutorial.

A tutorial is an complete project that takes you from an empty file to a working game.

2.5.3 How To: The ppb Cookbook

This section is for direct how tos to solve specific problems with ppb.

2.5.4 API Reference

For as simple as the tutorials make `ppb` look there's a lot of power under the hood. This section will cover the raw what of the ppb API. To find out why decisions are made, see the discussion section.

Starting Your Game

There are two major patterns for starting a game

```
import ppb

def setup(scene):
    ...

ppb.run(setup=setup)
```

```
import ppb

class MyScene(ppb.BaseScene):
    ...

ppb.run(starting_scene=MyScene)
```

Events

All game objects (the engine, scenes, sprites, systems, etc) receive events. Handlers are methods that start with `on_`, eg `on_update`, `on_button_pressed`.

The signature of these handlers are the same: `(event, signal)`:

- `event`: An object containing all the properties of the event, such as the button pressed, the position of the mouse, the current scene
- `signal`: A callable that accepts an object, which will be raised as an event: `signal(StartScene(new_scene=OtherScene()))`

Engine Events

These are core events from hardware and the engine itself.

class ppb.events.**Update**(*time_delta: float*, *scene: ppb.scenes.BaseScene = None*)
Fired on game tick

class ppb.events.**PreRender**(*scene: ppb.scenes.BaseScene = None*)
Fired before rendering.

class ppb.events.**Idle**(*time_delta: float*, *scene: ppb.scenes.BaseScene = None*)
An engine plumbing event to pump timing information to subsystems.

class ppb.events.**Render**(*scene: ppb.scenes.BaseScene = None*)
Fired at render.

class ppb.events.**ButtonPressed**(*button: ppb.buttons.MouseButton*, *position: ppb_vector.Vector*, *scene: ppb.scenes.BaseScene = None*)

Fired when a button is pressed

class ppb.events.**ButtonReleased**(*button: ppb.buttons.MouseButton*, *position: ppb_vector.Vector*, *scene: ppb.scenes.BaseScene = None*)

Fired when a button is released

class ppb.events.**KeyPressed**(*key: ppb.keycodes.KeyCode, mods: Set[ppb.keycodes.KeyCode], scene: ppb.scenes.BaseScene = None*)

class ppb.events.**KeyReleased**(*key: ppb.keycodes.KeyCode, mods: Set[ppb.keycodes.KeyCode], scene: ppb.scenes.BaseScene = None*)

class ppb.events.**MouseMotion**(*position: ppb_vector.Vector, screen_position: ppb_vector.Vector, delta: ppb_vector.Vector, buttons: Collection[ppb.buttons.MouseButton], scene: ppb.scenes.BaseScene = None*)

An event to represent mouse motion.

API Events

These "events" are more for code to call into the engine.

class ppb.events.**Quit**(*scene: ppb.scenes.BaseScene = None*)

Fired on an OS Quit event.

You may also fire this event to stop the engine.

class ppb.events.**StartScene**(*new_scene: Union[ppb.scenes.BaseScene, Type[ppb.scenes.BaseScene]], kwargs: Dict[str, Any] = None, scene: ppb.scenes.BaseScene = None*)

Fired to start a new scene.

new_scene can be an instance or a class. If a class, must include kwargs. If new_scene is an instance kwargs should be empty or None.

Before the previous scene pauses, a ScenePaused event will be fired. Any events signaled in response will be delivered to the new scene.

After the ScenePaused event and any follow up events have been delivered, a SceneStarted event will be sent.

Examples:

- *signal(new_scene=StartScene(MyScene(player=player))*
- *signal(new_scene=StartScene, kwargs={"player": player}*

class ppb.events.**ReplaceScene**(*new_scene: Union[ppb.scenes.BaseScene, Type[ppb.scenes.BaseScene]], kwargs: Dict[str, Any] = None, scene: ppb.scenes.BaseScene = None*)

Fired to replace the current scene with a new one.

new_scene can be an instance or a class. If a class, must include kwargs. If new_scene is an instance kwargs should be empty or None.

Before the previous scene stops, a SceneStopped event will be fired. Any events signaled in response will be delivered to the new scene.

After the SceneStopped event and any follow up events have been delivered, a SceneStarted event will be sent.

Examples:

- *signal(new_scene=ReplaceScene(MyScene(player=player))*

- *signal(new_scene=ReplaceScene, kwargs={"player": player}*

class ppb.events.**StopScene**(*scene: ppb.scenes.BaseScene = None*)
Fired to stop a scene.

Before the scene stops, a SceneStopped event will be fired. Any events signaled in response will be delivered to the previous scene if it exists.

If there is a paused scene on the stack, a SceneContinued event will be fired after the responses to the SceneStopped event.

class ppb.events.**PlaySound**(*sound: ppb.assets.Asset*)
Fire to start a sound playing.

Scene Transition Events

These are events triggered about the lifetime of a scene: it starting, stopping, etc.

The `scene` property on these events always refers to the scene these are about–`ScenePaused.scene` is the scene that is being paused.

class ppb.events.**SceneStarted**(*scene: ppb.scenes.BaseScene = None*)
Fired when a scene starts.

This is delivered to a Scene shortly after it starts. The beginning of the scene lifetime, ended with SceneStopped, paused with ScenePaused, and resumed from a pause with SceneContinued.

class ppb.events.**SceneStopped**(*scene: ppb.scenes.BaseScene = None*)
Fired when a scene stops.

This is delivered to a scene and it's objects when a StopScene or ReplaceScene event is sent to the engine.

The end of the scene lifetime, started with SceneStarted.

class ppb.events.**ScenePaused**(*scene: ppb.scenes.BaseScene = None*)
Fired when a scene pauses.

This is delivered to a scene about to be paused when a StartScene event is sent to the engine. When this scene resumes it will receive a SceneContinued event.

A middle event in the scene lifetime, started with SceneStarted.

class ppb.events.**SceneContinued**(*scene: ppb.scenes.BaseScene = None*)
Fired when a paused scene continues.

This is delivered to a scene as it resumes operation after being paused via a ScenePaused event.

From the middle of the event lifetime that begins with SceneStarted.

Clocks

PPB has several ways to mark time: fixed-rate updates, frames, and idle time. These are all exposed via the event system.

Updates

The *ppb.events.Update* event is fired at a regular, fixed rate (defaulting to 60 times a second). This is well-suited for simulation updates, such as motion, running NPC AIs, physics, etc.

Frames

The *ppb.events.PreRender* and *ppb.events.Render* are fired every frame. This is best used for particle systems, animations, and anything that needs to update every rendered frame (even if the framerate varies).

Note: While both *PreRender* and *Render* are fired every frame, it is encouraged that games only use *PreRender* to ensure proper sequencing. That is, it is not guaranteed when `on_render()` methods are called with respect to the actual rendering.

Idle

ppb.events.Idle is fired whenever the core event loop has no more events. While this is primarily used by systems for various polling things, it may be useful for games which have low-priority calculations to perform.

Assets

PursuedPyBear features a background, eager-loading asset system. The first time an asset is referenced, PPB starts reading and parsing it in a background thread.

The data is kept in memory for the lifetime of the `Asset`. When nothing is referencing it any more, the Python garbage collector will clean up the object and its data.

`Asset` instances are consolidated or "interned": if you ask for the same asset twice, you'll get the same instance back. Note that this is a performance optimization and should not be relied upon (do not do things like `asset1 is asset2`).

General Asset Interface

All assets inherit from `Asset`. It handles the background loading system and the data logistics.

class ppb.assetlib.**Asset**(*name*)

A resource to be loaded from the filesystem and used.

Meant to be subclassed, but in specific ways.

file_missing()

Called if the file could not be found, to produce a default value.

Subclasses may want to define this.

Called in the background thread.

background_parse(*data: bytes*)

Takes the data loaded from the file and returns the parsed data.

Subclasses probably want to override this.

Called in the background thread.

is_loaded()

Returns if the data has been loaded and parsed.

load(*timeout: float = None*)

Gets the parsed data.

Will block until the data is loaded.

Subclassing

`Asset` makes specific assumptions and is only suitable for loading file-based assets. These make the consolidation, background-loading, and other aspects of `Asset` possible.

You should really only implement two methods:

- `background_parse()`: This is called with the loaded data and returns an object constructed from that data. This is called from a background thread and its return value is accessible from `load()`

  This is an excellent place for decompression, data parsing, and other tasks needed to turn a pile of bytes into a useful data structure.

- `file_missing()`: This is called if the asset is not found. Defining this method surpresses `load()` from raising a `FileNotFoundError` and will instead call this, and `load()` will return what this returns.

  For example, *ppb.Image* uses this to produce the default square.

Concrete Assets

While `Asset` can load anything, it only produces bytes, limiting its usefulness. Most likely, you want a concrete subclass that does something more useful.

class ppb.**Image**(*name*)
Loads an image file and parses it into a form usable by the renderer.

class ppb.**Sound**(*name*)
Loads and decodes an image file. Wave and Ogg Vorbis are supported.

Asset Proxies and Virtual Assets

Asset Proxies and Virtual Assets are assets that implement the interface but either delegate to other Assets or are completely synthesized.

For example, *ppb.features.animation.Animation* is an asset proxy that delegates to actual *ppb.Image* instances.

class ppb.assetlib.**AbstractAsset**
The asset interface.

This defines the common interface for virtual assets, proxy assets, and real/file assets.

is_loaded()
Returns if the data is ready now or if *load()* will block.

load()
Get the data of this asset, in the appropriate form.

class ppb.**Circle**(*red: int*, *green: int*, *blue: int*)
A circle image of a single color.

class ppb.**Square**(*red: int*, *green: int*, *blue: int*)
A square image of a single color.

class ppb.**Triangle**(*red: int*, *green: int*, *blue: int*)
A triangle image of a single color.

All About Scenes

Scenes are the terrain where sprites act. Each game has multiple scenes and may transition at any time.

class ppb.**BaseScene**(*, *set_up: Callable = None*, *pixel_ratio: numbers.Number = 64*, ***kwargs*)

background_color = (0, 0, 100)
An RGB triple of the background, eg `(0, 127, 255)`

main_camera
An object representing the view of the scene that's rendered

add(*game_object: Hashable*, *tags: Iterable[T_co] = ()*) → None
Add a game_object to the scene.

game_object: Any GameObject object. The item to be added. tags: An iterable of Hashable objects. Values that can be used to

retrieve a group containing the game_object.

Examples: scene.add(MyGameObject())

scene.add(MyGameObject(), tags=("red", "blue")

get(*, *kind: Type[CT_co] = None*, *tag: Hashable = None*, ***kwargs*) → Iterator[T_co]
Get an iterator of GameObjects by kind or tag.

kind: Any type. Pass to get a subset of contained GameObjects with the given type.

tag: Any Hashable object. Pass to get a subset of contained GameObjects with the given tag.

Pass both kind and tag to get objects that are both that type and that tag.

Examples: scene.get(type=MyGameObject)

scene.get(tag="red")

scene.get(type=MyGameObject, tag="red")

remove(*game_object: Hashable*) → None
Remove the given object from the scene.

game_object: A game object.

Example: scene.remove(my_game_object)

sprite_layers() → Iterator[T_co]
Return an iterator of the contained Sprites in ascending layer order.

Sprites are part of a layer if they have a layer attribute equal to that layer value. Sprites without a layer attribute are considered layer 0.

This function exists primarily to assist the Renderer subsystem, but will be left public for other creative uses.

All About Sprites

Sprites are game objects.

In ppb all sprites are built from composition via mixins or subclassing via traditional Python inheritance. Sprite is provided as a default expectation used in ppb.

If you intend to build your own set of expectation, see BaseSprite.

Default Sprite

This is the class you should instantiate or subclass for your games unless you are changing the defaults.

class ppb.**Sprite**(***kwargs*)
The default Sprite class.

Sprite includes:

- BaseSprite
- SquareShapeMixin
- RenderableMixin
- RotatableMixin

New in 0.7.0: Use this in place of BaseSprite in your games.

bottom
The bottom side

center
The position of the center of the sprite

facing
The direction the "front" is facing

left
The left side

right
The right side

rotate(*degrees*)
Rotate the sprite by a given angle (in degrees).

rotation
The amount the sprite is rotated, in degrees

top
The top side

Note that `ppb.BaseSprite` is deprecated in favor of ppb.Sprite. Scheduled for removal in ppb v0.8.0.

Feature Mixins

These mixins are the various features already available in Sprite. Here for complete documentation.

class ppb.sprites.**RenderableMixin**
A class implementing the API expected by ppb.systems.renderer.Renderer.

You should include RenderableMixin before BaseSprite in your parent class definitions.

image = None
(*ppb.Image*): The image asset

class ppb.sprites.**RotatableMixin**
A simple rotation mixin. Can be included with sprites.

basis = Vector(0.0, -1.0)
The baseline vector, representing the "front" of the sprite

facing
The direction the "front" is facing

rotate(*degrees*)
Rotate the sprite by a given angle (in degrees).

rotation
The amount the sprite is rotated, in degrees

class ppb.sprites.**SquareShapeMixin**(***kwargs*)
A mixin that applies square shapes to sprites.

You should include SquareShapeMixin before ppb.sprites.BaseSprite in your parent classes.

bottom
The bottom side

center
The position of the center of the sprite

left
The left side

position = None
Just here for typing and linting purposes. Your sprite should already have a position.

right
The right side

size = 1
The width/height of the sprite (sprites are square)

top
The top side

Base Classes

The base class of Sprite, use this if you need to change the low level expectations.

class ppb.sprites.**BaseSprite**(***kwargs*)
The base Sprite class. All sprites should inherit from this (directly or indirectly).

The things that define a BaseSprite:

- The __event__ protocol (see ppb.eventlib.EventMixin)
- A position vector
- A layer

BaseSprite provides an __init__ method that sets attributes based on kwargs to make rapid prototyping easier.

layer = 0
The layer a sprite exists on.

position = Vector(0.0, 0.0)
(ppb.Vector): Location of the sprite

Internals

These classes are internals for various APIs included with mixins.

class ppb.sprites.**Side**(*parent: ppb.sprites.SquareShapeMixin*, *side: str*)
Acts like a float, but also has a variety of accessors.

bottom
Get the corner vector

center
Get the midpoint vector

left
Get the corner vector

right
Get the corner vector

top
Get the corner vector

Sound Effects

Sound effects can be triggered by sending an event:

```
def on_button_pressed(self, event, signal):
    signal(PlaySound(sound=ppb.Sound('toot.ogg')))
```

Both Ogg/Vorbis and WAV are supported audio formats.

Note: As is usual with assets, you should instantiate your *ppb.Sound* as soon as possible, such as at the class level.

Note: PyGame has fairly limited codec support. "Complex WAVE files" are not supported. Ogg/Opus appears to be unsupported. Additional formats and codecs might be supported but undocumented.

Reference

class ppb.events.**PlaySound**(*sound: ppb.assets.Asset*)
Fire to start a sound playing.

class ppb.**Sound**(*name*)
The asset to use for sounds. WAV and Ogg/Vorbis are supported.

Features

Features are additional libraries included with PursuedPyBear. They are not "core" in the sense that you can not write them youself, but they are useful tools to have when making games.

Animation

This is a simple animation tool, allowing individual frame files to be composed into a sprite animation, like so:

```
import ppb
from ppb.features.animation import Animation

class MySprite(ppb.BaseSprite):
    image = Animation("sprite_{1..10}.png", 4)
```

Multi-frame files, like GIF or APNG, are not supported.

Pausing

Animations support being paused and unpaused. In addition, there is a "pause level", where multiple calls to `pause()` cause the animation to become "more paused". This is useful for eg, pausing on both scene pause and effect.

```
import ppb
from ppb.features.animation import Animation

class MySprite(ppb.BaseSprite):
    image = Animation("sprite_{1..10}.png", 4)

    def on_scene_paused(self, event, signal):
        self.image.pause()

    def on_scene_continued(self, event, signal):
        self.image.unpause()

    def set_status(self, frozen):
        if frozen:
            self.image.pause()
        else:
            self.image.unpause()
```

Reference

class ppb.features.animation.**Animation**(*filename*, *frames_per_second*)
An "image" that actually rotates through numbered files at the specified rate.

__init__(*filename*, *frames_per_second*)

Parameters

- **filename** (*str*) – A path containing a `{2..4}` indicating the frame number
- **frames_per_second** (*number*) – The number of frames to show each second

copy()
Create a new Animation with the same filename and framerate. Pause status and starting time are reset.

current_frame
Compute the number of the current frame (0-indexed)

load()
Get the current frame path.

pause()
Pause the animation.

unpause()
Unpause the animation.

Two Phase Updates

A system for two phase updates: Update, and Commit.

class ppb.features.twophase.**Commit**
Fired after Update.

class ppb.features.twophase.**TwoPhaseMixin**
Mixin to apply to objects to handle two phase updates.

on_commit(*event*, *signal*)
Commit changes previously staged.

stage_changes(***kwargs*)
Stage changes for the next commit.

These are just properties on the current object to update.

class ppb.features.twophase.**TwoPhaseSystem**(**_)
Produces the Commit event.

Loading Screens

The loadingscene feature provides base classes for loading screens. *BaseLoadingScene* and its children all work by listening to the asset system and when all known assets are loaded, continuing on.

class ppb.features.loadingscene.**BaseLoadingScene**(***kwargs*)
Handles the basics of a loading screen.

get_progress_sprites()
Initialize the sprites in the scene, yielding the ones that should be tagged with `progress`.

Override me.

next_scene = None
The scene to transition to when loading is complete. May be a type or an instance.

update_progress(*progress*)
Updates the scene with the load progress (0->1).

Override me.

class ppb.features.loadingscene.**ProgressBarLoadingScene**(***kwargs*)
Assumes that a simple left-to-right progress bar composed of individual sprites is used.

Users should still override `get_progress_sprites()`.

loaded_image = None
Image to use for sprites in the "loaded" state (left side)

unloaded_image = <ppb.flags.DoNotRender object>
Image to use for sprites in the "unloaded" state (right side)

update_progress(*progress*)
Looks for sprites tagged `progress` and sets them to "loaded" or "unloaded" based on the progress.

The "progress bar" is assumed to be horizontal going from left to right.

2.5.5 Discussion

Discussion is a place to talk about the history and why of specific parts of `ppb`. These items can be heavily technical so primarily intended for advanced users.

Python Module Index

p

Index

Symbols

A

B

C

F

G

I

K

L

M

N

O

P

Q

R

S

T

U