Class HolovizOp

Table of contents

- Defined in File holoviz.hpp

Nested Relationships

Nested Types

- Struct HolovizOp::InputSpec
- Struct InputSpec::View

Inheritance Relationships

Base Type

- `public holoscan::Operator` (Class Operator)

Class Documentation

class HolovizOp : public holoscan::Operator

> Operator class for data visualization.
>
> This high-speed viewer handles compositing, blending, and visualization of RGB or RGBA images, masks, geometric primitives, text and depth maps. The operator can auto detect the format of the input tensors acquired at the `receivers` port. Else the input specification can be set at creation time using the `tensors` parameter or at runtime when passing input specifications to the `input_specs` port.
>
> Depth maps and 3D geometry are rendered in 3D and support camera movement. The camera is controlled using the mouse:
>
> - Orbit (LMB)
> - Pan (LMB + CTRL | MMB)
> - Dolly (LMB + SHIFT | RMB | Mouse wheel)
> - Look Around (LMB + ALT | LMB + CTRL + SHIFT)

- Zoom (Mouse wheel + SHIFT) Or by providing new values at the `camera_eye_input`, `camera_look_at_input` or `camera_up_input` input ports. The camera pose can be output at the `camera_pose_output` port when `enable_camera_pose_output` is set to `true`.

==Named Inputs==

- **receivers** : multi-receiver accepting `nvidia::gxf::Tensor` and/or `nvidia::gxf::VideoBuffer`
    - Any number of upstream ports may be connected to this `receivers` port. This port can accept either VideoBuffers or Tensors. These inputs can be in either host or device memory. Each tensor or video buffer will result in a layer. The operator autodetects the layer type for certain input types (e.g. a video buffer will result in an image layer). For other input types or more complex use cases, input specifications can be provided either at initialization time as a parameter or dynamically at run time (via `input_specs`). On each call to `compute`, tensors corresponding to all names specified in the `tensors` parameter must be found or an exception will be raised. Any extra, named tensors not present in the `tensors` parameter specification (or optional, dynamic `input_specs` input) will be ignored.
- **input_specs** : `holoscan::ops::HolovizOp::InputSpec` (optional)
    - A list of `InputSpec` objects. This port can be used to dynamically update the overlay specification at run time. No inputs are required on this port in order for the operator to `compute`.
- **render_buffer_input** : `nvidia::gxf::VideoBuffer` (optional)
    - An empty render buffer can optionally be provided. The video buffer must have format GXF_VIDEO_FORMAT_RGBA and be in device memory. This input port only exists if `enable_render_buffer_input` was set to true, in which case `compute` will only be called when a message arrives on this input.
- **camera_eye_input** : `std::array&lt;float, 3&gt;` (optional)

    - Camera eye position. The camera is animated to reach the new position.

- **camera_look_at_input** : `std::array<float, 3>` (optional)
    - Camera look at position. The camera is animated to reach the new position.

- **camera_up_input** : : `std::array<float, 3>` (optional)
    - Camera up vector. The camera is animated to reach the new vector.

==Named Outputs==

- **render_buffer_output** : `nvidia::gxf::VideoBuffer` (optional)
    - Output for a filled render buffer. If an input render buffer is specified, it is using that one, else it allocates a new buffer. The video buffer will have format GXF_VIDEO_FORMAT_RGBA and will be in device memory. This output is useful for offline rendering or headless mode. This output port only exists if `enable_render_buffer_output` was set to true.

- **camera_pose_output** : `std::array<float, 16>` or `nvidia::gxf::Pose3D` (optional)
    - Output the camera pose. Depending on the value of `camera_pose_output_type` this outputs a 4x4 row major projection matrix (type `std::array<float, 16>`) or the camera extrinsics model (type `nvidia::gxf::Pose3D`). This output port only exists if `enable_camera_pose_output` was set to `True`.

==Parameters==

- **receivers**: List of input queues to component accepting `gxf::Tensor` or `gxf::VideoBuffer`.
    - type: `std::vector<gxf::Handle<gxf::Receiver>>`

- **enable_render_buffer_input**: Enable `render_buffer_input` (default: `false`)

    - type: `bool`
- **enable_render_buffer_output**: Enable `render_buffer_output` (default: `false`)
    - type: `bool`
- **enable_camera_pose_output**: Enable `camera_pose_output` (default: `false`)
    - type: `bool`
- **tensors**: List of input tensor specifications (default: `[]`)
    - type: `InputSpec`
        - **name**: name of the tensor containing the input data to display
            - type: `std::string`
        - **type**: input type (default `"unknown"`)
            - type: `std::string`
            - possible values:
                - **unknown**: unknown type, the operator tries to guess the type by inspecting the tensor.
                - **color**: RGB or RGBA color 2d image.
                - **color_lut**: single channel 2d image, color is looked up.
                - **points**: point primitives, one coordinate (x, y) per primitive.
                - **lines**: line primitives, two coordinates (x0, y0) and (x1, y1) per primitive.
                - **line_strip**: line strip primitive, a line primitive i is defined by each coordinate (xi, yi) and the following (xi+1, yi+1).

- **triangles**: triangle primitive, three coordinates (x0, y0), (x1, y1) and (x2, y2) per primitive.
- **crosses**: cross primitive, a cross is defined by the center coordinate and the size (xi, yi, si).
- **rectangles**: axis aligned rectangle primitive, each rectangle is defined by two coordinates (xi, yi) and (xi+1, yi+1).
- **ovals**: oval primitive, an oval primitive is defined by the center coordinate and the axis sizes (xi, yi, sxi, syi).
- **text**: text is defined by the top left coordinate and the size (x, y, s) per string, text strings are defined by InputSpec member **text**.
- **depth_map**: single channel 2d array where each element represents a depth value. The data is rendered as a 3d object using points, lines or triangles. The color for the elements can be specified through `depth_map_color`. Supported format: 8-bit unsigned normalized format that has a single 8-bit depth component.
- **depth_map_color**: RGBA 2d image, same size as the depth map. One color value for each element of the depth map grid. Supported format: 32-bit unsigned normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.

- **opacity**: layer opacity, 1.0 is fully opaque, 0.0 is fully transparent (default: `1.0`)
  - type: `float`
- **priority**: layer priority, determines the render order, layers with higher priority values are rendered on top of layers with lower priority values (default: `0`)
  - type: `int32_t`

        - **color**: RGBA color of rendered geometry (default: `[1.f, 1.f, 1.f, 1.f]`)
            - type: `std::vector<float>`
        - **line_width**: line width for geometry made of lines (default: `1.0`)
            - type: `float`
        - **point_size**: point size for geometry made of points (default: `1.0`)
            - type: `float`
        - **text**: array of text strings, used when `type` is text. (default: `[]`)
            - type: `std::vector<std::string>`
        - **depth_map_render_mode**: depth map render mode (default: `points`)
            - type: `std::string`
            - possible values:
                - **points**: render as points
                - **lines**: render as lines
                - **triangles**: render as triangles
- **color_lut**: Color lookup table for tensors of type 'color_lut', vector of four float RGBA values
    - type: `std::vector<std::vector<float>>`
- **window_title**: Title on window canvas (default: `"Holoviz"`)
    - type: `std::string`

- **display_name**: In exclusive mode, name of display to use as shown with `xrandr` or `hwinfo --monitor` (default: `DP-0`)
  - type: `std::string`
- **width**: Window width or display resolution width if in exclusive or fullscreen mode (default: `1920`)
  - type: `uint32_t`
- **height**: Window height or display resolution height if in exclusive or fullscreen mode (default: `1080`)
  - type: `uint32_t`
- **framerate**: Display framerate if in exclusive mode (default: `60`)
  - type: `uint32_t`
- **use_exclusive_display**: Enable exclusive display (default: `false`)
  - type: `bool`
- **fullscreen**: Enable fullscreen window (default: `false`)
  - type: `bool`
- **headless**: Enable headless mode. No window is opened, the render buffer is output to `render_buffer_output`. (default: `false`)
  - type: `bool`
- **window_close_scheduling_term**: BooleanSchedulingTerm to stop the codelet from ticking when the window is closed
  - type: `gxf::Handle<gxf::BooleanSchedulingTerm>`
- **allocator**: Allocator used to allocate memory for `render_buffer_output`

    - type: `gxf::Handle<gxf::Allocator>`
- **font_path**: File path for the font used for rendering text (default: `""`)
    - type: `std::string`
- **cuda_stream_pool**: Instance of gxf::CudaStreamPool
    - type: `gxf::Handle<gxf::CudaStreamPool>`
- **camera_pose_output_type**: Type of data output at `camera_pose_output`. Supported values are `projection_matrix` and `extrinsics_model`. Default value is `projection_matrix`.
    - type: `std::string`
- **camera_eye**: Initial camera eye position.
    - type: `std::array<float, 3>`
- **camera_look_at**: Initial camera look at position.
    - type: `std::array<float, 3>`
- **camera_up**: Initial camera up vector.
    - type: `std::array<float, 3>`

==Device Memory Requirements==

If `render_buffer_input` is enabled, the provided buffer is used and no memory block will be allocated. Otherwise, when using this operator with a `BlockMemoryPool`, a single device memory block is needed (`storage_type` = 1). The size of this memory block can be determined by rounding the width and height up to the nearest even size and then padding the rows as needed so that the row stride is a multiple of 256 bytes. C++ code to calculate the block size is as follows:

```
#include <cstdint> int64_t get_block_size(int32_t height, int32_t width) { int32_t height_even = height + (height & 1); int32_t width_even = width + (width & 1);
```

```
int64_t row_bytes = width_even * 4; // 4 bytes per pixel for 8-bit RGBA int64_t row_stride = (row_bytes % 256 == 0) ? row_bytes : ((row_bytes / 256 + 1) * 256); return height_even * row_stride; }
```

==Notes==

1. Displaying Color Images

   Image data can either be on host or device (GPU). Multiple image formats are supported

   - R 8 bit unsigned
   - R 16 bit unsigned
   - R 16 bit float
   - R 32 bit unsigned
   - R 32 bit float
   - RGB 8 bit unsigned
   - BGR 8 bit unsigned
   - RGBA 8 bit unsigned
   - BGRA 8 bit unsigned
   - RGBA 16 bit unsigned
   - RGBA 16 bit float
   - RGBA 32 bit float

   When the `type` parameter is set to `color_lut` the final color is looked up using the values from the `color_lut` parameter. For color lookups these image formats are supported

   - R 8 bit unsigned

- R 16 bit unsigned
- R 32 bit unsigned

2. Drawing Geometry

   In all cases, `x` and `y` are normalized coordinates in the range `[0, 1]`. The `x` and `y` correspond to the horizontal and vertical axes of the display, respectively. The origin `(0, 0)` is at the top left of the display. Geometric primitives outside of the visible area are clipped. Coordinate arrays are expected to have the shape `(N, C)` where `N` is the coordinate count and `C` is the component count for each coordinate.

   - Points are defined by a `(x, y)` coordinate pair.
   - Lines are defined by a set of two `(x, y)` coordinate pairs.
   - Lines strips are defined by a sequence of `(x, y)` coordinate pairs. The first two coordinates define the first line, each additional coordinate adds a line connecting to the previous coordinate.
   - Triangles are defined by a set of three `(x, y)` coordinate pairs.
   - Crosses are defined by `(x, y, size)` tuples. `size` specifies the size of the cross in the `x` direction and is optional, if omitted it's set to `0.05`. The size in the `y` direction is calculated using the aspect ratio of the window to make the crosses square.
   - Rectangles (bounding boxes) are defined by a pair of 2-tuples defining the upper-left and lower-right coordinates of a box: `(x1, y1), (x2, y2)`.
   - Ovals are defined by `(x, y, size_x, size_y)` tuples. `size_x` and `size_y` are optional, if omitted they are set to `0.05`.
   - Texts are defined by `(x, y, size)` tuples. `size` specifies the size of the text in `y` direction and is optional, if omitted it's set to `0.05`. The size in the `x` direction is calculated using the aspect ratio of the window. The index of each coordinate references a text string from the `text` parameter and

the index is clamped to the size of the text array. For example, if there is one item set for the `text` parameter, e.g. `text=["my_text"]` and three coordinates, then `my_text` is rendered three times. If `text=["first text", "second text"]` and three coordinates are specified, then `first text` is rendered at the first coordinate, `second text` at the second coordinate and then `second text` again at the third coordinate. The `text` string array is fixed and can't be changed after initialization. To hide text which should not be displayed, specify coordinates greater than `(1.0, 1.0)` for the text item, the text is then clipped away.

- 3D Points are defined by a `(x, y, z)` coordinate tuple.
- 3D Lines are defined by a set of two `(x, y, z)` coordinate tuples.
- 3D Lines strips are defined by a sequence of `(x, y, z)` coordinate tuples. The first two coordinates define the first line, each additional coordinate adds a line connecting to the previous coordinate.
- 3D Triangles are defined by a set of three `(x, y, z)` coordinate tuples.

3. Displaying Depth Maps

   When `type` is `depth_map` the provided data is interpreted as a rectangular array of depth values. Additionally a 2d array with a color value for each point in the grid can be specified by setting `type` to `depth_map_color`.

   The type of geometry drawn can be selected by setting `depth_map_render_mode`.

   Depth maps are rendered in 3D and support camera movement.

4. Output

   By default a window is opened to display the rendering, but the extension can also be run in headless mode with the `headless` parameter.

   Using a display in exclusive mode is also supported with the `use_exclusive_display` parameter. This reduces the latency by avoiding the desktop compositor.

The rendered framebuffer can be output to `render_buffer_output`.

Public Types

enum class InputType

Input type.

All geometric primitives expect a 1d array of coordinates. Coordinates range from 0.0 (left, top) to 1.0 (right, bottom).

Values:

enumerator UNKNOWN

unknown type, the operator tries to guess the type by inspecting the tensor

enumerator COLOR

GRAY, RGB or RGBA 2d color image.

enumerator COLOR_LUT

single channel 2d image, color is looked up

enumerator POINTS

point primitives, one coordinate (x, y) per primitive

enumerator LINES

line primitives, two coordinates (x0, y0) and (x1, y1) per primitive

enumerator LINE_STRIP

line strip primitive, a line primitive i is defined by each coordinate (xi, yi) and the following (xi+1, yi+1)

enumerator TRIANGLES

triangle primitive, three coordinates (x0, y0), (x1, y1) and (x2, y2) per primitive

enumerator CROSSES

cross primitive, a cross is defined by the center coordinate and the size (xi, yi, si)

enumerator RECTANGLES

axis aligned rectangle primitive, each rectangle is defined by two coordinates (xi, yi) and (xi+1, yi+1)

enumerator OVALS

oval primitive, an oval primitive is defined by the center coordinate and the axis sizes (xi, yi, sxi, syi)

enumerator TEXT

text is defined by the top left coordinate and the size (x, y, s) per string, text strings are define by InputSpec::text_

enumerator DEPTH_MAP

single channel 2d array where each element represents a depth value. The data is rendered as a 3d object using points, lines or triangles. The color for the elements can be specified through `DEPTH_MAP_COLOR`. Supported format: 8-bit unsigned normalized format that has a single 8-bit depth component

enumerator DEPTH_MAP_COLOR

RGBA 2d image, same size as the depth map. One color value for each element of the depth map grid. Supported format: 32-bit unsigned normalized format that has an 8-bit R component in byte > 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3

enumerator POINTS_3D

3D point primitives, one coordinate (x, y, z) per primitive

enumerator LINES_3D

3D line primitives, two coordinates (x0, y0, z0) and (x1, y1, z1) per primitive

enumerator LINE_STRIP_3D

3D line strip primitive, a line primitive i is defined by each coordinate (xi, yi, zi) and the following (xi+1, yi+1, zi+1)

enumerator TRIANGLES_3D

3D triangle primitive, three coordinates (x0, y0, z0), (x1, y1, z1) and (x2, y2, z2) per primitive

enum class DepthMapRenderMode

Depth map render mode.

Values:

enumerator POINTS

render points

enumerator LINES

render lines

enumerator TRIANGLES

render triangles

Public Functions

HOLOSCAN_OPERATOR_FORWARD_ARGS (HolovizOp) HolovizOp()=default

virtual void setup(OperatorSpec &spec) override

Define the operator specification.

Parameters

spec – The reference to the operator specification.

virtual void initialize() override

Initialize the operator.

This function is called when the fragment is initialized by Executor::initialize_fragment().

virtual void start() override

Implement the startup logic of the operator.

This method is called multiple times over the lifecycle of the operator according to the order defined in the lifecycle, and used for heavy initialization tasks such as allocating memory resources.

virtual void compute(InputContext &op_input, OutputContext &op_output, ExecutionContext &context) override

Implement the compute method.

This method is called by the runtime multiple times. The runtime calls this method until the operator is stopped.

Parameters

- **op_input** – The input context of the operator.
- **op_output** – The output context of the operator.
- **context** – The execution context of the operator.

virtual void stop() override

Implement the shutdown logic of the operator.

This method is called multiple times over the lifecycle of the operator according to the order defined in the lifecycle, and used for heavy deinitialization tasks such as deallocation of all resources previously assigned in start.

struct InputSpec

Input specification

Public Functions

InputSpec() = default

inline InputSpec(const std::string &tensor_name, InputType type)

InputSpec(const std::string &tensor_name, const std::string &type_str)

explicit InputSpec(const std::string &yaml_description)

Returns

an InputSpec from the YAML form output by description()

inline explicit operator bool() const noexcept

Returns

true if the input spec is valid

std::string description() const

Returns

a YAML string representation of the InputSpec

Public Members

std::string tensor_name_

name of the tensor containing the input data

InputType type_ = InputType::UNKNOWN

input type

float opacity_ = 1.f

layer opacity, 1.0 is fully opaque, 0.0 is fully transparent

int32_t priority_ = 0

layer priority, determines the render order, layers with higher priority values are rendered on top of layers with lower priority values

std::vector<float> color_ = {1.f, 1.f, 1.f, 1.f}

color of rendered geometry

float line_width_ = 1.f

line width for geometry made of lines

float point_size_ = 1.f

point size for geometry made of points

std::vector<std::string> text_

array of text strings, used when type_ is TEXT.

DepthMapRenderMode depth_map_render_mode_ = DepthMapRenderMode::POINTS

depth map render mode, used if type_ is DEPTH_MAP or DEPTH_MAP_COLOR.

std::vector<View> views_

struct View

> Layer view.
>
> By default a layer will fill the whole window. When using a view the layer can be placed freely within the window.
>
> Layers can also be placed in 3D space by specifying a 3D transformation matrix. Note that for geometry layers there is a default matrix which allows coordinates in the range of [0 ... 1] instead of the Vulkan [-1 ... 1] range. When specifying a matrix for a geometry layer, this default matrix is overwritten.

When multiple views are specified the layer is drawn multiple times using the specified layer views.

It’s possible to specify a negative term for height, which flips the image. When using a negative height, one should also adjust the y value to point to the lower left corner of the viewport instead of the upper left corner.

Public Members

float offset_x_ = 0.f

float offset_y_ = 0.f

offset of top-left corner of the view. Top left coordinate of the window area is (0, 0) bottom right coordinate is (1, 1).

float width_ = 1.f

float height_ = 1.f

width and height of the view in normalized range. 1.0 is full size.

std::optional<std::array<float, 16>> matrix_

row major 4x4 transform matrix (optional, can be nullptr)