pymunk Documentation

Release 4.0.0

Victor Blomqvist

October 04, 2013

CONTENTS

pymunk 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 nice 2d physics library Chipmunk.

CHAPTER

ONE

GETTING STARTED

To get started quickly take a look in the *Readme*, it contains a summary of the most important things and is quick to read. When done its a good idea to take a look at the included *Examples*, read the *Tutorials* and take a look in the *API Reference*.

CHAPTER

TWO

THE PYMUNK 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 stuff should be needed to add physics to your game/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, OO, 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.

CHAPTER

THREE

PYTHON 2 & PYTHON 3

pymunk has been tested and runs fine on both Python 2 and Python 3. It has been tested on recent versions of CPython (2 and 3) and Pypy. For an exact list of tested versions see the *Readme*.

CHAPTER

FOUR

CONTACT & SUPPORT

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

Forum Currently pymunk has no separate forum, but uses the general Chipmunk forum at http://chipmunk-physics.net/forum/index.php Many issues are the same, like how to create a rag doll or why a fast moving object pass through a wall. If you have a pymunk specific question feel free to mark your post with [pymunk] to make it stand out a bit.

E-Mail You can email me directly at vb@viblo.se

Issue Tracker Please use the issue tracker at google code to report any issues you find: http://code.google.com/p/pymunk/

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 or the forum other people might help as well!)

Attention: If you like pymunk and find it useful and have a few minutes to spare I would love to hear from you. What would *really* be nice is a real postcard from the place where you are! :)
pymunk's address is:

pymunk c/o Victor Blomqvist
Carl Thunbergs Vag 9
169 69 Solna
SWEDEN

CHAPTER

FIVE

CONTENTS

5.1 Readme

5.1.1 About

pymunk is a easy-to-use pythonic 2d physics library that can be used whenever you need 2d rigid body physics from Python. It is built on top of the very nice 2d physics library Chipmunk.

2007 - 2013, Victor Blomqvist - vb@viblo.se, MIT License

This release is based on the latest pymunk release (4.0.0), using chipmunk 6.2 rev 3bdf1b7b3c (source included)

Homepage http://www.pymunk.org

Forum http://chipmunk-physics.net/forum/

Email vb@viblo.se

Getting the latest SVN copy svn checkout http://pymunk.googlecode.com/svn/trunk pymunk-read-only

Chipmunk http://chipmunk-physics.net/

5.1.2 How to Use

pymunk ships with a number of demos in the examples directory, and its complete documentation including API Reference.

If chipmunk doesnt ship with a chipmunk binary your platform can understand (currently Windows and Linux 32bit and 64 bit are included) you will have to compile chipmunk before install. See section CHIPMUNK in this readme for (very simple) instructions.

To install you can either run:

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

or simply put the pymunk folder where your program/game can find it. (like /my_python_scripts/yourgame/pymunk). The chipmunk binary library is located in the pymunk folder.

The easy way to get started is to check out the examples/ directory, and run 'run.py python arrows.py' and so on, and see what each one does :)

5.1.3 Example

Quick code example:

```
import pymunk                # Import pymunk..

space = pymunk.Space()       # Create a Space which contain the simulation
space.gravity = 0,-1000      # Set its gravity

body = pymunk.Body(1,1666)   # Create a Body with mass and moment
body.position = 50,100       # Set the position of the body

poly = pymunk.Poly.create_box(body) # Create a box shape and attach to body
space.add(body, poly)        # Add both body and shape to the simulation

while True:                  # Infinite loop simulation
    space.step(0.02)         # Step the simulation one step forward
```

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

5.1.4 Dependencies / Requirements

- python (tested on cpython 2.6, 2.7 and 3.2, 3.3. Also on pypy 2.1)
- chipmunk (pymunk ships with a set of chipmunk libraries)
- pygame (optional, you need it to run most of the demos)
- pyglet (optional, you need it to run the moon buggy demo)
- sphinx (optional, you need it to build documentation)
- ctypeslib & GCC_XML (optional, you need them to generate new bindings)

5.1.5 Chipmunk

Compiled libraries of Chipmunk compatible Windows and Linux 32bit and 64bit are distributed with pymunk. If pymunk doesnt have your particular platform included, you can compile Chipmunk by hand with a custom setup argument:

```
> python setup.py build_chipmunk
```

The compiled file goes into the /pymunk folder (same as _chipmunk.py, util.py and others). If the compile fail, please check the readme in chipmunk_src for generic instructions on how to compile with gcc, or download the relevant release from Chipmunk homepage and follow its instructions.

5.2 News

5.2.1 pymunk 4.0.0

Victor - 2013-08-25

A new release of pymunk is here!

This release is definatley a milestone, pymunk is now over 5 years old! (first version was released in February 2008, for the pyweek competition)

In this release a number of improvements have been made to pymunk. It includes debug drawing for pyglet (debug draw for pygame was introduced in pymunk 3), an updated Chipmunk version with the resulting API adjustments, more and better examples and overall polish as usual.

With the new Chipmunk version (6.2 beta), collision detection might behave a little bit differently as it uses a different algorithm compared to earlier versions. The new algorithm means that segments to segment collisions will be detected now. If you have some segments that you dont want to collide then you can use the sensor property, or a custom collision callback function.

To see the new pymunk.pyglet_util module in action check out the pyglet_util_demo.py example. It has an interface similar to the pygame_util, with a couple of changes because of differences between pyglet and pygame.

Some API additions and changes have been made. Its now legal to add and remove objects such as bodies and shapes during the simulation step (for example in a callback). The actual removal will be scheduled to occur as soon as the simulation step is complete. Other changes are the possibility to change body of a shape, to get the BB of a shape, and create a shape with empty body. On a body you can now retrieve the shapes and constraints attached to it.

This release has been tested and runs on CPython 2.5, 2.6, 2.7, 3.3 and Pypy 2.1. At least one run of the unit tests have been made on the following platforms: 32 bit CPython on Windows, 32 and 64 bit CPython on Linux, and 64 bit CPython on OSX. Pypy 2.1 on one of the above platforms.

Changes

- New draw module to help with pyglet prototyping
- Updated Chipmunk verison, with new collision detected code.
- Added, improved and fixed broken examples
- Possible to switch bodies on shapes
- Made it legal do add and remove bodies during a simulation step
- Added shapes and constraints properties to Body
- Possible to get BB of a Shape, and they now allow empty body in constructor
- Added radius property to Poly shapes
- Renamed Poly.get_points to get_vertices
- Renamed the Segment.a and Segment.b properties to unsafe_set
- Added example of using pyinstaller
- Fixed a number of bugs reported
- Improved docs in various places
- General polish and cleanup

I hope you will enjoy this new release!

5.2.2 pymunk 3.0.0

Victor - 2012-09-02

I'm happy to announce pymunk 3!

This release is a definite improvement over the 2.x release line of pymunk. It features a much improved documentation, an updated Chipmunk version with accompanying API adjustments, more and cooler examples. Also, to help to do quick prototyping pymunk now includes a new module pymunk.pygame_util that can draw most physics objects on a pygame surface. Check out the new pygame_util_demo.py example to get an understanding of how it works.

Another new feature is improved support to run in non-debug mode. Its now possible to pass a compile flag to setup.py to build Chipmunk in release mode and there's a new module, pymunkoptions that can be used to turn pymunk debug prints off.

This release has been tested and runs on CPython 2.6, 2.7, 3.2. At least one run of the unit tests have been made on the following platforms: 32 bit Python on Windows, 32 and 64 bit Python on Linux, and 32 and 64 bit Python on OSX.

This release has also been tested on Pypy 1.9, with all tests passed!

Changes

- Several new and interesting examples added
- New draw module to help with pygame prototyping
- New pymunkoptions module to allow disable of debug
- Tested on OSX, includes a compiled dylib file
- Much extended and reworked documentation and homepage
- Update of Chipmunk
- Smaller API changes
- General polish and cleanup
- Shining new domain: www.pymunk.org

I hope you will like it!

5.2.3 pymunk 2.1.0

Victor - 2011-12-03

A bugfix release of pymunk is here!

The most visible change in this release is that now the source release contains all of it including examples and chipmunk source. :) Other fixes are a new velocity limit property of the body, and some removed methods (Reasoning behind removing them and still on same version: You would get an exception calling them anyway. The removal should not affect code that works). Note, all users should create static bodies by setting the input parameters to None, not using infinity. inf will be removed in an upcoming release.

Changes

- Marked pymunk.inf as deprecated
- Added velocity limit property to the body
- Fixed bug on 64bit python
- Recompiled chipmunk.dll with python 2.5
- Updated chipmunk source.
- New method in Vec2d to get int tuple
- Removed slew and resize hash methods
- Removed pymunk.init calls from examples/tests
- Updated examples/tests to create static bodies the good way

Have fun with it!

5.2.4 pymunk 2.0.0

Victor - 2011-09-04

Today I'm happy to announce the new pymunk 2 release!

New goodies in this release comes mainly form the updated chipmunk library. Its now possible for bodies to sleep, there is a new data structure holding the objects and other smaller improvements. The updated version number comes mainly from the new sleep methods.

Another new item in the release is some simplification, you now don't need to initialize pymunk on your own, thats done automatically on import. Another cool feature is that pymunk passes all its unit tests on the latest pypy source which I think is a great thing! Have not had time to do any performance tests, but pypy claims improvements of the ctypes library over cpython.

Note, this release is not completely backwards compatible with pymunk 1.0, some minor adjustments will be necessary (one of the reasons the major version number were increased).

Changes from the last release:

- Removed init pymunk method, its done automatically on import
- Support for sleeping bodies.
- Updated to latest version of Chipmunk
- More API docs, more unit tests.
- Only dependent on msvcrt.dll on windows now.
- Removed dependency on setuptools
- Minor updates on other API, added some missing properties and methods.

Enjoy!

5.2.5 Older news

Older news items have been archived.

5.3 Installation

Tip: You will find the latest version at http://code.google.com/p/pymunk/downloads/list

pymunk does not need to be installed. However, you might need to compile the chipmunk library that pymunk uses internally if a precompiled library was not included for your platform. The default pymunk distribution ships with at least 32-bit Windows and 32- and 64-bit Linux versions. But even if a compiled library is not shipped you should not despair! It is very easy to compile chipmunk as long as you have a gcc-compatible c compiler installed.

5.3.1 Without installation

You might want to test pymunk before you install it on your development machine, or maybe decide to not install it at all. The easiest way to do this is to download the source release of pymunk and extract the archive to the folder where your code is and then tell python where to find it.

For example, lets say you have main.py and a folder named pymunk-3.0.0 that contain the full source of pymunk (files, including setup.py, README.txt and a folder named pymunk). Then you can add this before you include:

```
import os, sys

current_path = os.getcwd()
sys.path.insert(0, os.path.join( current_path, "pymunk-3.0.0" ) )

import pymunk
```

For a working demo, check out the run.py file in the examples folder of pymunk. It uses this exact technique to allow running the examples using the pymunk located on level up.

5.3.2 Install pymunk

pymunk can be installed with pip install:

```
> pip install pymunk
```

Another option is to install pymunk with one of the prebuilt installers if available (Windows) or you can use the standard setup.py way:

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

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.

Note: The setup will not check if the correct Chipmunk library was included or already compiled for you. It might be a good idea to first check that pymunk works before you actually install it. See *Without installation*

5.3.3 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.

To compile Chipmunk:

```
> python setup.py build_chipmunk
```

On Windows you will need to use a gcc-compatible compiler. The precompiled included windows binaries were compiled with a GCC installed from the TDM-GCC package http://tdm-gcc.tdragon.net/ To set the compiler use the compiler argument:

```
> python setup.py build_chipmunk --compiler=mingw32
```

To compile Chipmunk in Release mode use the release argument (Useful to avoid some debug prints and asserts):

```
> python setup.py build_chipmunk --release
```

See Also:

Module `pymunkoptions` Options module that control runtime options of pymunk such as debug settings. Use pymunkoptions together with release mode compilation to remove all debugs prints.

5.4 API Reference

5.4.1 `pymunk` Package

Submodules

`pymunk.constraint` Module

A constraint is something that describes how two bodies interact with each other. (how they constrain each other). Constraints can be simple joints that allow bodies to pivot around each other like the bones in your body, or they can be more abstract like the gear joint or motors.

This submodule contain all the constraints that are supported by pymunk.

Chipmunk has a good overview of the different constraint on youtube which works fine to showcase them in pymunk as well. http://www.youtube.com/watch?v=ZgJJZTS0aMM

class `pymunk.constraint.`**`Constraint`**(*constraint=None*)
Bases: `object`

Base class of all constraints.

You usually don't want to create instances of this class directly, but instead use one of the specific constraints such as the PinJoint.

`__init__`(*constraint=None*)

`a`
The first of the two bodies constrained

`activate_bodies`()
Activate the bodies this constraint is attached to

`b`
The second of the two bodies constrained

`error_bias`
The rate at which joint error is corrected.

Defaults to pow(1.0 - 0.1, 60.0) meaning that it will correct 10% of the error every 1/60th of a second.

`impulse`
Get the last impulse applied by this constraint.

`max_bias`
The maximum rate at which joint error is corrected. Defaults to infinity

`max_force`
The maximum force that the constraint can use to act on the two bodies. Defaults to infinity

class `pymunk.constraint.`**`DampedRotarySpring`**(*a*, *b*, *rest_angle*, *stiffness*, *damping*)
Bases: `pymunk.constraint.Constraint`

Like a damped spring, but works in an angular fashion

`__init__`(*a*, *b*, *rest_angle*, *stiffness*, *damping*)
Like a damped spring, but works in an angular fashion.

Parameters

rest_angle The relative angle in radians that the bodies want to have

stiffness The spring constant (Young's modulus).

damping How soft to make the damping of the spring.

a
The first of the two bodies constrained

activate_bodies()
Activate the bodies this constraint is attached to

b
The second of the two bodies constrained

damping
How soft to make the damping of the spring.

error_bias
The rate at which joint error is corrected.

Defaults to pow(1.0 - 0.1, 60.0) meaning that it will correct 10% of the error every 1/60th of a second.

impulse
Get the last impulse applied by this constraint.

max_bias
The maximum rate at which joint error is corrected. Defaults to infinity

max_force
The maximum force that the constraint can use to act on the two bodies. Defaults to infinity

rest_angle
The relative angle in radians that the bodies want to have

stiffness
The spring constant (Young's modulus).

torque_func
Set the torque function

func(self, relative_angle) -> torque

Callback Parameters

relative_angle [float] The relative angle

class pymunk.constraint.**DampedSpring**(*a*, *b*, *anchr1*, *anchr2*, *rest_length*, *stiffness*, *damping*)
Bases: pymunk.constraint.Constraint

A damped spring

__init__(*a*, *b*, *anchr1*, *anchr2*, *rest_length*, *stiffness*, *damping*)
Defined much like a slide joint.

Parameters

anchr1 [Vec2d or (x,y)] Anchor point 1, relative to body a

anchr2 [Vec2d or (x,y)] Anchor point 2, relative to body b

rest_length [float] The distance the spring wants to be.

stiffness [float] The spring constant (Young's modulus).

damping [float] How soft to make the damping of the spring.

a
The first of the two bodies constrained

activate_bodies()
Activate the bodies this constraint is attached to

anchr1

anchr2

b
The second of the two bodies constrained

damping
How soft to make the damping of the spring.

error_bias
The rate at which joint error is corrected.

Defaults to pow(1.0 - 0.1, 60.0) meaning that it will correct 10% of the error every 1/60th of a second.

impulse
Get the last impulse applied by this constraint.

max_bias
The maximum rate at which joint error is corrected. Defaults to infinity

max_force
The maximum force that the constraint can use to act on the two bodies. Defaults to infinity

rest_length
The distance the spring wants to be.

stiffness
The spring constant (Young's modulus).

class pymunk.constraint.**GearJoint**(*a*, *b*, *phase*, *ratio*)
Bases: pymunk.constraint.Constraint

Keeps the angular velocity ratio of a pair of bodies constant.

__init__(*a*, *b*, *phase*, *ratio*)
Keeps the angular velocity ratio of a pair of bodies constant. ratio is always measured in absolute terms. It is currently not possible to set the ratio in relation to a third body's angular velocity. phase is the initial angular offset of the two bodies.

a
The first of the two bodies constrained

activate_bodies()
Activate the bodies this constraint is attached to

b
The second of the two bodies constrained

error_bias
The rate at which joint error is corrected.

Defaults to pow(1.0 - 0.1, 60.0) meaning that it will correct 10% of the error every 1/60th of a second.

impulse
Get the last impulse applied by this constraint.

max_bias
The maximum rate at which joint error is corrected. Defaults to infinity

max_force
The maximum force that the constraint can use to act on the two bodies. Defaults to infinity

phase

ratio

class pymunk.constraint.**GrooveJoint**(*a*, *b*, *groove_a*, *groove_b*, *anchr2*)
Bases: pymunk.constraint.Constraint

Similar to a pivot joint, but one of the anchors is on a linear slide instead of being fixed.

__init__(*a*, *b*, *groove_a*, *groove_b*, *anchr2*)
The groove goes from groove_a to groove_b on body a, and the pivot is attached to anchr2 on body b. All coordinates are body local.

a
The first of the two bodies constrained

activate_bodies()
Activate the bodies this constraint is attached to

anchr2

b
The second of the two bodies constrained

error_bias
The rate at which joint error is corrected.

Defaults to pow(1.0 - 0.1, 60.0) meaning that it will correct 10% of the error every 1/60th of a second.

groove_a

groove_b

impulse
Get the last impulse applied by this constraint.

max_bias
The maximum rate at which joint error is corrected. Defaults to infinity

max_force
The maximum force that the constraint can use to act on the two bodies. Defaults to infinity

class pymunk.constraint.**PinJoint**(*a*, *b*, *anchr1=(0, 0)*, *anchr2=(0, 0)*)
Bases: pymunk.constraint.Constraint

Keeps the anchor points at a set distance from one another.

__init__(*a*, *b*, *anchr1=(0, 0)*, *anchr2=(0, 0)*)
a and b are the two bodies to connect, and anchr1 and anchr2 are the anchor points on those bodies.

a
The first of the two bodies constrained

activate_bodies()
Activate the bodies this constraint is attached to

anchr1

anchr2

b
The second of the two bodies constrained

distance

error_bias
The rate at which joint error is corrected.

Defaults to pow(1.0 - 0.1, 60.0) meaning that it will correct 10% of the error every 1/60th of a second.

impulse
Get the last impulse applied by this constraint.

max_bias
The maximum rate at which joint error is corrected. Defaults to infinity

max_force
The maximum force that the constraint can use to act on the two bodies. Defaults to infinity

class pymunk.constraint.**PivotJoint**(*a*, *b*, **args*)
Bases: pymunk.constraint.Constraint

Simply allow two objects to pivot about a single point.

__init__(*a*, *b*, **args*)
a and b are the two bodies to connect, and pivot is the point in world coordinates of the pivot. Because the pivot location is given in world coordinates, you must have the bodies moved into the correct positions already. Alternatively you can specify the joint based on a pair of anchor points, but make sure you have the bodies in the right place as the joint will fix itself as soon as you start simulating the space.

That is, either create the joint with PivotJoint(a, b, pivot) or PivotJoint(a, b, anchr1, anchr2).

a [*Body*] The first of the two bodies

b [*Body*] The second of the two bodies

args [[Vec2d] or [Vec2d,Vec2d]] Either one pivot point, or two anchor points

a
The first of the two bodies constrained

activate_bodies()
Activate the bodies this constraint is attached to

anchr1

anchr2

b
The second of the two bodies constrained

error_bias
The rate at which joint error is corrected.

Defaults to pow(1.0 - 0.1, 60.0) meaning that it will correct 10% of the error every 1/60th of a second.

impulse
Get the last impulse applied by this constraint.

max_bias
The maximum rate at which joint error is corrected. Defaults to infinity

max_force
The maximum force that the constraint can use to act on the two bodies. Defaults to infinity

class pymunk.constraint.**RatchetJoint**(*a*, *b*, *phase*, *ratchet*)
Bases: pymunk.constraint.Constraint

Works like a socket wrench.

__init__(*a*, *b*, *phase*, *ratchet*)
Works like a socket wrench. ratchet is the distance between "clicks", phase is the initial offset to use when deciding where the ratchet angles are.

a
The first of the two bodies constrained

activate_bodies()
Activate the bodies this constraint is attached to

angle

b
The second of the two bodies constrained

error_bias
The rate at which joint error is corrected.

Defaults to pow(1.0 - 0.1, 60.0) meaning that it will correct 10% of the error every 1/60th of a second.

impulse
Get the last impulse applied by this constraint.

max_bias
The maximum rate at which joint error is corrected. Defaults to infinity

max_force
The maximum force that the constraint can use to act on the two bodies. Defaults to infinity

phase

ratchet

class pymunk.constraint.**RotaryLimitJoint**(*a*, *b*, *min*, *max*)
Bases: pymunk.constraint.Constraint

Constrains the relative rotations of two bodies.

__init__(*a*, *b*, *min*, *max*)
Constrains the relative rotations of two bodies. min and max are the angular limits in radians. It is implemented so that it's possible to for the range to be greater than a full revolution.

a
The first of the two bodies constrained

activate_bodies()
Activate the bodies this constraint is attached to

b
The second of the two bodies constrained

error_bias
The rate at which joint error is corrected.

Defaults to pow(1.0 - 0.1, 60.0) meaning that it will correct 10% of the error every 1/60th of a second.

impulse
Get the last impulse applied by this constraint.

max

max_bias
The maximum rate at which joint error is corrected. Defaults to infinity

max_force
The maximum force that the constraint can use to act on the two bodies. Defaults to infinity

min

class pymunk.constraint.**SimpleMotor**(*a*, *b*, *rate*)
Bases: pymunk.constraint.Constraint

Keeps the relative angular velocity of a pair of bodies constant.

__init__(*a*, *b*, *rate*)
Keeps the relative angular velocity of a pair of bodies constant. rate is the desired relative angular velocity. You will usually want to set an force (torque) maximum for motors as otherwise they will be able to apply a nearly infinite torque to keep the bodies moving.

a
The first of the two bodies constrained

activate_bodies()
Activate the bodies this constraint is attached to

b
The second of the two bodies constrained

error_bias
The rate at which joint error is corrected.

Defaults to pow(1.0 - 0.1, 60.0) meaning that it will correct 10% of the error every 1/60th of a second.

impulse
Get the last impulse applied by this constraint.

max_bias
The maximum rate at which joint error is corrected. Defaults to infinity

max_force
The maximum force that the constraint can use to act on the two bodies. Defaults to infinity

rate
The desired relative angular velocity

class pymunk.constraint.**SlideJoint**(*a*, *b*, *anchr1*, *anchr2*, *min*, *max*)
Bases: pymunk.constraint.Constraint

Like pin joints, but have a minimum and maximum distance. A chain could be modeled using this joint. It keeps the anchor points from getting to far apart, but will allow them to get closer together.

__init__(*a*, *b*, *anchr1*, *anchr2*, *min*, *max*)
a and b are the two bodies to connect, anchr1 and anchr2 are the anchor points on those bodies, and min and max define the allowed distances of the anchor points.

a
The first of the two bodies constrained

activate_bodies()
Activate the bodies this constraint is attached to

anchr1

anchr2

b
The second of the two bodies constrained

error_bias
The rate at which joint error is corrected.

Defaults to pow(1.0 - 0.1, 60.0) meaning that it will correct 10% of the error every 1/60th of a second.

impulse
Get the last impulse applied by this constraint.

max

max_bias
The maximum rate at which joint error is corrected. Defaults to infinity

max_force
The maximum force that the constraint can use to act on the two bodies. Defaults to infinity

min

pymunk.vec2d Module

This module contain the Vec2d class that is used in all of pymunk when a vector is needed.

The Vec2d class is used almost everywhere in pymunk for 2d coordinates and vectors, for example to define gravity vector in a space. However, pymunk is smart enough to convert tuples or tuple like objects to Vec2ds so you usually do not need to explcitily do conversions if you happen to have a tuple:

```
>>> import pymunk
>>> space = pymunk.Space()
>>> print space.gravity
Vec2d(0.0, 0.0)
>>> space.gravity = 3,5
>>> print space.gravity
Vec2d(3.0, 5.0)
>>> space.gravity += 2,6
>>> print space.gravity
Vec2d(5.0, 11.0)
```

class pymunk.vec2d.**Vec2d**(*x_or_pair=None*, *y=None*)
Bases: _ctypes.Structure

2d vector class, supports vector and scalar operators, and also provides some high level functions.

__init__(*x_or_pair=None*, *y=None*)

angle
Gets or sets the angle (in radians) of a vector

angle_degrees
Gets or sets the angle (in degrees) of a vector

convert_to_basis(*x_vector*, *y_vector*)

cpvrotate(*other*)
Uses complex multiplication to rotate this vector by the other.

cpvunrotate(*other*)
The inverse of cpvrotate

cross(*other*)
The cross product between the vector and other vector v1.cross(v2) -> v1.x*v2.y - v2.y*v1.x

Returns The cross product

dot(*other*)
The dot product between the vector and other vector v1.dot(v2) -> v1.x*v2.x + v1.y*v2.y

Returns The dot product

classmethod from_param(*arg*)
Used by ctypes to automatically create Vec2ds

get_angle()

get_angle_between(*other*)
Get the angle between the vector and the other in radians

Returns The angle

get_angle_degrees()

get_angle_degrees_between(*other*)
Get the angle between the vector and the other in degrees

Returns The angle (in degrees)

get_dist_sqrd(*other*)
The squared distance between the vector and other vector It is more efficent to use this method than to call get_distance() first and then do a sqrt() on the result.

Returns The squared distance

get_distance(*other*)
The distance between the vector and other vector

Returns The distance

get_length()
Get the length of the vector.

Returns The length

get_length_sqrd()
Get the squared length of the vector. It is more efficent to use this method instead of first call get_length() or access .length and then do a sqrt().

Returns The squared length

int_tuple
Return the x and y values of this vector as ints

interpolate_to(*other*, *range*)

length
Gets or sets the magnitude of the vector

normalize_return_length()
Normalize the vector and return its length before the normalization

Returns The length before the normalization

normalized()
Get a normalized copy of the vector Note: This function will return 0 if the length of the vector is 0.

Returns A normalized vector

static ones()
A vector where both x and y is 1

perpendicular()

perpendicular_normal()

projection(*other*)

rotate(*angle_radians*)
Rotate the vector by angle_radians radians.

rotate_degrees(*angle_degrees*)
Rotate the vector by angle_degrees degrees.

rotated(*angle_radians*)
Create and return a new vector by rotating this vector by angle_radians radians.

Returns Rotated vector

rotated_degrees(*angle_degrees*)
Create and return a new vector by rotating this vector by angle_degrees degrees.

Returns Rotade vector

static **unit**()
A unit vector pointing up

x
Structure/Union member

y
Structure/Union member

static **zero**()
A vector of zero length

pymunk.util Module

This submodule contains utility functions, mainly to help with polygon creation.

pymunk.util.**is_clockwise**(*points*)
Check if the points given forms a clockwise polygon

Returns True if the points forms a clockwise polygon

pymunk.util.**reduce_poly**(*points*, *tolerance=0.5*)
Remove close points to simplify a polyline tolerance is the min distance between two points squared.

Returns The reduced polygon as a list of (x,y)

pymunk.util.**convex_hull**(*points*)
Create a convex hull from a list of points. This function uses the Graham Scan Algorithm.

Returns Convex hull as a list of (x,y)

pymunk.util.**calc_area**(*points*)
Calculate the area of a polygon

Returns Area of polygon

pymunk.util.**calc_center**(*points*)
Calculate the center of a polygon

Returns The center (x,y)

pymunk.util.**poly_vectors_around_center**(*pointlist*, *points_as_Vec2d=True*)
Rearranges vectors around the center If points_as_Vec2d, then return points are also Vec2d, else pos

Returns pointlist ([Vec2d/pos, ...])

pymunk.util.**is_convex**(*points*)
Test if a polygon (list of (x,y)) is convex or not

Returns True if the polygon is convex, False otherwise

pymunk.util.**calc_perimeter**(*points*)
Calculate the perimeter of a polygon

Returns Perimeter of polygon

pymunk.util.**triangulate**(*poly*)
Triangulates poly and returns a list of triangles

Parameters

poly list of points that form an anticlockwise polygon (self-intersecting polygons won't work, results are undefined)

pymunk.util.**convexise**(*triangles*)
Reduces a list of triangles (such as returned by triangulate()) to a non-optimum list of convex polygons

Parameters

triangles list of anticlockwise triangles (a list of three points) to reduce

pymunk.pygame_util Module

This submodule contains helper functions to help with quick prototyping using pymunk 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.

pymunk.pygame_util.**draw**(*surface*, **objs*)
Draw one or many pymunk objects on a pygame.Surface object.

This method currently supports drawing of

- pymunk.Space
- pymunk.Segment
- pymunk.Circle
- pymunk.Poly
- pymunk.Constraint objects

If a Space is passed in all shapes in that space will be drawn. Unrecognized objects will be ignored (for example if you pass in a constraint).

Typical usage:

```
>>> pymunk.pygame_util.draw(screen, my_space)
```

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

```
>>> my_shape.color = pygame.color.THECOLORS["pink"]
```

If you do not want a shape to be drawn, set shape.ignore_draw to True.

```
>>> my_shape.ignore_draw = True
```

Not all constraints are currently drawn in a very clear way, but all the different shapes should look fine both as static and dynamic objects.

See pygame_util.demo.py for a full example

Parameters

surface [pygame.Surface] Surface that the objects will be drawn on

objs [One or many objects to draw] Can be either a single object or a list like container with objects.

pymunk.pygame_util.**get_mouse_pos**(*surface*)
Get position of the mouse pointer in pymunk coordinates.

pymunk.pygame_util.**to_pygame**(*p*, *surface*)
Convenience method to convert pymunk coordinates to pygame surface local coordinates

pymunk.pygame_util.**from_pygame**(*p*, *surface*)
Convenience method to convert pygame surface local coordinates to pymunk coordinates

pymunk.pyglet_util Module

This submodule contains helper functions to help with quick prototyping using pymunk 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 is opinionated about your coordinate system and not very optimized (they use batched drawing, but there is probably room for optimizations still).

pymunk.pyglet_util.**draw**(**objs*, ***kwargs*)
Draw one or many pymunk objects. It is perfectly fine to pass in a whole Space object.

Objects that can be handled are:

- pymunk.Space
- pymunk.Segment
- pymunk.Circle
- pymunk.Poly

If a Space is passed in all shapes in that space will be drawn. Unrecognized objects will be ignored (for example if you pass in a constraint).

Typical usage:

```
>>> pymunk.pyglet_util.draw(my_space)
```

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

```
>>> my_shape.color = (255, 0, 0) # will draw my_shape in red
```

If you do not want a shape to be drawn, set shape.ignore_draw to True.

```
>>> my_shape.ignore_draw = True
```

(However, if you want to ignore most shapes its probably more performant to only pass in those shapes that you want to be drawn to the draw method)

You can optionally pass in a batch to use. Just remember that you need to call draw yourself.

```
>>> pymunk.pyglet_util.draw(my_shape, batch = my_batch)
>>> my_batch.draw()
```

See pyglet_util.demo.py for a full example

Param

objs [One or many objects to draw.] Can be either a single object or a list like container with objects.

kwargs [You can optionally pass in a pyglet.graphics.Batch] If a batch is given all drawing will use this batch to draw on. If no batch is given a a new batch will be used for the drawing. Remember that if you pass in your own batch you need to call draw on it yourself.

pymunkoptions Module

Use this module to set runtime options of pymunk.

Currently there is one option that can be changed, debug. By setting debug to false debug print outs will be limited. In order to remove all debug prints you will also need to compile chipmunk in release mode. See *Compile Chipmunk* for details on how to compile chipmunk.

pymunkoptions.**options = {'debug': True}**

Global dict of pymunk options. To change make sure you import pymunk before any sub-packages and then set the option you want. For example:

```
import pymunkoptions
pymunkoptions.options["debug"] = False
import pymunk

#..continue to use pymunk as you normally do
```

pymunk

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.pymunk.org

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

When you import this module it will automatically load the chipmunk library file. As long as you haven't turned off the debug mode a print will show exactly which Chipmunk library file it loaded. For example:

```
>>> import pymunk
Loading chipmunk for Windows (32bit) [C:\code\pymunk\chipmunk.dll]
```

pymunk.**inf = 1e+100**

Infinity that can be passed as mass or inertia to Body.

Useful when you for example want a body that cannot rotate, just set its moment to inf. Just remember that if two objects with both infinite masses collides the world might explode. Similary effects can happen with infinite moment.

Note: In previous versions of pymunk you used inf to create static bodies. This has changed and you should instead do it by invoking the body constructor without any arguments.

---

pymunk.**version = '4.0.0'**
The release version of this pymunk installation. Valid only if pymunk was installed from a source or binary distribution (i.e. not in a checked-out copy from svn).

pymunk.**chipmunk_version = '<Mock object at 0x130a1d0>R3bdf1b7b3c'**
The Chipmunk version compatible with this pymunk version. Other (newer) Chipmunk versions might also work if the new version does not contain any breaking API changes.

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

The string is in the following format: <cpVersionString>R<svn or github commit of chipmunk> where cpVersionString is a version string set by Chipmunk and the svn version corresponds to the svn version of the chipmunk source files included with pymunk or the github commit hash. If the Chipmunk version is a release then the second part will be empty

---

Note: This is also the version of the Chipmunk source files included in the chipmunk_src folder (normally included in the pymunk source distribution).

---

class pymunk.**Space**(*iterations=10*)
Bases: object

Spaces are the basic unit of simulation. You add rigid bodies, shapes and joints to it and then step them all forward together through time.

__init__(*iterations=10*)
Create a new instace of the Space

Its usually best to keep the elastic_iterations setting to 0. Only change if you have problem with stacking elastic objects on each other. If that is the case, try to raise it. However, a value other than 0 will affect other parts, most importantly you wont get reliable total_impulse readings from the *Arbiter* object in collsion callbacks!

Parameters

iterations [int] Number of iterations to use in the impulse solver to solve contacts.

add(**objs*)
Add one or many shapes, bodies or joints to the space

Unlike Chipmunk and earlier versions of pymunk its now allowed to add objects even from a callback during the simulation step. However, the add will not be performed until the end of the step.

add_collision_handler(*a*, *b*, *begin=None*, *pre_solve=None*, *post_solve=None*, *separate=None*, **args*, ***kwargs*)
Add a collision handler for given collision type pair.

Whenever a shapes with collision_type a and collision_type b collide, these callbacks will be used to process the collision. None can be provided for callbacks you do not wish to implement, however pymunk will call it's own default versions for these and not the default ones you've set up for the Space. If you need to fall back on the space's default callbacks, you'll have to provide them individually to each handler definition.

Parameters

a [int] Collision type of the first shape

b [int] Collision type of the second shape

begin [`func(space, arbiter, *args, **kwargs) -> bool`] Collision handler called when two shapes just started touching for the first time this step. Return false

from the callback to make pymunk ignore the collision or true to process it normally. Rejecting a collision from a begin() callback permanently rejects the collision until separation. Pass *None* if you wish to use the pymunk default.

pre_solve [`func(space, arbiter, *args, **kwargs) -> bool`] Collision handler called when two shapes are touching. Return false from the callback to make pymunk ignore the collision or true to process it normally. Additionally, you may override collision values such as *Arbiter.elasticity* and *Arbiter.friction* to provide custom friction or elasticity values. See *Arbiter* for more info. Pass *None* if you wish to use the pymunk default.

post_solve [`func(space, arbiter, *args, **kwargs)`] Collsion handler called when two shapes are touching and their collision response has been processed. You can retrieve the collision force at this time if you want to use it to calculate sound volumes or damage amounts. Pass *None* if you wish to use the pymunk default.

separate [`func(space, arbiter, *args, **kwargs)`] Collision handler called when two shapes have just stopped touching for the first time this frame. Pass *None* if you wish to use the pymunk default.

args Optional parameters passed to the collision handler functions.

kwargs Optional keyword parameters passed on to the collision handler functions.

add_post_step_callback(*callback_function*, *obj*, **args*, ***kwargs*)

Add a function to be called last in the next simulation step.

Post step callbacks are registered as a function and an object used as a key. You can only register one post step callback per object.

This function was more useful with earlier versions of pymunk where you weren't allowed to use the add and remove methods on the space during a simulation step. But this function is still available for other uses and to keep backwards compatibility.

Note: If you remove a shape from the callback it will trigger the collision handler for the 'separate' event if it the shape was touching when removed.

Parameters

callback_function [`func(obj, *args, **kwargs)`] The callback function.

obj [Any object] This object is used as a key, you can only have one callback for a single object. It is passed on to the callback function.

args Optional parameters passed to the callback function.

kwargs Optional keyword parameters passed on to the callback function.

Return True if key was not previously added, False otherwise

bb_query(*bb*, *layers=-1*, *group=0*)

Perform a fast rectangle query on the space.

Only the shape's bounding boxes are checked for overlap, not their full shape. Returns a list of shapes.

bodies

A list of the bodies added to this space

collision_bias

Determines how fast overlapping shapes are pushed apart.

Expressed as a fraction of the error remaining after each second. Defaults to pow(1.0 - 0.1, 60.0) meaning that pymunk fixes 10% of overlap each frame at 60Hz.

collision_persistence
Number of frames that contact information should persist.

Defaults to 3. There is probably never a reason to change this value.

collision_slop
Amount of allowed penetration.

Used to reduce oscillating contacts and keep the collision cache warm. Defaults to 0.1. If you have poor simulation quality, increase this number as much as possible without allowing visible amounts of overlap.

constraints
A list of the constraints added to this space

damping
Damping rate expressed as the fraction of velocity bodies retain each second.

A value of 0.9 would mean that each body's velocity will drop 10% per second. The default value is 1.0, meaning no damping is applied.

enable_contact_graph
Rebuild the contact graph during each step.

Must be enabled to use the get_arbiter() function on Body Disabled by default for a small performance boost. Enabled implicitly when the sleeping feature is enabled.

gravity
Default gravity to supply when integrating rigid body motions.

idle_speed_threshold
Speed threshold for a body to be considered idle. The default value of 0 means to let the space guess a good threshold based on gravity.

iterations
Number of iterations to use in the impulse solver to solve contacts.

nearest_point_query(*point*, *max_distance*, *layers=-1*, *group=0*)
Query space at point filtering out matches with the given layers and group. Return a list of all shapes within max_distance of the point.

If you don't want to filter out any matches, use -1 for the layers and 0 as the group.

Parameters

point [(x,y) or *Vec2d*] Define where to check for collision in the space.

max_distance [int] Maximumm distance of shape from point

layers [int] Only pick shapes matching the bit mask. i.e. (layers & shape.layers) != 0

group [int] Only pick shapes not in this group.

Return [dict(shape='Shape', distance = distance, point = Vec2d)]

nearest_point_query_nearest(*point*, *max_distance*, *layers=-1*, *group=0*)
Query space at point filtering out matches with the given layers and group. Return nearest of all shapes within max_distance of the point.

If you don't want to filter out any matches, use -1 for the layers and 0 as the group.

Parameters

point [(x,y) or *Vec2d*] Define where to check for collision in the space.

max_distance [int] Maximumm distance of shape from point

layers [int] Only pick shapes matching the bit mask. i.e. (layers & shape.layers) != 0

group [int] Only pick shapes not in this group.

Return dict(shape='Shape', distance = distance, point = Vec2d)

point_query(*point*, *layers=-1*, *group=0*)
Query space at point filtering out matches with the given layers and group. Return a list of found shapes.

If you don't want to filter out any matches, use -1 for the layers and 0 as the group.

Parameters

point [(x,y) or *Vec2d*] Define where to check for collision in the space.

layers [int] Only pick shapes matching the bit mask. i.e. (layers & shape.layers) != 0

group [int] Only pick shapes not in this group.

point_query_first(*point*, *layers=-1*, *group=0*)
Query space at point and return the first shape found matching the given layers and group. Returns None if no shape was found.

Parameters

point [(x,y) or *Vec2d*] Define where to check for collision in the space.

layers [int] Only pick shapes matching the bit mask. i.e. (layers & shape.layers) != 0

group [int] Only pick shapes not in this group.

reindex_shape(*shape*)
Update the collision detection data for a specific shape in the space.

reindex_static()
Update the collision detection info for the static shapes in the space. You only need to call this if you move one of the static shapes.

remove(**objs*)
Remove one or many shapes, bodies or constraints from the space

Unlike Chipmunk and earlier versions of pymunk its now allowed to remove objects even from a callback during the simulation step. However, the removal will not be performed until the end of the step.

Note: When removing objects from the space, make sure you remove any other objects that reference it. For instance, when you remove a body, remove the joints and shapes attached to it.

---

remove_collision_handler(*a*, *b*)
Remove a collision handler for a given collision type pair.

Parameters

a [int] Collision type of the first shape

b [int] Collision type of the second shape

segment_query(*start*, *end*, *layers=-1*, *group=0*)
Query space along the line segment from start to end filtering out matches with the given layers and group.

Segment queries are like ray casting, but because pymunk uses a spatial hash to process collisions, it cannot process infinitely long queries like a ray. In practice this is still very fast and you don't need to worry too much about the performance as long as you aren't using very long segments for your queries.

Return [*SegmentQueryInfo*] - One SegmentQueryInfo object for each hit.

segment_query_first(*start*, *end*, *layers=-1*, *group=0*)
Query space along the line segment from start to end filtering out matches with the given layers and group. Only the first shape encountered is returned and the search is short circuited. Returns None if no shape was found.

set_default_collision_handler(*begin=None*, *pre_solve=None*, *post_solve=None*, *separate=None*, **args*, ***kwargs*)
Register a default collision handler to be used when no specific collision handler is found. If you do nothing, the space will be given a default handler that accepts all collisions in begin() and pre_solve() and does nothing for the post_solve() and separate() callbacks.

Parameters

begin [`func(space, arbiter, *args, **kwargs) -> bool`] Collision handler called when two shapes just started touching for the first time this step. Return False from the callback to make pymunk ignore the collision or True to process it normally. Rejecting a collision from a begin() callback permanently rejects the collision until separation. Pass *None* if you wish to use the pymunk default.

pre_solve [`func(space, arbiter, *args, **kwargs) -> bool`] Collision handler called when two shapes are touching. Return False from the callback to make pymunk ignore the collision or True to process it normally. Additionally, you may override collision values such as Arbiter.elasticity and Arbiter.friction to provide custom friction or elasticity values. See Arbiter for more info. Pass *None* if you wish to use the pymunk default.

post_solve [`func(space, arbiter, *args, **kwargs)`] Collsion handler called when two shapes are touching and their collision response has been processed. You can retrieve the collision force at this time if you want to use it to calculate sound volumes or damage amounts. Pass *None* if you wish to use the pymunk default.

separate [`func(space, arbiter, *args, **kwargs)`] Collision handler called when two shapes have just stopped touching for the first time this frame. Pass *None* if you wish to use the pymunk default.

args Optional parameters passed to the collision handler functions.

kwargs Optional keyword parameters passed on to the collision handler functions.

shape_query(*shape*)
Query a space for any shapes overlapping the given shape

Returns a list of shapes.

shapes
A list of all the shapes added to this space (both static and non-static)

sleep_time_threshold
Time a group of bodies must remain idle in order to fall asleep.

Enabling sleeping also implicitly enables the the contact graph. The default value of *inf* disables the sleeping algorithm.

static_body
A convenience static body already added to the space

step(*dt*)
Update the space for the given time step. Using a fixed time step is highly recommended. Doing so will increase the efficiency of the contact persistence, requiring an order of magnitude fewer iterations to resolve the collisions in the usual case.

class pymunk.**Body**(*mass=None*, *moment=None*)

Bases: object

A rigid body

- Use forces to modify the rigid bodies if possible. This is likely to be the most stable.
- Modifying a body's velocity shouldn't necessarily be avoided, but applying large changes can cause strange results in the simulation. Experiment freely, but be warned.
- Don't modify a body's position every step unless you really know what you are doing. Otherwise you're likely to get the position/velocity badly out of sync.

__init__(*mass=None*, *moment=None*)

Create a new Body

To create a static body, pass in None for mass and moment.

activate()

Wake up a sleeping or idle body.

angle

The rotation of the body.

Note: If you get small/no changes to the angle when for example a ball is "rolling" down a slope it might be because the Circle shape attached to the body or the slope shape does not have any friction set.

---

angular_velocity

angular_velocity_limit

apply_force(*f*, *r=(0, 0)*)

Apply (accumulate) the force f on body at a relative offset (important!) r from the center of gravity.

Both r and f are in world coordinates.

Parameters

f [(x,y) or *Vec2d*] Force in world coordinates

r [(x,y) or *Vec2d*] Offset in world coordinates

apply_impulse(*j*, *r=(0, 0)*)

Apply the impulse j to body at a relative offset (important!) r from the center of gravity. Both r and j are in world coordinates.

Parameters

j [(x,y) or *Vec2d*] Impulse to be applied

r [(x,y) or *Vec2d*] Offset the impulse with this vector

constraints

Get the constraints this body is attached to.

each_arbiter(*func*, **args*, ***kwargs*)

Run func on each of the arbiters on this body.

```
func(arbiter, *args, **kwargs) -> None
```

Callback Parameters

arbiter [*Arbiter*] The Arbiter

args Optional parameters passed to the callback function.

kwargs Optional keyword parameters passed on to the callback function.

> **Warning:** Do not hold on to the Arbiter after the callback!

force

is_rogue
Returns true if the body has not been added to a space.

is_sleeping
Returns true if the body is sleeping.

is_static
Returns true if the body is a static body

kinetic_energy
Get the kinetic energy of a body.

local_to_world(*v*)
Convert body local coordinates to world space coordinates

Parameters

v [(x,y) or *Vec2d*] Vector in body local coordinates

mass

moment

position

position_func
The position callback function. The position callback function is called each time step and can be used to update the body's position.

```
func(body, dt) -> None
```

Callback Parameters

body [*Body*] Body that should have its velocity calculated

dt [float] Delta time since last step.

reset_forces()
Zero both the forces and torques accumulated on body

rotation_vector

shapes
Get the shapes attached to this body.

sleep()
Force a body to fall asleep immediately.

sleep_with_group(*body*)
Force a body to fall asleep immediately along with other bodies in a group.

torque

static update_position(*body*, *dt*)
Default rigid body position integration function.

Updates the position of the body using Euler integration. Unlike the velocity function, it's unlikely you'll want to override this function. If you do, make sure you understand it's source code (in Chipmunk) as it's an important part of the collision/joint correction process.

static update_velocity(*body*, *gravity*, *damping*, *dt*)
Default rigid body velocity integration function.

Updates the velocity of the body using Euler integration.

velocity

velocity_func
The velocity callback function. The velocity callback function is called each time step, and can be used to set a body's velocity.

```
func(body, gravity, damping, dt) -> None
```

Callback Parameters

body [*Body*] Body that should have its velocity calculated

gravity [*Vec2d*] The gravity vector

damping [float] The damping

dt [float] Delta time since last step.

velocity_limit

world_to_local(*v*)
Convert world space coordinates to body local coordinates

Parameters

v [(x,y) or *Vec2d*] Vector in world space coordinates

class pymunk.**Shape**(*shape=None*)
Bases: object

Base class for all the shapes.

You usually dont want to create instances of this class directly but use one of the specialized shapes instead.

__init__(*shape=None*)

bb
The bounding box of the shape. Only guaranteed to be valid after Shape.cache_bb() or Space.step() is called. Moving a body that a shape is connected to does not update it's bounding box. For shapes used for queries that aren't attached to bodies, you can also use Shape.update().

body
The body this shape is attached to. Can be set to None to indicate that this shape doesnt belong to a body.

cache_bb()
Update and returns the bouding box of this shape

collision_type
User defined collision type for the shape. See add_collisionpair_func function for more information on when to use this property

elasticity
Elasticity of the shape. A value of 0.0 gives no bounce, while a value of 1.0 will give a 'perfect' bounce. However due to inaccuracies in the simulation using 1.0 or greater is not recommended.

friction
Friction coefficient. pymunk uses the Coulomb friction model, a value of 0.0 is frictionless.

A value over 1.0 is perfectly fine.

Some real world example values from wikipedia (Remember that it is what looks good that is important, not the exact value).

Material	Other	Friction
Aluminium	Steel	0.61
Copper	Steel	0.53
Brass	Steel	0.51
Cast iron	Copper	1.05
Cast iron	Zinc	0.85
Concrete (wet)	Rubber	0.30
Concrete (dry)	Rubber	1.0
Concrete	Wood	0.62
Copper	Glass	0.68
Glass	Glass	0.94
Metal	Wood	0.5
Polyethene	Steel	0.2
Steel	Steel	0.80
Steel	Teflon	0.04
Teflon (PTFE)	Teflon	0.04
Wood	Wood	0.4

group
Shapes in the same non-zero group do not generate collisions. Useful when creating an object out of many shapes that you don't want to self collide. Defaults to 0

layers
Shapes only collide if they are in the same bit-planes. i.e. (a.layers & b.layers) != 0. By default, a shape occupies all 32 bit-planes, i.e. layers == -1

point_query(*p*)
Check if the given point lies within the shape.

segment_query(*start*, *end*)
Check if the line segment from start to end intersects the shape.

Return either SegmentQueryInfo object or None

sensor
A boolean value if this shape is a sensor or not. Sensors only call collision callbacks, and never generate real collisions.

surface_velocity
The surface velocity of the object. Useful for creating conveyor belts or players that move around. This value is only used when calculating friction, not resolving the collision.

update(*position*, *rotation_vector*)
Update, cache and return the bounding box of a shape with an explicit transformation.

Useful if you have a shape without a body and want to use it for querying.

class pymunk.**Circle**(*body*, *radius*, *offset=(0, 0)*)
Bases: pymunk.Shape

A circle shape defined by a radius

This is the fastest and simplest collision shape

__init__(*body*, *radius*, *offset=(0, 0)*)
body is the body attach the circle to, offset is the offset from the body's center of gravity in body local coordinates.

It is legal to send in None as body argument to indicate that this shape is not attached to a body.

bb
The bounding box of the shape. Only guaranteed to be valid after Shape.cache_bb() or Space.step() is called. Moving a body that a shape is connected to does not update it's bounding box. For shapes used for queries that aren't attached to bodies, you can also use Shape.update().

body
The body this shape is attached to. Can be set to None to indicate that this shape doesnt belong to a body.

cache_bb()
Update and returns the bouding box of this shape

collision_type
User defined collision type for the shape. See add_collisionpair_func function for more information on when to use this property

elasticity
Elasticity of the shape. A value of 0.0 gives no bounce, while a value of 1.0 will give a 'perfect' bounce. However due to inaccuracies in the simulation using 1.0 or greater is not recommended.

friction
Friction coefficient. pymunk uses the Coulomb friction model, a value of 0.0 is frictionless.

A value over 1.0 is perfectly fine.

Some real world example values from wikipedia (Remember that it is what looks good that is important, not the exact value).

Material	Other	Friction
Aluminium	Steel	0.61
Copper	Steel	0.53
Brass	Steel	0.51
Cast iron	Copper	1.05
Cast iron	Zinc	0.85
Concrete (wet)	Rubber	0.30
Concrete (dry)	Rubber	1.0
Concrete	Wood	0.62
Copper	Glass	0.68
Glass	Glass	0.94
Metal	Wood	0.5
Polyethene	Steel	0.2
Steel	Steel	0.80
Steel	Teflon	0.04
Teflon (PTFE)	Teflon	0.04
Wood	Wood	0.4

group
Shapes in the same non-zero group do not generate collisions. Useful when creating an object out of many shapes that you don't want to self collide. Defaults to 0

layers
Shapes only collide if they are in the same bit-planes. i.e. (a.layers & b.layers) != 0. By default, a shape occupies all 32 bit-planes, i.e. layers == -1

offset
Offset. (body space coordinates)

point_query(*p*)
Check if the given point lies within the shape.

radius
The Radius of the circle

segment_query(*start*, *end*)
Check if the line segment from start to end intersects the shape.

Return either SegmentQueryInfo object or None

sensor
A boolean value if this shape is a sensor or not. Sensors only call collision callbacks, and never generate real collisions.

surface_velocity
The surface velocity of the object. Useful for creating conveyor belts or players that move around. This value is only used when calculating friction, not resolving the collision.

unsafe_set_offset(*o*)
Unsafe set the offset of the circle.

Note: This change is only picked up as a change to the position of the shape's surface, but not it's velocity. Changing it will not result in realistic physical behavior. Only use if you know what you are doing!

---

unsafe_set_radius(*r*)
Unsafe set the radius of the circle.

Note: This change is only picked up as a change to the position of the shape's surface, but not it's velocity. Changing it will not result in realistic physical behavior. Only use if you know what you are doing!

---

update(*position*, *rotation_vector*)
Update, cache and return the bounding box of a shape with an explicit transformation.

Useful if you have a shape without a body and want to use it for querying.

class pymunk.**Poly**(*body*, *vertices*, *offset=(0, 0)*, *radius=0*)
Bases: pymunk.Shape

A convex polygon shape

Slowest, but most flexible collision shape.

It is legal to send in None as body argument to indicate that this shape is not attached to a body.

__init__(*body*, *vertices*, *offset=(0, 0)*, *radius=0*)
Create a polygon

body [*Body*] The body to attach the poly to

vertices [[(x,y)] or [*Vec2d*]] Define a convex hull of the polygon with a counterclockwise winding.

offset [(x,y) or *Vec2d*] The offset from the body's center of gravity in body local coordinates.

radius [int] Set the radius of the poly shape.

bb
The bounding box of the shape. Only guaranteed to be valid after Shape.cache_bb() or Space.step() is called. Moving a body that a shape is connected to does not update it's bounding box. For shapes used for queries that aren't attached to bodies, you can also use Shape.update().

body
The body this shape is attached to. Can be set to None to indicate that this shape doesnt belong to a body.

cache_bb()
Update and returns the bouding box of this shape

collision_type
User defined collision type for the shape. See add_collisionpair_func function for more information on when to use this property

static create_box(*body*, *size=(10, 10)*, *offset=(0, 0)*, *radius=0*)
Convenience function to create a box centered around the body position.

The size is given as as (w,h) tuple.

elasticity
Elasticity of the shape. A value of 0.0 gives no bounce, while a value of 1.0 will give a 'perfect' bounce. However due to inaccuracies in the simulation using 1.0 or greater is not recommended.

friction
Friction coefficient. pymunk uses the Coulomb friction model, a value of 0.0 is frictionless.

A value over 1.0 is perfectly fine.

Some real world example values from wikipedia (Remember that it is what looks good that is important, not the exact value).

Material	Other	Friction
Aluminium	Steel	0.61
Copper	Steel	0.53
Brass	Steel	0.51
Cast iron	Copper	1.05
Cast iron	Zinc	0.85
Concrete (wet)	Rubber	0.30
Concrete (dry)	Rubber	1.0
Concrete	Wood	0.62
Copper	Glass	0.68
Glass	Glass	0.94
Metal	Wood	0.5
Polyethene	Steel	0.2
Steel	Steel	0.80
Steel	Teflon	0.04
Teflon (PTFE)	Teflon	0.04
Wood	Wood	0.4

get_vertices()
Get the vertices in world coordinates for the polygon

Returns [*Vec2d*] in world coords

group
Shapes in the same non-zero group do not generate collisions. Useful when creating an object out of many shapes that you don't want to self collide. Defaults to 0

layers
Shapes only collide if they are in the same bit-planes. i.e. (a.layers & b.layers) != 0. By default, a shape occupies all 32 bit-planes, i.e. layers == -1

point_query(*p*)
Check if the given point lies within the shape.

radius
The radius of the poly shape. Extends the poly in all directions with the given radius

segment_query(*start*, *end*)
Check if the line segment from start to end intersects the shape.

Return either SegmentQueryInfo object or None

sensor
A boolean value if this shape is a sensor or not. Sensors only call collision callbacks, and never generate real collisions.

surface_velocity
The surface velocity of the object. Useful for creating conveyor belts or players that move around. This value is only used when calculating friction, not resolving the collision.

unsafe_set_radius(*radius*)
Unsafe set the radius of the poly.

Note: This change is only picked up as a change to the position of the shape's surface, but not it's velocity. Changing it will not result in realistic physical behavior. Only use if you know what you are doing!

---

unsafe_set_vertices(*vertices*, *offset=(0, 0)*)
Unsafe set the vertices of the poly.

Note: This change is only picked up as a change to the position of the shape's surface, but not it's velocity. Changing it will not result in realistic physical behavior. Only use if you know what you are doing!

---

update(*position*, *rotation_vector*)
Update, cache and return the bounding box of a shape with an explicit transformation.

Useful if you have a shape without a body and want to use it for querying.

class pymunk.**Segment**(*body*, *a*, *b*, *radius*)
Bases: pymunk.Shape

A line segment shape between two points

This shape can be attached to moving bodies, but don't currently generate collisions with other line segments. Can be beveled in order to give it a thickness.

It is legal to send in None as body argument to indicate that this shape is not attached to a body.

__init__(*body*, *a*, *b*, *radius*)
Create a Segment

Parameters

body [*Body*] The body to attach the segment to

a [(x,y) or *Vec2d*] The first endpoint of the segment

b [(x,y) or *Vec2d*] The second endpoint of the segment

radius [float] The thickness of the segment

a
The first of the two endpoints for this segment

b
The second of the two endpoints for this segment

bb
The bounding box of the shape. Only guaranteed to be valid after Shape.cache_bb() or Space.step() is

called. Moving a body that a shape is connected to does not update it's bounding box. For shapes used for queries that aren't attached to bodies, you can also use Shape.update().

body
The body this shape is attached to. Can be set to None to indicate that this shape doesnt belong to a body.

cache_bb()
Update and returns the bouding box of this shape

collision_type
User defined collision type for the shape. See add_collisionpair_func function for more information on when to use this property

elasticity
Elasticity of the shape. A value of 0.0 gives no bounce, while a value of 1.0 will give a 'perfect' bounce. However due to inaccuracies in the simulation using 1.0 or greater is not recommended.

friction
Friction coefficient. pymunk uses the Coulomb friction model, a value of 0.0 is frictionless.

A value over 1.0 is perfectly fine.

Some real world example values from wikipedia (Remember that it is what looks good that is important, not the exact value).

Material	Other	Friction
Aluminium	Steel	0.61
Copper	Steel	0.53
Brass	Steel	0.51
Cast iron	Copper	1.05
Cast iron	Zinc	0.85
Concrete (wet)	Rubber	0.30
Concrete (dry)	Rubber	1.0
Concrete	Wood	0.62
Copper	Glass	0.68
Glass	Glass	0.94
Metal	Wood	0.5
Polyethene	Steel	0.2
Steel	Steel	0.80
Steel	Teflon	0.04
Teflon (PTFE)	Teflon	0.04
Wood	Wood	0.4

group
Shapes in the same non-zero group do not generate collisions. Useful when creating an object out of many shapes that you don't want to self collide. Defaults to 0

layers
Shapes only collide if they are in the same bit-planes. i.e. (a.layers & b.layers) != 0. By default, a shape occupies all 32 bit-planes, i.e. layers == -1

point_query(*p*)
Check if the given point lies within the shape.

radius
The radius/thickness of the segment

segment_query(*start*, *end*)
Check if the line segment from start to end intersects the shape.

Return either SegmentQueryInfo object or None

sensor
A boolean value if this shape is a sensor or not. Sensors only call collision callbacks, and never generate real collisions.

surface_velocity
The surface velocity of the object. Useful for creating conveyor belts or players that move around. This value is only used when calculating friction, not resolving the collision.

unsafe_set_a(*a*)
Set the first of the two endpoints for this segment

Note: This change is only picked up as a change to the position of the shape's surface, but not it's velocity. Changing it will not result in realistic physical behavior. Only use if you know what you are doing!

---

unsafe_set_b(*b*)
Set the second of the two endpoints for this segment

Note: This change is only picked up as a change to the position of the shape's surface, but not it's velocity. Changing it will not result in realistic physical behavior. Only use if you know what you are doing!

---

unsafe_set_radius(*r*)
Set the radius of the segment

Note: This change is only picked up as a change to the position of the shape's surface, but not it's velocity. Changing it will not result in realistic physical behavior. Only use if you know what you are doing!

---

update(*position*, *rotation_vector*)
Update, cache and return the bounding box of a shape with an explicit transformation.

Useful if you have a shape without a body and want to use it for querying.

pymunk.**moment_for_circle**(*mass*, *inner_radius*, *outer_radius*, *offset=(0, 0)*)
Calculate the moment of inertia for a circle

pymunk.**moment_for_poly**(*mass*, *vertices*, *offset=(0, 0)*)
Calculate the moment of inertia for a polygon

pymunk.**moment_for_segment**(*mass*, *a*, *b*)
Calculate the moment of inertia for a segment

pymunk.**moment_for_box**(*mass*, *width*, *height*)
Calculate the momentn of inertia for a box

pymunk.**reset_shapeid_counter**()
Reset the internal shape counter

pymunk keeps a counter so that every new shape is given a unique hash value to be used in the spatial hash. Because this affects the order in which the collisions are found and handled, you should reset the shape counter every time you populate a space with new shapes. If you don't, there might be (very) slight differences in the simulation.

class pymunk.**SegmentQueryInfo**(*shape*, *start*, *end*, *t*, *n*)
Bases: object

Segment queries return more information than just a simple yes or no, they also return where a shape was hit and it's surface normal at the hit point. This object hold that information.

__init__(*shape*, *start*, *end*, *t*, *n*)
You shouldn't need to initialize SegmentQueryInfo objects on your own.

get_hit_distance()
Return the absolute distance where the segment first hit the shape

get_hit_point()
Return the hit point in world coordinates where the segment first intersected with the shape

n
Normal of hit surface

shape
Shape that was hit

t
Distance along query segment, will always be in the range [0, 1]

class pymunk.**Contact**(*_contact*)
Bases: object

Contact information

__init__(*_contact*)
Initialize a Contact object from the Chipmunk equivalent struct

---

Note: You should never need to create an instance of this class directly.

---

distance
Penetration distance

normal
Contact normal

position
Contact position

class pymunk.**Arbiter**(*_arbiter*, *space*)
Bases: object

Arbiters are collision pairs between shapes that are used with the collision callbacks.

Warning: Because arbiters are handled by the space you should never hold onto a reference to an arbiter as you don't know when it will be destroyed! Use them within the callback where they are given to you and then forget about them or copy out the information you need from them.

__init__(*_arbiter*, *space*)
Initialize an Arbiter object from the Chipmunk equivalent struct and the Space.

Note: You should never need to create an instance of this class directly.

---

contacts
Information on the contact points between the objects. Return [*Contact*]

elasticity
Elasticity

friction
Friction

is_first_contact
Returns true if this is the first step that an arbiter existed. You can use this from preSolve and postSolve to know if a collision between two shapes is new without needing to flag a boolean in your begin callback.

shapes
Get the shapes in the order that they were defined in the collision handler associated with this arbiter

stamp
Time stamp of the arbiter. (from the space)

surface_velocity
Used for surface_v calculations, implementation may change

total_impulse
Returns the impulse that was applied this step to resolve the collision.

This property should only be called from a post-solve, post-step

total_impulse_with_friction
Returns the impulse with friction that was applied this step to resolve the collision.

This property should only be called from a post-solve, post-step

total_ke
The amount of energy lost in a collision including static, but not dynamic friction.

This property should only be called from a post-solve, post-step

class pymunk.**BB**(**args*)
Bases: object

Simple bounding box class. Stored as left, bottom, right, top values.

__init__(**args*)
Create a new instance of a bounding box. Can be created with zero size with bb = BB() or with four args defining left, bottom, right and top: bb = BB(left, bottom, right, top)

bottom

clamp_vect(*v*)
Returns a copy of the vector v clamped to the bounding box

contains(*other*)
Returns true if bb completley contains the other bb

contains_vect(*v*)
Returns true if this bb contains the vector v

expand(*v*)
Return the minimal bounding box that contans both this bounding box and the vector v

intersects(*other*)
Returns true if the bounding boxes intersect

left

merge(*other*)
Return the minimal bounding box that contains both this bb and the other bb

right

top

wrap_vect(*v*)
Returns a copy of v wrapped to the bounding box. That is, BB(0,0,10,10).wrap_vect((5,5)) == Vec2d(10,10)

5.5 Examples

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

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

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

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

To run directly without installing anything. From the pymunk source folder:

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

Each example contains something unique. Not all of the examples use the same style. For example, some use the pymunk.pygame_util module to draw stuff, others contain the actual drawing code themselfs. However, each example is self contained. Except for external libraries (such as pygame) and pymunk 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 pymunk and would like it displayed here or in a showcase section of the site, feel free to contact me!

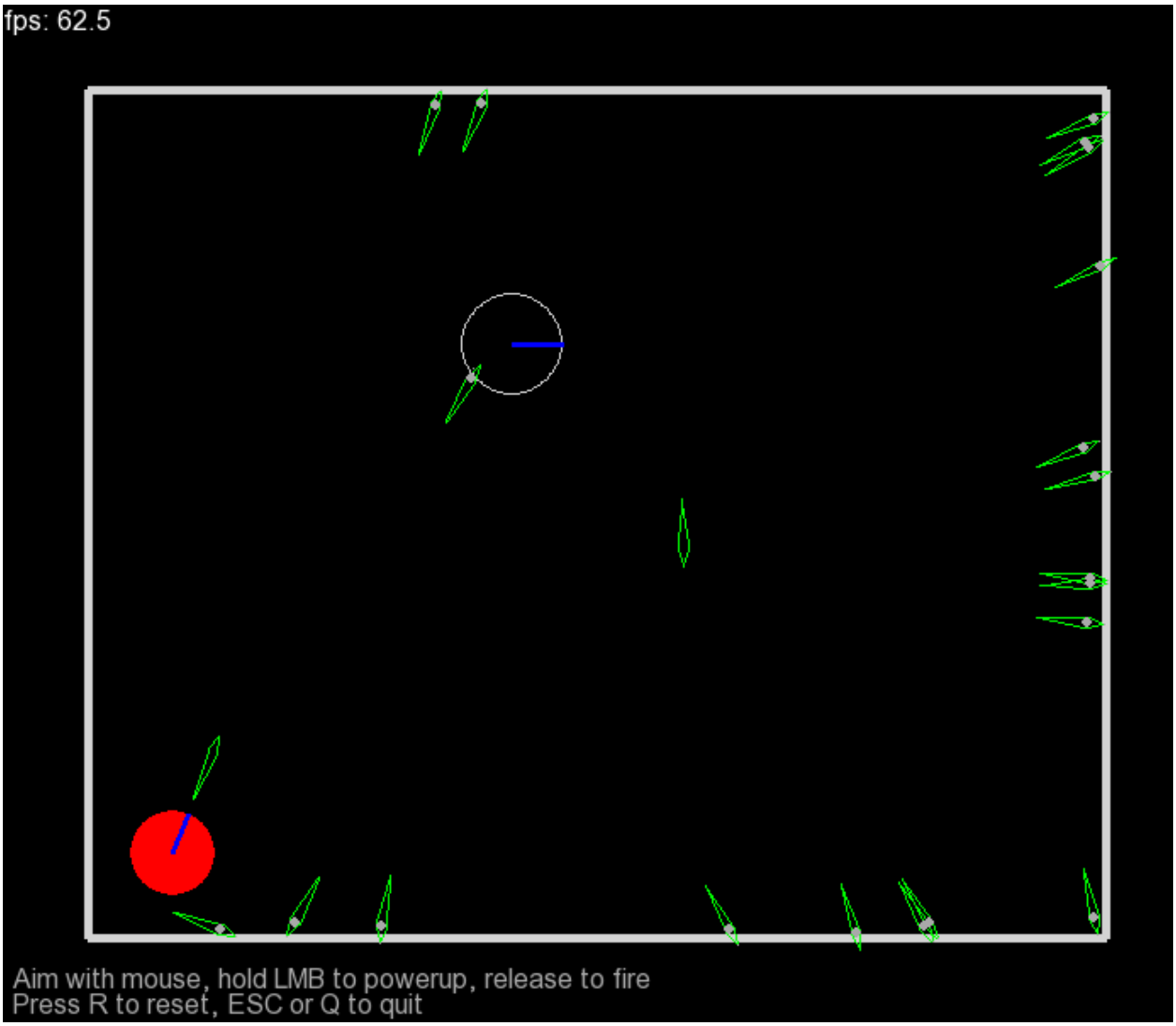
Example files

- arrows.py
- contact_with_friction.py
- basic_test.py
- damped_rotary_spring_pointer.py
- no_debug.py
- spiderweb.py
- pygame_util_demo.py
- bouncing_balls.py
- py2exe_setup__breakout.py
- playground.py
- balls_and_lines.py
- newtons_cradle.py
- flipper.py
- pyglet_util_demo.py
- contact_and_no_flipy.py
- box2d_pyramid.py
- shapes_for_draw_demos.py
- run.py
- slide_and_pinjoint.py
- point_query.py
- breakout.py
- platformer.py
- using_sprites.py
- using_sprites_pyglet.py
- polygon_triangulation.py
- py2exe_setup__basic_test.py
- box2d_vertical_stack.py

5.5.1 arrows.py

Location: *examples/arrows.py*

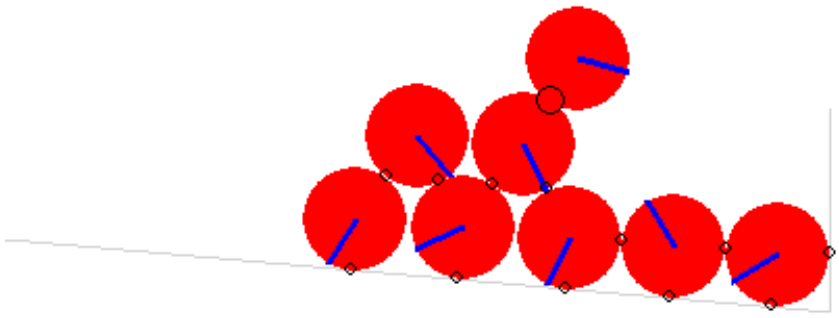
Showcase of flying arrows that can stick to objects in a somewhat realistic looking way.



5.5.2 contact_with_friction.py

Location: *examples/contact_with_friction.py*

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



5.5.3 basic_test.py

Location: *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.

5.5.4 damped_rotary_spring_pointer.py

Location: *examples/damped_rotary_spring_pointer.py*

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

5.5.5 no_debug.py

Location: *examples/no_debug.py*

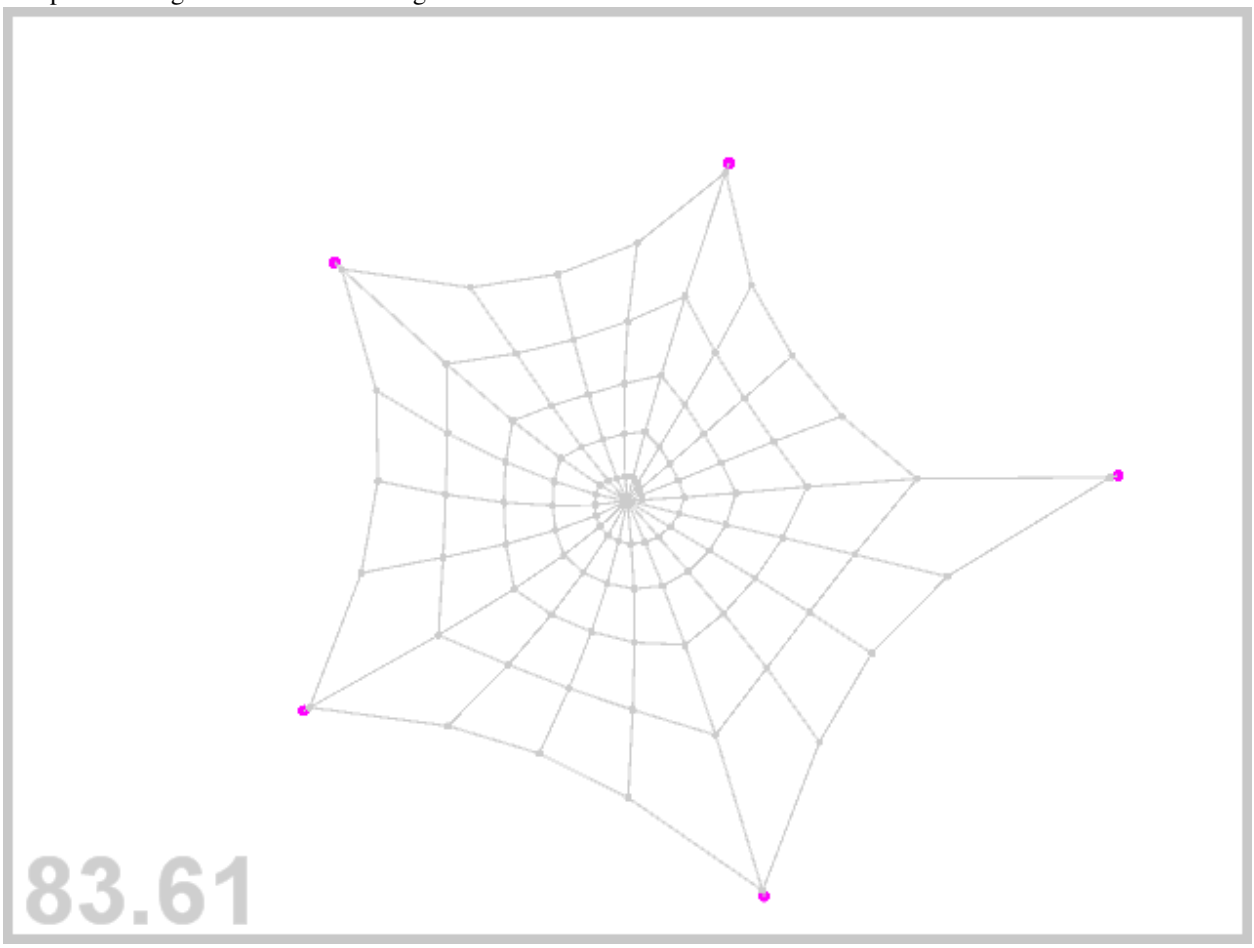
Very simple showcase of how to run pymunk with debug mode off

5.5.6 spiderweb.py

Location: *examples/spiderweb.py*

Showcase of a elastic spiderweb (drawing with pyglet)

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

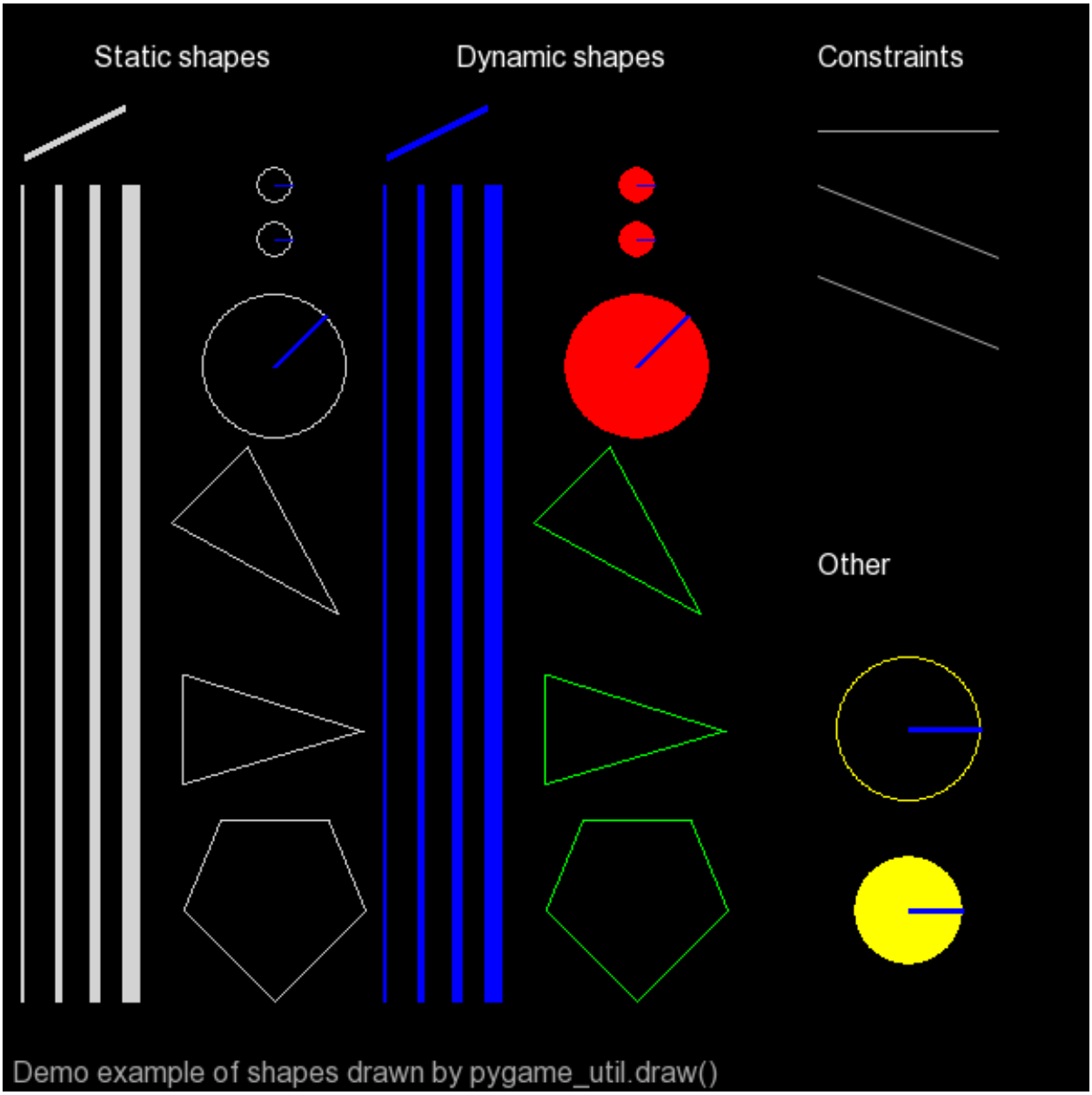


5.5.7 pygame_util_demo.py

Location: *examples/pygame_util_demo.py*

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

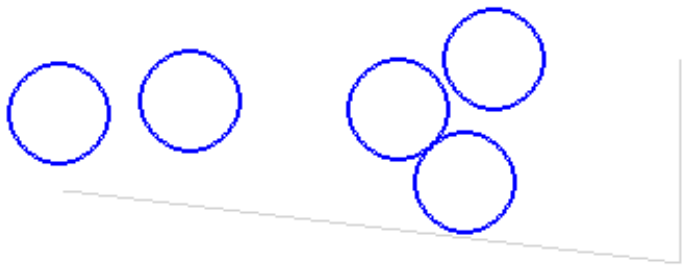
See pyglet_util_demo.py for a comparison to pyglet.



5.5.8 bouncing_balls.py

Location: *examples/bouncing_balls.py*

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



5.5.9 py2exe_setup__breakout.py

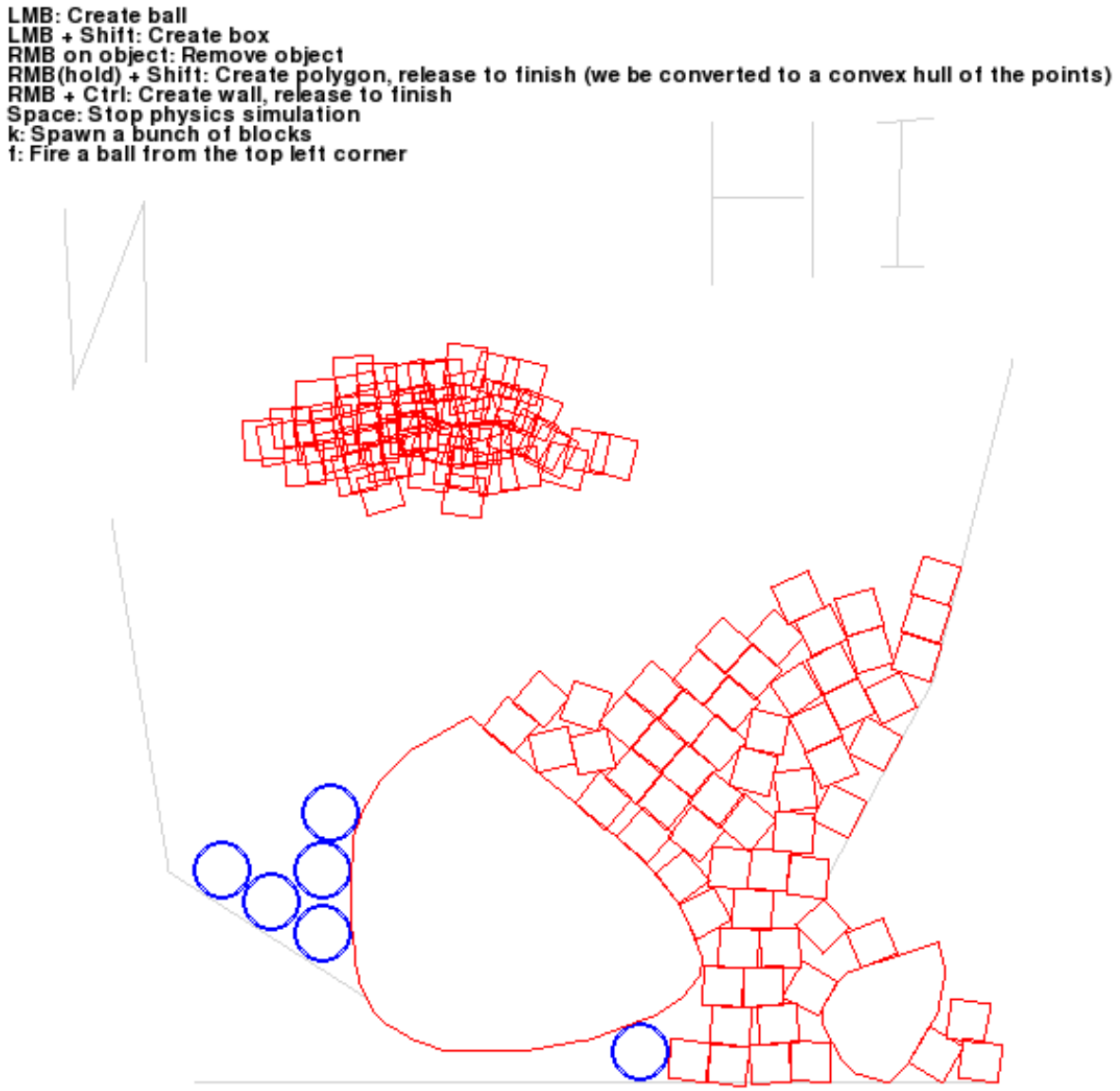
Location: *examples/py2exe_setup__breakout.py*

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

5.5.10 playground.py

Location: *examples/playground.py*

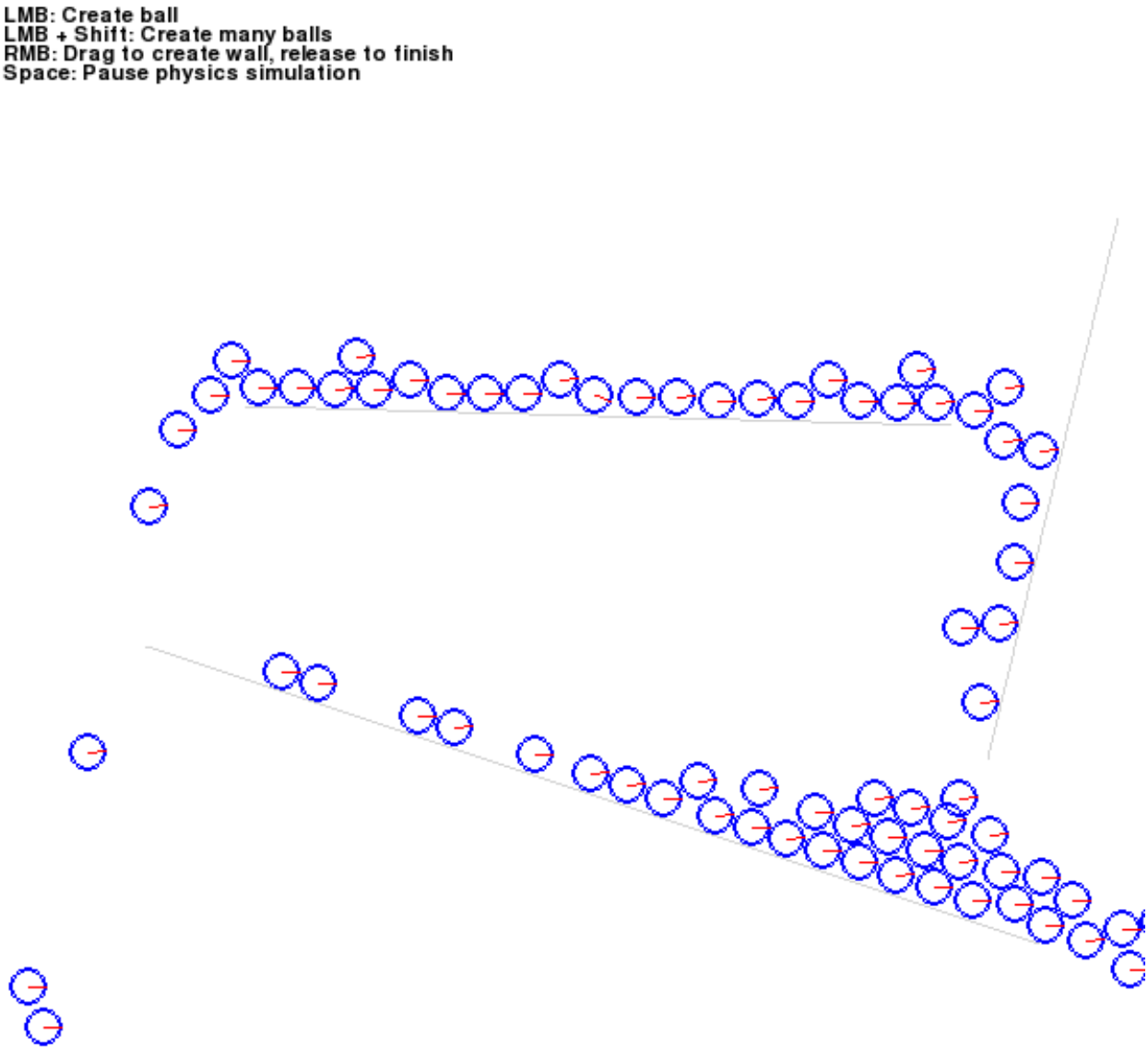
A basic playground. Most interesting function is draw a shape, basically move the mouse as you want and pymunk will approximate a Poly shape from the drawing.



5.5.11 balls_and_lines.py

Location: *examples/balls_and_lines.py*

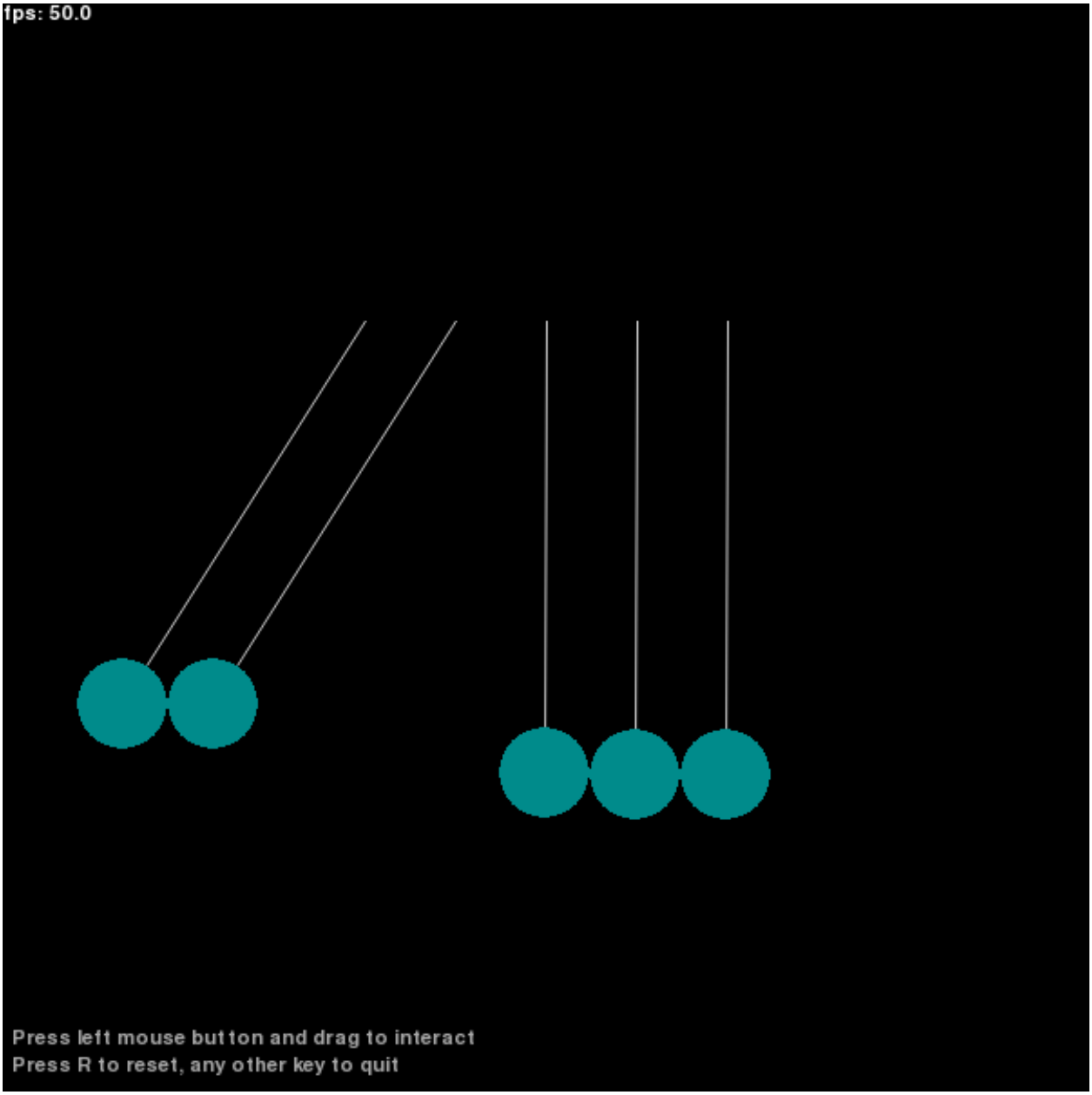
This example lets you dynamically create static walls and dynamic balls



5.5.12 newtons_cradle.py

Location: *examples/newtons_cradle.py*

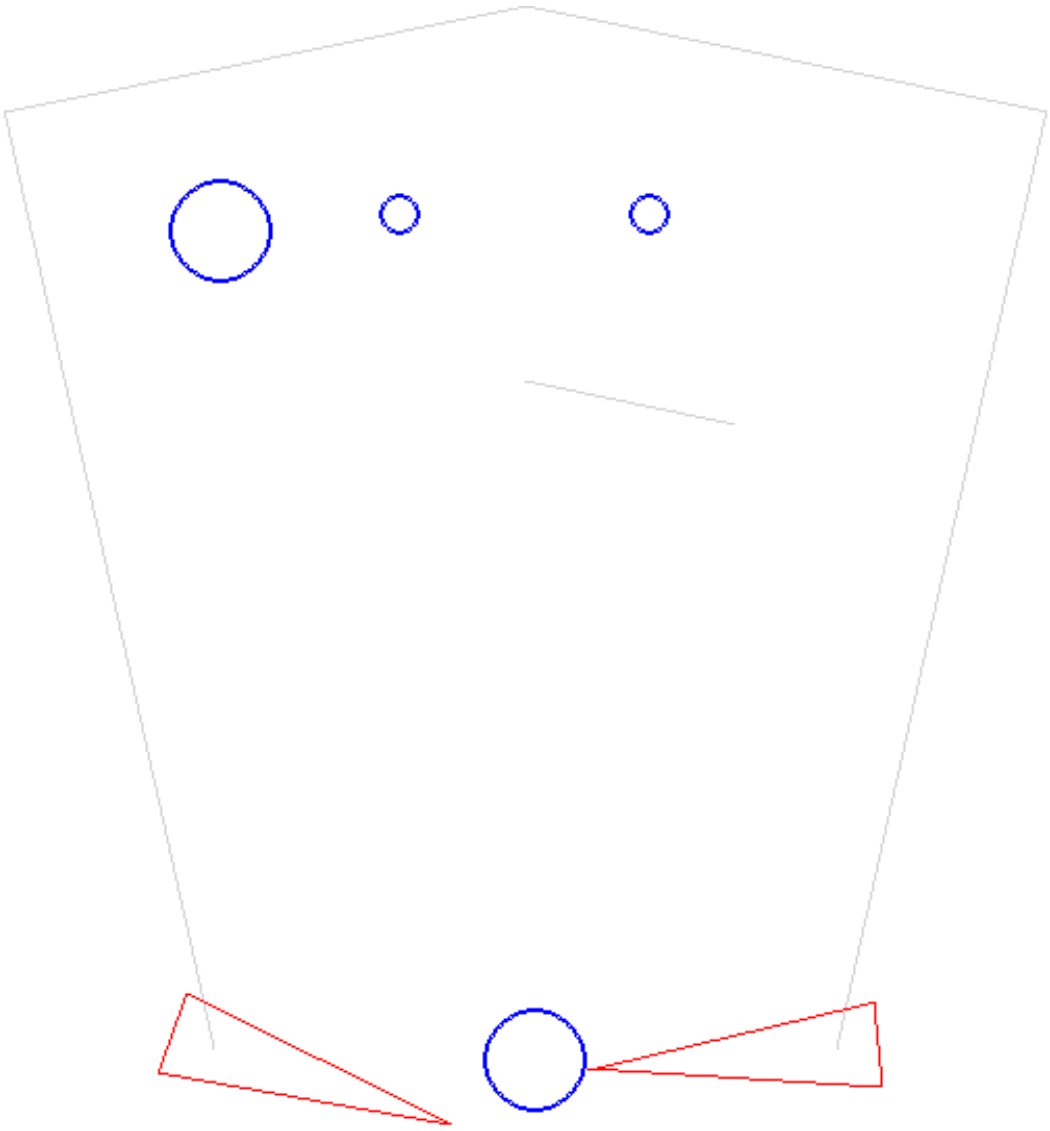
A screensaver version of Newton's Cradle with an interactive mode.



5.5.13 flipper.py

Location: *examples/flipper.py*

A very basic flipper game.

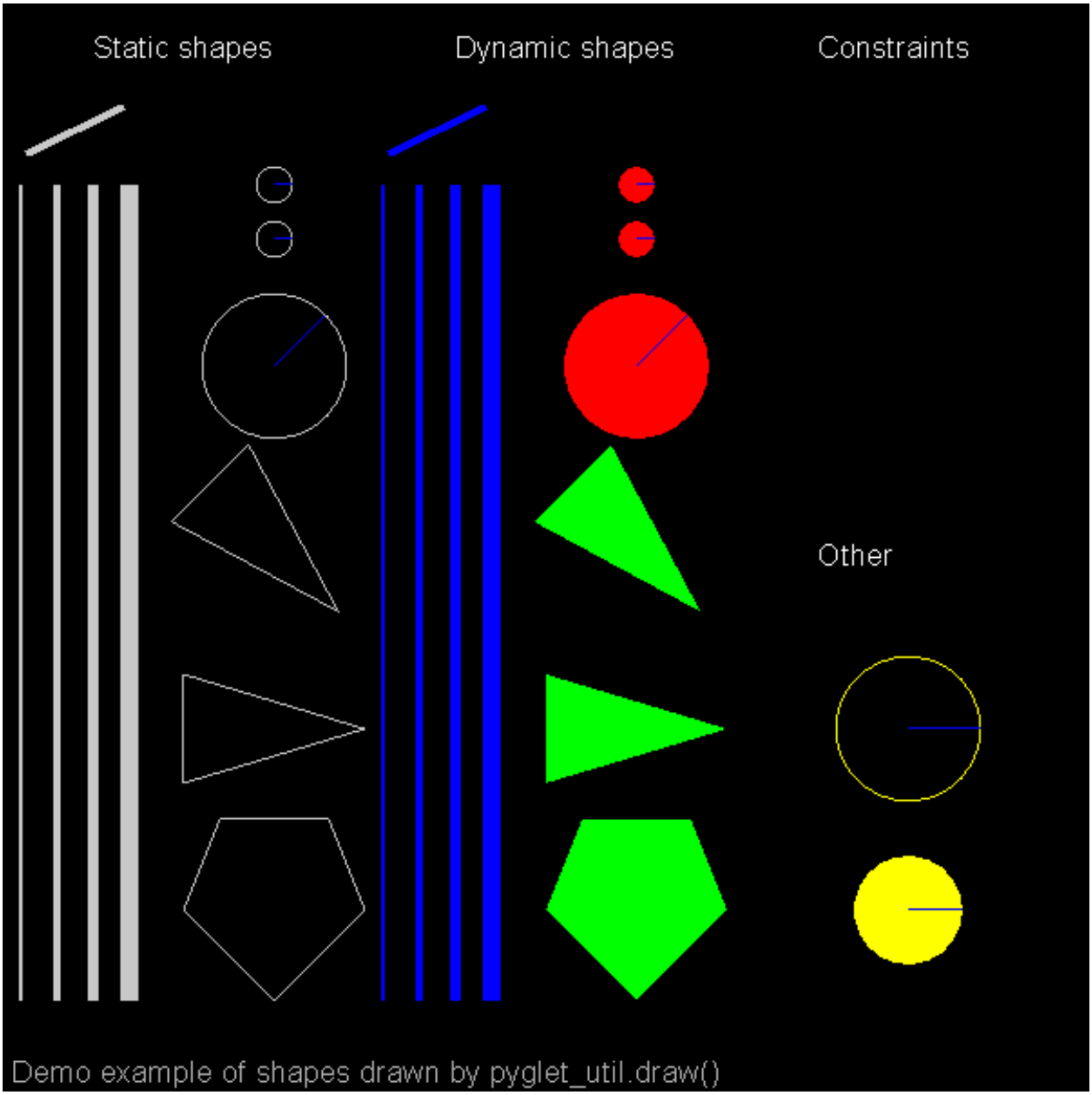


5.5.14 pyglet_util_demo.py

Location: *examples/pyglet_util_demo.py*

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

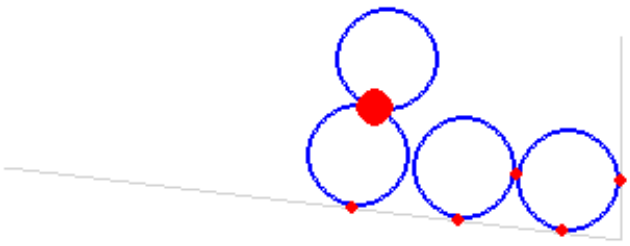
See pygame_util_demo.py for a comparison to pygame.



5.5.15 contact_and_no_flipy.py

Location: *examples/contact_and_no_flipy.py*

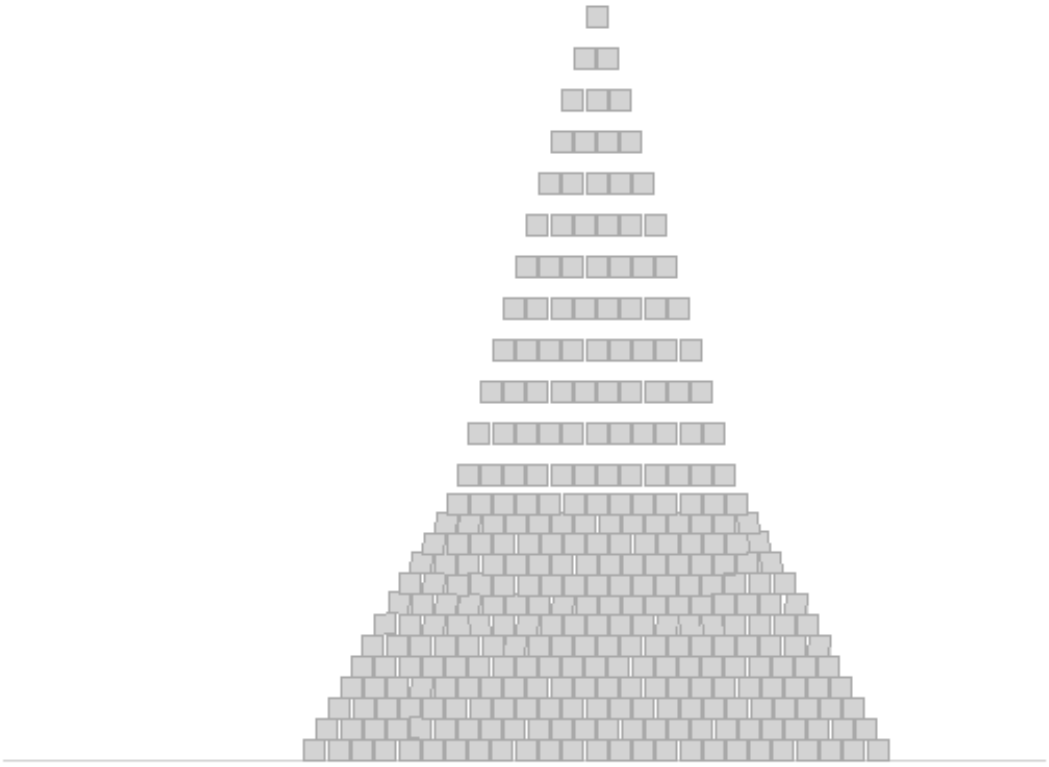
This example spawns (bouncing) balls randomly on a L-shape constructed of two segment shapes. For each collision it draws a red circle with size depending on collision strength. Not interactive.



5.5.16 box2d_pyramid.py

Location: *examples/box2d_pyramid.py*

Remake of the pyramid demo from the box2d testbed.



5.5.17 shapes_for_draw_demos.py

Location: *examples/shapes_for_draw_demos.py*

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

5.5.18 run.py

Location: *examples/run.py*

Use to run examples using pymunk located one folder level up. Useful if you have the whole pymunk source tree and want to run the examples in a quick and dirty way. (a poor man's virtualenv if you like)

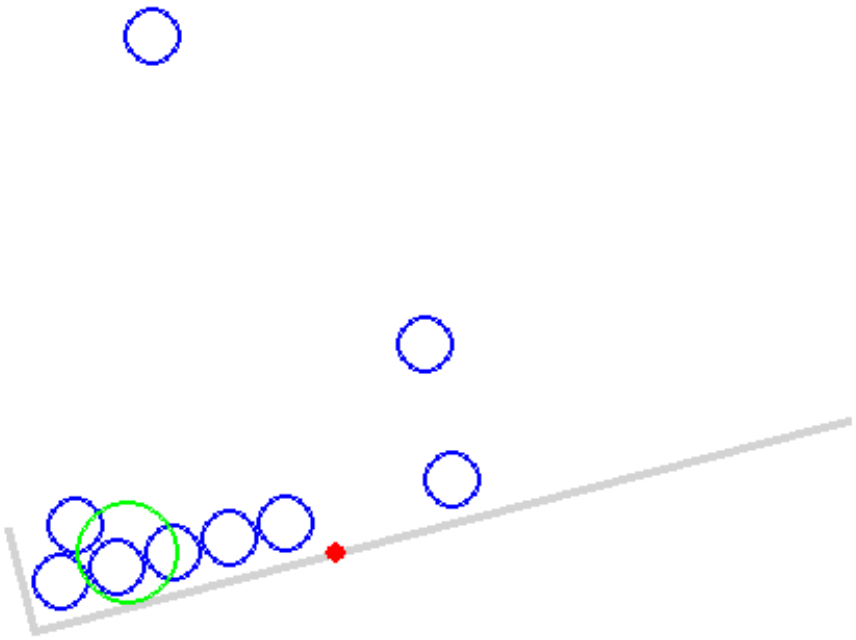
For example, to run the breakout demo:

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

5.5.19 slide_and_pinjoint.py

Location: *examples/slide_and_pinjoint.py*

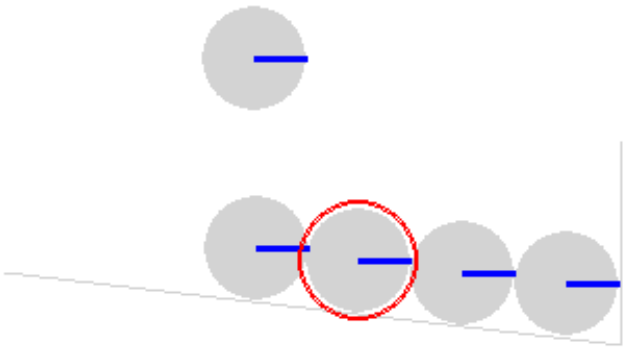
A L shape attached with a joint and constrained to not tip over.



5.5.20 point_query.py

Location: *examples/point_query.py*

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

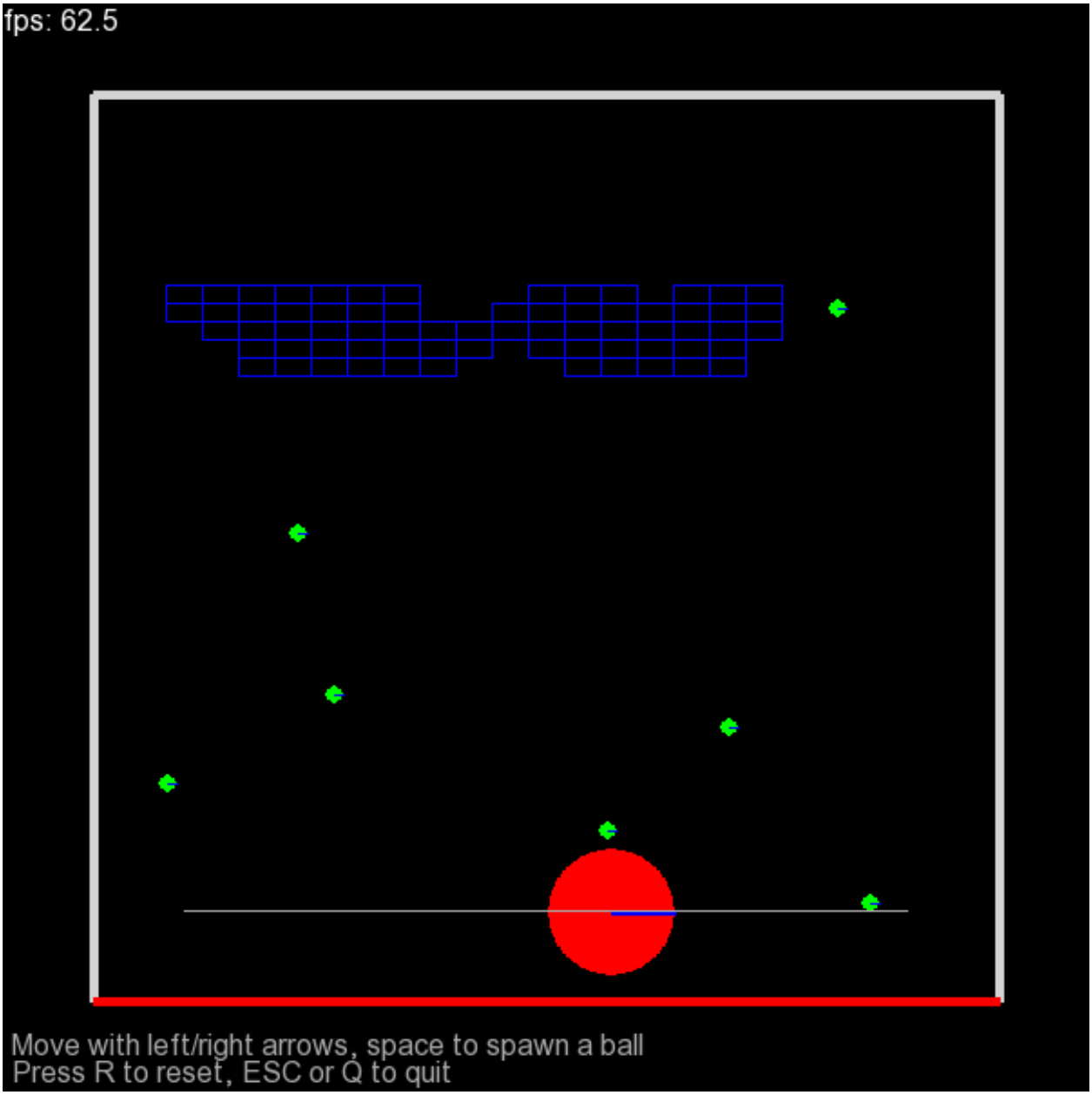


5.5.21 breakout.py

Location: *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 elasitcity, impulses, constant object speed, joints, collision handlers and post step callbacks.



5.5.22 platformer.py

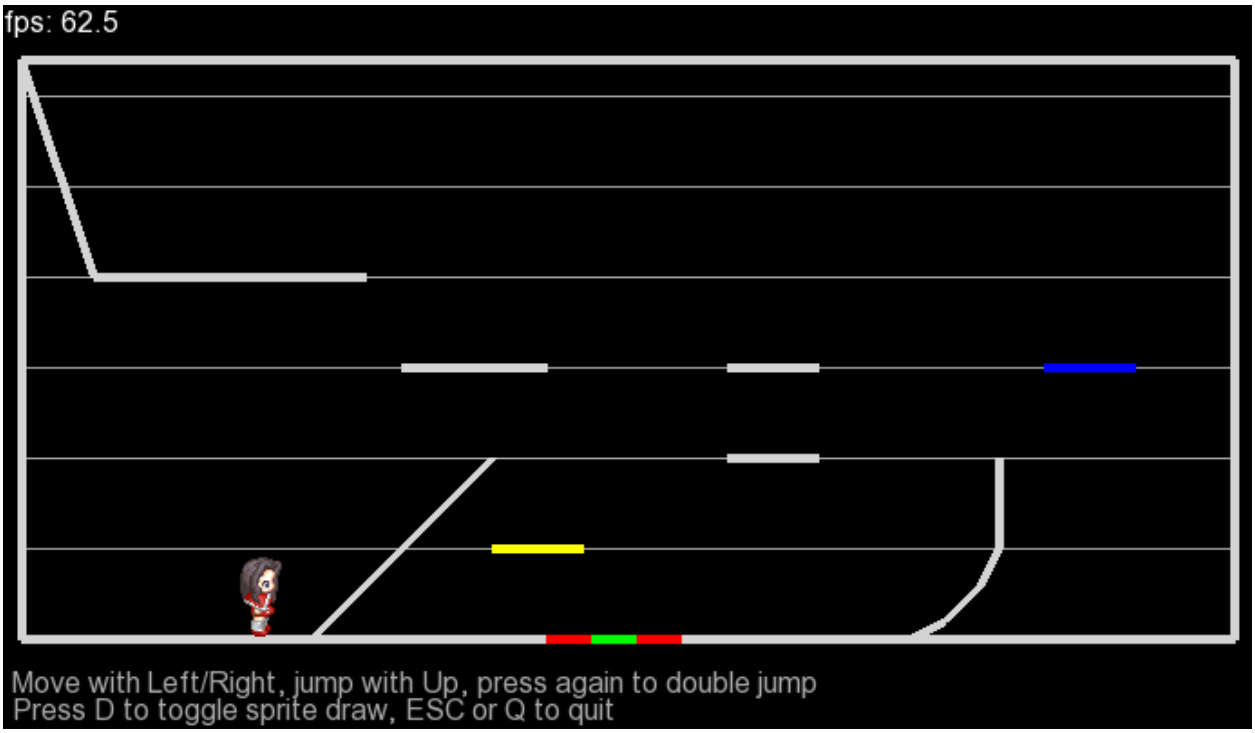
Location: *examples/platformer.py*

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

---

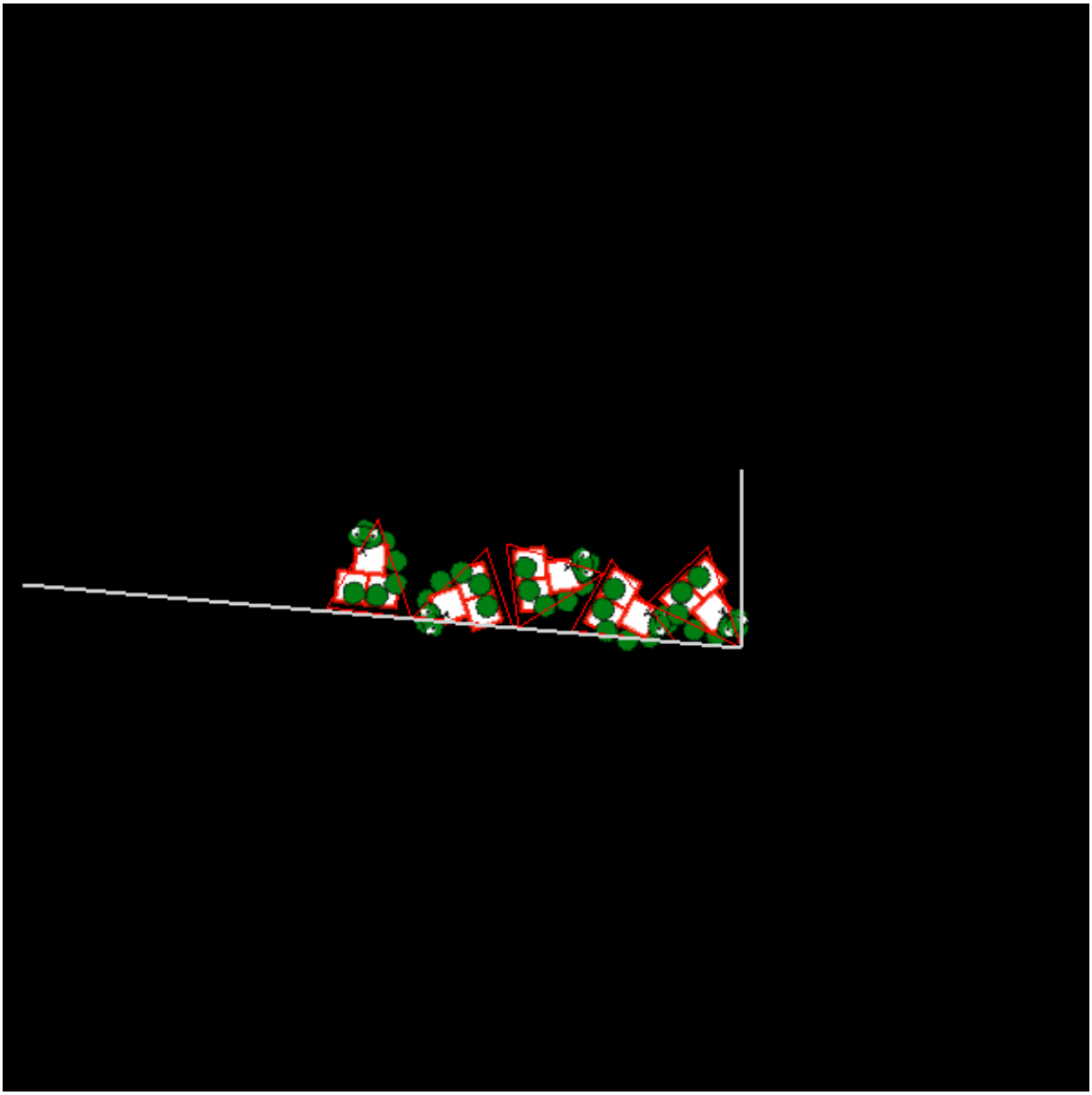
Note: The code of this example is a bit messy. If you adapt this to your own code you might want to structure it a bit differently.



5.5.23 using_sprites.py

Location: *examples/using_sprites.py*

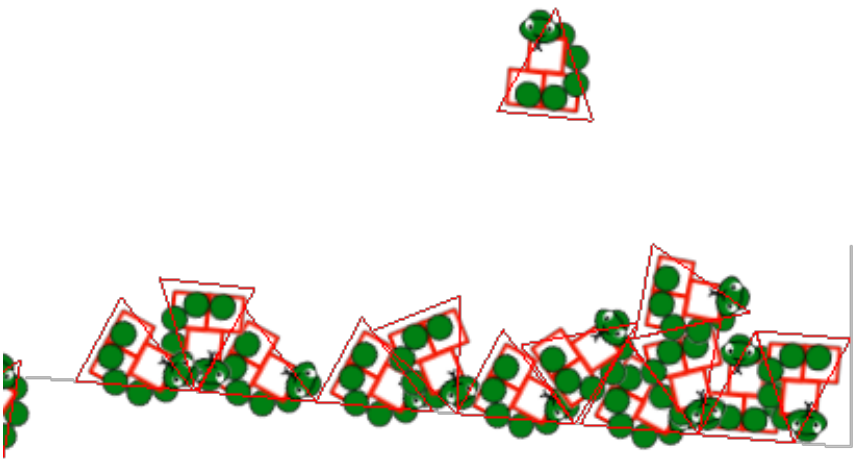
Very basic example of using a sprite image to draw a shape more similar how you would do it in a real game instead of the simple line drawings used by the other examples.



5.5.24 using_sprites_pyglet.py

Location: *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.



5.5.25 polygon_triangulation.py

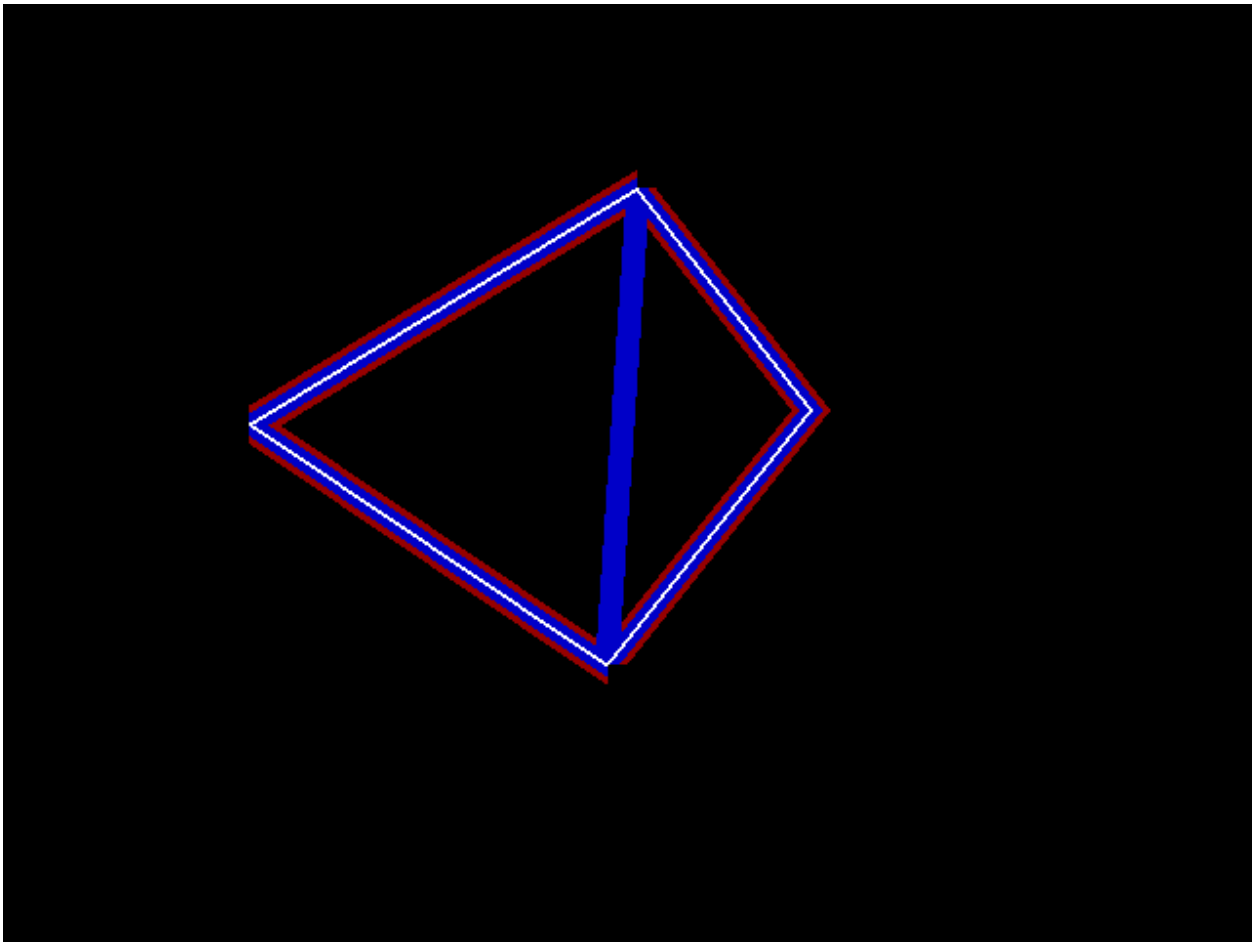
Location: *examples/polygon_triangulation.py*

Quick demo of using triangulate.py to triangulate/convexise(?) a concave polygon. Not good code as such, but functional and cheap

display: thick red line: drawn polygon medium blue lines: triangles after triangulation thin white lines: convex polygons after convexisation(?)

input: click points (in clockwise order)* to draw a polygon press space to reset

- triangulate() and convexise() actually work on anticlockwise polys to match pymunk, but this demo's coords are upside-down compared to pymunk (pygame style), so click clockwise to compensate :)



5.5.26 py2exe_setup__basic_test.py

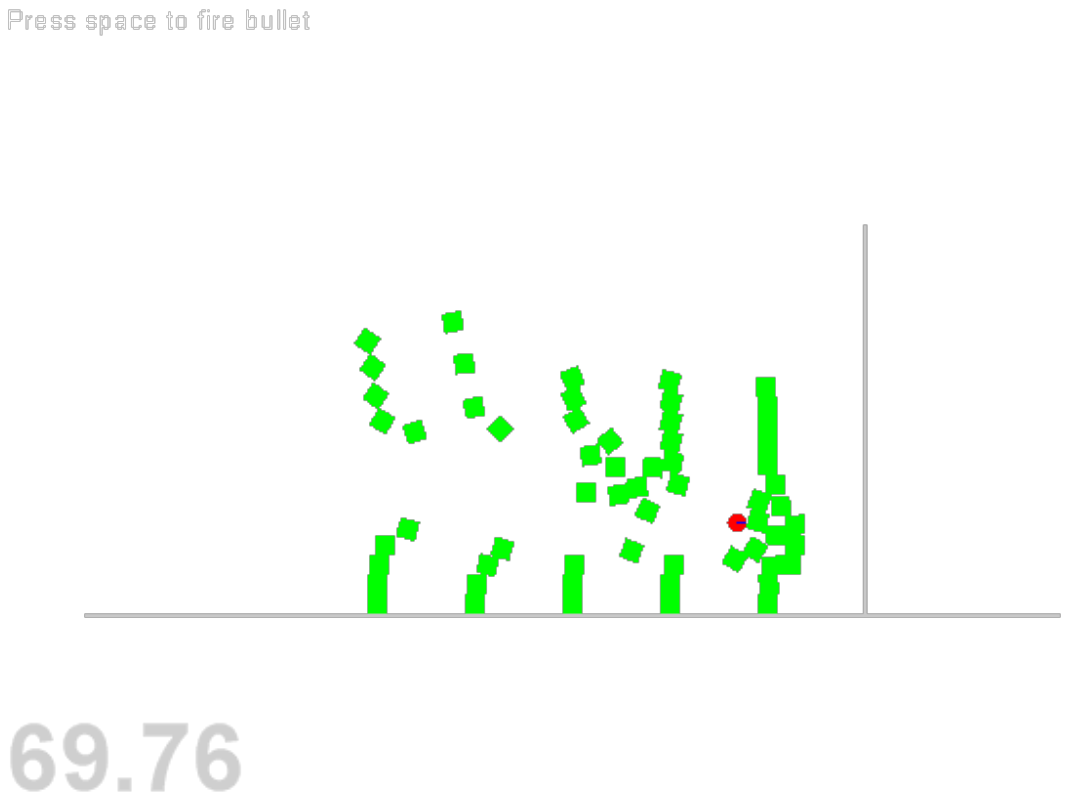
Location: *examples/py2exe_setup__basic_test.py*

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

5.5.27 box2d_vertical_stack.py

Location: *examples/box2d_vertical_stack.py*

Remake of the veritcal stack demo from the box2d testbed.



5.6 Tutorials

List of included tutorials. 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 pymunk.

It is a good idea to start with the Slide and Pin Joint tutorial. It will walk you through the basics for a simple simulation.

If you have made a tutorial that is using pymunk in any way and want it mentioned here please send me a link and I will happily link 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.

5.6.1 Arrows and other pointy sticky things

Attention: This tutorial is work in progress. Take a look at the arrows.py example file in *Examples* for fully working code for the arrows.

This tutorial will explain one way to make arrows/sticky projectiles that can stick on the target.

The tutorial is heavily inspired by the Box2d tutorial "Box2D C++ tutorials - Sticky projectiles" found here http://www.iforce2d.net/b2dtut/sticky-projectiles (but adjusted for python, pymunk and chipmunk).

Before we start

For this tutorial you will need to know some pymunk basics. I recommend that you read the other tutorial(s) and try out easier examples first before you continue.

Except for pymunk you will also need pygame to follow this tutorial. However, it should be no problem to use another graphics and input library if you want, for example pyglet.

We will try to accomplish

- An arrow that flies believable in the air
- Figure out when the arrow hits something and should stick
- Attach the arrow to an object when hit

In the end we should have a cannon like object shooting arrows that flies in a believable way and sticks to objects if they hit hard enough.

Basic scene

Before we start with the arrow we need a scene to contain it and a "cannon" that can aim:

```
import sys

import pygame
from pygame.locals import *
from pygame.color import *

import pymunk
from pymunk.vec2d import Vec2d
from pymunk.pygame_draw import draw_space, from_pygame

width = height = 600
def main():
    ### PyGame init
    pygame.init()
    screen = pygame.display.set_mode((width,height))
    clock = pygame.time.Clock()
    running = True
    font = pygame.font.SysFont("Arial", 16)

    ### Physics stuff
    space = pymunk.Space()

    # walls - the left-top-right-bottom walls
    static_lines = [pymunk.Segment(space.static_body, (50, 50), (50, 550), 5)
                ,pymunk.Segment(space.static_body, (50, 550), (550, 550), 5)
                ,pymunk.Segment(space.static_body, (550, 550), (550, 50), 5)
                ,pymunk.Segment(space.static_body, (50, 50), (550, 50), 5)
                ]
    space.add_static(static_lines)

    ### "Cannon" that can fire arrows
    cannon_body = pymunk.Body()
    player_shape = pymunk.Circle(cannon_body, 25)
    cannon_body.position = 100,100

    space.add(player_shape)
```

```
    while running:
        for event in pygame.event.get():
            if event.type == QUIT or \
                event.type == KEYDOWN and (event.key in [K_ESCAPE, K_q]):
                running = False

        mpos = pygame.mouse.get_pos()
        p = from_pygame( Vec2d(mpos), screen )
        mouse_position = p
        cannon_body.angle = (mouse_position - cannon_body.position).angle

        ### Clear screen
        screen.fill(pygame.color.THECOLORS["black"])

        ### Draw stuff
        draw_space(screen, space)

        ### Update physics
        fps = 60
        dt = 1./fps
        space.step(dt)

        ### Info and flip screen
        screen.blit(font.render("fps: " + str(clock.get_fps()), 1, THECOLORS["white"]), (0,0))
        screen.blit(font.render("Aim with mouse, click to fire", 1, THECOLORS["darkgrey"]), (5,height
        screen.blit(font.render("Press R to reset, ESC or Q to quit", 1, THECOLORS["darkgrey"]), (5,h

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

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

5.6.2 Slide and Pin Joint Demo Step by Step

This is a step by step tutorial explaining the demo demo_slide_and_pinjoint.py included in pymunk. 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.

Before we start

For this tutorial you will need:

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

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

Pymunk is built on top of the 2d physics library Chipmunk. Chipmunk itself is written in C meaning pymunk need to call into the c code. The ctypes 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!

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

```
>>> import pymunk
```

If you get an error message it usually is because pymunk could not find the chipmunk library because it was not compiled (should not happen on windows and ubuntu, as pymunk ships with the code compiled for those two). To compile chipmunk, do

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

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*

An empty simulation

Ok, lets start. Chipmunk (and therefore pymunk) 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 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 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 pymunk documentation.

The API documentation for pymunk can be found here: *API Reference*.

Anyway, we are now ready to write some code:

```
import sys
import pygame
from pygame.locals import *
from pygame.color import *
import pymunk #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()
    running = True

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

    while running:
        for event in pygame.event.get():
            if event.type == QUIT:
                running = False
```

```
            elif event.type == KEYDOWN and event.key == K_ESCAPE:
                running = False

        screen.fill(THECOLORS["white"])

        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 pymunk in order to use it...

#. 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.

#. In our game loop we call the step() function on our space. The step function steps the simulation one step forward in time. Note: It is best to keep the stepsize constant and not adjust it depending on the framrate. The physic simulation will work much better with a constant step size.

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 most 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 = 1
    radius = 14
    inertia = pymunk.moment_for_circle(mass, 0, radius) # 1
    body = pymunk.Body(mass, inertia) # 2
    x = random.randint(120,380)
    body.position = x, 550 # 3
    shape = pymunk.Circle(body, radius) # 4
    space.add(body, shape) # 5
    return shape
```

#. All bodies must have their moment of inertia set. If our object is a normal ball we can use the predefined function moment_for_circle to calculate it given its mass and radius. However, you could also select a value by experimenting with what looks good for your simulation.

1. After we have the inertia we can create the body of the ball.
2. And we set its position

#. And in order for it to collide with things, it needs to have one (or many) collision shape(s).

#. Finally we add the body and shape to the space to include it in our simulation.

Now that we can create balls we want to display them:

```
def draw_ball(screen, ball):
    p = int(ball.body.position.x), 600-int(ball.body.position.y)
    pygame.draw.circle(screen, THECOLORS["blue"], p, int(ball.radius), 2)
```

As I have used pygame in this example, we can use the draw.circle function to draw the balls. But first we must convert the position of the ball. We earlier set the gravity to -900 (that is, it will point down the y axis). Pygame thinks 0,0 is at the top left of the screen, with y increasing downwards. So we need to make a simple conversion of the y value.

An alternative way to handle the display would have been to use the pygame_util.draw_space function that is included in pymunk. However, in this tutorial I wanted to show how to draw things yourself as you are likley to want that in your own code anyway at some point.

With these two functions and a little code to spawn balls you should see a couple of balls falling. Yay!

```
import sys, random
import pygame
from pygame.locals import *
from pygame.color import *
import pymunk

#def add_ball(space):
#def draw_ball(screen, ball):

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()
    running = True

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

    balls = []

    ticks_to_next_ball = 10
    while running:
        for event in pygame.event.get():
            if event.type == QUIT:
                running = False
            elif event.type == KEYDOWN and event.key == K_ESCAPE:
                running = False

        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(THECOLORS["white"])

        for ball in balls:
            draw_ball(screen, ball)

        space.step(1/50.0)

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

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

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 = pymunk.Body() # 1
    body.position = (300, 300)
    l1 = pymunk.Segment(body, (-150, 0), (255, 0), 5) # 2
    l2 = pymunk.Segment(body, (-150, 0), (-150, 50), 5)

    space.add(l1, l2) # 3
    return l1,l2
```

1. We create a "static" body. The important step is to never add it to the space. Note how static bodies are created by not passing any arguments to the Body constructor.
2. A line shaped shape is created here.
3. Again, we only add the segments, not the body to the space.

Next we add a function to draw the L shape:

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

#. 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, so we don't really have to do this extra calculation, but we will soon make them move and rotate.

#. This is a little function to convert coordinates from pymunk to pygame world. Now that we have it we can use it in the draw_ball() function as well. We want to flip the y coordinate (-p.y), and then offset it with the screen height (+600). It looks like this:

```
def to_pygame(p):
    """Small hack to convert pymunk to pygame coordinates"""
    return int(p.x), int(-p.y+600)
```

We add a call to add_static_L() and one to draw_lines() and now we should see an inverted L shape in the middle will balls spawning and hitting the shape.

```
import sys, random
import pygame
from pygame.locals import *
from pygame.color import *
import pymunk as pm
import math

#def to_pygame(p):
#def add_ball(space):
#def draw_ball(screen, ball):
#def add_static_l(space):
#def draw_lines(screen, lines):
```

```
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()
    running = True

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

    lines = add_static_L(space)
    balls = []

    ticks_to_next_ball = 10
    while running:
        for event in pygame.event.get():
            if event.type == QUIT:
                running = False
            elif event.type == KEYDOWN and event.key == K_ESCAPE:
                running = False

        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(THECOLORS["white"])

        for ball in balls:
            draw_ball(screen, ball)

        draw_lines(screen, lines)

        space.step(1/50.0)

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

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

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 = pymunk.Body() # 1
    rotation_center_body.position = (300,300)

    body = pymunk.Body(10, 10000) # 2
    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)
```

```
    rotation_center_joint = pymunk.PinJoint(body, rotation_center_body, (0,0), (0,0)) # 3

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

#. 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.

#. The L shape will now be moving in the world, and therefor it can no longer have infinite mass. I have precalculated the inertia to 10000. (ok, I just took a number that worked, the important thing is that it looks good on screen!). #. 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.

To make it easy to see the point we draw a little red ball in its center

```
pygame.draw.circle(screen, THECOLORS["red"], (300,300), 5)
```

In a bigger program you will want to get the rotation_center_body.position instead of my little cheat here with (300,300), but it will work for this tutorial as the rotation center is static.

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 = pymunk.Body()
    rotation_center_body.position = (300,300)

    rotation_limit_body = pymunk.Body() # 1
    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)

    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_lir

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

1. We add a body..

#. 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.

And to make it a bit more clear, we draw a circle to do symbolize the joint with a green circle with its radius set to the joint max:

```
pygame.draw.circle(screen, THECOLORS["green"], (200,300), 25, 2)
```

The end

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 bellow the screen.

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

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

#. As we already have a loop we reuse it.. Check if the body.position is less than 0. #. If that is the case, we add it to our list of balls to remove. #. To remove an object from the space, we need to remove its shape and its body. #. 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 will balls, tipping over releasing them, tipping back and start over. You can check demo_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 add a comment in the bottom of the page. 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
import pygame
from pygame.locals import *
from pygame.color import *
import pymunk
import math

def to_pygame(p):
    """Small hack to convert pymunk to pygame coordinates"""
    return int(p.x), int(-p.y+600)

def add_ball(space):
    """Add a ball to the given space at a random position"""
    mass = 1
    radius = 14
    inertia = pymunk.moment_for_circle(mass, 0, radius, (0,0))
    body = pymunk.Body(mass, inertia)
    x = random.randint(120,380)
    body.position = x, 550
    shape = pymunk.Circle(body, radius, (0,0))
    space.add(body, shape)
    return shape

def draw_ball(screen, ball):
    """Draw a ball shape"""
    p = int(ball.body.position.x), 600-int(ball.body.position.y)
    pygame.draw.circle(screen, THECOLORS["blue"], p, int(ball.radius), 2)
```

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

    rotation_limit_body = pymunk.Body() # 1
    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)

    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_lir

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

def draw_lines(screen, lines):
    """Draw the lines"""
    for line in lines:
        body = line.body
        pv1 = body.position + line.a.rotated(body.angle)
        pv2 = body.position + line.b.rotated(body.angle)
        p1 = to_pygame(pv1)
        p2 = to_pygame(pv2)
        pygame.draw.lines(screen, THECOLORS["lightgray"], False, [p1,p2])


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()
    running = True

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

    lines = add_L(space)
    balls = []

    ticks_to_next_ball = 10
    while running:
        for event in pygame.event.get():
            if event.type == QUIT:
                running = False
            elif event.type == KEYDOWN and event.key == K_ESCAPE:
                running = False

        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(THECOLORS["white"])

        balls_to_remove = []
        for ball in balls:
            if ball.body.position.y < 150:
                balls_to_remove.append(ball)
            draw_ball(screen, ball)

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

        draw_lines(screen, lines)

        pygame.draw.circle(screen, THECOLORS["red"], (300,300), 5)
        pygame.draw.circle(screen, THECOLORS["green"], (200,300), 25, 2)

        space.step(1/50.0)

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

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

5.7 Advanced

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

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

To wrap Chipmunk pymunk uses ctypes. In the bottom an autogenerated layer, and then a handmade pythonic layer on top to make it nice to use from Python programs.

5.7.1 Why ctypes?

The reasons for ctypes instead of [your favorite wrapping solution] can be summarized as

- You only need to write pure python code when wrapping. This is good for several reasons. I can not really code in c. Sure, I can read it and write easy things, but Im not a good c coder. What I do know quite well is python. I imagine that the same is true for most people using pymunk, after all its a python library. :) Hopefully this means that users of pymunk can look at how stuff is actually done very easily, and for example add a missing chipmunk method/property on their own in their own code without much problem, and without being required to compile/build anything.
- ctypes is included in the standard library. Anyone with python has it already, no dependencies on 3rd party libraries, and some guarantee that it will stick around for a long time.
- The only thing required to run pymunk is python and a c compiler (in those cases a prebuilt version of chipmunk is not included). This should maximize the multiplatformness of pymunk, only thing that would even better would be a pure python library (which might be a bad idea for other reasons, mainly speed).
- Not much magic going on. Working with ctypes is quite straight forward. Sure, pymunk uses a generator which is a bit of a pain, but at least its possible to sidestep it if required, which Ive done in some cases. Ive also got a

share amount of problems when stuff didnt work as expected, but I imagine it would have been even worse with other solutions. At least its only the c library and python, and not some 3rd party in between.

- Non api-breaking fixes in chipmunk doesnt affect pymunk. If a bugfix, some optimization or whatever is done in chipmunk that doesnt affect the API, then its enough with a recompile of chipmunk with the new code to benefit from the fix. Easy for everyone.
- Ctypes can run on other python implementations than cpython. Right now pypy feels the most promising and it is be able to run ctypes just fine.

As I see it, the main benefit another solution could give would be speed. However, there are a couple of arguments why I dont find this as important as the benefits of ctypes

- You are writing your game in python in the first place, if you really required top performance than maybe rewrite the whole thing in c would be better anyway? Or make a optimized binding to chipmunk.

  For example, if you really need excellent performance then one possible optimization would be to write the drawing code in c as well, and have that interact with chipmunk directly. That way it can be made more performant than any generic wrapping solution as it would skip the whole layer.
- The bottleneck in a full game/application is somewhere else than in the physics wrapping in many cases. If your game has AI, logic and so on in python, then the wrapper overhead added by ctypes is not so bad in comparison.
- Pypy. ctypes on pypy has the potential to be very quick. However, right now with pypy-1.9 the speed of pymunk is actually a bit slower on pypy than on cpython. Hopefully this will improve in the future.

Note that pymunk has been around since late 2007 which means not all wrapping options that exist today did exist or was not stable/complete enough for use by pymunk in the beginning. There are more options available today, and using ctypes is not set in stone. If a better alternative comes around then pymunk might switch given the improvements are big enough.

5.7.2 Code Layout

Most of pymunk should be quite straight forward.

Except for the documented API pymunk has a couple of interesting parts. Low level bindings to Chipmunk, a custom library load function, 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 two files _chipmunk.py and _chipmunk_ffi.py. In order to locate and load the compiled chipmunk library file pymunk uses a custom load_library function in libload.py

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

pymunk/_chipmunk.py This file contains autogenerated low level ctypes binding created by the generate_bindings script.

pymunk/_chipmkunk_ffi.py This file contains manual bindings not automatically generated.

pymunk/libload.py This file contains the custom ctypes library load fucntion that is used by the rest of pymunk to load the Chipmunk library file.

setup.py Except for the standard setup stuff this file also contain the custom build commands to build Chipmunk from source.

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.

5.7.3 Tests

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

There is a helper script in the tools folder to easily run the tests:

```
> cd tools
> python run_tests.py
```

5.7.4 Working with non-wrapped parts of Chipmunk

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

Note: If you only want one or two new functions its probably easier to just add them manually to _chipmunk.py. See the ctypes documentation for instructions on what the function definitons/structs/whatever should look like.

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

First we need some imports:

```
from ctypes import *
from ._chipmunk import cpBody, chipmunk_lib, function_pointer
```

Then the actual ctypes wrapping:

```
cpBodyIsSleeping = (function_pointer(cpBool, POINTER(cpBody))).in_dll(chipmunk_lib, '_cpBodyIsSleepi
```

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

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

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

Full example:

```
from ctypes import *
from ._chipmunk import cpBody, chipmunk_lib, function_pointer

cpBodyIsSleeping = (function_pointer(cpBool, POINTER(cpBody))).in_dll(chipmunk_lib, '_cpBodyIsSleepi

def is_sleeping(body):
    return cpffi.cpBodyIsSleeping(body._body)

import pymunk
body = pymunk.Body(1,2)
print is_sleeping(body)
```

5.7.5 Regenerate bindings to Chipmunk

You need the ctypes code generator. It is part of the ctypeslib package. You will also need GCC_XML. See the ctypes wiki for instructions on how to set it up: http://starship.python.net/crew/theller/wiki/CodeGenerator

I have found that ctypeslib and gcc_xml are easiest to get to work under Linux. Even if you normally work under Windows I suggest you put up a virtual machine with a linux dist to make things easier.

When ctypeslib (h2xml and xml2py) and gcc_xml are installed you can use the helper file to regenerate the bindings. It is located in the tools folder:

```
> cd tools
> python generate_bindings.py
```

You have now created a _chipmunk.py file with generated bindings. (use –help to display options, you will most probably want to change the include path and possibly the lib path)

5.8 License

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

p

pymunk, **??**
pymunk.constraint, **??**
pymunk.pygame_util, **??**
pymunk.pyglet_util, **??**
pymunk.util, **??**
pymunk.vec2d, **??**
pymunkoptions, **??**