Winona LED Grid Manual

Version 1.2.5

David Sommerfield

April 18, 2024

Contents

1 Introduction

Welcome to the WiLED grid manual! You'll find the LED package provides a set of versatile and intuitive features, simplifying the process of drawing to and manipulating the LED grid no matter where you are as a programmer. This document provides an overview and explanation of the features the package provides and has extensive example code. Example projects that you can run can be found in the GitHub repository.

Many individuals helped bring this project to life and it is the culmination of these efforts that enabled me to explore this project over the past 2 years. The LED grid was initially created by Professor Zwiefelhofer of the Physics department for an art display that never came to fruition. From there it was passed onto the CS lab director Eric Wright, who had a few students take on projects for it, which are now deprecated. I was brought on board to see if I could help fix a bug, at the time the grid was only designed to display scrolling text, but coming from a background in video game development I got excited with the possibilities of this hardware. Thus began my exploration of Python and hundreds of hours of research and development into making a Python package for the LED grid. This project is currently housed in Watkins Hall at Winona State University, give it a try when you get the chance!

1.1 Getting started

To use the LED package, git clone or download the source from the GitHub. From there it must be put in the site-packages directory for Python. After that, it can be imported into a project by adding `import LED` to the start of your python file. Little prior experience is required due to the accessibility of python for those new or experienced with programming. You can find the site-packages directory by using the following command in your terminal:

```
python -m site
```

You may have to change the alias for python to python3 or your preferred version. Then just navigate to the folder under USER_SITE and put the LED library folder there.

A few libraries are needed for the LED package, you can install them with pip as shown below:

```
pip install numpy
pip install pygame
pip install pillow
```

1.2 Hello World

This is the most basic complete example of usage. It draws the words "Hello World!" in cyan to the top left corner with a 4 pixel margin.

```
from LED import *

draw_text(4,4,"Hello World!",CYAN)
draw()
```

If you want this to display to the board the same way the image does, you will need to add enable_networking(). The addition of enable_networking() will allow the pixels to be displayed on the grid.

1.3 LED Simulation

The LED package provides a simulation window that shows what is drawn to the LED grid. LED is a higher level wrapper for Pygame, which itself is based on SDL. Pygame and SDL provide most of the drawing and interfacing capabilities for the display window. It should be noted that the LED package is not just Pygame with a different presentation. It stream lines development of projects for the LED grid, by providing default pixel art fonts, and various functionalities specifically designed for the LED grid and packages them in a friendlier format than what the Pygame offers. LED significantly reduces the amount of work you have to do and makes prototyping and bug fixing much more straightforward. It is important to note that the LED package is not a fully-implemented game engine but merely a tool to display graphics on the LED grid. Using Pygame or other packages may be desirable for more complex applications, such as interfacing with OpenGL.

2 Grid Settings

The way an application is displayed on the LED grid is determined by the physical properties of the grid and there's a variety of ways these can be modified. However, WLED is designed to work out of the box; modifying these settings is entirely optional.

grid_config

Configures all the settings needed for a given LED grid.

Syntax

```
grid_config(address, width, height, orientation,
    brightness)
```

Parameters

Argument	Description
address	The IP address that pixel packets are sent to, inputted as a string
width	How many LEDs wide the board is in orientation 0
height	How many LEDs tall the board is in orientation 0
orientation	The orientation that you want LED to draw at
brightness	The value to multiply every pixel's RGB by, how bright the board is

Example

```
grid_config("199.17.162.78:7890", 60, 80, 1, 0.5)
draw_text(4,4,"Hello Winona!",(255,255,255))
```

2.1 Brightness

Depending on the lighting in the room, it may be desirable to change the brightness of the grid to avoid blinding yourself or others. The grid brightness is a multiplier for the color of every pixel drawn to the screen, so a brightness of 1 is considered full brightness, 0 is none, and 0.5 is half brightness. You can set the brightness higher than 1 and make all grid color approach white. Changing the brightness does not impact the simulated view of the program, only the way it displays on the hardware.

set_brightness

Sets a relative brightness multiplier for every pixel on the grid.

Syntax

```
set_brightness (brightness)
```

Parameters

Argument	Description
brightness	The new brightness to set the LED grid to

Example

```
if right_pressed:
    new_brightness = ( set_brightness ()+0.1) % 1
    set_brightness (new_brightness)

if left_pressed:
    new_brightness = ( set_brightness ()-0.1) % 1
    set_brightness (new_brightness)
```

The code above takes the current brightness of the grid and shifts it given user input. This new brightness then is capped between 0 and 1 and set.

get_brightness

Returns the LED grid's brightness multiplier.

Syntax

```
set_brightness ()
```

Return

Float	The brightness of the grid

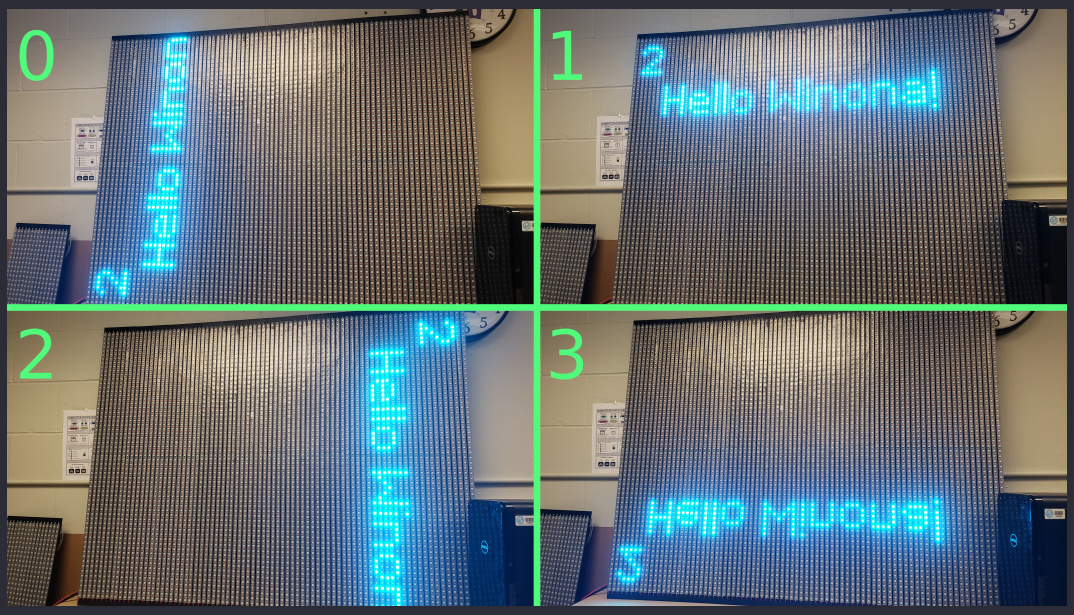
Example

```
draw_circle(get_width_adjusted()/2,get_height_adjusted()/
    2,8,(127,0,32))
center_text()
draw_text(get_width_adjusted()/2,get_height_adjusted()/2,
    "Flicker!",(255,0,0))
current_brightness =  set_brightness ()

if current_brightness != 1:
   # lerp it to 1, lerp(a, b, x) = a + (b -a ) * x;
   new_brightness = current_brightness + (1-
       current_brightness) * 0.01
    set_brightness (new_brightness)

if (random.randint(0,225) == 1):
     set_brightness (0)
```

The code above sets the brightness to 0 at random intervals, then linearly interpolates back to full brightness causing a flickering effect. This shift in brightness impacts the entire grid and therefore can be seen both on the circle and text.

2.2 Grid Orientation

The set of pixels drawn to the LED grid needs to be converted into a 1D array to account for the LEDs being addressed sequentially. By default, the surface is converted such that each row (individual LED strip) is addressed in relation to the y-coordinate of the program display. A caveat with the current hardware is that the grid was constructed to lend itself best to being used two-dimensionally in the third quadrant of the coordinate plane. However, this choice conflicts with the tradition of using the fourth quadrant in computer graphics. To account for this, WLED mirrors the image on the x-axis, which can make understanding the conversion process a bit more confusing. The grid orientation functions provided allow the grid to be used in any rotation and for easier dynamic content generation. These rotations are accomplished by changing the formula used to convert the screen to a 1D array. The grid's orientation impacts the width and height of the display, read section 2.4 to see how to work with this.



set_orientation

Sets the LED grid's orientation. It takes an integer input that is multiplied by 90 degrees. So an input of 3 will result in a three screen rotations of 90 degrees, for a final rotation of 270 degrees.

Syntax

```
set_orientation(orientation)
```

Parameters

Argument	Description
orientation	The new orientation to set the LED grid to, integer input

Example

```
# Changing orientations
draw_text(8,8,"Hello Winona!",(0,255,255))

if get_key_pressed("space"):
    set_orientation(get_orientation() + 1)
```

The above code rotates the screen 90 degrees everytime the user presses space. This is visible on the text "Hello Winona!"

get_orientation

Returns the LED grid's current orientation, which is an integer that represents how many 90 degree turns in the grid.

Syntax

```
get_orientation()
```

Return

Integer	Current orientation of the grid

Example

```
if get_orientation() == 1:
    gui_layout = 0
else:
    gui_layout = 1
```

The code above sets a variable gui_layout depending on the orientation of the screen. This allows a program to be dynamically displayed depending on the space allotted to the screen.

2.3 Server Settings

A Fadecandy server needs to be created to handle the USB communications between the host computer and the LED grid. For WLED to display to the grid, the Fadecandy server's IP address and port number need to be provided. WLED is set to run at localhost:7890 by default.

set_server

Sets the IP address and port to the server of your LED grid. This allows the LED package to send pixels to the grid.

Syntax

```
set_server(address)
```

Parameters

Argument	Description
address	The IP address that OPC packets are sent to

Example

```
import LED

LED.set_server("199.17.162.78:7890")
```

disable_networking

The LED simulator enables the development of programs for the LED grid without the physical hardware being present. However, attempting to send pixels to the LED grid without an LED grid server will result in a major performance decrease as the program fails to connect to the grid. To prevent this performance issue from occurring you can use disable_networking() to display a window without attempting to send pixels to a server. Similarly you can use enable_networking() to enable or re-enable a network connection.

Syntax

```
disable_networking()
```

enable_networking

By default networking is disabled, so if you want to display to the LED grid you will need to call this command at the start of your file.

Syntax

```
enable_networking()
```

Example

```
if right_pressed or enter_pressed:
        networked = not networked
        if networked:
            enable_networking()
        else:
            disable_networking()
```

This is code used in Tetris.py to toggle between networking in the options menu.

2.4 Frame Rate

There is a considerable difference between the frame rate of a program when it is simulation mode and when it is using networking to display to the LED grid. A given program can run much faster when it does not have to send the screen to the grid due to the latency that networking introduces. Therefore, if a project is designed to run at the same speed regardless of the latency of networking, it is important to cap the frame rate. Although many programs can run in the hundreds of frames per second, LED caps the frame rate to 120 FPS by default, and generally this will handle discrepancies between simulation and real world usage. However, the frame rate can be changed or even uncapped. Similarly, you may want to be able to run a game at multiple frame rates but have the game move at the same speed. For this reason, the LED package has delta timing functions built in.

set_fps

If you want to cap the FPS at a certain amount, thus making the game speed consistent, you can. This function sets the global target frame rate for get_delta() and caps what the game can run at.

Syntax

```
set_fps(fps)
```

Parameters

Argument	Description
fps	An integer variable of frames per second, used in other functions

get_fps

This function returns the target frames per second, set to 120 by default. This is not to be confused with the actual frame rate the game is running at.

Return

Integer	Number representing the frames per second constant

get_delta

Returns the the quotient of the time between frames and the target time between frames set by set_fps. Delta timing allows for frame rate dependent things like the speeds of objects in games to be dynamic and always appear to be the same speed, even if the frame rate is different. So if the game is running at 30 FPS, but the target FPS is set to 60, the delta value will be 2. Likewise, if the game is running at 120 FPS, the delta value will be 0.5.

Syntax

```
get_delta()
```

Return

Float	The delta value, a quotient between the game's frame rate and the target frame rate

Example

```
from LED import *
from random import choice

set_fps(choice([5,30,60,120]))

cx = get_width_adjusted()/2
cy = get_height_adjusted()/2
t = 0

while True:
    refresh()
    t += get_delta(30)
    print(f"delta: {get_delta(30)}, target_fps: {get_fps
        ()}")

    x = cos(t/10)*15
    y = sin(t/10)*15

    draw_circle(cx+x,cy+y,5,CYAN)
    draw()
```

In this example, a random fps is chosen for the game to run at, because why not? A circle is drawn to the center of the screen spinning and spins at the same speed regardless of if the game is running at 1/6th the target fps, twice the target fps, or quadruple the target fps. This is because of a feature known as delta timing.

2.5 Gride Size

The grid's default size is 60x80. Currently, every LED strip contains 60 LEDs, and there are ten panels of 8 LED strips each. However, the size of the board can be adjusted to allow for versatile usage of the current hardware and the potential to change or upgrade the hardware used. Changing the dimensions of the grid may result in unexpected graphical distortions because it changes the way pixels are sent to the grid, which is contingent on the physical number and order of the LEDs available. The LED package simplifies working with grid dimensions by providing functions for automatically adjusting to the orientation.

set_width

Sets the absolute width of the grid in orientation 0, regardless of the current orientation.

Syntax

```
set_width(width)
```

Parameters

Argument	Description
width	How many LEDs wide to configure the grid to

Example

```
grid_length = input("how many leds per strip? ")
set_width(int(grid_length))
```

The code above gets an input from the user to get the number of LEDs in a strip on the LED grid, this is equivalent to the width of the board.

set_height

Sets the absolute height of the grid in orientation 0, regardless of the current orientation.

Syntax

```
set_height(height)
```

Parameters

Argument	Description
height	How many LEDs tall to configure the grid to

Example

```
# make the grid display a perfect square
set_height(get_width())
```

The above code makes the grid space a perfect square. Unlike changing the grid width, changing the grid height does not unintentionally distort the grid.

get_width

Returns the LED grid's absolute width from orientation 0, regardless of the current orientation.

Syntax

```
get_width()
```

Return

Integer	How many LEDs are in an LED strip on the grid

get_height

Returns the LED grid's absolute height from orientation 0, regardless of the current orientation.

Syntax

```
get_height()
```

Return

Integer	How many LED strips the grid consists of

Example

```
if get_orientation() == 1:
    adjusted_height = get_width()
else:
    adjusted_height = get_height()
```

This code gives the height of the screen adjusted for the orientation of the grid.

set_width_adjusted

Sets the width of the grid adjusted for the current orientation. This means that rather than the width being strictly based on the number of LEDs in an LED strip, it can swap depending on the orientation set.

Syntax

```
set_width_adjusted(width)
```

Parameters

Argument	Description
width	How many LEDs wide to configure the grid to relative to the current orientation

set_height_adjusted

Sets the height of the grid adjusted for the current orientation.

Syntax

```
set_height_adjusted(height)
```

Parameters

Argument	Description
height	How many LEDs tall to configure the grid to relative to the current orientation

get_width_adjusted

Returns the LED grid's width relative to the current orientation. This provides functionality that eases the process of displaying of content dynamically.

Syntax

```
get_width_adjusted()
```

Return

Integer	How many LEDs wide the grid is from the current orientation

Example

```
center_text()
W = get_width_adjusted()
H = get_height_adjusted()
draw_text(W/2, H/2, "Centered :)",GREEN)
```

The code above draws text in the center of the screen and can adjust for if the screen is in landscape or portrait mode.

get_height_adjusted

Returns the LED grid's height relative to the current orientation. This provides functionality that eases the process of displaying of content dynamically.

Syntax

```
get_height_adjusted()
```

Return

Integer	How many LEDs tall the grid is from the current orientation

set_grid_size

Sets the LED grid's size regardless of orientation.

Syntax

```
set_grid_size(width,height)
```

Parameters

Argument	Description
width	How many LEDs wide to configure the grid to
height	How many LEDs tall to configure the grid to

Example

```
# Init grid
set_grid_size(60,80)
set_orientation(1)
```

The above code initializes the orientation grid size before a project.

set_grid_adjusted

Sets the LED grid's size adjusted for the current orientation.

Syntax

```
set_grid_size(width,height)
```

Parameters

Argument	Description
width	How many LEDs wide to configure the grid to relative to the orientation
height	How many LEDs tall to configure the grid to relative to the orientation

Example

```
# Init grid
set_orientation(1) # rotate grid 90 degrees
set_grid_size_adjusted(80,60)
```

The above code sets the grid orientation to a landscape mode by rotating in 90 degrees. If the hardware ever changes, the orientation may no longer be vertical by default and horizontal with a rotation be the case anymore.

3 Color

The LED package has a comprehensive selection of functions to create and manipulate colors and transparency. There are 16,777,216 ($(2^8)^3$) colors that can be represented by a computer, that is because each red, green, and blue channel are stored as a single byte (8bits). Thus in LED, colors are composed of three component parts, their red, green, and blue values represented as a byte. The RGB value of a color can be stored as either a tuple of the RGB values containing values 0 to 255 or a hexadecimal integer such as 0xFFFFFF for white.

Color Constants

Constant	Description
RED	#FF0000
ORANGE	#FF7F00
YELLOW	#FFFF00
GREEN	#00FF00
CYAN	#00FFFF
BLUE	#0000FF
PURPLE	#FF00FF
FUCHSIA	#7F00FF
BROWN	#C08060
WHITE	#FFFFFF
SILVER	#C4C4C4
GREY	#7F7F7F
BLACK	#000000
TRANSPARENT	This is the distinct lack of color, has subtractive properties and can be used like an eraser

color_hsv

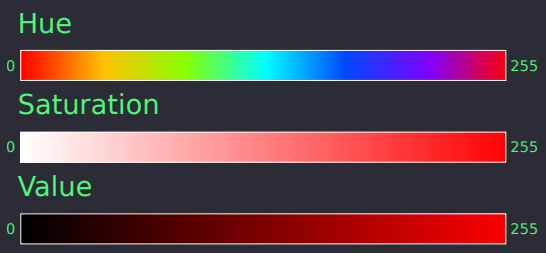
The RGB color system isn't always intuitive, and is better suited for use in technology than in art. For this reason LED has an alternative method for representing color – the HSV representation, which describes a color by its hue, saturation, and value (also sometimes called brightness). The color_hsv method creates an RGB color given a HSV representation.

Syntax

```
color_hsv(hue,saturation,value)
```

Parameters

Argument	Description
hue	Which color it is on a rainbow
saturation	The intensity of the color
value	The brightness of the color



Return

color tuple	the RGB color created from the HSV input

Example

```
from LED import *
hue = 0

while True:
    hue += 1
    rainbow = color_hsv(hue,255,255)
    draw_text(0,0,"Hello World~",rainbow)
```

The code above draws "Hello World" shifting through the colors of the rainbow.

merge_color

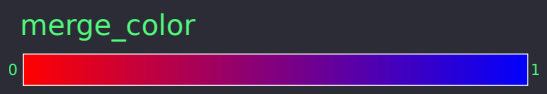
Creates a new color between two given colors, merging them on a scale from 0 to 1.

Syntax

```
merge_color(col1,col2,amount)
```

Parameters

Argument	Description
col1	A tuple of RGB values from 0 to 255, color when amount at 0
col2	A tuple of RGB values from 0 to 255, color when amount at 1
amount	A float representing the amount to interpolate between col1 and col2



Return

Color Tuple	An interpolated RGB color from the two colors input

Example

```
t = 0

while True:
    t += 0.01
    col = merge_color(RED,BLUE,sin(t)*0.5+0.5)
    set_background_color(col)
    refresh()
    draw()
```

Changes the background color from red to blue oscillating on a sine wave.

merge_palette

Syntax

```
merge_palette(colors, amount)
```

Parameters

Argument	Description
colors	A list of colors
amount	A float representing the amount to interpolate between the colors, with the first color in the list being represented by 0 and the last being 0

merge_palette

Return

Color Tuple	An interpolated RGB color from the color list

Example

```
t = 0
colors = [RED,YELLOW,BLUE]

while True:
    t += 0.01
    col = merge_palette(colors,sin(t)*0.5+0.5)
    set_background_color(col)
    refresh()
    draw()
```

Changes the background color from red to yellow to blue and backwards oscillating on a sine wave.

4 Blend Modes and Alpha

For those unfamiliar with computer graphics, blend modes and alpha can seem intimidating; however, they are easy concepts to understand. When drawing to a canvas, you can perform mathematical operations with the color values from the source and destination pixels rather than replacing pixels with new pixels. For example, an additive blend mode will take the source and destination values and add them together, resulting in a new color. Alpha is another property of pixels that is, in essence, a fancy name for transparency. An alpha of 255 will result in a completely opaque drawing, while an alpha of 0 will be transparent.

Blend Modes

Constant	Description
BM_NORMAL	This is the default blend mode, it just replaces the destination color
BM_ADD	This blend mode adds the values of the source and destination colors together ■ + ■ = ■
BM_SUBTRACT	This blend mode subtracts the source from the destination ■ - ■ = ■
BM_MAX	This selects the max value of each channel max(■, ■) = ■ #C4C4FF

set_alpha

Sets the alpha value for pixels being drawn to a canvas or the screen. Many functions in this package set an environment so to speak, so anything drawn after setting the alpha will be drawn with the new alpha. After which you can set it to a different alpha value or reset it to full opaqueness (255).

Syntax

```
set_alpha(amount)
```

Parameters

Argument	Description
amount	A value 0 to 255 representing the transparency of pixels drawn to the screen, with 0 being completely

get_alpha

Returns the alpha currently set that is used by draw functions.

Syntax

```
get_alpha()
```

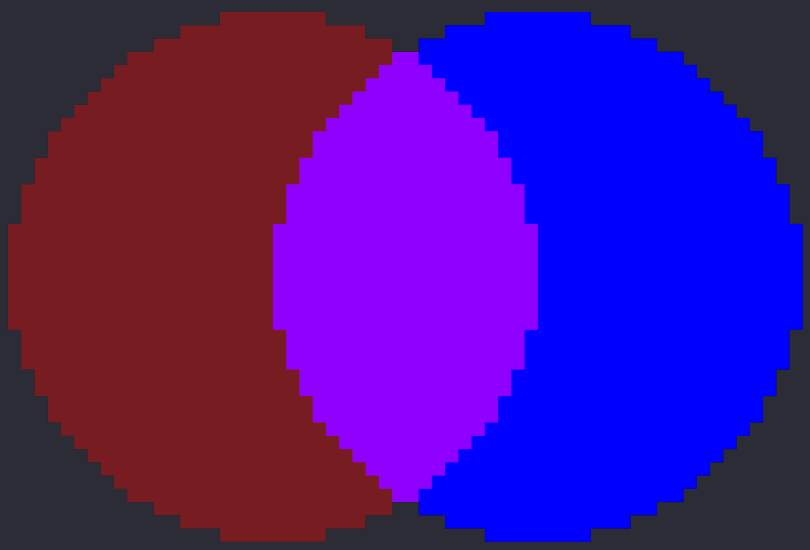
Return

Integer	A value 0 to 255 representing the current alpha value set for drawing

Example

```
cx = get_width_adjusted()/2
cy = get_height_adjusted()/2
t = 0

while True:
    refresh()

    draw_circle(cx+10,cy,20,BLUE)

    t += 0.01

    set_alpha(255*(sin(t)*0.5+0.5))
    draw_circle(cx-10,cy,20,RED)
    set_alpha(255)

    draw()
```

Draws a red circle that changes its transparency, blinking off and on smoothly, below is an image of this when the alpha is at 127.



set_blend_mode

Sets the blend mode used in draw functions for drawing pixels. This function can only take LED blend mode constants. Read section 4 to see a list of these constants and how blend modes work.

Syntax

```
set_blend_mode(blend_mode)
```

Parameters

Argument	Description
blend_mode	A blend_mode constant, such as BM_ADD

get_blend_mode

Returns the blend mode currently set that is used by draw functions.

Syntax

```
get_blend_mode()
```

Example

```
cx = get_width_adjusted()/2
cy = get_height_adjusted()/2
t = 0

while True:
    refresh()

    set_blend_mode(BM_ADD)
    t+=0.01
    for circle in range(3):
        angle = (t+2*circle*pi/3)
        col = color_hsv(circle*255/3,255,255)
        draw_circle(cx+cos(angle)*10,cy+sin(angle)*10,15,
            col)
    draw()
```

Draws three spinning circles that overlap with additive blend mode.

5 Draw Functions

Now that we got the drawing pipeline mostly out of the way, explaining how screen orientation, dimensions, and color works, we can look at the functions used for drawing.

draw

The draw function is perhaps the most important function for LED. It updates the screen with everything drawn and sends pixels to the grid if networking is enabled. Without calling this function, the screen will remain blank. This function also updates device input and delta timing and should be called every time you want to display a frame of the game.

Syntax

```
draw()
```

5.1 Background

The background color is the color used by the refresh function by default. If you want to draw background images, use the draw_image or draw_sprite functions.

set_background

Sets the refresh color for the background.

Syntax

```
set_background(color)
```

Parameters

Argument	Description
color	A tuple of RGB values from 0 to 255

get_background

Returns the color set for refreshing the background.

Syntax

```
get_background()
```

Return

Color Tuple	a color represented by a tuple of RGB values from 0 to 255

Example

```
if hit > 0:
    set_background_color(RED)
    hit -= 1
else:
    set_background_color(BLACK)
```

This code sets the background color to red when a variable hit is greater than 0 and then decrements hit. If the hit value is normal, the background goes to black. This can be used as a way of showing when a player loses health.

Example

```
if fade_to_black:
    color = get_background_color()
    set_background_color(tuple([value * 0.9 for value in
        color]))
```

This code above gets the current background color and fades it towards black with each execution of the code.

refresh

Refresh fills the entire game screen with the current background color. Without this, all things drawn to the screen stay drawn to the screen.

Syntax

```
refresh()
```

Example

```
x = 0
trail = True

while True:
    trail = not trail if x == 0 else trail
    if not trail:
        refresh()
    x = (x + 1) % get_width_adjusted()
    draw_text(x,5,"Scrollll",(0,x*3,255))
    draw()
```

The above code will draw text that pans across the screen changing colors. Every other cycle the text will leave a trail, because the canvas is not refreshing.

5.2 Basic Forms

draw_line

Draws a one pixel wide line.

Syntax

```
draw_line(x1,y1,x2,y2,color)
```

Parameters

Argument	Description
x1	x-coordinate at the start of the line
y1	y-coordinate at the start of the line
x2	x-coordinate at the end of the line
y2	y-coordinate at the end of the line
color	A tuple of RGB values from 0 to 255

Example

```
px, py = player.get_x(), player.get_y()
cx, cy = get_mouse_x(), get_mouse_y()

# makes a laser
draw_line(px,py,cx,cy,(255,0,0))
```

The code above draws a laser by taking a player's position as its first point and a cursor's position as its second, drawing a red line between it.

draw_line_width

Draws a line with a variable pixel width.

Syntax

```
draw_line(x1,y1,x2,y2,color,thickness)
```

Parameters

Argument	Description
x1	x-coordinate at the start of the line
y1	y-coordinate at the start of the line
x2	x-coordinate at the end of the line
y2	y-coordinate at the end of the line
color	A tuple of RGB values from 0 to 255
thickness	How thick the line is in pixels

Example

```
centx = get_width_adjusted()/2
centy = get_height_adjusted()/2

while True:
    refresh()
    mx = get_mouse_x()
    my = get_mouse_y()
    width = 7

    # makes a laser
    draw_line_width(centx,centy,mx,my,RED,width)
    draw_line_width(centx,centy,mx,my,WHITE,width/2)
    draw()
```

The code above draws a laser at the center of the screen to the mouse, with a red outer portion, and a white inner portion.

draw_rectangle

Draws a solid rectangle given an x and y coordinate for the top left corner and a width and height, which determine the other three points relative to that location.

Syntax

```
draw_rectangle(x,y,w,h,color)
```

Parameters

Argument	Description
x	x-coordinate of initial corner
y	y-coordinate of initial corner
w	width of the rectangle
h	height of the rectangle
color	A tuple of RGB values from 0 to 255

Example

```
# Draw menu background
grey = (96,96,96)
light_grey = (127,127,127)
draw_rectangle(10,10,get_width_adjusted()-20,
    get_height_adjusted()-20,grey))

# Draw button background
for button_num in range(0,4):
    draw_rectangle(20,20+30*button_num,get_width_adjusted
        ()-40,get_height_adjusted()-40,light_grey)
```

This code draws a menu background with a dynamic 10 pixel margin and 3 buttons on top with 10 pixel margins.

draw_circle

Draws a solid circle.

Syntax

```
draw_circle(x,y,r,color)
```

Parameters

Argument	Description
x	x-coordinate for the center of the circle
y	y-coordinate for the center of the circle
r	the radius of the circle
color	A tuple of RGB values from 0 to 255

Example

```
px, py = player.x(), player.y()
set_blend_mode(BM_ADD)
set_alpha(0.2)
draw_circle(px,py,20,(96,127,169))
draw_circle(px,py,16,(127,160,160))
draw_circle(px,py,12,(160,192,192))
```

This code draws a lighting effect around the player with concentric circles.

draw_rectangle_outline

Draws an outline of a rectangle with a specified width for its outline.

Syntax

```
draw_rectangle_outline(x,y,w,h,color,thickness)
```

Parameters

Argument	Description
x	x-coordinate of initial corner
y	y-coordinate of initial corner
w	width of the rectangle
h	height of the rectangle
color	A tuple of RGB values from 0 to 255
thickness	The thickness of the outline in pixels

draw_circle_outline

Draws an outline of a circle with a specified width.

Syntax

```
draw_circle(x,y,r,color,thickness)
```

Parameters

Argument	Description
x	x-coordinate for the center of the circle
y	y-coordinate for the center of the circle
r	the radius of the circle
color	A tuple of RGB values from 0 to 255
thickness	how many pixels wide the outline is

draw_point

Sets a single pixel on the screen to a chosen color.

Syntax

```
draw_point(x,y,color)
```

Parameters

Argument	Description
x	x-coordinate for the center of the circle
y	y-coordinate for the center of the circle
color	A tuple of RGB values from 0 to 255

Example

```
from LED import *
import random, math
    for degree in range(360):
        angle = math.radians(degree)
        x = get_width()/2+math.sin(angle)*20
        y = get_height()/2+math.cos(angle)*40
        draw_point(x,y,CYAN)
    draw()
```

The code above draws a cyan 20x40 ellipse using individual points.

This is an example of using draw_point for a Voronoi-esque visualization.

Example

```
import LED, random

points = []
colored_points = []
offset = 0
my_hue = 0
merge  = 1

W = get_width_adjusted()
H = get_height_adjusted()

for point in range(20):
    x = random.randint(0,W)
    y = random.randint(0,H)
    points.append([x,y,point*(255/20)])

for x in range(W):
    for y in range(H):
        prev_distance = 10000

        for point in points:
            distance = ((point[0]-x)**2+(point[1]-y)**2)

            if distance < prev_distance:
                prev_hue = my_hue
                merge = 0 if (prev_distance == 0) else (
                    distance/prev_distance)
                my_hue = point[2] + (prev_hue - point[2])
                     * merge
                prev_distance = distance
        colored_points.append((x,y,my_hue,merge))

while True:
    for point in colored_points:
        offset += 0.0005
        draw_point(point[0],point[1],color_hsv(point[2]+
            offset,255,255))
    draw()
```

5.3 Hypercubes

create_hypercube

This function creates a hypercube, that is a square, cube, tesseract, or a hypercube of higher dimensions.

Syntax

```
create_hypercube(dimensions)
```

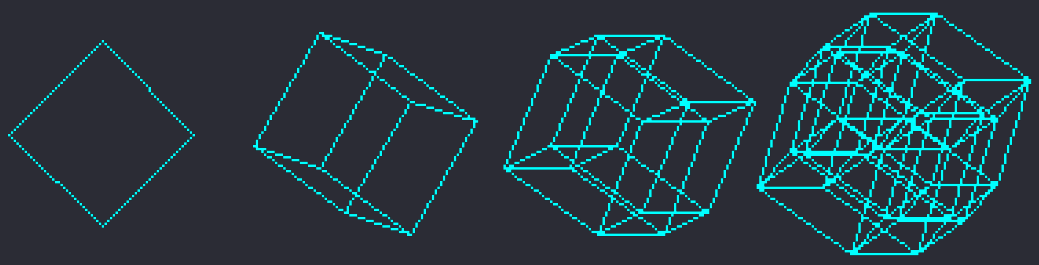
Parameters

Argument	Description
dimensions	The number of dimensions the hypercube has, 2 for a square, 3 for a cube, 4 for tesseract, and so on

Return

Hypercube	A hypercube object of a given dimension between 2 and 10

Diagram



Hypercubes, from the second to fifth dimension.

draw_hypercube

This function draws a hypercube at a given rotation, position, and scale. This function requires a hypercube object. For performant code, it is recommended that you define your hypercube before drawing it.

Syntax

```
draw_hypercube(x,y,hypercube,scale,color,rotations)
```

Parameters

Argument	Description
x	x-coordinate for the center of the hypercube
y	y-coordinate for the center of the hypercube
hypercube	a hypercube object
scale	how many pixels away from the center a given vertex can be
color	A tuple of RGB values from 0 to 255
rotations	a list of rotations for the planes associated with the hypercube.

Example

```
cube  = create_hypercube(3)

cx = get_width_adjusted()/2
cy = get_height_adjusted()/2

while True:
    refresh()
    cube[0] += 1
    cube[1] += 1
    draw_hypercube(cx,cy,cube,16,CYAN)
    draw()
```

Draws a rotating cyan cube to the middle of the screen.

6 Canvases

Canvases are spaces that can be drawn to and act as a placeholder for pixel information. The application canvas is a canvas that is sent to the grid and drawn to the LED simulator and remains the default canvas environment unless specified otherwise. All sprites and images are also canvases. You can create a canvas and draw to it just as you would the screen, but you are also able to draw it to the screen, or another canvas, at any position, rotation, transparency, or blend mode, making canvases very versatile in their usage. By default, canvases are transparent unless drawn to or filled with another color by the refresh function.

create_canvas

Creates a canvas object of a specified width and height.

Syntax

```
create_canvas(width,height)
```

Parameters

Argument	Description
width	how many pixels wide to initialize the canvas to
height	how many pixels tall to initialize the canvas to

set_canvas

Sets the canvas that draw functions operate on. By default, the canvas being drawn to is the game canvas.

Syntax

```
set_canvas(canvas)
```

Parameters

Argument	Description
canvas	the new draw canvas

Example

```
insert some code
```

reset_canvas

Resets the draw canvas to the application canvas, that is, the canvas that is sent to the screen and LED grid.

Syntax

```
reset_canvas()
```

draw_canvas

Draws a canvas at a given point on the screen.

Syntax

```
draw_canvas(x,y,canvas)
```

Parameters

Argument	Description
x	x-coordinate to draw the canvas at
y	y-coordinate to draw the canvas at
canvas	the canvas to draw

Example

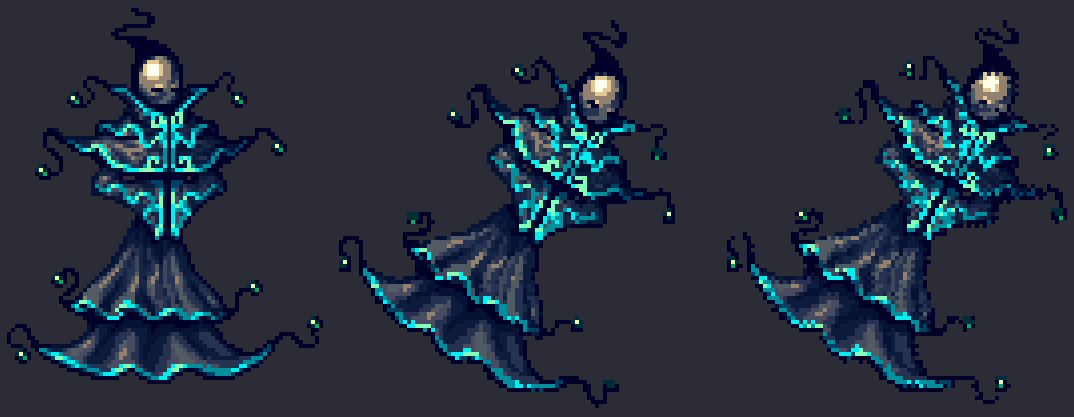
```
cx = get_width_adjusted()/2
cy = get_height_adjusted()/2
t = 0

while True:
    refresh()

    set_blend_mode(BM_ADD)
    t+=0.01
    for circle in range(3):
        angle = (t+2*circle*pi/3)
        col = color_hsv(circle*255/3,255,255)
        draw_circle(cx+cos(angle)*10,cy+sin(angle)*10,15,
            col)
    draw()
```

This example draws a red and blue circle at opposite corners of a canvas and then draws that canvas to the center of the screen. The resulting image is two overlapping quarter circles that have been masked by the canvas.

6.1 Origin and Rotation

Canvases support degree based rotation and have to have a specified point that the image rotates around, known as the origin. The origin also is the point on the canvas that is taken into consideration when calling draw_canvas and will draw the origin point where the x and y point are specified in the draw_canvas function. The origin and rotation of a canvas are by default (0,0), meaning the top left corner of the canvas.

Due to the small number of pixels the LED grid has, the LED package uses the RotSprite algorithm to handle rotation of low resolution images better. Below you can see the difference between a rotation of 30 degrees with LED RotSprite rotation (left) and with native Pygame rotation (right). RotSprite better preserves the original structure of an image and leads to less noise and broken lines.



6.2 Images

Images can be drawn three different ways, as a sprite object, a canvas, or as a string which represents a direct file path. Certain functions take all of these data types, while most functions are only designed to work with canvases and sprites. In essence, a sprite is an array of canvases, they can store 1 or more canvases as frames, and animate automatically every frame depending on their animation speed. Sprites also have an origin point considered in drawn functions and rotation.

draw_image

Draws an image given a file path. This function exists as an all in one way to draw an image, but this comes at the cost of performance and versatility. It's the quick and dirty alternative to sprites, I recommend you use sprites for anything that is going to need modularity and repeated usage.

Syntax

```
draw_image(x,y,file,w*,h*,color*,alpha*)
```

Parameters

Argument	Description
x	x-coordinate to draw the canvas at
y	y-coordinate to draw the canvas at
file	a string that points to the image file
w	an optional integer width of the image in pixels
h	an optional integer height of the image in pixels
color	an optional rgb color tuple
alpha	an optional value 0 to 255 representing the transparency of the image, with 0 being completely transparent, and 255 being opaque

Example

```
draw_image(0,0,"logo.png",get_width(),get_height(),CYAN)
```

Draws the WiLED logo stretched to the size of the screeen.

6.3 Sprites

draw_sprite

Example

```
insert some code
```

Example

```
insert some code
```

split

Takes an image, preferably a sprite sheet, and splits it up into multiple images storing it in array. This is useful for animation or different states needed of the same image.

Syntax

```
split(file_name,width,height,num_rows,num_cols,frames,
    x_margin,y_margin,x_padding,y_padding)
```

Parameters

Argument	Description
file_name	the path to the image file
width	width of each image in the sheet
height	height of each image in the sheet
num_rows	the number of rows in the sprite sheet to parse
num_cols	the number of columns in the sprite sheet to parse
frames	total number of frames needed from the sprite sheet, optional argument, defaults to number of columns times the rows
x_margin	margin on the left, optional argument
y_margin	margin on the top, optional argument
x_padding	horizontal padding between images, optional argument
y_padding	vertical padding between images, optional argument

colorize

Colorizes an image or canvas.

Syntax

```
colorize(image,color)
```

Parameters

Argument	Description
image	deceptive name here, this can be a canvas or image
color	the color to tint the image or canvas with

Example

```
col = RED if hit else WHITE
draw_text (0,0,colorize(player_sprite,col))
```

Colorizes the player if they're hit, otherwise the color of the original player sprite remains unchanged (white).

7 Text

The LED package comes with three default pixel art fonts, FNT_SMALL, FNT_NORMAL, and FNT_LARGE. The first of these was created by me, since few pixel art fonts support many of the spéciål characters I use. The small font is m3x6 font by Daniel Linssen and the large font is Cozette.

draw_text

Draws text to the screen with a given justification, position, size, font, and color.

Syntax

```
draw_text(x,y,text,color,size)
```

Parameters

Argument	Description
x	x-coordinate to draw to, where this is on the text is contingent on the text alignment
y	y-coordinate to draw to, contingent on the text alignment
text	String of text to be displayed
color	A tuple of RGB values from 0 to 255
size	The size of the text, optional

Example

```
draw_text(0,0,"Hello World!",CYAN)
```

This example draws text to the top left corner of the screen (0,0) that reads "Hello World!" in cyan.

set_font

Sets the font used in draw_text, this can either be a custom font in the local directory, or a font constant like FNT_NORMAL.

Syntax

```
set_font(font, size)
```

Parameters

Argument	Description
font	A string pointing to the font's file location relative to the local directory.
size	The size of the text, optional defaults to 16

Example

```
set_font(FNT_SMALL)
draw_text(0,0,"Hello World",CYAN)
```

Draws "Hello World" in the LED package's smaller default pixel art font.

reset_font

Resets the font back to the default FNT_NORMAL provided in the LED package.

Syntax

```
reset_font()
```

Example

```
set_font("comicsansms.ttf")
draw_text(0,0,"Hello",CYAN)
reset_font()
draw_text(0,10,"World",CYAN)
```

Draws the world hello in comic sans, followed by it in the default pixel art font.

7.1 Text Alignment

Alignment Functions

Function	Description
align_text_left	Aligns the left side of the text to the draw location
align_text_right	Aligns the right side of the text to the pixel draw location
align_text_top	Aligns the top of the text to be below the draw location
center_text_horizontal	Places the text such that the center x position of the text is aligned with the draw location
center_text_vertically	Places the text such that the center x position of the text is aligned with the draw location
center_text	Places the text such that the center x and y position of the text is the same as the draw location

reset_text

Resets the alignment for the text, with the default alignment being top and left aligned.

Syntax

```
reset_text()
```

Example

```
CX, CY = get_width_adjusted()/2, get_height_adjusted()/2
center_text_horizontal()
align_text_bottom()
draw_text(CX,CY,"Insert Title",WHITE)
align_text_top()
draw_text(CX,CY,"Here",WHITE)
```

This draws text to the center of the screen, centered horizontally. The text is split vertically into two rows by using the top and bottom alignment and drawing to the same point on the screen like the image below (draw point of the text shown in green).

8 Device Input

Allowing users to interface with a program can be done in a variety of ways, you can receive input directly from the console, or with an external device such as a keyboard, mouse, or controller. This section will cover the many functions for game input and control.

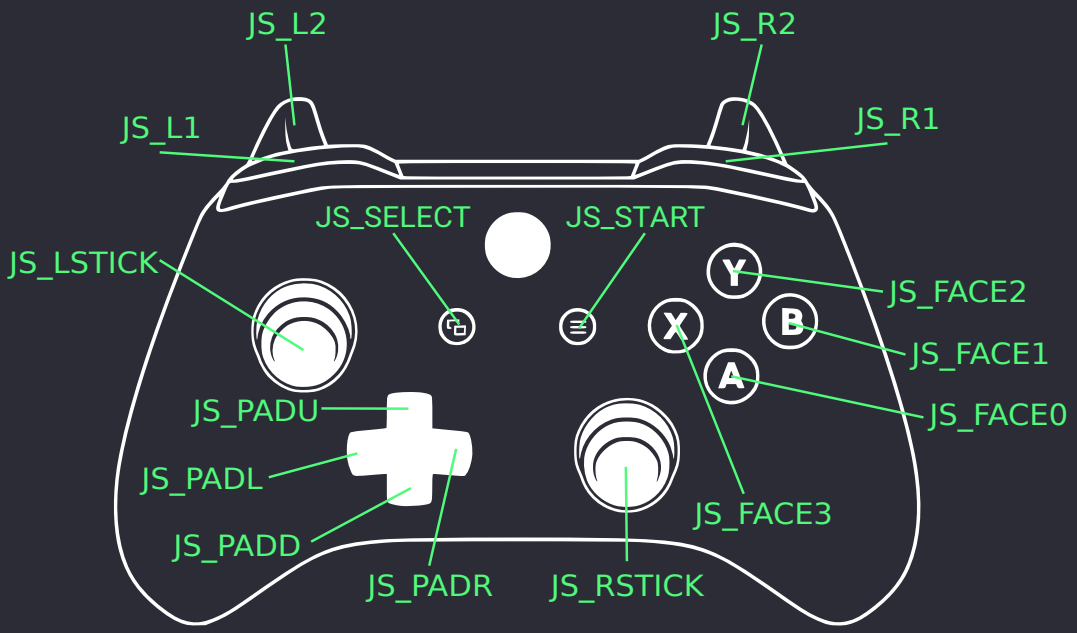
8.1 Keyboard Input

Keyboard input in the LED package is centered around strings, which can be used to identify keys on the keyboard. Single letter or character keys are referred to by their UTF8 equivalent character, so to get input from the A-key, you refer to it by "a" or "A". A key can be held, pressed, released, or inactive and the get_key functions can be used to get which of these states a given key has. Below is a list of string identifiers for other characters.

Key Aliases

String	Keyboard Description
"alt"	the right or left alt key
"left alt"	the left alt key
"right alt"	the right alt key
"control"	the right control or left control key
"left control"	the left control key
"right control"	the right control key
"enter", "return", "\n"	the enter/return/newline key
"escape", "esc"	the escape key
"f1", "f2", , "f3", . . . , "f12"	the function keys – F1, F2, etc
"shift"	the right or left shift key
"left shift", " "	the left shift key
"right shift", " "	the right shift key
"space", " "	the space key
"tab", "\t"	the tab key

8.2 Controller Input



9 Examples