PursuedPyBear Documentation

Piper Thunstrom

Oct 30, 2020

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. 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`.

Note: This document explains the practical and technical details of `ppb`. If you want to know more about what it is, its goals, or why you should use it, check out this page or our website.

CHAPTER 1

Using these docs

Our documentation is separated into practical steps and theoretical knowledge. The docs are made up of Tutorials, which allow you to learn quickly, and guides which bring you to a specific goal. We also have a standard API reference.

1.1 As a New User

We recommend that new users start with the *Getting Started* Guide. This document assumes that you have basic programming knowledge. If you don't, check out the Python docs at https://docs.python.org.

1.2 As an Experienced User

Experienced users will likely want to start with our *API Reference* or our *How To: The ppb Cookbook*

1.2.1 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 libsdl2-ttf-2.0-0
```

Fedora, CentOS, RHEL

```
sudo dnf install SDL2 SDL2_ttf SDL2_image SDL2_gfx SDL2_mixer libmodplug
```

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.Sprite):
    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.Sprite):
    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.Sprite):
    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.Sprite):
    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.Sprite):


    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)
```

(continues on next page)

(continued from previous page)

```
        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.Sprite):
    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.Sprite):

    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.Sprite):

    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*.

1.2.2 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.

1.2.3 How To: The ppb Cookbook

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

1.2.4 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.

A python game framework.

PursuedPyBear is object oriented and event driven. Practically, this means that most of your code will be organized into classes. Game objects in *ppb* are *Sprite* instances, which get contained in *BaseScenes*. In turn, the

GameEngine contains the scenes and `Systems`. `Events` are defined as simple classes and event handlers are based on their names.

The classes, modules, and methods exported directly are the most used parts of the library and intended to be used by users at all levels (barring make_engine). Advanced features tend to be in their own modules and subpackages.

Exports:

- `Vector`
- *BaseScene*
- *Circle*
- *Image*
- *Sprite*
- *Square*
- *Sound*
- *Triangle*
- `events`
- *Font*
- *Text*
- `directions`

ppb.**run**(*setup: Callable[[ppb.scenes.BaseScene], None] = None, *, log_level=30, starting_scene=<class 'ppb.scenes.BaseScene'>, title='PursuedPyBear', **engine_opts*)

Run a game.

This is the default entry point for ppb games.

Sample usage:

```
import ppb

def setup(scene):
    scene.add(ppb.Sprite())

ppb.run(setup)
```

Alternatively:

```
import ppb

class Game(ppb.BaseScene):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.add(ppb.Sprite())

ppb.run(starting_scene=Game)
```

See the *Getting Started* guide for a more complete guide to building games.

All parameters are optional.

Parameters

- **setup** (*Callable[[BaseScene], None]*) – Called with the first scene to allow initialization of your game.

- **log_level** – The logging level from `logging()` to send to the console.
- **starting_scene** (*type*) – A scene class to use. Defaults to `BaseScene`
- **title** (*str*) – The title of the rendered window.
- **engine_opts** – Additional keyword arguments passed to the `GameEngine`.

ppb.**make_engine**(*setup: Callable[[ppb.scenes.BaseScene], None] = None, *, starting_scene=<class 'ppb.scenes.BaseScene'>, title='PursedPyBear', **engine_opts*)

Setup a *GameEngine*.

This function exists for third party modules to use the same code paths as *run()* for setting up their engine. If you want to instantiate your own engine, you can do so directly using the *constructor*.

Parameters

- **setup** (*Callable[[BaseScene], None]*) – Called with the first scene to allow initialization of your game.
- **starting_scene** (*type*) – A scene class to use. Defaults to `BaseScene`
- **title** (*str*) – The title of the rendered window.
- **engine_opts** – Additional keyword arguments passed to the `GameEngine`

Returns A GameEngine instance.

Events

Events are the primary way objects in ppb communicate. This module contains the events defined by various systems in ppb.

To respond to an event your event handler should be snake_cased `on_event_name` and accept an event instance and a signal function as parameters. Example:

```
class MySprite(Sprite):
    def on_update(self, event: Update, signal):
        . . .
```

The *signal()* function accepts one parameter: an instance of an event class. You are not limited to predefined event types, but can provide arbitrary instances.

Events as defined here are `dataclasses.dataclass()`, but ppb does not expect dataclasses; they are just a simple way to quickly define new events. The name of the handler will always be based on the name of the class, with the TitleCase name of the class converted to on_event_name function. The instance passed to *signal()* will be passed to all the event handlers as the event parameter.

Basic Events

These events are the basic events you'll want to build your games. You can make a variety of games using just them.

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

A simulation tick.

Respond via `on_update` to advance the simulation of your game objects. Movement and other things that happen "over time" are best implemented in your on_update methods.

scene = None

The currently running scene.

time_delta = None
Seconds since last Update

class ppb.events.**PreRender**(*time_delta: float*, *scene: ppb.scenes.BaseScene = None*)
The Renderer is preparing to render.

PreRender is called before every frame is rendered. Things that are strictly for display purposes (like the text of a score board or the position of the camera) should happen on_pre_render.

scene = None
The currently running scene.

time_delta = None
Seconds since last PreRender.

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

The button is a MouseButton instance.

This represents the button in the active state. For acting when a button is released see *ButtonReleased*.

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

The button is a MouseButton instance.

This represents the button in the inactive state. For acting when a button is clicked see *ButtonPressed*.

button = None
A mouse button: Primary, Secondary, or Tertiary

position = None
The game-world position of the event.

scene = None
The currently running scene.

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

The buttons are defined in ppb.keycodes.

This represents the key entering an active state, to respond to when a key is released see *KeyReleased*.

key = None
A KeyCode flag.

mods = None
A set of KeyCodes

scene = None
The currently running scene

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

The buttons are defined in ppb.keycodes.

This represents the key entering an inactive state, to respond to when a key is pressed see *KeyPressed*.

key = None
A KeyCode flag.

mods = None
A set of KeyCodes

scene = None
The currently running scene

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

If something should be tracking the mouse, this is the event to listen to.

buttons = None
The state of the mouse buttons.

delta = None
The change in position since the last *MouseMotion* event.

position = None
The game-world location of the mouse cursor.

scene = None
The currently running scene.

Command Events

These events are used in your code to achieve certain effects.

class ppb.events.**Quit**(*scene: ppb.scenes.BaseScene = None*)
A request to quit the program.

Fired in response to a close request from the OS, but can be signaled from inside one of your handlers to end the program.

For example:

```
def on_update(self, event, signal):
    signal(Quit())
```

This will stop the engine.

Respond with on_quit to perform any shut down tasks (like saving data.)

scene = None
The currently running scene.

class ppb.events.**StartScene**(*new_scene: Union[ppb.scenes.BaseScene, Type[ppb.scenes.BaseScene]], kwargs: Dict[str, Any] = None, scene: ppb.scenes.BaseScene = None*)
An object requested a new scene to be started.

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}*

Warning: In general, you should not respond to `StartScene`, if you want to respond to a new scene, see `SceneStarted`.

kwargs = None
Keyword arguments to be passed to a scene type.

new_scene = None
A `BaseScene` class or instance

scene = None
The currently running scene.

class ppb.events.**ReplaceScene**(*new_scene: Union[ppb.scenes.BaseScene, Type[ppb.scenes.BaseScene]], kwargs: Dict[str, Any] = None, scene: ppb.scenes.BaseScene = None*)
An object requested a new scene to replace it.

`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(ReplaceScene(MyScene(player=player))*
- *signal(ReplaceScene(new_scene=ReplacementScene, kwargs={"player": player}))*

Warning: In general, you should not respond to `ReplaceScene`, if you want to respond to a new scene, see `SceneStarted`.

kwargs = None
Keyword arguments to be passed to a scene type.

new_scene = None
A `BaseScene` class or instance

scene = None
The currently running scene.

class ppb.events.**StopScene**(*scene: ppb.scenes.BaseScene = None*)
An object has requested the current scene be stopped.

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.

> **Warning:** In general, you should not respond to *StopScene*, if you want to respond to a scene ending, see *SceneStopped*.

scene = None
The scene that is stopping.

class ppb.events.**PlaySound**(*sound: ppb.assetlib.Asset*)
An object requested a sound be played.

Signal in an event handler to have a sound played.

Example:

```
signal(PlaySound(my_sound))
```

sound = None
A Sound asset.

Scene Transition Events

These are events triggered during 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*)
A new scene has started running.

This is delivered to a Scene shortly after it starts.

Responding to SceneStarted is a good choice for ppb.systems that change behavior based on the running scene, or if you have start up work that requires the initial state to be set before it happens.

The scene lifetime events happen in the following order:

1. Always: *SceneStarted*
2. Optionally, Repeatable: *ScenePaused*
3. Optionally, Repeatable: *SceneContinued*
4. Optionally: *SceneStopped*

scene = None
The scene that is starting.

class ppb.events.**ScenePaused**(*scene: ppb.scenes.BaseScene = None*)
A scene that is running is being paused to allow a new scene to run.

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*.

scene = None
The scene that has paused.

class ppb.events.**SceneContinued**(*scene: ppb.scenes.BaseScene = None*)
A scene that had been paused has continued.

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*.

scene = None
The scene that is resuming.

class ppb.events.**SceneStopped**(*scene: ppb.scenes.BaseScene = None*)
A scene is being stopped and will no longer be available.

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

This is technically an optional event, as not all scenes in the stack will receive a *SceneStopped* event if a *Quit* event was sent.

This is the end of the scene lifetime, see *SceneStarted*.

scene = None
The scene that is stopping.

Engine Events

These are additional events from the engine mostly for advanced purposes.

class ppb.events.**Idle**(*time_delta: float*, *scene: ppb.scenes.BaseScene = None*)
A complete loop of the GameEngine main loop has finished.

This is an engine plumbing event to pump timing information to subsystems.

scene = None
The currently running scene.

time_delta = None
Seconds since last Idle.

class ppb.events.**Render**(*scene: ppb.scenes.BaseScene = None*)
The Renderer is rendering.

> **Warning:** In general, responses to *Render* will not be reflected until the next render pass. If you want changes to effect this frame, see *PreRender*

scene = None
The currently running scene.

class ppb.events.**AssetLoaded**(*asset: ppb.assetlib.Asset*, *total_loaded: int*, *total_queued: int*)
An asset has finished loading.

asset = None
A *Asset*

total_loaded = None
The total count of loaded assets.

total_queued = None
The number of requested assets still waiting.

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.

load(*timeout: float = None*)
Gets the parsed data.

Will block until the data is loaded.

is_loaded()
Returns if the data has been loaded and parsed.

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.

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 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.

- `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(*timeout: float = None*)
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.

Game Object Model

The Game Object Model.

class ppb.gomlib.**Children**
A container for game objects.

Supports tagging.

add(*child: Hashable*, *tags: Iterable[Hashable] = ()*) → Hashable
Add a child.

Parameters

- **child** – Any Hashable object. The item to be added.
- **tags** – An iterable of Hashable objects. Values that can be used to retrieve a group containing the child.

Examples:

```
children.add(MyObject())

children.add(MyObject(), tags=("red", "blue")
```

get(*, *kind: Type[CT_co] = None*, *tag: Hashable = None*, **_) → Iterator[T_co]
Iterate over the objects by kind or tag.

Parameters

- **kind** – Any type. Pass to get a subset of contained items with the given type.
- **tag** – Any Hashable object. Pass to get a subset of contained items with the given tag.

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

Examples:

```
children.get(type=MyObject)

children.get(tag="red")

children.get(type=MyObject, tag="red")
```

kinds()
Generates all types of the children (including super types)

remove(*child: Hashable*) → Hashable
Remove the given object from the container.

Parameters **child** – A hashable contained by container.

Example:

```
container.remove(myObject)
```

tags()
Generates all of the tags currently in the collections

walk()
Iterate over the children and their children.

class ppb.gomlib.**GameObject**(***props*)
A generic parent class for game objects. Handles:

- Property-based init (`Sprite(position=pos, image=img)`)
- Children management

add(*child: Hashable*, *tags: Iterable[T_co] = ()*) → None
Shorthand for *Children.add()*

children = None
The children of this object

get(***, *kind: Type[CT_co] = None*, *tag: Hashable = None*, ***kwargs*) → Iterator[T_co]
Shorthand for *Children.get()*

kinds
Shorthand for *Children.kinds()*

Deprecated since version 0.10: Use `.children.kinds()` instead.

remove(*child: Hashable*) → None
Shorthand for *Children.remove()*

tags
Shorthand for *Children.tags()*

Deprecated since version 0.10: Use `.children.tags()` instead.

ppb.gomlib.**walk**(*root*)
Conducts a walk of the GOM tree from the root.

Includes the root.

Is non-recursive.

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*, ***props*)

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

camera_class
alias of *ppb.camera.Camera*

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.

Sprites

Sprites are game objects.

To use a sprite you use `BaseScene.add` to add it to a scene. When contained in an active scene, the engine will call the various *event* handlers on the sprite.

When defining your own custom sprites, we suggest you start with *Sprite*. By subclassing *Sprite*, you get a number of features automatically. You then define your event handlers as methods on your new class to produce behaviors.

All sprites in ppb are built from composition via mixins or subclassing via traditional Python inheritance.

If you don't need the built in features of *Sprite* see *BaseSprite*.

Concrete Sprites

Concrete sprites are a combination of *BaseSprite* and various mixins. They implement a number of useful features for game development and should be the primary classes you subclass when building game objects.

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

Sprite defines no additional methods or attributes, but is made up of `BaseSprite` with the mixins *RotatableMixin*, *RenderableMixin*, and *SquareShapeMixin*.

For most use cases, this is probably the class you want to subclass to make your game objects.

If you need rectangular sprites instead of squares, see *RectangleSprite*.

`BaseSprite` does not accept any positional arguments, and uses keyword arguments to set arbitrary state to the `BaseSprite` instance. This allows rapid prototyping.

Example:

```
sprite = BaseSprite(speed=6)
print(sprite.speed)
```

This sample will print the numeral 6.

You may add any arbitrary data values in this fashion. Alternatively, it is considered best practice to subclass `BaseSprite` and set the default values of any required attributes as class attributes.

Example:

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

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

add(*child: Hashable*, *tags: Iterable[T_co] = ()*) → None
Shorthand for `Children.add()`

bottom
The y-axis position of the bottom of the object.

Can be set to a number.

bottom_left
The coordinates of the bottom left corner of the object.

Can be set to a `ppb_vector.Vector`.

bottom_middle
The coordinates of the midpoint of the bottom of the object.

Can be set to a `ppb_vector.Vector`.

center
The coordinates of the center point of the object. Equivalent to the `position`.

Can be set to a `ppb_vector.Vector`.

facing
The direction the "front" is facing.

Can be set to an arbitrary facing by providing a facing vector.

get(*, *kind: Type[CT_co] = None*, *tag: Hashable = None*, ***kwargs*) → Iterator[T_co]
Shorthand for `Children.get()`

height
The height of the sprite.

Setting the height of the sprite changes the `size` and *`width()`*.

kinds
Shorthand for `Children.kinds()`

Deprecated since version 0.10: Use `.children.kinds()` instead.

left
The x-axis position of the left side of the object.

Can be set to a number.

left_middle
The coordinates of the midpoint of the left side of the object.

Can be set to a `ppb_vector.Vector`.

remove(*child: Hashable*) → None
Shorthand for `Children.remove()`

right
The x-axis position of the right side of the object.

Can be set to a number.

right_middle
The coordinates of the midpoint of the right side of the object.

Can be set to a `ppb_vector.Vector`.

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

rotation
The amount the sprite is rotated, in degrees

tags
Shorthand for `Children.tags()`

Deprecated since version 0.10: Use `.children.tags()` instead.

top
The y-axis position of the top of the object.

Can be set to a number.

top_left
The coordinates of the top left corner of the object.

Can be set to a `ppb_vector.Vector`.

top_middle
The coordinates of the midpoint of the top of the object.

Can be set to a `ppb_vector.Vector`.

top_right
The coordinates of the top right corner of the object.

Can be set to a `ppb_vector.Vector`.

width
The width of the sprite.

Setting the width of the sprite changes `size` and *`height()`*.

class ppb.**RectangleSprite**(***props*)
A rectangle sprite.

Similarly to *`Sprite`*, *`RectangleSprite`* does not introduce any new methods or attributes. It's made up of `BaseSprite` with the mixins `RotatableMixin`, `RenderableMixin`, and `RectangleShapeMixin`.

`BaseSprite` does not accept any positional arguments, and uses keyword arguments to set arbitrary state to the `BaseSprite` instance. This allows rapid prototyping.

Example:

```
sprite = BaseSprite(speed=6)
print(sprite.speed)
```

This sample will print the numeral 6.

You may add any arbitrary data values in this fashion. Alternatively, it is considered best practice to subclass `BaseSprite` and set the default values of any required attributes as class attributes.

Example:

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

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

add(*child: Hashable*, *tags: Iterable[T_co] = ()*) → None
Shorthand for `Children.add()`

bottom
The y-axis position of the bottom of the object.

Can be set to a number.

bottom_left
The coordinates of the bottom left corner of the object.

Can be set to a `ppb_vector.Vector`.

bottom_middle
The coordinates of the midpoint of the bottom of the object.

Can be set to a `ppb_vector.Vector`.

center
The coordinates of the center point of the object. Equivalent to the `position`.

Can be set to a `ppb_vector.Vector`.

facing
The direction the "front" is facing.

Can be set to an arbitrary facing by providing a facing vector.

get(*, *kind: Type[CT_co] = None*, *tag: Hashable = None*, ***kwargs*) → Iterator[T_co]
Shorthand for `Children.get()`

kinds
Shorthand for `Children.kinds()`

Deprecated since version 0.10: Use `.children.kinds()` instead.

left
The x-axis position of the left side of the object.

Can be set to a number.

left_middle
The coordinates of the midpoint of the left side of the object.

Can be set to a `ppb_vector.Vector`.

remove(*child: Hashable*) → None
Shorthand for `Children.remove()`

right
The x-axis position of the right side of the object.

Can be set to a number.

right_middle
The coordinates of the midpoint of the right side of the object.

Can be set to a `ppb_vector.Vector`.

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

rotation
The amount the sprite is rotated, in degrees

tags
Shorthand for `Children.tags()`

Deprecated since version 0.10: Use `.children.tags()` instead.

top
The y-axis position of the top of the object.

Can be set to a number.

top_left
The coordinates of the top left corner of the object.

Can be set to a `ppb_vector.Vector`.

top_middle
The coordinates of the midpoint of the top of the object.

Can be set to a `ppb_vector.Vector`.

top_right
The coordinates of the top right corner of the object.

Can be set to a `ppb_vector.Vector`.

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.

The render expects a width and height (see `RectangleMixin`) and will skip rendering if a sprite has no shape. You can use `RectangleMixin`, `SquareMixin`, or set the values yourself.

Additionally, if *`image`* is `None`, the sprite will not be rendered. If you just want a basic shape to be rendered, see *`ppb.assets`*.

image = Ellipsis
(*`ppb.Image`*): The image asset

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

> **Warning:** rotation does not affect underlying shape (the corners are still in the same place), it only rotates the sprites image and provides a facing.

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

facing
The direction the "front" is facing.

Can be set to an arbitrary facing by providing a facing vector.

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

rotation
The amount the sprite is rotated, in degrees

class `ppb.sprites.`**RectangleShapeMixin**
A Mixin that provides a rectangular area to sprites.

Classes derived from RectangleShapeMixin default to the same size and shape as all ppb Sprites: A 1 game unit by 1 game unit square. Just set the width and height in your constructor (Or as *class attributes*) to change this default.

Note: The concrete class using *RectangleShapeMixin* must have a `position` attribute.

---

bottom
The y-axis position of the bottom of the object.

Can be set to a number.

bottom_left
The coordinates of the bottom left corner of the object.

Can be set to a `ppb_vector.Vector`.

bottom_middle
The coordinates of the midpoint of the bottom of the object.

Can be set to a `ppb_vector.Vector`.

center
The coordinates of the center point of the object. Equivalent to the *position*.

Can be set to a `ppb_vector.Vector`.

height = 1
The height of the sprite.

left
The x-axis position of the left side of the object.

Can be set to a number.

left_middle
The coordinates of the midpoint of the left side of the object.

Can be set to a `ppb_vector.Vector`.

right
The x-axis position of the right side of the object.

Can be set to a number.

right_middle
The coordinates of the midpoint of the right side of the object.

Can be set to a `ppb_vector.Vector`.

top
The y-axis position of the top of the object.

Can be set to a number.

top_left
The coordinates of the top left corner of the object.

Can be set to a `ppb_vector.Vector`.

top_middle
The coordinates of the midpoint of the top of the object.

Can be set to a `ppb_vector.Vector`.

top_right
The coordinates of the top right corner of the object.

Can be set to a `ppb_vector.Vector`.

width = 1
The width of the sprite.

class ppb.sprites.**SquareShapeMixin**
A Mixin that provides a square area to sprites.

Extends the interface of *RectangleShapeMixin* by using the *size* attribute to determine *width()* and *height()*. Setting either *width()* or *height()* sets the *size* and maintains the square shape at the new size.

The default size of *SquareShapeMixin* is 1 game unit.

Please see *RectangleShapeMixin* for additional details.

height
The height of the sprite.

Setting the height of the sprite changes the *size* and *width()*.

size = 1
The width and height of the object. Setting size changes the *height()* and *width()* of the sprite.

width
The width of the sprite.

Setting the width of the sprite changes *size* and *height()*.

Base Classes

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

class ppb.sprites.**BaseSprite**(***props*)
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.

BaseSprite does not accept any positional arguments, and uses keyword arguments to set arbitrary state to the *BaseSprite* instance. This allows rapid prototyping.

Example:

```
sprite = BaseSprite(speed=6)
print(sprite.speed)
```

This sample will print the numeral 6.

You may add any arbitrary data values in this fashion. Alternatively, it is considered best practice to subclass *BaseSprite* and set the default values of any required attributes as class attributes.

Example:

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

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

add(*child: Hashable*, *tags: Iterable[T_co] = ()*) → None
Shorthand for `Children.add()`

get(*, *kind: Type[CT_co] = None*, *tag: Hashable = None*, ***kwargs*) → Iterator[T_co]
Shorthand for `Children.get()`

kinds
Shorthand for `Children.kinds()`

Deprecated since version 0.10: Use `.children.kinds()` instead.

layer = 0
The layer a sprite exists on.

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

remove(*child: Hashable*) → None
Shorthand for `Children.remove()`

tags
Shorthand for `Children.tags()`

Deprecated since version 0.10: Use `.children.tags()` instead.

Text Rendering

ppb supports basic text rendering: single font, single style, no wrapping. Rendered fonts are graphical Assets that can be used any place you'd use *ppb.Image*

```
class Label(ppb.sprite):
    image = ppb.Text("Hello, World", font=ppb.Font("resources/noto.ttf", size=12))
```

TrueType and OpenType fonts (both `.ttf`) are supported, but must be shipped with your game. (System fonts are not supported.)

Note that fonts require a size in points. This controls the size the text is rendered at, but the size on screen is still controlled by `Sprite.size`.

class ppb.**Font**(*name*, *, *size*, *index=None*)
A TrueType/OpenType Font

Parameters

- **name** – the filename to load
- **size** – the size in points
- **index** – the index of the font in a multi-font file (rare)

class ppb.**Text**(*txt*, *, *font*, *color=(0, 0, 0)*)
A bit of rendered text.

Parameters

- **txt** – The text to display.

- **font** – The font to use (a *ppb.Font*)
- **color** – The color to use.

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: Union[Type[CT_co], ppb.scenes.BaseScene], *, 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** (*Union[Type, scenes.BaseScene]*) – 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.

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 None

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*, *, *targets=None*)
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.

Events can be targetted–they will only be delivered to specific objects instead of the whole tree. Note that this might cause objects to receive an event if they are no longer part of the object tree.

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 the systems.

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.assetlib.Asset*)
An object requested a sound be played.

Signal in an event handler to have a sound played.

Example:

```
signal(PlaySound(my_sound))
```

sound = None
A Sound asset.

class ppb.**Sound**(*name*)
The asset to use for sounds. A variety of file formats are supported.

Camera

Cameras are objects that straddle the line between game space and screen space. The renderer uses the position of the camera to translate *Sprite's* positions to the screen in order to make them visible.

The Renderer inserts a *Camera* into the current scene in response to the *SceneStarted*.

class ppb.camera.**Camera**(*renderer, target_game_unit_width: numbers.Real, viewport_dimensions: Tuple[int, int]*)

A simple Camera.

Intentionally tightly coupled to the renderer to allow information flow back and forth.

There is a one-to-one relationship between cameras and scenes.

You can subclass Camera to add event handlers. If you do so, set the camera_class class attribute of your scene to your subclass. The renderer will instantiate the correct type.

You shouldn't instantiate your own camera in general. If you want to override the Camera, see above.

Parameters

- **renderer** (*Renderer*) – The renderer associated with the camera.
- **target_game_unit_width** (*Real*) – The number of game units wide you would like to display. The actual width may be less than this depending on the ratio to the viewport (as it can only be as wide as there are pixels.)
- **viewport_dimensions** (*Tuple[int, int]*) – The pixel dimensions of the rendered viewport in (width, height) form.

height

The game unit height of the viewport.

See *ppb.sprites* for details about game units.

When setting this property, the resulting height may be slightly different from the value provided based on the ratio between the height of the window in screen pixels and desired number of game units to represent.

When you set the height, the width will change as well.

point_is_visible(*point: ppb_vector.Vector*) → bool

Determine if a given point is in view of the camera.

Parameters **point** (*Vector*) – A vector representation of a point in game units.

Returns Whether the point is in view or not.

Return type bool

sprite_in_view(*sprite: ppb.sprites.Sprite*) → bool

Determine if a given sprite is in view of the camera.

Does not guarantee that the sprite will be rendered, only that it exists in the visible space.

A sprite without area (size=0 or lacking width, height, or any of the sides accessors) behave as *point_is_visible()*.

Parameters **sprite** – The sprite to check

Type *Sprite*

Returns Whether the sprite is in the space in view of the camera.

Return type bool

translate_point_to_game_space(*point: ppb_vector.Vector*) → ppb_vector.Vector

Convert a vector from screen position to game position.

Parameters **point** (*Vector*) – A vector in pixels

Returns A vector in game units.

Return type Vector

translate_point_to_screen(*point: ppb_vector.Vector*) → ppb_vector.Vector
Convert a vector from game position to screen position.

Parameters **point** (*Vector*) – A vector in game units

Returns A vector in pixels.

Return type Vector

width
The game unit width of the viewport.

See *ppb.sprites* for details about game units.

When setting this property, the resulting width may be slightly different from the value provided based on the ratio between the width of the window in screen pixels and desired number of game units to represent.

When you set the width, the height will change as well.

Warning: Setting the game unit dimensions of a camera (whether via *Camera.width*, *Camera.height*, or the `target_game_unit_width` of the *Camera* constructor) will affect both *Camera.width* and *Camera.height*. Their ratio is determined by the defined window.

Directions

The ordinal directions.

A collection of normalized vectors to be referenced by name.

Best used for the positions or facings of *Sprites*.

ppb.directions.**Down = Vector(0.0, -1.0)**
Unit vector to the bottom of the screen from center.

ppb.directions.**DownAndLeft = Vector(-0.7071067811865475, -0.7071067811865475)**
Unit vector diagonally down and to the left of the screen from center.

ppb.directions.**DownAndRight = Vector(0.7071067811865475, -0.7071067811865475)**
Unit vector diagonally down and to the right of the screen from center.

ppb.directions.**Left = Vector(-1.0, 0.0)**
Unit vector to the left of the screen from center.

ppb.directions.**Right = Vector(1.0, 0.0)**
Unit vector to the right of the screen from center.

ppb.directions.**Up = Vector(0.0, 1.0)**
Unit vector to the top of the screen from center.

ppb.directions.**UpAndLeft = Vector(-0.7071067811865475, 0.7071067811865475)**
Unit vector diagonally up and to the left of the screen from center.

ppb.directions.**UpAndRight = Vector(0.7071067811865475, 0.7071067811865475)**
Unit vector diagonally up and to the right of the screen from center.

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.Sprite):
    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.Sprite):
    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**(***props*)
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 = None
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.

1.2.5 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 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 ocassions 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 refering 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.

Maintenance Schedules

This is a record of the intended maintenance schedules for various releases and dependency support.

Python Version Support

PursuedPyBear projects will support at least the latest version of pypy and the latest two versions of C Python

We tend to start testing new versions of cPython when they reach the release candidate stage, but don't guarantee compatibility until at least one ppb release after the latest Python release.

We drop older versions of Python when we begin using features from a newer version of Python. A future example of this would be dropping Python 3.7 when we start using assignment operators. This means the oldest version of Python ppb supports may be older than the two versions we promise to support.

Release Schedules

PursuedPyBear targets four releases a year, based on the solstices and equinoxes:

- Around the northward equinox, which is about March 20.
- Around the northern solstice, which is about June 20.
- Around the southward equinox, which is about September 20.
- Around the southern solstice, which is about December 21st.

We prefer to use the directional names of the solstices and equinoxes as we support a global community. Naming them after their seasons would leave out portions of the world.

About four weeks before the target release date, we freeze any new feature merges. This means any PR that is a feature or enhancement (not a bug fix, documentation change, or examples) may be approved, but held until after the release.

At the same time as freeze, we try to release the first beta of the new version. If there has been any changes, we like to release a new beta the next week, and one more the week after if bug fixes are still being submitted.

If the beta has been stable, at 2 weeks before the release, we like to cut a release candidate. At this point the majority of accepted PRs are documentation and bug fix related. We will cut as many release candidates as we see fit over the following two weeks.

On or around the expected release date, we will cut a final release.

Deprecations and API Breakages

PursuedPyBear is currently in a pre-release state. Many of its APIs have proven effective and long lasting. Other portions are still highly experimental as we find the correct developer experience.

Because of this, we have come to support deprecating APIs as best we can. Some APIs cannot be deprecated cleanly, and we will make note that in change logs.

In general, we seek to provide a deprecation warning under an existing name for at least two releases. For example, if we deprecate or significantly change an API in v0.10 it will also be available under a deprecation warning in v0.11 and is likely to be, but not guaranteed to be, available in v0.12.

After freeze, we will reexamine this deprecation policy in light of major version changes.

While it is not true now, it has been requested by a number of educators to limit major version changes to the Northern Solstice release. If you have an opinion on this, please join one of the discussion channels to add to this understanding.

PursuedPyBear's Default Branch

The default branch for all ppb repositories is `canon`.

The maintainers had been discussing switching the default branch from `master` for multiple years. With the growing public discussion of removing biased, racist, and discriminatory language from technical projects, we received a request from a community member to prioritize this discussion.

We opened it up to the community to find a new name. The alternatives put forward were `main`, which is the name settled on by github. We also had `trunk` suggested to tie to the idea of the "tree" that git commits make.

The idea for `canon` came from a few sources. From one maintainer it was suggested because PursuedPyBear is named after a Shakespeare reference, and the idea of a "canon" was a silly enough connection to fit.

It was the most popular of the suggestions and thus we settled on it.

Python Module Index

p

Index

Symbols

A

B

C

D

F

G

S

T

U

W