
PursuedPyBear Documentation

Piper Thunstrom

Apr 20, 2020

Contents:

1	A Game Engine	3
2	Guiding Principles	5
2.1	Education Friendly	5
2.2	Idiomatic Python	5
2.3	Object Oriented and Event Driven	5
2.4	Hardware Library Agnostic	5
2.5	Fun	6
	Python Module Index	31
	Index	33

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)`.

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. 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
```

Additionally, on Linux only you must install the SDL library:

Debian, Ubuntu:

```
sudo apt install libsdl2-2.0-0 libsdl2-mixer-2.0-0 libsdl2-image-2.0-0 libsdl2-gfx-1.0-0
```

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 our `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)
```

(continues on next page)

(continued from previous page)

```
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 be 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):
        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.

free(object)

Called by `__del__()` if the data was loaded.

Meant to free any resources held outside of Python.

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 three 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 suppresses `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.

- `free()`: This is to clean up any resources that would not normally be cleaned up by Python's garbage collector. If you are integrating external libraries, you may need this.

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. A variety of formats 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`:

- 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

GameEngine

The `GameEngine` is the literal beating heart of `ppb`: It publishes the event queue, is the source of the `Idle` event, and is the root container of the object tree.

Some of the engine of the API is definitely intended for advanced users. Use the various methods of `GameEngine` with caution.

```
class ppb.GameEngine (first_scene:          Type[CT_co],          *,          basic_systems=(<class
                                'ppb.systems.renderer.Renderer'>, <class 'ppb.systems.clocks.Updater'>,
                                <class 'ppb.systems.inputs.EventPoller'>, <class
                                'ppb.systems.sound.SoundController'>, <class
                                'ppb.assetlib.AssetLoadingSystem'>), systems=(), scene_kwargs=None,
                                **kwargs)
```

The core component of ppb.

To use the engine directly, treat it as a context manager:

```
with GameEngine(BaseScene, **kwargs) as ge:
    ge.run()
```

Parameters

- **first_scene** (*Type*) – A *BaseScene* type.
- **basic_systems** (*Iterable[systemslib.System]*) – :class:systemslib.Systems that are considered the “default”. Includes: *Renderer*, *Updater*, *EventPoller*, *SoundController*, *AssetLoadingSystem*.
- **systems** (*Iterable[systemslib.System]*) – Additional user defined systems.
- **scene_kwargs** (*Dict[str, Any]*) – Keyword arguments passed along to the first scene.
- **kwargs** – Additional keyword arguments. Passed to the systems.

Warning: Passing in your own *basic_systems* can have unintended consequences. Consider passing via *systems* parameter instead.

activate (*next_scene: dict*)

Instantiates and sets up a new scene.

Parameters *next_scene* – A dictionary with the keys:

- “scene_class”: A *BaseScene* type.
- “args”: A *list* of positional arguments.
- “kwargs”: A *dict* of keyword arguments.

current_scene

The top of the scene stack.

Returns The currently running scene.

Return type *ppb.BaseScene*

loop_once ()

Iterate once.

If you’re embedding ppb in an external event loop call once per loop.

main_loop ()

Loop forever.

If you’re embedding ppb in an external event loop you should not use this method. Call *GameEngine.loop_once()* instead.

on_quit (*quit_event: ppb.events.Quit, signal: Callable[[Any], None]*)

Shut down the event loop.

Do not call this method directly. It is called by the GameEngine when a *Quit* event is fired.

on_replace_scene (*event: ppb.events.ReplaceScene, signal*)

Replace the running scene with a new one.

Do not call this method directly. It is called by the GameEngine when a *ReplaceScene* event is fired.

on_start_scene (*event: ppb.events.StartScene, signal: Callable[[Any], None]*)

Start a new scene. The current scene pauses.

Do not call this method directly. It is called by the GameEngine when a *StartScene* event is fired.

on_stop_scene (*event: ppb.events.StopScene, signal: Callable[[Any], None]*)

Stop a running scene. If there's a scene on the stack, it resumes.

Do not call this method directly. It is called by the GameEngine when a *StopScene* event is fired.

publish ()

Publish the next event to every object in the tree.

register (*event_type: Union[Type[CT_co], ellipsis], callback: Callable[[], Any]*)

Register a callback to be applied to an event at time of publishing.

Primarily to be used by subsystems.

The callback will receive the event. Your code should modify the event in place. It does not need to return it.

Parameters

- **event_type** – The class of an event.
- **callback** – A callable, must accept an event, and return no value.

Returns

run ()

Begin the main loop.

If you have not entered the *GameEngine*, this function will enter it for you before starting.

Example:

```
GameEngine(BaseScene, **kwargs).run()
```

signal (*event*)

Add an event to the event queue.

Thread-safe.

You will rarely call this directly from a *GameEngine* instance. The current *GameEngine* instance will pass it's signal method as part of publishing an event.

start ()

Starts the engine.

Called by *GameEngine.run()* before *GameEngine.main_loop()*.

You shouldn't call this yourself unless you're embedding ppb in another event loop.

start_systems ()

Initialize and enter the systems.

walk()

Walk the object tree.

Publication order: The *GameEngine*, the *System* list, the current *BaseScene*, then finally the *Sprite* objects in the current scene.

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')))
```

The following sound formats are supported:

- OGG (with both Vorbis and Opus)
- FLAC
- MP3
- WAV
- AIFF
- MOD
- VOC

Additionally, MIDI *may* be supported.

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

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. A variety of file formats are supported.

Features

Features are additional libraries included with PursuedPyBear. They are not “core” in the sense that you can not write them yourself, 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.

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

__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.

Principles and Values

PursuedPyBear is a principles driven project. From its earliest days, it's been guided by a set of ideals that suggest the shape and form of any problem we encounter. From our primary focus on students and learners to our embracing change of the code and our principles.

We have identified four principles as the root of all the others:

Students and Learners First

Our first commitment is to the new programmers and game makers who have made `ppb` their tool of choice. Whether introduced to them by a teacher or discovered on their own, we care about their experiences primarily. However, we won't forget educators, professional developers, or hobbyists while doing so. Our success is most easily measured by the diversity of our community, and that requires a focus on the early stages of use.

Creativity without Limits

The only limitation we accept is the limitations that come from the platform we've chosen: that being that the only limitation to what you can apply with `ppb` is what the developer is capable of what Python is capable of.

This is why we consider being a code first engine so critical to our design. It's one of many reasons we build with Python first.

Fun

We believe tools that are a joy to use are more likely to be picked up for the long term. Being fun means more than just joyful discovery. We care that complexity is neatly hidden until it's necessary to be addressed. We want to reduce "warts" in the API, increase overall discoverability, and allow you to work playfully.

Radical Acceptance

Over time, the needs of `ppb` as a project and as a community have changed. It started as a solo project to encourage reuse of common patterns in one developer's workflows. It's now an education-focused community with a team of active developers and a couple dozen contributors. As the needs shift, so too do our principles and processes.

By being willing to do hard work and experiment with improvements in code, we've been able to build better software. `ppb` has seen three major architectures since it started. And this document replaces a previous version of our principles. Those principles were:

- Education Friendly
- Idiomatic Python
- Object Oriented and Event Driven
- Hardware Library Agnostic
- Fun

You'll notice that some of them are still with us in this document. Others you'll find in the child documents of each of the ones explored here. Some are less principles and more design decisions.

In general, we are always willing to field the idea of “maybe we should do it this way?” Those questions, even when the answer is no, keep us thinking about how things are and how they could be.

Ultimately, ppb is aspirational as a project, and our principles reflect that.

Student First

PursuedPyBear is, above all other usages, a tool for learning. We continually find ways to reduce the amount of previous knowledge is required to get to your first functioning video game. The greatest example of this is the evolution of ppb start up throughout time:

Originally, ppb was a strict MVC framework with required dependency injection and little concept of sensible defaults. You had to know what each part of the system was, instantiate it, and then pass it to the next component.

We kept the dependency injection but rebuilt the engine to have strong opinions and defaults at every level of the system. However, you had to know what a context manager was, and how to use one.

Today, we can make a functional game in 15 lines of code, and you never need to see the underlying context manager.

Progressive Revealing of Complexity

We want to encourage exploration and flexibility of the underlying tool, and one of the ways we achieve this is through only revealing the complexity of the tool at the point you must understand it to do something. Our “Hello World!” example requires only understanding how to invoke functions and how to write your own: the fundamental building blocks of Python programming. In the next hour of exploration, it's possible to learn what objects are, how classes are defined and using them yourself. And from there, you can begin to learn more complex features of Python.

Whenever possible, we prefer to provide powerful and sensible defaults, but with as many options for advanced users as possible.

No Apologies

Every language and tool ends up with a number of quirks known as “wats”. In ppb we tend to call them “warts”: they're places where the knowledge you have of how a system works is thrown a curve ball that requires reassessing what you know. There are popular wat talks for both Python and Javascript to get a feel for what we mean.

One way this bears out is that no matter what level your knowledge of ppb, learning something new should only add to that knowledge, not require reassessment.

We also try to reduce the number of times a user is forced to ask “why is it like this and not like that?” Things that are like messages should use the event queue. State should be contained by objects at the right level of abstraction. Things should fit the model.

Creativity Without Limits

Our focus on creativity without limits is about supporting users at all skill levels, and to help guide them from their first lines of code through contributing to open source.

From a game perspective, we don't want to discourage any genre of game against any other. We don't want to discourage any given scale.

That isn't to mean we don't have some limitations: `ppb` is a 2d sprite-based engine, it's built in Python, and it is code first.

Code First

The primary reason we want `ppb` to be code first is because it allows the primary long term limitations set on users is the limitations of the Python language itself.

Code first also means that learning `ppb` means learning patterns that can be applied to other kinds of software. A student who learns with `ppb` shouldn't need to ask "what comes next", the answer should be apparent: Whatever the next project is that interests them.

No Early Optimization

This is one of those general rules of software development, but it's something that creates limitations. If we over optimize our toolset for one genre of video game, it adds friction to others. New features should be generally applicable or explicitly optional.

The primary example of this is in the basic setup for `ppb` as a simple event loop with the update pattern at its core. This is because it's the most generally applicable pattern we have available. We provide a multi-phase update system in features for games that need the ability to stage updates instead of immediately shifting the state.

Support All Users

We're Students First, but students aren't students forever, and we want `ppb` to grow with them. From their first tutorials through to their first shipped video game, and hopefully: to their first open source contribution. This is about making sure the resources and community are there to help develop `ppb` users.

From the user perspective, clear tutorials and example code. Open discussions about the design of the system. Type hinting to allow the tools to help guide. Progressive revelation of complexity. All of this is meant to guide a user from student to pro.

Once they're ready to contribute we care about well defined processes and guidelines. A strong description of how documentation is laid out. Where code lives and why.

Fun

This one is fairly self evident: Playing games is fun. Making games should be fun. And doing both with friends is better.

Delightful to Use

One of the core ideas that embedded early in the project is that if some piece of `ppb` wasn't fun to use, we should redo it. Our goal with `ppb` is a genuinely delightful API. Much of this comes from reducing friction in use, but once in a while it's about making a change aimed at giving more expressive constructions.

One of the changes that we did that demonstrates this is making class attributes the default way to initialize state. Combined with a powerful `__init__` it has made rapid prototyping faster and more fun for end users.

Encourage Playful Experimentation

This principle draws from many places: Our student focus, our focus on a delightful interface, and the observation that people learn through play. As such we have spent a lot of focus on the first few hours of game development. From our five minute live demos and various tutorials that are ripe for experimentation we want testing ideas to be painless and recovering from a misstep to be as inexpensive as possible.

Community Focused

While ppb started as a solo project, it's growth has been built on community. Decisions are made through discussion and offering ideas is rarely discouraged.

Beyond the development of the project, one of our long term focuses is game distribution. Part of the fun of making the small games ppb excel at is sharing them with others.

Radical Acceptance

While we do think the greater idea of radical acceptance is important, with regard to ppb, radical acceptance is about inclusion, experimentation, and willingness to question our assumptions.

Accept Significant Change

We don't want to be afraid of change. PursuedPyBear is a project about the API, and how humans interact with computers changes over times. We shouldn't be afraid to abandon API decisions if they stop proving useful.

The "back end" of ppb has changed significantly on four occasions so far, changing when some limitation was reached. Originally strictly powered by dependency injection, we learned that sensible defaults are incredibly important. That shifted to a single monolithic API that ran all code directly in line. Then we peeled out the first few subsystems. They were still called directly, but you could work on them separately from the engine itself. Then we moved to the `Idle` event and messaging as the way to interact between subsystems and the engine itself. In time, even this pattern may prove limiting and be changed.

Inclusion

PursuedPyBear started as the solo project of a trans woman with a non-standard education background. As it's grown, we have sought out and encouraged contributors from diverse backgrounds. We have a [code of conduct](#) that covers all participation in the project.

We like being a diverse project, and we will protect the environment that let's it be that way.

Education

While we are a tool for education, we acknowledge that not all learners learn the same way. The author was home schooled and self taught software after college. Many of the teachers who advise the project come from more traditional education backgrounds.

We seek to support learners no matter their education path.

Race

Similar to education, race should not be an obstacle to using or contributing to `ppb`. The maintainers recognize that while that might be a thing we can obtain in the project, society is racist and we must work to be anti-racist in how we manage the project and community.

Gender

The current team of maintainers are all trans feminine. We seek out women and gender minorities to contribute to the project. We embrace all genders and hope to keep `ppb` the kind of community where it is safe to be who you are.

Be Willing To Try

When someone is willing to do the work for an idea the rest of the team isn't sure about, let them take a chance at it. Usage will tell us if the solution is appropriate, not our personal biases.

The Asset System

The asset system (`ppb.assetlib`) is probably one of the more involved parts of the PPB engine, most likely because it is one of the very few places where multithreading takes place.

It was made to help give a handle on the problems surrounding loading data into games and the management of said data.

To that end, we had several goals when we built the asset system:

- Allow declaring resources at a fairly abstract level
- Optimistically load resources in the background as soon as possible
- Provide a layer of abstraction between how data is loaded and the use of that data

As part of this, we also built the VFS library (`ppb.vfs`), which treats the Python module import system as a filesystem and allows loading data from it, to make it clear where and how resource files should be added to a project, and provide all the flexibility of the Python module system.

Concepts

Out of this, we define a few high-level concepts:

- Asset: Some kind of way data is loaded & parsed. Usually the result is some internal engine data type.
- Real or File Asset: Loads data from the VFS (such as `ppb.Image`)
- Virtual Asset: Synthesizes data from nothing (such as `ppb.assets.Circle`)
- Proxy Asset: Wraps other asset types (such as `ppb.features.animation.Animation`)

The idea is that the place where the asset is used does not care what kind of asset is used, only that it produces the right kind of data—nothing in the world can make the renderer accept a `ppb.Sound`.

Implementation

So how did we do this?

A lot of the heavy lifting is provided by the `concurrent.futures` package from the standard library. On top of this, `AssetLoadingSystem` and `Asset` cooperate to implement background file reading. After the data is read, it is handed to the instance and processed into its final form.

Effort is taken to deduplicate assets: If two places refer to the same asset, it is normalized to the same instance. This reduces both load times and memory usage.

A minor wrinkle in this is that assets are defined before the engine starts. The asset system does not actually begin loading data until the engine and `AssetLoadingSystem` are initialized. This means that there's no problems delivering events and asset implementations know that initialization has happened.

Usage

None of this explains how you use the asset system for yourself.

Defining Assets

First of all, you have to define for yourself what kind of data the asset will produce. This is usually some kind of data object to be consumed.

Then, you make an `Asset` subclass. There's a few methods of note for overriding:

- `Asset.background_parse()`: Do the actual parsing. Accepts the bytes loaded from the file, and returns the data object that the asset is wrapping.
- `Asset.file_missing()`: If defined, this will be called if the file is not found, and is expected to return a synthesized stand-in object. If not defined, `Asset.load()` will raise an error.
- `Asset.free()`: Handles cleanup in the case where resources need to be explicitly cleaned. Note that because this is called in the context of `__del__()`, care must be taken around referring to globals or other modules.

At the point of use, all you need to do is call `Asset.load()` and you will get the object created by the asset. This will block if the background processing is incomplete.

Proxy Assets

Proxy assets are simply assets that wrap other, more concrete assets. One example of this is `ppb.features.animation`, where `Animation` wraps multiple `Image` instances.

Writing your own proxy asset just means returning the results of your inner asset's `load()` from your own.

p

- `ppb.features.animation`, [22](#)
- `ppb.features.loadingscene`, [24](#)
- `ppb.features.twophase`, [24](#)
- `ppb.sprites`, [17](#)

Symbols

`__init__()` (*ppb.features.animation.Animation method*), 23

A

`AbstractAsset` (*class in ppb.assetlib*), 16
`activate()` (*ppb.GameEngine method*), 20
`add()` (*ppb.BaseScene method*), 16
`Animation` (*class in ppb.features.animation*), 23
`Asset` (*class in ppb.assetlib*), 14

B

`background_color` (*ppb.BaseScene attribute*), 16
`background_parse()` (*ppb.assetlib.Asset method*), 15
`BaseLoadingScene` (*class in ppb.features.loadingscene*), 24
`BaseScene` (*class in ppb*), 16
`BaseSprite` (*class in ppb.sprites*), 19
`basis` (*ppb.sprites.RotatableMixin attribute*), 18
`bottom` (*ppb.Sprite attribute*), 17
`bottom` (*ppb.sprites.Side attribute*), 19
`bottom` (*ppb.sprites.SquareShapeMixin attribute*), 18
`ButtonPressed` (*class in ppb.events*), 12
`ButtonReleased` (*class in ppb.events*), 12

C

`center` (*ppb.Sprite attribute*), 17
`center` (*ppb.sprites.Side attribute*), 19
`center` (*ppb.sprites.SquareShapeMixin attribute*), 18
`Circle` (*class in ppb*), 16
`Commit` (*class in ppb.features.twophase*), 24
`copy()` (*ppb.features.animation.Animation method*), 23
`current_frame` (*ppb.features.animation.Animation attribute*), 23
`current_scene` (*ppb.GameEngine attribute*), 20

F

`facing` (*ppb.Sprite attribute*), 17

`facing` (*ppb.sprites.RotatableMixin attribute*), 18
`file_missing()` (*ppb.assetlib.Asset method*), 14
`free()` (*ppb.assetlib.Asset method*), 15

G

`GameEngine` (*class in ppb*), 19
`get()` (*ppb.BaseScene method*), 16
`get_progress_sprites()` (*ppb.features.loadingscene.BaseLoadingScene method*), 24

I

`Idle` (*class in ppb.events*), 12
`Image` (*class in ppb*), 15
`image` (*ppb.sprites.RenderableMixin attribute*), 18
`is_loaded()` (*ppb.assetlib.AbstractAsset method*), 16
`is_loaded()` (*ppb.assetlib.Asset method*), 15

K

`KeyPressed` (*class in ppb.events*), 12
`KeyReleased` (*class in ppb.events*), 12

L

`layer` (*ppb.sprites.BaseSprite attribute*), 19
`left` (*ppb.Sprite attribute*), 17
`left` (*ppb.sprites.Side attribute*), 19
`left` (*ppb.sprites.SquareShapeMixin attribute*), 18
`load()` (*ppb.assetlib.AbstractAsset method*), 16
`load()` (*ppb.assetlib.Asset method*), 15
`load()` (*ppb.features.animation.Animation method*), 23
`loaded_image` (*ppb.features.loadingscene.ProgressBarLoadingScene attribute*), 24
`loop_once()` (*ppb.GameEngine method*), 20

M

`main_camera` (*ppb.BaseScene attribute*), 16
`main_loop()` (*ppb.GameEngine method*), 20
`MouseMotion` (*class in ppb.events*), 12

N

`next_scene` (*ppb.features.loadingscene.BaseLoadingScene* attribute), 24

O

`on_commit` () (*ppb.features.twophase.TwoPhaseMixin* method), 24

`on_quit` () (*ppb.GameEngine* method), 20

`on_replace_scene` () (*ppb.GameEngine* method), 21

`on_start_scene` () (*ppb.GameEngine* method), 21

`on_stop_scene` () (*ppb.GameEngine* method), 21

P

`pause` () (*ppb.features.animation.Animation* method), 23

`PlaySound` (class in *ppb.events*), 13, 22

`position` (*ppb.sprites.BaseSprite* attribute), 19

`position` (*ppb.sprites.SquareShapeMixin* attribute), 18

`ppb.features.animation` (module), 22

`ppb.features.loadingscene` (module), 24

`ppb.features.twophase` (module), 24

`ppb.sprites` (module), 17

`PreRender` (class in *ppb.events*), 12

`ProgressBarLoadingScene` (class in *ppb.features.loadingscene*), 24

`publish` () (*ppb.GameEngine* method), 21

Q

`Quit` (class in *ppb.events*), 12

R

`register` () (*ppb.GameEngine* method), 21

`remove` () (*ppb.BaseScene* method), 17

`Render` (class in *ppb.events*), 12

`RenderableMixin` (class in *ppb.sprites*), 18

`ReplaceScene` (class in *ppb.events*), 12

`right` (*ppb.Sprite* attribute), 17

`right` (*ppb.sprites.Side* attribute), 19

`right` (*ppb.sprites.SquareShapeMixin* attribute), 18

`RotatableMixin` (class in *ppb.sprites*), 18

`rotate` () (*ppb.Sprite* method), 18

`rotate` () (*ppb.sprites.RotatableMixin* method), 18

`rotation` (*ppb.Sprite* attribute), 18

`rotation` (*ppb.sprites.RotatableMixin* attribute), 18

`run` () (*ppb.GameEngine* method), 21

S

`SceneContinued` (class in *ppb.events*), 13

`ScenePaused` (class in *ppb.events*), 13

`SceneStarted` (class in *ppb.events*), 13

`SceneStopped` (class in *ppb.events*), 13

`Side` (class in *ppb.sprites*), 19

`signal` () (*ppb.GameEngine* method), 21

`size` (*ppb.sprites.SquareShapeMixin* attribute), 18

`Sound` (class in *ppb*), 22

`Sprite` (class in *ppb*), 17

`sprite_layers` () (*ppb.BaseScene* method), 17

`Square` (class in *ppb*), 16

`SquareShapeMixin` (class in *ppb.sprites*), 18

`stage_changes` () (*ppb.features.twophase.TwoPhaseMixin* method), 24

`start` () (*ppb.GameEngine* method), 21

`start_systems` () (*ppb.GameEngine* method), 21

`StartScene` (class in *ppb.events*), 12

`StopScene` (class in *ppb.events*), 13

T

`top` (*ppb.Sprite* attribute), 18

`top` (*ppb.sprites.Side* attribute), 19

`top` (*ppb.sprites.SquareShapeMixin* attribute), 18

`Triangle` (class in *ppb*), 16

`TwoPhaseMixin` (class in *ppb.features.twophase*), 24

`TwoPhaseSystem` (class in *ppb.features.twophase*), 24

U

`unloaded_image` (*ppb.features.loadingscene.ProgressBarLoadingScene* attribute), 24

`unpause` () (*ppb.features.animation.Animation* method), 23

`Update` (class in *ppb.events*), 12

`update_progress` () (*ppb.features.loadingscene.BaseLoadingScene* method), 24

`update_progress` () (*ppb.features.loadingscene.ProgressBarLoadingScene* method), 24

W

`walk` () (*ppb.GameEngine* method), 21