tinytex

Release 0.2.0 a

Sam Izdat

May 15, 2025

HOW TO:

1 Getting started **3**

2 About **5**

3 How to **7**
- 3.1 Resample images . . . 7
- 3.2 Create texture atlases . . . 9
- 3.3 Work with surface geometry . . . 10
- 3.4 Use smoothstep . . . 11
  - 3.4.1 Stateful usage . . . 11
  - 3.4.2 Stateless (one-off) usage . . . 11
  - 3.4.3 Normalize inputs manually . . . 11
- 3.5 Make noise . . . 11
  - 3.5.1 Spatial-domain noise . . . 11
  - 3.5.2 Spectral-domain noise . . . 12
- 3.6 Create SDFs . . . 12
  - 3.6.1 Generating basic shapes . . . 12
  - 3.6.2 Converting images to SDF . . . 13
  - 3.6.3 Combining fields . . . 13
- 3.7 Work with tiles . . . 13
  - 3.7.1 Splitting and merging . . . 13
  - 3.7.2 Tile indexing helpers . . . 14
  - 3.7.3 Blending tiles . . . 14

4 Reference **15**
- 4.1 API Reference . . . 15
  - 4.1.1 rand module . . . 35
  - 4.1.2 ti module . . . 37
- 4.2 Release notes . . . 37
- 4.3 License . . . 38

5 Links **39**

6 Sibling projects **41**

Python Module Index **43**

Index **45**

Python texture sampling, processing and synthesis library for PyTorch-involved projects.

This library is a hodgepodge of tangentially-related procedures useful for sampling, creating and modifying various kinds of textures. This is primarily intended for batched or unbatched PyTorch image tensors. This library provides:

- image resampling/rescaling, cropping and padding
- tiling
  - split images into tiles
  - merge tiles back into images
  - seamlessly stitch textures with color or vector data for mutual tiling or self-tiling
- texture atlases
  - pack images into texture atlases
  - sample images from texture atlases
  - generate tiling masks from texture atlases
- computing and rendering 2D signed distance fields
- computing and approximating surface geometry
  - normals to height
  - height to normals
  - height/normals to curvature
- approximating ambient occlusion and bent normals
- blending multiple normal maps
- pseudo-random number generation
- generating tiling spatial-domain noise
- generating spectral-domain noise
- warping image coordinates
- transforming 1D and 2D images to and from Haar wavelet coefficients
- (experimental) backend-agnostic 1D/2D/3D textures for Taichi (if installed with Taichi optional dependency)
  - load from and save to the filesystem
  - convert textures to and from PyTorch tensors
  - sample textures with lower or higher-order interpolation/approximation

CHAPTER

ONE

GETTING STARTED

- Run `pip install tinytex`
- Run `ttex-setup`

CHAPTER

TWO

ABOUT

License

MIT License on all original code - see source for details

CHAPTER

THREE

HOW TO

3.1 Resample images

The *Resampling* class works on PyTorch tensors of shape [C, H, W] or [N, C, H, W].

Tile an image to a target size:

```
from tinytex import Resampling, fsio
import torch

img = fsio.load_image("input.png")      # shape [C, H, W]
tiled = Resampling.tile(img, (512, 512))
fsio.save_image(tiled, "tiled.png")
```

Tile by a fixed repeat count:

```
repeated = Resampling.tile_n(img, repeat_h=4, repeat_w=2)
fsio.save_image(repeated, "repeated.png")
```

Make a square by tiling:

```
square = Resampling.tile_to_square(img, target_size=256)
fsio.save_image(square, "square.png")
```

Crop to a box:

```
cropped = Resampling.crop(img, shape=(100, 150), start=(10, 20))
fsio.save_image(cropped, "cropped.png")
```

Resize to exact dimensions:

```
# simple bilinear down or up-sampling
resized = Resampling.resize(img, (200, 300), mode="bilinear")
fsio.save_image(resized, "resized.png")
```

Iterative downsample:

```
small = Resampling.resize(img, (64, 64), mode="bilinear", iterative_downsample=True)
fsio.save_image(small, "small.png")
```

Area downsample:

```
small = Resampling.resize(img, (64, 64), mode="area", iterative_downsample=False)
fsio.save_image(small, "small.png")
```

Aspect-preserving shortest-edge resize:

```
se = Resampling.resize_se(img, size=128, mode="bicubic")
fsio.save_image(se, "short_edge.png")
```

Aspect-preserving longest-edge resize:

```
le = Resampling.resize_le(img, size=256, mode="bicubic")
fsio.save_image(le, "long_edge.png")
```

Resize longest edge to next power-of-two:

```
pot = Resampling.resize_le_to_next_pot(img, mode="bicubic")
fsio.save_image(pot, "pot.png")
```

Pad right & bottom:

```
padded = Resampling.pad_rb(img, shape=(300,300), mode="replicate")
fsio.save_image(padded, "padded.png")
```

Pad up to square power-of-two:

```
square_pot = Resampling.pad_to_next_pot(img, mode="replicate")
fsio.save_image(square_pot, "square_pot.png")
```

Generate a mip pyramid tensor:

```
# builds tensor sized [C, H, W + W//2]
pyramid = Resampling.generate_mip_pyramid(img)
# you can inspect each level with compute_lod_offsets:
offsets = Resampling.compute_lod_offsets(img.size(1))
```

Sample a specific LOD with bilinear filtering:

```
# lod = 0 (base), 1, 2... can be fractional
out0 = Resampling.sample_lod_bilinear(pyramid, 128, 128, lod=1.5)
fsio.save_image(out0, "lod_bilinear.png")
```

Sample a specific LOD with 4-tap B-spline:

```
out1 = Resampling.sample_lod_bspline_hybrid(pyramid, 128, 128, lod=2.3)
fsio.save_image(out1, "lod_bspline.png")
```

Sample a specific LOD with dithered B-spline:

```
out2 = Resampling.sample_lod_bspline_dither(pyramid, 128, 128, lod=2.3)
fsio.save_image(out2, "lod_dither.png")
```

See: *Resampling*

3.2 Create texture atlases

The *Atlas* class packs multiple image tensors into a single texture atlas.

```
from tinycio import fsio
from tinytex import Atlas
```

Load and pack all images from a directory:

```
atlas = Atlas.from_dir(
    path='assets/',
    ext='.png',
    channels=3,
    allow_mismatch=True,
    max_h=1024,
    max_w=1024,
    crop=True,
    row=False,
    sort='height'
)
```

Save the packed atlas:

```
fsio.save_image(atlas.atlas, 'atlas.png')
```

Sample a texture by name or index:

```
tex = atlas.sample('stone')
tex2 = atlas.sample(0)

fsio.save_image(tex, 'stone_out.png')
```

Sample randomly:

```
rand_tex = atlas.sample_random()
fsio.save_image(rand_tex, 'rand.png')
```

Generate a tiling mask using randomly overlaid textures:

```
mask = atlas.generate_mask(
    shape=(512, 512),
    scale=0.5,
    samples=3
)

fsio.save_image(mask, 'mask.png')
```

Manual construction is also possible:

```
im1 = fsio.load_image('a.png')
im2 = fsio.load_image('b.png')

atlas = Atlas(min_size=256, max_size=2048, force_square=True)
atlas.add('a', im1)
```

(continues on next page)

(continued from previous page)

```
atlas.add('b', im2)
atlas.pack(crop=True)

fsio.save_image(atlas.atlas, 'manual.png')
```

To inspect bounds:

```
print(atlas.index['a'])  # -> (x0, y0, x1, y1)
```

The packing is either rectangular (default) or row-based (*row=True*), depending on your layout needs. Use rectangular unless everything's the same height.

See: *Atlas*

3.3 Work with surface geometry

The *SurfaceOps* class provides tools for converting and processing normal maps, height maps, curvature, and ambient occlusion in a y-up OpenGL-style tangent space.

Convert normal maps to angle maps:

```
from tinytex import SurfaceOps, fsio

normal_map = fsio.load_image("normal.png")
angle_map = SurfaceOps.normals_to_angles(normal_map, normalize=True, rescaled=True)
fsio.save_image(angle_map, "angles.png")
```

Convert angles back to normals:

```
normals = SurfaceOps.angles_to_normals(angle_map, normalize=True, rescaled=True)
fsio.save_image(normals, "reconstructed_normals.png")
```

Reorient/blend detail normals onto base:

```
detail = fsio.load_image("detail_normals.png")
blended = SurfaceOps.blend_normals(normal_map, detail, rescaled=True)
fsio.save_image(blended, "blended.png")
```

Generate normals from height:

```
height = fsio.load_image("height.png")
normals = SurfaceOps.height_to_normals(height, rescaled=True)
fsio.save_image(normals, "generated_normals.png")
```

Generate height from normals:

```
height, scale = SurfaceOps.normals_to_height(normal_map, self_tiling=True, rescaled=True)
fsio.save_image(height, "reconstructed_height.png")
```

Estimate curvature from height:

```
curvature, cavities, peaks = SurfaceOps.height_to_curvature(height)
fsio.save_image(curvature, "curvature.png")
```

Compute screen space AO and bent normals:

```
ao, bent = SurfaceOps.compute_occlusion(height_map=height, normal_map=normals,␣
↪rescaled=True)
fsio.save_image(ao, "ao.png")
fsio.save_image(bent, "bent_normals.png")
```

Recompute z channel for tangent-space normals:

```
fixed = SurfaceOps.recompute_z(normal_map, rescaled=True)
fsio.save_image(fixed, "fixed_normals.png")
```

See: *SurfaceOps*

3.4 Use smoothstep

3.4.1 Stateful usage

Create a reusable interpolator instance:

```
from tinytex import Smoothstep

# cubic smoothstep
s = Smoothstep('cubic_polynomial')
y = s.forward(0.0, 1.0, x)
x_back = s.inverse(y)

# rational smoothstep of order 4
r = Smoothstep('rational', n=4)
y2 = r.forward(0.0, 1.0, x)
```

3.4.2 Stateless (one-off) usage

Call directly on the class:

```
from tinytex import Smoothstep

y = Smoothstep.apply('quintic_polynomial', 0.0, 1.0, x)
y3 = Smoothstep.apply('rational', 0.0, 1.0, x, n=5)
```

3.4.3 Normalize inputs manually

If you need to clamp or remap before interpolation:

```
from tinytex import Smoothstep

x_norm = Smoothstep._normalize(x, edge0, edge1)
```

See: *Smoothstep*

3.5 Make noise

3.5.1 Spatial-domain noise

Make Perlin noise:

```
from tinytex import SpatialNoise

img = SpatialNoise.perlin((256, 256), density=8.)
```

Make noise from fractal layers:

```
img = SpatialNoise.fractal((256, 256), density=4., octaves=6)
```

Make turbulent noise:

```
img = SpatialNoise.turbulence((256, 256), density=6., ridge=True)
```

Make Worley noise:

```
img = SpatialNoise.worley((256, 256), density=10.)
```

3.5.2 Spectral-domain noise

Generate flat-spectrum white noise:

```
from tinytex import SpectralNoise

img = SpectralNoise.white(256, 256)
```

Generate other types of colored noise

```
img = SpectralNoise.pink(256, 256)
img = SpectralNoise.brownian(256, 256)
img = SpectralNoise.blue(256, 256)
img = SpectralNoise.violet(256, 256)
```

Define your own power spectrum for custom spectral shaping:

```
import torch
from tinytex import SpectralNoise

def bandpass(f):
    return torch.exp(-((f - 0.2)**2) / 0.01)

img = SpectralNoise.noise_psd_2d(256, 256, psd=bandpass)
```

See: *SpatialNoise*, *SpectralNoise*

3.6 Create SDFs

The *SDF* class provides signed distance field computation, primitive shape generation, and rendering. This is all 'borrowed' from "2D SDF functions" by Inigo Quilez.

3.6.1 Generating basic shapes

Create a soft circular mask:

```
from tinytex import SDF

sdf = SDF.circle(size=128, radius=40)
img = SDF.render(sdf, shape=(128, 128), edge0=0.4, edge1=0.6)
```

Create a rectangular SDF:

```
sdf = SDF.box(size=128, box_shape=(64, 32))
img = SDF.render(sdf, shape=(128, 128), edge0=0.45, edge1=0.5)
```

Create a segment-shaped SDF:

```
sdf = SDF.segment(size=128, a=(32, 32), b=(96, 96))
img = SDF.render(sdf, shape=(128, 128))
```

3.6.2 Converting images to SDF

Convert a binary mask into a normalized SDF:

```
binary_mask = torch.rand(1, 128, 128) > 0.5
sdf = SDF.compute(binary_mask.float(), threshold=0.5)
img = SDF.render(sdf, shape=(128, 128))
```

Tiling for seamless textures:

```
sdf = SDF.circle(size=64, radius=20, tile_to=128)
img = SDF.render(sdf, shape=(128, 128))
```

3.6.3 Combining fields

You can blend shapes via min/max:

```
a = SDF.circle(64, radius=20)
b = SDF.box(64, box_shape=(32, 32))
union = SDF.min(a, b)
intersect = SDF.max(a, b)
img = SDF.render(union, shape=(64, 64))
```

See: *SDF*

3.7 Work with tiles

The *Tiling* class handles image tiling, seamless merging, and Poisson-based tile blending. This is useful for working with large textures or tiled procedural data.

3.7.1 Splitting and merging

Split a large image into smaller tiles:

```
from tinytex import Tiling

tiles, rows, cols = Tiling.split(image, shape=(64, 64))
```

You can merge the tiles back:

```
merged = Tiling.merge(tiles, rows, cols)
```

Tiles are ordered row-major (left-to-right, top-to-bottom):

```
| 0 | 1 | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |
```

3.7.2 Tile indexing helpers

Use these if you're doing custom traversal or adjacency logic:

```
row, col = Tiling.get_tile_position(idx, cols)
idx = Tiling.get_tile_index(row, col, cols)
neighbors = Tiling.get_tile_neighbors(row, col, rows, cols, wrap=True)
```

3.7.3 Blending tiles

Blend adjacent tiles to remove seams between them:

```
blended = Tiling.blend(tiles, rows=rows, cols=cols)
```

This uses a Poisson solver to match gradients across tile edges. It's expensive but high quality.

For self-tiling output (no visible borders when tiled):

```
seamless = Tiling.blend(tiles, rows, cols, wrap=True)
```

If your tiles represent **vector data** (e.g., normal maps), pass *vector_data=True* to convert vectors into angles during blending:

```
seamless = Tiling.blend(normal_tiles, rows, cols, vector_data=True)
```

Note

This is a heavy operation. By default it uses SciPy's solver, but will auto-switch to PyAMG or AMGCL if available.

See: *Tiling*

CHAPTER

FOUR

REFERENCE

4.1 API Reference

Resampling()	Image resizing and padding.
Atlas([min_size, max_size, force_square])	Texture atlas packing and sampling.
SurfaceOps()	Geometric surface processing.
Smoothstep(interpolant[, n])	Smoothstep interpolation.
SpectralNoise()	Spectral-domain procedural noise generators.
SpatialNoise()	Spatial-domain procedural noise generators.
Voronoi()	Voronoi-based generators.
SDF()	Signed distance field computation and rendering.
Tiling()	Tiling and tile blending.
Wavelet()	Wavelet transforms and coefficient culling.
Warping()	Warping and coordinate system translation.

class `tinytex.`**Resampling**

Bases: `object`

Image resizing and padding.

classmethod compute_lod_offsets(*height*)

Compute the vertical offsets for each mip level in a mip pyramid.

Parameters

height (*int*) – Height of the base image, determines the number of mip levels and their vertical offsets.

Returns

List of vertical offsets (in pixels) for each mip level. The base level is not included.

Return type

List[int]

classmethod crop(*im*, *shape*, *start=(0, 0)*)

Crop image tensor to maximum target shape, if and only if a crop box target dimension is smaller than the boxed image dimension. Returned tensor can be smaller than target shape, depending on input image shape - i.e. no automatic padding.

Parameters

- **im** (*torch.Tensor*) – Image tensor sized [C, H, W] or [N, C, H, W].
- **shape** (*tuple*) – Target shape as (height, width) tuple.

- **start** (*tuple*) – Top-left corner coordinates of the crop box as (top, left) tuple.

Returns
Cropped image tensor sized [C, H, W] or [N, C, H, W].

`classmethod generate_mip_pyramid(im)`

Generate a mipmap pyramid from input image and store it in a single image tensor. Pyramid is stored by placing the base image in the left portion and stacking each downsampled mip level vertically in the right portion.

Parameters
im (*torch.Tensor*) – Input image tensor shape [C, H, W]

Returns
Pyramid tensor of shape [C, H, W + W//2]

Return type
torch.Tensor

`classmethod pad_rb(im, shape, mode='replicate')`

Pad image tensor to the right and bottom to target shape.

Parameters

- **im** (*torch.Tensor*) – Image tensor sized [C, H, W] or [N, C, H, W].
- **mode** (*str*) – Padding algorithm ('constant' | 'reflect' | 'replicate' | 'circular').
- **shape** (*tuple*)

Returns
Padded image tensor sized [C, H, W] or [N, C, H, W].

Return type
torch.Tensor

`classmethod pad_to_next_pot(im, mode='replicate')`

Pad image tensor to next highest power-of-two square dimensions.

Parameters

- **im** (*torch.Tensor*) – image tensor sized [C, H, W] or [N, C, H, W]
- **mode** (*str*) – padding algorithm ('constant' | 'reflect' | 'replicate' | 'circular')

Returns
padded image tensor sized [C, H, W] or [N, C, H, W] where H = W

Return type
torch.Tensor

`classmethod resize(im, shape, mode='bilinear', iterative_downsample=False)`

Resize image tensor to target shape.

Parameters

- **im** (*torch.Tensor*) – Image tensor sized [C, H, W] or [N, C, H, W].
- **shape** (*tuple*) – Target shape as (height, width) tuple.
- **mode** (*str*) – Resampleing algorithm ('nearest' | 'linear' | 'bilinear' | 'bicubic' | 'area').
- **iterative_downsample** – Iteratively average pixels if image dimension must be reduced 2x or more.

Returns
Resampled image tensor sized [C, H, W] or [N, C, H, W].

`classmethod resize_le`(*im*, *size*, *mode='bilinear'*, *iterative_downsample=False*)

Resize image tensor by longest edge, constraining proportions.

Parameters

- **im** (*torch.Tensor*) – Image tensor sized [C, H, W] or [N, C, H, W]
- **size** (*int*) – Target size for longest edge
- **mode** (*str*) – Resampleing algorithm ('nearest' | 'linear' | 'bilinear' | 'bicubic' | 'area')
- **iterative_downsample** – Iteratively average pixels if image dimension must be reduced 2x or more.

Returns
Resampled image tensor sized [C, H, W] or [N, C, H, W]

Return type
torch.Tensor

`resize_le_to_next_pot`(*mode='bilinear'*)

Resize image tensor by longest edge up to next higher power-of-two, constraining proportions.

Parameters

- **im** (*torch.tensor*) – Image tensor sized [C, H, W] or [N, C, H, W].
- **mode** (*str*)

Returns
Resampled image tensor sized [C, H, W] or [N, C, H, W].

`classmethod resize_se`(*im*, *size*, *mode='bilinear'*, *iterative_downsample=False*)

Resize image tensor by shortest edge, constraining proportions.

Parameters

- **im** (*torch.Tensor*) – Image tensor sized [C, H, W] or [N, C, H, W]
- **size** (*int*) – Target size for shortest edge
- **mode** (*str*) – Resampleing algorithm ('nearest' | 'linear' | 'bilinear' | 'bicubic' | 'area')
- **iterative_downsample** – Iteratively average pixels if image dimension must be reduced 2x or more.

Returns
Resampled image tensor sized [C, H, W] or [N, C, H, W]

Return type
torch.Tensor

`classmethod sample_lod_bilinear`(*pyramid*, *height*, *width*, *lod*)

Resample a mip pyramid using standard bilinear interpolation at a given LOD.

Parameters

- **pyramid** (*torch.Tensor*) – Input mip pyramid tensor of shape [C, H, W], with stacked mips along height.
- **height** (*int*) – Output image height.
- **width** (*int*) – Output image width.

- **lod** (*float*) – Level-of-detail value to sample from (can be fractional).

Returns
Resampled image tensor of shape [C, height, width].

classmethod sample_lod_bspline_dither(*height*, *width*, *lod*)

Resample a mip pyramid using stochastic B-spline dithering.

Converts the B-spline weights into a probability distribution and samples values probabilistically.

Parameters

- **pyramid** – Input mip pyramid tensor of shape [C, H, W], with stacked mips along height.
- **height** (*int*) – Output image height.
- **width** (*int*) – Output image width.
- **lod** (*float*) – Level-of-detail value to sample from (can be fractional).

Returns
Resampled image tensor of shape [C, height, width].

classmethod sample_lod_bspline_hybrid(*pyramid*, *height*, *width*, *lod*)

Resample a mip pyramid using a hybrid (4-tap bilinear) B-spline filter approximation.

Parameters

- **pyramid** (*torch.Tensor*) – Input mip pyramid tensor of shape [C, H, W], with stacked mips along height.
- **height** (*int*) – Output image height.
- **width** (*int*) – Output image width.
- **lod** (*float*) – Level-of-detail value to sample from (can be fractional).

Returns
Resampled image tensor of shape [C, height, width].

classmethod tile(*im*, *shape*)

Tile/repeat image tensor to match target shape.

Parameters

- **im** (*torch.Tensor*) – Image tensor sized [C, H, W] or [N, C, H, W].
- **shape** (*tuple*) – Target shape as (height, width) tuple.

Returns
Padded image tensor sized [C, H, W] or [N, C, H, W].

Return type
torch.Tensor

classmethod tile_n(*im*, *repeat_h*, *repeat_w*)

Tile/repeat image tensor by number of repetitions.

Parameters

- **im** (*torch.Tensor*) – Image tensor sized [C, H, W] or [N, C, H, W]
- **repeat_h** (*int*) – Number of times to repeat image vertically.
- **repeat_w** (*int*) – Number of times to repeat image horizontally.

Returns
Padded image tensor sized [C, H, W] or [N, C, H, W].

`classmethod tile_to_square`(*im*, *target_size*)

Tile image tensor to square dimensions of target size.

Parameters

- **im** (*torch.Tensor*) – Image tensor sized [C, H, W] or [N, C, H, W].
- **target_size** (*int*)

Returns
Padded image tensor sized [C, H, W] or [N, C, H, W] where H = W.

Return type
torch.Tensor

`class tinytex.Atlas`(*min_size=64*, *max_size=8192*, *force_square=False*)

Bases: `object`

Texture atlas packing and sampling.

`add`(*key*, *tensor*)

Add a named texture to the atlas.

Parameters

- **key** (*str*) – Identifier for the texture.
- **tensor** (*torch.Tensor*) – Image tensor in CHW format.

Return type
None

`classmethod from_dir`(*path*, *ext='.png'*, *channels=3*, *allow_mismatch=True*, *max_h=0*, *max_w=0*, *crop=True*, *row=False*, *sort='height'*)

Load all matching image files from a directory and pack them.

Parameters

- **path** (*str*) – Path to image directory.
- **ext** (*str*) – File extension to match.
- **channels** (*int*) – Expected channel count.
- **allow_mismatch** (*bool*) – Auto pad/trim to match channels.
- **max_h** (*int*) – Optional max height for atlas.
- **max_w** (*int*) – Optional max width for atlas.
- **crop** (*bool*) – Whether to crop excess space after packing.
- **row** (*bool*) – Whether to use row packing.
- **sort** (*str*) – Sorting mode.

Returns
Packed Atlas instance.

Return type
Atlas

generate_mask(*shape*, *scale=1.0*, *samples=2*)

Generate a tiling mask by randomly overlaying textures.

Parameters

- **shape** (*tuple*) – Output (H, W) of the canvas.
- **scale** (*float*) – Relative size of overlays.
- **samples** (*int*) – Density multiplier for overlays.

Returns
Output image tensor.

Return type
torch.Tensor

pack(*max_h=0*, *max_w=0*, *crop=True*, *row=False*, *sort='height'*)

Pack added textures into a single atlas image.

Parameters

- **max_h** (*int*) – Max atlas height. If 0, auto-expand.
- **max_w** (*int*) – Max atlas width. If 0, auto-expand.
- **crop** (*bool*) – Crop output to tight bounding box.
- **row** (*bool*) – Use row-based packing instead of rectangle packing.
- **sort** (*str*) – Sorting mode for texture order ('height', 'width', 'area').

Returns
(Atlas tensor, index dictionary with coordinates per texture).

Return type
tuple

sample(*key*)

Retrieve a single texture by name or index.

Parameters
key (*str | int*) – Texture name or integer index.

Returns
Image tensor sliced from the atlas.

Return type
torch.Tensor

sample_random()

Retrieve a randomly chosen texture from the atlas.

Returns
Image tensor sliced from the atlas.

Return type
torch.Tensor

class tinytex.**SurfaceOps**

Bases: `object`

Geometric surface processing. Where applicable, assumes a right-handed y-up, x-right, z-back coordinate system and OpenGL-style normal maps.

classmethod angles_to_normals(*angle_map*, *recompute_z=False*, *normalize=False*, *rescaled=False*)

Convert scaled spherical coordinates to tangent-space normal vectors.

Parameters

- **angle_map** (*torch.Tensor*) – Scaled spherical coordinates tensor sized [N, C=2, H, W] or [C=2, H, W], in range [0, 1].
- **reompute_z** – Discard and recompute normal map's z-channel after conversion.
- **normalize** (*bool*) – Normalize vectors after conversion.
- **rescaled** (*bool*) – Input and returned tensors should be in [0, 1] value range.
- **recompute_z** (*bool*)

Returns

Tensor of normals as unit vectors sized [N, C=3, H, W] or [C=3, H, W].

Return type

torch.Tensor

classmethod blend_normals(*normals_base*, *normals_detail*, *rescaled=False*, *eps=1e-08*)

Blend two normal maps with reoriented normal map algorithm.

Parameters

- **normals_base** (*torch.Tensor*) – Base normals tensor sized [N, C=3, H, W] or [C=3, H, W] as unit vectors of surface normals
- **normals_detail** (*torch.Tensor*) – Detail normals tensor sized [N, C=3, H, W] or [C=3, H, W] as unit vectors of surface normals
- **rescaled** (*bool*) – Input and returned unit vector tensors should be in [0, 1] value range.
- **eps** (*float*) – epsilon

Returns

blended normals tensor sized [N, C=3, H, W] or [C=3, H, W] as unit vectors of surface normals

classmethod compute_occlusion(*normal_map=None*, *height_map=None*, *height_scale=1.0*, *radius=0.08*, *n_samples=256*, *rescaled=False*)

Compute ambient occlusion and bent normals from normal map and/or height map.

Parameters

- **height_map** (*torch.Tensor*) – Height map tensor sized [N, C=1, H, W] or [C=1, H, W] in [0, 1] range.
- **normal_map** (*torch.Tensor*) – Normal map tensor sized [N, C=3, H, W] or [C=3, H, W] as unit vectors. of surface normals
- **height_scale** (*torch.Tensor | float*) – Height scale as tensor sized [N, C=1] or [C=1], or as float.
- **radius** (*float*) – Occlusion radius.
- **n_samples** (*int*) – Number of occlusion samples per pixel.
- **rescaled** (*bool*) – Input and returned unit vector tensors should be in [0, 1] value range.

Returns

Ambient occlusion tensor sized [N, C=1, H, W] or [C=1, H, W], bent normals tensor sized [N, C=3, H, W] or [C=3, H, W].

Return type
(torch.Tensor, torch.Tensor)

`classmethod height_to_curvature`(*height_map*, *blur_kernel_size=0.0078125*, *blur_iter=1*)

Estimate mean curvature from a height map.

Parameters

- **height_map** (*torch.Tensor*) – [N, 1, H, W] or [1, H, W] tensor.
- **blur_kernel_size** (*float*) – Relative kernel size for Gaussian blur.
- **blur_iter** (*int*) – Number of blur passes.

Returns
(curvature, cavities, peaks) — each [N, 1, H, W]

Return type
tuple

`classmethod height_to_normals`(*height_map*, *rescaled=False*, *eps=0.0001*)

Compute tangent-space normals form height.

Parameters

- **height_map** (*torch.Tensor*) – Height map tensor sized [N, C=1, H, W] or [C=1, H, W] in [0, 1] range
- **rescaled** (*bool*) – Return unit vector tensor in [0, 1] value range.
- **eps** (*float*) – Epsilon

Returns
normals tensor sized [N, C=3, H, W] or [C=3, H, W] as unit vectors of surface normals.

Return type
torch.Tensor

`classmethod normalize`(*normal_map*, *rescaled=False*)

Normalize xyz vectors to unit length.

Parameters

- **normal_map** (*torch.Tensor*) – Tensor of normal vectors sized [N, C=3, H, W] or [C=3, H, W].
- **rescaled** (*bool*) – Input and returned unit vector tensors should be in [0, 1] value range.

Returns
Normalized tensor sized [N, C=3, H, W] or [C=3, H, W].

Return type
torch.Tensor

`classmethod normals_to_angles`(*normal_map*, *recompute_z=False*, *normalize=False*, *rescaled=False*)

Convert tangent-space normal vectors to scaled spherical coordinates.

Parameters

- **normal_map** (*torch.Tensor*) – Input normal map sized [N, C=3, H, W] or [C=3, H, W].
- **recompute_z** (*bool*) – Discard and recompute normals' z-channel before conversion.
- **normalize** (*bool*) – Normalize vectors before conversion.
- **rescaled** (*bool*) – Input and returned tensors should be in [0, 1] value range.

Returns
Scaled z-axis and y-axis angles tensor sized [N, C=2, H, W] or [C=2, H, W], in range [0, 1].

`classmethod normals_to_height`(*normal_map*, *self_tiling=False*, *rescaled=False*, *eps=torch.finfo.eps*)

Compute height from normals. Frankot-Chellappa algorithm.

Parameters

- **normal_map** (*torch.Tensor*) – Normal map tensor sized [N, C=3, H, W] or [C=3, H, W] as unit vectors of surface normals.
- **self_tiling** (*bool*) – Treat surface as self-tiling.
- **rescaled** (*bool*) – Accept unit vector tensor in [0, 1] value range.
- **eps** (*float*)

Returns
Height tensor sized [N, C=1, H, W] or [C=1, H, W] in [0, 1] range and height scale tensor sized [N, C=1] or [C=1] in [0, inf] range.

Return type
(torch.Tensor, torch.Tensor)

`classmethod recompute_z`(*normal_map*, *rescaled=False*)

Discard and recompute the z component of xyz vectors for a tangent-space normal map.

Parameters

- **normal_map** (*torch.Tensor*) – Tensor of normal vectors sized [N, C=3, H, W] or [C=3, H, W].
- **rescaled** (*bool*) – Input and returned unit vector tensors should be in [0, 1] value range.

Returns
Normals tensor with reconstructed z-channel sized [N, C=3, H, W] or [C=3, H, W].

Return type
torch.Tensor

`class tinytex.Smoothstep`(*interpolant*, *n=None*)

Bases: `object`

Smoothstep interpolation.

Parameters

- **interpolant** (*str* | `Interpolant`)
- **n** (*int*)

`class Interpolant`(*value*)

Bases: `IntEnum`

Interpolant. See: https://iquilezles.org/articles/smoothsteps/

Table 4.2: Available Interpolants:

Identifier	Inverse Identifier	Continuity
CUBIC_POLYNOMIAL	INV_CUBIC_POLYNOMIAL	C1
QUARTIC_POLYNOMIAL	INV_QUARTIC_POLYNOMIAL	C1
QUINTIC_POLYNOMIAL		C2
QUADRATIC_RATIONAL	INV_QUADRATIC_RATIONAL	C1
CUBIC_RATIONAL	INV_CUBIC_RATIONAL	C2
RATIONAL	INV_RATIONAL	CV
PIECEWISE_QUADRATIC	INV_PIECEWISE_QUADRATIC	C1
PIECEWISE_POLYNOMIAL	INV_PIECEWISE_POLYNOMIAL	CV
TRIGONOMETRIC	INV_TRIGONOMETRIC	C1

`classmethod apply`(*f*, *e0*, *e1*, *x*, *n=None*)

Static smoothstep evaluation with interpolant.

Parameters

- **f** (`Interpolant | str`) – Interpolant.
- **e0** (*float*) – Lower edge (min).
- **e1** (*float*) – Upper edge (max).
- **x** (*float | torch.Tensor*) – Value.
- **n** (*int*) – Order (if applicable).

Returns

Interpolated result.

Return type

(<class 'float'>, torch.Tensor)

`classmethod cubic_polynomial`(*x*)

Cubic polynomial - Hermite interpolation.

`classmethod cubic_rational`(*x*)

Cubic rational interpolation.

`forward`(*e0*, *e1*, *x*)

Apply forward smoothstep interpolation.

Parameters

- **e0** – Lower bound.
- **e1** – Upper bound.
- **x** – Input value(s).

Returns

Interpolated output.

`classmethod interpolate`(*f*, *x*, *n=None*)

Dispatch interpolation function based on interpolant.

Parameters

- **f** (`Interpolant | str`) – Interpolant.
- **x** (*float | torch.Tensor*) – Normalized input.

- **n** (*int*) – Order (if applicable).

Returns
Interpolated output.

`classmethod inv_cubic_polynomial(x)`
Inverse cubic polynomial interpolation.

`classmethod inv_cubic_rational(x)`
Inverse cubic rational interpolation.

`classmethod inv_piecewise_polynomial(x, n)`
Inverse piecewise polynomial interpolation.

`classmethod inv_piecewise_quadratic(x)`
Inverse piecewise quadratic interpolation.

`classmethod inv_quadratic_rational(x)`
Inverse quadratic rational interpolation.

`classmethod inv_quartic_polynomial(x)`
Inverse quartic polynomial interpolation.

`classmethod inv_rational(x, n)`
Inverse rational interpolation.

`classmethod inv_trigonometric(x)`
Inverse trigonometric interpolation.

`inverse(y)`
Apply inverse interpolation.

Parameters
y – Interpolated value.

Returns
Original input value.

`classmethod piecewise_polynomial(x, n)`
Piecewise polynomial interpolation.

`classmethod piecewise_quadratic(x)`
Piecewise quadratic interpolation.

`classmethod quadratic_rational(x)`
Quadratic rational interpolation.

`classmethod quartic_polynomial(x)`
Quartic polynomial interpolation.

`classmethod quintic_polynomial(x)`
Quintic polynomial interpolation.

`classmethod rational(x, n)`
Rational interpolation.

`classmethod trigonometric(x)`
Trigonometric interpolation.

class tinytex.**SpectralNoise**

Bases: object

Spectral-domain procedural noise generators.

classmethod blue(*height*, *width*)

Generate 2D blue noise (f spectrum).

Parameters

- **height** (*int*) – Output height.
- **width** (*int*) – Output width.

Returns
2D tensor of blue noise with shape (height, width).

Return type
torch.Tensor

classmethod brownian(*height*, *width*)

Generate 2D brownian (red) noise ($1/f^2$ spectrum).

Parameters

- **height** (*int*) – Output height.
- **width** (*int*) – Output width.

Returns
2D tensor of brownian noise with shape (height, width).

Return type
torch.Tensor

classmethod noise_psd_2d(*height*, *width*, *psd=<function SpectralNoise.<lambda>>*)

Generate spectral 2D noise field. Shape (height, width) with a spectral shaping function psd.

Parameters

- **height** (*int*) – Field height.
- **width** (*int*) – Field width.
- **psd** – a function that accepts a tensor f of shape (height, width//2+1) of frequency magnitudes and returns a tensor of the same shape.

classmethod pink(*height*, *width*)

Generate 2D pink noise (1/f spectrum).

Parameters

- **height** (*int*) – Output height.
- **width** (*int*) – Output width.

Returns
2D tensor of pink noise with shape (height, width).

Return type
torch.Tensor

classmethod violet(*height*, *width*)

Generate 2D violet noise (f^2 spectrum).

Parameters

- **height** (*int*) – Output height.
- **width** (*int*) – Output width.

Returns
2D tensor of violet noise with shape (height, width).

Return type
torch.Tensor

classmethod white(*height*, *width*)

Generate 2D white noise (flat power spectrum).

Parameters

- **height** (*int*) – Output height.
- **width** (*int*) – Output width.

Returns
2D tensor of white noise with shape (height, width).

Return type
torch.Tensor

class tinytex.SpatialNoise

Bases: `object`

Spatial-domain procedural noise generators.

classmethod fractal(*shape*, *density=5.0*, *octaves=5*, *persistence=0.5*, *lacunarity=2*, *tileable=(True, True)*, *interpolant='quintic_polynomial'*)

Generate 2D fractal noise using layered Perlin noise.

Parameters

- **shape** (*tuple*) – Output shape as (height, width).
- **density** (*float*) – Base frequency scale.
- **octaves** (*int*) – Number of noise layers.
- **persistence** (*float*) – Amplitude falloff per octave.
- **lacunarity** (*int*) – Frequency multiplier per octave.
- **tileable** (*tuple*) – Whether noise should tile along each axis.
- **interpolant** (*str*) – Interpolation function name.

Returns
Tensor of shape [1, H, W] with values in [0, 1].

Return type
torch.Tensor

classmethod perlin(*shape*, *density=5.0*, *tileable=(True, True)*, *interpolant='quintic_polynomial'*)

Generate 2D Perlin noise.

Parameters

- **shape** (*tuple*) – Output shape as (height, width).
- **density** (*float*) – Controls frequency of the noise pattern.
- **tileable** (*tuple*) – Whether noise should tile along each axis.
- **interpolant** (*str*) – Interpolation function name (e.g., 'linear', 'quintic_polynomial').

Returns
Tensor of shape [1, H, W] with values in [0, 1].

Return type
torch.Tensor

`classmethod turbulence`(*shape*, *density=5.0*, *octaves=5*, *persistence=0.5*, *lacunarity=2*, *tileable=(True, True)*, *interpolant='quintic_polynomial'*, *ridge=False*)

Generate 2D turbulence noise (absolute layered Perlin).

Parameters

- **shape** (*tuple*) – Output shape as (height, width).
- **density** (*float*) – Base frequency scale.
- **octaves** (*int*) – Number of noise layers.
- **persistence** (*float*) – Amplitude falloff per octave.
- **lacunarity** (*int*) – Frequency multiplier per octave.
- **tileable** (*tuple*) – Whether noise should tile along each axis.
- **interpolant** (*str*) – Interpolation function name.
- **ridge** (*bool*) – If True, applies ridge-remapping for sharper features.

Returns
Tensor of shape [1, H, W] with values in [0, 1].

Return type
torch.Tensor

`classmethod worley`(*shape*, *density=5.0*, *intensity=1.0*, *tileable=(True, True)*)

Generate 2D Worley (cellular) noise.

Parameters

- **shape** (*tuple*) – Output shape as (height, width).
- **density** (*float*) – Number of feature points per axis.
- **intensity** (*float*) – Multiplier for the distance field.
- **tileable** (*tuple*) – Whether noise should tile along each axis.

Returns
Tensor of shape [1, H, W] with values in [0, 1].

Return type
torch.Tensor

`class tinytex.Voronoi`

Bases: `object`

Voronoi-based generators.

classmethod snap_nearest_as(*uv_screen*, *vn_scale*, *scale*, *zoom_to*, *aspect*, *seed*)

Perturb UVs with a Voronoi field.

Parameters

- **uv_screen** (*torch.Tensor*) – UV coordinates of shape [2, H, W] in the range [0, 1]
- **vn_scale** (*float*) – Voronoi scale factor that controls the size of the cells
- **scale** (*float*) – Scene scale (zoom) factor
- **zoom_to** (*torch.Tensor*) – Zoom center as a tensor of shape [2,] in normalized space
- **aspect** (*float*) – Aspect ratio (width/height) of the scene
- **seed** (*int*) – Random seed used to generate the Voronoi field

Returns

A tensor of shape [6, H, W] where:

- Channel 0: Perturbed UVs (u)
- Channel 1: Perturbed UVs (v)
- Channel 2: Voronoi cell ID (cell_id_x)
- Channel 3: Voronoi cell ID (cell_id_y)
- Channel 4: Distance from the edge of the Voronoi cell (edge_dist)
- Channel 5: Distance from the center of the Voronoi cell (center_dist)

Return type

torch.Tensor

classmethod snap_offset(*x*, *seed*)

Vectorized Voronoi offset and distances. Given 2D position *x* and seed, returns F1, F2, best_offset_x, best_offset_y.

Parameters

- **x** (*torch.Tensor*)
- **seed** (*int*)

class tinytex.SDF

Bases: object

Signed distance field computation and rendering.

classmethod box(*size=64*, *box_shape=(32, 32)*, *length_factor=0.1*, *tile_to=None*)

Generate a rectangular SDF centered in the image.

Parameters

- **size** (*int*) – Output image size (square).
- **box_shape** (*tuple*) – (height, width) of the rectangle.
- **length_factor** (*float*) – Distance scaling factor.
- **tile_to** (*int | None*) – Optional output tiling target.

Returns

SDF tensor of shape [1, size, size].

Return type
torch.Tensor

classmethod circle(*size=64*, *radius=20*, *length_factor=0.1*, *tile_to=None*)

Generate a circular SDF centered in the image.

Parameters

- **size** (*int*) – Output image size (square).
- **radius** (*int*) – Radius of the circle.
- **length_factor** (*float*) – Distance scaling factor.
- **tile_to** (*int | None*) – Optional output tiling target.

Returns
SDF tensor of shape [1, size, size].

Return type
torch.Tensor

classmethod compute(*im*, *periodic=True*, *length_factor=0.1*, *threshold=None*, *tile_to=None*)

Compute a signed distance field from a binary image.

Parameters

- **im** (*torch.Tensor*) – Input tensor of shape [1, H, W].
- **periodic** (*bool*) – Whether to use periodic boundary conditions.
- **length_factor** (*float*) – Scales the maximum measurable distance.
- **threshold** (*float | None*) – Binarization threshold if *im* isn't binary.
- **tile_to** (*tuple | None*) – Optional tiling target shape (H, W).

Returns
Tensor of shape [1, H, W] with values in [0, 1].

Return type
torch.Tensor

classmethod max(*sdf1*, *sdf2*)

Return the maximum of two SDFs (intersection).

Parameters

- **sdf1** (*torch.Tensor*)
- **sdf2** (*torch.Tensor*)

Return type
torch.Tensor

classmethod min(*sdf1*, *sdf2*)

Return the minimum of two SDFs (union).

Parameters

- **sdf1** (*torch.Tensor*)
- **sdf2** (*torch.Tensor*)

Return type
torch.Tensor

classmethod render(*sdf*, *shape*, *edge0=0.496*, *edge1=0.498*, *value0=0.0*, *value1=1.0*, *interpolant='quintic_polynomial'*, *mode='bilinear'*)

Render an SDF to a grayscale field using soft-thresholding.

Parameters

- **sdf** (*torch.Tensor*) – Input SDF tensor of shape [1, H, W].
- **shape** (*tuple*) – Output size (height, width).
- **edge0** (*float*) – Lower edge of the soft transition zone.
- **edge1** (*float*) – Upper edge of the soft transition zone.
- **value0** (*float*) – Output value below edge0.
- **value1** (*float*) – Output value above edge1.
- **interpolant** (*str*) – Interpolation curve used between edge0 and edge1.
- **mode** (*str*) – Interpolation method for resizing.

Returns

Rendered tensor of shape [1, H, W].

Return type

torch.Tensor

classmethod segment(*size=64*, *a=(32, 0)*, *b=(32, 64)*, *length_factor=0.1*, *tile_to=None*)

Generate an SDF for a finite line segment.

Parameters

- **size** (*int*) – Output image size (square).
- **a** (*tuple*) – Starting point of the segment.
- **b** (*tuple*) – Ending point of the segment.
- **length_factor** (*float*) – Distance scaling factor.
- **tile_to** (*int | None*) – Optional output tiling target.

Returns

SDF tensor of shape [1, size, size].

Return type

torch.Tensor

class tinytex.Tiling

Bases: `object`

Tiling and tile blending.

classmethod blend(*tiles*, *rows=1*, *cols=1*, *wrap=True*, *vector_data=False*)

Blend tiles to remove seams. Uses Poisson solver to match image gradients.

Note

This is a computationally expensive task. For much faster performance, install AMGCL or PyAMG and they will be used over SciPy's spsolve automatically.

Parameters

- **tiles** (*torch.Tensor*) – Tiles as pytorch image tensor sized [N, C, H, W] or [C, H, W].
- **rows** (*int*) – Total number of rows in tile grid.
- **cols** (*int*) – Total number of columns in tile grid.
- **wrap** (*bool*) – Wrap tile grid border (allows self-tiling).
- **vector_data** (*bool*) – Tiles contain directional unit vectors in [-1, 1] range - i.e. a normal map. If True, vectors will be converted to angles for blending so that component gradients can be matched independently.

Returns
Blended tiles sized [N, C, H, W] or [C, H, W].

Return type
torch.Tensor

`classmethod get_tile_index(r, c, cols)`

Get tile index, by row and column position.

Parameters

- **idx** – Tile index.
- **r** (*int*) – Tile's row position.
- **c** (*int*) – Tile's column position.
- **cols** (*int*) – Total number of columns in tile grid.

Returns
row and column position, respectively

Return type
int

`classmethod get_tile_neighbors(r, c, rows, cols, wrap=False)`

Get indices of adjacent tiles from tile's row and column position.

Parameters

- **r** (*int*) – Tile's row position.
- **c** (*int*) – Tile's column position.
- **rows** (*int*) – Total number of rows in tile grid.
- **cols** (*int*) – Total number of columns in tile grid.
- **wrap** (*bool*) – Wrap around edge tiles to opposite side of grid.

Returns
Tile indices of top, right, bottom, and left neighboring tiles, respectively, or -1 if no neighboring tile (when wrap is False).

Return type
(<class 'int'>, <class 'int'>, <class 'int'>, <class 'int'>)

`classmethod get_tile_position(idx, cols)`

Get row and column position of tile in tile grid, by tile index.

Parameters

- **idx** (*int*) – tile index

- **cols** (*int*) – total number of columns in tile grid

Returns
row and column position, respectively

Return type
(<class 'int'>, <class 'int'>)

classmethod merge(*tiles*, *rows*, *cols*)

Combine non-overlapping tiles into composite image tensor. Tiles are expected to be ordered left-to-right and top-to-bottom:

```
| 0 | 1 | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |
```

Parameters

- **tiles** (*torch.Tensor*) – Tiles as pytorch image tensor sized [N, C, H, W].
- **rows** (*int*) – Total number of rows in tile grid.
- **cols** (*int*) – Total number of columns in tile grid.

Returns
Combined image tensor sized [C, H, W].

Return type
torch.Tensor

classmethod split(*im*, *shape*)

Split image tensor into non-overlapping square tiles. Tiles are ordered left-to-right and top-to-bottom:

```
| 0 | 1 | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |
```

Warning

If image dimensions are not evenly divisible by tile size, image will be effectively cropped, based on how many tiles can fit.

Parameters

- **im** (*torch.Tensor*) – Image tensor sized [N=1, C, H, W] or [C, H, W].
- **shape** (*tuple*) – Tile shape (height, width) in pixels.

Returns
Image tensor sized [N, C, H, W], number of rows, number of columns.

Return type
(torch.Tensor, <class 'int'>, <class 'int'>)

class tinytex.**Wavelet**

Bases: `object`

Wavelet transforms and coefficient culling.

classmethod cull_haar_2d_aw(*a*, *ratio*)

Keep only the strongest Haar coefficients in a 2D image by area-weighted magnitude.

Parameters

- **a** (*torch.Tensor*) – 2D Haar wavelet coefficients tensor sized [H, W] or [N, H, W], where N is batch count, H is height and W is width.
- **ratio** (*float*) – Ratio of coefficients to cull, in range [0, 1].

Return type
torch.Tensor

classmethod cull_haar_magnitude(*a*, *ratio*)

Keep only the strongest Haar coefficients by magnitude.

Parameters

- **a** (*torch.Tensor*) – Haar wavelet coefficients tensor sized [F] or [N, F], where N is batch count and F is coefficient count.
- **ratio** (*float*) – Ratio of coefficients to cull, in range [0, 1].

Return type
torch.Tensor

classmethod haar(*a*)

1D Haar transform.

Parameters
a (*torch.Tensor*)

Return type
torch.Tensor

classmethod haar_2d(*im*)

2D Haar transform.

Parameters
im (*torch.Tensor*)

Return type
torch.Tensor

classmethod haar_bipolar(*im*)

Scales Haar coefficients to range [0, 1]. Returns [C=3, H, W] sized tensor where negative values are red, positive values are blue, and zero is black.

Parameters
im (*torch.Tensor*)

Return type
torch.Tensor

classmethod inverse_haar(*a*)

1D inverse Haar transform.

Parameters
`a` (*torch.Tensor*)

Return type
torch.Tensor

classmethod inverse_haar_2d(*coeffs*)

2D inverse Haar transform.

Parameters
`coeffs` (*torch.Tensor*)

Return type
torch.Tensor

class tinytex.Warping

Bases: `object`

Warping and coordinate system translation.

classmethod inverse_log_polar(*im*)

Inverse log polar transform

Parameters
`im` (*torch.Tensor*)

Return type
torch.Tensor

classmethod log_polar(*im*, *start_from=1*, *n_angular=None*, *n_radial=None*)

Log polar transform

Parameters

- `im` (*torch.Tensor*)
- `start_from` (*int*)

Return type
torch.Tensor

4.1.1 rand module

Hash

pt_hash_uint(x[, normalize])	1D XQO-style hash for PyTorch.
pt_hash2_uint(x, y[, normalize])	2D XQO-style hash for PyTorch.
pt_hash2_uv(uv[, seed, tile_size])	2D normalized UV hash for PyTorch.
pt_hash2_xy(xy[, seed, tile_size])	2D denormalized XY hash for PyTorch.
np_hash_uint(x[, normalize])	1D XQO-style hash for NumPy.
np_hash2_uint(x, y[, normalize])	2D XQO-style hash for NumPy.

Scramble

pt_noise2(p)	2D value noise scramble.
pt_noise3(p)	3D value noise scramble.

`tinytex.rand.`**`pt_hash_uint`**`(`*x*, *normalize=False*`)`

1D XQO-style hash for PyTorch.

Parameters

- **`x`** (*`torch.Tensor`*) – Input int tensor.
- **`normalize`** (*`bool`*) – If True, returns float32 values in [0, 1].

Returns

Hashed tensor (uint32 or float32).

Return type

torch.Tensor

`tinytex.rand.`**`pt_hash2_uint`**`(`*x*, *y*, *normalize=False*`)`

2D XQO-style hash for PyTorch.

Parameters

- **`x`** (*`torch.Tensor`*) – X coordinates (int tensor).
- **`y`** (*`torch.Tensor`*) – Y coordinates (int tensor).
- **`normalize`** (*`bool`*) – If True, returns float32 in [0, 1].

Returns

Hashed values (uint32 or float32).

Return type

torch.Tensor

`tinytex.rand.`**`pt_hash2_uv`**`(`*uv*, *seed=0*, *tile_size=1023*`)`

2D normalized UV hash for PyTorch. Expects normalized coordinates. Produces uniform pseudo-random values.

Parameters

- **`uv`** (*`torch.Tensor`*) – Tensor of shape [2, H, W] with values in [0, 1].
- **`seed`** (*`int`*) – Optional seed for scrambling.
- **`tile_size`** (*`int`*) – Maximum tile size (default: 1023).

Returns

Tensor of shape [H, W] with float32 values in [0, 1].

Return type

torch.Tensor

`tinytex.rand.`**`pt_hash2_xy`**`(`*xy*, *seed=0*, *tile_size=1023*`)`

2D denormalized XY hash for PyTorch. Expects denormalized integer coordinates. Produces uniform pseudo-random values.

Parameters

- **`xy`** (*`torch.Tensor`*) – Tensor of shape [2, H, W] with integer coordinates.
- **`seed`** (*`int`*) – Optional seed for scrambling.
- **`tile_size`** (*`int`*) – Maximum tile size (default: 1023).

Returns

Tensor of shape [H, W] with float32 values in [0, 1].

Return type

torch.Tensor

`tinytex.rand.np_hash_uint(x, normalize=False)`

1D XQO-style hash for NumPy.

Parameters

- **x** (*ndarray*) – Input array (uint32-compatible).
- **normalize** (*bool*) – If True, returns float32 values in [0, 1].

Returns
Hashed values (uint32 or float32).

Return type
ndarray

`tinytex.rand.np_hash2_uint(x, y, normalize=False)`

2D XQO-style hash for NumPy.

Parameters

- **x** (*ndarray*) – X array (uint32-compatible).
- **y** (*ndarray*) – Y array (uint32-compatible).
- **normalize** – If True, returns float32 values in [0, 1].

Returns
Hashed values (uint32 or float32).

Return type
ndarray

`tinytex.rand.pt_noise2(p)`

2D value noise scramble.

Parameters
p (*torch.Tensor*) – Input tensor of shape [2, H, W].

Returns
Scrambled tensor of shape [2, H, W], float32 in [0, 1].

Return type
torch.Tensor

`tinytex.rand.pt_noise3(p)`

3D value noise scramble.

Parameters
p (*torch.Tensor*) – Input tensor of shape [3, H, W].

Returns
Scrambled tensor of shape [3, H, W], float32 in [0, 1].

Return type
torch.Tensor

4.1.2 ti module

4.2 Release notes

v 0.2.0 a - May 2025

- Multiple bug fixes.

- Multiple API changes. Atlas class now has state.
- Moved PRNG to submodule.
- How-to sections for basic operations.
- Added smoothstep unit test as scaffolding for future tests.

v 0.1.5 a - May 2025

- Fix and consolidate class methods.

v 0.1.2 a - May 2025

- Fix broken occlusion and curvature functions.

v 0.1.1 a - May 2025

- Initial commit.

4.3 License

CHAPTER

FIVE

LINKS

- GitHub
- PyPi

CHAPTER

SIX

SIBLING PROJECTS

- tinycio
- tinyfilm

PYTHON MODULE INDEX

t

tinytex, 15
tinytex.rand, 35

INDEX

A

add() (*tinytex.Atlas method*), 19
angles_to_normals() (*tinytex.SurfaceOps class method*), 20
apply() (*tinytex.Smoothstep class method*), 24
Atlas (*class in tinytex*), 19

B

blend() (*tinytex.Tiling class method*), 31
blend_normals() (*tinytex.SurfaceOps class method*), 21
blue() (*tinytex.SpectralNoise class method*), 26
box() (*tinytex.SDF class method*), 29
brownian() (*tinytex.SpectralNoise class method*), 26

C

circle() (*tinytex.SDF class method*), 30
compute() (*tinytex.SDF class method*), 30
compute_lod_offsets() (*tinytex.Resampling class method*), 15
compute_occlusion() (*tinytex.SurfaceOps class method*), 21
crop() (*tinytex.Resampling class method*), 15
cubic_polynomial() (*tinytex.Smoothstep class method*), 24
cubic_rational() (*tinytex.Smoothstep class method*), 24
cull_haar_2d_aw() (*tinytex.Wavelet class method*), 34
cull_haar_magnitude() (*tinytex.Wavelet class method*), 34

F

forward() (*tinytex.Smoothstep method*), 24
fractal() (*tinytex.SpatialNoise class method*), 27
from_dir() (*tinytex.Atlas class method*), 19

G

generate_mask() (*tinytex.Atlas method*), 19
generate_mip_pyramid() (*tinytex.Resampling class method*), 16
get_tile_index() (*tinytex.Tiling class method*), 32
get_tile_neighbors() (*tinytex.Tiling class method*), 32
get_tile_position() (*tinytex.Tiling class method*), 32

H

haar() (*tinytex.Wavelet class method*), 34
haar_2d() (*tinytex.Wavelet class method*), 34
haar_bipolar() (*tinytex.Wavelet class method*), 34
height_to_curvature() (*tinytex.SurfaceOps class method*), 22
height_to_normals() (*tinytex.SurfaceOps class method*), 22

I

interpolate() (*tinytex.Smoothstep class method*), 24
inv_cubic_polynomial() (*tinytex.Smoothstep class method*), 25
inv_cubic_rational() (*tinytex.Smoothstep class method*), 25
inv_piecewise_polynomial() (*tinytex.Smoothstep class method*), 25
inv_piecewise_quadratic() (*tinytex.Smoothstep class method*), 25
inv_quadratic_rational() (*tinytex.Smoothstep class method*), 25
inv_quartic_polynomial() (*tinytex.Smoothstep class method*), 25
inv_rational() (*tinytex.Smoothstep class method*), 25
inv_trigonometric() (*tinytex.Smoothstep class method*), 25
inverse() (*tinytex.Smoothstep method*), 25
inverse_haar() (*tinytex.Wavelet class method*), 34
inverse_haar_2d() (*tinytex.Wavelet class method*), 35
inverse_log_polar() (*tinytex.Warping class method*), 35

L

log_polar() (*tinytex.Warping class method*), 35

M

max() (*tinytex.SDF class method*), 30
merge() (*tinytex.Tiling class method*), 33

min() (*tinytex.SDF class method*), 30
module
    tinytex, 15
    tinytex.rand, 35

N

noise_psd_2d() (*tinytex.SpectralNoise class method*), 26
normalize() (*tinytex.SurfaceOps class method*), 22
normals_to_angles() (*tinytex.SurfaceOps class method*), 22
normals_to_height() (*tinytex.SurfaceOps class method*), 23
np_hash2_uint() (*in module tinytex.rand*), 37
np_hash_uint() (*in module tinytex.rand*), 36

P

pack() (*tinytex.Atlas method*), 20
pad_rb() (*tinytex.Resampling class method*), 16
pad_to_next_pot() (*tinytex.Resampling class method*), 16
perlin() (*tinytex.SpatialNoise class method*), 27
piecewise_polynomial() (*tinytex.Smoothstep class method*), 25
piecewise_quadratic() (*tinytex.Smoothstep class method*), 25
pink() (*tinytex.SpectralNoise class method*), 26
pt_hash2_uint() (*in module tinytex.rand*), 36
pt_hash2_uv() (*in module tinytex.rand*), 36
pt_hash2_xy() (*in module tinytex.rand*), 36
pt_hash_uint() (*in module tinytex.rand*), 35
pt_noise2() (*in module tinytex.rand*), 37
pt_noise3() (*in module tinytex.rand*), 37

Q

quadratic_rational() (*tinytex.Smoothstep class method*), 25
quartic_polynomial() (*tinytex.Smoothstep class method*), 25
quintic_polynomial() (*tinytex.Smoothstep class method*), 25

R

rational() (*tinytex.Smoothstep class method*), 25
recompute_z() (*tinytex.SurfaceOps class method*), 23
render() (*tinytex.SDF class method*), 30
Resampling (*class in tinytex*), 15
resize() (*tinytex.Resampling class method*), 16
resize_le() (*tinytex.Resampling class method*), 17
resize_le_to_next_pot() (*tinytex.Resampling method*), 17
resize_se() (*tinytex.Resampling class method*), 17

S

sample() (*tinytex.Atlas method*), 20
sample_lod_bilinear() (*tinytex.Resampling class method*), 17
sample_lod_bspline_dither() (*tinytex.Resampling class method*), 18
sample_lod_bspline_hybrid() (*tinytex.Resampling class method*), 18
sample_random() (*tinytex.Atlas method*), 20
SDF (*class in tinytex*), 29
segment() (*tinytex.SDF class method*), 31
Smoothstep (*class in tinytex*), 23
Smoothstep.Interpolant (*class in tinytex*), 23
snap_nearest_as() (*tinytex.Voronoi class method*), 28
snap_offset() (*tinytex.Voronoi class method*), 29
SpatialNoise (*class in tinytex*), 27
SpectralNoise (*class in tinytex*), 25
split() (*tinytex.Tiling class method*), 33
SurfaceOps (*class in tinytex*), 20

T

tile() (*tinytex.Resampling class method*), 18
tile_n() (*tinytex.Resampling class method*), 18
tile_to_square() (*tinytex.Resampling class method*), 19
Tiling (*class in tinytex*), 31
tinytex
    module, 15
tinytex.rand
    module, 35
trigonometric() (*tinytex.Smoothstep class method*), 25
turbulence() (*tinytex.SpatialNoise class method*), 28

V

violet() (*tinytex.SpectralNoise class method*), 26
Voronoi (*class in tinytex*), 28

W

Warping (*class in tinytex*), 35
Wavelet (*class in tinytex*), 33
white() (*tinytex.SpectralNoise class method*), 27
worley() (*tinytex.SpatialNoise class method*), 28