RobotVisionSuite

算子介绍

RobotVisionSuite

图漾科技

2023.07

日期	版本	发布说明
2022.12	V1.0	第一次发布
2022.12	V1.1	新增17个算子简介
2022.01	V1.2	新增basic算子简介
2023.02	V2.0	算子更新，重新编写算子文档
2023.02	V2.1	新增calibration模块介绍
2023.03	V2.2	调整文档格式并新增 2d、basic、3d、file、communication 类算子案例 XML 与 数据
2023.04	V2.3	优化 selector、counter、MinimumBoundingBox、CloudProcess 算子

2d

ImageCrop 图像切割

ImageCrop 算子用于对 2D 图像进行裁剪处理。

ImageCrop [算子介绍视频]

算子参数

- `x1`：图像裁剪起点的列坐标。
- `x2`：图像裁剪终点的列坐标。
- `y1`：图像裁剪起点的行坐标。
- `y2`：图像裁剪终点的行坐标。
- `图像/image`：设置图像在 2D 视图中的可视化属性。
  -  打开图像可视化。
  -  关闭图像可视化。
- `图像列表/image_list`：设置图像列表在 2D 视图中的可视化属性。属性值描述与`图像`一致。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- `image`：
  - 数据类型：Image
  - 输入内容：待切割图像
- `image_list`：
  - 数据类型：ImageList
  - 输入内容：待切割图像列表

输出：

- `image`：
  - 数据类型：Image
  - 输出内容：切割后图像
- `image_list`：
  - 数据类型：ImageList
  - 输出内容：切割后图像列表

功能演示

使用 ImageCrop 算子截取待切割图像中的香蕉部分。

步骤1：算子准备

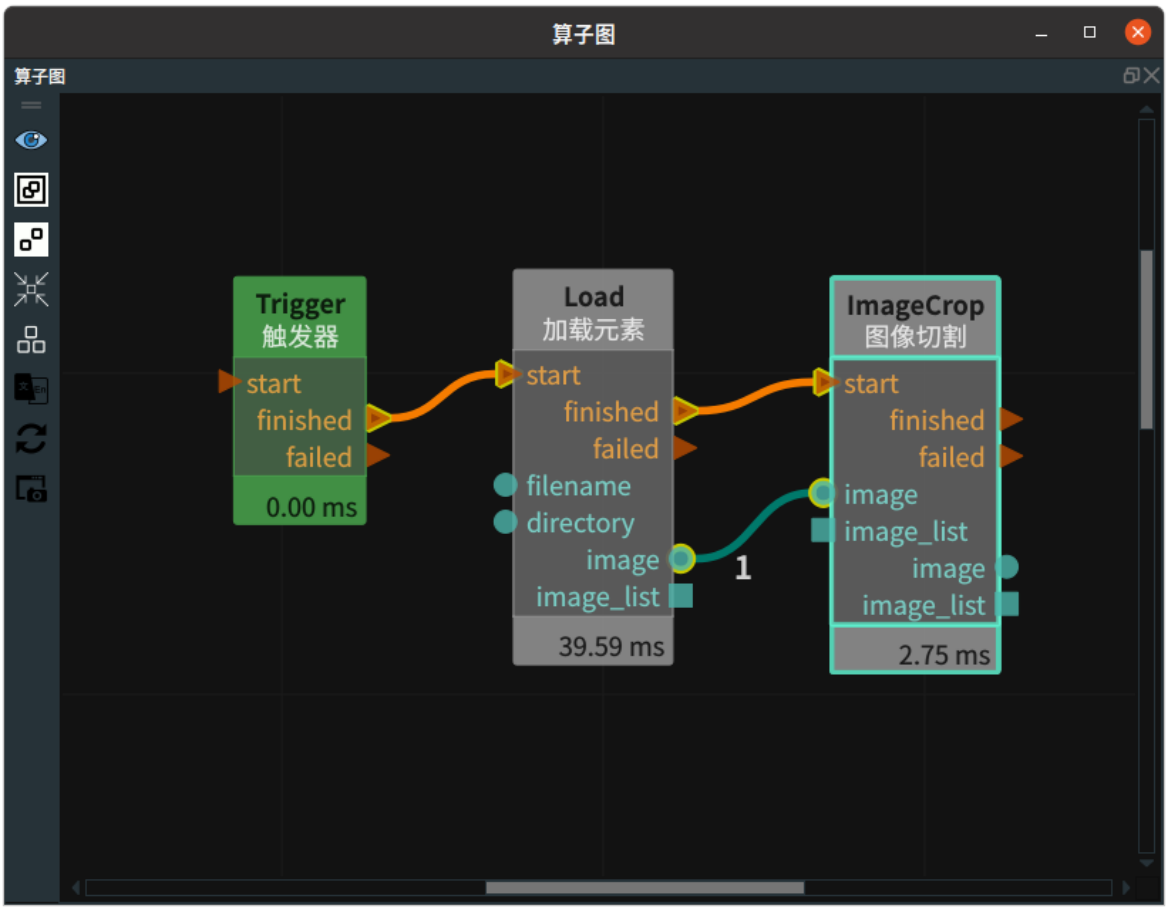
添加 Trigger、Load、ImageCrop 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 类型 → Image
    - 文件 → ••• → 选择 image 文件名 （*example_data/images/image.png*）
    - 图像 → 
2. 设置 ImageCrop 算子参数：
    - x1 → 0
    - x2 → 700
    - y1 → 0
    - y2 → 500
    - 图像 → 

    说明：当鼠标停留 2D 图片上，在 2D 视图下方可以显示当前像素点的坐标。

步骤3：连接算子

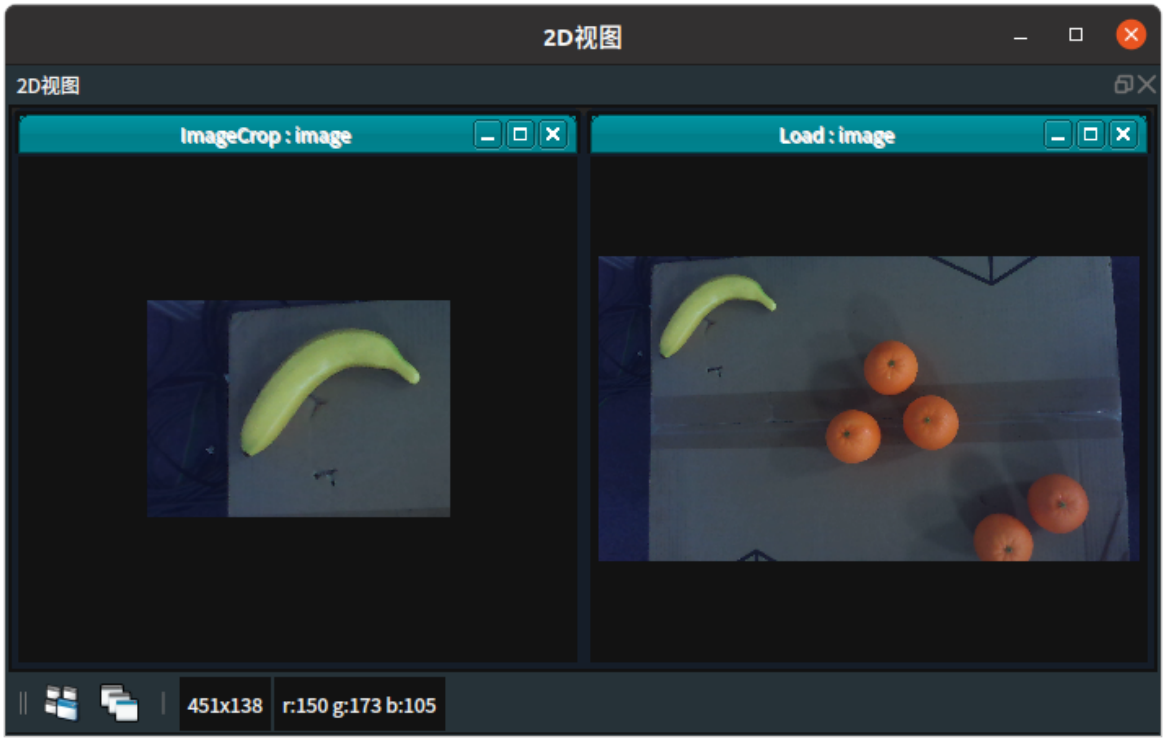


步骤4：运行

点击 RVS 的运行按钮，触发 Trigger 算子。

运行结果

如下图所示，2D 视图中显示 Load 算子和 ImageCrop 算子的结果。ImageCrop 算子截取出 700*500 的香蕉部分图像。


2D视图
2D视图
ImageCrop : image
Load : image
451x138
r:150 g:173 b:105

ImageZeroPadding 图像零位补丁

ImageZeroPadding 算子用于对2D图像进行零填充处理。即首先创建一个根据参数指定长宽并且灰度值全为 0 的基础图像，然后根据参数指定的起点坐标，将输入图“贴”在该基础图中并输出。

该算子常用于 ImageCrop之后的尺寸恢复。

ImageZeroPadding 算子介绍视频

算子参数

- **图像宽度补偿/ImgWidthOffset**：图像零填充起点的列坐标。
- **图像高度补偿/ImgHeightOffset**：图像零填充起点的行坐标。
- **图像宽度/ImgWidth**：图像零填充后的目标结果图像的宽度。
- **图像高度/ImgHeight**：图像零填充后的目标结果图像的高度。
- **图像/image**：设置图像在 2D 视图中的可视化属性。
  -  打开图像可视化。
  -  关闭图像可视化。
- **图像列表/image_list**：设置图像列表在 2D 视图中的可视化属性。属性值描述与**图像**一致。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **image**：
  - 数据类型：Image
  - 输入内容：待零填充处理图像
- **image_list**：
  - 数据类型：ImageList
  - 输入内容：待零填充处理图像列表

输出：

- **image**：
  - 数据类型：Image
  - 输出内容：零填充处理后图像
- **image**：
  - 数据类型：ImageList
  - 输出内容：零填充处理后图像列表

功能演示

使用 ImageZeroPadding 算子对切割后的 2D 图像进行零填充处理，根据图像原尺寸创建一个 1918*1076 且灰度值全为 0 的基础图像，并将切割后的图像“贴”在原始的位置。

图像切割部分参考 ImageCrop算子的功能演示部分。

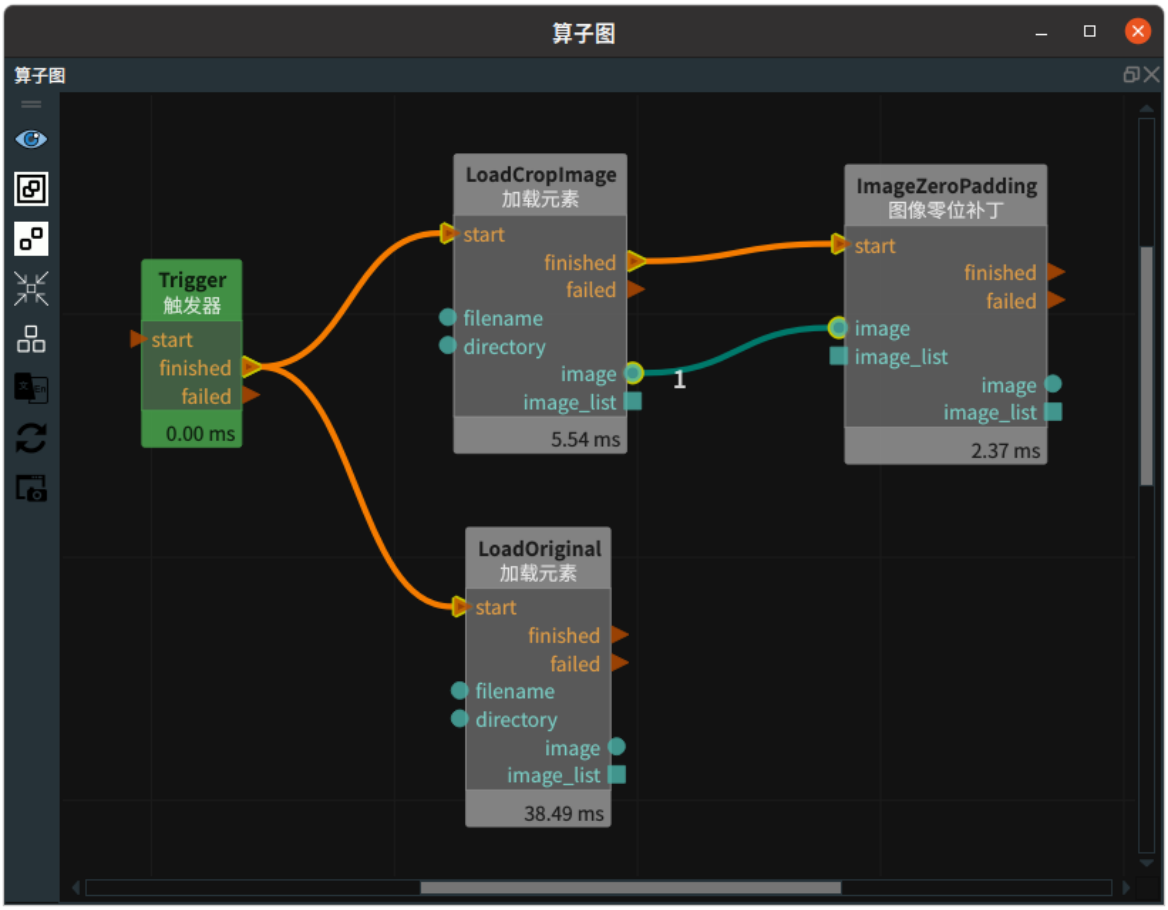
步骤1：算子准备

添加 Trigger、Load、ImageZeroPadding 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 算子名称 → LoadCropImage
    - 类型 → Image
    - 文件 → ••• → 选择切割后文件名 （*example_data/images/banana.png* ）
    - 图像 → 
2. 设置Load_1算子参数：
    - 算子名称 → ImageOriginal
    - 类型 → Image
    - 文件 → ••• → 选择切割前图像文件名 （*example_data/images/image.png* ）
    - 图像 → 
3. 设置 ImageZeroPadding 算子参数：
    - 图像宽度补偿 → 0
    - 图像高度补偿 → 0
    - 图像宽度 → 1918
    - 图像高度 → 1076
    - 图像 → 

步骤3：连接算子

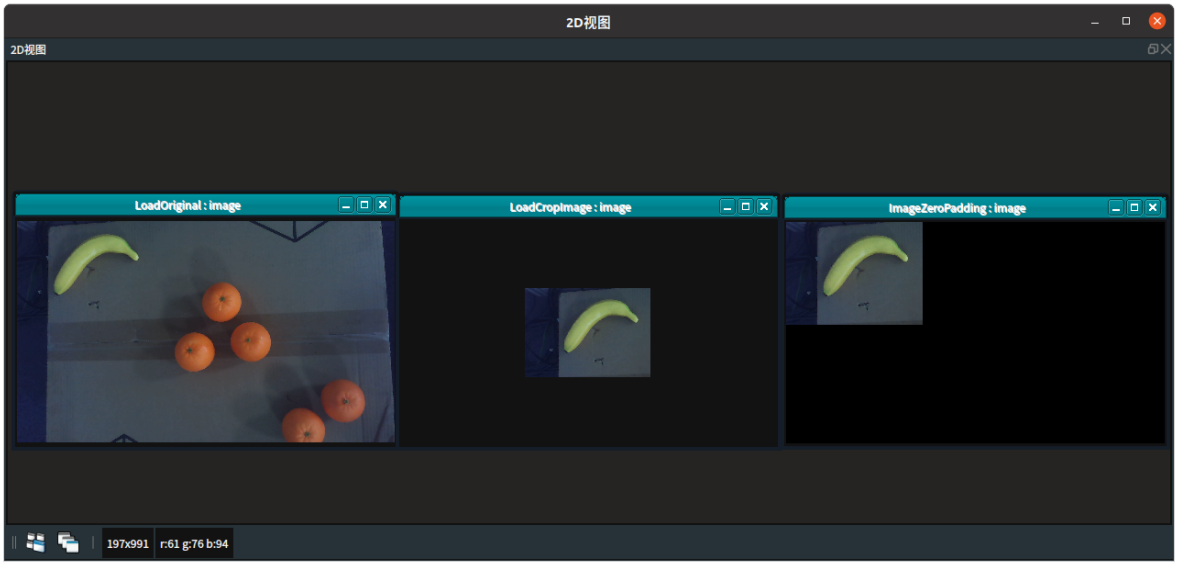


步骤4：运行

点击 RVS 的运行按钮，触发 Trigger 算子。

运行结果

如下图所示，2D 视图中显示 Load 算子和 ImageZeroPadding 算子的结果。ImageZeroPadding 算子的输出的基础图像尺寸：1918*1076，“贴”图的部分从原点开始。



ImageResize 图像尺寸改变

ImageResize 算子用于对 2D 图像进行缩放处理。

ImageResize 算子介绍视频

算子参数

- **宽度/width**：缩放后的图像宽度。
- **高度/height**：缩放后的图像高度。
- **调整大小模式/resize_mode**：缩放模式。
  - INTER_NEAREST：最近邻采样。
  - INTER_LINEAR：线性采样。
  - INTER_CUBIC：立方采样。

  说明： 具体详情可以网络搜索 opencv(C++) 对应的同名参数。
- **图像/image**：设置图像在 2D 视图中的可视化属性。
  -  打开图像可视化。
  -  关闭图像可视化。
- **图像列表/image_list**：设置图像列表在 2D 视图中的可视化属性。属性值描述与**图像**一致。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **image**：
  - 数据类型：Image
  - 输入内容：待缩放图像
- **image_list**：
  - 数据类型：Image
  - 输出内容：待缩放图像列表

输出：

- **image**：
  - 数据类型：Image
  - 输出内容：缩放后图像
- **image_list**：
  - 数据类型：Image
  - 输出内容：缩放后图像列表

功能演示

使用 ImageResize 算子将 1918 * 1076 的图像尺寸缩放为 400*400 。

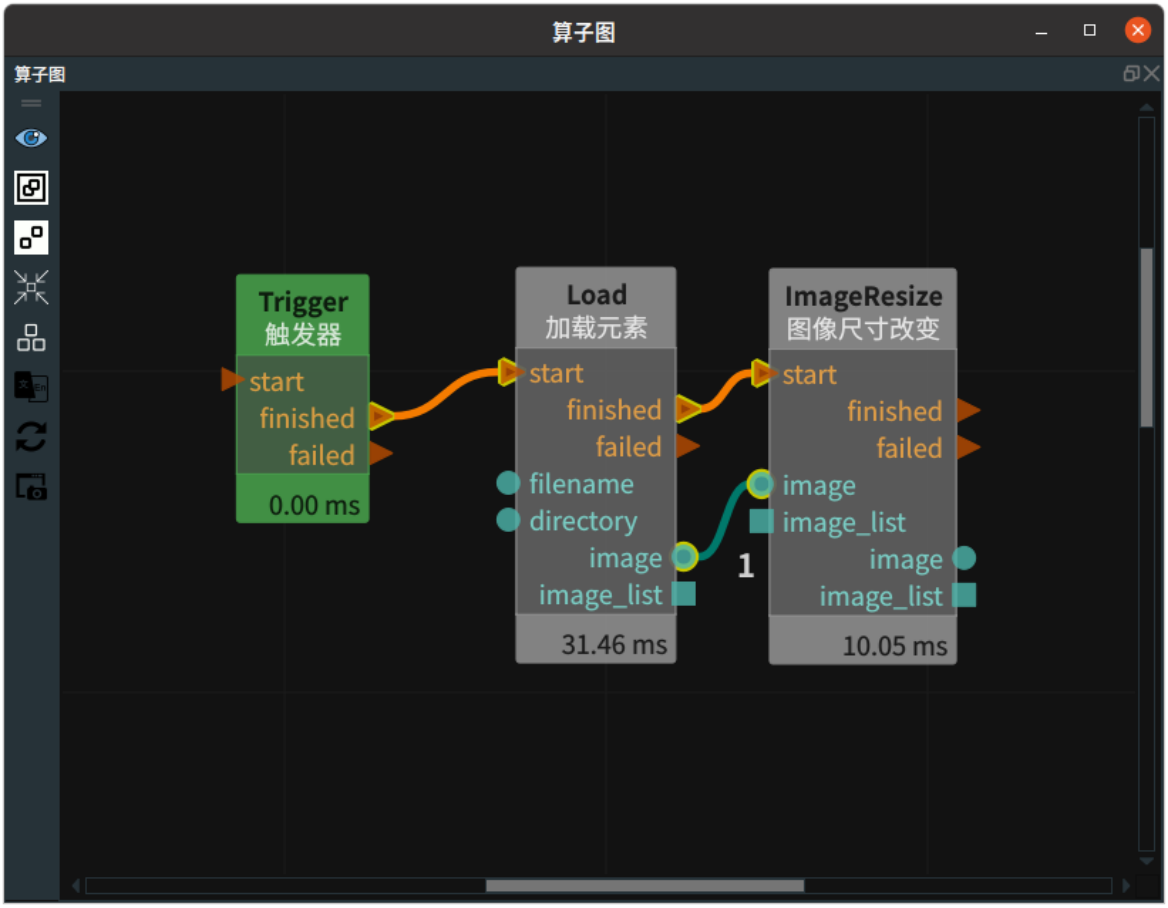
步骤1：算子准备

添加 Trigger、Load、ImageResize 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 类型 → Image
    - 文件 → ••• → 选择 image 文件名 （*example_data/images/image.png*）
    - 图像 → 
2. 设置 ImageResize 算子参数：
    - 宽度 → 400
    - 高度 → 400
    - 调整大小模式 → INTER_NEAREST
    - 图像 → 

步骤3：连接算子

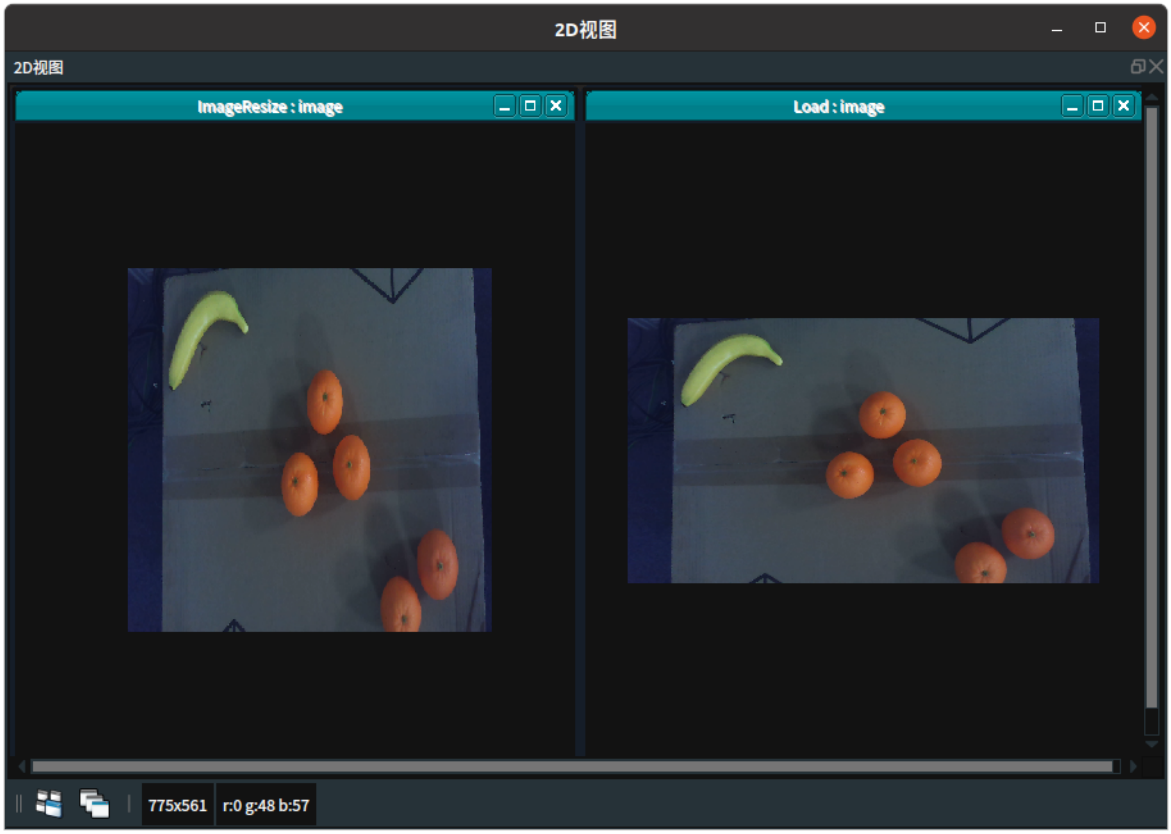


步骤4：运行

点击 RVS 的运行按钮，触发 Trigger 算子。

运行结果

如下图所示，2D 视图中显示 Load 算子和 ImageResize 算子的结果。ImageReize 算子将 1918*1076 的图像尺寸缩放为 400 * 400。


2D视图
2D视图
ImageResize : image
Load : image
775x561
r:0 g:48 b:57

3d

DownSampling 降采样

DownSampling 算子作用于对点云进行稀疏化处理。对于部分算子，少量稀疏的点云足以完成计算，过于稠密的点云只会降低算子执行效率，此时可以考虑首先对原始点云进行稀疏化处理再进行后续运算。

type	功能
DownSample	降采样。数据点结果分布较为均匀。
UniformSample	速度更快的降采样，但是数据点结果分布不均。

DownSample

将 DownSampling 算子 **类型** 设置为 DownSample ，通过对数据进行分割和重采样来降低数据的复杂度和密度，用于降采样。

算子参数

- **x方向采样/leaf_x**：指定 x 轴方向点云重采样间距。
- **y方向采样/leaf_y**：指定 y 轴方向点云重采样间距。
- **z方向采样/leaf_z**：指定 z 轴方向点云重采样间距。

  说明：默认都为0.01m，表示在 0.01m * 0.01m * 0.01m 的空间尺度内仅取一个点。
- **点云_cloud**：设置降采样后点云在 3D 视图中的可视化属性。
  -  打开过滤后点云可视化。
  -  关闭过滤后点云可视化。
  -  设置 3D 视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  -  设置点云中点的尺寸 。取值范围：[1,50] 。默认值：1 。
- **点云列表/cloud_list**：设置降采样后点云列表在 3D 视图中的可视化属性。值描述与 **点云** 一致。

数据端口输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **cloud**：
  - 数据类型：PointCloud
  - 输入内容：点云数据
- **cloud_list**：
  - 数据类型：PointCloudList
  - 输入内容：点云列表数据

输出：

- **cloud**：
  - 数据类型：PointCloud

- 输出内容：降采样后点云数据

- `cloud_list`：
  - 数据类型：PointCloudList
  - 输出内容：降采样后点云列表数据

功能演示

使用 DownSampling 中 DownSample 对加载的点云进行降采样。

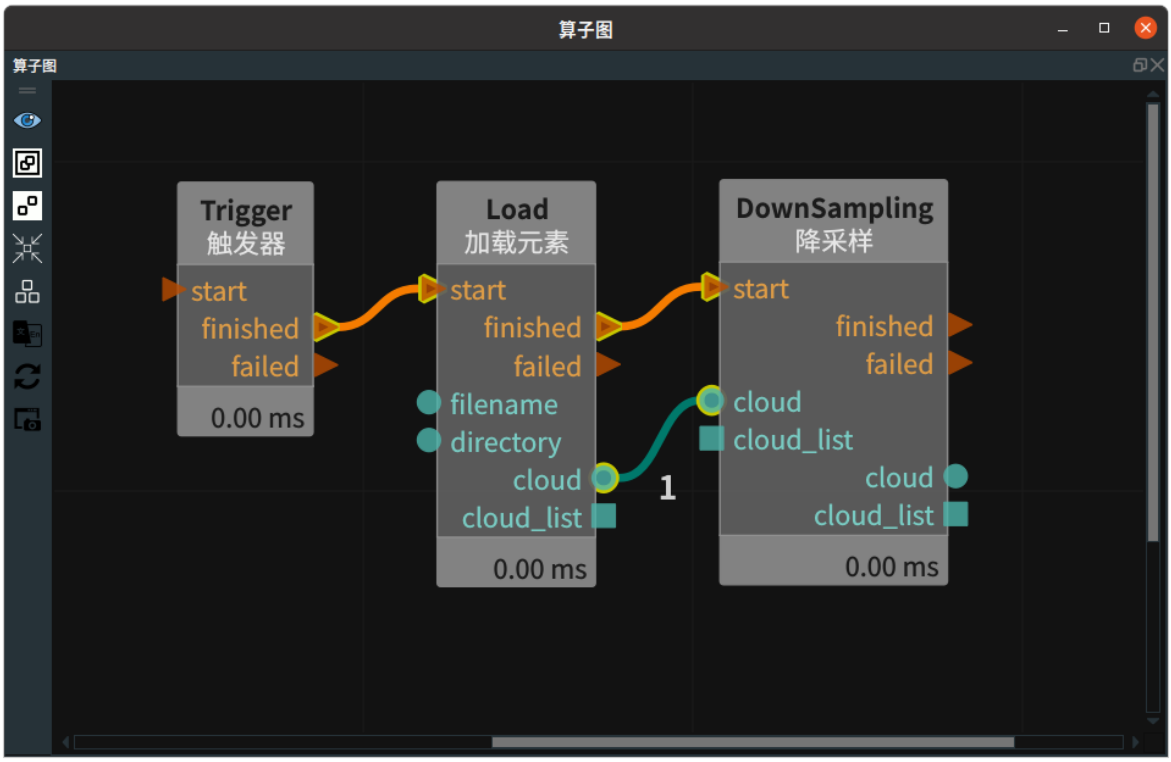
步骤1：算子准备

添加 Trigger 、Load 、DownSampling 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → PointCloud
   - 文件 → ••• → 选择点云文件名 (*example_data/pointcloud/fruit.pcd*)
   - 点云 →  可视→  -2
2. 设置 DownSampling 算子参数：
   - 类型 → DownSample
   - x方向采样 → 0.01
   - y方向采样 → 0.01
   - z方向采样 → 0.01
   - 点云 →  可视→  -2

步骤3：连接算子

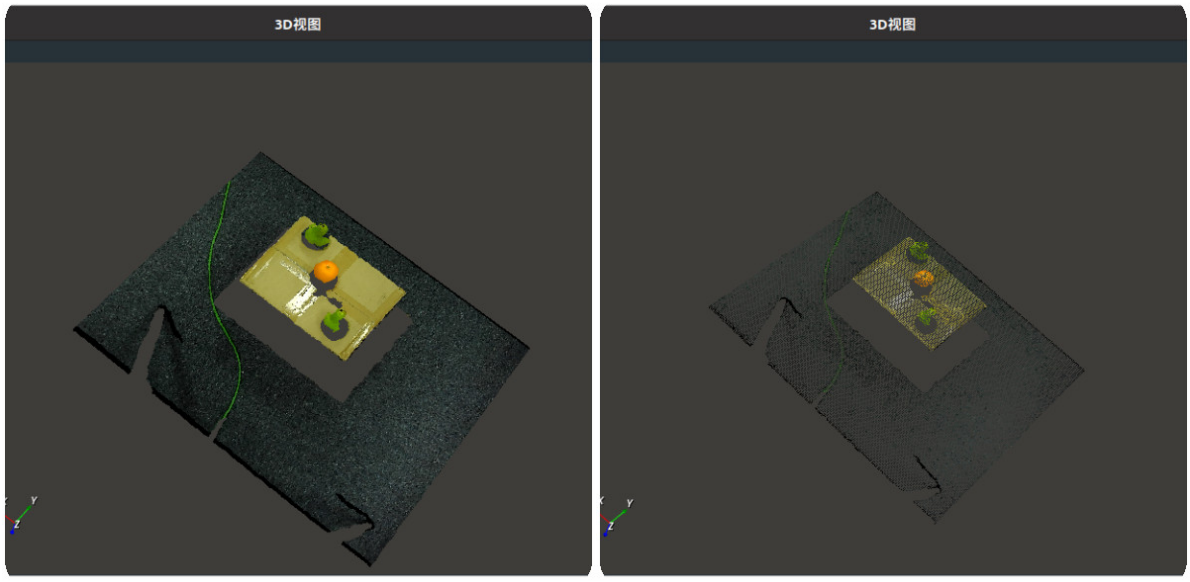


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，3D 视图中，左图为 Load 算子的点云可视化结果，右图为 DownSampling 算子的点云可视化结果。



UniformSample

将 DownSampling 算子 **类型** 设置为 UniformSample ，对数据进行均匀采样来保留数据的形状和分布，用于速度更快的降采样。

算子参数

- **采样规模/sampling_size**：抽样大小。
- **点云/cloud**：设置降采样后点云在3D视图中的可视化属性。
  - 打开过滤后点云可视化。
  - 关闭过滤后点云可视化。
  - 设置3D视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  - 设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。
- **点云列表/cloud_list**：设置降采样后点云列表在3D视图中的可视化属性。值描述与 **点云** 一致。

数据端口输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **cloud**：
  - 数据类型：PointCloud
  - 输入内容：点云数据
- **cloud_list**：
  - 数据类型：PointCloudList
  - 输入内容：点云列表数据

输出：

- **cloud**：
  - 数据类型：PointCloud

- 输入内容：降采样后点云数据

- `cloud_list`：
  - 数据类型：PointCloudList
  - 输出内容：降采样后点云列表数据

功能演示

使用 DownSampling 中 UniformSample 对加载的点云进行降采样。

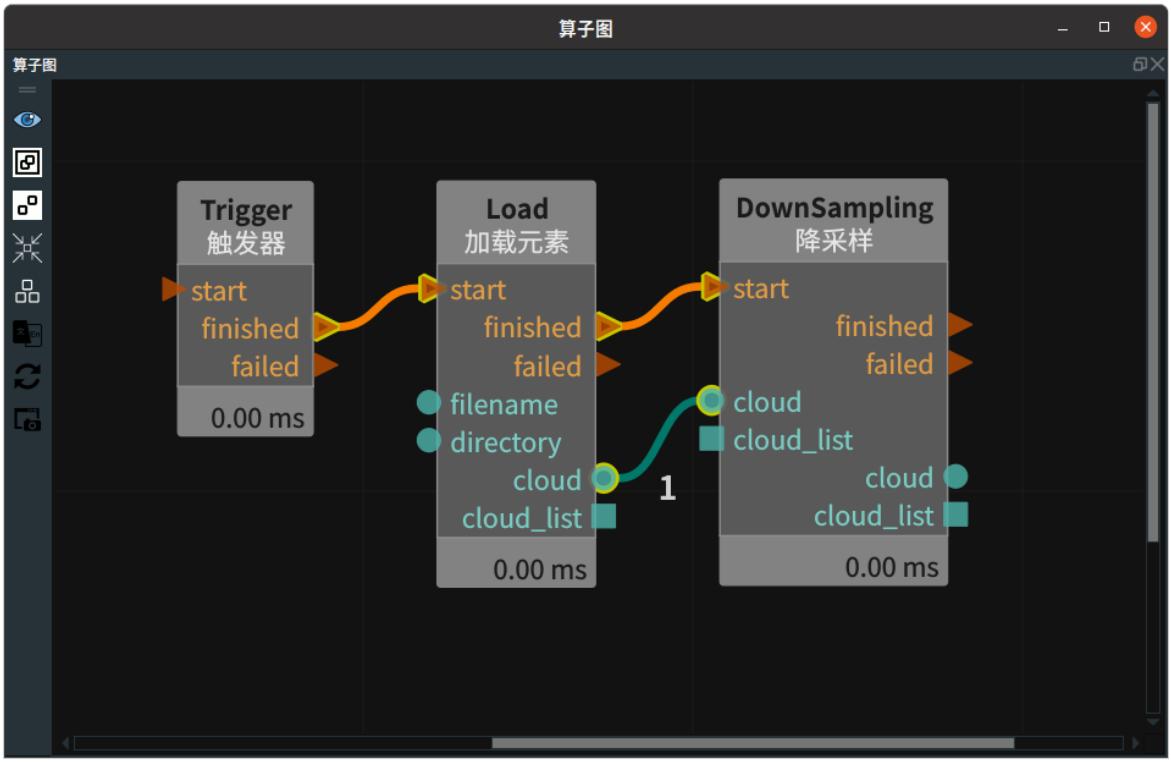
步骤1：算子准备

添加 Trigger、Load、DownSampling 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 点云 → PointCloud
   - 文件 → ••• → 选择点云文件名(*example_data/pointcloud/fruit.pcd*)
   - 点云 →  可视→  -2
2. 设置 DownSampling 算子参数：
   - 类型 → UniformSample
   - 采样规模 → 0.01
   - 点云→  可视→  -2

步骤3：连接算子

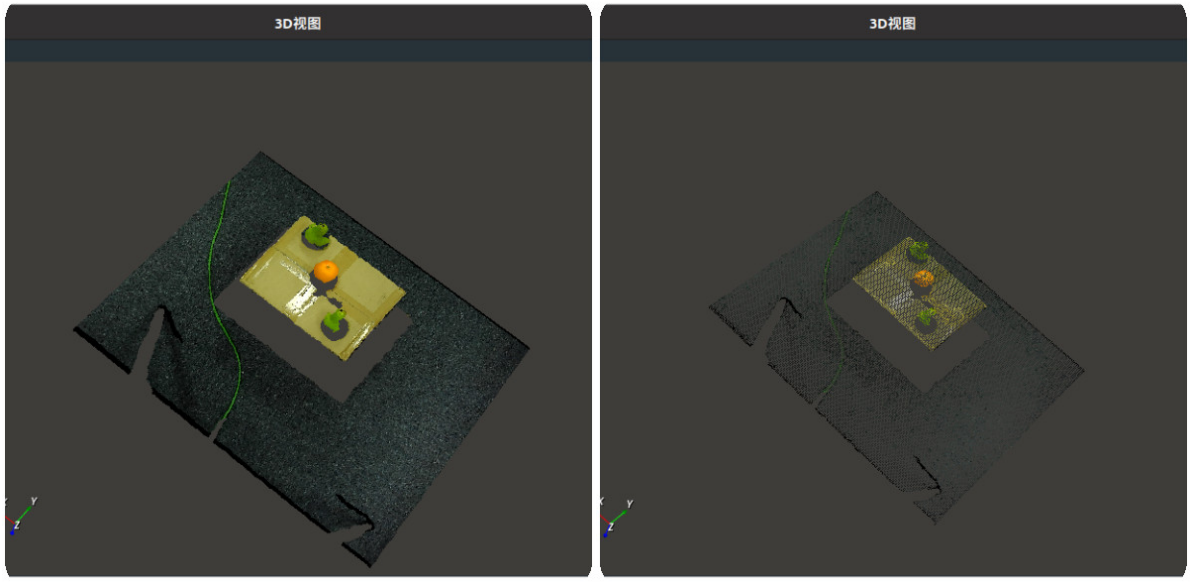


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，3D 视图中，左图为 Load 算子的点云可视化结果，右图为 DownSampling 算子的点云可视化结果。


3D视图
3D视图

CloudProcess 点云处理

CloudProcess 算子用于点云处理，包含 CloudCentroid 、EstimateNormal 、MergeCloud 、NANRemoval 、RadiusOutlierRemoval 。

type	功能
CloudCentroid	用于输出点云的 pose 中心。
EstimateNormal	求出点云相应的法向量。
MergeCloud	将多个点云合并为一个点云输出。
NANRemoval	用于去除 NAN（不是数）值。（点云图中的 NAN 值代表无效或未定义的点，需要注意检测和处理以避免影响点云的处理和分析结果。）
RadiusOutlierRemoval	半径滤波。对点云中的每一个点确定一个半径的邻域，若邻域范围内点数 < min_neighbors ，则认为该点为噪声点，并剔除。

CloudCentroid

将 CloudProcess 点云处理 **类型** 设置为 CloudCentroid ，用于输出点云的 pose 中心。

注意：输出的姿态自动选择为基坐标系而不是沿着目标的长宽方向。

算子参数

- **坐标/pose**：设置点云中心 pose 在 3D 视图中的可视化属性。
  -  打开点云中心 pose 可视化。
  -  关闭点云中心 pose 可视化。
  -  设置点云中心 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1。
- **坐标列表/pose_list**：设置逆 pose 列表在 3D 视图中的可视化属性。值描述与 pose 一致。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **cloud**：
  - 数据类型：PointCloud
  - 输入内容：点云数据
- **cloud_list**：
  - 数据类型：PointCloudList
  - 输入内容：点云列表数据

输出：

- **pose**：
  - 数据类型：Pose
  - 输出内容：点云中心 pose 数据
- **pose_list**：

- 数据类型：PoseList
- 输出内容：点云中心 pose 数据列表

功能演示

使用 CloudProcess 算子中 CloudCentroid 输出加载点云的 pose 中心。

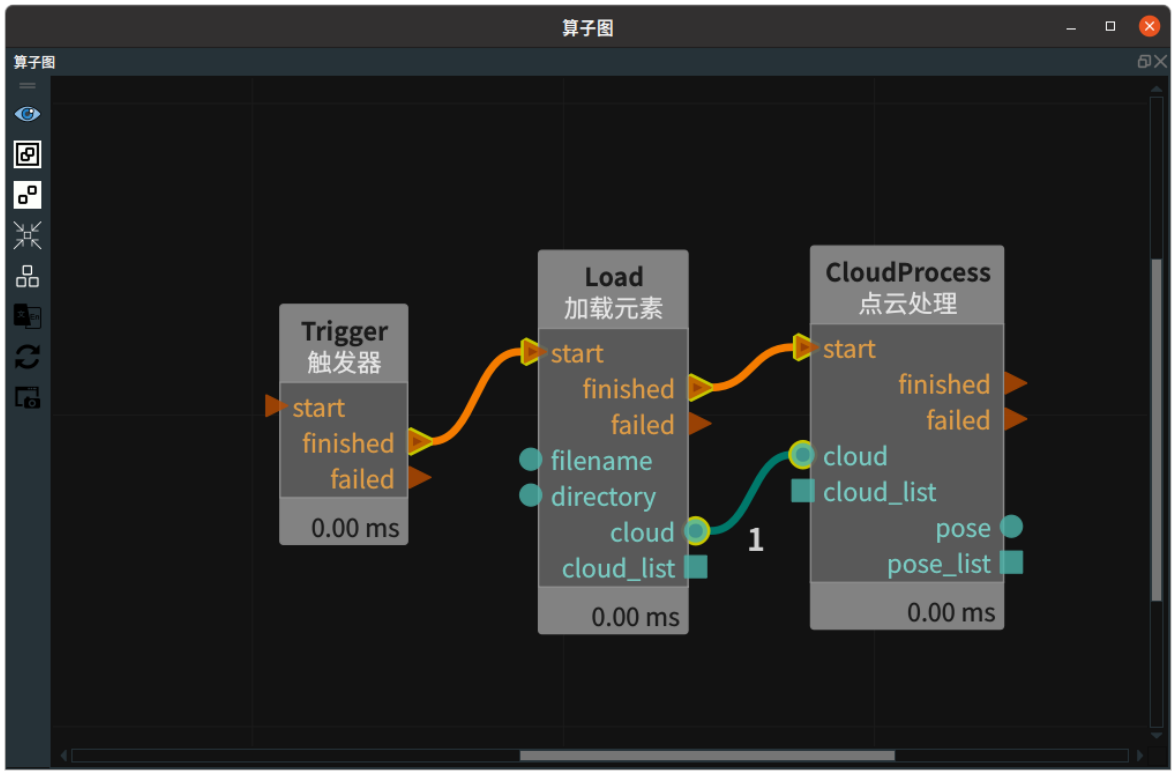
步骤1：算子准备

添加 Trigger 、Load 、CloudProcess 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → PointCloud
   - 文件 → ••• → 选择点云文件名（*example_data/pointcloud/line.pcd*）
   - 点云→ 可视
2. 设置CloudProcess算子参数：
   - 类型→ CloudCentroid
   - 坐标 → 可视

步骤3：连接算子

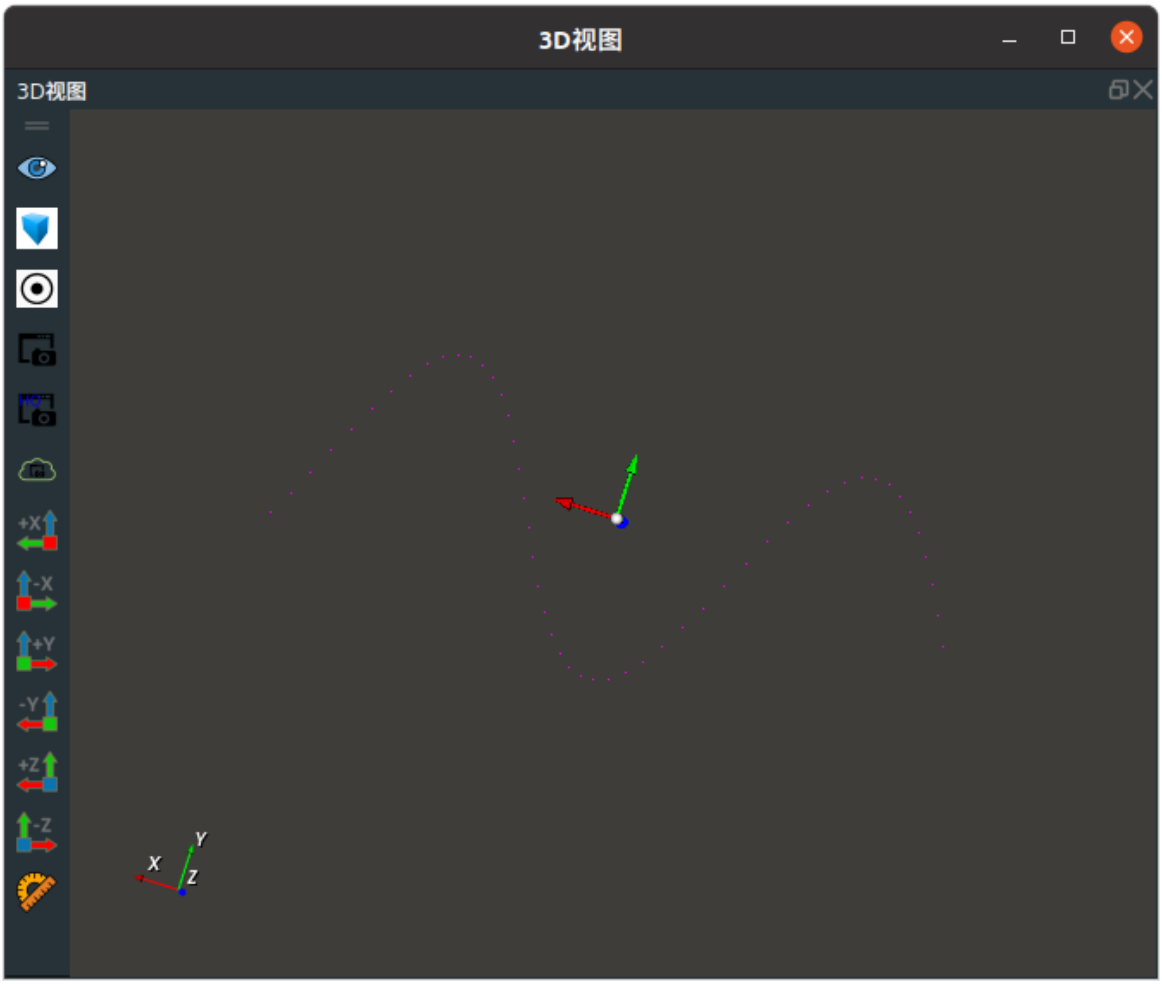


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，在 3D 视图中显示加载的点云及其中心 pose 。



EstimateNormal

将 CloudProcess 点云处理 **类型** 设置为 EstimateNormal ，用于求出点云相应的法向量。

- **搜索半径/search_radius**：该属性表示点云中的点的半径，半径范围内，搜索到多个相邻点拟合出的平面的法向量即当前点的法向量。
- **反转法线/flip_normals** ：法向量反转。
  - True：反转点云法向量的朝向。
  - False ： 点云法向量的朝向正常。
- **点云/cloud**：设置点云相应的法向量在 3D 视图中的可视化属性。
  - 打开点云相应的法向量可视化。
  - 关闭点云相应的法向量可视化。
  - 设置3D视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  - 设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。

数据信号输入输出

输入：

- **cloud**：
  - 数据类型：PointCloud
  - 输入内容：点云数据

输出：

- `cloud`：
  - 数据类型：PointCloud
  - 输出内容：点云数据及其法向量

功能演示

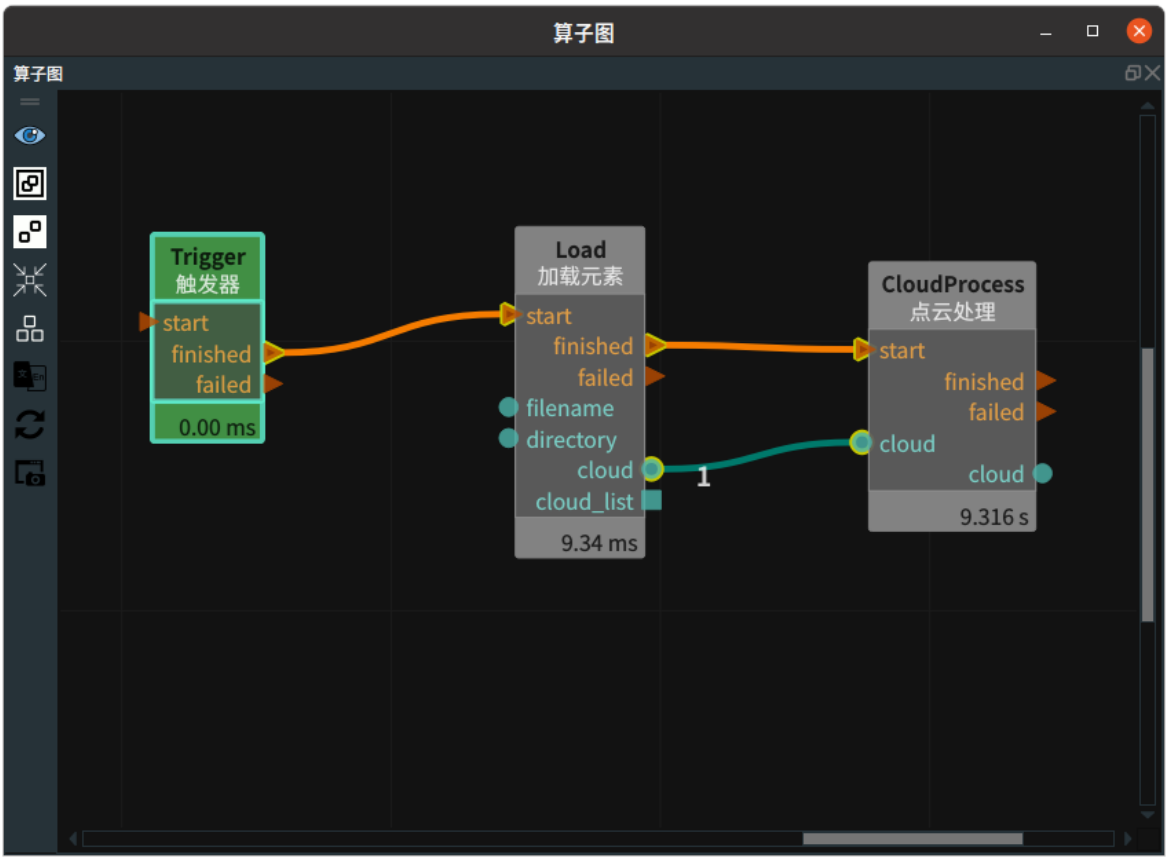
步骤1：算子准备

添加 Trigger 、Load 、CloudProcess 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → PointCloud
   - 文件 → ••• → 选择点云文件名(*example_data/pointcloud/model.pcd*)
2. 设置 CloudProcess 算子参数：
   - 类型 → EstimateNormal

步骤3：连接算子

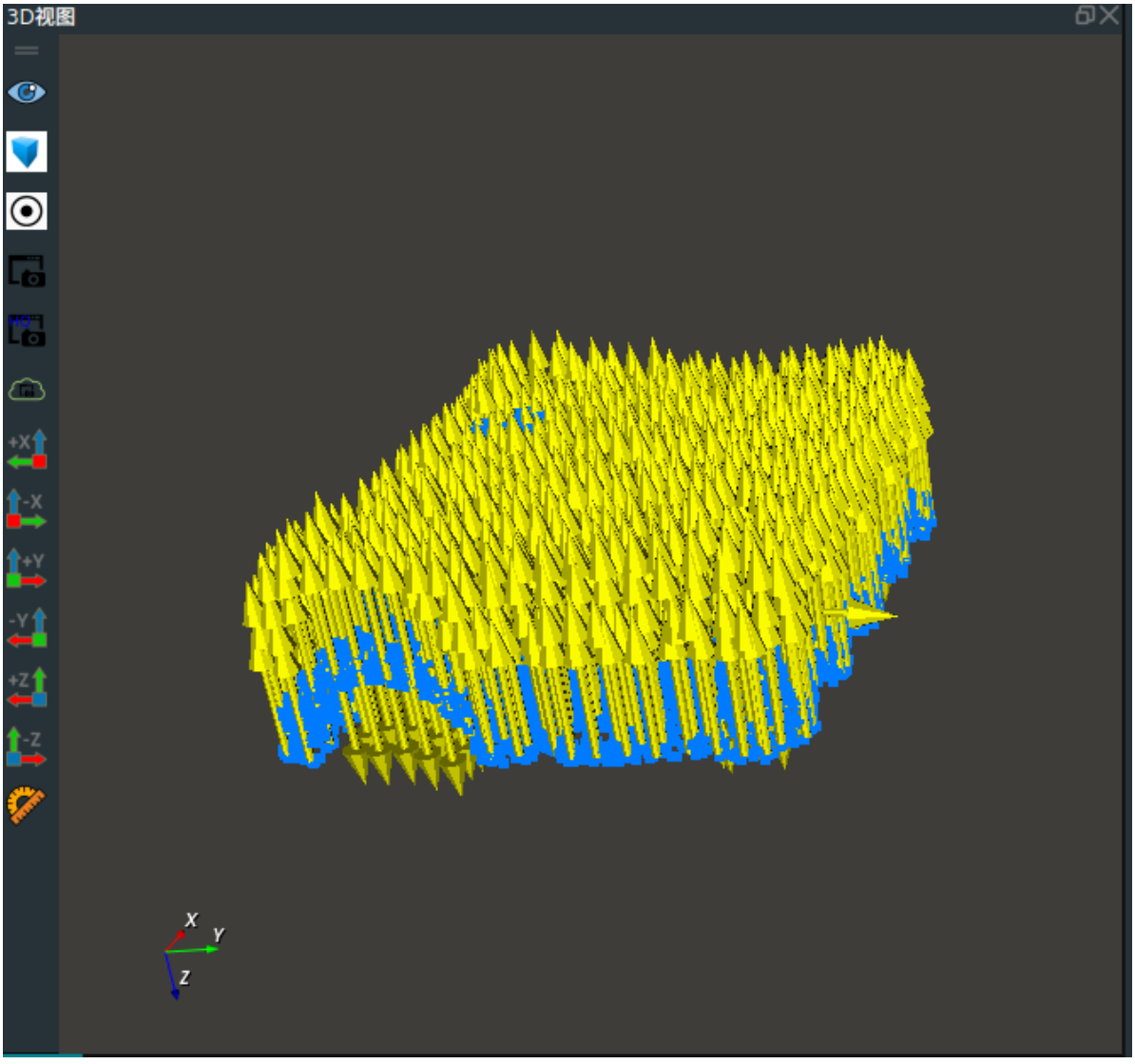


步骤4：运行

1. 点击 RVS 运行按钮，触发 Trigger 算子。
2. 双击3D视图中的点云，显示 POINTCLOUD 面板，勾选“显示法向量”。

运行结果

如下图所示，在 3D 视图中显示加载的点云及其法向量。



MergeCloud

将 CloudProcess 点云处理 **类型** 设置为 MergeCloud ，用于将多个点云合并为一个点云输出。

算子参数

- **输入数量/number_input**：决定该算子的输入端口 cloud 的数量。取值范围：[0,10]。默认值：1。
- **列表输入数量/number_input_list**：决定该算子的输入端口 cloud_list 的数量。取值范围：[0,10]。默认值：0 。
- **点云/cloud**：设置合并后点云在 3D 视图中的可视化属性。
  -  打开合并后点云可视化。
  -  关闭合并后点云可视化。
  -  设置 3D 视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  -  设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **input_0/1**：
  - 数据类型：PointCloud
  - 输入内容：点云数据

输出：

- **cloud**：
  - 数据类型：PointCloud
  - 输出内容：合并后点云数据

功能演示

使用 CloudProcess 算子中 MergeCloud 将加载的两个点云合并成一个点云输出。

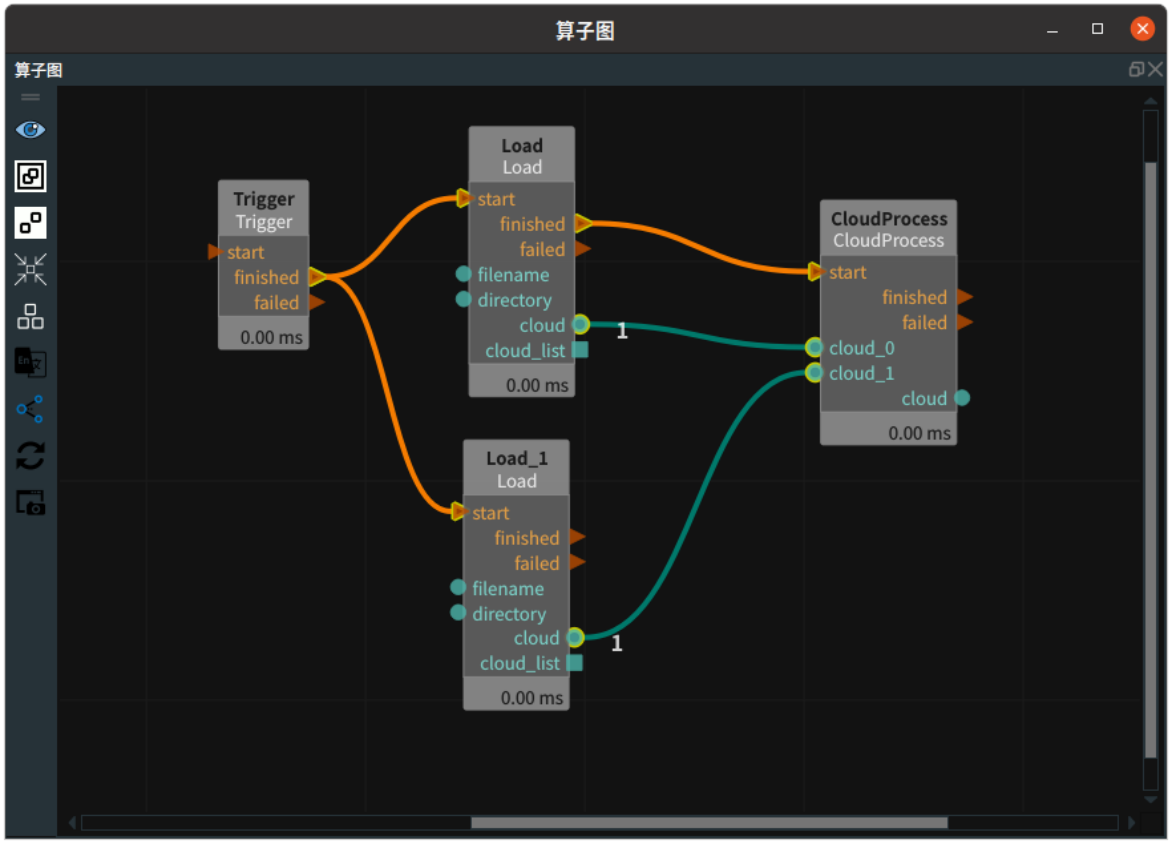
步骤1：算子准备

添加 Trigger、Load（2个）、 CloudProcess 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → PointCloud
   - 文件 → ••• → 选择点云文件名(*example_data/pointcloud/wolf1.pcd*)
2. 设置 Load_1 算子参数：
   - 类型 → PointCloud
   - 文件 → ••• → 选择点云文件名(*example_data/pointcloud/wolf2.pcd*)
3. 设置 CloudProcess 算子参数：
   - 类型 → MergeCloud
   - 文件 →  可视

步骤3：连接算子

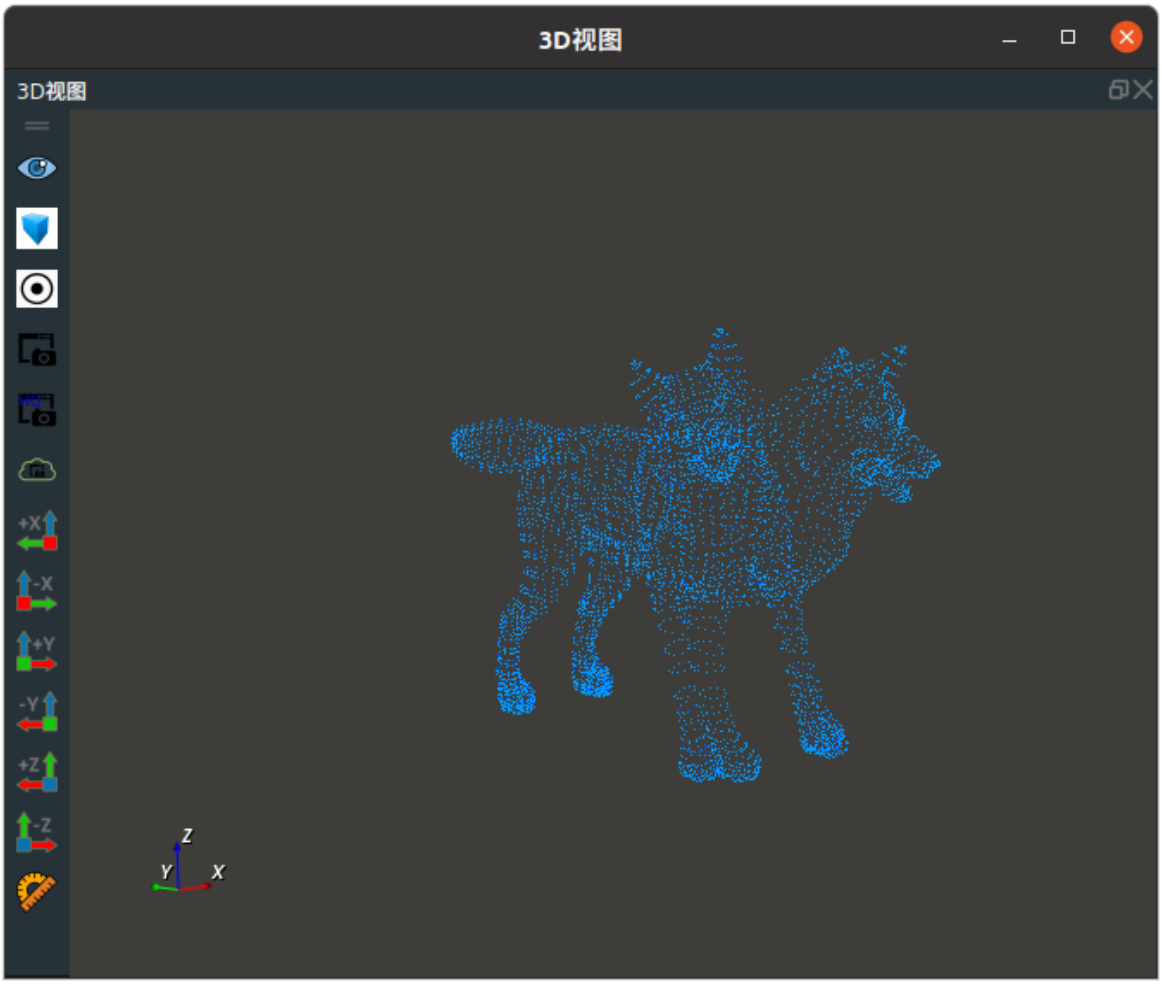


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，3D 视图中显示合并后的点云。



NANRemoval

将 CloudProcess 点云处理 **类型** 设置为 NANRemoval，用于去除 NAN 值。

算子参数

- **点云/cloud**：设置去除NAN值后点云在 3D 视图的可视化属性。
  - 打开去除 NAN 值后点云可视化
  - 关闭去除 NAN 值后点云可视化
  - 设置 3D 视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  - 设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。
- **点云列表/cloud_list**：设置去除 NAN 值后点云列表的可视化属性。值描述与 **点云** 一致。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **cloud**：
  - 数据类型：PointCloud
  - 输入内容：点云数据
- **cloud_list**：
  - 数据类型：PointCloudList

- 输入内容：点云列表数据

输出：

- `cloud`：
  - 数据类型：PointCloud
  - 输出内容：去除NAN值后点云数据
- `cloud_list`：
  - 数据类型：PointCloudList
  - 输出内容：去除 NAN 值后点云列表数据

功能演示

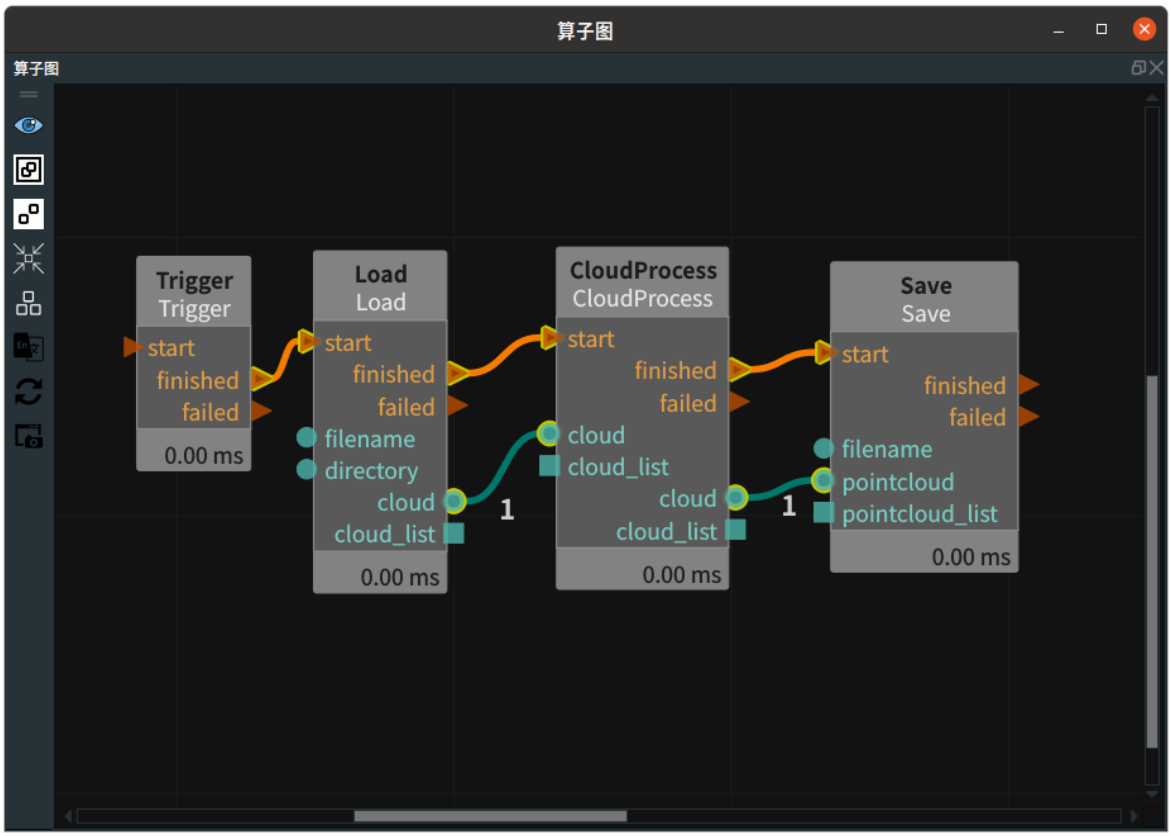
步骤1：算子准备

添加 Trigger 、Load、CloudProcess 、Save 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → PointCloud
   - 文件 → ••• → 选择点云文件名(*example_data/pointcloud/pointcloud_ascii.pcd*) ，此点云为 ascii 形式。
2. 设置 CloudProcess 算子参数：类型 → NANRemoval
3. 设置 Save 算子参数：
   - 类型 → PointCloud
   - format → ascii
   - 文件 → *NAN_pointcloud_ascii.pcd*

步骤3：连接算子

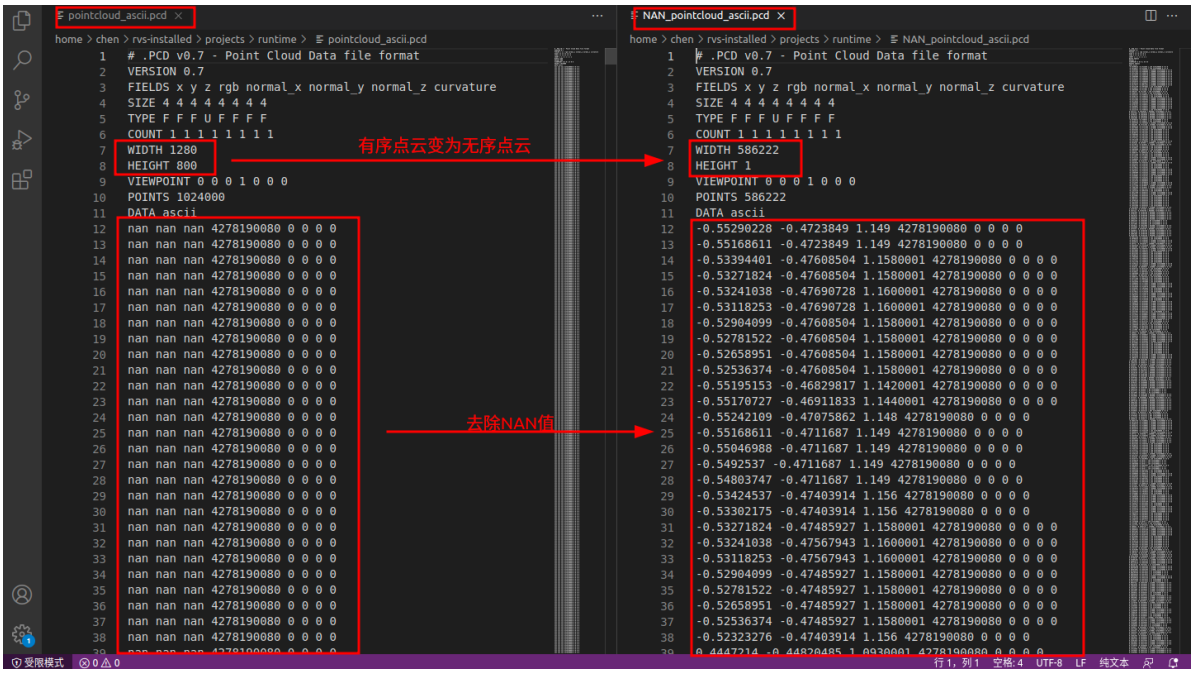


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。保存点云 *NAN_pointcloud_ascii.pcd* 。

运行结果

使用 vscode 编辑器打开两个点云文件进行对比，会发现 NAN_pointcloud_ascii.pcd 点云文件中去除了 NAN 值，并且将有序点云变成了无序点云。



RadiusOutlierRemoval

将 CloudProcess 点云处理 **类型** 设置为 RadiusOutlierRemoval ，用于半径滤波。对点云中的每一个点确定一个半径的邻域，若邻域范围内点数 < min_neighbors ，则认为该点为噪声点，并剔除。

算子参数

- **搜索半径/search_radius**：以点云中的点为圆心设置半径。默认值：0.1 。单位：m。

  注意：该属性值设置与点云的密度和单位有关。如果运行过程中出现算子卡死的情况，建议从小到大逐步调整该参数，并不断测试效果。

- **最小邻居数/min_neighbors**：点半径范围内最少的邻近点。当小于该值时，剔除该点。
- **点云/cloud**：设置半径滤波后点云在3D视图中的可视化属性。
  - 打开半径滤波后点云可视化。
  - 关闭半径滤波后点云可视化。
  - 设置 3D 视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  - 设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。

数据信号输入输出

输入：

- **cloud**：
  - 数据类型：PointCloud
  - 输入内容：点云数据

输出：

- `cloud`：
  - 数据类型：PointCloud
  - 输出内容：半径滤波后点云数据

功能演示

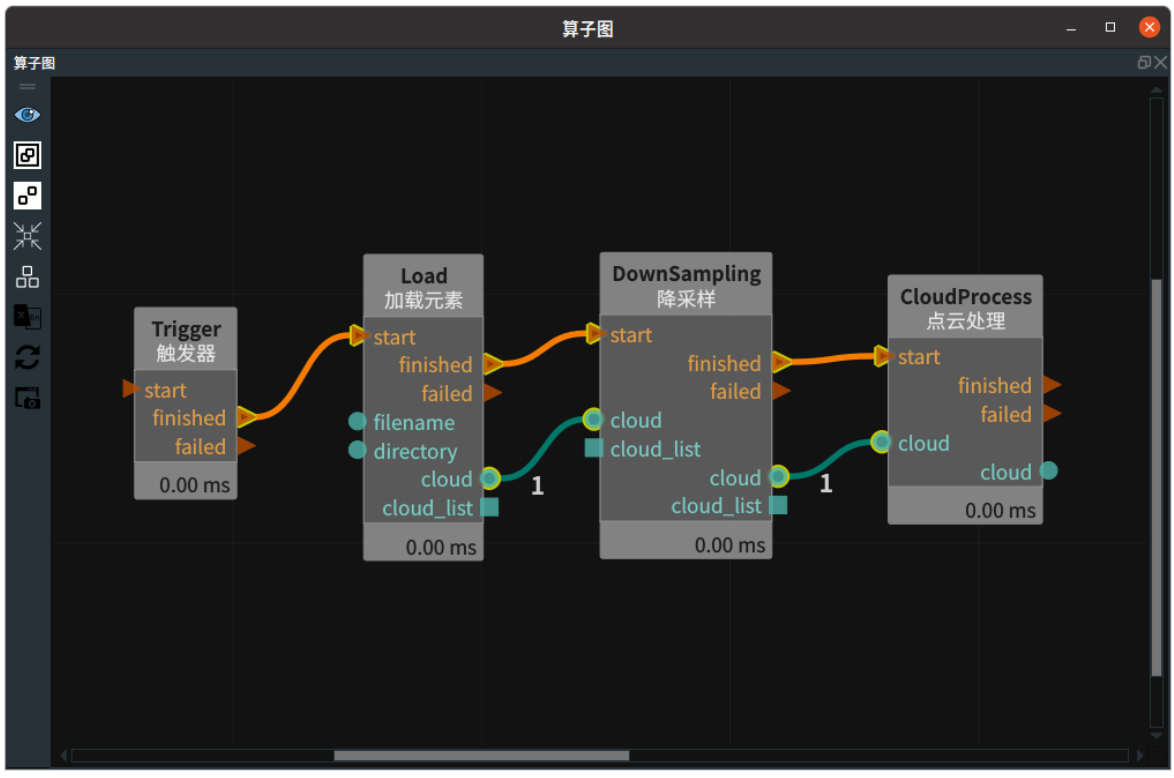
步骤1：算子准备

添加 Trigger 、Load 、DownSampling 、CloudProcess 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → PointCloud
   - 文件 → ••• → 选择点云文件名(*example_data/pointcloud/ClusterExtraction_cloud.pcd*)
2. 设置 DownSampling 算子参数：
   - 类型 → DownSample
   - x 方向采样 → 0.02
   - y 方向采样 → 0.02
   - z 方向采样 → 0.02
   - 点云 →  可视
3. 设置 CloudProcess算 子参数：
   - 类型 → RadiusOutlierRemoval
   - 搜索半径 → 0.05
   - 最小邻居数 → 40
   - 点云 →  可视

步骤3：连接算子

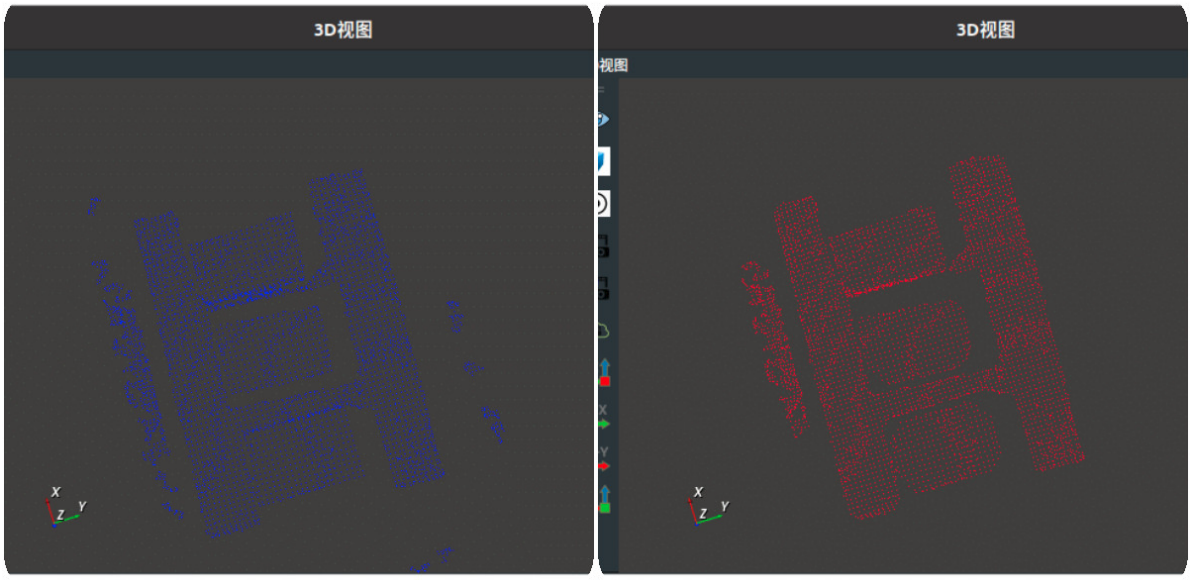


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，3D 视图中，左图为 DownSampling 算子结果，右图为半径滤波后的结果。



FindElement 探查元素

FindElement 算子可以自动查找当前点云的对应的几何图形，并根据点到平面的距离阈值来过滤点云。

type	功能
Plane	用于自动查找当前点云的回归平面。
Cylinder	用于自动查找当前点云中的圆柱体。
Sphere	用于自动查找当前点云中的球体。
Circle	用于自动查找当前点云中的拟合圆。
Line	用于自动查找当前点云中的直线。

Plane

将 FindElement 算子的 **类型** 设置为 Plane ，用于自动查找当前点云的回归平面 。

算子参数

- **最大迭代次数/max_iteration**：最大迭代次数。取值范围：[1,+∞)。默认值：1000次。
- **距离阈值/distance_threshold**：点云过滤的距离阈值。取值范围：(0,+∞)。默认值：0.001 。单位：m。
- **阈值百分比/percentage_threshold**：符合平面点云数占总点云数的比例。默认值：10 。当查找到的平面 percentage 小于该值时， 触发 error 信号。
- **反转法向量沿Z轴/flip_normal_to_z**：确定所找平面 pose z 轴的方向
  - True：调整平面 z 轴与基坐标轴的 z 轴夹角更接近。
  - False：平面 z 轴随机。
- **最小点数/min_points**：平面最小点云数。当输入数值超过平面点云的点数时，触发 error 信号。
- **平面立方体宽度/plane_cube_width**：所查找到的平面类似 cube 的宽。默认值：0.1 。
- **平面立方体高度/plane_cube_height**：所查找到的平面类似 cube 的长。默认值：0.15 。
- **平面立方体深度/plane_cube_depth**：所查找到的平面类似 cube 的高。默认值：0.00001 。
- **点云/cloud**：设置查找出平面的点云在3D视图中的可视化属性。
  -  打开点云可视化。
  -  关闭点云可视化。
  -  设置3D视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  -  设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。
- **坐标/pose**：设置平面点云中心点 pose 在 3D 视图中的可视化属性。
  -  打开平面点云中心点 pose 可视化。
  -  关闭平面点云中心点 pose 可视化。
  -  设置平面点云中心点 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。
- **立方体/cube**：设置平面类似 cube 在 3D 视图中的可视化属性。
  -  打开平面 cube 可视化

    -  关闭平面 cube 可视化
    -  设置平面 cube 的颜色。取值范围：[-2,360] 。默认值：-2 。
    -  设置平面 cube 的透明度。取值范围：[0,1] 。默认值：0.5 。
- **点云列表/cloud_list**：设置查找出的平面点云列表在 3D 视图中的可视化属性。值描述与 **点云** 一致。
- **坐标列表/pose_list**：设置平面点云中心点 pose 列表在 3D 视图中的可视化属性。值描述与**坐标**一致。
- **立方体列表/cube_list**：设置平面 cube 列表在 3D 视图中的可视化属性。值描述与 **立方体** 一致。

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **input_cloud**：
    - 数据类型：PointCloud
    - 输入内容：点云数据
- **input_cloud_list**：
    - 数据类型：PointCloudList
    - 输入内容：点云数据列表

输出：

- **output_cloud**：
    - 数据类型：PointCloud
    - 输出内容：平面查找出的点云数据
- **pose**：
    - 数据类型：Pose
    - 输出内容：点云中心点 pose 数据
- **cube**：
    - 数据类型：Cube
    - 输出内容：平面 cube 数据
- **output_cloud_list**：
    - 数据类型：PointCloudList
    - 输出内容：平面查找出的点云数据列表
- **pose_list**：
    - 数据类型：PoseList
    - 输出内容：点云中心点 pose 数据列表
- **cube_list**：
    - 数据类型：CubeList
    - 输出内容：平面 cube 数据列表

功能演示

使用 FindElement 算子 Plane 查找加载点云的回归平面 。

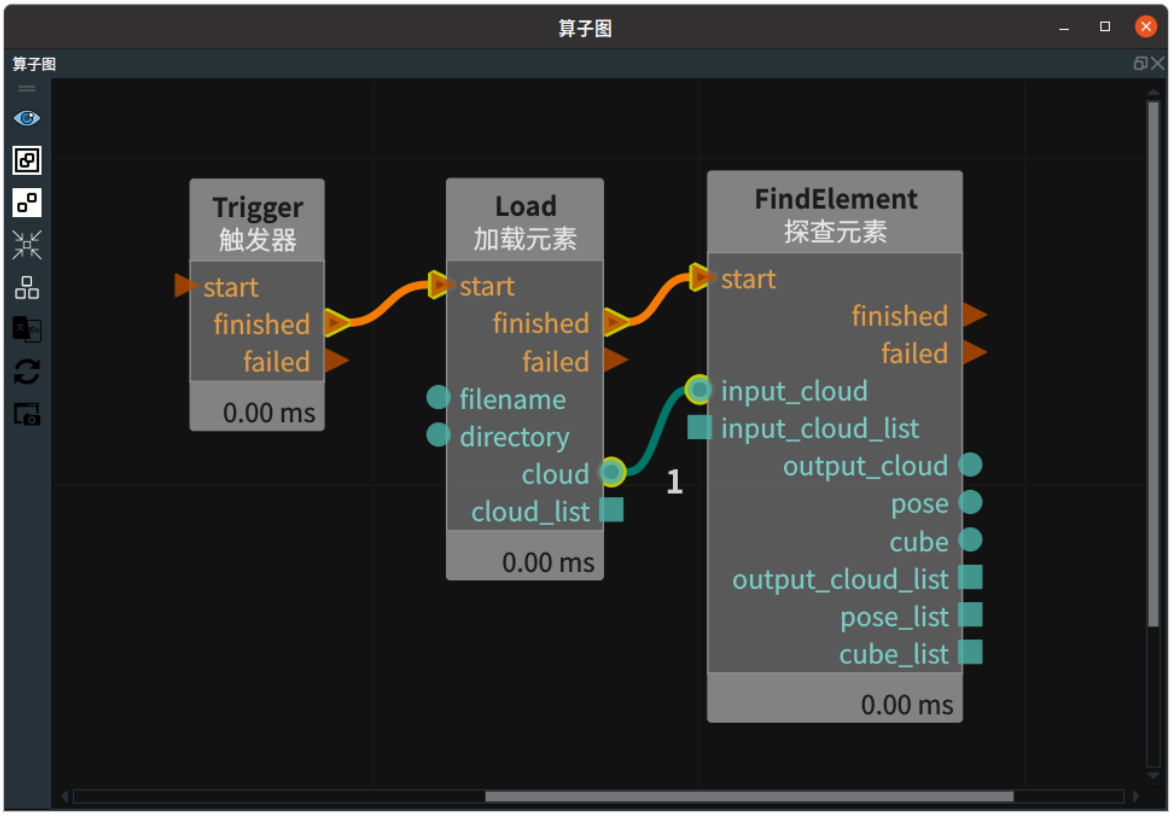
步骤1：算子准备

添加 Trigger 、Load 、FindElement 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 类型 → pointcloud
    - 文件 → ••• → 点云文件名(*example_data/pointcloud/milk.pcd*)
    - 坐标 → 可视
2. 设置 FindElement 算子参数：
    - 类型 → Plane
    - 点云 → 可视
    - 距离阈值 → 0.01

步骤3：连接算子

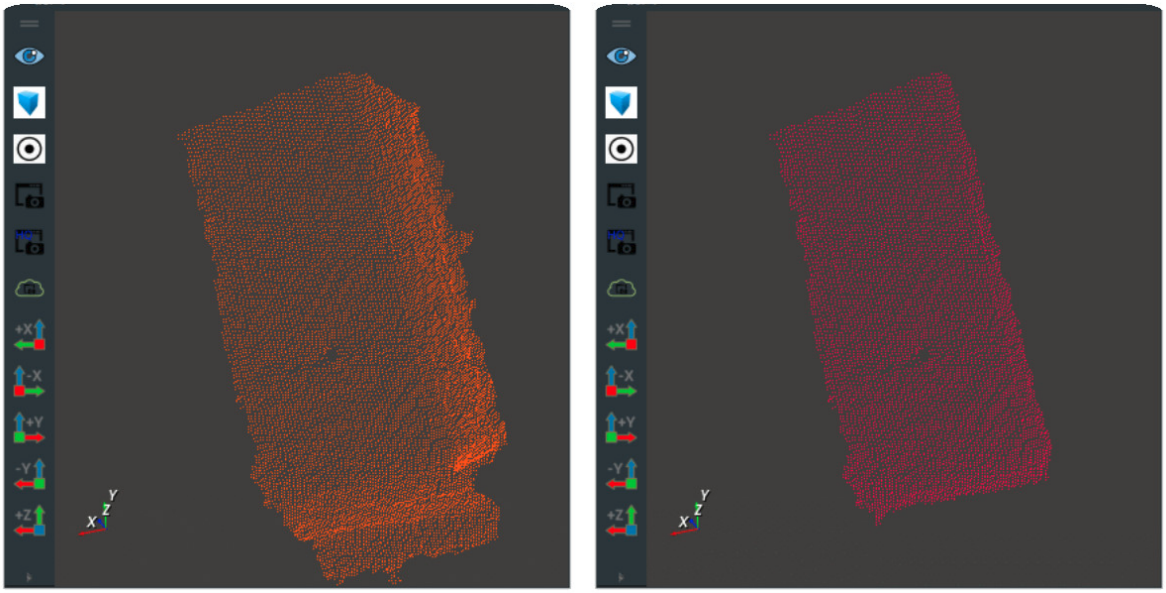


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，左图为 Load 算子加载出的点云，右图为 FindElement 算子 Plane 查找加载点云的回归平面。



Cylinder

将 FindElement 算子的 **类型** 设置为 Cylinder ，用于自动查找当前点云的圆柱体。

算子参数

- **最大迭代次数/max_iteration**：最大迭代次数，默认值：1000 。
- **距离阈值/distance_threshold**：点云过滤的距离阈值。单位：m 。默认值：0.001 。
- **最大半径/max_radius**：圆柱体最大半径。单位：m 。默认值：0.1 。当查找到的圆柱体半径大于该值时， 触发 error 信号。
- **圆柱体长度/cylinder_length**：设置圆柱体的长。单位：m 。默认值：0.05 。
- **点云/cloud**：设置查找出圆柱体点云在 3D 视图中的可视化属性。
  - 打开点云可视化。
  - 关闭点云可视化。
  - 设置 3D 视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  - 设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。
- **坐标/pose**：设置平面点云中心点 pose 在3D视图中的可视化属性。
  - 打开平面点云中心点 pose 可视化。
  - 关闭平面点云中心点 pose 可视化。
  - 设置平面点云中心点 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。
- **圆柱体/cylinder**：设置圆柱体在 3D 视图中的可视化属性。
  - 打开圆柱体可视化。
  - 关闭圆柱体可视化。
  - 设置3D视图中圆柱体的颜色。取值范围：[-2,360] 。默认值：-2 。
- **点云列表/cloud_list**：设置查找出的圆柱体点云列表在 3D 视图中的可视化属性。值描述**点云**一致。
- **坐标列表/pose_list**：设置圆柱体点云中心点 pose 列表在 3D 视图中的可视化属性。值描述 **坐标** 一致。
- **圆柱体列表/cylinder_list**：设置圆柱体列表在 3D 视图中的可视化属性。值描述与**圆柱体**一致。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- `input_cloud`：
  - 数据类型：PointCloud
  - 输入内容：点云数据
- `input_cloud_list`：
  - 数据类型：PointCloudList
  - 输入内容：点云数据列表

输出：

- `output_cloud`：
  - 数据类型：PointCloud
  - 输出内容：查找出的点云数据
- `pose`：
  - 数据类型：Pose
  - 输出内容：点云中心点 pose 数据
- `cylinder`：
  - 数据类型：Cylinder
  - 输出内容：圆柱体数据
- `output_cloud_list`：
  - 数据类型：PointCloudList
  - 输出内容：查找出的点云数据列表
- `pose_list`：
  - 数据类型：PoseList
  - 输出内容：点云中心点 pose 数据列表
- `cylinder_list`：
  - 数据类型：CylinderList
  - 输出内容：圆柱体数据列表

功能演示

使用 FindElement 算子 Cylinder 查找加载点云中的圆柱 。

步骤1：算子准备

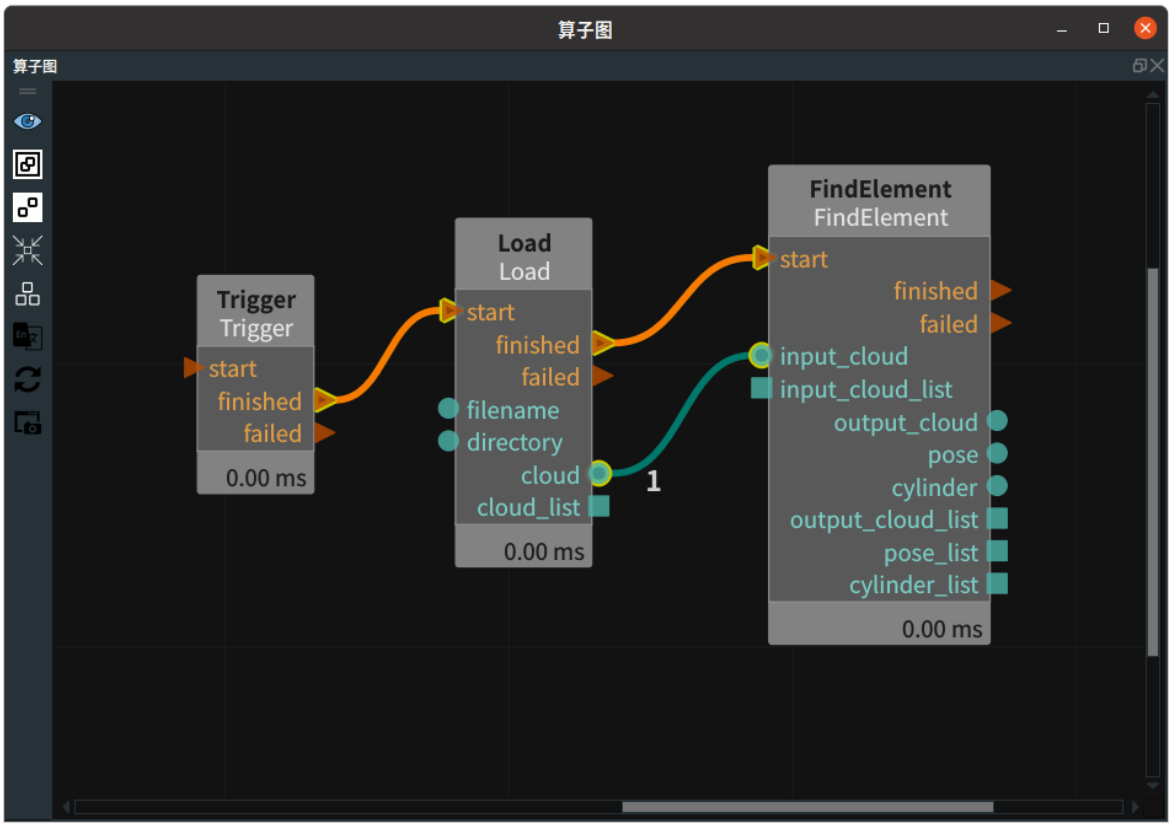
添加 Trigger 、Load 、FindElement 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 →PointCloud
   - 文件 → ••• → 点云文件名 (*example_data/pointcloud/cylinder.pcd*)
   - 点云 →  可视→  -2
   - 坐标 →  可视
2. 设置 FindElement 算子参数：
   - 类型 → Cylinder

- 最大迭代次数 → 1000
- 距离阈值 → 0.005
- 最大半径 → 0.1
- 圆柱体长度 → 0.2
- 点云 → 可视
- 坐标 → 可视
- 圆柱体 → 可视

步骤3：连接算子

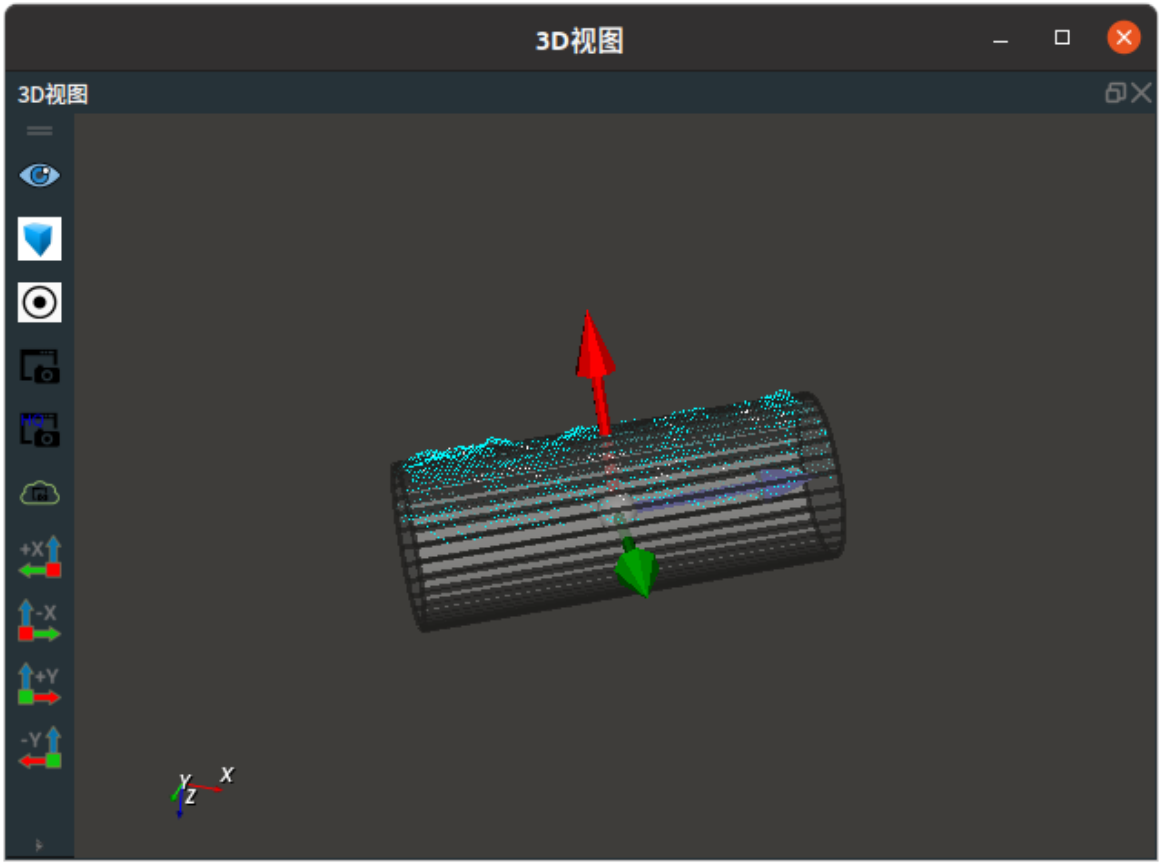


步骤四：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，3D 视图中蓝色部分和圆柱体为 FindElement 的结果。



Sphere

将 FindElement 算子的 **类型** 设置为 Sphere ，用于自动查找当前点云的球体。

算子参数

- **最大迭代次数/max_iteration**：最大迭代次数。默认值：1000 。
- **距离阈值/distance_threshold**：点云过滤的距离阈值。单位：m ，默认值：0.01 。
- **阈值百分比/percentage_threshold**：符合查找出点云数占总点云数的比例。默认值：50 。
- **球体半径最小值/sphere_radius_min**：球体最小半径。默认值：0.05。当查找到的球体半径小于该值时， 触发 error 信号。单位：m 。
- **球体半径最大值/sphere_radius_max**：球体最大半径。默认值：0.15 。当查找到的球体半径大于该值时， 触发 error 信号。单位：m 。
- **点云/cloud**：设置查找出球体点云在 3D 视图中的可视化属性。
  - 打开点云可视化。
  - 关闭点云可视化。
  - 设置3D视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  - 设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。
- **坐标/pose**：设置平面点云中心点 pose 在3D视图中的可视化属性。
  - 打开平面点云中心点 pose 可视化。
  - 关闭平面点云中心点 pose 可视化。
  - 设置平面点云中心点 pose 的尺寸大小 。取值范围：[0.001,10] 。默认值：0.1 。
- **球体/sphere**：设置球体在3D视图中的可视化属性。
  - 打开球体可视化 。

    -  关闭球体体可视化 。
    -  设置3D视图中球体的颜色。取值范围：[-2,360] 。默认值：-2 。
    -  设置平面立方球体的透明度。取值范围：[0,1] 。默认值：0.5 。
- **点云列表/cloud_list**：设置查找出的球体点云列表可视化属性。值描述与 **点云** 一致。
- **坐标列表/pose_list**：设置球体点云中心点 pose 列表可视化属性。值描述与 **坐标** 一致。
- **球体列表/sphere_list**：设置球体列表可视化属性。值描述与 **球体** 一致。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **input_cloud**：
    - 数据类型：PointCloud
    - 输入内容：点云数据
- **input_cloud_list**：
    - 数据类型：PointCloudList
    - 输入内容：点云数据列表

输出：

- **output_cloud**：
    - 数据类型：PointCloud
    - 输出内容：平面查找出的点云数据
- **pose**：
    - 数据类型：Pose
    - 输出内容：点云中心点 pose 数据
- **sphere**：
    - 数据类型：Sphere
    - 输出内容：sphere数据
- **output_cloud_list**：
    - 数据类型：PointCloudList
    - 输出内容：平面查找出的点云数据列表
- **pose_list**：
    - 数据类型：PoseList
    - 输出内容：点云中心点 pose 数据列表
- **sphere_list**：
    - 数据类型：SphereList
    - 输出内容：sphere数据列表

功能演示

使用 FindElement 算子 Sphere 查找加载点云中的球体 。

步骤1：算子准备

添加 Trigger 、Load 、Emit 、CloudSegment 、FindElement 算子至算子图。

步骤2：设置算子参数

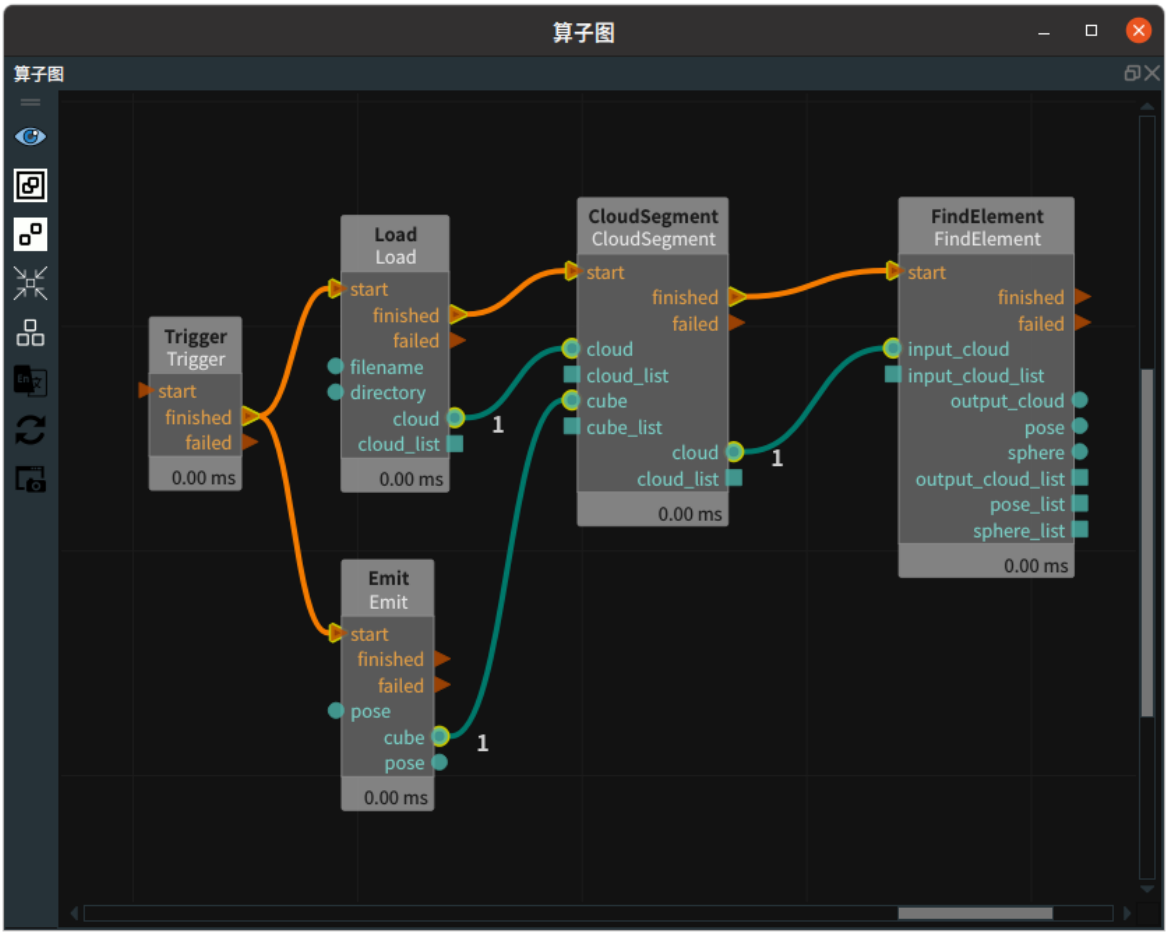
1. 设置 Load 算子参数：

- 类型 → pointcloud
- 文件 → ••• → 点云文件名(*example_data/pointcloud/sphere.pcd*)
- 点云 →  可视→  -2

2. 设置 Emit 算子参数：
   - 类型 → Cube
   - 坐标 → 0 -0.000457 1.10531 -0.110497 0 0
   - 宽度 → 1.6
   - 高度 → 1
   - 深度 → 1
3. 设置 CloudSegment 算子参数：

   说明：该算子用于去除背景点云。
   - 类型 → Cropboxsegment
   - 模式 → outside_points
4. 设置 FindElement 算子参数：
   - 类型 → Sphere
   - 最大迭代次数 → 1000
   - 距离阈值 → 0.01
   - 阈值百分比 → 25
   - 球体半径最小值 → 0.03
   - 球体半径最大值 → 0.05
   - 点云→  可视
   - 球体→  可视

步骤3：连接算子

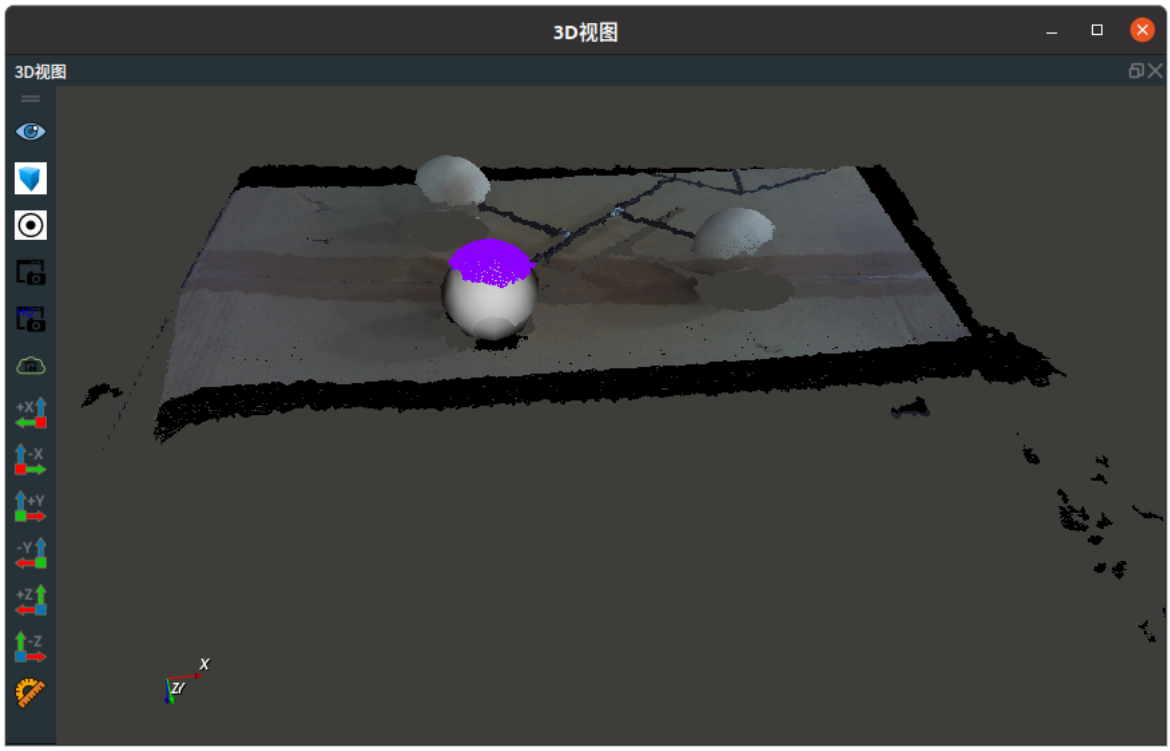


步骤四：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，3D 视图中紫色部分和球体为 FindElement 的结果。



Circle

将 FindElement 算子的 **类型** 设置为 Circle ，用于自动查找当前点云的拟合圆。圆心的朝向与 RVS 坐标系的 Z 轴相反。

算子参数

- **最大迭代次数/max_iteration**：最大迭代次数，默认为1000次。
- **距离阈值/distance_threshold**：单个点云与拟合圆的距离阈值，高于阈值的点不参与最终拟合。取值范围：(0,+∞)。默认值：0.01。单位：m。
- **点云/cloud**：设置查找出平面的点云在 3D 视图中的可视化属性。
  - 打开点云可视化。
  - 关闭点云可视化。
  - 设置3D视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  - 设置点云中点的尺寸。取值范围：[0,50] 。默认值：1 。
- **坐标/pose**：设置拟合圆中心点 pose 在 3D 视图中的可视化属性。
  - 打开拟合圆中心点 pose 可视化。
  - 关闭拟合圆中心点 pose 可视化。
  - 设置拟合圆中心点 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。
- **圆圈/circle**：设置拟合圆在3D视图中的可视化属性。
  - 打开拟合圆可视化。

    -  关闭拟合圆可视化。
    -  设置拟合圆的颜色。取值范围：[-2,360] 。默认值：-2 。
- **点云列表/cloud_list**：设置查找出平面的点云列表在 3D 视图中的可视化属性。值描述与 **点云** 一致。
- **坐标列表/pose_list**：设置拟合圆中心点 pose 列表在 3D 视图中的可视化属性。值描述与**坐标** 一致。
- **圆圈列表/circle_list**：设置拟合圆列表在 3D 视图中的可视化属性。值描述与**圆圈** 一致。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **cloud**：
    - 数据类型：PointCloud
    - 输入内容：点云数据
- **cloud_list**：
    - 数据类型：PointCloudList
    - 输入内容：点云数据列表

输出：

- **cloud**：
    - 数据类型：PointCloud
    - 输出内容：拟合圆所用的最终点云
- **pose**：
    - 数据类型：Pose
    - 输出内容：拟合圆的中心点位姿，z 轴为圆平面法向量
- **Circle**：
    - 数据类型：Circle
    - 输出内容：拟合圆
- **cloud_list**：
    - 数据类型：PointCloudList
    - 输出内容：拟合圆所用的最终点云列表，点云在列表中的位置同下述列表保持一致
- **pose_list**：
    - 数据类型：PoseList
    - 输出内容：点云中心点位姿数据列表，z轴为圆平面法向量
- **circle_list**：
    - 数据类型：CircleList
    - 输出内容：拟合圆列表

功能演示

使用 FindElement 算子中 Circle 自动查找加载点云的拟合圆。

步骤1：算子准备

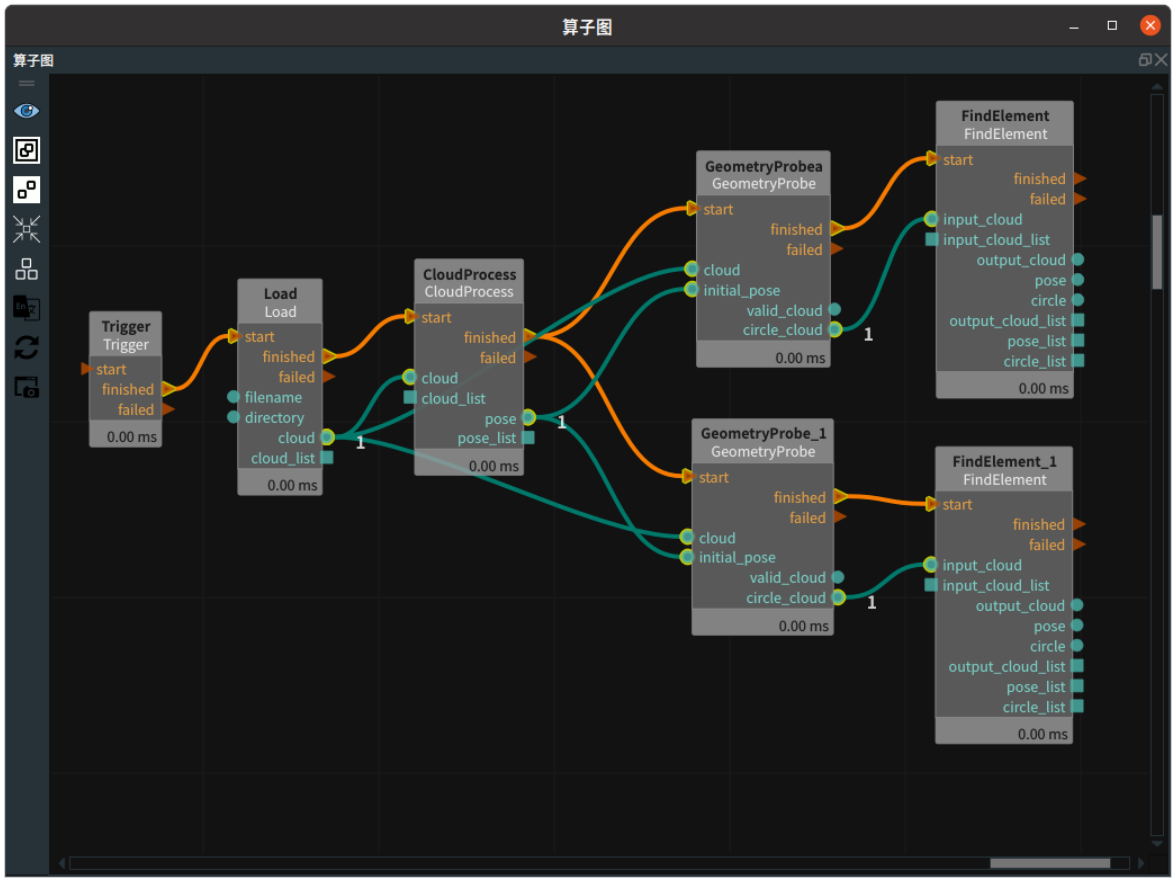
添加 Trigger 、Load 、CloudProcess 、GeometryProbe （2个） 、FindElement（2个） 、 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 类型 → pointcloud
    - 文件 → ••• → 选择点云文件名(*example_data/pointcloud/circle_crop.pcd*)
    - 点云 →  可视
2. 设置 CloudProcess 算子参数：
    - 类型 → CloudCentroid
3. 设置 GeometryProbe 算子参数：
    - 类型 → Circle
    - 探测方向 → external
    - 圆锥体顶角度→ 70
    - 有效点百分比数→ 0.05
    - 步进值 → 5
    - 圆圈点云→  可视
4. 设置 GeometryProbe_1 算子参数：
    - 类型 → Circle
    - 探测方向 → internal
    - 圆锥体顶角度→ 45
    - 有效点百分比数→ 0.05
    - 步进值 → 5
    - 圆圈点云→  可视
5. 设置 FindElement 算子参数：
    - 类型 → Circle
    - 最大迭代次数 → 1000
    - 距离阈值 → 0.001

      说明：由于此时案例所用的点云密度很高，所以这里设置的阈值参数很小。实际使用图漾相机时，设置到 0.001 到 0.01 之间比较合适。
    - 坐标 →  可视
    - 圆圈 →  可视
6. 设置 FindElement_1 算子参数：
    - 类型 → Circle
    - 最大迭代次数 → 1000
    - 距离阈值 → 0.0005
    - 坐标 →  可视 →  0.1
    - 圆圈 →  可视

步骤3：连接算子

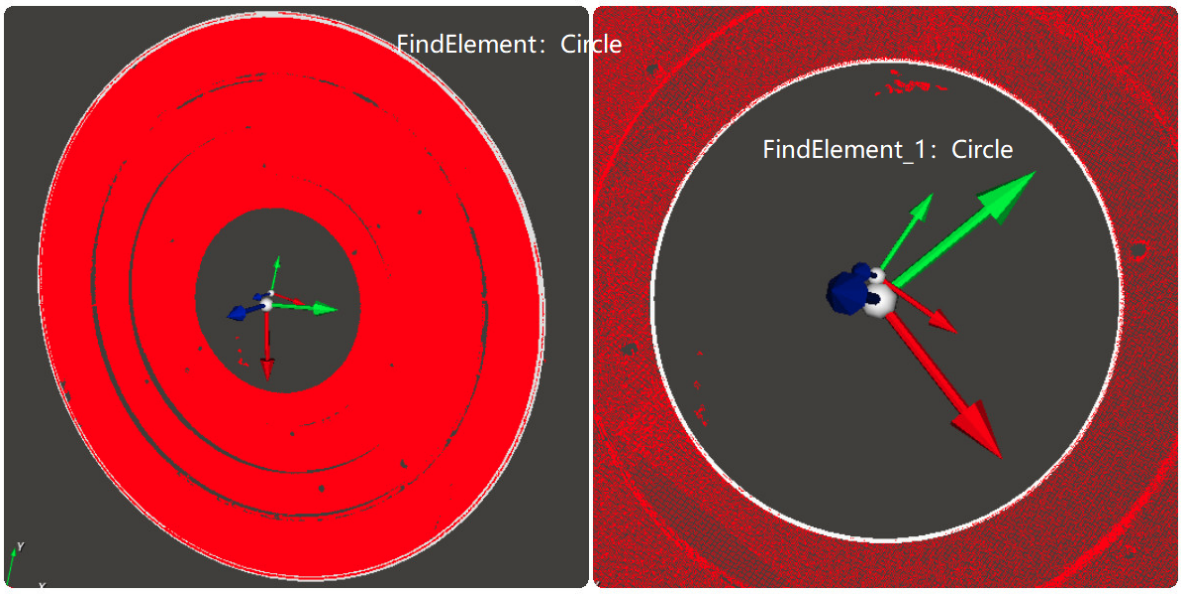


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

1. 结果如下图所示，比例较小的 pose 为 CloudProcess 算子的 pose 可视化结果，比例较大的 pose 为 FindElement 算子的 pose 参数的可视化结果。
2. 左图白色圆圈为 FindElement 算子中的 Circle 可视化结果，右图白色圆圈为 FindElement_1 算子的 Circle 可视化结果。



Line

将 FindElement 算子的 **类型** 设置为 Line，用于自动查找当前点云的拟合直线。线段 P1与 P2 的朝向 RVS 坐标系 X 轴正方向，P1_X < P2_X。

算子参数

- **最大迭代次数/max_iteration**：最大迭代次数，默认值。1000 。
- **距离阈值/distance_threshold**：单个点云与拟合直线的距离阈值，高于阈值的点不参与最终拟合。取值范围：(0,+∞)。默认值：0.003 。单位：m 。
- **线段长度/line_length**：最终拟合直线的显示长度。
- **点云/cloud**：设置拟合直线所用点云在 3D 视图中的可视化属性。
  -  打开点云可视化。
  -  关闭点云可视化。
  -  设置3D视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  -  设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。
- **线段/line**：设置拟合直线的可视化在 3D 视图中的可视化属性。
  -  打开拟合直线可视化。
  -  关闭拟合直线可视化。
  -  设置拟合直线的颜色。取值范围：[-2,360] 。默认值：60 。
  -  设置拟合直线的线宽。取值范围：[1,50] 。默认值：1 。
- **点云列表/cloud_list**：设置拟合直线所用点云列表在 3D 视图中的可视化属性。值描述与 **点云** 一致。
- **线段列表/line_list**：设置拟合直线列表的可视化在 3D 视图中的可视化属性。值描述与 **线段** 一致。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **cloud**：
  - 数据类型：PointCloud
  - 输入内容：点云数据
- **cloud_list**：
  - 数据类型：PointCloudList
  - 输入内容：点云数据列表

输出：

- **cloud**：
  - 数据类型：PointCloud
  - 输出内容：拟合直线所用的最终点云
- **line**：
  - 数据类型：Line
  - 输出内容：拟合直线
- **cloud_list**：
  - 数据类型：PointCloudList

    - 输出内容：拟合圆所用的最终点云列表，点云在列表中的位置同下述列表保持一致
- `line_list`：
    - 数据类型：LineList
    - 输出内容：拟合直线列表

功能演示

使用 FindElement 算子中 Line 自动查找加载点云的拟合直线。

步骤1：算子准备

添加 Trigger 、Load 、Emit 、LineOperation 、GeometryProbe （2个）、FindElement （2个）、WaitAllTriggers 算子至算子图。

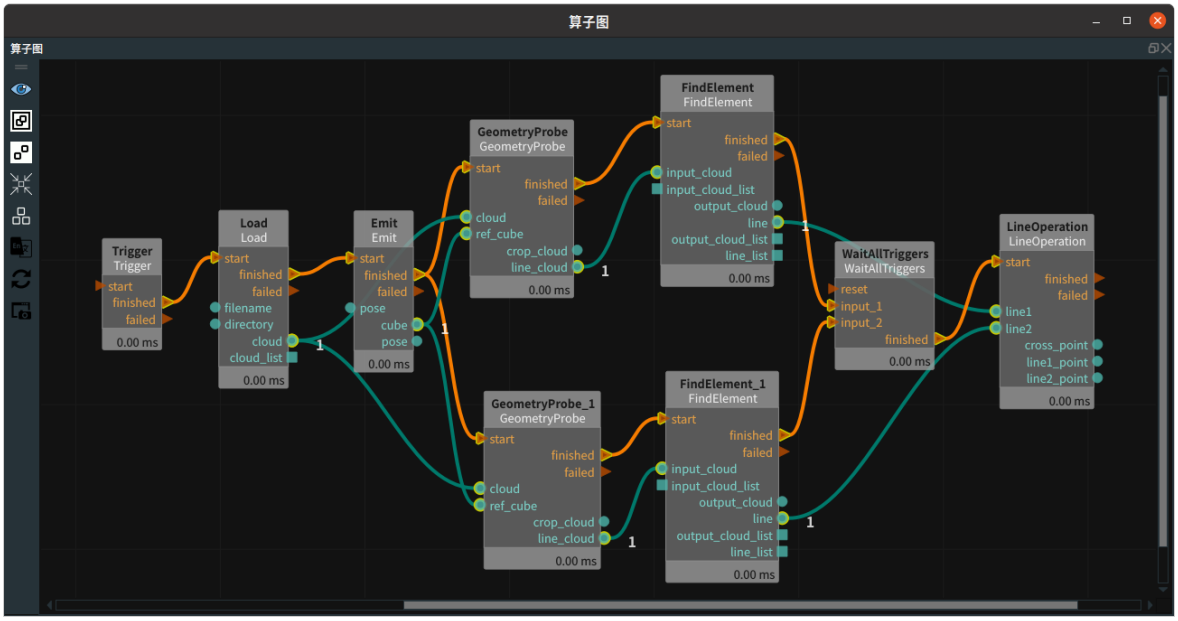
步骤2：设置算子参数

1. 设置 Emit算子参数：
    - 类型 → cube
    - 坐标 → 0.199179 0.26519 0.092586 2.61142 -2.82994 3.07943
    - 宽度 → 0.02
    - 高度 → 0.01
    - 深度 → 0.005
2. 设置 Load 算子参数：
    - 类型 → PointCloud
    - 文件 → ••• → 选择点云文件名(*example_data/pointcloud/cloud.pcd*)
3. 设置 FindElement 算子参数：
    - 类型 → Line
    - 最大迭代次数 → 1000
    - 距离阈值 → 0.0001
    - 线段长度 → 0.1
    - 线段 → 可视
4. 设置 FindElement_1 算子参数：
    - 类型 → Line
    - 最大迭代次数 → 1000
    - 距离阈值 → 0.0001
    - 线段长度 → 0.1
    - 线段 → 可视
5. 设置 GeometryProbe 算子参数
    - 类型 →Line
    - 探测方向 → y-
    - 圆锥体顶角度 → 45
    - 有效点百分比数 → 10
    - 步进值 → 5
    - 有效点云 → 可视 → 3
    - 线段点云 → 可视 → 3
6. 设置 GeometryProbe_1 算子参数
    - 类型 →Line
    - 探测方向 → x-
    - 圆锥体顶角度 → 45

- 有效点百分比数 → 10
- 步进值 → 5
- 有效点云 → 可视 → 3
- 线段点云→ 可视 → 3

7. 设置 LineOperation 算子参数：类型 →LineCrossPoint

步骤3：连接算子

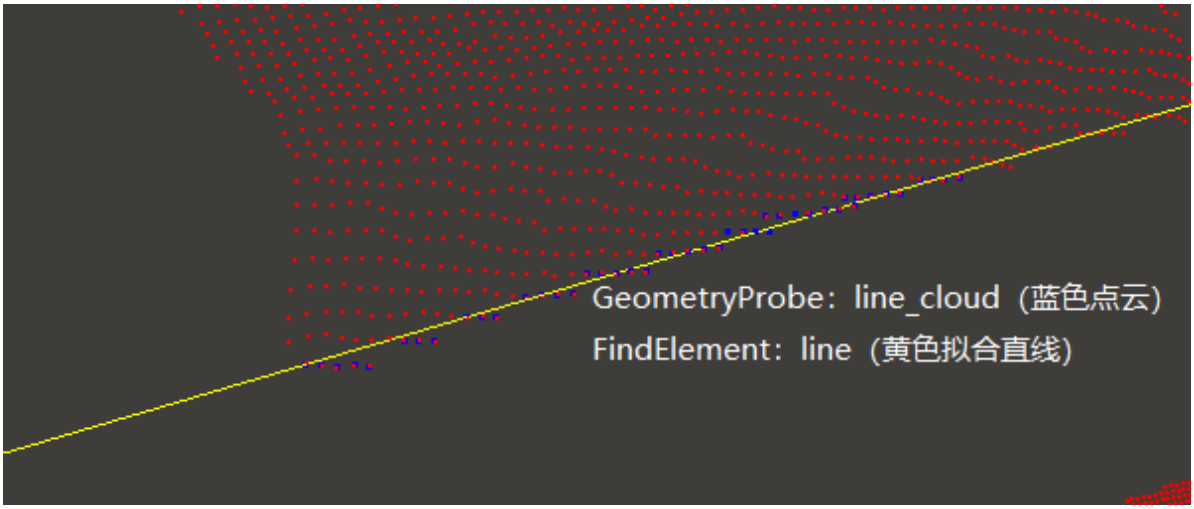


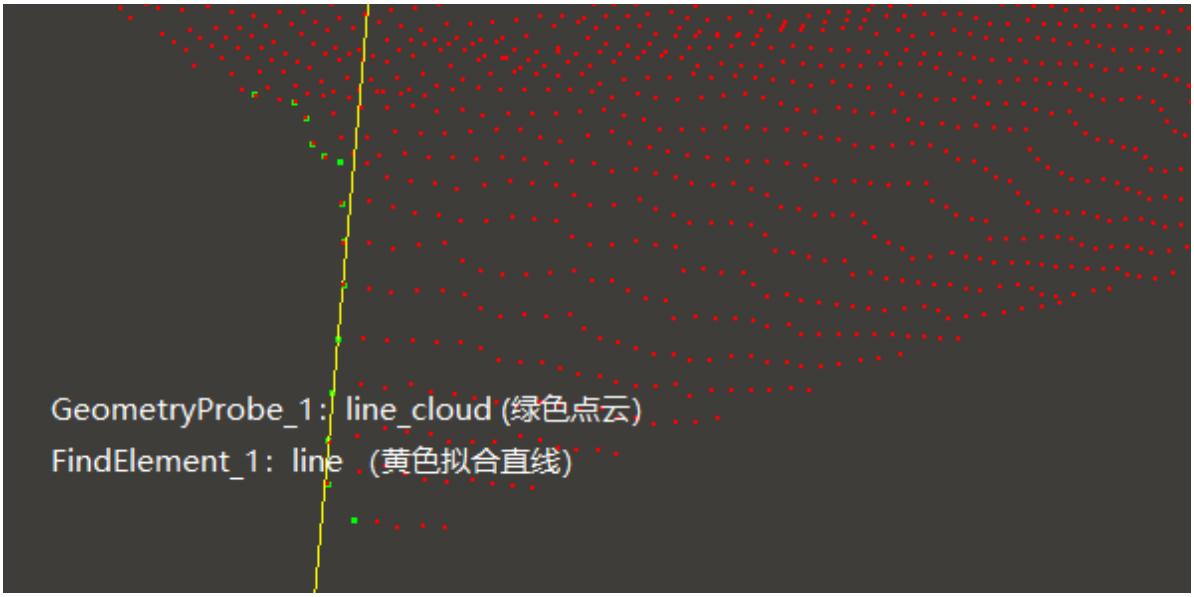
步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

结果如下图所示，蓝色/绿色点云为 GeometryProbe 算子的 line_cloud 可视化结果，黄色拟合直线为 FindElement 算子的 line 参数的可视化结果 。




GeometryProbe_1：line_cloud (绿色点云)
FindElement_1：line (黄色拟合直线)

SortList 列表排序

SortList 算子用于列表排序，该算子按 Weight_x * x + weight_y * y + weight_z * z 计算结果从小到大排序。适用于 Pose 、Cube 、PointCloud 。

type	功能
Pose	pose 列表排序。
Cube	立方体列表排序。根据立方体中心点 pose 进行排序。
PointCloud	点云列表排序。根据点云质心 pose 进行排序。

- **模式/mode**：排序后列表输出模式。默认值：LinearSort。
  - LinearSort：将输入数据列表按照权重排序输出。
  - ReverseIndex：将输入数据列表倒序输出。

Pose

将 SortList 算子的 **类型** 属性选择 Pose ，用于 pose 列表排序。

算子参数

- **X权重/weight_x**：x 轴 pose 权重。
- **Y权重/weight_y**：y 轴 pose 权重。
- **Z权重/weight_z**：z 轴 pose 权重。
- **坐标列表/pose_list**：设置排序后 pose 列表在 3D 视图中的可视化属性。
  -  打开排序后 pose 列表可视化。
  -  关闭排序后 pose 列表可视化。
  -  设置排序后 pose 列表的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。
- **索引列表/index_list**：设置排序后 pose 索引的曝光属性。可与交互面板中输出工具“表格”进行绑定。
  -  打开曝光。
  -  关闭曝光。

数据信号输入输出

- **poselist**：
  - 数据类型：PoseList
  - 输入内容： pose 列表数据

输出：

- **poselist**：

- 数据类型：PoseList
- 输出内容：排序 pose 列表数据

- `index_list`：
  - 数据类型：String
  - 输出内容：排序后索引列表

功能演示

使用 SortList 算子中 Pose 对加载的多个 pose 进行不同方式的排序。

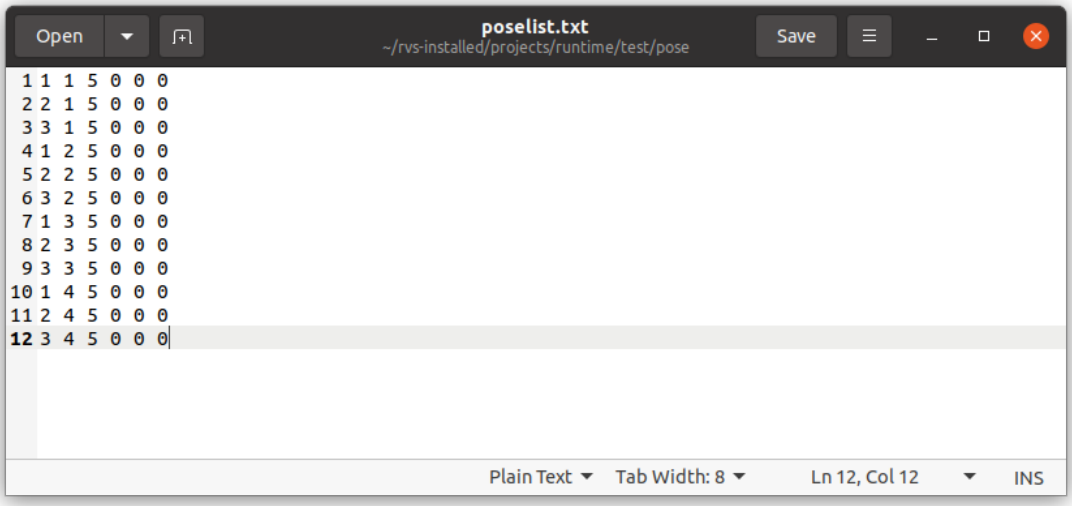
> 假设：1号坐标：（1,1,5）
>
> 2号坐标：（2,1,5）
>
> ...
>
> 5号坐标：（1,2,5）
>
> ...
>
> 12号坐标：（3,4,5）

步骤1：算子准备

添加 Trigger 、Load 、SortList 算子至算子图。

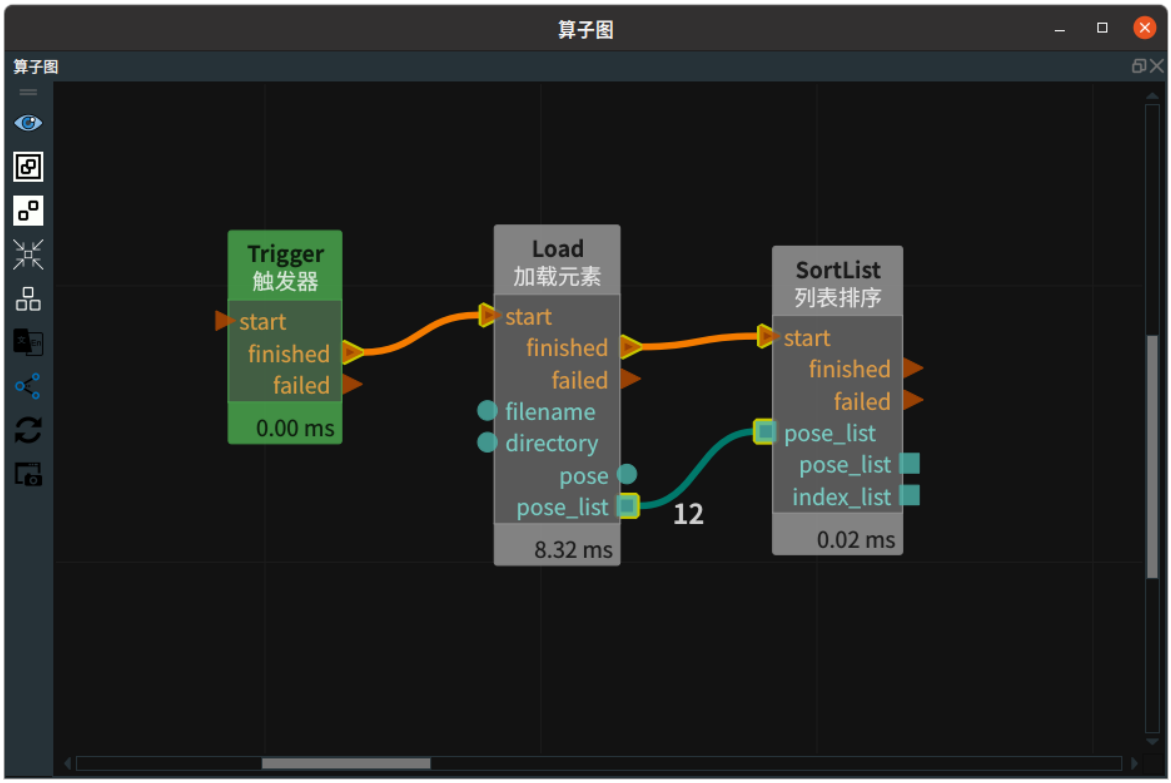
步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → pose
   - 文件 → ••• → 选择 pose 文件名(*example_data/pose/poselist.txt*)（文件内容如下)



2. 设置 SortList 算子参数：
   - 类型 → pose
   - 坐标 → 可视

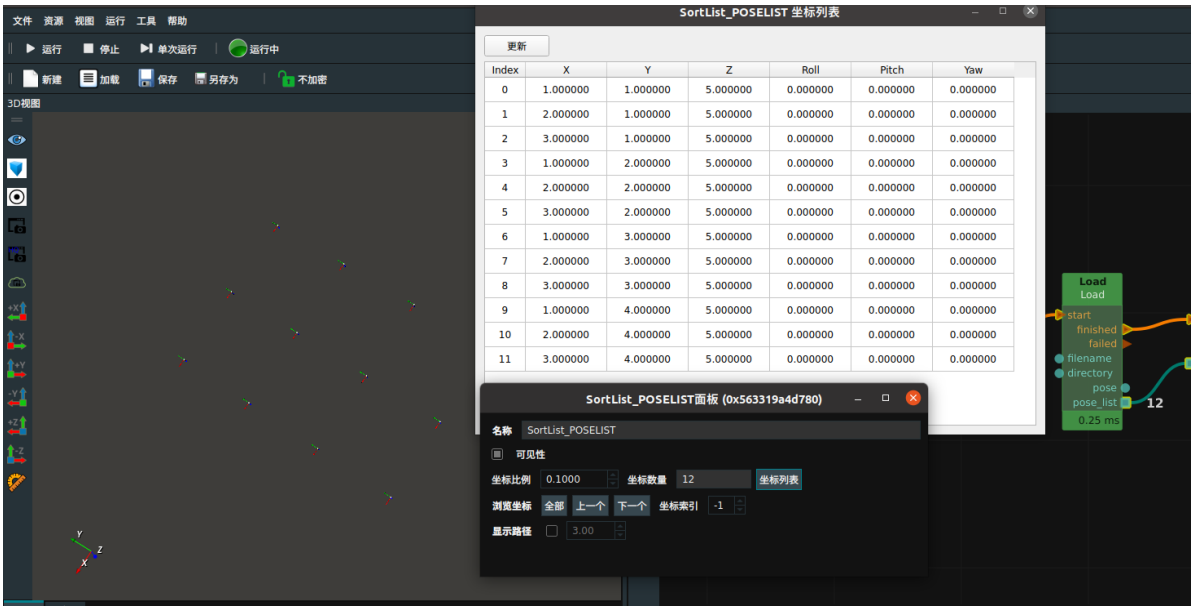
步骤3：连接算子



步骤4：运行与结果

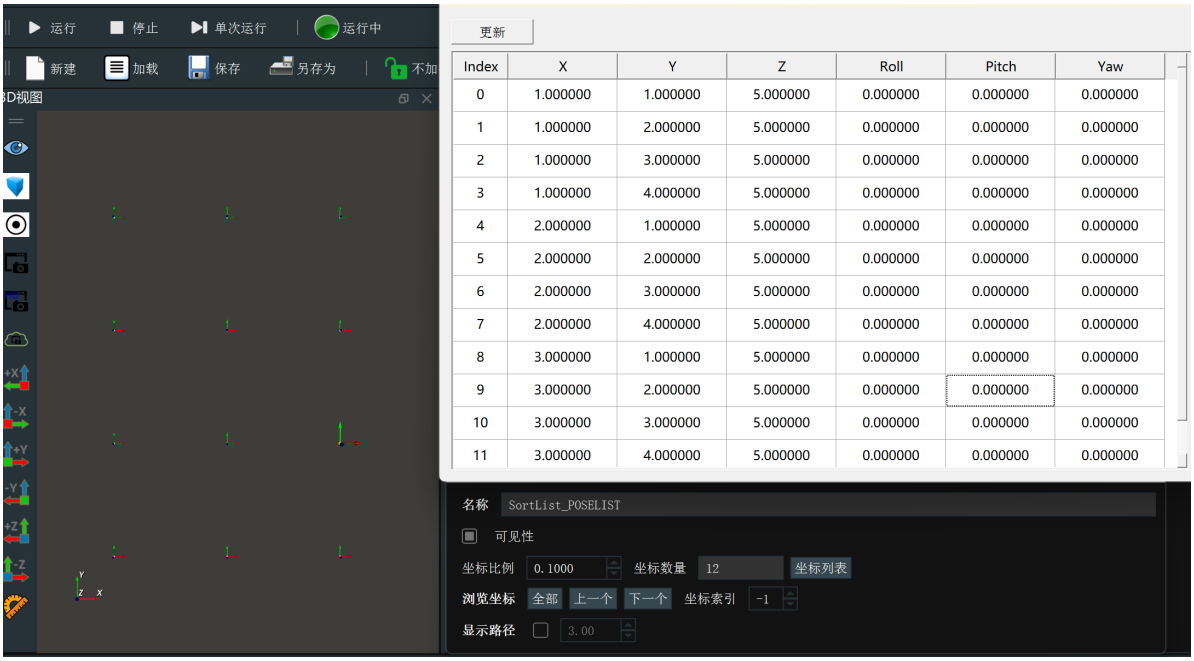
排序1： 按照1 -> 2 -> 3 -> 4 ->.....12 排序：

- 参考取值：
  - X权重： 1
  - Y权重 ： 4
  - Z权重 ： 0
- 双击 3D 视图中的 pose ，打开 POSELIST 面板。点击“坐标列表”按钮，显示排序后所有坐标参数。



排序2： 按照 1 -> 4 -> 7 -> 10 -> 2.....12 排序

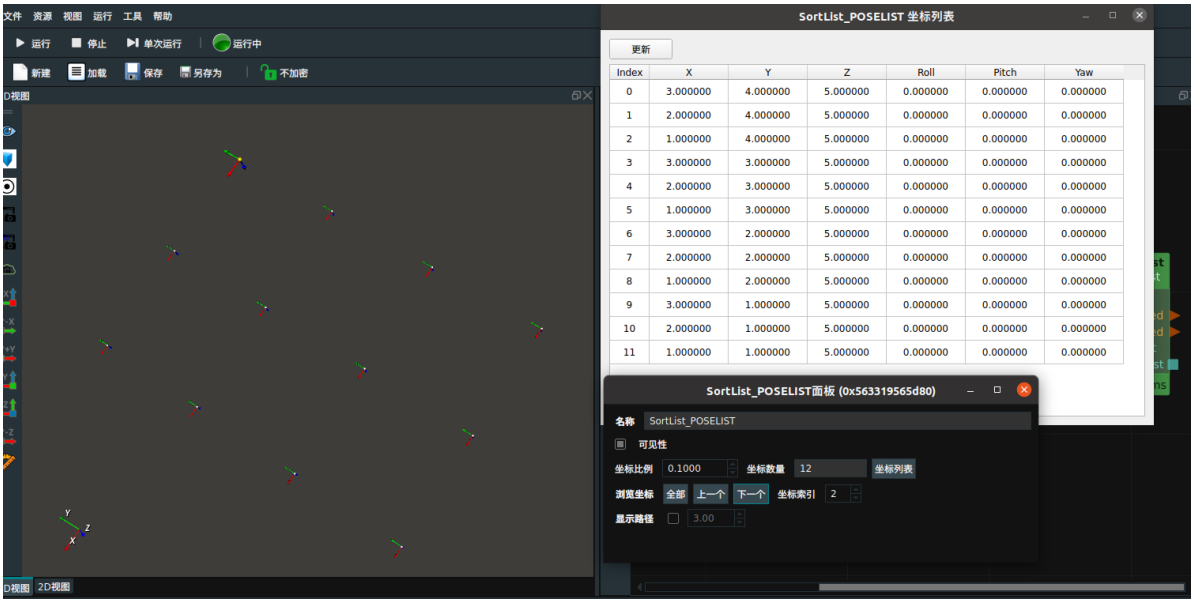
- 参考取值：
  - X权重：4
  - Y权重：1
  - Z权重 ： 0
- 双击 3D 视图中的 pose ，打开 POSELIST 面板。点击“坐标列表”按钮，显示排序后所有坐标参数。



Index	X	Y	Z	Roll	Pitch	Yaw
0	1.000000	1.000000	5.000000	0.000000	0.000000	0.000000
1	1.000000	2.000000	5.000000	0.000000	0.000000	0.000000
2	1.000000	3.000000	5.000000	0.000000	0.000000	0.000000
3	1.000000	4.000000	5.000000	0.000000	0.000000	0.000000
4	2.000000	1.000000	5.000000	0.000000	0.000000	0.000000
5	2.000000	2.000000	5.000000	0.000000	0.000000	0.000000
6	2.000000	3.000000	5.000000	0.000000	0.000000	0.000000
7	2.000000	4.000000	5.000000	0.000000	0.000000	0.000000
8	3.000000	1.000000	5.000000	0.000000	0.000000	0.000000
9	3.000000	2.000000	5.000000	0.000000	0.000000	0.000000
10	3.000000	3.000000	5.000000	0.000000	0.000000	0.000000
11	3.000000	4.000000	5.000000	0.000000	0.000000	0.000000

排序3： 按照 12 - > 11 -> 10- > 9.....1 排序

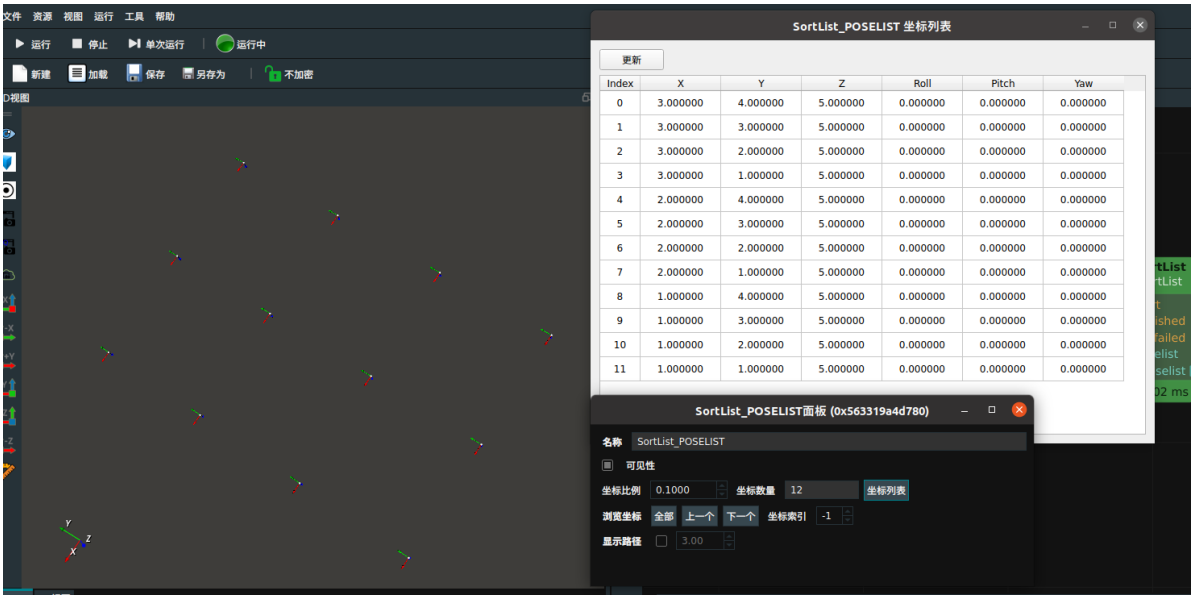
- 参考取值：
  - X权重：-1
  - Y权重： -4
  - Z权重： 0
- 双击 3D 视图中的 pose ，打开 POSELIST 面板。点击“坐标列表”按钮，显示排序后所有坐标参数。



Index	X	Y	Z	Roll	Pitch	Yaw
0	3.000000	4.000000	5.000000	0.000000	0.000000	0.000000
1	2.000000	4.000000	5.000000	0.000000	0.000000	0.000000
2	1.000000	4.000000	5.000000	0.000000	0.000000	0.000000
3	3.000000	3.000000	5.000000	0.000000	0.000000	0.000000
4	2.000000	3.000000	5.000000	0.000000	0.000000	0.000000
5	1.000000	3.000000	5.000000	0.000000	0.000000	0.000000
6	3.000000	2.000000	5.000000	0.000000	0.000000	0.000000
7	2.000000	2.000000	5.000000	0.000000	0.000000	0.000000
8	1.000000	2.000000	5.000000	0.000000	0.000000	0.000000
9	3.000000	1.000000	5.000000	0.000000	0.000000	0.000000
10	2.000000	1.000000	5.000000	0.000000	0.000000	0.000000
11	1.000000	1.000000	5.000000	0.000000	0.000000	0.000000

排序4：按照 12 ->9 -> 6 -> 3 -> 11.....1排序

- 参考取值：
  - X权重：-4
  - Y权重： -1
  - Z权重 ： 0
- 双击 3D 视图中的 pose ，打开 POSELIST 面板。点击“坐标列表”按钮，显示排序后所有坐标参数。



Cube

将 SortList 算子的 **类型** 属性选择 Cube ，用于立方体列表排序。根据立方体中心点坐标进行排序。

算子参数

- **X权重/weight_x**：x 轴坐标权重。
- **Y权重/weight_y**：y 轴坐标权重。
- **Z权重/weight_z**：z 轴坐标权重。
- **立方体列表/cube_list**：设置排序后立方体列表在 3D 视图中的可视化属性。
  - 打开立方体列表可视化。
  - 关闭立方体列表可视化。
  - 设置立方体列表的颜色。取值范围：[-2,360] 。默认值：-1 。
  - 设置立方体列表的透明度。取值范围：[0,1] 。默认值：0.5 。
- **索引列表/index_list**：设置排序后索引的曝光属性。可与交互面板中输出工具“表格”进行绑定。
  - 打开曝光。
  - 关闭曝光。

数据信号输入输出

- **cubelist**：
  - 数据类型：CubeList
  - 输入内容：立方体列表数据

输出：

- **cubelist**：
  - 数据类型：CubeList
  - 输出内容：排序后立方体列表数据
- **index_list**：
  - 数据类型：String
  - 输出内容：排序后索引列表

功能演示

使用 SortList 算子中 Cube 对加载的多个立方体进行不同方式的排序。

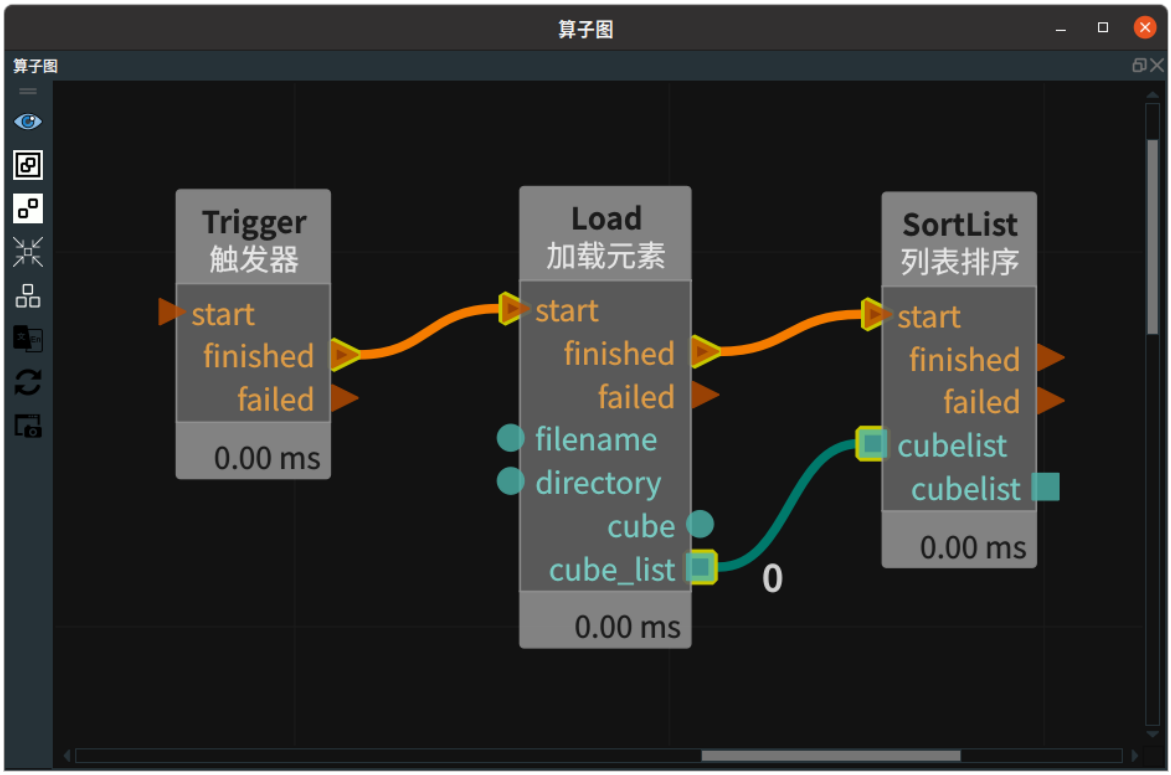
步骤1：算子准备

添加 Trigger 、Load 、SortList 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 类型 → Cube
    - 目录 → ••• → 选择 cube 文件目录名(*example_data/cubelist*)
2. 设置 SortList 算子参数：
    - 类型 → Cube
    - 立方体列表 → 可视

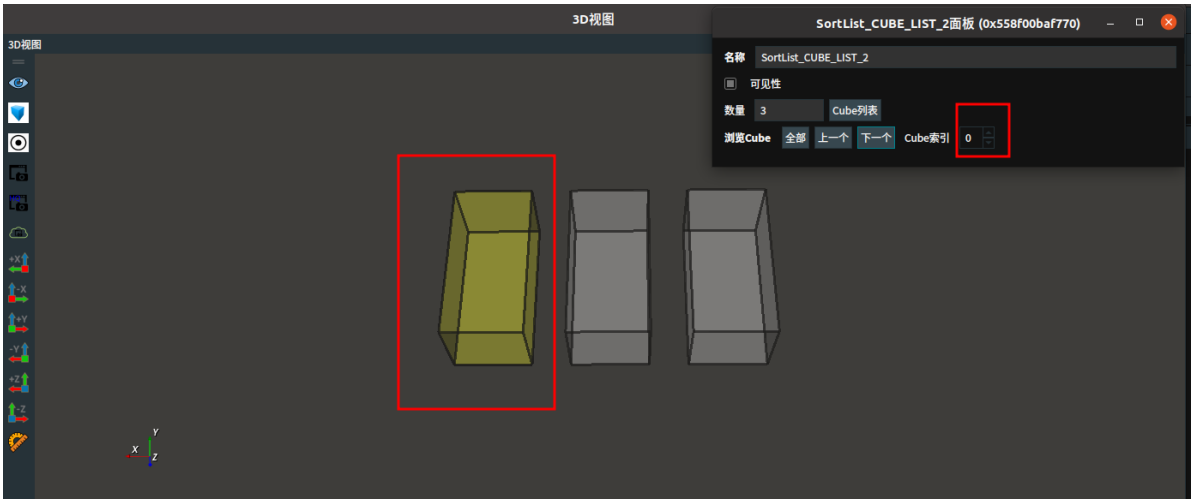
步骤3：连接算子



步骤4：运行

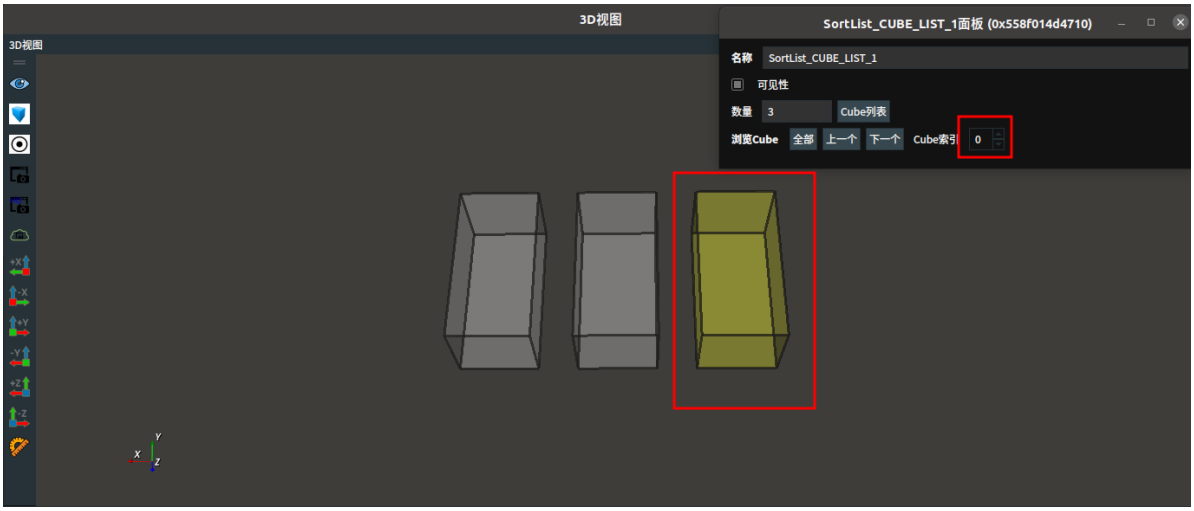
1. 将立方体按照 X 轴负方向排序

- 参考取值：
    - X权重：-1
    - Y权重：0
    - Z权重：0
- 双击 3D 视图中立方体，打开 CUBE_LIST 面板，点击按钮“ Cube 列表”，显示 Cube 索引 。当点击按钮“下一个”，索引会同步进行变换。



2. 将立方体 X 轴正方向排序

- 参考取值：
  - X权重：1
  - Y权重：0
  - Z权重：0
- 双击3D视图中立方体，打开CUBE_LIST面板，点击按钮“ Cube 列表”，显示 Cube 索引 。当点击按钮“下一个”，索引会同步进行变换。



PointCloud

将 SortList 算子的 **类型** 属性选择 PointCloud，用于点云列表排序。根据点云中心点坐标进行排序。

算子参数

- **X权重/weight_x**：x 轴坐标权重。
- **Y权重/weight_y**：y 轴坐标权重。
- **Z权重/weight_z**：z 轴坐标权重。
- **点云列表/cloud_list**：设置排序后点云在 3D 视图中的可视化属性。
  - 打开过滤后点云可视化。
  - 关闭过滤后点云可视化。
  - 设置 3D 视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。

-  设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。

- **索引列表/index_list**：设置排序后索引的曝光属性。可与交互面板中输出工具“表格”进行绑定。
  -  打开曝光。
  -  关闭曝光。

数据信号输入输出

输入：

- **cloud_list**：
  - 数据类型：PointCloudList
  - 输入内容：点云列表数据

输出：

- **cloud_list**：
  - 数据类型：PointCloudList
  - 输出内容：排序后点云列表数据
- **index_list**：
  - 数据类型：String
  - 输出内容：排序后索引列表

功能演示

使用 SortList 算子中 PointCloud 对加载的多个点云按照 X 轴正方向排序。

步骤1：算子准备

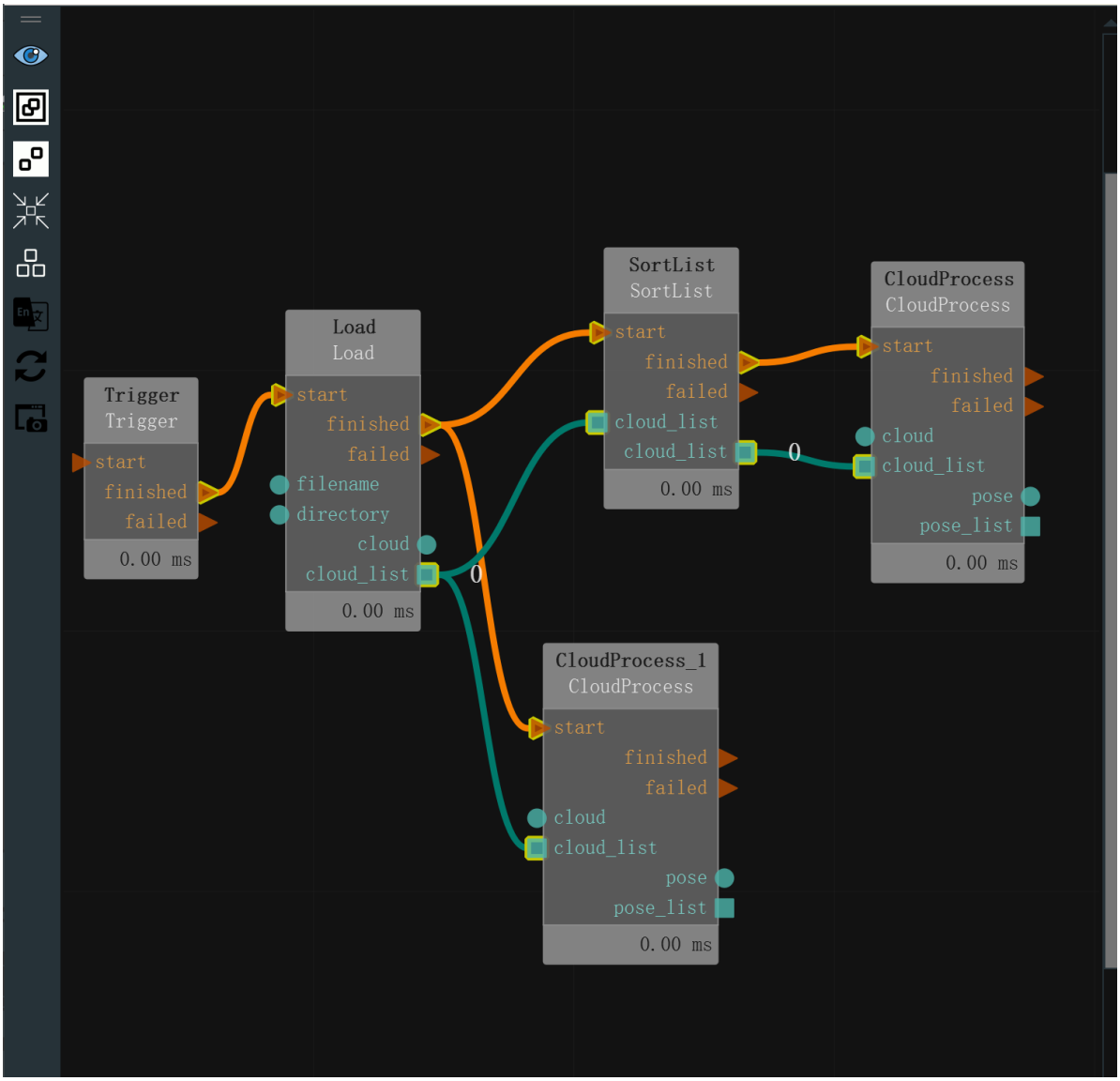
添加 Trigger 、Load 、SortList、CloudProcess (2 个）算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → pointcloud
   - 目录 → ••• → 选择点云文件目录名(*example_data/pointcloud/pointclouds*)
2. 设置 SortList 算子参数：
   - 类型 → PointCloud
   - X权重：1
   - Y权重：0
   - Z权重：0
   - 点云列表 →  可视
3. 设置 CloudProcess/CloudProcess_1 算子参数：
   - 类型 → CloudCentroid
   - 坐标列表 →  曝光

   说明：CloudProcess 算子用于输出点云的中心点 pose，便于查看排序后点云中心点坐标位置。

步骤3：连接算子

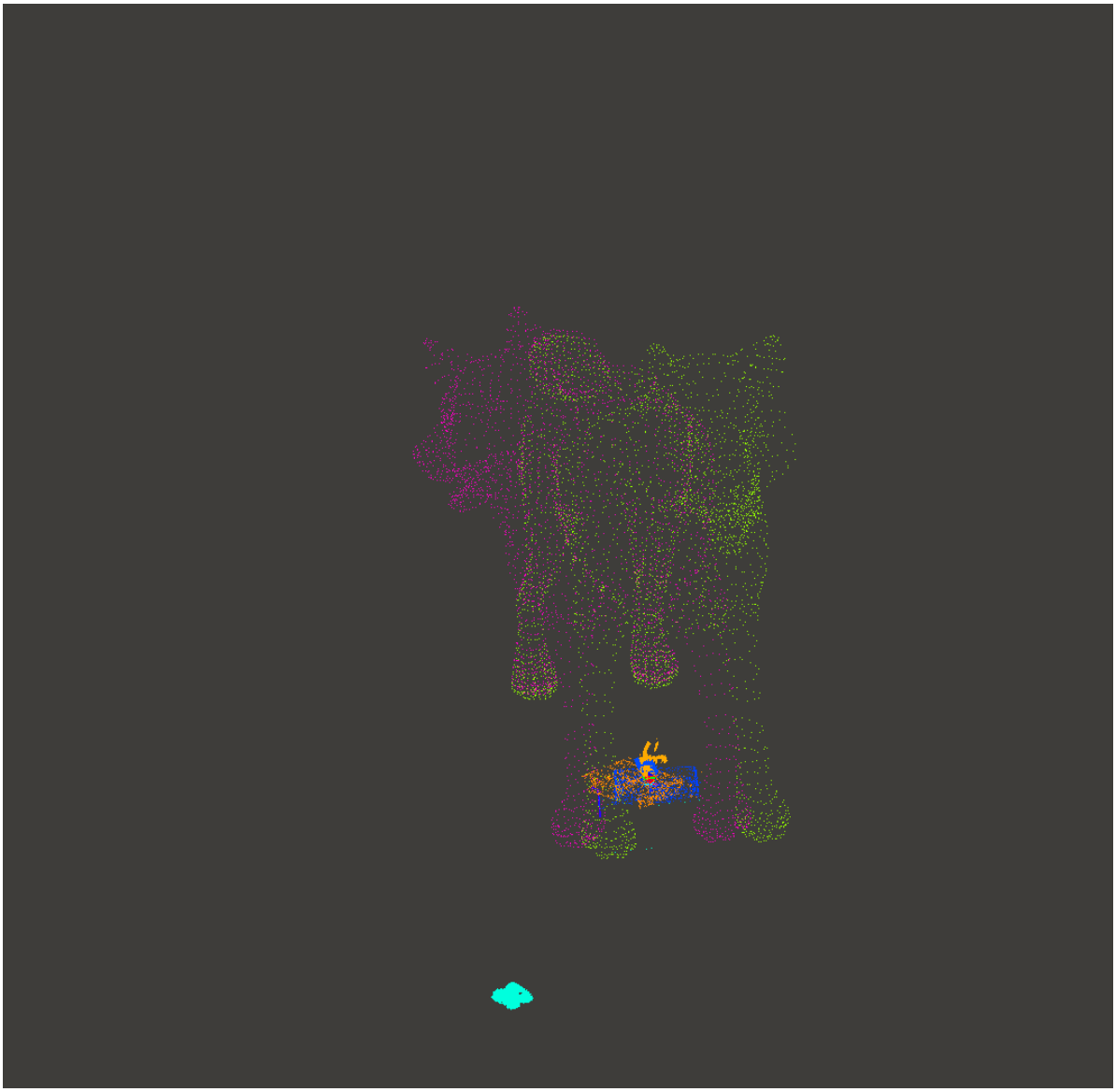


步骤4：运行

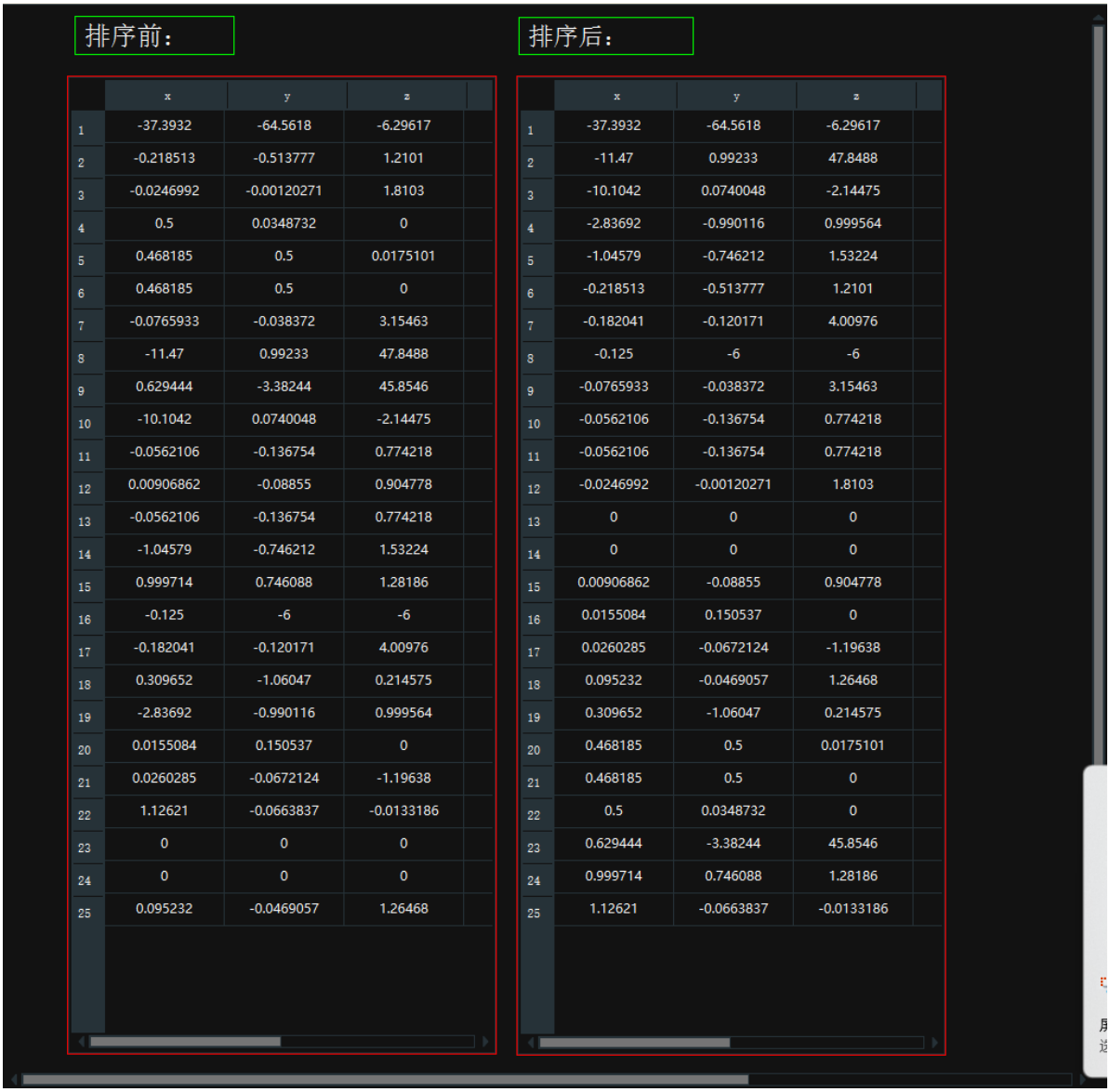
1. 在交互面板中拖入 2 个输入工具——“标签”，分别输入文字：排序前/排序后。
2. 再拖入 2 个 输出工具 ——”表格“，分别与CloudProcess/CloudProcess_1 的 pose_list 属性进行绑定。
3. 点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

1. 结果如下图所示，在 3D 视图中显示 SortList 算子 cloud_list 可视化结果。



3. 在交互面板中查看排序前后点云中心点 pose 对比。



	x	y	z
1	-37.3932	-64.5618	-6.29617
2	-0.218513	-0.513777	1.2101
3	-0.0246992	-0.00120271	1.8103
4	0.5	0.0348732	0
5	0.468185	0.5	0.0175101
6	0.468185	0.5	0
7	-0.0765933	-0.038372	3.15463
8	-11.47	0.99233	47.8488
9	0.629444	-3.38244	45.8546
10	-10.1042	0.0740048	-2.14475
11	-0.0562106	-0.136754	0.774218
12	0.00906862	-0.08855	0.904778
13	-0.0562106	-0.136754	0.774218
14	-1.04579	-0.746212	1.53224
15	0.999714	0.746088	1.28186
16	-0.125	-6	-6
17	-0.182041	-0.120171	4.00976
18	0.309652	-1.06047	0.214575
19	-2.83692	-0.990116	0.999564
20	0.0155084	0.150537	0
21	0.0260285	-0.0672124	-1.19638
22	1.12621	-0.0663837	-0.0133186
23	0	0	0
24	0	0	0
25	0.095232	-0.0469057	1.26468

	x	y	z
1	-37.3932	-64.5618	-6.29617
2	-11.47	0.99233	47.8488
3	-10.1042	0.0740048	-2.14475
4	-2.83692	-0.990116	0.999564
5	-1.04579	-0.746212	1.53224
6	-0.218513	-0.513777	1.2101
7	-0.182041	-0.120171	4.00976
8	-0.125	-6	-6
9	-0.0765933	-0.038372	3.15463
10	-0.0562106	-0.136754	0.774218
11	-0.0562106	-0.136754	0.774218
12	-0.0246992	-0.00120271	1.8103
13	0	0	0
14	0	0	0
15	0.00906862	-0.08855	0.904778
16	0.0155084	0.150537	0
17	0.0260285	-0.0672124	-1.19638
18	0.095232	-0.0469057	1.26468
19	0.309652	-1.06047	0.214575
20	0.468185	0.5	0.0175101
21	0.468185	0.5	0
22	0.5	0.0348732	0
23	0.629444	-3.38244	45.8546
24	0.999714	0.746088	1.28186
25	1.12621	-0.0663837	-0.0133186

CloudSegment 点云切割

CropboxSegment算子用于对点云进行切割，包含 CropboxSegment 、PlaneSegment 、PassThroughSegment 、DiffSegment 、NNPDSegment 。

类型	功能
CropboxSegment	裁剪框切割，根据输入的立方体区域对点云进行切割，需要结合 Emit-Cube 算子使用。
PlaneSegment	平面点云切割。可以选取切割平面或者平面上/下的点云。
PassThroughSegment	切割坐标某个方向上一定距离的点云。
DiffSegment	输入两个点云，切割两个点云中不同的部分。
NNPDSegment	输入两个点云中，保留源点云在目标点云中缺失的部分。

CropboxSegment

将 CloudSegment 点云切割 **类型** 设置为 CropboxSegment ，用于裁剪框切割，根据输入的立方体区域对点云进行切割，需要结合 Emit - Cube 算子使用。

算子参数

- **模式/mode**：设置点云切割的方法，共有三种，分别为：
  - **intersection**：当输入单个 cube 时，切割 cube 框选的部分；当输入的是 cubes ，切割各个 cube 中相交的部分。
  - **individual**：当多个 cube 切割同一块点云时，每个 cube 都对点云进行切割，输出的是点云列表。
  - **outside_points**：切割没有被 cube 框选的点云。
- **点云/cloud**：设置切割后点云在 3D 视图中的可视化属性。
  -  打开切割后点云可视化。
  -  关闭切割后点云可视化。
  -  设置 3D 视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  -  设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。
- **点云列表/cloud_list**：设置过滤后点云列表在 3D 视图中的的可视化属性。值描述与 **点云** 一致。

数据信号输入输出

输入：

说明：根据需求选择其中两种数据信号端口输入即可，如 cloud 和 cube 或者 cloud_list 和 cube_list 。

- **cloud**：
  - 数据类型：PointCloud
  - 输入内容：点云数据
- **cloud_list**：
  - 数据类型：PointCloudList
  - 输入内容：点云列表数据
- **cube**：

    - 数据类型：Cube
    - 输入内容：立方体数据
- `cubelist`：
    - 数据类型：CubeList
    - 输入内容：立方体列表数据

输出：

- `cloud`：
    - 数据类型：PointCloud
    - 输出内容：切割后点云数据
- `cloud_list`：
    - 数据类型：PointCloudList
    - 输出内容：切割后点云列表数据

功能演示

使用 CloudSegment 算子中 CropboxSegment 切割加载点云中的 Cube 包裹住的部分。

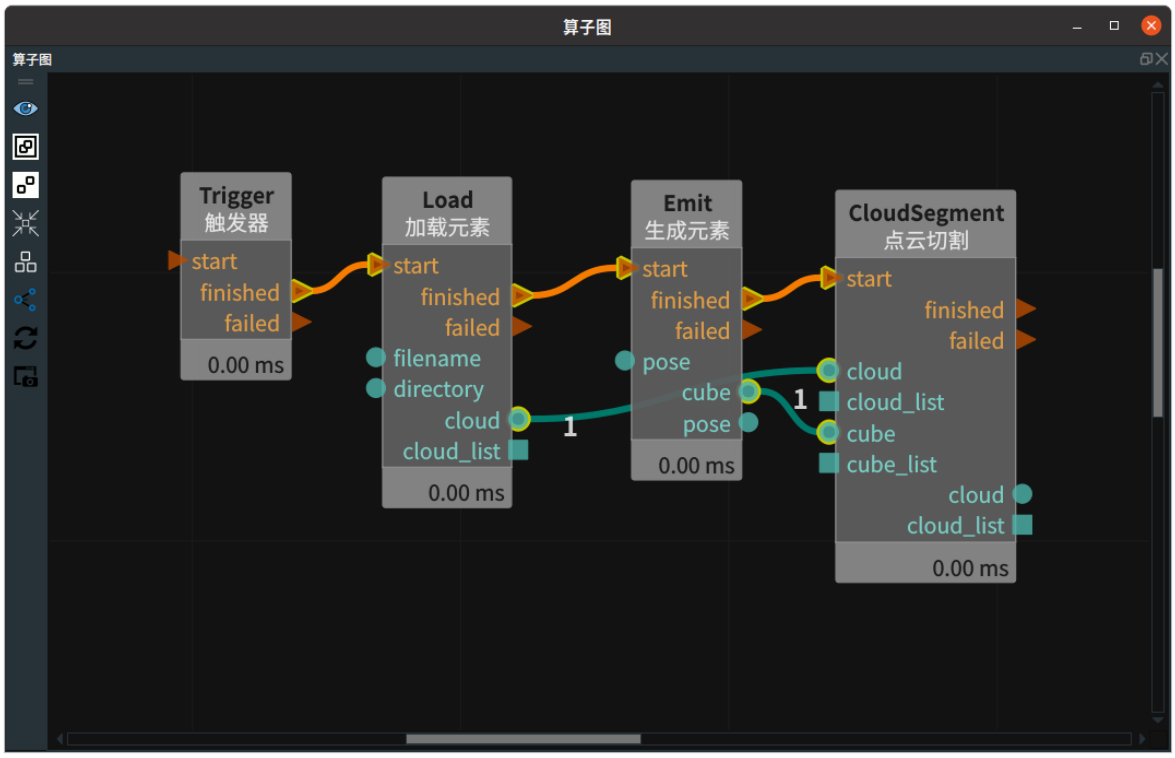
步骤1：算子准备

添加 Trigger 、Load 、Emit 、CloudSegment 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 类型 → PointCloud
    - 文件 → ••• → 选择点云文件名（*example_data/pointcloud/chessboard.pcd* ）
    - 点云 → 可视 → -2
2. 设置 Emit 算子参数：
    - 类型 → Cube
    - 坐标 → 0 0 0.7 0 0 0
    - 宽度 → 1
    - 高度 → 1
    - 深度 → 0.5
    - 立方体 → 可视
3. 设置 CloudSegment 算子参数：
    - 类型 → CropboxSegment
    - 模式 → intersection
    - 点云 → 可视 → -2

步骤3：连接算子

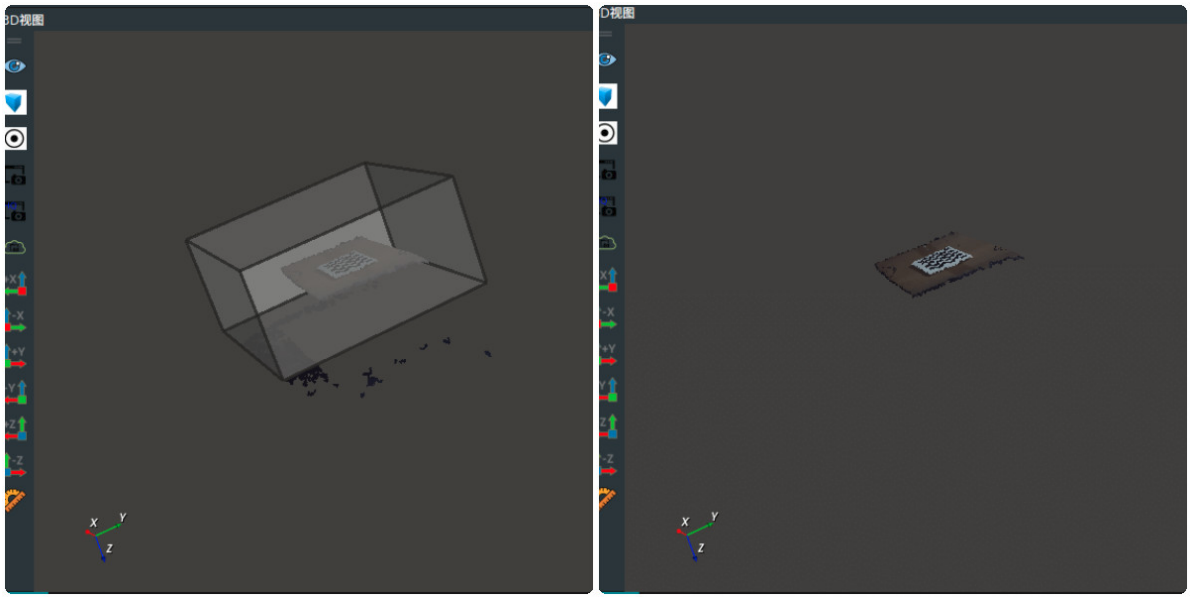


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，3D 视图中，左图为 Load 算子和 Emit 算子的可视化结果，右图为裁剪后的点云可视化结果。



PlaneSegment

将 CloudSegment 点云切割 **类型** 设置为 PlaneSegment ，用于平面点云切割。可以选取切割平面或者平面上/下的点云。

算子参数

- **切割模式/segment_mode**：设置点云切割的方法。共有三种，分别为：
  - ON_PLANE：切割平面点云。
  - ABOVE_PLANE：切割坐标 z 轴正方向的点云。
  - BELOW_PLANE：切割坐标 z 轴反方向的点云。
- **取反/negative**：取反，配合 segmentation_mode 使用。
  - False：裁剪的点云按照实际方法进行切割。
  - True：裁剪的点云为除了选中切割面的所有点云。
- **距离阈值/distance_threshold**：点云切割的距离阈值。FF默认值：0.001 。单位：m。
- **平面立方体宽度/plane_cube_width**：设置空间 cube 的宽度。取值范围：(0,+∞)。默认值：0.1 。单位：m。
- **平面立方体高度/plane_cube_height**：设置空间 cube 的高度。取值范围：(0,+∞)。默认值：0.15 。单位：m。
- **平面立方体深度/plane_cube_depth**：设置空间 cube 的深度。取值范围：(0,+∞)。默认值：0.00001 。单位：m。
- **点云/cloud**：设置切割后点云在 3D 视图中的可视化属性。
  -  打开切割后点云可视化。
  -  关闭切割后点云可视化。
  -  设置 3D 视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  -  设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。
- **平面立方体云/plane_cube**：设置平面立方体在 3D 视图中的可视化属性。
  -  打开平面立方体可视化。
  -  关闭平面立方体可视化。
  -  设置平面立方体的颜色。取值范围：[-2,360] 。默认值：-2 。
  -  设置平面立方体的透明度。取值范围：[0,1] 。默认值：0.5 。
- **点云列表/cloud_list**：设置切割后点云列表在 3D 视图中的可视化属性。参数值描述与 **点云** 一致。
- **平面立方体列表/plane_cube_list**：设置平面立方体列表在 3D 视图中的可视化属性。参数值描述与 **点云** 一致。

数据信号输入输出

输入：

- **cloud**：
  - 数据类型：PointCloud
  - 输入内容：点云数据
- **plane_pose**：
  - 数据类型：Pose
  - 输入内容：平面中心点 pose

输出：

- **cloud**：
  - 数据类型：PointCloud
  - 输出内容：切割后点云数据

- `plane_cube`：
  - 数据类型：Cube
  - 输出内容：平面立方体

功能演示

使用 CloudSegment 算子中 PlaneSegment 切割出平面点云。

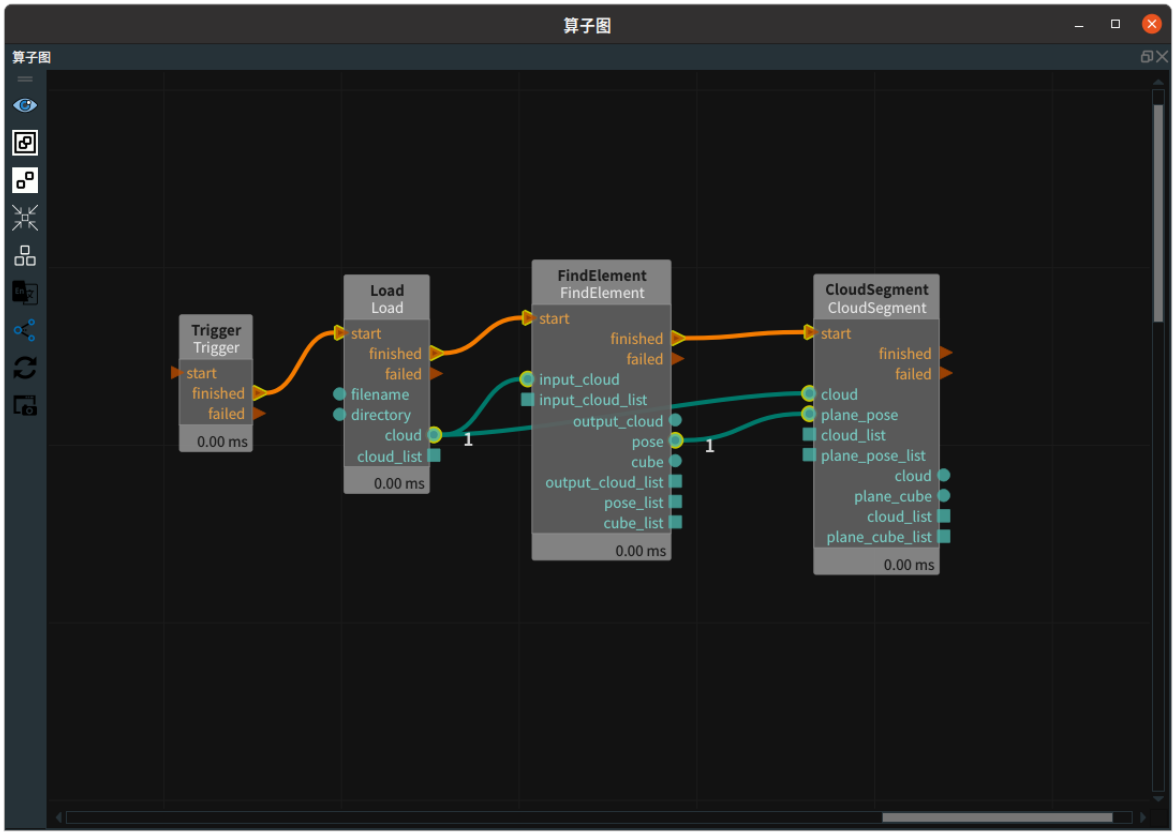
步骤1：算子准备

添加 Trigger 、Load 、FindElement 、CloudSegment 算子至算子图。

步骤2：设置算子参数

1. 设置 FindElement 算子参数：类型 → plane
2. 设置 Load 算子参数：
   - 类型 → PointCloud
   - 文件 → ••• → 选择点云文件名（*example_data/pointcloud/fruit.pcd* ）
   - 点云 → 可视→ -2
3. 设置 CloudSegment 算子参数：
   - 类型 → PlaneSegment
   - 模式 → BELOW_PLANE
   - 点云 → 可视→ -2

步骤3：连接算子

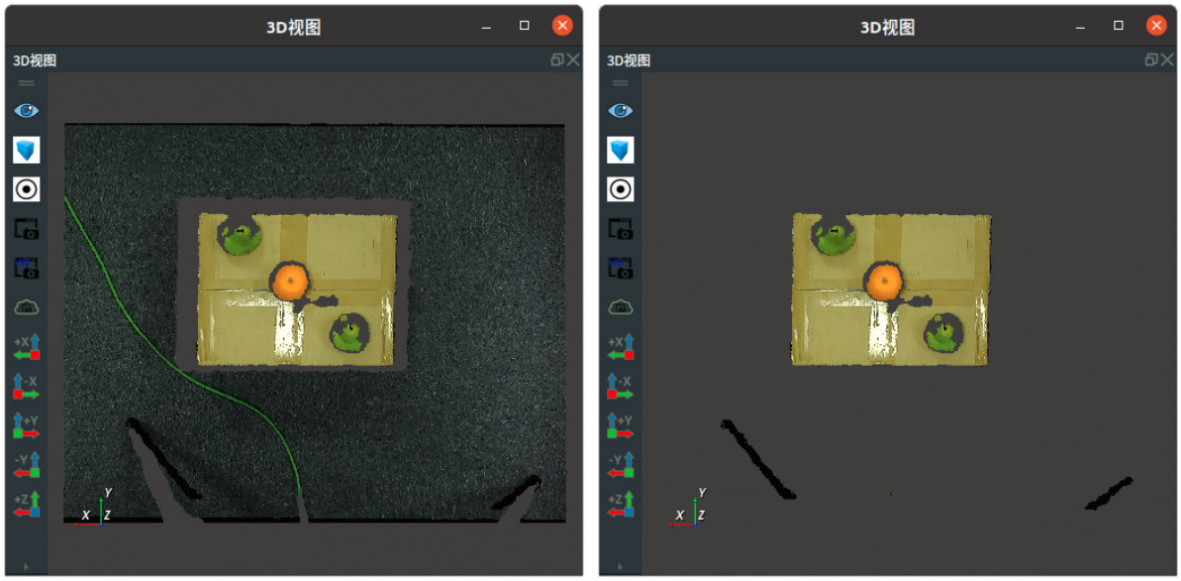


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，3D 视图中，左图为 Load 算子加载的点云可视化结果，右图为裁剪后的点云可视化结果。



PassThroughSegment

将 CloudSegment 点云切割 **类型** 设置为 CropboxSegment ，用于切割坐标某个方向上一定范围的点云。

算子参数

- **领域/field**：切割方向区域。默认值：z ，表示坐标 z 轴方向。
- **最小值/min**：切割的最小距离。取值范围：(-∞,+∞)。默认值：0 ，表示从 0 开始。单位：m。
- **最大值/max**：切割的最大距离。取值范围：(-∞,+∞)。默认值：1 ，表示到 1 结束。单位：m。
- **取内部/inside**：
  - True：选取 min-max 范围内的点云。
  - False：选取 min-max 范围外的点云。
- **点云/cloud**：设置切割后点云在 3D 视图中的可视化属性。
  - 打开切割后点云可视化。
  - 关闭切割后点云可视化。
  - 设置 3D 视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  - 设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。

数据信号输入输出

输入：

- **cloud**：
  - 数据类型：PointCloud
  - 输入内容：点云数据

输出：

- **cloud**：
  - 数据类型：PointCloud
  - 输出内容：切割后点云数据

功能演示

使用 CloudSegment 算子的 PassThroughSegment 切割出加载点云中的平面部分。

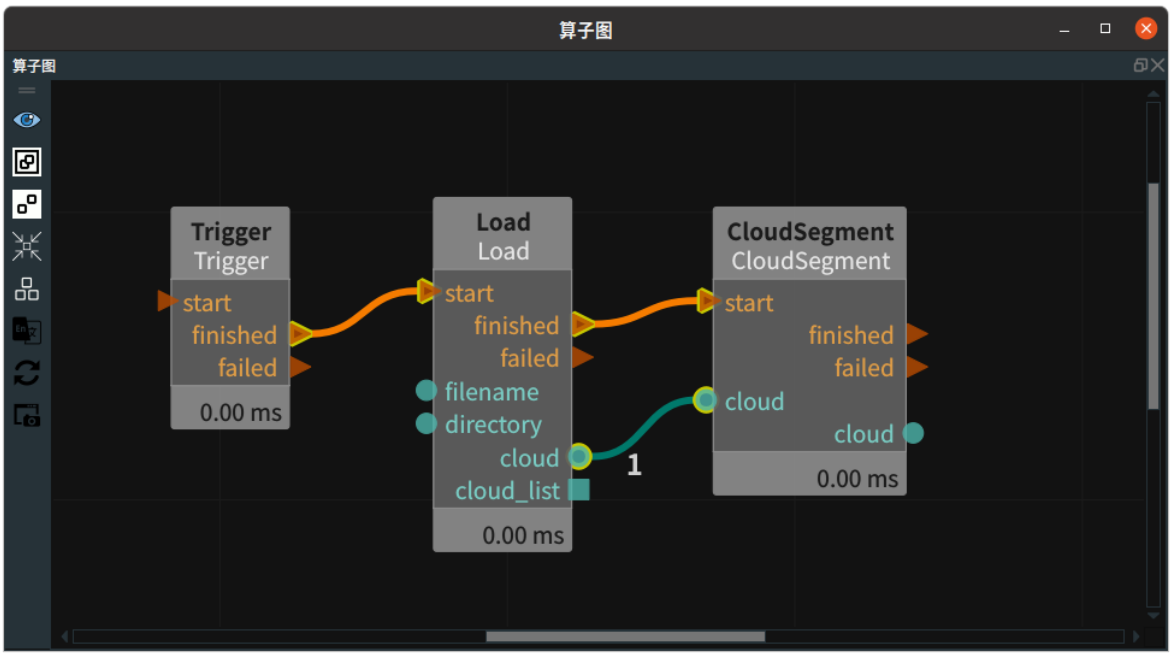
步骤1：算子准备

添加 Trigger 、Load 、CloudSegment 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 类型 → PointCloud
    - 文件 → ••• → 选择点云文件名(*example_data/pointcloud/chessboard.pcd*)
    - 点云 →  可视 →  -2
2. 设置CloudSegment算子参数：
    - 类型 → PassThroughSegment
    - 点云 →  可视 →  -2

步骤3：连接算子

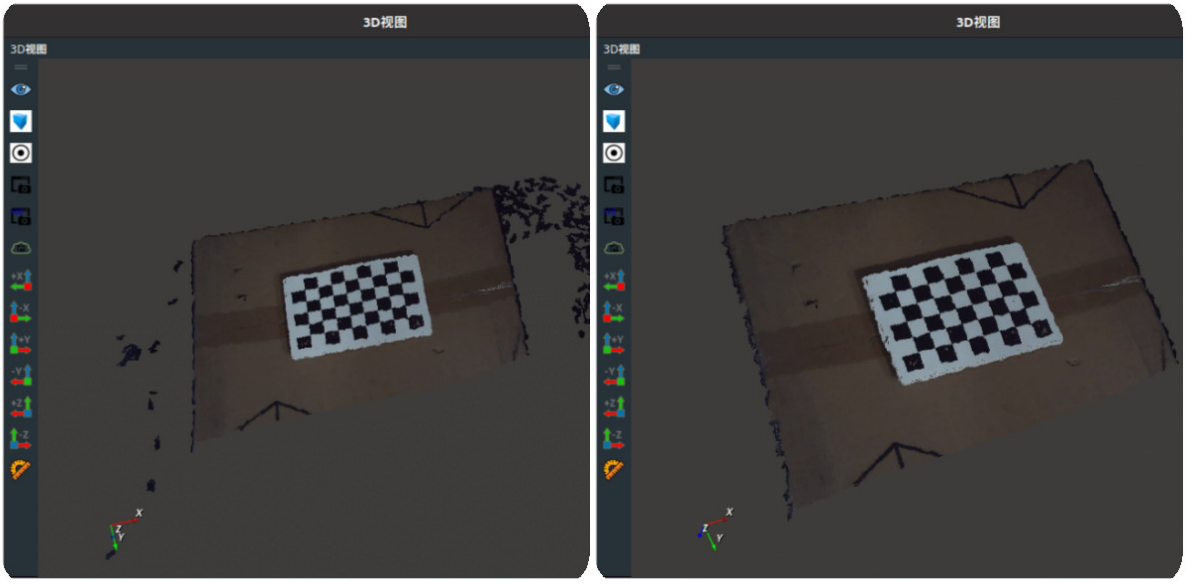


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，3D 视图中，左图为 Load 算子加载点云可视化结果，右图为裁剪后的点云可视化结果。



DiffSegment

将 CloudSegment 点云切割 **类型** 设置为 DiffSegment ，用于输入两个点云，切割两个点云中不同的部分。

算子参数

- **距离阈值/distance_threshold**：source_cloud（源点云）与target_cloud（目标点云）的距离阈值。取值范围：(0,+∞)。默认值：0.001 。单位：m。
- **差异点云/diff_cloud**：设置切割后点云在 3D 视图中的可视化属性。
  - 打开切割后点云可视化。
  - 关闭切割后点云可视化。
  - 设置 3D 视图中点云的颜色。取值范围：[-2,360]。默认值：-1 。
  - 设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。

数据信号输入输出

- **source_cloud**：
  - 数据类型：PointCloud
  - 输入内容：源点云数据
- **target_cloud**：
  - 数据类型：PointCloud
  - 输入内容：目标点云数据

输出：

- **diff_cloud**：
  - 数据类型：PointCloud
  - 输出内容：切割后点云数据

功能演示

使用 CloudSegment 算子中 DiffSegment 切割出加载点云中水果的部分。

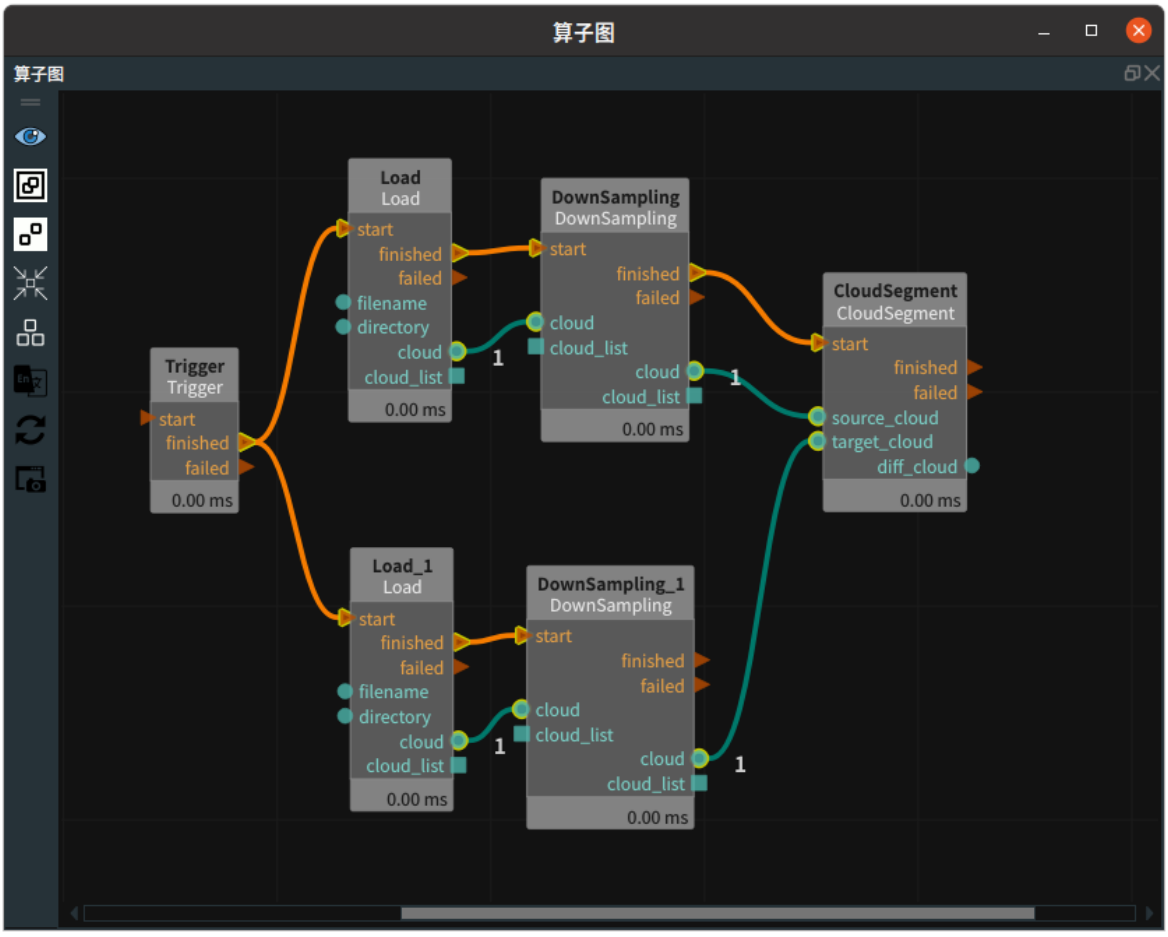
步骤1：算子准备

添加 Trigger 、Load （2个）、DownSampling 、DownSampling_1 、CloudSegment 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 类型 → PointCloud
    - 文件 → ••• → 选择点云文件名(*example_data/pointcloud/fruit2.pcd*)
    - 点云 →  可视→  -2
2. 置 Load_1 算子参数：
    - 类型 → PointCloud
    - 文件 → ••• → 选择点云文件名(*example_data/pointcloud/plane.pcd*)
    - 点云 →  可视→  -2
3. 设置 DownSampling 算子参数：
    - 类型→ DownSample
    - x 方向采样→ 0.001
    - y 方向采样→ 0.001
    - z 方向采样→ 0.001
4. 设置 DownSampling_1 算子参数：
    - 类型→ DownSample
    - x 方向采样 → 0.001
    - y 方向采样 → 0.001
    - z 方向采样→ 0.001
5. 设置 CloudSegment 算子参数：
    - 类型 → DiffSegment
    - 点云 →  可视→  -2

步骤3：连接算子

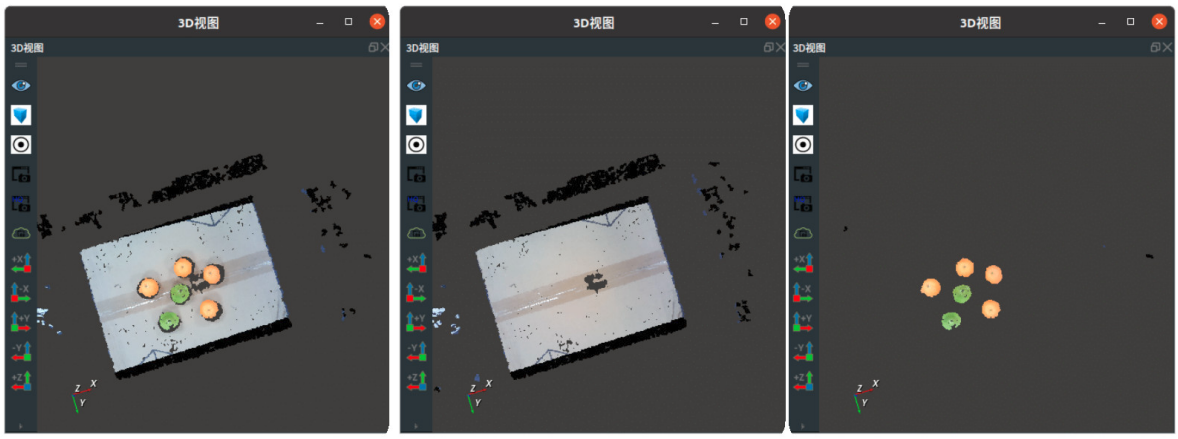


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，3D 视图中，左图为 Load 算子可视化结果，中图为 Load_1 算子可视化结果，右图为裁剪后的水果点云可视化结果。



NNPDSegment

将 CloudSegment 点云切割 **类型** 设置为 NNPDSegment ，用于输入两个点云中，保留源点云在目标点云中缺失的部分。

算子参数

- **距离阈值/distance_threshold**：source_cloud（源点云）与 target_cloud（目标点云）的距离阈值。取值范围：(0,+∞)。默认值：0.001 。单位：m。
- **剩下的点云/remaining_cloud**：设置切割后点云在 3D 视图中的可视化属性。
  - 打开切割后点云可视化。
  - 关闭切割后点云可视化。
  - 设置 3D 视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  - 设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。

数据信号输入输出

- **source_cloud**：
  - 数据类型：PointCloud
  - 输入内容：源点云数据
- **target_cloud**：
  - 数据类型：PointCloud
  - 输入内容：目标点云数据

输出：

- **remaining_cloud**：
  - 数据类型：PointCloud
  - 输出内容：切割后点云数据

功能演示

使用 CloudSegment 算子，先使用**类型**中 CropboxSegment 切割掉目标点云中的一部分，再使用 NNPDSegmet 保留缺失的部分。

步骤1：算子准备

添加 Trigger 、Load 、Emit 、DownSampling 、CloudSegment 、CloudSegment_1 算子至算子图。

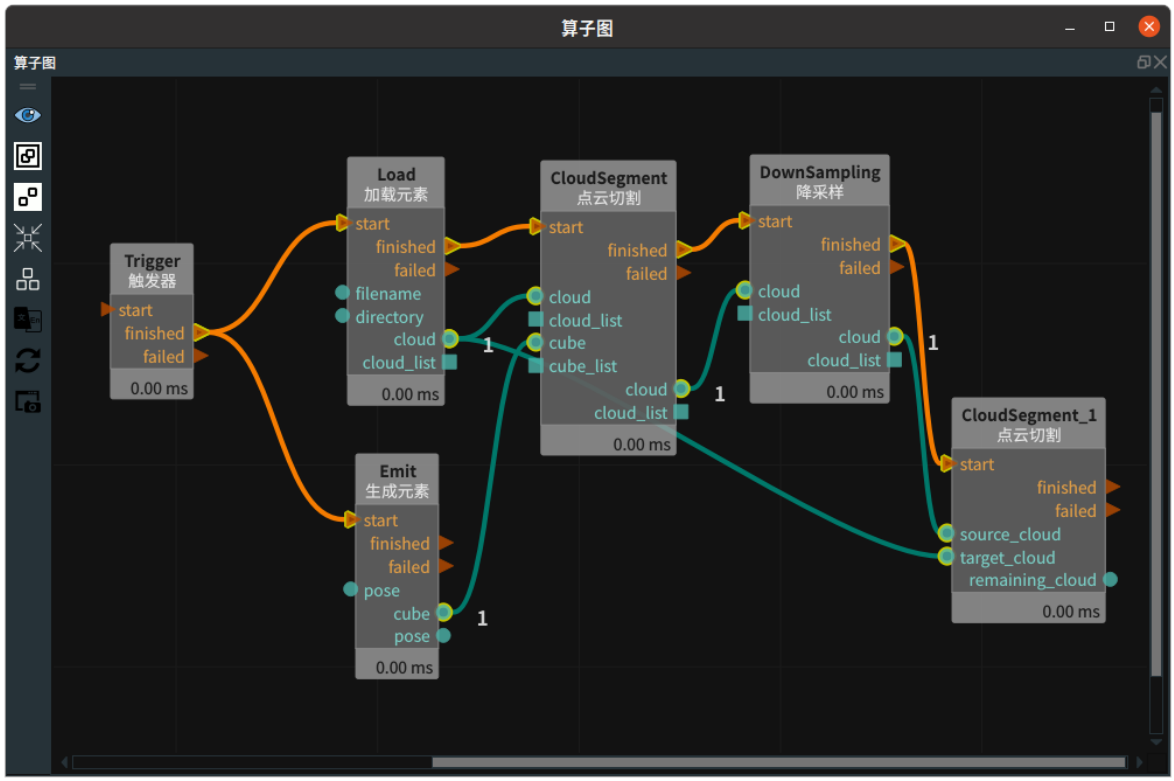
步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → PointCloud
   - 文件 → ••• → 选择点云文件名(*example_data/pointcloud/plane.pcd*)
   - 点云 → 可视→ -2
2. 设置 CloudSegment 算子参数：
   - 类型 → CropboxSegment
   - 模式 → intersection
   - 点云 → 可视 → -2
3. 设置 CloudSegment_1 算子参数：
   - 类型 → NNPDSegmet
   - 点云 → 可视 → -2
4. 设置 DownSampling 算子参数：
   - 类型→ DownSample
   - x 方向采样 → 0.001

- y 方向采样 → 0.001
- z 方向采样 → 0.001

5. 设置Emit算子参数：
   - 类型 → Cube
   - 坐标 →0.4 0 0.7 0 0 0
   - 宽度 →0.6
   - 高度→0.6
   - 深度 → 0.3

步骤3：连接算子



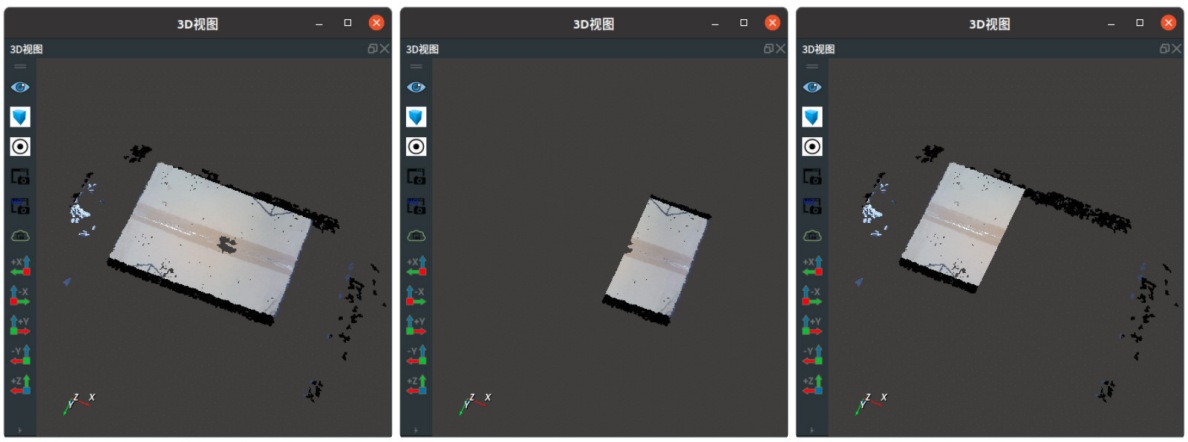
步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，3D 视图中：

- 左图为 Load 算子的点云可视化结果（ target_cloud ）。
- 中图为 CloudSegment 算子- CropBoxSegment 算子的点云可视化结果（ source_cloud ）。
- 右图为 CloudSegment 算子- NNPDSegment 中点云可视化结果（ remaining_cloud ）



ClusterExtraction 点云聚类提取

ClusterExtraction 算子根据设定的最小距离参数 tolerance，将彼此间距超过该距离的两个目标点归为两类，间距小于该距离的点云归为一类，最终将多个目标点云彼此分开。

算子参数

- **最小点数/min_points**：设置每一个点云类别的点云最小数量。
- **最大点数/max_points**：设置每一个点云类别的点云最大数量。
- **公差值/tolerance**：设置点云中两个目标点的最小距离。默认值：0.001。单位：m。

  注意：该属性值设置与点云的密度和单位有关。如果运行过程中出现算子卡死的情况，建议从小到大逐步调整该参数，并不断测试效果。
- **点云列表/cloud_list**：设置聚类后点云列表在 3D 视图中的可视化属性。
  -  打开过滤后点云可视化。
  -  关闭过滤后点云可视化。
  -  设置 3D 视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  -  设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。

算子输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **cloud**：
  - 数据类型：PointCloud
  - 输入内容：点云数据
- **cloud_list**：
  - 数据类型：PointCloudList
  - 输入内容：点云列表数据

输出：

- **cloud_list**：
  - 数据类型：PointCloudList
  - 输出内容：点云列表数据

功能演示

使用 ClusterExtraction 算子将点云中将多个目标点云彼此分开。

步骤1：算子准备

添加 Trigger 、Load 、ClusterExtraction 算子至算子图。

步骤2：设置算子参数

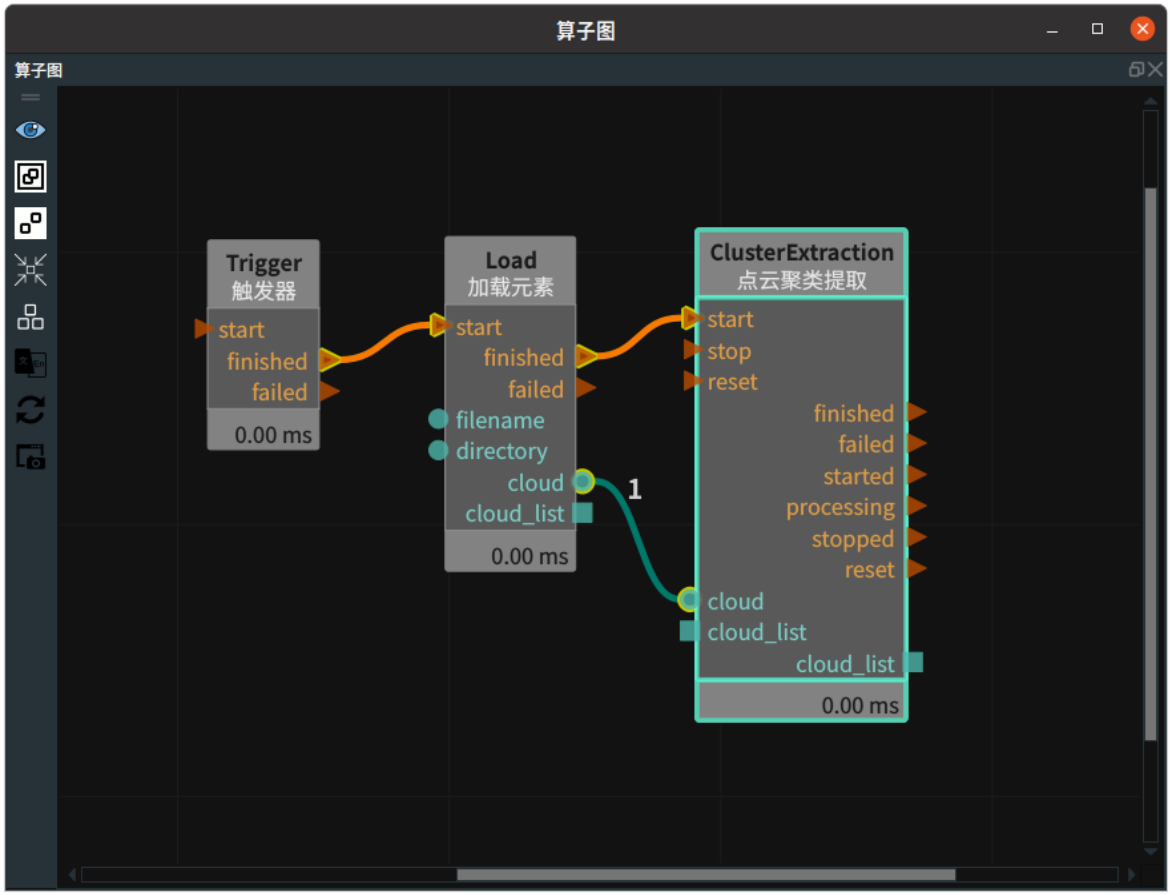
1. 设置 Load 算子参数：
   - 类型 → PointCloud
   - 文件 → ••• → 选择点云文件名 (*example_data/pointcloud/ClusterExtraction_cloud.pcd*)

- 点云 → 可视

2. 设置 ClusterExtraction 算子参数：

   - 点云列表 → 可视
   - 最小点数 → 5000
   - 最大点数 → 100000
   - 公差值 → 0.009

步骤3：连接算子

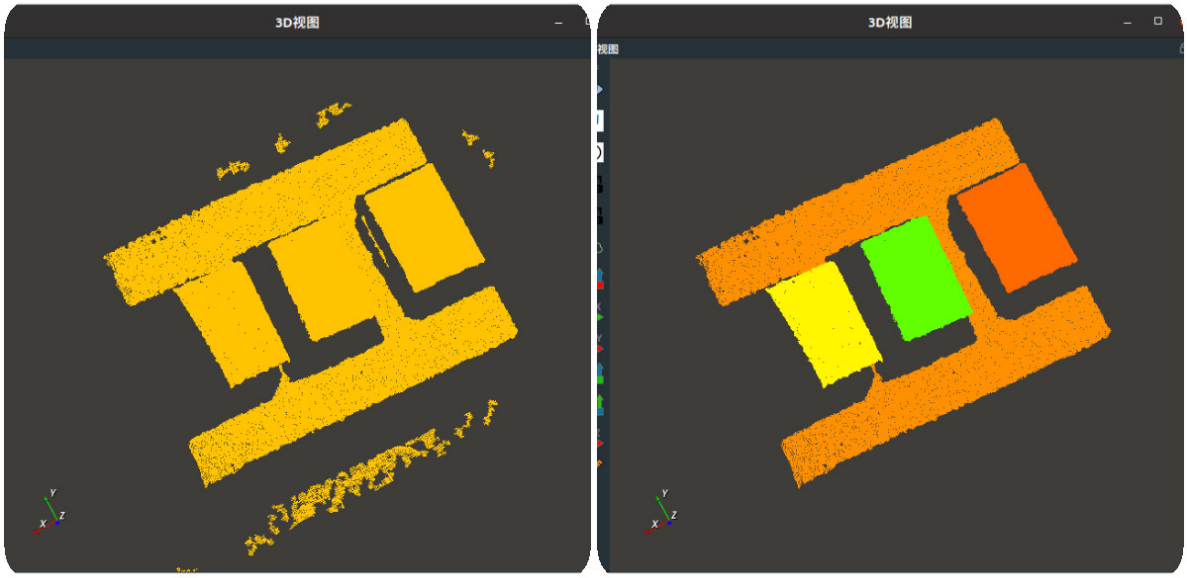


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，3D 视图中，左图为 Load 算子的点云可视化结果，右图为 ClusterExtraction 算子的点云可视化结果。



CloudFilter 点云过滤

CloudFilter 算子用于点云过滤。

类型	功能
CloudSize	通过选择不同模式输入参数值进行点云过滤。

CloudSize

算子参数

- **模式/mode**：点云过滤的模式。
  - LowerLimit：该模式依据 min_points 进行判断，当输入点云点数 ≤ min_points 时，触发算子 failed 信号。
  - UpperLimit：该模式依据 max_points 进行判断，当输入点云点数 ≥ max_points 时，触发算子 failed 信号。
  - BoundLimit：该模式依据 min_points 与 max_points 区间进行判断，当输入点云点数不在 min_points 与 max_points 区间内，触发算子 failed 信号。
- **最小点数/min_points**：限制点云中最小点数。默认值：0 。
- **最大点数/max_points**：限制点云中最大点数。默认值：10000 。
- **点云/cloud**：设置过滤后点云在 3D 视图中的可视化属性 。
  -  打开过滤后点云可视化。
  -  关闭过滤后点云可视化。
  -  设置 3D 视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  -  设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。
- **点云列表/cloud_list**：设置过滤后点云列表在 3D 视图中的的可视化属性。值描述与 **点云** 一致。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **cloud**：
  - 数据类型：PointCloud
  - 输入内容：点云数据
- **cloud_list**：
  - 数据类型：PointCloudList
  - 输入内容：点云列表数据

输出：

- **cloud**：
  - 数据类型：PointCloud
  - 输出内容：点云数据
- **cloud_list**：

- 数据类型：PointCloudList
- 输出内容：点云列表数据

功能演示

使用 CloudFilter 算子**类型**为 CloudSize UpperLimit 模式过滤点云。当点云中点数为 3600 时，输入不同的参考值，分别查看对应的结果。

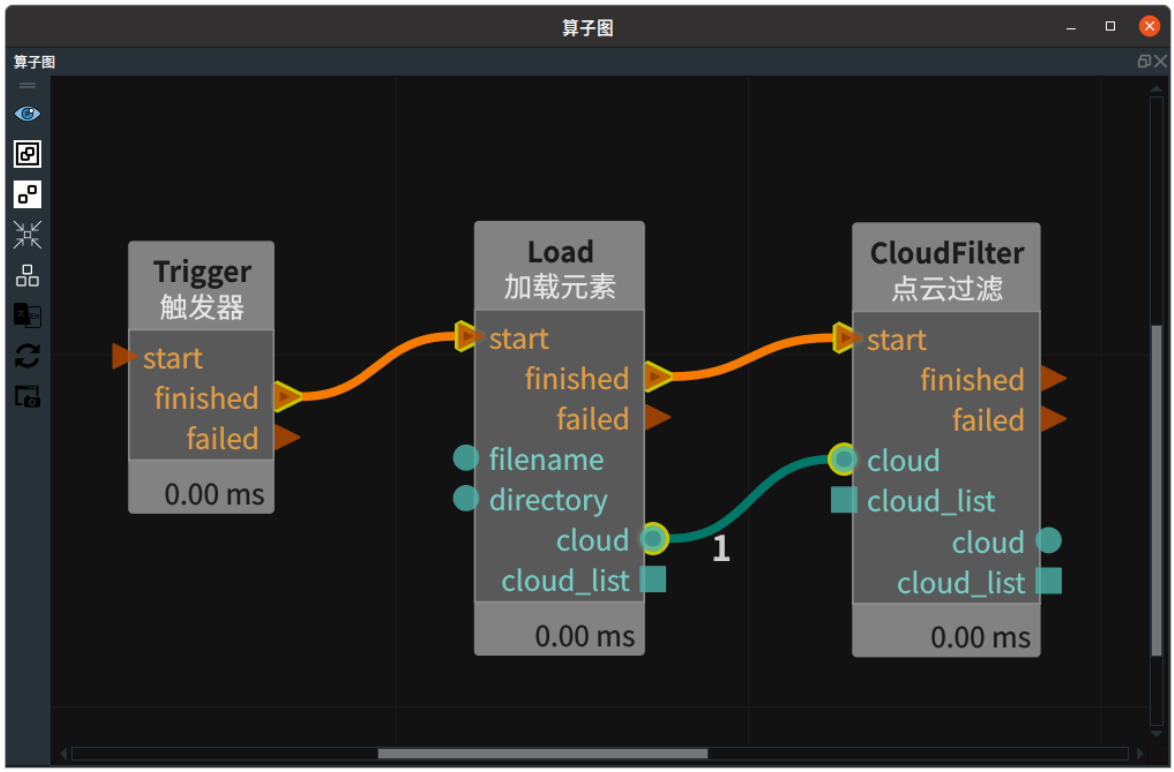
步骤1：算子准备

添加 Trigger 、Load 、CloudFilter 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → PointCloud
   - 文件 → ••• → 选择点云文件名(*example_data/pointcloud/wolf1.pcd* ： 此点云点数：3400)
2. 设置 CloudFilter算子参数：
   - 模式 → UpperLimit
   - 最大点数 → 3600

步骤3：连接算子



步骤4：运行

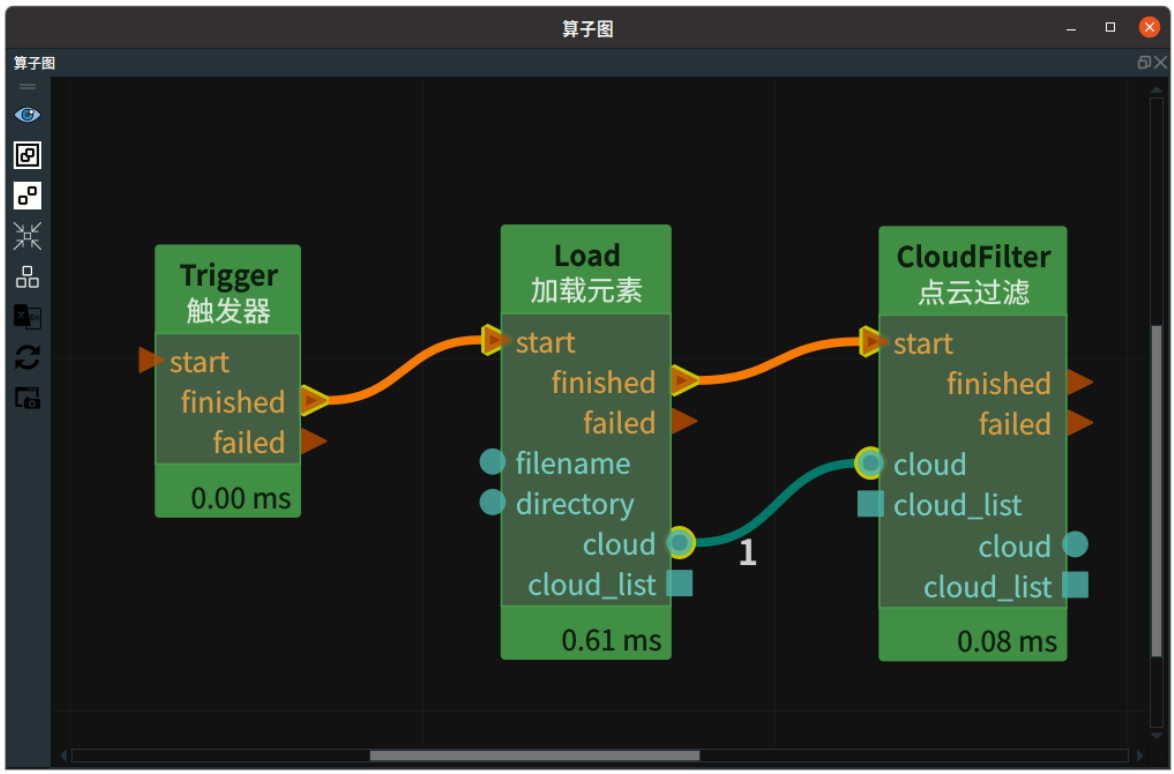
点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

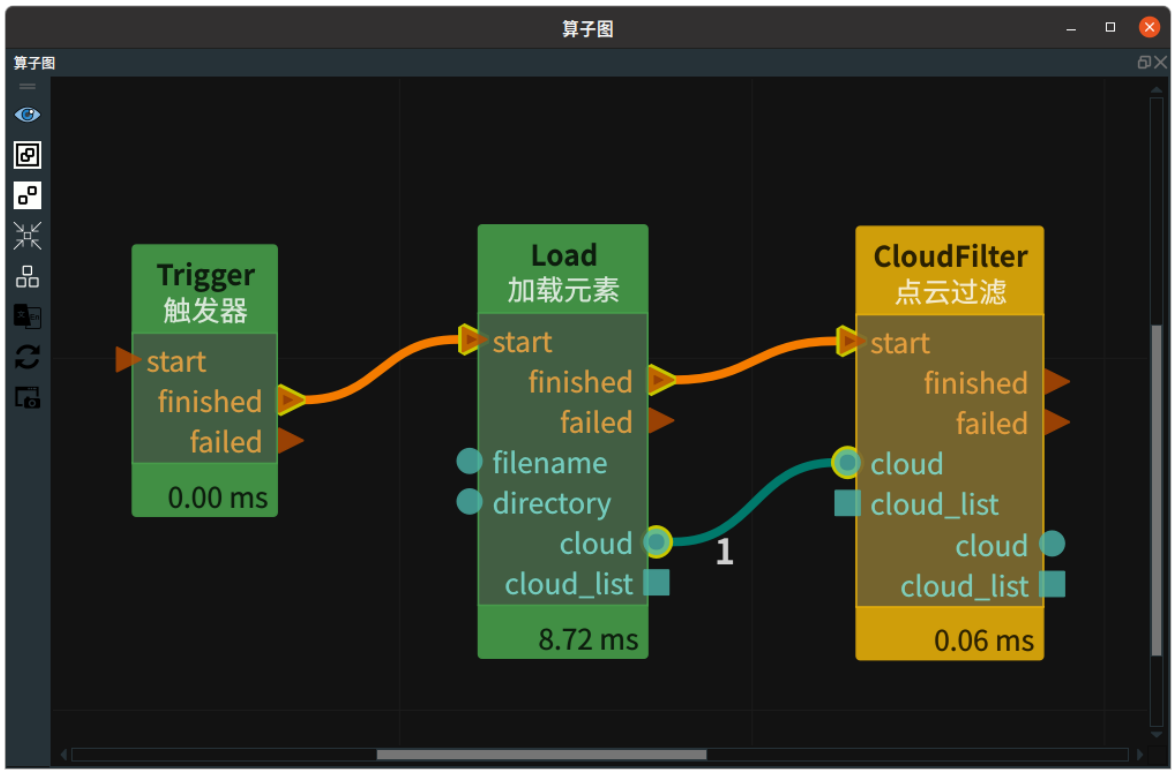
如下图所示，根据参考数值不同，运行结果不同。

参考数值

- 最大点数：3600



- 最小点数：3200 ，点云点数 < 3400，触发failed 信号



AdjustPose 调整位姿

AdjustPose 算子用于调整位姿，包含 InvertPose 、AveragePose 、RotateAxis 、ReferencePose 、NormalizePose 。

类型	功能
InvertPose	对原 pose 取逆。
AveragePose	求取输入 pose 列表的平均 pose 。
RotateAxis	/
ReferencePose	根据参考姿态调整 pose 。
NormalizePose	根据参考轴和角度调整pose的roll 、pitch 、yaw角

InvertPose

将 AdjustPose 算子的 **类型** 设置为 InvertPose ，用于对原 pose 取逆。

算子参数

- **坐标/pose**：设置逆 pose 在 3D 视图中的可视化属性。
  -  打开逆 pose 可视化。
  -  关闭逆 pose 可视化。
  -  设置逆 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。
- **坐标列表/pose_list**：设置逆 pose 列表在 3D 视图中的可视化属性。值描述与 **坐标** 一致。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **pose**：
  - 数据类型：Pose
  - 输入内容：pose 数据
- **pose_list**：
  - 数据类型：PoseList
  - 输入内容：pose 列表数据

输出：

- **pose**：
  - 数据类型：Pose
  - 输出内容：逆 pose 数据
- **pose_list**：
  - 数据类型：PoseList
  - 输出内容：逆 pose 列表数据

功能演示

使用 AdjustPose 算子中 `类型` 属性 InvertPose 将加载的 pose 取逆。

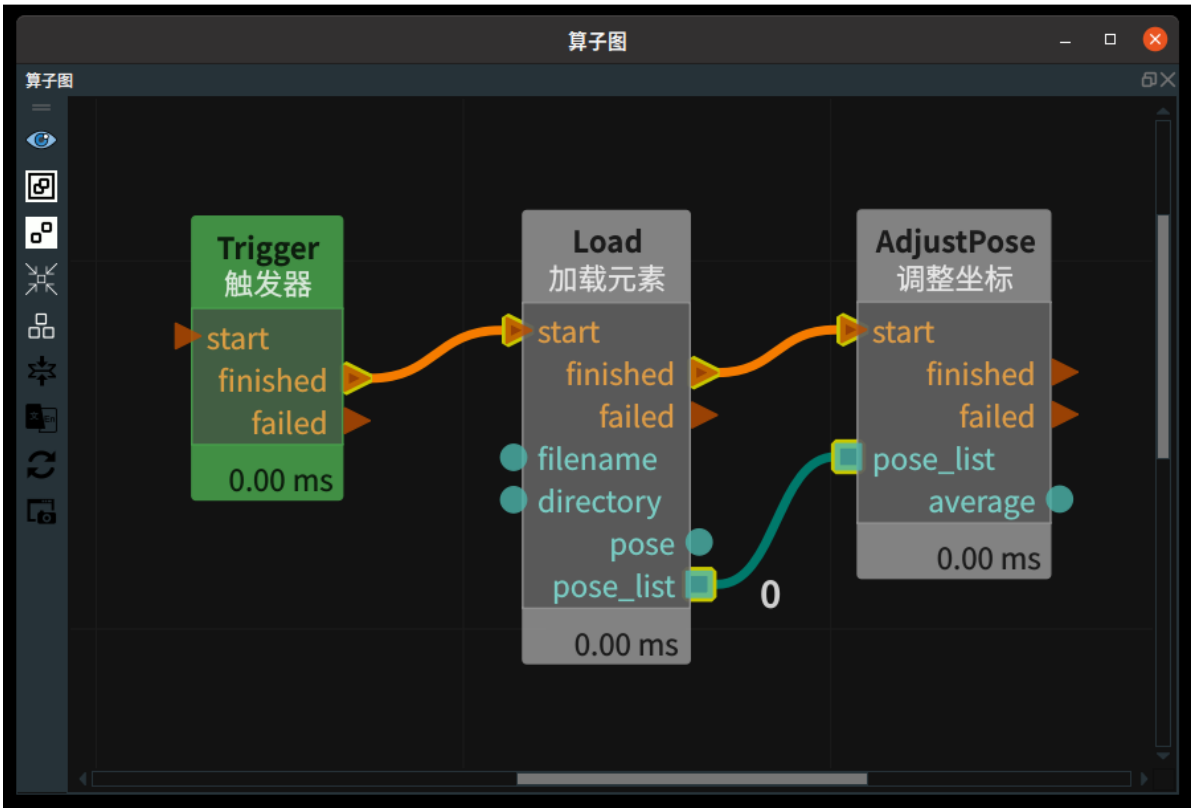
步骤1：算子准备

添加 Trigger 、Load 、AdjustPose 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 类型 → Pose
    - 文件 → ••• →选择 pose 文件名(*example_data/pose/tcp1.txt*)
    - 坐标 → 可视
2. 设置 AdjustPose 算子参数：
    - 类型 → InvertPose
    - 坐标 → 可视

步骤3：连接算子

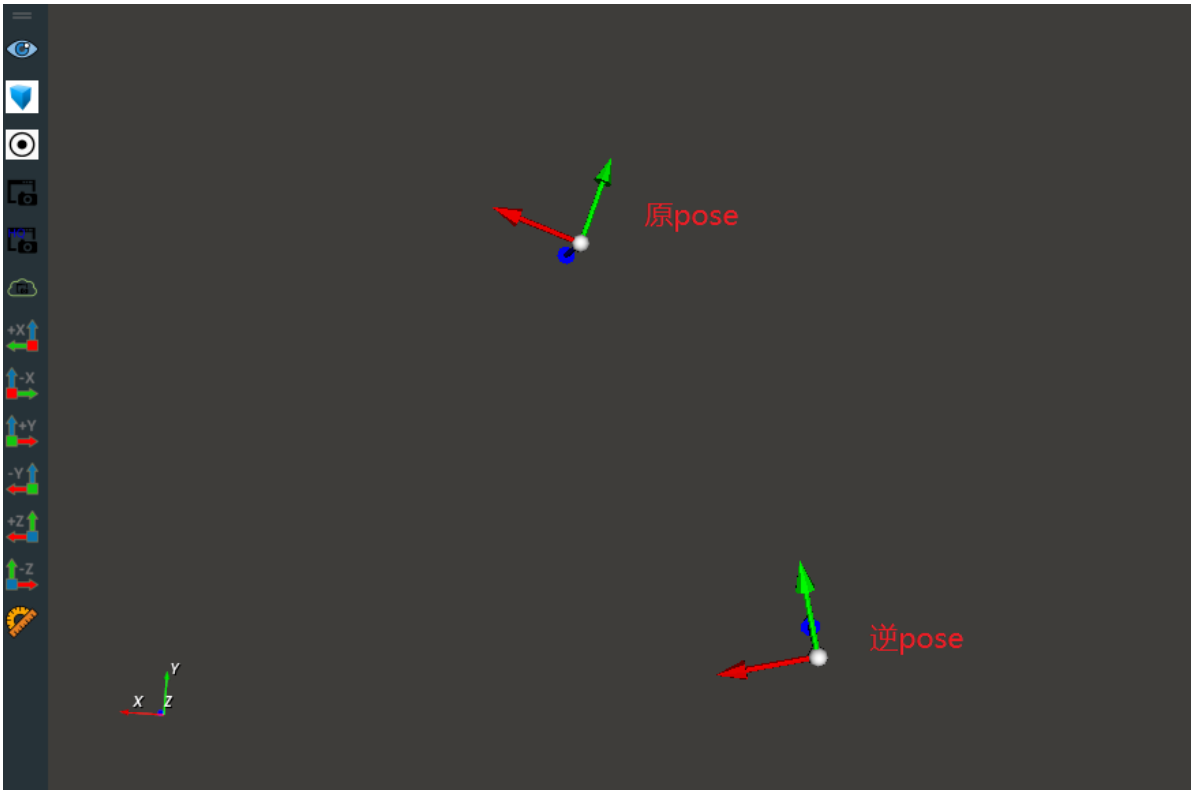


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，3D 视图中显示 Load 算子的 pose 和 AdjustPose 的结果。分别为加载的 pose 及其逆 pose 。



AveragePose

将 AdjustPose 算子的 **类型** 属性设置为 AveragePose ，用于求取输入 pose 列表的平均 pose 。

项目中，主要用于对同一目标多次计算所得的相近位姿求取平均作为最终结果位姿。

其中的位置信息 xyz 数值直接求平均值。而姿态信息对应的三个旋转角计算过程相对复杂，仅在输入 pose 的旋转角彼此相差不大时具有实际意义，一般彼此之间的旋转角差值标准差不超过 0.3 弧度为佳。

算子参数

- **坐标/pose**：设置平均 pose 在 3D 视图中的可视化属性。
  - 打开平均 pose 可视化。
  - 关闭平均 pose 可视化。
  - 设置平均 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

- **pose_list**：
  - 数据类型：Pose
  - 输入内容：pose 数据

输出：

- **average**：
  - 数据类型：Pose
  - 输出内容：平均 pose 数据

功能演示

使用 AdjustPose 算子中 `类型` 属性 AveragePose 求取加载的三个 pose 的平均 pose 。

步骤1：算子准备

添加 Trigger 、Load 、AdjustPose 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 类型 → Pose
    - 文件 → ••• → 选择 pose 文件目录名(*example_data/averagepose.txt*)
    - 坐标列表 →  可视
2. 设置 AdjustPose 算子参数：
    - 类型 → AveragePose
    - 坐标 →  可视

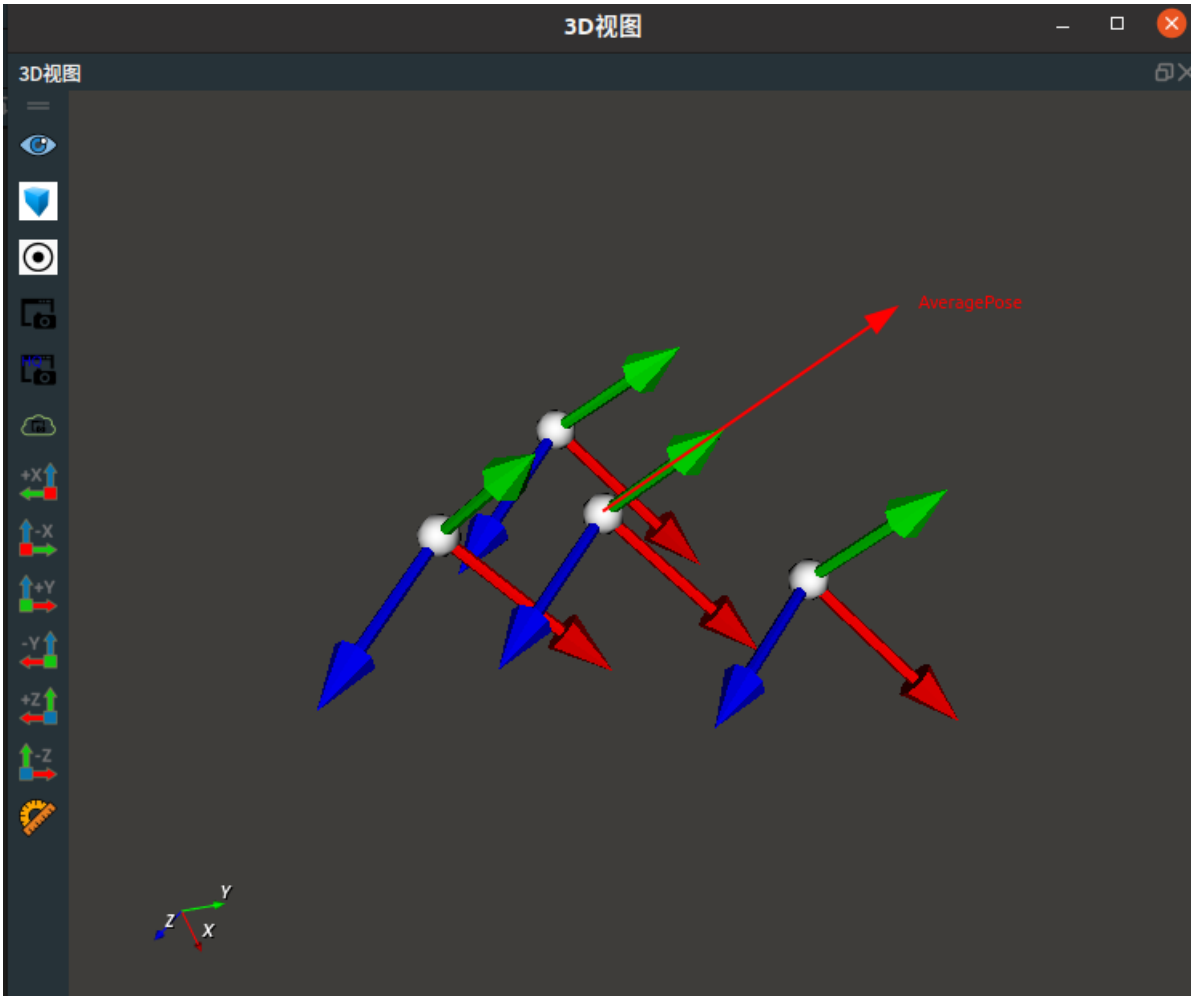
步骤3：连接算子

步骤4：运行

点击 RVS 的运行按钮，触发 Trigger 算子。

运行结果

如下图所示，3D 视图中显示 Load 算子和 AdjustPose 的结果。分别为加载的三个 pose 和平均 pose 后的结果。



ReferencePose

将 AdjustPose 算子的 `类型` 属性设置为 ReferencePose ，用于根据参考姿态调整 pose 姿态。

算子参数

- `坐标/pose`：设置调整姿态后的 pose 在3D视图中的可视化属性。
  - 打开 pose 可视化。
  - 关闭 pose 可视化。
  - 设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

- `pose`：
  - 数据类型：Pose
  - 输入内容：pose 数据
- `ref_pose`：
  - 数据类型：Pose
  - 输入内容：参考 pose 数据

输出：

- `pose`：

- 数据类型：Pose
- 输出内容：调整姿态后 pose 数据

功能演示

使用AdjustPose算子中 **类型** 属性 ReferencePose 根据参考姿态调整 pose 。

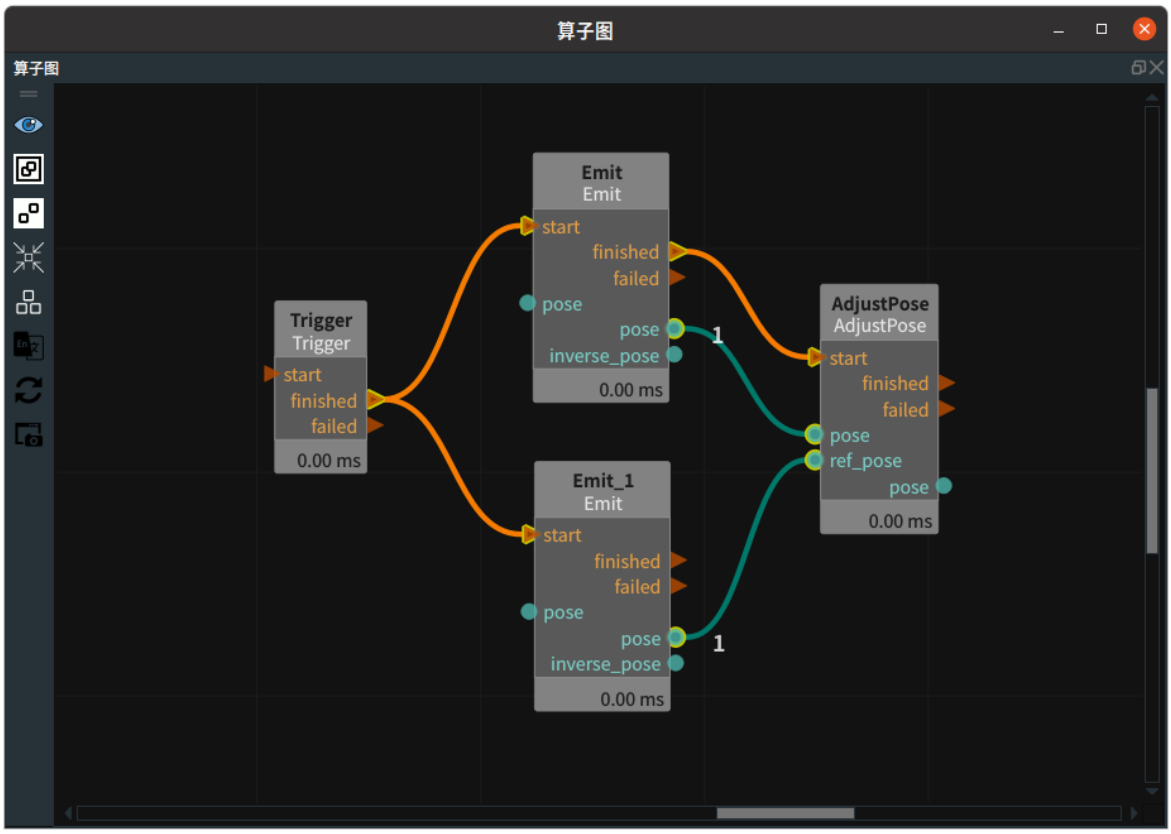
步骤1：算子准备

添加 Trigger 、Emit（2个） 、AdjustPose 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
   - 类型 → Pose
   - 坐标 → 0 0 0 0 0 0
   - 坐标 → 可视
2. 设置 Emit_1 算子参数：
   - 类型 → Pose
   - 坐标 → 0.5 0. 1 0.1 1.57 1.57 -1.57
   - 坐标 → 可视
3. 设置 AdjustPose 算子参数：
   - 类型 → ReferencePose
   - 坐标 → 可视 → 0.2

步骤3：连接算子

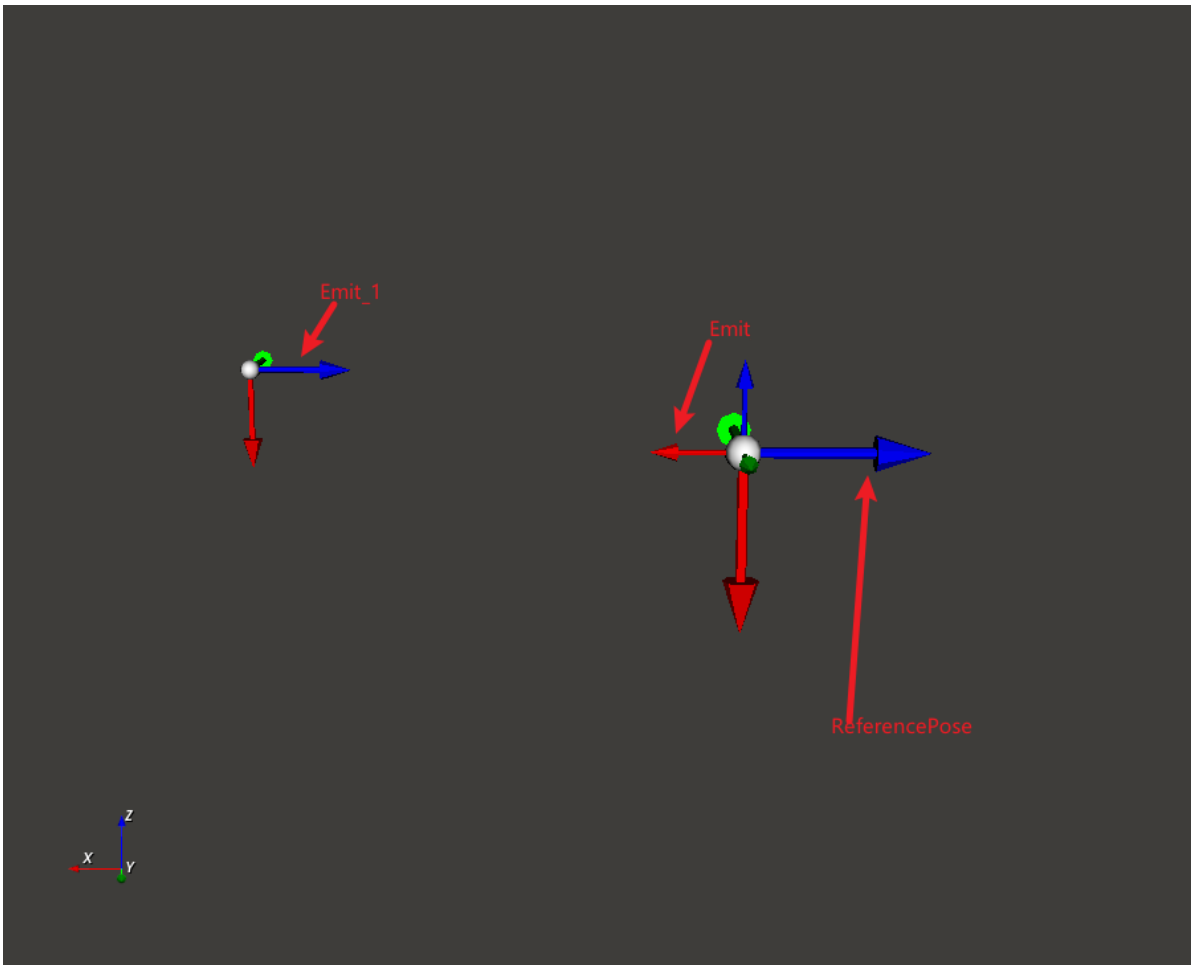


步骤4：运行

点击 RVS 的运行按钮，触发 Trigger 算子。

运行结果

如下图所示，3D 视图中显示 Emit 算子以及 AdjustPose 算子的结果。



NormalizePose

将 AdjustPose 算子中将 **类型** 设置为 NormalizePose ，用于计算并输出 pose 的两种不同姿态角表达形式。通常来说，同一姿态角在 -π 到 π 的定义域区间内至少有两种表达形式，其中一种姿态角的表达形式无法用机器人直接移动，因此需要筛选某一旋转轴的最优旋转角度，通过参数 ref_rot_axis 和 ref_angle 进行判断。原理是判断两种表达形式的某一旋转轴（由 ref_rot_axis 设置）是否更接近于参考值 ref_angle ，分别输出为 pose_positive 和 pose_negative 。判断旋转角度是否更接近需符合旋转角度周期原则，即 -π 和 π 认为是同一个角度，若参考值为 -3，那么认为 π 比 0 更接近于 -3 。

算子参数

- **参考旋转轴/ref_rot_axis**：参考轴：X / Y / Z 。
- **参考角度/ref_angle**：参考弧度。
- **正向坐标/pose_positive**：设置pose 最优表达形式在 3D 视图中的可视化属性。
  - 打开调整后 pose 位姿可视化。
  - 关闭调整后 pose 位姿可视化。
  - 设置调整后 pose 位姿的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。
- **负向坐标/pose_negative**：设置 pose 另一种表达形式在 3D 视图中的可视化属性。参数值与 pose_positive 描述一致。
- **pose_positive_list**：设置pose 列表最优表达形式在 3D 视图中的可视化属性。参数值与 pose_positive 描述一致。

- `pose_negative_list`：设置 pose 列表另一种表达形式在 3D 视图中的可视化属性。参数值与 pose_positive 描述一致。

数据信号输入输出

输入：

- `pose`：
  - 数据类型：Pose
  - 输入内容：pose 数据
- `pose_list`：
  - 数据类型：PoseList
  - 输入内容：pose 列表数据

输出：

- `pose_positive`：
  - 数据类型：Pose
  - 输出内容：姿态调整后与参考轴和弧度相近的 pose 数据
- `pose_negative`：
  - 数据类型：Pose
  - 输出内容：姿态调整后负向 pose 数据
- `pose_positive_list`：
  - 数据类型：PoseList
  - 输出内容：姿态调整后与参考轴和弧度相近的 pose 列表数据
- `pose_negative_list`：
  - 数据类型：PoseList
  - 输出内容：姿态调整后负向 pose 列表数据

功能演示

使用 AdjustPose 算子中 `类型` 属性 NormalizePose 根据 x 轴 在参考角度 3(弧度值) 进行姿态调整。

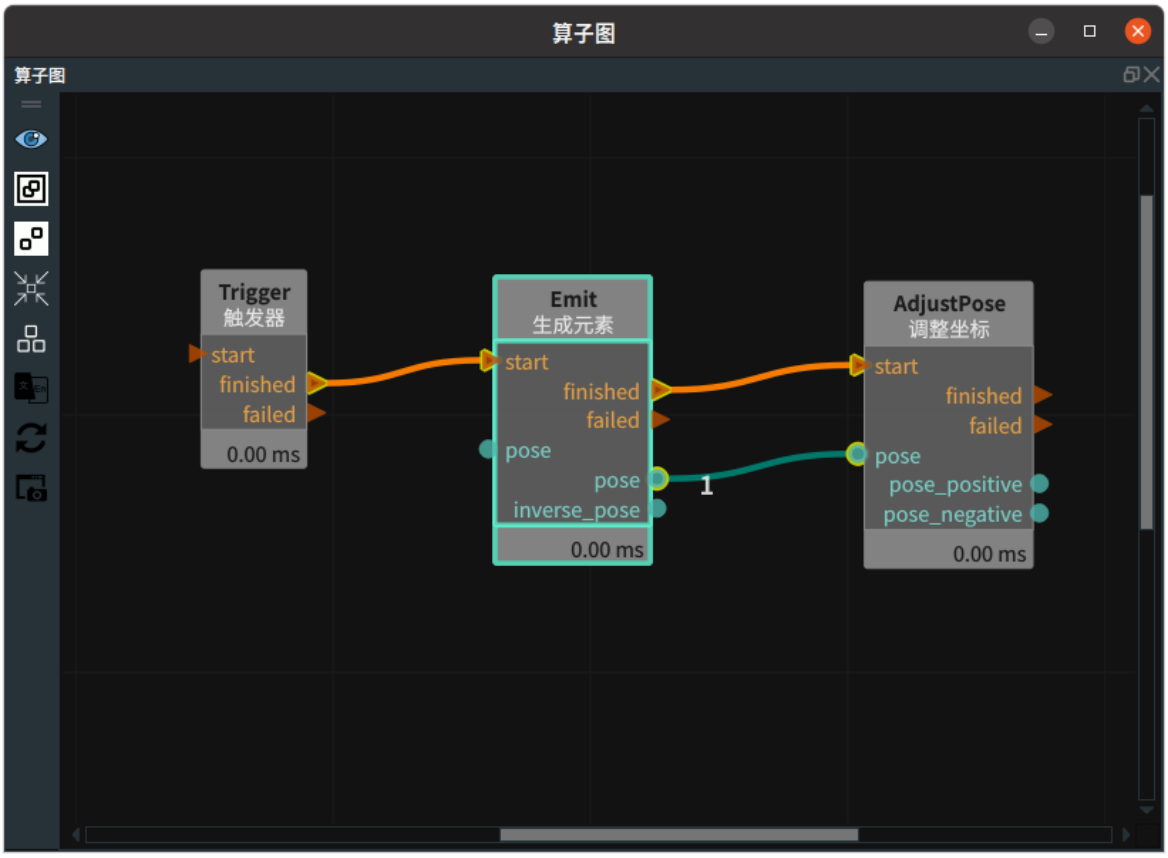
步骤1：算子准备

添加 Trigger、Emit 、AdjustPose 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
   - 类型 → Pose
   - 坐标 → 1 2 0.5 0.5 0.1 0.3
   - 坐标 →  可视
2. 设置 AdjustPose 算子参数：
   - 类型 → NormalizePose
   - 参考旋转轴 → x
   - 参考角度 → 3
   - 正向坐标 →  可视
   - 负向坐标 →  可视

步骤3：连接算子

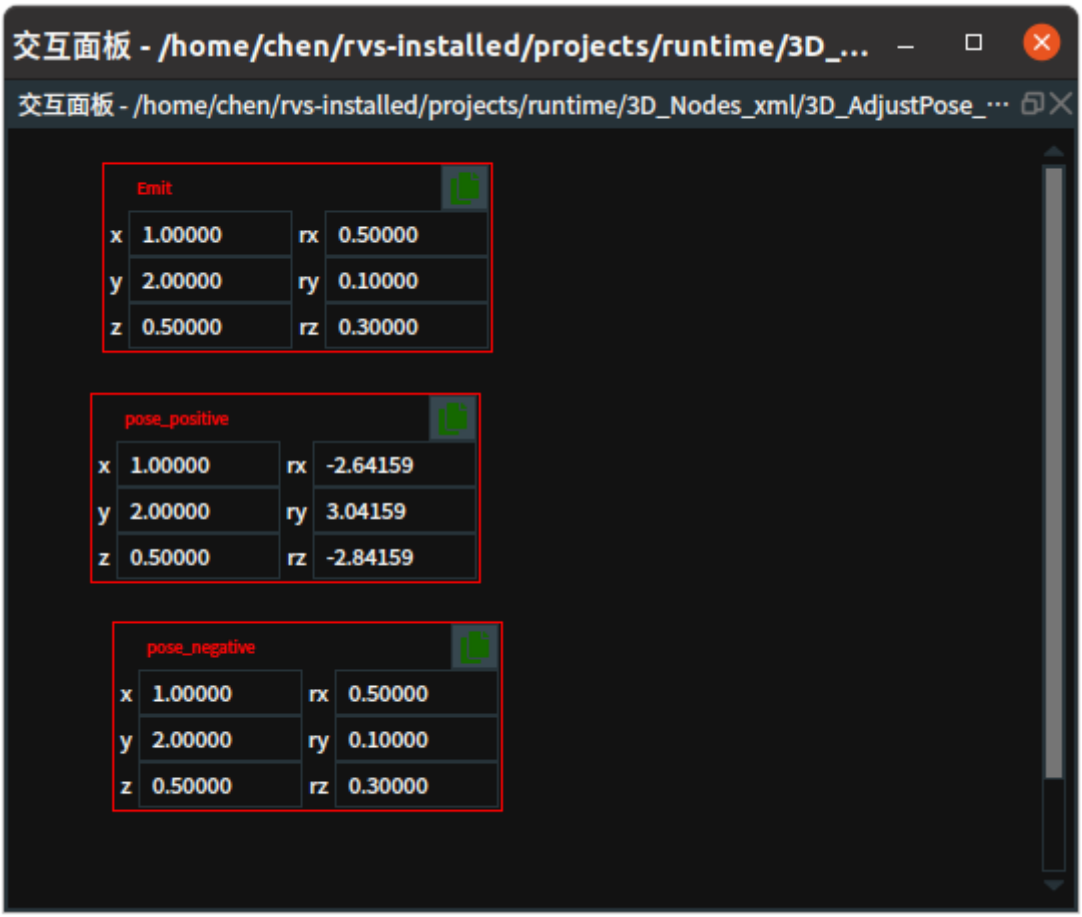


步骤4：运行

1. 点击打开 RVS 的运行按钮，触发 Trigger 算子。
2. 分别将三个 pose 与交互面板中的输出工具“ pose 输出” 进行绑定。

运行结果

如下图所示，在交互面板中显示 Emit 算子和 AdjustPose 算子输出端口的结果。


交互面板 - /home/chen/rvs-installed/projects/runtime/3D_...
交互面板 - /home/chen/rvs-installed/projects/runtime/3D_Nodes_xml/3D_AdjustPose_...
Emit
x 1.00000
y 2.00000
z 0.50000
rx 0.50000
ry 0.10000
rz 0.30000
pose_positive
x 1.00000
y 2.00000
z 0.50000
rx -2.64159
ry 3.04159
rz -2.84159
pose_negative
x 1.00000
y 2.00000
z 0.50000
rx 0.50000
ry 0.10000
rz 0.30000

GeometrySample 几何采样

GeometrySample算子用于对指定的几何目标进行等分。该算子将指定的几何目标划分为数个等分段，用于检测和识别几何目标。

type	功能
Line	将线段等分，输出分割点的坐标。
Circle	将圆的圆周等分，输出分割点的坐标。
CirclePart	起始点和结束点到圆心的连线投影到圆圈上，与圆圈生成 2 个交点，等分 2 个点之间的圆周部分，输出分割点的坐标。

Line

将 GeometrySample 算子 **类型** 设置为Line，用于将线段 sample_num-1 等分，输出一系列分割点(Pose 形式) 。

Line 分割后的每一个 pose ，其 X 轴都沿着 Line 的线段方向。

算子参数

- **采样数量/sample_num**：将线段分割成 (sample_num-1) 段，输出 sample_num 个分割点。
- **坐标列表/pose_list**：对分割点列表进行显示控制。
  -  打开分割点列表可视化。
  -  关闭分割点列表可视化。
  -  设置分割点列表的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

- **line**：
  - 数据类型：Line
  - 输入内容：线段

输出：

- **poses**：
  - 数据类型：PoseList
  - 输出内容：N 等分的分割点

功能演示

使用 GeometrySample 算子中 Line ，将生成的线段分成 4 等分，并输出一系列分割点(Pose 形式)。

步骤1：算子准备

添加 Trigger 、Emit 、Emit_1 、Emit_2 、GeometrySample 算子至算子图。

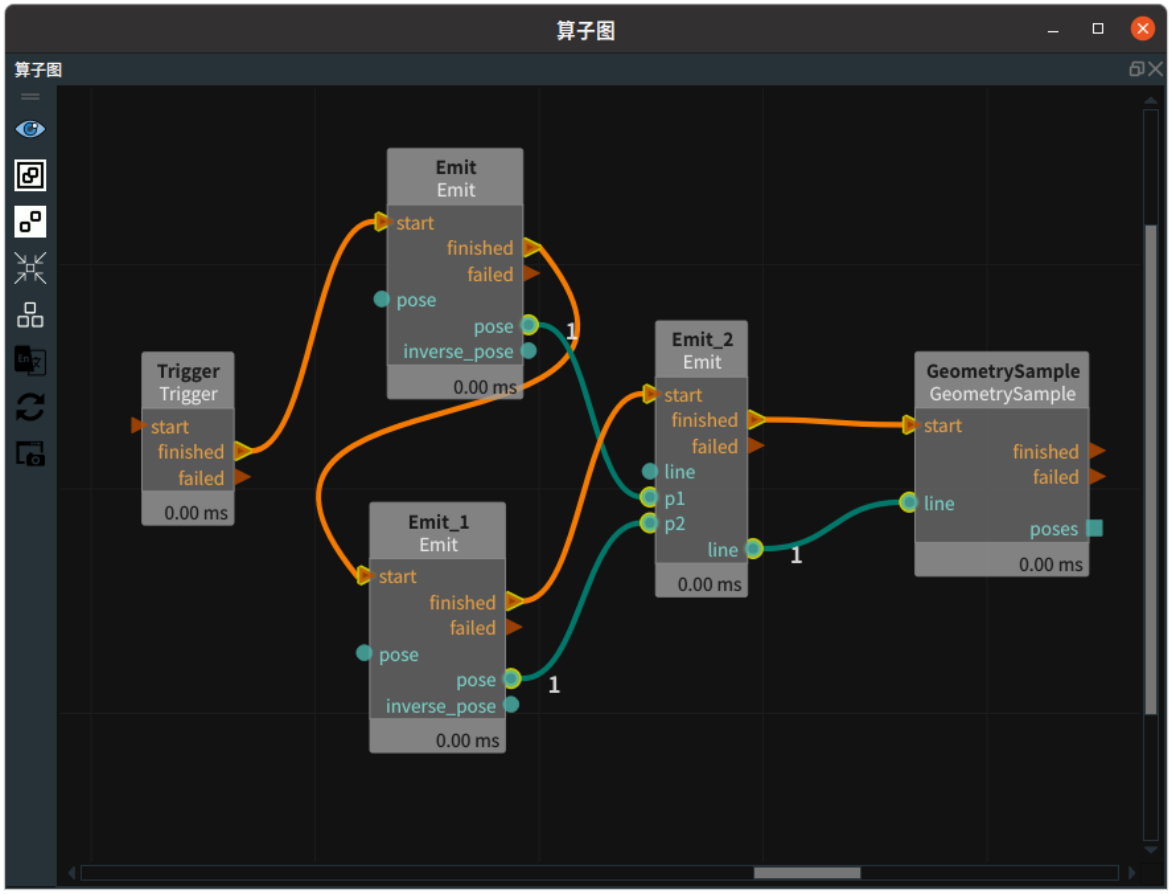
步骤2：设置算子参数

1. 设置 Emit 算子参数：

- 类型 → Pose
- 坐标 → 0.1 0.08 0 1 0 0

2. 设置 Emit_1 算子参数：
   - 类型 → Pose
   - 坐标 → 0.03 0.1 0.05 0 1 0
3. 设置 Emit_2 算子参数：
   - 类型 → Line
   - 线段 →  可视
4. 设置 GeometrySample 算子参数
   - 类型 →Line
   - 采样数量 → 5
   - 坐标列表 →  可视 →  0.01

步骤3：连接算子

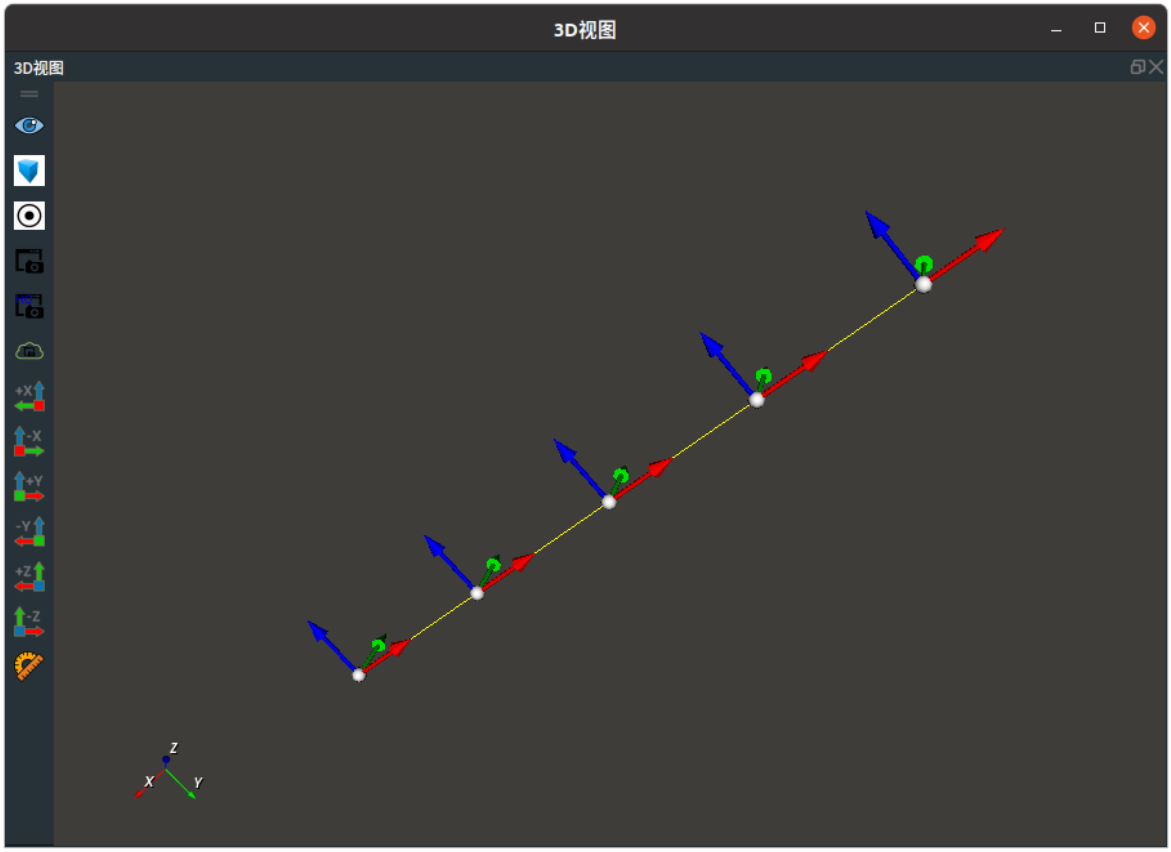


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

结果如下图所示，线段被分成 4等分，并输出一系列分割点。



Circle

将 GeometrySample算子 **类型** 设置为 Circle ，用于将圆的圆周 sample_num 等分，输出一系列分割点 (Pose 形式) 。Circle分割后的每一个 pose ，其 Z 轴都垂直于圆平面。

算子参数

- **采样数量/sample_num**：将圆分割成 sample_num 段，输出 sample_num 个分割点。取值范围：[1,+∞]。默认值：2。
- **other_rot**：设置生成的 pose_list 方向。
  - True：按照圆心 pose 顺时针方向。
  - False：按照圆心 pose 逆时针方向。
- **中心坐标/center_pose**：设置圆心在 3D 视图中的可视化属性。
  - 打开圆心 pose 可视化。
  - 关闭圆心 pose 可视化。
  - 设置圆心的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。
- **坐标列表/pose_list**：设置分割点列表在3D视图中的可视化属性。
  - 打开分割点列表可视化。
  - 关闭分割点列表可视化。
  - 设置分割点列表的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

- `circle`：
  - 数据类型：Circle
  - 输入内容：圆

输出：

- `center_pose`：
  - 数据类型：Pose
  - 输出内容：圆心
- `poses`：
  - 数据类型：PoseList
  - 输出内容：圆周分割点

功能演示

使用 GeometrySample 算子 中 Circle ，用于将生成的圆的圆周分成 5 等分，并输出一系列分割点(Pose 形式)。

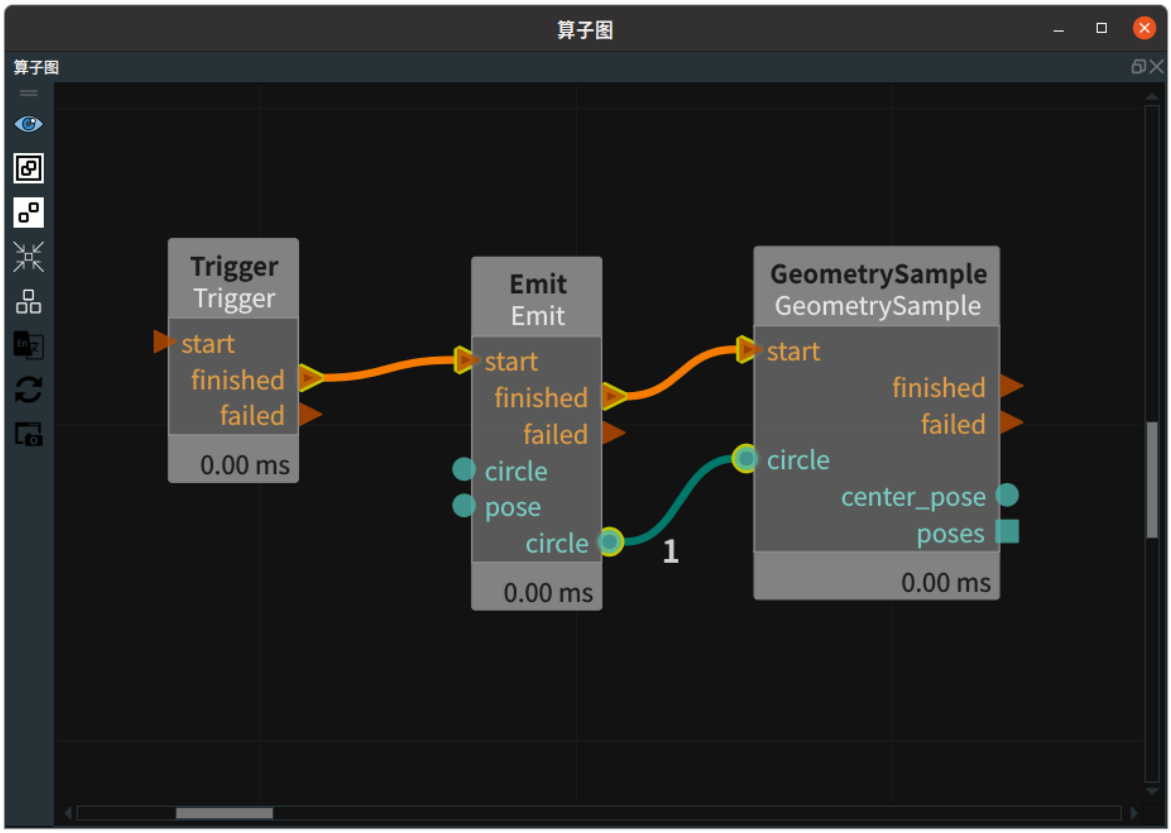
步骤1：算子准备

添加 Trigger 、Emit 、GeometrySample 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
   - 类型 → circle
   - 半径→ 0.3
   - 圆圈 →  可视
2. 设置 GeometrySample 算子参数：
   - 类型 → circle
   - 采样数量 → 5
   - 坐标列表 →  可视

步骤3：连接算子

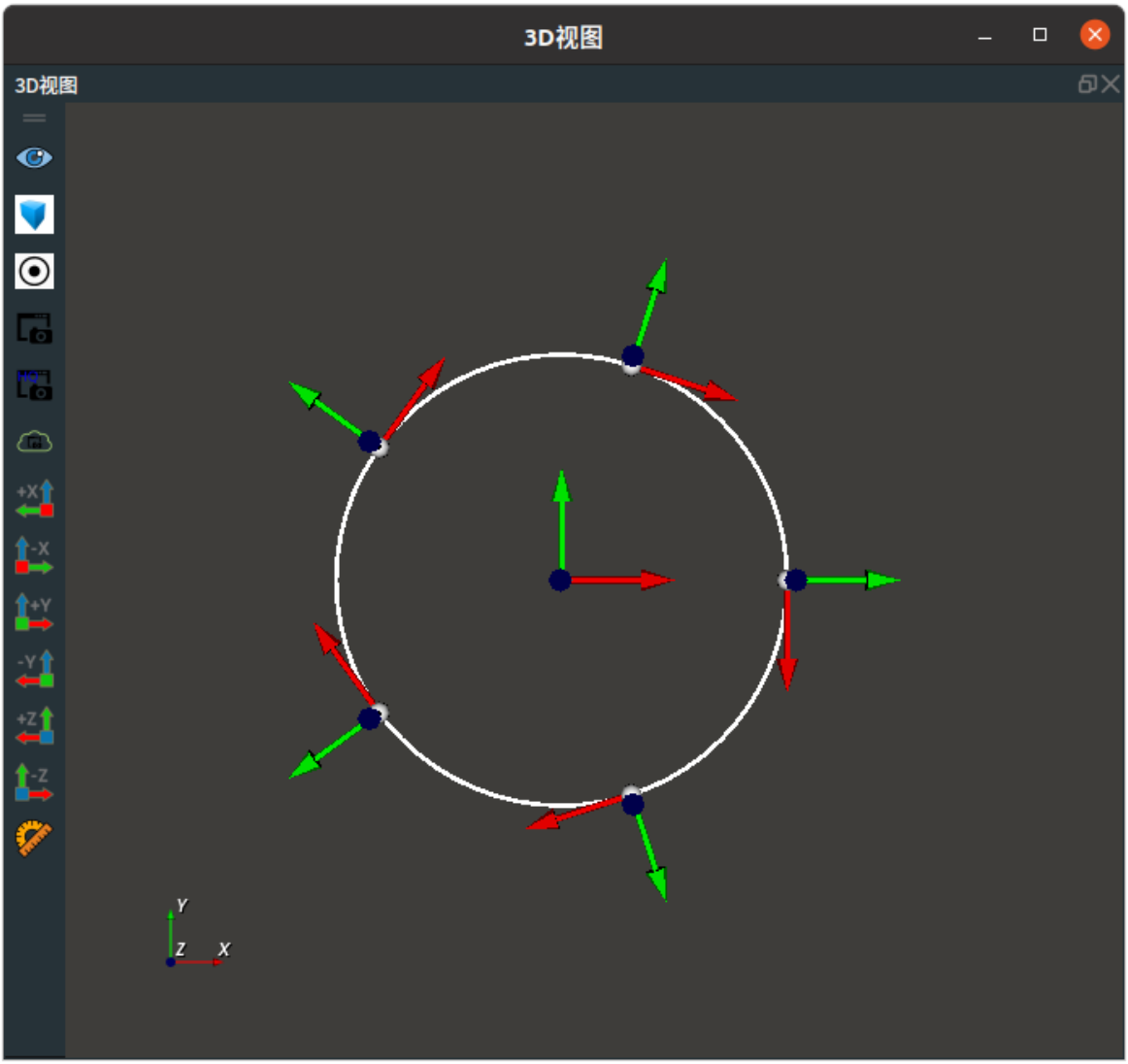


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

结果如下图所示，用于将圆的圆周分成 5 等分，并显示 5 个分割点。



CirclePart

将 GeometrySample 算子**类型**选择 CirclePart， 起始点和结束点到圆心的连线投影到圆圈上，与圆圈生成 2 个交点，等分 2 个点之间的圆周部分，输出分割点的坐标。

算子参数

- **采样数量/sample_num**：将选取的圆周分割成 （sample_num-1） 段，输出 sample_num 个分割点。取值范围：[1,+∞]。默认值：2。
- **another_part**：设置圆周等分的区域。
  - True： 圆中心 pose 的正方向（右手法则）。
  - False： 圆中心 pose 的反方向（右手法则）。
- **中心坐标/center_pose**：设置圆心在 3D 视图中的可视化属性。
  - 打开圆心 pose 可视化。
  - 关闭圆心 pose 可视化。
  - 设置圆心的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。
- **坐标列表/poses**：设置分割点列表在3D视图中的可视化属性。
  - 打开分割点列表可视化。
  - 关闭分割点列表可视化。
  - 设置分割点列表的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

- `circle`：
  - 数据类型：Circle
  - 输入内容：圆
- `start_point`：
  - 数据类型：Pose
  - 输入内容：圆周的起始角度
- `circle`：
  - 数据类型：Pose
  - 输入内容：圆周的结束角度

输出：

- `center_pose`：
  - 数据类型：Pose
  - 输出内容：圆心
- `poses`：
  - 数据类型：PoseList
  - 输出内容：选取圆周分割点

功能演示

使用 GeometrySample 算子 中 CirclePart ，起始点和结束点到圆心的连线投影到圆圈上，与圆圈生成 2 个交点，等分 2 个点之间的圆周部分，输出5个分割点的坐标。

步骤1：算子准备

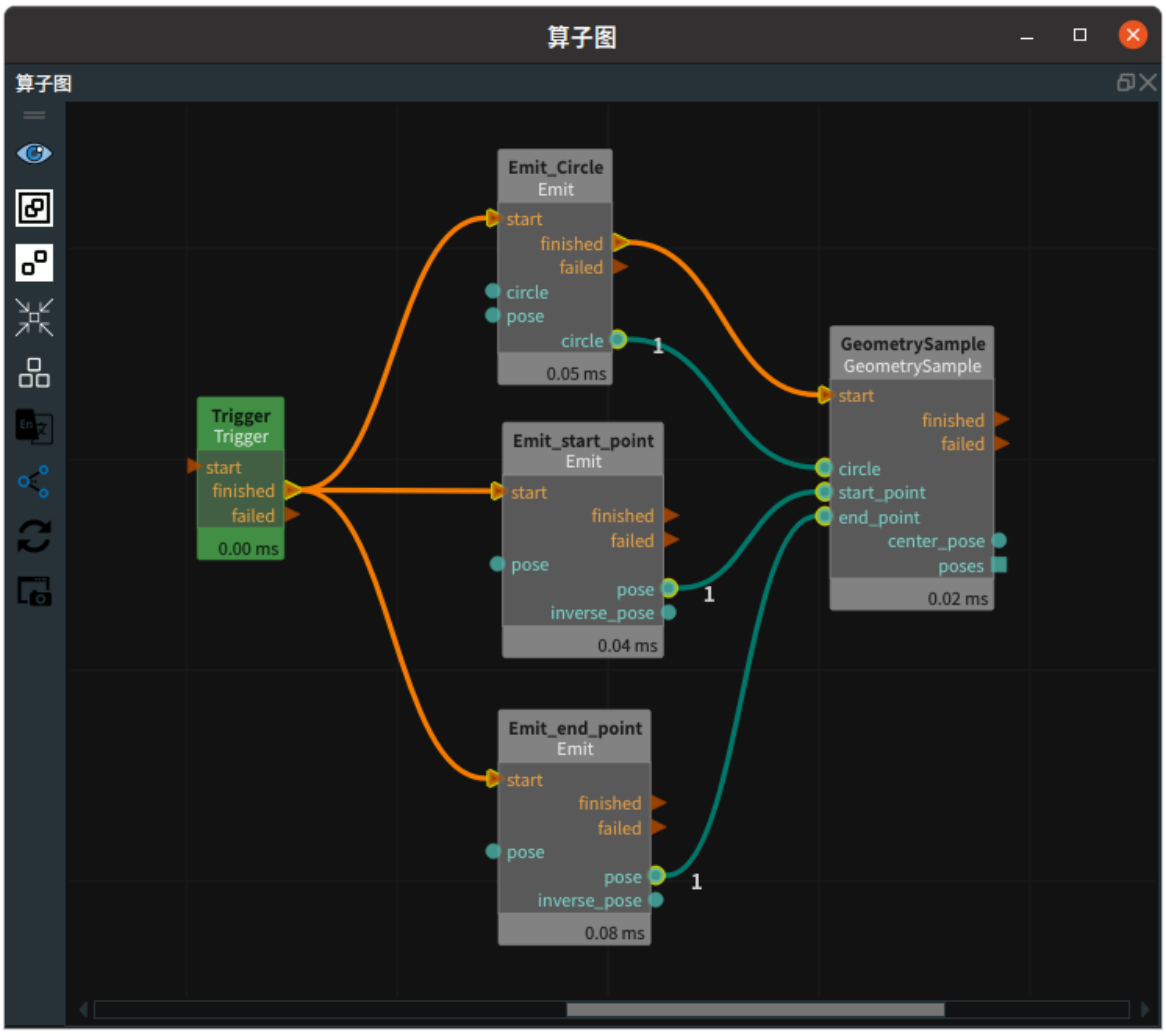
添加 Trigger 、Emit (3个）、GeometrySample 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
   - 算子名称 → Emit_Circle
   - 类型 → circle
   - 圆圈 →  可视
2. 设置 Emit_1 算子参数：
   - 算子名称 → Emit_start_point
   - 类型 → pose
   - 坐标 →1 0 0 0 0 0
   - 坐标 →  可视
3. 设置 Emit_2 算子参数：
   - 算子名称 → Emit_end_point
   - 类型 → pose
   - 坐标 →-0.5 0.5 0 0 0 0
   - 坐标 →  可视
4. 设置 GeometrySample 算子参数：
   - 类型 → CirclePart
   - 采样数量 → 5

- 中心坐标 → 可视 → 0.5
- 坐标列表 → 可视 → 0.3

步骤3：连接算子

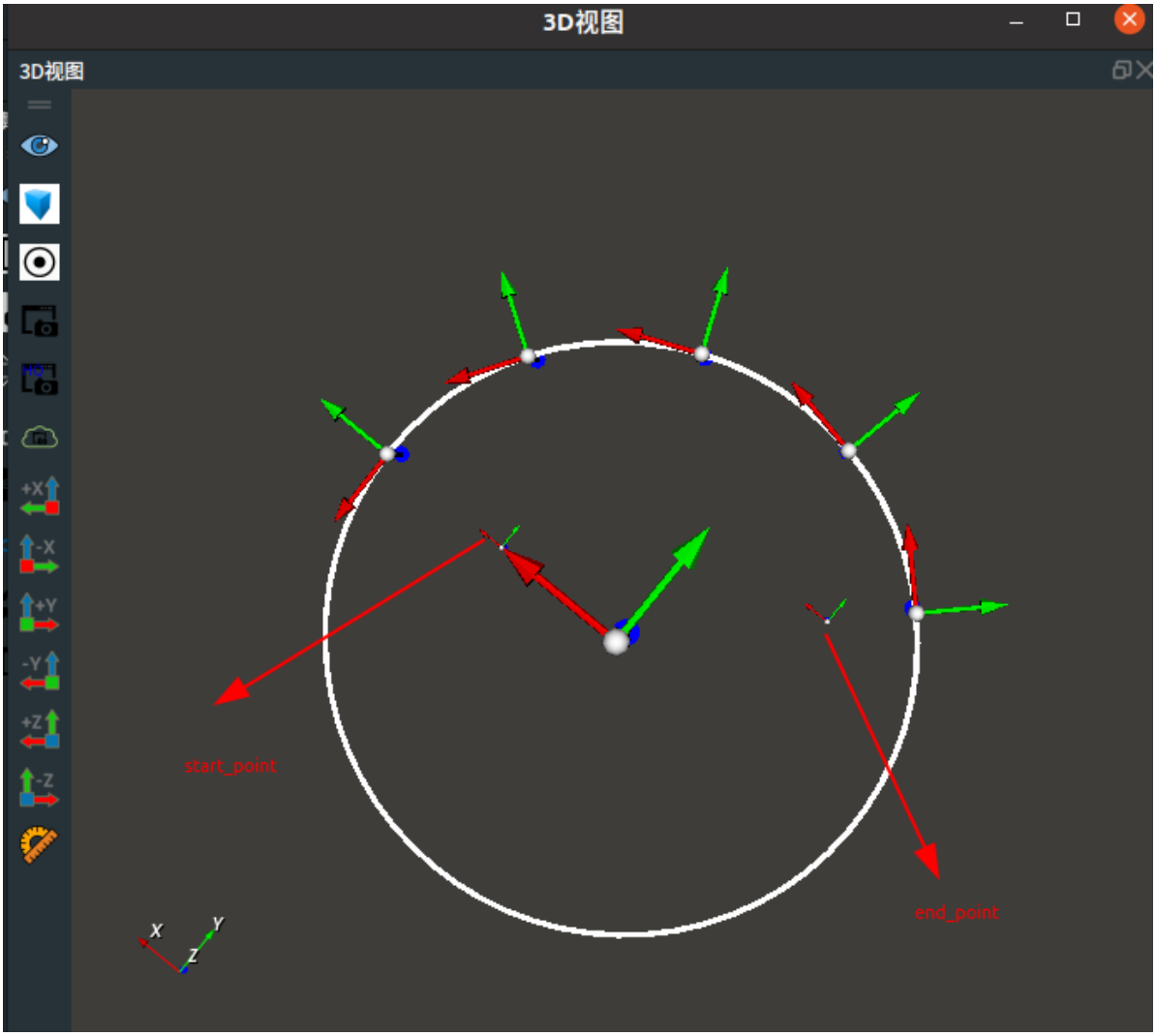


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

结果如下图所示， 起始点和结束点到圆心的连线投影到圆圈上，与圆圈生成 2 个交点，等分 2 个点之间的圆周部分，输出了5个分割点的坐标。


3D视图
3D视图
start_point
end_point
X
Y
Z

ConvertPose 位姿转换

ConvertPose 算子作用于对位姿进行转换。

类型	功能
PoseToEuler	pose 默认的旋转方式绕着 Z-Y-X 的顺序对固定轴进行旋转，PoseToEuler 算子用于将 pose 转换成欧拉角（绕着X-Y-Z的顺序对固定轴进行旋转）
EulerToPose	用于将欧拉角转换成 pose。
PoseToRotateVector	用于将 pose 转成旋转矢量。主要用于转换成优傲机器人所接受的表达方式。
RotateVectorToPose	用于将旋转矢量转成 pose 。
PoseToMatrix	用于将 pose 转换成变换矩阵。
MatrixToPose	用于将变换矩阵转换成 pose 。

PoseToEuler

将 ConvertPose 算子中 **类型** 设置为 PoseToEuler ，用于将 pose 转换成欧拉角（绕着 X - Y - Z 的顺序对固定轴进行旋转）

算子参数

- **第一旋转轴/asix_1**：调整旋转顺序，轴1 。默认值：X 。
- **第二旋转轴/axis_2**：调整旋转顺序，轴2 。默认值：Y 。
- **第三旋转轴/axis_3**：调整旋转顺序，轴3 。默认值：Z 。

  说明：三个轴的顺序表示旋转顺序，如上面的顺序表示为 X-Y-Z。
- **欧拉角/euler**：设置欧拉角在 3D 视图中的可视化属性。
  -  打开欧拉角可视化。
  -  关闭欧拉角可视化。
  -  设置欧拉角的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

- **pose**：
  - 数据类型：Pose
  - 输入内容：pose 数据 X 、Y 、Z 、yaw 、pitch 、roll

输出：

- **euler**：
  - 数据类型：Pose
  - 输出内容：欧拉角数据

功能演示

使用 ConvertPose 算子中 PoseToEuler 将加载的 pose 转换成欧拉角。

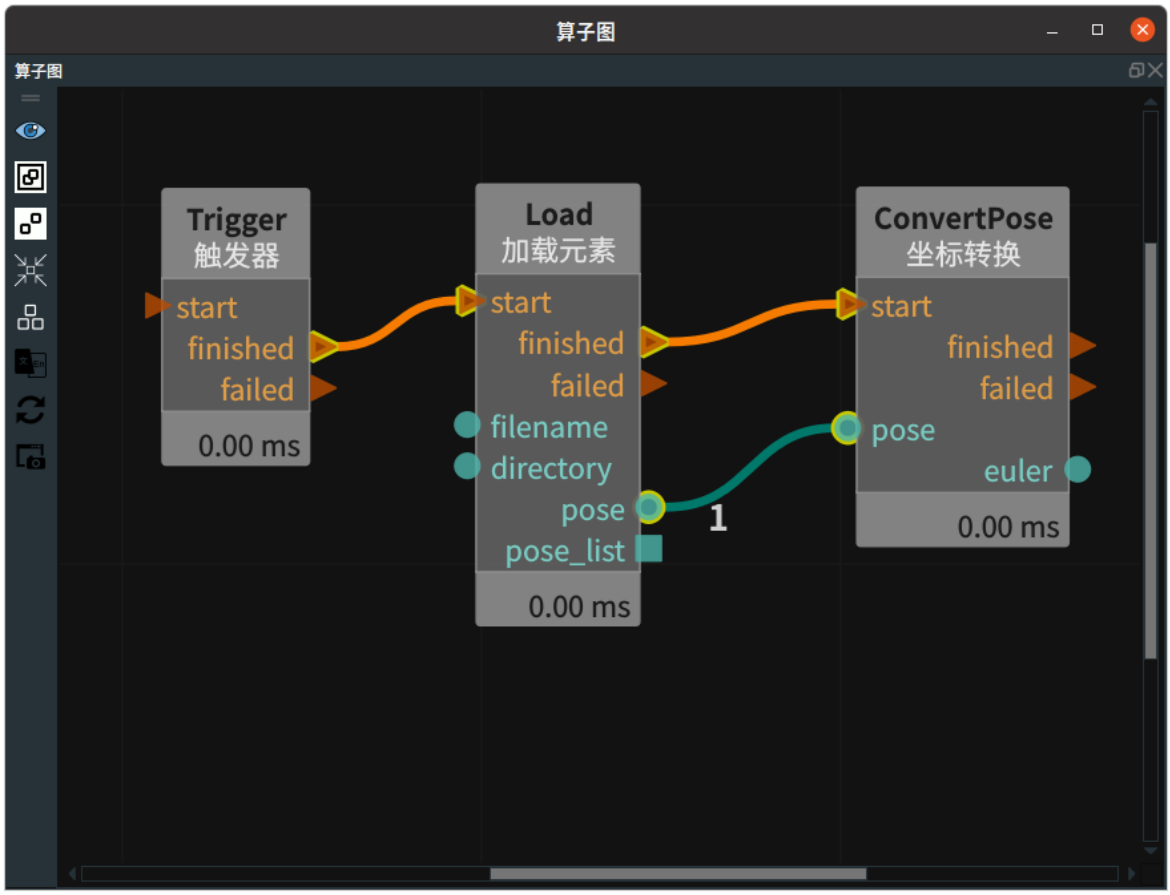
步骤1：算子准备

添加 Trigger 、Load 、ConvertPose 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 类型 → pose
    - 文件 → ••• → 选择 pose 文件名(*example_data/pose/tcp1.txt*)
    - 坐标 → 可视
2. 设置 ConvertPose 算子参数：
    - 类型 → PoseToEuler
    - 坐标 → 可视 → 0.2

步骤3：连接算子

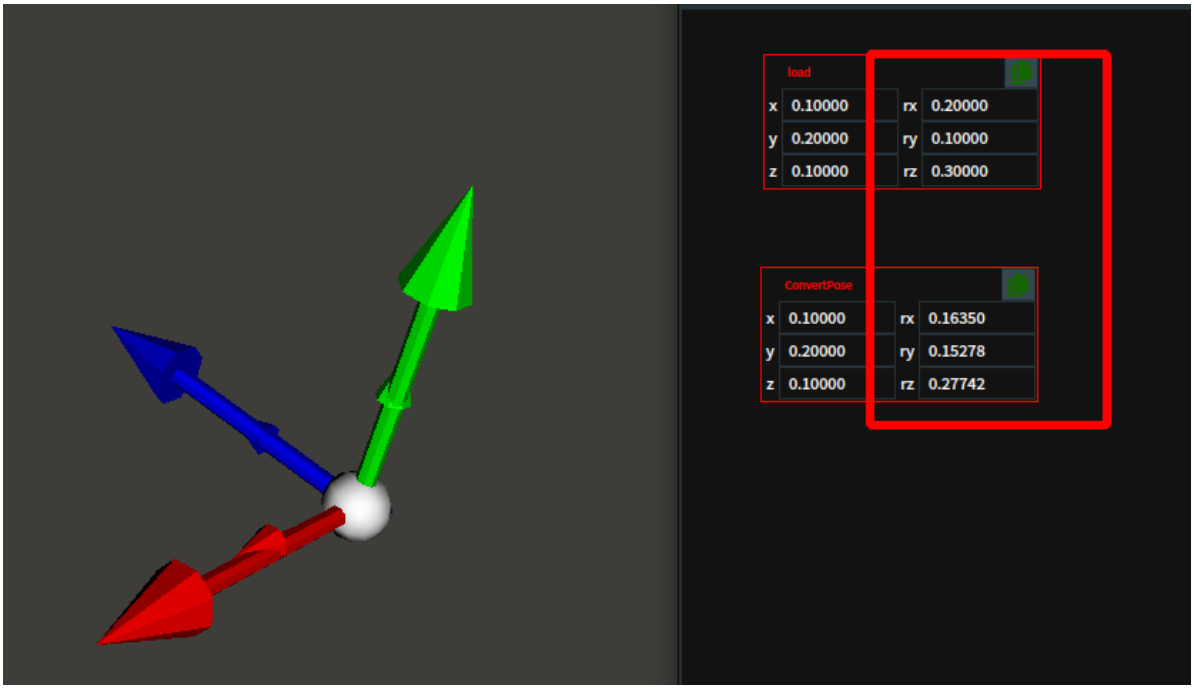


步骤4：运行

1. 将 Load 算子和 Convert 的可视化结果分别与交互面板中输出工具——“坐标输出”控件进行绑定。
2. 点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，在 3D 视图中看到两个算子的对比结果，在交互面板中控件显示已经完成了欧拉角切换。



EulerToPose

将 ConvertPose 算子中 **类型** 设置为 EulerToPose，用于将欧拉角转换成 pose。

算子参数

- **第一旋转轴/axis_1**：调整旋转顺序，轴 1 。默认值：X 。
- **第二旋转轴/axis_2**：调整旋转顺序，轴 2 。默认值：Y 。
- **第三旋转轴/axis_3**：调整旋转顺序，轴 3 。默认值：Z 。

  说明：三个轴的顺序表示旋转顺序，如上面的顺序表示为X - Y - Z。
- **坐标/pose**：设置 pose 在 3D 视图中的可视化属性。
  - 打开 pose 可视化。
  - 关闭 pose 可视化。
  - 设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

- **euler**：
  - 数据类型：Pose
  - 输入内容：欧拉角

输出：

- **pose**：
  - 数据类型：Pose
  - 输出内容： pose 数据

功能演示

本节将使用 ConverPose 算子中 EulerToPose 将欧拉角转换为位姿。这与 PoseToEuler 中展现的位姿转换为欧拉角方法相同，请参照该章节的功能演示。

PoseToRotateVector

将 ConvertPose 算子中 **类型** 设置为 PoseToRotateVector，用于将位姿转成旋转矢量。主要用于转换成优傲机器人所接受的表达方式。

算子参数

- **旋转矢量/rotate_vector**：设置旋转矢量在3D视图中的可视化属性。
  -  打开旋转矢量可视化。
  -  关闭旋转矢量可视化。
  -  设置旋转矢量的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

- **pose**：
  - 数据类型：Pose
  - 输入内容：pose 数据

输出：

- **rotate_vector**：
  - 数据类型：Pose
  - 输出内容：旋转矢量数据

功能演示

本节将使用 ConverPose 算子中 PoseToRotateVector 中将位姿转换为旋转矢量。这与 PoseToEuler 中展现的位姿转换为欧拉角方法相同，请参照该章节的功能演示。

RotateVectorToPose

将 ConvertPose 算子中 **类型** 设置为 RotateVectorToPose ，用于将旋转矢量转成 pose 。

算子参数

- **坐标/pose**：设置pose在3D视图中的可视化属性。
  -  打开 pose 可视化。
  -  关闭 pose 可视化。
  -  设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

- `rotate_vector`：
  - 数据类型：Pose
  - 输入内容：旋转矢量数据

输出：

- `pose`：
  - 数据类型：Pose
  - 输出内容：pose 数据

功能演示

本节将使用 ConvertPose 算子中 RotateVectorToPose 中将旋转矢量转换为位姿 。这与 PoseToEuler 中展现的位姿转换为欧拉角方法相同，请参照该章节的功能演示。

PoseToMatrix

将 ConvertPose 算子中 `类型` 设置为 PoseToMatrix ，用于将 pose 转换成变换矩阵。

算子参数

- `m_0/1/2/4/5/6/8/9/10`：转换后旋转矩阵的值。
- `m_3/7/11`：转换后平移矩阵的值。
- `m_12/13/14`：转换后缩放斜切的值，通常为 0 。
- `m_15`：如果要平移坐标，要将坐标维度增加 1 ，变成齐次坐标。默认值 1 。

数据信号输入输出

输入：

- `pose`：
  - 数据类型：Pose
  - 输入内容：pose 的数据 X 、Y 、Z 、yaw 、pitch 、roll

功能演示

将 ConvertPose 算子中 `类型` 设置为 PoseToMatrix ，将加载的 pose 转换为变换矩阵。

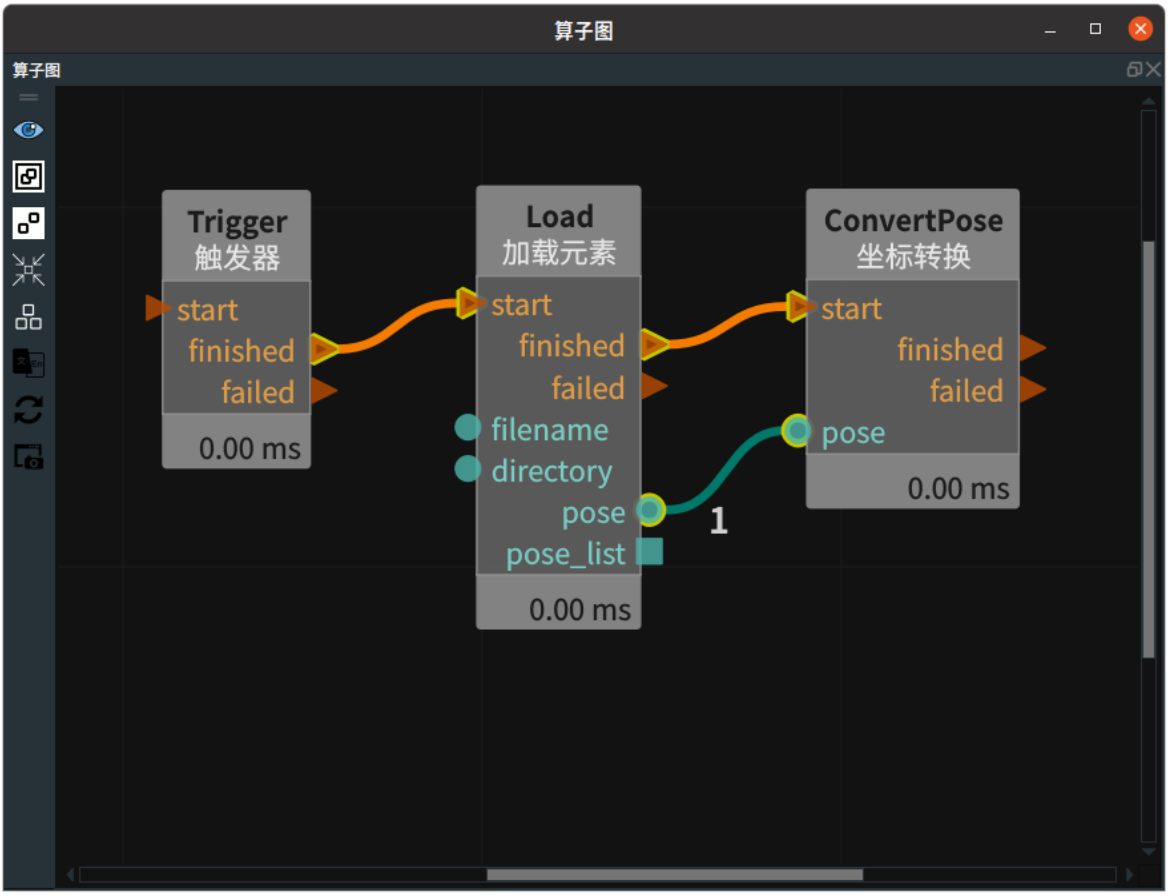
步骤1：算子准备

添加 Trigger、Load、ConvertPose 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → pose
   - 文件 → ••• → 选择 pose 文件名(*example_data/pose/tcp1.txt*)
2. 设置 ConvertPose 算子参数：
   - 类型 → PoseToMatrix

步骤3：连接算子

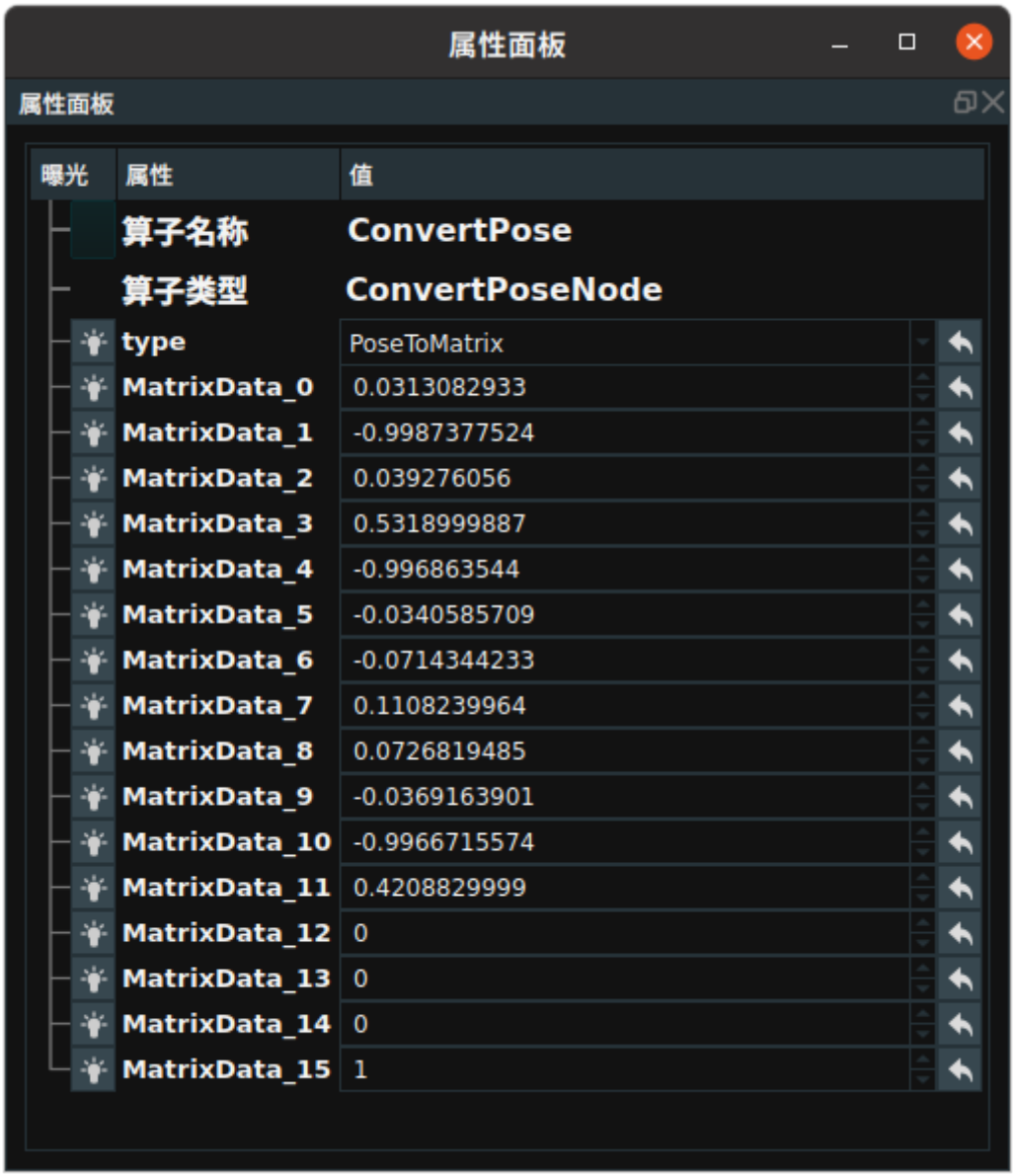


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，在属性面板中显示矩阵的值。



MatrixToPose

将 ConvertPose 算子中 **类型** 设置为 MatrixToPose ，用于将变换矩阵转换为 pose 。

算子参数

- **m_0/1/2/4/5/6/8/9/10**：输入4x4 矩阵中旋转矩阵的值。
- **m__3/7/11**：输入平移矩阵的值。
- **m_12/13/14**：缩放斜切的值。默认值：0 。
- **m_15**：如果要平移坐标，要将坐标维度增加1，变成齐次坐标。默认值：1 。
- **坐标/pose**：设置 pose 在 3D 视图中的可视化属性。
  - 打开 pose 可视化。
  - 关闭 pose 可视化。
  - 设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输出：

- `pose`：
  - 数据类型：Pose
  - 输出内容：位姿数据

功能演示

将 ConvertPose 算子中 `类型` 设置为 MatrixToPose ，将变换矩阵转换为位姿。

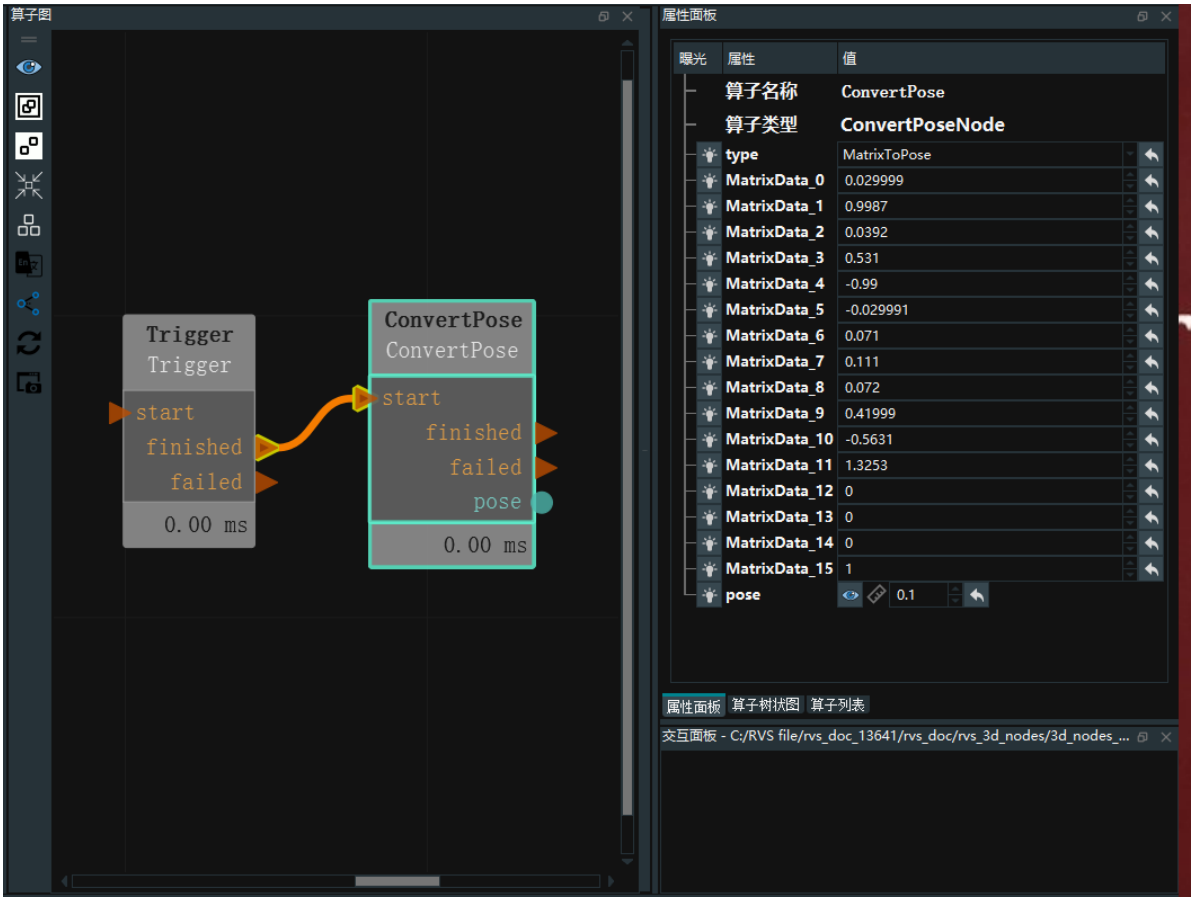
步骤1：算子准备

添加 Trigger 、ConvertPose 算子至算子图。

步骤2：设置算子参数

1. 设置 ConvertPose 算子参数：
   - 类型 → MatrixToPose
   - m_? → 参照下图属性面板中的值设定

步骤3：连接算子

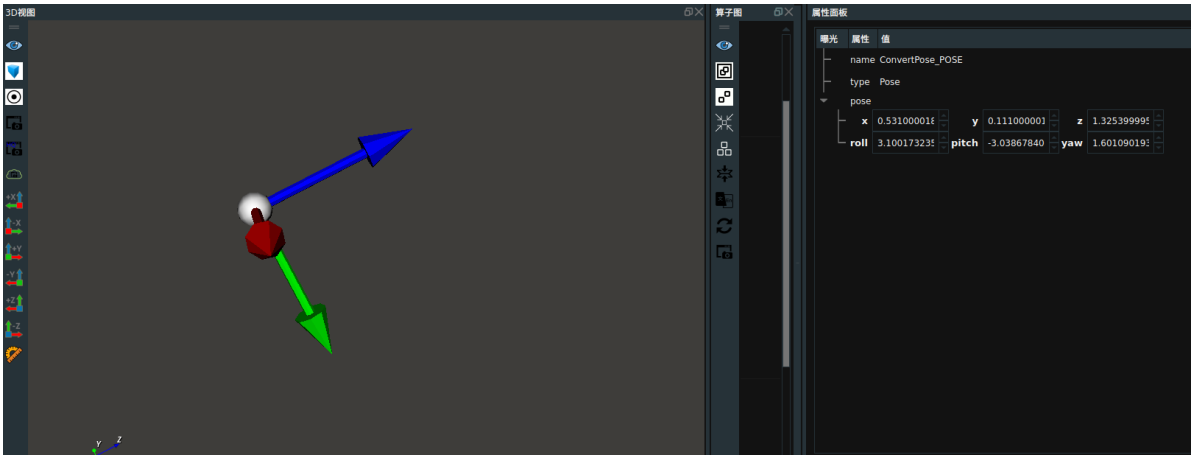


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，3D 视图中显示变换矩阵转换后的 pose 和点击 3D 视图中的 pose ，属性面板中显示其对应的 pose 值。


3D视图
算子图
属性面板
曝光
属性
值
name ConvertPose_POSE
type Pose
pose
x 0.531000016
y 0.111000001
z 1.325399995
roll 3.100173235
pitch -3.03867840
yaw 1.601090195

MinimumBoundingBox 最小包围立方体

MinimumBoundingBox 算子用于根据目标上表面点云来获取点云的最小立方体包围框，常用于箱包等物体的姿态定位。

type	功能
SimpleMVBB	获取点云的最小立方体包围框，与 ApproxMVBB 功能类似。推荐使用 SimpleMVBB，运算速度更快。
ApproxMVBB	获取点云的最小立方体包围框。本算子与 SimpleMVBB 算子功能一致，不再进行介绍。

算子参数

- **包围立方体/box**：设置包围框在 3D 视图中的可视化属性。
  -  打开包围框可视化。
  -  关闭包围框可视化 。
  -  设置 3D 视图中包围框的颜色。取值范围：[-2,360] 。默认值：-1 。
  -  设置包围框的透明度。取值范围：[0,1] 。默认值：0.5 。
- **包围立方体中心坐标/box_pose**：设置包围框中心点 pose 在 3D 视图中的可视化属性。
  -  打开包围框中心点 pose 可视化。
  -  关闭包围框中心点 pose 可视化。
  -  设置包围框中心点 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。
- **包围立方体列表/box_list**：设置包围框列表在 3D 视图中的可视化属性。 值描述与 **包围立方体** 一致。
- **包围立方体列表中心坐标/box_pose_list**：设置包围框中心点 pose 列表在 3D 视图中的可视化属性。值描述与 **包围立方体中心坐标** 一致。

数据信号输入输出

输入：

说明：根据需求选择 cloud 或者 cloud_list 其中一种数据信号输入即可。

- **cloud**：
  - 数据类型：PointCloud
  - 输入内容：点云数据
- **cloud_list**：
  - 数据类型：PointCloudLIst
  - 输入内容：点云数据列表
- **ref_pose**：
  - 数据类型：Pose
  - 输入内容：参照 pose 数据

说明：当连接 ref_pose ，结果 Cube 则会参照算子左侧的 ref_pose 端口输入的 pose 姿态进行 Height - Width - Depth 同 X Y Z 的匹配。如果该端口不连接数据，则 Height - Width - Depth 同 X Y Z 的匹配是随机的。一般我们选择连接该参数，并且给 ref_pose 端口输入 Pose(0,0,0,0,0,0) 即可。

输出：

- `box`：
  - 数据类型：Cube
  - 输出内容：最小立方体包围框
- `box_pose`：
  - 数据类型：Pose
  - 输出内容：最小立方体包围框 pose 数据
- `box_list`：
  - 数据类型：Cubelist
  - 输出内容：最小立方体包围框列表
- `box_pose_list`：
  - 数据类型：Poselist
  - 输出内容：最小立方体包围框 pose 列表数据

功能演示

使用 MinimumBoundingBox 获取加载点云的最小立方体包围框，并根据生成的原点 pose 进行姿态匹配。

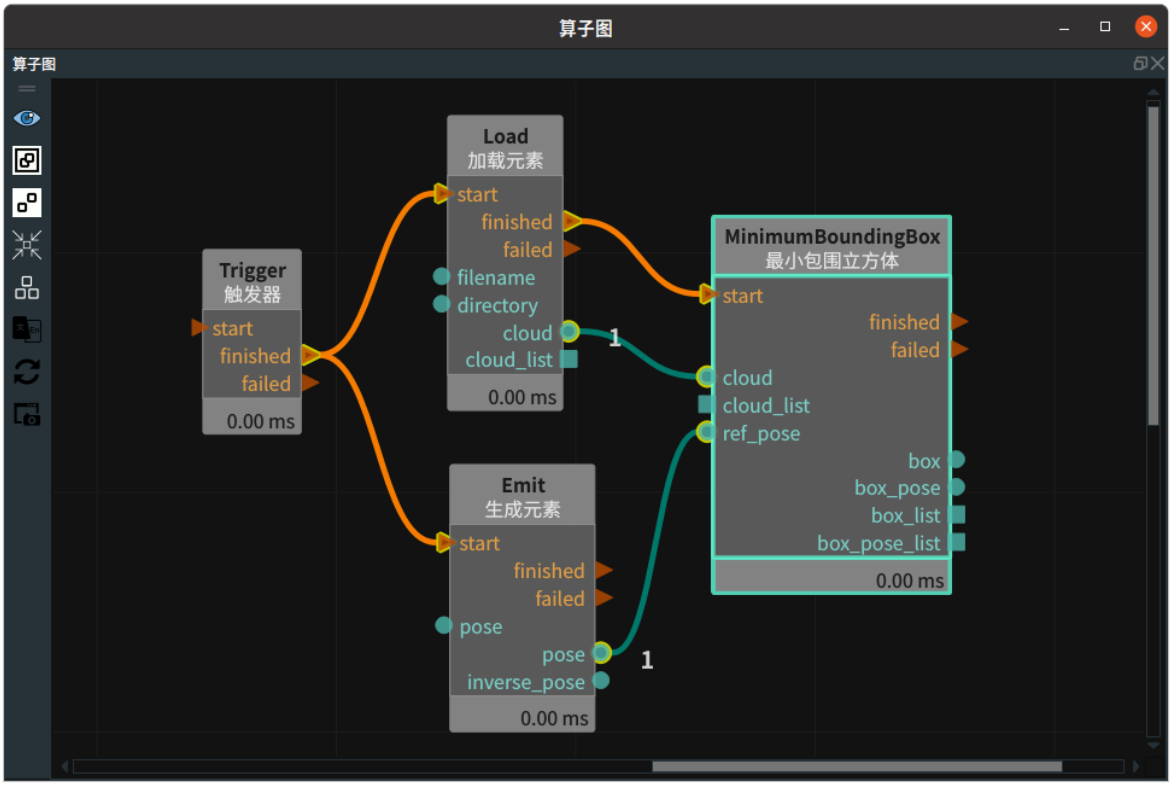
步骤1：算子准备

添加 Trigger 、Load 、Emit 、MinimumBoundingBox 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → PointCloud
   - 文件 → ••• → 选择点云文件名(*example_data/pointcloud/model.pcd*)
   - 点云 →  可视
2. 设置 Emit 算子参数：
   - 类型 → Pose
   - 坐标 → 0 0 0 0 0 0
   - 坐标 →  可视
3. 设置 MinimumBoundingBox 算子参数：
   - 包围立方体 →  可视
   - 包围立方体中心坐标 →  可视

步骤3：连接算子

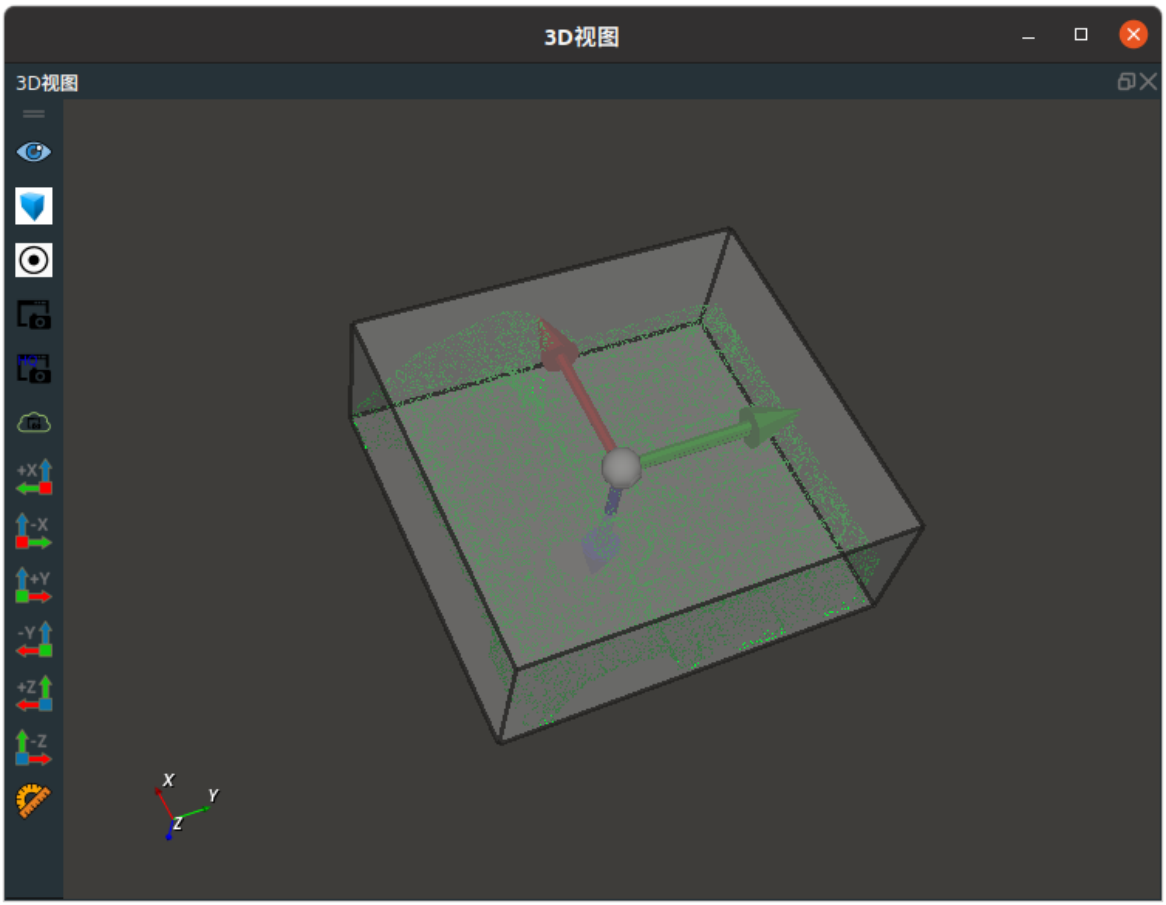


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

结果如下图所示，在 3D 视图中加载的点云以及 MinimumBoundingBox 的结果。最小包围框 Cube 和中心点 pose 。



Transform 元素空间变换

Transform 算子用于元素空间变换，适用于 Pose 、Cube 、PointCloud 。

type	功能
Pose	pose 空间变换。
Cube	立方体空间变化。
PointCloud	点云空间变换。

Pose

将 Transform 算子的 **类型** 属性选择 Pose ，用于 pose 空间变换。Pose 以算子左侧输入端口 base 坐标为基础坐标系，平移旋转 relative 的量。 relative 矩阵左乘 base 矩阵所得到的矩阵即为 transform 所得 pose 的矩阵。

算子参数

- **倒置/invert**：
  - **True**：输出空间变换后的逆 pose 。
  - **False**：输出空间变换后 pose 。
- **转换后坐标/transformed**：设置空间变换后的 pose 在 3D 视图中的可视化属性。
  -  打开 pose 可视化。
  -  关闭 pose 可视化。
  -  设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。
- **转换后坐标列表/transformed_list**：设置空间变换后的 pose 列表在 3D 视图中的可视化属性。值描述与 transformed 一致。

数据信号输入输出

输入：

说明：base / base_list 端口和 relative / relative_list 端口连接时，单个 base 可以和单个 relative 连接，也可以和 relative_list 连接。只需要连接两个端口即可。

- **base**：
  - 数据类型：Pose
  - 输入内容：需要变换的坐标
- **relative**：
  - 数据类型：Pose
  - 输入内容：相对坐标
- **base_list**：
  - 数据类型：PoseList
  - 输入内容：需要变换的坐标列表
- **relative_list**：
  - 数据类型：PoseList
  - 输入内容：相对坐标列表

输出：

- `transformed`：
  - 数据类型：Pose
  - 输出内容：变换后坐标数据
- `transformed_list`：
  - 数据类型：PoseList
  - 输出内容：变换后坐标列表数据

功能演示

使用 Transform 算子中将 Emit 生成的 pose 与 Emit_1 生成的 pose 二者进行空间转换。

步骤1：算子准备

添加 Trigger 、Emit 、Emit(2个) 、Transform 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
   - 类型 → Pose
   - 坐标 → 0.2 0 0 0 0 0
   - 坐标 → 可视
2. 设置Emit _1算子参数：
   - 类型 → Pose
   - 坐标 → 0 0 0 0 1.5708 0
   - 坐标 → 可视
3. 设置 Transform 算子参数：
   - 类型 → Pose
   - 转换后坐标 → 可视

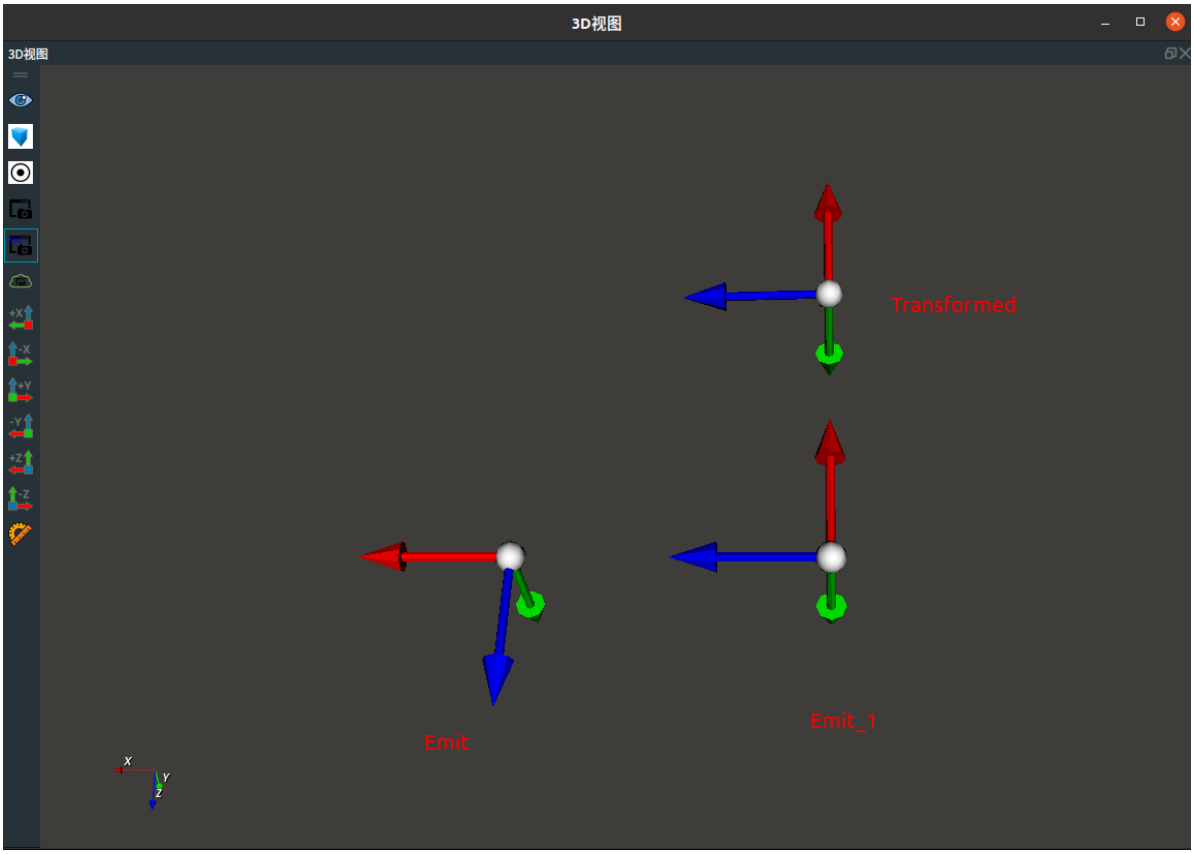
步骤3：连接算子

步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，在 3D 视图中显示 Emit 和 Emit_1 结果和 Transformed 结果，其为 Emit 生成的 pose 与 Emit_1 生成的 pose ，前者矩阵左乘后者矩阵所得到的矩阵即为 transform 所得 pose 的矩阵。



Cube

将 Transform 算子的 **类型** 属性选择 Cube ，用于立方体空间变换。其变换的原理是以 Cube 的中心点 pose 为基础进行变换。

算子参数

- **倒置/invert**：
  - **True**：输出空间变换后的逆 pose 。
  - **False**：输出空间变换后 pose 。
- **模式/mode**：设置 cube 位姿的模式。
  - **cube_base**：将 cube 位姿作为变换的坐标。
  - **cube_relative**：将 cube 位姿作为相对的坐标。
- **立方体/cube**：设置空间变换后的 cube 在3D视图中的可视化属性。
  -  打开立方体可视化 。
  -  关闭立方体可视化 。
  -  设置 3D 视图中立方体的颜色。取值范围：[-2,360] 。默认值：-2 。
  -  设置立方体的透明度。取值范围：[0,1] 。默认值：0.5
- **坐标/pose**：设置变换后立方体中心点 pose 在 3D 视图中的可视化属性。
  -  打开立方体中心点 pose 可视化。
  -  关闭立方体中心点 pose 可视化。
  -  设置立方体中心点 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。
- **立方体列表/cube_list**：设置 cube_list 的可视化属性，值描述与上述 **立方体** 一致。
- **坐标列表/pose_list**：设置空间变换后的 pose 列表可视化属性。值描述与 **坐标** 一致。

数据信号输入输出

说明：数据输入内容与属性面板中的 mode 模式相关。

输入：

- **cube**：
  - 数据类型：Cube
  - 输入内容：当模式为 cube_base 时为变换的 cube，当模式为 cube_relative 时为相对的 cube
- **cube_list**：
  - 数据类型：Cube
  - 输入内容：当模式为 cube_base 时为变换的 cubelist，当模式为 cube_relative 时为相对的 cubelist
- **pose**：
  - 数据类型：Pose
  - 输入内容：当模式为 cube_base 时为相对的 pose，当模式为 cube_relative 时为变换的 pose
- **pose_list**：
  - 数据类型：PoseList

- 输入内容：当模式为 cube_base 时为相对的 poselist，当模式为 cube_relative 时为变换的 poselist

输出：

- `cube`：
  - 数据类型：Cube
  - 输出内容：变换后立方体数据
- `cube_list`：
  - 数据类型：CubeList
  - 输出内容：变换后立方体列表数据
- `pose`：
  - 数据类型：Pose
  - 输出内容：变换后立方体中心点坐标
- `pose_list`：
  - 数据类型：PoseList
  - 输出内容：变换后立方体中心点坐标列表

功能演示

使用 Transform 算子中 Cube 将加载的立方体进行空间变换。

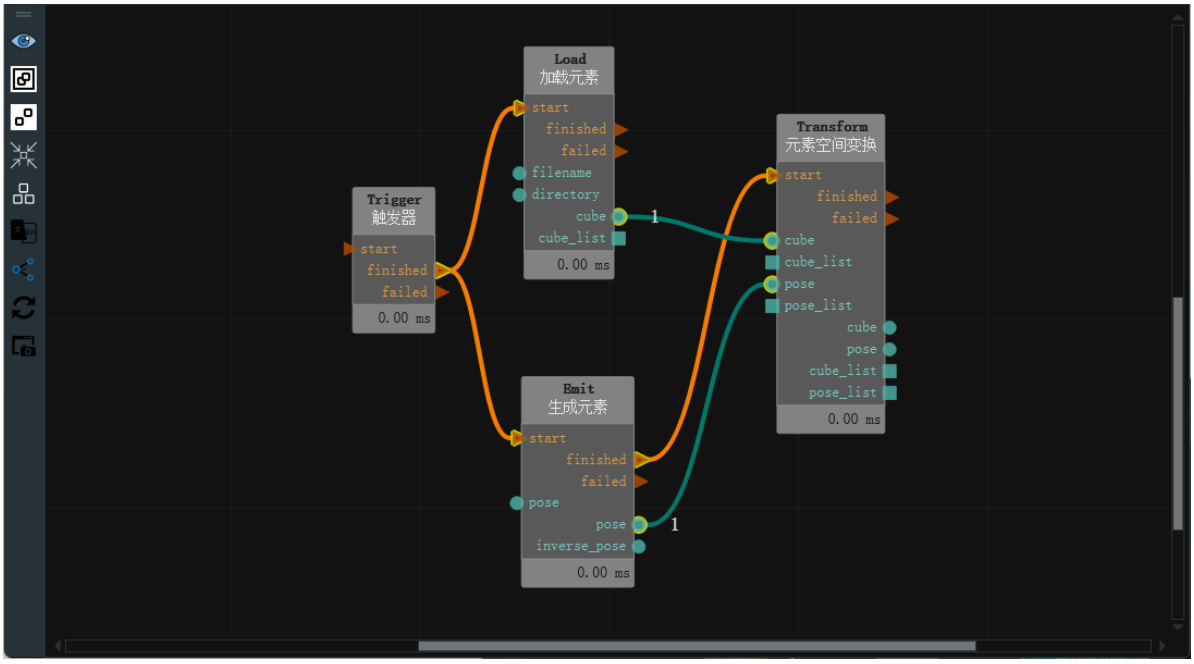
步骤1：算子准备

添加 Trigger 、Emit、Load 、Transform 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → Cube
   - 文件 → ••• → 选择立方体文件名 (*example_data/cube/cube.txt*)
   - 坐标 →  可视
2. 设置Emit 算子参数：
   - 类型 → Pose
   - 坐标 → 1 0 0 1.57 3.14 0
   - 坐标 →  可视 →  1
3. 设置 Transform 算子参数：
   - 类型 → Cube
   - 模式 → cube_base
   - 立方体 →  可视 →  120

步骤3：连接算子

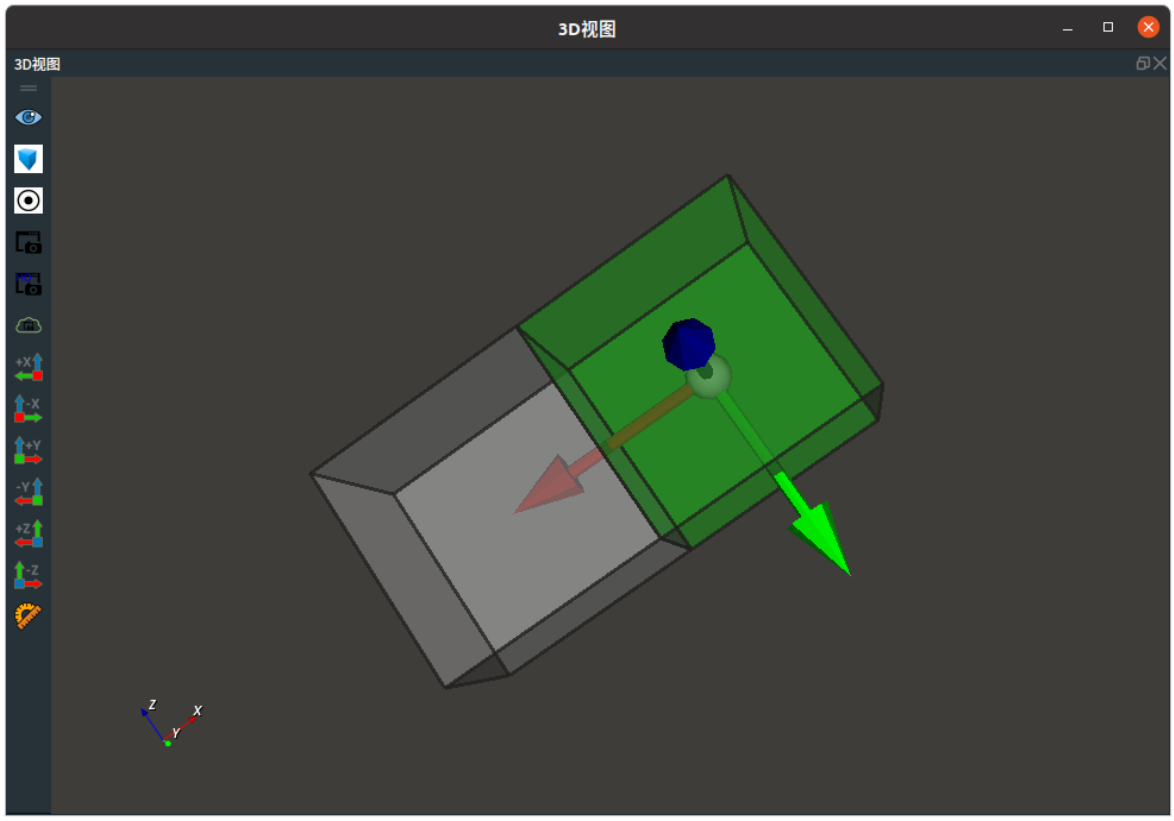


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，3D 视图中分别显示 cube 转换前后的对比图，灰色立方体为 Load 算子加载的 cube ，绿色立方体为 Transform 算子转换后的 cube 。pose 为 Emit 算子生成的 pose 。



PointCloud

将 Transform 算子的 **类型** 属性选择 PointCloud ，用于点云空间变换。其变换的原理是以点云的中心点 pose 为基础进行变换。

算子参数

- **点云/cloud**：设置空间变换后的点云在 3D 视图中的可视化属性。
  -  打开点云可视化。
  -  关闭点云可视化。
  -  设置 3D 视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  -  设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。
- **点云列表/cloud_list**：设置点云列表在 3D 视图中的可视化属性。值描述与 **点云** 一致。

数据信号输入输出

输入：

- **cloud**：
  - 数据类型：PointCloud
  - 输入内容：需要变换的点云数据
- **cloud_list**：
  - 数据类型：CloudList
  - 输入内容：需要变换的点云列表数据
- **pose**：
  - 数据类型：Pose
  - 输入内容：相对坐标数据
- **pose_list**：
  - 数据类型：PoseList
  - 输入内容：相对坐标列表数据

输出：

- **cloud**：
  - 数据类型：PointCloud
  - 输出内容：变换后点云数据
- **cloud_list**：
  - 数据类型：PointCloudList
  - 输出内容：变换后点云列表数据

功能演示

使用 Transform 算子中 PointCloud 将加载的点云进行空间变换。

步骤1：算子准备

添加 Trigger 、Load、Emit 、Transform 算子至算子图。

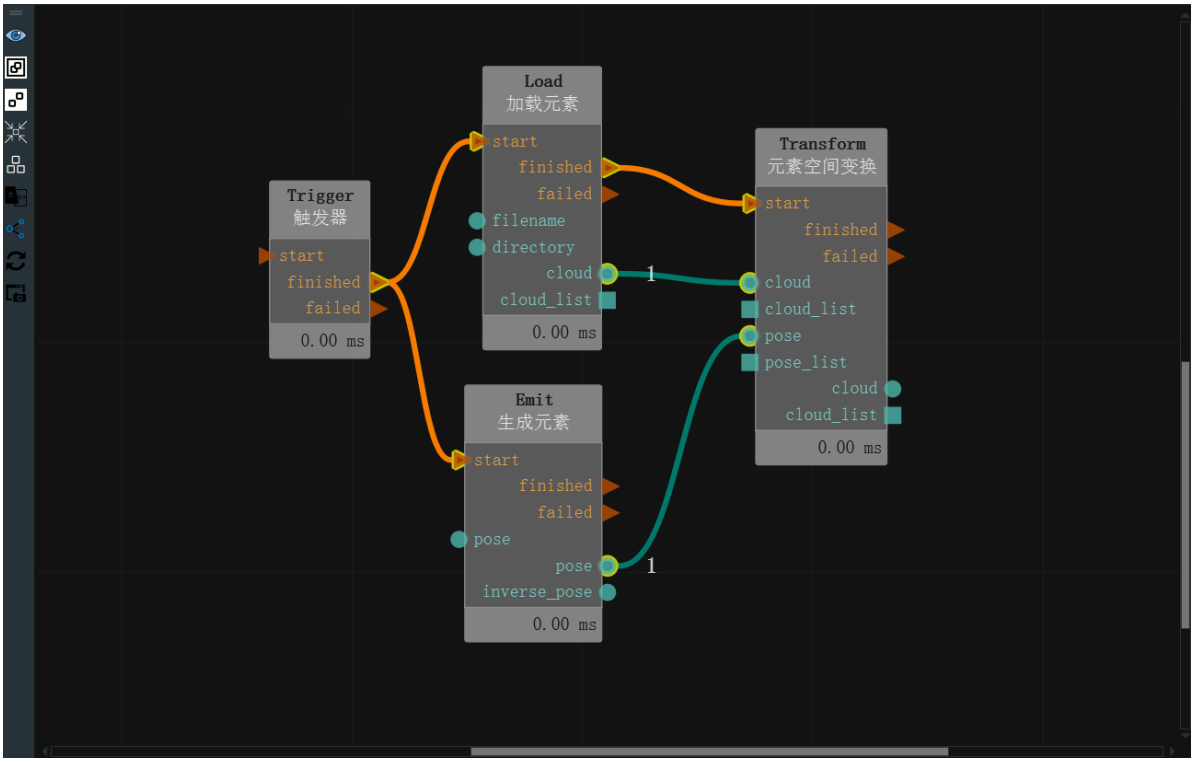
步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → pointcloud

- 文件 → ••• → 选择点云文件名(*example_data/pointcloud/wolf1.pcd*)
- 坐标 → 可视 → 280

2. 设置Emit 算子参数：
   - 类型 → Pose
   - 坐标 → 1 2 0 1.57 3.14 0
   - 坐标 → 可视 → 10
3. 设置 Transform 算子参数：
   - 类型 → PointCloud
   - 点云 → 可视 → 60

步骤3：连接算子

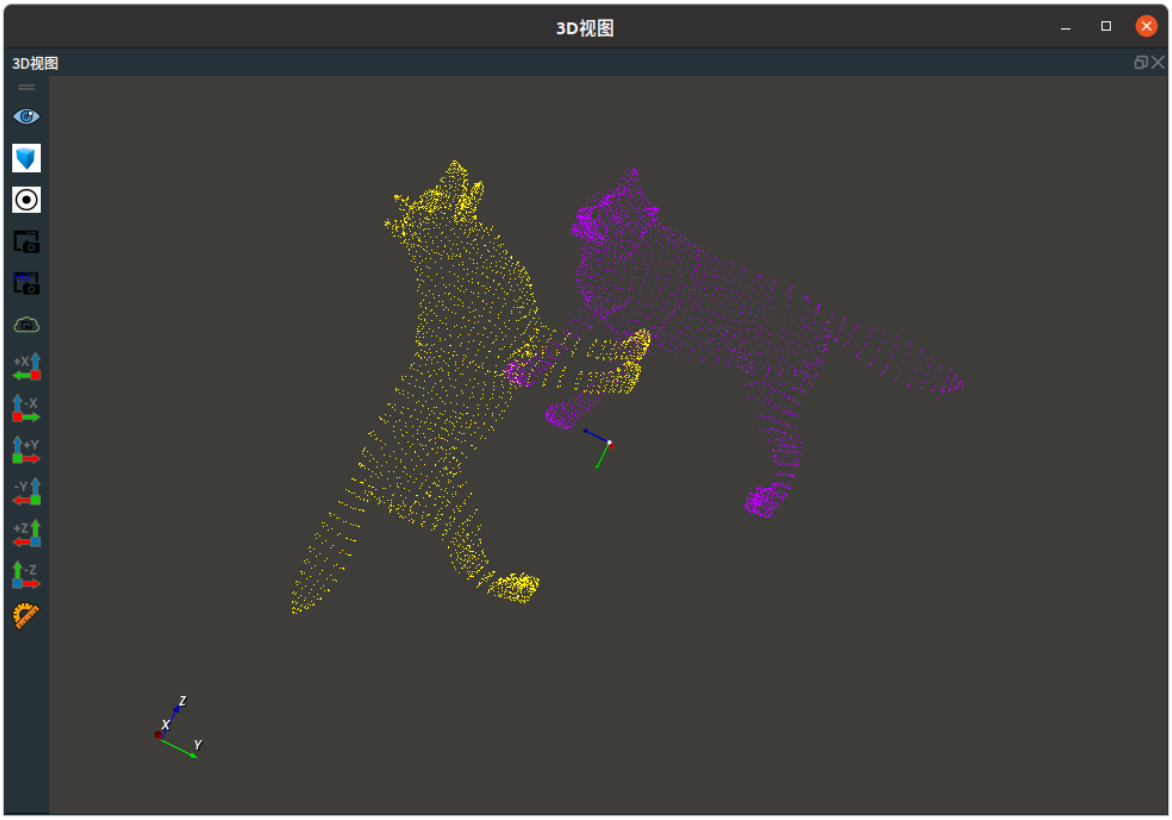


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，3D 视图中分别显示点云转换前后的对比图，紫色点云为 Load 算子加载的点云，黄色点云为 Transform 算子转换后的点云，pose 为 Emit 算子生成的 pose。


3D视图
3D视图
+X
-X
+Y
-Y
+Z
-Z
Z
X
Y

LineOperation 线段处理工具

LineOperation算子用于对 Line 以及 Pose 对象进行一些 3D 几何运算。

type	功能
PointToLineDistance	计算点(以 Pose 形式给出)到直线的距离，并给出垂足和垂线段。
LineCrossPoint	求两条直线交点或者两条异面直线距离最近处的中间点。
PoseStepByLine	过点(以 Pose 形式给出) 作直线的平行线，该平行线长度由算子参数给出注意，上述直线方向，由给定线段 Line 的 pose1 指向 pose2 。
LineProjectionAngle	求一条线段在 3 个空间基坐标面 (Oxy 、Oxz 、Oyz) 下的投影角度。
LineCrossAngle	

PointToLineDistance

将 LineOperation 算子的 **类型** 属性选择 PointToLineDistance，用于计算点 (以 Pose 形式给出) 到直线的距离，并给出垂足和垂线段。

注意：上述垂足 Pose ，仅有 xyz 数值是有效值，而三个旋转角全部同输入 Pose 保持一致。

算子参数

- **垂直线/vertical_line**：设置垂线段 Line 在 3D 视图中的可视化属性。
  -  打开线段可视化。
  -  关闭线段可视化。
  -  设置线段的颜色。取值范围：[-2,360] 。默认值：60 。
  -  设置线段的线宽。取值范围：[0,100] 。默认值：1 。
- **垂直点/vertical_point**：设置垂足 Pose 在 3D 视图中的可视化属性。
  -  打开 pose 可视化。
  -  关闭 pose 可视化。
  -  设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1。
- **距离/distance**：设置垂线段到垂足的距离曝光属性。 打开后可用于与交互面板中输出工具——“文本框”控件绑定。
  -  打开 distance 曝光。
  -  关闭 distance 曝光。

数据信号输入输出

输入：

- **line**：
  - 数据类型：Line
  - 输入内容：线段
- **point**：
  - 数据类型：Pose

- 输入内容：点（仅使用 Pose 的 xyz ）

输出：

- `vertical_line`：
  - 数据类型：Line
  - 输出内容：以输入点和垂足点为端点的垂线段
- `vertical_point`：
  - 数据类型：Pose
  - 输出内容：垂足
- `distance`：
  - 数据类型：String
  - 输出内容：点到直线的距离

功能演示

使用 LineOperation 算子中 PointToLineDistance 求点到直线的距离。

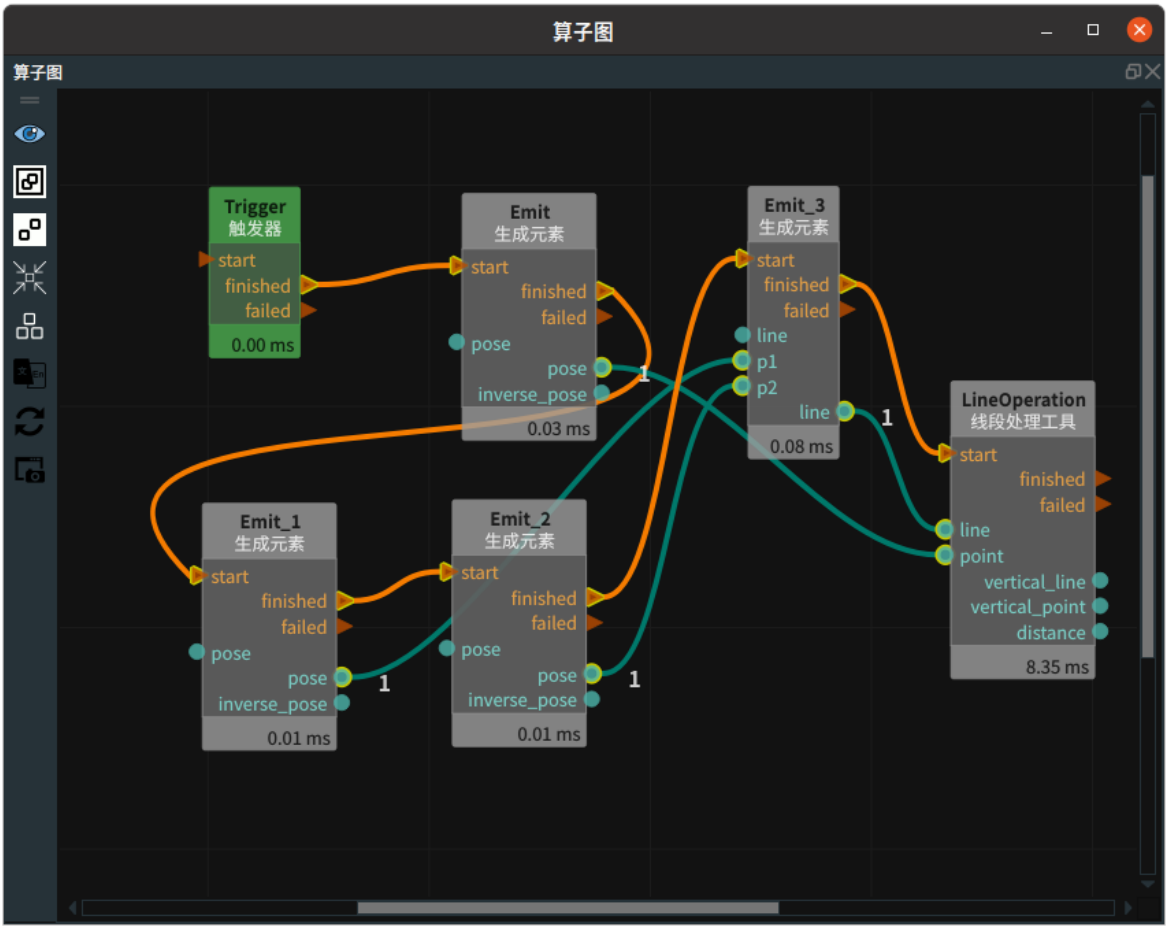
步骤1：算子准备

添加 Trigger 、Emit(4个) 、LineOperation 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
   - 类型 → Pose
   - 坐标 → 0.1 0 0 0 0 1
   - 坐标 → 可视
2. 设置 Emit_1 算子参数：
   - 类型 → Pose
   - 坐标 → -0.1 0.2 0 0 1 0
   - 坐标 → 可视
3. 设置 Emit_2 算子参数：
   - 类型 → Pose
   - 坐标 → 0.3 0.2 0 0 0 1
   - 坐标 → 可视
4. 设置 Emit_3 算子参数：
   - 类型 → Line
   - 线段 → 可视
5. 设置 LineOperation 算子参数：
   - 类型 → PointToLineDistance
   - 垂直线 → 可视
   - 垂直点→ 可视

步骤3：连接算子

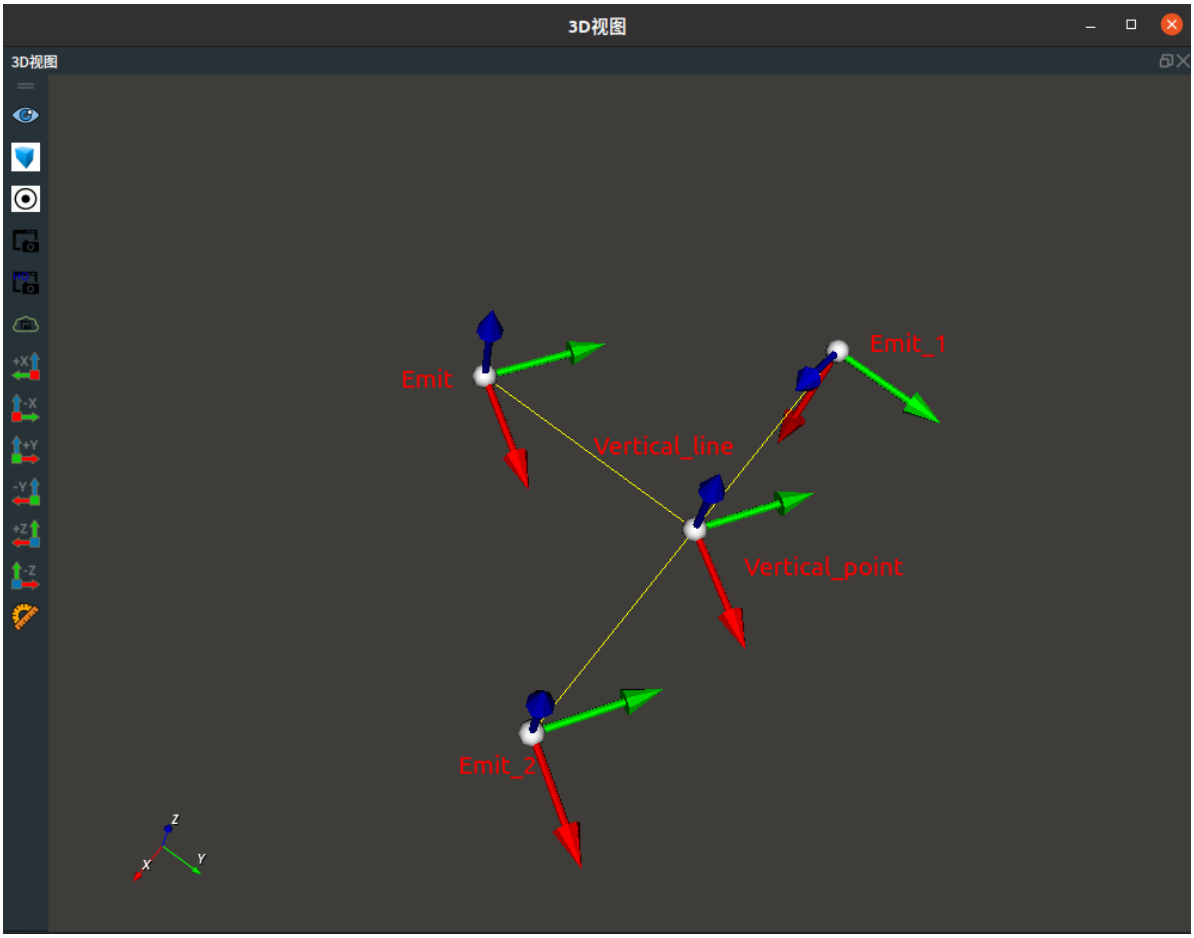


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

结果显示如下，分别显示 Emit / Emit_1 / Emit_2 生成的 pose ，Emit_3 生成的线段，LineOperation 的 Vertical_line 、vertical_point 。



LineCrossPoint

将 LineOperation 算子的 **类型** 属性选择 LineCrossPoint ，用于求两条直线交点或者两条异面直线距离最近处的中间点。

算子参数

- **最大距离/max_distance**：距离阈值。当两条直线之间的最短距离小于该阈值时，则算子触发failed信号。
- **中垂线中点/cross_point**：对 cross_point 输出端口的 Pose 进行显示控制，默认为关闭状态，参数介绍如下所示。
- **line1中垂线交点/line1_point**：对 line1_point 输出端口的 Pose 进行显示控制，默认为关闭状态，参数介绍如下所示。
- **line2中垂线交点/line2_point**：对 line2_point 输出端口的 Pose 进行显示控制，默认为关闭状态，参数介绍如下所示。
  - 打开 pose 可视化。
  - 关闭 pose 可视化。
  - 设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

- `line1`：
  - 数据类型：Line
  - 输入内容：线段1
- `line2`：
  - 数据类型：Line
  - 输入内容：线段2

输出：

- `cross_point`：
  - 数据类型：Pose
  - 输出内容：两条空间直线的“交点”。异面直线不相交时，则为两条直线相距最近点的中点。
- `line1_point`：
  - 数据类型：Pose
  - 输出内容：两条空间直线的“交点”。异面直线不相交时，则为直线line1上的相距最近点。
- `line2_point`：
  - 数据类型：Pose
  - 输出内容：两条空间直线的“交点”。异面直线不相交时，则为直线line2上的相距最近点。

注意，上述三个Pose输出仅有 xyz 数值是有效值，而三个旋转角全部是 0。

功能演示

使用 LineOperation 算子中 LineCrossPoint 求加载点云中两条直线角点或者两条异面直线距离最近处的中间点。

步骤1：算子准备

添加 Trigger、Emit 、Load 、FindElement （2个） 、LineOperation 、GeometryProbe 、（2个）、WaitAllTriggers 算子至算子图。

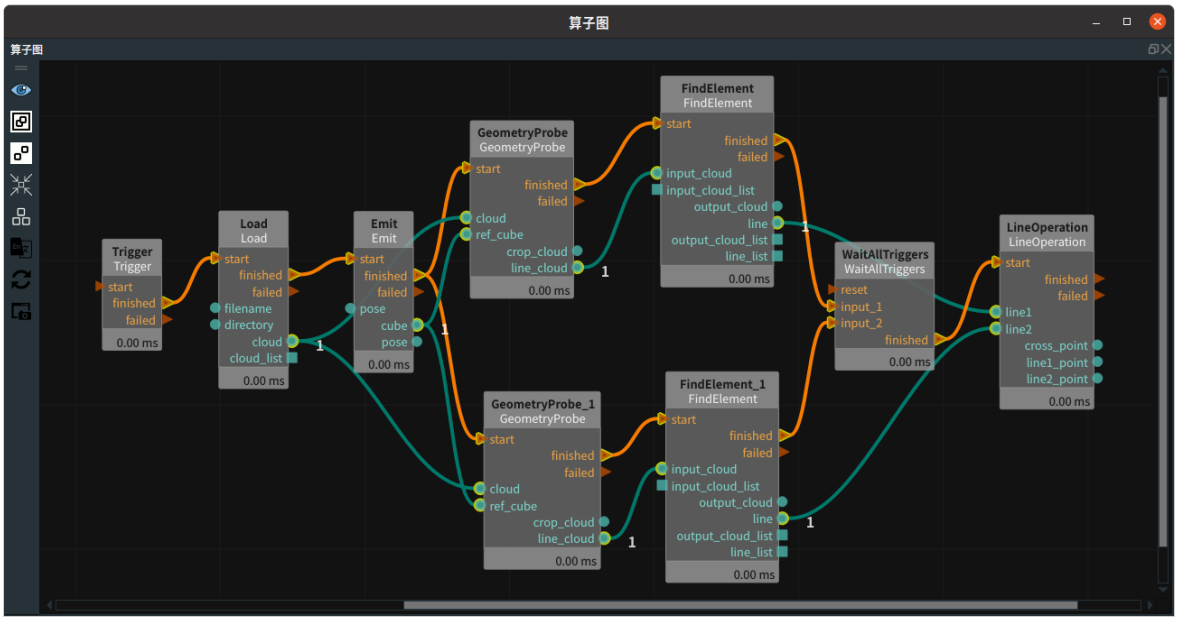
步骤2：设置算子参数

1. 设置 Emit算子参数：
   - 类型 → cube
   - 坐标 → 0.199179 0.26519 0.092586 2.61142 -2.82994 3.07943
   - 宽度 → 0.02
   - 高度 → 0.01
   - 深度 → 0.005
2. 设置 Load 算子参数：
   - 类型 → PointCloud
   - 文件 → ••• → 选择点云文件名(*example_data/pointcloud/cloud.pcd*)
3. 设置 LineOperation 算子参数：
   - 类型 → LineCrossPoint
   - 最大距离→ 0.003
   - 中垂线中点 → 可视 → 0.01
4. 设置 FindElement 算子参数：
   - 类型 → Line

- 最大迭代次数 → 1000
- 距离阈值 → 0.0001
- 线段长度 → 0.1
- 线段 → 可视

5. 设置 FindElement_1 算子参数：
   - 类型 → Line
   - 最大迭代次数 → 1000
   - 距离阈值 → 0.0001
   - 线段长度 → 0.1
   - 线段 → 可视
6. 设置 GeometryProbe 算子参数
   - 类型 → Line
   - 探测方向 → y-
   - 圆锥体顶角度 → 45
   - 有效点百分比 → 10
   - 步进值 → 5
   - 线段点云 → 可视
7. 设置 GeometryProbe_1 算子参数
   - 类型 → Line
   - 探测方向 → x-
   - 圆锥体顶角度 → 45
   - 有效点百分比→ 10
   - 步进值 → 5
   - 线段点云 → 可视
8. 设置 WaitAllTriggers 算子参数：输入数量 → 2

步骤3：连接算子

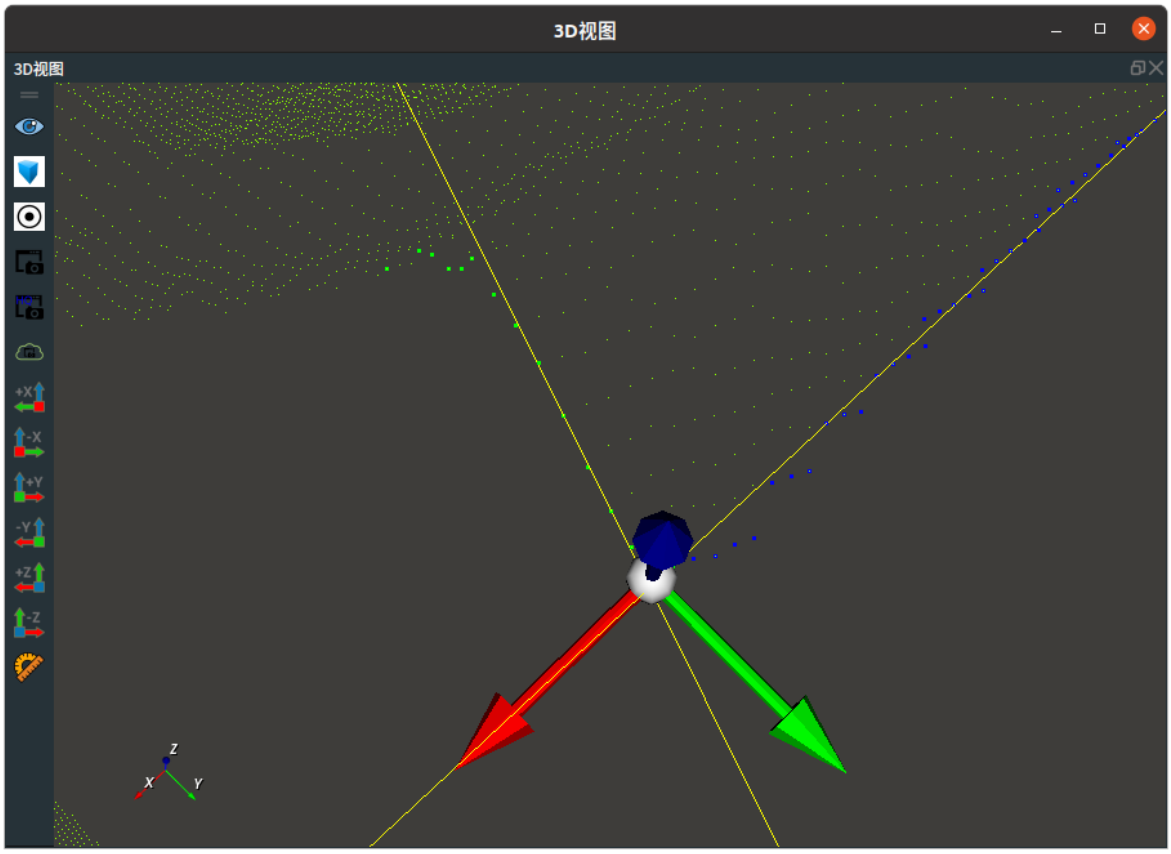


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

结果如下图所示，两条直线为 FindElement 中属性 line 的可视化结果，蓝色点云和绿色点云为 GeometryProbe 算子的属性 line_cloud 可视化结果。pose 为LineOperation 算子的可视化结果。



PoseStepByLine

将 LineOperation 算子的 **类型** 属性选择 PoseStepByLine ，过点(以 Pose 形式给出)作直线的平行线，该平行线长度由算子参数给出。

注意：上述直线方向，由给定线段 Line 的 pose1 指向 pose2 。

算子参数

- **步长距离/step_distance**：延伸长度。
- **方向与参考线段一致/direction_same_with_line**：默认不勾选，此时输出的 pose 的3个旋转角同输入 pose 保持一致。
  - True：输出的 pose 方向轴的 x 轴同线段方向保持一致。
  - False：输出的 pose 的3个旋转角同输入 pose 保持一致。
- **坐标/pose**：对 pose 输出端口的 Pose 进行显示控制。
  - 打开 pose 可视化。
  - 关闭 pose 可视化。
  - 设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

- **pose**：
  - 数据类型：Pose

    - 输入内容：pose 坐标
- `ref_line`：
    - 数据类型：Line
    - 输入内容：参考线段

输出：

- `pose`：
    - 数据类型：Pose
    - 输出内容：延伸点

功能演示

使用 LineOperation 算子中 PoseStepByLine 作直线的平行线。

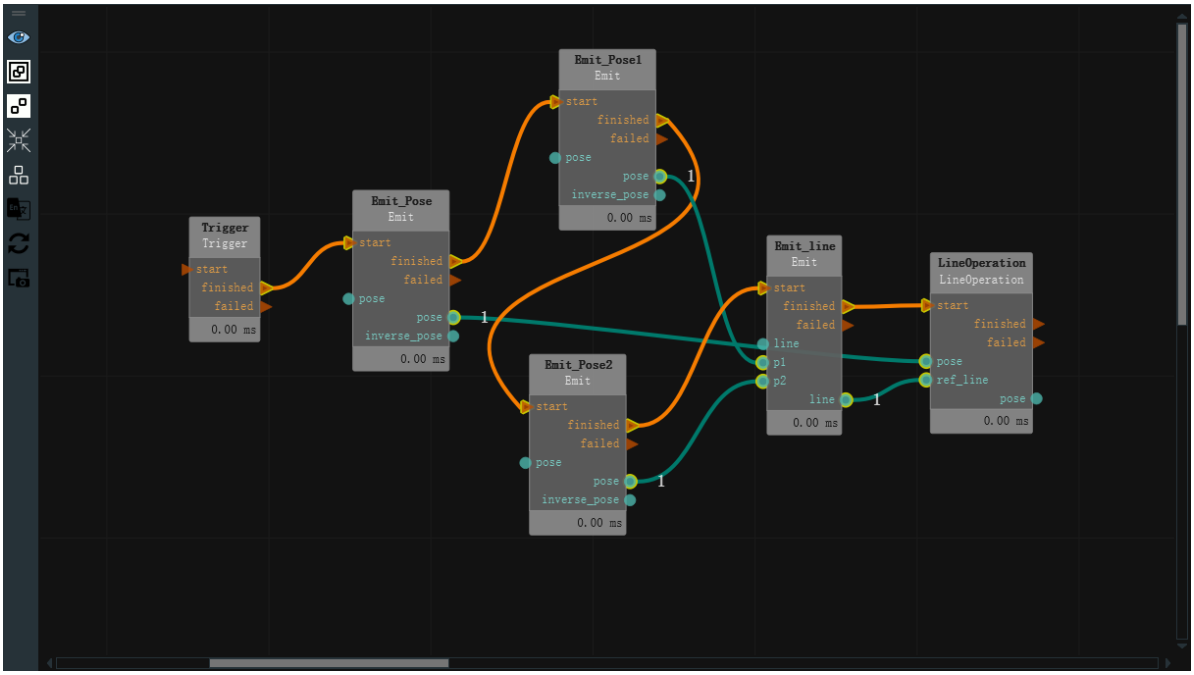
步骤1：算子准备

添加 Trigger 、Emit (4个) 、LineOperation 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
    - 算子名称 → Emit_Pose
    - 类型 → Pose
    - 坐标 → 0.1 0 0 1.5708 0 0
    - 坐标 →  可视 →  0.05
2. 设置 Emit_1 算子参数：
    - 算子名称 → Emit_Pose1
    - 类型 → Pose
    - 坐标 → 0 0 0 0 0 0
    - 坐标 →  可视 →  0.05
3. 设置 Emit_2 算子参数：
    - 算子名称 → Emit_Pose2
    - 类型 → Pose
    - 坐标 → 0 0 0.1 0 0 1
    - 坐标 →  可视 →  0.05
4. 设置 Emit_3 算子参数：
    - 算子名称 → Emit_line
    - 类型 → Line
    - 线段 →  可视
5. 设置 LineOperation 算子参数：
    - 类型 → PoseStepByLine
    - 步长距离 → 0.1
    - 坐标 →  可视 →  0.05

步骤3：连接算子

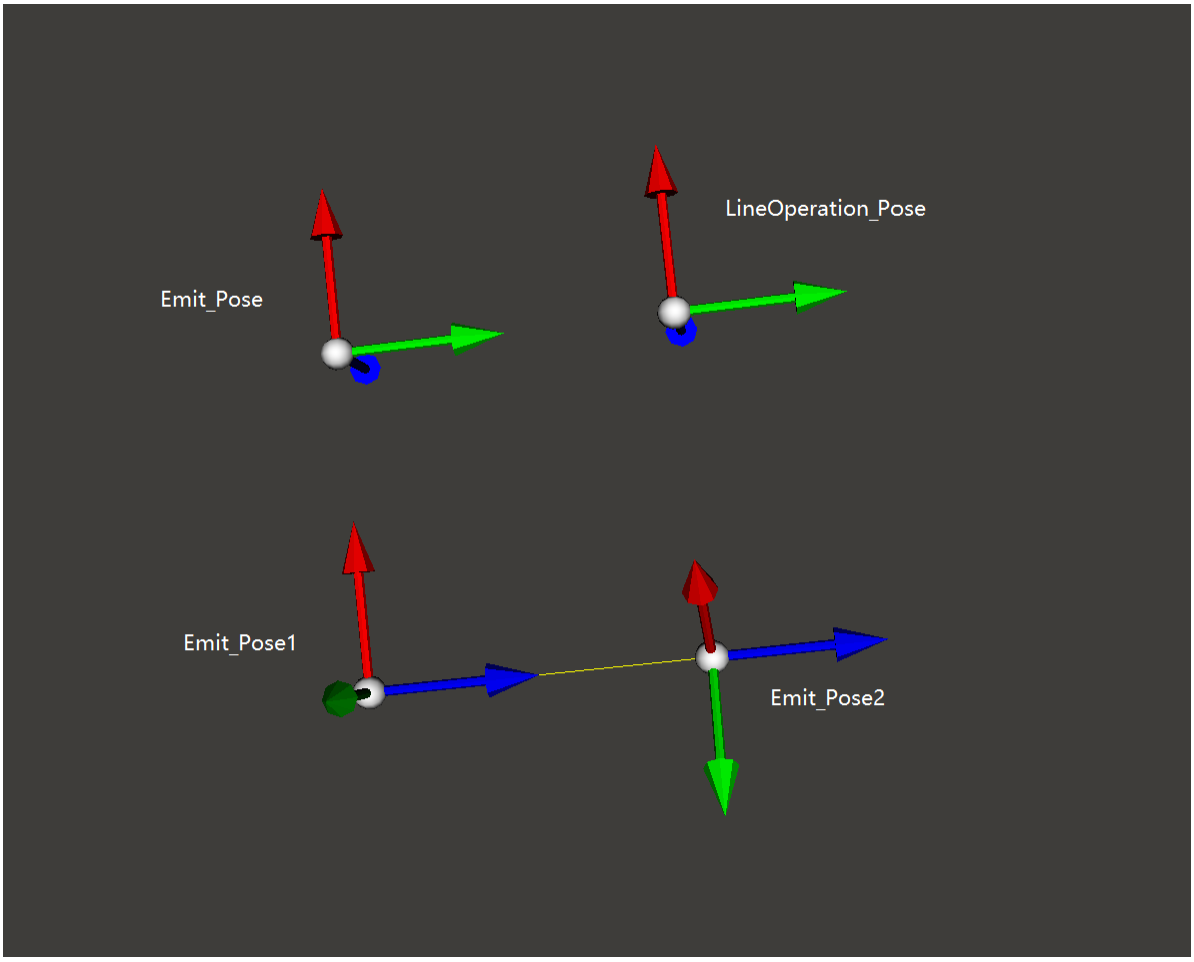


步骤4：运行

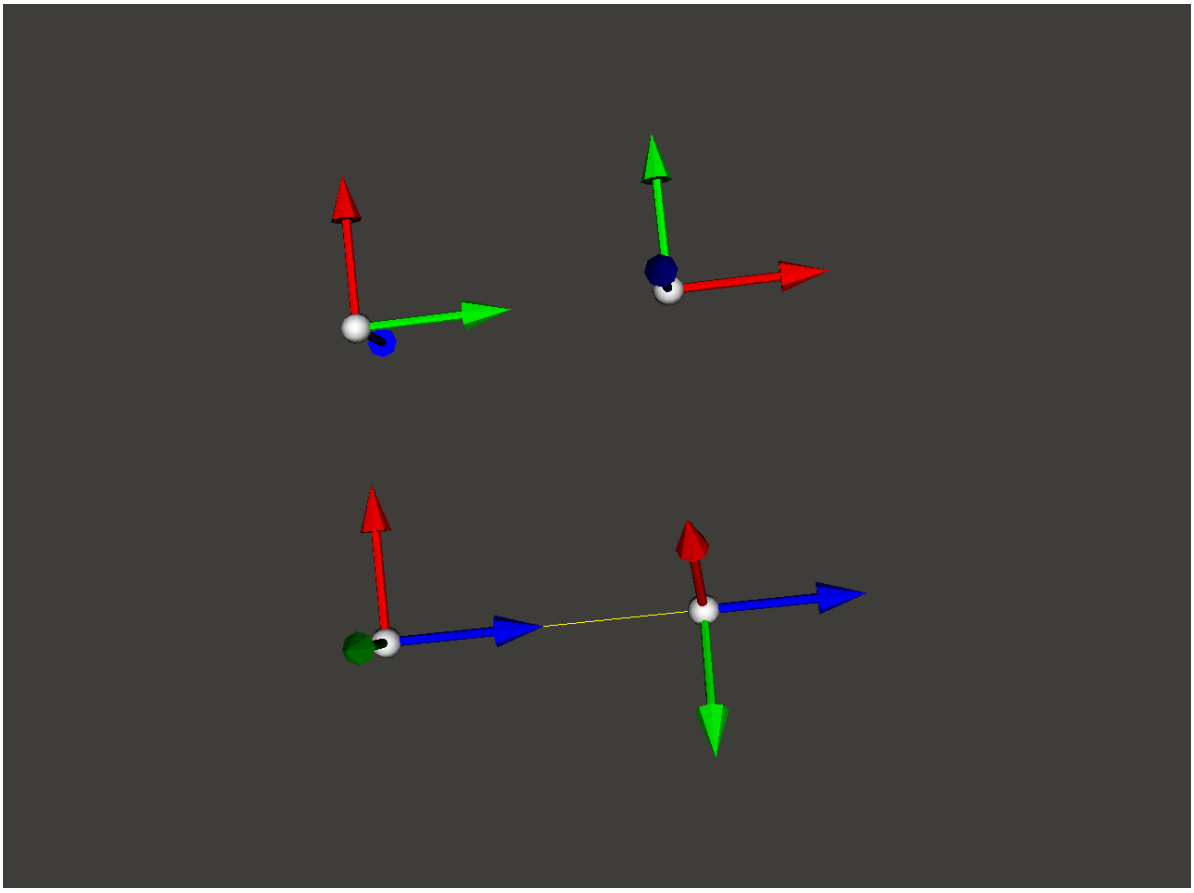
点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

1. Emit 算子的 Pose 或 Line 的可视化，LineOperation 算子的 pose 可视化，结果显示如下。



2. 勾选 LineOperation 算子的 direction_same_with_line 参数，重新触发 XML 中的 trigger 算子，运行结果显示如下。（可见下图右上角的结果 pose 的红色 x 轴已经沿着线段方向 ）



LineProjectionAngle

将 LineOperation 算子的 **类型** 属性选择 LineProjectAngle ，求一条线段在3个空间基坐标面 (Oxy 、Oxz 、Oyz) 下的投影角度。

算子参数

- **与xy|xz|yz平面夹角/angle_xy|xz|yz**：设置 angle_xy|xz|yz 投影角度在 3D 视图中的可视化属性。
  - 打开投影角度可视化。
  - 关闭投影角度可视化。
  - 设置 3D 视图中投影角度的颜色。取值范围：[-2,360] 。默认值：60 。
  - 设置投影角度线的线宽。取值范围：[0,100] 。默认值：1 。
  - 标注角显示。

数据信号输入输出

输入：

- **line**：
  - 数据类型：Line
  - 输入内容：线段

输出：

- **angle_xy**：
  - 数据类型：Angle
  - 输出内容：原线段在 Oxy 平面上的投影角度

- `angle_xz`：
  - 数据类型：Angle
  - 输出内容：原线段在 Oxz 平面上的投影角度
- `angle_yz`：
  - 数据类型：Angle
  - 输出内容：原线段在 Oyz 平面上的投影角度

    说明：RVS 中的 Angle 数据在数学形式上是三个点，第一和第三个点表征两条边，中间点是角点；Line 在数学形式上是两个点，第一个点为起点，第二个点为终点；该算子输出的三个 Angle 的角点全部是原始输入线段的起点。

功能演示

使用 LineOperation 算子中 LineProjectionAngle 求一条线段在3个空间基坐标面（ Oxy 、Oxz 、Oyz ）下的投影角度。

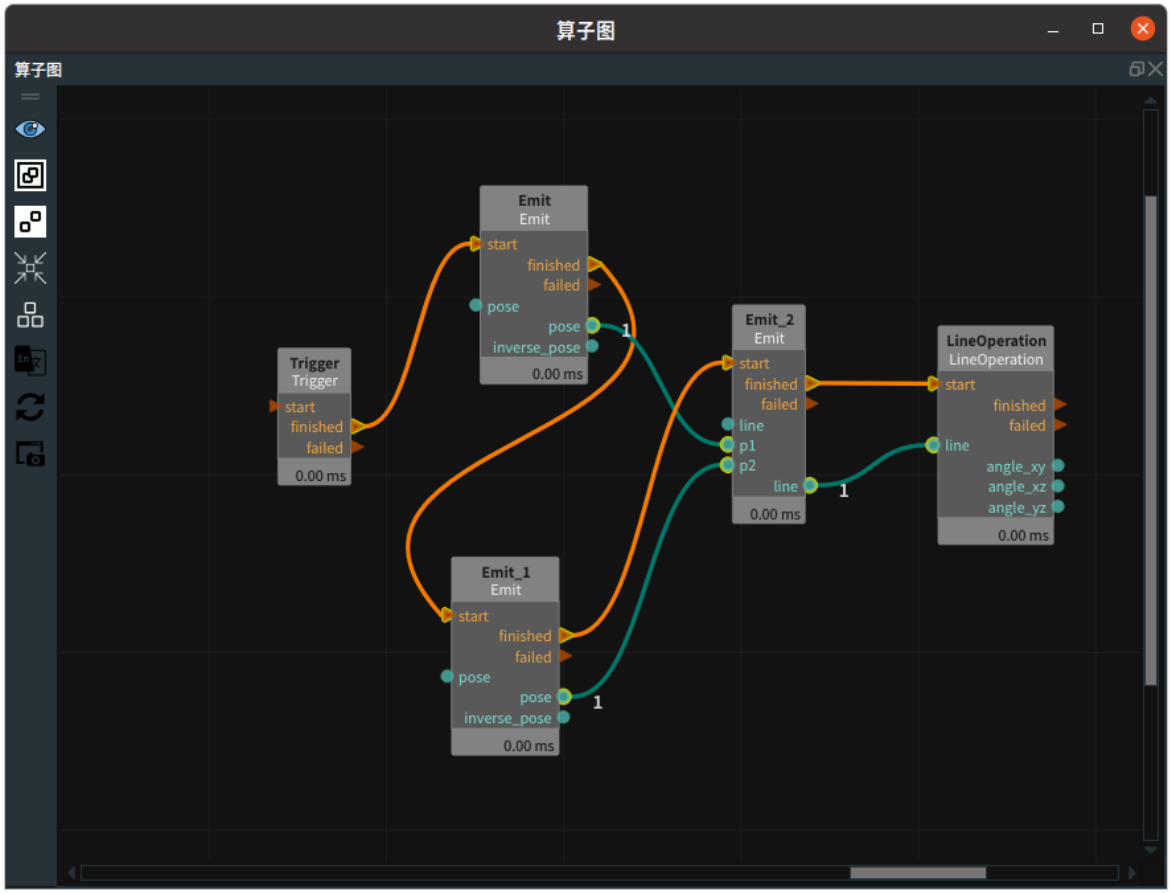
步骤1：算子准备

添加 Trigger 、Emit（3个） 、LineOperation 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
   - 类型 → Pose
   - 坐标 → 0 0 0 0 0 0
2. 设置 Emit_1 算子参数：
   - 类型 → Pose
   - 坐标 → 0.03 0.1 0.05 0 1 0
3. 设置 Emit_2 算子参数：
   - 类型 → Line
   - 线段 →  可视
4. 设置 LineOperation 算子参数：
   - 类型 → LineProjectionAngle
   - 与XY平面夹角 →  可视
   - 与XZ平面夹角 →  可视
   - 与YZ平面夹角 →  可视

步骤3：连接算子



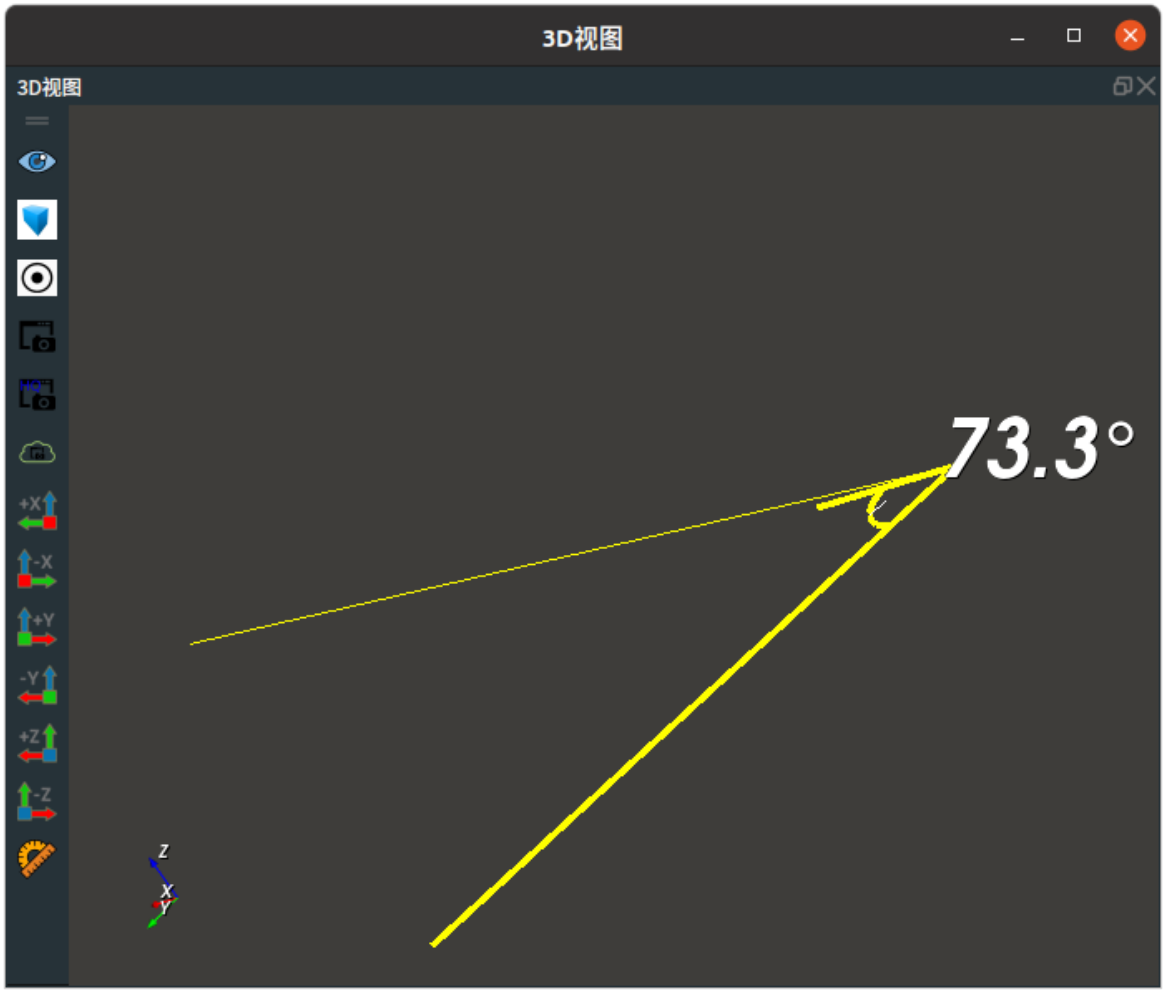
步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

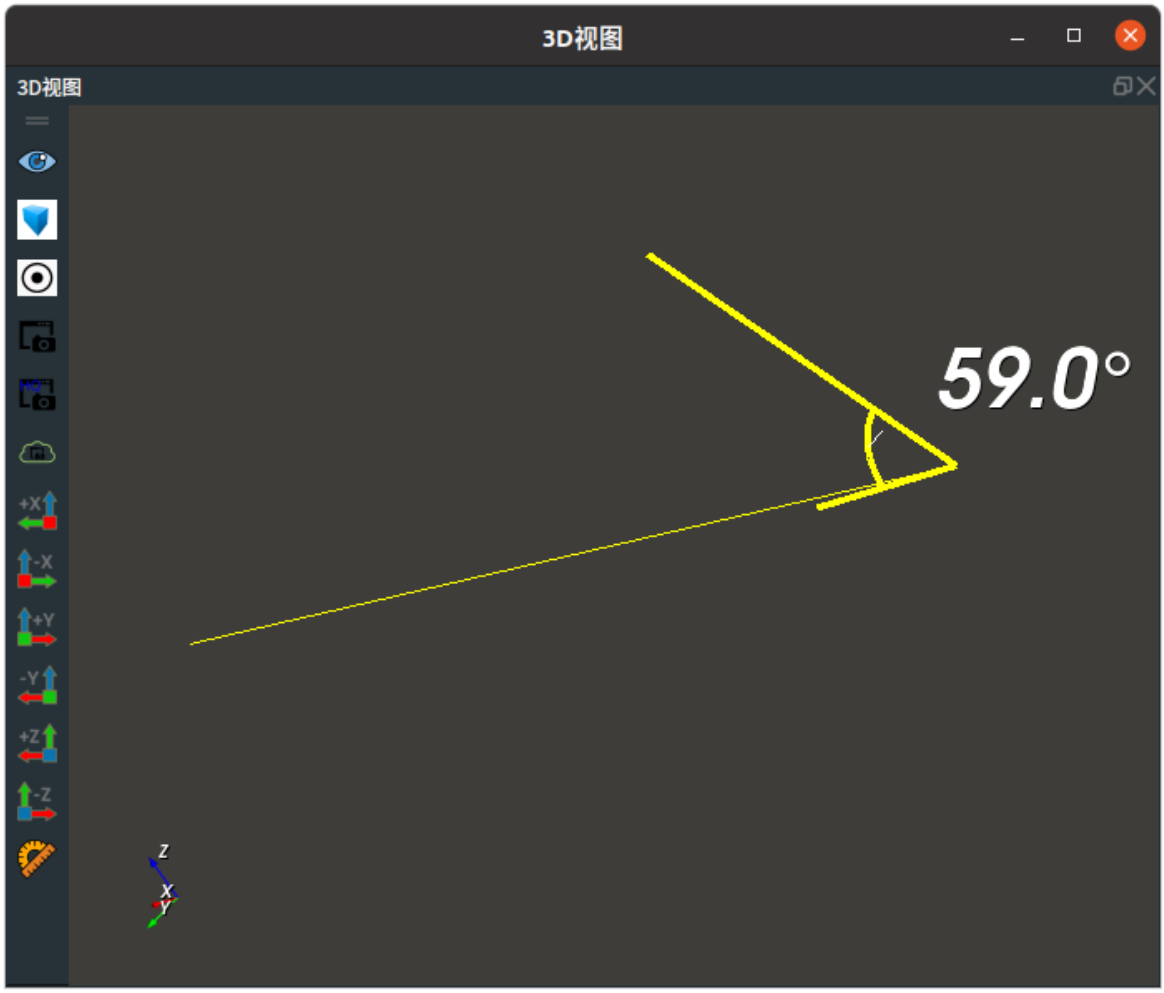
运行结果

如下图所示，展示了 LineOperation 算子的结果。

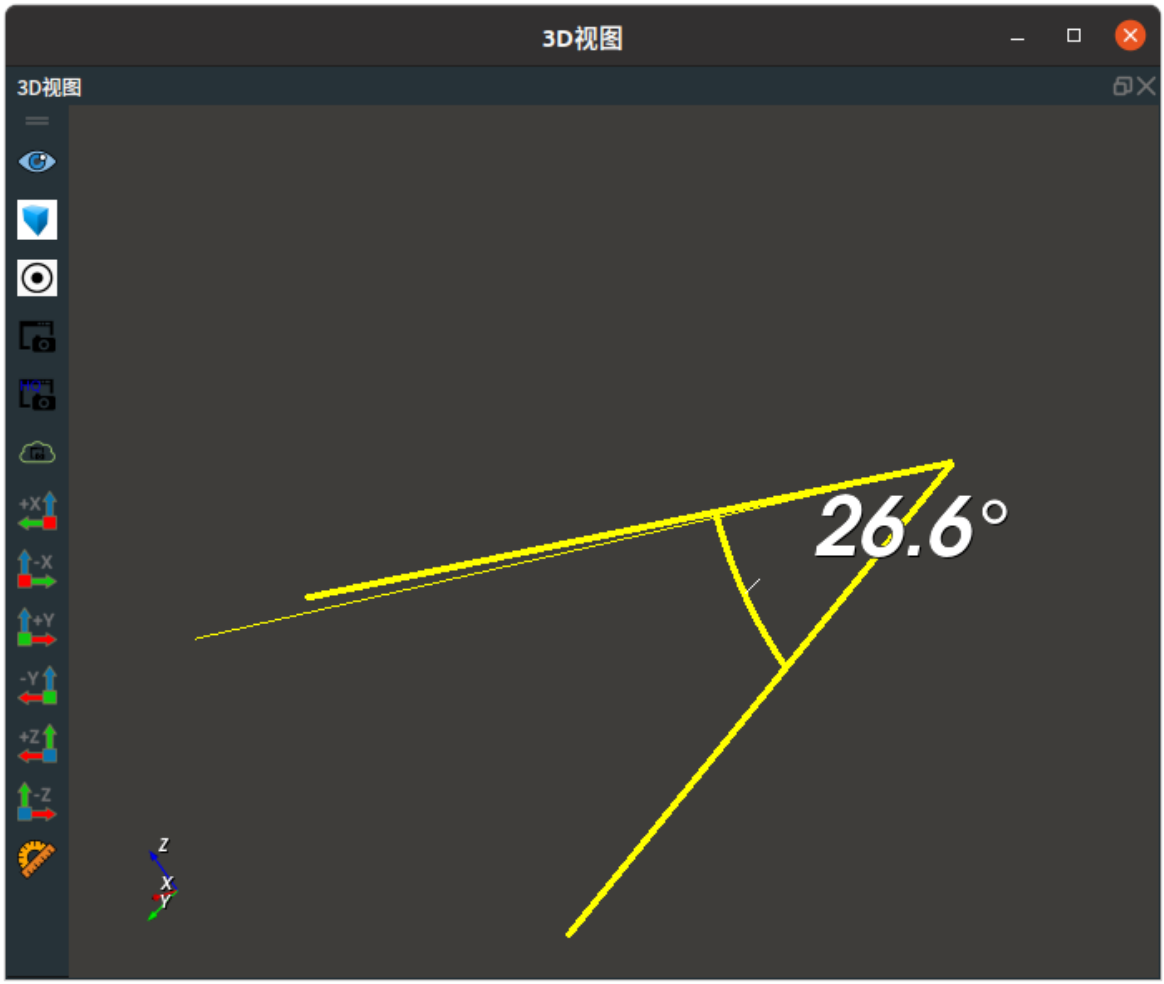
- 与XY平面夹角



- 与XZ平面夹角



- 与YZ平面夹角



LineCrossAngle

将 LineOperation 算子的 **类型** 属性选择 LineCrossAngle ，求两条线之间的夹角角度。夹角角度起点为 Line1 的 P2 ，角度终点为 Line2 的 P2。

算子参数

- **cross_angle**：设置 cross_angle 角度在 3D 视图中的可视化属性。
  - 打开 cross_angle 角度可视化。
  - 关闭 cross_angle 角度可视化。
  - 设置 3D 视图中 cross_angle 角度的颜色。取值范围：[-2,360] 。默认值：60 。
  - 设置 cross_angle 角度线的线宽。取值范围：[0,100] 。默认值：1 。
  - 标注角显示。

数据信号输入输出

输入：

line1：

- 数据类型：Line
- 输入内容：线段

line2：

- 数据类型：Line
- 输入内容：线段

输出：

`cross_angle`：

- 数据类型：Angle
- 输出内容：两条线之间的夹角角度

功能演示

使用 LineOperation 算子中 LineCrossAngle 求两条线的夹角角度。

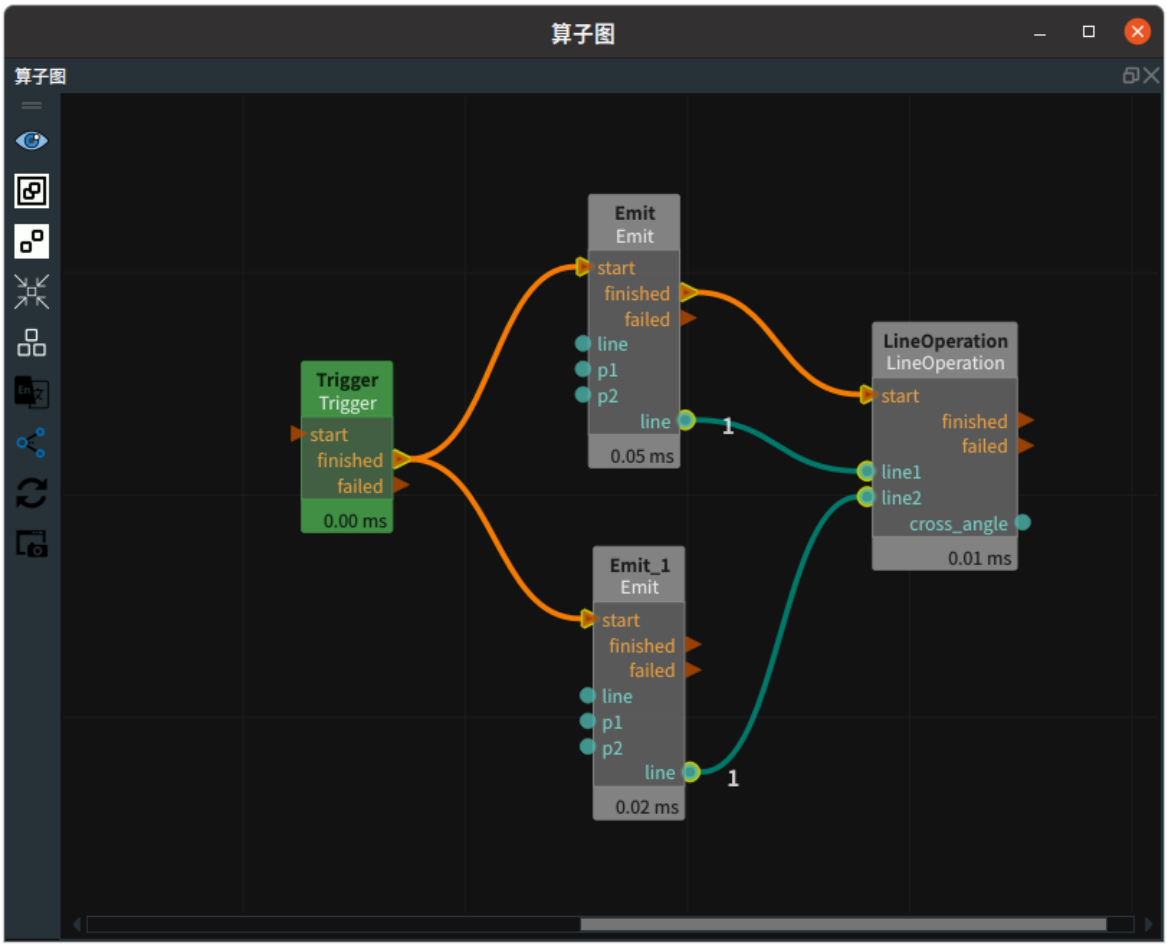
步骤1：算子准备

添加 Trigger 、Emit（2个） 、LineOperation 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
    - 类型 → Line
    - P1 → -1 0 0 0 0 0
    - P2 → 1 0 0 0 0 0
    - 线段 →  可视
2. 设置 Emit_1 算子参数：
    - 类型 → Line
    - P1 → 0 0 0 0 0 0
    - P2 → 1 1 0 0 0 0
    - 线段 →  可视
3. 设置 LineOperation 算子参数：
    - 类型 → LineCrossAngle
    - cross_angle →  可视

步骤3：连接算子

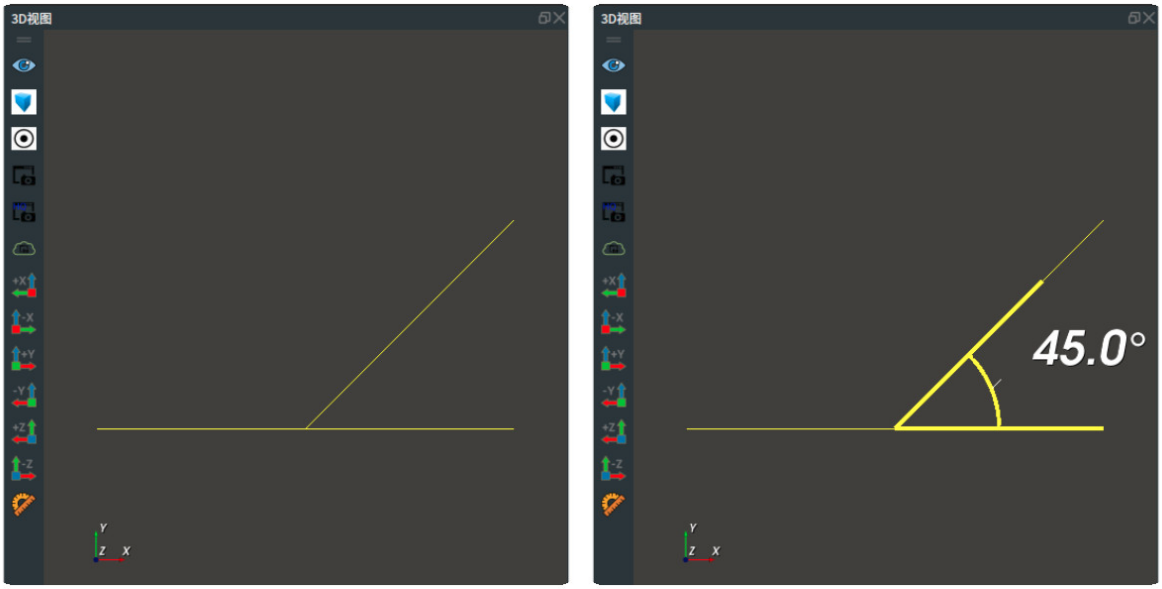


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，左图是生Emit 算子生成的两条线段。右图为 LineCrossAngle 计算出的两条线段的夹角。



LineCircleOperation 线段圆圈处理工具

LineCircleOperation 算子用于线段圆圈处理。

类型	功能
LineToCircleDistance	用于计算线段到圆圈的最短距离。

LineToCircleDistance

将 LineCircleOperation 算子的**类型**选择 LineToCircleDistance，用于计算线段到圆圈的最短距离。

算子参数

- **trans**：当线段和圆圈有多个交点时，确定交点。
  - True：选择圆圈中心点 pose 的 x 轴正方向的交点 。
  - False：选择圆圈中心点 pose 的 x 轴负方向的交点。
- **line_point**：设置线上点在 3D 视图中的可视化属性。
  -  打开 line_point 可视化。
  -  关闭 line_point 可视化。
  -  设置 line_point 在 3D 视图中的比例。取值范围：[0.001,10] 。默认值：0.1 。
- **Circle_point**：设置圆上点在 3D 视图中的可视化属性。
  -  打开 Circle_point 可视化。
  -  关闭 Circle_point 可视化。
  -  设置 Circle_point 在 3D 视图中的比例。取值范围：[0.001,10] 。默认值：0.1 。
- **距离/distance**：设置计算后结果的曝光属性。曝光后可与交互面板中输出工具“按钮”进行绑定。
  -  打开曝光。
  -  关闭曝光。

数据信号输入输出

输入：

- **line**：
  - 数据类型：Line
  - 输入内容：线段数据
- **circle**：
  - 数据类型：Circle
  - 输入内容：圆圈数据

输出：

- **line_point**：
  - 数据类型：Pose
  - 输出内容：线上点坐标数据
- **circle_point**：

    - 数据类型：Pose
    - 输出内容：圆圈上点坐标数据
- `distance`：
    - 数据类型：String
    - 输出内容：线段与圆圈的距离

功能演示

使用 LineCircleOperation 算子中 LineToCircleDistance，计算线段到圆圈的最短距离。

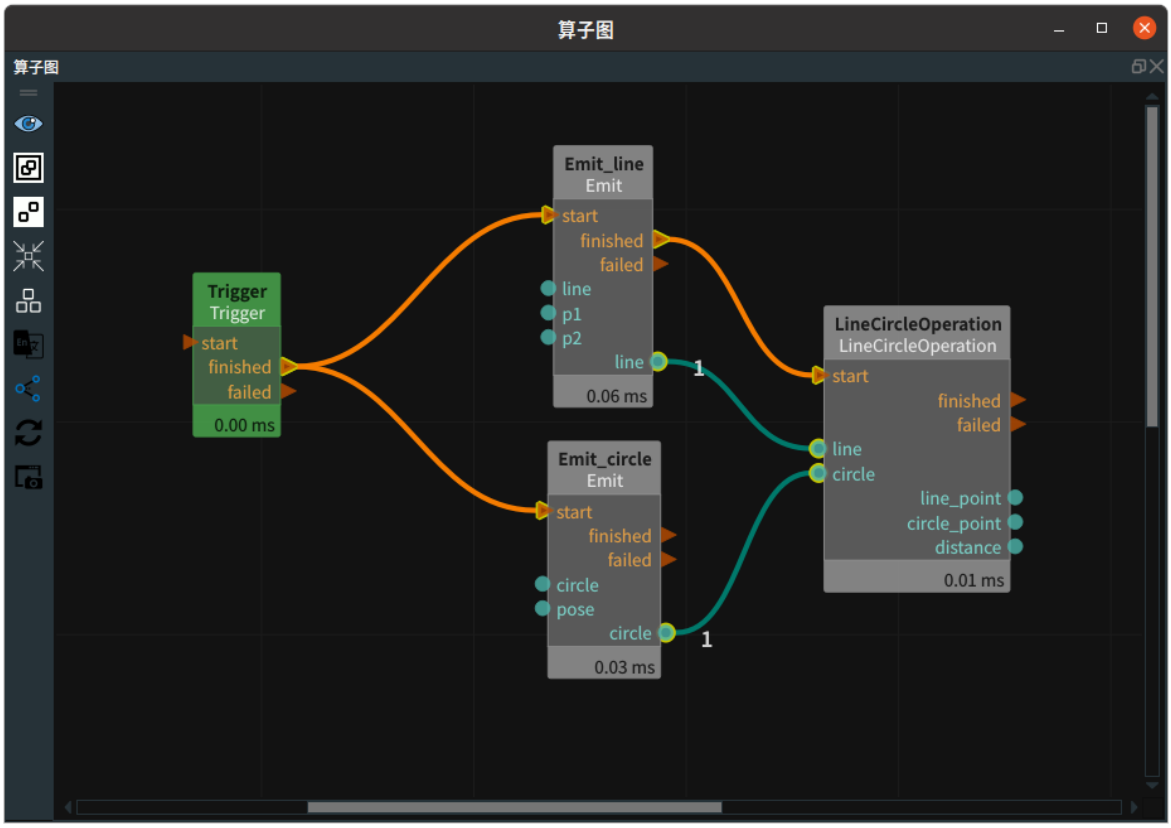
步骤1：算子准备

添加 Trigger 、Emit(2个) 、LineCircleOperation 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
    - 算子名称→ Emit_line
    - 类型 → Line
    - p1 → 3 1.5 0 0 0 0
    - p2 → -1 1.5 0 0 0 0
    - 线段 →  可视
2. 设置 Emit_1 算子参数：
    - 算子名称 → Emit_Circle
    - 类型 → circle
    - 坐标 → 1 0 0 0 0 0
    - 半径 → 1
    - 圆圈 →  可视
3. 设置 LineCircleOperation 算子参数：
    - 类型 → LineToCircleDistance
    - line_point →  可视
    - line_circle →  可视

步骤3：连接算子



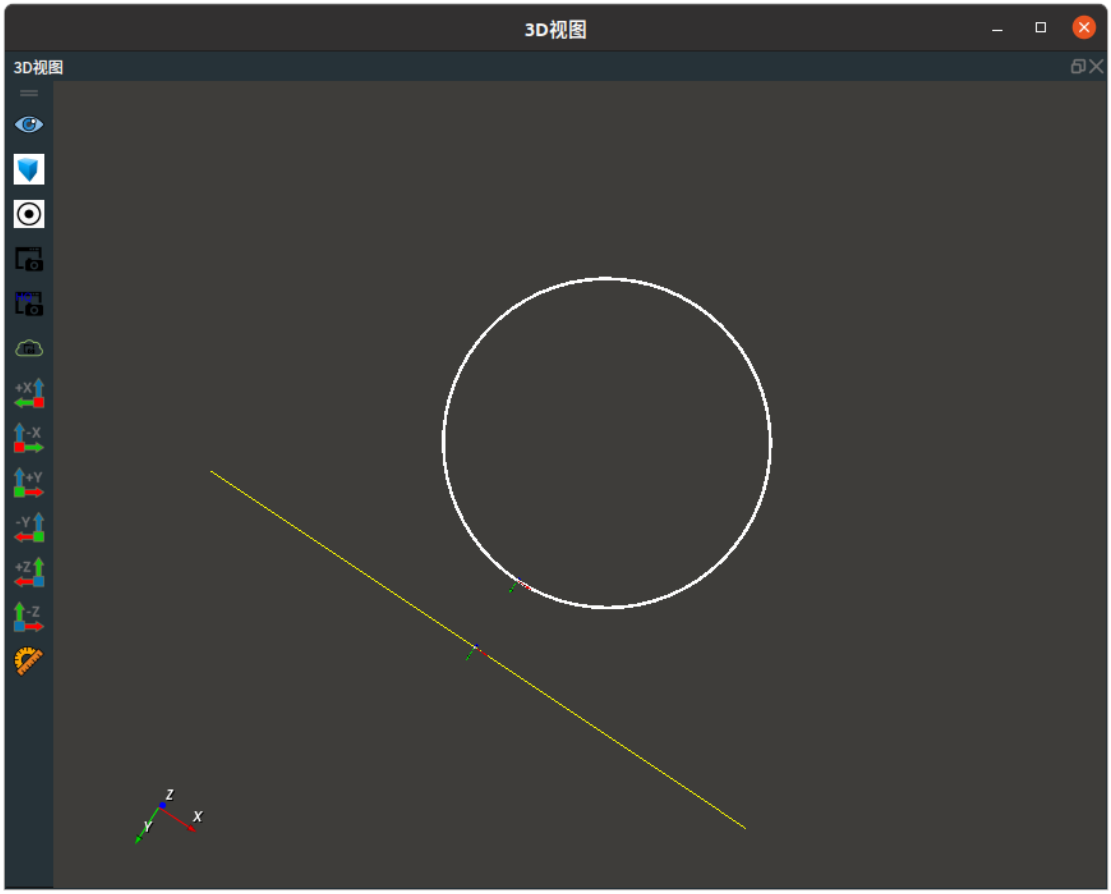
步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

1. 如下图所示，在日志视图中显示线段与圆圈的距离为 0.5m。

2. 在 3D 视图中分别显示 Emit（emit_line）、Emit_1(Emit_Circle)、LineCircleOperation 中 line_point、circle_point 的可视化结果。


3D视图
3D视图
+X
-X
+Y
-Y
+Z
-Z
Z
X
Y

GeometryProbe 几何探测

GeometryProbe 算子用于在原始点云的指定区域中筛选出符合要求的轮廓点云。

type	功能
Line	将原始点云在给定的旋转立方体内裁剪，并沿着立方体的某个指定轴方向进行线性轮廓探测。
Circle	根据原始点云以及初始圆心的位置，探测内圆或者外圆轮廓点云。

Line

将 GeometryProbe 算子的 **类型** 设置为 Line ，用于将原始点云在给定的旋转立方体内裁剪，并沿着立方体的某个指定轴方向进行线性轮廓探测。

算子参数

- **探测方向/probe_direction**：探测方向。包含 "x+" , "x-" , "y+" , "y-" , "z+" , "z-" 六个选项。例如，"x+" 代表沿着 ref_cube 的 x 正轴方向，从 ref_cube 的外面朝向里面进行探测。
- **圆锥体顶角度/circular_cone_angle**：排除锥的角度，一般保持默认的45度角即可。以原始点云在裁剪区域内的某个点为顶点，以上述探测方向作为轴线方向，根据该锥角可以在空间作一个圆锥，被该圆锥囊括的所有其他点会被排除，之后遍历其他点重复迭代，通俗的理解为切割出圆锥以外的点云进行处理。
- **有效点百分比数/valid_points_percentage**：有效点云比例。即对裁剪后的点云进行预排除时的保留比例。
- **步进值/step_value**：根据排除锥进行排除迭代运算时的步长 。一般保持默认即可。*该值一般对效率有所影响，不会影响精度*。
- **有效点云/valid_cloud**：设置裁剪后的点云在 3D 视图中的可视化属性。
- **线段点云/line_cloud**：设置最终的线性轮廓点云在 3D 视图中的可视化属性。
  -  打开线性轮廓点云可视化。
  -  关闭线性轮廓点云可视化。
  -  设置 3D 视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  -  设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。

数据信号输入端口

输入：

- **cloud**：
  - 数据类型：PointCloud
  - 输入内容：点云原始点云
- **ref_cube**：
  - 数据类型：Cube
  - 输入内容：参照立方体

输出：

- **crop_cloud**：

    - 数据类型：PointCloud
    - 输出内容：原始点云裁剪后的点云
- `line_cloud`：
    - 数据类型：PointCloud
    - 输出内容：根据裁剪点云进行探测获得的线性轮廓点云

功能演示

使用 GeometryProbe 算子中 Line 将加载点云在给定的旋转立方体内裁剪，并沿着立方体的某个指定轴方向进行线性轮廓探测。

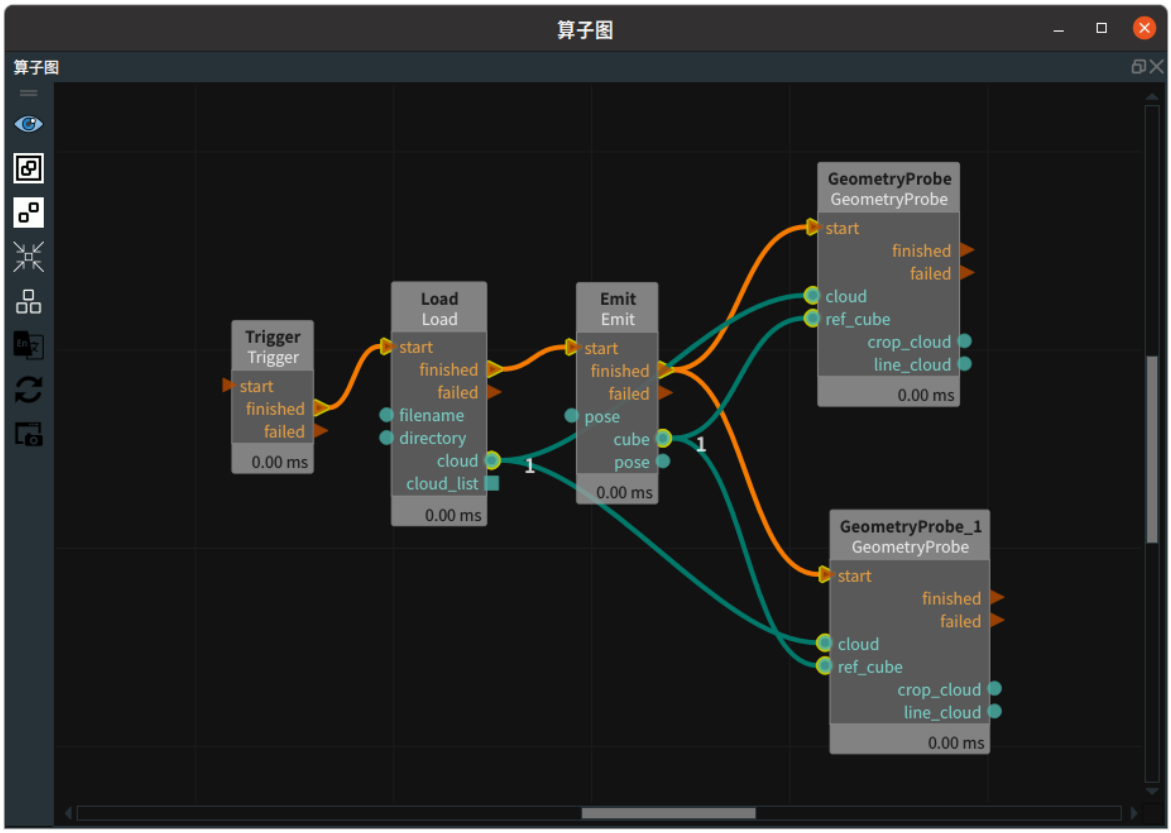
步骤1：算子准备

添加 Trigger 、Load 、Emit 、GeometryProbe(2个) 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
    - 类型 → cube
    - 坐标 → 0.199179 0.26519 0.092586 2.61142 -2.82994 3.07943
    - 宽度 → 0.02
    - 高度 → 0.01
    - 深度 → 0.005
    - 立方体 → 可视
    - 坐标 → 可视
2. 设置 Load 算子参数：
    - 类型 → PointCloud
    - 文件 → ••• → 选择点云文件名(*example_data/pointcloud/cloud.pcd*)
    - 点云 → 可视
3. 设置 GeometryProbe 算子参数
    - 类型 → Line
    - 探测方向 → y-
    - 圆锥体顶角度 → 45
    - 有效点百分比数 → 10
    - 步进值 → 5
    - 有效点云 → 可视
    - 线段点云 → 可视
4. 设置GeometryProbe_1算子参数
    - 类型 → Line
    - 探测方向 → x-
    - 圆锥体顶角度 → 45
    - 有效点百分比数 → 10
    - 步进值 → 5
    - 有效点云 → 可视
    - 线段点云→ 可视

步骤3：连接算子

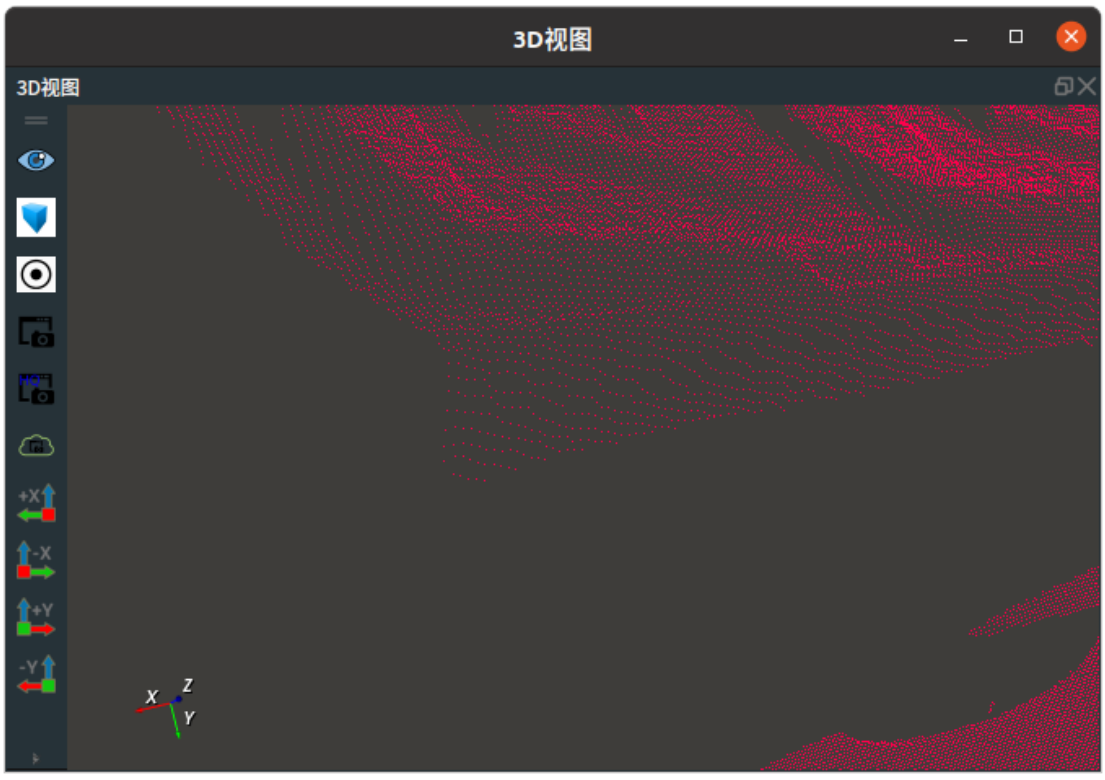


步骤4：运行

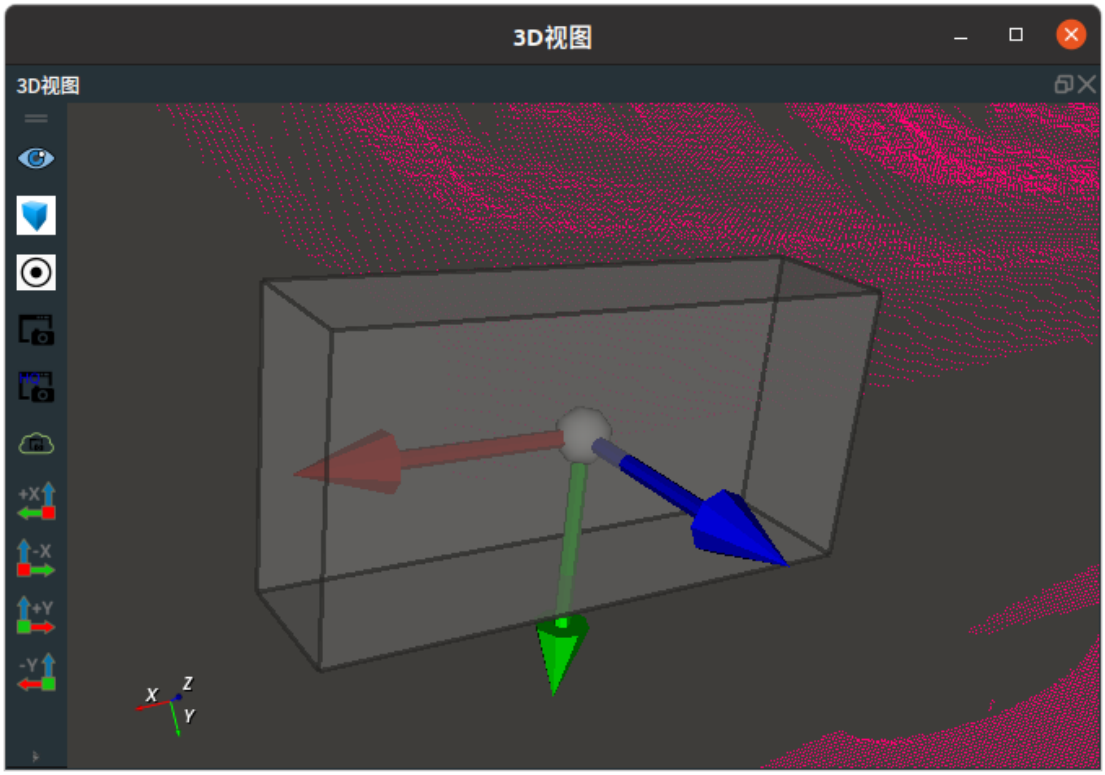
点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

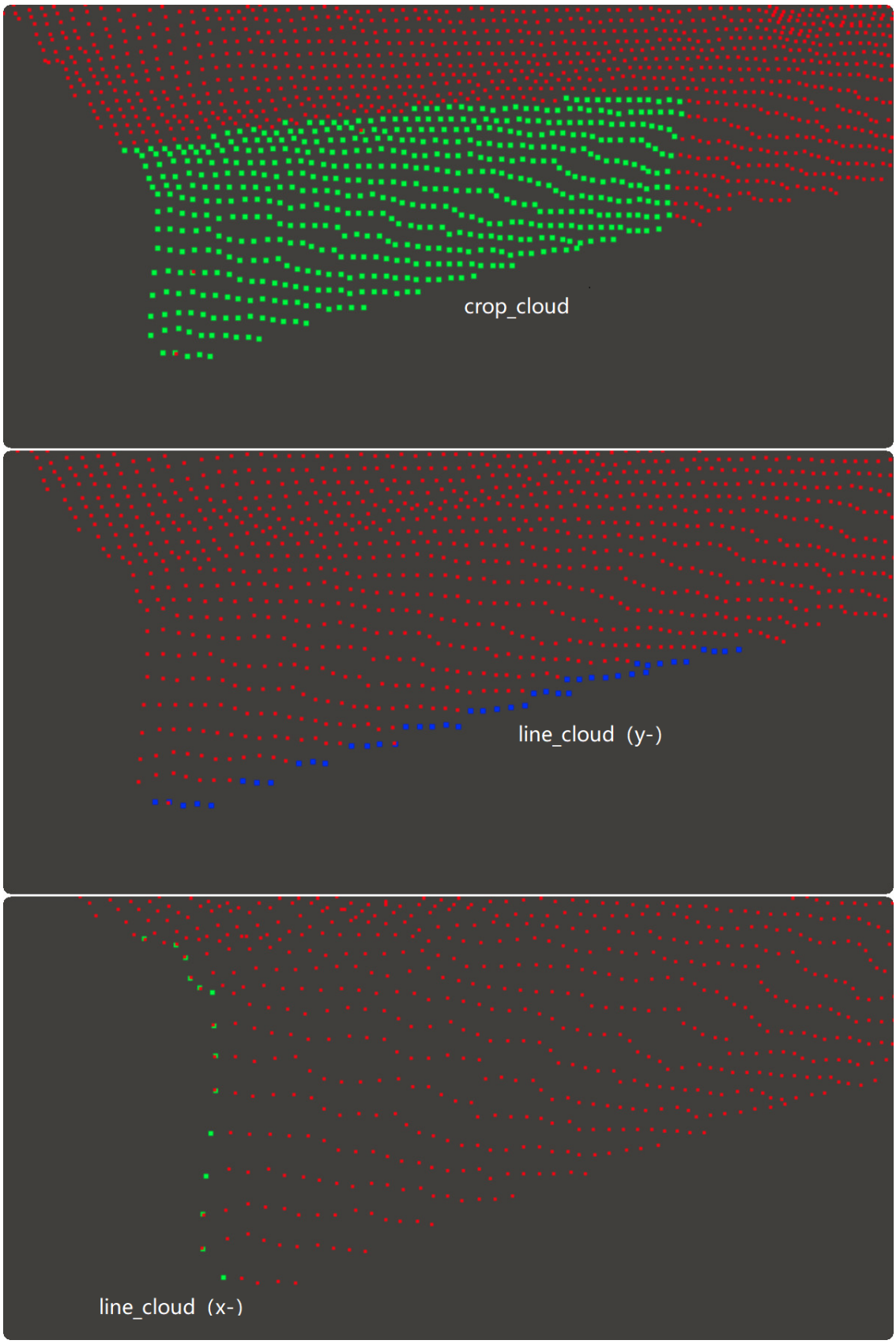
1. 如下图所示，可见到点云（局部）显示如下 。



2. 打开Emit算子的cube可视化，第二次运行 trigger 算子，并调整 cube 的长宽高以及轴向，生成的 Cube（红色轴为 x，绿色轴为 y，蓝色轴为 z ）如下所示。



3. 最后更改两个 GeometryProbe 算子的 probe_direction 属性，分别探测 y 轴负方向以及x轴负方向的线性轮廓，两者共同的 crop_cloud 的显示效果以及两者分别的点云探测结果如下。



Circle

将 GeometryProbe 算子的 **类型** 设置为 Circle ，用于根据原始点云以及初始圆心的位置，探测内圆或者外圆轮廓。

算子参数

- **探测方向/probe_direction**：探测方向。
  - external：代表从点云外面朝向预估圆心的方向探测外圆轮廓。
  - internal：代表从点云内部朝向预估圆心的方向探测内圆轮廓。
- **圆锥体顶角度/circular_cone_angle**：排除锥的角度，一般保持默认的45度角即可。" external " 模式下，以有效点云的某个点为顶点，同预估圆心的连线作为轴线终点，根据该锥角可以在空间作一个圆锥，被该圆锥囊括的所有其他点会被排除，之后遍历其他点重复迭代；" internal " 模式下，顶点改为预估圆心，轴线终点改为有效点云的某个点。
- **有效点百分比数/valid_points_percentage**：有效点百分比数。
- **步进值/step_value**：根据排除锥进行排除迭代运算时的步长，一般保持默认即可，该值一般对效率有所影响，不会影响精度。
- **有效点云/valid_cloud**：对预处理后的有效点云进行显示控制，默认为关闭状态，参数介绍如下所示。
- **圆圈点云/circle_cloud**：对最终圆形轮廓点云进行显示控制。
  -  打开最终圆形轮廓点云可视化。
  -  关闭最终圆形轮廓点云可视化。
  -  设置 3D 视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  -  设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。

数据信号输入端口

输入：

- **cloud**：
  - 数据类型：PointCloud
  - 输入内容：点云原始点云
- **initial_pose**：
  - 数据类型：Pose
  - 输入内容：原始点云的圆心位置估计，*实际仅使用 Pose 的 xyz 三个数值*

输出：

- **valid_cloud**：
  - 数据类型：PointCloud
  - 输出内容：原始点云预处理后保留的有效点云
- **circle_cloud**：
  - 数据类型：PointCloud
  - 输出内容：根据有效点云进行探测获得的圆形轮廓点云

功能演示

使用 GeometryProbe 算子中 Circle ，用于根据加载点云以及初始圆心的位置，探测内圆或者外圆轮廓点云。

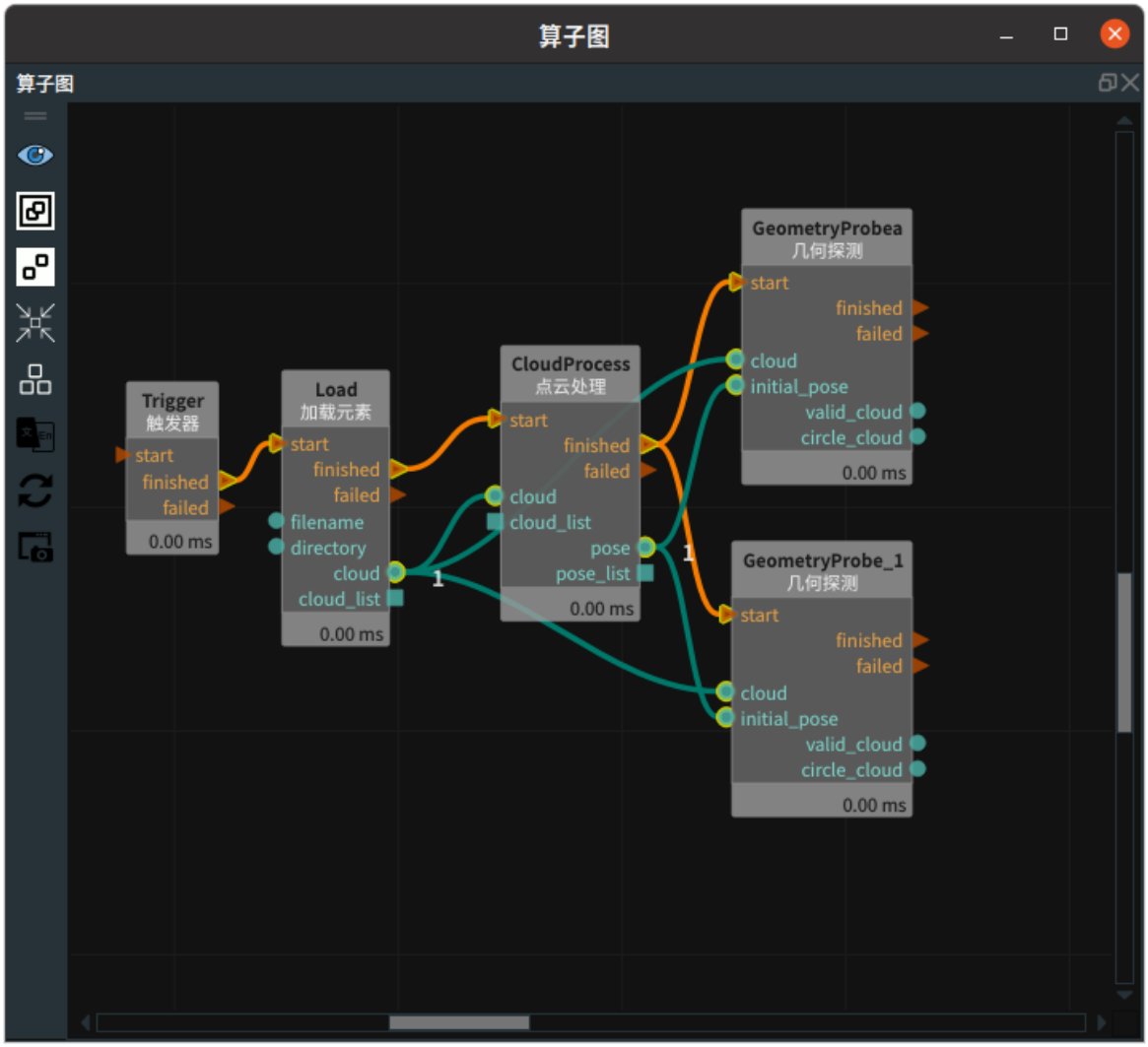
步骤1：算子准备

添加 Trigger 、Load 、CloudProcess、GeometryProbe (2个)算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 类型 → pointcloud
    - 文件 → ••• → 选择点云文件名(*example_data/pointcloud/circle_crop.pcd*)
    - 点云 →  可视
2. 设置 GeometryProbe 算子参数：
    - 类型 → Circle
    - 探测方向 → external
    - 圆锥体顶角度 → 70
    - 有效点百分比数 → 5
    - 步进值 → 5
    - 圆圈点云→  可视
3. 设置GeometryProbe_1算子参数：
    - 类型 → Circle
    - 探测方向 → internal
    - 圆锥体顶角度 → 45
    - 有效点百分比数 → 5
    - 步进值 → 5
    - 圆圈点云→  可视
4. 设置 CloudProcess 算子参数：
    - 类型 → CloudCentroid
    - 坐标→  可视

步骤3：连接算子

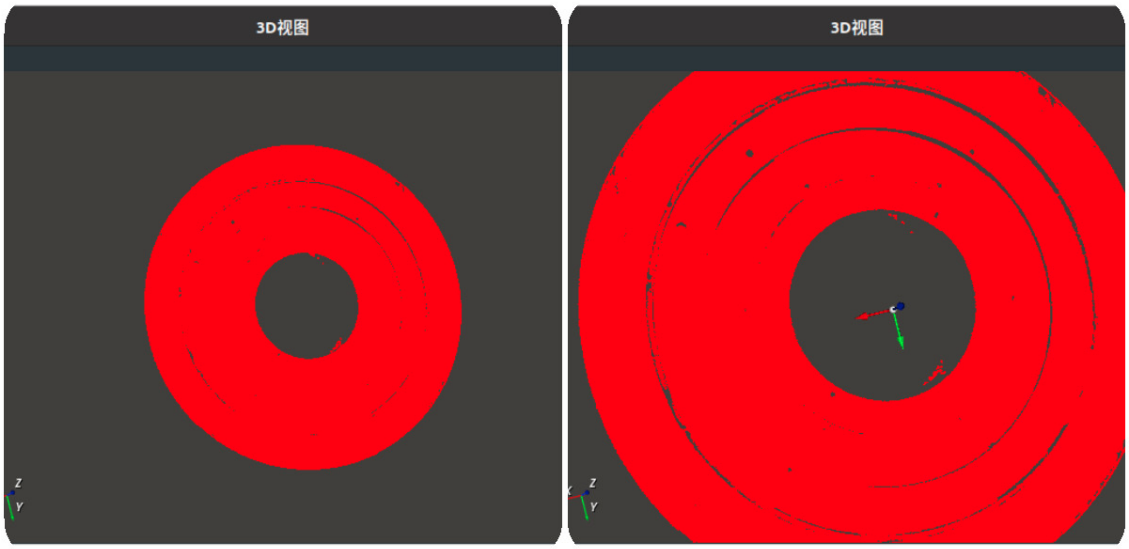


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

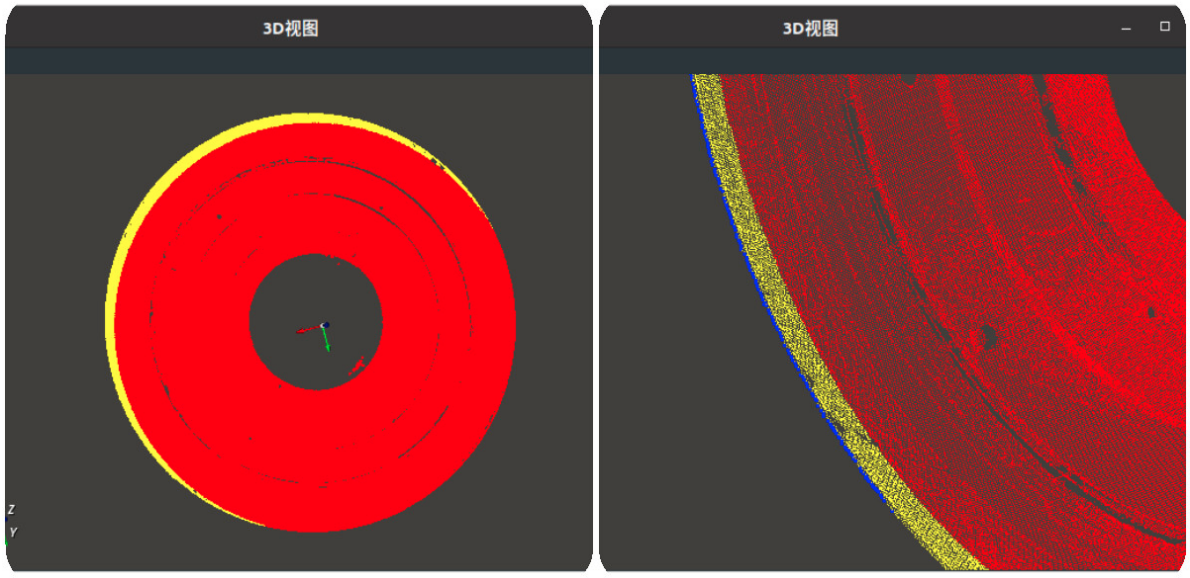
运行结果

1. 3D 视图中显示如下。左图为 Load 算子的点云可视化，右图为 CloudProcess 算子的pose可视化。

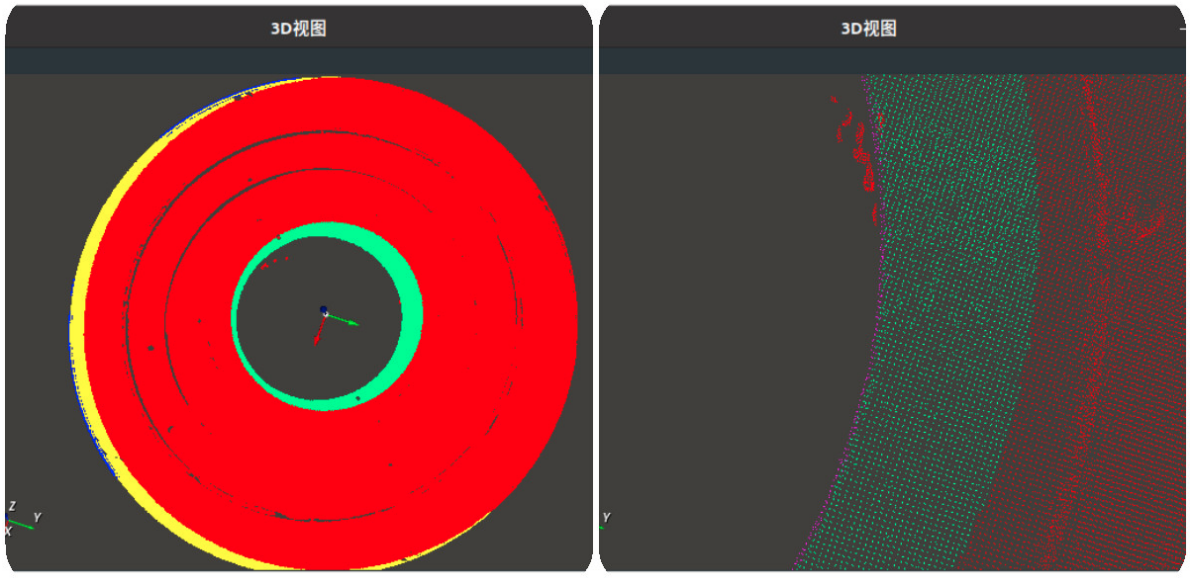


2. 后续运行效果示意图，分别展示如下：

- 左图：打开 GeometryProbe 算子(external 模式)的 valid_cloud 的可视化选项，右图：打开 GeometryProbe 算子(external 模式)的 circle_cloud 的可视化选项的局部放大图。



- 左图：打开 GeometryProbe 算子(internal 模式)的 valid_cloud 的可视化选项。右图：打开 GeometryProbe 算子 (internal 模式) 的 circle_cloud 的可视化选项的局部放大图。



ScalePose 修改坐标比例

ScalePose 算子用于调整坐标比例，分为 Normal 和 Flexible 两种。RVS X Y Z 默认单位：m。 Roll Pitch Yaw 默认单位：弧度制。

type	功能
Normal	分别调整 pose 的（X、Y 、Z）和（Roll、Pitch、Yaw）的比例。
Flexible	单独调整 pose 的 X、Y 、Z、Roll、Pitch、Yaw 比例。

Normal

将 ScalePose 算子的 **类型** 设置为 Normal ，分别调整 pose 的（ X 、Y 、Z ）和（ Roll 、Pitch 、Yaw ）比例。

算子参数

- **XYZ比例/scale_xyz**：修改 pose 的 x y z 的单位。输入数值后，x y z 与该数值相乘。默认值：0.001 。表示将毫米切换为米。反之设：1000 。即从米切换为毫米。
- **RPY比例/scale_rpy**：修改 pose 中 Roll Pitch Yaw 的单位。输入数值后，Roll Pitch Yaw 与该数值相乘。默认值：0.01745329 。表示将角度切换为弧度。反之设：57.2957795 。即从将弧度切换为角度。
- **坐标/pose**：设置调整比例后的 pose 在 3D 视图中的可视化属性。
  -  打开 pose 可视化。
  -  关闭 pose 可视化。
  -  设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。
- **坐标列表/pose_list**：设置调整比例后的 pose 列表在 3D 视图中的可视化属性。值描述与 pose 一致。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **pose**：
  - 数据类型：Pose
  - 输入内容：pose 数据
- **pose_list**：
  - 数据类型：PoseList
  - 输入内容：pose 列表数据

输出：

- **pose**：
  - 数据类型：Pose
  - 输出内容：调整比例后的 pose 数据
- **pose_list**：
  - 数据类型：PoseList

- 输出内容：调整比例后的 pose 列表数据

功能演示

使用ScalePose 中 Normal 将加载的 pose 中 x y z 从米切换为毫米， Roll、Pitch、Yaw 将弧度切换为角度。

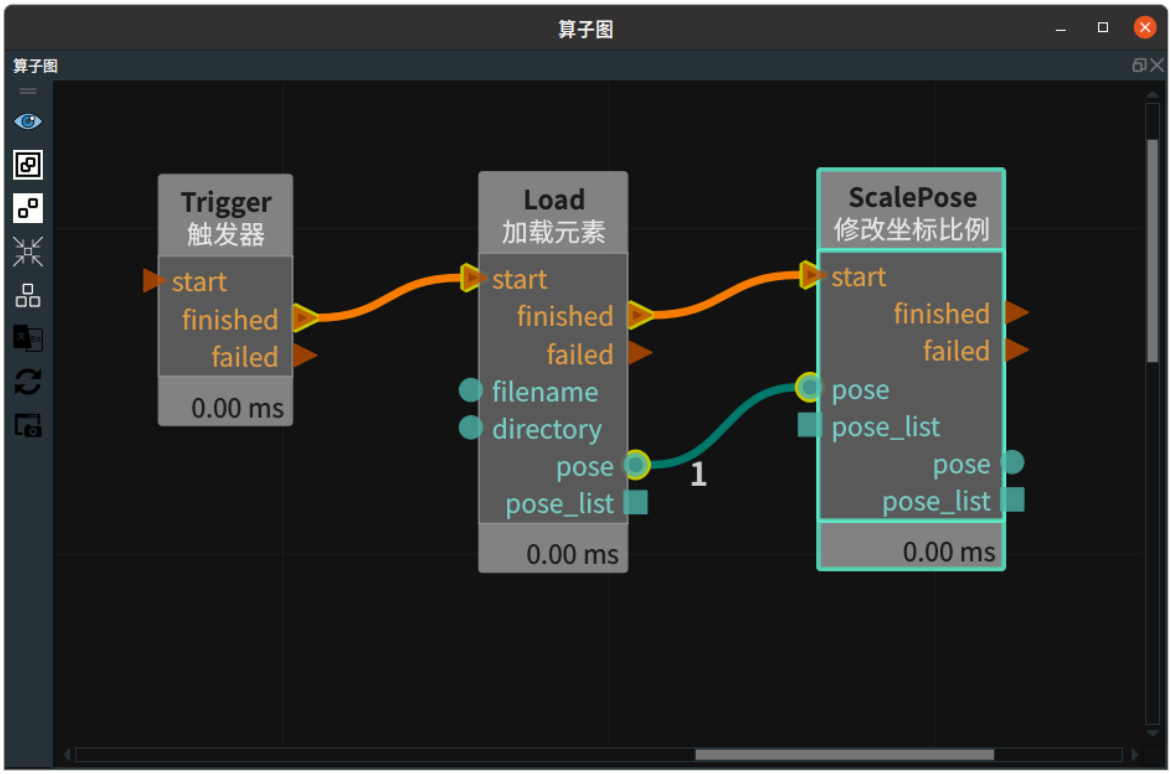
步骤1：算子准备

添加 Trigger 、Load 、ScalePose 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → pose
   - 文件 → ••• → 选择 pose 文件名(*example_data/pose/tcp.txt*)
   - 坐标 → 可视
2. 设置 ScalePose 算子参数：
   - 类型 → Normal
   - XYZ比例 → 1000
   - RPY比例 → 57.29578015
   - 坐标 → 可视

步骤3：连接算子



步骤4：运行

1. 分别将两个可视化属性与交互面板——“坐标输出”属性进行绑定。
2. 点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

结果如下图所示，在交互面板中分别显示修改比例前后的坐标。pose 的 x y z 从米切换为毫米， Roll、Pitch、Yaw 将弧度切换为角度。

Flexible

将 ScalePose 算子的 **类型** 设置为 Flexible ，单独调整 pose 的X 、Y 、Z 、Roll 、Pitch 、Yaw 比例。

算子参数

- **X比例/scale_x**：修改 pose 中 x 的比例。输入数值后，x 与该数值相乘。默认值：1。
- **Y比例/scale_y**：修改 pose 中 y 的比例。输入数值后，y 与该数值相乘。默认值：1。
- **Z比例/scale_z**：修改 pose 中 z 的比例。输入数值后，z 与该数值相乘。默认值：1。
- **Roll比例/scale_roll**：修改 pose 中 roll 的比例。输入数值后，roll 与该数值相乘。默认值：1。
- **Pitch比例/scale_pitch**：修改 pose 中 pitch 的比例。输入数值后，pitch 与该数值相乘。默认值：1。
- **Yaw比例/scale_yaw**：修改 pose 中 yaw 的比例。输入数值后，yaw 与该数值相乘。默认值：1。
- **坐标/pose**：设置调整比例后的 pose 在 3D 视图中的可视化属性。
  - 打开 pose 可视化。
  - 关闭 pose 可视化。
  - 设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。
- **坐标列表/pose_list**：设置调整比例后的 pose 列表在 3D 视图中的可视化属性。值描述与 **坐标** 一致。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- `pose`：
  - 数据类型：Pose
  - 输入内容：pose 数据
- `pose_list`：
  - 数据类型：PoseList
  - 输入内容：pose 列表数据

输出：

- `pose`：
  - 数据类型：Pose
  - 输出内容：调整比例后的 pose 数据
- `pose_list`：
  - 数据类型：PoseList
  - 输出内容：调整比例后的 pose 列表数据

功能演示

使用 ScalePose 中 Flexible 将加载的 pose 中 x 从米切换为毫米 。

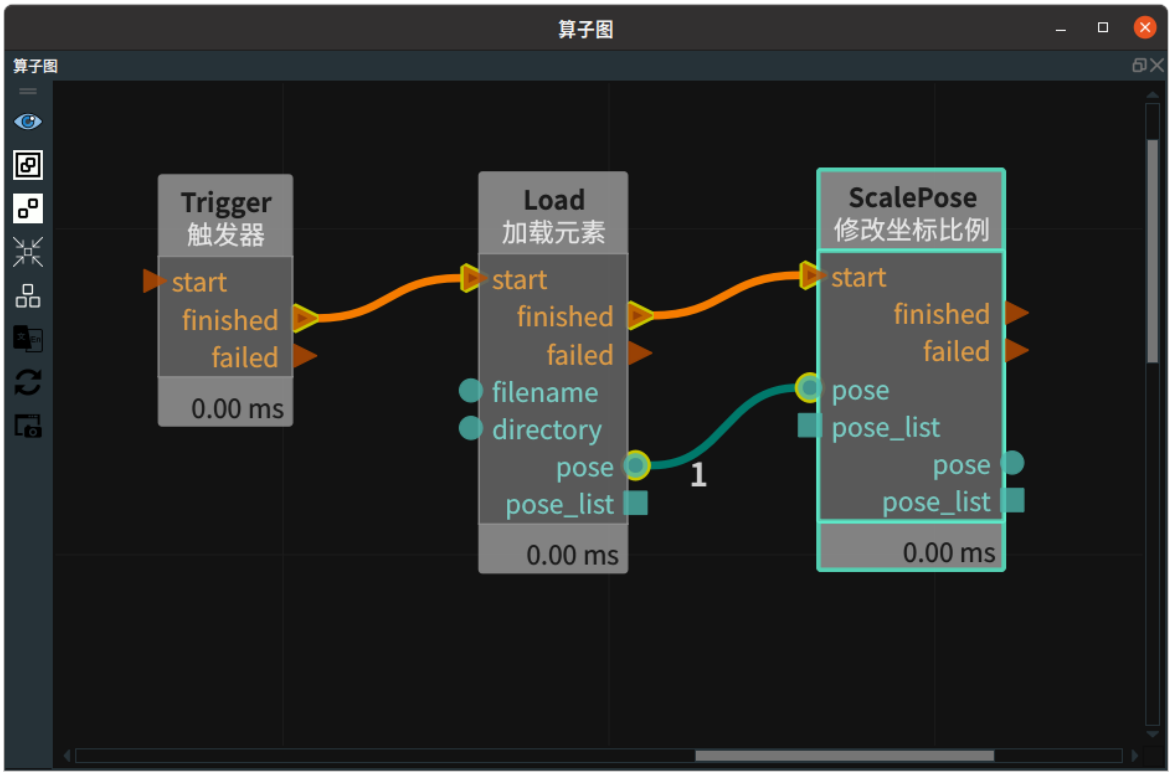
步骤1：算子准备

添加 Trigger、Load 、ScalePose 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → pose
   - 文件 → ••• → 选择 pose 文件名(*example_data/pose/tcp.txt*)
   - 坐标 →  可视
2. 设置 ScalePose 算子参数：
   - 类型 → Flexible
   - X比例 → 1000
   - 坐标 →  可视

步骤3：连接算子



步骤4：运行

1. 分别将两个可视化属性与交互面板——“坐标输出”属性进行绑定。
2. 点击 运行按钮，触发 Trigger 算子。

运行结果

结果如下图所示，在交互面板中分别显示修改比例前后的坐标。pose 中 x 从米切换为毫米。

GenerateSamplePose 生成样本坐标

GenerateSamplePose 算子用于生成样本坐标。包含 BoxGridSample 、BoxRandomSample 、SphereRandomSample 、CameraPoseSample 。

type	功能
BoxGridSample	在设定范围均匀生成样本坐标。
BoxRandomSample	在设定范围随机生成样本坐标。
SphereRandomSample	在球体表面随机生成样本坐标。
CameraPoseSample	根据棋盘的位置生成相机的多组拍照位。

BoxGridSample

将 GenerateSamplePose 算子的 **类型** 设置为 BoxGridSample ，用于设定范围均匀生成样本坐标。

算子参数

- **起始坐标/start_pose**：坐标起点。
- **终止坐标/end_pose**：坐标终点。
- **x方向步长/step_x**：在 X 轴方向，从 min_position 到 max_position 范围内均匀生成的样本坐标数量。默认值：1。
- **y方向步长/step_y**：在 Y 轴方向，从 min_position 到 max_position 范围内均匀生成的样本坐标数量。默认值：1。
- **z方向步长step_z**：在 Z 轴方向，从 min_position 到 max_position 范围内均匀生成的样本坐标数量。默认值：1。
- **Roll旋转步长/step_roll**：围绕 X 轴旋转方向，从 min_position 到 max_position 范围内均匀生成的样本坐标数量。默认值：1。
- **Pitch旋转步长/step_pitch**：围绕 Y 轴旋转方向，从 min_position 到 max_position 范围内均匀生成的样本坐标数量。默认值：1。
- **Yaw旋转步长/step_yaw**：围绕 Z 轴旋转方向，从 min_position 到 max_position 范围内均匀生成的样本坐标数量。默认值：1。
- **坐标列表/pose_list**：设置样本坐标 3D 视图中的可视化属性。
  -  打开样本坐标可视化。
  -  关闭样本坐标可视化。
  -  设置样本坐标的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

- **pose**：
  - 数据类型：Pose
  - 输入内容：pose 数据

输出：

- `pose_list`：
  - 数据类型：PoseList
  - 输出内容：样本坐标列表数据

功能演示

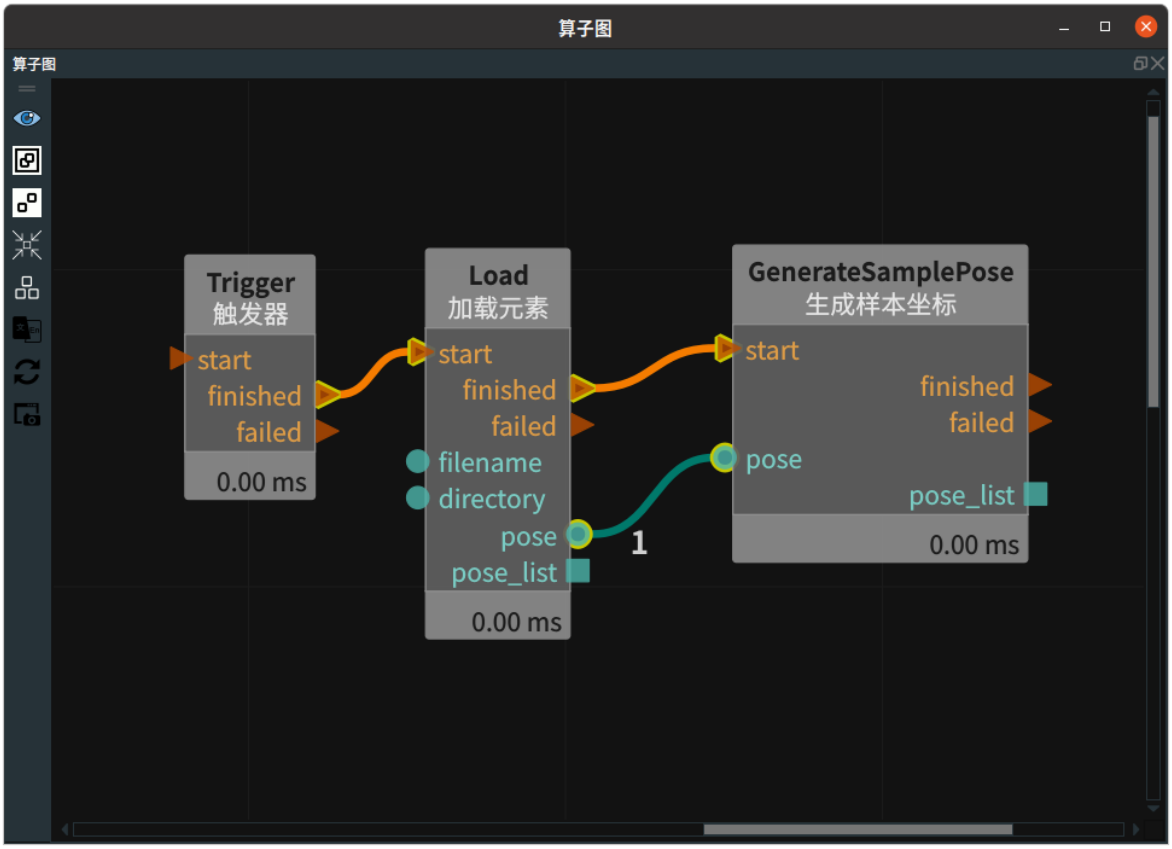
使用 GenerateSamplePose 中 BoxGridSample 根据加载的坐标在设置范围内均匀生成4个样本坐标。

步骤1：算子准备

添加 Trigger 、Load 、GenerateSamplePose 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → pose
   - 文件 → ••• → 选择 pose 坐标文件名(*example_data/pose/tcp.txt*)
2. 设置 GenerateSamplePose 算子参数：
   - 类型 → BoxGridSample
   - 起始坐标 → 1 0 0 0 0 0
   - 终止坐标 → 3 0 0 0 0 0
   - x方向步长 → 4
   - 坐标列表 →  可视

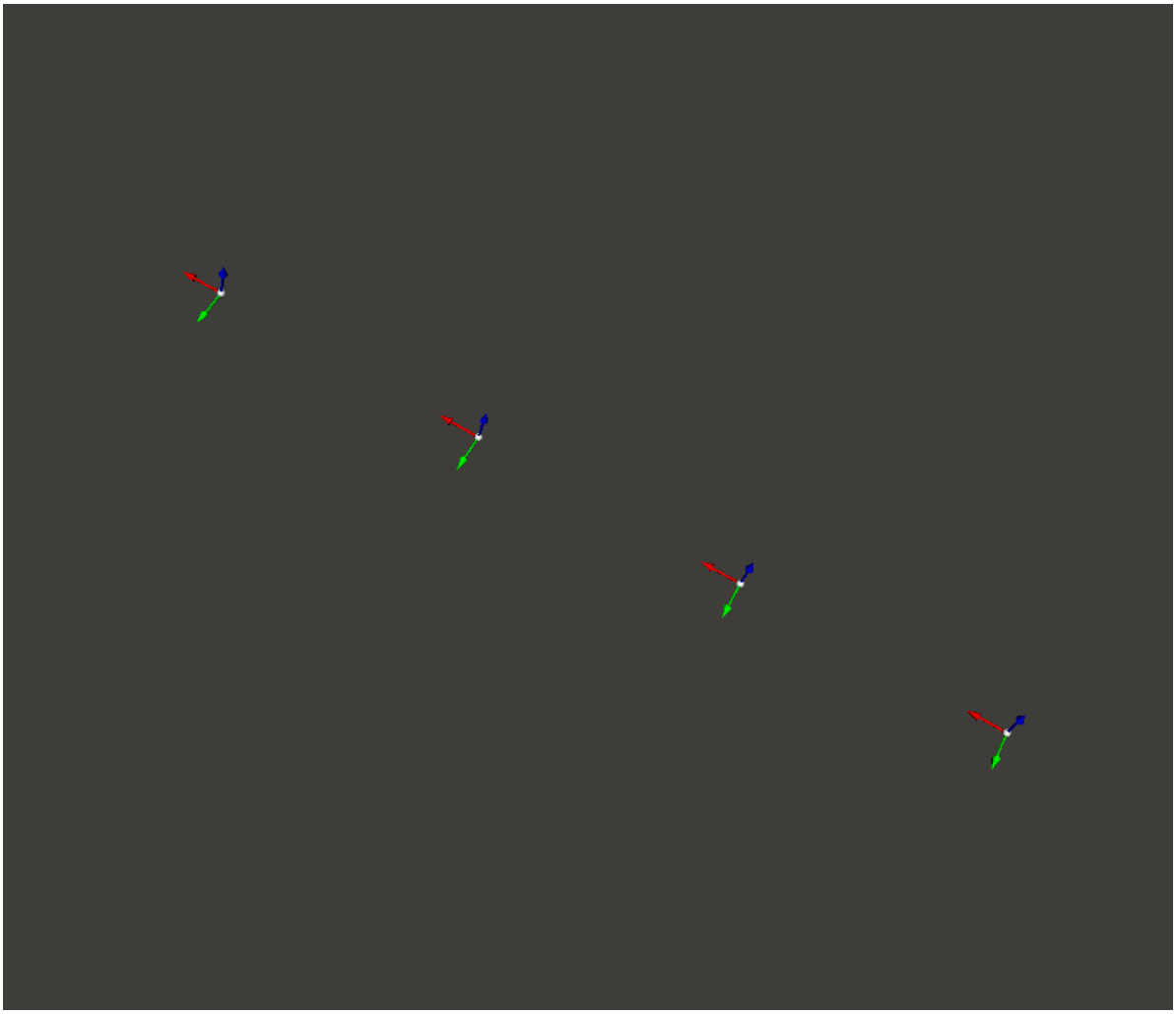


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

运行结果如下图所示，在 3D 视图中显示生成的 4 个样本坐标。



BoxRandomSample

将 GenerateSamplePose 算子的 **类型** 设置为 BoxRandomSample ，用于在设定范围随机生成样本坐标。

算子参数

- **起始坐标/start_pose**：坐标起点。
- **终止坐标/end_pose**：坐标终点。
- **随机采样数量/random_sample_number**：随机生成的坐标数量。默认值：1 。
- **坐标列表/pose_list**：设置随机生成的坐标 3D 视图中的可视化属性。
  - 打开样本坐标可视化。
  - 关闭样本坐标可视化。
  - 设置样本坐标的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据型号输入输出

输入：

- **pose**：
  - 数据类型：Pose
  - 输入内容：pose 数据

输出：

- `pose_list`：
  - 数据类型：PoseList
  - 输出内容：样本坐标列表数据

功能演示

使用 GenerateSamplePose算子中 BoxRandomSample 根据加载的坐标随机生成2个样本坐标。

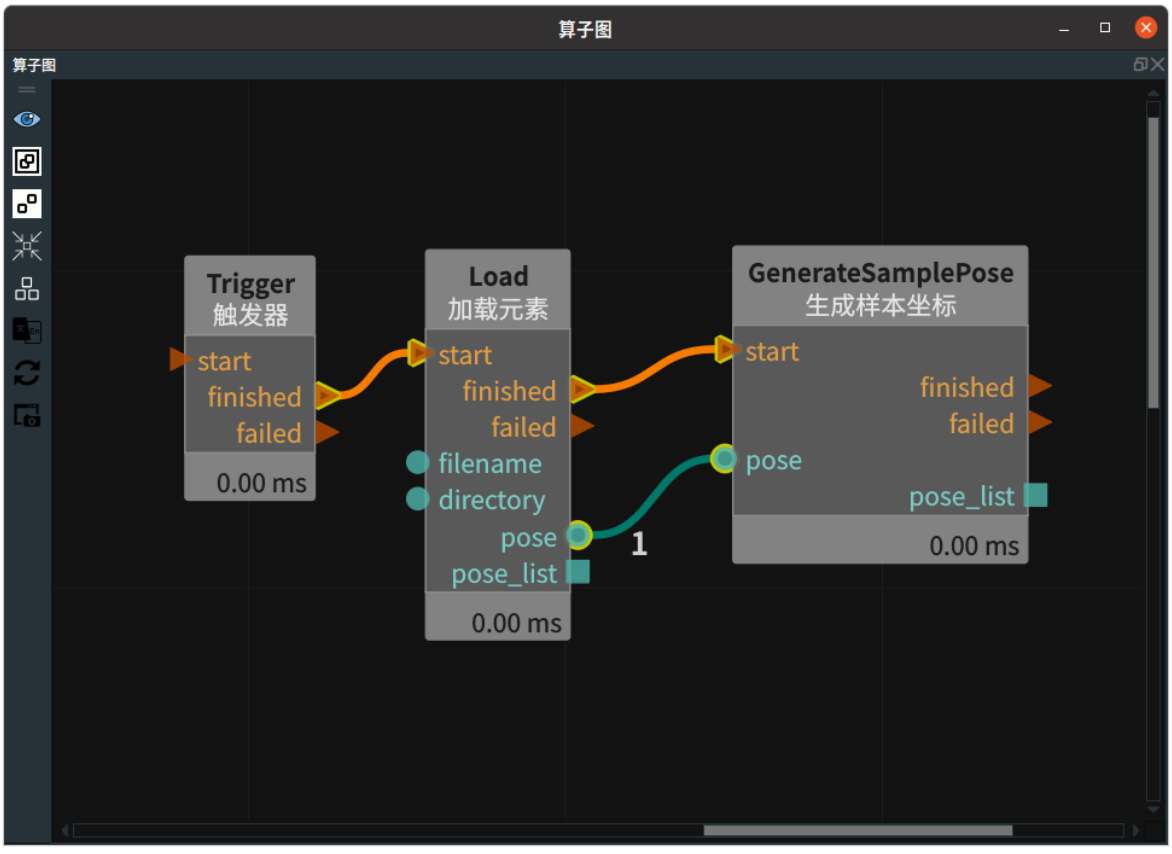
步骤1：算子准备

添加 Trigger 、Load 、GenerateSamplePose 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → pose
   - 文件 → ••• → 选择 pose 坐标文件名(*example_data/pose/tcp2.txt*)
   - 坐标 → 可视 → 0.2
2. 设置 GenerateSamplePose 算子参数：
   - 类型 → BoxRandomSample
   - 起始坐标 → -0.3 -0.2 -0.15 0 0 0
   - 终止坐标 → 0.3 0.2 0.15 1 2 3
   - 随机采样坐标 → 2
   - 坐标列表 → 可视

步骤3：连接算子

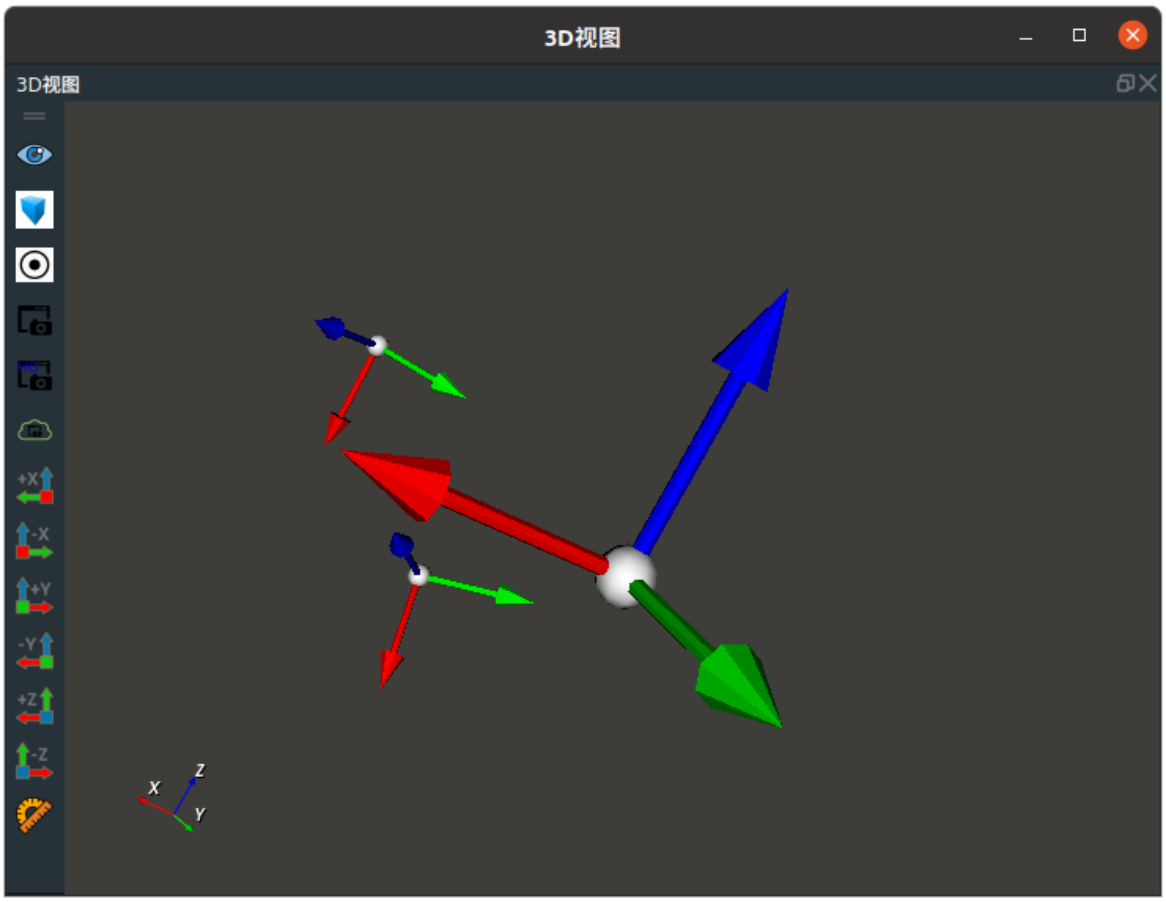


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

结果如下图所示，在 3D 视图中显示加载的坐标和随机生成的2个样本坐标。



SphereRandomSample

将 GenerateSamplePose 算子的 **类型** 设置为 SphereRandomSample ，在球体表面随机生成样本坐标。。

算子参数

- **初始方向数量/init_direction_number**：生成样本坐标的数量。默认值：6 。表示在球体表面随机生成6个坐标。
- **半径/radius**：球体半径。默认值：1 ，单位：m 。
- **只有半球/only_half_sphere**：生成的坐标的位置。
  - False :生成的坐标仅 Z 轴正方向的全球
  - True：生成的坐标仅 Z 轴正方向的半球。
- **坐标列表/pose_list**：设置球体随机生成的样本坐标在 3D 视图中的可视化属性。
  - 打开样本坐标可视化。
  - 关闭样本坐标可视化。
  - 设置样本坐标的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

- **pose**：
  - 数据类型：Pose

- 输入内容：pose 数据

输出：

- `pose_list`：
  - 数据类型：PoseList
  - 输出内容：样本坐标列表数据

功能演示

使用 GenerateSamplePose 中 SphereRandomSample 在生成的球体表面生成6个样本坐标。

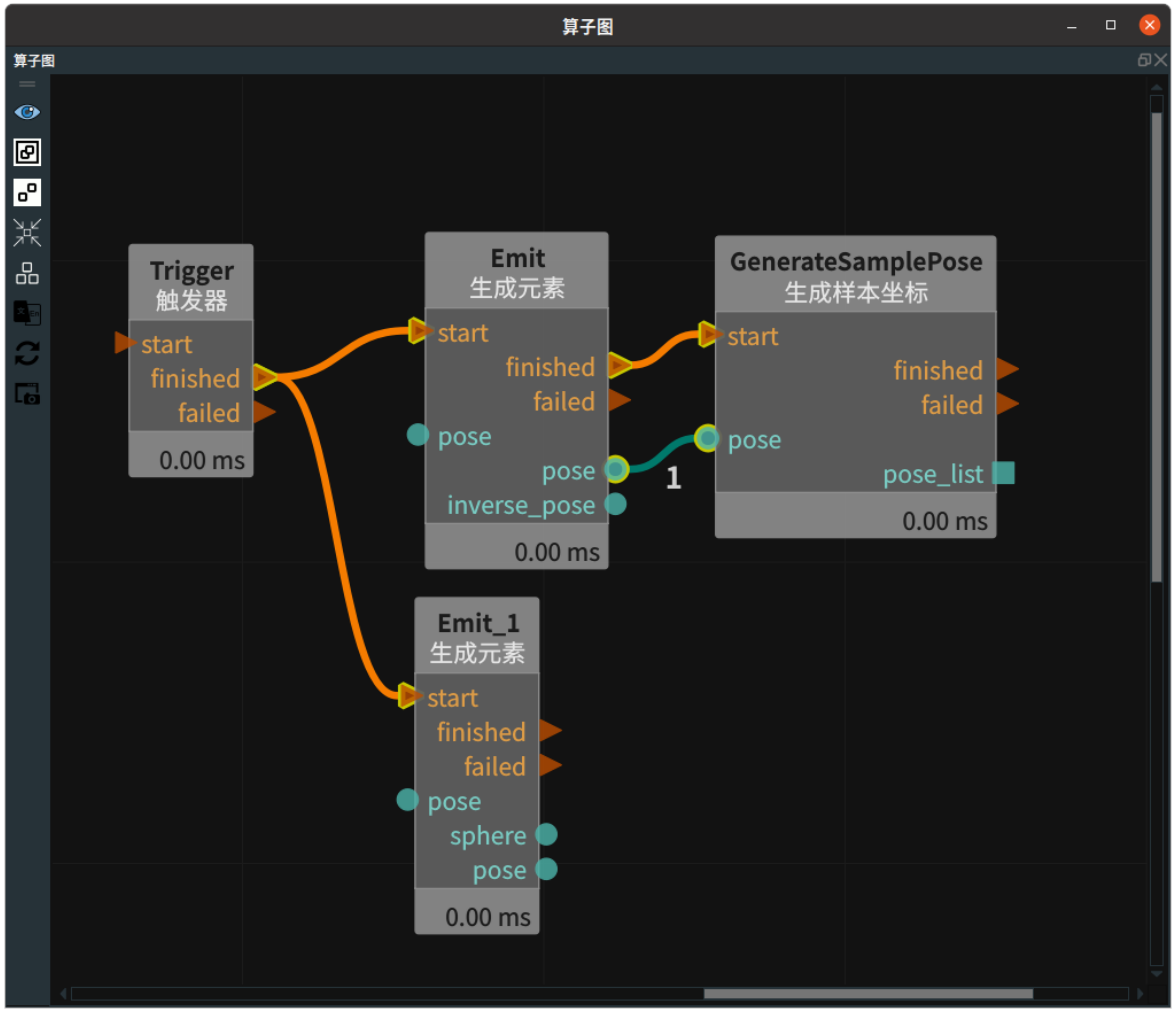
步骤1：算子准备

添加 Trigger 、Emit（2个） 、GenerateSamplePose 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
   - 类型 → Pose
   - 坐标 →  可视
2. 设置 Emit_1 算子参数：
   - 类型 → Sphere
   - 半径 → 1
   - 球体 →  可视 →  0.5
3. 设置GenerateSamplePose算子参数：
   - 类型 → SphereRandomSample
   - 坐标列表 →  可视 →  0.2
   - 其余参数默认

步骤3：连接算子

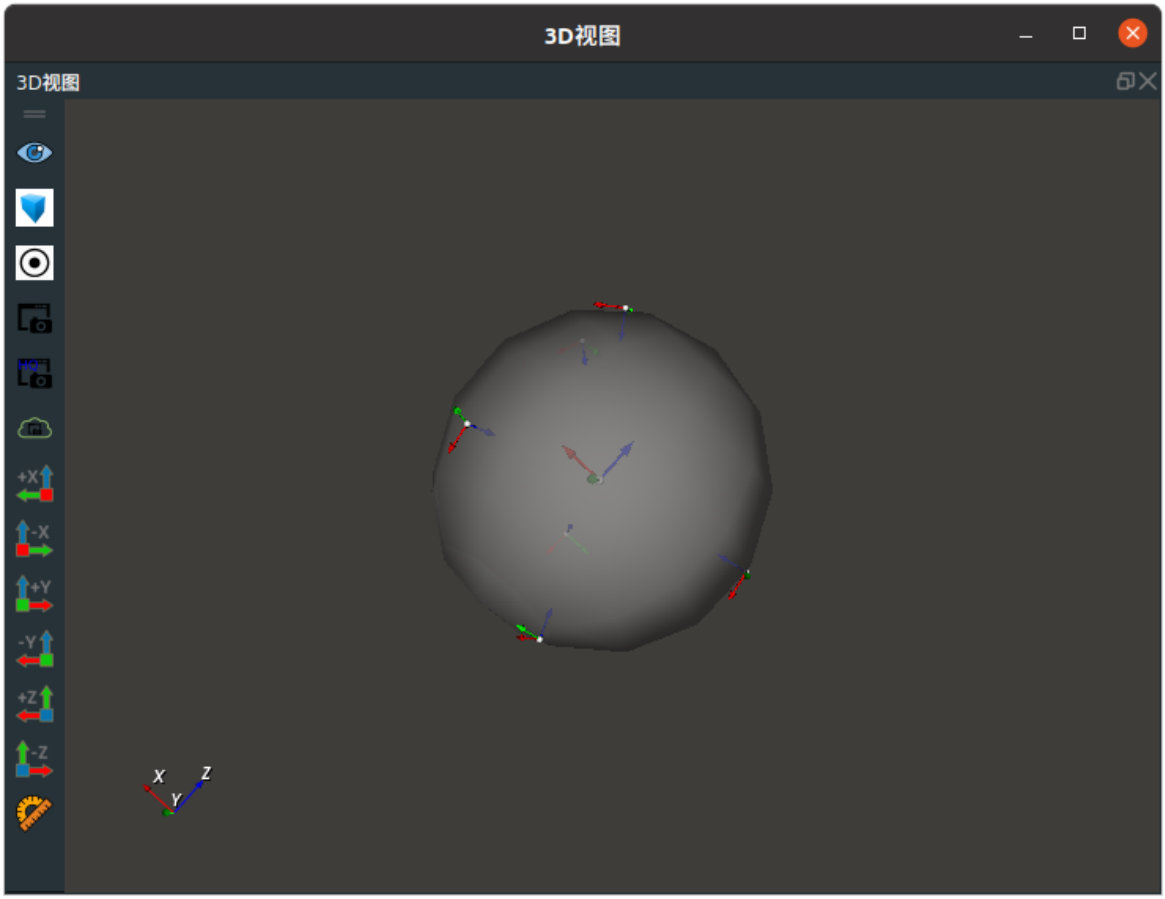


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，在 3D 视图中显示加载的球体，其表面随机生成的6个坐标样本。



CameraPoseSample

将 GenerateSamplePose 算子的 **类型** 设置为 CameraPoseSample ，用于根据棋盘的位置生成相机的多组拍照位。棋盘的坐标分别绕 XYZ 轴做旋转，然后得到的旋转后的 pose 沿其 Z 轴负方向平移 radius ，得到的 pose 就是相机拍照位。

算子参数

- **半径/radius**：相机拍照位到棋盘的距离。
- **Roll度数范围/degree_range_roll**：棋盘绕着 x 轴旋转的角度步长。
- **Pitch度数范围/degree_range_pitch**：棋盘绕着 y 轴旋转的角度步长。
- **Yaw度数范围/degree_range_yaw**：棋盘绕着 z 轴随机旋转的角度范围。
- **采样数量/num_size**：拍照位网格边长，默认值：3 。表示 3*3 的网格位置。
- **取反/inverse**：
  - True：拍照位排序为由外到内螺旋排序。
  - False：拍照位排序为由内到外螺旋排序。
- **坐标列表/pose_list**：设置生成的相机拍照位坐标列表的可视化属性。
  - 打开机拍照位坐标列表可视化。
  - 关闭机拍照位坐标列表可视化。
  - 设置机拍照位坐标列表的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

- `pose`：
  - 数据类型：Pose
  - 输入内容：棋盘坐标数据

输出：

- `pose_list`：
  - 数据类型：PoseList
  - 输出内容：相机拍照位列表数据

功能演示

将 GenerateSamplePose 算子中 CameraPoseSample 根据棋盘的位置生成相机的 9 组拍照位。

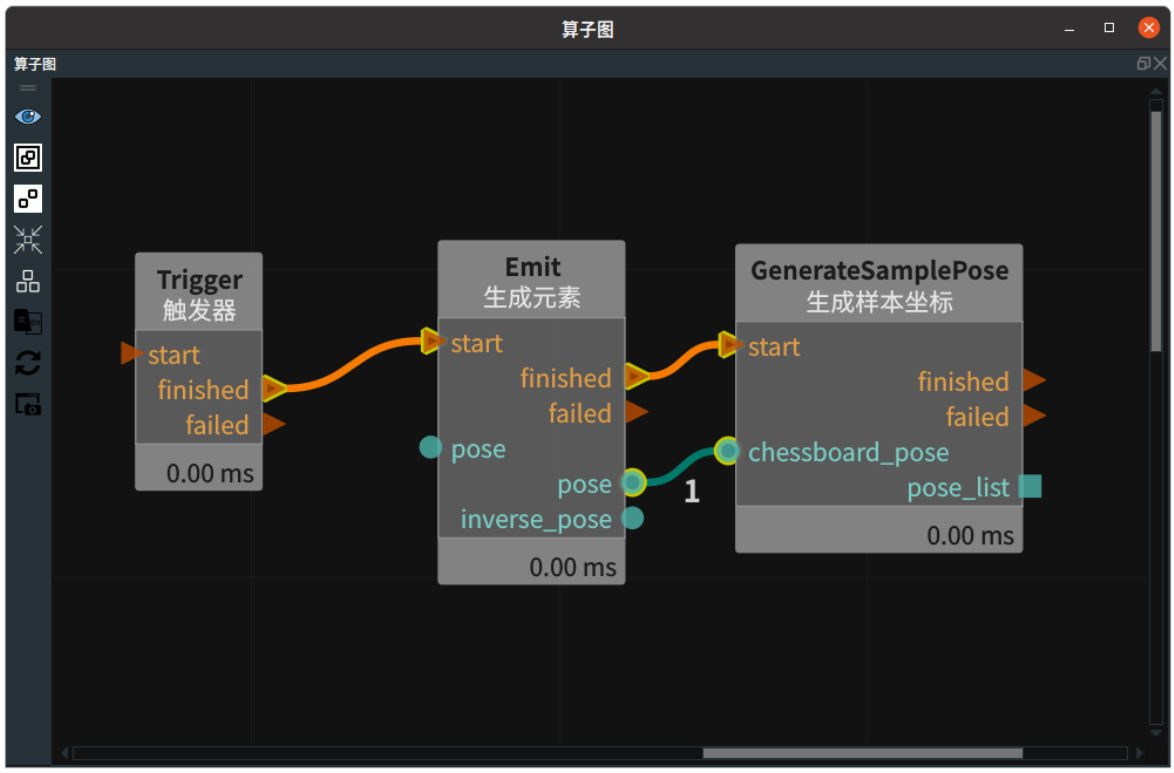
步骤1：算子准备

添加 Trigger 、Emit 、GenerateSamplePose 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
   - 类型 → pose
   - 坐标 →  可视 →  0.2
2. 设置 GenerateSamplePose 算子参数：
   - 类型 → CameraPoseSample
   - 坐标列表 →  可视

步骤3：连接算子

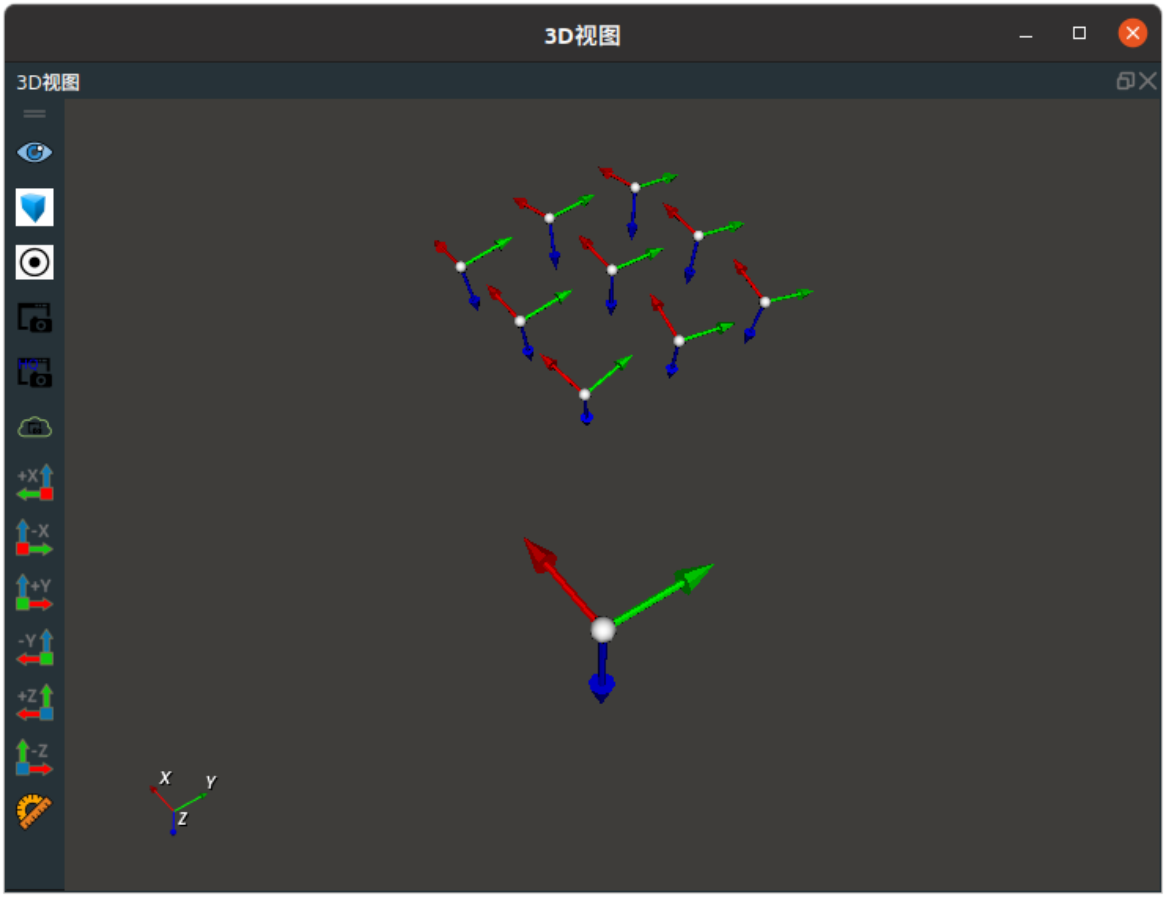


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，在 3D 视图中显示棋盘的 9 个相机拍照位。



basic

ElementToPose 元素转 pose

ElementToPose 算子用于将元素或者元素中心点输出为 pose 或者 pose_list 。使用元素类型有：Cube 、Cylinder 、Path 。

类型	功能
Cube	将 cube 或者 cube_list 中心点输出格式为 pose 或者 pose_list 。cube 的 Width/Height/Depth 作为 pose 的 X/Y/Z 轴方向。
Cylinder	将 cylinder 或者 cylinder_list 中心点输出格式为 pose 或者 pose_list 。cylinder 的 Length 作为 pose 的 Z 的方向。
Path	将 path 或者 path_list 中心点输出格式为 poselist 。

Cube

将 ElementToPose 算子 **类型** 属性选择 Cube ，用于将 cube 或者 cube_list 中心点输出格式为 pose 或者 pose_list 。cube 的Width-Height-Depth 作为 pose 的三个轴 X-Y-Z 的方向。

算子参数

- **坐标/pose**：设置 pose 在 3D 视图中的可视化属性。
  -  打开 pose 可视化。
  -  关闭 pose 可视化。
  -  设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。
- **坐标列表/pose_list**：设置 pose 列表在 3D 视图中的可视化属性。参数值描述与 **坐标** 一致。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **cube**：
  - 数据类型：Cube
  - 输入内容：立方体数据
- **cube_list**：
  - 数据类型：CubeList
  - 输入内容：立方体数据列表

输出：

- **pose**：
  - 数据类型：Pose
  - 输出内容：单个 pose 数据
- **pose_list**：

- 数据类型：PoseList
- 输出内容： pose 数据列表

功能演示

使用 ElementToPose 算子中 Cube ，将 cube 中心点输出格式为 pose 。

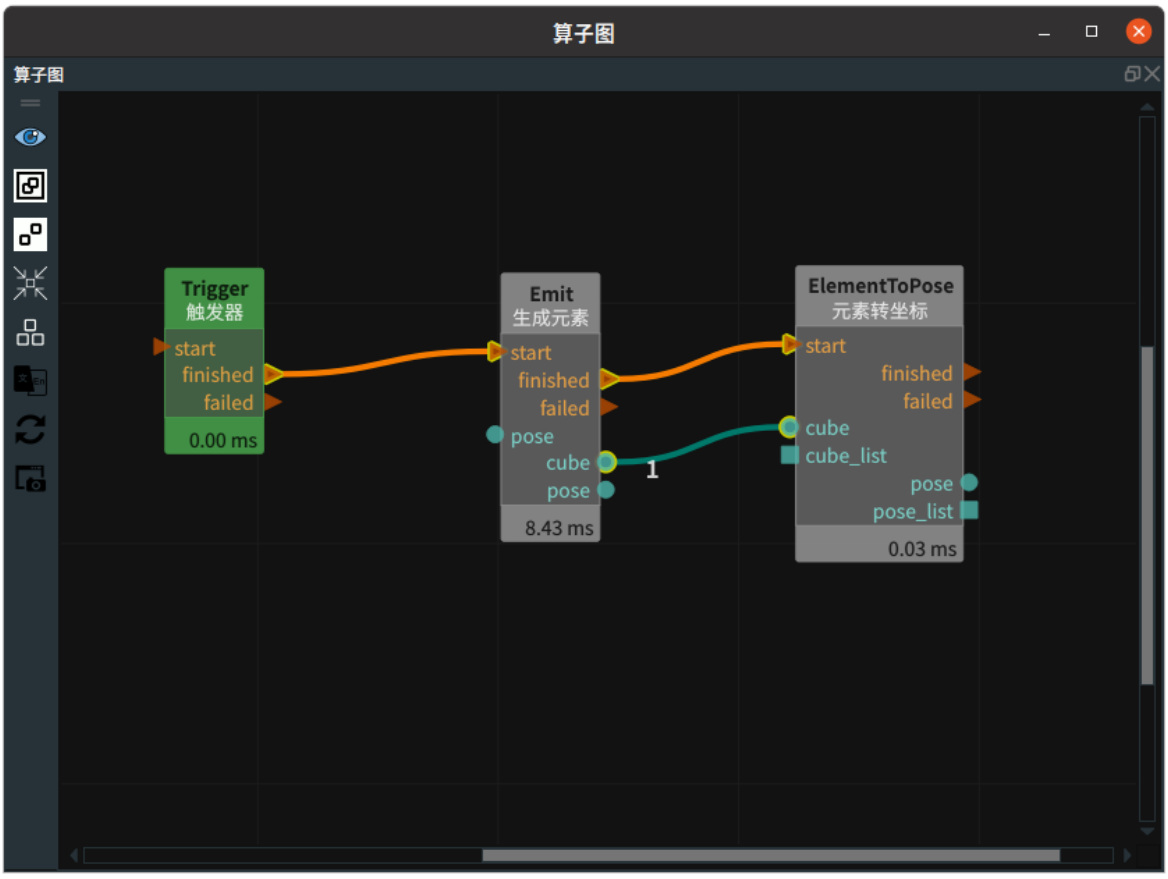
步骤1：算子准备

添加 Trigger 、Emit 、ElementToPose 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
   - 类型 → Cube
   - 坐标 → 0 0 0 0 0 0
   - 宽度 → 1
   - 高度 → 1
   - 深度 → 1
   - 立方体 → 可视
2. 设置 ElementToPose 算子参数：
   - 类型 → Cube
   - 坐标 → 可视

步骤3：连接算子

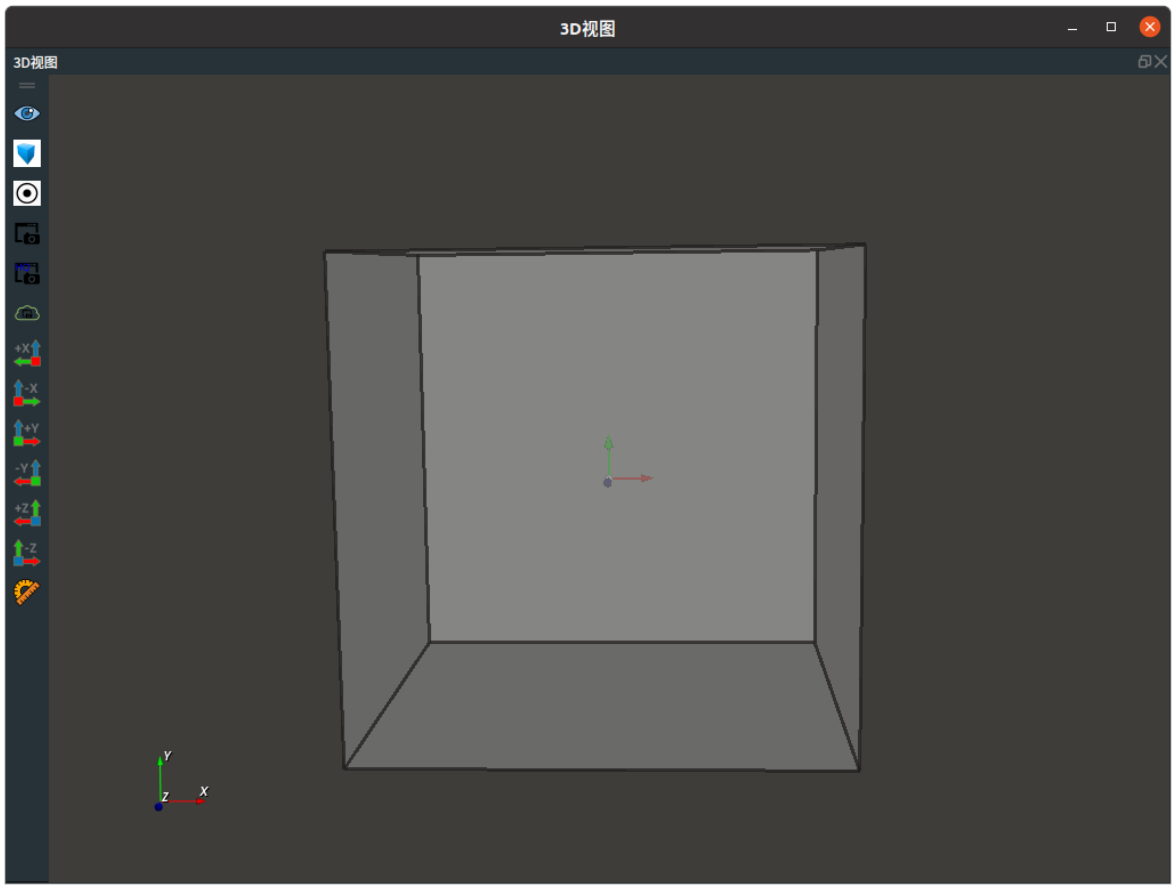


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

结果如下图所示，3D 视图中显示加载的 cube 以及中心点 pose ，pose 的 X 轴的方向为 cube 的 width ，pose 的 Y 轴的方向为 cube 的 height ，pose 的Z 轴的方向为 cube 的 depth 。



Cylinder

选择 ElementToPose 算子中 Cylinder ，用于将 cylinder 或者 cylinder_list 中心点输出格式为 pose 或者 pose_list 。Cylinder 的 Length 作为 pose 的 Z 的方向。

算子参数

- **坐标/pose**：设置 pose 在 3D 视图中的可视化属性。
  - 打开 pose 可视化。
  - 关闭 pose 可视化。
  - 设置 pose 的尺寸大小 。取值范围：[0.001,10] 。默认值：0.1 。
- **坐标列表/pose_list**：设置 pose 列表在 3D 视图中的可视化属性。参数值描述与 **坐标** 一致

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **cylinder**：
  - 数据类型：Cylinder
  - 输入内容：圆柱体数据
- **cylinder_list**：
  - 数据类型：CylinderList
  - 输入内容：圆柱体数据列表

输出：

- `pose`：
  - 数据类型：Pose
  - 输出内容：单个 pose 数据
- `pose_list`：
  - 数据类型：PoseList
  - 输出内容： pose 数据列表

功能演示

本节将使用 ElementToPose 算子中 Cylinder ，将生成的 cylinder 中心点输出格式为 pose 。这与 ElementToPose算子中 cube 属性将 cube 中心点输出格式为 pose 或者 pose_list的方法相同，请参照该章节的功能演示。

Path

将 ElementToPose 算子 `类型` 属性选择 Path ，用于将 path 或者 path_list 中的点输出格式为 poselist 。

算子参数

- `坐标列表/pose_list`：设置 pose 列表在 3D 视图中的可视化属性。
  -  打开 pose 列表可视化。
  -  关闭 pose 列表可视化。
  -  设置 pose 列表的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- `path`：
  - 数据类型：Path
  - 输入内容：Path 数据
- `path_list`：
  - 数据类型：PathList
  - 输入内容：Path 数据列表

输出：

- `pose_list`：
  - 数据类型：PoseList
  - 输出内容： pose 数据列表

功能演示

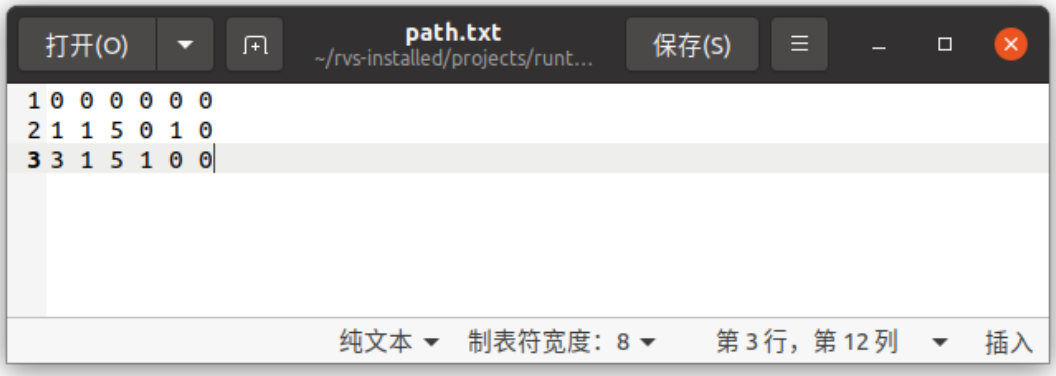
使用 ElementToPose 算子中 Path ，将 path 中的点输出为 poselist 。

步骤1：算子准备

添加 Trigger 、Load 、ElementToPose 算子至算子图。

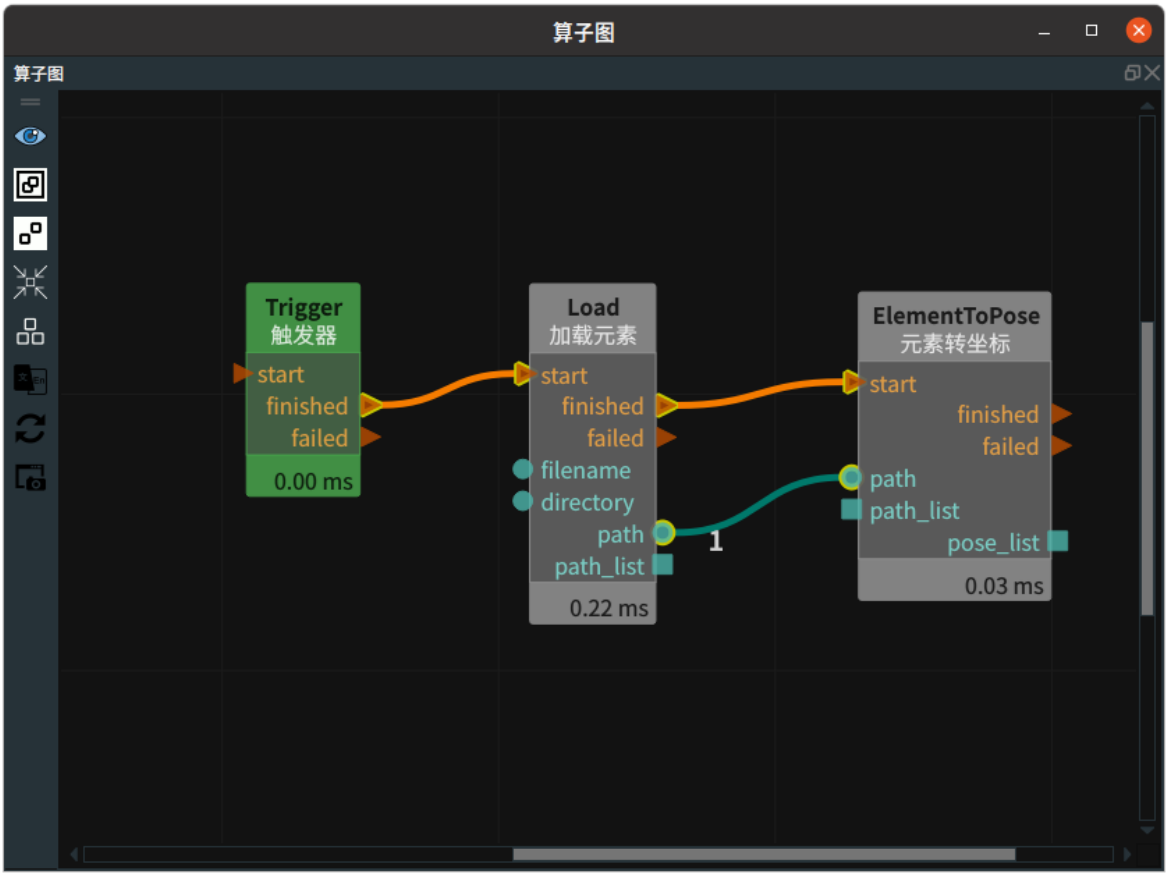
步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 类型 → Path
    - 文件 → ••• → 选择 path 文件名（ *example_data/paths/path.txt*：文件内容如下 ）



    - path → 可视
2. 设置 ElementToPose 算子参数：
    - 类型 → Path
    - 坐标列表 → 可视

步骤3：连接算子

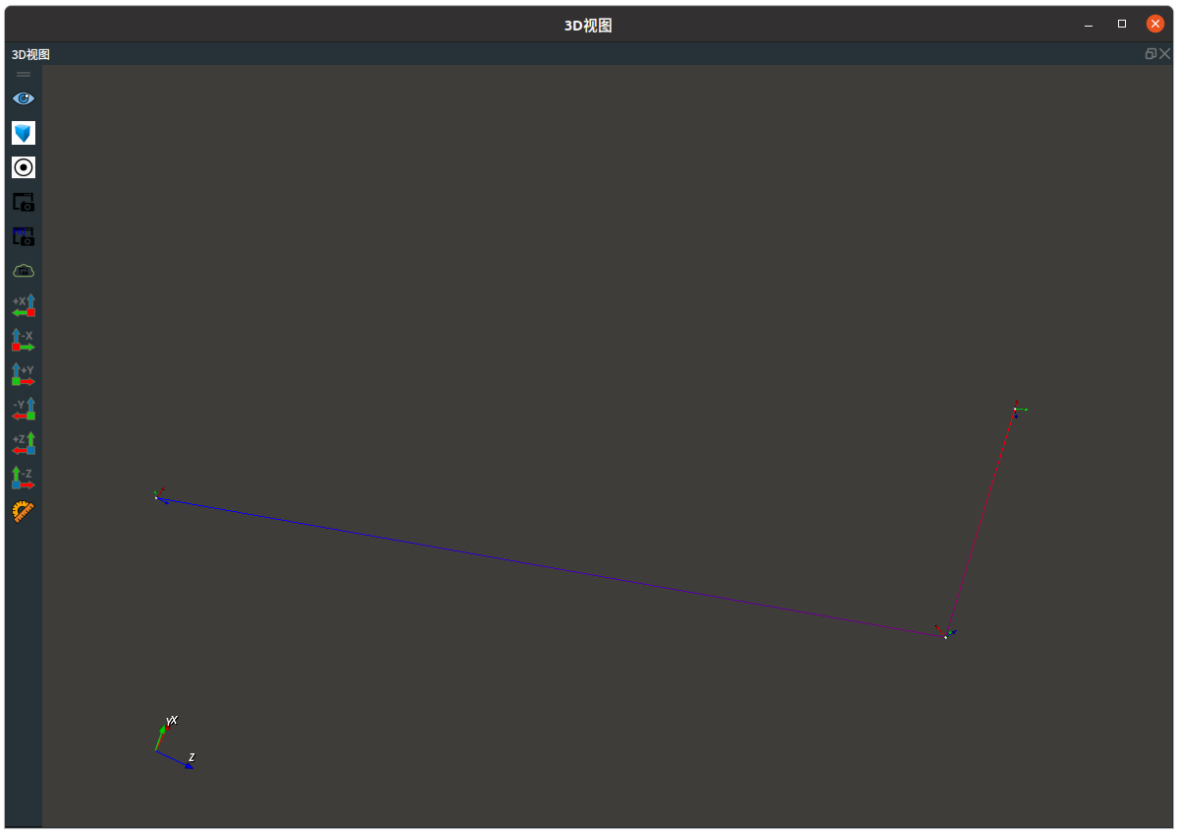


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

结果如下图所示，3D 视图中显示加载的 Path 以及转换后的点 pose 。



WaitAllTriggers 等待所有触发

WaitAllTriggers 算子用于将多条控制信号流通过逻辑与的方式并联在一起，合并后再触发后续信号。

算子参数

- **输入数量/number_input**：决定该算子的输入端口 input_? 的数量。取值范围：[1,10]。默认值：2。

控制信号输入

输入：

- **reset**：该端口被触发时，重置所有的 input_? 端口的触发状态为False。
- **input_?**：功能：当所有的 input_? 端口都被触发时，才会触发 finished 端口。

功能演示

WaitAllTriggers 算子将多条控制信号流通过逻辑与的方式并联在一起。本案例只展示连接方法。

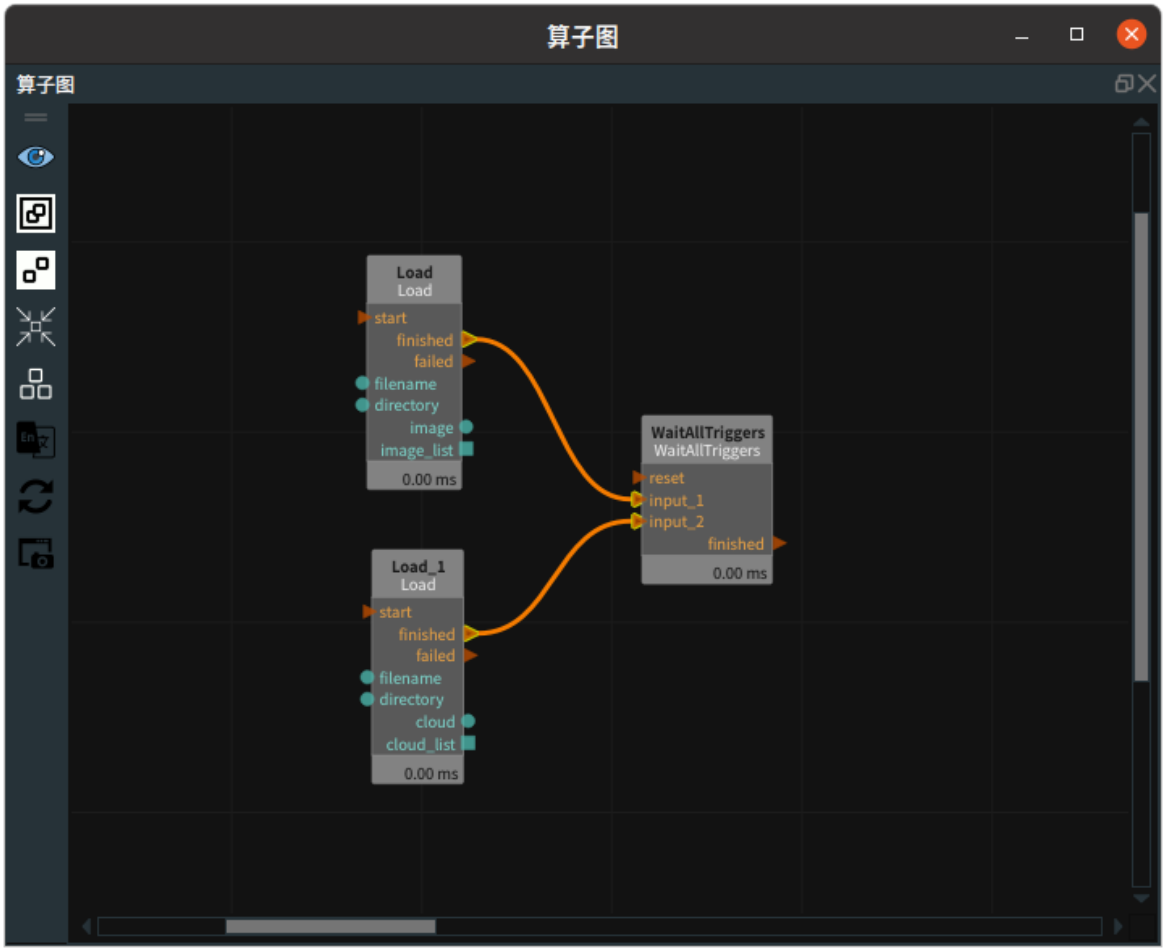
步骤1：算子准备

添加 Load 、WaitAllTrigger 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 类型 → image
2. 设置 Load_1 算子参数：
    - 类型 → pointcloud

步骤3：连接算子


算子图
算子图
Load
Load
start
finished
failed
filename
directory
image
image_list
0.00 ms
Load_1
Load
start
finished
failed
filename
directory
cloud
cloud_list
0.00 ms
WaitAllTriggers
WaitAllTriggers
reset
input_1
input_2
finished
0.00 ms

Load 加载元素

Load 算子为加载元素，用于加载单个 Cube 、Image 、ImagePoints 、JointArray 、Line 、Path 、PointCloud 、PolyData 、Pose 、Sphere 、Voxels 。

类型	功能
Cube	用于加载单个或多个立方体。
Image	用于加载单张或多张图像。
ImagePoints	用于加载单个或多个图像关键点。
JointArray	用于加载单组或多组机器人关节弧度值。
Line	用于加载单条或多条线段。
Path	用于加载单条或多条路径。
PointCloud	用于加载单个或多个点云。
PolyData	用于加载单个或多个 3D 模型。
Pose	用于加载单个或多个位姿。
Sphere	用于加载单个或多个球体。
Voxels	/

Cube

将 Load 算子的 **类型** 属性选择 Cube ，用于加载单个或多个立方体。

算子参数

- **文件/filename**：读取单个 cube 数据时，输入内容：cube 文件名。文件格式：txt 。x y z roll pitch yaw 单位：弧度。width height depth 单位：m。cube文件格式如下：

```
x y z roll pitch yaw width height depth
如：0 0 0 0 0 0 0 1 1 1
```

- **目录/directory**：读取多个 cube 数据时，输入内容：cube 文件目录名。
- **立方体/cube**：设置立方体在 3D 视图中的可视化属性。
  - 打开立方体可视化。
  - 关闭立方体可视化。
  - 设置立方体的颜色。取值范围：[-2,360] 。默认值：-2 。
  - 设置立方体的透明度。取值范围：[0,1] 。默认值：0.5 。
- **立方体列表/cube_list**：设置立方体列表在 3D 视图中的可视化属性。参数值描述与 **立方体** 一致。

数据信号输入输出

输入：

- `filename`：
  - 数据类型：String
  - 输入内容：cube 文件名
- `directory`：
  - 数据类型：String
  - 输入内容：cube 文件目录名

输出：

- `cube`：
  - 数据类型：Cube
  - 输出内容：单个 cube 数据
- `cube_list`：
  - 数据类型：CubeList
  - 输出内容：cube 数据列表

功能演示

使用 Load 算子中 Cube ，加载单个立方体。

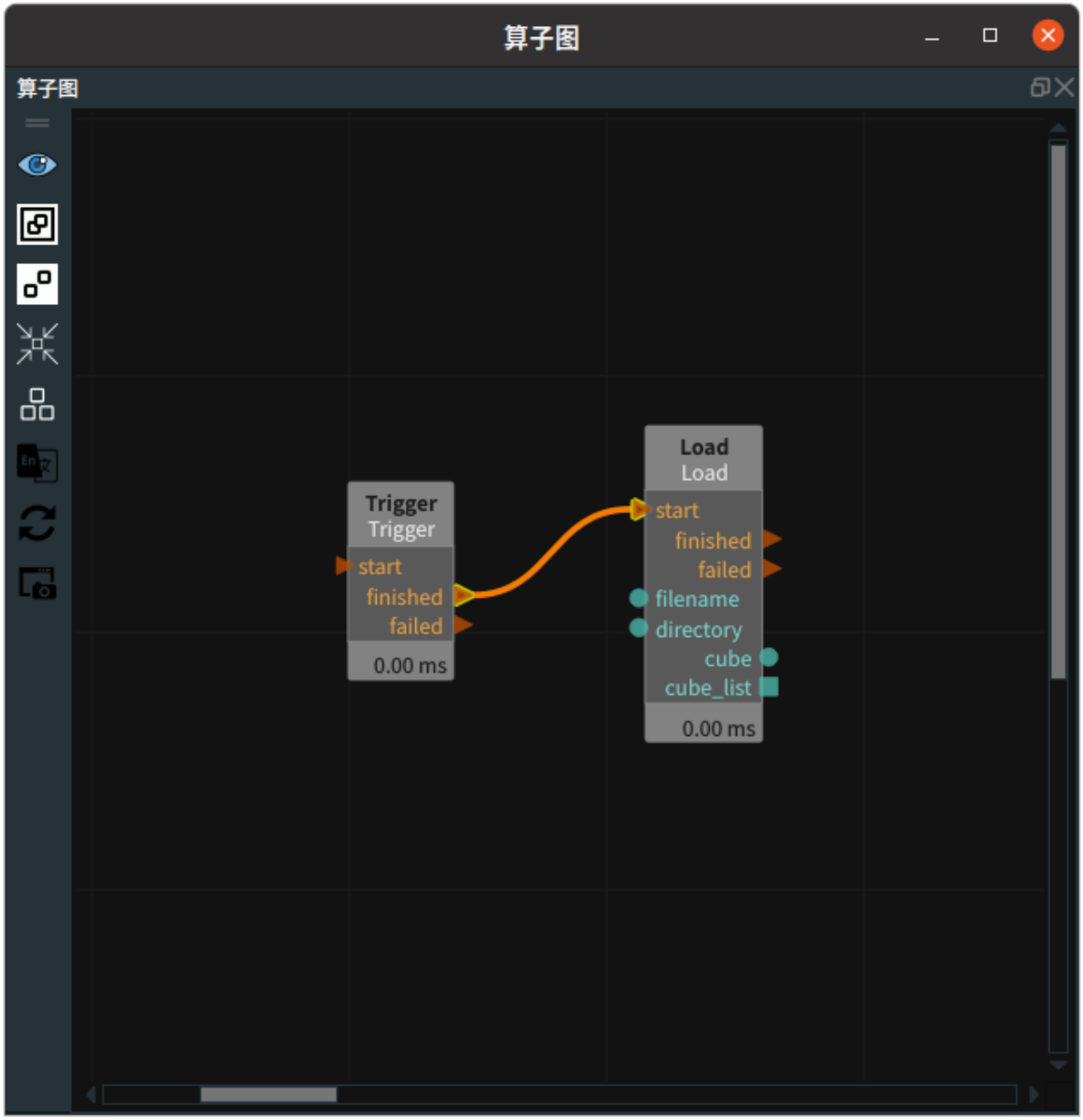
步骤1：算子准备

添加 Trigger 、Load 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → Cube
   - 文件 → ••• → 选择 cube 文件名（*example_data/cube/cube.txt*）
   - 立方体 →  可视

步骤3：连接算子

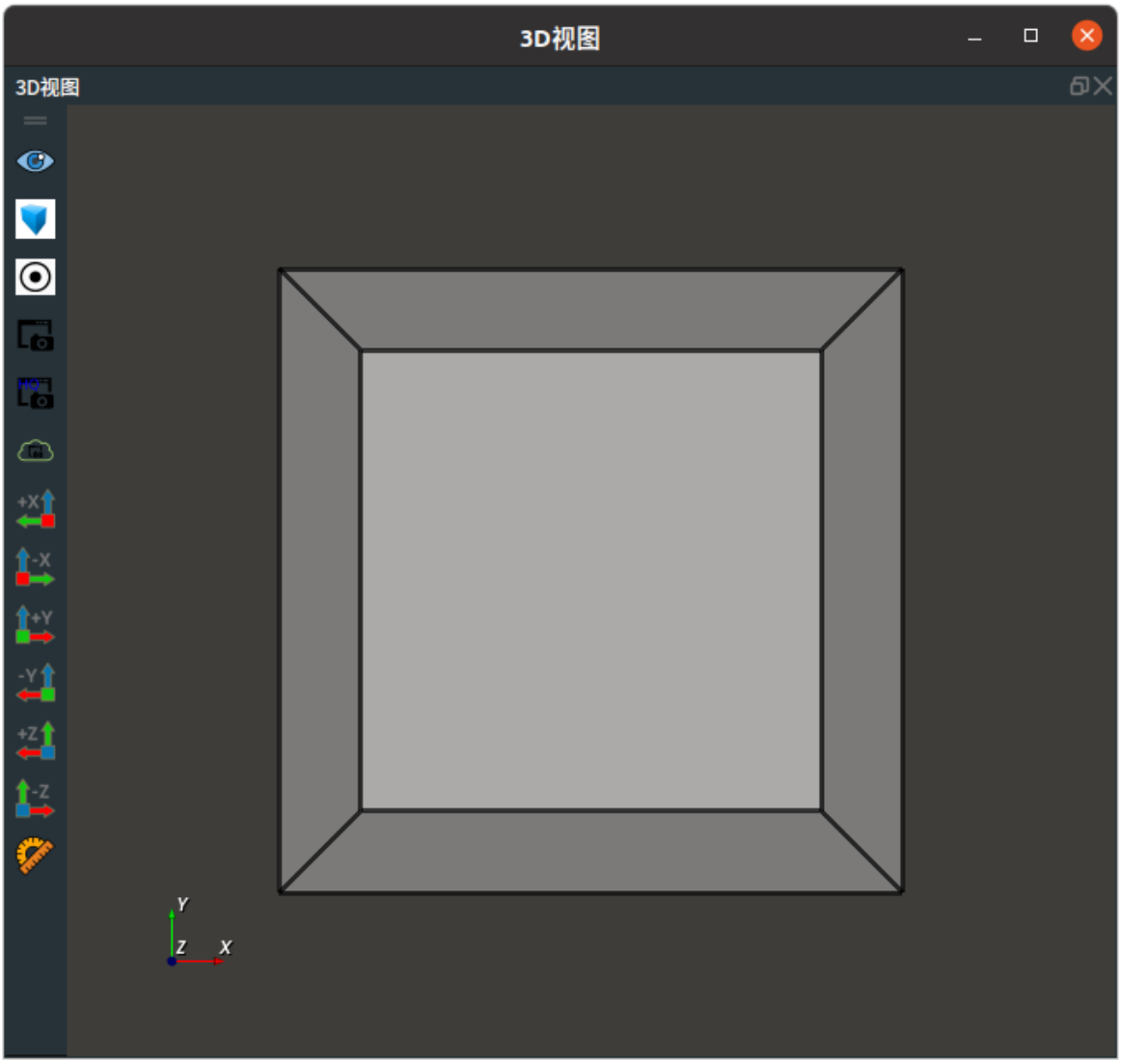


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，在 3D 视图中显示加载的 cube 。



Image

将 Load 算子的 **类型** 属性选择 Image ，用于加载单张或多张图像。

算子参数

- **文件/filename**：读取单张图像时，输入内容：图像文件名。文件格式：png。
- **目录/directory**：读取多张图像时，输入内容：图像文件目录名。
- **图像/image**：设置图像在 2D 视图中的可视化属性。
  - 打开图像可视化。
  - 关闭图像可视化。
- **图像列表/image_list**：设置图像列表在 2D 视图中的可视化属性。参数值描述与 **图像** 一致。

数据信号输入输出

输入：

- **filename**：
  - 数据类型：String
  - 输入内容：图像文件名
- **directory**：

- 数据类型：String
- 输入内容：图像文件目录名

输出：

- `image`：
  - 数据类型：Image
  - 输出内容：单张图像数据
- `image_list`：
  - 数据类型：ImageList
  - 输出内容：图像数据列表

功能演示

本节将使用 Load 算子中 Image ，加载单张图像。这与 Load 算子中 Cube 属性加载单个立方体方法相同，请参照该章节的功能演示。

ImagePoints

将 Load 算子的 `类型` 属性选择 ImagePoints ，用于加载单个或多个图像关键点。

算子参数

- `文件/filename`：读取图像关键点坐标时，输入内容：图像关键点坐标文件名。文件格式：txt 。图像关键点坐标格式如下：

```
x1 y1
如： 1 2
```

- `目录/directory`：读取多个图像关键点时，输入内容：图像关键点坐标文件目录名。

数据信号输入输出

输入：

- `filename`：
  - 数据类型：String
  - 输入内容：图像关键点文件名
- `directory`：
  - 数据类型：String
  - 输入内容：图像关键点文件目录名

输出：

- `image_points`：
  - 数据类型：ImagePoints
  - 输出内容：单个图像关键点坐标数据
- `image_points_list`：
  - 数据类型：ImagePointsList
  - 输出内容：图像关键点数据坐标列表

功能演示

本节将使用 Load 算子中 ImagePoints ，加载单个图像关键点。这与 Load 算子中 Cube 属性加载单个立方体方法相同，请参照该章节的功能演示。

JointArray

将 Load 算子的 **类型** 属性选择 JointArray ，用于加载单组或多组机器人关节弧度值。

算子参数

- **文件/filename**：读取单张图像时，输入内容：jointarray 文件名。文件格式：txt 。文件内容如下：

```
J0 j1 j2 j3 j4 j5
如： 1 1 1 1 1 1
```

- **目录/directory**：读取多张图像时，输入内容：jointarray 文件目录名。

数据信号输入输出

输入：

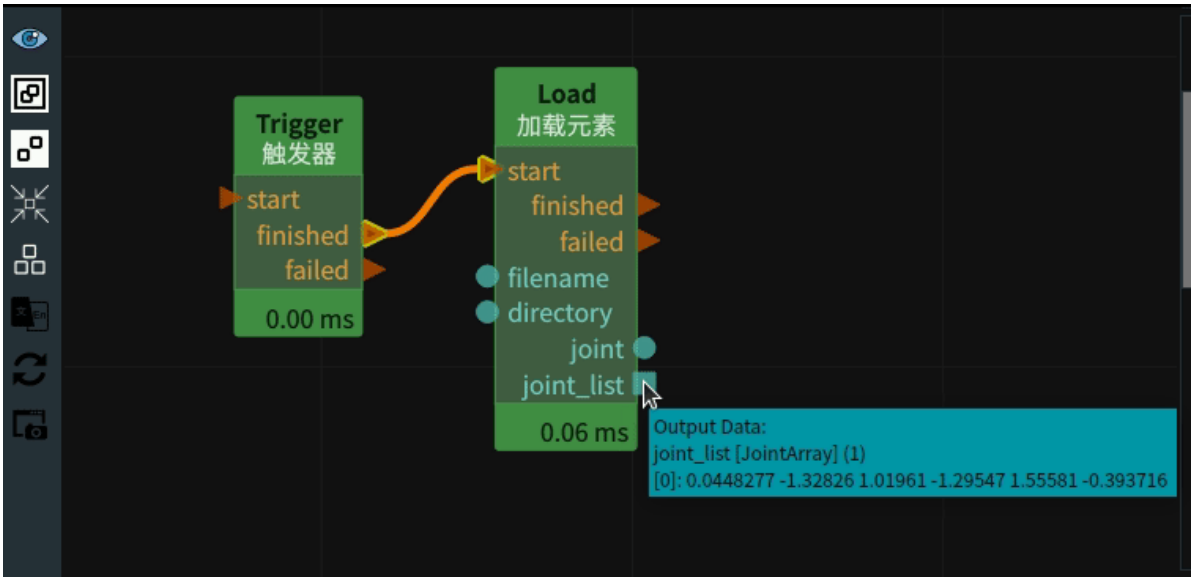
- **filename**：
  - 数据类型：String
  - 输入内容：jointarray 文件名
- **directory**：
  - 数据类型：String
  - 输入内容：jointarray 文件目录名

输出：

- **joint**：
  - 数据类型：JointArray
  - 输出内容：单组 jointarray 数据
- **joint_list**：
  - 数据类型：JointArrayList
  - 输出内容：jointarray 数据列表

功能演示

使用 Load 算子中 JointArray ，加载单组机器人关节弧度值。这与 Load 算子中 Cube 属性加载单个立方体方法相同，请参照该章节的功能演示。



Line

将 Load 算子的 **类型** 属性选择 Line ，用于加载单条或多条线段。

算子参数

- **文件/filename**：读取单条线时，输入内容：Line 文件名。文件格式：txt 。文件格式内容如下：

```
x y z roll pitch yaw
x y z roll pitch yaw
如：
0 0 0 0 0 0
1 1 5 0 1 0
```

- **目录/directory**：读取多条线时，输入内容：Line 文件目录名。
- **线段/line**：设置线段在 3D 视图中的可视化属性。
  - 打开线段可视化。
  - 关闭线段可视化。
  - 设置线条的颜色。取值范围：[-2,360] 。默认值：60 。
  - 设置线条的线宽。取值范围：[1,100] 。默认值：1 。
- **线段列表/line_list**：设置线段列表在 3D 视图中的可视化属性。参数值描述与 **线段** 一致。

数据信号输入输出

输入：

- **filename**：
  - 数据类型：String
  - 输入内容：line 文件名
- **directory**：
  - 数据类型：String
  - 输入内容：line 文件目录名

输出：

- **line**：

    - 数据类型：Line
    - 输出内容：单条线数据
- `line_list`：
    - 数据类型：LineList
    - 输出内容：线列表数据

功能演示

将 Load 算子的 type 属性选择 Line ，加载单条线段。这与 Load 算子中 Cube 属性加载单个立方体的方法相同，请参照该章节的功能演示。

Path

将 Load 算子的 `类型` 属性选择 Path ，用于加载单条或多条路径。

算子参数

- `文件/filename`：读取单条路径时，输入内容：path 文件名。文件格式：txt 。文件格式如下：

```
x y z roll pitch yaw
x y z roll pitch yaw
如：
0 0 0 0 0 0
1 1 5 0 1 0
3 1 5 1 0 0
```

- `目录/directory`：读取多条路径时，输入内容：path 文件目录名。
- `path`：设置路径在 3D 视图中的可视化属性。
    -  打开路径可视化。
    -  关闭路径可视化。
    -  设置路径的线宽。默认值：1 。
- `path_list`：设置路径列表在 3D 视图中的可视化属性。参数值描述与 `path` 一致。

数据信号输入输出

输入：

- `filename`：
    - 数据类型：String
    - 输入内容：path 文件名
- `directory`：
    - 数据类型：String
    - 输入内容：path 文件目录名

输出：

- `path`：
    - 数据类型：Path
    - 输出内容：单条路径数据
- `path_list`：

- 数据类型：PathList
- 输出内容：路径数据列表

功能演示

本节将使用 Load 算子中 Path ，加载单条路径。这与 Load 算子中 Cube 属性加载单个立方体方法相同，请参照该章节的功能演示。

PointCloud

将 Load 算子的 **类型** 属性选择 PointCloud ，用于加载单个或多个点云。

算子参数

- **文件/filename**：读取单张点云时，输入内容：点云文件名。文件格式：pcd。
- **目录/directory**：读取多张点云时，输入内容：点云文件目录名。
- **比例/scale**：调整点云的大小。默认值：1 。
- **点云/cloud**：设置点云在 3D 视图中的可视化属性。
  -  打开点云可视化。
  -  关闭点云可视化。
  -  设置3D视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  -  设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。
- **点云列表/cloud_list**：设置点云列表在 3D 视图中的可视化属性。参数值描述与 **点云** 一致。

数据信号输入输出

输入：

- **filename**：
  - 数据类型：String
  - 输入内容：点云文件名
- **directory**：
  - 数据类型：String
  - 输入内容：点云文件目录名

输出：

- **cloud**：
  - 数据类型：Cloud
  - 输出内容：单张点云数据
- **cloud_list**：
  - 数据类型：CloudList
  - 输出内容：点云数据列表

功能演示

本节将使用 Load 算子中 PointCloud ，加载单个点云。这与 Load 算子中 Cube 属性加载单个立方体的方法相同，请参照该章节的功能演示。

PolyData

将 Load 算子的 **类型** 属性选择 PolyData ，用于加载单个或多个 3D 模型。

算子参数

- **文件/filename**：读取单个 3D 模型时，输入内容：polydata 文件名。支持的后缀格式：ply、stl、obj、 gltf、glb 。
- **目录/directory**：读取多个 3D 模型时，输入内容：polydata 文件目录名。
- **多边形/polydata**：设置 3D 模型在 3D 视图中的可视化属性。
  -  打开 3D 模型可视化。
  -  关闭 3D 模型可视化。
- **点云/cloud**：设置 3D 模型点云在 3D 视图中的可视化属性。
  -  打开点云可视化。
  -  关闭点云可视化。
  -  设置 3D 视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  -  设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。

数据信号输入输出

输入：

- **filename**：
  - 数据类型：String
  - 输入内容：polydata 文件名
- **directory**：
  - 数据类型：String
  - 输入内容：polydata 文件目录名

输出：

- **surface**：
  - 数据类型：PolyData
  - 输出内容： 3D 模型数据
- **cloud**：
  - 数据类型：PointCloud
  - 输出内容：3D 模型点云数据

功能演示

本节将使用 Load 算子的 type 属性选择 PolyData ，用于加载单个 3D 模型。这与 Load 算子中 Cube 属性加载单个立方体的方法相同，请参照该章节的功能演示。

Pose

将 Load 算子的 **类型** 属性选择 Pose ，用于加载单个或多个位姿。

算子参数

- **文件/filename**：读取单个 pose 时，输入内容：pose 文件名。文件格式：txt。文件内容示例如下：

```
x y z roll pitch yaw
如：
0 0 0 0 0 0
```

- **目录/directory**：读取多个 pose 时，输入内容：pose 文件目录名。
- **坐标/pose**：设置 pose 在 3D 视图中的可视化属性。
  -  打开 pose 可视化。
  -  关闭 pose 可视化。
  -  设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。
- **坐标列表/pose_list**：设置 poselist 在 3D 视图中的可视化属性。参数值描述与 **坐标** 一致。

数据信号输入输出

输入：

- **filename**：
  - 数据类型：String
  - 输入内容：pose 文件名
- **directory**：
  - 数据类型：String
  - 输入内容：pose 文件目录名

输出：

- **pose**：
  - 数据类型：Pose
  - 输出内容：单个 pose 数据
- **pose_list**：
  - 数据类型：PoseList
  - 输出内容：pose 数据列表

功能演示

本节将使用 Load 算子中 Pose ，加载单个 pose 。这与 Load 算子中 Cube 属性加载单个立方体方法相同，请参照该章节的功能演示。

Sphere

将 Load 算子的 **类型** 属性选择 Sphere ，用于加载单个或多个球体。

算子参数

- **文件/filename**：读取单个球形时，输入内容：sphere 文件名。文件格式：txt。文件格式如下：

```
x y z roll pitch yaw radius
如：
0 0 0 0 0 0 0.75
```

- **目录/directory**：读取多个球形时，输入内容：“ sphere 文件目录名”。
- **球体/sphere**：设置球体在 3D 视图中的可视化属性。
  -  打开球体可视化。
  -  关闭球体可视化。
  -  设置球体的颜色。取值范围：[-2,360] 。默认值：-2 。
  -  设置球体的透明度。取值范围：[0,1] 。默认值：0.8 。
- **球体列表/sphere_list**：设置球体列表在 3D 视图中的可视化属性。参数值描述与 **球体** 一致。

数据信号输入输出

输入：

- **filename**：
  - 数据类型：String
  - 输入内容：sphere 文件名
- **directory**：
  - 数据类型：String
  - 输入内容：sphere 文件目录名

输出：

- **sphere**：
  - 数据类型：Sphere
  - 输出内容：单个球形数据
- **sphere_list**：
  - 数据类型：SphereList
  - 输出内容：球形数据列表

功能演示

本节将使用 Load 算子中 Sphere ，加载单个球体。这与 Load 算子中 Cube 属性加载单个立方体的方法相同，请参照该章节的功能演示。

Save 保存元素

Save 算子为保存元素，用于保存 Cube 、Image 、ImagePoints 、JointArray 、Path 、PointCloud 、PolyData 、Pose 、String 。

类型	功能
Cube	用于保存立方体。
Image	用于保存图像。
ImagePoints	用于保存图像关键点坐标。
JointArray	用于保存机器人关节弧度值。
Path	用于保存路径。
PointCloud	用于保存点云。
PolyData	用于保存多边形。
Pose	用于保存位姿。
String	用于保存字符串

Cube

将 Save 算子的 **类型** 属性选择 Cube ，用于保存立方体。

算子参数

- **文件/filename**：保存立方体的文件名，如 *cube.txt* 。

数据信号输入输出

输入：

说明：根据需求选择 cube 或者 cube_list 其中一种信号输入即可。

- **filename**：
  - 数据类型：String
  - 输入内容：保存立方体的文件名
- **cube**：
  - 数据类型：Cube
  - 输入内容：立方体数据
- **cube_list**：
  - 数据类型：CubeList
  - 输入内容：立方体列表数据

功能演示

使用 Save 算子中 Cube ，保存立方体。

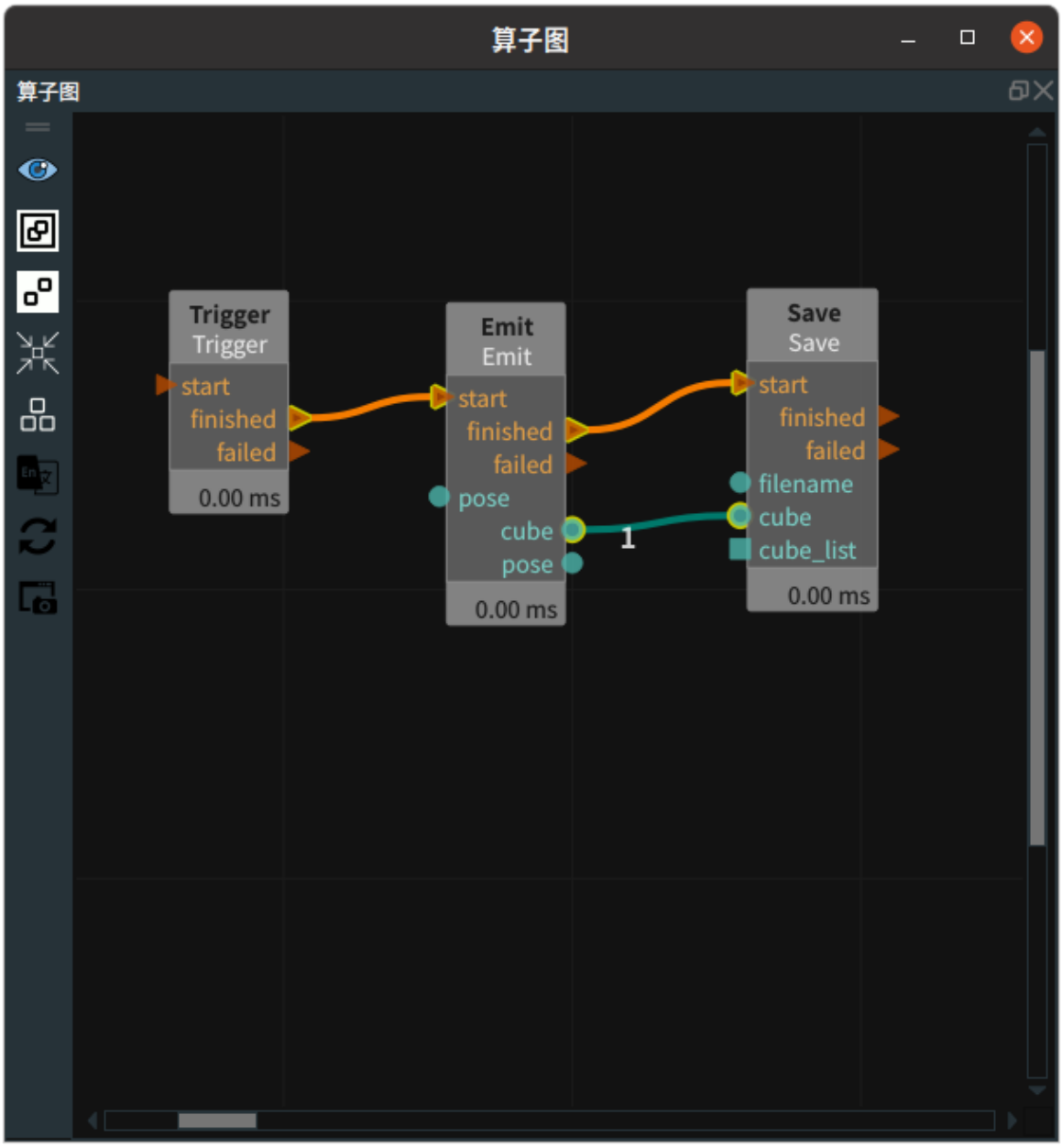
步骤1：算子准备

添加 Trigger 、Emit 、Save 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
    - 类型 → Cube
    - 坐标 → 0 0 0 0 0 0
    - 宽度 → 1
    - 高度 → 1
    - 深度 → 1
2. 设置 Save 算子参数：
    - 类型 → Cube
    - 文件→ cube.txt

步骤3：连接算子

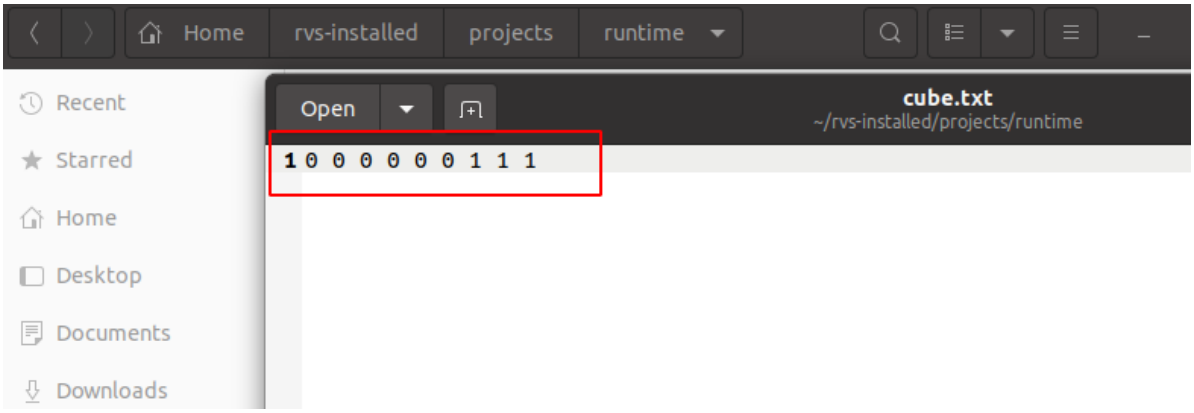


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，运行算子结束后，可以在runtime/下看到保存的cube.txt。



Image

将 Save 算子的 **类型** 属性选择 Image ，用于保存图像。

算子参数

- **文件/filename**：保存 Image 的文件名。如 *image.png* 。

数据信号输入输出

输入：

说明：根据需求选择 image 或者 image_list 其中一种信号输入即可。

- **filename**：
  - 数据类型：String
  - 输入内容：保存图像的文件名
- **image**：
  - 数据类型：Image
  - 输入内容：图像数据
- **image_list**：
  - 数据类型：ImageList
  - 输入内容：图像列表数据

功能演示

将 Save 算子中 Image ，保存图像。

步骤1：算子准备

添加 TyCameraResource、TyCameraAccess、Trigger、Save 算子至算子图。

说明：本章节使用到图漾相机资源，如学习此案例时没有相机可以使用 Load 算子完成案例。

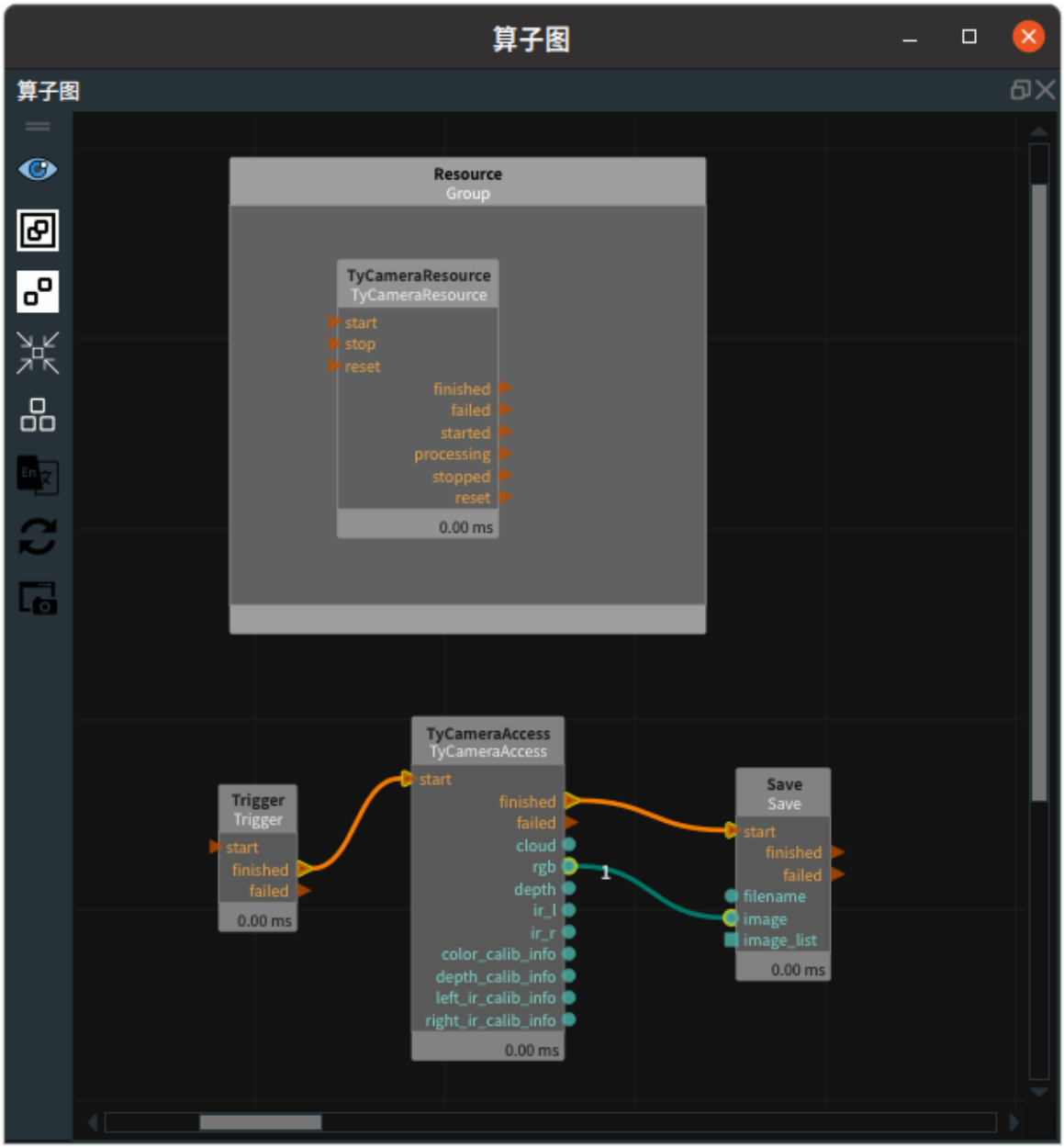
步骤2：设置算子参数

1. 设置 Save 算子参数：

- 类型 → Image
- 文件→ rgb.png

2. 设置 TycameraAccess 算子参数：
   - 彩色→ 可视

步骤3：连接算子

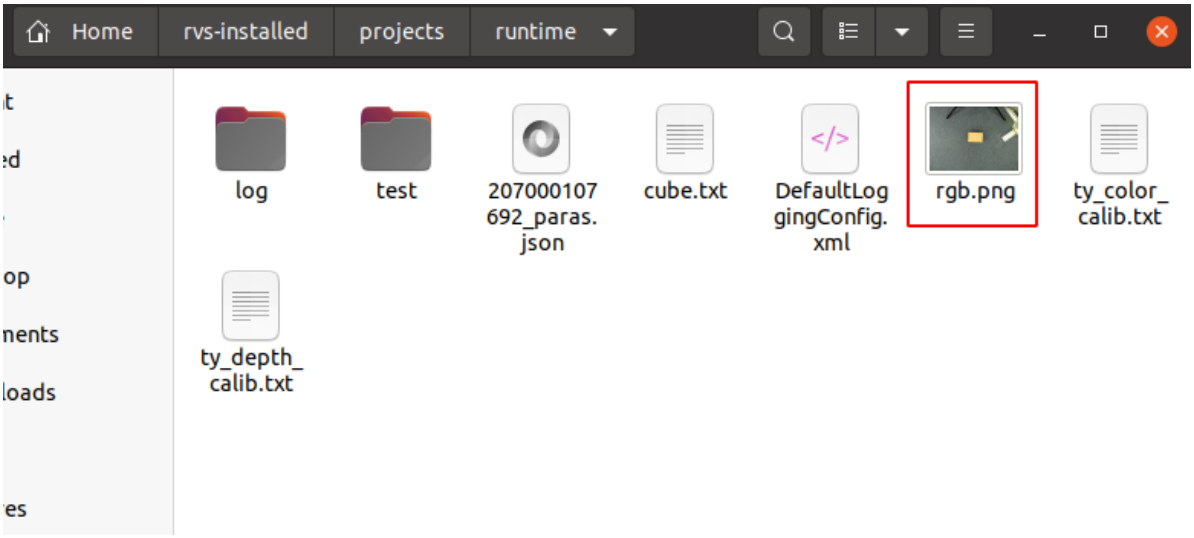


步骤4：运行

1. 点击 RVS 运行按钮。
2. 连接 TyCameraResource 相机资源。
3. 触发 Trigger 算子。

运行结果

如下图所示，运行算子结束后，在 runtime/ 下显示保存的 rgb.png 。



ImagePoints

将 Save 算子的 **类型** 属性选择 ImagePoints ，用于保存图像关键点坐标。

算子参数

- **文件/filename**：保存 imagepoints 的文件名。如 *ImagePoints.txt* 。

数据信号输入输出

输入：

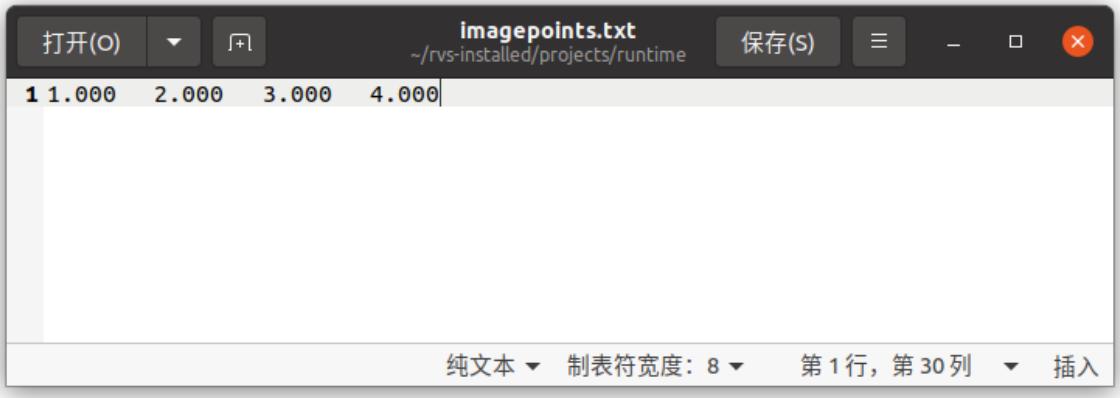
说明：根据需求选择 image_points 或者 image_points_list 其中一种信号输入即可。

- **filename**：
  - 数据类型：String
  - 输入内容：保存图像关键点坐标的文件名
- **image_points**：
  - 数据类型：ImagePoints
  - 输入内容：图像关键点数据
- **image_points_list**：
  - 数据类型：ImagePointsList
  - 输入内容：图像关键点坐标列表数据

功能演示

本节将使用 Save 算子中 ImagePoints ，保存图像关键点坐标。这与 Save 算子中 Cube 属性进行保存立方体的方法相同，请参照该章节的功能演示。

1. 设置 Emit 算子：类型→ ImagePoints，图像点 → 1 2 3 4 。
2. 设置 Save 算子：类型 → ImagePoints，文件 → imagepoints.txt。
3. 算子运行成功后，在 runtime/ 目录下保存 imagepoints.txt。



JointArray

将 Save 算子的 **类型** 属性选择 JointArray ，用于保存机器人关节弧度值。

算子参数

- **文件/filename**：保存 jointarray 的文件名。如 *joints.txt* 。

数据信号输入输出

输入：

说明：根据需求选择 jointarray 或者 jointarray_list 其中一种信号输入即可。

- **filename**：
  - 数据类型：String
  - 输入内容：保存机器人关节值的文件名
- **joint**：
  - 数据类型：JointArray
  - 输入内容：机器人关节值数据
- **joint_list**：
  - 数据类型：JointArrayList
  - 输入内容：机器人关节值列表数据

功能演示

使用 Save 算子中 JointArray ，保存机器人关节弧度值。

步骤1：算子准备

添加 Trigger 、Load 、Save 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → JointArray
   - 文件 → 选择机器人关节弧度值（ *example_data/joints/joints.txt* ）
2. 设置 Save 算子参数：
   - 类型 → JointArray
   - 文件→ joints.txt

步骤3：连接算子

步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，运行算子结束后，在runtime/下显示保存的joints.txt。

Path

将 Save 算子的 **类型** 属性选择 Path ，用于保存路径。

算子参数

- `文件/filename`：保存 path 的文件名。如 *path.txt* 。

数据信号输入输出

输入：

说明：根据需求选择 path 或者 path_list 其中一种信号输入即可。

- `filename`：
  - 数据类型：String
  - 输入内容：保存 path 的文件名
- `path`：
  - 数据类型：Path
  - 输入内容：path 数据
- `path_list`：
  - 数据类型：PathList
  - 输入内容：path 列表数据

功能演示

本节将使用 Save 算子中 Path ，保存路径。这与 Save 算子中 JointArray 属性进行保存机器人关节弧度值的方法相同，请参照该章节的功能演示。

PointCloud

将 Save 算子的 type 属性选择 PointCloud ，用于保存点云。

算子参数

- `文件/filename`：保存点云的文件名。如 *pointcloud.pcd* 。
- `format`：
  - binary：使用二进制格式。
  - ascii：使用 ascii 格式。

数据信号输入输出

输入：

说明：根据需求选择 pointcloud 或者 pointcloud_list 其中一种信号输入即可。

- `filename`：
  - 数据类型：String
  - 输入内容：保存点云的文件名
- `cloud`：
  - 数据类型：PointCloud
  - 输入内容：点云数据
- `cloud_list`：
  - 数据类型：PointCloudList
  - 输入内容：点云列表数据

功能演示

本节将使用 Save 算子中 PointCloud ，用于保存点云，这与 Save 算子中 image 属性保存相机图像的方法相同的方法相同，请参照该章节的功能演示。

PolyData

将 Save 算子的 **类型** 属性选择 PolyData ，用于保存 3D 模型。

算子参数

- **文件/filename**：保存 polydata 的文件名。如 *polydata.obj* 。支持的后缀格式：.obj 。

数据信号输入输出

输入：

说明：根据需求选择 polydata 或者 polydata_list 其中一种信号输入即可。

- **filename**：
  - 数据类型：String
  - 输入内容：保存 polydata 的文件名
- **polydata**：
  - 数据类型：PolyData
  - 输入内容：polydata 数据
- **polydata_list**：
  - 数据类型：PolyDataList
  - 输入内容：polydata 列表数据

功能演示

本节将使用 Save 算子中 PloyData ，保存 3D 模型。这与 Save 算子中 JointArray 属性进行保存机器人关节弧度值的方法相同，请参照该章节的功能演示。

Pose

将 Save 算子的 **类型** 属性选择 Pose ，用于保存坐标。

算子参数

- **文件/filename**：保存 pose 的文件名。如 *pose.txt* 。

数据信号输入输出

输入：

说明：根据需求选择 pose 或者 PoseList 其中一种信号输入即可。

- **filename**：
  - 数据类型：String
  - 输入内容：保存 pose 的文件名
- **pose**：

    - 数据类型：Pose
    - 输入内容：pose 数据
- **pose_list**：
    - 数据类型：PoseList
    - 输入内容：pose 列表数据

功能演示

使用 Save 算子中 Pose ，保存坐标。这与 Save 算子中 Cube 属性保存立方体的方法相同，请参照该章节的功能演示。

String

将 Save 算子的 **类型** 属性选择 String ，用于保存字符串。

算子参数

- **文件/filename**：保存 string 的文件名。如 *string.txt* 。

数据信号输入输出

输入：

说明：根据需求选择 string 或者 string_list 其中一种信号输入即可。

- **filename**：
    - 数据类型：String
    - 输入内容：保存 string 的文件名
- **string**：
    - 数据类型：String
    - 输入内容：string 数据
- **string_list**：
    - 数据类型：StringList
    - 输入内容：string 列表数据

功能演示

本节将使用 Save 算子中 String ，保存字符串。这与 Save 算子中 Cube 属性进行保存立方体的方法相同，请参照该章节的功能演示。

AtList 列表提取元素

AtList 算子为列表提取元素，用于提取 Cube 、Image 、JointArray 、PointCloud 、Pose、ImagePoints 列表中的单个或多个元素。

AtList 算子介绍视频

类型	功能
Cube	用于提取立方体列表元素。
Image	用于提取图像列表元素。
JointArray	用于提取机器人关节弧度值列表元素。
PointCloud	用于提取点云列表元素。
Pose	用于提取坐标列表元素。
ImagePoints	用于提取图像关键点坐标列表元素。
RotatedRect	/

模式	功能
SingleByIndex	提取列表中单个元素。
ListByIndex	提取列表中多个元素，可与 SortList、SubList 算子配合使用。

Cube

将 AtList 算子的 **类型** 属性选择 Cube ，用于提取立方体列表元素。

算子参数

- **索引/index**：索引。默认值：0 ，表示输出列表中的第一个。
- **立方体/cube**：设置立方体在 3D 视图中的可视化属性。
  - 打开立方体可视化。
  - 关闭立方体可视化。
  - 设置立方体的颜色。取值范围：[-2,360] 。默认值：-2 。
  - 设置立方体的透明度。取值范围：[0,1] 。默认值：0.5 。

数据信号输入输出

输入：

- **cube_list**：
  - 数据类型：CubeList
  - 输入内容：立方体数据列表

输出：

- `cube`：
  - 数据类型：Cube
  - 输出内容：指定索引的立方体数据

功能演示

mode：SingleByIndex

使用 AtList 算子中 Cube 提取加载立方体列表中索引为 0 的 cube 元素。

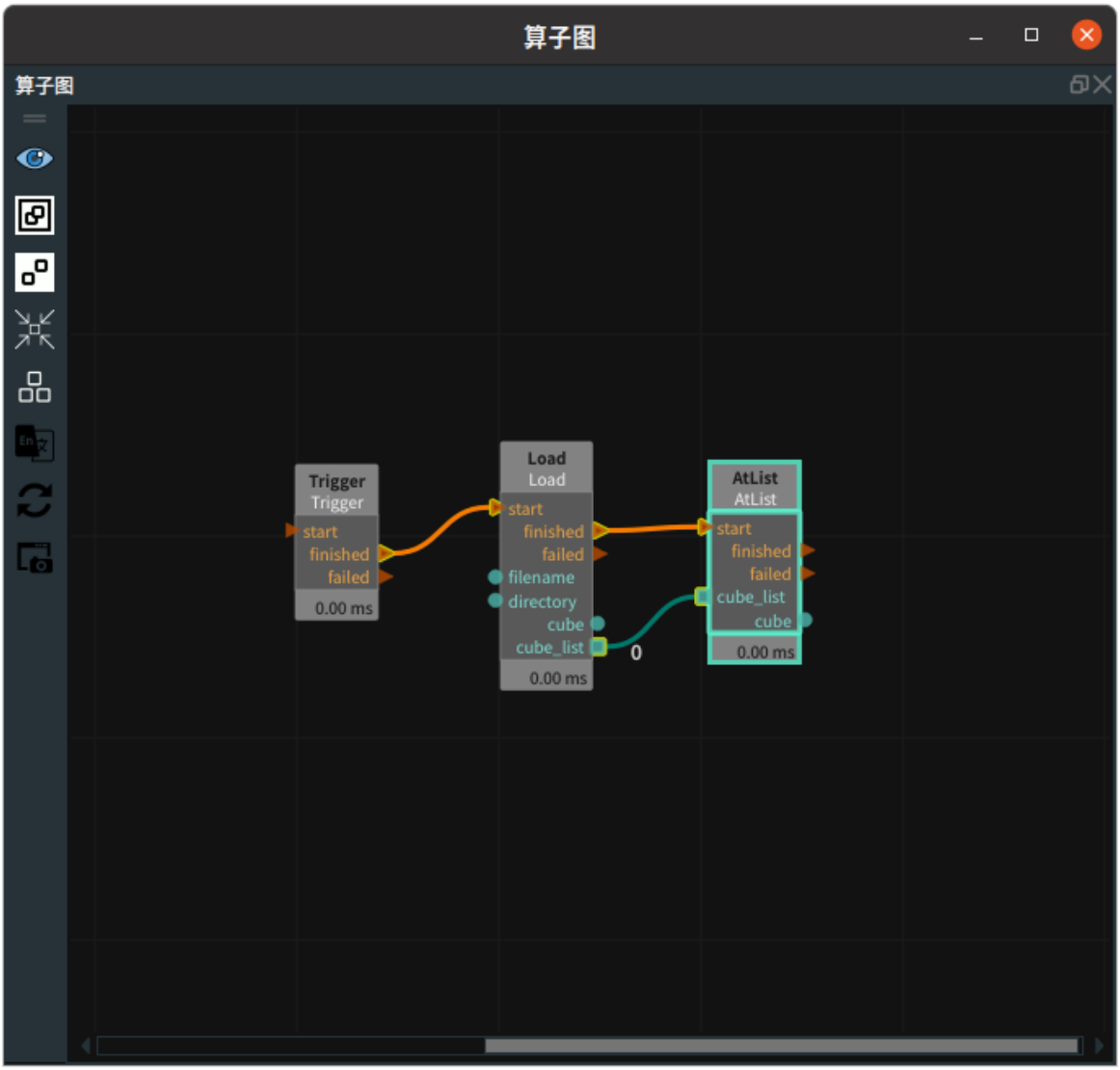
步骤1：算子准备

添加 Trigger 、Load 、AtList 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - type → Cube
   - directory → ••• → 选择 cube 文件目录名（ *example_data/cube* ）
2. 设置 AtList 算子参数：
   - type → Cube
   - mode → SingleByIndex
   - index → 0
   - cube →  可视

步骤3：连接算子

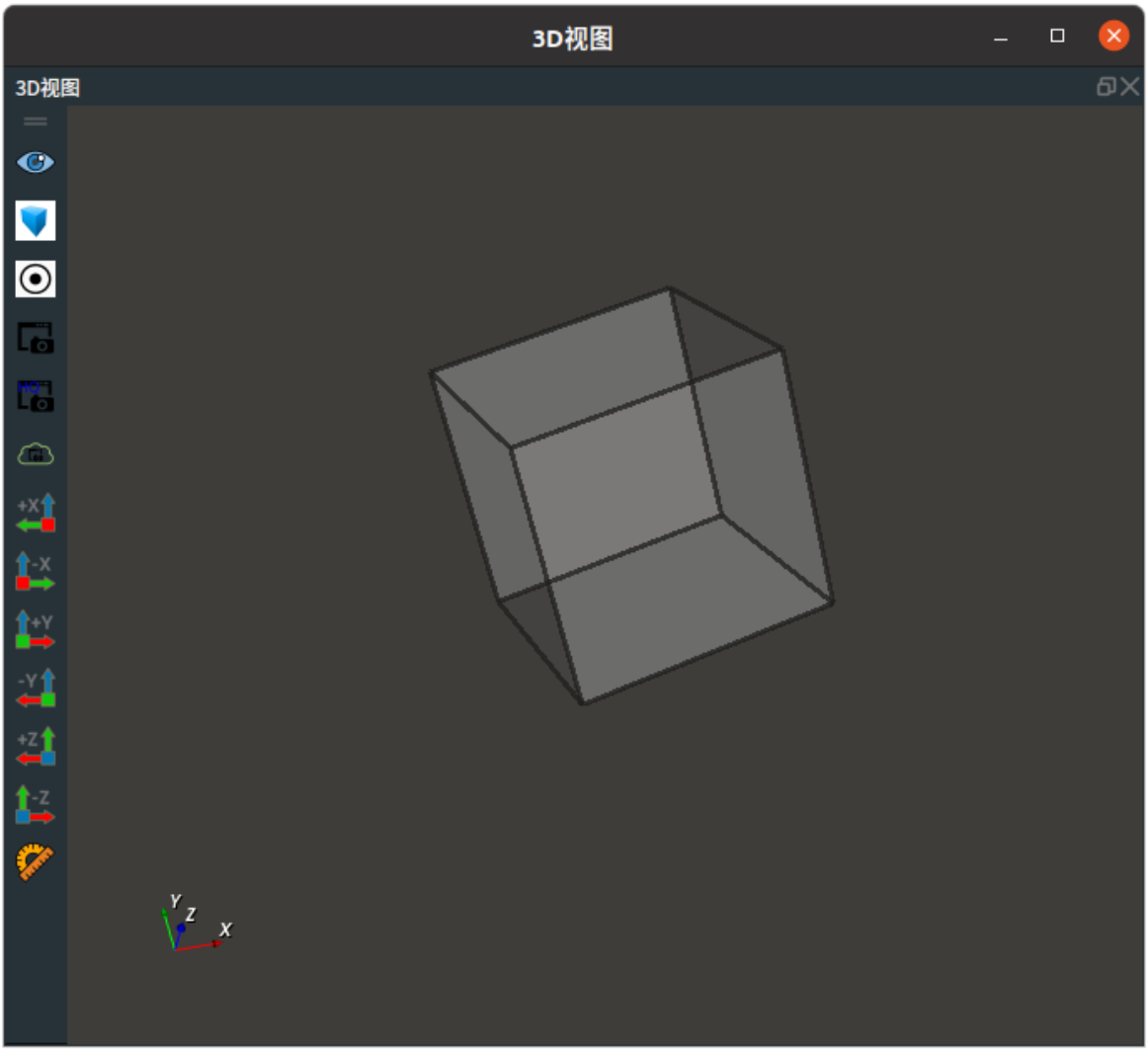


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，在 3D 视图中显示立方体列表索引为 0 的 cube 。



mode：ListByIndex

使用 AtList 算子中 Cube 提取排序后点云索引为 0~2 的 Cube 列表。

步骤1：算子准备

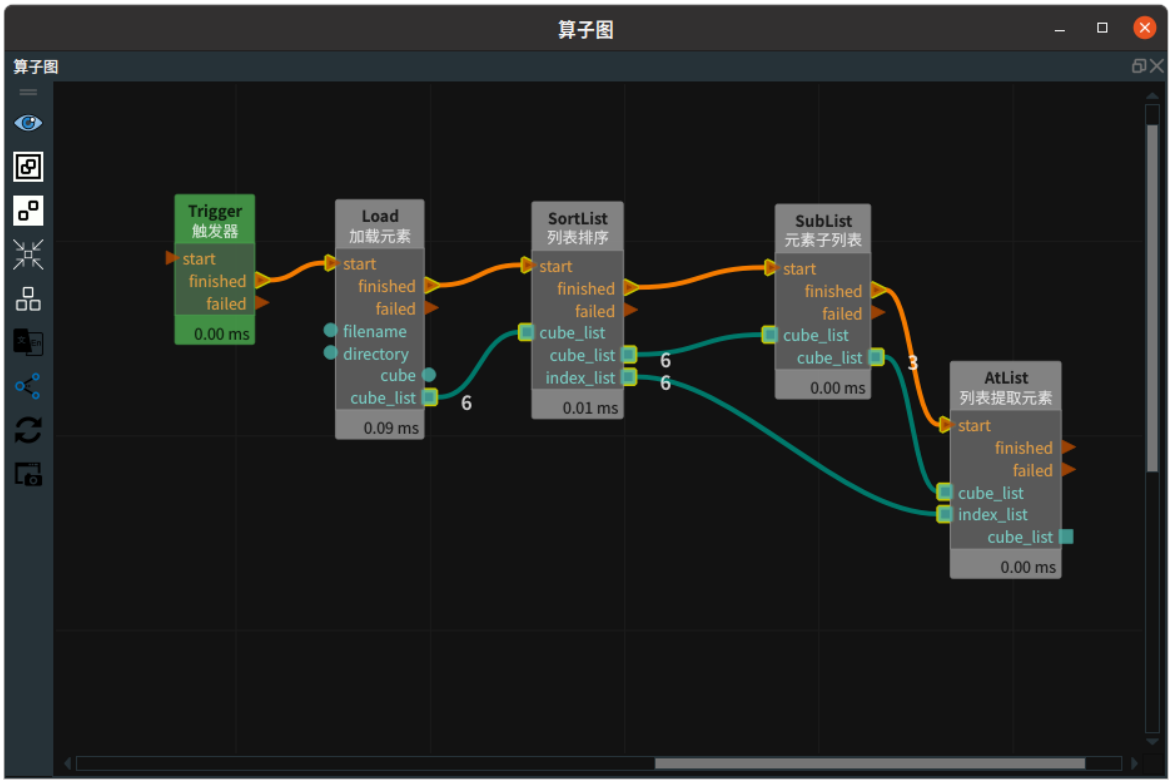
添加 Trigger 、Load 、SortList、SubList、AtList 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 类型 → Cube
    - 目录 → ••• → 选择 cube 文件目录名（ *example_data/cubelist.txt* ）
2. 设置 SortList 算子参数：
    - 类型 → Cube
    - X权重 → 1
    - Y权重 → 0
    - Z权重 → 0
3. 设置 SubList 算子参数：
    - 类型 → Cube
    - 起始索引 → 0
    - 终止索引 → 2
4. 设置 AtList 算子参数：
    - 类型 → Cube

- 模式 →ListByIndex
- 立方体 → 可视

步骤3：连接算子

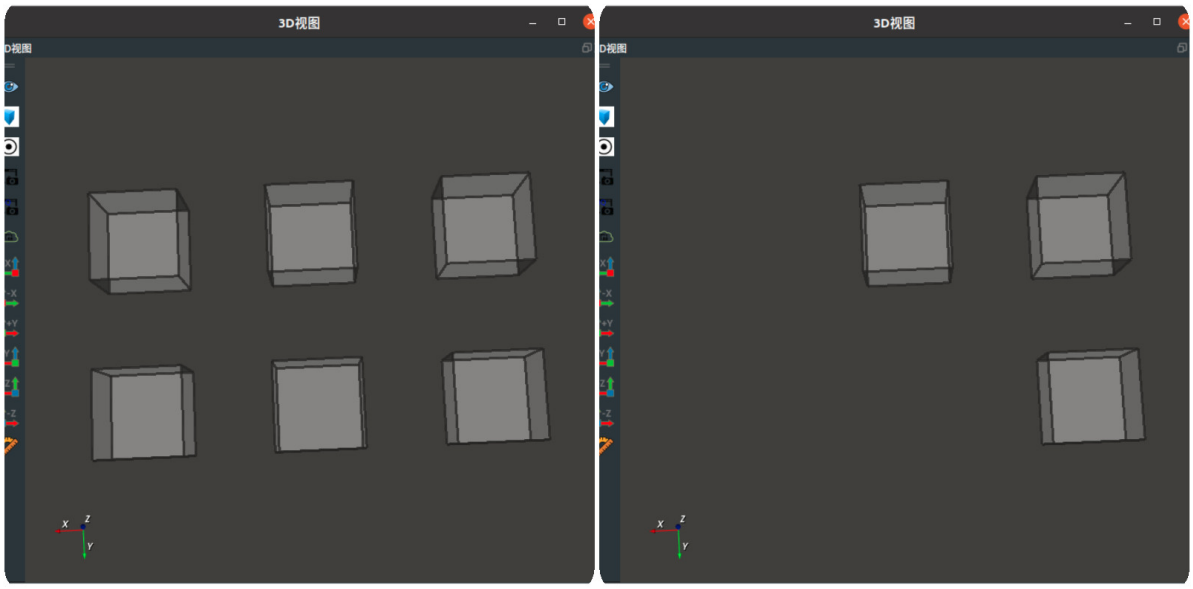


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，左图为 Load 算子的 cube 列表可视化结果，右图为排序后索引为 0~2 的 cube 列表。



Image

将 AtList 算子的 **类型** 属性选择 Image ，用于提取图像列表元素。

算子参数

- **索引/index**：索引。默认值： 0 ，表示输出列表中的第一个。
- **图像/image**：设置图像在 2D 视图中的可视化属性。
  -  打开图像可视化。
  -  关闭图像可视化。

数据信号输入输出

输入：

- **image_list**：
  - 数据类型：ImageList
  - 输入内容：图像数据列表

输出：

- **image**：
  - 数据类型：Image
  - 输输出内容：指定索引的图像数据

功能演示

本节将使用 AtList 中 Image ，提取加载图像列表索引为 0 的 image 元素。这与 AtList 算子中 cube 属性的提取立方体列表元素的方法相同，请参照该章节的功能演示。

JointArray

将 AtList 算子的 **类型** 属性选择 JointArray ，用于提取机器人关节弧度值列表元素。

算子参数

- **index**：索引。默认值：0 ，表示输出列表中的第一个。

数据信号输入输出

输入：

- **joint_list**：
  - 数据类型：JointArrayList
  - 输入内容：机器人关节弧度值列表

输出：

- **joint**：
  - 数据类型：JointArray
  - 输出内容：指定索引的机器人关节弧度值

功能演示

使用 AtList 算子中 JointArray ，提取加载机器人关节弧度值列表中索引为 0 的 jointarray 元素。

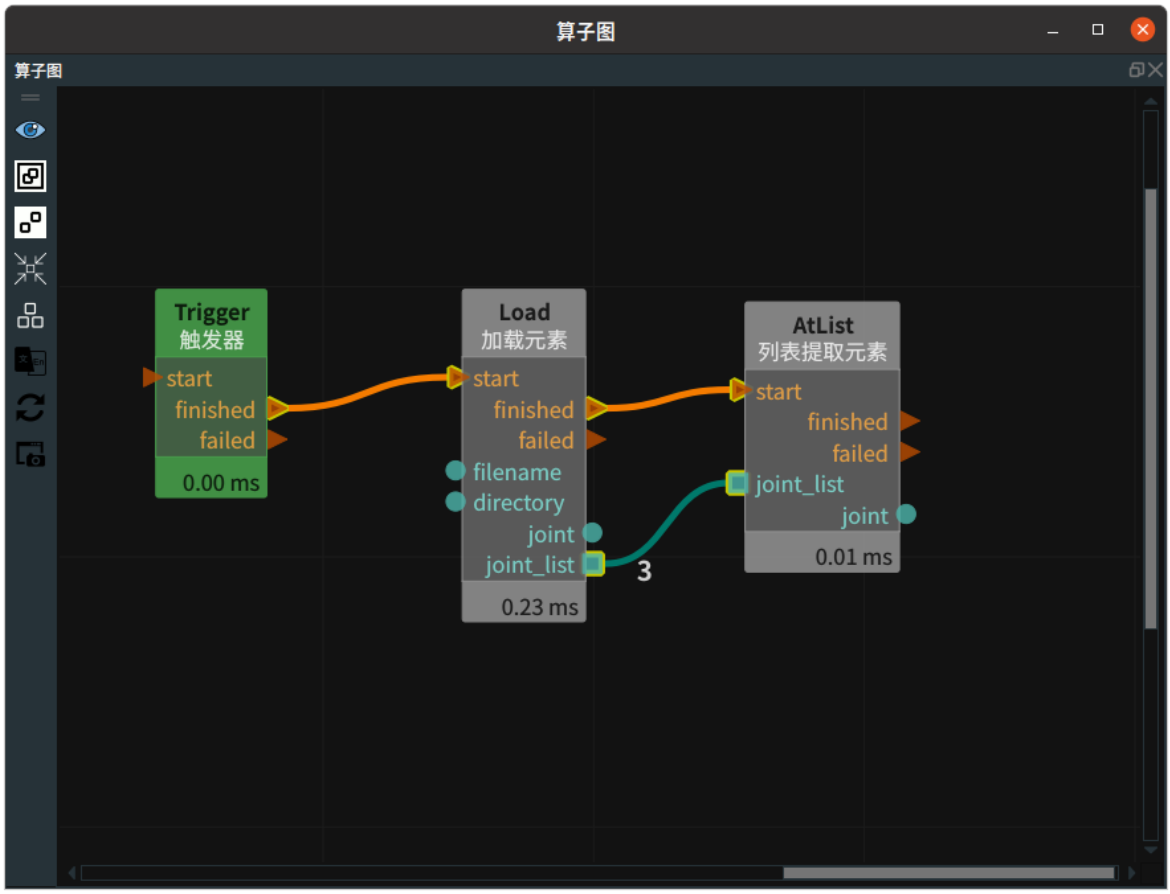
步骤1：算子准备

添加 Trigger、Load、AtList算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 类型 → JointArray
    - 目录 → ••• → 选择 JointArray 文件目录名（ *example_data/joints* ）
2. 设置 AtList 算子参数：
    - 类型 → JointArray
    - 模式 → SingleByIndex
    - 索引 → 0

步骤3：连接算子

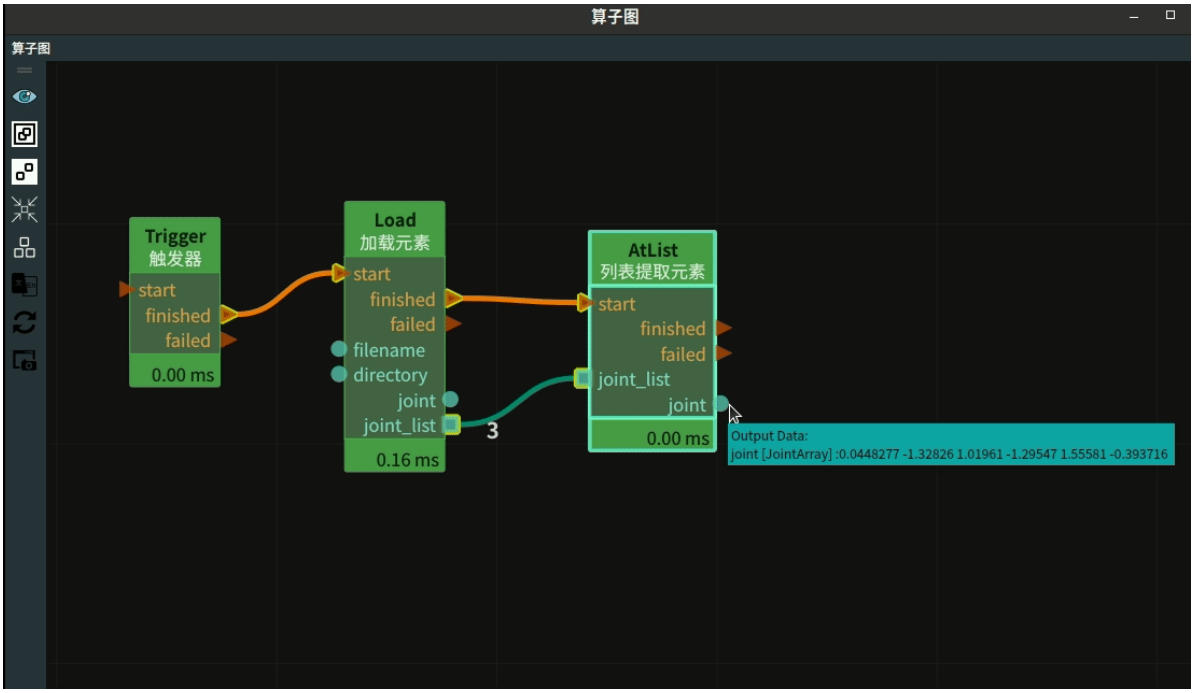


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，在算子图中 AtList 右侧数据输出端口显示索引为 0 的 JointArray 。



PointCloud

将 AtList 算子的 **类型** 属性选择 PointCloud ，用于提取点云列表元素。

算子参数

- **索引/index**：索引。默认值：0，表示输出列表中的第一个。
- **点云/cloud**：设置点云在 3D 视图中的可视化属性。
  - 打开点云可视化。
  - 关闭点云可视化。
  - 设置 3D 视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  - 设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。

数据信号输入输出

输入：

- **cloud_list**：
  - 数据类型：PointCloudList
  - 输入内容：点云列表数据

输出：

- **cloud**：
  - 数据类型：PointCloud
  - 输出内容：指定索引的点云数据

功能演示

本节将使用 AtList 算子的中 PointCloud ，提取加载点云列表中索引为 0 的点云元素。这与 AtList 算子中 cube 属性的提取立方体列表元素的方法相同，请参照该章节的功能演示。

Pose

将 AtList 算子的 **类型** 属性选择 Pose ，用于提取坐标列表元素。

算子参数

- **索引/index**：索引。默认值：0 。表示输出列表中的第一个。
- **坐标/pose**：设置 pose 在 3D 视图中的可视化属性。
  -  打开坐标可视化。
  -  关闭坐标可视化。
  -  设置坐标的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

- **pose_list**：
  - 数据类型：PoseList
  - 输入内容：pose 列表数据

输出：

- **pose**：
  - 数据类型：Pose
  - 输出内容：指定索引的 pose 数据

功能演示

本节将使用 AtList 算子的中 Pose ，提取坐标列表中索引为 0 的 pose 元素。这与 AtList 算子中 cube 属性的提取立方体列表元素的方法相同，请参照该章节的功能演示。

ImagePoints

将 AtList 算子的 **类型** 属性选择 ImagePoints ，用于提取图像关键点坐标列表元素。

算子参数

- **索引/index**：索引。默认值：0 。表示输出列表中的第一个。

数据信号输入输出

输入：

- **image_points_list**：
  - 数据类型：imagepointslist
  - 输入内容：图像关键点坐标列表数据

输出：

- **image_points**：
  - 数据类型：imagepoints
  - 输出内容：指定索引的图像关键点坐标数据

功能演示

使用 AtList 算子中 ImagePoints ，提取图像关键点坐标列表索引为 0 的 imagepoints 元素。这与 AtList 算子中 JointArray 属性的提取机器人关节值列表元素的方法相同，请参照该章节的功能演示。

SubList 元素子列表

SubList算子为元素子列表，获取列表中指定范围的子列表。适用于：Cube 、Image 、JointArray 、PointCloud 、Pose 列表。

类型	功能
Cube	用于获取立方体列表中指定范围的子列表。
Image	用于获取图像列表中指定范围的子列表。
JointArray	用于获取机器人关节值列表中指定范围的子列表。
PointCloud	用于获取点云列表中指定范围的子列表。
Pose	用于获取 pose 列表中指定范围的子列表。
ImagePoints	用于获取图像关键点坐标列表中指定范围的子列表。

Cube

将 SubList 算子的 **类型** 属性选择 Cube ，用于获取立方体列表中指定范围的子列表。

算子参数

- **起始索引/begin_index**：表示子列表的开始索引。默认值：0 。表示从原列表的第一个开始选取。
- **终止索引/end_index**：表示子列表的结束索引。默认值：-1 。表示 size - 1 ，为原列表的最后一个。
- **立方体列表/cube_list**：设置立方体子列表在 3D 视图中的可视化属性。
  -  打开立方体子列表可视化。
  -  关闭立方体子列表可视化。
  -  设置立方体子列表的颜色。取值范围：[-2,360] 。默认值：-2 。
  -  设置立方体子列表的透明度。取值范围：[1,10] 。默认值：0.5 。

数据信号输入输出

输入：

- **cube_list**：
  - 数据类型：CubeList
  - 输入内容：原立方体列表数据

输出：

- **cube_list**：
  - 数据类型：CubeList
  - 输出内容：立方体子列表数据

功能演示

使用 SubList 算子中 Cube ，获取加载立方体列表中索引 0 ~ 1 的子列表。

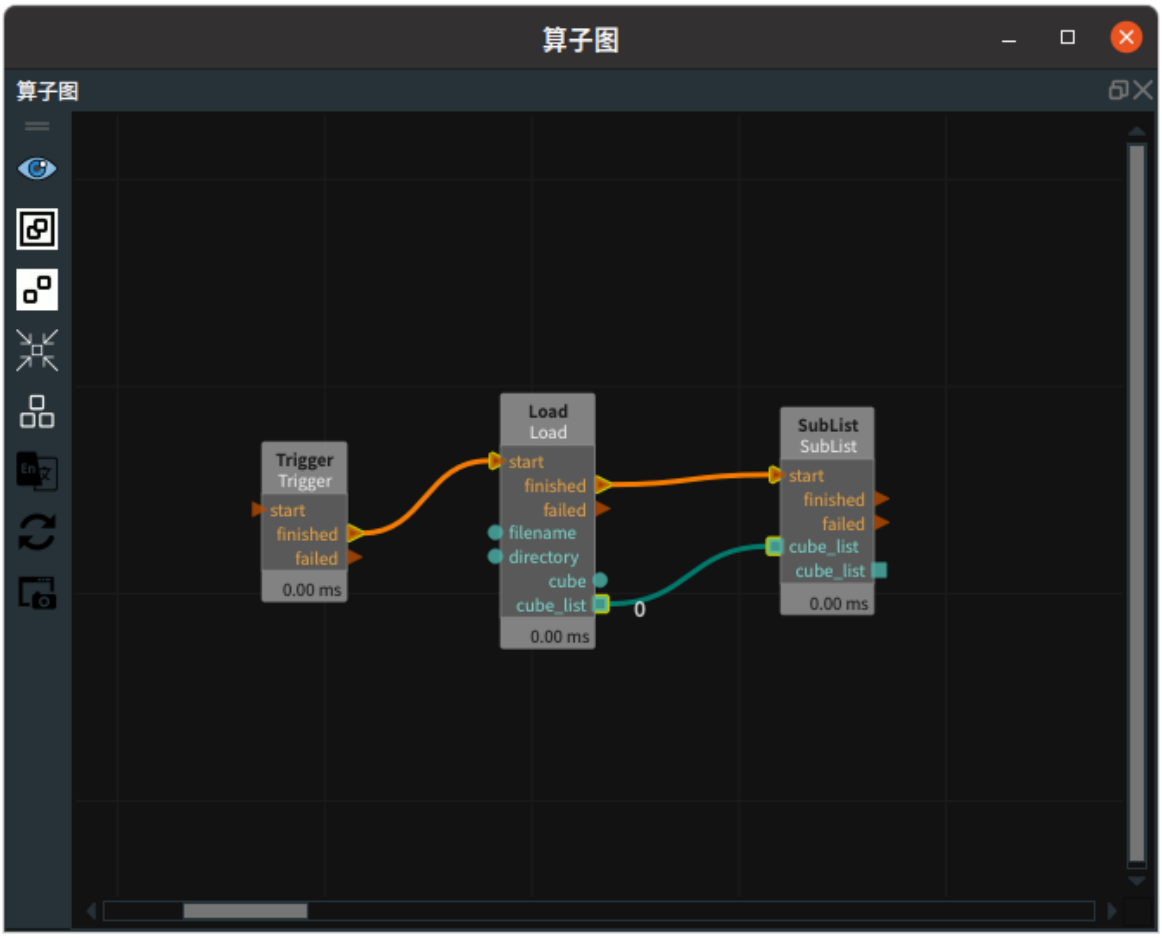
步骤1：算子准备

添加 Trigger 、Load 、SubList 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 类型 → Cube
    - 目录 → ••• → 选择 cube 文件目录名（*example_data/cube*）
2. 设置 SubList 算子参数：
    - 类型 → Cube
    - 起始索引 → 0
    - 终止索引 → 1
    - 立方体列表 → 可视

步骤3：连接算子

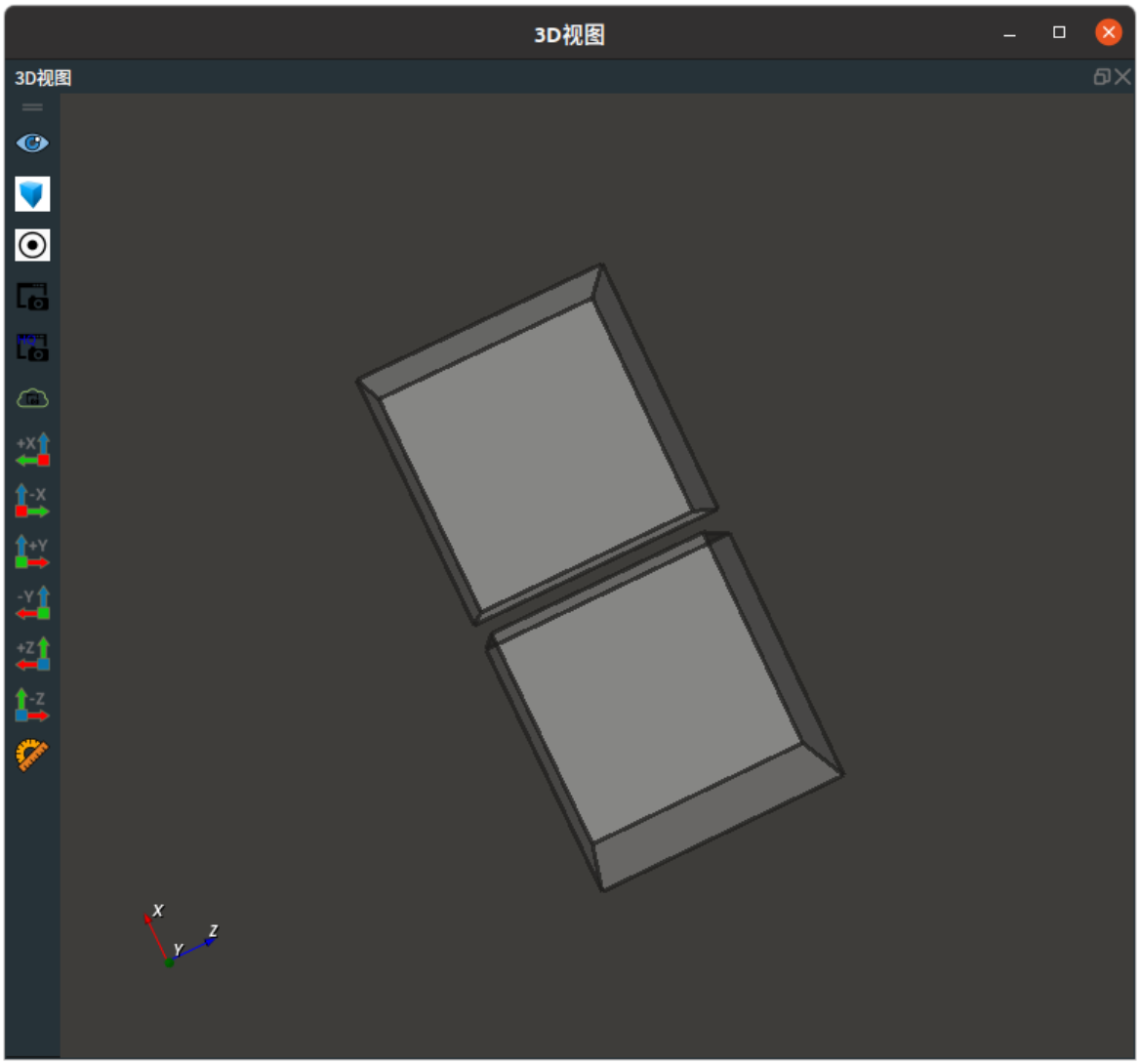


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

3D 视图显示如下，在 3D 视图中显示加载的 cube 列表中索引 0 ~ 1 的 cube 子列表。



Image

将 SubList 算子的 **类型** 属性选择 Image ，用于获取图像列表中指定范围的子列表。

算子参数

- **起始索引/begin_index**：表示子列表的开始索引。默认值：0 ，表示从原列表的第一个开始选取。
- **终止索引/end_index**：表示子列表的结束索引。默认值：-1 ，表示 size-1 ，为原列表的最后一个。
- **图像列表/image_list**：设置图像子列表在 3D 视图中的可视化属性。
  - 打开图像子列表可视化。
  - 关闭图像子列表可视化。

数据信号输入输出

输入：

- **image_list**：
  - 数据类型：ImageList
  - 输入内容：原图像列表数据

输出：

- **image_list**：

- 数据类型：ImageList
- 输出内容：图像子列表数据

功能演示

本节将使用 SubList 算子中 Image ，获取图像列表中索引 0 ~ 1 的子列表。这与 SubList 算子中 Cube 属性获取加载立方体列表中索引 0 ~ 1 的子列表方法相同，请参照该章节的功能演示。

JointArray

将 SubList 算子的 `类型` 属性选择 JointArray ，用于获取机器人关节值列表中指定范围的子列表。

算子参数

- `起始索引/begin_index`：表示子列表的开始索引。默认值：0 ，表示从原列表的第一个开始选取。
- `终止索引/end_index`：表示子列表的结束索引。默认值：-1 ，表示 size-1 ，为原列表的最后一个。

数据信号输入输出

输入：

- `joint_list`：
  - 数据类型：JointArrayList
  - 输入内容：原机器人关节值列表数据

输出：

- `joint_list`：
  - 数据类型：JointArrayList
  - 输出内容：机器人关节值子列表数据

功能演示：

本节将使用 SubList 算子中 JointArray ，获取机器人关节值列表中索引 0 ~ 1 的子列表。这与 SubList 算子中 Cube 属性获取加载立方体列表中索引 0 ~ 1 的子列表方法相同，请参照该章节的功能演示。

PointCloud

将 SubList 算子的 `类型` 属性选择 PointCloud ，用于点云列表中指定范围的子列表。

算子参数

- `起始索引/begin_index`：表示子列表的开始索引。默认值：0，表示从原列表的第一个开始选取。
- `终止索引/end_index`：表示子列表的结束索引。默认值：-1，表示 size-1 ，为原列表的最后一个。
- `点云列表/cloud_list`：设置点云子列表在 3D 视图中的可视化属性。
  -  打开点云子列表可视化。
  -  关闭点云子列表可视化。
  -  设置 3D 视图中点云子列表的颜色。取值范围：[-2,360] 。默认值：-1 。
  -  设置点云子列表中点的尺寸。取值范围：[1,50] 。默认值：1 。

数据信号输入输出

输入：

- `cloud_list`：
  - 数据类型：PointCloudList
  - 输入内容：原点云列表数据

输出：

- `cloud_list`：
  - 数据类型：PointCloudList
  - 输出内容：点云子列表数据

功能演示

本节将使用 SubList 算子中 PointCloud ，获取点云列表中索引 0 ~ 1 的子列表。这与 SubList 算子中 Cube 属性获取加载立方体列表中索引 0 ~ 1 的子列表方法相同，请参照该章节的功能演示。

Pose

将 SubList 算子的 `类型` 属性选择 Pose ，用于 pose 列表中指定范围的子列表。

算子参数

- `起始索引/begin_index`：表示子列表的开始索引。默认值：0 ，表示从原列表的第一个开始选取。
- `终止索引/end_index`：表示子列表的结束索引。默认值：-1 ，表示 size-1 ，为原列表的最后一个。
- `坐标列表/pose_list`：设置 pose 子列表的可视化属性，默认为关闭状态。
  -  打开 pose 子列表可视化。
  -  关闭 pose 子列表可视化。
  -  设置 pose 子列表的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

- `pose_list`：
  - 数据类型：PoseList
  - 输入内容：原 pose 列表数据

输出：

- `pose_list`：
  - 数据类型：PoseList
  - 输出内容：pose 子列表数据

功能演示

本节将使用 SubList 算子中 Pose ，获取 pose 列表中索引 0 ~ 1 的子列表。这与 SubList 算子中 Cube 属性获取加载立方体列表中索引 0 ~ 1 的子列表方法相同，请参照该章节的功能演示。

ImagePoints

将 SubList 算子的 **类型** 属性选择 ImagePoints ，用于获取图像关键点坐标列表中指定范围的子列表。

算子参数

- **起始索引/begin_index**：表示子列表的开始索引。默认值：0 ，表示从原列表的第一个开始选取。
- **终止索引/end_index**：表示子列表的结束索引。默认值：-1 ，表示 size-1 ，为原列表的最后一个。

数据信号输入输出

输入：

- **image_points_list**：
  - 数据类型：ImagePointsList
  - 输入内容：原图像关键点坐标列表数据

输出：

- **image_points_list**：
  - 数据类型：ImagePointsList
  - 输出内容：图像关键点坐标子列表数据

功能演示

本节将使用 SubList 算子中 ImagePoints ，获取图像关键点坐标列表中索引 0 ~ 1 的子列表。这与 SubList 算子中 Cube 属性获取加载立方体列表中索引 0 ~ 1 的子列表方法相同，请参照该章节的功能演示。

Concatenate 连接元素

Concatenate 算子为连接元素，用于连接多个相同类型。适用的类型有 Cube 、JointArray 、PointCloud 、PolyData 、Pose、ImagePoints 。

类型	功能
Cube	用于连接多个立方体或立方体列表。
Image	用于连接多个图像或图像列表。
JointArray	用于连接多个机器人关节弧度值或机器人关节弧度值列表。
PointCloud	用于连接多个点云或者点云列表。
PolyData	用于连接多个 3D 模型或者 3D 模型列表。
Pose	用于连接多个坐标或者坐标列表。
ImagePoints	用于连接多个图像关键点坐标或者坐标列表。

Cube

将 Concatenate 算子的 `类型` 属性选择为 Cube ，用于连接多个立方体或立方体列表。

算子参数

- `输入数量/port_number`：决定该算子的输入端口 input_? 的数量。取值范围：[0,10] 。默认值：2 。
- `列表输入数量/list_port_number`：决定该算子的输入端口 input_list_? 的数量。取值范围：[0,10] 。默认值：2 。
- `立方体列表/cube_list`：设置立方体列表在 3D 视图中的可视化属性。
  -  打开立方体列表可视化。
  -  关闭立方体列表可视化。
  -  设置立方体列表的颜色。取值范围：[-2,360] 。默认值：-2 。
  -  设置立方体列表的透明度。取值范围：[0,1] 。默认值：0.5 。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- `cube_?`：
  - 数据类型：Cube
  - 输入内容：立方体数据列表
- `cube_list_?`：
  - 数据类型：CubeList
  - 输入内容：立方体数据列表

输出：

- `cube_list`：
    - 数据类型：CubeList
    - 输出内容：连接后的立方体数据

功能演示

使用 Concatenate 算子中 Cube ，连接多2个立方体组成一个立方体列表。

步骤1：算子准备

添加 Trigger 、Emit （2 个）、Concatenate 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
    - 类型 → Cube
    - 坐标 → 0 0 0 0 0 0
    - 宽度→ 1
    - 高度→ 1
    - 深度 → 1
2. 设置 Emit_1 算子参数：
    - 类型 → Cube
    - 坐标 → 0 0 0.75 0 0 0
    - 宽度→ 0.5
    - 高度 → 0.5
    - 深度 → 0.5
3. 设置 Concatenate 算子参数：
    - 类型 → Cube
    - 立方体列表 →  可视

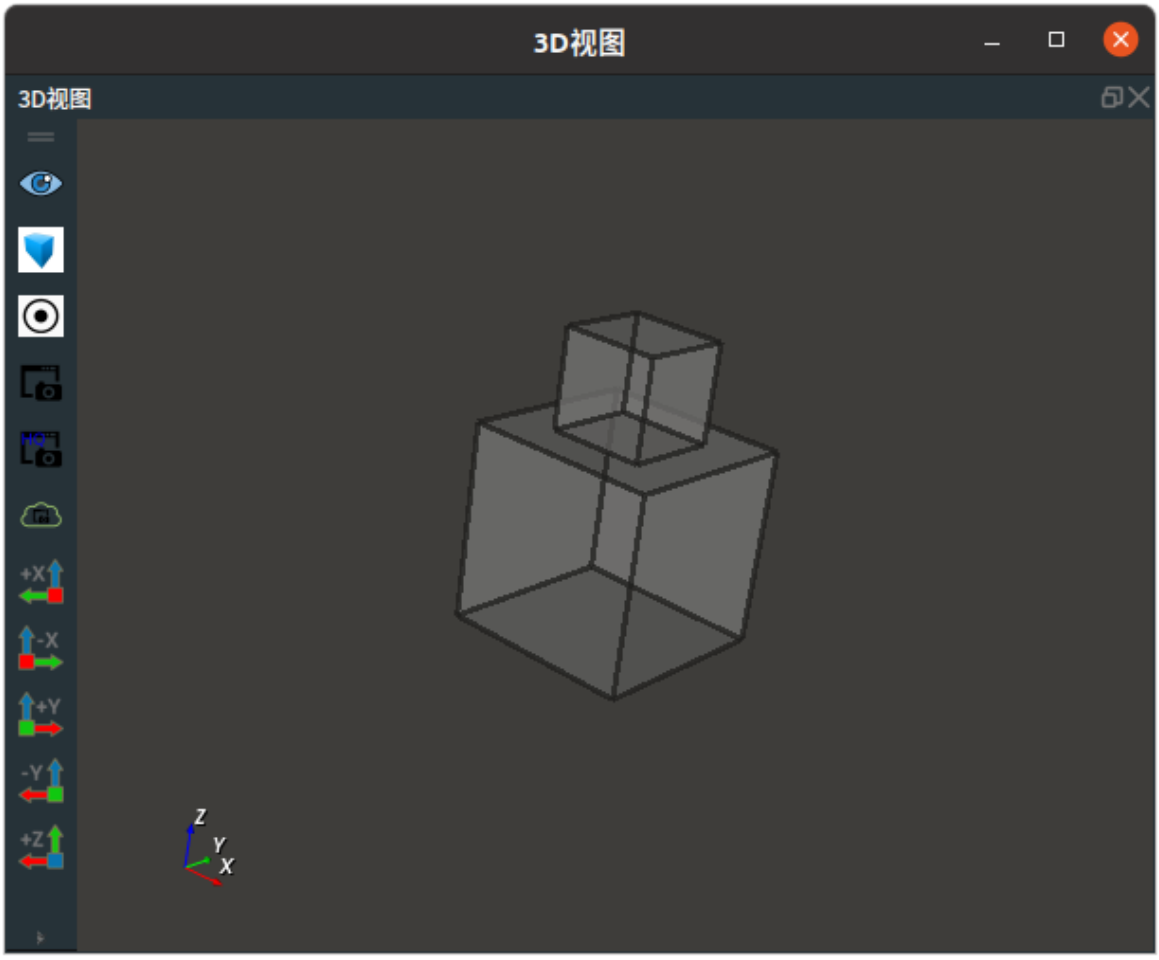
步骤3：连接算子

步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

结果如下图所示，在 3D 视图中显示生成的两个 cube 所连接的一个 cube 列表。



Image

将 Concatenate 算子的 **类型** 属性选择为 Image ，用于连接多个图像或图像列表。

算子参数

- **输入数量/port_number**：决定该算子的输入端口 input_? 的数量。取值范围：[0,10] 。默认值：2 。
- **列表输入数量/list_port_number**：决定该算子的输入端口 input_list_? 的数量。取值范围：[0,10] 。默认值：2 。
- **图像列表/image_list**：设置图像列表在 3D 视图中的可视化属性。
  - 打开图像列表可视化。
  - 关闭图像列表可视化。

数据信号输入输出

输入：

- **image_?**：
  - 数据类型：Image
  - 输入内容：图像数据
- **image_list_?**：
  - 数据类型：ImageList
  - 输入内容：图像数据列表

输出：

- `image_list_?`：
  - 数据类型：ImageList
  - 输出内容：连接后的图像数据列表

功能演示

本节将使用 Concatenate 算子中 Image ，连接2个加载的图像组成一个图像列表。

这与 Concatenate 算子中 JointArray 属性的连接多个机器人关节弧度值或机器人关节弧度值列表的方法相同，请参照该章节的功能演示。

JointArray

将 Concatenate 算子 `类型` 属性选择为 JointArray ，用于连接多个机器人关节弧度值或机器人关节弧度值列表。

算子参数

- `输入数量/port_number`：决定该算子的输入端口 input_? 的数量。取值范围：[0,10] 。默认值：2 。
- `列表输入数量/list_port_number`：决定该算子的输入端口 input_list_? 的数量。取值范围：[0,10] 。默认值：2 。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- `joint_?`：
  - 数据类型：JointArray
  - 输入内容：机器人关节弧度值
- `joint_list_?`：
  - 数据类型：JointArrayList
  - 输入内容：机器人关节弧度值列表

输出：

- `joint_list`：
  - 数据类型：JointArray
  - 输出内容：连接后的机器人关节弧度值列表

功能演示

使用 Concatenate 算子中 JointArray ，连接 2 个加载的机器人关节弧度值，组成一个机器人关节弧度值列表。

步骤1：算子准备

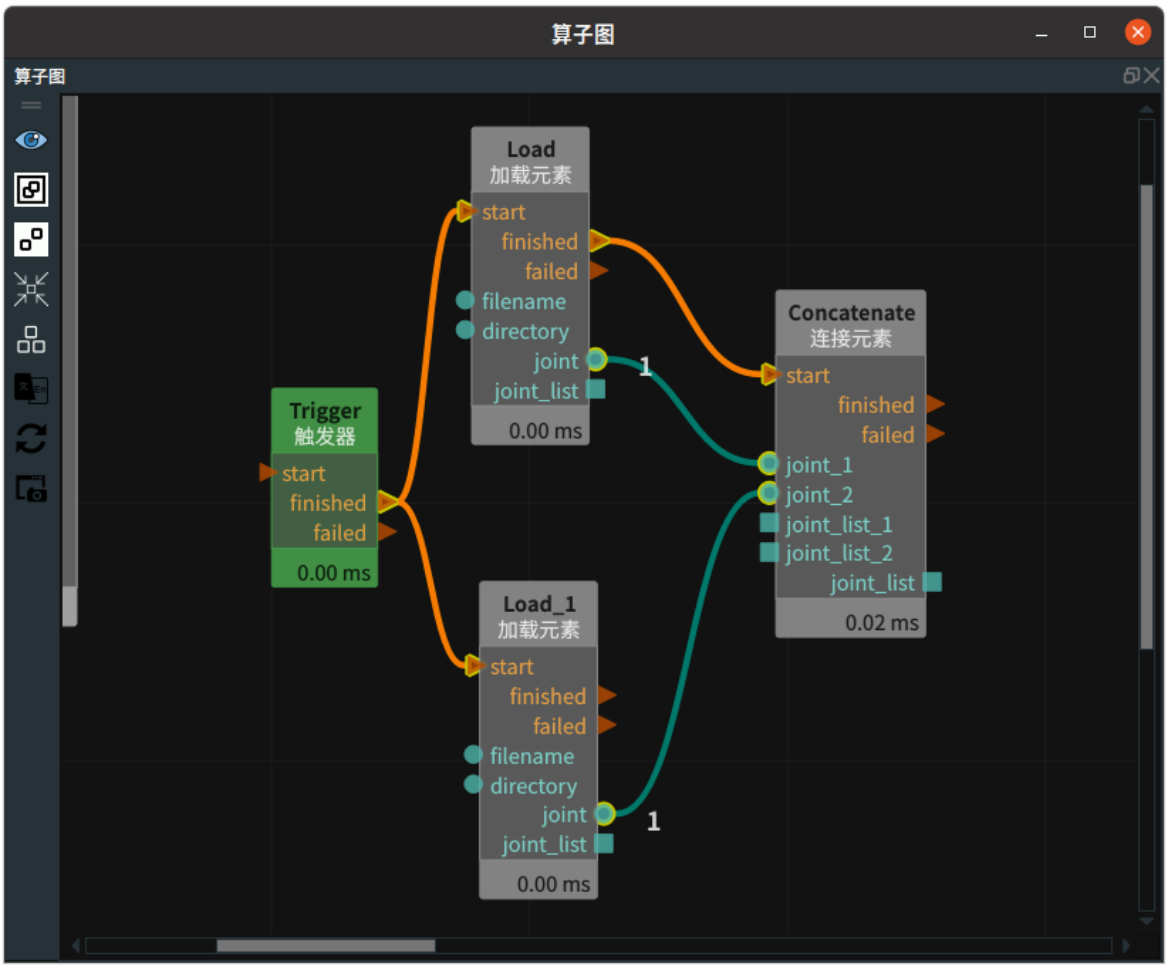
添加 Trigger 、Load（2个） 、Concatenate 算子至算子图。

步骤2：设置算子参数

1. 设置 Load / Load_1 算子参数：
   - 类型 → JointArray
   - 文件 → ●●● → 选择 jointarray 文件名（ *Joints.txt / Joints1.txt* ）

2. 设置 Concatenate 算子参数：
    - 类型 → JointArray

步骤3：连接算子

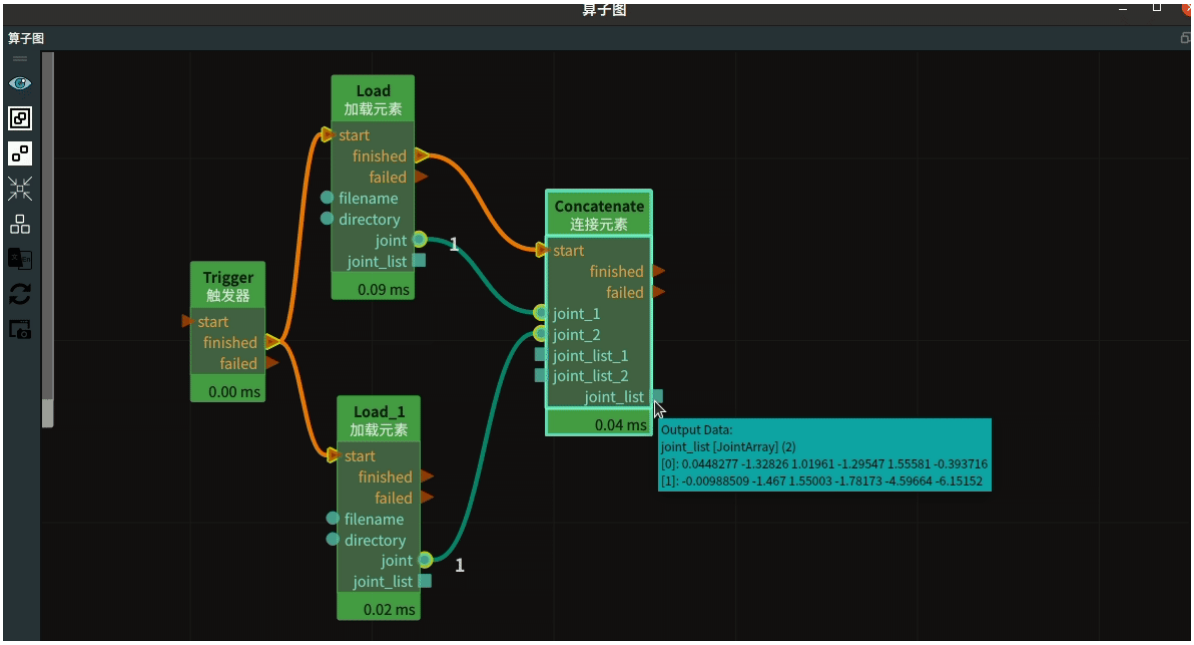


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

结果如下图所示，Concatenate 算子的右侧输出端口输出两个加载出的 joint 所连接成的一个 joint_list 。



PointCloud

将 Concatenate 算子 **类型** 属性选择为 PointCloud ，用于连接多个点云或者点云列表。

算子参数

- **输入数量/port_number**：决定该算子的输入端口 input_? 的数量。取值范围：[0,10] 。默认值：2 。
- **列表输入数量/list_port_number**：决定该算子的输入端口 input_list_? 的数量。取值范围：[0,10] 。默认值：2 。
- **点云列表/cloud_list**：设置点云列表在 3D 视图中的可视化属性。
  - 打开点云列表可视化。
  - 关闭点云列表可视化。
  - 设置 3D 视图中点云列表的颜色。取值范围：[-2,360] 。默认值：-1 。
  - 设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。

数据信号输入输出

输入：

- **cloud_?**：
  - 数据类型：PointCloud
  - 输入内容：点云数据
- **cloud_list_?**：
  - 数据类型：PointCloudList
  - 输入内容：点云列表数据

输出：

- **cloud_list**：
  - 数据类型：PointCloudList
  - 输出内容：连接后的点云列表数据

功能演示

本节将使用 Concatenate 算子中 PointCloud ，连接2个加载的点云组成一个点云列表。

这与 Concatenate 算子中 JointArray 属性的连接多个机器人关节弧度值或机器人关节弧度值列表的方法相同，请参照该章节的功能演示。

PolyData

将 Concatenate 算子 `类型` 属性选择为 PolyData ，用于连接多个 3D 模型或者 3D 模型列表。

算子参数

- `输入数量/port_number`：决定该算子的输入端口 input_? 的数量。取值范围：[0,10] 。默认值：2 。
- `列表输入数量/list_port_number`：决定该算子的输入端口 input_list_? 的数量。取值范围：[0,10] 。默认值：2 。
- `多边形列表/polydata_list`：设置 3D 模型列表在 3D 视图中的可视化属性。
  -  打开 3D 模型列表可视化。
  -  关闭 3D 模型列表可视化。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- `polydata_?`：
  - 数据类型：PolyData
  - 输入内容：polydata 数据
- `polydata_list_?`：
  - 数据类型：PolyDataList
  - 输入内容：polydata 数据列表

输出：

- `polydata_list`：
  - 数据类型：PolyDataList
  - 输出内容：连接后的 PolyData 列表数据

功能演示

本节将使用 Concatenate 算子中 PolyData ，用于连接 2 个 3D 模型列表，组成一个 3D 模型列表。

这与 Concatenate 算子中 JointArray 属性的连接多个机器人关节弧度值或机器人关节弧度值列表的方法相同，请参照该章节的功能演示。

Pose

将 Concatenate 算子 **类型** 属性选择为 Pose ，用于连接多个 pose 或者 pose 列表。

算子参数

- **输入数量/port_number**：决定该算子的输入端口 input_? 的数量。取值范围：[0,10] 。默认值：2 。
- **列表输入数量/list_port_number**：决定该算子的输入端口 input_list_? 的数量。取值范围：[0,10] 。默认值：2 。
- **坐标列表/pose_list**：设置 pose 列表在 3D 视图中的可视化属性。
  -  打开 pose 列表列表可视化。
  -  关闭 pose 列表列表可视化。
  -  设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

算子输入输出

数据信号输入输出：

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **pose_?** ：
  - 数据类型：Pose
  - 输入内容：pose 数据
- **pose_list?**：
  - 数据类型：PoseList
  - 输入内容：pose 列表数据

输出：

- **pose_list**：
  - 数据类型：PoseList
  - 输出内容：连接后的 pose 列表数据

功能演示

本节将使用 Concatenate 算子中 Pose ，连接 2 个 pose 组成一个 pose 列表。

这与 Concatenate 算子中 JointArray 属性的连接 2 个机器人关节弧度值方法相同，请参照该章节的功能演示。

ImagePoints

将 Concatenate 算子 **类型** 属性选择为 ImagePoints ，用于连接多个图像关键点坐标或者图像关键点坐标列表。

算子参数

- **输入数量/port_number**：决定该算子的输入端口 input_? 的数量。取值范围：[0,10] 。默认值：2 。
- **列表输入数量/list_port_number**：决定该算子的输入端口 input_list_? 的数量。取值范围：[0,10] 。默认值：2 。

数据信号输入输出

输入：

- **image_points_?**：
  - 数据类型：ImagePoints
  - 输入内容：图像关键点坐标数据
- **image_points_list?**：
  - 数据类型：ImagePointsList
  - 输入内容：图像关键点坐标列表数据

输出：

- **image_points_list**：
  - 数据类型：PointCloudList
  - 输出内容：连接后的图像关键点坐标列表数据

功能演示

本节将使用 Concatenate 算子中 ImagePoints ，连接 2 个图像关键点坐标组成一个图像关键点坐标列表。

这与 Concatenate 算子中 JointArray 属性的连接2个机器人关节弧度值方法相同，请参照该章节的功能演示。

Selector 选择元素

Selector 算子为选择元素。将多条控制信号流和数据流通过逻辑或的方式并联在一起，合并后再触发后续算子。在 start 被触发的基础上，任意一个 input _? 端口被触发，都会触发 finished 端口，对应的 input _? 也会被触发并输出对应数据。适用于：Cube、Cylinder、Image 、PointCloud 、PoseString 。

类型	功能
Cube	立方体选择，将多条立方体数据合并在一起，根据触发 input_? 端口输出对应的立方体数据。
Cylinder	圆柱体选择，将多条圆柱体数据合并在一起，根据触发 input_? 端口输出对应的圆柱体数据。
Image	图像选择，将多条图像数据合并在一起，根据触发 input_? 端口输出对应的图像数据。
PointCloud	点云选择，将多条点云数据合并在一起，根据触发 input_? 端口输出对应的点云数据。
Pose	pose 选择，将多条 pose 数据合并在一起，根据触发 input_? 端口输出对应的 pose 数据。
String	字符串选择，将多条 string 数据合并在一起，根据触发 input_? 端口输出对应的字符串数据。

Cube

将 Selector 算子的 **类型** 属性选择 Cube，用于立方体选择，将多条立方体数据合并在一起，根据触发 input_? 端口输出对应的立方体数据。

算子参数

- **是否为列表/is_list**：数据类型为：bool 。设置是否将算子输入输出端口变为列表形式。
  - True：元素类型变为 cube_list。
  - False：元素类型为 cube。
- **输入数量/number_input**：决定该算子的输入端口 input_? 的数量。范围：[1,10] 。默认值：2。
- **立方体/cube**：设置立方体在 3D 视图中的可视化属性。
  -  打开立方体可视化。
  -  关闭立方体可视化。
  -  设置立方体的颜色。取值范围：[-2,360] 。默认值：-2 。
  -  设置立方体的透明度。取值范围：[0,1] 。默认值：0.5 。

控制信号输入输出

- input_？
    - 任意一个输入 input_？端口被触发，都会触发对应 output_? 输出端口。

数据信号输入输出

输入：

- `cube_?`：
    - 数据类型：Cube
    - 输入内容：立方体数据

输出：

- `cube`：
    - 数据类型：Cube
    - 输出内容：立方体数据

功能演示

使用 Selector 算子中 Cube 进行图像选择。将 2 条立方体数据合并在一起，根据触发 input_? 端口输出对应的cube数据。

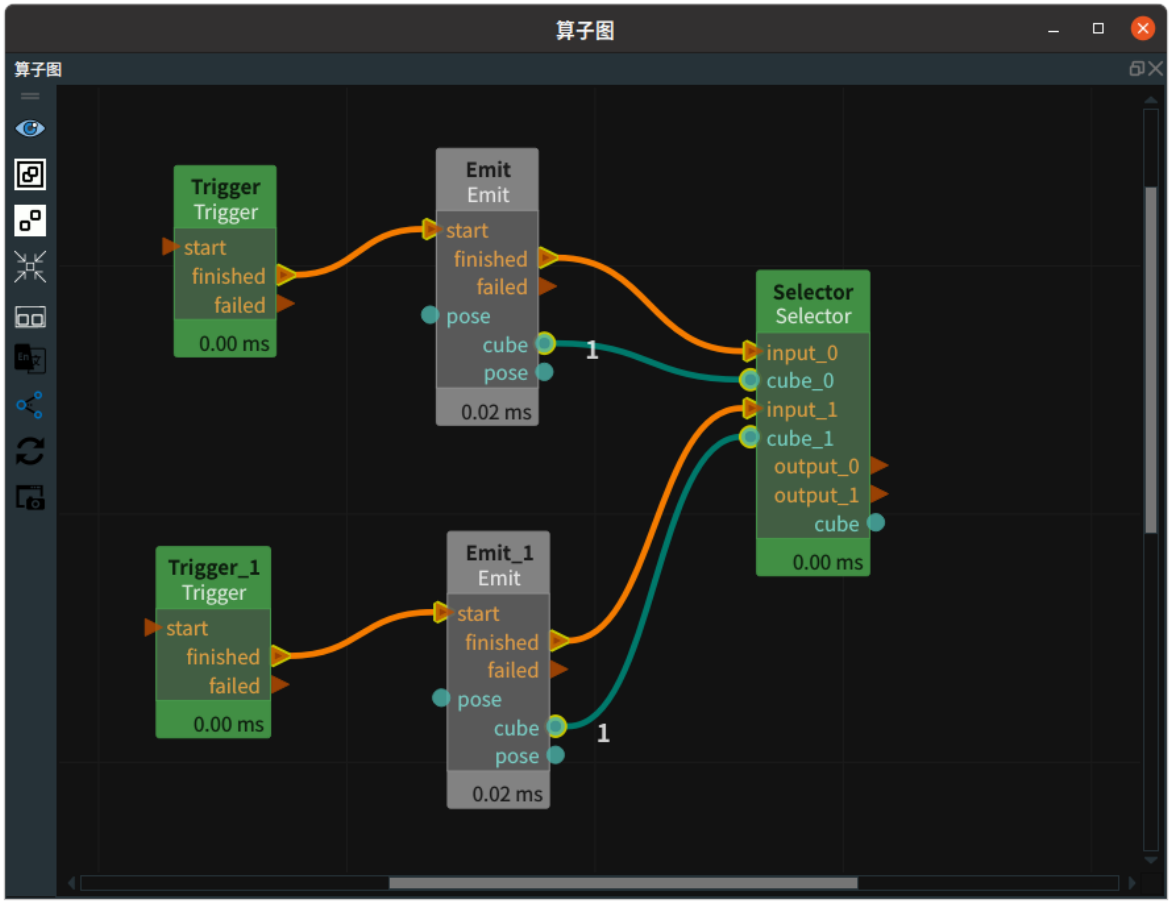
步骤1：算子准备

添加 Trigger （2个）、Emit（2个） 、Selector 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：type → Cube
2. 设置 Emit_1 算子参数：
    - 类型 → Cube
    - 宽度 → 0.3
3. 设置 Selecor 算子参数：
    - 类型 → Cube
    - 输入数量 → 2
    - 立方体 →  可视

步骤3：连接算子

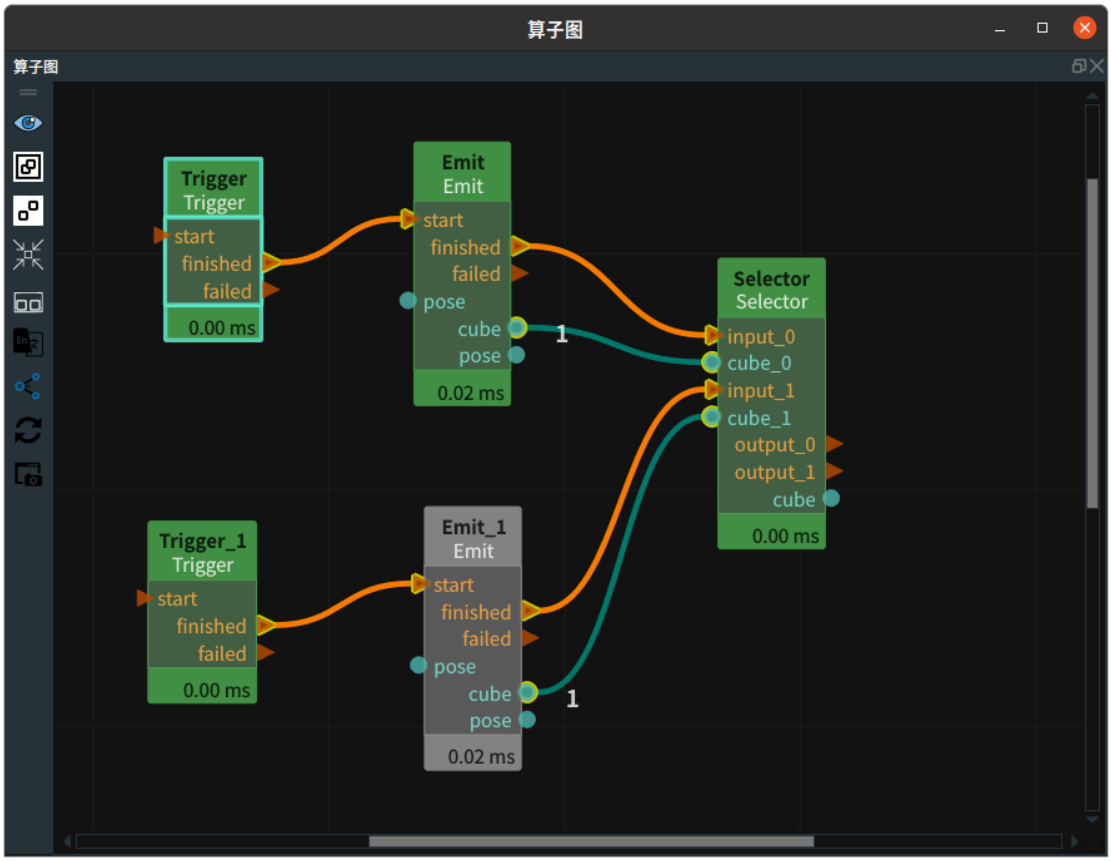


步骤4：运行

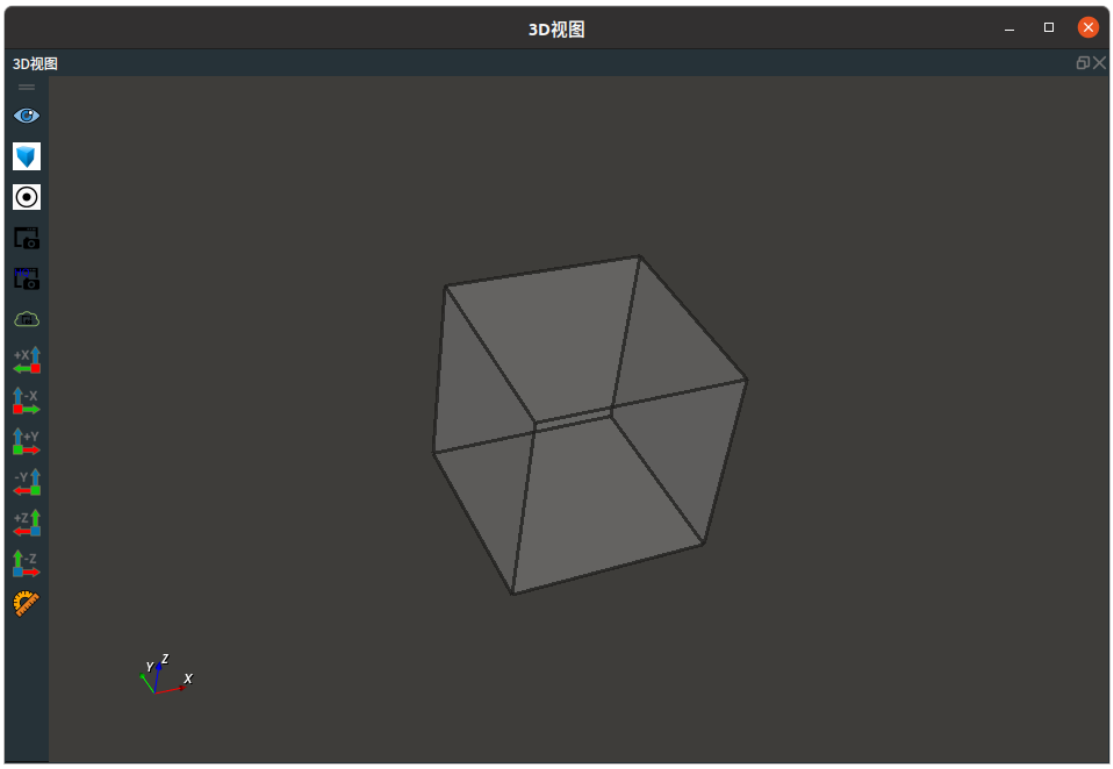
点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

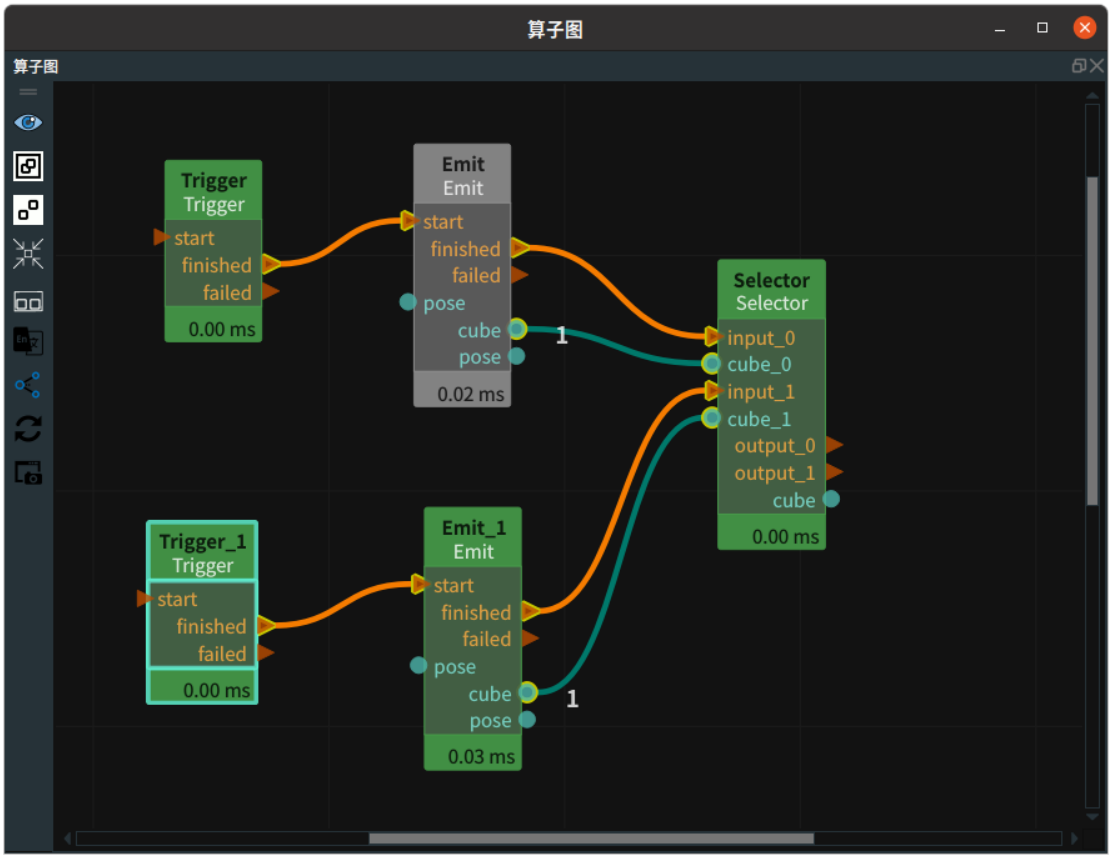
1. 当触发 Trigger 算子，会触发 input_0 端口，算子图运行如下：



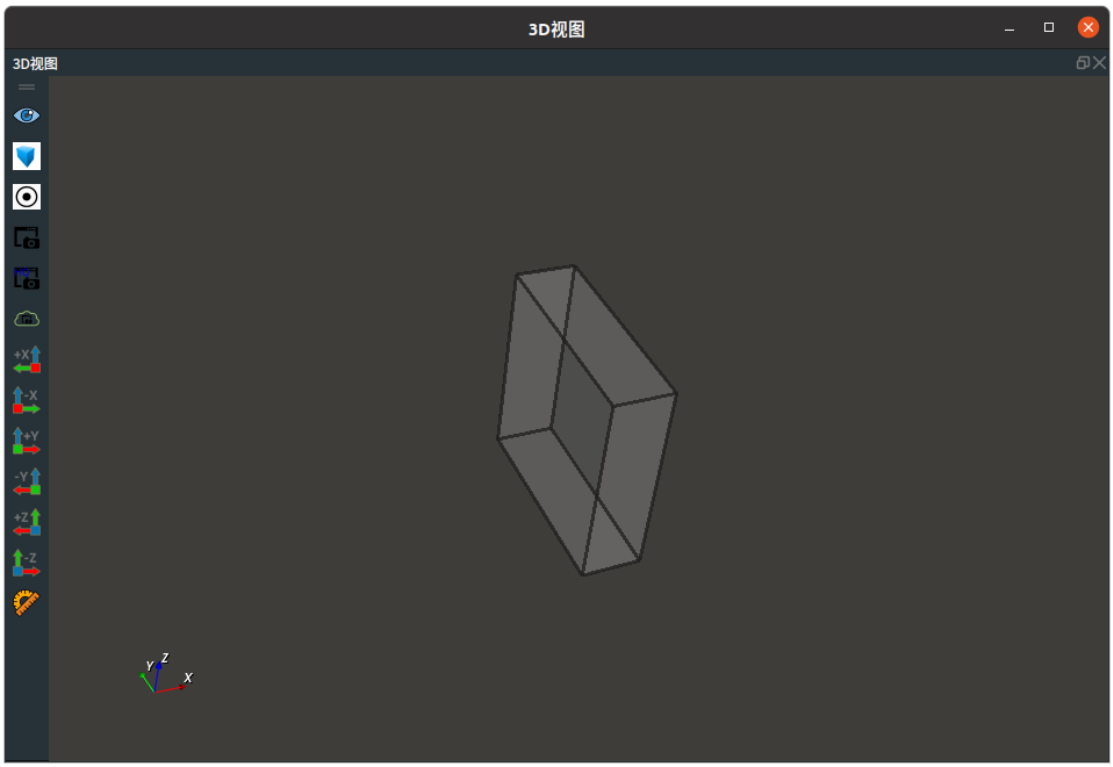
2. 结果如下图所示，3D 视图中显示 image_0 端口的立方体数据：



3. 当触发 Trigger 算子，会触发 input_1 端口，算子图运行如下：



4. 结果如下图所示，3D 视图中显示 image_1 端口的立方体数据：



Cylinder

将 Selector 算子的 **类型** 属性选择 Cylinder，用于圆柱体选择，将多条圆柱体数据合并在一起，根据触发 input_? 端口输出对应的圆柱体数据。

算子参数

- **是否为列表/is_list**：数据类型为：bool 。设置是否将算子输入输出端口变为列表形式。
  - True：元素类型变为 cylinder_list。
  - False：元素类型为 cylinder。
- **输入数量/number_input**：决定该算子的输入端口 input_? 的数量。范围：[1,10] 。默认值：2。
- **圆柱体/Cylinder**：设置圆柱体在 3D 视图中的可视化属性。
  - 打开圆柱体可视化。
  - 关闭圆柱体可视化。
  - 设置圆柱体的颜色。取值范围：[-2,360] 。默认值：-2 。

控制信号输入输出

- input_？
  - 任意一个输入 input_？端口被触发，都会触发对应 output_? 输出端口。

数据信号输入输出

输入：

- **cylinder_?**：
  - 数据类型：Cylinder
  - 输入内容：圆柱体数据

输出：

- **cylinder**：

- 数据类型：Cylinder
- 输出内容：圆柱体数据

功能演示

本节将使用 Selector 算子中 Cylinder ，将 2 条圆柱体数据合并在一起，根据触input_？端口输出对应的圆柱体数据。

这与 Selector 算子中 Cube 属性将 2 条立方体数据合并在一起，根据触发 input_? 端口输出对应的立方体数据的方法相同，请参照该章节的功能演示。

Image

将 Selector 算子的 type 属性选择 Image ，用于图像选择，将多条图像数据合并在一起，根据触发 input_? 端口输出对应的图像数据。

算子参数

- **是否为列表/is_list**：数据类型为：bool 。设置是否将算子输入输出端口变为列表形式。
  - True：元素类型变为 image_list。
  - False：元素类型为 image。
- **输入数量/number_input**：决定该算子的输入端口 input_? 的数量。范围：[1,10] 。默认值：2。
- **图像/image**：设置图像在 2D 视图中的可视化属性。
  -  打开图像可视化。
  -  关闭图像可视化。

控制信号输入输出

- input_？
  - 任意一个输入 input_？端口被触发，都会触发对应 output_? 输出端口。

数据信号输入输出

输入：

- **image_?**：
  - 数据类型：Image
  - 输入内容：图像数据

输出：

- **image**：
  - 数据类型：Image
  - 输出内容：图像数据

功能演示

使用 Selector 算子中 Image 进行图像选择。将 2 条图像数据合并在一起，根据触发 input_? 端口输出对应的图像数据。

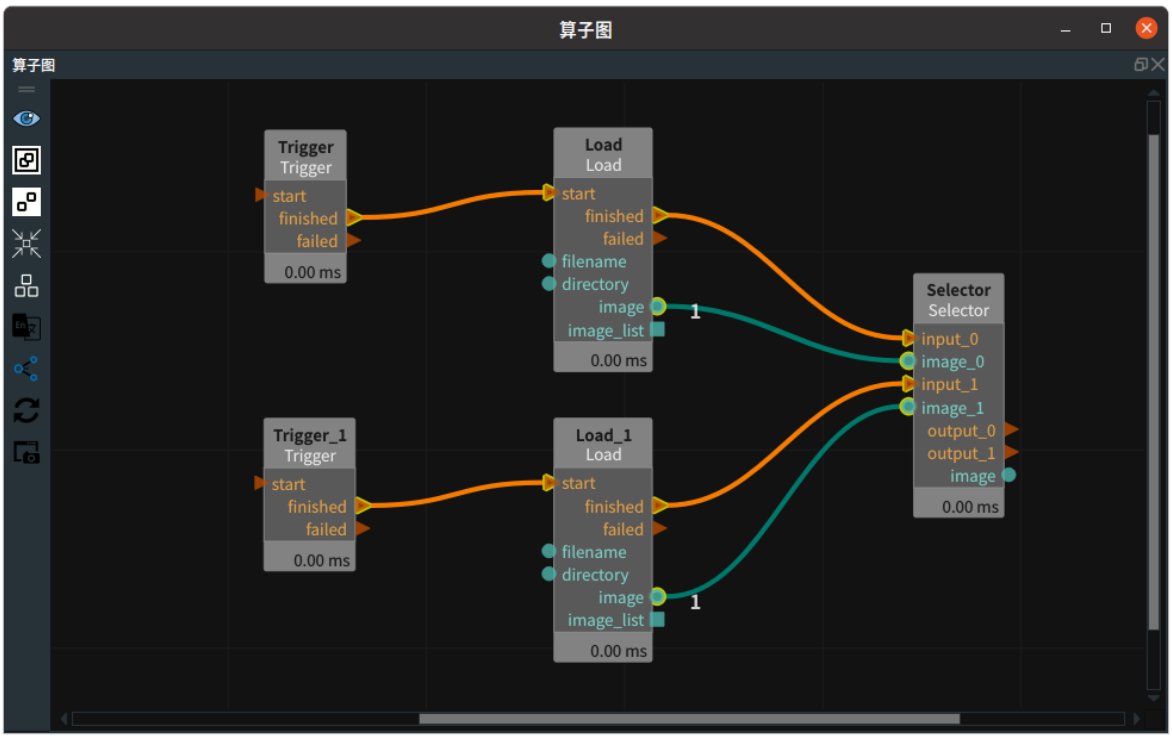
步骤1：算子准备

添加 Trigger （2个）、Load（2个） 、Selector 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 类型 → Image
    - 文件 → ••• → 选择 image 文件名（*example_data/images/bird.png*）
2. 设置 Load_1 算子参数：
    - 类型 → Image
    - 文件 → ••• → 选择 image 文件名（*example_data/images/cat.png*）
3. 设置 Selecor 算子参数：
    - 类型 → Image
    - 输入数量 → 2
    - 图像 → 可视

步骤3：连接算子

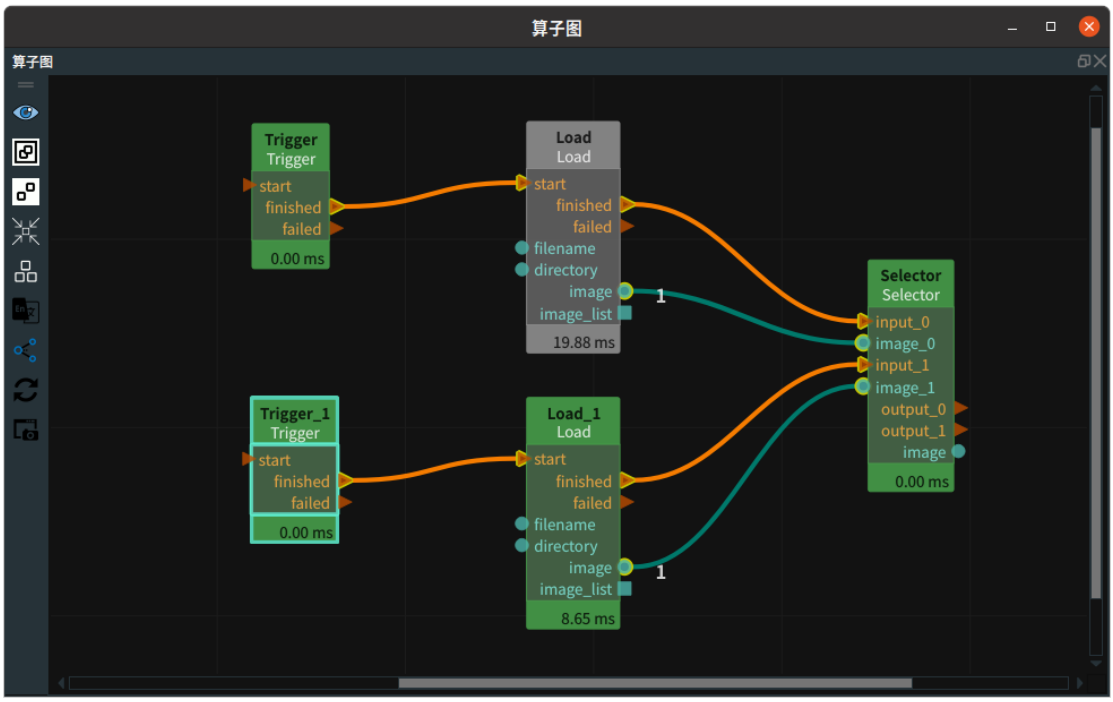


步骤4：运行

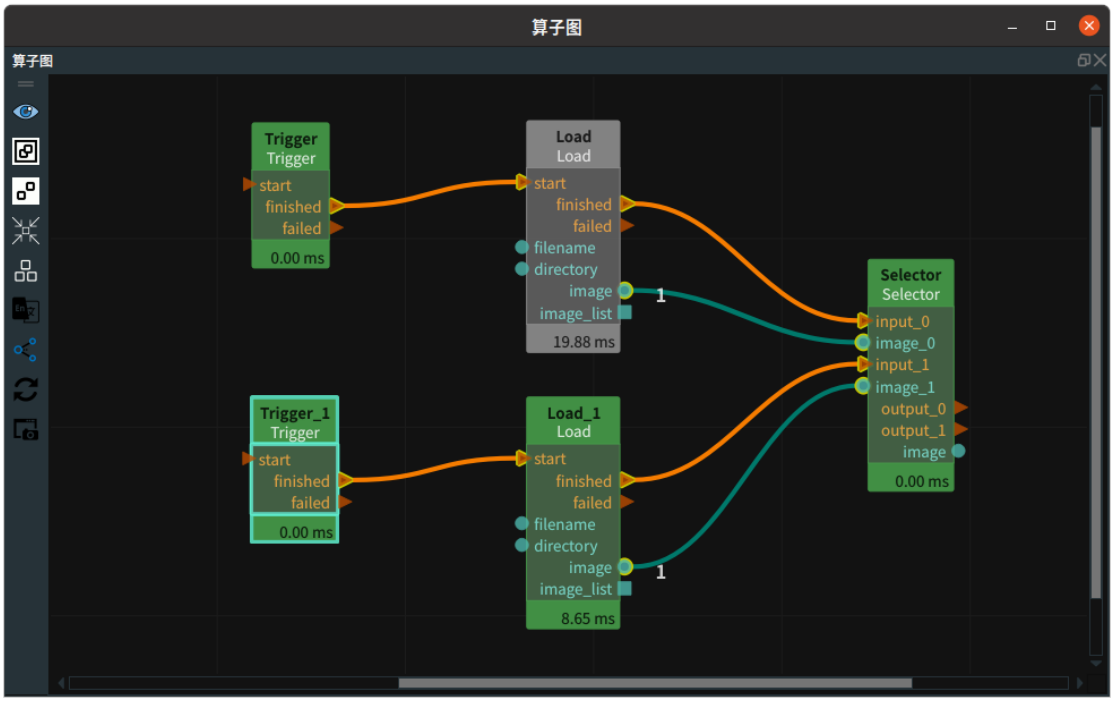
点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

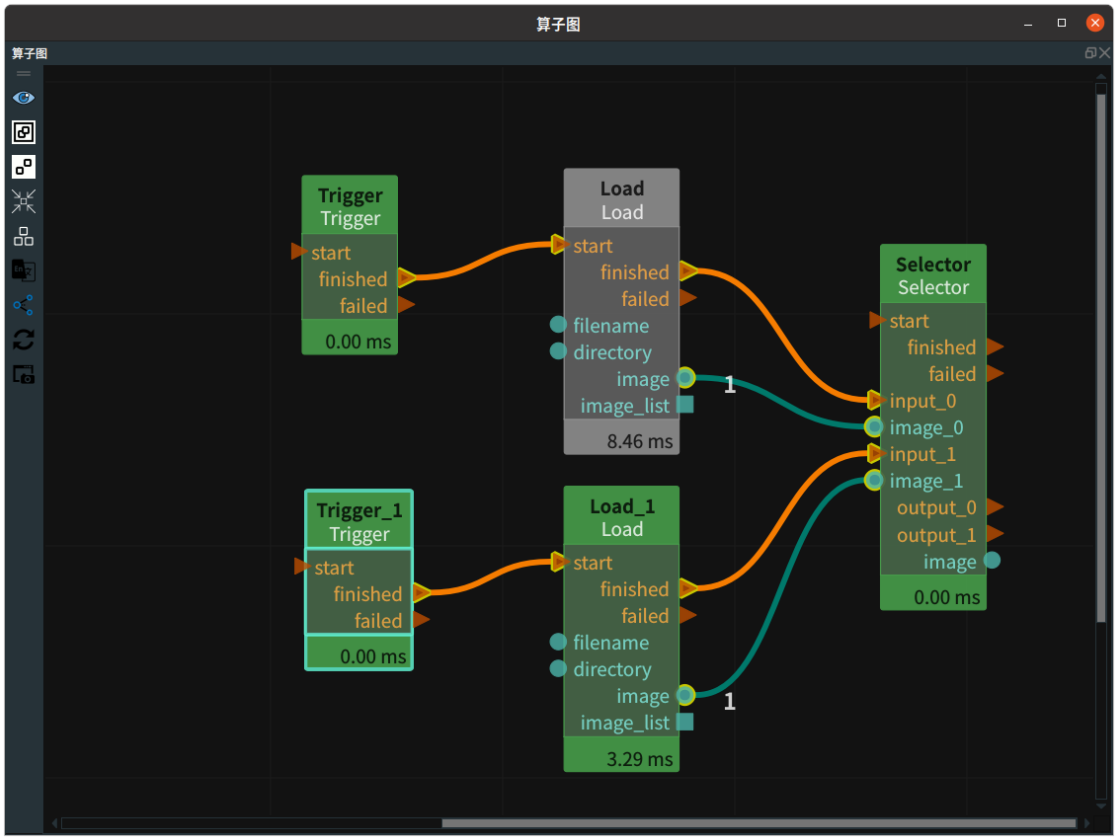
1. 当触发 Trigger 算子，会触发 input_0 端口，算子图运行如下：



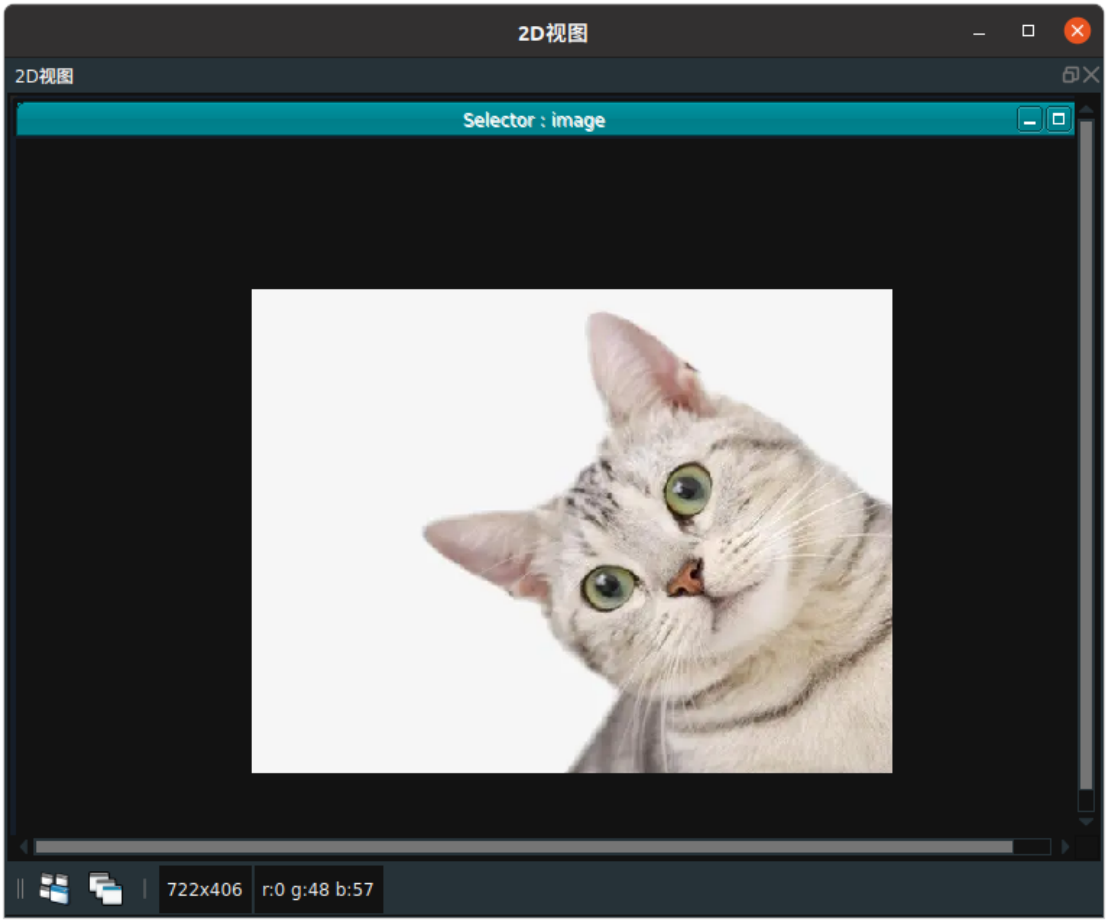
2. 结果如下图所示，2D 视图中显示 image_0 端口的图像数据：



3. 当触发 Trigger_1 算子，会触发 input_1 端口，算子图运行如下：



4. 结果如下图所示，2D 视图中显示 image_1 端口的图像数据：



PointCloud

将 Selector 算子的 **类型** 属性选择 PointCloud ，用于点云选择，将多条点云数据合并在一起，根据触发 input_? 端口输出对应的点云数据。

算子参数

- **是否为列表/is_list**：数据类型: bool 。设置是否将算子输入输出端口变为列表形式。
  - True：元素类型变为 cloud_list。
  - False：元素类型变为 cloud。
- **输入数量/number_input**：决定该算子的输入端口 input_? 的数量。范围：[1,10] 。默认值：2 。
- **点云/cloud**：设置点云在 3D 视图中的可视化属性。
  -  打开点云可视化。
  -  关闭点云可视化。
  -  设置3D视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  -  设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。

控制信号输入输出

- input_ ?：任意一个输入 input_ ? 端口被触发，都会触发对应 output_? 输出端口。

数据信号输入输出

输入：

- **cloud_?** ：
  - 数据类型：PointCloud
  - 输入内容：点云数据

输出：

- **cloud**：
  - 数据类型：PointCloud
  - 输出内容：点云数据

功能演示

本节将使用 Selector 算子中 PointCloud ，将 2 条点云数据合并在一起，根据触input_ ? 端口输出对应的点云数据。

这与 Selector 算子中 image 属性将 2 条图像数据合并在一起，根据触发 input_? 端口输出对应的图像数据的方法相同，请参照该章节的功能演示。

Pose

将 Selector 算子的 **类型** 属性选择 Pose ，用于 pose 选择，将多条 pose 数据合并在一起，根据触发 input_? 端口输出对应的 pose 数据。

算子参数

- **是否为列表/is_list**：数据类型： bool 。设置是否将算子输入输出端口变为列表形式。
  - True：元素类型变为 pose_list。
  - False：元素类型为 pose。
- **输入数量/number_input**：决定该算子的输入端口 input_? 的数量。范围：[1,10] 。默认值：2 。
- **坐标列表/pose**：设置 pose 在 3D 视图中的可视化属性。
  -  打开 pose 可视化。
  -  关闭 pose 可视化。
  -  设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

控制信号输入输出

- input_ ?：任意一个输入 input_ ? 端口被触发，都会触发对应 output_? 输出端口。

数据信号输入输出

输入：

- **pose_?** ：
  - 数据类型：Pose
  - 输入内容： pose 数据

输出：

- **pose**：
  - 数据类型：Pose
  - 输出内容： pose 数据

功能演示

本章节将使用 Selector 算子中 Pose ，将 2 条 pose 数据合并在一起，根据触发 input_ ? 端口输出对应的 pose 数据。

这与 Selector 算子中 image 属性将 2 条图像数据合并在一起，根据触发 input_ ? 端口输出对应的图像数据的方法相同，请参照该章节的功能演示。

String

将 Selector 算子的 **类型** 属性选择 String ，用于字符串选择，将多条 string 数据合并在一起，根据触发 input_? 端口输出对应的 字符串数据。

算子参数

- **是否为列表/is_list**：数据类型: bool 。设置是否将算子输入输出端口变为列表形式。
  - True：元素类型变为 string_list。
  - False：元素类型变为 string。
- **输入数量/number_input**：决定该算子的输入端口 input_? 的数量。范围：[1,10] 。默认值：2 。
- **字符串/string**：设置字符串的曝光属性。可与交互面板中输出工具“表格”、“文本框”空间进行绑定。
  -  打开字符串的曝光属性。
  -  关闭字符串的曝光属性。

控制信号输入输出

- input_ ?：任意一个输入 input_ ? 端口被触发，都会触发对应 output_? 输出端口。

数据信号输入输出

输入：

- `string_?`：
    - 数据类型：String
    - 输入内容： 字符串数据

输出：

- `string`：
    - 数据类型：String
    - 输出内容：字符串数据

功能演示

本章节将使用 Selector 算子中 String ，将 2 条字符串数据合并在一起，根据触发 input_ ? 端口输出对应的字符串数据。

这与 Selector 算子中 image 属性将 2 条图像数据合并在一起，根据触发 input_ ? 端口输出对应的图像数据的方法相同，请参照该章节的功能演示。

EventGate 信号开关

EventGate 算子为信号开关， 用于输出不同的信号状态。

算子参数

- `开关激活/gate_active`：数据类型： bool 。信号开关。
  - True：触发 finish 端口。
  - False：触发 failed 端口。

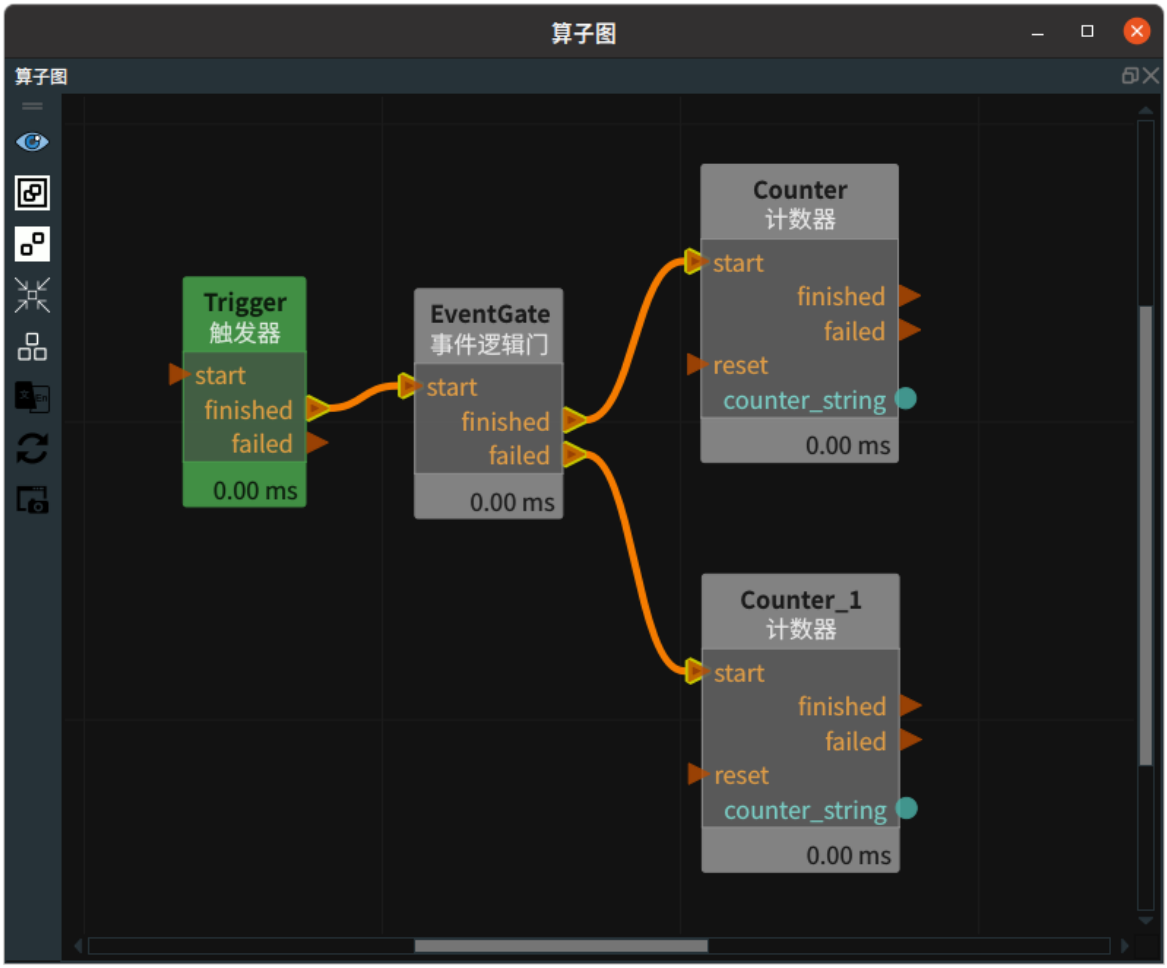
功能演示

使用 EventGate 算子输出不同的信号状态，分别触发 finished 或者 failed 端口，查看 Counter 和 Counter_1 算子的触发状态。

步骤1：算子准备

添加 Trigger 、EventGate、Counter（2 个）算子至算子图。

步骤2：连接算子

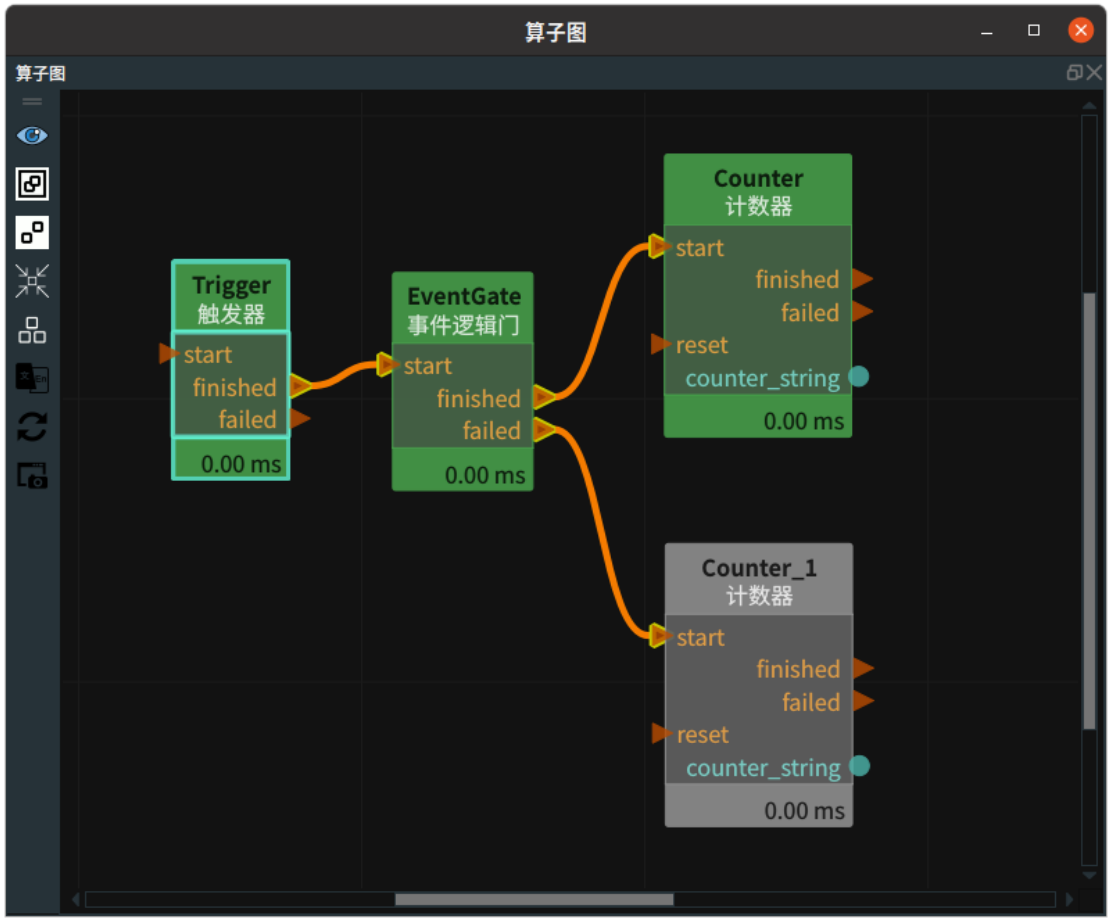


步骤3：运行

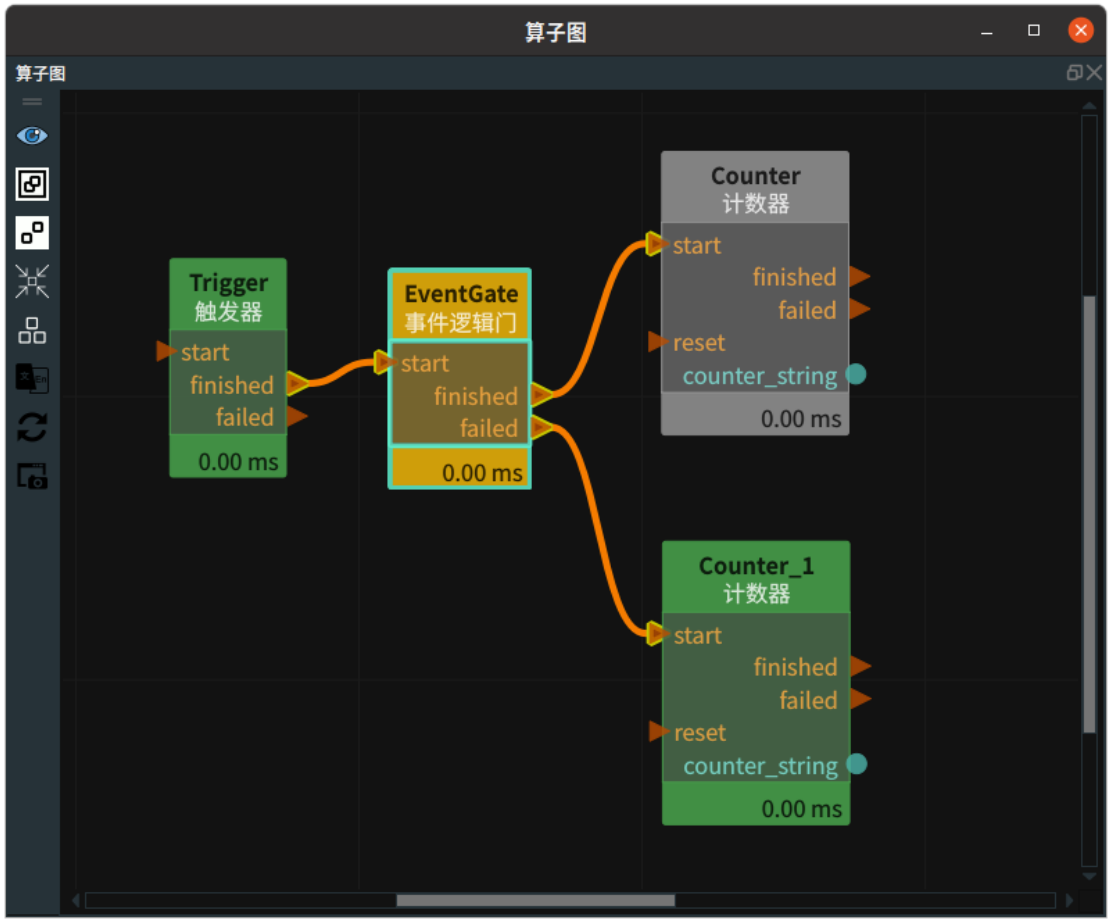
点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

1. 开关激活 → True ： 算子触发 finished 端口，Counter 算子运行。



2. 开关激活 → False： 算子触发 failed 端口，Counter_1 算子运行。



Buffer 缓冲元素

Buffer 算子为缓冲元素，可以暂时存放数据。 适用类型有：Cube 、Image 、JointArray 、PointCloud 、PolyData 、Pose 。

类型	功能
Cube	缓冲立方体。
Image	缓冲图像。
JointArray	缓冲机器人关节弧度值。
PointCloud	缓冲点云。
PolyData	缓冲 3D 模型。
Pose	缓冲坐标。

Cube

将 Buffer 算子的 **类型** 选择 Cube ，用于缓冲立方体。

算子参数

- **是否为列表/is_list**：数据类型： bool 。是否将算子数据输入输出端口变为列表形式。
  - True：元素类型变为 cube_list。
  - False：元素类型为 cube。
- **立方体/cube**：设置当前输入的立方体在 3D 视图中的可视化属性。
  -  打开立方体可视化。
  -  关闭立方体可视化。
  -  设置立方体的颜色。取值范围：[-2,360] 。默认值：-2 。
  -  设置立方体的透明度。取值范围：[0,1] 。默认值：0.5 。
- **上一个立方体/last_cube**：设置缓冲的立方体在 3D 视图中的可视化属性。参数值描述与 **立方体** 一致。

数据信号输入输出

输入：

- **cube**：
  - 数据类型：Cube
  - 输入内容：立方体数据

输出：

- **cube**：
  - 数据类型：Cube
  - 输出内容：当前输入的立方体数据
- **last_cube**：
  - 数据类型：Cube

- 输出内容：缓冲的立方体数据

功能演示

使用 Buffer 算子中 Cube ，缓冲上次加载的立方体，正常加载本次的立方体。

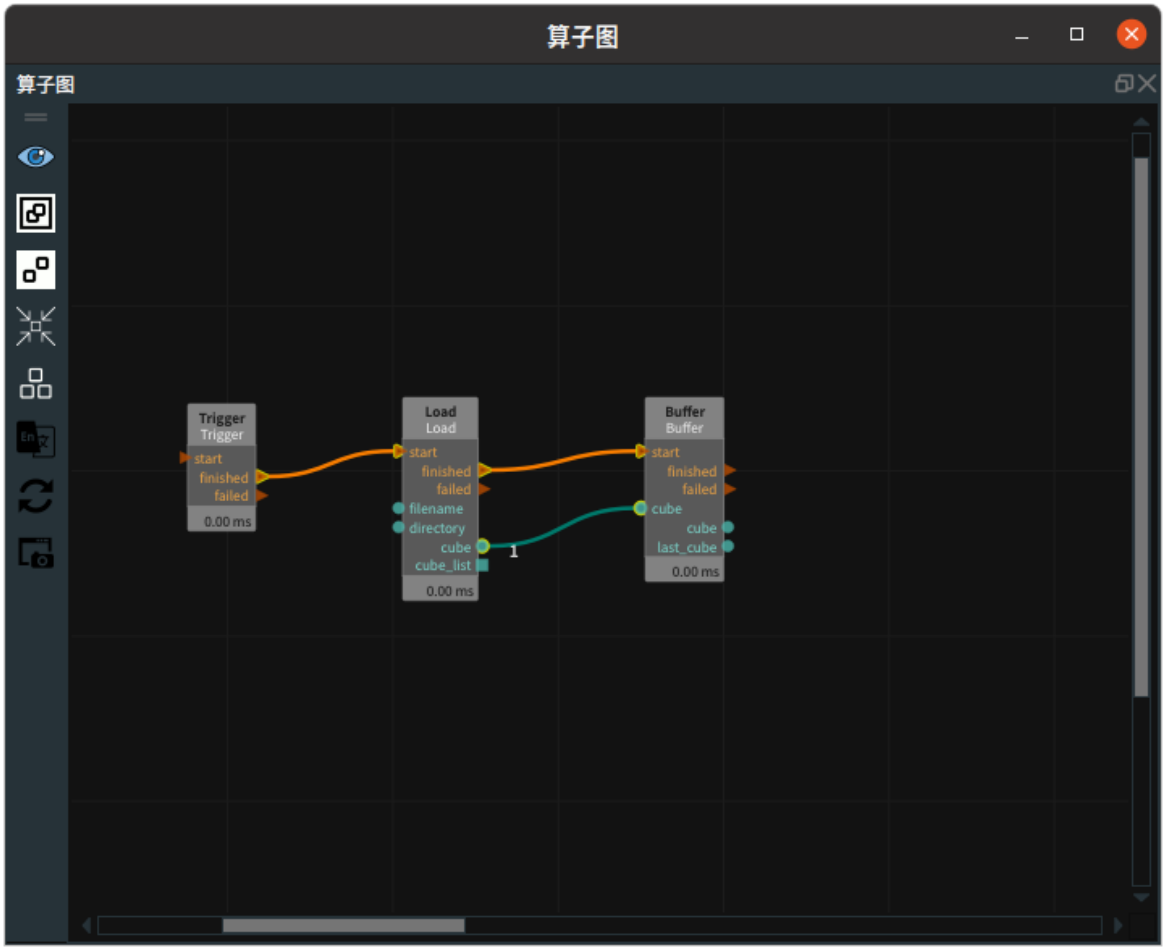
步骤1：算子准备

添加 Trigger、Load、Buffer 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → Cube
   - 文件 → ••• →选择cube 文件名（ *example_data/cube/cube.txt* ）
2. 设置 Buffer 算子参数：
   - 类型 → Cube
   - 立方体 →  可视
   - 上一个立方体 →  可视 →  200

步骤3：连接算子

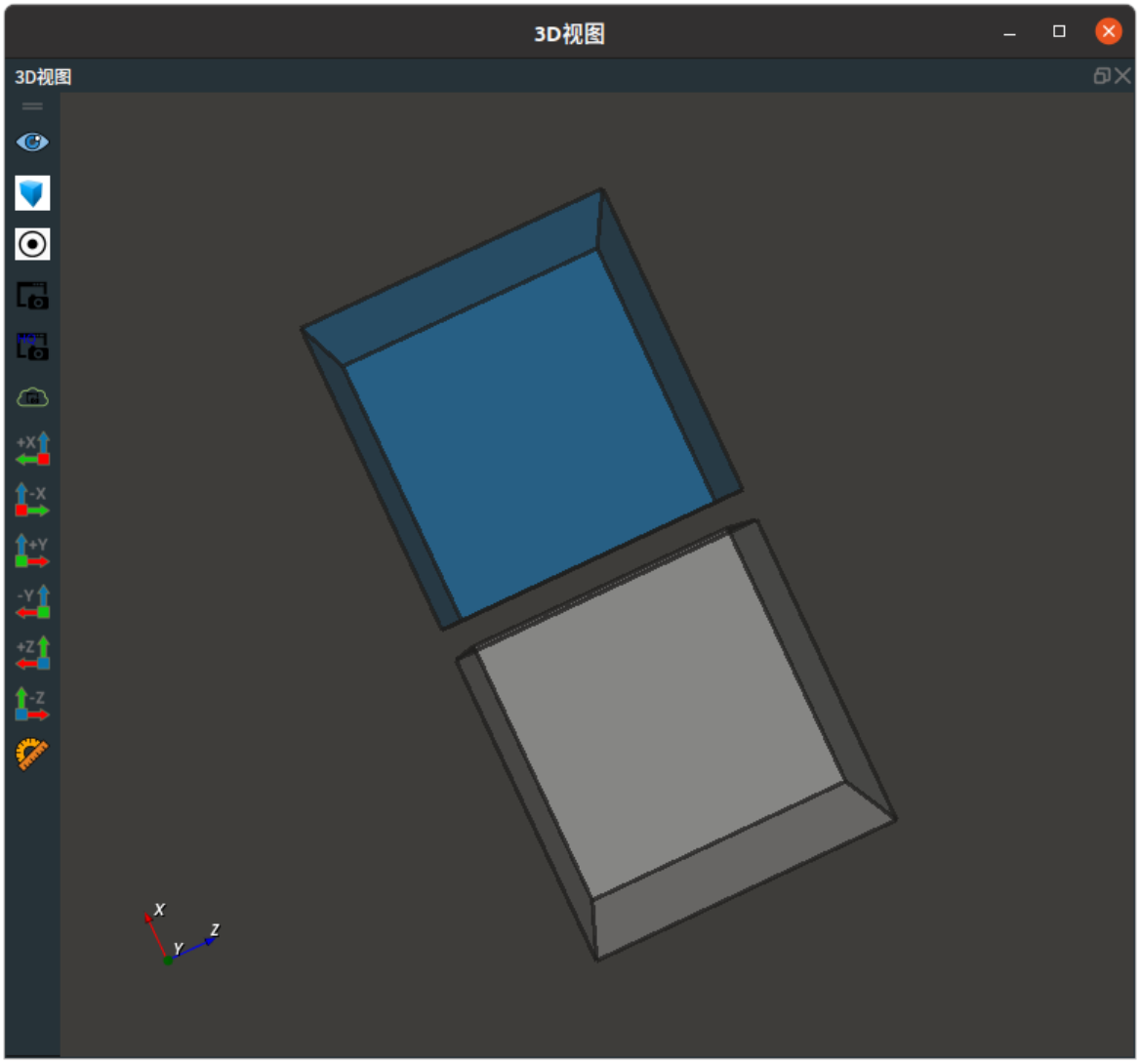


步骤4：运行

1. 点击运行按钮，触发 Trigger 算子。
2. 重新选择 Load 算子中 filename 属性 → example_data/cube/cube1.txt，再次触发Trigger。

运行结果

结果显示如下，在 3D 视图中显示上次加载的 last_cube 缓冲数据和本次加载的 cube 数据。蓝色立方体： last_cube 。



Image

将 Buffer 算子的 **类型** 选择 Image ，用于缓冲图像。

算子参数

- **是否为列表/is_list**：数据类型为：bool 。设置是否将算子数据输入输出端口变为列表形式。
  - True：元素类型变为 image_list。
  - False：元素类型为 image 。
- **图像/image**：设置当前输入的图像在 2D 视图中的可视化属性。
  - 打开图像可视化。
  - 关闭图像可视化。
- **上一个图像/last_image**：设置缓冲的图片在 2D 视图中的可视化属性。参数值描述与 **图像** 一致。

数据信号输入输出

输入：

- `image`：
  - 数据类型：Image
  - 输入内容：图像数据

输出：

- `image`：
  - 数据类型：Image
  - 输出内容：当前输入的图像数据
- `last_image`：
  - 数据类型：Image
  - 输出内容：缓冲的图像数据

功能演示

将 Buffer 算子的 type 选择 Image ，缓冲上次加载的图像，正常加载本次的图像。

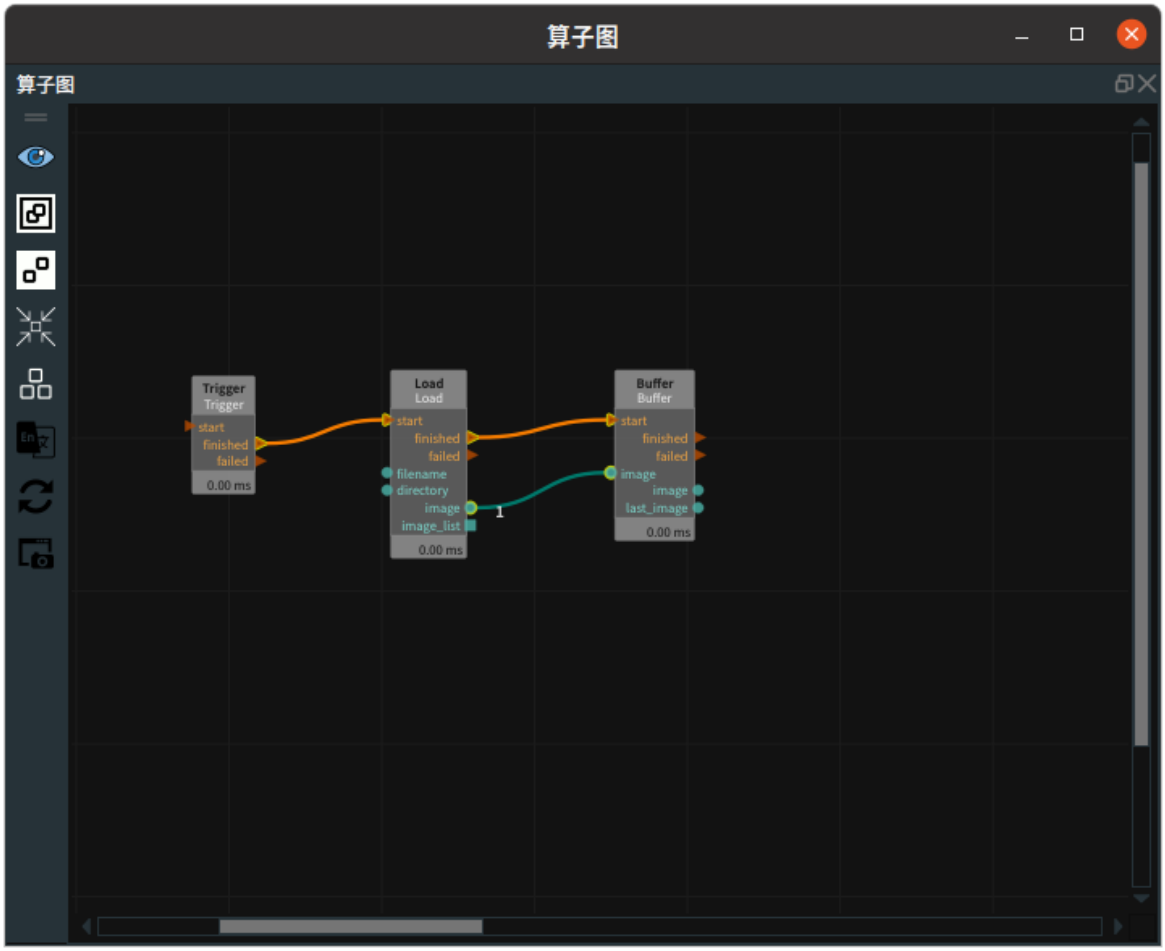
步骤1：算子准备

添加 Trigger 、Load 、Buffer 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → Image
   - 文件 → ••• → 选择图像文件名 （*example_data/images/cat.png*）
2. 设置 Buffer 算子参数：
   - 类型 → Image
   - 图像 →  可视
   - 上一个图像 →  可视

步骤3：连接算子

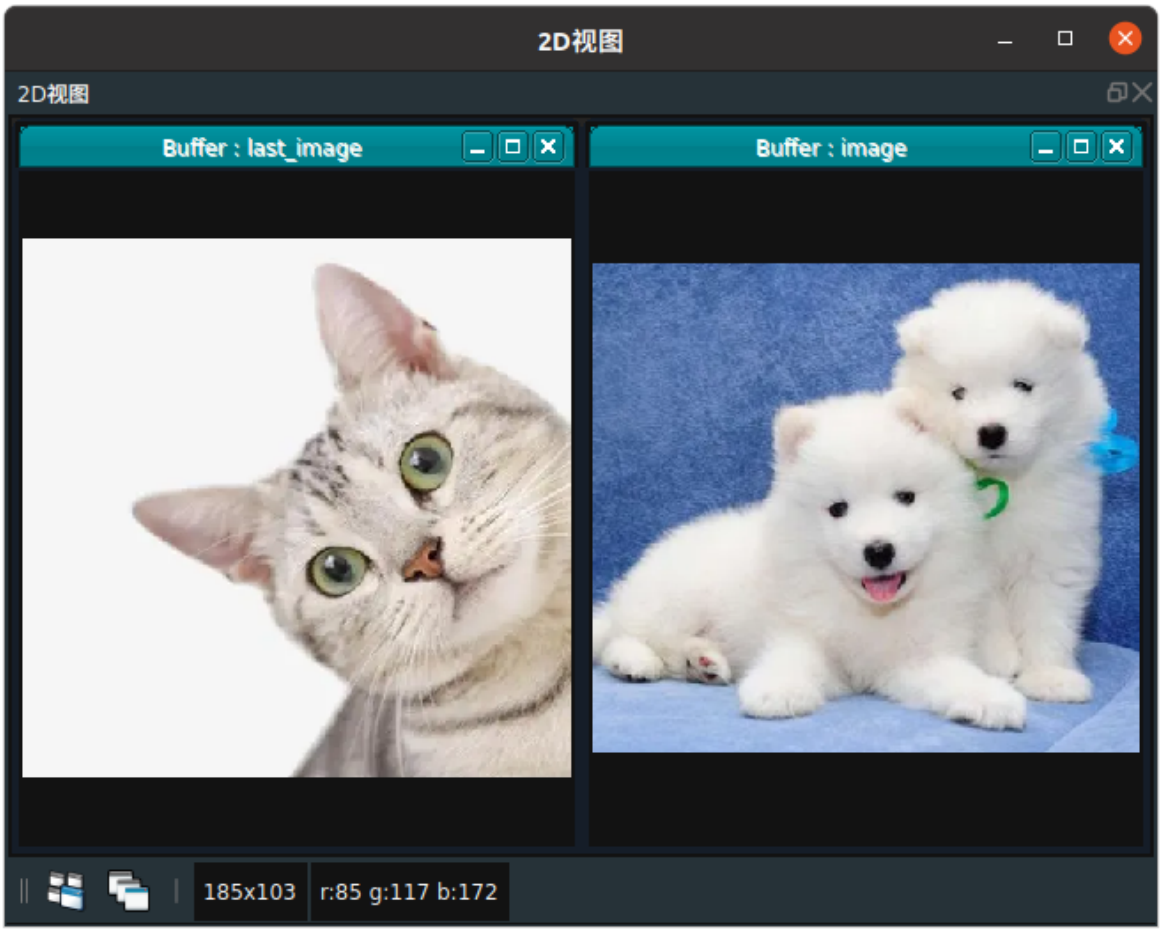


步骤4：运行

1. 点击运行按钮，触发 Trigger 算子。
2. 重新选择 Load 算子中 filename 属性 → example_data/images/dog .png。
3. 再次触发 Trigger 。

运行结果

结果显示如下，在 2D 视图显示上次加载的 last_image 缓冲图像和本次加载的 image 图像。



JointArray

将 Buffer 算子的 **类型** 选择 JointArray ，用于缓冲机器人关节弧度值。

算子参数

- **是否为列表/is_list**：数据类型： bool 。设置是否将算子数据输入输出端口变为列表形式。
  - True：元素类型变为 joint_list 。
  - False：元素类型为 joint。

数据信号输入输出

输入：

- **joint**：
  - 数据类型：JointArray
  - 输入内容：机器人关节弧度值数据 J0 J1 J2 J3 J4 J5

输出：

- **joint**：
  - 数据类型：JointArray
  - 输出内容：当前输入的机器人关节弧度值数据
- **last_joint**：
  - 数据类型：JointArray
  - 输出内容：缓冲的机器人关节弧度值数据

功能演示

使用 Buffer 算子中 JointArray ，缓冲上次的机器人关节弧度值，正常加载本次的机器人关节弧度值。

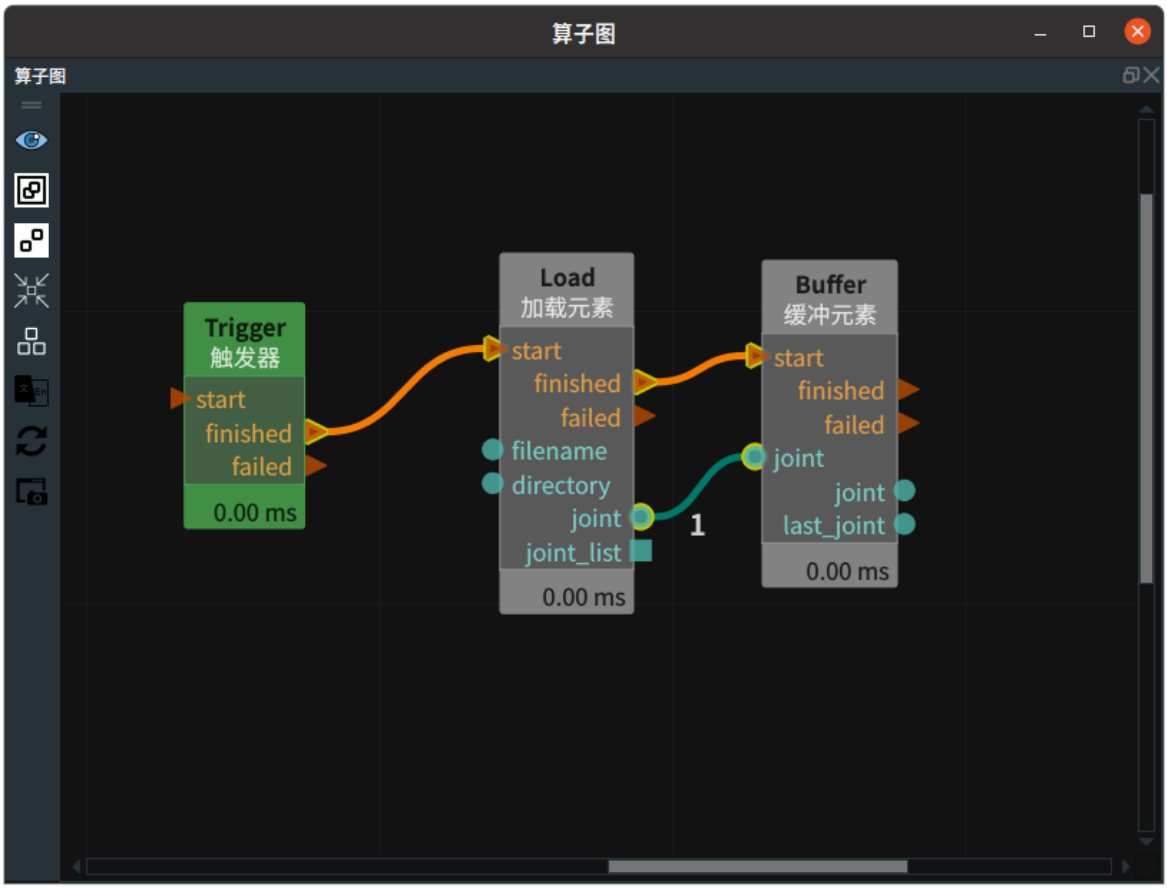
步骤1：算子准备

添加 Trigger 、Load 、Buffer 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - type → JointArray
    - filename → ●●● →选择机器人关节值文件名（*example_data/joints/ joints.txt*)
2. 设置 Buffer 算子参数：
    - type → JointArray

步骤3：连接算子



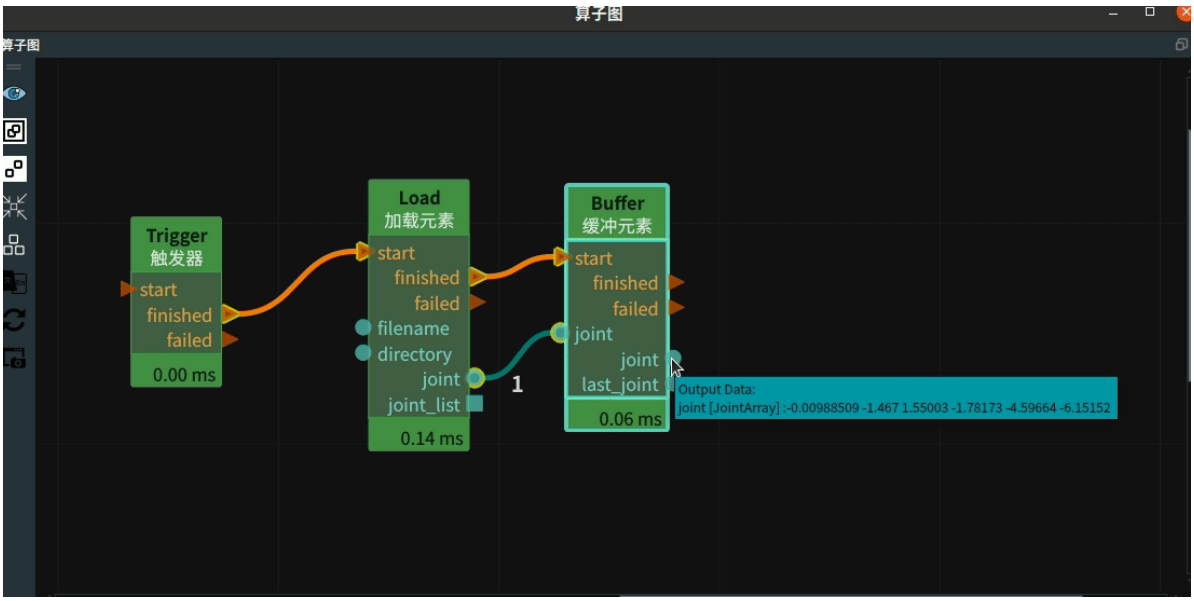
步骤4：运行

1. 点击 RVS 运行按钮，触发 Trigger 算子。
2. 重新选择 Load 算子中 filename 属性→example_data/joints/ joints1.txt，再次触发 Trigger。

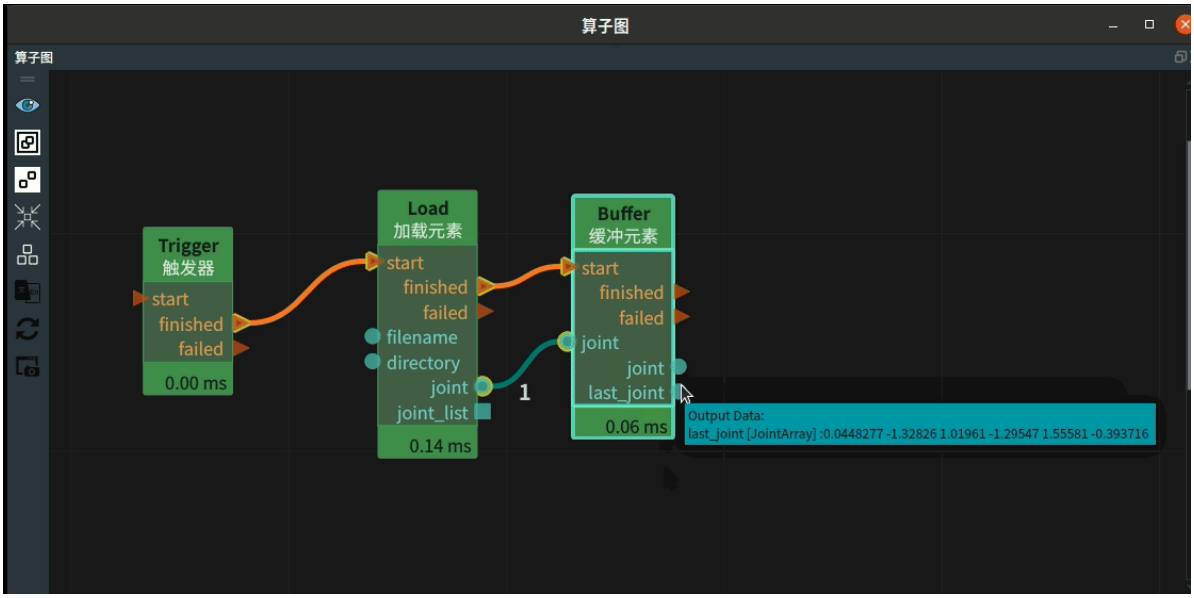
运行结果

如下图所示，上次加载的 last_joint 缓冲和本次加载的 joint ，数据端口输出显示如下。

- 鼠标放置在 joint 处，数据端口输出显示如下所示。



- 鼠标放置在 last_joint处，数据端口输出显示如下所示。



PointCloud

将 Buffer 算子的 **类型** 选择 PointCloud ，用于缓冲点云。

算子参数

- **是否为列表/is_list**：数据类型： bool 。设置是否将算子输入输出端口变为列表形式。
  - True：元素类型变为 cloud_list。
  - False：元素类型为 cloud。
- **点云/cloud**：设置当前输入点云在 3D 视图中的可视化属性。
  - 打开点云可视化。
  - 关闭点云可视化。
  - 设置 3D 视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  - 设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。
- **上一个点云/last_cloud**：设置缓冲点云在 3D 视图中的可视化属性。参数值描述与 **点云** 一致。

数据信号输入输出

输入：

- `cloud`：
  - 数据类型：PointCloud
  - 输入内容：点云数据

输出：

- `cloud`：
  - 数据类型：PointCloud
  - 输出内容：当前输入的点云数据
- `last_cloud`：
  - 数据类型：PointCloud
  - 输出内容：缓冲的点云数据

功能演示

本节将使用 Buffer 算子中 PointCloud ，缓冲上次加载的点云，正常加载本次点云。这与 Buffer 算子中 Cube 属性的缓冲立方体的方法相同，请参照该章节的功能演示。

PolyData

将 Buffer 算子的 `类型` 选择 PolyData ，用于缓冲 3D 模型。

算子参数

- `是否为列表/is_list`：数据类型： bool 。设置是否将算子数据输入输出端口变为列表形式。
  - True：元素类型变为 polydata_list。
  - False：元素类型为 polydata。
- `多边形/polydata`：设置当前输入的 3D 模型在 3D 视图中的可视化属性。
  -  打开 3D 模型可视化。
  -  关闭 3D 模型可视化。
- `上一个多边形/last_polydata`：设置缓冲的 3D 模型在 3D 视图中的可视化属性。参数值描述与 `多边形` 一致。

数据信号输入输出

输入：

- `polydata`：
  - 数据类型：PolyData
  - 输入内容：3D 模型数据

输出：

- `polydata`：
  - 数据类型：PolyData
  - 输出内容：当前输入的 3D 模型数据
- `last_polydata`：

- 数据类型：PolyData
- 输出内容：缓冲的 3D 模型数据

功能演示

本节使用 Buffer 算子中 PolyData ，缓冲加载上次的 3D 模型，正常加载本次的 3D 模型。这与 Buffer 算子中 Cube 属性的缓冲立方体的方法相同，请参照该章节的功能演示。

Pose

将 Buffer 算子的 **类型** 选 Pose ，用于缓冲坐标。

算子参数

- **是否为列表/is_list**：数据类型： bool 。设置是否将算子数据输入输出端口变为列表形式。
  - True：元素类型变为 pose_list 。
  - False：元素类型为 pose 。
- **坐标/pose**：设置当前输入的 pose 在 3D 视图中的可视化属性。
  -  打开 pose 可视化。
  -  关闭 pose 可视化。
  -  设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。
- **上一个坐标/last_pose**：设置缓冲的 pose 在 3D 视图中的可视化属性。参数值描述与 **坐标** 一致。

数据信号输入输出

输入：

- **pose**：
  - 数据类型：Pose
  - 输入内容：pose 数据

输出：

- **pose**：
  - 数据类型：Pose
  - 输出内容：当前输入的 pose 数据
- **last_pose**：
  - 数据类型：Pose
  - 输出内容：缓冲的 pose 数据

功能演示

本节将使用 Buffer 算子中 Pose ，缓冲加载上次的 pose 数据，正常加载本次的 pose 数据。这与 Buffer 算子中 Cube 属性的缓冲立方体的方法相同，请参照该章节的功能演示。

Trigger 触发器

Trigger 算子用于启动其他算子，包含 Trigger 、InitTrigger 、DemultiplexerTrigger 。

类型	功能
Trigger	分为单触发（ trigger ）和循环触发（ loop ）两种工作模式。
InitTrigger	当程序刚打开，首次进入 running 状态后，无需人为设置其 trigger 属性，会自动触发一次。
DemultiplexerTrigger	信号分支器；根据输入的 string 信号不同，来触发后续不同的分支。

Trigger

Trigger 分为单触发（ trigger ）和循环触发（ loop ）两种工作模式。

算子参数

- **触发器/trigger**：数据类型：bool 。
  - True：当程序处于运行模式下，则其 finished 端口会被触发。之后，trigger 的值会自动重置为 False 。
  - False：不触发Trigger时状态为False。
- **激活/active**：数据类型：bool 。
  - True：勾选为 True 后可以触发 Trigger 。
  - False：单触发和循环触发都会失效。
- **循环/loop**：数据类型为 bool 。
  - True：循环给出触发信号。如果程序顺利执行完一次后续节点图，且中途没有遇到错误中断，循环往复，直到遇到错误或者结束运行模式为止。
  - False：关闭循环触发。
- **计数器/counter**：固定自动触发次数。

InitTrigger

InitTrigger 当程序刚打开，首次进入运行状态后，无需人为设置其 trigger 属性，会自动触发一次。

算子参数

- **触发器/Trigger**：首次进入运行状态后，无需人为设置其 trigger 属性，它会自动触发一次。我们常常将 InitTrigger 作为“全局变量”节点的触发器。

DemultiplexerTrigger

DemultiplexerTrigger 信号分支器。根据输入的 string 信号不同，来触发后续不同的分支。

算子参数

- **number_output**：分支数量。取值范围：[1,10]。默认值：1。
- **选择器/selector**：数据类型： string 。其内容决定了输出端口所采用的实际数据。

PoseToElement 坐标转元素

PoseToElement 算子为坐标转元素，基于 pose 或 poselist 转成对应的数据类型或列表，适用于 Cube 、Cylinder 、Circle 、Object 、Path 、Sphere 、Text 。

类型	功能
Cube	Pose 转 Cube 。
Cylinder	Pose 转 Cylinder 。
Circle	Pose 转 Circle 。
Object	Pose 转 Object 。
Path	Pose_list 转 Path 。
Sphere	Pose 转 Sphere 。
Text	Pose 转 Text 。

Cube

将 PoseToElement 算子的 **类型** 属性选择 Cube ，用于将 Pose 转 Cube 。

算子参数

- **宽度/width**：立方体的宽。pose 的 X 轴方向。
- **高度/height**：立方体的高。pose 的 Y 轴方向。
- **深度/depth**：立方体的长。pose 的 Z 轴方向。
- **立方体/cube**：设置立方体在 3D 视图中的可视化属性。
  -  打开立方体可视化。
  -  关闭立方体可视化。
  -  设置立方体的颜色。取值范围：[-2,360] 。默认值：-2 。
  -  设置立方体的透明度。取值范围：[1,10] 。默认值：0.5 。
- **立方体列表/cube_list**：设置立方体列表在 3D 视图中的可视化属性。参数值描述与 **立方体** 一致。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **pose**：
  - 数据类型：Pose
  - 输入内容：单个 pose 数据
- **pose_list**：
  - 数据类型：PoseList
  - 输入内容：pose 数据列表

输出：

- `cube`：
  - 数据类型：Cube
  - 输出内容：单个 cube 数据
- `cube_list`：
  - 数据类型：CubeList
  - 输出内容：cube 数据列表

功能演示

将 PoseToElement 算子的 `类型` 属性选择 Cube ，将 Pose 转成立方体元素 。

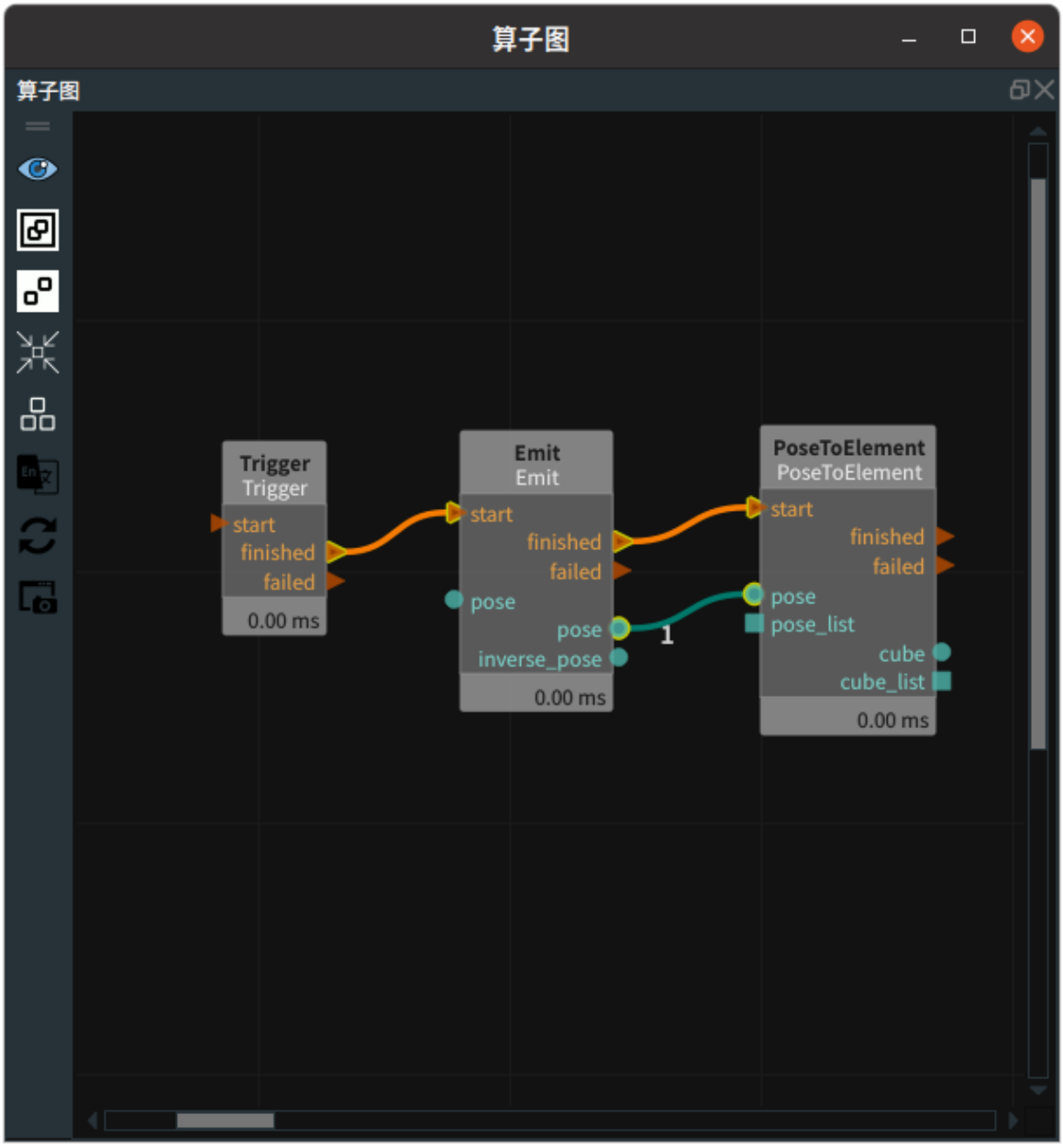
步骤1：算子准备

添加 Trigger 、Emit 、PoseToElement 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
   - 类型 → pose
   - 坐标 →0 0 0 0 0 0
2. 设置 PoseToElement 算子参数：
   - 类型 → Cube
   - 立方体→  可视

步骤3：连接算子

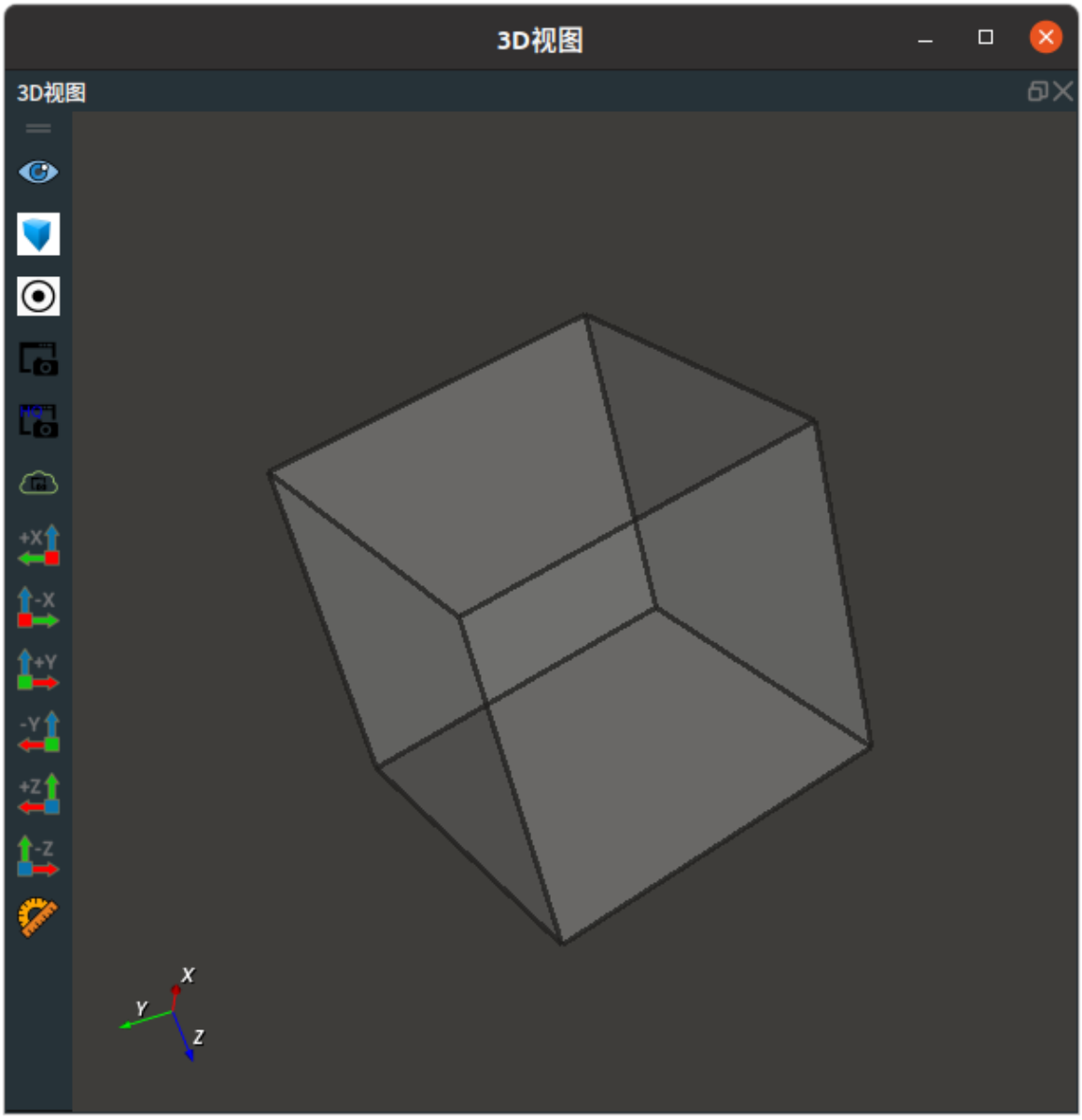


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，在 3D 视图中显示转成的立方体元素。



Cylinder

将 PoseToElement 算子的 **类型** 属性选择 Cylinder ，用于将 Pose 转 Cylinder 。

算子参数

- **半径/radius**：设置圆柱体的半径。
- **长度/length**：设置圆柱体的高。
- **圆柱体/cylinder**：设置圆柱体在 3D 视图中的可视化属性。
  - 打开圆柱体可视化。
  - 关闭圆柱体可视化。
  - 设置圆柱体的颜色。取值范围：[-2,360] 。默认值：-2 。
- **圆柱体列表/cylinder_list**：圆柱体列表在 3D 视图中的可视化属性。属性值描述与 **圆柱体** 一致。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- `pose`：
  - 数据类型：Pose
  - 输入内容：单个 pose 数据
- `pose_list`：
  - 数据类型：PoseList
  - 输入内容：pose 数据列表

输出：

- `cylinder`：
  - 数据类型：Cylinder
  - 输出内容：单个 cylinder 数据
- `cylinder_list`：
  - 数据类型：CylinderList
  - 输出内容：cylinder 数据列表

功能演示

本节将使用 PoseToElement 算子中 Cylinder ，将 Pose 转 Cylinder 元素。这与 PoseToElement 算子的 Cube 属性将 Pose 转 Cube的方法相同，请参照该章节的功能演示。

Circle

将 PoseToElement 算子的 `类型` 属性选择 Circle ，用于将 Pose 转 Circle 。

算子参数

- `半径/radius`：设置拟合圆的半径。
- `圆圈/circle`：设置拟合圆在 3D 视图中的可视化属性。
  -  打开拟合圆可视化。
  -  关闭拟合圆可视化。
  -  设置拟合圆的颜色。取值范围：[-2,360] 。默认值：-2。
- `圆圈列表/circle_list`：设置拟合圆列表在 3D 视图中的可视化属性。参数值描述与 `圆圈` 一致。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- `pose`：
  - 数据类型：Pose
  - 输入内容：单个 pose 数据
- `pose_list`：
  - 数据类型：PoseList

    - 输入内容：pose 数据列表

输出：

- `circle`：
    - 数据类型：Circle
    - 输出内容：单个 Circle 数据
- `circle_list`：
    - 数据类型：CircleList
    - 输出内容：Circle 数据列表

功能演示

本节将使用 PoseToElement 算子中 Circle ，将 Pose 转 Circle 元素。这与PoseToElement 算子的 Cube 属性将 Pose 转 Cube的方法相同，请参照该章节的功能演示。

Object

将 PoseToElement 算子的 `类型` 属性选择 Object ，用于将 Pose 转 Object 。

算子参数

- `文件/file`：输入或选择：3D 模型文件名（ obj 格式）。
- `物体/object`：设置 3D 模型在 3D 视图中的可视化属性。
    -  打开 3D 模型可视化。
    -  关闭 3D 模型可视化。
- `物体列表/object_list`：设置 3D 模型列表在 3D 视图中的可视化属性。参数值描述与 `物体` 一致。

数据信号输入输出

输入：

说明：pose 和 pose_list 根据需求选择其中一种数据信号输入即可。

- `pose`：
    - 数据类型：Pose
    - 输入内容：单个 pose 数据
- `pose_list`：
    - 数据类型：PoseList
    - 输入内容：pose 数据列表
- `filename`：
    - 数据类型：String
    - 输入内容：3D 模型文件名

输出：

- `object`：
    - 数据类型：Object
    - 输出内容：单个 object 数据
- `object_list`：
    - 数据类型：ObjectList

- 输出内容：object 数据列表

功能演示

本节将使用 PoseToElement 算子中 Object ，将 Pose 转 Object 元素。这与PoseToElement 算子的 Cube 属性将 Pose 转 Cube的方法相同，请参照该章节的功能演示。

Path

将 PoseToElement 算子的 **类型** 属性选择 Path ，用于将 Pose 转 Path 。

算子参数

- **path**：设置路径在 3D 视图中的可视化属性。
  -  打开路径可视化。
  -  关闭路径可视化。
- **path_line_width**：设置路径的线宽。默认值：1。

数据信号输入输出

输入：

- **pose_list**：
  - 数据类型：PoseList
  - 输入内容：pose 数据列表

输出：

- **path**：
  - 数据类型：Path
  - 输出内容：Path 数据

功能演示

使用 PoseToElement 算子中 Path ，将 Pose 转 Path 元素。

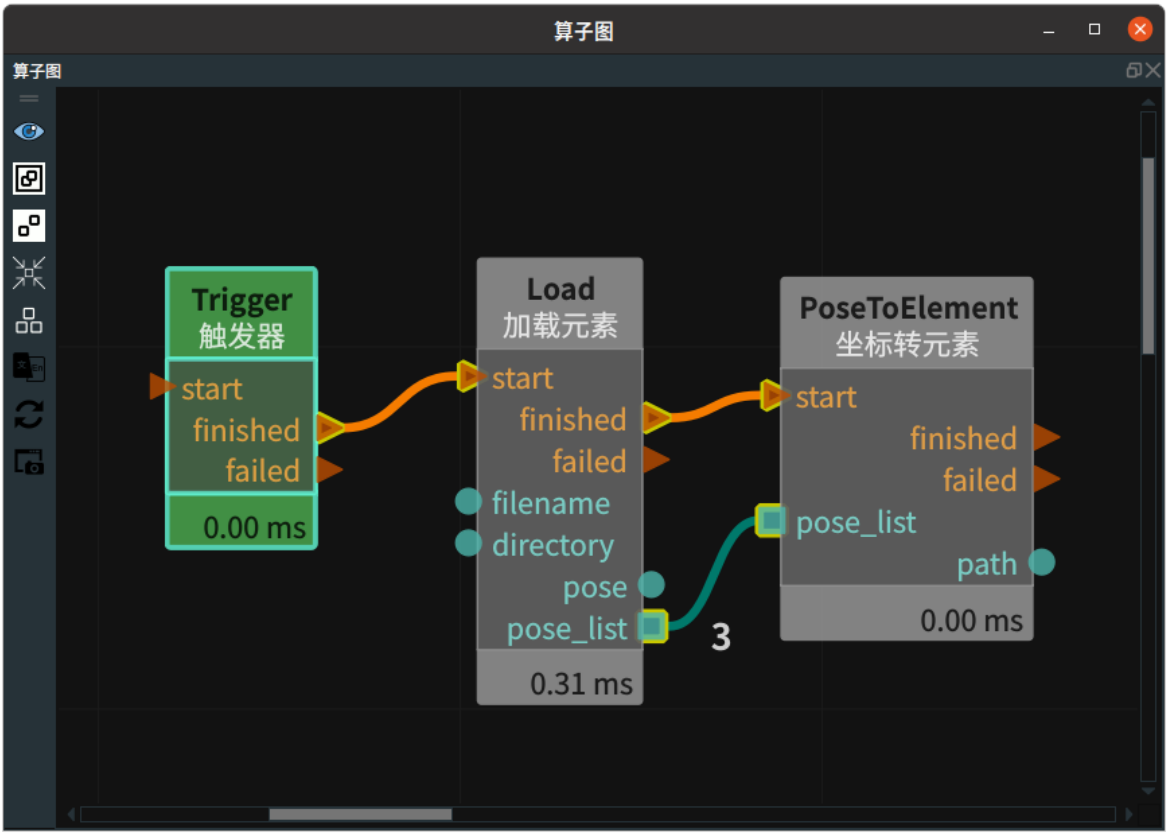
步骤1：算子准备

添加 Trigger 、Load 、PoseToElement 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → pose
   - 目录 → ••• → 选择 pose 文件目录名 （ *example_data/pose* ）
2. 设置 PoseToElement 算子参数：
   - 类型 → path
   - path→  可视

步骤3：连接算子

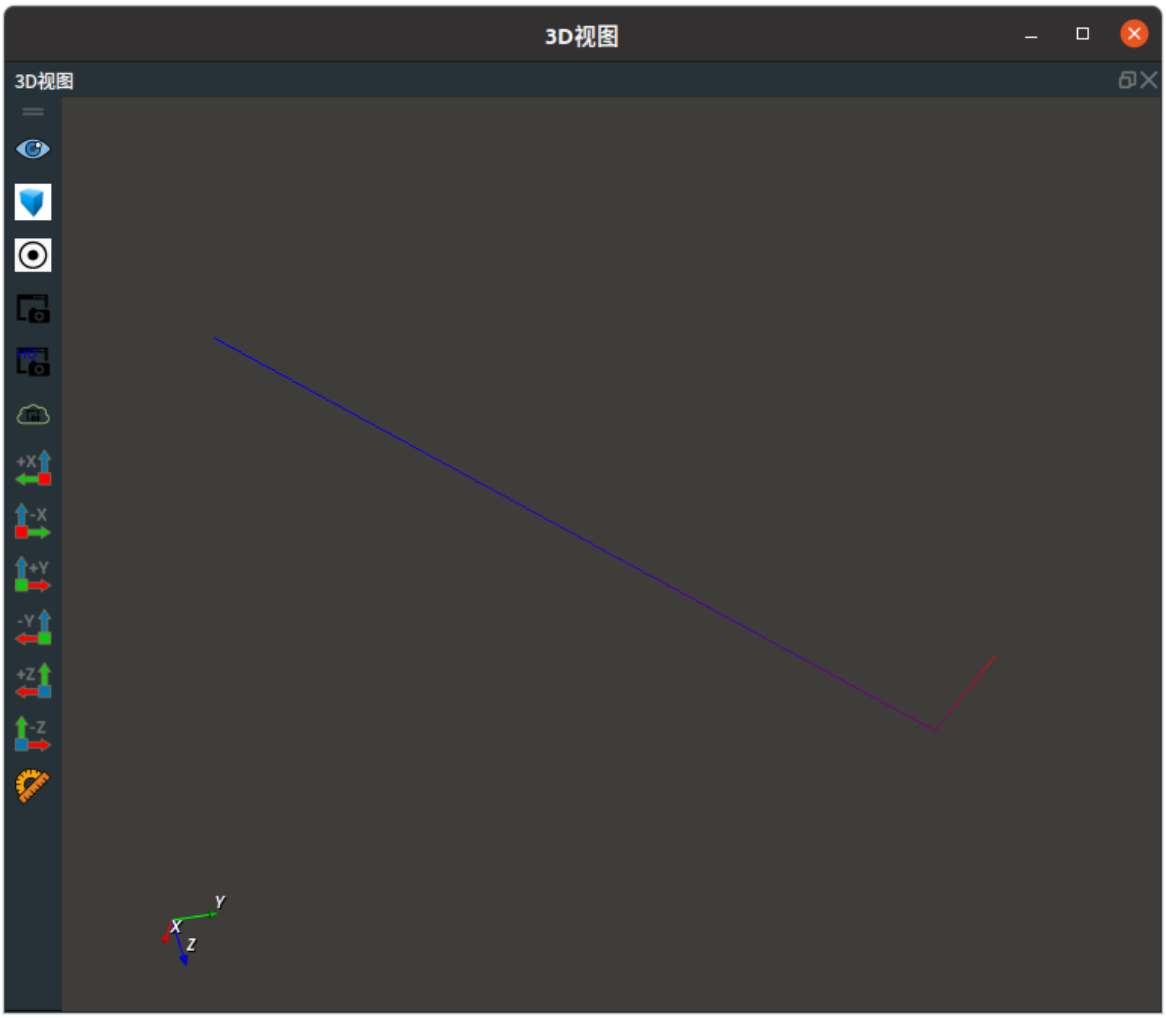


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，在 3D 视图中显示转成的 path 元素。



Sphere

将 PoseToElement 算子的 **类型** 属性选择 Sphere ，用于将 Pose 转 Sphere 。

算子参数

- **半径/radius**：设置球体的半径。
- **球体/sphere**：设置球体在 3D 视图中的可视化属性。
  - 打开球体可视化。
  - 关闭球体可视化。
  - 设置球体的颜色。取值范围：[-2,360] 。默认值：-2 。
  - 设置球体的透明度。取值范围：[1,10] 。默认值：0.8 。
- **球体列表/sphere_list**：设置球体列表在 3D 视图中的可视化属性。参数值描述与 **球体** 一致。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **pose**：
  - 数据类型：Pose
  - 输入内容：单个 pose 数据
- **pose_list**：
  - 数据类型：PoseList

- 输入内容：pose 数据列表

输出：

- `sphere`：
  - 数据类型：Sphere
  - 输出内容：单个球体数据
- `sphere_list`：
  - 数据类型：SphereList
  - 输出内容：球体数据列表

功能演示

本节将使用 PoseToElement 算子中 Sphere ，将 Pose 转 Sphere元素。这与PoseToElement 算子的 Cube 属性将 Pose 转 Cube 的方法相同，请参照该章节的功能演示。

Text

将 PoseToElement 算子的 `类型` 属性选择 Text ，用于将 Pose 转 Text 。

算子参数

- `文本/text`：设置要转成的文字。默认值：%d 。表示显示 pose_list 的 index 。

  说明：只有输入 pose_list 时才会显示索引。
- `文本/text`：设置文字在 3D 视图中的可视化属性。
  -  打开文字可视化。
  -  关闭文字可视化。
  -  设置文字的颜色。取值范围：[-2 , 360]。默认值：-1 。
  -  设置文字的尺寸大小。取值范围：[0 , 99.99]。默认值：0.05。
- `文本列表/text_list`：设置文字列表在 3D 视图中的可视化属性。参数值描述与 `文本` 一致。

数据信号输入输出

输入：

说明：pose 和 pose_list 根据需求选择其中一种数据信号输入即可。

- `pose`：
  - 数据类型：Pose
  - 输入内容：单个 pose 数据
- `pose_list`：
  - 数据类型：PoseList
  - 输入内容：pose 数据列表

输出：

- `text`：
  - 数据类型：Text
  - 输出内容：单个文字数据
- `text_list`：

- 数据类型：TextList
- 输出内容：文字数据列表

功能演示

使用 PoseToElement 算子中 Text ，显示加载 pose 列表的对应索引。

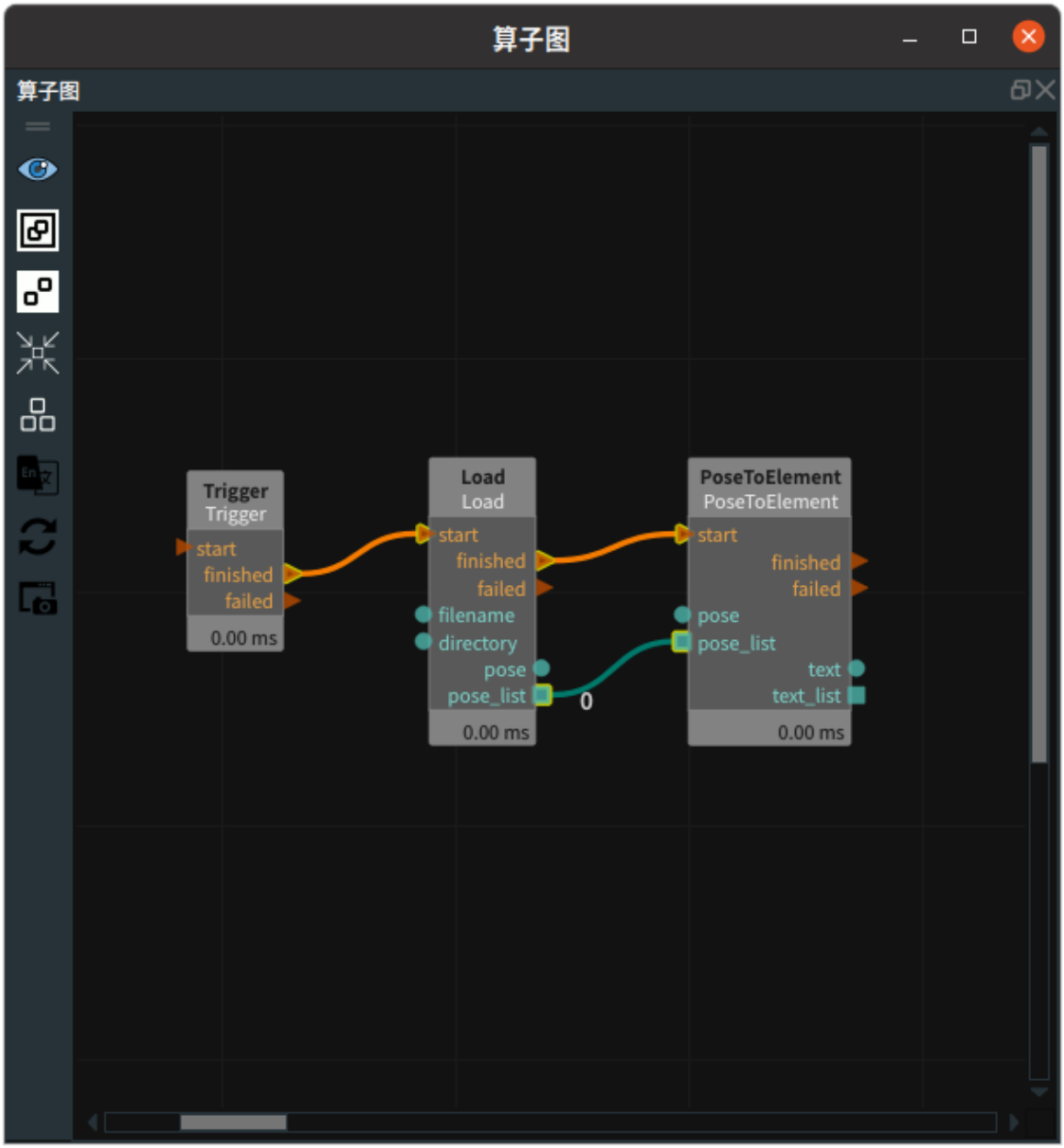
步骤1：算子准备

添加 Trigger 、Load 、PoseToElement 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → Pose
   - 目录 → ••• → 选择 pose 文件目录名 （*example_data/pose*）
   - 坐标列表→  可视
2. 设置 PoseToElement 算子参数：
   - 类型 → Text
   - 文本列表→  可视

步骤3：连接算子

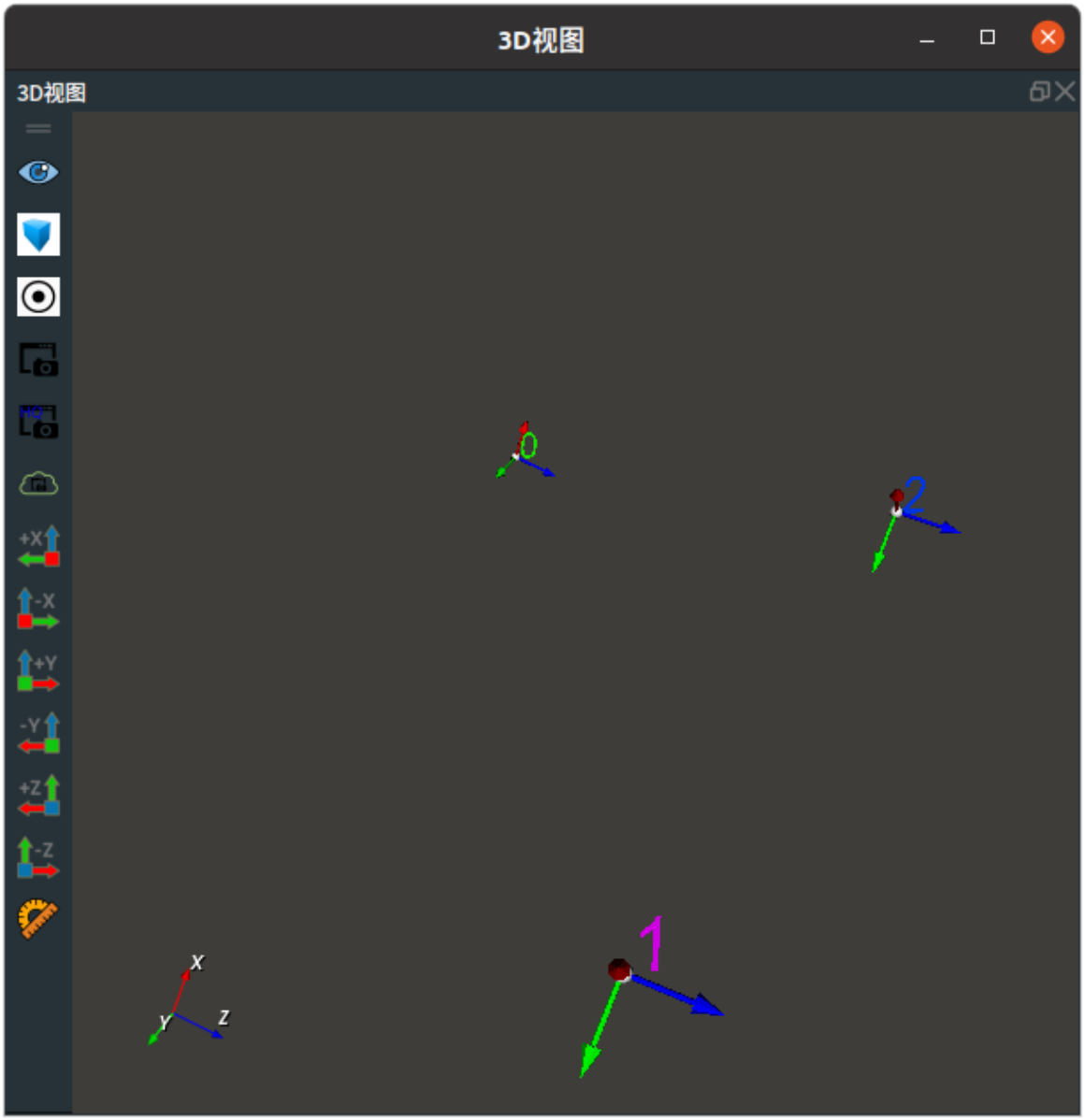


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，在 3D 视图中显示对应的 pose 和对应的 index 。


3D视图
3D视图
+X
-X
+Y
-Y
+Z
-Z
0
1
2
X
Y
Z

Join 联合元素

Join 算子为联合元素，适用于 Pose 、String 。

类型	功能
Pose	用于将单个或者多个pose 组合成列表输出 。
String	用于将单个或者多个以上的 string 组合成单个 string 输出，或者组合成列表输出。

Pose

将 Join 算子的 **类型** 属性选择 pose ，用于将单个或者多个以上的 pose 组合成列表输出。

算子参数

- **输入数量/number_input**：算子数据类型输入端口的数量。取值范围：[1,10] 。默认值：2 。
- **坐标列表/pose_list**：设置 pose 列表在 3D 视图中的可视化属性。
  -  打开 pose 列表可视化。
  -  关闭 pose 列表可视化。
  -  设置 pose 列表的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

- **pose_?** ：
  - 数据类型：Pose
  - 输入内容：pose 数据

输出：

- **pose_list** ：
  - 数据类型：PoseList
  - 输出内容：联合后的 pose 列表数据

功能演示

使用 Join 算子中 pose ，将2个 pose 组合成1个 pose 列表输出。

步骤1：算子准备

添加 Trigger 、Emit（2个） 、Join 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
   - 类型 → pose
   - 坐标 → 0 0 0 0 0 0
2. 设置 Emit_1 算子参数：
   - 类型 → pose
   - 坐标 → 0.3 0 0 0 0 0

3. 设置 Join 算子参数：
    - 类型 → pose
    - 坐标列表 →  可视

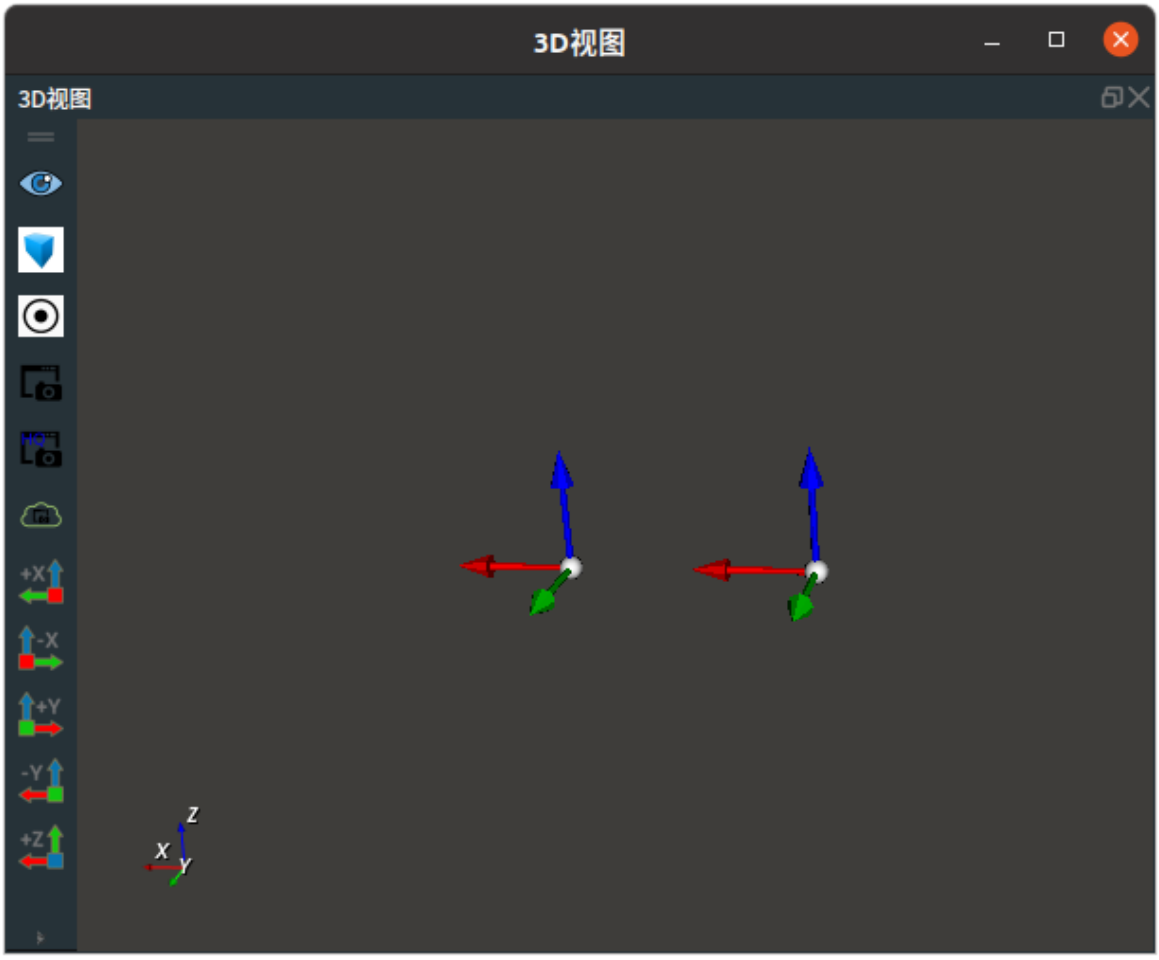
步骤3：连接算子

步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

结果如下图所示，在 3D 视图中显示出创建的两个 pose 所联合的一个 pose 列表。



String

将 Join 算子的 **类型** 属性选择 String ，用于将单个或者多个以上的 string 组合成单个 string 输出，或者组合成列表输出。

算子参数

- **输入数量/number_input**：算子数据类型输入端口的数量。取值范围：[1,10] 。默认值：2 。
- **字符串列表/string_list**：设置字符串列表的曝光属性。可与交互面板中输出工具——“表格”控件绑定。
  - 打开曝光。
  - 关闭曝光。
- **字符串/string**：设置字符串的曝光属性。可与交互面板中输出工具——“文本框”控件绑定。参数值描述与 string_list 一致。

数据信号输入输出

输入：

- **string_?**：
  - 数据类型：String
  - 输入内容：String 数据

输出：

- **string_list**：

- 数据类型：StringList
- 输出内容：string 列表数据

- `string`：
  - 数据类型：String
  - 输出内容：单个 String 数据

功能演示

使用 Join 算子中 String ，生成 2 个 string 组合成 1 个 string 列表输出。

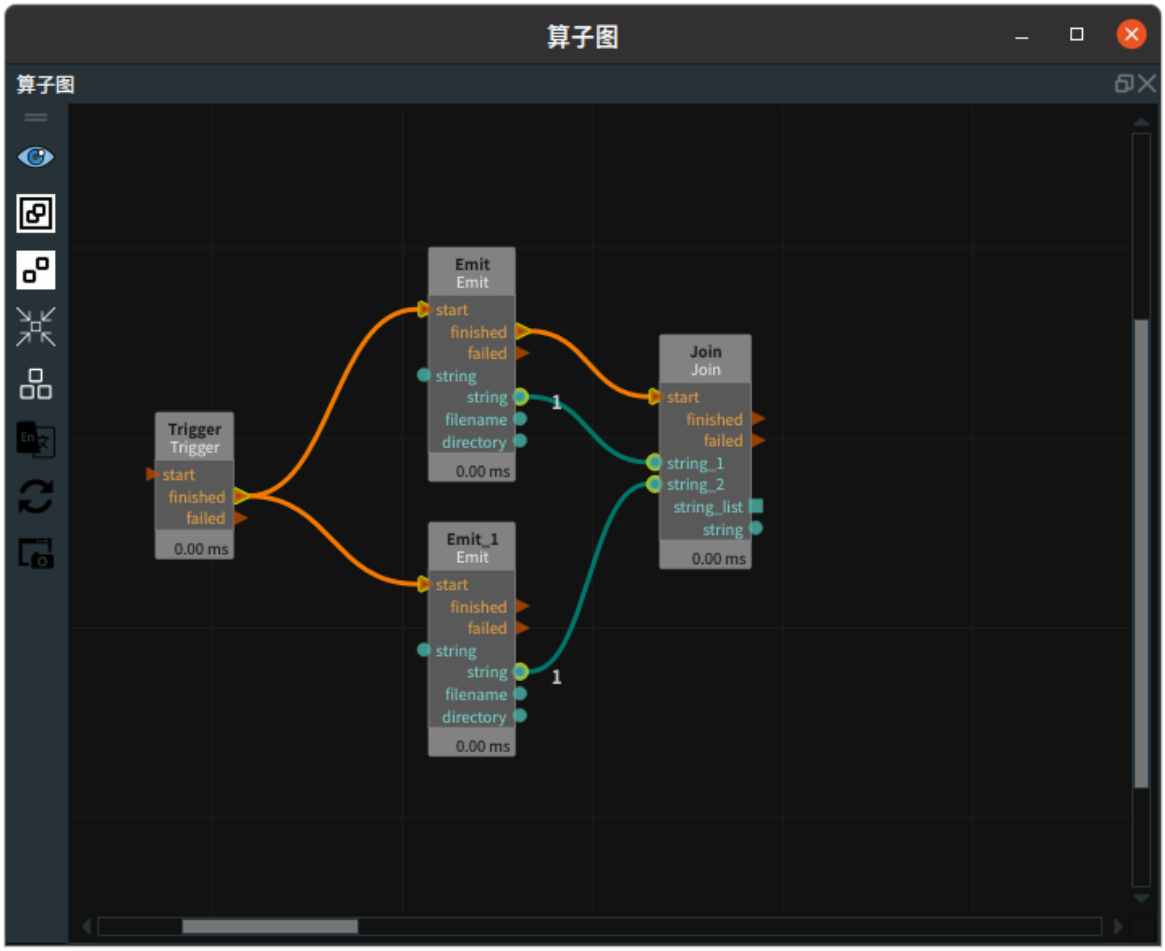
步骤1：算子准备

添加 Trigger 、Emit （2个）、Join 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
   - type → String
   - String → RVS
2. 设置 Emit_1 算子参数：
   - type → String
   - String → 2013
3. 设置 Join 算子参数：
   - type → String
   - stringlist → 曝光

步骤3：连接算子



步骤4：运行

1. 将 string_list 与交互面板中输出工具——“表格”控件进行绑定。
2. 点击运行按钮，触发 Trigger 算子。

运行结果

结果如下图所示，在交互面板中显示出两个string组成的一个 string 列表。

Foreach 遍历元素列表

Foreach 算子为遍历元素列表，用于遍历 Cube 、Image 、JointArray 、MotionPlan 、Path 、PlannedGrasp 、PointCloud 、Pose 、String 、ImagePoints列表。

类型	功能
Cube	用于遍历立方体列表。
Image	用于遍历图像列表。
JointArray	用于遍历机器人关节弧度值列表。
MotionPlan	/
Path	用于遍历路径数组。
PlannedGrasp	/
PointCloud	用于遍历点云列表。
Pose	用于遍历 pose 列表。
String	用于遍历字符串列表。
ImagePoints	用于遍历图像关键点坐标列表。
RotatedRect	/

Cube

将 Foreach 算子的 **类型** 属性选择 Cube ，用于遍历 Cube 列表。

算子参数

- **启动便迭代/iterate_at_start**：数据类型：Bool 。使能/去使能迭代触发。
  - True：当算子左侧 start 端口被触发时，会自动触发一次 iterate ，此时会输出当前计数序号对应的数据，然后右侧 iterate 被触发，计数序号自加 1 。
  - False：当算子左侧 start 端口被触发时，iterate 不会被触发。
- **最大迭代次数/max_iteration**：计数序号的最大迭代次数。输入的数值表示最大迭代几次。默认值：-1 ，表示迭代整个列表。
- **下限/lower_bound**：表示迭代计数的最小边界值。默认值：0 。表示从列表第一个开始迭代。最小值：0。
- **上限/upper_bound**：表示迭代计数的最大边界值。默认值：-1 。表示 size-1 ，为列表最后一个。
- **立方体/cube**：设置立方体在 3D 视图中的可视化属性。
  -  打开立方体可视化。
  -  关闭立方体可视化。
  -  设置立方体的颜色。取值范围：[-2,360] 。默认值：-2。
  -  设置立方体的透明度。取值范围：[0,1] 。默认值：0.5 。

数据信号输入输出

输入：

- `cube_list`：
  - 数据类型：CubeList
  - 输入内容：立方体数据列表

输出：

- `cube`：
  - 数据类型：Cube
  - 输出内容：单个立方体数据

功能演示

使用 Foreach 算子中 Cube ，遍历输出加载 cube_list 里的 cube 数据。

步骤1：算子准备

添加 Trigger（2个）、Load、Foreach、算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → Cube
   - 目录 → ••• → 选择 cube 文件目录名（*example_data/cube*）
2. 设置 Foreach 算子参数：
   - 类型 → Cube
   - 立方体 →  可视

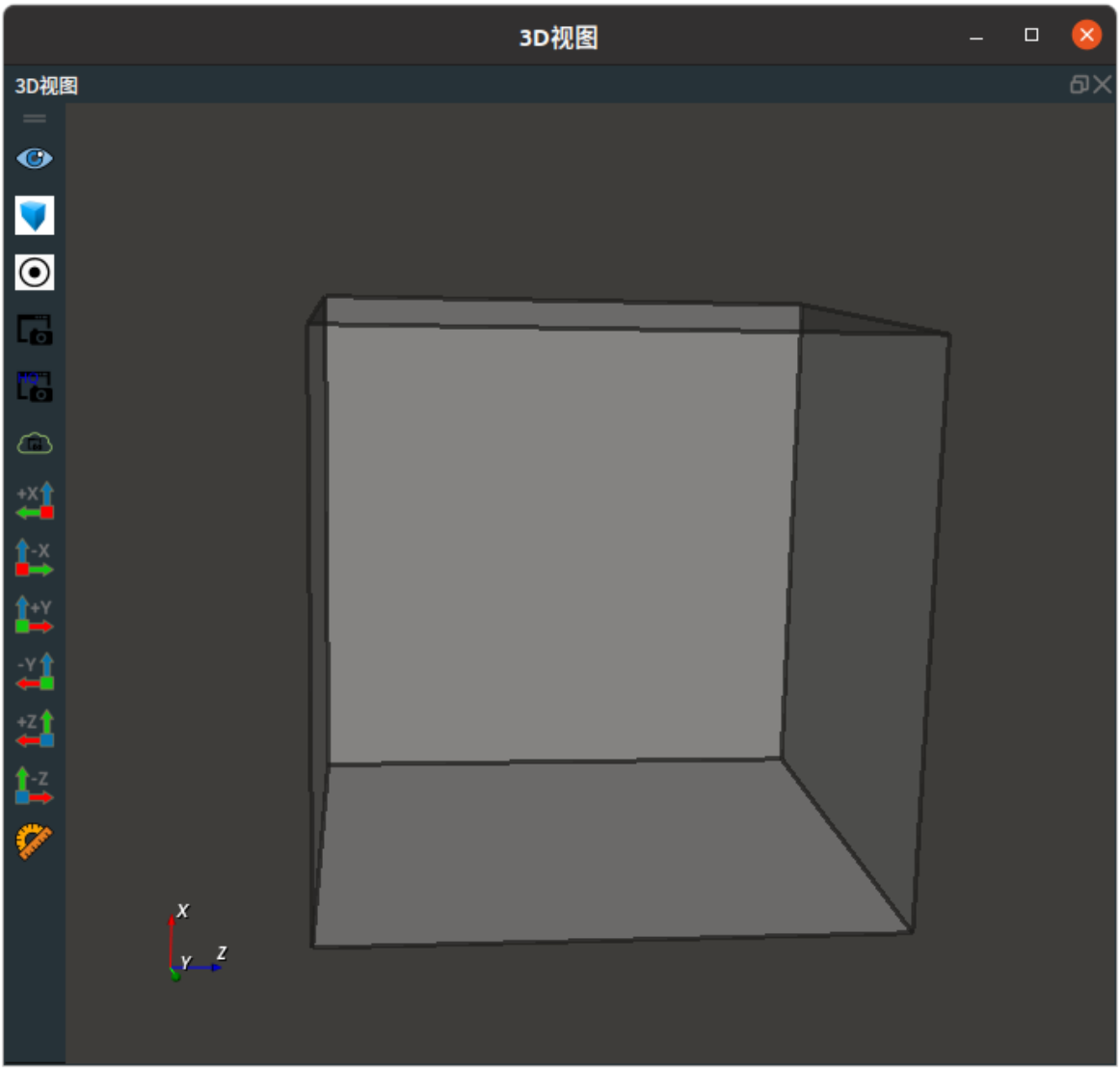
步骤3：连接算子

步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子，再触发 Trigger_1 。

运行结果

如下图所示，在 3D 视图中显示当前遍历的 cube 。



Image

将 Foreach 算子的 **类型** 属性选择 Image ，用于遍历图像列表。

算子参数

- **其余参数**：同本算子 Cube 章节——算子参数。
- **图像**：设置图像在 2D 视图中的可视化属性。
  - 打开图像可视化。
  - 关闭图像可视化。

数据信号输入输出

输入：

- **image_list**：
  - 数据类型：ImageList
  - 输入内容：图像列表数据

输出：

- **Image**：

- 数据类型：Image
- 输出内容：单个图像数据

功能演示

本节将使用 Foreach 算子中 Image ，遍历加载图像列表中的图像。这与 Foreach 算子中 Cube 属性遍历 Cube 列表的方法相同，请参照该章节的功能演示。

JointArray

将 Foreach 算子的 `类型` 属性选择 JointArray ，用于遍历机器人关节弧度值。

算子参数

- `所有参数`：同本算子 Cube 章节——算子参数。

数据信号输入输出

输入：

- `joint_list`：
  - 数据类型：JointArrayList
  - 输入内容：jointarray 列表数据

输出：

- `joint`：
  - 数据类型：JointArrayList
  - 输出内容：jointarray 数据

功能演示

本节将使用 Foreach 算子中 JointArray ，遍历加载 jointarray_list 里的单组机器人关节弧度值数据。这与 Foreach 算子中 Cube 属性遍历 Cube 列表的方法相同，请参照该章节的功能演示。

Path

将 Foreach 算子的 `类型` 属性选择 Path ，用于遍历机器人路径。

算子参数

- `其余参数`：同本算子 Cube 章节——算子参数。
- `path`：设置路径在 3D 视图中的可视化属性。
  -  打开路径可视化。
  -  关闭路径可视化。
  -  设置路径的线宽。默认值：1 。

数据信号输入输出

输入：

- `path_list`：
  - 数据类型：PathList
  - 输入内容：路径列表数据

输出：

- `path`：
  - 数据类型：Path
  - 输出内容：单个路径数据

功能演示

本节将使用 Foreach 算子中 Path ，遍历加载 path_list 里的路径数据。这与 Foreach 算子中 Cube 属性遍历 Cube 列表的方法相同，请参照该章节的功能演示。

PointCloud

将 Foreach 算子的 `类型` 属性选择 PointCloud ，用于遍历点云。

算子参数

- `其余参数`：同本算子 Cube 章节——算子参数。
- `点云`：设置当前遍历的点云在 3D 视图中的可视化属性。
  -  打开点云可视化。
  -  关闭点云可视化。
  -  设置3D视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  -  设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。

数据信号输入输出

输入：

- `cloud_list`：
  - 数据类型：PointCloudList
  - 输入内容：点云列表数据

输出：

- `cloud`：
  - 数据类型：PointCloud
  - 输出内容：单个点云数据

功能演示

本节将使用 Foreach 算子中 PointCloud ，遍历加载点云列表中的点云数据。这与 Foreach 算子中 Cube 属性遍历 Cube 列表的方法相同，请参照该章节的功能演示。

Pose

将 Foreach 算子的 **类型** 属性选择 Pose ，用于遍历 pose 。

算子参数

- **其余参数**：同本算子 Cube 章节——算子参数。
- **坐标**：设置当前遍历的 pose 在 3D 视图中的可视化属性。
  -  打开 pose 可视化。
  -  关闭 pose 可视化。
  -  设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

- **pose_list**：
  - 数据类型：PoseList
  - 输入内容：pose 列表数据

输出：

- **pose**：
  - 数据类型：Pose
  - 输出内容：单个 pose 数据

功能演示

本节将使用 Foreach 算子中 Pose ，遍历加载 pose_list 里的 pose 数据。这与 Foreach 算子中 Cube 属性遍历 Cube 列表的方法相同，请参照该章节的功能演示。

String

将 Foreach 算子的 **类型** 属性选择 String ，用于遍历字符串。

算子参数

- **其余参数**：同本算子 Cube 章节——算子参数。
- **字符串**：设置当前遍历的 string 的曝光属性。打开后可与交互面板中输出工具——“文本框”控件绑定
  -  打开曝光。
  -  关闭曝光。

数据信号输入输出

输入：

- **string_list**：
  - 数据类型：String
  - 输入内容：String 列表数据

输出：

- `string`：
  - 数据类型：String
  - 输出内容：单个 String 数据

功能演示

使用 Foreach 算子中 String ，遍历目录中的子目录。

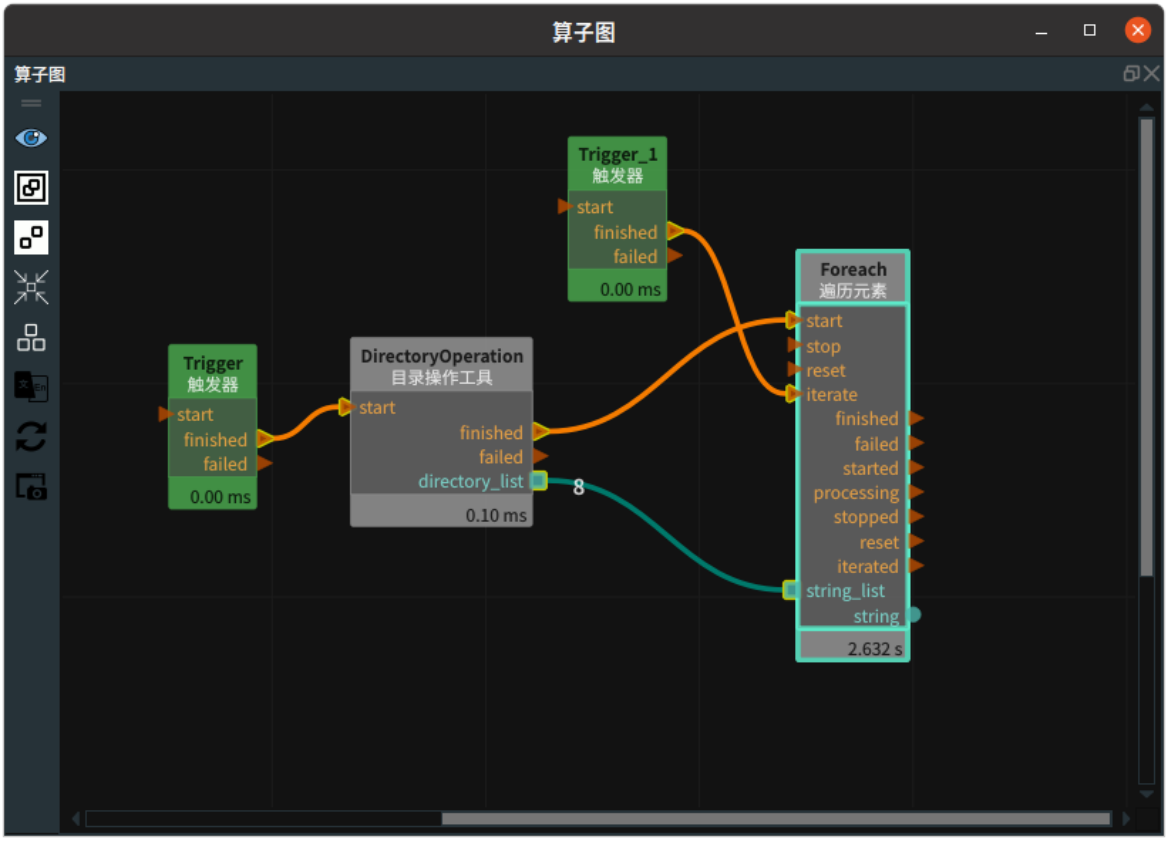
步骤1：算子准备

添加 Trigger（2个） 、DirectoryOperation 、Foreach 算子至算子图。

步骤2：设置算子参数

1. 设置 DirectoryOperation 算子参数：
   - 类型 → ReadDirectory
   - 父目录 → ••• → 选择 example_data 文件目录名（ *example_data* ）
2. 设置 Foreach 算子参数：
   - 类型 → String
   - 字符串 → 曝光

步骤3：连接算子

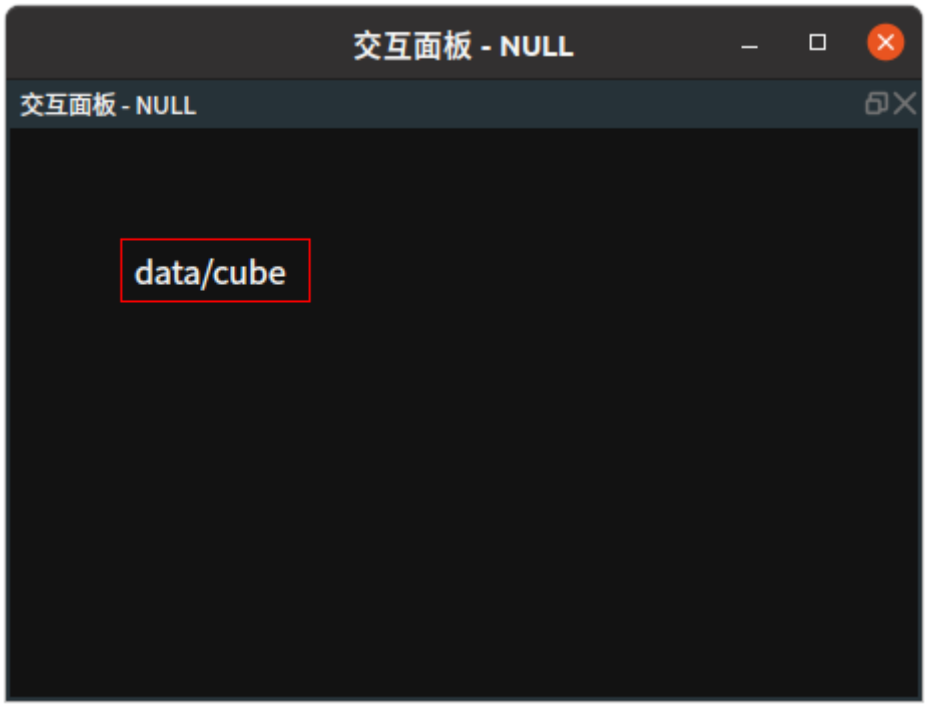


步骤4：运行

1. 将 string 属性与交互面板中输出工具——“文本框”控件进行绑定。
2. 点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

结果如下图所示， 交互面板中显示当前遍历的子目录名。



ImagePoints

将 Foreach 算子的 `类型` 属性选择 ImagePoints ，用于遍历图像关键点坐标 。

算子参数

- `所有参数`：同本算子 Cube 章节——算子参数。

数据信号输入输出

输入：

- `image_points_list`：
  - 数据类型：ImagePoints
  - 输入内容：ImagePoints 列表数据

输出：

- `image_points`：
  - 数据类型：ImagePoints
  - 输出内容：单组 ImagePoints 数据

功能演示

本节将使用 Foreach 算子中 imagePoints ，遍历图像关键点坐标列表里的单组图像关键点坐标。这与 Foreach 算子中 Cube 属性遍历 Cube 列表的方法相同，请参照该章节的功能演示。

Multiplexer 元素多路复用器

Multiplexer 算子为元素多路复用器。用于将多条数据信号流并联在一起，合并后再统一传给后续处理环节。适用于：Cube 、Image 、JointArray 、PointCloud 、Pose 、String 、ImagePoints。

类型	功能
Cube	将多条立方体数据信号流并联在一起，合并后再统一传给后续处理环节。
Image	将多条图像数据信号流并联在一起，合并后再统一传给后续处理环节。
JointArray	将多条机器人关节值数据信号流并联在一起，合并后再统一传给后续处理环节。
PointCloud	将多条点云数据信号流并联在一起，合并后再统一传给后续处理环节。
Pose	将多条坐标数据信号流并联在一起，合并后再统一传给后续处理环节。
String	将多条字符串数据信号流并联在一起，合并后再统一传给后续处理环节。
ImagePoints	将多条图像关键点坐标数据信号流并联在一起，合并后再统一传给后续处理环节。

Cube

将 Multiplexer 算子的 **类型** 属性选择 Cube ，用于将多条立方体数据信号流并联在一起，合并后再统一传给后续处理环节。

算子参数

- **是否为列表/is_list**：数据类型：bool 。将算子输入输出端口设置为列表形式。
  - True：元素类型变为 cube_list。
  - False：元素类型为 cube。
- **输入数量/number_input**：决定该算子的输入端口 cube_? 的数量。范围：[1,10] 。默认值：2 。
- **选择器/selector**：其内容为输出端口所采用的 cube _ ? 。
- **立方体/cube**：设置立方体在 3D 视图中的可视化属性。
  -  打开立方体可视化。
  -  关闭立方体可视化。
  -  设置立方体的颜色 。取值范围：[-2,360] 。默认值：-2。
  -  设置立方体的透明度 。取值范围：[0,1] 。默认值：0.5 。

数据信号输入输出

输入：

- **selector**：
  - 数据类型：String
  - 输入内容：该端口通常是配合 Or 算子的 selector 输出端口使用,也可以通过 Emit-String 给定。
- **cube_?**：
  - 数据类型：Cube

- 输入内容：cube 数据

输出：

- `cube`：
  - 数据类型：Cube
  - 输出内容：当算子 selector 端口输入为 “0” 时，输入端口 cube_0 中的数据会作为输出端口 cube 的数据输出；selector 端口输入为 “1” 时对应输出 cube_1 中的数据，以此类推。

功能演示

使用 Multiplexer 算子中 Cube ，将生成的 2 条立方体数据信号流并联在一起，分别触发查看输出效果。

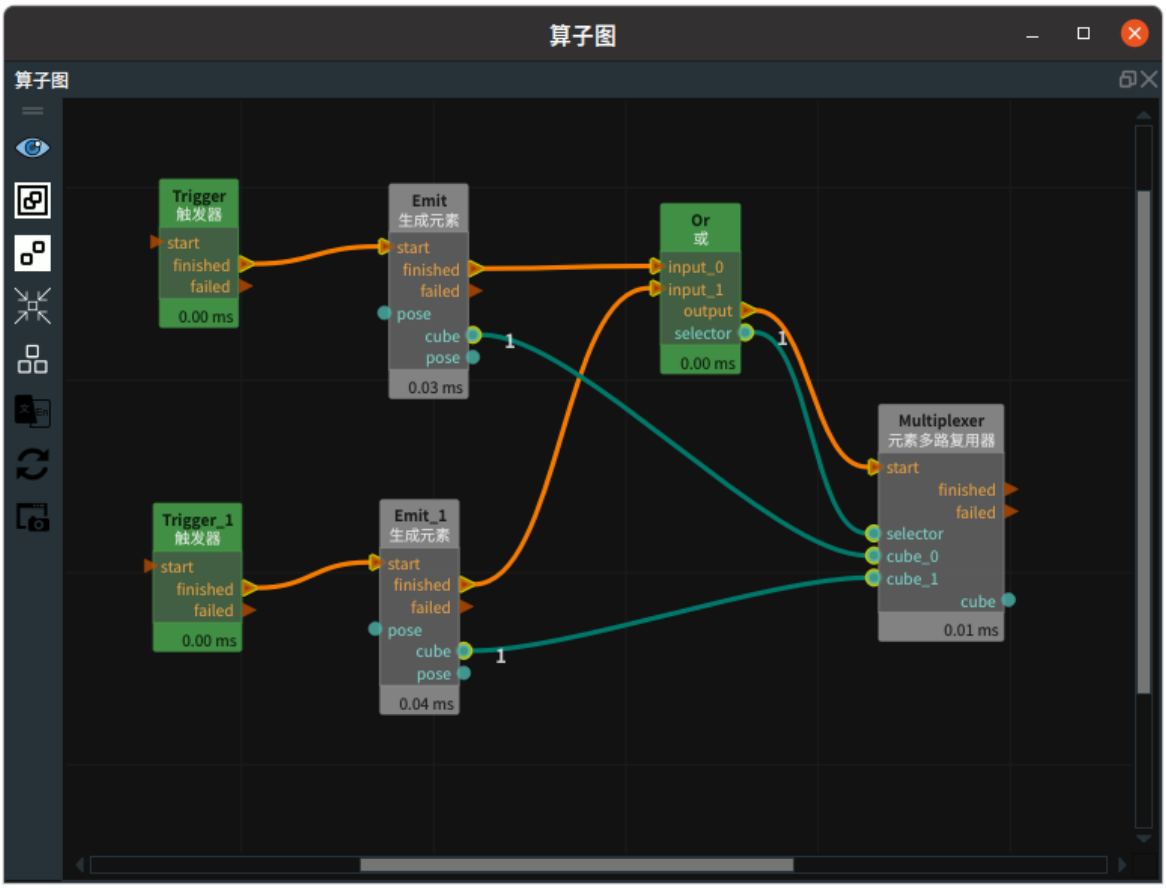
步骤1：算子准备

添加 Trigger 、Emit（2个） 、Or 、Multiplexer 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
   - 类型 → Cube
   - 坐标 → 0 0 0 0 0 0
   - 宽度 → 1
   - 高度 → 1
   - 深度 → 1
2. 设置 Emit_1 算子参数：
   - 类型 → Cube
   - 坐标 → 0.5 0 0 0 0 0
   - 宽度 → 1
   - 高度 → 2
   - 深度 → 3
3. 设置 Multiplexer 算子参数：
   - 类型 → Cube
   - 输入数量 → 2
   - 立方体 →  可视
   - 选择器 → 1

步骤3：连接算子

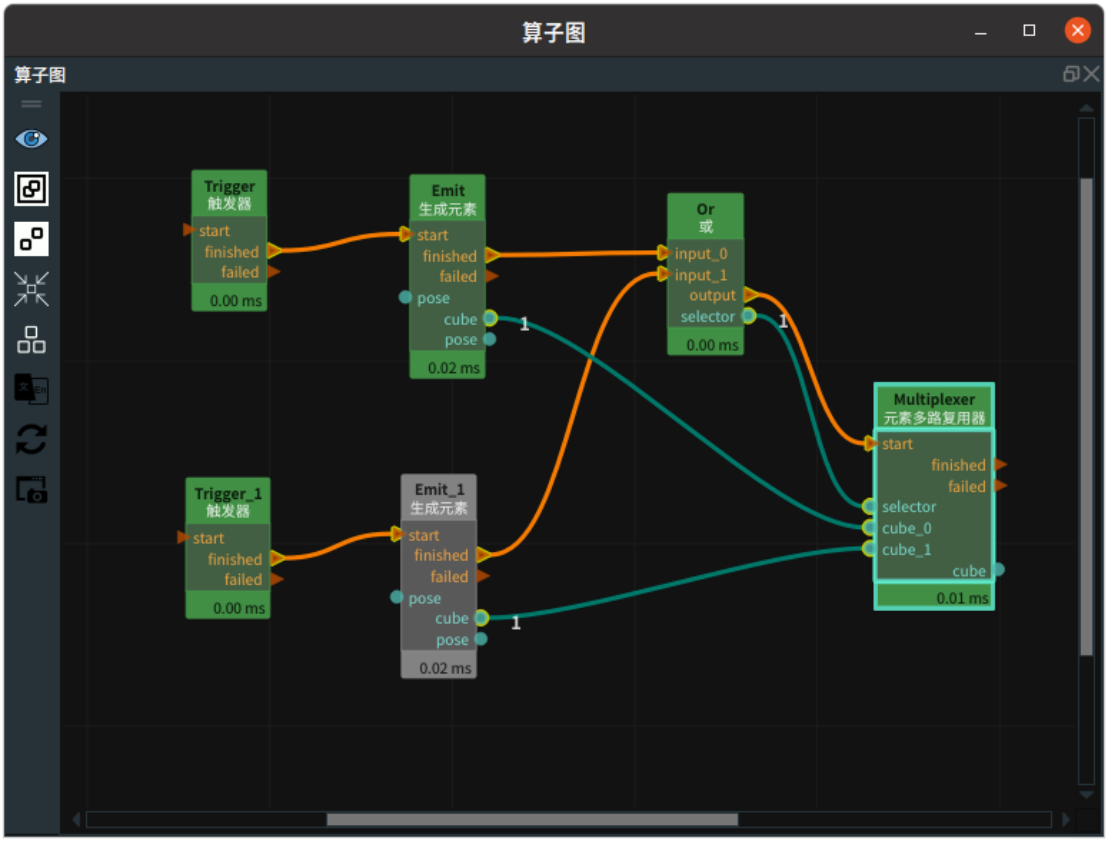


步骤4：运行

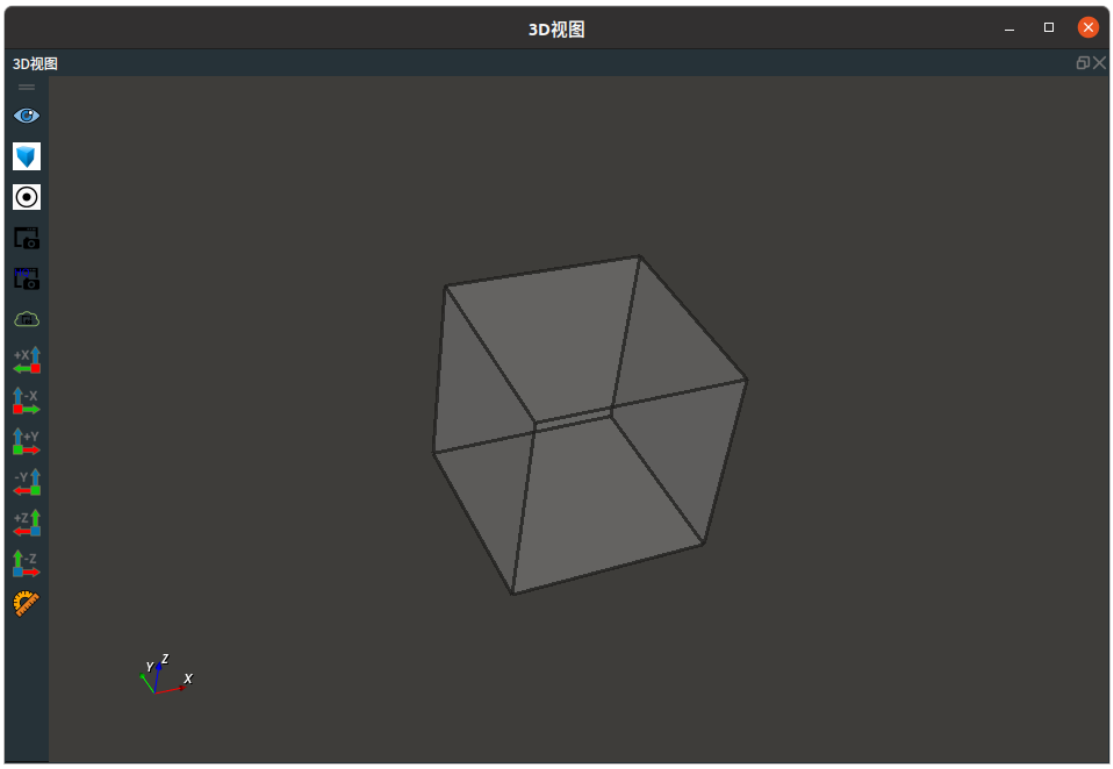
点击 RVS 运行按钮，触发 Trigger 算子。

运行结果：

1. 如下图所示，触发 Trigger ，触发 Or 算子 input_0 端口，因此 Multiplexer 输出的是 Cube_0 的数据。

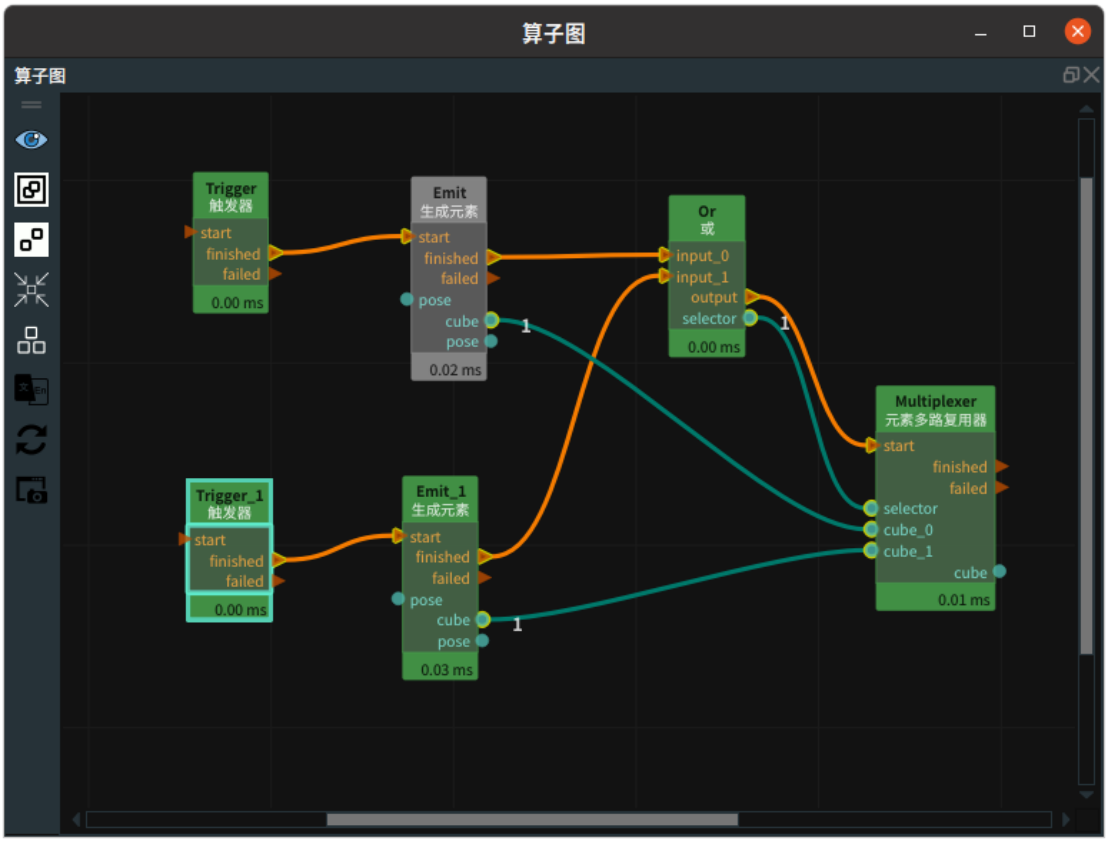


2. 3D 视图中显示 Emit 算子生成的 Cube 。

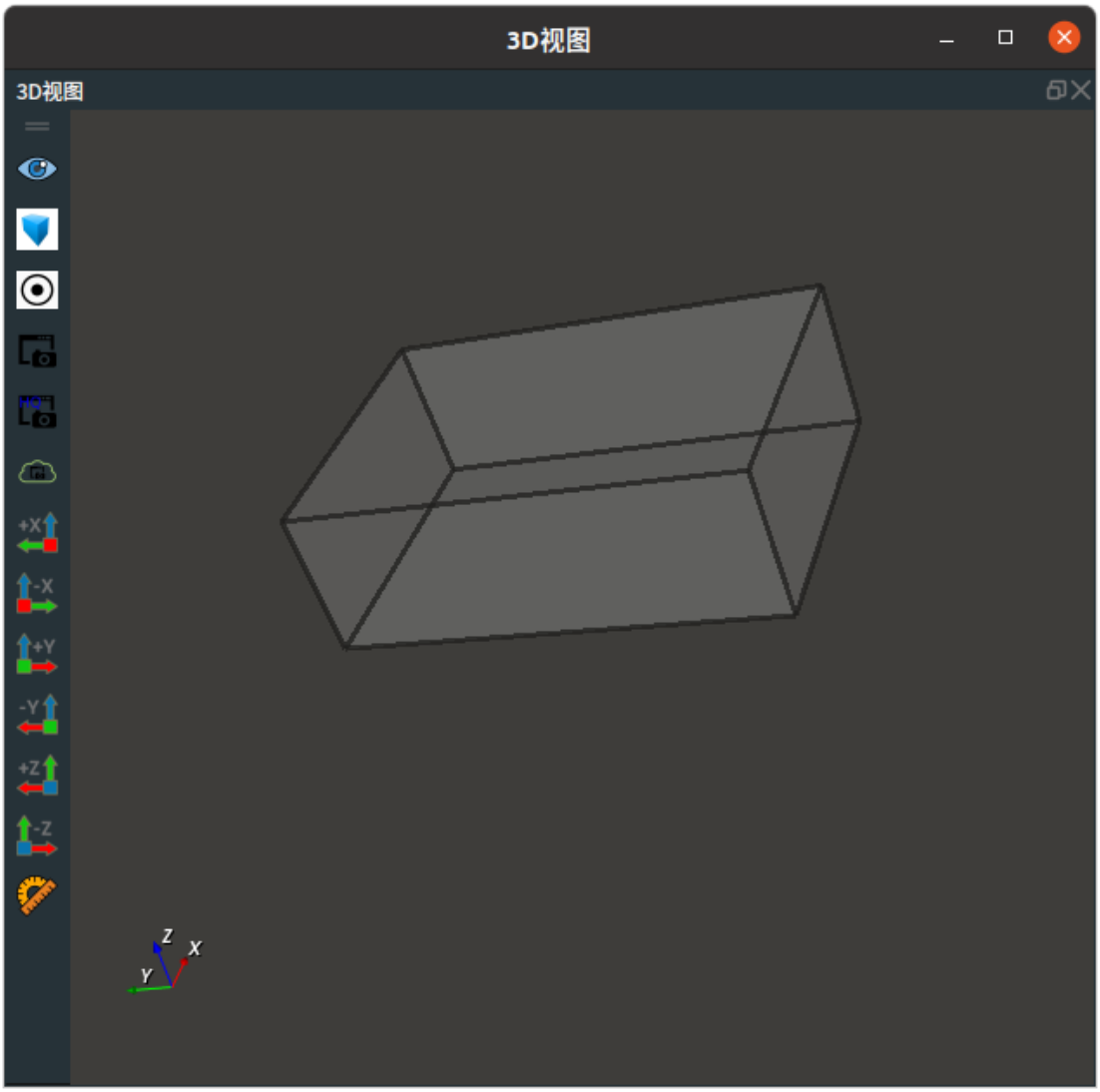


3. 触发 Trigger_1 ，触发 Or 算子 input_1 端口，因此 Multiplexer 右侧 cube 端口输出的是 Cube_1 的数据。

   注意：触发 Trigger 后，若没有进行刷新（算子图左侧侧边栏倒数第二个为“刷新”按钮），再次触发Trigger_1，Emit 算子 和 Emit_1 算子都显示绿色。



4. 3D 视图中显示 Emit_1 算子生成的 Cube 。



Image

将 Multiplexer 算子的 **类型** 属性选择 Image ，用于将多条图像数据信号流并联在一起，合并后再统一传给后续处理环节。

算子参数

- **是否为列表/is_list**：数据类型：bool 。将算子输入输出端口设置为列表形式。
  - True：元素类型变为 image_list。
  - False：元素类型为 image。
- **输入数量/number_input**：决定该算子的输入端口 image_? 的数量。范围：[1,10] 。默认值：2 。
- **选择器/selector**：其内容为输出端口所采用的 image _ ? 。
- **图像/image**：设置图像在 2D 视图中的可视化属性。
  - 打开图像可视化。
  - 关闭图像可视化。

数据信号输入输出

输入：

- **selector**：
  - 数据类型：String

    - 输入内容：其内容决定了输出端口所采用的实际数据。该端口通常是配合 Or 算子的 selector 输出端口使用；也可以通过 Emit-String 给定。
- `image_?`：
    - 数据类型：Image
    - 输入内容：图像数据

输出：

- `image`：
    - 数据类型：Image
    - 输出内容：当算子 selector 端口输入为 “0” 时，输入端口 image_0 中的数据会作为输出端口 image 的数据输出；selector 端口输入为 “1” 时对应输出image_1 中的数据，以此类推。

功能演示

使用 Multiplexer 算子中 Image ，将 2 条加载的图像数据信号流并联在一起，分别触发查看输出效果。

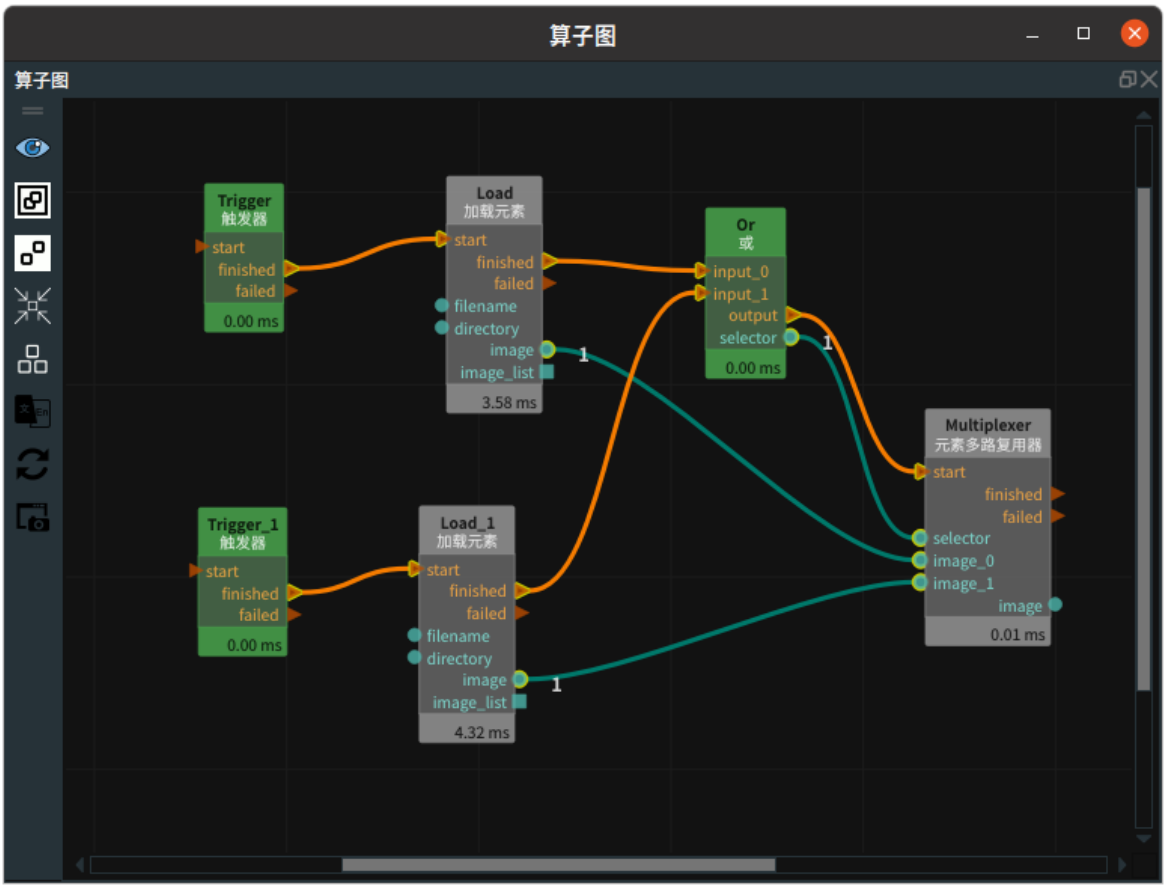
步骤1：算子准备

添加 Trigger 、Load（2个） 、Or 、Multiplexer 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 类型 → image
    - 文件 → ••• → 选择图像文件名（ *example_data/images/cat.png* ）
2. 设置 Load_1 算子参数：
    - 类型 → image
    - 文件 → ••• → 选择图像文件名（ *example_data/images/dog.png* ）
3. 设置 Multiplexer 算子参数：
    - 类型 → image
    - 输入数量 → 2
    - 图像 →  可视

步骤3：连接算子

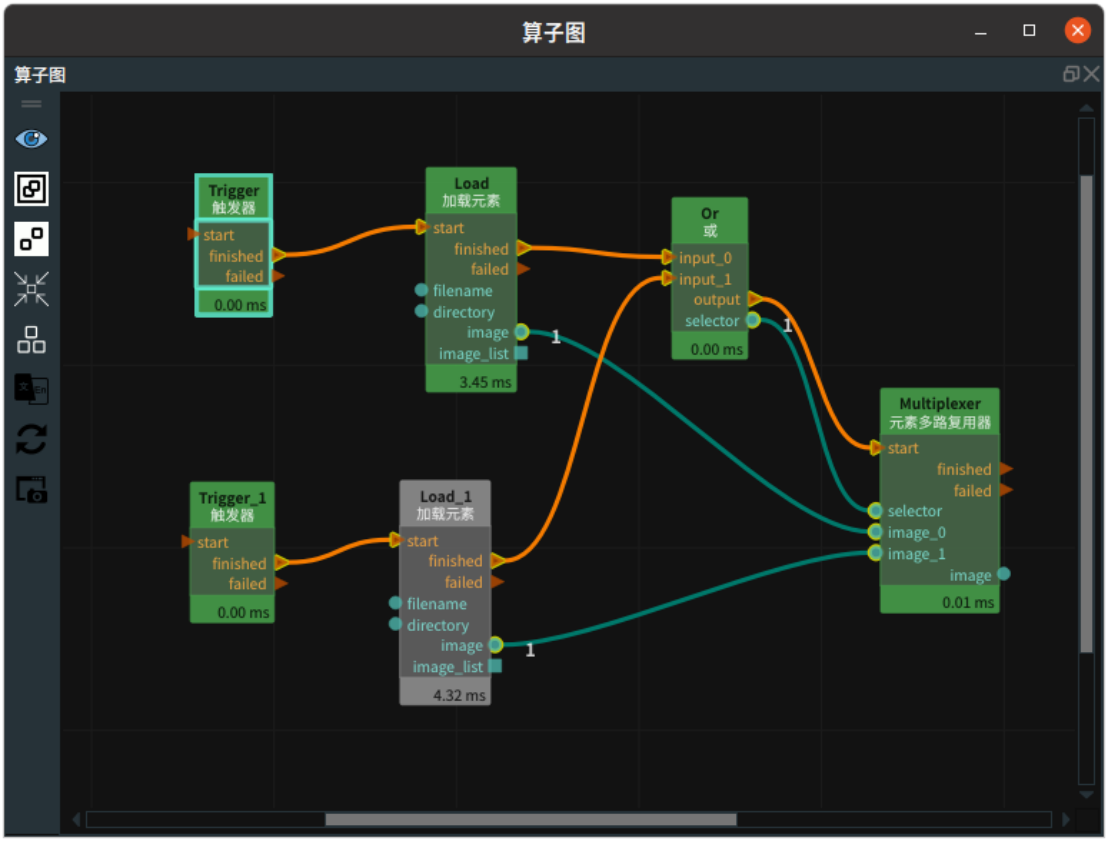


步骤4：运行

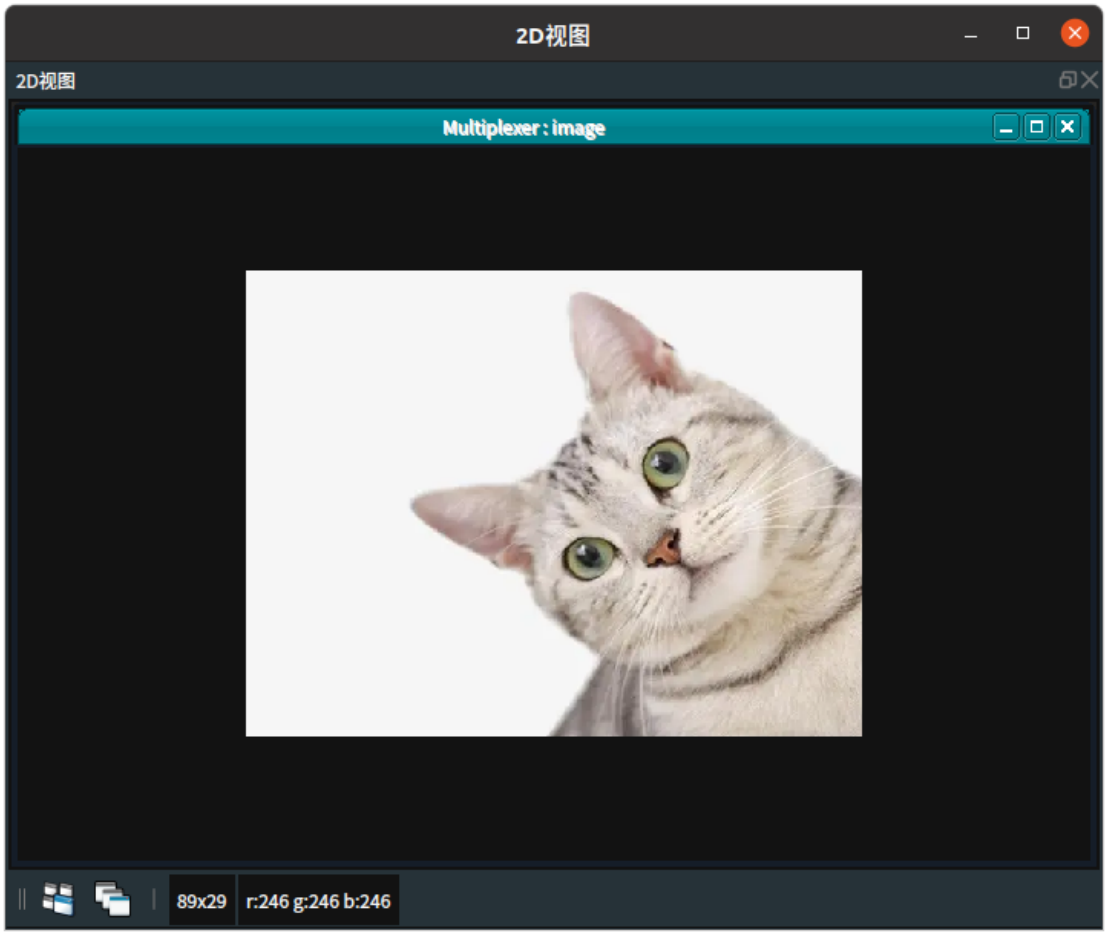
点击 RVS 运行按钮，触发 Trigger 算子。

运行结果：

1. 如下图所示，触发 Trigger ，触发 Or 算子 input_0 端口，因此 Multiplexer 输出的是 image_0 的数据。

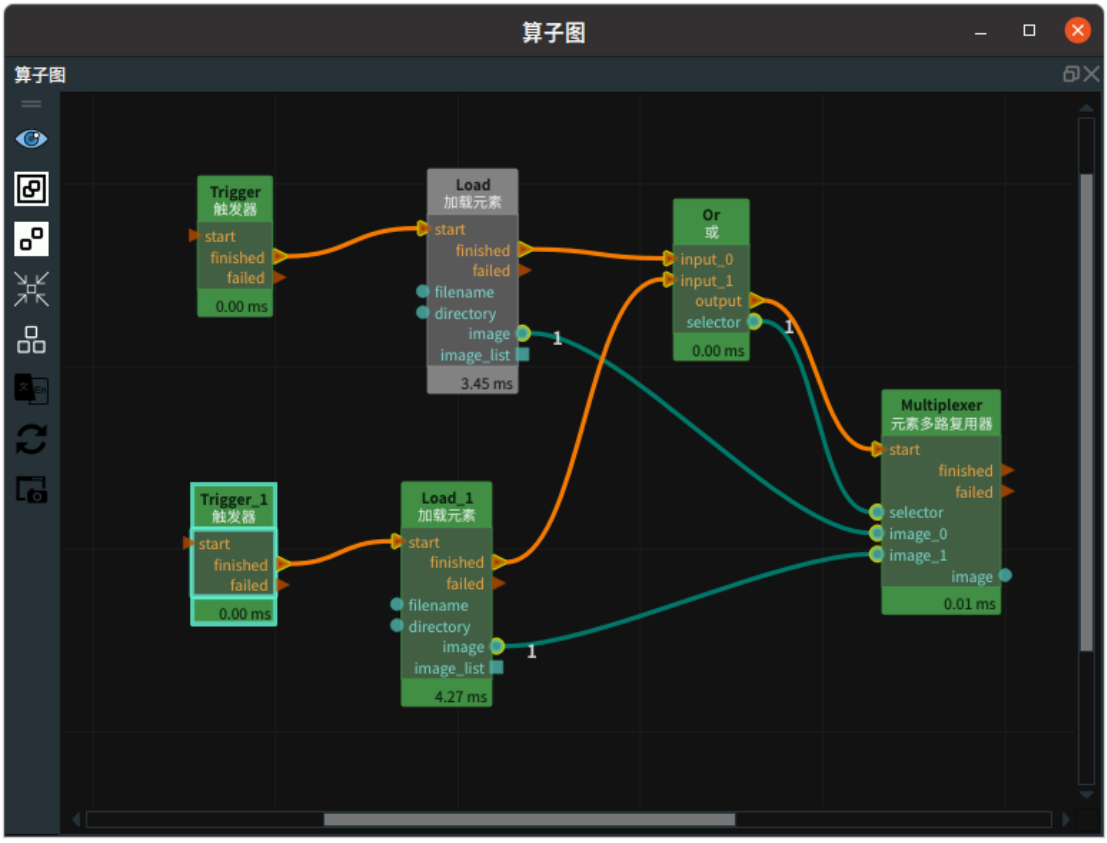


2. 2D 视图中显示 Load 算子加载的图像 。

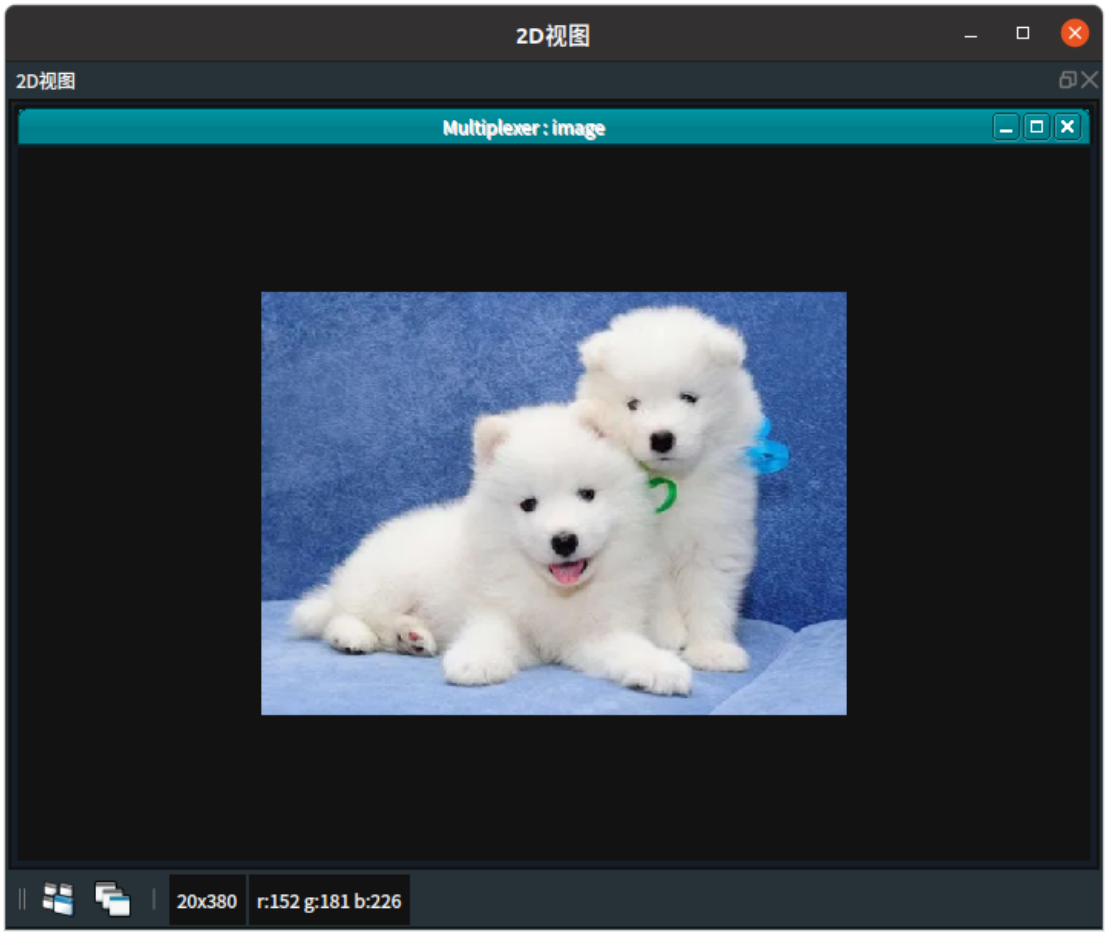


3. 触发 Trigger_1 ，触发 Or 算子 input_1 端口，因此 Multiplexer 右侧 image 端口输出的是 image_1 的数据。

   注意：触发 Trigger 后，若没有进行刷新，再次触发Trigger_1，Load 算子 和 Load_1 算子都显示绿色。



4. 2D 视图中显示 Load_1 算子加载的图像 。



JointArray

将 Multiplexer 算子的 **类型** 属性选择 JointArray ，用于将多条机器人关节弧度值数据信号流并联在一起，合并后再统一传给后续处理环节。

算子参数

- **是否为列表/is_list**：数据类型：bool 。将算子输入输出端口设置为列表形式。
  - True：元素类型变为 joint_list。
  - False：元素类型为 joint。
- **输入数量/number_input**：决定该算子的输入端口 joint_? 的数量。范围：[1,10] 。默认值：2 。
- **选择器/selector**：其内容为输出端口所采用的 joint_? 。

数据信号输入输出

输入：

- **selector**：
  - 数据类型：String
  - 输入内容：其内容决定了输出端口所采用的实际数据。该端口通常是配合 Or 算子的 selector 输出端口使用；也可以通过 Emit-String 给定。
- **joint_?**：
  - 数据类型：JointArray
  - 输入内容：机器人关节值数据

输出：

- **joint**：
  - 数据类型：JointArray
  - 输出内容：
  - 当算子 selector 端口输入为 “0” 时，输入端口 joint_0 中的数据会作为输出端口 joint 的数据输出；selector 端口输入为 “1” 时对应输 joint_1 中的数据，以此类推。

功能演示

本节将使用 Multiplexer 算子中 JointArray ，将 2 条机器人关节弧度值数据信号流并联在一起，分别触发查看输出效果。这与 Multiplexer 算子中 Image 属性将 2 条立方体数据信号流并联在一起，分别触发查看输出效果的结果方法相同。请参照该章节的功能演示。

PointCloud

将 Multiplexer 算子的 **类型** 属性选择 PointCloud ，用于将多条点云数据信号流并联在一起，合并后再统一传给后续处理环节。

算子参数

- **是否为列表/is_list**：数据类型：bool 。将算子输入输出端口设置为列表形式。
  - True：元素类型变为 cloud_list。
  - False：元素类型为 cloud。
- **输入数量/number_input**：决定该算子的输入端口 cloud_? 的数量。范围：[1,10] 。默认值：2 。
- **选择器/selector**：其内容为输出端口所采用的 cloud _ ? 。
- **点云/cloud**：设置点云在 3D 视图中的可视化属性。
  -  打开点云可视化。
  -  关闭点云可视化。
  -  设置 3D 视图中点云的颜色。取值范围：[-2,360] 。默认值：-1 。
  -  设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。

数据信号输入输出

输入：

- **selector**：
  - 数据类型：String
  - 输入内容：其内容决定了输出端口所采用的实际数据。该端口通常是配合 Or 算子的 selector 输出端口使用；也可以通过 Emit-String 给定。
- **cloud_?**：
  - 数据类型：PointCloud
  - 输入内容：点云数据

输出：

- **cloud**：
  - 数据类型：PointCloud
  - 输出内容：当算子 selector 端口输入为 “0” 时，输入端口 cloud_0 中的数据会作为输出端口 cloud 的数据输出；selector 端口输入为 “1” 时对应输出 cloud_1 中的数据，以此类推。

功能演示

本节将使用 Multiplexer 算子中 PointCloud ，将 2 条点云数据信号流并联在一起，分别触发查看输出效果。这与 Multiplexer 算子中 Image 属性将 2 条图像数据信号流并联在一起，分别触发查看输出效果相同，请参照该章节的功能演示。

Pose

将 Multiplexer 算子的 **类型** 属性选择 Pose ，用于将多条坐标数据信号流并联在一起，合并后再统一传给后续处理环节。

算子参数

- **是否为列表/is_list**：数据类型：bool 。将算子输入输出端口设置为列表形式。
  - True：元素类型变为 pose_list。
  - False：元素类型为 pose。
- **输入数量/number_input**：决定该算子的输入端口 pose_?的数量。范围：[1,10] 。默认值：2 。
- **选择器/selector**：其内容为输出端口所采用的 pose_? 。
- **坐标/pose**：设置 pose 在 3D 视图中的可视化属性。
  -  打开 pose 可视化。
  -  关闭 pose 可视化。
  -  设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

- **selector**：
  - 数据类型：String
  - 输入内容：其内容决定了输出端口所采用的实际数据。该端口通常是配合 Or 算子的 selector 输出端口使用；也可以通过 Emit-String 给定。
- **pose_?**：
  - 数据类型：Pose
  - 输入内容：pose 数据

输出：

- **pose**：
  - 数据类型：Pose
  - 输出内容：当算子 selector 端口输入为 “0” 时，输入端口 pose_0 中的数据会作为输出端口 pose 的数据输出；selector 端口输入为 “1” 时对应输出 pose_1 中的数据，以此类推。

功能演示

本节将使用 Multiplexer 算子中 Pose ，将 2 条坐标数据信号流并联在一起，分别触发查看输出效果。这与 Multiplexer 算子中 Cube 属性将 2 条立方体数据信号流并联在一起，分别触发查看输出效果方法相同，请参照该章节的功能演示。

String

将 Multiplexer 算子的 **类型** 属性选择 String ，用于将多条字符串数据信号流并联在一起，合并后再统一传给后续处理环节。

算子参数

- **是否为列表/is_list**：数据类型：bool 。将算子输入输出端口设置为列表形式。
  - True：元素类型变为 string_list。
  - False：元素类型为 string。
- **输入数量/number_input**：决定该算子的输入端口 string_? 的数量。范围：[1,10] 。默认值：2 。
- **选择器/selector**：其内容为输出端口所采用的 string_? 。
- **字符串/string**：设置字符串曝光属性。可与交互面板中输出工具——“文本框”控件进行绑定。
  -  打开曝光。
  -  关闭曝光。

数据信号输入输出

输入：

- **selector**：
  - 数据类型：String
  - 输入内容：其内容决定了输出端口所采用的实际数据。该端口通常是配合 Or 算子的 selector 输出端口使用；也可以通过 Emit-String 给定。
- **string**：
  - 数据类型：String
  - 输入内容：string 数据

输出：

- **string**：
  - 数据类型：String
  - 输出内容：当算子 selector 端口输入为 “0” 时，输入端口 string_0 中的数据会作为输出端口 string 的数据输出；selector 端口输入为 “1” 时对应输出 string_1 中的数据，以此类推。

功能演示

本节将使用 Multiplexer 算子中 String ，将 2 条字符串数据信号流并联在一起，分别触发查看输出效果。这与 Multiplexer 算子中 Cube 属性将 2 条立方体数据信号流并联在一起，分别触发查看输出效果方法相同，请参照该章节的功能演示。

ImagePoints

将 Multiplexer 算子的 **类型** 属性选择 ImagePoints ，用于将多条图像关键点坐标数据信号流并联在一起，合并后再统一传给后续处理环节。

算子参数

- **是否为列表/is_list**：数据类型：bool 。将算子输入输出端口设置为列表形式。
  - True：元素类型变为 image_points_list。
  - False：元素类型为 image_points 。
- **输入数量/number_input**：决定该算子的输入端口 image_point_? 的数量。范围：[1,10] 。默认值：2 。
- **选择器/selector**：其内容为输出端口所采用的 image_point_? 。

数据信号输入输出

输入：

- **selector**：
  - 数据类型：String
  - 输入内容：其内容决定了输出端口所采用的实际数据。该端口通常是配合 Or 算子的 selector 输出端口使用；也可以通过 Emit-String 给定。
- **image_points_?**：
  - 数据类型：ImagePoints
  - 输入内容：imagepoints 数据

输出：

- **image_points**：
  - 数据类型：ImagePoints
  - 输出内容：当算子 selector 端口输入为 “0” 时，输入端口 image_points_0 中的数据会作为输出端口 image_points 的数据输出；selector 端口输入为 “1” 时对应输出 image_points_1 中的数据，以此类推。

功能演示

本节将使用 Multiplexer 算子中 ImagePoints ，将 2 条图像关键点坐标数据信号流并联在一起，分别触发查看输出效果。这与 Multiplexer 算子中 Cube 属性将 2 条立方体数据信号流并联在一起，分别触发查看输出效果方法相同，请参照该章节的功能演示。

Sleep 休眠停止

Sleep 算子作用是休眠停止，程序暂停若干时间后继续执行。

算子参数

- `休眠毫秒数/sleep_msec`：设置暂停时间。单位：毫秒。

功能演示

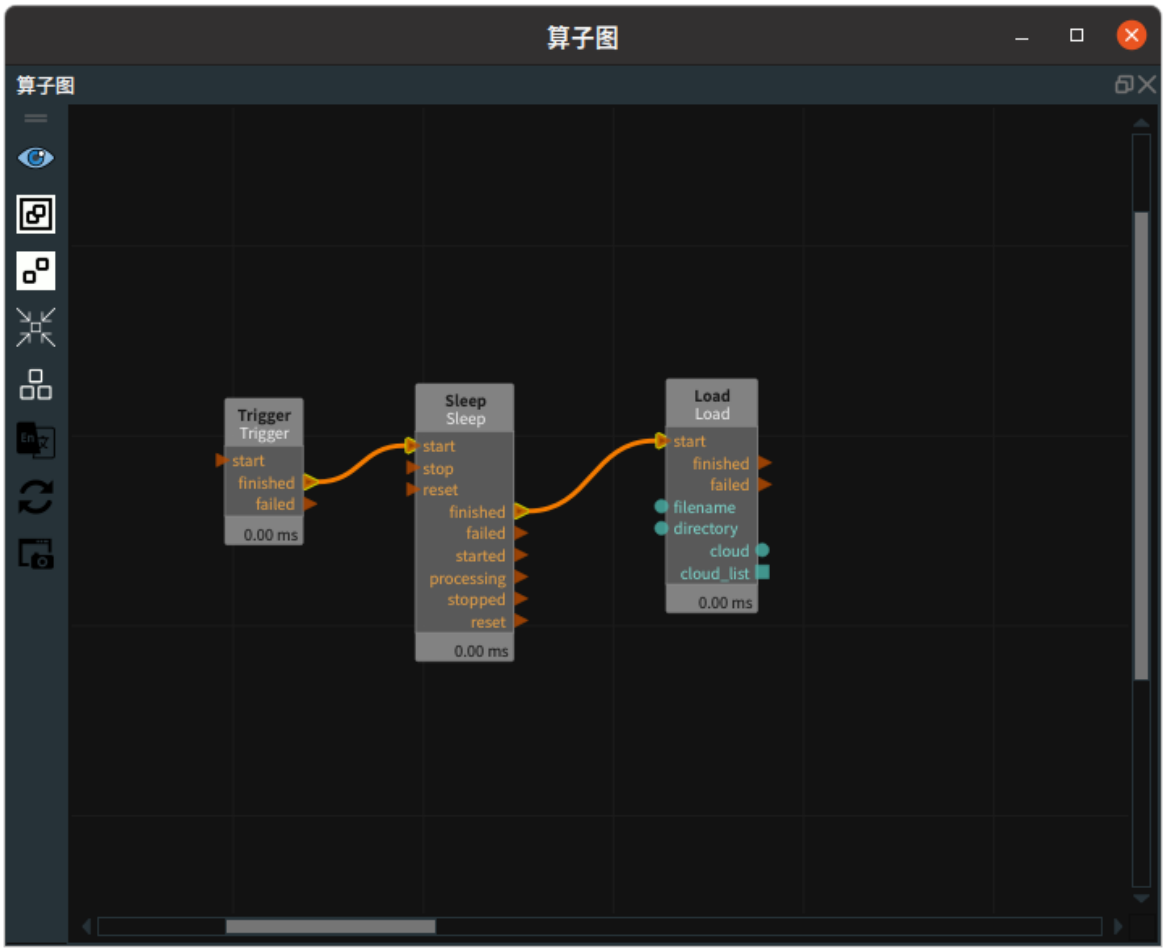
步骤1：算子准备

添加 Trigger 、Load 、Sleep 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 类型 → PointCloud
    - 文件 → ••• → 选择 PointCloud 文件名（*example_data/pointcloud/cloud.pcd*）
2. 设置 Sleep 算子参数：
    - 休眠毫秒数 → 5000

步骤3：连接算子

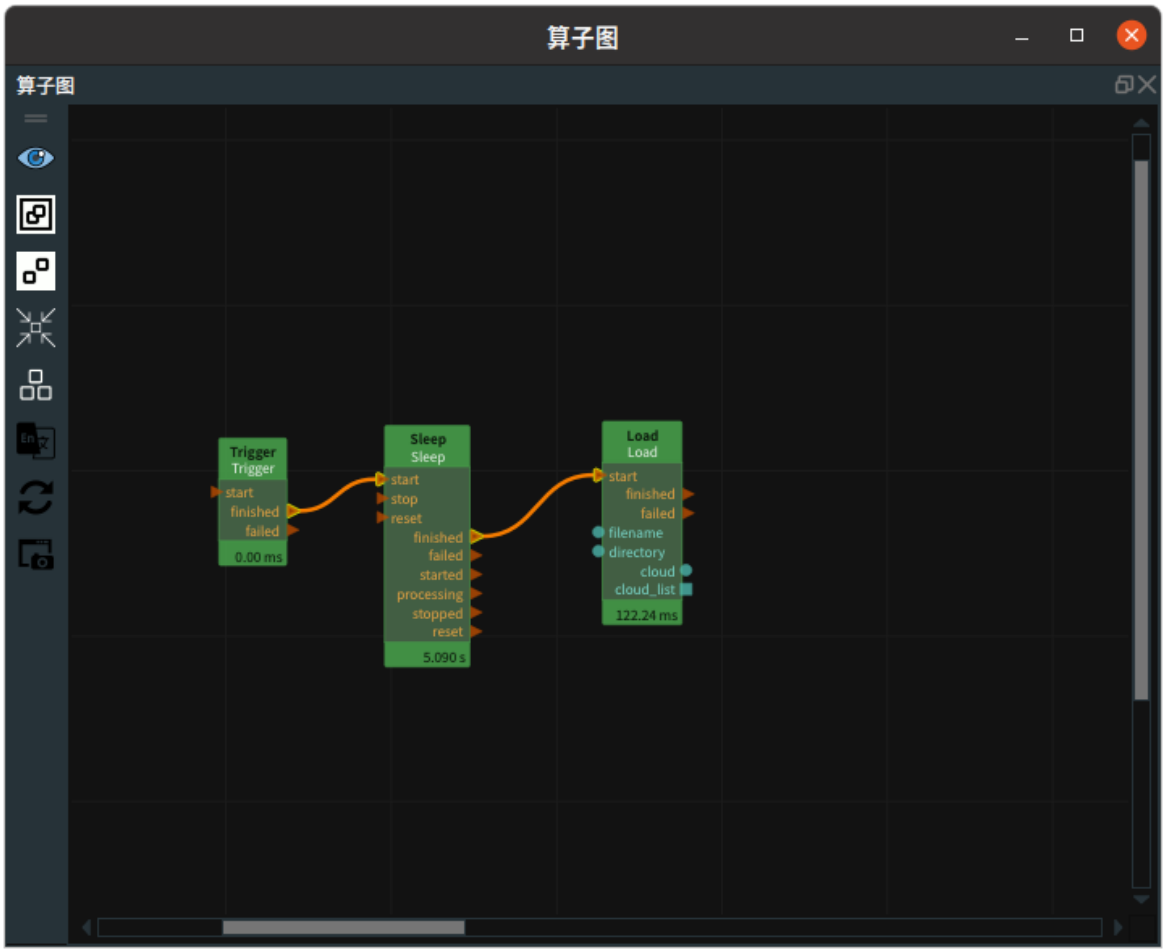


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，算子图中 Sleep 算子在运行 5s 后触发 Load 算子。



Collect 收集元素

Collect 算子为收集元素，用于收集元素组合成列表。适用类型有：Cube 、Cylinder 、Image 、ImagePoints 、JointArray 、Object 、PointCloud 、PolyData 、Pose 、String 。

类型	功能
Cube	收集立方体，组合成立方体列表。
Cylinder	收集圆柱体，组合成圆柱体列表。
Image	收集图像，组合成图像列表。
ImagePoints	收集图像关键点坐标，组合成关键点坐标列表。
JointArray	收集机器人关节弧度值，组合成机器人关节弧度值列表。
Object	/
PointCloud	收集点云，组合成点云列表。
PolyData	收集 3D 模型，组合成 3D 模型列表。
Pose	收集坐标，组合成坐标列表。
String	收集字符串，组合成字符串列表。

Cube

将 Collect 算子的 `类型` 属性选择为 Cube ，用于收集立方体，组合成立方体列表。

算子参数

- `启动便迭代/iterate_at_start`：数据类型：Bool 。使能/去使能迭代触发。
  - True：当算子左侧 start 端口或左侧 iterated 端口被触发时，会自动触发一次 iterate ，此时会输出当前计数序号对应的数据，然后右侧 iterated 被触发，计数序号自加 1 。
  - False：当算子左侧 start 端口被触发时，iterated 不会被触发。
- `立方体列表/cube_list`：设置立方体列表在 3D 视图中的可视化属性。
  -  打开立方体列表可视化。
  -  关闭立方体列表可视化。
  -  设置立方体列表的颜色。取值范围：[-2,360] 。默认值：-2 。
  -  设置立方体列表的透明度。取值范围：[0,1] 。默认值：0.5 。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- `cube`：
  - 数据类型：Cube
  - 输入内容：立方体数据
- `cube_list`：

- 数据类型：CubeList
- 输入内容：立方体数据列表

输出：

- `cube_list`：
  - 数据类型：CubeList
  - 输出内容：立方体数据列表

功能演示

使用 Collect 算子中 Cube ，收集2个立方体，将2个立方体组合成一个立方体列表。

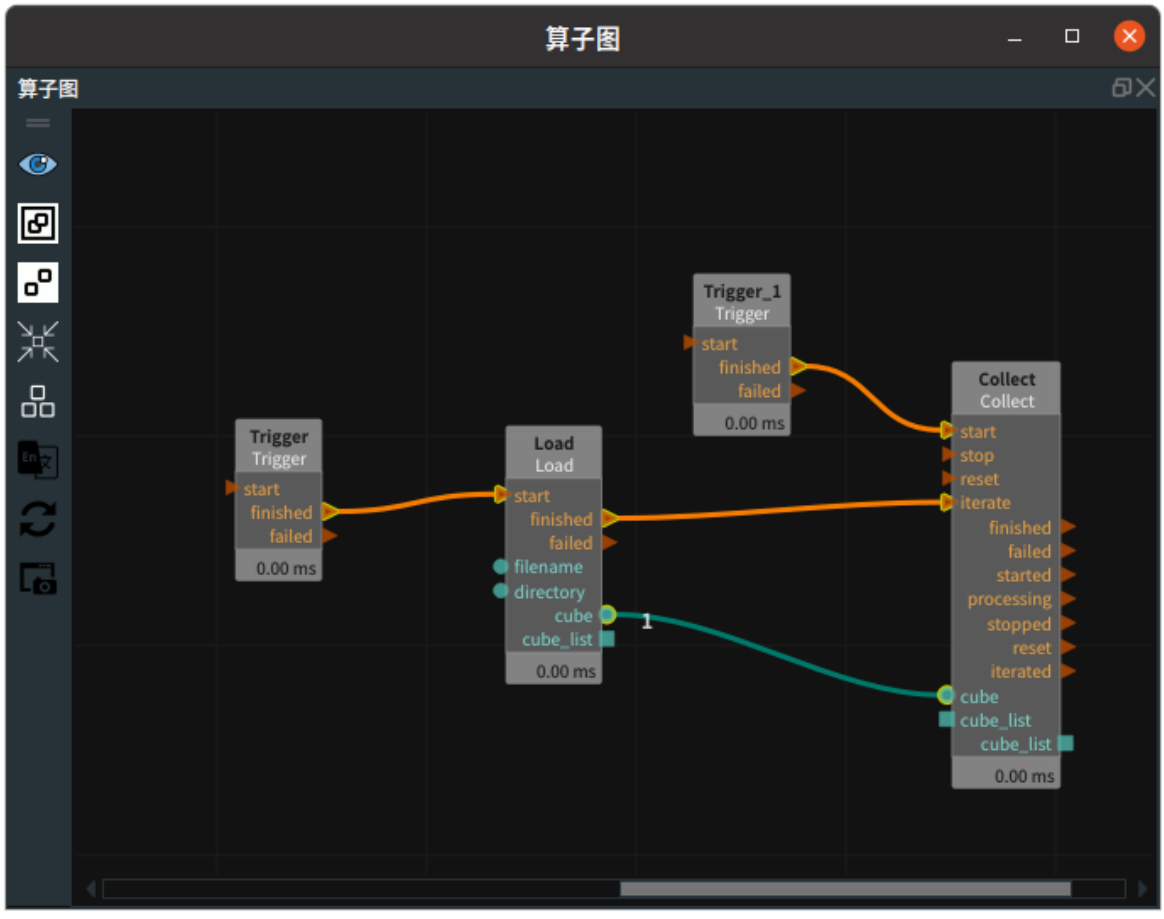
步骤1：算子准备

添加 Trigger（ 2 个） 、Load 、Collect 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → Cube
   - 文件 → ●●● → 选择 cube 文件名 （ *example_data/cube/cube.txt* ）
2. 设置 Collect 算子参数：
   - 类型 → Cube
   - 启动便迭代 → True
   - 立方体列表 → 可视

步骤3：连接算子

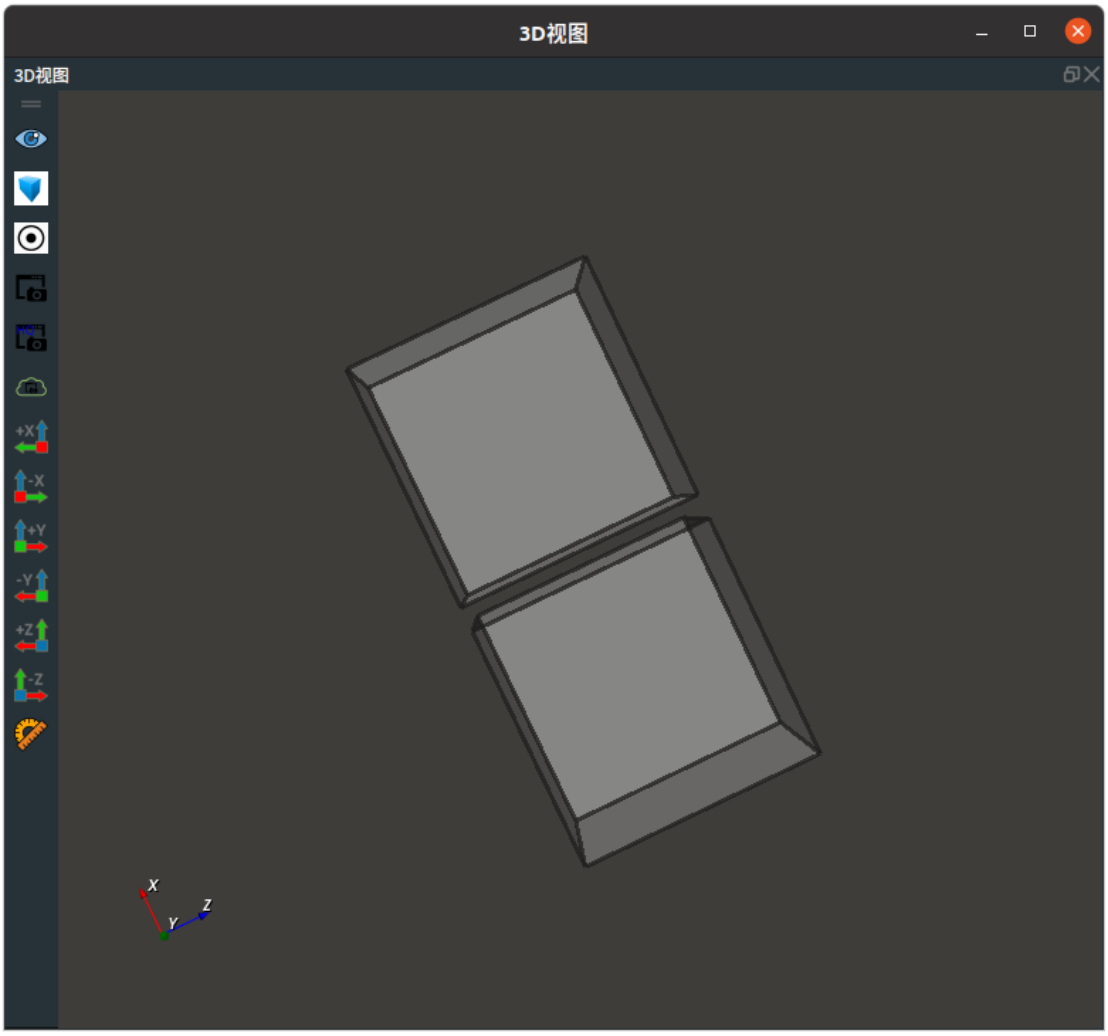


步骤4：运行

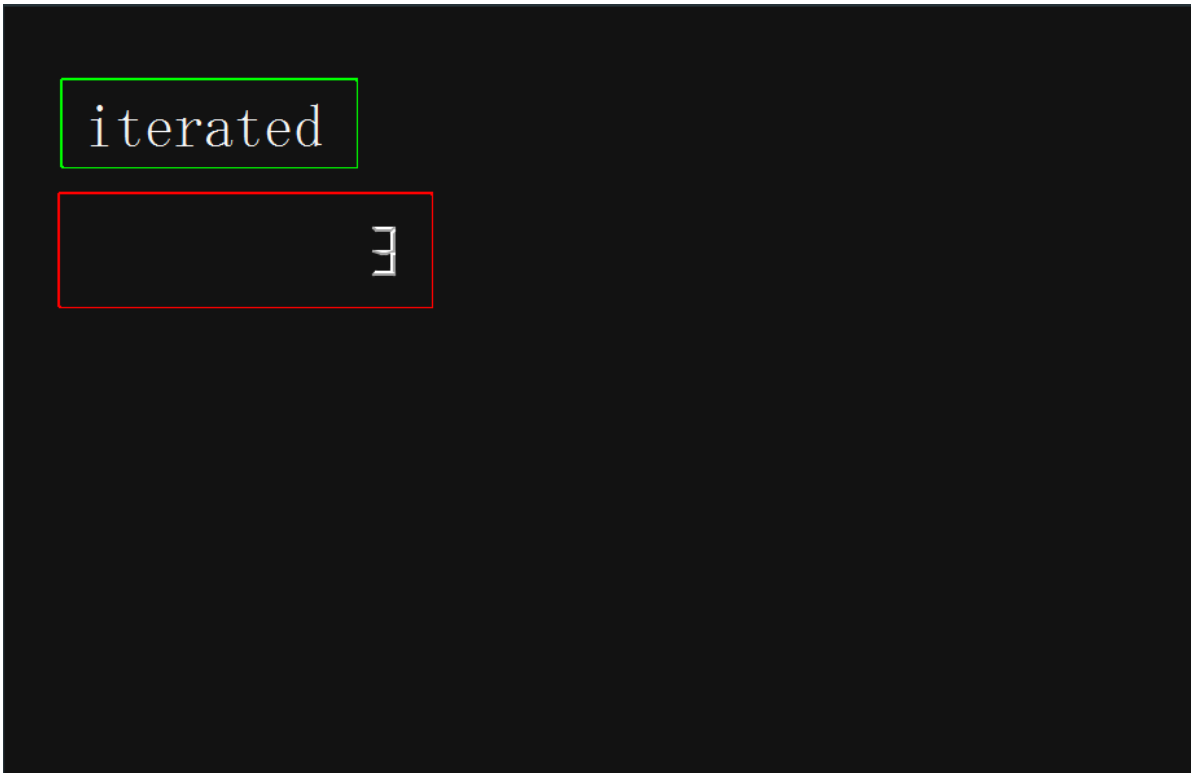
1. 交互面板中选择输出工具数码管命名为 iterated ； 选择输出工具——“数码管”控件，与 counter_string 属性绑定。
2. 点击运行按钮，先触发 Trigger_1 ，启动 Collect 算子。
3. 触发 Trigger 后，Collect 完成一次收集。
4. Load 算子中 filename 属性→ 选择 cube 文件名 （*example_data/cube/cube1.txt* ），再次触发 Trigger ，完成第二次收集。

运行结果

1. 结果如下图所示，3D视图中显示收集的两个 cube 所组成的一个 cubelist 。



2. 如下图所示，触发 start 一次和触发 iterate 两次，一共三次，此时数码管显示为 3。



Cylinder

将 Collect 算子的 **类型** 属性选择为 Cylinder ，用于收集圆柱体，组合成圆柱体列表。

算子参数

- **启动便迭代/iterate_at_start**：数据类型：Bool 。
  - True：当算子左侧 start 端口被触发时，会自动触发一次 iterate ，此时会输出当前计数序号对应的数据，然后右侧 iterate 被触发，计数序号自加 1 。
  - False：当算子左侧 start 端口被触发时，iterate 不会被触发。
- **圆柱体列表/cylinder_list**：设置圆柱体列表在 3D 视图中的可视化属性。
  - 打开圆柱体列表可视化。
  - 关闭圆柱体列表可视化。
  - 设置圆柱体列表的颜色。取值范围：[-2,360] 。默认值：-2。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **cylinder**：
  - 数据类型：Cylinder
  - 输入内容：圆柱体数据
- **cylinder_list**：
  - 数据类型：CylinderList
  - 输入内容：圆柱体数据列表

输出：

- **cylinder_list**：

- 数据类型：CylinderList
- 输出内容：圆柱体数据列表

功能演示

使用 Collect 算子中 Cylinder ，收集2个圆柱体，将2个圆柱体组合成一个圆柱体列表。

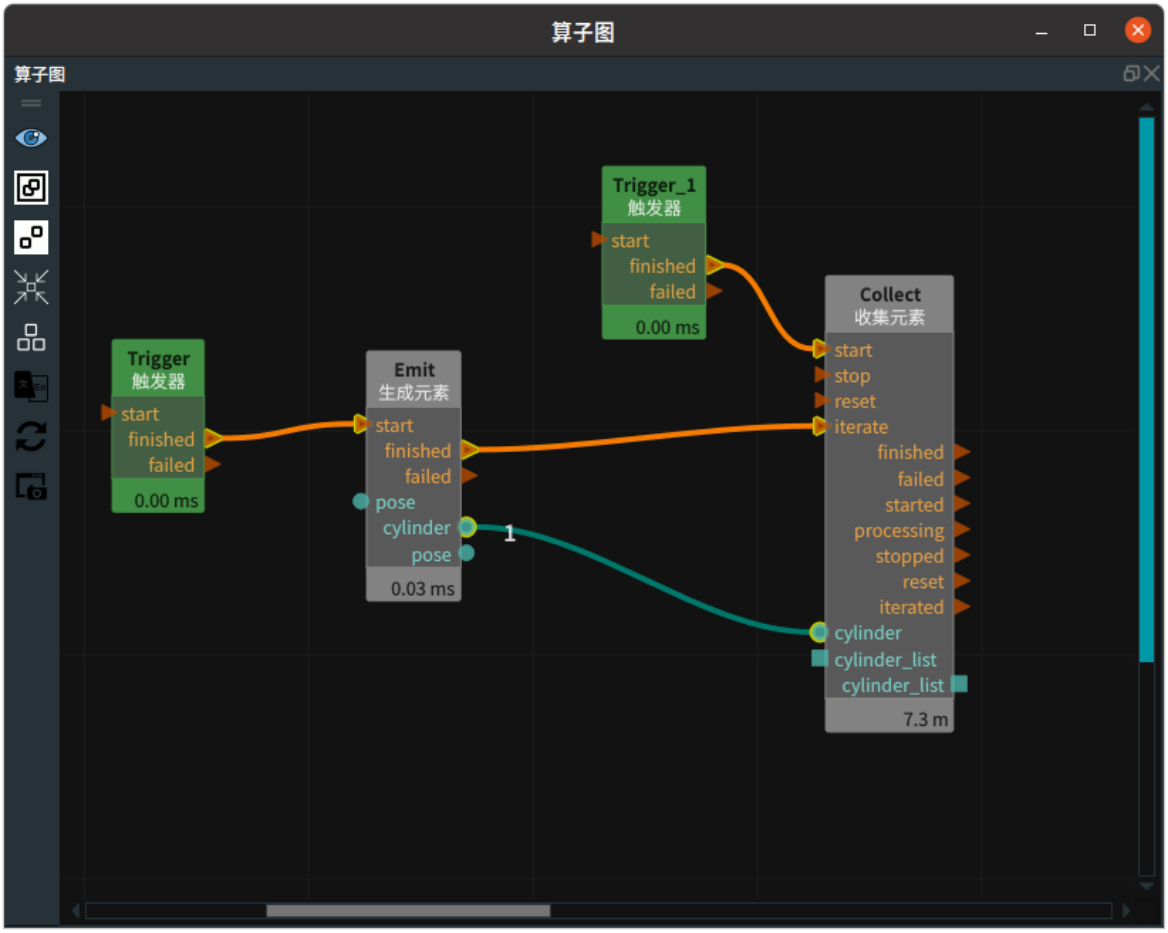
步骤1：算子准备

添加 Trigger（2 个）、Emit、Collect 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
   - 类型 → Cylinder
   - 坐标 → 0 0 0 0 0 0
   - 半径 → 0.075
   - 长度 → 0.4
2. 设置 Collect 算子参数：
   - 类型 → Cylinder
   - 圆柱体列表 → 可视

步骤3：连接算子



步骤4：运行

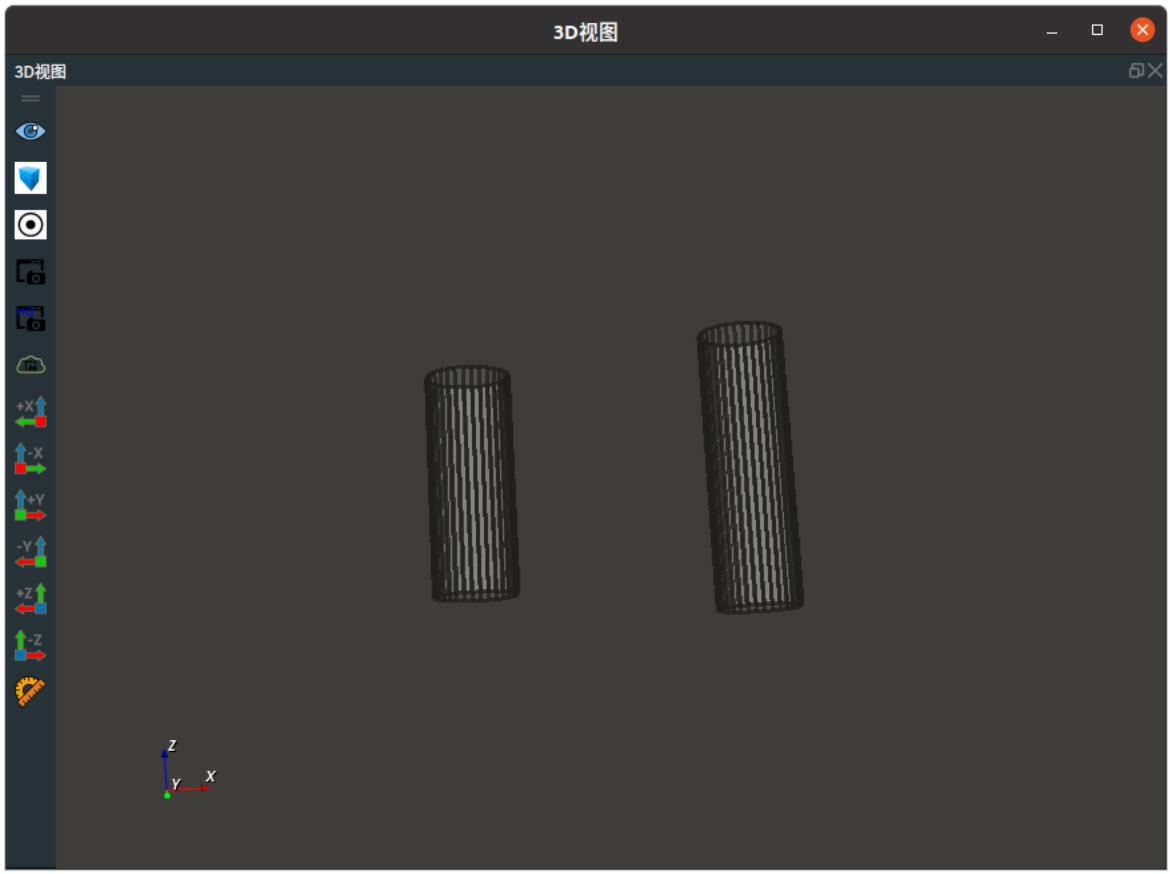
1. 点击运行按钮，先触发 Trigger_1 ，启动 Collect 算子。
2. 触发 Trigger 后，Collect 完成一次收集。
3. 再次调整 Emit 算子：

- 坐标 → 0.5 0 0 0 0 0
- 长度 → 0.5

4. 再次触发 Trigger ，完成第二次收集。

运行结果

结果如下图所示，3D视图中显示收集的 2 个 cylinder 所组成的一个 cylinder 列表 。



Image

使用 Collect 算子中 Image ，用于收集图像，组合成图像列表。

算子参数

- **启动便迭代/iterate_at_start**：数据类型：Bool 。
  - True：当算子左侧 start 端口被触发时，会自动触发一次 iterate ，此时会输出当前计数序号对应的数据，然后右侧 iterate 被触发，计数序号自加 1 。
  - False：当算子左侧 start 端口被触发时，iterate 不会被触发。
- **图像列表/image_list**：设置图像列表在 3D 视图中的可视化属性。
  - 打开图像列表可视化。
  - 关闭图像列表可视化。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **image**：
  - 数据类型：Image
  - 输入内容：图像数据

- **image_list**：
  - 数据类型：ImageList
  - 输入内容：图像数据列表

输出：

- **image_list**：
  - 数据类型：ImageList
  - 输出内容：图像数据列表

功能演示

本节将使用 Collect 算子中 Image ，收集2张图像，将2个图像组合成一个图像列表。这与 Collect 中 cube 属性的收集立方体组成立方体列表的方法相同，请参照该章节的功能演示。

ImagePoints

将 Collect 算子的 **类型** 属性选择为 ImagePoints ，用于收集图像关键点坐标，组合关键点坐标成列表。

算子参数

- **启动便迭代/iterate_at_start**：数据类型：Bool 。
  - True：当算子左侧 start 端口被触发时，会自动触发一次 iterate ，此时会输出当前计数序号对应的数据，然后右侧 iterate 被触发，计数序号自加 1 。
  - False：当算子左侧 start 端口被触发时，iterate 不会被触发。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **image_points**：
  - 数据类型：ImagePoints
  - 输入内容：图像关键点坐标数据
- **image_points_list**：
  - 数据类型：ImagePointsList
  - 输入内容：图像关键点坐标数据列表

输出：

- **image_points_list**：
  - 数据类型：ImagePointsList
  - 输出内容：图像关键点坐标数据列表

功能演示

本节将使用 Collect 算子中 ImagePoints ，收集2个图像关键点坐标，将 2 个关键点坐标组合成一个关键点坐标列表。这与 Collect 中 cube 属性的收集立方体组成立方体列表的方法相同，请参照该章节的功能演示。

JointArray

将 Collect 算子的 **类型** 属性为 JointArray ，收集机器人关节弧度值，组合成机器人关节弧度值列表。

算子参数

- **启动便迭代/iterate_at_start**：数据类型：Bool 。
  - True：当算子左侧 start 端口被触发时，会自动触发一次 iterate ，此时会输出当前计数序号对应的数据，然后右侧 iterate 被触发，计数序号自加 1。
  - False：当算子左侧 start 端口被触发时，iterate 不会被触发。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **joint**：
  - 数据类型：JointArray
  - 输入内容：机器人关节弧度值
- **joint_list**：
  - 数据类型：JointArrayList
  - 输入内容：机器人关节弧度值列表

输出：

- **joint_list**：
  - 数据类型：JointArrayList
  - 输出内容：机器人关节弧度值列表

功能演示

本节将使用 Collect 算子中 JointArray ，收集2组机器人关节弧度值，将2组机器人关节弧度值组合成一个机器人关节弧度值列表。这与 Collect 中 cube 属性的收集立方体组成立方体列表的方法相同，请参照该章节的功能演示。

PointCloud

将 Collect 算子的 **类型** 属性为 PointCloud ，用于收集点云，组合成点云列表。

算子参数

- **启动便迭代/iterate_at_start**：数据类型：Bool 。
  - True：当算子左侧 start 端口被触发时，会自动触发一次 iterate ，此时会输出当前计数序号对应的数据，然后右侧 iterate 被触发，计数序号自加 1 。
  - False：当算子左侧 start 端口被触发时，iterate 不会被触发。
- **点云列表/cloud_list**：设置点云列表在 3D 视图中的可视化属性。
  -  打开点云列表可视化。
  -  关闭点云列表可视化。
  -  设置3D视图中点云列表的颜色。取值范围：[-2,360] 。默认值：-1 。
  -  设置点云中点的尺寸。取值范围：[1,50] 。默认值：1 。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- `cloud`：
  - 数据类型：PointCloud
  - 输入内容：点云数据
- `cloud_list`：
  - 数据类型：PointCloudList
  - 输入内容：点云数据列表

输出：

- `cloud_list`：
  - 数据类型：PointCloudList
  - 输出内容：点云数据列表

功能演示

本节将使用 Collect 算子中 PointCloud ，收集2个点云，将2个点云组合成一个点云列表。这与 Collect 中 cube 属性的收集立方体组成立方体列表的方法相同，请参照该章节的功能演示。

PolyData

将 Collect 算子的 `类型` 属性为 PolyData ，用于收集 3D 模型，组合成 3D 模型列表。

算子参数

- `启动便迭代/iterate_at_start`：数据类型：Bool 。
  - True：当算子左侧 start 端口被触发时，会自动触发一次 iterate ，此时会输出当前计数序号对应的数据，然后右侧 iterate 被触发，计数序号自加 1 。
  - False：当算子左侧 start 端口被触发时，iterate 不会被触发。
- `多边形列表/polydata_list`：设置 3D 模型列表在 3D 视图中的可视化属性。
  -  打开 3D 模型列表可视化。
  -  关闭 3D 模型列表可视化。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- `polydata`：
  - 数据类型：PolyData
  - 输入内容：polydata 数据
- `polydata_list`：
  - 数据类型：PolyDataList
  - 输入内容：polydata 数据列表

输出：

- `polydata_list`：
  - 数据类型：PolyDataList
  - 输出内容：polydata 数据列表

功能演示

使用 Collect 算子中 PolyData ，收集2个 3D 模型，将2个 3D 模型组合成一个 3D 模型列表。这与 Collect 中 cube 属性的收集立方体组成立方体列表的方法相同，请参照该章节的功能演示。

Pose

将 Collect 算子的 `类型` 属性为 Pose ，用于收集 pose ，组合成 pose 列表。

算子参数

- `启动便迭代/iterate_at_start`：数据类型：Bool 。
  - True：当算子左侧 start 端口被触发时，会自动触发一次 iterate ，此时会输出当前计数序号对应的数据，然后右侧 iterate 被触发，计数序号自加 1 。
  - False：当算子左侧 start 端口被触发时，iterate 不会被触发。
- `坐标列表/pose_list`：设置 pose 列表在 3D 视图中的可视化属性。
  -  打开 pose 列表列表可视化。
  -  关闭 pose 列表列表可视化。
  -  设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出：

输入：

说明：根据需求选择其中一种数据信号输入即可。

- `pose`：
  - 数据类型：Pose
  - 输入内容：pose 数据
- `pose_list`：
  - 数据类型：PoseList
  - 输入内容：pose 数据列表

输出：

- `pose_list`：
  - 数据类型：PoseList
  - 输出内容：pose 数据列表

功能演示

本节将使用 Collect 算子中 Pose ，收集2个坐标，将2个坐标组合成一个坐标列表。这与 Collect 中 cube 属性的收集立方体组成立方体列表的方法相同，请参照该章节的功能演示。

String

将 Collect 算子的 **类型** 属性为 String ，用于收集字符串，组合成字符串列表

算子参数

- **启动便迭代/iterate_at_start**：数据类型：Bool 。
  - True：当算子左侧 start 端口被触发时，会自动触发一次 iterate ，此时会输出当前计数序号对应的数据，然后右侧 iterate 被触发，计数序号自加 1 。
  - False：当算子左侧 start 端口被触发时，iterate 不会被触发。
- **字符串列表/string_list**：设置字符串列表曝光属性。可用于绑定交互面板中输出工具——“表格”控件。
  -  打开曝光。
  -  关闭曝光。

数据信号输入输出

输入：

说明：根据需求选择其中一种数据信号输入即可。

- **string**：
  - 数据类型：String
  - 输入内容：string 数据
- **string_list**：
  - 数据类型：StringList
  - 输入内容：string 数据列表

输出：

- **string_list**：
  - 数据类型：StringList
  - 输出内容：string 数据列表

功能演示

使用 Collect 算子的 type 属性为 String ，收集3个字符串，将3个字符串组合成一个字符串列表。

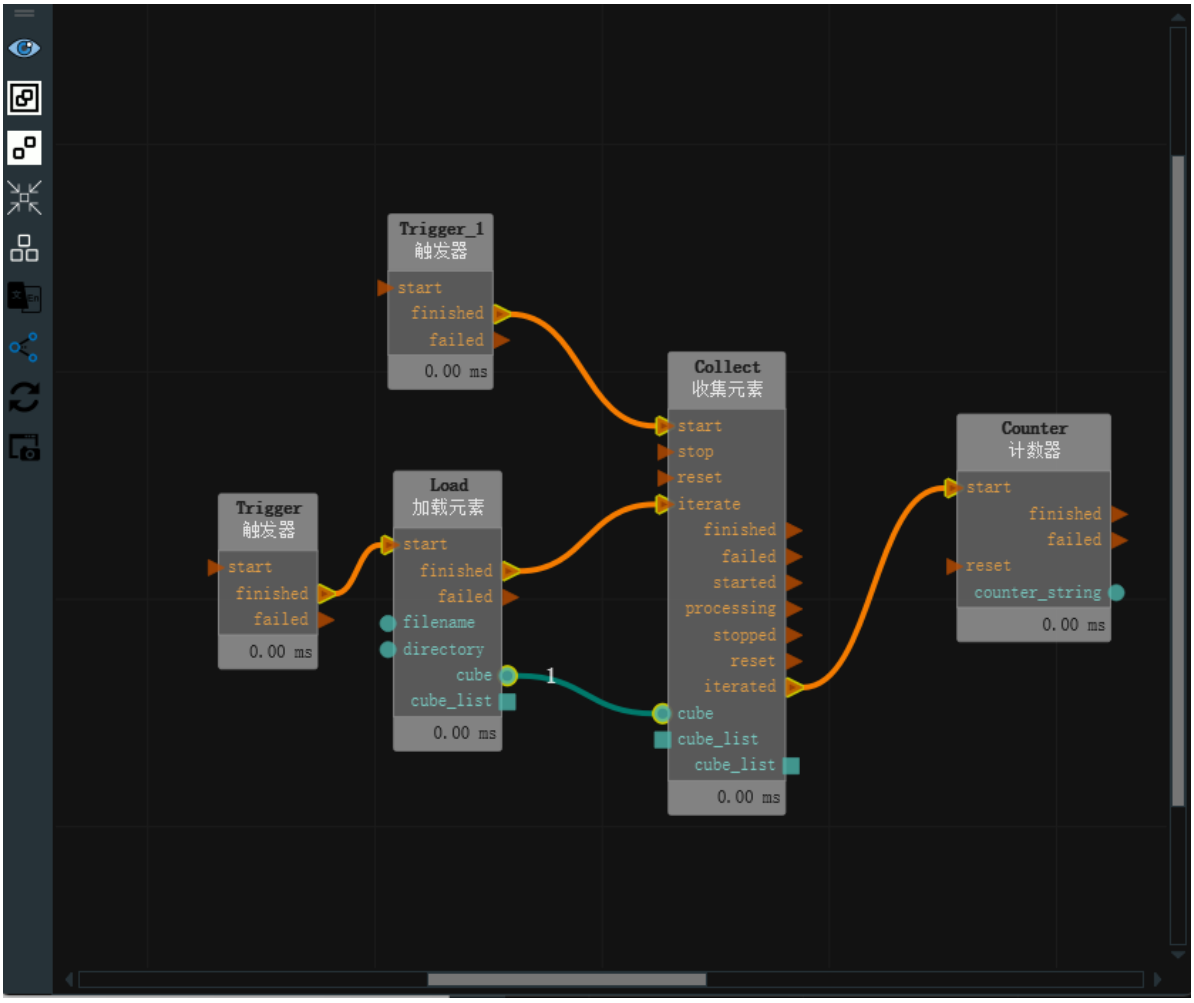
步骤1：算子准备

添加 Trigger 、Emit 、Collect 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
   - 类型 → String
   - 字符串 → RVS
2. 设置 Collect 算子参数：
   - 类型 → String
   - 字符串列表 →  曝光

步骤3：连接算子

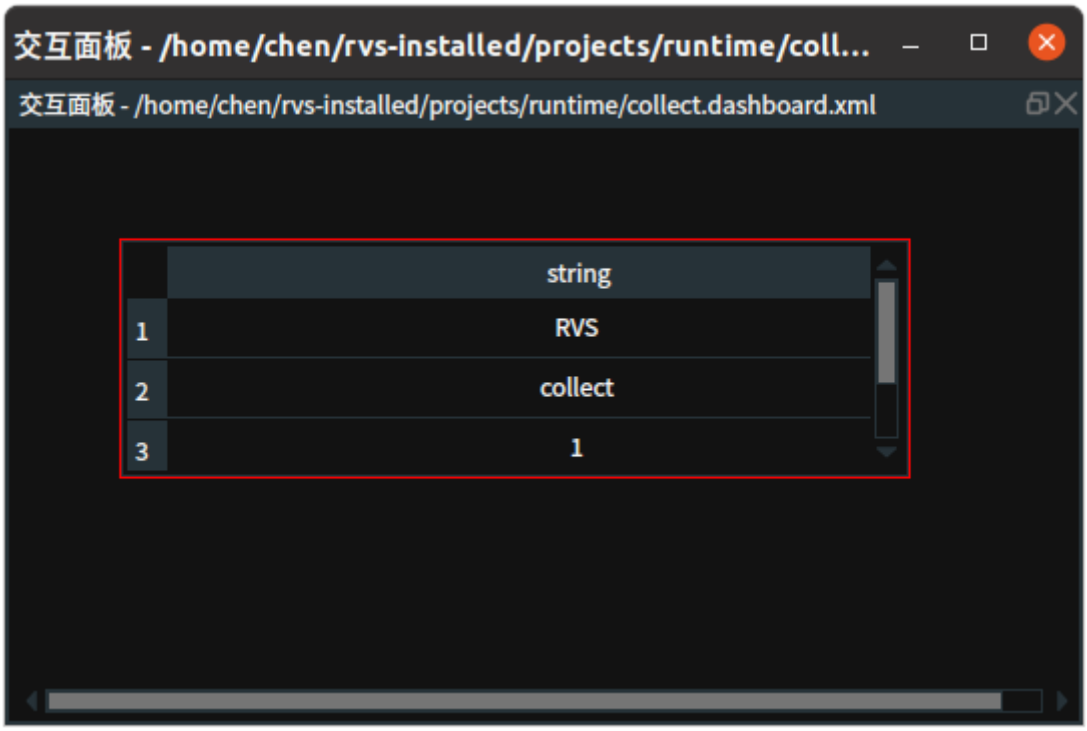


步骤4：运行

1. 交互面板中选择输出工具——“表格”控件，并与“ string_list ”绑定。
2. 点击 RVS 运行按钮，先触发 Trigger_1 ，启动 Collect 算子。
3. 触发 Trigger 后，Collect 完成一次收集。
4. 调整 Emit 算子中 string 属性，输入 *collect* ，再次触发 Trigger ，完成第二次收集。
5. 调整 Emit 算子中 string 属性，输入 *1* ，再次触发 Trigger ，完成第三次收集。

运行结果

结果如下图所示，在交互面板中显示收集的多个 string 。


交互面板 - /home/chen/rvs-installed/projects/runtime/coll...
交互面板 - /home/chen/rvs-installed/projects/runtime/collect.dashboard.xml
string
1
RVS
2
collect
3
1

Counter 计数器

Counter 算子为计数器，用于计算算子触发的次数。从 1 开始计数，触发的第一次显示为 1。

算子参数

- `计数器字符串/counter_string`：设置计数器数值曝光属性。可与交互面板中输出工具——“数码管”控件绑定。
  -  打开曝光。
  -  关闭曝光。

数据信号输入输出

输出：

- `counter_string`：
  - 数据类型：String
  - 输出内容：当前计数数值

功能演示

使用 Counter 算子计算算子触发的次数。

步骤1：算子准备

添加 Trigger 、Counter 算子至算子图。

步骤2：设置算子参数

设置 Counter 算子参数：

- 类型 → Cube
- 计数器字符串 →  曝光

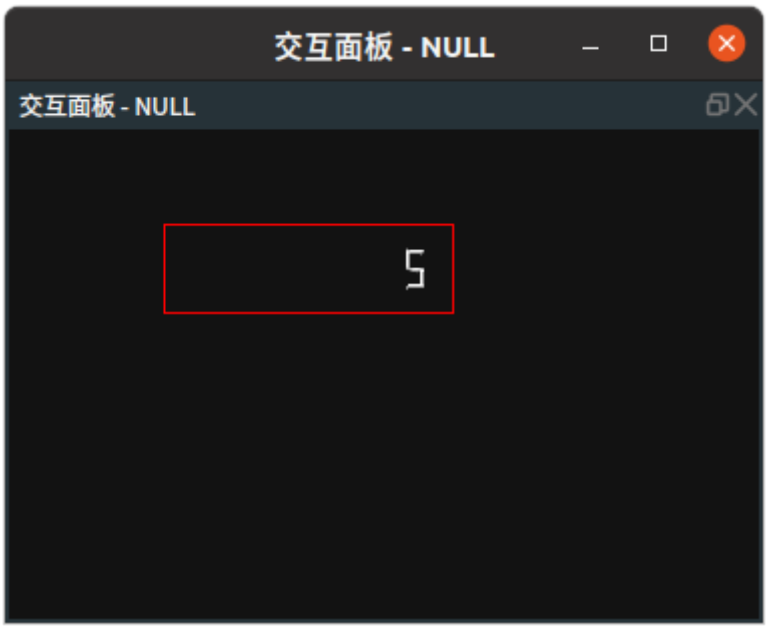
步骤3：连接算子

步骤4：运行

1. 交互面板中选择输出工具——“数码管”控件，与 counter_string 属性绑定。
2. 点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，在交互面板显示触发次数。



ModuloCounter 模余计数器

ModuloCounter 算子为模运算计数触发器。根据触发次数与 count 属性值进行模运算，如果余数为 0 触发 finished 信号，反之，触发 failed 信号。

算子参数

- **计数/count**：模运算中被除数数值。取值范围：[1,+∞）。

控制信号端口

- **start**：触发 start 信号端口运行算子。
- **finished**：count 属性值对触发次数取模，如果余数为 0 ，触发 finished 信号。
- **failed**：count 属性值对触发次数取模，如果余数不为 0 ，触发 failed 信号。

功能演示

根据设置的 count 属性值对触发次数进行模运算，如果余数为 0 触发 finished 信号，反之，触发 failed 信号。

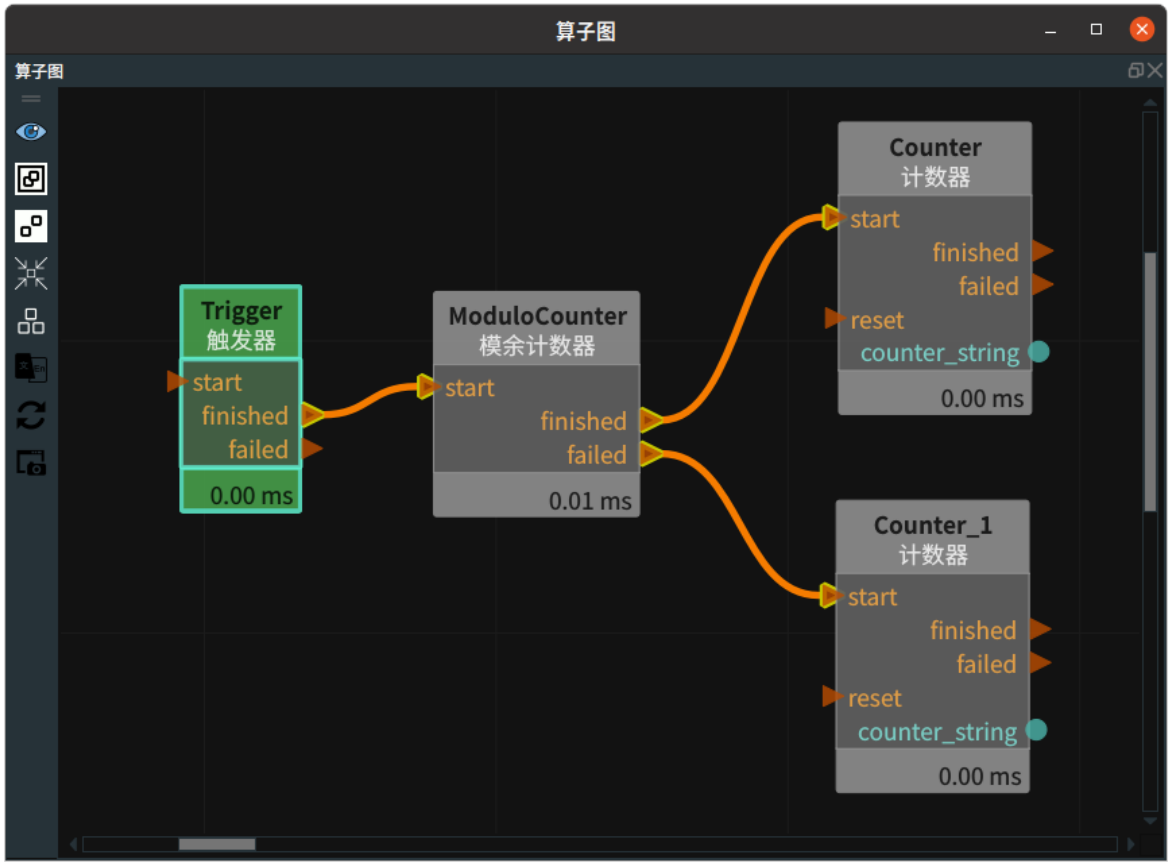
步骤1：算子准备

添加 Trigger 、ModuloCounter 、counter（ 2 个）算子至算子图。

步骤2：设置算子参数

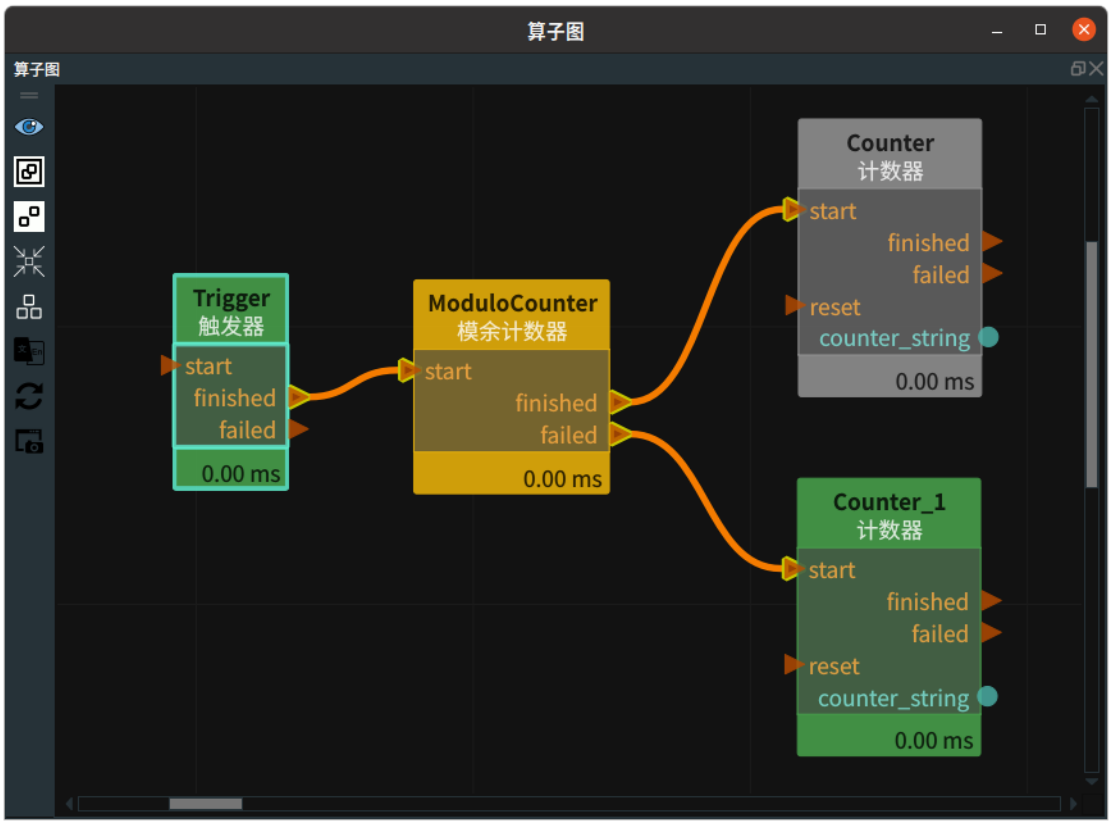
1. 设置 ModuloCounter 算子参数：计数 → 4
2. 设置 Counter / Counter_1 算子参数：计数器字符串 → 曝光

步骤3：连接算子

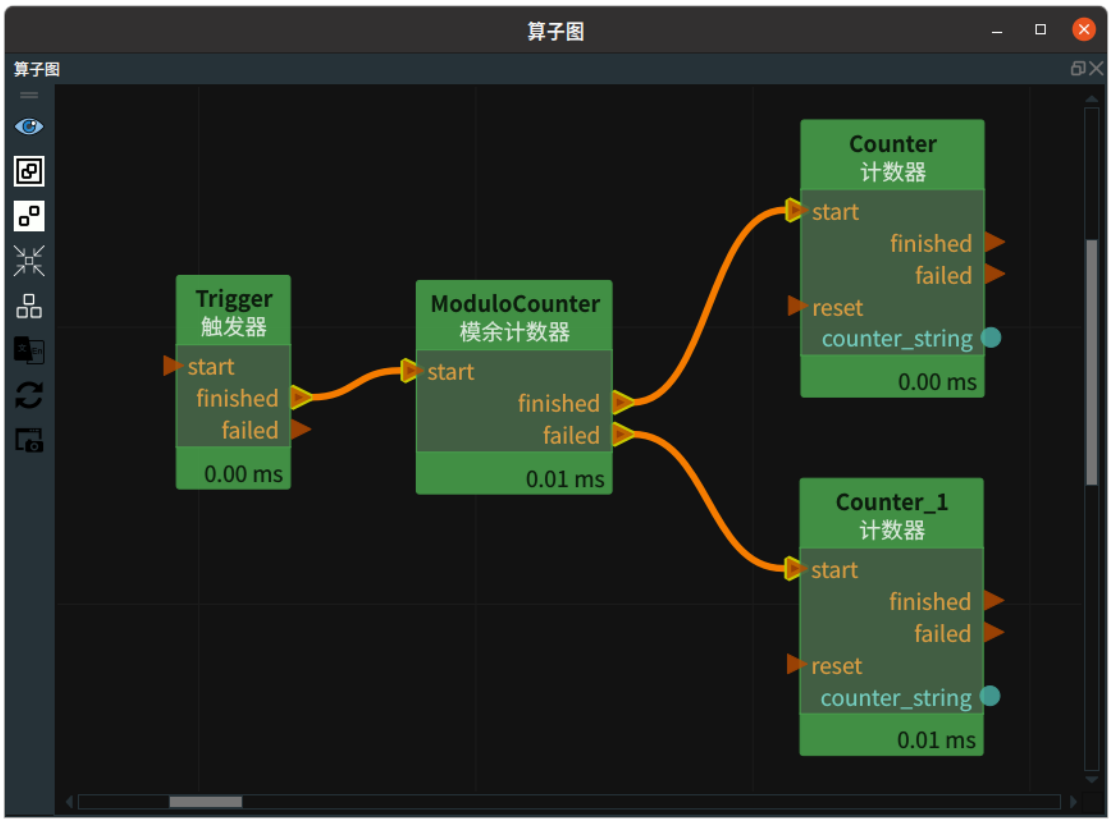


步骤4：运行

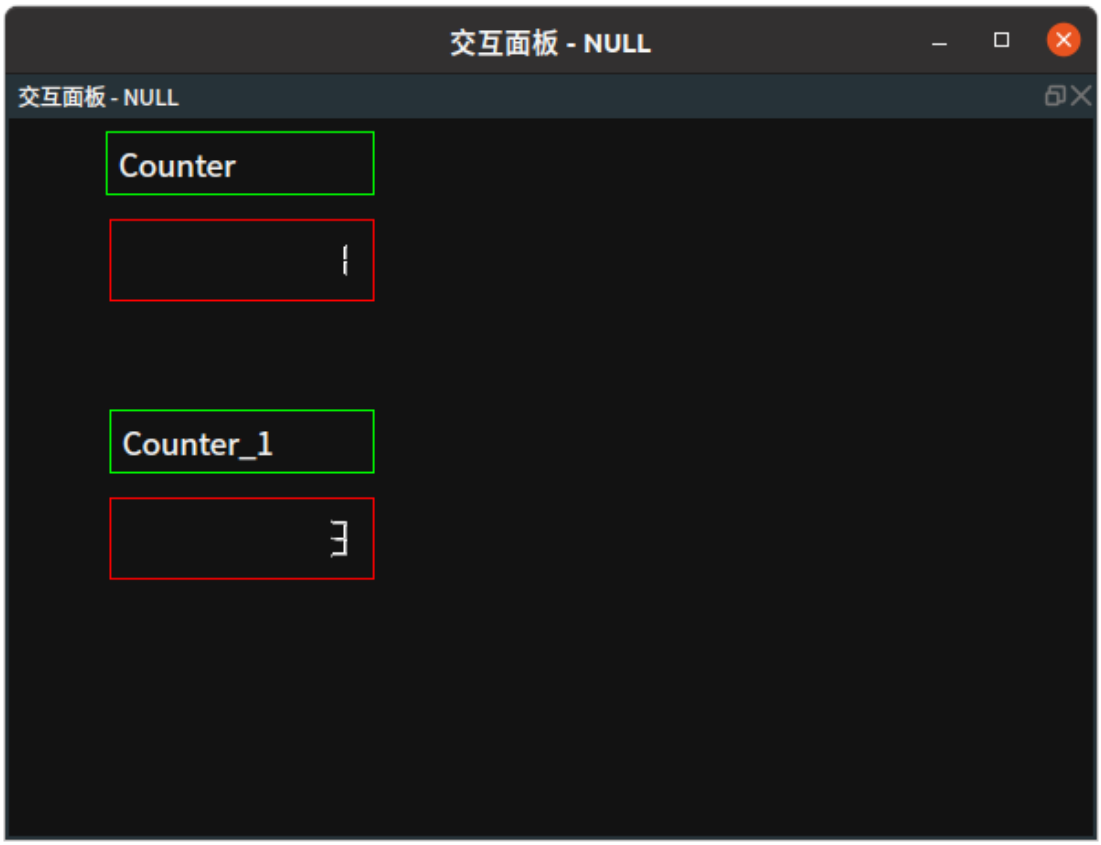
1. 在交互面板中拖出2个输入工具：标签。重命名 → Counter/Counter_1。
2. 拖出 2 个输出工具 → 数码管。分别与曝光属性 counter_string 进行绑定。
3. 点击 RVS 运行按钮，触发 Trigger 算子。
4. 当第1、2、3次触发 Trigger 时，ModuloCounter 算子触发 failed 信号。



5. 当第4次触发 Trigger 时，ModuloCounter 算子触发 finished 信号。



6. 在交互面板中查看 2 个 Counter 算子分别被触发的次数。Counter_1 被触发 3 次，Counter 被触发 1 次。



Or 或

Or 算子用于将多条控制信号流通过逻辑或的方式并联在一起，合并后再触发后续信号。

算子参数

- **输入数量/number_input**：决定该算子的输入端口 input_? 的数量，默认值：2 。
- **选择器/selector**：设置 selector 曝光属性。打开后可与交互面板中输出工具——“数码管”控件绑定
  -  打开曝光。
  -  关闭曝光。

控制信号输入

输入：

- **input_?**：功能：任意一个输入 input_? 端口被触发，都会触发 output 输出端口。

数据信号输出

输出：

- **selector**：
  - 数据类型：String
  - 输出内容：当算子左侧 input_0 端口被触发时，该端口会输出 “0” ，input_1 端口对应 “1” ，以此类推。

功能演示

使用 Or 算子 与 ModuloCounter 算子连接，分别触发不同 input_? 端口，查看其输出结果。

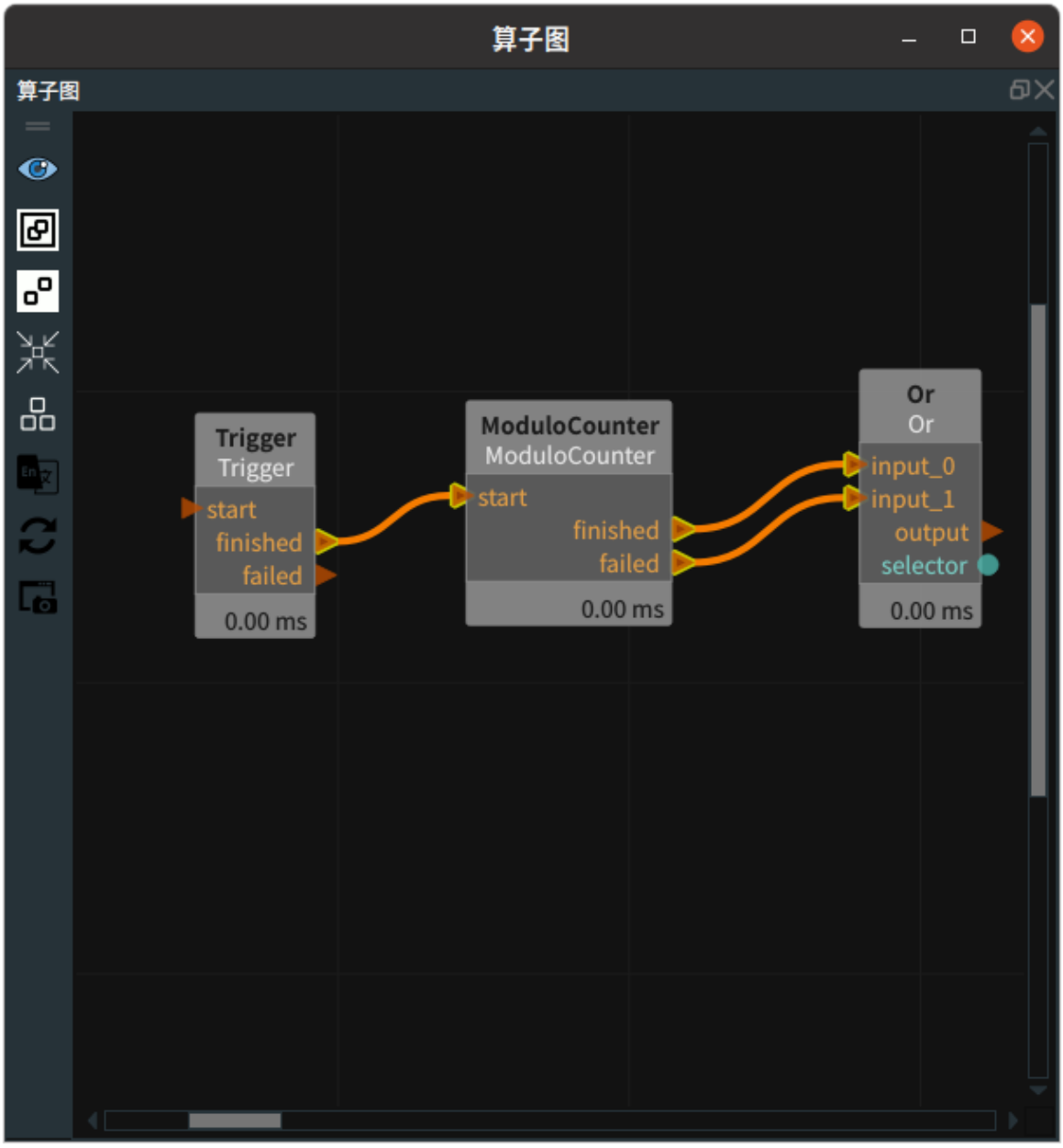
步骤1：算子准备

添加 Trigger 、ModuloCounter 、Or 算子至算子图。

步骤2：设置算子参数

设置 ModuloCounter 算子参数：计数 → 2

步骤3：连接算子

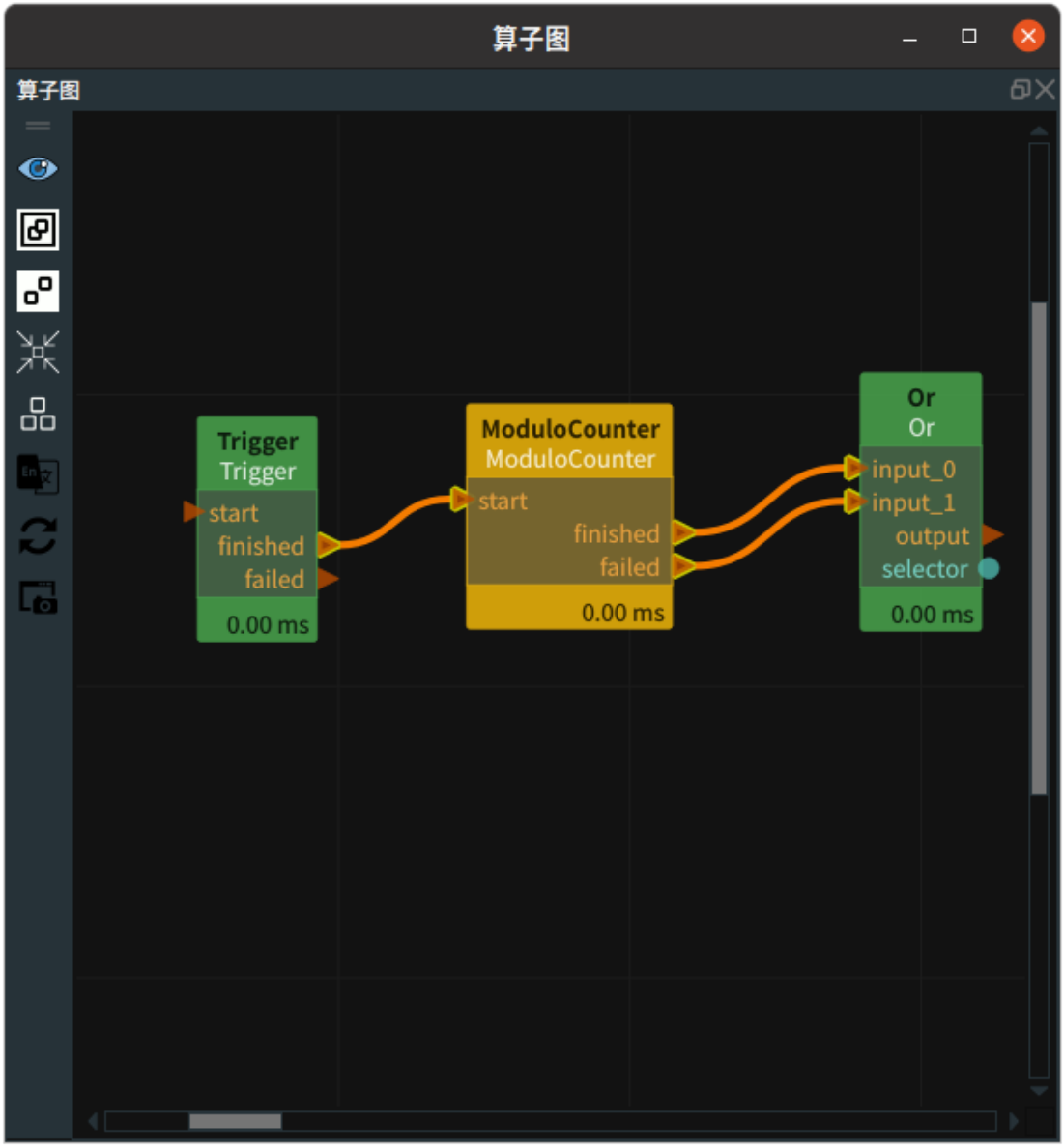


步骤4：运行

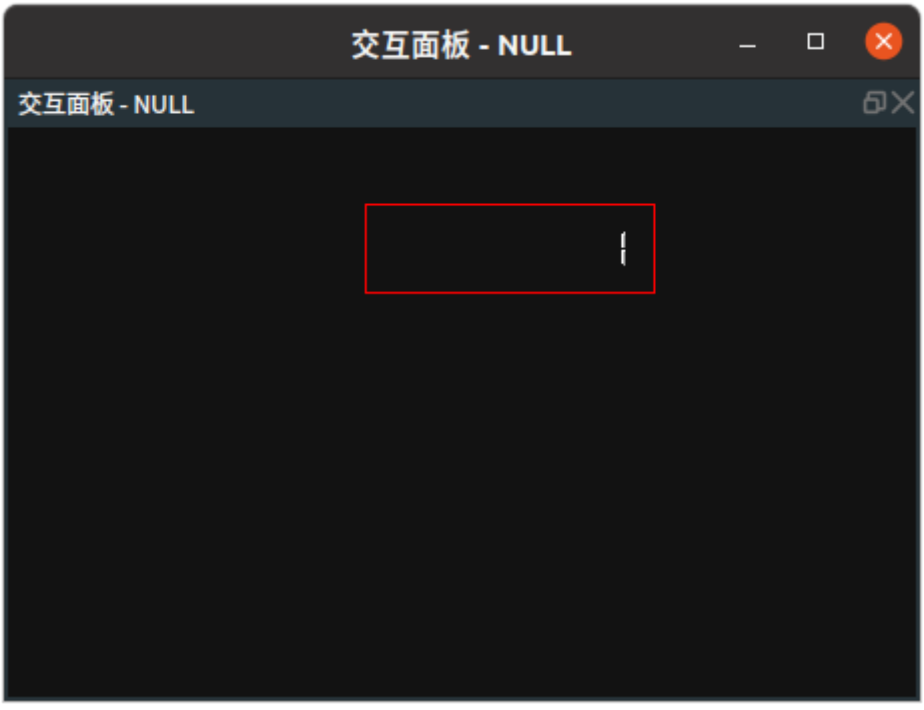
1. 将 selector 与交互面板选择输出工具—— “数码管” 进行绑定。
2. 点击 RVS 运行按钮，触发 Trigger 算子。

运行结果：

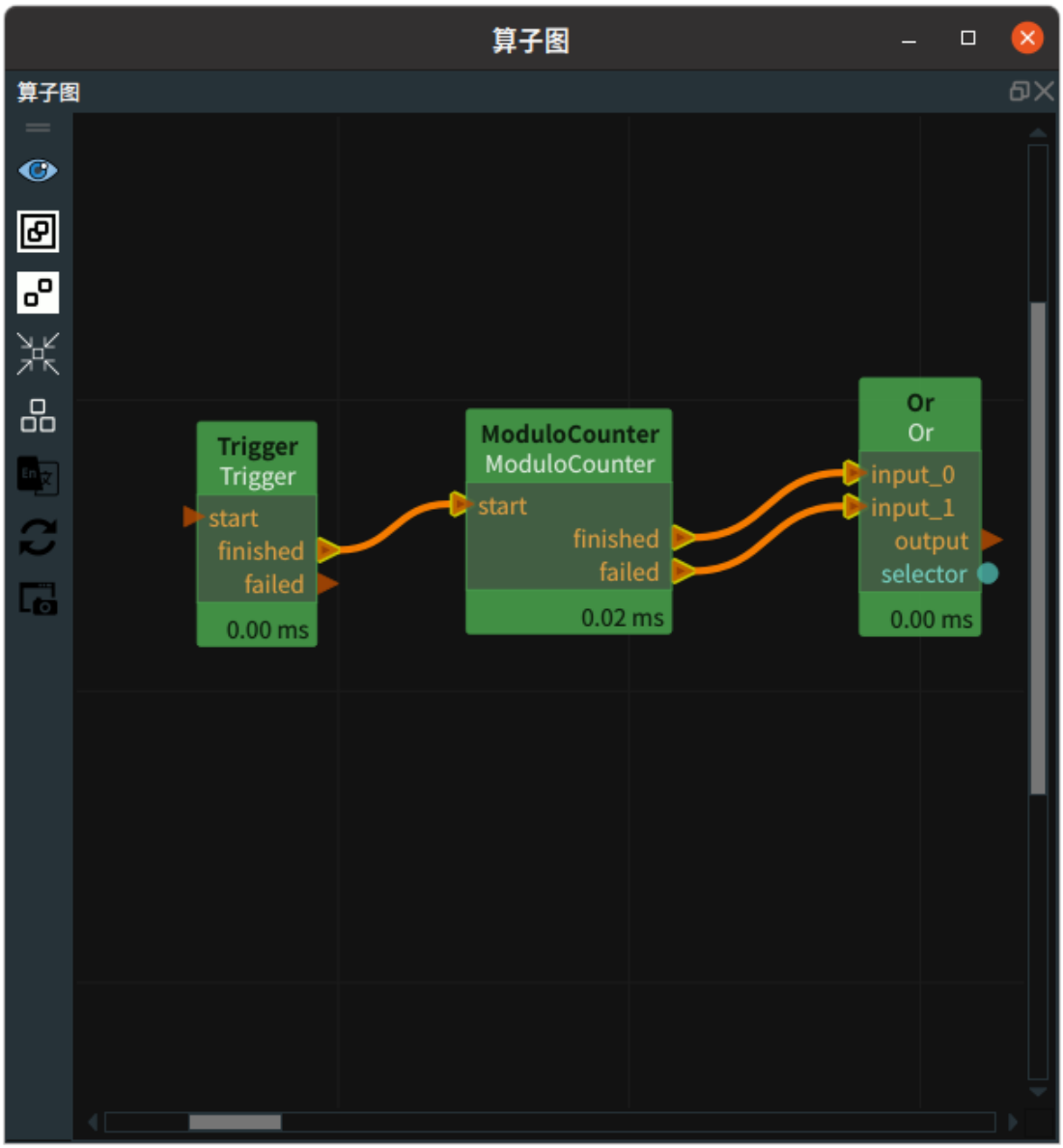
1. 如下图所示，当第一次触发 Trigger 时，ModuloCounter 算子触发 failed 信号。因此触发 Or 算子左侧 input_1 端口。右侧 selector 端口输出 “1”。



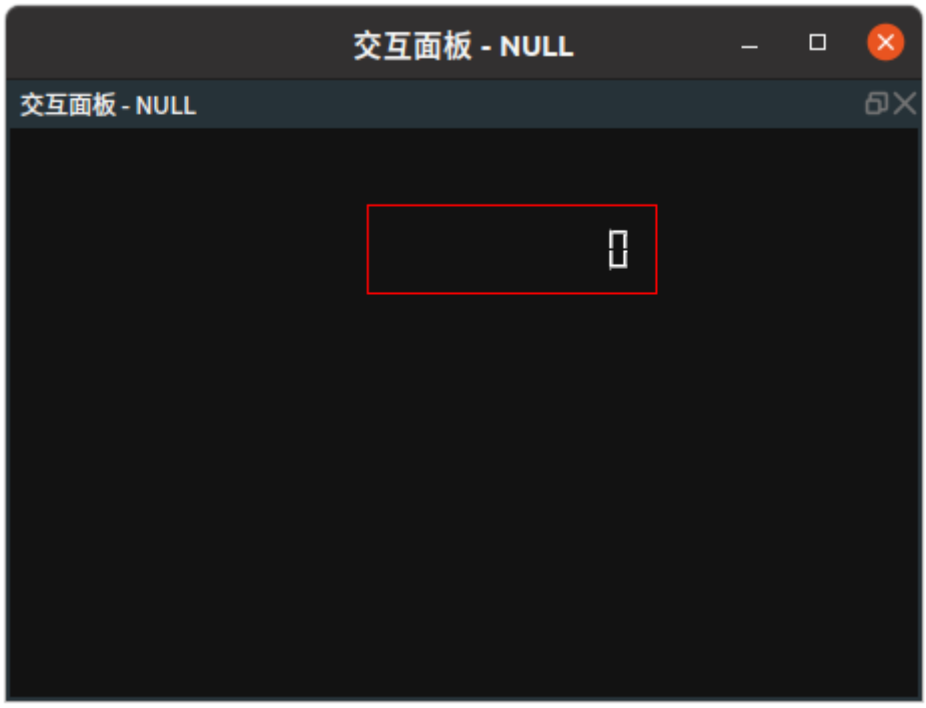
2. 此时交互面板中数码管显示的值为 1 。



3. 如下图所示，当第二次触发 Trigger 时，ModuloCounter 算子触发 finished 信号。因此触发 Or 算子左侧 input_0 端口。右侧 selector 端口输出 “0”。



4. 此时交互面板中数码管显示的值为 0 。


交互面板 - NULL
交互面板 - NULL

Emit 生成元素

Emit 算子为生成元素，用于生成单个 Angle 、CameraInfo 、Cube 、Cylinder 、Line 、Circle 、Object 、Pose 、Sphere 、String 、Text 、ImagePoints。

类型	功能
Angle	用于生成单个角。
CameraInfo	用于生成相机信息。
Cube	用于生成单个立方体。
Cylinder	用于生成单个圆柱体。
Line	用于生成单个线。
Circle	用于生成单个圆。
Object	用于生成单个物体。
Pose	用于生成单个 pose 。
Sphere	用于生成单个球体。
String	用于生成单个字符串。
Text	用于生成单个 3D 立体文字。
ImagePoints	用于生成关键点坐标（x , y ）。

Angle

将 Emit 算子的 **类型** 属性选择 Angle ，用于生成单个角。

算子参数

- **p1**：生成角度的 p1 点 pose 。
- **p2**：生成角度的 p2 点 pose 。
- **p3**：生成角度的 p3 点 pose 。
- **夹角/angle**：设置角度在 3D 视图中的可视化属性。
  -  打开角度可视化。
  -  关闭角度可视化。
  -  设置角度的颜色。取值范围：[-2,360] 。默认值：60 。
  -  设置角的线宽。取值范围：[1,100] 。默认值：4 。
  -  标注角显示。

数据信号输入输出

输入：

- `angle`：
  - 数据类型：Angle
  - 输入内容：angle 数据

输出：

- `angle`：
  - 数据类型：Angle
  - 输出内容：angle 数据

功能演示

使用 Emit 算子中 Angle ，生成单个空间角。

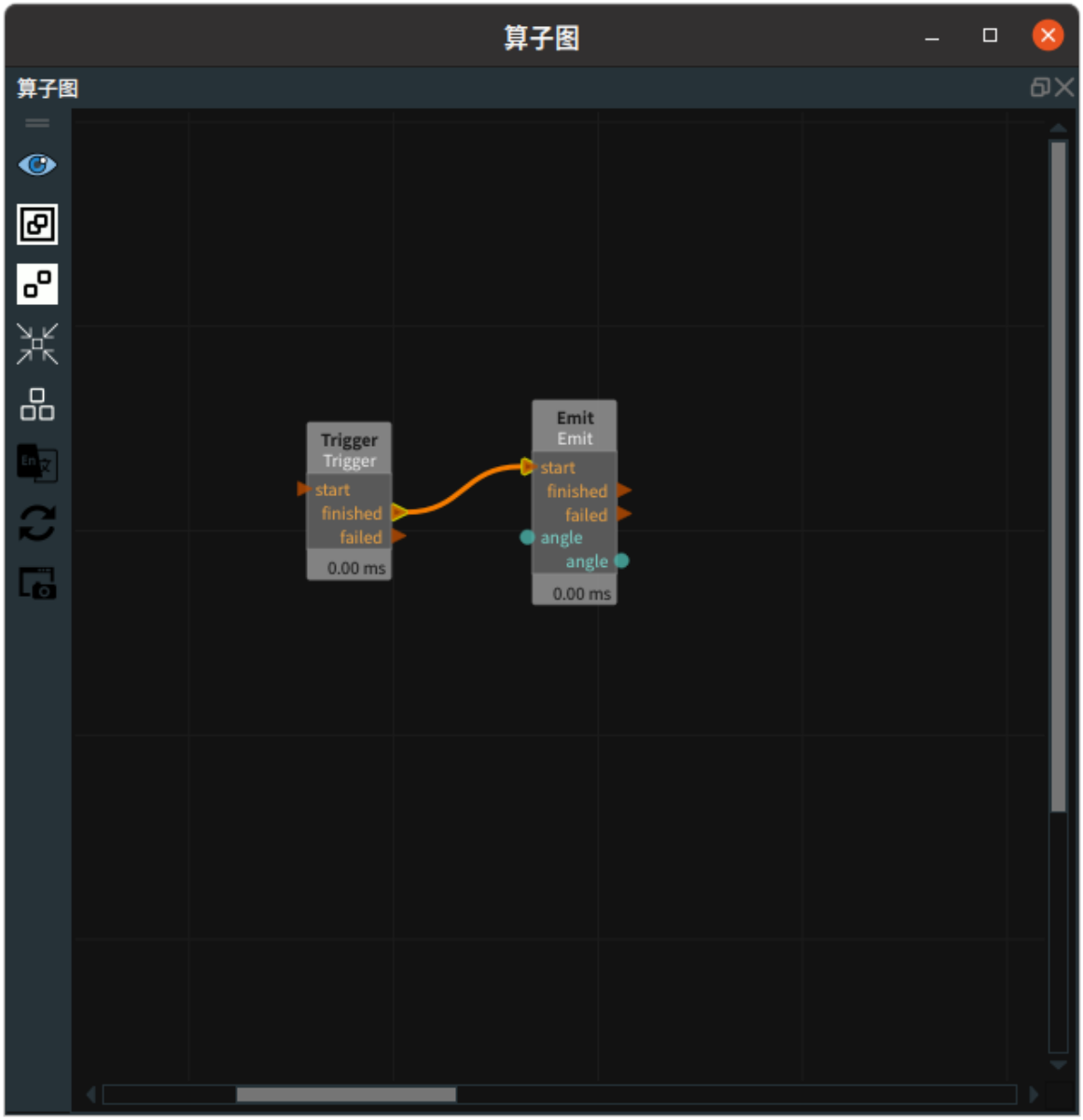
步骤1：算子准备

添加 Trigger 、Emit 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
   - 类型 → Angle
   - P1 → 0 0 0 0 0 0
   - P2 → 0 1 0 0 0 0
   - p3 → 0 0 1 0 0 0
   - 夹角 →  可视 → 打开 

步骤3：连接算子

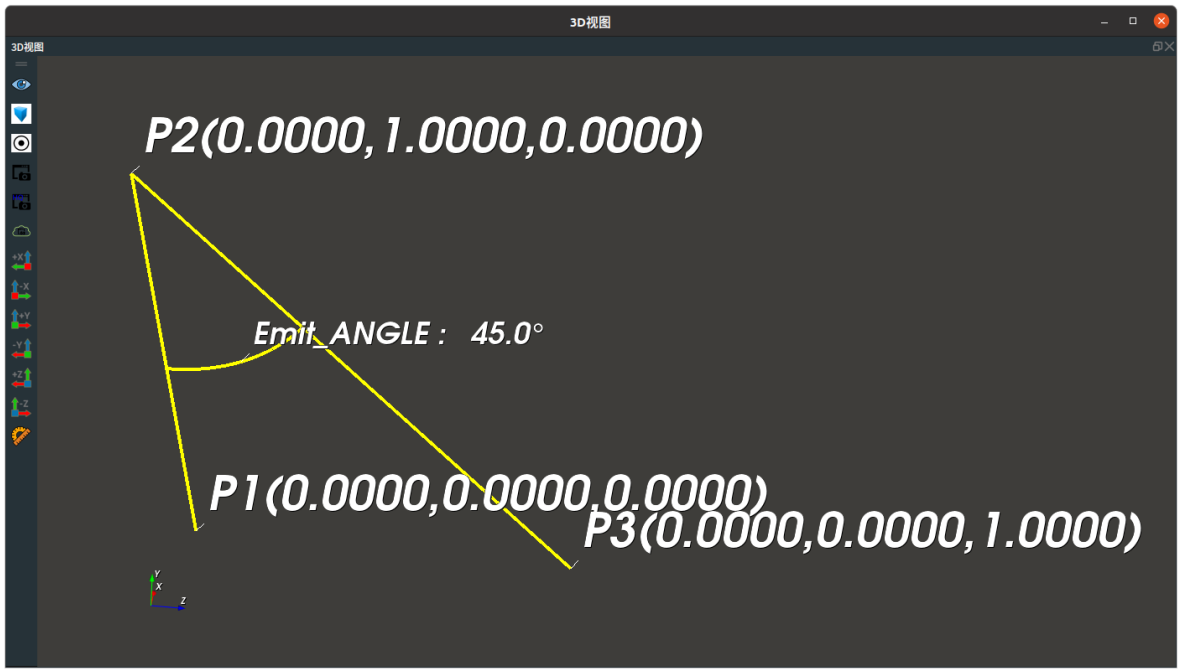


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，3D 视图中显示生成空间角的角度和三个点的标注。



Cube

将 Emit 算子的 **类型** 属性选择 Cube ，用于生成单个立方体。

算子参数

- **坐标/pose**： 立方体中心点 pose 的 x y z roll pitch yaw 。
- **宽度/width**：立方体的宽。pose 的 X 轴方向。
- **高度/height**：立方体的高。pose 的 Y 轴方向。
- **深度/depth**：立方体的长。pose 的 Z 轴方向。
- **立方体/cube**：设置立方体在 3D 视图中的可视化属性。
  - 打开立方体可视化。
  - 关闭立方体可视化。
  - 设置立方体的颜色。取值范围：[-2,360] 。默认值：-2 。
  - 设置立方体的透明度。取值范围：[0,1] 。默认值：0.5 。
- **坐标/pose**：设置 pose 在 3D 视图中的可视化属性。
  - 打开 pose 可视化。
  - 关闭 pose 可视化。
  - 设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

- **pose**：
  - 数据类型：Pose
  - 输入内容：pose 数据

输出：

- **cube**：

    - 数据类型：Cube
    - 输出内容：cube 数据
- **pose**：
    - 数据类型：Pose
    - 输出内容：pose 数据

功能演示

本节将使用 Emit 算子中 Cube ，生成单个立方体。这与 Emit 算子中 Angle 属性生成单个角的方法相同，请参照该章节的功能演示。

Cylinder

将 Emit 算子的 **类型** 属性选择 Cylinder ，用于生成单个圆柱体。

算子参数

- **坐标/pose**：圆柱体的中心点 pose 的 x y z roll pitch yaw 。
- **半径/radius**：圆柱体的圆的半径 。默认值：0.075。
- **长度/length**：圆柱体的长 。默认值：0.4。
- **圆柱体/cylinder**：设置圆柱体在3D视图中的可视化属性。
    -  打开圆柱体可视化。
    -  关闭圆柱体可视化。
    -  设置圆柱体的颜色。取值范围：[-2,360] 。默认值：-2 。
- **坐标/pose**：设置 pose 在 3D 视图中的可视化属性。
    -  打开 pose 可视化。
    -  关闭 pose 可视化。
    -  设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

- **pose**：
    - 数据类型：Pose
    - 输入内容：pose 数据

输出：

- **cylinder**：
    - 数据类型：Cylinder
    - 输出内容：cylinder 数据
- **pose**：
    - 数据类型：Pose
    - 输出内容：pose 数据

功能演示

本节将使用 Emit 算子中 Cylinder ，生成单个圆柱体。这与 Emit 算子中 Angle 属性生成单个角的方法相同，请参照该章节的功能演示。

Line

将 Emit 算子的 **类型** 属性选择 Line ，用于生成单条线。

算子参数

- **取反/inverse**：两个点取反。
  - **True**：line 数据输出端口结果：p2 <->p1。
  - **False**：line 数据输出端口结果：p1 <->p2。
- **p1**：生成线的 p1 点 pose 。
- **p2**：生成线的 p2 点 pose 。
- **线段/line**：设置线条在 3D 视图中的可视化属性。
  -  打开线条可视化。
  -  关闭线条可视化。
  -  设置线条的颜色。取值范围：[-2,360] 。默认值：60 。
  -  设置线条的线宽。取值范围：[1,100] 。默认值：1 。

数据信号输入输出

输入：

- **line**：
  - 数据类型：Line
  - 输入内容：line 数据（2个 pose 组成的空间线段）。
- **p1**：
  - 数据类型：Pose
  - 输入内容：pose 数据
- **p2**：
  - 数据类型：Pose
  - 输入内容：pose 数据

输出：

- **line**：
  - 数据类型：Line
  - 输出内容：line 数据

功能演示

本节将使用 Emit 算子中 Line ，生成单条线。这与 Emit 算子中 Angle 属性生成单个角的方法相同，请参照该章节的功能演示。

Circle

将 Emit 算子的 **类型** 属性选择 Circle ，用于生成单个圆。

算子参数

- **坐标/pose**：圆的中心点 pose 的 x y z roll pitch yaw 。
- **半径/radius**：圆的半径 。
- **圆圈/circle**：设置圆在 3D 视图中的可视化属性。
  -  打开圆可视化。
  -  关闭圆可视化。
  -  设置圆的颜色。取值范围：[-2,360] 。默认值：-2 。

数据信号输入输出

输入：

- **circle**：
  - 数据类型：Circle
  - 输入内容：circle 数据
- **pose**：
  - 数据类型：Pose
  - 输入内容：pose 数据

输出：

- **circle**：
  - 数据类型：circle
  - 输出内容：circle 数据

功能演示

本节将使用 Emit 算子中 Circle ，生成单个圆。这与 Emit 算子中 Angle 属性生成单个角的方法相同，请参照该章节的功能演示。

Pose

将 Emit 算子的 **类型** 属性选择 Pose ，用于生成单个 pose 。

算子参数

- **坐标/pose**： pose 的 x y z roll pitch yaw 。
- **坐标/pose**：设置 pose 在 3D 视图中的可视化属性。
  -  打开 pose 可视化。
  -  关闭 pose 可视化。
  -  设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。
- **倒置坐标/inverse_pose**：设置逆 pose 在 3D 视图中的可视化属性。参数值描述 与 pose 一致。

数据信号输入输出

输入：

- `pose`：
  - 数据类型：Pose
  - 输入内容：pose 数据

输出：

- `pose`：
  - 数据类型：Pose
  - 输出内容：pose 数据
- `inverse_pose`：
  - 数据类型：Pose
  - 输出内容：逆 pose 数据

功能演示

本节将使用 Emit 算子中 Pose ，生成单个 pose 。这与 Emit 算子中 Angle 属性生成单个角的方法相同，请参照该章节的功能演示。

Sphere

将 Emit 算子的 `类型` 属性选择 Sphere ，用于生成单个球体。

算子参数

- `坐标/pose`：生成球体的中心点 pose 的 x y z roll pitch yaw 。
- `半径/radius`：生成球体的半径。
- `球体/sphere`：设置球体在 3D 视图中的可视化属性。
  -  打开球体可视化。
  -  关闭球体可视化。
  -  设置球体的颜色。取值范围：[-2,360] 。默认值：-2 。
  -  设置球体的透明度。取值范围：[0,1] 。默认值：0.8 。
- `坐标/pose`：设置 pose 在 3D 视图中的可视化属性。
  -  打开 pose 可视化。
  -  关闭 pose 可视化。
  -  设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。

数据信号输入输出

输入：

- `pose`：
  - 数据类型：Pose
  - 输入内容：pose 数据

输出：

- `sphere`：
  - 数据类型：Sphere
  - 输出内容：sphere 数据
- `pose`：
  - 数据类型：Pose
  - 输出内容：pose 数据

功能演示

本节将使用 Emit 算子的 type 属性选择 Sphere ，生成单个球体。这与 Emit 算子中 Angle 属性生成单个角的方法相同，请参照该章节的功能演示。

String

将 Emit 算子的 `类型` 属性选择 String ，用于生成字符串。

算子参数

- `字符串/string`：字符串内容。
- `文件/filename`：文件名。
- `目录/directory`：文件目录名。
- `字符串/string`：设置字符串曝光属性。打开后可与交互面板中输出工具——“文本框”控件进行绑定。
  -  打开曝光。
  -  关闭曝光。
- `文件/filename`：设置文件名曝光属性。打开后可与交互面板中输出工具——“文本框”控件进行绑定。参数值描述与 string 一致。
- `目录/directory`：设置文件目录名曝光属性。打开后可与交互面板中输出工具——“文本框”控件进行绑定。参数值描述与 string 一致。

数据信号输入输出

输入：

- `string`：
  - 数据类型：String
  - 输入内容：String 数据

输出：

- `stirng`：
  - 数据类型：String
  - 输出内容：string 数据
- `filename`：
  - 数据类型：String
  - 输出内容：string 数据
- `directory`：
  - 数据类型：String
  - 输出内容：string 数据

功能演示

使用 Emit 算子中 string ，输入string、filename、directory，分别与交互面板中“文本框”控件进行绑定。

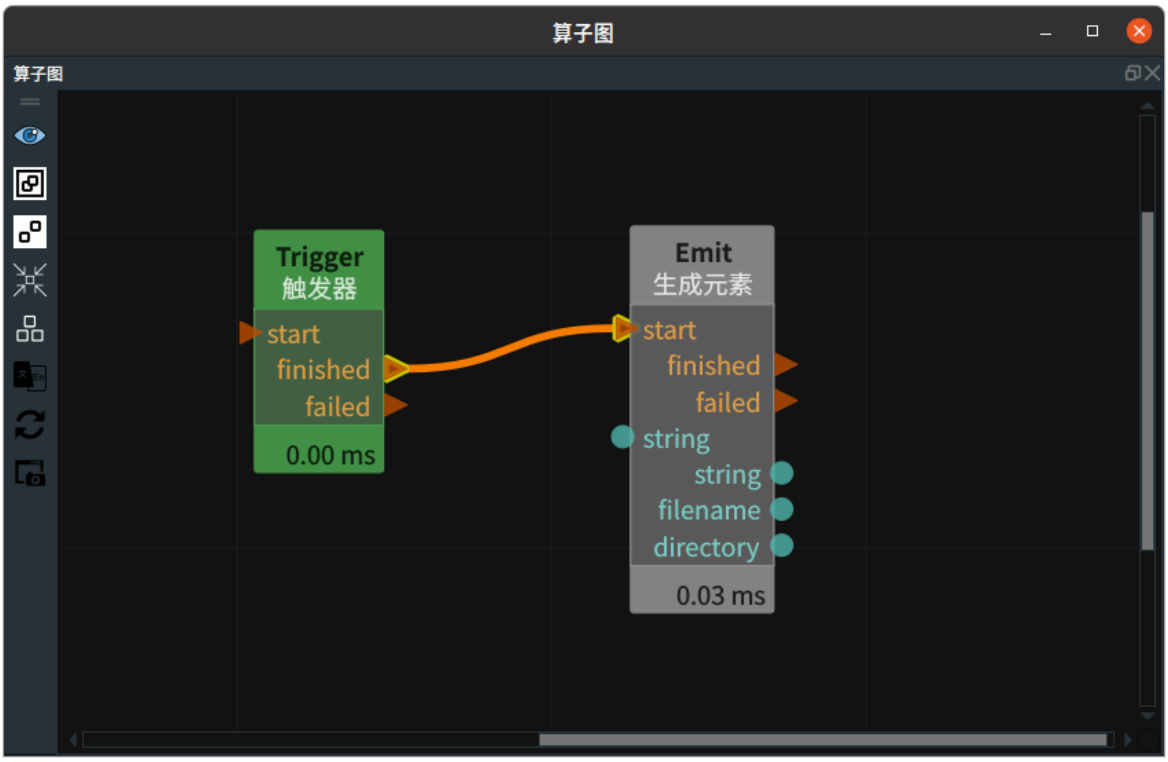
步骤1：算子准备

添加 Trigger 、Emit 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
    - 类型 → String
    - 字符串 → RVS
    - 文件 → ••• → 选择文件名 （ *example_data/cube/cube.txt* ）
    - 目录 → 选择文件目录名 （ *example_data/cube* ）
    - 字符串 → 曝光
    - 文件 → 曝光
    - 目录 → 曝光

步骤3：连接算子

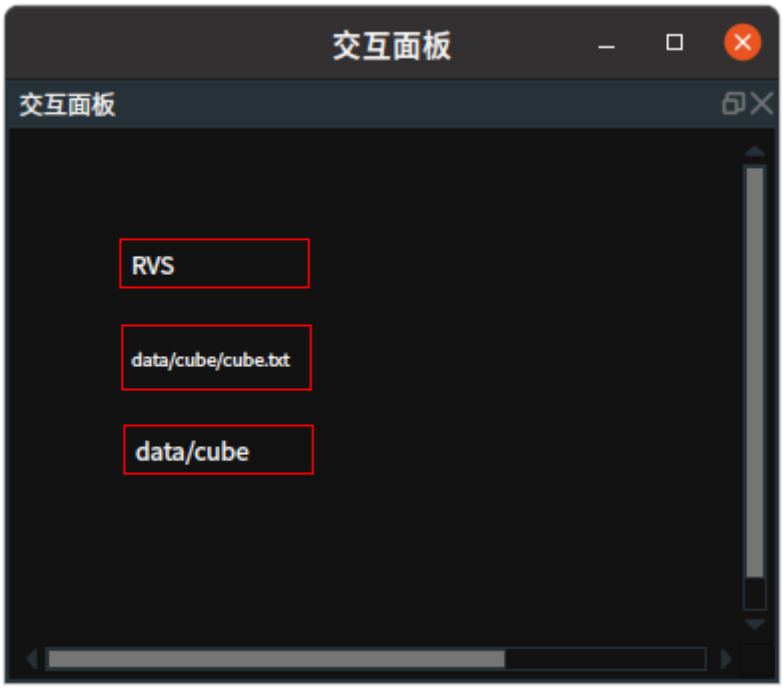


步骤4：运行

1. 鼠标右键交互面板空白处 → 解锁，拖出输出工具中三个——“文本框”控件。
2. 鼠标右键交互面板空白出 → 点击锁定，鼠标中键点击拖出的“文本框”，依次将三个曝光属性进行绑定。
3. 点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

结果如下图所示， 交互面板中分别显示算子参数 string、filename、directory 中的内容。



Text

将 Emit 算子的 **类型** 属性选择 Text ，用于生成单个 3D 立体文字。

算子参数

- **文本/text**：生成 3D 立体文字的内容。
- **坐标/pose**： text 的 pose 的 x y z roll pitch yaw 。
- **文本/text**：设置文字在 3D 视图中的可视化属性。
  - 打开文字可视化。
  - 关闭文字可视化。
  - 设置文字的颜色。取值范围：[-2,360]。默认值：-1 。
  - 设置文字的尺寸大小。取值范围：[0,99.99] 。默认值：0.05 。

数据信号输入输出

输入：

- **string**：
  - 数据类型：String
  - 输入内容：String 数据
- **pose**：
  - 数据类型：Pose
  - 输入内容：pose 数据

输出：

- **text**：
  - 数据类型：Text
  - 输出内容：text 数据

功能演示

本节将使用 Emit 算子中 Text ，生成单个 3D 立体文字。这与 Emit 算子中 Angle 属性生成单个角的方法相同，请参照该章节的功能演示。

1. 使用 Emit 算子中：类型 → Text。
2. 设置 Emit 算子参数：文本 → RVS
3. 算子运行成功后，在 3D 视图中显示 text——“RVS” 。

ImagePoints

将 Emit 算子的 **类型** 属性选择 ImagePoints ，用于生成图像关键点坐标（ x , y ）。

算子参数

- **图像点/points**：生成的图像关键点坐标，可以填单组或多组图像关键点坐标。

注意：这里数值需要填写偶数个。输入内容：x1 y1 x2 y2.....

数据信号输入输出

输入：

- **string**：
  - 数据类型：string
  - 输入内容：string 数据

输出：

- **image_points**：
  - 数据类型：ImagePoints
  - 输出内容：图像关键点数据

功能演示

使用 Emit 算子中 ImagePoints ，生成单个 2D 数据关键点列表。

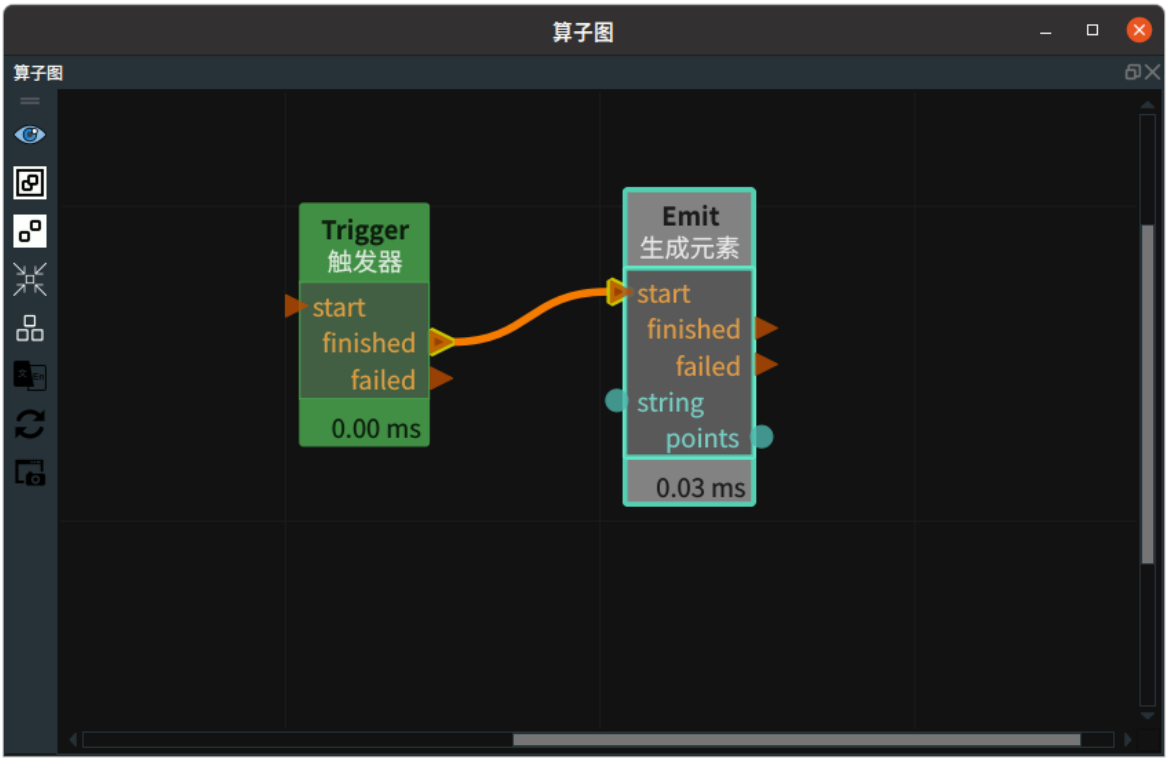
步骤1：算子准备

添加 Trigger 、Emit 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
    - 类型 → ImagePoints
    - 图像点 → 1 2 3 4

步骤3：连接算子

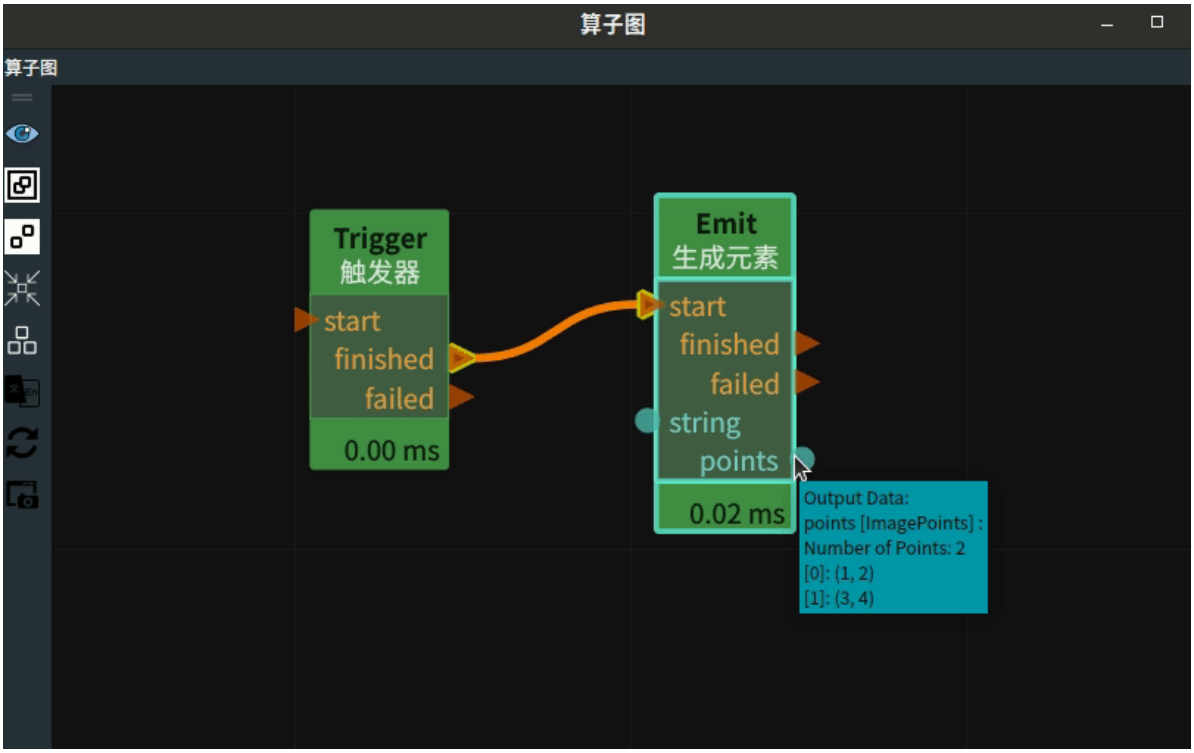


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

结果如下图所示，Emit 算子右侧 points 输出端口输出两组关键点坐标。


算子图
算子图
Trigger
触发器
start
finished
failed
0.00 ms
Emit
生成元素
start
finished
failed
string
points
0.02 ms
Output Data:
points [ImagePoints] :
Number of Points: 2
[0]: (1, 2)
[1]: (3, 4)

box_unstack

PackageMeasure 测量包裹

PackageMeasure 算子用于包裹测量算法，算子功能是根据保存的背景文件，测量当前图像中的包裹。

算子参数

- **文件路径/file_path**：保存背景数据的路径。
- **文件名后缀/file_suffix**：保存背景数据的文件后缀。
- **m_roi_x**：ROI 包裹测量区域的左顶点图像列坐标。
- **m_roi_y**：ROI 包裹测量区域的左顶点图像行坐标。
- **m_roi_w**：ROI 包裹测量区域的图像像素宽度。
- **m_roi_h**：ROI 包裹测量区域的图像像素高度。
- **onlySafeAreaOnDepth**：只测量完全在深度图 ROI 区域内的包裹。
- **measureTotal**：
  - True：测量所有的包裹。
  - False：随机测量某个包裹。
- **leastHeight**：深度图与背景平面的最小高度差，高度差小于该值则当做深度图数据误差和波动。默认值：20。单位：mm。
- **minArea**：面积小于该值的包裹忽略不计。默认值：200。单位：mm^2。
- **useRectBounding**：使用不旋转的包围框框出包裹在图像中的区
- **multiObj**：
  - True：进行多包裹测量，识别出多个不同目标。
  - False：将所有包裹当成一个包裹。
- **onlyAllInSafeArea**：只测量完全在 ROI 区域内的包裹，通常与 onlySafeAreaOnDepth 参数保持一致。
- **深度图像/depth_image**：设置 ROI 区域在 2D 视图中的可视化属性。
  -  打开 ROI 区域可视化。
  -  关闭 ROI 区域可视化。
- **彩色图像/color_image**：设置相机的彩色图在 2D 视图中的可视化属性。
  -  打开相机的彩色图可视化。
  -  关闭相机的彩色图可视化。
- **package_result_list**：将箱子的长、宽、高、箱子的边在图像中角度、中心点像素坐标 x 、中心点像素坐标 y 曝光。单位：mm。与箱子的积分体积曝光。单位：mm^3。用于与交互面板中输出工具“表格”进行绑定。
  -  打开曝光。
  -  关闭曝光。

数据信号输入输出

输入：

- `depth`：
  - 数据类型：Image
  - 输入内容：相机的深度图
- `color`：
  - 数据类型：Image
  - 输入内容：相机的彩色图
- `depth_calib`：
  - 数据类型：CalibInfo
  - 输入内容：相机的深度图标定数据
- `color_calib`：
  - 数据类型：CalibInfo
  - 输入内容：相机的彩色图标定数据

输出：

- `output_depth`：
  - 数据类型：Image
  - 输出内容：相机的深度图转换为 RGB 显示，并显示 ROI 区域
- `output_color`：
  - 数据类型：Image
  - 输出内容：相机的彩色图
- `package_result_list`：
  - 数据类型：String
  - 输出内容：箱子的长、宽、高、箱子的边在图像中角度、中心点像素坐标x、中心点像素坐标y、积分体积

功能演示

使用 PackageMeasure 建立背景，用于包裹测量。

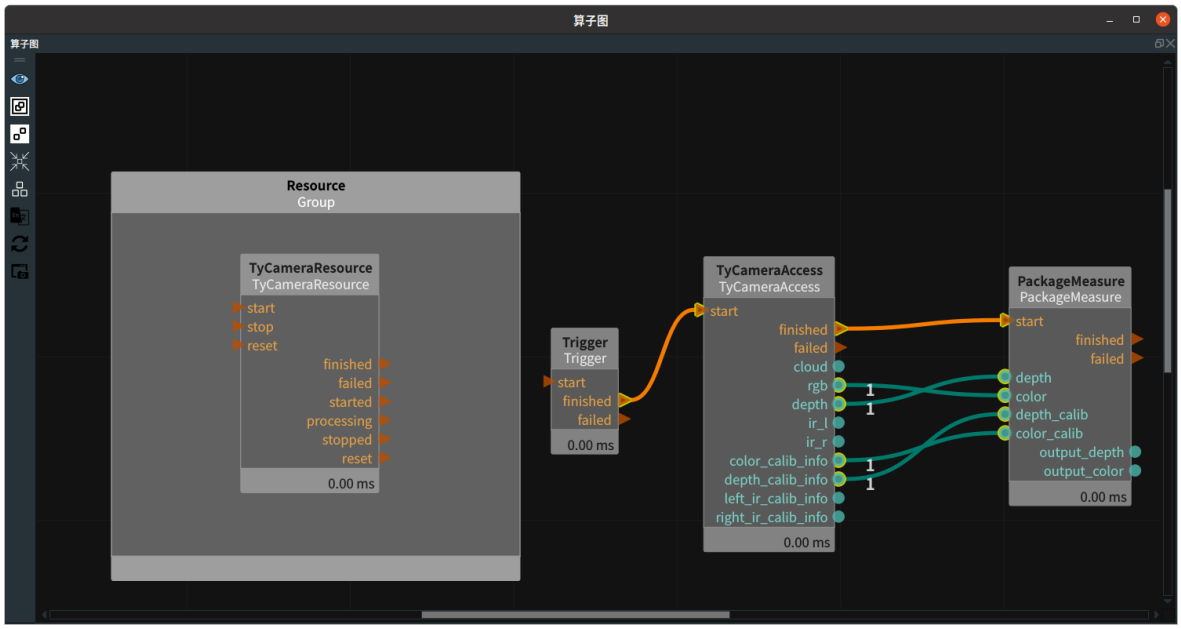
步骤1：算子准备

1. 添加 TyCameraResource 至Resource Group。
2. 添加TyCameraAccesss 、Trigger 、PackageMeasure 算子至算子图。

步骤2：设置算子参数

1. 设置 PackageMeasure 算子参数：
   - 文件路径 → ./bg
   - m_roi_x→350
   - m_roi_y→250
   - m_roi_w→600
   - m_roi_h→500
   - 深度图像 →  可视
   - 彩色图像 →  可视

步骤3：连接算子

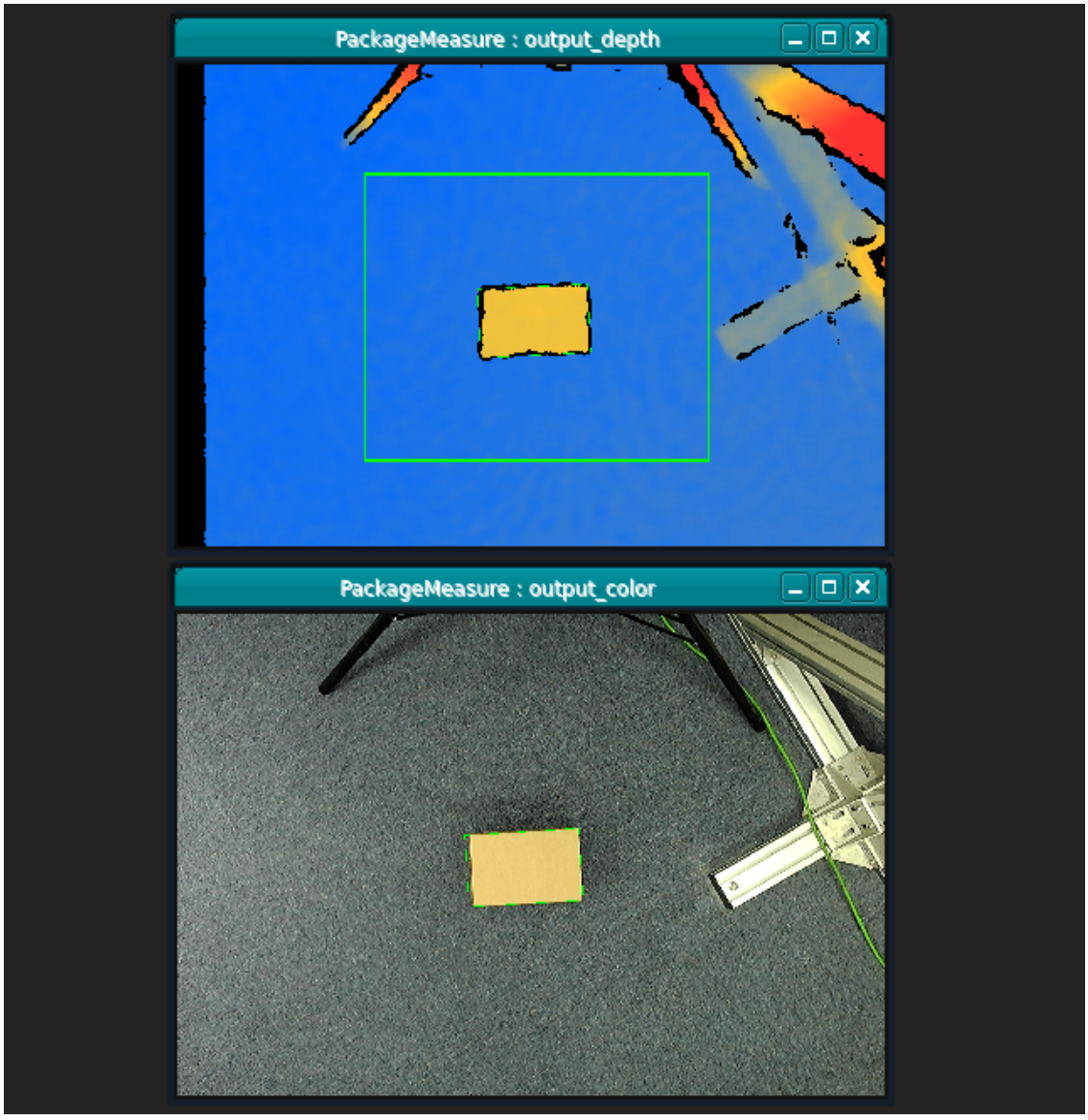


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子

运行结果

1. 打开 PackageMeasure 算子的 `深度图像` 和 `彩色图像` 可视化属性值，2D 视图显示如下：



2. 运行成功后，会在显示的深度图和彩色图上框出包裹的位置区域，同时在日志视图中打印出包裹的尺寸和位置信息：

2023-02-08 17:04:37.190647	rvs_box_unstack	warning	Num of obj: 1
2023-02-08 17:04:37.213835	rvs_box_unstack	info	Obj 0 width: 220.906
2023-02-08 17:04:37.213993	rvs_box_unstack	info	Obj 0 height: 139.068
2023-02-08 17:04:37.214010	rvs_box_unstack	info	Obj 0 depth: 116.469
2023-02-08 17:04:37.214022	rvs_box_unstack	info	Obj 0 angle: -2.96094
2023-02-08 17:04:37.214032	rvs_box_unstack	info	Obj 0 centerX: 647.215
2023-02-08 17:04:37.214042	rvs_box_unstack	info	Obj 0 centerY: 513.816
2023-02-08 17:04:37.214054	rvs_box_unstack	info	Obj 0 volume: 3.21317e+06

FilterBoxList 筛选箱子

FilterBoxList 算子用于按照一定的需求来筛选箱子点云列表 cloudlist 。

type	功能
ByAxisZ	按照箱子点云列表 cloudlist 的 Z 轴坐标值进行筛选，筛选出最上层箱子，并对最上层箱子进行排序。
ByArea	按照箱子点云列表 cloudlist 的抓取区域进行筛选，筛选出最先需要抓取的箱子及其附近某范围内的箱子。

ByAxisZ

算子参数

- **选择模式/select_mode**：选择筛选的模式。
  - Z_MAX：在机器人坐标系下最高的箱子。
  - Z_MIN：在相机坐标系下最高的箱子。
- **选择阈值/select_thresh**：选择筛选的阈值。如果某个箱子的点云中心 Z 值与最高的箱子的点云中心 Z 值的差值小于阈值，则判定为最上层箱子。
- **x权重/weight_x**：筛选出最上层箱子列表后，对其按照 X 值和 Y 值进行排序，weight_x 为 X 值的权重。
- **y权重/weight_y**：筛选出最上层箱子列表后，对其按照 X 值和 Y 值进行排序，weight_y 为 Y 值的权重。
- **点云列表/cloud_list**：设置筛选后的点云列表在 3D 视图中的可视化属性。
  -  打开筛选后的点云列表可视化。
  -  关闭筛选后的点云列表可视化。
  -  设置筛选后的点云列表的颜色。取值范围：[-2,360] 。默认值：-1 。
  -  设置筛选后的点云列表的尺寸。取值范围：[1,50] 。默认值：1 。

数据信号输入输出

输入：

- **cloudlist**：
  - 数据类型：PointCloudList
  - 输入内容：筛选前的点云列表

输出：

- **cloudlist**：
  - 数据类型：PointCloudList
  - 输出内容：筛选后的点云列表

功能演示

使用 FilterBoxList 算子中 ByAxisZ 对 cloudlist 进行筛选。

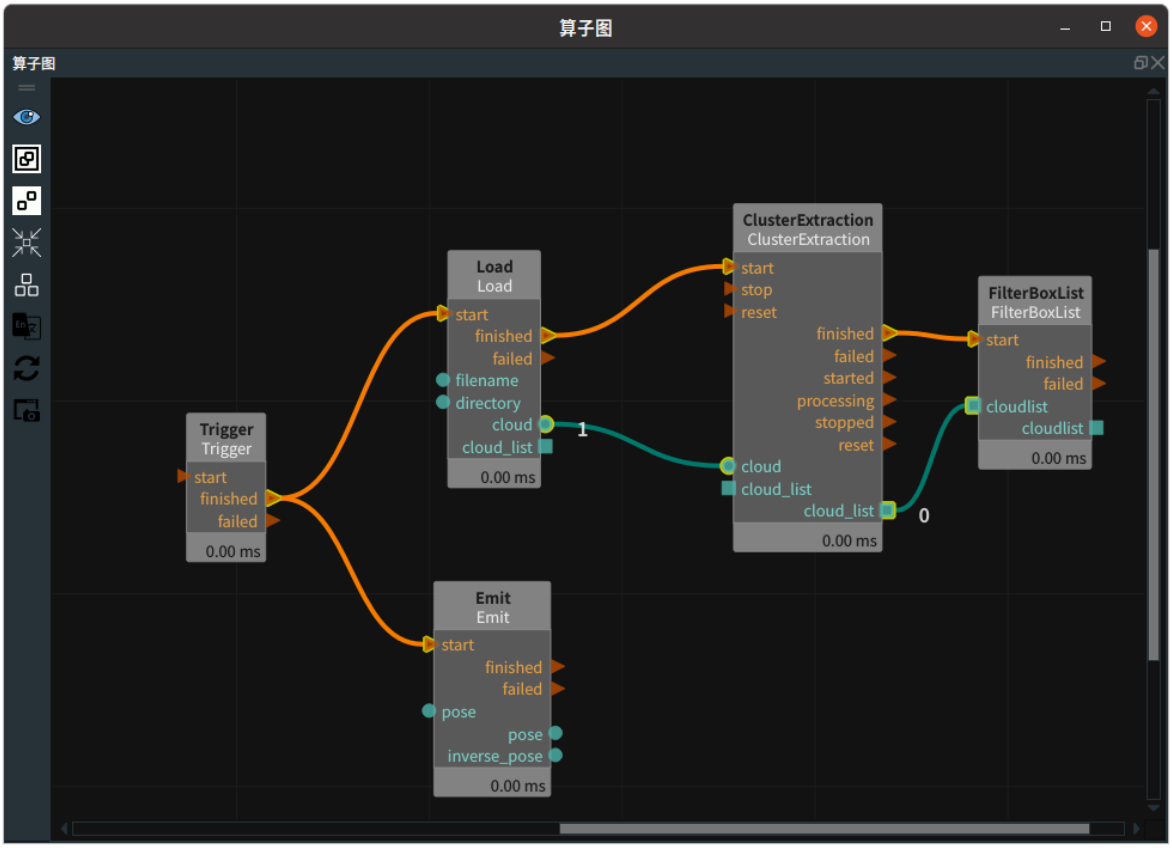
步骤1：算子准备

添加 Trigger 、Load 、Emit 、ClusterExtraction 、FilterBoxList 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 类型 → pointcloud
    - 文件 → ••• → 选择点云文件名 (*example_data/pointcloud/box_stack.pcd*)
2. 设置 Emit 算子参数：
    - 类型 → pose
    - 坐标 → 生成相机坐标系的 Pose (0 0 0 0 0 0) 。
    - 坐标 →  可视
3. 设置 ClusterExtraction 算子参数：
    - 最小点数 → 10000
    - 最大点数 → 100000
    - 公差值 → 0.005
    - 点云列表 →  可视
4. 设置 FilterBoxList 算子参数：
    - 类型→ ByAxisZ
    - 选择模式→ Z_MIN
    - 选择阈值→ 0.1
    - x权重 → 1
    - y权重→ 1
    - 点云列表 →  可视

步骤3：连接算子

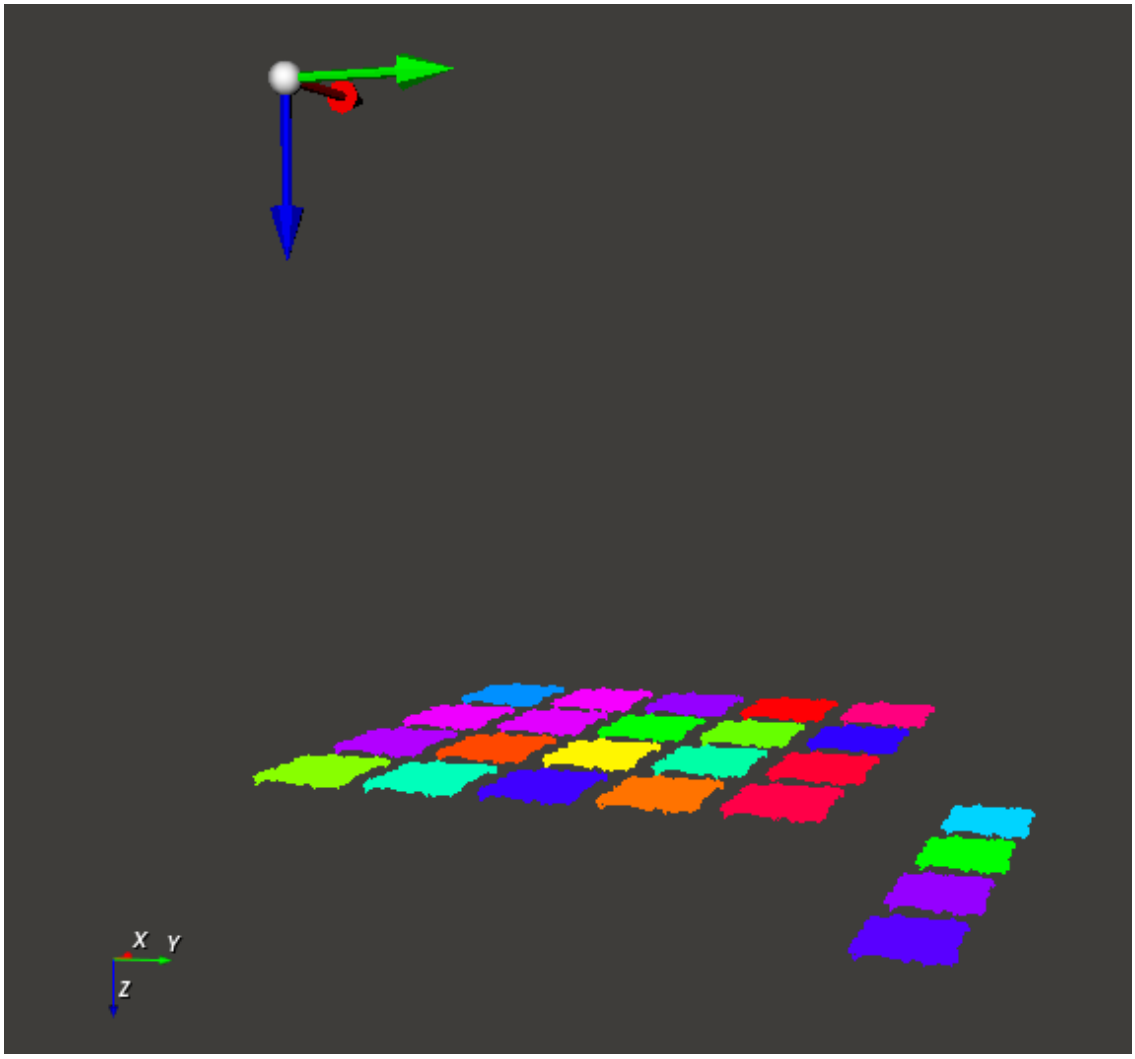


步骤4：运行

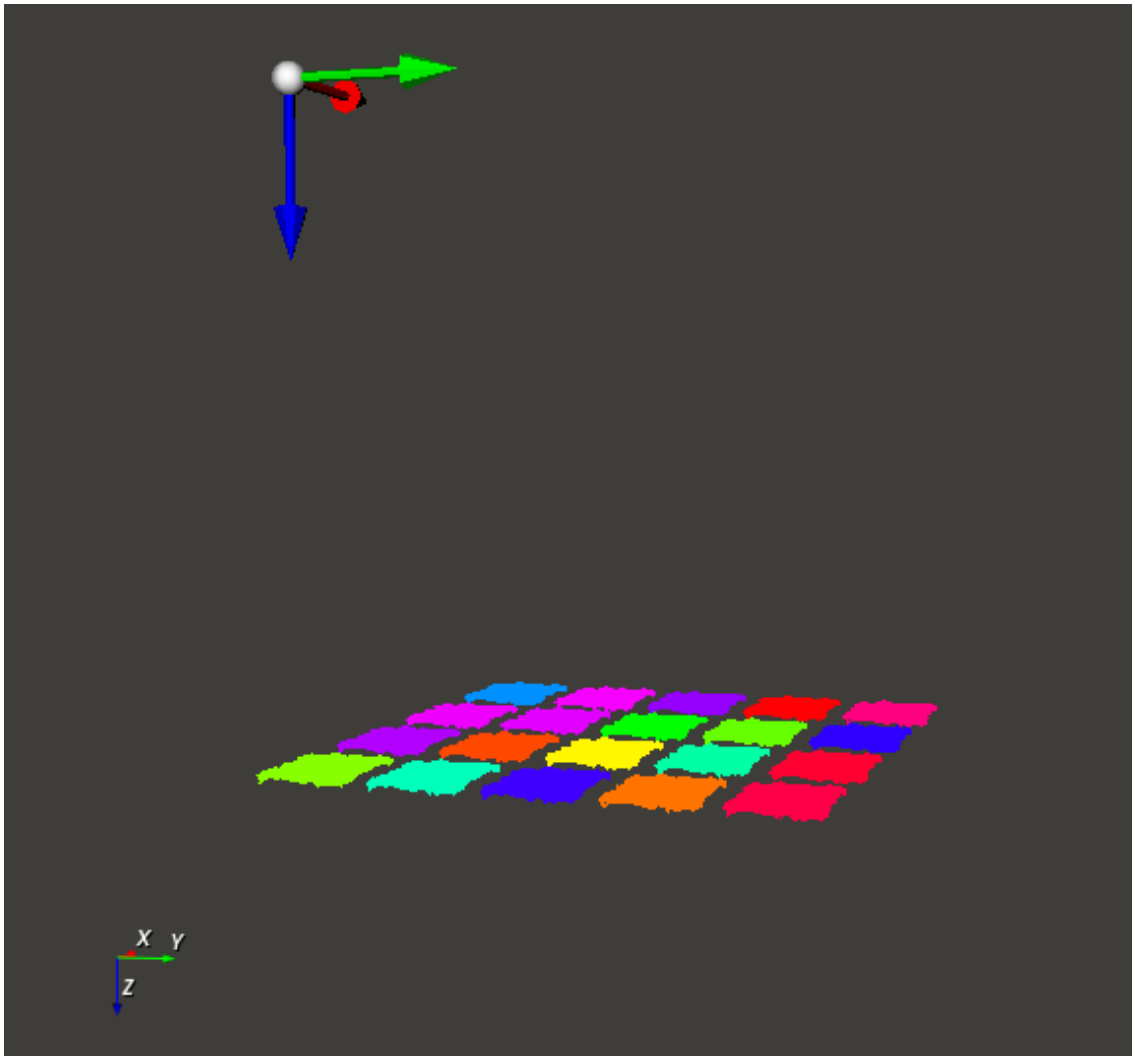
点击 RVS 运行按钮，触发 Trigger 算子

运行结果

1. 打开 ClusterExtraction 算子的 **点云列表** 可视化属性值，3D视图显示如下。



2. 打开 FilterBoxList 算子的**点云列表**可视化属性值，筛选出点云列表的最上层的箱子，去掉图像右侧的四个下层箱子。3D 视图显示如下。



ByArea

算子参数

- **x权重/weight_x**：按照 X 值和 Y 值选取最先抓取的箱子，x权重为 X 值的权重。
- **y权重/weight_y**：按照 X 值和 Y 值选取最先抓取的箱子，y权重为 Y 值的权重。
- **选择半径/select_radius**：根据最先抓取的箱子，筛选出与最先抓取的箱子中心点在 OXY 平面上的距离小于 select_radius 的箱子。
- **点云列表/cloud_list**：设置筛选后的点云列表在 3D 视图中的可视化属性。
  - 打开筛选后的点云列表可视化。
  - 关闭筛选后的点云列表可视化。
  - 设置筛选后的点云列表的颜色。取值范围：[-2,360] 。默认值：-1 。
  - 设置筛选后的点云列表的尺寸。取值范围：[1,50] 。默认值：1 。

数据信号输入输出

输入：

- **cloudlist**：
  - 数据类型：PointCloudList
  - 输入内容：筛选前的点云列表

输出：

- `cloudlist`：
  - 数据类型：PointCloudList
  - 输出内容：筛选后的点云列表

功能演示

使用 FilterBoxList 中 ByArea 对 cloudlist 进行筛选。

步骤1：算子准备

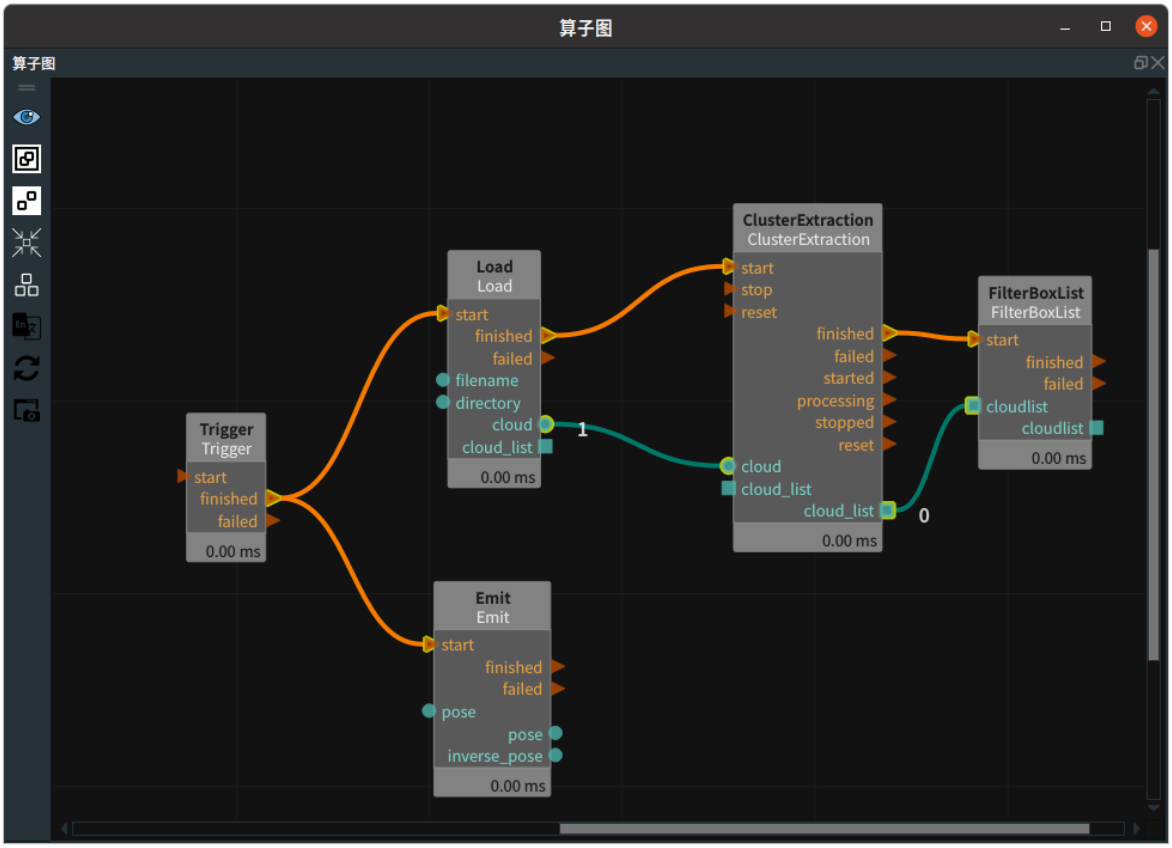
添加 Trigger 、Load 、Emit 、ClusterExtraction 、FilterBoxList 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → pointcloud
   - 文件 → ••• → 选择点云文件名 (*example_data/pointcloud/box_stack.pcd*)
   - 点云 → 可视
2. 设置 Emit算子参数：
   - 类型 →Pose
   - 坐标 →生成相机坐标系的Pose （0 0 0 0 0 0 ）。
   - 坐标 → 可视
3. 设置 ClusterExtraction 算子参数：
   - 最小点数 → 10000
   - 最大点数 → 100000
   - 公差值 → 0.005
   - 点云列表 → 可视
4. 设置 FilterBoxList 算子参数：
   - 类型 → ByArea
   - x权重 → 1
   - y权重→ 1
   - 选择半径 → 0.5
   - 点云列表 → 可视

   说明：按照 **x权重** 与 **y权重** 计算出最先抓取的箱子为左下角的箱子（ x权重 * x + y权重 * y 从小到大排序），然后筛选出左下角箱子及其周边 **选择半径**范围内的箱子，这样的筛选结果便于开展多个箱子的组合抓取工作。

步骤3：连接算子

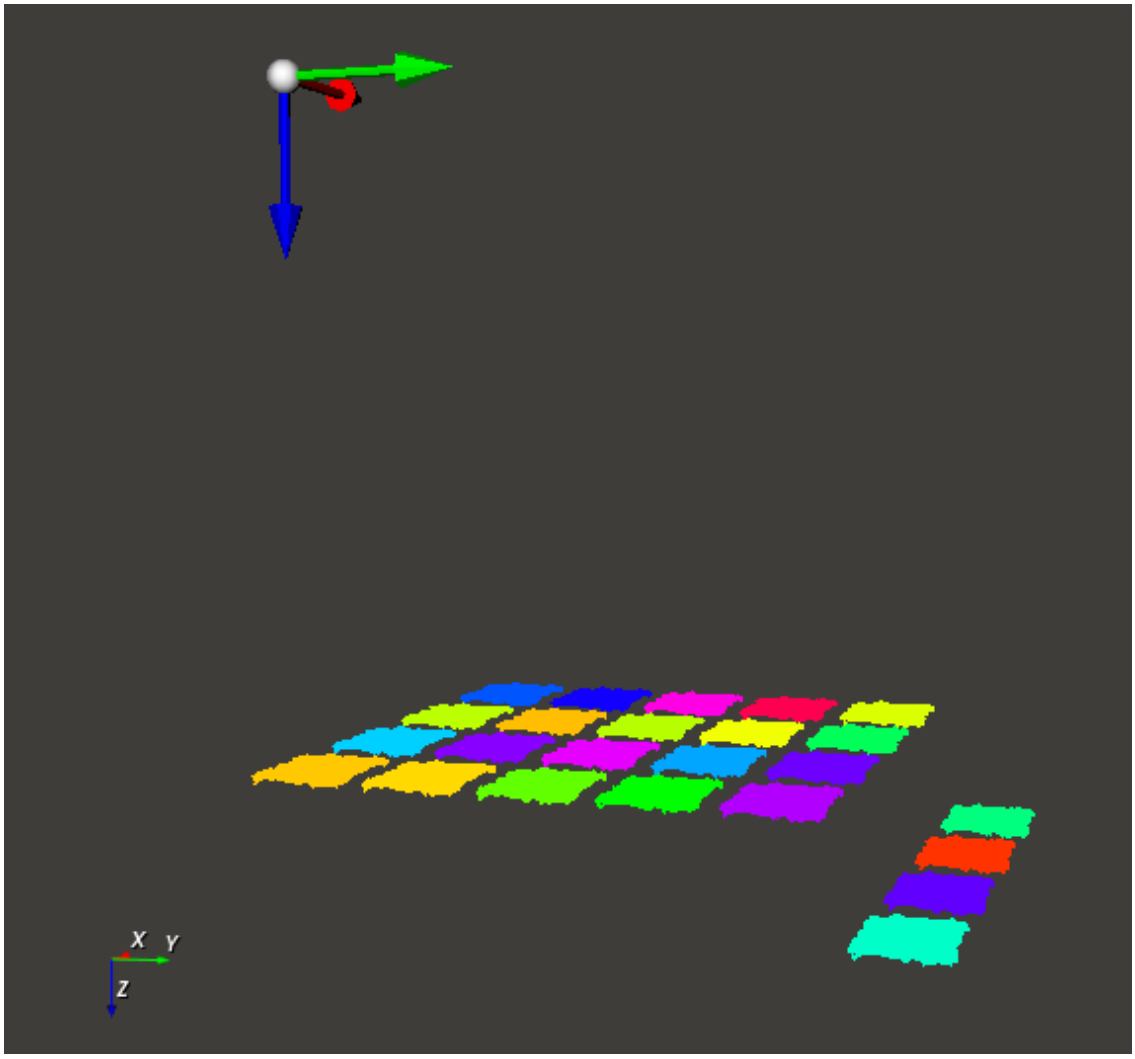


步骤4：运行

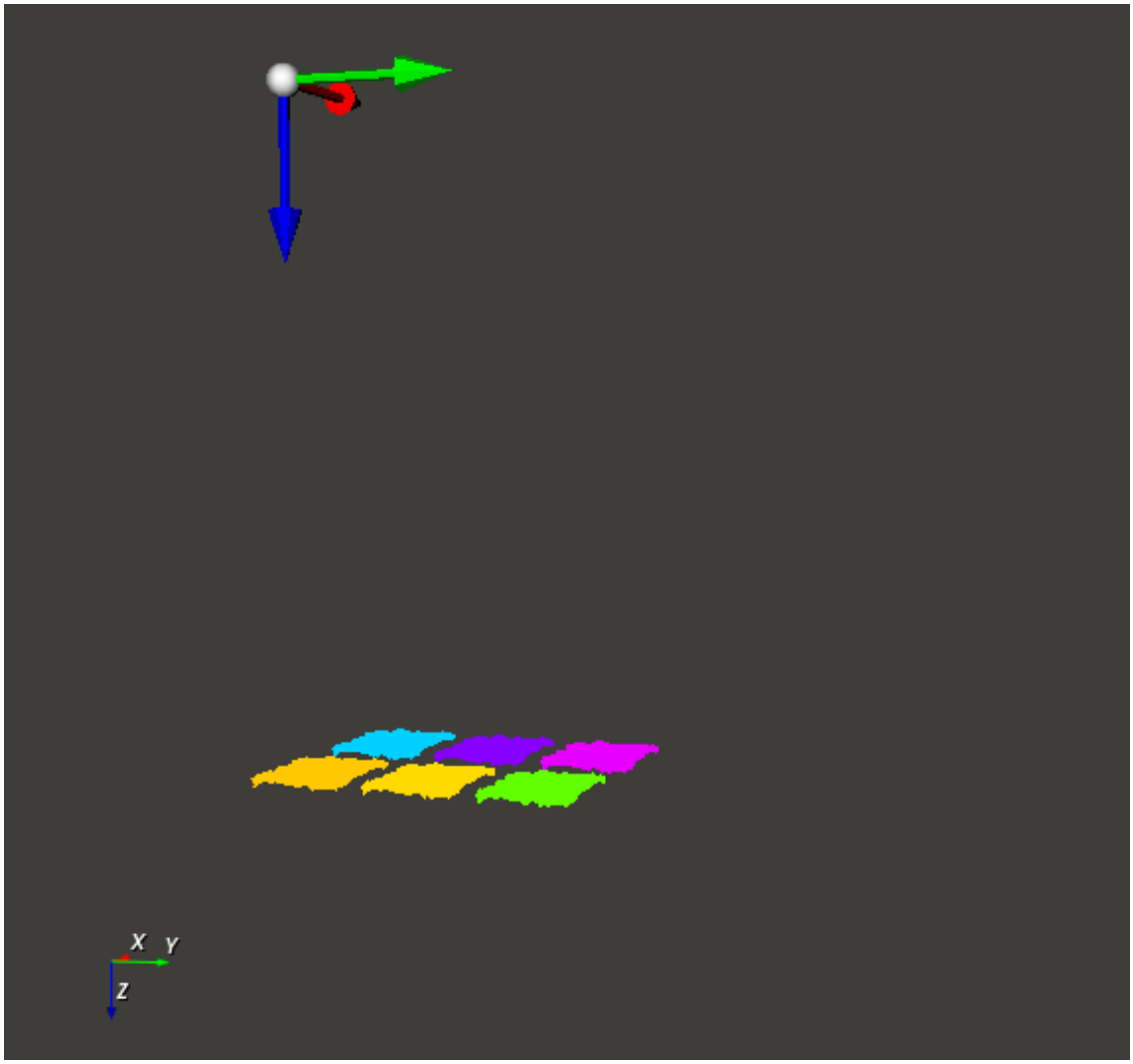
点击 RVS 运行按钮，触发 Trigger 算子

运行结果

1. 打开 ClusterExtraction 算子的 **点云列表** 可视化属性值，3D 视图显示如下：



2. 打开 FilterBoxList 算子的 **点云列表** 可视化属性值。3D 视图筛选出点云列表的左下角箱子及其周边 0.5 米范围内的箱子，3D 视图显示如下：


X
Y
Z

AdjustBox 调整箱子姿态

AdjustBox 算子用于调整箱子 cube 的姿态 pose，在保持箱子位置不变的前提下，将箱子的姿态 pose 调整到便于机械臂抓取。

算子参数

- **选择箱子/select_box**：将满足所选条件的箱子进行调整，不满足所选条件的箱子保持不变。
  - all_boxes：调整所有的箱子。
  - square_boxes：只调整正方形箱子，即 width 和 height 相差小于5%的箱子。
  - width > height：只调整 width > height 的箱子，即 x 轴对应的边长 > y 轴对应的边长。
  - height > width：只调整 height > width 的箱子，即 y 轴对应的边长 > x 轴对应的边长。
  - width > depth：只调整 width > depth 的箱子，即 x 轴对应的边长 > z 轴对应的边长。
  - depth > width：只调整 depth > width 的箱子，即 z 轴对应的边长 > x 轴对应的边长。
  - height > depth：只调整 height > depth 的箱子，即 y 轴对应的边长 > z 轴对应的边长。
  - depth > height：只调整 depth > height 的箱子，即 z 轴对应的边长 > y 轴对应的边长。
- **调整模式/adjust_mode**：选择调整的方式。
  - transform_pose：以箱子的 pose 为基准，根据 adjust_pose 的 yaw、pitch、roll 参数依次旋转。参数一般设置为 π/2 的整数倍，若不等于 π/2 的整数倍则默认调整到最接近的 nπ/2。
  - reference_pose：将箱子的 pose 调整到最接近参数 adjust_pose 的姿态上。
- **调整坐标/adjust_pose**：设置调整的姿态。只参照 roll、pitch、yaw 的值，x、y、z 的值忽略。
- **立方体/cube**：设置 cube 在 3D 视图中的可视化属性。
  -  打开立方体可视化。
  -  关闭立方体可视化。
  -  设置立方体的颜色。取值范围：[-2,360] 。默认值：-2 。
  -  设置立方体的透明度。取值范围：[0,1] 。默认值：0.5 。
- **坐标/pose**：设置 pose 在 3D 视图中的可视化属性。
  -  打开 pose 可视化。
  -  关闭 pose 可视化。
  -  设置 pose 的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。
- **立方体列表/cube_list**：设置 cube 列表在 3D 视图中的可视化属性。参数值描述与 **立方体** 一致。
- **坐标列表/pose_list**：设置 pose 列表在 3D 视图中的可视化属性。参数值描述与 **坐标** 一致。

数据信号输入输出

输入：

- **cube**：
  - 数据类型：Cube
  - 输入内容：立方体数据
- **cube_list**：
  - 数据类型：CubeList
  - 输入内容：立方体列表数据

输出：

- `cube`：
  - 数据类型：Cube
  - 输出内容：调整后的箱子
- `pose`：
  - 数据类型：Pose
  - 输出内容：调整后箱子的姿态
- `cubelist`：
  - 数据类型：CubeList
  - 输出内容：调整后的箱子列表
- `poselist`：
  - 数据类型：PoseList
  - 输出内容：调整后箱子的姿态列表

功能演示

使用 AdjustBox 算子调整箱子的姿态。

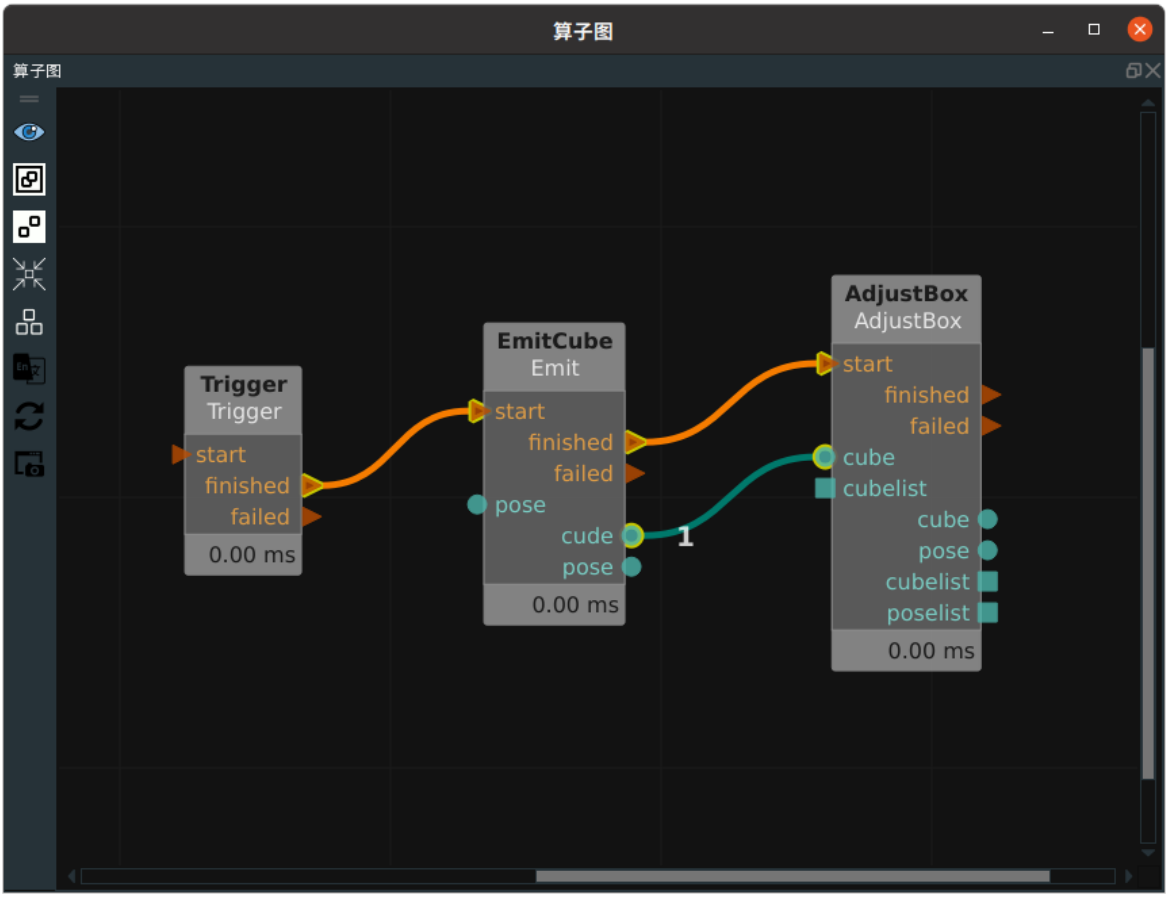
步骤1：算子准备

添加 Trigger 、Emit 、AdjustBox 算子至算子图。

步骤2：设置算子参数

1. 设置 Emit 算子参数：
   - 算子名称 → EmitCube
   - 类型 → Cube
   - 坐标→ 0.5 0.5 0 0.6 0.8 1.2
   - 宽度→ 0.5
   - 高度→ 0.3
   - 深度 → 0.01
   - 立方体 →  可视
2. 设置 AdjustBox 算子参数：
   - 选择箱子 → all_boxes
   - 调整模式 → transform_pose
   - 调整坐标 → 0 0 0 0 0 1.5708
   - 立方体 →  可视
   - 坐标 →  可视

步骤3：连接算子

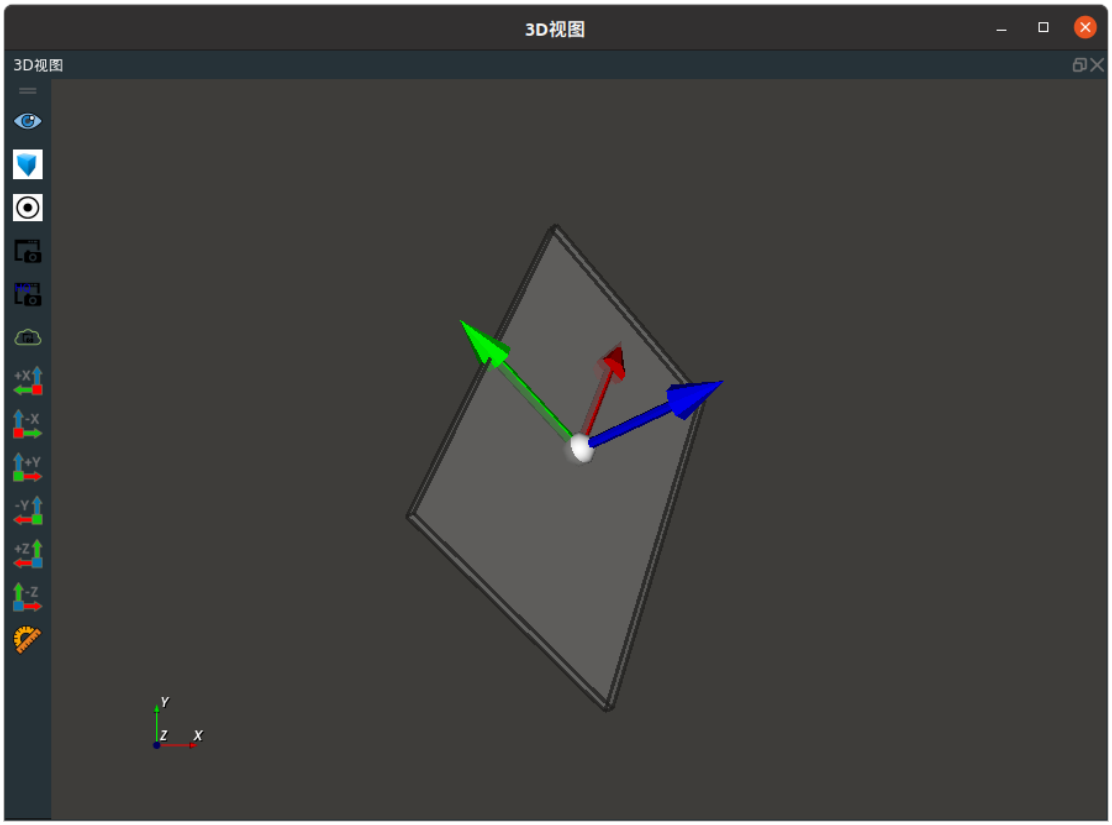


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

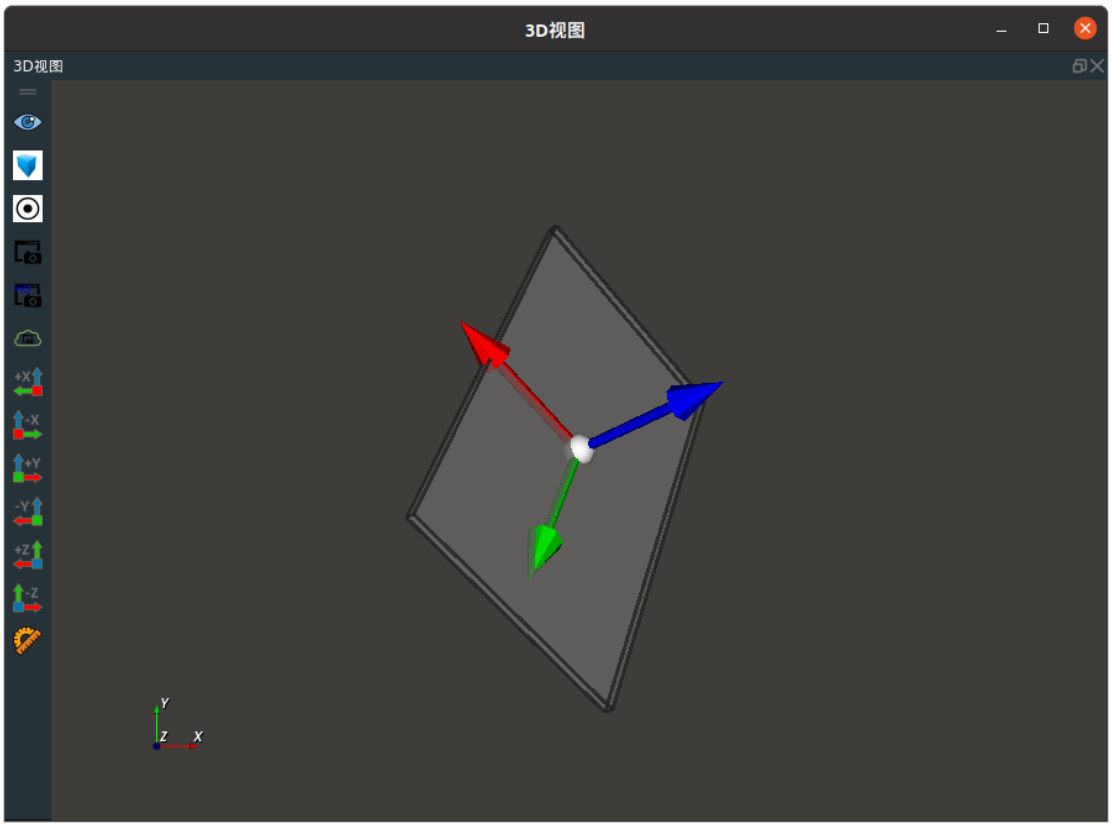
运行结果

1. 打开 EmitCube 算子的 `立方体` 和 `坐标` 可视化属性值。3D 视图显示如下：



2. 关闭 EmitCube 算子的 `立方体`和 `坐标` 可视化属性值。

3. 打开 AdjustBox 算子的 **立方体** 和 **坐标** 可视化。在 3D 视图中显示 AdjustBox 算子的结果。调整后的箱子 pose 由原姿态绕 z 轴旋转 90° 。



BackgroundBuilder 建立背景

BackgroundBuilder 算子用于包裹测量算法，算子功能是建立背景，保存背景数据到指定路径和文件。

算子参数

- `文件路径/file_path`：保存背景数据的路径。
- `文件名后缀/file_suffix`：保存背景数据的文件后缀。
- `m_roi_x`：ROI 区域的左顶点图像列坐标。
- `m_roi_y`：ROI 区域的左顶点图像行坐标。
- `m_roi_w`：ROI 区域的图像像素宽度。
- `m_roi_h`：ROI 区域的图像像素高度。
- `深度图像/depth_image`：设置 ROI 区域在 2D 视图中的可视化属性。
  -  打开 ROI 区域可视化。
  -  关闭 ROI 区域可视化。
- `彩色图像/color_image`：设置相机的彩色图 2D 视图中的可视化属性。
  -  打开相机的彩色图可视化。
  -  关闭相机的彩色图可视化。

数据信号输入输出

输入：

- `depth`：
  - 数据类型：Image
  - 输入内容：相机的深度图
- `color`：
  - 数据类型：Image
  - 输入内容：相机的彩色图
- `depth_calib`：
  - 数据类型：CalibInfo
  - 输入内容：相机的深度图标定数据
- `color_calib`：
  - 数据类型：CalibInfo
  - 输入内容：相机的彩色图标定数据

输出：

- `depthView`：
  - 数据类型：Image
  - 输出内容：相机的深度图转换为 RGB 显示，并显示 ROI 区域
- `colorView`：
  - 数据类型：Image
  - 输出内容：相机的彩色图

功能演示

使用 BackgroundBuilder 建立背景，用于包裹测量。

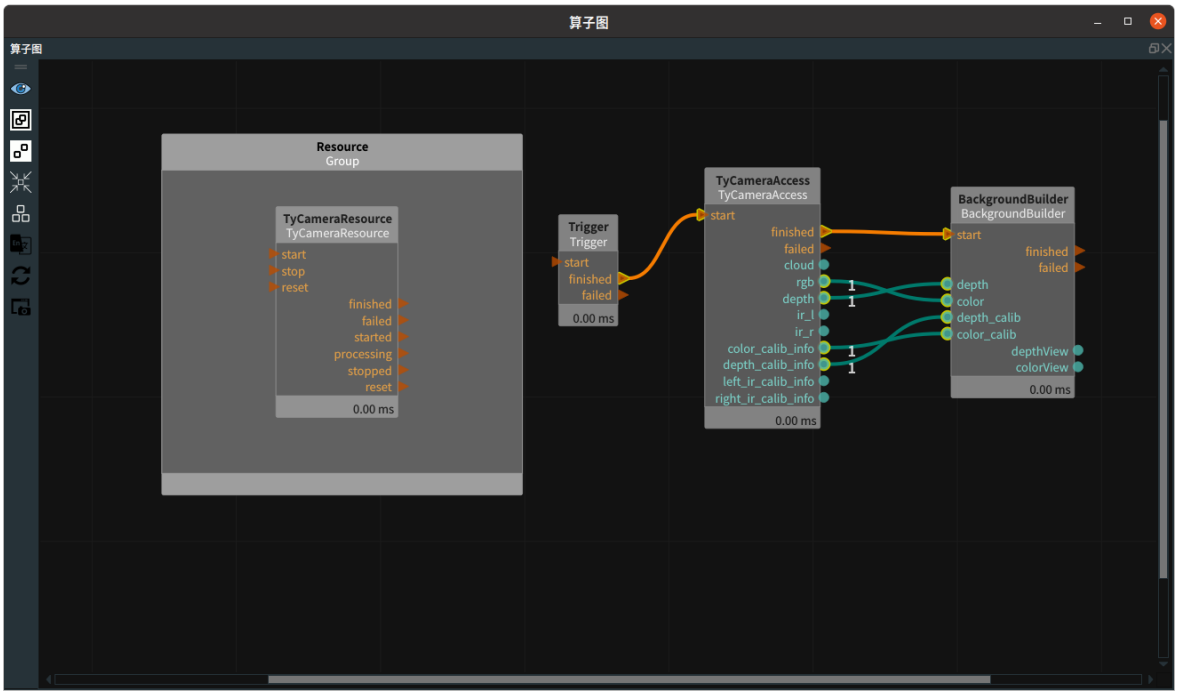
步骤1：算子准备

1. 添加 TyCameraResource 至 Resource Group。
2. 添加 TyCameraAccesss 、Trigger 、BackgroundBuilder 算子至算子图。

步骤2：设置算子参数

1. 设置 BackgroundBuilder 算子参数：
    - 文件路径 → ./bg
    - m_roi_x → 350
    - m_roi_y → 250
    - m_roi_w → 600
    - m_roi_h → 500
    - 深度图像 →  可视
    - 彩色图像 →  可视

步骤3：连接算子

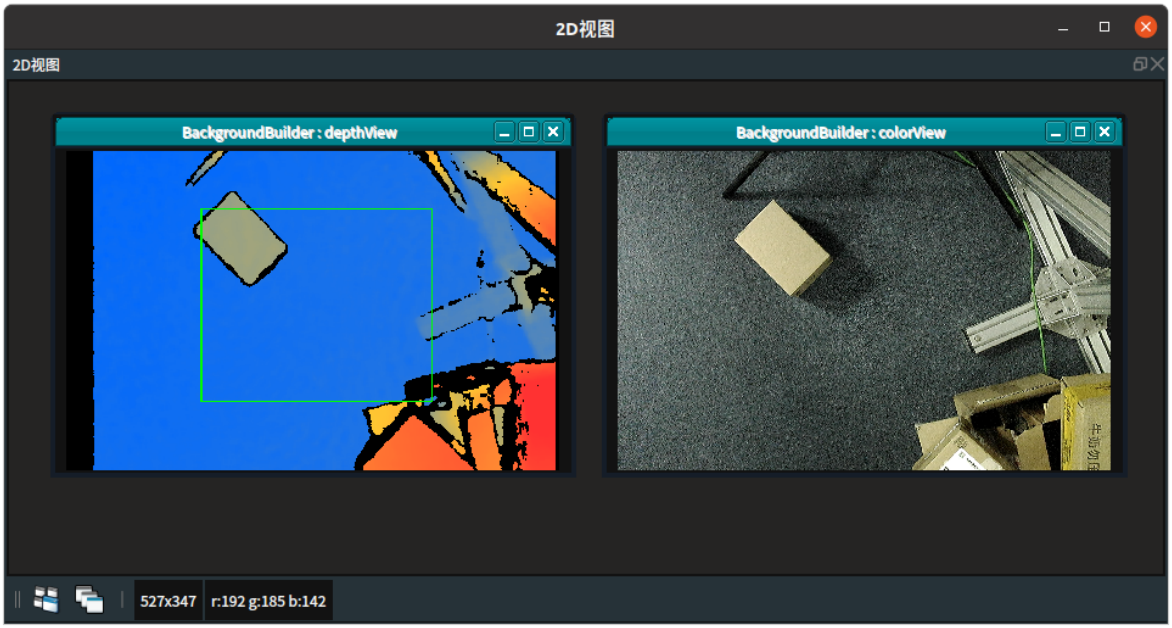


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子

运行结果

如下图所示，2D 视图显示 BackgroundBuilder 算子的 深度图像 和 彩色图像 。



运行成功后，会在 ./bg 目录下会生成下面三个背景文件：背景彩色图，背景深度图，背景平面方程：

calibration

AXXBSolver AXXB标定法

AXXBSolver 算子通过求解 *AX = XB 方程* 中的最小误差转换关系，获得相机坐标系与机器人坐标系的最佳转换矩阵，即手眼标定的结果，从而实现机器人与相机之间的坐标转换。其中， A 为机器人连杆末端在机器人坐标系下的位姿数据组，B 为标定板在相机坐标系下的位姿数据组。标定结果以 Pose 类型形式呈现。

算子参数

- **结果坐标/result_pose**：设置标定结果 Pose 在 3D 视图中的可视化属性。
  -  打开标定结果 Pose 可视化。
  -  关闭标定结果 Pose 可视化。
  -  设置标定结果 Pose 的尺寸大小。取值范围：[0.001~10]。默认值：0.1。

数据信号输入输出

输入：

- **tcp_poses**：
  - 数据类型：PoseList
  - 输入内容：机器人末端坐标组
- **chessboard_poses**：
  - 数据类型：PoseList
  - 输入内容：标定板坐标组

说明：建议使用15组（或更多组）棋盘格标定板坐标组数据。最少不低于6组，否则无法保证精度。

输出：

- **result_pose**：
  - 数据类型：Pose
  - 输出内容：手眼标定结果

功能演示

使用 AXXBSolver 算子获得相机坐标系与机器人坐标系的手眼标定的结果。

步骤1：算子准备

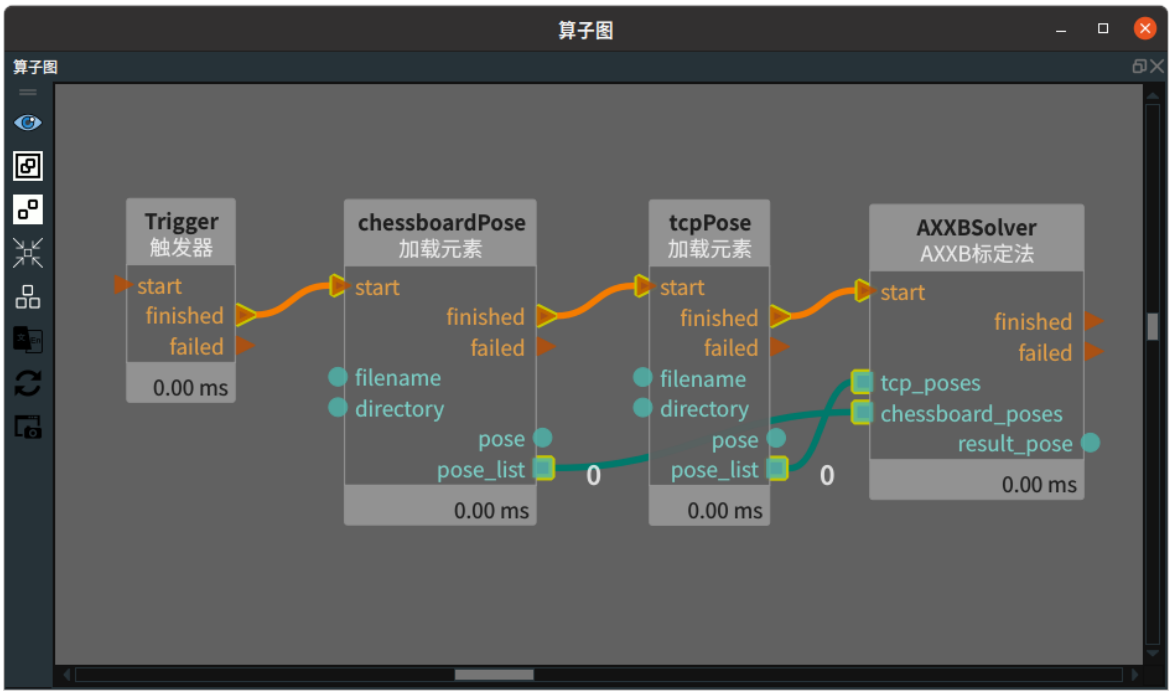
添加 Trigger、Load （2个）、AXXBSolver算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 算子名称 → chessboardPose
   - 类型 → pose
   - 文件 → ••• → 选择pose文件名 (*example_data/AXXB/chessboard.txt*)
   - 坐标列表 →  可视

2. 设置 Load_1 算子参数：
    - 算子名称 → tcpPose
    - 类型 → pose
    - 文件 → ••• → 选择pose文件名 (*example_data/AXXB/tcpPose.txt*)
    - 坐标 →  可视
3. 设置 AXXBSolver 算子参数：结果坐标 →  可视

步骤3：连接算子

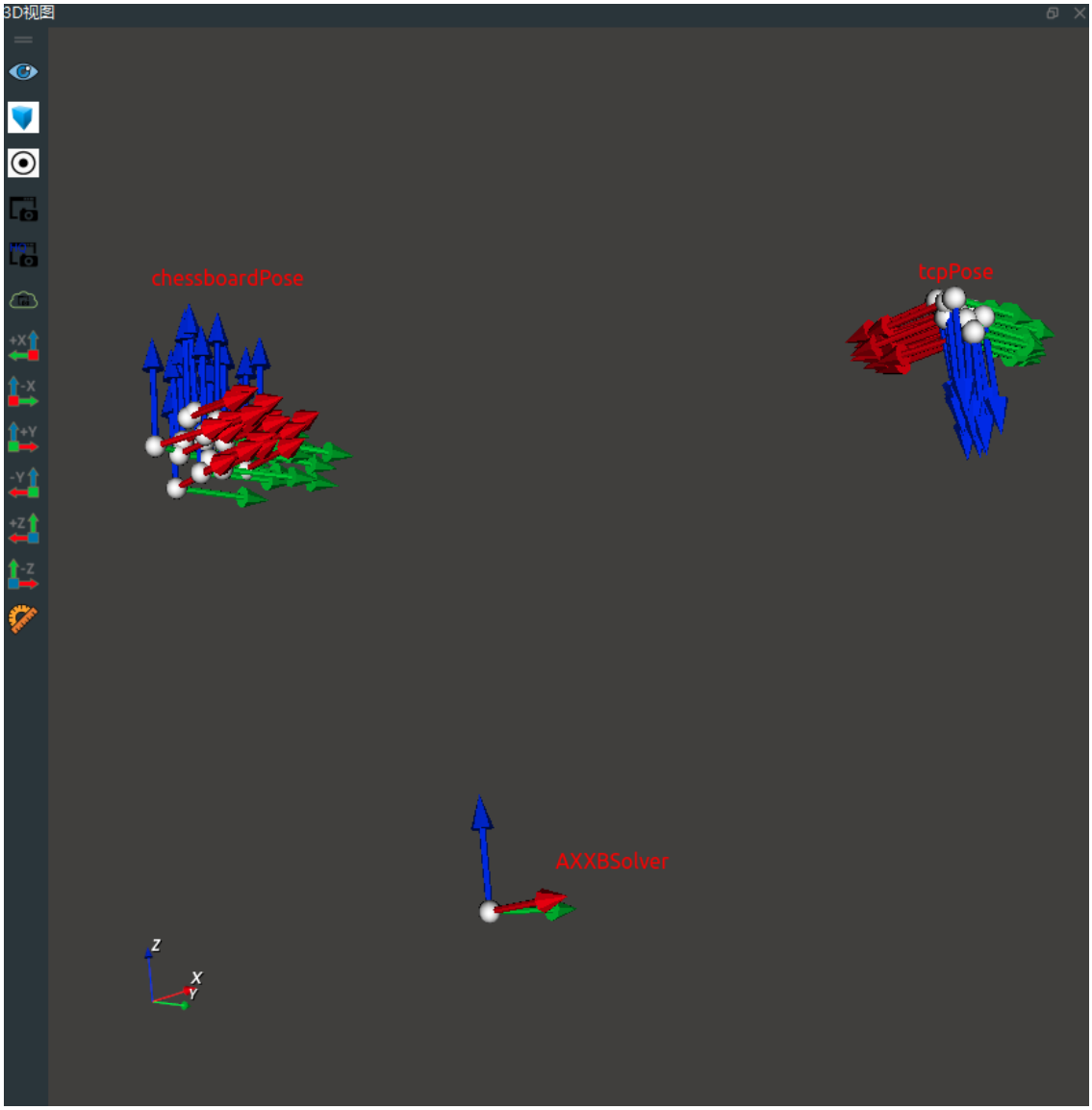


步骤4：运行

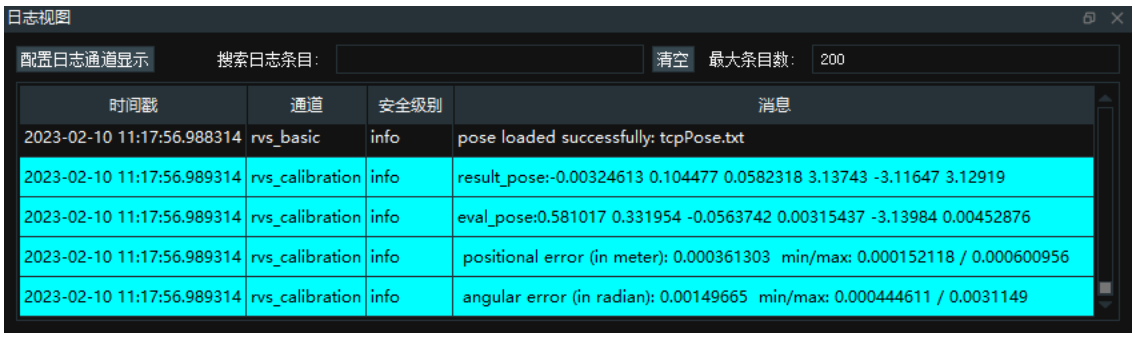
点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

1. 运行结果显示如下，3D 视图中分别展示了 Load(chessboardPose)、Load（tcpPose）、AXXBSolver 算子的可视化结果。



2. 查看日志，显示标定结果和误差值。



以上图为例：

- result_pose 为标定结果 Pose，与算子 AXXBSolver 输出一致。
- positional error 为标定位置误差（与输入数据单位一致，本例中为“米”），在本例中 0.000361303 为平均误差，0.000152118 和 0.000600956 分别为最小误差和最大误差。

说明：精度误差值符合项目需求即标定完成，若误差略大，超过项目需求可再次重复标定流程，录制更多数据进行标定。若反复多次标定结果均不理想，需结合硬件设备性能考虑精度需求。手眼标定误差由相机误差和机械臂误差叠加。若标定误差巨大，有可能是录入数据格式或顺序不正确，详见 教学案例 -> 自动拆垛 -> 手眼标定部分。

ChessboardLocalization 棋盘定位

ChessboardLocalization 算子用于对含有黑白棋盘格的标定板 2D 图像进行角点识别以及空间定位，从而得到棋盘格内角点的像素列表，以及棋盘格原点在RGB镜头3D坐标系下的位姿（位置和姿态）。

“棋盘格原点”对应内角点的四个边缘点的某个点，具体的对应方式可以参照文档下述的案例。

每一张标定图像中必须要包含黑白格所有的内角点。在内角点清晰可见的基础上，最边缘的黑白格被遮挡一部分不影响结果识别。

拍摄标定板图像时，相机和标定板要保持静止，并且两者距离不能过远，否则容易导致角点像素模糊。

该算子一般用于手眼标定。手眼标定过程中标定图像不应低于6张，建议15张或更多。拍摄完一张标定板图像后，相机和标定板之间的距离以及相对位姿需要重新调整。调整范围需要尽量覆盖到实际工作场景中的整个3D空间。

算子参数

- **棋盘行数/number_rows**：棋盘格标定板的格子行数。默认值：6。
- **棋盘列数/number_colors**：棋盘格标定板的格子列数。默认值：9。（行列区分见功能演示）
- **棋盘格尺寸毫米/square_size_in_millimeter**：棋盘格标定板的每个小格的长度。默认值：30。单位：mm。
- **相机标定文件/calib_file**：相机RGB镜头的出厂标定参数文件路径。参数文件包括外参（ RGB 镜头到 3D 镜头的转换矩阵），内参以及畸变参数。当 calib_info 的输入端口有连接时，则该参数无效。当 calib_info 的输入端口没有连接时，则必须给该参数赋值。
- **使用畸变参数/use_diff_coeff**：是否启用畸变校正。默认：False。因为 TyCameraResource 默认去畸变，我们传入的图像一般都是已经去畸变的图像。
  - True：启用畸变矫正。
  - False：关闭畸变矫正。
- **坐标/pose**：设置棋盘格原点 pose 在 3D 视图中的可视化属性。
  -  打开 Pose 可视化。
  -  关闭 Pose 可视化。
  -  设置 Pose 的尺寸大小。取值范围：[0.001,10]。默认值：0.1。
- **图像/image**：设置 result 在 2D 视图中的可视化属性。
  -  打开 result 可视化。
  -  关闭 result 可视化。

数据信号输入输出

输入：

- **image**：
  - 数据类型：Image
  - 输入内容：标定板图像
- **calib_info**：
  - 数据类型：CalibInfo
  - 输入内容：相机RGB镜头出厂标定参数

输出：

- `pose`：
  - 数据类型：Pose
  - 输出内容：棋盘格标定板的原点在相机 RGB镜头 3D 坐标系下的位姿
- `points`：
  - 数据类型：ImagePoints
  - 输出内容：棋盘格标定板的像素列表(一系列 2d 点)
- `result`：
  - 数据类型：Image
  - 输出内容：使用像素列表渲染后的结果图像

功能演示

使用 ChessboardLocalization 算子对含有标定板的 2D 图像进行角点识别以及空间定位。得出棋盘格标定板的像素列表，同时得出棋盘格原点在 3D 空间中的位置和姿态。

步骤1：算子准备

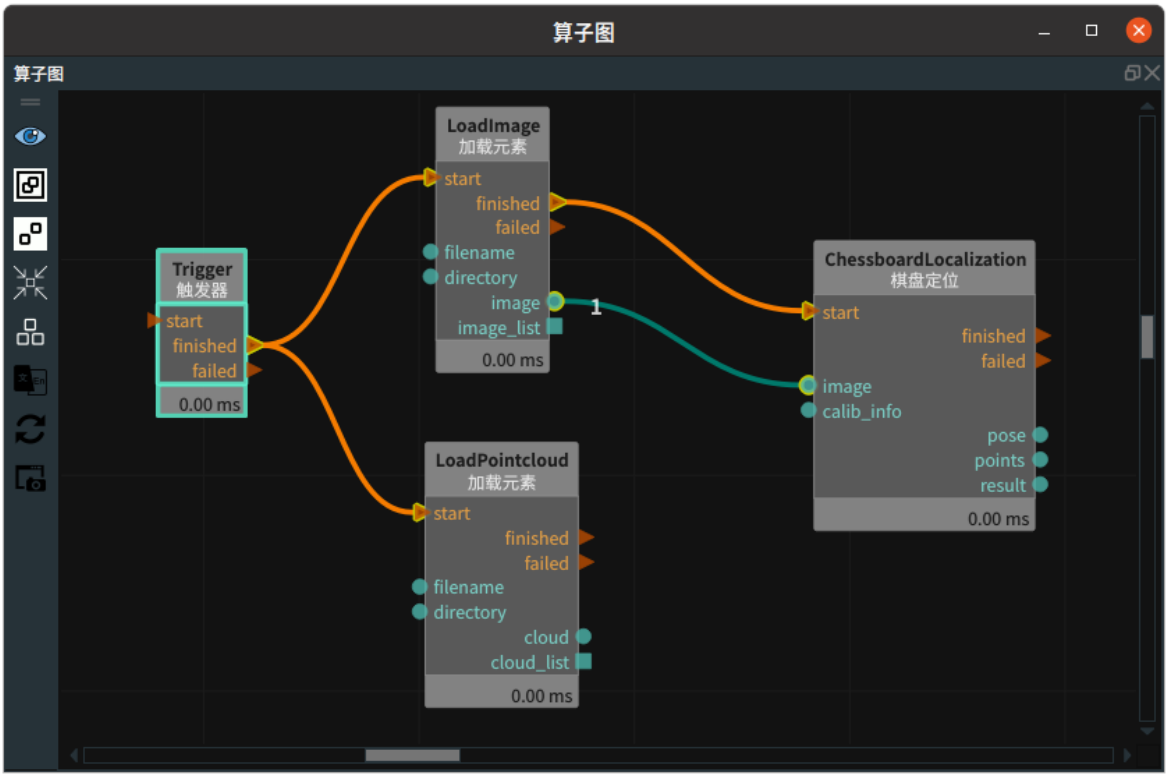
添加 Trigger、Load（2个）、ChessboardLocalization 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → Image
   - 文件 → ••• → 选择图像文件名（ *example_data/ChessboardLocalization/rgb.png* ）
   - 图像 → 可视
2. 设置 Load 算子参数：
   - 算子名称 →LoadPointCloud
   - 类型 → PointCloud
   - 文件 → ••• → 与加载的图像匹配的点云 （ *example_data/ChessboardLocalization/cloud.pcd* ）
   - 点云 → 可视
3. 设置 ChessboardLocalization 算子参数：
   - 棋盘行数 → 6（实际工作中根据所拍摄的标定板的实际数据更改）
   - 棋盘列数 → 9（实际工作中根据所拍摄的标定板的实际数据更改）
   - 棋盘格尺寸毫米 → 30 （实际工作中根据所拍摄的标定板的实际数据更改）

     相机标定文件 → example_example_data/ty_chessboard_calib.txt (实际工作中赋值实际所用相机的rgb参数文件赋值)
   - 坐标 → 可视
   - 图像 → 可视

步骤3：连接算子



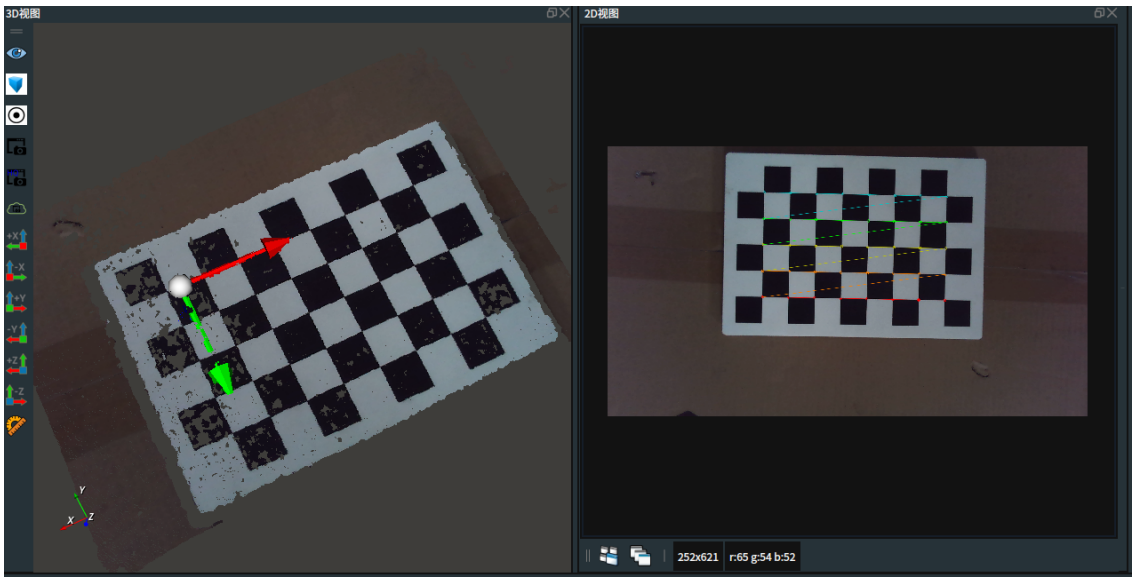
步骤4：运行

点击 RVS 的运行按钮，触发 Trigger 完成第一次的运行。

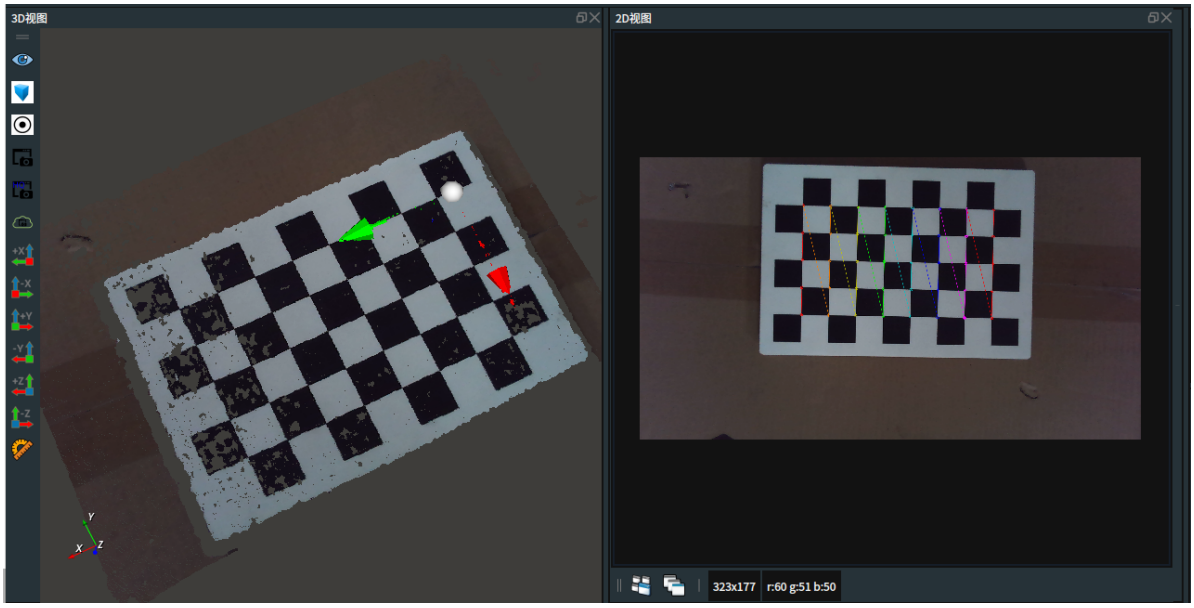
运行结果

1. 结果如下图所示。得出棋盘格标定板的像素列表，同时得出棋盘格原点在 3D 空间中的位置和姿态。

   说明：受限于成像误差、出厂标定参数误差以及计算误差，下图所得的棋盘格原点位姿同真实值有一些偏移，其真实值应该同棋盘格左上角内角点重合。所以我们在进行手眼标定计算时，一般使用多组标定图像来进行回归运算减少误差。



2. 将 ChessboardLocalization 算子的 **棋盘行数**、**棋盘列数** 数值对换。重新运行，结果如下图所示（两幅图的 3d 观察视角保持一致）。容易发现，红色的 x 轴始终沿着目标设定的 **棋盘列数** 方向。上图运行时 **棋盘列数** ：9，下图运行时 **棋盘列数** ： 6。


3D视图
2D视图
323x177
r:60 g:51 b:50

LoadCalibFile 加载标定文件

LoadCalibFile 算子用于从图漾相机的参数文件中读取相机内参、外参、畸变参数，并将结果显示在算子属性栏中。参数文件目前包含rgb镜头参数文件以及depth镜头参数文件两类。

以rgb镜头参数文件为例：

序号1：表示相机的RGB镜头在进行出厂标定时所采用的镜头分辨率。

序号2：表示相机RGB镜头的内参。总共9个数值，是一个按行排列的3 * 3矩阵。矩阵形式为[fx,0,cx,0,fy,cy,0,0,1],其中fx表示x轴方向的相机焦距(单位为像素)，其中cx表示x轴方向的相机中心(单位为像素)；fy、cy类似。

序号3：表示相机RGB镜头的外参，即RGB镜头到左IR镜头的空间坐标系转换矩阵。总共16个数值，是一个按行排列的4 * 4矩阵。左上角的3 * 3模块表示旋转，右上角的1 * 3表示平移。平移值的单位为毫米。

序号4：表示相机RGB镜头的畸变参数，总共12个数值。可以用来对原始的RGB图像进行畸变校正。

说明：关于depth_calib_file,其说明同上述说明保持一致。由于深度相机是虚拟相机，所以其畸变参数以及外参都是零矩阵。

算子参数

- **相机标定文件/calib_file**：同输入端口的 calib_filename 作用一致。当输入端口 calib_filename 没有连接时，则必须给该 calib_filename 参数赋值。当输入端口calib_filename有连接时，则在执行算子后会自动将输入端口的数值覆盖该 calib_filename 参数的值。
- **res_x/y**：相机的 x 轴像素分辨率-相机的 y 轴像素分辨率。
- **fx/fy/cx/cy**：相机的 x 轴像素焦距-y轴像素焦距 -x 轴像素中心 -y 轴像素中心。
- **相机外参坐标/extrinsic**：外参，Pose 形式的转换矩阵。侧边按钮可以快捷复制、粘贴、重置操作。
- **k1~s4**：畸变参数。
- **相机外参坐标/extrinsic_pose**：设置 extrinsic_pose 在 3D 视图中的可视化属性。
  - 打开extrinsic_pose 可视化。
  - 关闭 extrinsic_pose 可视化。
  - 设置 extrinsic_pose 的尺寸大小。取值范围：[0.001~10]。默认值：0.1。

数据信号输入输出

输入：

- `calib_filename`：
  - 数据类型：String
  - 输入内容：标定参数文件路径

输出：

- `extrinsic_pose`：
  - 数据类型：Pose
  - 输出内容：相机RGB镜头的外参转换矩阵
- `calib_info`：
  - 数据类型：CalibInfo
  - 输出内容：相机RGB镜头的所有标定参数

功能演示

使用 LoadCalibFile 算子加载图漾相机的参数文件。

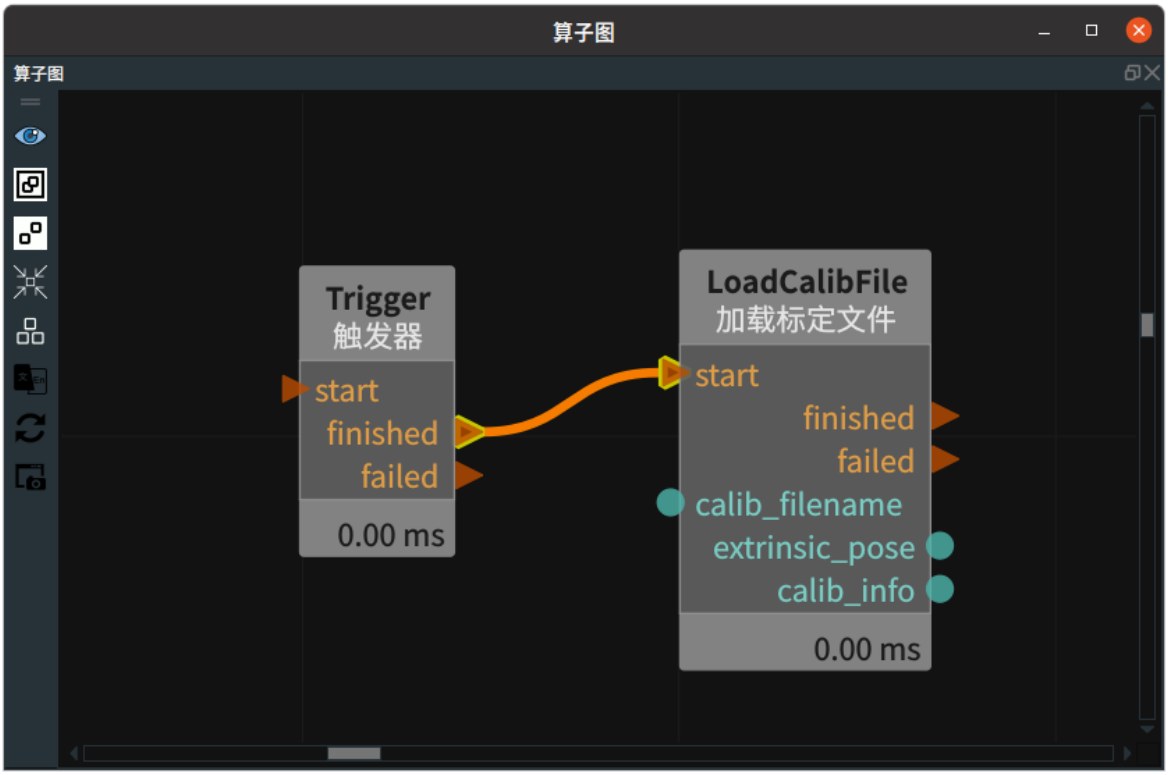
步骤1：算子准备

添加 Trigger、LoadCalibFile 算子至算子图。

步骤2：设置算子参数

设置 LoadCalibFile 算子参数：相机标定文件 → ●●● → ty_chessboard_calib.txt

步骤3：连接算子

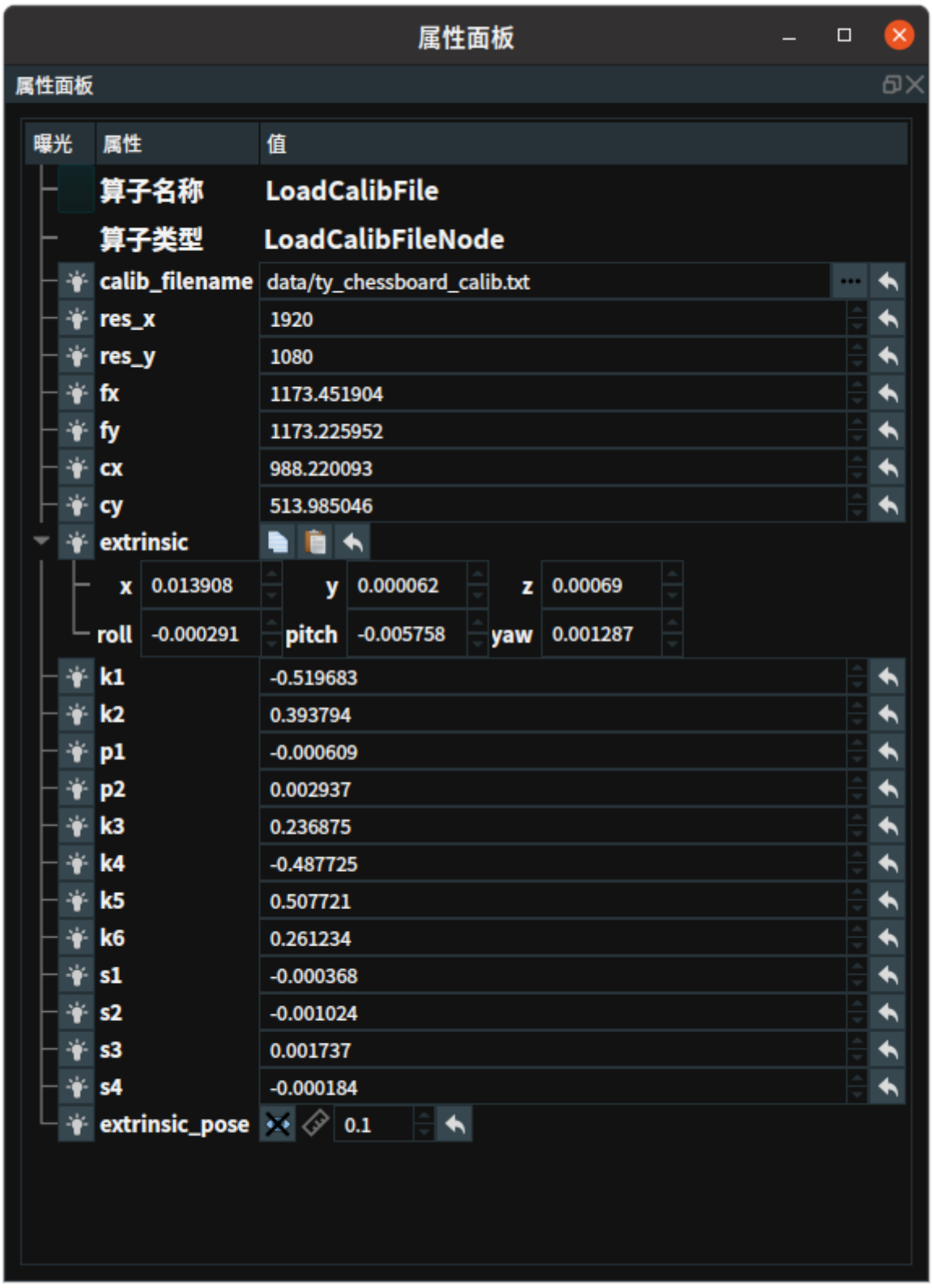


步骤4：运行

点击 RVS 的运行按钮，触发Trigger算子。

运行结果

结果如下图所示，在属性面板中显示图漾相机的参数文件中读取相机内参(像素焦距、像素中心)、外参(RGB 镜头到 3D 镜头的转换矩阵)、畸变参数。



MaskToRotatedRect Mask转旋转矩形

MaskToRotatedRect 用于将 AIDetect 算子推理的 mask_list 和 mask_class_list 转换为旋转矩形。可选择指定的类别输出。

类型	功能
MaskList	将 Mask 列表转换为旋转矩形。
MaskClassList	将 Mask 类别列表转换为旋转矩形。

MaskList

将 MaskToRotatedRect 算子`类型`选择 MaskList，用于将 Mask 列表转换为旋转矩形。

算子参数

- `选择类别/select_class`：当有数据连接到 mask_name_list 端口时，可以输入指定的类别来输出对应的旋转矩形。
- `矩形列表/rect_list`：设置旋转矩形列表的曝光属性。
  -  打开曝光。
  -  关闭曝光。

数据信号输入输出

输入：

- `mask_list`：
  - 数据类型：Image
  - 输入内容：Mask 列表
- `mask_name_list`：
  - 数据类型：String
  - 输入内容：Mask 名称列表

说明：该端口连接/不连接，输出所有类别的旋转矩阵。连接时，可填写 select_class 来输出指定类别的旋转矩形。

输出：

- `rect_list`：
  - 数据类型：RotatedRectList
  - 输出内容：旋转矩形列表

功能演示

使用 MaskToRotatedRect 算子中选择 MaskList，用于将 Mask 列表中 pear 转换为旋转矩形。

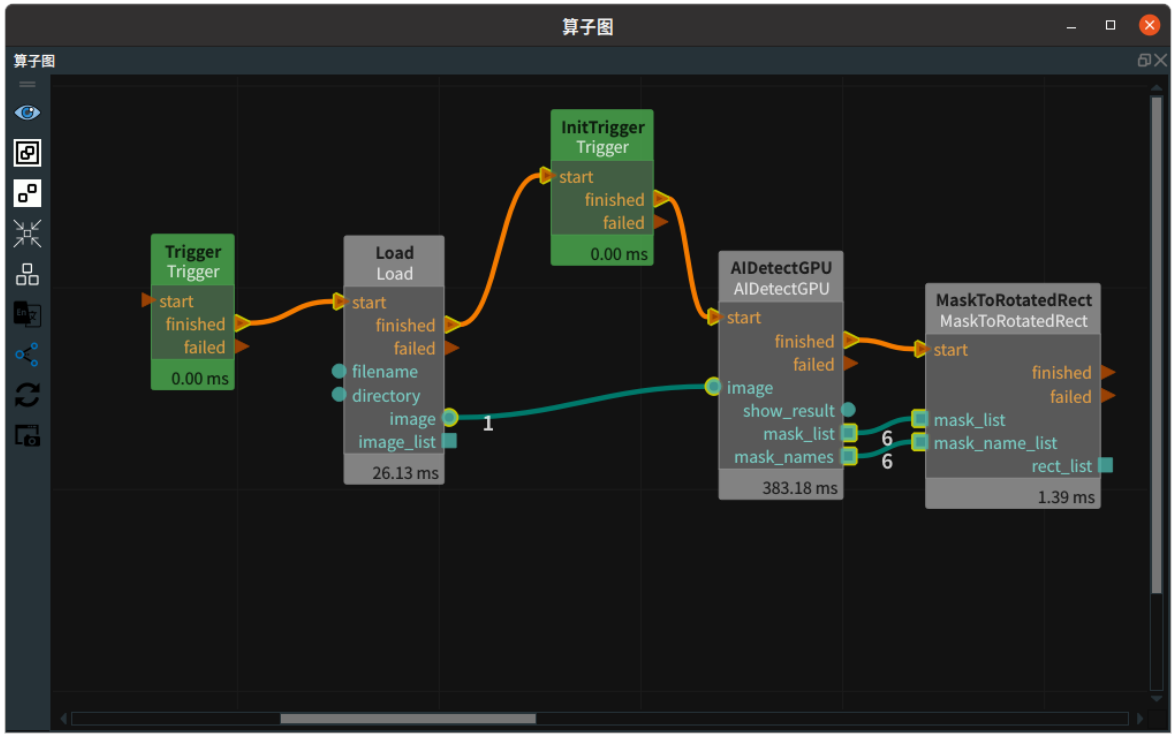
步骤1：算子准备

添加 Trigger（2个）、Load 、AIDetectGPU、MaskToRotatedRect 算子至算子图。

步骤2：设置算子参数

1. 设置 Trigger 算子参数：
    - 算子名称 → InitTrigger
    - 类型 → InitTrigger
2. 设置 Load 算子参数：
    - 类型 → Image
    - 文件 → ••• → 选择pose文件名 (*example_data/mask_data_train/20221010141101489/rgb.png*)
3. 设置 AIDetectGPU 算子参数：
    - 类型 → MaskRCNN
    - 类名文件路径 → ••• → 选择相应文件名 （ *example_data/mask_data_train/fruits.txt*)
    - 权重文件路径 → ••• → 选择相应权重文件名 （ *example_data/mask_data_train/train_output/model_final.pth*)
    - 配置文件路径 → ••• → 选择相应配置文件名 （ *example_data/mask_data_train/train_output/config.yaml*)
    - 物体得分阈值 → ••• → 0.75
    - 识别结果图像 →  可视
4. 设置 MaskToRotatedRect 算子参数：
    - 类型 → MaskList
    - 选择列表 → pear

步骤3：连接算子

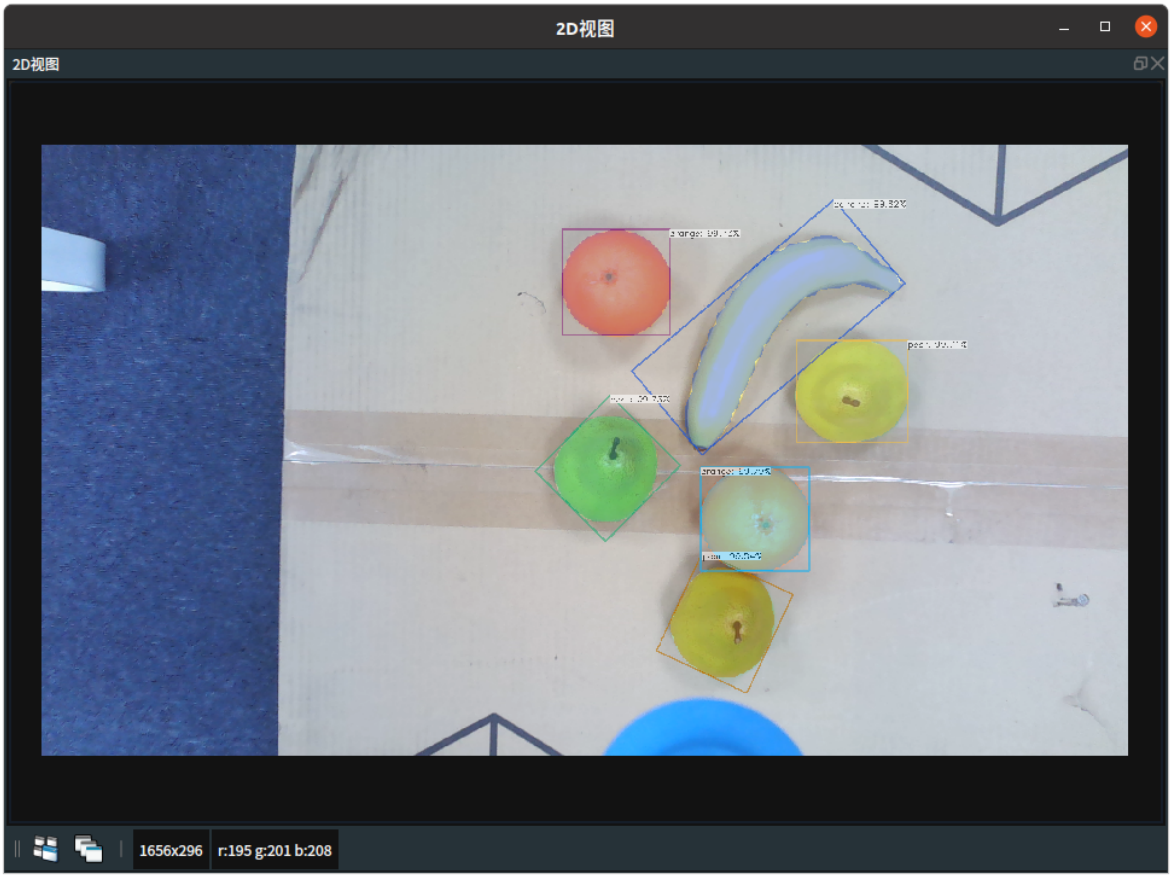


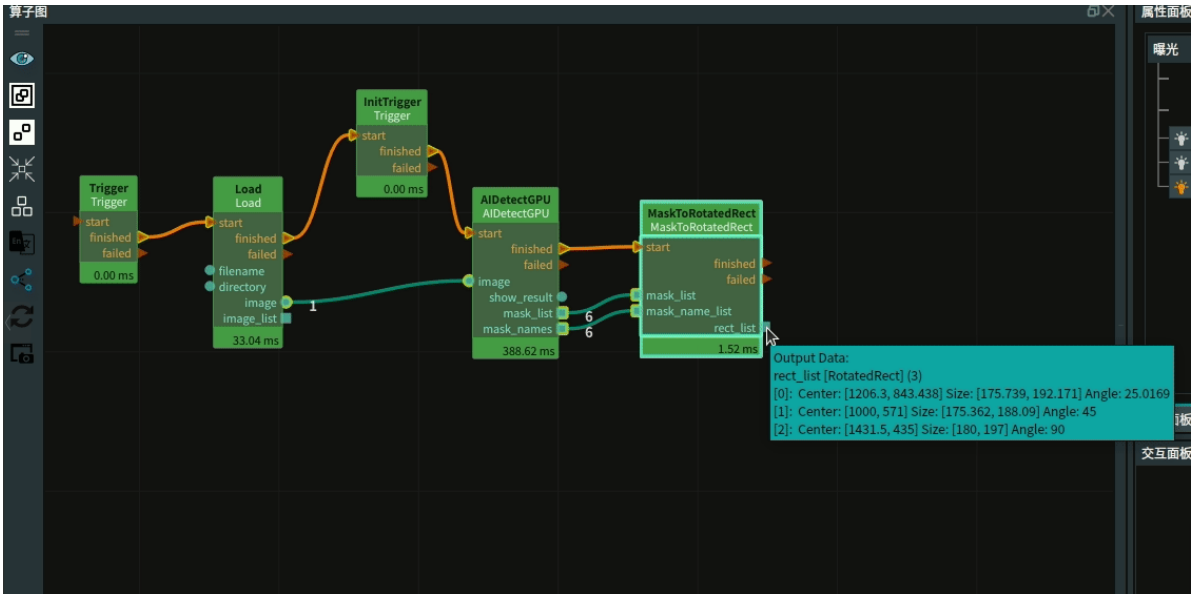
步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

运行结果显示如下，2D 视图中输出 AIDetectGPU 算子的结果。可以看到有 3个梨子，MaskToRotatedRect 算子输出端口输出类别为梨子的旋转矩形列表。





MaskClassList

将 MaskToRotatedRect 算子类型选择 MaskClassList，用于将 Mask 类别列表转换为旋转矩形。

算子参数

- **选择类别/select_class**：当有数据连接到 mask_name_list 端口时，可以输入指定的类别来输出对应的旋转矩形。
- **矩阵列表/rect_list**：设置旋转矩形列表的曝光属性。
  - 打开曝光。
  - 关闭曝光。

数据信号输入输出

输入：

- `mask_class_list`：
  - 数据类型：Image
  - 输入内容：Mask 类别列表
- `mask_name_list`：
  - 数据类型：String
  - 输入内容：Mask 名称列表

说明：该端口连接/不连接，输出所有类别的旋转矩阵。连接时，可填写 select_class 来输出指定类别的旋转矩形。

输出：

- `rect_list`：
  - 数据类型：RotatedRectList
  - 输出内容：旋转矩形列表

功能演示

与上述 MaskList 模块类似，请参照该章节功能演示模块。

ProjectPoints 映射像素点

ProjectPoints 算子用于将 rgb 图像中 2D 像素点 (ImagePoints 数据格式)转换为 3D 空间点(Pose 格式)。

type	功能
General	根据输入的点云图将2D像素点推理转换为 3D 点
Fast	根据输入的深度图将 2D 像素点推理转换为 3D 点, 相比于 General 模式处理速度提升十倍以上

General

算子参数

- **像素映射半径/pixel_project_radius**：由于点云成像受干扰可能存在缺失, 目标 2D 像素点可能没有对应的 3D 点云。此时算子会在该 2D 像素点周边 pixel_project_radius 的像素半径区域内查找所有具有对应 3D 点的像素点，将距离目标像素点最近的像素点对应的3D点作为映射结果。
- **相机坐标/camera_pose**：同输入端口的 camera_pose 作用一致。当输入端口 camera_pose 没有连接时，则必须给该 camera_pose 参数赋值。当输入端口camera_pose 有连接时，则在执行算子后会自动将输入端口的数值覆盖该 camera_pose 参数的值。
- **使用畸变参数/use_dist_coeff**：是否使用 rgb 图的畸变校正参数。默认不勾选，因为本算子的 image_points输入端口的像素点一般都是基于去畸变的图像。如果算子的image_points 输入端口的像素点是基于畸变图，则需要勾选该参数。
- **相机彩色图标定文件/color_calib_file**：图漾相机 rgb 镜头的出厂标定参数文件所在的文件地址，作用同输入端口的 color_calib_info 一致。当输入端口 color_calib_info没有连接时，则必须给该参数赋值。当输入端口 color_calib_info 有连接时，则优先使用输入端口的数值。
- **坐标列表/pose_list**：设置 pose 列表在 3D 视图中的可视化属性。
  -  打开 pose 列表可视化。
  -  关闭 pose 列表可视化。
  -  设置 Pose 的尺寸大小。取值范围：[0.001~10]。默认值：0.1。

数据信号输入输出

输入：

- **cloud**：
  - 数据类型：PointCloud
  - 输入内容：掩码图的原图所对应的点云
- **camera_pose**：
  - 数据类型：Pose
  - 输入内容：“下述输入端口的像素点所在 RGB 图对应的 RGB 空间坐标系“ 到 “上述输入端口的点云所在的空间坐标系“ 的转换矩阵。如果上述输入端口的点云位于相机3D坐标系，则这里输入RGB空间坐标系到相机3D坐标系的转换 Pose 即可。如果上述输入端口的点云位于机器人坐标系，则这里需要输入 RGB空间坐标系到机器人3D坐标系的的转换 Pose 。在点云是相机坐标系的情况，该Pose 即为 TyCameraResource 的 extrinsic_pose 参数，也可以直接由 LoadCalibFile 算子给出。

- `color_calib_info`：
  - 数据类型：CalibInfo
  - 输入内容：图漾相机的出厂标定参数，可以从 TyCameraAccess 算子输出，也可以由 LoadCalibFile 算子输出。
- `rgb`：
  - 数据类型：Image
  - 输入内容：**一般不需要连接该端口。**假设通过 image_points 输入的像素点是基于 M-N 尺寸的图像，与 color_calib_info 的出厂标定信息对应的图像尺寸一致时(常规情况) ，则无需连接该端口。与 color_calib_info 的出厂标定信息对应的图像尺寸不一致时，此时需通过该端口输入一张 M-N 尺寸的图像

输出：

- `pose_list`：
  - 数据类型：PoseList
  - 输出内容：将像素点转换成 3D 点后的 Pose 列表(仅位置信息有效，姿态角度全部为0)

功能演示

使用 ProjectPoints 算子将加载的 RGB 掩码图中 2D 像素点 (ImagePoints 数据格式)转换为 3D 空间点(Pose 格式)。

步骤1：算子准备

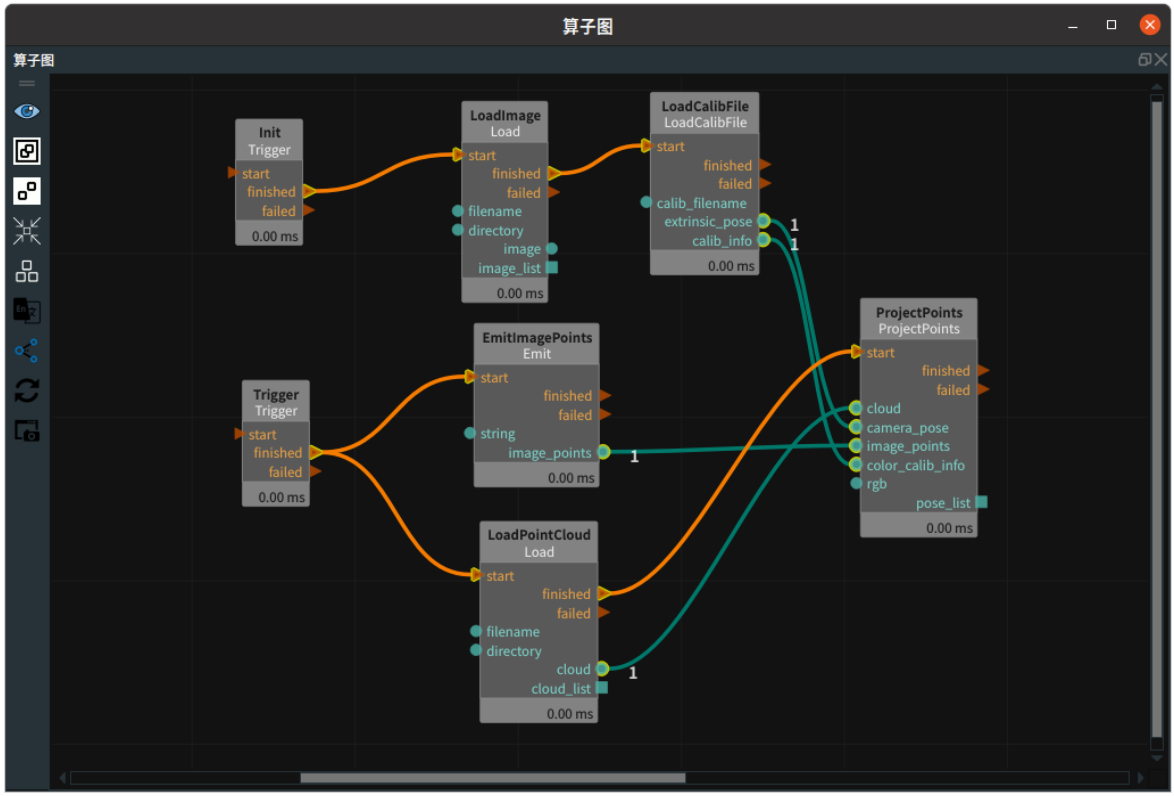
添加 Trigger（2 个）、Load（2 个）、LoadCalibFile、Emit、ProjectPoints算子至算子图。

步骤2：设置算子参数

1. 设置 Trigger 算子参数：
   - 算子名称 → Init
   - 类型 → InitTrigger
2. 设置 Load 算子参数：
   - 算子名称 → LoadImage
   - 类型 → Image
   - 文件→ ••• →选择图像文件名 （ *example_data/ProjectPoints/rgb.png* ）
   - 图像 →  可视
3. 设置 Load_1 算子参数：
   - 算子名称 → LoadPointCloud
   - 类型 → PointCloud
   - 文件→ ••• →选择与图像匹配点云文件名 （ *example_data/ProjectPoints/cloud.pcd* ）
   - 点云 →  可视 →  -2 →  2
4. 设置 LoadCalibFile 算子参数：
   - 算子名称 → rgb_Calib
   - 文件 → 图漾相机出厂标定文件 （ *example_data/ProjectPoints/ty_color_calib.txt* ）
5. 设置 Emit 算子参数：
   - 算子名称 → EmitImagePoints
   - 类型 →ImagePoints
6. 设置 ProjectPoints 算子参数：

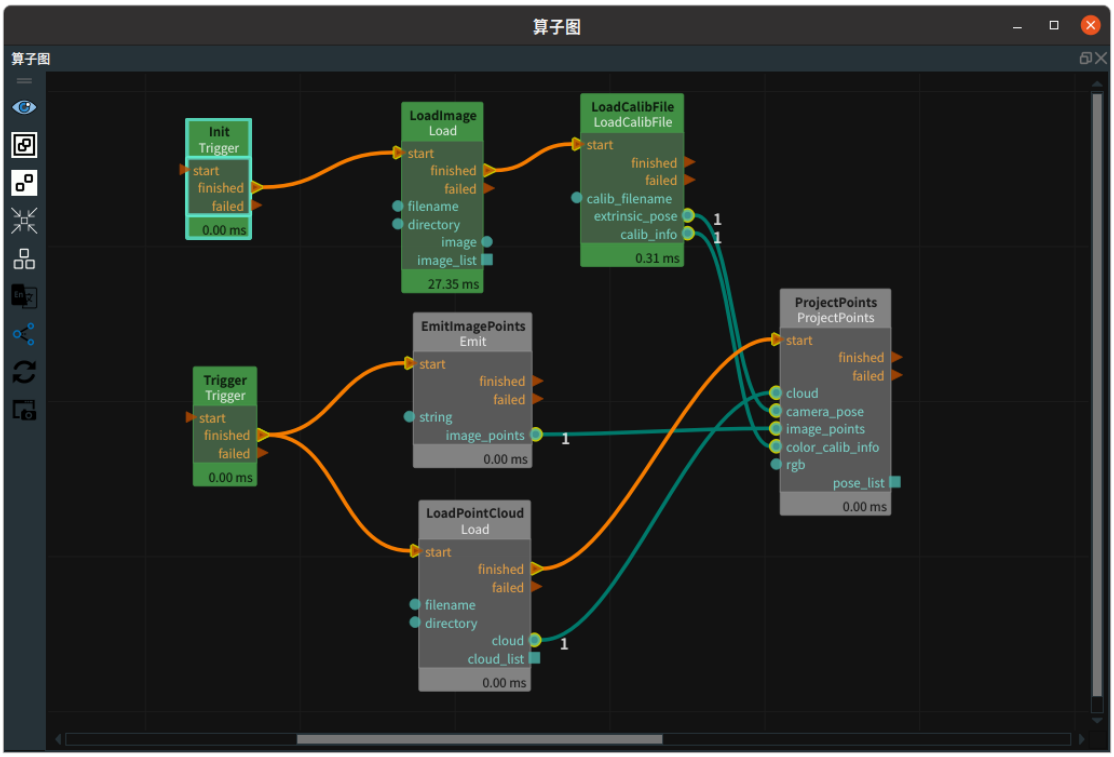
- 像素映射半径→5
- 坐标列表 → 可视

步骤3：连接算子

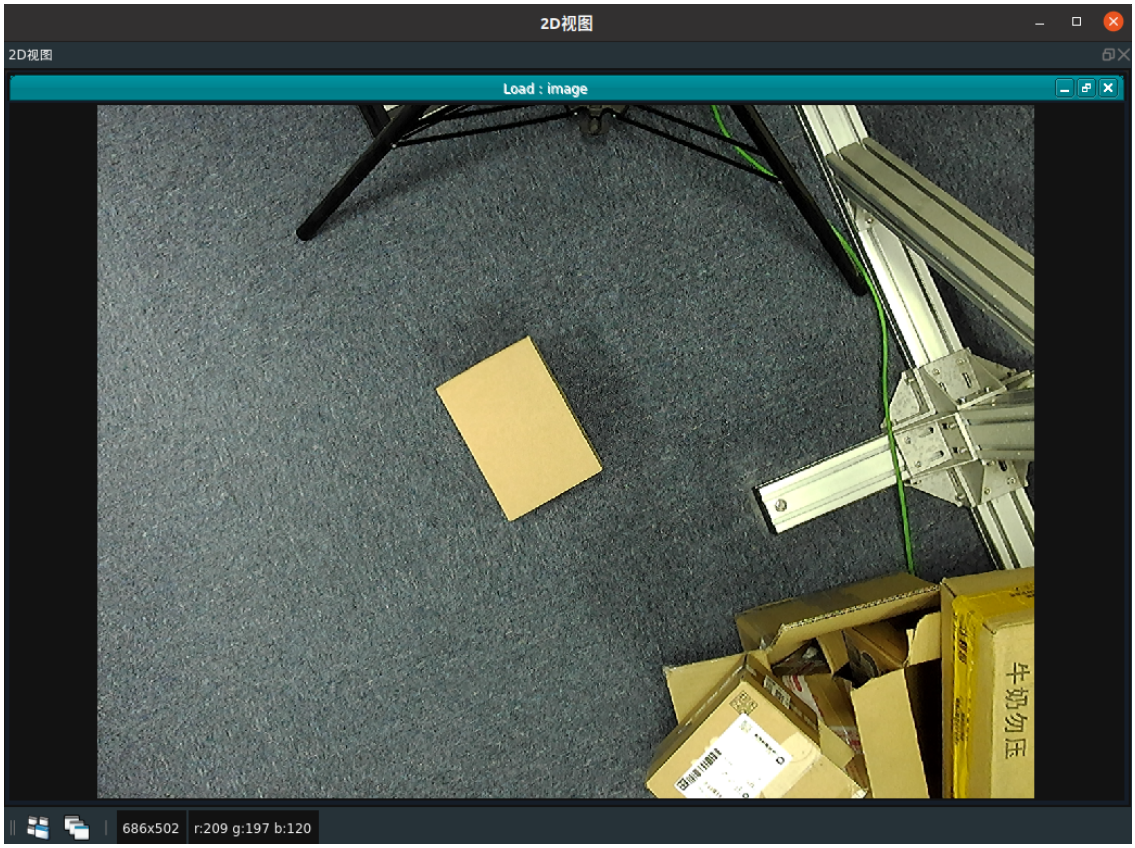


步骤4：设置 Emit 算子的 points 属性

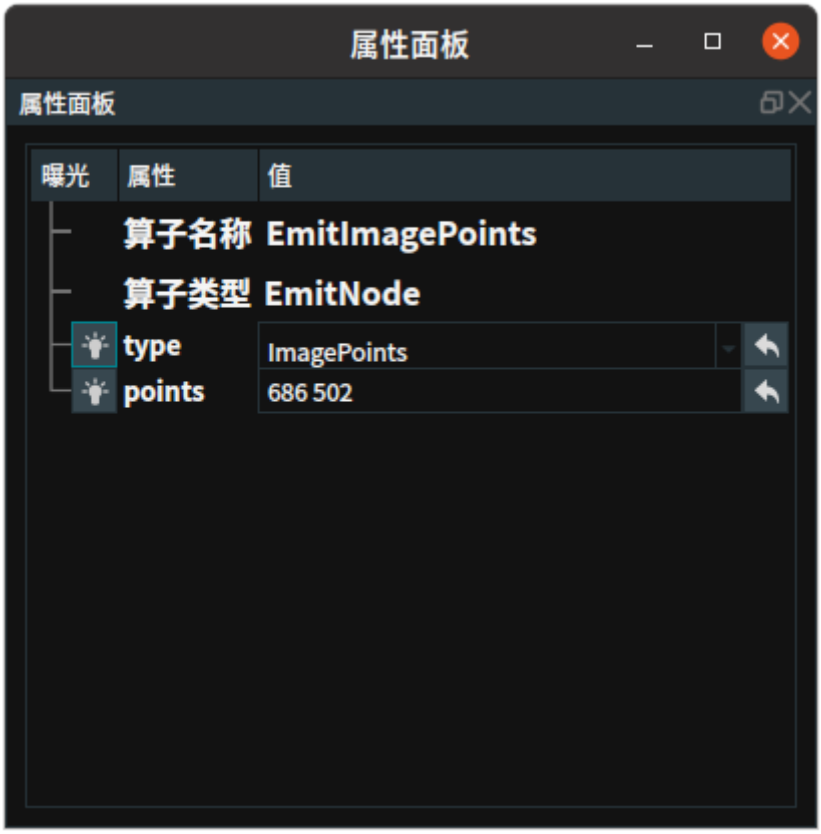
1. 点击 RVS 的运行按钮，Init (InitTrigger) 算子自动运行，并触发 Load (image) 算子。



2. 在 2D 视图查看 Load (image) 图像。将鼠标移动到目标像素位置（本案例选择下图中间盒子的右下角），此时在 2D 视图左下方实时查看鼠标光标对应位置的像素值(x,y)。

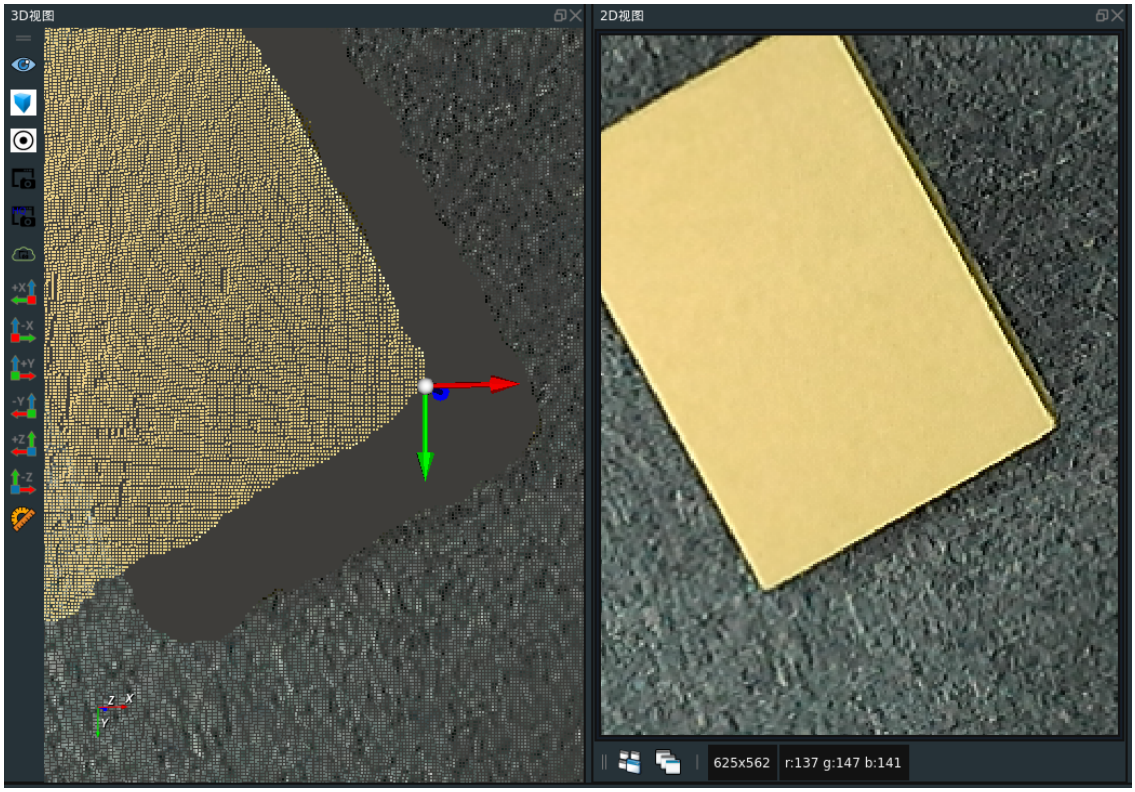


3. 将目标像素的数值填写到 Emit (ImagePoints) 算子的 points 属性中，如中间盒子的右下角像素值 686 502。



运行结果

1. 触发 Trigger 算子，运行效果如下图所示。
2. 由于盒子右下角的点云信息不全，在该点周边 5 (pixel_project_radius 设定值) 的像素半径区域内查找所有具有对应 3D 点的像素点，将其中距离该像素点最近的 3D 点作为映射结果。



Fast

算子参数

- **像素映射半径/pixel_project_radius**：由于IR镜头成像受干扰较多，所以某个 2D 像素点可能没有对应的深度值。此时算子会在该 2D 像素点所对应的深度像素点附近pixel_project_radius 的像素半径区域内查找所有具有深度值的像素点，将其中距离该像素点最近的像素点的对应深度值作为该像素点的替代深度值进行后续运算、

  说明：同上述 General 模式的区别：这里仅仅使用了 Z 数值作为替代，但是 X Y 仍然使用自身的数值。

- **相机深度图标定文件/depth_calib_filename**：图漾相机 depth 出厂标定参数文件路径，作用同输入端口的 depth_calib_info 一致。当输入端口 depth_calib_info没有连接时，则必须给该参数赋值。当输入端口 depth_calib_info 有连接时，则优先使用输入端口的数值。
- **其余参数**：同上述General模式的描述一致。

数据信号输入输出

输入：

- **depth**：
  - 数据类型：Image
  - 输入内容：掩码图的原图所对应的深度图。一般来自于 TyCameraAccess 算子的 depth 输出端口。
- **depth_calib_info**：
  - 数据类型：CalibInfo
  - 输入内容：图漾相机 depth 出厂标定参数，可以从 TyCameraAccess 算子输出，也可以由 LoadCalibFile 算子输出。
- **其余参数**：同上述 General 模式的描述一致。

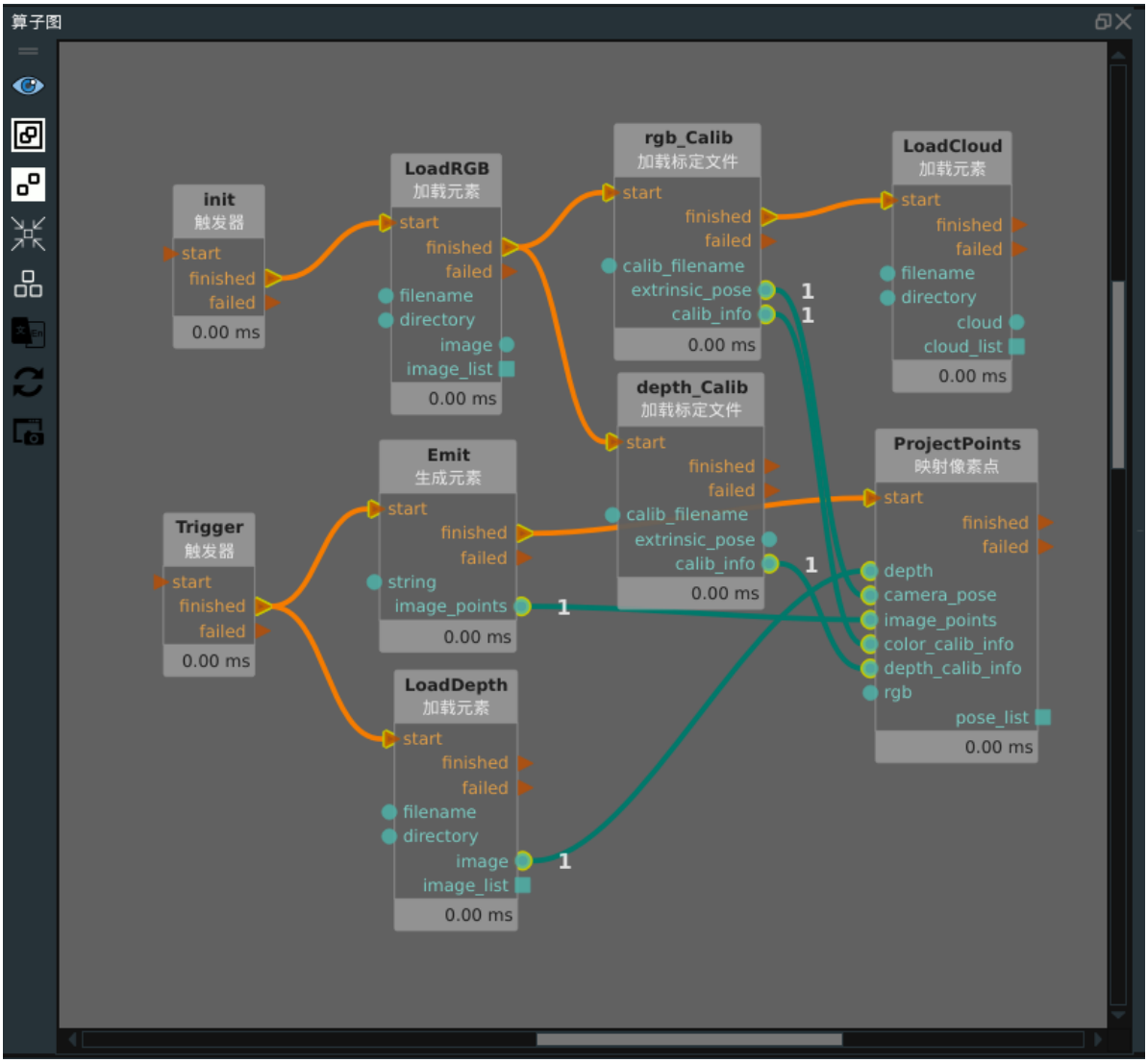
输出：

- **pose_list**：

- 数据类型：PoseList
- 输出内容：将像素点转换成 3D 点后的 Pose 列表(仅位置信息有效，姿态角度全部为 0)

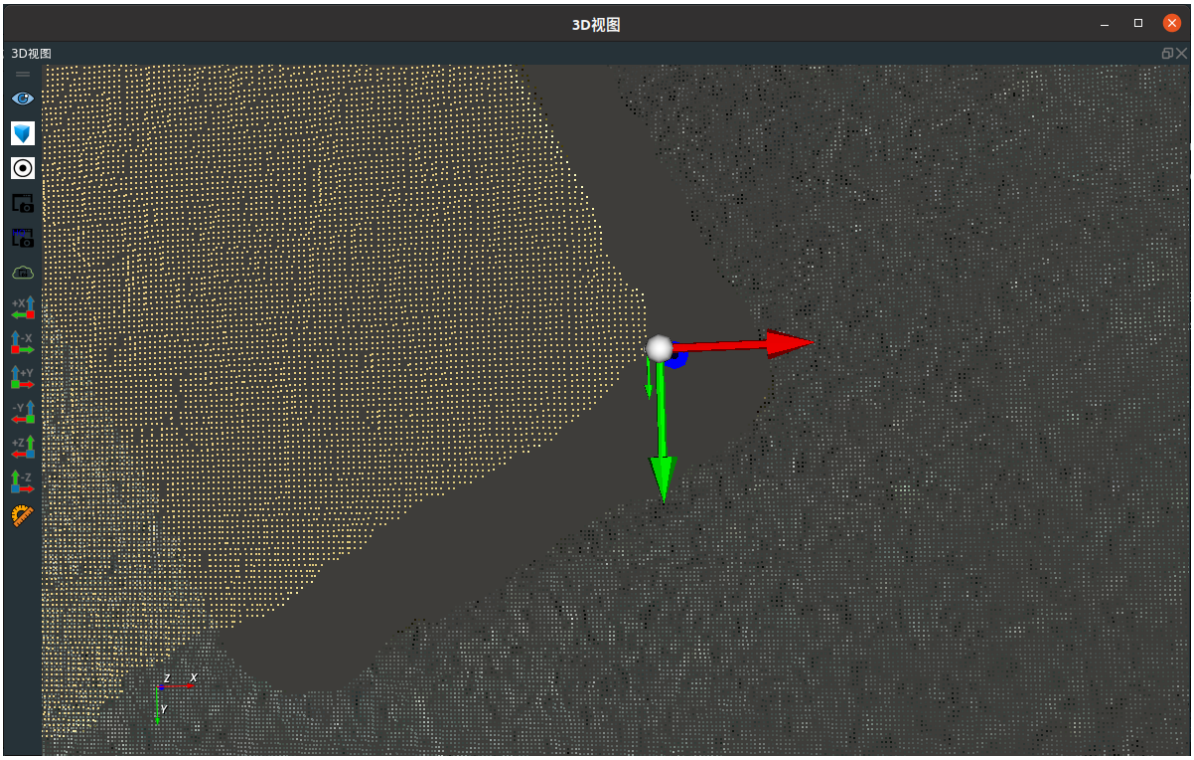
功能演示

运行步骤以及选取的 2D 像素点同上述 General 模式，连线如下。



运行结果如下。

说明：下图中较大的 Pose 是本次运行的结果，较小的 Pose 是上述 General 模式下运行的结果。通过比较两个 Pose 可以看出两种模式的区别。较大的Pose 更贴近我们选取的 2D 像素点对应的位置，但较大 Pose 本身的位置处并没有实际的成像点，该点位是我们人工生成的。而较小的 pose 对应的位置是距离较大 Pose 最近的真实成像点。


3D视图
3D视图
Z
X
Y

ProjectMask 映射图像Mask

ProjectMask 算子用于对 MaskRCNN、KeyPoint 以及 RotatedYOLO 神经网络模型推理所得的掩码图（目标 2D 像素区域）进行后续处理，将推理所得的掩码图列表转换为 3D 点云列表。

> 说明：算子运行前需准备推理得到的掩码图、图漾相机外参矩阵、目标物体3D点云。

type	功能
MaskList	将掩码图列表的所有目标同时转换成 3D 点云。支持按照类别转换。
MaskClassList	将类掩码图列表的所有目标同时转换成 3D 点云。支持按照类别转换。

MaskList

算子参数

- **相机坐标/camera_pose**：手动设置 camera_pose 的 x/y/z/roll/pitch/yaw。作用同数据信号输入端口的 camera_pose 一致。若输入端口 camera_pose 有输入，执行算子后则会自动将输入端口的值覆盖该参数的值。
- **相机彩色图标定文件/color_calib_file**：设置图漾相机 RGB 镜头的出厂标定参数文件的路径。作用同数据信号输入端口的 color_calib_info 一致。若输入端口 color_calib_info 有输入，优先使用输入端口的值。
- **使用畸变参数/use_dist_coeff**：是否使用 rgb 图的畸变校正参数。由于我们往往都在图漾机资源算子 (TyCameraResource) 的参数中默认勾选了 false（去畸变），所以我们使用的rgb图都是已经去除畸变的图，所以这里保持默认的不勾选状态即可。
- **选择类别/select_class**：如果仅需要将原始掩码图列表中的某一类目标进行 3D 转换，则这里填写对应的类别即可。该参数默认为空，即将所有类别全部转换。
- **点云列表/cloud_list**：设置点云列表在 3D 视图中的可视化属性。
  -  打开点云列表可视化。
  -  关闭点云列表可视化。
  -  设置3D视图中点云列表的颜色。取值范围：[-2,360]。默认值：-1。
  -  设置点云列表中点的尺寸。取值范围：[1,50]。默认值：1。

数据信号输入输出

输入：

- **cloud**：
  - 数据类型：PointCloud
  - 输入内容：掩码图的原图所对应的点云

  说明：当输入的点云中背景点云占比较多时，建议将大部分背景点云裁剪后再输入。
- **camera_pose**：
  - 数据类型：Pose
  - 输入内容：掩码图的原图所在的 RGB 空间坐标系到上述点云所在的空间坐标系的转换关系。如果上述输入端口的点云位于相机3D坐标系，则这里输入RGB空间坐标系到相机3D坐标系的转换 Pose 即可。如果上述输入端口的点云位于机器人坐标系，则这里需要输入 RGB空间坐标系到机器人3D坐标系的的转换 Pose 。

- `color_calib_info`：
  - 数据类型：CalibInfo
  - 输入内容：图漾相机 RGB 镜头的出厂标定参数，可以从 TyCameraAccess 算子输出，也可以由 LoadCalibFile 算子输出。
- `mask_list`：
  - 数据类型：ImageList
  - 输入内容：神经网络推理算子所得的掩码图像列表
- `mask_name_list`：
  - 数据类型：StringList
  - 输入内容：神经网络推理算子所得的目标类别列表

输出：

- `cloud_list`：
  - 数据类型：PointCloudList
  - 输出内容：将掩码图列表中的目标转换到 3D 坐标系后的点云列表

功能演示

使用 ProjectMask 算子的MaskList模式 ，将掩码图列表的所有目标同时转换成 3D 点云，输出 PointCloudList 。

步骤1：算子准备

添加 Trigger(3 个）、Load（ 2 个）、LoadCalibFile、AIDetectGPU、ProjectMask 算子至算子图。

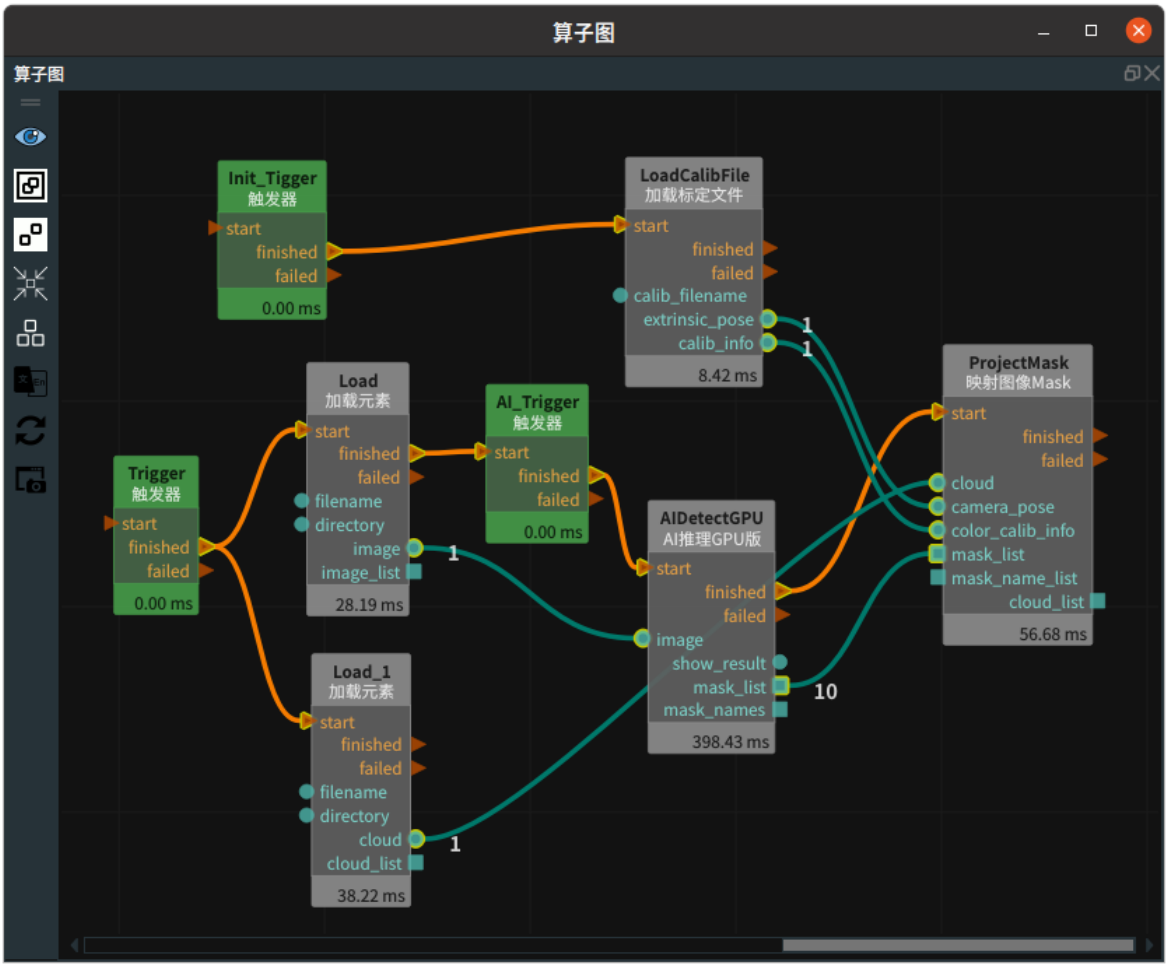
步骤2：设置算子参数

1. 设置 Trigger_1 算子参数：
   - 算子名称→Init_Trigger
   - 类型 → InitTrigger
2. 设置 Trigger_2 算子参数：
   - 算子名称→AI_Trigger
   - 类型 → InitTrigger
3. 设置 Load 算子参数：
   - 类型 → Image
   - 文件 → ••• →选择图像文件名 （ *example_data/mask_data_train/20221010113840917/rgb.png*)
   - 图像 → 👁可视
4. 设置 Load_1 算子参数：
   - 类型 → PointCloud
   - 文件 → ••• → 选择与图像匹配点云文件名 （ *example_data/mask_data_train/20221010113840917/cloud.pcd*)
   - 点云 → 👁可视
5. 设置 LoadCalibFile 算子参数：
   - 相机标定文件 → 图漾相机出厂标定文件 (*example_data/mask_data_train/mask_ty_color_calib.txt*)
6. 设置 AIDetectGPU 算子参数：
   - 说明：该端口连接/不连接，输出所有类别的旋转矩阵。连接时，可填写 select_class 来输出指定类别的旋转矩形。

- 类名文件路径 → ••• → 选择相应文件名（*example_data/mask_data_train/fruits.txt*）
- 权重文件路径 → ••• → 选择相应权重文件名（*example_data/mask_data_train/train_output/model_final.pth*）
- 配置文件路径 → ••• → 选择相应权重文件名（*example_data/mask_data_train/train_output/config.yaml*）
- 物体得分阈值 → ••• → 0.75
- 识别结果图像 → 可视

7. 设置 ProjectMask 算子参数：
   - 类型→ MaskList
   - 点云列表 → 可视

步骤3：连接算子



步骤4：运行

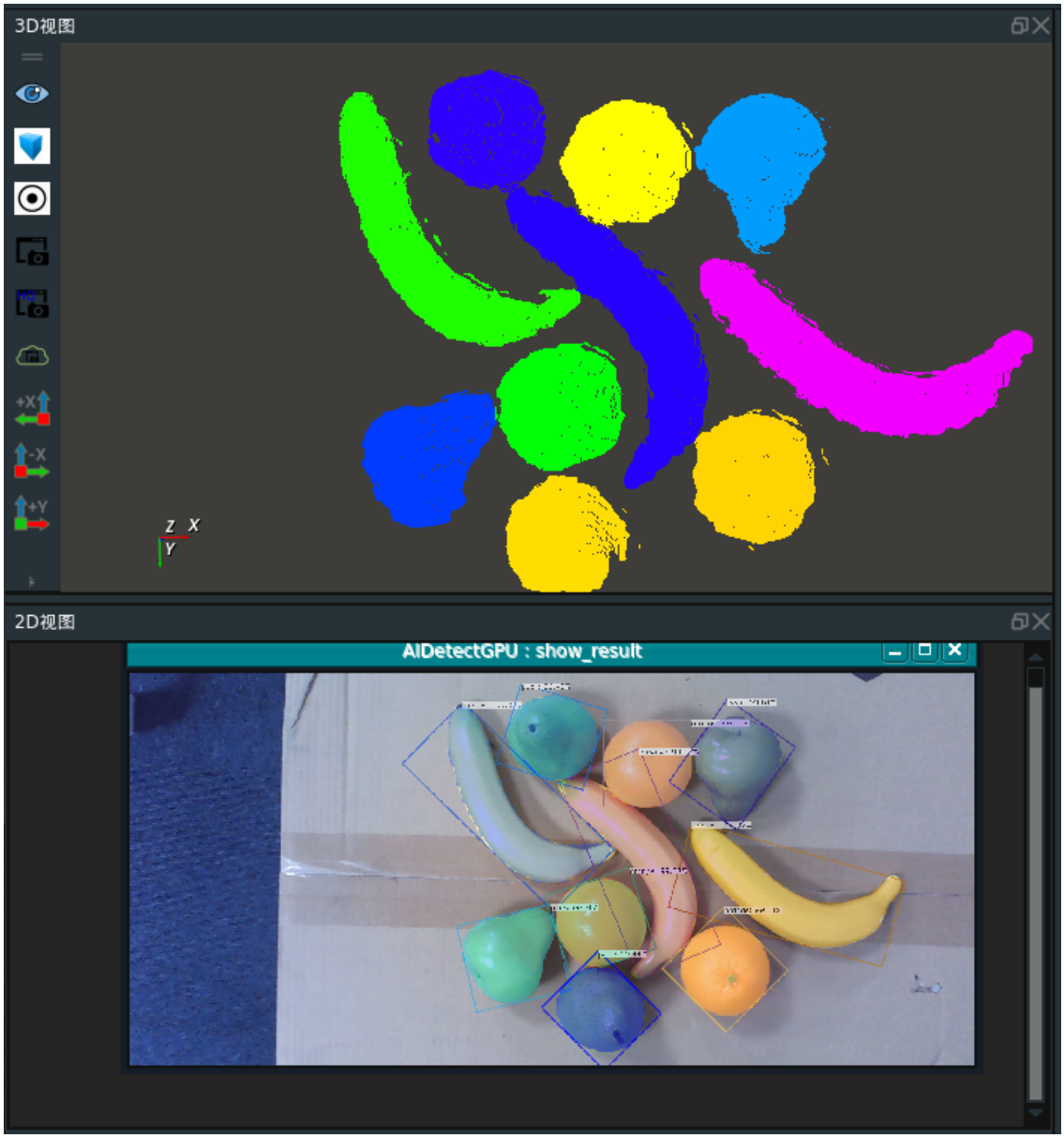
1. 点击RVS的运行按钮，InitTrigger 和AI_Trigger 会自动运行，分别触发 LoadCalibFile 算子以及 AIDetectGPU 。然而，AIDetectGPU 算子的初始化同时会通过finished 端口触发后续的 ProjectMask 算子。由于 ProjectMask 算子的输入端口没有有效数值，因此会报告一个端口输入无效的警告。这个警告不会影响后续的运行。如下图所示。

2023-02-08 14:51:38.232589	rvs_python	info	AIDetect is Loading module
2023-02-08 14:51:39.761171	rvs_python	info	module loaded : MaskRCNN
2023-02-08 14:51:42.662675	rvs_python	info	AIDetect initialised
2023-02-08 14:51:42.662763	rvs_python	info	GetClassNames done
2023-02-08 14:51:42.662812	rvs_kernel	debug	ProjectMask: input_trigger:start
2023-02-08 14:51:42.662837	rvs_calibration	warning	ProjectMask: no input cloud is given

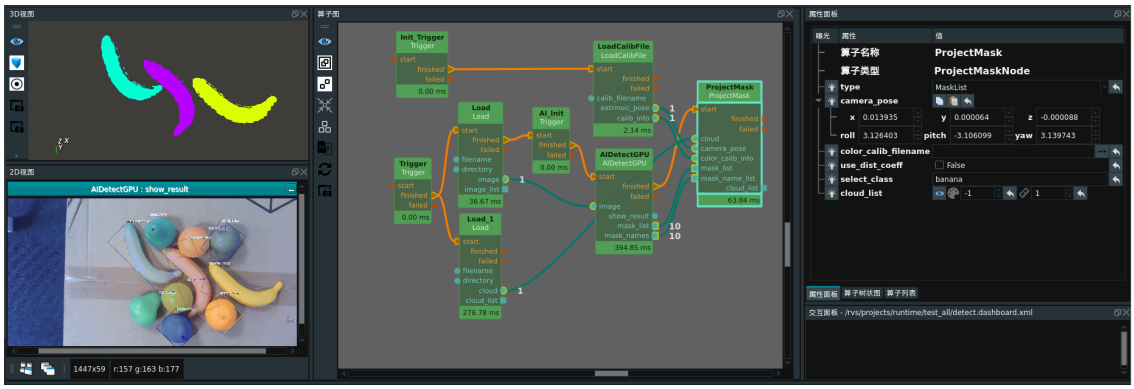
2. 正常运行触发左侧 Trigger 算子。

运行结果

1. 如果勾选了AIDetectGPU算子的 **识别结果图像** 参数以及 ProjectMask 算子的 **点云列表** 属性的可视化选项，则会显示一个包含点云和掩膜的三维可视化结果。其中，点云的颜色表示掩膜的值，即蓝色表示掩膜值为 0，白色表示掩膜值为 1。掩膜将点云分割成不同的区域，以便进行后续的处理和分析。如下图所示。



2. 如果在 ProjectMask 算子的 **选择类别** 属性中填写了某个具体的类别，比如 banana ，则还额外需要将 AIDetectGPU 算子的 mask_names 端口连接到ProjectMask 算子的 mask_name_list 端口，运行效果如下所示。



MaskClassList

将 ProjectMask 算子的 **类型** 属性选择 MaskClassList ，将类掩码图转换成 3D 点云列表并输出。

算子参数

- **所有参数**：同上述 MaskList 。

数据信号输入输出

输入：

- **mask_class_list**：
  - 数据类型：Image
  - 输入内容：神经网络推理算子在 Class 模式下 (MaskRCNNClass 以及 YOLOClass) 所得的类掩码图列表
- **其余端口**：同上述MaskList 描述一致

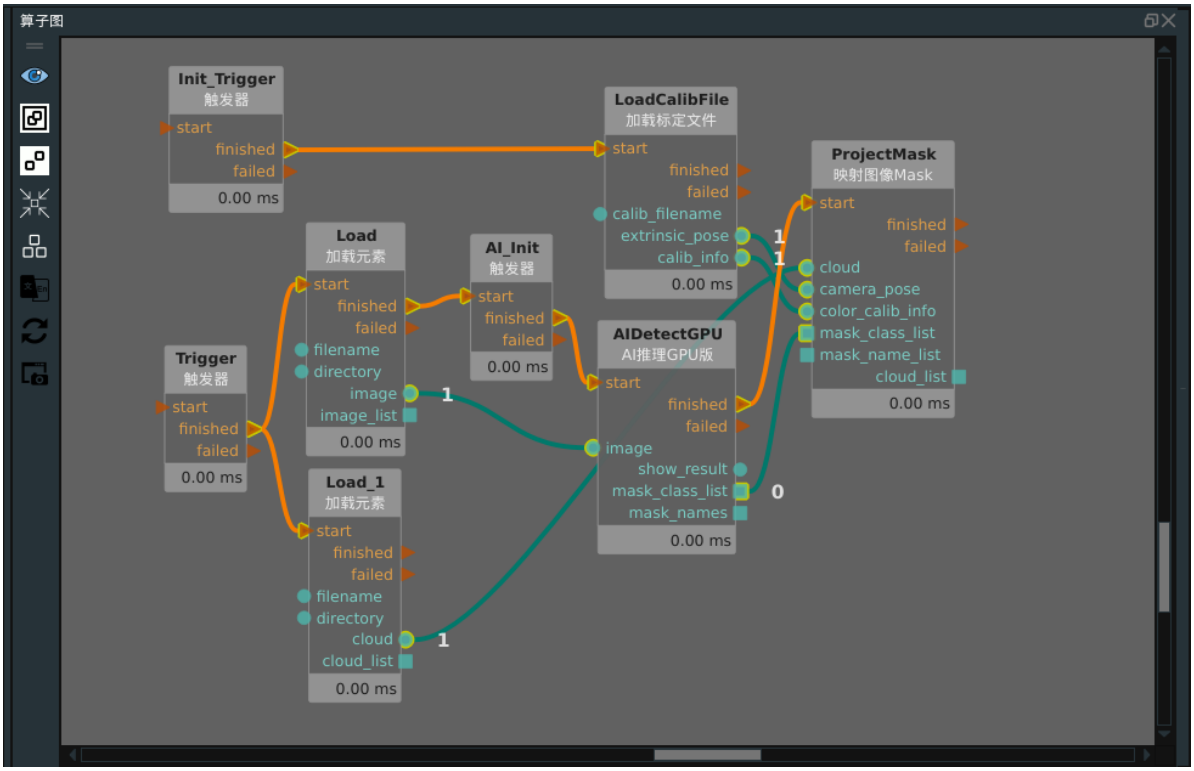
输出：

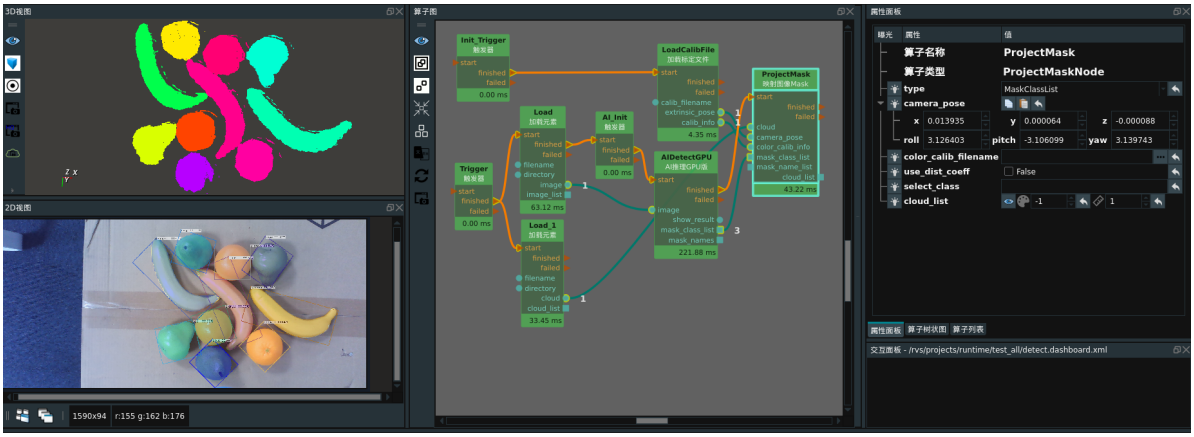
- **cloud_list**：
  - 数据类型：PointCloud
  - 输出内容：将类掩码图中的目标转换到 3D 坐标系中的点云

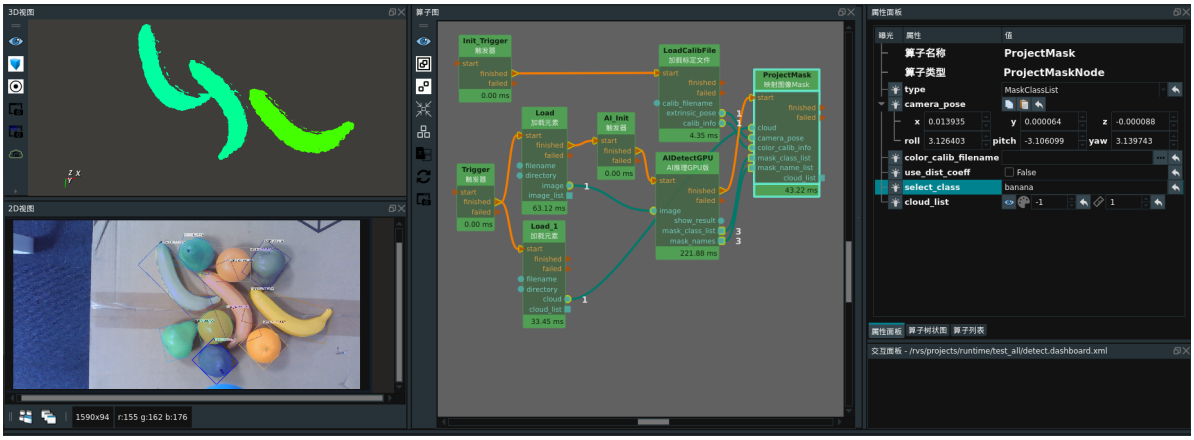
功能演示

使用 ProjectMask 算子中 MaskClassList 模式，分别将所有类掩码图以及单个类掩码图 (类别仍然选为 banana) 的目标转换成 3D 点云。

算子连接参照 MaskList ，连接细节以及两次运行结果图如下所示。




3D视图
2D视图
算子图
属性面板
Init_Trigger
触发器
LoadCalibFile
加载标定文件
ProjectMask
映射图像Mask
Load
加载元素
AI_Init
触发器
AIDetectGPU
AI推理GPU版
Trigger
触发器
Load_1
加载元素
算子名称
ProjectMask
算子类型
ProjectMaskNode
type
MaskClassList
camera_pose
x 0.013935
y 0.000064
z -0.000088
roll 3.126403
pitch -3.106099
yaw 3.139743
color_calib_filename
use_dist_coeff
False
select_class
cloud_list
交互面板 - /rvs/projects/runtime/test_all/detect.dashboard.xml


3D视图
2D视图
算子图
属性面板
Init_Trigger
触发器
LoadCalibFile
加载标定文件
ProjectMask
映射图像Mask
Load
加载元素
AI_Init
触发器
AIDetectGPU
AI推理GPU版
Trigger
触发器
Load_1
加载元素
算子名称
ProjectMask
算子类型
ProjectMaskNode
type
MaskClassList
camera_pose
x 0.013935
y 0.000064
z -0.000088
roll 3.126403
pitch -3.106099
yaw 3.139743
color_calib_filename
use_dist_coeff
False
select_class
banana
cloud_list
交互面板 - /rvs/projects/runtime/test_all/detect.dashboard.xml

communication

TCPServerResource TCP服务器资源

TCPServerResource 算子属于资源类线程算子，用于在 RVS 的分线程中启动一个 TCP server 服务器。

客户端可以通过该算子对 RVS 中的数据进行读写。通信数据是以字符串形式表达的 json 格式。

1. 客户端发送更改算子参数的消息。发送字符串格式为:

   ```
   {"SetPara":[

   {"node_name":"算子名称","para_name":"要更改的参数名称","para_value":"要更改的值"},

   ...

   ]}+delimiter(消息结束符)
   ```

   如果发送消息的格式不合法，则本算子会回复：

   ```
   ParseJson;0
   ```

   如果参数设置失败，则本算子会回复 ：

   ```
   SetPara;0
   ```

   如果参数设置成功，则本算子会回复：

   ```
   SetPara;1
   ```

2. 客户端想读取算子的输出端口的值。发送字符串格式为：

   ```
   {"GetOutput":[

   {"node_name":"算子名称","index":int_value,"type":"算子类型"},

   ...

   ]}+delimiter(消息结束符)
   ```

   > 注意：
   >
   > 1. 上文中的 index 表示某个算子的所有数据输出端口从上往下(从0开始)排列的端口序号。排序时不包含 finished 和 failed 等控制端口。
   > 2. 当前的 GetOutput 仅支持 String 和 Pose 两种算子类型，也可以支持 StringList 以及 PoseList ，但是这里的 type 统一填写成 String 以及 Pose 。
   > 3. 同时获取多个输出端口的回复值时，回复内容通过分号隔开。

   如果发送消息的格式不合法，则本算子会回复：

```
ParseJson;0
```

如果数据读取失败(找不到对应算子，或找不到算子对应端口，或算子对应端口属性不匹配)，则本算子会回复：

```
GetOutput;0
```

如果参数设置成功，则本算子会回复对应的读取内容。

算子参数

- `auto_start`：自动运行。如果勾选该选项，则 RVS 第一次运行后，会自动触发该算子执行。算子运行后，开始创建 TCP 服务端，创建成功后监听服务端口并等待客户端访问。
- `start`：勾选后，开始触发算子执行。算子运行后，开始创建 TCP 服务端，创建成功后监听服务端口并等待客户端访问。
- `stop`：勾选后，开始停止 TCP 服务端的监听。
- `reset`：在该资源算子已经运行后，如果重新更改了 port 、connections 、server_mode 、delimiter 4个参数中的任意一个之后，若需要生效，都必须勾选该reset 选项进行资源重置，然后重新勾选 start 运行。
- `port`：创建TCP server服务器所使用的本机服务端口。

  > 说明：实际运行时，如果提示 TCP 服务端创建失败，往往是所选取的端口已经被其他服务所占用，更换端口即可。如果在同一个 RVS 软件中，创建多个TCP 服务端算子，彼此的 port 也要互斥。

- `connections`：可同时支持的最大客户端连接数量。
- `server_mode`：运行模式。
  - Once：表示完成一次 TCP 对话以后自动断开同客户端的链接(比如客户端首先同该算子建立了 TCP 连接，然后第一次发送了消息请求之后，继续发送就会报错)。
  - Continous：表示客户端建立连接后可以无限次数的对话。
- `delimiter`：消息结束符，包含RT 换行符、“；”、“#”、“$”。当选择了其中某一种时，另外三种就会被视作普通字符随意使用 。服务端在接收客户端发送的消息时，会捕获第一个消息结束符之前的所有消息。

  > 说明：由于不同的通信软件、电脑系统软件、机器人操作系统软件对回车换行的定义不一致，容易导致信息发送或者接收失败，所以一般不建议选择RT作为消息结束符。

控制信号输入输出

> 说明：Resource 资源类算子的控制信号端口无法触发使用。

功能演示

下面分别针对读写2类通讯信号，进行通讯演示。

在选用演示所用的 TCP 客户端时，由于 RVS 自带的 CommonTCPClient 算子在使用时要求服务端返回的信息必须以四类固定字符结尾，而 TCPServerResource 算子在作为 TCP 的服务端时，返回给 TCP Client 的信息是固定的并且没有额外添加上述四类固定结尾字符，所以这里不再选用 CommonTCPClient 算子作为客户端。

> 说明：这里的演示案例是基于 ubuntu 系统给出的，使用了 "echo + nc " 的指令方式实现 TCP 客户端的功能；如果是在 Windows 版本，建议使用通讯助手等工具。

步骤1：算子准备

添加 Trigger 、Emit 、Concatenate 、TCPServerResource 算子至算子图。

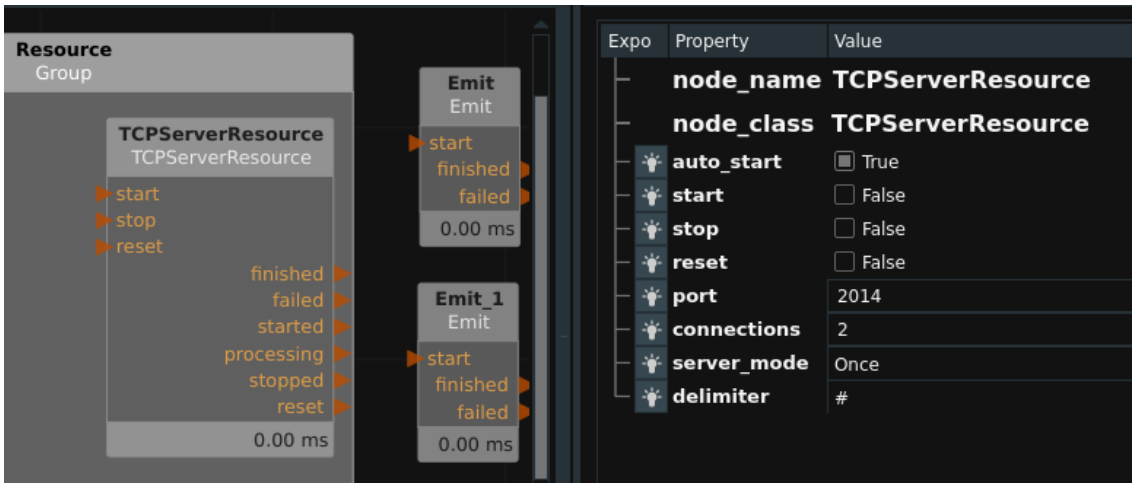
步骤2：设置算子参数

设置 TCPServerResource 算子参数：

- auto_start → True
- delimiter → #
- server_mode → Once

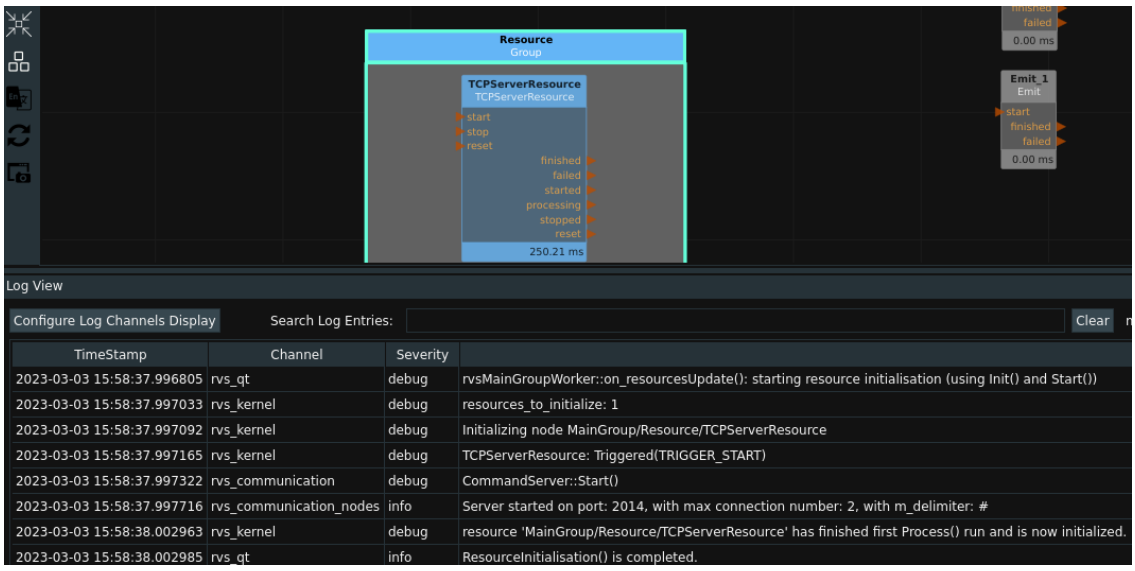
步骤3：连接算子

1. 打开 RVS 软件界面，在算子图拖入 TCPServerResource 算子以及拖入两个 Emit 算子如下图所示。TCPServerResource 选用了 “echo + nc” 的指令方式，所以这里每发送-接收一次消息后都要断开连接，所以选择了 Once 模式。



步骤4：运行及运行结果

1. 打开 RVS 的运行按钮，TCPServerResource 算子会自动触发，并变为蓝色，日志栏会同时打印算子运行说明如下图所示，表示 TCP 的服务端已经建立完毕。



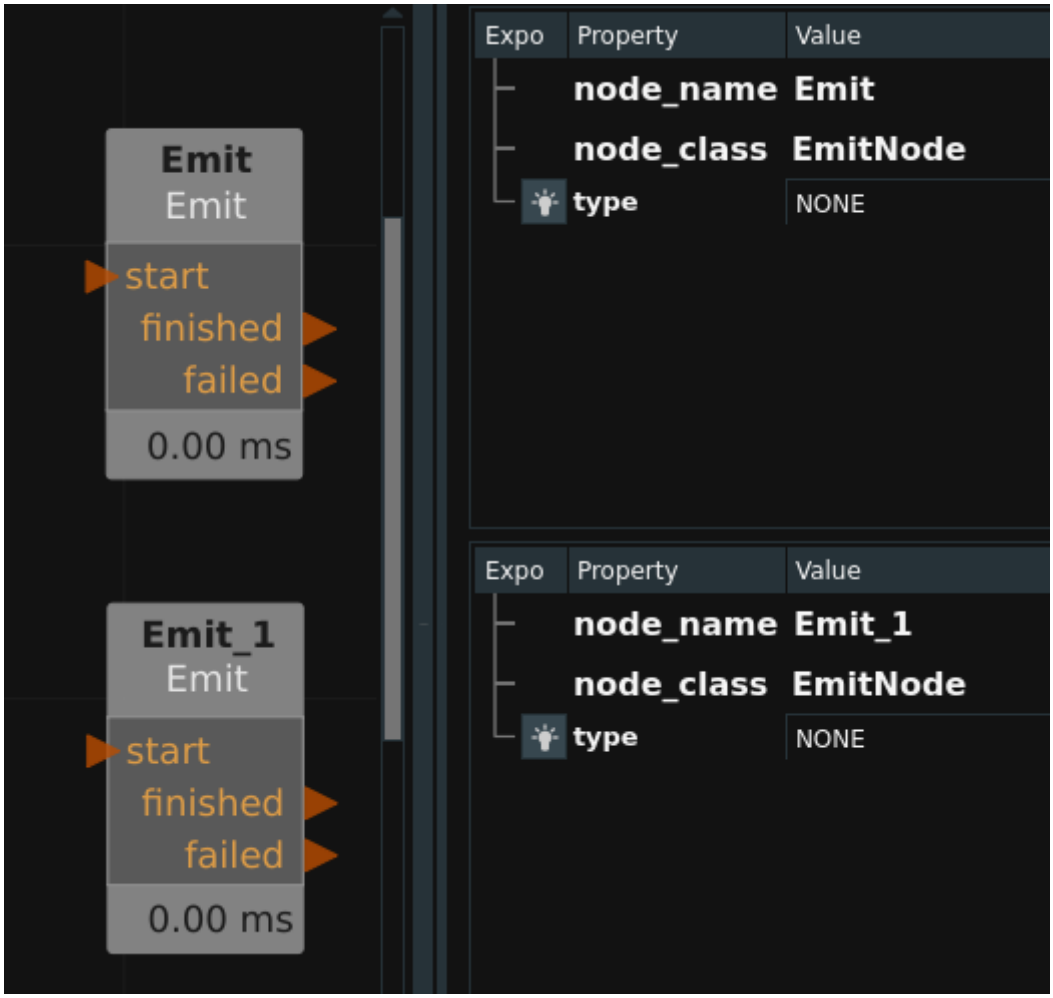
2. 重新打开一个终端，输入命令：

```
echo "{\"SetPara\":[{\"node_name\":\"Emit\",\"para_name\":\"type\",\"para_value\":\"String\"},
{\"node_name\":\"Emit_1\",\"para_name\":\"type\",\"para_value\":\"Pose\"}]}#" | nc localhost 2014
```

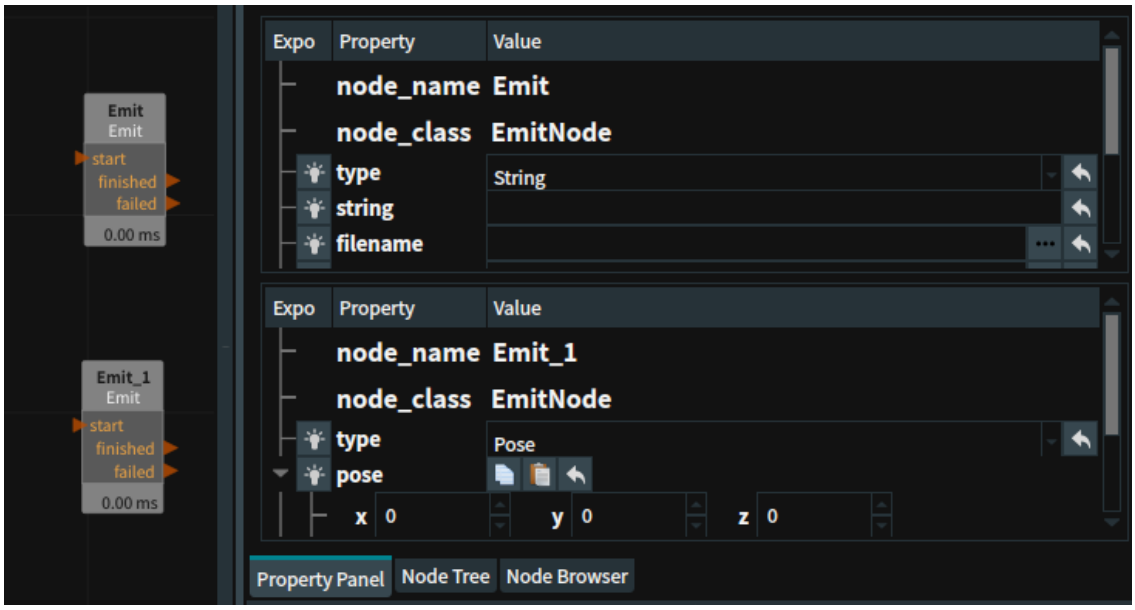
并单击回车键，运行结束后，我们会在这个终端窗口看到回复字符串如下图所示。

```
wjl@ferrari:/work/exec_shell$ echo "{\"SetPara\":[{\"node_name\":\"Emit\",
\"para_name\":\"type\",\"para_value\":\"String\"},{\"node_name\":\"Emit_1\
",\"para_name\":\"type\",\"para_value\":\"Pose\"}]}#" | nc localhost 2014
SetPara;1wjl@ferrari:/work/exec_shell$
```

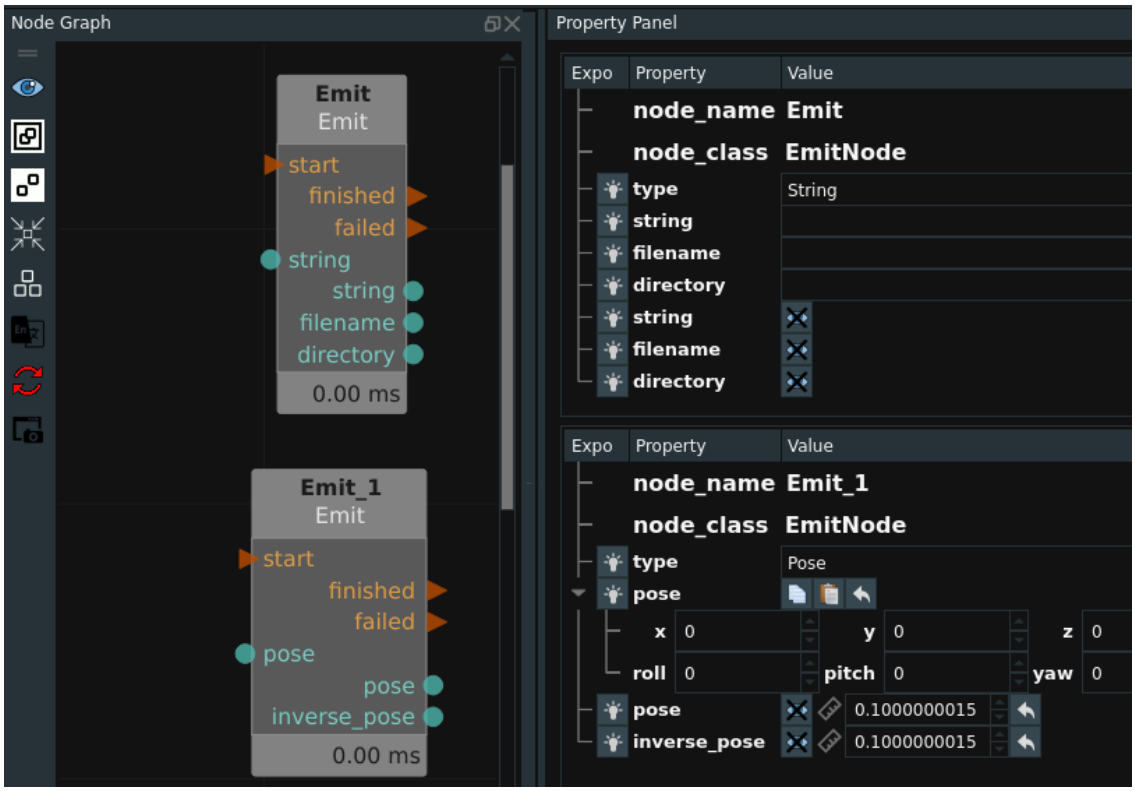
3. 但是 RVS 执行结束后，对应的两个 Emit 算子属性以及端口都没有进行更改如下图所示。



4. 此时需要我们手动分别单击一下两个 Emit 算子，然后发现其属性栏已经更新，更新后如下图所示。



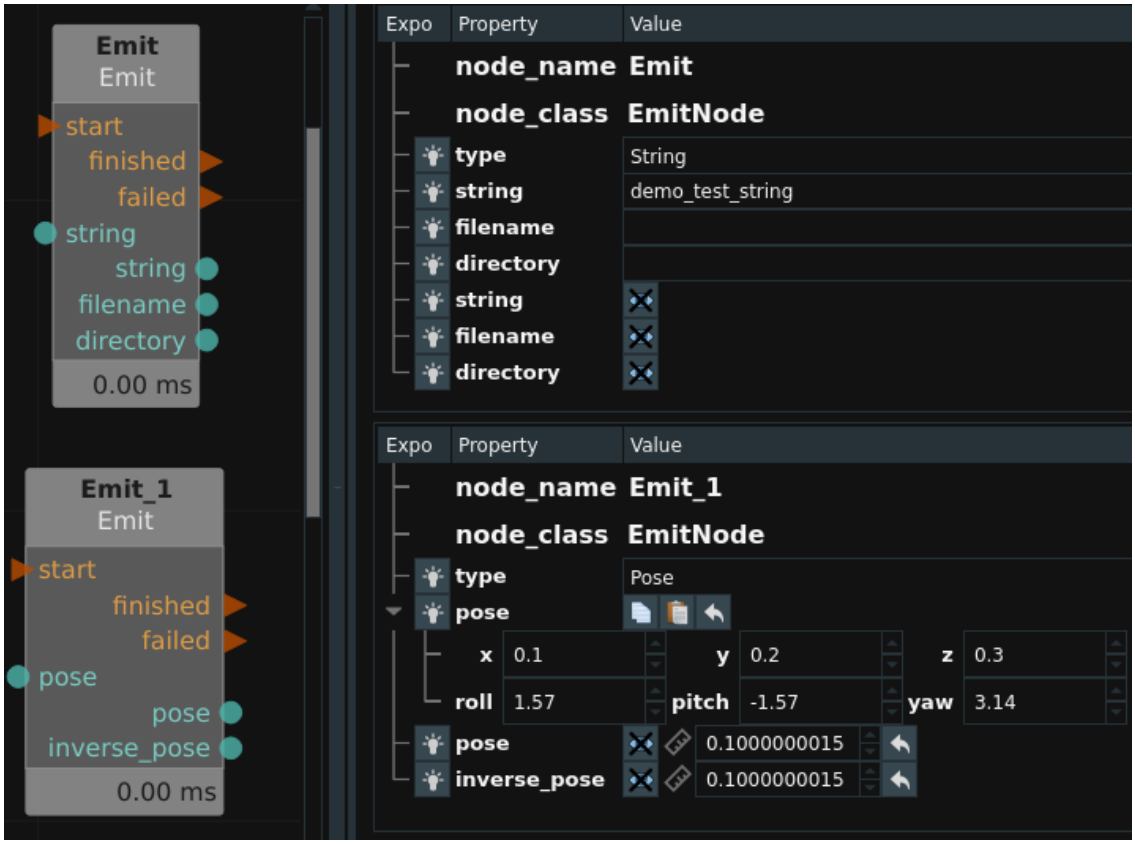
5. 然而两个算子的输入输出端口没有更新，则我们需要单击算子图左侧的更新按钮（下图左侧红色标注的按钮）。更新后如下图所示。



6. 在终端中，继续输入命令echo "{"SetPara":
[{"node_name":"Emit","para_name":"string","para_value":"demo_test_string"},
{"node_name":"Emit_1","para_name":"pose_x","para_value":"0.1"},
{"node_name":"Emit_1","para_name":"pose_y","para_value":"0.2"},
{"node_name":"Emit_1","para_name":"pose_z","para_value":"0.3"},
{"node_name":"Emit_1","para_name":"pose_roll","para_value":"1.57"},
{"node_name":"Emit_1","para_name":"pose_pitch","para_value":"-1.57"},
{"node_name":"Emit_1","para_name":"pose_yaw","para_value":"3.14"}]}#" | nc localhost 2014
并单击回车键，运行结束后，我们会在这个终端窗口看到回复字符串如下图所示。

```
wjl@ferrari:/work/exec_shell$ echo "{\"SetPara\":[{\"node_name\":\"Emit\",
\"para_name\":\"string\",\"para_value\":\"demo_test_string\"},{\"node_name
\":\"Emit_1\",\"para_name\":\"pose_x\",\"para_value\":\"0.1\"},{\"node_nam
e\":\"Emit_1\",\"para_name\":\"pose_y\",\"para_value\":\"0.2\"},{\"node_na
me\":\"Emit_1\",\"para_name\":\"pose_z\",\"para_value\":\"0.3\"},{\"node_n
ame\":\"Emit_1\",\"para_name\":\"pose_roll\",\"para_value\":\"1.57\"},{\"n
ode_name\":\"Emit_1\",\"para_name\":\"pose_pitch\",\"para_value\":\"-1.57\
"},{\"node_name\":\"Emit_1\",\"para_name\":\"pose_yaw\",\"para_value\":\"3
.14\"}]}#" | nc localhost 2014
\SetPara;1wjl@ferrari:/work/exec_shell$ \
```

7. 然而两个算子的对应属性值不会立刻更新，需要我们手动各自单击一下两个 Emit 算子。更改后的属性显示如下。



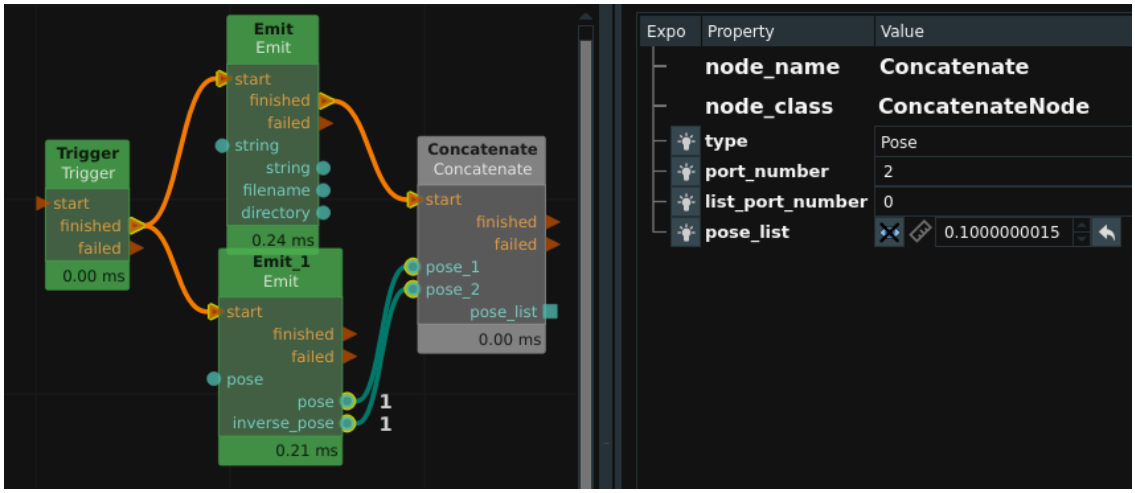
8. 在终端中，继续输入 echo "{"GetOutput":[{"node_name":"Emit","index":0,"type":"String"},{"node_name":"Emit_1","index":0,"type":"Pose"}]}#" | nc localhost 2014 命令并单击回车键，运行结束后，我们会在这个终端窗口看到回复字符串如下图所示。

```
wjl@ferrari:/work/exec_shell$ echo "{\"GetOutput\":[{\"node_name\":\"Emit\",\"index\":0,\"type\":\"String\"},{\"node_
name\":\"Emit_1\",\"index\":0,\"type\":\"Pose\"}]}#" | nc localhost 2014
GetOutput;1;;0 0 0 0 0 0wjl@ferrari:/work/exec_shell$
```

9. 发现得到的回复值并非刚才设置的数值而是这些参数的默认值 (string 默认为空字符，pose默认为全零)。这是因为两个Emit算子尚未被触发，其数据输出端口尚未更新。此时单击 RVS 的 stop，将一个 Trigger 算子拖入到算子图中，连接触发两个 Emit 算子，然后重新单击 RVS 的运行，并触发执行 Trigger。此时重新发送上述命令，可以得到回复如下：

```
wjl@ferrari:/work/exec_shell$ echo "{\"GetOutput\":[{\"node_name\":\"Emit\",\"index\":0,\"type\":\"String\"},{\"node_
name\":\"Emit_1\",\"index\":0,\"type\":\"Pose\"}]}#" | nc localhost 2014
GetOutput;1;demo_test_string;0.1 0.2 0.3 1.57 -1.57 3.14wjl@ferrari:/work/exec_shell$
```

10. 之后单击 RVS 的 stop，将一个 Concatenate 算子拖入到算子图中，更改其 type 类型为 pose ，更改其 port_number 为 2 ，更改 list_port_number 为 0，并按照下图连接算子。



11. 重新单击 RVS 的运行，并触发执行 Trigger，此时 Concatenate 算子的输出端口会获得实际数值。
12. 然后在终端中，继续输入命令 echo "{"GetOutput":[{"node_name":"Concatenate","index":0,"type":"Pose"}]}#" | nc localhost 2014 并单击回车键，运行结束后，我们会在这个终端窗口看到回复字符串如下图所示。

> 注意，对于List的回复，最后会附加List中含有的目标总数，如下图的 2 。

```
wjl@ferrari:/work/exec_shell$ echo "{\"GetOutput\":[{\"node_name\":\"Concatenate\",\"index\":0,\"type\":\"Pose\"}]}#"
 | nc localhost 2014
GetOutput;1;0.1 0.2 0.3 1.57 -1.57 3.14;-0.2999205589 -0.09976076335 -0.200238511 1.570795655 -0.002388599329 1.57159
245;2wjl@ferrari:/work/exec_shell$
```

CommonTCPServer 通用TCP服务器

CommonTCPServer 算子属于线程类算子，用于在 RVS 的分线程中启动一个 TCP server 服务器。

该服务器只能接收固定的7类字符串信息，并且对应的回复也是固定的字符串。

这 7 类字符串分别以 7 个命令作为起始值，分别是 CAPTURE、SET_POSE、GET_POSE、SET_JOINT、GET_JOINT、SET_STRING、GET_STRING。

1. 客户端发送以命令字符 SET_POSE 、SET_JOINT、SET_STRING 开始的消息。

   通讯方式：

   ```
   SET_POSE command_delimiter  x y z rx ry rz delimiter
   ```

   ```
   SET_JOINT command_delimiter  j1 j2 j3 j4 j5 j6 delimiter
   ```

   ```
   SET_STRING command_delimiter  string delimiter
   ```

   从客户端获得的实际数据最终分别触发算子右下侧的 pose 、joints、string 端口。

   运行结束后，最终都会回复客户端空字符。

2. 客户端发送以命令字符 GET_POSE 、GET_JOINT、GET_STRING 开始的消息。

   通讯方式：

   ```
   GET_POSE  command_delimiter delimiter
   ```

   ```
   GET_JOINT  command_delimiter delimiter
   ```

   ```
   GET_STRING  command_delimiter delimiter
   ```

   最后需要返回给客户端的实际数据分别通过算子左下侧的 pose 、joints、string 端口传递数据。

   运行结束后，最终都会回复客户端对应的实际数值字符串，中间用空格隔开。

3. 客户端发送以命令字符 “CAPTURE” 开始的消息。

   通讯方式：

   ```
   CAPTURE  command_delimiter  其他内容 delimiter
   ```

   运行结束后，最终都会回复客户端空字符。

4. 任意其他消息，不会作任何响应。

算子参数

- **端口/port**：创建 TCP server 服务器所使用的本机服务端口。

  说明：实际运行时，如果提示 TCP 服务端创建失败，往往是所选取的端口已经被其他服务所占用，更换端口即可。如果在同一个 RVS 软件中，创建多个TCP 服务端算子，彼此的 port 也要互斥。

- **连接数量/connections**：可支持同时连接的客户端数量。

- **服务器模式/server_mode**：运行模式。
  - Once：表示完成一次 TCP 对话以后自动断开同客户端的连接 。(比如客户端首先同该算子建立了 TCP 连接，然后第一次发送了 test1 命令的字符串之后，继续发送 test2 命令的字符串，此时就会报错)。
  - Continous：表示客户端建立链接后可以无限次数的对话。
- **分隔符/delimiter**：消息结束符，包含RT 换行符、 “；”、“#”、“$”。当选择了其中某一种时，另外三种就会被视作普通字符随意使用 。服务端在接收客户端发送的消息时，会捕获第一个消息结束符之前的所有消息。

  注意：由于不同的通信软件、电脑系统软件、机器人操作系统软件对回车换行的定义不一致，一般不建议选择RT作为消息结束符。
- **命令分隔符/command_delimiter**：命令结束符。当选择了其中某一种时，另外一种就会被视作普通字符随意使用。服务端在接收客户端发送的消息时，会将第一个命令结束符之前的所有消息视为命令字符，之后的消息视为内容信息。如果命令字符不是 SET_POSE1 、SET_POSE2 、GET_POSE1 、GET_POSE2 、SET_JOINT 、GET_JOINT 、CAPTURE 中的任意一个，则是无效命令，不做任何后续处理与响应。
  - Space：空格。
  - ，：逗号。
- **回复结束符/arc_eof**：回传值结束符。当输入该值时，在回传值结尾添加结束符。默认为空。
- **坐标/pose**：设置 pose 命令消息接受到的 pose 在 3D 视图中的可视化属性。
  -  打开 pose 可视化。
  -  关闭 pose 可视化。
  -  设置坐标的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。
- **字符串/string**：设置 string 命令消息接受到的字符的曝光属性，打开后则可以将 string 输出端口的内容绑定到交互面板上的文本框并输出显示。
  -  打开 string 曝光。
  -  关闭 string 曝光。

控制信号输入输出

输入：

- **start**：
  - 触发start信号端口运行算子，开始创建 TCP 服务端，创建成功后监听服务端口并等待客户端访问。

    说明：本算子仅需要初始化运行一次即可。
- **stop**：
  - 触发 stop 信号端口后，开始停止 TCP 服务端的监听。
- **reset**：
  - 该功能保留。

输出：

- **finished**：
  - 该功能保留。
- **failed**：
  - 算子运行失败后触发该端口。

- `started`：
  - 算子成功建立 TCP 服务端后触发该端口。
- `processing`：
  - 该功能保留。
- `stopped`：
  - 算子成功停止 TCP 服务端的监听后触发该端口。
- `reset`：
  - 该功能保留。

功能演示

本节功能演示与 TestTCPCommand 算子类似。请参照 TestTCPCommand 算子的案例演示。

CommonTCPClient 通用TCP客户端

CommonTCPClient 算子用于启动一个 TCP 客户端，向服务端发送连接请求，建立连接后自动发送字符串信息。

算子参数

- **主机地址/host**：申请访问的 TCP server 服务器的 ip 地址。默认为 localhost ，即本机访问 。

说明：Windows 版本默认为127.0.0.1 ，即本机访问 。

- **端口/port**：申请访问的 TCP server 服务器的服务端口。默认值：2015。
- **客户端模式/client_mode**：运行模式。
  - Once：该算子每次被触发，都会先同TCP服务端建立链接再发送消息请求，获得回复后直接断开链接。
  - Continous：每次触发算子运行，不会建立新的链接，而是在第一次链接的基础上继续发送消息请求。
- **分隔符/delimiter**：消息结束符，包含 RT 换行符 、“；”、“#”、“$”。当选择了其中某一种时，另外三种就会被视作普通字符随意使用 。客户端在接收服务端的回复消息时，会捕获第一个消息结束符之前的所有消息。

  注意：由于不同的通信软件、电脑系统软件、机器人操作系统软件对回车换行的定义不一致，一般不建议选择RT作为消息结束符。
- **只发送模式/only_send**：只发送模式。
  - True：向服务器发送信息后，服务器需要回复，否则报错。
  - False：向服务器发送信息后，服务器不需要回复。
- **发送信息/send_message**：向服务端发送的字符串内容。同输入端口 send_message 的作用一致。当输入端口 send_message 没有连接时，则必须给该send_message 参数赋值。

  注意：在算子运行后，重新更改了port 、 client_mode 、delimiter 中的任意一个之后，如果需要生效，都必须重启 RVS 。

数据信号输入输出

输入：

- **send_message**：
  - 数据类型：String
  - 输入内容：向服务端发送的字符串内容

功能演示

使用两个 RVS 窗口，一个建立TCP服务端，另一个建立TCP客户端，实现一次连接通信。

注意：可以在一个 RVS 软件界面中同时建立服务端和建立客户端，但不允许两者之间建立链接、发送申请。

步骤1：算子准备

添加 Trigger、TestTCPServer 、CommonTCPClient 算子至算子图。

步骤2：设置算子参数

RVS 软件界面1：

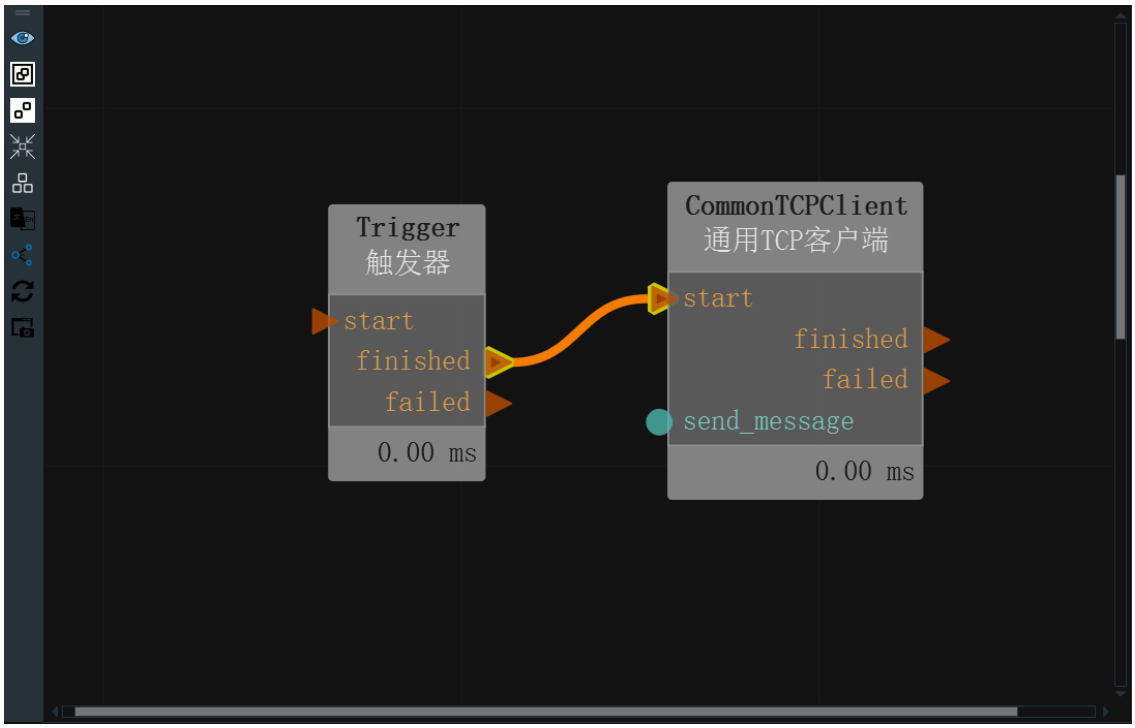
1. 设置 Trigger 算子参数：
    - 算子名称 → InitTrigger
    - 类型 → InitTrigger
2. 设置 TestTCPServer 算子参数：
    - 端口 → 2015
    - 连接数量 → 2
    - 服务器模式 → Once
    - 分隔符 → #
    - 回复消息 → OK$

RVS 软件界面2：

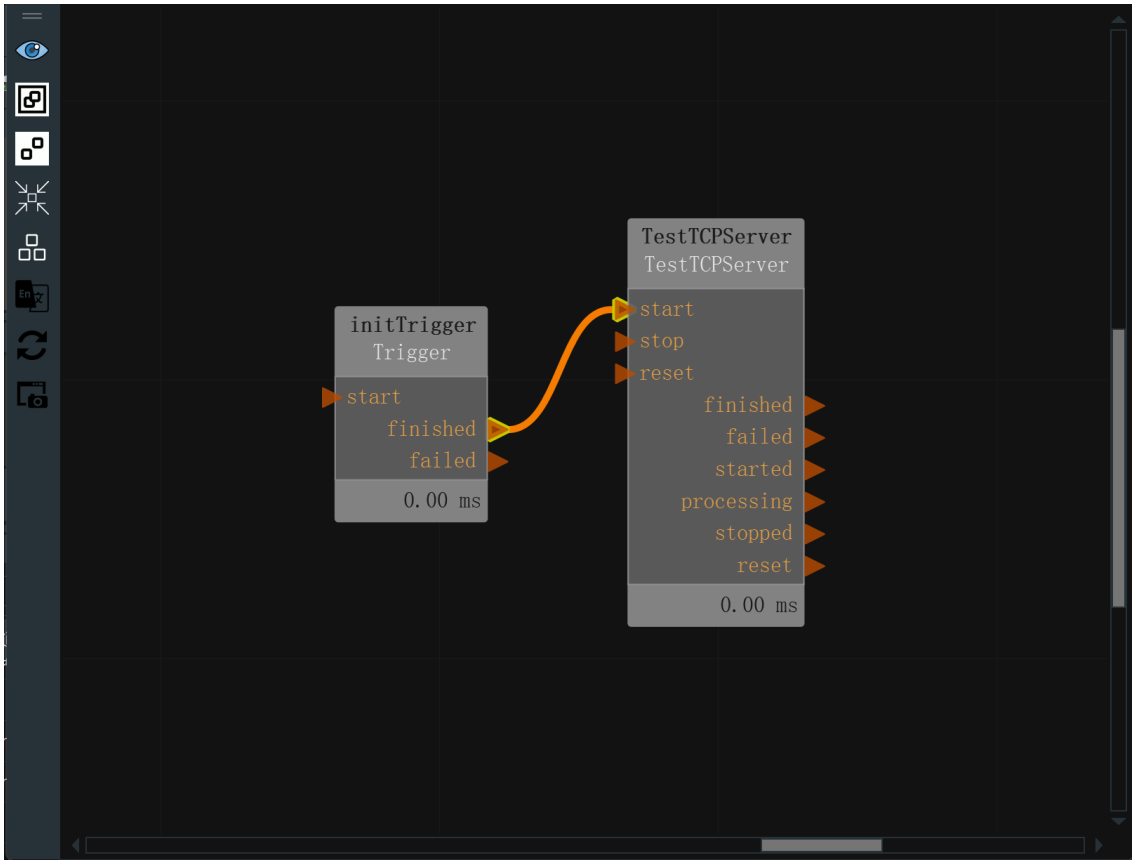
1. 设置 Trigger 算子参数：
    - 类型 → Trigger
2. 设置 CommonTCPClient 算子参数：
    - 主机地址 → localhost
    - 端口 → 2015
    - 客户端模式 → Once
    - 分隔符 → $
    - 发送消息 → test#

步骤3：连接算子

1. 打开第一个RVS软件界面，算子连接如下图所示。



2. 重新打开一个 RVS 软件界面，算子连接如下图所示。

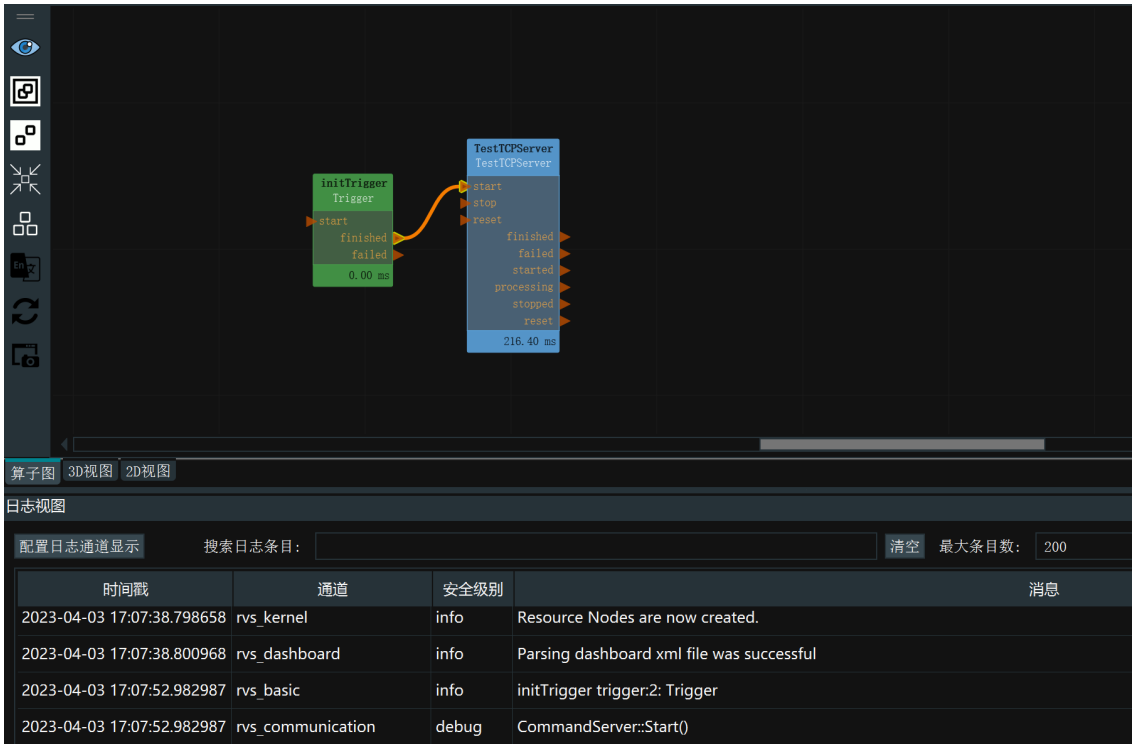


步骤4：运行

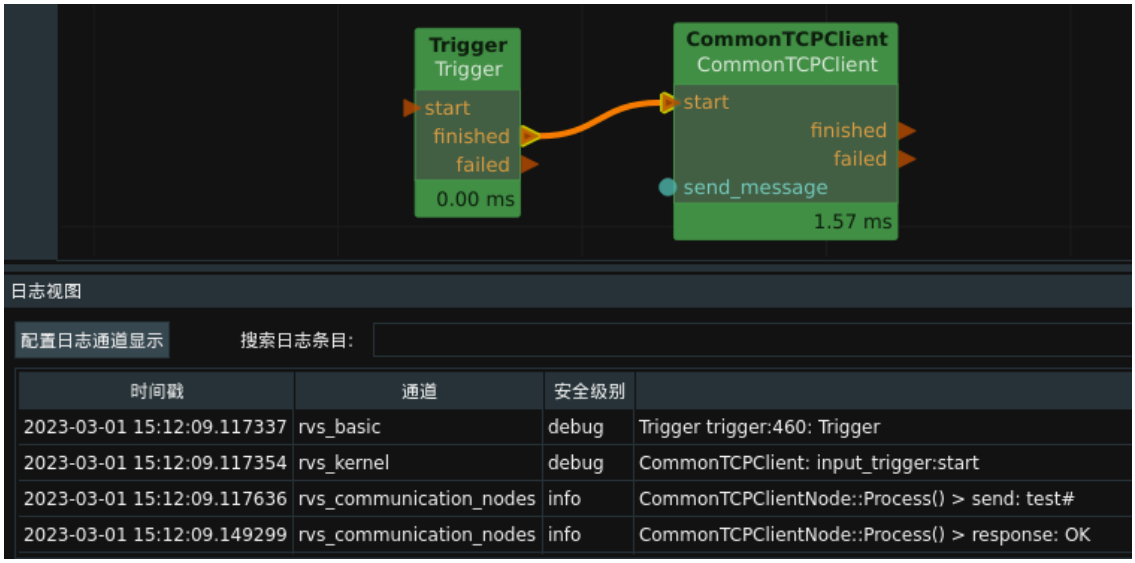
分别点击两个 RVS 软件的运行按钮。

运行结果

1. 其中 TestTCPServer 算子所在的 XML 会自动运行一次 trigger（type 为 InitTrigger），此时会自动触发 TestTCPServer 算子完成第一次的初始化运行，运行成功后界面显示如下图，表示 TCP 的服务端已经建立完毕。



2. 在 CommonTCPClient 算子所在的 RVS 手动触发一次 Trigger 算子，运行结果如下图所示。



3. 同时 TestTCPServer 算子所在的 RVS 也会有日志打印，如下所示。观察下方日志，发现 TestTCPServer 算子接收到的消息并不是 CommonTCPClient 算子发送的 test# ，而是 test#$ ，即 CommonTCPClient 算子实际发送时会自动给发送消息的末尾添加上自己的字符结束符。

TestTCPServer 测试TCP服务器

TestTCPServer 算子属于线程类算子，用于在 RVS 的分线程中启动一个 TCP server 服务器，并按照固定的字符串信息对所有的访问请求进行回复。

算子参数

- **端口/port**：创建 TCP server 服务器所使用的本机服务端口。

  说明：实际运行时，如果提示 TCP 服务端创建失败，往往是所选取的端口已经被其他服务所占用，更换端口即可。如果在同一个 RVS 软件中，创建多个TCP 服务端算子，彼此的 port 也要互斥。

- **连接数量/connections**：可支持同时连接的客户端数量。
- **服务器模式/server_mode**：运行模式。
  - Once：表示完成一次 TCP 对话以后自动断开同客户端的链接。
  - Continous：表示客户端建立链接后可以无限次数的对话。
- **分隔符/delimiter**：消息结束符。包含 RT 换行符、“；”、“#”、“$”。当选择了其中某一种时，另外三种就会被视作普通字符随意使用 。服务端在接收客户端发送的消息时，会捕获第一个消息结束符之前的所有消息。

  说明：由于不同的通信软件、电脑系统软件、机器人操作系统软件对回车换行的定义不一致，容易导致信息发送或者接收失败，所以一般不建议选择RT作为消息结束符。

- **回复结束符/arc_eof**：回传值结束符。当输入该值时，在回传值结尾添加结束符。默认为空。
- **response**：对于每一次访问请求，都固定回复的内容。

  注意：在算子运行后，重新更改了port 、connections 、server_mode 、delimiter 四个属性中的任意一个之后，如果需要生效，都必须重启 RVS 。

控制信号输入输出

输入：

- **start**：
  - 触发 start 信号端口运行算子，开始创建 TCP 服务端，创建成功后监听服务端口并等待客户端访问。

    说明：本算子仅需要初始化运行一次即可。

- **stop**：
  - 触发 stop 信号端口后，开始停止 TCP 服务端的监听。
- **reset**：
  - 该功能保留。

输出：

- **finished**：
  - 该功能保留。
- **failed**：
  - 算子运行失败后触发该端口。
- **started**：
  - 算子成功建立 TCP 服务端后触发该端口。
- **processing**：

    - 该功能保留。
    - 算子成功停止 TCP 服务端的监听后触发该端口。
- `reset`：
    - 该功能保留。

功能演示

本节功能演示与 CommonTCPClient 算子类似。请参照 CommonTCPClient 算子的功能演示。

HandEyeTCPServer 手眼标定TCP服务器

HandEyeTCPServer 算子属于线程类算子，用于在 RVS 的分线程中启动一个 TCP server 服务器。

该算子主要应用于手眼协作过程(当前用的最多的是拆垛场景)中, RVS 同机械手(或其他工控机程序)的通信。该服务器仅能接收固定的 8 类字符串信息并对应的回复固定字符串。

8 类字符串分别以 8 个命令作为起始值，分别是 ROBOT_TCP 、ROBOT_JOINTS 、CAPTURE 、CAPTURE_MESSAGE 、GET_JOINT 、GET_POSE 、GET_POSES 、GET_POSES_NEXT 。

说明：代码块中命令参数设置：command_delimiter →space、delimiter → RT，均为默认值。请根据实际场景进行调整。

其中，如果客户希望获得多个目标 pose 姿态列表，可以修改 HandEyeTCPServer 的 `number_port` 属性数量。客户端仍然按照下述 `GET_POSES` 的方式发送消息，但是此时回复的消息起始位就不再是 1 ，而是 poselist 的数量，后面跟每一个 poselist 中包含的 pose 数量。

```
GET_POSES \n
```

1. 客户端发送以命令字符 `ROBOT_TCP` 开始的消息，要求在该 `ROBOT_TCP` 命令字符后紧跟 `command_delimiter+'x y z rx ry rz'+delimiter` 。接收到该信息后，该算子（即服务端）会将命令字符后续的 `x y z rx ry rz` 字符内容转换为 6 个 float 数值，进而赋值给一个 pose 。

   如果上述字符串消息可以合法转换为 6 个 float 则会触发执行该算子右下侧的 tcp 数据端口，并通过算子右下侧的 tcp 端口输出所得的 pose，如果字符串消息不合法，则不进行任何后续响应。

   我们实际使用时，需要将该算子右下侧 ROBOT_TCP 端口后续的触发结束信号返还连接到该算子的左侧 ROBOT_TCP 端口，因为只有成功触发该端口才会执行 TCP 服务端的消息回复。

   对于命令字符 `ROBOT_TCP”` 开始的消息访问，最终统一回复 `ROBOT_TCP` 。

   客户端发送：

   ```
   ROBOT_TCP x y z rx ry rz \n
   ```

   算子右侧 tcp 端口输出：

   ```
   x y z rx ry rz
   ```

   服务端回复：

   ```
   ROBOT_TCP
   ```

2. 客户端发送以命令字符 `ROBOT_JOINTS` 开始的消息，要求在该 `ROBOT_JOINTS` 命令字符后紧跟 `command_delimiter+'J1 J2 J3 J4 J5 J6'+delimiter` 。

   接收到该信息后，服务端会将命令字符后续的 `J1 J2 J3 J4 J5 J6` 字符内容转换为 6 个 float 数值，进而赋值给一个 JointArray 。

   如果上述字符串消息可以合法转换为 6 个 float 则会触发执行该算子右下侧的 joints 数据端口，并通过算子右下侧的 joints 端口输出所得的 JointArray ,如果字符串消息不合法，则不进行任何后续响应。

   我们实际使用时，需要将该算子右下侧 ROBOT_JOINTS 端口后续的触发结束信号返还连接到该算子的左侧 ROBOT_JOINTS 端口，因为只有成功触发该端口才会执行 TCP 服务端的消息回复。

对于命令字符 `ROBOT_JOINTS` 开始的消息访问，最终统一回复 `ROBOT_JOINTS` 。

客户端发送：

```
ROBOT_JOINTS J1 J2 J3 J4 J5 J6 \n
```

算子右侧 joints 端口输出：

```
J1 J2 J3 J4 J5 J6
```

服务端回复：

```
ROBOT_JOINTS
```

3. 客户端发送以命令字符 `CAPTURE` 开始的消息，要求在该 `CAPTURE` 命令字符后紧跟 `command_delimiter+ 任意内容 +delimiter` 。

   接收到该信息后，该算子会自动触发右下侧的 CAPTURE 端口，进而触发其他后续算子进行运算。

   我们实际使用时，需要将该算子右下侧 CAPTURE 端口后续的触发结束信号返还连接到该算子的左侧 CAPTURE 端口，因为只有成功触发该端口才会执行 TCP 服务端的消息回复。

   对于命令字符 `CAPTURE` 开始的消息访问，服务端对命令字符后续的其他字符内容不做任何处理，并最终统一回复 “CAPTURE” 字符串消息。

   客户端发送：

   ```
   CAPTURE \n 或者 CAPTURE any other message\n
   ```

   服务端回复：

   ```
   CAPTURE
   ```

4. 如果需要在 `CAPTURE` 命令字符消息中发送一些其他字符串给 RVS ，则需要使用 `CAPTURE_MESSAGE` ，并要求在该 `CAPTURE_MESSAGE` 命令字符后紧跟 `command_delimiter+ 其他内容 +delimiter` 。

   使用原理同上述第 3 条 ，不同的是，这些从客户端发来的 `其他内容` 的字符串会通过算子右下侧的 capture_str 端口传递到 RVS 。

   假设项目现场有两台相机一左一右。

   通过给RVS 发送以下命令表示拍摄左侧相机：

   ```
   CAPTURE_MESSAGE 0 \n
   ```

   通过给RVS 发送以下命令表示拍摄右侧相机：

   ```
   CAPTURE_MESSAGE 1 \n
   ```

5. 客户端发送以命令字符 `GET_JOINT` 开始的消息，要求在该 `GET_JOINT` 命令字符后紧跟 `command_delimiter+ 其他内容 +delimiter` 。接收到该信息后，该算子会自动触发右下侧的 GET_JOINT 端口，进而触发其他后续算子进行运算。

我们实际使用时，需要将该算子右下侧 GET_JOINT 端口后续的触发结束信号返还连接到该算子的左侧 GET_JOINT 端口，因为只有成功触发该端口才会执行 TCP 服务端的消息回复。

对于命令字符 `GET_JOINT` 开始的消息访问，服务端对命令字符后续的其他字符内容不做任何处理。当左侧 GET_JOINT 端口被触发执行消息回复时，如果算子左侧的 all_finished 端口已被触发，则会回复 `000000000 000000000 000000000 000000000 000000000 000000000 2` 字符串信息，如果算子左侧的 detect_error 端口已被触发,则会回复 `000000000 000000000 000000000 000000000 000000000 000000000 3` 字符串信息，如果算子左侧的 all_finished 端口和 detect_error 端口均未被触发，则算子内部会检查算子左侧的 joint 端口是否被连接，如果该 joint 端口有连接并且数值有效，则会回复 `J1 J2 J3 J4 J5 J6 1`给客户端。

注意

末尾的1，所有回复字符串的末尾均是标志位。

1 表示获得正常数值。

2 表示虽然回复结果无效但是成功结束。

3 表示回复结果无效且失败。

另外，字符串前面的六位数值都保留 9 位有效数字，所以为了对齐字符串长度，所有无效数值也都用九个 0 表示。单位：弧度制。

如果 joint 端口没有连接或者数值无效，则会回复 `000000000 000000000 000000000 000000000 000000000 000000000 3` 给客户端。以command_delimiter 参数设置为逗号，delimiter 参数设置为分号为例，左侧 joint 端口输入 JointArrayJ(0 0 0 3.14159 -1.5709 0) ，则当客户端发送 `GET_JOINT,;` 时，则最终服务端会回复 `+0.000000 +0.000000 +0.000000 +3.141590 -1.570900 +0.000000 1` ，算上中间的空格，字符串总长度是 61 位。

客户端发送：

```
GET_JOINT 其他内容 \n
```

6. 客户端发送以命令字符 `GET_POSE` 开始的消息，原理同上述第 5 点 `GET_JOINT` 的说明相同。

   客户端发送：

```
GET_POSE command_delimiter 其他内容 delimiter
```

7. 客户端发送以命令字符 `GET_POSES` 开始的消息，要求在该 `GET_POSES` 命令字符后紧跟 `command_delimiter+ 其他内容 +delimiter`。接收到该信息后，该算子会自动触发右下侧的 GET_POSES 端口，进而触发其他后续算子进行运算。

   我们实际使用时，需要将该算子右下侧 GET_POSES 端口后续的触发结束信号返还连接到该算子的左侧 GET_POSES 端口，因为只有成功触发该端口才会执行 TCP 服务端的消息回复。对于命令字符 `GET_POSES` 开始的消息访问，服务端对命令字符后续的其他字符内容不做任何处理。

   ```
   GET_POSES 其他内容 \n
   ```

   该命令字符的作用：通过 RVS 软件的服务端将一个乃至多个 PoseList 发送给客户端。具体的 PoseList 的数量可以根据算子属性 `number_port` 进行设置，该属性默认是 1，对应算子左侧的 poses_1 端口。一旦更新该属性比如更改为 3 ，则算子左侧会新增 poses_2 、poses_3 端口。

   服务端最终回复的信息格式是：

   ```
   PoseList 的总个数  每个 PoseList 的 pose 个数  所有 pose 的实际数值  标志位
   ```

   最后的标志位包含 0、1、2、3 四种情形。

   - 0：数据正常且发送完毕
   - 1：数据正常但尚未发送完
   - 2：数据无效但正常运行
   - 3：数据无效且运行异常

   当总共有 3 组 PoseList (各自含有2、1、2个 pose)需要发送时，最后回复给客户端的字符串为：

   ```
   3 2 1 2 poselist1_pose1_x  poselist1_pose1_y poselist1_pose1_z poselist1_pose1_rotated_x poselist1_pose1_rotated_y  poselist1_pose1_rotated_z ..... poselist3_pose2_x poselist3_pose2_y poselist3_pose2_z poselist3_pose2_rotated_x  poselist3_pose2_rotated_y poselist3_pose2_rotated_z 0
   ```

   但是由于传送的数据量比较大，所以算子设置了每次回传信息的最大字节长度，对应算子属性 receive_byte_length 。

   - 当上述字符串字节 (不包含末尾的空格和 0)总长度小于等于 receive_byte_length-2 时，即直接回复上述字符串。
   - 若大于receive_byte_length-2 ，则会将上述字符串拆分，仅截取字符串前面长度为 receive_byte_length-2 的一段，然后在该截取片段的末尾加上空格和 1 回复给客户端。客户端接收到末尾为 1 的字符串后，需要立刻继续向本算子发送 GET_POSES_NEXT 的请求，此时服务端会将上述字符串剩余的部分继续截取一段长度为 receive_byte_length-2 的字符串并补充空格和 1 ，继续回复给客户端，循环该过程直到发送完成为止，最后一次发送的字符串长度不一定是 receive_byte_length ，但是其末尾一定是补充的空格和 0 。

   根据上述规则，当左侧 GET_POSES 端口被触发执行消息回复时，如果算子左侧的 all_finished 端口已被触发，则会回复 `0 2` 字符串信息，如果算子左侧的 detect_error 端口已被触发,则会回复 `0 3` 字符串信息，如果算子左侧的 all_finished 端口和 detect_error 端口均未被触发，则算子内部会检查算子左侧从poses_1 端口到 pose_n 端口中是否有连接以及有效数据，如果有至少一个端口有有效数据，则会按照上述过程回复客户端，如果所有端口都没有有效数值，则会回复 `0 3` 给客户端。

8. 客户端发送以命令字符 `GET_POSES_NEXT` 开始的消息，要求在该 `GET_POSES_NEXT` 命令字符后紧跟 `command_delimiter+ 其他内容 +delimiter` 。这里的 `其他内容` 为无效数值。该 GET_POSES_NEXT 命令字符是 GET_POSES 命令字符的补充，直接单独发送该命令字符串没有意义，使用方法详见上述 GET_POSES 的使用介绍。

```
GET_POSES_NEXT 其他内容 \n
```

算子参数

- **端口/port**：创建TCP server服务器所使用的本机服务端口。

  说明：实际运行时，如果提示 TCP 服务端创建失败，往往是所选取的端口已经被其他服务所占用，更换端口即可。如果在同一个 RVS 软件中，创建多个 TCP 服务端算子，彼此的 port 也要互斥。

- **连接数量/connections**：可支持同时连接的客户端数量。
- **服务器模式/server_mode**：运行模式。
  - Once：表示完成一次 TCP 对话以后自动断开同客户端的连接(比如客户端首先同该算子建立了 TCP 连接，然后第一次发送了 ROBOT_TCP 命令的字符串之后，继续发送 ROBOT_JOINTS 命令的字符串，此时就会报错)。
  - Continous： 表示客户端建立链接后可以无限次数的对话。
- **分割符/delimiter**：消息结束符。包含 RT 换行符、“；”、“#”、“$”。当选择了其中某一种时，另外三种就会被视作普通字符随意使用 。服务端在接收客户端发送的消息时，会捕获第一个消息结束符之前的所有消息。

  注意：由于不同的通信软件、电脑系统软件、机器人操作系统软件对回车换行的定义不一致，一般不建议选择RT作为消息结束符。

- **命令分割符/command_delimiter**：命令结束符。当选择了其中某一种时，另外一种就会被视作普通字符随意使用。服务端在接收客户端发送的消息时，会将第一个命令结束符之前的所有消息视为命令字符，之后的消息视为内容信息。如果命令字符不是上述功能总结部分的 8个命令中的任意一个，则是无效命令，不做任何后续处理与响应。
  - Space：空格。
  - , ：逗号。
- **回复结束符/arc_eof**：回传值结束符。当输入该值时，在回传值结尾添加结束符。默认为空。
- **机器人TCP/tcp**：设置通过 ROBOT_TCP 命令消息接收到的 pose 可视化属性。
  -  打开 pose 可视化。
  -  关闭 pose 可视化。
  -  设置坐标的尺寸大小。取值范围：[0.001,10] 。默认值：0.1 。
- **拍摄信息/capture_str**：设置曝光属性。打开后则可以将 capture_str 输出端口的内容绑定到交互面板上的文本框并输出显示。
  -  打开曝光。
  -  关闭曝光。
- **输入数量/number_port**： 仅供发送 GET_POSES 命令字符串时使用，详见上述功能总结部分的第7条。
- **接受字节长度/receive_byte_length**：仅供发送 GET_POSES 命令字符串时使用，详见上述功能总结部分的第7条。默认值：1024。单位：byte。取值范围：[1,+∞]。

注意：在算子运行后，重新更改了 port 、connections 、server_mode 、delimiter 、command_delimiter、arc_eof 6个属性中的任意一个之后，如果需要生效，都必须重启或者重置 RVS 。

控制信号输入输出

输入：

- `start`：
  - 触发 start 信号端口运行算子，开始创建 TCP 服务端，创建成功后监听服务端口并等待客户端访问。

    说明：本算子仅需要初始化运行一次即可。
- `stop`：
  - 触发 stop 信号端口后，开始停止 TCP 服务端的监听。
- `reset`：
  - 该功能保留。

输出：

- `finished`：
  - 该功能保留。
- `failed`：
  - 算子运行失败后触发该端口。
- `started`：
  - 算子成功建立TCP服务端后触发该端口。
- `processing`：
  - 该功能保留。
- `stopped`：
  - 算子成功停止TCP服务端的监听后触发该端口。
- `reset`：
  - 该功能保留。

功能演示

下面分别针对 8 类命令，进行通讯演示。

在选用演示所用的 TCP 客户端时，由于 RVS 自带的 CommonTCPClient 算子在使用时要求服务端返回的信息必须以四类固定字符结尾，而 HandEyeTCPServer 算子在作为 TCP 的服务端时，返回给 TCP Client 的信息是固定的并且没有额外添加上述四类固定结尾字符，所以这里不再选用 CommonTCPClient 算子作为客户端。

说明：这里的演示案例是基于ubuntu系统给出的，使用了"echo + nc "的指令方式实现TCP客户端的功能；如果是在Windows版本，建议使用通讯助手等工具。

步骤1：算子准备

右击创建新 Group ，添加 Trigger、Emit、Concatenate、HandEyeTCPServer 算子至 Group 算子图。

步骤2：设置算子参数

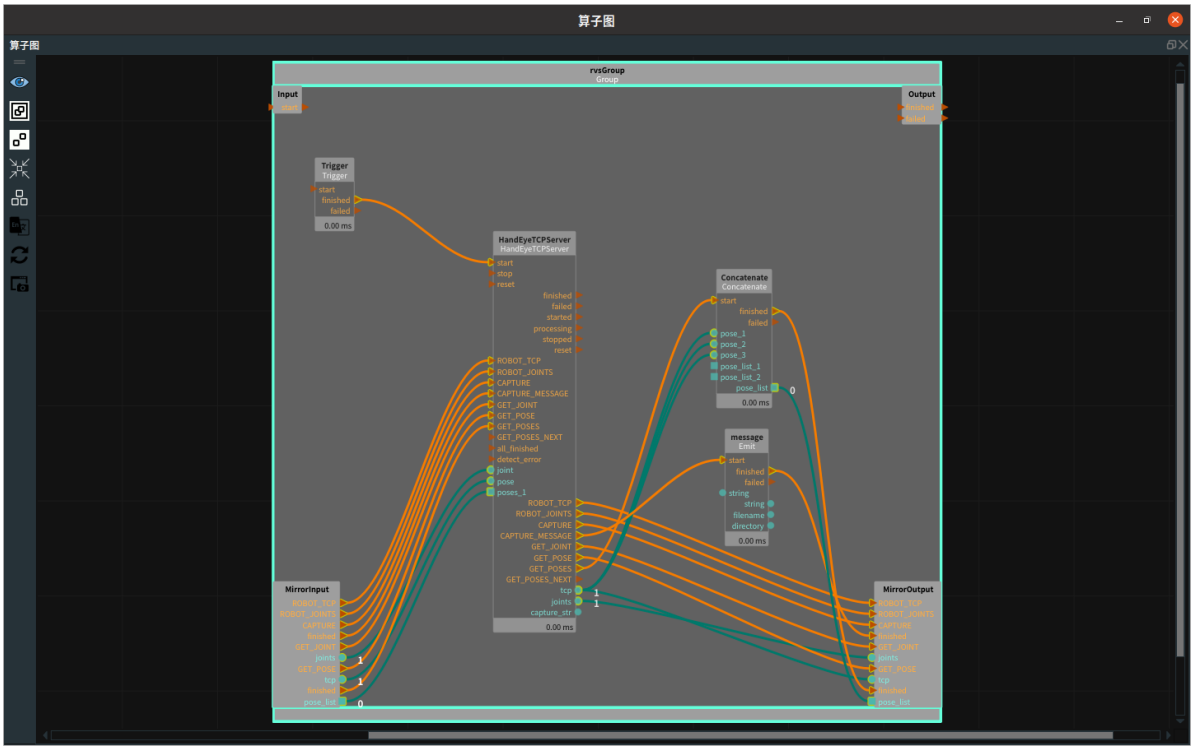
1. 设置 Trigger 算子参数：
   - 类型 → InitTrigger
2. 设置 HandEyeTCPServer 算子参数：
   - 分隔符 → ;

- 命令分隔符 → ,
- 回复结束符 → end

3. 设置 Concatenate 算子参数：
   - 类型→ pose
   - 输入数量 → 3
4. 设置 Emit 算子参数：
   - 算子名称 → Message
   - 类型 → String

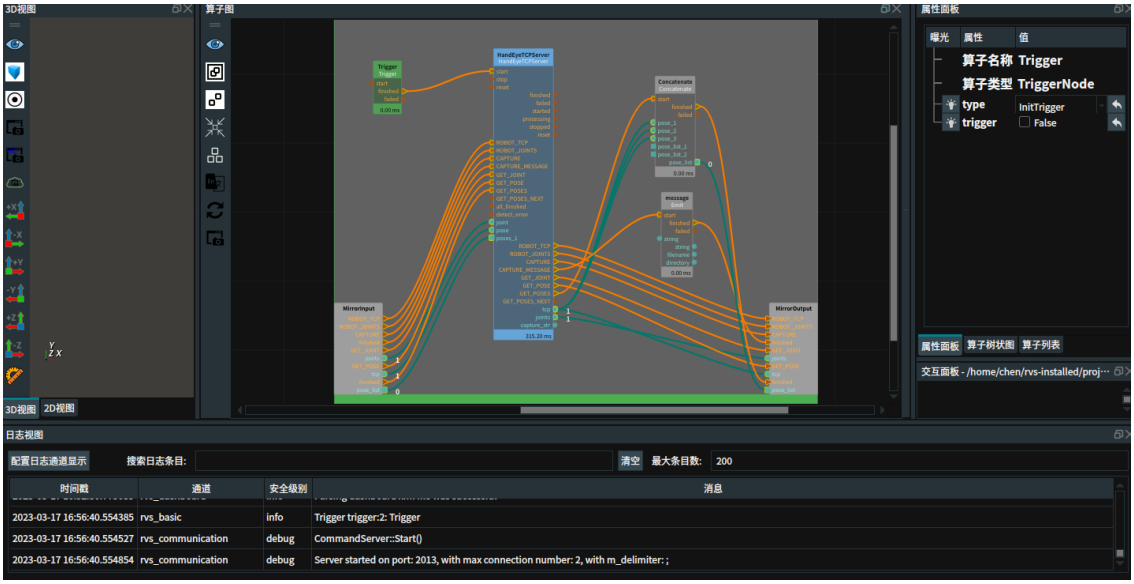
步骤3：连接算子

打开RVS软件界面，在算子图拖入下图所述算子并连接。由于选用了“echo + nc”的指令方式，所以这里每发送-接收一次消息后都要断开连接，所以选择了Once模式。



步骤3：运行与结果

1. 打开 RVS 的运行按钮，HandEyeTCPServer 算子会被 Trigger 算子自动触发，并变为蓝色，日志栏会同时打印算子运行说明如下图所示，表示TCP的服务端已经建立完毕。



2. 重新打开一个终端，输入命令 echo "ROBOT_TCP,0.1 0.2 0.1 3.14159 1.5709 1;" | nc localhost 2013 并单击回车键，运行结束后，我们会在这个终端窗口看到回复字符串如下图所示。

```
zmt@pink:~$ echo "ROBOT_TCP,0.1 0.2 0.1 3.14159 1.5709 1;" | nc localhost 2013
ROBOT_TCPendzmt@pink:~$
```

   - 同时 RVS 的运行日志如下。

2023-03-02 15:31:06.450545	rvs_communication	debug	CommandService::StartHandling()
2023-03-02 15:31:06.450735	rvs_communication	debug	CommandService::onRequestReceived()
2023-03-02 15:31:06.499141	rvs_communication	debug	rvsCommandServerNode::ProcessRequest
2023-03-02 15:31:06.499164	rvs_communication	debug	first_command:ROBOT_TCP
2023-03-02 15:31:06.499175	rvs_communication	debug	second_data:0.1 0.2 0.1 3.14159 1.5709 1
2023-03-02 15:31:06.499184	rvs_communication	debug	Handling command: ROBOT_TCP
2023-03-02 15:31:06.500170	rvs_communication_nodes	info	Rec > ROBOT_TCP: 0.1 0.2 0.1 3.14159 1.5709 1
2023-03-02 15:31:06.500200	rvs_communication	info	HandEyeTCPServer: Triggered command via server: 5815: Trigger
2023-03-02 15:31:06.603466	rvs_communication_nodes	info	Ack > ROBOT_TCP
2023-03-02 15:31:06.603483	rvs_communication	debug	HandEyeTCPServer: ProcessResponse for command: ROBOT_TCP with response: ROBOT_TCP
2023-03-02 15:31:06.603618	rvs_communication	debug	CommandService::onResponseSent()
2023-03-02 15:31:06.603650	rvs_communication	debug	CommandService::onFinish()
2023-03-02 15:31:06.603667	rvs_communication	debug	CommandAcceptor::UnregisterCommandService()

3. 在终端中，继续输入命令echo "ROBOT_JOINTS,0 0 0 3.14159 1.5709 0;" | nc localhost 2013并单击回车键，运行结束后，我们会在这个终端窗口看到回复字符串如下图所示。

```
zmt@pink:~$ echo "ROBOT_JOINTS,0 0 0 3.14159 1.5709 0;" | nc localhost 2013
ROBOT_JOINTSendzmt@pink:~$
```

   - 同时RVS的运行日志如下。

2023-03-02 15:38:41.054052	rvs_communication	debug	CommandService::StartHandling()
2023-03-02 15:38:41.054232	rvs_communication	debug	CommandService::onRequestReceived()
2023-03-02 15:38:41.105070	rvs_communication	debug	rvsCommandServerNode::ProcessRequest
2023-03-02 15:38:41.105087	rvs_communication	debug	first_command:ROBOT_JOINTS
2023-03-02 15:38:41.105098	rvs_communication	debug	second_data:0 0 0 3.14159 1.5709 0
2023-03-02 15:38:41.105107	rvs_communication	debug	Handling command: ROBOT_JOINTS
2023-03-02 15:38:41.105155	rvs_communication_nodes	info	Rec > ROBOT_JOINTS: 0 0 0 3.14159 1.5709 0
2023-03-02 15:38:41.105179	rvs_communication	info	HandEyeTCPServer: Triggered command via server: 13749: Trigger
2023-03-02 15:38:41.209667	rvs_communication_nodes	info	Ack > ROBOT_JOINTS
2023-03-02 15:38:41.209687	rvs_communication	debug	HandEyeTCPServer: ProcessResponse for command: ROBOT_JOINTS with response: ROBOT_JOINTS
2023-03-02 15:38:41.209784	rvs_communication	debug	CommandService::onResponseSent()
2023-03-02 15:38:41.209808	rvs_communication	debug	CommandService::onFinish()
2023-03-02 15:38:41.209820	rvs_communication	debug	CommandAcceptor::UnregisterCommandService()

4. 在终端中，继续输入命令echo "CAPTURE,;" | nc localhost 2013并单击回车键，运行结束后，我们会在这个终端窗口看到回复字符串如下图所示。

```
zmt@pink:~$ echo "CAPTURE,;" | nc localhost 2013
CAPTUREendzmt@pink:~$
```

   - 同时RVS的运行日志如下。

2023-03-02 15:51:14.698263	rvs_communication	debug	CommandService::StartHandling()
2023-03-02 15:51:14.698422	rvs_communication	debug	CommandService::onRequestReceived()
2023-03-02 15:51:14.749180	rvs_communication	debug	rvsCommandServerNode::ProcessRequest
2023-03-02 15:51:14.749198	rvs_communication	debug	first_command:CAPTURE
2023-03-02 15:51:14.749208	rvs_communication	debug	second_data:
2023-03-02 15:51:14.749216	rvs_communication	debug	Handling command: CAPTURE
2023-03-02 15:51:14.749233	rvs_communication_nodes	info	Rec > CAPTURE
2023-03-02 15:51:14.749242	rvs_communication	info	HandEyeTCPServer: Triggered command via server: 26920: Trigger
2023-03-02 15:51:14.865817	rvs_communication_nodes	info	Ack > CAPTURE
2023-03-02 15:51:14.865878	rvs_communication	debug	HandEyeTCPServer: ProcessResponse for command: CAPTURE with response: CAPTURE
2023-03-02 15:51:14.866096	rvs_communication	debug	CommandService::onResponseSent()
2023-03-02 15:51:14.866192	rvs_communication	debug	CommandService::onFinish()
2023-03-02 15:51:14.866258	rvs_communication	debug	CommandAcceptor::UnregisterCommandService()

5. 在终端中，继续输入命令echo "CAPTURE_MESSAGE,0;" | nc localhost 2013并单击回车键，运行结束后，我们会在这个终端窗口看到回复字符串如下图所示。

```
zmt@pink:~$ echo "CAPTURE_MESSAGE,0;" | nc localhost 2013
CAPTURE_MESSAGEendzmt@pink:~$
```

   - 同时RVS的运行日志如下。

```
2023-03-02 16:05:49.393273 rvs_communication       debug CommandService::StartHandling()
2023-03-02 16:05:49.393470 rvs_communication       debug CommandService::onRequestReceived()
2023-03-02 16:05:49.457833 rvs_communication       debug rvsCommandServerNode::ProcessRequest
2023-03-02 16:05:49.457853 rvs_communication       debug  first_command:CAPTURE_MESSAGE
2023-03-02 16:05:49.457861 rvs_communication       debug  second_data:0
2023-03-02 16:05:49.457867 rvs_communication       debug Handling command: CAPTURE_MESSAGE
2023-03-02 16:05:49.457913 rvs_communication_nodes info  Rec > CAPTURE_MESSAGE0
2023-03-02 16:05:49.457924 rvs_communication       info  HandEyeTCPServer: Triggered command via server: 487: Trigger
2023-03-02 16:05:49.458041 rvs_kernel              debug Message: input_trigger:start
2023-03-02 16:05:49.458105 rvs_basic               debug output_string: 0
2023-03-02 16:05:49.458113 rvs_basic               debug output_filename: 0
2023-03-02 16:05:49.458120 rvs_basic               debug output_directory: 0
2023-03-02 16:05:49.564433 rvs_communication_nodes info  Ack > CAPTURE_MESSAGE
2023-03-02 16:05:49.564456 rvs_communication       debug HandEyeTCPServer: ProcessResponse for command: CAPTURE_MESSAGE with response: CAPTURE_MESSAGE
2023-03-02 16:05:49.564605 rvs_communication       debug CommandService::onResponseSent()
2023-03-02 16:05:49.564645 rvs_communication       debug CommandService::onFinish()
2023-03-02 16:05:49.564664 rvs_communication       debug CommandAcceptor::UnregisterCommandService()
```

6. 在终端中，继续输入命令echo "GET_JOINT,;" | nc localhost 2013并单击回车键，运行结束后，我们会在这个终端窗口看到回复字符串如下图所示。

```
zmt@pink:~$ echo "GET_JOINT,;" | nc localhost 2013
+0.000000 +0.000000 +0.000000 +3.141590 +1.570900 +0.000000 1endzmt@pink:~$
```

   - 同时RVS的运行日志如下

```
2023-03-02 16:25:59.099057 rvs_communication       debug CommandService::StartHandling()
2023-03-02 16:25:59.099213 rvs_communication       debug CommandService::onRequestReceived()
2023-03-02 16:25:59.151096 rvs_communication       debug rvsCommandServerNode::ProcessRequest
2023-03-02 16:25:59.151115 rvs_communication       debug  first_command:GET_JOINT
2023-03-02 16:25:59.151129 rvs_communication       debug  second_data:
2023-03-02 16:25:59.151138 rvs_communication       debug Handling command: GET_JOINT
2023-03-02 16:25:59.151157 rvs_communication_nodes info  Rec > GET_JOINT
2023-03-02 16:25:59.151168 rvs_communication       info  HandEyeTCPServer: Triggered command via server: 21624: Trigger
2023-03-02 16:25:59.255427 rvs_communication_nodes info  Ack > +0.000000 +0.000000 +0.000000 +3.141590 +1.570900 +0.000000 1
2023-03-02 16:25:59.255447 rvs_communication       debug HandEyeTCPServer: ProcessResponse for command: GET_JOINT with response: +0.000000 +0.000000 +0.000000 +3.141590 +1.570900 +0.000000 1
2023-03-02 16:25:59.255548 rvs_communication       debug CommandService::onResponseSent()
2023-03-02 16:25:59.255571 rvs_communication       debug CommandService::onFinish()
2023-03-02 16:25:59.255581 rvs_communication       debug CommandAcceptor::UnregisterCommandService()
```

7. 在终端中，继续输入命令echo "GET_POSE,;" | nc localhost 2013并单击回车键，运行结束后，我们会在这个终端窗口看到回复字符串如下图所示，同时RVS的运行日志如下。

```
zmt@pink:~$ echo "GET_POSE,;" | nc localhost 2013
+0.100000 +0.200000 +0.100000 +3.141590 +1.570900 +1.000000 1endzmt@pink:~$
```

   - 同时RVS的运行日志如下

```
2023-03-02 16:31:52.389775 rvs_communication       debug CommandService::StartHandling()
2023-03-02 16:31:52.389933 rvs_communication       debug CommandService::onRequestReceived()
2023-03-02 16:31:52.442504 rvs_communication       debug rvsCommandServerNode::ProcessRequest
2023-03-02 16:31:52.442521 rvs_communication       debug  first_command:GET_POSE
2023-03-02 16:31:52.442532 rvs_communication       debug  second_data:
2023-03-02 16:31:52.442541 rvs_communication       debug Handling command: GET_POSE
2023-03-02 16:31:52.442557 rvs_communication_nodes info  Rec > GET_POSE
2023-03-02 16:31:52.442567 rvs_communication       info  HandEyeTCPServer: Triggered command via server: 27789: Trigger
2023-03-02 16:31:52.547338 rvs_communication_nodes info  Ack > +0.100000 +0.200000 +0.100000 +3.141590 +1.570900 +1.000000 1
2023-03-02 16:31:52.547363 rvs_communication       debug HandEyeTCPServer: ProcessResponse for command: GET_POSE with response: +0.100000 +0.200000 +0.100000 +3.141590 +1.570900 +1.000000 1
2023-03-02 16:31:52.547470 rvs_communication       debug CommandService::onResponseSent()
2023-03-02 16:31:52.547500 rvs_communication       debug CommandService::onFinish()
2023-03-02 16:31:52.547511 rvs_communication       debug CommandAcceptor::UnregisterCommandService()
```

8. 在终端中，继续输入命令echo "GET_POSES,;" | nc localhost 2013并单击回车键，运行结束后，我们会在这个终端窗口看到回复字符串如下图所示

```
zmt@pink:~$ echo "GET_POSES,;" | nc localhost 2013
1 3 0.10 0.20 0.10 3.1416 1.5709 1.0000 0.10 0.20 0.10 3.1416 1.5709 1.0000 0.10
 0.20 0.10 3.1416 1.5709 1.0000 0endzmt@pink:~$
```

   - 同时RVS的运行日志如下

```
2023-03-02 16:33:51.886910 rvs_communication       debug CommandService::StartHandling()
2023-03-02 16:33:51.887067 rvs_communication       debug CommandService::onRequestReceived()
2023-03-02 16:33:51.930167 rvs_communication       debug rvsCommandServerNode::ProcessRequest
2023-03-02 16:33:51.930180 rvs_communication       debug  first_command:GET_POSES
2023-03-02 16:33:51.930188 rvs_communication       debug  second_data:
2023-03-02 16:33:51.930193 rvs_communication       debug Handling command: GET_POSES
2023-03-02 16:33:51.930205 rvs_communication_nodes info  Rec > GET_POSES
2023-03-02 16:33:51.930211 rvs_communication       info  HandEyeTCPServer: Triggered command via server: 29881: Trigger
2023-03-02 16:33:51.930309 rvs_kernel              debug Concatenate: input_trigger:start
2023-03-02 16:33:52.039756 rvs_communication_nodes info  Ack > 1 3 0.10 0.20 0.10 3.1416 1.5709 1.0000 0.10 0.20 0.10 3.1416 1.5709 1.0000 0.10 0.20 0.10 3.1416 1.5709 1.0000 0
2023-03-02 16:33:52.039777 rvs_communication       debug HandEyeTCPServer: ProcessResponse for command: GET_POSES with response: 1 3 0.10 0.20 0.10 3.1416 1.5709 1.0000 0.10 0.20 0.10 3.1416 1.5709 1.0000 0.10 0.20 0.10 3.1416 1.5709 1.0000 0
2023-03-02 16:33:52.039857 rvs_communication       debug CommandService::onResponseSent()
2023-03-02 16:33:52.039884 rvs_communication       debug CommandService::onFinish()
2023-03-02 16:33:52.039896 rvs_communication       debug CommandAcceptor::UnregisterCommandService()
```

TestTCPCommand 测试TCP命令服务器

TestTCPCommand 算子属于线程类算子，用于在 RVS 的分线程中启动一个 TCP server 服务器。

该服务器仅能接收固定的 3 类字符串信息并对应的回复固定字符串。

3 类字符串分别以 3 个命令作为起始值，分别是 test1 、test2 、test3 。

1. 客户端发送以命令字符 `test1` 开始的消息，要求在该 `test1` 命令字符后紧跟 `command_delimiter + 其他内容 +delimiter` 。接收到该信息后，该算子会自动触发右下侧的 test1 端口，进而触发其他后续算子进行运算。

   对于命令字符 `test1` 开始的消息访问，服务端对命令字符后续的其他字符内容不做任何处理，并最终统一回复 `OK` 字符串消息。

   实际使用时，需要将该算子右下侧 test1 端口后续的触发结束信号返还连接到该算子的左侧 test1 端口，因为只有成功触发该端口才会执行 TCP 服务端的消息回复。

   客户端发送：

   ```
   test1  command_delimiter   其他内容  delimiter
   ```

   服务端回复：

   ```
   ok
   ```

2. 客户端发送以命令字符 `test2` 开始的消息，要求在该 `test2` 命令字符后紧跟 `command_delimiter+'x y z rx ry rz'+delimiter`。接收到该信息后，服务端会将命令字符后续的 'x y z rotated_x rotated_y rotated_z' 字符内容转换为 6个 float 数值，进而赋值给一个 pose。

   如果上述字符串消息可以合法转换为 6个 float 则会触发执行该算子右下侧的 test2 端口，并通过算子右下侧的 test2_pose 端口输出所得的 pose ，如果字符串消息不合法，则不进行任何后续响应。

   实际使用时，需要将该算子右下侧 test2 端口后续的触发结束信号返还连接到该算子的左侧 test2 端口，因为只有成功触发该端口才会执行 TCP 服务端的消息回复。对于命令字符 `test2` 开始的消息访问，最终统一回复 `test2` 。

   客户端发送：

   ```
   test2 command_delimiter  x y z rx ry rz delimiter
   ```

   算子右侧 test2_pose 端口输出：

   ```
   x y z rx ry rz
   ```

   服务端回复：

   ```
   test2
   ```

3. 客户端发送以命令字符 **test3** 开始的消息，要求在该 **test3** 命令字符后紧跟 **command_delimiter+其他内容 +delimiter** 。

   接收到该信息后，该算子会自动触发右下侧的 test3 端口，进而触发其他后续算子进行运算。

   实际使用时，需要将该算子右下侧 test3 端口后续的触发结束信号返还连接到该算子的左侧 test3 端口，因为只有成功触发该端口才会执行 TCP 服务端的消息回复。

   对于命令字符 **test3** 开始的消息访问，服务端对命令字符后续的其他字符内容不做任何处理，执行消息回复时，服务端会检查算子左侧的 test3_pose 端口是否被连接。如果该 pose 端口有连接并且数值有效，则会回复该 Pose 数值给客户端，但是如果该 pose 端口没有连接，则会回复空字符串。

   客户端发送：

   ```
   test3  command_delimiter  其他内容  delimiter
   ```

   算子左侧 test3_pose 有连接并且数值有效则端口则服务端回复该 Pose 数值：

   ```
   x y z rx ry rz
   ```

4. 任意其他消息，不会作任何响应。

算子参数

- **端口/port**：创建 TCP server 服务器所使用的本机服务端口。

  说明：实际运行时，如果提示 TCP 服务端创建失败，往往是所选取的端口已经被其他服务所占用，更换端口即可。如果在同一个RVS软件中，创建多个TCP服务端算子，彼此的port也要互斥。

- **连接数量/connections**：可同时支持的最大客户端连接数量。
- **服务器模式/server_mode**：运行模式。
  - Once：表示完成一次 TCP 对话以后自动断开同客户端的链接。比如客户端首先同该算子建立了TCP连接，然后第一次发送了 A 字符串消息请求并获得回复之后，继续发送 B 字符串的消息请求，此时就会报错。
  - Continous： 表示客户端建立链接后可以无限次数的对话。
- **分隔符/delimiter**：消息结束符，包含RT 换行符、“；”、“#”、“$”。当选择了其中某一种时，另外三种就会被视作普通字符随意使用 。服务端在接收客户端发送的消息时，会捕获第一个消息结束符之前的所有消息。

  说明：由于不同的通信软件、电脑系统软件、机器人操作系统软件对回车换行的定义不一致，容易导致信息发送或者接收失败，所以一般不建议选择RT作为消息结束符。

- **命令分隔符/command_delimiter**：命令结束符。当选择了其中某一种时，另外一种就会被视作普通字符随意使用。服务端在接收客户端发送的消息时，会将第一个命令结束符之前的所有消息视为命令字符，之后的消息视为内容信息。如果命令字符不是上 test1 、test2 、test3 中的任意一个，则是无效命令，不做任何后续处理与响应。
  - Space：空格。
  - , ：逗号。
- **回复结束符/arc_eof**：回传值结束符。当输入该值时，在回传值结尾添加结束符。默认为空。
- **test2_pose**：设置通过 test2 命令消息接收到的 pose 可视化属性。
  - 👁 打开 pose 可视化。

-  关闭 pose 可视化。
-  设置坐标的尺寸大小 。取值范围：[0.001,10] 。默认值：0.1 。

注意：在算子运行后，重新更改了 port 、connections 、server_mode 、delimiter 、command_delimiter 5个属性中的任意一个之后，如果需要生效，都必须重启 RVS 。

控制信号输入输出

输入：

- `start`：
  - 触发start信号端口运行算子，开始创建 TCP 服务端，创建成功后监听服务端口并等待客户端访问。

    说明：本算子仅需要初始化运行一次即可。
- `stop`：
  - 触发 stop 信号端口后，开始停止 TCP 服务端的监听。
- `reset`：
  - 该功能保留。
- `finished`：
  - 该功能保留。
- `failed`：
  - 算子运行失败后触发该端口。
- `started`：
  - 算子成功建立 TCP 服务端后触发该端口。
- `processing`：
  - 该功能保留。
- `stopped`：
  - 算子成功停止 TCP 服务端的监听后触发该端口。
- `reset`：
  - 该功能保留。

功能演示

下面分别针对test1 、test2 、test3 三类命令，进行通讯演示。

在选用演示所用的 TCP 客户端时，由于 RVS 自带的 CommonTCPClient 算子在使用时要求服务端返回的信息必须以四类固定字符结尾，而 TestTCPCommand 算子在作为 TCP 的服务端时，返回给 TCP Client 的信息是固定的并且没有额外添加上述四类固定结尾字符，所以这里不再选用 CommonTCPClient 算子作为客户端。

说明：这里的演示案例是基于 ubuntu系统给出的，使用了"echo + nc "的指令方式实现TCP客户端的功能；如果是在 Windows 版本，建议使用通讯助手等工具。

步骤1：算子准备

右击创建新 Group ，添加 Trigger 、TestTCPCommand 算子至 Group 算子图。

步骤2：设置算子参数

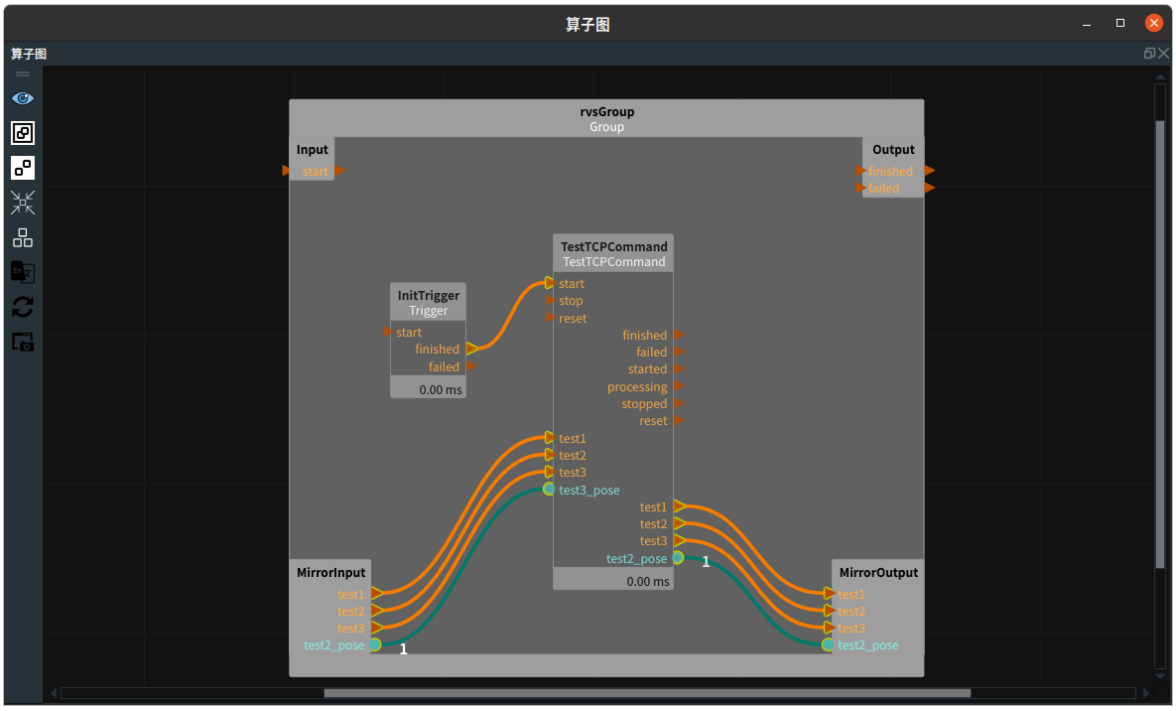
1. 设置 Trigger 算子参数：
   - 算子名称 → InitTrigger

- 类型 → InitTrigger

2. 设置 TestTCPCommand 算子参数：
   - 分隔符 → ；
   - 其余参数保持默认值。

步骤3：连接算子

TestTCPCommand算子选用了 “echo + nc” 的指令方式，所以这里每发送-接收一次消息后都要断开连接，所以选择了Once模式。

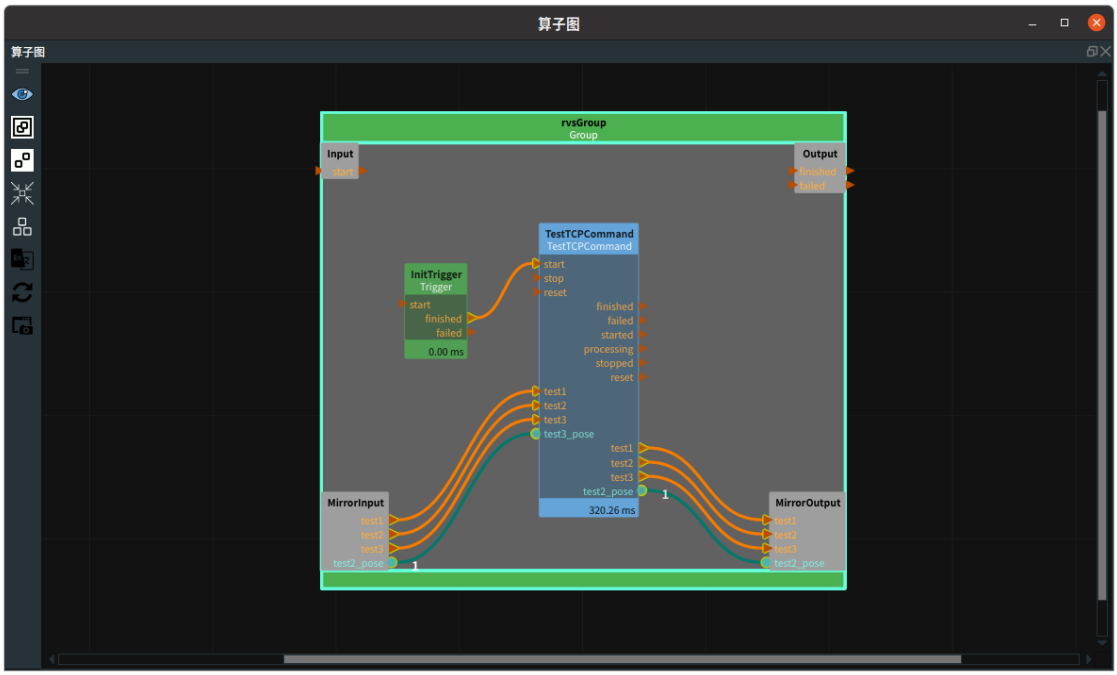


步骤4：运行

点击配置日志通道显示按钮设置对于所有通道安全级别为 debug 级别。点击 RVS 软件的运行按钮。

运行结果

1. TestTCPCommand 算子会被 Trigger 算子自动触发，并变为蓝色。

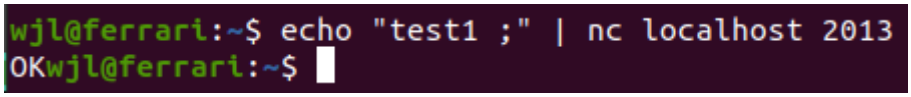


   - 日志栏同时打印算子运行说明如下图所示，表示TCP的服务端已经建立完毕。

2023-02-27 11:32:53.801237	rvs_kernel	debug	TestTCPCommand: input_trigger:start
2023-02-27 11:32:53.801259	rvs_kernel	debug	TestTCPCommand: Triggered(TRIGGER_START)
2023-02-27 11:32:53.801293	rvs_communication	debug	CommandServer::Start()
2023-02-27 11:32:53.801444	rvs_communication	debug	Server started on port: 2013, with max connection number: 2, with m_delimiter: ;

2. 重新打开一个终端，输入命令 echo "test1 ;" | nc localhost 2013 并单击回车键。

```
echo "test1 ;" | nc localhost 2013
```

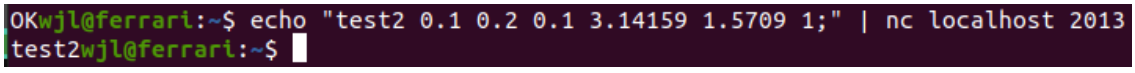


- 运行结束后，我们会在这个终端窗口看到回复字符串 ”OK“ 如下图所示，同时RVS的运行日志如下。

2023-02-27 11:33:46.643046	rvs_communication	debug	CommandService::StartHandling()
2023-02-27 11:33:46.643160	rvs_communication	debug	CommandService::onRequestReceived()
2023-02-27 11:33:46.745316	rvs_communication	debug	rvsCommandServerNode::ProcessRequest
2023-02-27 11:33:46.745371	rvs_communication	debug	first_command:test1
2023-02-27 11:33:46.745397	rvs_communication	debug	second_data:
2023-02-27 11:33:46.745415	rvs_communication	debug	Handling command: test1
2023-02-27 11:33:46.745477	rvs_communication	info	TestTCPCommand: Triggered command via server: 865: Trigger
2023-02-27 11:33:46.871724	rvs_communication	debug	TestTCPCommand: ProcessResponse for command: test1 with response: OK
2023-02-27 11:33:46.872073	rvs_communication	debug	CommandService::onResponseSent()
2023-02-27 11:33:46.872184	rvs_communication	debug	CommandService::onFinish()
2023-02-27 11:33:46.872241	rvs_communication	debug	CommandAcceptor::UnregisterCommandService()

3. 在终端中，继续输入命令 echo "test2 0.1 0.2 0.1 3.14159 1.5709 1;" | nc localhost 2013并单击回车键。运行结束后，我们会在这个终端窗口看到回复字符串”test2“如下图所示。

```
echo "test2 0.1 0.2 0.1 3.14159 1.5709 1;" | nc localhost 2013
```

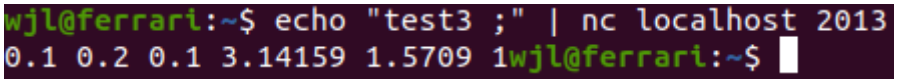


- RVS的运行日志如下。

2023-02-27 11:35:32.255456	rvs_communication	debug	CommandService::StartHandling()
2023-02-27 11:35:32.255589	rvs_communication	debug	CommandService::onRequestReceived()
2023-02-27 11:35:32.258657	rvs_communication	debug	rvsCommandServerNode::ProcessRequest
2023-02-27 11:35:32.258695	rvs_communication	debug	first_command:test2
2023-02-27 11:35:32.258715	rvs_communication	debug	second_data:0.1 0.2 0.1 3.14159 1.5709 1
2023-02-27 11:35:32.258735	rvs_communication	debug	Handling command: test2
2023-02-27 11:35:32.259362	rvs_communication	info	TestTCPCommand: Triggered command via server: 2571: Trigger
2023-02-27 11:35:32.364965	rvs_communication	debug	TestTCPCommand: ProcessResponse for command: test2 with response: test2
2023-02-27 11:35:32.365068	rvs_communication	debug	CommandService::onResponseSent()
2023-02-27 11:35:32.365085	rvs_communication	debug	CommandService::onFinish()
2023-02-27 11:35:32.365092	rvs_communication	debug	CommandAcceptor::UnregisterCommandService()

4. 在终端中，继续输入命令 echo "test3 ;" | nc localhost 2013 并单击回车键，运行结束后，我们会在这个终端窗口看到回复字符串 ”0.1 0.2 0.1 3.14159 1.5709 1“ 如下图所示。

```
echo "test3 ;" | nc localhost 2013
```



- 同时RVS的运行日志如下

2023-02-27 11:36:44.798499	rvs_communication	debug	CommandService::StartHandling()
2023-02-27 11:36:44.798665	rvs_communication	debug	CommandService::onRequestReceived()
2023-02-27 11:36:44.821041	rvs_communication	debug	rvsCommandServerNode::ProcessRequest
2023-02-27 11:36:44.821054	rvs_communication	debug	first_command:test3
2023-02-27 11:36:44.821061	rvs_communication	debug	second_data:
2023-02-27 11:36:44.821066	rvs_communication	debug	Handling command: test3
2023-02-27 11:36:44.821072	rvs_communication	info	TestTCPCommand: Triggered command via server: 3747: Trigger
2023-02-27 11:36:44.929249	rvs_communication	debug	TestTCPCommand: ProcessResponse for command: test3 with response: 0.1 0.2 0.1 3.14159 1.5709 1
2023-02-27 11:36:44.929369	rvs_communication	debug	CommandService::onResponseSent()
2023-02-27 11:36:44.929392	rvs_communication	debug	CommandService::onFinish()
2023-02-27 11:36:44.929402	rvs_communication	debug	CommandAcceptor::UnregisterCommandService()

file

DirectoryOperation 目录操作工具

DirectoryOperation 算子用于进行文件夹的读写等操作。

type	功能
ReadDirectory	读取某一个文件夹下的所有子文件夹的名称，并以 StringList 的形式输出
MakeDirectory	给定名称或路径，创建一个文件夹
ClearDirectory	清空某个文件夹内的所有文件和子文件夹
LimitDirectory	约束文件夹的占用空间大小，如果超限则自动删除最旧的文件 （所有文件按照最后一次编辑的顺序排序）

ReadDirectory

将 DirectoryOperation 算子的 type 属性选择 ReadDirectory，用于读取某一个文件夹下的所有子文件夹的名称，并以 StringList 的形式输出。

算子参数

- **父目录/parent_directory**：所需读取子文件夹的父文件夹目录。
- **倒序排列/reverse_order**：是否逆序排列。
- **目录列表/directory_list**：设置曝光属性。打开后可用于与交互面板中输出工具——“表格”控件绑定。
  -  打开曝光
  -  关闭曝光

数据信号输入输出

说明：本算子仅需要初始化运行一次即可。

- **directory_list**：
  - 数据类型：StringList
  - 输出内容：所有子文件夹的名称

功能演示

使用 DirectoryOperation 算子type 属性中 ReadDirectory，读取 example_data 目录下的所有子文件夹的名称。

步骤1：算子准备

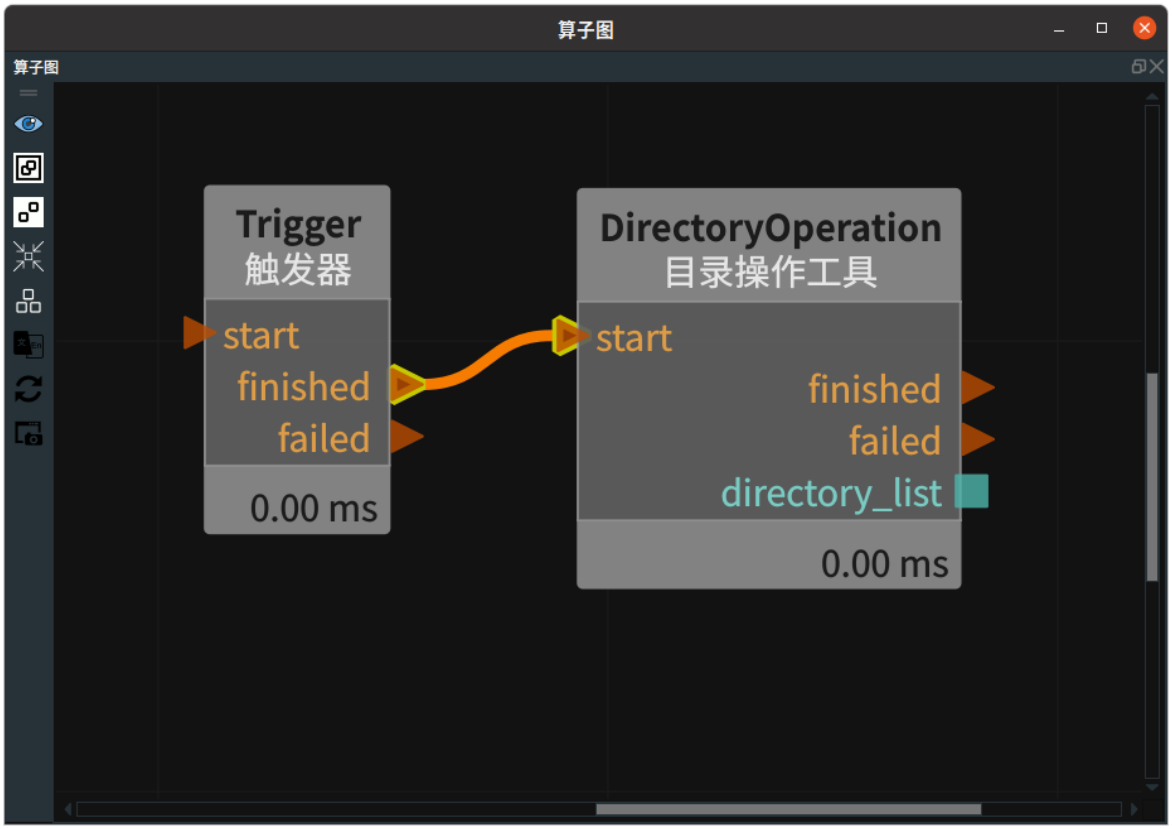
添加Trigger、DirectoryOperation 算子至算子图。

步骤2：设置算子参数

1. 设置 DirectoryOperation 算子参数：

- 类型 → ReadDirectory
- 父目录/parent_directory → ••• → 选择一个含有多个子文件夹的文件目录名 （*example_data* ）
- 目录列表/directory_list → 曝光

步骤3：连接算子

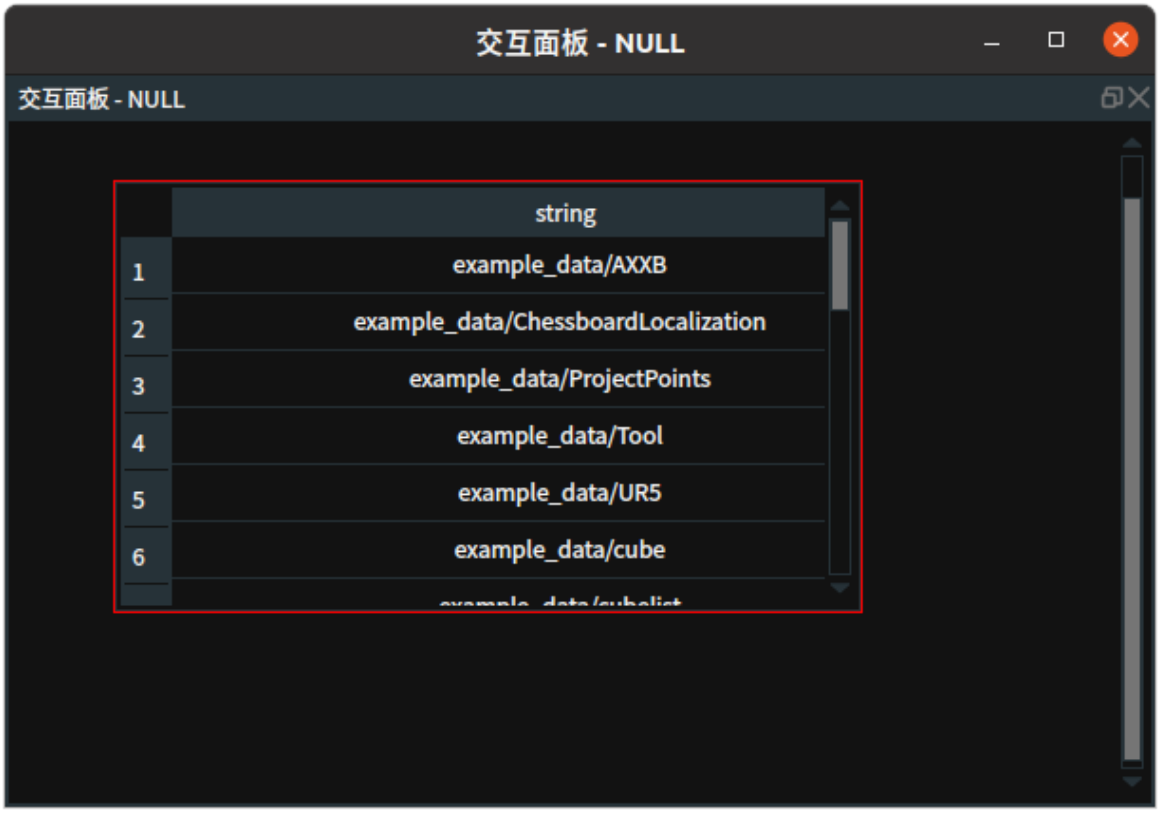


步骤4：运行算子

1. 在 RVS 的交互面板拖出输出工具中“表格”并绑定 directory_list 参数。
2. 打开 RVS 的运行按钮，触发 Trigger 算子。

运行结果

如下图所示，交互面板中显示 example_data 文件夹中所有子文件名。



MakeDirectory

将 DirectoryOperation 算子的 type 属性选择 MakeDirectory，给定名称或路径，创建一个文件夹。

算子参数

- `directory_name`：所需创建文件夹的名称。同数据输入端口的 directory_name 作用一致。当输入端口 directory_name 没有连接时，则必须给该参数赋值。当输入端口 directory_name 有连接时，则在执行算子后会自动将输入端口的数值覆盖该参数的值。
- `path_name`：设置文件夹路径名称曝光属性。打开后则可用于与交互面板中的输出工具——“文本框”绑定。
  - 打开曝光
  - 关闭曝光

数据信号输入输出

输入：

- `parent_name`：
  - 数据类型：String
  - 输入内容：所需创建文件夹的父目录

  > 说明：该端口可以不用，因为directory_name本身也可以包含父目录

- `directory_name`：
  - 数据类型：String
  - 输入内容：所需创建文件夹的名称

  说明：一般可以通过 Emit 算子（String 模式）给出。

输出：

- `path_name`：
  - 数据类型：String
  - 输出内容：所创建的文件夹路径

功能演示

使用 DirectoryOperation 算子中 MakeDirectory，创建一个名为 test 的文件夹。

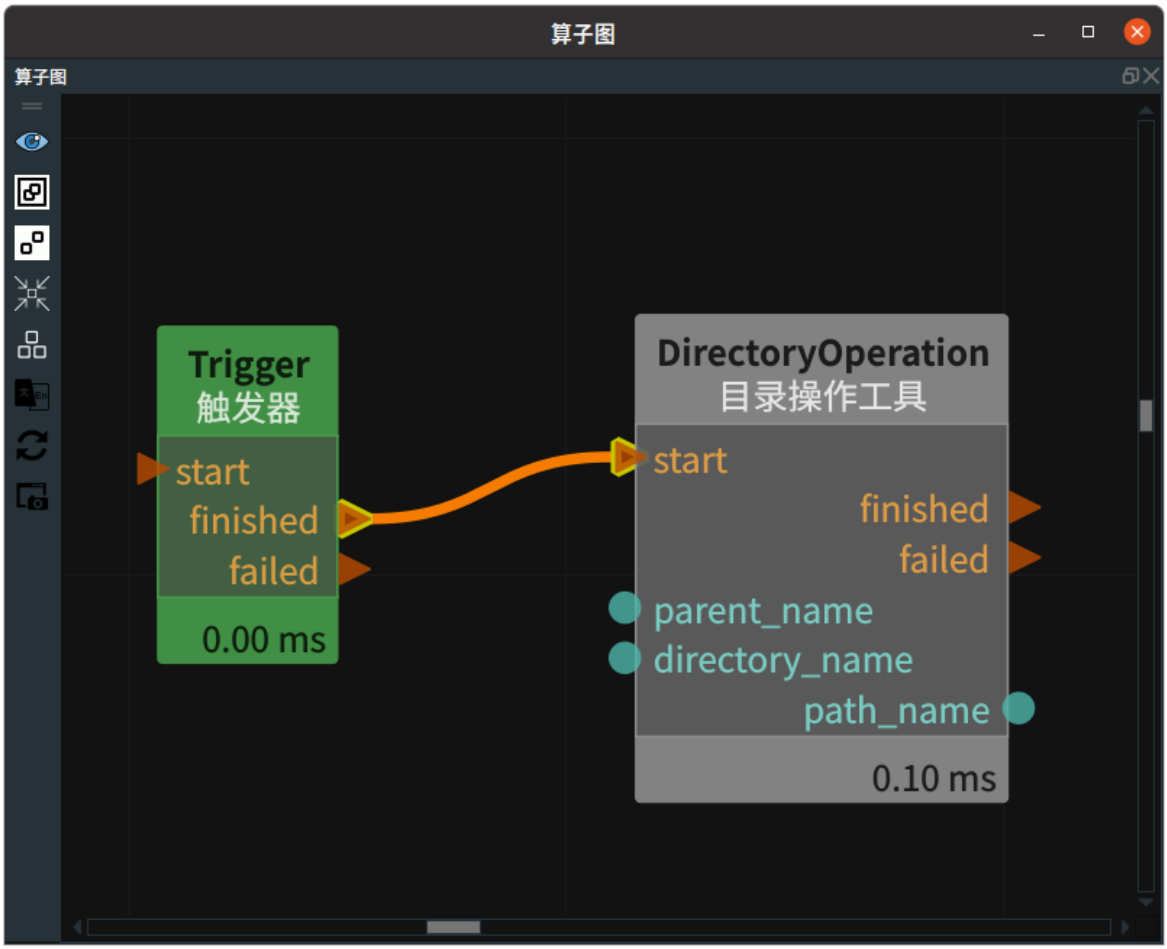
步骤1：算子准备

添加Trigger、DirectoryOperation 算子至算子图。

步骤2：设置算子参数

1. 设置 DirectoryOperation 算子参数：
   - 类型 → MakeDirectory
   - directory_name → test
   - path_name → 曝光

步骤3：连接算子



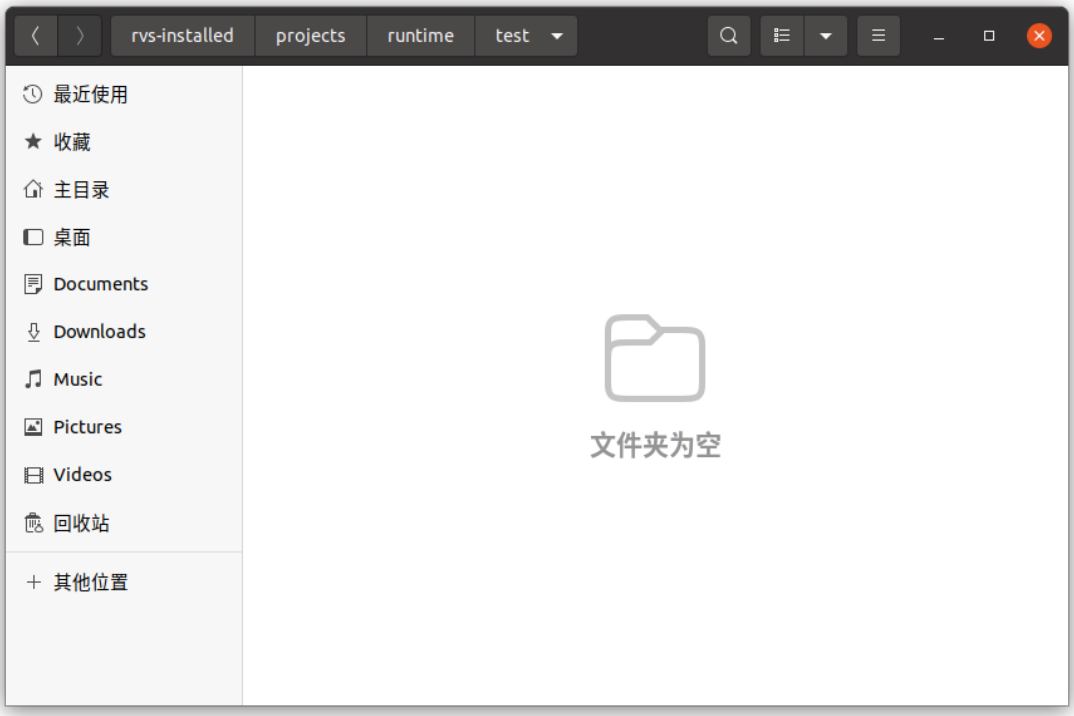
步骤4：运行算子

1. 在 RVS 的交互面板拖出输出工具“文本框”控件并绑定 path_name 参数。
2. 打开 RVS 的运行按钮，触发 Trigger 算子。

运行结果

1. 结果如下图所示，交互面板中显示创建文件夹的名称。

2. 创建的文件夹显示如下。



ClearDirectory

将 DirectoryOperation 算子的type属性选择 ClearDirectory，用于清空某个文件夹内的所有文件和子文件夹。

算子参数

- `path_name`：所需清空内容的文件夹路径。同输入端口的path_name作用一致。当输入端口path_name 没有连接时，则必须给该参数赋值。当输入端口path_name 有连接时，则在执行算子后会自动将输入端口的数值覆盖该参数的值。

数据信号输入输出

输入：

- `path_name`：
  - 数据类型：String
  - 输入内容：要清空内容的文件夹路径

功能演示

使用 DirectoryOperation 算子中 ClearDirectory 清空 test 文件夹下的所有文件和子文件夹。

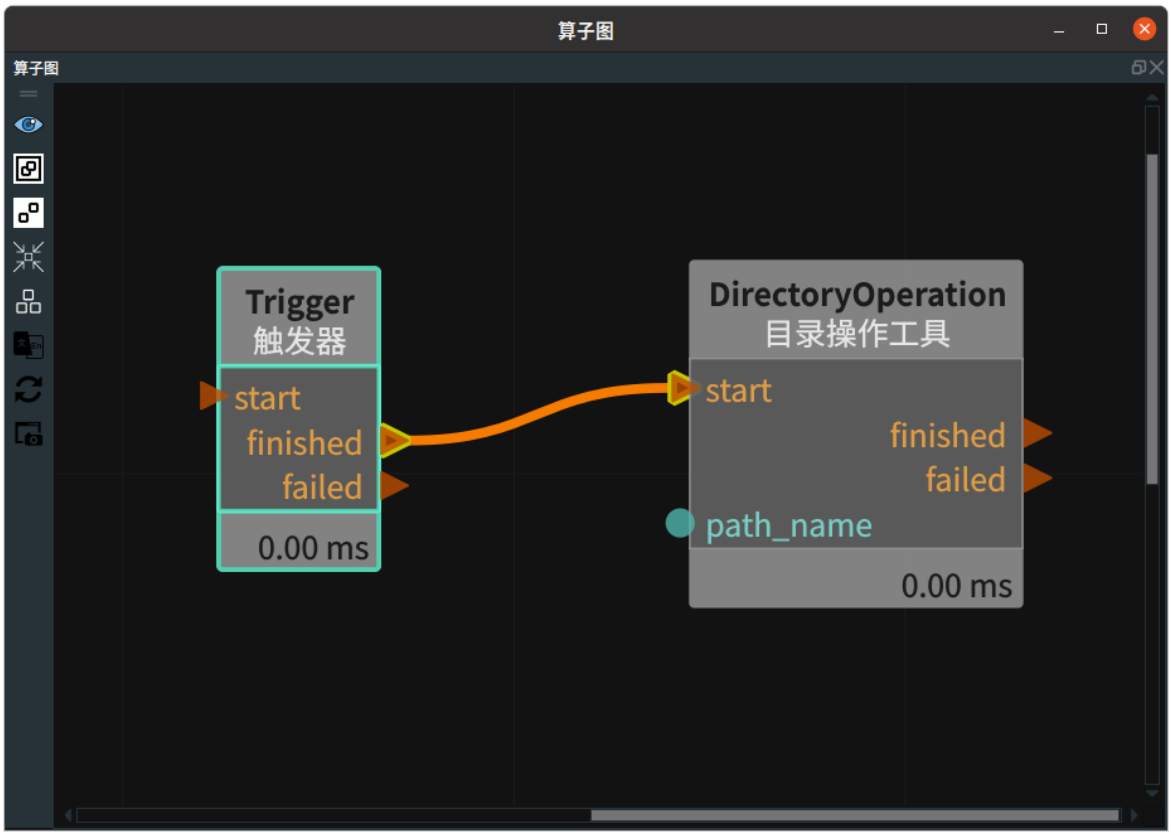
步骤1：算子准备

添加Trigger、DirectoryOperation 算子至算子图。

步骤2：设置算子参数

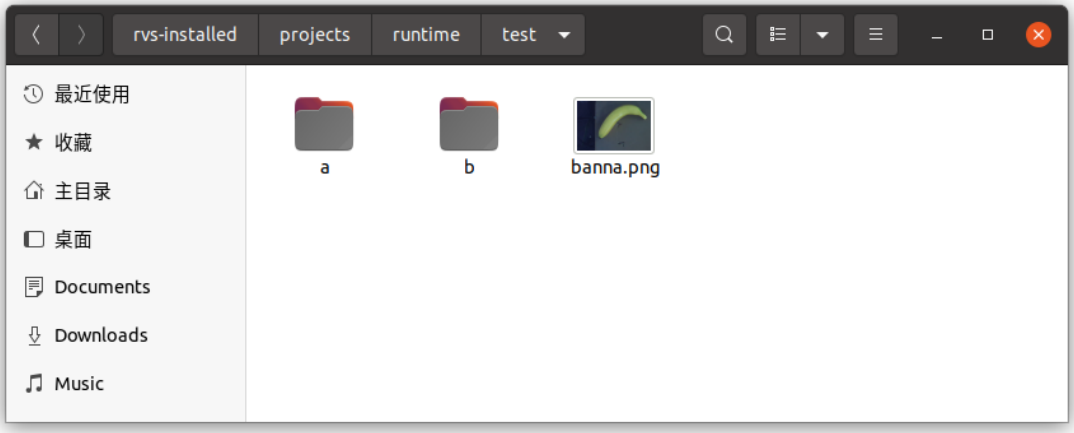
1. 设置 DirectoryOperation 算子参数：
   - 类型 → ClearDirectory
   - path_name → ●●● → 选择文件目录名（*test*）

步骤3：连接算子



步骤4：运行算子

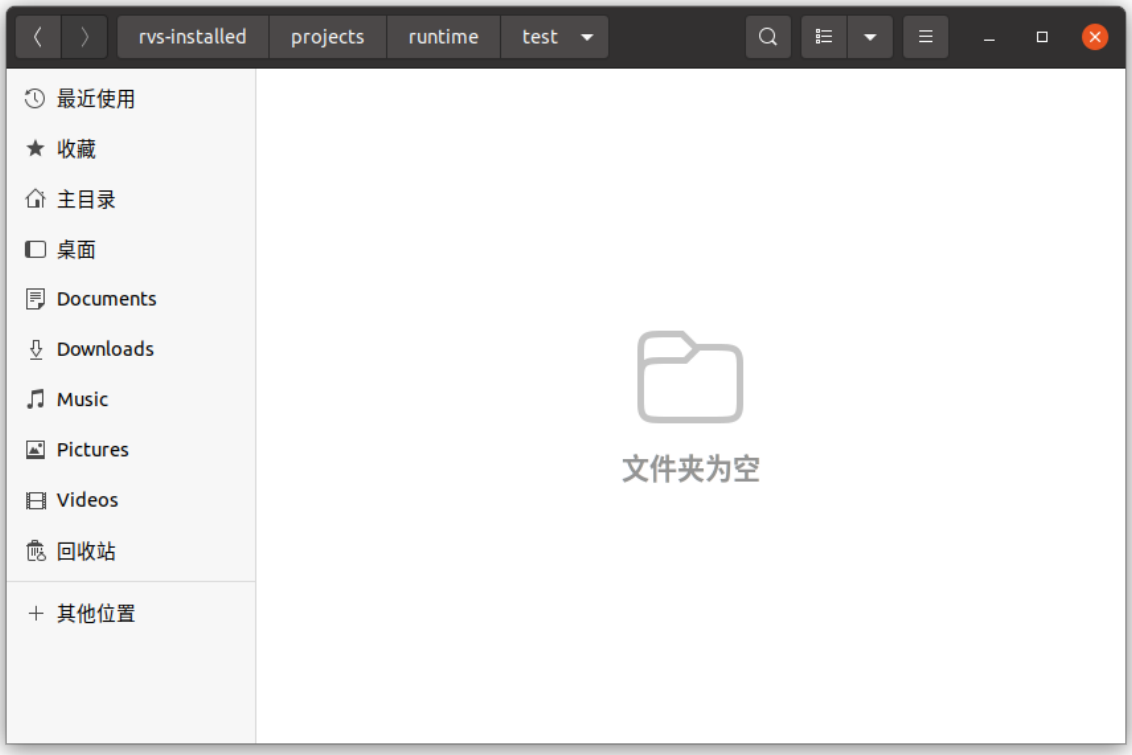
1. 在test文件夹中放置一些文件和文件夹，如下图所示。



2. 打开 RVS 的运行按钮，触发 Trigger 算子。

运行结果

结果如下图所示，文件夹 test 内的所有内容都被清空。



LimitDirectory

将 DirectoryOperation 算子的 type 属性选择 LimitDirectory，用于约束文件夹的占用空间大小，如果超限则自动删除最旧的文件（所有文件按照最后一次编辑的顺序排序）。

算子参数

- `directory_name`：文件夹地址。
- `limit_size_MB`：文件夹的最大占用硬盘空间。单位：MB 。

功能演示

使用 DirectoryOperation 算子中 LimitDirectory，约束 test 文件夹的占用空间大小为1。

说明：案例中使用到 Emit、Timestamp、Emit_1 、Join 算子用于生成文件目录名/文件名.png。如 test/20230308141759451.png。

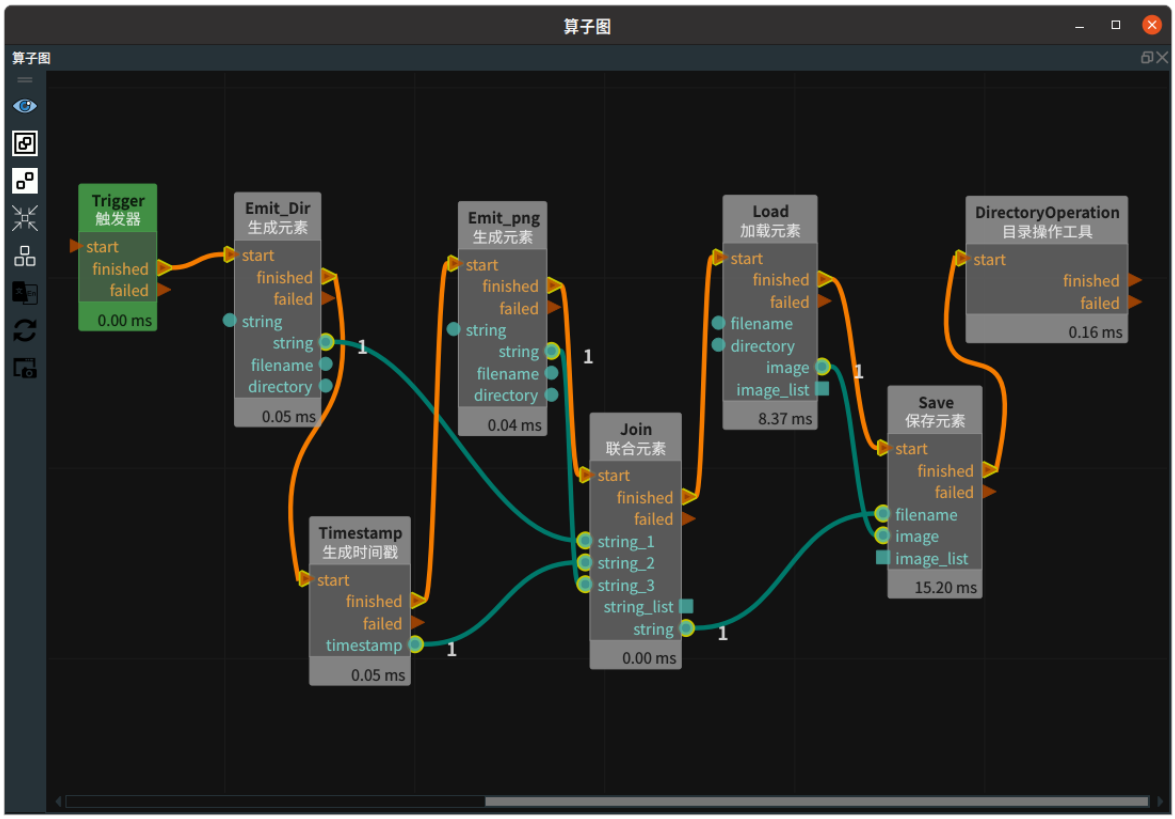
步骤1：算子准备

添加Trigger、Load、Save、Emit (2个）、Timestamp、Join、DirectoryOperation 算子至算子图。

步骤2：设置算子参数

1. 设置 DirectoryOperation 算子参数：
    - 类型 → LimitDirectory
    - directory_name → test
    - limit_size_MB → 1
2. 设置 Emit 算子参数：
    - 算子名称 → Emit_Dir
    - 类型 → String
    - 字符串 → test/ (注意：反斜杠不可省略)
3. 设置 Emit_1 算子参数：
    - 算子名称 → Emit_png
    - 类型 → String
    - 字符串 → .png
4. 设置 join 算子参数：
    - 类型 → Image
    - 输入数量 → 3
5. 设置 Load 算子参数：
    - 类型 → Image
    - 文件 → ••• → 选择图像文件名 （ *example_data/images/banana.png* ）

步骤3：连接算子



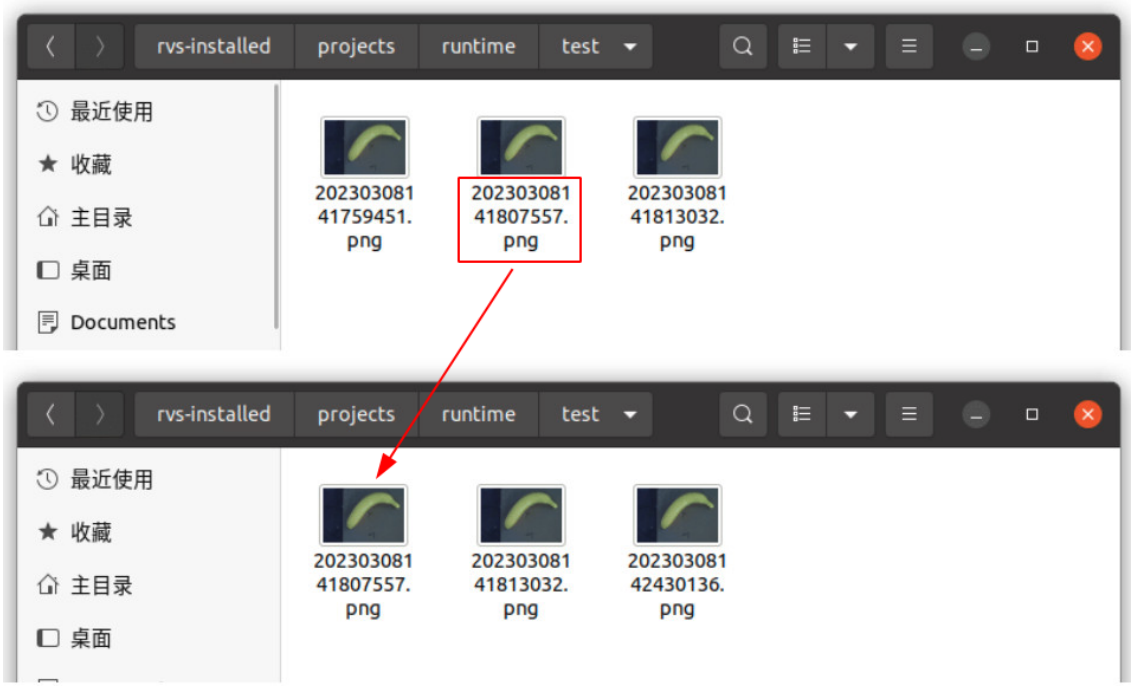
步骤4：运行

打开 RVS 的运行按钮，多次触发 Trigger 算子。

运行结果

1. 结果如下图所示，文件夹空间超限之前和超限之后的示例。可以发现触发超限之后，DirectoryOperation 算子会将文件夹内按最后一次编辑的时间顺序排序，依次将最旧的文件删除，直到满足空间尺寸限制为止。

   说明：文件名为文件生成时间。



2. 当超限之后，日志视图中也会有相应的提示信息。


日志视图
日志视图
配置日志通道显示
搜索日志条目:
清空
最大条目数:
200
时间戳
通道
安全级别
消息
2023-03-08 14:34:12.209248 rvs_basic debug output_string: .png
2023-03-08 14:34:12.209253 rvs_basic debug output_filename:
2023-03-08 14:34:12.209258 rvs_basic debug output_directory:
2023-03-08 14:34:12.209273 rvs_kernel debug Join: input_trigger:start
2023-03-08 14:34:12.209300 rvs_kernel debug Load: input_trigger:start
2023-03-08 14:34:12.213663 rvs_basic info image loaded successfully: banna.png
2023-03-08 14:34:12.213710 rvs_kernel debug Save: input_trigger:start
2023-03-08 14:34:12.222919 rvs_kernel debug DirectoryOperation: input_trigger:start
2023-03-08 14:34:12.222974 rvs_file info checking directory: size is:1053520 (1MB)
2023-03-08 14:34:12.222982 rvs_file debug >> [DIRECTORY SIZE REACHED LIMIT]
2023-03-08 14:34:12.223045 rvs_file info checking directory: size is:790140 (0MB)
2023-03-08 14:34:12.223051 rvs_file debug >> [DIRECTORY SIZE BELOW LIMIT]

LogMessage 生成日志

LogMessage 算子用于将一段字符串信息写入到日志文件中，同时会显示在 RVS 软件界面的 Log 区域。

算子参数

- `日志等级/log_level`：写入到日志时的标志等级，分别为 fatal 、error 、warning 、 info 、debug 、 trace 。不同的输出级别在RVS软件界面的 Log 区域显示时，对应不同的颜色显示。
  - fatal：致命。背景显示红色。
  - error：错误。背景显示红色。
  - warning：警告。背景显示黄色。
  - info：信息。默认背景颜色。如果输出的字符串内容以“[H] ”开头，会变更为蓝色高亮。
  - debug：调试。默认背景颜色。
  - trace：跟踪。默认背景颜色。
- `日志内容/log_content`：需要输出的日志内容。

功能演示

使用 LogMessage 算子将 RVS_doc test 写入到日志文件中，查看在 RVS 软件界面的 Log 区域中不同日志等级的显示。

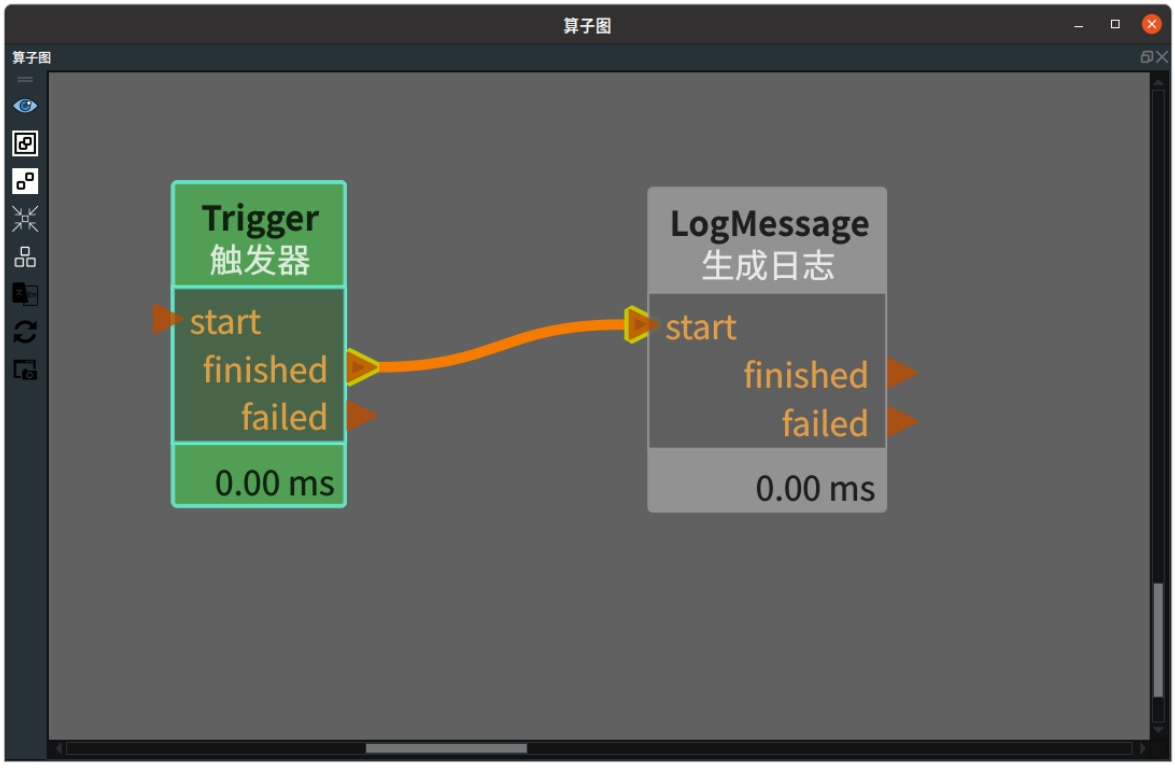
步骤1：算子准备

添加 Trigger、LogMessage算子至算子图。

步骤2：设置算子参数

1. 设置 LogMessage 算子参数：
   - type → fatal
   - log_content → RVS_doc test

步骤3：连接算子



步骤4：运行

1. 打开 RVS 的运行按钮，触发 Trigger 算子。
2. 更改 LogMessage 算子的 type 选择为 “error”、“warning”、“info”、“debug”，分别重新触发 Trigger 。

运行结果

1. 结果如下图所示，日志视图中分别展示了不同日志等级显示的颜色。

2. 如果选择 info 模式，同时将 log_content 内容更改为 “[H] RVS_doc test" ，效果如下所示。

Timestamp 生成时间戳

Timestamp 算子用于从本地读取当前时间戳并以字符串的形式输出。格式：年月日时分秒毫秒。

算子参数

- **时间戳/timestamp**：设置时间戳曝光属性。打开后可用于与交互面板中输出工具“文本框”控件进行绑定。
  -  打开曝光。
  -  关闭曝光。

数据信号输入输出

输出：

- **timestamp**：
  - 数据类型：String
  - 输出内容：17位时间戳。例如：20230213091802410 的实际时间为 2023 年 2 月 13 日 9 时 18 分 2 秒 410 毫秒。

功能演示

该算子常用于创建文件夹或文件名，具体用法的请参考 DirectoryOperation 算子在 LimitDirectory 模式下的示例。

FileExists 判断文件存在

FileExists 算子用于根据文件地址（包含文件名）来判断文件是否存在。

算子参数

- `文件/filename`：文件地址。同输入端口的 input_filename 作用一致。当输入端口 input_filename 没有连接时，则必须给该 filename 参数赋值。当输入端口input_filename 有连接时，则在执行算子后会自动将输入端口的数值覆盖该 filename 参数的值。

数据信号输入输出

输出：

- `input_filename`：
  - 数据类型：String
  - 输出内容：文件地址

功能演示

使用 FileExists 算子判断文件是否存在。

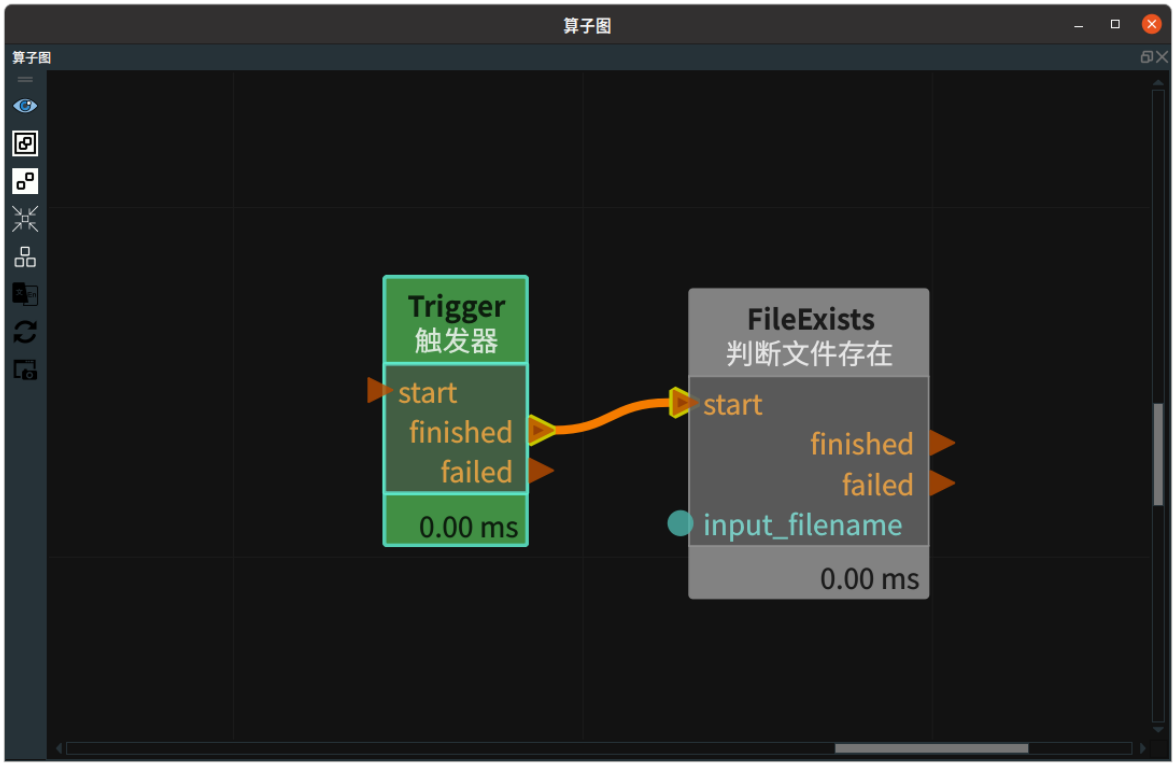
步骤1：算子准备

添加 Trigger、FileExists算子至算子图。

步骤2：设置算子参数

设置 FileExists 算子参数：文件 → cloud.pcd

步骤3：连接算子

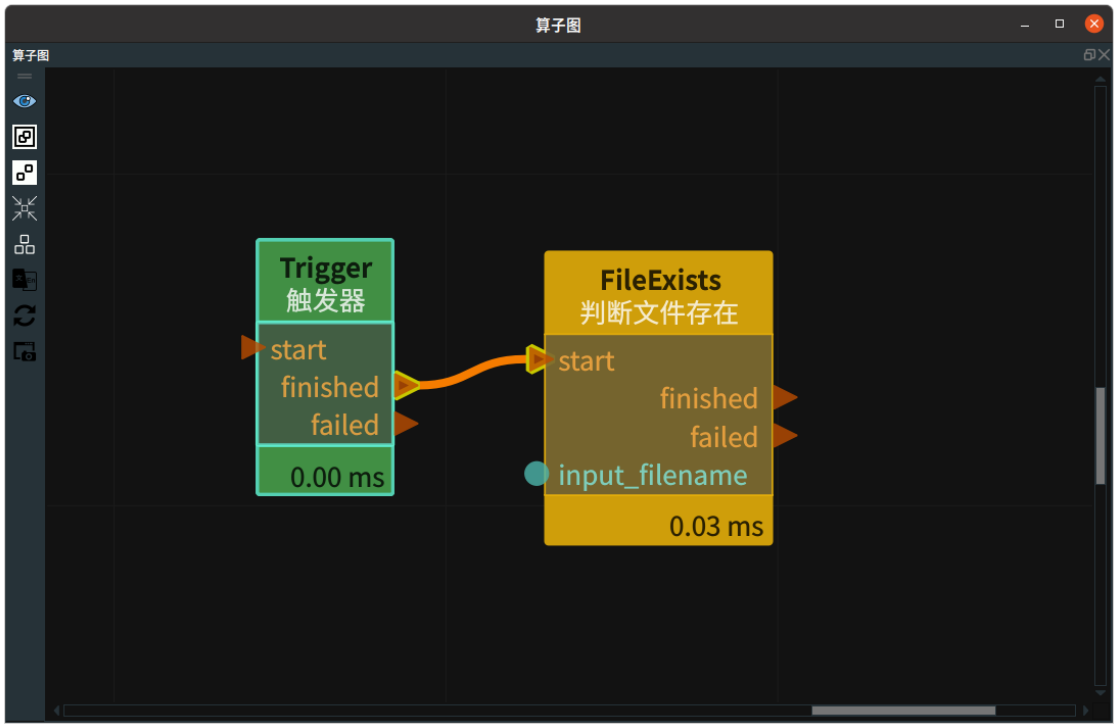


步骤4：运行

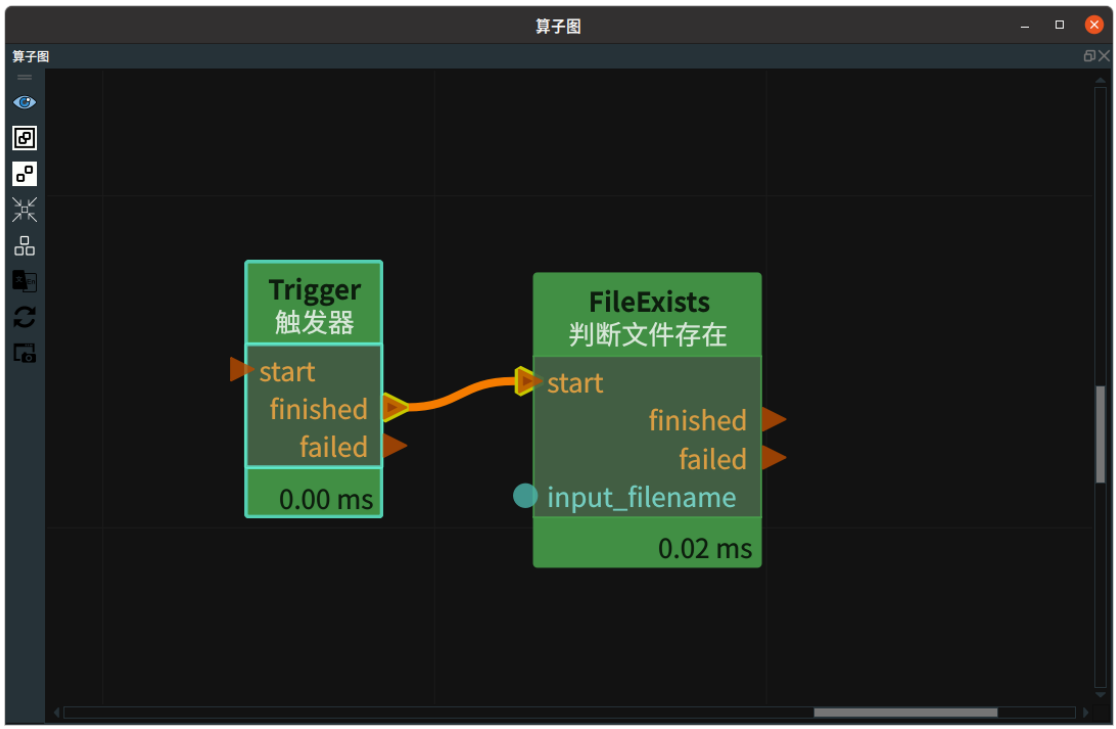
打开 RVS 的运行按钮，触发 Trigger 算子。

运行结果

1. 结果如下图所示，当输入的文件不存在时，触发 FileExists 算子的 failed 端口。



2. 结果如下图所示，当输入的文件存在时，触发 FileExists 算子的 finished 端口。



icp

ICP ICP算法

ICP 算子用于点云模板匹配。

算子参数

- **最大相关点距离/max_correspondence_distance**：目标点云与模板点云匹配时两个匹配点之间的最大空间距离。默认值：0.05。

  说明：超过该距离认为该组匹配点无效。过多的无效匹配意味当前的匹配效果很差。

- **RANSAC异常值否决阈值/ransac_outlier_rejection_threshold**：去除目标点云指定值外的杂点。默认值：0.05。
- **平移精度/trans_epsilon**：两次相邻迭代之间的平移变换矩阵的最大调整值。默认值：0.000001。
- **旋转精度/rot_epsilon**：两次相邻迭代之间的旋转变换矩阵的最大调整值。默认值：0.000001。

  说明：通常 trans_epsilon与rot_epsilon的值一致。

- **最大迭代次数/max_iteration**：ICP 算子会在所有的初始评估值的基础上，分别进行不断的迭代优化，每一个初始评估值的最大优化迭代次数是**最大迭代次数** 对应的数值。默认值：1000。
- **得分阈值/score_threshold**：分数阈值，得分越低，匹配效果越好。如果最终的最优匹配结果得分超过该阈值，算子触发 failed 信号。默认值：-1，表示在计算结果中取最优值。
- **结果坐标/result_pose**：设置 ICP 算子结果坐标在 3D 视图中的可视化属性。
  -  打开结果坐标可视化。
  -  关闭结果坐标可视化。

数据信号输入输出

输入：

- **source_cloud**：
  - 数据类型：PointCloud
  - 输入内容：模板点云
- **target_cloud**：
  - 数据类型：PointCloud
  - 输入内容：目标点云
- **initial_guess_list**：
  - 数据类型：PoseList
  - 输入内容：初值推测值

输出：

- **result_pose**：
  - 数据类型：Pose
  - 输出内容：最优结果坐标

功能演示

请参考 模板匹配 教程。

python

PythonSimple Python算子

PythonSimple 算子用于在RVS的主线程中执行 python 脚本。

PythonSimple 算子从属于普通算子，运行会占用主线程，多个 PythonSimple 算子即便并行连接，实际执行时还是先后依次运行。

> 注意：该python脚本需要在 GeneratePython 算子生成的模板中进行编写。

算子参数

- **Python配置文件/python_config_file**：定义 python 脚本同 RVS 交互时的输入输出端口。

  注意：必须同所需执行的 python 脚本命名一致。

- **重置reinit函数/re_ini**：bool 变量。默认 True。
  - True：在算子执行时，调用一次 python 脚本中的 re_ini 函数(执行顺序在 process 函数之前)，算子触发一次以后 re_ini 变量自动重置为 False 状态。
  - False：在算子执行时，不调用python 脚本中的 re_ini 函数。

功能演示

基于 GeneratePython 算子的案例演示，这里使用 PythonSimple 算子对一张测试图进行缩放处理。

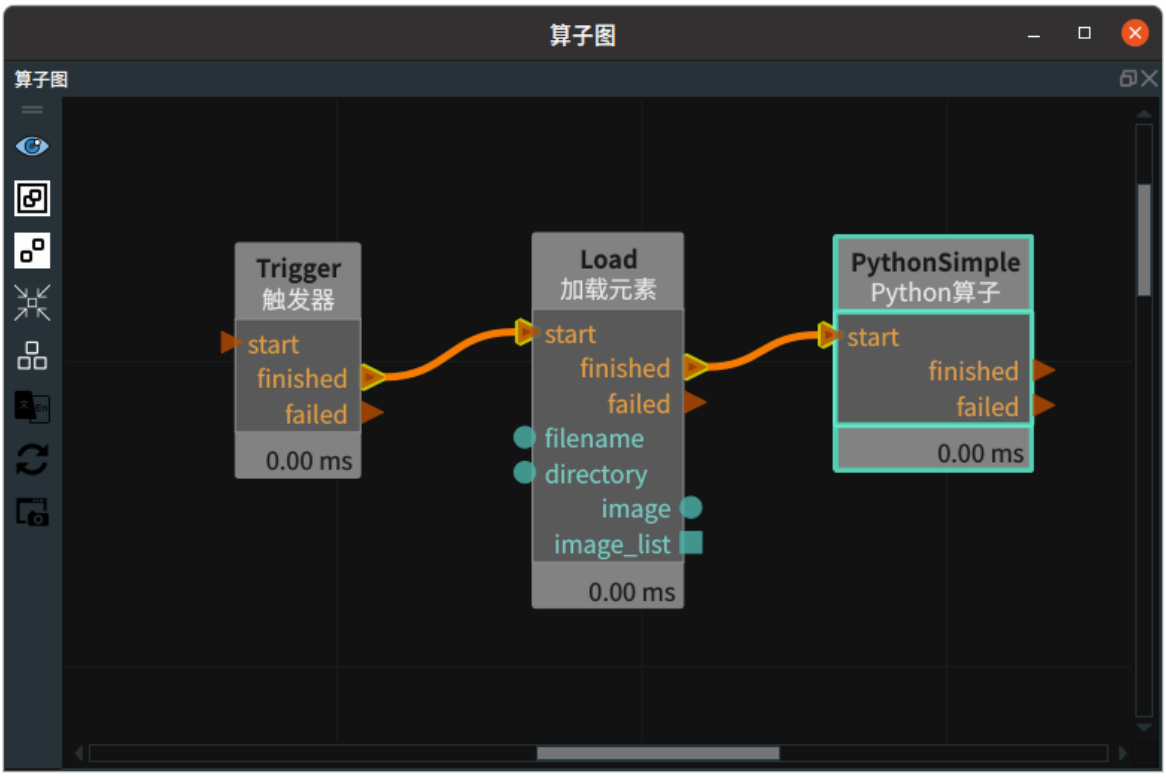
步骤1：算子准备

添加 Trigger、Load、PythonSimple 算子至算子图。

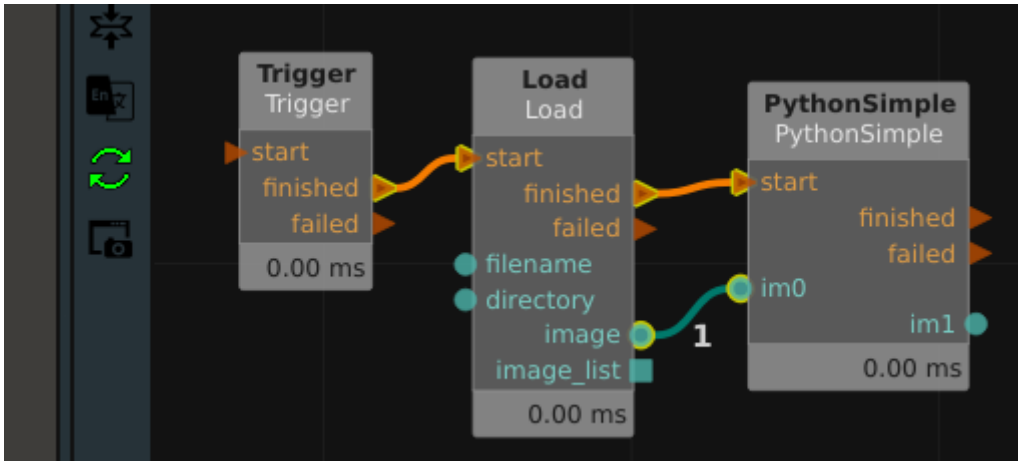
步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型→ Image
   - 文件 → ••• → 选择图像文件名 (example_data/images/bird.png)
   - 图片→ 可视
2. 设置 PythonSimple 算子参数：
   - Python配置文件 → example_data/python_text.xml
   - im1→ 可视

步骤3：连接算子



手动点击 RVS 的算子图区域左侧的 “刷新” （见下图左侧绿色点亮按钮）。此时 PythonSimple 算子会刷新出对应的端口与属性，按照下图连线。



步骤4：运行

点击 RVS 的运行按钮，触发 Trigger 算子。

运行结果

1. 如下图所示，2D 视图中分别显示 Load 、PythonSimple 算子运行成功后的结果。

2. 在日志视图中显示 python_file。

PythonThread Python线程算子

PythonThread 算子用于在 RVS 的分线程中执行 python 脚本。

由于 PythonThread 算子从属于 Thread 类算子，所以多个并列的 PythonThread 算子可以同时运行。

而 PythonSimple 算子从属于普通算子，运行会占用主线程，所以多个 PythonSimple 算子即便并行连接，实际执行时还是先后依次运行。

关于RVS中算子并行连接时的执行顺序的简单说明：

1. RVS 中的 ThreadNode 结尾的算子都是线程类算子，运行时间较长。（下文说明基于普通算子的运行时间较短的情况）
2. Thread 算子在被触发后，会占用主线程极短暂的时间，然后进入分线程执行功能函数，释放主线程，从而可以继续触发执行下一个算子。
3. 在某一个并行连接结构中，比如算子 A 触发算子 B ，算子 B 触发算子 C ； 且算子 A 触发算子 D ，算子 D 触发算子 E 。无论 B 或者 D 两者中是否含有 thread算子，RVS 的执行都是等两者全部运行完成，才会去考虑执行 C 或者 E（此时 C/E 虽然不是源自同一个起点，但是执行的先后顺序也是按照并行连接来考虑）。
4. 当 B 和 D 都是 thread 算子时，两个并行的 thread 可以同时运行，此时能有效节省运行时间。比如 B 运行 3 秒，D 运行 5 秒，则两者的并行连接只需要运行 5s 就可以执行到 C/E 算子，而串形连接需要运行 8s 才能执行到 C/E 。
5. 将一个 thread 算子同一个普通算子并行连接，理论上只能节省该普通算子的运行时间，意义不大，不建议这种连接方式。
6. 多个算子(普通算子或者线程算子)并行连接，按照算子名称的字符串从小到大的顺序，依次触发。

注意：该 python 脚本需要在 GeneratePython 算子生成的模板中进行编写。

算子参数

- **Python配置文件/python_config_file**：定义 python 脚本同 RVS 交互时的输入输出端口。

  > 说明：必须同所需执行的 python 脚本保持命名一致。

- **重置reinit函数/re_ini**：bool 变量，默认 True。
  - True：在算子执行时，调用一次 python 脚本中的 re_ini 函数(执行顺序在 process 函数之前)，算子触发一次以后 re_ini 变量自动重置为 False 状态。
  - False：在算子执行时，不调用python 脚本中的 re_ini 函数。

功能演示

本节将使用 PythonThread 进行线程算子案例演示。这与 PythonSimple 算子中展现的演示方法相同，请参照该章节的功能演示。

RotatedYOLO YOLO推理

RotatedYOLO 算子用于使用 cpu 对 RotatedYOLO 神经网络模型进行推理。RotatedYOLO网络是基于YOLO（You Only Look Once）网络的改进版，用于检测倾斜的物体。

> 说明：RVS 的 GPU/CPU 版本均可使用该算子。

type	功能
YOLO	用于 RotatedYOLO 网络的推理，可以获得图像中目标的位置掩码(像素区域) 这里的掩码区域实际是目标的旋转矩形外边框，所以掩码图中也会包含一小部分的背景。
YOLOClass	与 YOLO 模式描述一致。 同 YOLO 模式的区别：输出的目标位置掩码按照类别划分，每个类别输出一张 mask 图。 对于同一个类别中多个目标之间的重叠像素区域，将该区域判定给得分更高的目标。

YOLO

算子参数

- **权重文件路径/weight_file_path**：训练完成后的权重文件地址。
- **图像尺寸/image_size**：推理时的图像尺寸。应该同训练时设置尺寸保持一致。
- **物体得分阈值/obj_score_threshold**：目标得分阈值，得分低于阈值的目标会被淘汰，范围取值：[0,1] 。默认值：0.3。
- **IOU阈值/iou_threshold**：重叠目标交叠阈值，若两个目标的重叠度高于阈值，则其中得分较低的目标会被淘汰。当推理结果中，出现了较多“同一个目标有多个推理框”的现象时，说明这些推理框虽然有交叠但是交叠度低于阈值以至于没有被移除，此时可以适当降低该阈值。但阈值不应过低，否则容易将两个相互靠近的真实目标的推理框误移除。
- **使用设备/device**：推理时选用的物理设备，包含 gpu 和 cpu 两个选项。

  注意：在CPU 版本的 RVS 中仅能选用 cpu 。
- **重置/reset**：参数重置。在执行了一次推理之后，如果对上述参数中的任何一个进行了修改，则还需要勾选该 reset 参数，否则上述参数的更改不会生效。重新触发执行一次算子，该参数会自动由勾选状态重新变为非勾选状态。
- **识别结果图像/show_result**：设置推理结果示意图在 2D 视图中的可视化属性。
  -  打开推理结果示意图可视化。**打开能更直观显示推理结果，利于调试。**
  -  关闭推理结果示意图可视化。**关闭可视化可以缩短算子对结果图像进行着色处理的时间。**
- **Mask列表/mask_list**：是否显示 mask_list 输出端口的图像内容在 2D 视图中的可视化属性。其中每一张图像都是一张灰度图，背景像素为 0，目标区域为 255。
- **Mask名称列表/mask_names**：设置目标对应的类名曝光属性。勾选后则可以将 mask_names 输出端口的内容绑定到交互面板上的表格并输出显示。
  -  打开曝光。
  -  关闭曝光。

数据信号输入输出

输入：

- `image`：
  - 数据类型：Image
  - 输入内容：需要推理的图像数据

输出：

- `show_result`：
  - 数据类型：Image
  - 输出内容：图像推理结果的效果示意图
- `mask_list`：
  - 数据类型：ImageList
  - 输出内容：图像推理后获得的所有目标的掩码图，每个目标在List中的位置同下述mask_names保持一致。各个目标按照对应的得分从高到低排列。
- `mask_names`：
  - 数据类型：StringList
  - 输出内容：以String形式存放各个目标对应的类名

功能演示

使用 RotatedYOLO 算子对训练好的数据进行推理。

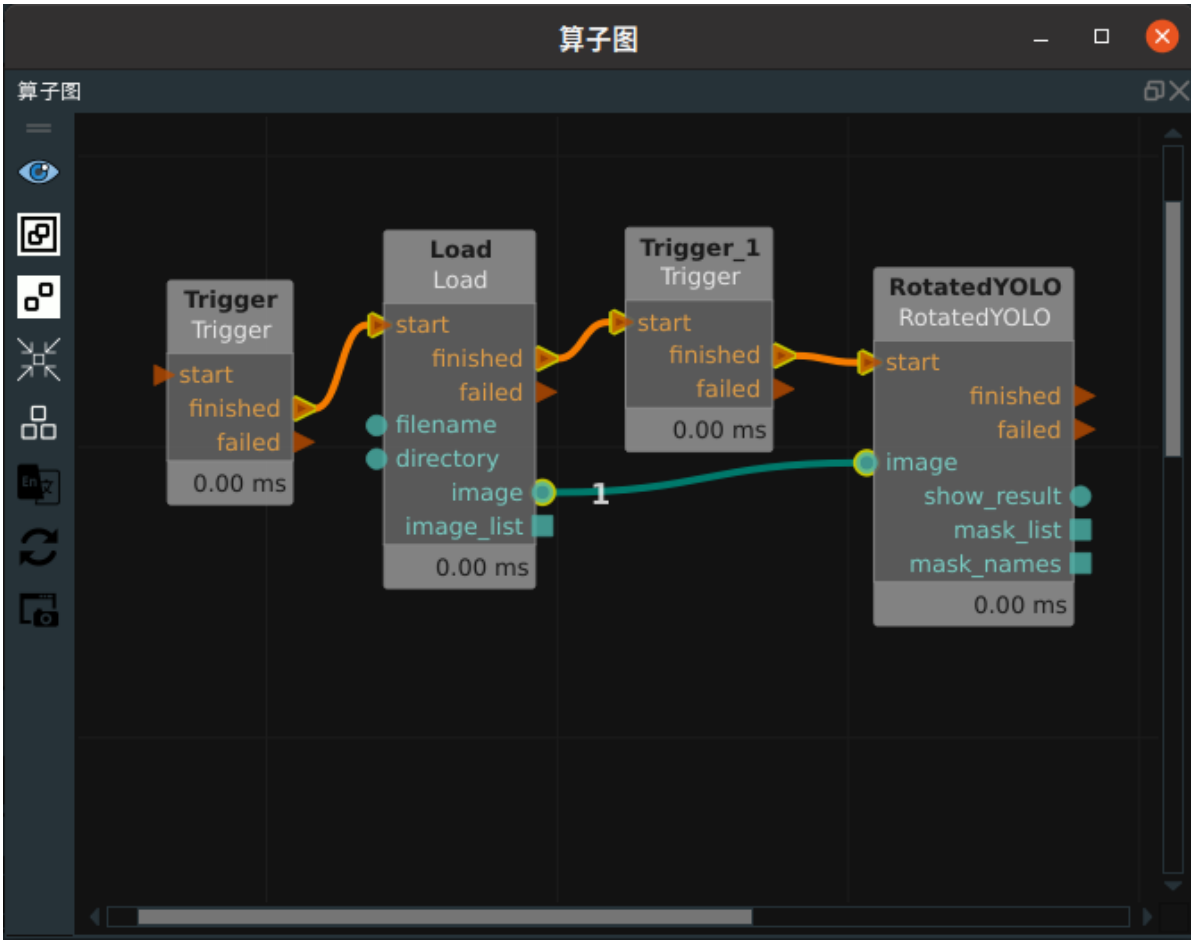
步骤1：算子准备

添加 Trigger(2 个）、Load、RotatedYOLO 算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → Image
   - 文件 → ••• → 选择图像文件名 (*example_data/yolo_data/20210119150022784/rgb.png*)
   - 图像 → 可视
2. 设置 Trigger_1 算子参数：类型 → InitTrigger
3. 设置 RotatedYOLO 算子参数：
   - 权重文件路径 → 训练完成后的权重文件地址 (*example_data/yolo_data/used_data/train_output/weights/last.pt*)
   - 图像石村→ 640
   - 物体得分阈值→ 0.3
   - IOU阈值→ 0.4
   - 使用设备→gpu
   - 识别结果图像 → 可视

步骤3：连接算子



步骤4：运行算子与结果展示

1. 点击 RVS 的运行按钮，XML 会自动运行第二个 trigger（type 为 InitTrigger ）。此时会触发 RotatedYOLO 算子完成第一次的初始化运行（首次运行不会对输入的图像进行推理，仅仅是运行环境检测以及将模型文件从本地载入到内存），运行成功后界面显示如下图，在日志栏依次打印了 "YOLO is Loading module"..."GetClassNames done" 等四条语句。

时间戳	通道	安全级别	
2023-01-28 16:51:15.867236	rvs_kernel	debug	RotatedYOLO: input_trigger:start
2023-01-28 16:51:15.969209	rvs_python	info	YOLO is Loading module
2023-01-28 16:51:17.446808	rvs_python	info	module loaded : YOLO_detect
2023-01-28 16:51:20.011039	rvs_python	info	YOLO initialised
2023-01-28 16:51:20.011122	rvs_python	info	GetClassNames done

2. 在推理过程中，会在日志栏高亮打印 “YOLO is processing” 的输出信息，推理完成后，会在日志栏打印该图像推理得到的目标个数，如下图所示。

时间戳	通道	安全级别	
2023-01-28 16:53:44.598040	rvs_kernel	debug	Trigger_1: input_trigger:start
2023-01-28 16:53:44.598062	rvs_basic	info	Trigger_1 was triggered
2023-01-28 16:53:44.598070	rvs_kernel	debug	RotatedYOLO: input_trigger:start
2023-01-28 16:53:44.598085	rvs_python	info	YOLO is processing
2023-01-28 16:53:44.641541	rvs_python	debug	YoloObb number of detection: 3

3. 推理完成后，如果勾选了 RotatedYOLO 算子的 show_result 参数，可以在 RVS 的 2D 图区域看到推理结果示意图，如下所示。

YOLOClass

算子参数

- **Mask类别名称/mask_class_list**：设置 mask_class_list 输出端口的图像内容在 2D 视图中的可视化属性。其中每一张图像都是一张灰度图，代表某一个类别的所有目标的掩码图。mask_class_list中的类别，按照类名文件中类别的先后顺序进行排序。每张mask图像的背景像素为 0，每一个目标按照得分从大到小的顺序依次使用1、2、3...等像素。比如一张mask图中像素为1的一块区域，代表该类别中得分最高的目标的像素位置。
- **其余参数**：与 YOLO 一致。

数据信号输入输出

输入：

- **image**：
  - 数据类型：Image
  - 输入内容：需要推理的图像数据

输出：

- **show_result**：
  - 数据类型：Image
  - 输出内容：图像推理结果的效果示意图
- **mask_class_list**：
  - 数据类型：ImageList
  - 输出内容：图像推理后获得的所有类别目标的掩码图，详细说明见上述“算子参数”--mask_class_list的介绍。
- **mask_names**：

- 数据类型：StringList
- 输出内容：以String形式存放所有的类名，顺序同类名文件保持一致，并且同上述mask_class_list保持一致。

功能演示

与上述 YOLO 模块功能演示一致。

GeneratePython 生成Python算子

GeneratePython 算子用于创建可供 RVS 执行的python脚本模板。

算子参数

- **python_config_file**：xml 文件，定义了所创建的python脚本同RVS交互时的输入输出端口，算子执行成功后会自动创建一个同名的python脚本文件。

功能演示

使用 GeneratePython 算子创建可供 RVS 执行的 python 脚本模板。

步骤1：文件准备

准备 xml 配置文件 python_test.xml。

注意：下面案例使用了 “Image” 的类型数据，当前 python 算子可支持的所有数据类型以及使用说明详见二次开发文档和样例

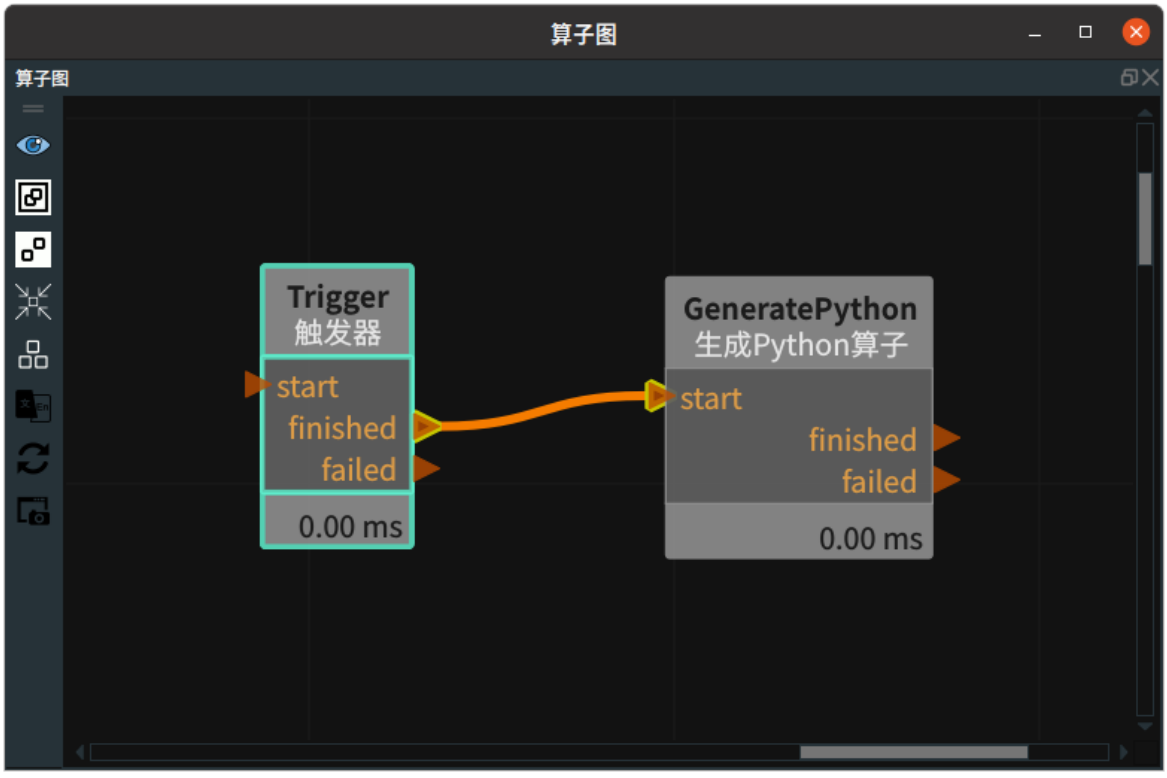
内容如下所示。

```
<config>
<input type="Image"> im0 </input>
<output type="Image"> im1 </output>
</config>
```

步骤2：算子准备

1. 添加 Trigger、GeneratePython算子至算子图。
2. 设置 GeneratePython 算子参数：python_config_file → example_data/python_text.xml

步骤3：连接算子



步骤4：运行

点击 RVS 的运行按钮，触发 Trigger 算子完成一次运行。如果使用了 Windows 版本的 RVS ，或者在 Linux 版本中通过终端来启动 RVS，则执行成功时还会在客户端打印信息，如下图所示。（第二行表示文件夹下已经有了同名文件，则新生成的文件会对旧文件进行覆盖）

```
config_xml file python_test.xml finished reading.
Func generate para: python_file is already existed, and the old file will be removed
```

运行结果

运行结束后在同名目录（example）下生成一个 python_test.py 文件， python 模板文件内容如下图所示。

注意：该模板中自动导入了RVS_LOG模块: **import RVS_LOG** 使用方法是 **RVS_LOG.write("any string")** 在该python脚本被python算子调用时，每执行到一次 **RVS_LOG.write("any string")** 均会在RVS程序的log栏高亮显示这条“any string”语句，同时写入到本地RVS的日志文件中。

打开(O) python_test.py /work/rvs_core/projects/runtime 保存(S)

```python
# Note: this file is a template python file and it was generated from config_file :python_test.xml.
##############################################
import RVS_LOG
# write log to RVS_log_file, for example:
# RVS_LOG.write("your own log string")
import numpy
# ToDo: import the moudle you need


# The following class 'RVSPyClass' is the interface-class between your python_moudle and RVS_python_node.
class RVSPyClass:
  def __init__(self):
    # all io_para_name and it's type/size have been list as below.
    self.im0 = numpy.ndarray(shape=(0,0,0)) # dtype=numpy.uint8 or numpy.uint16
    self.im1 = numpy.ndarray(shape=(0,0,0)) # dtype=numpy.uint8
    ##########################################
    # ToDo: remove 'pass' and add your code for initial
    pass


  # Must not do any change of the func inputData as below.
  def inputData(self,input):
    self.im0 = input[0]
    pass

  # Must not do any change of the func outputData as below.
  def outputData(self):
    return [ self.im1 ]

  def re_ini(self):
    ##########################################
    # ToDo: remove 'pass' and add your code
    pass


  def process(self):
    ##########################################
    # ToDo: remove 'pass' and add your code
    pass

  ##############################################
  # ToDo: add your own function


##############################################
# ToDo: add your own code
```

3. 在python脚本中增加 ”import cv2" ，并修改上述文件的 process 函数，实现将原图进行缩放的效果，同时打印日志到RVS，代码如下图所示。后续关于 python 脚本的运行详见 PythonSimpleNode 。

打开(O) python_test.py /work/rvs_core/projects/runtime 保存(S)

```
1 # Note: this file is a template python file and it was generated from
  config_file :python_test.xml.
2 ##########################################
3 import RVS_LOG
4 # write log to RVS_log_file, for example:
5 # RVS_LOG.write("your own log string")
6 import numpy
7 # ToDo: import the moudle you need
8 import cv2
9
```

```
36   def process(self):
37     ##########################################
38     # ToDo: remove 'pass' and add your code
39     RVS_LOG.write("start resize ...")
40     self.im1 = cv2.resize(self.im0,(400,200),interpolation = cv2.INTER_AREA)
41     RVS_LOG.write("finished!")
42
43   ##########################################
44   # ToDo: add your own function
45
```

AIDetectGPU AI推理GPU版

AIDetectGPU 算子用于在安装 NVIDIA 独显情况下使用独显对几类 CNN 神经网络模型进行图像分割任务。与 AIDetectCPU 算子相比，GPU具有更强的计算能力和并行处理能力，因此使用GPU进行神经网络推理可以大幅提高计算速度和效率。

type	功能
MaskRCNN	用于 MaskRCNN 网络的推理，可以获得图像中目标的位置掩码(像素区域)
MaskRCNNClass	用于 MaskRCNNClass 网络的推理，可以获得图像中所有目标的位置掩码(像素区域) 同一个类别的所有目标的掩码共用一张输出图像
KeyPoint	用于关键点网络的推理，在 MaskRCNN 网络基础上可以额外获得关键点的像素值

说明： 还有一种RotatedYOLO网络的推理模式，同 MaskRCNN 网络差异较大，需要使用名为RotatedYOLONode的算子进行推理。

MaskRCNN

- **类名文件路径/class_name_file_path**：类名文件地址是一个指定了训练时所用的类别名称的文本文件。该文件的文件名应以 .txt 结尾。为了兼容旧版本的训练结果，该文件也可以使用 .names 结尾，或者使用训练时所生成的 json 文件。
- **权重文件路径/weight_file_path**：训练完成后的权重文件地址。
- **配置文件路径/config_file_path**：训练完成后的配置文件地址。**对于MaskRCNN网络在 RVS 1.3 及之前旧版本训练的结果，该配置文件参数设为空即可。**
- **物体得分阈值/obj_score_threshold**：目标得分阈值。得分低于阈值的目标会被淘汰。范围取值：[0,1]。默认值：0.75。
- **重置/reset**：参数重置。在执行了一次推理之后，如果对其中任何一个参数进行了修改，需要勾选 reset 参数，否则修改不会生效。勾选 reset 参数后，重新触发执行一次算子，该参数会自动由勾选状态重新变为非勾选状态，以确保下一次推理时使用新的参数。
- **识别结果图像/show_result**：设置推理结果示意图在 2D 视图中的可视化属性。
  -  打开推理结果示意图可视化。**打开能更直观显示推理结果，利于调试。**
  -  关闭推理结果示意图可视化。**关闭可视化可以缩短算子对结果图像进行着色处理的时间。**
- **Mask列表/mask_list**：设置 mask_list 输出端口的图像内容在 2D 视图中的可视化属性。其中每一张图像都是一张灰度图，背景像素为 0，目标区域为 255 。
- **Mask名称列表/mask_names**：设置各个目标对应的类名曝光属性。打开后则可以将 mask_names 输出端口的内容绑定到交互面板上的表格并输出显示。
  -  打开曝光。
  -  关闭曝光。

数据信号输入输出

输入：

- `image`：
  - 数据类型：Image
  - 输入内容：需要推理的图像数据

输出：

- `show_result`：
  - 数据类型：Image
  - 输出内容：图像推理结果的效果示意图
- `mask_list`：
  - 数据类型：ImageList
  - 输出内容：图像推理后获得的所有目标的掩码图，每个目标在 List 中的位置同下述 mask_names 保持一致。各个目标按照对应的得分从高到低排列。
- `mask_names`：
  - 数据类型：StringList
  - 输出内容：以 String 形式存放各个目标对应的类名

功能演示

使用 AIDetectGPU 算子的MaskRCNN神经网络模型对加载的图像进行推理。获得图像中的目标位置掩码(像素区域)。

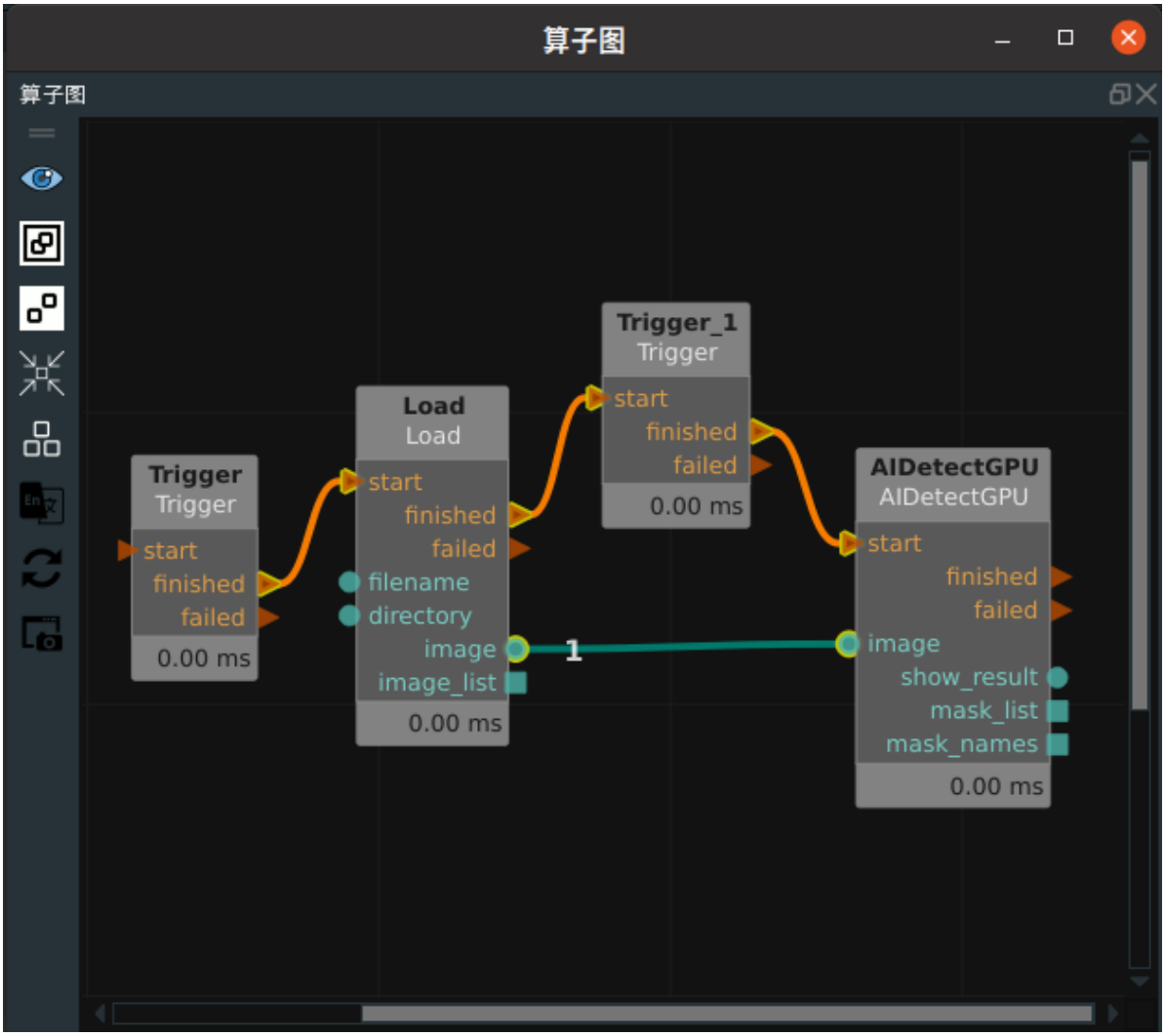
步骤1：算子准备

添加 Trigger （2 个）、Load、AIDetectGPU算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
   - 类型 → Image
   - 文件 → ••• → 选择图像文件名 （ *example_data/mask_data_train/20221010140850486/rgb.png* ）
   - 图像 → 👁 可视
2. 设置 Trigger_1 算子参数：类型 → InitTrigger
3. 设置 AIDetectGPU 算子参数：
   - 类型 → MaskRCNN
   - 类名文件路径 → ••• → 选择相应文件名（ *example_data/mask_data_train/fruits.txt* ）
   - 权重文件路径 → ••• → 选择相应权重文件名 （ *example_data/mask_data_train/train_output/model_final.pth* ）
   - 配置文件路径 → ••• → 选择相应配置文件名（ *example_data/mask_data_train/train_output/config.yaml* ）
   - 物体得分阈值 → ••• → 0.75
   - 识别结果图像 → 👁 可视

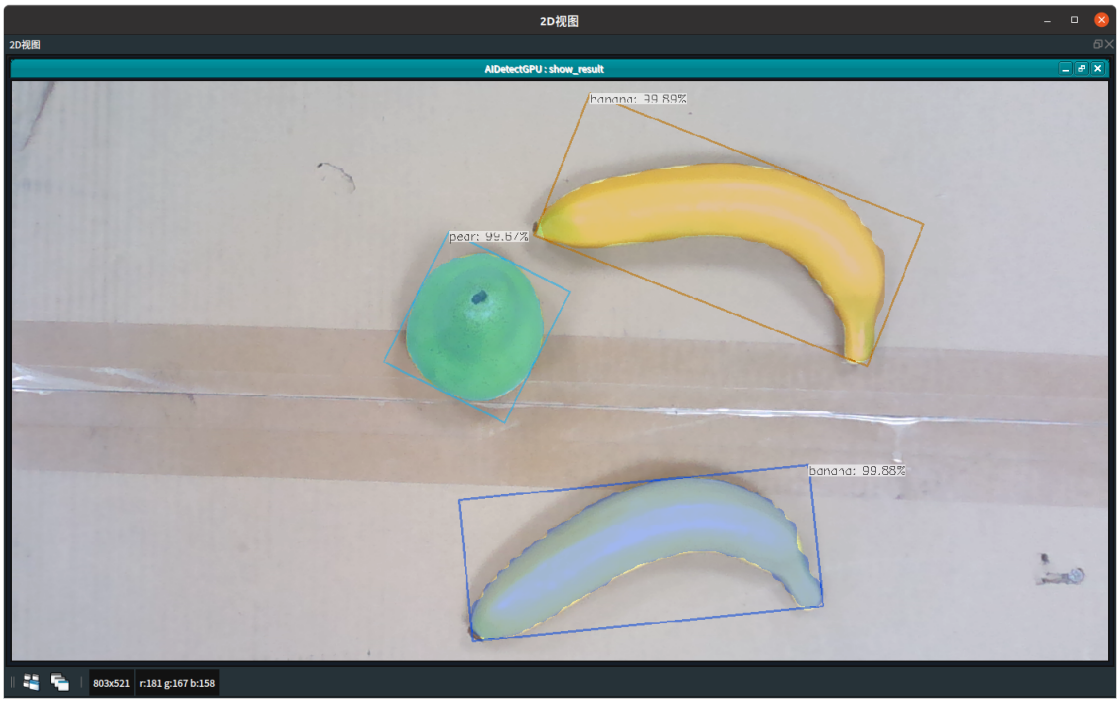
步骤3：连接算子



步骤4：运行与结果

1. 打开 RVS 的运行按钮，XML 会自动运行Trigger_1（type为InitTrigger）。此时会触发AIDetectGPU算子完成第一次的初始化运行（首次运行不会对输入的图像进行推理，仅仅是运行环境检测以及将模型文件从本地载入到内存），运行成功后界面显示如下图，在日志栏依次打印了"AIDetectis loading module"..."GetClassNames done"等四条语句。
2. 在推理过程中，会在日志栏高亮打印“AIDetect is processing”的输出信息，推理完成后，会在日志栏打印该图像推理得到的目标个数，如下图所示。

3. 推理完成后，在 RVS 的 2D 图区域显示推理结果示意图，如下所示。



MaskRCNNClass

算子参数

- **Mask名称列表/mask_class_list**：设置 mask_class_list 输出端口的图像内容在 2D 视图中的可视化属性。其中每一张图像都是一张灰度图，代表某一个类别的所有目标的掩码图。mask_class_list 中的类别，按照类名文件中类别的先后顺序进行排序。每张mask图像的背景像素为 0，每一个目标按照得分从大到小的顺序依次使用1、2、3... 等像素。比如一张 mask 图中像素为 1 的一块区域，代表该类别中得分最高的目标的像素位置。
- **其余参数**：与 MaskRCNN 一致。

数据信号输入输出

输入：

- **image**：
  - 数据类型：Image
  - 输入内容：需要推理的图像数据

输出：

- **show_result**：
  - 数据类型：Image
  - 输出内容：图像推理结果的效果示意图
- **mask_class_list**：
  - 数据类型：ImageList
  - 输出内容：图像推理后获得的所有类别目标的掩码图，详细说明见上述“算子参数” -- mask_class_list 的介绍。
- **mask_names**：
  - 数据类型：StringList
  - 输出内容：以 String 形式存放所有的类名，顺序同类名文件保持一致，并且同上述 mask_class_list 保持一致。

功能演示

与上述 MaskRCNN 模块类似，请参照该章节功能演示模块。

KeyPoint

算子参数

- **类名文件路径/class_name_file_path**：训练时生成的 json 文件，一般是 annotations.json 。
- **关键点得分阈值/keypt_score_threshold**：关键点的得分阈值，得分低于阈值的关键点会被淘汰。范围取值：[0,10]。默认值：0.8。
- **Mask旋转矩阵列表/mask_boxes**：设置曝光 mask_boxes 输出端口。打开后则可以将该输出端口的内容绑定到交互面板上的表格并输出显示。
  -  打开曝光。
  -  关闭曝光。
- **其余参数**：同 MaskRCNN 保持一致。

数据信号输入输出

输入：

- **image**：
  - 数据类型：Image
  - 输入内容：需要推理的图像数据

输出：

- **show_result**：
  - 数据类型：Image
  - 输出内容：图像推理结果的效果示意图
- **mask_list**：
  - 数据类型：ImageList
  - 输出内容：图像推理后获得的所有目标的掩码图，每个目标在 List 中的位置同下述 3 个 List 一致。各个目标按照对应的得分从高到低排列。
- **mask_names**：
  - 数据类型：StringList
  - 输出内容：以 String 形式存放各个目标对应的类名
- **mask_boxes**：
  - 数据类型：RotatedRectList
  - 输出内容：以旋转矩形框的形式输出各个目标的外边框
- **keypt_list**：
  - 数据类型：ImagePointsList
  - 输出内容：图像推理后获得的所有目标的关键点信息，List 中的每一个 ImagePoints 。代表一个目标上的所有关键点。每一个 ImagePoints 的内容，均按照关键点顺序以 "x1 y1 x2 y2 ..." 的形式依次排列。

功能演示

使用 AIDetectGPU 算子中 KeyPoint 神经网络模型对加载的图像进行推理。获得图像中目标的位置掩码(像素区域)和关键点的像素值。

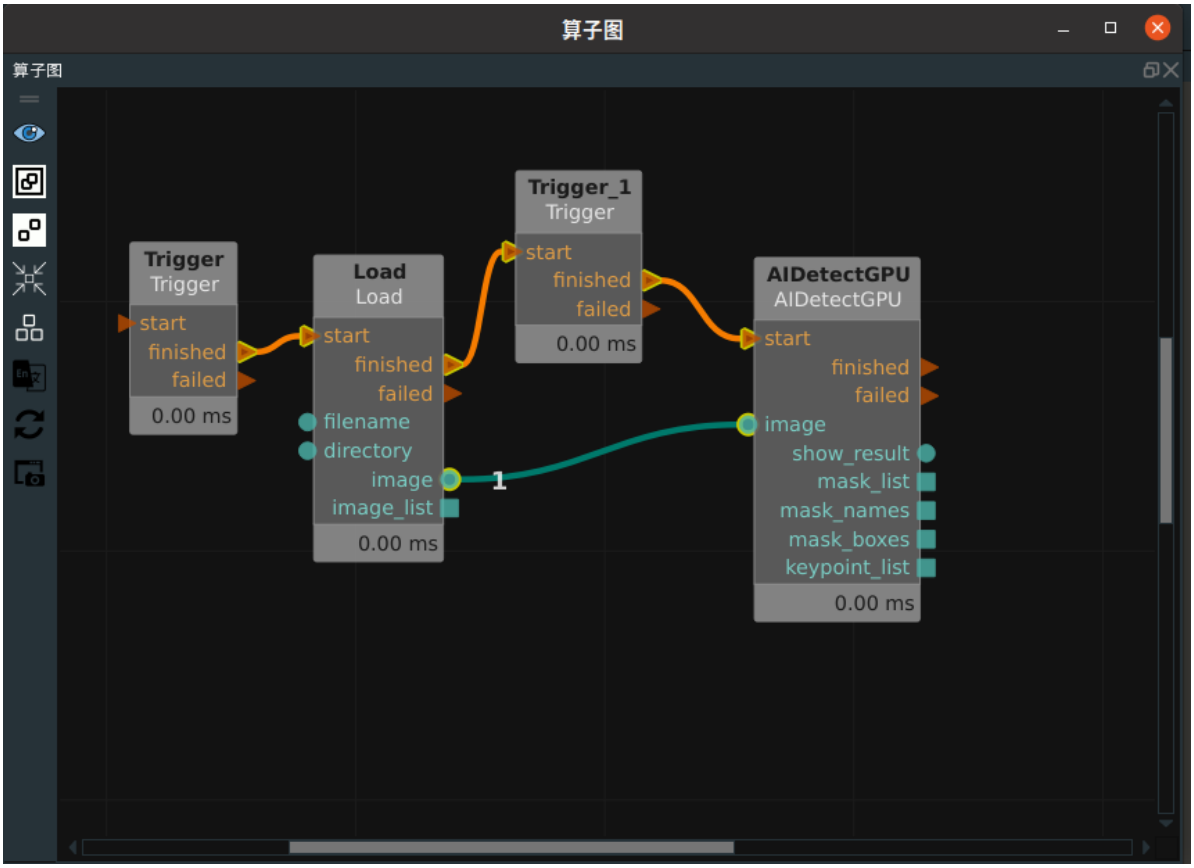
步骤1：算子准备

添加 Trigger (2 个）、Load、AIDetectGPU算子至算子图。

步骤2：设置算子参数

1. 设置 Load 算子参数：
    - 类型 → Image
    - 文件 → ••• → 选择图像文件名 （ *example_data/keypoint_data_train/20221010113840917/rgb.png*)
    - 图像 →  可视
2. 设置 Trigger_1 算子参数：
    - 类型 → InitTrigger
3. 设置 AIDetectGPU 算子参数：
    - 类型 → KeyPoint
    - 类名文件路径 → ••• → 选择相应文件名（ *example_data/keypoint_data_train/annotations.json*)
    - 权重文件路径 → ••• → 选择相应权重文件名 （ *example_data/keypoint_data_train/train_output/model_final.pth*)
    - 配置文件路径 → ••• → 选择相应配置文件名 （ example_data/keypoint_data_train/train_output/config.yaml)
    - 物体得分阈值 → 0.75
    - 关键点得分阈值 → 0.8
    - 识别结果图像 →  可视

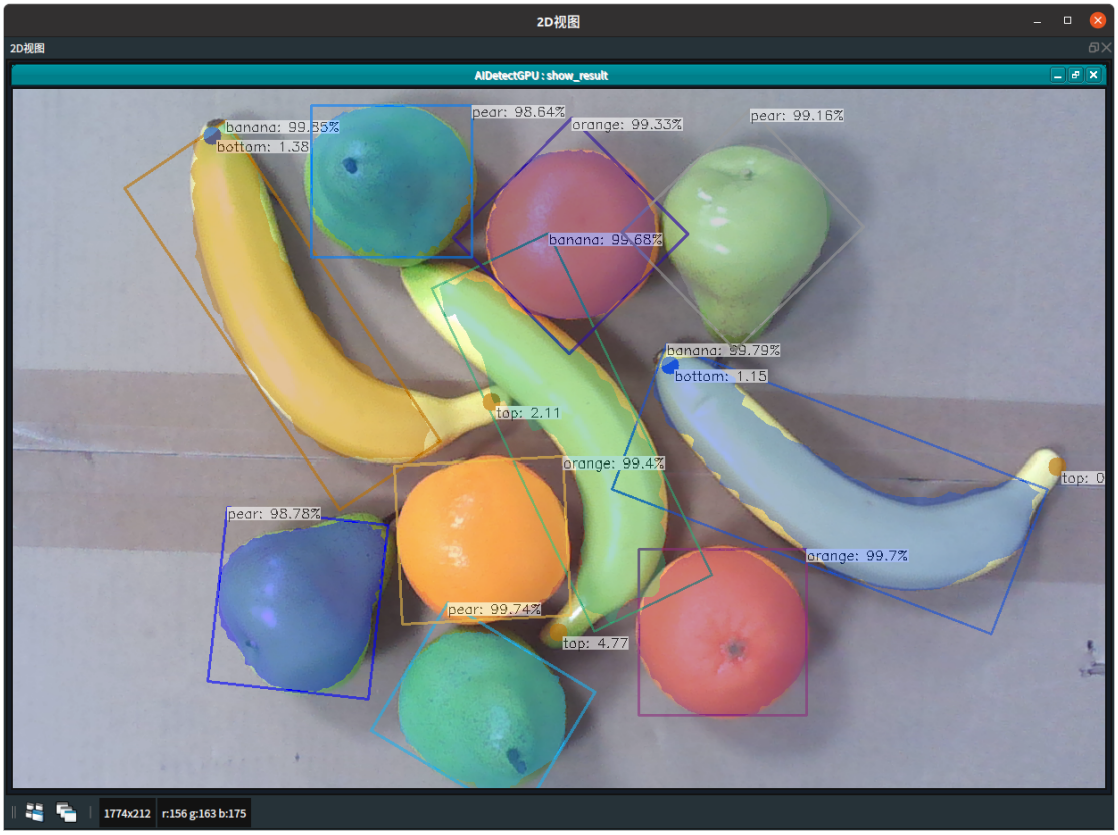
步骤3：连接算子



步骤4：运行与结果

1. 打开 RVS 的运行按钮，XML 会自动运行Trigger_1（type为InitTriggerr）。此时会触发AIDetectGPU算子完成第一次的初始化运行（首次运行不会对输入的图像进行推理，仅仅是运行环境检测以及将模型文件从本地载入到内存），运行成功后界面显示如下图，在日志栏依次打印了"AIDetect is loading module"..."GetClassNames done"等四条语句。
2. 在推理过程中，会在日志栏高亮打印“AIDetect is processing”的输出信息，推理完成后，会在日志栏打印该图像推理得到的目标个数，如下图所示。

3. 推理完成后，在 RVS 的 2D 图区域显示推理结果示意图，如下所示。


2D视图
2D视图
AIDetectGPU : show_result
banana: 99.35%
bottom: 1.38
pear: 98.64%
orange: 99.33%
pear: 99.16%
banana: 99.68%
banana: 99.79%
bottom: 1.15
top: 2.11
orange: 99.4%
top: 0
pear: 98.78%
orange: 99.7%
pear: 99.74%
top: 4.77
1774x212
r:156 g:163 b:175

AITrain AI训练

AITrain 算子用于对几类 CNN 神经网络模型的训练。算子执行过程中包含数据格式转换以及训练两个环节。

type	功能
MaskRCNNTrain	用于 MaskRCNN 网络的训练，可以获得图像中目标的位置掩码(像素区域)
RotatedYOLOTrain	用于旋转 YOLO 网络的训练，可以获得图像中目标位置的旋转矩形框
KeyPointTrain	用于关键点网络的训练，在 MaskRCNN 网络的基础上可以额外获得关键点的像素值

说明：MaskRCNN 网络训练耗时短，所需训练数据量少，推理准确度高，但是推理速度慢。KeyPoint 网络训练耗时较短，所需训练数据量较少，推理准确度高，推理速度慢。RotatedYOLO 网络训练时间长，所需训练数据量较大，推理准确度明显低于前两个网络，但是推理速度很快，可以轻松适用于 CPU 环境，并且可以获取旋转矩形框，对于大多数长方体目标的检测是适用的。

MaskRCNNTrain

算子参数

- **数据目录/data_directory**：训练数据文件夹路径。

  注意：该路径内包含多个子文件夹，每个子文件夹代表一组图像数据，其中必须包含有一张命名为 rgb.png 的图像以及一个命名为 rgb.json 的标注文件。

- **类名文件路/classnames_filepath**：训练数据的类名文件地址。

  注意：训练所用的类名文件必须以 txt 作为文件后缀，文件中每一个类别名各自占一行。类名中不允许包含空格，且必须同标注文件中的类名保持一致。

- **长边长度/long side size**：训练图像的长边像素长度。

  注意：在执行训练时，以及训练完成后进行推理时，算子都会首先将原图按照这里设定的尺寸进行缩放。所以这里的尺寸设置直接影响了此时训练以及后期推理的速度！

- **短边长度/short side size**：训练图像的短边像素长度。注意事项同上。

- **每批图像张数/batch_size**：训练时每次对权重文件进行一次迭代更新优化时所用的图像张数。该数据必须是 2 的整幂次倍，1、2、4、8...。适当的增加该数值能提升训练效果，但如果设置较大，往往可能超出独显显存导致训练时出现“ CUDA out of memory "的报错。以 8G 显存为例，在 img_size 参数设置为 640/360 或 1920/1080 时，可以将 batch_size 设为 4 或 2 。

- **数据扩容倍数/data_cycle_times**：对原始数据进行复制的倍数。在训练数据较少比如低于 100 张图像时，可以将该数值设为 2 或者 4 来达到增加图像数量的效果。

- **训练循环次数/epoch_times**：在上述复制的基础上，对所有原始数据训练一遍即为一次 epoch，该参数表示对所有训练数据重复使用的次数。每一次 epoch 结束都会将当前的权重文件进行临时保存。一般可以设置为 3、4、5 等。

- **图像翻转方式/img_flip**：数据增强——随机翻转原始图像。
  - horizontal：水平翻转。
  - vertical：上下翻转。
  - none：禁用翻转。

- **是否随机裁剪/img_crop**：数据增强——随机裁剪原始图像

注意：裁剪后的保留比例已经固定为 0.9～1 。

- **基础模型配置文件/model_cfg_file**：常规情况下置为空。当需要对某次的训练结果进行加训时，在这里输入上次训练结果的配置文件地址。
- **基础模型权重文件/model_weights_file**：常规情况下置为空。当需要加训时，在这里输入上次训练结果的权重文件地址。

功能演示

使用 AITrain 算子中 MaskRCNNTrain 对标注好的数据进行训练。

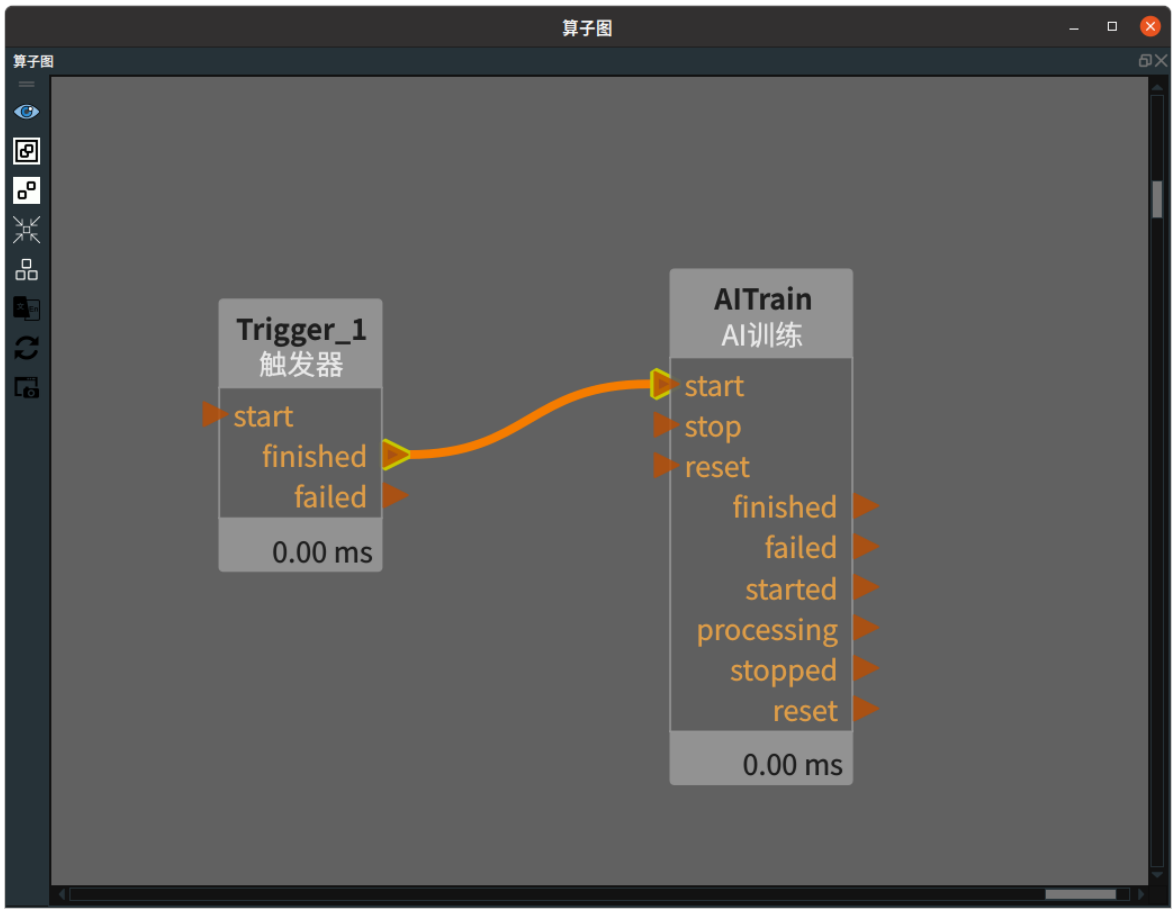
步骤1：算子准备

添加 Trigger、AITrain 算子至算子图。

步骤2：设置算子参数

1. 设置 AITrain 算子参数：
    - 类型 → MaskRCNNTrain
    - 数据目录 → ••• → 选择训练数据文件夹路径。（ *example_data/mask_data_train* ）
    - 类名文件路径 → ••• → 选择训练数据的类名文件地址。（ *example_data/mask_data_train/fruits.txt* ）
    - 长边长度 → 1920
    - 短边长度 → 1080
    - 每批图像张数 → 2
    - 数据扩容倍数 → 4
    - 训练循环次数 → 5
    - 是否随机裁剪 → ••• → True
    - 其余参数保持默认

步骤3：连接算子

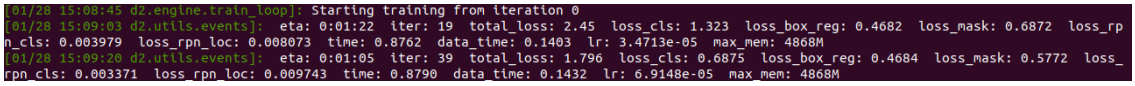


步骤4：运行

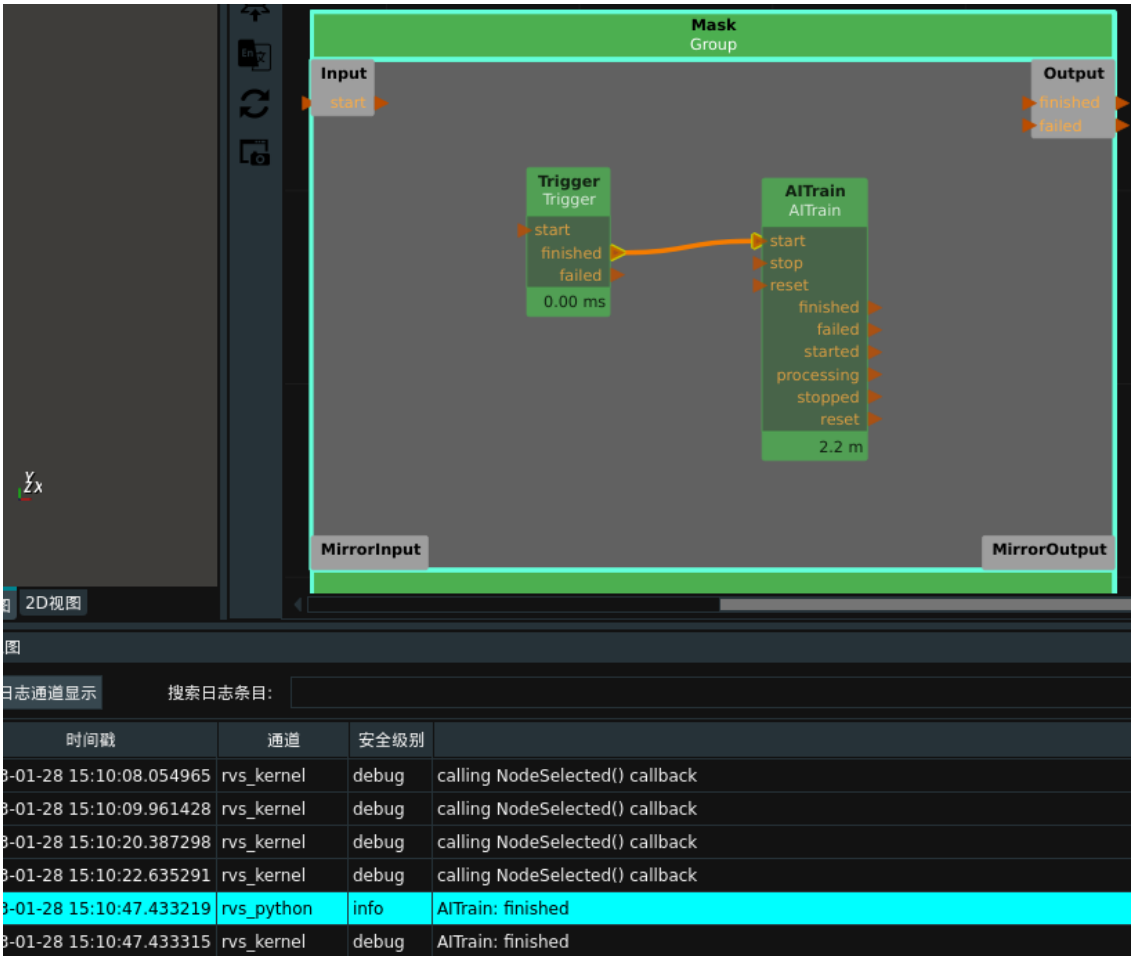
打开 RVS 的运行按钮，触发Trigger 算子。

运行结果

1. 在训练过程中，算子呈现蓝色，同时在日志栏高亮打印输出信息。如果使用了 Windows / Linux 版本 启动RVS，则训练时还会在终端实时打印当前的训练进度，如下图所示。



2. 训练完成后，算子呈现绿色，同时在日志栏高亮打印完成信息，RVS界面显示如下所示。



3. 终端显示如下所示，在最后打印出 finished 。

```
[01/28 15:10:46 d2.engine.defaults]: Evaluation results for coco_my_val in csv format:
[01/28 15:10:46 d2.evaluation.testing]: copypaste: Task: bbox
[01/28 15:10:46 d2.evaluation.testing]: copypaste: AP,AP50,AP75,APs,APm,APl
[01/28 15:10:46 d2.evaluation.testing]: copypaste: 54.4066,92.8536,43.7262,64.1914,nan,nan
[01/28 15:10:46 d2.evaluation.testing]: copypaste: Task: segm
[01/28 15:10:46 d2.evaluation.testing]: copypaste: AP,AP50,AP75,APs,APm,APl
[01/28 15:10:46 d2.evaluation.testing]: copypaste: 72.8336,92.8536,87.6231,87.9323,nan,nan
finished
```

说明

训练失败主要有以下三种情况。

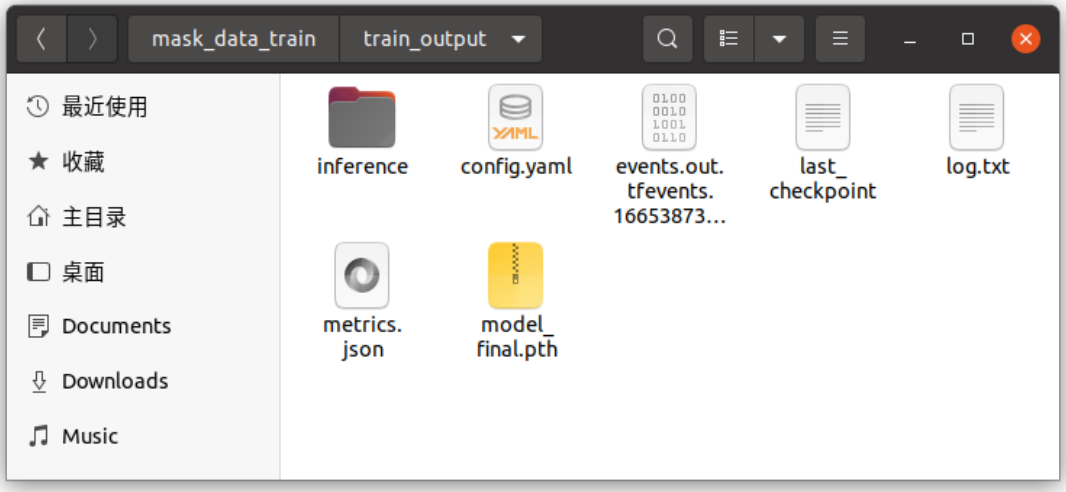
1.请检查MIVDIA独立显卡是否正常运行。

2.出现“ CUDA out of memory "的报错。可以调整算子参数，可降低batch_size或者降低img_size.

3.初次训练需要联网下载预训练的权重文件，所以可能是网络异常导致下载失败。可进行如下检查：Linux 版本在根目录下搜索“model_final_f10217.pkl.lock”文件（Windows 版本在 C 盘目录下搜索该文件），检查该文件所在目录是否有“model_final_f10217.pkl”文件。如果没有“model_final_f10217.pkl”文件，可从 RVS 安装目录下的 rvs_sdk 文件夹内找到

“model_final_f10217.pkl”文件，将该文件复制到“model_final_f10217.pkl.lock”文件所在目录下。再次进行训练即可。

4. 训练完成后，可在参数 data_directory 所在的文件夹内看到一个 train_output 的文件夹，其内容如下所示。其中，model_final.pth 是最终训练出来的网络权重文件，config.yaml 是对应的网络配置文件。



RotatedYOLOTrain

算子参数

- **数据目录/data_directory**：同 MaskRCNNTrain 描述一致 。
- **类名文件路径/classnames_filepath**：同 MaskRCNNTrain 描述一致。
- **使用设备/device**：训练网络时所用的设备。可选参数：“gpu" 与 ”cpu“ 。建议默认选择 “gpu” ，可以提高 AITrain 的速度。
- **图像尺寸/image_size**：同 MaskRCNNTrain 的 max_img_size/min_img_size 。区别：该网络在实际训练与推理的时候，默认将图像尺寸设为长宽一致。
- **每批图像张数/batch_size**：同 MaskRCNNTrain 的 batch_size 描述一致。但是由于该网络模型较小，所以该参数可以适当放大。
    - 以 8G 显存为例：在 img_size 参数设置为 640 或1280 时，可以将 batch_size 设为 16 或 4 。

注意：该网络由于模型较小，需要更大的训练数据量以及更多的训练批次，所以训练时所用的数据量不应低于 100 。

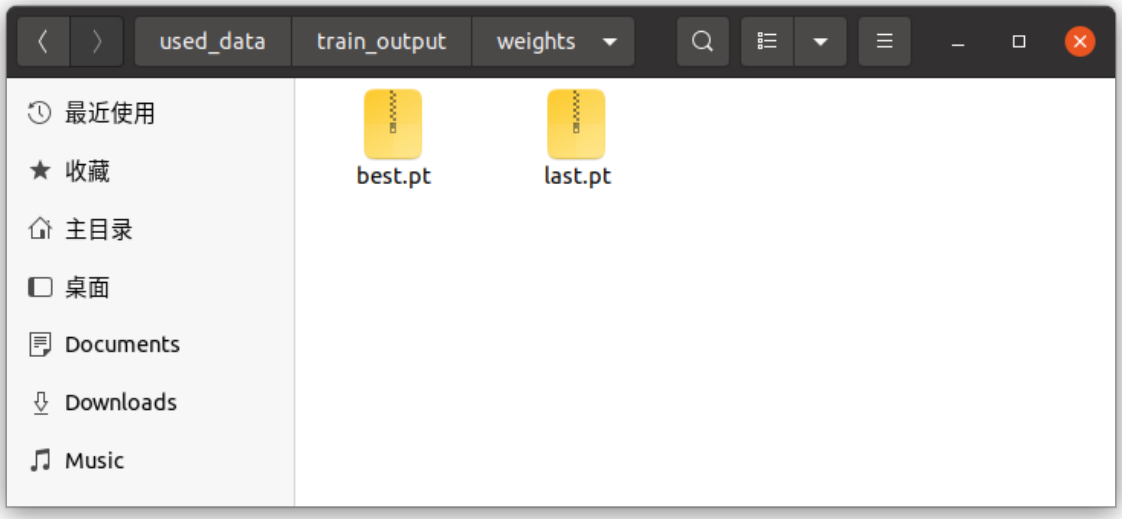
- **max_epochs**：训练时所有数据使用一遍为一个epoch，该参数表示训练数据的最大循环次数。当训练数据中的所有目标个数低于200时，该参数一般设置为1000。其他情况一般设为 2000。

  说明：该网络设置了自动提前终止训练的功能。在某次训练结束后，后续的连续 200 次 epoch 没有获得训练提升则会自动提前终止训练。

功能演示

本节将使用 RotatedYOLOTrain 对标注好的数据进行训练。这与 MaskRCNNTrain 中展现的训练方法相同，请参照该章节的功能演示。

注意：该网络模型训练完成后，会在参数 data_directory 所在的文件夹内看到一个 used_data 的文件夹，used_data/train_output/weights/last.pt 即为最终的训练权重文件。如下图所示：



KeyPointTrain

算子参数

- **类名文件路径/classnames_filepath**：类名文件，训练所用的类名文件必须以 txt 作为文件后缀。每一个类名单独占一行。如果某个类别拥有关键点，则需要将关键点名依次用空格隔开填写到该类名所在行。例如：

```
pear

banana top bottom

orange
```

  则上述文件代表总共有三个类别分别为 pear、banana、orange，并且其中的 banana 类别的目标还具有 top 和 bottom 两个关键点。KeyPointTrain 算子仅允许给一个类别设置关键点。在特别的情况下，比如多个类别分别是 red_box、blue_box 、black_box，或者是man、woman这种，允许给这些多类别的对象同时设置关键点，但是要求他们的关键点保持一致，此时可以对这些不同颜色的箱子同时设置center、top的关键点，对人类同时设置头、左脚、右脚的关键点。

- **其余参数**：同 MaskRCNNTrain 算子一致 。

功能演示

本节将使用 KeyPointTrain 对标注好的数据进行训练。这与 MaskRCNNTrain 中展现的训练方法相同，请参照该章节的功能演示。

AIDetectCPU AI推理CPU版

AIDetectCPU 算子使用 cpu 对几类 CNN 神经网络模型进行推理。

说明：RVS 的 GPU/CPU 版本均可使用该算子。

type	功能
MaskRCNN	用于 MaskRCNN 网络的推理，可以获得图像中目标的位置掩码(像素区域)
MaskRCNNClass	用于 MaskRCNNClass 网络的推理，可以获得图像中所有目标的位置掩码(像素区域) 同一个类别的所有目标的掩码共用一张输出图像
KeyPoint	用于关键点网络的推理，在 MaskRCNN 网络基础上可以额外获得关键点的像素值

说明： 还有一种RotatedYOLO网络的推理模式，同 MaskRCNN 网络差异较大，需要使用名为 RotatedYOLONode 的算子进行推理。

MaskRCNN

算子参数

- **类名文件路径/class_name_file_path**：类名文件地址，即训练时所用的类名文件。要求文件名以 .txt 结尾。为了兼容旧版本的训练结果，该文件也可以用 .names 结尾，也可以使用训练时所生成的 json文件。
- **权重文件路径/weight_file_path**：训练完成后的权重文件地址。
- **配置文件路径/config_file_path**：训练完成后的配置文件地址。对于MaskRCNN 网络在 RVS1.3 及之前旧版本训练的结果，该配置文件参数设为空即可。
- **物体得分阈值/obj_score_threshold**：目标得分阈值，得分低于阈值的目标会被淘汰，取值范围：[0,1] 。默认值：0.75。
- **重置/reset**：参数重置。在执行了一次推理之后，如果对上述参数中的任何一个进行了修改，则还需要勾选该 reset 参数，否则上述参数的更改不会生效。重新触发执行一次算子，该参数会自动由勾选状态重新变为非勾选状态。
- **停止/stop**：停止推理，该参数仅适用于 Ubuntu 版本的 RVS 。在执行推理时，程序底层的神经网络是独立运行的，勾选该参数并触发算子执行，算子底层会杀死对应的神经网络运行程序。
- **识别结果图像/show_result**：设置推理结果示意图在 2D 视图中的可视化属性。
  -  打开推理结果示意图可视化。**打开能更直观显示推理结果，利于调试。**
  -  关闭推理结果示意图可视化。**关闭可视化可以缩短算子对结果图像进行着色处理的时间。**
- **Mask列表/mask_list**：设置 mask_list 输出端口的图像内容在 2D 视图中的可视化属性。其中每一张图像都是一张灰度图，背景像素为 0，目标区域为 255 。
- **Mask名称列表/mask_names**：设置目标对应的类名曝光属性。打开后则可以将 mask_names 输出端口的内容绑定到交互面板上的输出工具表格并输出显示。
  -  打开曝光。

-  关闭曝光。

数据信号输入输出

输入：

- `image`：
  - 数据类型：Image
  - 输入内容：需要推理的图像数据

输出：

- `show_result`：
  - 数据类型：Image
  - 输出内容：图像推理结果的效果示意图
- `mask_list`：
  - 数据类型：ImageList
  - 输出内容：图像推理后获得的所有目标的掩码图，每个目标在List中的位置同下述mask_names保持一致。各个目标按照对应的得分从高到低排列。
- `mask_names`：
  - 数据类型：StringList
  - 输出内容：以String形式存放各个目标对应的类名

功能演示

本节将使用 AIDetectCPU 算子的 MaskRCNN 神经网络模型对加载的图像进行推理。获得图像中的目标位置掩码(像素区域)。与 AIDetectGPU 算子的MaskRCNN模块类似，请参照该章节功能演示模块。

MaskRCNNClass

算子参数

- `Mask类别列表/mask_class_list`：设置 mask_class_list 输出端口的图像内容在 2D 视图中的可视化属性。其中每一张图像都是一张灰度图，代表某一个类别的所有目标的掩码图。mask_class_list中的类别，按照类名文件中类别的先后顺序进行排序。每张mask图像的背景像素为 0，每一个目标按照得分从大到小的顺序依次使用1、2、3...等像素。比如一张mask图中像素为1的一块区域，代表该类别中得分最高的目标的像素位置。
- `其余参数`：与 MaskRCNN 描述一致。

数据信号输入输出

输入：

- `image`：
  - 数据类型：Image
  - 输入内容：需要推理的图像数据

输出：

- `show_result`：
  - 数据类型：Image
  - 输出内容：图像推理结果的效果示意图

- **mask_class_list**：
  - 数据类型：ImageList
  - 输出内容：图像推理后获得的所有类别目标的掩码图，详细说明见上述“算子参数”——mask_class_list的介绍。
- **mask_names**：
  - 数据类型：StringList
  - 输出内容：以String形式存放所有的类名，顺序同类名文件保持一致，并且同上述mask_class_list保持一致。

功能演示

本节将使用 AIDetectCPU 算子的 MaskRCNNClass 神经网络模型对加载的图像进行推理。获得图像中的目标位置掩码(像素区域)。与 AIDetectGPU 算子的MaskRCNN模块类似，请参照该章节功能演示模块。

KeyPoint

算子参数

- **类名文件路径/class_name_file_path**：训练时生成的 json 文件，一般是 annotations.json 。
- **关键点得分阈值/keypt_score_threshold**：关键点的得分阈值，得分低于阈值的关键点会被淘汰。
- **Mask旋转矩阵列表/mask_boxes**：设置曝光 mask_boxes 输出端口。打开后则可以将该输出端口的内容绑定到交互面板上的表格并输出显示。
  -  打开曝光。
  -  关闭曝光。
- **其他参数**：与 MaskRCNN 描述一致。

数据信号输入输出

输入：

- **image**：
  - 数据类型：Image
  - 输入内容：需要推理的图像数据

输出：

- **show_result**：
  - 数据类型：Image
  - 输出内容：图像推理结果的效果示意图
- **mask_list**：
  - 数据类型：ImageList
  - 输出内容：图像推理后获得的所有目标的掩码图，每个目标在List中的位置同下述3个List一致。各个目标按照对应的得分从高到低排列。
- **mask_names**：
  - 数据类型：StringList
  - 输出内容：以String形式存放各个目标对应的类名
- **mask_boxes**：
  - 数据类型：RotatedRectList
  - 输出内容：以旋转矩形框的形式输出各个目标的外边框
- **keypoint_list**：

- 数据类型：ImagePointsList
- 输出内容：图像推理后获得的所有目标的关键点信息，List中的每一个ImagePoints代表一个目标上的所有关键点。每一个ImagePoints的内容，均按照关键点顺序以"x1 y1 x2 y2 ..."的形式依次排列。

功能演示

本节将使用 AIDetectCPU 算子的 KeyPoint 神经网络模型对加载的图像进行推理。获得图像中目标的位置掩码(像素区域)和关键点的像素值。与 AIDetectGPU 算子的KeyPoint模块类似，请参照该章节功能演示模块。

resource

TCPServerResource TCP服务器资源

TCPServerResource 算子属于资源类线程算子，用于在 RVS 的分线程中启动一个 TCP server 服务器。

客户端可以通过该算子对 RVS 中的数据进行读写。通信数据是以字符串形式表达的 json 格式。

1. 客户端发送更改算子参数的消息。发送字符串格式为:

```
{"SetPara":[

{"node_name":"算子名称","para_name":"要更改的参数名称","para_value":"要更改的值"},

...

]}+delimiter(消息结束符)
```

如果发送消息的格式不合法，则本算子会回复：

```
ParseJson;0
```

如果参数设置失败，则本算子会回复 ：

```
SetPara;0
```

如果参数设置成功，则本算子会回复：

```
SetPara;1
```

2. 客户端想读取算子的输出端口的值。发送字符串格式为：

```
{"GetOutput":[

{"node_name":"算子名称","index":int_value,"type":"算子类型"},

...

]}+delimiter(消息结束符)
```

注意

1. 上文中的 index 表示某个算子的所有数据输出端口从上往下(从0开始)排列的端口序号。排序时不包含 finished 和 failed 等控制端口。

2. 当前的 GetOutput 仅支持 String 和 Pose 两种算子类型，也可以支持 StringList 以及 PoseList，但是这里的 type 统一填写成 String 以及 Pose 。

3. 同时获取多个输出端口的回复值时，回复内容通过分号隔开。

如果发送消息的格式不合法，则本算子会回复：

```
ParseJson;0
```

如果数据读取失败(找不到对应算子，或找不到算子对应端口，或算子对应端口属性不匹配)，则本算子会回复：

```
GetOutput;0
```

如果参数设置成功，则本算子会回复对应的读取内容。

算子参数

- **自动启动/auto_start**：自动运行。如果勾选该选项，则 RVS 第一次运行后，会自动触发该算子执行。算子运行后，开始创建 TCP 服务端，创建成功后监听服务端口并等待客户端访问。
- **启动/start**：勾选后，开始触发算子执行。算子运行后，开始创建 TCP 服务端，创建成功后监听服务端口并等待客户端访问。
- **停止/stop**：勾选后，开始停止 TCP 服务端的监听。
- **重置reset**：在该资源算子已经运行后，如果重新更改了 **端口** 、**连接数量** 、**服务器模式** 、**分隔符** 4个参数中的任意一个之后，若需要生效，都必须勾选**重置** 选项进行资源重置，然后重新勾选 **启动** 运行。
- **端口/port**：创建 TCP server 服务器所使用的本机服务端口。

  说明：实际运行时，如果提示 TCP 服务端创建失败，往往是所选取的端口已经被其他服务所占用，更换端口即可。如果在同一个 RVS 软件中，创建多个TCP 服务端算子，彼此的**端口**也要互斥。
- **连接数量/connections**：可同时支持的最大客户端连接数量。
- **服务器模式/server_mode**：运行模式。
  - Once：表示完成一次 TCP 对话以后自动断开同客户端的链接(比如客户端首先同该算子建立了 TCP 连接，然后第一次发送了消息请求之后，继续发送就会报错)。
  - Continous：表示客户端建立连接后可以无限次数的对话。
- **分隔符/delimiter**：消息结束符，包含 RT 换行符、“；”、“#”、“$”。当选择了其中某一种时，另外三种就会被视作普通字符随意使用 。服务端在接收客户端发送的消息时，会捕获第一个消息结束符之前的所有消息。

  说明：由于不同的通信软件、电脑系统软件、机器人操作系统软件对回车换行的定义不一致，容易导致信息发送或者接收失败，所以一般不建议选择 **RT** 作为消息结束符。

控制信号输入输出

说明：Resource 资源类算子的控制信号端口无法触发使用。

功能演示

下面分别针对读写 2 类通讯信号，进行通讯演示。

在选用演示所用的 TCP 客户端时，由于 RVS 自带的 CommonTCPClient 算子在使用时要求服务端返回的信息必须以四类固定字符结尾，而 TCPServerResource 算子在作为 TCP 的服务端时，返回给 TCP Client 的信息是固定的并且没有额外添加上述四类固定结尾字符，所以这里不再选用 CommonTCPClient 算子作为客户端。

说明：这里的演示案例是基于 ubuntu 系统给出的，使用了 "echo + nc " 的指令方式实现 TCP 客户端的功能；如果是在 Windows 版本，建议使用通讯助手等工具。

步骤1：算子准备

添加 Trigger 、Emit （2个）、Concatenate 、TCPServerResource 算子至算子图。

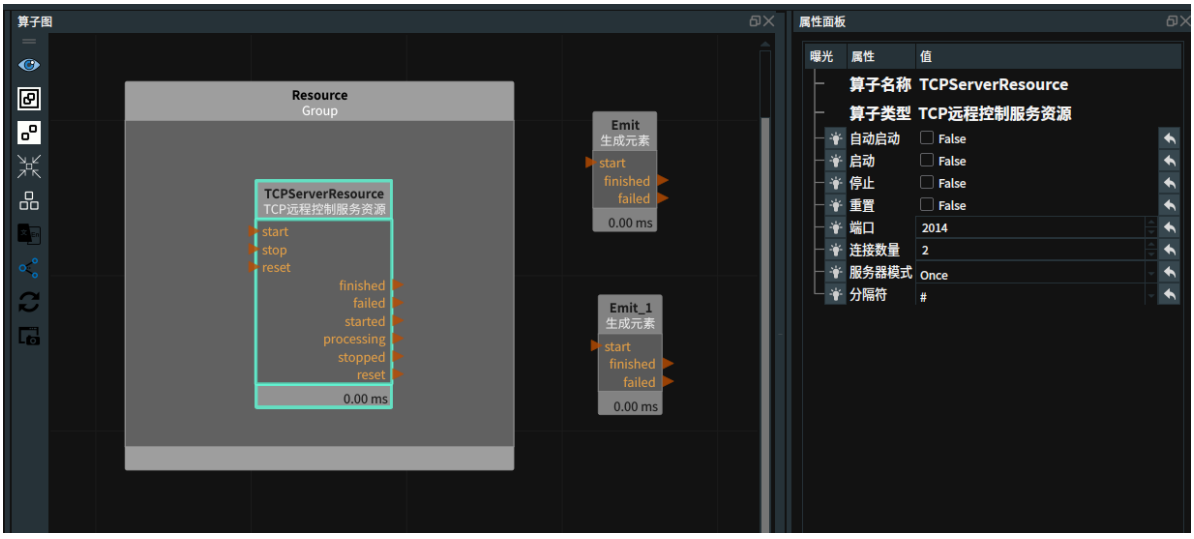
步骤2：设置算子参数

设置 TCPServerResource 算子参数：

- 自动启动 → True
- 分隔符 → #
- 服务器模式 → Once

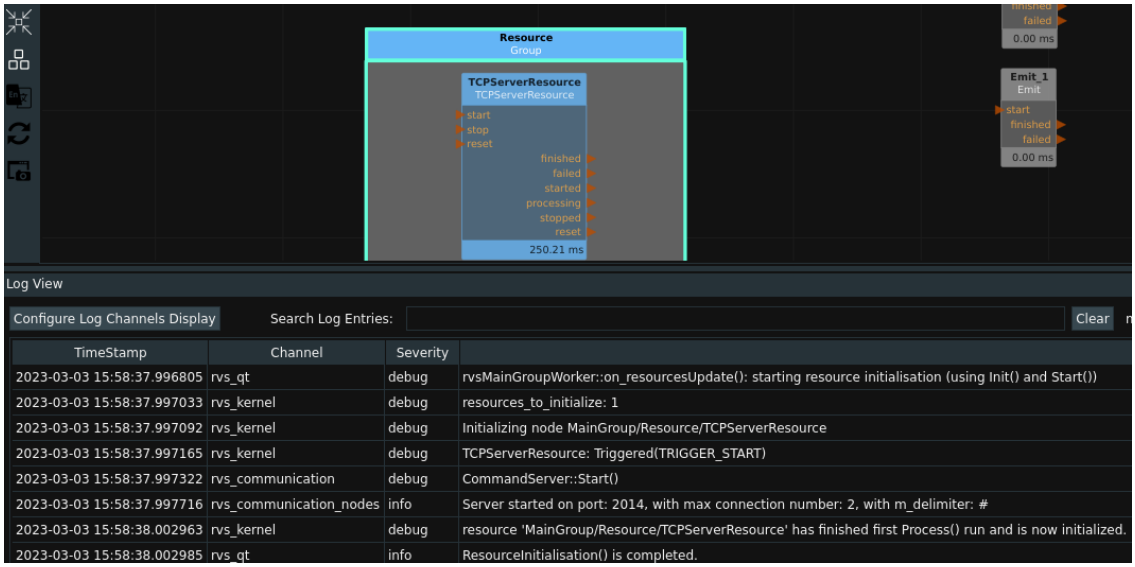
步骤3：连接算子

打开 RVS 软件界面，在算子图拖入 TCPServerResource 算子以及拖入两个 Emit 算子如下图所示。TCPServerResource 选用了 “echo + nc” 的指令方式，所以这里每发送-接收一次消息后都要断开连接，所以选择了 Once 模式。



步骤4：运行及运行结果

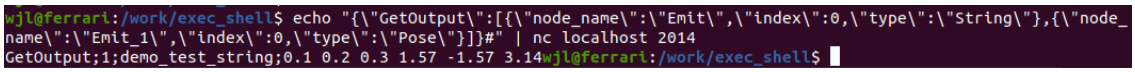
1. 打开 RVS 的运行按钮，TCPServerResource 算子会自动触发，并变为蓝色，日志栏会同时打印算子运行说明如下图所示，表示 TCP 的服务端已经建立完毕。



2. 重新打开一个终端，输入命令：

```
echo "{\"SetPara\":[{\"node_name\":\"Emit\",\"para_name\":\"type\",\"para_value\":\"String\"},{\"node_name\":\"Emit_1\",\"para_name\":\"type\",\"para_value\":\"Pose\"}]}#" | nc localhost 2014
```

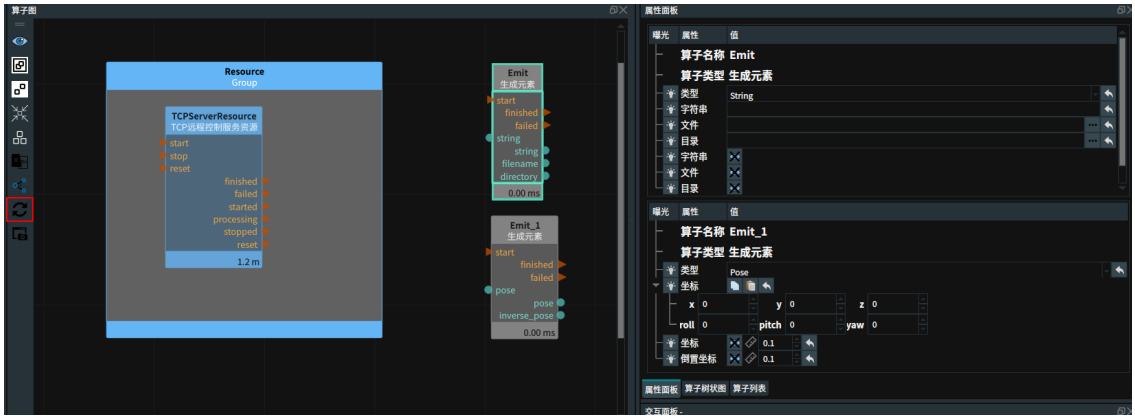
并单击回车键，运行结束后，我们会在这个终端窗口看到回复字符串如下图所示。



3. 但是 RVS 执行结束后，对应的两个 Emit 算子属性以及端口都没有进行更改如下图所示。

4. 此时需要我们手动分别单击一下两个 Emit 算子，然后发现其属性栏已经更新，更新后如下图所示。

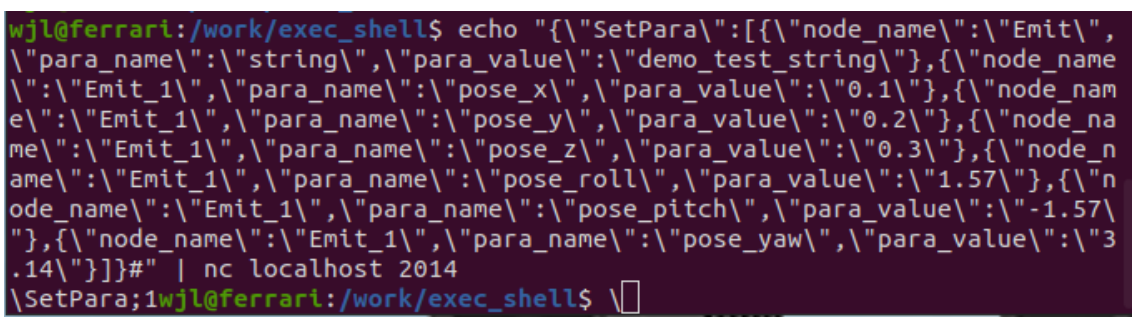
5. 然而两个算子的输入输出端口没有更新，则我们需要单击算子图左侧的更新按钮（下图左侧红色标注的按钮）。更新后如下图所示。



6. 在终端中，继续输入命令

```
echo "{\"SetPara\":
[{\"node_name\":\"Emit\",\"para_name\":\"string\",\"para_value\":\"demo_test_string\"},
{\"node_name\":\"Emit_1\",\"para_name\":\"pose_x\",\"para_value\":\"0.1\"},
{\"node_name\":\"Emit_1\",\"para_name\":\"pose_y\",\"para_value\":\"0.2\"},
{\"node_name\":\"Emit_1\",\"para_name\":\"pose_z\",\"para_value\":\"0.3\"},
{\"node_name\":\"Emit_1\",\"para_name\":\"pose_roll\",\"para_value\":\"1.57\"},
{\"node_name\":\"Emit_1\",\"para_name\":\"pose_pitch\",\"para_value\":\"-1.57\"},
{\"node_name\":\"Emit_1\",\"para_name\":\"pose_yaw\",\"para_value\":\"3.14\"}]}#" | nc localhost
2014
```

并单击回车键，运行结束后，我们会在这个终端窗口看到回复字符串如下图所示。

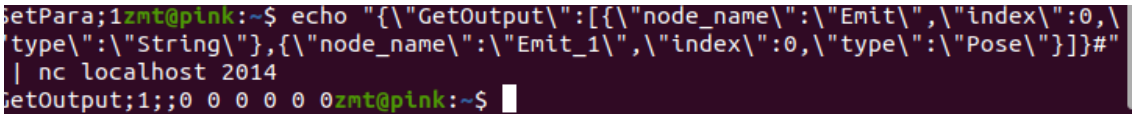


7. 然而两个算子的对应属性值不会立刻更新，需要我们手动各自单击一下两个 Emit 算子。更改后的属性显示如下。

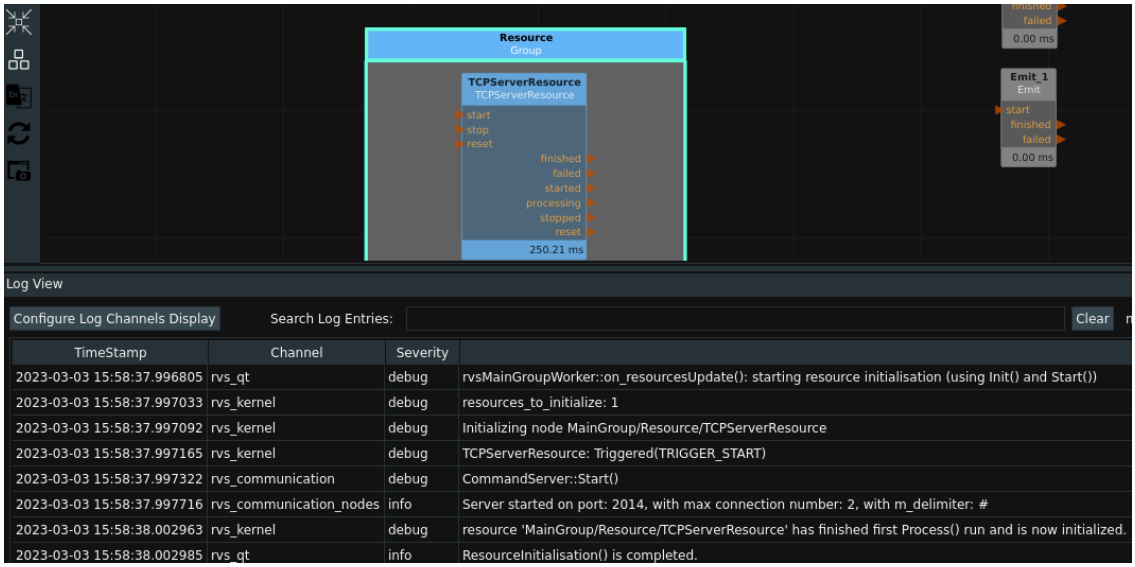
8. 在终端中，继续输入

```
echo "{\"GetOutput\":[{\"node_name\":\"Emit\",\"index\":0,\"type\":\"String\"},
{\"node_name\":\"Emit_1\",\"index\":0,\"type\":\"Pose\"}]}#" | nc localhost 2014
```

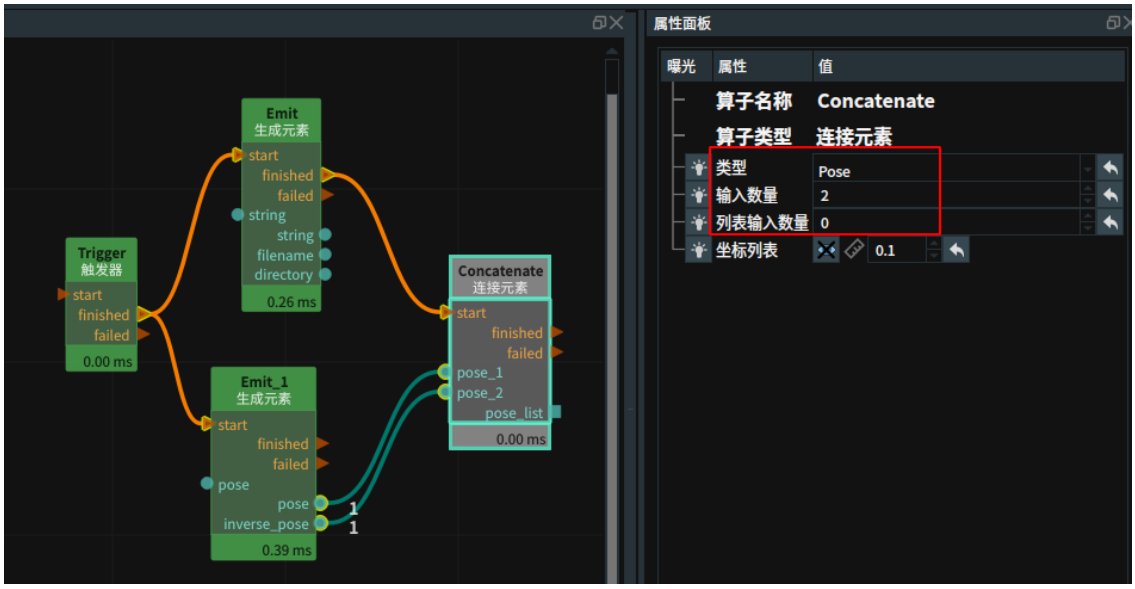
命令并单击回车键，运行结束后，我们会在这个终端窗口看到回复字符串如下图所示。



9. 发现得到的回复值并非刚才设置的数值而是这些参数的默认值 (string 默认为空字符，pose默认为全零)。这是因为两个Emit算子尚未被触发，其数据输出端口尚未更新。此时单击 RVS 的 stop，将一个 Trigger 算子拖入到算子图中，连接触发两个 Emit 算子，然后重新单击 RVS 的运行，并触发执行 Trigger。此时重新发送上述命令，可以得到回复如下：



10. 之后单击 RVS 的 stop，将一个 Concatenate 算子拖入到算子图中，更改其**类型**为 pose ，更改其**输入数量** 为 2 ，更改 **列表输入数量** 为 0，并按照下图连接算子。



11. 重新单击 RVS 的运行，并触发执行 Trigger，此时 Concatenate 算子的输出端口会获得实际数值。
12. 然后在终端中，继续输入命令

```
echo "{\"GetOutput\":[{\"node_name\":\"Concatenate\",\"index\":0,\"type\":\"Pose\"}]}#" | nc localhost 2014
```

并单击回车键，运行结束后，我们会在这个终端窗口看到回复字符串如下图所示。

注意，对于List的回复，最后会附加List中含有的目标总数，如下图的 2 。

```
wjl@ferrari:/work/exec_shell$ echo "{\"GetOutput\":[{\"node_name\":\"Concatenate\",\"index\":0,\"type\":\"Pose\"}]}#"
 | nc localhost 2014
GetOutput;1;0.1 0.2 0.3 1.57 -1.57 3.14;-0.2999205589 -0.09976076335 -0.200238511 1.570795655 -0.002388599329 1.57159
245;2wjl@ferrari:/work/exec_shell$
```

EliteRobotsResource Elite机器人控制资源

EliteRobotsResource 算子属于资源类线程算子，用于在 RVS 的分线程中启动一个 Elite 机器人控制资源。

通过添加 Elite 机器人控制资源，使用 RVS 中 RobotMovement 算子（机器人运动控制工具）和 RobotOperator算子（机器人操作工具）对 Elite 机器人进行运动控制和操作。

算子参数

- **自动启动/auto_start**：用于自动开启资源算子。
  - True：打开 RVS 软件后第一次进入运行状态时自动开启资源线程。
  - False：不自动开启资源线程。
- **启动/start**：用于开启资源算子。
  - True：勾选为True，开启资源线程。
  - False：不启动资源线程。
- **停止/stop**：用于停止资源算子。
  - True：勾选为True，停止资源线程。
  - False：不停止资源线程。
- **重置/reset**：重置该资源。在该资源算子已经运行后，如果重新更改属性参数，需要点击**重置**，然后重新勾选 **启动** 运行。
- **机器人名称/robot_name**：EliteRobot 机器人名称。默认：EliteRobot。
- **机器人模型文件/robot_file**： Elite 机器人模型文件名。文件格式： *.rob 。默认值：data/Elibot/EC66/EC66.rob

  说明：需要提前向图漾技术支持人员申请所需要的机器人数模文件。
- **工具模型文件/tool_file**：夹具吸盘等工具等文件名。
- **最大关节速度/max_joint_velocity**：用于控制机器人 MoveJoints 时的关节最大速度。单位：弧度每秒。默认值：100。
- **最大关节加速度/max_joint_acceleration**：用于控制机器人 MoveJoints 时的关节最大加速度。单位：弧度每平方秒。默认值：10。
- **最大线性速度/max_linear_velocity**：用于控制机器人线性运动时的最大线速度。单位：米每秒。默认值：100。
- **最大线性加速度/max_linear_acceleration**：用于控制机器人线性运动时的线速最大加速度。单位：米每平方秒。默认值：3。
- **机器人/robot**：设置 Elite 机器人数模在 3D 视图中的可视化属性。
  -  打开 Elite 机器人仿真模型可视化。
  -  关闭 Elite 机器人仿真模型可视化。
- **机器人更新/robot_update**：设置机器人是否实时加载更新。默认不勾选，该参数主要用于生成机器人模型时调参查看效果。
- **机器人IP/arm_ip**：填写 Elite 机器人的通讯 IP。 默认值：192.168.1.200。
- **机器人端口/arm_port**：填写 Elite 机器人的通讯 IP 的端口号。 默认值：8055。

控制信号输入输出

说明：Resource 资源类算子的控制信号端口无法触发使用。

功能演示

使用 EliteRobotsResource 加载控制 Elite 机器人通过 Joint 移动机器人。

说明：本案例需要有 Elite 机器人才可进行。

步骤1：算子准备

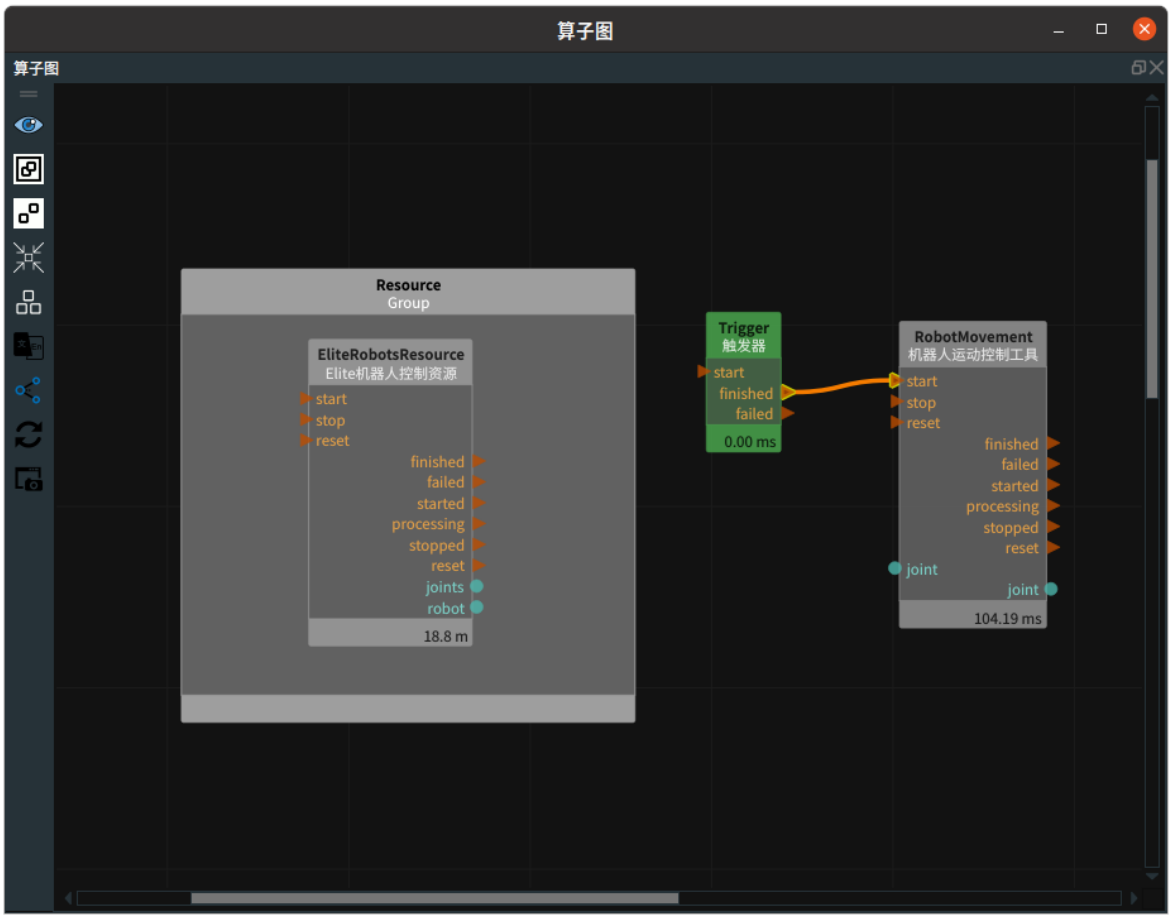
添加 EliteRobotsResource、Trigger、RobotMovement 算子至算子图。

步骤2：设置算子参数

1. 设置 EliteRobotsResource 算子参数：
    - 自动启动 → True
    - 机器人模型文件 → Elite 数模文件名（example_data/data/EC66/EC66.rob）
    - 工具模型文件 → 吸盘工具文件名（example_data/data/Tool/EliteRobotSucker1.tool.xml）
    - 机器人 → 
2. 设置 RobotMovement 算子参数：
    - 机器人资源名称 → EliteRobotsResource
    - 类型 → MoveJoint
    - 关节 → 0.0863019 -1.61578 1.61096 -1.50041 1.585874 -3.12127

    说明：移动机器人需要打开机器人的远程控制模式（模式根据机器人品牌型号而定）。

步骤3：连接算子

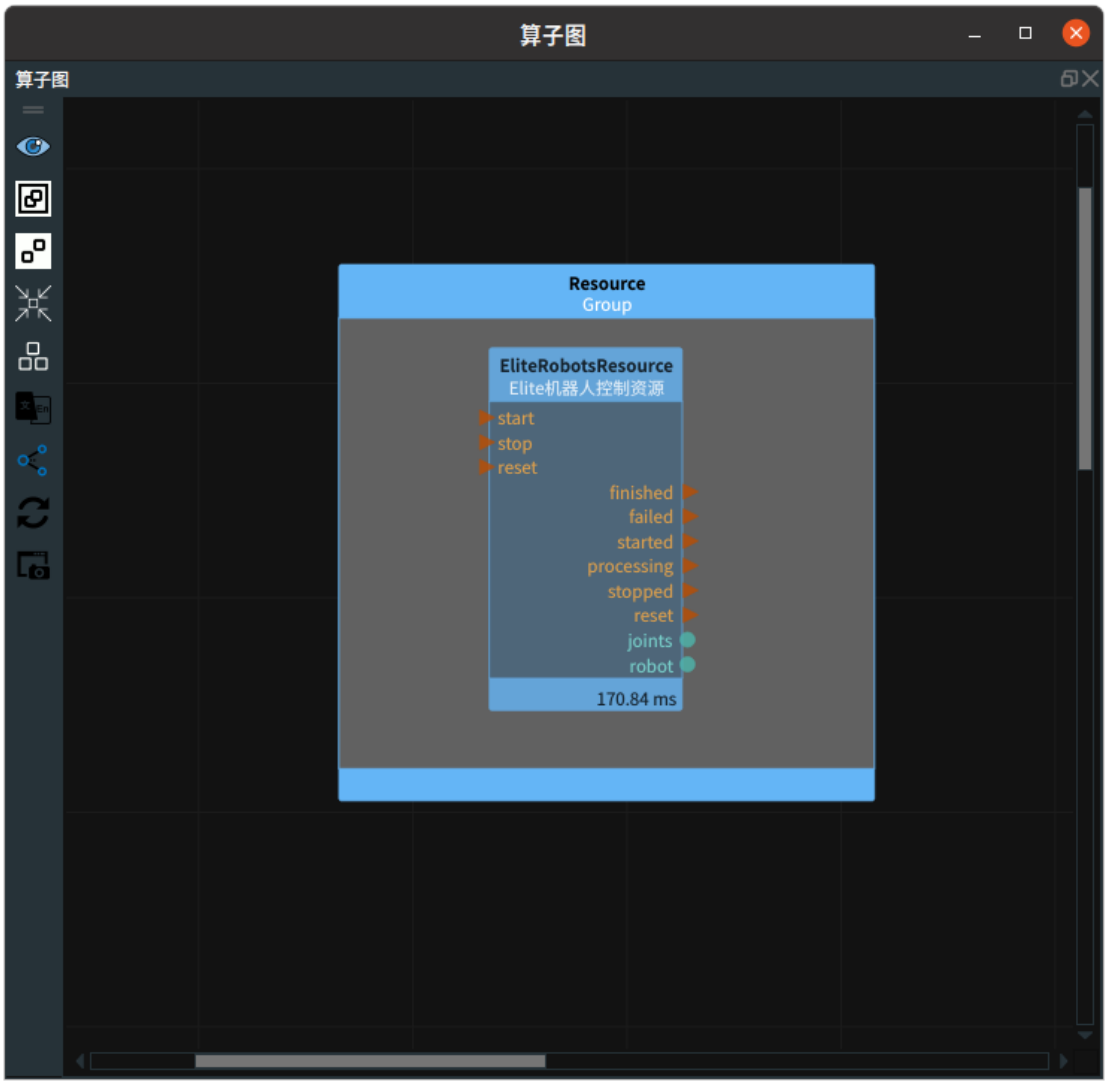


步骤4：运行

点击 RVS 运行按钮， EliteRobotsResource 资源算子启动成功。

运行结果

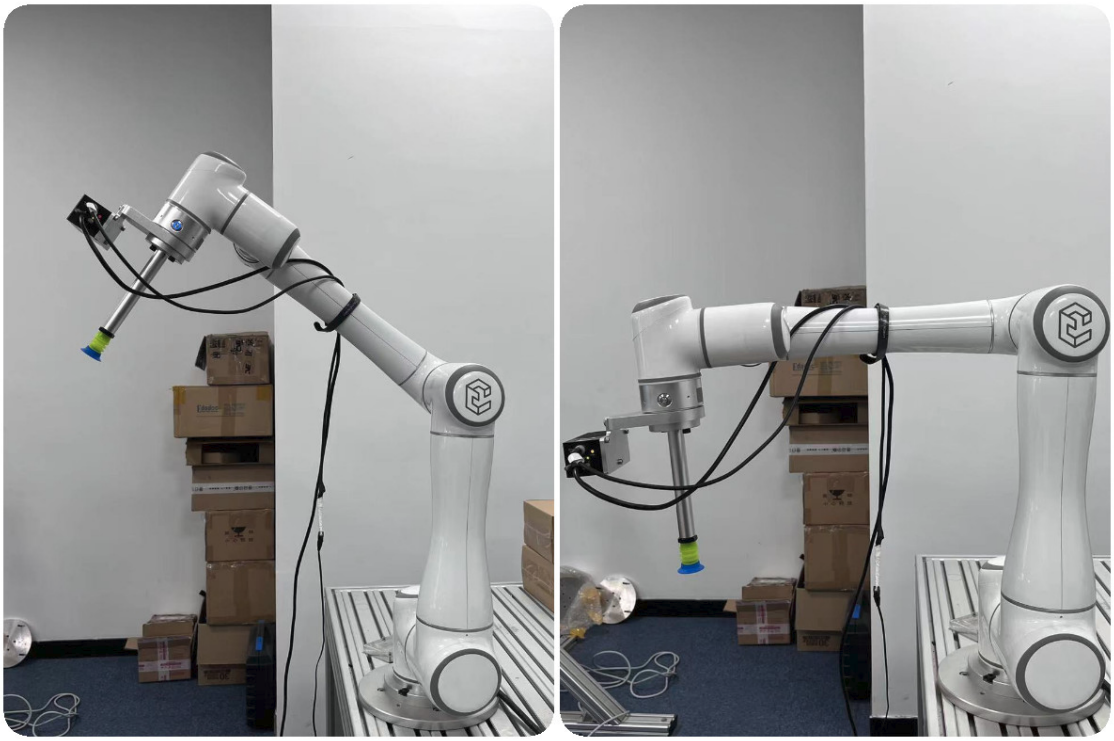
1. 打开 RVS 的运行按钮，EliteRobotsResource 算子会自动触发，并变为蓝色，表示 Elite 机器人控制资源算子启动成功。



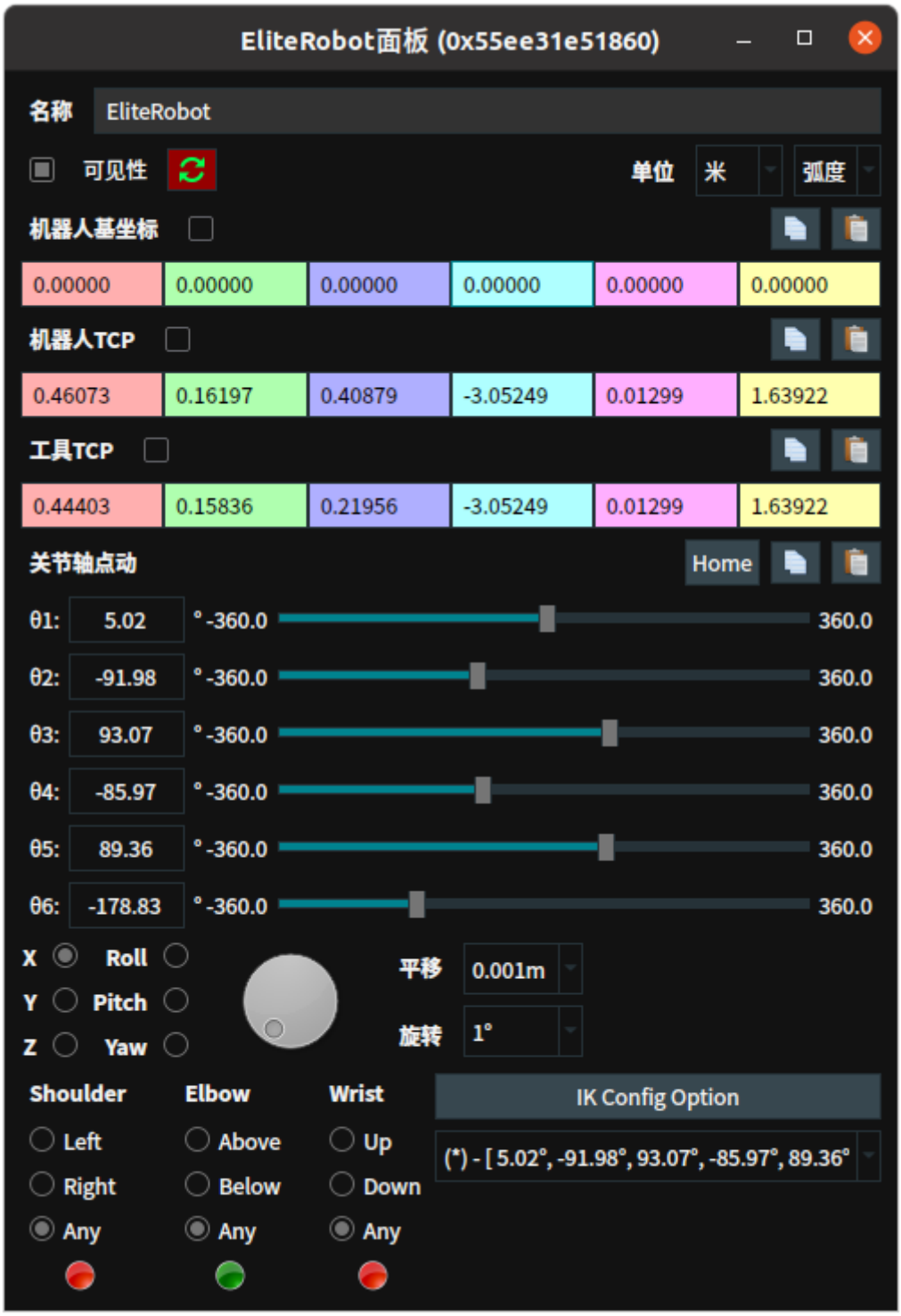
2. 触发 Trigger算子，算子运行成功后，在 3D 视图中显示加载 Elite 机器人数模移动前后的对比图。左图为移动前的机器人数模状态，右图为移动后的机器人数模状态。

3. 下图为 Elite 机器人移动前后的对比图。

   左图为移动前的机器人状态，右图为移动后的机器人状态。



4. **双击** 3D 视图中为 Elite 机器人数模，弹出机器人面板，可查看此时机器人关节轴数值 。


EliteRobot面板 (0x55ee31e51860)
名称
EliteRobot
可见性
单位
米
弧度
机器人基坐标
0.00000
0.00000
0.00000
0.00000
0.00000
0.00000
机器人TCP
0.46073
0.16197
0.40879
-3.05249
0.01299
1.63922
工具TCP
0.44403
0.15836
0.21956
-3.05249
0.01299
1.63922
关节轴点动
Home
θ1:
5.02
° -360.0
360.0
θ2:
-91.98
° -360.0
360.0
θ3:
93.07
° -360.0
360.0
θ4:
-85.97
° -360.0
360.0
θ5:
89.36
° -360.0
360.0
θ6:
-178.83
° -360.0
360.0
X
Roll
Y
Pitch
Z
Yaw
平移
0.001m
旋转
1°
Shoulder
Left
Right
Any
Elbow
Above
Below
Any
Wrist
Up
Down
Any
IK Config Option
(*) - [5.02°, -91.98°, 93.07°, -85.97°, 89.36°

UniversalRobotsResource UR机器人控制资源

UniversalRobotsResource 算子属于资源类线程算子，用于在 RVS 的分线程中启动一个 UR 机器人控制资源。

通过添加 UR 机器人控制资源，使用 RVS 中 RobotMovement 算子（机器人运动控制工具）和 RobotOperator算子（机器人操作工具）对 UR 机器人进行运动控制和操作。

算子参数

- **自动启动/auto_start**：用于自动开启资源算子。
  - True：打开 RVS 软件后第一次进入运行状态时自动开启资源线程。
  - False：不自动开启资源线程。
- **启动/start**：用于开启资源算子。
  - True：勾选为True，开启资源线程。
  - False：不启动资源线程。
- **停止/stop**：用于停止资源算子。
  - True：勾选为True，停止资源线程。
  - False：不停止资源线程。
- **重置/reset**：重置该资源。在该资源算子已经运行后，如果重新更改属性参数，需要点击**重置**，然后重新勾选 **启动** 运行。
- **机器人名称/robot_name**：UR机器人名称。默认：UR。
- **机器人模型文件/robot_file**： UR 机器人模型文件名。文件格式： *.rob 。默认值：data/UR/UR5/UR5.rob

  说明：需要提前向图漾技术支持人员申请所需要的机器人数模文件。
- **工具模型文件/tool_file**：夹具吸盘等工具等文件名。
- **最大关节速度/max_joint_velocity**：用于控制机器人 MoveJoints 时的关节最大速度。单位：弧度每秒。默认值：3.1。
- **最大关节加速度/max_joint_acceleration**：用于控制机器人 MoveJoints 时的关节最大加速度。单位：弧度每平方秒。默认值：10。
- **最大线性速度/max_linear_velocity**：用于控制机器人线性运动时的最大线速度。单位：米每秒。默认值：1。
- **最大线性加速度/max_linear_acceleration**：用于控制机器人线性运动时的线速最大加速度。单位：米每平方秒。默认值：3。
- **机器人/robot**：设置 UR 机器人数模在 3D 视图中的可视化属性。
  -  打开 UR 机器人仿真模型可视化。
  -  关闭 UR 机器人仿真模型可视化。
- **机器人更新/robot_update**：设置机器人是否实时加载更新。默认不勾选，该参数主要用于生成机器人模型时调参查看效果。
- **机器人IP/robot_ip**：填写 UR 机器人的通讯 IP。 默认值：127.0.0.1。

控制信号输入输出

说明：Resource 资源类算子的控制信号端口无法触发使用。

功能演示

与 EliteRobotsResource 资源算子功能演示类似，请参考该算子功能演示模块进行操作。

TyCameraResource 图漾相机资源

TyCameraResource 算子属于资源类线程算子，用于在 RVS 的分线程中启动一个图漾相机资源。

通过添加图漾相机资源，实现图漾相机图像和点云的采集。

算子参数

- **自动启动/auto_start**：用于自动开启资源算子。
  - True：打开 RVS 软件后第一次进入运行状态时自动开启资源线程。
  - False：不自动开启资源线程。
- **启动/start**：用于开启资源算子。
  - True：勾选为True，开启资源线程。
  - False：不启动资源线程。
- **停止/stop**：用于停止资源算子。
  - True：勾选为True，停止资源线程。
  - False：不停止资源线程。
- **重置/reset**：重置该资源。该资源算子已经运行后，如果重新更改属性参数，需要点击**重置**，然后重新勾选 **启动** 运行。
- **相机型号/camera_model**：默认不填写。连接设备后，根据相机ID显示对应的相机型号。
- **相机ID/camera_id**：默认不填写。在打开该资源线程后会自动查找当前工控机所在网段的所有图漾相机并将这些设备的硬件信息打印在 terminal 中，然后随机选中一台设备进行连接。

  说明：如果需要指定连接某台图漾相机设备，则需要将该设备的编码填写到**相机ID**选项。编码可以在相机设备硬件上查看，也可以在上述 terminal 的打印信息中查找。
- **深度图像模式/selected_depth_mode**：默认不填写。连接设备后会自动选择并显示深度图的的图像尺寸。

  说明：影响最终形成的点云图的尺寸。后续可以更改，但是需要设置**重置**选项
- **彩色图像模式/selected_rgb_mode**：默认不填写。连接设备后会自动选择并显示2D图的图像尺寸。

  说明：后续可以更改，但是需要设置**重置**选项
- **畸变矫正/undistortion**：设置是否对 rgb 进行去畸变处理后再输出。
  - True：对 rgb 进行去畸变处理。
  - False：对 rgb 不进行去畸变处理。
- **输出彩色图像/output_color**：设置是否输出彩色图像。
  - True：输出。
  - False：不输出。
- **输出深度图像/output_depth**：设置是否输出深度图像。
- **输出左IR图/output_ir_left**：设置是否输出左IR图。
- **输出右IR图/output_ir_right**：设置是否输出右IR图。
- **输出点云/output_cloud**：设置是否输出点云。需要勾选**输出深度图像**和**输出点云**。
- **激光自动控制/laser_auto_ctrl**：激光自适应模式。
  - True：激光自动控制。
  - False：根据**激光强度**数值设置激光的强度。
- **激光强度/laser_power**：设置激光强度。取值范围：[0,100]。
- **彩色图像Isp/rgb_isp**：设置是否对 rgb 图像的预处理。

- **点云比例/cloud_scale**：调整该值设置点云在 RVS 中的比例。

  说明：相机 SDK 中点云单位为毫米，RVS 中点云单位为米，默认值：0.001，表示将相机 SDK 中点云毫米转换为米在 RVS 中输出。

- **相机深度图标定文件/depth_calib_file**：存放在相机设备内的深度相机的出厂标定参数，开启资源线程后会自动读取该标定参数并按照该参数对应的文件路径保存成文件。
- **相机彩色图标定文件/color_calib_file**：存放在相机设备内的2D 相机的出厂标定参数，开启资源线程后会自动读取该标定参数并按照该参数对应的文件路径保存成文件。

  文件内容说明

  color_calib_file

  第一行（下图红色区域）表示相机的 RGB 镜头在进行出厂标定时所采用的镜头分辨率。实际使用相机时，只有将相机分辨率同这里的数值保持一致，才可以直接使用下述的内参矩阵。否则需要对应的进行数值缩放（但是会有一定的精度损失）。

  第二行（下图蓝色区域）表示相机 RGB 镜头的内参。总共 9 个数值，是一个按行排列的 3 * 3 矩阵。矩阵形式为 [fx,0,cx,0,fy,cy,0,0,1] ,其中 fx 表示 x 轴方向的相机焦距 (单位为像素)，其中 cx 表示 x 轴方向的相机中心(单位为像素)；fy 、cy 类似。

  第三行（下图绿色区域）表示相机 RGB 镜头的外参，即 RGB 镜头到左 IR 镜头的空间坐标系转换矩阵。总共 16 个数值，是一个按行排列的 4 * 4 矩阵。左上角的 3 * 3 模块表示旋转，右上角的 1 * 3 表示平移。平移值的单位为毫米。

  第四行（下图黄色区域）表示相机 RGB 镜头的畸变参数，总共 12 个数值。可以用来对原始的RGB图像进行畸变校正。

  另外，关于 **depth_calib_file**，其说明同上述说明保持一致。特别的，由于深度相机是虚拟相机，所以其畸变参数以及外参都是零矩阵。

- **触发模式/trigger_mode**：设置触发方式。
  - TY_SOFTWARE_TRIGGER_MODE：为软件触发方式，每一次发送 trigger 命令时相机采集一帧图像
  - TY_HARDWARE_TRIGGER_MODE：为硬件触发方式，通过硬件触发相机采集。
  - TY_TRIGGER_MODE_OFF：则为相机不停的采集图像，每一次发送 trigger 命令时，相机回传一帧数据给 RVS 。TY_TRIGGER_MODE_OFF 时间上效率更高，但请注意，t 时刻发送 trigger 获得的图像并非是 t 时刻采集的，而是 t-t0 时刻采集的，t0 由相机型号来定，一般大于 0.5 秒。
- **是否再发/resend**：
  - True：在每一次相机回传 RVS 数据发生丢包时会重新发送。
  - False：在每一次相机回传 RVS 数据发生丢包时不会重新发送。
- **相机外参坐标/extrinsic_pose**：无法修改，是从当前相机资源载入后，从 sdk 中读取当前相机的 rgb 相机的外参（即 rgb 相机到左 ir 深度相机的转换矩阵）并转换成 pose 形式进行展示。
- **相机参数展开/camera_params_exposure**：是否暴露相机的一些功能属性参数。
  - True：在属性栏展开相机的功能属性参数。
  - False：默认不展开相机的功能属性参数。

控制信号输入输出

说明：Resource 资源类算子的控制信号端口无法触发使用。

功能演示

使用 TyCameraResource 进行图漾相机型号为 FS820-E1 的点云和图像单次采集。

说明：本案例需要有图漾相机才可进行。

步骤1：算子准备

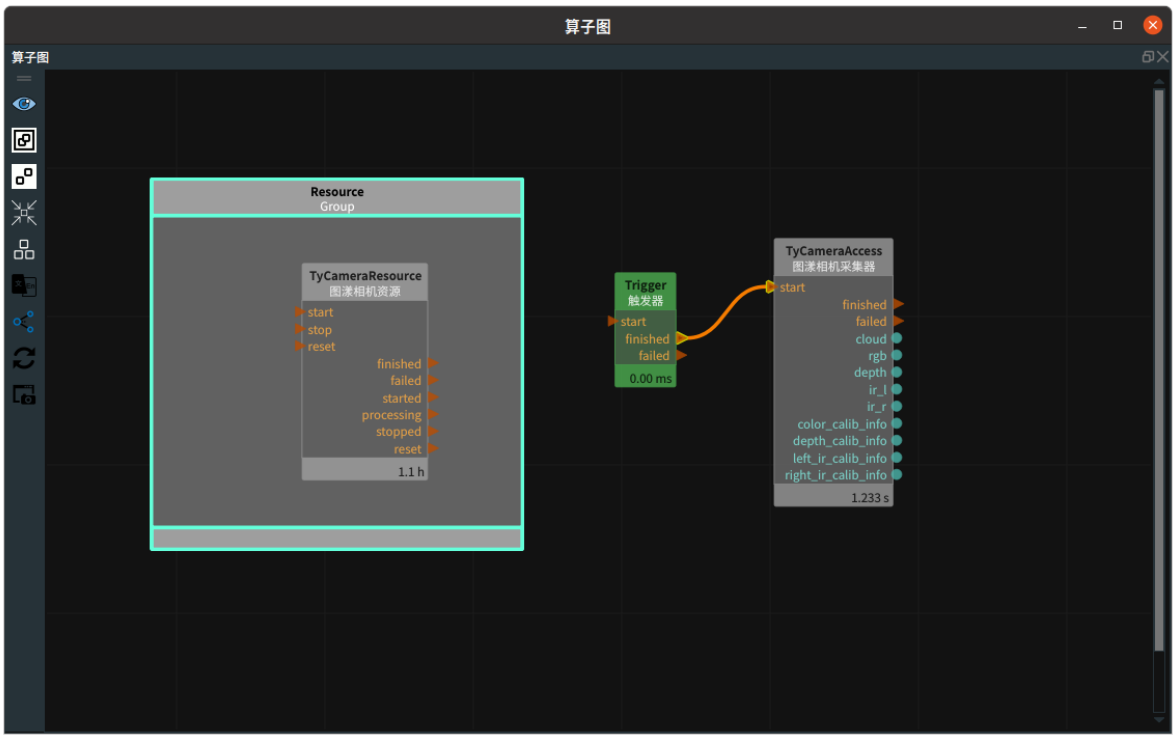
添加 TyCameraResource 、Trigger 、TyCameraAccess 算子至算子图。

步骤2：设置算子参数

1. 设置 TyCameraResource 算子参数：
   - 自动启动 → True
2. 设置 TyCameraAccess 算子参数：

- 点云 → 
- 彩色 → 

步骤3：连接算子



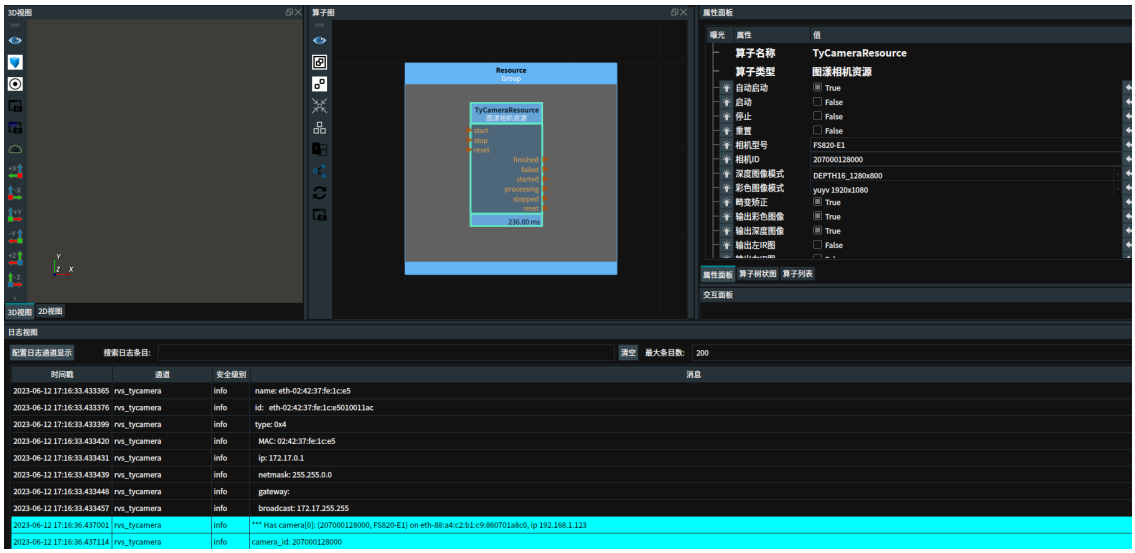
步骤4：运行

1. 点击 RVS 运行按钮， TyCameraResource 资源算子连接成功。
2. 成功后，触发 Trigger 算子。

   说明：若 Trigger 算子的**循环** 属性勾选为True，则会连续采集图像。

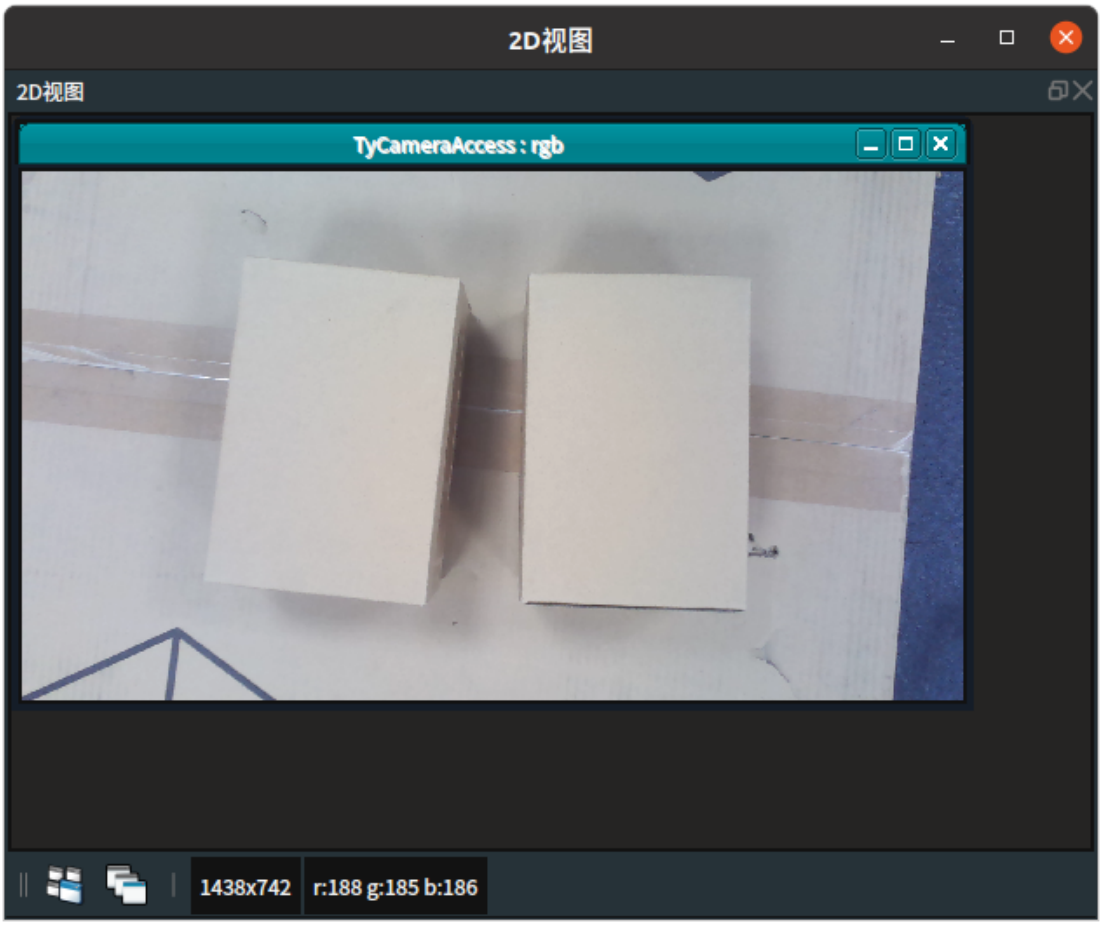
运行结果

1. 打开 RVS 的运行按钮，TyCameraResource 算子会自动触发，并变为蓝色，日志栏会同时打印算子运行说明如下图所示，表示连接图漾相机成功。

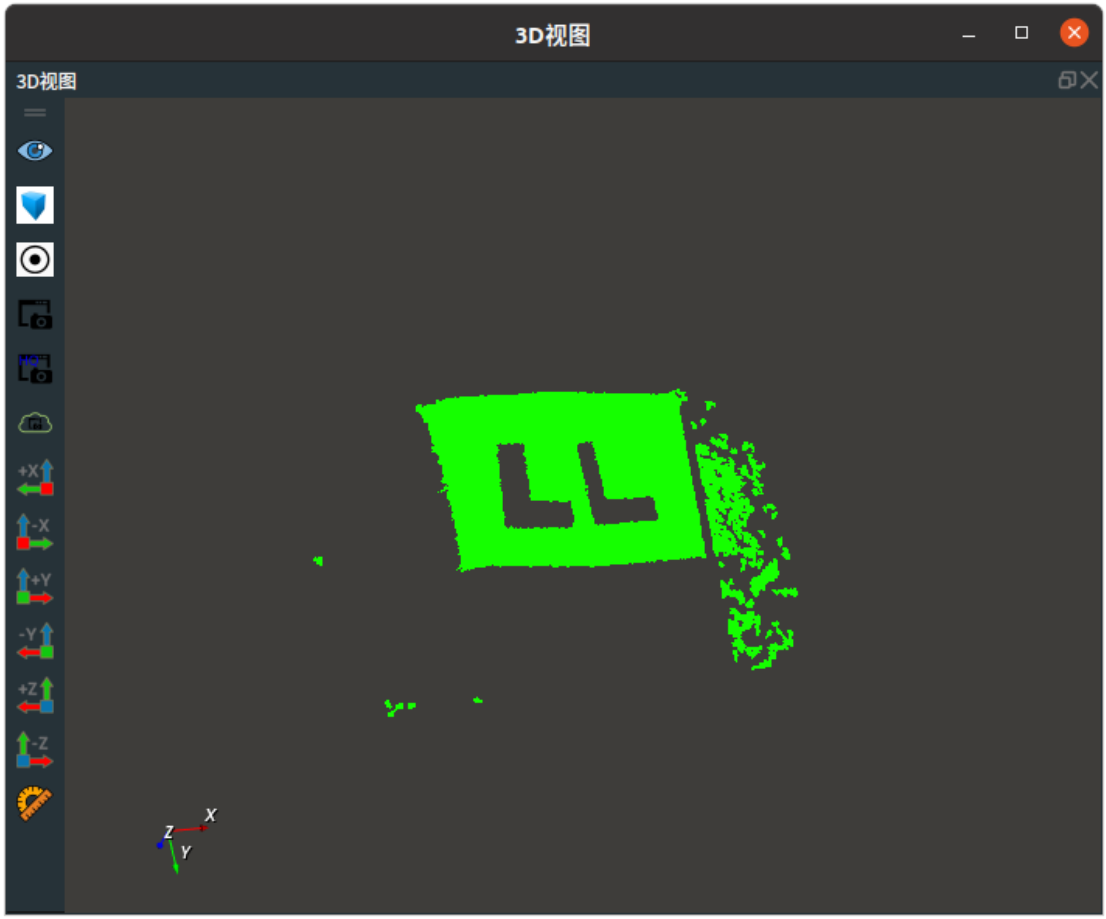


2. 下图为相机资源连接前后属性面板的对比图。

3. 当触发 Trigger 算子后，TycameraAccess 运行完成后，在 2D 视图中显示采集时的图像。



4. 在 3D 视图中显示采集时的点云。


3D视图
3D视图
+X
-X
+Y
-Y
+Z
-Z
X
Z
Y

SimulatedRobotResource 机器人仿真控制资源

SimulatedRobotResource 算子属于资源类线程算子，用于在 RVS 的分线程中启动一个机器人相机仿真控制资源。

通过添加机器人仿真控制资源，加载数模文件，可以在 RVS 软件内控制机器人仿真移动与姿态控制。这样可以在没有实际硬件的情况下进行仿真和测试。

算子参数

- **自动启动/auto_start**：用于自动开启资源算子。
  - True：打开 RVS 软件后第一次进入运行状态时自动开启资源线程。
  - False：不自动开启资源线程。
- **启动/start**：用于开启资源算子。
  - True：勾选为True，开启资源线程。
  - False：不启动资源线程。
- **停止/stop**：用于停止资源算子。
  - True：勾选为True，停止资源线程。
  - False：不停止资源线程。
- **重置/reset**：重置该资源。资源算子已经运行后，如果重新更改属性参数，需要点击**重置**，然后重新勾选 **启动** 运行。
- **机器人名称/robot_name**：可填写仿真机器人名称。默认：SimulatedRobot。
- **机器人模型文件/robot_file**：机器人数模文件名。文件格式： *.rob 。

  说明：需要提前向图漾技术支持人员申请所需要的机器人数模文件。
- **工具模型文件/tool_file**：夹具吸盘等工具等文件名。
- **最大关节速度/max_joint_velocity**：用于控制机器人 MoveJoints 时的关节最大速度。单位：弧度每秒。默认值：3.1。
- **最大关节加速度/max_joint_acceleration**：用于控制机器人 MoveJoints 时的关节最大加速度。单位：弧度每平方秒。默认值：10。
- **最大线性速度/max_linear_velocity**：用于控制机器人线性运动时的最大线速度。单位：米每秒。默认值：1。
- **最大线性加速度/max_linear_acceleration**：用于控制机器人线性运动时的线速最大加速度。单位：米每平方秒。默认值：3。
- **机器人/robot**：设置机器人数模在 3D 视图中的可视化属性。
  -  打开机器人仿真模型可视化。
  -  关闭机器人仿真模型可视化。
- **机器人更新/robot_update**：设置机器人是否实时加载更新。默认不勾选，该参数主要用于生成机器人模型时调参查看效果。

控制信号输入输出

说明：Resource 资源类算子的控制信号端口无法触发使用。

功能演示

使用 SimulatedRobotResource 加载 UR5 机器人数模。

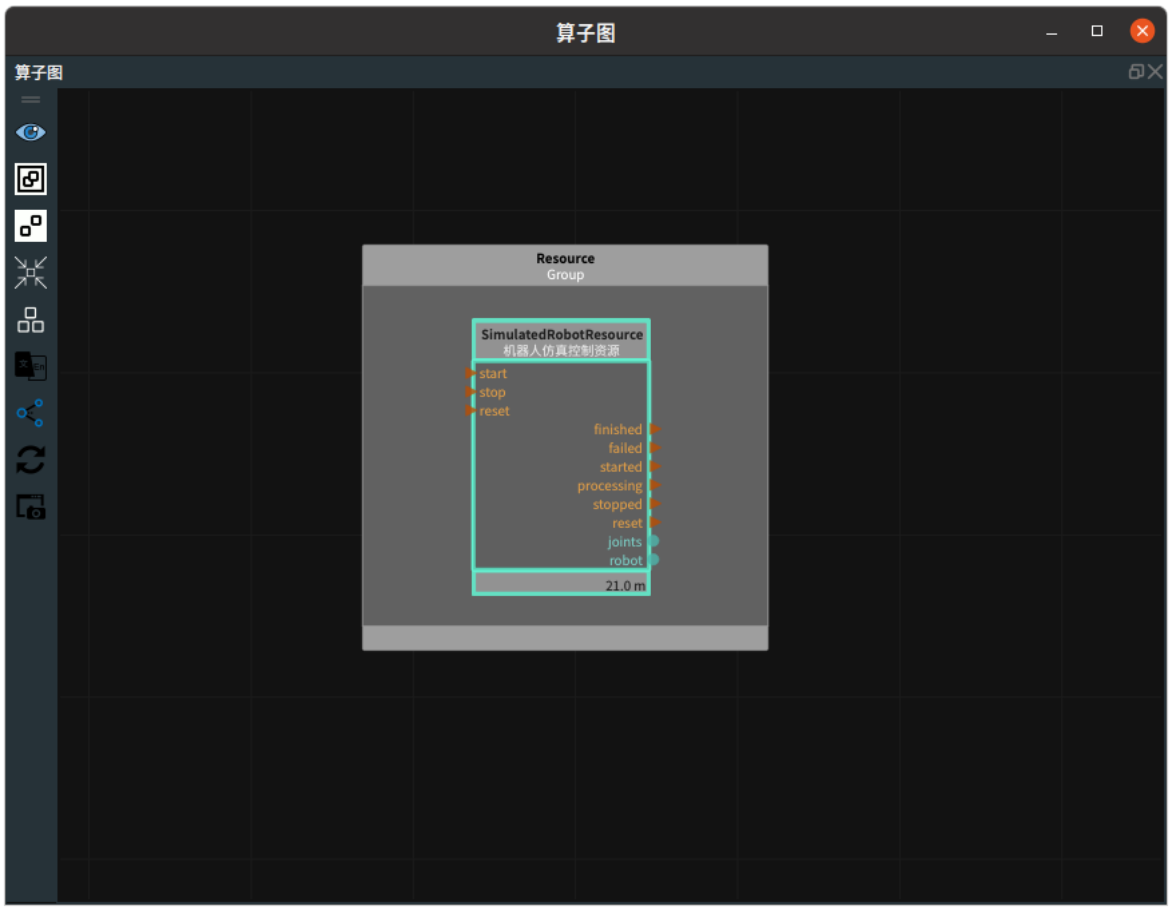
步骤1：算子准备

添加 SimulatedRobotResource 算子至算子图。

步骤2：设置算子参数

1. 设置 SimulatedRobotResource 算子参数：
    - 自动启动 → True
    - 机器人模型文件 → UR5 数模文件名（example_data/UR5/UR5.rob）
    - 工具模型文件 → 吸盘工具文件名（example_data/Tool/EliteRobotSucker1.tool.xml）
    - 机器人 → 

步骤3：连接算子

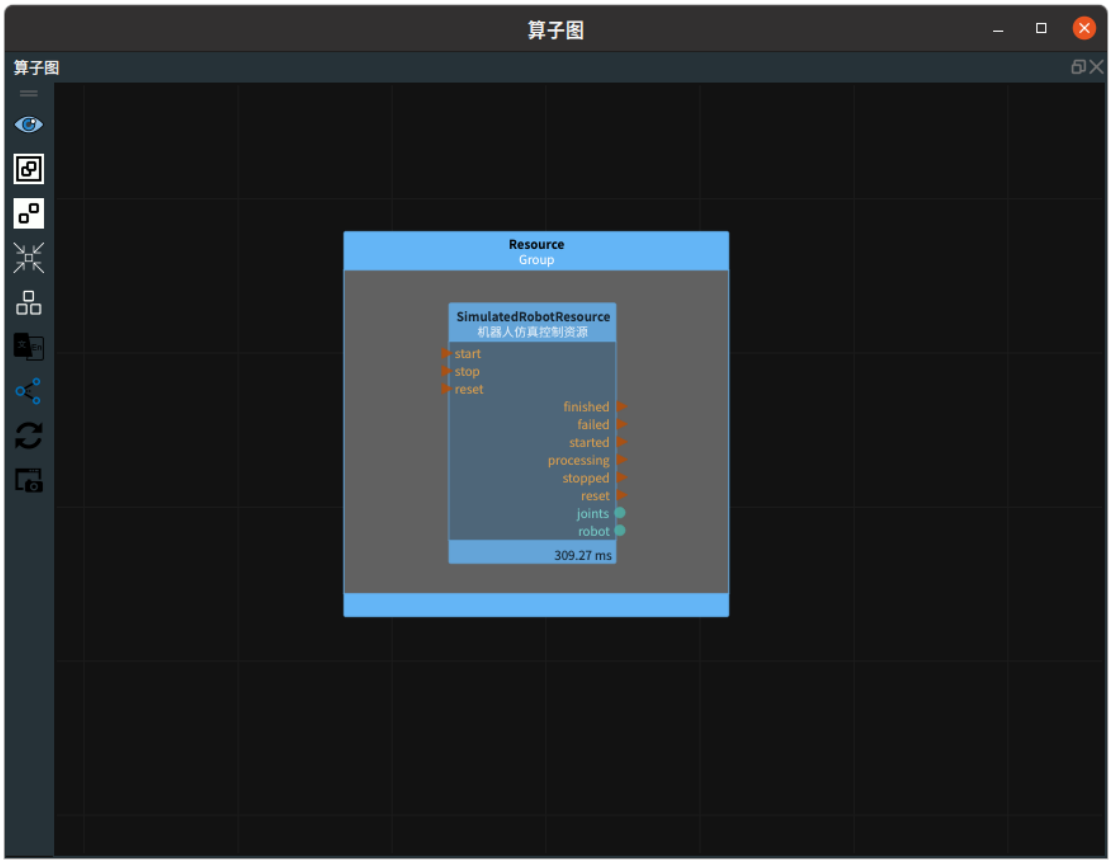


步骤4：运行

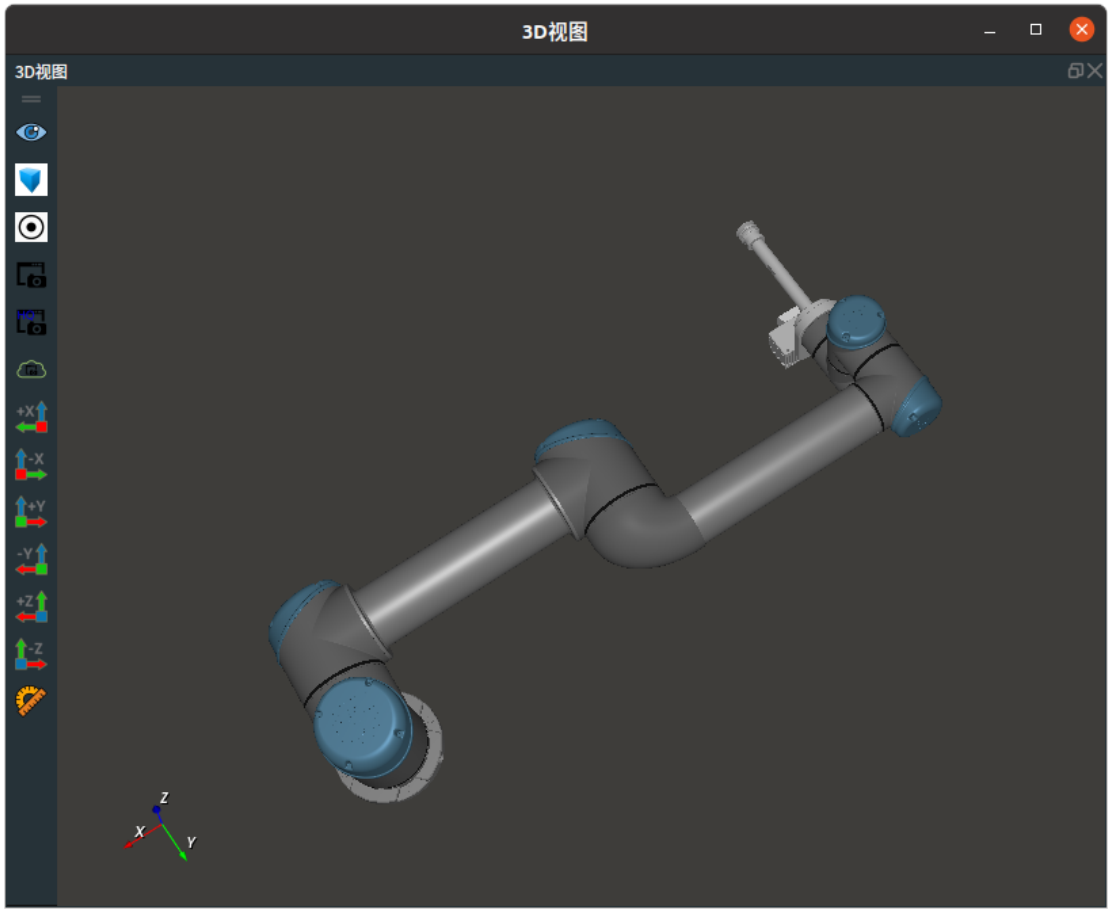
点击 RVS 运行按钮， SimulatedRobotResource 资源算子启动成功。

运行结果

1. 打开 RVS 的运行按钮，SimulatedRobotResource 算子会自动触发，并变为蓝色，表示机器人仿真控制资源算子启动成功。

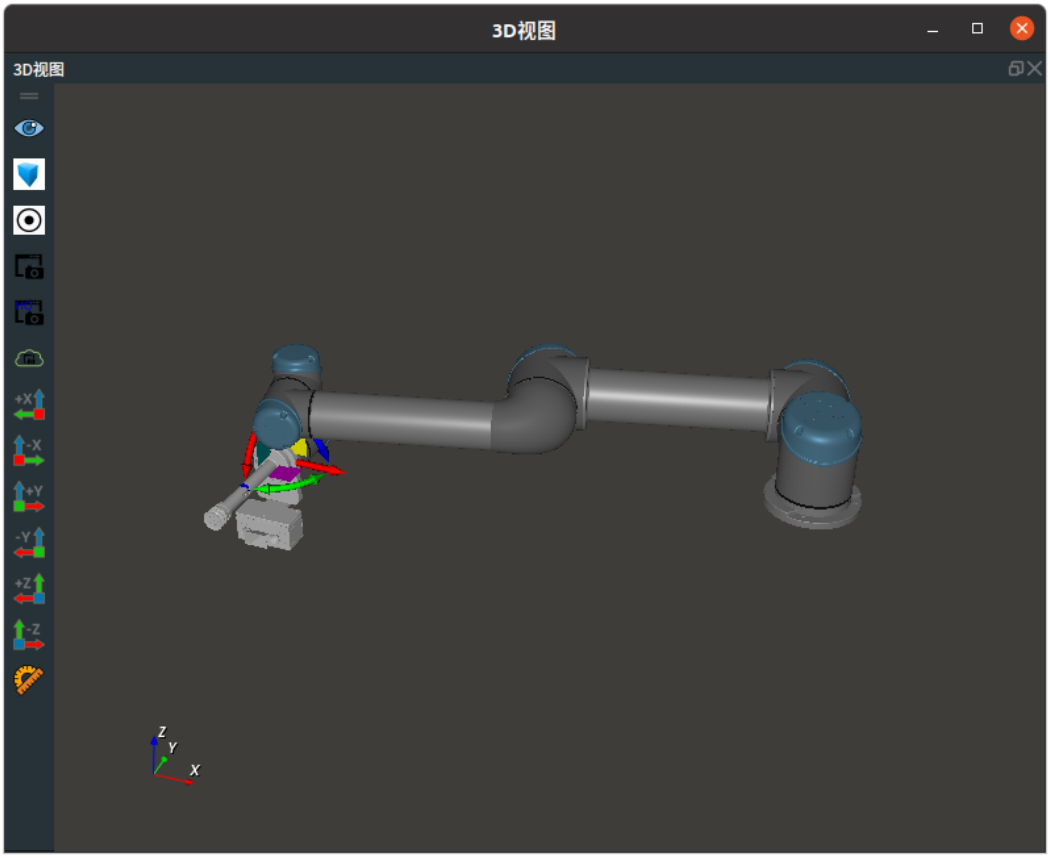


2. 在 3D 视图中显示加载 UR5 机器人和其吸盘工具。



3. **双击**机器人弹出机器人控制面板。调整参数控制机器人。

- 机器人基座标、机器人TCP、工具TCP 勾选后，在 3D 视图中显示对应坐标。
- 关节值编辑栏（黄色框）输入对应的角度来调整机器人关节角度。
- 直接拖动右侧的浮标（绿色框）控制机器人关节转动。
- 主动控制机器人移动到某个位置：勾选 Motion（紫色框），勾选机器人TCP，查看到机器人末端的 TCP pose 姿态。拖动下图的箭头，可以带动整个机器人进行移动。移动过程中的机器人姿态信息可以随时在控制面板中查阅。


3D视图
3D视图
+X
-X
+Y
-Y
+Z
-Z
Z
Y
X

TyCameraSimResource 图漾相机仿真资源

TyCameraSimResource 算子属于资源类线程算子，用于在 RVS 的分线程中启动一个图漾相机仿真资源。

如果没有图漾相机时，可以通过添加图漾相机仿真资源来实现加载采集的点云、彩色图、深度图的功能。这样可以在没有实际硬件的情况下进行仿真和测试。

算子参数

- **自动启动/auto_start**：用于自动开启资源算子。
  - True：打开 RVS 软件后第一次进入运行状态时自动开启资源线程。
  - False：不自动开启资源线程。
- **启动/start**：用于开启资源算子。
  - True：勾选为True，开启资源线程。
  - False：不启动资源线程。
- **停止/stop**：用于停止资源算子。
  - True：勾选为True，停止资源线程。
  - False：不停止资源线程。
- **重置/reset**：重置该资源。在该资源算子已经运行后，如果重新更改属性参数，需要点击**重置**，然后重新勾选 **启动** 运行。
- **相机ID/camera_id**：填入仿真图漾相机 ID 。默认：tycam_sim。
- **输出彩色图像/output_color**：设置是否输出彩色图像。
  - True：输出。
  - False：不输出。
- **输出深度图像/output_depth**：设置是否输出深度图像。
- **输出点云/output_color**：设置是否输出点云。
- **彩色图像文件/color_image_file**：加载彩色图像文件名。默认：rgb.png。
- **深度图像文件/depth_image_file**：加载深度图像文件名。默认：depth.png。
- **点云文件/pointcloud_file**：加载点云文件名。默认：cloud.pcd。

控制信号输入输出

说明：Resource 资源类算子的控制信号端口无法触发使用。

功能演示

使用 TyCameraSimResource 加载使用图像相机资源采集的点云、彩色图、深度图。

步骤1：算子准备

添加 TyCameraSimResource 、Trigger 、TyCameraAccess 算子至算子图。

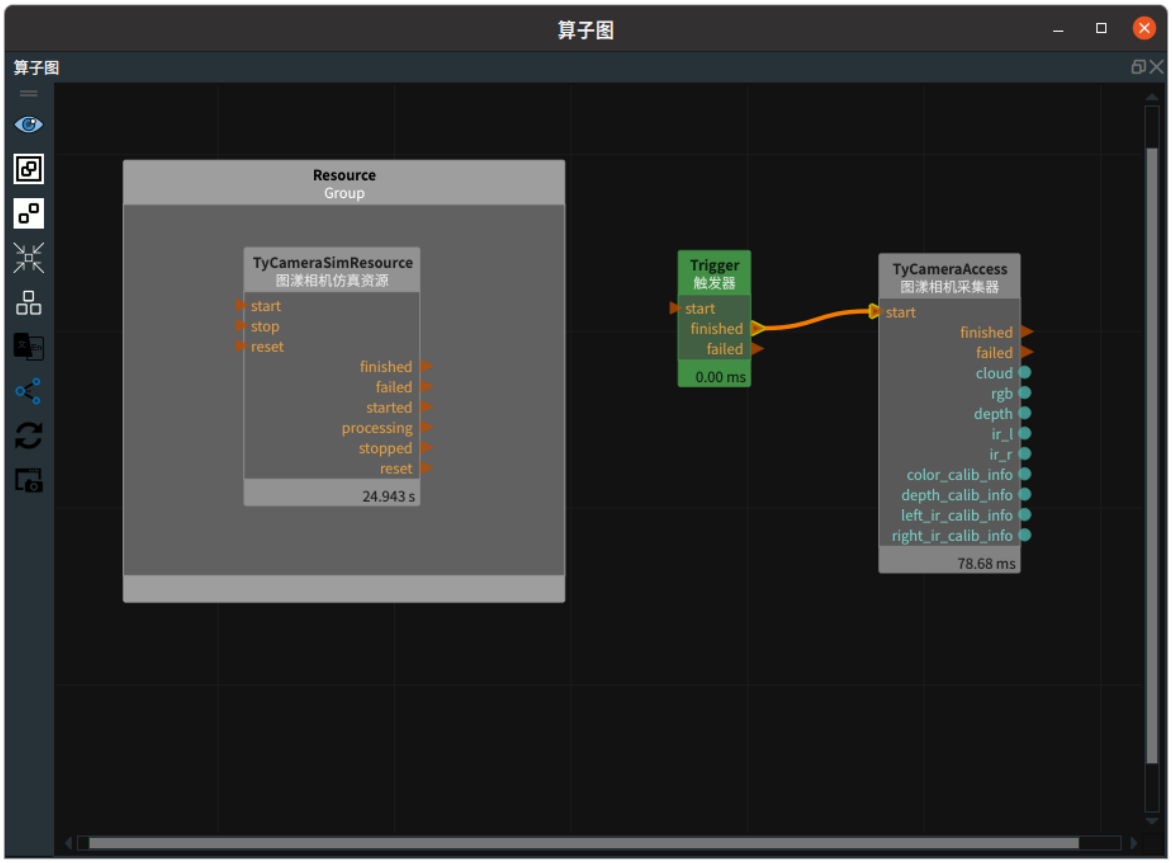
步骤2：设置算子参数

1. 设置 TyCameraSimResource 算子参数：
   - 自动启动 → True
   - 彩色图像文件 →彩色图像文件名（example_data/TyCameraResource/rgb.png）
   - 深度图像文件 → 深度图像文件名（example_data/TyCameraResource/depth.png）

- 点云文件 → 点云文件名（example_data/TyCameraResource/cloud.pcd）

2. 设置 TyCameraAccess 算子参数：
   - 相机资源 → TyCameraSimResource
   - 点云 → 
   - 彩色 → 
   - 深度 → 

步骤3：连接算子

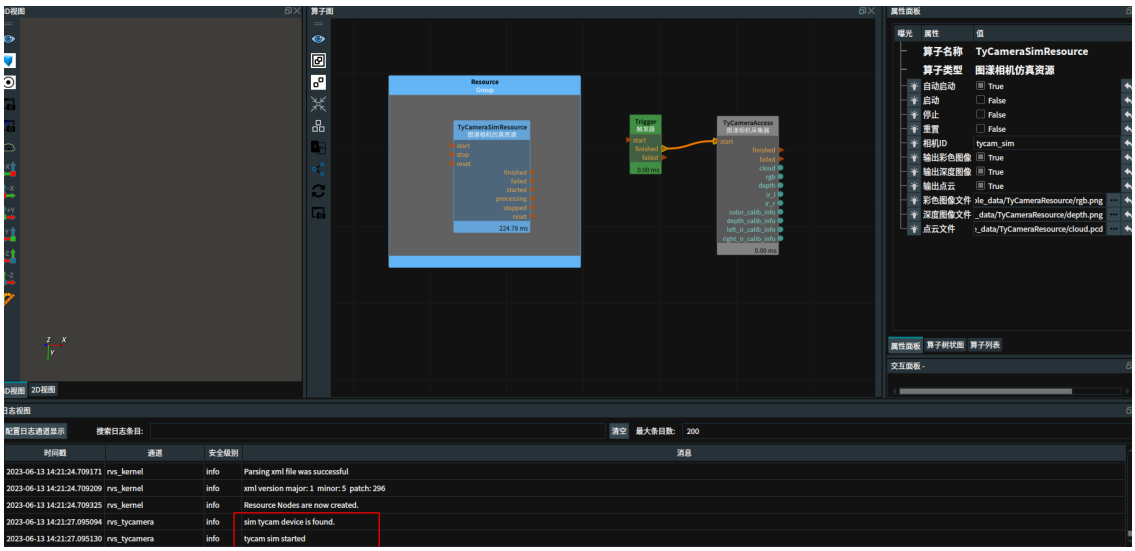


步骤4：运行

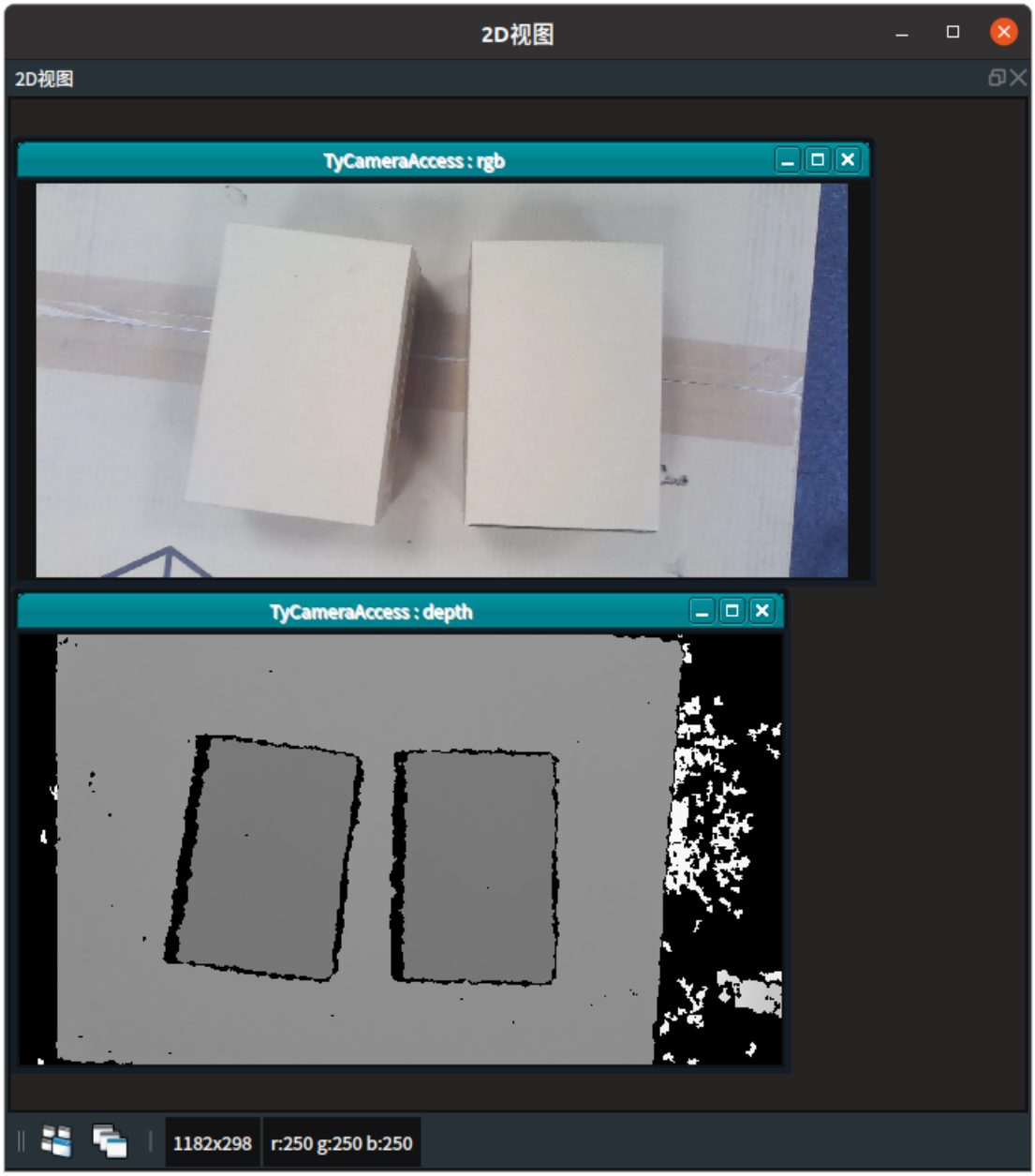
1. 点击 RVS 运行按钮， TyCameraSimResource 资源算子启动成功。
2. 成功后，触发 Trigger 算子。

运行结果

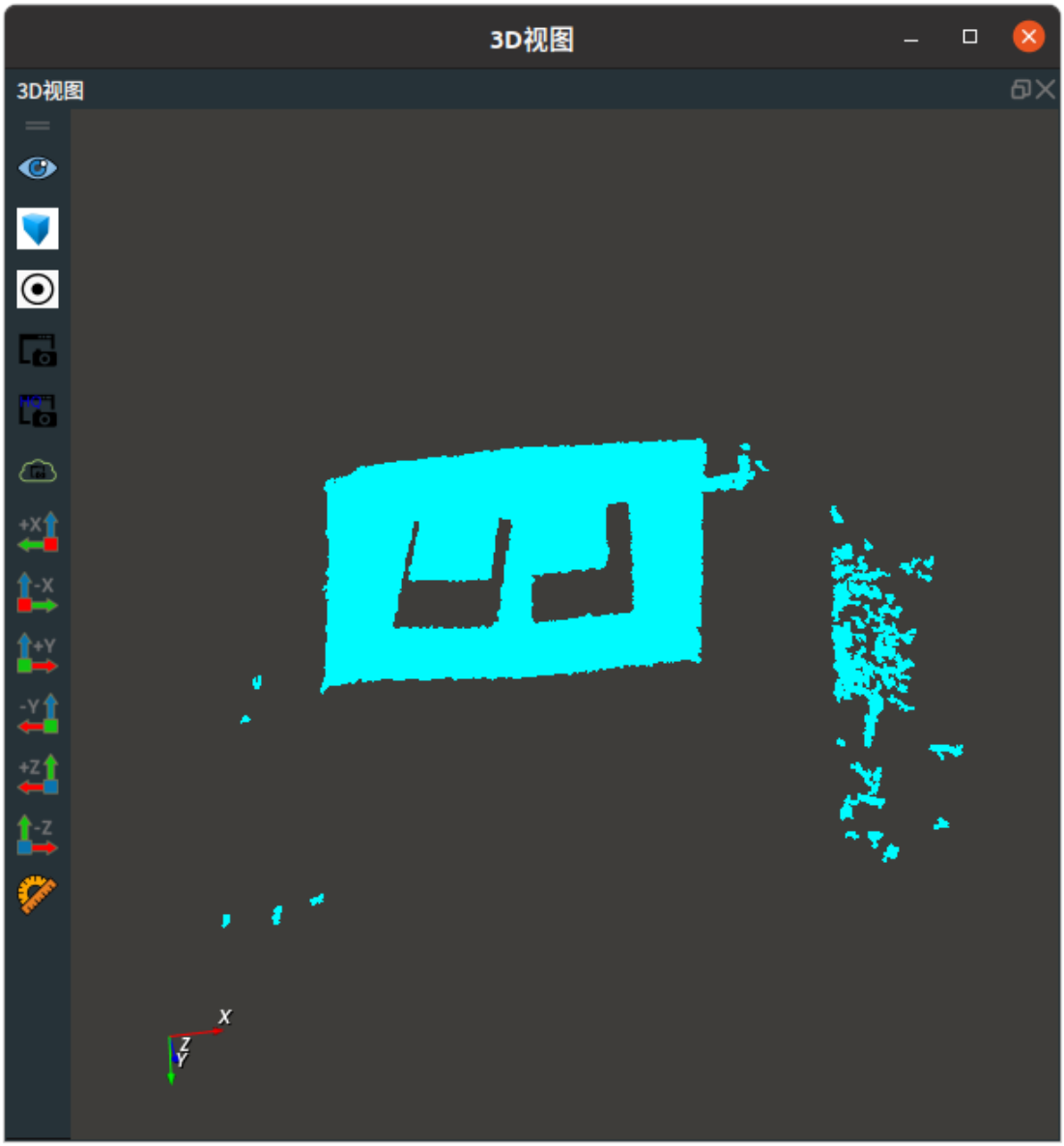
1. 打开 RVS 的运行按钮，TyCameraSimResource 算子会自动触发，并变为蓝色，日志栏会同时打印算子运行说明如下图所示，表示图漾仿真资源算子启动成功。



2. 当触发 Trigger 算子后，TycameraAccess 运行完成后，在 2D 视图中显示彩色图像和深度图像。



3. 在 3D 视图中显示点云。


3D视图
3D视图
+X
-X
+Y
-Y
+Z
-Z
X
Z
Y

JakaRobotsResource Jaka机器人控制资源

JakaRobotsResource 算子属于资源类线程算子，用于在 RVS 的分线程中启动一个 Jaka 机器人控制资源。

通过添加 Jaka 机器人控制资源，使用 RVS 中 RobotMovement 算子（机器人运动控制工具）和 RobotOperator算子（机器人操作工具）对 Jaka 机器人进行运动控制和操作。

算子参数

- **自动启动/auto_start**：用于自动开启资源算子。
  - True：打开 RVS 软件后第一次进入运行状态时自动开启资源线程。
  - False：不自动开启资源线程。
- **启动/start**：用于开启资源算子。
  - True：勾选为True，开启资源线程。
  - False：不启动资源线程。
- **停止/stop**：用于停止资源算子。
  - True：勾选为True，停止资源线程。
  - False：不停止资源线程。
- **重置/reset**：重置该资源。在该资源算子已经运行后，如果重新更改属性参数，需要点击**重置**，然后重新勾选 **启动** 运行。
- **机器人名称/robot_name**：Jaka 机器人名称。默认：JakaRobot。
- **机器人模型文件/robot_file**： Jaka 机器人模型文件名。文件格式： *.rob 。默认值：data/Jaka/MiniCobo/MiniCobo.rob。

  说明：需要提前向图漾技术支持人员申请所需要的机器人数模文件。
- **工具模型文件/tool_file**：夹具吸盘等工具等文件名。
- **最大关节速度/max_joint_velocity**：用于控制机器人 MoveJoints 时的关节最大速度。单位：弧度每秒。默认值：1。
- **最大关节加速度/max_joint_acceleration**：用于控制机器人 MoveJoints 时的关节最大加速度。单位：弧度每平方秒。默认值：1。
- **最大线性速度/max_linear_velocity**：用于控制机器人线性运动时的最大线速度。单位：米每秒。默认值：1。
- **最大线性加速度/max_linear_acceleration**：用于控制机器人线性运动时的线速最大加速度。单位：米每平方秒。默认值：1。
- **机器人/robot**：设置 Jaka 机器人数模在 3D 视图中的可视化属性。
  -  打开 Jaka 机器人仿真模型可视化。
  -  关闭 Jaka 机器人仿真模型可视化。
- **机器人更新/robot_update**：设置机器人是否实时加载更新。默认不勾选，该参数主要用于生成机器人模型时调参查看效果。
- **机器人IP/robot_ip**：填写 Jaka 机器人的通讯 IP。 默认值：192.168.1.236。
- **机器人端口/arm_port**：填写 Jaka 机器人的通讯 IP 的端口号。 默认值：1。

控制信号输入输出

说明：Resource 资源类算子的控制信号端口无法触发使用。

功能演示

与 EliteRobotsResource 资源算子功能演示类似，请参考该算子功能演示模块进行操作。

robot

ScaleJoint 关节值比例转换

ScaleJointNode 算子用于转换机器人关节值的角度和弧度单位。

转换类型	功能
deg2rad	从角度转换到弧度。
rad2deg	从弧度转换到角度。

rad2deg

将 ScaleJoint 算子的 **转换类型** 属性选择 rad2deg ，将机器人关节值从弧度转换到角度。

数据信号输入输出

输入：

- **joint**：
  - 数据类型：JointArray
  - 输入内容：转换前的机器人关节值
- **joint_list**：
  - 数据类型：JointArrayList
  - 输入内容：转换前的机器人关节值列表

输出：

- **joint**：
  - 数据类型：JointArray
  - 输出内容：转换后的机器人关节值
- **joint_list**：
  - 数据类型：JointArrayList
  - 输出内容：转换后的机器人关节值列表

功能演示

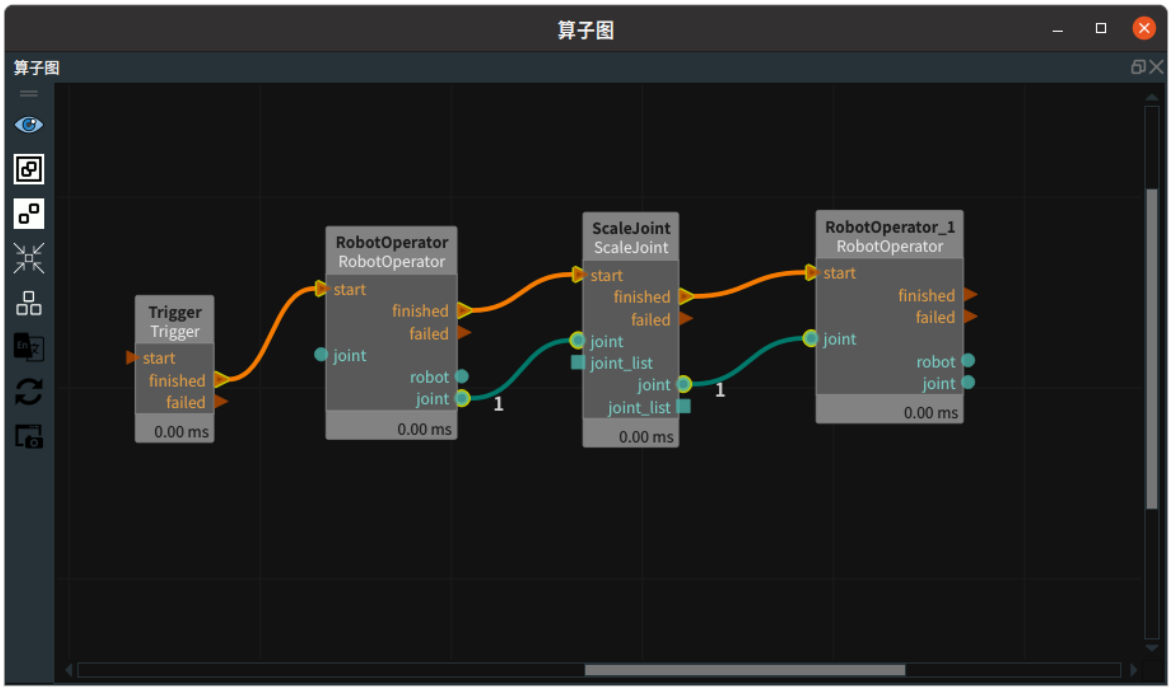
使用 ScaleJoint 算子中 rad2deg ，对机器人 joint 从弧度到角度进行转换。

步骤1：算子准备

添加 Trigger 、RobotOperater（2 个） 、ScaleJoint 算子至算子图。

步骤2：设置算子参数

1. 设置 RobotOperater 算子参数：
   - 类型 → EmitJoint
   - 关节 → 0 0 1.5708 0 1.5708 0
2. 设置 ScaleJoint 算子参数：转换类型 → rad2deg
3. 设置 RobotOperater_1 算子参数：类型 → EmitJoint

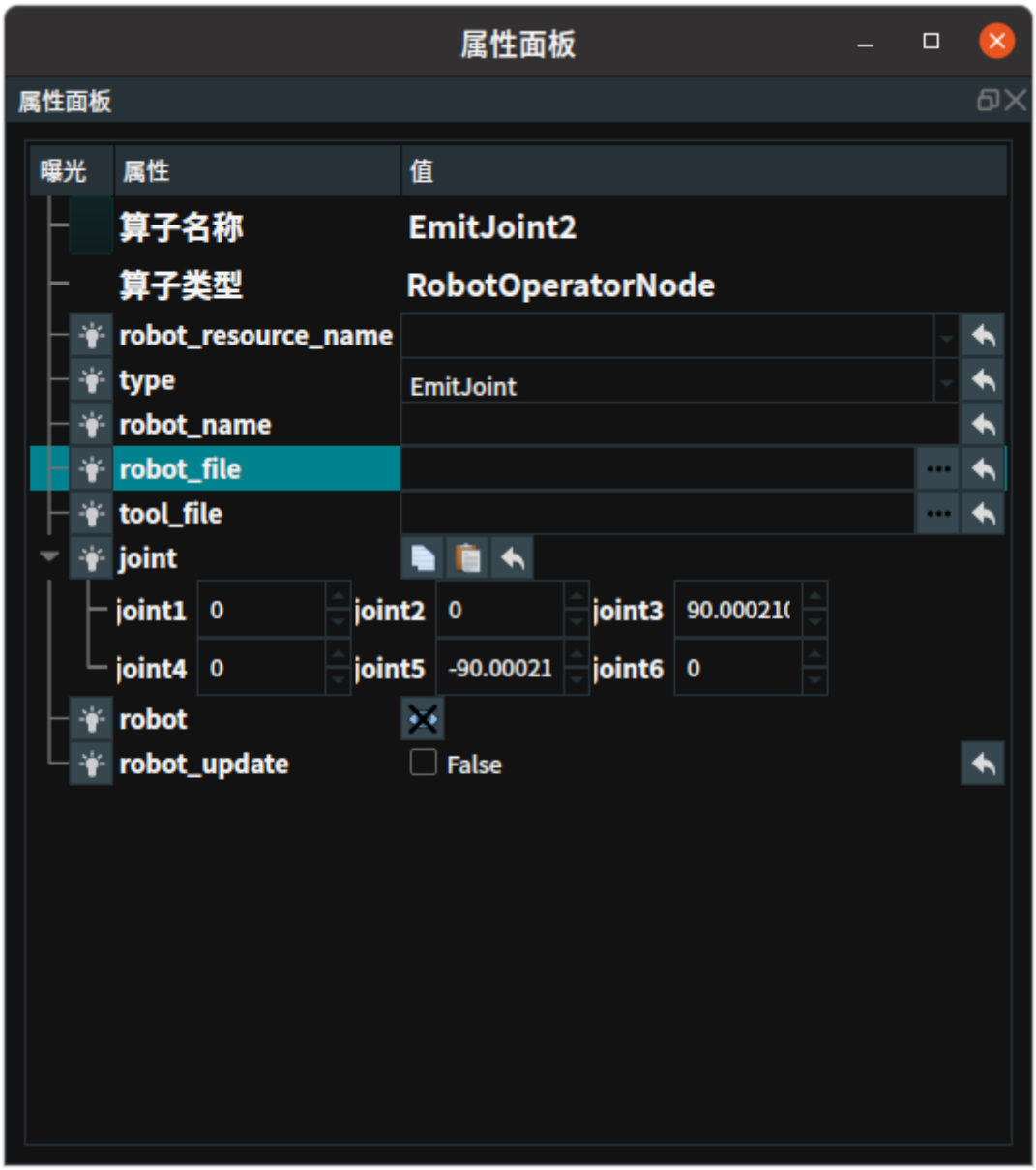


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，RobotOperater_1 算子的属性面板，弧度值已经转换成角度。



deg2rad

将 ScaleJoint 算子的 **转换类型** 属性选择 deg2rad ，从角度转换到弧度。

数据信号输入输出

输入：

- **joint**：
  - 数据类型：JointArray
  - 输入内容：转换前的机器人关节值
- **joint_list**：
  - 数据类型：JointArrayList
  - 输入内容：转换前的机器人关节值列表

输出：

- **joint**：
  - 数据类型：JointArray

    - 输出内容：转换后的机器人关节值
- `joint_list`：
    - 数据类型：JointArrayList
    - 输出内容：转换后的机器人关节值列表

功能演示

本节将使用 ScaleJoint 算子中 deg2rad ，将机器人 joint 从角度到弧度进行转换。这与 ScaleJoint 算子中 rad2deg 属性的从弧度转换到角度的方法相同，请参照该章节的功能演示。

RobotMovement 机器人运动控制工具

RobotMovement 算子为机器人运动控制工具，用于控制机器人移动。

type	功能
MoveJoint	通过 joint 移动机器人。
MoveJointList	通过 joint_list 移动机器人。
MoveTCP	通过 TCP 移动机器人。
StopMotion	停止机器人
SetRobotIO	设置机器人 IO 端口信号。
WaitRobotIO	在设置时间内，等待机器人 IO 端口信号进行判断输出。

MoveJoint

将 RobotMovement 算子的 **类型** 属性选择 MoveJoint ，通过 joint 移动机器人。

算子参数

- **机器人资源名称/robot_resource_name**：机器人资源名。选择在Resource Group 中添加的机器人资源。
- **关节/joint**：机器人移动的关节值。
- **速度/velocity**：机器人移动的速度。默认值：1。单位：m/s。
- **加速/acceleration**：机器人移动的加速度。默认值：1。单位：m/s^2。
- **线性运动/move_linear**：机器人移动方式。
  - True：机器人直线移动。
  - False：机器人正常移动。

数据信号输入输出

输入：

- **joint**
  - 数据类型：JointArray
  - 输出内容：机器人关节弧度值数据

输出：

- **joint**
  - 数据类型：JointArray
  - 输入内容：机器人关节弧度值数据 joint 数据

功能演示

使用 RobotMovement 中 MoveJoint ，通过 joint 的方式将机器人轴1 移动至 90°。

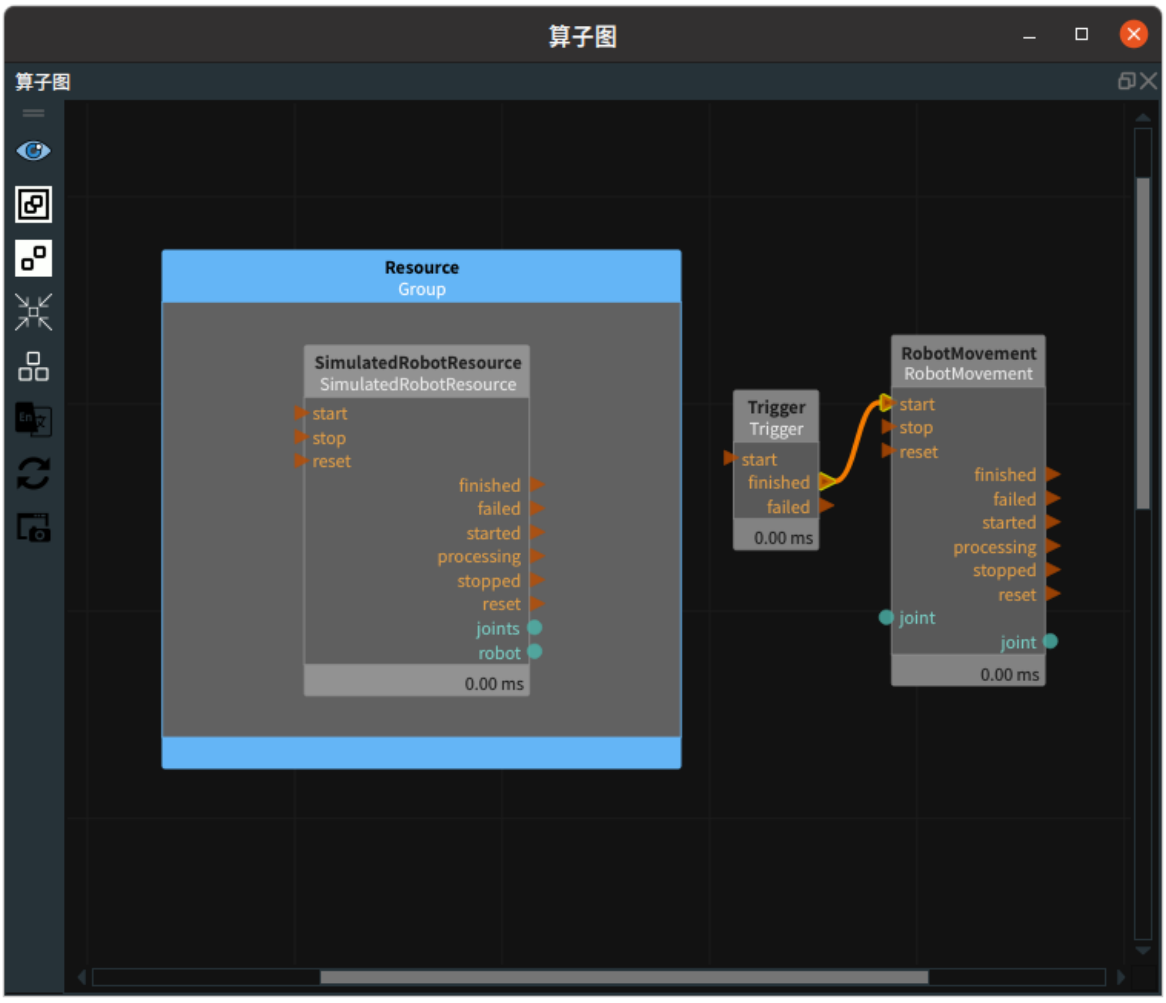
步骤1：算子准备

1. 添加 SimulatedRobotResource 资源算子至 Resource Group 。
2. 添加 Trigger 、RobotMovement 算子至算子图。

步骤2：设置算子参数

1. 设置 SimulatedRobotResource 算子参数：
    - 自动启动 → True
    - 机器人模型文件 → ••• → 选择 robot 文件名 （*example_data/UR5/UR5.rob* ）
    - 机器人 → 可视
2. 设置 RobotMovement 算子参数：
    - 机器人资源名称 → SimulatedRobotResource
    - 类型 → MoveJoint
    - 关节 → 1.5708 0 0 0 0 0

步骤3：连接算子

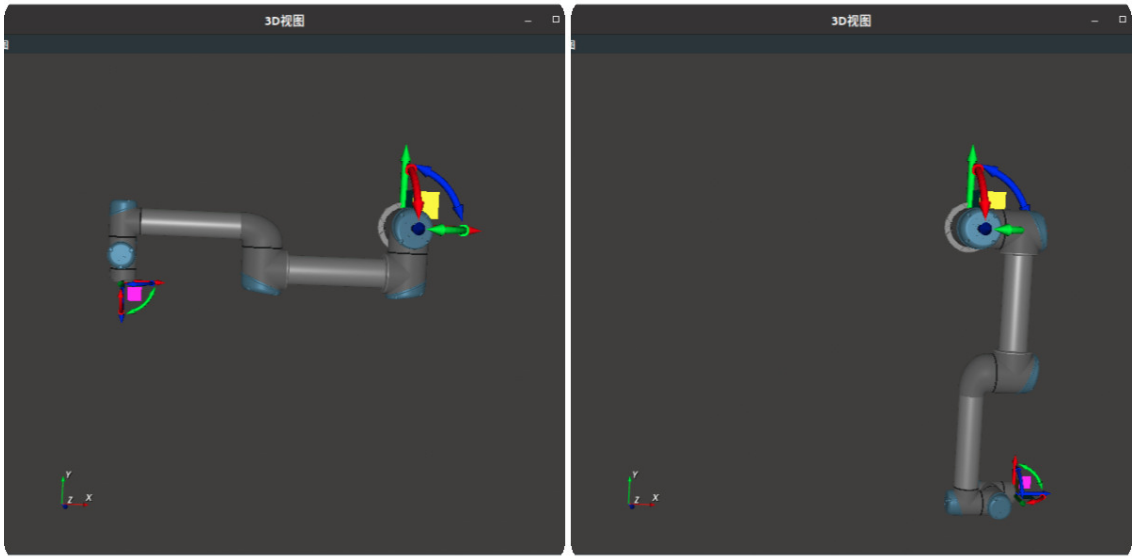


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

1. 结果如下图所示，左侧为 SimulatedRobotResource 算子 robot 可视化初始状态。右侧为 joint1 旋转 1.5708 后的结果。
2. 鼠标左键双击 3D 视图中的机器人，弹出机器人面板，将机器人基坐标和机器人 TCP 勾选 True 。



3. 在机器人面板中查看此时机器人关节轴 1 为 90° 。

MoveJointList

将 RobotMovement 算子的 `类型` 属性选择 MoveJointList ，通过 joint_list 的方式移动机器人。

算子参数

- `机器人资源名称/robot_resource_name`：机器人资源名。选择在Resource Group 中添加的机器人资源。
- `速度/velocity`：机器人移动的速度。默认值：1。单位：m/s。
- `加速/acceleration`：机器人移动的加速度。默认值：2。单位：m/s²。
- `线性运行/move_linear`：机器人移动方式。
  - True：机器人直线移动。
  - False：机器人正常移动。

数据信号输入输出

输入：

- `joint_list`
  - 数据类型：JointArrayList
  - 输出内容：机器人关节弧度值列表数据

功能演示

使用 RobotMovement 中 MoveJointList ，通过 joint_list 的方式移动机器人。

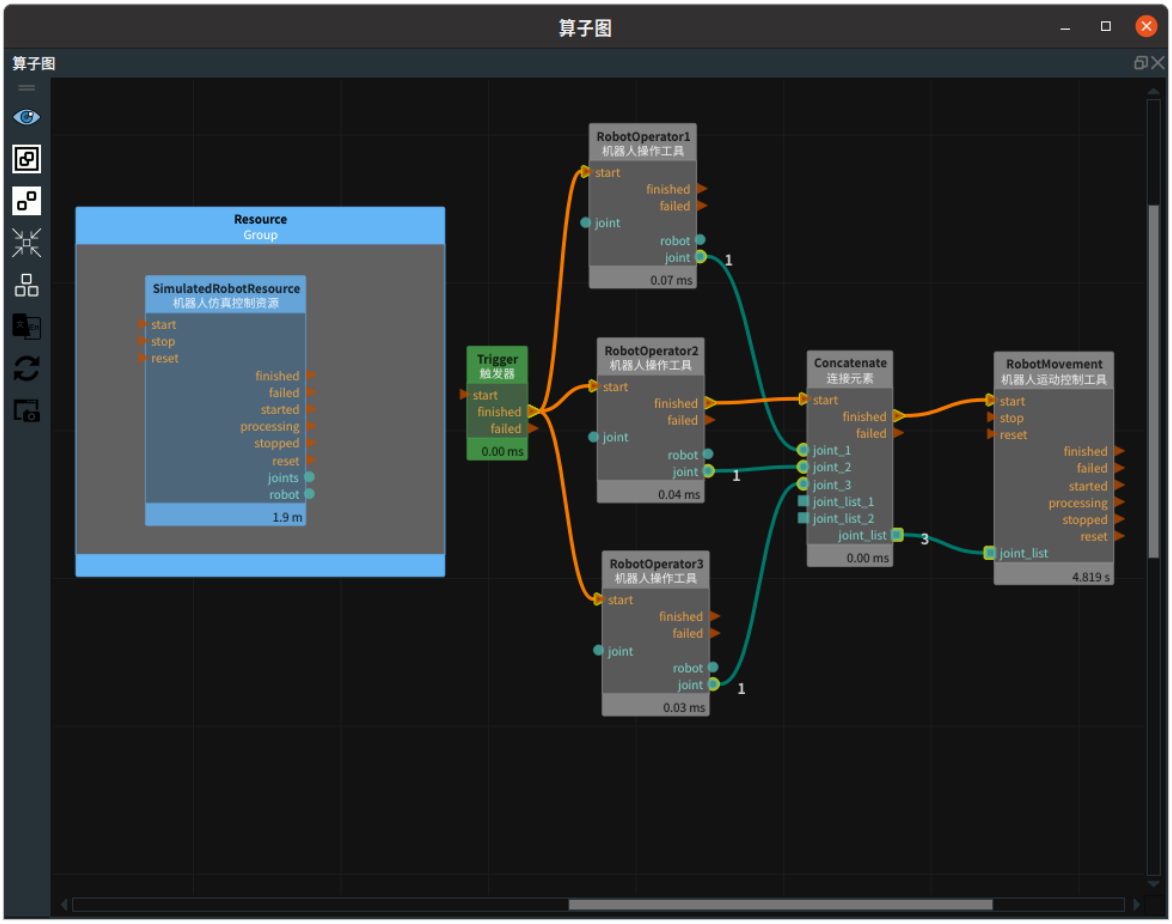
步骤1：算子准备

1. 添加 SimulatedRobotResource 资源算子至 Resource Group 。
2. 添加 Trigger 、RobotOperator (3 个）、Concatenate、RobotMovement 算子至算子图。

步骤2：设置算子参数

1. 设置 SimulatedRobotResource 算子参数：
   - 机器人模型文件 → ••• → 选择 robot 文件名 （ *example_data/UR5/UR5.rob* ）
   - 机器人 →  可视
2. 设置 RobotOperator 算子参数：
   - 算子名称 → RobotOperator1
   - 类型 → EmitJoint
   - 关节 → 1.5708 0 0 0 0 0
3. 设置 RobotOperator_1 算子参数：
   - 算子名称 → RobotOperator2
   - 类型 → EmitJoint
   - 关节 → 1.5708 -1.5708 0 0 0 0
4. 设置 RobotOperator_2 算子参数：
   - 算子名称 → RobotOperator3
   - 类型 → EmitJoint
   - 关节 → 1.5708 -1.5708 1.5708 0 0 0
5. 设置 RobotMovement 算子参数：
   - 机器人资源名称 → SimulatedRobotResource
   - 类型 → MoveJointList

步骤3：连接算子

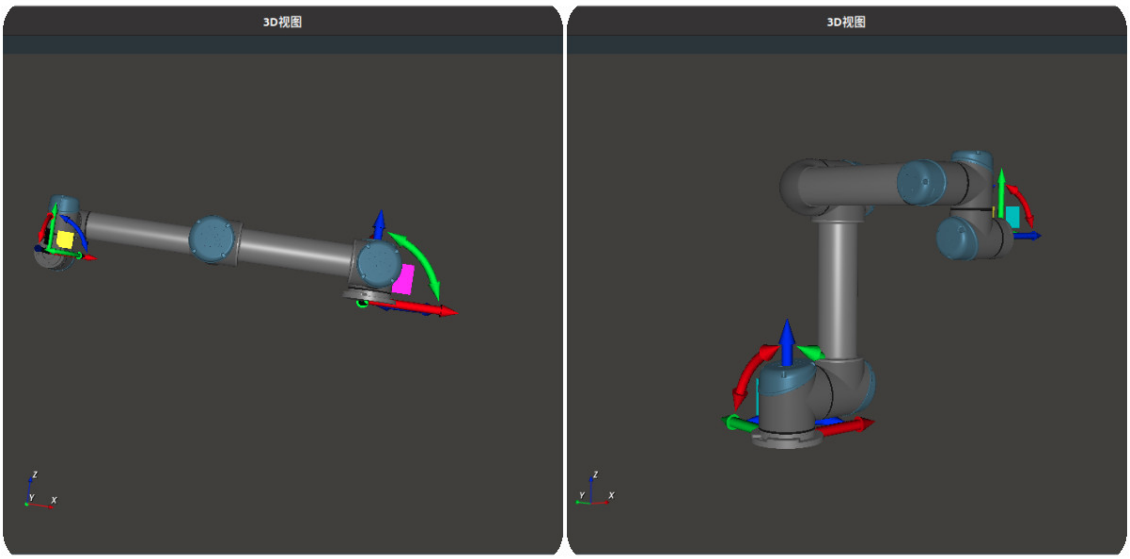


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

1. 在机器人运动过程中，会根据 Concatenate 算子的输出结果 joints_list 的索引分别移动。
2. 最终结果如下图所示，左侧为 SimulatedRobotResource 算子 robot 可视化初始状态。右侧为移动最后一组 RobotOperator3 机器人关节弧度值结果。
3. 鼠标左键双击 3D 视图中的机器人，弹出机器人面板，将机器人基坐标和机器人 TCP 勾选 True 。



4. 在机器人面板中查看此时机器人关节轴 1： 90°，轴2：-90°，轴3：90° 。该值为最后 RobotOperator3 算子中生成的关节值弧度值。

MoveTCP

将 RobotMovement 算子的 `类型` 属性选择 MoveTCP ，通过 TCP 移动机器人。

算子参数

- `机器人资源名称/robot_resource_name`：机器人资源名。选择在 Resource Group 中添加的机器人资源。
- `tcp`：机器人末端坐标。
- `速度/velocity`：机器人移动的速度。默认值：1。单位：m/s。
- `加速/acceleration`：机器人移动的加速度。默认值：1。单位：m/s^2。
- `线性运动/move_linear`：机器人移动方式。
  - True：机器人直线移动。
  - False：机器人正常移动。

数据信号输入输出

输入：

- `goal_pose`
  - 数据类型：Pose
  - 输出内容：pose 数据
- `ref_joint`
  - 数据类型：JointArray
  - 输出内容：机器人关节弧度值数据

输出：

- `ik_solution_joint`
  - 数据类型：JointArray
  - 输出内容：机器人关节弧度值数据

功能演示

使用 RobotMovement 中 MoveTCP ，通过 TCP 的方式移动机器人。

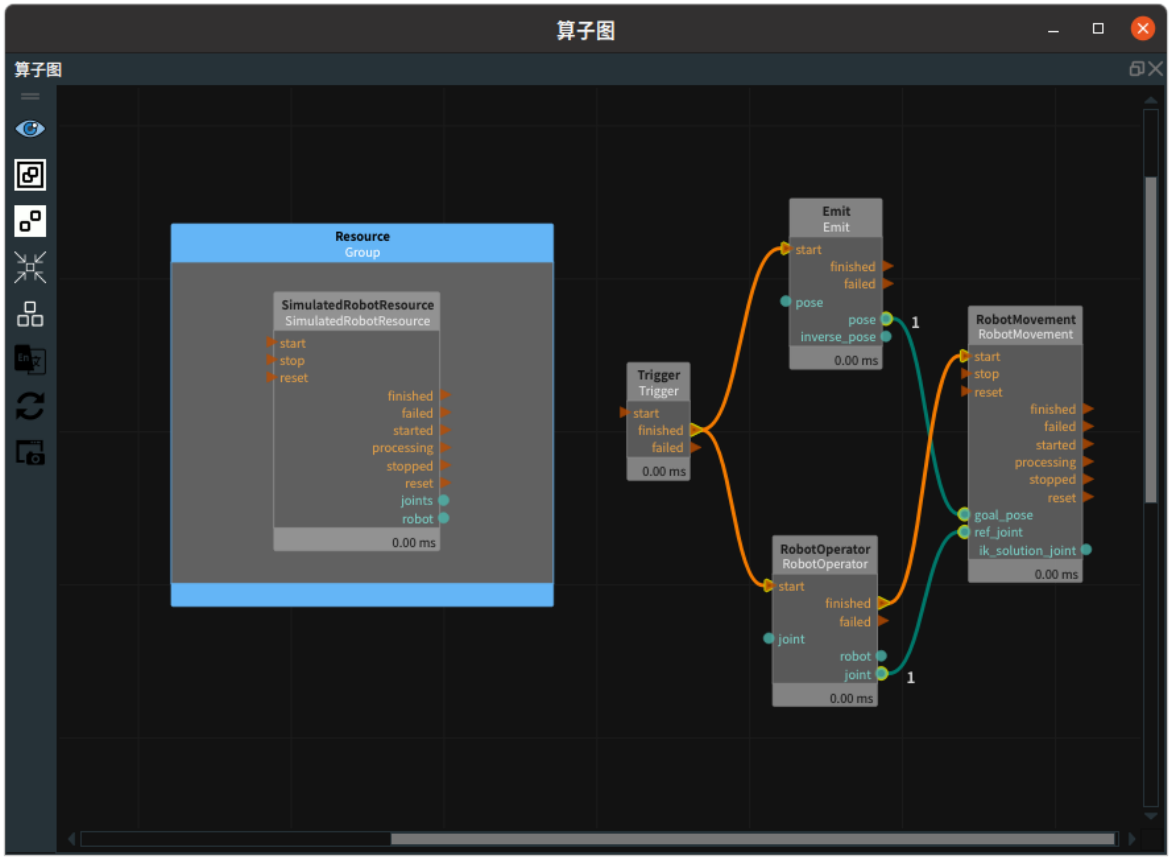
步骤1：算子准备

1. 添加 SimulatedRobotResource 资源算子至 Resource Group
2. 添加 Trigger 、Emit、RobotOperator、RobotMovement 算子至算子图。

步骤2：设置算子参数

1. 设置 SimulatedRobotResource 算子参数：
   - 机器人模型文件 → ••• → 选择 robot 文件名 （ *example_data/UR5/UR5.rob* ）
2. 设置 RobotMovement 算子参数：
   - 机器人资源名称 → SimulatedRobotResource
   - 类型 → MoveTCP
3. 设置 Emit 算子参数：
   - 类型 → Pose
   - 坐标 → 0.10158 -0.5167 0.39675 -3.11585 0.02054 3.00946
4. 设置 RobotOperator 算子参数：
   - 机器人资源名称 → SimulatedRobotResource
   - 类型 → EmitJoint
   - 关节 → 1.57 -1.57 0 0 0 0

步骤3：连接算子

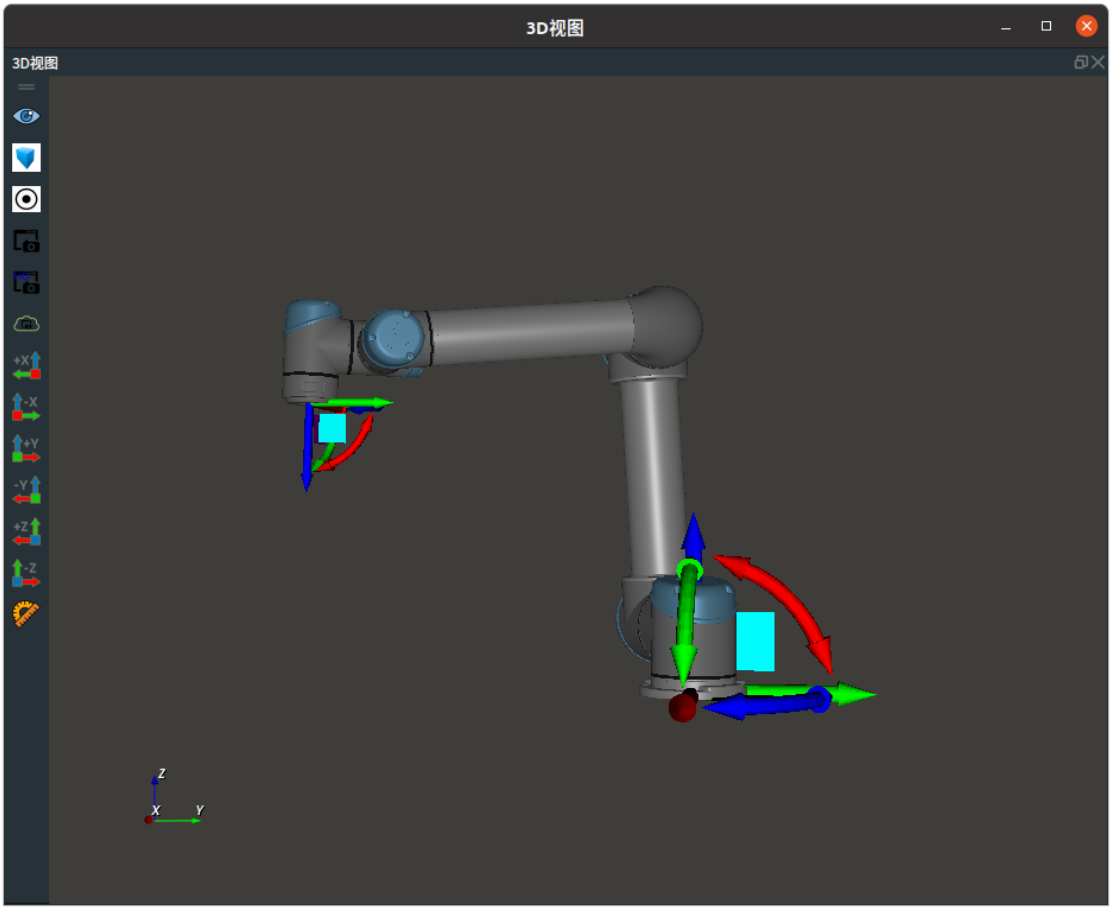


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

1. 如下图所示，在 3D 视图中显示当前加载机器人 RobotMovement 后的结果 。
2. 鼠标左键双击3D视图中的机器人，弹出机器人面板，将机器人基坐标和机器人 TCP 设为 True 。



3. SimulatedRobot 面板中显示机器人当前机器人 TCP 值 和 关节轴值。

StopMotion

将 RobotMovement 算子的 **类型** 属性选择 StopMotion ，用于停止机器人。

算子参数

机器人资源名称/robot_resource_name：机器人资源名。选择在 Resource Group 中添加的机器人资源。

功能演示

使用 RobotMovement 中 StopMotion ，控制机器人停止。

步骤1：算子准备

1. 添加 SimulatedRobotResource 资源算子至 Resource Group 。
2. 添加 Trigger 、RobotMovement 算子至算子图。

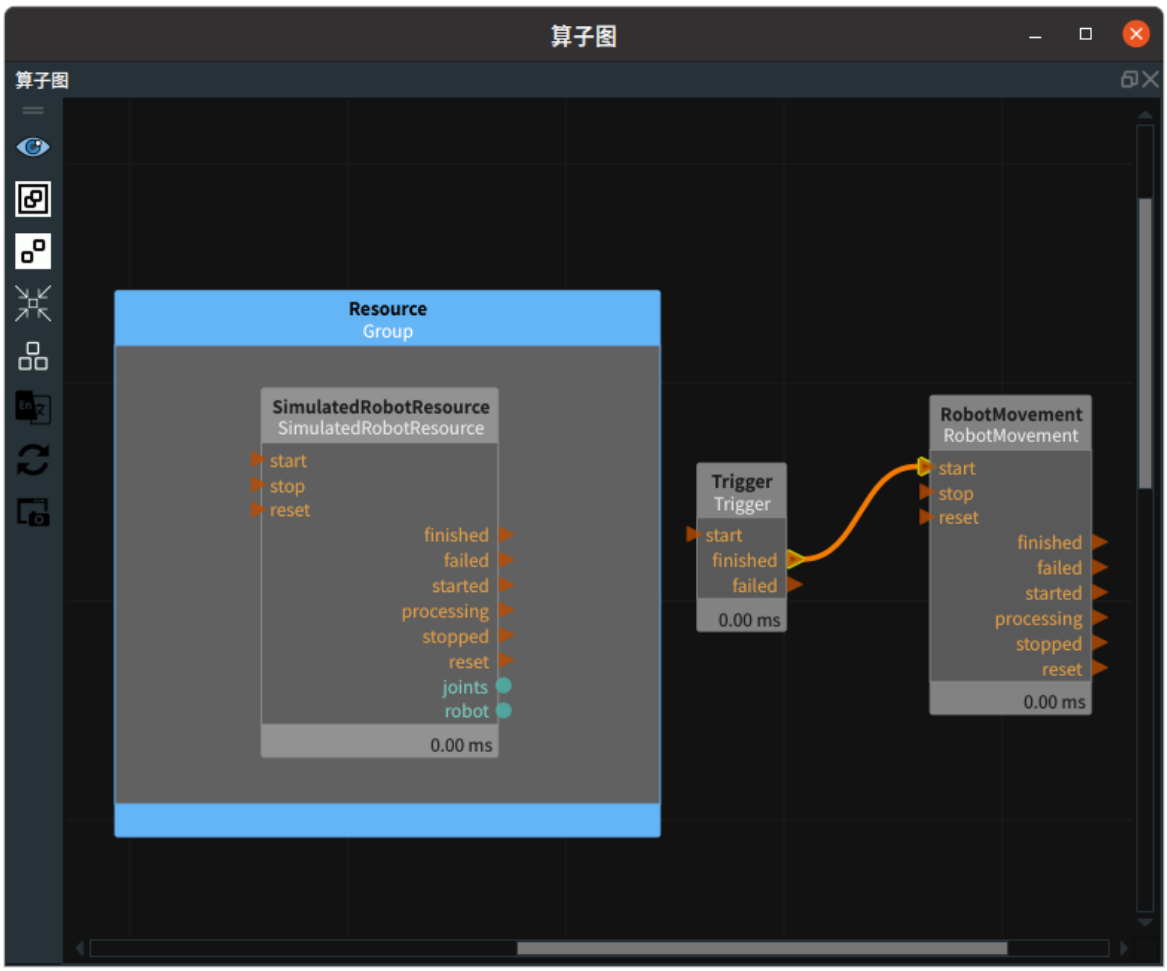
步骤2：设置算子参数

1. 设置 SimulatedRobotResource 算子参数：
    - 机器人模型文件 → ●●● → 选择 robot 文件名 （ *example_data/UR5/UR5.rob* ）

- 机器人 → 可视

2. 设置 RobotMovement 算子参数：
   - 机器人资源名称 → SimulatedRobotResource
   - 类型 → StopMotion

步骤3：连接算子

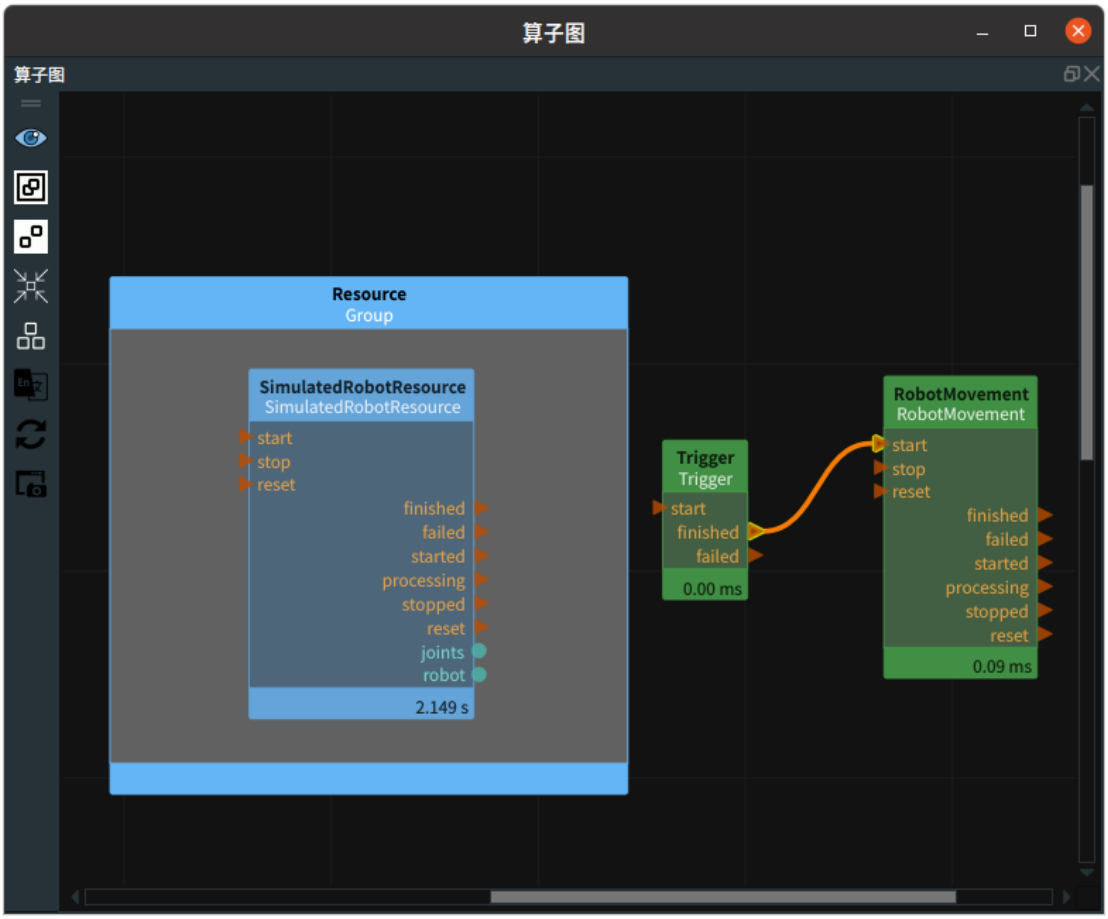


步骤4：运行

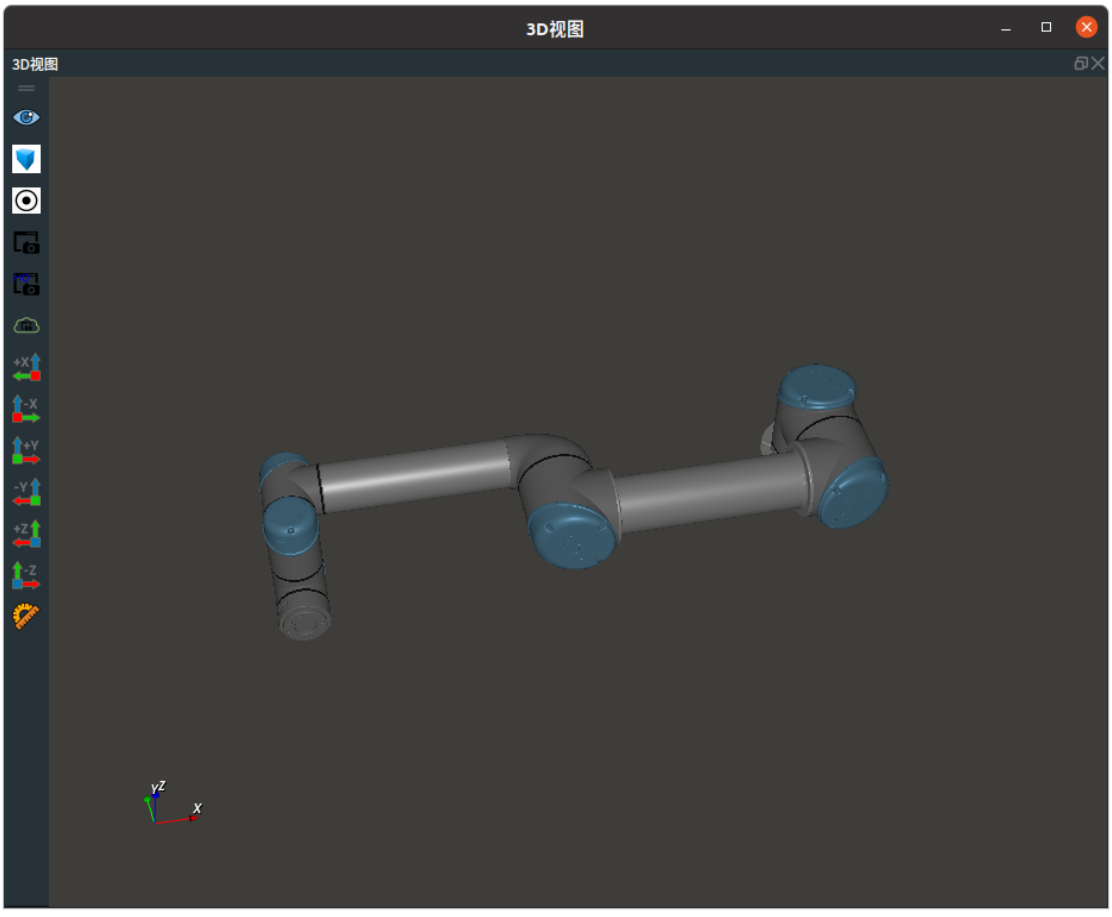
点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

1. 如下图所示，触发该算子后，机器人停止运动。



2. 如下图所示，3D 视图中机器人处于停止状态。



RobotOperator 机器人操作工具

RobotOperator 算子为机器人操作工具，用于对机器人的操作

type	功能
EmitJoint	生成机器人的关节值joint。
EmitActualJoint	获得当前机器人的关节值joint。
EmitActualTCP	获得当前机器人的TCP。
EmitToolPose	获得当前机器人工具的pose。
ComputeDK	计算机器人正运动学。
ComputeIK	计算机器人逆运动学。
ComputeJoinPose	计算各个旋转轴所在的位置。
VisualizeJoin	/
AttachObject	用于给机器人添加工具。
DetachObject	用于删除机器人工具。

EmitJoint

将 RobotOperator 算子的 **类型** 属性选择 EmitJoint ，用于生成机器人的关节值 joint。

算子参数

- **机器人资源名称/robot_resource_name**：机器人资源名。选择在Resource Group 中添加的机器人资源。
- **机器人名称/robot_name**：机器人名称。
- **机器人模型文件/robot_file**：机器人文件名。
- **工具模型文件/tool_file**：工具文件名。
- **关节/joint**：机器人关节弧度值。
- **机器人/robot**：设置机器人在 3D 视图中的可视化属性。
  - 打开机器人可视化。
  - 关闭机器人可视化。
- **机器人更新/robot_updata**：设置机器人更新。
  - True：机器人更新操作。
  - False：机器人未更新操作。

数据信号输入输出

输入：

- **joint**：
  - 数据类型：JointArray

- 输入内容：机器人关节弧度值数据

输出：

- `robot`：
  - 数据类型：Robot
  - 输出内容：机器人数据
- `joint`：
  - 数据类型：JointArray
  - 输出内容：机器人关节弧度值数据

功能演示

使用 RobotOperator 算子中 EmitJoint ，生成机器人的关节弧度值 joint。

步骤1：算子准备

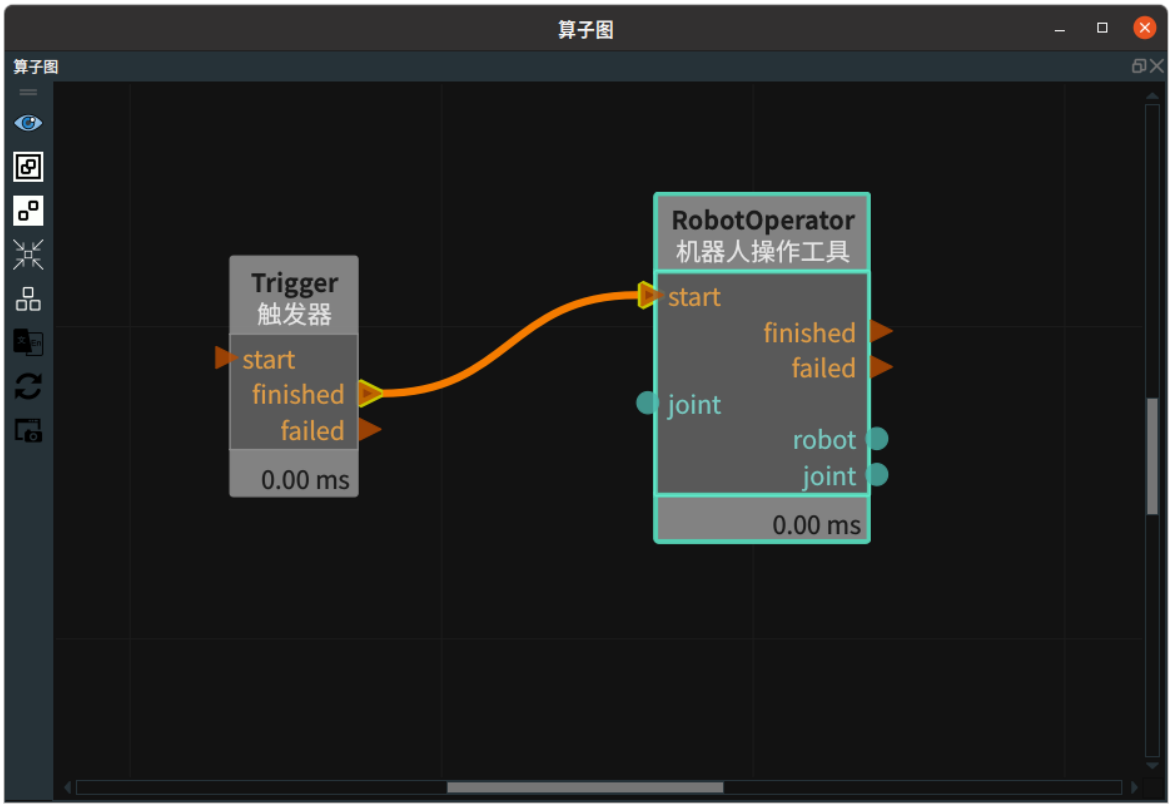
添加 Trigger 、RobotOperator 算子至算子图。

步骤2：设置算子参数

设置 RobotOperator 算子参数：

- 类型 → EmitJoint
- 机器人名称 → UniversalRobot
- 机器人模型文件夹 → ••• → 选择 robot 文件名 (*example_data/UR5/UR5.rob*)
- 关节 → 1.5708 0 0 0 0 0
- 机器人 → 可视

步骤3：连接算子

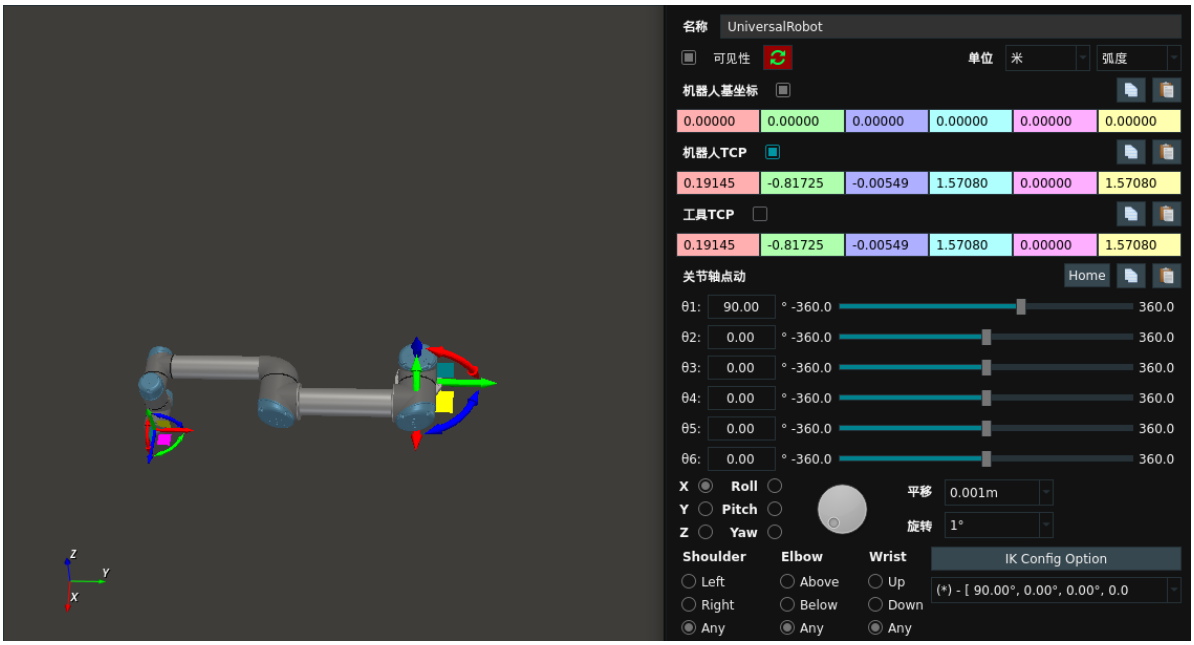


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，在 3D 视图中显示当前加载的机器人，鼠标左键双击 3D 视图中的机器人，弹出机器人面板，将机器人基坐标和机器人 TCP 设为 True 。机器人当前关节值为“ 90° 0° 0° 0° 0° 0° ”，与 RobotOperator 算子 joint 参数 “ 1.5708 0 0 0 0 0 ” 一致。



EmitActualJoint

将 RobotOperator 算子的 **类型** 属性选择 EmitActualJoint ，用于获得当前机器人的关节弧度值。

算子参数

- **机器人资源名称/robot_resource_name**：机器人资源名。选择在Resource Group 中添加的机器人资源。

数据信号输出

输出：

- **joint**：
  - 数据类型：JointArray
  - 输出内容：机器人关节弧度值数据

功能演示

使用 RobotOperator 算子中 EmitActualJoint ，获得当前机器人的关节弧度值。

步骤1：算子准备

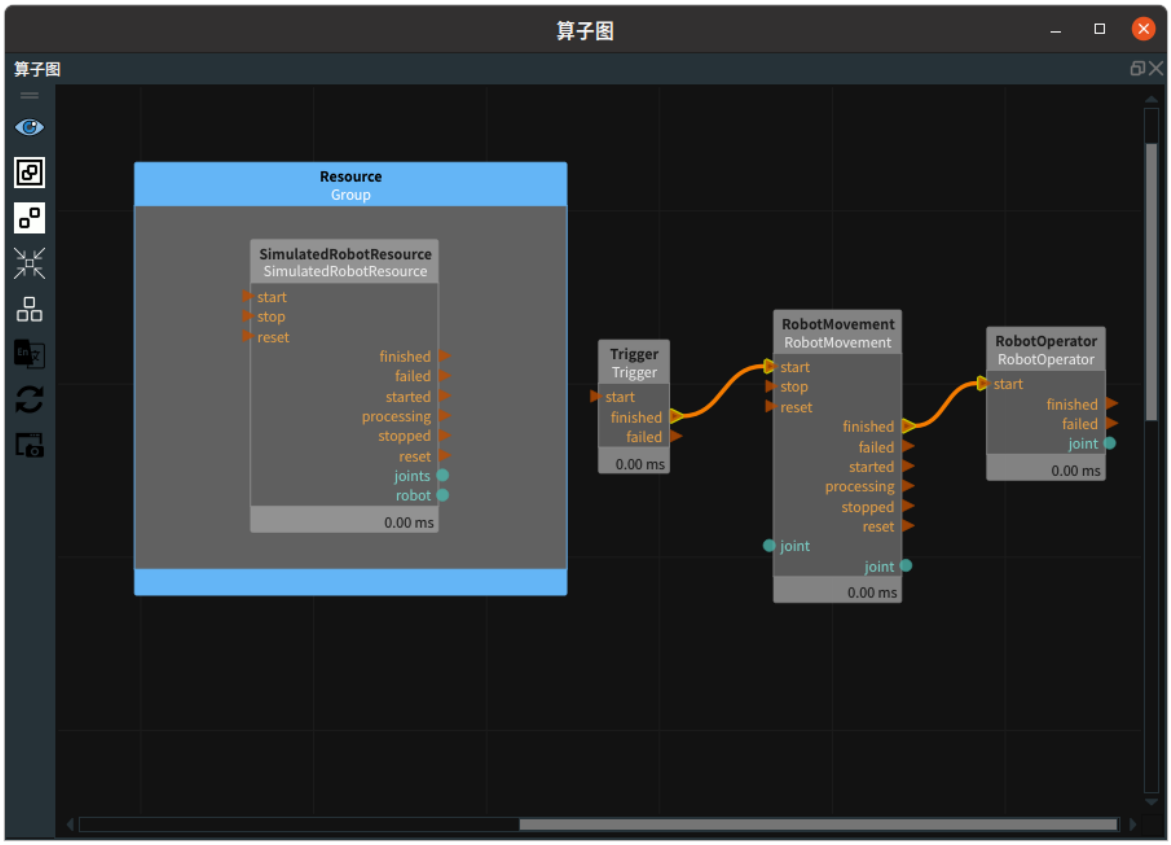
1. 添加 SimulatedRobotResource 资源算子至 Resource Group 。
2. 添加 Trigger 、RobotMovement 、RobotOperator 算子至算子图。

步骤2：设置算子参数

1. 设置 SimulatedRobotResource 算子参数：
   - 机器人模型文件 → ••• → 选择 robot 文件名 （ *example_data/UR5/UR5.rob* ）
   - 机器人 → 可视

2. 设置 RobotMovement 算子参数：
    - 机器人资源名称 → SimulatedRobotResource
    - 类型 → MoveJoint
    - 关节 → 1.5708 0 0 0 0 0
3. 设置 RobotOperator 算子参数：
    - 机器人资源名称 → SimulatedRobotResource
    - 类型 → EmitActualJoint

步骤3：连接算子

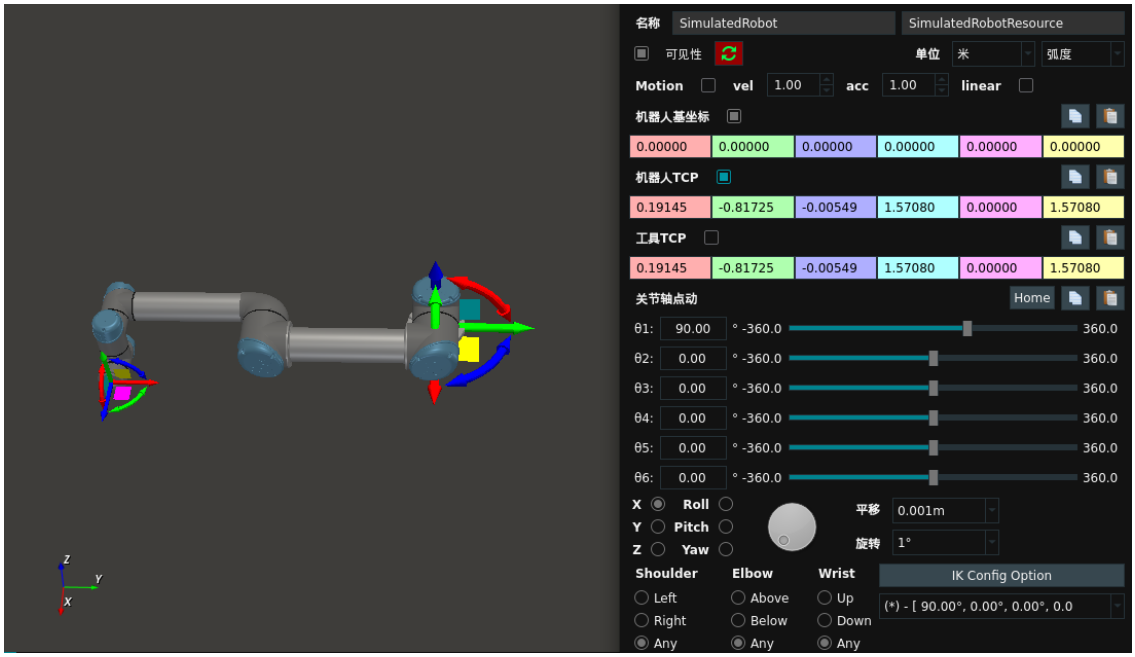


步骤4：运行

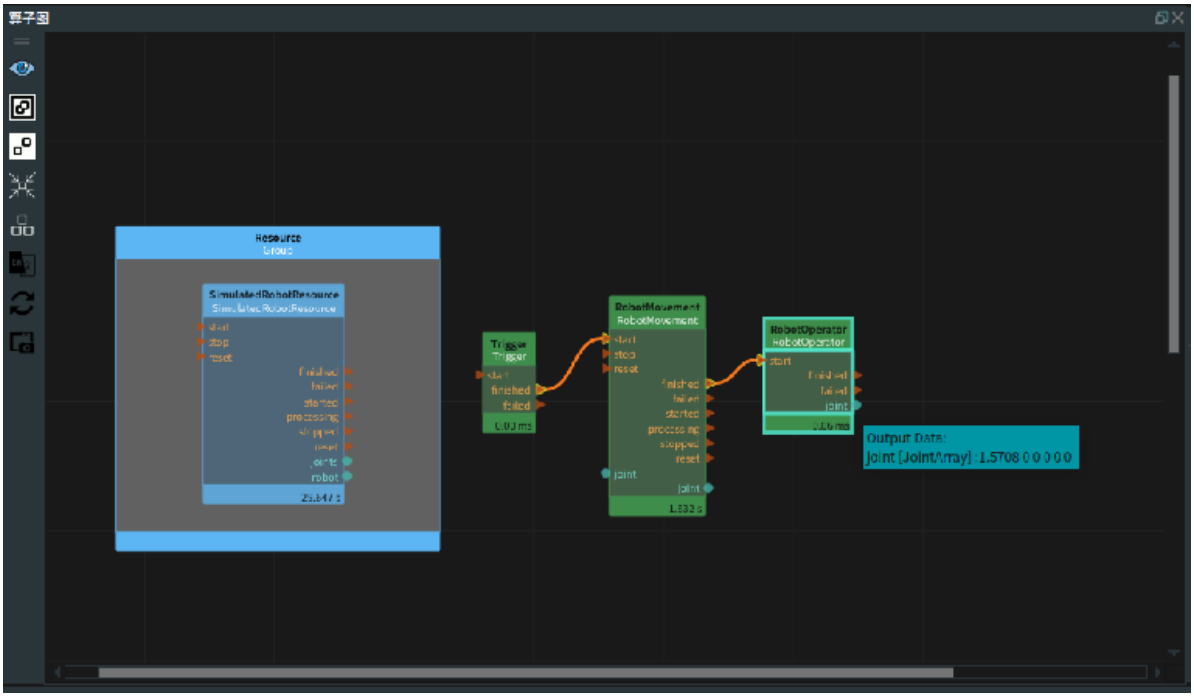
点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

1. 如下图所示，在 3D 视图中显示当前加载的机器人 ，鼠标左键双击 3D 视图中的机器人，弹出机器人面板，将机器人基坐标和机器人 TCP 设为 True 。仿真机器人按照 MoveJoint 方式移动关节值 “ 1.5708 0 0 0 0 0 ”。RobotOperator算子可以获得当前机器人的关节值为 “ 1.5708 0 0 0 0 0 ”。



2.如下图所示，在 RobotOperator 算子的右侧输出端口显示当前机器人的关节值为 “ 1.5708 0 0 0 0 0 ”。



EmitActualTCP

将 RobotOperator 算子的 **类型** 属性选择 EmitActualTCP ，用于获得当前机器人的 TCP 值。

算子参数

- **机器人资源名称/robot_resource_name**：机器人资源名。选择在Resource Group 中添加的机器人资源。
- **机器人TCP/tcp**：设置机器人 TCP 可视化属性。
  - 打开 TCP 可视化。
  - 关闭 TCP 可视化。
  - 设置 TCP 的尺寸。取值范围：[0.001,10]。默认值：0.1。

数据信号输出

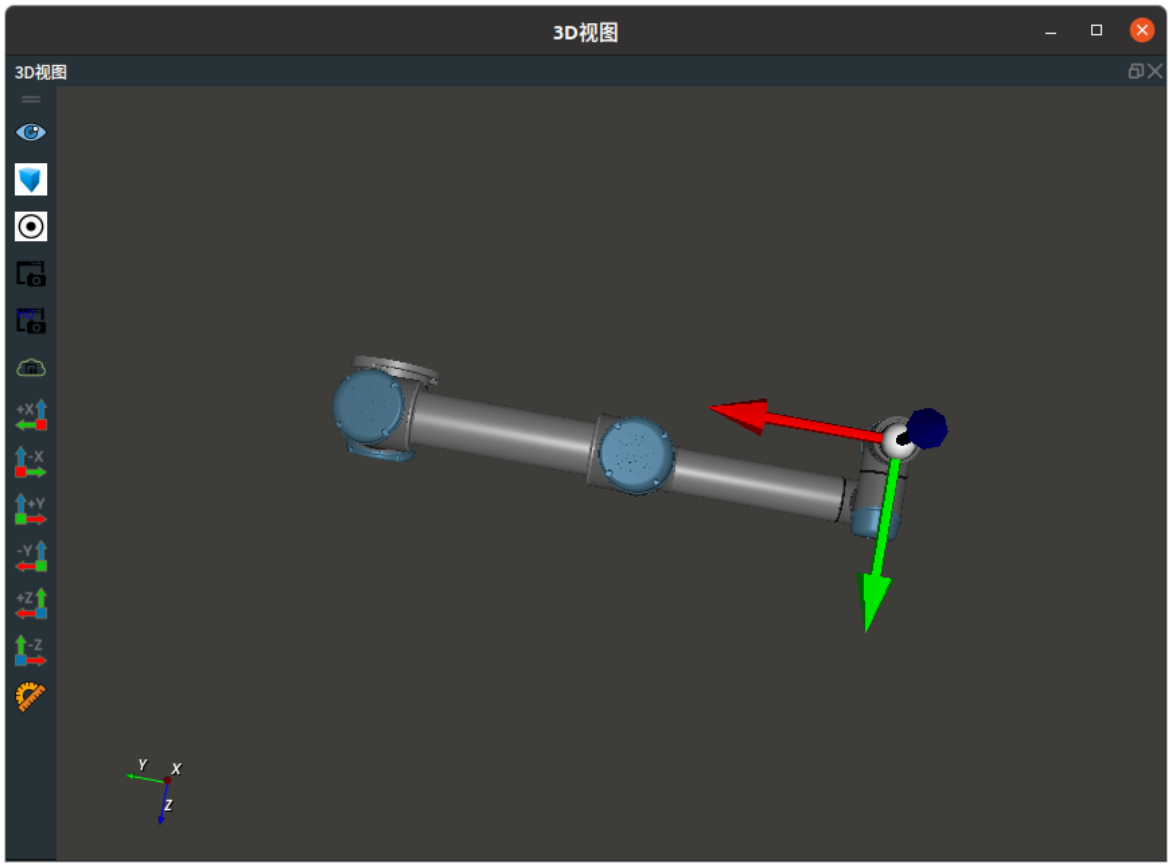
输出：

- `tcp`：
  - 数据类型：TCP
  - 输出内容：机器人 tcp 值数据

功能演示

本节将使用 RobotOperator 算子中 EmitActualTCP ，获得当前机器人的 TCP 值。这与 RobotOperator 算子中 EmitActualJoint 属性获得当前机器人的关节弧度值的方法相同，请参照该章节的功能演示。

打开 TCP 可视化结果，最终在 3D 视图中显示如下。



EmitToolPose

将 RobotOperator 算子的 `类型` 属性选择 EmitToolPose ，用于获得当前机器人工具的 pose 。

算子参数

- `机器人资源名称/robot_resource_name`：机器人资源名。选择在Resource Group 中添加的机器人资源。
- `工具坐标补偿/tool_offset`：当前机器人工具在机器人法兰坐标系下的 pose 。
- `工具坐标/tool_pose`：当前机器人工具的 pose 。
- `工具坐标补偿/tool_offset`：设置 tool_offset 在 3D 视图中的可视化属性。
  - 打开 tool_offset 可视化。
  - 关闭 tool_offset 可视化。
- `工具坐标/tool_pose`：设置 tool_pose 在 3D 视图中的可视化属性。

-  打开 tool_pose 可视化。
-  关闭 tool_pose 可视化。

数据信号输出

输出：

- `tool_offset`：
  - 数据类型：Pose
  - 输出内容：机器人工具在机器人法兰坐标系下的 pose
- `tool_pose`：
  - 数据类型：Pose
  - 输出内容：机器人工具的 pose

功能演示

将 RobotOperator 算子的 type 属性选择 EmitToolPose ，获得当前机器人工具的 pose 和 机器人工具在机器人法兰坐标系下的 pose 。

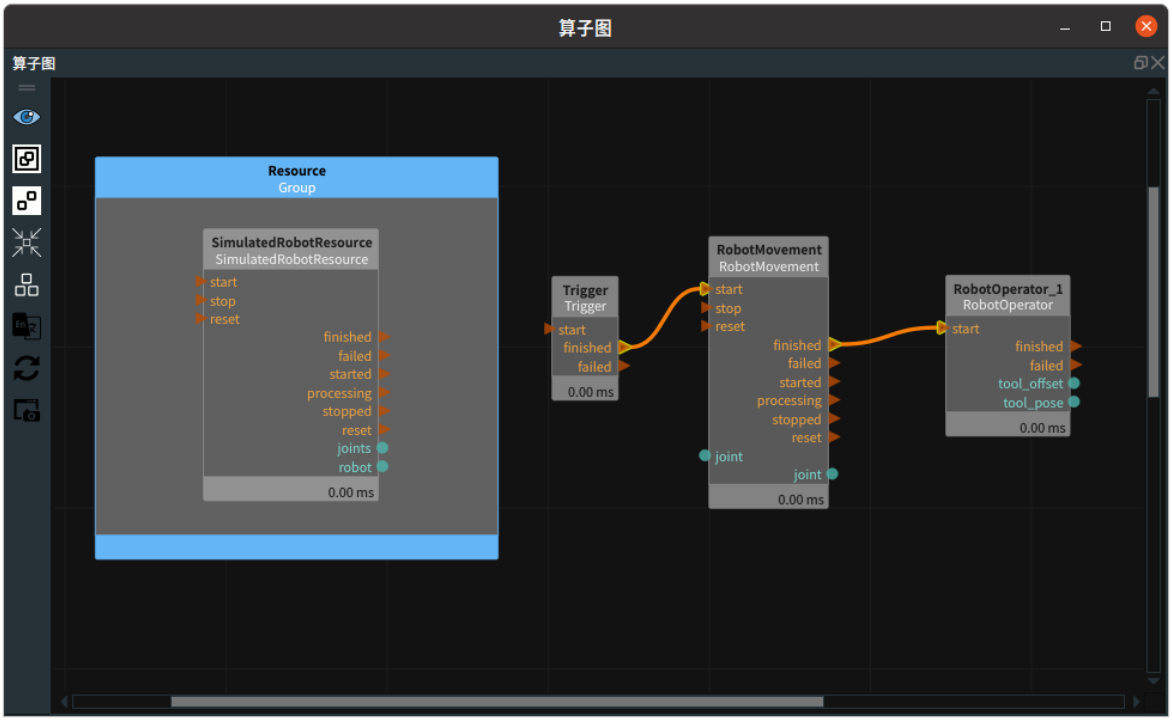
步骤1：算子准备

1. 添加 SimulatedRobotResource 资源算子至 Resource Group 。
2. 添加 Trigger 、RobotMovement 、RobotOperator 算子至算子图。

步骤2：设置算子参数

1. 设置 SimulatedRobotResource 算子参数：
   - 机器人模型文件 → ••• → 选择 robot 文件名 （ *example_data/UR5/UR5.rob* ）
   - 工具模型文件 → ••• → 选择 tool 文件名 （ *example_data/Tool/EliteRobotSucker1.tool.xml* ）
   - 机器人 →  可视
2. 设置 RobotMovement 算子参数：
   - 机器人资源名称 → SimulatedRobotResource
   - 类型 → MoveJoint
   - 关节 → 1.5708 0 0 0 0 0
3. 设置 RobotOperator 算子参数：
   - 机器人资源名称 → SimulatedRobotResource
   - 类型 → EmitToolPose
   - 工具坐标 →  可视

步骤3：连接算子



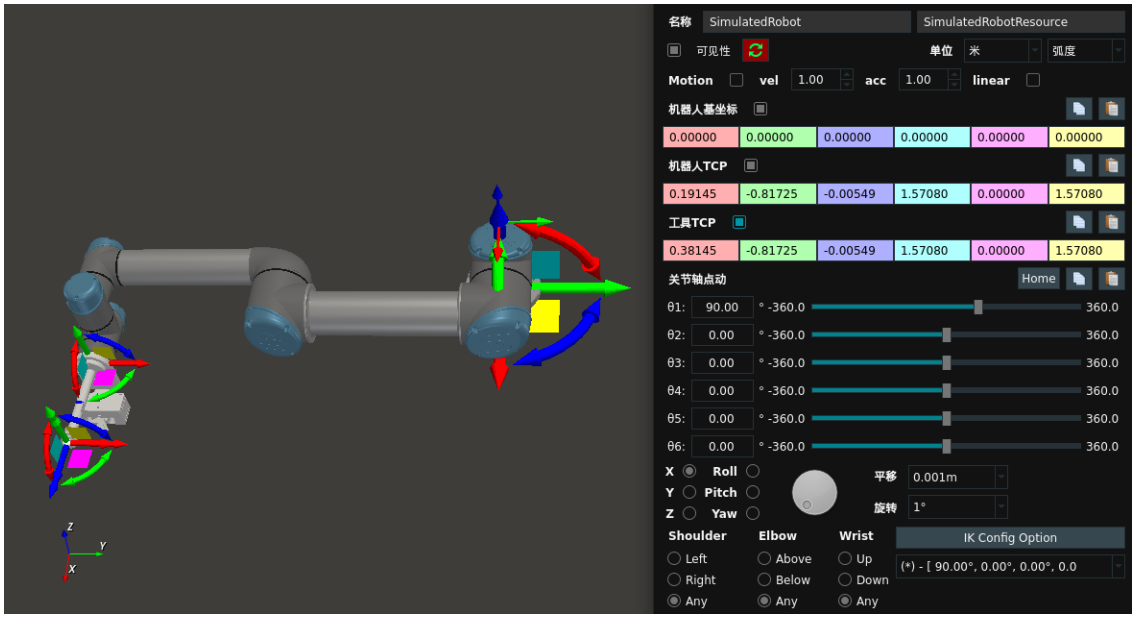
步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

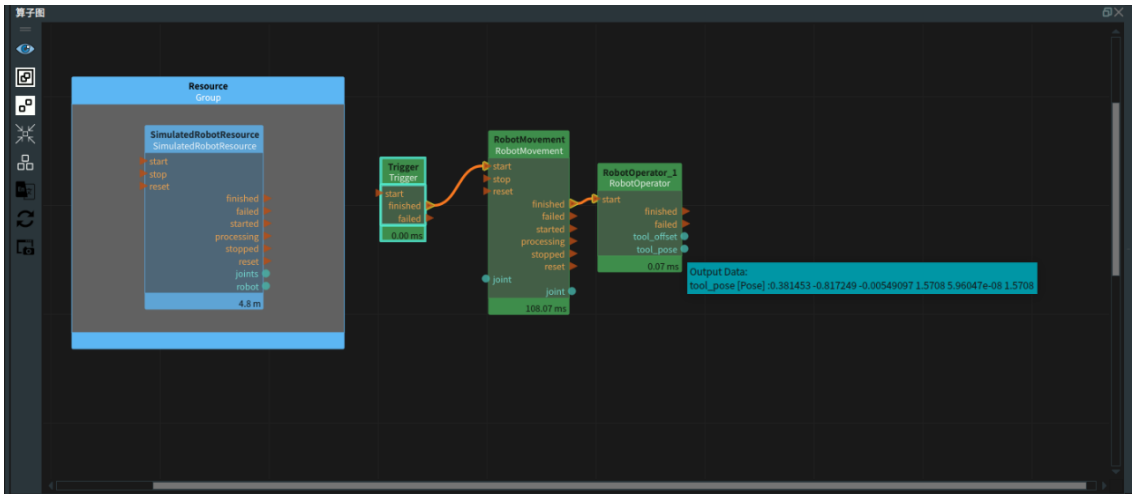
运行结果

1. 当算子运行结束后，可以在 RobotOperator 算子属性面板中看到 tool_pose 的参数值。

2. 在 3D 视图中显示当前加载的机器人 ，鼠标左键双击 3D 视图中的机器人，弹出机器人面板，将机器人基坐标、机器人 TCP 、工具 TCP 设为 True 。此时机器人当前工具 TCP 值为“ 0.38145 -0.81725 -0.00549 1.57080 0 1.57080 ”。



3. 在RobotOperator 算子右侧端口输出 tool_pose 数据。



ComputeDK

将 RobotOperator 算子的 **类型** 属性选择 ComputeDK ，用于计算机器人正运动学。

算子参数

- **机器人资源名称/robot_resource_name**：机器人资源名。选择在Resource Group 中添加的机器人资源。
- **TCP坐标/tcp_pose**：设置 tcp_pose 在 3D 视图中的可视化属性。
  - 打开 tcp_pose 可视化。
  - 关闭 tcp_pose 可视化。
  - 设置 tcp_pose 的尺寸。取值范围：[0.001,10]。默认值：0.1。
- **TCP坐标列表/tcp_pose_list**：设置 tcp_pose_list 在 3D 视图中的可视化属性。
  - 打开 tcp_pose_list 可视化。
  - 关闭 tcp_pose_list 可视化。

- 设置 tcp_pose_list 的尺寸。取值范围：[0.001,10]。默认值：0.1。

数据信号输入输出

输入：

说明：根据需求输入一种输入数据信号即可。

- `joint`：
  - 数据类型：JointArray
  - 输入内容：机器人关节弧度值数据
- `joint_list`：
  - 数据类型：JointArrayList
  - 输入内容：机器人关节弧度值数据列表

输出：

- `tcp_pose`：
  - 数据类型：Pose
  - 输出内容：机器人 tcp 位置 pose
- `tcp_pose_list`：
  - 数据类型：PoseList
  - 输出内容：机器人 tcp 位置 pose_list

功能演示

使用 RobotOperator 算子中 ComputeDK ，计算机器人正运动学。

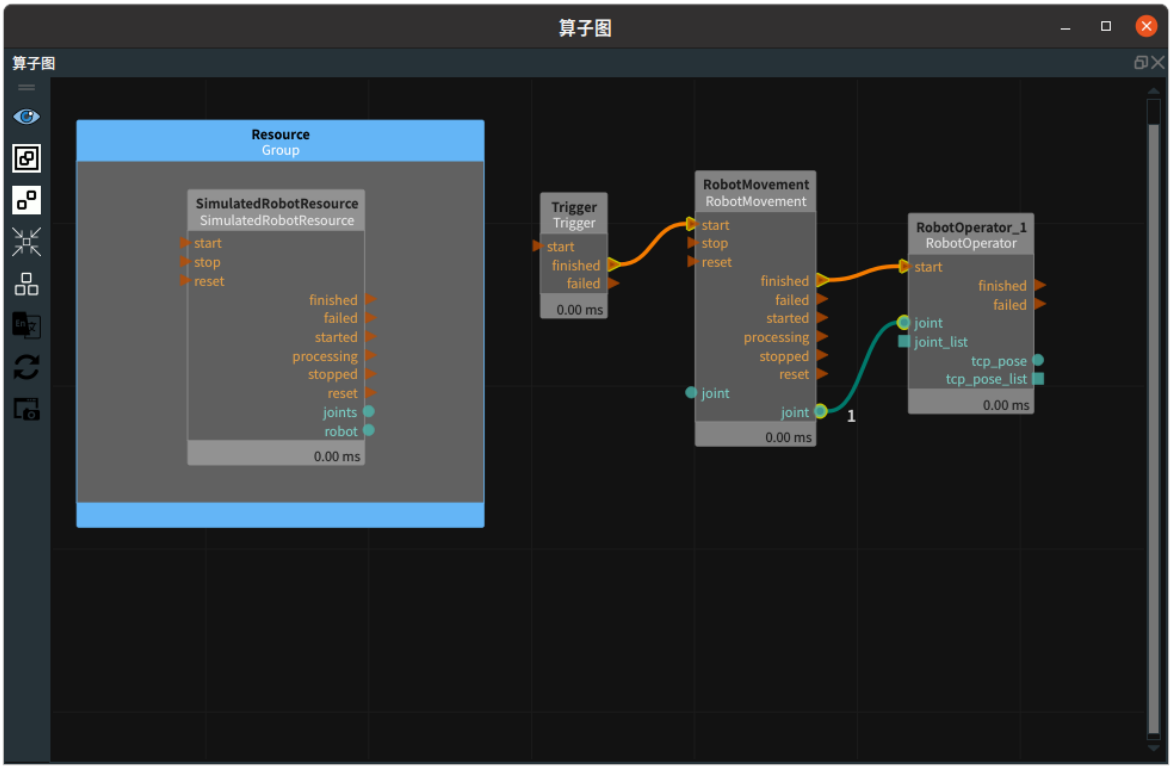
步骤1：算子准备

1. 添加 SimulatedRobotResource 资源算子至 Resource Group 。
2. 添加 Trigger 、RobotMovement 、RobotOperator 算子至算子图。

步骤2：设置算子参数

1. 设置 SimulatedRobotResource 算子参数：
   - 机器人模型文件 → ••• → 选择 robot 文件名 （ *example_data/UR5/UR5.rob* ）
   - 机器人 → 可视
2. 设置 RobotMovement 算子参数：
   - 机器人资源名称 → SimulatedRobotResource
   - 类型 → MoveJoint
   - 关节 → 1.5708 0 0 0 0 0
3. 设置 RobotOperator 算子参数：
   - 机器人资源名称 → SimulatedRobotResource
   - 类型 → ComputeDK
   - TCP坐标 → 可视

步骤3：连接算子

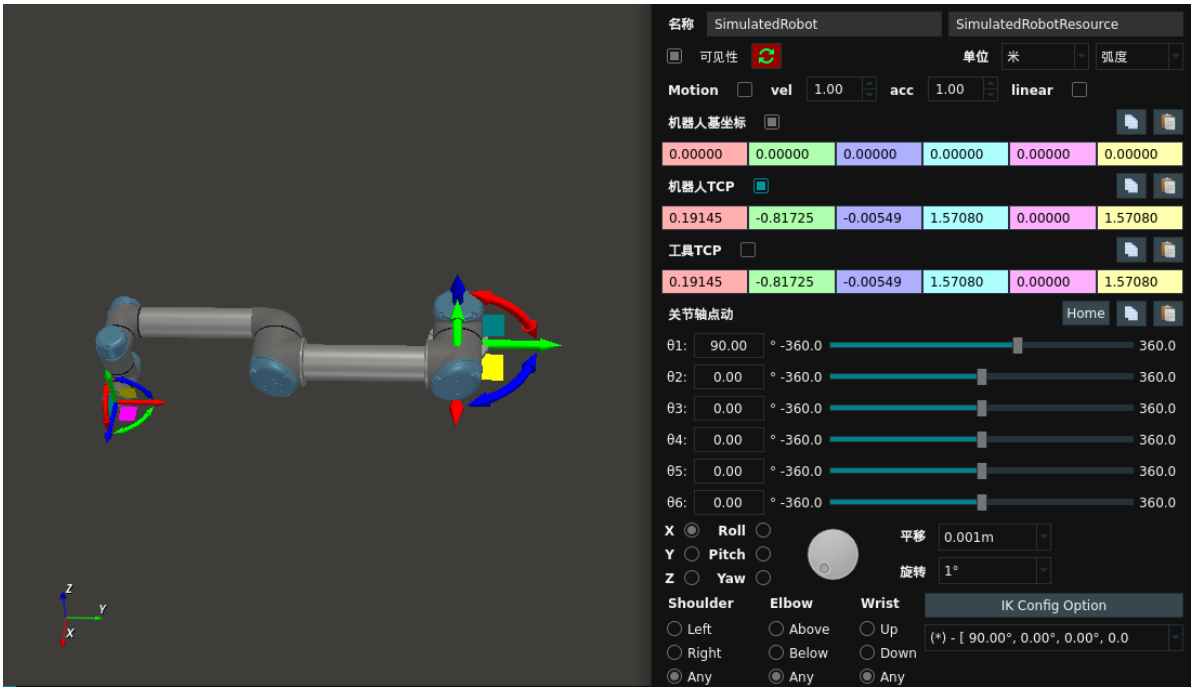


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

如下图所示，在 3D 视图中显示当前加载的机器人，仿真机器人按照 MoveJoint 方式移动关节值“1.5708 0 0 0 0 0”，鼠标左键双击 3D 视图中的机器人，弹出机器人面板，将机器人基坐标和机器人 TCP 设为 True ，获得当前机器人 tcp_pose ，与当前机器人面板的 TCP 保持一致。



ComputeIK

将 RobotOperator 算子的 **类型** 属性选择 ComputeIK ，用于计算机器人逆运动学。

算子参数

- **机器人资源名称/robot_resource_name**：机器人资源名。选择在Resource Group 中添加的机器人资源。
- **shoulder_flag**：机器人的肩关节。
  - Any：不限定肩关节的位置。
  - ShoulderLeft：优先输出肩关节在机械臂左侧的逆运动学的解。
  - ShoulderRight：优先输出肩关节在机械臂右侧的逆运动学的解。
- **elbow_flag**：机器人的肘关节。
  - Any：不限定肘关节的位置。
  - ElbowUp：优先输出肘关节朝上的逆运动学的解。
  - ElbowDown：优先输出肘关节朝下的逆运动学的解。
- **wrist_flag**：机器人的腕关节。
  - Any：不限定腕关节的位置。
  - Flip：翻转腕关节，一般情况下腕关节默认朝下。
  - NonFlip：不翻转腕关节。
- **机器人TCP/tcp**：机器人tcp 值。

数据信号输入输出

输入：

- **goal_pose**：
  - 数据类型：Pose
  - 输入内容：目标 pose 数据
- **ref_joint**：
  - 数据类型：JointArray
  - 输入内容：机器人参考关节弧度值数据

输出：

- **ik_solution**：
  - 数据类型：JointArray
  - 输出内容：机器人逆运动学解 joint 数据

功能演示

使用 RobotOperator 算子的中 ComputeIK ，计算机器人逆运动学。

步骤1：算子准备

1. 添加 SimulatedRobotResource 资源算子至 Resource Group 。
2. 添加 Trigger 、Emit 、RobotMovement 、RobotOperator 、MoveIK 算子至算子图。

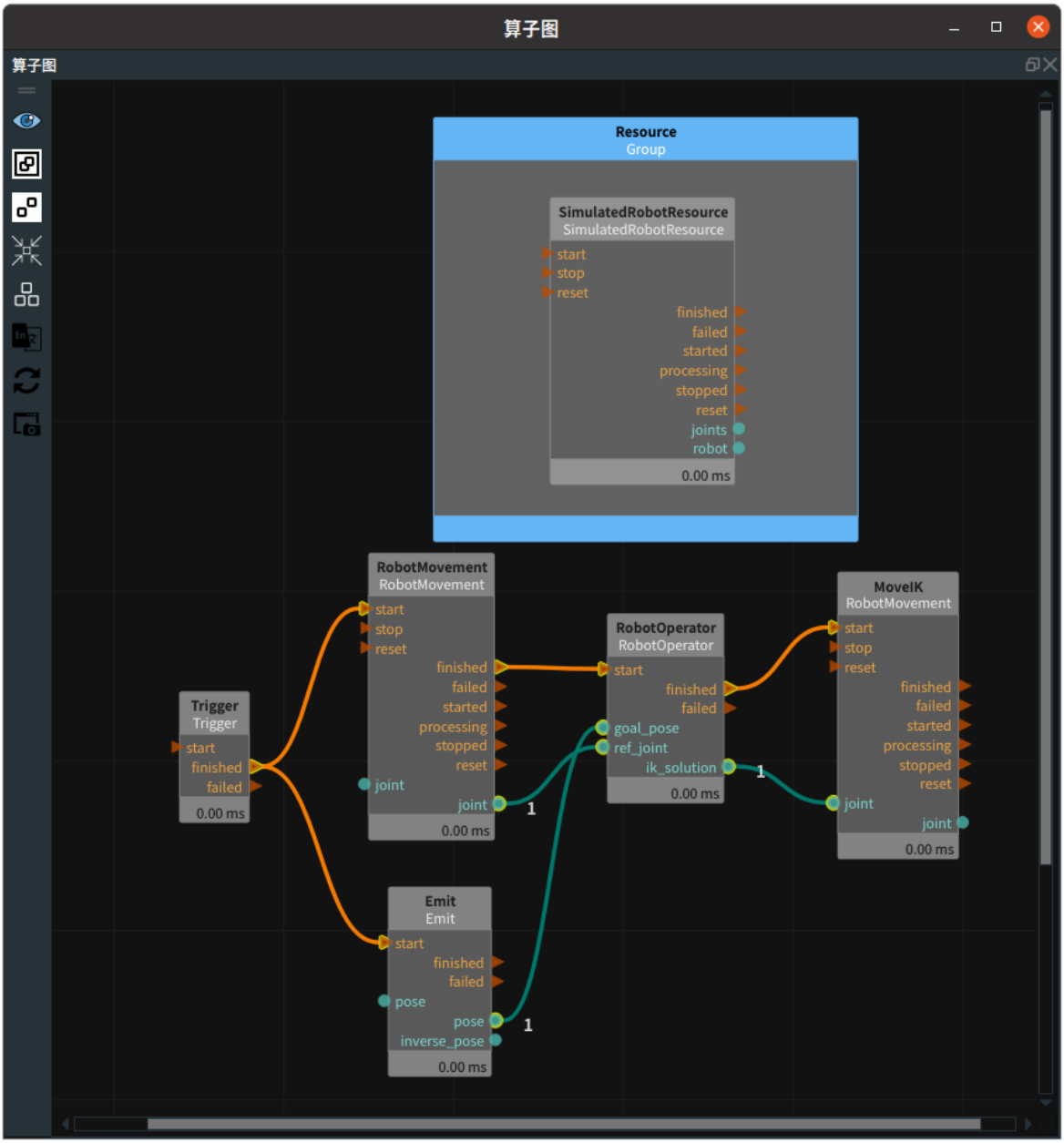
步骤2：设置算子参数

1. 设置 SimulatedRobotResource 算子参数：
   - 机器人模型文件 → ••• → 选择 robot 文件名 （ *example_data/UR5/UR5.rob* ）

- 机器人 → 可视

2. 设置 RobotMovement 算子参数：
   - 机器人资源名称 → SimulatedRobotResource
   - 类型 → MoveJoint
   - 关节 → 1.5708 0 0 0 0 0
3. 设置 RobotOperator 算子参数：
   - 机器人资源名称 → SimulatedRobotResource
   - 类型 → ComputeIK
4. 设置 Emit 算子参数：
   - 类型 → Pose
   - 坐标 → 0.2 -0.7 0.2 1.5708 0 1.5708

步骤3：连接算子

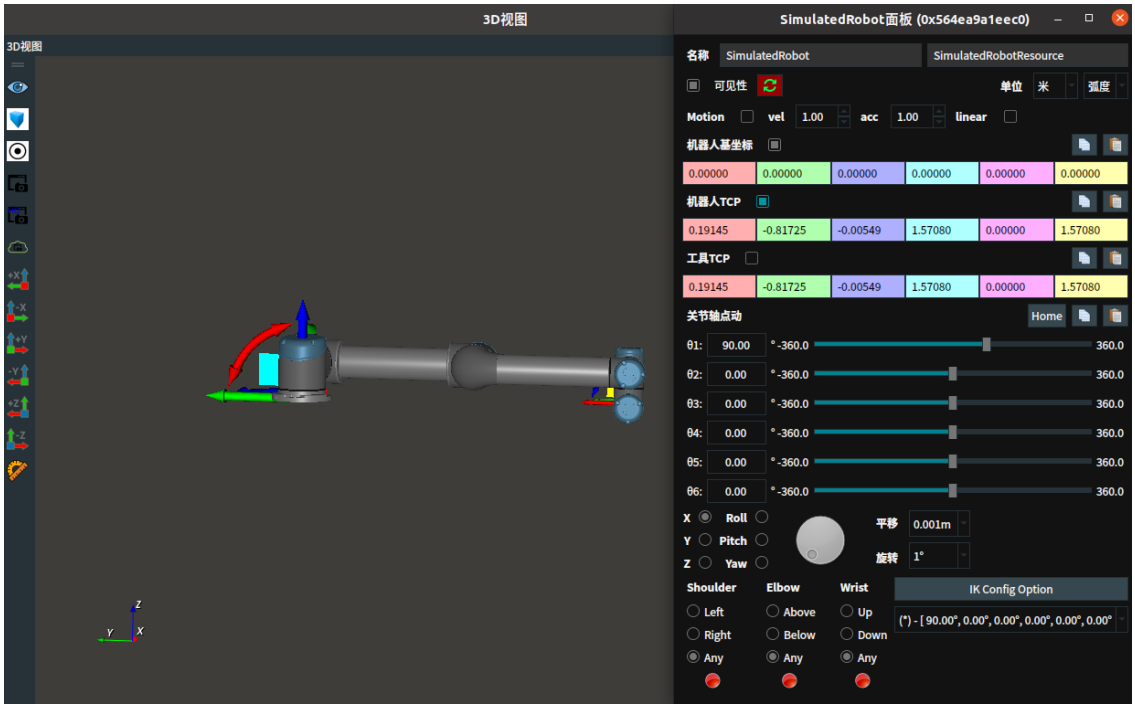


步骤4：运行

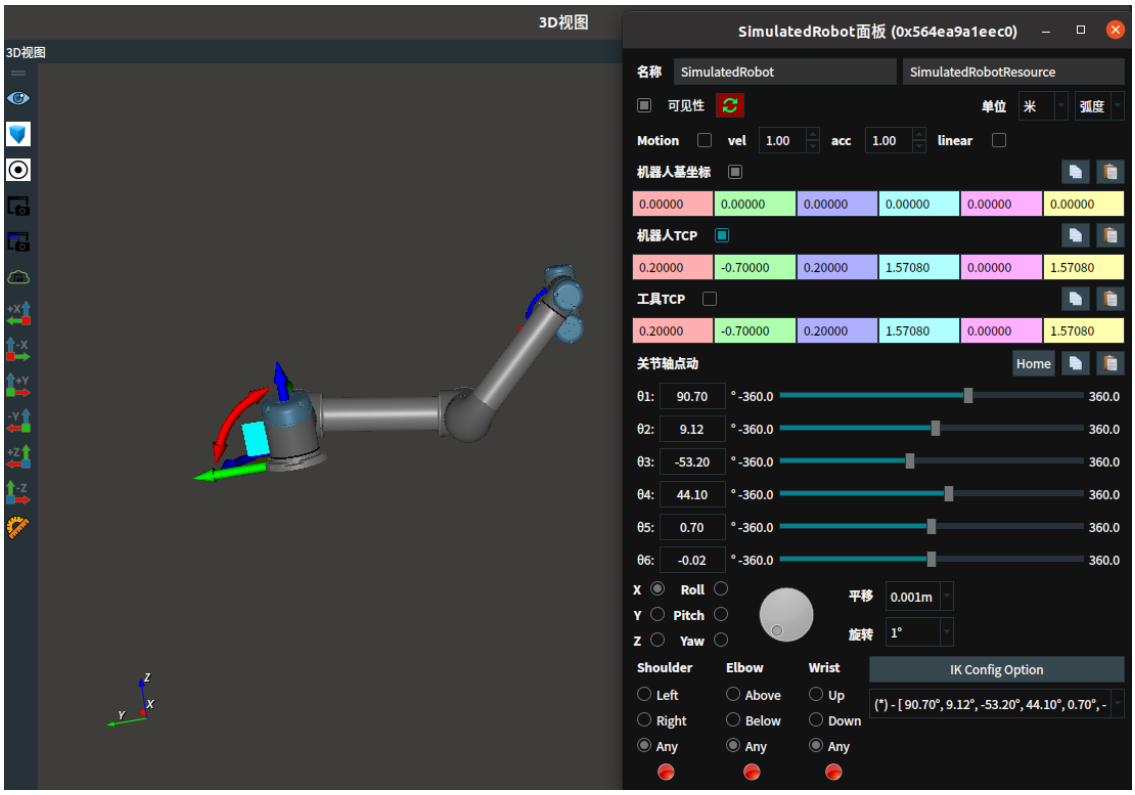
点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

1. 如下图所示，在 3D 视图中显示当前加载的机器人。仿真机器人按照 MoveJoint 方式移动关节值 “1.5708 0 0 0 0 0”，RobotOperator 算子通过 ComputeIK 计算得到 ik_solution，机器人再次移动到新的位置。



2. 如下图所示，RobotOperator 算子通过 ComputeIK 计算得到 ik_solution。



ComputeJoinPose

将 RobotOperator 算子的 **类型** 属性选择 ComputeJoinPose，用于计算各个旋转轴所在的 pose 位姿。

算子参数

- **机器人资源名称/robot_resource_name**：机器人资源名。选择在Resource Group 中添加的机器人资源。
- **关节坐标列表/joint_pose_list**：设置 joint_pose_list 在 3D 视图中的可视化属性。
  - 打开 joint_pose_list 可视化。
  - 关闭 joint_pose_list 可视化。
  - 设置 joint_pose_list 中 各个 pose 的尺寸。取值范围：[0.001,10]。默认值：0.1。

数据信号输入输出

输入

- **joint**：
  - 数据类型：JointArray
  - 输入内容：机器人关节弧度值

输出

- **joint_pose_list**：
  - 数据类型：PoseList
  - 输出内容：各个旋转轴所在的位姿

功能演示

使用 RobotOperator 算子的中 ComputeJoinPose ，计算各个旋转轴所在的 pose 位姿。

步骤1：算子准备

1. 添加 SimulatedRobotResource 资源算子至 Resource Group 。
2. 添加 Trigger 、RobotMovement 、RobotOperator 算子至算子图。

步骤2：设置算子参数

1. 设置 SimulatedRobotResource 算子参数：
   - 机器人模型文件 → ••• → 选择 robot 文件名 （ *example_data/UR5/UR5.rob* ）
   - 机器人 → 可视
2. 设置 RobotMovement 算子参数：
   - 机器人资源名称 → SimulatedRobotResource
   - 类型 → MoveJoint
   - 关节 → 1.5708 0 0 0 0 0
3. 设置 RobotOperator 算子参数：
   - 机器人资源名称 → SimulatedRobotResource
   - 类型 → ComputeJoinPose
   - 关节坐标列表 → 可视

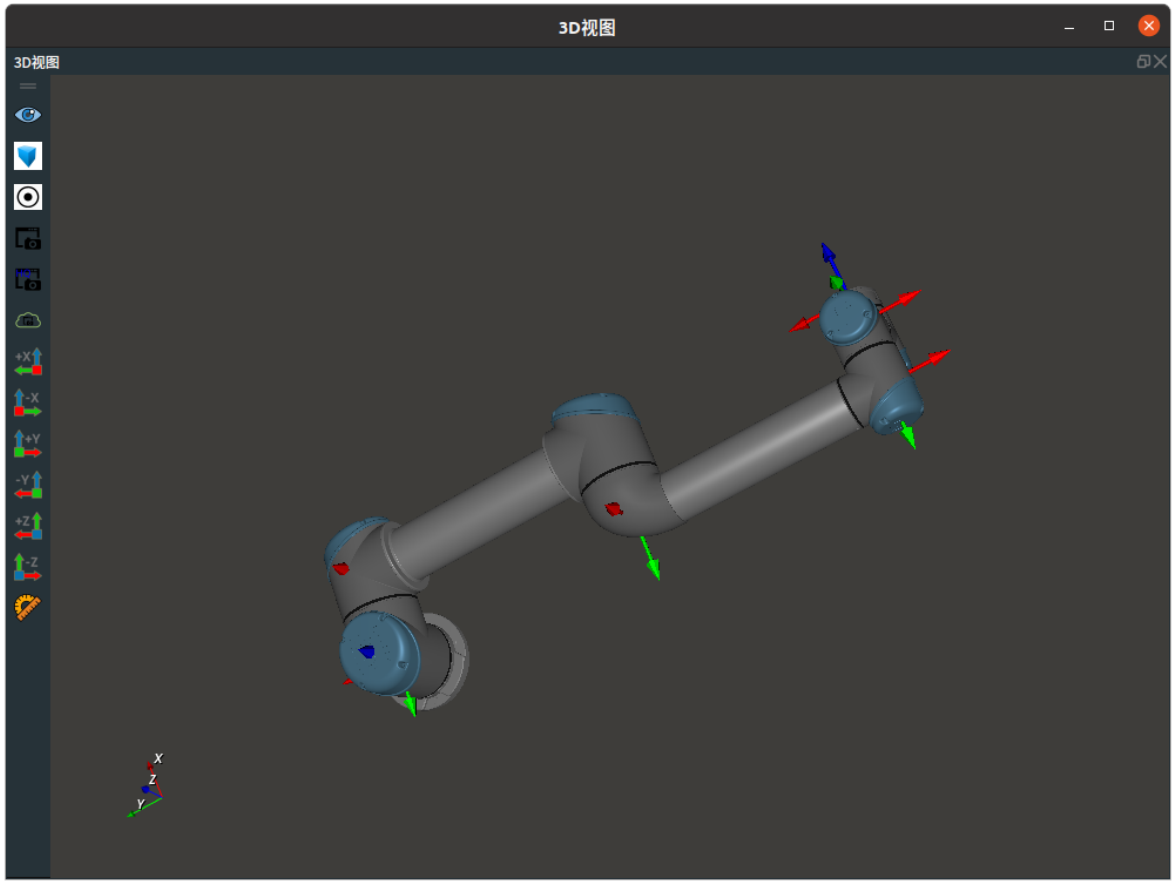
步骤3：连接算子

步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

1. 如下图所示，在 3D 视图中显示当前加载的机器人。仿真机器人按照 MoveJoint 方式移动关节值 “1.5708 0 0 0 0 0”，RobotOperator 算子通过 ComputeJoinPose 计算得到旋转轴所在的 pose 位姿 。



AttachObject

将 RobotOperator 算子的 **类型** 属性选择 AttachObject ，用于给机器人添加工具。

算子参数

- **机器人资源名称/robot_resource_name**：机器人资源名。选择在Resource Group 中添加的机器人资源。
- **物体模型文件/object_filename**：机器人工具名。

数据信号输入输出

输入

- **obj_pose**：
  - 数据类型：Pose
  - 输入内容：当未输入该值时，工具添加默认位置为机器人 TCP 位置。若输入，则默认位置以 obj_pose 进行平移和旋转。

  注意：当输入该值时，需要先打开 SimulatedRobotResource 算子中 robot_update 属性，再次触发算子。

功能演示

使用 RobotOperator 算子的中 AttachObject ，在机器人 TCP 处添加工具。

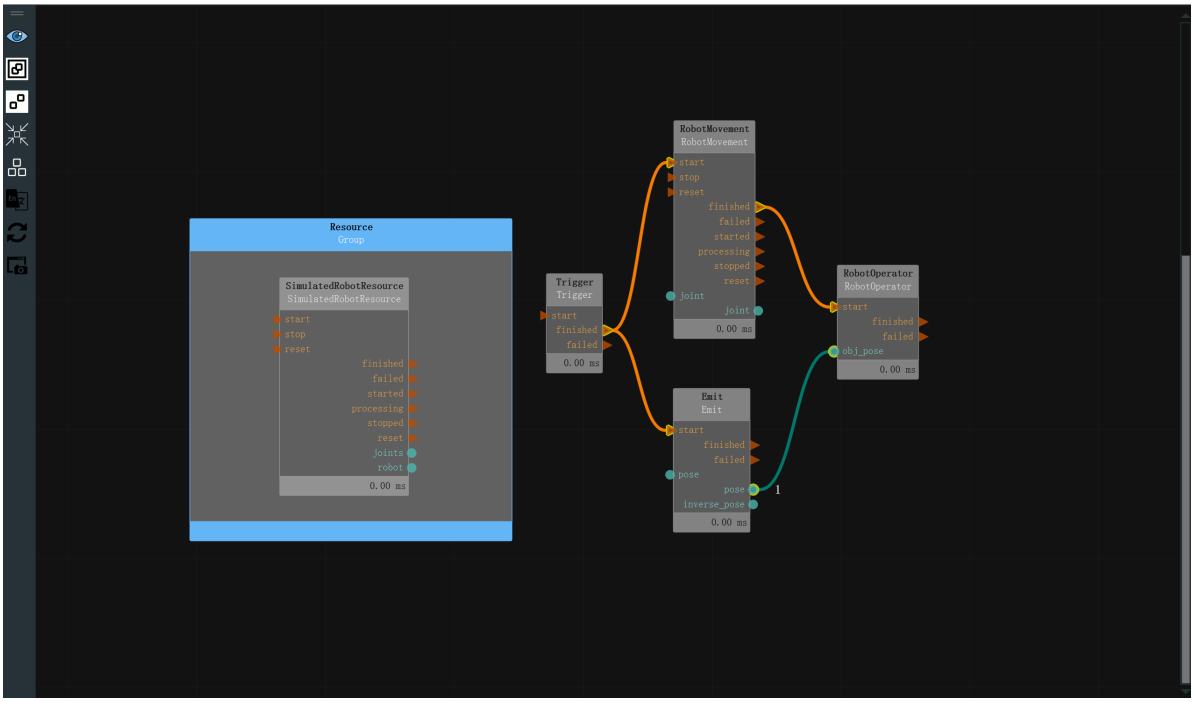
步骤1：算子准备

1. 添加 SimulatedRobotResource 资源算子至 Resource Group 。
2. 添加 Trigger 、Emit 、RobotOperator 算子至算子图。

步骤2：设置算子参数

1. 设置 SimulatedRobotResource 算子参数：
    - 机器人模型文件 → ●●● → 选择 robot 文件名 （*example_data/UR5/UR5.rob* ）
    - 机器人 →  可视
2. 设置 RobotOperator 算子参数：
    - 机器人资源名称 → SimulatedRobotResource
    - 类型 → AttachObject
    - 物体模型文件 → ●●● → 选择 tool 文件名（*example_data/Tool/EliteRobotSucker2.obj* ）

步骤3：连接算子

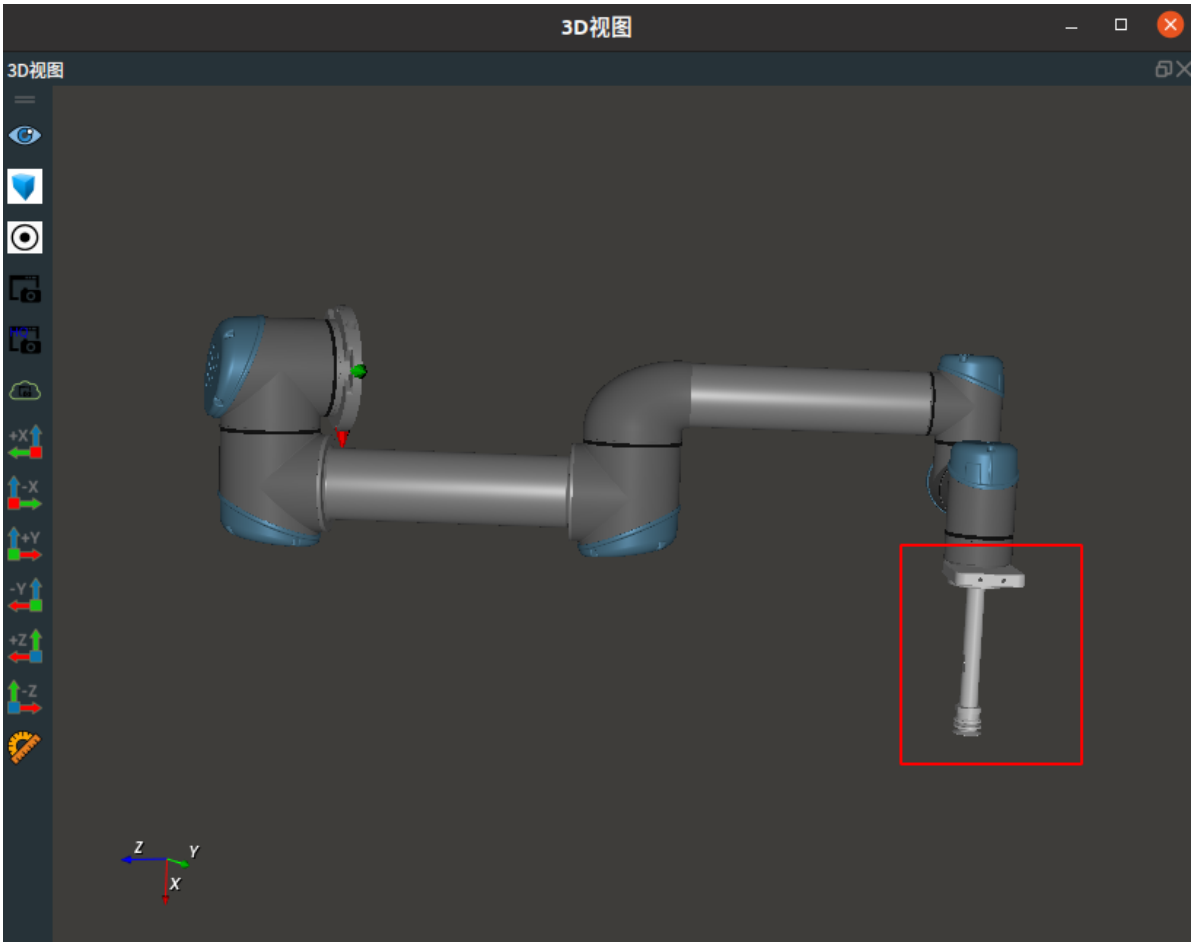


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

结果如下图所示，在机器人 TCP 处添加了工具。



DetachObject

将 RobotOperator 算子的 **类型** 属性选择 DetachObject，用于删除机器人工具。

算子参数

- **机器人资源名称/robot_resource_name**：机器人资源名。选择在Resource Group 中添加的机器人资源。

功能演示

使用 RobotOperator 算子的中 DetachObject，删除机器人仿真控制资源中加载的机器人工具。

步骤1：算子准备

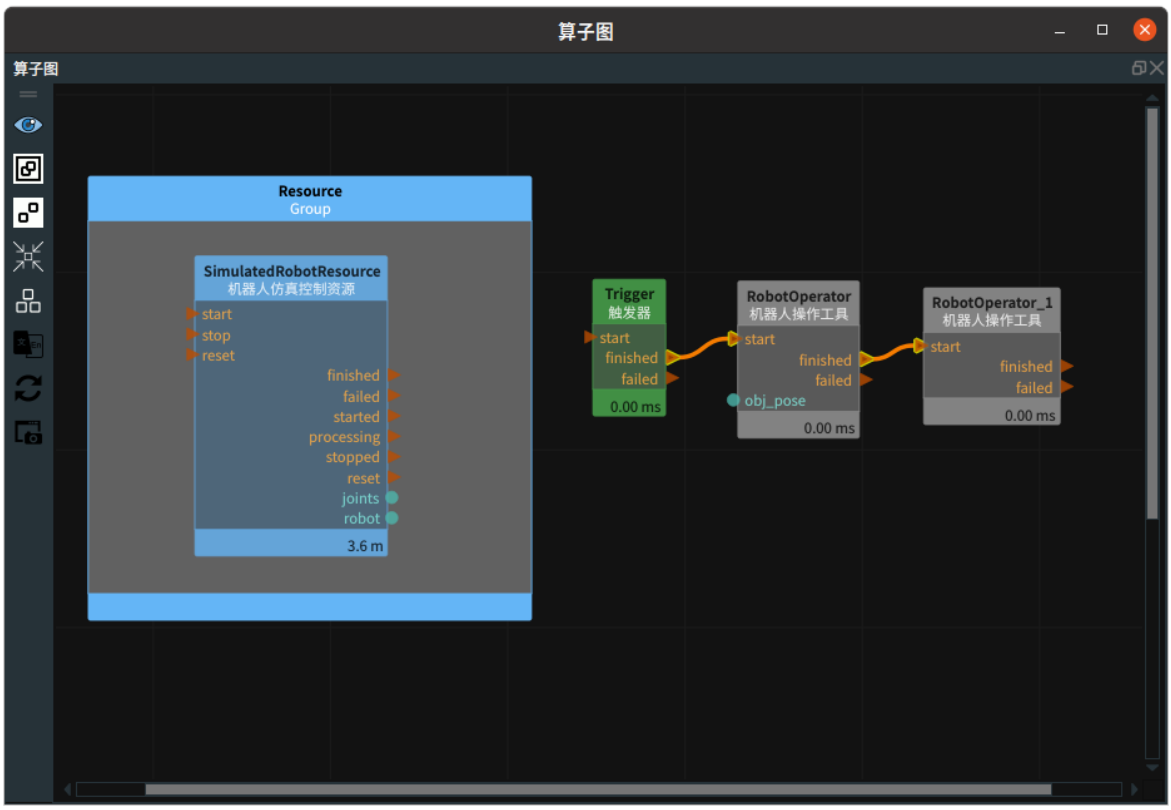
1. 添加 SimulatedRobotResource 资源算子至 Resource Group 。
2. 添加 Trigger 、RobotOperator （2个） 算子至算子图。

步骤2：设置算子参数

1. 设置 SimulatedRobotResource 算子参数：
    - 机器人模型文件 → ••• → 选择 robot 文件名 （*example_data/UR5/UR5.rob* ）
    - 机器人 → 可视
2. 设置 RobotOperator 算子参数：
    - 机器人资源名称 → SimulatedRobotResource
    - 类型 → AttachObject
    - 物体模型文件 → ••• → 选择 tool 文件名（*example_data/Tool/EliteRobotSucker2.obj* ）

3. 设置 RobotOperator_1 算子参数：
    - 机器人资源名称 → SimulatedRobotResource
    - 类型 → DetachObject

步骤3：连接算子

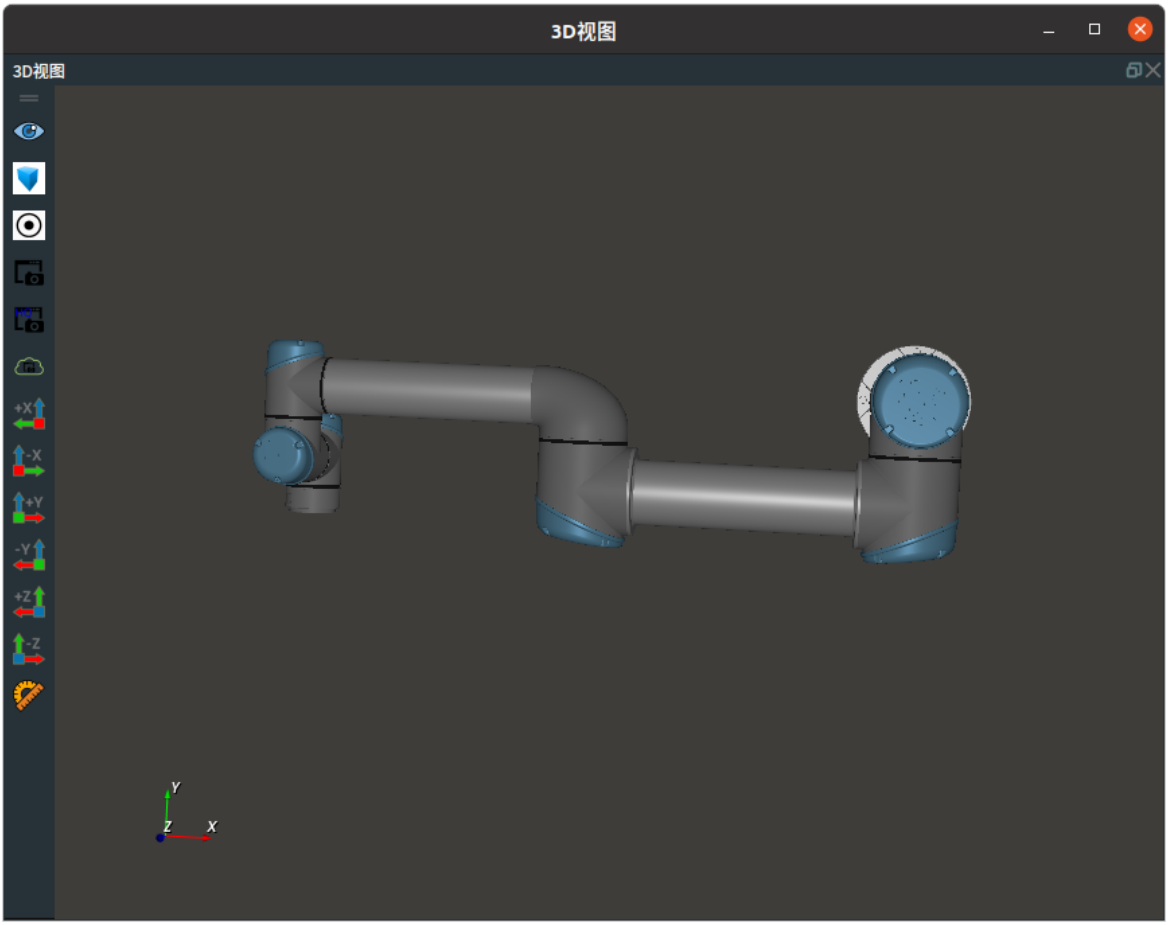


步骤4：运行

点击 RVS 运行按钮，触发 Trigger 算子。

运行结果

结果如下图所示，已删除添加的机器人工具。添加机器人工具结果请查看本文 AttachObject 章节。


3D视图
3D视图
+X
-X
+Y
-Y
+Z
-Z
Y
Z
X