# Almog Draw Library

Generated by Doxygen 1.9.1

# Chapter 1

# Class Index

## 1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

# Chapter 2

# File Index

## 2.1 File List

Here is a list of all files with brief descriptions:

# Chapter 3

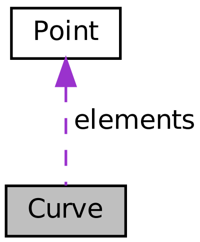
# Class Documentation

## 3.1 Curve Struct Reference

Polyline of points with a uniform color.

`#include <Almog_Draw_Library.h>`

Collaboration diagram for Curve:




### Public Attributes

- uint32_t color
- size_t length
- size_t capacity
- Point ∗ elements

### 3.1.1 Detailed Description

Polyline of points with a uniform color.

Definition at line 94 of file Almog_Draw_Library.h.

## 3.1.2 Member Data Documentation

### 3.1.2.1 capacity

```
size_t Curve::capacity
```

Allocated capacity.

Definition at line 97 of file Almog_Draw_Library.h.

### 3.1.2.2 color

```
uint32_t Curve::color
```

ARGB color (0xAARRGGBB) for the entire curve.

Definition at line 95 of file Almog_Draw_Library.h.

Referenced by adl_curve_add_to_figure(), and adl_curves_plot_on_figure().

### 3.1.2.3 elements

```
Point* Curve::elements
```

Point array.

Definition at line 98 of file Almog_Draw_Library.h.

Referenced by adl_curves_plot_on_figure(), adl_grid_draw(), and setup().

### 3.1.2.4 length

```
size_t Curve::length
```

Number of points used.

Definition at line 96 of file Almog_Draw_Library.h.

Referenced by adl_curves_plot_on_figure(), adl_grid_draw(), and setup().

The documentation for this struct was generated from the following file:

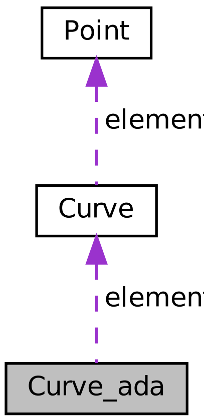
- Almog_Draw_Library.h

## 3.2 Curve_ada Struct Reference

Dynamic array of curves (polyline container).

`#include <Almog_Draw_Library.h>`

Collaboration diagram for Curve_ada:




### Public Attributes

- size_t length
- size_t capacity
- Curve ∗ elements

### 3.2.1 Detailed Description

Dynamic array of curves (polyline container).

Definition at line 107 of file Almog_Draw_Library.h.

### 3.2.2 Member Data Documentation

#### 3.2.2.1 capacity

`size_t Curve_ada::capacity`

Allocated capacity.

Definition at line 109 of file Almog_Draw_Library.h.

#### 3.2.2.2 elements

`Curve* Curve_ada::elements`

Curves array.

Definition at line 110 of file Almog_Draw_Library.h.

Referenced by adl_curves_plot_on_figure(), and adl_grid_draw().

#### 3.2.2.3 length

`size_t Curve_ada::length`

Number of curves used.

Definition at line 108 of file Almog_Draw_Library.h.

Referenced by adl_curves_plot_on_figure(), and adl_grid_draw().

The documentation for this struct was generated from the following file:

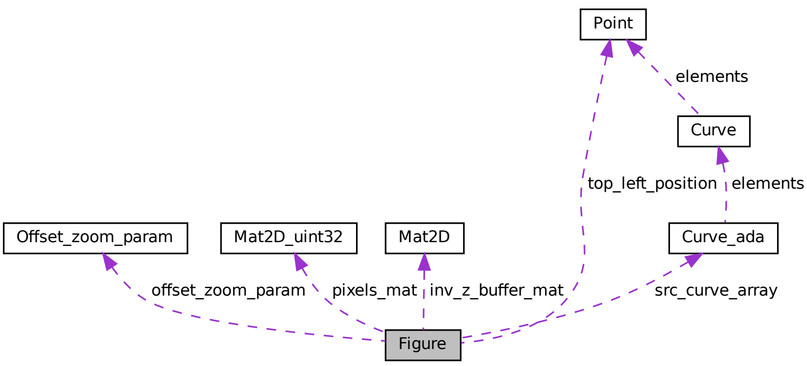
- Almog_Draw_Library.h

## 3.3 Figure Struct Reference

Plotting figure holding a pixel buffer, z-buffer and plot state.

`#include <Almog_Draw_Library.h>`

Collaboration diagram for Figure:



## Public Attributes

- int min_x_pixel
- int max_x_pixel
- int min_y_pixel
- int max_y_pixel
- float min_x
- float max_x
- float min_y
- float max_y
- int x_axis_head_size
- int y_axis_head_size
- Offset_zoom_param offset_zoom_param
- Curve_ada src_curve_array
- Point top_left_position
- Mat2D_uint32 pixels_mat
- Mat2D inv_z_buffer_mat
- uint32_t background_color
- bool to_draw_axis
- bool to_draw_max_min_values

### 3.3.1 Detailed Description

Plotting figure holding a pixel buffer, z-buffer and plot state.

A Figure owns an internal pixel buffer and an inverse-Z buffer used by the plotting utilities. It also stores axis extents, paddings and appearance flags.

Definition at line 174 of file Almog_Draw_Library.h.

### 3.3.2 Member Data Documentation

#### 3.3.2.1 background_color

```
uint32_t Figure::background_color
```

Clear color for figure.

Definition at line 195 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_curves_plot_on_figure(), and setup().

#### 3.3.2.2 inv_z_buffer_mat

```
Mat2D Figure::inv_z_buffer_mat
```

Owned inverse-Z buffer (double).

Definition at line 193 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_curves_plot_on_figure(), and adl_figure_alloc().

#### 3.3.2.3 max_x

```
float Figure::max_x
```

Max X value in source data.

Definition at line 181 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_curve_add_to_figure(), adl_curves_plot_on_figure(), adl_figure_alloc(), and adl_max_min_values_draw_on_figure().

#### 3.3.2.4 max_x_pixel

```
int Figure::max_x_pixel
```

Right bound (pixel space).

Definition at line 176 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_axis_draw_on_figure(), adl_curves_plot_on_figure(), adl_figure_alloc(), and adl_max_min_values_draw_on_figure().

#### 3.3.2.5 max_y

```
float Figure::max_y
```

Max Y value in source data.

Definition at line 183 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_curve_add_to_figure(), adl_curves_plot_on_figure(), adl_figure_alloc(), and adl_max_min_values_draw_on_figure().

#### 3.3.2.6 max_y_pixel

```
int Figure::max_y_pixel
```

Bottom bound (pixel space).

Definition at line 178 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_axis_draw_on_figure(), adl_curves_plot_on_figure(), adl_figure_alloc(), and adl_max_min_values_draw_on_figure().

#### 3.3.2.7 min_x

```
float Figure::min_x
```

Min X value in source data.

Definition at line 180 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_curve_add_to_figure(), adl_curves_plot_on_figure(), adl_figure_alloc(), and adl_max_min_values_draw_on_figure().

#### 3.3.2.8 min_x_pixel

```
int Figure::min_x_pixel
```

Left padding (pixel space).

Definition at line 175 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_axis_draw_on_figure(), adl_curves_plot_on_figure(), adl_figure_alloc(), and adl_max_min_values_draw_on_figure().

#### 3.3.2.9 min_y

```
float Figure::min_y
```

Min Y value in source data.

Definition at line 182 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_curve_add_to_figure(), adl_curves_plot_on_figure(), adl_figure_alloc(), and adl_max_min_values_draw_on_figure().

#### 3.3.2.10 min_y_pixel

```
int Figure::min_y_pixel
```

Top padding (pixel space).

Definition at line 177 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_axis_draw_on_figure(), adl_curves_plot_on_figure(), adl_figure_alloc(), and adl_max_min_values_draw_on_figure().

#### 3.3.2.11 offset_zoom_param

```
Offset_zoom_param Figure::offset_zoom_param
```

Pan/zoom parameters.

Definition at line 188 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_axis_draw_on_figure(), adl_curves_plot_on_figure(), adl_figure_alloc(), and adl_max_min_values_draw_on_figure().

#### 3.3.2.12 pixels_mat

```
Mat2D_uint32 Figure::pixels_mat
```

Owned ARGB pixel buffer.

Definition at line 192 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_axis_draw_on_figure(), adl_curves_plot_on_figure(), adl_figure_alloc(), adl_figure_copy_to_screen(), and adl_max_min_values_draw_on_figure().

#### 3.3.2.13 src_curve_array

```
Curve_ada Figure::src_curve_array
```

Curves to plot.

Definition at line 189 of file Almog_Draw_Library.h.

Referenced by adl_curve_add_to_figure(), adl_curves_plot_on_figure(), and adl_figure_alloc().

#### 3.3.2.14 to_draw_axis

```
bool Figure::to_draw_axis
```

Draw axes when plotting.

Definition at line 196 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_curves_plot_on_figure(), and setup().

#### 3.3.2.15 to_draw_max_min_values

```
bool Figure::to_draw_max_min_values
```

Draw min/max labels.

Definition at line 197 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_curves_plot_on_figure(), and setup().

#### 3.3.2.16 top_left_position

```
Point Figure::top_left_position
```

On-screen copy position.

Definition at line 190 of file Almog_Draw_Library.h.

Referenced by adl_figure_alloc(), and adl_figure_copy_to_screen().

#### 3.3.2.17 x_axis_head_size

```
int Figure::x_axis_head_size
```

Computed X-axis arrow head size (px).

Definition at line 185 of file Almog_Draw_Library.h.

Referenced by adl_axis_draw_on_figure(), and adl_max_min_values_draw_on_figure().

#### 3.3.2.18 y_axis_head_size

```
int Figure::y_axis_head_size
```

Computed Y-axis arrow head size (px).

Definition at line 186 of file Almog_Draw_Library.h.

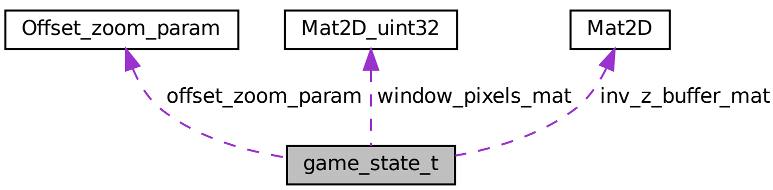
Referenced by adl_axis_draw_on_figure(), and adl_max_min_values_draw_on_figure().

The documentation for this struct was generated from the following file:

- Almog_Draw_Library.h

## 3.4 game_state_t Struct Reference

Collaboration diagram for game_state_t:




### Public Attributes

- int game_is_running
- float delta_time
- float elapsed_time
- float const_fps
- float fps
- float frame_target_time
- int to_render
- int to_update
- size_t previous_frame_time
- int left_button_pressed
- int to_limit_fps
- int to_clear_renderer
- int space_bar_was_pressed
- int w_was_pressed
- int s_was_pressed
- int a_was_pressed
- int d_was_pressed
- int e_was_pressed
- int q_was_pressed

- SDL_Window * window
- int window_w
- int window_h
- SDL_Renderer * renderer
- TTF_Font * font
- SDL_Surface * window_surface
- SDL_Texture * window_texture
- Mat2D_uint32 window_pixels_mat
- Mat2D inv_z_buffer_mat
- Offset_zoom_param offset_zoom_param

## 3.4.1 Detailed Description

Definition at line 38 of file display.c.

## 3.4.2 Member Data Documentation

### 3.4.2.1 a_was_pressed

`int game_state_t::a_was_pressed`

Definition at line 55 of file display.c.

Referenced by main().

### 3.4.2.2 const_fps

`float game_state_t::const_fps`

Definition at line 42 of file display.c.

Referenced by main(), setup(), and update_window().

### 3.4.2.3 d_was_pressed

`int game_state_t::d_was_pressed`

Definition at line 56 of file display.c.

Referenced by main().

#### 3.4.2.4 delta_time

```
float game_state_t::delta_time
```

Definition at line 40 of file display.c.

Referenced by fix_framerate(), main(), and update_window().

#### 3.4.2.5 e_was_pressed

```
int game_state_t::e_was_pressed
```

Definition at line 57 of file display.c.

Referenced by main().

#### 3.4.2.6 elapsed_time

```
float game_state_t::elapsed_time
```

Definition at line 41 of file display.c.

Referenced by main(), and update_window().

#### 3.4.2.7 font

```
TTF_Font* game_state_t::font
```

Definition at line 64 of file display.c.

Referenced by main().

#### 3.4.2.8 fps

```
float game_state_t::fps
```

Definition at line 43 of file display.c.

Referenced by main(), and update_window().

#### 3.4.2.9 frame_target_time

```
float game_state_t::frame_target_time
```

Definition at line 44 of file display.c.

Referenced by fix_framerate(), main(), and update_window().

#### 3.4.2.10 game_is_running

```
int game_state_t::game_is_running
```

Definition at line 39 of file display.c.

Referenced by main(), and process_input_window().

#### 3.4.2.11 inv_z_buffer_mat

```
Mat2D game_state_t::inv_z_buffer_mat
```

Definition at line 70 of file display.c.

Referenced by check_window_mat_size(), render(), render_window(), and setup_window().

#### 3.4.2.12 left_button_pressed

```
int game_state_t::left_button_pressed
```

Definition at line 48 of file display.c.

Referenced by main(), and process_input_window().

#### 3.4.2.13 offset_zoom_param

```
Offset_zoom_param game_state_t::offset_zoom_param
```

Definition at line 72 of file display.c.

Referenced by main(), process_input_window(), and render().

#### 3.4.2.14 previous_frame_time

```
size_t game_state_t::previous_frame_time
```

Definition at line 47 of file display.c.

Referenced by fix_framerate(), main(), and process_input_window().

#### 3.4.2.15 q_was_pressed

```
int game_state_t::q_was_pressed
```

Definition at line 58 of file display.c.

Referenced by main().

#### 3.4.2.16 renderer

```
SDL_Renderer* game_state_t::renderer
```

Definition at line 63 of file display.c.

Referenced by destroy_window(), initialize_window(), and main().

#### 3.4.2.17 s_was_pressed

```
int game_state_t::s_was_pressed
```

Definition at line 54 of file display.c.

Referenced by main().

#### 3.4.2.18 space_bar_was_pressed

```
int game_state_t::space_bar_was_pressed
```

Definition at line 52 of file display.c.

Referenced by main(), and process_input_window().

#### 3.4.2.19 to_clear_renderer

```
int game_state_t::to_clear_renderer
```

Definition at line 50 of file display.c.

Referenced by main(), and render_window().

#### 3.4.2.20 to_limit_fps

```
int game_state_t::to_limit_fps
```

Definition at line 49 of file display.c.

Referenced by fix_framerate(), main(), setup(), and update_window().

#### 3.4.2.21 to_render

```
int game_state_t::to_render
```

Definition at line 45 of file display.c.

Referenced by main(), and process_input_window().

#### 3.4.2.22 to_update

```
int game_state_t::to_update
```

Definition at line 46 of file display.c.

Referenced by main(), and process_input_window().

#### 3.4.2.23 w_was_pressed

```
int game_state_t::w_was_pressed
```

Definition at line 53 of file display.c.

Referenced by main().

#### 3.4.2.24 window

```
SDL_Window* game_state_t::window
```

Definition at line 60 of file display.c.

Referenced by check_window_mat_size(), destroy_window(), initialize_window(), main(), render_window(), setup_window(), and update_window().

#### 3.4.2.25 window_h

```
int game_state_t::window_h
```

Definition at line 62 of file display.c.

Referenced by check_window_mat_size(), initialize_window(), main(), setup_window(), and update_window().

#### 3.4.2.26 window_pixels_mat

```
Mat2D_uint32 game_state_t::window_pixels_mat
```

Definition at line 69 of file display.c.

Referenced by check_window_mat_size(), copy_mat_to_surface_RGB(), destroy_window(), render(), render_window(), and setup_window().

#### 3.4.2.27 window_surface

```
SDL_Surface* game_state_t::window_surface
```

Definition at line 66 of file display.c.

Referenced by check_window_mat_size(), copy_mat_to_surface_RGB(), destroy_window(), and setup_window().

#### 3.4.2.28 window_texture

```
SDL_Texture* game_state_t::window_texture
```

Definition at line 67 of file display.c.

Referenced by destroy_window().

#### 3.4.2.29 window_w

```
int game_state_t::window_w
```

Definition at line 61 of file display.c.

Referenced by check_window_mat_size(), initialize_window(), main(), setup_window(), and update_window().

The documentation for this struct was generated from the following file:

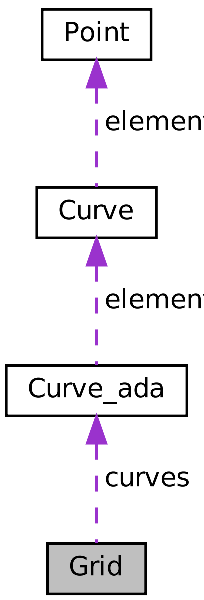
- display.c

## 3.5 Grid Struct Reference

Grid definition (as lines) in a chosen plane.

```
#include <Almog_Draw_Library.h>
```

Collaboration diagram for Grid:




### Public Attributes

- Curve_ada curves
- float min_e1
- float max_e1
- float min_e2
- float max_e2
- int num_samples_e1
- int num_samples_e2
- float de1
- float de2
- char plane [3]

### 3.5.1 Detailed Description

Grid definition (as lines) in a chosen plane.

Definition at line 203 of file Almog_Draw_Library.h.

### 3.5.2 Member Data Documentation

#### 3.5.2.1 curves

`Curve_ada Grid::curves`

Line segments implementing the grid.

Definition at line 204 of file Almog_Draw_Library.h.

Referenced by adl_cartesian_grid_create(), and adl_grid_draw().

#### 3.5.2.2 de1

`float Grid::de1`

Step size along axis 1.

Definition at line 213 of file Almog_Draw_Library.h.

Referenced by adl_cartesian_grid_create().

#### 3.5.2.3 de2

`float Grid::de2`

Step size along axis 2.

Definition at line 214 of file Almog_Draw_Library.h.

Referenced by adl_cartesian_grid_create().

#### 3.5.2.4 max_e1

```
float Grid::max_e1
```

Axis 1 max.

Definition at line 207 of file Almog_Draw_Library.h.

Referenced by adl_cartesian_grid_create().

#### 3.5.2.5 max_e2

```
float Grid::max_e2
```

Axis 2 max.

Definition at line 209 of file Almog_Draw_Library.h.

Referenced by adl_cartesian_grid_create().

#### 3.5.2.6 min_e1

```
float Grid::min_e1
```

Axis 1 min.

Definition at line 206 of file Almog_Draw_Library.h.

Referenced by adl_cartesian_grid_create().

#### 3.5.2.7 min_e2

```
float Grid::min_e2
```

Axis 2 min.

Definition at line 208 of file Almog_Draw_Library.h.

Referenced by adl_cartesian_grid_create().

#### 3.5.2.8 num_samples_e1

```
int Grid::num_samples_e1
```

Number of divisions along axis 1.

Definition at line 211 of file Almog_Draw_Library.h.

Referenced by adl_cartesian_grid_create().

#### 3.5.2.9 num_samples_e2

```
int Grid::num_samples_e2
```

Number of divisions along axis 2.

Definition at line 212 of file Almog_Draw_Library.h.

Referenced by adl_cartesian_grid_create().

#### 3.5.2.10 plane

```
char Grid::plane[3]
```

Plane tag: "XY","XZ","YZ","YX","ZX","ZY".

Definition at line 216 of file Almog_Draw_Library.h.

Referenced by adl_cartesian_grid_create().

The documentation for this struct was generated from the following file:

- Almog_Draw_Library.h

## 3.6 Mat2D Struct Reference

Dense row-major matrix of doubles.

```
#include <Matrix2D.h>
```

### Public Attributes

- size_t rows
- size_t cols
- size_t stride_r
- double * elements

### 3.6.1 Detailed Description

Dense row-major matrix of doubles.

- rows: number of rows (height)
- cols: number of columns (width)
- stride_r: number of elements between successive rows in memory (for contiguous storage, stride_r == cols)
- elements: pointer to contiguous storage of size rows ∗ cols

Definition at line 81 of file Matrix2D.h.

### 3.6.2 Member Data Documentation

#### 3.6.2.1 cols

```
size_t Mat2D::cols
```

Definition at line 83 of file Matrix2D.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_curves_plot_on_figure(), adl_figure_alloc(), mat2D_add(), mat2D_add_col_to_col(), mat2D_add_row_time_factor_to_row(), mat2D_add_row_to_row(), mat2D_alloc(), mat2D_calc_norma(), mat2D_col_is_all_digit(), mat2D_copy(), mat2D_copy_mat_to_mat_at_window(), mat2D_cross(), mat2D_det(), mat2D_det_2x2_mat(), mat2D_dot(), mat2D_dot_product(), mat2D_fill(), mat2D_fill_sequence(), mat2D_get_col(), mat2D_get_row(), mat2D_invert(), mat2D_LUP_decomposition_with_swap(), mat2D_make_identity(), mat2D_mat_is_all_digit(), mat2D_minor_alloc_fill_from_mat(), mat2D_mult(), mat2D_mult_row(), mat2D_offset2d(), mat2D_print(), mat2D_print_as_col(), mat2D_rand(), mat2D_row_is_all_digit(), mat2D_set_identity(), mat2D_set_rot_mat_x(), mat2D_set_rot_mat_y(), mat2D_set_rot_mat_z(), mat2D_solve_linear_sys_LUP_decomposition(), mat2D_sub(), mat2D_sub_col_to_col(), mat2D_sub_row_time_factor_to_row(), mat2D_sub_row_to_row(), mat2D_swap_rows(), mat2D_transpose(), mat2D_triangulate(), render(), and render_window().

#### 3.6.2.2 elements

```
double* Mat2D::elements
```

Definition at line 85 of file Matrix2D.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_curves_plot_on_figure(), adl_figure_alloc(), mat2D_alloc(), mat2D_free(), mat2D_print_as_col(), and render_window().

#### 3.6.2.3 rows

```
size_t Mat2D::rows
```

Definition at line 82 of file Matrix2D.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_curves_plot_on_figure(), adl_figure_alloc(), mat2D_add(), mat2D_add_col_to_col(), mat2D_add_row_to_row(), mat2D_alloc(), mat2D_calc_norma(), mat2D_copy(), mat2D_copy_mat_to_mat_at_window(), mat2D_cross(), mat2D_det(), mat2D_det_2x2_mat(), mat2D_dot(), mat2D_dot_product(), mat2D_fill(), mat2D_fill_sequence(), mat2D_get_col(), mat2D_get_row(), mat2D_invert(), mat2D_LUP_decomposition_with_swap(), mat2D_make_identity(), mat2D_mat_is_all_digit(), mat2D_minor_alloc_fill_from_mat(), mat2D_mult(), mat2D_offset2d(), mat2D_print(), mat2D_print_as_col(), mat2D_rand(), mat2D_set_identity(), mat2D_set_rot_mat_x(), mat2D_set_rot_mat_y(), mat2D_set_rot_mat_z(), mat2D_solve_linear_sys_LUP_decomposition(), mat2D_sub(), mat2D_sub_col_to_col(), mat2D_sub_row_to_row(), mat2D_transpose(), mat2D_triangulate(), render(), and render_window().

#### 3.6.2.4 stride_r

```
size_t Mat2D::stride_r
```

Definition at line 84 of file Matrix2D.h.

Referenced by mat2D_alloc(), and mat2D_offset2d().

The documentation for this struct was generated from the following file:

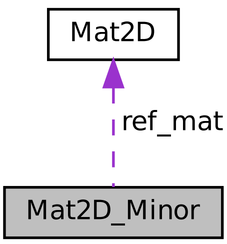
- Matrix2D.h

## 3.7 Mat2D_Minor Struct Reference

A minor "view" into a reference matrix.

```
#include <Matrix2D.h>
```

Collaboration diagram for Mat2D_Minor:



## Public Attributes

- size_t rows
- size_t cols
- size_t stride_r
- size_t * rows_list
- size_t * cols_list
- Mat2D ref_mat

### 3.7.1 Detailed Description

A minor "view" into a reference matrix.

Represents a minor by excluding one row and one column of a reference matrix. It holds index lists mapping into the reference matrix, without owning the data of the reference matrix itself.

Memory ownership:

- rows_list and cols_list are heap-allocated by minor allocators and must be freed with mat2D_minor_free.
- The underlying matrix data (ref_mat.elements) is not owned by the minor and must not be freed by the minor functions.

Definition at line 119 of file Matrix2D.h.

### 3.7.2 Member Data Documentation

#### 3.7.2.1 cols

```
size_t Mat2D_Minor::cols
```

Definition at line 121 of file Matrix2D.h.

Referenced by mat2D_det_2x2_mat_minor(), mat2D_minor_alloc_fill_from_mat(), mat2D_minor_alloc_fill_from_mat_minor(), mat2D_minor_det(), and mat2D_minor_print().

#### 3.7.2.2 cols_list

```
size_t* Mat2D_Minor::cols_list
```

Definition at line 124 of file Matrix2D.h.

Referenced by mat2D_minor_alloc_fill_from_mat(), mat2D_minor_alloc_fill_from_mat_minor(), and mat2D_minor_free().

#### 3.7.2.3 ref_mat

```
Mat2D Mat2D_Minor::ref_mat
```

Definition at line 125 of file Matrix2D.h.

Referenced by mat2D_minor_alloc_fill_from_mat(), and mat2D_minor_alloc_fill_from_mat_minor().

#### 3.7.2.4 rows

```
size_t Mat2D_Minor::rows
```

Definition at line 120 of file Matrix2D.h.

Referenced by mat2D_det_2x2_mat_minor(), mat2D_minor_alloc_fill_from_mat(), mat2D_minor_alloc_fill_from_mat_minor(), mat2D_minor_det(), and mat2D_minor_print().

#### 3.7.2.5 rows_list

```
size_t* Mat2D_Minor::rows_list
```

Definition at line 123 of file Matrix2D.h.

Referenced by mat2D_minor_alloc_fill_from_mat(), mat2D_minor_alloc_fill_from_mat_minor(), and mat2D_minor_free().

#### 3.7.2.6 stride_r

```
size_t Mat2D_Minor::stride_r
```

Definition at line 122 of file Matrix2D.h.

Referenced by mat2D_minor_alloc_fill_from_mat(), and mat2D_minor_alloc_fill_from_mat_minor().

The documentation for this struct was generated from the following file:

- Matrix2D.h

## 3.8 Mat2D_uint32 Struct Reference

Dense row-major matrix of uint32_t.

```
#include <Matrix2D.h>
```

## Public Attributes

- size_t rows
- size_t cols
- size_t stride_r
- uint32_t ∗ elements

### 3.8.1 Detailed Description

Dense row-major matrix of uint32_t.

- rows: number of rows (height)
- cols: number of columns (width)
- stride_r: number of elements between successive rows in memory (for contiguous storage, stride_r == cols)
- elements: pointer to contiguous storage of size rows ∗ cols

Definition at line 98 of file Matrix2D.h.

### 3.8.2 Member Data Documentation

#### 3.8.2.1 cols

```
size_t Mat2D_uint32::cols
```

Definition at line 100 of file Matrix2D.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_axis_draw_on_figure(), adl_figure_alloc(), adl_figure_copy_to_screen(), adl_line_draw(), adl_point_draw(), adl_quad_fill(), adl_quad_fill_interpolate_color_mean_value(), adl_quad_fill_interpolate_normal_m adl_tri_fill_Pinedas_rasterizer(), adl_tri_fill_Pinedas_rasterizer_interpolate_color(), adl_tri_fill_Pinedas_rasterizer_interpolate_normal check_window_mat_size(), copy_mat_to_surface_RGB(), mat2D_alloc_uint32(), mat2D_fill_uint32(), mat2D_offset2d_uint32(), and render_window().

#### 3.8.2.2 elements

```
uint32_t* Mat2D_uint32::elements
```

Definition at line 102 of file Matrix2D.h.

Referenced by copy_mat_to_surface_RGB(), mat2D_alloc_uint32(), mat2D_free_uint32(), and render_window().

#### 3.8.2.3 rows

```
size_t Mat2D_uint32::rows
```

Definition at line 99 of file Matrix2D.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_axis_draw_on_figure(), adl_figure_alloc(), adl_figure_copy_to_screen(), adl_line_draw(), adl_max_min_values_draw_on_figure(), adl_point_draw(), adl_quad_fill(), adl_quad_fill_interpolate_color_mean_val adl_quad_fill_interpolate_normal_mean_value(), adl_tri_fill_Pinedas_rasterizer(), adl_tri_fill_Pinedas_rasterizer_interpolate_color(), adl_tri_fill_Pinedas_rasterizer_interpolate_normal(), check_window_mat_size(), copy_mat_to_surface_RGB(), mat2D_alloc_uint32(), mat2D_fill_uint32(), mat2D_offset2d_uint32(), and render_window().

#### 3.8.2.4 stride_r

```
size_t Mat2D_uint32::stride_r
```

Definition at line 101 of file Matrix2D.h.

Referenced by mat2D_alloc_uint32(), and mat2D_offset2d_uint32().

The documentation for this struct was generated from the following file:

- Matrix2D.h

## 3.9 Offset_zoom_param Struct Reference

Pan/zoom parameters relative to screen center.

```
#include <Almog_Draw_Library.h>
```

### Public Attributes

- float zoom_multiplier
- float offset_x
- float offset_y
- int mouse_x
- int mouse_y

### 3.9.1 Detailed Description

Pan/zoom parameters relative to screen center.

The coordinates are shifted by (offset_x, offset_y) and scaled by zoom_multiplier about the screen center. The mouse fields are optional and can be used by UI code that updates the pan/zoom.

Definition at line 65 of file Almog_Draw_Library.h.

## 3.9.2 Member Data Documentation

### 3.9.2.1 mouse_x

```
int Offset_zoom_param::mouse_x
```

Optional: last mouse x (pixels).

Definition at line 69 of file Almog_Draw_Library.h.

### 3.9.2.2 mouse_y

```
int Offset_zoom_param::mouse_y
```

Optional: last mouse y (pixels).

Definition at line 70 of file Almog_Draw_Library.h.

### 3.9.2.3 offset_x

```
float Offset_zoom_param::offset_x
```

Horizontal pan offset (pixels).

Definition at line 67 of file Almog_Draw_Library.h.

Referenced by adl_line_draw(), adl_point_draw(), and process_input_window().

### 3.9.2.4 offset_y

```
float Offset_zoom_param::offset_y
```

Vertical pan offset (pixels).

Definition at line 68 of file Almog_Draw_Library.h.

Referenced by adl_line_draw(), adl_point_draw(), and process_input_window().

#### 3.9.2.5 zoom_multiplier

```
float Offset_zoom_param::zoom_multiplier
```

Zoom scale factor (>0).

Definition at line 66 of file Almog_Draw_Library.h.

Referenced by adl_line_draw(), adl_point_draw(), main(), and process_input_window().

The documentation for this struct was generated from the following file:

- Almog_Draw_Library.h

## 3.10 Point Struct Reference

Homogeneous 2D/3D point with per-vertex depth (z) and w.

```
#include <Almog_Draw_Library.h>
```

### Public Attributes

- float x
- float y
- float z
- float w

### 3.10.1 Detailed Description

Homogeneous 2D/3D point with per-vertex depth (z) and w.

x,y are screen-space coordinates for rasterization. z,w are used for perspective-correct interpolation via inverse-Z buffering.

Definition at line 81 of file Almog_Draw_Library.h.

### 3.10.2 Member Data Documentation

#### 3.10.2.1 w

```
float Point::w
```

Homogeneous w.

Definition at line 85 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_cartesian_grid_create(), adl_quad_fill(), adl_quad_fill_interpolate_color_mean_v adl_quad_fill_interpolate_normal_mean_value(), adl_tri_fill_Pinedas_rasterizer(), adl_tri_fill_Pinedas_rasterizer_interpolate_color(), and adl_tri_fill_Pinedas_rasterizer_interpolate_normal().

#### 3.10.2.2 x

```
float Point::x
```

X coordinate (pixels).

Definition at line 82 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_cartesian_grid_create(), adl_curve_add_to_figure(), adl_curves_plot_on_figure(), adl_figure_copy_to_screen(), adl_quad_fill(), adl_quad_fill_interpolate_color_mean_value(), adl_quad_fill_interpolate_normal_mean_value(), adl_tan_half_angle(), adl_tri_draw(), adl_tri_fill_Pinedas_rasterizer(), adl_tri_fill_Pinedas_rasterizer_interpolate_color(), and adl_tri_fill_Pinedas_rasterizer_interpolate_normal().

#### 3.10.2.3 y

```
float Point::y
```

Y coordinate (pixels).

Definition at line 83 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_cartesian_grid_create(), adl_curve_add_to_figure(), adl_curves_plot_on_figure(), adl_figure_copy_to_screen(), adl_quad_fill(), adl_quad_fill_interpolate_color_mean_value(), adl_quad_fill_interpolate_normal_mean_value(), adl_tan_half_angle(), adl_tri_draw(), adl_tri_fill_Pinedas_rasterizer(), adl_tri_fill_Pinedas_rasterizer_interpolate_color(), and adl_tri_fill_Pinedas_rasterizer_interpolate_normal().

#### 3.10.2.4 z

```
float Point::z
```

Depth value.

Definition at line 84 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_cartesian_grid_create(), adl_quad_fill(), adl_quad_fill_interpolate_color_mean_v adl_quad_fill_interpolate_normal_mean_value(), adl_tri_fill_Pinedas_rasterizer(), adl_tri_fill_Pinedas_rasterizer_interpolate_color(), and adl_tri_fill_Pinedas_rasterizer_interpolate_normal().

The documentation for this struct was generated from the following file:

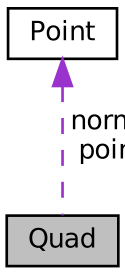
- Almog_Draw_Library.h

## 3.11 Quad Struct Reference

Quad primitive with optional per-vertex attributes.

`#include <Almog_Draw_Library.h>`

Collaboration diagram for Quad:




### Public Attributes

- Point points [4]
- Point normals [4]
- uint32_t colors [4]
- bool to_draw
- float light_intensity [4]

### 3.11.1 Detailed Description

Quad primitive with optional per-vertex attributes.

Definition at line 134 of file Almog_Draw_Library.h.

### 3.11.2 Member Data Documentation

#### 3.11.2.1 colors

`uint32_t Quad::colors[4]`

Optional per-vertex ARGB colors.

Definition at line 137 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_quad2tris(), adl_quad_fill_interpolate_color_mean_value(), and setup().

#### 3.11.2.2 light_intensity

```
float Quad::light_intensity[4]
```

Per-vertex light intensity multiplier.

Definition at line 139 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_quad2tris(), adl_quad_fill(), adl_quad_fill_interpolate_color_mean_value(), adl_quad_fill_interpolate_normal_mean_value(), and setup().

#### 3.11.2.3 normals

```
Point Quad::normals[4]
```

Optional normals (unused here).

Definition at line 136 of file Almog_Draw_Library.h.

#### 3.11.2.4 points

```
Point Quad::points[4]
```

Quad vertices (0..3 order).

Definition at line 135 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_quad2tris(), adl_quad_draw(), adl_quad_fill(), adl_quad_fill_interpolate_color_me
adl_quad_fill_interpolate_normal_mean_value(), and setup().

#### 3.11.2.5 to_draw

```
bool Quad::to_draw
```

Whether to include in rendering.

Definition at line 138 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_quad2tris(), adl_quad_mesh_draw(), adl_quad_mesh_fill(), adl_quad_mesh_fill_interpolate_color(), adl_quad_mesh_fill_interpolate_normal(), and setup().

The documentation for this struct was generated from the following file:

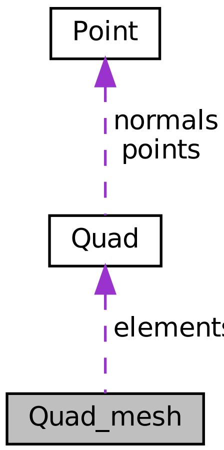
- Almog_Draw_Library.h

## 3.12 Quad_mesh Struct Reference

Dynamic array of quads (quad mesh).

`#include <Almog_Draw_Library.h>`

Collaboration diagram for Quad_mesh:




### Public Attributes

- size_t length
- size_t capacity
- Quad ∗ elements

### 3.12.1 Detailed Description

Dynamic array of quads (quad mesh).

Definition at line 160 of file Almog_Draw_Library.h.

### 3.12.2 Member Data Documentation

#### 3.12.2.1 capacity

`size_t Quad_mesh::capacity`

Allocated capacity.

Definition at line 162 of file Almog_Draw_Library.h.

#### 3.12.2.2 elements

```
Quad* Quad_mesh::elements
```

Quad array.

Definition at line 163 of file Almog_Draw_Library.h.

Referenced by adl_quad_mesh_draw(), adl_quad_mesh_fill(), adl_quad_mesh_fill_interpolate_color(), and adl_quad_mesh_fill_interpolate_normal().

#### 3.12.2.3 length

```
size_t Quad_mesh::length
```

Number of quads used.

Definition at line 161 of file Almog_Draw_Library.h.

Referenced by adl_quad_mesh_draw(), adl_quad_mesh_fill(), adl_quad_mesh_fill_interpolate_color(), and adl_quad_mesh_fill_interpolate_normal().

The documentation for this struct was generated from the following file:

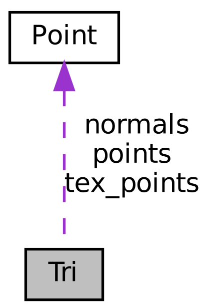
- Almog_Draw_Library.h

## 3.13 Tri Struct Reference

Triangle primitive with optional per-vertex attributes.

```
#include <Almog_Draw_Library.h>
```

Collaboration diagram for Tri:



## Public Attributes

- Point points [3]
- Point tex_points [3]
- Point normals [3]
- uint32_t colors [3]
- bool to_draw
- float light_intensity [3]

### 3.13.1 Detailed Description

Triangle primitive with optional per-vertex attributes.

Definition at line 119 of file Almog_Draw_Library.h.

### 3.13.2 Member Data Documentation

#### 3.13.2.1 colors

```
uint32_t Tri::colors[3]
```

Optional per-vertex ARGB colors.

Definition at line 123 of file Almog_Draw_Library.h.

Referenced by adl_quad2tris(), adl_tri_fill_Pinedas_rasterizer_interpolate_color(), and setup().

#### 3.13.2.2 light_intensity

```
float Tri::light_intensity[3]
```

Per-vertex light intensity multiplier.

Definition at line 125 of file Almog_Draw_Library.h.

Referenced by adl_quad2tris(), adl_tri_fill_Pinedas_rasterizer(), adl_tri_fill_Pinedas_rasterizer_interpolate_color(), adl_tri_fill_Pinedas_rasterizer_interpolate_normal(), and setup().

#### 3.13.2.3 normals

```
Point Tri::normals[3]
```

Optional normals (unused here).

Definition at line 122 of file Almog_Draw_Library.h.

#### 3.13.2.4 points

```
Point Tri::points[3]
```

Triangle vertices.

Definition at line 120 of file Almog_Draw_Library.h.

Referenced by adl_quad2tris(), adl_tri_draw(), adl_tri_fill_Pinedas_rasterizer(), adl_tri_fill_Pinedas_rasterizer_interpolate_color(), adl_tri_fill_Pinedas_rasterizer_interpolate_normal(), and setup().

#### 3.13.2.5 tex_points

```
Point Tri::tex_points[3]
```

Optional texture coordinates (unused here).

Definition at line 121 of file Almog_Draw_Library.h.

#### 3.13.2.6 to_draw

```
bool Tri::to_draw
```

Whether to include in rendering.

Definition at line 124 of file Almog_Draw_Library.h.

Referenced by adl_quad2tris(), adl_tri_mesh_draw(), adl_tri_mesh_fill_Pinedas_rasterizer(), adl_tri_mesh_fill_Pinedas_rasterizer_int adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_normal(), and setup().

The documentation for this struct was generated from the following file:

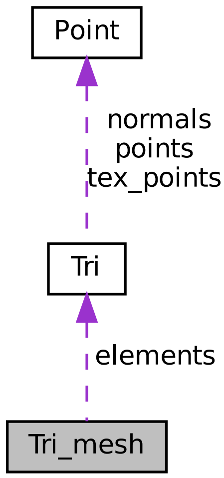
- Almog_Draw_Library.h

## 3.14 Tri_mesh Struct Reference

Dynamic array of triangles (triangle mesh).

`#include <Almog_Draw_Library.h>`

Collaboration diagram for Tri_mesh:




### Public Attributes

- size_t length
- size_t capacity
- Tri ∗ elements

### 3.14.1 Detailed Description

Dynamic array of triangles (triangle mesh).

Definition at line 148 of file Almog_Draw_Library.h.

### 3.14.2 Member Data Documentation

#### 3.14.2.1 capacity

```
size_t Tri_mesh::capacity
```

Allocated capacity.

Definition at line 150 of file Almog_Draw_Library.h.

#### 3.14.2.2 elements

```
Tri* Tri_mesh::elements
```

Triangle array.

Definition at line 151 of file Almog_Draw_Library.h.

Referenced by adl_tri_mesh_draw(), adl_tri_mesh_fill_Pinedas_rasterizer(), adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_color(), and adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_normal().

#### 3.14.2.3 length

```
size_t Tri_mesh::length
```

Number of triangles used.

Definition at line 149 of file Almog_Draw_Library.h.

Referenced by adl_tri_mesh_draw(), adl_tri_mesh_fill_Pinedas_rasterizer(), adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_color(), and adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_normal().

The documentation for this struct was generated from the following file:

- Almog_Draw_Library.h

# Chapter 4

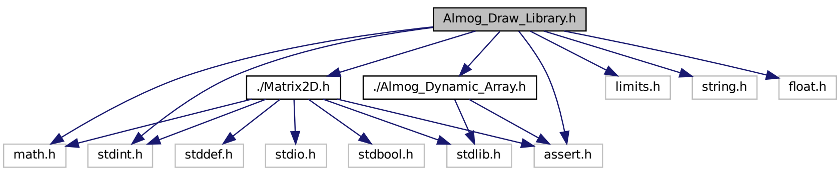
# File Documentation

## 4.1 Almog_Draw_Library.h File Reference

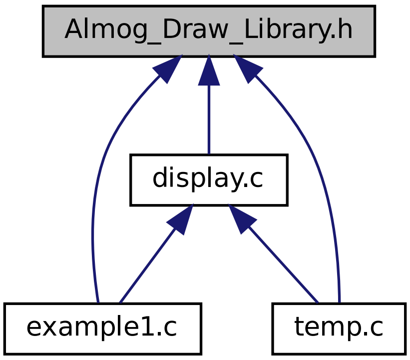
Immediate-mode 2D/3D raster helpers for drawing onto Mat2D_uint32 pixel buffers.

```
#include <math.h>
#include <stdint.h>
#include <limits.h>
#include <string.h>
#include <float.h>
#include "./Matrix2D.h"
#include "./Almog_Dynamic_Array.h"
#include <assert.h>
```

Include dependency graph for Almog_Draw_Library.h:



This graph shows which files directly or indirectly include this file:




## Classes

- struct Offset_zoom_param

  *Pan/zoom parameters relative to screen center.*
- struct Point

  *Homogeneous 2D/3D point with per-vertex depth (z) and w.*
- struct Curve

  *Polyline of points with a uniform color.*
- struct Curve_ada

  *Dynamic array of curves (polyline container).*
- struct Tri

  *Triangle primitive with optional per-vertex attributes.*
- struct Quad

  *Quad primitive with optional per-vertex attributes.*
- struct Tri_mesh

  *Dynamic array of triangles (triangle mesh).*
- struct Quad_mesh

  *Dynamic array of quads (quad mesh).*
- struct Figure

  *Plotting figure holding a pixel buffer, z-buffer and plot state.*
- struct Grid

  *Grid definition (as lines) in a chosen plane.*

## Macros

- #define ADL_ASSERT assert

  *Assertion macro used by this header (defaults to assert).*
- #define POINT
- #define CURVE
- #define CURVE_ADA

- #define TRI
- #define QUAD
- #define TRI_MESH
- #define QUAD_MESH
- #define HexARGB_RGBA(x) ((x)>>(8*2)&0xFF), ((x)>>(8*1)&0xFF), ((x)>>(8*0)&0xFF), ((x)>>(8*3)&0x←FF)
- #define HexARGB_RGB_VAR(x, r, g, b) r = ((x)>>(8*2)&0xFF); g = ((x)>>(8*1)&0xFF); b = ((x)>>(8*0)&0xFF);
- #define HexARGB_RGBA_VAR(x, r, g, b, a) r = ((x)>>(8*2)&0xFF); g = ((x)>>(8*1)&0xFF); b = ((x)>>(8*0)&0xFF); a = ((x)>>(8*3)&0xFF)
- #define RGB_hexRGB(r, g, b) (int)(0x010000*(int)(r) + 0x000100*(int)(g) + 0x000001*(int)(b))
- #define RGBA_hexARGB(r, g, b, a) (int)(0x01000000l*(int)(fminf(a, 255)) + 0x010000*(int)(r) + 0x000100*(int)(g) + 0x000001*(int)(b))
- #define RED_hexARGB 0xFFFF0000
- #define GREEN_hexARGB 0xFF00FF00
- #define BLUE_hexARGB 0xFF0000FF
- #define PURPLE_hexARGB 0xFFFF00FF
- #define CYAN_hexARGB 0xFF00FFFF
- #define YELLOW_hexARGB 0xFFFFFF00
- #define edge_cross_point(a1, b, a2, p) (b.x-a1.x)*(p.y-a2.y)-(b.y-a1.y)*(p.x-a2.x)
- #define is_top_edge(x, y) (y == 0 && x > 0)
- #define is_left_edge(x, y) (y < 0)
- #define is_top_left(ps, pe) (is_top_edge(pe.x-ps.x, pe.y-ps.y) || is_left_edge(pe.x-ps.x, pe.y-ps.y))
- #define ADL_MAX_POINT_VAL 1e5
- #define adl_assert_point_is_valid(p) ADL_ASSERT(isfinite(p.x) && isfinite(p.y) && isfinite(p.z) && isfinite(p.←w))
- #define adl_assert_tri_is_valid(tri)
- #define adl_assert_quad_is_valid(quad)
- #define ADL_FIGURE_PADDING_PRECENTAGE 20
- #define ADL_MAX_FIGURE_PADDING 70
- #define ADL_MIN_FIGURE_PADDING 20
- #define ADL_MAX_HEAD_SIZE 15
- #define ADL_FIGURE_HEAD_ANGLE_DEG 30
- #define ADL_FIGURE_AXIS_COLOR 0xff000000
- #define ADL_MAX_CHARACTER_OFFSET 10
- #define ADL_MIN_CHARACTER_OFFSET 5
- #define ADL_MAX_SENTENCE_LEN 256
- #define ADL_MAX_ZOOM 1e3
- #define ADL_DEFAULT_OFFSET_ZOOM (Offset_zoom_param){1,0,0,0,0}
- #define adl_offset_zoom_point(p, window_w, window_h, offset_zoom_param)
- #define adl_offset2d(i, j, ni) (j) * (ni) + (i)

## Functions

- void adl_point_draw (Mat2D_uint32 screen_mat, int x, int y, uint32_t color, Offset_zoom_param offset_←zoom_param)

  *Draw a single pixel with alpha blending.*
- void adl_line_draw (Mat2D_uint32 screen_mat, const float x1_input, const float y1_input, const float x2_input, const float y2_input, uint32_t color, Offset_zoom_param offset_zoom_param)

  *Draw an anti-aliased-like line by vertical spans (integer grid).*
- void adl_lines_draw (const Mat2D_uint32 screen_mat, const Point *points, const size_t len, const uint32_t color, Offset_zoom_param offset_zoom_param)

  *Draw a polyline connecting an array of points.*

- void adl_lines_loop_draw (const Mat2D_uint32 screen_mat, const Point *points, const size_t len, const uint32_t color, Offset_zoom_param offset_zoom_param)

  *Draw a closed polyline (loop).*
- void adl_arrow_draw (Mat2D_uint32 screen_mat, int xs, int ys, int xe, int ye, float head_size, float angle_deg, uint32_t color, Offset_zoom_param offset_zoom_param)

  *Draw an arrow from start to end with a triangular head.*
- void adl_character_draw (Mat2D_uint32 screen_mat, char c, int width_pixel, int hight_pixel, int x_top_left, int y_top_left, uint32_t color, Offset_zoom_param offset_zoom_param)

  *Draw a vector glyph for a single ASCII character.*
- void adl_sentence_draw (Mat2D_uint32 screen_mat, const char sentence[ ], size_t len, const int x_top_left, const int y_top_left, const int hight_pixel, const uint32_t color, Offset_zoom_param offset_zoom_param)

  *Draw a horizontal sentence using vector glyphs.*
- void adl_rectangle_draw_min_max (Mat2D_uint32 screen_mat, int min_x, int max_x, int min_y, int max_y, uint32_t color, Offset_zoom_param offset_zoom_param)

  *Draw a rectangle outline defined by min/max corners (inclusive).*
- void adl_rectangle_fill_min_max (Mat2D_uint32 screen_mat, int min_x, int max_x, int min_y, int max_↩ y, uint32_t color, Offset_zoom_param offset_zoom_param)

  *Fill a rectangle defined by min/max corners (inclusive).*
- void adl_quad_draw (Mat2D_uint32 screen_mat, Mat2D inv_z_buffer, Quad quad, uint32_t color, Offset_zoom_param offset_zoom_param)

  *Draw the outline of a quad (four points, looped).*
- void adl_quad_fill (Mat2D_uint32 screen_mat, Mat2D inv_z_buffer, Quad quad, uint32_t color, Offset_zoom_param offset_zoom_param)

  *Fill a quad using mean-value (Barycentric) coordinates and flat base color.*
- void adl_quad_fill_interpolate_normal_mean_value (Mat2D_uint32 screen_mat, Mat2D inv_z_buffer, Quad quad, uint32_t color, Offset_zoom_param offset_zoom_param)

  *Fill a quad with per-pixel light interpolation (mean value coords).*
- void adl_quad_fill_interpolate_color_mean_value (Mat2D_uint32 screen_mat, Mat2D inv_z_buffer, Quad quad, Offset_zoom_param offset_zoom_param)

  *Fill a quad with per-vertex colors (mean value coords).*
- void adl_quad_mesh_draw (Mat2D_uint32 screen_mat, Mat2D inv_z_buffer_mat, Quad_mesh mesh, uint32_t color, Offset_zoom_param offset_zoom_param)

  *Draw outlines for all quads in a mesh.*
- void adl_quad_mesh_fill (Mat2D_uint32 screen_mat, Mat2D inv_z_buffer_mat, Quad_mesh mesh, uint32_t color, Offset_zoom_param offset_zoom_param)

  *Fill all quads in a mesh with a uniform base color.*
- void adl_quad_mesh_fill_interpolate_normal (Mat2D_uint32 screen_mat, Mat2D inv_z_buffer_mat, Quad_mesh mesh, uint32_t color, Offset_zoom_param offset_zoom_param)

  *Fill all quads in a mesh using interpolated lighting.*
- void adl_quad_mesh_fill_interpolate_color (Mat2D_uint32 screen_mat, Mat2D inv_z_buffer_mat, Quad_mesh mesh, Offset_zoom_param offset_zoom_param)

  *Fill all quads in a mesh using per-vertex colors.*
- void adl_circle_draw (Mat2D_uint32 screen_mat, float center_x, float center_y, float r, uint32_t color, Offset_zoom_param offset_zoom_param)

  *Draw an approximate circle outline (1px thickness).*
- void adl_circle_fill (Mat2D_uint32 screen_mat, float center_x, float center_y, float r, uint32_t color, Offset_zoom_param offset_zoom_param)

  *Fill a circle.*
- void adl_tri_draw (Mat2D_uint32 screen_mat, Tri tri, uint32_t color, Offset_zoom_param offset_zoom_param)

  *Draw the outline of a triangle.*
- void adl_tri_fill_Pinedas_rasterizer (Mat2D_uint32 screen_mat, Mat2D inv_z_buffer, Tri tri, uint32_t color, Offset_zoom_param offset_zoom_param)

  *Fill a triangle using Pineda's rasterizer with flat base color.*

- void adl_tri_fill_Pinedas_rasterizer_interpolate_color (Mat2D_uint32 screen_mat, Mat2D inv_z_buffer, Tri tri, Offset_zoom_param offset_zoom_param)

  *Fill a triangle using Pineda's rasterizer with per-vertex colors.*
- void adl_tri_fill_Pinedas_rasterizer_interpolate_normal (Mat2D_uint32 screen_mat, Mat2D inv_z_buffer, Tri tri, uint32_t color, Offset_zoom_param offset_zoom_param)

  *Fill a triangle with interpolated lighting over a uniform color.*
- void adl_tri_mesh_draw (Mat2D_uint32 screen_mat, Tri_mesh mesh, uint32_t color, Offset_zoom_param offset_zoom_param)

  *Draw outlines for all triangles in a mesh.*
- void adl_tri_mesh_fill_Pinedas_rasterizer (Mat2D_uint32 screen_mat, Mat2D inv_z_buffer_mat, Tri_mesh mesh, uint32_t color, Offset_zoom_param offset_zoom_param)

  *Fill all triangles in a mesh with a uniform base color.*
- void adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_color (Mat2D_uint32 screen_mat, Mat2D inv_z_↩ buffer_mat, Tri_mesh mesh, Offset_zoom_param offset_zoom_param)

  *Fill all triangles in a mesh with a uniform base color.*
- void adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_normal (Mat2D_uint32 screen_mat, Mat2D inv_z_↩ buffer_mat, Tri_mesh mesh, uint32_t color, Offset_zoom_param offset_zoom_param)

  *Fill all triangles in a mesh with interpolated lighting.*
- float adl_tan_half_angle (Point vi, Point vj, Point p, float li, float lj)

  *Compute tan(alpha/2) for the angle at point p between segments p->vi and p->vj.*
- float adl_linear_map (float s, float min_in, float max_in, float min_out, float max_out)

  *Affine map from one scalar range to another (no clamping).*
- void adl_quad2tris (Quad quad, Tri ∗tri1, Tri ∗tri2, char split_line[ ])

  *Split a quad into two triangles along a chosen diagonal.*
- void adl_linear_sRGB_to_okLab (uint32_t hex_ARGB, float ∗L, float ∗a, float ∗b)

  *Convert a linear sRGB color (ARGB) to Oklab components.*
- void adl_okLab_to_linear_sRGB (float L, float a, float b, uint32_t ∗hex_ARGB)

  *Convert Oklab components to a linear sRGB ARGB color.*
- void adl_linear_sRGB_to_okLch (uint32_t hex_ARGB, float ∗L, float ∗c, float ∗h_deg)

  *Convert a linear sRGB color (ARGB) to OkLch components.*
- void adl_okLch_to_linear_sRGB (float L, float c, float h_deg, uint32_t ∗hex_ARGB)

  *Convert OkLch components to a linear sRGB ARGB color.*
- void adl_interpolate_ARGBcolor_on_okLch (uint32_t color1, uint32_t color2, float t, float num_of_rotations, uint32_t ∗color_out)

  *Interpolate between two ARGB colors in OkLch space.*
- Figure adl_figure_alloc (size_t rows, size_t cols, Point top_left_position)

  *Allocate and initialize a Figure with an internal pixel buffer.*
- void adl_figure_copy_to_screen (Mat2D_uint32 screen_mat, Figure figure)

  *Blit a Figure's pixels onto a destination screen buffer.*
- void adl_axis_draw_on_figure (Figure ∗figure)

  *Draw X/Y axes with arrowheads into a Figure.*
- void adl_max_min_values_draw_on_figure (Figure figure)

  *Draw min/max numeric labels for the current data range.*
- void adl_curve_add_to_figure (Figure ∗figure, Point ∗src_points, size_t src_len, uint32_t color)

  *Add a curve (polyline) to a Figure and update its data bounds.*
- void adl_curves_plot_on_figure (Figure figure)

  *Render all added curves into a Figure's pixel buffer.*
- void adl_2Dscalar_interp_on_figure (Figure figure, double ∗x_2Dmat, double ∗y_2Dmat, double ∗scalar_2↩ Dmat, int ni, int nj, char color_scale[ ], float num_of_rotations)

  *Visualize a scalar field on a Figure by colored quads.*
- Grid adl_cartesian_grid_create (float min_e1, float max_e1, float min_e2, float max_e2, int num_samples_e1, int num_samples_e2, char plane[ ], float third_direction_position)

*Create a Cartesian grid (as curves) on one of the principal planes.*

- void adl_grid_draw (Mat2D_uint32 screen_mat, Grid grid, uint32_t color, Offset_zoom_param offset_zoom↩_param)

    *Draw a previously created Grid as line segments.*

### 4.1.1 Detailed Description

Immediate-mode 2D/3D raster helpers for drawing onto Mat2D_uint32 pixel buffers.

This single-header library provides a minimal software rasterizer for drawing into a 32-bit ARGB pixel buffer (Mat2D_uint32). It supports:

- Points, lines, circles, triangles and quads (wire and filled)
- Z-buffered triangle/quad rasterization (inverse-Z convention)
- Per-vertex color and simple light-intensity interpolation
- Basic vector-text drawing (ASCII subset)
- Plotting helper types (Figure) and utilities for curve plots and 2D scalar-field visualization using perceptual color interpolation in the OKLab/OKLch color spaces
- Cartesian grid generation in common planes

All draw calls may accept an Offset_zoom_param that enables simple pan/zoom behavior around the screen center.

Types Mat2D and Mat2D_uint32 are provided by Matrix2D.h.

Usage:

- Include this header wherever you use the API.
- In exactly one translation unit (source file) define ALMOG_DRAW_LIBRARY_IMPLEMENTATION before including this header to compile the function definitions.

**Note**

- Colors are ARGB in 0xAARRGGBB packed 32-bit format.
- Z buffering uses an inverse-Z buffer (bigger is closer).
- The OKLab/OKLch conversions here assume linear sRGB channels.

Definition in file Almog_Draw_Library.h.

### 4.1.2 Macro Definition Documentation

#### 4.1.2.1 ADL_ASSERT

```
#define ADL_ASSERT assert
```

Assertion macro used by this header (defaults to assert).

Define ADL_ASSERT before including this file to override. When NDEBUG is defined, standard assert() is disabled.

Definition at line 55 of file Almog_Draw_Library.h.

#### 4.1.2.2 adl_assert_point_is_valid

```
#define adl_assert_point_is_valid(
              p ) ADL_ASSERT(isfinite(p.x) && isfinite(p.y) && isfinite(p.z) && isfinite(p.w))
```

Definition at line 310 of file Almog_Draw_Library.h.

#### 4.1.2.3 adl_assert_quad_is_valid

```
#define adl_assert_quad_is_valid(
              quad )
```

**Value:**

```
adl_assert_point_is_valid(quad.points[0]);   \
adl_assert_point_is_valid(quad.points[1]);                              \
adl_assert_point_is_valid(quad.points[2]);                              \
adl_assert_point_is_valid(quad.points[3])
```

Definition at line 314 of file Almog_Draw_Library.h.

#### 4.1.2.4 adl_assert_tri_is_valid

```
#define adl_assert_tri_is_valid(
              tri )
```

**Value:**

```
adl_assert_point_is_valid(tri.points[0]); \
adl_assert_point_is_valid(tri.points[1]);                              \
adl_assert_point_is_valid(tri.points[2])
```

Definition at line 311 of file Almog_Draw_Library.h.

#### 4.1.2.5 ADL_DEFAULT_OFFSET_ZOOM

```
#define ADL_DEFAULT_OFFSET_ZOOM (Offset_zoom_param){1,0,0,0,0}
```

Definition at line 331 of file Almog_Draw_Library.h.

#### 4.1.2.6 ADL_FIGURE_AXIS_COLOR

```
#define ADL_FIGURE_AXIS_COLOR 0xff000000
```

Definition at line 324 of file Almog_Draw_Library.h.

#### 4.1.2.7 ADL_FIGURE_HEAD_ANGLE_DEG

```
#define ADL_FIGURE_HEAD_ANGLE_DEG 30
```

Definition at line 323 of file Almog_Draw_Library.h.

#### 4.1.2.8 ADL_FIGURE_PADDING_PRECENTAGE

```
#define ADL_FIGURE_PADDING_PRECENTAGE 20
```

Definition at line 319 of file Almog_Draw_Library.h.

#### 4.1.2.9 ADL_MAX_CHARACTER_OFFSET

```
#define ADL_MAX_CHARACTER_OFFSET 10
```

Definition at line 326 of file Almog_Draw_Library.h.

#### 4.1.2.10 ADL_MAX_FIGURE_PADDING

```
#define ADL_MAX_FIGURE_PADDING 70
```

Definition at line 320 of file Almog_Draw_Library.h.

#### 4.1.2.11 ADL_MAX_HEAD_SIZE

```
#define ADL_MAX_HEAD_SIZE 15
```

Definition at line 322 of file Almog_Draw_Library.h.

#### 4.1.2.12 ADL_MAX_POINT_VAL

```
#define ADL_MAX_POINT_VAL 1e5
```

Definition at line 309 of file Almog_Draw_Library.h.

#### 4.1.2.13 ADL_MAX_SENTENCE_LEN

```
#define ADL_MAX_SENTENCE_LEN 256
```

Definition at line 328 of file Almog_Draw_Library.h.

#### 4.1.2.14 ADL_MAX_ZOOM

```
#define ADL_MAX_ZOOM 1e3
```

Definition at line 329 of file Almog_Draw_Library.h.

#### 4.1.2.15 ADL_MIN_CHARACTER_OFFSET

```
#define ADL_MIN_CHARACTER_OFFSET 5
```

Definition at line 327 of file Almog_Draw_Library.h.

#### 4.1.2.16 ADL_MIN_FIGURE_PADDING

```
#define ADL_MIN_FIGURE_PADDING 20
```

Definition at line 321 of file Almog_Draw_Library.h.

#### 4.1.2.17 adl_offset2d

```
#define adl_offset2d(
              i,
              j,
              ni ) (j) * (ni) + (i)
```

Definition at line 2294 of file Almog_Draw_Library.h.

#### 4.1.2.18 adl_offset_zoom_point

```
#define adl_offset_zoom_point(
              p,
              window_w,
              window_h,
              offset_zoom_param )
```

**Value:**

```
(p).x = ((p).x - (window_w)/2 + offset_zoom_param.offset_x) * offset_zoom_param.zoom_multiplier +
   (window_w)/2; \
(p).y = ((p).y - (window_h)/2 + offset_zoom_param.offset_y) * offset_zoom_param.zoom_multiplier +
   (window_h)/2
```

Definition at line 332 of file Almog_Draw_Library.h.

#### 4.1.2.19 BLUE_hexARGB

```
#define BLUE_hexARGB 0xFF0000FF
```

Definition at line 299 of file Almog_Draw_Library.h.

#### 4.1.2.20 CURVE

```
#define CURVE
```

Definition at line 90 of file Almog_Draw_Library.h.

#### 4.1.2.21 CURVE_ADA

```
#define CURVE_ADA
```

Definition at line 103 of file Almog_Draw_Library.h.

#### 4.1.2.22 CYAN_hexARGB

```
#define CYAN_hexARGB 0xFF00FFFF
```

Definition at line 301 of file Almog_Draw_Library.h.

#### 4.1.2.23 edge_cross_point

```
#define edge_cross_point(
             a1,
             b,
             a2,
             p ) (b.x-a1.x)*(p.y-a2.y)-(b.y-a1.y)*(p.x-a2.x)
```

Definition at line 304 of file Almog_Draw_Library.h.

#### 4.1.2.24 GREEN_hexARGB

```
#define GREEN_hexARGB 0xFF00FF00
```

Definition at line 298 of file Almog_Draw_Library.h.

#### 4.1.2.25 HexARGB_RGB_VAR

```
#define HexARGB_RGB_VAR(
             x,
             r,
             g,
             b ) r = ((x)>>(8*2)&0xFF); g = ((x)>>(8*1)&0xFF); b = ((x)>>(8*0)&0xFF);
```

Definition at line 224 of file Almog_Draw_Library.h.

#### 4.1.2.26 HexARGB_RGBA

```
#define HexARGB_RGBA(
             x ) ((x)>>(8*2)&0xFF), ((x)>>(8*1)&0xFF), ((x)>>(8*0)&0xFF), ((x)>>(8*3)&0xFF)
```

Definition at line 221 of file Almog_Draw_Library.h.

#### 4.1.2.27 HexARGB_RGBA_VAR

```
#define HexARGB_RGBA_VAR(
            x,
            r,
            g,
            b,
            a ) r = ((x)>>(8*2)&0xFF); g = ((x)>>(8*1)&0xFF); b = ((x)>>(8*0)&0xFF); a =
((x)>>(8*3)&0xFF)
```

Definition at line 227 of file Almog_Draw_Library.h.

#### 4.1.2.28 is_left_edge

```
#define is_left_edge(
            x,
            y ) (y < 0)
```

Definition at line 306 of file Almog_Draw_Library.h.

#### 4.1.2.29 is_top_edge

```
#define is_top_edge(
            x,
            y ) (y == 0 && x > 0)
```

Definition at line 305 of file Almog_Draw_Library.h.

#### 4.1.2.30 is_top_left

```
#define is_top_left(
            ps,
            pe ) (is_top_edge(pe.x-ps.x, pe.y-ps.y) || is_left_edge(pe.x-ps.x, pe.y-ps.y))
```

Definition at line 307 of file Almog_Draw_Library.h.

#### 4.1.2.31 POINT

```
#define POINT
```

Definition at line 74 of file Almog_Draw_Library.h.

#### 4.1.2.32 PURPLE_hexARGB

```
#define PURPLE_hexARGB 0xFFFF00FF
```

Definition at line 300 of file Almog_Draw_Library.h.

#### 4.1.2.33 QUAD

```
#define QUAD
```

Definition at line 130 of file Almog_Draw_Library.h.

#### 4.1.2.34 QUAD_MESH

```
#define QUAD_MESH
```

Definition at line 156 of file Almog_Draw_Library.h.

#### 4.1.2.35 RED_hexARGB

```
#define RED_hexARGB 0xFFFF0000
```

Definition at line 297 of file Almog_Draw_Library.h.

#### 4.1.2.36 RGB_hexRGB

```
#define RGB_hexRGB(
              r,
              g,
              b ) (int)(0x010000*(int)(r) + 0x000100*(int)(g) + 0x000001*(int)(b))
```

Definition at line 230 of file Almog_Draw_Library.h.

#### 4.1.2.37 RGBA_hexARGB

```
#define RGBA_hexARGB(
              r,
              g,
              b,
              a ) (int)(0x01000000l*(int)(fminf(a, 255)) + 0x010000*(int)(r) + 0x000100*(int)(g)
+ 0x000001*(int)(b))
```

Definition at line 233 of file Almog_Draw_Library.h.

#### 4.1.2.38 TRI

```
#define TRI
```

Definition at line 115 of file Almog_Draw_Library.h.

#### 4.1.2.39 TRI_MESH

```
#define TRI_MESH
```

Definition at line 144 of file Almog_Draw_Library.h.

#### 4.1.2.40 YELLOW_hexARGB

```
#define YELLOW_hexARGB 0xFFFFFF00
```

Definition at line 302 of file Almog_Draw_Library.h.

### 4.1.3 Function Documentation

#### 4.1.3.1 adl_2Dscalar_interp_on_figure()

```
void adl_2Dscalar_interp_on_figure (
            Figure figure,
            double * x_2Dmat,
            double * y_2Dmat,
            double * scalar_2Dmat,
            int ni,
            int nj,
            char color_scale[],
            float num_of_rotations )
```

Visualize a scalar field on a Figure by colored quads.

Treats x_2Dmat and y_2Dmat as a structured 2D grid of positions (column-major with stride ni) and colors each cell using scalar_2Dmat mapped through a two-color OkLch gradient. Also updates figure bounds from the provided data. Depth-tested inside the figure's buffers.

**Parameters**

| | |
|---|---|
| *figure* | Figure to render into (uses its own pixel buffers). |
| *x_2Dmat* | Grid X coordinates, size ni∗nj. |
| *y_2Dmat* | Grid Y coordinates, size ni∗nj. |
| *scalar_2Dmat* | Scalar values per grid node, size ni∗nj. |
| *ni* | Number of samples along the first index (rows). |
| *nj* | Number of samples along the second index (cols). |
| *color_scale* | Two-letter code of endpoints ("b-c","b-g","b-r", "b-y","g-y","g-p","g-r","r-y"). |
| *num_of_rotations* | Hue turns for the OkLch interpolation (can be fractional/negative). |

Definition at line 2314 of file Almog_Draw_Library.h.

References adl_axis_draw_on_figure(), ADL_DEFAULT_OFFSET_ZOOM, adl_interpolate_ARGBcolor_on_okLch(), adl_linear_map(), adl_max_min_values_draw_on_figure(), adl_offset2d, adl_offset_zoom_point, adl_quad_fill_interpolate_color_mea Figure::background_color, BLUE_hexARGB, Quad::colors, Mat2D::cols, Mat2D_uint32::cols, CYAN_hexARGB, Mat2D::elements, GREEN_hexARGB, Figure::inv_z_buffer_mat, Quad::light_intensity, mat2D_fill_uint32(), Figure::max_x, Figure::max_x_pixel, Figure::max_y, Figure::max_y_pixel, Figure::min_x, Figure::min_x_pixel, Figure::min_y, Figure::min_y_pixel, Figure::offset_zoom_param, Figure::pixels_mat, Quad::points, PURPLE_hexARGB, RED_hexARGB, Mat2D::rows, Mat2D_uint32::rows, Quad::to_draw, Figure::to_draw_axis, Figure::to_draw_max_min_values, Point::w, Point::x, Point::y, YELLOW_hexARGB, and Point::z.

#### 4.1.3.2 adl_arrow_draw()

```
void adl_arrow_draw (
            Mat2D_uint32 screen_mat,
            int xs,
            int ys,
            int xe,
            int ye,
            float head_size,
            float angle_deg,
            uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Draw an arrow from start to end with a triangular head.

The head is constructed by rotating around the arrow tip by +/- angle_deg and using head_size as a fraction of the shaft length.

**Note**

: This function is a bit complicated and expansive but this is what I could come up with

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *xs* | Start X (before pan/zoom). |
| *ys* | Start Y (before pan/zoom). |
| *xe* | End X (before pan/zoom), i.e., the arrow tip. |
| *ye* | End Y (before pan/zoom), i.e., the arrow tip. |
| *head_size* | Head size as a fraction of total length in [0,1]. |
| *angle_deg* | Head wing rotation angle in degrees. |
| *color* | Arrow color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 518 of file Almog_Draw_Library.h.

References adl_line_draw(), mat2D_add(), mat2D_alloc(), MAT2D_AT, mat2D_copy(), mat2D_dot(), mat2D_fill(), mat2D_free(), mat2D_set_rot_mat_z(), and mat2D_sub().

Referenced by adl_axis_draw_on_figure().

#### 4.1.3.3 adl_axis_draw_on_figure()

```
void adl_axis_draw_on_figure (
            Figure * figure )
```

Draw X/Y axes with arrowheads into a Figure.

Uses the current figure's pixel extents and padding to place axes, and stores the computed head sizes for later label layout.

**Parameters**

| | |
|---|---|
| *figure* | [in,out] Figure to draw onto. |

Definition at line 2144 of file Almog_Draw_Library.h.

References adl_arrow_draw(), ADL_FIGURE_AXIS_COLOR, ADL_FIGURE_HEAD_ANGLE_DEG, ADL_FIGURE_PADDING_PREC ADL_MAX_FIGURE_PADDING, ADL_MAX_HEAD_SIZE, ADL_MIN_FIGURE_PADDING, Mat2D_uint32::cols, Figure::max_x_pixel, Figure::max_y_pixel, Figure::min_x_pixel, Figure::min_y_pixel, Figure::offset_zoom_param, Figure::pixels_mat, Mat2D_uint32::rows, Figure::x_axis_head_size, and Figure::y_axis_head_size.

Referenced by adl_2Dscalar_interp_on_figure(), and adl_curves_plot_on_figure().

#### 4.1.3.4 adl_cartesian_grid_create()

```
Grid adl_cartesian_grid_create (
            float min_e1,
            float max_e1,
            float min_e2,
            float max_e2,
            int num_samples_e1,
            int num_samples_e2,
            char plane[ ],
            float third_direction_position )
```

Create a Cartesian grid (as curves) on one of the principal planes.

Supported planes (case-insensitive): "XY","xy","XZ","xz","YX","yx","YZ","yz","ZX","zx","ZY","zy". The third_↩
direction_position places the grid along the axis normal to the plane (e.g., Z for "XY").

**Parameters**

| | |
|---|---|
| *min_e1* | Minimum coordinate along the first axis of the plane. |
| *max_e1* | Maximum coordinate along the first axis of the plane. |
| *min_e2* | Minimum coordinate along the second axis of the plane. |
| *max_e2* | Maximum coordinate along the second axis of the plane. |
| *num_samples_e1* | Number of segments along first axis. |
| *num_samples_e2* | Number of segments along second axis. |
| *plane* | Plane code string ("XY","xy","XZ","xz","YX","yx","YZ","yz","ZX","zx","ZY","zy"). |
| *third_direction_position* | Position along the axis normal to plane. |

**Returns**

Grid structure containing the generated curves and spacing.

Definition at line 2513 of file Almog_Draw_Library.h.

References ada_append, ada_init_array, Grid::curves, Grid::de1, Grid::de2, Grid::max_e1, Grid::max_e2, Grid::min_e1, Grid::min_e2, Grid::num_samples_e1, Grid::num_samples_e2, Grid::plane, Point::w, Point::x, Point::y, and Point::z.

### 4.1.3.5 adl_character_draw()

```
void adl_character_draw (
            Mat2D_uint32 screen_mat,
            char c,
            int width_pixel,
            int hight_pixel,
            int x_top_left,
            int y_top_left,
            uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Draw a vector glyph for a single ASCII character.

Only a limited set of characters is supported (A–Z, a–z, 0–9, space, '.', ':', '-', '+'). Unsupported characters are rendered as a framed box with an 'X'. Coordinates are for the character's top-left corner.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *c* | The character to draw. |
| *width_pixel* | Character box width in pixels. |
| *hight_pixel* | Character box height in pixels (spelled as in API). |
| *x_top_left* | X of top-left corner (before pan/zoom). |
| *y_top_left* | Y of top-left corner (before pan/zoom). |
| *color* | Stroke color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 586 of file Almog_Draw_Library.h.

References adl_line_draw(), adl_rectangle_draw_min_max(), and adl_rectangle_fill_min_max().

Referenced by adl_sentence_draw(), and render().

### 4.1.3.6 adl_circle_draw()

```
void adl_circle_draw (
            Mat2D_uint32 screen_mat,
```

```
            float center_x,
            float center_y,
            float r,
            uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Draw an approximate circle outline (1px thickness).

The outline is approximated on the integer grid by sampling a band around radius r.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *center_x* | Circle center X (before pan/zoom). |
| *center_y* | Circle center Y (before pan/zoom). |
| *r* | Circle radius in pixels. |
| *color* | Stroke color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 1427 of file Almog_Draw_Library.h.

References adl_point_draw().

#### 4.1.3.7 adl_circle_fill()

```
void adl_circle_fill (
            Mat2D_uint32 screen_mat,
            float center_x,
            float center_y,
            float r,
            uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Fill a circle.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *center_x* | Circle center X (before pan/zoom). |
| *center_y* | Circle center Y (before pan/zoom). |
| *r* | Circle radius in pixels. |
| *color* | Fill color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 1449 of file Almog_Draw_Library.h.

References adl_point_draw().

#### 4.1.3.8 adl_curve_add_to_figure()

```
void adl_curve_add_to_figure (
            Figure * figure,
            Point * src_points,
            size_t src_len,
            uint32_t color )
```

Add a curve (polyline) to a Figure and update its data bounds.

The input points are copied into the figure's source curve array with the given color. Figure min/max bounds are updated to include them.

**Parameters**

| | |
|---|---|
| *figure* | [in,out] Target figure. |
| *src_points* | Array of source points (in data space). |
| *src_len* | Number of points. |
| *color* | Curve color (0xAARRGGBB). |

Definition at line 2230 of file Almog_Draw_Library.h.

References ada_append, ada_init_array, Curve::color, Figure::max_x, Figure::max_y, Figure::min_x, Figure::min_y, Figure::src_curve_array, Point::x, and Point::y.

Referenced by setup().

#### 4.1.3.9 adl_curves_plot_on_figure()

```
void adl_curves_plot_on_figure (
            Figure figure )
```

Render all added curves into a Figure's pixel buffer.

Clears the pixel buffer to background_color, draws axes if enabled, maps data-space points to pixel-space using current min/max bounds, draws the polylines, and optionally draws min/max labels.

**Parameters**

| | |
|---|---|
| *figure* | Figure to render into (uses its own pixel buffer). |

Definition at line 2265 of file Almog_Draw_Library.h.

References adl_axis_draw_on_figure(), adl_line_draw(), adl_linear_map(), adl_max_min_values_draw_on_figure(), Figure::background_color, Curve::color, Mat2D::cols, Curve::elements, Curve_ada::elements, Mat2D::elements, Figure::inv_z_buffer_mat, Curve::length, Curve_ada::length, mat2D_fill_uint32(), Figure::max_x, Figure::max_x_pixel, Figure::max_y, Figure::max_y_pixel, Figure::min_x, Figure::min_x_pixel, Figure::min_y, Figure::min_y_pixel, Figure::offset_zoom_param, Figure::pixels_mat, Mat2D::rows, Figure::src_curve_array, Figure::to_draw_axis, Figure::to_draw_max_min_values, Point::x, and Point::y.

Referenced by render().

#### 4.1.3.10 adl_figure_alloc()

```
Figure adl_figure_alloc (
            size_t rows,
            size_t cols,
            Point top_left_position )
```

Allocate and initialize a Figure with an internal pixel buffer.

Initializes the pixel buffer (rows x cols), an inverse-Z buffer (zeroed), an empty source curve array, and default padding/axes bounds. The background_color, to_draw_axis, and to_draw_max_min_values should be set by the caller before rendering.

**Parameters**

| | |
|---|---|
| *rows* | Height of the figure in pixels. |
| *cols* | Width of the figure in pixels. |
| *top_left_position* | Target position when copying to a screen. |

**Returns**

A new Figure with allocated buffers.

Definition at line 2081 of file Almog_Draw_Library.h.

References ada_init_array, ADL_ASSERT, adl_assert_point_is_valid, ADL_DEFAULT_OFFSET_ZOOM, ADL_FIGURE_PADDING_PRECENTAGE, ADL_MAX_FIGURE_PADDING, Mat2D::cols, Mat2D_uint32::cols, Mat2D::elements, Figure::inv_z_buffer_mat, mat2D_alloc(), mat2D_alloc_uint32(), Figure::max_x, Figure::max_x_pixel, Figure::max_y, Figure::max_y_pixel, Figure::min_x, Figure::min_x_pixel, Figure::min_y, Figure::min_y_pixel, Figure::offset_zoom_param, Figure::pixels_mat, Mat2D::rows, Mat2D_uint32::rows, Figure::src_curve_array, and Figure::top_left_position.

Referenced by setup().

#### 4.1.3.11 adl_figure_copy_to_screen()

```
void adl_figure_copy_to_screen (
            Mat2D_uint32 screen_mat,
            Figure figure )
```

Blit a Figure's pixels onto a destination screen buffer.

Performs per-pixel blending using adl_point_draw and the identity transform. The figure's top_left_position is used as the destination offset.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *figure* | Source figure to copy from. |

Definition at line 2124 of file Almog_Draw_Library.h.

References adl_point_draw(), Mat2D_uint32::cols, MAT2D_AT_UINT32, Figure::pixels_mat, Mat2D_uint32::rows, Figure::top_left_position, Point::x, and Point::y.

Referenced by render().

#### 4.1.3.12 adl_grid_draw()

```
void adl_grid_draw (
            Mat2D_uint32 screen_mat,
            Grid grid,
            uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Draw a previously created Grid as line segments.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *grid* | Grid to draw (curves are 2-point polylines). |
| *color* | Line color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 2791 of file Almog_Draw_Library.h.

References adl_lines_draw(), Grid::curves, Curve::elements, Curve_ada::elements, Curve::length, and Curve_ada::length.

#### 4.1.3.13 adl_interpolate_ARGBcolor_on_okLch()

```
void adl_interpolate_ARGBcolor_on_okLch (
            uint32_t color1,
            uint32_t color2,
            float t,
            float num_of_rotations,
            uint32_t * color_out )
```

Interpolate between two ARGB colors in OkLch space.

Lightness and chroma are interpolated linearly. Hue is interpolated in degrees after adding 360∗num_of_rotations to the second hue, allowing control over the winding direction.

**Parameters**

| | |
|---|---|
| *color1* | Start color (0xAARRGGBB). |
| *color2* | End color (0xAARRGGBB). |
| *t* | Interpolation factor in [0,1]. |
| *num_of_rotations* | Number of hue turns to add to color2 (can be fractional/negative). |
| *color_out* | [out] Interpolated ARGB color (A=255). |

Definition at line 2053 of file Almog_Draw_Library.h.

References adl_linear_sRGB_to_okLch(), and adl_okLch_to_linear_sRGB().

Referenced by adl_2Dscalar_interp_on_figure().

#### 4.1.3.14 adl_line_draw()

```
void adl_line_draw (
            Mat2D_uint32 screen_mat,
            const float x1_input,
            const float y1_input,
            const float x2_input,
            const float y2_input,
            uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Draw an anti-aliased-like line by vertical spans (integer grid).

The line is rasterized with a simple integer-span approach. Pan/zoom is applied about the screen center prior to rasterization.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *x1_input* | Line start X (before pan/zoom). |
| *y1_input* | Line start Y (before pan/zoom). |
| *x2_input* | Line end X (before pan/zoom). |
| *y2_input* | Line end Y (before pan/zoom). |
| *color* | Line color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 383 of file Almog_Draw_Library.h.

References ADL_ASSERT, ADL_MAX_POINT_VAL, adl_point_draw(), Mat2D_uint32::cols, Offset_zoom_param::offset_x, Offset_zoom_param::offset_y, Mat2D_uint32::rows, and Offset_zoom_param::zoom_multiplier.

Referenced by adl_arrow_draw(), adl_character_draw(), adl_curves_plot_on_figure(), adl_lines_draw(), adl_lines_loop_draw(), adl_rectangle_draw_min_max(), adl_rectangle_fill_min_max(), and adl_tri_draw().

#### 4.1.3.15 adl_linear_map()

```
float adl_linear_map (
            float s,
            float min_in,
            float max_in,
            float min_out,
            float max_out )
```

Affine map from one scalar range to another (no clamping).

**Parameters**

| | |
|---|---|
| *s* | Input value. |
| *min_in* | Input range minimum. |
| *max_in* | Input range maximum. |
| *min_out* | Output range minimum. |
| *max_out* | Output range maximum. |

**Returns**

Mapped value in the output range (may exceed if s is out-of-range).

Definition at line 1865 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_curves_plot_on_figure(), and render().

#### 4.1.3.16 adl_linear_sRGB_to_okLab()

```
void adl_linear_sRGB_to_okLab (
            uint32_t hex_ARGB,
            float * L,
            float * a,
            float * b )
```

Convert a linear sRGB color (ARGB) to Oklab components.

Oklab components are returned in ranges: L in [0,1], a in [-0.5,0.5], b in [-0.5,0.5] (typical). Input is assumed to be linear sRGB.

**Parameters**

| | |
|---|---|
| *hex_ARGB* | Input color (0xAARRGGBB). Alpha is ignored. |
| *L* | [out] Perceptual lightness. |
| *a* | [out] First opponent axis. |
| *b* | [out] Second opponent axis. |

Definition at line 1945 of file Almog_Draw_Library.h.

References HexARGB_RGB_VAR.

Referenced by adl_linear_sRGB_to_okLch().

#### 4.1.3.17 adl_linear_sRGB_to_okLch()

```
void adl_linear_sRGB_to_okLch (
            uint32_t hex_ARGB,
```

```
float * L,
float * c,
float * h_deg )
```

Convert a linear sRGB color (ARGB) to OkLch components.

**Parameters**

| | |
|---|---|
| *hex_ARGB* | Input color (0xAARRGGBB). Alpha is ignored. |
| *L* | [out] Lightness in [0,1]. |
| *c* | [out] Chroma (non-negative). |
| *h_deg* | [out] Hue angle in degrees [-180,180] from atan2. |

Definition at line 2012 of file Almog_Draw_Library.h.

References adl_linear_sRGB_to_okLab(), and PI.

Referenced by adl_interpolate_ARGBcolor_on_okLch().

#### 4.1.3.18 adl_lines_draw()

```
void adl_lines_draw (
            const Mat2D_uint32 screen_mat,
            const Point * points,
            const size_t len,
            const uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Draw a polyline connecting an array of points.

Draws segments between consecutive points: p[0]-p[1]-...-p[len-1].

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *points* | Array of points in pixel space (before pan/zoom). |
| *len* | Number of points in the array (>= 1). |
| *color* | Line color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 470 of file Almog_Draw_Library.h.

References adl_line_draw(), and points.

Referenced by adl_grid_draw().

#### 4.1.3.19 adl_lines_loop_draw()

```
void adl_lines_loop_draw (
            const Mat2D_uint32 screen_mat,
            const Point * points,
            const size_t len,
            const uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Draw a closed polyline (loop).

Same as adl_lines_draw, plus an extra segment from the last point back to the first point.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *points* | Array of points in pixel space (before pan/zoom). |
| *len* | Number of points in the array (>= 1). |
| *color* | Line color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 490 of file Almog_Draw_Library.h.

References adl_line_draw(), and points.

Referenced by adl_quad_draw().

#### 4.1.3.20 adl_max_min_values_draw_on_figure()

```
void adl_max_min_values_draw_on_figure (
            Figure figure )
```

Draw min/max numeric labels for the current data range.

Renders textual min/max values for both axes inside the figure area. Assumes figure.min_x/max_x/min_y/max_y have been populated.

**Parameters**

| | |
|---|---|
| *figure* | Figure whose labels are drawn into its own pixel buffer. |

Definition at line 2170 of file Almog_Draw_Library.h.

References ADL_FIGURE_AXIS_COLOR, ADL_MAX_CHARACTER_OFFSET, ADL_MIN_CHARACTER_OFFSET, adl_sentence_draw(), Figure::max_x, Figure::max_x_pixel, Figure::max_y, Figure::max_y_pixel, Figure::min_x, Figure::min_x_pixel, Figure::min_y, Figure::min_y_pixel, Figure::offset_zoom_param, Figure::pixels_mat, Mat2D_uint32::rows, Figure::x_axis_head_size, and Figure::y_axis_head_size.

Referenced by adl_2Dscalar_interp_on_figure(), and adl_curves_plot_on_figure().

#### 4.1.3.21 adl_okLab_to_linear_sRGB()

```
void adl_okLab_to_linear_sRGB (
            float L,
            float a,
            float b,
            uint32_t * hex_ARGB )
```

Convert Oklab components to a linear sRGB ARGB color.

Output RGB components are clamped to [0,255], alpha is set to 255.

**Parameters**

| | |
|---|---|
| *L* | Oklab lightness. |
| *a* | Oklab a component. |
| *b* | Oklab b component. |
| *hex_ARGB* | [out] Output color (0xAARRGGBB, A=255). |

Definition at line 1980 of file Almog_Draw_Library.h.

References RGBA_hexARGB.

Referenced by adl_okLch_to_linear_sRGB().

#### 4.1.3.22 adl_okLch_to_linear_sRGB()

```
void adl_okLch_to_linear_sRGB (
            float L,
            float c,
            float h_deg,
            uint32_t * hex_ARGB )
```

Convert OkLch components to a linear sRGB ARGB color.

Hue is wrapped to [0,360). Output RGB is clamped to [0,255], alpha=255.

**Parameters**

| | |
|---|---|
| *L* | Lightness. |
| *c* | Chroma. |
| *h_deg* | Hue angle in degrees. |
| *hex_ARGB* | [out] Output color (0xAARRGGBB, A=255). |

Definition at line 2031 of file Almog_Draw_Library.h.

References adl_okLab_to_linear_sRGB(), and PI.

Referenced by adl_interpolate_ARGBcolor_on_okLch().

#### 4.1.3.23 adl_point_draw()

```
void adl_point_draw (
            Mat2D_uint32 screen_mat,
            int x,
            int y,
            uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Draw a single pixel with alpha blending.

Applies the pan/zoom transform and writes the pixel if it falls inside the destination bounds. The source color is blended over the existing pixel using the source alpha; the stored alpha is set to 255.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *x* | X coordinate in pixels (before pan/zoom). |
| *y* | Y coordinate in pixels (before pan/zoom). |
| *color* | Source color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 350 of file Almog_Draw_Library.h.

References Mat2D_uint32::cols, HexARGB_RGBA_VAR, MAT2D_AT_UINT32, Offset_zoom_param::offset_x, Offset_zoom_param::offset_y, RGBA_hexARGB, Mat2D_uint32::rows, and Offset_zoom_param::zoom_multiplier.

Referenced by adl_circle_draw(), adl_circle_fill(), adl_figure_copy_to_screen(), adl_line_draw(), adl_quad_fill(), adl_quad_fill_interpolate_color_mean_value(), adl_quad_fill_interpolate_normal_mean_value(), adl_tri_fill_Pinedas_rasterizer(), adl_tri_fill_Pinedas_rasterizer_interpolate_color(), and adl_tri_fill_Pinedas_rasterizer_interpolate_normal().

#### 4.1.3.24 adl_quad2tris()

```
void adl_quad2tris (
            Quad quad,
            Tri * tri1,
            Tri * tri2,
            char split_line[] )
```

Split a quad into two triangles along a chosen diagonal.

The split is controlled by split_line:

- "02" splits along diagonal from vertex 0 to vertex 2.
- "13" splits along diagonal from vertex 1 to vertex 3.

The function copies positions, per-vertex colors, light_intensity, and the to_draw flag into the output triangles.

**Parameters**

| | |
|---|---|
| *quad* | Input quad. |
| *tri1* | [out] First output triangle. |
| *tri2* | [out] Second output triangle. |
| *split_line* | Null-terminated code: "02" or "13". |

Definition at line 1885 of file Almog_Draw_Library.h.

References Tri::colors, Quad::colors, Tri::light_intensity, Quad::light_intensity, Tri::points, Quad::points, Tri::to_draw, and Quad::to_draw.

#### 4.1.3.25 adl_quad_draw()

```
void adl_quad_draw (
            Mat2D_uint32 screen_mat,
            Mat2D inv_z_buffer,
            Quad quad,
            uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Draw the outline of a quad (four points, looped).

Depth buffer is not used in this outline variant.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *inv_z_buffer* | Unused for outline; safe to pass a dummy Mat2D. |
| *quad* | Quad to draw in pixel space (before transform). |
| *color* | Stroke color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 1010 of file Almog_Draw_Library.h.

References adl_lines_loop_draw(), and Quad::points.

Referenced by adl_quad_mesh_draw(), and render().

#### 4.1.3.26 adl_quad_fill()

```
void adl_quad_fill (
            Mat2D_uint32 screen_mat,
            Mat2D inv_z_buffer,
            Quad quad,
```

```
            uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Fill a quad using mean-value (Barycentric) coordinates and flat base color.

Performs a depth test against inv_z_buffer and modulates the base color with the average light_intensity of the quad's vertices.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *inv_z_buffer* | Inverse-Z buffer (larger is closer). |
| *quad* | Quad in pixel space; points carry z and w for depth. |
| *color* | Base color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 1028 of file Almog_Draw_Library.h.

References adl_point_draw(), Mat2D_uint32::cols, edge_cross_point, HexARGB_RGBA_VAR, Quad::light_intensity, MAT2D_AT, Quad::points, RGBA_hexARGB, Mat2D_uint32::rows, Point::w, Point::x, Point::y, and Point::z.

Referenced by adl_quad_mesh_fill().

#### 4.1.3.27 adl_quad_fill_interpolate_color_mean_value()

```
void adl_quad_fill_interpolate_color_mean_value (
            Mat2D_uint32 screen_mat,
            Mat2D inv_z_buffer,
            Quad quad,
            Offset_zoom_param offset_zoom_param )
```

Fill a quad with per-vertex colors (mean value coords).

Interpolates ARGB vertex colors using mean-value coordinates, optionally modulated by the average light_intensity. Depth-tested.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *inv_z_buffer* | Inverse-Z buffer (larger is closer). |
| *quad* | Quad in pixel space with quad.colors[] set. |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 1216 of file Almog_Draw_Library.h.

References adl_point_draw(), adl_tan_half_angle(), Quad::colors, Mat2D_uint32::cols, edge_cross_point, HexARGB_RGBA_VAR, Quad::light_intensity, MAT2D_AT, Quad::points, RGBA_hexARGB, Mat2D_uint32::rows, Point::w, Point::x, Point::y, and Point::z.

Referenced by adl_2Dscalar_interp_on_figure(), adl_quad_mesh_fill_interpolate_color(), and render().

#### 4.1.3.28 adl_quad_fill_interpolate_normal_mean_value()

```
void adl_quad_fill_interpolate_normal_mean_value (
            Mat2D_uint32 screen_mat,
            Mat2D inv_z_buffer,
            Quad quad,
            uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Fill a quad with per-pixel light interpolation (mean value coords).

Interpolates light_intensity across the quad using mean-value coordinates and modulates a uniform base color. Depth-tested.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *inv_z_buffer* | Inverse-Z buffer (larger is closer). |
| *quad* | Quad in pixel space; points carry z and w for depth. |
| *color* | Base color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 1122 of file Almog_Draw_Library.h.

References adl_point_draw(), adl_tan_half_angle(), Mat2D_uint32::cols, edge_cross_point, HexARGB_RGBA_VAR, Quad::light_intensity, MAT2D_AT, Quad::points, RGBA_hexARGB, Mat2D_uint32::rows, Point::w, Point::x, Point::y, and Point::z.

Referenced by adl_quad_mesh_fill_interpolate_normal().

#### 4.1.3.29 adl_quad_mesh_draw()

```
void adl_quad_mesh_draw (
            Mat2D_uint32 screen_mat,
            Mat2D inv_z_buffer_mat,
            Quad_mesh mesh,
            uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Draw outlines for all quads in a mesh.

Skips elements with to_draw == false. Depth buffer is not used.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *inv_z_buffer_mat* | Unused for outline; safe to pass a dummy Mat2D. |
| *mesh* | Quad mesh (array + length). |
| *color* | Stroke color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 1320 of file Almog_Draw_Library.h.

References adl_assert_quad_is_valid, adl_quad_draw(), Quad_mesh::elements, Quad_mesh::length, and Quad::to_draw.

#### 4.1.3.30 adl_quad_mesh_fill()

```
void adl_quad_mesh_fill (
            Mat2D_uint32 screen_mat,
            Mat2D inv_z_buffer_mat,
            Quad_mesh mesh,
            uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Fill all quads in a mesh with a uniform base color.

Applies per-quad average light_intensity. Depth-tested.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *inv_z_buffer_mat* | Inverse-Z buffer (larger is closer). |
| *mesh* | Quad mesh (array + length). |
| *color* | Base color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 1344 of file Almog_Draw_Library.h.

References adl_assert_quad_is_valid, adl_quad_fill(), Quad_mesh::elements, Quad_mesh::length, and Quad::to_draw.

#### 4.1.3.31 adl_quad_mesh_fill_interpolate_color()

```
void adl_quad_mesh_fill_interpolate_color (
            Mat2D_uint32 screen_mat,
            Mat2D inv_z_buffer_mat,
            Quad_mesh mesh,
            Offset_zoom_param offset_zoom_param )
```

Fill all quads in a mesh using per-vertex colors.

Interpolates quad.colors[] across each quad with mean-value coordinates. Depth-tested.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *inv_z_buffer_mat* | Inverse-Z buffer (larger is closer). |
| *mesh* | Quad mesh (array + length). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 1401 of file Almog_Draw_Library.h.

References adl_assert_quad_is_valid, adl_quad_fill_interpolate_color_mean_value(), Quad_mesh::elements, Quad_mesh::length, and Quad::to_draw.

#### 4.1.3.32 adl_quad_mesh_fill_interpolate_normal()

```
void adl_quad_mesh_fill_interpolate_normal (
            Mat2D_uint32 screen_mat,
            Mat2D inv_z_buffer_mat,
            Quad_mesh mesh,
            uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Fill all quads in a mesh using interpolated lighting.

Interpolates light_intensity across quads and modulates a uniform base color. Depth-tested.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *inv_z_buffer_mat* | Inverse-Z buffer (larger is closer). |
| *mesh* | Quad mesh (array + length). |
| *color* | Base color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 1371 of file Almog_Draw_Library.h.

References adl_assert_quad_is_valid, adl_quad_fill_interpolate_normal_mean_value(), Quad_mesh::elements, HexARGB_RGBA_VAR, Quad_mesh::length, and Quad::to_draw.

#### 4.1.3.33 adl_rectangle_draw_min_max()

```
void adl_rectangle_draw_min_max (
            Mat2D_uint32 screen_mat,
            int min_x,
            int max_x,
            int min_y,
            int max_y,
            uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Draw a rectangle outline defined by min/max corners (inclusive).

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *min_x* | Minimum X (before pan/zoom). |

**Parameters**

| | |
|---|---|
| *max_x* | Maximum X (before pan/zoom). |
| *min_y* | Minimum Y (before pan/zoom). |
| *max_y* | Maximum Y (before pan/zoom). |
| *color* | Stroke color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 973 of file Almog_Draw_Library.h.

References adl_line_draw().

Referenced by adl_character_draw().

#### 4.1.3.34 adl_rectangle_fill_min_max()

```
void adl_rectangle_fill_min_max (
            Mat2D_uint32 screen_mat,
            int min_x,
            int max_x,
            int min_y,
            int max_y,
            uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Fill a rectangle defined by min/max corners (inclusive).

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *min_x* | Minimum X (before pan/zoom). |
| *max_x* | Maximum X (before pan/zoom). |
| *min_y* | Minimum Y (before pan/zoom). |
| *max_y* | Maximum Y (before pan/zoom). |
| *color* | Fill color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 992 of file Almog_Draw_Library.h.

References adl_line_draw().

Referenced by adl_character_draw().

#### 4.1.3.35 adl_sentence_draw()

```
void adl_sentence_draw (
            Mat2D_uint32 screen_mat,
```

```
const char sentence[],
size_t len,
const int x_top_left,
const int y_top_left,
const int hight_pixel,
const uint32_t color,
Offset_zoom_param offset_zoom_param )
```

Draw a horizontal sentence using vector glyphs.

Characters are laid out left-to-right with a spacing derived from the character height. All characters share the same height.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *sentence* | ASCII string buffer. |
| *len* | Number of characters to draw from sentence. |
| *x_top_left* | X of top-left of the first character (before transform). |
| *y_top_left* | Y of top-left of the first character (before transform). |
| *hight_pixel* | Character height in pixels (spelled as in API). |
| *color* | Stroke color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 949 of file Almog_Draw_Library.h.

References adl_character_draw(), ADL_MAX_CHARACTER_OFFSET, and ADL_MIN_CHARACTER_OFFSET.

Referenced by adl_max_min_values_draw_on_figure().

### 4.1.3.36 adl_tan_half_angle()

```
float adl_tan_half_angle (
    Point vi,
    Point vj,
    Point p,
    float li,
    float lj )
```

Compute tan(alpha/2) for the angle at point p between segments p->vi and p->vj.

Uses the identity tan(alpha/2) = |a×b| / (|a||b| + a·b), where a = vi - p and b = vj - p. The lengths li = |a| and lj = |b| are passed in to avoid recomputation.

**Parameters**

| | |
|---|---|
| *vi* | Vertex i. |
| *vj* | Vertex j. |
| *p* | Pivot point. |
| *li* | Precomputed \|vi - p\|. |
| *lj* | Precomputed \|vj - p\|. |

**Returns**

tan(alpha/2) (non-negative).

Definition at line 1845 of file Almog_Draw_Library.h.

References Point::x, and Point::y.

Referenced by adl_quad_fill_interpolate_color_mean_value(), and adl_quad_fill_interpolate_normal_mean_value().

#### 4.1.3.37 adl_tri_draw()

```
void adl_tri_draw (
            Mat2D_uint32 screen_mat,
            Tri tri,
            uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Draw the outline of a triangle.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *tri* | Triangle in pixel space (before transform). |
| *color* | Stroke color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 1469 of file Almog_Draw_Library.h.

References adl_line_draw(), Tri::points, tri, Point::x, and Point::y.

Referenced by adl_tri_mesh_draw(), and render().

#### 4.1.3.38 adl_tri_fill_Pinedas_rasterizer()

```
void adl_tri_fill_Pinedas_rasterizer (
            Mat2D_uint32 screen_mat,
            Mat2D inv_z_buffer,
            Tri tri,
            uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Fill a triangle using Pineda's rasterizer with flat base color.

Uses the top-left fill convention and performs a depth test using inverse-Z computed from per-vertex z and w.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *inv_z_buffer* | Inverse-Z buffer (larger is closer). |
| *tri* | Triangle in pixel space; points carry z and w for depth. |
| *color* | Base color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 1492 of file Almog_Draw_Library.h.

References adl_point_draw(), Mat2D_uint32::cols, edge_cross_point, HexARGB_RGBA_VAR, is_top_left, Tri::light_intensity, MAT2D_AT, MATRIX2D_ASSERT, Tri::points, RGBA_hexARGB, Mat2D_uint32::rows, tri, Point::w, Point::x, Point::y, and Point::z.

Referenced by adl_tri_mesh_fill_Pinedas_rasterizer().

### 4.1.3.39 adl_tri_fill_Pinedas_rasterizer_interpolate_color()

```
void adl_tri_fill_Pinedas_rasterizer_interpolate_color (
            Mat2D_uint32 screen_mat,
            Mat2D inv_z_buffer,
            Tri tri,
            Offset_zoom_param offset_zoom_param )
```

Fill a triangle using Pineda's rasterizer with per-vertex colors.

Interpolates tri.colors[] and optionally modulates by average light_intensity. Depth-tested.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *inv_z_buffer* | Inverse-Z buffer (larger is closer). |
| *tri* | Triangle in pixel space with colors set. |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 1573 of file Almog_Draw_Library.h.

References adl_point_draw(), Tri::colors, Mat2D_uint32::cols, edge_cross_point, HexARGB_RGBA_VAR, is_top_left, Tri::light_intensity, MAT2D_AT, MATRIX2D_ASSERT, Tri::points, RGBA_hexARGB, Mat2D_uint32::rows, tri, Point::w, Point::x, Point::y, and Point::z.

Referenced by adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_color(), and render().

### 4.1.3.40 adl_tri_fill_Pinedas_rasterizer_interpolate_normal()

```
void adl_tri_fill_Pinedas_rasterizer_interpolate_normal (
            Mat2D_uint32 screen_mat,
```

```
            Mat2D inv_z_buffer,
            Tri tri,
            uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Fill a triangle with interpolated lighting over a uniform color.

Interpolates light_intensity across the triangle and modulates a uniform base color. Depth-tested.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *inv_z_buffer* | Inverse-Z buffer (larger is closer). |
| *tri* | Triangle in pixel space; points carry z and w for depth. |
| *color* | Base color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 1664 of file Almog_Draw_Library.h.

References adl_point_draw(), Mat2D_uint32::cols, edge_cross_point, HexARGB_RGBA_VAR, is_top_left, Tri::light_intensity, MAT2D_AT, MATRIX2D_ASSERT, Tri::points, RGBA_hexARGB, Mat2D_uint32::rows, tri, Point::w, Point::x, Point::y, and Point::z.

Referenced by adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_normal().

#### 4.1.3.41 adl_tri_mesh_draw()

```
void adl_tri_mesh_draw (
            Mat2D_uint32 screen_mat,
            Tri_mesh mesh,
            uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Draw outlines for all triangles in a mesh.

Skips elements with to_draw == false.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *mesh* | Triangle mesh (array + length). |
| *color* | Stroke color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 1746 of file Almog_Draw_Library.h.

References adl_tri_draw(), Tri_mesh::elements, Tri_mesh::length, Tri::to_draw, and tri.

#### 4.1.3.42 adl_tri_mesh_fill_Pinedas_rasterizer()

```
void adl_tri_mesh_fill_Pinedas_rasterizer (
            Mat2D_uint32 screen_mat,
            Mat2D inv_z_buffer_mat,
            Tri_mesh mesh,
            uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Fill all triangles in a mesh with a uniform base color.

Applies average light_intensity per triangle. Depth-tested.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *inv_z_buffer_mat* | Inverse-Z buffer (larger is closer). |
| *mesh* | Triangle mesh (array + length). |
| *color* | Base color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 1768 of file Almog_Draw_Library.h.

References adl_assert_tri_is_valid, adl_tri_fill_Pinedas_rasterizer(), Tri_mesh::elements, Tri_mesh::length, Tri::to_draw, and tri.

#### 4.1.3.43 adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_color()

```
void adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_color (
            Mat2D_uint32 screen_mat,
            Mat2D inv_z_buffer_mat,
            Tri_mesh mesh,
            Offset_zoom_param offset_zoom_param )
```

Fill all triangles in a mesh with a uniform base color.

Applies average light_intensity per triangle. Depth-tested.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *inv_z_buffer_mat* | Inverse-Z buffer (larger is closer). |
| *mesh* | Triangle mesh (array + length). |
| *color* | Base color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 1792 of file Almog_Draw_Library.h.

References adl_assert_tri_is_valid, adl_tri_fill_Pinedas_rasterizer_interpolate_color(), Tri_mesh::elements, Tri_mesh::length, Tri::to_draw, and tri.

#### 4.1.3.44 adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_normal()

```
void adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_normal (
            Mat2D_uint32 screen_mat,
            Mat2D inv_z_buffer_mat,
            Tri_mesh mesh,
            uint32_t color,
            Offset_zoom_param offset_zoom_param )
```

Fill all triangles in a mesh with interpolated lighting.

Interpolates light_intensity across each triangle and modulates a uniform base color. Depth-tested.

**Parameters**

| | |
|---|---|
| *screen_mat* | Destination ARGB pixel buffer. |
| *inv_z_buffer_mat* | Inverse-Z buffer (larger is closer). |
| *mesh* | Triangle mesh (array + length). |
| *color* | Base color (0xAARRGGBB). |
| *offset_zoom_param* | Pan/zoom transform. Use ADL_DEFAULT_OFFSET_ZOOM for identity. |

Definition at line 1817 of file Almog_Draw_Library.h.

References adl_assert_tri_is_valid, adl_tri_fill_Pinedas_rasterizer_interpolate_normal(), Tri_mesh::elements, Tri_mesh::length, Tri::to_draw, and tri.

## 4.2 Almog_Draw_Library.h

```

#ifndef ALMOG_DRAW_LIBRARY_H_
#define ALMOG_DRAW_LIBRARY_H_

#include <math.h>
#include <stdint.h>
#include <limits.h>
#include <string.h>
#include <float.h>

#include "./Matrix2D.h"
#include "./Almog_Dynamic_Array.h"

#ifndef ADL_ASSERT
#include <assert.h>
#define ADL_ASSERT assert
#endif

typedef struct {
    float zoom_multiplier;
    float offset_x;
    float offset_y;
    int mouse_x;
    int mouse_y;
} Offset_zoom_param;

#ifndef POINT
#define POINT
typedef struct {
    float x;
    float y;
    float z;
    float w;
} Point ;
```

```
#endif

#ifndef CURVE
#define CURVE
typedef struct {
    uint32_t color;
    size_t length;
    size_t capacity;
    Point *elements;
} Curve;
#endif

#ifndef CURVE_ADA
#define CURVE_ADA
typedef struct {
    size_t length;
    size_t capacity;
    Curve *elements;
} Curve_ada;
#endif

#ifndef TRI
#define TRI
typedef struct {
    Point points[3];
    Point tex_points[3];
    Point normals[3];
    uint32_t colors[3];
    bool to_draw;
    float light_intensity[3];
} Tri;
#endif

#ifndef QUAD
#define QUAD
typedef struct {
    Point points[4];
    Point normals[4];
    uint32_t colors[4];
    bool to_draw;
    float light_intensity[4];
} Quad;
#endif

#ifndef TRI_MESH
#define TRI_MESH
typedef struct {
    size_t length;
    size_t capacity;
    Tri *elements;
} Tri_mesh; /* Tri ada array */
#endif

#ifndef QUAD_MESH
#define QUAD_MESH
typedef struct {
    size_t length;
    size_t capacity;
    Quad *elements;
} Quad_mesh; /* Quad ada array */
#endif

typedef struct {
    int min_x_pixel;
    int max_x_pixel;
    int min_y_pixel;
    int max_y_pixel;
    float min_x;
    float max_x;
    float min_y;
    float max_y;
    int x_axis_head_size;
    int y_axis_head_size;
    Offset_zoom_param offset_zoom_param;
    Curve_ada src_curve_array;
    Point top_left_position;
    Mat2D_uint32 pixels_mat;
    Mat2D inv_z_buffer_mat;
    uint32_t background_color;
    bool to_draw_axis;
    bool to_draw_max_min_values;
} Figure;

typedef struct {
    Curve_ada curves;
    float min_e1;
    float max_e1;
```

```
    float min_e2;
    float max_e2;
    int num_samples_e1;
    int num_samples_e2;
    float de1;
    float de2;
    char plane[3];
} Grid; /* direction: e1, e2 */


#ifndef HexARGB_RGBA
#define HexARGB_RGBA(x) ((x)>>(8*2)&0xFF), ((x)>>(8*1)&0xFF), ((x)>>(8*0)&0xFF), ((x)>>(8*3)&0xFF)
#endif
#ifndef HexARGB_RGB_VAR
#define HexARGB_RGB_VAR(x, r, g, b) r = ((x)>>(8*2)&0xFF); g = ((x)>>(8*1)&0xFF); b = ((x)>>(8*0)&0xFF);
#endif
#ifndef HexARGB_RGBA_VAR
#define HexARGB_RGBA_VAR(x, r, g, b, a) r = ((x)>>(8*2)&0xFF); g = ((x)>>(8*1)&0xFF); b = ((x)>>(8*0)&0xFF); a = ((x)>>(8*3)&0xFF)
#endif
#ifndef RGB_hexRGB
#define RGB_hexRGB(r, g, b) (int)(0x010000*(int)(r) + 0x000100*(int)(g) + 0x000001*(int)(b))
#endif
#ifndef RGBA_hexARGB
#define RGBA_hexARGB(r, g, b, a) (int)(0x01000000l*(int)(fminf(a, 255)) + 0x010000*(int)(r) + 0x000100*(int)(g) + 0x000001*(int)(b))
#endif


void    adl_point_draw(Mat2D_uint32 screen_mat, int x, int y, uint32_t color, Offset_zoom_param offset_zoom_param);
void    adl_line_draw(Mat2D_uint32 screen_mat, const float x1_input, const float y1_input, const float x2_input, const float y2_input, uint32_t color, Offset_zoom_param offset_zoom_param);
void    adl_lines_draw(const Mat2D_uint32 screen_mat, const Point *points, const size_t len, const uint32_t color, Offset_zoom_param offset_zoom_param);
void    adl_lines_loop_draw(const Mat2D_uint32 screen_mat, const Point *points, const size_t len, const uint32_t color, Offset_zoom_param offset_zoom_param);
void    adl_arrow_draw(Mat2D_uint32 screen_mat, int xs, int ys, int xe, int ye, float head_size, float angle_deg, uint32_t color, Offset_zoom_param offset_zoom_param);

void    adl_character_draw(Mat2D_uint32 screen_mat, char c, int width_pixel, int hight_pixel, int x_top_left, int y_top_left, uint32_t color, Offset_zoom_param offset_zoom_param);
void    adl_sentence_draw(Mat2D_uint32 screen_mat, const char sentence[], size_t len, const int x_top_left, const int y_top_left, const int hight_pixel, const uint32_t color, Offset_zoom_param offset_zoom_param);

void    adl_rectangle_draw_min_max(Mat2D_uint32 screen_mat, int min_x, int max_x, int min_y, int max_y, uint32_t color, Offset_zoom_param offset_zoom_param);
void    adl_rectangle_fill_min_max(Mat2D_uint32 screen_mat, int min_x, int max_x, int min_y, int max_y, uint32_t color, Offset_zoom_param offset_zoom_param);

void    adl_quad_draw(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer, Quad quad, uint32_t color, Offset_zoom_param offset_zoom_param);
void    adl_quad_fill(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer, Quad quad, uint32_t color, Offset_zoom_param offset_zoom_param);
void    adl_quad_fill_interpolate_normal_mean_value(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer, Quad quad, uint32_t color, Offset_zoom_param offset_zoom_param);
void    adl_quad_fill_interpolate_color_mean_value(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer, Quad quad, Offset_zoom_param offset_zoom_param);

void    adl_quad_mesh_draw(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer_mat, Quad_mesh mesh, uint32_t color, Offset_zoom_param offset_zoom_param);
void    adl_quad_mesh_fill(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer_mat, Quad_mesh mesh, uint32_t color, Offset_zoom_param offset_zoom_param);
void    adl_quad_mesh_fill_interpolate_normal(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer_mat, Quad_mesh mesh, uint32_t color, Offset_zoom_param offset_zoom_param);
void    adl_quad_mesh_fill_interpolate_color(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer_mat, Quad_mesh mesh, Offset_zoom_param offset_zoom_param);

void    adl_circle_draw(Mat2D_uint32 screen_mat, float center_x, float center_y, float r, uint32_t color, Offset_zoom_param offset_zoom_param);
void    adl_circle_fill(Mat2D_uint32 screen_mat, float center_x, float center_y, float r, uint32_t color, Offset_zoom_param offset_zoom_param);

void    adl_tri_draw(Mat2D_uint32 screen_mat, Tri tri, uint32_t color, Offset_zoom_param offset_zoom_param);
void    adl_tri_fill_Pinedas_rasterizer(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer, Tri tri, uint32_t color, Offset_zoom_param offset_zoom_param);
void    adl_tri_fill_Pinedas_rasterizer_interpolate_color(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer, Tri tri, Offset_zoom_param offset_zoom_param);
void    adl_tri_fill_Pinedas_rasterizer_interpolate_normal(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer, Tri tri, uint32_t color, Offset_zoom_param offset_zoom_param);

void    adl_tri_mesh_draw(Mat2D_uint32 screen_mat, Tri_mesh mesh, uint32_t color, Offset_zoom_param offset_zoom_param);
void    adl_tri_mesh_fill_Pinedas_rasterizer(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer_mat, Tri_mesh mesh, uint32_t color, Offset_zoom_param offset_zoom_param);
```

```c
void    adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_color(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer_mat, Tri_mesh mesh, Offset_zoom_param offset_zoom_param);
void    adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_normal(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer_mat, Tri_mesh mesh, uint32_t color, Offset_zoom_param offset_zoom_param);

float   adl_tan_half_angle(Point vi, Point vj, Point p, float li, float lj);
float   adl_linear_map(float s, float min_in, float max_in, float min_out, float max_out);
void    adl_quad2tris(Quad quad, Tri *tri1, Tri *tri2, char split_line[]);
void    adl_linear_sRGB_to_okLab(uint32_t hex_ARGB, float *L, float *a, float *b);
void    adl_okLab_to_linear_sRGB(float L, float a, float b, uint32_t *hex_ARGB);
void    adl_linear_sRGB_to_okLch(uint32_t hex_ARGB, float *L, float *c, float *h_deg);
void    adl_okLch_to_linear_sRGB(float L, float c, float h_deg, uint32_t *hex_ARGB);
void    adl_interpolate_ARGBcolor_on_okLch(uint32_t color1, uint32_t color2, float t, float num_of_rotations, uint32_t *color_out);

Figure  adl_figure_alloc(size_t rows, size_t cols, Point top_left_position);
void    adl_figure_copy_to_screen(Mat2D_uint32 screen_mat, Figure figure);
void    adl_axis_draw_on_figure(Figure *figure);
void    adl_max_min_values_draw_on_figure(Figure figure);
void    adl_curve_add_to_figure(Figure *figure, Point *src_points, size_t src_len, uint32_t color);
void    adl_curves_plot_on_figure(Figure figure);
void    adl_2Dscalar_interp_on_figure(Figure figure, double *x_2Dmat, double *y_2Dmat, double *scalar_2Dmat, int ni, int nj, char color_scale[], float num_of_rotations);

Grid    adl_cartesian_grid_create(float min_e1, float max_e1, float min_e2, float max_e2, int num_samples_e1, int num_samples_e2, char plane[], float third_direction_position);
void    adl_grid_draw(Mat2D_uint32 screen_mat, Grid grid, uint32_t color, Offset_zoom_param offset_zoom_param);

#endif /*ALMOG_RENDER_SHAPES_H_*/

#ifdef ALMOG_DRAW_LIBRARY_IMPLEMENTATION
#undef ALMOG_DRAW_LIBRARY_IMPLEMENTATION

#define RED_hexARGB    0xFFFF0000
#define GREEN_hexARGB  0xFF00FF00
#define BLUE_hexARGB   0xFF0000FF
#define PURPLE_hexARGB 0xFFFF00FF
#define CYAN_hexARGB   0xFF00FFFF
#define YELLOW_hexARGB 0xFFFFFF00

#define edge_cross_point(a1, b, a2, p) (b.x-a1.x)*(p.y-a2.y)-(b.y-a1.y)*(p.x-a2.x)
#define is_top_edge(x, y) (y == 0 && x > 0)
#define is_left_edge(x, y) (y < 0)
#define is_top_left(ps, pe) (is_top_edge(pe.x-ps.x, pe.y-ps.y) || is_left_edge(pe.x-ps.x, pe.y-ps.y))

#define ADL_MAX_POINT_VAL 1e5
#define adl_assert_point_is_valid(p) ADL_ASSERT(isfinite(p.x) && isfinite(p.y) && isfinite(p.z) && isfinite(p.w))
#define adl_assert_tri_is_valid(tri) adl_assert_point_is_valid(tri.points[0]); \
        adl_assert_point_is_valid(tri.points[1]);                              \
        adl_assert_point_is_valid(tri.points[2])
#define adl_assert_quad_is_valid(quad) adl_assert_point_is_valid(quad.points[0]);   \
        adl_assert_point_is_valid(quad.points[1]);                                  \
        adl_assert_point_is_valid(quad.points[2]);                                  \
        adl_assert_point_is_valid(quad.points[3])

#define ADL_FIGURE_PADDING_PRECENTAGE 20
#define ADL_MAX_FIGURE_PADDING 70
#define ADL_MIN_FIGURE_PADDING 20
#define ADL_MAX_HEAD_SIZE 15
#define ADL_FIGURE_HEAD_ANGLE_DEG 30
#define ADL_FIGURE_AXIS_COLOR 0xff000000

#define ADL_MAX_CHARACTER_OFFSET 10
#define ADL_MIN_CHARACTER_OFFSET 5
#define ADL_MAX_SENTENCE_LEN 256
#define ADL_MAX_ZOOM 1e3

#define ADL_DEFAULT_OFFSET_ZOOM (Offset_zoom_param){1,0,0,0,0}
#define adl_offset_zoom_point(p, window_w, window_h, offset_zoom_param)                 \
    (p).x = ((p).x - (window_w)/2 + offset_zoom_param.offset_x) * offset_zoom_param.zoom_multiplier + (window_w)/2; \
    (p).y = ((p).y - (window_h)/2 + offset_zoom_param.offset_y) * offset_zoom_param.zoom_multiplier + (window_h)/2

void adl_point_draw(Mat2D_uint32 screen_mat, int x, int y, uint32_t color, Offset_zoom_param offset_zoom_param)
{
    float window_w = (float)screen_mat.cols;
    float window_h = (float)screen_mat.rows;

    x = (x - window_w/2 + offset_zoom_param.offset_x) * offset_zoom_param.zoom_multiplier + window_w/2;
    y = (y - window_h/2 + offset_zoom_param.offset_y) * offset_zoom_param.zoom_multiplier + window_h/2;
```

```c

    if ((x < (int)screen_mat.cols && y < (int)screen_mat.rows) && (x >= 0 && y >= 0)) { /* point is in screen */
        uint8_t r_new, g_new, b_new, a_new;
        uint8_t r_current, g_current, b_current, a_current;
        HexARGB_RGBA_VAR(MAT2D_AT_UINT32(screen_mat, y, x), r_current, g_current, b_current, a_current);
        HexARGB_RGBA_VAR(color, r_new, g_new, b_new, a_new);
        MAT2D_AT_UINT32(screen_mat, y, x) = RGBA_hexARGB(r_current*(1-a_new/255.0f) + r_new*a_new/255.0f, g_current*(1-a_new/255.0f) + g_new*a_new/255.0f, b_current*(1-a_new/255.0f) + b_new*a_new/255.0f, 255);
        (void)a_current;
    }
}

void adl_line_draw(Mat2D_uint32 screen_mat, const float x1_input, const float y1_input, const float x2_input, const float y2_input, uint32_t color, Offset_zoom_param offset_zoom_param)
{
    /* This function is inspired by the Olive.c function developed by 'Tsoding' on his YouTube channel. You can fined the video in this link: https://youtu.be/LmQKZmQh1ZQ?list=PLpM-Dvs8t0Va-Gb0Dp4d9t8yvNFHaKH6N&t=4683. */

    float window_w = (float)screen_mat.cols;
    float window_h = (float)screen_mat.rows;

    int x1 = (x1_input - window_w/2 + offset_zoom_param.offset_x) * offset_zoom_param.zoom_multiplier + window_w/2;
    int x2 = (x2_input - window_w/2 + offset_zoom_param.offset_x) * offset_zoom_param.zoom_multiplier + window_w/2;
    int y1 = (y1_input - window_h/2 + offset_zoom_param.offset_y) * offset_zoom_param.zoom_multiplier + window_h/2;
    int y2 = (y2_input - window_h/2 + offset_zoom_param.offset_y) * offset_zoom_param.zoom_multiplier + window_h/2;

    ADL_ASSERT((int)fabsf(fabsf((float)x2) - fabsf((float)x1)) < ADL_MAX_POINT_VAL);
    ADL_ASSERT((int)fabsf(fabsf((float)y2) - fabsf((float)y1)) < ADL_MAX_POINT_VAL);

    int x = x1;
    int y = y1;
    int dx, dy;

    adl_point_draw(screen_mat, x, y, color, (Offset_zoom_param){1,0,0,0,0});

    dx = x2 - x1;
    dy = y2 - y1;

    ADL_ASSERT(dy > INT_MIN && dy < INT_MAX);
    ADL_ASSERT(dx > INT_MIN && dx < INT_MAX);

    if (0 == dx && 0 == dy) return;
    if (0 == dx) {
        while (x != x2 || y != y2) {
            if (dy > 0) {
                y++;
            }
            if (dy < 0) {
                y--;
            }
            adl_point_draw(screen_mat, x, y, color, (Offset_zoom_param){1,0,0,0,0});
        }
        return;
    }
    if (0 == dy) {
        while (x != x2 || y != y2) {
            if (dx > 0) {
                x++;
            }
            if (dx < 0) {
                x--;
            }
            adl_point_draw(screen_mat, x, y, color, (Offset_zoom_param){1,0,0,0,0});
        }
        return;
    }

    /* float m = (float)dy / dx */
    int b = y1 - dy * x1 / dx;

    if (x1 > x2) {
        int temp_x = x1;
        x1 = x2;
        x2 = temp_x;
    }
    for (x = x1; x < x2; x++) {
        int sy1 = dy * x / dx + b;
        int sy2 = dy * (x + 1) / dx + b;
        if (sy1 > sy2) {
```

```
            int temp_y = sy1;
            sy1 = sy2;
            sy2 = temp_y;
        }
        for (y = sy1; y <= sy2; y++) {
            adl_point_draw(screen_mat, x, y, color, (Offset_zoom_param){1,0,0,0,0});
        }
    }

}

void adl_lines_draw(const Mat2D_uint32 screen_mat, const Point *points, const size_t len, const uint32_t color, Offset_zoom_param offset_zoom_param)
{
    if (len == 0) return;
    for (size_t i = 0; i < len-1; i++) {
        adl_line_draw(screen_mat, points[i].x, points[i].y, points[i+1].x, points[i+1].y, color, offset_zoom_param);
    }
}

void adl_lines_loop_draw(const Mat2D_uint32 screen_mat, const Point *points, const size_t len, const uint32_t color, Offset_zoom_param offset_zoom_param)
{
    if (len == 0) return;
    for (size_t i = 0; i < len-1; i++) {
        adl_line_draw(screen_mat, points[i].x, points[i].y, points[i+1].x, points[i+1].y, color, offset_zoom_param);
    }
    adl_line_draw(screen_mat, points[len-1].x, points[len-1].y, points[0].x, points[0].y, color, offset_zoom_param);
}


void adl_arrow_draw(Mat2D_uint32 screen_mat, int xs, int ys, int xe, int ye, float head_size, float angle_deg, uint32_t color, Offset_zoom_param offset_zoom_param)
{
    Mat2D pe = mat2D_alloc(3, 1);
    mat2D_fill(pe, 0);
    MAT2D_AT(pe, 0, 0) = xe;
    MAT2D_AT(pe, 1, 0) = ye;
    Mat2D v1 = mat2D_alloc(3, 1);
    mat2D_fill(v1, 0);
    Mat2D v2 = mat2D_alloc(3, 1);
    mat2D_fill(v2, 0);
    Mat2D temp_v = mat2D_alloc(3, 1);
    mat2D_fill(temp_v, 0);
    Mat2D DCM_p = mat2D_alloc(3, 3);
    mat2D_fill(DCM_p, 0);
    mat2D_set_rot_mat_z(DCM_p, angle_deg);
    Mat2D DCM_m = mat2D_alloc(3, 3);
    mat2D_fill(DCM_m, 0);
    mat2D_set_rot_mat_z(DCM_m, -angle_deg);

    int x_center = xs*head_size + xe*(1-head_size);
    int y_center = ys*head_size + ye*(1-head_size);

    MAT2D_AT(v1, 0, 0) = x_center;
    MAT2D_AT(v1, 1, 0) = y_center;
    mat2D_copy(v2, v1);

    /* v1 */
    mat2D_copy(temp_v, v1);
    mat2D_sub(temp_v, pe);
    mat2D_fill(v1, 0);
    mat2D_dot(v1, DCM_p, temp_v);
    mat2D_add(v1, pe);

    /* v2 */
    mat2D_copy(temp_v, v2);
    mat2D_sub(temp_v, pe);
    mat2D_fill(v2, 0);
    mat2D_dot(v2, DCM_m, temp_v);
    mat2D_add(v2, pe);

    adl_line_draw(screen_mat, MAT2D_AT(v1, 0, 0), MAT2D_AT(v1, 1, 0), xe, ye, color, offset_zoom_param);
    adl_line_draw(screen_mat, MAT2D_AT(v2, 0, 0), MAT2D_AT(v2, 1, 0), xe, ye, color, offset_zoom_param);
    adl_line_draw(screen_mat, xs, ys, xe, ye, color, offset_zoom_param);

    mat2D_free(pe);
    mat2D_free(v1);
    mat2D_free(v2);
    mat2D_free(temp_v);
    mat2D_free(DCM_p);
    mat2D_free(DCM_m);
```

```c
}

void adl_character_draw(Mat2D_uint32 screen_mat, char c, int width_pixel, int hight_pixel, int x_top_left, int y_top_left, uint32_t color, Offset_zoom_param offset_zoom_param)
{
    switch (c)
    {
    case 'a':
    case 'A':
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel, x_top_left+width_pixel/2, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/2, y_top_left, x_top_left+width_pixel, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/6, y_top_left+2*hight_pixel/3, x_top_left+5*width_pixel/6, y_top_left+2*hight_pixel/3, color, offset_zoom_param);
        break;
    case 'b':
    case 'B':
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left+2*width_pixel/3, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left, x_top_left+width_pixel, y_top_left+hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel/6, x_top_left+width_pixel, y_top_left+hight_pixel/3, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel/3, x_top_left+2*width_pixel/3, y_top_left+hight_pixel/2, color, offset_zoom_param);

        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel/2, x_top_left, y_top_left+hight_pixel/2, color, offset_zoom_param);

        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel/2, x_top_left+width_pixel, y_top_left+2*hight_pixel/3, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+2*hight_pixel/3, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, x_top_left, y_top_left+hight_pixel, color, offset_zoom_param);
        break;
    case 'c':
    case 'C':
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left, x_top_left+width_pixel/3, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left, x_top_left, y_top_left+hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/6, x_top_left, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+5*hight_pixel/6, x_top_left+width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left+hight_pixel, color, offset_zoom_param);
        break;
    case 'd':
    case 'D':
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left+2*width_pixel/3, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left, x_top_left+width_pixel, y_top_left+hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel/6, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, x_top_left, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel, x_top_left, y_top_left, color, offset_zoom_param);
        break;
    case 'e':
    case 'E':
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left, x_top_left, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left+hight_pixel, color, offset_zoom_param);

        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/2, x_top_left+width_pixel, y_top_left+hight_pixel/2, color, offset_zoom_param);
        break;
    case 'f':
    case 'F':
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left, x_top_left, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left, y_top_left+hight_pixel, color, offset_zoom_param);
```

```

        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/2, x_top_left+width_pixel, y_top_left+hight_pixel/2, color, offset_zoom_param);
        break;
    case 'g':
    case 'G':
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel/6, x_top_left+2*width_pixel/3, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left, x_top_left+width_pixel/3, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left, x_top_left, y_top_left+hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/6, x_top_left, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+5*hight_pixel/6, x_top_left+width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left+hight_pixel, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, x_top_left+width_pixel, y_top_left+hight_pixel/2, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel/2, x_top_left+width_pixel/2, y_top_left+hight_pixel/2, color, offset_zoom_param);
        break;
    case 'h':
    case 'H':
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left, x_top_left+width_pixel, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/2, x_top_left+width_pixel, y_top_left+hight_pixel/2, color, offset_zoom_param);
        break;
    case 'i':
    case 'I':
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left+width_pixel, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/2, y_top_left, x_top_left+width_pixel/2, y_top_left+hight_pixel, color, offset_zoom_param);
        break;
    case 'j':
    case 'J':
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left+width_pixel, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left, x_top_left+2*width_pixel/3, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+5*hight_pixel/6, x_top_left+width_pixel/2, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/2, y_top_left+hight_pixel, x_top_left+width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left+hight_pixel, x_top_left+width_pixel/6, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        break;
    case 'k':
    case 'K':
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/2, x_top_left+width_pixel, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/2, x_top_left+width_pixel, y_top_left, color, offset_zoom_param);
        break;
    case 'l':
    case 'L':
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left+hight_pixel, color, offset_zoom_param);
        break;
    case 'm':
    case 'M':
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel, x_top_left, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left+width_pixel/2, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/2, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left, x_top_left+width_pixel, y_top_left+hight_pixel, color, offset_zoom_param);
        break;
    case 'n':
    case 'N':
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel, x_top_left, y_top_left, color, offset_zoom_param);
```

```
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left+width_pixel, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left, color, offset_zoom_param);
        break;
    case 'o':
    case 'O':
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left, x_top_left+width_pixel/3, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left, x_top_left, y_top_left+hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/6, x_top_left, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+5*hight_pixel/6, x_top_left+width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left+hight_pixel, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, x_top_left+width_pixel, y_top_left+hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel/6, x_top_left+2*width_pixel/3, y_top_left, color, offset_zoom_param);
        break;
    case 'p':
    case 'P':
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left+2*width_pixel/3, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left, x_top_left+width_pixel, y_top_left+hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel/6, x_top_left+width_pixel, y_top_left+hight_pixel/3, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel/3, x_top_left+2*width_pixel/3, y_top_left+hight_pixel/2, color, offset_zoom_param);

        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel/2, x_top_left, y_top_left+hight_pixel/2, color, offset_zoom_param);
        break;
    case 'q':
    case 'Q':
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left, x_top_left+width_pixel/3, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left, x_top_left, y_top_left+hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/6, x_top_left, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+5*hight_pixel/6, x_top_left+width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left+hight_pixel, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, x_top_left+width_pixel, y_top_left+hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel/6, x_top_left+2*width_pixel/3, y_top_left, color, offset_zoom_param);

        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+5*hight_pixel/6, x_top_left+width_pixel, y_top_left+hight_pixel, color, offset_zoom_param);
        break;
    case 'r':
    case 'R':
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left+2*width_pixel/3, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left, x_top_left+width_pixel, y_top_left+hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel/6, x_top_left+width_pixel, y_top_left+hight_pixel/3, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel/3, x_top_left+2*width_pixel/3, y_top_left+hight_pixel/2, color, offset_zoom_param);

        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel/2, x_top_left, y_top_left+hight_pixel/2, color, offset_zoom_param);

        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel/2, x_top_left+width_pixel, y_top_left+hight_pixel, color, offset_zoom_param);
        break;
    case 's':
    case 'S':
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel/6, x_top_left+2*width_pixel/3, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left, x_top_left+width_pixel/3, y_top_left, color, offset_zoom_param);
```

```
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left, x_top_left, y_top_left+hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/6, x_top_left, y_top_left+hight_pixel/3, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/3, x_top_left+width_pixel/3, y_top_left+hight_pixel/2, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left+hight_pixel/2, x_top_left+2*width_pixel/3, y_top_left+hight_pixel/2, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel/2, x_top_left+width_pixel, y_top_left+2*hight_pixel/3, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+5*hight_pixel/6, x_top_left+width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left+hight_pixel, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, x_top_left+width_pixel, y_top_left+2*hight_pixel/3, color, offset_zoom_param);
        break;
    case 't':
    case 'T':
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left+width_pixel, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/2, y_top_left, x_top_left+width_pixel/2, y_top_left+hight_pixel, color, offset_zoom_param);
        break;
    case 'u':
    case 'U':
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left, y_top_left+hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/6, x_top_left, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+5*hight_pixel/6, x_top_left+width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left+hight_pixel, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, x_top_left+width_pixel, y_top_left, color, offset_zoom_param);
        break;
    case 'v':
    case 'V':
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left+width_pixel/2, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/2, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left, color, offset_zoom_param);
        break;
    case 'w':
    case 'W':
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left+width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left+hight_pixel, x_top_left+width_pixel/2, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/2, y_top_left, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left, color, offset_zoom_param);
        break;
    case 'x':
    case 'X':
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left+width_pixel, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left, color, offset_zoom_param);
        break;
    case 'y':
    case 'Y':
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left+width_pixel/2, y_top_left+hight_pixel/2, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left, x_top_left+width_pixel/2, y_top_left+hight_pixel/2, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/2, y_top_left+hight_pixel/2, x_top_left+width_pixel/2, y_top_left+hight_pixel, color, offset_zoom_param);
        break;
    case 'z':
    case 'Z':
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left+width_pixel, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left, x_top_left, y_top_left+hight_pixel, color, offset_zoom_param);
        break;
    case '.':
```

```
        adl_rectangle_fill_min_max(screen_mat, x_top_left+width_pixel/6, x_top_left+width_pixel/3, y_top_left+5*hight_pixel/6, y_top_left+hight_pixel, color, offset_zoom_param);
        break;
    case ':':
        adl_rectangle_fill_min_max(screen_mat, x_top_left+width_pixel/6, x_top_left+width_pixel/3, y_top_left+5*hight_pixel/6, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_rectangle_fill_min_max(screen_mat, x_top_left+width_pixel/6, x_top_left+width_pixel/3, y_top_left, y_top_left+hight_pixel/6, color, offset_zoom_param);
        break;
    case '0':
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left, x_top_left+width_pixel/3, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left, x_top_left, y_top_left+hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/6, x_top_left, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+5*hight_pixel/6, x_top_left+width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left+hight_pixel, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, x_top_left+width_pixel, y_top_left+hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel/6, x_top_left+2*width_pixel/3, y_top_left, color, offset_zoom_param);

        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel/6, x_top_left, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        break;
    case '1':
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/6, x_top_left+width_pixel/2, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/2, y_top_left, x_top_left+width_pixel/2, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left+hight_pixel, color, offset_zoom_param);
        break;
    case '2':
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/6, x_top_left+width_pixel/3, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left, x_top_left+2*width_pixel/3, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left, x_top_left+width_pixel, y_top_left+hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel/6, x_top_left+width_pixel, y_top_left+hight_pixel/3, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel/3, x_top_left, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left+hight_pixel, color, offset_zoom_param);
        break;
    case '3':
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/6, x_top_left+width_pixel/3, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left, x_top_left+2*width_pixel/3, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left, x_top_left+width_pixel, y_top_left+hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel/6, x_top_left+width_pixel, y_top_left+hight_pixel/3, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel/3, x_top_left+2*width_pixel/3, y_top_left+hight_pixel/2, color, offset_zoom_param);

        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel/2, x_top_left+width_pixel/3, y_top_left+hight_pixel/2, color, offset_zoom_param);

        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel/2, x_top_left+width_pixel, y_top_left+2*hight_pixel/3, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+2*hight_pixel/3, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, x_top_left+width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left+hight_pixel, x_top_left, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        break;
    case '4':
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, x_top_left+2*width_pixel/3, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left, x_top_left, y_top_left+2*hight_pixel/3, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+2*hight_pixel/3, x_top_left+width_pixel, y_top_left+2*hight_pixel/3, color, offset_zoom_param);
        break;
    case '5':
```

```
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left, x_top_left, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left, y_top_left+hight_pixel/2, color, offset_zoom_param);

        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/2, x_top_left+2*width_pixel/3, y_top_left+hight_pixel/2, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel/2, x_top_left+width_pixel, y_top_left+2*hight_pixel/3, color, offset_zoom_param);

        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, x_top_left+width_pixel, y_top_left+2*hight_pixel/3, color, offset_zoom_param);
        break;
    case '6':
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel/6, x_top_left+2*width_pixel/3, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left, x_top_left+width_pixel/3, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left, x_top_left, y_top_left+hight_pixel/6, color, offset_zoom_param);

        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/6, x_top_left, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+5*hight_pixel/6, x_top_left+width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left+hight_pixel, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, x_top_left+width_pixel, y_top_left+2*hight_pixel/3, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+2*hight_pixel/3, x_top_left+2*width_pixel/3, y_top_left+hight_pixel/2, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel/2, x_top_left+width_pixel/3, y_top_left+hight_pixel/2, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left+hight_pixel/2, x_top_left, y_top_left+2*hight_pixel/3, color, offset_zoom_param);
        break;
    case '7':
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left+width_pixel, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left, x_top_left+width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        break;
    case '8':
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel/2, x_top_left+width_pixel, y_top_left+hight_pixel/3, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel/3, x_top_left+width_pixel, y_top_left+hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel/6, x_top_left+2*width_pixel/3, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left, x_top_left+width_pixel/3, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left, x_top_left, y_top_left+hight_pixel/6, color, offset_zoom_param);

        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/6, x_top_left, y_top_left+hight_pixel/3, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/3, x_top_left+width_pixel/3, y_top_left+hight_pixel/2, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left+hight_pixel/2, x_top_left+2*width_pixel/3, y_top_left+hight_pixel/2, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel/2, x_top_left+width_pixel, y_top_left+2*hight_pixel/3, color, offset_zoom_param);

        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left+hight_pixel/2, x_top_left, y_top_left+2*hight_pixel/3, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+2*hight_pixel/3, x_top_left, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+5*hight_pixel/6, x_top_left+width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left+hight_pixel, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, x_top_left+width_pixel, y_top_left+2*hight_pixel/3, color, offset_zoom_param);
        break;
    case '9':
        adl_line_draw(screen_mat, x_top_left, y_top_left+5*hight_pixel/6, x_top_left+width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left+hight_pixel, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, color, offset_zoom_param);
```

```
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+5*hight_pixel/6, x_top_left+width_pixel, y_top_left+hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel, y_top_left+hight_pixel/6, x_top_left+2*width_pixel/3, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left, x_top_left+width_pixel/3, y_top_left, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left, x_top_left, y_top_left+hight_pixel/6, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/6, x_top_left, y_top_left+hight_pixel/3, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/3, x_top_left+width_pixel/3, y_top_left+hight_pixel/2, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/3, y_top_left+hight_pixel/2, x_top_left+2*width_pixel/3, y_top_left+hight_pixel/2, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+2*width_pixel/3, y_top_left+hight_pixel/2, x_top_left+width_pixel, y_top_left+hight_pixel/3, color, offset_zoom_param);
        break;
    case '-':
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/2, x_top_left+width_pixel, y_top_left+hight_pixel/2, color, offset_zoom_param);
        break;
    case '+':
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel/2, x_top_left+width_pixel, y_top_left+hight_pixel/2, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left+width_pixel/2, y_top_left, x_top_left+width_pixel/2, y_top_left+hight_pixel, color, offset_zoom_param);
        break;
    case ' ':
        break;
    default:
        adl_rectangle_draw_min_max(screen_mat, x_top_left, x_top_left+width_pixel, y_top_left, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left, x_top_left+width_pixel, y_top_left+hight_pixel, color, offset_zoom_param);
        adl_line_draw(screen_mat, x_top_left, y_top_left+hight_pixel, x_top_left+width_pixel, y_top_left, color, offset_zoom_param);
        break;
    }
}

void adl_sentence_draw(Mat2D_uint32 screen_mat, const char sentence[], size_t len, const int x_top_left, const int y_top_left, const int hight_pixel, const uint32_t color, Offset_zoom_param offset_zoom_param)
{
    int character_width_pixel = hight_pixel/2;
    int current_x_top_left = x_top_left;
    int character_x_offset = (int)fmaxf(fminf(ADL_MAX_CHARACTER_OFFSET, character_width_pixel / 5), ADL_MIN_CHARACTER_OFFSET);

    for (size_t char_index = 0; char_index < len; char_index++) {
        adl_character_draw(screen_mat, sentence[char_index], character_width_pixel, hight_pixel, current_x_top_left, y_top_left, color, offset_zoom_param);
        current_x_top_left += character_width_pixel + character_x_offset;
    }

}

void adl_rectangle_draw_min_max(Mat2D_uint32 screen_mat, int min_x, int max_x, int min_y, int max_y, uint32_t color, Offset_zoom_param offset_zoom_param)
{
    adl_line_draw(screen_mat, min_x, min_y, max_x, min_y, color, offset_zoom_param);
    adl_line_draw(screen_mat, min_x, max_y, max_x, max_y, color, offset_zoom_param);
    adl_line_draw(screen_mat, min_x, min_y, min_x, max_y, color, offset_zoom_param);
    adl_line_draw(screen_mat, max_x, min_y, max_x, max_y, color, offset_zoom_param);
}

void adl_rectangle_fill_min_max(Mat2D_uint32 screen_mat, int min_x, int max_x, int min_y, int max_y, uint32_t color, Offset_zoom_param offset_zoom_param)
{
    for (int y = min_y; y <= max_y; y++) {
        adl_line_draw(screen_mat, min_x, y, max_x, y, color, offset_zoom_param);
    }
}

void adl_quad_draw(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer, Quad quad, uint32_t color, Offset_zoom_param offset_zoom_param)
{
    (void)inv_z_buffer;
    adl_lines_loop_draw(screen_mat, quad.points, 4, color, offset_zoom_param);
}

void adl_quad_fill(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer, Quad quad, uint32_t color, Offset_zoom_param offset_zoom_param)
{
    Point p0 = quad.points[0];
```

```c
    Point p1 = quad.points[1];
    Point p2 = quad.points[2];
    Point p3 = quad.points[3];

    int x_min = fminf(p0.x, fminf(p1.x, fminf(p2.x, p3.x)));
    int x_max = fmaxf(p0.x, fmaxf(p1.x, fmaxf(p2.x, p3.x)));
    int y_min = fminf(p0.y, fminf(p1.y, fminf(p2.y, p3.y)));
    int y_max = fmaxf(p0.y, fmaxf(p1.y, fmaxf(p2.y, p3.y)));

    if (x_min < 0) x_min = 0;
    if (y_min < 0) y_min = 0;
    if (x_max >= (int)screen_mat.cols) x_max = (int)screen_mat.cols - 1;
    if (y_max >= (int)screen_mat.rows) y_max = (int)screen_mat.rows - 1;

    float w = edge_cross_point(p0, p1, p1, p2) + edge_cross_point(p2, p3, p3, p0);
    if (fabs(w) < 1e-6) {
        // adl_quad_draw(screen_mat, inv_z_buffer, quad, quad.colors[0], offset_zoom_param);
        return;
    }

    float size_p3_to_p0 = sqrt((p0.x - p3.x)*(p0.x - p3.x) + (p0.y - p3.y)*(p0.y - p3.y));
    float size_p0_to_p1 = sqrt((p1.x - p0.x)*(p1.x - p0.x) + (p1.y - p0.y)*(p1.y - p0.y));
    float size_p1_to_p2 = sqrt((p2.x - p1.x)*(p2.x - p1.x) + (p2.y - p1.y)*(p2.y - p1.y));
    float size_p2_to_p3 = sqrt((p3.x - p2.x)*(p3.x - p2.x) + (p3.y - p2.y)*(p3.y - p2.y));

    int r, g, b, a;
    HexARGB_RGBA_VAR(color, r, g, b, a);
    float light_intensity = (quad.light_intensity[0] + quad.light_intensity[1] + quad.light_intensity[2] + quad.light_intensity[3]) / 4;
    uint8_t base_r = (uint8_t)fmaxf(0, fminf(255, r * light_intensity));
    uint8_t base_g = (uint8_t)fmaxf(0, fminf(255, g * light_intensity));
    uint8_t base_b = (uint8_t)fmaxf(0, fminf(255, b * light_intensity));

    for (int y = y_min; y <= y_max; y++) {
        for (int x = x_min; x <= x_max; x++) {
            Point p = {.x = x, .y = y, .z = 0};
            bool in_01, in_12, in_23, in_30;

            in_01 = (edge_cross_point(p0, p1, p0, p) >= 0) != (w < 0);
            in_12 = (edge_cross_point(p1, p2, p1, p) >= 0) != (w < 0);
            in_23 = (edge_cross_point(p2, p3, p2, p) >= 0) != (w < 0);
            in_30 = (edge_cross_point(p3, p0, p3, p) >= 0) != (w < 0);

            /* https://www.mn.uio.no/math/english/people/aca/michaelf/papers/mv3d.pdf. */
            float size_p_to_p0 = sqrt((p0.x - p.x)*(p0.x - p.x) + (p0.y - p.y)*(p0.y - p.y));
            float size_p_to_p1 = sqrt((p1.x - p.x)*(p1.x - p.x) + (p1.y - p.y)*(p1.y - p.y));
            float size_p_to_p2 = sqrt((p2.x - p.x)*(p2.x - p.x) + (p2.y - p.y)*(p2.y - p.y));
            float size_p_to_p3 = sqrt((p3.x - p.x)*(p3.x - p.x) + (p3.y - p.y)*(p3.y - p.y));

            /* tangent of half the angle directly using vector math */
            float tan_theta_3_over_2 = size_p3_to_p0 / (size_p_to_p3 + size_p_to_p0);
            float tan_theta_0_over_2 = size_p0_to_p1 / (size_p_to_p0 + size_p_to_p1);
            float tan_theta_1_over_2 = size_p1_to_p2 / (size_p_to_p1 + size_p_to_p2);
            float tan_theta_2_over_2 = size_p2_to_p3 / (size_p_to_p2 + size_p_to_p3);
            float w0 = (tan_theta_3_over_2 + tan_theta_0_over_2) / size_p_to_p0;
            float w1 = (tan_theta_0_over_2 + tan_theta_1_over_2) / size_p_to_p1;
            float w2 = (tan_theta_1_over_2 + tan_theta_2_over_2) / size_p_to_p2;
            float w3 = (tan_theta_2_over_2 + tan_theta_3_over_2) / size_p_to_p3;

            float inv_w_tot = 1.0f / (w0 + w1 + w2 + w3);
            float alpha = w0 * inv_w_tot;
            float beta  = w1 * inv_w_tot;
            float gamma = w2 * inv_w_tot;
            float delta = w3 * inv_w_tot;

            if (in_01 && in_12 && in_23 && in_30) {

                double inv_w = alpha * (1.0f / p0.w) + beta  * (1.0f / p1.w) + gamma * (1.0f / p2.w) + delta * (1.0f / p3.w);
                double z_over_w = alpha * (p0.z / p0.w) + beta  * (p1.z / p1.w) + gamma * (p2.z / p2.w) + delta * (p3.z / p3.w);
                double inv_z = inv_w / z_over_w;

                if (inv_z >= MAT2D_AT(inv_z_buffer, y, x)) {
                    adl_point_draw(screen_mat, x, y, RGBA_hexARGB(base_r, base_g, base_b, a), offset_zoom_param);
                    MAT2D_AT(inv_z_buffer, y, x) = inv_z;
                }
            }
        }
    }
}

void adl_quad_fill_interpolate_normal_mean_value(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer, Quad quad, uint32_t color, Offset_zoom_param offset_zoom_param)
{
    Point p0 = quad.points[0];
```

```c
    Point p1 = quad.points[1];
    Point p2 = quad.points[2];
    Point p3 = quad.points[3];

    int x_min = fminf(p0.x, fminf(p1.x, fminf(p2.x, p3.x)));
    int x_max = fmaxf(p0.x, fmaxf(p1.x, fmaxf(p2.x, p3.x)));
    int y_min = fminf(p0.y, fminf(p1.y, fminf(p2.y, p3.y)));
    int y_max = fmaxf(p0.y, fmaxf(p1.y, fmaxf(p2.y, p3.y)));

    if (x_min < 0) x_min = 0;
    if (y_min < 0) y_min = 0;
    if (x_max >= (int)screen_mat.cols) x_max = (int)screen_mat.cols - 1;
    if (y_max >= (int)screen_mat.rows) y_max = (int)screen_mat.rows - 1;

    float w = edge_cross_point(p0, p1, p1, p2) + edge_cross_point(p2, p3, p3, p0);
    if (fabs(w) < 1e-6) {
        // adl_quad_draw(screen_mat, inv_z_buffer, quad, quad.colors[0], offset_zoom_param);
        return;
    }

    int r, g, b, a;
    HexARGB_RGBA_VAR(color, r, g, b, a);

    for (int y = y_min; y <= y_max; y++) {
        for (int x = x_min; x <= x_max; x++) {
            Point p = {.x = x, .y = y, .z = 0};
            bool in_01, in_12, in_23, in_30;

            in_01 = (edge_cross_point(p0, p1, p0, p) >= 0) != (w < 0);
            in_12 = (edge_cross_point(p1, p2, p1, p) >= 0) != (w < 0);
            in_23 = (edge_cross_point(p2, p3, p2, p) >= 0) != (w < 0);
            in_30 = (edge_cross_point(p3, p0, p3, p) >= 0) != (w < 0);

            /* using 'mean value coordinates'
             * https://www.mn.uio.no/math/english/people/aca/michaelf/papers/mv3d.pdf. */
            float size_p_to_p0 = sqrt((p0.x - p.x)*(p0.x - p.x) + (p0.y - p.y)*(p0.y - p.y));
            float size_p_to_p1 = sqrt((p1.x - p.x)*(p1.x - p.x) + (p1.y - p.y)*(p1.y - p.y));
            float size_p_to_p2 = sqrt((p2.x - p.x)*(p2.x - p.x) + (p2.y - p.y)*(p2.y - p.y));
            float size_p_to_p3 = sqrt((p3.x - p.x)*(p3.x - p.x) + (p3.y - p.y)*(p3.y - p.y));

            /* calculating the tangent of half the angle directly using vector math */
            float t0 = adl_tan_half_angle(p0, p1, p, size_p_to_p0, size_p_to_p1);
            float t1 = adl_tan_half_angle(p1, p2, p, size_p_to_p1, size_p_to_p2);
            float t2 = adl_tan_half_angle(p2, p3, p, size_p_to_p2, size_p_to_p3);
            float t3 = adl_tan_half_angle(p3, p0, p, size_p_to_p3, size_p_to_p0);

            float w0 = (t3 + t0) / size_p_to_p0;
            float w1 = (t0 + t1) / size_p_to_p1;
            float w2 = (t1 + t2) / size_p_to_p2;
            float w3 = (t2 + t3) / size_p_to_p3;

            float inv_w_tot = 1.0f / (w0 + w1 + w2 + w3);
            float alpha = w0 * inv_w_tot;
            float beta  = w1 * inv_w_tot;
            float gamma = w2 * inv_w_tot;
            float delta = w3 * inv_w_tot;

            if (in_01 && in_12 && in_23 && in_30) {
                float light_intensity = quad.light_intensity[0]*alpha + quad.light_intensity[1]*beta + quad.light_intensity[2]*gamma + quad.light_intensity[3]*delta;

                float rf = r * light_intensity;
                float gf = g * light_intensity;
                float bf = b * light_intensity;
                uint8_t r8 = (uint8_t)fmaxf(0, fminf(255, rf));
                uint8_t g8 = (uint8_t)fmaxf(0, fminf(255, gf));
                uint8_t b8 = (uint8_t)fmaxf(0, fminf(255, bf));

                double inv_w = alpha * (1.0f / p0.w) + beta  * (1.0f / p1.w) + gamma * (1.0f / p2.w) + delta * (1.0f / p3.w);
                double z_over_w = alpha * (p0.z / p0.w) + beta  * (p1.z / p1.w) + gamma * (p2.z / p2.w) + delta * (p3.z / p3.w);
                double inv_z = inv_w / z_over_w;

                if (inv_z >= MAT2D_AT(inv_z_buffer, y, x)) {
                    adl_point_draw(screen_mat, x, y, RGBA_hexARGB(r8, g8, b8, a), offset_zoom_param);
                    MAT2D_AT(inv_z_buffer, y, x) = inv_z;
                }
            }
        }
    }
}

void adl_quad_fill_interpolate_color_mean_value(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer, Quad quad, Offset_zoom_param offset_zoom_param)
{
    Point p0 = quad.points[0];
```

```
    Point p1 = quad.points[1];
    Point p2 = quad.points[2];
    Point p3 = quad.points[3];

    int x_min = fminf(p0.x, fminf(p1.x, fminf(p2.x, p3.x)));
    int x_max = fmaxf(p0.x, fmaxf(p1.x, fmaxf(p2.x, p3.x)));
    int y_min = fminf(p0.y, fminf(p1.y, fminf(p2.y, p3.y)));
    int y_max = fmaxf(p0.y, fmaxf(p1.y, fmaxf(p2.y, p3.y)));

    if (x_min < 0) x_min = 0;
    if (y_min < 0) y_min = 0;
    if (x_max >= (int)screen_mat.cols) x_max = (int)screen_mat.cols - 1;
    if (y_max >= (int)screen_mat.rows) y_max = (int)screen_mat.rows - 1;

    float w = edge_cross_point(p0, p1, p1, p2) + edge_cross_point(p2, p3, p3, p0);
    if (fabs(w) < 1e-6) {
        // adl_quad_draw(screen_mat, inv_z_buffer, quad, quad.colors[0], offset_zoom_param);
        return;
    }

    for (int y = y_min; y <= y_max; y++) {
        for (int x = x_min; x <= x_max; x++) {
            Point p = {.x = x, .y = y, .z = 0};
            bool in_01, in_12, in_23, in_30;

            in_01 = (edge_cross_point(p0, p1, p0, p) >= 0) != (w < 0);
            in_12 = (edge_cross_point(p1, p2, p1, p) >= 0) != (w < 0);
            in_23 = (edge_cross_point(p2, p3, p2, p) >= 0) != (w < 0);
            in_30 = (edge_cross_point(p3, p0, p3, p) >= 0) != (w < 0);

            /* using 'mean value coordinates'
             * https://www.mn.uio.no/math/english/people/aca/michaelf/papers/mv3d.pdf. */
            float size_p_to_p0 = sqrt((p0.x - p.x)*(p0.x - p.x) + (p0.y - p.y)*(p0.y - p.y));
            float size_p_to_p1 = sqrt((p1.x - p.x)*(p1.x - p.x) + (p1.y - p.y)*(p1.y - p.y));
            float size_p_to_p2 = sqrt((p2.x - p.x)*(p2.x - p.x) + (p2.y - p.y)*(p2.y - p.y));
            float size_p_to_p3 = sqrt((p3.x - p.x)*(p3.x - p.x) + (p3.y - p.y)*(p3.y - p.y));

            /* calculating the tangent of half the angle directly using vector math */
            float t0 = adl_tan_half_angle(p0, p1, p, size_p_to_p0, size_p_to_p1);
            float t1 = adl_tan_half_angle(p1, p2, p, size_p_to_p1, size_p_to_p2);
            float t2 = adl_tan_half_angle(p2, p3, p, size_p_to_p2, size_p_to_p3);
            float t3 = adl_tan_half_angle(p3, p0, p, size_p_to_p3, size_p_to_p0);

            float w0 = (t3 + t0) / size_p_to_p0;
            float w1 = (t0 + t1) / size_p_to_p1;
            float w2 = (t1 + t2) / size_p_to_p2;
            float w3 = (t2 + t3) / size_p_to_p3;

            float inv_w_tot = 1.0f / (w0 + w1 + w2 + w3);
            float alpha = w0 * inv_w_tot;
            float beta  = w1 * inv_w_tot;
            float gamma = w2 * inv_w_tot;
            float delta = w3 * inv_w_tot;

            if (in_01 && in_12 && in_23 && in_30) {
                int r0, g0, b0, a0;
                int r1, g1, b1, a1;
                int r2, g2, b2, a2;
                int r3, g3, b3, a3;
                HexARGB_RGBA_VAR(quad.colors[0], r0, g0, b0, a0);
                HexARGB_RGBA_VAR(quad.colors[1], r1, g1, b1, a1);
                HexARGB_RGBA_VAR(quad.colors[2], r2, g2, b2, a2);
                HexARGB_RGBA_VAR(quad.colors[3], r3, g3, b3, a3);

                uint8_t current_r = r0*alpha + r1*beta + r2*gamma + r3*delta;
                uint8_t current_g = g0*alpha + g1*beta + g2*gamma + g3*delta;
                uint8_t current_b = b0*alpha + b1*beta + b2*gamma + b3*delta;
                uint8_t current_a = a0*alpha + a1*beta + a2*gamma + a3*delta;

                float light_intensity = (quad.light_intensity[0] + quad.light_intensity[1] +
quad.light_intensity[2] + quad.light_intensity[3]) / 4;
                float rf = current_r * light_intensity;
                float gf = current_g * light_intensity;
                float bf = current_b * light_intensity;
                uint8_t r8 = (uint8_t)fmaxf(0, fminf(255, rf));
                uint8_t g8 = (uint8_t)fmaxf(0, fminf(255, gf));
                uint8_t b8 = (uint8_t)fmaxf(0, fminf(255, bf));

                double inv_w = alpha * (1.0f / p0.w) + beta  * (1.0f / p1.w) + gamma * (1.0f / p2.w) +
delta * (1.0f / p3.w);
                double z_over_w = alpha * (p0.z / p0.w) + beta  * (p1.z / p1.w) + gamma * (p2.z /
p2.w) + delta * (p3.z / p3.w);
                double inv_z = inv_w / z_over_w;

                if (inv_z >= MAT2D_AT(inv_z_buffer, y, x)) {
                    adl_point_draw(screen_mat, x, y, RGBA_hexARGB(r8, g8, b8, current_a),
offset_zoom_param);
```

```
                    MAT2D_AT(inv_z_buffer, y, x) = inv_z;
                }
            }
        }
    }
}

void adl_quad_mesh_draw(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer_mat, Quad_mesh mesh, uint32_t color, Offset_zoom_param offset_zoom_param)
{
    for (size_t i = 0; i < mesh.length; i++) {
        Quad quad = mesh.elements[i];
        /* Reject invalid quad */
        adl_assert_quad_is_valid(quad);

        if (!quad.to_draw) continue;

        adl_quad_draw(screen_mat, inv_z_buffer_mat, quad, color, offset_zoom_param);
    }
}

void adl_quad_mesh_fill(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer_mat, Quad_mesh mesh, uint32_t color, Offset_zoom_param offset_zoom_param)
{
    for (size_t i = 0; i < mesh.length; i++) {
        Quad quad = mesh.elements[i];
        /* Reject invalid quad */
        adl_assert_quad_is_valid(quad);

        if (!quad.to_draw) continue;

        // color = rand_double() * 0xFFFFFFFF;

        adl_quad_fill(screen_mat, inv_z_buffer_mat, quad, color, offset_zoom_param);
    }
}

void adl_quad_mesh_fill_interpolate_normal(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer_mat, Quad_mesh mesh, uint32_t color, Offset_zoom_param offset_zoom_param)
{
    for (size_t i = 0; i < mesh.length; i++) {
        Quad quad = mesh.elements[i];
        /* Reject invalid quad */
        adl_assert_quad_is_valid(quad);

        uint8_t a, r, g, b;
        HexARGB_RGBA_VAR(color, a, r, g, b);
        (void)r;
        (void)g;
        (void)b;

        if (!quad.to_draw && a == 255) continue;

        adl_quad_fill_interpolate_normal_mean_value(screen_mat, inv_z_buffer_mat, quad, color, offset_zoom_param);
    }
}

void adl_quad_mesh_fill_interpolate_color(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer_mat, Quad_mesh mesh, Offset_zoom_param offset_zoom_param)
{
    for (size_t i = 0; i < mesh.length; i++) {
        Quad quad = mesh.elements[i];
        /* Reject invalid quad */
        adl_assert_quad_is_valid(quad);

        if (!quad.to_draw) continue;

        adl_quad_fill_interpolate_color_mean_value(screen_mat, inv_z_buffer_mat, quad, offset_zoom_param);
    }
}

void adl_circle_draw(Mat2D_uint32 screen_mat, float center_x, float center_y, float r, uint32_t color, Offset_zoom_param offset_zoom_param)
{
    for (int dy = -r; dy <= r; dy++) {
        for (int dx = -r; dx <= r; dx ++) {
            float diff = dx * dx + dy * dy - r * r;
            if (diff < 0 && diff > -r*2) {
                adl_point_draw(screen_mat, center_x + dx, center_y + dy, color, offset_zoom_param);
            }
        }
    }
}

void adl_circle_fill(Mat2D_uint32 screen_mat, float center_x, float center_y, float r, uint32_t color,
```

```
Offset_zoom_param offset_zoom_param)
{
    for (int dy = -r; dy <= r; dy++) {
        for (int dx = -r; dx <= r; dx ++) {
            float diff = dx * dx + dy * dy - r * r;
            if (diff < 0) {
                adl_point_draw(screen_mat, center_x + dx, center_y + dy, color, offset_zoom_param);
            }
        }
    }
}

void adl_tri_draw(Mat2D_uint32 screen_mat, Tri tri, uint32_t color, Offset_zoom_param offset_zoom_param)
{
    adl_line_draw(screen_mat, tri.points[0].x, tri.points[0].y, tri.points[1].x, tri.points[1].y, color, offset_zoom_param);
    adl_line_draw(screen_mat, tri.points[1].x, tri.points[1].y, tri.points[2].x, tri.points[2].y, color, offset_zoom_param);
    adl_line_draw(screen_mat, tri.points[2].x, tri.points[2].y, tri.points[0].x, tri.points[0].y, color, offset_zoom_param);

    // adl_draw_arrow(screen_mat, tri.points[0].x, tri.points[0].y, tri.points[1].x, tri.points[1].y, 0.3, 22, color);
    // adl_draw_arrow(screen_mat, tri.points[1].x, tri.points[1].y, tri.points[2].x, tri.points[2].y, 0.3, 22, color);
    // adl_draw_arrow(screen_mat, tri.points[2].x, tri.points[2].y, tri.points[0].x, tri.points[0].y, 0.3, 22, color);
}

void adl_tri_fill_Pinedas_rasterizer(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer, Tri tri, uint32_t color, Offset_zoom_param offset_zoom_param)
{
    /* This function follows the rasterizer of 'Pikuma' shown in his YouTube video. You can fine the video in this link: https://youtu.be/k5wtuKWmV48. */

    Point p0, p1, p2;
    p0 = tri.points[0];
    p1 = tri.points[1];
    p2 = tri.points[2];

    /* finding bounding box */
    int x_min = fmin(p0.x, fmin(p1.x, p2.x));
    int x_max = fmax(p0.x, fmax(p1.x, p2.x));
    int y_min = fmin(p0.y, fmin(p1.y, p2.y));
    int y_max = fmax(p0.y, fmax(p1.y, p2.y));

    /* Clamp to screen bounds */
    if (x_min < 0) x_min = 0;
    if (y_min < 0) y_min = 0;
    if (x_max >= (int)screen_mat.cols) x_max = screen_mat.cols - 1;
    if (y_max >= (int)screen_mat.rows) y_max = screen_mat.rows - 1;

    /* draw only outline of the tri if there is no area */
    float w = edge_cross_point(p0, p1, p1, p2);
    if (fabsf(w) < 1e-6) {
        // adl_tri_draw(screen_mat, tri, tri.colors[0], offset_zoom_param);
        return;
    }
    MATRIX2D_ASSERT(fabsf(w) > 1e-6 && "triangle must have area");

    /* fill conventions */
    int bias0 = is_top_left(p0, p1) ? 0 : -1;
    int bias1 = is_top_left(p1, p2) ? 0 : -1;
    int bias2 = is_top_left(p2, p0) ? 0 : -1;

    for (int y = y_min; y <= y_max; y++) {
        for (int x = x_min; x <= x_max; x++) {
            Point p = {.x = x, .y = y, .z = 0};

            float w0 = edge_cross_point(p0, p1, p0, p) + bias0;
            float w1 = edge_cross_point(p1, p2, p1, p) + bias1;
            float w2 = edge_cross_point(p2, p0, p2, p) + bias2;

            float alpha = fabs(w1 / w);
            float beta  = fabs(w2 / w);
            float gamma = fabs(w0 / w);

            if (w0 * w >= 0 && w1 * w >= 0 &&  w2 * w >= 0) {
                int r, b, g, a;
                HexARGB_RGBA_VAR(color, r, g, b, a);
                float light_intensity = (tri.light_intensity[0] + tri.light_intensity[1] + tri.light_intensity[2]) / 3;
                float rf = r * light_intensity;
                float gf = g * light_intensity;
                float bf = b * light_intensity;
                uint8_t r8 = (uint8_t)fmaxf(0, fminf(255, rf));
```

```
                uint8_t g8 = (uint8_t)fmaxf(0, fminf(255, gf));
                uint8_t b8 = (uint8_t)fmaxf(0, fminf(255, bf));

                double inv_w = alpha * (1.0 / p0.w) + beta  * (1.0 / p1.w) + gamma * (1.0 / p2.w);
                double z_over_w = alpha * (p0.z / p0.w) + beta  * (p1.z / p1.w) + gamma * (p2.z / p2.w);
                double inv_z = inv_w / z_over_w;

                if (inv_z >= MAT2D_AT(inv_z_buffer, y, x)) {
                    adl_point_draw(screen_mat, x, y, RGBA_hexARGB(r8, g8, b8, a), offset_zoom_param);
                    MAT2D_AT(inv_z_buffer, y, x) = inv_z;
                }
            }
        }
    }
}

void adl_tri_fill_Pinedas_rasterizer_interpolate_color(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer, Tri tri, Offset_zoom_param offset_zoom_param)
{
    /* This function follows the rasterizer of 'Pikuma' shown in his YouTube video. You can fine the video in this link: https://youtu.be/k5wtuKWmV48. */
    Point p0, p1, p2;
    p0 = tri.points[0];
    p1 = tri.points[1];
    p2 = tri.points[2];

    float w = edge_cross_point(p0, p1, p1, p2);
    if (fabsf(w) < 1e-6) {
        // adl_tri_draw(screen_mat, tri, tri.colors[0], offset_zoom_param);
        return;
    }
    MATRIX2D_ASSERT(w != 0 && "triangle has area");

    /* fill conventions */
    int bias0 = is_top_left(p0, p1) ? 0 : -1;
    int bias1 = is_top_left(p1, p2) ? 0 : -1;
    int bias2 = is_top_left(p2, p0) ? 0 : -1;

    /* finding bounding box */
    int x_min = fmin(p0.x, fmin(p1.x, p2.x));
    int x_max = fmax(p0.x, fmax(p1.x, p2.x));
    int y_min = fmin(p0.y, fmin(p1.y, p2.y));
    int y_max = fmax(p0.y, fmax(p1.y, p2.y));
    // printf("xmin: %d, xmax: %d || ymin: %d, ymax: %d\n", x_min, x_max, y_min, y_max);

    /* Clamp to screen bounds */
    if (x_min < 0) x_min = 0;
    if (y_min < 0) y_min = 0;
    if (x_max >= (int)screen_mat.cols) x_max = screen_mat.cols - 1;
    if (y_max >= (int)screen_mat.rows) y_max = screen_mat.rows - 1;

    for (int y = y_min; y <= y_max; y++) {
        for (int x = x_min; x <= x_max; x++) {
            Point p = {.x = x, .y = y, .z = 0};

            float w0 = edge_cross_point(p0, p1, p0, p) + bias0;
            float w1 = edge_cross_point(p1, p2, p1, p) + bias1;
            float w2 = edge_cross_point(p2, p0, p2, p) + bias2;

            float alpha = fabs(w1 / w);
            float beta  = fabs(w2 / w);
            float gamma = fabs(w0 / w);

            if (w0 * w >= 0 && w1 * w >= 0 &&  w2 * w >= 0) {
                int r0, b0, g0, a0;
                int r1, b1, g1, a1;
                int r2, b2, g2, a2;
                HexARGB_RGBA_VAR(tri.colors[0], r0, g0, b0, a0);
                HexARGB_RGBA_VAR(tri.colors[1], r1, g1, b1, a1);
                HexARGB_RGBA_VAR(tri.colors[2], r2, g2, b2, a2);

                uint8_t current_r = r0*alpha + r1*beta + r2*gamma;
                uint8_t current_g = g0*alpha + g1*beta + g2*gamma;
                uint8_t current_b = b0*alpha + b1*beta + b2*gamma;
                uint8_t current_a = a0*alpha + a1*beta + a2*gamma;

                float light_intensity = (tri.light_intensity[0] + tri.light_intensity[1] + tri.light_intensity[2]) / 3;
                float rf = current_r * light_intensity;
                float gf = current_g * light_intensity;
                float bf = current_b * light_intensity;
                uint8_t r8 = (uint8_t)fmaxf(0, fminf(255, rf));
                uint8_t g8 = (uint8_t)fmaxf(0, fminf(255, gf));
                uint8_t b8 = (uint8_t)fmaxf(0, fminf(255, bf));

                double inv_w = alpha * (1.0 / p0.w) + beta  * (1.0 / p1.w) + gamma * (1.0 / p2.w);
```

```
                double z_over_w = alpha * (p0.z / p0.w) + beta  * (p1.z / p1.w) + gamma * (p2.z / p2.w);
                double inv_z = inv_w / z_over_w;

                if (inv_z >= MAT2D_AT(inv_z_buffer, y, x)) {
                    adl_point_draw(screen_mat, x, y, RGBA_hexARGB(r8, g8, b8, current_a), offset_zoom_param);
                    MAT2D_AT(inv_z_buffer, y, x) = inv_z;
                }
            }
        }
    }
}

void adl_tri_fill_Pinedas_rasterizer_interpolate_normal(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer, Tri tri, uint32_t color, Offset_zoom_param offset_zoom_param)
{
    /* This function follows the rasterizer of 'Pikuma' shown in his YouTube video. You can fine the video in this link: https://youtu.be/k5wtuKWmV48. */
    Point p0, p1, p2;
    p0 = tri.points[0];
    p1 = tri.points[1];
    p2 = tri.points[2];

    float w = edge_cross_point(p0, p1, p1, p2);
    if (fabsf(w) < 1e-6) {
        // adl_tri_draw(screen_mat, tri, tri.colors[0], offset_zoom_param);
        return;
    }
    MATRIX2D_ASSERT(w != 0 && "triangle has area");

    /* fill conventions */
    int bias0 = is_top_left(p0, p1) ? 0 : -1;
    int bias1 = is_top_left(p1, p2) ? 0 : -1;
    int bias2 = is_top_left(p2, p0) ? 0 : -1;

    /* finding bounding box */
    int x_min = fmin(p0.x, fmin(p1.x, p2.x));
    int x_max = fmax(p0.x, fmax(p1.x, p2.x));
    int y_min = fmin(p0.y, fmin(p1.y, p2.y));
    int y_max = fmax(p0.y, fmax(p1.y, p2.y));
    // printf("xmin: %d, xmax: %d || ymin: %d, ymax: %d\n", x_min, x_max, y_min, y_max);

    /* Clamp to screen bounds */
    if (x_min < 0) x_min = 0;
    if (y_min < 0) y_min = 0;
    if (x_max >= (int)screen_mat.cols) x_max = screen_mat.cols - 1;
    if (y_max >= (int)screen_mat.rows) y_max = screen_mat.rows - 1;

    int r, b, g, a;
    HexARGB_RGBA_VAR(color, r, g, b, a);

    for (int y = y_min; y <= y_max; y++) {
        for (int x = x_min; x <= x_max; x++) {
            Point p = {.x = x, .y = y, .z = 0};

            float w0 = edge_cross_point(p0, p1, p0, p) + bias0;
            float w1 = edge_cross_point(p1, p2, p1, p) + bias1;
            float w2 = edge_cross_point(p2, p0, p2, p) + bias2;

            float alpha = fabs(w1 / w);
            float beta  = fabs(w2 / w);
            float gamma = fabs(w0 / w);

            if (w0 * w >= 0 && w1 * w >= 0 &&  w2 * w >= 0) {

                float light_intensity = tri.light_intensity[0]*alpha + tri.light_intensity[1]*beta + tri.light_intensity[2]*gamma;

                float rf = r * light_intensity;
                float gf = g * light_intensity;
                float bf = b * light_intensity;
                uint8_t r8 = (uint8_t)fmaxf(0, fminf(255, rf));
                uint8_t g8 = (uint8_t)fmaxf(0, fminf(255, gf));
                uint8_t b8 = (uint8_t)fmaxf(0, fminf(255, bf));

                double inv_w = alpha * (1.0 / p0.w) + beta  * (1.0 / p1.w) + gamma * (1.0 / p2.w);
                double z_over_w = alpha * (p0.z / p0.w) + beta  * (p1.z / p1.w) + gamma * (p2.z / p2.w);
                double inv_z = inv_w / z_over_w;

                if (inv_z >= MAT2D_AT(inv_z_buffer, y, x)) {
                    adl_point_draw(screen_mat, x, y, RGBA_hexARGB(r8, g8, b8, a), offset_zoom_param);
                    MAT2D_AT(inv_z_buffer, y, x) = inv_z;
                }
            }
        }
```

```
    }
}

void adl_tri_mesh_draw(Mat2D_uint32 screen_mat, Tri_mesh mesh, uint32_t color, Offset_zoom_param offset_zoom_param)
{
    for (size_t i = 0; i < mesh.length; i++) {
        Tri tri = mesh.elements[i];
        if (tri.to_draw) {
            // color = rand_double() * 0xFFFFFFFF;
            adl_tri_draw(screen_mat, tri, color, offset_zoom_param);
        }
    }
}

void adl_tri_mesh_fill_Pinedas_rasterizer(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer_mat, Tri_mesh mesh, uint32_t color, Offset_zoom_param offset_zoom_param)
{
    for (size_t i = 0; i < mesh.length; i++) {
        Tri tri = mesh.elements[i];
        /* Reject invalid triangles */
        adl_assert_tri_is_valid(tri);

        if (!tri.to_draw) continue;

        adl_tri_fill_Pinedas_rasterizer(screen_mat, inv_z_buffer_mat, tri, color, offset_zoom_param);
    }
}

void adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_color(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer_mat, Tri_mesh mesh, Offset_zoom_param offset_zoom_param)
{
    for (size_t i = 0; i < mesh.length; i++) {
        Tri tri = mesh.elements[i];
        /* Reject invalid triangles */
        adl_assert_tri_is_valid(tri);

        if (!tri.to_draw) continue;

        adl_tri_fill_Pinedas_rasterizer_interpolate_color(screen_mat, inv_z_buffer_mat, tri, offset_zoom_param);
    }
}

void adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_normal(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer_mat, Tri_mesh mesh, uint32_t color, Offset_zoom_param offset_zoom_param)
{
    for (size_t i = 0; i < mesh.length; i++) {
        Tri tri = mesh.elements[i];
        /* Reject invalid triangles */
        adl_assert_tri_is_valid(tri);

        if (!tri.to_draw) continue;

        adl_tri_fill_Pinedas_rasterizer_interpolate_normal(screen_mat, inv_z_buffer_mat, tri, color, offset_zoom_param);
    }
}

float adl_tan_half_angle(Point vi, Point vj, Point p, float li, float lj)
{
    float ax = vi.x - p.x, ay = vi.y - p.y;
    float bx = vj.x - p.x, by = vj.y - p.y;
    float dot = ax * bx + ay * by;
    float cross = ax * by - ay * bx;               // signed 2D cross (scalar)
    float denom = dot + li * lj;                    // = |a||b|(1 + cos(alpha))
    return fabsf(cross) / fmaxf(1e-20f, denom);     // tan(alpha/2)
}

float adl_linear_map(float s, float min_in, float max_in, float min_out, float max_out)
{
    return (min_out + ((s-min_in)*(max_out-min_out))/(max_in-min_in));
}

void adl_quad2tris(Quad quad, Tri *tri1, Tri *tri2, char split_line[])
{
    if (!strncmp(split_line, "02", 2)) {
        tri1->points[0] = quad.points[0];
        tri1->points[1] = quad.points[1];
        tri1->points[2] = quad.points[2];
        tri1->to_draw = quad.to_draw;
        tri1->light_intensity[0] = quad.light_intensity[0];
        tri1->light_intensity[1] = quad.light_intensity[1];
        tri1->light_intensity[2] = quad.light_intensity[2];
        tri1->colors[0] = quad.colors[0];
        tri1->colors[1] = quad.colors[1];
        tri1->colors[2] = quad.colors[2];
```

```
        tri2->points[0] = quad.points[2];
        tri2->points[1] = quad.points[3];
        tri2->points[2] = quad.points[0];
        tri2->to_draw = quad.to_draw;
        tri1->light_intensity[0] = quad.light_intensity[2];
        tri1->light_intensity[1] = quad.light_intensity[3];
        tri1->light_intensity[2] = quad.light_intensity[0];
        tri2->colors[0] = quad.colors[2];
        tri2->colors[1] = quad.colors[3];
        tri2->colors[2] = quad.colors[0];
    } else if (!strncmp(split_line, "13", 2)) {
        tri1->points[0] = quad.points[1];
        tri1->points[1] = quad.points[2];
        tri1->points[2] = quad.points[3];
        tri1->to_draw = quad.to_draw;
        tri1->light_intensity[0] = quad.light_intensity[1];
        tri1->light_intensity[1] = quad.light_intensity[2];
        tri1->light_intensity[2] = quad.light_intensity[3];
        tri1->colors[0] = quad.colors[1];
        tri1->colors[1] = quad.colors[2];
        tri1->colors[2] = quad.colors[3];

        tri2->points[0] = quad.points[3];
        tri2->points[1] = quad.points[0];
        tri2->points[2] = quad.points[1];
        tri2->to_draw = quad.to_draw;
        tri1->light_intensity[0] = quad.light_intensity[3];
        tri1->light_intensity[1] = quad.light_intensity[0];
        tri1->light_intensity[2] = quad.light_intensity[1];
        tri2->colors[0] = quad.colors[3];
        tri2->colors[1] = quad.colors[0];
        tri2->colors[2] = quad.colors[1];
    }
}

void adl_linear_sRGB_to_okLab(uint32_t hex_ARGB, float *L, float *a, float *b)
{
    /* https://bottosson.github.io/posts/oklab/
       https://en.wikipedia.org/wiki/Oklab_color_space */
    int R_255, G_255, B_255;
    HexARGB_RGB_VAR(hex_ARGB, R_255, G_255, B_255);

    float R = R_255;
    float G = G_255;
    float B = B_255;

    float l = 0.4122214705f * R + 0.5363325363f * G + 0.0514459929f * B;
    float m = 0.2119034982f * R + 0.6806995451f * G + 0.1073969566f * B;
    float s = 0.0883024619f * R + 0.2817188376f * G + 0.6299787005f * B;

    float l_ = cbrtf(l);
    float m_ = cbrtf(m);
    float s_ = cbrtf(s);

    *L = 0.2104542553f * l_ + 0.7936177850f * m_ - 0.0040720468f * s_;
    *a = 1.9779984951f * l_ - 2.4285922050f * m_ + 0.4505937099f * s_;
    *b = 0.0259040371f * l_ + 0.7827717662f * m_ - 0.8086757660f * s_;

}

void adl_okLab_to_linear_sRGB(float L, float a, float b, uint32_t *hex_ARGB)
{
    /* https://bottosson.github.io/posts/oklab/
       https://en.wikipedia.org/wiki/Oklab_color_space */

    float l_ = L + 0.3963377774f * a + 0.2158037573f * b;
    float m_ = L - 0.1055613458f * a - 0.0638541728f * b;
    float s_ = L - 0.0894841775f * a - 1.2914855480f * b;

    float l = l_ * l_ * l_;
    float m = m_ * m_ * m_;
    float s = s_ * s_ * s_;

    float R = + 4.0767416621f * l - 3.3077115913f * m + 0.2309699292f * s;
    float G = - 1.2684380046f * l + 2.6097574011f * m - 0.3413193965f * s;
    float B = - 0.0041960863f * l - 0.7034186147f * m + 1.7076147010f * s;

    R = fmaxf(fminf(R, 255), 0);
    G = fmaxf(fminf(G, 255), 0);
    B = fmaxf(fminf(B, 255), 0);

    *hex_ARGB = RGBA_hexARGB(R, G, B, 0xFF);
}

void adl_linear_sRGB_to_okLch(uint32_t hex_ARGB, float *L, float *c, float *h_deg)
{
```

```
    float a, b;
    adl_linear_sRGB_to_okLab(hex_ARGB, L, &a, &b);

    *c = sqrtf(a * a + b * b);
    *h_deg = atan2f(b, a) * 180 / PI;
}

void adl_okLch_to_linear_sRGB(float L, float c, float h_deg, uint32_t *hex_ARGB)
{
    h_deg = fmodf((h_deg + 360), 360);
    float a = c * cosf(h_deg * PI / 180);
    float b = c * sinf(h_deg * PI / 180);
    adl_okLab_to_linear_sRGB(L, a, b, hex_ARGB);
}

void adl_interpolate_ARGBcolor_on_okLch(uint32_t color1, uint32_t color2, float t, float num_of_rotations, uint32_t *color_out)
{
    float L_1, c_1, h_1;
    float L_2, c_2, h_2;
    adl_linear_sRGB_to_okLch(color1, &L_1, &c_1, &h_1);
    adl_linear_sRGB_to_okLch(color2, &L_2, &c_2, &h_2);
    h_2 = h_2 + 360 * num_of_rotations;

    float L, c, h;
    L = L_1 * (1 - t) + L_2 * (t);
    c = c_1 * (1 - t) + c_2 * (t);
    h = h_1 * (1 - t) + h_2 * (t);
    adl_okLch_to_linear_sRGB(L, c, h, color_out);
}

Figure adl_figure_alloc(size_t rows, size_t cols, Point top_left_position)
{
    ADL_ASSERT(rows && cols);
    adl_assert_point_is_valid(top_left_position);

    Figure figure = {0};
    figure.pixels_mat = mat2D_alloc_uint32(rows, cols);
    figure.inv_z_buffer_mat = mat2D_alloc(rows, cols);
    memset(figure.inv_z_buffer_mat.elements, 0x0, sizeof(double) * figure.inv_z_buffer_mat.rows * figure.inv_z_buffer_mat.cols);
    ada_init_array(Curve, figure.src_curve_array);

    figure.top_left_position = top_left_position;

    int max_i    = (int)(figure.pixels_mat.rows);
    int max_j    = (int)(figure.pixels_mat.cols);
    int offset_i = (int)fminf(figure.pixels_mat.rows * ADL_FIGURE_PADDING_PRECENTAGE / 100.0f, ADL_MAX_FIGURE_PADDING);
    int offset_j = (int)fminf(figure.pixels_mat.cols * ADL_FIGURE_PADDING_PRECENTAGE / 100.0f, ADL_MAX_FIGURE_PADDING);

    figure.min_x_pixel = offset_j;
    figure.max_x_pixel = max_j - offset_j;
    figure.min_y_pixel = offset_i;
    figure.max_y_pixel = max_i - offset_i;

    figure.min_x = + FLT_MAX;
    figure.max_x = - FLT_MAX;
    figure.min_y = + FLT_MAX;
    figure.max_y = - FLT_MAX;

    figure.offset_zoom_param = ADL_DEFAULT_OFFSET_ZOOM;

    return figure;
}

void adl_figure_copy_to_screen(Mat2D_uint32 screen_mat, Figure figure)
{
    for (size_t i = 0; i < figure.pixels_mat.rows; i++) {
        for (size_t j = 0; j < figure.pixels_mat.cols; j++) {
            int offset_i = figure.top_left_position.y;
            int offset_j = figure.top_left_position.x;

            adl_point_draw(screen_mat, offset_j+j, offset_i+i, MAT2D_AT_UINT32(figure.pixels_mat, i, j), (Offset_zoom_param){1,0,0,0,0});
        }
    }
}

void adl_axis_draw_on_figure(Figure *figure)
{
    int max_i    = (int)(figure->pixels_mat.rows);
    int max_j    = (int)(figure->pixels_mat.cols);
    int offset_i = (int)fmaxf(fminf(figure->pixels_mat.rows * ADL_FIGURE_PADDING_PRECENTAGE / 100.0f, ADL_MAX_FIGURE_PADDING), ADL_MIN_FIGURE_PADDING);
    int offset_j = (int)fmaxf(fminf(figure->pixels_mat.cols * ADL_FIGURE_PADDING_PRECENTAGE / 100.0f,
```

```
ADL_MAX_FIGURE_PADDING), ADL_MIN_FIGURE_PADDING);

    int arrow_head_size_x = (int)fminf(ADL_MAX_HEAD_SIZE, ADL_FIGURE_PADDING_PRECENTAGE / 100.0f *
(max_j - 2 * offset_j));
    int arrow_head_size_y = (int)fminf(ADL_MAX_HEAD_SIZE, ADL_FIGURE_PADDING_PRECENTAGE / 100.0f *
(max_i - 2 * offset_i));

    adl_arrow_draw(figure->pixels_mat, figure->min_x_pixel, figure->max_y_pixel, figure->max_x_pixel,
figure->max_y_pixel, (float)arrow_head_size_x / (max_j-2*offset_j), ADL_FIGURE_HEAD_ANGLE_DEG,
ADL_FIGURE_AXIS_COLOR, figure->offset_zoom_param);
    adl_arrow_draw(figure->pixels_mat, figure->min_x_pixel, figure->max_y_pixel, figure->min_x_pixel,
figure->min_y_pixel, (float)arrow_head_size_y / (max_i-2*offset_i), ADL_FIGURE_HEAD_ANGLE_DEG,
ADL_FIGURE_AXIS_COLOR, figure->offset_zoom_param);
    // adl_draw_rectangle_min_max(figure->pixels_mat, figure->min_x_pixel, figure->max_x_pixel,
figure->min_y_pixel, figure->max_y_pixel, 0);

    figure->x_axis_head_size = arrow_head_size_x;
    figure->y_axis_head_size = arrow_head_size_y;
}

void adl_max_min_values_draw_on_figure(Figure figure)
{
    char x_min_sentence[256];
    char x_max_sentence[256];
    snprintf(x_min_sentence, 256, "%g", figure.min_x);
    snprintf(x_max_sentence, 256, "%g", figure.max_x);

    int x_sentence_hight_pixel = (figure.pixels_mat.rows - figure.max_y_pixel -
ADL_MIN_CHARACTER_OFFSET * 3);
    int x_min_char_width_pixel = x_sentence_hight_pixel / 2;
    int x_max_char_width_pixel = x_sentence_hight_pixel / 2;

    int x_min_sentence_width_pixel = (int)fminf((figure.max_x_pixel - figure.min_x_pixel)/2,
(x_min_char_width_pixel + ADL_MAX_CHARACTER_OFFSET)*strlen(x_min_sentence));
    x_min_char_width_pixel = x_min_sentence_width_pixel / strlen(x_min_sentence) -
ADL_MIN_CHARACTER_OFFSET;

    int x_max_sentence_width_pixel = (int)fminf((figure.max_x_pixel - figure.min_x_pixel)/2,
(x_max_char_width_pixel + ADL_MAX_CHARACTER_OFFSET)*strlen(x_max_sentence)) -
figure.x_axis_head_size;
    x_max_char_width_pixel = (x_max_sentence_width_pixel + figure.x_axis_head_size) /
strlen(x_max_sentence) - ADL_MIN_CHARACTER_OFFSET;

    int x_min_sentence_hight_pixel = (int)fminf(x_min_char_width_pixel * 2, x_sentence_hight_pixel);
    int x_max_sentence_hight_pixel = (int)fminf(x_max_char_width_pixel * 2, x_sentence_hight_pixel);

    x_min_sentence_hight_pixel = (int)fminf(x_min_sentence_hight_pixel, x_max_sentence_hight_pixel);
    x_max_sentence_hight_pixel = x_min_sentence_hight_pixel;

    int x_max_x_top_left = figure.max_x_pixel - strlen(x_max_sentence) * (x_max_sentence_hight_pixel /
2 + ADL_MIN_CHARACTER_OFFSET) - figure.x_axis_head_size;

    adl_sentence_draw(figure.pixels_mat, x_min_sentence, strlen(x_min_sentence), figure.min_x_pixel,
figure.max_y_pixel+ADL_MIN_CHARACTER_OFFSET*2, x_min_sentence_hight_pixel, ADL_FIGURE_AXIS_COLOR,
figure.offset_zoom_param);
    adl_sentence_draw(figure.pixels_mat, x_max_sentence, strlen(x_max_sentence), x_max_x_top_left,
figure.max_y_pixel+ADL_MIN_CHARACTER_OFFSET*2, x_max_sentence_hight_pixel, ADL_FIGURE_AXIS_COLOR,
figure.offset_zoom_param);

    char y_min_sentence[256];
    char y_max_sentence[256];
    snprintf(y_min_sentence, 256, "%g", figure.min_y);
    snprintf(y_max_sentence, 256, "%g", figure.max_y);

    int y_sentence_width_pixel = figure.min_x_pixel - ADL_MAX_CHARACTER_OFFSET -
figure.y_axis_head_size;
    int y_max_char_width_pixel = y_sentence_width_pixel;
    y_max_char_width_pixel /= strlen(y_max_sentence);
    int y_max_sentence_hight_pixel = y_max_char_width_pixel * 2;

    int y_min_char_width_pixel = y_sentence_width_pixel;
    y_min_char_width_pixel /= strlen(y_min_sentence);
    int y_min_sentence_hight_pixel = y_min_char_width_pixel * 2;

    y_min_sentence_hight_pixel = (int)fmaxf(fminf(y_min_sentence_hight_pixel,
y_max_sentence_hight_pixel), 1);
    y_max_sentence_hight_pixel = y_min_sentence_hight_pixel;

    adl_sentence_draw(figure.pixels_mat, y_max_sentence, strlen(y_max_sentence),
ADL_MAX_CHARACTER_OFFSET/2, figure.min_y_pixel, y_max_sentence_hight_pixel, ADL_FIGURE_AXIS_COLOR,
figure.offset_zoom_param);
    adl_sentence_draw(figure.pixels_mat, y_min_sentence, strlen(y_min_sentence),
ADL_MAX_CHARACTER_OFFSET/2, figure.max_y_pixel-y_min_sentence_hight_pixel,
y_min_sentence_hight_pixel, ADL_FIGURE_AXIS_COLOR, figure.offset_zoom_param);
}

void adl_curve_add_to_figure(Figure *figure, Point *src_points, size_t src_len, uint32_t color)
```

```
{
    Curve src_points_ada;
    ada_init_array(Point, src_points_ada);
    src_points_ada.color = color;

    for (size_t i = 0; i < src_len; i++) {
        Point current_point = src_points[i];
        if (current_point.x > figure->max_x) {
            figure->max_x = current_point.x;
        }
        if (current_point.y > figure->max_y) {
            figure->max_y = current_point.y;
        }
        if (current_point.x < figure->min_x) {
            figure->min_x = current_point.x;
        }
        if (current_point.y < figure->min_y) {
            figure->min_y = current_point.y;
        }
        ada_append(Point, src_points_ada, current_point);
    }

    ada_append(Curve, figure->src_curve_array, src_points_ada);
}

void adl_curves_plot_on_figure(Figure figure)
{
    mat2D_fill_uint32(figure.pixels_mat, figure.background_color);
    memset(figure.inv_z_buffer_mat.elements, 0x0, sizeof(double) * figure.inv_z_buffer_mat.rows * figure.inv_z_buffer_mat.cols);
    if (figure.to_draw_axis) adl_axis_draw_on_figure(&figure);

    for (size_t curve_index = 0; curve_index < figure.src_curve_array.length; curve_index++) {
        size_t src_len = figure.src_curve_array.elements[curve_index].length;
        Point *src_points = figure.src_curve_array.elements[curve_index].elements;
        for (size_t i = 0; i < src_len-1; i++) {
            Point src_start = src_points[i];
            Point src_end   = src_points[i+1];
            Point des_start = {0};
            Point des_end = {0};

            des_start.x = adl_linear_map(src_start.x, figure.min_x, figure.max_x, figure.min_x_pixel, figure.max_x_pixel);
            des_start.y = ((figure.max_y_pixel + figure.min_y_pixel) - adl_linear_map(src_start.y, figure.min_y, figure.max_y, figure.min_y_pixel, figure.max_y_pixel));

            des_end.x = adl_linear_map(src_end.x, figure.min_x, figure.max_x, figure.min_x_pixel, figure.max_x_pixel);
            des_end.y = ((figure.max_y_pixel + figure.min_y_pixel) - adl_linear_map(src_end.y, figure.min_y, figure.max_y, figure.min_y_pixel, figure.max_y_pixel));

            adl_line_draw(figure.pixels_mat, des_start.x, des_start.y, des_end.x, des_end.y, figure.src_curve_array.elements[curve_index].color, figure.offset_zoom_param);
        }
    }

    if (figure.to_draw_max_min_values) adl_max_min_values_draw_on_figure(figure);
}

/* check offset2D. might convert it to a Mat2D */
#define adl_offset2d(i, j, ni) (j) * (ni) + (i)
void adl_2Dscalar_interp_on_figure(Figure figure, double *x_2Dmat, double *y_2Dmat, double *scalar_2Dmat, int ni, int nj, char color_scale[], float num_of_rotations)
{
    mat2D_fill_uint32(figure.pixels_mat, figure.background_color);
    memset(figure.inv_z_buffer_mat.elements, 0x0, sizeof(double) * figure.inv_z_buffer_mat.rows * figure.inv_z_buffer_mat.cols);
    if (figure.to_draw_axis) adl_axis_draw_on_figure(&figure);

    float min_scalar = FLT_MAX;
    float max_scalar = FLT_MIN;
    for (int i = 0; i < ni; i++) {
        for (int j = 0; j < nj; j++) {
            float val = scalar_2Dmat[adl_offset2d(i, j, ni)];
            if (val > max_scalar) max_scalar = val;
            if (val < min_scalar) min_scalar = val;
            float current_x = x_2Dmat[adl_offset2d(i, j, ni)];
            float current_y = y_2Dmat[adl_offset2d(i, j, ni)];
            if (current_x > figure.max_x) {
                figure.max_x = current_x;
            }
            if (current_y > figure.max_y) {
                figure.max_y = current_y;
            }
            if (current_x < figure.min_x) {
                figure.min_x = current_x;
            }
```

```
            if (current_y < figure.min_y) {
                figure.min_y = current_y;
            }
        }
    }

    float window_w = (float)figure.pixels_mat.cols;
    float window_h = (float)figure.pixels_mat.rows;

    for (int i = 0; i < ni-1; i++) {
        for (int j = 0; j < nj-1; j++) {
            Quad quad = {0};
            quad.light_intensity[0] = 1;
            quad.light_intensity[1] = 1;
            quad.light_intensity[2] = 1;
            quad.light_intensity[3] = 1;
            quad.to_draw = 1;

            quad.points[3].x = x_2Dmat[adl_offset2d(i  , j  , ni)];
            quad.points[3].y = y_2Dmat[adl_offset2d(i  , j  , ni)];
            quad.points[2].x = x_2Dmat[adl_offset2d(i+1, j  , ni)];
            quad.points[2].y = y_2Dmat[adl_offset2d(i+1, j  , ni)];
            quad.points[1].x = x_2Dmat[adl_offset2d(i+1, j+1, ni)];
            quad.points[1].y = y_2Dmat[adl_offset2d(i+1, j+1, ni)];
            quad.points[0].x = x_2Dmat[adl_offset2d(i  , j+1, ni)];
            quad.points[0].y = y_2Dmat[adl_offset2d(i  , j+1, ni)];

            for (int p_index = 0; p_index < 4; p_index++) {
                quad.points[p_index].z = 1;
                quad.points[p_index].w = 1;
                quad.points[p_index].x = adl_linear_map(quad.points[p_index].x, figure.min_x, figure.max_x, figure.min_x_pixel, figure.max_x_pixel);
                quad.points[p_index].y = ((figure.max_y_pixel + figure.min_y_pixel) - adl_linear_map(quad.points[p_index].y, figure.min_y, figure.max_y, figure.min_y_pixel, figure.max_y_pixel));

                adl_offset_zoom_point(quad.points[p_index], window_w, window_h, figure.offset_zoom_param);
            }

            float t3 = adl_linear_map(scalar_2Dmat[adl_offset2d(i  , j  , ni)], min_scalar, max_scalar, 0, 1);
            float t2 = adl_linear_map(scalar_2Dmat[adl_offset2d(i+1, j  , ni)], min_scalar, max_scalar, 0, 1);
            float t1 = adl_linear_map(scalar_2Dmat[adl_offset2d(i+1, j+1, ni)], min_scalar, max_scalar, 0, 1);
            float t0 = adl_linear_map(scalar_2Dmat[adl_offset2d(  i, j+1, ni)], min_scalar, max_scalar, 0, 1);

            /* https://en.wikipedia.org/wiki/Oklab_color_space */
            if (!strcmp(color_scale, "b-c")) {
                uint32_t color = 0, color1 = BLUE_hexARGB, color2 = CYAN_hexARGB;
                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t0, num_of_rotations, &color);
                quad.colors[0] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t1, num_of_rotations, &color);
                quad.colors[1] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t2, num_of_rotations, &color);
                quad.colors[2] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t3, num_of_rotations, &color);
                quad.colors[3] = color;
            } else if (!strcmp(color_scale, "b-g")) {
                uint32_t color = 0, color1 = BLUE_hexARGB, color2 = GREEN_hexARGB;
                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t0, num_of_rotations, &color);
                quad.colors[0] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t1, num_of_rotations, &color);
                quad.colors[1] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t2, num_of_rotations, &color);
                quad.colors[2] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t3, num_of_rotations, &color);
                quad.colors[3] = color;
            } else if (!strcmp(color_scale, "b-r")) {
                uint32_t color = 0, color1 = BLUE_hexARGB, color2 = RED_hexARGB;
                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t0, num_of_rotations, &color);
                quad.colors[0] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t1, num_of_rotations, &color);
                quad.colors[1] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t2, num_of_rotations, &color);
                quad.colors[2] = color;

```

```
                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t3, num_of_rotations, &color);
                quad.colors[3] = color;
            } else if (!strcmp(color_scale, "b-y")) {
                uint32_t color = 0, color1 = BLUE_hexARGB, color2 = YELLOW_hexARGB;
                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t0, num_of_rotations, &color);
                quad.colors[0] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t1, num_of_rotations, &color);
                quad.colors[1] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t2, num_of_rotations, &color);
                quad.colors[2] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t3, num_of_rotations, &color);
                quad.colors[3] = color;
            } else if (!strcmp(color_scale, "g-y")) {
                uint32_t color = 0, color1 = GREEN_hexARGB, color2 = YELLOW_hexARGB;
                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t0, num_of_rotations, &color);
                quad.colors[0] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t1, num_of_rotations, &color);
                quad.colors[1] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t2, num_of_rotations, &color);
                quad.colors[2] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t3, num_of_rotations, &color);
                quad.colors[3] = color;
            } else if (!strcmp(color_scale, "g-p")) {
                uint32_t color = 0, color1 = GREEN_hexARGB, color2 = PURPLE_hexARGB;
                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t0, num_of_rotations, &color);
                quad.colors[0] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t1, num_of_rotations, &color);
                quad.colors[1] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t2, num_of_rotations, &color);
                quad.colors[2] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t3, num_of_rotations, &color);
                quad.colors[3] = color;
            } else if (!strcmp(color_scale, "g-r")) {
                uint32_t color = 0, color1 = GREEN_hexARGB, color2 = RED_hexARGB;
                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t0, num_of_rotations, &color);
                quad.colors[0] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t1, num_of_rotations, &color);
                quad.colors[1] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t2, num_of_rotations, &color);
                quad.colors[2] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t3, num_of_rotations, &color);
                quad.colors[3] = color;
            } else if (!strcmp(color_scale, "r-y")) {
                uint32_t color = 0, color1 = RED_hexARGB, color2 = YELLOW_hexARGB;
                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t0, num_of_rotations, &color);
                quad.colors[0] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t1, num_of_rotations, &color);
                quad.colors[1] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t2, num_of_rotations, &color);
                quad.colors[2] = color;

                adl_interpolate_ARGBcolor_on_okLch(color1, color2, t3, num_of_rotations, &color);
                quad.colors[3] = color;
            }

            adl_quad_fill_interpolate_color_mean_value(figure.pixels_mat, figure.inv_z_buffer_mat,
 quad, ADL_DEFAULT_OFFSET_ZOOM);
        }
    }

    if (figure.to_draw_max_min_values) {
        adl_max_min_values_draw_on_figure(figure);
    }

}

Grid adl_cartesian_grid_create(float min_e1, float max_e1, float min_e2, float max_e2, int
 num_samples_e1, int num_samples_e2, char plane[], float third_direction_position)
{
    Grid grid;
    ada_init_array(Curve, grid.curves);

    grid.min_e1 = min_e1;
```

```
    grid.max_e1 = max_e1;
    grid.min_e2 = min_e2;
    grid.max_e2 = max_e2;
    grid.num_samples_e1 = num_samples_e1;
    grid.num_samples_e2 = num_samples_e2;
    strncpy(grid.plane, plane, 2);

    float del_e1 = (max_e1 - min_e1) / num_samples_e1;
    float del_e2 = (max_e2 - min_e2) / num_samples_e2;

    grid.de1 = del_e1;
    grid.de2 = del_e2;

    if (!strncmp(plane, "XY", 3) || !strncmp(plane, "xy", 3)) {
        for (int e1_index = 0; e1_index <= num_samples_e1; e1_index++) {
            Curve curve = {0};
            ada_init_array(Point, curve);
            Point point_max = {0}, point_min = {0};

            point_min.x = min_e1 + e1_index * del_e1;
            point_min.y = min_e2;
            point_min.z = third_direction_position;
            point_min.w = 1;

            point_max.x = min_e1 + e1_index * del_e1;
            point_max.y = max_e2;
            point_max.z = third_direction_position;
            point_max.w = 1;

            ada_append(Point, curve, point_min);
            ada_append(Point, curve, point_max);

            ada_append(Curve, grid.curves, curve);
        }
        for (int e2_index = 0; e2_index <= num_samples_e2; e2_index++) {
            Curve curve = {0};
            ada_init_array(Point, curve);
            Point point_max = {0}, point_min = {0};

            point_min.x = min_e1;
            point_min.y = min_e2 + e2_index * del_e2;
            point_min.z = third_direction_position;
            point_min.w = 1;

            point_max.x = max_e1;
            point_max.y = min_e2 + e2_index * del_e2;
            point_max.z = third_direction_position;
            point_max.w = 1;

            ada_append(Point, curve, point_min);
            ada_append(Point, curve, point_max);

            ada_append(Curve, grid.curves, curve);
        }
    } else if (!strncmp(plane, "XZ", 3) || !strncmp(plane, "xz", 3)) {
        for (int e1_index = 0; e1_index <= num_samples_e1; e1_index++) {
            Curve curve = {0};
            ada_init_array(Point, curve);
            Point point_max = {0}, point_min = {0};

            point_min.x = min_e1 + e1_index * del_e1;
            point_min.y = third_direction_position;
            point_min.z = min_e2;
            point_min.w = 1;

            point_max.x = min_e1 + e1_index * del_e1;
            point_max.y = third_direction_position;
            point_max.z = max_e2;
            point_max.w = 1;

            ada_append(Point, curve, point_min);
            ada_append(Point, curve, point_max);

            ada_append(Curve, grid.curves, curve);
        }
        for (int e2_index = 0; e2_index <= num_samples_e2; e2_index++) {
            Curve curve = {0};
            ada_init_array(Point, curve);
            Point point_max = {0}, point_min = {0};

            point_min.x = min_e1;
            point_min.y = third_direction_position;
            point_min.z = min_e2 + e2_index * del_e2;
            point_min.w = 1;

            point_max.x = max_e1;
            point_max.y = third_direction_position;
```

```
            point_max.z = min_e2 + e2_index * del_e2;
            point_max.w = 1;

            ada_append(Point, curve, point_min);
            ada_append(Point, curve, point_max);

            ada_append(Curve, grid.curves, curve);
        }
    } else if (!strncmp(plane, "YX", 3) || !strncmp(plane, "yx", 3)) {
        for (int e1_index = 0; e1_index <= num_samples_e1; e1_index++) {
            Curve curve = {0};
            ada_init_array(Point, curve);
            Point point_max = {0}, point_min = {0};

            point_min.x = min_e2;
            point_min.y = min_e1 + e1_index * del_e1;
            point_min.z = third_direction_position;
            point_min.w = 1;

            point_max.x = max_e2;
            point_max.y = min_e1 + e1_index * del_e1;
            point_max.z = third_direction_position;
            point_max.w = 1;

            ada_append(Point, curve, point_min);
            ada_append(Point, curve, point_max);

            ada_append(Curve, grid.curves, curve);
        }
        for (int e2_index = 0; e2_index <= num_samples_e2; e2_index++) {
            Curve curve = {0};
            ada_init_array(Point, curve);
            Point point_max = {0}, point_min = {0};

            point_min.x = min_e2 + e2_index * del_e2;
            point_min.y = min_e1;
            point_min.z = third_direction_position;
            point_min.w = 1;

            point_max.x = min_e2 + e2_index * del_e2;
            point_max.y = max_e1;
            point_max.z = third_direction_position;
            point_max.w = 1;

            ada_append(Point, curve, point_min);
            ada_append(Point, curve, point_max);

            ada_append(Curve, grid.curves, curve);
        }
    } else if (!strncmp(plane, "YZ", 3) || !strncmp(plane, "yz", 3)) {
        for (int e1_index = 0; e1_index <= num_samples_e1; e1_index++) {
            Curve curve = {0};
            ada_init_array(Point, curve);
            Point point_max = {0}, point_min = {0};

            point_min.x = third_direction_position;
            point_min.y = min_e1 + e1_index * del_e1;
            point_min.z = min_e2;
            point_min.w = 1;

            point_max.x = third_direction_position;
            point_max.y = min_e1 + e1_index * del_e1;
            point_max.z = max_e2;
            point_max.w = 1;

            ada_append(Point, curve, point_min);
            ada_append(Point, curve, point_max);

            ada_append(Curve, grid.curves, curve);
        }
        for (int e2_index = 0; e2_index <= num_samples_e2; e2_index++) {
            Curve curve = {0};
            ada_init_array(Point, curve);
            Point point_max = {0}, point_min = {0};

            point_min.x = third_direction_position;
            point_min.y = min_e1;
            point_min.z = min_e2 + e2_index * del_e2;
            point_min.w = 1;

            point_max.x = third_direction_position;
            point_max.y = max_e1;
            point_max.z = min_e2 + e2_index * del_e2;
            point_max.w = 1;

            ada_append(Point, curve, point_min);
            ada_append(Point, curve, point_max);
```

```
            ada_append(Curve, grid.curves, curve);
        }
    } else if (!strncmp(plane, "ZX", 3) || !strncmp(plane, "zx", 3)) {
        for (int e1_index = 0; e1_index <= num_samples_e1; e1_index++) {
            Curve curve = {0};
            ada_init_array(Point, curve);
            Point point_max = {0}, point_min = {0};

            point_min.x = min_e2;
            point_min.y = third_direction_position;
            point_min.z = min_e1 + e1_index * del_e1;
            point_min.w = 1;

            point_max.x = max_e2;
            point_max.y = third_direction_position;
            point_max.z = min_e1 + e1_index * del_e1;
            point_max.w = 1;

            ada_append(Point, curve, point_min);
            ada_append(Point, curve, point_max);

            ada_append(Curve, grid.curves, curve);
        }
        for (int e2_index = 0; e2_index <= num_samples_e2; e2_index++) {
            Curve curve = {0};
            ada_init_array(Point, curve);
            Point point_max = {0}, point_min = {0};

            point_min.x = min_e2 + e2_index * del_e2;
            point_min.y = third_direction_position;
            point_min.z = min_e1;
            point_min.w = 1;

            point_max.x = min_e2 + e2_index * del_e2;
            point_max.y = third_direction_position;
            point_max.z = max_e1;
            point_max.w = 1;

            ada_append(Point, curve, point_min);
            ada_append(Point, curve, point_max);

            ada_append(Curve, grid.curves, curve);
        }
    } else if (!strncmp(plane, "ZY", 3) || !strncmp(plane, "zy", 3)) {
        for (int e1_index = 0; e1_index <= num_samples_e1; e1_index++) {
            Curve curve = {0};
            ada_init_array(Point, curve);
            Point point_max = {0}, point_min = {0};

            point_min.x = third_direction_position;
            point_min.y = min_e2;
            point_min.z = min_e1 + e1_index * del_e1;
            point_min.w = 1;

            point_max.x = third_direction_position;
            point_max.y = max_e2;
            point_max.z = min_e1 + e1_index * del_e1;
            point_max.w = 1;

            ada_append(Point, curve, point_min);
            ada_append(Point, curve, point_max);

            ada_append(Curve, grid.curves, curve);
        }
        for (int e2_index = 0; e2_index <= num_samples_e2; e2_index++) {
            Curve curve = {0};
            ada_init_array(Point, curve);
            Point point_max = {0}, point_min = {0};

            point_min.x = third_direction_position;
            point_min.y = min_e2 + e2_index * del_e2;
            point_min.z = min_e1;
            point_min.w = 1;

            point_max.x = third_direction_position;
            point_max.y = min_e2 + e2_index * del_e2;
            point_max.z = max_e1;
            point_max.w = 1;

            ada_append(Point, curve, point_min);
            ada_append(Point, curve, point_max);

            ada_append(Curve, grid.curves, curve);
        }
    }

```

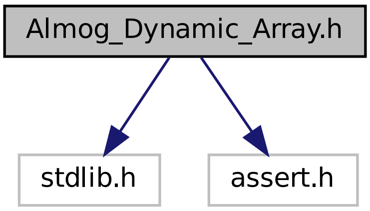
```
    return grid;
}

void adl_grid_draw(Mat2D_uint32 screen_mat, Grid grid, uint32_t color, Offset_zoom_param
 offset_zoom_param)
{
    for (size_t curve_index = 0; curve_index < grid.curves.length; curve_index++) {
        adl_lines_draw(screen_mat, grid.curves.elements[curve_index].elements,
 grid.curves.elements[curve_index].length, color, offset_zoom_param);
    }
}

#endif /*ALMOG_DRAW_LIBRARY_IMPLEMENTATION*/
```

## 4.3 Almog_Dynamic_Array.h File Reference

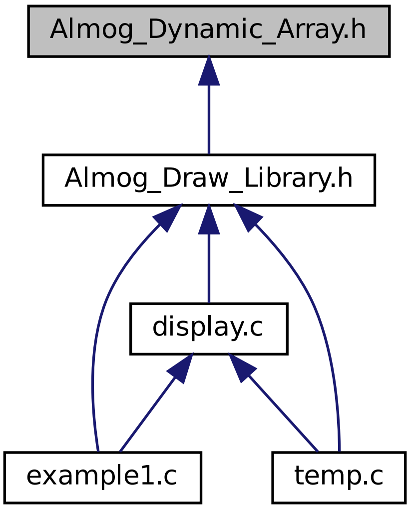
Header-only C macros that implement a simple dynamic array.

```
#include <stdlib.h>
#include <assert.h>
```

Include dependency graph for Almog_Dynamic_Array.h:



This graph shows which files directly or indirectly include this file:




## Macros

- #define ADA_INIT_CAPACITY 10

  *Default initial capacity used by ada_init_array.*
- #define ADA_MALLOC malloc

  *Allocation function used by this header (defaults to malloc).*
- #define ADA_REALLOC realloc

  *Reallocation function used by this header (defaults to realloc).*
- #define ADA_ASSERT assert

  *Assertion macro used by this header (defaults to assert).*
- #define ada_init_array(type, header)

  *Initialize an array header and allocate its initial storage.*
- #define ada_resize(type, header, new_capacity)

  *Resize the underlying storage to hold new_capacity elements.*
- #define ada_append(type, header, value)

  *Append a value to the end of the array, growing if necessary.*
- #define ada_insert(type, header, value, index)

  *Insert value at position index, preserving order (O(n)).*
- #define ada_insert_unordered(type, header, value, index)

  *Insert value at index without preserving order (O(1) amortized).*
- #define ada_remove(type, header, index)

  *Remove element at index, preserving order (O(n)).*
- #define ada_remove_unordered(type, header, index)

  *Remove element at index by moving the last element into its place (O(1)); order is not preserved.*

### 4.3.1 Detailed Description

Header-only C macros that implement a simple dynamic array.

This header provides a minimal, macro-based dynamic array for POD-like types. The array "header" is a user-defined struct with three fields:

- size_t length; current number of elements
- size_t capacity; allocated capacity (in elements)
- T∗ elements; pointer to contiguous storage of elements (type T)

How to use: 1) Define a header struct with length/capacity/elements fields. 2) Initialize it with ada_init_array(T, header). 3) Modify it with ada_append (append), ada_insert, remove variants, etc. 4) When done, free(header.elements) (or your custom deallocator).

Customization:

- Define ADA_MALLOC, ADA_REALLOC, and ADA_ASSERT before including this header to override allocation and assertion behavior.

Complexity (n = number of elements):

- Append: amortized O(1)
- Ordered insert/remove: O(n)
- Unordered insert/remove: O(1)

Notes and limitations:

- These are macros; arguments may be evaluated multiple times. Pass only simple lvalues (no side effects).
- Index checks rely on ADA_ASSERT; with NDEBUG they may be compiled out.
- ada_resize exits the process (exit(1)) if reallocation fails.
- ada_insert reads header.elements[header.length - 1] internally; inserting into an empty array via ada_insert is undefined behavior. Use ada_append or ada_insert_unordered for that case.
- No automatic shrinking; you may call ada_resize manually.

Example: typedef struct { size_t length; size_t capacity; int∗ elements; } ada_int_array;

ada_int_array arr; ada_init_array(int, arr); ada_append(int, arr, 42); ada_insert(int, arr, 7, 0); // requires arr.length > 0 ada_remove(int, arr, 1); free(arr.elements);

Definition in file Almog_Dynamic_Array.h.

### 4.3.2 Macro Definition Documentation

#### 4.3.2.1 ada_append

```
#define ada_append(
            type,
            header,
            value )
```

**Value:**

```
do {                                    \
    if (header.length >= header.capacity) {                \
        ada_resize(type, header, (int)(header.capacity*1.5));  \
    }                                                      \
    header.elements[header.length] = value;                \
    header.length++;                                       \
} while (0)
```

Append a value to the end of the array, growing if necessary.

**Parameters**

| | |
|---|---|
| *type* | Element type stored in the array. |
| *header* | Lvalue of the header struct. |
| *value* | Value to append. |

**Postcondition**

header.length is incremented by 1; the last element equals value.

**Note**

Growth factor is (int)(header.capacity $*$ 1.5). Because of truncation, very small capacities may not grow (e.g., from 1 to 1). With the default INIT_CAPACITY=10 this is typically not an issue unless you manually shrink capacity. Ensure growth always increases capacity by at least 1 if you customize this macro.

Definition at line 169 of file Almog_Dynamic_Array.h.

#### 4.3.2.2 ADA_ASSERT

```
#define ADA_ASSERT assert
```

Assertion macro used by this header (defaults to assert).

Define ADA_ASSERT before including this file to override. When NDEBUG is defined, standard assert() is disabled.

Definition at line 96 of file Almog_Dynamic_Array.h.

#### 4.3.2.3 ada_init_array

```
#define ada_init_array(
            type,
            header )
```

**Value:**

```
do {                                              \
    header.capacity = ADA_INIT_CAPACITY;                       \
    header.length = 0;                                         \
    header.elements = (type *)ADA_MALLOC(sizeof(type) * header.capacity);  \
    ADA_ASSERT(header.elements != NULL);                       \
} while (0)
```

Initialize an array header and allocate its initial storage.

**Parameters**

| | |
|---|---|
| *type* | Element type stored in the array (e.g., int). |
| *header* | Lvalue of the header struct containing fields: length, capacity, and elements. |

**Precondition**

header is a modifiable lvalue; header.elements is uninitialized or ignored and will be overwritten.

**Postcondition**

header.length == 0, header.capacity == INIT_CAPACITY, header.elements != NULL (or ADA_ASSERT fails).

**Note**

Allocation uses ADA_MALLOC and is checked via ADA_ASSERT.

Definition at line 120 of file Almog_Dynamic_Array.h.

#### 4.3.2.4 ADA_INIT_CAPACITY

```
#define ADA_INIT_CAPACITY 10
```

Default initial capacity used by ada_init_array.

You may override this by defining INIT_CAPACITY before including this file.

Definition at line 64 of file Almog_Dynamic_Array.h.

#### 4.3.2.5 ada_insert

```
#define ada_insert(
                type,
                header,
                value,
                index )
```

**Value:**

```
    do {                                                        \
    ADA_ASSERT((int)(index) >= 0);
        \
    ADA_ASSERT((float)(index) - (int)(index) == 0);
        \
    ada_append(type, header, header.elements[header.length-1]);
        \
    for (size_t ada_for_loop_index = header.length-2; ada_for_loop_index > (index); ada_for_loop_index--) {
        \
        header.elements[ada_for_loop_index] = header.elements [ada_for_loop_index-1];
        \
    }
        \
    header.elements[(index)] = value;
        \
} while (0)
```

Insert value at position index, preserving order (O(n)).

**Parameters**

| | |
|---|---|
| *type* | Element type stored in the array. |
| *header* | Lvalue of the header struct. |
| *value* | Value to insert. |
| *index* | Destination index in the range [0, header.length]. |

**Precondition**

0 <= index <= header.length.

header.length > 0 if index == header.length (this macro reads the last element internally). For inserting into an empty array, use ada_append or ada_insert_unordered.

**Postcondition**

Element is inserted at index; subsequent elements are shifted right; header.length is incremented by 1.

**Note**

This macro asserts index is non-negative and an integer value using ADA_ASSERT. No explicit upper-bound assert is performed.

Definition at line 196 of file Almog_Dynamic_Array.h.

#### 4.3.2.6 ada_insert_unordered

```
#define ada_insert_unordered(
                type,
                header,
                value,
                index )
```

**Value:**

```
    do {    \
    ADA_ASSERT((int)(index) >= 0);                              \
    ADA_ASSERT((float)(index) - (int)(index) == 0);             \
    if ((size_t)(index) == header.length) {                     \
        ada_append(type, header, value);                        \
    } else {                                                    \
        ada_append(type, header, header.elements[(index)]);     \
        header.elements[(index)] = value;                       \
    }                                                           \
} while (0)
```

Insert value at index without preserving order (O(1) amortized).

If index == header.length, this behaves like an append. Otherwise, the current element at index is moved to the end, and value is written at index.

**Parameters**

| | |
|---|---|
| *type* | Element type stored in the array. |
| *header* | Lvalue of the header struct. |
| *value* | Value to insert. |
| *index* | Index in the range [0, header.length]. |

**Precondition**

0 <= index <= header.length.

**Postcondition**

header.length is incremented by 1; array order is not preserved.

Definition at line 222 of file Almog_Dynamic_Array.h.

#### 4.3.2.7 ADA_MALLOC

```
#define ADA_MALLOC malloc
```

Allocation function used by this header (defaults to malloc).

Define ADA_MALLOC to a compatible allocator before including this file to override the default.

Definition at line 74 of file Almog_Dynamic_Array.h.

#### 4.3.2.8 ADA_REALLOC

```
#define ADA_REALLOC realloc
```

Reallocation function used by this header (defaults to realloc).

Define ADA_REALLOC to a compatible reallocator before including this file to override the default.

Definition at line 85 of file Almog_Dynamic_Array.h.

#### 4.3.2.9 ada_remove

```
#define ada_remove(
                type,
                header,
                index )
```

**Value:**

```
    do {                                                                    \
    ADA_ASSERT((int)(index) >= 0);
        \
    ADA_ASSERT((float)(index) - (int)(index) == 0);
        \
    for (size_t ada_for_loop_index = (index); ada_for_loop_index < header.length-1; ada_for_loop_index++) {
        \
        header.elements[ada_for_loop_index] = header.elements[ada_for_loop_index+1];
        \
    }
        \
    header.length--;
        \
} while (0)
```

Remove element at index, preserving order (O(n)).

**Parameters**

| | |
|---|---|
| *type* | Element type stored in the array. |
| *header* | Lvalue of the header struct. |
| *index* | Index in the range [0, header.length - 1]. |

**Precondition**

0 <= index < header.length.

**Postcondition**

header.length is decremented by 1; subsequent elements are shifted left by one position. The element beyond the new length is left uninitialized.

Definition at line 246 of file Almog_Dynamic_Array.h.

#### 4.3.2.10 ada_remove_unordered

```
#define ada_remove_unordered(
              type,
              header,
              index )
```

**Value:**

```
    do {            \
    ADA_ASSERT((int)(index) >= 0);                              \
    ADA_ASSERT((float)(index) - (int)(index) == 0);             \
    header.elements[index] = header.elements[header.length-1];  \
    header.length--;                                            \
} while (0)
```

Remove element at index by moving the last element into its place (O(1)); order is not preserved.

**Parameters**

| | |
|---|---|
| *type* | Element type stored in the array. |
| *header* | Lvalue of the header struct. |
| *index* | Index in the range [0, header.length - 1]. |

**Precondition**

0 <= index < header.length and header.length > 0.

**Postcondition**

header.length is decremented by 1; array order is not preserved.

Definition at line 267 of file Almog_Dynamic_Array.h.

#### 4.3.2.11 ada_resize

```
#define ada_resize(
            type,
            header,
            new_capacity )
```

**Value:**

```
    do {                                                                         \
        type *ada_temp_pointer = (type *)ADA_REALLOC((void *)(header.elements), new_capacity*sizeof(type));
        \
        if (ada_temp_pointer == NULL) {
        \
            exit(1);
        \
        }
        \
        header.elements = ada_temp_pointer;
        \
        ADA_ASSERT(header.elements != NULL);
        \
        header.capacity = new_capacity;
        \
    } while (0)
```

Resize the underlying storage to hold new_capacity elements.

**Parameters**

| | |
|---|---|
| *type* | Element type stored in the array. |
| *header* | Lvalue of the header struct. |
| *new_capacity* | New capacity in number of elements. |

**Precondition**

new_capacity >= header.length (otherwise elements beyond new_capacity are lost and length will not be adjusted).

**Postcondition**

header.capacity == new_capacity and header.elements points to a block large enough for new_capacity elements.

**Warning**

On allocation failure, this macro calls exit(1).

**Note**

Reallocation uses ADA_REALLOC and is also checked via ADA_ASSERT.

Definition at line 143 of file Almog_Dynamic_Array.h.

## 4.4 Almog_Dynamic_Array.h

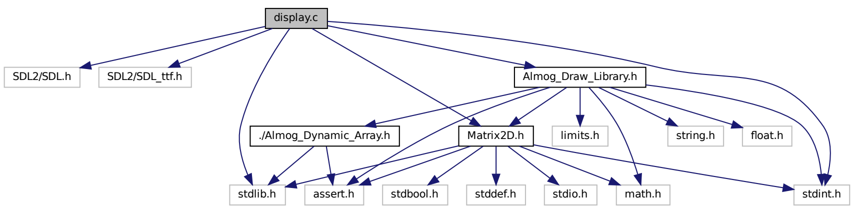
```
#ifndef ALMOG_DYNAMIC_ARRAY_H_
#define ALMOG_DYNAMIC_ARRAY_H_

#include <stdlib.h>
#include <assert.h>


#define ADA_INIT_CAPACITY 10

#ifndef ADA_MALLOC
#define ADA_MALLOC malloc
#endif /*ADA_MALLOC*/

#ifndef ADA_REALLOC
#define ADA_REALLOC realloc
#endif /*ADA_REALLOC*/

#ifndef ADA_ASSERT
#define ADA_ASSERT assert
#endif /*ADA_ASSERT*/

/* typedef struct {
    size_t length;
    size_t capacity;
    int* elements;
} ada_int_array; */

#define ada_init_array(type, header) do {                                      \
        header.capacity = ADA_INIT_CAPACITY;                                        \
        header.length = 0;                                                     \
        header.elements = (type *)ADA_MALLOC(sizeof(type) * header.capacity);  \
        ADA_ASSERT(header.elements != NULL);                                   \
    } while (0)

#define ada_resize(type, header, new_capacity) do { \
        type *ada_temp_pointer = (type *)ADA_REALLOC((void *)(header.elements), new_capacity*sizeof(type)); \
        if (ada_temp_pointer == NULL) { \
            exit(1); \
        } \
        header.elements = ada_temp_pointer; \
        ADA_ASSERT(header.elements != NULL); \
        header.capacity = new_capacity; \
    } while (0)

#define ada_append(type, header, value) do {                        \
        if (header.length >= header.capacity) {                     \
            ada_resize(type, header, (int)(header.capacity*1.5));   \
        }                                                           \
        header.elements[header.length] = value;                     \
        header.length++;                                            \
    } while (0)

#define ada_insert(type, header, value, index) do { \
    ADA_ASSERT((int)(index) >= 0); \
    ADA_ASSERT((float)(index) - (int)(index) == 0); \
    ada_append(type, header, header.elements[header.length-1]); \
    for (size_t ada_for_loop_index = header.length-2; ada_for_loop_index > (index); ada_for_loop_index--) { \
        header.elements[ada_for_loop_index] = header.elements [ada_for_loop_index-1]; \
    } \
    header.elements[(index)] = value; \
} while (0)


#define ada_insert_unordered(type, header, value, index) do {   \
    ADA_ASSERT((int)(index) >= 0);                              \
    ADA_ASSERT((float)(index) - (int)(index) == 0);             \
    if ((size_t)(index) == header.length) {                     \
        ada_append(type, header, value);                        \
```

```
    } else {                                                   \
        ada_append(type, header, header.elements[(index)]);    \
        header.elements[(index)] = value;                      \
    }                                                          \
} while (0)

#define ada_remove(type, header, index) do {
      \
    ADA_ASSERT((int)(index) >= 0);
      \
    ADA_ASSERT((float)(index) - (int)(index) == 0);
      \
    for (size_t ada_for_loop_index = (index); ada_for_loop_index < header.length-1;
 ada_for_loop_index++) { \
        header.elements[ada_for_loop_index] = header.elements[ada_for_loop_index+1];
      \
    }
      \
    header.length--;
      \
} while (0)

#define ada_remove_unordered(type, header, index) do {         \
    ADA_ASSERT((int)(index) >= 0);                             \
    ADA_ASSERT((float)(index) - (int)(index) == 0);            \
    header.elements[index] = header.elements[header.length-1]; \
    header.length--;                                           \
} while (0)


#endif /*ALMOG_DYNAMIC_ARRAY_H_*/
```

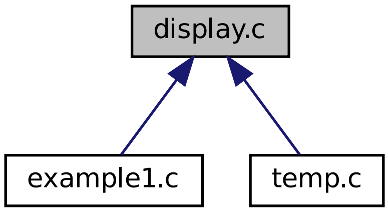
## 4.5 display.c File Reference

```
#include <SDL2/SDL.h>
#include <SDL2/SDL_ttf.h>
#include "Matrix2D.h"
#include <stdlib.h>
#include <stdint.h>
#include "Almog_Draw_Library.h"
```

Include dependency graph for display.c:



This graph shows which files directly or indirectly include this file:




## Classes

- struct game_state_t

## Macros

- #define WINDOW_WIDTH (16 ∗ 80)
- #define WINDOW_HEIGHT (9 ∗ 80)
- #define FPS 100
- #define FRAME_TARGET_TIME (1000 / FPS)
- #define dprintSTRING(expr) printf(#expr " = %s\n", expr)
- #define dprintCHAR(expr) printf(#expr " = %c\n", expr)
- #define dprintINT(expr) printf(#expr " = %d\n", expr)
- #define dprintD(expr) printf(#expr " = %g\n", expr)
- #define dprintSIZE_T(expr) printf(#expr " = %zu\n", expr)
- #define SETUP
- #define UPDATE
- #define RENDER

## Functions

- int initialize_window (game_state_t ∗game_state)
- void setup_window (game_state_t ∗game_state)
- void process_input_window (game_state_t ∗game_state)
- void update_window (game_state_t ∗game_state)
- void render_window (game_state_t ∗game_state)
- void destroy_window (game_state_t ∗game_state)
- void fix_framerate (game_state_t ∗game_state)
- void setup (game_state_t ∗game_state)
- void update (game_state_t ∗game_state)
- void render (game_state_t ∗game_state)
- void check_window_mat_size (game_state_t ∗game_state)
- void copy_mat_to_surface_RGB (game_state_t ∗game_state)
- int main ()

### 4.5.1 Macro Definition Documentation

#### 4.5.1.1 dprintCHAR

```
#define dprintCHAR(
             expr ) printf(#expr " = %c\n", expr)
```

Definition at line 25 of file display.c.

#### 4.5.1.2 dprintD

```
#define dprintD(
             expr ) printf(#expr " = %g\n", expr)
```

Definition at line 27 of file display.c.

#### 4.5.1.3 dprintINT

```
#define dprintINT(
             expr ) printf(#expr " = %d\n", expr)
```

Definition at line 26 of file display.c.

#### 4.5.1.4 dprintSIZE_T

```
#define dprintSIZE_T(
             expr ) printf(#expr " = %zu\n", expr)
```

Definition at line 28 of file display.c.

#### 4.5.1.5 dprintSTRING

```
#define dprintSTRING(
             expr ) printf(#expr " = %s\n", expr)
```

Definition at line 24 of file display.c.

#### 4.5.1.6 FPS

```
#define FPS 100
```

Definition at line 17 of file display.c.

#### 4.5.1.7 FRAME_TARGET_TIME

```
#define FRAME_TARGET_TIME (1000 / FPS)
```

Definition at line 21 of file display.c.

#### 4.5.1.8 RENDER

```
#define RENDER
```

Definition at line 351 of file display.c.

#### 4.5.1.9 SETUP

```
#define SETUP
```

Definition at line 341 of file display.c.

#### 4.5.1.10 UPDATE

```
#define UPDATE
```

Definition at line 346 of file display.c.

#### 4.5.1.11 WINDOW_HEIGHT

```
#define WINDOW_HEIGHT (9 * 80)
```

Definition at line 13 of file display.c.

#### 4.5.1.12 WINDOW_WIDTH

```
#define WINDOW_WIDTH (16 * 80)
```

Definition at line 9 of file display.c.

### 4.5.2 Function Documentation

#### 4.5.2.1 check_window_mat_size()

```
void check_window_mat_size (
            game_state_t * game_state )
```

Definition at line 355 of file display.c.

References Mat2D_uint32::cols, game_state_t::inv_z_buffer_mat, mat2D_alloc(), mat2D_alloc_uint32(), mat2D_free(), mat2D_free_uint32(), Mat2D_uint32::rows, game_state_t::window, game_state_t::window_h, game_state_t::window_pixels_mat, game_state_t::window_surface, and game_state_t::window_w.

Referenced by update_window().

#### 4.5.2.2 copy_mat_to_surface_RGB()

```
void copy_mat_to_surface_RGB (
            game_state_t * game_state )
```

Definition at line 369 of file display.c.

References Mat2D_uint32::cols, Mat2D_uint32::elements, Mat2D_uint32::rows, game_state_t::window_pixels_mat, and game_state_t::window_surface.

Referenced by render_window().

#### 4.5.2.3 destroy_window()

```
void destroy_window (
            game_state_t * game_state )
```

Definition at line 312 of file display.c.

References mat2D_free_uint32(), game_state_t::renderer, game_state_t::window, game_state_t::window_pixels_mat, game_state_t::window_surface, and game_state_t::window_texture.

Referenced by main().

#### 4.5.2.4 fix_framerate()

```
void fix_framerate (
            game_state_t * game_state )
```

Definition at line 327 of file display.c.

References game_state_t::delta_time, game_state_t::frame_target_time, game_state_t::previous_frame_time, and game_state_t::to_limit_fps.

Referenced by update_window().

#### 4.5.2.5 initialize_window()

```
int initialize_window (
            game_state_t * game_state )
```

Definition at line 141 of file display.c.

References game_state_t::renderer, game_state_t::window, game_state_t::window_h, and game_state_t::window_w.

Referenced by main().

#### 4.5.2.6 main()

```
int main ( )
```

Definition at line 89 of file display.c.

References game_state_t::a_was_pressed, game_state_t::const_fps, game_state_t::d_was_pressed, game_state_t::delta_time, destroy_window(), game_state_t::e_was_pressed, game_state_t::elapsed_time, game_state_t::font, FPS, game_state_t::fps, FRAME_TARGET_TIME, game_state_t::frame_target_time, game_state_t::game_is_running, initialize_window(), game_state_t::left_button_pressed, game_state_t::offset_zoom_param, game_state_t::previous_frame_time, process_input_window(), game_state_t::q_was_pressed, render_window(), game_state_t::renderer, game_state_t::s_was_pressed, setup_window(), game_state_t::space_bar_was_pressed, game_state_t::to_clear_renderer, game_state_t::to_limit_fps, game_state_t::to_render, game_state_t::to_update, update_window(), game_state_t::w_was_pressed, game_state_t::window, game_state_t::window_h, WINDOW_HEIGHT, game_state_t::window_w, WINDOW_WIDTH, and Offset_zoom_param::zoom_multipl

#### 4.5.2.7 process_input_window()

```
void process_input_window (
            game_state_t * game_state )
```

Definition at line 196 of file display.c.

References ADL_MAX_ZOOM, game_state_t::game_is_running, game_state_t::left_button_pressed, Offset_zoom_param::offset_x, Offset_zoom_param::offset_y, game_state_t::offset_zoom_param, game_state_t::previous_frame_time, game_state_t::space_bar_w game_state_t::to_render, game_state_t::to_update, and Offset_zoom_param::zoom_multiplier.

Referenced by main().

#### 4.5.2.8 render()

```
void render (
            game_state_t * game_state )
```

Definition at line 352 of file display.c.

Referenced by render_window().

#### 4.5.2.9 render_window()

```
void render_window (
            game_state_t * game_state )
```

Definition at line 291 of file display.c.

References Mat2D::cols, Mat2D_uint32::cols, copy_mat_to_surface_RGB(), Mat2D::elements, Mat2D_uint32::elements, game_state_t::inv_z_buffer_mat, render(), Mat2D::rows, Mat2D_uint32::rows, game_state_t::to_clear_renderer, game_state_t::window, and game_state_t::window_pixels_mat.

Referenced by main().

#### 4.5.2.10 setup()

```
void setup (
            game_state_t * game_state )
```

Definition at line 342 of file display.c.

Referenced by setup_window().

#### 4.5.2.11 setup_window()

```
void setup_window (
            game_state_t * game_state )
```

Definition at line 182 of file display.c.

References game_state_t::inv_z_buffer_mat, mat2D_alloc(), mat2D_alloc_uint32(), setup(), game_state_t::window, game_state_t::window_h, game_state_t::window_pixels_mat, game_state_t::window_surface, and game_state_t::window_w.

Referenced by main().

#### 4.5.2.12 update()

```
void update (
            game_state_t * game_state )
```

Definition at line 347 of file display.c.

Referenced by update_window().

#### 4.5.2.13 update_window()

```
void update_window (
            game_state_t * game_state )
```

Definition at line 263 of file display.c.

References check_window_mat_size(), game_state_t::const_fps, game_state_t::delta_time, game_state_t::elapsed_time, fix_framerate(), game_state_t::fps, game_state_t::frame_target_time, game_state_t::to_limit_fps, update(), game_state_t::window, game_state_t::window_h, and game_state_t::window_w.

Referenced by main().

## 4.6 display.c

```
#include <SDL2/SDL.h>
#include <SDL2/SDL_ttf.h>
#include "Matrix2D.h"
#include <stdlib.h>
#include <stdint.h>
#include "Almog_Draw_Library.h"

#ifndef WINDOW_WIDTH
#define WINDOW_WIDTH (16 * 80)
#endif

#ifndef WINDOW_HEIGHT
#define WINDOW_HEIGHT (9 * 80)
#endif

#ifndef FPS
#define FPS 100
#endif

#ifndef FRAME_TARGET_TIME
#define FRAME_TARGET_TIME (1000 / FPS)
#endif

#define dprintSTRING(expr) printf(#expr " = %s\n", expr)
#define dprintCHAR(expr) printf(#expr " = %c\n", expr)
#define dprintINT(expr) printf(#expr " = %d\n", expr)
#define dprintD(expr) printf(#expr " = %g\n", expr)
#define dprintSIZE_T(expr) printf(#expr " = %zu\n", expr)

#ifndef PI
    #ifndef __USE_MISC
    #define __USE_MISC
    #endif
    #include <math.h>
    #define PI M_PI
#endif

typedef struct {
    int game_is_running;
    float delta_time;
    float elapsed_time;
    float const_fps;
    float fps;
```

```
    float frame_target_time;
    int to_render;
    int to_update;
    size_t previous_frame_time;
    int left_button_pressed;
    int to_limit_fps;
    int to_clear_renderer;

    int space_bar_was_pressed;
    int w_was_pressed;
    int s_was_pressed;
    int a_was_pressed;
    int d_was_pressed;
    int e_was_pressed;
    int q_was_pressed;

    SDL_Window *window;
    int window_w;
    int window_h;
    SDL_Renderer *renderer;
    TTF_Font *font;

    SDL_Surface *window_surface;
    SDL_Texture *window_texture;

    Mat2D_uint32 window_pixels_mat;
    Mat2D inv_z_buffer_mat;

    Offset_zoom_param offset_zoom_param;
} game_state_t;

int initialize_window(game_state_t *game_state);
void setup_window(game_state_t *game_state);
void process_input_window(game_state_t *game_state);
void update_window(game_state_t *game_state);
void render_window(game_state_t *game_state);
void destroy_window(game_state_t *game_state);
void fix_framerate(game_state_t *game_state);
void setup(game_state_t *game_state);
void update(game_state_t *game_state);
void render(game_state_t *game_state);

void check_window_mat_size(game_state_t *game_state);
void copy_mat_to_surface_RGB(game_state_t *game_state);

int main()
{
    game_state_t game_state = {0};

    game_state.game_is_running = 0;
    game_state.delta_time = 0;
    game_state.elapsed_time = 0;
    game_state.const_fps = FPS;
    game_state.fps = 0;
    game_state.frame_target_time = FRAME_TARGET_TIME;

    game_state.space_bar_was_pressed = 0;
    game_state.w_was_pressed = 0;
    game_state.s_was_pressed = 0;
    game_state.a_was_pressed = 0;
    game_state.d_was_pressed = 0;
    game_state.e_was_pressed = 0;
    game_state.q_was_pressed = 0;

    game_state.to_render = 1;
    game_state.to_update = 1;
    game_state.previous_frame_time = 0;
    game_state.left_button_pressed = 0;
    game_state.to_limit_fps = 1;
    game_state.to_clear_renderer = 1;
    game_state.window = NULL;
    game_state.window_w = WINDOW_WIDTH;
    game_state.window_h = WINDOW_HEIGHT;
    game_state.renderer = NULL;
    game_state.font = NULL;

    game_state.offset_zoom_param.zoom_multiplier = 1;

    game_state.game_is_running = !initialize_window(&game_state);

    setup_window(&game_state);

    while (game_state.game_is_running) {
        process_input_window(&game_state);
        if (game_state.to_update) {
            update_window(&game_state);
        }
```

```
        if (game_state.to_render) {
            render_window(&game_state);
        }

    }
    destroy_window(&game_state);

    return 0;
}

int initialize_window(game_state_t *game_state)
{
    if (SDL_Init(SDL_INIT_EVERYTHING) != 0) {
        fprintf(stderr, "%s:%d: [Error] initializing SDL.\n", __FILE__, __LINE__);
        return -1;
    }

    game_state->window = SDL_CreateWindow(NULL,
                                SDL_WINDOWPOS_CENTERED,
                                SDL_WINDOWPOS_CENTERED,
                                game_state->window_w,
                                game_state->window_h,
                                SDL_WINDOW_RESIZABLE
                                );
    if (!game_state->window) {
        fprintf(stderr, "%s:%d: [Error] creating SDL window.\n", __FILE__, __LINE__);
        return -1;
    }

    game_state->renderer = SDL_CreateRenderer(game_state->window, -1, 0);
    if (!game_state->renderer) {
        fprintf(stderr, "%s:%d: [Error] creating SDL renderer.\n", __FILE__, __LINE__);
        return -1;
    }

    if (TTF_Init() == -1) {
        fprintf(stderr, "%s:%d: [Error] initializing SDL_ttf.\n", __FILE__, __LINE__);
        return -1;
    }

    // game_state->font = TTF_OpenFont("./font/Gabriely Black.ttf",32);
    // if (!game_state->font) {
    //     fprintf(stderr, "%s:%d: [Error] loading font.\n", __FILE__, __LINE__);
    //     return -1;
    // }

    (void)game_state;

    return 0;
}

void setup_window(game_state_t *game_state)
{

    game_state->window_surface = SDL_GetWindowSurface(game_state->window);

    game_state->window_pixels_mat = mat2D_alloc_uint32(game_state->window_h, game_state->window_w);
    game_state->inv_z_buffer_mat = mat2D_alloc(game_state->window_h, game_state->window_w);

    /*-----------------------------------*/

    setup(game_state);

}

void process_input_window(game_state_t *game_state)
{
    SDL_Event event;
    while (SDL_PollEvent(&event)) {
        switch (event.type) {
            case SDL_QUIT:
                game_state->game_is_running = 0;
                break;
            case SDL_KEYDOWN:
                if (event.key.keysym.sym == SDLK_ESCAPE) {
                    game_state->game_is_running = 0;
                }
                if (event.key.keysym.sym == SDLK_SPACE) {
                    if (!game_state->space_bar_was_pressed) {
                        game_state->to_render = 0;
                        game_state->to_update = 0;
                        game_state->space_bar_was_pressed = 1;
                        break;
                    }
                    if (game_state->space_bar_was_pressed) {
                        game_state->to_render = 1;
                        game_state->to_update = 1;
```

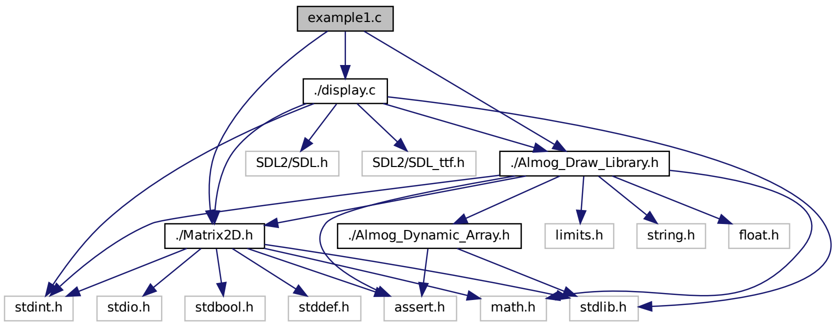
```
                        game_state->previous_frame_time = SDL_GetTicks();
                        game_state->space_bar_was_pressed = 0;
                        break;
                    }
                }
                if (event.key.keysym.sym == SDLK_w) {
                    game_state->offset_zoom_param.offset_y +=
 5/game_state->offset_zoom_param.zoom_multiplier;
                }
                if (event.key.keysym.sym == SDLK_s) {
                    game_state->offset_zoom_param.offset_y -=
 5/game_state->offset_zoom_param.zoom_multiplier;
                }
                if (event.key.keysym.sym == SDLK_a) {
                    game_state->offset_zoom_param.offset_x +=
 5/game_state->offset_zoom_param.zoom_multiplier;
                }
                if (event.key.keysym.sym == SDLK_d) {
                    game_state->offset_zoom_param.offset_x -=
 5/game_state->offset_zoom_param.zoom_multiplier;
                }
                if (event.key.keysym.sym == SDLK_e) {
                    game_state->offset_zoom_param.zoom_multiplier +=
 0.1*game_state->offset_zoom_param.zoom_multiplier;
                    game_state->offset_zoom_param.zoom_multiplier =
 fminf(game_state->offset_zoom_param.zoom_multiplier, ADL_MAX_ZOOM);
                }
                if (event.key.keysym.sym == SDLK_q) {
                    game_state->offset_zoom_param.zoom_multiplier -=
 0.1*game_state->offset_zoom_param.zoom_multiplier;
                    game_state->offset_zoom_param.zoom_multiplier =
 fminf(game_state->offset_zoom_param.zoom_multiplier, ADL_MAX_ZOOM);
                }
                if (event.key.keysym.sym == SDLK_r) {
                    game_state->offset_zoom_param.zoom_multiplier = 1;
                    game_state->offset_zoom_param.offset_x = 0;
                    game_state->offset_zoom_param.offset_y = 0;
                }
                break;
            case SDL_MOUSEBUTTONDOWN:
                if (event.button.button == SDL_BUTTON_LEFT) {
                    game_state->left_button_pressed = 1;
                }
                break;
            case SDL_MOUSEBUTTONUP:
                if (event.button.button == SDL_BUTTON_LEFT) {
                    game_state->left_button_pressed = 0;
                }
                break;
        }
    }
}

void update_window(game_state_t *game_state)
{
    SDL_GetWindowSize(game_state->window, &(game_state->window_w), &(game_state->window_h));

    fix_framerate(game_state);
    game_state->elapsed_time += game_state->delta_time;
    game_state->fps = 1.0f / game_state->delta_time;
    game_state->frame_target_time = 1000/game_state->const_fps;

    char fps_count[100];
    if (!game_state->to_limit_fps) {
        sprintf(fps_count, "dt = %5.02f [ms]", game_state->delta_time*1000);
    } else {
        sprintf(fps_count, "FPS = %5.2f", game_state->fps);
    }

    if (game_state->elapsed_time*10-(int)(game_state->elapsed_time*10) < 0.1) {
        SDL_SetWindowTitle(game_state->window, fps_count);
    }

    check_window_mat_size(game_state);

    /*----------------------------------------------------------------------------*/

    update(game_state);

}

void render_window(game_state_t *game_state)
{
    if (game_state->to_clear_renderer) {
        // SDL_SetRenderDrawColor(game_state->renderer, HexARGB_RGBA(0xFF181818));
        // SDL_RenderClear(game_state->renderer);
        // mat2D_fill(game_state->window_pixels_mat, 0x181818);
```

```c
        memset(game_state->window_pixels_mat.elements, 0x20, sizeof(uint32_t) * game_state->window_pixels_mat.rows * game_state->window_pixels_mat.cols);
        /* not using mat2D_fill but using memset because it is way faster, so the buffer needs to be of 1/z */
        memset(game_state->inv_z_buffer_mat.elements, 0x0, sizeof(double) * game_state->inv_z_buffer_mat.rows * game_state->inv_z_buffer_mat.cols);
    }
    /*----------------------------------------------------------------------*/

    render(game_state);

    /*----------------------------------------------------------------------*/

    copy_mat_to_surface_RGB(game_state);
    SDL_UpdateWindowSurface(game_state->window);

}

void destroy_window(game_state_t *game_state)
{
    mat2D_free_uint32(game_state->window_pixels_mat);

    if (!game_state->window_surface) SDL_FreeSurface(game_state->window_surface);
    if (!game_state->window_texture) SDL_DestroyTexture(game_state->window_texture);

    SDL_DestroyRenderer(game_state->renderer);
    SDL_DestroyWindow(game_state->window);

    SDL_Quit();

    (void)game_state;
}

void fix_framerate(game_state_t *game_state)
{
    int time_ellapsed = SDL_GetTicks() - game_state->previous_frame_time;
    int time_to_wait = game_state->frame_target_time - time_ellapsed;
    if (time_to_wait > 0 && time_to_wait < game_state->frame_target_time) {
        if (game_state->to_limit_fps) {
            SDL_Delay(time_to_wait);
        }
    }
    game_state->delta_time = (SDL_GetTicks() - game_state->previous_frame_time) / 1000.0f;
    game_state->previous_frame_time = SDL_GetTicks();
}

#ifndef SETUP
#define SETUP
void setup(game_state_t *game_state) { (void)game_state; }
#endif

#ifndef UPDATE
#define UPDATE
void update(game_state_t *game_state) { (void)game_state; }
#endif

#ifndef RENDER
#define RENDER
void render(game_state_t *game_state) { (void)game_state; }
#endif

void check_window_mat_size(game_state_t *game_state)
{
    if (game_state->window_h != (int)game_state->window_pixels_mat.rows || game_state->window_w != (int)game_state->window_pixels_mat.cols) {
        mat2D_free_uint32(game_state->window_pixels_mat);
        mat2D_free(game_state->inv_z_buffer_mat);
        SDL_FreeSurface(game_state->window_surface);

        game_state->window_pixels_mat = mat2D_alloc_uint32(game_state->window_h, game_state->window_w);
        game_state->inv_z_buffer_mat = mat2D_alloc(game_state->window_h, game_state->window_w);

        game_state->window_surface = SDL_GetWindowSurface(game_state->window);
    }
}

void copy_mat_to_surface_RGB(game_state_t *game_state)
{
    SDL_LockSurface(game_state->window_surface);

    memcpy(game_state->window_surface->pixels, game_state->window_pixels_mat.elements, sizeof(uint32_t) * game_state->window_pixels_mat.rows * game_state->window_pixels_mat.cols);

    SDL_UnlockSurface(game_state->window_surface);
}
```

## 4.7 example1.c File Reference

```
#include "./Almog_Draw_Library.h"
#include "./display.c"
#include "./Matrix2D.h"
```

Include dependency graph for example1.c:




### Macros

- #define SETUP
- #define UPDATE
- #define RENDER
- #define ALMOG_DRAW_LIBRARY_IMPLEMENTATION
- #define MATRIX2D_IMPLEMENTATION

### Functions

- void setup (game_state_t ∗game_state)
- void update (game_state_t ∗game_state)
- void render (game_state_t ∗game_state)

### Variables

- Figure figure1
- Figure figure2
- Curve points
- Curve points1

### 4.7.1 Macro Definition Documentation

#### 4.7.1.1 ALMOG_DRAW_LIBRARY_IMPLEMENTATION

```
#define ALMOG_DRAW_LIBRARY_IMPLEMENTATION
```

Definition at line 4 of file example1.c.

#### 4.7.1.2 MATRIX2D_IMPLEMENTATION

```
#define MATRIX2D_IMPLEMENTATION
```

Definition at line 7 of file example1.c.

#### 4.7.1.3 RENDER

```
#define RENDER
```

Definition at line 3 of file example1.c.

#### 4.7.1.4 SETUP

```
#define SETUP
```

Definition at line 1 of file example1.c.

#### 4.7.1.5 UPDATE

```
#define UPDATE
```

Definition at line 2 of file example1.c.

### 4.7.2 Function Documentation

#### 4.7.2.1 render()

```
void render (
            game_state_t * game_state )
```

Definition at line 64 of file example1.c.

References adl_character_draw(), adl_curves_plot_on_figure(), ADL_DEFAULT_OFFSET_ZOOM, adl_figure_copy_to_screen(), figure1, figure2, and game_state_t::window_pixels_mat.

#### 4.7.2.2 setup()

```
void setup (
            game_state_t * game_state )
```

Definition at line 15 of file example1.c.

References ada_append, ada_init_array, adl_curve_add_to_figure(), adl_figure_alloc(), Figure::background_color, game_state_t::const_fps, Curve::elements, figure1, figure2, Curve::length, points, points1, Figure::to_draw_axis, and Figure::to_draw_max_min_values.

#### 4.7.2.3 update()

```
void update (
            game_state_t * game_state )
```

Definition at line 60 of file example1.c.

### 4.7.3 Variable Documentation

#### 4.7.3.1 figure1

```
Figure figure1
```

Definition at line 11 of file example1.c.

Referenced by render(), and setup().

#### 4.7.3.2 figure2

```
Figure figure2
```

Definition at line 12 of file example1.c.

Referenced by render(), and setup().

#### 4.7.3.3 points

```
Curve points
```

Definition at line 13 of file example1.c.

Referenced by adl_lines_draw(), adl_lines_loop_draw(), and setup().

#### 4.7.3.4 points1

```
Curve points1
```

Definition at line 14 of file example1.c.

Referenced by setup().

## 4.8 example1.c

```
#define SETUP
#define UPDATE
#define RENDER
#define ALMOG_DRAW_LIBRARY_IMPLEMENTATION
#include "./Almog_Draw_Library.h"
#include "./display.c"
#define MATRIX2D_IMPLEMENTATION
#include "./Matrix2D.h"


Figure figure1;
Figure figure2;
Curve points;
Curve points1;
void setup(game_state_t *game_state)
{
    game_state->const_fps = 30;
    // game_state->to_limit_fps = 0;

    figure1 = adl_figure_alloc(100, 70, (Point){100, 100, 0, 0});
    figure2 = adl_figure_alloc(600, 500, (Point){190, 100, 0, 0});

    ada_init_array(Point, points);
    ada_init_array(Point, points1);
    Point temp_point = (Point){1,1,0,0};
    ada_append(Point, points, temp_point);
    ada_append(Point, points1, temp_point);
    temp_point = (Point){2,2,0,0};
    ada_append(Point, points, temp_point);
    ada_append(Point, points1, temp_point);
    temp_point = (Point){3,1,0,0};
    ada_append(Point, points, temp_point);
    ada_append(Point, points1, temp_point);
    temp_point = (Point){4,10,0,0};
    ada_append(Point, points, temp_point);
```

```
    temp_point = (Point){5,-10,0,0};
    ada_append(Point, points, temp_point);
    temp_point = (Point){3,-20,0,0};
    ada_append(Point, points, temp_point);

    temp_point = (Point){3.5,-10,0,0};
    ada_append(Point, points1, temp_point);

    figure1.background_color = 0xFFFFFFFF;
    figure1.to_draw_axis = true;
    figure1.to_draw_max_min_values = true;

    figure2.background_color = 0xFFFFFFFF;
    figure2.to_draw_axis = true;
    figure2.to_draw_max_min_values = true;

    adl_curve_add_to_figure(&figure1, points.elements, points.length, 0xFFFF0000);
    adl_curve_add_to_figure(&figure2, points.elements, points.length, 0xFFFF0000);

    adl_curve_add_to_figure(&figure1, points1.elements, points1.length, 0xFF0000FF);
    adl_curve_add_to_figure(&figure2, points1.elements, points1.length, 0xFF0000FF);

}

void update(game_state_t *game_state)
{
}

void render(game_state_t *game_state)
{
    adl_curves_plot_on_figure(figure1);
    adl_curves_plot_on_figure(figure2);

    adl_figure_copy_to_screen(game_state->window_pixels_mat, figure1);
    adl_figure_copy_to_screen(game_state->window_pixels_mat, figure2);


    adl_character_draw(game_state->window_pixels_mat, 'A', 50, 100, 700 , 200, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'B', 50, 100, 755 , 200, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'C', 50, 100, 810 , 200, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'D', 50, 100, 865 , 200, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'E', 50, 100, 920 , 200, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'F', 50, 100, 975 , 200, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'G', 50, 100, 1030, 200, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'H', 50, 100, 1085, 200, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'I', 50, 100, 1140, 200, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'J', 50, 100, 1195, 200, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'K', 50, 100, 700 , 305, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'L', 50, 100, 755 , 305, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'M', 50, 100, 810 , 305, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'N', 50, 100, 865 , 305, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'O', 50, 100, 920 , 305, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'P', 50, 100, 975 , 305, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'Q', 50, 100, 1030, 305, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'R', 50, 100, 1085, 305, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'S', 50, 100, 1140, 305, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'T', 50, 100, 1195, 305, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'U', 50, 100, 700 , 410, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'V', 50, 100, 755 , 410, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'W', 50, 100, 810 , 410, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'X', 50, 100, 865 , 410, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, 'Y', 50, 100, 920 , 410, 0xFFFFFFFF,
  ADL_DEFAULT_OFFSET_ZOOM);
```

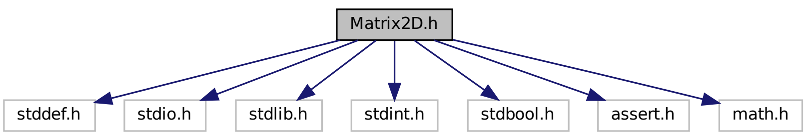
```c
    adl_character_draw(game_state->window_pixels_mat, 'Z', 50, 100, 975 , 410, 0xFFFFFFFF, ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, '.', 50, 100, 1030, 410, 0xFFFFFFFF, ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, ':', 50, 100, 1085, 410, 0xFFFFFFFF, ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, '0', 50, 100, 700 , 515, 0xFFFFFFFF, ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, '1', 50, 100, 755 , 515, 0xFFFFFFFF, ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, '2', 50, 100, 810 , 515, 0xFFFFFFFF, ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, '3', 50, 100, 865 , 515, 0xFFFFFFFF, ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, '4', 50, 100, 920 , 515, 0xFFFFFFFF, ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, '5', 50, 100, 975 , 515, 0xFFFFFFFF, ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, '6', 50, 100, 1030, 515, 0xFFFFFFFF, ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, '7', 50, 100, 1085, 515, 0xFFFFFFFF, ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, '8', 50, 100, 1140, 515, 0xFFFFFFFF, ADL_DEFAULT_OFFSET_ZOOM);
    adl_character_draw(game_state->window_pixels_mat, '9', 50, 100, 1195, 515, 0xFFFFFFFF, ADL_DEFAULT_OFFSET_ZOOM);

}

```

## 4.9 Matrix2D.h File Reference

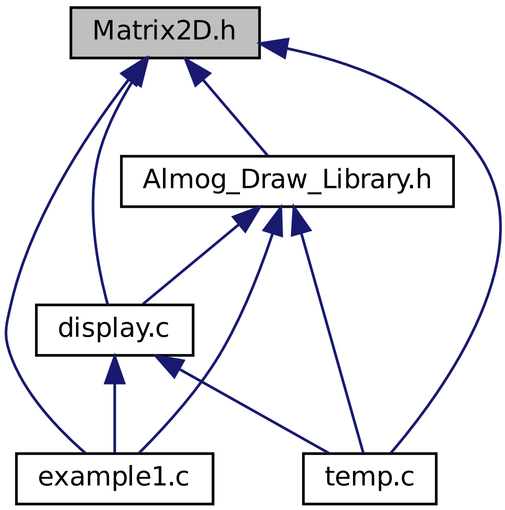
A single-header C library for simple 2D matrix operations on doubles and uint32_t, including allocation, basic arithmetic, linear algebra, and helpers (LUP, inverse, determinant, DCM, etc.).

```c
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include <assert.h>
#include <math.h>
```

Include dependency graph for Matrix2D.h:



This graph shows which files directly or indirectly include this file:




## Classes

- struct Mat2D

  *Dense row-major matrix of doubles.*
- struct Mat2D_uint32

  *Dense row-major matrix of uint32_t.*
- struct Mat2D_Minor

  *A minor "view" into a reference matrix.*

## Macros

- #define MATRIX2D_MALLOC malloc

  *Allocation function used by the library.*
- #define MATRIX2D_ASSERT assert

  *Assertion macro used by the library for parameter validation.*
- #define MAT2D_AT(m, i, j) (m).elements[i ∗ m.stride_r + j]

  *Access element (i, j) of a Mat2D (0-based).*
- #define MAT2D_AT_UINT32(m, i, j) (m).elements[i ∗ m.stride_r + j]

  *Access element (i, j) of a Mat2D_uint32 (0-based).*
- #define __USE_MISC
- #define PI M_PI
- #define MAT2D_MINOR_AT(mm, i, j) MAT2D_AT(mm.ref_mat, mm.rows_list[i], mm.cols_list[j])

  *Access element (i, j) of a Mat2D_Minor (0-based), dereferencing into the underlying reference matrix.*
- #define MAT2D_PRINT(m) mat2D_print(m, #m, 0)

  *Convenience macro to print a matrix with its variable name.*
- #define MAT2D_PRINT_AS_COL(m) mat2D_print_as_col(m, #m, 0)

*Convenience macro to print a matrix as a single column with its name.*
- #define MAT2D_MINOR_PRINT(mm) mat2D_minor_print(mm, #mm, 0)

    *Convenience macro to print a minor with its variable name.*
- #define mat2D_normalize(m) mat2D_mult((m), 1.0 / mat2D_calc_norma((m)))

    *In-place normalization of all elements so that the Frobenius norm becomes 1.*

## Functions

- double mat2D_rand_double (void)

    *Return a pseudo-random double in the range [0, 1].*
- Mat2D mat2D_alloc (size_t rows, size_t cols)

    *Allocate a rows x cols matrix of doubles.*
- Mat2D_uint32 mat2D_alloc_uint32 (size_t rows, size_t cols)

    *Allocate a rows x cols matrix of uint32_t.*
- void mat2D_free (Mat2D m)

    *Free the memory owned by a Mat2D (elements pointer).*
- void mat2D_free_uint32 (Mat2D_uint32 m)

    *Free the memory owned by a Mat2D_uint32 (elements pointer).*
- size_t mat2D_offset2d (Mat2D m, size_t i, size_t j)

    *Compute the linear offset of element (i, j) in a Mat2D.*
- size_t mat2D_offset2d_uint32 (Mat2D_uint32 m, size_t i, size_t j)

    *Compute the linear offset of element (i, j) in a Mat2D_uint32.*
- void mat2D_fill (Mat2D m, double x)

    *Fill all elements of a matrix of doubles with a scalar value.*
- void mat2D_fill_sequence (Mat2D m, double start, double step)

    *Fill a matrix with an arithmetic sequence laid out in row-major order.*
- void mat2D_fill_uint32 (Mat2D_uint32 m, uint32_t x)

    *Fill all elements of a matrix of uint32_t with a scalar value.*
- void mat2D_rand (Mat2D m, double low, double high)

    *Fill a matrix with random doubles in [low, high).*
- void mat2D_dot (Mat2D dst, Mat2D a, Mat2D b)

    *Matrix product: dst = a ∗ b.*
- double mat2D_dot_product (Mat2D a, Mat2D b)

    *Dot product between two vectors.*
- void mat2D_cross (Mat2D dst, Mat2D a, Mat2D b)

    *3D cross product: dst = a x b for 3x1 vectors.*
- void mat2D_add (Mat2D dst, Mat2D a)

    *In-place addition: dst += a.*
- void mat2D_add_row_time_factor_to_row (Mat2D m, size_t des_r, size_t src_r, double factor)

    *Row operation: row(des_r) += factor ∗ row(src_r).*
- void mat2D_sub (Mat2D dst, Mat2D a)

    *In-place subtraction: dst -= a.*
- void mat2D_sub_row_time_factor_to_row (Mat2D m, size_t des_r, size_t src_r, double factor)

    *Row operation: row(des_r) -= factor ∗ row(src_r).*
- void mat2D_mult (Mat2D m, double factor)

    *In-place scalar multiplication: m ∗= factor.*
- void mat2D_mult_row (Mat2D m, size_t r, double factor)

    *In-place row scaling: row(r) ∗= factor.*
- void mat2D_print (Mat2D m, const char ∗name, size_t padding)

    *Print a matrix to stdout with a name and indentation padding.*

- void mat2D_print_as_col (Mat2D m, const char *name, size_t padding)

  *Print a matrix as a flattened column vector to stdout.*
- void mat2D_set_identity (Mat2D m)

  *Set a square matrix to the identity matrix.*
- double mat2D_make_identity (Mat2D m)

  *Reduce a matrix to identity via Gauss-Jordan elimination and return the cumulative scaling factor.*
- void mat2D_set_rot_mat_x (Mat2D m, float angle_deg)

  *Set a 3x3 rotation matrix for rotation about the X-axis.*
- void mat2D_set_rot_mat_y (Mat2D m, float angle_deg)

  *Set a 3x3 rotation matrix for rotation about the Y-axis.*
- void mat2D_set_rot_mat_z (Mat2D m, float angle_deg)

  *Set a 3x3 rotation matrix for rotation about the Z-axis.*
- void mat2D_set_DCM_zyx (Mat2D DCM, float yaw_deg, float pitch_deg, float roll_deg)

  *Build a 3x3 direction cosine matrix (DCM) from Z-Y-X Euler angles.*
- void mat2D_copy (Mat2D des, Mat2D src)

  *Copy all elements from src to des.*
- void mat2D_copy_mat_to_mat_at_window (Mat2D des, Mat2D src, size_t is, size_t js, size_t ie, size_t je)

  *Copy a rectangular window from src into des.*
- void mat2D_get_col (Mat2D des, size_t des_col, Mat2D src, size_t src_col)

  *Copy a column from src into a column of des.*
- void mat2D_add_col_to_col (Mat2D des, size_t des_col, Mat2D src, size_t src_col)

  *Add a source column into a destination column: des[:, des_col] += src[:, src_col].*
- void mat2D_sub_col_to_col (Mat2D des, size_t des_col, Mat2D src, size_t src_col)

  *Subtract a source column from a destination column: des[:, des_col] -= src[:, src_col].*
- void mat2D_swap_rows (Mat2D m, size_t r1, size_t r2)

  *Swap two rows of a matrix in-place.*
- void mat2D_get_row (Mat2D des, size_t des_row, Mat2D src, size_t src_row)

  *Copy a row from src into a row of des.*
- void mat2D_add_row_to_row (Mat2D des, size_t des_row, Mat2D src, size_t src_row)

  *Add a source row into a destination row: des[des_row, :] += src[src_row, :].*
- void mat2D_sub_row_to_row (Mat2D des, size_t des_row, Mat2D src, size_t src_row)

  *Subtract a source row from a destination row: des[des_row, :] -= src[src_row, :].*
- double mat2D_calc_norma (Mat2D m)

  *Compute the Frobenius norm of a matrix, sqrt(sum(m_ij^2)).*
- bool mat2D_mat_is_all_digit (Mat2D m, double digit)

  *Check if all elements of a matrix equal a given digit.*
- bool mat2D_row_is_all_digit (Mat2D m, double digit, size_t r)

  *Check if all elements of a row equal a given digit.*
- bool mat2D_col_is_all_digit (Mat2D m, double digit, size_t c)

  *Check if all elements of a column equal a given digit.*
- double mat2D_det_2x2_mat (Mat2D m)

  *Determinant of a 2x2 matrix.*
- double mat2D_triangulate (Mat2D m)

  *Forward elimination to transform a matrix to upper triangular form.*
- double mat2D_det (Mat2D m)

  *Determinant of an NxN matrix via Gaussian elimination.*
- void mat2D_LUP_decomposition_with_swap (Mat2D src, Mat2D l, Mat2D p, Mat2D u)

  *Compute LUP decomposition: P*A = L*U with L unit diagonal.*
- void mat2D_transpose (Mat2D des, Mat2D src)

  *Transpose a matrix: des = src^T.*
- void mat2D_invert (Mat2D des, Mat2D src)

*Invert a square matrix using Gauss-Jordan elimination.*
- void mat2D_solve_linear_sys_LUP_decomposition (Mat2D A, Mat2D x, Mat2D B)

  *Solve the linear system A x = B using LUP decomposition.*
- Mat2D_Minor mat2D_minor_alloc_fill_from_mat (Mat2D ref_mat, size_t i, size_t j)

  *Allocate a minor view by excluding row i and column j of ref_mat.*
- Mat2D_Minor mat2D_minor_alloc_fill_from_mat_minor (Mat2D_Minor ref_mm, size_t i, size_t j)

  *Allocate a nested minor view from an existing minor by excluding row i and column j of the minor.*
- void mat2D_minor_free (Mat2D_Minor mm)

  *Free the index arrays owned by a minor.*
- void mat2D_minor_print (Mat2D_Minor mm, const char *name, size_t padding)

  *Print a minor matrix to stdout with a name and indentation padding.*
- double mat2D_det_2x2_mat_minor (Mat2D_Minor mm)

  *Determinant of a 2x2 minor.*
- double mat2D_minor_det (Mat2D_Minor mm)

  *Determinant of a minor via recursive expansion by minors.*

### 4.9.1 Detailed Description

A single-header C library for simple 2D matrix operations on doubles and uint32_t, including allocation, basic arithmetic, linear algebra, and helpers (LUP, inverse, determinant, DCM, etc.).

- Storage is contiguous row-major (C-style). The element at row i, column j (0-based) is located at `elements[i * stride_r + j]`.
- Dense matrices of `double` are represented by Mat2D, and dense matrices of `uint32_t` are represented by Mat2D_uint32.
- Some routines assert shape compatibility using MATRIX2D_ASSERT.
- Random number generation uses the C library `rand()`; it is not cryptographically secure.
- Inversion is done via Gauss-Jordan elimination with partial pivoting only when a pivot is zero; this can be numerically unstable for ill-conditioned matrices. See notes below.
- To compile the implementation, define MATRIX2D_IMPLEMENTATION in exactly one translation unit before including this header.

Example: #define MATRIX2D_IMPLEMENTATION #include "matrix2d.h"

**Note**

This one-file library is heavily inspired by Tsoding's nn.h implementation of matrix creation and operations: `https://github.com/tsoding/nn.h` and the video: `https://youtu.be/L1TbWe8bVOc?list=PLpM-Dvs8t0VZPZKggcql-MmjaBdZKeDMw`

**Warning**

Numerical stability:

- There is a set of functions for minors that can be used to compute the determinant, but that approach is factorial in complexity and too slow for larger matrices. This library uses Gaussian elimination instead.
- The inversion function can fail or be unstable if pivot values become very small. Consider preconditioning or using a more robust decomposition (e.g., full pivoting, SVD) for ill-conditioned problems.

Definition in file Matrix2D.h.

## 4.9.2 Macro Definition Documentation

### 4.9.2.1 __USE_MISC

```
#define __USE_MISC
```

Definition at line 151 of file Matrix2D.h.

### 4.9.2.2 MAT2D_AT

```
#define MAT2D_AT(
            m,
            i,
            j ) (m).elements[i * m.stride_r + j]
```

Access element (i, j) of a Mat2D (0-based).

**Warning**

This macro does not perform bounds checking in the fast configuration. Use carefully.

Definition at line 145 of file Matrix2D.h.

### 4.9.2.3 MAT2D_AT_UINT32

```
#define MAT2D_AT_UINT32(
            m,
            i,
            j ) (m).elements[i * m.stride_r + j]
```

Access element (i, j) of a Mat2D_uint32 (0-based).

**Warning**

This macro does not perform bounds checking in the fast configuration. Use carefully.

Definition at line 146 of file Matrix2D.h.

#### 4.9.2.4 MAT2D_MINOR_AT

```
#define MAT2D_MINOR_AT(
              mm,
              i,
              j ) MAT2D_AT(mm.ref_mat, mm.rows_list[i], mm.cols_list[j])
```

Access element (i, j) of a Mat2D_Minor (0-based), dereferencing into the underlying reference matrix.

Definition at line 162 of file Matrix2D.h.

#### 4.9.2.5 MAT2D_MINOR_PRINT

```
#define MAT2D_MINOR_PRINT(
              mm ) mat2D_minor_print(mm, #mm, 0)
```

Convenience macro to print a minor with its variable name.

Definition at line 177 of file Matrix2D.h.

#### 4.9.2.6 mat2D_normalize

```
#define mat2D_normalize(
              m ) mat2D_mult((m), 1.0 / mat2D_calc_norma((m)))
```

In-place normalization of all elements so that the Frobenius norm becomes 1.

Equivalent to: m $*=$ 1.0 / mat2D_calc_norma(m).

Definition at line 184 of file Matrix2D.h.

#### 4.9.2.7 MAT2D_PRINT

```
#define MAT2D_PRINT(
              m ) mat2D_print(m, #m, 0)
```

Convenience macro to print a matrix with its variable name.

Definition at line 167 of file Matrix2D.h.

#### 4.9.2.8 MAT2D_PRINT_AS_COL

```
#define MAT2D_PRINT_AS_COL(
              m ) mat2D_print_as_col(m, #m, 0)
```

Convenience macro to print a matrix as a single column with its name.

Definition at line 172 of file Matrix2D.h.

#### 4.9.2.9 MATRIX2D_ASSERT

```
#define MATRIX2D_ASSERT assert
```

Assertion macro used by the library for parameter validation.

Defaults to C `assert`. Override by defining MATRIX2D_ASSERT before including this header if you want custom behavior.

Definition at line 68 of file Matrix2D.h.

#### 4.9.2.10 MATRIX2D_MALLOC

```
#define MATRIX2D_MALLOC malloc
```

Allocation function used by the library.

Defaults to `malloc`. Override by defining MATRIX2D_MALLOC before including this header if you want to use a custom allocator.

Definition at line 56 of file Matrix2D.h.

#### 4.9.2.11 PI

```
#define PI M_PI
```

Definition at line 154 of file Matrix2D.h.

### 4.9.3 Function Documentation

#### 4.9.3.1 mat2D_add()

```
void mat2D_add (
             Mat2D dst,
             Mat2D a )
```

In-place addition: dst += a.

**Parameters**

| | |
|---|---|
| *dst* | Destination matrix to be incremented. |
| *a* | Summand of same shape as dst. |

**Precondition**

Shapes match.

Definition at line 496 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, MATRIX2D_ASSERT, and Mat2D::rows.

Referenced by adl_arrow_draw().

#### 4.9.3.2 mat2D_add_col_to_col()

```
void mat2D_add_col_to_col (
            Mat2D des,
            size_t des_col,
            Mat2D src,
            size_t src_col )
```

Add a source column into a destination column: des[:, des_col] += src[:, src_col].

**Parameters**

| | |
|---|---|
| *des* | Destination matrix (same row count as src). |
| *des_col* | Column index in destination. |
| *src* | Source matrix. |
| *src_col* | Column index in source. |

Definition at line 828 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, MATRIX2D_ASSERT, and Mat2D::rows.

#### 4.9.3.3 mat2D_add_row_time_factor_to_row()

```
void mat2D_add_row_time_factor_to_row (
            Mat2D m,
            size_t des_r,
            size_t src_r,
            double factor )
```

Row operation: row(des_r) += factor $*$ row(src_r).

**Parameters**

| | |
|---|---|
| *m* | Matrix. |
| *des↩_r* | Destination row index. |
| *src↩_r* | Source row index. |
| *factor* | Scalar multiplier. |

Definition at line 514 of file Matrix2D.h.

References Mat2D::cols, and MAT2D_AT.

#### 4.9.3.4 mat2D_add_row_to_row()

```
void mat2D_add_row_to_row (
            Mat2D des,
            size_t des_row,
            Mat2D src,
            size_t src_row )
```

Add a source row into a destination row: des[des_row, :] += src[src_row, :].

**Parameters**

| | |
|---|---|
| *des* | Destination matrix (same number of columns as src). |
| *des_row* | Row index in destination. |
| *src* | Source matrix. |
| *src_row* | Row index in source. |

Definition at line 897 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, MATRIX2D_ASSERT, and Mat2D::rows.

#### 4.9.3.5 mat2D_alloc()

```
Mat2D mat2D_alloc (
            size_t rows,
            size_t cols )
```

Allocate a rows x cols matrix of doubles.

**Parameters**

| | |
|---|---|
| *rows* | Number of rows (>= 1). |
| *cols* | Number of columns (>= 1). |

**Returns**

A Mat2D with contiguous storage; must be freed with mat2D_free.

**Postcondition**

m.stride_r == cols.

Definition at line 278 of file Matrix2D.h.

References Mat2D::cols, Mat2D::elements, MATRIX2D_ASSERT, MATRIX2D_MALLOC, Mat2D::rows, and Mat2D::stride_r.

Referenced by adl_arrow_draw(), adl_figure_alloc(), check_window_mat_size(), mat2D_det(), mat2D_invert(), mat2D_set_DCM_zyx(), mat2D_solve_linear_sys_LUP_decomposition(), and setup_window().

#### 4.9.3.6 mat2D_alloc_uint32()

```
Mat2D_uint32 mat2D_alloc_uint32 (
             size_t rows,
             size_t cols )
```

Allocate a rows x cols matrix of uint32_t.

**Parameters**

| | |
|---|---|
| *rows* | Number of rows (>= 1). |
| *cols* | Number of columns (>= 1). |

**Returns**

A Mat2D_uint32 with contiguous storage; free with mat2D_free_uint32.

**Postcondition**

m.stride_r == cols.

Definition at line 297 of file Matrix2D.h.

References Mat2D_uint32::cols, Mat2D_uint32::elements, MATRIX2D_ASSERT, MATRIX2D_MALLOC, Mat2D_uint32::rows, and Mat2D_uint32::stride_r.

Referenced by adl_figure_alloc(), check_window_mat_size(), and setup_window().

#### 4.9.3.7 mat2D_calc_norma()

```
double mat2D_calc_norma (
             Mat2D m )
```

Compute the Frobenius norm of a matrix, sqrt(sum(m_ij^2)).

**Parameters**

| | |
|---|---|
| *m* | Matrix. |

**Returns**

Frobenius norm.

Definition at line 931 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, and Mat2D::rows.

#### 4.9.3.8 mat2D_col_is_all_digit()

```
bool mat2D_col_is_all_digit (
            Mat2D m,
            double digit,
            size_t c )
```

Check if all elements of a column equal a given digit.

**Parameters**

| | |
|---|---|
| *m* | Matrix. |
| *digit* | Value to compare. |
| *c* | Column index. |

**Returns**

true if every element equals digit, false otherwise.

Definition at line 985 of file Matrix2D.h.

References Mat2D::cols, and MAT2D_AT.

Referenced by mat2D_det().

#### 4.9.3.9 mat2D_copy()

```
void mat2D_copy (
            Mat2D des,
            Mat2D src )
```

Copy all elements from src to des.

**Parameters**

| | |
|---|---|
| *des* | Destination matrix. |
| *src* | Source matrix. |

**Precondition**

Shapes match.

Definition at line 768 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, MATRIX2D_ASSERT, and Mat2D::rows.

Referenced by adl_arrow_draw(), mat2D_det(), mat2D_invert(), and mat2D_LUP_decomposition_with_swap().

#### 4.9.3.10 mat2D_copy_mat_to_mat_at_window()

```
void mat2D_copy_mat_to_mat_at_window (
            Mat2D des,
            Mat2D src,
            size_t is,
            size_t js,
            size_t ie,
            size_t je )
```

Copy a rectangular window from src into des.

**Parameters**

| | |
|---|---|
| *des* | Destination matrix. Must have size (ie - is + 1) x (je - js + 1). |
| *src* | Source matrix. |
| *is* | Start row index in src (inclusive). |
| *js* | Start column index in src (inclusive). |
| *ie* | End row index in src (inclusive). |
| *je* | End column index in src (inclusive). |

**Precondition**

0 <= is <= ie < src.rows, 0 <= js <= je < src.cols.

Definition at line 790 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, MATRIX2D_ASSERT, and Mat2D::rows.

#### 4.9.3.11 mat2D_cross()

```
void mat2D_cross (
            Mat2D dst,
            Mat2D a,
            Mat2D b )
```

3D cross product: dst = a x b for 3x1 vectors.

**Parameters**

| | |
|---|---|
| *dst* | 3x1 destination vector. |
| *a* | 3x1 input vector. |
| *b* | 3x1 input vector. |

**Precondition**

All matrices have shape 3x1.

Definition at line 479 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, MATRIX2D_ASSERT, and Mat2D::rows.

#### 4.9.3.12 mat2D_det()

```
double mat2D_det (
            Mat2D m )
```

Determinant of an NxN matrix via Gaussian elimination.

**Parameters**

| | |
|---|---|
| *m* | Square matrix. |

**Returns**

det(m).

Copies m internally, triangulates it, and returns the product of diagonal elements (adjusted by any scaling factor as implemented).

Definition at line 1052 of file Matrix2D.h.

References Mat2D::cols, mat2D_alloc(), MAT2D_AT, mat2D_col_is_all_digit(), mat2D_copy(), mat2D_free(), mat2D_row_is_all_digit(), mat2D_triangulate(), MATRIX2D_ASSERT, and Mat2D::rows.

Referenced by mat2D_invert().

#### 4.9.3.13 mat2D_det_2x2_mat()

```
double mat2D_det_2x2_mat (
            Mat2D m )
```

Determinant of a 2x2 matrix.

**Parameters**

| | |
|---|---|
| *m* | Matrix (must be 2x2). |

**Returns**

det(m) = a11 a22 - a12 a21.

Definition at line 1000 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, MATRIX2D_ASSERT, and Mat2D::rows.

#### 4.9.3.14 mat2D_det_2x2_mat_minor()

```
double mat2D_det_2x2_mat_minor (
            Mat2D_Minor mm )
```

Determinant of a 2x2 minor.

**Parameters**

| | |
|---|---|
| *mm* | Minor (must be 2x2). |

**Returns**

det(mm).

Definition at line 1383 of file Matrix2D.h.

References Mat2D_Minor::cols, MAT2D_MINOR_AT, MATRIX2D_ASSERT, and Mat2D_Minor::rows.

Referenced by mat2D_minor_det().

#### 4.9.3.15 mat2D_dot()

```
void mat2D_dot (
            Mat2D dst,
            Mat2D a,
            Mat2D b )
```

Matrix product: dst = a $*$ b.

**Parameters**

| | |
|---|---|
| *dst* | Destination matrix (size a.rows x b.cols). |
| *a* | Left matrix (size a.rows x a.cols). |
| *b* | Right matrix (size a.cols x b.cols). |

**Precondition**

a.cols == b.rows, dst.rows == a.rows, dst.cols == b.cols.

**Postcondition**

dst is overwritten.

Definition at line 424 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, MATRIX2D_ASSERT, and Mat2D::rows.

Referenced by adl_arrow_draw(), mat2D_set_DCM_zyx(), and mat2D_solve_linear_sys_LUP_decomposition().

#### 4.9.3.16 mat2D_dot_product()

```
double mat2D_dot_product (
            Mat2D a,
            Mat2D b )
```

Dot product between two vectors.

**Parameters**

| | |
|---|---|
| *a* | Vector (shape n x 1 or 1 x n). |
| *b* | Vector (same shape as a). |

**Returns**

The scalar dot product sum.

**Precondition**

a.rows == b.rows, a.cols == b.cols, and one dimension equals 1.

Definition at line 450 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, MATRIX2D_ASSERT, and Mat2D::rows.

### 4.9.3.17 mat2D_fill()

```
void mat2D_fill (
            Mat2D m,
            double x )
```

Fill all elements of a matrix of doubles with a scalar value.

**Parameters**

| | |
|---|---|
| *m* | Matrix to fill. |
| *x* | Value to assign to every element. |

Definition at line 362 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, and Mat2D::rows.

Referenced by adl_arrow_draw(), mat2D_invert(), mat2D_LUP_decomposition_with_swap(), and mat2D_solve_linear_sys_LUP_deco

### 4.9.3.18 mat2D_fill_sequence()

```
void mat2D_fill_sequence (
            Mat2D m,
            double start,
            double step )
```

Fill a matrix with an arithmetic sequence laid out in row-major order.

**Parameters**

| | |
|---|---|
| *m* | Matrix to fill. |
| *start* | First value in the sequence. |
| *step* | Increment between consecutive elements. |

Element at linear index k gets value start + step $*$ k.

Definition at line 378 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, mat2D_offset2d(), and Mat2D::rows.

### 4.9.3.19 mat2D_fill_uint32()

```
void mat2D_fill_uint32 (
            Mat2D_uint32 m,
            uint32_t x )
```

Fill all elements of a matrix of uint32_t with a scalar value.

**Parameters**

| | |
|---|---|
| *m* | Matrix to fill. |
| *x* | Value to assign to every element. |

Definition at line 391 of file Matrix2D.h.

References Mat2D_uint32::cols, MAT2D_AT_UINT32, and Mat2D_uint32::rows.

Referenced by adl_2Dscalar_interp_on_figure(), and adl_curves_plot_on_figure().

#### 4.9.3.20 mat2D_free()

```
void mat2D_free (
            Mat2D m )
```

Free the memory owned by a Mat2D (elements pointer).

**Parameters**

| | |
|---|---|
| *m* | Matrix whose elements were allocated via MATRIX2D_MALLOC. |

**Note**

Safe to call with m.elements == NULL.

Definition at line 314 of file Matrix2D.h.

References Mat2D::elements.

Referenced by adl_arrow_draw(), check_window_mat_size(), mat2D_det(), mat2D_invert(), mat2D_set_DCM_zyx(), and mat2D_solve_linear_sys_LUP_decomposition().

#### 4.9.3.21 mat2D_free_uint32()

```
void mat2D_free_uint32 (
            Mat2D_uint32 m )
```

Free the memory owned by a Mat2D_uint32 (elements pointer).

**Parameters**

| | |
|---|---|
| *m* | Matrix whose elements were allocated via MATRIX2D_MALLOC. |

**Note**

Safe to call with m.elements == NULL.

Definition at line 324 of file Matrix2D.h.

References Mat2D_uint32::elements.

Referenced by check_window_mat_size(), and destroy_window().

#### 4.9.3.22 mat2D_get_col()

```
void mat2D_get_col (
            Mat2D des,
            size_t des_col,
            Mat2D src,
            size_t src_col )
```

Copy a column from src into a column of des.

**Parameters**

| | |
|---|---|
| *des* | Destination matrix (same row count as src). |
| *des_col* | Column index in destination. |
| *src* | Source matrix. |
| *src_col* | Column index in source. |

Definition at line 810 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, MATRIX2D_ASSERT, and Mat2D::rows.

#### 4.9.3.23 mat2D_get_row()

```
void mat2D_get_row (
            Mat2D des,
            size_t des_row,
            Mat2D src,
            size_t src_row )
```

Copy a row from src into a row of des.

**Parameters**

| | |
|---|---|
| *des* | Destination matrix (same number of columns as src). |
| *des_row* | Row index in destination. |
| *src* | Source matrix. |
| *src_row* | Row index in source. |

Definition at line 879 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, MATRIX2D_ASSERT, and Mat2D::rows.

#### 4.9.3.24 mat2D_invert()

```
void mat2D_invert (
            Mat2D des,
            Mat2D src )
```

Invert a square matrix using Gauss-Jordan elimination.

**Parameters**

| | |
|---|---|
| *des* | Destination matrix (same shape as src). |
| *src* | Source square matrix. |

**Precondition**

src is square and nonsingular.

If det(src) == 0, prints an error and sets des to all zeros.

**Warning**

May be numerically unstable for ill-conditioned matrices.

Definition at line 1169 of file Matrix2D.h.

References Mat2D::cols, mat2D_alloc(), MAT2D_AT, mat2D_copy(), mat2D_det(), mat2D_fill(), mat2D_free(), mat2D_mult_row(), mat2D_set_identity(), mat2D_sub_row_time_factor_to_row(), mat2D_swap_rows(), MATRIX2D_ASSERT, and Mat2D::rows.

Referenced by mat2D_solve_linear_sys_LUP_decomposition().

#### 4.9.3.25 mat2D_LUP_decomposition_with_swap()

```
void mat2D_LUP_decomposition_with_swap (
            Mat2D src,
            Mat2D l,
            Mat2D p,
            Mat2D u )
```

Compute LUP decomposition: $P*A = L*U$ with L unit diagonal.

**Parameters**

| | |
|---|---|
| *src* | Input matrix A (not modified). |
| *l* | Lower triangular matrix with unit diagonal (output). |
| *p* | Permutation matrix (output). |
| *u* | Upper triangular matrix (output). |

**Precondition**

l, p, u are allocated to match src shape; src is square.

Definition at line 1107 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, mat2D_copy(), mat2D_fill(), mat2D_set_identity(), mat2D_sub_row_time_factor_to_row(), mat2D_swap_rows(), and Mat2D::rows.

Referenced by mat2D_solve_linear_sys_LUP_decomposition().

#### 4.9.3.26 mat2D_make_identity()

```
double mat2D_make_identity (
            Mat2D m )
```

Reduce a matrix to identity via Gauss-Jordan elimination and return the cumulative scaling factor.

**Parameters**

| | |
|---|---|
| *m* | Matrix reduced in-place to identity (if nonsingular). |

**Returns**

The product of row scaling factors applied during elimination.

**Note**

Intended as a helper for determinant-related operations.

**Warning**

Not robust to singular or ill-conditioned matrices.

Definition at line 643 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, mat2D_mult_row(), mat2D_sub_row_time_factor_to_row(), mat2D_swap_rows(), and Mat2D::rows.

#### 4.9.3.27 mat2D_mat_is_all_digit()

```
bool mat2D_mat_is_all_digit (
            Mat2D m,
            double digit )
```

Check if all elements of a matrix equal a given digit.

**Parameters**

| | |
|---|---|
| *m* | Matrix. |
| *digit* | Value to compare. |

**Returns**

true if every element equals digit, false otherwise.

Definition at line 949 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, and Mat2D::rows.

#### 4.9.3.28 mat2D_minor_alloc_fill_from_mat()

```
Mat2D_Minor mat2D_minor_alloc_fill_from_mat (
            Mat2D ref_mat,
            size_t i,
            size_t j )
```

Allocate a minor view by excluding row i and column j of ref_mat.

**Parameters**

| | |
|---|---|
| *ref_mat* | Reference square matrix. |
| *i* | Excluded row index in ref_mat. |
| *j* | Excluded column index in ref_mat. |

**Returns**

A Mat2D_Minor that references ref_mat.

**Note**

Free rows_list and cols_list with mat2D_minor_free when done.

Definition at line 1279 of file Matrix2D.h.

References Mat2D::cols, Mat2D_Minor::cols, Mat2D_Minor::cols_list, MATRIX2D_ASSERT, MATRIX2D_MALLOC, Mat2D_Minor::ref_mat, Mat2D::rows, Mat2D_Minor::rows, Mat2D_Minor::rows_list, and Mat2D_Minor::stride_r.

#### 4.9.3.29 mat2D_minor_alloc_fill_from_mat_minor()

```
Mat2D_Minor mat2D_minor_alloc_fill_from_mat_minor (
            Mat2D_Minor ref_mm,
            size_t i,
            size_t j )
```

Allocate a nested minor view from an existing minor by excluding row i and column j of the minor.

**Parameters**

| | |
|---|---|
| *ref_mm* | Reference minor. |
| *i* | Excluded row index in the minor. |
| *j* | Excluded column index in the minor. |

**Returns**

A new Mat2D_Minor that references the same underlying matrix.

**Note**

Free rows_list and cols_list with mat2D_minor_free when done.

Definition at line 1318 of file Matrix2D.h.

References Mat2D_Minor::cols, Mat2D_Minor::cols_list, MATRIX2D_ASSERT, MATRIX2D_MALLOC, Mat2D_Minor::ref_mat, Mat2D_Minor::rows, Mat2D_Minor::rows_list, and Mat2D_Minor::stride_r.

Referenced by mat2D_minor_det().

#### 4.9.3.30 mat2D_minor_det()

```
double mat2D_minor_det (
            Mat2D_Minor mm )
```

Determinant of a minor via recursive expansion by minors.

**Parameters**

| | |
|---|---|
| *mm* | Square minor. |

**Returns**

det(mm).

**Warning**

Exponential complexity (factorial). Intended for educational or very small matrices only.

Definition at line 1396 of file Matrix2D.h.

References Mat2D_Minor::cols, mat2D_det_2x2_mat_minor(), mat2D_minor_alloc_fill_from_mat_minor(), MAT2D_MINOR_AT, mat2D_minor_free(), MATRIX2D_ASSERT, and Mat2D_Minor::rows.

#### 4.9.3.31 mat2D_minor_free()

```
void mat2D_minor_free (
            Mat2D_Minor mm )
```

Free the index arrays owned by a minor.

**Parameters**

| | |
|---|---|
| *mm* | Minor to free. |

**Note**

After this call, mm.rows_list and mm.cols_list are invalid.

Definition at line 1353 of file Matrix2D.h.

References Mat2D_Minor::cols_list, and Mat2D_Minor::rows_list.

Referenced by mat2D_minor_det().

#### 4.9.3.32 mat2D_minor_print()

```
void mat2D_minor_print (
            Mat2D_Minor mm,
            const char * name,
            size_t padding )
```

Print a minor matrix to stdout with a name and indentation padding.

**Parameters**

| | |
|---|---|
| *mm* | Minor to print. |
| *name* | Label to print. |
| *padding* | Left padding in spaces. |

Definition at line 1365 of file Matrix2D.h.

References Mat2D_Minor::cols, MAT2D_MINOR_AT, and Mat2D_Minor::rows.

#### 4.9.3.33 mat2D_mult()

```
void mat2D_mult (
            Mat2D m,
            double factor )
```

In-place scalar multiplication: m $*=$ factor.

**Parameters**

| | |
|---|---|
| *m* | Matrix. |
| *factor* | Scalar multiplier. |

Definition at line 557 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, and Mat2D::rows.

#### 4.9.3.34 mat2D_mult_row()

```
void mat2D_mult_row (
            Mat2D m,
            size_t r,
            double factor )
```

In-place row scaling: row(r) $*=$ factor.

**Parameters**

| | |
|---|---|
| *m* | Matrix. |
| *r* | Row index. |
| *factor* | Scalar multiplier. |

Definition at line 572 of file Matrix2D.h.

References Mat2D::cols, and MAT2D_AT.

Referenced by mat2D_invert(), and mat2D_make_identity().

#### 4.9.3.35 mat2D_offset2d()

```
size_t mat2D_offset2d (
            Mat2D m,
```

```
size_t i,
size_t j )
```

Compute the linear offset of element (i, j) in a Mat2D.

**Parameters**

| | |
|---|---|
| *m* | Matrix. |
| *i* | Row index (0-based). |
| *j* | Column index (0-based). |

**Returns**

The linear offset i $*$ stride_r + j.

**Precondition**

0 <= i < rows, 0 <= j < cols (asserted).

Definition at line 337 of file Matrix2D.h.

References Mat2D::cols, MATRIX2D_ASSERT, Mat2D::rows, and Mat2D::stride_r.

Referenced by mat2D_fill_sequence().

#### 4.9.3.36 mat2D_offset2d_uint32()

```
size_t mat2D_offset2d_uint32 (
            Mat2D_uint32 m,
            size_t i,
            size_t j )
```

Compute the linear offset of element (i, j) in a Mat2D_uint32.

**Parameters**

| | |
|---|---|
| *m* | Matrix. |
| *i* | Row index (0-based). |
| *j* | Column index (0-based). |

**Returns**

The linear offset i $*$ stride_r + j.

**Precondition**

0 <= i < rows, 0 <= j < cols (asserted).

Definition at line 351 of file Matrix2D.h.

References Mat2D_uint32::cols, MATRIX2D_ASSERT, Mat2D_uint32::rows, and Mat2D_uint32::stride_r.

#### 4.9.3.37 mat2D_print()

```
void mat2D_print (
            Mat2D m,
            const char * name,
            size_t padding )
```

Print a matrix to stdout with a name and indentation padding.

**Parameters**

| | |
|---|---|
| *m* | Matrix to print. |
| *name* | Label to print. |
| *padding* | Left padding in spaces. |

Definition at line 585 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, and Mat2D::rows.

#### 4.9.3.38 mat2D_print_as_col()

```
void mat2D_print_as_col (
            Mat2D m,
            const char * name,
            size_t padding )
```

Print a matrix as a flattened column vector to stdout.

**Parameters**

| | |
|---|---|
| *m* | Matrix to print (flattened in row-major). |
| *name* | Label to print. |
| *padding* | Left padding in spaces. |

Definition at line 604 of file Matrix2D.h.

References Mat2D::cols, Mat2D::elements, and Mat2D::rows.

#### 4.9.3.39 mat2D_rand()

```
void mat2D_rand (
            Mat2D m,
            double low,
            double high )
```

Fill a matrix with random doubles in [low, high).

**Parameters**

| | |
|---|---|
| *m* | Matrix to fill. |
| *low* | Lower bound (inclusive). |
| *high* | Upper bound (exclusive). |

**Precondition**

high > low.

Definition at line 407 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, mat2D_rand_double(), and Mat2D::rows.

#### 4.9.3.40 mat2D_rand_double()

```
double mat2D_rand_double (
            void  )
```

Return a pseudo-random double in the range [0, 1].

**Note**

Uses C library rand() and RAND_MAX. Not cryptographically secure.

Definition at line 266 of file Matrix2D.h.

Referenced by mat2D_rand().

#### 4.9.3.41 mat2D_row_is_all_digit()

```
bool mat2D_row_is_all_digit (
            Mat2D m,
            double digit,
            size_t r )
```

Check if all elements of a row equal a given digit.

**Parameters**

| | |
|---|---|
| *m* | Matrix. |
| *digit* | Value to compare. |
| *r* | Row index. |

**Returns**

true if every element equals digit, false otherwise.

Definition at line 968 of file Matrix2D.h.

References Mat2D::cols, and MAT2D_AT.

Referenced by mat2D_det().

#### 4.9.3.42 mat2D_set_DCM_zyx()

```
void mat2D_set_DCM_zyx (
            Mat2D DCM,
            float yaw_deg,
            float pitch_deg,
            float roll_deg )
```

Build a 3x3 direction cosine matrix (DCM) from Z-Y-X Euler angles.

**Parameters**

| | |
|---|---|
| *DCM* | 3x3 destination matrix. |
| *yaw_deg* | Rotation about Z in degrees. |
| *pitch_deg* | Rotation about Y in degrees. |
| *roll_deg* | Rotation about X in degrees. |

Computes DCM = R_x(roll) ∗ R_y(pitch) ∗ R_z(yaw).

Definition at line 743 of file Matrix2D.h.

References mat2D_alloc(), mat2D_dot(), mat2D_free(), mat2D_set_rot_mat_x(), mat2D_set_rot_mat_y(), and mat2D_set_rot_mat_z().

#### 4.9.3.43 mat2D_set_identity()

```
void mat2D_set_identity (
            Mat2D m )
```

Set a square matrix to the identity matrix.

**Parameters**

| | |
|---|---|
| *m* | Matrix (must be square). |

**Precondition**

m.rows == m.cols.

Definition at line 619 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, MATRIX2D_ASSERT, and Mat2D::rows.

Referenced by mat2D_invert(), mat2D_LUP_decomposition_with_swap(), mat2D_set_rot_mat_x(), mat2D_set_rot_mat_y(), and mat2D_set_rot_mat_z().

#### 4.9.3.44 mat2D_set_rot_mat_x()

```
void mat2D_set_rot_mat_x (
            Mat2D m,
            float angle_deg )
```

Set a 3x3 rotation matrix for rotation about the X-axis.

**Parameters**

| | |
|---|---|
| *m* | 3x3 destination matrix. |
| *angle_deg* | Angle in degrees. |

Definition at line 689 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, mat2D_set_identity(), MATRIX2D_ASSERT, PI, and Mat2D::rows.

Referenced by mat2D_set_DCM_zyx().

#### 4.9.3.45 mat2D_set_rot_mat_y()

```
void mat2D_set_rot_mat_y (
            Mat2D m,
            float angle_deg )
```

Set a 3x3 rotation matrix for rotation about the Y-axis.

**Parameters**

| | |
|---|---|
| *m* | 3x3 destination matrix. |
| *angle_deg* | Angle in degrees. |

Definition at line 706 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, mat2D_set_identity(), MATRIX2D_ASSERT, PI, and Mat2D::rows.

Referenced by mat2D_set_DCM_zyx().

#### 4.9.3.46 mat2D_set_rot_mat_z()

```
void mat2D_set_rot_mat_z (
            Mat2D m,
            float angle_deg )
```

Set a 3x3 rotation matrix for rotation about the Z-axis.

**Parameters**

| | |
|---|---|
| *m* | 3x3 destination matrix. |
| *angle_deg* | Angle in degrees. |

Definition at line 723 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, mat2D_set_identity(), MATRIX2D_ASSERT, PI, and Mat2D::rows.

Referenced by adl_arrow_draw(), and mat2D_set_DCM_zyx().

#### 4.9.3.47 mat2D_solve_linear_sys_LUP_decomposition()

```
void mat2D_solve_linear_sys_LUP_decomposition (
            Mat2D A,
            Mat2D x,
            Mat2D B )
```

Solve the linear system A x = B using LUP decomposition.

**Parameters**

| | |
|---|---|
| *A* | Coefficient matrix (NxN). |
| *x* | Solution vector (N x 1) (output). |
| *B* | Right-hand side vector (N x 1). |

Internally computes LUP and uses explicit inverses of L and U.

**Warning**

> Forming inverses explicitly can be less stable; a forward/backward substitution would be preferable for production-quality code.

Definition at line 1236 of file Matrix2D.h.

References Mat2D::cols, mat2D_alloc(), mat2D_dot(), mat2D_fill(), mat2D_free(), mat2D_invert(), mat2D_LUP_decomposition_with_s
MATRIX2D_ASSERT, and Mat2D::rows.

#### 4.9.3.48 mat2D_sub()

```
void mat2D_sub (
            Mat2D dst,
            Mat2D a )
```

In-place subtraction: dst -= a.

**Parameters**

| | |
|---|---|
| *dst* | Destination matrix to be decremented. |
| *a* | Subtrahend of same shape as dst. |

**Precondition**

Shapes match.

Definition at line 527 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, MATRIX2D_ASSERT, and Mat2D::rows.

Referenced by adl_arrow_draw().

#### 4.9.3.49 mat2D_sub_col_to_col()

```
void mat2D_sub_col_to_col (
            Mat2D des,
            size_t des_col,
            Mat2D src,
            size_t src_col )
```

Subtract a source column from a destination column: des[:, des_col] -= src[:, src_col].

**Parameters**

| | |
|---|---|
| *des* | Destination matrix (same row count as src). |
| *des_col* | Column index in destination. |
| *src* | Source matrix. |
| *src_col* | Column index in source. |

Definition at line 846 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, MATRIX2D_ASSERT, and Mat2D::rows.

#### 4.9.3.50 mat2D_sub_row_time_factor_to_row()

```
void mat2D_sub_row_time_factor_to_row (
            Mat2D m,
```

```
             size_t des_r,
             size_t src_r,
             double factor )
```

Row operation: row(des_r) -= factor ∗ row(src_r).

**Parameters**

| | |
|---|---|
| *m* | Matrix. |
| *des←↩_r* | Destination row index. |
| *src←↩_r* | Source row index. |
| *factor* | Scalar multiplier. |

Definition at line 545 of file Matrix2D.h.

References Mat2D::cols, and MAT2D_AT.

Referenced by mat2D_invert(), mat2D_LUP_decomposition_with_swap(), mat2D_make_identity(), and mat2D_triangulate().

#### 4.9.3.51 mat2D_sub_row_to_row()

```
void mat2D_sub_row_to_row (
             Mat2D des,
             size_t des_row,
             Mat2D src,
             size_t src_row )
```

Subtract a source row from a destination row: des[des_row, :] -= src[src_row, :].

**Parameters**

| | |
|---|---|
| *des* | Destination matrix (same number of columns as src). |
| *des_row* | Row index in destination. |
| *src* | Source matrix. |
| *src_row* | Row index in source. |

Definition at line 915 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, MATRIX2D_ASSERT, and Mat2D::rows.

#### 4.9.3.52 mat2D_swap_rows()

```
void mat2D_swap_rows (
             Mat2D m,
             size_t r1,
             size_t r2 )
```

Swap two rows of a matrix in-place.

**Parameters**

| | |
|---|---|
| *m* | Matrix. |
| *r1* | First row index. |
| *r2* | Second row index. |

Definition at line 863 of file Matrix2D.h.

References Mat2D::cols, and MAT2D_AT.

Referenced by mat2D_invert(), mat2D_LUP_decomposition_with_swap(), mat2D_make_identity(), and mat2D_triangulate().

#### 4.9.3.53 mat2D_transpose()

```
void mat2D_transpose (
            Mat2D des,
            Mat2D src )
```

Transpose a matrix: des = src^T.

**Parameters**

| | |
|---|---|
| *des* | Destination matrix (shape src.cols x src.rows). |
| *src* | Source matrix. |

Definition at line 1149 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, MATRIX2D_ASSERT, and Mat2D::rows.

#### 4.9.3.54 mat2D_triangulate()

```
double mat2D_triangulate (
            Mat2D m )
```

Forward elimination to transform a matrix to upper triangular form.

**Parameters**

| | |
|---|---|
| *m* | Matrix transformed in-place. |

**Returns**

Product of row scaling factors (currently 1 in this implementation).

**Note**

Used as part of determinant computation via triangularization.

**Warning**

Not robust for linearly dependent rows or tiny pivots.

Definition at line 1013 of file Matrix2D.h.

References Mat2D::cols, MAT2D_AT, mat2D_sub_row_time_factor_to_row(), mat2D_swap_rows(), and Mat2D::rows.

Referenced by mat2D_det().

## 4.10 Matrix2D.h

```
#ifndef MATRIX2D_H_
#define MATRIX2D_H_

#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>

#ifndef MATRIX2D_MALLOC
#define MATRIX2D_MALLOC malloc
#endif //MATRIX2D_MALLOC

#ifndef MATRIX2D_ASSERT
#include <assert.h>
#define MATRIX2D_ASSERT assert
#endif //MATRIX2D_ASSERT

typedef struct {
    size_t rows;
    size_t cols;
    size_t stride_r; /* how many element you need to traves to get to the element underneath */
    double *elements;
} Mat2D;

typedef struct {
    size_t rows;
    size_t cols;
    size_t stride_r; /* how many element you need to traves to get to the element underneath */
    uint32_t *elements;
} Mat2D_uint32;

typedef struct {
    size_t rows;
    size_t cols;
    size_t stride_r; /* how many element you need to traves to get to the element underneath */
    size_t *rows_list;
    size_t *cols_list;
    Mat2D ref_mat;
} Mat2D_Minor;

#if 0
#define MAT2D_AT(m, i, j) (m).elements[mat2D_offset2d((m), (i), (j))]
#define MAT2D_AT_UINT32(m, i, j) (m).elements[mat2D_offset2d_uint32((m), (i), (j))]
#else /* use this macro for batter performance but no assertion */
#define MAT2D_AT(m, i, j) (m).elements[i * m.stride_r + j]
#define MAT2D_AT_UINT32(m, i, j) (m).elements[i * m.stride_r + j]
#endif

#ifndef PI
    #ifndef __USE_MISC
    #define __USE_MISC
    #endif
    #include <math.h>
    #define PI M_PI
#endif

```

```c
#define MAT2D_MINOR_AT(mm, i, j) MAT2D_AT(mm.ref_mat, mm.rows_list[i], mm.cols_list[j])
#define MAT2D_PRINT(m) mat2D_print(m, #m, 0)
#define MAT2D_PRINT_AS_COL(m) mat2D_print_as_col(m, #m, 0)
#define MAT2D_MINOR_PRINT(mm) mat2D_minor_print(mm, #mm, 0)
#define mat2D_normalize(m) mat2D_mult((m), 1.0 / mat2D_calc_norma((m)))

double mat2D_rand_double(void);

Mat2D mat2D_alloc(size_t rows, size_t cols);
Mat2D_uint32 mat2D_alloc_uint32(size_t rows, size_t cols);
void mat2D_free(Mat2D m);
void mat2D_free_uint32(Mat2D_uint32 m);
size_t mat2D_offset2d(Mat2D m, size_t i, size_t j);
size_t mat2D_offset2d_uint32(Mat2D_uint32 m, size_t i, size_t j);

void mat2D_fill(Mat2D m, double x);
void mat2D_fill_sequence(Mat2D m, double start, double step);
void mat2D_fill_uint32(Mat2D_uint32 m, uint32_t x);
void mat2D_rand(Mat2D m, double low, double high);

void mat2D_dot(Mat2D dst, Mat2D a, Mat2D b);
double mat2D_dot_product(Mat2D a, Mat2D b);
void mat2D_cross(Mat2D dst, Mat2D a, Mat2D b);

void mat2D_add(Mat2D dst, Mat2D a);
void mat2D_add_row_time_factor_to_row(Mat2D m, size_t des_r, size_t src_r, double factor);

void mat2D_sub(Mat2D dst, Mat2D a);
void mat2D_sub_row_time_factor_to_row(Mat2D m, size_t des_r, size_t src_r, double factor);

void mat2D_mult(Mat2D m, double factor);
void mat2D_mult_row(Mat2D m, size_t r, double factor);

void mat2D_print(Mat2D m, const char *name, size_t padding);
void mat2D_print_as_col(Mat2D m, const char *name, size_t padding);

void mat2D_set_identity(Mat2D m);
double mat2D_make_identity(Mat2D m);
void mat2D_set_rot_mat_x(Mat2D m, float angle_deg);
void mat2D_set_rot_mat_y(Mat2D m, float angle_deg);
void mat2D_set_rot_mat_z(Mat2D m, float angle_deg);
void mat2D_set_DCM_zyx(Mat2D DCM, float yaw_deg, float pitch_deg, float roll_deg);

void mat2D_copy(Mat2D des, Mat2D src);
void mat2D_copy_mat_to_mat_at_window(Mat2D des, Mat2D src, size_t is, size_t js, size_t ie, size_t
    je);

void mat2D_get_col(Mat2D des, size_t des_col, Mat2D src, size_t src_col);
void mat2D_add_col_to_col(Mat2D des, size_t des_col, Mat2D src, size_t src_col);
void mat2D_sub_col_to_col(Mat2D des, size_t des_col, Mat2D src, size_t src_col);

void mat2D_swap_rows(Mat2D m, size_t r1, size_t r2);
void mat2D_get_row(Mat2D des, size_t des_row, Mat2D src, size_t src_row);
void mat2D_add_row_to_row(Mat2D des, size_t des_row, Mat2D src, size_t src_row);
void mat2D_sub_row_to_row(Mat2D des, size_t des_row, Mat2D src, size_t src_row);

double mat2D_calc_norma(Mat2D m);

bool mat2D_mat_is_all_digit(Mat2D m, double digit);
bool mat2D_row_is_all_digit(Mat2D m, double digit, size_t r);
bool mat2D_col_is_all_digit(Mat2D m, double digit, size_t c);

double mat2D_det_2x2_mat(Mat2D m);
double mat2D_triangulate(Mat2D m);
double mat2D_det(Mat2D m);
void mat2D_LUP_decomposition_with_swap(Mat2D src, Mat2D l, Mat2D p, Mat2D u);
void mat2D_transpose(Mat2D des, Mat2D src);
void mat2D_invert(Mat2D des, Mat2D src);
void mat2D_solve_linear_sys_LUP_decomposition(Mat2D A, Mat2D x, Mat2D B);

Mat2D_Minor mat2D_minor_alloc_fill_from_mat(Mat2D ref_mat, size_t i, size_t j);
Mat2D_Minor mat2D_minor_alloc_fill_from_mat_minor(Mat2D_Minor ref_mm, size_t i, size_t j);
void mat2D_minor_free(Mat2D_Minor mm);
void mat2D_minor_print(Mat2D_Minor mm, const char *name, size_t padding);
double mat2D_det_2x2_mat_minor(Mat2D_Minor mm);
double mat2D_minor_det(Mat2D_Minor mm);

#endif // MATRIX2D_H_

#ifdef MATRIX2D_IMPLEMENTATION
#undef MATRIX2D_IMPLEMENTATION


double mat2D_rand_double(void)
{
    return (double) rand() / (double) RAND_MAX;
}
```

```
Mat2D mat2D_alloc(size_t rows, size_t cols)
{
    Mat2D m;
    m.rows = rows;
    m.cols = cols;
    m.stride_r = cols;
    m.elements = (double*)MATRIX2D_MALLOC(sizeof(double)*rows*cols);
    MATRIX2D_ASSERT(m.elements != NULL);

    return m;
}

Mat2D_uint32 mat2D_alloc_uint32(size_t rows, size_t cols)
{
    Mat2D_uint32 m;
    m.rows = rows;
    m.cols = cols;
    m.stride_r = cols;
    m.elements = (uint32_t*)MATRIX2D_MALLOC(sizeof(uint32_t)*rows*cols);
    MATRIX2D_ASSERT(m.elements != NULL);

    return m;
}

void mat2D_free(Mat2D m)
{
    free(m.elements);
}

void mat2D_free_uint32(Mat2D_uint32 m)
{
    free(m.elements);
}

size_t mat2D_offset2d(Mat2D m, size_t i, size_t j)
{
    MATRIX2D_ASSERT(i < m.rows && j < m.cols);
    return i * m.stride_r + j;
}

size_t mat2D_offset2d_uint32(Mat2D_uint32 m, size_t i, size_t j)
{
    MATRIX2D_ASSERT(i < m.rows && j < m.cols);
    return i * m.stride_r + j;
}

void mat2D_fill(Mat2D m, double x)
{
    for (size_t i = 0; i < m.rows; ++i) {
        for (size_t j = 0; j < m.cols; ++j) {
            MAT2D_AT(m, i, j) = x;
        }
    }
}

void mat2D_fill_sequence(Mat2D m, double start, double step) {
    for (size_t i = 0; i < m.rows; i++) {
        for (size_t j = 0; j < m.cols; j++) {
            MAT2D_AT(m, i, j) = start + step * mat2D_offset2d(m, i, j);
        }
    }
}

void mat2D_fill_uint32(Mat2D_uint32 m, uint32_t x)
{
    for (size_t i = 0; i < m.rows; ++i) {
        for (size_t j = 0; j < m.cols; ++j) {
            MAT2D_AT_UINT32(m, i, j) = x;
        }
    }
}

void mat2D_rand(Mat2D m, double low, double high)
{
    for (size_t i = 0; i < m.rows; ++i) {
        for (size_t j = 0; j < m.cols; ++j) {
            MAT2D_AT(m, i, j) = mat2D_rand_double()*(high - low) + low;
        }
    }
}

void mat2D_dot(Mat2D dst, Mat2D a, Mat2D b)
{
    MATRIX2D_ASSERT(a.cols == b.rows);
    MATRIX2D_ASSERT(a.rows == dst.rows);
    MATRIX2D_ASSERT(b.cols == dst.cols);
```

```
    size_t i, j, k;

    for (i = 0; i < dst.rows; i++) {
        for (j = 0; j < dst.cols; j++) {
            MAT2D_AT(dst, i, j) = 0;
            for (k = 0; k < a.cols; k++) {
                MAT2D_AT(dst, i, j) += MAT2D_AT(a, i, k)*MAT2D_AT(b, k, j);
            }
        }
    }

}

double mat2D_dot_product(Mat2D a, Mat2D b)
{
    MATRIX2D_ASSERT(a.rows == b.rows);
    MATRIX2D_ASSERT(a.cols == b.cols);
    MATRIX2D_ASSERT((1 == a.cols && 1 == b.cols) || (1 == a.rows && 1 == b.rows));

    double dot_product = 0;

    if (1 == a.cols) {
        for (size_t i = 0; i < a.rows; i++) {
            dot_product += MAT2D_AT(a, i, 0) * MAT2D_AT(b, i, 0);
        }
    } else {
        for (size_t j = 0; j < a.cols; j++) {
            dot_product += MAT2D_AT(a, 0, j) * MAT2D_AT(b, 0, j);
        }
    }

    return dot_product;

}

void mat2D_cross(Mat2D dst, Mat2D a, Mat2D b)
{
    MATRIX2D_ASSERT(3 == dst.rows && 1 == dst.cols);
    MATRIX2D_ASSERT(3 == a.rows && 1 == a.cols);
    MATRIX2D_ASSERT(3 == b.rows && 1 == b.cols);

    MAT2D_AT(dst, 0, 0) = MAT2D_AT(a, 1, 0) * MAT2D_AT(b, 2, 0) - MAT2D_AT(a, 2, 0) * MAT2D_AT(b, 1, 0);
    MAT2D_AT(dst, 1, 0) = MAT2D_AT(a, 2, 0) * MAT2D_AT(b, 0, 0) - MAT2D_AT(a, 0, 0) * MAT2D_AT(b, 2, 0);
    MAT2D_AT(dst, 2, 0) = MAT2D_AT(a, 0, 0) * MAT2D_AT(b, 1, 0) - MAT2D_AT(a, 1, 0) * MAT2D_AT(b, 0, 0);
}

void mat2D_add(Mat2D dst, Mat2D a)
{
    MATRIX2D_ASSERT(dst.rows == a.rows);
    MATRIX2D_ASSERT(dst.cols == a.cols);
    for (size_t i = 0; i < dst.rows; ++i) {
        for (size_t j = 0; j < dst.cols; ++j) {
            MAT2D_AT(dst, i, j) += MAT2D_AT(a, i, j);
        }
    }
}

void mat2D_add_row_time_factor_to_row(Mat2D m, size_t des_r, size_t src_r, double factor)
{
    for (size_t j = 0; j < m.cols; ++j) {
        MAT2D_AT(m, des_r, j) += factor * MAT2D_AT(m, src_r, j);
    }
}

void mat2D_sub(Mat2D dst, Mat2D a)
{
    MATRIX2D_ASSERT(dst.rows == a.rows);
    MATRIX2D_ASSERT(dst.cols == a.cols);
    for (size_t i = 0; i < dst.rows; ++i) {
        for (size_t j = 0; j < dst.cols; ++j) {
            MAT2D_AT(dst, i, j) -= MAT2D_AT(a, i, j);
        }
    }
}

void mat2D_sub_row_time_factor_to_row(Mat2D m, size_t des_r, size_t src_r, double factor)
{
    for (size_t j = 0; j < m.cols; ++j) {
        MAT2D_AT(m, des_r, j) -= factor * MAT2D_AT(m, src_r, j);
    }
}

void mat2D_mult(Mat2D m, double factor)
```

```
{
    for (size_t i = 0; i < m.rows; ++i) {
        for (size_t j = 0; j < m.cols; ++j) {
            MAT2D_AT(m, i, j) *= factor;
        }
    }
}

void mat2D_mult_row(Mat2D m, size_t r, double factor)
{
    for (size_t j = 0; j < m.cols; ++j) {
        MAT2D_AT(m, r, j) *= factor;
    }
}

void mat2D_print(Mat2D m, const char *name, size_t padding)
{
    printf("%*s%s = [\n", (int) padding, "", name);
    for (size_t i = 0; i < m.rows; ++i) {
        printf("%*s    ", (int) padding, "");
        for (size_t j = 0; j < m.cols; ++j) {
            printf("%9.6f ", MAT2D_AT(m, i, j));
        }
        printf("\n");
    }
    printf("%*s]\n", (int) padding, "");
}

void mat2D_print_as_col(Mat2D m, const char *name, size_t padding)
{
    printf("%*s%s = [\n", (int) padding, "", name);
    for (size_t i = 0; i < m.rows*m.cols; ++i) {
            printf("%*s    ", (int) padding, "");
            printf("%f\n", m.elements[i]);
    }
    printf("%*s]\n", (int) padding, "");
}

void mat2D_set_identity(Mat2D m)
{
    MATRIX2D_ASSERT(m.cols == m.rows);
    for (size_t i = 0; i < m.rows; ++i) {
        for (size_t j = 0; j < m.cols; ++j) {
            MAT2D_AT(m, i, j) = i == j ? 1 : 0;
            // if (i == j) {
            //     MAT2D_AT(m, i, j) = 1;
            // }
            // else {
            //     MAT2D_AT(m, i, j) = 0;
            // }
        }
    }
}

double mat2D_make_identity(Mat2D m)
{
    /* make identity matrix using Gauss elimination */
    /* preforming Gauss elimination: https://en.wikipedia.org/wiki/Gaussian_elimination */
    /* returns the factor multiplying the determinant */

    double factor_to_return = 1;

    for (size_t i = 0; i < (size_t)fmin(m.rows-1, m.cols); i++) {
        /* check if it is the biggest first number (absolute value) */
        size_t biggest_r = i;
        for (size_t index = i; index < m.rows; index++) {
            if (fabs(MAT2D_AT(m, index, index)) > fabs(MAT2D_AT(m, biggest_r, 0))) {
                biggest_r = index;
            }
        }
        if (i != biggest_r) {
            mat2D_swap_rows(m, i, biggest_r);
            factor_to_return *= -1;
        }
        for (size_t j = i+1; j < m.cols; j++) {
            double factor = 1 / MAT2D_AT(m, i, i);
            mat2D_sub_row_time_factor_to_row(m, j, i, MAT2D_AT(m, j, i) * factor);
            mat2D_mult_row(m, i, factor);
            factor_to_return *= factor;
        }
    }
    double factor = 1 / MAT2D_AT(m, m.rows-1, m.cols-1);
    mat2D_mult_row(m, m.rows-1, factor);
    factor_to_return *= factor;
    for (size_t c = m.cols-1; c > 0; c--) {
        for (int r = c-1; r >= 0; r--) {
            double factor = 1 / MAT2D_AT(m, c, c);
```

```
            mat2D_sub_row_time_factor_to_row(m, r, c, MAT2D_AT(m, r, c) * factor);
        }
    }


    return factor_to_return;
}

void mat2D_set_rot_mat_x(Mat2D m, float angle_deg)
{
    MATRIX2D_ASSERT(3 == m.cols && 3 == m.rows);

    float angle_rad = angle_deg * PI / 180;
    mat2D_set_identity(m);
    MAT2D_AT(m, 1, 1) =  cos(angle_rad);
    MAT2D_AT(m, 1, 2) =  sin(angle_rad);
    MAT2D_AT(m, 2, 1) = -sin(angle_rad);
    MAT2D_AT(m, 2, 2) =  cos(angle_rad);
}

void mat2D_set_rot_mat_y(Mat2D m, float angle_deg)
{
    MATRIX2D_ASSERT(3 == m.cols && 3 == m.rows);

    float angle_rad = angle_deg * PI / 180;
    mat2D_set_identity(m);
    MAT2D_AT(m, 0, 0) =  cos(angle_rad);
    MAT2D_AT(m, 0, 2) = -sin(angle_rad);
    MAT2D_AT(m, 2, 0) =  sin(angle_rad);
    MAT2D_AT(m, 2, 2) =  cos(angle_rad);
}

void mat2D_set_rot_mat_z(Mat2D m, float angle_deg)
{
    MATRIX2D_ASSERT(3 == m.cols && 3 == m.rows);

    float angle_rad = angle_deg * PI / 180;
    mat2D_set_identity(m);
    MAT2D_AT(m, 0, 0) =  cos(angle_rad);
    MAT2D_AT(m, 0, 1) =  sin(angle_rad);
    MAT2D_AT(m, 1, 0) = -sin(angle_rad);
    MAT2D_AT(m, 1, 1) =  cos(angle_rad);
}

void mat2D_set_DCM_zyx(Mat2D DCM, float yaw_deg, float pitch_deg, float roll_deg)
{
    Mat2D RotZ = mat2D_alloc(3,3);
    mat2D_set_rot_mat_z(RotZ, yaw_deg);
    Mat2D RotY = mat2D_alloc(3,3);
    mat2D_set_rot_mat_y(RotY, pitch_deg);
    Mat2D RotX = mat2D_alloc(3,3);
    mat2D_set_rot_mat_x(RotX, roll_deg);
    Mat2D temp = mat2D_alloc(3,3);

    mat2D_dot(temp, RotY, RotZ);
    mat2D_dot(DCM, RotX, temp); /* I have a DCM */

    mat2D_free(RotZ);
    mat2D_free(RotY);
    mat2D_free(RotX);
    mat2D_free(temp);
}

void mat2D_copy(Mat2D des, Mat2D src)
{
    MATRIX2D_ASSERT(des.cols == src.cols);
    MATRIX2D_ASSERT(des.rows == src.rows);

    for (size_t i = 0; i < des.rows; ++i) {
        for (size_t j = 0; j < des.cols; ++j) {
            MAT2D_AT(des, i, j) = MAT2D_AT(src, i, j);
        }
    }
}

void mat2D_copy_mat_to_mat_at_window(Mat2D des, Mat2D src, size_t is, size_t js, size_t ie, size_t je)
{
    MATRIX2D_ASSERT(je > js && ie > is);
    MATRIX2D_ASSERT(je-js+1 == des.cols);
    MATRIX2D_ASSERT(ie-is+1 == des.rows);

    for (size_t index = 0; index < des.rows; ++index) {
        for (size_t jndex = 0; jndex < des.cols; ++jndex) {
            MAT2D_AT(des, index, jndex) = MAT2D_AT(src, is+index, js+jndex);
        }
    }
}
```

```c

void mat2D_get_col(Mat2D des, size_t des_col, Mat2D src, size_t src_col)
{
    MATRIX2D_ASSERT(src_col < src.cols);
    MATRIX2D_ASSERT(des.rows == src.rows);
    MATRIX2D_ASSERT(des_col < des.cols);

    for (size_t i = 0; i < des.rows; i++) {
        MAT2D_AT(des, i, des_col) = MAT2D_AT(src, i, src_col);
    }
}

void mat2D_add_col_to_col(Mat2D des, size_t des_col, Mat2D src, size_t src_col)
{
    MATRIX2D_ASSERT(src_col < src.cols);
    MATRIX2D_ASSERT(des.rows == src.rows);
    MATRIX2D_ASSERT(des_col < des.cols);

    for (size_t i = 0; i < des.rows; i++) {
        MAT2D_AT(des, i, des_col) += MAT2D_AT(src, i, src_col);
    }
}

void mat2D_sub_col_to_col(Mat2D des, size_t des_col, Mat2D src, size_t src_col)
{
    MATRIX2D_ASSERT(src_col < src.cols);
    MATRIX2D_ASSERT(des.rows == src.rows);
    MATRIX2D_ASSERT(des_col < des.cols);

    for (size_t i = 0; i < des.rows; i++) {
        MAT2D_AT(des, i, des_col) -= MAT2D_AT(src, i, src_col);
    }
}

void mat2D_swap_rows(Mat2D m, size_t r1, size_t r2)
{
    for (size_t j = 0; j < m.cols; j++) {
        double temp = MAT2D_AT(m, r1, j);
        MAT2D_AT(m, r1, j) = MAT2D_AT(m, r2, j);
        MAT2D_AT(m, r2, j) = temp;
    }
}

void mat2D_get_row(Mat2D des, size_t des_row, Mat2D src, size_t src_row)
{
    MATRIX2D_ASSERT(src_row < src.rows);
    MATRIX2D_ASSERT(des.cols == src.cols);
    MATRIX2D_ASSERT(des_row < des.rows);

    for (size_t j = 0; j < des.cols; j++) {
        MAT2D_AT(des, des_row, j) = MAT2D_AT(src, src_row, j);
    }
}

void mat2D_add_row_to_row(Mat2D des, size_t des_row, Mat2D src, size_t src_row)
{
    MATRIX2D_ASSERT(src_row < src.rows);
    MATRIX2D_ASSERT(des.cols == src.cols);
    MATRIX2D_ASSERT(des_row < des.rows);

    for (size_t j = 0; j < des.cols; j++) {
        MAT2D_AT(des, des_row, j) += MAT2D_AT(src, src_row, j);
    }
}

void mat2D_sub_row_to_row(Mat2D des, size_t des_row, Mat2D src, size_t src_row)
{
    MATRIX2D_ASSERT(src_row < src.rows);
    MATRIX2D_ASSERT(des.cols == src.cols);
    MATRIX2D_ASSERT(des_row < des.rows);

    for (size_t j = 0; j < des.cols; j++) {
        MAT2D_AT(des, des_row, j) -= MAT2D_AT(src, src_row, j);
    }
}

double mat2D_calc_norma(Mat2D m)
{
    double sum = 0;

    for (size_t i = 0; i < m.rows; ++i) {
        for (size_t j = 0; j < m.cols; ++j) {
            sum += MAT2D_AT(m, i, j) * MAT2D_AT(m, i, j);
        }
    }
    return sqrt(sum);
}
```

```
bool mat2D_mat_is_all_digit(Mat2D m, double digit)
{
    for (size_t i = 0; i < m.rows; ++i) {
        for (size_t j = 0; j < m.cols; ++j) {
            if (MAT2D_AT(m, i, j) != digit) {
                return false;
            }
        }
    }
    return true;
}

bool mat2D_row_is_all_digit(Mat2D m, double digit, size_t r)
{
    for (size_t j = 0; j < m.cols; ++j) {
        if (MAT2D_AT(m, r, j) != digit) {
            return false;
        }
    }
    return true;
}

bool mat2D_col_is_all_digit(Mat2D m, double digit, size_t c)
{
    for (size_t i = 0; i < m.cols; ++i) {
        if (MAT2D_AT(m, i, c) != digit) {
            return false;
        }
    }
    return true;
}

double mat2D_det_2x2_mat(Mat2D m)
{
    MATRIX2D_ASSERT(2 == m.cols && 2 == m.rows && "Not a 2x2 matrix");
    return MAT2D_AT(m, 0, 0) * MAT2D_AT(m, 1, 1) - MAT2D_AT(m, 0, 1) * MAT2D_AT(m, 1, 0);
}

double mat2D_triangulate(Mat2D m)
{
    /* preforming Gauss elimination: https://en.wikipedia.org/wiki/Gaussian_elimination */
    /* returns the factor multiplying the determinant */

    double factor_to_return = 1;

    for (size_t i = 0; i < (size_t)fmin(m.rows-1, m.cols); i++) {
        if (!MAT2D_AT(m, i, i)) {   /* swapping only if it is zero */
            /* finding biggest first number (absolute value) */
            size_t biggest_r = i;
            for (size_t index = i; index < m.rows; index++) {
                if (fabs(MAT2D_AT(m, index, i)) > fabs(MAT2D_AT(m, biggest_r, i))) {
                    biggest_r = index;
                }
            }
            if (i != biggest_r) {
                mat2D_swap_rows(m, i, biggest_r);
            }
        }
        for (size_t j = i+1; j < m.cols; j++) {
            double factor = 1 / MAT2D_AT(m, i, i);
            if (!isfinite(factor)) {
                printf("%s:%d: [Error] unable to transfrom into uperr triangular matrix. Probably some of the rows are not independent.\n", __FILE__, __LINE__);
            }
            double mat_value = MAT2D_AT(m, j, i);
            mat2D_sub_row_time_factor_to_row(m, j, i, mat_value * factor);
        }
    }
    return factor_to_return;
}

double mat2D_det(Mat2D m)
{
    MATRIX2D_ASSERT(m.cols == m.rows && "should be a square matrix");

    /* checking if there is a row or column with all zeros */
    /* checking rows */
    for (size_t i = 0; i < m.rows; i++) {
        if (mat2D_row_is_all_digit(m, 0, i)) {
            return 0;
        }
    }
    /* checking cols */
    for (size_t j = 0; j < m.rows; j++) {
        if (mat2D_col_is_all_digit(m, 0, j)) {
            return 0;
```

```
        }
    }

    /* This is an implementation of naive determinant calculation using minors. This is too slow */

    // double det = 0;
    // /* TODO: finding beast row or col? */
    // for (size_t i = 0, j = 0; i < m.rows; i++) { /* first column */
    //     if (MAT2D_AT(m, i, j) < 1e-10) continue;
    //     Mat2D_Minor sub_mm = mat2D_minor_alloc_fill_from_mat(m, i, j);
    //     int factor = (i+j)%2 ? -1 : 1;
    //     if (sub_mm.cols != 2) {
    //         MATRIX2D_ASSERT(sub_mm.cols == sub_mm.rows && "should be a square matrix");
    //         det += MAT2D_AT(m, i, j) * (factor) * mat2D_minor_det(sub_mm);
    //     } else if (sub_mm.cols == 2 && sub_mm.rows == 2) {
    //         det += MAT2D_AT(m, i, j) * (factor) * mat2D_det_2x2_mat_minor(sub_mm);;
    //     }
    //     mat2D_minor_free(sub_mm);
    // }

    Mat2D temp_m = mat2D_alloc(m.rows, m.cols);
    mat2D_copy(temp_m, m);
    double factor = mat2D_triangulate(temp_m);
    double diag_mul = 1;
    for (size_t i = 0; i < temp_m.rows; i++) {
        diag_mul *= MAT2D_AT(temp_m, i, i);
    }
    mat2D_free(temp_m);

    return diag_mul / factor;
}

void mat2D_LUP_decomposition_with_swap(Mat2D src, Mat2D l, Mat2D p, Mat2D u)
{
    /* performing LU decomposition Following the Wikipedia page:
  https://en.wikipedia.org/wiki/LU_decomposition */

    mat2D_copy(u, src);
    mat2D_set_identity(p);
    mat2D_fill(l, 0);

    for (size_t i = 0; i < (size_t)fmin(u.rows-1, u.cols); i++) {
        if (!MAT2D_AT(u, i, i)) {    /* swapping only if it is zero */
            /* finding biggest first number (absolute value) */
            size_t biggest_r = i;
            for (size_t index = i; index < u.rows; index++) {
                if (fabs(MAT2D_AT(u, index, i)) > fabs(MAT2D_AT(u, biggest_r, i))) {
                    biggest_r = index;
                }
            }
            if (i != biggest_r) {
                mat2D_swap_rows(u, i, biggest_r);
                mat2D_swap_rows(p, i, biggest_r);
                mat2D_swap_rows(l, i, biggest_r);
            }
        }
        for (size_t j = i+1; j < u.cols; j++) {
            double factor = 1 / MAT2D_AT(u, i, i);
            if (!isfinite(factor)) {
                printf("%s:%d: [Error] unable to transfrom into uper triangular matrix. Probably some
  of the rows are not independent.\n", __FILE__, __LINE__);
            }
            double mat_value = MAT2D_AT(u, j, i);
            mat2D_sub_row_time_factor_to_row(u, j, i, mat_value * factor);
            MAT2D_AT(l, j, i) = mat_value * factor;
        }
        MAT2D_AT(l, i, i) = 1;
    }
    MAT2D_AT(l, l.rows-1, l.cols-1) = 1;
}

void mat2D_transpose(Mat2D des, Mat2D src)
{
    MATRIX2D_ASSERT(des.cols == src.rows);
    MATRIX2D_ASSERT(des.rows == src.cols);

    for (size_t index = 0; index < des.rows; ++index) {
        for (size_t jndex = 0; jndex < des.cols; ++jndex) {
            MAT2D_AT(des, index, jndex) = MAT2D_AT(src, jndex, index);
        }
    }
}

void mat2D_invert(Mat2D des, Mat2D src)
{
    MATRIX2D_ASSERT(src.cols == src.rows && "should be an NxN matrix");
    MATRIX2D_ASSERT(des.cols == src.cols && des.rows == des.cols);
```

```

    Mat2D m = mat2D_alloc(src.rows, src.cols);
    mat2D_copy(m, src);

    mat2D_set_identity(des);

    if (!mat2D_det(m)) {
        mat2D_fill(des, 0);
        printf("%s:%d: [Error] Can't invert the matrix. Determinant is zero! Set the inverse matrix to all zeros\n", __FILE__, __LINE__);
        return;
    }

    for (size_t i = 0; i < (size_t)fmin(m.rows-1, m.cols); i++) {
        if (!MAT2D_AT(m, i, i)) {   /* swapping only if it is zero */
            /* finding biggest first number (absolute value) */
            size_t biggest_r = i;
            for (size_t index = i; index < m.rows; index++) {
                if (fabs(MAT2D_AT(m, index, i)) > fabs(MAT2D_AT(m, biggest_r, i))) {
                    biggest_r = index;
                }
            }
            if (i != biggest_r) {
                mat2D_swap_rows(m, i, biggest_r);
                mat2D_swap_rows(des, i, biggest_r);
                printf("%s:%d: [INFO] swapping row %zu with row %zu.\n", __FILE__, __LINE__, i, biggest_r);
            } else {
                MATRIX2D_ASSERT(0 && "can't inverse");
            }
        }
        for (size_t j = i+1; j < m.cols; j++) {
            double factor = 1 / MAT2D_AT(m, i, i);
            double mat_value = MAT2D_AT(m, j, i);
            mat2D_sub_row_time_factor_to_row(m, j, i, mat_value * factor);
            mat2D_mult_row(m, i, factor);

            mat2D_sub_row_time_factor_to_row(des, j, i, mat_value * factor);
            mat2D_mult_row(des, i, factor);
        }
    }
    double factor = 1 / MAT2D_AT(m, m.rows-1, m.cols-1);
    mat2D_mult_row(m, m.rows-1, factor);
    mat2D_mult_row(des, des.rows-1, factor);
    for (size_t c = m.cols-1; c > 0; c--) {
        for (int r = c-1; r >= 0; r--) {
            double factor = 1 / MAT2D_AT(m, c, c);
            double mat_value = MAT2D_AT(m, r, c);
            mat2D_sub_row_time_factor_to_row(m, r, c, mat_value * factor);
            mat2D_sub_row_time_factor_to_row(des, r, c, mat_value * factor);
        }
    }

    mat2D_free(m);
}

void mat2D_solve_linear_sys_LUP_decomposition(Mat2D A, Mat2D x, Mat2D B)
{
    MATRIX2D_ASSERT(A.cols == x.rows);
    MATRIX2D_ASSERT(1 == x.cols);
    MATRIX2D_ASSERT(A.rows == B.rows);
    MATRIX2D_ASSERT(1 == B.cols);

    Mat2D y     = mat2D_alloc(x.rows, x.cols);
    Mat2D l     = mat2D_alloc(A.rows, A.cols);
    Mat2D p     = mat2D_alloc(A.rows, A.cols);
    Mat2D u     = mat2D_alloc(A.rows, A.cols);
    Mat2D inv_l = mat2D_alloc(l.rows, l.cols);
    Mat2D inv_u = mat2D_alloc(u.rows, u.cols);

    mat2D_LUP_decomposition_with_swap(A, l, p, u);

    mat2D_invert(inv_l, l);
    mat2D_invert(inv_u, u);

    mat2D_fill(x, 0);   /* x here is only a temp mat*/
    mat2D_fill(y, 0);
    mat2D_dot(x, p, B);
    mat2D_dot(y, inv_l, x);

    mat2D_fill(x, 0);
    mat2D_dot(x, inv_u, y);

    mat2D_free(y);
    mat2D_free(l);
    mat2D_free(p);
    mat2D_free(u);
```

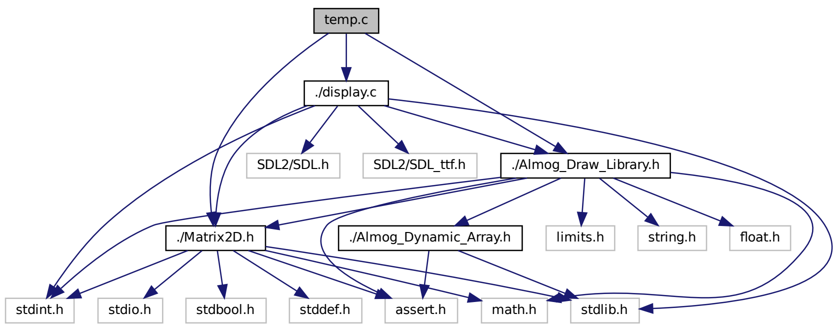
```
    mat2D_free(inv_l);
    mat2D_free(inv_u);
}

Mat2D_Minor mat2D_minor_alloc_fill_from_mat(Mat2D ref_mat, size_t i, size_t j)
{
    MATRIX2D_ASSERT(ref_mat.cols == ref_mat.rows && "minor is defined only for square matrix");

    Mat2D_Minor mm;
    mm.cols = ref_mat.cols-1;
    mm.rows = ref_mat.rows-1;
    mm.stride_r = ref_mat.cols-1;
    mm.cols_list = (size_t*)MATRIX2D_MALLOC(sizeof(double)*(ref_mat.cols-1));
    mm.rows_list = (size_t*)MATRIX2D_MALLOC(sizeof(double)*(ref_mat.rows-1));
    mm.ref_mat = ref_mat;

    MATRIX2D_ASSERT(mm.cols_list != NULL && mm.rows_list != NULL);

    for (size_t index = 0, temp_index = 0; index < ref_mat.rows; index++) {
        if (index != i) {
            mm.rows_list[temp_index] = index;
            temp_index++;
        }
    }
    for (size_t jndex = 0, temp_jndex = 0; jndex < ref_mat.rows; jndex++) {
        if (jndex != j) {
            mm.cols_list[temp_jndex] = jndex;
            temp_jndex++;
        }
    }

    return mm;
}

Mat2D_Minor mat2D_minor_alloc_fill_from_mat_minor(Mat2D_Minor ref_mm, size_t i, size_t j)
{
    MATRIX2D_ASSERT(ref_mm.cols == ref_mm.rows && "minor is defined only for square matrix");

    Mat2D_Minor mm;
    mm.cols = ref_mm.cols-1;
    mm.rows = ref_mm.rows-1;
    mm.stride_r = ref_mm.cols-1;
    mm.cols_list = (size_t*)MATRIX2D_MALLOC(sizeof(double)*(ref_mm.cols-1));
    mm.rows_list = (size_t*)MATRIX2D_MALLOC(sizeof(double)*(ref_mm.rows-1));
    mm.ref_mat = ref_mm.ref_mat;

    MATRIX2D_ASSERT(mm.cols_list != NULL && mm.rows_list != NULL);

    for (size_t index = 0, temp_index = 0; index < ref_mm.rows; index++) {
        if (index != i) {
            mm.rows_list[temp_index] = ref_mm.rows_list[index];
            temp_index++;
        }
    }
    for (size_t jndex = 0, temp_jndex = 0; jndex < ref_mm.rows; jndex++) {
        if (jndex != j) {
            mm.cols_list[temp_jndex] = ref_mm.cols_list[jndex];
            temp_jndex++;
        }
    }

    return mm;
}

void mat2D_minor_free(Mat2D_Minor mm)
{
    free(mm.cols_list);
    free(mm.rows_list);
}

void mat2D_minor_print(Mat2D_Minor mm, const char *name, size_t padding)
{
    printf("%*s%s = [\n", (int) padding, "", name);
    for (size_t i = 0; i < mm.rows; ++i) {
        printf("%*s    ", (int) padding, "");
        for (size_t j = 0; j < mm.cols; ++j) {
            printf("%f ", MAT2D_MINOR_AT(mm, i, j));
        }
        printf("\n");
    }
    printf("%*s]\n", (int) padding, "");
}

double mat2D_det_2x2_mat_minor(Mat2D_Minor mm)
{
    MATRIX2D_ASSERT(2 == mm.cols && 2 == mm.rows && "Not a 2x2 matrix");
    return MAT2D_MINOR_AT(mm, 0, 0) * MAT2D_MINOR_AT(mm, 1, 1) - MAT2D_MINOR_AT(mm, 0, 1) *
```

```
    MAT2D_MINOR_AT(mm, 1, 0);
}

double mat2D_minor_det(Mat2D_Minor mm)
{
    MATRIX2D_ASSERT(mm.cols == mm.rows && "should be a square matrix");

    double det = 0;
    /* TODO: finding beast row or col? */
    for (size_t i = 0, j = 0; i < mm.rows; i++) { /* first column */
        if (MAT2D_MINOR_AT(mm, i, j) < 1e-10) continue;
        Mat2D_Minor sub_mm = mat2D_minor_alloc_fill_from_mat_minor(mm, i, j);
        int factor = (i+j)%2 ? -1 : 1;
        if (sub_mm.cols != 2) {
            MATRIX2D_ASSERT(sub_mm.cols == sub_mm.rows && "should be a square matrix");
            det += MAT2D_MINOR_AT(mm, i, j) * (factor) * mat2D_minor_det(sub_mm);
        } else if (sub_mm.cols == 2 && sub_mm.rows == 2) {
            det += MAT2D_MINOR_AT(mm, i, j) * (factor) * mat2D_det_2x2_mat_minor(sub_mm);;
        }
        mat2D_minor_free(sub_mm);
    }
    return det;
}


#endif // MATRIX2D_IMPLEMENTATION
```

## 4.11 temp.c File Reference

```
#include "./Almog_Draw_Library.h"
#include "./display.c"
#include "./Matrix2D.h"
```

Include dependency graph for temp.c:




### Macros

- #define SETUP
- #define UPDATE
- #define RENDER
- #define ALMOG_DRAW_LIBRARY_IMPLEMENTATION
- #define MATRIX2D_IMPLEMENTATION

### Functions

- void setup (game_state_t *game_state)
- void update (game_state_t *game_state)
- void render (game_state_t *game_state)

## Variables

- Quad quad1
- Tri tri

### 4.11.1 Macro Definition Documentation

#### 4.11.1.1 ALMOG_DRAW_LIBRARY_IMPLEMENTATION

```
#define ALMOG_DRAW_LIBRARY_IMPLEMENTATION
```

Definition at line 4 of file temp.c.

#### 4.11.1.2 MATRIX2D_IMPLEMENTATION

```
#define MATRIX2D_IMPLEMENTATION
```

Definition at line 7 of file temp.c.

#### 4.11.1.3 RENDER

```
#define RENDER
```

Definition at line 3 of file temp.c.

#### 4.11.1.4 SETUP

```
#define SETUP
```

Definition at line 1 of file temp.c.

#### 4.11.1.5 UPDATE

```
#define UPDATE
```

Definition at line 2 of file temp.c.

## 4.11.2 Function Documentation

### 4.11.2.1 render()

```
void render (
            game_state_t * game_state )
```

Definition at line 50 of file temp.c.

References adl_linear_map(), adl_quad_draw(), adl_quad_fill_interpolate_color_mean_value(), adl_tri_draw(), adl_tri_fill_Pinedas_rasterizer_interpolate_color(), Mat2D::cols, game_state_t::inv_z_buffer_mat, MAT2D_AT, MAT2D_AT_UINT32, game_state_t::offset_zoom_param, quad1, RGB_hexRGB, Mat2D::rows, tri, and game_state_t::window_pixels_mat.

### 4.11.2.2 setup()

```
void setup (
            game_state_t * game_state )
```

Definition at line 13 of file temp.c.

References Tri::colors, Quad::colors, Tri::light_intensity, Quad::light_intensity, Tri::points, Quad::points, quad1, Tri::to_draw, Quad::to_draw, game_state_t::to_limit_fps, and tri.

### 4.11.2.3 update()

```
void update (
            game_state_t * game_state )
```

Definition at line 45 of file temp.c.

## 4.11.3 Variable Documentation

### 4.11.3.1 quad1

```
Quad quad1
```

Definition at line 10 of file temp.c.

Referenced by render(), and setup().

#### 4.11.3.2 tri

```
Tri tri
```

Definition at line 11 of file temp.c.

Referenced by adl_tri_draw(), adl_tri_fill_Pinedas_rasterizer(), adl_tri_fill_Pinedas_rasterizer_interpolate_color(), adl_tri_fill_Pinedas_rasterizer_interpolate_normal(), adl_tri_mesh_draw(), adl_tri_mesh_fill_Pinedas_rasterizer(), adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_color(), adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_normal(), render(), and setup().

## 4.12 temp.c

```
00001 #define SETUP
00002 #define UPDATE
00003 #define RENDER
00004 #define ALMOG_DRAW_LIBRARY_IMPLEMENTATION
00005 #include "./Almog_Draw_Library.h"
00006 #include "./display.c"
00007 #define MATRIX2D_IMPLEMENTATION
00008 #include "./Matrix2D.h"
00009
00010 Quad quad1;
00011 Tri tri;
00012
00013 void setup(game_state_t *game_state)
00014 {
00015     // game_state->const_fps = 30;
00016     game_state->to_limit_fps = 0;
00017
00018     quad1.points[3] = (Point){200, 100, 1, 1};
00019     quad1.points[2] = (Point){600, 50 , 1, 1};
00020     quad1.points[1] = (Point){200, 700, 1, 1};
00021     quad1.points[0] = (Point){100, 300, 1, 1};
00022     quad1.to_draw = true;
00023     quad1.light_intensity[0] = 1;
00024     quad1.light_intensity[1] = 1;
00025     quad1.light_intensity[2] = 1;
00026     quad1.light_intensity[3] = 1;
00027     quad1.colors[0] = 0xFFFFFFFF;
00028     quad1.colors[1] = 0xFF0000FF;
00029     quad1.colors[2] = 0xFF00FF00;
00030     quad1.colors[3] = 0xFFFF0000;
00031
00032     tri.points[2] = (Point){750 , 100, 1, 1};
00033     tri.points[1] = (Point){1250, 700, 1, 1};
00034     tri.points[0] = (Point){650 , 500, 1, 1};
00035     tri.to_draw = true;
00036     tri.light_intensity[0] = 1;
00037     tri.light_intensity[1] = 1;
00038     tri.light_intensity[2] = 1;
00039     tri.colors[0] = 0xFFFFFFFF;
00040     tri.colors[1] = 0xFF0000FF;
00041     tri.colors[2] = 0xFF00FF00;
00042
00043 }
00044
00045 void update(game_state_t *game_state)
00046 {
00047     SDL_Delay(1);
00048 }
00049
00050 void render(game_state_t *game_state)
00051 {
00052     adl_quad_fill_interpolate_color_mean_value(game_state->window_pixels_mat,
       game_state->inv_z_buffer_mat, quad1, game_state->offset_zoom_param);
00053     adl_quad_draw(game_state->window_pixels_mat, game_state->inv_z_buffer_mat, quad1, 0xFF000000,
       game_state->offset_zoom_param);
00054
00055     adl_tri_fill_Pinedas_rasterizer_interpolate_color(game_state->window_pixels_mat,
       game_state->inv_z_buffer_mat, tri, game_state->offset_zoom_param);
00056     adl_tri_draw(game_state->window_pixels_mat, tri, 0xff000000, game_state->offset_zoom_param);
00057
00058     #if 0
00059     Mat2D inv_z_buffer = game_state->inv_z_buffer_mat;
00060     double max_inv_z = 0;
00061     double min_inv_z = DBL_MAX;
00062     for (size_t i = 0; i < inv_z_buffer.rows; i++) {
```

```
        for (size_t j = 0; j < inv_z_buffer.cols; j++) {
            if (MAT2D_AT(inv_z_buffer, i, j) > max_inv_z) {
                max_inv_z = MAT2D_AT(inv_z_buffer, i, j);
            }
            if (MAT2D_AT(inv_z_buffer, i, j) < min_inv_z && MAT2D_AT(inv_z_buffer, i, j) > 0) {
                min_inv_z = MAT2D_AT(inv_z_buffer, i, j);
            }
        }
    }
    for (size_t i = 0; i < inv_z_buffer.rows; i++) {
        for (size_t j = 0; j < inv_z_buffer.cols; j++) {
            double z_fraq = MAT2D_AT(inv_z_buffer, i, j);
            z_fraq = fmax(z_fraq, min_inv_z);
            z_fraq = adl_linear_map(z_fraq, min_inv_z, max_inv_z, 0.1, 1);
            uint32_t color = RGB_hexRGB(0xFF*z_fraq, 0xFF*z_fraq, 0xFF*z_fraq);
            MAT2D_AT_UINT32(game_state->window_pixels_mat, i, j) = color;
        }
    }
    #endif
}

```

# Index