
easymunk Documentation

Release 0.9.1

Fábio Mendes

Apr 04, 2021

CONTENTS

1	Installation	3
2	Example	5
3	Documentation	7
4	The Easymunk Vision	9
5	Contact & Support	11
6	Dependencies / Requirements	13
7	Install from source / Chipmunk Compilation	15
8	Contents	17
8.1	Installation	17
8.1.1	Install Easymunk	17
8.1.2	Examples & Documentation	17
8.1.3	Troubleshooting	18
8.1.4	Advanced - Android Install	18
8.1.4.1	Kivy	18
8.1.4.2	Termux	18
8.1.5	Advanced - Install	18
8.1.5.1	Advanced - Running without installation	19
8.1.6	Compile Chipmunk	19
8.1.7	CFFI Installation	20
8.2	Overview	20
8.2.1	Basics	20
8.2.2	Model your physics objects	20
8.2.2.1	Object shape	20
8.2.2.2	Mass, weight and units	21
8.2.2.3	Looks before realism	21
8.2.3	Game loop / moving time forward	21
8.2.4	Object tunneling	22
8.2.5	Unstable simulation?	22
8.2.6	Performance	23
8.2.7	Copy and Load/Save Easymunk objects	23
8.2.8	Additional info	23
8.3	API Reference	23
8.3.1	easymunk Package	23
8.3.1.1	easymunk.geometry Module	24

8.3.1.2	<code>easymunk.core</code> Module	24
8.3.1.3	<code>easymunk.linalg</code> Module	24
8.3.1.4	<code>easymunk.matplotlib</code> Module	24
8.3.1.5	<code>easymunk.pygame</code> Module	26
8.3.1.6	<code>easymunk.pyglet</code> Module	28
8.3.1.7	<code>easymunk.pyxel</code> Module	30
8.4	Examples	31
8.4.1	Jupyter Notebooks	31
8.4.1.1	<code>matplotlib_util_demo.ipynb</code>	31
8.4.1.2	<code>newtons_cradle.ipynb</code>	32
8.4.2	Standalone Python	32
8.4.2.1	<code>arrows.py</code>	34
8.4.2.2	<code>balls_and_lines.py</code>	35
8.4.2.3	<code>basic_test.py</code>	36
8.4.2.4	<code>bouncing_balls.py</code>	36
8.4.2.5	<code>box2d_pyramid.py</code>	37
8.4.2.6	<code>box2d_vertical_stack.py</code>	38
8.4.2.7	<code>breakout.py</code>	38
8.4.2.8	<code>constraints.py</code>	39
8.4.2.9	<code>contact_and_no_flipy.py</code>	40
8.4.2.10	<code>contact_with_friction.py</code>	41
8.4.2.11	<code>copy_and_pickle.py</code>	42
8.4.2.12	<code>damped_rotary_spring_pointer.py</code>	43
8.4.2.13	<code>deformable.py</code>	44
8.4.2.14	<code>flipper.py</code>	45
8.4.2.15	<code>index_video.py</code>	46
8.4.2.16	<code>kivy_pymunk_demo</code>	47
8.4.2.17	<code>logo.py</code>	48
8.4.2.18	<code>newtons_cradle.py</code>	48
8.4.2.19	<code>platformer.py</code>	49
8.4.2.20	<code>playground.py</code>	49
8.4.2.21	<code>point_query.py</code>	50
8.4.2.22	<code>py2exe_setup__basic_test.py</code>	51
8.4.2.23	<code>py2exe_setup__breakout.py</code>	52
8.4.2.24	<code>pygame_demo.py</code>	52
8.4.2.25	<code>pyglet_demo.py</code>	52
8.4.2.26	<code>shapes_for_draw_demos.py</code>	52
8.4.2.27	<code>slide_and_pinjoint.py</code>	52
8.4.2.28	<code>spiderweb.py</code>	53
8.4.2.29	<code>tangram.py</code>	54
8.4.2.30	<code>tank.py</code>	54
8.4.2.31	<code>using_sprites.py</code>	55
8.4.2.32	<code>using_sprites_pyglet.py</code>	56
8.5	Showcase	57
8.5.1	Games	59
8.5.2	Non-Games	61
8.5.3	Papers / Science	62
8.5.3.1	Cite Pymunk	64
8.6	Tutorials	64
8.6.1	Slide and Pin Joint Demo Step by Step	64
8.6.1.1	Before we start	65
8.6.1.2	An empty simulation	66
8.6.1.3	Falling balls	67
8.6.1.4	A static L	69

8.6.1.5	Joints (1)	71
8.6.1.6	Joints (2)	71
8.6.1.7	Ending	72
8.6.2	External Tutorials	74
8.7	Benchmarks	74
8.7.1	Micro benchmarks	75
8.7.1.1	Results:	75
8.7.2	Compared to Other Physics Libraries	76
8.7.2.1	Cymunk	76
8.8	Advanced	77
8.8.1	Why CFFI?	77
8.8.2	Code Layout	78
8.8.3	Tests	78
8.8.4	Working with non-wrapped parts of Chipmunk	79
8.8.5	Weak References and free Methods	79
8.9	Changelog	79
8.9.1	Easymunk 0.9.0 (2021-03-01)	79
8.10	License	79
9	Indices and tables	81
	Python Module Index	83
	Index	85

Easymunk is a easy-to-use pythonic 2d physics library that can be used whenever you need 2d rigid body physics from Python. Perfect when you need 2d physics in your game, demo or other application! It is built on top of the very capable 2d physics library [Chipmunk](#).

Easymunk is a fork of the excellent pymunk project, but it allows itself to deviate more from the original C-library API. The goal is to explore a more Pythonic interface and tends to be easier to use and require less code to accomplish the same effects.

The first version was released in 2021, based on Pymunk 6.0. It owns greatly from Pymunk's maturity and 10 years of active development. Easymunk is a laboratory and we hope to give back code to Pymunk upstream and collaborate with its development.

Pymunk: 2007 - 2020, Victor Blomqvist - vb@viblo.se, MIT License **Easymunk:** 2021, Fábio Macêdo Mendes - fabiomacedomendese@gmail.com, MIT License

INSTALLATION

In the normal case Easymunk can be installed from PyPI with pip:

```
> pip install easymunk-physics
```

It has a few dependencies that are installed automatically.

EXAMPLE

Quick code example:

```
import easymunk as mk          # Import easymunk.

space = mk.Space(              # Create a Space which contain the simulation
    gravity=(0, -10),          # setting its gravity
)

body = space.create_box(       # Create a Body with mass, moment,
    shape=(10, 20),           # position and shape.
    mass=10,
    moment=150,
    position=(50,100),
)

while True:                   # Infinite loop simulation
    space.step(0.01)           # Step the simulation one step forward
    space.debug_draw()         # Print the state of the simulation
```

For more detailed and advanced examples, take a look at the included demos (in examples/).

Examples are not included if you install with *pip install easymunk*. Instead you need to download the source archive (easymunk-x.y.z.zip). Download available from <https://pypi.org/project/easymunk/#files>

DOCUMENTATION

The source distribution of Easymunk ships with a number of demos of different simulations in the examples directory, and it also contains the full documentation including API reference.

You can also find the full documentation including examples and API reference on the Easymunk homepage, <http://fabioimmendes.github.io/easymunk>.

THE EASYMUNK VISION

“Make 2d physics easy to include in your game”

It is (or is striving to be):

- **Easy to use** - It should be easy to use, no complicated code should be needed to add physics to your game or program.
- **“Pythonic”** - It should not be visible that a c-library (Chipmunk) is in the bottom, it should feel like a Python library (no strange naming, no memory handling and more)
- **Simple to build & install** - You shouldn't need to have a zillion of libraries installed to make it install, or do a lot of command line tricks.
- **Multi-platform** - Should work on both Windows, *nix and OSX.
- **Non-intrusive** - It should not put restrictions on how you structure your program and not force you to use a special game loop, it should be possible to use with other libraries like Pygame and Pyglet.

CONTACT & SUPPORT

Homepage <http://fabioimmendes.github.io/easymunk>

Stackoverflow You can ask questions/browse old ones at Stackoverflow, just look for the Easymunk tag. <http://stackoverflow.com/questions/tagged/easymunk>

Issue Tracker Please use the issue tracker at github to report any issues you find: <https://github.com/fabioimmendes/easymunk/issues>

Regardless of the method you use I will try to answer your questions as soon as I see them. (And if you ask on SO other people might help as well!)

DEPENDENCIES / REQUIREMENTS

Basically Easymunk have been made to be as easy to install and distribute as possible, usually *pip install easymunk-physics* will take care of everything for you.

- Python (Runs on CPython 3.8 and later)
- Chipmunk (Compiled library already included on common platforms)
- CFFI (will be installed automatically by Pip)
- Setuptools (should be included with Pip)
- GCC and friends (optional, you need it to compile Easymunk from source. On windows Visual Studio is required to compile)
- Pygame (optional, you need it to run the Pygame based demos)
- Pyglet (optional, you need it to run the Pyglet based demos)
- Pyxel (optional, you need it to run the Pyxel based demos)
- Streamlit (optional, you need it to run the streamlit based demos)
- Matplotlib & Jupyter Notebook (optional, you need it to run the Matplotlib based demos)
- Sphinx & afigure & sphinx_autodoc_typehints (optional, you need it to build documentation)

INSTALL FROM SOURCE / CHIPMUNK COMPILATION

This section is only required in case you do not install easymunk from the prebuild binary wheels (normally if you do not use *pip install* or you are on a uncommon platform).

Easymunk is built on top of the c library Chipmunk. It uses CFFI to interface with the Chipmunk library file. Because of this Chipmunk has to be compiled together with Easymunk as an extension module.

There are basically two options, either building it automatically as part of installation using for example Pip:

```
> pip install easymunk-source-dist.zip
```

And Pip even accepts URL arguments, which can be used to fetch directly a commit or the latest version in main:

```
> pip install https://github.com/fabioimmendes/easymunk/archive/refs/heads/main.zip
```

If you want to contribute to this project or simply want to study Easymunk's code, it is recommended to clone the git repository and build from there:

```
> git clone http://github.com/fabioimmendes/easymunk
```

After cloning, initialize the repository with git submodules:

```
> cd easymunk  
> git submodule update --init --recursive
```

This will download the Chipmunk2D source tree, which is necessary to compile the C-extension module used by easymunk. Now that the source code is available, build the extension module with:

```
> python setup.py build_ext
```

Finally, install it with:

```
> python setup.py develop --user
```

Easymunk requires Python 3.8+.

CONTENTS

8.1 Installation

Tip: You will find the latest released version at pypi: <https://pypi.python.org/pypi/pymunk>

8.1.1 Install Easymunk

Easymunk can be installed with pip install:

```
> pip install easymunk
```

Easymunk can also be installed with conda install, from the conda-forge channel:

```
> conda install -c conda-forge easymunk
```

Sometimes on more uncommon platforms you will need to have a GCC-compatible c-compiler installed.

On OSX you can install one with:

```
> xcode-select --install
```

On Linux you can install one with the package manager, for example on Ubuntu with:

```
> sudo apt-get install build-essential
```

8.1.2 Examples & Documentation

Because of their size the examples and the documentation are available in the source distribution of Easymunk, but not the wheels. The source distribution is available from PyPI at <https://pypi.org/project/easymunk/#files> (Named easymunk-x.y.z.zip)

8.1.3 Troubleshooting

Check that no files are named easymunk.py

Check that conda install works <https://stackoverflow.com/questions/39811929/package-installed-by-conda-python-cannot-find-it>

8.1.4 Advanced - Android Install

Easymunk can run on Android phones/tablets/computers.

8.1.4.1 Kivy

Kivy is a open source Python library for rapid development of applications that make use of innovative user interfaces, such as multi-touch apps, and can run on Android (and a number of other platforms such as Linux, Windows, OS X, iOS and Raspberry Pi).

Easymunk should work out of the box when used with Kivy. Note however that the recipe used to build Easymunk specifies a specific version of Easymunk that might not be the latest, see the recipe script here: https://github.com/kivy/python-for-android/blob/master/pythonforandroid/recipes/pymunk/__init__.py

8.1.4.2 Termux

Termux is an Android terminal emulator and Linux environment app that works directly with no rooting or setup required.

There are no binary wheels of pymunk for Termux/Android, or for its dependency cffi, so you will need to install a couple of packages first, before pymunk can be installed.

1. Install python and other needed dependencies (run inside Termux):

```
$ pkg install python python-dev clang libffi-dev
```

2. Install pymunk with pip:

```
$ pip install pymunk
```

3. Verify that it works:

```
$ python -m pymunk.tests test
```

8.1.5 Advanced - Install

Another option is to use the standard setup.py way, in case you have downloaded the source distribution:

```
> python setup.py install
```

Note that this require a GCC compiler, which can be a bit tricky on Windows. If you are on Mac OS X or Linux you will probably need to run as a privileged user; for example using sudo:

```
> sudo python setup.py install
```

Once installed you should be able to to import pymunk just as any other installed library. pymunk should also work just fine with virtualenv in case you want it installed in a contained environment.

8.1.5.1 Advanced - Running without installation

If you do not want to install Easymunk, for example because you want to bundle it with your code, its also possible to run it directly inplace. Given that you have the source code the first thing to do is to compile chipmunk with the inplace option, as described in the [Compile Chipmunk](#) section.

To actually import pymunk from its folder you need to do a small path hack, since the pymunk root folder (where setup.py and the README are located) is not part of the package. Instead you should add the path to the pymunk package folder (where files such as space.py and body.py are located):

```
mycodefolder/
|-- mycode.py
|-- ...
|-- easymunk/
|   |-- README.rst
|   |-- setup.py
|   |-- easymunk/
|       |-- space.py
|       |-- body.py
|       |-- ...
|       |-- ...
```

Then inside you code file (*mycode.py*) import sys and add the pymunk folder to the path:

```
import sys
sys.path.insert(1, 'easymunk')
import easymunk as mk
```

8.1.6 Compile Chipmunk

If a compiled binary library of Chipmunk that works on your platform is not included in the release you will need to compile Chipmunk yourself. Another reason to compile chipmunk is if you want to run it in release mode to get rid of the debug prints it generates. If you just use pip install the compilation will happen automatically given that a compiler is available. You can also specifically compile Chipmunk as described below.

To compile Chipmunk:

```
> python setup.py build_ext
```

If you got the source and just want to use it directly you probably want to compile Chipmunk in-place, that way the output is put directly into the correct place in the source folder:

```
> python setup.py build_ext --inplace
```

On Windows you will need to use Visual Studio matching your Python version.

8.1.7 CFFI Installation

Sometimes you need to manually install the (non-python) dependencies of CFFI. Usually you will notice this as a installation failure when pip tries to install CFFI since CFFI is a dependency of Easymunk. This is not really part of Easymunk, but a brief description is available for your convenience.

You need to install two extra dependencies for CFFI to install properly. This can be handled by the package manager. The dependencies are *python-dev* and *libffi-dev*. Note that they might have slightly different names depending on the distribution, this is for Debian/Ubuntu. Just install them the normal way, for example like this if you use apt and Pip should be able to install CFFI properly:

```
> sudo apt-get install python-dev libffi-dev
```

8.2 Overview

8.2.1 Basics

There are 4 basic classes you will use in Easymunk.

Rigid Bodies (`easymunk.Body`) A rigid body holds the physical properties of an object. (mass, position, rotation, velocity, etc.) It does not have a shape by itself. If you've done physics with particles before, rigid bodies differ mostly in that they are able to rotate. Rigid bodies generally tend to have a 1:1 correlation to sprites in a game. You should structure your game so that you use the position and rotation of the rigid body for drawing your sprite.

Collision Shapes (`easymunk.Circle`, `easymunk.Segment` and `easymunk.Poly`) By attaching shapes to bodies, you can define the a body's shape. You can attach many shapes to a single body to define a complex shape, or none if it doesn't require a shape.

Constraints/Joints (`easymunk.constraint.PinJoint`, `easymunk.constraint.SimpleMotor` and many others) You can attach constraints between two bodies to constrain their behavior, for example to keep a fixed distance between two bodies.

Spaces (`easymunk.Space`) Spaces are the basic simulation unit in Easymunk. You add bodies, shapes and constraints to a space, and then update the space as a whole. They control how all the rigid bodies, shapes, and constraints interact together.

The actual simulation is done by the Space. After adding the objects that should be simulated to the Space time is moved forward in small steps with the `easymunk.Space.step()` function.

8.2.2 Model your physics objects

8.2.2.1 Object shape

What you see on the screen doesn't necessarily have to be exactly the same shape as the actual physics object. Usually the shape used for collision detection (and other physics simulation) is much simplified version of what is drawn on the screen. Even high end AAA games separate the collision shape from what is drawn on screen.

There are a number of reasons why its good to separate the collision shape and what is drawn.

- Using simpler collision shapes are faster. So if you have a very complicated object, for example a pine tree, maybe it can make sense to simplify its collision shape to a triangle for performance.

- Using a simpler collision shape make the simulation better. Lets say you have a floor made of stone with a small crack in the middle. If you drag a box over this floor it will get stuck on the crack. But if you simplify the floor to just a plane you avoid having to worry about stuff getting stuck in the crack.
- Making the collision shape smaller (or bigger) than the actual object makes gameplay better. Lets say you have a player controlled ship in a shoot-em-up type game. Many times it will feel more fun to play if you make the collision shape a little bit smaller compared to what it should be based on how it looks.

You can see an example of this in the [using_sprites.py](#) example included in Easymunk. There the physics shape is a triangle, but what is drawn is 3 boxes in a pyramid with a snake on top. Another example is in the [platformer.py](#) example, where the player is drawn as a girl in red and gray. However the physics shape is just a couple of circle shapes on top of each other.

8.2.2.2 Mass, weight and units

Sometimes users of Easymunk can be confused as to what unit everything is defined in. For example, is the mass of a Body in grams or kilograms? Easymunk is unit-less and does not care which unit you use. If you pass in seconds to a function expecting time, then your time unit is seconds. If you pass in pixels to functions that expect a distance, then your unit of distance is pixels.

Then derived units are just a combination of the above. So in the case with seconds and pixels the unit of velocity would be pixels / second.

(This is in contrast to some other physics engines which can have fixed units that you should use)

8.2.2.3 Looks before realism

How heavy is a bird in angry birds? It does matter, its a cartoon!

Together with the units another key insight when setting up your simulation is to remember that it is a simulation, and in many cases the look and feel is much more important than actual realism. So for example, if you want to model a flipper game, the real power of the flipper and launchers doesn't matter at all, what is important is that the game feels "right" and is fun to use for your users.

Sometimes it make sense to start out with realistic units, to give you a feel for how big mass should be in comparison to gravity for example.

There are exceptions to this of course, when you actually want realism over the looks. In the end it is up to you as a user of Easymunk to decide.

8.2.3 Game loop / moving time forward

The most important part in your game loop is to keep the `dt` argument to the `easymunk.Space.step()` function constant. A constant time step makes the simulation much more stable and reliable.

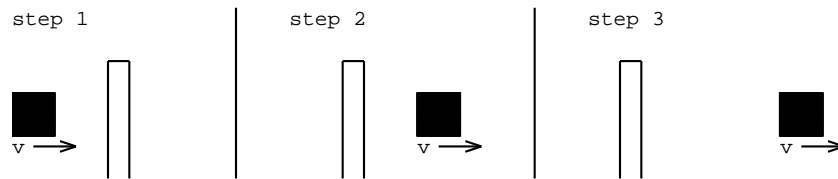
There are several ways to do this, some more complicated than others. Which one is best for a particular program depends on the requirements.

Some good articles:

- <http://gameprogrammingpatterns.com/game-loop.html>
- <http://gafferongames.com/game-physics/fix-your-timestep/>
- <http://www.koonsolo.com/news/dewitters-gameloop/>

8.2.4 Object tunneling

Sometimes an object can pass through another object even though its not supposed to. Usually this happens because the object is moving so fast, that during a single call to `space.step()` the object moves from one side to the other.



There are several ways to mitigate this problem. Sometimes it might be a good idea to do more than one of these.

- Make sure the velocity of objects never get too high. One way to do that is to use a custom velocity function with a limit built in on the bodies that have a tendency to move too fast:

```
def limit_velocity(body, gravity, damping, dt):
    max_velocity = 1000
    easymunk.Body.update_velocity(body, gravity, damping, dt)
    l = body.velocity.length
    if l > max_velocity:
        scale = max_velocity / l
        body.velocity = body.velocity * scale

body_to_limit.velocity_func = limit_velocity
```

Depending on the requirements it might make more sense to clamp the velocity over multiple frames instead. Then the limit function could look like this instead:

```
def limit_velocity(body, gravity, damping, dt):
    max_velocity = 1000
    easymunk.Body.update_velocity(body, gravity, damping, dt)
    if body.velocity.length > max_velocity:
        body.velocity = body.velocity * 0.99
```

- For objects such as bullets, use a space query such as `space.segment_query` or `space.segment_first`.
- Use a smaller value for `dt` in the call to `space.step`. A simple way is to call `space.step` multiple times each frame in your application. This will also help to make the overall simulation more stable.
- Double check that the center of gravity is at a reasonable point for all objects.

8.2.5 Unstable simulation?

Sometimes the simulation might not behave as expected. In extreme cases it can “blow up” and parts move anywhere without logic.

There are a number of things to try if this happens:

- Make all the bodies of similar mass. It is easier for the physics engine to handle bodies with similar weight.
- Don't let two objects with infinite mass touch each other.
- Make the center of gravity in the middle of shapes instead of at the edge.
- Very thin shapes can behave strange, try to make them a little wider.
- Have a fixed time step (see the other sections of this guide).

- Call the `Space.step` function several times with smaller `dt` instead of only one time but with a bigger `dt`. (See the docs of *Space.step*)
- If you use a Motor joint, make sure to set its max force. Otherwise its power will be near infinite.
- Double check that the center of gravity is at a reasonable point for all objects.

(Most of these suggestions are the same for most physics engines, not just Easymunk.)

8.2.6 Performance

Various tips that can improve performance:

- Run Python with optimizations on (will disable various useful but non-critical asserts). `python -O mycode.py`
- Tweak the `Space.iterations` property.
- If possible let objects fall asleep with `Space.sleep_time_threshold`.
- Reduce usage of callback methods (like collision callbacks or custom update functions). These are much slower than the default built in code.

Note that many times the actual simulation is quick enough, but reading out the result after each step and manipulating the objects manually can have a significant overhead and performance cost.

8.2.7 Copy and Load/Save Easymunk objects

Most Easymunk objects can be copied and/or saved with pickle from the standard library. Since the implementation is generic it will also work to use other serializer libraries such as [jsonpickle](#) (in contrast to pickle the jsonpickle serializes to/from json) as long as they make use of the pickle infrastructure.

See the *copy_and_pickle.py* example for an example on how to save, load and copy Easymunk objects.

Note that the version of Easymunk used must be the same for the code saving as the version used when loading the saved object.

8.2.8 Additional info

As a complement to the Easymunk docs it can be good to read the [Chipmunk docs](#). Its made for Chipmunk, but Easymunk is build on top of Chipmunk and share most of the concepts, with the main difference being that Easymunk is used from Python while Chipmunk is a C-library.

8.3 API Reference

8.3.1 easymunk Package

Submodules

8.3.1.1 easymunk.geometry Module

8.3.1.2 easymunk.core Module

8.3.1.3 easymunk.linalg Module

8.3.1.4 easymunk.matplotlib Module

This submodule contains helper functions to help with working with datascience tools such as Jupyter notebooks and Streamlit via matplotlib.

class easymunk.matplotlib.DrawOptions (*ax=None, bb=None, dot_scale=0.1*)

Bases: easymunk.drawing.DrawOptions

__init__ (*ax=None, bb=None, dot_scale=0.1*)

DrawOptions for space.debug_draw() to draw a space on a ax object.

Typical usage:

```
>>> space = mk.Space()
>>> space.debug_draw("matplotlib")
```

You can control the color of a Shape by setting shape.color to the color you want it drawn in.

```
>>> shape = space.static_body.create_circle(10)
>>> shape.color = (1, 0, 0, 1) # will draw shape in red
```

See matplotlib_util.demo.py for a full example

Param

ax: matplotlib.Axes A matplotlib Axes object.

property ax

Return type <class 'Axes'>

draw_circle (*pos, radius, angle=0.0, outline_color=Color(255, 0, 0, 255), fill_color=Color(255, 0, 0, 255)*)

Draw circle from position, radius, angle, and colors.

Return type None

draw_segment (*a, b, color=Color(255, 0, 0, 255)*)

Draw simple thin segment.

Return type None

draw_fat_segment (*a, b, radius=0.0, outline_color=Color(255, 0, 0, 255), fill_color=Color(255, 0, 0, 255)*)

Draw fat segment/capsule.

Return type None

draw_polygon (*verts, radius=0.0, outline_color=Color(255, 0, 0, 255), fill_color=Color(255, 0, 0, 255)*)

Draw polygon from list of vertices.

Return type None

draw_dot (*size, pos, color*)

Draw a dot/point.

Return type None

finalize_frame ()

Executed after debug-draw. The default implementation is a NO-OP.

DRAW_COLLISION_POINTS = 4

Draw collision points.

Use on the flags property to control if collision points should be drawn or not.

DRAW_CONSTRAINTS = 2

Draw constraints.

Use on the flags property to control if constraints should be drawn or not.

DRAW_SHAPES = 1

Draw shapes.

Use on the flags property to control if shapes should be drawn or not.

property collision_point_color

The color of collisions.

Should be a tuple of 4 ints between 0 and 255 (r, g, b, a).

color_for_shape (*shape*)

Return type Color

property constraint_color

The color of constraints.

Should be a tuple of 4 ints between 0 and 255 (r, g, b, a).

draw_bb (*bb*)

Draw bounding box.

Return type None

draw_circle_shape (*circle*)

Default implementation that draws a circular shape.

This function is not affected by overriding the draw method of shape.

Return type None

draw_object (*obj*)

Draw Easymunk object.

draw_poly_shape (*shape*)

Default implementation that draws a polygonal shape.

This function is not affected by overriding the draw method of shape.

Return type None

draw_segment_shape (*shape*)

Default implementation that draws a segment shape.

This function is not affected by overriding the draw method of shape.

Return type None

draw_shape (*shape*)

Draw shape using other drawing primitives.

Return type None

draw_vec2d (*vec*)

Draw point from vector.

property flags

Bit flags which of shapes, joints and collisions should be drawn.

By default all 3 flags are set, meaning shapes, joints and collisions will be drawn.

Example using the basic text only DebugDraw implementation (normally you would the desired backend instead, such as *pygame_util.DrawOptions* or *pyglet_util.DrawOptions*):

```
shape_dynamic_color = Color(52, 152, 219, 255)
```

```
shape_kinematic_color = Color(39, 174, 96, 255)
```

property shape_outline_color

The outline color of shapes.

Should be a tuple of 4 ints between 0 and 255 (r, g, b, a).

```
shape_sleeping_color = Color(114, 148, 168, 255)
```

```
shape_static_color = Color(149, 165, 166, 255)
```

8.3.1.5 easymunk.pygame Module

This submodule contains helper functions to help with quick prototyping using easymunk together with pygame.

Intended to help with debugging and prototyping, not for actual production use in a full application. The methods contained in this module is opinionated about your coordinate system and not in any way optimized.

class easymunk.pygame.**DrawOptions** (*surface=None, flip_y=False*)

Bases: easymunk.drawing.DrawOptions

surface: None.Surface

draw_circle (*pos, radius, angle=0.0, outline_color=Color(255, 0, 0, 255), fill_color=Color(255, 0, 0, 255)*)

Draw circle from position, radius, angle, and colors.

Return type None

draw_segment (*a, b, color=Color(255, 0, 0, 255)*)

Draw simple thin segment.

Return type None

draw_fat_segment (*a, b, radius=0.0, outline_color=Color(255, 0, 0, 255), fill_color=Color(255, 0, 0, 255)*)

Draw fat segment/capsule.

Return type None

draw_polygon (*verts, radius=0.0, outline_color=Color(255, 0, 0, 255), fill_color=Color(255, 0, 0, 255)*)

Draw polygon from list of vertices.

Return type None

draw_dot (*size, pos, color*)

Draw a dot/point.

Return type None

DRAW_COLLISION_POINTS = 4

Draw collision points.

Use on the flags property to control if collision points should be drawn or not.

DRAW_CONSTRAINTS = 2

Draw constraints.

Use on the flags property to control if constraints should be drawn or not.

DRAW_SHAPES = 1

Draw shapes.

Use on the flags property to control if shapes should be drawn or not.

property collision_point_color

The color of collisions.

Should be a tuple of 4 ints between 0 and 255 (r, g, b, a).

color_for_shape (*shape*)

Return type Color

property constraint_color

The color of constraints.

Should be a tuple of 4 ints between 0 and 255 (r, g, b, a).

draw_bb (*bb*)

Draw bounding box.

Return type None

draw_circle_shape (*circle*)

Default implementation that draws a circular shape.

This function is not affected by overriding the draw method of shape.

Return type None

draw_object (*obj*)

Draw Easymunk object.

draw_poly_shape (*shape*)

Default implementation that draws a polygonal shape.

This function is not affected by overriding the draw method of shape.

Return type None

draw_segment_shape (*shape*)

Default implementation that draws a segment shape.

This function is not affected by overriding the draw method of shape.

Return type None

draw_shape (*shape*)

Draw shape using other drawing primitives.

Return type None

draw_vec2d (*vec*)

Draw point from vector.

finalize_frame ()

Executed after debug-draw. The default implementation is a NO-OP.

property flags

Bit flags which of shapes, joints and collisions should be drawn.

By default all 3 flags are set, meaning shapes, joints and collisions will be drawn.

Example using the basic text only DebugDraw implementation (normally you would use the desired backend instead, such as *pygame_util.DrawOptions* or *pyglet_util.DrawOptions*):

mouse_pos ()

Get position of the mouse pointer in pymunk coordinates.

Return type *Vec2d*

shape_dynamic_color = *Color*(52, 152, 219, 255)

shape_kinematic_color = *Color*(39, 174, 96, 255)

property shape_outline_color

The outline color of shapes.

Should be a tuple of 4 ints between 0 and 255 (r, g, b, a).

shape_sleeping_color = *Color*(114, 148, 168, 255)

shape_static_color = *Color*(149, 165, 166, 255)

to_pygame (*p*, *surface=None*)

Convenience method to convert pymunk coordinates to pygame surface local coordinates.

Note that in case *positive_y_is_up* is False, this function won't actually do anything except converting the point to integers.

Return type *Vec2d*

from_pygame (*p*)

Convenience method to convert pygame surface local coordinates to pymunk coordinates

Return type *Vec2d*

8.3.1.6 easymunk.pyglet Module

This submodule contains helper functions to help with quick prototyping using easymunk together with pyglet.

Intended to help with debugging and prototyping, not for actual production use in a full application. The methods contained in this module are opinionated about your coordinate system and not very optimized (they use batched drawing, but there is probably room for optimizations still).

class *easymunk.pyglet.DrawOptions* (***kwargs*)

Bases: *easymunk.drawing.DrawOptions*

draw_circle (*pos*, *radius*, *angle=0.0*, *outline_color=Color*(255, 0, 0, 255), *fill_color=Color*(255, 0, 0, 255))

Draw circle from position, radius, angle, and colors.

Return type *None*

draw_segment (*a*, *b*, *color=Color*(255, 0, 0, 255))

Draw simple thin segment.

Return type None

draw_fat_segment (*a, b, radius=0.0, outline_color=Color(255, 0, 0, 255), fill_color=Color(255, 0, 0, 255)*)

Draw fat segment/capsule.

Return type None

draw_polygon (*verts, radius=0.0, outline_color=Color(255, 0, 0, 255), fill_color=Color(255, 0, 0, 255)*)

Draw polygon from list of vertices.

Return type None

draw_dot (*size, pos, color*)

Draw a dot/point.

Return type None

DRAW_COLLISION_POINTS = 4

Draw collision points.

Use on the flags property to control if collision points should be drawn or not.

DRAW_CONSTRAINTS = 2

Draw constraints.

Use on the flags property to control if constraints should be drawn or not.

DRAW_SHAPES = 1

Draw shapes.

Use on the flags property to control if shapes should be drawn or not.

property collision_point_color

The color of collisions.

Should be a tuple of 4 ints between 0 and 255 (r, g, b, a).

color_for_shape (*shape*)

Return type Color

property constraint_color

The color of constraints.

Should be a tuple of 4 ints between 0 and 255 (r, g, b, a).

draw_bb (*bb*)

Draw bounding box.

Return type None

draw_circle_shape (*circle*)

Default implementation that draws a circular shape.

This function is not affected by overriding the draw method of shape.

Return type None

draw_object (*obj*)

Draw Easymunk object.

draw_poly_shape (*shape*)

Default implementation that draws a polygonal shape.

This function is not affected by overriding the draw method of shape.

Return type None

draw_segment_shape (*shape*)

Default implementation that draws a segment shape.

This function is not affected by overriding the draw method of shape.

Return type None

draw_shape (*shape*)

Draw shape using other drawing primitives.

Return type None

draw_vec2d (*vec*)

Draw point from vector.

finalize_frame ()

Executed after debug-draw. The default implementation is a NO-OP.

property flags

Bit flags which of shapes, joints and collisions should be drawn.

By default all 3 flags are set, meaning shapes, joints and collisions will be drawn.

Example using the basic text only DebugDraw implementation (normally you would use the desired backend instead, such as *pygame_util.DrawOptions* or *pyglet_util.DrawOptions*):

```
shape_dynamic_color = Color(52, 152, 219, 255)
```

```
shape_kinematic_color = Color(39, 174, 96, 255)
```

property shape_outline_color

The outline color of shapes.

Should be a tuple of 4 ints between 0 and 255 (r, g, b, a).

```
shape_sleeping_color = Color(114, 148, 168, 255)
```

```
shape_static_color = Color(149, 165, 166, 255)
```

8.3.1.7 easymunk.pyxel Module

This submodule contains helper functions to help with quick prototyping using easymunk together with pyxel.

Intended to help with debugging and prototyping, not for actual production use in a full application. The methods contained in this module are opinionated about your coordinate system and not very optimized (they use batched drawing, but there is probably room for optimizations still).

Easymunk

Pymunk is a easy-to-use pythonic 2d physics library that can be used whenever you need 2d rigid body physics from Python.

Homepage: <http://www.easymunk.org>

This is the main containing module of easymunk. It contains among other things the very central Space, Body and Shape classes.

```
easymunk.chipmunk_version: str = '7.0.3-080c51480f018040b567e8f0440b121ae3acbae4 '
```

The Chipmunk version used with this Pymunk version.

This property does not show a valid value in the compiled documentation, only when you actually import easymunk and do `easymunk.chipmunk_version`

The string is in the following format: `<cpVersionString>R<github commit of chipmunk>` where `cpVersionString` is a version string set by Chipmunk and the git commit hash corresponds to the git hash of the chipmunk source from github.com/viblo/Chipmunk2D included with easymunk.

8.4 Examples

Here you will find a list of the included examples. Each example have a short description and a screenshot (if applicable).

To look at the source code of an example open it on github by following the link. The examples are also included in the source distribution of Easymunk (but not if you install using the wheel file). You can find the source distribution at PyPI, <https://pypi.org/project/pymunk/#files> (file named `pymunk-x.y.z.zip`).

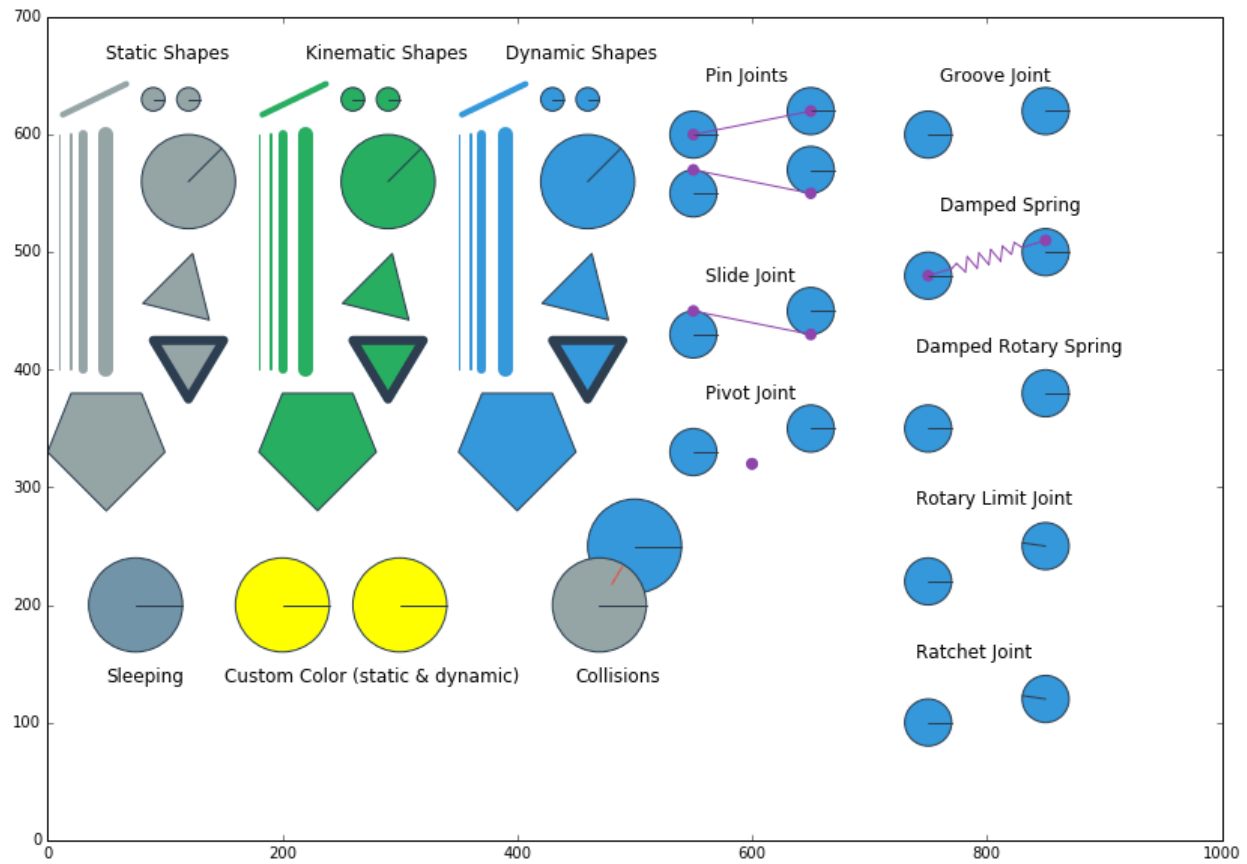
8.4.1 Jupyter Notebooks

There are a couple examples that are provided as Jupyter Notebooks (.ipynb). They are possible to either view online in a browser directly on github, or opened as a Notebook.

8.4.1.1 matplotlib_util_demo.ipynb

Displays the same space as the pygame and pygamelet draw demos, but using matplotlib and the notebook.

Source: [examples/matplotlib_util_demo.ipynb](#)



8.4.1.2 newtons_cradle.ipynb

Similar simulation as newtons_cradle.py, but this time as a Notebook. Compared to the draw demo this demo will output a animation of the simulated space.

Source: [examples/newtons_cradle.ipynb](#)

8.4.2 Standalone Python

To run the examples yourself either install easymunk or run it using the convenience run.py script.

Given that easymunk is installed where your python will find it:

```
>cd examples
>python breakout.py
```

Each example contains something unique. Not all of the examples use the same style. For example, some use the easymunk.pygame_util module to draw stuff, others contain the actual drawing code themselves. However, each example is self contained. Except for external libraries (such as pygame) and easymunk each example can be run directly to make it easy to read the code and understand what happens even if it means that some code is repeated for each example.

If you have made something that uses easymunk and would like it displayed here or in a showcase section of the site, feel free to contact me!

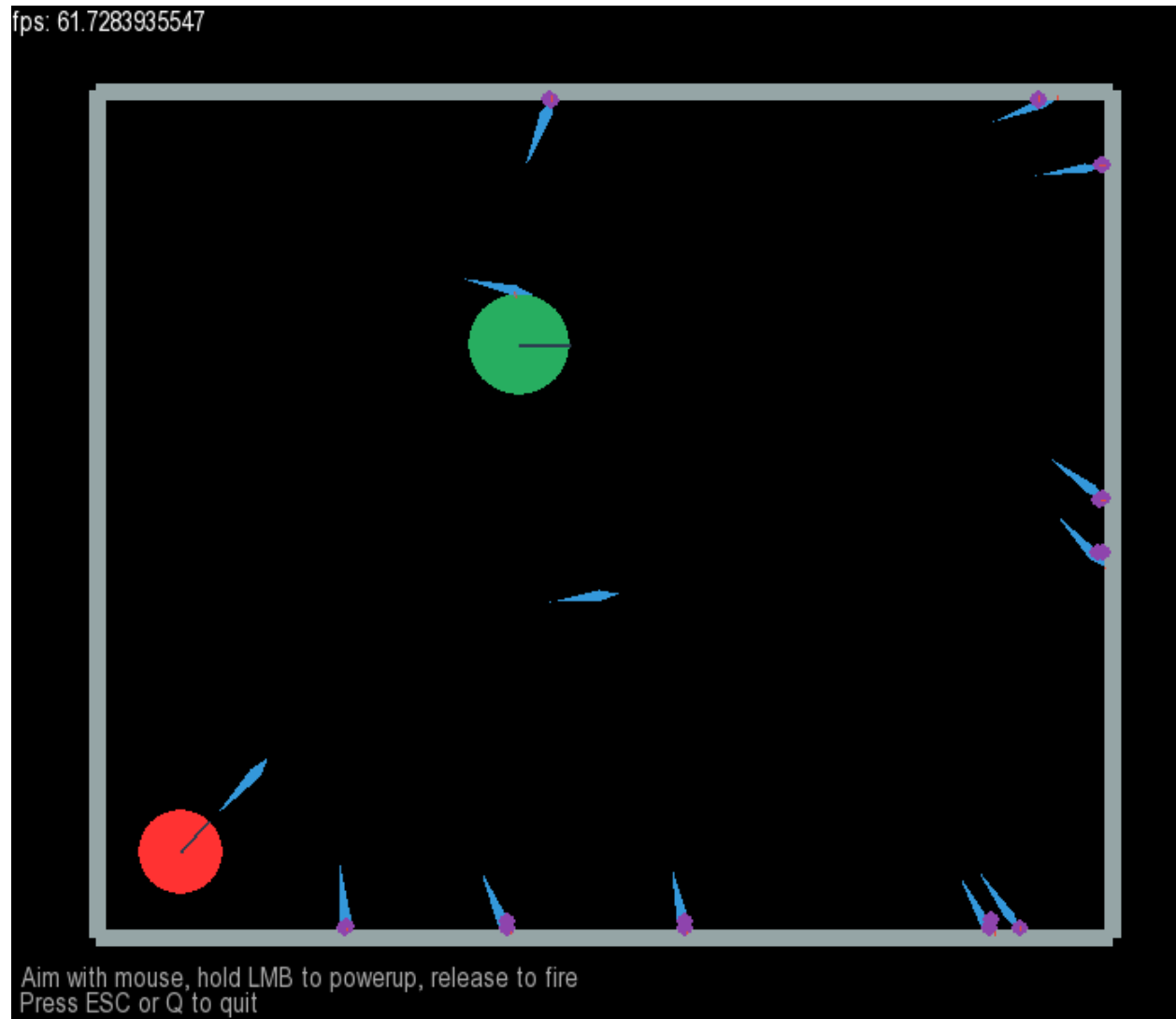
Example files

- *arrows.py*
- *balls_and_lines.py*
- *basic_test.py*
- *bouncing_balls.py*
- *box2d_pyramid.py*
- *box2d_vertical_stack.py*
- *breakout.py*
- *constraints.py*
- *contact_and_no_flip.py*
- *contact_with_friction.py*
- *copy_and_pickle.py*
- *damped_rotary_spring_pointer.py*
- *deformable.py*
- *flipper.py*
- *index_video.py*
- *kivy_pymunk_demo*
- *logo.py*
- *newtons_cradle.py*
- *platformer.py*
- *playground.py*
- *point_query.py*
- *py2exe_setup__basic_test.py*
- *py2exe_setup__breakout.py*
- *pygame_demo.py*
- *pyglet_demo.py*
- *shapes_for_draw_demos.py*
- *slide_and_pinjoint.py*
- *spiderweb.py*
- *tangram.py*
- *tank.py*
- *using_sprites.py*
- *using_sprites_pyglet.py*

8.4.2.1 arrows.py

Source: [examples/arrows.py](#)

Showcase of flying arrows that can stick to objects in a somewhat realistic looking way.

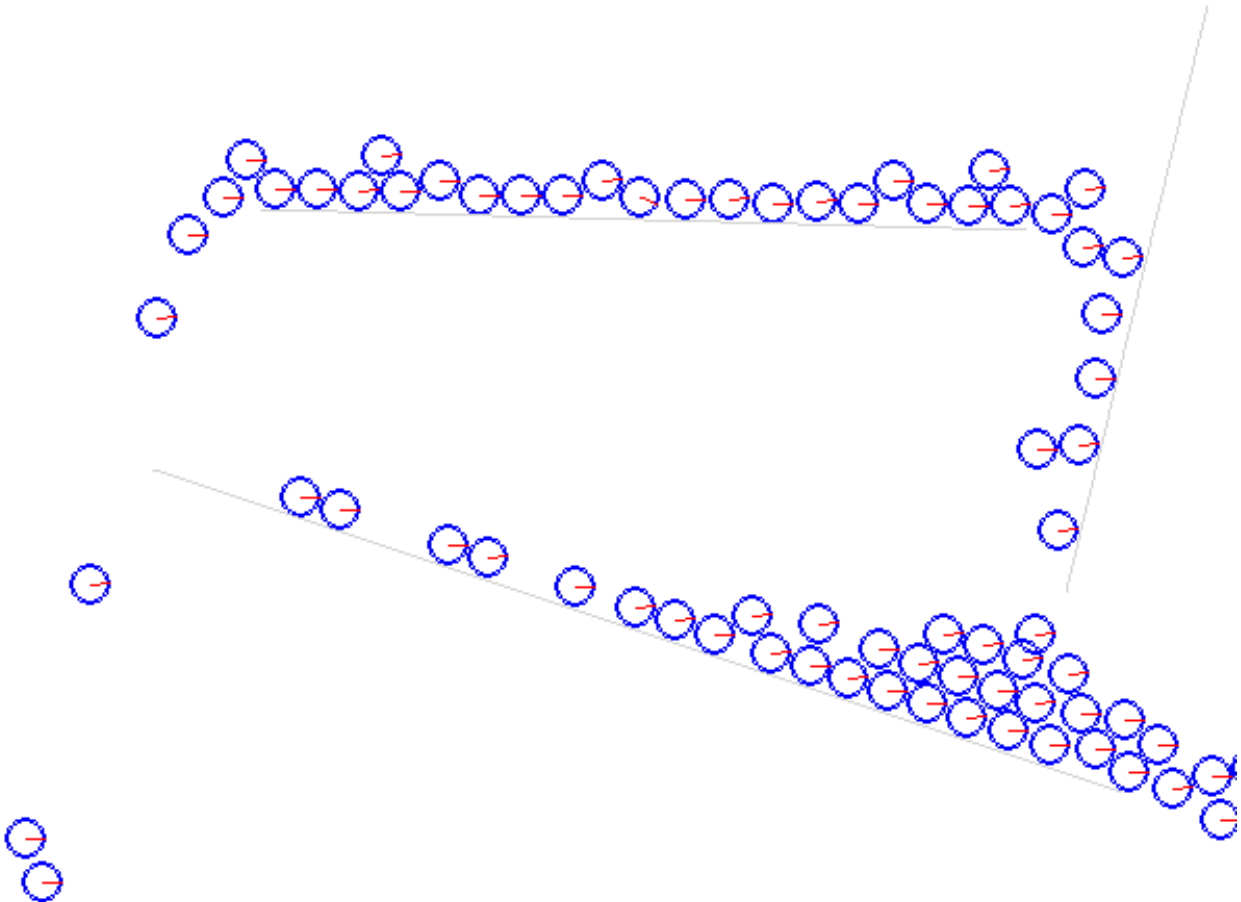


8.4.2.2 balls_and_lines.py

Source: `examples/balls_and_lines.py`

This example lets you dynamically create static walls and dynamic balls

LMB: Create ball
LMB + Shift: Create many balls
RMB: Drag to create wall, release to finish
Space: Pause physics simulation



8.4.2.3 basic_test.py

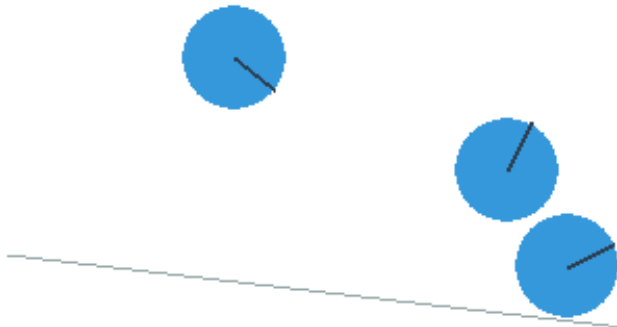
Source: [examples/basic_test.py](#)

Very simple example that does not depend on any third party library such as pygame or pyglet like the other examples.

8.4.2.4 bouncing_balls.py

Source: [examples/bouncing_balls.py](#)

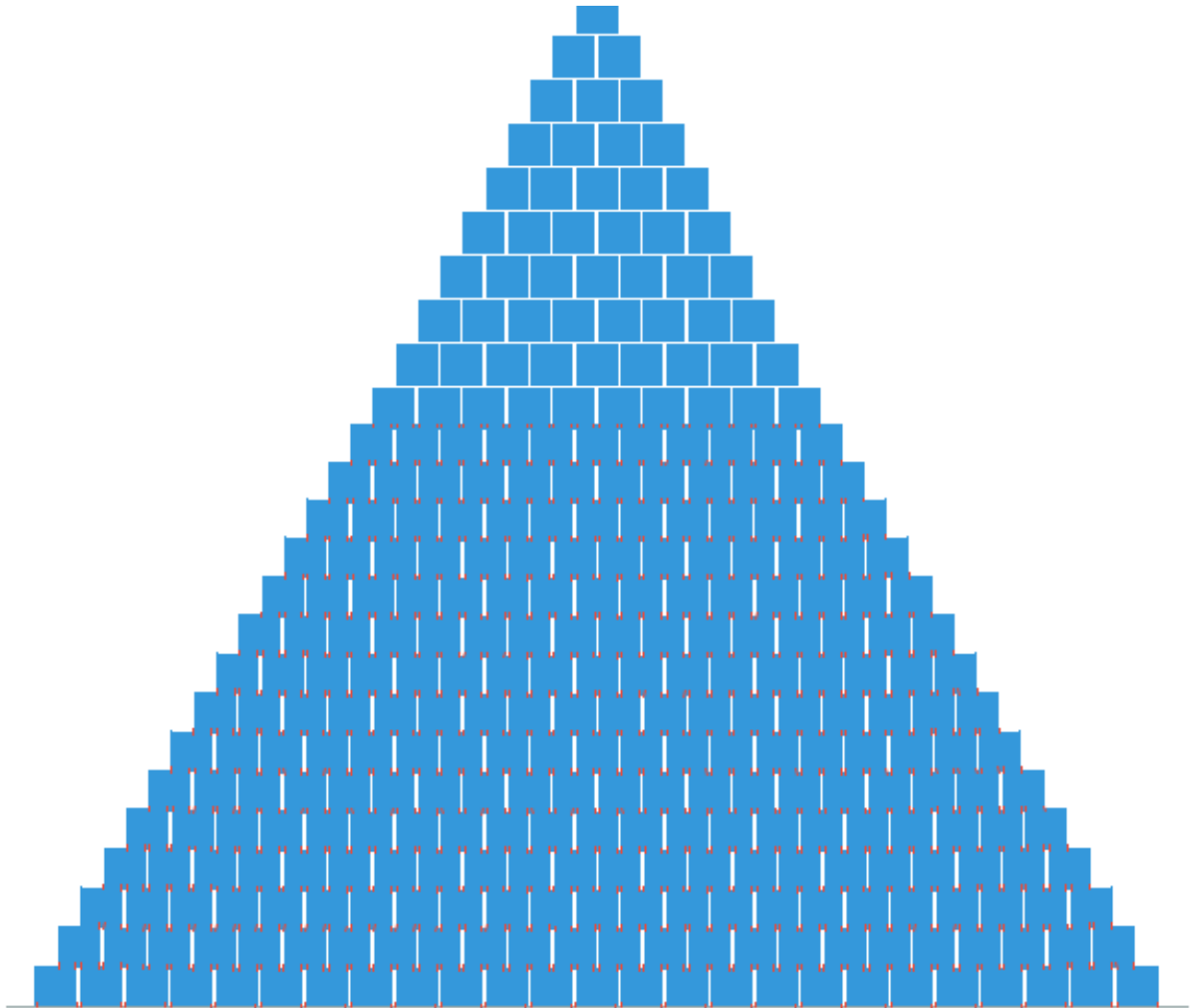
This example spawns (bouncing) balls randomly on a L-shape constructed of two segment shapes. Not interactive.



8.4.2.5 box2d_pyramid.py

Source: [examples/box2d_pyramid.py](#)

Remake of the pyramid demo from the box2d testbed.

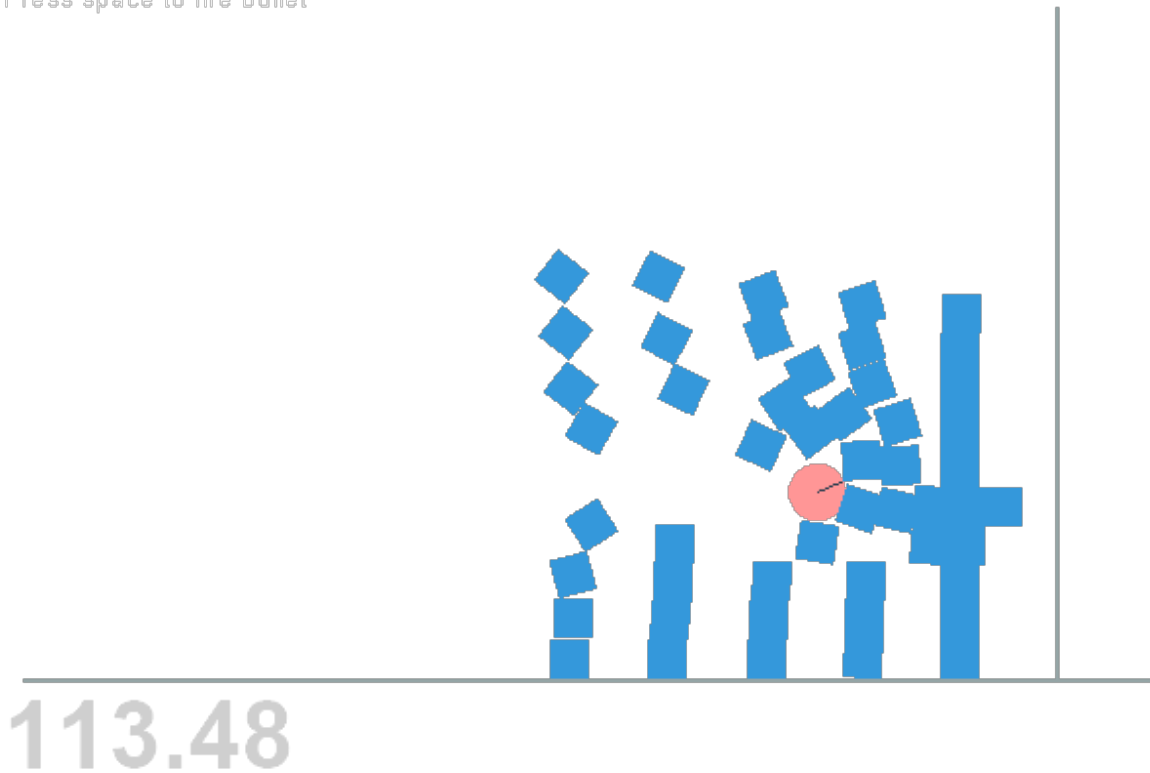


8.4.2.6 box2d_vertical_stack.py

Source: [examples/box2d_vertical_stack.py](#)

Remake of the veritcal stack demo from the box2d testbed.

Press space to fire bullet

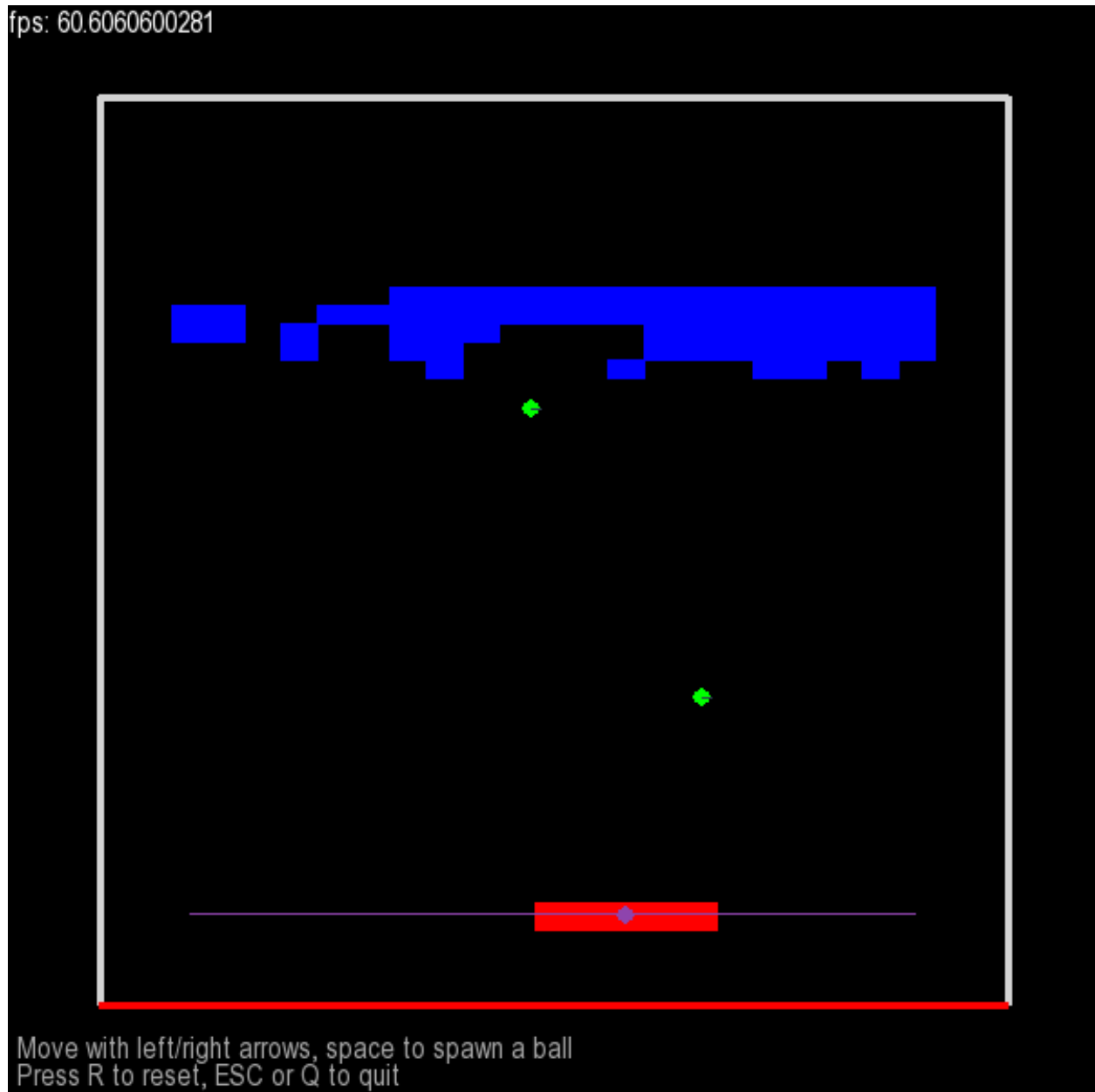


8.4.2.7 breakout.py

Source: [examples/breakout.py](#)

Very simple breakout clone. A circle shape serves as the paddle, then breakable bricks constructed of Poly-shapes.

The code showcases several pymunk concepts such as elasticity, impulses, constant object speed, joints, collision handlers and post step callbacks.

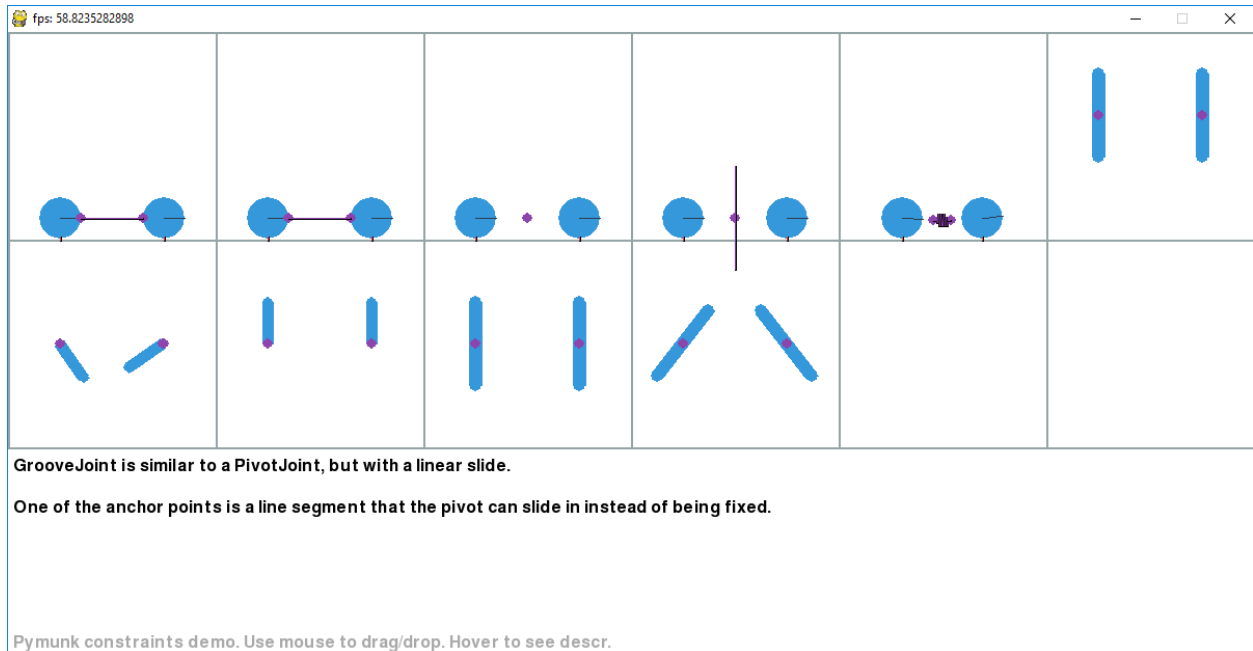


8.4.2.8 constraints.py

Source: [examples/constraints.py](#)

Pymunk constraints demo. Showcase of all the constraints included in easymunk.

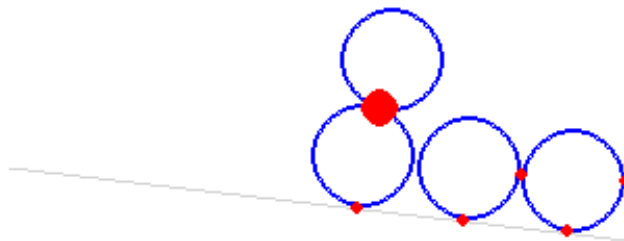
Adapted from the Chipmunk Joints demo: <https://github.com/slembcke/Chipmunk2D/blob/master/demo/Joints.c>



8.4.2.9 contact_and_no_flipy.py

Source: [examples/contact_and_no_flipy.py](#)

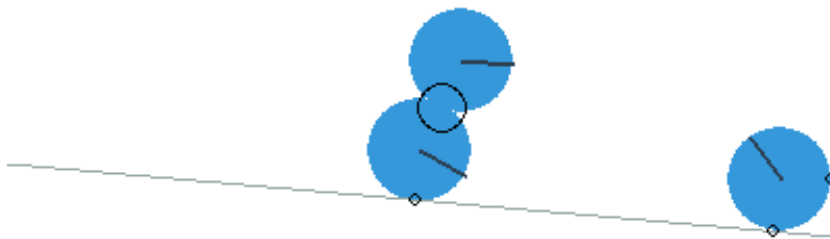
This example spawns (bouncing) balls randomly on a L-shape constructed of two segment shapes. For each collision it draws a red circle with size depending on collision strength. Not interactive.



8.4.2.10 `contact_with_friction.py`

Source: [examples/contact_with_friction.py](#)

This example spawns (bouncing) balls randomly on a L-shape constructed of two segment shapes. Displays collision strength and rotating balls thanks to friction. Not interactive.



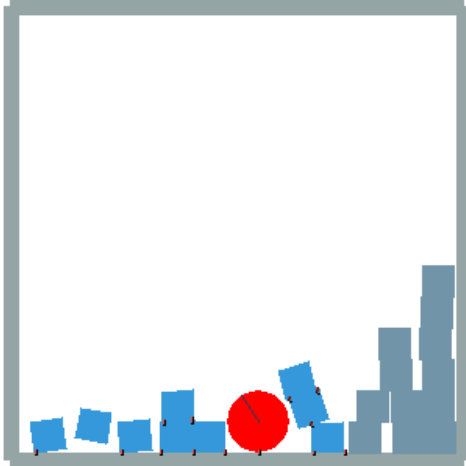
8.4.2.11 `copy_and_pickle.py`

Source: [examples/copy_and_pickle.py](#)

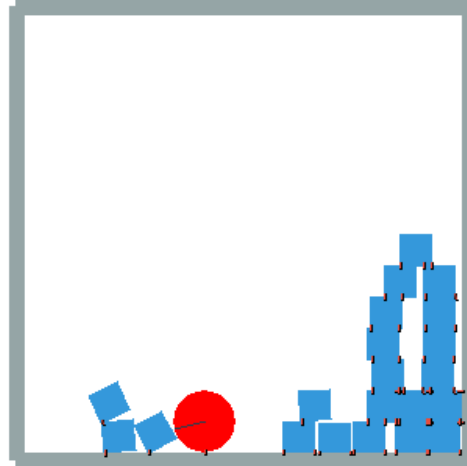
This example shows how you can copy, save and load a space using pickle.

fps: 60.2409629822

space.sleep_time_threshold set to 0.5 seconds



space.sleep_time_threshold set to inf (disabled)



Press SPACE to give an impulse to the ball.
Press S to save the current state to file, press L to load it.
Press R to reset, ESC or Q to quit

8.4.2.12 damped_rotary_spring_pointer.py

Source: [examples/damped_rotary_spring_pointer.py](#)

This example showcase an arrow pointing or aiming towards the cursor.

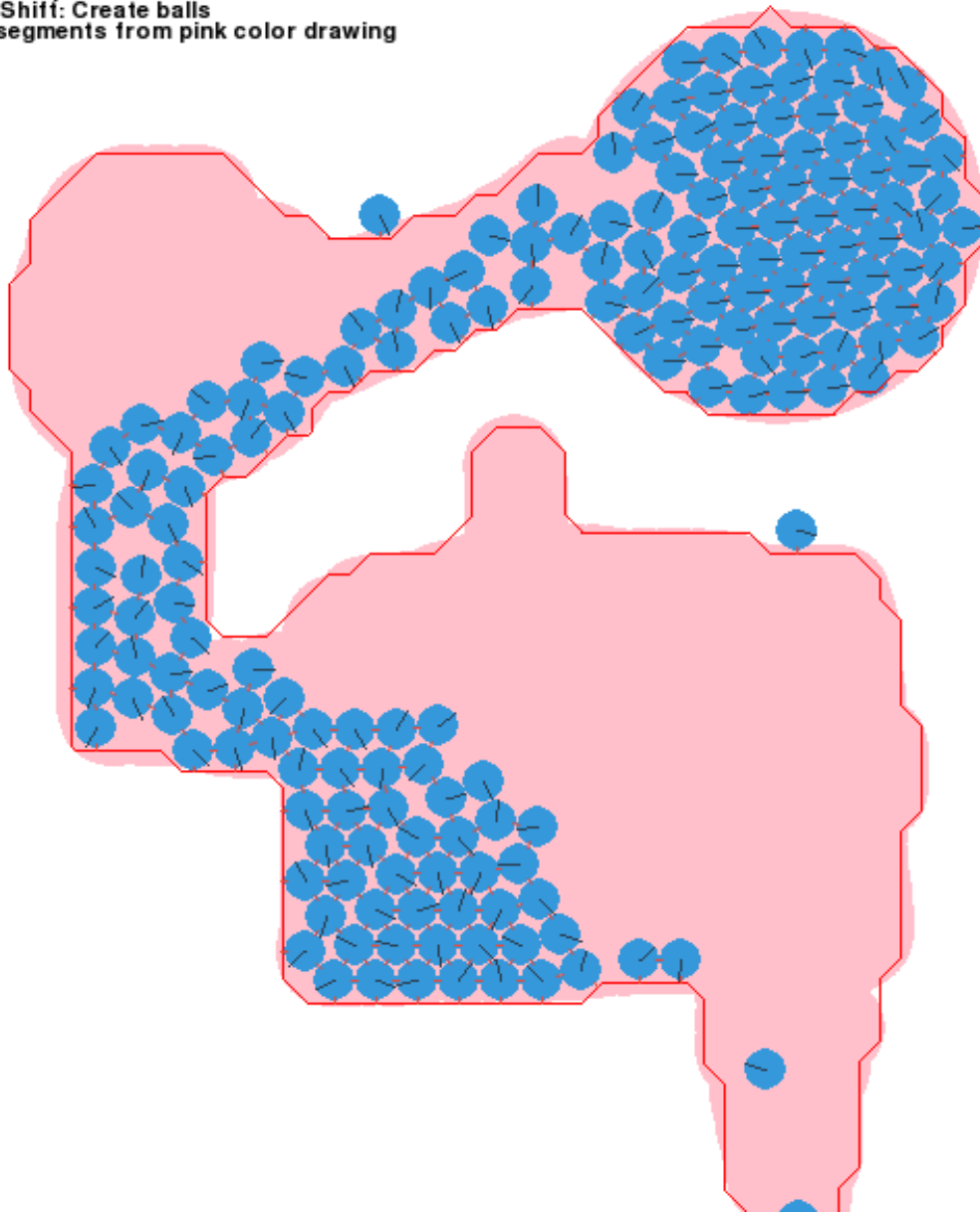


8.4.2.13 deformable.py

Source: [examples/deformable.py](#)

This is an example on how the autogeometry can be used for deformable terrain.

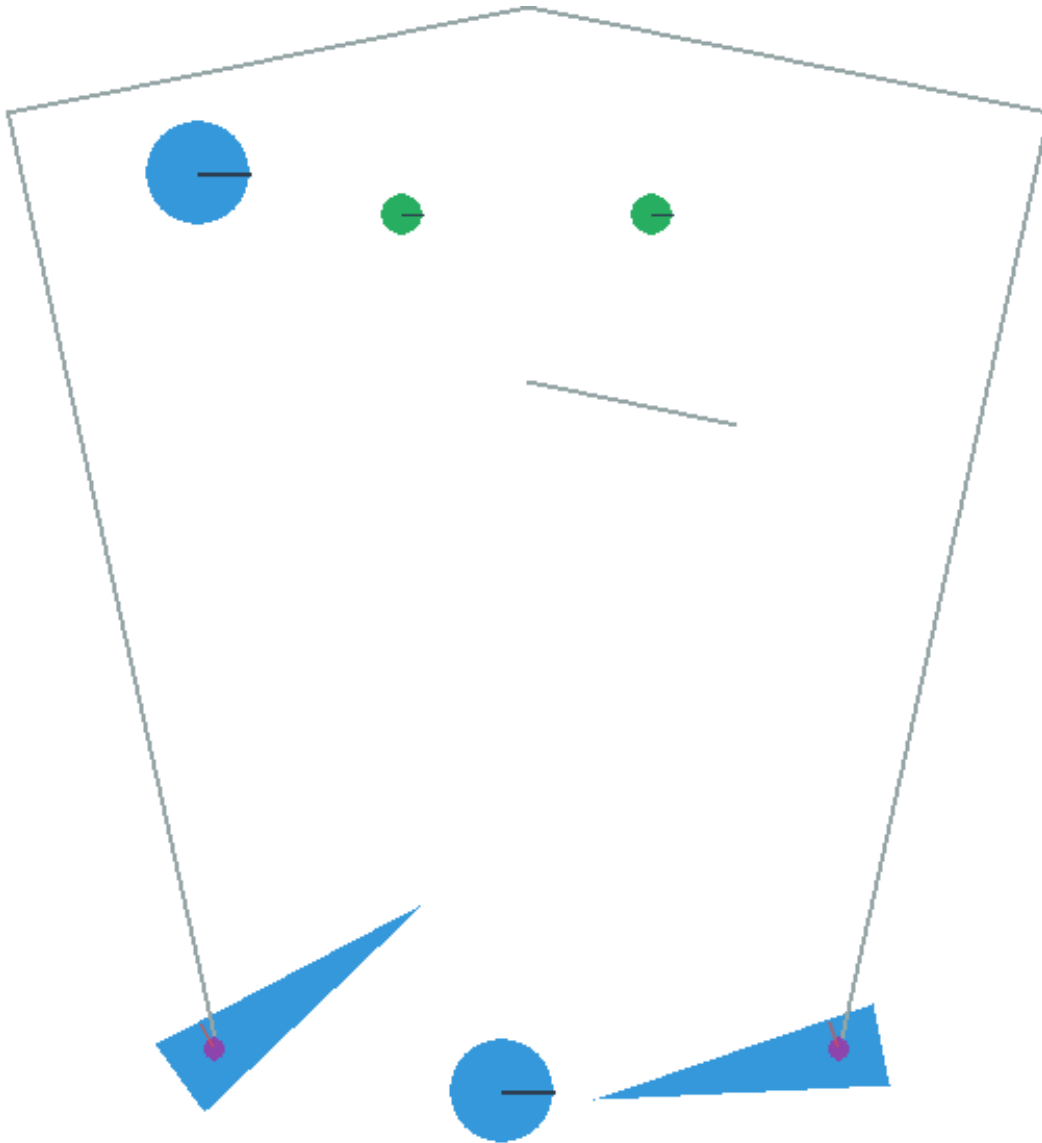
LMB(hold): Draw pink color
LMB(hold) + Shift: Create balls
g: Generate segments from pink color drawing
r: Reset



8.4.2.14 flipper.py

Source: [examples/flipper.py](#)

A very basic flipper game.

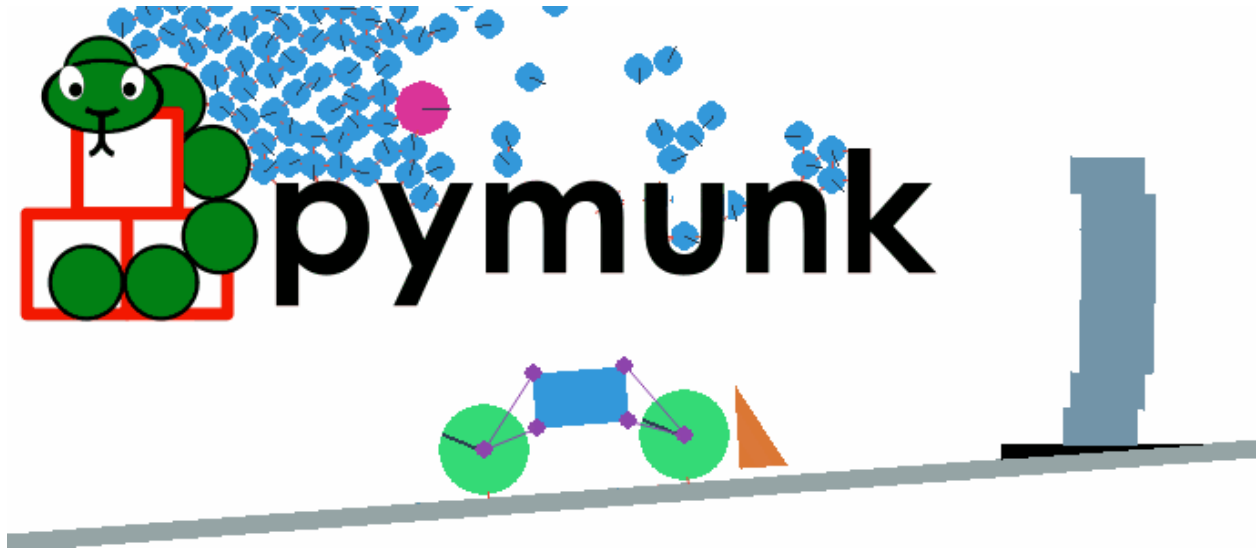


8.4.2.15 index_video.py

Source: [examples/index_video.py](#)

Program used to generate the logo animation on the pymunk main page.

This program will showcase several features of Pymunk, such as collisions, debug drawing, automatic generation of shapes from images, motors, joints and sleeping bodies.

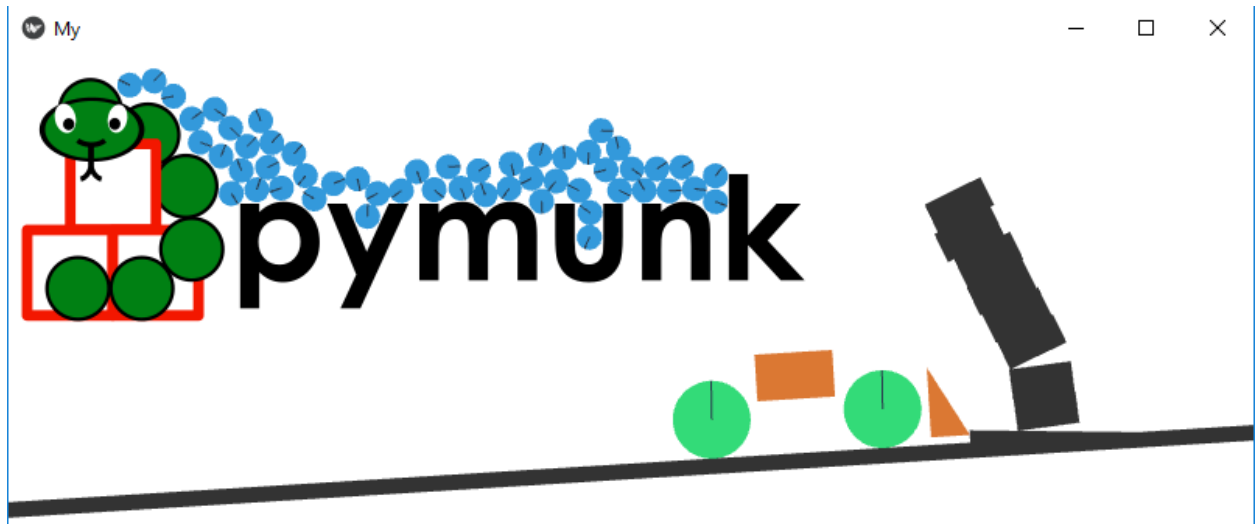


8.4.2.16 kivy_pymunk_demo

Source: [examples/kivy_pymunk_demo](#)

A rudimentary port of the intro video used for the intro animation on [pymunk.org](#). The code is tested on both Windows and Android.

Note that it doesn't display Kivy best practices, the `intro_video` code was just converted to Kivy in the most basic way to show that its possible, its not supposed to show the best way to structure a Kivy application using Pymunk.



8.4.2.17 logo.py

Source: [examples/logo.py](#)

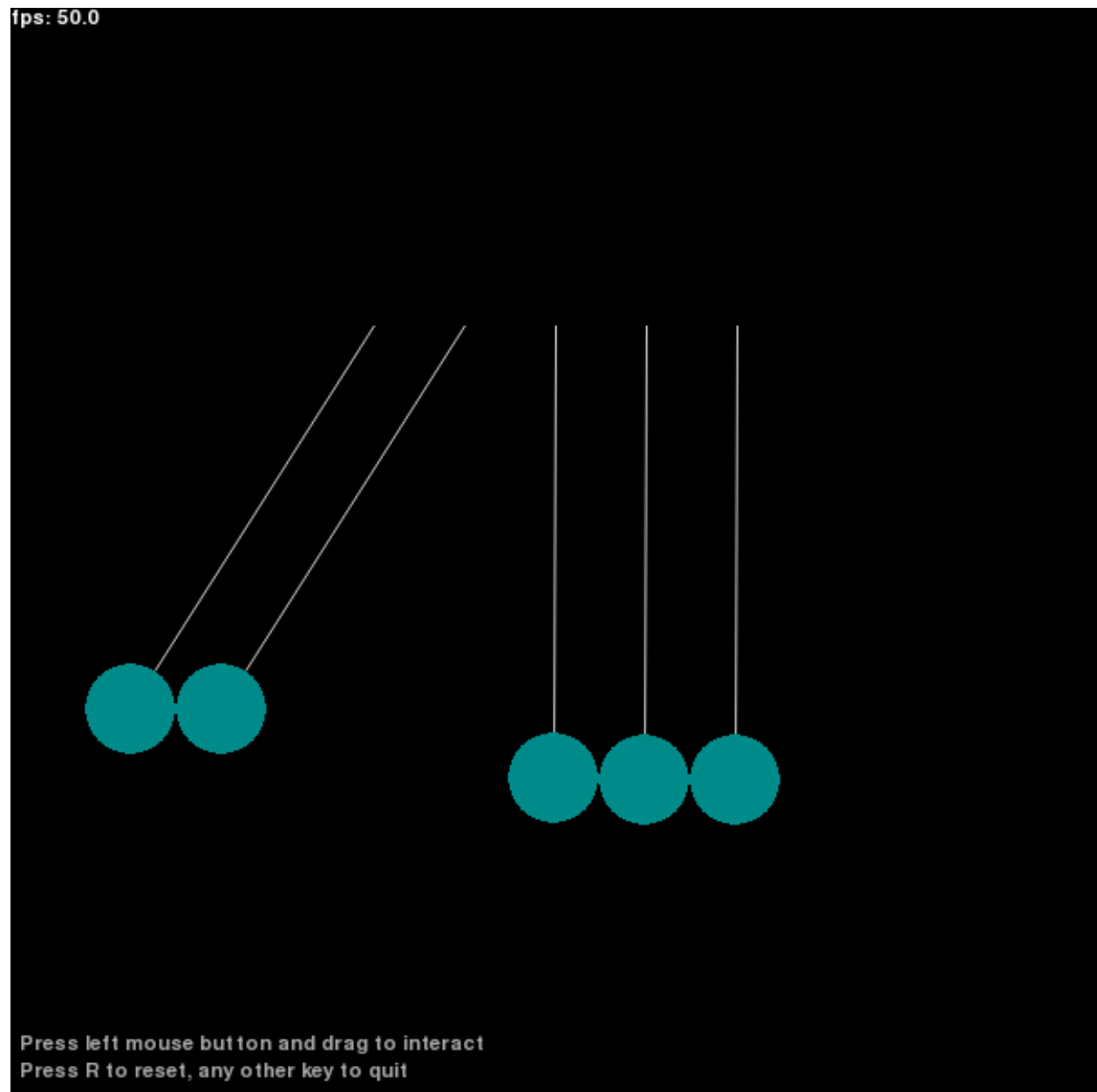
A very simple script to create the Easymunk Logo from pixel-art.

It opens a screen with the logo. Press Alt+1 to save a screenshot to the desktop.

8.4.2.18 newtons_cradle.py

Source: [examples/newtons_cradle.py](#)

A screensaver version of Newton's Cradle with an interactive mode.



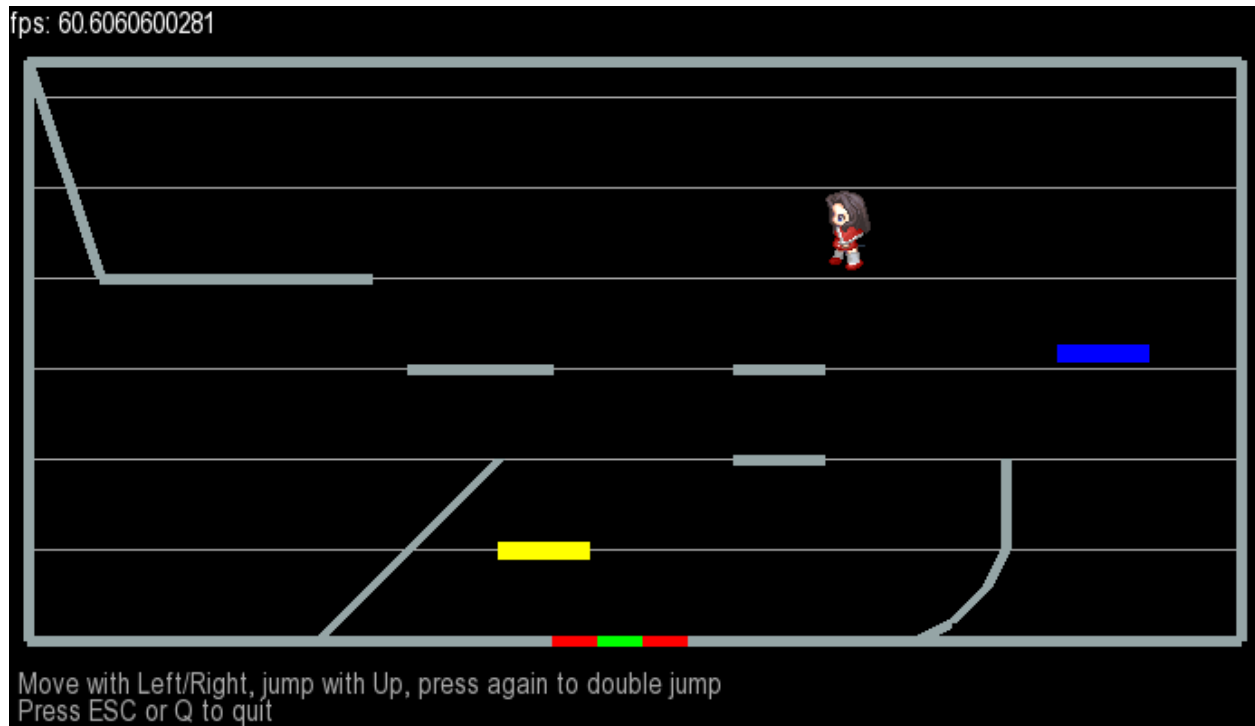
8.4.2.19 platformer.py

Source: [examples/platformer.py](#)

Showcase of a very basic 2d platformer

The red girl sprite is taken from Sithjester's RMXP Resources: <http://untamed.wild-refuge.net/rmxpresources.php?characters>

Note: The code of this example is a bit messy. If you adapt this to your own code you might want to structure it a bit differently.

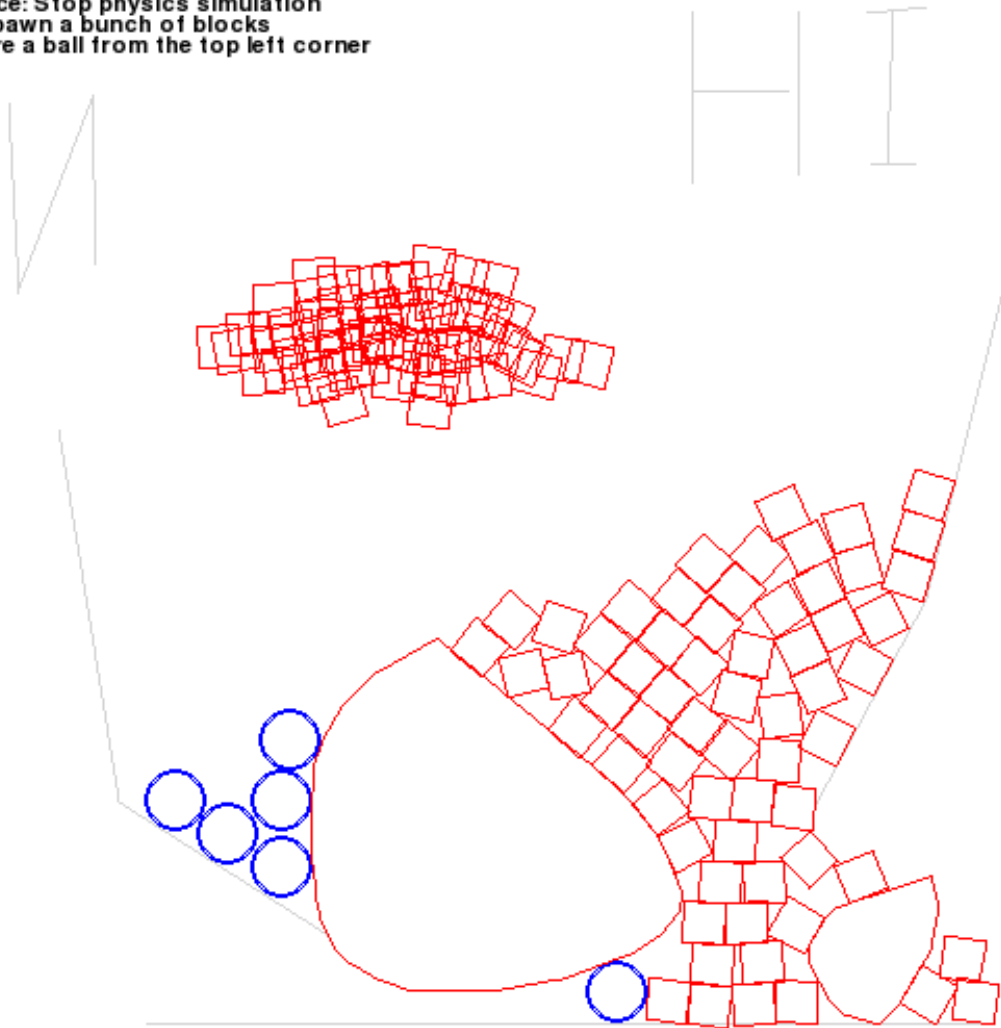


8.4.2.20 playground.py

Source: [examples/playground.py](#)

A basic playground. Most interesting function is draw a shape, basically move the mouse as you want and pymunk will approximate a Poly shape from the drawing.

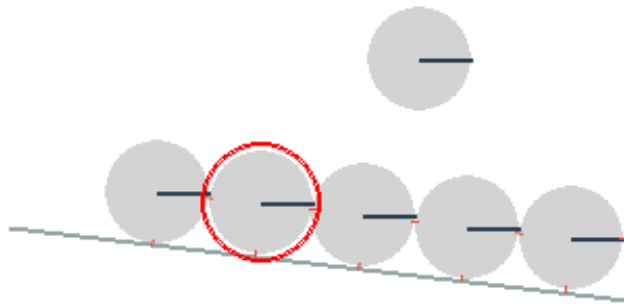
LMB: Create ball
LMB + Shift: Create box
RMB on object: Remove object
RMB(hold) + Shift: Create polygon, release to finish (we be converted to a convex hull of the points)
RMB + Ctrl: Create wall, release to finish
Space: Stop physics simulation
k: Spawn a bunch of blocks
t: Fire a ball from the top left corner



8.4.2.21 point_query.py

Source: [examples/point_query.py](#)

This example showcase point queries by highlighting the shape under the mouse pointer.



8.4.2.22 py2exe_setup__basic_test.py

Source: [examples/py2exe_setup__basic_test.py](#)

Simple example of py2exe to create a exe of the basic_test example.

Tested on py2exe 0.9.2.2 on python 3.4

8.4.2.23 py2exe_setup_breakout.py

Source: [examples/py2exe_setup_breakout.py](#)

Example script to create a exe of the breakout example using py2exe.

Tested on py2exe 0.9.2.2 on python 3.4

8.4.2.24 pygame_demo.py

Source: [examples/pygame_demo.py](#)

Showcase what the output of easymunk.pygame_util draw methods will look like.

See pygamelet_util_demo.py for a comparison to pygamelet.

8.4.2.25 pygamelet_demo.py

Source: [examples/pyglet_demo.py](#)

Showcase what the output of easymunk.pyglet_util draw methods will look like.

See pygame_util_demo.py for a comparison to pygame.

8.4.2.26 shapes_for_draw_demos.py

Source: [examples/shapes_for_draw_demos.py](#)

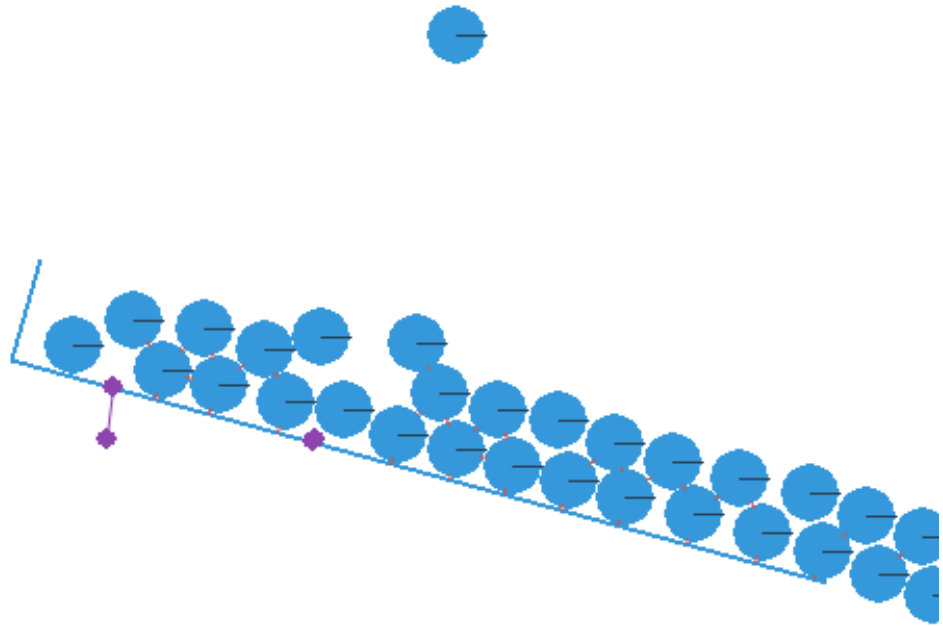
Helper function fill_space for the draw demos. Adds a lot of stuff to a space.

8.4.2.27 slide_and_pinjoint.py

Source: [examples/slide_and_pinjoint.py](#)

A L shape attached with a joint and constrained to not tip over.

This example is also used in the Get Started Tutorial.

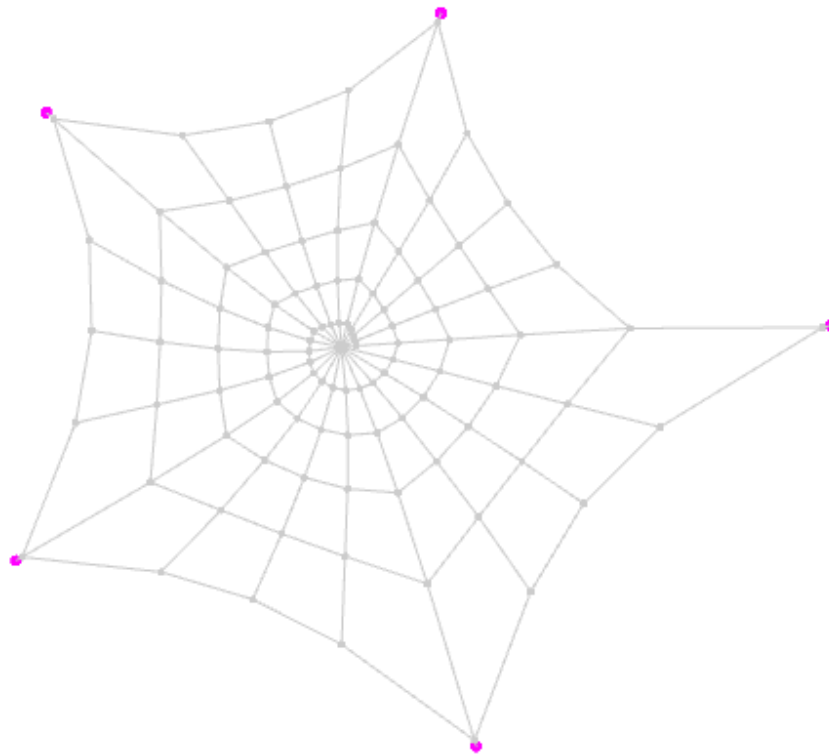


8.4.2.28 spiderweb.py

Source: [examples/spiderweb.py](#)

Showcase of a elastic spiderweb (drawing with pyglet)

It is possible to grab one of the crossings with the mouse



59.42

8.4.2.29 tangram.py

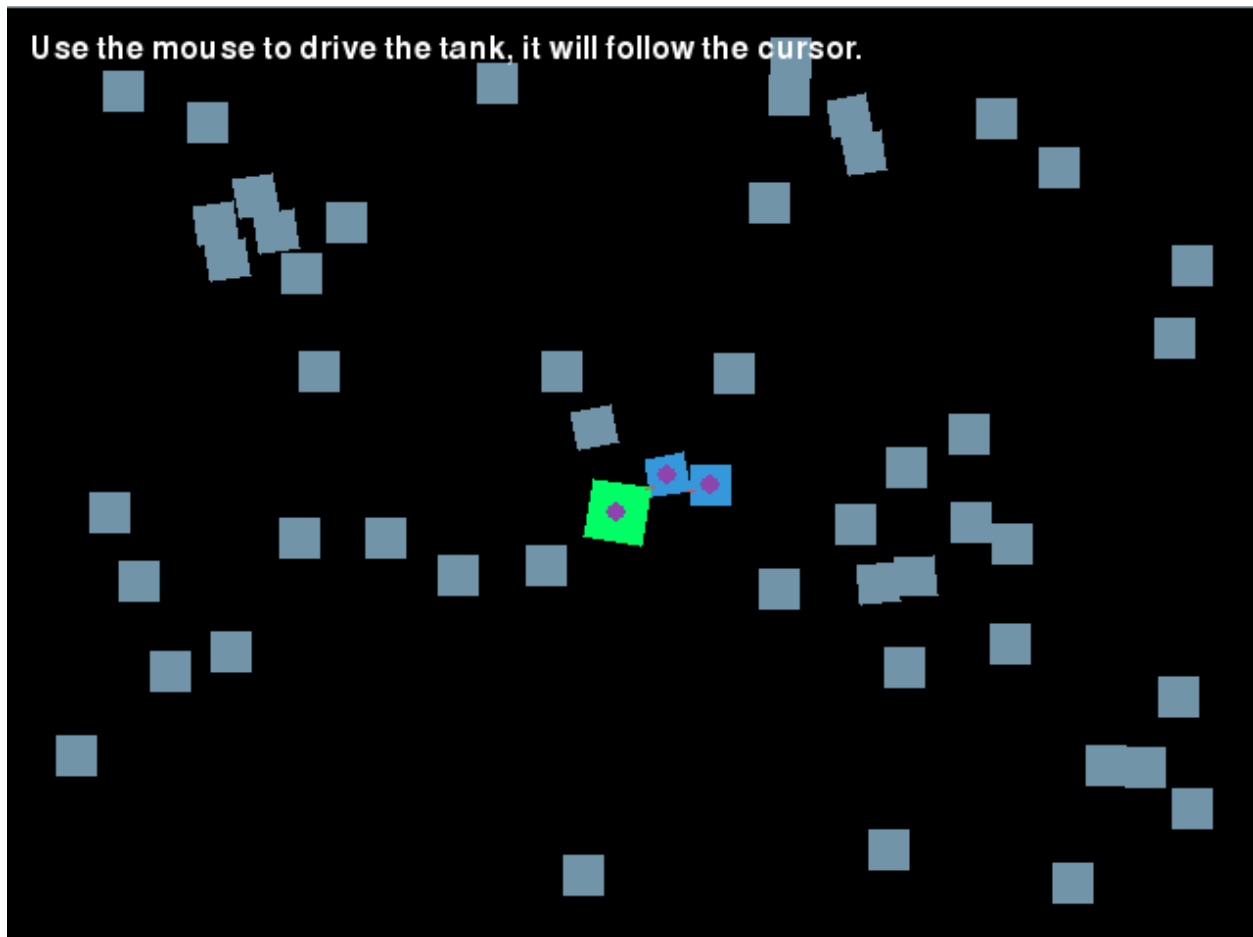
Source: [examples/tangram.py](#)

Remake of the pyramid demo from the box2d testbed.

8.4.2.30 tank.py

Source: [examples/tank.py](#)

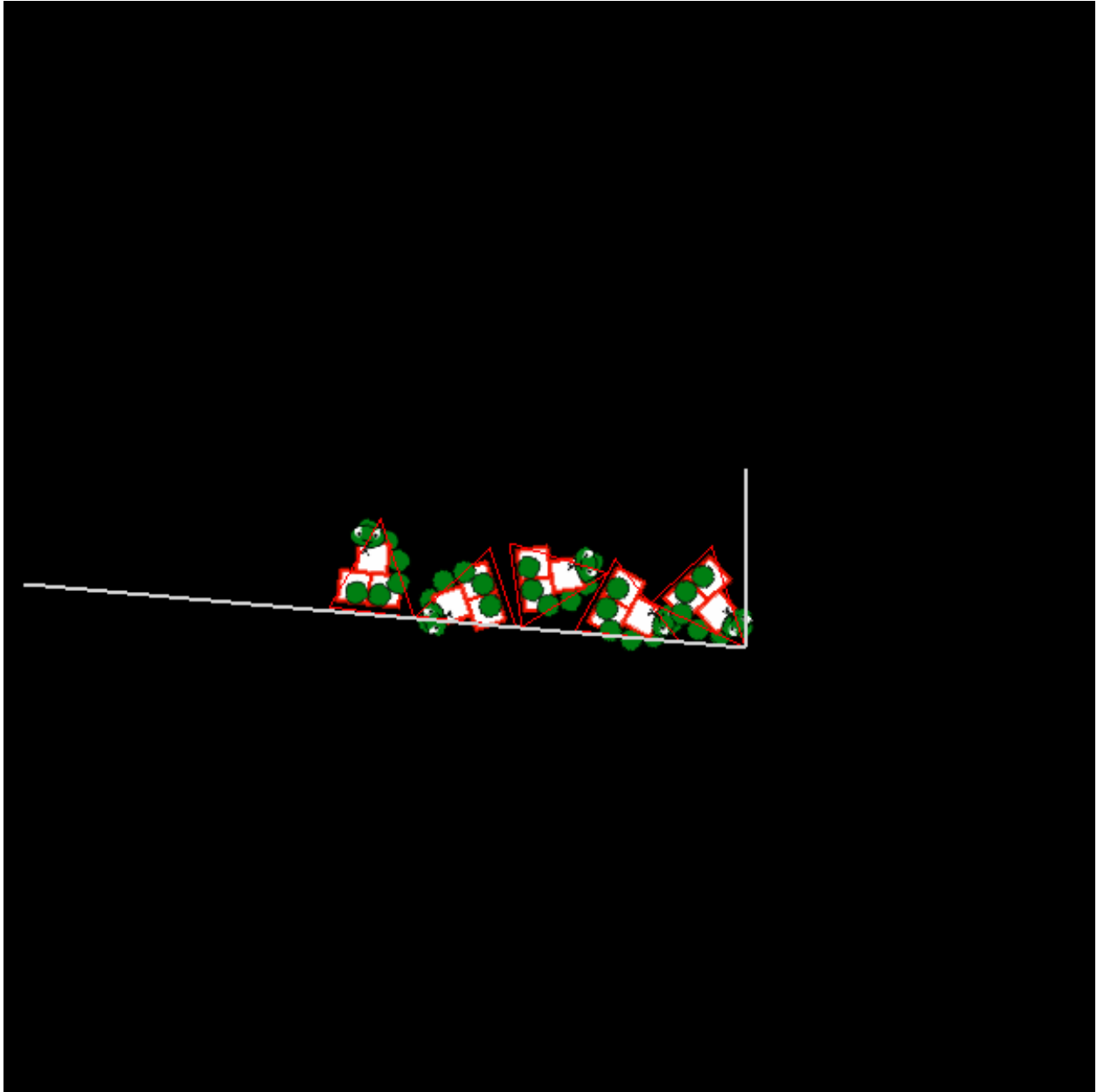
Port of the Chipmunk tank demo. Showcase a topdown tank driving towards the mouse, and hitting obstacles on the way.



8.4.2.31 using_sprites.py

Source: [examples/using_sprites.py](#)

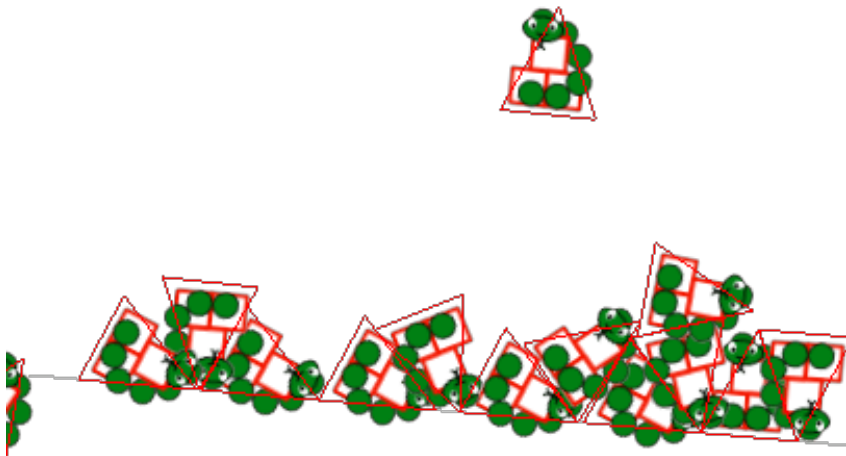
Very basic example of using a sprite image to draw a shape more similar how you would do it in a real game instead of the simple line drawings used by the other examples.



8.4.2.32 using_sprites_pyglet.py

Source: [examples/using_sprites_pyglet.py](#)

This example is a clone of the `using_sprites` example with the difference that it uses `pyglet` instead of `pygame` to showcase sprite drawing.



58.98

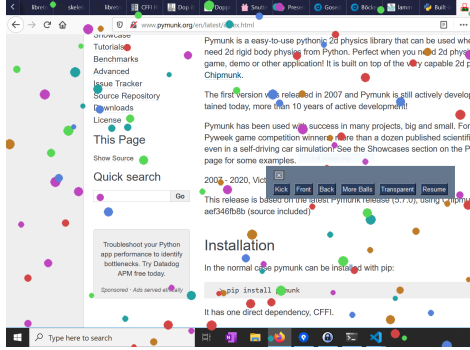

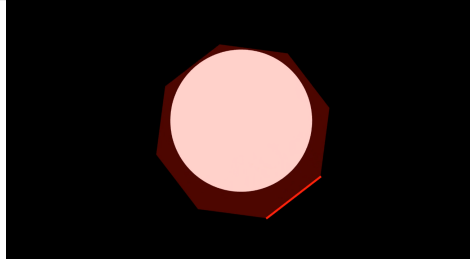

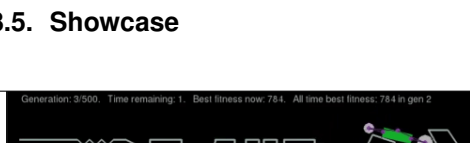
8.5 Showcase

This page shows some uses of Easymunk. If you also have done something using Easymunk please let me know and I can add it here!

8.5.1 Games

	<p>My Sincerest Apologies</p> <p>made by The Larry and Dan show (mauve, larry). Retrieved 2018-10-25</p> <p>Winner of PyWeek 24 (Overall Team Entry)</p> <p>A game of fun, shooting, and “I’m sorry to put you through this”. A fabricator robot on Mars was supposed to make a bunch of robots! But it got lazy and made robots that could make other robots. And it made them smarter than they should have been. Now they’ve all gone off and hidden away behind various tanks and computers. Happily, he knew how to construct <i>you</i>, a simple fighting robot. It’s your job to clean out each area!</p> <p>See Daniel Popes teardown here for additional details</p>
	<p>Beneath the Ice</p> <p>made by Team Chimera (mit-mit, Lucid Design Ar). Retrieved 2016-09-25</p> <p>Winner of PyWeek 22 (Overall Team Entry)</p> <p>Beneath the Ice is a submarine exploration game and puzzle solving adventure! Uncover a mysterious pariah who can’t let you discover his secrets, who can’t let you in! Team Chimera take 3!</p>
	

8.5.2 Non-Games

	<p>PySimpleGUI Desktop Demo</p> <p>made by PySimpleGUI/Mike. Retrieved 2020-10-13</p> <p>Demo of using PySimpleGUI together with Easymunk to create bouncing balls directly on the desktop, thanks to a transparent container window.</p>
	<p>Legged robot using differential evolution and perception</p> <p>made by Nav. Retrieved 2020-08-20</p> <p>Legged robot first using Differential Evolution to navigate terrain and then learning to recognise the world via perception from its senses.</p>
	<p>Simulation of ambient chimes Circle in a hexagon</p> <p>made by Jan Abraham. Retrieved 2019-11-17</p> <p>An ambient piano chord produced by the simulation of a bouncing ball. The calculations were carried out using pymunk library. Tuning: Kirnberger III</p>
	<p>I teach AI to move with using NEAT</p> <p>made by Cheesy AI. Retrieved 2019-11-17</p> <p>Recently I learned Easymunk 2d physics library. It is very cool so with that I made 2d Humanoid for my AI. Today I'm going to teach AI to move forward with NEAT. NEAT is a genetic algorithm for the</p>
<p>8.5. Showcase</p> 	<p>generation of evolving artificial neural networks. Results are quite weird but it will be fun. Have fun!</p> <p>61</p>

8.5.3 Papers / Science

Pymunk has been used or referenced in a number of scientific papers.

List of papers which has used or mentioned Pymunk:

- Mori, Hiroki, Masayuki Masuda, and Tetsuya Ogata. “Tactile-based curiosity maximizes tactile-rich object-oriented actions even without any extrinsic rewards.” In 2020 Joint IEEE 10th International Conference on Development and Learning and Epigenetic Robotics (ICDL-EpiRob), pp. 1-7. IEEE, 2020.
- Jiang, Lincheng. “A Computational Method to Generate One-story Floor Plans for Nursing Homes Based on Daylight Hour Potential and Shortest Path of Circulations.” (2020).
- Chen, Ricky TQ, Brandon Amos, and Maximilian Nickel. “Learning Neural Event Functions for Ordinary Differential Equations.” arXiv preprint arXiv:2011.03902 (2020).
- Jain, Ayush, Andrew Szot, and Joseph J. Lim. “Generalization to New Actions in Reinforcement Learning.” arXiv preprint arXiv:2011.01928 (2020).
- Petitgirard, Julien, Tony Piguet, Philippe Baucour, Didier Chamagne, Eric Fouillien, and Jean-Christophe Delmare. “Steady State and 2D Thermal Equivalence Circuit for Winding Heads—A New Modelling Approach.” *Mathematical and Computational Applications* 25, no. 4 (2020): 70.
- Hook, Joosep, Seif El-Sedky, Varuna De Silva, and Ahmet Kondo. “Learning Data-Driven Decision-Making Policies in Multi-Agent Environments for Autonomous Systems.” *Cognitive Systems Research* (2020).
- Matthews, Elizabeth A., and Juan E. Gilbert. “ATLAS CHRONICLE: DEVELOPMENT AND VERIFICATION OF A SYSTEM FOR PROCEDURAL GENERATION OF STORY-DRIVEN GAMES.”
- Ipe, Navin. “Context and event-based cognitive memory constructs for embodied intelligence machines.”
- Ipe, Navin. “An In-Memory Physics Environment as a World Model for Robot Motion Planning.” (2020).
- Li, Yunzhu, Antonio Torralba, Animashree Anandkumar, Dieter Fox, and Animesh Garg. “Causal Discovery in Physical Systems from Videos.” arXiv preprint arXiv:2007.00631 (2020).
- Suh, H. J., and Russ Tedrake. “The Surprising Effectiveness of Linear Models for Visual Foresight in Object Pile Manipulation.” arXiv preprint arXiv:2002.09093 (2020).
- Vos, Bastiaan. “The Sailing Tug: A feasibility study on the application of Wind-Assisted towing of the Thialf.” (2019).
- Wong, Eric C. “Example Based Hebbian Learning may be sufficient to support Human Intelligence.” *bioRxiv* (2019): 758375.
- Manoury, Alexandre, and Cédric Buche. “Hierarchical Affordance Discovery using Intrinsic Motivation.” 2019.
- Mounsif, Mehdi, Sebastien Lengagne, Benoit Thuilot, and Lounis Adouane. “Universal Notice Network: Transferable Knowledge Among Agents.” In 2019 6th International Conference on Control, Decision and Information Technologies (CoDIT), pp. 563-568. IEEE, 2019.
- Du, Yilun, and Karthik Narasimhan. “Task-Agnostic Dynamics Priors for Deep Reinforcement Learning.” In International Conference on Machine Learning, pp. 1696-1705. 2019.
- Siegel, Max Harmon. “Compositional simulation in perception and cognition.” PhD diss., Massachusetts Institute of Technology, 2018.
- Caselles-Dupré, Hugo, Louis Annabi, Oksana Hagen, Michael Garcia-Ortiz, and David Filliat. “Flatland: a Lightweight First-Person 2-D Environment for Reinforcement Learning.” arXiv preprint arXiv:1809.00510 (2018).
- Yingzhen, Li, and Stephan Mandt. “Disentangled Sequential Autoencoder.” In International Conference on Machine Learning, pp. 5656-5665. 2018.

- Melnik, Andrew. “Sensorimotor Processing in the Human Brain and in Cognitive Architectures.” (2018).
- Li, Yingzhen, and Stephan Mandt. “A Deep Generative Model for Disentangled Representations of Sequential Data.” arXiv preprint arXiv:1803.02991 (2018).
- Hongsuk Yi, Eunsoo Park and Seungil Kim (, , and .) “Deep Reinforcement Learning for Autonomous Vehicle Driving” (“ .”) 2017 Korea Software Engineering Conference ((2017): 784-786.)
- Fraccaro, Marco, Simon Kamronn, Ulrich Paquet, and Ole Winther. “A Disentangled Recognition and Nonlinear Dynamics Model for Unsupervised Learning.” arXiv preprint arXiv:1710.05741 (2017).
- Kister, Ulrike, Konstantin Klamka, Christian Tominski, and Raimund Dachsel. “GraSp: Combining Spatially-aware Mobile Devices and a Display Wall for Graph Visualization and Interaction.” In Computer Graphics Forum, vol. 36, no. 3, pp. 503-514. 2017.
- Kim, Neil H., Gloria Lee, Nicholas A. Sherer, K. Michael Martini, Nigel Goldenfeld, and Thomas E. Kuhlman. “Real-time transposable element activity in individual live cells.” *Proceedings of the National Academy of Sciences* 113, no. 26 (2016): 7278-7283.
- Baheti, Ashutosh, and Arobinda Gupta. “Non-linear barrier coverage using mobile wireless sensors.” In Computers and Communications (ISCC), 2017 IEEE Symposium on, pp. 804-809. IEEE, 2017.
- Espeso, David R., Esteban Martínez-García, Victor De Lorenzo, and Ángel Goñi-Moreno. “Physical forces shape group identity of swimming *Pseudomonas putida* cells.” *Frontiers in Microbiology* 7 (2016).
- Goni-Moreno, Angel, and Martyn Amos. “DiSCUS: A Simulation Platform for Conjugation Computing.” In International Conference on Unconventional Computation and Natural Computation, pp. 181-191. Springer International Publishing, 2015.
- Amos, Martyn, et al. “Bacterial computing with engineered populations.” *Phil. Trans. R. Soc. A* 373.2046 (2015): 20140218.
- Crane, Beth, and Stephen Sherratt. “rUNSWift 2D Simulator; Behavioural Simulation Integrated with the rUNSWift Architecture.” *UNSW School of Computer Science and Engineering* (2013).
- Miller, Chreston Allen. “Structural model discovery in temporal event data streams.” Diss. Virginia Polytechnic Institute and State University, 2013.
- Pumar García, César. “Simulación de evolución dirigida de bacteriófagos en poblaciones de bacterias en 2D.” (2013).
- Simoes, Manuel, and Caroline GL Cao. “Leonardo: a first step towards an interactive decision aid for port-placement in robotic surgery.” *Systems, Man, and Cybernetics (SMC), 2013 IEEE International Conference on*. IEEE, 2013.
- Goni-Moreno, Angel, and Martyn Amos. “Discrete modelling of bacterial conjugation dynamics.” *arXiv preprint arXiv:1211.1146* (2012).
- Matthews, Elizabeth A. “ATLAS CHRONICLE: A STORY-DRIVEN SYSTEM TO CREATE STORY-DRIVEN MAPS.” Diss. Clemson University, 2012.
- Matthews, Elizabeth, and Brian Malloy. “Procedural generation of story-driven maps.” *Computer Games (CGAMES), 2011 16th International Conference on*. IEEE, 2011.
- Miller, Chreston, and Francis Quek. “Toward multimodal situated analysis.” *Proceedings of the 13th international conference on multimodal interfaces*. ACM, 2011.
- Verdie, Yannick. “Surface gesture & object tracking on tabletop devices.” Diss. Virginia Polytechnic Institute and State University, 2010.
- Agrawal, Vivek, and Ryan Kerwin. “Dynamic Robot Path Planning Among Crowds in Emergency Situations.”

List last updated 2020-11-17. If something is missing or wrong, please contact me!

8.5.3.1 Cite Pymunk

If you use Pymunk in a published work and want to cite it, below is a bibtex example. Feel free to modify to fit your style. (Make sure to modify the version number if included.):

```
@misc{pymunk,  
  author = {Victor Blomqvist},  
  title = {Pymunk: A easy-to-use pythonic rigid body 2d physics library (version 6.0.  
→0)},  
  year = {2007},  
  url = {https://www.pymunk.org},  
}
```

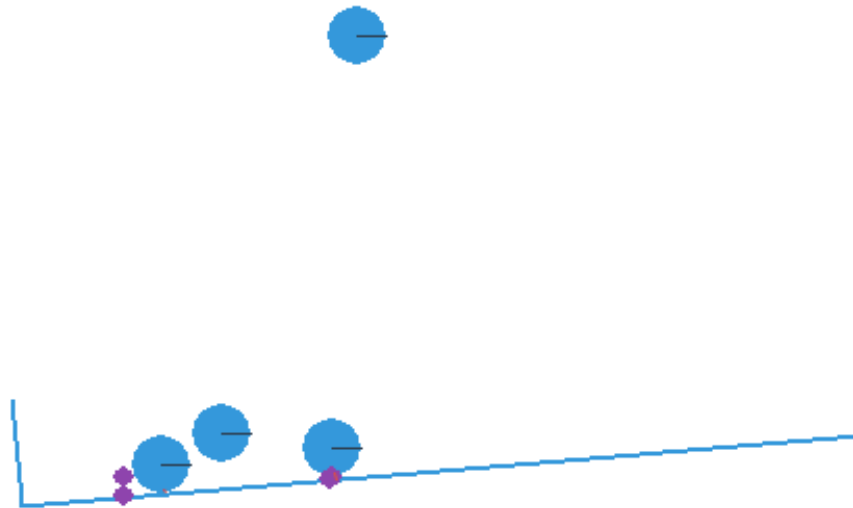
8.6 Tutorials

Easymunk has one tutorial that show a simple simulation from start to end.

After reading it make sure to also check out the [Examples](#) as most of them are easy to follow and showcase many of the things you can do with easymunk.

8.6.1 Slide and Pin Joint Demo Step by Step

This is a step by step tutorial explaining the demo `slide_and_pinjoint.py` included in easymunk. You will find a screenshot of it in the list of [examples](#). It is probably a good idea to have the file near by if I miss something in the tutorial or something is unclear.



8.6.1.1 Before we start

For this tutorial you will need:

- Python (of course)
- Pygame (found at www.pygame.org)
- Easymunk

Pygame is required for this tutorial and some of the included demos, but it is not required to run just easymunk. Easymunk should work just fine with other similar libraries as well, for example you could easily translate this tutorial to use Pyglet instead.

Easymunk is built on top of the 2d physics library Chipmunk. Chipmunk itself is written in C meaning Easymunk

need to call into the c code. The Cffi library helps with this, however if you are on a platform that I haven't been able to compile it on you might have to do it yourself. The good news is that it is very easy to do, in fact if you got Easymunk by Pip install its arelady done!

When you have easymunk installed, try to import it from the python prompt to make sure it works and can be imported:

```
>>> import easymunk
```

More information on installation can be found here: [Installation](#)

If it doesnt work or you have some kind of problem, feel free to write a post in the chipmunk forum, contact me directly or add your problem to the issue tracker: [Contact & Support](#)

8.6.1.2 An empty simulation

Ok, lets start. Chipmunk (and therefore Easymunk) has a couple of central concepts, which is explained pretty good in this citation from the Chipmunk docs:

Rigid bodies A rigid body holds the physical properties of an object. (mass, position, rotation, velocity, etc.) It does not have a shape by itself. If you've done physics with particles before, rigid bodies differ mostly in that they are able to rotate.

Collision shapes By attaching shapes to bodies, you can define the body's shape. You can attach many shapes to a single body to define a complex shape, or none if it doesn't require a shape.

Constraints/joints You can attach joints between two bodies to constrain their behavior.

Spaces Spaces are the basic simulation unit in Chipmunk. You add bodies, shapes and joints to a space, and then update the space as a whole.

The documentation for Chipmunk can be found here: <http://chipmunk-physics.net/release/ChipmunkLatest-Docs/> It is for the c-library but is a good complement to the Easymunk documentation as the concepts are the same, just that Easymunk is more pythonic to use.

The API documentation for Easymunk can be found here: [API Reference](#).

Anyway, we are now ready to write some code:

```
import sys
import pygame
import easymunk #1

def main():
    pygame.init()
    screen = pygame.display.set_mode((600, 600))
    pygame.display.set_caption("Joints. Just wait and the L will tip over")
    clock = pygame.time.Clock()

    space = easymunk.Space() #2
    space.gravity = (0.0, 900.0)

    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit(0)
            elif event.type == pygame.KEYDOWN and event.key == pygame.K_ESCAPE:
                sys.exit(0)

        screen.fill((255,255,255))
```

(continues on next page)

(continued from previous page)

```

        space.step(1/50.0) #3

        pygame.display.flip()
        clock.tick(50)

if __name__ == '__main__':
    sys.exit(main())

```

The code will display a blank window, and will run a physics simulation of an empty space.

1. We need to import easymunk in order to use it. . .
2. We then create a space and set its gravity to something good. Remember that what is important is what looks good on screen, not what the real world value is. 900 will make a good looking simulation, but feel free to experiment when you have the full code ready.
3. In our game loop we call the step() function on our space. The step function steps the simulation one step forward in time each time called.

Note: It is best to keep the step size constant and not adjust it depending on the framerate. The physic simulation will work much better with a constant step size.

8.6.1.3 Falling balls

The easiest shape to handle (and draw) is the circle. Therefore our next step is to make a ball spawn once in while. In many of the example demos all code is in one big pile in the main() function as they are so small and easy, but I will extract some methods in this tutorial to make it more easy to follow. First, a function to add a ball to a space:

```

def add_ball(space):
    mass = 3
    radius = 25
    body = easymunk.Body() # 1
    x = random.randint(120, 300)
    body.position = x, 50 # 2
    shape = easymunk.Circle(body, radius) # 3
    shape.mass = mass # 4
    shape.friction = 1
    space.add(body, shape) # 5
    return shape

```

1. We first create the body of the ball.
2. And we set its position
3. And in order for it to collide with things, it needs to have one (or many) collision shape(s).
4. All bodies must have their moment of inertia set. In most cases its easiest to let Easymunk handle calculation from shapes. So we set the mass of each shape, and then when added to space the body will automatically get a proper mass and moment set. Another option is to set the density of each shape, or its also possible to set the values directly on the body (or even adjust them afterwards).
5. To make the balls roll we set friction on the shape. (By default its 0).
6. Finally we add the body and shape to the space to include it in our simulation. Note that the body must always be added to the space before or at the same time as any shapes attached to it.

Now that we can create balls we want to display them. Either we can use the built in `pymunk_util` package to draw the whole space directly, or we can do it manually. The debug drawing functions included with Easymunk are good for putting something together easy and quickly, while for example a polished game most probably will want to make its own drawing code.

If we want to draw manually, our draw function could look something like this:

```
def draw_ball(screen, ball):
    p = int(ball.body.position.x), int(ball.body.position.y)
    pygame.draw.circle(screen, (0,0,255), p, int(ball.radius), 2)
```

And then called in this way (given we collected all the ball shapes in a list called `balls`):

```
for ball in balls:
    draw_ball(screen, ball)
```

However, as we use `pygame` in this example we can instead use the `debug_draw` method already included in Easymunk to simplify a bit. It first needs to be imported, and next we have to create a `DrawOptions` object with the options (what surface to draw on in the case of `Pygame`):

```
import easymunk.pygame_util
...
draw_options = easymunk.pygame_util.DrawOptions(screen)
```

And after that when we want to draw all our shapes we would just do it in this way:

```
space.debug_draw(draw_options)
```

Most of the examples included with Easymunk uses this way of drawing.

With the `add_ball` function and the `debug_draw` call and a little code to spawn balls you should see a couple of balls falling. Yay!

```
import sys, random
random.seed(1) # make the simulation the same each time, easier to debug
import pygame
import easymunk
import easymunk.pygame_util

#def add_ball(space):

def main():
    pygame.init()
    screen = pygame.display.set_mode((600, 600))
    pygame.display.set_caption("Joints. Just wait and the L will tip over")
    clock = pygame.time.Clock()

    space = easymunk.Space()
    space.gravity = (0.0, 900.0)

    balls = []
    draw_options = easymunk.pygame_util.DrawOptions(screen)

    ticks_to_next_ball = 10
    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
```

(continues on next page)

(continued from previous page)

```

        sys.exit(0)
    elif event.type == pygame.KEYDOWN and event.key == pygame.K_ESCAPE:
        sys.exit(0)

    ticks_to_next_ball -= 1
    if ticks_to_next_ball <= 0:
        ticks_to_next_ball = 25
        ball_shape = add_ball(space)
        balls.append(ball_shape)

    space.step(1/50.0)

    screen.fill((255,255,255))
    space.debug_draw(draw_options)

    pygame.display.flip()
    clock.tick(50)

if __name__ == '__main__':
    main()

```

8.6.1.4 A static L

Falling balls are quite boring. We don't see any physics simulation except basic gravity, and everyone can do gravity without help from a physics library. So lets add something the balls can land on, two static lines forming an L. As with the balls we start with a function to add an L to the space:

```

def add_static_L(space):
    body = easymunk.Body(body_type = easymunk.Body.STATIC) # 1
    body.position = (300, 300)
    l1 = easymunk.Segment(body, (-150, 0), (255, 0), 5) # 2
    l2 = easymunk.Segment(body, (-150, 0), (-150, -50), 5)
    l1.friction = 1 # 3
    l2.friction = 1

    space.add(body, l1, l2) # 4
    return l1,l2

```

1. We create a “static” body. The important step is to never add it to the space like the dynamic ball bodies. Note how static bodies are created by setting the `body_type` of the body. Many times its easier to use the already existing static body in the space (`space.static_body`), but we will make the L shape dynamic in just a little bit.
2. A line shaped shape is created here.
3. Set the friction.
4. Again, we only add the segments, not the body to the space.

Since we use `Space.debug_draw` to draw the space we dont need to do any special draw code for the Segments, but I still include a possible draw function here just to show what it could look like:

```

def draw_lines(screen, lines):
    for line in lines:
        body = line.body
        pv1 = body.position + line.a.rotated(body.angle) # 1
        pv2 = body.position + line.b.rotated(body.angle)

```

(continues on next page)

(continued from previous page)

```
p1 = to_pygame(pv1) # 2
p2 = to_pygame(pv2)
pygame.draw.lines(screen, THECOLORS["lightgray"], False, [p1,p2])
```

1. In order to get the position with the line rotation we use this calculation. line.a is the first endpoint of the line, line.b the second. At the moment the lines are static, and not rotated so we don't really have to do this extra calculation, but we will soon make them move and rotate.
2. This is a little function to convert coordinates from easymunk to pygame world. Now that we have it we can use it in the draw_ball() function as well.

```
def to_pygame(p):
    """Small helper to convert easymunk vec2d to pygame integers"""
    return round(p.x), round(p.y)
```

With the full code we should something like the below, and now we should see an inverted L shape in the middle will balls spawning and hitting the shape.

```
import sys, random
random.seed(1) # make the simulation the same each time, easier to debug
import pygame
import easymunk
import easymunk.pymunk_util

#def to_pygame(p):
#def add_ball(space):
#def add_static_l(space):

def main():
    pygame.init()
    screen = pygame.display.set_mode((600, 600))
    pygame.display.set_caption("Joints. Just wait and the L will tip over")
    clock = pygame.time.Clock()

    space = easymunk.Space()
    space.gravity = (0.0, 900.0)

    lines = add_static_L(space)
    balls = []
    draw_options = easymunk.pygame_util.DrawOptions(screen)

    ticks_to_next_ball = 10
    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit(0)
            elif event.type == pygame.KEYDOWN and event.key == pygame.K_ESCAPE:
                sys.exit(0)

        ticks_to_next_ball -= 1
        if ticks_to_next_ball <= 0:
            ticks_to_next_ball = 25
            ball_shape = add_ball(space)
            balls.append(ball_shape)

        space.step(1/50.0)
```

(continues on next page)

(continued from previous page)

```

        screen.fill((255,255,255))
        space.debug_draw(draw_options)

        pygame.display.flip()
        clock.tick(50)

if __name__ == '__main__':
    main()

```

8.6.1.5 Joints (1)

A static L shape is pretty boring. So lets make it a bit more exciting by adding two joints, one that it can rotate around, and one that prevents it from rotating too much. In this part we only add the rotation joint, and in the next we constrain it. As our static L shape won't be static anymore we also rename the function to add_L().

```

def add_L(space):
    rotation_center_body = easymunk.Body(body_type=easymunk.Body.STATIC) # 1
    rotation_center_body.position = (300, 300)

    body = easymunk.Body()
    body.position = (300, 300)
    l1 = easymunk.Segment(body, (-150, 0), (255.0, 0.0), 5.0)
    l2 = easymunk.Segment(body, (-150.0, 0), (-150.0, -50.0), 5.0)
    l1.friction = 1
    l2.friction = 1
    l1.mass = 8 # 2
    l2.mass = 1
    rotation_center_joint = easymunk.PinJoint(
        body, rotation_center_body, (0, 0), (0, 0)
    ) # 3

    space.add(l1, l2, body, rotation_center_joint)
    return l1, l2

```

1. This is the rotation center body. Its only purpose is to act as a static point in the joint so the line can rotate around it. As you see we never add any shapes to it.
2. The L shape will now be moving in the world, and therefor it can no longer be a static body. Here we see the benefit of setting the mass on the shapes instead of the body, no need to figure out how big the moment should be, and Easymunk will automatically calculate the center of gravity.
3. A pin joint allow two objects to pivot about a single point. In our case one of the objects will be stuck to the world.

8.6.1.6 Joints (2)

In the previous part we added a pin joint, and now its time to constrain the rotating L shape to create a more interesting simulation. In order to do this we modify the add_L() function:

```

def add_L(space):
    rotation_center_body = easymunk.Body(body_type = pymunk.Body.STATIC)
    rotation_center_body.position = (300,300)

    rotation_limit_body = pymunk.Body(body_type = pymunk.Body.STATIC) # 1

```

(continues on next page)

(continued from previous page)

```

rotation_limit_body.position = (200,300)

body = pymunk.Body()
body.position = (300,300)
l1 = pymunk.Segment(body, (-150, 0), (255.0, 0.0), 5.0)
l2 = pymunk.Segment(body, (-150.0, 0), (-150.0, -50.0), 5.0)
l1.friction = 1
l2.friction = 1
l1.mass = 8
l2.mass = 1

rotation_center_joint = pymunk.PinJoint(body, rotation_center_body, (0,0), (0,0))
joint_limit = 25
rotation_limit_joint = pymunk.SlideJoint(body, rotation_limit_body, (-100,0), (0,
↪0), 0, joint_limit) # 2

space.add(l1, l2, body, rotation_center_joint, rotation_limit_joint)
return l1,l2

```

1. We add a body..
2. Create a slide joint. It behaves like pin joints but have a minimum and maximum distance. The two bodies can slide between the min and max, and in our case one of the bodies is static meaning only the body attached with the shapes will move.

8.6.1.7 Ending

You might notice that we never delete balls. This will make the simulation require more and more memory and use more and more cpu, and this is of course not what we want. So in the final step we add some code to remove balls from the simulation when they are below the screen.

```

balls_to_remove = []
for ball in balls:
    if ball.body.position.y < 0: # 1
        balls_to_remove.append(ball) # 2

for ball in balls_to_remove:
    space.remove(ball, ball.body) # 3
    balls.remove(ball) # 4

```

1. Loop the balls and check if the body.position is less than 0.
2. If that is the case, we add it to our list of balls to remove.
3. To remove an object from the space, we need to remove its shape and its body.
4. And then we remove it from our list of balls.

And now, done! You should have an inverted L shape in the middle of the screen being filled with balls, tipping over releasing them, tipping back and start over. You can check `slide_and_pinjoint.py` included in `pymunk`, but it doesn't follow this tutorial exactly as I factored out a couple of blocks to functions to make it easier to follow in tutorial form.

If anything is unclear, not working feel free to raise an issue on github. If you have an idea for another tutorial you want to read, or some example code you want to see included in `pymunk`, please write it somewhere (like in the chipmunk forum)

The full code for this tutorial is:

```

import sys, random
random.seed(1) # make the simulation the same each time, easier to debug
import pygame
import pymunk
import pymunk.pygame_util

def add_ball(space):
    """Add a ball to the given space at a random position"""
    mass = 3
    radius = 25
    inertia = pymunk.moment_for_circle(mass, 0, radius, (0,0))
    body = pymunk.Body(mass, inertia)
    x = random.randint(120,300)
    body.position = x, 50
    shape = pymunk.Circle(body, radius, (0,0))
    shape.friction = 1
    space.add(body, shape)
    return shape

def add_L(space):
    """Add a inverted L shape with two joints"""
    rotation_center_body = pymunk.Body(body_type = pymunk.Body.STATIC)
    rotation_center_body.position = (300,300)

    rotation_limit_body = pymunk.Body(body_type = pymunk.Body.STATIC)
    rotation_limit_body.position = (200,300)

    body = pymunk.Body(10, 10000)
    body.position = (300,300)
    l1 = pymunk.Segment(body, (-150, 0), (255.0, 0.0), 5.0)
    l2 = pymunk.Segment(body, (-150.0, 0), (-150.0, -50.0), 5.0)
    l1.friction = 1
    l2.friction = 1
    l1.mass = 8
    l2.mass = 1

    rotation_center_joint = pymunk.PinJoint(body, rotation_center_body, (0,0), (0,0))
    joint_limit = 25
    rotation_limit_joint = pymunk.SlideJoint(body, rotation_limit_body, (-100,0), (0,
→0), 0, joint_limit)

    space.add(l1, l2, body, rotation_center_joint, rotation_limit_joint)
    return l1,l2

def main():
    pygame.init()
    screen = pygame.display.set_mode((600, 600))
    pygame.display.set_caption("Joints. Just wait and the L will tip over")
    clock = pygame.time.Clock()

    space = pymunk.Space()
    space.gravity = (0.0, 900.0)

    lines = add_L(space)
    balls = []
    draw_options = pymunk.pygame_util.DrawOptions(screen)

```

(continues on next page)

(continued from previous page)

```
ticks_to_next_ball = 10
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit(0)
        elif event.type == pygame.KEYDOWN and event.key == pygame.K_ESCAPE:
            sys.exit(0)

    ticks_to_next_ball -= 1
    if ticks_to_next_ball <= 0:
        ticks_to_next_ball = 25
        ball_shape = add_ball(space)
        balls.append(ball_shape)

    screen.fill((255,255,255))

    balls_to_remove = []
    for ball in balls:
        if ball.body.position.y > 550:
            balls_to_remove.append(ball)

    for ball in balls_to_remove:
        space.remove(ball, ball.body)
        balls.remove(ball)

    space.debug_draw(draw_options)

    space.step(1/50.0)

    pygame.display.flip()
    clock.tick(50)

if __name__ == '__main__':
    main()
```

8.6.2 External Tutorials

If you have made a tutorial that is using Easymunk in any way and want it mentioned here please send me a link and I will happily add it. I also accept full tutorials to include directly here if you prefer, as long as they are of reasonable quality and style. Check the source to see how the existing ones are built.

8.7 Benchmarks

To get a grip of the actual performance of Easymunk this page contains a number of benchmarks.

The full code of all benchmarks are available under the [benchmarks](#) folder.

Note that the the benchmarks are not yet updated for Easymunk 6.0, but tests look promising.

8.7.1 Micro benchmarks

In order to measure the overhead created by Easymunk in the most common cases I have created two micro benchmarks. They should show the speed of the actual wrapping code, which can tell how big overhead Easymunk creates, and how big difference different wrapping methods does.

The most common thing a typical program using Easymunk does is to read out the position and angle from a Easymunk object. Usually this is done each frame for every object in the simulation, so this is a important factor in how fast something will be.

Given this our first test is:

```
t += b.position.x + b.position.y + b.angle
```

(see *pymunk-get.py*)

Running it is simple, for example like this for pymunk 4.0:

```
> python -m pip install pymunk==4.0
> python pymunk-get.py
```

The second test we do is based on the second heavy thing we can do, and that is using a callback, for example as a collision handler or a position function:

```
def f(b, dt):
    b.position += (1, 0)

s.step(0.01)
```

(see *pymunk-callback.py*)

8.7.1.1 Results:

Tests run on a HP G1 1040 laptop with a Intel i7-4600U. Laptop runs Windows, and the tests were run inside a VirtualBox VM running 64bit Debian. The CPython tests uses CPython from Conda, while the Pypy tests used a manually downloaded Pypy. CPython 2.7 is using Cffi 1.7, the other tests Cffi 1.8.

Remember that these results doesn't tell you how you game/application will perform, they can more be seen as a help to identify performance issues and know differences between Pythons.

Pymunk-Get:

	CPython 2.7.12	CPython 3.5.2	Pypy 5.4.1
Pymunk 5.1	2.1s	2.2s	0.36s
Pymunk 5.0	4.3s	4.5s	0.37s
Pymunk 4.0	1.0s	0.9s	0.52s

Pymunk-Callback:

	CPython 2.7.12	CPython 3.5.2	Pypy 5.4.1
Pymunk 5.1	5.7s	6.8s	1.1s
Pymunk 5.0	6.5s	7.3s	1.0s
Pymunk 4.0	5.1s	6.5s	4.5s

What we can see from these results is that you should use Pypy if you have the possibility since that is much faster than regular CPython. We can also see that moving from Ctypes to Cffi between Pymunk 4 and 5 had a negative impact in CPython, but positive impact on Pypy, and Pymunk 5 together with Pypy is with a big margin the fastest option.

The speed increase between 5.0 and 5.1 happened because the Vec2d class and how its handled internally in Easymunk was changed to improve performance.

8.7.2 Compared to Other Physics Libraries

8.7.2.1 Cymunk

Cymunk is an alternative wrapper around Chipmunk. In contrast to Pymunk it uses Cython for wrapping (Pymunk uses CFFI) which gives it a different performance profile. However, since both are built around Chipmunk the overall speed will be very similar, only when information passes from/to Chipmunk will there be a difference. This is exactly the kind of overhead that the micro benchmarks are made to measure.

Cymunk is not as feature complete as Easymunk, so in order to compare with Easymunk we have to make some adjustments. A major difference is that it does not implement the *position_func* function, so instead we do an alternative callback test using the collision handler:

```
h = s.add_default_collision_handler()
def f(arb):
    return false
h.pre_solve = f

s.step(0.01)
```

(see *pymunk-collision-callback.py* and *cymunk-collision-callback.py*)

Results

Tests run on a HP G1 1040 laptop with a Intel i7-4600U. Laptop runs Windows, and the tests were run inside a VirtualBox VM running 64bit Debian. The CPython tests uses CPython from Conda, while the Pypy tests used a manually downloaded Pypy. Cffi version 1.10.0 and Cython 0.25.2.

Since Cymunk doesnt have a proper release I used the latest master from its Github repository, hash 24845cc retrieved on 2017-09-16.

Get:

	CPython 3.5.3	Pypy 5.8
Pymunk 5.3	2.14s	0.33s
Cymunk 20170916	0.41s	(10.0s)

Collision-Callback:

	CPython 3.5.3	Pypy 5.8
Pymunk 5.3	3.71s	0.58s
Pymunk 20170916	0.95s	(7.01s)

(Cymunk results on Pypy within parentheses since Cython is well known to be slow on Pypy)

What we can see from these results is that Cymunk on CPython is much faster than Easymunk on CPython, but Pymunk takes the overall victory when we include Pypy.

Something we did not take into account is that you can trade convenience for performance and use Cython in the application code as well to speed things up. I think this is the approach used in KivEnt which is the primary user of Cymunk. However, that requires a much more complicated setup when you develop your application because of the compiler requirements and code changes.

8.8 Advanced

In this section different “Advanced” topics are covered, things you normally dont need to worry about when you use Easymunk but might be of interest if you want a better understanding of Easymunk for example to extend it.

First off, Easymunk is a pythonic wrapper around the C-library Chipmunk.

To wrap Chipmunk Easymunk uses CFFI in API mode. On top of the CFFI wrapping is a handmade pythonic layer to make it nice to use from Python code.

8.8.1 Why CFFI?

This is a straight copy from the github issue tracking the CFFI upgrade. <https://github.com/viblo/pymunk/issues/99>

CFFI have a number of advantages but also a downsides.

Advantages (compared to ctypes):

- Its an active project. The developers and users are active, there are new releases being made and its possible to ask and get answers within a day on the CFFI mailing list.
- Its said to be the way forward for Pypy, with promise of better performance compares to ctypes.
- A little easier than ctypes to wrap things since you can just copy-paste the c headers.

Disadvantages (compared to ctypes):

- ctypes is part of the CPython standard library, CFFI is not. That means that it will be more difficult to install Easymunk if it uses CFFI, since a copy-paste install is no longer possible in an easy way.

For me I see the 1st advantage as the main point. I have had great difficulties with strange segfaults with 64bit python on windows, and also sometimes on 32bit python, and support for 64bit python on both windows and linux is something I really want. Hopefully those problems will be easier to handle with CFFI since it has an active community.

Then comes the 3rd advantage, that its a bit easier to wrap the c code. For ctypes I have a automatic wrapping script that does most of the low level wrapping, but its not supported, very difficult to set up (I only managed inside a VM with linux) and quite annoying. CFFI would be a clear improvement.

For the disadvantage of ctypes I think it will be acceptable, even if not ideal. Many python packages have to be installed in some way (like pygame), and nowadays with pip its very easy to do. So I hope that it will be ok.

8.8.2 Code Layout

Most of Easymunk should be quite straight forward.

Except for the documented API Easymunk has a couple of interesting parts. Low level bindings to Chipmunk, a custom documentation generation extension and a customized setup.py file to allow compilation of Chipmunk.

The low level chipmunk bindings are located in the file extension_build.py.

docs/src/ext/autoexample.py A Sphinx extension that scans a directory and extracts the toplevel docstring. Used to autogenerate the examples documentation.

easymunk/_chipmunk_cffi.py This file only contains a call to _chipmunk_cffi_abi.py, and exists mostly as a wrapper to be able to switch between abi and api mode of Cffi. This is currently not in use in the relased code, but is used during experimentation.

easymunk/_chipmkunk_cffi_abi.py This file contains the pure Cffi wrapping definitons. Bascially a giant string created by copy-paster from the relevant header files of Chipmunk.

setup.py Except for the standard setup stuff this file also contain the custom build commands to build Chipmunk from source, using a build_ext extension.

easymunk/tests/* Collection of (unit) tests. Does not cover all cases, but most core things are there. The tests require a working chipmunk library file.

tools/* Collection of helper scripts that can be used to various development tasks such as generating documentation.

8.8.3 Tests

There are a number of unit tests included in the easymunk.tests package (easymunk/tests). Not exactly all the code is tested, but most of it (at the time of writing its about 85% of the core parts).

The tests can be run by calling the module

```
> python -m pymunk.tests
```

Its possible to control which tests to run, by specifying a filtering argument. The matching is as broad as possible, so *Test* matches all the unit tests, *test_arbiter* all tests in *test_arbiter.py* and *testResetitution* matches the exact *testRestitution* test case

```
> python -m pymunk.tests -f testRestitution
```

To see all options to the tests command use -h

```
> python -m pymunk.tests -h
```

Since the tests cover even the optional parts, you either have to make sure all the optional dependencies are installed, or filter out those tests.

8.8.4 Working with non-wrapped parts of Chipmunk

In case you need to use something that exist in Chipmunk but currently is not included in easymunk the easiest method is to add it manually.

For example, lets assume that the `is_sleeping` property of a body was not wrapped by easymunk. The Chipmunk method to get this property is named `cpBodyIsSleeping`.

First we need to check if its included in the `cdef` definition in `extension_build.py`. If its not just add it.

```
cpBool cpBodyIsSleeping(const cpBody *body);
```

Then to make it easy to use we want to create a python method that looks nice:

```
def is_sleeping(body) :
    return cp.cpBodyIsSleeping(body._body)
```

Now we are ready with the mapping and ready to use our new method.

8.8.5 Weak References and free Methods

Internally Easymunk allocates structs from Chipmunk (the c library). For example a `Body` struct is created from inside the constructor method when a `easymunk.Body` is created. Because of this its important that the corresponding c side memory is deallocated properly when not needed anymore, usually when the Python side object is garbage collected. Most Easymunk objects use `ffi.gc` with a custom free function to do this. Note that the order of freeing is very important to avoid errors.

8.9 Changelog

8.9.1 Easymunk 0.9.0 (2021-03-01)

Forked from easymunk 6.0.0

This is the first release of Easymunk.

Highlights - Major changes relative to Pymunk:

- Default to using angles rather than radians.
- Avoid using `space.add/remove` methods by adding factory functions in `space`.
- Create the `Junction` object to control multiple constraints between two objects.

8.10 License

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

c

`easymunk.core`, [24](#)

e

`easymunk`, [30](#)

g

`easymunk.geometry`, [24](#)

l

`easymunk.linalg`, [24](#)

m

`easymunk.matplotlib`, [24](#)

p

`easymunk.pygame`, [26](#)

`easymunk.pyglet`, [28](#)

`easymunk.pyxel`, [30](#)

Symbols

`__init__()` (*easymunk.matplotlib.DrawOptions* method), 24

A

`ax()` (*easymunk.matplotlib.DrawOptions* property), 24

C

`chipmunk_version` (in module *easymunk*), 30

`collision_point_color()` (*easy-munk.matplotlib.DrawOptions* property), 25

`collision_point_color()` (*easy-munk.pygame.DrawOptions* property), 27

`collision_point_color()` (*easy-munk.pyglet.DrawOptions* property), 29

`color_for_shape()` (*easy-munk.matplotlib.DrawOptions* method), 25

`color_for_shape()` (*easy-munk.pygame.DrawOptions* method), 27

`color_for_shape()` (*easy-munk.pyglet.DrawOptions* method), 29

`constraint_color()` (*easy-munk.matplotlib.DrawOptions* property), 25

`constraint_color()` (*easy-munk.pygame.DrawOptions* property), 27

`constraint_color()` (*easy-munk.pyglet.DrawOptions* property), 29

D

`draw_bb()` (*easymunk.matplotlib.DrawOptions* method), 25

`draw_bb()` (*easymunk.pygame.DrawOptions* method), 27

`draw_bb()` (*easymunk.pyglet.DrawOptions* method), 29

`draw_circle()` (*easymunk.matplotlib.DrawOptions* method), 24

`draw_circle()` (*easymunk.pygame.DrawOptions* method), 26

`draw_circle()` (*easymunk.pyglet.DrawOptions* method), 28

`draw_circle_shape()` (*easy-munk.matplotlib.DrawOptions* method), 25

`draw_circle_shape()` (*easy-munk.pygame.DrawOptions* method), 27

`draw_circle_shape()` (*easy-munk.pyglet.DrawOptions* method), 29

`DRAW_COLLISION_POINTS` (*easy-munk.matplotlib.DrawOptions* attribute), 25

`DRAW_COLLISION_POINTS` (*easy-munk.pygame.DrawOptions* attribute), 27

`DRAW_COLLISION_POINTS` (*easy-munk.pyglet.DrawOptions* attribute), 29

`DRAW_CONSTRAINTS` (*easy-munk.matplotlib.DrawOptions* attribute), 25

`DRAW_CONSTRAINTS` (*easy-munk.pygame.DrawOptions* attribute), 27

`DRAW_CONSTRAINTS` (*easymunk.pyglet.DrawOptions* attribute), 29

`draw_dot()` (*easymunk.matplotlib.DrawOptions* method), 24

`draw_dot()` (*easymunk.pygame.DrawOptions* method), 26

`draw_dot()` (*easymunk.pyglet.DrawOptions* method), 29

`draw_fat_segment()` (*easy-munk.matplotlib.DrawOptions* method), 24

`draw_fat_segment()` (*easy-munk.pygame.DrawOptions* method), 26

`draw_fat_segment()` (*easy-munk.pyglet.DrawOptions* method), 29

`draw_object()` (*easymunk.matplotlib.DrawOptions* method), 25

`draw_object()` (*easymunk.pygame.DrawOptions* method), 27

`draw_object()` (*easymunk.pyglet.DrawOptions* method), 29

`draw_poly_shape()` (*easymunk.matplotlib.DrawOptions* method), 25
`draw_poly_shape()` (*easymunk.pygame.DrawOptions* method), 27
`draw_poly_shape()` (*easymunk.pyglet.DrawOptions* method), 29
`draw_polygon()` (*easymunk.matplotlib.DrawOptions* method), 24
`draw_polygon()` (*easymunk.pygame.DrawOptions* method), 26
`draw_polygon()` (*easymunk.pyglet.DrawOptions* method), 29
`draw_segment()` (*easymunk.matplotlib.DrawOptions* method), 24
`draw_segment()` (*easymunk.pygame.DrawOptions* method), 26
`draw_segment()` (*easymunk.pyglet.DrawOptions* method), 28
`draw_segment_shape()` (*easymunk.matplotlib.DrawOptions* method), 25
`draw_segment_shape()` (*easymunk.pygame.DrawOptions* method), 27
`draw_segment_shape()` (*easymunk.pyglet.DrawOptions* method), 30
`draw_shape()` (*easymunk.matplotlib.DrawOptions* method), 25
`draw_shape()` (*easymunk.pygame.DrawOptions* method), 27
`draw_shape()` (*easymunk.pyglet.DrawOptions* method), 30
`DRAW_SHAPES` (*easymunk.matplotlib.DrawOptions* attribute), 25
`DRAW_SHAPES` (*easymunk.pygame.DrawOptions* attribute), 27
`DRAW_SHAPES` (*easymunk.pyglet.DrawOptions* attribute), 29
`draw_vec2d()` (*easymunk.matplotlib.DrawOptions* method), 26
`draw_vec2d()` (*easymunk.pygame.DrawOptions* method), 27
`draw_vec2d()` (*easymunk.pyglet.DrawOptions* method), 30
`DrawOptions` (class in *easymunk.matplotlib*), 24
`DrawOptions` (class in *easymunk.pygame*), 26
`DrawOptions` (class in *easymunk.pyglet*), 28

E

`easymunk`
 module, 30
`easymunk.core`
 module, 24
`easymunk.geometry`

 module, 24
`easymunk.linalg`
 module, 24
`easymunk.matplotlib`
 module, 24
`easymunk.pygame`
 module, 26
`easymunk.pyglet`
 module, 28
`easymunk.pxel`
 module, 30

F

`finalize_frame()` (*easymunk.matplotlib.DrawOptions* method), 25
`finalize_frame()` (*easymunk.pygame.DrawOptions* method), 28
`finalize_frame()` (*easymunk.pyglet.DrawOptions* method), 30
`flags()` (*easymunk.matplotlib.DrawOptions* property), 26
`flags()` (*easymunk.pygame.DrawOptions* property), 28
`flags()` (*easymunk.pyglet.DrawOptions* property), 30
`from_pygame()` (*easymunk.pygame.DrawOptions* method), 28

M

`module`
 `easymunk`, 30
 `easymunk.core`, 24
 `easymunk.geometry`, 24
 `easymunk.linalg`, 24
 `easymunk.matplotlib`, 24
 `easymunk.pygame`, 26
 `easymunk.pyglet`, 28
 `easymunk.pxel`, 30
`mouse_pos()` (*easymunk.pygame.DrawOptions* method), 28

S

`shape_dynamic_color` (*easymunk.matplotlib.DrawOptions* attribute), 26
`shape_dynamic_color` (*easymunk.pygame.DrawOptions* attribute), 28
`shape_dynamic_color` (*easymunk.pyglet.DrawOptions* attribute), 30
`shape_kinematic_color` (*easymunk.matplotlib.DrawOptions* attribute), 26
`shape_kinematic_color` (*easymunk.pygame.DrawOptions* attribute), 28

`shape_kinematic_color` (*easy-*
munk.pyglet.DrawOptions attribute), [30](#)

`shape_outline_color()` (*easy-*
munk.matplotlib.DrawOptions *property*),
[26](#)

`shape_outline_color()` (*easy-*
munk.pygame.DrawOptions property), [28](#)

`shape_outline_color()` (*easy-*
munk.pyglet.DrawOptions property), [30](#)

`shape_sleeping_color` (*easy-*
munk.matplotlib.DrawOptions *attribute*),
[26](#)

`shape_sleeping_color` (*easy-*
munk.pygame.DrawOptions attribute), [28](#)

`shape_sleeping_color` (*easy-*
munk.pyglet.DrawOptions attribute), [30](#)

`shape_static_color` (*easy-*
munk.matplotlib.DrawOptions *attribute*),
[26](#)

`shape_static_color` (*easy-*
munk.pygame.DrawOptions attribute), [28](#)

`shape_static_color` (*easy-*
munk.pyglet.DrawOptions attribute), [30](#)

`surface` (*easymunk.pygame.DrawOptions* *attribute*),
[26](#)

T

`to_pygame()` (*easymunk.pygame.DrawOptions*
method), [28](#)