redner

Tzu-Mao Li

Mar 28, 2022

CONTENTS:

redner is a differentiable renderer that can take the derivatives of rendering output with respect to arbitrary scene parameters, that is, you can backpropagate from the image to your 3D scene. One of the major usages of redner is inverse rendering (hence the name redner) through gradient descent. What sets redner apart are: 1) it computes correct rendering gradients stochastically without any approximation and 2) it has a physically-based mode – which means it can simulate photons and produce realistic lighting phenomena, such as shadow and global illumination, and it handles the derivatives of these features correctly. You can also use redner in a fast deferred rendering mode for local shading: in this mode it still has correct gradient estimation and more elaborate material models compared to most differentiable renderers out there.

For tutorials see https://github.com/BachiLi/redner/wiki
For the theory of redner see https://people.csail.mit.edu/tzumao/diffrt/ and Tzu-Mao Li's PhD thesis.

CHAPTER

ONE

PYREDNER

class pyredner.**AmbientLight**(*intensity: torch.Tensor*)

Ambient light for deferred rendering.

render(*self*, *position: torch.Tensor*, *normal: torch.Tensor*, *albedo: torch.Tensor*)

class pyredner.**AreaLight**(*shape_id: int*, *intensity: torch.Tensor*, *two_sided: bool = False*, *directly_visible: bool = True*)

A mesh-based area light that points to a shape and assigns intensity.

Parameters

- **shape_id** (*int*) –
- **intensity** (*torch.Tensor*) – 1-d tensor with size 3 and type float32
- **two_sided** (*bool*) – Is the light emitting light from the two sides of the faces?
- **directly_visible** (*bool*) – Can the camera see the light source directly?

classmethod **load_state_dict**(*cls*, *state_dict*)

state_dict(*self*)

class pyredner.**Camera**(*position: Optional[torch.Tensor] = None*, *look_at: Optional[torch.Tensor] = None*, *up: Optional[torch.Tensor] = None*, *fov: Optional[torch.Tensor] = None*, *clip_near: float = 0.0001*, *resolution: Tuple[int, int] = (256, 256)*, *viewport: Optional[Tuple[int, int, int, int]] = None*, *cam_to_world: Optional[torch.Tensor] = None*, *intrinsic_mat: Optional[torch.Tensor] = None*, *distortion_params: Optional[torch.Tensor] = None*, *camera_type=pyredner.camera_type.perspective*, *fisheye: bool = False*)

Redner supports four types of cameras: perspective, orthographic, fisheye, and panorama. The camera takes a look at transform or a cam_to_world matrix to transform from camera local space to world space. It also can optionally take an intrinsic matrix that models field of view and camera skew.

Parameters

- **position** (*Optional[torch.Tensor]*) – the origin of the camera, 1-d tensor with size 3 and type float32
- **look_at** (*Optional[torch.Tensor]*) – the point camera is looking at, 1-d tensor with size 3 and type float32
- **up** (*Optional[torch.Tensor]*) – the up vector of the camera, 1-d tensor with size 3 and type float32
- **fov** (*Optional[torch.Tensor]*) – the field of view of the camera in angle, no effect if the camera is a fisheye or panorama camera, 1-d tensor with size 1 and type float32

- **clip_near** (*float*) – the near clipping plane of the camera, need to > 0
- **resolution** (*Tuple[int, int]*) – the size of the output image in (height, width)
- **viewport** (*Optional[Tuple[int, int, int, int]]*) – optional viewport argument for rendering only a region of an image in (left_top_y, left_top_x, bottom_right_y, bottom_right_x), bottom_right is not inclusive. if set to None the viewport is the whole image (i.e., (0, 0, cam.height, cam.width))
- **cam_to_world** (*Optional[torch.Tensor]*) – overrides position, look_at, up vectors 4x4 matrix, optional
- **intrinsic_mat** (*Optional[torch.Tensor]*) – a matrix that transforms a point in camera space before the point is projected to 2D screen space used for modelling field of view and camera skewing after the multiplication the point should be in [-1, 1/aspect_ratio] x [1, -1/aspect_ratio] in homogeneous coordinates the projection is then carried by the specific camera types perspective camera normalizes the homogeneous coordinates while orthogonal camera drop the Z coordinate. ignored by fisheye or panorama cameras overrides fov 3x3 matrix, optional
- **distortion_params** (*Optional[torch.Tensor]*) – an array describing the coefficient of a Brown–Conrady lens distortion model. the array is expected to be 1D with size of 8. the first six coefficients describes the parameters of the rational polynomial for radial distortion (k1~k6) and the last two coefficients are for the tangential distortion (p1~p2). see https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html for more details.
- **camera_type** (*render.camera_type*) – the type of the camera (perspective, orthographic, fisheye, or panorama)
- **fisheye** (*bool*) – whether the camera is a fisheye camera (legacy parameter just to ensure compatibility).

property cam_to_world(*self*)

property fov(*self*)

property intrinsic_mat(*self*)

classmethod load_state_dict(*cls*, *state_dict*)

state_dict(*self*)

class pyredner.**Context**

class pyredner.**DeferredLight**

class pyredner.**DirectionalLight**(*direction: torch.Tensor*, *intensity: torch.Tensor*)

Directional light for deferred rendering.

render(*self*, *position: torch.Tensor*, *normal: torch.Tensor*, *albedo: torch.Tensor*)

class pyredner.**EnvironmentMap**(*values: Union[torch.Tensor, pyredner.Texture]*, *env_to_world: torch.Tensor = torch.eye(4, 4)*, *directly_visible: bool = True*)

A class representing light sources infinitely far away using an image.

Parameters

- **values** (*Union[torch.Tensor, pyredner.Texture]*) – a float32 tensor with size 3 or [height, width, 3] or a Texture

- **env_to_world** (*torch.Tensor*) – a float32 4x4 matrix that transforms the environment map
- **directly_visible** (*bool*) – can the camera sees the light source directly?

property env_to_world(*self*)

generate_envmap_pdf(*self*)

classmethod load_state_dict(*cls*, *state_dict*)

state_dict(*self*)

property values(*self*)

class pyredner.Material(*diffuse_reflectance: Optional[Union[torch.Tensor, pyredner.Texture]] = None*, *specular_reflectance: Optional[Union[torch.Tensor, pyredner.Texture]] = None*, *roughness: Optional[Union[torch.Tensor, pyredner.Texture]] = None*, *generic_texture: Optional[Union[torch.Tensor, pyredner.Texture]] = None*, *normal_map: Optional[Union[torch.Tensor, pyredner.Texture]] = None*, *two_sided: bool = False*, *use_vertex_color: bool = False*)

redner currently employs a two-layer diffuse-specular material model. More specifically, it is a linear blend between a Lambertian model and a microfacet model with Phong distribution, with Schilick's Fresnel approximation. It takes either constant color or 2D textures for the reflectances and roughness, and an optional normal map texture. It can also use vertex color stored in the Shape. In this case the model fallback to a diffuse model.

Parameters

- **diffuse_reflectance** (*Optional[Union[torch.Tensor, pyredner.Texture]]*) – A float32 tensor with size 3 or [height, width, 3] or a Texture. Optional if use_vertex_color is True.
- **specular_reflectance** (*Optional[Union[torch.Tensor, pyredner.Texture]]*) – A float32 tensor with size 3 or [height, width, 3] or a Texture.
- **roughness** (*Optional[Union[torch.Tensor, pyredner.Texture]]*) – A float32 tensor with size 1 or [height, width, 1] or a Texture.
- **generic_texture** (*Optional[Union[torch.Tensor, pyredner.Texture]]*) – A float32 tensor with dimension 1 or 3, arbitrary number of channels use render_g_buffer to visualize this texture.
- **normal_map** (*Optional[Union[torch.Tensor, pyredner.Texture]]*) – A float32 tensor with size 3 or [height, width, 3] or a Texture.
- **two_sided** (*bool*) – By default, the material only reflect lights on the side the normal is pointing to. Set this to True to make the material reflects from both sides.
- **use_vertex_color** (*bool*) – Ignores the reflectances and use the vertex color as diffuse color

classmethod load_state_dict(*cls*, *state_dict*)

property specular_reflectance(*self*)

state_dict(*self*)

class pyredner.Object(*vertices: torch.Tensor*, *indices: torch.Tensor*, *material: pyredner.Material*, *light_intensity: Optional[torch.Tensor] = None*, *light_two_sided: bool = False*, *uvs: Optional[torch.Tensor] = None*, *normals: Optional[torch.Tensor] = None*, *uv_indices: Optional[torch.Tensor] = None*, *normal_indices: Optional[torch.Tensor] = None*, *colors: Optional[torch.Tensor] = None*, *directly_visible: bool = True*)

Object combines geometry, material, and lighting information and aggregate them in a single class. This is a convinent class for constructing redner scenes.

redner supports only triangle meshes for now. It stores a pool of vertices and access the pool using integer index. Some times the two vertices can have the same 3D position but different texture coordinates, because UV mapping creates seams and need to duplicate vertices. In this can we can use an additional "uv_indices" array to access the uv pool.

Parameters

- **vertices** (*torch.Tensor*) – 3D position of vertices float32 tensor with size num_vertices x 3
- **indices** (*torch.Tensor*) – vertex indices of triangle faces. int32 tensor with size num_triangles x 3
- **material** (pyredner.Material) –
- **light_intensity** (*Optional[torch.Tensor]*) – make this object an area light float32 tensor with size 3
- **light_two_sided** (*boolean*) – Does the light emit from two sides of the shape?
- **uvs** (*Optional[torch.Tensor]:*) – optional texture coordinates. float32 tensor with size num_uvs x 2 doesn't need to be the same size with vertices if uv_indices is None
- **normals** (*Optional[torch.Tensor]*) – shading normal float32 tensor with size num_normals x 3 doesn't need to be the same size with vertices if normal_indices is None
- **uv_indices** (*Optional[torch.Tensor]*) – overrides indices when accessing uv coordinates int32 tensor with size num_uvs x 3
- **normal_indices** (*Optional[torch.Tensor]*) – overrides indices when accessing shading normals int32 tensor with size num_normals x 3
- **colors** (*Optional[torch.Tensor]*) – optional per-vertex color float32 tensor with size num_vertices x 3
- **directly_visible** (*boolean*) – optional setting to see if object is visible to camera during rendering.

`class pyredner.`**`PointLight`**(*position: torch.Tensor*, *intensity: torch.Tensor*)

Point light with squared distance falloff for deferred rendering.

`render`(*self*, *position: torch.Tensor*, *normal: torch.Tensor*, *albedo: torch.Tensor*)

`class pyredner.`**`RenderFunction`**

The PyTorch interface of C++ redner.

`static` **`backward`**(*ctx*, *grad_img*)

`static` **`create_gradient_buffers`**(*ctx*)

`static` **`forward`**(*ctx*, *seed*, **args*)

Forward rendering pass: given a serialized scene and output an image.

`static` **`serialize_scene`**(*scene:* pyredner.Scene, *num_samples: Union[int, Tuple[int, int]]*, *max_bounces: int*, *channels: List = [redner.channels.radiance]*, *sampler_type=redner.SamplerType.independent*, *use_primary_edge_sampling: bool = True*, *use_secondary_edge_sampling: bool = True*, *sample_pixel_center: bool = False*, *device: Optional[torch.device] = None*)

Given a pyredner scene & rendering options, convert them to a linear list of argument, so that we can use it in PyTorch.

Parameters

- **scene** (pyredner.Scene) –
- **num_samples** (*int*) – Number of samples per pixel for forward and backward passes. Can be an integer or a tuple of 2 integers. If a single integer is provided, use the same number of samples for both.
- **max_bounces** (*int*) – Number of bounces for global illumination, 1 means direct lighting only.
- **channels** (*List[redner.channels]*) –

  A list of channels that should present in the output image
  following channels are supported:
  redner.channels.radiance,
  redner.channels.alpha,
  redner.channels.depth,
  redner.channels.position,
  redner.channels.geometry_normal,
  redner.channels.shading_normal,
  redner.channels.uv,
  redner.channels.barycentric_coordinates,
  redner.channels.diffuse_reflectance,
  redner.channels.specular_reflectance,
  redner.channels.vertex_color,
  redner.channels.roughness,
  redner.channels.generic_texture,
  redner.channels.shape_id,
  redner.channels.triangle_id,
  redner.channels.material_id
  all channels, except for shape id, triangle id, and material id, are differentiable
- **sampler_type** (*redner.SamplerType*) –

  Which sampling pattern to use?
  see *Chapter 7 of the PBRT book* <*http://www.pbr-book.org/3ed-2018/Sampling_and_Reconstruction.html*> for an explanation of the difference between different samplers.
  Following samplers are supported:
  redner.SamplerType.independent
  redner.SamplerType.sobol
- **use_primary_edge_sampling** (*bool*) –
- **use_secondary_edge_sampling** (*bool*) –
- **sample_pixel_center** (*bool*) – Always sample at the pixel center when rendering. This trades noise with aliasing. If this option is activated, the rendering becomes non-differentiable (since there is no antialiasing integral), and redner's edge sampling becomes an approximation to the gradients of the aliased rendering.

- **device** (*Optional[torch.device]*) – Which device should we store the data in. If set to None, use the device from pyredner.get_device().

static unpack_args(*seed*, *args*, *use_primary_edge_sampling=None*, *use_secondary_edge_sampling=None*)

Given a list of serialized scene arguments, unpack all information into a Context.

static visualize_screen_gradient(*grad_img: torch.Tensor*, *seed: int*, *scene: pyredner.Scene*, *num_samples: Union[int, Tuple[int, int]]*, *max_bounces: int*, *channels: List = [redner.channels.radiance]*, *sampler_type=redner.SamplerType.independent*, *use_primary_edge_sampling: bool = True*, *use_secondary_edge_sampling: bool = True*, *sample_pixel_center: bool = False*)

Given a serialized scene and output an 2-channel image, which visualizes the derivatives of pixel color with respect to the screen space coordinates.

Parameters

- **grad_img** (*Optional[torch.Tensor]*) – The "adjoint" of the backpropagation gradient. If you don't know what this means just give None
- **seed** (*int*) – seed for the Monte Carlo random samplers
- **arguments.** (*See serialize_scene for the explanation of the rest of the*) –

pyredner.**SH**(*l*, *m*, *theta*, *phi*)

pyredner.**SH_reconstruct**(*coeffs*, *res*)

pyredner.**SH_renormalization**(*l*, *m*)

class pyredner.**Scene**(*camera: pyredner.Camera*, *shapes: List[pyredner.Shape] = []*, *materials: List[pyredner.Material] = []*, *area_lights: List[pyredner.AreaLight] = []*, *objects: Optional[List[pyredner.Object]] = None*, *envmap: Optional[pyredner.EnvironmentMap] = None*)

A scene is a collection of camera, geometry, materials, and light. Currently there are two ways to construct a scene: one is through lists of Shape, Material, and AreaLight. The other one is through a list of Object. It is more recommended to use the Object construction. The Shape/Material/AreaLight options are here for legacy issue.

Parameters

- **shapes** (*List[pyredner.Shape] = [],*) –
- **materials** (*List[pyredner.Material] = [],*) –
- **area_lights** (*List[pyredner.AreaLight] = [],*) –
- **objects** (*Optional[List[pyredner.Object]] = None,*) –
- **envmap** (*Optional[pyredner.EnvironmentMap] = None*) –

classmethod load_state_dict(*cls*, *state_dict*)

state_dict(*self*)

class pyredner.**Shape**(*vertices: torch.Tensor*, *indices: torch.Tensor*, *material_id: int*, *uvs: Optional[torch.Tensor] = None*, *normals: Optional[torch.Tensor] = None*, *uv_indices: Optional[torch.Tensor] = None*, *normal_indices: Optional[torch.Tensor] = None*, *colors: Optional[torch.Tensor] = None*) redner supports only triangle meshes for now. It stores a pool of vertices and access the pool using integer index. Some times the two vertices can have the same 3D position but different texture coordinates, because UV mapping creates seams and need to duplicate vertices. In this can we can use an additional "uv_indices" array to access the uv pool. :param vertices: 3D position of vertices

float32 tensor with size num_vertices x 3

Parameters

- **indices** (*torch.Tensor*) – vertex indices of triangle faces. int32 tensor with size num_triangles x 3
- **uvs** (*Optional[torch.Tensor]:*) – optional texture coordinates. float32 tensor with size num_uvs x 2 doesn't need to be the same size with vertices if uv_indices is not None
- **normals** (*Optional[torch.Tensor]*) – shading normal float32 tensor with size num_normals x 3 doesn't need to be the same size with vertices if normal_indices is not None
- **uv_indices** (*Optional[torch.Tensor]*) – overrides indices when accessing uv coordinates int32 tensor with size num_uvs x 3
- **normal_indices** (*Optional[torch.Tensor]*) – overrides indices when accessing shading normals int32 tensor with size num_normals x 3

`classmethod` **load_state_dict**(*cls*, *state_dict*)

state_dict(*self*)

`class pyredner.`**SpotLight**(*position: torch.Tensor*, *spot_direction: torch.Tensor*, *spot_exponent: torch.Tensor*, *intensity: torch.Tensor*)

Spot light with cosine falloff for deferred rendering. Note that we do not provide the cosine cutoff here since it is not differentiable.

render(*self*, *position: torch.Tensor*, *normal: torch.Tensor*, *albedo: torch.Tensor*)

`class pyredner.`**Texture**(*texels: torch.Tensor*, *uv_scale: Optional[torch.Tensor] = None*)

Representing a texture and its mipmap.

Parameters

- **texels** (*torch.Tensor*) – a float32 tensor with size C or [height, width, C]
- **uv_scale** (*Optional[torch.Tensor]*) – scale the uv coordinates when mapping the texture a float32 tensor with size 2

`property` **device**(*self*)

generate_mipmap(*self*)

`classmethod` **load_state_dict**(*cls*, *state_dict*)

state_dict(*self*)

`property` **texels**(*self*)

`pyredner.`**associated_legendre_polynomial**(*l*, *m*, *x*) `pyredner.`**`automatic_camera_placement`**(*shapes: List*, *resolution: Tuple[int, int]*)

Given a list of objects or shapes, generates camera parameters automatically using the bounding boxes of the shapes. Place the camera at some distances from the shapes, so that it can see all of them. Inspired by https://github.com/mitsuba-renderer/mitsuba/blob/master/src/librender/scene.cpp#L286

Parameters

- **`shapes`** (*`List`*) – a list of redner Shape or Object
- **`resolution`** (*`Tuple[int, int]`*) – the size of the output image in (height, width)

Returns a camera that can see all the objects.

Return type *pyredner.Camera*

`pyredner.`**`bound_vertices`**(*vertices: torch.Tensor*, *indices: torch.Tensor*)

Calculate the indices of boundary vertices of a mesh and express it in Tensor form.

Parameters

- **`vertices`** (*`torch.Tensor`*) – 3D position of vertices. float32 tensor with size num_vertices x 3
- **`indices`** (*`torch.Tensor`*) – Vertex indices of triangle faces. int32 tensor with size num_triangles x 3

Returns **bound** – float32 Tensor with size num_vertices representing vertex normal bound[i] = 0. if i-th vertices is on boundary of mesh; else 1.

Return type torch.Tensor

`pyredner.`**`camera_type`**

`pyredner.`**`channels`**

`pyredner.`**`compute_uvs`**(*vertices*, *indices*, *print_progress=True*)

Compute UV coordinates of a given mesh using a charting algorithm with least square conformal mapping. This calls the xatlas library. :param vertices: 3D position of vertices

float32 tensor with size num_vertices x 3

Parameters **`indices`** (*`torch.Tensor`*) – vertex indices of triangle faces. int32 tensor with size num_triangles x 3

Returns

- *torch.Tensor* – uv vertices pool, float32 Tensor with size num_uv_vertices x 3
- *torch.Tensor* – uv indices, int32 Tensor with size num_triangles x 3

`pyredner.`**`compute_vertex_normal`**(*vertices: torch.Tensor*, *indices: torch.Tensor*, *weighting_scheme: str = 'max'*)

Compute vertex normal by weighted average of nearby face normals. :param vertices: 3D position of vertices.

float32 tensor with size num_vertices x 3

Parameters

- **`indices`** (*`torch.Tensor`*) – Vertex indices of triangle faces. int32 tensor with size num_triangles x 3

- **weighting_scheme** (*str*) – How do we compute the weighting. Currently we support two weighting methods: 'max' and 'cotangent'. 'max' corresponds to Nelson Max's algorithm that uses the inverse length and sine of the angle as the weight (see Weights for Computing Vertex Normals from Facet Vectors), 'cotangent' corresponds to weights derived through a discretization of the gradient of triangle area (see, e.g., "Implicit Fairing of Irregular Meshes using Diffusion and Curvature Flow" from Desbrun et al.)

Returns float32 Tensor with size num_vertices x 3 representing vertex normal

Return type torch.Tensor

pyredner.**device**

pyredner.**gen_look_at_matrix**(*pos*, *look*, *up*)

pyredner.**gen_perspective_matrix**(*fov*, *clip_near*, *clip_far*)

pyredner.**gen_rotate_matrix**(*angles: torch.Tensor*)

Given a 3D Euler angle vector, outputs a rotation matrix.

Parameters **angles** (*torch.Tensor*) – 3D Euler angle

Returns 3x3 rotation matrix

Return type torch.Tensor

pyredner.**gen_scale_matrix**(*scale*)

pyredner.**gen_translate_matrix**(*translate*)

pyredner.**generate_geometry_image**(*size: int*, *device: Optional[torch.device] = None*)

Generate an spherical geometry image [Gu et al. 2002 and Praun and Hoppe 2003] of size [2 * size + 1, 2 * size + 1]. This can be used for encoding a genus-0 surface into a regular image, so that it is more convienent for a CNN to process. The topology is given by a tesselated octahedron. UV is given by the spherical mapping. Duplicated vertex are mapped to the one with smaller index (so some vertices on the geometry image is unused by the indices).

Parameters

- **size** (*int*) – Size of the geometry image.
- **device** (*Optional[torch.device]*) – Which device should we store the data in. If set to None, use the device from pyredner.get_device().

Returns

- *torch.Tensor* – vertices of size [(2 * size + 1 * 2 * size + 1), 3]
- *torch.Tensor* – indices of size [2 * (2 * size + 1 * 2 * size + 1), 3]
- *torch.Tensor* – uvs of size [(2 * size + 1 * 2 * size + 1), 2]

pyredner.**generate_intrinsic_mat**(*fx: torch.Tensor*, *fy: torch.Tensor*, *skew: torch.Tensor*, *x0: torch.Tensor*, *y0: torch.Tensor*)

Generate the following 3x3 intrinsic matrix given the parameters.

fx, skew, x0

0, fy, y0

0, 0, 1

Parameters

- **fx** (*torch.Tensor*) – Focal length at x dimension. 1D tensor with size 1.
- **fy** (*torch.Tensor*) – Focal length at y dimension. 1D tensor with size 1.
- **skew** (*torch.Tensor*) – Axis skew parameter describing shearing transform. 1D tensor with size 1.
- **x0** (*torch.Tensor*) – Principle point offset at x dimension. 1D tensor with size 1.
- **y0** (*torch.Tensor*) – Principle point offset at y dimension. 1D tensor with size 1.

Returns 3x3 intrinsic matrix

Return type torch.Tensor

`pyredner.`**`generate_quad_light`**(*position: torch.Tensor*, *look_at: torch.Tensor*, *size: torch.Tensor*, *intensity: torch.Tensor*, *directly_visible: bool = True*)

Generate a pyredner.Object that is a quad light source.

Parameters

- **position** (*torch.Tensor*) – 1-d tensor of size 3
- **look_at** (*torch.Tensor*) – 1-d tensor of size 3
- **size** (*torch.Tensor*) – 1-d tensor of size 2
- **intensity** (*torch.Tensor*) – 1-d tensor of size 3
- **directly_visible** (*boolean*) – Can the camera see the light source directly?

Returns quad light source

Return type *pyredner.Object*

`pyredner.`**`generate_sphere`**(*theta_steps: int*, *phi_steps: int*, *device: Optional[torch.device] = None*)

Generate a triangle mesh representing a UV sphere, center at (0, 0, 0) with radius 1.

Parameters

- **theta_steps** (*int*) – zenith subdivision
- **phi_steps** (*int*) – azimuth subdivision
- **device** (*Optional[torch.device]*) – Which device should we store the data in. If set to None, use the device from pyredner.get_device().

Returns

- *torch.Tensor* – vertices
- *torch.Tensor* – indices
- *torch.Tensor* – uvs
- *torch.Tensor* – normals

`pyredner.`**`get_device`**()

Get the torch device we are using.

`pyredner.`**`get_print_timing`**()

Get whether we print time measurements or not.

`pyredner.`**`get_use_correlated_random_number`**()

See set_use_correlated_random_number

`pyredner.`**`get_use_gpu`**`()`

Get whether we are using CUDA or not.

`pyredner.`**`imread`**`(`*filename: str*, *gamma: float = 2.2*`)`

read img from filename

Parameters

- **`filename`** (*`str`*) –
- **`gamma`** (*`float`*) – if the image is not an OpenEXR file, apply gamma correction

Returns a float32 tensor with size [height, width, channel]

Return type torch.Tensor

`pyredner.`**`imwrite`**`(`*img: torch.Tensor*, *filename: str*, *gamma: float = 2.2*, *normalize: bool = False*`)`

write img to filename

Parameters

- **`img`** (*`torch.Tensor`*) – with size [height, width, channel]
- **`filename`** (*`str`*) –
- **`gamma`** (*`float`*) – if the image is not an OpenEXR file, apply gamma correction
- **`normalize`** – normalize img to the range [0, 1] before writing

`pyredner.`**`linear_to_srgb`**`(`*x*`)`

`pyredner.`**`load_mitsuba`**`(`*filename: str*, *device: Optional[torch.device] = None*`)`

Load from a Mitsuba scene file as PyTorch tensors.

Parameters

- **`filename`** (*`str`*) – Path to the Mitsuba scene file.
- **`device`** (*`Optional[torch.device]`*) – Which device should we store the data in. If set to None, use the device from pyredner.get_device().

Return type *pyredner.Scene*

`pyredner.`**`load_obj`**`(`*filename: str*, *obj_group: bool = True*, *flip_tex_coords: bool = True*, *use_common_indices: bool = False*, *return_objects: bool = False*, *device: Optional[torch.device] = None*`)`

Load from a Wavefront obj file as PyTorch tensors.

Parameters

- **`filename`** (*`str`*) – Path to the obj file.
- **`obj_group`** (*`bool`*) – Split the meshes based on materials.
- **`flip_tex_coords`** (*`bool`*) – Flip the v coordinate of uv by applying v' = 1 - v.
- **`use_common_indices`** (*`bool`*) – Use the same indices for position, uvs, normals. Not recommended since texture seams in the objects sharing. The same positions would cause the optimization to "tear" the object.
- **`return_objects`** (*`bool`*) – Output list of Object instead. If there is no corresponding material for a shape, assign a grey material.
- **`device`** (*`Optional[torch.device]`*) – Which device should we store the data in. If set to None, use the device from pyredner.get_device().

Returns

- *if return_objects == True, return a list of Object*
- *if return_objects == False, return (material_map, mesh_list, light_map),*
- *material_map -> Map[mtl_name, WavefrontMaterial]*
- *mesh_list -> List[TriangleMesh]*
- *light_map -> Map[mtl_name, torch.Tensor]*

`pyredner.`**`normalize`**`(`*v*`)`

`pyredner.`**`print_timing`**` = `**`True`**

`pyredner.`**`radians`**`(`*deg*`)`

`pyredner.`**`render_albedo`**`(`*scene: Union[*pyredner.Scene*, List[*pyredner.Scene*]], alpha: bool = False, num_samples: Union[int, Tuple[int, int]] = (16, 4), seed: Optional[Union[int, List[int], Tuple[int, int], List[Tuple[int, int]]]] = None, sample_pixel_center: bool = False, use_primary_edge_sampling: bool = True, device: Optional[torch.device] = None*`)`

Render the diffuse albedo colors of the scenes.

Parameters

- **scene** (*Union[pyredner.Scene, List[pyredner.Scene]]*) – pyredner Scene containing camera, geometry and material. Can be a single scene or a list for batch render. For batch rendering all scenes need to have the same resolution.
- **alpha** (*bool*) – If set to False, generates a 3-channel image, otherwise generates a 4-channel image where the fourth channel is alpha.
- **num_samples** (*Union[int, Tuple[int, int]]*) – number of samples for forward and backward passes, respectively if a single integer is provided, use the same number of samples for both
- **seed** (*Optional[Union[int, List[int]]]*) – Random seed used for sampling. Randomly assigned if set to None. For batch render, if seed it not None, need to provide a list of seeds.
- **sample_pixel_center** (*bool*) – Always sample at the pixel center when rendering. This trades noise with aliasing. If this option is activated, the rendering becomes non-differentiable (since there is no antialiasing integral), and redner's edge sampling becomes an approximation to the gradients of the aliased rendering.
- **device** (*Optional[torch.device]*) – Which device should we store the data in. If set to None, use the device from pyredner.get_device().

Returns

if input scene is a list: a tensor with size [N, H, W, C], N is the list size
else: a tensor with size [H, W, C]
if alpha == True, C = 4.
else, C = 3.

Return type torch.Tensor or List[torch.Tensor] `pyredner.`**`render_deferred`**(*scene: Union[pyredner.Scene, List[pyredner.Scene]], lights: Union[List[DeferredLight], List[List[DeferredLight]]], alpha: bool = False, aa_samples: int = 2, seed: Optional[Union[int, List[int], Tuple[int, int], List[Tuple[int, int]]]] = None, sample_pixel_center: bool = False, use_primary_edge_sampling: bool = True, device: Optional[torch.device] = None*)

Render the scenes using deferred rendering. We generate G-buffer images containing world-space position, normal, and albedo using redner, then shade the G-buffer using PyTorch code. Assuming Lambertian shading and does not compute shadow.

Parameters

- **scene** (*Union[*`pyredner.Scene`*, List[*`pyredner.Scene`*]]*) – pyredner Scene containing camera, geometry and material. Can be a single scene or a list for batch render. For batch rendering all scenes need to have the same resolution.
- **lights** (*Union[List[*`DeferredLight`*], List[List[*`DeferredLight`*]]]*) – Lights for deferred rendering. If the scene is a list, and only a single list of lights is provided, the same lights are applied to all scenes. If a list of lists of lights is provided, each scene is lit by the corresponding lights.
- **alpha** (*bool*) – If set to False, generates a 3-channel image, otherwise generates a 4-channel image where the fourth channel is alpha.
- **aa_samples** (*int*) – Number of samples used for anti-aliasing at both x, y dimensions (e.g. if aa_samples=2, 4 samples are used).
- **seed** (*Optional[Union[int, List[int], Tuple[int, int], List[Tuple[int, int]]]]*) – Random seed used for sampling. Randomly assigned if set to None. For batch render, if seed it not None, need to provide a list of seeds.
- **sample_pixel_center** (*bool*) – Always sample at the pixel center when rendering. This trades noise with aliasing. If this option is activated, the rendering becomes non-differentiable (since there is no antialiasing integral), and redner's edge sampling becomes an approximation to the gradients of the aliased rendering.
- **use_primary_edge_sampling** (*bool*) – debug option
- **device** (*Optional[torch.device]*) – Which device should we store the data in. If set to None, use the device from pyredner.get_device().

Returns

if input scene is a list: a tensor with size [N, H, W, C], N is the list size
else: a tensor with size [H, W, C]
if alpha == True, C = 4.
else, C = 3.

Return type torch.Tensor or List[torch.Tensor]

`pyredner.`**`render_g_buffer`**(*scene: Union[pyredner.Scene, List[pyredner.Scene]], channels: List, num_samples: Union[int, Tuple[int, int]] = (1, 1), seed: Optional[Union[int, List[int], Tuple[int, int], List[Tuple[int, int]]]] = None, sample_pixel_center: bool = False, use_primary_edge_sampling: bool = True, use_secondary_edge_sampling: bool = True, device: Optional[torch.device] = None*)

Render G buffers from the scene.

Parameters

- **scene** (*Union[pyredner.Scene, List[pyredner.Scene]]*) – pyredner Scene containing camera, geometry and material. Can be a single scene or a list for batch render. For batch rendering all scenes need to have the same resolution.
- **channels** (*List[pyredner.channels]*) –

  A list of the following channels:
  pyredner.channels.radiance,
  pyredner.channels.alpha
  pyredner.channels.depth
  pyredner.channels.position
  pyredner.channels.geometry_normal
  pyredner.channels.shading_normal
  pyredner.channels.uv
  pyredner.channels.barycentric_coordinates
  pyredner.channels.diffuse_reflectance
  pyredner.channels.specular_reflectance
  pyredner.channels.roughness
  pyredner.channels.generic_texture
  pyredner.channels.vertex_color
  pyredner.channels.shape_id
  pyredner.channels.triangle_id
  pyredner.channels.material_id
- **num_samples** (*Union[int, Tuple[int, int]]*) – Number of samples for forward and backward passes, respectively. If a single integer is provided, use the same number of samples for both.
- **seed** (*Optional[Union[int, List[int], Tuple[int, int], List[Tuple[int, int]]]]*) – Random seed used for sampling. Randomly assigned if set to None. For batch render, if seed it not None, need to provide a list of seeds.
- **sample_pixel_center** (*bool*) – Always sample at the pixel center when rendering. This trades noise with aliasing. If this option is activated, the rendering becomes non-differentiable (since there is no antialiasing integral), and redner's edge sampling becomes an approximation to the gradients of the aliased rendering.
- **use_primary_edge_sampling** (*bool*) – debug option
- **use_secondary_edge_sampling** (*bool*) – debug option
- **device** (*Optional[torch.device]*) – Which device should we store the data in. If set to None, use the device from pyredner.get_device().

Returns

if input scene is a list: a tensor with size [N, H, W, C], N is the list size
else: a tensor with size [H, W, C]

Return type torch.Tensor or List[torch.Tensor] `pyredner.render_generic`(*scene: pyredner.Scene*, *channels: List*, *max_bounces: int = 1*, *sampler_type=pyredner.sampler_type.sobol*, *num_samples: Union[int, Tuple[int, int]] = (4, 4)*, *seed: Optional[Union[int, List[int], Tuple[int, int], List[Tuple[int, int]]]] = None*, *sample_pixel_center: bool = False*, *use_primary_edge_sampling: bool = True*, *use_secondary_edge_sampling: bool = True*, *device: Optional[torch.device] = None*)

A generic rendering function that can be either pathtracing or g-buffer rendering or both.

Parameters

- **scene** (*Union[pyredner.Scene, List[pyredner.Scene]]*) – pyredner Scene containing camera, geometry and material. Can be a single scene or a list for batch render. For batch rendering all scenes need to have the same resolution.
- **channels** (*List[pyredner.channels]*) –

  A list of the following channels:

  pyredner.channels.radiance,

  pyredner.channels.alpha

  pyredner.channels.depth

  pyredner.channels.position

  pyredner.channels.geometry_normal

  pyredner.channels.shading_normal

  pyredner.channels.uv

  pyredner.channels.barycentric_coordinates

  pyredner.channels.diffuse_reflectance

  pyredner.channels.specular_reflectance

  pyredner.channels.roughness

  pyredner.channels.generic_texture

  pyredner.channels.vertex_color

  pyredner.channels.shape_id

  pyredner.channels.triangle_id

  pyredner.channels.material_id
- **max_bounces** (*int*) – Number of bounces for global illumination, 1 means direct lighting only.
- **sampler_type** (*pyredner.sampler_type*) –

  Which sampling pattern to use? See Chapter 7 of the PBRT book for an explanation of the difference between different samplers.

  Following samplers are supported:

  pyredner.sampler_type.independent

  pyredner.sampler_type.sobol
- **num_samples** (*int*) – Number of samples per pixel for forward and backward passes. Can be an integer or a tuple of 2 integers.
- **seed** (*Optional[Union[int, List[int], Tuple[int, int], List[Tuple[int, int]]]]*) – Random seed used for sampling. Randomly assigned if set to None. For batch render, if seed it not None, need to provide a list of seeds.
- **sample_pixel_center** (*bool*) – Always sample at the pixel center when rendering. This trades noise with aliasing. If this option is activated, the rendering becomes non-differentiable (since there is no antialiasing integral), and redner's edge sampling becomes an approximation to the gradients of the aliased rendering.

- **use_primary_edge_sampling** (*bool*) – debug option
- **use_secondary_edge_sampling** (*bool*) – debug option
- **device** (*Optional[torch.device]*) – Which device should we store the data in. If set to None, use the device from pyredner.get_device().

Returns

if input scene is a list: a tensor with size [N, H, W, C], N is the list size

else: a tensor with size [H, W, C]

Return type torch.Tensor or List[torch.Tensor]

pyredner.**render_pathtracing**(*scene: Union[pyredner.Scene, List[pyredner.Scene]], alpha: bool = False, max_bounces: int = 1, sampler_type=pyredner.sampler_type.sobol, num_samples: Union[int, Tuple[int, int]] = (4, 4), seed: Optional[Union[int, List[int], Tuple[int, int], List[Tuple[int, int]]]] = None, sample_pixel_center: bool = False, use_primary_edge_sampling: bool = True, use_secondary_edge_sampling: bool = True, device: Optional[torch.device] = None*)

Render a pyredner scene using pathtracing.

Parameters

- **scene** (*Union[pyredner.Scene, List[pyredner.Scene]]*) – pyredner Scene containing camera, geometry and material. Can be a single scene or a list for batch render. For batch rendering all scenes need to have the same resolution.
- **max_bounces** (*int*) – Number of bounces for global illumination, 1 means direct lighting only.
- **sampler_type** (*pyredner.sampler_type*) –

  Which sampling pattern to use? See Chapter 7 of the PBRT book for an explanation of the difference between different samplers.

  Following samplers are supported:

  pyredner.sampler_type.independent

  pyredner.sampler_type.sobol
- **num_samples** (*int*) – Number of samples per pixel for forward and backward passes. Can be an integer or a tuple of 2 integers.
- **seed** (*Optional[Union[int, List[int], Tuple[int, int], List[Tuple[int, int]]]]*) – Random seed used for sampling. Randomly assigned if set to None. For batch render, if seed it not None, need to provide a list of seeds.
- **sample_pixel_center** (*bool*) – Always sample at the pixel center when rendering. This trades noise with aliasing. If this option is activated, the rendering becomes non-differentiable (since there is no antialiasing integral), and redner's edge sampling becomes an approximation to the gradients of the aliased rendering.
- **use_primary_edge_sampling** (*bool*) – debug option
- **use_secondary_edge_sampling** (*bool*) – debug option
- **device** (*Optional[torch.device]*) – Which device should we store the data in. If set to None, use the device from pyredner.get_device().

Returns

if input scene is a list: a tensor with size [N, H, W, C], N is the list size
else: a tensor with size [H, W, C]
if alpha == True, C = 4.
else, C = 3.

Return type torch.Tensor or List[torch.Tensor]

`pyredner.`**`sampler_type`**

`pyredner.`**`save_mtl`**(*m:* pyredner.Material, *filename: str*)

`pyredner.`**`save_obj`**(*shape: Union[*pyredner.Object, pyredner.Shape*]*, *filename: str*, *flip_tex_coords=True*)

Save to a Wavefront obj file from an Object or a Shape. :param shape: :type shape: Union[pyredner.Object, pyredner.Shape] :param filename: :type filename: str :param flip_tex_coords: flip the v coordinate of uv by applying v' = 1 - v :type flip_tex_coords: bool

`pyredner.`**`serialize_texture`**(*texture*, *args*, *device*)

`pyredner.`**`set_device`**(*d: torch.device*)

Set the torch device we are using.

`pyredner.`**`set_print_timing`**(*v: bool*)

Set whether to print time measurements or not.

`pyredner.`**`set_use_correlated_random_number`**(*v: bool*)

There is a bias-variance trade off in the backward pass.
If the forward pass and the backward pass are correlated
the gradients are biased for L2 loss.
E[d/dx(f(x) - y)^2] = E[(f(x) - y) d/dx f(x)]
= E[f(x) - y] E[d/dx f(x)]
The last equation only holds when f(x) and d/dx f(x) are independent.
It is usually better to use the unbiased one, but we left it as an option here

`pyredner.`**`set_use_gpu`**(*v: bool*)

Set whether to use CUDA or not.

`pyredner.`**`smooth`**(*vertices: torch.Tensor*, *indices: torch.Tensor*, *lmd: torch.float32*, *weighting_scheme: str = 'reciprocal'*, *control: torch.Tensor = None*)

Update positions of vertices in a mesh. The shift amount of a vertex equals to lmd times weight sum of all edges to neighbors.

$v_i += lmd *

rac {sum_{j in neighbors(i)} w_{ij}(v_j - v_i)} {sum_{j in neighbors(i)} w_{ij}}$

vertices: torch.Tensor 3D position of vertices. float32 tensor with size num_vertices x 3

indices: torch.Tensor Vertex indices of triangle faces. int32 tensor with size num_triangles x 3

lmd: torch.float32 step length coefficient

weighting_scheme: str = 'reciprocal'

Different weighting schemes:

'reciprocal': (default) w[i][j] = 1 / len(v[j] - v[i])

'uniform': w[i][j] = 1

'cotangent': w[i][j] = cot(angle(i-m-j)) + cot(angle(i-n-j)) m and n are vertices that form triangles with i and j

control: torch.Tensor extra coefficient deciding which vertices to be update. In default case, do not update boundary vertices of the mesh

control (default) = bound_vertices(vertices, indices)

type help(pyredner.bound_vertices)

`pyredner.`**`srgb_to_linear`**`(`*x*`)`

`pyredner.`**`use_correlated_random_number = False`**

`pyredner.`**`use_gpu`**

CHAPTER

TWO

PYREDNER_TENSORFLOW

class pyredner_tensorflow.**AmbientLight**(*intensity: tensorflow.Tensor*)

Ambient light for deferred rendering.

render(*self*, *position: tensorflow.Tensor*, *normal: tensorflow.Tensor*, *albedo: tensorflow.Tensor*)

class pyredner_tensorflow.**AreaLight**(*shape_id: int*, *intensity: tensorflow.Tensor*, *two_sided: bool = False*, *directly_visible: bool = True*)

A mesh-based area light that points to a shape and assigns intensity.

Parameters

- **shape_id** (*int*) –
- **intensity** (*tf.Tensor*) – 1-d tensor with size 3 and type float32
- **two_sided** (*bool*) – is the light emitting light from the two sides of the faces?
- **directly_visible** (*bool*) – can the camera sees the light source directly?

classmethod **load_state_dict**(*cls*, *state_dict*)

state_dict(*self*)

class pyredner_tensorflow.**Camera**(*position: Optional[tensorflow.Tensor] = None*, *look_at: Optional[tensorflow.Tensor] = None*, *up: Optional[tensorflow.Tensor] = None*, *fov: Optional[tensorflow.Tensor] = None*, *clip_near: float = 0.0001*, *resolution: Tuple[int] = (256, 256)*, *viewport: Optional[Tuple[int, int, int, int]] = None*, *cam_to_world: Optional[tensorflow.Tensor] = None*, *intrinsic_mat: Optional[tensorflow.Tensor] = None*, *distortion_params: Optional[tensorflow.Tensor] = None*, *camera_type=pyredner.camera_type.perspective*, *fisheye: bool = False*)

redner supports four types of cameras: perspective, orthographic, fisheye, and panorama. The camera takes a look at transform or a cam_to_world matrix to transform from camera local space to world space. It also can optionally take an intrinsic matrix that models field of view and camera skew.

Parameters

- **position** (*Optional[tf.Tensor]*) – the origin of the camera, 1-d tensor with size 3 and type float32
- **look_at** (*Optional[tf.Tensor]*) – the point camera is looking at, 1-d tensor with size 3 and type float32
- **up** (*Optional[tf.tensor]*) – the up vector of the camera, 1-d tensor with size 3 and type float32

- **fov** (*Optional[tf.Tensor]*) – the field of view of the camera in angle no effect if the camera is a fisheye or panorama camera 1-d tensor with size 1 and type float32
- **clip_near** (*float*) – the near clipping plane of the camera, need to > 0
- **resolution** (*Tuple[int, int]*) – the size of the output image in (height, width)
- **viewport** (*Optional[Tuple[int, int, int, int]]*) – optional viewport argument for rendering only a region of an image in (left_top_y, left_top_x, bottom_right_y, bottom_right_x), bottom_right is not inclusive. if set to None the viewport is the whole image (i.e., (0, 0, cam.height, cam.width))
- **cam_to_world** (*Optional[tf.Tensor]*) – overrides position, look_at, up vectors 4x4 matrix, optional
- **intrinsic_mat** (*Optional[tf.Tensor]*) – a matrix that transforms a point in camera space before the point is projected to 2D screen space used for modelling field of view and camera skewing after the multiplication the point should be in [-1, 1/aspect_ratio] x [1, -1/aspect_ratio] in homogeneous coordinates the projection is then carried by the specific camera types perspective camera normalizes the homogeneous coordinates while orthogonal camera drop the Z coordinate. ignored by fisheye or panorama cameras overrides fov 3x3 matrix, optional
- **distortion_params** (*Optional[tf.Tensor]*) – an array describing the coefficient of a Brown–Conrady lens distortion model. the array is expected to be 1D with size of 8. the first six coefficients describes the parameters of the rational polynomial for radial distortion (k1~k6) and the last two coefficients are for the tangential distortion (p1~p2). see https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html for more details.
- **camera_type** (*render.camera_type*) – the type of the camera (perspective, orthographic, fisheye, or panorama)
- **fisheye** (*bool*) – whether the camera is a fisheye camera (legacy parameter just to ensure compatibility).

property cam_to_world(*self*)

property fov(*self*)

property intrinsic_mat(*self*)

classmethod load_state_dict(*cls*, *state_dict*)

state_dict(*self*)

class pyredner_tensorflow.**CameraType**

class pyredner_tensorflow.**Channel**

class pyredner_tensorflow.**Context**

class pyredner_tensorflow.**DeferredLight**

class pyredner_tensorflow.**DirectionalLight**(*direction: tensorflow.Tensor*, *intensity: tensorflow.Tensor*)

Directional light for deferred rendering.

render(*self*, *position: tensorflow.Tensor*, *normal: tensorflow.Tensor*, *albedo: tensorflow.Tensor*) `class pyredner_tensorflow.EnvironmentMap`(*values: tensorflow.Tensor*, *env_to_world: tensorflow.Tensor = tf.eye(4, 4)*, *directly_visible: bool = True*)

A class representing light sources infinitely far away using an image.

Parameters

- **values** (*Union[tf.Tensor,* pyredner.Texture*]*) – a float32 tensor with size 3 or [height, width, 3] or a Texture
- **env_to_world** (*tf.Tensor*) – a float32 4x4 matrix that transforms the environment map
- **directly_visible** (*bool*) – can the camera sees the light source directly?

`property env_to_world`(*self*)

`generate_envmap_pdf`(*self*)

`classmethod load_state_dict`(*cls*, *state_dict*)

`state_dict`(*self*)

`property values`(*self*)

`class pyredner_tensorflow.Material`(*diffuse_reflectance: Optional[Union[tensorflow.Tensor,* pyredner_tensorflow.Texture*]] = None*, *specular_reflectance: Optional[Union[tensorflow.Tensor,* pyredner_tensorflow.Texture*]] = None*, *roughness: Optional[Union[tensorflow.Tensor,* pyredner_tensorflow.Texture*]] = None*, *generic_texture: Optional[Union[tensorflow.Tensor,* pyredner_tensorflow.Texture*]] = None*, *normal_map: Optional[Union[tensorflow.Tensor,* pyredner_tensorflow.Texture*]] = None*, *two_sided: bool = False*, *use_vertex_color: bool = False*)

redner currently employs a two-layer diffuse-specular material model. More specifically, it is a linear blend between a Lambertian model and a microfacet model with Phong distribution, with Schilick's Fresnel approximation. It takes either constant color or 2D textures for the reflectances and roughness, and an optional normal map texture. It can also use vertex color stored in the Shape. In this case the model fallback to a diffuse model.

Parameters

- **diffuse_reflectance** (*Optional[Union[tf.Tensor,* pyredner.Texture*]]*) – a float32 tensor with size 3 or [height, width, 3] or a Texture optional if use_vertex_color is True
- **specular_reflectance** (*Optional[Union[tf.Tensor,* pyredner.Texture*]]*) – a float32 tensor with size 3 or [height, width, 3] or a Texture
- **roughness** (*Optional[Union[tf.Tensor,* pyredner.Texture*]]*) – a float32 tensor with size 1 or [height, width, 1] or a Texture
- **generic_texture** (*Optional[Union[tf.Tensor,* pyredner.Texture*]]*) – a float32 tensor with dimension 1 or 3, arbitrary number of channels use render_g_buffer to visualize this texture
- **normal_map** (*Optional[Union[tf.Tensor,* pyredner.Texture*]]*) – a float32 tensor with size 3 or [height, width, 3] or a Texture
- **two_sided** (*bool*) – By default, the material only reflect lights on the side the normal is pointing to. Set this to True to make the material reflects from both sides.
- **use_vertex_color** (*bool*) – ignores the reflectances and use the vertex color as diffuse color

classmethod load_state_dict(*cls*, *state_dict*)

property specular_reflectance(*self*)

state_dict(*self*)

class pyredner_tensorflow.**Object**(*vertices: tensorflow.Tensor*, *indices: tensorflow.Tensor*, *material:* pyredner_tensorflow.Material, *light_intensity: Optional[tensorflow.Tensor] = None*, *light_two_sided: bool = False*, *uvs: Optional[tensorflow.Tensor] = None*, *normals: Optional[tensorflow.Tensor] = None*, *uv_indices: Optional[tensorflow.Tensor] = None*, *normal_indices: Optional[tensorflow.Tensor] = None*, *colors: Optional[tensorflow.Tensor] = None*, *directly_visible: bool = True*)

Object combines geometry, material, and lighting information and aggregate them in a single class. This is a convinent class for constructing redner scenes.

redner supports only triangle meshes for now. It stores a pool of vertices and access the pool using integer index. Some times the two vertices can have the same 3D position but different texture coordinates, because UV mapping creates seams and need to duplicate vertices. In this can we can use an additional "uv_indices" array to access the uv pool.

Parameters

- **vertices** (*tf.Tensor*) – 3D position of vertices float32 tensor with size num_vertices x 3
- **indices** (*tf.Tensor*) – vertex indices of triangle faces. int32 tensor with size num_triangles x 3
- **material** (pyredner.Material) –
- **light_intensity** (*Optional[tf.Tensor]*) – make this object an area light float32 tensor with size 3
- **light_two_sided** (*boolean*) – Does the light emit from two sides of the shape?
- **uvs** (*Optional[tf.Tensor]:*) – optional texture coordinates. float32 tensor with size num_uvs x 2 doesn't need to be the same size with vertices if uv_indices is None
- **normals** (*Optional[tf.Tensor]*) – shading normal float32 tensor with size num_normals x 3 doesn't need to be the same size with vertices if normal_indices is None
- **uv_indices** (*Optional[tf.Tensor]*) – overrides indices when accessing uv coordinates int32 tensor with size num_uvs x 3
- **normal_indices** (*Optional[tf.Tensor]*) – overrides indices when accessing shading normals int32 tensor with size num_normals x 3
- **colors** (*Optional[tf.Tensor]*) – optional per-vertex color float32 tensor with size num_vertices x 3
- **directly_visible** (*boolean*) – optional setting to see if object is visible to camera during rendering.

class pyredner_tensorflow.**PointLight**(*position: tensorflow.Tensor*, *intensity: tensorflow.Tensor*)

Point light with squared distance falloff for deferred rendering.

render(*self*, *position: tensorflow.Tensor*, *normal: tensorflow.Tensor*, *albedo: tensorflow.Tensor*)

class pyredner_tensorflow.**RednerCameraType**

static asCameraType(*index: tensorflow.Tensor*)

static asTensor(*cameratype: redner.CameraType*)

`class pyredner_tensorflow.`**RednerChannels**

static asChannel(*index: tensorflow.Tensor*)

static asTensor(*channel: redner.channels*)

`class pyredner_tensorflow.`**RednerSamplerType**

static asSamplerType(*index: tensorflow.Tensor*)

static asTensor(*samplertype: redner.SamplerType*)

`pyredner_tensorflow.`**SH**(*l*, *m*, *theta*, *phi*)

`pyredner_tensorflow.`**SH_reconstruct**(*coeffs*, *res*)

`pyredner_tensorflow.`**SH_renormalization**(*l*, *m*)

`class pyredner_tensorflow.`**SamplerType**

`class pyredner_tensorflow.`**Scene**(*camera*, *shapes=[]*, *materials=[]*, *area_lights=[]*, *objects=None*, *envmap=None*)

A scene is a collection of camera, geometry, materials, and light. Currently there are two ways to construct a scene: one is through lists of Shape, Material, and AreaLight. The other one is through a list of Object. It is more recommended to use the Object construction. The Shape/Material/AreaLight options are here for legacy issue.

Parameters

- **shapes** (*List[pyredner.Shape] = [],*) –
- **materials** (*List[pyredner.Material] = [],*) –
- **area_lights** (*List[pyredner.AreaLight] = [],*) –
- **objects** (*Optional[List[pyredner.Object]] = None,*) –
- **envmap** (*Optional[pyredner.EnvironmentMap] = None*) –

classmethod load_state_dict(*cls*, *state_dict*)

state_dict(*self*)

`class pyredner_tensorflow.`**Shape**(*vertices: tensorflow.Tensor*, *indices: tensorflow.Tensor*, *material_id: int*, *uvs: Optional[tensorflow.Tensor] = None*, *normals: Optional[tensorflow.Tensor] = None*, *uv_indices: Optional[tensorflow.Tensor] = None*, *normal_indices: Optional[tensorflow.Tensor] = None*, *colors: Optional[tensorflow.Tensor] = None*)

redner supports only triangle meshes for now. It stores a pool of vertices and access the pool using integer index. Some times the two vertices can have the same 3D position but different texture coordinates, because UV mapping creates seams and need to duplicate vertices. In this can we can use an additional “uv_indices” array to access the uv pool.

Parameters

- **vertices** (*tf.Tensor*) – 3D position of vertices float32 tensor with size num_vertices x 3

- **indices** (*tf.Tensor*) – vertex indices of triangle faces. int32 tensor with size num_triangles x 3
- **uvs** (*Optional[tf.Tensor]:*) – optional texture coordinates. float32 tensor with size num_uvs x 2 doesn't need to be the same size with vertices if uv_indices is not None
- **normals** (*Optional[tf.Tensor]*) – shading normal float32 tensor with size num_normals x 3 doesn't need to be the same size with vertices if normal_indices is not None
- **uv_indices** (*Optional[tf.Tensor]*) – overrides indices when accessing uv coordinates int32 tensor with size num_uvs x 3
- **normal_indices** (*Optional[tf.Tensor]*) – overrides indices when accessing shading normals int32 tensor with size num_normals x 3

classmethod **load_state_dict**(*cls*, *state_dict*)

state_dict(*self*)

class pyredner_tensorflow.**SpotLight**(*position: tensorflow.Tensor*, *spot_direction: tensorflow.Tensor*, *spot_exponent: tensorflow.Tensor*, *intensity: tensorflow.Tensor*)

Spot light with cosine falloff for deferred rendering. Note that we do not provide the cosine cutoff here since it is not differentiable.

render(*self*, *position: tensorflow.Tensor*, *normal: tensorflow.Tensor*, *albedo: tensorflow.Tensor*)

class pyredner_tensorflow.**Texture**(*texels*, *uv_scale=tf.constant([1.0, 1.0])*)

Representing a texture and its mipmap.

Parameters

- **texels** (*torch.Tensor*) – a float32 tensor with size C or [height, width, C]
- **uv_scale** (*Optional[torch.Tensor]*) – scale the uv coordinates when mapping the texture a float32 tensor with size 2

generate_mipmap(*self*)

classmethod **load_state_dict**(*cls*, *state_dict*)

state_dict(*self*)

property **texels**(*self*)

pyredner_tensorflow.**associated_legendre_polynomial**(*l*, *m*, *x*)

pyredner_tensorflow.**automatic_camera_placement**(*shapes: List*, *resolution: Tuple[int, int]*)

Given a list of shapes, generates camera parameters automatically using the bounding boxes of the shapes. Place the camera at some distances from the shapes, so that it can see all of them. Inspired by https://github.com/mitsuba-renderer/mitsuba/blob/master/src/librender/scene.cpp#L286

pyredner_tensorflow.**camera_type**

pyredner_tensorflow.**channels**

pyredner_tensorflow.**compute_uvs**(*vertices*, *indices*, *print_progress=True*)

Compute UV coordinates of a given mesh using a charting algorithm with least square conformal mapping. This calls the xatlas library.

Parameters

- **vertices** (*tf.Tensor*) – 3D position of vertices float32 tensor with size num_vertices x 3

- **indices** (*tf.Tensor*) – vertex indices of triangle faces. int32 tensor with size num_triangles x 3

Returns

- *tf.Tensor* – uv vertices pool, float32 Tensor with size num_uv_vertices x 3
- *tf.Tensor* – uv indices, int32 Tensor with size num_triangles x 3

`pyredner_tensorflow.`**`compute_vertex_normal`**(*vertices: tensorflow.Tensor*, *indices: tensorflow.Tensor*, *weighting_scheme: str = 'max'*)

Compute vertex normal by weighted average of nearby face normals using Nelson Max's algorithm. See Weights for Computing Vertex Normals from Facet Vectors.

Parameters

- **vertices** (*tf.Tensor*) – 3D position of vertices float32 tensor with size num_vertices x 3
- **indices** (*tf.Tensor*) – vertex indices of triangle faces. int32 tensor with size num_triangles x 3
- **weighting_scheme** (*str*) – How do we compute the weighting. Currently we support two weighting methods: 'max' and 'cotangent'. 'max' corresponds to Nelson Max's algorithm that uses the inverse length and sine of the angle as the weight (see Weights for Computing Vertex Normals from Facet Vectors), 'cotangent' corresponds to weights derived through a discretization of the gradient of triangle area (see, e.g., "Implicit Fairing of Irregular Meshes using Diffusion and Curvature Flow" from Desbrun et al.)

Returns per-vertex normal, float32 Tensor with size num_vertices x 3

Return type tf.Tensor

`pyredner_tensorflow.`**`cpu_device_id = 0`**

`pyredner_tensorflow.`**`create_gradient_buffers`**(*ctx*)

`pyredner_tensorflow.`**`data_ptr`**(*tensor*)

`pyredner_tensorflow.`**`forward`**(*seed: int*, **args*)

Forward rendering pass: given a serialized scene and output an image.

`pyredner_tensorflow.`**`gen_look_at_matrix`**(*pos*, *look*, *up*)

`pyredner_tensorflow.`**`gen_perspective_matrix`**(*fov*, *clip_near*, *clip_far*)

`pyredner_tensorflow.`**`gen_rotate_matrix`**(*angles: tensorflow.Tensor*)

Given a 3D Euler angle vector, outputs a rotation matrix.

Parameters **angles** (*torch.Tensor*) – 3D Euler angle

Returns 3x3 rotation matrix

Return type tf.Tensor

`pyredner_tensorflow.`**`gen_scale_matrix`**(*scale*)

`pyredner_tensorflow.`**`gen_translate_matrix`**(*translate*)

`pyredner_tensorflow.`**`generate_geometry_image`**(*size: int*)

Generate an spherical geometry image [Gu et al. 2002 and Praun and Hoppe 2003] of size [2 * size + 1, 2 * size + 1]. This can be used for encoding a genus-0 surface into a regular image, so that it is more convienent for a CNN to process. The topology is given by a tesselated octahedron. UV is given by the spherical mapping.

Duplicated vertex are mapped to the one with smaller index (so some vertices on the geometry image is unused by the indices).

Parameters

- **size** (*int*) – size of the geometry image
- **device_name** (*Optional[str]*) – Which device should we store the data in. If set to None, use the device from pyredner.get_device_name().

Returns

- *tf.Tensor* – vertices of size [(2 * size + 1 * 2 * size + 1), 3]
- *tf.Tensor* – indices of size [2 * (2 * size + 1 * 2 * size + 1), 3]
- *tf.Tensor* – uvs of size [(2 * size + 1 * 2 * size + 1), 2]

pyredner_tensorflow.**generate_intrinsic_mat**(*fx: tensorflow.Tensor*, *fy: tensorflow.Tensor*, *skew: tensorflow.Tensor*, *x0: tensorflow.Tensor*, *y0: tensorflow.Tensor*)

Generate the following 3x3 intrinsic matrix given the parameters.

fx, skew, x0

0, fy, y0

0, 0, 1

Parameters

- **fx** (*tf.Tensor*) – Focal length at x dimension. 1D tensor with size 1.
- **fy** (*tf.Tensor*) – Focal length at y dimension. 1D tensor with size 1.
- **skew** (*tf.Tensor*) – Axis skew parameter describing shearing transform. 1D tensor with size 1.
- **x0** (*tf.Tensor*) – Principle point offset at x dimension. 1D tensor with size 1.
- **y0** (*tf.Tensor*) – Principle point offset at y dimension. 1D tensor with size 1.

Returns 3x3 intrinsic matrix

Return type tf.Tensor

pyredner_tensorflow.**generate_quad_light**(*position: tensorflow.Tensor*, *look_at: tensorflow.Tensor*, *size: tensorflow.Tensor*, *intensity: tensorflow.Tensor*, *directly_visible: bool = True*)

Generate a pyredner.Object that is a quad light source.

Parameters

- **position** (*tf.Tensor*) – 1-d tensor of size 3
- **look_at** (*tf.Tensor*) – 1-d tensor of size 3
- **size** (*tf.Tensor*) – 1-d tensor of size 2
- **intensity** (*tf.Tensor*) – 1-d tensor of size 3

Returns quad light source

Return type *pyredner.Object*

`pyredner_tensorflow.`**`generate_sphere`**(*theta_steps: int*, *phi_steps: int*)

Generate a triangle mesh representing a UV sphere, center at (0, 0, 0) with radius 1.

Parameters

- **`theta_steps`** (*`int`*) – zenith subdivision
- **`phi_steps`** (*`int`*) – azimuth subdivision

Returns

- *tf.Tensor* – vertices
- *tf.Tensor* – indices
- *tf.Tensor* – uvs
- *tf.Tensor* – normals

`pyredner_tensorflow.`**`get_cpu_device_id`**()

Get the cpu device id we are using.

`pyredner_tensorflow.`**`get_device_name`**()

Get the current tensorflow device name we are using.

`pyredner_tensorflow.`**`get_gpu_device_id`**()

Get the gpu device id we are using.

`pyredner_tensorflow.`**`get_print_timing`**()

Get whether we print time measurements or not.

`pyredner_tensorflow.`**`get_tensor_dimension`**(*t*)

Return dimension of the TF tensor in Int

get_shape() returns *TensorShape*.

`pyredner_tensorflow.`**`get_use_correlated_random_number`**()

See set_use_correlated_random_number

`pyredner_tensorflow.`**`get_use_gpu`**()

Get whether we are using CUDA or not.

`pyredner_tensorflow.`**`gpu_device_id = 0`**

`pyredner_tensorflow.`**`imread`**(*filename: str*, *gamma: float = 2.2*)

read img from filename

Parameters

- **`filename`** (*`str`*) –
- **`gamma`** (*`float`*) – if the image is not an OpenEXR file, apply gamma correction

Return type A float32 tensor with size [height, width, channel]

`pyredner_tensorflow.`**`imwrite`**(*img: tensorflow.Tensor*, *filename: str*, *gamma: float = 2.2*, *normalize: bool = False*)

write img to filename

Parameters

- **`img`** (*`tf.Tensor`*) – with size [height, width, channel]
- **`filename`** (*`str`*) –

- **gamma** (*float*) – if the image is not an OpenEXR file, apply gamma correction
- **normalize** – normalize img to the range [0, 1] before writing

pyredner_tensorflow.**is_empty_tensor**(*tensor*)

pyredner_tensorflow.**linear_to_srgb**(*x*)

pyredner_tensorflow.**load_mitsuba**(*filename*)

Load from a Mitsuba scene file as PyTorch tensors.

pyredner_tensorflow.**load_obj**(*filename: str*, *obj_group: bool = True*, *flip_tex_coords: bool = True*, *use_common_indices: bool = False*, *return_objects: bool = False*)

Load from a Wavefront obj file as PyTorch tensors.

Parameters

- **obj_group** (*bool*) – split the meshes based on materials
- **flip_tex_coords** (*bool*) – flip the v coordinate of uv by applying v' = 1 - v
- **use_common_indices** (*bool*) – Use the same indices for position, uvs, normals. Not recommended since texture seams in the objects sharing the same positions would cause the optimization to "tear" the object
- **return_objects** (*bool*) – Output list of Object instead. If there is no corresponding material for a shape, assign a grey material.

Returns

- *if return_objects == True, return a list of Object*
- *if return_objects == False, return (material_map, mesh_list, light_map),*
- *material_map -> Map[mtl_name, WavefrontMaterial]*
- *mesh_list -> List[TriangleMesh]*
- *light_map -> Map[mtl_name, torch.Tensor]*

pyredner_tensorflow.**normalize**(*v*)

NOTE: torch.norm() uses Frobineus norm which is Euclidean and L2

pyredner_tensorflow.**print_timing = True**

pyredner_tensorflow.**radians**(*deg*)

pyredner_tensorflow.**read_tensor**(*filename*, *shape*)

Parameters

- **filename** (*str*) –
- **shape** (*np.array*) –

pyredner_tensorflow.**render**(**x*)

The main TensorFlow interface of C++ redner.

pyredner_tensorflow.**render_albedo**(*scene: Union[pyredner_tensorflow.Scene, List[pyredner_tensorflow.Scene]]*, *alpha: bool = False*, *num_samples: Union[int, Tuple[int, int]] = (16, 4)*, *seed: Optional[Union[int, List[int]]] = None*, *sample_pixel_center: bool = False*, *device_name: Optional[str] = None*)

Render the diffuse albedo colors of the scenes.

Parameters

- **scene** (*Union[pyredner.Scene, List[pyredner.Scene]]*) – pyredner Scene containing camera, geometry and material. Can be a single scene or a list for batch render. For batch rendering all scenes need to have the same resolution.
- **alpha** (*bool*) – If set to False, generates a 3-channel image, otherwise generates a 4-channel image where the fourth channel is alpha.
- **num_samples** (*Union[int, Tuple[int, int]]*) – number of samples for forward and backward passes, respectively if a single integer is provided, use the same number of samples for both
- **seed** (*Optional[Union[int, List[int]]]*) – Random seed used for sampling. Randomly assigned if set to None. For batch render, if seed it not None, need to provide a list of seeds.
- **sample_pixel_center** (*bool*) – Always sample at the pixel center when rendering. This trades noise with aliasing. If this option is activated, the rendering becomes non-differentiable (since there is no antialiasing integral), and redner's edge sampling becomes an approximation to the gradients of the aliased rendering.
- **device_name** (*Optional[str]*) – Which device should we store the data in. If set to None, use the device from pyredner.get_device_name().

Returns

if input scene is a list: a tensor with size [N, H, W, C], N is the list size
else: a tensor with size [H, W, C]
if alpha == True, C = 4.
else, C = 3.

Return type tf.Tensor or List[tf.Tensor]

`pyredner_tensorflow.`**`render_deferred`**(*scene: Union[pyredner_tensorflow.Scene, List[pyredner_tensorflow.Scene]], lights: Union[List[DeferredLight], List[List[DeferredLight]]], alpha: bool = False, aa_samples: int = 2, seed: Optional[Union[int, List[int]]] = None, sample_pixel_center: bool = False, device_name: Optional[str] = None*)

Render the scenes using deferred rendering. We generate G-buffer images containing world-space position, normal, and albedo using redner, then shade the G-buffer using TensorFlow code. Assuming Lambertian shading and does not compute shadow.

Parameters

- **scene** (*Union[pyredner.Scene, List[pyredner.Scene]]*) – pyredner Scene containing camera, geometry and material. Can be a single scene or a list for batch render. For batch rendering all scenes need to have the same resolution.
- **lights** (*Union[List[DeferredLight], List[List[DeferredLight]]]*) – Lights for deferred rendering. If the scene is a list, and only a single list of lights is provided, the same lights are applied to all scenes. If a list of lists of lights is provided, each scene is lit by the corresponding lights.
- **alpha** (*bool*) – If set to False, generates a 3-channel image, otherwise generates a 4-channel image where the fourth channel is alpha.

- **aa_samples** (*int*) – Number of samples used for anti-aliasing at both x, y dimensions (e.g. if aa_samples=2, 4 samples are used).
- **seed** (*Optional[Union[int, List[int]]]*) – Random seed used for sampling. Randomly assigned if set to None. For batch render, if seed it not None, need to provide a list of seeds.
- **sample_pixel_center** (*bool*) – Always sample at the pixel center when rendering. This trades noise with aliasing. If this option is activated, the rendering becomes non-differentiable (since there is no antialiasing integral), and redner's edge sampling becomes an approximation to the gradients of the aliased rendering.
- **device_name** (*Optional[str]*) – Which device should we store the data in. If set to None, use the device from pyredner.get_device_name().

Returns

if input scene is a list: a tensor with size [N, H, W, C], N is the list size
else: a tensor with size [H, W, C]
if alpha == True, C = 4.
else, C = 3.

Return type tf.Tensor or List[tf.Tensor]

`pyredner_tensorflow.`**`render_g_buffer`**(*scene: pyredner_tensorflow.Scene*, *channels: List[redner.channels]*, *num_samples: Union[int, Tuple[int, int]] = (1, 1)*, *seed: Optional[int] = None*, *sample_pixel_center: bool = False*, *device_name: Optional[str] = None*)

Render a G buffer from the scene.

Parameters

- **scene** (`pyredner.Scene`) – pyredner Scene containing camera, geometry, material, and lighting
- **channels** (*List[pyredner.channels]*) –

  A list of the following channels:
  pyredner.channels.alpha
  pyredner.channels.depth
  pyredner.channels.position
  pyredner.channels.geometry_normal
  pyredner.channels.shading_normal
  pyredner.channels.uv
  pyredner.channels.barycentric_coordinates
  pyredner.channels.diffuse_reflectance
  pyredner.channels.specular_reflectance
  pyredner.channels.roughness
  pyredner.channels.generic_texture
  pyredner.channels.vertex_color
  pyredner.channels.shape_id
  pyredner.channels.triangle_id
  pyredner.channels.material_id

- **num_samples** (*Union[int, Tuple[int, int]]*) – Number of samples for forward and backward passes, respectively. If a single integer is provided, use the same number of samples for both.
- **seed** (*Optional[int]*) – Random seed used for sampling. Randomly assigned if set to None.
- **sample_pixel_center** (*bool*) – Always sample at the pixel center when rendering. This trades noise with aliasing. If this option is activated, the rendering becomes non-differentiable (since there is no antialiasing integral), and redner's edge sampling becomes an approximation to the gradients of the aliased rendering.
- **device_name** (*Optional[str]*) – Which device should we store the data in. If set to None, use the device from pyredner.get_device_name().

Returns a tensor with size [H, W, C]

Return type tf.Tensor

`pyredner_tensorflow.`**`render_generic`**`(`*scene: pyredner_tensorflow.Scene*, *channels: List*, *max_bounces: int = 1*, *sampler_type=pyredner.sampler_type.sobol*, *num_samples: Union[int, Tuple[int, int]] = (4, 4)*, *seed: Optional[int] = None*, *sample_pixel_center: bool = False*, *device_name: Optional[str] = None*`)`

A generic rendering function that can be either pathtracing or g-buffer rendering or both.

Parameters

- **scene** (*Union[pyredner.Scene, List[pyredner.Scene]]*) – pyredner Scene containing camera, geometry and material. Can be a single scene or a list for batch render. For batch rendering all scenes need to have the same resolution.
- **channels** (*List[pyredner.channels]*) –

  A list of the following channels:

  pyredner.channels.alpha

  pyredner.channels.depth

  pyredner.channels.position

  pyredner.channels.geometry_normal

  pyredner.channels.shading_normal

  pyredner.channels.uv

  pyredner.channels.barycentric_coordinates

  pyredner.channels.diffuse_reflectance

  pyredner.channels.specular_reflectance

  pyredner.channels.roughness

  pyredner.channels.generic_texture

  pyredner.channels.vertex_color

  pyredner.channels.shape_id

  pyredner.channels.triangle_id

  pyredner.channels.material_id
- **max_bounces** (*int*) – Number of bounces for global illumination, 1 means direct lighting only.
- **sampler_type** (*pyredner.sampler_type*) –

  Which sampling pattern to use? See Chapter 7 of the PBRT book for an explanation of the difference between different samplers.

Following samplers are supported:

pyredner.sampler_type.independent

pyredner.sampler_type.sobol

- **num_samples** (*int*) – Number of samples per pixel for forward and backward passes. Can be an integer or a tuple of 2 integers.
- **seed** (*Optional[Union[int, List[int]]]*) – Random seed used for sampling. Randomly assigned if set to None. For batch render, if seed it not None, need to provide a list of seeds.
- **sample_pixel_center** (*bool*) – Always sample at the pixel center when rendering. This trades noise with aliasing. If this option is activated, the rendering becomes non-differentiable (since there is no antialiasing integral), and redner's edge sampling becomes an approximation to the gradients of the aliased rendering.
- **device_name** (*Optional[str]*) – Which device should we store the data in. If set to None, use the device from pyredner.get_device_name().

Returns

if input scene is a list: a tensor with size [N, H, W, C], N is the list size

else: a tensor with size [H, W, C]

Return type tf.Tensor or List[tf.Tensor]

`pyredner_tensorflow.`**`render_pathtracing`**(*scene: Union[*pyredner_tensorflow.Scene*, List[*pyredner_tensorflow.Scene*]], alpha: bool = False, max_bounces: int = 1, sampler_type=pyredner.sampler_type.sobol, num_samples: Union[int, Tuple[int, int]] = (4, 4), seed: Optional[Union[int, List[int]]] = None, sample_pixel_center: bool = False, device_name: Optional[str] = None*)

Render a pyredner scene using pathtracing.

Parameters

- **scene** (*Union[*`pyredner.Scene`*, List[*`pyredner.Scene`*]]*) – pyredner Scene containing camera, geometry and material. Can be a single scene or a list for batch render. For batch rendering all scenes need to have the same resolution.
- **max_bounces** (*int*) – Number of bounces for global illumination, 1 means direct lighting only.
- **sampler_type** (*pyredner.sampler_type*) –

  Which sampling pattern to use? See Chapter 7 of the PBRT book for an explanation of the difference between different samplers.

  Following samplers are supported:

  pyredner.sampler_type.independent

  pyredner.sampler_type.sobol
- **num_samples** (*int*) – Number of samples per pixel for forward and backward passes. Can be an integer or a tuple of 2 integers.

- **seed** (*Optional[Union[int, List[int]]]*) – Random seed used for sampling. Randomly assigned if set to None. For batch render, if seed it not None, need to provide a list of seeds.
- **sample_pixel_center** (*bool*) – Always sample at the pixel center when rendering. This trades noise with aliasing. If this option is activated, the rendering becomes non-differentiable (since there is no antialiasing integral), and redner's edge sampling becomes an approximation to the gradients of the aliased rendering.
- **device_name** (*Optional[str]*) – Which device should we store the data in. If set to None, use the device from pyredner.get_device_name().

Returns

if input scene is a list: a tensor with size [N, H, W, C], N is the list size
else: a tensor with size [H, W, C]
if alpha == True, C = 4.
else, C = 3.

Return type tf.Tensor or List[tf.Tensor]

pyredner_tensorflow.**sampler_type**

pyredner_tensorflow.**save_obj**(*shape: Union[*pyredner_tensorflow.Object, pyredner_tensorflow.Shape*]*, *filename: str*, *flip_tex_coords=True*)

Save to a Wavefront obj file from an Object or a Shape.

Parameters

- **shape** (*Union[*pyredner.Object, pyredner.Shape*]*) –
- **filename** (*str*) –
- **flip_tex_coords** (*bool*) – flip the v coordinate of uv by applying v' = 1 - v

pyredner_tensorflow.**serialize_scene**(*scene:* pyredner_tensorflow.Scene, *num_samples: Union[int, Tuple[int, int]]*, *max_bounces: int*, *channels=[redner.channels.radiance]*, *sampler_type=redner.SamplerType.independent*, *use_primary_edge_sampling=True*, *use_secondary_edge_sampling=True*, *sample_pixel_center: bool = False*, *device_name: Optional[str] = None*)

Given a pyredner scene & rendering options, convert them to a linear list of argument, so that we can use it in TensorFlow.

Parameters

- **scene** (pyredner.Scene) –
- **num_samples** (*int*) – number of samples per pixel for forward and backward passes can be an integer or a tuple of 2 integers if a single integer is provided, use the same number of samples for both
- **max_bounces** (*int*) – number of bounces for global illumination 1 means direct lighting only
- **channels** (*List[redner.channels]*) –

  A list of channels that should present in the output image

following channels are supported:
redner.channels.radiance,
redner.channels.alpha,
redner.channels.depth,
redner.channels.position,
redner.channels.geometry_normal,
redner.channels.shading_normal,
redner.channels.uv,
redner.channels.barycentric_coordinates,
redner.channels.diffuse_reflectance,
redner.channels.specular_reflectance,
redner.channels.vertex_color,
redner.channels.roughness,
redner.channels.generic_texture,
redner.channels.shape_id,
redner.channels.triangle_id,
redner.channels.material_id
all channels, except for shape id, triangle id and material id, are differentiable

- **`sampler_type`** (*`redner.SamplerType`*) –

  Which sampling pattern to use?
  see *Chapter 7 of the PBRT book <http://www.pbr-book.org/3ed-2018/Sampling_and_Reconstruction.html>* for an explanation of the difference between different samplers.
  Following samplers are supported:
  redner.SamplerType.independent
  redner.SamplerType.sobol

- **`use_primary_edge_sampling`** (*`bool`*) –
- **`use_secondary_edge_sampling`** (*`bool`*) –
- **`sample_pixel_center`** (*`bool`*) – Always sample at the pixel center when rendering. This trades noise with aliasing. If this option is activated, the rendering becomes non-differentiable (since there is no antialiasing integral), and redner's edge sampling becomes an approximation to the gradients of the aliased rendering.
- **`device_name`** (*`Optional[str]`*) – Which device should we store the data in. If set to None, use the device from pyredner.get_device_name().

`pyredner_tensorflow.`**`serialize_texture`**(*texture*, *args*, *device_name*)

`pyredner_tensorflow.`**`set_cpu_device_id`**(*did: int*)

Set the cpu device id we are using.

`pyredner_tensorflow.`**`set_gpu_device_id`**(*did: int*)

Set the gpu device id we are using.

`pyredner_tensorflow.`**`set_print_timing`**(*v: bool*)

Set whether to print time measurements or not.

`pyredner_tensorflow.`**`set_use_correlated_random_number`**(*v: bool*)

There is a bias-variance trade off in the backward pass.

If the forward pass and the backward pass are correlated
the gradients are biased for L2 loss.
(E[d/dx(f(x) - y)^2] = E[(f(x) - y) d/dx f(x)])
    = E[f(x) - y] E[d/dx f(x)]
The last equation only holds when f(x) and d/dx f(x) are independent.
It is usually better to use the unbiased one, but we left it as an option here

`pyredner_tensorflow.`**`set_use_gpu`**(*v: bool*)

Set whether to use CUDA or not.

`pyredner_tensorflow.`**`srgb_to_linear`**(*x*)

`pyredner_tensorflow.`**`unpack_args`**(*seed*, *args*, *use_primary_edge_sampling=None*, *use_secondary_edge_sampling=None*)

Given a list of serialized scene arguments, unpack all information into a Context.

`pyredner_tensorflow.`**`use_correlated_random_number = False`**

`pyredner_tensorflow.`**`use_gpu`**

`pyredner_tensorflow.`**`visualize_screen_gradient`**(*grad_img: tensorflow.Tensor*, *seed: int*, *scene: pyredner_tensorflow.Scene*, *num_samples: Union[int, Tuple[int, int]]*, *max_bounces: int*, *channels: List = [redner.channels.radiance]*, *sampler_type=redner.SamplerType.independent*, *use_primary_edge_sampling: bool = True*, *use_secondary_edge_sampling: bool = True*, *sample_pixel_center: bool = False*)

Given a serialized scene and output an 2-channel image, which visualizes the derivatives of pixel color with respect to the screen space coordinates.

Parameters

- **grad_img** (*`Optional[tf.Tensor]`*) – The “adjoint” of the backpropagation gradient. If you don’t know what this means just give None
- **seed** (*`int`*) – seed for the Monte Carlo random samplers
- **arguments.** (*`See serialize_scene for the explanation of the rest of the`*) –

CHAPTER THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX