Pygame AI Documentation

Release 0.1

Nek

Jun 10, 2019

Contents

Pygame AI is a package that aims at implementing a bunch of common algorithms and techniques often used in Videogame AI. It is still a work in progress.

CHAPTER 1

Installation

The package is available at PyPI, you can simply do:

```
pip install pygame-ai
```

CHAPTER 2

Usage

After installing it you only need to import like:

```
import pygame_ai
```

or

```
import pygame_ai as pai
```

CHAPTER 3

Core

All of this library's implementations work using this implementation of `GameObject`.

You can see how the library works by downloading our *Example Game*, or you can visit the *PyGame AI Guide* to see how to implement a simple game using the library's core functions.

So far these are the major techniquest that have been implemented:

CHAPTER 4

Steering Behaviors

Basic movement behaviors that make in-game characters move in a pseudo-intelligent way. You will mainly be using *BlendedSteering* and *PrioritySteering*, but feel free to use any of the behaviors found in *kinematic* and *static* modules.

- *StaticSteeringBehavior*
- *KinematicSteeringBehavior*
- *BlendedSteering*
- *PrioritySteering*

CHAPTER 5

Table of Contents

5.1 Game Object

General-Purpose Game Object

This module implements the GameObject class, core of this AI engine, along with other useful classes, methods and constants

class gameobject.**GameObject**(*img_surf=<Surface(0x0x32 SW)>*, *pos=(0, 0)*, *max_speed=30*, *max_accel=20*, *max_rotation=60*, *max_angular_accel=50*)

General-Purpose Game Object.

Derives from pygame.sprite.Sprite.

Holds values relevant to any non-static entity in the game.

Parameters

- **img_surf** (pygame.Surface) – It is asigned to self.image, defaults to *null_surface*
- **pos** (*list_like(int, int), optional*) – Initial Position, it is assigned to self.rect.center
- **max_speed** (*int, optional*) – Maximum linear speed
- **max_accel** (*int, optional*) – Maximum linear acceleration
- **max_rotation** (*int, optional*) – Maximum angular speed
- **max_angular_accel** (*int, optional*) – Maximum angular acceleration

This class exposes the following public properties and methods

image

Surface to be blited to screen

Type pygame.Surface

rect

Derived from image, it's center is the GameObejct's position

Type pygame.Rect

position
Current position

Type pygame.math.Vector2

velocity
Current velocity

Type pygame.math.Vector2

max_speed
Maximum linear speed

Type int

max_accel
Maximum linear acceleration

Type int

orientation
Current orientation in degrees

Type int

rotation
Current angular velocity

Type int

max_rotation
Maximum angular speed

Type int

max_angular_accel
Maximum angular acceleration

Type int

get_lines()
Reruns what it returns, can you guess what it is?

steer(*steering*, *tick*)
Updates GameObject's velocity and rotation

Parameters

- **steering** (*kinematic.SteeringOutput* or :py:class:.'static.SteeringOutput') – The steering request to update velocity and rotation
- **tick** (*int*) – Time passed since the last loop

steer_angular(*steering*, *tick*)
Updates GameObject's rotation

Parameters

- **steering** (*kinematic.SteeringOutput* or :py:class:.'static.SteeringOutput') – The steering request to update velocity and rotation
- **tick** (*int*) – Time passed since the last loop

steer_x(*steering*, *tick*)
Updates GameObject's velocity along the x axis

Parameters

- **steering** (*kinematic.SteeringOutput* or :py:class:.‘static.SteeringOutput‘) – The steering request to update velocity and rotation
- **tick** (*int*) – Time passed since the last loop

steer_y(*steering*, *tick*)
Updates GameObject’s velocity along the y axis

Parameters

- **steering** (*kinematic.SteeringOutput* or :py:class:.‘static.SteeringOutput‘) – The steering request to update velocity and rotation
- **tick** (*int*) – Time passed since the last loop

class gameobject.**DummyGameObject**(*position=(0, 0)*)
A Dummy with *GameObject* properties

Derives from *GameObject*.

Used for quick instantiation when creating *GameObject* s that will only be used as palceholders and are not meant to appear on screen.

Parameters **position** (*list_like(int, int)*) – Current position

gameobject.**null_surface = <Surface(0x0x32 SW)>**
Empty Surface with size 0

Type (pygame.Surface)

5.2 Static

Static movement

This module implements a series of classes and methods that emulate the behavior of objects moving in a 2D space in a static way (not involving acceleration)

Notes

This might need a slightly better explaination

5.2.1 SteeringOutput

class steering.static.**SteeringOutput**(*velocity=None*, *rotation=None*)
Container for Steering data

This class is used as a container for the output of the *StaticSteeringBehavior* algorithms.

Parameters

- **velocity** (pygame.math.Vector2, optional) – Linear velocity, defaults to (0, 0)
- **rotation** (*int, optional*) – Angular velocity, defaults to 0

velocity
Linear velocity

Type pygame.math.Vector2

rotation
Angular velocity

Type int

update(*gameobject*, *tick*)
Update a *GameObject*'s velocity and rotation

This method should be called once per loop, it updates the given *gameobject.GameObject*'s velocity and rotation based on this *SteeringOutput*'s acceleration request

Parameters

- **gameobject** (*GameObject*) – The GameObject that will be updated
- **tick** (*int*) – Time transcurred since last loop

steering.static.**null_steering**
Constant with 0 linear velocity and 0 angular velocity

Type *SteeringOutput*

5.2.2 StaticSteeringBehavior

class steering.static.**StaticSteeringBehavior**
Template StaticSteeringBehavior class

This class is a template to supply base methods for StaticSteeringBehaviors. This class is meant to be subclassed since the methods here are just placeholders

draw_indicators(*screen*, *offset=<function StaticSteeringBehavior.<lambda>>*)
Draws appropiate indicators for each *StaticSteeringBehavior*

Parameters

- **screen** (pygame.Surface) – Surface in which to draw indicators, normally this would be the screen Surface
- **offset** (*function, optional*) – Function that applies an offset to the object's position

  This is meant to be used together with scrolling cameras, leave empty if your game doesn't implement one,it defaults to a linear function f(pos) -> pos

get_steering()
Returns a steering request

Returns Requested steering

Return type *SteeringOutput*

class steering.static.**Arrive**(*character*, *target*, *radius=None*, *time_to_arrive=0.25*)
StaticSteeringBehavior that makes the character **Arrive** at a target

Parameters

- **character** (*GameObject*) – Character with this behavior
- **target** (*GameObject*) – Target to **Arrive** at
- **radius** (*int, optional*) – Distance from the center of the target at which the character will stop
- **time_to_arrive** (*float, optional*) – Estimated time, in seconds, to **Arrive** at the target

class steering.static.**Flee**(*character*, *target*)
StaticSteeringBehavior that makes the character **Flee** from a target

Parameters

- **character** (*GameObject*) – Character with this behavior
- **target** (*GameObject*) – Target to **Flee** from

class steering.static.**Seek**(*character*, *target*)
StaticSteeringBehavior that makes the character **Seek** a target

Parameters

- **character** (*GameObject*) – Character with this behavior
- **target** (*GameObject*) – Target to **Seek**

class steering.static.**Wander**(*character*)
StaticSteeringBehavior that makes the character **Wander**

This behavior makes the character move with it's maximum speed in a particular direction for a random period of time, after that the character's orientation is changed randomly using the character's *max_rotation*.

Parameters **character** (*GameObject*) – Character with this behavior

5.3 Kinematic

Kinematic movement

This module implements a series of classes and methods that emulate the behavior of objects moving in a 2D space in a kinematic way (involving acceleration)

Notes

This might need a slightly better explaination

5.3.1 SteeringOutput

class steering.kinematic.**SteeringOutput**(*linear=None*, *angular=None*)
Container for Steering data

This class is used as a container for the output of the *KinematicSteeringBehavior* algorithms.

These objects can be added, multiplied, and compared to eachother. Each of these operations will be executed element-wise

Parameters

- **linear** (pygame.math.Vector2, optional) – Linear acceleration, defaults to (0, 0)
- **angular** (*int, optional*) – Angular acceleration, defaults to 0

linear
Linear acceleration

Type pygame.math.Vector2

angular
Angular acceleration

Type int

update(*gameobject*, *tick*)
Update a *GameObject*'s velocity and rotation

This method should be called once per loop, it updates the given *GameObject*'s velocity and rotation based on this *SteeringOutput*'s acceleration request

Parameters

- **gameobject** (*GameObject*) – The Game Objectthat will be updated
- **tick** (*int*) – Time transcurred since last loop

kinematic.**negative_steering**(*angular*)
Returns a steering request opposite to the linear and angular accelerations provided.

Parameters

- **linear** (pygame.math.Vector2) – Linear acceleration
- **angular** (*int*) – Angular acceleration

Returns

Return type *SteeringOutput*

steering.kinematic.**null_steering**
Constant with 0 linear acceleration and 0 angular acceleration

Type *SteeringOutput*

5.3.2 KinematicSteeringBehavior

class steering.kinematic.**KinematicSteeringBehavior**
Template KinematicSteeringBehavior class

This class is a template to supply base methods for KinematicSteeringBehaviors. This class is meant to be subclassed since the methods here are just placeholders

draw_indicators(*screen*, *offset=<function KinematicSteeringBehavior.<lambda>>*)
Draws appropiate indicators for each *KinematicSteeringBehavior*

Parameters

- **screen** (pygame.Surface) – Surface in which to draw indicators, normally this would be the screen Surface
- **offset** (*function, optional*) – Function that applies an offset to the object's position

  This is meant to be used together with scrolling cameras, leave empty if your game doesn't implement one,it defaults to a linear function f(pos) -> pos

get_steering()
Returns a steering request

Returns Requested steering

Return type *SteeringOutput*

class steering.kinematic.**Align**(*character*, *target*, *target_radius=1*, *slow_radius=20*, *time_to_target=0.1*)
KinematicSteeringBehavior that makes the character **Align** with the target's orientation

Parameters

- **character** (*GameObject*) – Character with this behavior
- **target** (*GameObject*) – Target to **Align** with it's orientation at
- **target_radius** (*int, optional*) – Distance, in degrees, from the target orientation at which the character will stop rotation
- **slow_radius** (*int, optional*) – Distance, in degrees, from the target orientation at which the character will start to slow rotation
- **time_to_target** (*float, optional*) – Estimated time, in seconds, to Align with the target's orientation

class steering.kinematic.**Arrive**(*character*, *target*, *target_radius=None*, *slow_radius=None*, *time_to_target=0.2*)

KinematicSteeringBehavior that makes the character **Arrive** at a target

Parameters

- **character** (*GameObject*) – Character with this behavior
- **target** (*GameObject*) – Target to **Arrive** at
- **target_radius** (*int, optional*) – Distance from the center of the target at which the character will stop
- **slow_radius** (*int, optional*) – Distance from the center of the target at which the character will start to slow down
- **time_to_target** (*float, optional*) – Estimated time, in seconds, to **Arrive** at the target

class steering.kinematic.**CollisionAvoidance**(*character*, *targets*, *radius=None*)

KinematicSteeringBehavior that makes the character **Avoid Collision** with a list of targets

This behavior looks at the velocities of the character and the targets to determine if they will collide in the next few loops, and if they will, it accelerates away from the collision point

This behavior is meant to be used in combination with other behaviors, see *steering.blended.BlendedSteering* .

Parameters

- **character** (*GameObject*) – Character with this behavior
- **targets** (list(*GameObject*)) – Targets to avoid collision with
- **radius** (*int, optional*) – Distance at which the future positions of the character and any target are are considered as *colliding*

class steering.kinematic.**Drag**(*linear_strength=10*, *angular_strength=1*)

KinematicSteeringBehavior that applies a **Drag** to the character

This behavior should be applied to every *GameObject* in every loop (unless it's meant to be permanently stationary). It applies an acceleration contrary to it's current linear and angular velocity.

Parameters **strenght** (*float, optional*) – The strength of the drag to apply, should be a number in the range (0, 1], any number outside of that range will have unexpected behavior.

class steering.kinematic.**Evade**(*character*, *target*, *max_prediction_time=0.2*)

KinematicSteeringBehavior that makes the character **Evade** the target

This behavior tries to predict the target's future position based on the direction it is currently moving, and then *Flee* s from that

Parameters

- **character** (*GameObject*) – Character with this behavior
- **target** (*GameObject*) – Target to **Evade**
- **max_prediction_time** (*float, optional*) – Maximum time, in seconds, to look ahead while predicting future position

class steering.kinematic.**Face**(*character*, *target*, *target_radius=1*, *slow_radius=10*, *time_to_target=0.1*)

KinematicSteeringBehavior that makes the character **Face** the target

This behavior creates a *DummyGameObject* that is looking in the direction of the target and then *Align* s with that dummy's orientation

Parameters

- **character** (*GameObject*) – Character with this behavior
- **target** (*GameObject*) – Target to **Face**
- **target_radius** (*int, optional*) – Distance, in degrees, from the target orientation at which the character will stop rotation
- **slow_radius** (*int, optional*) – Distance, in degrees, from the target orientation at which the character will start to slow rotation
- **time_to_target** (*float, optional*) – Estimated time, in seconds, to **Face** the target

class steering.kinematic.**Flee**(*character*, *target*)

KinematicSteeringBehavior that makes the character **Flee** from a target

Parameters

- **character** (*GameObject*) – Character with this behavior
- **target** (*GameObject*) – Target to **Flee** from

class steering.kinematic.**FollowPath**(*character*, *path*)

KinematicSteeringBehavior that makes the character **Follow a Path**

This behavior makes the character follow a particular *Path*. It will do so until the character has traversed all points in it.

Parameters

- **character** (*GameObject*) – Character with this behavior
- **path** (*steering.path.Path*) – Path that will be Followed

class steering.kinematic.**LookWhereYoureGoing**(*character*, *target_radius=1*, *slow_radius=20*, *time_to_target=0.1*)

KinematicSteeringBehavior that makes the character **Look Where He's Going**

This behavior makes the character face in the direction it's moving by creating a *DummyGameObject* that is looking in the direction of the character's velocity and then it *Align* s with that.

This behavior is meant to be used in combination with other behaviors, see *steering.blended.BlendedSteering* .

Parameters

- **character** (*GameObject*) – Character with this behavior

- **target_radius** (*int, optional*) – Distance, in degrees, from the target orientation at which the character will stop rotation
- **slow_radius** (*int, optional*) – Distance, in degrees, from the target orientation at which the character will start to slow rotation
- **time_to_target** (*float, optional*) – Estimated time, in seconds, to **LookWhereYoureGoing**

class steering.kinematic.**NullSteering**

KinematicSteeringBehavior that makes the character **Stay Still**

class steering.kinematic.**ObstacleAvoidance**(*character*, *obstacles*, *avoid_distance=None*, *lookahead=None*)

KinematicSteeringBehavior that makes the character **Avoid Obstacles**

This behavior looks ahead in the current direction the character is moving to see if it will collide with any obstacle, and if it does, creates a target *away* from the collision point and *Seek* s that.

The difference between this and *CollisionAvoidance* is that the **Obstacles** are considered to be a rectangular shape of a any size, while the targets are normally almost-square-sized.

This behavior is meant to be used in combination with other behaviors, see *steering.blended.BlendedSteering* .

Parameters

- **character** (*GameObject*) – Character with this behavior
- **obstacles** (list(*GameObject*)) – Obstacles to avoid collision with
- **avoid_distance** (*int, optional*) – Distance from the collision point at which the target that the algorithm uses to avoid collision will be generated
- **lookahead** (*int, optional*) – Distance to *look ahead* in the direction of the player's velocity

class steering.kinematic.**Pursue**(*character*, *target*, *max_prediction_time=0.2*)

KinematicSteeringBehavior that makes the character **Purse** the target

This behavior tries to predict the target's future position based on the direction it is currently moving, and then *Seek* s that

Parameters

- **character** (*GameObject*) – Character with this behavior
- **target** (*GameObject*) – Target to **Pursue**
- **max_prediction_time** (*float, optional*) – Maximum time, in seconds, to look ahead while predicting future position

class steering.kinematic.**Seek**(*character*, *target*)

KinematicSteeringBehavior that makes the character **Seek** a target

Parameters

- **character** (*GameObject*) – Character with this behavior
- **target** (*GameObject*) – Target to **Seek**

class steering.kinematic.**Separation**(*character*, *targets*, *treshold=None*)

KinematicSteeringBehavior that makes the character **Separate** itself from a list of targets

Parameters

- **character** (*GameObject*) – Character with this behavior

- **targets** (list(*GameObject*)) – Targets to stay separated from
- **treshold** (*int, optional*) – Distance from any of the targets at which the character will start separate from them

class steering.kinematic.**VelocityMatch**(*character*, *target*, *time_to_target=0.1*)

KinematicSteeringBehavior that makes the character match the velocity of the target

Parameters

- **character** (*GameObject*) – Character with this behavior
- **target** (*GameObject*) – Target to match it's velocity
- **time_to_target** (*float, optional*) – Estimated time, in seconds, to reach the target's velocity

class steering.kinematic.**Wander**(*character*, *wander_offset=50*, *wander_radius=15*, *wander_rate=20*)

KinematicSteeringBehavior that makes the character **Wander**

This behavior makes the character move with it's maximum speed in a random direction that feels smooth, meaning that it does not rotate too abruptly. This generates a target in front of the character and *Seek* s it while applying *:py:class:'LookWhereYoureGoing*, you can use the *KinematicSteeringBehavior.draw_indicators()* to see how the target is generated. This Behavior also uses *LookWhereYoureGoing*.

Parameters

- **character** (*GameObject*) – Character with this behavior
- **wander_offset** (*int, optional*) – Distance in front of the character to generate target to *Seek*
- **wander_radius** (*int, optional*) – Radius of the circumference in front of the character in which the target will generated
- **wander_rate** (*int, optional*) – Angles, in degrees, that the target is allowed to move along the circumference
- **align_target_radius** (*int, optional*) – Distance, in degrees, from the target orientation at which the character will stop rotation
- **slow_radius** (*int, optional*) – Distance, in degrees, from the target orientation at which the character will start to slow rotation
- **align_time** (*float, optional*) – Estimated time, in seconds, to **LookWhereYoureGoing**

5.4 Blended

Blended Steering Behaviors

This module implements a class that Blends a list of *KinematicSteeringBehavior* s and provides a weighted sum of their outputs as a steering request.

This is the bread and butter of **Steering Behaviors** since it easily combines different behaviors that allow for semi-complex AI behaviors.

Derives from *KinematicSteeringBehavior*.

Example

This is how you would normally create your own *BlendedSteering* , in this case we are making a more complex version of *Arrive* where the character looks where it's going and also tries to avoid any obstacle in the way.

```
flocking_behavior = BlendedSteering(
    character = character
    behaviors = [
        BehaviorAndWeight(kinematic.Arrive(character, target), weight = 1),
        BehaviorAndWeight(kinematic.LookWhereYoureGoing(character), weight = 1),
        BehaviorAndWeight(kinematic.ObstacleAvoidance(character, obstacles), weight =
↪2),
    ]
)
```

This module also includes a couple of pre-implemented *BlendedSteering*.

Todo: Make BehaviorAndWeight prettier, maybe use named tuples?

5.4.1 BehaviorAndWeight

class steering.blended.**BehaviorAndWeight**(*behavior*, *weight*)
Container for Behavior and Weight values

Parameters

- **behavior** (*KinematicSteeringBehavior*) –
- **weight** (*int*) –

5.4.2 BlendedSteering

class steering.blended.**BlendedSteering**(*character*, *behaviors*)
Base Blended Steering

This class provides methods neccesary to combine the list of steering behaviors and produce a single *SteeringOutput*.

Derives from *KinematicSteeringBehavior*, currently *BlendedSteering* with *StaticSteeringBehavior* is not supported.

Parameters

- **character** (*GameObject*) – Character with this behavior
- **behaviors** (list(*BehaviorAndWeight*)) – List of behaviors that compose this *BlendedSteering*

draw_indicators(*screen*, *offset=<function BlendedSteering.<lambda>>*)
Draws appropiate indicators for this *BlendedSteering*

Draws the indicators of all *KinematicSteeringBehavior* that compose this *BlendedSteering*.

Parameters

- **screen** (pygame.Surface) – Surface in which to draw indicators, normally this would be the screen Surface

- **offset** (*function, optional*) – Function that applies an offset to the object's position

  This is meant to be used together with scrolling cameras, leave empty if your game doesn't implement one,it defaults to a linear function f(pos) -> pos

get_steering()
Returns the combined steering request of this *BlendedSteering*

Returns Requested steering

Return type SteeringOutput

class steering.blended.**Arrive**(*character*, *target*, *obstacles*, *target_radius=None*, *slow_radius=None*)
BlendedSteering that makes the character **Arrive** at a target

This is behavior is a more complex version of *Arrive* that also **Looks Where it's Going** and tries to **Avoid Obstacles**.

Parameters

- **character** (*GameObject*) –
- **target** (*GameObject*) –
- **obstacles** (iterable(pygame.sprite.Sprite)) – Solid obstacles

class steering.blended.**Flocking**(*character*, *swarm*, *target*)
BlendedSteering that makes the character move in a flock-like way

This behavior is meant to be used with several characters, they will all try to **Arive** at the same target location while **Looking Where They're Going** and keeping **Separated** from eachother.

Parameters

- **character** (*GameObject*) –
- **swarm** (iterable(pygame.sprite.Sprite)) – Rest of the entities that conform the Flock
- **target** (*GameObject*) –

class steering.blended.**Wander**(*character*, *obstacles*)
BlendedSteering that makes the character **Wander** around.

This behaviors is a more complex version of *Wander* that also tries to **Avoid Obstacles**.

Parameters

- **character** (*GameObject*) –
- **obstacles** (iterable(pygame.sprite.Sprite)) – Solid obstacles

5.5 Priority

Priority Steering Behaviors

This module implements a class that holds a list of *KinematicSteeringBehavior*s and applies them in order, keeping only the first one that produces an output greater than a certain treshold. This means that some behaviors which are considered more important (like *ObstacleAvoidance* and *CollisionAvoidance*) but are not always neccesary to reach the character's goal can be ignored when they don't produce a meaningful output, it also means that when they do produce a meaningful output they will be the only ones in action.

This is a very simple form of decision making that involves only steering algorithms, and therefore it is classified as a steering behavior.

Derives from *KinematicSteeringBehavior*.

Example

This is how you would normally create your own *PrioritySteering*, in this case we are making a behavior that will most of the time **Pursue** a target, but will prioritize **Avoiding Obstacles** when that behavior returns a steering greater than the treshold.

```
mybehavior = PrioritySteering(
    behaviors = [
        kinematic.ObstacleAvoidance(character, obstacles),
        kinematic.Pursue(character, target),
    ],
)
```

5.5.1 PrioritySteering

class steering.priority.**PrioritySteering**(*behaviors*, *epsilon=0.1*)

draw_indicators(*screen*, *offset=<function PrioritySteering.<lambda>>*)
Draws appropiate indicators for each KinematicSteeringBehavior

Parameters

- **screen** (pygame.Surface) – Surface in which to draw indicators, normally this would be the screen Surface
- **offset** (*function, optional*) – Function that applies an offset to the object's position

  This is meant to be used together with scrolling cameras, leave empty if your game doesn't implement one,it defaults to a linear function f(pos) -> pos

get_steering()
Returns a steering request

Returns Requested steering

Return type SteeringOutput

5.6 Path

Iterator that describes a Path

This module implements an iterator *Path* to be used the descriptions of the points that form a particular path. There are also the following classes with specialised paths:

- *CyclicPath*
- *MirroredPath*

Aswell as the following pre-implemented useful paths:

- *PathCircumference*

- *PathParabola*

5.6.1 Path

class steering.path.**Path**(*path_func*, *domain_end*, *domain_start=0*, *increment=1*)

Iterator that describes a **Path**

Provides a flexible interface to describe dynamic paths as an iterator, it uses a **Path Function** in the form of f(x) = y to describe the path.

Parameters

- **path_func** (*function number -> number*) – The function that describes the path
- **domain_start** (*int*) – Start point for the function domain, defaults to 0
- **domain_end** (*int*) – End point for the function domain
- **increment** (*int*) – Step to generate every point on the path

Example

The way this class is meant to be used is by sub-classing it and creating your own class with your own properties, this is a slightly useluess implementation of `CircumferencePath` that only traverses the path once.

```
class OnceCircumferencePath(Path):

    def __init__(self, center, radius):
        self.center = center
        self.radius = radius

        def circumference_path(self, x):
            angle = math.radians(x)
            center = self.center
            x = center[0] + math.cos(angle)*self.radius
            y = center[1] + math.sin(angle)*self.radius
            return x, y

        super(OnceCircumferencePath, self).__init__(parabola_path, domain_start =␣
↪0, domain_end = 360, increment = 15)

>>> mypath = OnceCircumferencePath(center = (50, 50), radius = 50)
>>> next(mypath)
(100.0, 50.0)
>>> next(mypath)
(98.29629131445341, 62.940952255126035)
>>> next(mypath)
(93.30127018922194, 75.0)
```

A good tip is to use lambda functions in order to have dynamically updated paths, this allows to have attributes like 'center' update with the position of something in the game, which will alter the points the path will produce

```
class OnceCircumferencePath(Path):

    def __init__(self, center, radius):
        self.center = center
        self.radius = radius
```

(continues on next page)

(continued from previous page)

```
        def circumference_path(self, x):
            angle = math.radians(x)
            # Notice that we are now calling the attribute 'center' as a function
            center = self.center()
            x = center[0] + math.cos(angle)*self.radius
            y = center[1] + math.sin(angle)*self.radius
            return x, y

        super(OnceCircumferencePath, self).__init__(parabola_path, domain_start =␣
↪0, domain_end = 360, increment = 15)

>>> character = SomeGameObjectWithARect()
# The 'center' parameter is now defined as a lambda functions that gets the␣
↪position of a character
>>> mypath = OnceCircumferencePath(center = (lambda: character.rect.center),␣
↪radius = 50)
```

as_list()
Returns the path as a list of points

This ignores the infinity of *CyclicPath* and *MirroredPath* and returns a finite list. Nevertheless, you should keep in mind that if for your own sub-classes this methods does not return the expected results, it's probabbly the method's fault (my faul) and you should implement your own since this is used for drawing indicators.

Returns

Return type list(tuple(float, float))

reset()
Returns the iterator to it's initial point

5.6.2 Special Paths

class steering.path.**CyclicPath**(*path_func*, *domain_end*, *domain_start=0*, *increment=1*)
Iterator that implements Cyclic Paths

This is a sub-class of *Path* that returns to the path's starting point once it reaches the end, this produces an infinite iterator.

Uses the same parameters as *Path*.

class steering.path.**MirroredPath**(*path_func*, *domain_end*, *domain_start=0*, *increment=1*)
Iterator that implements Mirrored Paths

This is a sub-class of *Path* that **Mirrors** the path produced by the given function, this produces an infinite iterator that backtracks on the traversed path once it reaches it's domain_end, and does the same after it reaches domain_start.

Uses the same parameters as *Path*.

as_list()
Returns the path as a list of points

This ignores the infinity of *CyclicPath* and *MirroredPath* and returns a finite list. Nevertheless, you should keep in mind that if for your own sub-classes this methods does not return the expected results, it's probabbly the method's fault (my faul) and you should implement your own since this is used for drawing indicators.

Returns

Return type list(tuple(float, float))

5.6.3 Pre-implemented Paths

class steering.path.**PathCircumference**(*center*, *radius*, *start=0*)
Circumference-like *CyclicPath*

Parameters

- **center** (*tuple(int, int) or function -> tuple(int, int)*) –
- **radius** (*int*) –

class steering.path.**PathParabola**(*origin*, *width=400*, *height=100*)
Parabola-like *MirroredPath*

Parameters

- **origin** (*tuple(int, int) or function -> tuple(int, int)*) – Lowest point of the parabola
- **width** (*int*) –
- **height** (*int*) –

5.7 Example Game

This game shows examples of what things can be done with the library, to run it just unzip and run main.py while having **pygame** and **pygame_ai** installed.

- Download Example Game

5.8 PyGame AI Guide

This guide is meant to explain the basic concepts you need to know to integrate this library into your game; it is not meant to explain how to create games with PyGame, although it does include a very basic game structure that you can use as a template.

5.8.1 Contents

- *More Complex Stuff*

5.8.2 Game Structure

This is the basic game structure that I'll be working with in this guide, it contains the basic things that any PyGame game should have.

```
import sys

import pygame
from pygame.locals import *
import pygame_ai as pai

RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)

# Custom classes and definitions
# ...

def main():

    # Create screen
    screen_width, screen_height = 800, 600
    screen = pygame.display.set_mode((screen_width, screen_height))
    pygame.display.set_caption('PyGame AI Guide')

    # Create white background
    background = pygame.Surface((screen_width, screen_height)).convert()
    background.fill((255, 255, 255))

    # Initialize clock
    clock = pygame.time.Clock()

    # Variables that you will use in your game loop
    # ...

    # Game loop
    while True:

        # Get loop time, convert milliseconds to seconds
        tick = clock.tick(60)/1000

        # Handle input
        for event in pygame.event.get():
            if event.type == QUIT:
                sys.exit(2)

        # Erase previous frame by bliting background
        screen.blit(background, background.get_rect())

        # Update the entities in your game
        # ...

        # Blit all your entities
        # . . .
```

(continues on next page)

(continued from previous page)

```
        # Update display
        pygame.display.update()

if __name__ == '__main__':
    pygame.init()
    main()
    pygame.quit()
```

With this in place we can move on.

5.8.3 Game Objects

I used the word *entities* before, by that I mean all the moving things in your game, like the player, enemies, NPCs, you name it. The way these entities will be better represented (and the only way they should be, unless you really know what you're doing) in a game that uses this library is by using the *GameObject* class, it contains all the necessary properties and methods the library uses to do all its calculations.

The way you create your own entities is by subclassing *GameObject* like this:

```
class Player(pai.gameobject.GameObject):

    def __init__(self, pos = (0, 0)):
        # First we create the image by filling a surface with blue color
        img = pygame.Surface( (10, 15) ).convert()
        img.fill(BLUE)
        # Call GameObject init with appropiate values
        super(Player, self).__init__(
            img_surf = img,
            pos = pos,
            max_speed = 15,
            max_accel = 40,
            max_rotation = 40,
            max_angular_accel = 30
        )
```

Note that the first thing I do is create a pygame.Surface; this is the image that will be displayed in the game, in this case it is a blue rectangle. Also note that we hand picked most of the parameters for the *GameObject*; this is unfortunately still done through trial and error, the values that I used work sort of smoothly but they can be improved.

Now, that is not enough to call it done; we still need to implement a way for this entity to move, this is usually done through an **update** function, in fact, every *GameObject* has one, but it doesn't do anything.

This is an example Player with its update function:

```
class Player(pai.gameobject.GameObject):

    def __init__(self, pos = (0, 0)):
        # First we create the image by filling a surface with blue color
        img = pygame.Surface( (10, 15) ).convert()
        img.fill(BLUE)
        # Call GameObject init with appropiate values
        super(Player, self).__init__(
            img_surf = img,
            pos = pos,
            max_speed = 15,
            max_accel = 40,
```

(continues on next page)

(continued from previous page)

```
            max_rotation = 40,
            max_angular_accel = 30
        )

    def update(self, steering, tick):
        self.steer(steering, tick)
        self.rect.move_ip(self.velocity)
```

Essentially what it does is to **accelerate** the entity in the direction and strength dictated by the **steering** parameter and then move the entity's rect with the direction and strength of the entity's **velocity**.

5.8.4 Steering

Steering Algorithms are the core of movement in this library; the *SteeringOutput* is the way these algorithms communicate how an object should **accelerate** in order to achieve its goal.

In the previous example we saw that the player does not produce its own steering, that is because normally the player is controlled by user input; we'll see later how we can create and modify our own `SteeringOutput` to move the player, for now let's move on and actually implement something useful with this library.

5.8.5 Steering as an AI

You can have NPCs whose behavior is only composed by a *KinematicSteeringBehavior*, this would be a very simple but often useful AI design, this is how you can implement an NPC whose only AI behavior is a *KinematicSteeringBehavior*:

```
class CircleNPC(pai.gameobject.GameObject):

    def __init__(self, pos = (0, 0)):
        # First create the circle image with alpha channel to have transparency
        img = pygame.Surface( (10, 10) ).convert_alpha()
        img.fill( (255, 255, 255, 0) )
        # Draw the circle
        pygame.draw.circle(img, RED, (5, 5), 5)
        # Call GameObject init with appropiate values
        super(CircleNPC, self).__init__(
            img_surf = img,
            pos = pos,
            max_speed = 25,
            max_accel = 40,
            max_rotation = 40,
            max_angular_accel = 30
        )
        # Create a placeholder for the AI
        self.ai = pai.steering.kinematic.NullSteering()

    def update(self, tick):
        steering = self.ai.get_steering()
        self.steer(steering, tick)
        self.rect.move_ip(self.velocity)
```

The main differences between this and the **Player** entity are that:

1) The image is a circle

2) The update actually generates its own steering

The way that point (2) is achieved is by calling the *KinematicSteeringBehavior*'s *get_steering()* method; this returns the behaviors' *SteeringOutput*; it is then applied to the *GameObject* with the *steer()* method. This is not the only steering method that exists, we will see more about these methods later.

5.8.6 Very Simple Game

With all we have learned so far, we can make a very simple game that consists only of one input-controlled player and one NPC that chases the player.

First we need to see how to make the player input-controlled, for that we need to create an artificial *SteeringOutput* that we can modify, let's add that:

```
# . . .

# Variables that you will use in your game loop
# Create player steering
player_steering = pai.steering.kinematic.SteeringOutput()

# Game loop
while True:

    # Get loop time, convert milliseconds to seconds
    tick = clock.tick(60)/1000

    # Restart player steering
    player_steering.reset()

    # Handle input
    for event in pygame.event.get():
        if event.type == QUIT:
            sys.exit(2)

    # . . .
```

With that we simply created an empty *SteeringOutput* that gets `reset()` every frame, this is to guarantee that it will not grow infinitely.

Then we need to catch user input and modify the steering accordingly, this piece of code is horrible but I'll use it to avoid complicating the guide with things that do not relate to the library.

```
# . . .

# Handle input
for event in pygame.event.get():
    if event.type == QUIT:
        sys.exit(2)

keys = pygame.key.get_pressed()
if keys[K_w]:
    player_steering.linear[1] -= player.max_accel
if keys[K_a]:
    player_steering.linear[0] -= player.max_accel
if keys[K_s]:
    player_steering.linear[1] += player.max_accel
if keys[K_d]:
```

(continues on next page)

(continued from previous page)

```
    player_steering.linear[0] += player.max_accel

# . . .
```

Now it is time to actually apply this to the player using the **update** method that we wrote, for that we first need to instantiate the **Player** class that we created. We will also need to instantiate the **CircleNPC** class and do the same with it:

```
# . . .

# Variables that you will use in your game loop
# Create player steering
player_steering = pai.steering.kinematic.SteeringOutput()

# Instantiate game objects
player = Player(pos = (screen_width//2, screen_height//2))
circle = CircleNPC(pos = (screen_width//4, screen_height//2))

# Set the NPC AI
circle.ai = pai.steering.kinematic.Arrive(circle, player)

# Game loop
while True:

    # . . .

    # Erase previous frame by bliting background
    screen.blit(background, background.get_rect())

    # Update player and NPCs
    player.update(player_steering, tick)
    circle.update(tick)

    # Blit all your entities
    screen.blit(player.image, player.rect)
    screen.blit(circle.image, circle.rect)

    pygame.display.update()
```

Apart from instantiating the **CircleNPC** we changed its **ai** property to be *Arrive*, this will make the NPC arrive near the player and then stop accelerating, We also added the code necessary to blit our entity's images to the screen.

If you were to run this code now, it should run properly, you should see the player in the center of the screen and the NPC chasing the player, the problem is, if you try to move the player, well... it won't stop moving. That is because when we *steer()* the player its velocity increases, and since we are moving its position based on its velocity, it will never stop moving (unless you steer it correctly to negate its current velocity). The NPC will also never stop beside the player as it should.

5.8.7 Drag

To avoid this behavior we need to apply some sort of **Drag** to our entities, luckily, I've implemented a *KinematicSteeringBehavior* that does just that, enter *Drag*.

The only particular thing about this behavior is that you will not normally create an individual instance for every entity, instead you should create one for every *surface* or *environment* your entity is in. This is because an entity will have less drag trying to run in plain land than trying to run with its body half-submerged in water.

For this example we are using only one instance of *Drag* and applying it to all entities, but you can get creative.

```
# . . .

# Create drag
drag = pai.steering.kinematic.Drag(15)

# Game loop
while True:

    # . . .

    # Erase previous frame by bliting background
    screen.blit(background, background.get_rect())

    # Update player and NPCs
    player.update(player_steering, tick)
    circle.update(tick)

    # Apply drag
    player.steer(drag.get_steering(player), tick)
    circle.steer(drag.get_steering(circle), tick)

    # Blit all your entities
    screen.blit(player.image, player.rect)
    screen.blit(circle.image, circle.rect)

    pygame.display.update()
```

Now you will be able to run the code and it should behave as expected. Note that you can totally add the drag instructions in your entity's **update** function to make the code less cluttered; I just added it there to avoid having to pass it as an argument or putting it inside the class.

That was a very basic game, and you should be able to use most of the library's movement behaviors only with that, the only thing that is a little different is the *Path* class used by the *FollowPath* behavior.

5.8.8 Paths

You can create very light-weight paths using the *Path* class, the only "problem" is that the paths are defined as mathematic functions, for people unfamiliar with that it can be quite spooky, and I would recommend them to use the pre-implemented paths. Otherwise it is very easy to define paths with this class; let's define a very simple cosine-wave-shaped path and make an NPC follow it:

```
import math

# . . .

class PathCosine(pai.steering.path.Path):

    def __init__(self, start, height, length):
        self.start = start
        self.height = height
        self.length = length

        def cosine_path(self, x):
            y = self.start[1] + math.cos(x) * self.height
```

(continues on next page)

(continued from previous page)

```
            return x, y

        super(PathCosine, self).__init__(
            path_func = cosine_path,
            domain_start = int(self.start[0]),
            domain_end = int(self.start[0] + length),
            increment = 30
        )
```

And that is it, the *Path* class handles everything, you just need to specify the function, its domain and a discrete increment (for each point to be generated).

Now we need to put it into the game, let's create another NPC instance and assign *FollowPath* with our **PathCosine** to its AI behavior.

```
# . . .

# Instantiate game objects
player = Player(pos = (screen_width//2, screen_height//2))
circle = CircleNPC(pos = (screen_width//4, screen_height//2))
circle2 = CircleNPC(pos = (screen_width//5, screen_height//2))

# Set the NPC AI
circle.ai = pai.steering.kinematic.Arrive(circle, player)
path_cosine = PathCosine(
    start = circle2.position,
    height = 200,
    length = 500
)
circle2.ai = pai.steering.kinematic.FollowPath(circle2, path_cosine)

# . . .
```

Remember to also update, blit and apply drag to circle2.

```
# . . .

# Update player and NPCs
player.update(player_steering, tick)
circle.update(tick)
circle2.update(tick)

# Apply drag
player.steer(drag.get_steering(player), tick)
circle.steer(drag.get_steering(circle), tick)
circle2.steer(drag.get_steering(circle2), tick)

# Blit all your entities
screen.blit(player.image, player.rect)
screen.blit(circle.image, circle.rect)
screen.blit(circle2.image, circle2.rect)

# . . .
```

Now you should see an NPC that follows a cosine-wave like path, it will only go trough it once, you can use *reset()* to make the path reset at any point, or you can take a look into *CyclicPath* and *MirroredPath* for special Path implementations.

5.8.9 Other Behaviors

Finally, this library also implements a couple of different kinds of *KinematicSteeringBehavior*s which are *BlendedSteering* and *PrioritySteering*. These allow you to combine different basic behaviors to create more complicated ones. Take a look at the pre-implemented behaviors to see what is possible by trying them out.

5.8.10 Gravity

Many games include gravity as a core feature (so core that most people won't consider it a feature). There are a couple of things we need to consider when adding gravity into our game, but here I'll show a very basic NPC that has gravity applied.

First, if we are going to have falling entities, we need to make sure they don't fall off-screen. For that we can add a very simple check to make sure nothing moves under the screen:

```
# . . .

# Entities affected by gravity
gravity_entities = []

# . . .

# Game loop
while True:

    # . . .

    # Update player and NPCs
    player.update(player_steering, tick)
    circle.update(tick)
    circle2.update(tick)

    # Check if our gravity-affected entities are falling off-screen
    for gentity in gravity_entities:
        if gentity.rect.bottom > screen_height:
            gentity.rect.bottom = screen_height

    # . . .
```

Now we need to actually implement an entity that is affected by gravity, let's make that an NPC:

```
class GravityCircleNPC(pai.gameobject.GameObject):

    def __init__(self, pos = (0, 0)):
        # First create the circle image with alpha channel to have transparency
        img = pygame.Surface( (10, 10) ).convert_alpha()
        img.fill( (255, 255, 255, 0) )
        # Draw the circle
        pygame.draw.circle(img, RED, (5, 5), 5)
        # Call GameObject init with appropiate values
        super(GravityCircleNPC, self).__init__(
            img_surf = img,
            pos = pos,
            max_speed = 10,
            max_accel = 40,
            max_rotation = 40,
```

(continues on next page)

(continued from previous page)

```
            max_angular_accel = 30
        )
        # Create a placeholder for the AI
        self.ai = pai.steering.kinematic.NullSteering()

    def update(self, tick):
        # Gravity steering
        gravity = pai.steering.kinematic.SteeringOutput()
        gravity.linear[1] = 300 # This value is arbitrary, it just works

        # Steer only along x axis
        steering = self.ai.get_steering()
        self.steer_x(steering, tick)

        # Get total velocity considering gravity
        velocity = self.velocity + gravity.linear * tick

        # Move with that velocity
        self.rect.move_ip(velocity)
```

The only difference between this and the regular **CircleNPC** is in the **update** function; in this one we create a *SteeringOutput* to act as the **gravity**; we then only consider the AI steering along the x axis, finally we get a total **velocity** composed of the NPC's velocity plus the velocity induced by gravity. This way we separate the velocity produced by the actual NPC from the one produced by any external force (in this case gravity).

Now we only need to instantiate this NPC and do all necessary actions to have it function like the rest of the entities (add it to the gravity entities list, assign it an AI behavior, update, blit and apply drag).

```
# . . .

# Instantiate game objects
player = Player(pos = (screen_width//2, screen_height//2))
circle = CircleNPC(pos = (screen_width//4, screen_height//2))
circle2 = CircleNPC(pos = (screen_width//5, screen_height//2))
circle3 = GravityCircleNPC(pos = (screen_width//6, screen_height//2))

# Remember to add it to our gravity_entitites list for collision
gravity_entities.append(circle3)

# Set the NPC AI
circle.ai = pai.steering.kinematic.Arrive(circle, player)
path_cosine = PathCosine(
    start = circle2.position,
    height = 200,
    length = 500
)
circle2.ai = pai.steering.kinematic.FollowPath(circle2, path_cosine)
circle3.ai = pai.steering.kinematic.Seek(circle3, player)

# . . .

# Game loop
while True:

    # . . .

    # Update player and NPCs
```

(continues on next page)

(continued from previous page)

```
    player.update(player_steering, tick)
    circle.update(tick)
    circle2.update(tick)
    circle3.update(tick)

    # Check if our gravity-affected entities are falling off-screen
    for gentity in gravity_entities:
        if gentity.rect.bottom > screen_height:
            gentity.rect.bottom = screen_height

    # Apply drag
    player.steer(drag.get_steering(player), tick)
    circle.steer(drag.get_steering(circle), tick)
    circle2.steer(drag.get_steering(circle2), tick)
    circle3.steer(drag.get_steering(circle3), tick)

    # Blit all your entities
    screen.blit(player.image, player.rect)
    screen.blit(circle.image, circle.rect)
    screen.blit(circle2.image, circle2.rect)
    screen.blit(circle3.image, circle3.rect)

    # . . .
```

You can now run the code again and you should see a falling NPC that also seeks the player while staying stuck to the ground.

5.8.11 More Complex Stuff

This was a very simple game to show the basic concepts that this library uses. You can download the game here. You only need to run main.py while having pygame and pygame_ai installed.

If you are interested in knowing what else you can do with this library you should check out the *Example Game*. You can take a look at how I implement things there, but you will find a lot of gibberish that is not directly related to the library.

CHAPTER 6

Indices and tables

- genindex
- search

Python Module Index

g

s

Index