fireform Documentation

Release 0.0.0

DXsmiley

Jun 23, 2018

Contents

CHAPTER 1

Installation

1.1 System Requirements

Fireform is currently being developed and tested with Python 3.5.1, but python versions 3.4 and upward should work.

1.2 Installing Fireform

Fireform is currently in pre-alpha. Because of this, there's no PYPI entry yet. You can install the very latest version from the git repository with the following command:

```
pip install https://github.com/DXsmiley/fireform/archive/master.zip
```

1.3 Installing Engines

Fireform is designed so that it can work with a variety of backends, referred to as *engines*.

The engine handles the image loading, rendering, window creation and user input.

Currently, pyglet 1.2.4 is the only supported backend. Fireform ships with pyglet so you don't have to install it separately.

CHAPTER 2

Tutorials

2.1 First Project

This tutorial will walk you through creating your first game with Fireform.

If you haven't done so already, you should install Fireform.

2.1.1 Creating The World

Starting a Fireform game requires some boilerplate code, so open up a new python file and I'll walk you through it.

The first thing that we need to do is import fireform.

```
import fireform
```

For the sake of completeness, all things in the library will be referenced their full name.

Immediately after importing the library, we need to specify which engine we want to use.

```
fireform.engine.load('pyglet')
```

Pyglet is a lightweight media library for python.

Note: There are currently no other engines supported, so you don't actually get a choice here. This may seem redundant, but it's a form of future-proofing.

Next, we need to create a world. Worlds are used to hold all the objects in the game, and give them a space in which they can interact.

```
world = fireform.world.world()
```

The world won't actually do anything by itself, however. Systems are used to define the rules that govern the world. Systems would be akin to the forces of gravity and electromagnetism that govern our universe.

```
world.add_system(fireform.system.motion())
world.add_system(fireform.system.camera())
world.add_system(fireform.system.image())
world.add_system(fireform.system.debug(allow_edit = True))
```

These four 'standard' systems serve different purposes.

The *motion* system handles the movement of objects, and detects when they collide with each other.

The *camera* system allows the viewer to move around the world. Otherwise the viewpoint (camera) would be stuck in the one location the entire time.

The *image* system is responsible for rendering graphics.

The *debug* system gives us a number of tools to examine and manipulate entities when we run the game.

Finally, we have to start the game. This line will block, so make sure it is at the end of your code.

```
fireform.main.run(world)
```

If you run the code now you won't see much. That's because there's no entities in the game.

2.1.2 Creating The Player

Add the following code to the script. Make sure it comes before `fireform.world.run`.

```
player = fireform.entity(
        fireform.data.box(x = 0, y = 0, width = 60, height = 60)
)

world.add_entity(player)
```

The `fireform.entity` function accepts an arbitrary number of arguments. Each argument is a single aspect of the entity. *fireform.data.box* represents the bounding box of the entity.

Note: You may see some code that uses `fireform.entity.entity` instead of just `fireform.entity`. Either one works.

If you run the code now you should see a green square in the middle of the window. This is the entity we have just created. If you hover your mouse over it you should get a description of the entity on the left hand side of the screen. You can use the left mouse button to drag the box around the screen, and the right button to resize it.

Now lets add some motion to the object. We can add a velocity component to make the entity move, and we can add an acceleration component in order to make it accelerate in a particular direction.

```
player = fireform.entity(
        fireform.data.box(x = 0, y = 0, width = 60, height = 60),
        fireform.data.velocity(7, 4),
        fireform.data.acceleration(-0.2, -0.05)
)
```

If you run the game now, you should see the box move upwards and to the right, then turn around and exit on the left hand side of the screen.

For this object to actually be the 'player', the user will have to be able to control it. Remove the arguments passed to the velocity and acceleration components so that the box is initially at rest.

From here, we can add two more components to make the box controllable.

```
player = fireform.entity(
        fireform.data.box(x = 0, y = 0, width = 60, height = 60),
        fireform.data.velocity(),
        fireform.data.acceleration(),
        fireform.data.friction(0.9, 0.9),
        fireform.util.behaviour_eight_direction_movement(speed = 3)
)
```

The friction component will ensure that the player doesn't reach ridiculous speeds and spiral out of control.

Unlike the other components, `behaviour_eight_direction_movement` is a *behaviour*. This means that it responds to events that occur and will modify the entity. This particular behaviour listens for key press events and will set the acceleration of the entity when they happen.

If you run the game now, you should be able to move the green box around the screen using the arrow keys.

You can fiddle around with the values passed to friction and the movement behaviour in order to change how the box handles.

2.1.3 Adding Graphics

First, you'll need an image to add to the game. Open up your favourite image editor and draw something. The image should be 60 by 60 pixels, the same size as the player object. Save the image in the same directory as the code, and call it `my_image.png`.

Note: Currently, Fireform only supports PNG image files.

Next, you'll need to create a file to describe all your resources. Create a new text file in the same directory as the code, and call it *resources.json*. Paste the following code into it.

```
{
        "images":
        {
                "my_image":
                {
                        "x_offset": "50%",
                        "y_offset": "50%"
                }
        }
}
```

It doesn't look like much, but it will tell Fireform to load `my_image.png`. The `x_offset` and `y_offset` specify the image's centre. The `"50%"` specifies that it should be in the middle of the image. Setting it to an actual integer will specify the offset in number of pixels from the top left.

We then need to get Fireform to read this file. To do that we put the following line of code after `fireform.engine.load('pyglet')`.

```
fireform.resource.load('.')
```

The function takes any number of strings, representing the directories that Fireform should search when looking for images. The `.` represents the working directory (the one that the code is in).

If you had some of your resources separated into folders, you would have to mention those explicitly:

```
fireform.resource.load('.', './images', './audio')
```

Finally, we add the image to our player entity from earlier:

```
player = fireform.entity(
        fireform.data.box(x = 0, y = 0, width = 60, height = 60),
        fireform.data.velocity(),
        fireform.data.acceleration(),
        fireform.data.friction(0.9, 0.9),
        fireform.util.behaviour_eight_direction_movement(speed = 3),
        fireform.data.image('my_image')
)
```

If you run the game now you should now see your beautifully drawn picture running around the screen.

2.1.4 Completed Code

code.py

```
import fireform

fireform.engine.load('pyglet')

fireform.resource.load('.')

world = fireform.world.world()

world.add_system(fireform.system.motion())
world.add_system(fireform.system.camera())
world.add_system(fireform.system.image())
world.add_system(fireform.system.debug(allow_edit = True))

player = fireform.entity(
        fireform.data.box(x = 0, y = 0, width = 60, height = 60),
        fireform.data.velocity(),
        fireform.data.acceleration(),
        fireform.data.friction(0.9, 0.9),
        fireform.util.behaviour_eight_direction_movement(speed = 3),
        fireform.data.image('my_image')
)

world.add_entity(player)

fireform.main.run(world)
```

resources.json

```
{
        "images":
        {
                "my_image":
                {
                        "x_offset": "50%",
```

(continues on next page)

(continued from previous page)

```
                            "y_offset": "50%"
                    }
            }
}
```

2.2 Platformer

This tutorial will walk you through making a simple platformer game. It is assumed that you've followed the *previous tutorial*

Warning: This tutorial is incomplete.

2.2.1 Boilerplate

Here's the framework you'll need to get started. It should be somewhat familiar to you. The key difference is that the `motion` system has an additional parameter passed to it, `collision_mode = 'split'`. This seperates the collision detection step into two phases and helps us to determine what direction an object is moving when it collides with another.

```
import fireform

fireform.engine.load('pyglet')

fireform.resource.load('.')

world = fireform.world.world()

world.add_system(fireform.system.motion(collision_mode = 'split'))
world.add_system(fireform.system.camera())
world.add_system(fireform.system.image())
world.add_system(fireform.system.debug(allow_edit = True))

# We'll put the cool stuff here.

fireform.main.run(world)
```

2.2.2 Creating the Platforms

The player is going to need something to jump around on, so we'll create the platforms first.

We'll make a function to create them.

```
def make_platform(x, y, width, height):
        return fireform.entity(
                fireform.data.box(x = x, y = y, width = width, height = height),
                fireform.data.collision_bucket(extrovert = True),
                tags = 'solid'
        )
```

Note that the x and y values specify the centre of the object.

The `collision_bucket` component is used to specify some rules regarding what the object can and cannot collide with. We don't actually specify a bucket here so it will use the default one. We *do* however, specify that the entity is extroverted. Extroverted entities cannot collide with other extroverted entities.

Note: You don't need to specify a collision bucket for every entity. Entities without a `collision_bucket` component will be placed in the default bucket, and are considered to not be extroverted.

The last line tags the entity as `'solid'` so that we can identify what it is once the player collides with it.

We can then create platforms and add them to the world.

```
world.add_entity(make_platform(0, 0, 400, 30))
world.add_entity(make_platform(200, 100, 180, 10))
world.add_entity(make_platform(-200, 150, 180, 10))
```

If you run the game now you should see three green rectangles representing the platforms. Note that the green border is being produced by the debugging features.

Note: You can press `tab` while running the game to toggle the debug overlay.

2.2.3 A falling box

Here's the hard part. We need to define what the player should do when it runs into a wall, from any direction.

The following block of code is what is known as a *behaviour*. Behaviours in fireform need to inherit from the specified base class.

The functions that start with `m_` are called when certain messages are send to the entity. Excluding the `self` argument, they all take three arguments:

- `world` : The world that transmitted the message.
- `entity` : The entity the behaviour is attached to.
- `message` : The message itself.

```
class platformer(fireform.behaviour.base):

        def __init__(self):
                self.on_ground = True

        def m_tick(self, world, entity, message):
                if entity.velocity.y != 0:
                        self.on_ground = False

        def m_collision(self, world, entity, message):
                other = message.other
                if 'solid' in other.tags:
                        if message.direction == 'horisontal':
                                if entity.velocity.x > 0: # Moving to the right
                                        entity.box.right = other.box.left
                                if entity.velocity.x < 0: # Moving to the left
                                        entity.box.left = other.box.right
```

(continues on next page)

(continued from previous page)

```
                                entity.velocity.x = 0
                        if message.direction == 'vertical':
                                if entity.velocity.y > 0: # Moving upwards
                                        entity.box.top = other.box.bottom
                                if entity.velocity.y < 0: # Moving downwards
                                        entity.box.bottom = other.box.top
                                        self.on_ground = True
                                entity.velocity.y = 0
```

`m_tick` is fired on each and every game step. It's `message` parameter doesn't actually contain any data. Here, we check if the object is moving vertically. If it is, then we know that it's not sitting on the ground.

`m_collision` is fired during every game tick where the entity is overlaping with another. `message.other` is the entity that it overlaps with. `message.direction` is a string that can be used to figure out which the way in which the entities hit each other.

If we attach this behaviour (and a few other things) to an entity, we can see it in action.

```
world.add_entity(fireform.entity(
        fireform.data.box(x = 0, y = 100, width = 60, height = 60),
        fireform.data.velocity(),
        fireform.data.acceleration(0, -0.3),
        platformer()
))
```

You should see a box accelerate downwards and come to rest on the ground.

2.2.4 User input

Create another behaviour to handle the user input.

```
class controller(fireform.behaviour.base):

        def m_key_press(self, world, entity, message):
                if message.key == fireform.input.key.LEFT:
                        entity.acceleration.x -= 2
                if message.key == fireform.input.key.RIGHT:
                        entity.acceleration.x += 2
                if message.key == fireform.input.key.SPACE:
                        if entity[platformer].on_ground:
                                entity[platformer].on_ground = False
                                entity.velocity.y = 15

        def m_key_release(self, world, entity, message):
                if message.key == fireform.input.key.LEFT:
                        entity.acceleration.x += 2
                if message.key == fireform.input.key.RIGHT:
                        entity.acceleration.x -= 2
```

Change player entity:

```
world.add_entity(fireform.entity(
        fireform.data.box(x = 0, y = 300, width = 60, height = 60),
        fireform.data.velocity(),
        fireform.data.acceleration(0, -0.7),
        fireform.data.friction(0.8, 1),
```

(continues on next page)

(continued from previous page)

```
        platformer(),
        controller()
))
```

If you run the game you should be able to move using the arrow keys and the space bar.

CHAPTER 3

API Reference

3.1 engine

Handles loading and accessing of engines

Attributes

current The module of the currently loaded engine.

`fireform.engine.load(name)`

Load an engine.

This should not be called more than once per program execution.

Parameters

***name*: string** The name of the engine. Currently, only the `'pyglet'` engine is valid.

3.2 main

Module used to start the master game loop.

`fireform.main.run(*args, **kwargs)`

Parameters

the_world [*fireform.world.world*] The world to simulate.

window_width [int] The initial width of the window. Defaults to 1280.

window_height [int] The initial height of the window. Defaults to 800.

fullscreen [bool] Wheather the window should be fullscreen. Defaults to False.

clear_colour [tuple] A tuple of four ints representing a colour. Used to clear the window after each frame. Defaults to white.

show_fps [bool] Shows the FPS. Defaults to True.

ticks_per_second [int] The number of tick events that occur every second. Defaults to 60.

draw_rate [int] The frequency at which to re-draw the screen. A value of `1` will redraw it every tick. A value of `2` will redraw it every second tick. etc... Defaults to `1`.

borderless [bool] Creates a borderless window. Defaults to false.

position [tuple of two ints] Places the window at a particular location on the screen. If not specified, the window will be positioned by the operating system.

vsync [bool] Enables vsync. Defaults to False.

draw_handler [callable] Experimental.

`fireform.main.`**`stop`**`()`

Stops the main game loop.

Warning: Not Implemented

3.3 resource

3.3.1 resources.json

The `resources.json` file is the file that is used to list all the resources used by your game.

3.3.2 module

`fireform.resource.`**`load`**`(`**paths*, *smooth_images=False*`)`

Load resources.

Parameters

paths [str] File paths to search relative to the working directory of the game.

Note: This may be changed to be relative to the program directory of the game in the future.

smooth_images [bool] Set to `True` to smooth images when they are resized. Most games should enable this. Pixel art games should not. Defaults to false.

`fireform.resource.`**`open_data`**`(`*filename*, *mode='r'*`)`

Opens a file found in the search paths.

Parameters

filename [str] The filename.

mode [str] The mode to open the file in. This is the same as the `mode` parameter on the builtin `open` function. Defaults to `'r'` (read, text mode).

3.4 world

`class` `fireform.world.`**`world`**

World used to control game events and ticks. Holds a list of entities.

`add_entities`(*entities*)
Add multiple entities to the world.

Parameters

***entities*: iterable** An iterable that produces *entity* objects.

`add_entity`(*entity*)
Add a single entity to the world.

Parameters

entity*: *`fireform.entity.entity` The entity to add to the world.

`add_system`(*system*)
Add a system to the world

`destroy_all_entities`()
Kills all entities in the world. This thing is *merciless*.

`handle_message`(*message*)
Used internally.

`handle_message_private`(*message*, *entities*)
Used internally.

`post_message`(*message*)
Sends a message to all systems and entities in the world that are listening for it.

Parameters

message*: *fireform.message.base The message to send.

`post_message_private`(*message*, *entities*)
Sends a message to a few entities and anything that is observing those entities. Also sends the message to the systems.

Parameters

message*: *fireform.message.base The message to send.

`refresh_entities`()
Remove all dead entities from the entity list, and sort the living ones by ordering.

Any entities added or destroyed will invalidate the list. You can call this to clean it up (but it is somewhat expensive).

3.5 entity

This class can also be access with `fireform.entity` if `import fireform` was used.

`class` `fireform.entity.`**`entity`**(**contents*, *ordering=0*, *tags=set()*)
A game entity.

This class should not be inherited from (because that's not how things work).

An entities' attibutes and behaviours are defined by the 'blobs' that they are made from.

`attach`(*c*)
Add either a data or behaviour object to the entity.

Data and behaviour objects can only be assigned to one entity. An entity cannot have more than one of any type of named data of behaviour. Note that is may have multiple instances of anonymous datum or behaviours (future feature).

Do not use this method once the entity has been added to the world.

Parameters

c*: *fireform.data.base* or *fireform.behaviour.base The component to add

`kill``()`

Destroy the entity.

The entity will be removed from the world on the next tick, so you should still check if it is alive when interacting with it.

3.6 data

3.6.1 base

class `fireform.data.`**`base`**

Base class from which other data objects should inherit

Attributes

***name*: string** Specifies the name of the component. Should be overridden in the child class. This is used when accessing the component as `my_entity['name']`. Leave unset to make this method of access not available.

***attribute_name*: string** This is used when accessing the component as `my_entity.attribute_name`. Leave unset to make this method of access not available. It is advisiable not to set this except for components that are accessed very frequently.

3.6.2 box

class `fireform.data.`**`box`**(*x=0*, *y=0*, *pos=None*, *width=10*, *height=10*, *size=None*, *anchor_x=0.5*, *anchor_y=0.5*, *anchor=None*)

Represents the bounding box of an entity.

Attributes

x [float] The x ordinate of the box's origin.

y [float] The y ordinate of the box's origin.

width [float] The width of the box.

height [float] The height of the box.

anchor_x [float] The position of the box's x origin relative to the sides of the box. 0 is on the left, 1 is on the right and 0.5 is in the middle. Defaults to 0.5.

anchor_y [float] The position of the box's y origin relative to the sides of the box. 0 is on the bottom, 1 is on the top and 0.5 is in the middle. Defaults to 0.5.

`area`

The box's area.

bottom
The y co-ordinate of the bottom of the box.

Setting this property will change the box's position and keep its height the same.

contains (*point*)
Returns true iff the point lies inside the box.

Parameters

point [`fireform.geom.vector`] The point to test.

left
The x co-ordinate of the left hand side of the box.

Setting this property will change the box's position and keep its width the same.

rectangle
A `fireform.geom.rectangle` in the same space as the box.

right
The x co-ordinate of the right hand side of the box.

Setting this property will change the box's position and keep its width the same.

top
The y co-ordinate of the top of the box.

Setting this property will change the box's position and keep its height the same.

3.6.3 image

class `fireform.data.`**image**(*image=None*, *depth=0*, *frame=0*, *speed=0*, *scale=1*, *rotation=0*, *blend=None*, *alpha=255*, *scissor=None*)
Used to show an image!

Attirbutes

image [string] The name of the image to display.

frame [float] The frame of the animation to display. This will be rounded down when actually determining which frame to show.

roation [float] Rotation of the image. Clockwise, in degrees.

scale [`fireform.geom.vector`] How the image should be stretched.

speed [float] The number of frames to advance per tick.

alpha [int] The opacity of the image, in the range of 0 to 255 inclusive. 255 is completely solid, 0 is invisible.

blend [string] Experimental.

scissor [*`fireform.entity.entity`*] Experimental.

3.6.4 velocity

class `fireform.data.`**velocity**(**args*, *x=None*, *y=None*)
The speed of an entity.

3.6.5 acceleration

`class fireform.data.acceleration(*args, x=None, y=None)`

Acceleration datum.

Acceleration is defined as change in velocity per tick. This will have no impact on the instance unless it also has the velocity and position datum objects.

Attributes

***x*: int** Acceleration on the x-axis in pixels per tick per tick.

***y*: int** Acceleration on the y-axis in pixels per tick per tick.

3.6.6 friction

`class fireform.data.friction(*args)`

Multiplies the velocity of an entity every tick, causing it to accelerate or decelerate.

3.6.7 collision_bucket

`class fireform.data.collision_bucket(bucket='default', extrovert=False)`

Specifies the bucket of collisions into which this entity should be entered. Entities need to be in the same bucket in order to recieve collision events.

Parameters

***bucket*: string** The name of the bucket. Technically this can actually be any type that can be hashed in order to be put into a dictionary

***extrovert*: bool** Whether the entity is 'extroverted'. Two entities that are both extroverted cannot collide with each other. Defaults to False.

3.6.8 camera

`class fireform.data.camera(zoom=1, weight=1)`

Camera data. You can attach this to entities and the camera will follow them around.

If you have multiple entities with a camera, the view will be positioned at their average location, however the view will NOT zoom if they get too far aprt.

3.7 system

3.7.1 base

`class fireform.system.base`

Base class for systems to inherit from.

3.7.2 motion

class `fireform.system.`**`motion`**(*collision_mode='normal'*, *ignore_masks=False*)

Moves entities around, handling velocity acceleration and friction.

Is also resposible for detecting collisions.

Parameters

collision_mode [string]

- If `'disabled'`, collisions will not be checked.
- If `'normal'`, collisions will be checked.
- If `'split'`, motion will occur on each axis sperately. Collision events will have the `direction` attribute set either `'horisontal'` or `'vertical'`.

ignore_masks [bool] If enabled, all entities will be treated as if they had rectangular collision masks. Defaults to `False`.

3.7.3 image

class `fireform.system.`**`image`**

This system draws images for you.

3.7.4 camera

class `fireform.system.`**`camera`**(*letterbox=False*)

Shove this in your world for cameras to work.

You probably want cameras to work.

3.7.5 debug

class `fireform.system.`**`debug`**(***kwargs*)

Usefull for debugging.

Draws boxes around objects that have a position and size.

Parameters

text_colour [tuple] Colour used to render text. Defaults to black.

outline_colour [tuple] Outline used for normal entities. Defaults to green.

outline_colour_solid [tuple] Outline used for solid entities. Defaults to bright green.

outline_colour_hover [tuple] Colour used to outline the entity the mouse is hovering over. Defaults to cyan.

outline_colour_hover [tuple] Colour used to outline errored entities. Defaults to red.

allow_edit [bool] Setting to `True` will allow entities to be inspected, moved and resized using the mouse. Defaults to `False`.

display [bool] Setting to `True` will display information about the state of the program. Defaults to `True`.

3.8 message

All inbuilt messages.

`fireform.message.`**`animate`**`()`

Dispatches once per game tick, after the tick events and before the drae event. It exists to seperate the tick-dependent graphics logic from the tick-independent logic.

`class` `fireform.message.`**`base`**

Base message from which all other messages need to inherit.

Attributes

name [string] The name of the message. Behaviour and systems that want to listen for a message have to implement a function called `m_name`. For example, if the message's name was `tick`, the behaviour would have to implement `m_tick`.

`decipher_name``()`

Returns the name of the message.

This should be used instead of `name`, since some older messages may use a function to implement their name.

`class` `fireform.message.`**`collision`**`(`*first*, *second*, *direction*`)`

Signals that two entities have overlapped.

fireform.system.motion will send these once per tick until they stop colliding.

This is sent as a private message, so only the entities that actually collide will receive it.

Attributes

other [*`fireform.entity.entity`*] The *other* entity.

direction [string] The direction in which the entities were moving when they collided. This is only applicable when *`the collision mode is set to split`*.

`class` `fireform.message.`**`collision_enter`**`(`*other*`)`

Signals that two entities have overlapped.

This is sent as a private message, so only the entities that actually collide will receive it.

Attributes

other [*`fireform.entity.entity`*] The *other* entity.

`class` `fireform.message.`**`collision_exit`**`(`*other*`)`

Signals that two entities have stopped overlapping.

This is sent as a private message, so only the entities that actually collide will receive it.

Attributes

other [*`fireform.entity.entity`*] The *other* entity.

`class` `fireform.message.`**`collision_late`**`(`*first*, *second*`)`

Signals that two entities have overlapped.

This event is fired after the main round of collision events, after all objects have finished moving. If a collision event is fired, a collision_late event will be fired.

Collision late events are usefull for when the velocity of an object needs to be changed, because that could otherwise mess with some physics logic that relies on knowing the direction that the object is moving in.

class `fireform.message.`**`dead_entity`**(*entity*)

Signals that an entitity has been killed.

This is triggered when the world decides to remove it, not when entity.kill() is called.

Attributes

entity [`fireform.entity.entity`] The dead entity.

class `fireform.message.`**`draw`**(*layer*)

Dispatched whenever a layer is drawn.

`fireform.message.`**`frozen_tick`**()

This message is dispatched once per tick while the system is paused.

class `fireform.message.`**`generic`**(*my_name*)

Generic messages which hold no data.

Attributes

name [string] Name of the message.

class `fireform.message.`**`key_press`**(*key*, *modifiers*)

Signifies that a key was pressed.

Attributes

key [*fireform.input.key*] The key that was pressed.

modifiers [something] The modifier keys (shift, control, etc.) That were being held when the key was pressed.

class `fireform.message.`**`key_release`**(*key*, *modifiers*)

Signifies that a key was released.

Attributes

key [*fireform.input.key*] The key that was released.

modifiers [something] The modifier keys (shift, control, etc.) That were being held when the key was released.

class `fireform.message.`**`mouse_click`**(*x*, *y*, *button*)

Signifies the a mouse button was clicked.

Attributes

x [float] The x position of the cursor, within the world.

y [float] The y position of the cursor, within the world.

button [something] The button that was pressed.

class `fireform.message.`**`mouse_click_raw`**(*x*, *y*, *button*)

Signifies the a mouse button was clicked.

Attributes

x [float] The x position of the cursor, relative to the corner of the window.

y [float] The y position of the cursor, relative to the corner of the window.

button [something] The button that was pressed.

class `fireform.message.`**`mouse_move`**(*x*, *y*)

Signifies that the mouse was moved.

This may not trigger if the mouse is moved outside the window.

Attributes

x [float] The x position of the cursor, within the world.

y [float] The y position of the cursor, within the world.

class `fireform.message.`**`mouse_move_raw`**(*x*, *y*)
Signifies that the mouse was moved.

This may not trigger if the mouse is moved outside the window.

Attributes

x [float] The x position of the cursor, relative to the corner of the window.

y [float] The y position of the cursor, relative to the corner of the window.

class `fireform.message.`**`mouse_release`**(*x*, *y*, *button*)
Signifies the a mouse button was released.

Attributes

x [float] The x position of the cursor, within the world.

y [float] The y position of the cursor, within the world.

button [something] The button that was released.

class `fireform.message.`**`mouse_release_raw`**(*x*, *y*, *button*)
Signifies the a mouse button was released.

Attributes

x [float] The x position of the cursor, relative to the corner of the window.

y [float] The y position of the cursor, relative to the corner of the window.

button [something] The button that was released.

class `fireform.message.`**`new_entity`**(*entity*)
Signifies that an entitiy was added to the world.

Attributes

entity [*`fireform.entity.entity`*] The entity that was just added to the world.

`fireform.message.`**`surpass_frozen`**(*function*)
Decorator to apply to a message handler if it should be called even if the world is frozen.

`fireform.message.`**`tick`**()
This message is dispatched once per game tick.

class `fireform.message.`**`update_tracked_value`**(*key*, *value*)

class `fireform.message.`**`window_resized`**(*width*, *height*)
Signifies that the window was resized.

Attributes

width [int] The width of the window.

height [int] The height of the window.

3.9 tiled

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index