Pygame Documentation

Release 1.9.2

Pygame Developers

March 16, 2014

Contents

CHAPTER 1

pygame.camera

pygame module for camera use

Pygame currently supports only Linux and v4l2 cameras.

EXPERIMENTAL!: This api may change or disappear in later pygame releases. If you use this, your code will very likely break with the next pygame release.

The Bayer to RGB function is based on:

```
Sonix SN9C101 based webcam basic I/F routines
Copyright (C) 2004 Takafumi Mizuno <taka-qce@ls-a.jp>
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

New in pygame 1.9.0.

pygame.camera.**colorspace**()

> Surface colorspace conversion
> colorspace(Surface, format, DestSurface = None) -> Surface
>
> Allows for conversion from "RGB" to a destination colorspace of "HSV" or "YUV". The source and destination surfaces must be the same size and pixel depth. This is useful for computer vision on devices with limited processing power. Capture as small of an image as possible, `transform.scale()` it even smaller, and then convert the colorspace to `YUV` or `HSV` before doing any processing on it.

pygame.camera.**list_cameras**()

returns a list of available cameras
list_cameras() -> [cameras]

Checks the computer for available cameras and returns a list of strings of camera names, ready to be fed into pygame.camera.Camera.

class pygame.camera.**Camera**

load a camera
Camera(device, (width, height), format) -> Camera

Loads a v4l2 camera. The device is typically something like "/dev/video0". Default width and height are 640 by 480. Format is the desired colorspace of the output. This is useful for computer vision purposes. The default is RGB. The following are supported:

- RGB - Red, Green, Blue
- YUV - Luma, Blue Chrominance, Red Chrominance
- HSV - Hue, Saturation, Value

start()

opens, initializes, and starts capturing
start() -> None

Opens the camera device, attempts to initialize it, and begins recording images to a buffer. The camera must be started before any of the below functions can be used.

stop()

stops, uninitializes, and closes the camera
stop() -> None

Stops recording, uninitializes the camera, and closes it. Once a camera is stopped, the below functions cannot be used until it is started again.

get_controls()

gets current values of user controls
get_controls() -> (hflip = bool, vflip = bool, brightness)

If the camera supports it, get_controls will return the current settings for horizontal and vertical image flip as bools and brightness as an int. If unsupported, it will return the default values of (0, 0, 0). Note that the return values here may be different than those returned by set_controls, though these are more likely to be correct.

set_controls()

changes camera settings if supported by the camera
set_controls(hflip = bool, vflip = bool, brightness) -> (hflip = bool, vflip = bool, brightness)

Allows you to change camera settings if the camera supports it. The return values will be the input values if the camera claims it succeeded or the values previously in use if not. Each argument is optional, and the

desired one can be chosen by supplying the keyword, like hflip. Note that the actual settings being used by the camera may not be the same as those returned by set_controls.

get_size()

returns the dimensions of the images being recorded

get_size() -> (width, height)

Returns the current dimensions of the images being captured by the camera. This will return the actual size, which may be different than the one specified during initialization if the camera did not support that size.

query_image()

checks if a frame is ready

query_image() -> bool

If an image is ready to get, it returns true. Otherwise it returns false. Note that some webcams will always return False and will only queue a frame when called with a blocking function like `get_image()`. This is useful to separate the framerate of the game from that of the camera without having to use threading.

get_image()

captures an image as a Surface

get_image(Surface = None) -> Surface

Pulls an image off of the buffer as an `RGB` Surface. It can optionally reuse an existing Surface to save time. The bit depth of the surface is either 24bits or the same as the optionally supplied Surface.

get_raw()

returns an unmodified image as a string

get_raw() -> string

Gets an image from a camera as a string in the native pixelformat of the camera. Useful for integration with other libraries.

CHAPTER 2

pygame.cdrom

pygame module for audio cdrom control

The cdrom module manages the CD and DVD drives on a computer. It can also control the playback of audio cd's. This module needs to be initialized before it can do anything. Each CD object you create represents a cdrom drive and must also be initialized individually before it can do most things.

pygame.cdrom.**init**()

initialize the cdrom module
init() -> None

Initialize the cdrom module. This will scan the system for all CD devices. The module must be initialized before any other functions will work. This automatically happens when you call pygame.init().

It is safe to call this function more than once.

pygame.cdrom.**quit**()

uninitialize the cdrom module
quit() -> None

Uninitialize the cdrom module. After you call this any existing CD objects will no longer work.

It is safe to call this function more than once.

pygame.cdrom.**get_init**()

true if the cdrom module is initialized
get_init() -> bool

Test if the cdrom module is initialized or not. This is different than the CD.init() since each drive must also be initialized individually.

pygame.cdrom.**get_count**()

number of cd drives on the system
get_count() -> count

Return the number of cd drives on the system. When you create CD objects you need to pass an integer id that must be lower than this count. The count will be 0 if there are no drives on the system.

class pygame.cdrom.**CD**

class to manage a cdrom drive
CD(id) -> CD

You can create a CD object for each cdrom on the system. Use pygame.cdrom.get_count() to determine how many drives actually exist. The id argument is an integer of the drive, starting at zero.

The CD object is not initialized, you can only call CD.get_id() and CD.get_name() on an uninitialized drive.

It is safe to create multiple CD objects for the same drive, they will all cooperate normally.

init()

initialize a cdrom drive for use
init() -> None

Initialize the cdrom drive for use. The drive must be initialized for most CD methods to work. Even if the rest of pygame has been initialized.

There may be a brief pause while the drive is initialized. Avoid CD.init() if the program should not stop for a second or two.

quit()

uninitialize a cdrom drive for use
quit() -> None

Uninitialize a drive for use. Call this when your program will not be accessing the drive for awhile.

get_init()

true if this cd device initialized
get_init() -> bool

Test if this CDROM device is initialized. This is different than the pygame.cdrom.init() since each drive must also be initialized individually.

play()

start playing audio
play(track, start=None, end=None) -> None

Playback audio from an audio cdrom in the drive. Besides the track number argument, you can also pass a starting and ending time for playback. The start and end time are in seconds, and can limit the section of an audio track played.

If you pass a start time but no end, the audio will play to the end of the track. If you pass a start time and 'None' for the end time, the audio will play to the end of the entire disc.

See the CD.get_numtracks() and CD.get_track_audio() to find tracks to playback.

Note, track 0 is the first track on the CD. Track numbers start at zero.

stop()

stop audio playback

stop() -> None

Stops playback of audio from the cdrom. This will also lose the current playback position. This method does nothing if the drive isn't already playing audio.

pause()

temporarily stop audio playback
pause() -> None

Temporarily stop audio playback on the CD. The playback can be resumed at the same point with the CD.resume() method. If the CD is not playing this method does nothing.

Note, track 0 is the first track on the CD. Track numbers start at zero.

resume()

unpause audio playback
resume() -> None

Unpause a paused CD. If the CD is not paused or already playing, this method does nothing.

eject()

eject or open the cdrom drive
eject() -> None

This will open the cdrom drive and eject the cdrom. If the drive is playing or paused it will be stopped.

get_id()

the index of the cdrom drive
get_id() -> id

Returns the integer id that was used to create the CD instance. This method can work on an uninitialized CD.

get_name()

the system name of the cdrom drive
get_name() -> name

Return the string name of the drive. This is the system name used to represent the drive. It is often the drive letter or device name. This method can work on an uninitialized CD.

get_busy()

true if the drive is playing audio
get_busy() -> bool

Returns True if the drive busy playing back audio.

get_paused()

true if the drive is paused
get_paused() -> bool

Returns True if the drive is currently paused.

get_current()

the current audio playback position
get_current() -> track, seconds

Returns both the current track and time of that track. This method works when the drive is either playing or paused.

Note, track 0 is the first track on the CD. Track numbers start at zero.

get_empty()

False if a cdrom is in the drive
get_empty() -> bool

Return False if there is a cdrom currently in the drive. If the drive is empty this will return True.

get_numtracks()

the number of tracks on the cdrom
get_numtracks() -> count

Return the number of tracks on the cdrom in the drive. This will return zero of the drive is empty or has no tracks.

get_track_audio()

true if the cdrom track has audio data
get_track_audio(track) -> bool

Determine if a track on a cdrom contains audio data. You can also call CD.num_tracks() and CD.get_all() to determine more information about the cdrom.

Note, track 0 is the first track on the CD. Track numbers start at zero.

get_all()

get all track information
get_all() -> [(audio, start, end, lenth), ...]

Return a list with information for every track on the cdrom. The information consists of a tuple with four values. The audio value is True if the track contains audio data. The start, end, and length values are floating point numbers in seconds. Start and end represent absolute times on the entire disc.

get_track_start()

start time of a cdrom track
get_track_start(track) -> seconds

Return the absolute time in seconds where at start of the cdrom track.

Note, track 0 is the first track on the CD. Track numbers start at zero.

get_track_length()

length of a cdrom track
get_track_length(track) -> seconds

Return a floating point value in seconds of the length of the cdrom track.

Note, track 0 is the first track on the CD. Track numbers start at zero.

CHAPTER 3

pygame.Color

class pygame.**Color**

pygame object for color representations
Color(name) -> Color
Color(r, g, b, a) -> Color
Color(rgbvalue) -> Color

The Color class represents RGBA color values using a value range of 0-255. It allows basic arithmetic operations to create new colors, supports conversions to other color spaces such as HSV or HSL and lets you adjust single color channels. Alpha defaults to 255 when not given.

'rgbvalue' can be either a color name, an HTML color format string, a hex number string, or an integer pixel value. The HTML format is '#rrggbbaa', where rr, gg, bb, and aa are 2 digit hex numbers. The alpha aa is optional. A hex number string has the form '0xrrggbbaa', where aa is optional.

Color objects support equality comparison with other color objects and 3 or 4 element tuples of integers (New in 1.9.0). There was a bug in pygame 1.8.1 where the default alpha was 0, not 255 like previously.

New implementation of Color was done in pygame 1.8.1.

r

Gets or sets the red value of the Color.
r -> int

The red value of the Color.

g

Gets or sets the green value of the Color.
g -> int

The green value of the Color.

b

Gets or sets the blue value of the Color.
b -> int

The blue value of the Color.

a

Gets or sets the alpha value of the Color.
a -> int

The alpha value of the Color.

cmy

Gets or sets the CMY representation of the Color.
cmy -> tuple

The CMY representation of the Color. The CMY components are in the ranges C = [0, 1], M = [0, 1], Y = [0, 1]. Note that this will not return the absolutely exact CMY values for the set RGB values in all cases. Due to the RGB mapping from 0-255 and the CMY mapping from 0-1 rounding errors may cause the CMY values to differ slightly from what you might expect.

hsva

Gets or sets the HSVA representation of the Color.
hsva -> tuple

The HSVA representation of the Color. The HSVA components are in the ranges H = [0, 360], S = [0, 100], V = [0, 100], A = [0, 100]. Note that this will not return the absolutely exact HSV values for the set RGB values in all cases. Due to the RGB mapping from 0-255 and the HSV mapping from 0-100 and 0-360 rounding errors may cause the HSV values to differ slightly from what you might expect.

hsla

Gets or sets the HSLA representation of the Color.
hsla -> tuple

The HSLA representation of the Color. The HSLA components are in the ranges H = [0, 360], S = [0, 100], V = [0, 100], A = [0, 100]. Note that this will not return the absolutely exact HSL values for the set RGB values in all cases. Due to the RGB mapping from 0-255 and the HSL mapping from 0-100 and 0-360 rounding errors may cause the HSL values to differ slightly from what you might expect.

i1i2i3

Gets or sets the I1I2I3 representation of the Color.
i1i2i3 -> tuple

The I1I2I3 representation of the Color. The I1I2I3 components are in the ranges I1 = [0, 1], I2 = [-0.5, 0.5], I3 = [-0.5, 0.5]. Note that this will not return the absolutely exact I1I2I3 values for the set RGB values in all cases. Due to the RGB mapping from 0-255 and the I1I2I3 mapping from 0-1 rounding errors may cause the I1I2I3 values to differ slightly from what you might expect.

normalize()

Returns the normalized RGBA values of the Color.
normalize() -> tuple

Returns the normalized RGBA values of the Color as floating point values.

correct_gamma()

Applies a certain gamma value to the Color.
correct_gamma (gamma) -> Color

Applies a certain gamma value to the Color and returns a new Color with the adjusted RGBA values.

set_length()

Set the number of elements in the Color to 1,2,3, or 4.
set_length(len) -> None

The default Color length is 4. Colors can have lengths 1,2,3 or 4. This is useful if you want to unpack to r,g,b and not r,g,b,a. If you want to get the length of a Color do `len(acolor)`.

New in pygame 1.9.0.

CHAPTER 4

pygame.cursors

pygame module for cursor resources

Pygame offers control over the system hardware cursor. Pygame only supports black and white cursors for the system. You control the cursor with functions inside pygame.mouse.

This cursors module contains functions for loading and unencoding various cursor formats. These allow you to easily store your cursors in external files or directly as encoded python strings.

The module includes several standard cursors. The pygame.mouse.set_cursor() function takes several arguments. All those arguments have been stored in a single tuple you can call like this:

```
>>> pygame.mouse.set_cursor(*pygame.cursors.arrow)
```

This module also contains a few cursors as formatted strings. You'll need to pass these to pygame.cursors.compile() function before you can use them. The example call would look like this:

```
>>> cursor = pygame.cursors.compile(pygame.cursors.textmarker_strings)
>>> pygame.mouse.set_cursor(*cursor)
```

The following variables are cursor bitmaps that can be used as cursor:

- pygame.cursors.arrow
- pygame.cursors.diamond
- pygame.cursors.broken_x
- pygame.cursors.tri_left
- pygame.cursors.tri_right

The following strings can be converted into cursor bitmaps with pygame.cursors.compile() :

- pygame.cursors.thickarrow_strings
- pygame.cursors.sizer_x_strings
- pygame.cursors.sizer_y_strings
- pygame.cursors.sizer_xy_strings

pygame.cursors.**compile**()

create binary cursor data from simple strings
compile(strings, black='X', white='.', xor='o') -> data, mask

A sequence of strings can be used to create binary cursor data for the system cursor. The return values are the same format needed by `pygame.mouse.set_cursor()`.

If you are creating your own cursor strings, you can use any value represent the black and white pixels. Some system allow you to set a special toggle color for the system color, this is also called the xor color. If the system does not support xor cursors, that color will simply be black.

The width of the strings must all be equal and be divisible by 8. An example set of cursor strings looks like this

```
thickarrow_strings = (               #sized 24x24
  "XX                      ",
  "XXX                     ",
  "XXXX                    ",
  "XX.XX                   ",
  "XX..XX                  ",
  "XX...XX                 ",
  "XX....XX                ",
  "XX.....XX               ",
  "XX......XX              ",
  "XX.......XX             ",
  "XX........XX            ",
  "XX........XXX           ",
  "XX......XXXXX           ",
  "XX.XXX..XX              ",
  "XXXX XX..XX             ",
  "XX   XX..XX             ",
  "     XX..XX             ",
  "      XX..XX            ",
  "      XX..XX            ",
  "       XXXX             ",
  "       XX               ",
  "                        ",
  "                        ",
  "                        ")
```

`pygame.cursors.load_xbm()`

load cursor data from an xbm file

load_xbm(cursorfile) -> cursor_args

load_xbm(cursorfile, maskfile) -> cursor_args

This loads cursors for a simple subset of `XBM` files. `XBM` files are traditionally used to store cursors on unix systems, they are an ascii format used to represent simple images.

Sometimes the black and white color values will be split into two separate `XBM` files. You can pass a second maskfile argument to load the two images into a single cursor.

The cursorfile and maskfile arguments can either be filenames or filelike object with the readlines method.

The return value cursor_args can be passed directly to the `pygame.mouse.set_cursor()` function.

CHAPTER 5

pygame.display

pygame module to control the display window and screen

This module offers control over the pygame display. Pygame has a single display Surface that is either contained in a window or runs full screen. Once you create the display you treat it as a regular Surface. Changes are not immediately visible onscreen, you must choose one of the two flipping functions to update the actual display.

The origin of the display, where x = 0, and y = 0 is the top left of the screen. Both axis increase positively towards the bottom right of the screen.

The pygame display can actually be initialized in one of several modes. By default the display is a basic software driven framebuffer. You can request special modules like hardware acceleration and OpenGL support. These are controlled by flags passed to `pygame.display.set_mode()`.

Pygame can only have a single display active at any time. Creating a new one with `pygame.display.set_mode()` will close the previous display. If precise control is needed over the pixel format or display resolutions, use the functions `pygame.display.mode_ok()`, `pygame.display.list_modes()`, and `pygame.display.Info()` to query information about the display.

Once the display Surface is created, the functions from this module effect the single existing display. The Surface becomes invalid if the module is uninitialized. If a new display mode is set, the existing Surface will automatically switch to operate on the new display.

Then the display mode is set, several events are placed on the pygame event queue. `pygame.QUIT` is sent when the user has requested the program to shutdown. The window will receive `pygame.ACTIVEEVENT` events as the display gains and loses input focus. If the display is set with the `pygame.RESIZABLE` flag, `pygame.VIDEORESIZE` events will be sent when the user adjusts the window dimensions. Hardware displays that draw direct to the screen will get `pygame.VIDEOEXPOSE` events when portions of the window must be redrawn.

`pygame.display.`**`init`**`()`

> Initialize the display module
>
> init() -> None
>
> Initializes the pygame display module. The display module cannot do anything until it is initialized. This is usually handled for you automatically when you call the higher level `pygame.init()`.
>
> Pygame will select from one of several internal display backends when it is initialized. The display mode will be chosen depending on the platform and permissions of current user. Before the display module is initialized the environment variable `SDL_VIDEODRIVER` can be set to control which backend is used. The systems with multiple choices are listed here.

```
Windows : windib, directx
Unix    : x11, dga, fbcon, directfb, ggi, vgl, svgalib, aalib
```

On some platforms it is possible to embed the pygame display into an already existing window. To do this, the environment variable SDL_WINDOWID must be set to a string containing the window id or handle. The environment variable is checked when the pygame display is initialized. Be aware that there can be many strange side effects when running in an embedded display.

It is harmless to call this more than once, repeated calls have no effect.

pygame.display.**quit**()

Uninitialize the display module
quit() -> None

This will shut down the entire display module. This means any active displays will be closed. This will also be handled automatically when the program exits.

It is harmless to call this more than once, repeated calls have no effect.

pygame.display.**get_init**()

Returns True if the display module has been initialized
get_init() -> bool

Returns True if the pygame.display module is currently initialized.

pygame.display.**set_mode**()

Initialize a window or screen for display
set_mode(resolution=(0,0), flags=0, depth=0) -> Surface

This function will create a display Surface. The arguments passed in are requests for a display type. The actual created display will be the best possible match supported by the system.

The resolution argument is a pair of numbers representing the width and height. The flags argument is a collection of additional options. The depth argument represents the number of bits to use for color.

The Surface that gets returned can be drawn to like a regular Surface but changes will eventually be seen on the monitor.

If no resolution is passed or is set to (0, 0) and pygame uses SDL version 1.2.10 or above, the created Surface will have the same size as the current screen resolution. If only the width or height are set to 0, the Surface will have the same width or height as the screen resolution. Using a SDL version prior to 1.2.10 will raise an exception.

It is usually best to not pass the depth argument. It will default to the best and fastest color depth for the system. If your game requires a specific color format you can control the depth with this argument. Pygame will emulate an unavailable color depth which can be slow.

When requesting fullscreen display modes, sometimes an exact match for the requested resolution cannot be made. In these situations pygame will select the closest compatible match. The returned surface will still always match the requested resolution.

The flags argument controls which type of display you want. There are several to choose from, and you can even combine multiple types using the bitwise or operator, (the pipe "|" character). If you pass 0 or no flags argument it will default to a software driven window. Here are the display flags you will want to choose from:

```
pygame.FULLSCREEN    create a fullscreen display
pygame.DOUBLEBUF     recommended for HWSURFACE or OPENGL
pygame.HWSURFACE     hardware accelerated, only in FULLSCREEN
pygame.OPENGL        create an OpenGL renderable display
pygame.RESIZABLE     display window should be sizeable
pygame.NOFRAME       display window will have no border or controls
```

For example:

```
# Open a window on the screen
screen_width=700
screen_height=400
screen=pygame.display.set_mode([screen_width,screen_height])
```

pygame.display.**get_surface**()

Get a reference to the currently set display surface
get_surface() -> Surface

Return a reference to the currently set display Surface. If no display mode has been set this will return None.

pygame.display.**flip**()

Update the full display Surface to the screen
flip() -> None

This will update the contents of the entire display. If your display mode is using the flags pygame.HWSURFACE and pygame.DOUBLEBUF, this will wait for a vertical retrace and swap the surfaces. If you are using a different type of display mode, it will simply update the entire contents of the surface.

When using an pygame.OPENGL display mode this will perform a gl buffer swap.

pygame.display.**update**()

Update portions of the screen for software displays
update(rectangle=None) -> None
update(rectangle_list) -> None

This function is like an optimized version of pygame.display.flip() for software displays. It allows only a portion of the screen to updated, instead of the entire area. If no argument is passed it updates the entire Surface area like pygame.display.flip().

You can pass the function a single rectangle, or a sequence of rectangles. It is more efficient to pass many rectangles at once than to call update multiple times with single or a partial list of rectangles. If passing a sequence of rectangles it is safe to include None values in the list, which will be skipped.

This call cannot be used on pygame.OPENGL displays and will generate an exception.

pygame.display.**get_driver**()

Get the name of the pygame display backend
get_driver() -> name

Pygame chooses one of many available display backends when it is initialized. This returns the internal name used for the display backend. This can be used to provide limited information about what display capabilities might be accelerated. See the SDL_VIDEODRIVER flags in pygame.display.set_mode() to see some of the common options.

pygame.display.**Info**()

Create a video display information object
Info() -> VideoInfo

Creates a simple object containing several attributes to describe the current graphics environment. If this is called before pygame.display.set_mode() some platforms can provide information about the default display mode. This can also be called after setting the display mode to verify specific display options were satisfied. The VidInfo object has several attributes:

```
hw:         True if the display is hardware accelerated
wm:         True if windowed display modes can be used
video_mem:  The megabytes of video memory on the display. This is 0 if unknown
bitsize:    Number of bits used to store each pixel
bytesize:   Number of bytes used to store each pixel
masks:      Four values used to pack RGBA values into pixels
shifts:     Four values used to pack RGBA values into pixels
losses:     Four values used to pack RGBA values into pixels
blit_hw:    True if hardware Surface blitting is accelerated
blit_hw_CC: True if hardware Surface colorkey blitting is accelerated
blit_hw_A:  True if hardware Surface pixel alpha blitting is accelerated
blit_sw:    True if software Surface blitting is accelerated
blit_sw_CC: True if software Surface colorkey blitting is accelerated
blit_sw_A:  True if software Surface pixel alpha blitting is acclerated
current_h, current_h:  Width and height of the current video mode, or of the
  desktop mode if called before the display.set_mode is called.
  (current_h, current_w are available since SDL 1.2.10, and pygame 1.8.0)
  They are -1 on error, or if an old SDL is being used.
```

pygame.display.**get_wm_info**()

Get information about the current windowing system
get_wm_info() -> dict

Creates a dictionary filled with string keys. The strings and values are arbitrarily created by the system. Some systems may have no information and an empty dictionary will be returned. Most platforms will return a "window" key with the value set to the system id for the current display.

New with pygame 1.7.1

pygame.display.**list_modes**()

Get list of available fullscreen modes
list_modes(depth=0, flags=pygame.FULLSCREEN) -> list

This function returns a list of possible dimensions for a specified color depth. The return value will be an empty list if no display modes are available with the given arguments. A return value of -1 means that any requested resolution should work (this is likely the case for windowed modes). Mode sizes are sorted from biggest to smallest.

If depth is 0, SDL will choose the current/best color depth for the display. The flags defaults to pygame.FULLSCREEN, but you may need to add additional flags for specific fullscreen modes.

pygame.display.**mode_ok**()

Pick the best color depth for a display mode
mode_ok(size, flags=0, depth=0) -> depth

This function uses the same arguments as pygame.display.set_mode(). It is used to determine if a requested display mode is available. It will return 0 if the display mode cannot be set. Otherwise it will return a pixel depth that best matches the display asked for.

Usually the depth argument is not passed, but some platforms can support multiple display depths. If passed it will hint to which depth is a better match.

The most useful flags to pass will be pygame.HWSURFACE, pygame.DOUBLEBUF, and maybe pygame.FULLSCREEN. The function will return 0 if these display flags cannot be set.

pygame.display.**gl_get_attribute**()

Get the value for an OpenGL flag for the current display

gl_get_attribute(flag) -> value

After calling pygame.display.set_mode() with the pygame.OPENGL flag, it is a good idea to check the value of any requested OpenGL attributes. See pygame.display.gl_set_attribute() for a list of valid flags.

pygame.display.**gl_set_attribute**()

Request an OpenGL display attribute for the display mode

gl_set_attribute(flag, value) -> None

When calling pygame.display.set_mode() with the pygame.OPENGL flag, Pygame automatically handles setting the OpenGL attributes like color and doublebuffering. OpenGL offers several other attributes you may want control over. Pass one of these attributes as the flag, and its appropriate value. This must be called before pygame.display.set_mode()

The OPENGL flags are;

```
GL_ALPHA_SIZE, GL_DEPTH_SIZE, GL_STENCIL_SIZE, GL_ACCUM_RED_SIZE,
GL_ACCUM_GREEN_SIZE,  GL_ACCUM_BLUE_SIZE, GL_ACCUM_ALPHA_SIZE,
GL_MULTISAMPLEBUFFERS, GL_MULTISAMPLESAMPLES, GL_STEREO
```

pygame.display.**get_active**()

Returns True when the display is active on the display

get_active() -> bool

After pygame.display.set_mode() is called the display Surface will be visible on the screen. Most windowed displays can be hidden by the user. If the display Surface is hidden or iconified this will return False.

pygame.display.**iconify**()

Iconify the display surface

iconify() -> bool

Request the window for the display surface be iconified or hidden. Not all systems and displays support an iconified display. The function will return True if successful.

When the display is iconified pygame.display.get_active() will return False. The event queue should receive a ACTIVEEVENT event when the window has been iconified.

pygame.display.**toggle_fullscreen**()

Switch between fullscreen and windowed displays

toggle_fullscreen() -> bool

Switches the display window between windowed and fullscreen modes. This function only works under the unix x11 video driver. For most situations it is better to call `pygame.display.set_mode()` with new display flags.

`pygame.display.`**`set_gamma`**`()`

Change the hardware gamma ramps
set_gamma(red, green=None, blue=None) -> bool

Set the red, green, and blue gamma values on the display hardware. If the green and blue arguments are not passed, they will both be the same as red. Not all systems and hardware support gamma ramps, if the function succeeds it will return True.

A gamma value of 1.0 creates a linear color table. Lower values will darken the display and higher values will brighten.

`pygame.display.`**`set_gamma_ramp`**`()`

Change the hardware gamma ramps with a custom lookup
set_gamma_ramp(red, green, blue) -> bool

Set the red, green, and blue gamma ramps with an explicit lookup table. Each argument should be sequence of 256 integers. The integers should range between 0 and 0xffff. Not all systems and hardware support gamma ramps, if the function succeeds it will return True.

`pygame.display.`**`set_icon`**`()`

Change the system image for the display window
set_icon(Surface) -> None

Sets the runtime icon the system will use to represent the display window. All windows default to a simple pygame logo for the window icon.

You can pass any surface, but most systems want a smaller image around 32x32. The image can have colorkey transparency which will be passed to the system.

Some systems do not allow the window icon to change after it has been shown. This function can be called before `pygame.display.set_mode()` to create the icon before the display mode is set.

`pygame.display.`**`set_caption`**`()`

Set the current window caption
set_caption(title, icontitle=None) -> None

If the display has a window title, this function will change the name on the window. Some systems support an alternate shorter title to be used for minimized displays.

`pygame.display.`**`get_caption`**`()`

Get the current window caption
get_caption() -> (title, icontitle)

Returns the title and icontitle for the display Surface. These will often be the same value.

`pygame.display.`**`set_palette`**`()`

Set the display color palette for indexed displays
set_palette(palette=None) -> None

This will change the video display color palette for 8bit displays. This does not change the palette for the actual display Surface, only the palette that is used to display the Surface. If no palette argument is passed, the system default palette will be restored. The palette is a sequence of `RGB` triplets.

CHAPTER 6

pygame.draw

pygame module for drawing shapes

Draw several simple shapes to a Surface. These functions will work for rendering to any format of Surface. Rendering to hardware Surfaces will be slower than regular software Surfaces.

Most of the functions take a width argument to represent the size of stroke around the edge of the shape. If a width of 0 is passed the function will actually solid fill the entire shape.

All the drawing functions respect the clip area for the Surface, and will be constrained to that area. The functions return a rectangle representing the bounding area of changed pixels.

Most of the arguments accept a color argument that is an `RGB` triplet. These can also accept an `RGBA` quadruplet. The alpha value will be written directly into the Surface if it contains pixel alphas, but the draw function will not draw transparently. The color argument can also be an integer pixel value that is already mapped to the Surface's pixel format.

These functions must temporarily lock the Surface they are operating on. Many sequential drawing calls can be sped up by locking and unlocking the Surface object around the draw calls.

`pygame.draw.`**`rect`**`()`

draw a rectangle shape
rect(Surface, color, Rect, width=0) -> Rect

Draws a rectangular shape on the Surface. The given Rect is the area of the rectangle. The width argument is the thickness to draw the outer edge. If width is zero then the rectangle will be filled.

Keep in mind the `Surface.fill()` method works just as well for drawing filled rectangles. In fact the `Surface.fill()` can be hardware accelerated on some platforms with both software and hardware display modes.

`pygame.draw.`**`polygon`**`()`

draw a shape with any number of sides
polygon(Surface, color, pointlist, width=0) -> Rect

Draws a polygonal shape on the Surface. The pointlist argument is the vertices of the polygon. The width argument is the thickness to draw the outer edge. If width is zero then the polygon will be filled.

For aapolygon, use aalines with the 'closed' parameter.

pygame.draw.**circle**()

draw a circle around a point
circle(Surface, color, pos, radius, width=0) -> Rect

Draws a circular shape on the Surface. The pos argument is the center of the circle, and radius is the size. The width argument is the thickness to draw the outer edge. If width is zero then the circle will be filled.

pygame.draw.**ellipse**()

draw a round shape inside a rectangle
ellipse(Surface, color, Rect, width=0) -> Rect

Draws an elliptical shape on the Surface. The given rectangle is the area that the circle will fill. The width argument is the thickness to draw the outer edge. If width is zero then the ellipse will be filled.

pygame.draw.**arc**()

draw a partial section of an ellipse
arc(Surface, color, Rect, start_angle, stop_angle, width=1) -> Rect

Draws an elliptical arc on the Surface. The rect argument is the area that the ellipse will fill. The two angle arguments are the initial and final angle in radians, with the zero on the right. The width argument is the thickness to draw the outer edge.

pygame.draw.**line**()

draw a straight line segment
line(Surface, color, start_pos, end_pos, width=1) -> Rect

Draw a straight line segment on a Surface. There are no endcaps, the ends are squared off for thick lines.

pygame.draw.**lines**()

draw multiple contiguous line segments
lines(Surface, color, closed, pointlist, width=1) -> Rect

Draw a sequence of lines on a Surface. The pointlist argument is a series of points that are connected by a line. If the closed argument is true an additional line segment is drawn between the first and last points.

This does not draw any endcaps or miter joints. Lines with sharp corners and wide line widths can have improper looking corners.

pygame.draw.**aaline**()

draw fine antialiased lines
aaline(Surface, color, startpos, endpos, blend=1) -> Rect

Draws an anti-aliased line on a surface. This will respect the clipping rectangle. A bounding box of the affected area is returned as a rectangle. If blend is true, the shades will be be blended with existing pixel shades instead of overwriting them. This function accepts floating point values for the end points.

pygame.draw.**aalines**()

draw a connected sequence of antialiased lines
aalines(Surface, color, closed, pointlist, blend=1) -> Rect

Draws a sequence on a surface. You must pass at least two points in the sequence of points. The closed argument is a simple Boolean and if true, a line will be draw between the first and last points. The Boolean blend argument set to true will blend the shades with existing shades instead of overwriting them. This function accepts floating point values for the end points.

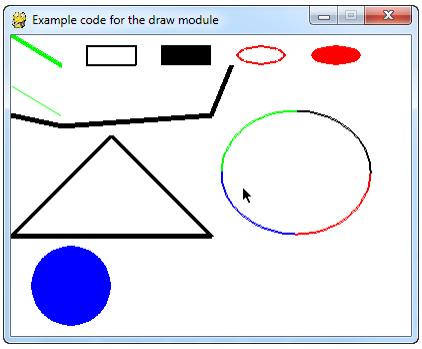

Figure 6.1: Example code for draw module.

```
# Import a library of functions called 'pygame'
import pygame
from math import pi

# Initialize the game engine
pygame.init()

# Define the colors we will use in RGB format
BLACK = (  0,   0,   0)
WHITE = (255, 255, 255)
BLUE =  (  0,   0, 255)
GREEN = (  0, 255,   0)
RED =   (255,   0,   0)

# Set the height and width of the screen
size = [400, 300]
screen = pygame.display.set_mode(size)

pygame.display.set_caption("Example code for the draw module")

#Loop until the user clicks the close button.
done = False
clock = pygame.time.Clock()

while not done:

    # This limits the while loop to a max of 10 times per second.
    # Leave this out and we will use all CPU we can.
    clock.tick(10)

    for event in pygame.event.get(): # User did something
        if event.type == pygame.QUIT: # If user clicked close
            done=True # Flag that we are done so we exit this loop

    # All drawing code happens after the for loop and but
    # inside the main while done==False loop.

    # Clear the screen and set the screen background
```

```
    screen.fill(WHITE)

    # Draw on the screen a GREEN line from (0,0) to (50.75)
    # 5 pixels wide.
    pygame.draw.line(screen, GREEN, [0, 0], [50,30], 5)

    # Draw on the screen a GREEN line from (0,0) to (50.75)
    # 5 pixels wide.
    pygame.draw.lines(screen, BLACK, False, [[0, 80], [50, 90], [200, 80], [220, 30]], 5)

    # Draw on the screen a GREEN line from (0,0) to (50.75)
    # 5 pixels wide.
    pygame.draw.aaline(screen, GREEN, [0, 50],[50, 80], True)

    # Draw a rectangle outline
    pygame.draw.rect(screen, BLACK, [75, 10, 50, 20], 2)

    # Draw a solid rectangle
    pygame.draw.rect(screen, BLACK, [150, 10, 50, 20])

    # Draw an ellipse outline, using a rectangle as the outside boundaries
    pygame.draw.ellipse(screen, RED, [225, 10, 50, 20], 2)

    # Draw an solid ellipse, using a rectangle as the outside boundaries
    pygame.draw.ellipse(screen, RED, [300, 10, 50, 20])

    # This draws a triangle using the polygon command
    pygame.draw.polygon(screen, BLACK, [[100, 100], [0, 200], [200, 200]], 5)

    # Draw an arc as part of an ellipse.
    # Use radians to determine what angle to draw.
    pygame.draw.arc(screen, BLACK,[210, 75, 150, 125], 0, pi/2, 2)
    pygame.draw.arc(screen, GREEN,[210, 75, 150, 125], pi/2, pi, 2)
    pygame.draw.arc(screen, BLUE, [210, 75, 150, 125], pi,3*pi/2, 2)
    pygame.draw.arc(screen, RED,  [210, 75, 150, 125], 3*pi/2, 2*pi, 2)

    # Draw a circle
    pygame.draw.circle(screen, BLUE, [60, 250], 40)

    # Go ahead and update the screen with what we've drawn.
    # This MUST happen after all the other drawing commands.
    pygame.display.flip()

# Be IDLE friendly
pygame.quit()
```

CHAPTER 7

pygame.event

pygame module for interacting with events and queues

Pygame handles all its event messaging through an event queue. The routines in this module help you manage that event queue. The input queue is heavily dependent on the pygame display module. If the display has not been initialized and a video mode not set, the event queue will not really work.

The queue is a regular queue of Event objects, there are a variety of ways to access the events it contains. From simply checking for the existance of events, to grabbing them directly off the stack.

All events have a type identifier. This event type is in between the values of `NOEVENT` and `NUMEVENTS`. All user defined events can have the value of `USEREVENT` or higher. It is recommended make sure your event id's follow this system.

To get the state of various input devices, you can forego the event queue and access the input devices directly with their appropriate modules; mouse, key, and joystick. If you use this method, remember that pygame requires some form of communication with the system window manager and other parts of the platform. To keep pygame in synch with the system, you will need to call `pygame.event.pump()` to keep everything current. You'll want to call this function usually once per game loop.

The event queue offers some simple filtering. This can help performance slightly by blocking certain event types from the queue, use the `pygame.event.set_allowed()` and `pygame.event.set_blocked()` to work with this filtering. All events default to allowed.

The event subsystem should be called from the main thread. If you want to post events into the queue from other threads, please use the fastevent package.

Joysticks will not send any events until the device has been initialized.

An Event object contains an event type and a readonly set of member data. The Event object contains no method functions, just member data. Event objects are retrieved from the pygame event queue. You can create your own new events with the `pygame.event.Event()` function.

Your program must take steps to keep the event queue from overflowing. If the program is not clearing or getting all events off the queue at regular intervals, it can overflow. When the queue overflows an exception is thrown.

All Event objects contain an event type identifier in the `Event.type` member. You may also get full access to the Event's member data through the `Event.dict` method. All other member lookups will be passed through to the Event's dictionary values.

While debugging and experimenting, you can print the Event objects for a quick display of its type and members. Events that come from the system will have a guaranteed set of member items based on the type. Here is a list of the Event members that are defined with each type.

```
QUIT             none
ACTIVEEVENT      gain, state
KEYDOWN          unicode, key, mod
KEYUP            key, mod
MOUSEMOTION      pos, rel, buttons
MOUSEBUTTONUP    pos, button
MOUSEBUTTONDOWN  pos, button
JOYAXISMOTION    joy, axis, value
JOYBALLMOTION    joy, ball, rel
JOYHATMOTION     joy, hat, value
JOYBUTTONUP      joy, button
JOYBUTTONDOWN    joy, button
VIDEORESIZE      size, w, h
VIDEOEXPOSE      none
USEREVENT        code
```

Events support equality comparison. Two events are equal if they are the same type and have identical attribute values. Inequality checks also work.

New in version 1.9.2: On MacOSX, USEREVENT can have *code* = *pygame.USEREVENT_DROPFILE*. That means the user is trying to open a file with your application. The filename can be found at *event.filename*

pygame.event.**pump**()

internally process pygame event handlers

pump() -> None

For each frame of your game, you will need to make some sort of call to the event queue. This ensures your program can internally interact with the rest of the operating system. If you are not using other event functions in your game, you should call `pygame.event.pump()` to allow pygame to handle internal actions.

This function is not necessary if your program is consistently processing events on the queue through the other `pygame.event` functions.

There are important things that must be dealt with internally in the event queue. The main window may need to be repainted or respond to the system. If you fail to make a call to the event queue for too long, the system may decide your program has locked up.

pygame.event.**get**()

get events from the queue

get() -> Eventlist

get(type) -> Eventlist

get(typelist) -> Eventlist

This will get all the messages and remove them from the queue. If a type or sequence of types is given only those messages will be removed from the queue.

If you are only taking specific events from the queue, be aware that the queue could eventually fill up with the events you are not interested.

pygame.event.**poll**()

get a single event from the queue

poll() -> Event

Returns a single event from the queue. If the event queue is empty an event of type `pygame.NOEVENT` will be returned immediately. The returned event is removed from the queue.

pygame.event.**wait**()

wait for a single event from the queue
wait() -> Event

Returns a single event from the queue. If the queue is empty this function will wait until one is created. The event is removed from the queue once it has been returned. While the program is waiting it will sleep in an idle state. This is important for programs that want to share the system with other applications.

pygame.event.**peek**()

test if event types are waiting on the queue
peek(type) -> bool
peek(typelist) -> bool

Returns true if there are any events of the given type waiting on the queue. If a sequence of event types is passed, this will return True if any of those events are on the queue.

pygame.event.**clear**()

remove all events from the queue
clear() -> None
clear(type) -> None
clear(typelist) -> None

Remove all events or events of a specific type from the queue. This has the same effect as pygame.event.get() except nothing is returned. This can be slightly more effecient when clearing a full event queue.

pygame.event.**event_name**()

get the string name from and event id
event_name(type) -> string

Pygame uses integer ids to represent the event types. If you want to report these types to the user they should be converted to strings. This will return a the simple name for an event type. The string is in the WordCap style.

pygame.event.**set_blocked**()

control which events are allowed on the queue
set_blocked(type) -> None
set_blocked(typelist) -> None
set_blocked(None) -> None

The given event types are not allowed to appear on the event queue. By default all events can be placed on the queue. It is safe to disable an event type multiple times.

If None is passed as the argument, this has the opposite effect and ALL of the event types are allowed to be placed on the queue.

pygame.event.**set_allowed**()

control which events are allowed on the queue
set_allowed(type) -> None
set_allowed(typelist) -> None

set_allowed(None) -> None

The given event types are allowed to appear on the event queue. By default all events can be placed on the queue. It is safe to enable an event type multiple times.

If None is passed as the argument, NONE of the event types are allowed to be placed on the queue.

pygame.event.**get_blocked**()

test if a type of event is blocked from the queue
get_blocked(type) -> bool

Returns true if the given event type is blocked from the queue.

pygame.event.**set_grab**()

control the sharing of input devices with other applications
set_grab(bool) -> None

When your program runs in a windowed environment, it will share the mouse and keyboard devices with other applications that have focus. If your program sets the event grab to True, it will lock all input into your program.

It is best to not always grab the input, since it prevents the user from doing other things on their system.

pygame.event.**get_grab**()

test if the program is sharing input devices
get_grab() -> bool

Returns true when the input events are grabbed for this application. Use pygame.event.set_grab() to control this state.

pygame.event.**post**()

place a new event on the queue
post(Event) -> None

This places a new event at the end of the event queue. These Events will later be retrieved from the other queue functions.

This is usually used for placing pygame.USEREVENT events on the queue. Although any type of event can be placed, if using the sytem event types your program should be sure to create the standard attributes with appropriate values.

pygame.event.**Event**()

create a new event object
Event(type, dict) -> Event
Event(type, **attributes) -> Event

Creates a new event with the given type. The event is created with the given attributes and values. The attributes can come from a dictionary argument with string keys, or from keyword arguments. The event object exposes its dictionary as attribute __dict__, and also as dict for backward compatibility.

Attributes type, __dict__, and dict are readonly. Other attributes are mutable. There are no methods attached to an Event object.

Mutable attributes are new to Pygame 1.9.2.

CHAPTER 8

pygame.examples

module of example programs

These examples should help get you started with pygame. Here is a brief rundown of what you get. The source code for these examples is in the public domain. Feel free to use for your own projects.

There are several ways to run the examples. First they can be run as stand-alone programs. Second they can be imported and their `main()` methods called (see below). Finally, the easiest way is to use the python -m option:

```
python -m pygame.examples.<example name> <example arguments>
```

eg:

```
python -m pygame.examples.scaletest someimage.png
```

Resources such as images and sounds for the examples are found in the pygame/examples/data subdirectory.

You can find where the example files are installed by using the following commands inside the python interpreter.

```
>>> import pygame.examples.scaletest
>>> pygame.examples.scaletest.__file__
'/usr/lib/python2.6/site-packages/pygame/examples/scaletest.py'
```

On each OS and version of python the location will be slightly different. For example on windows it might be in 'C:/Python26/Lib/site-packages/pygame/examples/' On Mac OS X it might be in '/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/site-packages/pygame/examples/'

You can also run the examples in the python interpreter by calling each modules main() function.

```
>>> import pygame.examples.scaletest
>>> pygame.examples.scaletest.main()
```

We're always on the lookout for more examples and/or example requests. Code like this is probably the best way to start getting involved with python gaming.

examples as a package is new to pygame 1.9.0. But most of the examples came with pygame much earlier.

`aliens.main()`

play the full aliens example
aliens.main() -> None

This started off as a port of the `SDL` demonstration, Aliens. Now it has evolved into something sort of resembling fun. This demonstrates a lot of different uses of sprites and optimized blitting. Also transparency, colorkeys, fonts, sound, music, joystick, and more. (PS, my high score is 117! goodluck)

oldalien.**main**()

play the original aliens example
oldalien.main() -> None

This more closely resembles a port of the SDL Aliens demo. The code is a lot simpler, so it makes a better starting point for people looking at code for the first times. These blitting routines are not as optimized as they should/could be, but the code is easier to follow, and it plays quick enough.

stars.**main**()

run a simple starfield example
stars.main() -> None

A simple starfield example. You can change the center of perspective by leftclicking the mouse on the screen.

chimp.**main**()

hit the moving chimp
chimp.main() -> None

This simple example is derived from the line-by-line tutorial that comes with pygame. It is based on a 'popular' web banner. Note there are comments here, but for the full explanation, follow along in the tutorial.

moveit.**main**()

display animated objects on the screen
moveit.main() -> None

This is the full and final example from the Pygame Tutorial, "How Do I Make It Move". It creates 10 objects and animates them on the screen.

Note it's a bit scant on error checking, but it's easy to read. :] Fortunately, this is python, and we needn't wrestle with a pile of error codes.

fonty.**main**()

run a font rendering example
fonty.main() -> None

Super quick, super simple application demonstrating the different ways to render fonts with the font module

vgrade.**main**()

display a vertical gradient
vgrade.main() -> None

Demonstrates creating a vertical gradient with pixelcopy and NumPy python. The app will create a new gradient every half second and report the time needed to create and display the image. If you're not prepared to start working with the NumPy arrays, don't worry about the source for this one :]

eventlist.**main**()

display pygame events
eventlist.main() -> None

Eventlist is a sloppy style of pygame, but is a handy tool for learning about pygame events and input. At the top of the screen are the state of several device values, and a scrolling list of events are displayed on the bottom.

This is not quality 'ui' code at all, but you can see how to implement very non-interactive status displays, or even a crude text output control.

`arraydemo.main()`

show various surfarray effects
arraydemo.main(arraytype=None) -> None

Another example filled with various surfarray effects. It requires the surfarray and image modules to be installed. This little demo can also make a good starting point for any of your own tests with surfarray

If arraytype is provided then use that array package. Valid values are 'numeric' or 'numpy'. Otherwise default to NumPy, or fall back on Numeric if NumPy is not installed. As a program `surfarray.py` accepts an optional –numeric or –numpy flag. (New pygame 1.9.0)

`sound.main()`

load and play a sound
sound.main(file_path=None) -> None

Extremely basic testing of the mixer module. Load a sound and play it. All from the command shell, no graphics.

If provided, use the audio file 'file_path', otherwise use a default file.

`sound.py` optional command line argument: an audio file

`sound_array_demos.main()`

play various sndarray effects
sound_array_demos.main(arraytype=None) -> None

If arraytype is provided then use that array package. Valid values are 'numeric' or 'numpy'. Otherwise default to NumPy, or fall back on Numeric if NumPy is not installed.

Uses sndarray and NumPy (or Numeric) to create offset faded copies of the original sound. Currently it just uses hardcoded values for the number of echos and the delay. Easy for you to recreate as needed. Run as a program `sound_array_demos.py` takes an optional command line option, –numpy or –numeric, specifying which array package to use.

`liquid.main()`

display an animated liquid effect
liquid.main() -> None

This example was created in a quick comparison with the BlitzBasic gaming language. Nonetheless, it demonstrates a quick 8-bit setup (with colormap).

`glcube.main()`

display an animated 3D cube using OpenGL
glcube.main() -> None

Using PyOpenGL and pygame, this creates a spinning 3D multicolored cube.

`scrap_clipboard.main()`

access the clipboard
scrap_clipboard.main() -> None

A simple demonstration example for the clipboard support.

mask.**main**()

display multiple images bounce off each other using collision detection
mask.main(*args) -> None

Positional arguments:

```
one or more image file names.
```

This pygame.masks demo will display multiple moving sprites bouncing off each other. More than one sprite image can be provided.

If run as a program then mask.py takes one or more image files as command line arguments.

testsprite.**main**()

show lots of sprites moving around
testsprite.main(update_rects = True, use_static = False, use_FastRenderGroup = False, screen_dims = [640, 480], use_alpha = False, flags = 0) -> None

Optional keyword arguments:

```
update_rects - use the RenderUpdate sprite group class
use_static - include non-moving images
use_FastRenderGroup - Use the FastRenderGroup sprite group
screen_dims - pygame window dimensions
use_alpha - use alpha blending
flags - additional display mode flags
```

Like the testsprite.c that comes with sdl, this pygame version shows lots of sprites moving around.

If run as a stand-alone program then no command line arguments are taken.

headless_no_windows_needed.**main**()

write an image file that is smoothscaled copy of an input file
headless_no_windows_needed.main(fin, fout, w, h) -> None

arguments:

```
fin - name of an input image file
fout - name of the output file to create/overwrite
w, h - size of the rescaled image, as integer width and height
```

How to use pygame with no windowing system, like on headless servers.

Thumbnail generation with scaling is an example of what you can do with pygame.

NOTE: the pygame scale function uses mmx/sse if available, and can be run in multiple threads.

If headless_no_windows_needed.py is run as a program it takes the following command line arguments:

```
-scale inputimage outputimage new_width new_height
eg. -scale in.png outpng 50 50
```

fastevents.**main**()

stress test the fastevents module
fastevents.main() -> None

This is a stress test for the fastevents module.

- Fast events does not appear faster!
-

So far it looks like normal pygame.event is faster by up to two times. So maybe fastevent isn't fast at all.

Tested on windowsXP sp2 athlon, and freebsd.

However... on my debian duron 850 machine fastevents is faster.

overlay.**main**()

play a .pgm video using overlays
overlay.main(fname) -> None

Play the .pgm video fila a path fname.

If run as a program overlay.py takes the file name as a command line argument.

blend_fill.**main**()

demonstrate the various surface.fill method blend options
blend_fill.main() -> None

A interactive demo that lets one choose which BLEND_xxx option to apply to a surface.

blit_blends.**main**()

uses alternative additive fill to that of surface.fill
blit_blends.main() -> None

Fake additive blending. Using NumPy. it doesn't clamp. Press r,g,b Somewhat like blend_fill.

cursors.**main**()

display two different custom cursors
cursors.main() -> None

Display an arrow or circle with crossbar cursor.

pixelarray.**main**()

display various pixelarray generated effects
pixelarray.main() -> None

Display various pixelarray generated effects.

scaletest.**main**()

interactively scale an image using smoothscale
scaletest.main(imagefile, convert_alpha=False, run_speed_test=True) -> None

arguments:

```
imagefile - file name of source image (required)
convert_alpha - use convert_alpha() on the surf (default False)
run_speed_test - (default False)
```

A smoothscale example that resized an image on the screen. Vertical and horizontal arrow keys are used to change the width and height of the displayed image. If the convert_alpha option is True then the source image is forced to have source alpha, whether or not the original images does. If run_speed_test is True then a background timing test is performed instead of the interactive scaler.

If `scaletest.py` is run as a program then the command line options are:

```
ImageFile [-t] [-convert_alpha]
[-t] = Run Speed Test
[-convert_alpha] = Use convert_alpha() on the surf.
```

`midi.`**`main`**`()`

run a midi example
midi.main(mode='output', device_id=None) -> None

Arguments:

```
mode - if 'output' run a midi keyboard output example
          'input' run a midi event logger input example
          'list' list available midi devices
       (default 'output')
device_id - midi device number; if None then use the default midi input or
            output device for the system
```

The output example shows how to translate mouse clicks or computer keyboard events into midi notes. It implements a rudimentary button widget and state machine.

The input example shows how to translate midi input to pygame events.

With the use of a virtual midi patch cord the output and input examples can be run as separate processes and connected so the keyboard output is displayed on a console.

new to pygame 1.9.0

`scroll.`**`main`**`()`

run a Surface.scroll example that shows a magnified image
scroll.main(image_file=None) -> None

This example shows a scrollable image that has a zoom factor of eight. It uses the `Surface.scroll()` function to shift the image on the display surface. A clip rectangle protects a margin area. If called as a function, the example accepts an optional image file path. If run as a program it takes an optional file path command line argument. If no file is provided a default image file is used.

When running click on a black triangle to move one pixel in the direction the triangle points. Or use the arrow keys. Close the window or press `ESC` to quit.

`movieplayer.`**`main`**`()`

play an MPEG movie
movieplayer.main(filepath) -> None

A simple movie player that plays an MPEG movie in a Pygame window. It showcases the pygame.movie module. The window adjusts to the size of the movie image. It is given a border to demonstrate that a movie can play autonomously in a sub- window. Also, the file is copied to a file like object to show that not just Python files can be used as a movie source.

The pygame.movie module is problematic and may not work on all systems. It is intended to replace it with an ffmpeg based version.

camera.**main**()

display video captured live from an attached camera

camera.main() -> None

A simple live video player, it uses the first available camera it finds on the system.

CHAPTER 9

pygame.font

pygame module for loading and rendering fonts

The font module allows for rendering TrueType fonts into a new Surface object. It accepts any UCS-2 character ('u0001' to 'uFFFF'). This module is optional and requires SDL_ttf as a dependency. You should test that pygame.font is available and initialized before attempting to use the module.

Most of the work done with fonts are done by using the actual Font objects. The module by itself only has routines to initialize the module and create Font objects with pygame.font.Font().

You can load fonts from the system by using the pygame.font.SysFont() function. There are a few other functions to help lookup the system fonts.

Pygame comes with a builtin default font. This can always be accessed by passing None as the font name.

To use the pygame.freetype based pygame.ftfont as pygame.font define the enviroment variable PYGAME_FREETYPE before the first import of pygame. pygame.ftfont is a pygame.font compatible module that passes all but one of the font module unit tests: it does not have the UCS-2 limitation of the SDL_ttf based font module, so fails to raise an exception for a code point greater than 'uFFFF'. If pygame.freetype is unavailable then the SDL_ttf font module will be loaded instead.

pygame.font.**init**()

initialize the font module
init() -> None

This method is called automatically by pygame.init(). It initializes the font module. The module must be initialized before any other functions will work.

It is safe to call this function more than once.

pygame.font.**quit**()

uninitialize the font module
quit() -> None

Manually uninitialize SDL_ttf's font system. This is called automatically by pygame.quit().

It is safe to call this function even if font is currently not initialized.

pygame.font.**get_init**()

true if the font module is initialized

get_init() -> bool

Test if the font module is initialized or not.

pygame.font.**get_default_font**()

get the filename of the default font
get_default_font() -> string

Return the filename of the system font. This is not the full path to the file. This file can usually be found in the same directory as the font module, but it can also be bundled in separate archives.

pygame.font.**get_fonts**()

get all available fonts
get_fonts() -> list of strings

Returns a list of all the fonts available on the system. The names of the fonts will be set to lowercase with all spaces and punctuation removed. This works on most systems, but some will return an empty list if they cannot find fonts.

pygame.font.**match_font**()

find a specific font on the system
match_font(name, bold=False, italic=False) -> path

Returns the full path to a font file on the system. If bold or italic are set to true, this will attempt to find the correct family of font.

The font name can actually be a comma separated list of font names to try. If none of the given names are found, None is returned.

Example:

```
print pygame.font.match_font('bitstreamverasans')
# output is: /usr/share/fonts/truetype/ttf-bitstream-vera/Vera.ttf
# (but only if you have Vera on your system)
```

pygame.font.**SysFont**()

create a Font object from the system fonts
SysFont(name, size, bold=False, italic=False) -> Font

Return a new Font object that is loaded from the system fonts. The font will match the requested bold and italic flags. If a suitable system font is not found this will fallback on loading the default pygame font. The font name can be a comma separated list of font names to look for.

class pygame.font.**Font**

create a new Font object from a file
Font(filename, size) -> Font
Font(object, size) -> Font

Load a new font from a given filename or a python file object. The size is the height of the font in pixels. If the filename is None the Pygame default font will be loaded. If a font cannot be loaded from the arguments given an exception will be raised. Once the font is created the size cannot be changed.

Font objects are mainly used to render text into new Surface objects. The render can emulate bold or italic features, but it is better to load from a font with actual italic or bold glyphs. The rendered text can be regular strings or unicode.

render()

draw text on a new Surface

render(text, antialias, color, background=None) -> Surface

This creates a new Surface with the specified text rendered on it. Pygame provides no way to directly draw text on an existing Surface: instead you must use `Font.render()` to create an image (Surface) of the text, then blit this image onto another Surface.

The text can only be a single line: newline characters are not rendered. Null characters ('x00') raise a TypeError. Both Unicode and char (byte) strings are accepted. For Unicode strings only UCS-2 characters ('u0001' to 'uFFFF') are recognized. Anything greater raises a UnicodeError. For char strings a `LATIN1` encoding is assumed. The antialias argument is a boolean: if true the characters will have smooth edges. The color argument is the color of the text [e.g.: (0,0,255) for blue]. The optional background argument is a color to use for the text background. If no background is passed the area outside the text will be transparent.

The Surface returned will be of the dimensions required to hold the text. (the same as those returned by Font.size()). If an empty string is passed for the text, a blank surface will be returned that is one pixel wide and the height of the font.

Depending on the type of background and antialiasing used, this returns different types of Surfaces. For performance reasons, it is good to know what type of image will be used. If antialiasing is not used, the return image will always be an 8bit image with a two color palette. If the background is transparent a colorkey will be set. Antialiased images are rendered to 24-bit `RGB` images. If the background is transparent a pixel alpha will be included.

Optimization: if you know that the final destination for the text (on the screen) will always have a solid background, and the text is antialiased, you can improve performance by specifying the background color. This will cause the resulting image to maintain transparency information by colorkey rather than (much less efficient) alpha values.

If you render 'n' a unknown char will be rendered. Usually a rectangle. Instead you need to handle new lines yourself.

Font rendering is not thread safe: only a single thread can render text at any time.

size()

determine the amount of space needed to render text

size(text) -> (width, height)

Returns the dimensions needed to render the text. This can be used to help determine the positioning needed for text before it is rendered. It can also be used for wordwrapping and other layout effects.

Be aware that most fonts use kerning which adjusts the widths for specific letter pairs. For example, the width for "ae" will not always match the width for "a" + "e".

set_underline()

control if text is rendered with an underline

set_underline(bool) -> None

When enabled, all rendered fonts will include an underline. The underline is always one pixel thick, regardless of font size. This can be mixed with the bold and italic modes.

get_underline()

check if text will be rendered with an underline
get_underline() -> bool

Return True when the font underline is enabled.

set_bold()

enable fake rendering of bold text
set_bold(bool) -> None

Enables the bold rendering of text. This is a fake stretching of the font that doesn't look good on many font types. If possible load the font from a real bold font file. While bold, the font will have a different width than when normal. This can be mixed with the italic and underline modes.

get_bold()

check if text will be rendered bold
get_bold() -> bool

Return True when the font bold rendering mode is enabled.

set_italic()

enable fake rendering of italic text
set_italic(bool) -> None

Enables fake rendering of italic text. This is a fake skewing of the font that doesn't look good on many font types. If possible load the font from a real italic font file. While italic the font will have a different width than when normal. This can be mixed with the bold and underline modes.

metrics()

Gets the metrics for each character in the pased string.
metrics(text) -> list

The list contains tuples for each character, which contain the minimum `X` offset, the maximum `X` offset, the minimum `Y` offset, the maximum `Y` offset and the advance offset (bearing plus width) of the character. [(minx, maxx, miny, maxy, advance), (minx, maxx, miny, maxy, advance), ...]. None is entered in the list for each unrecognized character.

get_italic()

check if the text will be rendered italic
get_italic() -> bool

Return True when the font italic rendering mode is enabled.

get_linesize()

get the line space of the font text
get_linesize() -> int

Return the height in pixels for a line of text with the font. When rendering multiple lines of text this is the recommended amount of space between lines.

get_height()

get the height of the font
get_height() -> int

Return the height in pixels of the actual rendered text. This is the average size for each glyph in the font.

get_ascent()

get the ascent of the font
get_ascent() -> int

Return the height in pixels for the font ascent. The ascent is the number of pixels from the font baseline to the top of the font.

get_descent()

get the descent of the font
get_descent() -> int

Return the height in pixels for the font descent. The descent is the number of pixels from the font baseline to the bottom of the font.

CHAPTER 10

pygame.freetype

Enhanced Pygame module for loading and rendering computer fonts

— Note that some features may change before a formal release

The pygame.freetype module allows for the rendering of all font file formats supported by FreeType, namely TTF, Type1, CFF, OpenType, SFNT, PCF, FNT, BDF, PFR and Type42 fonts. It can render any UTF-32 character in a font file.

This module is a replacement for pygame.font. It has all of the functionality of the original, plus many new features. Yet is has absolutely no dependencies on the SDL_ttf library. The pygame.freetype module is not itself backward compatible with pygame.font. Instead, a new pygame.ftfont provides a drop-in replacement for pygame.font.

Most of the work done with fonts is done by using the actual Font objects. The module by itself only has routines to initialize itself and create Font objects with pygame.freetype.Font().

You can load fonts from the system by using the pygame.freetype.SysFont() function. There are a few other functions to help find system fonts.

For now undefined character codes are replaced with the undefined character. How undefined codes are handled may become configurable in a future release.

Pygame comes with a builtin default font. This can always be accessed by passing None as the font name to the Font constructor.

New in Pygame 1.9.2

pygame.freetype.**get_error**()

Return the latest FreeType2 error
get_error() -> str

Return a description of the last error which occurred in the FreeType2 library, or None if no errors have occurred.

pygame.freetype.**get_version**()

Return the FreeType 2 version
get_version() -> (int, int, int)

Returns the version of the FreeType2 library which was used to build the 'freetype' module.

Note that the freetype module depends on the FreeType 2 library. It will not compile with the original FreeType 1.0. Hence, the first element of the tuple will always be "2".

pygame.freetype.**init**()

Initialize the underlying FreeType 2 library.

init(cache_size=64, resolution=72) -> None

This function initializes the underlying FreeType 2 library and must be called before trying to use any of the functionality of the 'freetype' module.

However, this function will be automatically called by `pygame.init()`. It is safe to call this function more than once.

Optionally, you may specify a default size for the Glyph cache: this is the maximum number of glyphs that will be cached at any given time by the module. Exceedingly small values will be automatically tuned for performance. Also a default pixel resolution, in dots per inch, can be given to adjust font scaling.

pygame.freetype.**quit**()

Shut down the underlying FreeType 2 library.

quit() -> None

This function de-initializes the `freetype` module. After calling this function, you should not invoke any class, method or function related to the `freetype` module as they are likely to fail or might give unpredictable results. It is safe to call this function even if the module hasn't been initialized yet.

pygame.freetype.**was_init**()

Return whether the the FreeType 2 library is initialized.

was_init() -> bool

Returns whether the the FreeType 2 library is initialized.

pygame.freetype.**get_default_resolution**()

Return the default pixel size in dots per inch

get_default_resolution() -> long

Returns the default pixel size, in dots per inch for the module. If not changed it will be 72.

pygame.freetype.**set_default_resolution**()

Set the default pixel size in dots per inch for the module

set_default_resolution([resolution]) -> None

Set the default pixel size, in dots per inch, for the module. If the optional argument is omitted or zero the resolution is reset to 72.

pygame.freetype.**get_default_font**()

Get the filename of the default font

get_default_font() -> string

Return the filename of the system font. This is not the full path to the file. This file can usually be found in the same directory as the font module, but it can also be bundled in separate archives.

class pygame.freetype.**Font**

Create a new Font instance from a supported font file.
Font(file, style=STYLE_NONE, ptsize=-1, font_index=0, vertical=0, ucs4=0, resolution=0) -> Font

Argument *file* can be either a string representing the font's filename, a file-like object containing the font, or None; if None, the default, built-in font is used.

Optionally, a *ptsize* argument may be specified to set the default size in points, which will be used when rendering the font. The size can also be passed explicitly to each method call. Because of the way the caching system works, specifying a default size on the constructor doesn't imply a performance gain over manually passing the size on each function call.

If the font file has more than one font, the font to load can be chosen with the *index* argument. An exception is raised for an out-of-range font index value.

The *style* argument will set the default style (oblique, underline, strong) used to draw this font. This style may be overridden on any `Font.render()` call.

The optional vertical argument, an integer, sets the default orientation for the font: 0 (False) for horizontal, any other value (True) for vertical. See `Font.vertical`.

The optional ucs4 argument, an integer, sets the default text translation mode: 0 (False) recognize UTF-16 surrogate pairs, any other value (True), to treat Unicode text as UCS-4, with no surrogate pairs. See `Font.ucs4`.

The optional resolution argument sets the pixel size, in dots per inch, for use in scaling glyphs for this Font instance. If 0 then the default module value, set by `freetype.init()`, is used. The Font object's resolution can only be changed by reinitializing the Font instance.

name

Proper font name.
name -> string

Read only. Returns the real (long) name of the font, as recorded in the font file.

path

Font file path
path -> unicode

Read only. Returns the path of the loaded font file

get_rect()

Return the size and offset of rendered text
get_rect(text, style=STYLE_DEFAULT, rotation=0, ptsize=default) -> rect

Gets the final dimensions and origin, in pixels, of 'text' using the current point size, style, rotation and orientation. These are either taken from the arguments, if given, else from the default values set for the font object.

Returns a rect containing the width and height of the text's bounding box and the position of the text's origin. The origin can be used to align separately rendered pieces of text. It gives the baseline position and bearing at the start of the text.

If text is a char (byte) string, then its encoding is assumed to be `LATIN1`.

get_metrics()

Return the glyph metrics for the given text

get_metrics(text, ptsize=default) -> [(...), ...]

Returns the glyph metrics for each character in 'text'.

The glyph metrics are returned inside a list; each character will be represented as a tuple inside the list with the following values:

```
(min_x, max_x, min_y, max_y, horizontal_advance_x, horizontal_advance_y)
```

The bounding box min_x, max_y, min_y, and max_y values are returned as grid-fitted pixel coordinates of type int. The advance values are float values.

The calculations are done using the font's default size in points. Optionally you may specify another point size to use.

The metrics are adjusted for the current rotation, strong, and oblique settings.

If text is a char (byte) string, then its encoding is assumed to be `LATIN1`.

height

The unscaled height of the font in font units
height -> int

Read only. Gets the height of the font. This is the average value of all glyphs in the font.

ascender()

The unscaled ascent of the font in font units
ascender -> int

Read only. Return the number of units from the font's baseline to the top of the bounding box.

descender

The unscaled descent of the font in font units
descender -> int

Read only. Return the height in font units for the font descent. The descent is the number of units from the font's baseline to the bottom of the bounding box.

get_sized_ascender

The scaled ascent of the font in pixels
get_sized_ascender() -> int

Return the number of units from the font's baseline to the top of the bounding box. It is not adjusted for strong or rotation.

get_sized_descender()

The scaled descent of the font in pixels
get_sized_descender() -> int

Return the number of pixels from the font's baseline to the top of the bounding box. It is not adjusted for strong or rotation.

get_sized_height

The scaled height of the font in pixels

get_sized_height() -> int

Read only. Gets the height of the font. This is the average value of all glyphs in the font. It is not adjusted for strong or rotation.

get_sized_glyph_height()

The scaled bounding box height of the font in pixels

get_sized_glyph_height() -> int

Return the glyph bounding box height of the font in pixels. This is the average value of all glyphs in the font. It is not adjusted for strong or rotation.

render()

Return rendered text as a surface

render(text, fgcolor, bgcolor=None, style=STYLE_DEFAULT, rotation=0, ptsize=default) -> (Surface, Rect)

Returns a new `pygame.Surface`, with the text rendered to it in the color given by 'fgcolor'. If `bgcolor` is given, the surface will be filled with this color. If no background color is given, the surface is filled with zero alpha opacity. Normally the returned surface has a 32 bit pixel size. However, if `bgcolor` is `None` and anti-aliasing is disabled a two color 8 bit surface with colorkey set for the background color is returned.

The return value is a tuple: the new surface and the bounding rectangle giving the size and origin of the rendered text.

If an empty string is passed for text then the returned Rect is zero width and the height of the font. If dest is None the returned surface is the same dimensions as the boundary rect. The rect will test False.

The rendering is done using the font's default size in points and its default style, without any rotation, and taking into account fonts which are set to be drawn vertically via the `Font.vertical()` attribute. Optionally you may specify another point size to use via the 'ptsize' argument, a text rotation via the 'rotation' argument, or a new text style via the 'style' argument.

If text is a char (byte) string, then its encoding is assumed to be `LATIN1`.

render_to()

Render text onto an existing surface

render(surf, dest, text, fgcolor, bgcolor=None, style=STYLE_DEFAULT, rotation=0, ptsize=default) -> Rect

Renders the string 'text' to a `pygame.Surface` 'surf', using the color 'fgcolor'.

Argument 'dest' is an (x, y) surface coordinate pair. If either x or y is not an integer it is converted to one if possible. Any sequence, including Rect, for which the first two elements are positions x and y is accepted.

If a background color is given, the surface is first filled with that color. The text is blitted next. Both the background fill and text rendering involve full alpha blits. That is, the alpha values of both the foreground and background colors, as well as those of the destination surface if it has per-pixel alpha.

The return value is a rectangle giving the size and position of the rendered text within the surface.

If an empty string is passed for text then the returned Rect is zero width and the height of the font. The rect will test False.

By default, the point size and style set for the font are used if not passed as arguments. The text is unrotated unless a non-zero rotation value is given.

If text is a char (byte) string, then its encoding is assumed to be `LATIN1`.

render_raw()

Return rendered text as a string of bytes

render_raw(text, style=STYLE_DEFAULT, rotation=0, ptsize=default, invert=False) -> (bytes, (int, int))

Like `Font.render()` but the tuple returned is an 8 bit monochrome string of bytes and its size. The foreground color is 255, the background 0, useful as an alpha mask for a foreground pattern.

render_raw_to()

Render text into an array of ints

render_raw_to(array, text, dest=None, style=STYLE_DEFAULT, rotation=0, ptsize=default, invert=False) -> (int, int)

Render to an array object exposing an array struct interface. The array must be two dimensional with integer items. The default dest value, None, is equivalent to (0, 0).

style

The font's style flags

style <-> int

Gets or sets the default style of the Font. This default style will be used for all text rendering and size calculations unless overridden specifically in the 'render()' or 'get_size()' calls. The style value may be a bit-wise `OR` of one or more of the following constants:

```
STYLE_NONE
STYLE_UNDERLINE
STYLE_OBLIQUE
STYLE_STRONG
STYLE_WIDE
```

These constants may be found on the FreeType constants module. Optionally, the default style can be modified or obtained accessing the individual style attributes (underline, oblique, strong).

underline

The state of the font's underline style flag

underline <-> bool

Gets or sets whether the font will be underlined when drawing text. This default style value will be used for all text rendering and size calculations unless overridden specifically in the 'render()' or 'get_size()' calls, via the 'style' parameter.

strong

The state of the font's strong style flag

strong <-> bool

Gets or sets whether the font will be bold when drawing text. This default style value will be used for all text rendering and size calculations unless overridden specifically in the 'render()' or 'get_size()' calls, via the 'style' parameter.

oblique

The state of the font's oblique style flag

oblique <-> bool

Gets or sets whether the font will be rendered as oblique. This default style value will be used for all text rendering and size calculations unless overridden specifically in the 'render()' or 'get_size()' calls, via the 'style' parameter.

wide

The state of the font's wide style flag

wide <-> bool

Gets or sets whether the font will be stretched horizontally when drawing text. It produces a result similar to font.Font's bold. This style is only available for unrotated text.

strength

The strength associated with the strong or wide font styles

strength <-> float

The amount by which a font glyph's size is enlarged for the strong or wide transformations, as a fraction of the untransformed size. For the wide style only the horizontal dimension is increased. For strong text both the horizontal and vertical dimensions are enlarged. A wide style of strength 1/12 is equivalent to the font.Font bold style. The default is 1/36.

underline_adjustment

Adjustment factor for the underline position

underline_adjustment <-> float

Gets or sets a factor which, when positive, is multiplied with the font's underline offset to adjust the underline position. A negative value turns an underline into a strike-through or overline. It is multiplied with the ascender. Accepted values are between -2.0 and 2.0 inclusive. A value of 0.5 closely matches Tango underlining. A value of 1.0 mimics SDL_ttf.

fixed_width

Gets whether the font is fixed-width

fixed_width -> bool

Read only. Returns whether this Font is a fixed-width (bitmap) font.

Note that scalable fonts whose glyphs are all the same width (i.e. monospace `TTF` fonts used for programming) are not considered fixed width.

antialiased

Font anti-aliasing mode

antialiased <-> bool

Gets or sets the font's anti-aliasing mode. This defaults to `True` on all fonts, which are rendered with full 8 bit blending.

Setting this to `False` will enable monochrome rendering. This should provide a small speed gain and reduce cache memory size.

kerning

Character kerning mode
kerning -> bool

Gets or sets the font's kerning mode. This defaults to False on all fonts, which will be rendered by default without kerning.

Setting this to true will change all rendering methods to do kerning between character pairs for surface size calculation and all render operations.

vertical

Font vertical mode
vertical -> bool

Gets or sets whether the font is a vertical font such as fonts in fonts representing Kanji glyphs or other styles of vertical writing.

Changing this attribute will cause the font to be rendering vertically, and affects all other methods which manage glyphs or text layouts to use vertical metrics accordingly.

Note that the FreeType library doesn't automatically detect whether a font contains glyphs which are always supposed to be drawn vertically, so this attribute must be set manually by the user.

Also note that several font formats (especially bitmap based ones) don't contain the necessary metrics to draw glyphs vertically, so drawing in those cases will give unspecified results.

origin

Font render to text origin mode
vertical -> bool

If set True, then when rendering to an existing surface, the position is taken to be that of the text origin. Otherwise the render position is the top-left corner of the text bounding box.

pad

padded boundary mode
pad -> bool

If set True, then the text boundary rectangle will be inflated to match that of font.Font. Otherwise, the boundary rectangle is just large enough for the text.

ucs4

Enable UCS-4 mode
ucs4 <-> bool

Gets or sets the decoding of Unicode text. By default, the freetype module performs UTF-16 surrogate pair decoding on Unicode text. This allows 32-bit escape sequences ('Uxxxxxxxx') between 0x10000 and 0x10FFFF to represent their corresponding UTF-32 code points on Python interpreters built with a UCS-2 unicode type (on Windows, for instance). It also means character values within the UTF-16 surrogate area (0xD800 to 0xDFFF) are considered part of a surrogate pair. A malformed surrogate pair will raise

an UnicodeEncodeError. Setting ucs4 True turns surrogate pair decoding off, letting interpreters with a UCS-4 unicode type access the full UCS-4 character range.

resolution

Pixel resolution in dots per inch
resolution -> int

Gets the pixel size used in scaling font glyphs for this Font instance.

CHAPTER 11

pygame.gfxdraw

pygame module for drawing shapes

EXPERIMENTAL!: meaning this api may change, or dissapear in later pygame releases. If you use this, your code will break with the next pygame release.

Draw several shapes to a surface.

Most of the functions accept a color argument that is an `RGB` triplet. These can also accept an `RGBA` quadruplet. The color argument can also be an integer pixel value that is already mapped to the Surface's pixel format.

For all functions the arguments are strictly positional. Only integers are accepted for coordinates and radii.

For functions like rectangle that accept a rect argument any (x, y, w, h) sequence is accepted, though `pygame.Rect` instances are prefered. Note that for a `pygame.Rect` the drawing will not include `Rect.bottomright`. The right and bottom attributes of a Rect lie one pixel outside of the Rect's boarder.

To draw an anti aliased and filled shape, first use the aa* version of the function, and then use the filled version. For example

```
col = (255, 0, 0)
surf.fill((255, 255, 255))
pygame.gfxdraw.aacircle(surf, x, y, 30, col)
pygame.gfxdraw.filled_circle(surf, x, y, 30, col)
```

Note that pygame does not automatically import pygame.gfxdraw, so you need to import pygame.gfxdraw before using it.

Threading note: each of the functions releases the GIL during the C part of the call.

The pygame.gfxdraw module differs from the draw module in the API it uses, and also the different functions available to draw. It also wraps the primitives from the library called SDL_gfx, rather than using modified versions.

New in pygame 1.9.0.

`pygame.gfxdraw.`**`pixel`**`()`

> place a pixel
> pixel(surface, x, y, color) -> None
>
> Draws a single pixel onto a surface.

`pygame.gfxdraw.`**`hline`**`()`

draw a horizontal line
hline(surface, x1, x2, y, color) -> None

Draws a straight horizontal line on a Surface from x1 to x2 for the given y coordinate.

pygame.gfxdraw.**vline**()

draw a vertical line
vline(surface, x, y1, y2, color) -> None

Draws a straight vertical line on a Surface from y1 to y2 on the given x coordinate.

pygame.gfxdraw.**rectangle**()

draw a rectangle
rectangle(surface, rect, color) -> None

Draws the rectangle edges onto the surface. The given Rect is the area of the rectangle.

Keep in mind the `Surface.fill()` method works just as well for drawing filled rectangles. In fact the `Surface.fill()` can be hardware accelerated on some platforms with both software and hardware display modes.

pygame.gfxdraw.**box**()

draw a box
box(surface, rect, color) -> None

Draws a box (a rect) onto a surface.

pygame.gfxdraw.**line**()

draw a line
line(surface, x1, y1, x2, y2, color) -> None

Draws a straight line on a Surface. There are no endcaps.

pygame.gfxdraw.**circle**()

draw a circle
circle(surface, x, y, r, color) -> None

Draws the edges of a circular shape on the Surface. The pos argument is the center of the circle, and radius is the size. The circle is not filled with color.

pygame.gfxdraw.**arc**()

draw an arc
arc(surface, x, y, r, start, end, color) -> None

Draws an arc onto a surface.

pygame.gfxdraw.**aacircle**()

draw an anti-aliased circle
aacircle(surface, x, y, r, color) -> None

Draws the edges of an anti aliased circle onto a surface.

pygame.gfxdraw.**filled_circle**()

draw a filled circle
filled_circle(surface, x, y, r, color) -> None

Draws a filled circle onto a surface. So the inside of the circle will be filled with the given color.

pygame.gfxdraw.**ellipse**()

draw an ellipse
ellipse(surface, x, y, rx, ry, color) -> None

Draws the edges of an ellipse onto a surface.

pygame.gfxdraw.**aaellipse**()

draw an anti-aliased ellipse
aaellipse(surface, x, y, rx, ry, color) -> None

Draws anti aliased edges of an ellipse onto a surface.

pygame.gfxdraw.**filled_ellipse**()

draw a filled ellipse
filled_ellipse(surface, x, y, rx, ry, color) -> None

Draws a filled ellipse onto a surface. So the inside of the elipse will be filled with the given color.

pygame.gfxdraw.**pie**()

draw a pie
pie(surface, x, y, r, start, end, color) -> None

Draws a pie onto the surface.

pygame.gfxdraw.**trigon**()

draw a triangle
trigon(surface, x1, y1, x2, y2, x3, y3, color) -> None

Draws the edges of a trigon onto a surface. A trigon is a triangle.

pygame.gfxdraw.**aatrigon**()

draw an anti-aliased triangle
aatrigon(surface, x1, y1, x2, y2, x3, y3, color) -> None

Draws the anti aliased edges of a trigon onto a surface. A trigon is a triangle.

pygame.gfxdraw.**filled_trigon**()

draw a filled trigon
filled_trigon(surface, x1, y1, x2, y2, x3, y3, color) -> None

Draws a filled trigon onto a surface. So the inside of the trigon will be filled with the given color.

pygame.gfxdraw.**polygon**()

draw a polygon
polygon(surface, points, color) -> None

Draws the edges of a polygon onto a surface.

pygame.gfxdraw.**aapolygon**()

draw an anti-aliased polygon
aapolygon(surface, points, color) -> None

Draws the anti aliased edges of a polygon onto a surface.

pygame.gfxdraw.**filled_polygon**()

draw a filled polygon
filled_polygon(surface, points, color) -> None

Draws a filled polygon onto a surface. So the inside of the polygon will be filled with the given color.

pygame.gfxdraw.**textured_polygon**()

draw a textured polygon
textured_polygon(surface, points, texture, tx, ty) -> None

Draws a textured polygon onto a surface.

A per-pixel alpha texture blit to a per-pixel alpha surface will differ from a `Surface.blit()` blit. Also, a per-pixel alpha texture cannot be used with an 8-bit per pixel destination.

pygame.gfxdraw.**bezier**()

draw a bezier curve
bezier(surface, points, steps, color) -> None

Draws a bezier onto a surface.

CHAPTER 12

pygame.image

pygame module for image transfer

The image module contains functions for loading and saving pictures, as well as transferring Surfaces to formats usable by other packages. .

Note that there is no Image class; an image is loaded as a Surface object. The Surface class allows manipulation (drawing lines, setting pixels, capturing regions, etc.).

The image module is a required dependency of Pygame, but it only optionally supports any extended file formats. By default it can only load uncompressed BMP images. When built with full image support, the pygame.image.load() function can support the following formats.

- JPG
- PNG
- GIF (non animated)
- BMP
- PCX
- TGA (uncompressed)
- TIF
- LBM (and PBM)
- PBM (and PGM, PPM)
- XPM

Saving images only supports a limited set of formats. You can save to the following formats.

- BMP
- TGA
- PNG
- JPEG

PNG, JPEG saving new in pygame 1.8.

pygame.image.**load**()

load new image from a file

load(filename) -> Surface
load(fileobj, namehint="") -> Surface

Load an image from a file source. You can pass either a filename or a Python file-like object.

Pygame will automatically determine the image type (e.g., GIF or bitmap) and create a new Surface object from the data. In some cases it will need to know the file extension (e.g., GIF images should end in ".gif"). If you pass a raw file-like object, you may also want to pass the original filename as the namehint argument.

The returned Surface will contain the same color format, colorkey and alpha transparency as the file it came from. You will often want to call Surface.convert() with no arguments, to create a copy that will draw more quickly on the screen.

For alpha transparency, like in .png images use the convert_alpha() method after loading so that the image has per pixel transparency.

Pygame may not always be built to support all image formats. At minimum it will support uncompressed BMP. If pygame.image.get_extended() returns 'True', you should be able to load most images (including png, jpg and gif).

You should use os.path.join() for compatibility.

```
eg. asurf = pygame.image.load(os.path.join('data', 'bla.png'))
```

pygame.image.**save**()

save an image to disk
save(Surface, filename) -> None

This will save your Surface as either a BMP, TGA, PNG, or JPEG image. If the filename extension is unrecognized it will default to TGA. Both TGA, and BMP file formats create uncompressed files.

PNG, JPEG saving new in pygame 1.8.

pygame.image.**get_extended**()

test if extended image formats can be loaded
get_extended() -> bool

If pygame is built with extended image formats this function will return True. It is still not possible to determine which formats will be available, but generally you will be able to load them all.

pygame.image.**tostring**()

transfer image to string buffer
tostring(Surface, format, flipped=False) -> string

Creates a string that can be transferred with the 'fromstring' method in other Python imaging packages. Some Python image packages prefer their images in bottom-to-top format (PyOpenGL for example). If you pass True for the flipped argument, the string buffer will be vertically flipped.

The format argument is a string of one of the following values. Note that only 8bit Surfaces can use the "P" format. The other formats will work for any Surface. Also note that other Python image packages support more formats than Pygame.

- P, 8bit palettized Surfaces
- RGB, 24bit image
- RGBX, 32bit image with unused space

- RGBA, 32bit image with an alpha channel
- ARGB, 32bit image with alpha channel first
- RGBA_PREMULT, 32bit image with colors scaled by alpha channel
- ARGB_PREMULT, 32bit image with colors scaled by alpha channel, alpha channel first

pygame.image.**fromstring**()

create new Surface from a string buffer
fromstring(string, size, format, flipped=False) -> Surface

This function takes arguments similar to pygame.image.tostring(). The size argument is a pair of numbers representing the width and height. Once the new Surface is created you can destroy the string buffer.

The size and format image must compute the exact same size as the passed string buffer. Otherwise an exception will be raised.

See the pygame.image.frombuffer() method for a potentially faster way to transfer images into Pygame.

pygame.image.**frombuffer**()

create a new Surface that shares data inside a string buffer
frombuffer(string, size, format) -> Surface

Create a new Surface that shares pixel data directly from the string buffer. This method takes the same arguments as pygame.image.fromstring(), but is unable to vertically flip the source data.

This will run much faster than pygame.image.fromstring(), since no pixel data must be allocated and copied.

CHAPTER 13

pygame.joystick

Pygame module for interacting with joysticks, gamepads, and trackballs.

The joystick module manages the joystick devices on a computer. Joystick devices include trackballs and video-game-style gamepads, and the module allows the use of multiple buttons and "hats". Computers may manage multiple joysticks at a time.

Each instance of the Joystick class represents one gaming device plugged into the computer. If a gaming pad has multiple joysticks on it, than the joystick object can actually represent multiple joysticks on that single game device.

For a quick way to initialise the joystick module and get a list of Joystick instances use the following code:

```
pygame.joystick.init()
joysticks = [pygame.joystick.Joystick(x) for x in range(pygame.joystick.get_count())]
```

The following event types will be generated by the joysticks

```
JOYAXISMOTION JOYBALLMOTION JOYBUTTONDOWN JOYBUTTONUP JOYHATMOTION
```

The event queue needs to be pumped frequently for some of the methods to work. So call one of pygame.event.get, pygame.event.wait, or pygame.event.pump regularly.

pygame.joystick.**init**()

Initialize the joystick module.
init() -> None

This function is called automatically by pygame.init().

It initializes the joystick module. This will scan the system for all joystick devices. The module must be initialized before any other functions will work.

It is safe to call this function more than once.

pygame.joystick.**quit**()

Uninitialize the joystick module.
quit() -> None

Uninitialize the joystick module. After you call this any existing joystick objects will no longer work.

It is safe to call this function more than once.

pygame.joystick.**get_init**()

Returns True if the joystick module is initialized.
get_init() -> bool

Test if the pygame.joystick.init() function has been called.

pygame.joystick.**get_count**()

Returns the number of joysticks.
get_count() -> count

Return the number of joystick devices on the system. The count will be 0 if there are no joysticks on the system.

When you create Joystick objects using Joystick(id), you pass an integer that must be lower than this count.

class pygame.joystick.**Joystick**

Create a new Joystick object.
Joystick(id) -> Joystick

Create a new joystick to access a physical device. The id argument must be a value from 0 to pygame.joystick.get_count()-1.

To access most of the Joystick methods, you'll need to init() the Joystick. This is separate from making sure the joystick module is initialized. When multiple Joysticks objects are created for the same physical joystick device (i.e., they have the same ID number), the state and values for those Joystick objects will be shared.

The Joystick object allows you to get information about the types of controls on a joystick device. Once the device is initialized the Pygame event queue will start receiving events about its input.

You can call the Joystick.get_name() and Joystick.get_id() functions without initializing the Joystick object.

init()

initialize the Joystick
init() -> None

The Joystick must be initialized to get most of the information about the controls. While the Joystick is initialized the Pygame event queue will receive events from the Joystick input.

It is safe to call this more than once.

quit()

uninitialize the Joystick
quit() -> None

This will unitialize a Joystick. After this the Pygame event queue will no longer receive events from the device.

It is safe to call this more than once.

get_init()

check if the Joystick is initialized

get_init() -> bool

Returns True if the `init()` method has already been called on this Joystick object.

get_id()

get the Joystick ID
get_id() -> int

Returns the integer `ID` that represents this device. This is the same value that was passed to the `Joystick()` constructor. This method can safely be called while the Joystick is not initialized.

get_name()

get the Joystick system name
get_name() -> string

Returns the system name for this joystick device. It is unknown what name the system will give to the Joystick, but it should be a unique name that identifies the device. This method can safely be called while the Joystick is not initialized.

get_numaxes()

get the number of axes on a Joystick
get_numaxes() -> int

Returns the number of input axes are on a Joystick. There will usually be two for the position. Controls like rudders and throttles are treated as additional axes.

The `pygame.JOYAXISMOTION` events will be in the range from -1.0 to 1.0. A value of 0.0 means the axis is centered. Gamepad devices will usually be -1, 0, or 1 with no values in between. Older analog joystick axes will not always use the full -1 to 1 range, and the centered value will be some area around 0. Analog joysticks usually have a bit of noise in their axis, which will generate a lot of rapid small motion events.

get_axis()

get the current position of an axis
get_axis(axis_number) -> float

Returns the current position of a joystick axis. The value will range from -1 to 1 with a value of 0 being centered. You may want to take into account some tolerance to handle jitter, and joystick drift may keep the joystick from centering at 0 or using the full range of position values.

The axis number must be an integer from zero to get_numaxes()-1.

get_numballs()

get the number of trackballs on a Joystick
get_numballs() -> int

Returns the number of trackball devices on a Joystick. These devices work similar to a mouse but they have no absolute position; they only have relative amounts of movement.

The `pygame.JOYBALLMOTION` event will be sent when the trackball is rolled. It will report the amount of movement on the trackball.

get_ball()

get the relative position of a trackball
get_ball(ball_number) -> x, y

Returns the relative movement of a joystick button. The value is a x, y pair holding the relative movement since the last call to get_ball.

The ball number must be an integer from zero to get_numballs()-1.

get_numbuttons()

get the number of buttons on a Joystick
get_numbuttons() -> int

Returns the number of pushable buttons on the joystick. These buttons have a boolean (on or off) state.

Buttons generate a `pygame.JOYBUTTONDOWN` and `pygame.JOYBUTTONUP` event when they are pressed and released.

get_button()

get the current button state
get_button(button) -> bool

Returns the current state of a joystick button.

get_numhats()

get the number of hat controls on a Joystick
get_numhats() -> int

Returns the number of joystick hats on a Joystick. Hat devices are like miniature digital joysticks on a joystick. Each hat has two axes of input.

The `pygame.JOYHATMOTION` event is generated when the hat changes position. The position attribute for the event contains a pair of values that are either -1, 0, or 1. A position of (0, 0) means the hat is centered.

get_hat()

get the position of a joystick hat
get_hat(hat_number) -> x, y

Returns the current position of a position hat. The position is given as two values representing the `X` and `Y` position for the hat. (0, 0) means centered. A value of -1 means left/down and a value of 1 means right/up: so (-1, 0) means left; (1, 0) means right; (0, 1) means up; (1, 1) means upper-right; etc.

This value is digital, `i.e.`, each coordinate can be -1, 0 or 1 but never in-between.

The hat number must be between 0 and get_numhats()-1.

```
import pygame

# Define some colors
BLACK    = (   0,   0,   0)
WHITE    = ( 255, 255, 255)
```

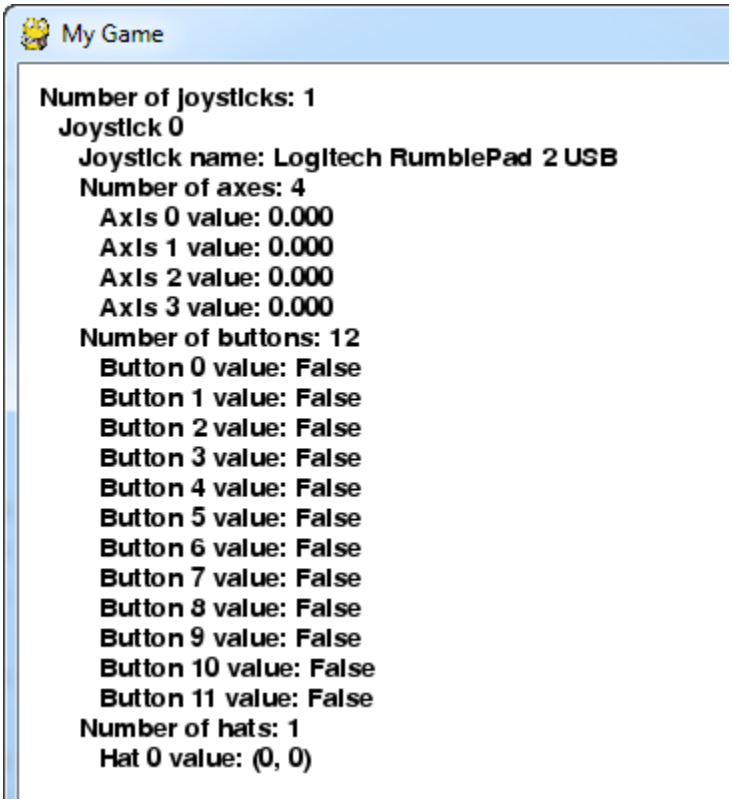

Figure 13.1: Example code for joystick module.

```
# This is a simple class that will help us print to the screen
# It has nothing to do with the joysticks, just outputing the
# information.
class TextPrint:
    def __init__(self):
        self.reset()
        self.font = pygame.font.Font(None, 20)

    def print(self, screen, textString):
        textBitmap = self.font.render(textString, True, BLACK)
        screen.blit(textBitmap, [self.x, self.y])
        self.y += self.line_height

    def reset(self):
        self.x = 10
        self.y = 10
        self.line_height = 15

    def indent(self):
        self.x += 10

    def unindent(self):
        self.x -= 10


pygame.init()
```

```
# Set the width and height of the screen [width,height]
size = [500, 700]
screen = pygame.display.set_mode(size)

pygame.display.set_caption("My Game")

#Loop until the user clicks the close button.
done = False

# Used to manage how fast the screen updates
clock = pygame.time.Clock()

# Initialize the joysticks
pygame.joystick.init()

# Get ready to print
textPrint = TextPrint()

# -------- Main Program Loop -----------
while done==False:
    # EVENT PROCESSING STEP
    for event in pygame.event.get(): # User did something
        if event.type == pygame.QUIT: # If user clicked close
            done=True # Flag that we are done so we exit this loop

        # Possible joystick actions: JOYAXISMOTION JOYBALLMOTION JOYBUTTONDOWN JOYBUTTONUP JOYHATMOTI
        if event.type == pygame.JOYBUTTONDOWN:
            print("Joystick button pressed.")
        if event.type == pygame.JOYBUTTONUP:
            print("Joystick button released.")


    # DRAWING STEP
    # First, clear the screen to white. Don't put other drawing commands
    # above this, or they will be erased with this command.
    screen.fill(WHITE)
    textPrint.reset()

    # Get count of joysticks
    joystick_count = pygame.joystick.get_count()

    textPrint.print(screen, "Number of joysticks: {}".format(joystick_count) )
    textPrint.indent()

    # For each joystick:
    for i in range(joystick_count):
        joystick = pygame.joystick.Joystick(i)
        joystick.init()

        textPrint.print(screen, "Joystick {}".format(i) )
        textPrint.indent()

        # Get the name from the OS for the controller/joystick
        name = joystick.get_name()
        textPrint.print(screen, "Joystick name: {}".format(name) )

        # Usually axis run in pairs, up/down for one, and left/right for
        # the other.
```

```
        axes = joystick.get_numaxes()
        textPrint.print(screen, "Number of axes: {}".format(axes) )
        textPrint.indent()

        for i in range( axes ):
            axis = joystick.get_axis( i )
            textPrint.print(screen, "Axis {} value: {:>6.3f}".format(i, axis) )
        textPrint.unindent()

        buttons = joystick.get_numbuttons()
        textPrint.print(screen, "Number of buttons: {}".format(buttons) )
        textPrint.indent()

        for i in range( buttons ):
            button = joystick.get_button( i )
            textPrint.print(screen, "Button {:>2} value: {}".format(i,button) )
        textPrint.unindent()

        # Hat switch. All or nothing for direction, not like joysticks.
        # Value comes back in an array.
        hats = joystick.get_numhats()
        textPrint.print(screen, "Number of hats: {}".format(hats) )
        textPrint.indent()

        for i in range( hats ):
            hat = joystick.get_hat( i )
            textPrint.print(screen, "Hat {} value: {}".format(i, str(hat)) )
        textPrint.unindent()

        textPrint.unindent()


    # ALL CODE TO DRAW SHOULD GO ABOVE THIS COMMENT

    # Go ahead and update the screen with what we've drawn.
    pygame.display.flip()

    # Limit to 20 frames per second
    clock.tick(20)

# Close the window and quit.
# If you forget this line, the program will 'hang'
# on exit if running from IDLE.
pygame.quit ()
```

CHAPTER 14

pygame.key

pygame module to work with the keyboard

This module contains functions for dealing with the keyboard.

The event queue gets pygame.KEYDOWN and pygame.KEYUP events when the keyboard buttons are pressed and released. Both events have a key attribute that is a integer id representing every key on the keyboard.

The pygame.KEYDOWN event has an additional attributes unicode, and scancode. unicode represents a single character string that is the fully translated character entered. This takes into account the shift and composition keys. scancode represents the platform specific key code. This could be different from keyboard to keyboard, but is useful for key selection of weird keys like the multimedia keys.

There are many keyboard constants, they are used to represent keys on the keyboard. The following is a list of all keyboard constants

```
KeyASCII      ASCII   Common Name
K_BACKSPACE   \b      backspace
K_TAB         \t      tab
K_CLEAR               clear
K_RETURN      \r      return
K_PAUSE               pause
K_ESCAPE      ^[      escape
K_SPACE               space
K_EXCLAIM     !       exclaim
K_QUOTEDBL    "       quotedbl
K_HASH        #       hash
K_DOLLAR      $       dollar
K_AMPERSAND   &       ampersand
K_QUOTE               quote
K_LEFTPAREN   (       left parenthesis
K_RIGHTPAREN  )       right parenthesis
K_ASTERISK    *       asterisk
K_PLUS        +       plus sign
K_COMMA       ,       comma
K_MINUS       -       minus sign
K_PERIOD      .       period
K_SLASH       /       forward slash
K_0           0       0
K_1           1       1
K_2           2       2
K_3           3       3
K_4           4       4
```

```
K_5            5       5
K_6            6       6
K_7            7       7
K_8            8       8
K_9            9       9
K_COLON        :       colon
K_SEMICOLON    ;       semicolon
K_LESS         <       less-than sign
K_EQUALS       =       equals sign
K_GREATER      >       greater-than sign
K_QUESTION     ?       question mark
K_AT           @       at
K_LEFTBRACKET  [       left bracket
K_BACKSLASH    \       backslash
K_RIGHTBRACKET ]       right bracket
K_CARET        ^       caret
K_UNDERSCORE   _       underscore
K_BACKQUOTE    `       grave
K_a            a       a
K_b            b       b
K_c            c       c
K_d            d       d
K_e            e       e
K_f            f       f
K_g            g       g
K_h            h       h
K_i            i       i
K_j            j       j
K_k            k       k
K_l            l       l
K_m            m       m
K_n            n       n
K_o            o       o
K_p            p       p
K_q            q       q
K_r            r       r
K_s            s       s
K_t            t       t
K_u            u       u
K_v            v       v
K_w            w       w
K_x            x       x
K_y            y       y
K_z            z       z
K_DELETE               delete
K_KP0                  keypad 0
K_KP1                  keypad 1
K_KP2                  keypad 2
K_KP3                  keypad 3
K_KP4                  keypad 4
K_KP5                  keypad 5
K_KP6                  keypad 6
K_KP7                  keypad 7
K_KP8                  keypad 8
K_KP9                  keypad 9
K_KP_PERIOD    .       keypad period
K_KP_DIVIDE    /       keypad divide
K_KP_MULTIPLY  *       keypad multiply
```

```
K_KP_MINUS     -       keypad minus
K_KP_PLUS      +       keypad plus
K_KP_ENTER     \r      keypad enter
K_KP_EQUALS    =       keypad equals
K_UP                   up arrow
K_DOWN                 down arrow
K_RIGHT                right arrow
K_LEFT                 left arrow
K_INSERT               insert
K_HOME                 home
K_END                  end
K_PAGEUP               page up
K_PAGEDOWN             page down
K_F1                   F1
K_F2                   F2
K_F3                   F3
K_F4                   F4
K_F5                   F5
K_F6                   F6
K_F7                   F7
K_F8                   F8
K_F9                   F9
K_F10                  F10
K_F11                  F11
K_F12                  F12
K_F13                  F13
K_F14                  F14
K_F15                  F15
K_NUMLOCK              numlock
K_CAPSLOCK             capslock
K_SCROLLOCK            scrollock
K_RSHIFT               right shift
K_LSHIFT               left shift
K_RCTRL                right ctrl
K_LCTRL                left ctrl
K_RALT                 right alt
K_LALT                 left alt
K_RMETA                right meta
K_LMETA                left meta
K_LSUPER               left windows key
K_RSUPER               right windows key
K_MODE                 mode shift
K_HELP                 help
K_PRINT                print screen
K_SYSREQ               sysrq
K_BREAK                break
K_MENU                 menu
K_POWER                power
K_EURO                 euro
```

The keyboard also has a list of modifier states that can be assembled bit bitwise ORing them together.

```
KMOD_NONE, KMOD_LSHIFT, KMOD_RSHIFT, KMOD_SHIFT, KMOD_CAPS,
KMOD_LCTRL, KMOD_RCTRL, KMOD_CTRL, KMOD_LALT, KMOD_RALT,
KMOD_ALT, KMOD_LMETA, KMOD_RMETA, KMOD_META, KMOD_NUM, KMOD_MODE
```

pygame.key.**get_focused**()

true if the display is receiving keyboard input from the system
get_focused() -> bool

This is true when the display window has keyboard focus from the system. If the display needs to ensure it does not lose keyboard focus, it can use `pygame.event.set_grab()` to grab all input.

`pygame.key.`**`get_pressed`**`()`

get the state of all keyboard buttons
get_pressed() -> bools

Returns a sequence of boolean values representing the state of every key on the keyboard. Use the key constant values to index the array. A True value means the that button is pressed.

Getting the list of pushed buttons with this function is not the proper way to handle text entry from the user. You have no way to know the order of keys pressed, and rapidly pushed keys can be completely unnoticed between two calls to `pygame.key.get_pressed()`. There is also no way to translate these pushed keys into a fully translated character value. See the `pygame.KEYDOWN` events on the event queue for this functionality.

`pygame.key.`**`get_mods`**`()`

determine which modifier keys are being held
get_mods() -> int

Returns a single integer representing a bitmask of all the modifier keys being held. Using bitwise operators you can test if specific shift keys are pressed, the state of the capslock button, and more.

`pygame.key.`**`set_mods`**`()`

temporarily set which modifier keys are pressed
set_mods(int) -> None

Create a bitmask of the modifier constants you want to impose on your program.

`pygame.key.`**`set_repeat`**`()`

control how held keys are repeated
set_repeat() -> None
set_repeat(delay, interval) -> None

When the keyboard repeat is enabled, keys that are held down will generate multiple `pygame.KEYDOWN` events. The delay is the number of milliseconds before the first repeated `pygame.KEYDOWN` will be sent. After that another `pygame.KEYDOWN` will be sent every interval milliseconds. If no arguments are passed the key repeat is disabled.

When pygame is initialized the key repeat is disabled.

`pygame.key.`**`get_repeat`**`()`

see how held keys are repeated
get_repeat() -> (delay, interval)

When the keyboard repeat is enabled, keys that are held down will generate multiple `pygame.KEYDOWN` events. The delay is the number of milliseconds before the first repeated `pygame.KEYDOWN` will be sent. After that another `pygame.KEYDOWN` will be sent every interval milliseconds.

When pygame is initialized the key repeat is disabled.

New in pygame 1.8.

pygame.key.**name**()

get the name of a key identifier
name(key) -> string

Get the descriptive name of the button from a keyboard button id constant.

CHAPTER 15

pygame.locals

pygame constants

This module contains various constants used by Pygame. It's contents are automatically placed in the pygame module namespace. However, an application can use `pygame.locals` to include only the Pygame constants with a 'from `pygame.locals` import *'.

Detailed descriptions of the various constants are found throughout the Pygame documentation. `pygame.display.set_mode()` flags like `HWSURFACE` are found in the Display section. Event types are explained in the Event section. Keyboard `K_` constants relating to the key attribute of a `KEYDOWN` or `KEYUP` event are listed in the Key section. Also found there are the various `MOD_` key modifiers. Finally, `TIMER_RESOLUTION` is defined in Time.

CHAPTER 16

pygame.mask

pygame module for image masks.

Useful for fast pixel perfect collision detection. A Mask uses 1bit per pixel to store which parts collide.

New in pygame 1.8.

pygame.mask.**from_surface**()

Returns a Mask from the given surface.
from_surface(Surface, threshold = 127) -> Mask

Makes the transparent parts of the Surface not set, and the opaque parts set.

The alpha of each pixel is checked to see if it is greater than the given threshold.

If the Surface is color keyed, then threshold is not used.

pygame.mask.**from_threshold**()

Creates a mask by thresholding Surfaces
from_threshold(Surface, color, threshold = (0,0,0,255), othersurface = None, palette_colors = 1) -> Mask

This is a more featureful method of getting a Mask from a Surface. If supplied with only one Surface, all pixels within the threshold of the supplied color are set in the Mask. If given the optional othersurface, all pixels in Surface that are within the threshold of the corresponding pixel in othersurface are set in the Mask.

class pygame.mask.**Mask**

pygame object for representing 2d bitmasks
Mask((width, height)) -> Mask

get_size()

Returns the size of the mask.
get_size() -> width,height

get_at()

Returns nonzero if the bit at (x,y) is set.
get_at((x,y)) -> int

Coordinates start at (0,0) is top left - just like Surfaces.

set_at()

Sets the position in the mask given by x and y.
set_at((x,y),value) -> None

overlap()

Returns the point of intersection if the masks overlap with the given offset - or None if it does not overlap.
overlap(othermask, offset) -> x,y

The overlap tests uses the following offsets (which may be negative):

```
+----+----------..
|A   | yoffset
|  +-+----------..
+--|B
|xoffset
|  |
:  :
```

overlap_area()

Returns the number of overlapping 'pixels'.
overlap_area(othermask, offset) -> numpixels

You can see how many pixels overlap with the other mask given. This can be used to see in which direction things collide, or to see how much the two masks collide. An approximate collision normal can be found by calculating the gradient of the overlap area through the finite difference.

```
dx = Mask.overlap_area(othermask,(x+1,y)) - Mask.overlap_area(othermask,(x-1,y))
dy = Mask.overlap_area(othermask,(x,y+1)) - Mask.overlap_area(othermask,(x,y-1))
```

overlap_mask()

Returns a mask of the overlapping pixels
overlap_mask(othermask, offset) -> Mask

Returns a Mask the size of the original Mask containing only the overlapping pixels between Mask and othermask.

fill()

Sets all bits to 1
fill() -> None

Sets all bits in a Mask to 1.

clear()

Sets all bits to 0
clear() -> None

Sets all bits in a Mask to 0.

invert()

Flips the bits in a Mask
invert() -> None

Flips all of the bits in a Mask, so that the set pixels turn to unset pixels and the unset pixels turn to set pixels.

scale()

Resizes a mask
scale((x, y)) -> Mask

Returns a new Mask of the Mask scaled to the requested size.

draw()

Draws a mask onto another
draw(othermask, offset) -> None

Performs a bitwise OR, drawing othermask onto Mask.

erase()

Erases a mask from another
erase(othermask, offset) -> None

Erases all pixels set in othermask from Mask.

count()

Returns the number of set pixels
count() -> pixels

Returns the number of set pixels in the Mask.

centroid()

Returns the centroid of the pixels in a Mask
centroid() -> (x, y)

Finds the centroid, the center of pixel mass, of a Mask. Returns a coordinate tuple for the centroid of the Mask. In the event the Mask is empty, it will return (0,0).

angle()

Returns the orientation of the pixels
angle() -> theta

Finds the approximate orientation of the pixels in the image from -90 to 90 degrees. This works best if performed on one connected component of pixels. It will return 0.0 on an empty Mask.

outline()

list of points outlining an object
outline(every = 1) -> [(x,y), (x,y) ...]

Returns a list of points of the outline of the first object it comes across in a Mask. For this to be useful, there should probably only be one connected component of pixels in the Mask. The every option allows you to skip pixels in the outline. For example, setting it to 10 would return a list of every 10th pixel in the outline.

convolve()

Return the convolution of self with another mask.

convolve(othermask, outputmask = None, offset = (0,0)) -> Mask

Returns a mask with the (i-offset[0],j-offset[1]) bit set if shifting othermask so that its lower right corner pixel is at (i,j) would cause it to overlap with self.

If an outputmask is specified, the output is drawn onto outputmask and outputmask is returned. Otherwise a mask of size `self.get_size()` + `othermask.get_size()` - (1,1) is created.

connected_component()

Returns a mask of a connected region of pixels.

connected_component((x,y) = None) -> Mask

This uses the `SAUF` algorithm to find a connected component in the Mask. It checks 8 point connectivity. By default, it will return the largest connected component in the image. Optionally, a coordinate pair of a pixel can be specified, and the connected component containing it will be returned. In the event the pixel at that location is not set, the returned Mask will be empty. The Mask returned is the same size as the original Mask.

connected_components()

Returns a list of masks of connected regions of pixels.

connected_components(min = 0) -> [Masks]

Returns a list of masks of connected regions of pixels. An optional minimum number of pixels per connected region can be specified to filter out noise.

get_bounding_rects()

Returns a list of bounding rects of regions of set pixels.

get_bounding_rects() -> Rects

This gets a bounding rect of connected regions of set pixels. A bounding rect is one for which each of the connected pixels is inside the rect.

CHAPTER 17

pygame.math

pygame module for vector classes

!!!EXPERIMENTAL!!! Note: This Modul is still in development and the API might change. Please report bug and suggestions to pygame-users@seul.org

The pygame math module currently provides Vector classes in two and three dimensions, Vector2 and Vector3 respectively.

They support the following numerical operations: vec+vec, vec-vec, vec*number, number*vec, vec/number, vec//number, vec+=vec, vec-=vec, vec*=number, vec/=number, vec//=number. All these operations will be performed elementwise. In addition vec*vec will perform a scalar-product (a.k.a. dot-product). If you want to multiply every element from vector v with every element from vector w you can use the elementwise method: `v.elementwise() \* w`

New in Pygame 1.10

`pygame.math.`**`enable_swizzling`**`()`

globally enables swizzling for vectors.
enable_swizzling() -> None

Enables swizzling for all vectors until `disable_swizzling()` is called. By default swizzling is disabled.

`pygame.math.`**`disable_swizzling`**`()`

globally disables swizzling for vectors.
disable_swizzling() -> None

Disables swizzling for all vectors until `enable_swizzling()` is called. By default swizzling is disabled.

class `pygame.math.`**`Vector2`**

a 2-Dimensional Vector
Vector2() -> Vector2
Vector2(Vector2) -> Vector2
Vector2(x, y) -> Vector2
Vector2((x, y)) -> Vector2

Some general information about the Vector2 class.

dot()

calculates the dot- or scalar-product with the other vector
dot(Vector2) -> float

cross()

calculates the cross- or vector-product
cross(Vector2) -> float

calculates the third component of the cross-product.

length()

returns the euclidic length of the vector.
length() -> float

calculates the euclidic length of the vector which follows from the Pythagorean theorem: `vec.length() == math.sqrt(vec.x**2 + vec.y**2)`

length_squared()

returns the squared euclidic length of the vector.
length_squared() -> float

calculates the euclidic length of the vector which follows from the Pythagorean theorem: `vec.length_squared()` == vec.x**2 + vec.y**2 This is faster than `vec.length()` because it avoids the square root.

normalize()

returns a vector with the same direction but length 1.
normalize() -> Vector2

Returns a new vector that has length == 1 and the same direction as self.

normalize_ip()

normalizes the vector in place so that its length is 1.
normalize_ip() -> None

Normalizes the vector so that it has length == 1. The direction of the vector is not changed.

is_normalized()

tests if the vector is normalized i.e. has length == 1.
is_normalized() -> Bool

Returns True if the vector has length == 1. Otherwise it returns False.

scale_to_length()

scales the vector to a given length.
scale_to_length(float) -> None

Scales the vector so that it has the given length. The direction of the vector is not changed. You can also scale to length 0. If the vector is the zero vector (i.e. has length 0 thus no direction) an ZeroDivisionError is raised.

reflect()

returns a vector reflected of a given normal.
reflect(Vector2) -> Vector2

Returns a new vector that points in the direction as if self would bounce of a surface characterized by the given surface normal. The length of the new vector is the same as self's.

reflect_ip()

reflect the vector of a given normal in place.
reflect_ip(Vector2) -> None

Changes the direction of self as if it would have been reflected of a surface with the given surface normal.

distance_to()

calculates the euclidic distance to a given vector.
distance_to(Vector2) -> float

distance_squared_to()

calculates the squared euclidic distance to a given vector.
distance_squared_to(Vector2) -> float

lerp()

returns a linear interpolation to the given vector.
lerp(Vector2, float) -> Vector2

Returns a Vector which is a linear interpolation between self and the given Vector. The second parameter determines how far between self an other the result is going to be. It must be a value between 0 and 1 where 0 means self an 1 means other will be returned.

slerp()

returns a spherical interpolation to the given vector.
slerp(Vector2, float) -> Vector2

Calculates the spherical interpolation from self to the given Vector. The second argument - often called t - must be in the range [-1, 1]. It parametrizes where - in between the two vectors - the result should be. If a negative value is given the interpolation will not take the complement of the shortest path.

elementwise()

The next operation will be performed elementwize.
elementwise() -> VectorElementwizeProxy

Applies the following operation to each element of the vector.

rotate()

rotates a vector by a given angle in degrees.
rotate(float) -> Vector2

Returns a vector which has the same length as self but is rotated counterclockwise by the given angle in degrees.

rotate_ip()

rotates the vector by a given angle in degrees in place.
rotate_ip(float) -> None

Rotates the vector counterclockwise by the given angle in degrees. The length of the vector is not changed.

angle_to()

calculates the angle to a given vector in degrees.
angle_to(Vector2) -> float

Returns the angle between self and the given vector.

as_polar()

returns a tuple with radial distance and azimuthal angle.
as_polar() -> (r, phi)

Returns a tuple (r, phi) where r is the radial distance, and phi is the azimuthal angle.

from_polar()

Sets x and y from a polar coordinates tuple.
from_polar((r, phi)) -> None

Sets x and y from a tuple (r, phi) where r is the radial distance, and phi is the azimuthal angle.

class pygame.math.**Vector3**

a 3-Dimensional Vector
Vector3() -> Vector3
Vector3(Vector3) -> Vector3
Vector3(x, y, z) -> Vector3
Vector3((x, y, z)) -> Vector3

Some general information about the Vector3 class.

dot()

calculates the dot- or scalar-product with the other vector
dot(Vector3) -> float

cross()

calculates the cross- or vector-product
cross(Vector3) -> float

calculates the cross-product.

length()

returns the euclidic length of the vector.
length() -> float

calculates the euclidic length of the vector which follows from the Pythagorean theorem: `vec.length() == math.sqrt(vec.x**2 + vec.y**2 + vec.z**2)`

length_squared()

returns the squared euclidic length of the vector.
length_squared() -> float

calculates the euclidic length of the vector which follows from the Pythagorean theorem: `vec.length_squared()` == vec.x**2 + vec.y**2 + vec.z**2 This is faster than `vec.length()` because it avoids the square root.

normalize()

returns a vector with the same direction but length 1.
normalize() -> Vector3

Returns a new vector that has length == 1 and the same direction as self.

normalize_ip()

normalizes the vector in place so that its length is 1.
normalize_ip() -> None

Normalizes the vector so that it has length == 1. The direction of the vector is not changed.

is_normalized()

tests if the vector is normalized i.e. has length == 1.
is_normalized() -> Bool

Returns True if the vector has length == 1. Otherwise it returns False.

scale_to_length()

scales the vector to a given length.
scale_to_length(float) -> None

Scales the vector so that it has the given length. The direction of the vector is not changed. You can also scale to length 0. If the vector is the zero vector (i.e. has length 0 thus no direction) an ZeroDivisionError is raised.

reflect()

returns a vector reflected of a given normal.
reflect(Vector3) -> Vector3

Returns a new vector that points in the direction as if self would bounce of a surface characterized by the given surface normal. The length of the new vector is the same as self's.

reflect_ip()

reflect the vector of a given normal in place.
reflect_ip(Vector3) -> None

Changes the direction of self as if it would have been reflected of a surface with the given surface normal.

distance_to()

calculates the euclidic distance to a given vector.
distance_to(Vector3) -> float

distance_squared_to()

calculates the squared euclidic distance to a given vector.
distance_squared_to(Vector3) -> float

lerp()

returns a linear interpolation to the given vector.
lerp(Vector3, float) -> Vector3

Returns a Vector which is a linear interpolation between self and the given Vector. The second parameter determines how far between self an other the result is going to be. It must be a value between 0 and 1 where 0 means self an 1 means other will be returned.

slerp()

returns a spherical interpolation to the given vector.
slerp(Vector3, float) -> Vector3

Calculates the spherical interpolation from self to the given Vector. The second argument - often called t - must be in the range [-1, 1]. It parametrizes where - in between the two vectors - the result should be. If a negative value is given the interpolation will not take the complement of the shortest path.

elementwise()

The next operation will be performed elementwize.
elementwise() -> VectorElementwizeProxy

Applies the following operation to each element of the vector.

rotate()

rotates a vector by a given angle in degrees.
rotate(Vector3, float) -> Vector3

Returns a vector which has the same length as self but is rotated counterclockwise by the given angle in degrees around the given axis.

rotate_ip()

rotates the vector by a given angle in degrees in place.
rotate_ip(Vector3, float) -> None

Rotates the vector counterclockwise around the given axis by the given angle in degrees. The length of the vector is not changed.

rotate_x()

rotates a vector around the x-axis by the angle in degrees.
rotate_x(float) -> Vector3

Returns a vector which has the same length as self but is rotated counterclockwise around the x-axis by the given angle in degrees.

rotate_x_ip()

rotates the vector around the x-axis by the angle in degrees in place.
rotate_x_ip(float) -> None

Rotates the vector counterclockwise around the x-axis by the given angle in degrees. The length of the vector is not changed.

rotate_y()

rotates a vector around the y-axis by the angle in degrees.
rotate_y(float) -> Vector3

Returns a vector which has the same length as self but is rotated counterclockwise around the y-axis by the given angle in degrees.

rotate_y_ip()

rotates the vector around the y-axis by the angle in degrees in place.
rotate_y_ip(float) -> None

Rotates the vector counterclockwise around the y-axis by the given angle in degrees. The length of the vector is not changed.

rotate_z()

rotates a vector around the z-axis by the angle in degrees.
rotate_z(float) -> Vector3

Returns a vector which has the same length as self but is rotated counterclockwise around the z-axis by the given angle in degrees.

rotate_z_ip()

rotates the vector around the z-axis by the angle in degrees in place.
rotate_z_ip(float) -> None

Rotates the vector counterclockwise around the z-axis by the given angle in degrees. The length of the vector is not changed.

angle_to()

calculates the angle to a given vector in degrees.
angle_to(Vector3) -> float

Returns the angle between self and the given vector.

as_spherical()

returns a tuple with radial distance, inclination and azimuthal angle.

as_spherical() -> (r, theta, phi)

Returns a tuple (r, theta, phi) where r is the radial distance, theta is the inclination angle and phi is the azimuthal angle.

from_spherical()

Sets x, y and z from a spherical coordinates 3-tuple.

from_spherical((r, theta, phi)) -> None

Sets x, y and z from a tuple (r, theta, phi) where r is the radial distance, theta is the inclination angle and phi is the azimuthal angle.

CHAPTER 18

pygame.midi

pygame module for interacting with midi input and output.

The midi module can send output to midi devices, and get input from midi devices. It can also list midi devices on the system.

Including real midi devices, and virtual ones.

It uses the portmidi library. Is portable to which ever platforms portmidi supports (currently windows, OSX, and linux).

This uses pyportmidi for now, but may use its own bindings at some point in the future. The pyportmidi bindings are included with pygame.

New in pygame 1.9.0.

class pygame.midi.**Input**

Input is used to get midi input from midi devices.
Input(device_id) -> None
Input(device_id, buffer_size) -> None

buffer_size -the number of input events to be buffered waiting to

be read using Input.read()

close()

closes a midi stream, flushing any pending buffers.
close() -> None

PortMidi attempts to close open streams when the application exits – this is particularly difficult under Windows.

poll()

returns true if there's data, or false if not.
poll() -> Bool

raises a MidiException on error.

read()

reads num_events midi events from the buffer.
read(num_events) -> midi_event_list

Reads from the Input buffer and gives back midi events. [[[status,data1,data2,data3],timestamp],

```
[[status,data1,data2,data3],timestamp],...]
```

pygame.midi.**MidiException**()

exception that pygame.midi functions and classes can raise
MidiException(errno) -> None

class pygame.midi.**Output**

Output is used to send midi to an output device
Output(device_id) -> None
Output(device_id, latency = 0) -> None
Output(device_id, buffer_size = 4096) -> None
Output(device_id, latency, buffer_size) -> None

The buffer_size specifies the number of output events to be buffered waiting for output. (In some cases – see below – PortMidi does not buffer output at all and merely passes data to a lower-level API, in which case buffersize is ignored.)

latency is the delay in milliseconds applied to timestamps to determine when the output should actually occur. (If latency is <<0, 0 is assumed.)

If latency is zero, timestamps are ignored and all output is delivered immediately. If latency is greater than zero, output is delayed until the message timestamp plus the latency. (NOTE: time is measured relative to the time source indicated by time_proc. Timestamps are absolute, not relative delays or offsets.) In some cases, PortMidi can obtain better timing than your application by passing timestamps along to the device driver or hardware. Latency may also help you to synchronize midi data to audio data by matching midi latency to the audio buffer latency.

abort()

terminates outgoing messages immediately
abort() -> None

The caller should immediately close the output port; this call may result in transmission of a partial midi message. There is no abort for Midi input because the user can simply ignore messages in the buffer and close an input device at any time.

close()

closes a midi stream, flushing any pending buffers.
close() -> None

PortMidi attempts to close open streams when the application exits – this is particularly difficult under Windows.

note_off()

turns a midi note off. Note must be on.
note_off(note, velocity=None, channel = 0) -> None

Turn a note off in the output stream. The note must already be on for this to work correctly.

note_on()

turns a midi note on. Note must be off.
note_on(note, velocity=None, channel = 0) -> None

Turn a note on in the output stream. The note must already be off for this to work correctly.

set_instrument()

select an instrument, with a value between 0 and 127
set_instrument(instrument_id, channel = 0) -> None

write()

writes a list of midi data to the Output
write(data) -> None

writes series of MIDI information in the form of a list:

```
write([[[status <,data1><,data2><,data3>],timestamp],
       [[status <,data1><,data2><,data3>],timestamp],...])
```

<<ata> fields are optional example: choose program change 1 at time 20000 and send note 65 with velocity 100 500 ms later.

```
write([[[0xc0,0,0],20000],[[0x90,60,100],20500]])
```

notes:

```
1. timestamps will be ignored if latency = 0.
2. To get a note to play immediately, send MIDI info with
   timestamp read from function Time.
3. understanding optional data fields:
     write([[[0xc0,0,0],20000]]) is equivalent to
     write([[[0xc0],20000]])
```

Can send up to 1024 elements in your data list, otherwise an

```
IndexError exception is raised.
```

write_short()

write_short(status <, data1><, data2>)
write_short(status) -> None
write_short(status, data1 = 0, data2 = 0) -> None

output MIDI information of 3 bytes or less. data fields are optional status byte could be:

```
0xc0 = program change
0x90 = note on
etc.
data bytes are optional and assumed 0 if omitted
```

example: note 65 on with velocity 100

```
write_short(0x90,65,100)
```

write_sys_ex()

writes a timestamped system-exclusive midi message.
write_sys_ex(when, msg) -> None

msg - can be a *list* or a *string* when - a timestamp in miliseconds example:

```
(assuming o is an onput MIDI stream)
  o.write_sys_ex(0,'\xF0\x7D\x10\x11\x12\x13\xF7')
is equivalent to
  o.write_sys_ex(pygame.midi.time(),
                 [0xF0,0x7D,0x10,0x11,0x12,0x13,0xF7])
```

pygame.midi.**get_count**()

gets the number of devices.
get_count() -> num_devices

Device ids range from 0 to `get_count()` -1

pygame.midi.**get_default_input_id**()

gets default input device number
get_default_input_id() -> default_id

Return the default device `ID` or -1 if there are no devices. The result can be passed to the Input()/Ouput() class.

On the `PC`, the user can specify a default device by setting an environment variable. For example, to use device #1.

```
set PM_RECOMMENDED_INPUT_DEVICE=1
```

The user should first determine the available device `ID` by using the supplied application "testin" or "testout".

In general, the registry is a better place for this kind of info, and with `USB` devices that can come and go, using integers is not very reliable for device identification. Under Windows, if `PM_RECOMMENDED_OUTPUT_DEVICE` (or `PM_RECOMMENDED_INPUT_DEVICE`) is *NOT* found in the environment, then the default device is obtained by looking for a string in the registry under:

```
HKEY_LOCAL_MACHINE/SOFTWARE/PortMidi/Recommended_Input_Device
```

and HKEY_LOCAL_MACHINE/SOFTWARE/PortMidi/Recommended_Output_Device for a string. The number of the first device with a substring that matches the string exactly is returned. For example, if the string in the registry is "USB", and device 1 is named "In `USB` MidiSport 1x1", then that will be the default input because it contains the string "USB".

In addition to the name, `get_device_info()` returns "interf", which is the interface name. (The "interface" is the underlying software system or `API` used by PortMidi to access devices. Examples are MMSystem, DirectX (not implemented), `ALSA`, `OSS` (not implemented), etc.) At present, the only Win32 interface is "MMSystem", the only Linux interface is "ALSA", and the only Max `OS X` interface is "CoreMIDI". To specify both the interface and the device name in the registry, separate the two with a comma and a space, `e.g.`:

```
MMSystem, In USB MidiSport 1x1
```

In this case, the string before the comma must be a substring of the "interf" string, and the string after the space must be a substring of the "name" name string in order to match the device.

Note: in the current release, the default is simply the first device (the input or output device with the lowest PmDeviceID).

pygame.midi.**get_default_output_id**()

gets default output device number

get_default_output_id() -> default_id

Return the default device ID or -1 if there are no devices. The result can be passed to the Input()/Ouput() class.

On the PC, the user can specify a default device by setting an environment variable. For example, to use device #1.

```
set PM_RECOMMENDED_OUTPUT_DEVICE=1
```

The user should first determine the available device ID by using the supplied application "testin" or "testout".

In general, the registry is a better place for this kind of info, and with USB devices that can come and go, using integers is not very reliable for device identification. Under Windows, if PM_RECOMMENDED_OUTPUT_DEVICE (or PM_RECOMMENDED_INPUT_DEVICE) is *NOT* found in the environment, then the default device is obtained by looking for a string in the registry under:

```
HKEY_LOCAL_MACHINE/SOFTWARE/PortMidi/Recommended_Input_Device
```

and HKEY_LOCAL_MACHINE/SOFTWARE/PortMidi/Recommended_Output_Device for a string. The number of the first device with a substring that matches the string exactly is returned. For example, if the string in the registry is "USB", and device 1 is named "In USB MidiSport 1x1", then that will be the default input because it contains the string "USB".

In addition to the name, get_device_info() returns "interf", which is the interface name. (The "interface" is the underlying software system or API used by PortMidi to access devices. Examples are MMSystem, DirectX (not implemented), ALSA, OSS (not implemented), etc.) At present, the only Win32 interface is "MMSystem", the only Linux interface is "ALSA", and the only Max OS X interface is "CoreMIDI". To specify both the interface and the device name in the registry, separate the two with a comma and a space, e.g.:

```
MMSystem, In USB MidiSport 1x1
```

In this case, the string before the comma must be a substring of the "interf" string, and the string after the space must be a substring of the "name" name string in order to match the device.

Note: in the current release, the default is simply the first device (the input or output device with the lowest PmDeviceID).

pygame.midi.**get_device_info**()

returns information about a midi device

get_device_info(an_id) -> (interf, name, input, output, opened)

interf - a text string describing the device interface, eg 'ALSA'. name - a text string for the name of the device, eg 'Midi Through Port-0' input - 0, or 1 if the device is an input device. output - 0, or 1 if the device is an output device. opened - 0, or 1 if the device is opened.

If the id is out of range, the function returns None.

pygame.midi.**init**()

initialize the midi module

init() -> None

Call the initialisation function before using the midi module.

It is safe to call this more than once.

pygame.midi.**midis2events**()

converts midi events to pygame events
midis2events(midis, device_id) -> [Event, ...]

Takes a sequence of midi events and returns list of pygame events.

pygame.midi.**quit**()

uninitialize the midi module
quit() -> None

Called automatically atexit if you don't call it.

It is safe to call this function more than once.

pygame.midi.**time**()

returns the current time in ms of the PortMidi timer
time() -> time

The time is reset to 0, when the module is inited.

CHAPTER 19

pygame.mixer

pygame module for loading and playing sounds

This module contains classes for loading Sound objects and controlling playback. The mixer module is optional and depends on SDL_mixer. Your program should test that `pygame.mixer` is available and intialized before using it.

The mixer module has a limited number of channels for playback of sounds. Usually programs tell pygame to start playing audio and it selects an available channel automatically. The default is 8 simultaneous channels, but complex programs can get more precise control over the number of channels and their use.

All sound playback is mixed in background threads. When you begin to play a Sound object, it will return immediately while the sound continues to play. A single Sound object can also be actively played back multiple times.

The mixer also has a special streaming channel. This is for music playback and is accessed through the `pygame.mixer.music` module.

The mixer module must be initialized like other pygame modules, but it has some extra conditions. The `pygame.mixer.init()` function takes several optional arguments to control the playback rate and sample size. Pygame will default to reasonable values, but pygame cannot perform Sound resampling, so the mixer should be initialized to match the values of your audio resources.

`NOTE`: Not to get less laggy sound, use a smaller buffer size. The default is set to reduce the chance of scratchy sounds on some computers. You can change the default buffer by calling `pygame.mixer.pre_init()` before `pygame.mixer.init()` or `pygame.init()` is called. For example: `pygame.mixer.pre_init(44100,-16,2, 1024)` The default size was changed from 1024 to 3072 in pygame 1.8.

`pygame.mixer.`**`init`**`()`

initialize the mixer module

init(frequency=22050, size=-16, channels=2, buffer=4096) -> None

Initialize the mixer module for Sound loading and playback. The default arguments can be overridden to provide specific audio mixing. Keyword arguments are accepted. For backward compatibility where an argument is set zero the default value is used (possible changed by a pre_init call).

The size argument represents how many bits are used for each audio sample. If the value is negative then signed sample values will be used. Positive values mean unsigned audio samples will be used. An invalid value raises an exception.

The channels argument is used to specify whether to use mono or stereo. 1 for mono and 2 for stereo. No other values are supported (negative values are treated as 1, values greater than 2 as 2).

The buffer argument controls the number of internal samples used in the sound mixer. The default value should work for most cases. It can be lowered to reduce latency, but sound dropout may occur. It can be raised to larger values to ensure playback never skips, but it will impose latency on sound playback. The buffer size must be a power of two (if not it is rounded up to the next nearest power of 2).

Some platforms require the pygame.mixer module to be initialized after the display modules have initialized. The top level pygame.init() takes care of this automatically, but cannot pass any arguments to the mixer init. To solve this, mixer has a function pygame.mixer.pre_init() to set the proper defaults before the toplevel init is used.

It is safe to call this more than once, but after the mixer is initialized you cannot change the playback arguments without first calling pygame.mixer.quit().

pygame.mixer.**pre_init**()

preset the mixer init arguments

pre_init(frequency=22050, size=-16, channels=2, buffersize=4096) -> None

Call pre_init to change the defaults used when the real pygame.mixer.init() is called. Keyword arguments are accepted. The best way to set custom mixer playback values is to call pygame.mixer.pre_init() before calling the top level pygame.init(). For backward compatibility argument values of zero is replaced with the startup defaults.

pygame.mixer.**quit**()

uninitialize the mixer

quit() -> None

This will uninitialize pygame.mixer. All playback will stop and any loaded Sound objects may not be compatible with the mixer if it is reinitialized later.

pygame.mixer.**get_init**()

test if the mixer is initialized

get_init() -> (frequency, format, channels)

If the mixer is initialized, this returns the playback arguments it is using. If the mixer has not been initialized this returns None

pygame.mixer.**stop**()

stop playback of all sound channels

stop() -> None

This will stop all playback of all active mixer channels.

pygame.mixer.**pause**()

temporarily stop playback of all sound channels

pause() -> None

This will temporarily stop all playback on the active mixer channels. The playback can later be resumed with pygame.mixer.unpause()

pygame.mixer.**unpause**()

resume paused playback of sound channels

unpause() -> None

This will resume all active sound channels after they have been paused.

pygame.mixer.**fadeout**()

fade out the volume on all sounds before stopping
fadeout(time) -> None

This will fade out the volume on all active channels over the time argument in milliseconds. After the sound is muted the playback will stop.

pygame.mixer.**set_num_channels**()

set the total number of playback channels
set_num_channels(count) -> None

Sets the number of available channels for the mixer. The default value is 8. The value can be increased or decreased. If the value is decreased, sounds playing on the truncated channels are stopped.

pygame.mixer.**get_num_channels**()

get the total number of playback channels
get_num_channels() -> count

Returns the number of currently active playback channels.

pygame.mixer.**set_reserved**()

reserve channels from being automatically used
set_reserved(count) -> None

The mixer can reserve any number of channels that will not be automatically selected for playback by Sounds. If sounds are currently playing on the reserved channels they will not be stopped.

This allows the application to reserve a specific number of channels for important sounds that must not be dropped or have a guaranteed channel to play on.

pygame.mixer.**find_channel**()

find an unused channel
find_channel(force=False) -> Channel

This will find and return an inactive Channel object. If there are no inactive Channels this function will return None. If there are no inactive channels and the force argument is True, this will find the Channel with the longest running Sound and return it.

If the mixer has reserved channels from `pygame.mixer.set_reserved()` then those channels will not be returned here.

pygame.mixer.**get_busy**()

test if any sound is being mixed
get_busy() -> bool

Returns True if the mixer is busy mixing any channels. If the mixer is idle then this return False.

class pygame.mixer.**Sound**

Create a new Sound object from a file or buffer object
Sound(filename) -> Sound
Sound(file=filename) -> Sound
Sound(buffer) -> Sound
Sound(buffer=buffer) -> Sound
Sound(object) -> Sound
Sound(file=object) -> Sound
Sound(array=object) -> Sound

Load a new sound buffer from a filename, a python file object or a readable buffer object. Limited resampling will be performed to help the sample match the initialize arguments for the mixer. A Unicode string can only be a file pathname. A Python 2.x string or a Python 3.x bytes object can be either a pathname or a buffer object. Use the 'file' or 'buffer' keywords to avoid ambiguity; otherwise Sound may guess wrong. If the array keyword is used, the object is expected to export a version 3, C level array interface or, for Python 2.6 or later, a new buffer interface (The object is checked for a buffer interface first.)

The Sound object represents actual sound sample data. Methods that change the state of the Sound object will the all instances of the Sound playback. A Sound object also exports an array interface, and, for Python 2.6 or later, a new buffer interface.

The Sound can be loaded from an OGG audio file or from an uncompressed WAV.

Note: The buffer will be copied internally, no data will be shared between it and the Sound object.

For now buffer and array support is consistent with sndarray.make_sound for Numeric arrays, in that sample sign and byte order are ignored. This will change, either by correctly handling sign and byte order, or by raising an exception when different. Also, source samples are truncated to fit the audio sample size. This will not change.

pygame.mixer.Sound(buffer) is new in pygame 1.8 pygame.mixer.Sound keyword arguments and array interface support new in pygame 1.9.2

play()

begin sound playback
play(loops=0, maxtime=0, fade_ms=0) -> Channel

Begin playback of the Sound (i.e., on the computer's speakers) on an available Channel. This will forcibly select a Channel, so playback may cut off a currently playing sound if necessary.

The loops argument controls how many times the sample will be repeated after being played the first time. A value of 5 means that the sound will be played once, then repeated five times, and so is played a total of six times. The default value (zero) means the Sound is not repeated, and so is only played once. If loops is set to -1 the Sound will loop indefinitely (though you can still call stop() to stop it).

The maxtime argument can be used to stop playback after a given number of milliseconds.

The fade_ms argument will make the sound start playing at 0 volume and fade up to full volume over the time given. The sample may end before the fade-in is complete.

This returns the Channel object for the channel that was selected.

stop()

stop sound playback
stop() -> None

This will stop the playback of this Sound on any active Channels.

fadeout()

stop sound playback after fading out
fadeout(time) -> None

This will stop playback of the sound after fading it out over the time argument in milliseconds. The Sound will fade and stop on all actively playing channels.

set_volume()

set the playback volume for this Sound
set_volume(value) -> None

This will set the playback volume (loudness) for this Sound. This will immediately affect the Sound if it is playing. It will also affect any future playback of this Sound. The argument is a value from 0.0 to 1.0.

get_volume()

get the playback volume
get_volume() -> value

Return a value from 0.0 to 1.0 representing the volume for this Sound.

get_num_channels()

count how many times this Sound is playing
get_num_channels() -> count

Return the number of active channels this sound is playing on.

get_length()

get the length of the Sound
get_length() -> seconds

Return the length of this Sound in seconds.

get_buffer()

acquires a buffer object for the samples of the Sound.
get_buffer() -> BufferProxy

Return a buffer object for the Sound samples. The buffer can be used for direct access and manipulation.

New in pygame 1.8.

class pygame.mixer.**Channel**

Create a Channel object for controlling playback
Channel(id) -> Channel

Return a Channel object for one of the current channels. The id must be a value from 0 to the value of pygame.mixer.get_num_channels().

The Channel object can be used to get fine control over the playback of Sounds. A channel can only playback a single Sound at time. Using channels is entirely optional since pygame can manage them by default.

play()

play a Sound on a specific Channel
play(Sound, loops=0, maxtime=0, fade_ms=0) -> None

This will begin playback of a Sound on a specific Channel. If the Channel is currently playing any other Sound it will be stopped.

The loops argument has the same meaning as in `Sound.play()`: it is the number of times to repeat the sound after the first time. If it is 3, the sound will be played 4 times (the first time, then three more). If loops is -1 then the playback will repeat indefinitely.

As in `Sound.play()`, the maxtime argument can be used to stop playback of the Sound after a given number of milliseconds.

As in `Sound.play()`, the fade_ms argument can be used fade in the sound.

stop()

stop playback on a Channel
stop() -> None

Stop sound playback on a channel. After playback is stopped the channel becomes available for new Sounds to play on it.

pause()

temporarily stop playback of a channel
pause() -> None

Temporarily stop the playback of sound on a channel. It can be resumed at a later time with `Channel.unpause()`

unpause()

resume pause playback of a channel
unpause() -> None

Resume the playback on a paused channel.

fadeout()

stop playback after fading channel out
fadeout(time) -> None

Stop playback of a channel after fading out the sound over the given time argument in milliseconds.

set_volume()

set the volume of a playing channel
set_volume(value) -> None
set_volume(left, right) -> None

Set the volume (loudness) of a playing sound. When a channel starts to play its volume value is reset. This only affects the current sound. The value argument is between 0.0 and 1.0.

If one argument is passed, it will be the volume of both speakers. If two arguments are passed and the mixer is in stereo mode, the first argument will be the volume of the left speaker and the second will be the volume of the right speaker. (If the second argument is None, the first argument will be the volume of both speakers.)

If the channel is playing a Sound on which `set_volume()` has also been called, both calls are taken into account. For example:

```
sound = pygame.mixer.Sound("s.wav")
channel = s.play()       # Sound plays at full volume by default
sound.set_volume(0.9)    # Now plays at 90% of full volume.
sound.set_volume(0.6)    # Now plays at 60% (previous value replaced).
channel.set_volume(0.5) # Now plays at 30% (0.6 * 0.5).
```

get_volume()

get the volume of the playing channel
get_volume() -> value

Return the volume of the channel for the current playing sound. This does not take into account stereo separation used by `Channel.set_volume()`. The Sound object also has its own volume which is mixed with the channel.

get_busy()

check if the channel is active
get_busy() -> bool

Returns true if the channel is activily mixing sound. If the channel is idle this returns False.

get_sound()

get the currently playing Sound
get_sound() -> Sound

Return the actual Sound object currently playing on this channel. If the channel is idle None is returned.

queue()

queue a Sound object to follow the current
queue(Sound) -> None

When a Sound is queued on a Channel, it will begin playing immediately after the current Sound is finished. Each channel can only have a single Sound queued at a time. The queued Sound will only play if the current playback finished automatically. It is cleared on any other call to `Channel.stop()` or `Channel.play()`.

If there is no sound actively playing on the Channel then the Sound will begin playing immediately.

get_queue()

return any Sound that is queued
get_queue() -> Sound

If a Sound is already queued on this channel it will be returned. Once the queued sound begins playback it will no longer be on the queue.

set_endevent()

have the channel send an event when playback stops

set_endevent() -> None

set_endevent(type) -> None

When an endevent is set for a channel, it will send an event to the pygame queue every time a sound finishes playing on that channel (not just the first time). Use `pygame.event.get()` to retrieve the endevent once it's sent.

Note that if you called `Sound.play(n)` or `Channel.play(sound,n)`, the end event is sent only once: after the sound has been played "n+1" times (see the documentation of Sound.play).

If `Channel.stop()` or `Channel.play()` is called while the sound was still playing, the event will be posted immediately.

The type argument will be the event id sent to the queue. This can be any valid event type, but a good choice would be a value between `pygame.locals.USEREVENT` and `pygame.locals.NUMEVENTS`. If no type argument is given then the Channel will stop sending endevents.

get_endevent()

get the event a channel sends when playback stops

get_endevent() -> type

Returns the event type to be sent every time the Channel finishes playback of a Sound. If there is no endevent the function returns `pygame.NOEVENT`.

CHAPTER 20

pygame.mouse

pygame module to work with the mouse

The mouse functions can be used to get the current state of the mouse device. These functions can also alter the system cursor for the mouse.

When the display mode is set, the event queue will start receiving mouse events. The mouse buttons generate pygame.MOUSEBUTTONDOWN and pygame.MOUSEBUTTONUP events when they are pressed and released. These events contain a button attribute representing which button was pressed. The mouse wheel will generate pygame.MOUSEBUTTONDOWN and pygame.MOUSEBUTTONUP events when rolled. The button will be set to 4 when the wheel is rolled up, and to button 5 when the wheel is rolled down. Anytime the mouse is moved it generates a pygame.MOUSEMOTION event. The mouse movement is broken into small and accurate motion events. As the mouse is moving many motion events will be placed on the queue. Mouse motion events that are not properly cleaned from the event queue are the primary reason the event queue fills up.

If the mouse cursor is hidden, and input is grabbed to the current display the mouse will enter a virtual input mode, where the relative movements of the mouse will never be stopped by the borders of the screen. See the functions pygame.mouse.set_visible() and pygame.event.set_grab() to get this configured.

pygame.mouse.**get_pressed**()

get the state of the mouse buttons
get_pressed() -> (button1, button2, button3)

Returns a sequence of booleans representing the state of all the mouse buttons. A true value means the mouse is currently being pressed at the time of the call.

Note, to get all of the mouse events it is better to use either

```
pygame.event.wait() or pygame.event.get() and check all of those events
```

to see if they are MOUSEBUTTONDOWN, MOUSEBUTTONUP, or MOUSEMOTION.

Note, that on X11 some XServers use middle button emulation. When you click both buttons 1 and 3 at the same time a 2 button event can be emitted.

Note, remember to call pygame.event.get() before this function. Otherwise it will not work.

pygame.mouse.**get_pos**()

get the mouse cursor position
get_pos() -> (x, y)

Returns the X and Y position of the mouse cursor. The position is relative the the top-left corner of the display. The cursor position can be located outside of the display window, but is always constrained to the screen.

pygame.mouse.**get_rel**()

get the amount of mouse movement

get_rel() -> (x, y)

Returns the amount of movement in X and Y since the previous call to this function. The relative movement of the mouse cursor is constrained to the edges of the screen, but see the virtual input mouse mode for a way around this. Virtual input mode is described at the top of the page.

pygame.mouse.**set_pos**()

set the mouse cursor position

set_pos([x, y]) -> None

Set the current mouse position to arguments given. If the mouse cursor is visible it will jump to the new coordinates. Moving the mouse will generate a new pygame.MOUSEMOTION event.

pygame.mouse.**set_visible**()

hide or show the mouse cursor

set_visible(bool) -> bool

If the bool argument is true, the mouse cursor will be visible. This will return the previous visible state of the cursor.

pygame.mouse.**get_focused**()

check if the display is receiving mouse input

get_focused() -> bool

Returns true when pygame is receiving mouse input events (or, in windowing terminology, is "active" or has the "focus").

This method is most useful when working in a window. By contrast, in full-screen mode, this method always returns true.

Note: under MS Windows, the window that has the mouse focus also has the keyboard focus. But under X-Windows, one window can receive mouse events and another receive keyboard events. pygame.mouse.get_focused() indicates whether the pygame window receives mouse events.

pygame.mouse.**set_cursor**()

set the image for the system mouse cursor

set_cursor(size, hotspot, xormasks, andmasks) -> None

When the mouse cursor is visible, it will be displayed as a black and white bitmap using the given bitmask arrays. The size is a sequence containing the cursor width and height. Hotspot is a sequence containing the cursor hotspot position. xormasks is a sequence of bytes containing the cursor xor data masks. Lastly is andmasks, a sequence of bytes containting the cursor bitmask data.

Width must be a multiple of 8, and the mask arrays must be the correct size for the given width and height. Otherwise an exception is raised.

See the pygame.cursor module for help creating default and custom masks for the system cursor.

pygame.mouse.**get_cursor**()

get the image for the system mouse cursor
get_cursor() -> (size, hotspot, xormasks, andmasks)

Get the information about the mouse system cursor. The return value is the same data as the arguments passed into pygame.mouse.set_cursor().

CHAPTER 21

pygame.movie

pygame module for playback of mpeg video

NOTE: On NT derived Windows versions (e.g. XT) the default SDL directx video driver is problematic. For pygame.movie, use the windib driver instead. To enable windib set the SDL_VIDEODRIVER environment variable to 'windib' before importing pygame (see the pygame.examples.movieplayer.main() example).

Pygame can playback video and audio from basic encoded MPEG-1 video files. Movie playback happens in background threads, which makes playback easy to manage.

The audio for Movies must have full control over the sound system. This means the pygame.mixer module must be uninitialized if the movie's sound is to be played. The common solution is to call pygame.mixer.quit() before the movie begins. The mixer can be reinitialized after the movie is finished.

The video overlay planes are drawn on top of everything in the display window. To draw the movie as normal graphics into the display window, create an offscreen Surface and set that as the movie target. Then once per frame blit that surface to the screen.

Videos can be converted to the .mpg file format (MPEG-1 video, MPEG-1 Audio Layer III (MP3) sound) using ffmpeg video conversion program (http://ffmpeg.org/):

```
ffmpeg -i <infile> -vcodec mpeg1video -acodec libmp3lame -intra <outfile.mpg>
```

class pygame.movie.**Movie**

load an mpeg movie file
Movie(filename) -> Movie
Movie(object) -> Movie

Load a new MPEG movie stream from a file or a python file object. The Movie object operates similar to the Sound objects from pygame.mixer.

Movie objects have a target display Surface. The movie is rendered into this Surface in a background thread. If the target Surface is the display Surface, the movie will try to use the hardware accelerated video overlay. The default target is the display Surface.

play()

start playback of a movie
play(loops=0) -> None

Starts playback of the movie. Sound and video will begin playing if they are not disabled. The optional loops argument controls how many times the movie will be repeated. A loop value of -1 means the movie will repeat forever.

stop()

stop movie playback
stop() -> None

Stops the playback of a movie. The video and audio playback will be stopped at their current position.

pause()

temporarily stop and resume playback
pause() -> None

This will temporarily stop or restart movie playback.

skip()

advance the movie playback position
skip(seconds) -> None

Advance the movie playback time in seconds. This can be called before the movie is played to set the starting playback time. This can only skip the movie forward, not backwards. The argument is a floating point number.

rewind()

restart the movie playback
rewind() -> None

Sets the movie playback position to the start of the movie. The movie will automatically begin playing even if it stopped.

The can raise a ValueError if the movie cannot be rewound. If the rewind fails the movie object is considered invalid.

render_frame()

set the current video frame
render_frame(frame_number) -> frame_number

This takes an integer frame number to render. It attempts to render the given frame from the movie to the target Surface. It returns the real frame number that got rendered.

get_frame()

get the current video frame
get_frame() -> frame_number

Returns the integer frame number of the current video frame.

get_time()

get the current vide playback time

get_time() -> seconds

Return the current playback time as a floating point value in seconds. This method currently seems broken and always returns 0.0.

get_busy()

check if the movie is currently playing
get_busy() -> bool

Returns true if the movie is currently being played.

get_length()

the total length of the movie in seconds
get_length() -> seconds

Returns the length of the movie in seconds as a floating point value.

get_size()

get the resolution of the video
get_size() -> (width, height)

Gets the resolution of the movie video. The movie will be stretched to the size of any Surface, but this will report the natural video size.

has_video()

check if the movie file contains video
has_video() -> bool

True when the opened movie file contains a video stream.

has_audio()

check if the movie file contains audio
has_audio() -> bool

True when the opened movie file contains an audio stream.

set_volume()

set the audio playback volume
set_volume(value) -> None

Set the playback volume for this movie. The argument is a value between 0.0 and 1.0. If the volume is set to 0 the movie audio will not be decoded.

set_display()

set the video target Surface
set_display(Surface, rect=None) -> None

Set the output target Surface for the movie video. You may also pass a rectangle argument for the position, which will move and stretch the video into the given area.

If None is passed as the target Surface, the video decoding will be disabled.

CHAPTER 22

pygame.mixer.music

pygame module for controlling streamed audio

The music module is closely tied to pygame.mixer. Use the music module to control the playback of music in the sound mixer.

The difference between the music playback and regular Sound playback is that the music is streamed, and never actually loaded all at once. The mixer system only supports a single music stream at once.

Be aware that MP3 support is limited. On some systems an unsupported format can crash the program, e.g. Debian Linux. Consider using OGG instead.

pygame.mixer.music.**load**()

Load a music file for playback
load(filename) -> None
load(object) -> None

This will load a music filename/file object and prepare it for playback. If a music stream is already playing it will be stopped. This does not start the music playing.

pygame.mixer.music.**play**()

Start the playback of the music stream
play(loops=0, start=0.0) -> None

This will play the loaded music stream. If the music is already playing it will be restarted.

The loops argument controls the number of repeats a music will play. play(5) will cause the music to played once, then repeated five times, for a total of six. If the loops is -1 then the music will repeat indefinitely.

The starting position argument controls where in the music the song starts playing. The starting position is dependent on the format of music playing. MP3 and OGG use the position as time (in seconds). MOD music it is the pattern order number. Passing a startpos will raise a NotImplementedError if it cannot set the start position

pygame.mixer.music.**rewind**()

restart music
rewind() -> None

Resets playback of the current music to the beginning.

pygame.mixer.music.**stop**()

stop the music playback
stop() -> None

Stops the music playback if it is currently playing.

pygame.mixer.music.**pause**()

temporarily stop music playback
pause() -> None

Temporarily stop playback of the music stream. It can be resumed with the `pygame.mixer.music.unpause()` function.

pygame.mixer.music.**unpause**()

resume paused music
unpause() -> None

This will resume the playback of a music stream after it has been paused.

pygame.mixer.music.**fadeout**()

stop music playback after fading out
fadeout(time) -> None

This will stop the music playback after it has been faded out over the specified time (measured in milliseconds).

Note, that this function blocks until the music has faded out.

pygame.mixer.music.**set_volume**()

set the music volume
set_volume(value) -> None

Set the volume of the music playback. The value argument is between 0.0 and 1.0. When new music is loaded the volume is reset.

pygame.mixer.music.**get_volume**()

get the music volume
get_volume() -> value

Returns the current volume for the mixer. The value will be between 0.0 and 1.0.

pygame.mixer.music.**get_busy**()

check if the music stream is playing
get_busy() -> bool

Returns True when the music stream is actively playing. When the music is idle this returns False.

pygame.mixer.music.**set_pos**()

set position to play from

set_pos(pos) -> None

This sets the position in the music file where playback will start. The meaning of "pos", a float (or a number that can be converted to a float), depends on the music format. Newer versions of SDL_mixer have better positioning support than earlier. An SDLError is raised if a particular format does not support positioning.

pygame.mixer.music.**get_pos**()

get the music play time
get_pos() -> time

This gets the number of milliseconds that the music has been playing for. The returned time only represents how long the music has been playing; it does not take into account any starting position offsets.

pygame.mixer.music.**queue**()

queue a music file to follow the current
queue(filename) -> None

This will load a music file and queue it. A queued music file will begin as soon as the current music naturally ends. If the current music is ever stopped or changed, the queued song will be lost.

The following example will play music by Bach six times, then play music by Mozart once:

```
pygame.mixer.music.load('bach.ogg')
pygame.mixer.music.play(5)        # Plays six times, not five!
pygame.mixer.music.queue('mozart.ogg')
```

pygame.mixer.music.**set_endevent**()

have the music send an event when playback stops
set_endevent() -> None
set_endevent(type) -> None

This causes Pygame to signal (by means of the event queue) when the music is done playing. The argument determines the type of event that will be queued.

The event will be queued every time the music finishes, not just the first time. To stop the event from being queued, call this method with no argument.

pygame.mixer.music.**get_endevent**()

get the event a channel sends when playback stops
get_endevent() -> type

Returns the event type to be sent every time the music finishes playback. If there is no endevent the function returns pygame.NOEVENT.

CHAPTER 23

pygame.Overlay

class pygame.**Overlay**

pygame object for video overlay graphics
Overlay(format, (width, height)) -> Overlay

The Overlay objects provide support for accessing hardware video overlays. Video overlays do not use standard RGB pixel formats, and can use multiple resolutions of data to create a single image.

The Overlay objects represent lower level access to the display hardware. To use the object you must understand the technical details of video overlays.

The Overlay format determines the type of pixel data used. Not all hardware will support all types of overlay formats. Here is a list of available format types:

```
YV12_OVERLAY, IYUV_OVERLAY, YUV2_OVERLAY, UYVY_OVERLAY, YVYU_OVERLAY
```

The width and height arguments control the size for the overlay image data. The overlay image can be displayed at any size, not just the resolution of the overlay.

The overlay objects are always visible, and always show above the regular display contents.

display()

set the overlay pixel data
display((y, u, v)) -> None
display() -> None

Display the yuv data in SDL's overlay planes. The y, u, and v arguments are strings of binary data. The data must be in the correct format used to create the Overlay.

If no argument is passed in, the Overlay will simply be redrawn with the current data. This can be useful when the Overlay is not really hardware accelerated.

The strings are not validated, and improperly sized strings could crash the program.

set_location()

control where the overlay is displayed
set_location(rect) -> None

Set the location for the overlay. The overlay will always be shown relative to the main display Surface. This does not actually redraw the overlay, it will be updated on the next call to Overlay.display().

get_hardware()

test if the Overlay is hardware accelerated

get_hardware(rect) -> int

Returns a True value when the Overlay is hardware accelerated. If the platform does not support acceleration, software rendering is used.

CHAPTER 24

pygame.PixelArray

class pygame.**PixelArray**

pygame object for direct pixel access of surfaces
PixelArray(Surface) -> PixelArray

The PixelArray wraps a Surface and provides direct access to the surface's pixels. A pixel array can be one or two dimensional. A two dimensional array, like its surface, is indexed [column, row]. Pixel arrays support slicing, both for returning a subarray or for assignment. A pixel array sliced on a single column or row returns a one dimensional pixel array. Arithmetic and other operations are not supported. A pixel array can be safely assigned to itself. Finally, pixel arrays export an array struct interface, allowing them to interact with pygame.pixelcopy methods and NumPy arrays.

A PixelArray pixel item can be assigned a raw integer values, a pygame.Color instance, or a (r, g, b[, a]) tuple.

```
pxarray[x, y] = 0xFF00FF
pxarray[x, y] = pygame.Color(255, 0, 255)
pxarray[x, y] = (255, 0, 255)
```

However, only a pixel's integer value is returned. So, to compare a pixel to a particular color the color needs to be first mapped using the Surface.map_rgb() method of the Surface object for which the PixelArray was created.

```
pxarray = pygame.PixelArray(surface)
# Check, if the first pixel at the topleft corner is blue
if pxarray[0, 0] == surface.map_rgb((0, 0, 255)):
    ...
```

When assigning to a range of of pixels, a non tuple sequence of colors or a PixelArray can be used as the value. For a sequence, the length must match the PixelArray width.

```
pxarray[a:b] = 0xFF00FF                   # set all pixels to 0xFF00FF
pxarray[a:b] = (0xFF00FF, 0xAACCEE, ... ) # first pixel = 0xFF00FF,
                                          # second pixel  = 0xAACCEE, ...
pxarray[a:b] = [(255, 0, 255), (170, 204, 238), ...] # same as above
pxarray[a:b] = [(255, 0, 255), 0xAACCEE, ...]        # same as above
pxarray[a:b] = otherarray[x:y]            # slice sizes must match
```

For PixelArray assignment, if the right hand side array has a row length of 1, then the column is broadcast over the target array's rows. An array of height 1 is broadcast over the target's columns, and is equivalent to assigning a 1D PixelArray.

Subscipt slices can also be used to assign to a rectangular subview of the target PixelArray.

```
# Create some new PixelArray objects providing a different view
# of the original array/surface.
newarray = pxarray[2:4, 3:5]
otherarray = pxarray[::2, ::2]
```

Subscript slices can also be used to do fast rectangular pixel manipulations instead of iterating over the x or y axis. The

```
pxarray[::2, :] = (0, 0, 0)               # Make even columns black.
pxarray[::2] = (0, 0, 0)                  # Same as [::2, :]
```

During its lifetime, the PixelArray locks the surface, thus you explicitly have to delete it once its not used anymore and the surface should perform operations in the same scope. A simple `:` slice index for the column can be omitted.

```
pxarray[::2, ...] = (0, 0, 0)             # Same as pxarray[::2, :]
pxarray[...] = (255, 0, 0)                # Same as pxarray[:]
```

A note about PixelArray to PixelArray assignment, for arrays with an item size of 3 (created from 24 bit surfaces) pixel values are translated from the source to the destinations format. The red, green, and blue color elements of each pixel are shifted to match the format of the target surface. For all other pixel sizes no such remapping occurs. This should change in later Pygame releases, where format conversions are performed for all pixel sizes. To avoid code breakage when full mapped copying is implemented it is suggested PixelArray to PixelArray copies be only between surfaces of identical format.

New in pygame 1.9.2

- •array struct interface
- •transpose method
- •broadcasting for a length 1 dimension

Changed in pyame 1.9.2

- •A 2D PixelArray can have a length 1 dimension. Only an integer index on a 2D PixelArray returns a 1D array.
- •For assignment, a tuple can only be a color. Any other sequence type is a sequence of colors.

surface

Gets the Surface the PixelArray uses.
surface -> Surface

The Surface the PixelArray was created for.

itemsize

Returns the byte size of a pixel array item
itemsize -> int

This is the same as `Surface.get_bytesize()` for the pixel array's surface.

New in pygame 1.9.2

ndim

Returns the number of dimensions.
ndim -> int

A pixel array can be 1 or 2 dimensional.

New in pygame 1.9.2

shape

Returns the array size.
shape -> tuple of int's

A tuple or length `ndim` giving the length of each dimension. Analogous to `Surface.get_size()`.

New in pygame 1.9.2

strides

Returns byte offsets for each array dimension.
strides -> tuple of int's

A tuple or length `ndim` byte counts. When a stride is multiplied by the corresponding index it gives the offset of that index from the start of the array. A stride is negative for an array that has is inverted (has a negative step).

New in pygame 1.9.2

make_surface()

Creates a new Surface from the current PixelArray.
make_surface() -> Surface

Creates a new Surface from the current PixelArray. Depending on the current PixelArray the size, pixel order etc. will be different from the original Surface.

```
# Create a new surface flipped around the vertical axis.
sf = pxarray[:,::-1].make_surface ()
```

New in pygame 1.8.1.

replace()

Replaces the passed color in the PixelArray with another one.
replace(color, repcolor, distance=0, weights=(0.299, 0.587, 0.114)) -> None

Replaces the pixels with the passed color in the PixelArray by changing them them to the passed replacement color.

It uses a simple weighted euclidian distance formula to calculate the distance between the colors. The distance space ranges from 0.0 to 1.0 and is used as threshold for the color detection. This causes the replacement to take pixels with a similar, but not exactly identical color, into account as well.

This is an in place operation that directly affects the pixels of the PixelArray.

New in pygame 1.8.1.

extract()

Extracts the passed color from the PixelArray.
extract(color, distance=0, weights=(0.299, 0.587, 0.114)) -> PixelArray

Extracts the passed color by changing all matching pixels to white, while non-matching pixels are changed to black. This returns a new PixelArray with the black/white color mask.

It uses a simple weighted euclidian distance formula to calculate the distance between the colors. The distance space ranges from 0.0 to 1.0 and is used as threshold for the color detection. This causes the extraction to take pixels with a similar, but not exactly identical color, into account as well.

New in pygame 1.8.1.

compare()

Compares the PixelArray with another one.
compare(array, distance=0, weights=(0.299, 0.587, 0.114)) -> PixelArray

Compares the contents of the PixelArray with those from the passed PixelArray. It returns a new PixelArray with a black/white color mask that indicates the differences (white) of both arrays. Both PixelArray objects must have indentical bit depths and dimensions.

It uses a simple weighted euclidian distance formula to calculate the distance between the colors. The distance space ranges from 0.0 to 1.0 and is used as threshold for the color detection. This causes the comparision to mark pixels with a similar, but not exactly identical color, as black.

New in pygame 1.8.1.

transpose()

Exchanges the x and y axis.
transpose() -> PixelArray

This method returns a new view of the pixel array with the rows and columns swapped. So for a (w, h) sized array a (h, w) slice is returned. If an array is one dimensional, then a length 1 x dimension is added, resulting in a 2D pixel array.

New in pygame 1.9.2

CHAPTER 25

pygame.pixelcopy

pygame module for general pixel array copying

EXPERIMENTAL, though any future changes are likely to be backward-compatible.

The pygame.pixelcopy module contains methods for copying between surfaces and objects exporting an array structure interface. It is a backend for pygame.surfarray, adding NumPy support. But pixelcopy is more, general, and intended for direct use (see the pygame.examples.pixelcopy.main() example.

The array struct interface exposes an array's data in a standard way. It was introduced in NumPy. In Python 2.7 and above it is replaced by the new buffer protocol, though the buffer protocol is still a work in progress. The array struct interface, on the other hand, is stable and works with earlier Python versions. So for now the array struct interface is the predominate way Pygame handles array introspection.

New in pygame 1.9.2.

pygame.pixelcopy.**surface_to_array**()

copy surface pixels to an array object

surface_to_array(array, surface, kind='P', opaque=255, clear=0) -> None

The surface_to_array function copies pixels from a Surface object to a 2D or 3D array. Depending on argument kind and the target array dimension, a copy may be raw pixel value, RGB, a color component slice, or colorkey alpha transparency value. Recognized kind values are the single character codes 'P', 'R', 'G', 'B', 'A', and 'C'. Kind codes are case insensitive, so 'p' is equivalent to 'P'. The first two dimensions of the target must be the surface size (w, h).

The default 'P' kind code does a direct raw integer pixel (mapped) value copy to a 2D array and a 'RGB' pixel component (unmapped) copy to a 3D array having shape (w, h, 3). For an 8 bit colormap surface this means the table index is copied to a 2D array, not the table value itself. A 2D array's item size must be at least as large as the surface's pixel byte size. The item size of a 3D array must be at least one byte.

For the 'R', 'G', 'B', and 'A' copy kinds a single color component of the unmapped surface pixels are copied to the target 2D array. For kind 'A' and surfaces with source alpha (the surface was created with the SRCALPHA flag), has a colorkey (set with Surface.set_colorkey()), or has a blanket alpha (set with Surface.set_alpha()) then the alpha values are those expected for a SDL surface. If a surface has no explicit alpha value, then the target array is filled with the value of the optional opaque surface_to_array argument (default 255: not transparent).

Copy kind 'C' is a special case for alpha copy of a source surface with colorkey. Unlike the 'A' color component copy, the clear argument value is used for colorkey matches, opaque otherwise. By default, a match has alpha 0 (totally transparent), while everything else is alpha 255 (totally opaque). It is a more general implementation of pygame.surfarray.array_colorkey().

Specific to surface_to_array, a ValueError is raised for target arrays with incorrect shape or item size. A TypeError is raised for an incorrect kind code. Surface specific problems, such as locking, raise a pygame.error.

pygame.pixelcopy.**array_to_surface**()

copy an array object to a surface

array_to_surface(<surface>, <array>) -> None

See pygame.surfarray.blit_array().

pygame.pixelcopy.**map_array**()

copy an array to another array, using surface format

map_array(<array>, <array>, <surface>) -> None

Map an array of color element values - (w, h, ..., 3) - to an array of pixels - (w, h) according to the format of <<urface>.

pygame.pixelcopy.**make_surface**()

Copy an array to a new surface

pygame.pixelcopy.make_surface(array) -> Surface

Create a new Surface that best resembles the data and format of the array. The array can be 2D or 3D with any sized integer values.

CHAPTER 26

pygame

the top level pygame package

The pygame package represents the top-level package for others to use. Pygame itself is broken into many submodules, but this does not affect programs that use Pygame.

As a convenience, most of the top-level variables in pygame have been placed inside a module named 'pygame.locals'. This is meant to be used with 'from `pygame.locals` import *', in addition to 'import pygame'.

When you 'import pygame' all available pygame submodules are automatically imported. Be aware that some of the pygame modules are considered "optional", and may not be available. In that case, Pygame will provide a placeholder object instead of the module, which can be used to test for availability.

`pygame.`**`init`**`()`

initialize all imported pygame modules

init() -> (numpass, numfail)

Initialize all imported Pygame modules. No exceptions will be raised if a module fails, but the total number if successful and failed inits will be returned as a tuple. You can always initialize individual modules manually, but `pygame.init()` is a convenient way to get everything started. The `init()` functions for individual modules will raise exceptions when they fail.

You may want to initalise the different modules seperately to speed up your program or to not use things your game does not.

It is safe to call this `init()` more than once: repeated calls will have no effect. This is true even if you have `pygame.quit()` all the modules.

`pygame.`**`quit`**`()`

uninitialize all pygame modules

quit() -> None

Uninitialize all pygame modules that have previously been initialized. When the Python interpreter shuts down, this method is called regardless, so your program should not need it, except when it wants to terminate its pygame resources and continue. It is safe to call this function more than once: repeated calls have no effect.

Note, that `pygame.quit()` will not exit your program. Consider letting your program end in the same way a normal python program will end.

exception `pygame.`**`error`**

standard pygame exception
raise pygame.error(message)

This exception is raised whenever a pygame or SDL operation fails. You can catch any anticipated problems and deal with the error. The exception is always raised with a descriptive message about the problem.

Derived from the RuntimeError exception, which can also be used to catch these raised errors.

pygame.**get_error**()

get the current error message
get_error() -> errorstr

SDL maintains an internal error message. This message will usually be given to you when pygame.error() is raised. You will rarely need to call this function.

pygame.**set_error**()

set the current error message
set_error(error_msg) -> None

SDL maintains an internal error message. This message will usually be given to you when pygame.error() is raised. You will rarely need to call this function.

pygame.**get_sdl_version**()

get the version number of SDL
get_sdl_version() -> major, minor, patch

Returns the three version numbers of the SDL library. This version is built at compile time. It can be used to detect which features may not be available through Pygame.

get_sdl_version is new in pygame 1.7.0

pygame.**get_sdl_byteorder**()

get the byte order of SDL
get_sdl_byteorder() -> int

Returns the byte order of the SDL library. It returns LIL_ENDIAN for little endian byte order and BIG_ENDIAN for big endian byte order.

get_sdl_byteorder is new in pygame 1.8

pygame.**register_quit**()

register a function to be called when pygame quits
register_quit(callable) -> None

When pygame.quit() is called, all registered quit functions are called. Pygame modules do this automatically when they are initializing. This function is not be needed for regular pygame users.

pygame.**encode_string**()

Encode a unicode or bytes object
encode_string([obj [, encoding [, errors [, etype]]]]) -> bytes or None

obj: If unicode, encode; if bytes, return unaltered; if anything else, return None; if not given, raise SyntaxError.

encoding (string): If present, encoding to use. The default is 'unicode_escape'.

errors (string): If given, how to handle unencodable characters. The default is 'backslashreplace'.

etype (exception type): If given, the exception type to raise for an encoding error. The default is UnicodeEncodeError, as returned by `PyUnicode_AsEncodedString()`. For the default encoding and errors values there should be no encoding errors.

This function is used in encoding file paths. Keyword arguments are supported.

Added in Pygame 1.9.2 (primarily for use in unit tests)

`pygame.`**`encode_file_path`**`()`

Encode a unicode or bytes object as a file system path
encode_file_path([obj [, etype]]) -> bytes or None

obj: If unicode, encode; if bytes, return unaltered; if anything else, return None; if not given, raise SyntaxError.

etype (exception type): If given, the exception type to raise for an encoding error. The default is UnicodeEncodeError, as returned by `PyUnicode_AsEncodedString()`.

This function is used to encode file paths in Pygame. Encoding is to the codec as returned by `sys.getfilesystemencoding()`. Keyword arguments are supported.

Added in Pygame 1.9.2 (primarily for use in unit tests)

CHAPTER 27

pygame.version

small module containing version information

This module is automatically imported into the pygame package and offers a few variables to check with version of pygame has been imported.

pygame.version.**ver**

version number as a string
ver = '1.2'

This is the version represented as a string. It can contain a micro release number as well, e.g., '1.5.2'

pygame.version.**vernum**

tupled integers of the version
vernum = (1, 5, 3)

This variable for the version can easily be compared with other version numbers of the same format. An example of checking Pygame version numbers would look like this:

```
if pygame.version.vernum < (1, 5):
    print 'Warning, older version of Pygame (%s)' %  pygame.version.ver
    disable_advanced_features = True
```

CHAPTER 28

pygame.Rect

class pygame.**Rect**

pygame object for storing rectangular coordinates

Rect(left, top, width, height) -> Rect

Rect((left, top), (width, height)) -> Rect

Rect(object) -> Rect

Pygame uses Rect objects to store and manipulate rectangular areas. A Rect can be created from a combination of left, top, width, and height values. Rects can also be created from python objects that are already a Rect or have an attribute named "rect".

Any Pygame function that requires a Rect argument also accepts any of these values to construct a Rect. This makes it easier to create Rects on the fly as arguments to functions.

The Rect functions that change the position or size of a Rect return a new copy of the Rect with the affected changes. The original Rect is not modified. Some methods have an alternate "in-place" version that returns None but effects the original Rect. These "in-place" methods are denoted with the "ip" suffix.

The Rect object has several virtual attributes which can be used to move and align the Rect:

```
x,y
top, left, bottom, right
topleft, bottomleft, topright, bottomright
midtop, midleft, midbottom, midright
center, centerx, centery
size, width, height
w,h
```

All of these attributes can be assigned to:

```
rect1.right = 10
rect2.center = (20,30)
```

Assigning to size, width or height changes the dimensions of the rectangle; all other assignments move the rectangle without resizing it. Notice that some attributes are integers and others are pairs of integers.

If a Rect has a nonzero width or height, it will return True for a nonzero test. Some methods return a Rect with 0 size to represent an invalid rectangle.

The coordinates for Rect objects are all integers. The size values can be programmed to have negative values, but these are considered illegal Rects for most operations.

There are several collision tests between other rectangles. Most python containers can be searched for collisions against a single Rect.

The area covered by a Rect does not include the right- and bottom-most edge of pixels. If one Rect's bottom border is another Rect's top border (i.e., rect1.bottom=rect2.top), the two meet exactly on the screen but do not overlap, and `rect1.colliderect(rect2)` returns false.

Though Rect can be subclassed, methods that return new rectangles are not subclass aware. That is, move or copy return a new `pygame.Rect` instance, not an instance of the subclass. This may change. To make subclass awareness work though, subclasses may have to maintain the same constructor signature as Rect.

copy()

copy the rectangle
copy() -> Rect

Returns a new rectangle having the same position and size as the original.

New in pygame 1.9

move()

moves the rectangle
move(x, y) -> Rect

Returns a new rectangle that is moved by the given offset. The x and y arguments can be any integer value, positive or negative.

move_ip()

moves the rectangle, in place
move_ip(x, y) -> None

Same as the `Rect.move()` method, but operates in place.

inflate()

grow or shrink the rectangle size
inflate(x, y) -> Rect

Returns a new rectangle with the size changed by the given offset. The rectangle remains centered around its current center. Negative values will shrink the rectangle.

inflate_ip()

grow or shrink the rectangle size, in place
inflate_ip(x, y) -> None

Same as the `Rect.inflate()` method, but operates in place.

clamp()

moves the rectangle inside another
clamp(Rect) -> Rect

Returns a new rectangle that is moved to be completely inside the argument Rect. If the rectangle is too large to fit inside, it is centered inside the argument Rect, but its size is not changed.

clamp_ip()

moves the rectangle inside another, in place
clamp_ip(Rect) -> None

Same as the `Rect.clamp()` method, but operates in place.

clip()

crops a rectangle inside another
clip(Rect) -> Rect

Returns a new rectangle that is cropped to be completely inside the argument Rect. If the two rectangles do not overlap to begin with, a Rect with 0 size is returned.

union()

joins two rectangles into one
union(Rect) -> Rect

Returns a new rectangle that completely covers the area of the two provided rectangles. There may be area inside the new Rect that is not covered by the originals.

union_ip()

joins two rectangles into one, in place
union_ip(Rect) -> None

Same as the `Rect.union()` method, but operates in place.

unionall()

the union of many rectangles
unionall(Rect_sequence) -> Rect

Returns the union of one rectangle with a sequence of many rectangles.

unionall_ip()

the union of many rectangles, in place
unionall_ip(Rect_sequence) -> None

The same as the `Rect.unionall()` method, but operates in place.

fit()

resize and move a rectangle with aspect ratio
fit(Rect) -> Rect

Returns a new rectangle that is moved and resized to fit another. The aspect ratio of the original Rect is preserved, so the new rectangle may be smaller than the target in either width or height.

normalize()

correct negative sizes
normalize() -> None

This will flip the width or height of a rectangle if it has a negative size. The rectangle will remain in the same place, with only the sides swapped.

contains()

test if one rectangle is inside another
contains(Rect) -> bool

Returns true when the argument is completely inside the Rect.

collidepoint()

test if a point is inside a rectangle
collidepoint(x, y) -> bool
collidepoint((x,y)) -> bool

Returns true if the given point is inside the rectangle. A point along the right or bottom edge is not considered to be inside the rectangle.

colliderect()

test if two rectangles overlap
colliderect(Rect) -> bool

Returns true if any portion of either rectangle overlap (except the top+bottom or left+right edges).

collidelist()

test if one rectangle in a list intersects
collidelist(list) -> index

Test whether the rectangle collides with any in a sequence of rectangles. The index of the first collision found is returned. If no collisions are found an index of -1 is returned.

collidelistall()

test if all rectangles in a list intersect
collidelistall(list) -> indices

Returns a list of all the indices that contain rectangles that collide with the Rect. If no intersecting rectangles are found, an empty list is returned.

collidedict()

test if one rectangle in a dictionary intersects
collidedict(dict) -> (key, value)

Returns the key and value of the first dictionary value that collides with the Rect. If no collisions are found, None is returned.

Rect objects are not hashable and cannot be used as keys in a dictionary, only as values.

collidedictall()

test if all rectangles in a dictionary intersect
collidedictall(dict) -> [(key, value), ...]

Returns a list of all the key and value pairs that intersect with the Rect. If no collisions are found an empty dictionary is returned.

Rect objects are not hashable and cannot be used as keys in a dictionary, only as values.

CHAPTER 29

pygame.scrap

pygame module for clipboard support.

EXPERIMENTAL!: meaning this api may change, or dissapear in later pygame releases. If you use this, your code will break with the next pygame release.

The scrap module is for getting and putting stuff from the clipboard. So you can copy and paste things between pygame, and other application types. It defines some basic own data types

```
SCRAP_PPM
SCRAP_PBM
SCRAP_BMP
SCRAP_TEXT
```

to be placed into the clipboard and allows to use define own clipboard types. SCRAP_PPM, SCRAP_PBM and SCRAP_BMP are suitable for surface buffers to be shared with other applications, while SCRAP_TEXT is an alias for the plain text clipboard type.

The SCRAP_* types refer to the following MIME types and register those as well as the default operating system type for this type of data:

```
SCRAP_TEXT text/plain    for plain text
SCRAP_PBM  image/pbm     for PBM encoded image data
SCRAP_PPM  image/ppm     for PPM encoded image data
SCRAP_BMP  image/bmp     for BMP encoded image data
```

Depending on the platform additional types are automatically registered when data is placed into the clipboard to guarantee a consistent sharing behaviour with other applications. The following listed types can be used as string to be passed to the respective pygame.scrap module functions.

For Windows platforms, the additional types are supported automatically and resolve to their internal definitions:

```
text/plain;charset=utf-8 for UTF-8 encoded text
audio/wav                for WAV encoded audio
image/tiff               for TIFF encoded image data
```

For X11 platforms, the additional types are supported automatically and resolve to their internal definitions:

```
UTF8_STRING               for UTF-8 encoded text
text/plain;charset=utf-8  for UTF-8 encoded text
COMPOUND_TEXT             for COMPOUND text
```

As stated before you can define own types for the clipboard, those however might not be usable by other applications. Thus data pasted into the clipboard using

```
pygame.scrap.put ("own_data", data)
```

can only be used by applications, which query the clipboard for the "own_data" type.

New in pygame 1.8. Only works for Windows, X11 and Mac OS X so far. On Mac OSX only text works at the moment - other types will be supported in the next release.

pygame.scrap.**init**()

Initializes the scrap module.

init () -> None

Tries to initialize the scrap module and raises an exception, if it fails. Note that this module requires a set display surface, so you have to make sure, you acquired one earlier using pygame.display.set_mode().

pygame.scrap.**get**()

Gets the data for the specified type from the clipboard.

get (type) -> bytes

Returns the data for the specified type from the clipboard. The data is returned as a byte string and might need further processing, such as decoding to Unicode. If no data for the passed type is available, None is returned.

```
text = pygame.scrap.get (SCRAP_TEXT)
if text:
    # Do stuff with it.
else:
    print "There does not seem to be text in the clipboard."
```

pygame.scrap.**get_types**()

Gets a list of the available clipboard types.

get_types () -> list

Gets a list of strings with the identifiers for the available clipboard types. Each identifier can be used in the scrap.get() method to get the clipboard content of the specific type. If there is no data in the clipboard, an empty list is returned.

```
types = pygame.scrap.get_types ()
for t in types:
    if "text" in t:
        # There is some content with the word "text" in it. It's
        # possibly text, so print it.
        print pygame.scrap.get (t)
```

pygame.scrap.**put**()

Places data into the clipboard.

put(type, data) -> None

Places data for a specific clipboard type into the clipboard. The data must be a string buffer. The type is a string identifying the type of data placed into the clipboard. This can be one of the native SCRAP_PBM, SCRAP_PPM, SCRAP_BMP or SCRAP_TEXT values or an own string identifier.

The method raises an exception, if the content could not be placed into the clipboard.

```
    fp = open ("example.bmp", "rb")
    pygame.scrap.put (SCRAP_BMP, fp.read ())
    fp.close ()
    # Now you can acquire the image data from the clipboard in other
    # applications.
    pygame.scrap.put (SCRAP_TEXT, "A text to copy")
    pygame.scrap.put ("Plain text", "A text to copy")
```

pygame.scrap.**contains**()

Checks, whether a certain type is available in the clipboard.
contains (type) -> bool

Returns True, if data for the passed type is available in the clipboard, False otherwise.

```
    if pygame.scrap.contains (SCRAP_TEXT):
        print "There is text in the clipboard."
    if pygame.scrap.contains ("own_data_type"):
        print "There is stuff in the clipboard."
```

pygame.scrap.**lost**()

Checks whether the clipboard is currently owned by the application.
lost() -> bool

Returns True, if the clipboard is currently owned by the pygame application, False otherwise.

```
    if pygame.scrap.lost ():
       print "No content from me anymore. The clipboard is used by someone else."
```

pygame.scrap.**set_mode**()

Sets the clipboard access mode.
set_mode(mode) -> None

Sets the access mode for the clipboard. This is only of interest for X11 environments, where clipboard modes for mouse selections (SRAP_SELECTION) and the clipboard (SCRAP_CLIPBOARD) are available. Setting the mode to SCRAP_SELECTION in other environments will not cause any difference.

If a value different from SCRAP_CLIPBOARD or SCRAP_SELECTION is passed, a ValueError will be raised.

CHAPTER 30

pygame.sndarray

pygame module for accessing sound sample data

Functions to convert between Numeric or numpy arrays and Sound objects. This module will only be available when pygame can use the external numpy or Numeric package.

Sound data is made of thousands of samples per second, and each sample is the amplitude of the wave at a particular moment in time. For example, in 22-kHz format, element number 5 of the array is the amplitude of the wave after 5/22000 seconds.

Each sample is an 8-bit or 16-bit integer, depending on the data format. A stereo sound file has two values per sample, while a mono sound file only has one.

Supported array systems are

```
numpy
numeric (deprecated; to be removed in Pygame 1.9.3.)
```

The default will be numpy, if installed. Otherwise, Numeric will be set as default if installed, and a deprecation warning will be issued. If neither numpy nor Numeric are installed, the module will raise an ImportError.

The array type to use can be changed at runtime using the `use_arraytype()` method, which requires one of the above types as string.

Note: numpy and Numeric are not completely compatible. Certain array manipulations, which work for one type, might behave differently or even completely break for the other.

Additionally, in contrast to Numeric numpy can use unsigned 16-bit integers. Sounds with 16-bit data will be treated as unsigned integers, if the sound sample type requests this. Numeric instead always uses signed integers for the representation, which is important to keep in mind, if you use the module's functions and wonder about the values.

numpy support added in Pygame 1.8 Official Numeric deprecation begins in Pygame 1.9.2.

`pygame.sndarray.`**`array`**`()`

copy Sound samples into an array

array(Sound) -> array

Creates a new array for the sound data and copies the samples. The array will always be in the format returned from `pygame.mixer.get_init()`.

`pygame.sndarray.`**`samples`**`()`

reference Sound samples into an array

samples(Sound) -> array

Creates a new array that directly references the samples in a Sound object. Modifying the array will change the Sound. The array will always be in the format returned from `pygame.mixer.get_init()`.

`pygame.sndarray.`**`make_sound`**`()`

convert an array into a Sound object
make_sound(array) -> Sound

Create a new playable Sound object from an array. The mixer module must be initialized and the array format must be similar to the mixer audio format.

`pygame.sndarray.`**`use_arraytype`**`()`

Sets the array system to be used for sound arrays
use_arraytype (arraytype) -> None

Uses the requested array type for the module functions. Currently supported array types are:

```
numeric (deprecated; will be removed in Pygame 1.9.3.)
numpy
```

If the requested type is not available, a ValueError will be raised.

New in pygame 1.8.

`pygame.sndarray.`**`get_arraytype`**`()`

Gets the currently active array type.
get_arraytype () -> str

Returns the currently active array type. This will be a value of the `get_arraytypes()` tuple and indicates which type of array module is used for the array creation.

New in pygame 1.8

`pygame.sndarray.`**`get_arraytypes`**`()`

Gets the array system types currently supported.
get_arraytypes () -> tuple

Checks, which array systems are available and returns them as a tuple of strings. The values of the tuple can be used directly in the `pygame.sndarray.use_arraytype()` () method. If no supported array system could be found, None will be returned.

New in pygame 1.8.

CHAPTER 31

pygame.sprite

pygame module with basic game object classes

This module contains several simple classes to be used within games. There is the main Sprite class and several Group classes that contain Sprites. The use of these classes is entirely optional when using Pygame. The classes are fairly lightweight and only provide a starting place for the code that is common to most games.

The Sprite class is intended to be used as a base class for the different types of objects in the game. There is also a base Group class that simply stores sprites. A game could create new types of Group classes that operate on specially customized Sprite instances they contain.

The basic Sprite class can draw the Sprites it contains to a Surface. The `Group.draw()` method requires that each Sprite have a `Surface.image` attribute and a `Surface.rect`. The `Group.clear()` method requires these same attributes, and can be used to erase all the Sprites with background. There are also more advanced Groups: `pygame.sprite.RenderUpdates()` and `pygame.sprite.OrderedUpdates()`.

Lastly, this module contains several collision functions. These help find sprites inside multiple groups that have intersecting bounding rectangles. To find the collisions, the Sprites are required to have a `Surface.rect` attribute assigned.

The groups are designed for high efficiency in removing and adding Sprites to them. They also allow cheap testing to see if a Sprite already exists in a Group. A given Sprite can exist in any number of groups. A game could use some groups to control object rendering, and a completely separate set of groups to control interaction or player movement. Instead of adding type attributes or bools to a derived Sprite class, consider keeping the Sprites inside organized Groups. This will allow for easier lookup later in the game.

Sprites and Groups manage their relationships with the `add()` and `remove()` methods. These methods can accept a single or multiple targets for membership. The default initializers for these classes also takes a single or list of targets for initial membership. It is safe to repeatedly add and remove the same Sprite from a Group.

While it is possible to design sprite and group classes that don't derive from the Sprite and AbstractGroup classes below, it is strongly recommended that you extend those when you add a Sprite or Group class.

Sprites are not thread safe. So lock them yourself if using threads.

class `pygame.sprite.`**Sprite**

Simple base class for visible game objects.

Sprite(*groups) -> Sprite

The base class for visible game objects. Derived classes will want to override the `Sprite.update()` and assign a `Sprite.image` and `Sprite.rect` attributes. The initializer can accept any number of Group instances to be added to.

When subclassing the Sprite, be sure to call the base initializer before adding the Sprite to Groups. For example:

```
class Block(pygame.sprite.Sprite):

    # Constructor. Pass in the color of the block,
    # and its x and y position
    def __init__(self, color, width, height):
       # Call the parent class (Sprite) constructor
       pygame.sprite.Sprite.__init__(self)

       # Create an image of the block, and fill it with a color.
       # This could also be an image loaded from the disk.
       self.image = pygame.Surface([width, height])
       self.image.fill(color)

       # Fetch the rectangle object that has the dimensions of the image
       # Update the position of this object by setting the values of rect.x and rect.y
       self.rect = self.image.get_rect()
```

update()

method to control sprite behavior

update(*args) -> None

The default implementation of this method does nothing; it's just a convenient "hook" that you can override. This method is called by `Group.update()` with whatever arguments you give it.

There is no need to use this method if not using the convenience method by the same name in the Group class.

add()

add the sprite to groups

add(*groups) -> None

Any number of Group instances can be passed as arguments. The Sprite will be added to the Groups it is not already a member of.

remove()

remove the sprite from groups

remove(*groups) -> None

Any number of Group instances can be passed as arguments. The Sprite will be removed from the Groups it is currently a member of.

kill()

remove the Sprite from all Groups

kill() -> None

The Sprite is removed from all the Groups that contain it. This won't change anything about the state of the Sprite. It is possible to continue to use the Sprite after this method has been called, including adding it to Groups.

alive()

does the sprite belong to any groups
alive() -> bool

Returns True when the Sprite belongs to one or more Groups.

groups()

list of Groups that contain this Sprite
groups() -> group_list

Return a list of all the Groups that contain this Sprite.

class pygame.sprite.**DirtySprite**

A subclass of Sprite with more attributes and features.
DirtySprite(*groups) -> DirtySprite

Extra DirtySprite attributes with their default values:

dirty = 1

```
if set to 1, it is repainted and then set to 0 again
if set to 2 then it is always dirty ( repainted each frame,
flag is not reset)
0 means that it is not dirty and therefor not repainted again
```

blendmode = 0

```
its the special_flags argument of blit, blendmodes
```

source_rect = None

```
source rect to use, remember that it is relative to
topleft (0,0) of self.image
```

visible = 1

```
normally 1, if set to 0 it will not be repainted
(you must set it dirty too to be erased from screen)
```

layer = 0

```
(READONLY value, it is read when adding it to the
LayeredDirty, for details see doc of LayeredDirty)
```

class pygame.sprite.**Group**

A container class to hold and manage multiple Sprite objects.
Group(*sprites) -> Group

A simple container for Sprite objects. This class can be inherited to create containers with more specific behaviors. The constructor takes any number of Sprite arguments to add to the Group. The group supports the following standard Python operations:

```
in      test if a Sprite is contained
len     the number of Sprites contained
bool    test if any Sprites are contained
iter    iterate through all the Sprites
```

The Sprites in the Group are not ordered, so drawing and iterating the Sprites is in no particular order.

sprites()

list of the Sprites this Group contains
sprites() -> sprite_list

Return a list of all the Sprites this group contains. You can also get an iterator from the group, but you cannot iterator over a Group while modifying it.

copy()

duplicate the Group
copy() -> Group

Creates a new Group with all the same Sprites as the original. If you have subclassed Group, the new object will have the same (sub-)class as the original. This only works if the derived class's constructor takes the same arguments as the Group class's.

add()

add Sprites to this Group
add(*sprites) -> None

Add any number of Sprites to this Group. This will only add Sprites that are not already members of the Group.

Each sprite argument can also be a iterator containing Sprites.

remove()

remove Sprites from the Group
remove(*sprites) -> None

Remove any number of Sprites from the Group. This will only remove Sprites that are already members of the Group.

Each sprite argument can also be a iterator containing Sprites.

has()

test if a Group contains Sprites
has(*sprites) -> None

Return True if the Group contains all of the given sprites. This is similar to using the "in" operator on the Group ("if sprite in group: ..."), which tests if a single Sprite belongs to a Group.

Each sprite argument can also be a iterator containing Sprites.

update()

call the update method on contained Sprites
update(*args) -> None

Calls the `update()` method on all Sprites in the Group. The base Sprite class has an update method that takes any number of arguments and does nothing. The arguments passed to `Group.update()` will be passed to each Sprite.

There is no way to get the return value from the `Sprite.update()` methods.

draw()

blit the Sprite images
draw(Surface) -> None

Draws the contained Sprites to the Surface argument. This uses the `Sprite.image` attribute for the source surface, and `Sprite.rect` for the position.

The Group does not keep sprites in any order, so the draw order is arbitrary.

clear()

draw a background over the Sprites
clear(Surface_dest, background) -> None

Erases the Sprites used in the last `Group.draw()` call. The destination Surface is cleared by filling the drawn Sprite positions with the background.

The background is usually a Surface image the same dimensions as the destination Surface. However, it can also be a callback function that takes two arguments; the destination Surface and an area to clear. The background callback function will be called several times each clear.

Here is an example callback that will clear the Sprites with solid red:

```
def clear_callback(surf, rect):
    color = 255, 0, 0
    surf.fill(color, rect)
```

empty()

remove all Sprites
empty() -> None

Removes all Sprites from this Group.

class `pygame.sprite.`**RenderPlain**

Same as pygame.sprite.Group

This class is an alias to `pygame.sprite.Group()`. It has no additional functionality.

class `pygame.sprite.`**RenderClear**

Same as pygame.sprite.Group

This class is an alias to `pygame.sprite.Group()`. It has no additional functionality.

class `pygame.sprite.`**RenderUpdates**

Group sub-class that tracks dirty updates.
RenderUpdates(*sprites) -> RenderUpdates

This class is derived from `pygame.sprite.Group()`. It has an extended `draw()` method that tracks the changed areas of the screen.

draw()

blit the Sprite images and track changed areas

draw(surface) -> Rect_list

Draws all the Sprites to the surface, the same as `Group.draw()`. This method also returns a list of Rectangular areas on the screen that have been changed. The returned changes include areas of the screen that have been affected by previous `Group.clear()` calls.

The returned Rect list should be passed to `pygame.display.update()`. This will help performance on software driven display modes. This type of updating is usually only helpful on destinations with non-animating backgrounds.

`pygame.sprite.`**`OrderedUpdates`**`()`

RenderUpdates sub-class that draws Sprites in order of addition.

OrderedUpdates(*spites) -> OrderedUpdates

This class derives from `pygame.sprite.RenderUpdates()`. It maintains the order in which the Sprites were added to the Group for rendering. This makes adding and removing Sprites from the Group a little slower than regular Groups.

class `pygame.sprite.`**`LayeredUpdates`**

LayeredUpdates is a sprite group that handles layers and draws like OrderedUpdates.

LayeredUpdates(*spites, **kwargs) -> LayeredUpdates

This group is fully compatible with `pygame.sprite.Sprite`.

You can set the default layer through kwargs using 'default_layer' and an integer for the layer. The default layer is 0.

If the sprite you add has an attribute layer then that layer will be used. If the **kwarg contains 'layer' then the sprites passed will be added to that layer (overriding the `sprite.layer` attribute). If neither sprite has attribute layer nor **kwarg then the default layer is used to add the sprites.

New in pygame 1.8.0

`add``()`

add a sprite or sequence of sprites to a group

add(*sprites, **kwargs) -> None

If the `sprite(s)` have an attribute layer then that is used for the layer. If **kwargs contains 'layer' then the `sprite(s)` will be added to that argument (overriding the sprite layer attribute). If neither is passed then the `sprite(s)` will be added to the default layer.

`sprites``()`

returns a ordered list of sprites (first back, last top).

sprites() -> sprites

`draw``()`

draw all sprites in the right order onto the passed surface.

draw(surface) -> Rect_list

`get_sprites_at``()`

returns a list with all sprites at that position.

get_sprites_at(pos) -> colliding_sprites

Bottom sprites first, top last.

get_sprite()

returns the sprite at the index idx from the groups sprites
get_sprite(idx) -> sprite

Raises IndexOutOfBounds if the idx is not within range.

remove_sprites_of_layer()

removes all sprites from a layer and returns them as a list.
remove_sprites_of_layer(layer_nr) -> sprites

layers()

returns a list of layers defined (unique), sorted from botton up.
layers() -> layers

change_layer()

changes the layer of the sprite
change_layer(sprite, new_layer) -> None

sprite must have been added to the renderer. It is not checked.

get_layer_of_sprite()

returns the layer that sprite is currently in.
get_layer_of_sprite(sprite) -> layer

If the sprite is not found then it will return the default layer.

get_top_layer()

returns the top layer
get_top_layer() -> layer

get_bottom_layer()

returns the bottom layer
get_bottom_layer() -> layer

move_to_front()

brings the sprite to front layer
move_to_front(sprite) -> None

Brings the sprite to front, changing sprite layer to topmost layer (added at the end of that layer).

move_to_back()

moves the sprite to the bottom layer

move_to_back(sprite) -> None

Moves the sprite to the bottom layer, moving it behind all other layers and adding one additional layer.

get_top_sprite()

returns the topmost sprite
get_top_sprite() -> Sprite

get_sprites_from_layer()

returns all sprites from a layer, ordered by how they where added
get_sprites_from_layer(layer) -> sprites

Returns all sprites from a layer, ordered by how they where added. It uses linear search and the sprites are not removed from layer.

switch_layer()

switches the sprites from layer1 to layer2
switch_layer(layer1_nr, layer2_nr) -> None

The layers number must exist, it is not checked.

class pygame.sprite.**LayeredDirty**

LayeredDirty group is for DirtySprite objects. Subclasses LayeredUpdates.
LayeredDirty(*spites, **kwargs) -> LayeredDirty

This group requires pygame.sprite.DirtySprite or any sprite that has the following attributes:

```
image, rect, dirty, visible, blendmode (see doc of DirtySprite).
```

It uses the dirty flag technique and is therefore faster than the pygame.sprite.RenderUpdates if you have many static sprites. It also switches automatically between dirty rect update and full screen drawing, so you do no have to worry what would be faster.

Same as for the pygame.sprite.Group. You can specify some additional attributes through kwargs:

```
_use_update: True/False   default is False
_default_layer: default layer where sprites without a layer are added.
_time_threshold: treshold time for switching between dirty rect mode
    and fullscreen mode, defaults to 1000./80  == 1000./fps
```

New in pygame 1.8.0

draw()

draw all sprites in the right order onto the passed surface.
draw(surface, bgd=None) -> Rect_list

You can pass the background too. If a background is already set, then the bgd argument has no effect.

clear()

used to set background
clear(surface, bgd) -> None

repaint_rect()

repaints the given area
repaint_rect(screen_rect) -> None

screen_rect is in screencoordinates.

set_clip()

clip the area where to draw. Just pass None (default) to reset the clip
set_clip(screen_rect=None) -> None

get_clip()

clip the area where to draw. Just pass None (default) to reset the clip
get_clip() -> Rect

change_layer()

changes the layer of the sprite
change_layer(sprite, new_layer) -> None

sprite must have been added to the renderer. It is not checked.

set_timing_treshold()

sets the treshold in milliseconds
set_timing_treshold(time_ms) -> None

Default is 1000./80 where 80 is the fps I want to switch to full screen mode.

pygame.sprite.**GroupSingle**()

Group container that holds a single sprite.
GroupSingle(sprite=None) -> GroupSingle

The GroupSingle container only holds a single Sprite. When a new Sprite is added, the old one is removed.

There is a special property, `GroupSingle.sprite`, that accesses the Sprite that this Group contains. It can be None when the Group is empty. The property can also be assigned to add a Sprite into the GroupSingle container.

pygame.sprite.**spritecollide**()

Find sprites in a group that intersect another sprite.
spritecollide(sprite, group, dokill, collided = None) -> Sprite_list

Return a list containing all Sprites in a Group that intersect with another Sprite. Intersection is determined by comparing the `Sprite.rect` attribute of each Sprite.

The dokill argument is a bool. If set to True, all Sprites that collide will be removed from the Group.

The collided argument is a callback function used to calculate if two sprites are colliding. it should take two sprites as values, and return a bool value indicating if they are colliding. If collided is not passed, all sprites must have a "rect" value, which is a rectangle of the sprite area, which will be used to calculate the collision.

collided callables:

```
collide_rect, collide_rect_ratio, collide_circle,
collide_circle_ratio, collide_mask
```

Example:

```
# See if the Sprite block has collided with anything in the Group block_list
# The True flag will remove the sprite in block_list
blocks_hit_list = pygame.sprite.spritecollide(player, block_list, True)

# Check the list of colliding sprites, and add one to the score for each one
for block in blocks_hit_list:
    score +=1
```

pygame.sprite.**collide_rect**()

Collision detection between two sprites, using rects.
collide_rect(left, right) -> bool

Tests for collision between two sprites. Uses the pygame rect colliderect function to calculate the collision. Intended to be passed as a collided callback function to the *collide functions. Sprites must have a "rect" attributes.

New in pygame 1.8.0

pygame.sprite.**collide_rect_ratio**()

Collision detection between two sprites, using rects scaled to a ratio.
collide_rect_ratio(ratio) -> collided_callable

A callable class that checks for collisions between two sprites, using a scaled version of the sprites rects.

Is created with a ratio, the instance is then intended to be passed as a collided callback function to the *collide functions.

A ratio is a floating point number - 1.0 is the same size, 2.0 is twice as big, and 0.5 is half the size.

New in pygame 1.8.1

pygame.sprite.**collide_circle**()

Collision detection between two sprites, using circles.
collide_circle(left, right) -> bool

Tests for collision between two sprites, by testing to see if two circles centered on the sprites overlap. If the sprites have a "radius" attribute, that is used to create the circle, otherwise a circle is created that is big enough to completely enclose the sprites rect as given by the "rect" attribute. Intended to be passed as a collided callback function to the *collide functions. Sprites must have a "rect" and an optional "radius" attribute.

New in pygame 1.8.1

pygame.sprite.**collide_circle_ratio**()

Collision detection between two sprites, using circles scaled to a ratio.
collide_circle_ratio(ratio) -> collided_callable

A callable class that checks for collisions between two sprites, using a scaled version of the sprites radius.

Is created with a floating point ratio, the instance is then intended to be passed as a collided callback function to the *collide functions.

A ratio is a floating point number - 1.0 is the same size, 2.0 is twice as big, and 0.5 is half the size.

The created callable tests for collision between two sprites, by testing to see if two circles centered on the sprites overlap, after scaling the circles radius by the stored ratio. If the sprites have a "radius" attribute, that is used to create the circle, otherwise a circle is created that is big enough to completely enclose the sprites rect as given by the "rect" attribute. Intended to be passed as a collided callback function to the *collide functions. Sprites must have a "rect" and an optional "radius" attribute.

New in pygame 1.8.1

`pygame.sprite.`**`collide_mask`**`()`

Collision detection between two sprites, using masks.
collide_mask(SpriteLeft, SpriteRight) -> point

Returns first point on the mask where the masks collided, or None if there was no collision.

Tests for collision between two sprites, by testing if thier bitmasks overlap. If the sprites have a "mask" attribute, that is used as the mask, otherwise a mask is created from the sprite image. Intended to be passed as a collided callback function to the *collide functions. Sprites must have a "rect" and an optional "mask" attribute.

You should consider creating a mask for your sprite at load time if you are going to check collisions many times. This will increase the performance, otherwise this can be an expensive function because it will create the masks each time you check for collisions.

```
sprite.mask = pygame.mask.from_surface(sprite.image)
```

New in pygame 1.8.0

`pygame.sprite.`**`groupcollide`**`()`

Find all sprites that collide between two groups.
groupcollide(group1, group2, dokill1, dokill2, collided = None) -> Sprite_dict

This will find collisions between all the Sprites in two groups. Collision is determined by comparing the `Sprite.rect` attribute of each Sprite or by using the collided function if it is not None.

Every Sprite inside group1 is added to the return dictionary. The value for each item is the list of Sprites in group2 that intersect.

If either dokill argument is True, the colliding Sprites will be removed from their respective Group.

The collided argument is a callback function used to calculate if two sprites are colliding. It should take two sprites as values and return a bool value indicating if they are colliding. If collided is not passed, then all sprites must have a "rect" value, which is a rectangle of the sprite area, which will be used to calculate the collision.

`pygame.sprite.`**`spritecollideany`**`()`

Simple test if a sprite intersects anything in a group.
spritecollideany(sprite, group, collided = None) -> Sprite Collision with the returned sprite.
spritecollideany(sprite, group, collided = None) -> None No collision

If the sprite collides with any single sprite in the group, a single sprite from the group is returned. On no collision None is returned.

If you don't need all the features of the `pygame.sprite.spritecollide()` function, this function will be a bit quicker.

The collided argument is a callback function used to calculate if two sprites are colliding. It should take two sprites as values and return a bool value indicating if they are colliding. If collided is not passed, then all sprites must have a "rect" value, which is a rectangle of the sprite area, which will be used to calculate the collision.

CHAPTER 32

pygame.Surface

class pygame.**Surface**

pygame object for representing images

Surface((width, height), flags=0, depth=0, masks=None) -> Surface

Surface((width, height), flags=0, Surface) -> Surface

A pygame Surface is used to represent any image. The Surface has a fixed resolution and pixel format. Surfaces with 8bit pixels use a color palette to map to 24bit color.

Call pygame.Surface() to create a new image object. The Surface will be cleared to all black. The only required arguments are the sizes. With no additional arguments, the Surface will be created in a format that best matches the display Surface.

The pixel format can be controlled by passing the bit depth or an existing Surface. The flags argument is a bitmask of additional features for the surface. You can pass any combination of these flags:

```
HWSURFACE, creates the image in video memory
SRCALPHA, the pixel format will include a per-pixel alpha
```

Both flags are only a request, and may not be possible for all displays and formats.

Advance users can combine a set of bitmasks with a depth value. The masks are a set of 4 integers representing which bits in a pixel will represent each color. Normal Surfaces should not require the masks argument.

Surfaces can have many extra attributes like alpha planes, colorkeys, source rectangle clipping. These functions mainly effect how the Surface is blitted to other Surfaces. The blit routines will attempt to use hardware acceleration when possible, otherwise they will use highly optimized software blitting methods.

There are three types of transparency supported in Pygame: colorkeys, surface alphas, and pixel alphas. Surface alphas can be mixed with colorkeys, but an image with per pixel alphas cannot use the other modes. Colorkey transparency makes a single color value transparent. Any pixels matching the colorkey will not be drawn. The surface alpha value is a single value that changes the transparency for the entire image. A surface alpha of 255 is opaque, and a value of 0 is completely transparent.

Per pixel alphas are different because they store a transparency value for every pixel. This allows for the most precise transparency effects, but it also the slowest. Per pixel alphas cannot be mixed with surface alpha and colorkeys.

There is support for pixel access for the Surfaces. Pixel access on hardware surfaces is slow and not recommended. Pixels can be accessed using the get_at() and set_at() functions. These methods are fine for simple access, but will be considerably slow when doing of pixel work with them. If you plan on doing a lot of pixel level work, it is recommended to use a pygame.PixelArray, which gives an array like view of the

surface. For involved mathematical manipulations try the pygame.surfarray module (It's quite quick, but requires NumPy.)

Any functions that directly access a surface's pixel data will need that surface to be lock()'ed. These functions can lock() and unlock() the surfaces themselves without assistance. But, if a function will be called many times, there will be a lot of overhead for multiple locking and unlocking of the surface. It is best to lock the surface manually before making the function call many times, and then unlocking when you are finished. All functions that need a locked surface will say so in their docs. Remember to leave the Surface locked only while necessary.

Surface pixels are stored internally as a single number that has all the colors encoded into it. Use the Surface.map_rgb() and Surface.unmap_rgb() to convert between individual red, green, and blue values into a packed integer for that Surface.

Surfaces can also reference sections of other Surfaces. These are created with the Surface.subsurface() method. Any change to either Surface will effect the other.

Each Surface contains a clipping area. By default the clip area covers the entire Surface. If it is changed, all drawing operations will only effect the smaller area.

blit()

draw one image onto another
blit(source, dest, area=None, special_flags = 0) -> Rect

Draws a source Surface onto this Surface. The draw can be positioned with the dest argument. Dest can either be pair of coordinates representing the upper left corner of the source. A Rect can also be passed as the destination and the topleft corner of the rectangle will be used as the position for the blit. The size of the destination rectangle does not effect the blit.

An optional area rectangle can be passed as well. This represents a smaller portion of the source Surface to draw.

An optional special flags is for passing in new in 1.8.0: BLEND_ADD, BLEND_SUB, BLEND_MULT, BLEND_MIN, BLEND_MAX new in 1.8.1: BLEND_RGBA_ADD, BLEND_RGBA_SUB, BLEND_RGBA_MULT, BLEND_RGBA_MIN, BLEND_RGBA_MAX BLEND_RGB_ADD, BLEND_RGB_SUB, BLEND_RGB_MULT, BLEND_RGB_MIN, BLEND_RGB_MAX With other special blitting flags perhaps added in the future.

The return rectangle is the area of the affected pixels, excluding any pixels outside the destination Surface, or outside the clipping area.

Pixel alphas will be ignored when blitting to an 8 bit Surface.

special_flags new in pygame 1.8.

For a surface with colorkey or blanket alpha, a blit to self may give slightly different colors than a non self-blit.

convert()

change the pixel format of an image
convert(Surface) -> Surface
convert(depth, flags=0) -> Surface
convert(masks, flags=0) -> Surface
convert() -> Surface

Creates a new copy of the Surface with the pixel format changed. The new pixel format can be determined from another existing Surface. Otherwise depth, flags, and masks arguments can be used, similar to the `pygame.Surface()` call.

If no arguments are passed the new Surface will have the same pixel format as the display Surface. This is always the fastest format for blitting. It is a good idea to convert all Surfaces before they are blitted many times.

The converted Surface will have no pixel alphas. They will be stripped if the original had them. See `Surface.convert_alpha()` for preserving or creating per-pixel alphas.

convert_alpha()

change the pixel format of an image including per pixel alphas

convert_alpha(Surface) -> Surface

convert_alpha() -> Surface

Creates a new copy of the surface with the desired pixel format. The new surface will be in a format suited for quick blitting to the given format with per pixel alpha. If no surface is given, the new surface will be optimized for blitting to the current display.

Unlike the `Surface.convert()` method, the pixel format for the new image will not be exactly the same as the requested source, but it will be optimized for fast alpha blitting to the destination.

copy()

create a new copy of a Surface

copy() -> Surface

Makes a duplicate copy of a Surface. The new Surface will have the same pixel formats, color palettes, and transparency settings as the original.

fill()

fill Surface with a solid color

fill(color, rect=None, special_flags=0) -> Rect

Fill the Surface with a solid color. If no rect argument is given the entire Surface will be filled. The rect argument will limit the fill to a specific area. The fill will also be contained by the Surface clip area.

The color argument can be either a `RGB` sequence, a `RGBA` sequence or a mapped color index. If using `RGBA`, the Alpha (A part of `RGBA`) is ignored unless the surface uses per pixel alpha (Surface has the `SRCALPHA` flag).

An optional special_flags is for passing in new in 1.8.0: `BLEND_ADD`, `BLEND_SUB`, `BLEND_MULT`, `BLEND_MIN`, `BLEND_MAX` new in 1.8.1: `BLEND_RGBA_ADD`, `BLEND_RGBA_SUB`, `BLEND_RGBA_MULT`, `BLEND_RGBA_MIN`, `BLEND_RGBA_MAX` `BLEND_RGB_ADD`, `BLEND_RGB_SUB`, `BLEND_RGB_MULT`, `BLEND_RGB_MIN`, `BLEND_RGB_MAX` With other special blitting flags perhaps added in the future.

This will return the affected Surface area.

scroll()

Shift the surface image in place

scroll(dx=0, dy=0) -> None

Move the image by dx pixels right and dy pixels down. dx and dy may be negative for left and up scrolls respectively. Areas of the surface that are not overwritten retain their original pixel values. Scrolling is contained by the Surface clip area. It is safe to have dx and dy values that exceed the surface size.

New in Pygame 1.9

set_colorkey()

Set the transparent colorkey
set_colorkey(Color, flags=0) -> None
set_colorkey(None) -> None

Set the current color key for the Surface. When blitting this Surface onto a destination, and pixels that have the same color as the colorkey will be transparent. The color can be an RGB color or a mapped color integer. If None is passed, the colorkey will be unset.

The colorkey will be ignored if the Surface is formatted to use per pixel alpha values. The colorkey can be mixed with the full Surface alpha value.

The optional flags argument can be set to pygame.RLEACCEL to provide better performance on non accelerated displays. An RLEACCEL Surface will be slower to modify, but quicker to blit as a source.

get_colorkey()

Get the current transparent colorkey
get_colorkey() -> RGB or None

Return the current colorkey value for the Surface. If the colorkey is not set then None is returned.

set_alpha()

set the alpha value for the full Surface image
set_alpha(value, flags=0) -> None
set_alpha(None) -> None

Set the current alpha value fo r the Surface. When blitting this Surface onto a destination, the pixels will be drawn slightly transparent. The alpha value is an integer from 0 to 255, 0 is fully transparent and 255 is fully opaque. If None is passed for the alpha value, then the Surface alpha will be disabled.

This value is different than the per pixel Surface alpha. If the Surface format contains per pixel alphas, then this alpha value will be ignored. If the Surface contains per pixel alphas, setting the alpha value to None will disable the per pixel transparency.

The optional flags argument can be set to pygame.RLEACCEL to provide better performance on non accelerated displays. An RLEACCEL Surface will be slower to modify, but quicker to blit as a source.

get_alpha()

get the current Surface transparency value
get_alpha() -> int_value or None

Return the current alpha value for the Surface. If the alpha value is not set then None is returned.

lock()

lock the Surface memory for pixel access
lock() -> None

Lock the pixel data of a Surface for access. On accelerated Surfaces, the pixel data may be stored in volatile video memory or nonlinear compressed forms. When a Surface is locked the pixel memory becomes available to access by regular software. Code that reads or writes pixel values will need the Surface to be locked.

Surfaces should not remain locked for more than necessary. A locked Surface can often not be displayed or managed by Pygame.

Not all Surfaces require locking. The `Surface.mustlock()` method can determine if it is actually required. There is no performance penalty for locking and unlocking a Surface that does not need it.

All pygame functions will automatically lock and unlock the Surface data as needed. If a section of code is going to make calls that will repeatedly lock and unlock the Surface many times, it can be helpful to wrap the block inside a lock and unlock pair.

It is safe to nest locking and unlocking calls. The surface will only be unlocked after the final lock is released.

unlock()

unlock the Surface memory from pixel access
unlock() -> None

Unlock the Surface pixel data after it has been locked. The unlocked Surface can once again be drawn and managed by Pygame. See the `Surface.lock()` documentation for more details.

All pygame functions will automatically lock and unlock the Surface data as needed. If a section of code is going to make calls that will repeatedly lock and unlock the Surface many times, it can be helpful to wrap the block inside a lock and unlock pair.

It is safe to nest locking and unlocking calls. The surface will only be unlocked after the final lock is released.

mustlock()

test if the Surface requires locking
mustlock() -> bool

Returns True if the Surface is required to be locked to access pixel data. Usually pure software Surfaces do not require locking. This method is rarely needed, since it is safe and quickest to just lock all Surfaces as needed.

All pygame functions will automatically lock and unlock the Surface data as needed. If a section of code is going to make calls that will repeatedly lock and unlock the Surface many times, it can be helpful to wrap the block inside a lock and unlock pair.

get_locked()

test if the Surface is current locked
get_locked() -> bool

Returns True when the Surface is locked. It doesn't matter how many times the Surface is locked.

get_locks()

Gets the locks for the Surface
get_locks() -> tuple

Returns the currently existing locks for the Surface.

get_at()

get the color value at a single pixel
get_at((x, y)) -> Color

Return a copy of the RGBA Color value at the given pixel. If the Surface has no per pixel alpha, then the alpha value will always be 255 (opaque). If the pixel position is outside the area of the Surface an IndexError exception will be raised.

Getting and setting pixels one at a time is generally too slow to be used in a game or realtime situation. It is better to use methods which operate on many pixels at a time like with the blit, fill and draw methods - or by using surfarray/PixelArray.

This function will temporarily lock and unlock the Surface as needed.

Returning a Color instead of tuple, New in pygame 1.9.0. Use `tuple(surf.get_at((x,y)))` if you want a tuple, and not a Color. This should only matter if you want to use the color as a key in a dict.

set_at()

set the color value for a single pixel
set_at((x, y), Color) -> None

Set the RGBA or mapped integer color value for a single pixel. If the Surface does not have per pixel alphas, the alpha value is ignored. Settting pixels outside the Surface area or outside the Surface clipping will have no effect.

Getting and setting pixels one at a time is generally too slow to be used in a game or realtime situation.

This function will temporarily lock and unlock the Surface as needed.

get_at_mapped()

get the mapped color value at a single pixel
get_at_mapped((x, y)) -> Color

Return the integer value of the given pixel. If the pixel position is outside the area of the Surface an IndexError exception will be raised.

This method is intended for Pygame unit testing. It unlikely has any use in an application.

This function will temporarily lock and unlock the Surface as needed.

New in pygame. 1.9.2.

get_palette()

get the color index palette for an 8bit Surface
get_palette() -> [RGB, RGB, RGB, ...]

Return a list of up to 256 color elements that represent the indexed colors used in an 8bit Surface. The returned list is a copy of the palette, and changes will have no effect on the Surface.

Returning a list of `Color(with length 3)` instances instead of tuples, New in pygame 1.9.0

get_palette_at()

get the color for a single entry in a palette
get_palette_at(index) -> RGB

Returns the red, green, and blue color values for a single index in a Surface palette. The index should be a value from 0 to 255.

Returning `Color(with length 3)` instance instead of a tuple, New in pygame 1.9.0

set_palette()

set the color palette for an 8bit Surface
set_palette([RGB, RGB, RGB, ...]) -> None

Set the full palette for an 8bit Surface. This will replace the colors in the existing palette. A partial palette can be passed and only the first colors in the original palette will be changed.

This function has no effect on a Surface with more than 8bits per pixel.

set_palette_at()

set the color for a single index in an 8bit Surface palette
set_palette_at(index, RGB) -> None

Set the palette value for a single entry in a Surface palette. The index should be a value from 0 to 255.

This function has no effect on a Surface with more than 8bits per pixel.

map_rgb()

convert a color into a mapped color value
map_rgb(Color) -> mapped_int

Convert an `RGBA` color into the mapped integer value for this Surface. The returned integer will contain no more bits than the bit depth of the Surface. Mapped color values are not often used inside Pygame, but can be passed to most functions that require a Surface and a color.

See the Surface object documentation for more information about colors and pixel formats.

unmap_rgb()

convert a mapped integer color value into a Color
unmap_rgb(mapped_int) -> Color

Convert an mapped integer color into the `RGB` color components for this Surface. Mapped color values are not often used inside Pygame, but can be passed to most functions that require a Surface and a color.

See the Surface object documentation for more information about colors and pixel formats.

set_clip()

set the current clipping area of the Surface
set_clip(rect) -> None
set_clip(None) -> None

Each Surface has an active clipping area. This is a rectangle that represents the only pixels on the Surface that can be modified. If None is passed for the rectangle the full Surface will be available for changes.

The clipping area is always restricted to the area of the Surface itself. If the clip rectangle is too large it will be shrunk to fit inside the Surface.

get_clip()

get the current clipping area of the Surface
get_clip() -> Rect

Return a rectangle of the current clipping area. The Surface will always return a valid rectangle that will never be outside the bounds of the image. If the Surface has had None set for the clipping area, the Surface will return a rectangle with the full area of the Surface.

subsurface()

create a new surface that references its parent
subsurface(Rect) -> Surface

Returns a new Surface that shares its pixels with its new parent. The new Surface is considered a child of the original. Modifications to either Surface pixels will effect each other. Surface information like clipping area and color keys are unique to each Surface.

The new Surface will inherit the palette, color key, and alpha settings from its parent.

It is possible to have any number of subsurfaces and subsubsurfaces on the parent. It is also possible to subsurface the display Surface if the display mode is not hardware accelerated.

See the `Surface.get_offset()`, `Surface.get_parent()` to learn more about the state of a subsurface.

get_parent()

find the parent of a subsurface
get_parent() -> Surface

Returns the parent Surface of a subsurface. If this is not a subsurface then None will be returned.

get_abs_parent()

find the top level parent of a subsurface
get_abs_parent() -> Surface

Returns the parent Surface of a subsurface. If this is not a subsurface then this surface will be returned.

get_offset()

find the position of a child subsurface inside a parent
get_offset() -> (x, y)

Get the offset position of a child subsurface inside of a parent. If the Surface is not a subsurface this will return (0, 0).

get_abs_offset()

find the absolute position of a child subsurface inside its top level parent
get_abs_offset() -> (x, y)

Get the offset position of a child subsurface inside of its top level parent Surface. If the Surface is not a subsurface this will return (0, 0).

get_size()

get the dimensions of the Surface

get_size() -> (width, height)

Return the width and height of the Surface in pixels.

get_width()

get the width of the Surface
get_width() -> width

Return the width of the Surface in pixels.

get_height()

get the height of the Surface
get_height() -> height

Return the height of the Surface in pixels.

get_rect()

get the rectangular area of the Surface
get_rect(**kwargs) -> Rect

Returns a new rectangle covering the entire surface. This rectangle will always start at 0, 0 with a width. and height the same size as the image.

You can pass keyword argument values to this function. These named values will be applied to the attributes of the Rect before it is returned. An example would be 'mysurf.get_rect(center=(100,100))' to create a rectangle for the Surface centered at a given position.

get_bitsize()

get the bit depth of the Surface pixel format
get_bitsize() -> int

Returns the number of bits used to represent each pixel. This value may not exactly fill the number of bytes used per pixel. For example a 15 bit Surface still requires a full 2 bytes.

get_bytesize()

get the bytes used per Surface pixel
get_bytesize() -> int

Return the number of bytes used per pixel.

get_flags()

get the additional flags used for the Surface
get_flags() -> int

Returns a set of current Surface features. Each feature is a bit in the flags bitmask. Typical flags are `HWSURFACE`, `RLEACCEL`, `SRCALPHA`, and `SRCCOLORKEY`.

Here is a more complete list of flags. A full list can be found in `SDL_video.h`

```
SWSURFACE       0x00000000      # Surface is in system memory
HWSURFACE       0x00000001      # Surface is in video memory
ASYNCBLIT       0x00000004      # Use asynchronous blits if possible
```

Available for `pygame.display.set_mode()`

```
ANYFORMAT       0x10000000      # Allow any video depth/pixel-format
HWPALETTE       0x20000000      # Surface has exclusive palette
DOUBLEBUF       0x40000000      # Set up double-buffered video mode
FULLSCREEN      0x80000000      # Surface is a full screen display
OPENGL        0x00000002      # Create an OpenGL rendering context
OPENGLBLIT      0x0000000A      # Create an OpenGL rendering context
                              #   and use it for blitting.  Obsolete.
RESIZABLE       0x00000010      # This video mode may be resized
NOFRAME       0x00000020        # No window caption or edge frame
```

Used internally (read-only)

```
HWACCEL       0x00000100        # Blit uses hardware acceleration
SRCCOLORKEY     0x00001000      # Blit uses a source color key
RLEACCELOK      0x00002000      # Private flag
RLEACCEL        0x00004000      # Surface is RLE encoded
SRCALPHA        0x00010000      # Blit uses source alpha blending
PREALLOC        0x01000000      # Surface uses preallocated memory
```

get_pitch()

get the number of bytes used per Surface row
get_pitch() -> int

Return the number of bytes separating each row in the Surface. Surfaces in video memory are not always linearly packed. Subsurfaces will also have a larger pitch than their real width.

This value is not needed for normal Pygame usage.

get_masks()

the bitmasks needed to convert between a color and a mapped integer
get_masks() -> (R, G, B, A)

Returns the bitmasks used to isolate each color in a mapped integer.

This value is not needed for normal Pygame usage.

set_masks()

set the bitmasks needed to convert between a color and a mapped integer
set_masks((r,g,b,a)) -> None

This is not needed for normal Pygame usage. New in pygame 1.8.1

get_shifts()

the bit shifts needed to convert between a color and a mapped integer
get_shifts() -> (R, G, B, A)

Returns the pixel shifts need to convert between each color and a mapped integer.

This value is not needed for normal Pygame usage.

set_shifts()

sets the bit shifts needed to convert between a color and a mapped integer

set_shifts((r,g,b,a)) -> None

This is not needed for normal Pygame usage. New in pygame 1.8.1

get_losses()

the significant bits used to convert between a color and a mapped integer
get_losses() -> (R, G, B, A)

Return the least significant number of bits stripped from each color in a mapped integer.

This value is not needed for normal Pygame usage.

get_bounding_rect()

find the smallest rect containing data
get_bounding_rect(min_alpha = 1) -> Rect

Returns the smallest rectangular region that contains all the pixels in the surface that have an alpha value greater than or equal to the minimum alpha value.

This function will temporarily lock and unlock the Surface as needed.

New in pygame 1.8.

get_view()

return a view of a surface's pixel data.
get_view(kind='2') -> <view>

Return an object which exposes a surface's internal pixel buffer to a NumPy array. For now a custom object with an array struct interface is returned. A Python memoryview may be returned in the future. The buffer is writeable.

The kind argument is the length 1 string '2', '3', 'r', 'g', 'b', or 'a'. The letters are case insensitive; 'A' will work as well. The argument can be either a Unicode or byte (char) string. The default is '2'.

A kind '2' view is a (surface-width, surface-height) array of raw pixels. The pixels are surface bytesized unsigned integers. The pixel format is surface specific. The 3 byte unsigned integers of 24 bit surfaces are unlikely accepted by anything other than other Pygame functions.

'3' returns a (surface-width, surface-height, 3) view of `RGB` color components. Each of the red, green, and blue components are unsigned bytes. Only 24-bit and 32-bit surfaces are supported. The color components must be in either `RGB` or `BGR` order within the pixel.

'r' for red, 'g' for green, 'b' for blue, and 'a' for alpha return a (surface-width, surface-height) view of a single color component within a surface: a color plane. Color components are unsigned bytes. Both 24-bit and 32-bit surfaces support 'r', 'g', and 'b'. Only 32-bit surfaces with `SRCALPHA` support 'a'.

This method implicitly locks the Surface. The lock will be released, once the returned view object is deleted.

New in pygame 1.9.2.

get_buffer()

acquires a buffer object for the pixels of the Surface.
get_buffer() -> BufferProxy

Return a buffer object for the pixels of the Surface. The buffer can be used for direct pixel access and manipulation.

This method implicitly locks the Surface. The lock will be released, once the returned BufferProxy object is deleted.

New in pygame 1.8.

CHAPTER 33

pygame.surfarray

pygame module for accessing surface pixel data using array interfaces

Functions to convert pixel data between pygame Surfaces and arrays. This module will only be functional when pygame can use the external Numpy or Numeric packages.

Every pixel is stored as a single integer value to represent the red, green, and blue colors. The 8bit images use a value that looks into a colormap. Pixels with higher depth use a bit packing process to place three or four values into a single number.

The arrays are indexed by the `X` axis first, followed by the `Y` axis. Arrays that treat the pixels as a single integer are referred to as 2D arrays. This module can also separate the red, green, and blue color values into separate indices. These types of arrays are referred to as 3D arrays, and the last index is 0 for red, 1 for green, and 2 for blue.

Supported array systems are

```
numpy
numeric (deprecated; to be removed in Pygame 1.9.3.)
```

The default will be numpy, if installed. Otherwise, Numeric will be set as default if installed, and a deprecation warning will be issued. If neither numpy nor Numeric are installed, the module will raise an ImportError.

The array type to use can be changed at runtime using the use_arraytype () method, which requires one of the above types as string.

Note: numpy and Numeric are not completely compatible. Certain array manipulations, which work for one type, might behave differently or even completely break for the other.

Additionally, in contrast to Numeric, numpy does use unsigned 16-bit integers. Images with 16-bit data will be treated as unsigned integers. Numeric instead uses signed integers for the representation, which is important to keep in mind, if you use the module's functions and wonder about the values.

The support of numpy is new in Pygame 1.8. Official Numeric deprecation begins in Pygame 1.9.2.

`pygame.surfarray.`**`array2d`**`()`

> Copy pixels into a 2d array
>
> array2d(Surface) -> array
>
> Copy the pixels from a Surface into a 2D array. The bit depth of the surface will control the size of the integer values, and will work for any type of pixel format.
>
> This function will temporarily lock the Surface as pixels are copied (see the `Surface.lock()` - lock the Surface memory for pixel access method).

pygame.surfarray.**pixels2d**()

Reference pixels into a 2d array
pixels2d(Surface) -> array

Create a new 2D array that directly references the pixel values in a Surface. Any changes to the array will affect the pixels in the Surface. This is a fast operation since no data is copied.

Pixels from a 24-bit Surface cannot be referenced, but all other Surface bit depths can.

The Surface this references will remain locked for the lifetime of the array (see the `Surface.lock()` - lock the Surface memory for pixel access method).

pygame.surfarray.**array3d**()

Copy pixels into a 3d array
array3d(Surface) -> array

Copy the pixels from a Surface into a 3D array. The bit depth of the surface will control the size of the integer values, and will work for any type of pixel format.

This function will temporarily lock the Surface as pixels are copied (see the `Surface.lock()` - lock the Surface memory for pixel access method).

pygame.surfarray.**pixels3d**()

Reference pixels into a 3d array
pixels3d(Surface) -> array

Create a new 3D array that directly references the pixel values in a Surface. Any changes to the array will affect the pixels in the Surface. This is a fast operation since no data is copied.

This will only work on Surfaces that have 24-bit or 32-bit formats. Lower pixel formats cannot be referenced.

The Surface this references will remain locked for the lifetime of the array (see the `Surface.lock()` - lock the Surface memory for pixel access method).

pygame.surfarray.**array_alpha**()

Copy pixel alphas into a 2d array
array_alpha(Surface) -> array

Copy the pixel alpha values (degree of transparency) from a Surface into a 2D array. This will work for any type of Surface format. Surfaces without a pixel alpha will return an array with all opaque values.

This function will temporarily lock the Surface as pixels are copied (see the `Surface.lock()` - lock the Surface memory for pixel access method).

pygame.surfarray.**pixels_alpha**()

Reference pixel alphas into a 2d array
pixels_alpha(Surface) -> array

Create a new 2D array that directly references the alpha values (degree of transparency) in a Surface. Any changes to the array will affect the pixels in the Surface. This is a fast operation since no data is copied.

This can only work on 32-bit Surfaces with a per-pixel alpha value.

The Surface this array references will remain locked for the lifetime of the array.

pygame.surfarray.**pixels_red**()

Reference pixel red into a 2d array.
pixels_red (Surface) -> array

Create a new 2D array that directly references the red values in a Surface. Any changes to the array will affect the pixels in the Surface. This is a fast operation since no data is copied.

This can only work on 24-bit or 32-bit Surfaces.

The Surface this array references will remain locked for the lifetime of the array.

pygame.surfarray.**pixels_green**()

Reference pixel green into a 2d array.
pixels_green (Surface) -> array

Create a new 2D array that directly references the green values in a Surface. Any changes to the array will affect the pixels in the Surface. This is a fast operation since no data is copied.

This can only work on 24-bit or 32-bit Surfaces.

The Surface this array references will remain locked for the lifetime of the array.

pygame.surfarray.**pixels_blue**()

Reference pixel blue into a 2d array.
pixels_blue (Surface) -> array

Create a new 2D array that directly references the blue values in a Surface. Any changes to the array will affect the pixels in the Surface. This is a fast operation since no data is copied.

This can only work on 24-bit or 32-bit Surfaces.

The Surface this array references will remain locked for the lifetime of the array.

pygame.surfarray.**array_colorkey**()

Copy the colorkey values into a 2d array
array_colorkey(Surface) -> array

Create a new array with the colorkey transparency value from each pixel. If the pixel matches the colorkey it will be fully tranparent; otherwise it will be fully opaque.

This will work on any type of Surface format. If the image has no colorkey a solid opaque array will be returned.

This function will temporarily lock the Surface as pixels are copied.

pygame.surfarray.**make_surface**()

Copy an array to a new surface
make_surface(array) -> Surface

Create a new Surface that best resembles the data and format on the array. The array can be 2D or 3D with any sized integer values. Function make_surface uses the array struct interface to aquire array properties, so is not limited to just NumPy arrays. See pygame.pixelcopy.

New in Pygame 1.9.2: array struct interface support.

pygame.surfarray.**blit_array**()

Blit directly from a array values
blit_array(Surface, array) -> None

Directly copy values from an array into a Surface. This is faster than converting the array into a Surface and blitting. The array must be the same dimensions as the Surface and will completely replace all pixel values. Only integer, ascii character and record arrays are accepted.

This function will temporarily lock the Surface as the new values are copied.

pygame.surfarray.**map_array**()

Map a 3d array into a 2d array
map_array(Surface, array3d) -> array2d

Convert a 3D array into a 2D array. This will use the given Surface format to control the conversion. Palette surface formats are supported for numpy arrays.

pygame.surfarray.**use_arraytype**()

Sets the array system to be used for surface arrays
use_arraytype (arraytype) -> None

Uses the requested array type for the module functions. Currently supported array types are:

```
numeric (deprecated; will be removed in Pygame 1.9.3.)
numpy
```

If the requested type is not available, a ValueError will be raised.

New in pygame 1.8.

pygame.surfarray.**get_arraytype**()

Gets the currently active array type.
get_arraytype () -> str

Returns the currently active array type. This will be a value of the `get_arraytypes()` tuple and indicates which type of array module is used for the array creation.

New in pygame 1.8

pygame.surfarray.**get_arraytypes**()

Gets the array system types currently supported.
get_arraytypes () -> tuple

Checks, which array systems are available and returns them as a tuple of strings. The values of the tuple can be used directly in the `pygame.surfarray.use_arraytype()` () method. If no supported array system could be found, None will be returned.

New in pygame 1.8.

CHAPTER 34

pygame.tests

Pygame unit test suite package

A quick way to run the test suite package from the command line is to import the go submodule with the Python -m option:

```
python -m pygame.tests [<test options>]
```

Command line option –help displays a usage message. Available options correspond to the pygame.tests.run() arguments.

The xxxx_test submodules of the tests package are unit test suites for individual parts of Pygame. Each can also be run as a main program. This is useful if the test, such as cdrom_test, is interactive.

For Pygame development the test suite can be run from a Pygame distribution root directory. Program run_tests.py is provided for convenience, though test/go.py can be run directly.

Module level tags control which modules are included in a unit test run. Tags are assigned to a unit test module with a corresponding <<ame>_tags.py module. The tags module has the global __tags__, a list of tag names. For example, cdrom_test.py has a tag file cdrom_tags.py containing a tags list that has the 'interactive' string. The 'interactive' tag indicates cdrom_test.py expects user input. It is excluded from a run_tests.py or pygame.tests.go run. Two other tags that are excluded are 'ignore' and 'subprocess_ignore'. These two tags indicate unit tests that will not run on a particular platform, or for which no corresponding Pygame module is available. The test runner will list each excluded module along with the tag responsible.

pygame.tests.**run**()

Run the Pygame unit test suite
run(*args, **kwds) -> tuple

Positional arguments (optional):

```
The names of tests to include. If omitted then all tests are run. Test names
need not include the trailing '_test'.
```

Keyword arguments:

```
incomplete - fail incomplete tests (default False)
nosubprocess - run all test suites in the current process
               (default False, use separate subprocesses)
dump - dump failures/errors as dict ready to eval (default False)
file - if provided, the name of a file into which to dump failures/errors
timings - if provided, the number of times to run each individual test to
```

```
          get an average run time (default is run each test once)
exclude - A list of TAG names to exclude from the run
show_output - show silenced stderr/stdout on errors (default False)
all - dump all results, not just errors (default False)
randomize - randomize order of tests (default False)
seed - if provided, a seed randomizer integer
multi_thread - if provided, the number of THREADS in which to run
               subprocessed tests
time_out - if subprocess is True then the time limit in seconds before
           killing a test (default 30)
fake - if provided, the name of the fake tests package in the
       run_tests__tests subpackage to run instead of the normal
       Pygame tests
python - the path to a python executable to run subprocessed tests
         (default sys.executable)
```

Return value:

```
A tuple of total number of tests run, dictionary of error information.
The dictionary is empty if no errors were recorded.
```

By default individual test modules are run in separate subprocesses. This recreates normal Pygame usage where `pygame.init()` and `pygame.quit()` are called only once per program execution, and avoids unfortunate interactions between test modules. Also, a time limit is placed on test execution, so frozen tests are killed when there time allotment expired. Use the single process option if threading is not working properly or if tests are taking too long. It is not guaranteed that all tests will pass in single process mode.

Tests are run in a randomized order if the randomize argument is True or a seed argument is provided. If no seed integer is provided then the system time is used.

Individual test modules may have a __tags__ attribute, a list of tag strings used to selectively omit modules from a run. By default only 'interactive' modules such as cdrom_test are ignored. An interactive module must be run from the console as a Python program.

This function can only be called once per Python session. It is not reentrant.

CHAPTER 35

pygame.time

pygame module for monitoring time

Times in pygame are represented in milliseconds (1/1000 seconds). Most platforms have a limited time resolution of around 10 milliseconds. This resolution, in milliseconds, is given in the TIMER_RESOLUTION constant.

pygame.time.**get_ticks**()

get the time in milliseconds
get_ticks() -> milliseconds

Return the number of millisconds since pygame.init() was called. Before pygame is initialized this will always be 0.

pygame.time.**wait**()

pause the program for an amount of time
wait(milliseconds) -> time

Will pause for a given number of milliseconds. This function sleeps the process to share the processor with other programs. A program that waits for even a few milliseconds will consume very little processor time. It is slightly less accurate than the pygame.time.delay() function.

This returns the actual number of milliseconds used.

pygame.time.**delay**()

pause the program for an amount of time
delay(milliseconds) -> time

Will pause for a given number of milliseconds. This function will use the processor (rather than sleeping) in order to make the delay more accurate than pygame.time.wait().

This returns the actual number of milliseconds used.

pygame.time.**set_timer**()

repeatedly create an event on the event queue
set_timer(eventid, milliseconds) -> None

Set an event type to appear on the event queue every given number of milliseconds. The first event will not appear until the amount of time has passed.

Every event type can have a separate timer attached to it. It is best to use the value between `pygame.USEREVENT` and `pygame.NUMEVENTS`.

To disable the timer for an event, set the milliseconds argument to 0.

class `pygame.time.`**Clock**

create an object to help track time
Clock() -> Clock

Creates a new Clock object that can be used to track an amount of time. The clock also provides several functions to help control a game's framerate.

tick()

update the clock
tick(framerate=0) -> milliseconds
:sg:' -> '

This method should be called once per frame. It will compute how many milliseconds have passed since the previous call.

If you pass the optional framerate argument the function will delay to keep the game running slower than the given ticks per second. This can be used to help limit the runtime speed of a game. By calling `Clock.tick(40)` once per frame, the program will never run at more than 40 frames per second.

Note that this function uses SDL_Delay function which is not accurate on every platform, but does not use much cpu. Use tick_busy_loop if you want an accurate timer, and don't mind chewing cpu.

tick_busy_loop()

update the clock
tick_busy_loop(framerate=0) -> milliseconds
:sg:' -> '

This method should be called once per frame. It will compute how many milliseconds have passed since the previous call.

If you pass the optional framerate argument the function will delay to keep the game running slower than the given ticks per second. This can be used to help limit the runtime speed of a game. By calling `Clock.tick_busy_loop(40)` once per frame, the program will never run at more than 40 frames per second.

Note that this function uses `pygame.time.delay()`, which uses lots of cpu in a busy loop to make sure that timing is more acurate.

New in pygame 1.8.0.

get_time()

time used in the previous tick
get_time() -> milliseconds

Returns the parameter passed to the last call to `Clock.tick()`. It is the number of milliseconds passed between the previous two calls to `Pygame.tick()`.

get_rawtime()

actual time used in the previous tick
get_rawtime() -> milliseconds

Similar to `Clock.get_time()`, but this does not include any time used while `Clock.tick()` was delaying to limit the framerate.

get_fps()

compute the clock framerate
get_fps() -> float

Compute your game's framerate (in frames per second). It is computed by averaging the last ten calls to `Clock.tick()`.

CHAPTER 36

pygame.transform

pygame module to transform surfaces

A Surface transform is an operation that moves or resizes the pixels. All these functions take a Surface to operate on and return a new Surface with the results.

Some of the transforms are considered destructive. These means every time they are performed they lose pixel data. Common examples of this are resizing and rotating. For this reason, it is better to retransform the original surface than to keep transforming an image multiple times. (For example, suppose you are animating a bouncing spring which expands and contracts. If you applied the size changes incrementally to the previous images, you would lose detail. Instead, always begin with the original image and scale to the desired size.)

pygame.transform.**flip**()

flip vertically and horizontally

flip(Surface, xbool, ybool) -> Surface

This can flip a Surface either vertically, horizontally, or both. Flipping a Surface is nondestructive and returns a new Surface with the same dimensions.

pygame.transform.**scale**()

resize to new resolution

scale(Surface, (width, height), DestSurface = None) -> Surface

Resizes the Surface to a new resolution. This is a fast scale operation that does not sample the results.

An optional destination surface can be used, rather than have it create a new one. This is quicker if you want to repeatedly scale something. However the destination must be the same size as the (width, height) passed in. Also the destination surface must be the same format.

pygame.transform.**rotate**()

rotate an image

rotate(Surface, angle) -> Surface

Unfiltered counterclockwise rotation. The angle argument represents degrees and can be any floating point value. Negative angle amounts will rotate clockwise.

Unless rotating by 90 degree increments, the image will be padded larger to hold the new size. If the image has pixel alphas, the padded area will be transparent. Otherwise pygame will pick a color that matches the Surface colorkey or the topleft pixel value.

pygame.transform.**rotozoom**()

filtered scale and rotation

rotozoom(Surface, angle, scale) -> Surface

This is a combined scale and rotation transform. The resulting Surface will be a filtered 32-bit Surface. The scale argument is a floating point value that will be multiplied by the current resolution. The angle argument is a floating point value that represents the counterclockwise degrees to rotate. A negative rotation angle will rotate clockwise.

pygame.transform.**scale2x**()

specialized image doubler

scale2x(Surface, DestSurface = None) -> Surface

This will return a new image that is double the size of the original. It uses the AdvanceMAME Scale2X algorithm which does a 'jaggie-less' scale of bitmap graphics.

This really only has an effect on simple images with solid colors. On photographic and antialiased images it will look like a regular unfiltered scale.

An optional destination surface can be used, rather than have it create a new one. This is quicker if you want to repeatedly scale something. However the destination must be twice the size of the source surface passed in. Also the destination surface must be the same format.

pygame.transform.**smoothscale**()

scale a surface to an arbitrary size smoothly

smoothscale(Surface, (width, height), DestSurface = None) -> Surface

Uses one of two different algorithms for scaling each dimension of the input surface as required. For shrinkage, the output pixels are area averages of the colors they cover. For expansion, a bilinear filter is used. For the amd64 and i686 architectures, optimized MMX routines are included and will run much faster than other machine types. The size is a 2 number sequence for (width, height). This function only works for 24-bit or 32-bit surfaces. An exception will be thrown if the input surface bit depth is less than 24.

New in pygame 1.8

pygame.transform.**get_smoothscale_backend**()

return smoothscale filter version in use: 'GENERIC', 'MMX', or 'SSE'

get_smoothscale_backend() -> String

Shows whether or not smoothscale is using MMX or SSE acceleration. If no acceleration is available then "GENERIC" is returned. For a x86 processor the level of acceleration to use is determined at runtime.

This function is provided for Pygame testing and debugging.

pygame.transform.**set_smoothscale_backend**()

set smoothscale filter version to one of: 'GENERIC', 'MMX', or 'SSE'

set_smoothscale_backend(type) -> None

Sets smoothscale acceleration. Takes a string argument. A value of 'GENERIC' turns off acceleration. 'MMX' uses MMX instructions only. 'SSE' allows SSE extensions as well. A value error is raised if type is not recognized or not supported by the current processor.

This function is provided for Pygame testing and debugging. If smoothscale causes an invalid instruction error then it is a Pygame/SDL bug that should be reported. Use this function as a temporary fix only.

pygame.transform.**chop**()

gets a copy of an image with an interior area removed

chop(Surface, rect) -> Surface

Extracts a portion of an image. All vertical and horizontal pixels surrounding the given rectangle area are removed. The corner areas (diagonal to the rect) are then brought together. (The original image is not altered by this operation.)

NOTE: If you want a "crop" that returns the part of an image within a rect, you can blit with a rect to a new surface or copy a subsurface.

pygame.transform.**laplacian**()

find edges in a surface

laplacian(Surface, DestSurface = None) -> Surface

Finds the edges in a surface using the laplacian algorithm.

New in pygame 1.8

pygame.transform.**average_surfaces**()

find the average surface from many surfaces.

average_surfaces(Surfaces, DestSurface = None, palette_colors = 1) -> Surface

Takes a sequence of surfaces and returns a surface with average colors from each of the surfaces.

palette_colors - if true we average the colors in palette, otherwise we average the pixel values. This is useful if the surface is actually greyscale colors, and not palette colors.

Note, this function currently does not handle palette using surfaces correctly.

New in pygame 1.8 palette_colors argument new in pygame 1.9

pygame.transform.**average_color**()

finds the average color of a surface

average_color(Surface, Rect = None) -> Color

Finds the average color of a Surface or a region of a surface specified by a Rect, and returns it as a Color.

pygame.transform.**threshold**()

finds which, and how many pixels in a surface are within a threshold of a color.

threshold(DestSurface, Surface, color, threshold = (0,0,0,0), diff_color = (0,0,0,0), change_return = 1, Surface = None, inverse = False) -> num_threshold_pixels

Finds which, and how many pixels in a surface are within a threshold of a color.

It can set the destination surface where all of the pixels not within the threshold are changed to diff_color. If inverse is optionally set to True, the pixels that are within the threshold are instead changed to diff_color.

If the optional second surface is given, it is used to threshold against rather than the specified color. That is, it will find each pixel in the first Surface that is within the threshold of the pixel at the same coordinates of the second Surface.

If change_return is set to 0, it can be used to just count the number of pixels within the threshold if you set

If change_return is set to 1, the pixels set in DestSurface will be those from the color.

If change_return is set to 2, the pixels set in DestSurface will be those from the first Surface.

You can use a threshold of (r,g,b,a) where the r,g,b can have different thresholds. So you could use an r threshold of 40 and a blue threshold of 2 if you like.

New in pygame 1.8

CHAPTER 37

File Path Function Arguments

37.1 File Path Function Arguments

A Pygame function or method which takes a file path argument will accept either an Unicode or a byte—8-bit or ASCII character—string. Unicode strings are translated to Python's default file system encoding, as returned by sys.getfilesystemencoding(). An Unicode code point above U+FFFF—'uFFFF'—can be coded directly with a 32-bit escape sequences—'Uxxxxxxxx'—, even for Python interpreters built with an UCS-2 (16-bit character) unicode type. Byte strings are passed to the operating system unchanged.

Null characters—'x00'— are not permitted in the path, raising an exception. An exception is also raised if an Unicode file path cannot be encoded. How UTF-16 surrogate codes are handled is Python interpreter dependent. Use UTF-32 code points and 32-bit escape sequences instead. The exception types are function dependent.

CHAPTER 38

Documents

Readme Basic information about Pygame, what it is, who is involved, and where to find it.

Install Steps needed to compile Pygame on several platforms. Also help on finding and installing prebuilt binaries for your system.

File Path Function Arguments How Pygame handles file system paths.

LGPL License This is the license Pygame is distributed under. It provides for Pygame to be distributed with open source and commercial software. Generally, if Pygame is not changed, it can be used with any type of program.

CHAPTER 39

Tutorials

Introduction to Pygame An introduction to the basics of Pygame. This is written for users of Python and appeared in volume two of the Py magazine.

Import and Initialize The beginning steps on importing and initializing Pygame. The Pygame package is made of several modules. Some modules are not included on all platforms.

How do I move an Image? A basic tutorial that covers the concepts behind 2D computer animation. Information about drawing and clearing objects to make them appear animated.

Chimp Tutorial, Line by Line The pygame examples include a simple program with an interactive fist and a chimpanzee. This was inspired by the annoying flash banner of the early 2000's. This tutorial examines every line of coded used in the example.

`Sprite Module Introduction` Pygame includes a higher level sprite module to help organize games. The sprite module includes several classes that help manage details found in almost all games types. The Sprite classes are a bit more advanced than the regular Pygame modules, and need more understanding to be properly used.

Surfarray Introduction Pygame used the Numpy python module to allow efficient per pixel effects on images. Using the surfae arrays is an advanced feature that allows custom effects and filters. This also examines some of the simple effects from the Pygame example, arraydemo.py.

Camera Module Introduction Pygame, as of 1.9, has a camera module that allows you to capture images, watch live streams, and do some basic computer vision. This tutorial covers those use cases.

`Newbie Guide` A list of thirteen helpful tips for people to get comfortable using Pygame.

Making Games Tutorial A large tutorial that covers the bigger topics needed to create an entire game.

CHAPTER 40

Reference

genindex A list of all functions, classes, and methods in the Pygame package.

pygame.cdrom How to access and control the CD audio devices.

pygame.Color Color representation.

pygame.cursors Loading and compiling cursor images.

pygame.display Configure the display surface.

pygame.draw Drawing simple shapes like lines and ellipses to surfaces.

pygame.event Manage the incoming events from various input devices and the windowing platform.

pygame.examples Various programs demonstrating the use of individual pyame modules.

pygame.font Loading and rendering Truetype fonts.

pygame.freetype Enhanced Pygame module for loading and rendering font faces.

pygame.gfxdraw Anti-aliasing draw functions.

pygame.image Loading, saving, and transferring of surfaces.

pygame.joystick Manage the joystick devices.

pygame.key Manage the keyboard device.

pygame.locals Pygame constants.

pygame.mixer Load and play sounds

pygame.mouse Manage the mouse device and display.

pygame.movie Video playback from MPEG movies.

pygame.mixer.music Play streaming music tracks.

pygame.Overlay Access advanced video overlays.

pygame Top level functions to manage Pygame.

pygame.PixelArray Manipulate image pixel data.

pygame.Rect Flexible container for a rectangle.

pygame.scrap Native clipboard access.

pygame.sndarray Manipulate sound sample data.

pygame.sprite Higher level objects to represent game images.

pygame.Surface Objects for images and the screen.

pygame.surfarray Manipulate image pixel data.

pygame.tests Test Pygame.

pygame.time Manage timing and framerate.

pygame.transform Resize and move images.

search Search Pygame documents by keyword.

.

p