pySSV Documentation

Release 0.7.1

Thomas Mathieson

Feb 12, 2024

INSTALLATION AND USAGE

Version: 0.7.1

Leverage the power of shaders for scientific visualisation in Jupyter

CHAPTER

ONE

CONTENTS

1.1 Introduction

pySSV makes it easy to use shaders in Jupyter notebooks.

To get started, create a suitable python environment (optional if you already have a python environment with jupyterlab setup), and install the package using `pip`:

```
conda create -n pySSV -c conda-forge python jupyterlab
conda activate pySSV

pip install pySSV
```

Download the example notebook from github `introduction.ipynb`

The example notebook can also be viewed here (*Introduction*).

Start JupyterLab and play around with the notebook:

```
jupyter lab .\introduction.ipynb
```

1.2 Built In Shader Templates

To reduce boilerplate code, *pySSV* includes a selection of shader templates which generate any platform/compiler specific code needed for the shader. These templates should handle most needs for scientific visualisation, but if you want to write your own shader templates to add new functionality or simplify your workflow refer to *Writing Shader Templates* for details on writing shader templates.

1.2.1 Built In Shader Uniforms

All built in shader templates include the `global_uniforms.glsl` library which defines a variety of useful shader uniforms which are set automatically by *pySSV*. It also includes any automatically defined uniforms (such as textures and render buffer textures).

float **`uTime`**

The current time since the shader was started in seconds.

int **`uFrame`**

The current frame number of the shader.

vec4 **uResolution**

The resolution of the render buffer this shader is rendering to.

vec2 **uMouse**

The current mouse position in pixel coordinates.

bool **uMouseDown**

Whether a mouse button is pressed.

mat4x4 **uViewMat**

The view matrix for the `SSVCanvas`'s main camera.

mat4x4 **uProjMat**

The projection matrix for the `SSVCanvas`'s main camera.

vec3 **uViewDir**

The view direction for the `SSVCanvas`'s main camera.

If `SHADERTOY_COMPAT` is `#define` before importing the `global_uniforms.glsl` file (which is the case in the Shadertoy template) then the following uniforms are also defined as aliases of the above uniforms.

iTime

= `uTime`

iFrame

= `uFrame`

iResolution

= `uResolution`

iMouse

= `vec4(uMouse, uMouse*(uMouseDown?1.:-1.))`

This doesn't quite match the implementation of shadertoy, but it's close enough for many shaders.

_DYNAMIC_UNIFORMS

This macro is defined automatically by *pySSV* and expands to include the declarations of all automatically declared uniforms, such as user-defined textures and render buffer textures.

1.2.2 Pixel Shader Template

```
#pragma SSV pixel
```

This template exposes a single entrypoint to a pixel shader.

Entrypoint Signature

vec4 **entrypoint**(vec2 fragPos)

Parameters

- **fragPos** – the position of the pixel being processed by this shader invocation in pixel coordinates.

Returns

the pixel's colour.

Template Arguments

entrypoint

positional

type: `str`

The name of the entrypoint function to pixel the shader.

--z_value

default: `0.999`

type: `float`

The constant value to write into the depth buffer. 0 is close to the camera, 1 is far away.

Example

```
#pragma SSV pixel frag
// The entrypoint to the fragment shader
vec4 frag(vec2 fragPos)
{
    vec2 uv = fragPos.xy / uResolution.xy;

    return mix(uv.xyx, uv.yyx, sin(uTime)*0.5+0.5);
}
```

1.2.3 Vertex Shader Template

`#pragma SSV vert`

This template exposes a single entrypoint to a vertex shader.

Entrypoint Signature

VertexOutput **mainVert**()

Returns
a `VertexOutput` struct containing the transformed vertex data.

struct **VertexOutput**

vec4 **position**

vec4 **color**

The shader is expected to take input from the following vertex attributes:

vec4 **in_vert**

vec4 **in_color**

Template Arguments

entrypoint

positional

type: `str`

The name of the entrypoint function to vertex the shader.

Example

```
#pragma SSV vert mainVert
VertexOutput mainVert()
{
    VertexOutput o;
    vec4 pos = vec4(in_vert, 1., 1.0);
    pos = uViewMat * pos;
    pos = uProjMat * pos;
    o.position = pos;
    o.color = vec4(in_color, 1.);
    return o;
}
```

1.2.4 ShaderToy Template

`#pragma SSV shadertoy`

This template exposes a single entrypoint to a pixel shader. It's designed to mimic the API of ShaderToy

Entrypoint Signature

void **mainImage**(vec4 fragColor, vec2 fragCoord)

Parameters

- **fragColor** (`out`) – the pixel's final colour.
- **fragPos** – the position of the pixel being processed by this shader invocation in pixel coordinates.

Template Arguments

None

Example

```
#pragma SSV shadertoy
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/iResolution.yy;
    // Colour changing over time
    vec3 col = sin(uv.xyx + iTime * vec3(3, 4, 5)) * 0.5 + 0.5;
    // Output to screen
    fragColor = vec4(vec3(col), 1.);
}
```

1.2.5 Vertex/Pixel Shader Template

`#pragma SSV vert_pixel`

This template exposes an entrypoint to a vertex shader and an entrypoint to a pixel shader.

Entrypoint Signature

Vertex Stage

void **mainVert**()

Returns
a `VertexOutput` struct containing the transformed vertex data.

The shader is expected to take input from the following vertex attributes:

vec4 **in_vert**

vec4 **in_color**

The shader is expected to take write to the following vertex attributes:

vec3 **color**

vec4 **gl_Position**

Pixel Stage

vec4 **mainPixel**(vec3 position)

Parameters

- **position** – the position written to gl_Position by the vertex shader.

Returns
the final colour of the fragment.

The shader is expected to take input from the following interpolated vertex attributes:

vec4 **color**

Template Arguments

entrypoint_vert

positional

type: `str`

The name of the entrypoint function to vertex the shader.

entrypoint_pixel

positional

type: `str`

The name of the entrypoint function to pixel the shader.

Example

```
#pragma SSV vert_pixel vert pixel

#ifdef SHADER_STAGE_VERTEX
// Additional vertex->fragment interpolators can be defined as follows
layout(location = 3) out vec2 uv;

void vert() {
    uv = in_vert.xy;
    gl_Position = vec4(in_vert.xyz, 1.);
    gl_Position = uProjMat * uViewMat * gl_Position;
}
#endif // SHADER_STAGE_VERTEX

#ifdef SHADER_STAGE_FRAGMENT
// Make sure to also define any custom interpolators in the fragment stage
layout(location = 3) in vec2 uv;

vec4 pixel(vec3 pos) {
    vec3 col = vec3(0.);
    col.rg = uv;
    col.b = pos.z;
    return col;
}
#endif //SHADER_STAGE_FRAGMENT
```

1.2.6 Signed Distance Field Template

`#pragma SSV sdf`

This template exposes a single entrypoint to a pixel shader. It's designed to mimic the API of ShaderToy

Entrypoint Signature

float **map**(vec3 pos)

Parameters

- **pos** – the position to sample sample the signed distance field at.

Returns
the signed distance to the surface.

Template Arguments

entrypoint

positional

type: `str`

The name of the sdf function in the shader.

--camera_mode

choices: `INTERACTIVE, AUTO`

default: `AUTO`

How the camera should behave. `INTERACTIVE`, uses the canvas' camera. `AUTO`, automatically rotates around the scene using the `--camera_distance` and `--rotate_speed` variables.

--camera_distance

default: `10.0`

type: `float`

The distance of the camera from the centre of the distance field.

--rotate_speed

default: `0.1`

type: `float`

The orbit speed of the camera around the SDF, in radians/second.

--raymarch_steps

default: `128`

type: `int`

The number of raymarching steps to use when rendering, turn this up if the edges of surfaces look soft.

--raymarch_distance

default: `32.0`

type: `float`

The maximum distance to raymarch.

--light_dir

default: `normalize(vec3(0.5, 0.5, -0.9))`

type: `vec3`

The maximum distance to raymarch.

--render_mode

choices: `SOLID, DEPTH, XRAY, ISOLINES`

default: `SOLID`

How the distance field should be rendered. Check the documentation for more information about each mode.

Example

```
#pragma SSV sdf sdf_main --camera_distance 2. --rotate_speed 1.5 --render_mode SOLID

// SDF taken from: https://iquilezles.org/articles/distfunctions/
float sdCappedTorus(vec3 p, vec2 sc, float ra, float rb) {
  p.x = abs(p.x);
  float k = (sc.y*p.x>sc.x*p.y) ? dot(p.xy,sc) : length(p.xy);
  return sqrt( dot(p,p) + ra*ra - 2.0*ra*k ) - rb;
}

float sdf_main(vec3 p) {
    float t = 2.*(sin(uTime)*0.5+0.5)+0.2;
    return sdCappedTorus(p, vec2(sin(t), cos(t)), 0.5, 0.2);
}
```

1.2.7 Point Cloud Template

`#pragma SSV point_cloud`

This template exposes a single entrypoint to a vertex shader. It treats input vertices as points and uses a geometry shader to turn each vertex into a camera-facing sprite.

Entrypoint Signature

VertexOutput **vert()**

Returns
a VertexOutput struct containing the transformed vertex.

struct **VertexOutput**

vec4 **position**

float **size**
The size of the sprite representing the point.

vec4 **color**

The shader is expected to take input from the following vertex attributes:

vec4 **in_vert**

vec4 **in_color**

Template Arguments

None

Example

```
#pragma SSV point_cloud mainPoint
VertexOutput mainPoint()
{
    VertexOutput o;
    vec4 pos = vec4(in_vert, 1.0);
    pos = uViewMat * pos;
    pos = uProjMat * pos;
    o.position = pos;
    o.color = vec4(in_color, 1.);
    o.size = 30.0/uResolution.x;
    return o;
}
```

1.2.8 Render Test

`#pragma SSV render_test`

This template generates a simple pixel shader which displays a colour changing gradient. Useful as a shorthand to make a quick shader to test the rendering system.

Entrypoint Signature

None

Template Arguments

None

Example

```
#pragma SSV render_test
```

1.2.9 Geometry Shader Template

`#pragma SSV geometry`

This template exposes an entrypoint to a vertex shader and an entrypoint to a geometry shader. It treats input vertices as points which are processed by the user defined vertex shader and can then be turned into triangle primitives by the user defined geometry shader.

Entrypoint Signature

Vertex Stage

VertexOutput **vert**()

Where `VertexOutput` is substituted for the value of `--vertex_output_struct`.

returns
a <VertexOutput> struct containing the transformed vertex.

If the `--vertex_output_struct` argument isn't set then the <`VertexOutput`> struct is as follows:

struct **DefaultVertexOutput**

vec4 **position**

vec4 **color**

float **size**

The size of the sprite representing the point.

If the `--custom_vertex_input` flag isn't specified then the vertex shader is expected to take input from the following vertex attributes:

vec4 **in_vert**

vec4 **in_color**

Geometry Stage

void **geo**(VertexOutput i)

Parameters

- **i** – the struct containing the processed vertex data

The geometry function is expected to write to these output variables before each call to `EmitVertex()`:

vec4 **gl_Position**

The transformed (clip space) position of the vertex to emit.

vec4 **out_color**

The final colour of the vertex to be emitted.

The geometry function is responsible for calling `EmitVertex()` and `EndPrimitive()` as needed and must not emit more vertices in a single invocation than what is specified in `--geo_max_vertices` (`default=4`).

Template Arguments

entrypoint_vert

positional

type: `str`

The name of the entrypoint function to vertex the shader.

entrypoint_geo

positional

type: `str`

The name of the entrypoint function to geometry the shader.

--vertex_output_struct

default: `DefaultVertexOutput`

type: `float`

The name of the struct containing data to be transferred from the vertex stage to the geometry stage.

--geo_max_vertices

default: `4`

type: `const int`

The maximum number of vertices which can be output be the geometry stage per input vertex. Must be a constant.

--custom_vertex_input

type: `flag`

When this flag is passed, the default vertex input attributes are not created and must be declared by the user.

Example

```
#pragma SSV geometry mainPoint mainGeo
#ifdef SHADER_STAGE_VERTEX
DefaultVertexOutput mainPoint()
{
    DefaultVertexOutput o;
    // Transform the points using the camera matrices
    vec4 pos = vec4(in_vert, 1.0);
    pos = uViewMat * pos;
    pos = uProjMat * pos;
    o.position = pos;
    o.color = vec4(in_color, 1.);
    o.size = 10.0/uResolution.x;
    return o;
}
#endif // SHADER_STAGE_VERTEX
#ifdef SHADER_STAGE_GEOMETRY
void mainGeo(DefaultVertexOutput i) {
    // Generate a quad for each point
    vec4 position = i.position;
    float size = i.size;
    out_color = i.color;
    vec4 aspect_ratio = vec4(1., uResolution.x/uResolution.y, 1., 1.);
    gl_Position = position + size * vec4(-1., -1., 0.0, 0.0) * aspect_ratio;
    EmitVertex();
    gl_Position = position + size * vec4(1., -1., 0.0, 0.0) * aspect_ratio;
    EmitVertex();
    gl_Position = position + size * vec4(-1., 1., 0.0, 0.0) * aspect_ratio;
    EmitVertex();
```

(continues on next page)

(continued from previous page)

```
    gl_Position = position + size * vec4(1., 1., 0.0, 0.0) * aspect_ratio;
    EmitVertex();
    EndPrimitive();
}
#endif // SHADER_STAGE_GEOMETRY
```

1.3 Shader Library

pySSV provides a small library of commonly used GLSL functions to reduce boilerplate. These can be imported into any shader using the `#include "xxxx.glsl"` directive.

1.3.1 Library Reference

Colour Utilities

`#include "color_utils.glsl"`

This file includes a number of functions related to colour space transforms and colour maps.

const float **_HCV_EPSILON**

const float **_HSL_EPSILON**

const float **_HCY_EPSILON**

const float **SRGB_GAMMA**

const float **SRGB_INVERSE_GAMMA**

const float **SRGB_ALPHA**

const mat3 **RGB_2_XYZ**
: Used to convert from linear RGB to XYZ space

const mat3 **XYZ_2_RGB**
: Used to convert from XYZ to linear RGB space

const vec3 **LUMA_COEFFS**
: The RGB coefficients used to compute luminosity.

float **luminance**(vec3 rgb)
: Converts a **linear** rgb colour to its luminance.

  Parameters

  - **rgb** – the linear rgb value.

  Returns
  the linear luminance of the colour.

vec3 **rgb_to_srgb_approx**(vec3 rgb)
: Converts a linear rgb colour to sRGB using an approximation.

  Parameters

- **rgb** – the linear rgb value.

Returns
the colour in sRGB.

vec3 **srgb_to_rgb_approx**(vec3 srgb)

Converts an sRGB colour to linear rgb using an approximation.

Parameters

- **rgb** – the sRGB colour.

Returns
the colour in linear rgb.

float **linear_to_srgb**(float channel)

Converts a linear value to sRGB.

Parameters

- **rgb** – the linear value.

Returns
the value in sRGB space.

float **srgb_to_linear**(float channel)

Converts an sRGB value to linear space.

Parameters

- **rgb** – the sRGB value.

Returns
the value in linear space.

vec3 **rgb_to_srgb**(vec3 rgb)

Converts a linear rgb colour to sRGB (exact).

Parameters

- **rgb** – the linear rgb value.

Returns
the colour in sRGB.

vec3 **srgb_to_rgb**(vec3 srgb)

Converts an sRGB colour to linear rgb (exact).

Parameters

- **rgb** – the sRGB colour.

Returns
the colour in linear rgb.

vec3 **rgb_to_xyz**(vec3 rgb)

Converts a color from linear RGB to XYZ space

vec3 **xyz_to_rgb**(vec3 xyz)

Converts a color from XYZ to linear RGB space

vec3 **xyz_to_xyY**(vec3 xyz)

Converts a color from XYZ to xyY space (Y is luminosity)

vec3 **xyY_to_xyz**(vec3 xyY)
Converts a color from xyY space to XYZ space

vec3 **rgb_to_xyY**(vec3 rgb)
Converts a color from linear RGB to xyY space

vec3 **xyY_to_rgb**(vec3 xyY)
Converts a color from xyY space to linear RGB

vec3 **rgb_to_hcv**(vec3 rgb)
Converts a value from linear RGB to HCV (Hue, Chroma, Value)

vec3 **hue_to_rgb**(float hue)
Converts from pure Hue to linear RGB

vec3 **hsv_to_rgb**(vec3 hsv)
Converts from HSV to linear RGB

vec3 **hsl_to_rgb**(vec3 hsl)
Converts from HSL to linear RGB

vec3 **hcy_to_rgb**(vec3 hcy)
Converts from HCY to linear RGB

vec3 **rgb_to_hsv**(vec3 rgb)
Converts from linear RGB to HSV

vec3 **rgb_to_hsl**(vec3 rgb)
Converts from linear rgb to HSL

vec3 **rgb_to_hcy**(vec3 rgb)
Converts from rgb to hcy (Hue, Chroma, Luminance)

vec3 **rgb_to_ycbcr**(vec3 rgb)
RGB to YCbCr, ranges [0, 1]

vec3 **ycbcr_to_rgb**(vec3 yuv)
YCbCr to RGB

vec3 **xyz_to_srgb**(vec3 xyz)

vec3 **xyY_to_srgb**(vec3 xyY)

vec3 **hue_to_srgb**(float hue)

vec3 **hsv_to_srgb**(vec3 hsv)

vec3 **hsl_to_srgb**(vec3 hsl)

vec3 **hcy_to_srgb**(vec3 hcy)

vec3 **ycbcr_to_srgb**(vec3 yuv)

vec3 **srgb_to_xyz**(vec3 srgb)

vec3 **hue_to_xyz**(float hue)

vec3 **hsv_to_xyz**(vec3 hsv)

vec3 **hsl_to_xyz**(vec3 hsl)

vec3 **hcy_to_xyz**(vec3 hcy)

vec3 **ycbcr_to_xyz**(vec3 yuv)

vec3 **srgb_to_xyY**(vec3 srgb)

vec3 **hue_to_xyY**(float hue)

vec3 **hsv_to_xyY**(vec3 hsv)

vec3 **hsl_to_xyY**(vec3 hsl)

vec3 **hcy_to_xyY**(vec3 hcy)

vec3 **ycbcr_to_xyY**(vec3 yuv)

vec3 **srgb_to_hcv**(vec3 srgb)

vec3 **xyz_to_hcv**(vec3 xyz)

vec3 **xyY_to_hcv**(vec3 xyY)

vec3 **hue_to_hcv**(float hue)

vec3 **hsv_to_hcv**(vec3 hsv)

vec3 **hsl_to_hcv**(vec3 hsl)

vec3 **hcy_to_hcv**(vec3 hcy)

vec3 **ycbcr_to_hcv**(vec3 yuv)

vec3 **srgb_to_hsv**(vec3 srgb)

vec3 **xyz_to_hsv**(vec3 xyz)

vec3 **xyY_to_hsv**(vec3 xyY)

vec3 **hue_to_hsv**(float hue)

vec3 **hsl_to_hsv**(vec3 hsl)

vec3 **hcy_to_hsv**(vec3 hcy)

vec3 **ycbcr_to_hsv**(vec3 yuv)

vec3 **srgb_to_hsl**(vec3 srgb)

vec3 **xyz_to_hsl**(vec3 xyz)

vec3 **xyY_to_hsl**(vec3 xyY)

vec3 **hue_to_hsl**(float hue)

vec3 **hsv_to_hsl**(vec3 hsv)

vec3 **hcy_to_hsl**(vec3 hcy)

vec3 **ycbcr_to_hsl**(vec3 yuv)

vec3 **srgb_to_hcy**(vec3 srgb)

vec3 **xyz_to_hcy**(vec3 xyz)

vec3 **xyY_to_hcy**(vec3 xyY)

vec3 **hue_to_hcy**(float hue)

vec3 **hsv_to_hcy**(vec3 hsv)

vec3 **hsl_to_hcy**(vec3 hsl)

vec3 **ycbcr_to_hcy**(vec3 yuv)

vec3 **srgb_to_ycbcr**(vec3 srgb)

vec3 **xyz_to_ycbcr**(vec3 xyz)

vec3 **xyY_to_ycbcr**(vec3 xyY)

vec3 **hue_to_ycbcr**(float hue)

vec3 **hsv_to_ycbcr**(vec3 hsv)

vec3 **hsl_to_ycbcr**(vec3 hsl)

vec3 **hcy_to_ycbcr**(vec3 hcy)

vec3 **oklab_to_rgb**(const vec3 lab)

Converts a colour from OKLAB space to linear RGB.

https://bottosson.github.io/posts/oklab/

Parameters

- **lab** – the colour in OKLAB space.

Returns

the colour in linear rgb.

vec3 **rgb_to_oklab**(const vec3 rgb)

Converts a colour from linear RGB space to OKLAB.

https://bottosson.github.io/posts/oklab/

Parameters

- **rgb** – the colour in linear rgb.

Returns

the colour in OKLAB space.

vec3 **colmap_greys**(float x)

Maps a value between [0-1] to a colour.

Maps from black to white.

Parameters

- **x** – the value to colour map.

Returns

the colour mapped to the given value.

vec3 **colmap_tinted**(float x, vec3 oklabCol)

Maps a value between [0-1] to a colour.

Maps from `oklabCol` to white.

Parameters

- **x** – the value to colour map.
- **oklabCol** – the base colour to use in OKLAB space.

Returns
the colour mapped to the given value.

vec3 **colmap_mix**(float x, vec3 oklabColA, vec3 oklabColB)

Maps a value between [0-1] to a colour.

Maps from `oklabColA` to `oklabColB` (linear interpolation in OKLAB space).

Parameters

- **x** – the value to colour map.
- **oklabColA** – the start colour to use in OKLAB space.
- **oklabColB** – the end colour to use in OKLAB space.

Returns
the colour mapped to the given value.

vec3 **colmap_mix_3**(float x, vec3 oklabColA, vec3 oklabColB, vec3 oklabColC)

Maps a value between [0-1] to a colour.

Maps from `oklabColA` to `oklabColB` to `oklabColC` (linear interpolation in OKLAB space).

Parameters

- **x** – the value to colour map.
- **oklabColA** – the start colour to use in OKLAB space.
- **oklabColB** – the middle colour to use in OKLAB space.
- **oklabColC** – the end colour to use in OKLAB space.

Returns
the colour mapped to the given value.

vec3 **colmap_purples**(float x)

Maps a value between [0-1] to a colour.

Maps from purple to white.

Parameters

- **x** – the value to colour map.

Returns
the colour mapped to the given value.

vec3 **colmap_blues**(float x)

Maps a value between [0-1] to a colour.

Maps from blue to white.

Parameters

- **x** – the value to colour map.

Returns
the colour mapped to the given value.

vec3 **colmap_greens**(float x)

Maps a value between [0-1] to a colour.

Maps from green to white.

Parameters

- **x** – the value to colour map.

Returns
the colour mapped to the given value.

vec3 **colmap_oranges**(float x)

Maps a value between [0-1] to a colour.

Maps from orange to white.

Parameters

- **x** – the value to colour map.

Returns
the colour mapped to the given value.

vec3 **colmap_reds**(float x)

Maps a value between [0-1] to a colour.

Maps from red to white.

Parameters

- **x** – the value to colour map.

Returns
the colour mapped to the given value.

vec3 **colmap_PiYG**(float x)

Maps a value between [0-1] to a colour.

Maps from pink to white to green.

Parameters

- **x** – the value to colour map.

Returns
the colour mapped to the given value.

vec3 **colmap_PRGn**(float x)

Maps a value between [0-1] to a colour.

Maps from purple to white to green.

Parameters

- **x** – the value to colour map.

Returns
the colour mapped to the given value.

vec3 **colmap_PuOr**(float x)

Maps a value between [0-1] to a colour.

Maps from orange to white to purple.

Parameters

- **x** – the value to colour map.

Returns
the colour mapped to the given value.

vec3 **colmap_RdBu**(float x)

Maps a value between [0-1] to a colour.

Maps from red to white to blue.

Parameters

- **x** – the value to colour map.

Returns
the colour mapped to the given value.

vec3 **colmap_coolwarm**(float x)

Maps a value between [0-1] to a colour.

Maps from blue to red.

Parameters

- **x** – the value to colour map.

Returns
the colour mapped to the given value.

vec3 **colmap_PurGnYl**(float x)

Maps a value between [0-1] to a colour.

Maps from purple to green to yellow. Looks a bit like viridis.

Parameters

- **x** – the value to colour map.

Returns
the colour mapped to the given value.

vec3 **colmap_twilight**(float x)

Maps a value between [0-1] to a colour.

Maps from white to blue to black to red and back to white.

Parameters

- **x** – the value to colour map.

Returns
the colour mapped to the given value.

vec3 **colmap_viridis**(float t)

Maps a value between [0-1] to a colour.

This is an approximation of the popular 'viridis' colormap taken from: https://www.shadertoy.com/view/WlfXRN

Parameters

- **x** – the value to colour map.

Returns
the colour mapped to the given value.

vec3 **colmap_plasma**(float t)

Maps a value between [0-1] to a colour.

This is an approximation of the popular 'plasma' colormap taken from: https://www.shadertoy.com/view/WlfXRN

Parameters

- **x** – the value to colour map.

Returns
the colour mapped to the given value.

vec3 **colmap_magma**(float t)

Maps a value between [0-1] to a colour.

This is an approximation of the popular 'magma' colormap taken from: https://www.shadertoy.com/view/WlfXRN

Parameters

- **x** – the value to colour map.

Returns
the colour mapped to the given value.

vec3 **colmap_inferno**(float t)

Maps a value between [0-1] to a colour.

This is an approximation of the popular 'inferno' colormap taken from: https://www.shadertoy.com/view/WlfXRN

Parameters

- **x** – the value to colour map.

Returns
the colour mapped to the given value.

Random Number Generation / Hashing

```
#include "random.glsl"
```

This file includes a number of functions related to random number generation, hashing, and screen space dithering.

float **hash11**(float p)

Hashes the given input. Uses the "Hash without Sine" algorithm (https://www.shadertoy.com/view/4djSRW). Tends to fail with small changes in p.

Parameters

- **p** – the input to the hash function as a `float`.

Returns
the hash of p as a `float`.

float **hash12**(vec2 p)

Hashes the given input. Uses the "Hash without Sine" algorithm (https://www.shadertoy.com/view/4djSRW). Tends to fail with small changes in p.

Parameters

- **p** – the input to the hash function as a `vec2`.

Returns
the hash of p as a `float`.

float **hash13**(vec3 p3)

Hashes the given input. Uses the "Hash without Sine" algorithm (https://www.shadertoy.com/view/4djSRW). Tends to fail with small changes in p.

Parameters

- **p** – the input to the hash function as a `vec3`.

Returns
the hash of p as a `float`.

float **hash14**(vec4 p4)

Hashes the given input. Uses the "Hash without Sine" algorithm (https://www.shadertoy.com/view/4djSRW). Tends to fail with small changes in p.

Parameters

- **p** – the input to the hash function as a `vec4`.

Returns
the hash of p as a `float`.

vec2 **hash21**(float p)

Hashes the given input. Uses the "Hash without Sine" algorithm (https://www.shadertoy.com/view/4djSRW). Tends to fail with small changes in p.

Parameters

- **p** – the input to the hash function as a `float`.

Returns
the hash of p as a `vec2`.

vec2 **hash22**(vec2 p)

Hashes the given input. Uses the "Hash without Sine" algorithm (https://www.shadertoy.com/view/4djSRW). Tends to fail with small changes in p.

Parameters

- **p** – the input to the hash function as a `vec2`.

Returns
the hash of p as a `vec2`.

vec2 **hash23**(vec3 p3)

Hashes the given input. Uses the "Hash without Sine" algorithm (https://www.shadertoy.com/view/4djSRW). Tends to fail with small changes in p.

Parameters

- **p** – the input to the hash function as a `vec3`.

Returns
the hash of p as a `vec2`.

vec3 **hash31**(float p)

Hashes the given input. Uses the "Hash without Sine" algorithm (https://www.shadertoy.com/view/4djSRW). Tends to fail with small changes in p.

Parameters

- **p** – the input to the hash function as a `float`.

Returns
the hash of p as a `vec3`.

vec3 **hash32**(vec2 p)

Hashes the given input. Uses the "Hash without Sine" algorithm (https://www.shadertoy.com/view/4djSRW). Tends to fail with small changes in p.

Parameters

- **p** – the input to the hash function as a `vec2`.

Returns
the hash of p as a `vec3`.

vec3 **hash33**(vec3 p3)

Hashes the given input. Uses the "Hash without Sine" algorithm (https://www.shadertoy.com/view/4djSRW). Tends to fail with small changes in p.

Parameters

- **p** – the input to the hash function as a `vec3`.

Returns
the hash of p as a `vec3`.

vec4 **hash41**(float p)

Hashes the given input. Uses the "Hash without Sine" algorithm (https://www.shadertoy.com/view/4djSRW). Tends to fail with small changes in p.

Parameters

- **p** – the input to the hash function as a `float`.

Returns
the hash of p as a `vec4`.

vec4 **hash42**(vec2 p)

Hashes the given input. Uses the "Hash without Sine" algorithm (https://www.shadertoy.com/view/4djSRW). Tends to fail with small changes in p.

Parameters

- **p** – the input to the hash function as a `vec2`.

Returns
the hash of p as a `vec4`.

vec4 **hash43**(vec3 p)

Hashes the given input. Uses the "Hash without Sine" algorithm (https://www.shadertoy.com/view/4djSRW). Tends to fail with small changes in p.

Parameters

- **p** – the input to the hash function as a `vec3`.

Returns
the hash of p as a `vec4`.

vec4 **hash44**(vec4 p4)

Hashes the given input. Uses the "Hash without Sine" algorithm (https://www.shadertoy.com/view/4djSRW). Tends to fail with small changes in p.

Parameters

- **p** – the input to the hash function as a `vec4`.

Returns
the hash of p as a `vec4`.

uint **pcg**(uint v)

Hashes the given input. Uses the PCG algorithm (https://www.pcg-random.org/). This algorithm strikes a very good balance between performance and high quality hashing.

Parameters

- **v** – the input to the hash function as a `uint`.

Returns
the hash of p as a `uint`.

uvec2 **pcg2d**(uvec2 v)

Hashes the given input. Uses the PCG algorithm (https://www.pcg-random.org/). This algorithm strikes a very good balance between performance and high quality hashing.

Parameters

- **v** – the input to the hash function as a `uvec2`.

Returns
the hash of p as a `uvec2`.

uvec3 **pcg3d**(uvec3 v)

Hashes the given input. Uses the PCG algorithm (https://www.pcg-random.org/). This algorithm strikes a very good balance between performance and high quality hashing.

Parameters

- **v** – the input to the hash function as a `uvec3`.

Returns
the hash of p as a `uvec3`.

uvec4 **pcg4d**(uvec4 v)

Hashes the given input. Uses the PCG algorithm (https://www.pcg-random.org/). This algorithm strikes a very good balance between performance and high quality hashing.

Parameters

- **v** – the input to the hash function as a `uvec4`.

Returns
the hash of p as a `uvec4`.

vec3 **_dither_col**(vec3 col, vec2 p, const int bits)

Dithers the input colour using triangular distributed value noise.

Parameters

- **col** – the colour to dither.
- **p** – the screen-space position in pixels.
- **bits** – how many least significant bits should be dithered.

Returns
the dithered colour.

vec3 **_dither_col_ordered**(vec3 col, vec2 p, const int bits)

Dithers the input colour using Valve's ordered dithering algorithm. http://alex.vlachos.com/graphics/Alex_Vlachos_Advanced_VR_Rendering_GDC2015.pdf

Parameters

- **col** – the colour to dither.
- **p** – the screen-space position in pixels.
- **bits** – how many least significant bits should be dithered.

Returns
the dithered colour.

vec3 **dither_col**(vec3 col, vec2 p)

Dithers the input colour using triangular distributed value noise. Dithers to 8 bit per pixel precision (256 values).

Parameters

- **col** – the colour to dither.
- **p** – the screen-space position in pixels.

Returns
the dithered colour.

vec3 **dither_col_ordered**(vec3 col, vec2 p)

Dithers the input colour using Valve's ordered dithering algorithm. Dithers to 8 bit per pixel precision (256 values). http://alex.vlachos.com/graphics/Alex_Vlachos_Advanced_VR_Rendering_GDC2015.pdf

Parameters

- **col** – the colour to dither.
- **p** – the screen-space position in pixels.

Returns
the dithered colour.

Signed Distance Field Operators

```
#include "sdf_ops.glsl"
```

This file includes a number of functions related to signed distance field operations.

float **op_not**(float a)

Inverts a signed distance field. (Logical NOT)

Parameters

- **a** – the sdf to invert.

Returns
the new sdf.

float **op_union**(float a, float b)

Computes the union between two distance fields. (logical OR)

Parameters

- **a** – the first sdf.
- **b** – the second sdf.

Returns
the combined sdf.

float **op_intersect**(float a, float b)

Computes the intersection between two distance fields. (logical AND)

Parameters

- **a** – the first sdf.
- **b** – the second sdf.

Returns
the combined sdf.

float **op_subtract**(float a, float b)

Computes the difference between two distance fields. (logical SUBTRACT)

Parameters

- **a** – the first sdf.
- **b** – the second sdf.

Returns
the combined sdf.

float **op_xor**(float a, float b)

Computes the exclusive OR between two distance fields. (logical XOR)

Parameters

- **a** – the first sdf.
- **b** – the second sdf.

Returns
the combined sdf.

float **op_sminCubic**(float a, float b, float k)

float **op_smaxCubic**(float a, float b, float k)

float **op_smoothUnion**(float a, float b, float k)

Computes the union between two distance fields with a soft intersection. (logical OR)

Parameters

- **a** – the first sdf.
- **b** – the second sdf.
- **k** – the amount of smoothing to apply to the intersection.

Returns
the combined sdf.

float **op_smoothIntersect**(float a, float b, float k)

Computes the intersection between two distance fields with a soft intersection. (logical AND)

Parameters

- **a** – the first sdf.
- **b** – the second sdf.
- **k** – the amount of smoothing to apply to the intersection.

Returns
the combined sdf.

float **op_smoothSubtract**(float a, float b, float k)

Computes the difference between two distance fields with a soft intersection. (logical SUBTRACT)

Parameters

- **a** – the first sdf.
- **b** – the second sdf.
- **k** – the amount of smoothing to apply to the intersection.

Returns
the combined sdf.

float **op_smoothXor**(float a, float b, float k)

Computes the exclusive OR between two distance fields with a soft intersection. (logical XOR)

Parameters

- **a** – the first sdf.
- **b** – the second sdf.
- **k** – the amount of smoothing to apply to the intersection.

Returns
the combined sdf.

Text Rendering Utilities

```
#include "text.glsl"
```

This file includes a number of functions related to text drawing.

1.3.2 Internal Utilities

These glsl files are usually included automatically by the shader template.

Compiler Compatibility

```
#include "compat.glsl"
```

This file serves a compatibility layer allowing different GLSL compilers to be used; it relies on special preprocessor pragma to work. Including this, automatically defines the GLSL version, any needed compiler extensions, and the default precision.

Global Uniform Declarations

```
#include "global_uniforms.glsl"
```

This file declares all the built in uniforms set by pySSV and any uniforms declared dynamically by the preprocessor.

float **uTime**
: The time in seconds since the canvas started running.

int **uFrame**
: The current frame number, starting from 0.

vec4 **uResolution**
: The resolution of the current render buffer in pixels.

vec2 **uMouse**
: The coordinates of the mouse relative to the canvas in pixels.

bool **uMouseDown**
: Whether the mouse button is pressed.

mat4x4 **uViewMat**
: The main camera's view matrix.

mat4x4 **uProjMat**
: The main camera's projection matrix.

vec3 **uViewDir**
: The forward vector of the main camera.

1.4 Writing Shader Templates

pySSV has a powerful shader templating system which aims to reduce boilerplate code and improve compatibility of shaders with different rendering backends.

When a user writes a shader they specify which shader template to compile with using the `#pragma SSV <template_name>` directive. The preprocessor searches for the template specified by the user in the following order:

1. Templates passed in to the `shader()` method along with the source code.
2. Templates in the folder passed in to the `shader()` method.
3. Templates in the built in templates folder (packaged with pySSV).

Shader templates must have a filename in the form `template_<template_name>.glsl` to be found. For instance, if the following shader source code is passed in the `shader()` method:

```
#pragma SSV pixel mainImage
vec4 mainImage(in vec2 fragCoord)
{
    vec2 uv = fragCoord/uResolution.xy;
    vec3 col = sin(uv.xyx + iTime * vec3(3, 4, 5)) * 0.5 + 0.5;
    return vec4(vec3(col), 1.0);
}
```

The shader preprocessor will search for the template: `template_shadertoy.glsl` this template is parsed to determine to parse further arguments in the `#pragma SSV <template_name> ...` directive. The template file is then preprocessed with the user's source code and template parameters injected in to the template.

In this example the `pixel` shader template looks like this:

```
#pragma SSVTemplate define pixel --author "Thomas Mathieson" --description "A simple␣
↪full screen pixel shader."
#pragma SSVTemplate stage vertex
#pragma SSVTemplate stage fragment
// Arguments get converted into compiler defines by the preprocessor
// an argument's name is transformed to match our naming convention:
//    entrypoint -> T_ENTRYPOINT
//    _varying_struct -> T_VARYING_STRUCT
#pragma SSVTemplate arg entrypoint -d "The name of the entrypoint function to the shader.
↪"
#pragma SSVTemplate arg _z_value -d "The constant value to write into the depth buffer.␣
↪0 is close to the camera, 1 is far away." --default 0.999
// Prefixing an argument name with an underscore is shorthand for --non_positional
// #pragma SSVTemplate arg _varying_struct --type str
// An example for an SDF shader
// #pragma SSVTemplate arg _render_mode --choices solid xray isolines 2d

// Include any default includes we think the user might want
// compat.glsl automatically declares the #version and precision directives when needed,␣
↪it should always be the
// first file to be included in the template.
#include "compat.glsl"
// global_uniforms.glsl contains the declarations for all uniforms which are␣
↪automatically passed in by pySSV
#include "global_uniforms.glsl"


// Use these preprocessor blocks to specify what code to compile for each shader stage.
// These macros (SHADER_STAGE_<stage_name>) are defined automatically by the␣
↪preprocessor.
#ifdef SHADER_STAGE_VERTEX
layout(location = 0) in vec2 in_vert;
layout(location = 1) in vec3 in_color;
layout(location = 0) out vec3 color;
layout(location = 1) out vec2 position;
void main() {
    gl_Position = vec4(in_vert, (T_Z_VALUE)*2.-1., 1.0);
    color = in_color;
    position = in_vert*0.5+0.5;
```

(continues on next page)

(continued from previous page)

```
}
#endif //SHADER_STAGE_VERTEX


#ifdef SHADER_STAGE_FRAGMENT
out vec4 fragColor;
layout(location = 0) in vec3 color;
layout(location = 1) in vec2 position;

// Including the magic string "TEMPLATE_DATA" causes the user's shader code to be␣
↪injected here.
#include "TEMPLATE_DATA"

void main() {
    // T_ENTRYPOINT is a macro that was defined automatically when the argument defined
    // by '#pragma SSVTemplate arg entrypoint' was passed in.
    fragColor = T_ENTRYPOINT(position * uResolution.xy);
    // Despite the explicit layout, sometimes in_color still gets stripped...
    fragColor.a += color.r*1e-20;
}
#endif //SHADER_STAGE_FRAGMENT
```

When preprocessing the template the arguments passed in to the `#pragma SSV <template_name>` directive are converted to preprocessor defines. Argument names are converted to uppercase and prefixed with `T_`, so the argument `entrypoint` is passed in to the shader template as `#define T_ENTRYPOINT <value>`. The glsl source passed in by the user to the template is injected in the shader template using the special `#include "TEMPLATE_DATA"` directive which simply expands to the user's glsl code when preprocessed. Arguments are passed in to the shader as defines exactly as they are specified by the user:

```
// If the user specifies these arguments
#pragma SSV sdf sdf_main --camera_speed -1.5 --light_dir "normalize(vec3(0.1, 0.2, 0.3))"

// They will be defined by the preprocessor as
#define T_ENTRYPOINT sdf_main
#define T_CAMERA_SPEED -1.5
// Notice in this case that to specify a value which contains whitespace, it must be␣
↪wrapped in quotation marks.
// A few basic c++ style escape sequences are supported in this case as well (\", \n, \
↪t).
#define T_LIGHT_DIR normalize(vec3(0.1, 0.2, 0.3))
```

1.4.1 `SSVTemplate` Directives

The template is parametrised using `#pragma SSVTemplate` directives.

Define Directive

`#pragma SSVTemplate define`

This directive is used to define a shader template and any metadata associated with it.

Parameters:

name

The name of the shader template. This should only consist of characters valid in filenames and should not contain spaces.

--author

The shader template's author.

--description

A brief description of the shader template and what it does.

Stage Directive

`#pragma SSVTemplate stage`

This directive specifies a shader stage to compile this template for.

Parameters:

stage

The stage(s) to compile for. Accepts one or more of: `vertex`, `fragment`, `tess_control`, `tess_evaluation`, `geometry`, or `compute`.

Arg Directive

`#pragma SSVTemplate arg`

This directive defines an argument to be passed in to the shader template in the `#pragma SSV <template_name> [args]` directive.

Parameters:

name

The name of the argument to be passed in to the shader; prefixing the name with an underscore implies the `--non_positional` flag.

--non_positional

[Flag] Treat this as a non-positional argument; it's name is automatically prefixed with `--`.

--action

What to do when this argument is encountered. Accepts the following options:

1. `store` (default) Stores the value the user passes in to the argument in the argument.
2. `store_const` Stores a constant value (defined in `--const`) in the argument when this flag is specified.

3. `store_true` A special case of `store_const` which stores `true` when the flag is specified and `false` if it isn't.
4. `store_false` The inverse of `store_true`.

`--default`

The default value for this argument if it isn't specified.

`--choices`

Limits the valid values of this argument to those specified here. This parameter accepts multiple values. The choices are defined as compiler macros allowing you to test for choices as follows:

```
#pragma SSVTemplate arg _camera_mode --choices INTERACTIVE AUTO

#if T_CAMERA_MODE == AUTO
...
```

`--const`

When using the 'store_const' action, specifies what value to store.

Input Primitive Directive

`#pragma SSVTemplate input_primitive`

This directive allows the shader to specify what type of OpenGL input primitive it's expecting. If this directive is not specified, the renderer defaults to `TRIANGLES`.

Parameters:

`primitive_type`

The primitive type for the renderer to dispatch the shader with. Accepts one of the following options:

1. `POINTS` treat the input vertices as points.
2. `LINES` treat the input vertices as an array of line segments; each line consumes 2 vertices.
3. `LINE_LOOP`
4. `LINE_STRIP` (unsupported) treat the input vertices as a line strip; each line consumes 1 vertex.
5. `TRIANGLES` (default) treat the input vertices as a an array of triangles; each triangle consumes 3 vertices.
6. `TRIANGLE_STRIP`
7. `TRIANGLE_FAN`
8. `LINES_ADJACENCY`
9. `LINE_STRIP_ADJACENCY`
10. `TRIANGLES_ADJACENCY`
11. `TRIANGLE_STRIP_ADJACENCY`
12. `PATCHES`

1.5 Examples

This section contains several examples generated from Jupyter notebooks. The widgets have been embedded into the page for demonstrative purposes.

1.5.1 Additional Examples

This notebook contains more advanced examples using pySSV.

[ ]:
```
# Google colab support
try:
    # Try enabling custom widgets, this will fail silently if we're not in Google Colab
    from google.colab import output
    output.enable_custom_widget_manager()
    # Install pySSV for this session
    %pip install pySSV
except:
    pass
```

Video

This example takes advantage of the point cloud shader template to render video in real time. In this case, the video is compressed into a quadtree (this is obviously not a very good compression algorithm for video, but it's easy to encode/decode so it makes for a good demonstration) which is stored in a texture. Each row of the animation texture stores the quadtree for one frame where each pixel is one cell in the quadtree. The way this is implemented means the cells don't strictly need to be part of a quad tree, they just each represent a single square of a given size, colour, and location.

The video quadtree is created by a separate program the code for which can be found here: https://github.com/space928/badapple-quadtree-encoder

[ ]:
```
import os.path

# Download the compressed video file from the internet if needed (with the user's␣
↪permission)
filename = "badapple_quad.pkl"
if not os.path.isfile(filename):
    if input("Encoded video file not found! Do you want to download it now (yes/no)?
↪")[0] == "y":
        url = "https://github.com/space928/badapple-quadtree-encoder/releases/download/0.
↪1.0/badapple_quad.pkl"
        import urllib.request
        try:
            print("Downloading...")
            urllib.request.urlretrieve(url, filename)
            print("Successfully downloaded encoded video file!")
        except Exception as e:
            print(f"Failed to download video: {e}")
else:
    print(f"Video file '{filename}' already exists, using existing version...")
```

[ ]:
```
import pySSV as ssv
import numpy as np
import pickle as pkl

canvas5 = ssv.canvas(use_renderdoc=True)
with open("badapple_quad.pkl", "rb") as f:
    anim, frame_lengths = pkl.load(f)
    print(f"Loaded animation! Animation has shape:{anim.shape}")

canvas5.main_render_buffer.full_screen_vertex_buffer.update_vertex_buffer(np.zeros((anim.
↪shape[0]*6), dtype=np.float32))

anim = np.swapaxes(anim, 0, 1)
# Dcelare textures, make sure that these textures as treated as ints instead of floats
anim_tex = canvas5.texture(anim, "uAnimTex", treat_as_normalized_integer=False)
frame_lengths_tex = canvas5.texture(frame_lengths, "uFrameLengthsTex", treat_as_
↪normalized_integer=False)
# Setup texture samplers
anim_tex.repeat_x, anim_tex.repeat_y = False, False
anim_tex.linear_filtering = False
frame_lengths_tex.repeat_x, frame_lengths_tex.repeat_y = False, False
frame_lengths_tex.linear_filtering = False

canvas5.shader("""
#pragma SSV point_cloud mainPoint --non_square_points
// These are automatically declared by the preprocessor
//uniform isampler2D uAnimTex;
//uniform isampler2D uFrameLengthsTex;
VertexOutput mainPoint()
{
    VertexOutput o;
    // Synchronise the playback to the time uniform, 30 FPS
    int frame = int(uTime*30.-20.);

    int frameLen = texelFetch(uFrameLengthsTex, ivec2(0, frame), 0).r;
    if(gl_VertexID > frameLen)
    {
        // Early out for verts not needed in this frame; no geometry will be generated␣
↪for these as the size is set to 0
        o.size = vec2(0.);
        return o;
    }
    // This contains the data for the current quad to rendered (value (0-255), x␣
↪(pixels), y (pixels), subdivision (0-n))
    ivec4 quad = texelFetch(uAnimTex, ivec2(gl_VertexID, frame), 0);
    // The size is determined by the subdivision level of the cell in the quad tree.
    o.size = vec2(1./pow(2., quad.w-0.1));
    if(quad.w == 0)
        o.size = vec2(0.);
    vec4 pos = vec4(float(quad.z)/480., 1.-float(quad.y)/360., 0., 1.);
    pos.xy += o.size/vec2(2., -2.);  // Centre the point
    pos = pos*2.-1.;  // To clip space (-1 to 1)
    pos += vec4(in_vert, 0.)*1e-8;  // If in_vert is not used, the shader compiler␣
```

(continues on next page)

(continued from previous page)

```
↪optimises it out which makes OpenGL unhappy; this may be fixed in the future
    o.position = pos;
    o.color = vec4(vec3(float(quad.x)/255.0)+in_color, 1.0);
    return o;
}
""")
canvas5.run()
```

Geometry shaders

This shader demonstrates the use of custom geometry shaders to render a vector field.

[ ]:
```
import pySSV as ssv
import numpy as np

# Generate some points
def generate_points():
    width, depth = 64, 64
    scale = 3
    v_scale = 0.5
    f = 0.01
    verts = np.zeros((width, depth, 9), dtype='f4')
    for z in range(depth):
        for x in range(width):
            dx = width/2 - x
            dz = depth/2 - z
            y = np.sin((dx*dx+dz*dz)*f) * v_scale
            # Pos
            verts[z, x, :3] = [x/width * scale, y, z/depth * scale]
            # Colour
            verts[z, x, 3:6] = [y/v_scale, abs(y/v_scale), np.sin(y/v_scale*10.)*0.5+0.5]
            # Direction
            verts[z, x, 6:9] = [dx/width, 0.1, dz/depth]

    return verts.flatten()

canvas5 = ssv.canvas(use_renderdoc=True)
# Set the contents of default vertex buffer on the main pass (normally used for full-
↪screen shaders, but in this case hijacked for this example)
canvas5.main_render_buffer.full_screen_vertex_buffer.update_vertex_buffer(generate_
↪points(), ("in_vert", "in_color", "in_dir"))
canvas5.main_camera.target_pos = np.array((1.5, 0, 1.5))
#print(canvas5.dbg_preprocess_shader("""
canvas5.shader("""
#pragma SSV geometry mainPoint mainGeo --vertex_output_struct VertexOutput --geo_max_
↪vertices 7 --custom_vertex_input
struct VertexOutput {
    vec4 position;
    vec4 color;
    vec3 dir;
    float size;
```

(continues on next page)

(continued from previous page)

```
};

#ifdef SHADER_STAGE_VERTEX
in vec3 in_vert;
in vec3 in_color;
in vec3 in_dir;

VertexOutput mainPoint()
{
    VertexOutput o;
    vec4 pos = vec4(in_vert, 1.0);
    //pos = uViewMat * pos;
    //pos = uProjMat * pos;
    o.position = pos;
    o.color = vec4(in_color, 1.);
    o.size = 30.0/uResolution.x;
    o.dir = normalize(in_dir);
    return o;
}
#endif

#ifdef SHADER_STAGE_GEOMETRY
void mainGeo(VertexOutput i) {
    vec4 position = i.position;
    float size = i.size;
    // This output variable is defined by the template and must be written to before the␣
↪first EmitVertex() call to take effect
    out_color = i.color;
    vec3 fwd = normalize((uViewMat * vec4(0., 0., 1., 0.)).xyz);
    vec3 perp = normalize(cross(i.dir, fwd));
    vec4 aspect_ratio = vec4(1., uResolution.x/uResolution.y, 1., 1.);
    float baseWidth = 0.05;
    float headWidth = 0.2;
    float headLength = 0.4;
    // Now we draw an arrow
    // Base
    out_color = vec4(0.,0.,0.,1.);
    gl_Position = position + size * vec4(perp*baseWidth, 0.0) * aspect_ratio;
    gl_Position = uProjMat * uViewMat * gl_Position;
    EmitVertex();
    gl_Position = position + size * vec4(-perp*baseWidth, 0.0) * aspect_ratio;
    gl_Position = uProjMat * uViewMat * gl_Position;
    EmitVertex();
    out_color = i.color;
    gl_Position = position + size * vec4(i.dir + perp*baseWidth, 0.0) * aspect_ratio;
    gl_Position = uProjMat * uViewMat * gl_Position;
    EmitVertex();
    gl_Position = position + size * vec4(i.dir - perp*baseWidth, 0.0) * aspect_ratio;
    gl_Position = uProjMat * uViewMat * gl_Position;
    EmitVertex();
    EndPrimitive();
    // Head
```

(continues on next page)

(continued from previous page)

```
    gl_Position = position + size * vec4(i.dir + perp*headWidth, 0.0) * aspect_ratio;
    gl_Position = uProjMat * uViewMat * gl_Position;
    EmitVertex();
    gl_Position = position + size * vec4(i.dir + -perp*headWidth, 0.0) * aspect_ratio;
    gl_Position = uProjMat * uViewMat * gl_Position;
    EmitVertex();
    gl_Position = position + size * vec4(i.dir * (1.+headLength), 0.0) * aspect_ratio;
    gl_Position = uProjMat * uViewMat * gl_Position;
    EmitVertex();
    EndPrimitive();
}
#endif
""")#)
canvas5.run(stream_quality=100)
```

Streaming Modes

pySSV supports a number of different video streaming modes to get rendered frames from the OpenGL backend into Jupyter. They each have their own advantages and disadvantages, so you can experiment with which method works best for you. `JPG` should be supported everywhere, but if your platform supports it, I would recommend `VP8` or `MJPEG`.

Not all streaming modes are supported on all platforms. Google Colab is notoriously difficult to get working nicely.

Here we present a particularly difficult example for video encoders, a point cloud (taken from the introduction.ipynb notebook) and how the different encoding settings affect it.

The following streaming modes are supported:

- JPG
- PNG
- VP8
- VP9
- H264
- MJPEG

The streaming mode is controlled using the `stream_mode` parameter of the `canvas.run()` method which accepts a `str` or an `SSVStreamingMode` (`from pySSV.ssv_render_process_server import SSVStreamingMode`). The `run()` method also takes a `stream_quality` parameter which can be used to control the compression of the encoder. It accepts a value from 0-100 (some encoders will work with values greater than 100, others clamp it) which, depending on the encoder, is scaled to give the constant bit rate or quality factor. Higher values give better quality images at the cost of higher bandwidth utilisation. When the `stream_quality` is above or equal to 90, chroma subsampling is disabled for formats that support yuv444p.

Technical Details

Internally, *pySSV* opens a dedicated websocket with the Jupyter frontend to stream video. On platforms where this isn't supported (notably, Google Colab) this falls back to using Jupyter Widget messages which are a bit less efficient due to the protocol's need to json encode everything. The `MJPEG` format is an exception to this as it communicates using a local HTTP server, relying on the browser's native support for `MJPEG` over HTTP; this has the advantage that `MJPEG` frames don't need to be json encoded or parsed in JS which helps a lot with latency.

The image formats `JPG` and `PNG` are encoded using Pillow as base64 encoded data URLs which are passed to an `<img>`. Whereas as the video formats are encoded by libavformat (FFmpeg's encoding library) and decoded in javascript using the WebCodecs API and blitted to a canvas; hence the lack of support for Firefox for these formats. `MJPEG` is encoded by libavformat and passed directly as a URL to the local HTTP server to an `<img>`.

[ ]:
```
import pySSV as ssv
import numpy as np

# Make the canvas size a bit bigger to put a bit more pressure on the encocers
CANVAS_SIZE = (1280, 720)

# Generate some points
def generate_points():
    width, depth = 64, 64
    scale = 3
    v_scale = 0.5
    f = 0.01
    verts = np.zeros((width, depth, 6), dtype='f4')
    for z in range(depth):
        for x in range(width):
            dx = width/2 - x
            dz = depth/2 - z
            y = np.sin((dx*dx+dz*dz)*f) * v_scale
            verts[z, x, :3] = [x/width * scale, y, z/depth * scale]
            verts[z, x, 3:6] = [y/v_scale, abs(y/v_scale), np.sin(y/v_scale*10.)*0.5+0.5]

    return verts.flatten()

def make_canvas():
    canvas = ssv.canvas(use_renderdoc=True, size=CANVAS_SIZE)
    # Set the contents of default vertex buffer on the main pass (normally used for full-
↪screen shaders, but in this case hijacked for this example)
    canvas.main_render_buffer.full_screen_vertex_buffer.update_vertex_buffer(generate_
↪points())
    canvas.main_camera.target_pos = np.array((1.5, 0, 1.5))
    canvas.shader("""
    #pragma SSV point_cloud mainPoint
    VertexOutput mainPoint()
    {
        VertexOutput o;
        vec4 pos = vec4(in_vert, 1.0);
        pos = uViewMat * pos;
        pos = uProjMat * pos;
        o.position = pos;
        o.color = vec4(in_color, 1.);
```

(continues on next page)

(continued from previous page)

```
        o.size = 30.0/uResolution.x;
        float d = length(uMouse/uResolution.xy*2.-1.-pos.xy/pos.z);
        o.size += clamp(pow(smoothstep(.5, 0., d), 3.)*0.03, 0., 0.3);
        o.color += step(d, o.size);
        return o;
    }
    """)
    return canvas
```

The default settings when `canvas.run()` is called are `stream_mode="jpg"` and `stream_quality=75`. When `stream_quality` is unset it defaults to the encoder's default quality.

[ ]:
```
make_canvas().run()
```

[ ]:
```
# For JPG streams, setting the stream quality to 100 can actually *improve* encoding␣
↪performance (if not limited by bandwidth) as some of the optimisations can be skipped.
make_canvas().run(stream_quality=100)
```

[ ]:
```
# For PNG is always lossless so the stream quality can't be controlled.
# This format is currently the only one which supports transparency in the output.
# It's also VERY slow to encode.

# With streaming formats the produce very large frames, such as png, Jupyter/the web␣
↪browser can
# get backed up with frames, in this case the frame rate may still be reasonable, but␣
↪extremely
# high latency (and memory usage!) will be apparent. In this case you need to switch to␣
↪a streaming
# format that offers more compression or decrease the streaming quality.

make_canvas().run(stream_mode="png")
```

Video Formats

[ ]:
```
# VP8 offers a good balance between quality and encoding time while offering very good␣
↪compression
# Latency is also generally fairly low
make_canvas().run(stream_mode="vp8", stream_quality=100)
```

[ ]:
```
# VP9 has improved compression efficiency but is much slower at encoding
make_canvas().run(stream_mode="vp9", stream_quality=100)
```

[ ]:
```
# H264 is fast to encode, but the compressions isn't quite as efficient as VP8
make_canvas().run(stream_mode="h264", stream_quality=100)
```

[ ]:
```
# MJPEG has very low latency and fast encoding/decooding time, but worse compression␣
↪efficiency than other video formats.
make_canvas().run(stream_mode="mjpeg", stream_quality=100)
```

Heightmap Demo

This example downloads DEM (digital elevation model) data from an online API and renders it in 3d using a shader.

The DEM data in question is derived from the SRTM1 (https://www2.jpl.nasa.gov/srtm/) dataset which covers most of the world (from 56°S to 60°N) at a resolution of 1 arc second (roughly 30m at the equator). Voids, trees, and buildings in the dataset have been removed.

The API returns a single tile which covers 1 degree by 1 degree. Special thanks to Adam Mathieson (https://github.com/amathieson) for hosting and maintaining this API, it is only to be used for the purpose of this demo.

The colouring of the data is purely for aesthetic interest.

[ ]:
```
# Select a point of interest
lat, lon = 46.2059915, 6.1475919  # Geneva, Switzerland
poi_name = "Geneva"

# Glasgow doesn't work very well...
# lat, lon = 55.8579612,-4.2582393  # Glasgow, Scotland
# poi_name = "Glasgow Central Station"

lat, lon = 35.36283,138.7312618  # Mount Fuji, Japan
poi_name = "Mount Fuji"
```

[ ]:
```
import math
import zlib
from PIL import Image
import os.path
import numpy as np
import pySSV as ssv

# Download the needed DEM tiles
# This snippet is derived from code written by Adam Mathieson, reused with permission
api_url = "https://cdn.whats-that-mountain.site/"
def latlon2ne(lat, lon):
    return f"{'s' if lat<0 else 'n'}{abs(math.floor(lat)):02d}{'w' if lon<0 else 'e'}
↪{abs(math.floor(lon)):03d}"

def get_tile(lat, lon):
    tile_name = f"{latlon2ne(lat, lon)}.hgt.gz"
    if not os.path.isfile(tile_name):
        import urllib.request
        try:
            print(f"Downloading {api_url + tile_name}...")
            opener = urllib.request.URLopener()
            opener.addheader('User-Agent', 'python-ssv-demo')
            opener.retrieve(api_url + tile_name, tile_name)
            print("Done!")
        except Exception as e:
            print(f"Failed to heightmap tile: {e}")
    return tile_name

corners = [
    (lat+.5, lon-.5),  # top left
    (lat+.5, lon+.5),  # top right
```

(continues on next page)

(continued from previous page)

```
    (lat-.5, lon-.5),  # bottom left
    (lat-.5, lon+.5),  # bottom right
]

# Open and decompress the tile
tile_name = get_tile(lat, lon)
with open(tile_name, "rb") as f:
    data = zlib.decompress(f.read())
    tile = np.array(np.frombuffer(data, np.int16))
    tile = tile.byteswap(inplace=True)
    tile = tile.reshape((3601, 3601))
    # print(tile)
```

[ ]:
```
# Create a new canvas and shader to render the tile
canvas = ssv.canvas(use_renderdoc=True)
tex = canvas.texture(tile, "uHeightMap", force_2d=True)
tex.repeat_x, tex.repeat_y = True, True
tex.linear_filtering = True

# In this example we use a simple SDF to view the heightmap in 3D
canvas.shader("""
#pragma SSV sdf sdf_main --render_mode SOLID --camera_mode INTERACTIVE

float sdf_main(vec3 p) {
    // Sample the heightfield and scale it as desired
    float disp = texture(uHeightMap, fract(p.xz*0.1+.5)).r*5.+.3;
    // Return the signed distance to the surface. Note that this is not an exact SDF and␣
↪fails anywhere where
    // the steepness exceeds 67.5deg. The final *.5, scales the distance field such that␣
↪steeper angles can
    // be used (67.5deg instead of 45deg) at the cost of more marching steps be required.
    return (p.y-disp)*.5;
}
""")
canvas.run()
```

Now we try rendering the heightfield as a mesh:

[ ]:
```
# Generate the mesh vertices
res = 3601
res = res//4  # Reduce vertex count for performance

# Make a grid of x, y, r, g, b
lin = np.linspace(0, 1, res)
zeros = np.zeros((res, res))
x,y = np.meshgrid(lin, lin)
verts = np.dstack((x,y,zeros, zeros,zeros))
# print(verts.shape)

# Define the triangles for the grid
inds = np.arange(0, res*(res-1))  # Create an array of indexes (skipping the last row of␣
↪vertices)
```

(continues on next page)

(continued from previous page)

```
inds = inds[(inds+1)%res!=0]  # Now skip the last column of vertices
inds = np.dstack((inds, inds+1, inds+res, inds+1, inds+res+1, inds+res))  # Now create␣
↪the indices for a quad for each point. A quad is defined by the indices (n, n+1, n+w,␣
↪n+1, n+w+1, n+w)
# print(inds)

inds = np.array(inds.flatten(), dtype=np.int32)
verts = np.array(verts.flatten(), dtype=np.float32)
```

[ ]:
```
# Create a new canvas and shader to render the tile
canvas = ssv.canvas((1280, 720), use_renderdoc=True)

# Bind the texture
tex = canvas.texture(tile, "uHeightMap", force_2d=True)
tex.repeat_x, tex.repeat_y = False, False
tex.linear_filtering = True

# Update the vertex buffer with a grid of vertices
vb = canvas.main_render_buffer.vertex_buffer()
vb.update_vertex_buffer(verts, index_array=inds)

# Create a GUI to interact with the example, see the gui_examples notebook for more info␣
↪on using GUIs
from pySSV.ssv_gui import create_gui, SSVGUI
from pySSV import ssv_colour
class MyGUI:
    slider_vertical_scale = 2.2
    slider_sun_p = 50.
    slider_sun_h = 180.
    slider_snow = 40.
    slider_dbg = 1.

    def on_gui_draw(self, gui: SSVGUI):
        gui.begin_vertical(pad=True)
        gui.rounded_rect(ssv_colour.ui_base_bg, overlay_last=True)
        gui.button("pySSV DEM Terrain Demo", ssv_colour.orange)
        self.slider_vertical_scale = gui.slider(f"Vertical scale: {float(self.slider_
↪vertical_scale):.3f}",
                                                self.slider_vertical_scale, min_value=0,␣
↪max_value=50, power=3.)
        self.slider_sun_p = gui.slider(f"Sun pitch: {float(self.slider_sun_p):.3f}",
                                       self.slider_sun_p, min_value=0, max_value=90)
        self.slider_sun_h = gui.slider(f"Sun heading: {float(self.slider_sun_h):.3f}",
                                       self.slider_sun_h, min_value=0, max_value=360)
        self.slider_snow = gui.slider(f"Snow height: {float(self.slider_snow):.3f}",
                                      self.slider_snow, min_value=0, max_value=100,␣
↪power=3.)
        self.slider_dbg = gui.slider(f"Debug: {float(self.slider_dbg):.3f}",
                                      self.slider_dbg, min_value=0, max_value=10,␣
↪power=3.)

        gui.space(height=30)
```

(continues on next page)

(continued from previous page)

```
        gui.end_vertical()
        horiz_scale = 2.
        x, z = ((lat-math.floor(lat))*2.-1.)*horiz_scale, -((lon-math.floor(lon))*2.-1.
↪)*horiz_scale
        # print(z, x)
        gui.label_3d(poi_name, (x, 0.025, z), font_size=12., shadow=True)

    def on_post_gui(self, gui: SSVGUI):
        gui.canvas.update_uniform("uVerticalScale", float(self.slider_vertical_scale))
        gui.canvas.update_uniform("uSunPitch", float(self.slider_sun_p))
        gui.canvas.update_uniform("uSunHeading", float(self.slider_sun_h))
        gui.canvas.update_uniform("uSnowHeight", float(self.slider_snow))
        gui.canvas.update_uniform("uDebug", float(self.slider_dbg))

gui = create_gui(canvas)
my_gui = MyGUI()
# Register a callback to the on_gui event
gui.on_gui(lambda x: my_gui.on_gui_draw(x))
gui.on_post_gui(lambda x: my_gui.on_post_gui(x))
canvas.update_uniform("uVerticalScale", my_gui.slider_vertical_scale)

# Create a shader to render the sky
canvas.shader("""
#pragma SSV pixel pixel

uniform float uSunPitch;
uniform float uSunHeading;
uniform float uDebug;

const mat3 xyzToSrgb = mat3 (
     3.2410032329763605, -0.96922425220251640,  0.05563941985197549,
    -1.53739896948878640,  1.87592998369517530, -0.20401120612391013,
    -0.49861588199636320,  0.04155422634008475,  1.05714897718753330
);

vec3 reinhard2(vec3 x) {
    const float L_white = 4.0;
    return (x * (1.0 + x / (L_white * L_white))) / (1.0 + x);
}

vec4 pixel(vec2 fragCoord) {
    vec2 uv = (fragCoord/uResolution.xy)*2.-1.;
    vec3 eye = normalize((vec4(uv, -1., 0.) * uViewMat).xyz);

    float sp = cos(radians(uSunPitch));
    vec3 sun = vec3(cos(radians(uSunHeading))*sp, sin(radians(uSunPitch)),␣
↪sin(radians(uSunHeading))*sp);
    float mie = pow(max(dot(eye, sun), 0.), 4.);

    float p = (1.-sp)*0.5+0.25;
    vec2 g = pow(.8-max(eye.yy, 0.)*0.5, vec2(3.0+p*0.1, 3.0));
    vec3 col = xyzToSrgb * vec3(g.x, g.y, pow((1.-abs(eye.y)), 2.)*p+p);
```

(continues on next page)

(continued from previous page)

```
    col += vec3(0.95, 0.93, 0.9) * mie*0.5;
    col = smoothstep(0., 1., reinhard2(col)*1.8);

    return vec4(col, 1.);
}
""")

# Create a shader to render the terrain
vb.shader("""
#pragma SSV vert_pixel vert pixel

uniform float uVerticalScale;
uniform float uSunPitch;
uniform float uSunHeading;
uniform float uSnowHeight;
uniform float uDebug;

const float horizScale = 2.;
const float texelSize = 1./3601.;

#ifdef SHADER_STAGE_VERTEX
layout(location = 3) out vec2 uv;

void vert() {
    uv = in_vert.xy;
    uv = 1.-uv;
    uv = uv.yx;
    float disp = texture(uHeightMap, uv.xy).r;
    vec2 v = (in_vert.xy*2.-1.)*horizScale;
    gl_Position = vec4(v.x, disp*uVerticalScale, v.y, 1.);
    gl_Position = uProjMat * uViewMat * gl_Position;
}
#endif

#ifdef SHADER_STAGE_FRAGMENT
layout(location = 3) in vec2 uv;

float radians(float x) {
    return x/180.*3.14159265;
}

vec3 reinhard2(vec3 x) {
    const float L_white = 4.0;
    return (x * (1.0 + x / (L_white * L_white))) / (1.0 + x);
}

// BRDF functions taken from:
// https://www.shadertoy.com/view/XlKSDR
float pow5(float x) {
    float x2 = x*x;
    return x2*x2*x;
}
```

(continues on next page)

(continued from previous page)

```
float dGGX(float linearRoughness, float ndoth, const vec3 h) {
    // Walter et al. 2007, "Microfacet Models for Refraction through Rough Surfaces"
    float oneMinusNoHSquared = 1.0 - ndoth * ndoth;
    float a = ndoth * linearRoughness;
    float k = linearRoughness / (oneMinusNoHSquared + a * a);
    float d = k * k * (1.0 / 3.141592);
    return d;
}
float vSmithGGXCorrelated(float linearRoughness, float NoV, float NoL) {
    // Heitz 2014, "Understanding the Masking-Shadowing Function in Microfacet-Based␣
↪BRDFs"
    float a2 = linearRoughness * linearRoughness;
    float GGXV = NoL * sqrt((NoV - a2 * NoV) * NoV + a2);
    float GGXL = NoV * sqrt((NoL - a2 * NoL) * NoL + a2);
    return 0.5 / (GGXV + GGXL);
}
vec3 fSchlick(const vec3 f0, float VoH) {
    // Schlick 1994, "An Inexpensive BRDF Model for Physically-Based Rendering"
    return f0 + (vec3(1.0) - f0) * pow5(1.0 - VoH);
}

vec4 pixel(vec3 pos) {
    float h = texture(uHeightMap, uv).r;
    // Compute the normals by finite differences
    vec3 nrm = vec3(0.);
    nrm.z = h - texture(uHeightMap, uv - vec2(texelSize, 0.)).r;
    nrm.x = h - texture(uHeightMap, uv - vec2(0., texelSize)).r;
    nrm.xz *= uVerticalScale/horizScale/texelSize/2.;
    nrm.y = 1.;//-sqrt(nrm.x*nrm.x + nrm.z*nrm.z);
    nrm = normalize(nrm);

    // Lighting
    float sp = cos(radians(uSunPitch));
    vec3 sun = vec3(cos(radians(uSunHeading))*sp, sin(radians(uSunPitch)),␣
↪sin(radians(uSunHeading))*sp);
    vec2 screenPos = (gl_FragCoord.xy/uResolution.xy*2.-1.) * vec2(1., uResolution.y/
↪uResolution.x);
    vec3 eye = normalize((vec4(screenPos, 1., 0.) * uViewMat).xyz);
    float ndotl = dot(nrm, sun);
    float ndotv = dot(nrm, eye);
    vec3 half = normalize(eye + sun);
    float ndoth = clamp(dot(nrm, half), 0., 1.);
    vec3 sunCol = pow(vec3(1., 0.9, 0.75), vec3(sp*sp*sp*2.5+1.));

    // Texturing
    const vec3 water = vec3(0, 40./255., 49./255.)*1.2;
    const vec3 city = vec3(101./255, 115./255., 88./255.)*1.2;
    const vec3 forest = vec3(10./255., 42./255., 24./255.)*1.2;
    const vec3 cliff = vec3(214./255., 220./255., 173./255.)*.5;
    const vec3 snow = vec3(247./255., 247./255., 255./255.)*1.2;

    float water_city = smoothstep(11.034/1000., 12.052/1000., h);
```

(continues on next page)

(continued from previous page)

```
    float city_forest = smoothstep(10.4/1000., 24./1000., h);
    float forest_cliff = smoothstep(0.6, 0., abs(nrm.y));
    float cliff_snow = smoothstep((uSnowHeight)/1000., (45.+uSnowHeight)/1000., h) *␣
↪smoothstep(0.3, 0.5, abs(nrm.y));
    vec3 alb = water;
    alb = mix(alb, city, water_city);
    alb = mix(alb, forest, city_forest);
    alb = mix(alb, cliff, forest_cliff);
    alb = mix(alb, snow, cliff_snow);

    // Specular
    float rough = clamp((alb.g*0.6+water_city*0.9)*0.7*uDebug, 0., 1.);
    rough *= rough;
    float specD = dGGX(rough, ndoth, half);
    specD *= vSmithGGXCorrelated(rough, clamp(ndotv, 0., 1.), clamp(ndotl, 0., 1.));
    vec3 spec = vec3(specD);
    spec *= fSchlick(vec3(0.04), clamp(dot(eye, half), 0., 1.));

    // Composition
    vec3 col = alb;
    col *= (ndotl*.5+.5) * sunCol;
    col += clamp(spec, 0., 10.) * clamp(ndotl, 0., 1.) * sunCol;
    col *= 2.;
    col = smoothstep(0., 1., reinhard2(col)*1.2);
    //col = vec3(rough);

    return vec4(col, 1.);
}
#endif
""")
canvas.run(stream_quality=90)
```

[ ]:

[ ]:

1.5.2 pySSV GUI Library Examples

pySSV includes a powerful and easy to use immediate mode GUI library to create interactive user interfaces for your shaders.

Check out the documentation for more details about the GUI library: SSVGUI Documentation

[ ]:
```
# Google colab support
try:
    # Try enabling custom widgets, this will fail silently if we're not in Google Colab
    from google.colab import output
    output.enable_custom_widget_manager()
    # Install pySSV for this session
    %pip install pySSV
```

(continues on next page)

(continued from previous page)

```
except:
    pass
```

[ ]:
```
import pySSV as ssv
from pySSV.ssv_gui import create_gui, SSVGUI
from pySSV import ssv_colour

class MyGUI:
    # We create a new class for the GUI so that state can be stored between on_gui_draw␣
↪calls
    checkbox_value = False
    slider_font_size = 10.
    slider_font_weight = 0.4
    slider_rounding = 4
    slider_int = 0
    slider_non_lin = 0

    def on_gui_draw(self, gui: SSVGUI):
        # Begin drawing our GUI in here.
        # This is an immediate mode gui, so any time the GUI is invalidated (on mouse␣
↪click for instance), we redraw the entire GUI.
        # The GUI drawing methods used here should *only* be used within the on_gui␣
↪callback.
        gui.begin_vertical(pad=True)
        gui.rounded_rect(ssv_colour.ui_base_bg, overlay_last=True)
        gui.button("Hello world!")
        gui.button("GUIs are Awesome", ssv_colour.orange)

        gui.begin_horizontal()
        gui.rounded_rect(ssv_colour.violet)
        gui.rounded_rect(ssv_colour.darkgrey)
        gui.end_horizontal()

        gui.label("Shaders are cool!", font_size=float(self.slider_font_size), x_
↪offset=5, shadow=True, overlay_last=True)

        gui.begin_horizontal()
        gui.button("Horizontal", ssv_colour.red)
        gui.button("Layout", ssv_colour.green)
        gui.button("Groups", ssv_colour.blue)
        gui.end_horizontal()

        gui.space()
        self.checkbox_value = gui.checkbox("We can enable layout groups", self.checkbox_
↪value)
        gui.begin_toggle(self.checkbox_value)
        gui.button("This control can be hidden", ssv_colour.olive)
        gui.button("Peekaboo", ssv_colour.emeraldgreen)
        gui.end_toggle()
        gui.button("Controls are automatically layed out", ssv_colour.navy)
        gui.label("Text can be styled", font_size=22., x_offset=5, colour=ssv_colour.
↪orange)
```

(continues on next page)

(continued from previous page)

```
        gui.label("Text can be styled", font_size=30., x_offset=5, colour=ssv_colour.
↪white, weight=float(self.slider_font_weight))
        gui.label("Text can be styled", font_size=30., x_offset=5, colour=ssv_colour.
↪limegreen, weight=0.75)
        gui.label("Text can be styled", font_size=16., x_offset=5, colour=ssv_colour.
↪cyan, italic=True, shadow=True)

        self.slider_font_size = gui.slider("Sliders!", self.slider_font_size, min_
↪value=0, max_value=100, step_size=1, power=1., track_thickness=4)
        self.slider_font_weight = gui.slider("Font weight", self.slider_font_weight, min_
↪value=0.25, max_value=0.8)
        self.slider_rounding = gui.slider(f"Corner radius = {float(self.slider_rounding):
↪.2f}", self.slider_rounding, min_value=0, max_value=20)
        self.slider_int = gui.slider(f"Integer slider = {self.slider_int}", self.slider_
↪int, min_value=0, max_value=100, step_size=20, track_thickness=10)
        self.slider_non_lin = gui.slider(f"Nonlinear slider = {float(self.slider_non_
↪lin):.2f}", self.slider_non_lin, min_value=0, max_value=100, power=2., track_
↪thickness=10)
        gui.rounding_radius = self.slider_rounding.result

        gui.space(height=30)

        gui.end_vertical()

# Set up a basic canvas
canvas = ssv.canvas(use_renderdoc=True)
# Now create an SSVGUI
gui = create_gui(canvas)
my_gui = MyGUI()
# Register a callback to the on_gui event
gui.on_gui(lambda x: my_gui.on_gui_draw(x))
# Now we render a shader as usual
canvas.shader("""
#pragma SSV shadertoy
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/iResolution.yy;
    // Colour changing over time
    vec3 col = sin(uv.xyx + iTime * vec3(3, 4, 5)) * 0.5 + 0.5;
    float alpha = smoothstep(0.1, 0.1+2./iResolution.y, 1.-length(uv*2.-1.));
    // Output to screen
    fragColor = vec4(vec3(col), 1.0);
}
""")
canvas.run(stream_quality=100)
```

[ ]:

1.5.3 Introduction

This notebook demonstrates a few example uses for pySSV and includes examples for many of the supported features.

In these first few examples we demonstrate some basic fragment shaders.

[ ]:
```
# Google colab support
try:
    # Try enabling custom widgets, this will fail silently if we're not in Google Colab
    from google.colab import output
    output.enable_custom_widget_manager()
    # Install pySSV for this session
    %pip install pySSV
except:
    pass
```

[ ]:
```
import pySSV as ssv
import logging
ssv.ssv_logging.set_severity(logging.INFO)
```

[ ]:
```
# Create a new SSVCanvas, the canvas is responsible for managing the OpenGL context, the␣
↪render widget, and the state of the renderer.
canvas = ssv.canvas()
# Check what graphics adapter we're using
canvas.dbg_log_context()
# Set up a very basic shader program to check it's working
canvas.shader("""
#pragma SSV shadertoy
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/iResolution.yy;
    // Colour changing over time
    vec3 col = sin(uv.xyx + iTime * vec3(3, 4, 5)) * 0.5 + 0.5;
    float alpha = smoothstep(0.1, 0.1+2./iResolution.y, 1.-length(uv*2.-1.));
    // Output to screen
    fragColor = vec4(vec3(col), alpha);
}
""")
```

[ ]:
```
# run() starts the render loop, it will continuously render frames until stop() is␣
↪called or the widget is destroyed.
# We set the stream mode to png here as it supports transparency. In general though, jpg␣
↪(the default) is much faster.
canvas.run(stream_mode="png")
```

[ ]:
```
canvas.stop()
```

Mouse input

Here's a basic example of a shader that makes use of mouse position. With the `dbg_shader()` method, glsl code is generated around your shader to support ShaderToy-like shaders. In this case the canvas resolution is passed is in as `iResolution` and the mouse position as `iMouse`.

```
[ ]: import pySSV as ssv
canvas1 = ssv.canvas()
canvas1.shader("""
#pragma SSV pixel mainImage
vec4 mainImage( in vec2 fragCoord )
{
    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/uResolution.xx;
    float aaScale = 1./uResolution.x;

    vec2 mouse = uv-uMouse.xy / uResolution.xx;

    // Time varying pixel color
    vec3 col = vec3(smoothstep(0.9, .95, 1.-length(mouse)));
    col -= 1.-vec3(step(dot(step(abs(mouse), vec2(0.8/uResolution.x, 5./uResolution.x)),␣
↪vec2(0.5)), 0.5));
    col -= 1.-vec3(step(dot(step(abs(mouse), vec2(5./uResolution.x, 0.8/uResolution.x)),␣
↪vec2(0.5)), 0.5));

    // Output to screen
    return vec4(vec3(col), 1.0);
}
""")
canvas1.run()
```

Here's a more complex shader taken almost directly from ShaderToy.

```
[ ]: canvas2 = ssv.canvas()
canvas2.shader("""
#pragma SSV shadertoy
// Copyright Thomas Mathieson all rights reserved
// https://www.shadertoy.com/view/DsffWM
const float motionBlur = 0.3;
const float aa = 0.6;
const vec3 col1 = vec3(13., 45., 140.)/100.;
const vec3 col2 = vec3(255., 20., 50.)/255.;
const vec3 col3 = vec3(21., 191., 112.)/600.;
const vec3 col4 = vec3(0.35, 1., 0.7)*0.65;
const float speed = 0.1;

float sigmoid(float x)
{
    return 1.*x/(abs(x)+1.);
}
vec3 sigmoid(vec3 x)
{
    return x/(abs(x)+vec3(1.));
}
```

(continues on next page)

(continued from previous page)

```
vec3 saturate(vec3 x)
{
    return clamp(x, 0., 1.);
}
vec3 blend(float x, vec3 c)
{
    c = pow(c, vec3(x+2.));
    return mix(x*c, x*(1.-c), step(x, 0.));
}

float f(vec2 p, float t, vec4 o, vec4 o1, float s, vec4 scale)
{
    vec4 i0 = cos(t+o)*vec4(o.xw, o1.xw);
    vec4 i1 = sin(t+o1)*vec4(o.xw, o1.xw);
    vec4 x0 = i0*s*sin(scale*length(p*o.xy+4.*scale.zw)+o.z+t*o.w);
    vec4 x1 = i1*s*sin(scale*length(p*o1.xy)+o1.z+t*o1.w);
    return sigmoid(dot(x0+x1, vec4(1.)));
}

vec3 scene(float t, float emphasis, vec2 uv)
{
    // "Beautiful" randomness, tuned for aesthetics, not performance
    vec2 p = uv * 3.;
    t += 160.;
    t *= speed;
    vec4 scale = vec4(sin(t*vec3(0.25, .5, .75)), cos(t*.95))*.25+.5;
    float s0 = f(p, t, vec4(6.,9.,2.,1.5), vec4(2.,9.,7.,3.), .25, scale);
    float s1 = f(p, t, vec4(2.,6.5,1.5,4.0), vec4(3.,2.5,3.8,1.6), .5, scale);
    float s2 = sigmoid(s0/s1)*0.5;
    float s3 = f(p, t, vec4(2.,9.,7.,3.), vec4(6.,3.,2.,1.5), .125, scale);
    float s6 = f(p*1.5, t, vec4(6.,4.,8.,2.5), vec4(3.2,1.6,9.7,7.9), .25, scale);
    float s7 = f(p*1.3, t, vec4(2.,6.5,1.5,4.0), vec4(3.,2.5,3.8,1.6), .5, scale);
    float s8 = sigmoid(s6/s7+s0)*0.7;

    vec3 c = vec3(sigmoid((blend(s8,col1)+blend(s2,col2)+blend(s1,col3)+s7*1.)*1.1)*.7+.
↪5);
    float grad = sigmoid(pow(length(uv*2.-1.)+s3*.3, 5.))*1.5;
    float accent = 1.-sigmoid((pow(2.5, abs(sigmoid(s8+s0+s1))-1.)-.45-(emphasis*0.
↪1))*1000./(1.+30.*grad+20.*emphasis));
    c = mix(c, c.r*.3+col4*.8, accent);
    return clamp(vec3(c), 0., 1.);
}

void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/iResolution.xx;
    float aaScale = 1./iResolution.x;

    vec2 mouse = uv-iMouse.xy /iResolution.xx;
    float emp = sigmoid(1./pow(length(mouse*1.), 1.8)*.02);
```

(continues on next page)

(continued from previous page)

```
    // Time varying pixel color
    vec3 col = scene(iTime, emp, uv);
    //col     += scene(iTime + motionBlur*0.001, emp, uv + aaScale*aa*vec2(0.,1.))
    //         + scene(iTime + motionBlur*0.002, emp, uv + aaScale*aa*vec2(1.,0.));
    //col /= 3.;

    // Output to screen
    fragColor = vec4(vec3(col), 1.0);
}
""")
canvas2.run(stream_quality=100)
```

Shader Templates

pySSV makes use of a shader templating system to reduce boilerplate. Many shader templates are provided but you can of course write your own (instructions for which are in the documentation). Shader templates can be just a thin layer glsl boilerplate or can contain significant a amounts of high level functionality as shown in the example below which uses the `sdf` template for signed distance field rendering. This template takes a distance function as an entrypoint and generates the renderer code.

Shader templates are specified using the `#pragma SSV <template_name> [template arguments...]` directive. Arguments are defined in the shader template and are specified similar to command line arguments (they are parsed by python's `argparse` module internally).

[ ]:
```
import pySSV as ssv
canvas3 = ssv.canvas()
canvas3.shader("""
#pragma SSV sdf sdf_main --camera_distance 2. --rotate_speed 1.5 --render_mode SOLID

// SDF taken from: https://iquilezles.org/articles/distfunctions/
float sdCappedTorus(vec3 p, vec2 sc, float ra, float rb) {
  p.x = abs(p.x);
  float k = (sc.y*p.x>sc.x*p.y) ? dot(p.xy,sc) : length(p.xy);
  return sqrt( dot(p,p) + ra*ra - 2.0*ra*k ) - rb;
}

float sdf_main(vec3 p) {
    float t = 2.*(sin(uTime)*0.5+0.5)+0.2;
    return sdCappedTorus(p, vec2(sin(t), cos(t)), 0.5, 0.2);
}
""")
canvas3.run(stream_quality=100)
```

Multi-Pass Rendering

pySSV provides support for multi-pass rendering and multiple draw calls within a pass.

The rendering system renders draw calls belonging to render buffers belonging to canvases (`SSVCanvas` -owns-> `SSVRenderBuffer` -owns-> `SSVVertexBuffer`). When you create an `SSVCanvas`, internally, it creates an `SSVRenderBuffer` which itself creates an `SSVVertexBuffer` to draw into. `canvas.shader()` is actually shorthand for `canvas.main_render_buffer.full_screen_vertex_buffer.shader()` since a shader must belong to an individual draw call (since vertex buffers and draw calls are linked 1-1, we use the terms interchangably here).

To start creating your own render buffers and draw calls use the following API:

[ ]:
```
import pySSV as ssv
import numpy as np

canvas4 = ssv.canvas(use_renderdoc=True)
# Create a new render buffer on this canvas
rb = canvas4.render_buffer(size=(640, 480), name="renderBuffer1")

# Now we can render full-screen shaders on both the main render buffer and our new␣
↪render buffer
### Draw diagonal stripes in the background, and composite renderBuffer1 on top
canvas4.shader("""
#pragma SSV pixel mainImage
vec4 mainImage(in vec2 fragCoord)
{
    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/uResolution.xy;
    // Some diagonal stripes for the background
    vec3 col = vec3(step(fract((fragCoord.x+fragCoord.y+uTime*50.)/50.0), 0.5));
    // Now blend the output of renderBuffer1 on top
    vec4 rb1 = texture(renderBuffer1, uv);
    col = mix(col, rb1.rgb, rb1.a);
    // Output to screen
    return vec4(vec3(col), 1.0);
}
""")
### Draw a circle wwith a colour changing gradient in renderBuffer1
rb.shader("#pragma SSV render_test")

# If we wanted to add another draw call to our new render buffer we would use the vertex_
↪buffer() method
vb = rb.vertex_buffer()
# Now we can populate this vertex buffer
vb.update_vertex_buffer(np.array([
    # X   Y     R    G    B
    -1.0, -1.0, 1.0, 0.0, 0.0,
    1.0, -1.0, 0.0, 1.0, 0.0,
    0.0, 1.0, 0.0, 0.0, 1.0],  # This should make a single triangle
    dtype='f4',
))
# And assign a shader to it's draw call
### Draw a colourful triangle on top of renderBuffer1
vb.shader("""
```

(continues on next page)

(continued from previous page)

```
#pragma SSV pixel mainImage
vec4 mainImage(in vec2 fragCoord)
{
    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/uResolution.xy;
    // Color from the vertex colours
    vec3 col = color.rgb;
    // Output to screen
    return vec4(vec3(col), 1.0);
}
""")

canvas4.run(stream_quality=100)
```

Camera

When you create a canvas, an `SSVCamera` is also created and attached to it. This automatically recevies input from the canvas and makes a view and projection matrix available in the shader to be used by shaders requiring 3d perspective transformations.

[ ]:
```
import pySSV as ssv
import numpy as np

canvas5 = ssv.canvas()
canvas5.main_render_buffer.full_screen_vertex_buffer.update_vertex_buffer(np.array([
    # X   Y     R    G    B
    -1.0, -1.0, 1.0, 0.0, 0.0,
    1.0, -1.0, 0.0, 1.0, 0.0,
    0.0, 1.0, 0.0, 0.0, 1.0],  # This should make a single triangle
    dtype='f4',
))
canvas5.shader(
"""
#pragma SSV vert mainVert
VertexOutput mainVert()
{
    VertexOutput o;
    vec4 pos = vec4(in_vert, 1., 1.0);
    pos = uViewMat * pos;
    pos = uProjMat * pos;
    o.position = pos;
    o.color = vec4(in_color, 1.);
    return o;
}
""")
# We can configure the camera settings using the `main_camera` field of the canvas
canvas5.main_camera.fov = 60
canvas5.run(stream_quality=100)
```

Data Input And Custom Textures

There are 3 ways of getting data into a shader for rendering:

- As vertex data
- As a uniform
- As a texture

Uniforms

Uniforms are the simplest to use, and we've already seen them in previous examples. Uniforms are great for small amounts of data which you might want to change frequently such as lighting paramaters, camera transformations, etc… There is a limit to the number of uniforms you can declare and the total amount of memory they consume, this limit depends on your platform but is generally big enough that you don't need to consider it; that is as long as you don't declare large (>1000 elements) arrays as uniforms.

To assign values from python to a uniform it must be in a compatible type. Uniforms generally only accept numeric data (ints and floats) as scalars, vectors (up to 4 components), or matrices (up to 4x4 components); any appropriately sized array-like python type can be assigned to a uniform vector. See https://www.khronos.org/opengl/wiki/Data_Type_(GLSL) for more details on supported GLSL types to use with uniforms.

[ ]:
```
import pySSV as ssv

canvas5 = ssv.canvas()
canvas5.shader("""
#pragma SSV pixel mainImage
uniform vec3 customColour;

vec4 mainImage(in vec2 fragCoord)
{
    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/uResolution.xy;
    // Color from the uniform
    // Some diagonal stripes
    vec3 col = vec3(step(fract((fragCoord.x+fragCoord.y+uTime*50.)/50.0), 0.5));
    // Now colour them with the uniform
    col *= customColour;
    // Output to screen
    return vec4(vec3(col), 1.0);
}
""")
canvas5.update_uniform("customColour", (1, 0, 1))  # Magenta
canvas5.run(stream_quality=100)
```

Vertex Data

For sparse spatial data, the vertex buffer is a good method of getting data into the shader.

Triangle meshes and point clouds are natural choices data which can use the vertex buffer. Vertices can contain any number of attributes (limited by the graphics backend) in addition to position, such as colour, normals, etc… Vertex data is processed in parallel in the vertex shader which is a good place to put any per-vertex transformations. The outputs of the vertex shader are then interpolated automatically into the fragment (pixel) shader.

You can define your own vertex data structures and specify how vertex attributes are bound in the shader using the `vertex_attributes` parameter in the `update_vertex_buffer()` method. To take full advantage of custom vertex_attribtues though you'll need to write your own shader template.

In the following example we generate a point cloud with the shape `(64, 64, 6)` (which is later flattened into 1D array). The vertex attributes in this example are `vec3 in_vert; vec3 in_color` this maps onto the 6 components in the point cloud. The `point_cloud` shader template used here, exposes an entrypoint in the vertex stage to perform transformations on the points themselves (in our case we perform the perspective transformation for the camera). The template then passes this to a geometry shader, which generates sprites (2-triangle primitives) for each vertex (use the `dbg_preprocess_shader()` described below to see the final shader generated by the template preprocessor to see how this works).

[ ]:
```
# Generate some points
def generate_points():
    width, depth = 64, 64
    scale = 3
    v_scale = 0.5
    f = 0.01
    verts = np.zeros((width, depth, 6), dtype='f4')
    for z in range(depth):
        for x in range(width):
            dx = width/2 - x
            dz = depth/2 - z
            y = np.sin((dx*dx+dz*dz)*f) * v_scale
            verts[z, x, :3] = [x/width * scale, y, z/depth * scale]
            verts[z, x, 3:6] = [y/v_scale, abs(y/v_scale), np.sin(y/v_scale*10.)*0.5+0.5]

    return verts.flatten()
```

[ ]:
```
import pySSV as ssv
import numpy as np

canvas5 = ssv.canvas(use_renderdoc=True)
# Set the contents of default vertex buffer on the main pass (normally used for full-
↪screen shaders, but in this case hijacked for this example)
canvas5.main_render_buffer.full_screen_vertex_buffer.update_vertex_buffer(generate_
↪points())
canvas5.main_camera.target_pos = np.array((1.5, 0, 1.5))
canvas5.shader("""
#pragma SSV point_cloud mainPoint
VertexOutput mainPoint()
{
    VertexOutput o;
    vec4 pos = vec4(in_vert, 1.0);
```

(continues on next page)

(continued from previous page)

```
    pos = uViewMat * pos;
    pos = uProjMat * pos;
    o.position = pos;
    o.color = vec4(in_color, 1.);
    o.size = 30.0/uResolution.x;
    return o;
}
""")
canvas5.run(stream_quality=100)
```

Textures

To get large amounts of data into a shader textures are ideal.

Most GPUs support 2D and 3D textures in a variety of formats (usually from 8, 16, and 32 bits per component, as floats of uints, up to 4 components per pixel (RGBA)); *pySSV* automatically attempts to determine the correct texture dimensions and format from a NumPy array. Different GPUs have different limitations as to the maximum dimensions (width, height, and depth) for textures which cannot be exceeded, for 2D textures 16384x16384 is usually limit, for 3D textures this is often smaller. That being said a 16384x16384 texture with 4x32 bit components per pixel represents 4 GB of memory.

Textures have a few useful features which can be exploited pretty much for free:

- Texture interpolation: textures can be sampled using nearest neighbour interpolation or bilinear / trilinear (for mipmaps or 3d textures), interpolation.
- Texture repetition: when sampling textures outside of the usual 0-1 range, they can be set to either repeat or clamp to edge values.
- Texture mipmaps: (only recommended for textures with power-of-2 dimensions) the GPU can efficiently generate image pyramids for textures which can be sampled with linear interpolation between levels in the shader. These can be used to approximate blurring operations, compute averages, or to prevent texture aliasing.

[ ]:
```
import pySSV as ssv
import numpy as np

canvas5 = ssv.canvas(use_renderdoc=True)
# Here we generate a simple 3x3 single component texture. By default, pySSV will attempt␣
↪to treat this as a 3x1
# texture with 3 components (since the height is less than the maximum number of␣
↪components in a texture (4)), as
# such we use the `force_2d` parameter to tell pySSV to treat the 2nd dimension of the␣
↪array as height instead of
# components.
texture = canvas5.texture(np.array([
    [0., 0.1, 0.2],
    [0.3, 0.4, 0.5],
    [0.6, 0.7, 0.8]
], dtype=np.float16), "uTexture1", force_2d=True)
# texture.linear_filtering = False
texture.repeat_x, texture.repeat_y = False, False
texture.linear_filtering = False
canvas5.shader("""
```

(continues on next page)

(continued from previous page)

```
#pragma SSV pixel mainImage
// Provided that the texture is declared on the canvas *before* the shader is, then it's␣
↪uniform will be
// automatically declared in the shader by the preprocessor.
//uniform sampler2D uTexture1;
vec4 mainImage(in vec2 fragCoord)
{
    // Normalized pixel coordinates (from 0 t 1)
    vec2 uv = fragCoord/uResolution.xy;
    // Color from the uniform
    // Some diagonal stripes
    vec3 col = vec3(step(fract((fragCoord.x+fragCoord.y+uTime*50.)/50.0), 0.5));
    // Now colour them with the texture
    col *= vec3(texture(uTexture1, uv).r);
    // Output to screen
    return vec4(vec3(col), 1.0);
}
""")
canvas5.run()
```

```
[ ]: if "canvas" in globals():
    canvas.stop()
if "canvas1" in globals():
    canvas1.stop()
if "canvas2" in globals():
    canvas2.stop()
if "canvas3" in globals():
    canvas3.stop()
if "canvas4" in globals():
    canvas4.stop()
if "canvas5" in globals():
    canvas5.stop()
if "canvas6" in globals():
    canvas6.stop()
```

Debugging Shaders

Shaders can get quite complex so pySSV provides a few tools to simplify debugging your shaders.

Preprocessor Dump

It can be helpful to view the GLSL generated by the pre processor to understand why things are going wrong:

```
[ ]: import pySSV as ssv
canvas1 = ssv.canvas()
shader = canvas1.dbg_preprocess_shader("""
#pragma SSV pixel mainImage
vec4 mainImage( in vec2 fragCoord )
```

(continues on next page)

(continued from previous page)

```
{
    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/uResolution.xx;
    float aaScale = 1./uResolution.x;

    vec2 mouse = uv-uMouse.xy / uResolution.xx;

    // Time varying pixel color
    vec3 col = vec3(smoothstep(0.9, .95, 1.-length(mouse)));
    col -= 1.-vec3(step(dot(step(abs(mouse), vec2(0.8/uResolution.x, 5./uResolution.x)),␣
↪vec2(0.5)), 0.5));
    col -= 1.-vec3(step(dot(step(abs(mouse), vec2(5./uResolution.x, 0.8/uResolution.x)),␣
↪vec2(0.5)), 0.5));

    // Output to screen
    return vec4(vec3(col), 1.0);
}
""")
print(shader)
```

List Shader Templates

You can also get a list of all the installed shader templates.

[ ]:
```
print(canvas1.dbg_query_shader_templates(additional_template_directory=None))
```

Query Shader Template Arguments

And you can query a shader template for it's arguments.

[ ]:
```
print(canvas1.dbg_query_shader_template("sdf"))
```

OpenGL Context

If you're trying to track down a driver bug or platform specific oddity, having the graphics adapter information can be helpful

[ ]:
```
canvas1.dbg_log_context(full=True)
```

Frame Times

pySSV also provides rudimentry frame time logging to identify bottlenecks.

[ ]:
```
canvas1.dbg_log_frame_times(enabled=True)
# Then you just need to run the canvas
# canvas1.dbg_render_test()
# canvas1.run()
```

[ ]:
```
canvas1.stop(force=True)
```

Debugging Shaders With Renderdoc

Renderdoc is a powerful open-source graphics debugger. To use Renderdoc with *pySSV*, install the python bindings for the Renderdoc in-app API (https://github.com/space928/pyRenderdocApp/):

[ ]:
```
%pip install pyRenderdocApp
```

Then simply create a new `SSVCanvas` with `use_renderdoc_api` set to `True` and the Renderdoc API will be loaded automatically. To capture a frame simply press the Renderdoc logo button in the widget.

[ ]:
```
try:
    import pySSV as ssv
    import pyRenderdocApp

    canvas = ssv.canvas(use_renderdoc=True)
    canvas.shader("#pragma SSV render_test")
    canvas.run()
except ImportError:
    print("Couldn't import pyRenderdocApp!")
```

[ ]:
```
canvas.stop(force=True)
```

[ ]:
```

```

1.6 Developer install

To install a developer version of pySSV, you will first need to clone the repository:

```
git clone https://github.com/space928/Shaders-For-Scientific-Visualisation
cd Shaders-For-Scientific-Visualisation
```

You can optionally create a new Python dev environment:

```
conda create -n pySSV-dev -c conda-forge nodejs yarn python jupyterlab=4
conda activate pySSV-dev
```

Install the python package. This will also build the TS package:

```
pip install -e ".[test, examples]"
```

The `jlpm` command is JupyterLab's pinned version of [yarn](https://yarnpkg.com/) that is installed with JupyterLab. You may use `yarn` or `npm` in lieu of `jlpm` below. Using `jlpm` and `yarn` sometimes breaks the package cache if this happens, just delete the `yarn.lock` file and the `.yarn` folder and rerun `jlpm install`.

When developing your extensions, you need to manually enable your extensions with the notebook / lab frontend. For Jupyter Notebook, this is done with the command:

> jupyter nbextension install [–sys-prefix / –user / –system] –symlink –py pySSV jupyter nbextension enable [–sys-prefix / –user / –system] –py pySSV

with the appropriate flag.

Or, if you are using Jupyterlab:

```
jupyter labextension develop --overwrite .
```

Build the frontend of the plugin:

```
jlpm run build
```

If you plan on making changes to the plugin frontend (any of the typescript), then run the watch task to automatically rebuild the plugin when there are changes. Changes to the python are automatically reloaded when the Jupyter Kernel is restarted:

```
jlpm run watch
```

If you're building the documentation, you'll also need to have the libclang binaries installed. On Windows these can be installed with chocolatey:

```
choco install llvm
```

1.7 PySSV Module Reference

1.7.1 Public API

The *pySSV.ssv_canvas.SSVCanvas* is one of the core components of *pySSV*, it's what manages and provides and interface for the three major internal components: *the renderer*, *the widget*, and *the shader preprocessor*.

pySSV.**canvas**(*size: Tuple[int, int] | None = (640, 480)*, *backend: str = 'opengl'*, *gl_version: Tuple[int, int] | None = None*, *standalone: bool = False*, *target_framerate: int = 60*, *use_renderdoc: bool = False*, *supports_line_directives: bool | None = None*)

Creates a new SSVCanvas which contains the render widget and manages the render context.

Parameters

- **size** (*Tuple[int, int] | None*) – the default resolution of the renderer as a tuple: (width: int, height: int).
- **backend** (*str*) – the rendering backend to use; currently supports: "opengl".
- **gl_version** (*Tuple[int, int] | None*) – optionally, the minimum version of OpenGL to support. Accepts a tuple of (major, minor), eg: gl_version=(4, 2) for OpenGL 4.2 Core.
- **standalone** (*bool*) – whether the canvas should run standalone, or attempt to create a Jupyter Widget for rendering.
- **target_framerate** (*int*) – the default framerate to target when running.
- **use_renderdoc** (*bool*) – optionally, an instance of the Renderdoc in-app api to provide support for frame capturing and analysis in renderdoc.
- **supports_line_directives** (*bool | None*) – whether the shader compiler supports #line directives (Nvidia GPUs only). Set to None for automatic detection. If you get 'extension not supported: GL_ARB_shading_language_include' errors, set this to False.

pySSV.ssv_canvas.**OnMouseDelegate**

A callable with parameters matching the signature:

```
on_mouse(mouse_down: tuple[bool, bool, bool], mouse_pos: tuple[int, int], mouse_
↪wheel_delta: float) -> None:
    ...
```

mouse_down: a tuple of booleans representing the mouse button state (left, right, middle).
mouse_pos: a tuple of ints representing the mouse position relative to the canvas in pixels.
mouse_wheel_delta: how many pixels the mousewheel has scrolled since the last callback.

alias of Callable[[Tuple[bool, bool, bool], Tuple[int, int], float], None]

pySSV.ssv_canvas.**OnKeyDelegate**

A callable with parameters matching the signature:

```
on_key(key: str, down: bool) -> None:
    ...
```

key: the name of key in this event. Possible values can be found here: https://developer.mozilla.org/en-US/docs/Web/API/UI_Events/Keyboard_event_key_values

down: whether the key is being pressed.

alias of `Callable[[str, bool], None]`

pySSV.ssv_canvas.**OnFrameRenderedDelegate**

A callable with parameters matching the signature:

```
on_key(delta_time: float) -> None:
    ...
```

key: the time in seconds since the last frame was rendered.

alias of `Callable[[float], None]`

class pySSV.ssv_canvas.**SSVCanvas**(*size: Tuple[int, int] | None*, *backend: str = 'opengl'*, *gl_version: Tuple[int, int] | None = None*, *standalone: bool = False*, *target_framerate: int = 60*, *use_renderdoc: bool = False*, *supports_line_directives: bool | None = None*)

An SSV canvas manages the OpenGL rendering context, shaders, and the jupyter widget

Creates a new SSV Canvas object which manages the graphics context and render widget/window.

Parameters

- **size** (*Tuple[int, int] | None*) – the default resolution of the renderer as a tuple: `(width: int, height: int)`.
- **backend** (*str*) – the rendering backend to use; currently supports: `"opengl"`.
- **gl_version** (*Tuple[int, int] | None*) – optionally, the minimum version of OpenGL to support. Accepts a tuple of (major, minor), eg: gl_version=(4, 2) for OpenGL 4.2 Core.
- **standalone** (*bool*) – whether the canvas should run standalone, or attempt to create a Jupyter Widget for rendering.
- **target_framerate** (*int*) – the default framerate to target when running.
- **use_renderdoc** (*bool*) – optionally, an instance of the Renderdoc in-app api to provide support for frame capturing and analysis in renderdoc.
- **supports_line_directives** (*bool | None*) – whether the shader compiler supports `#line` directives (Nvidia GPUs only). Set to `None` for automatic detection. If you get 'extension not supported: GL_ARB_shading_language_include' errors, set this to `False`.

property main_render_buffer: *SSVRenderBuffer*

Gets the main render buffer associated with this `SSVCanvas`.

Returns
the main render buffer.

property size: Tuple[int, int]

Gets the size of the canvas in pixels

property standalone: bool

Gets whether this canvas outputs to a standalone window as opposed to a jupyter widget.

property widget: *SSVRenderWidget* | None

Gets the render widget object. Only defined when `standalone` is `False`.

property main_camera: *SSVCameraController*

Gets the main camera controller.

property mouse_down: Tuple[bool, bool, bool]

Gets the current mouse button state as a tuple of `bool`. [left, right, middle]

property mouse_pos: Tuple[int, int]

Gets the current mouse position relative to the canvas in pixels.

property preprocessor

Gets this canvas' preprocessor instance. Useful if you need to manually add compiler defines/macros.

property frame_number

Gets the number of frames that have been rendered since this canvas was run.

property canvas_time: float

Gets or sets the current canvas time (this is the value used by the shader's `uTime` uniform). Note that due to renderer latency there will be some offset to time value get/set here.

on_start(*callback: Callable[[], None]*, *remove: bool = False*)

Registers/unregisters a callback which is invoked when this canvas's `run()` method is called, before any frames are rendered.

Parameters

- **callback** (*Callable[[], None]*) – the callback to invoke.
- **remove** (*bool*) – whether the given callback should be removed.

on_mouse_event(*callback: Callable[[Tuple[bool, bool, bool], Tuple[int, int], float], None]*, *remove: bool = False*)

Registers/unregisters a callback which is invoked when.

Parameters

- **callback** (*Callable[[Tuple[bool, bool, bool], Tuple[int, int], float], None]*) – the callback to invoke.
- **remove** (*bool*) – whether the given callback should be removed.

on_keyboard_event(*callback: Callable[[str, bool], None]*, *remove: bool = False*)

Registers/unregisters a callback which is invoked when.

Parameters

- **callback** (*Callable[[str, bool], None]*) – the callback to invoke.
- **remove** (*bool*) – whether the given callback should be removed.

on_frame_rendered(*callback: Callable[[float], None]*, *remove: bool = False*)

Registers/unregisters a callback which is invoked when.

Parameters

- **callback** (*Callable[[float], None]*) – the callback to invoke.
- **remove** (*bool*) – whether the given callback should be removed.

run(*stream_mode: str | SSVStreamingMode = SSVStreamingMode.JPG*, *stream_quality: float | None = None*, *never_kill=False*) → None

Starts the render loop and displays the Jupyter Widget (or render window if in standalone mode).

Parameters

- **stream_mode** (*str | SSVStreamingMode*) – the encoding format to use to transmit rendered frames from the render process to the Jupyter widget.
- **stream_quality** (*float | None*) – the encoding quality to use for the given encoding format. Takes a float between 0-100 (some stream modes support values larger than 100, others clamp it internally), where 100 results in the highest quality. This value is scaled to give a bit rate target or quality factor for the chosen encoder. Pass in `None` to use the encoder's default quality settings.
- **never_kill** – disables the watchdog responsible for stopping the render process when the widget is no longer being displayed. *Warning*: The only way to stop a renderer started with this enabled is to restart the Jupyter kernel.

Return type
None

stop(*force=False*) → None

Stops the current canvas from rendering continuously. The renderer is not released and can be restarted.

Parameters
force – kills the render process and releases resources. SSVCanvases cannot be restarted if they have been force stopped.

Return type
None

pause() → None

Pauses the current canvas, preventing new frames from being rendered and freezing time.

Return type
None

unpause() → None

Unpauses the current canvas, and resumes playback at the time the canvas was paused at.

Return type
None

shader(*shader_source: str*, *additional_template_directory: str | None = None*, *additional_templates: List[str] | None = None*, *shader_defines: Dict[str, str] | None = None*, *compiler_extensions: List[str] | None = None*)

Registers, compiles and attaches a shader to the main render buffer.

Parameters

- **shader_source** (*str*) – the shader source code to preprocess. It should contain the necessary `#pragma SSV <template_name>` directive see *Built In Shader Templates* for more information.
- **additional_template_directory** (*str | None*) – a path to a directory containing custom shader templates. See *Writing Shader Templates* for information about using custom shader templates.

- **additional_templates** (*List[str] | None*) – a list of custom shader templates (source code, not paths).See *Writing Shader Templates* for information about using custom shader templates.
- **shader_defines** (*Dict[str, str] | None*) – extra preprocessor defines to be enabled globally.
- **compiler_extensions** (*List[str] | None*) – a list of GLSL extensions required by this shader (eg: `GL_EXT_control_flow_attributes`)

update_uniform(*uniform_name: str*, *value: Any*, *share_with_render_buffer: bool = True*, *share_with_canvas: bool = True*) → None

Sets the value of a uniform associated with the main full-screen shader.

Parameters

- **uniform_name** (*str*) – the name of the uniform to set.
- **value** (*Any*) – the value to set. Must be compatible with the destination uniform.
- **share_with_render_buffer** (*bool*) – update this uniform across all shaders in this render buffer.
- **share_with_canvas** (*bool*) – update this uniform across all shaders in this canvas.

Return type
None

render_buffer(*size: Tuple[int, int]*, *name: str | None = None*, *order: int = 0*, *dtype: str = 'f1'*, *components: int = 4*) → *SSVRenderBuffer*

Creates a new render buffer for this canvas. Useful for effects requiring multi pass rendering.

Parameters

- **size** (*Tuple[int, int]*) – the resolution of the new render buffer to create.
- **name** (*str | None*) – the name of this render buffer. This is the name given to the automatically generated uniform declaration. If set to `None` a name is automatically generated.
- **order** (*int*) – a number to hint the renderer as to the order to render the buffers in. Smaller values are rendered first; the main render buffer has an order of 999999.
- **dtype** (*str*) – the data type for each pixel component (see: https://moderngl.readthedocs.io/en/5.8.2/topics/texture_formats.html).
- **components** (*int*) – how many vector components should each pixel have (RGB=3, RGBA=4).

Returns
a new render buffer object.

Return type
SSVRenderBuffer

texture(*data: ndarray[Any, dtype[_ScalarType_co]] | Image*, *uniform_name: str | None*, *force_2d: bool = False*, *force_3d: bool = False*, *override_dtype: str | None = None*, *treat_as_normalized_integer: bool = True*, *declare_uniform: bool = True*) → *SSVTexture*

Creates or updates a texture from the NumPy array provided.

Parameters

- **data** (*ndarray[Any, dtype[_ScalarType_co]] | Image*) – a NumPy array or a PIL/Pillow Image containing the image data to copy to the texture.

- **uniform_name** (*str | None*) – optionally, the name of the shader uniform to associate this texture with. If `None` is specified, a name is automatically generated in the form 'uTexture{n}'
- **force_2d** (*bool*) – when set, forces the texture to be treated as 2-dimensional, even if it could be represented by a 1D texture. This only applies in the ambiguous case where a 2D single component texture has a height <= 4 (eg: `np.array([[0.0, 0.1, 0.2], [0.3, 0.4, 0.5], [0.6, 0.7, 0.8]])`), with this parameter set to `False`, the array would be converted to a 1D texture with a width of 3 and 3 components; setting this to `True` ensures that it becomes a 3x3 texture with 1 component.
- **force_3d** (*bool*) – when set, forces the texture to be treated as 3-dimensional, even if it could be represented by a 2D texture. See the description of the `force_2d` parameter for a full explanation.
- **override_dtype** (*str | None*) – optionally, a moderngl datatype to force on the texture. See https://moderngl.readthedocs.io/en/latest/topics/texture_formats.html for the full list of available texture formats.
- **treat_as_normalized_integer** (*bool*) – when enabled, integer types (singed/unsigned) are treated as normalized integers by OpenGL, such that when the texture is sampled values in the texture are mapped to floats in the range [0, 1] or [-1, 1]. See: https://www.khronos.org/opengl/wiki/Normalized_Integer for more details.
- **declare_uniform** (*bool*) – when set, a shader uniform is automatically declared for this uniform in shaders.

Return type
SSVTexture

get_texture(*uniform_name: str*) → *SSVTexture* | None

Gets a texture that's already been defined on this canvas.

Parameters
uniform_name (*str*) – the name of the uniform associated with this texture.

Returns
the texture object or `None` if no texture was found with that name.

Return type
SSVTexture | None

save_image(*image_type: SSVStreamingMode = SSVStreamingMode.JPG*, *quality: float = 95*, *size: Tuple[int, int] | None = None*, *render_buffer: int = 0*, *suppress_ui: bool = False*) → bytes

Saves the current frame as an image.

Parameters

- **image_type** (*SSVStreamingMode*) – the image compression algorithm to use.
- **quality** (*float*) – the encoding quality to use for the given encoding format. Takes a float between 0-100 (some stream modes support values larger than 100, others clamp it internally), where 100 results in the highest quality. This value is scaled to give a bit rate target or quality factor for the chosen encoder.
- **size** (*Tuple[int, int] | None*) – optionally, the width and height of the saved image. If set to `None` uses the current resolution of the render buffer.
- **render_buffer** (*int*) – the uid of the render buffer to save.
- **suppress_ui** (*bool*) – whether any active *SSVGUI* should be suppressed.

Returns
the bytes representing the compressed image.

Return type
bytes

dbg_query_shader_template(*shader_template_name: str*, *additional_template_directory: str | None = None*, *additional_templates=None*) → str

Gets the list of arguments a given shader template expects and returns a string containing their usage info.

Parameters

- **shader_template_name** (*str*) – the name of the template to look for.
- **additional_template_directory** (*str | None*) – a path to a directory containing custom shader templates.
- **additional_templates** – a list of custom shader templates (source code, not paths).

Returns
the shader template's auto generated help string.

Return type
str

dbg_query_shader_templates(*additional_template_directory: str | None = None*) → str

Gets a list of all the shader templates available to the preprocessor.

Parameters
additional_template_directory (*str | None*) – a path to a directory containing custom shader templates.

Returns
A string of all the shader templates which were found.

Return type
str

dbg_preprocess_shader(*shader_source: str*, *additional_template_directory: str | None = None*, *additional_templates: List[str] | None = None*, *shader_defines: Dict[str, str] | None = None*, *compiler_extensions: List[str] | None = None*, *pretty_print: bool = True*) → str | Dict[str, str]

Runs the preprocessor on a shader and returns the results. Useful for debugging shaders.

Parameters

- **shader_source** (*str*) – the shader source code to preprocess. It should contain the necessary `#pragma SSV <template_name>` directive see *Built In Shader Templates* for more information.
- **additional_template_directory** (*str | None*) – a path to a directory containing custom shader templates. See *Writing Shader Templates* for information about using custom shader templates.
- **additional_templates** (*List[str] | None*) – a list of custom shader templates (source code, not paths).See *Writing Shader Templates* for information about using custom shader templates.
- **shader_defines** (*Dict[str, str] | None*) – extra preprocessor defines to be enabled globally.
- **compiler_extensions** (*List[str] | None*) – a list of GLSL extensions required by this shader (eg: `GL_EXT_control_flow_attributes`)

- **pretty_print** (*bool*) – when enabled returns a single string containing all the preprocessed shaders

Return type
str | *Dict*[str, str]

dbg_render_test()

Sets up the render pipeline to render a demo shader.

dbg_log_context(*full=False*)

Logs the OpenGL information to the console for debugging.

Parameters
full – whether to log *all* of the OpenGL context information (including extensions).

dbg_log_frame_times(*enabled=True*)

Enables or disables frame time logging.

Parameters
enabled – whether to log frame times.

dbg_capture_frame()

Triggers a frame capture with RenderDoc if this canvas has it enabled.

Due to the asynchronous nature of the renderer, the frame may be capture 1 frame late.

Canvas Objects

These classes must be constructed by an *pySSV.ssv_canvas.SSVCanvas*.

class pySSV.ssv_render_buffer.**SSVRenderBuffer**(*canvas: SSVCanvas*, *render_process_client: SSVRenderProcessClient*, *preprocessor: SSVShaderPreprocessor*, *render_buffer_uid: int | None*, *render_buffer_name: str*, *order: int*, *size: tuple[int, int]*, *dtype: str*, *components: int*)

A lightweight class representing a native render buffer.

Used Internally

Note that SSVRenderBuffer objects should be constructed using the factory method on an SSVCanvas.

Parameters

- **canvas** (SSVCanvas) – the canvas creating this render buffer.
- **render_process_client** (SSVRenderProcessClient) – the render process connection belonging to the canvas.
- **preprocessor** (SSVShaderPreprocessor) – the preprocessor belonging to the canvas.
- **render_buffer_uid** (*Optional[int]*) – the UID to give this render buffer. Set to None to generate one automatically.
- **render_buffer_name** (*str*) – the name of this render buffer. This is the name given to the automatically generated uniform declaration.
- **order** (*int*) – the render order of the render buffer. Smaller numbers are rendered first.
- **size** (*tuple[int, int]*) – the resolution of the render buffer.
- **dtype** (*str*) – the moderngl datatype of the render buffer.

- **components** (*int*) – how many vector components should each pixel have (RGB=3, RGBA=4).

property render_buffer_uid: int

Gets the internal UID of this render buffer.

property render_buffer_name: str

Gets the name of this render buffer.

property canvas: *SSVCanvas*

Gets the canvas associated with this render buffer.

property full_screen_vertex_buffer: *SSVVertexBuffer*

Gets the default full-screen vertex buffer for this render buffer.

property order: int

Gets or sets the render order of this render buffer. This number to hint the renderer as to the order to render the buffers in. Smaller values are rendered first; the main render buffer has an order of 999999.

property size: tuple[int, int]

Gets or sets the resolution of this render buffer.

property components: int

Gets or sets how many vector components each pixel should have (RGB=3, RGBA=4).

property dtype: str

Gets or sets the data type for each pixel component (see: https://moderngl.readthedocs.io/en/5.8.2/topics/texture_formats.html).

property vertex_buffers: Tuple[*SSVVertexBuffer*, ...]

Gets the tuple of vertex buffers registered to this render buffer.

shader(*shader_source: str*, *additional_template_directory: str | None = None*, *additional_templates: list[str] | None = None*, *shader_defines: dict[str, str] | None = None*, *compiler_extensions: list[str] | None = None*)

Registers, compiles and attaches a full-screen shader to this render buffer.

Parameters

- **shader_source** (*str*) – the shader source code to preprocess. It should contain the necessary `#pragma SSV <template_name>` directive see *Built In Shader Templates* for more information.
- **additional_template_directory** (*str | None*) – a path to a directory containing custom shader templates. See *Writing Shader Templates* for information about using custom shader templates.
- **additional_templates** (*list[str] | None*) – a list of custom shader templates (source code, not paths).See *Writing Shader Templates* for information about using custom shader templates.
- **shader_defines** (*dict[str, str] | None*) – extra preprocessor defines to be enabled globally.
- **compiler_extensions** (*list[str] | None*) – a list of GLSL extensions required by this shader (eg: `GL_EXT_control_flow_attributes`)

update_uniform(*uniform_name: str*, *value: Any*, *share_with_render_buffer: bool = True*, *share_with_canvas: bool = False*) → None

Sets the value of a uniform associated with this buffer's full-screen shader.

Parameters

- **uniform_name** (*str*) – the name of the uniform to set.
- **value** (*Any*) – the value to set. Must be compatible with the destination uniform.
- **share_with_render_buffer** (*bool*) – update this uniform across all shaders in this render buffer.
- **share_with_canvas** (*bool*) – update this uniform across all shaders in this canvas.

Return type
None

vertex_buffer() → *SSVVertexBuffer*

Creates a new draw call and associated vertex buffer on this render buffer.

Returns
A new vertex buffer object.

Return type
SSVVertexBuffer

delete_vertex_buffer(*buffer:* SSVVertexBuffer)

Removes a vertex buffer from this render buffer, releasing its resources.

Parameters
buffer (SSVVertexBuffer) – the vertex buffer to remove.

class pySSV.ssv_vertex_buffer.**SSVVertexBuffer**(*draw_call_uid: int | None*, *render_buffer:* SSVRenderBuffer, *render_process_client:* SSVRenderProcessClient, *preprocessor:* SSVShaderPreprocessor)

A lightweight class representing a vertex buffer associated with a draw call.

Used internally

Note that SSVVertexBuffer objects should be constructed using the factory method on either an SSVCanvas or an SSVRenderBuffer.

Parameters

- **draw_call_uid** (*Optional[int]*) –
- **render_buffer** (SSVRenderBuffer) –
- **render_process_client** (SSVRenderProcessClient) –
- **preprocessor** (SSVShaderPreprocessor) –

property draw_call_uid: int

Gets the internal uid of this draw call.

property is_valid: bool

release()

Destroys this vertex buffer.

update_vertex_buffer(*vertex_array: ndarray[Any, dtype[_ScalarType_co]]*, *vertex_attributes: tuple[str, ...] = ('in_vert', 'in_color')*, *index_array: ndarray[Any, dtype[_ScalarType_co]] | None = None*)

Updates the data inside a vertex buffer.

Parameters

- **vertex_array** (*ndarray[Any, dtype[_ScalarType_co]]*) – a numpy array containing the new vertex data.
- **index_array** (*ndarray[Any, dtype[_ScalarType_co]] | None*) – optionally, a numpy array containing the indices of vertices ordered to make triangles.
- **vertex_attributes** (*tuple[str, ...]*) – a tuple of the names of the vertex attributes to map to in the shader, in the order that they appear in the vertex array.

shader(*shader_source: str*, *additional_template_directory: str | None = None*, *additional_templates: list[str] | None = None*, *shader_defines: dict[str, str] | None = None*, *compiler_extensions: list[str] | None = None*)

Registers, compiles and attaches a shader to the draw call associated with this vertex buffer.

Parameters

- **shader_source** (*str*) – the shader source code to preprocess. It should contain the necessary `#pragma SSV <template_name>` directive see *Built In Shader Templates* for more information.
- **additional_template_directory** (*str | None*) – a path to a directory containing custom shader templates. See *Writing Shader Templates* for information about using custom shader templates.
- **additional_templates** (*list[str] | None*) – a list of custom shader templates (source code, not paths).See *Writing Shader Templates* for information about using custom shader templates.
- **shader_defines** (*dict[str, str] | None*) – extra preprocessor defines to be enabled globally.
- **compiler_extensions** (*list[str] | None*) – a list of GLSL extensions required by this shader (eg: `GL_EXT_control_flow_attributes`)

update_uniform(*uniform_name: str*, *value: Any*, *share_with_render_buffer: bool = False*, *share_with_canvas: bool = False*) → None

Sets the value of a uniform associated with this draw call.

Parameters

- **uniform_name** (*str*) – the name of the uniform to set.
- **value** (*Any*) – the value to set. Must be compatible with the destination uniform.
- **share_with_render_buffer** (*bool*) – update this uniform across all shaders in this render buffer.
- **share_with_canvas** (*bool*) – update this uniform across all shaders in this canvas.

Return type
None

class pySSV.ssv_camera.**MoveDir**(*value*)

Represents a cardinal direction to move in.

NONE = 0

FORWARD = 1

BACKWARD = 2

LEFT = 3

RIGHT = 4

UP = 5

DOWN = 6

class pySSV.ssv_camera.**SSVCamera**

A simple class representing a camera.

position: ndarray[Any, dtype[float32]]

The camera's position in 3D space.

direction: ndarray[Any, dtype[float32]]

A normalised vector pointing in the direction the camera is facing.

fov: float

The field of view of the camera in degrees.

clip_dist: Tuple[float, float]

The distances of the near and far clipping planes respectively.

aspect_ratio: float

The aspect ratio of the render buffer.

property rotation_matrix: ndarray[Any, dtype[float32]]

Gets the current view matrix of the camera, without the translation component.

property view_matrix: ndarray[Any, dtype[float32]]

Gets the current view matrix of the camera.

property projection_matrix

Gets the current projection matrix of the camera.

class pySSV.ssv_camera.**SSVCameraController**

A simple class representing a camera controller.

inhibit: bool = False

Whether the camera controls should be inhibited

move_speed: float

The movement speed of the camera.

zoom_speed: float

The zooming speed of the camera.

pan_speed: float

The panning speed of the camera in radians per pixel of mouse movement.

abstract mouse_change(*mouse_pos: Tuple[int, int]*, *mouse_down: Tuple[bool, bool, bool]*)

Parameters

- **mouse_pos** (*Tuple[int, int]*) –
- **mouse_down** (*Tuple[bool, bool, bool]*) –

abstract move(*direction:* MoveDir, *distance: float = 1.0*)

Parameters

- **direction** (MoveDir) –

- **distance** (*float*) –

class pySSV.ssv_camera.**SSVLookCameraController**

A camera controller which supports mouse look controls.

mouse_change(*mouse_pos: Tuple[int, int]*, *mouse_down: Tuple[bool, bool, bool]*)

Updates the camera with a mouse event.

Parameters

- **mouse_pos** (*Tuple[int, int]*) – the new mouse position.
- **mouse_down** (*Tuple[bool, bool, bool]*) – whether the mouse button is pressed.

move(*direction:* MoveDir, *distance: float = 1.0*)

Updates the camera position with a movement event.

Parameters

- **direction** (*MoveDir*) – the direction to move in.
- **distance** (*float*) – the distance to move.

class pySSV.ssv_camera.**SSVOrbitCameraController**

A camera controller which supports orbiting around a given target.

property target_pos

Gets or sets the point around which to orbit.

property orbit_dist

Gets or sets the distance from the target position to orbit at.

mouse_change(*mouse_pos: Tuple[int, int]*, *mouse_down: Tuple[bool, bool, bool]*)

Updates the camera with a mouse event.

Parameters

- **mouse_pos** (*Tuple[int, int]*) – the new mouse position.
- **mouse_down** (*Tuple[bool, bool, bool]*) – whether the mouse button is pressed.

zoom(*distance: float*)

Updates the orbit distance with a zoom event.

Parameters
distance (*float*) – how far to zoom in.

move(*direction:* MoveDir, *distance: float = 1.0*)

Updates the camera position with a movement event.

Parameters

- **direction** (*MoveDir*) – the direction to move in.
- **distance** (*float*) – the distance to move.

pySSV.ssv_texture.**determine_texture_shape**(*data: ndarray[Any, dtype[_ScalarType_co]]*, *override_dtype: str | None*, *treat_as_normalized_integer: bool = True*) → tuple[int, int, int, int, str | None]

Attempts to determine suitable texture parameters given an ndarray. This method returns (0,0,0,0,None) if a suitable format cannot be found.

Parameters

- **data** (*ndarray[Any, dtype[_ScalarType_co]]*) – the ndarray to parse.
- **override_dtype** (*str | None*) – optionally, a moderngl dtype string to use instead of the numpy dtype.
- **treat_as_normalized_integer** (*bool*) – when enabled, integer types (singed/unsigned) are treated as normalized integers by OpenGL, such that when the texture is sampled values in the texture are mapped to floats in the range [0, 1] or [-1, 1]. See: https://www.khronos.org/opengl/wiki/Normalized_Integer for more details.

Returns
(components, depth, height, width, dtype)

Return type
tuple[int, int, int, int, str | None]

class pySSV.ssv_texture.**SSVTexture**(*texture_uid: int | None*, *render_process_client: SSVRenderProcessClient*, *preprocessor: SSVShaderPreprocessor*, *data: npt.NDArray | Image.Image*, *uniform_name: str*, *force_2d: bool = False*, *force_3d: bool = False*, *override_dtype: str | None = None*, *treat_as_normalized_integer: bool = True*, *declare_uniform: bool = True*)

A lightweight class representing a Texture object.

Used Internally

Note that `SSVTexture` objects should be constructed using the factory method on either an `SSVCanvas`.

Parameters

- **texture_uid** (*Optional[int]*) – the UID to give this texture buffer. Set to `None` to generate one automatically.
- **render_process_client** (`SSVRenderProcessClient`) – the render process connection belonging to the canvas.
- **preprocessor** (`SSVShaderPreprocessor`) – the preprocessor belonging to the canvas.
- **data** (*Union[npt.NDArray, Image.Image]*) – a NumPy array or a PIL/Pillow Image containing the image data to copy to the texture.
- **uniform_name** (*str*) – the name of the shader uniform to associate this texture with.
- **force_2d** (*bool*) – when set, forces the texture to be treated as 2-dimensional, even if it could be represented by a 1D texture. This only applies in the ambiguous case where a 2D single component texture has a height <= 4 (eg: `np.array([[0.0, 0.1, 0.2], [0.3, 0.4, 0.5], [0.6, 0.7, 0.8]])`), with this parameter set to `False`, the array would be converted to a 1D texture with a width of 3 and 3 components; setting this to `True` ensures that it becomes a 3x3 texture with 1 component.
- **force_3d** (*bool*) – when set, forces the texture to be treated as 3-dimensional, even if it could be represented by a 2D texture. See the description of the `force_2d` parameter for a full explanation.
- **override_dtype** (*Optional[str]*) – optionally, a moderngl datatype to force on the texture.
- **treat_as_normalized_integer** (*bool*) – when enabled, integer types (singed/unsigned) are treated as normalized integers by OpenGL, such that when the texture is sampled values in the texture are mapped to floats in the range [0, 1] or [-1, 1]. See: https://www.khronos.org/opengl/wiki/Normalized_Integer for more details.

- **declare_uniform** (*bool*) – when set, a shader uniform is automatically declared for this uniform in shaders.

property texture_uid: int

Gets the internal UID of this texture object.

property uniform_name: str

Gets the shader uniform name associated with this texture.

property components: int

Gets the number of components for a single pixel (RGB=3, RGBA=4). Always at least 1, never more than 4.

property depth: int

Gets the depth of the texture. Always returns 1 for 1D and 2D textures.

property height: int

Gets the height of the texture. Always returns 1 for 1D textures.

property width: int

Gets the width of the texture.

property dtype: str

Gets the data type of a single component in the texture.

See https://moderngl.readthedocs.io/en/latest/topics/texture_formats.html for a full list of available data types.

property repeat_x: None

Sets whether the texture should repeat or be clamped in the x-axis.

property repeat_y: None

Sets whether the texture should repeat or be clamped in the y-axis.

property linear_filtering: None

Sets whether the texture should use nearest neighbour (`False`) or linear (`True`) interpolation.

property linear_mipmap_filtering: None

Sets whether different mipmap levels should blend linearly (`True`) or not (`False`).

property anisotropy: None

Sets the number of anisotropy samples to use. (minimum of 1 = disabled, maximum of 16)

property is_valid: bool

Gets whether this texture object represents a valid texture that hasn't been destroyed yet.

update_texture(*data: ndarray[Any, dtype[_ScalarType_co]]*, *rect: tuple[int, int, int, int] | tuple[int, int, int, int, int, int] | None = None*)

Updates the contents of this texture from the NumPy array provided.

Parameters

- **data** (*ndarray[Any, dtype[_ScalarType_co]]*) – a NumPy array containing the image data to copy to the texture.
- **rect** (*tuple[int, int, int, int] | tuple[int, int, int, int, int, int] | None*) – optionally, a rectangle (left, top, right, bottom) specifying the area of the target texture to update.

build_mipmaps()

Generates mipmaps for the texture.

release()

Destroys this texture object and releases the associated GPU resources.

GUI Library

pySSV provides a simple to use immediate-mode GUI library to add interactivity to your shaders. A GUI can be created for a given canvas by calling the *pySSV.ssv_gui.create_gui()* method which returns a new *pySSV.ssv_gui.SSVGUI*.

class pySSV.ssv_gui.**SSVGUIShaderMode**(*value*)

Represents the shader features needed to render a GUI element when using the built-in UI shader.

SOLID = 0

TRANSPARENT = 1

TEXT = 2

TEXTURE = 4

SHADOWED = 8

ROUNDING = 16

OUTLINE = 32

static get_vertex_attributes(*shader_mode: int*) → Tuple[str, ...]

Gets the tuple of vertex attribute names required to support this shader mode.

Parameters
shader_mode (*int*) –

Returns

Return type
Tuple[str, . . .]

class pySSV.ssv_gui.**TextAlign**(*value*)

An enumeration.

LEFT = 0

CENTRE = 1

CENTER = 1

RIGHT = 2

class pySSV.ssv_gui.**Rect**(*x: int = 0*, *y: int = 0*, *width: int = 20*, *height: int = 20*)

Represents a 2D rectangle in pixel space.

Parameters

- **x** (*int*) –
- **y** (*int*) –
- **width** (*int*) –

- **height** (*int*) –

x: int

y: int

width: int

height: int

pySSV.ssv_gui.**SSVGUIDrawDelegate**

draw(gui: SSVGUI, rect: Rect) -> None

Type
A delegate for a GUIElement draw function. It should follow the signature

alias of Callable[[*SSVGUI*, *Rect*], None]

pySSV.ssv_gui.**SSVGUIPreLayoutDelegate**

A delegate for a GUIElement pre layout function. It should follow the signature: draw(gui: SSVGUI) -> tuple[width: int, height: int]

alias of Callable[[*SSVGUI*], Tuple[int, int]]

class pySSV.ssv_gui.**SSVGUIElement**(*draw_func: Callable[[SSVGUI, Rect], None]*, *control_width: int*, *control_height: int*, *expand: bool*, *layout: bool*, *overlay_last: bool*, *pre_layout_func: Callable[[SSVGUI], Tuple[int, int]] | None*)

A class representing a single GUI element for use by the layout engine.

Parameters

- **draw_func** (*Callable[[SSVGUI, Rect], None]*) –
- **control_width** (*int*) –
- **control_height** (*int*) –
- **expand** (*bool*) –
- **layout** (*bool*) –
- **overlay_last** (*bool*) –
- **pre_layout_func** (*Callable[[SSVGUI], Tuple[int, int]] | None*) –

draw_func: Callable[[*SSVGUI*, *Rect*], None]

control_width: int

control_height: int

expand: bool

layout: bool

overlay_last: bool

pre_layout_func: Callable[[*SSVGUI*], Tuple[int, int]] | None

class pySSV.ssv_gui.**SSVGUILayoutContainer**(*gui:* SSVGUI, *vertical:* *bool* = *True*, *enabled:* *bool* | Reference[*bool*] = *True*, *squeeze:* *bool* = *False*, *pad:* *bool* = *False*)

A GUILayoutContainer stores a list gui elements (represented by their draw functions). It automatically lays out all of its elements either vertically or horizontally when its `draw()` method is called. An `SSVGUILayoutContainer` can itself be put inside another layout container.

Parameters

- **gui** (SSVGUI) –
- **vertical** (*bool*) –
- **enabled** (*bool | Reference[bool]*) –
- **squeeze** (*bool*) –
- **pad** (*bool*) –

draw(*gui:* SSVGUI, *max_bounds:* Rect)

Lays out and draws all the elements within this container in the order they were added.

Parameters

- **gui** (SSVGUI) – the parent `SSVGUI` instance.
- **max_bounds** (Rect) – the rect representing the space to fit elements within.

property min_width: int

Measures the minimum width of a given layout group by recursively measuring all of its children.

property min_height: int

Measures the minimum height of a given layout group by recursively measuring all of its children.

add_element(*draw_callback:* *Callable[[SSVGUI, Rect], None]*, *control_width:* *int*, *control_height:* *int*, *expand:* *bool* = *False*, *layout:* *bool* = *True*, *overlay_last:* *bool* = *False*, *pre_layout_callback:* *Callable[[SSVGUI], Tuple[int, int]]* | *None* = *None*)

Adds a GUI element to this Layout Container.

Parameters

- **draw_callback** (*Callable[[SSVGUI, Rect], None]*) – the draw function of the GUI element
- **control_width** (*int*) – the requested width of the control. The layout engine can give a larger width than this if the `expand` option is enabled; if the layout group has `squeeze` enabled, the actual width may be smaller than requested.
- **control_height** (*int*) – the requested height of the control. The layout engine can give a larger height than this if the `expand` option is enabled; if the layout group has `squeeze` enabled, the actual height may be smaller than requested.
- **expand** (*bool*) – whether the element should attempt to fill all remaining space in the container. If multiple elements have `expand` set, then the remaining space is shared. The element's minimum size is still determined by the defined layout size.
- **layout** (*bool*) – whether this element should participate in automatic layout. If disabled, the element doesn't count towards layout calculations and is given the full `Rect` of the Layout Container. The element will still be drawn in the order specified.
- **overlay_last** (*bool*) – whether this element should be overlaid on top of the last element drawn.

- **pre_layout_callback** (*Callable[[SSVGUI], Tuple[int, int]] | None*) – the callback is invoked just before the element is laid out, it's useful for Layout Group elements which might not know their minimum size until just before layout.

class pySSV.ssv_gui.**SSVGUI**(*canvas:* SSVCanvas, *render_buffer:* SSVRenderBuffer)

An immediate mode GUI library for pySSV.

Creates a new GUI and binds its event listeners to the given canvas.

Parameters

- **canvas** (SSVCanvas) – the canvas to get events from.
- **render_buffer** (SSVRenderBuffer) – the buffer to render into.

property **layout_control_height**: int

Gets or sets the default GUI element height.

property **layout_control_width**: int

Gets or sets the default GUI element width.

property **padding**: Tuple[int, int, int, int]

Gets or sets the amount of padding between GUI elements in pixels.

property **rounding_radius**: float

Gets or sets the default corner radius for GUI elements, in pixels.

on_gui(*callback: Callable[[*SSVGUI*], None]*, *remove: bool = False*)

Registers/unregisters a callback to this GUI's on_gui event which is called any time the GUI is invalidated and needs to be redrawn.

All GUI drawing operations should occur within the callback registered here. Calling GUI drawing functions outside of this callback results in undefined behaviour.

Parameters

- **callback** (*Callable[[SSVGUI], None]*) – the callback function to register to the on_gui event.
- **remove** (*bool*) – whether the function passed in should be removed from the callback list.

on_post_gui(*callback: Callable[[*SSVGUI*], None]*, *remove: bool = False*)

Registers/unregisters a callback to this GUI's on_post_gui event which is called just after the GUI drawn.

GUI drawing operations are not permitted within this callback; but any `Reference` values returned by GUI elements *will* have been updated by the time this callback is invoked.

Parameters

- **callback** (*Callable[[SSVGUI], None]*) – the callback function to register to the on_post_gui event.
- **remove** (*bool*) – whether the function passed in should be removed from the callback list.

begin_horizontal(*width: int | None = None*, *height: int | None = None*, *pad: bool = False*, *squeeze: bool = True*)

Starts a new horizontal layout group. All GUI elements created after this call will flow horizontally, left to right until `end_horizontal()` is called.

Parameters

- **width** (*int | None*) – optionally override the width of this layout group. Defaults to the current `layout_control_width`.

- **height** (*int | None*) – optionally override the height of this layout group. Defaults to the current `layout_control_height`.
- **pad** (*bool*) – whether padding should be created between this layout group and the last GUI element.
- **squeeze** (*bool*) – whether this layout group should attempt to squeeze the elements contained within if they would have otherwise overflowed.

`end_horizontal()`

Ends a horizontal layout group.

`begin_vertical`(*width: int | None = None*, *height: int | None = None*, *pad: bool = False*, *squeeze: bool = False*)

Starts a new vertical layout group. All GUI elements created after this call will flow vertically, top to bottom until `end_vertical()` is called.

Parameters

- **width** (*int | None*) – optionally override the width of this layout group. Defaults to the current `layout_control_width`.
- **height** (*int | None*) – optionally override the height of this layout group. Defaults to the current `layout_control_height`.
- **pad** (*bool*) – whether padding should be created between this layout group and the last GUI element.
- **squeeze** (*bool*) – whether this layout group should attempt to squeeze the elements contained within if they would have otherwise overflowed.

`end_vertical()` → *SSVGUILayoutContainer*

Ends a vertical layout group.

Return type
SSVGUILayoutContainer

`begin_toggle`(*enabled: bool | Reference[bool]*, *width: int | None = None*, *height: int | None = None*, *pad: bool = False*, *squeeze: bool = False*)

Starts a new toggle layout group. GUI elements contained within this group can be shown or hidden using the `enabled` field. All GUI elements created after this call will flow vertically, top to bottom until `end_toggle()` is called.

Parameters

- **enabled** (*bool | Reference[bool]*) – a boolean or a reference to a boolean for whether the contents of this group should be shown.
- **width** (*int | None*) – optionally override the width of this layout group. Defaults to the current `layout_control_width`.
- **height** (*int | None*) – optionally override the height of this layout group. Defaults to the current `layout_control_height`.
- **pad** (*bool*) – whether padding should be created between this layout group and the last GUI element.
- **squeeze** (*bool*) – whether this layout group should attempt to squeeze the elements contained within if they would have otherwise overflowed.

`end_toggle()`

Ends a toggle layout group.

space(*width: int | None = None*, *height: int | None = None*)

Creates a blank space element.

Parameters

- **width** (*int | None*) – optionally, the width of the element.
- **height** (*int | None*) – optionally, the height of the element.

rect(*colour:* Colour, *rect:* Rect | *None = None*, *overlay_last: bool = False*)

Creates a rectangle GUI element.

Parameters

- **colour** (Colour) – the colour of the rectangle.
- **rect** (Rect | *None*) – optionally, the absolute coordinates of the rectangle to draw. These will be clipped to fit within the current layout group.
- **overlay_last** (*bool*) – whether the layout engine should overlay this element onto the last drawn element.

rounded_rect(*colour:* Colour, *outline: bool = False*, *radius: float | None = None*, *rect:* Rect | *None = None*, *overlay_last: bool = False*)

Creates a rounded rectangle GUI element.

Parameters

- **colour** (Colour) – the colour of the rectangle.
- **outline** (*bool*) – whether the rectangle should be outlined.
- **radius** (*float | None*) – the rounding radius in pixels. This can be set to an arbitrarily high number to create circles/pills. Set to `None` to use the GUI's `rounding_radius`.
- **rect** (Rect | *None*) – optionally, the absolute coordinates of the rectangle to draw. These will be clipped to fit within the current layout group.
- **overlay_last** (*bool*) – whether the layout engine should overlay this element onto the last drawn element.

label(*text: str*, *colour: ~pySSV.ssv_colour.Colour = <pySSV.ssv_colour.Colour object>*, *font_size: float | None = None*, *x_offset: int = 0*, *weight: float = 0.5*, *italic: bool = False*, *shadow: bool = False*, *align: ~pySSV.ssv_gui.TextAlign = TextAlign.LEFT*, *enforce_hinting: bool = True*, *rect: ~pySSV.ssv_gui.Rect | None = None*, *overlay_last: bool = False*)

Creates a label GUI element.

Parameters

- **text** (*str*) – the text to display.
- **colour** (Colour) – the colour of the rectangle.
- **font_size** (*float | None*) – the font size in pt.
- **x_offset** (*int*) – how far to indent the text in pixels.
- **weight** (*float*) – the font weight [0-1], where 0.5 is the native font weight. The font renderer uses SDF fonts which allows variable font weight rendering for free within certain limits (since this is only an effect, at the extremes type quality is degraded).
- **italic** (*bool*) – whether the text should be rendered in faux italic. This effect simply applies a shear transformation to the rendered characters, so it will work on any font, but won't look as good as a proper italic font.

- **shadow** (*bool*) – whether the text should be rendered with a shadow. This incurs a very small extra rendering cost, and tends to have visual artifacts when the font weight is high.
- **align** (TextAlign) – the horizontal alignment of the text.
- **enforce_hinting** (*bool*) – this option applies rounding to the font size and position to force it to line up with the pixel grid to improve sharpness. This is only effective if the font texture was rendered with hinting enabled in the first place. This can result in aliasing when animating font size/text position.
- **rect** (Rect | *None*) – optionally, the absolute coordinates of the rectangle to draw. These will be clipped to fit within the current layout group.
- **overlay_last** (*bool*) – whether the layout engine should overlay this element onto the last drawn element.

label_3d(*text: str, pos: ~typing.Tuple[float, float, float], colour: ~pySSV.ssv_colour.Colour = <pySSV.ssv_colour.Colour object>, font_size: float | None = None, weight: float = 0.5, italic: bool = False, shadow: bool = False, align: ~pySSV.ssv_gui.TextAlign = TextAlign.CENTRE, enforce_hinting: bool = True*)

Creates a label GUI element which is transformed in 3d space using the canvas's camera.

Parameters

- **text** (*str*) – the text to display.
- **pos** (*Tuple[float, float, float]*) – the 3d position of the label.
- **colour** (Colour) – the colour of the rectangle.
- **font_size** (*float | None*) – the font size in pt.
- **weight** (*float*) – the font weight [0-1], where 0.5 is the native font weight. The font renderer uses SDF fonts which allows variable font weight rendering for free within certain limits (since this is only an effect, at the extremes type quality is degraded).
- **italic** (*bool*) – whether the text should be rendered in faux italic. This effect simply applies a shear transformation to the rendered characters, so it will work on any font, but won't look as good as a proper italic font.
- **shadow** (*bool*) – whether the text should be rendered with a shadow. This incurs a very small extra rendering cost, and tends to have visual artifacts when the font weight is high.
- **align** (TextAlign) – the horizontal alignment of the text.
- **enforce_hinting** (*bool*) – this option applies rounding to the font size and position to force it to line up with the pixel grid to improve sharpness. This is only effective if the font texture was rendered with hinting enabled in the first place. This can result in aliasing when animating font size/text position.

button(*text: str*, *colour:* Colour | *None* = *None*, *radius: float* | *None* = *None*, *rect:* Rect | *None* = *None*) → *Reference*[bool]

Creates a button GUI element.

Since the actual drawing of GUI elements is deferred till after layout has been updated (which occurs just after the on_gui event finishes), the result of whether the button has been clicked or not is not known when this method returns. Wait until the `on_post_gui` event (or the start of the next `on_gui`) event to get the value of the button.

Parameters

- **text** (*str*) – the button text.

- **colour** (Colour | *None*) – the colour of the button rectangle.
- **radius** (*float* | *None*) – optionally, the corner radius of the button rectangle.
- **rect** (Rect | *None*) – optionally, the absolute coordinates of the rectangle to draw. These will be clipped to fit within the current layout group.

Returns
a reference to a boolean which will be set to `True` if the button was clicked.

Return type
Reference[bool]

slider(*text: str*, *value: float*, *min_value: float = 0.0*, *max_value: float = 1.0*, *step_size: float = 0*, *power: float = 1.0*, *colour:* Colour | *None = None*, *track_thickness: float = 4*, *rect:* Rect | *None = None*) → *Reference*[float]

Creates a slider GUI element.

Since the actual drawing of GUI elements is deferred till after layout has been updated (which occurs just after the on_gui event finishes), the updated value of this slider is not known when this method returns. Wait until the `on_post_gui` event (or the start of the next `on_gui`) event to get the new value of this slider. Until then the value returned by the slider will be the value passed in to it.

Parameters

- **text** (*str*) – the label of the slider.
- **value** (*float*) – the current value of the slider.
- **min_value** (*float*) – the minimum value of the slider.
- **max_value** (*float*) – the maximum value of the slider.
- **step_size** (*float*) – the step size to round the slider value to.
- **power** (*float*) – an exponent to raise the value of the slider to, useful for creating non-linear sliders.
- **colour** (Colour | *None*) – the colour of the rectangle.
- **track_thickness** (*float*) – the thickness of the slider track in pixels.
- **rect** (Rect | *None*) – optionally, the absolute coordinates of the rectangle to draw. These will be clipped to fit within the current layout group.

Returns
a reference to a float which will be set to the new value of the slider.

Return type
Reference[float]

checkbox(*text: str*, *value: bool* | Reference*[bool]*, *colour:* Colour | *None = None*, *radius: float* | *None = None*, *rect:* Rect | *None = None*) → *Reference*[bool]

Creates a checkbox GUI element.

Since the actual drawing of GUI elements is deferred till after layout has been updated (which occurs just after the on_gui event finishes), the updated value of this checkbox is not known when this method returns. Wait until the `on_post_gui` event (or the start of the next `on_gui`) event to get the new value of this checkbox. Until then the value returned by the checkbox will be the value passed in to it.

Parameters

- **text** (*str*) – the label of the checkbox.
- **value** (*bool* | Reference*[bool]*) – whether the checkbox is currently checked.

- **colour** (Colour | *None*) – the colour of the checkbox.
- **radius** (*float* | *None*) – optionally, the corner radius of the checkbox.
- **rect** (Rect | *None*) – optionally, the absolute coordinates of the rectangle to draw. These will be clipped to fit within the current layout group.

Returns
a reference to a float which will be set to the new value of the checkbox.

Return type
Reference[bool]

pySSV.ssv_gui.**create_gui**(*canvas:* SSVCanvas) → *SSVGUI*

Creates a new full screen GUI and render buffer and binds it to the canvas (the render buffer's order defaults to 100).

Parameters
canvas (SSVCanvas) – the canvas to bind to.

Returns
a new SSVGUI object

Return type
SSVGUI

class pySSV.ssv_colour.**Colour**(*r: float = 0*, *g: float = 0*, *b: float = 0*, *a: float = 1*)

Represents an RGBA colour.

Creates a new colour object

Parameters
- **r** (*float*) – red (0-1)
- **g** (*float*) – green (0-1)
- **b** (*float*) – blue (0-1)
- **a** (*float*) – alpha (0-1)

r: float

g: float

b: float

a: float

property astuple: Tuple[float, float, float, float]

Gets the (r, g, b, a) tuple of this colour.

static from_hex(*hex_colour: str*) → *Colour*

Parameters
hex_colour (*str*) –

Return type
Colour

static from_int(*r: int*, *g: int*, *b: int*, *a: int = 255*) → *Colour*

Parameters
- **r** (*int*) –

- **g** (*int*) –
- **b** (*int*) –
- **a** (*int*) –

Return type
Colour

class pySSV.ssv_fonts.**SSVCharacterDefinition**(*id: int*, *char: str*, *x: int*, *y: int*, *width: int*, *height: int*, *x_offset: int*, *y_offset: int*, *x_advance: int*)

Parameters

- **id** (*int*) –
- **char** (*str*) –
- **x** (*int*) –
- **y** (*int*) –
- **width** (*int*) –
- **height** (*int*) –
- **x_offset** (*int*) –
- **y_offset** (*int*) –
- **x_advance** (*int*) –

id: int
The id of the character. (Usually the ascii character code)

char: str
The character being represented.

x: int
The x coordinate of the character in the bitmap from the left in pixels.

y: int
The y coordinate of the character in the bitmap from the top in pixels.

width: int
The width of the character in the bitmap in pixels.

height: int
The height of the character in the bitmap in pixels.

x_offset: int
How much to offset the character by in the x axis when rendering in pixels.

y_offset: int
How much to offset the character by in the y axis when rendering in pixels.

x_advance: int
How far to advance before drawing the next character.

class pySSV.ssv_fonts.**SSVFont**(*font_path: str*)

Constructs a new SSVFont instance from an existing `.fnt` file.

A `.fnt` file is a Bitmap Font file which is an xml file following the schema defined here: https://www.angelcode.com/products/bmfont/doc/file_format.html

Font files can be generated using: https://github.com/soimy/msdf-bmfont-xml

Parameters
font_path (*str*) – the path to the font file to load.

1.7.2 Internal Modules

As a user you generally won't need to interact with these modules, the documentation provided here is for developers wanting to extend pySSV and power users wanting to better understand how *pySSV* works internally.

Utilities

class pySSV.ssv_logging.**SSVFormatter**(*fmt*)

Initialize the formatter with specified format strings.

Initialize the formatter either with the specified format string, or a default as described above. Allow for specialized date formatting with the optional datefmt argument. If datefmt is omitted, you get an ISO8601-like (or RFC 3339-like) format.

Use a style parameter of '%', '{' or '$' to specify that you want to use one of %-formatting, `str.format()` ({}) formatting or `string.Template` formatting in your format string.

Changed in version 3.2: Added the `style` parameter.

format(*record: LogRecord*) → str

Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

Parameters
record (*LogRecord*) –

Return type
str

class pySSV.ssv_logging.**SSVLogStream**

A StringIO pipe for sending log messages with a severity level.

abstract write(*text: str*, *severity: int = 20*) → int

Called to write a log message to the stream.

Parameters

- **text** (*str*) – the message to log.
- **severity** (*int*) – the severity to log the message with.

Returns
the number of characters written to the stream.

Return type
int

class pySSV.ssv_logging.**SSVStreamHandler**(*stream=None*)

Initialize the handler.

If stream is not specified, sys.stderr is used.

Parameters
stream (SSVLogStream | TextIO) –

stream: SSVLogStream | TextIO

emit(*record: LogRecord*) → None

Emit a record.

If a formatter is specified, it is used to format the record. The record is then written to the stream with a trailing newline. If exception information is present, it is formatted using traceback.print_exception and appended to the stream. If the stream has an 'encoding' attribute, it is used to determine how to do the output to the stream.

Parameters
record (LogRecord) –

Return type
None

pySSV.ssv_logging.**make_formatter**(*prefix='pySSV'*)

pySSV.ssv_logging.**set_output_stream**(*stream: TextIOBase*, *level=20*, *prefix='pySSV'*)

Parameters
stream (TextIOBase) –

pySSV.ssv_logging.**set_severity**(*severity: int*)

Sets the minimum message severity to be logged.

Parameters
severity (int) – the logging severity as an integer. Preset severity levels are defined in the `logging` module. Possible values include: `logging.DEBUG`, `logging.INFO`, `logging.WARN`, `logging.ERROR`, etc…

pySSV.ssv_logging.**get_severity**() → int

Gets the minimum severity level of the current logger.

Returns
the minimum severity level of the logger.

Return type
int

pySSV.ssv_logging.**log**(*msg*, **args*, *severity=10*, *raw=False*)

Logs a message to the console.

Parameters

- **msg** – message to log to the console
- **args** – objects to log to the console

- **severity** – the severity to log the message with, severity levels are defined in the `logging` module.
- **raw** – logs the message without any formatting

class pySSV.environment.**Env**(*value*)

An enumeration.

JUPYTER_NOTEBOOK = 'notebook'

JUPYTERLAB = 'lab'

JUPYTERLITE = 'lite'

SAGEMAKER = 'sagemaker'

HYPHA = 'hypha'

COLAB = 'colab'

pySSV.environment.**find_env**()

class pySSV.ssv_callback_dispatcher.**SSVCallbackDispatcher**

A simple event callback dispatcher class similar to the `ipywidgets.widgets.widget.CallbackDispatcher`.

register_callback(*callback: T*, *remove: bool = False*)

Registers/unregisters a callback to this dispatcher.

Parameters

- **callback** (*T*) – the callback to add/remove.
- **remove** (*bool*) – whether the callback should be removed.

class pySSV.ssv_future.**Future**

Represents a lightweight, low-level Event-backed future.

For more complex async requirements, the asyncio library is probably a better idea.

property result: T

property is_available: bool

set_result(*val: T*)

Sets the result of the Future object and notifies objects waiting for the result.

Parameters
val (*T*) – the result to set.

wait_result(*timeout: float | None = None*) → T | None

Waits synchronously until the result is available and then returns it.

Parameters
timeout (*float | None*) – the maximum amount of time in seconds to wait for the result. Set to `None` to wait indefinitely.

Returns
the awaited result or `None` if the operation timed out.

Return type
T | None

class pySSV.ssv_future.**Reference**(*value: T*)

Represents an object/value which can be passed by reference.

Parameters
value (*T*) –

property **result: T**

Renderer

class pySSV.ssv_render.**ShaderStage**(*value*)

An enum representing an OpenGL shader stage.

VERTEX = 'vertex'

TESSELLATION = 'tessellation'

GEOMETRY = 'geometry'

PIXEL = 'pixel'

COMPUTE = 'compute'

class pySSV.ssv_render.**SSVStreamingMode**(*value*)

Represents an image/video streaming mode for pySSV. Note that some of these streaming formats may not be supported on all platforms.

JPG = 'jpg'

PNG = 'png'

VP8 = 'vp8'

VP9 = 'vp9'

H264 = 'h264'

HEVC = 'hevc'

Not supported

MPEG4 = 'mpeg4'

Not supported

MJPEG = 'mjpeg'

class pySSV.ssv_render.**SSVRender**(*gl_version: int | None = None*, *use_renderdoc_api: bool = False*)

An abstract rendering backend for SSV

Initialises a new renderer (and backing graphics context) with the given options.

Parameters

- **gl_version** (*int | None*) – optionally, the minimum version of OpenGL to support.
- **use_renderdoc_api** (*bool*) – whether the renderer should attempt to load the RenderDoc API.

abstract render() → bool

Renders a complete frame.

Returns
whether the frame rendered successfully.

Return type
bool

abstract read_frame(*components: int = 4*, *frame_buffer_uid: int = 0*) → bytes

Gets the current contents of the frame buffer as a byte array.

Parameters

- **components** (*int*) – how many components to read from the frame (out of RGBA).
- **frame_buffer_uid** (*int*) – the frame buffer to read from.

Returns
the contents of the frame buffer as a bytearray in the RGBA format.

Return type
bytes

abstract read_frame_into(*buffer: bytearray*, *components: int = 4*, *frame_buffer_uid: int = 0*) → None

Gets the current contents of the frame buffer as a byte array.

Parameters

- **buffer** (*bytearray*) – the buffer to copy the frame into.
- **components** (*int*) – how many components to read from the frame (out of RGBA).
- **frame_buffer_uid** (*int*) – the frame buffer to read from.

Return type
None

abstract log_context_info(*full=False*) → None

Logs the OpenGL information to the console for debugging.

Parameters
full – whether to log *all* of the OpenGL context information (including extensions).

Return type
None

abstract get_context_info() → Dict[str, str]

Returns the OpenGL context information.

Return type
Dict[str, str]

abstract get_supported_extensions() → Set[str]

Gets the set of supported OpenGL shader compiler extensions.

Return type
Set[str]

abstract update_frame_buffer(*frame_buffer_uid: int*, *order: int | None*, *size: Tuple[int, int] | None*, *uniform_name: str | None*, *components: int | None = 4*, *dtype: str | None = 'f1'*) → None

Updates the resolution/format of the given frame buffer. Note that framebuffer 0 is always used for output. If the given framebuffer id does not exist, it is created.

Setting a parameter to `None` preserves the current value for that frame buffer.

Parameters

- **frame_buffer_uid** (*int*) – the uid of the framebuffer to update/create. Buffer 0 is the output framebuffer.
- **order** (*int | None*) – the sorting order to render the frame buffers in, smaller values are rendered first.
- **size** (*Tuple[int, int] | None*) – the new resolution of the framebuffer.
- **uniform_name** (*str | None*) – the name of the uniform to bind this frame buffer to.
- **components** (*int | None*) – how many vector components should each pixel have (RGB=3, RGBA=4).
- **dtype** (*str | None*) – the data type for each pixel component (see: https://moderngl.readthedocs.io/en/5.8.2/topics/texture_formats.html).

Return type
None

abstract delete_frame_buffer(*frame_buffer_uid: int*) → None

Destroys the given framebuffer. *Note* that framebuffer 0 can't be destroyed as it is the output framebuffer.

Parameters
frame_buffer_uid (*int*) – the uid of the framebuffer to destroy.

Return type
None

abstract update_uniform(*frame_buffer_uid: int | None*, *draw_call_uid: int | None*, *uniform_name: str*, *value: Any*) → None

Updates the value of a named shader uniform.

Parameters

- **frame_buffer_uid** (*int | None*) – the uid of the framebuffer of the uniform to update. Set to `None` to update across all buffers.
- **draw_call_uid** (*int | None*) – the uid of the draw call of the uniform to update. Set to `None` to update across all buffers.
- **uniform_name** (*str*) – the name of the shader uniform to update.
- **value** (*Any*) – the new value of the shader uniform. (Must be convertible to a GLSL type)

Return type
None

abstract update_vertex_buffer(*frame_buffer_uid: int*, *draw_call_uid: int*, *vertex_array: ndarray[Any, dtype[_ScalarType_co]] | None*, *index_array: ndarray[Any, dtype[_ScalarType_co]] | None*, *vertex_attributes: Tuple[str] | None*) → None

Updates the data inside a vertex buffer.

Parameters

- **frame_buffer_uid** (*int*) – the uid of the framebuffer of the vertex buffer to update.

- **draw_call_uid** (*int*) – the uid of the draw call of the vertex buffer to update.
- **vertex_array** (*ndarray[Any, dtype[_ScalarType_co]] | None*) – a numpy array containing the new vertex data.
- **index_array** (*ndarray[Any, dtype[_ScalarType_co]] | None*) – optionally, a numpy array containing the indices of vertices ordered to make triangles.
- **vertex_attributes** (*Tuple[str] | None*) – a tuple of the names of the vertex attributes to map to in the shader, in the order that they appear in the vertex array.

Return type
None

abstract delete_vertex_buffer(*frame_buffer_uid: int*, *draw_call_uid: int*) → None

Deletes an existing vertex buffer.

Parameters

- **frame_buffer_uid** (*int*) – the uid of the framebuffer of the vertex buffer to delete.
- **draw_call_uid** (*int*) – the uid of the draw call of the vertex buffer to delete.

Return type
None

abstract register_shader(*frame_buffer_uid: int*, *draw_call_uid: int*, *vertex_shader: str*, *fragment_shader: str | None*, *tess_control_shader: str | None*, *tess_evaluation_shader: str | None*, *geometry_shader: str | None*, *compute_shader: str | None*, *primitive_type: str | None = None*) → None

Compiles and registers a shader to a given framebuffer.

Parameters

- **frame_buffer_uid** (*int*) – the uid of the framebuffer to register the shader to.
- **draw_call_uid** (*int*) – the uid of the draw call to register the shader to.
- **vertex_shader** (*str*) – the preprocessed vertex shader GLSL source.
- **fragment_shader** (*str | None*) – the preprocessed fragment shader GLSL source.
- **tess_control_shader** (*str | None*) – the preprocessed tessellation control shader GLSL source.
- **tess_evaluation_shader** (*str | None*) – the preprocessed tessellation evaluation shader GLSL source.
- **geometry_shader** (*str | None*) – the preprocessed geometry shader GLSL source.
- **compute_shader** (*str | None*) – *[Not implemented]* the preprocessed compute shader GLSL source.
- **primitive_type** (*str | None*) – what type of input primitive to treat the vertex data as. One of (“TRIANGLES”, “LINES”, “POINTS), defaults to “TRIANGLES” if `None`.

Return type
None

abstract update_texture(*texture_uid: int*, *data: ndarray[Any, dtype[_ScalarType_co]]*, *uniform_name: str | None*, *override_dtype: str | None*, *rect: Tuple[int, int, int, int] | Tuple[int, int, int, int, int, int] | None*, *treat_as_normalized_integer: bool*) → None

Creates or updates a texture from the NumPy array provided.

Parameters

- **texture_uid** (*int*) – the uid of the texture to create or update.
- **data** (*ndarray[Any, dtype[_ScalarType_co]]*) – a NumPy array containing the image data to copy to the texture.
- **uniform_name** (*str | None*) – the name of the shader uniform to associate this texture with.
- **override_dtype** (*str | None*) – optionally, a moderngl override
- **rect** (*Tuple[int, int, int, int] | Tuple[int, int, int, int, int, int] | None*) – optionally, a rectangle (left, top, right, bottom) specifying the area of the target texture to update.
- **treat_as_normalized_integer** (*bool*) – when enabled, integer types (singed/unsigned) are treated as normalized integers by OpenGL, such that when the texture is sampled values in the texture are mapped to floats in the range [0, 1] or [-1, 1]. See: https://www.khronos.org/opengl/wiki/Normalized_Integer for more details.

Return type
None

abstract update_texture_sampler(*texture_uid: int*, *repeat_x: bool | None*, *repeat_y: bool | None*, *linear_filtering: bool | None*, *linear_mipmap_filtering: bool | None*, *anisotropy: int | None*, *build_mip_maps: bool*) → None

Updates a texture's sampling settings. Parameters set to `None` are not updated.

Parameters

- **texture_uid** (*int*) – the uid of the texture to update.
- **repeat_x** (*bool | None*) – whether the texture should repeat or be clamped in the x-axis.
- **repeat_y** (*bool | None*) – whether the texture should repeat or be clamped in the y-axis.
- **linear_filtering** (*bool | None*) – whether the texture should use nearest neighbour (`False`) or linear (`True`) interpolation.
- **linear_mipmap_filtering** (*bool | None*) – whether different mipmap levels should blend linearly (`True`) or not (`False`).
- **anisotropy** (*int | None*) – the number of anisotropy samples to use. (minimum of 1 = disabled, maximum of 16)
- **build_mip_maps** (*bool*) – when set to `True`, immediately builds mipmaps for the texture.

Return type
None

abstract delete_texture(*texture_uid: int*) → None

Destroys the given texture object.

Parameters
texture_uid (*int*) – the uid of the texture to destroy.

Return type
None

abstract renderdoc_capture_frame(*filename: str | None*) → None

Triggers a frame capture with Renderdoc if it's initialised.

Parameters
filename (*str | None*) – optionally, the filename and path to save the capture with.

Return type
None

abstract set_start_time(*start_time: float*) → None

Sets the renderer's start time; this is used by the renderer to compute the canvas time which is injected into shaders.

Parameters
start_time (*float*) – the start time of the renderer in seconds since the start of the epoch.

Return type
None

pySSV.ssv_render_opengl.**load_render_doc**(*renderdoc_path: str | None = None*) → *RENDERDOC_API_1_6_0*

Parameters
renderdoc_path (*str | None*) –

Return type
RENDERDOC_API_1_6_0

class pySSV.ssv_render_opengl.**RENDERDOC_API_1_6_0**

class pySSV.ssv_render_opengl.**SSVDrawCall**

Stores a reference to all the objects needed to represent a single draw call belonging to a render buffer.

order: int

vertex_buffer: Buffer | None

index_buffer: Buffer | None

vertex_attributes: Tuple[str, ...]

gl_vertex_array: VertexArray | None

shader_program: Program | None

primitive_type: int

release(*needs_gc: bool*, *release_vb: bool = True*)

Parameters
- **needs_gc** (*bool*) –
- **release_vb** (*bool*) –

class pySSV.ssv_render_opengl.**SSVRenderBufferOpenGL**(*order: int*, *needs_gc: bool*, *frame_buffer: Framebuffer*, *render_texture: Texture*, *draw_calls: Dict[int,* SSVDrawCall*]*, *uniform_name: str*)

Stores a reference to all the OpenGL objects needed to render a single render buffer.

Parameters
- **order** (*int*) –
- **needs_gc** (*bool*) –
- **frame_buffer** (*Framebuffer*) –

- **render_texture** (*Texture*) –
- **draw_calls** (*Dict[int, SSVDrawCall]*) –
- **uniform_name** (*str*) –

order: int

needs_gc: bool

frame_buffer: Framebuffer

render_texture: Texture

draw_calls: Dict[int, *SSVDrawCall*]

uniform_name: str

release()

Releases the resources within this render buffer, clearing the draw call list.

class pySSV.ssv_render_opengl.**SSVTextureOpenGL**(*texture: Texture | Texture3D*, *uniform_name: str*)

Stores a reference to an OpenGL texture object.

Parameters

- **texture** (*Texture | Texture3D*) –
- **uniform_name** (*str*) –

texture: Texture | Texture3D

uniform_name: str

release(*needs_gc: bool*)

Parameters
needs_gc (*bool*) –

class pySSV.ssv_render_opengl.**SSVRenderOpenGL**(*gl_version: int | None = None*, *use_renderdoc_api: bool = False*)

A rendering backend for SSV based on OpenGL

Initialises a new renderer (and backing graphics context) with the given options.

Parameters

- **gl_version** (*int | None*) – optionally, the minimum version of OpenGL to support.
- **use_renderdoc_api** (*bool*) – whether the renderer should attempt to load the RenderDoc API.

log_context_info(*full=False*)

Logs the OpenGL information to the console for debugging.

Parameters
full – whether to log *all* of the OpenGL context information (including extensions)

get_context_info() → Dict[str, str]

Returns the OpenGL context information.

Return type
Dict[str, str]

get_supported_extensions() → Set[str]

Gets the set of supported OpenGL shader compiler extensions.

Return type
Set[str]

update_frame_buffer(*frame_buffer_uid: int*, *order: int | None*, *size: Tuple[int, int] | None*, *uniform_name: str | None*, *components: int | None = 4*, *dtype: str | None = 'f1'*)

Updates the resolution/format of the given frame buffer. Note that framebuffer 0 is always used for output. If the given framebuffer id does not exist, it is created.

Setting a parameter to `None` preserves the current value for that frame buffer.

Parameters

- **frame_buffer_uid** (*int*) – the uid of the framebuffer to update/create. Buffer 0 is the output framebuffer.
- **order** (*int | None*) – the sorting order to render the frame buffers in, smaller values are rendered first.
- **size** (*Tuple[int, int] | None*) – the new resolution of the framebuffer.
- **uniform_name** (*str | None*) – the name of the uniform to bind this frame buffer to.
- **components** (*int | None*) – how many vector components should each pixel have (RGB=3, RGBA=4).
- **dtype** (*str | None*) – the data type for each pixel component (see: https://moderngl.readthedocs.io/en/5.8.2/topics/texture_formats.html).

delete_frame_buffer(*frame_buffer_uid: int*)

Destroys the given framebuffer. *Note* that framebuffer 0 can't be destroyed as it is the output framebuffer.

Parameters
frame_buffer_uid (*int*) – the uid of the framebuffer to destroy.

update_uniform(*frame_buffer_uid: int | None*, *draw_call_uid: int | None*, *uniform_name: str*, *value: Any*)

Updates the value of a named shader uniform.

Parameters

- **frame_buffer_uid** (*int | None*) – the uid of the framebuffer of the uniform to update. Set to `None` to update across all buffers.
- **draw_call_uid** (*int | None*) – the uid of the draw call of the uniform to update. Set to `None` to update across all buffers.
- **uniform_name** (*str*) – the name of the shader uniform to update.
- **value** (*Any*) – the new value of the shader uniform. (Must be convertible to a GLSL type)

update_vertex_buffer(*frame_buffer_uid: int*, *draw_call_uid: int*, *vertex_array: ndarray[Any, dtype[_ScalarType_co]] | None*, *index_array: ndarray[Any, dtype[_ScalarType_co]] | None*, *vertex_attributes: Tuple[str] | None*)

Updates the data inside a vertex buffer.

Parameters

- **frame_buffer_uid** (*int*) – the uid of the framebuffer of the vertex buffer to update.
- **draw_call_uid** (*int*) – the uid of the draw call of the vertex buffer to update.

- **vertex_array** (*ndarray[Any, dtype[_ScalarType_co]] | None*) – a numpy array containing the new vertex data.
- **index_array** (*ndarray[Any, dtype[_ScalarType_co]] | None*) – optionally, a numpy array containing the indices of vertices ordered to make triangles.
- **vertex_attributes** (*Tuple[str] | None*) – a tuple of the names of the vertex attributes to map to in the shader, in the order that they appear in the vertex array.

delete_vertex_buffer(*frame_buffer_uid: int*, *draw_call_uid: int*)

Deletes an existing vertex buffer.

Parameters

- **frame_buffer_uid** (*int*) – the uid of the framebuffer of the vertex buffer to delete.
- **draw_call_uid** (*int*) – the uid of the draw call of the vertex buffer to delete.

register_shader(*frame_buffer_uid: int*, *draw_call_uid: int*, *vertex_shader: str*, *fragment_shader: str | None*, *tess_control_shader: str | None*, *tess_evaluation_shader: str | None*, *geometry_shader: str | None*, *compute_shader: str | None*, *primitive_type: str | None = None*)

Compiles and registers a shader to a given framebuffer.

Parameters

- **frame_buffer_uid** (*int*) – the uid of the framebuffer to register the shader to.
- **draw_call_uid** (*int*) – the uid of the draw call to register the shader to.
- **vertex_shader** (*str*) – the preprocessed vertex shader GLSL source.
- **fragment_shader** (*str | None*) – the preprocessed fragment shader GLSL source.
- **tess_control_shader** (*str | None*) – the preprocessed tessellation control shader GLSL source.
- **tess_evaluation_shader** (*str | None*) – the preprocessed tessellation evaluation shader GLSL source.
- **geometry_shader** (*str | None*) – the preprocessed geometry shader GLSL source.
- **compute_shader** (*str | None*) – *[Not implemented]* the preprocessed compute shader GLSL source.
- **primitive_type** (*str | None*) – what type of input primitive to treat the vertex data as. One of ("TRIANGLES", "LINES", "POINTS), defaults to "TRIANGLES" if None.

update_texture(*texture_uid: int*, *data: ndarray[Any, dtype[_ScalarType_co]]*, *uniform_name: str | None*, *override_dtype: str | None*, *rect: Tuple[int, int, int, int] | Tuple[int, int, int, int, int, int] | None*, *treat_as_normalized_integer: bool*)

Creates or updates a texture from the NumPy array provided.

Parameters

- **texture_uid** (*int*) – the uid of the texture to create or update.
- **data** (*ndarray[Any, dtype[_ScalarType_co]]*) – a NumPy array containing the image data to copy to the texture.
- **uniform_name** (*str | None*) – the name of the shader uniform to associate this texture with.
- **override_dtype** (*str | None*) – optionally, a moderngl override

- **rect** (*Tuple[int, int, int, int] | Tuple[int, int, int, int, int, int] | None*) – optionally, a rectangle (left, top, right, bottom) specifying the area of the target texture to update.
- **treat_as_normalized_integer** (*bool*) – when enabled, integer types (singed/unsigned) are treated as normalized integers by OpenGL, such that when the texture is sampled values in the texture are mapped to floats in the range [0, 1] or [-1, 1]. See: https://www.khronos.org/opengl/wiki/Normalized_Integer for more details.

update_texture_sampler(*texture_uid: int*, *repeat_x: bool | None*, *repeat_y: bool | None*, *linear_filtering: bool | None*, *linear_mipmap_filtering: bool | None*, *anisotropy: int | None*, *build_mip_maps: bool*)

Updates a texture's sampling settings. Parameters set to `None` are not updated.

Parameters

- **texture_uid** (*int*) – the uid of the texture to update.
- **repeat_x** (*bool | None*) – whether the texture should repeat or be clamped in the x-axis.
- **repeat_y** (*bool | None*) – whether the texture should repeat or be clamped in the y-axis.
- **linear_filtering** (*bool | None*) – whether the texture should use nearest neighbour (`False`) or linear (`True`) interpolation.
- **linear_mipmap_filtering** (*bool | None*) – whether different mipmap levels should blend linearly (`True`) or not (`False`).
- **anisotropy** (*int | None*) – the number of anisotropy samples to use. (minimum of 1 = disabled, maximum of 16)
- **build_mip_maps** (*bool*) – when set to `True`, immediately builds mipmaps for the texture.

delete_texture(*texture_uid: int*)

Destroys the given texture object.

Parameters

texture_uid (*int*) – the uid of the texture to destroy.

render()

Renders a complete frame.

Returns

whether the frame rendered successfully.

read_frame(*components: int = 4*, *frame_buffer_uid: int = 0*)

Gets the current contents of the frame buffer as a byte array.

Parameters

- **components** (*int*) – how many components to read from the frame (out of `RGBA`).
- **frame_buffer_uid** (*int*) – the frame buffer to read from.

Returns

the contents of the frame buffer as a bytearray in the `RGBA` format.

read_frame_into(*buffer*, *components: int = 4*, *frame_buffer_uid: int = 0*)

Gets the current contents of the frame buffer as a byte array.

Parameters

- **buffer** – the buffer to copy the frame into.

- **components** (*int*) – how many components to read from the frame (out of RGBA).
- **frame_buffer_uid** (*int*) – the frame buffer to read from.

renderdoc_capture_frame(*filename: str | None*)

Triggers a frame capture with Renderdoc if it's initialised.

Parameters

filename (*str | None*) – optionally, the filename and path to save the capture with.

set_start_time(*start_time: float*)

Sets the renderer's start time; this is used by the renderer to compute the canvas time which is injected into shaders.

Parameters

start_time (*float*) – the start time of the renderer in seconds since the start of the epoch.

class pySSV.ssv_render_process_client.**SSVRenderProcessClient**(*backend: str*, *gl_version: int | None = None*, *timeout: float | None = 1*, *use_renderdoc_api: bool = False*)

This class creates, manages, and provides a communication interface for the render process (an SSVRenderProcessServer).

Initialises a new Render Process Client and starts the render process.

Parameters

- **backend** (*str*) – the rendering backend to use.
- **gl_version** (*int | None*) – optionally, the minimum version of OpenGL to support.
- **timeout** (*float | None*) – the render process watchdog timeout, set to None to disable.
- **use_renderdoc_api** (*bool*) – whether the renderdoc_api should be initialised.

property is_alive

subscribe_on_render(*observer: Callable[[bytes], None]*)

Subscribes an event handler to the on_render event, triggered after each frame is rendered.

Parameters

observer (*Callable[[bytes], None]*) – a function to handle the event (must have the signature: *callback(data: bytes) -> None*).

unsubscribe_on_render(*observer: Callable[[bytes], None]*)

Unsubscribes an event handler from the on_render event.

Parameters

observer (*Callable[[bytes], None]*) – a function currently registered to handle the event.

subscribe_on_log(*observer: Callable[[str], None]*)

Subscribes an event handler to the on_log event, triggered when the render process logs a message.

Parameters

observer (*Callable[[str], None]*) – a function to handle the event (must have the signature: *callback(data: bytes) -> None*).

unsubscribe_on_log(*observer: Callable[[str], None]*)

Unsubscribes an event handler from the on_log event.

Parameters

observer (*Callable[[str], None]*) – a function currently registered to handle the event.

update_frame_buffer(*frame_buffer_uid: int*, *order: int | None*, *size: Tuple[int, int] | None*, *uniform_name: str | None*, *components: int | None = 4*, *dtype: str | None = 'f1'*)

Updates the resolution/format of the given frame buffer. Note that framebuffer 0 is always used for output. If the given framebuffer id does not exist, it is created.

Setting a parameter to `None` preserves the current value for that frame buffer.

Parameters

- **frame_buffer_uid** (*int*) – the uid of the framebuffer to update/create. Buffer 0 is the output framebuffer.
- **order** (*int | None*) – the sorting order to render the frame buffers in, smaller values are rendered first.
- **size** (*Tuple[int, int] | None*) – the new resolution of the framebuffer.
- **uniform_name** (*str | None*) – the name of the uniform to bind this frame buffer to.
- **components** (*int | None*) – how many vector components should each pixel have (RGB=3, RGBA=4).
- **dtype** (*str | None*) – the data type for each pixel component (see: https://moderngl.readthedocs.io/en/5.8.2/topics/texture_formats.html).

delete_frame_buffer(*buffer_uid: int*)

Destroys the given framebuffer. *Note* that framebuffer 0 can't be destroyed as it is the output framebuffer.

Parameters

buffer_uid (*int*) – the id of the framebuffer to destroy.

render(*target_framerate: float*, *stream_mode: str*, *encode_quality: float | None = None*)

Starts rendering frames at the given framerate.

Parameters

- **target_framerate** (*float*) – the framerate to render at. Set to -1 to render a single frame.
- **stream_mode** (*str*) – the streaming format to use to send the frames to the widget.
- **encode_quality** (*float | None*) – the encoding quality to use for the given encoding format. Takes a float between 0-100 (some stream modes support values larger than 100, others clamp it internally), where 100 results in the highest quality. This value is scaled to give a bit rate target or quality factor for the chosen encoder. Pass in `None` to use the encoder's default quality settings.

stop()

Kills the render process.

send_heartbeat()

Sends a heartbeat to the render process to keep it alive.

set_timeout(*time: float | None = 1*)

Sets the maximum time the render process will wait for a heartbeat before killing itself. Set to None to disable the watchdog.

Parameters

time (*float | None*) – timeout in seconds.

update_uniform(*frame_buffer_uid: int | None*, *draw_call_uid: int | None*, *uniform_name: str*, *value: Any*)

Updates the value of a named shader uniform.

Parameters

- **frame_buffer_uid** (*int | None*) – the uid of the framebuffer of the uniform to update. Set to None to update across all buffers.
- **draw_call_uid** (*int | None*) – the uid of the draw call of the uniform to update. Set to None to update across all buffers.
- **uniform_name** (*str*) – the name of the shader uniform to update.
- **value** (*Any*) – the new value of the shader uniform. (Must be convertible to a GLSL type)

update_vertex_buffer(*frame_buffer_uid: int*, *draw_call_uid: int*, *vertex_array: ndarray[Any, dtype[_ScalarType_co]] | None*, *index_array: ndarray[Any, dtype[_ScalarType_co]] | None*, *vertex_attributes: Tuple[str, ...] | None*)

Updates the data inside a vertex buffer.

Parameters

- **frame_buffer_uid** (*int*) – the uid of the framebuffer of the vertex buffer to update.
- **draw_call_uid** (*int*) – the uid of the draw call of the vertex buffer to update.
- **vertex_array** (*ndarray[Any, dtype[_ScalarType_co]] | None*) – a numpy array containing the new vertex data.
- **index_array** (*ndarray[Any, dtype[_ScalarType_co]] | None*) – optionally, a numpy array containing the indices of vertices ordered to make triangles.
- **vertex_attributes** (*Tuple[str, ...] | None*) – a tuple of the names of the vertex attributes to map to in the shader, in the order that they appear in the vertex array.

delete_vertex_buffer(*frame_buffer_uid: int*, *draw_call_uid: int*)

Deletes an existing vertex buffer.

Parameters

- **frame_buffer_uid** (*int*) – the uid of the framebuffer of the vertex buffer to delete.
- **draw_call_uid** (*int*) – the uid of the draw call of the vertex buffer to delete.

update_texture(*texture_uid: int*, *data: ndarray[Any, dtype[_ScalarType_co]]*, *uniform_name: str | None*, *override_dtype: str | None*, *rect: Tuple[int, int, int, int] | Tuple[int, int, int, int, int, int] | None*, *treat_as_normalized_integer: bool*)

Creates or updates a texture from the NumPy array provided.

Parameters

- **texture_uid** (*int*) – the uid of the texture to create or update.
- **data** (*ndarray[Any, dtype[_ScalarType_co]]*) – a NumPy array containing the image data to copy to the texture.
- **uniform_name** (*str | None*) – the name of the shader uniform to associate this texture with.
- **override_dtype** (*str | None*) – Optionally, a moderngl override
- **rect** (*Tuple[int, int, int, int] | Tuple[int, int, int, int, int, int] | None*) – optionally, a rectangle (left, top, right, bottom) specifying the area of the target texture to update.

- **treat_as_normalized_integer** (*bool*) – when enabled, integer types (singed/unsigned) are treated as normalized integers by OpenGL, such that when the texture is sampled values in the texture are mapped to floats in the range [0, 1] or [-1, 1]. See: https://www.khronos.org/opengl/wiki/Normalized_Integer for more details.

update_texture_sampler(*texture_uid: int*, *repeat_x: bool | None = None*, *repeat_y: bool | None = None*, *linear_filtering: bool | None = None*, *linear_mipmap_filtering: bool | None = None*, *anisotropy: int | None = None*, *build_mip_maps: bool = False*)

Updates a texture's sampling settings. Parameters set to `None` are not updated.

Parameters

- **texture_uid** (*int*) – the uid of the texture to update.
- **repeat_x** (*bool | None*) – whether the texture should repeat or be clamped in the x-axis.
- **repeat_y** (*bool | None*) – whether the texture should repeat or be clamped in the y-axis.
- **linear_filtering** (*bool | None*) – whether the texture should use nearest neighbour (`False`) or linear (`True`) interpolation.
- **linear_mipmap_filtering** (*bool | None*) – whether different mipmap levels should blend linearly (`True`) or not (`False`).
- **anisotropy** (*int | None*) – the number of anisotropy samples to use. (minimum of 1 = disabled, maximum of 16)
- **build_mip_maps** (*bool*) – when set to `True`, immediately builds mipmaps for the texture.

delete_texture(*texture_uid: int*)

Destroys the given texture object.

Parameters

texture_uid (*int*) – the uid of the texture to destroy.

register_shader(*frame_buffer_uid: int*, *draw_call_uid: int*, *vertex_shader: str*, *fragment_shader: str | None = None*, *tess_control_shader: str | None = None*, *tess_evaluation_shader: str | None = None*, *geometry_shader: str | None = None*, *compute_shader: str | None = None*, *primitive_type: str | None = None*)

Compiles and registers a shader to a given framebuffer.

Parameters

- **frame_buffer_uid** (*int*) – the uid of the framebuffer to register the shader to.
- **draw_call_uid** (*int*) – the uid of the draw call to register the shader to.
- **vertex_shader** (*str*) – the preprocessed vertex shader GLSL source.
- **fragment_shader** (*str | None*) – the preprocessed fragment shader GLSL source.
- **tess_control_shader** (*str | None*) – the preprocessed tessellation control shader GLSL source.
- **tess_evaluation_shader** (*str | None*) – the preprocessed tessellation evaluation shader GLSL source.
- **geometry_shader** (*str | None*) – the preprocessed geometry shader GLSL source.
- **compute_shader** (*str | None*) – *[Not implemented]* the preprocessed compute shader GLSL source.
- **primitive_type** (*str | None*) – what type of input primitive to treat the vertex data as. One of ("TRIANGLES", "LINES", "POINTS), defaults to "TRIANGLES" if `None`.

renderdoc_capture_frame(*filename: str | None*)

Triggers a frame capture with Renderdoc if it's initialised.

Parameters
filename (*str | None*) – optionally, the filename and path to save the capture with.

set_start_time(*start_time: float*) → None

Sets the renderer's start time; this is used by the renderer to compute the canvas time which is injected into shaders.

Parameters
start_time (*float*) – the start time of the renderer in seconds since the start of the epoch.

Return type
None

get_context_info(*timeout: float | None = None*) → Dict[str, str] | None

Returns the OpenGL context information.

Parameters
timeout (*float | None*) – the maximum amount of time in seconds to wait for the result. Set to `None` to wait indefinitely.

Return type
Dict[str, str] | None

get_frame_times(*timeout: float | None = None*) → Tuple[float, float, float, float] | None

Gets the frame time statistics from the renderer. This function is blocking and shouldn't be called too often.

Returns the following statistics:

- avg_frame_time: Average time taken to render a frame (calculated each frame using: `last_avg_frame_time * 0.9 + frame_time * 0.1`)
- max_frame_time: The maximum frame time since this function was last called. (Using `dbg_log_frame_times` interferes with this value.)
- avg_encode_time: Average time taken to encode a frame for streaming (calculated each frame using: `last_avg_encode_time * 0.9 + encode_time * 0.1`)
- max_encode_time: The maximum time taken to encode a frame for streaming since this function was last called. (Using `dbg_log_frame_times` interferes with this value.)

Parameters
timeout (*float | None*) – the maximum amount of time in seconds to wait for the result. Set to `None` to wait indefinitely.

Returns
(avg_frame_time, max_frame_time, avg_encode_time, max_encode_time)

Return type
Tuple[float, float, float, float] | None

get_supported_extensions(*timeout: float | None = None*) → Set[str] | None

Gets the set of supported OpenGL shader compiler extensions.

Parameters
timeout (*float | None*) – the maximum amount of time in seconds to wait for the result. Set to `None` to wait indefinitely.

Return type
Set[str] | None

save_image(*image_type: SSVStreamingMode*, *quality: float*, *size: Tuple[int, int] | None*, *render_buffer: int*, *suppress_ui: bool*) → *Future*[bytes]

Saves the current frame as an image.

Parameters

- **image_type** (*SSVStreamingMode*) – the image compression algorithm to use.
- **quality** (*float*) – the encoding quality to use for the given encoding format. Takes a float between 0-100 (some stream modes support values larger than 100, others clamp it internally), where 100 results in the highest quality. This value is scaled to give a bit rate target or quality factor for the chosen encoder.
- **size** (*Tuple[int, int] | None*) – optionally, the width and height of the saved image. If set to `None` uses the current resolution of the render buffer.
- **render_buffer** (*int*) – the uid of the render buffer to save.
- **suppress_ui** (*bool*) – whether any active SSVGUIs should be suppressed.

Returns
the bytes representing the compressed image.

Return type
Future[bytes]

dbg_log_context_info(*full=False*)

Logs the OpenGL information to the console for debugging.

Parameters
full – whether to log *all* of the OpenGL context information (including extensions).

dbg_log_frame_times(*enabled=True*)

Enables or disables frame time logging.

Parameters
enabled – whether to log frame times.

dbg_render_test()

DEPRECATED

[For debugging only] Sets up the pipeline to render with a demo shader.

dbg_render_command(*command: str*, **args*)

[For debugging only] Sends a custom command to the render process.

Parameters

- **command** (*str*) – the custom command to send
- **args** – the arguments to send with the command

class pySSV.ssv_render_process_server.**SSVRenderProcessLogger**(*tx_queue: Queue*)

A StringIO pipe for sending log messages to, this class pipes incoming messages to "LogM" commands.

Parameters
tx_queue (*Queue*) –

write(*text: str*, *severity: int = 21*) → int

Called to write a log message to the stream.

Parameters

- **text** (*str*) – the message to log.
- **severity** (*int*) – the severity to log the message with.

Returns
the number of characters written to the stream.

Return type
int

class pySSV.ssv_render_process_server.**SSVRenderProcessServer**(*backend: str*, *gl_version: int | None*, *command_queue_tx: Queue*, *command_queue_rx: Queue*, *log_severity: int*, *timeout: float | None*, *use_renderdoc_api: bool = False*)

This class listens for render commands and dispatches them to the renderer. This class is intended to be constructed in a dedicated process by SSVRenderProcessClient.

Parameters

- **backend** (*str*) –
- **gl_version** (*int | None*) –
- **command_queue_tx** (*Queue*) –
- **command_queue_rx** (*Queue*) –
- **log_severity** (*int*) –
- **timeout** (*float | None*) –
- **use_renderdoc_api** (*bool*) –

Jupyter Widget

pySSV.ssv_render_widget.**OnSaveImageDelegate**

A callable with parameters matching the signature:

```
on_save_image(image_type: SSVStreamingMode, quality: float, size:␣
↪Optional[Tuple[int, int]], render_buffer: int):
    ...
```

image_type: the image codec to save the image with.
quality: a value between 0-100, indicating the image quality (larger values are higher quality). For png this represents compression quality (higher values result in smaller files, but take longer to compress).
size: the resolution of the saved image. When set to None, uses the current resolution of the render buffer (this also prevents an additional frame from being rendered).
render_buffer: the uid of the render buffer to save.
suppress_ui: whether any active SSVGUIs should be supressed.

alias of Callable[[*SSVStreamingMode*, float, Optional[Tuple[int, int]], int, bool], None]

class pySSV.ssv_render_widget.**SSVRenderWidget**(***kwargs: Any*)

An SSV Render Widget manages the state and communication of the Jupyter Widget responsible for displaying rendered results embedded in a Jupyter notebook.

Public constructor

streaming_mode

An enum whose value must be in a given sequence.

stream_data_binary

A trait for byte strings.

stream_data_ascii

A trait for unicode strings.

use_websockets

A boolean (True, False) trait.

websocket_url

A trait for unicode strings.

canvas_width

An int trait.

canvas_height

An int trait.

status_connection

A boolean (True, False) trait.

status_logs

A trait for unicode strings.

mouse_pos_x

An int trait.

mouse_pos_y

An int trait.

enable_renderdoc

A boolean (True, False) trait.

frame_rate

A float trait.

frame_times

A trait for unicode strings.

on_heartbeat(*callback: Callable[[], None]*, *remove=False*)

Register a callback to execute when the widget receives a heartbeat from the client.

Parameters

- **callback** (*Callable[[], None]*) – the function to be called when a heartbeat is received.
- **remove** – set to true to remove the callback from the list of callbacks.

on_play(*callback: Callable[[], None]*, *remove=False*)

Register a callback to execute when the widget's play button is pressed.

Parameters

- **callback** (*Callable[[], None]*) – the function to be called when the play button is pressed.
- **remove** – set to true to remove the callback from the list of callbacks.

on_stop(*callback: Callable[[], None]*, *remove=False*)

Register a callback to execute when the widget's stop button is pressed.

Parameters

- **callback** (*Callable[[], None]*) – the function to be called when the stop button is pressed.
- **remove** – set to true to remove the callback from the list of callbacks.

on_click(*callback: Callable[[bool, int], None]*, *remove=False*)

Register a callback to execute when the widget receives a mouseup or mousedown event.

Parameters

- **callback**(*Callable[[bool, int], None]*) – the function to be called when the event is raised.
- **remove** – set to true to remove the callback from the list of callbacks.

on_key(*callback: Callable[[str, bool], None]*, *remove=False*)

Register a callback to execute when the widget receives a keyup or keydown event.

Parameters

- **callback**(*Callable[[str, bool], None]*) – the function to be called when the event is raised.
- **remove** – set to true to remove the callback from the list of callbacks.

on_mouse_wheel(*callback: Callable[[float], None]*, *remove=False*)

Register a callback to execute when the widget receives a mouse wheel event.

Parameters

- **callback** (*Callable[[float], None]*) – the function to be called when the event is raised.
- **remove** – set to true to remove the callback from the list of callbacks.

on_save_image(*callback: Callable[[SSVStreamingMode, float, Tuple[int, int] | None, int, bool], None]*, *remove=False*)

Register a callback to execute when the widget's 'save image' button is pressed.

Parameters

- **callback** (*Callable[[SSVStreamingMode, float, Tuple[int, int] | None, int, bool], None]*) – the function to be called when the event is raised.
- **remove** – set to true to remove the callback from the list of callbacks.

on_renderdoc_capture(*callback: Callable[[], None]*, *remove=False*)

Register a callback to execute when the widget's renderdoc capture button is pressed.

Parameters

- **callback** (*Callable[[], None]*) – the function to be called when the event is raised.
- **remove** – set to true to remove the callback from the list of callbacks.

download_file(*filename: str*, *data: bytes*)

Triggers a file download in the client's web browser.

Parameters

- **filename** (*str*) – the file name of the file to download.
- **data** (*bytes*) – the data to be downloaded.

class pySSV.ssv_render_widget.**SSVRenderWidgetLogIO**(*widget: SSVRenderWidget*)

Parameters

widget (SSVRenderWidget) –

write(*_SSVRenderWidgetLogIO__s: str*) → int

Write string to stream. Returns the number of characters written (which is always equal to the length of the string).

Parameters

_SSVRenderWidgetLogIO__s (*str*) –

Return type

int

class pySSV.ssv_canvas_stream_server.**SSVCanvasStreamServerHTTP**(*msg_queue: SimpleQueue*, *is_alive: Callable[[], bool]*, **args*, ***kwargs*)

Parameters

- **msg_queue** (*SimpleQueue*) –
- **is_alive** (*Callable[[], bool]*) –

do_GET()

class pySSV.ssv_canvas_stream_server.**SSVCanvasStreamServer**(*http: bool = False*, *port: int | None = None*, *timeout: float = 10*)

A basic websocket/http server which serves frame data to the SSV canvas.

Parameters

- **http** (*bool*) –
- **port** (*int | None*) –
- **timeout** (*float*) –

property url: str

Gets the URL of this websocket.

property is_alive

close()

Shuts down the websocket server.

send(*msg: bytes*)

Sends a bytes packet to the websocket.

Parameters
msg (*bytes*) – the packet to send.

heartbeat()

Sends a heartbeat to the server to keep it alive.

Shader Preprocessor

class pySSV.ssv_shader_preprocessor.**SSVShaderPreprocessor**(*gl_version: str*, *supports_line_directives: bool*)

This class is responsible for preprocessing shader source code and shader templates into platform specific, ready to compile shaders for each pipeline stage.

Parameters

- **gl_version** (*str*) –
- **supports_line_directives** (*bool*) –

property global_defines: Dict[str, str]

Gets the (mutable) dictionary of compiler macros which are defined globally in all shaders. This does not include any macros defined by the `SSVShaderPreprocessor` itself. The dict contains key-value pairs where the key is the macro name and the value is the macro definition (as a C preprocessor macro).

preprocess(*source: str*, *filepath: str | None = None*, *additional_template_directory: str | None = None*, *additional_templates: List[str] | None = None*, *shader_defines: Dict[str, str] | None = None*, *compiler_extensions: List[str] | None = None*) → Dict[str, str]

Preprocesses an SSV shader into multiple processed shaders for each pipeline.

Parameters

- **source** (*str*) – the shader source code to preprocess. It should contain the necessary `#pragma SSV <template_name>` directive see *Built In Shader Templates* for more information.
- **filepath** (*str | None*) – the filepath of the shader, used to help the preprocessor give more meaningful error messages.
- **additional_template_directory** (*str | None*) – a path to a directory containing custom shader templates. See *Writing Shader Templates* for information about using custom shader templates.
- **additional_templates** (*List[str] | None*) – a list of custom shader templates (source code, not paths). See *Writing Shader Templates* for information about using custom shader templates.
- **shader_defines** (*Dict[str, str] | None*) – extra preprocessor defines to be enabled globally.
- **compiler_extensions** (*List[str] | None*) – a list of GLSL extensions required by this shader (eg: `GL_EXT_control_flow_attributes`)

Returns
a dict of compiled shaders for each of the required pipeline stages.

Return type
Dict[str, str]

dbg_query_shader_templates(*additional_template_directory: str | None = None*) → List[*SSVTemplatePragmaData*]

Gets a list of all the shader templates available to the preprocessor.

Parameters
additional_template_directory (*str | None*) – a path to a directory containing custom shader templates.

Returns
A list of shader template metadata definitions.

Return type
List[SSVTemplatePragmaData]

dbg_query_shader_template(*template_name: str*, *additional_template_directory: str | None = None*, *additional_templates: List[str] | None = None*) → str

Gets the list of arguments a given shader template expects and returns a string containing their usage info.

Parameters

- **template_name** (*str*) – the name of the template to look for.
- **additional_template_directory** (*str | None*) – a path to a directory containing custom shader templates.
- **additional_templates** (*List[str] | None*) – a list of custom shader templates (source code, not paths).

Returns
the shader template's auto generated help string.

Return type
str

add_dynamic_uniform(*name: str*, *glsl_type: str*)

Adds a uniform declaration to _DYNAMIC_UNIFORMS macro.

Parameters

- **name** (*str*) – the name of the uniform to add. Must be a valid GLSL identifier.
- **glsl_type** (*str*) – the glsl type of the uniform.

remove_dynamic_uniform(*name: str*)

Removes a uniform declaration from the dynamic uniforms.

Parameters
name (*str*) – the name of the uniform to remove.

class pySSV.ssv_shader_source_preprocessor.**SSVShaderSourcePreprocessor**(*shader_source*)

This class is responsible for the actual preprocessing of the shader template. It extends the `pcpp.Preprocessor` to add functionality needed to support our custom #include and #pragma directives.

class pySSV.ssv_pragma_parser.**SSVTemplatePragmaData**(***kwargs*)

command: str

name: str

author: str | None = None

description: str | None = None

shader_stage: List[str] | None = None

non_positional: bool = False

action: str | None = None

default: str | None = None

choices: List[str] | None = None

const: str | None = None

primitive_type: str | None = None

class pySSV.ssv_pragma_parser.**SSVShaderPragmaData**(***kwargs*)

template: str

args: List[str] = []

class pySSV.ssv_pragma_parser.**SSVTemplatePragmaParser**

This class is responsible for parsing #pragma definitions in SSV shader templates.

Refer to *Writing Shader Templates* for details on writing shader templates.

on_include_not_found(*is_malformed*, *is_system_include*, *curdir*, *includepath*)

Used internally by the parser.

on_error(*file*, *line*, *msg*)

Used internally by the parser.

on_directive_unknown(*directive*, *toks*, *ifpassthru*, *precedingtoks*)

Used internally by the parser.

parse(*input*, *source=None*, *ignore=None*) → Dict[str, List[*SSVTemplatePragmaData*]]

Parses the #pragma directives of a shader template.

Parameters

- **input** – the source of the shader template.
- **source** – the path to the source file.
- **ignore** –

Returns
a dictionary of parsed shader template commands.

Return type
Dict[str, *List*[SSVTemplatePragmaData]]

write(*oh=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

The pragma parser only parses #pragma arguments, it should not be used to preprocess shader files.

Parameters
oh – unused

class pySSV.ssv_pragma_parser.**SSVShaderPragmaParser**

This class is responsible for parsing #pragma definitions in SSV shaders.

on_directive_unknown(*directive*, *toks*, *ifpassthru*, *precedingtoks*)

Used internally by the parser.

parse(*input*, *source=None*, *ignore=None*) → *SSVShaderPragmaData*

Parses the #pragma directives of a shader.

Parameters

- **input** – the source of the shader.
- **source** – the path to the source file.
- **ignore** –

Returns
an object of parsed shader template info.

Return type
SSVShaderPragmaData

write(*oh=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

The pragma parser only parses #pragma arguments, it should not be used to preprocess shader files.

Parameters
oh – unused

class pySSV.ssv_shader_args_tokenizer.**SSVShaderArgsTokenizer**

static correct_tokens(*tokens: List[Any]*, *preprocessor: Preprocessor*)

This method takes a list of PCPP LexTokens and converts them to a list of arguments for argparse.

This method handles string un-escaping and whitespace stripping.

Parameters

- **tokens** (*List[Any]*) – the list of PCPP tokens to parse.
- **preprocessor** (*Preprocessor*) – the preprocessor used to generate the tokens.

Returns
a list of arguments.

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

R

S

T

U

V

W

X

Y

Z