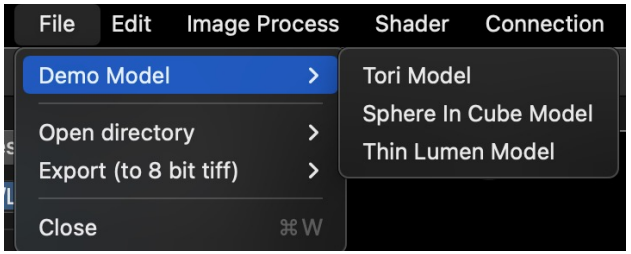
Acto3D Instruction

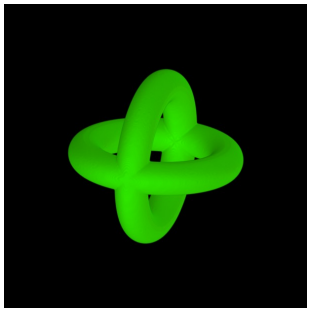
– Version 1.7.8 –

Operation Verification (Using demo model)

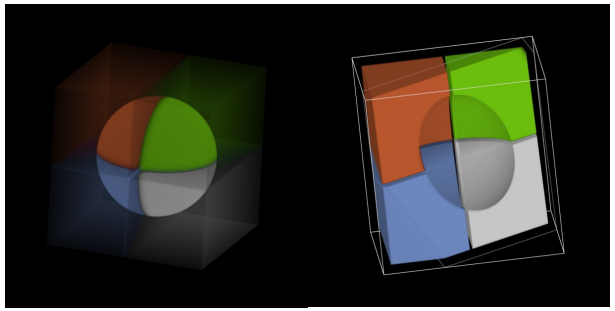


From **File > Demo Model**, you can load some demo models used in our article.

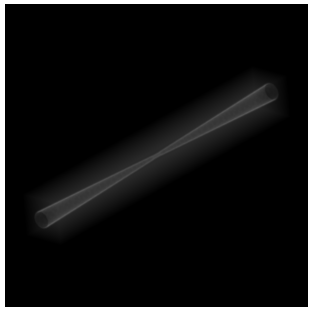
Tori model



Sphere in cube model



Lumen model



First, please ensure that these models can be loaded without any issues.

Data Preparation

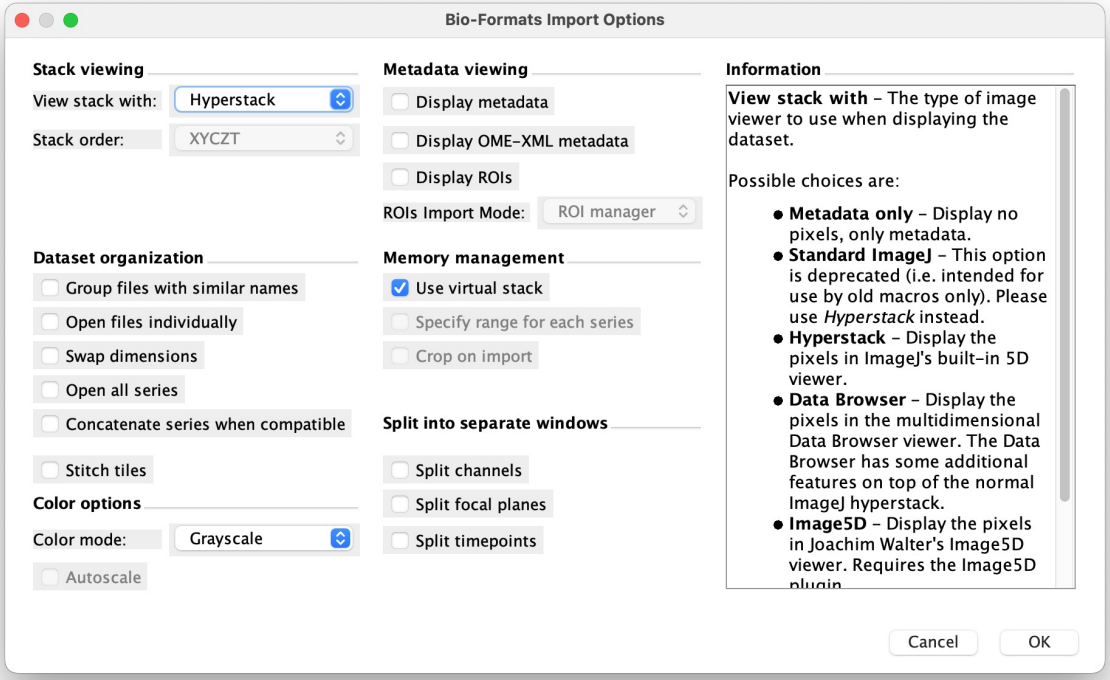
Acto3D accepts following data formats

	16 bits / channel	8 bits / channel
Multipage TIFF converted by Fiji	1 – 4 channels	1 – 4 channels
The information on voxel resolution stored within the TIFF files is utilized for isotropic display.		
TIFF stacks (†)	16 bits Grayscale images (Single channel)	32 bits RGBA images (*) 24 bits RGB images 8 bits Gray scale images
PNG stacks (†) JPG stacks (†)(††)		32 bits RGBA images (*) 24 bits RGB images 8 bits Gray scale images
Binary Data (TCP)	When using 'Acto3D_py', it's possible to directly input data via TCP. This allows for the observation of ndarray objects, which are being worked on in Python, in three dimensions directly within Acto3D.	

(†) : When providing a sequence of stack files, ensure the filenames are in a consecutive numerical order, such as "~~001.tif", "~~002.tif", "~~003.tif", and store them all in the same folder.
(††) : JPG is generally not recommended to use as it undergoes irreversible compression.
(*) : Normally, the "A" in a 32-bit image refers to the opacity channel, but in Acto3D, the "A" can be used as the data for the fourth channel by storing the data of each fluorescent channel in R, G, B, and A.

Convert images to TIFF format

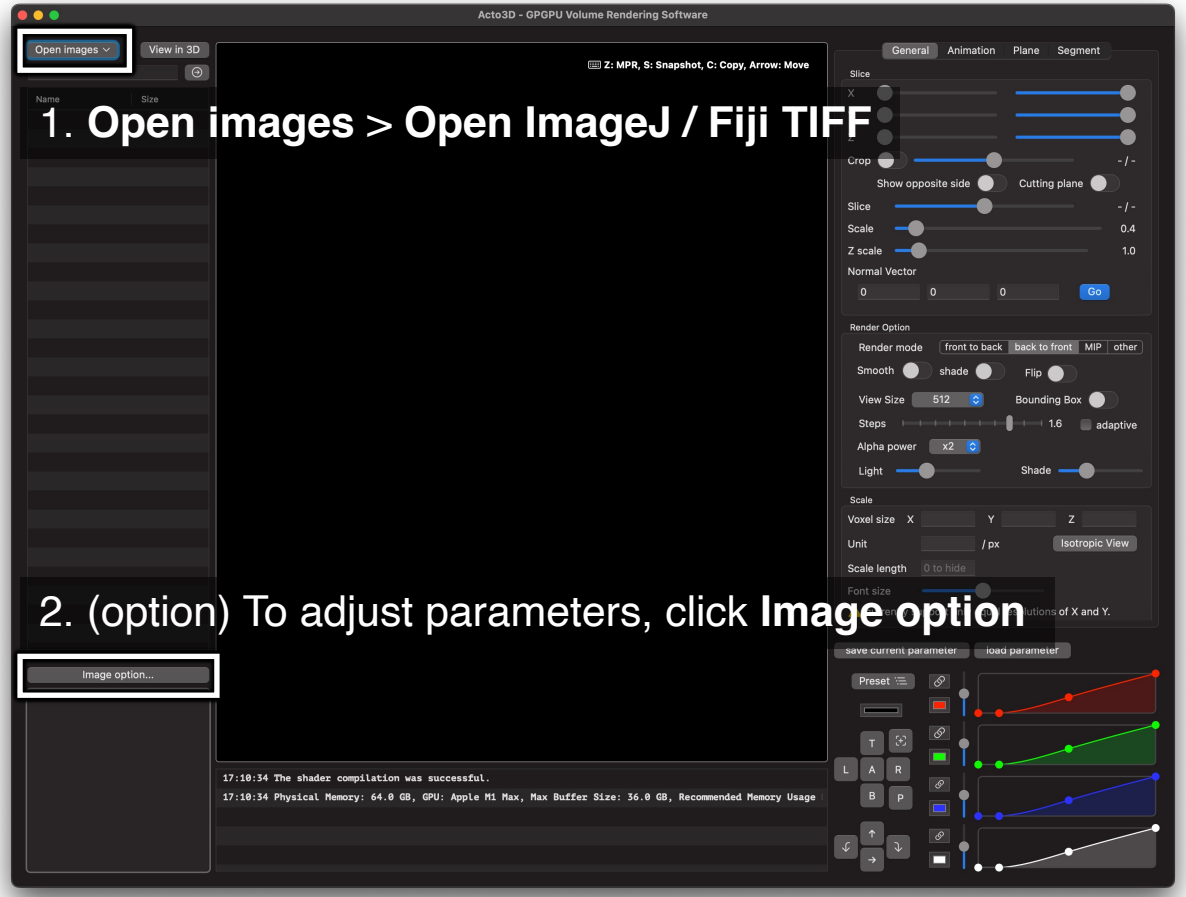
First, you need to convert manufactures format to TIFF using Fiji



Fiji
(version 2.3.0 / 1.53s)

Click **File** > **Save As** > **Tiff...**.

Load images in Acto3D




```
Physical Memory: 64.0 GB, GPU: Apple M1 Max, Max Buffer Size: 36.0 GB,
```

The **Max Buffer Size** represents the maximum amount of memory that can be allocated on the GPU for one 3D image. Images must fit within this capacity.

To calculate the memory required for 3D visualization of the image, use the following formula:

$$\text{Max Buffer Size} = [\text{width}] \times [\text{height}] \times [\text{depth}] \times [\text{channel}] / 1024^3$$

Note: This formula applies regardless of whether the image is stored in 8bit or 16bit.

Adjust parameters

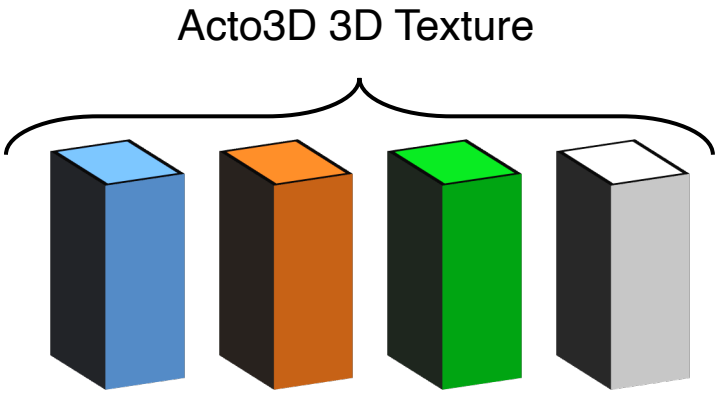
- Adjust display ranges (if you have adjusted this in Fiji, Acto3D uses the value)

- Setting the Voxel Size
  - When loading the microscope manufacturer's format in Fiji, it is automatically set in the TIFF metadata. If it is not set, you will need to determine it yourself based on the parameters at the time of imaging.

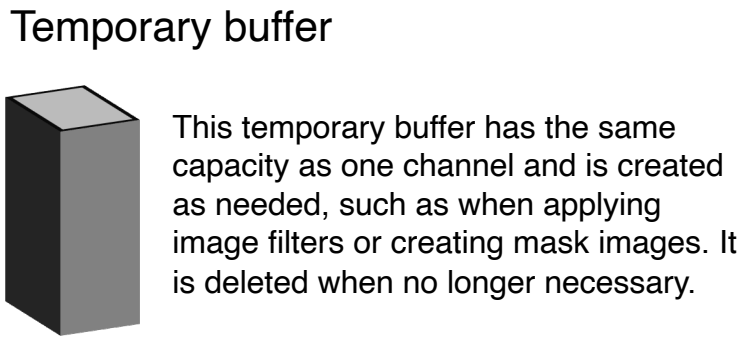
- Selecting the Number of Channels
  - If sufficient memory is available, select 4. If you need to conserve memory usage, select 1 if it's a single-channel image, 2 for a two-channel image, and 4 for a three or four-channel image.

- Ignore saturated pixels
  - If there is a strong non-specific signal and you want to ignore areas with extremely high pixel values, you can set the corresponding pixel areas to 0 by turning this ON.

(Additional Information on Channels)

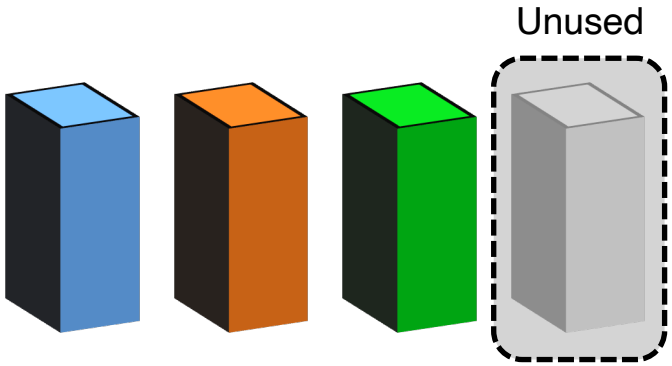


In Acto3D, an area for four channels is reserved regardless of the number of channels in the input image. If there are any unused channels, they can be used to store image filters or mask results.



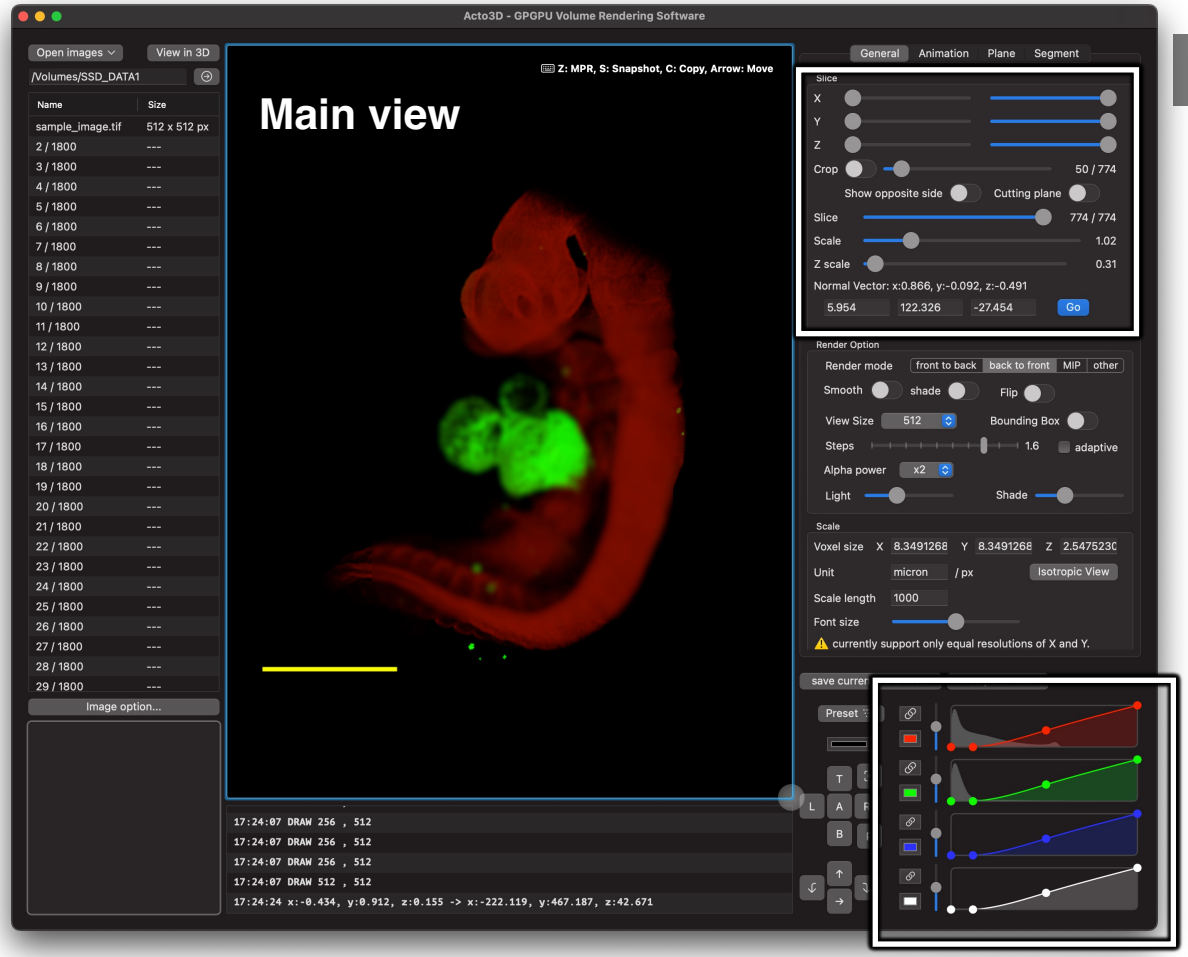
This temporary buffer has the same capacity as one channel and is created as needed, such as when applying image filters or creating mask images. It is deleted when no longer necessary.

If the input image has three channels and Acto3D creates a four-channel 3D texture, users have the option to store the outcomes of image filters or mask results in the unused channel area.



If the number of channels reserved by Acto3D is fewer than the channels of the input image, there will be no unused channels, thus any created results will overwrite an existing channel.

Screen Layout

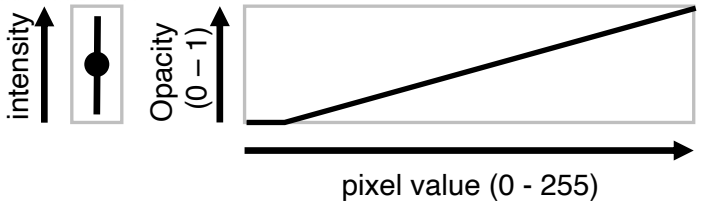


Slice setting

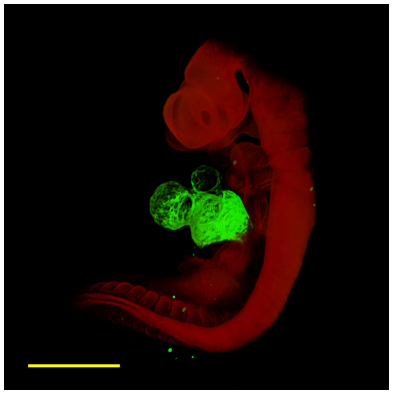
Set the Z scale as the Z voxel size divided by the XY voxel size.
For example, if XY is 2μm/pixel and Z is 5μm/pixel, set it to 2.5. If it has already been set on page 3, the correct value will be set automatically.

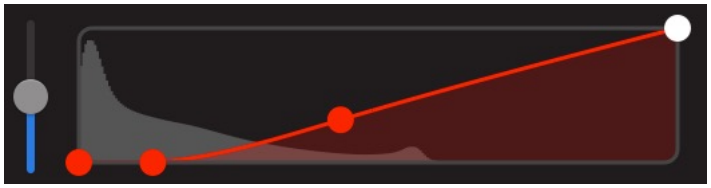
Color tone and transfer function

In this section, you determine the opacity corresponding to each pixel value. Clicking allows you to freely increase the control points. Right-clicking on a control point allows you to delete it. Right-clicking elsewhere will display additional options.



Adjust transfer function (adjust opacity) (1 / 2)





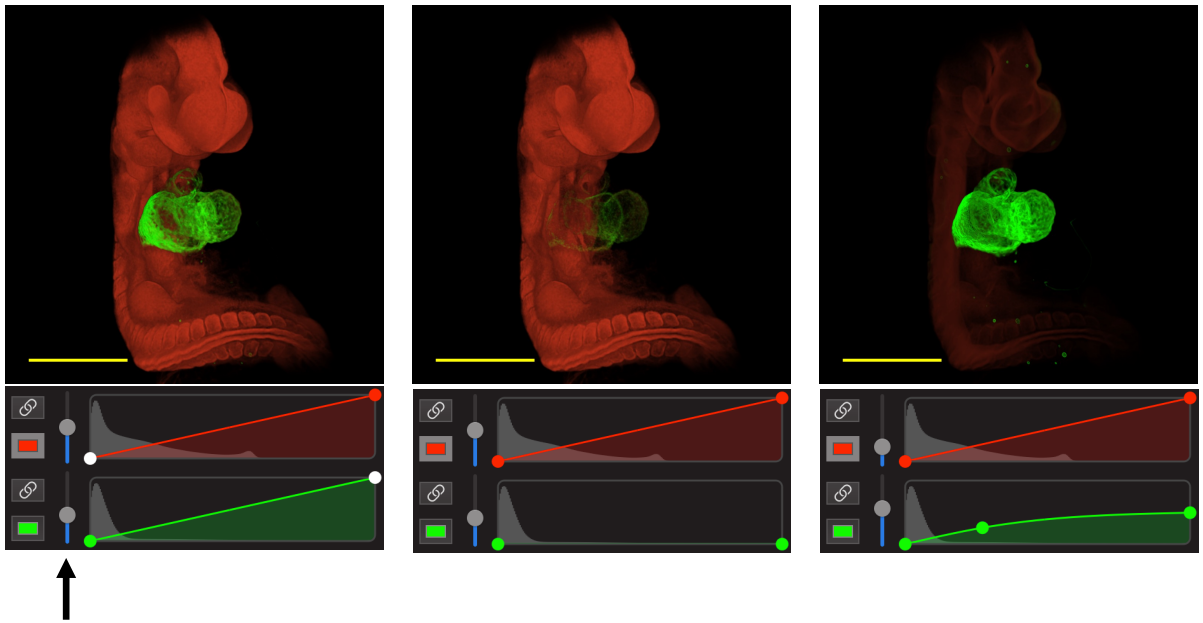
This setting makes parts with low pixel values transparent, which is effective when there is noise at a noticeably lower brightness than the original signal, such as non-specific signals.

This setting makes parts with pixel values close to zero transparent, while still displaying areas of weak brightness. Areas where pixel values exist become opaque, making it easier to observe the overall object.

This setting allows parts with low pixel values to be transparent, and by giving a certain opacity to areas of strong brightness that are considered signals, it enables the construction of the interior further beyond the surface.

Based on these settings, make fine adjustments to construct your ideal image. Although it is possible to specify the opacity for each channel, by

Adjust transfer function (adjust opacity) (2 / 2)

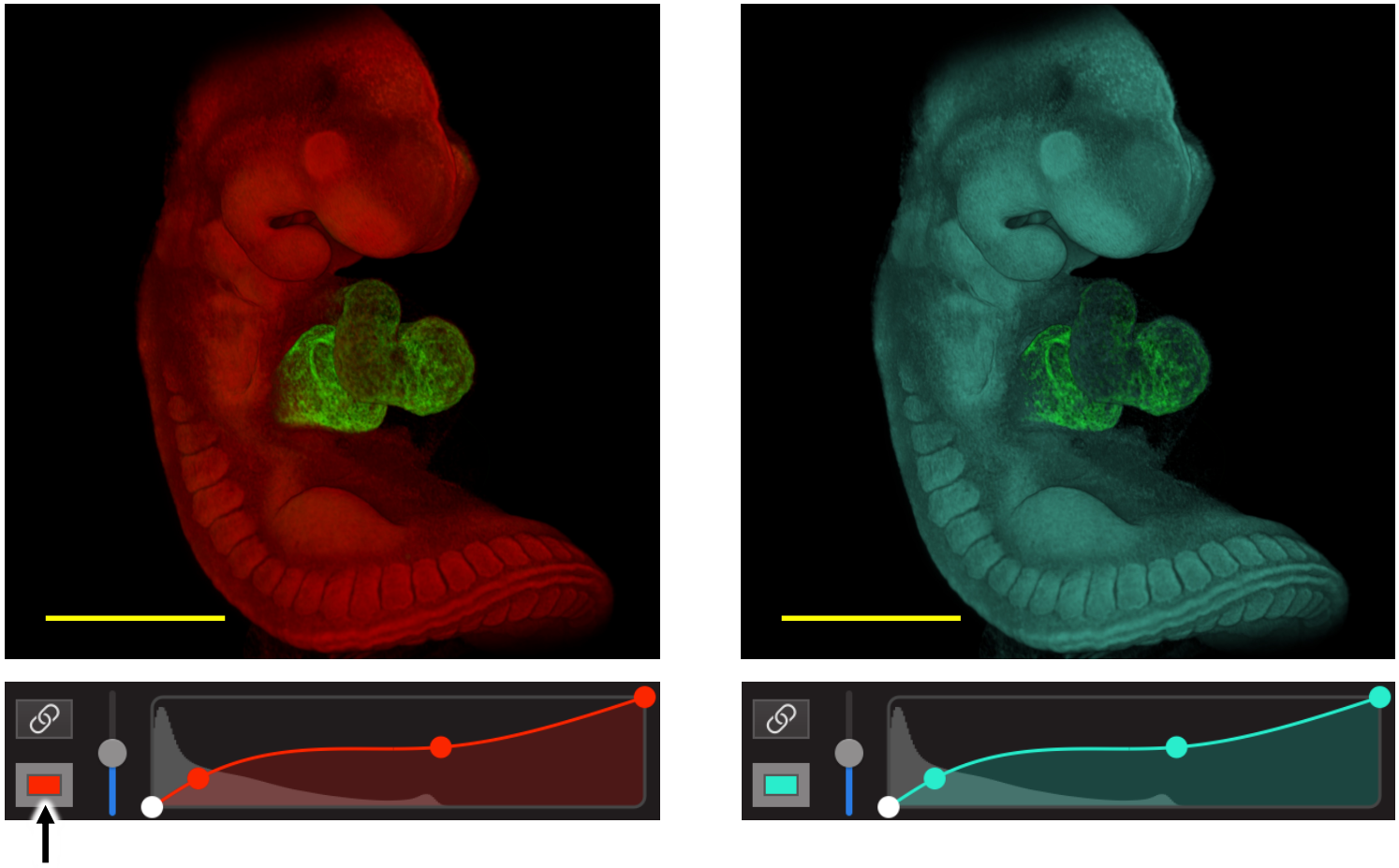


The slider to adjust pixel intensity for the channel.

Based on these settings, make fine adjustments to construct your ideal image.

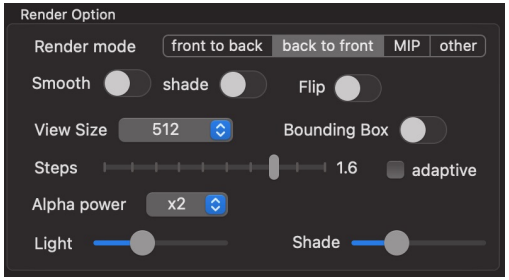
Although it is possible to specify the opacity for each channel, by default, in order to prevent unnatural displays (such as semi-transparent structures in the back being displayed even though there are opaque objects in the front), the opacity at a given sampling point is set to the maximum value among the four channels. If you want to make a certain channel more transparent, you can adjust its intensity using the slider.

Change color



You can specify the color tone for each channel.

Rendering options (1 / 2)



Render mode **Press ‘Z’ key** at any time to generate **MPR** images

Select the algorithm to use for volume rendering. The default selection is ‘front to back.’ ‘back to front’ rendering is considered to provide more accurate visuals, however, it increases the computational cost. If you wish to change the background color from black to another color, please select the back to front. MIP (Maximum Intensity Projection) is a method that adopts the pixel with the highest brightness in the line of sight. With a image in MIP, the depth becomes indistinguishable.

Smooth This setting uses linear interpolation to make the sampling along the line of sight smooth.

Shade When sampling along the line of sight, this option takes into account the gradient with surrounding pixels.

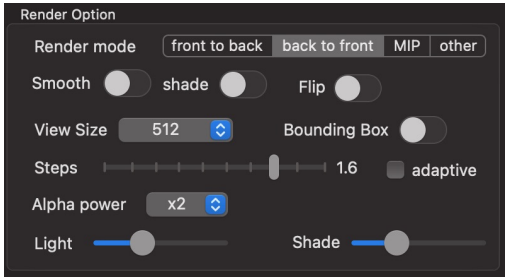
Flip This option allows you to change whether to arrange the image stack from back to front or from front to back, effectively allowing you to flip the image horizontally.

View size Select the size of the image to be created.
Regardless of this value, the original image used for rendering is not resized.

Bounding box Display the outer frame of the 3D volume.

Steps Specify the interval for sampling along the line of sight. Specify 1.0 for sampling at the same interval as 1 pixel. However, since this can drastically increase the cost of calculations, by default it is set to 1.6, with some degree of thinning. Note that when the number of images is small, or the resolution of the image is low, jagged edges may appear in the image, so setting it below 1.0 will smooth it out. Please note that in volume rendering, the final image changes depending on the number of times the same transparency level is overlaid.

Rendering options (2 / 2)



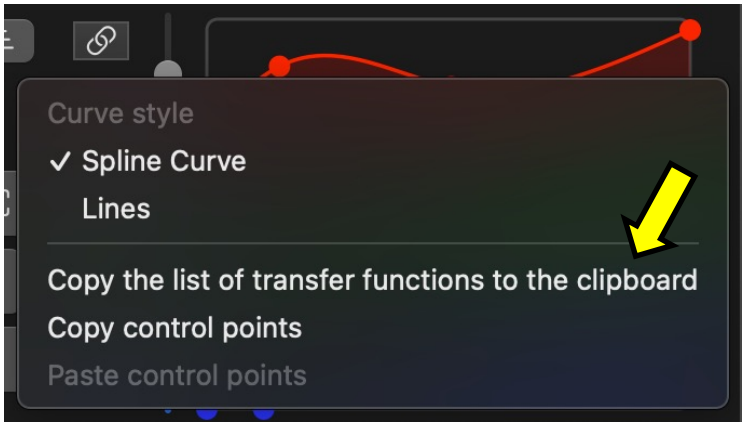
Adaptive This is a setting related to the above view size and steps. When OFF, a 3D image is created from the original image and finally adjusted to the view size. When ON, the image size is adjusted (usually shrinked) to match the view size first, and the 3D image is constructed based on that image. Both methods do not change the isotopic display, but in many cases, the number of calculations drastically decreases when ON.

Alpha power In volume rendering, many images are overlaid, so if the transparency is not set to a very low value, only the surface is often observed. Therefore, by default, the alpha value specified in the graph (in the range of 0.0-1.0) is squared. If it becomes too transparent, you can ignore this setting by setting it to x1. If you want to make it more transparent, you can set it to x3 to cube it.

Light Adjust the overall brightness.

Shade If shade is turned on, it increases the effect of the gradient with surrounding pixels.

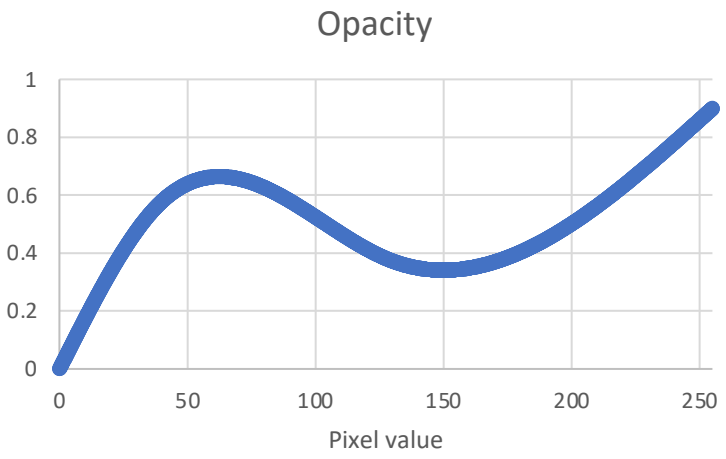
Check transfer function



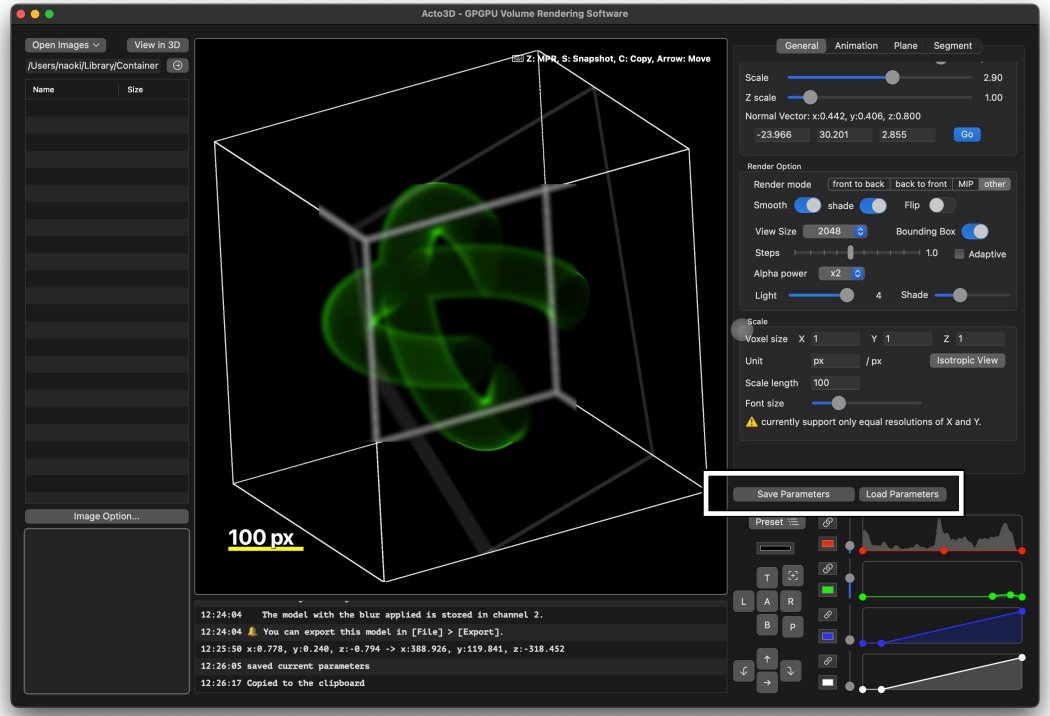
To obtain the the transfer function:
Right click in the graph area
(in places other than control points)

	A	B
1	Pixel value	Opacity
2	0	0
3	0.1	0.0017631
4	0.2	0.00352618
5	0.3	0.00528924
6	0.4	0.00705227
7	0.5	0.00881525
8	0.6	0.01057816
9	0.7	0.01234101
10	0.8	0.01410378
11	0.9	0.01586645
12	1	0.01762901
13	1.1	0.01939145
14	1.2	0.02115377
15	1.3	0.02291594
16	1.4	0.02467796
17	1.5	0.02643981
18	1.6	0.02820148

These values can be pasted into Excel

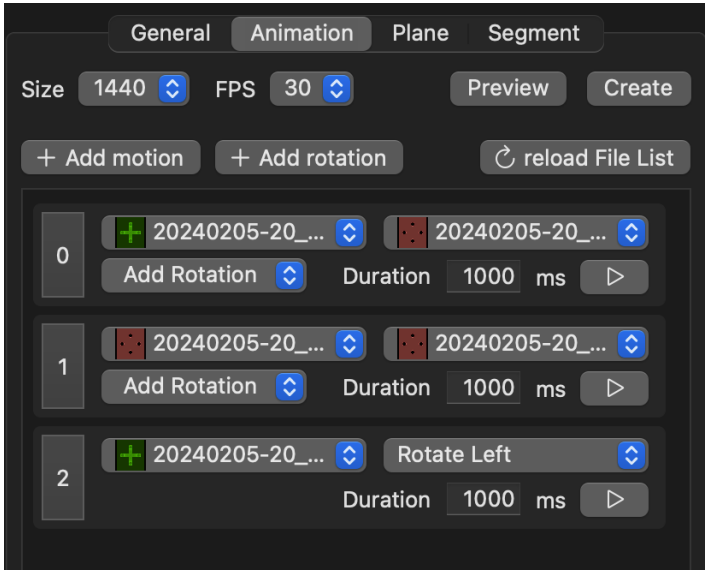


Save and load parameters



Using these buttons, you can save and load the parameters for rendering.

Animation



You can create videos in the Animation tab.
To create an animation, save the parameters
at the beginning and the end using [Save Parameters] button.

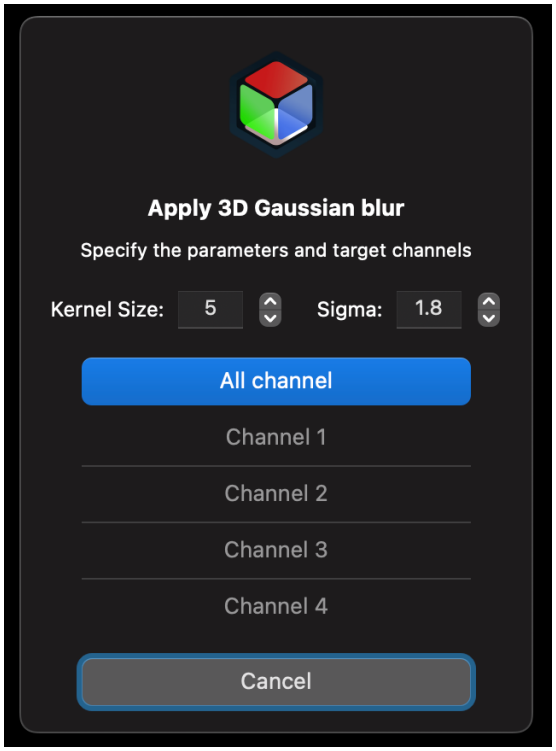
By setting the parameters for both the beginning and the end,
you can smoothly transition through most of the settings available
in Acto3D, such as movement, rotation, and color adjustments.

The video size will match the pixel width specified in the [Size].
Additionally, you can change the frame rate of the movie in [FPS].

Note:

- **Rendering may take longer due to the use of high-resolution settings during movie creation.**
- **Acto3D may pause if you navigate away or switch to another application during operation.**
- **Uncompressed TIFF files are created for every frame. Please delete them if not needed.**

Apply filter (Gaussian 3D filter)



In menu bar, click: **ImageProcess** > **Gaussian 3D**

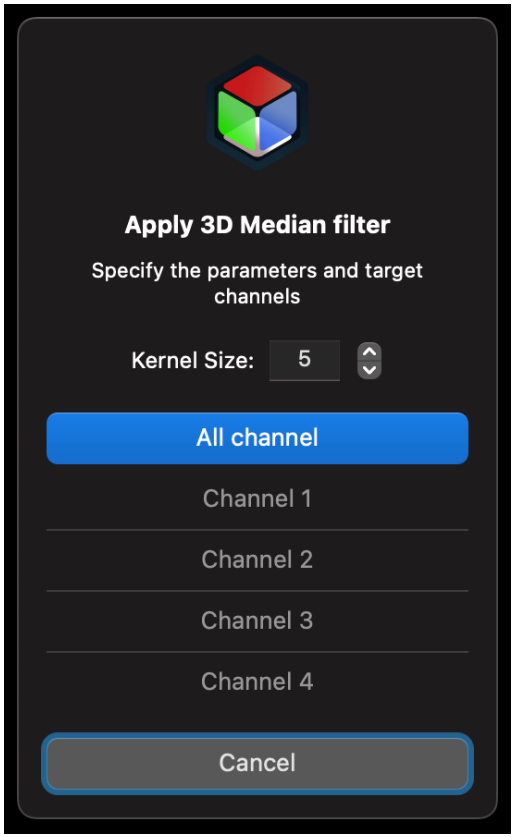
Specify the kernel size, sigma and target channels.
The kernel size **must be an odd integer**.

This process requires additional memory allocation.
When 'All' is selected, the size of the temporary buffer needed is 'width x height x depth x 4' bytes.
If a specific channel is selected, the size reduces to 'width x height x depth' bytes.

If there is insufficient memory available, the filter should be applied to each channel individually.

As the kernel size increases, the time required for the process also increases.

Apply filter (Median 3D filter)



In menu bar, click: **ImageProcess** > **Median 3D**

Specify the kernel size and target channels.
The kernel size **must be an odd integer**.

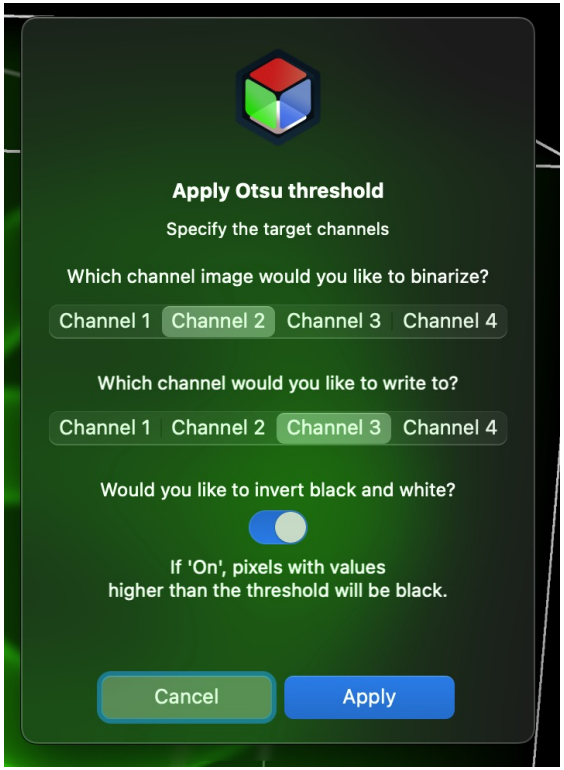
The median filter is time-consuming to process.

This process requires additional memory allocation.
When 'All' is selected, the size of the temporary buffer needed is 'width x height x depth x 4' bytes.
If a specific channel is selected, the size reduces to 'width x height x depth' bytes.

If there is insufficient memory available, the filter should be applied to each channel individually.

As the kernel size increases,
the time required for the process also significantly increases.

Apply filter (Binarization)



In menu bar, click: **ImageProcess** > **Binarization**

In Acto3D, you have the option to use either standard thresholding or Otsu's thresholding for binarization. Similar settings screens are displayed for both.

Please specify the channel to be binarized and the channel where the results will be stored. If you wish to invert black and white, check the corresponding box (Default is OFF, and higher pixel values are set to white).

Currently, there is no support for a preview before applying the binarization.

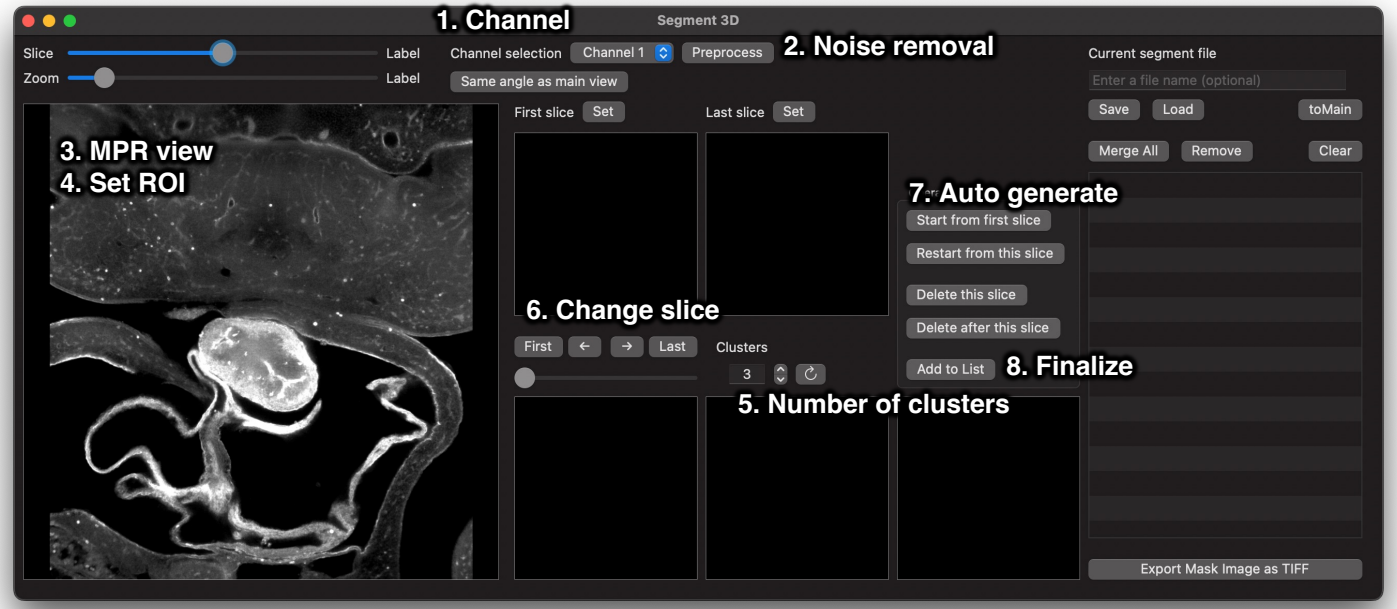
Segmentation

The application operates based on the “iterative” k-means approach as described in the paper.
For details on operation, please refer to Supplementary Video 3.

Load the Image, first

Navigate to the Segment tab and select **'3D segment**’.

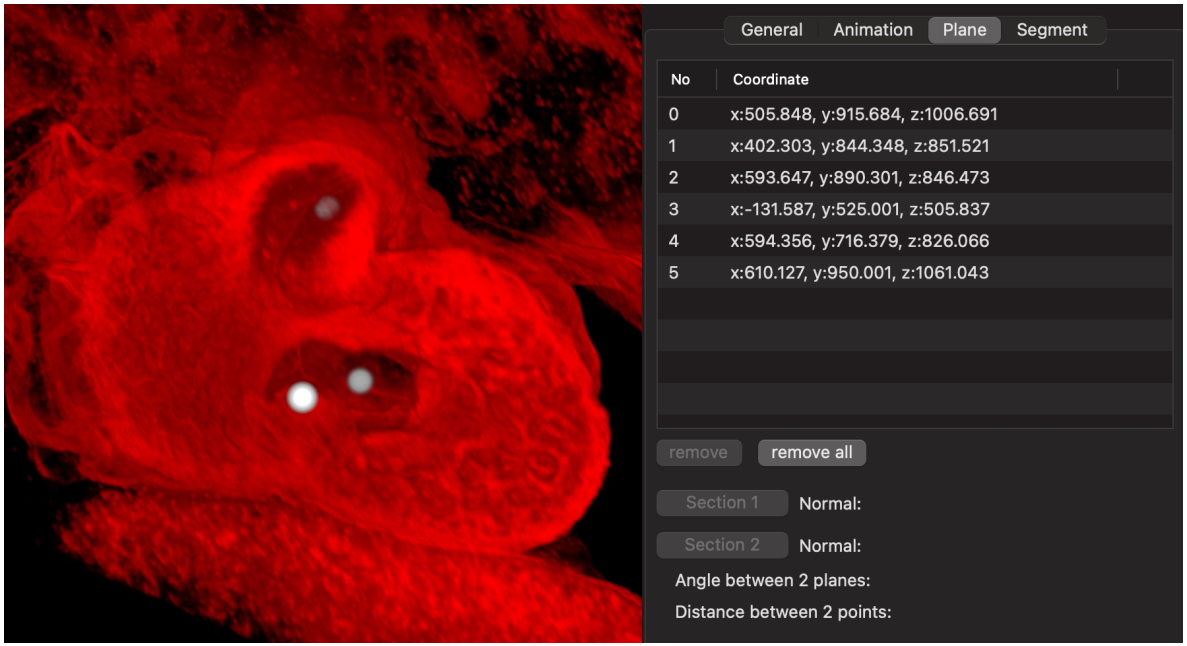
1. **Channel Selection**: Choose the desired channel you wish to work on.
2. **Noise Removal**:
   Click on the 'Preprocess' button to apply a 3D gaussian blur with
   kernel size 7x7x7 and sigma 1.8. If you wish to utilize different parameters,
   either perform the aforementioned process or use external applications like Fiji for noise removal.
3. **Manipulating MPR View**:
   The MPR view can be adjusted using the mouse. If the orientation in MPR is difficult,
   adjust to the desired angle in the main view and click '**Same angle as main view**' to match the rotation.

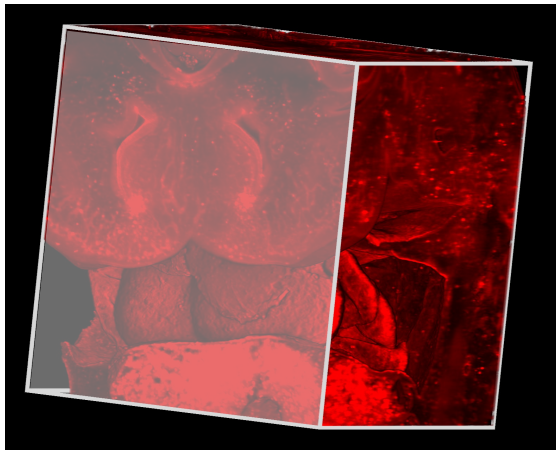


Segmentation

4. **Selecting the Region of Interest**:
   Press and **hold the Command (⌘) button** and use the mouse to highlight the desired region.
   Use the slider to adjust the region, then **set the first** and **last slices**.
5. **Clustering**:
   Set the number of clusters (default is 3). Click the **'Update**' button to execute clustering.
   If the clustering doesn't appear accurate, either repeat the update or adjust the number of clusters.
   Once satisfactory clustering is achieved, click on the desired region within the cluster image.
   This will display a mask image on the right side.
6. **Change slice**:
   Use the ← and → buttons to manually process each slice or click the 'Auto-process' button.
   During auto-processing, if a significant change is detected in the generated mask image,
   a confirmation dialog will be displayed. If everything is as desired, press 'Restart' to continue from the last slice.
   If you wish to modify the number of clusters midway, you can delete the clustering for
   subsequent slices and redo the process.
7. **Finalizing the Segmentation**:
   Once satisfied, click 'Add to List'.
   Due to the complex tubular structure of the pharyngeal arch artery, adjust the view angle to get
   a perpendicular cross-section and repeat the steps for each segment from multiple viewpoints.
   Click **'Merge All**' on the right to consolidate all segmentation data and create a mask image.
   This allows for a 3D observation of the segmented structure when reflected onto the main texture.

Point plotting





Even if the object is transparent and you can see through it, it still exists as image data. Therefore, if you don't click on the appropriate slice, the surface will be plotted.

By **left-clicking** within the 3D image, you can plot a specified coordinate.
However, if you click where the entire image is displayed, only the surface of the cube (not the 3D object) will be plotted.

Thus, it's **necessary to adjust to the appropriate slice before clicking**.
Tip: By pressing the Z key and switching to MPR view before clicking, you can plot at the correct location.

You can verify these in the Plane tab, where you can make cuts across planes passing through multiple plots, or calculate distances between points and angles between planes.

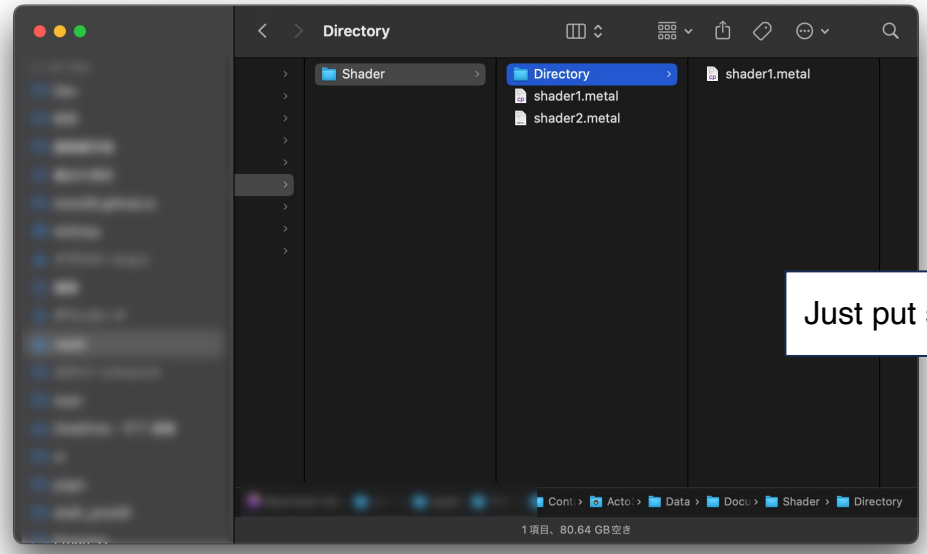
Custom shader – How to add a custom shader

In menu bar, click: **Shader** > **Open Shader Directory**

Place the **.metal** files written in Metal Shading Language (MSL) in this directory.
You can also create subdirectories if needed.

The **.metal** files located in this directory will be compiled
when Acto3D is launched or when the **Re-compile** button is pressed.

Caution: If there's an error in any of the .metal files,
Acto3D will disable all custom shaders and will only use the built-in default shaders.



Just put shader files in the Shader directory

Custom shader – How to create custom shaders

In menu bar, click: **Shader** > **Copy Sample Shaders (or Copy Template Shaders)** to copy sample or template shader files. Then, the directory will automatically open in Finder.

You can edit the shader using “template_FTB.metal” as a reference.
This file is structured as follows.

template_FTB.metal

```
// Label: Template for Front To Back Rendering
// Author: Naoki Takeshita
// Description: Template for standard front to back rendering
kernel void TEMPLATE_FTB(device RenderingArguments    &args       [[buffer(0)]],
                         uint2                        position    [[thread_position_in_grid]]){
    …
    …
}
```

These 3 lines are necessary to identify the shader file.

This function name must be unique

Please follow the rules provided below.

- The first three lines should be placed at the beginning of the shader file.
  These are necessary for Acto3D to recognize the shaders and display them in the menu.
- To avoid internal errors, please refrain from using special characters.
- The kernel function name must be unique and not duplicate any other shader files.

Custom shader – How to create custom shaders

template_FTB.metal

```
// Label: Template for Front To Back Rendering
// Author: Naoki Takeshita
// Description: Template for standard front to back rendering
kernel void TEMPLATE_FTB(device RenderingArguments    &args       [[buffer(0)]],
                         uint2                        position    [[thread_position_in_grid]]){
```

Geometrical calculation

This section is for coordinate calculation when moving the observation point, and **generally does not need to be changed**.

Codes for MPR (Macro)

In Acto3D, you can switch to MPR view at any time with the Z key.
It's a macro that calls our MPR code, and can be deleted if not needed.

```
for loop{
```

Bounding Box

Crop and Cutting plane

These codes are for drawing bounding boxes, cropping processes, and rendering planes during cropping. They can be removed if not needed.

Sampling and Compositing process
Apply transfer function
Apply shade

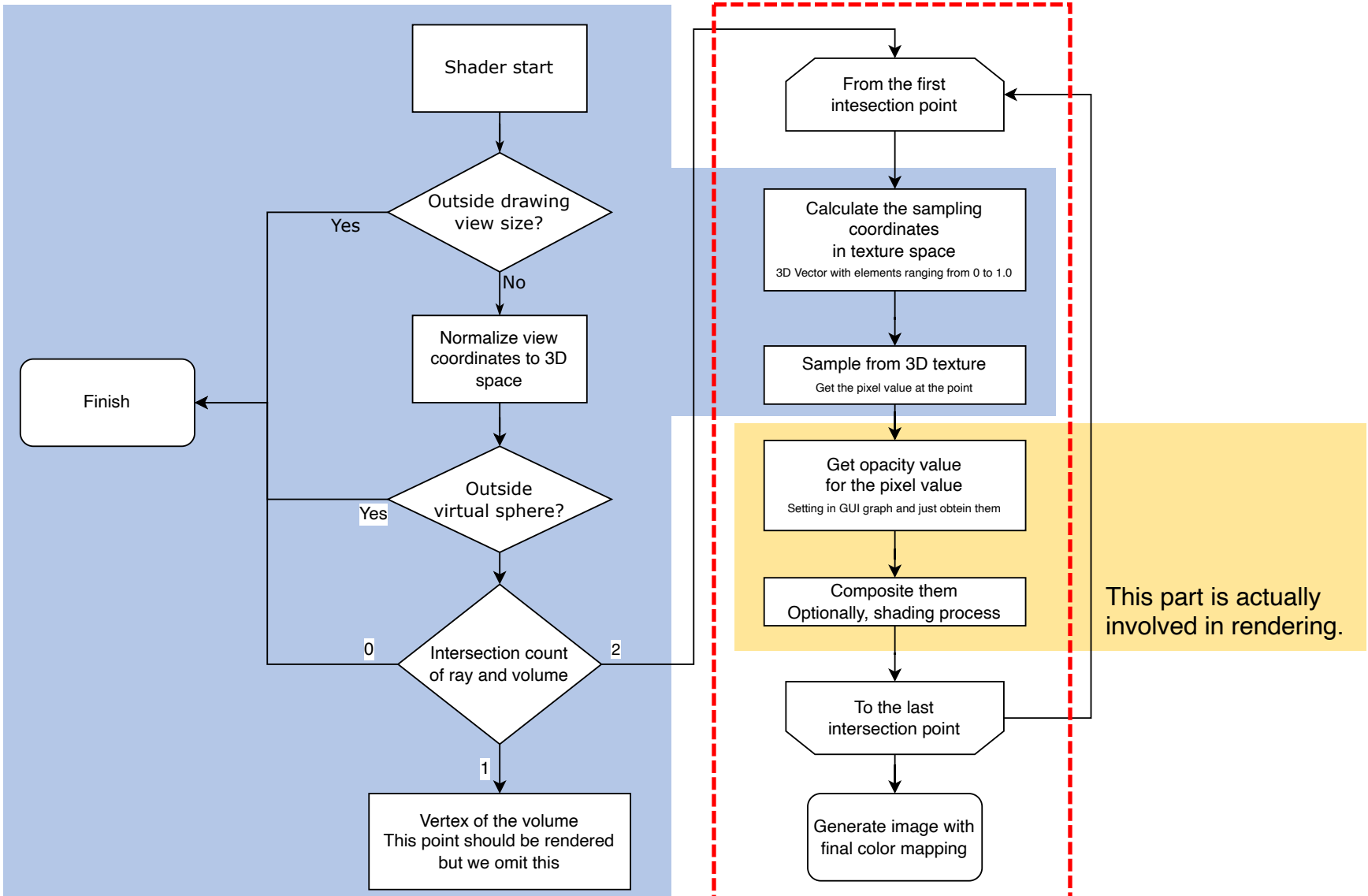
In this section, the process of obtaining and compositing the pixel value and opacity values at each coordinate is performed.
This will be explained in the following section.

```
}
```

This is the loop part that samples along the line of sight and processes the pixel values.

```
}
```

Essential Shader Workflow Diagram



The following section will explain the code on the next page.

The actual code for rendering part

```
for (……){
    Cin = Cout;
    Ain = Aout;

    Cvoxel = args.tex.sample(args.smp, texCoordinate);

    float4 intensityRatio = float4(modelParameter.intensityRatio[0],
                                   modelParameter.intensityRatio[1],
                                   modelParameter.intensityRatio[2],
                                   modelParameter.intensityRatio[3]);

    Cvoxel *= intensityRatio;

    float4 alpha = float4(pow(args.tone1[int(clamp(Cvoxel[0] * 2550.0f, 0.0f, 2550.0f))] ,modelParameter.alphaPower),
                          pow(args.tone2[int(clamp(Cvoxel[1] * 2550.0f, 0.0f, 2550.0f))] ,modelParameter.alphaPower),
                          pow(args.tone3[int(clamp(Cvoxel[2] * 2550.0f, 0.0f, 2550.0f))] ,modelParameter.alphaPower),
                          pow(args.tone4[int(clamp(Cvoxel[3] * 2550.0f, 0.0f, 2550.0f))] ,modelParameter.alphaPower));

    float4 alphaMax = max(max(alpha[0], alpha[1]) , max(alpha[2], alpha[3]));

    float4 light_intensity = modelParameter.light;
    Cvoxel *= light_intensity;

    Cout = Cin + Cvoxel * (1.0f - Ain) * alphaMax;
    Aout = Ain + (1.0f - Ain) * alphaMax;

    float opacityThreshold = 0.99;
    if(max(max(Aout.x, Aout.y), max(Aout.z, Aout.a)) > opacityThreshold){
        break;
    }
}


float3 lut_c1 = Cout[0] * modelParameter.color.ch1.rgb;
float3 lut_c2 = Cout[1] * modelParameter.color.ch2.rgb;
float3 lut_c3 = Cout[2] * modelParameter.color.ch3.rgb;
float3 lut_c4 = Cout[3] * modelParameter.color.ch4.rgb;

float cR = max(max(lut_c1[0], lut_c2[0]), max(lut_c3[0], lut_c4[0]));
float cG = max(max(lut_c1[1], lut_c2[1]), max(lut_c3[1], lut_c4[1]));
float cB = max(max(lut_c1[2], lut_c2[2]), max(lut_c3[2], lut_c4[2]));

args.outputData[index + 0] = uint8_t(clamp(cR, 0.0f, 1.0f) * 255.0);
args.outputData[index + 1] = uint8_t(clamp(cG, 0.0f, 1.0f) * 255.0);
args.outputData[index + 2] = uint8_t(clamp(cB, 0.0f, 1.0f) * 255.0);

return;
```

Shader Code Explained: Core Concepts

```
//
// Begin main sampling and compositing process.
//

Cin = Cout;
Ain = Aout;
```

In front-to-back rendering, the Cin = Cout; and Ain = Aout; lines are used to carry forward the color (Cout) and alpha (Aout) values from the previous voxel to the next one in the rendering sequence.

```
// Cvoxel is of type float4, accessed like Cvoxel[0] or Cvoxel[3].
// It stores pixel values at the sampled coordinates as float values ranging from 0 to 1.0.
// Depending on the number of channels in the image, values for each channel are stored in [0]-[3].
// Attempting to access a non-existent channel does not result in an error, but returns zero.
// Cvoxel can be accessed using [0]-[3], as well as .r, .g, .b, .a, or .x, .y, .z, .w.
Cvoxel = args.tex.sample(args.smp, texCoordinate);


// The intensityRatio stores the brightness levels for each channel as specified in the GUI, in float format.
// The default value is 0.
float4 intensityRatio = float4(modelParameter.intensityRatio[0],
                               modelParameter.intensityRatio[1],
                               modelParameter.intensityRatio[2],
                               modelParameter.intensityRatio[3]);

// Multiply the color by its intensity.
Cvoxel *= intensityRatio;
```

Retrieve the values from each channel's intensity slider in the GUI to adjust the color of each channel.

Shader Code Explained: Core Concepts

```
        // Get opacity for the voxel

        // Although the texture is 8-bit with pixel values ranging from 0-255 for which we define opacity,
        // we're transferring to the GPU a transfer function with ten times the precision (to the first decimal place)
for smoother results.
        // Hence, calculations are performed in the range of 0-2550 instead of 0-255.


        // When using the pixel value (C) that has considered the above intensity,
        // there are cases where the luminance value may exceed 2550, so it is clamped to 2550.
        // (No need to consider this if fetching opacity before multiplying pixel value with intensity.)

        // args.tone1 to args.tone4 store opacity for each channel's pixel values as floats in the range 0 to 1.0.
        // For example, to access the opacity for a pixel value of 128,
        //   use args.tone1[1280].

        // Here, the pow function is used to exponentiate the opacity from 0 to 1.0.
        // modelParameter.alphaPower, specified in the GUI and defaulting to 2, squares the opacity.
        float4 alpha = float4(pow(args.tone1[int(clamp(Cvoxel.r * 2550.0f, 0.0f, 2550.0f))] ,modelParameter.alphaPower),
                              pow(args.tone2[int(clamp(Cvoxel.g * 2550.0f, 0.0f, 2550.0f))] ,modelParameter.alphaPower),
                              pow(args.tone3[int(clamp(Cvoxel.b * 2550.0f, 0.0f, 2550.0f))] ,modelParameter.alphaPower),
                              pow(args.tone4[int(clamp(Cvoxel.a * 2550.0f, 0.0f, 2550.0f))] ,modelParameter.alphaPower));
```

Cvoxel is a float4 type, holding four elements as floats which may exceed 1.0 due to earlier intensity multiplication. Thus, we ensure values are within the 0-2550 range using the clamp function, retrieve the corresponding alpha, and then raise it to the power specified by the alphaPower in the GUI.

Shader Code Explained: Core Concepts

```
// While different opacities are defined for the four channels,
// applying the same transparency to a specific voxel ensures the depth is rendered accurately.
// If you wish to apply distinct transparencies for each channel, the following section is unnecessary.
float4 alphaMax = max(max(alpha.r, alpha.g) , max(alpha.b, alpha.a));
```

Although individual opacities are assigned to each of the four channels, the highest alpha value is ultimately selected. By determining a single level of transparency for each voxel, we ensure the accuracy of depth perception in the rendered image.

```
float4 light_intensity = modelParameter.light;
Cvoxel *= light_intensity;
```

The value for 'light' set in the GUI is further multiplied to adjust the brightness.

```
Cout = Cin + Cvoxel * (1.0f - Ain) * alphaMax;
Aout = Ain + (1.0f - Ain) * alphaMax;
```

Finally, apply the rendering equation, substituting the values accordingly.

$$C_{\text{out}} = C_{\text{in}} + (1 - \alpha_{\text{in}})\alpha C$$

$$\alpha_{\text{out}} = \alpha_{\text{in}} + (1 - \alpha_{\text{in}})\alpha$$

Shader Code Explained: Core Concepts

```
// Early termination for front-to-back rendering
// If the accumulated opacity surpasses a certain threshold, further processing can be skipped.

// Setting a higher opacityThreshold results in an image closer to back-to-front rendering,
// but at the cost of increased computation.
// A realistic range for the threshold is between 0.9 and 0.99.
float opacityThreshold = 0.99;
if(max(max(Aout.x, Aout.y), max(Aout.z, Aout.a)) > opacityThreshold){
    break;
}
```

In front-to-back rendering, if an opaque voxel is encountered along the line of sight, it obscures anything behind it, allowing rendering to cease. The threshold at which this occurs is defined here, with a realistic range being between 0.9 and 0.99.

Shader Code Explained: Core Concepts

```
float3 lut_c1 = Cout.r * modelParameter.color.ch1.rgb;
float3 lut_c2 = Cout.g * modelParameter.color.ch2.rgb;
float3 lut_c3 = Cout.b * modelParameter.color.ch3.rgb;
float3 lut_c4 = Cout.a * modelParameter.color.ch4.rgb;

float cR = max(max(lut_c1.r, lut_c2.r), max(lut_c3.r, lut_c4.r));
float cG = max(max(lut_c1.g, lut_c2.g), max(lut_c3.g, lut_c4.g));
float cB = max(max(lut_c1.b, lut_c2.b), max(lut_c3.b, lut_c4.b));

args.outputData[index + 0] = uint8_t(clamp(cR, 0.0f, 1.0f) * 255.0);
args.outputData[index + 1] = uint8_t(clamp(cG, 0.0f, 1.0f) * 255.0);
args.outputData[index + 2] = uint8_t(clamp(cB, 0.0f, 1.0f) * 255.0);

return;
```

Upon completing the color accumulation within the loop, the final step is to assign colors to each channel

.

The variables lut_c1 to lut_c4 represent the color hues for each channel in RGB. For each RGB element across all channels, cR, cG, and cB perform a maximum comparison to find the highest value.

Finally, the values of cR, cG, and cB are clamped within the range of 0 to 1.0, multiplied by 255, and written to the corresponding coordinates in args.outputData.

User Available Arguments

```
struct RenderingArguments {
    texture3d<float, access::sample> tex          [[id(0)]];
    constant RenderingParameters &params          [[id(1)]];
    device uint8_t *outputData                    [[id(2)]];
    device float* tone1                           [[id(3)]];
    device float* tone2                           [[id(4)]];
    device float* tone3                           [[id(5)]];
    device float* tone4                           [[id(6)]];
    constant uint16_t &flags                      [[id(7)]];
    constant float4 &quaternions                  [[id(8)]];
    constant uint16_t &targetViewSize             [[id(9)]];
    sampler smp                                   [[id(10)]];
    constant uint16_t &pointSetCount              [[id(11)]];
    constant uint16_t &pointSelectedIndex         [[id(12)]];
    constant float3* pointSet                     [[id(13)]];
};
```

```
struct RenderingParameters{
    float scale ;
    float zScale ;
    uint16_t sliceNo;
    uint16_t sliceMax;
    float trimX_min;
    float trimX_max;
    float trimY_min;
    float trimY_max;
    float trimZ_min;
    float trimZ_max;
    PackedColor color ;
    float4 cropLockQuaternions ;
    uint16_t cropSliceNo ;
    float eularX ;
    float eularY ;
    float eularZ ;
    float translationX ;
    float translationY ;
    uint16_t viewSize;
    float pointX ;
    float pointY ;
    uint8_t alphaPower;
    float renderingStep ;
    float4 intensityRatio;
    float light;
    float shade;

    float3 backgroundColor;
};
```

Access the RenderingArguments structure in your code with the args prefix.
For accessing members of the RenderingParameters structure, use the args.params prefix.

e.g., `args.outputData[]`
`args.pointSetCount`
`args.params.sliceNo`

Shader Code Explained: Additional rendering (Shade)

```
if(flags & (1 << SHADE)){
    // Very simple light and shade

    float eps = 2;
    float3 gradient_diff[3] = {
        float3(1.0 / width, 0, 0) * eps,
        float3(0, 1.0 / height, 0) * eps,
        float3(0, 0, 1.0 / depth) * eps
    };

    // Calculate the gradient
    float4 gradient_x = Cvoxel - intensityRatio * args.tex.sample(args.smp, texCoordinate - gradient_diff[0]);
    float4 gradient_y = Cvoxel - intensityRatio * args.tex.sample(args.smp, texCoordinate - gradient_diff[1]);
    float4 gradient_z = Cvoxel - intensityRatio * args.tex.sample(args.smp, texCoordinate - gradient_diff[2]);

    float3 grad_0 = float3(gradient_x[0], gradient_y[0], gradient_z[0]);
    float3 grad_1 = float3(gradient_x[1], gradient_y[1], gradient_z[1]);
    float3 grad_2 = float3(gradient_x[2], gradient_y[2], gradient_z[2]);
    float3 grad_3 = float3(gradient_x[3], gradient_y[3], gradient_z[3]);

    float diffuse_ratio = modelParameter.shade;

    // The vector used for shading calculations is currently fixed at (1,1,0).
    float diffuse0 = diffuse_ratio * max(0.0f, dot(normalize(grad_0), normalize(float3(1,1,0))));
    float diffuse1 = diffuse_ratio * max(0.0f, dot(normalize(grad_1), normalize(float3(1,1,0))));
    float diffuse2 = diffuse_ratio * max(0.0f, dot(normalize(grad_2), normalize(float3(1,1,0))));
    float diffuse3 = diffuse_ratio * max(0.0f, dot(normalize(grad_3), normalize(float3(1,1,0))));

    light_intensity = float4(
                            max(0.0f, light_intensity[0] - diffuse0),
                            max(0.0f, light_intensity[1] - diffuse1),
                            max(0.0f, light_intensity[2] - diffuse2),
                            max(0.0f, light_intensity[3] - diffuse3)
                            );

    Cvoxel *= light_intensity;

}
```

The code calculates the gradient at the current sampling point by taking the difference in color at positions offset by eps pixels along the X, Y, and Z axes. It then computes the dot product of this gradient vector with a fixed vector, (1,1,0), to create a pseudo-shading effect, which simulates light and shadow based on the shape and contours of the volume being rendered.

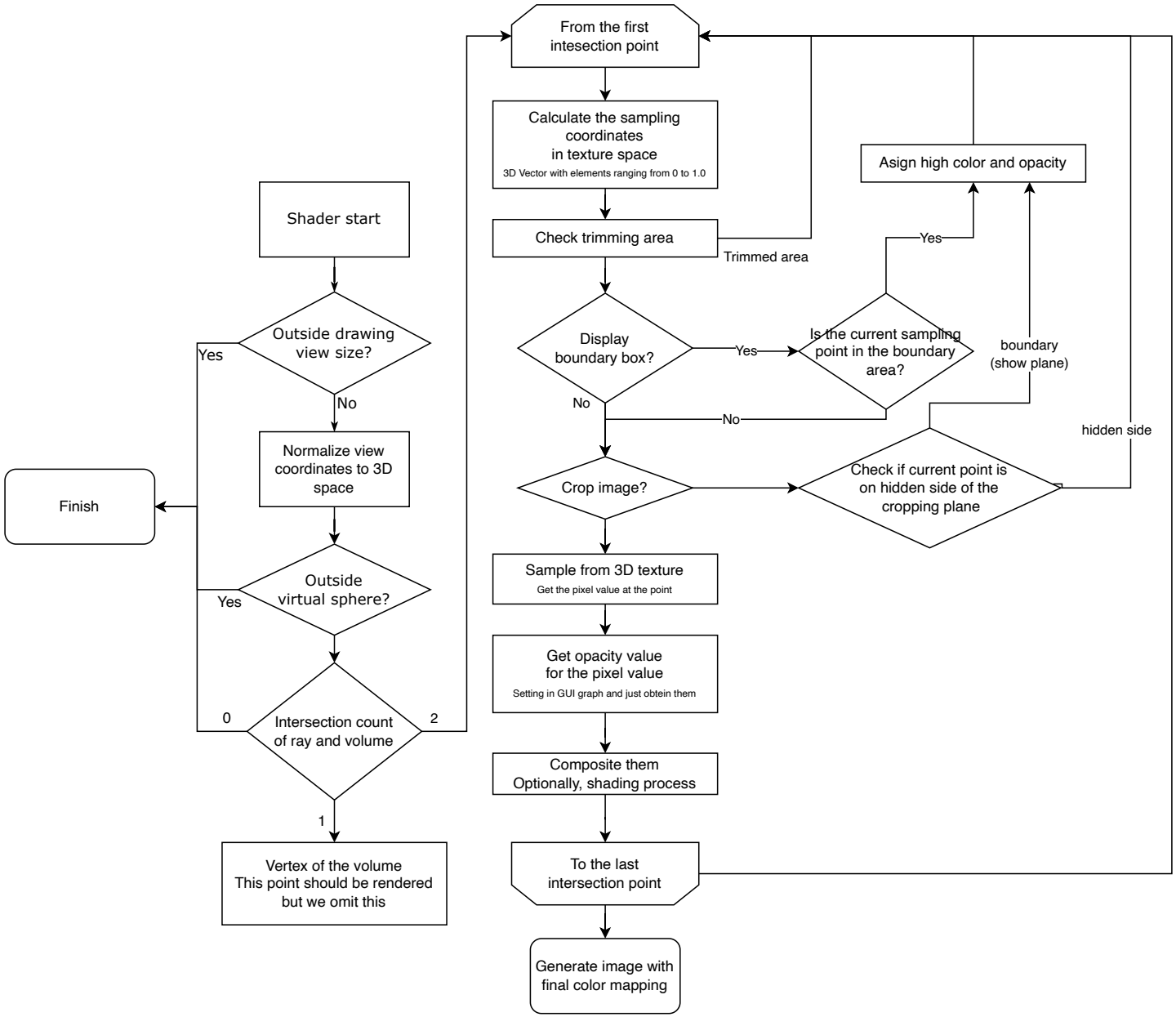
(commonly referred to as "diffuse shading")

Shader Code Explained: Additional rendering (Ball 3D plot)

```
// Render a small sphere at the specified coordinate if it's marked within the space.
// Note: Processing speed may be affected if there are many registered coordinates.

// Sphere radius definition
float ballRadius = 20.0f;

for (uint8_t p=0; p<pointSetCount; p++){
    Cin = Cout;
    Ain = Aout;

    float3 _vec = coordinatePos.xyz - pointSet[p];
    float _length = length(_vec);

    if(_length < ballRadius){
        float _ballColor = pow(1.0f - (_length / ballRadius) * 0.3, 2);
        float _ballAlpha = 1.0f - (_length / ballRadius);

        // For more soft rendering result, use the following code.
        // float _ballAlpha = pow(1.0f - (_length / ballRadius), 2);

        if(p == pointSelectedIndex){
            // Brighter for currently selected point
            _ballColor += 0.25;
        }

        Cout = Cin + (1.0 - Ain) * _ballColor * _ballAlpha;
        Aout = Ain + (1.0 - Ain) * _ballAlpha;
    }
}
```

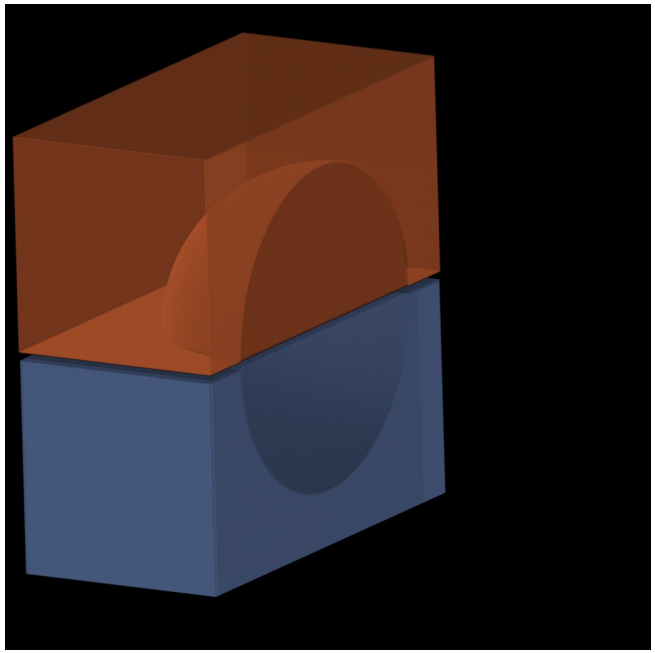
The provided code snippet appears to render small spheres at coordinates stored in a pointSet array, which holds float3 type coordinates. The number of elements in the pointSet array is stored in pointSetCount since the count of elements cannot be obtained directly in Metal. For each coordinate in pointSet, the code checks if it lies within the radius of a sphere defined by ballRadius. If the coordinate is within the sphere, it assigns pixel values proportional to the radius, decreasing from the sphere's surface to its center.

Detail Diagram of Our Shaders



Sample (sample_ExtractValues.metal)

```
// If you wish to perform operations on specific pixel values, you can add your processing logic here.
// Remember, Cvoxel values are floats ranging from 0 to 1.
// If you need to work with values ranging from 0 to 255, multiply by 255 as shown in the following code:
if(Cvoxel[0] * 255 > 20 && Cvoxel[0] * 255 < 190){

}else{
    Cvoxel[0] = 0;
}

if(Cvoxel[2] * 255 > 20 && Cvoxel[2] * 255 < 190){
    Cvoxel[2] = 0;
}else{

}
```



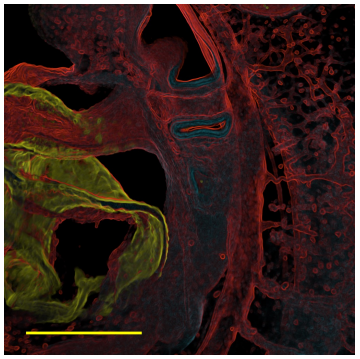
During the sampling step, direct manipulation of the Cvoxel values allows for influencing specific color tones.

Sample (sample_ConsiderGradients.metal)

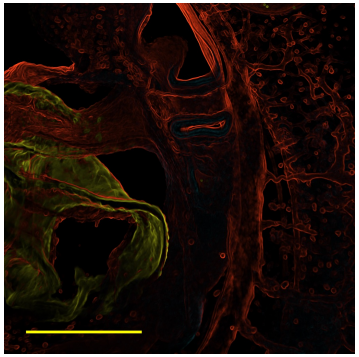
As mentioned in the main text of the paper, the shader has been modified to replace the pixel values at the sampling points with the difference in luminance from their surroundings. This change ensures areas with significant color tone variations appear brighter.
Please adjust the brightness in conjunction with the light parameter on the GUI.

```
// Very simple light and shade

 float eps = 1.0;
 float3 gradient_diff[3] = {
     float3(1.0 / width, 0, 0) * eps,
     float3(0, 1.0 / height, 0)* eps,
     float3(0, 0, 1.0 / depth)* eps
 };

 // Calculate the gradient
 float4 gradient_x = args.tex.sample(args.smp, texCoordinate + gradient_diff[0])
                       -  args.tex.sample(args.smp, texCoordinate - gradient_diff[0]);
 float4 gradient_y = args.tex.sample(args.smp, texCoordinate + gradient_diff[1])
                       -  args.tex.sample(args.smp, texCoordinate - gradient_diff[1]);
 float4 gradient_z = args.tex.sample(args.smp, texCoordinate + gradient_diff[2])
                       -  args.tex.sample(args.smp, texCoordinate - gradient_diff[2]);

 float3 grad_0 = float3(gradient_x[0], gradient_y[0], gradient_z[0]);
 float3 grad_1 = float3(gradient_x[1], gradient_y[1], gradient_z[1]);
 float3 grad_2 = float3(gradient_x[2], gradient_y[2], gradient_z[2]);
 float3 grad_3 = float3(gradient_x[3], gradient_y[3], gradient_z[3]);

 float v0 = pow(length(grad_0), 1);
 float v1 = pow(length(grad_1), 1);
 float v2 = pow(length(grad_2), 1);
 float v3 = pow(length(grad_3), 1);

 // To emphasize the boundaries more,
 // increase the second argument of the `pow` function
 // to raise the gradient length to a higher power.
 // float v0 = pow(length(grad_0), 2);
 // float v1 = pow(length(grad_1), 2);
 // float v2 = pow(length(grad_2), 2);
 // float v3 = pow(length(grad_3), 2);
```

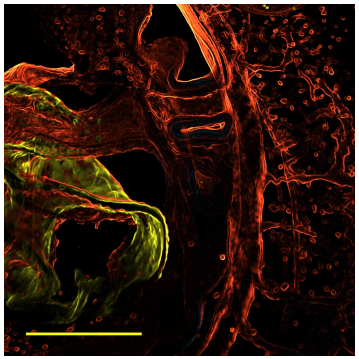
power: 1



power: 2



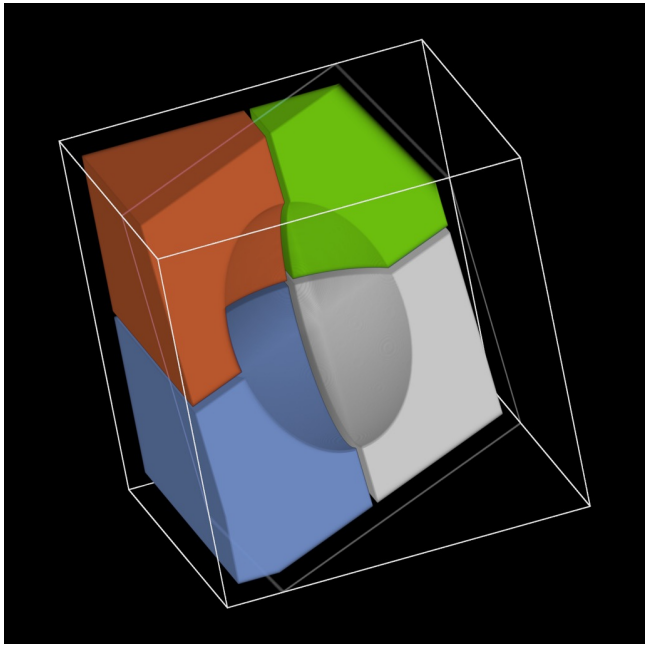
power: 3



Sample (sample_BoundaryHighlight.metal)

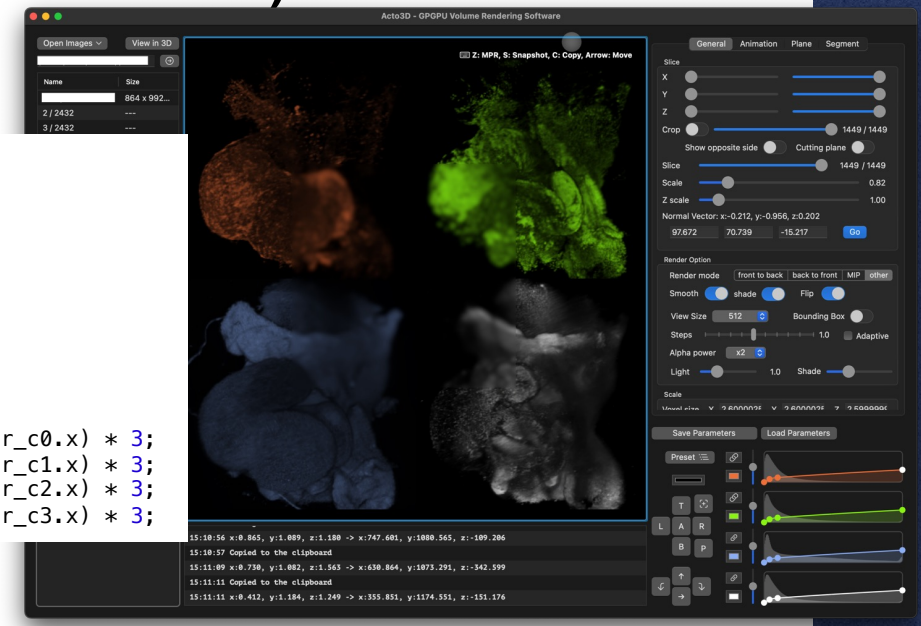
This modification does not display a plane on the cropped surfaces but instead only shows the boundary lines.

```
if ((texCoordinate.x < boundaryWidth ||
     texCoordinate.y < boundaryWidth ||
     texCoordinate.z < boundaryWidth ||
     (1.0 - texCoordinate.x) < boundaryWidth ||
     (1.0 - texCoordinate.y) < boundaryWidth ||
     (1.0 - texCoordinate.z) < boundaryWidth ))
{
    if(abs(t_crop) <= threshold_plane_thickness){
        // Draw the boundary if the current texture coordinates are located on any face of the volume.
        Cin = Cout;
        Ain = Aout;

        Cvoxel = float4(0.85, 0.85, 0.85, 0.85);
        float Alpha = 0.1;

        Cout = Cin + Cvoxel * (1.0 - Ain) * Alpha;
        Aout = Ain + (1.0 - Ain) * Alpha;

        continue;
    }
}
```

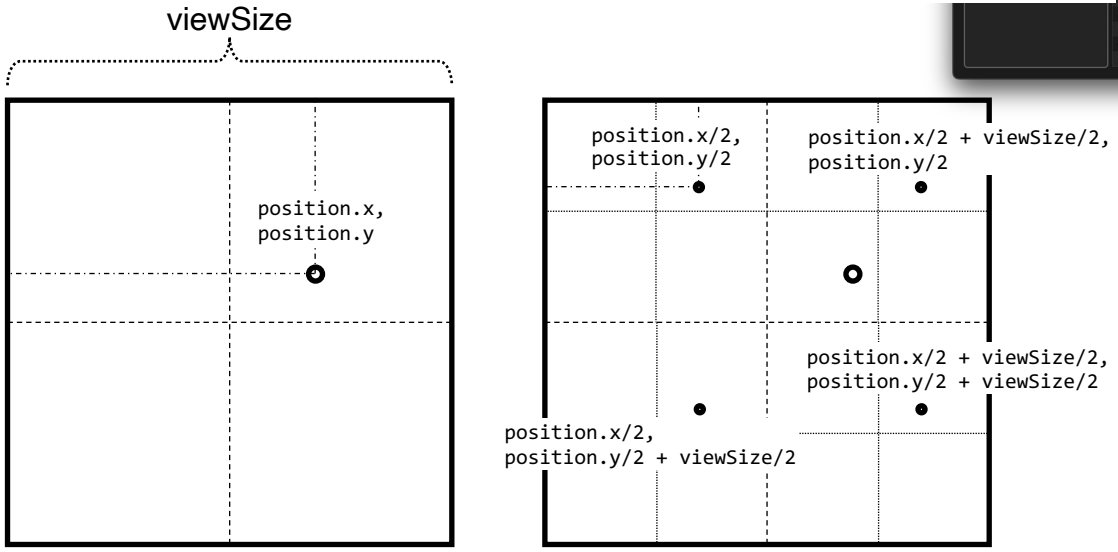


Sample (sample_QuadChannelRenderer.metal)

With this modification, it is possible to render the 4 channels separately and display them simultaneously.




```
// index for output texture (output texture is 24 bits RGB image)
uint index = (position.y * viewSize + position.x) * 3;


uint halfViewSize = viewSize / 2;
uint2 scaledPosition = uint2(float2(position) / 2.0f);
uint2 scaledPosition_for_c0 = scaledPosition;
uint2 scaledPosition_for_c1 = scaledPosition + uint2(halfViewSize, 0);
uint2 scaledPosition_for_c2 = scaledPosition + uint2(0, halfViewSize);
uint2 scaledPosition_for_c3 = scaledPosition + halfViewSize;

uint index_0 = (scaledPosition_for_c0.y * viewSize + scaledPosition_for_c0.x) * 3;
uint index_1 = (scaledPosition_for_c1.y * viewSize + scaledPosition_for_c1.x) * 3;
uint index_2 = (scaledPosition_for_c2.y * viewSize + scaledPosition_for_c2.x) * 3;
uint index_3 = (scaledPosition_for_c3.y * viewSize + scaledPosition_for_c3.x) * 3;
```



This shader modifies the 'Geometrical calculation' area. It transforms the coordinates for displaying the original image into four coordinates on a quartered grid and assigns individual rendering results to those coordinates.

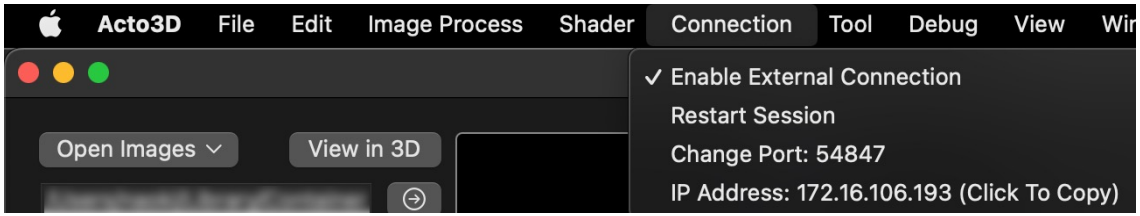
```
for(int c=0; c<3; c++){
    args.outputData[index_0 + c] = uint8_t(clamp(lut_c1[c] * 255.0f, 0.0f, 255.0f));
    args.outputData[index_1 + c] = uint8_t(clamp(lut_c2[c] * 255.0f, 0.0f, 255.0f));
    args.outputData[index_2 + c] = uint8_t(clamp(lut_c3[c] * 255.0f, 0.0f, 255.0f));
    args.outputData[index_3 + c] = uint8_t(clamp(lut_c4[c] * 255.0f, 0.0f, 255.0f));
}
```

Modification of the block that assigns the final color.

External connection

Acto3D_py is available in https://github.com/Acto3D/Acto3D_py

When using this package, be aware that TCP-based data transmission is **turned off** by default.
To enable this feature, navigate through the menu: **Connection** > **Enable external connection**



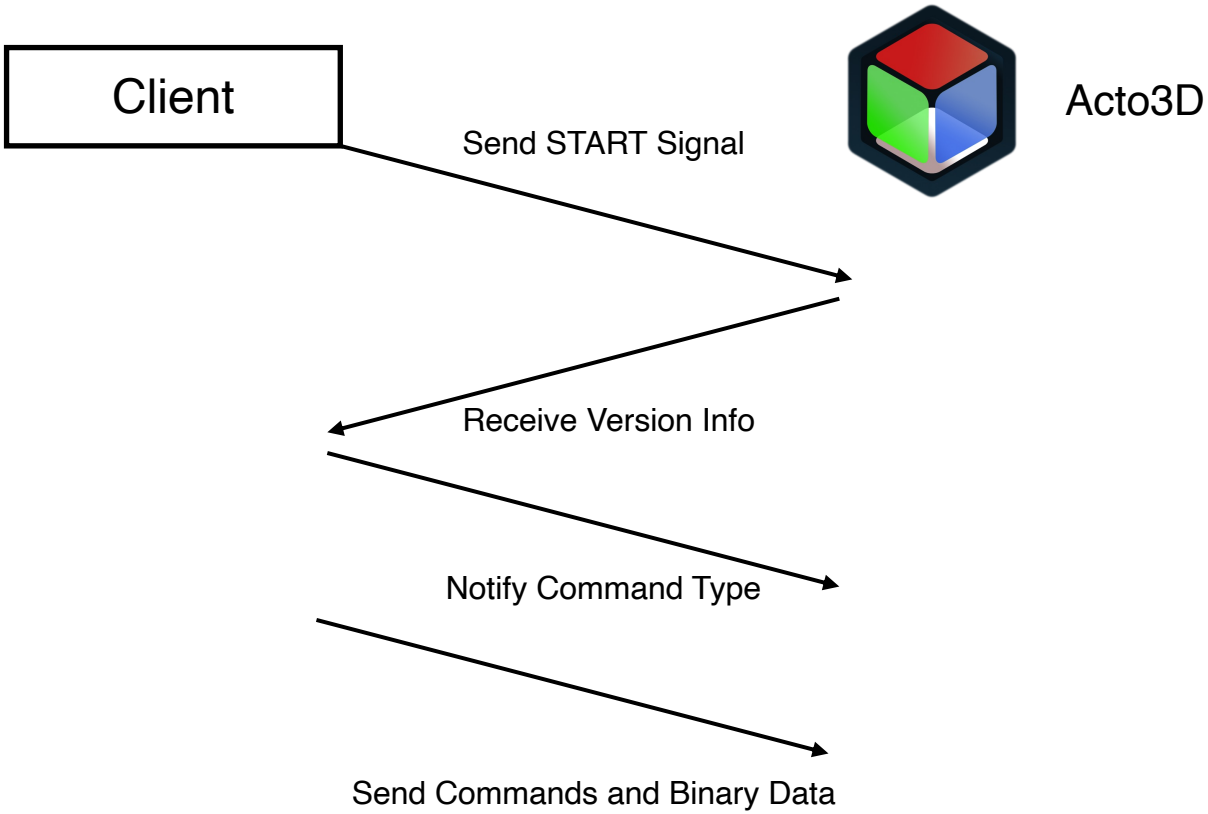
Please note the following important points regarding this functionality:

- Data transmission is not encrypted. This functionality is primarily intended for operations within your local PC. If you use this feature, keep in mind that data sent or received is not protected.

- Performance considerations for remote transmissions: Sending and receiving data remotely is not optimized for parallel processing and may result in slower performance from remote PCs.
- However, we have plans in place to address this issue and improve performance in future updates.

- Avoid exposing ports publicly. Given that the data transfer is not encrypted, exposing your ports to the external network could pose significant security risks. Do not make your ports publicly accessible.

Data Transfer Protocol



Acto3D_py automats this process in python.

Install

```
pip install git+https://github.com/Acto3D/Acto3D_py.git
```

Usage

```
import tifffile
import numpy as np
import Acto3D as a3d

# Load multichannel, zstack tif.
image = tifffile.imread('./image.tif’)

# `image` is object of numpy.ndarray
# type(image)
# numpy.ndarray

image.shape
# (664, 3, 1344, 1344)
# In this case, channel order is ZCYX.

# Transfer image data to Acto3D.
# Specify the dimension order correctly.
a3d.transferImage(image, order='ZCYX’)

# Also you can set display ranges as [[min, max], ...].
a3d.transferImage(image, order='ZCYX’,
                  display_ranges=[[500,2000],[500,2000],[500,2000]])

# Set voxel size for isotropic view
a3d.setVoxelSize(1.4, 1.4, 3.2, 'micron’)

# Set zoom value
a3d.setScale(1.5)

# Set slice no a3d.setSlice(450)

# Get the current image
current_image = a3d.getCurrentSlice()
```