tcod-camera

Release 1.0.0

Kyle Benesch

Apr 01, 2024

CONTENTS:

1 About 1

2 API reference 5

3 Glossary 9

4 Indices and tables 11

Python Module Index 13

Index 15

CHAPTER

ONE

ABOUT

This packages contains a set of tools for working with cameras which translate between world and screen coordinates.

It is intended to be used with Python-tcod and NumPy but requires neither.

```
# This library works with the idea that you have a world array you want projected onto a␣
↪screen array.
>>> import numpy as np
>>> import tcod.camera
>>> screen = np.arange(3 * 3, dtype=int).reshape(3, 3)
>>> world = np.arange(9 * 10, dtype=int).reshape(9, 10)
>>> screen
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> world
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
       [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
       [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
       [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
       [80, 81, 82, 83, 84, 85, 86, 87, 88, 89]])

# This example uses `ij` coordinates, but `xy` coordinates are also an option.
# The most basic example is to get the camera and use it to project the world and screen␣
↪shapes.
>>> camera_ij = tcod.camera.get_camera(screen.shape, center=(2, 2))  # Get the camera␣
↪position centered on (2, 2).
>>> camera_ij  # The actual camera position is always which world position to project␣
↪onto screen[0, 0].
(1, 1)
>>> screen_slice, world_slice = tcod.camera.get_slices(screen.shape, world.shape, camera_
↪ij)
>>> screen_slice
(slice(0, 3, None), slice(0, 3, None))
>>> world_slice
(slice(1, 4, None), slice(1, 4, None))
```

(continues on next page)

(continued from previous page)

```
>>> screen[screen_slice] = world[world_slice]  # Project the values of screen onto the␣
↪world.
>>> screen
array([[11, 12, 13],
       [21, 22, 23],
       [31, 32, 33]])

# Out-of-bounds camera coordinates result in partial views.
# Fully out-of-bounds cameras will result in zero element views.
>>> camera_ij = tcod.camera.get_camera(screen.shape, (4, 10))  # A camera position␣
↪beyond the right side of the world.
>>> screen_slice, world_slice = tcod.camera.get_slices(screen.shape, world.shape, camera_
↪ij)
>>> screen[screen_slice].shape  # Because this is partially out-of-bounds not all of the␣
↪screen is in view.
(3, 1)
>>> screen_slice
(slice(0, 3, None), slice(0, 1, None))
>>> world_slice
(slice(3, 6, None), slice(9, 10, None))
>>> screen[:] = -1  # The screen will be cleared with -1, this value now means out-of-
↪bounds.
>>> screen[screen_slice] = world[world_slice]  # The parts which do overlap will be␣
↪projected.
>>> screen
array([[39, -1, -1],
       [49, -1, -1],
       [59, -1, -1]])

# By adding the shape of the world to camera functions the camera can be clamped to the␣
↪bounds of the world.
# All screen indexes will be in-view as long as the screen is never larger than the␣
↪world.
>>> camera_ij = tcod.camera.clamp_camera(screen.shape, world.shape, camera_ij)
>>> screen_slice, world_slice = tcod.camera.get_slices(screen.shape, world.shape, camera_
↪ij)
>>> screen[screen_slice] = world[world_slice]
>>> screen  # The camera was moved left to fit the screen to the world.
array([[37, 38, 39],
       [47, 48, 49],
       [57, 58, 59]])

# If the world is divided into chunks then this library can be used to project each␣
↪chunk onto a single screen.
# You'll have to manage your own chunks.  Possibly in a `dict[tuple[int, int],␣
↪NDArray[Any]]`-like container.
>>> screen = np.zeros((10, 10), dtype=int)
>>> CHUNK_SIZE = (4, 4)
>>> for screen_slice, chunk_ij, chunk_slice in tcod.camera.get_chunked_slices(screen.
↪shape, CHUNK_SIZE, camera=(0, 0)):
...     screen[screen_slice] = chunk_ij[0] + chunk_ij[1] * 10
...     print(f"{screen_slice=}, {chunk_ij=}, {chunk_slice=}")
```

(continues on next page)

(continued from previous page)

```
screen_slice=(slice(0, 4, None), slice(0, 4, None)), chunk_ij=(0, 0), chunk_
↪slice=(slice(0, 4, None), slice(0, 4, None))
screen_slice=(slice(0, 4, None), slice(4, 8, None)), chunk_ij=(0, 1), chunk_
↪slice=(slice(0, 4, None), slice(0, 4, None))
screen_slice=(slice(0, 4, None), slice(8, 10, None)), chunk_ij=(0, 2), chunk_
↪slice=(slice(0, 4, None), slice(0, 2, None))
screen_slice=(slice(4, 8, None), slice(0, 4, None)), chunk_ij=(1, 0), chunk_
↪slice=(slice(0, 4, None), slice(0, 4, None))
screen_slice=(slice(4, 8, None), slice(4, 8, None)), chunk_ij=(1, 1), chunk_
↪slice=(slice(0, 4, None), slice(0, 4, None))
screen_slice=(slice(4, 8, None), slice(8, 10, None)), chunk_ij=(1, 2), chunk_
↪slice=(slice(0, 4, None), slice(0, 2, None))
screen_slice=(slice(8, 10, None), slice(0, 4, None)), chunk_ij=(2, 0), chunk_
↪slice=(slice(0, 2, None), slice(0, 4, None))
screen_slice=(slice(8, 10, None), slice(4, 8, None)), chunk_ij=(2, 1), chunk_
↪slice=(slice(0, 2, None), slice(0, 4, None))
screen_slice=(slice(8, 10, None), slice(8, 10, None)), chunk_ij=(2, 2), chunk_
↪slice=(slice(0, 2, None), slice(0, 2, None))
>>> screen
array([[ 0,  0,  0,  0, 10, 10, 10, 10, 20, 20],
       [ 0,  0,  0,  0, 10, 10, 10, 10, 20, 20],
       [ 0,  0,  0,  0, 10, 10, 10, 10, 20, 20],
       [ 0,  0,  0,  0, 10, 10, 10, 10, 20, 20],
       [ 1,  1,  1,  1, 11, 11, 11, 11, 21, 21],
       [ 1,  1,  1,  1, 11, 11, 11, 11, 21, 21],
       [ 1,  1,  1,  1, 11, 11, 11, 11, 21, 21],
       [ 1,  1,  1,  1, 11, 11, 11, 11, 21, 21],
       [ 2,  2,  2,  2, 12, 12, 12, 12, 22, 22],
       [ 2,  2,  2,  2, 12, 12, 12, 12, 22, 22]])
```

CHAPTER

TWO

API REFERENCE

Camera helper tools for 2D tile-based projects.

tcod.camera.**clamp_camera**(*screen*, *world*, *camera*, *justify=0.5*)

Clamp the camera to the screen/world shapes. Preventing the camera from leaving the world boundary.

Parameters

- **screen** (*tuple[int, ...]*) – The *screen* shape.
- **world** (*tuple[int, ...]*) – The *world* shape.
- **camera** (*tuple[int, ...]*) – The current *camera* position.
- **justify** (*float | tuple[float, ...]*) – The justification to use when the world is smaller than the screen. Defaults to `0.5` which will center the world when it is smaller than the screen.

  A value of zero will move a world smaller to the screen to inner corner. One would do the same but to the opposite corner. You may also give a tuple with a value for each axis.

Returns

The new *camera* position clamped using the given shapes and justification rules.

Like the other functions, this camera position still assumes that the screen offset is `(0, 0)`. This means that no other code changes are necessary to add or remove this clamping effect. This also means that changing `justify` also requires no external changes.

Return type
tuple[int, ...]

tcod.camera.**get_camera**(*screen*, *center*, *clamping=None*)

Return the translation position for the camera from the given center position, screen size, and clamping rule.

Parameters

- **screen** (*tuple[int, ...]*) – The *screen* shape.
- **center** (*tuple[int, ...]*) – The *world* position which the camera will center on.
- **clamping** (*tuple[tuple[int, ...], float | tuple[float, ...]] | None*) – The clamping rules, this is `(world, justify)` as if provided to *clamp_camera*. If clamping is *None* then this function only does the minimum of subtracting half the screen size to get the camera position.

  world is the world shape. *justify* can be *(0.5, 0.5)* to center the world when it's smaller than the camera, or *(0, 0)* to place the world towards zero. This would be the upper-left corner with libtcod.

Returns
The clamped *camera* position.

Return type
tuple[int, ...]

tcod.camera.**get_chunked_slices**(*screen*, *chunk_shape*, *camera*)

Iterate over map chunks covered by the screen.

Parameters

- **screen** (*tuple[int, ...]*) – The shape of the *screen*.
- **chunk_shape** (*tuple[int, ...]*) – The shape of individual chunks.
- **camera** (*tuple[int, ...]*) – The *camera* position.

Yields
`(screen_slice, chunk_index, chunk_slice)`

For the chunk at *chunk_index* it should be sliced with *chunk_slice* to match a screen sliced with *screen_slice*.

Return type
Iterator[tuple[tuple[slice, ...], tuple[int, ...], tuple[slice, ...]]]

Example:

```
CHUNK_SIZE: tuple[int, int]
screen: NDarray  # Screen array.
chunks: dict[tuple[int, int], NDarray]  # Mapping of chunked arrays.
camera: tuple[int, int]
for screen_slice, chunk_ij, chunk_slice in tcod.camera.get_chunked_slices(screen.
↪shape, CHUNK_SIZE, camera):
    if chunk_ij in chunks:
        screen[screen_slice] = chunks[chunk_ij][chunk_slice]
```

```
>>> list(get_chunked_slices((10,10),(10,10),(0,0)))
[((slice(0, 10, None), slice(0, 10, None)), (0, 0), (slice(0, 10, None), slice(0,␣
↪10, None)))]
>>> list(get_chunked_slices((10,10),(10,10),(-5,-5)))
[((slice(0, 5, None), slice(0, 5, None)), (-1, -1), (slice(5, 10, None), slice(5,␣
↪10, None))), ((slice(0, 5, None), slice(5, 10, None)), (-1, 0), (slice(5, 10,␣
↪None), slice(0, 5, None))), ((slice(5, 10, None), slice(0, 5, None)), (0, -1),␣
↪(slice(0, 5, None), slice(5, 10, None))), ((slice(5, 10, None), slice(5, 10,␣
↪None)), (0, 0), (slice(0, 5, None), slice(0, 5, None)))]
```

tcod.camera.**get_slices**(*screen*, *world*, *camera*)

Return (screen_slices, world_slices) for the given parameters.

This function takes any number of dimensions. The *screen*, *world*, and *camera* tuples must be the same length.

Parameters

- **screen** (*tuple[int, ...]*) – The *screen* shape.
- **world** (*tuple[int, ...]*) – The *world* shape.
- **camera** (*tuple[int, ...]*) – The *camera* position.

Returns

The (screen_slice, world_slice) slices which can be used to index arrays of the given shapes.

Arrays indexed with these slices will always result in the same shape. The slices will be narrower than the screen when the camera is partially out-of-bounds. The slices will be zero-width if the camera is entirely out-of-bounds.

Return type
tuple[tuple[slice, ...], tuple[slice, ...]]

Example:

```
console: tcod.console.Console  # Libtcod console, C order.
player_ij: tuple[int, int]  # Player (y, x) position.
world: NDArray[Any]  # Array created with `dtype=tcod.console.rgb_graphic`, C order.

console.clear()  # Clear the console in case any areas are not covered by tcod.
↪camera.get_slices.

# Get the camera position centered on the player.
camera_ij = tcod.camera.get_camera(console.rgb.shape, player_ij)

# Get the screen/world slices at the camera position.
screen_slice, world_slice = tcod.camera.get_slices(console.rgb, world, camera_ij)
console.rgb[screen_slice] = world[world_slice]  # Render world graphics.

# Render the player.
player_screen_y, player_screen_x = player_ij[0] - camera_ij[0], player_ij[1] -␣
↪camera_ij[1]
console.print(player_screen_x, player_screen_y, "@")
```

tcod.camera.**get_views**(*screen*, *world*, *camera*)

Return (screen_view, world_view) for the given parameters.

This function takes any number of dimensions. The *screen*, *world*, and *camera* tuples must be the same length.

Parameters

- **screen** (*_ScreenArray*) – The NumPy array for the *screen*.
- **world** (*_WorldArray*) – The NumPy array for the *world*.
- **camera** (*tuple[int, ...]*) – The *camera* position.

Returns

The given arrays pre-sliced into (screen_view, world_view) views.

These will always be the same shape. They will be sliced into a zero-width views once the camera is far enough out-of-bounds.

Convenient when you only have one screen and one world array to work with, otherwise you should call *get_slices* instead.

Return type
tuple[*_ScreenArray*, *_WorldArray*]

CHAPTER

THREE

GLOSSARY

Camera
: This is a vector used to translate between *screen*-space and *world*-space coordinates.

  The camera is positioned in world-space. *Screen* position `(0, 0)` is projected onto the *world* position at where the camera is placed. This is the upper-left corner of the screen in libtcod or SDL.

  Once you get the camera position via *`get_camera`* or by manually placing it you can convert between screen coordinates and world coordinates by applying vector math. This position is also used by *`get_slices`*, *`get_views`*, or *`get_chunked_slices`*.

  Add the camera position to a screen position (such as a mouse tile position) to get the world position (such as where in the world itself the mouse is hovering over.) Subtract the camera position from a world position (such as a player object position) to get the screen position (such as where to draw the player on the screen.)

  See the RogueBasin article on Scrolling maps for more details and a visual example.

Screen
: Screen-space is the array which is projected into the *world* using a *camera*.

  Normally this array is something like a tcod console, such as `tcod.console.Console.rgb`. However, this can be any temporary array projected into the world.

World
: This is the map data which is stored normally as one or more arrays.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

tcod.camera, 5

INDEX

C

Camera, **9**
clamp_camera() (*in module tcod.camera*), 5

G

get_camera() (*in module tcod.camera*), 5
get_chunked_slices() (*in module tcod.camera*), 6
get_slices() (*in module tcod.camera*), 6
get_views() (*in module tcod.camera*), 7

M

module
    tcod.camera, 5

S

Screen, **9**

T

tcod.camera
    module, 5

W

World, **9**