Visualization

Table of contents

List of Figures

Overview

Holoviz provides the functionality to composite real time streams of frames with multiple different other layers like segmentation mask layers, geometry layers and GUI layers.

For maximum performance Holoviz makes use of Vulkan, which is already installed as part of the Nvidia GPU driver.

Holoscan provides the Holoviz operator which is sufficient for many, even complex visualization tasks. The Holoviz operator is used by multiple Holoscan example applications.

Additionally, for more advanced use cases, the Holoviz module can be used to create application specific visualization operators. The Holoviz module provides a C++ API and is also used by the Holoviz operator.

The term Holoviz is used for both the Holoviz operator and the Holoviz module below. Both the operator and the module roughly support the same features set. Where applicable information how to use a feature with the operator and the module is provided. It's explicitly mentioned below when features are not supported by the operator.

Layers

The core entity of Holoviz are layers. A layer is a two-dimensional image object. Multiple layers are composited to create the final output.

These layer types are supported by Holoviz:

- image layer
- geometry layer
- GUI layer

All layers have common attributes which define the look and also the way layers are finally composited.

The priority determines the rendering order of the layers. Before rendering the layers they are sorted by priority, the layers with the lowest priority are rendered first so that

the layer with the highest priority is rendered on top of all other layers. If layers have the same priority then the render order of these layers is undefined.

The example below draws a transparent geometry layer on top of an image layer (geometry data and image data creation is omitted in the code). Although the geometry layer is specified first, it is drawn last because it has a higher priority (`1`) than the image layer (`0`).

Ingested Tab Module

Image Layers

Ingested Tab Module

Supported Image Formats

Ingested Tab Module

Geometry Layers

A geometry layer is used to draw 2d or 3d geometric primitives. 2d primitives are points, lines, line strips, rectangles, ovals or text and are defined with 2d coordinates (x, y). 3d primitives are points, lines, line strips or triangles and are defined with 3d coordinates (x, y, z).

Coordinates start with (0, 0) in the top left and end with (1, 1) in the bottom right for 2d primitives.

Ingested Tab Module

ImGui Layers

> **Note**
>
> ImGui layers are not supported when using the Holoviz operator.

The Holoviz module supports user interface layers created with Dear ImGui.

Calls to the Dear ImGui API are allowed between `viz::BeginImGuiLayer()` and `viz::EndImGuiLayer()` are used to draw to the ImGui layer. The ImGui layer behaves like other layers and is rendered with the layer opacity and priority.

The code below creates a Dear ImGui window with a checkbox used to conditionally show a image layer.

```
namespace viz = holoscan::viz; bool show_image_layer = false; while (!viz::WindowShouldClose()) { viz::Begin(); viz::BeginImGuiLayer(); ImGui::Begin("Options"); ImGui::Checkbox("Image layer", &show_image_layer); ImGui::End(); viz::EndLayer(); if (show_image_layer) { viz::BeginImageLayer(); viz::ImageHost(...); viz::EndLayer(); } viz::End(); }
```

ImGUI is a static library and has no stable API. Therefore the application and Holoviz have to use the same ImGUI version. Therefore the link target `holoscan::viz::imgui` is exported, make sure to link your app against that target.

Depth Map Layers

A depth map is a 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 also be specified.

Supported format for the depth map:

- 8-bit unsigned normalized format that has a single 8-bit depth component

Supported format for the depth color map:

- 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

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

Ingested Tab Module

Views

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**
>
> 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 view.

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.

Ingested Tab Module

Camera

When rendering 3d geometry using a geometry layer with 3d primitives or using a depth map layer the camera properties can either be set by the application or interactively changed by the user.

To interactively change the camera, use 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)

Ingested Tab Module

Using a display in exclusive mode

Usually Holoviz opens a normal window on the Linux desktop. In that case the desktop compositor is combining the Holoviz image with all other elements on the desktop. To avoid this extra compositing step, Holoviz can render to a display directly.

Configure a display for exclusive use

Ingested Tab Module

Enable exclusive display in Holoviz

Ingested Tab Module

The name of the display can either be the EDID name as displayed in the NVIDIA Settings, or the output name provided by `xrandr` or `hwinfo --monitor`.

> **Tip**
> Ingested Tab Module

CUDA streams

By default Holoviz is using CUDA stream `0` for all CUDA operations. Using the default stream can affect concurrency of CUDA operations, see stream synchronization behavior for more information.

Ingested Tab Module

Reading the framebuffer

The rendered frame buffer can be read back. This is useful when when doing offscreen rendering or running Holoviz in a headless environment.

> **Note**
>
> Reading the depth buffer is not supported when using the Holoviz operator.

Ingested Tab Module

Holoviz operator

Class documentation

`C++`

`Python`.

Examples

There are multiple examples both in Python and C++ showing how to use various features of the Holoviz operator.

Holoviz module

Concepts

The Holoviz module uses the concept of the immediate mode design pattern for its API, inspired by the Dear ImGui library. The difference to the retained mode, for which most APIs are designed for, is, that there are no objects created and stored by the application. This makes it fast and easy to make visualization changes in a Holoscan application.

Instances

The Holoviz module uses a thread-local instance object to store its internal state. The instance object is created when calling the Holoviz module is first called from a thread. All Holoviz module functions called from that thread use this instance.

When calling into the Holoviz module from other threads other than the thread from which the Holoviz module functions were first called, make sure to call `viz::GetCurrent()` and `viz::SetCurrent()` in the respective threads.

There are usage cases where multiple instances are needed, for example, to open multiple windows. Instances can be created by calling `viz::Create()`. Call `viz::SetCurrent()` to make the instance current before calling the Holoviz module function to be executed for the window the instance belongs to.

Getting started

The code below creates a window and displays an image.

First the Holoviz module needs to be initialized. This is done by calling `viz::Init()`.

The elements to display are defined in the render loop, termination of the loop is checked with `viz::WindowShouldClose()`.

The definition of the displayed content starts with `viz::Begin()` and ends with `viz::End()`. `viz::End()` starts the rendering and displays the rendered result.

Finally the Holoviz module is shutdown with `viz::Shutdown()`.

```
#include "holoviz/holoviz.hpp" namespace viz = holoscan::viz; viz::Init("Holoviz Example"); while (!viz::WindowShouldClose()) { viz::Begin(); viz::BeginImageLayer(); viz::ImageHost(width, height, viz::ImageFormat::R8G8B8A8_UNORM, image_data); viz::EndLayer(); viz::End(); } viz::Shutdown();
```

Result:

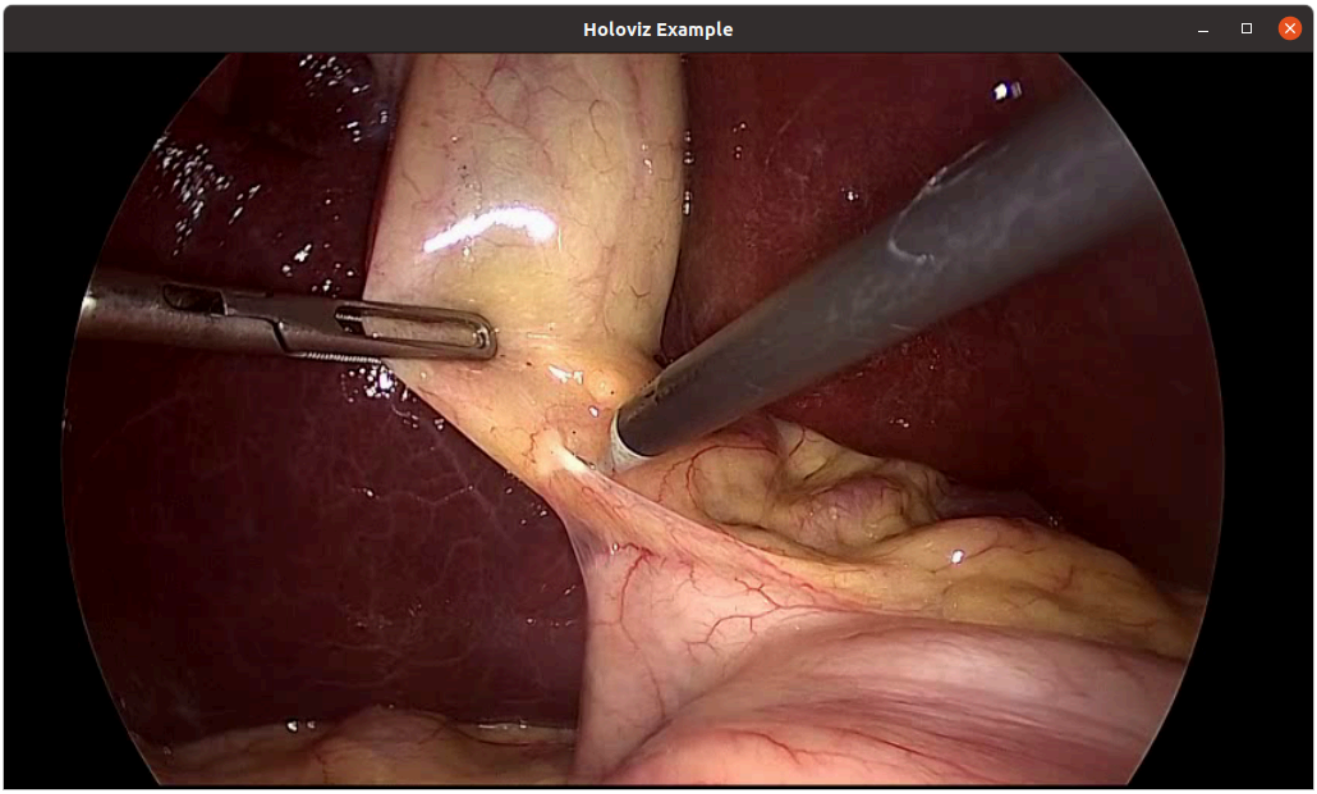

Fig. 20 *Holoviz example app*

API

Holoviz module API

Examples

There are multiple examples showing how to use various features of the Holoviz module.