Almog Engine

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:

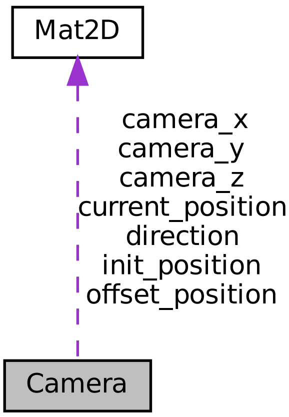
Chapter 3

Class Documentation

3.1 Camera Struct Reference

`#include <Almog_Engine.h>`

Collaboration diagram for Camera:



Public Attributes

- Mat2D init_position
- Mat2D current_position
- Mat2D offset_position
- Mat2D direction
- float z_near
- float z_far
- float fov_deg
- float aspect_ratio

- float roll_offset_deg
- float pitch_offset_deg
- float yaw_offset_deg
- Mat2D camera_x
- Mat2D camera_y
- Mat2D camera_z

3.1.1 Detailed Description

Definition at line 144 of file Almog_Engine.h.

3.1.2 Member Data Documentation

3.1.2.1 aspect_ratio

```
float Camera::aspect_ratio
```

Definition at line 152 of file Almog_Engine.h.

Referenced by ae_camera_init(), ae_scene_init(), check_window_mat_size(), and update().

3.1.2.2 camera_x

```
Mat2D Camera::camera_x
```

Definition at line 156 of file Almog_Engine.h.

Referenced by ae_camera_free(), ae_camera_init(), ae_camera_reset_pos(), and ae_view_mat_set().

3.1.2.3 camera_y

```
Mat2D Camera::camera_y
```

Definition at line 157 of file Almog_Engine.h.

Referenced by ae_camera_free(), ae_camera_init(), ae_camera_reset_pos(), and ae_view_mat_set().

3.1.2.4 camera_z

`Mat2D Camera::camera_z`

Definition at line 158 of file Almog_Engine.h.

Referenced by ae_camera_free(), ae_camera_init(), ae_camera_reset_pos(), and ae_view_mat_set().

3.1.2.5 current_position

`Mat2D Camera::current_position`

Definition at line 146 of file Almog_Engine.h.

Referenced by ae_camera_free(), ae_camera_init(), ae_camera_reset_pos(), ae_quad_calc_light_intensity(), ae_quad_project_world2screen(), ae_tri_calc_light_intensity(), ae_tri_project_world2screen(), and ae_view_mat_set().

3.1.2.6 direction

`Mat2D Camera::direction`

Definition at line 148 of file Almog_Engine.h.

Referenced by ae_camera_free(), ae_camera_init(), and ae_view_mat_set().

3.1.2.7 fov_deg

`float Camera::fov_deg`

Definition at line 151 of file Almog_Engine.h.

Referenced by ae_camera_init(), ae_scene_init(), and update().

3.1.2.8 init_position

`Mat2D Camera::init_position`

Definition at line 145 of file Almog_Engine.h.

Referenced by ae_camera_free(), ae_camera_init(), and ae_camera_reset_pos().

3.1.2.9 offset_position

```
Mat2D Camera::offset_position
```

Definition at line 147 of file Almog_Engine.h.

Referenced by ae_camera_free(), ae_camera_init(), ae_camera_reset_pos(), ae_view_mat_set(), and process_input_window().

3.1.2.10 pitch_offset_deg

```
float Camera::pitch_offset_deg
```

Definition at line 154 of file Almog_Engine.h.

Referenced by ae_camera_init(), ae_camera_reset_pos(), ae_view_mat_set(), and process_input_window().

3.1.2.11 roll_offset_deg

```
float Camera::roll_offset_deg
```

Definition at line 153 of file Almog_Engine.h.

Referenced by ae_camera_init(), ae_camera_reset_pos(), ae_view_mat_set(), and process_input_window().

3.1.2.12 yaw_offset_deg

```
float Camera::yaw_offset_deg
```

Definition at line 155 of file Almog_Engine.h.

Referenced by ae_camera_init(), ae_camera_reset_pos(), and ae_view_mat_set().

3.1.2.13 z_far

```
float Camera::z_far
```

Definition at line 150 of file Almog_Engine.h.

Referenced by ae_camera_init(), ae_scene_init(), and update().

3.1.2.14 z_near

```
float Camera::z_near
```

Definition at line 149 of file Almog_Engine.h.

Referenced by ae_camera_init(), ae_line_project_world2screen(), ae_quad_project_world2screen(), ae_scene_init(), ae_tri_project_world2screen(), and update().

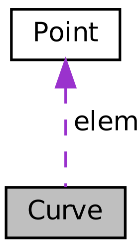
The documentation for this struct was generated from the following file:

- src/include/Almog_Engine.h

3.2 Curve Struct Reference

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

Collaboration diagram for Curve:



Public Attributes

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

3.2.1 Detailed Description

Definition at line 60 of file Almog_Draw_Library.h.

3.2.2 Member Data Documentation

3.2.2.1 capacity

```
size_t Curve::capacity
```

Definition at line 63 of file Almog_Draw_Library.h.

3.2.2.2 color

```
uint32_t Curve::color
```

Definition at line 61 of file Almog_Draw_Library.h.

Referenced by adl_curve_add_to_figure(), and adl_curves_plot_on_figure().

3.2.2.3 elements

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

Definition at line 64 of file Almog_Draw_Library.h.

Referenced by adl_curves_plot_on_figure(), adl_grid_draw(), ae_curve_copy(), ae_curve_project_world2screen(), ae_print_points(), and ae_tri_mesh_get_from_obj_file().

3.2.2.4 length

```
size_t Curve::length
```

Definition at line 62 of file Almog_Draw_Library.h.

Referenced by adl_curves_plot_on_figure(), adl_grid_draw(), ae_curve_copy(), ae_curve_project_world2screen(), and ae_print_points().

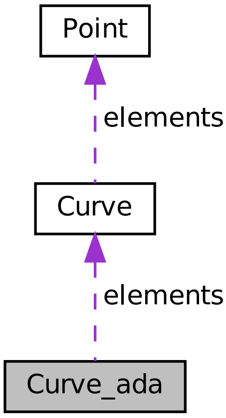
The documentation for this struct was generated from the following file:

- src/include/Almog_Draw_Library.h

3.3 Curve_ada Struct Reference

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

Collaboration diagram for Curve_ada:



Public Attributes

- size_t length
- size_t capacity
- Curve ∗ elements

3.3.1 Detailed Description

Definition at line 70 of file Almog_Draw_Library.h.

3.3.2 Member Data Documentation

3.3.2.1 capacity

```
size_t Curve_ada::capacity
```

Definition at line 72 of file Almog_Draw_Library.h.

3.3.2.2 elements

```
Curve* Curve_ada::elements
```

Definition at line 73 of file Almog_Draw_Library.h.

Referenced by adl_curves_plot_on_figure(), adl_grid_draw(), ae_curve_ada_project_world2screen(), and ae_grid_project_world2screen().

3.3.2.3 length

```
size_t Curve_ada::length
```

Definition at line 71 of file Almog_Draw_Library.h.

Referenced by adl_curves_plot_on_figure(), adl_grid_draw(), ae_curve_ada_project_world2screen(), and ae_grid_project_world2screen().

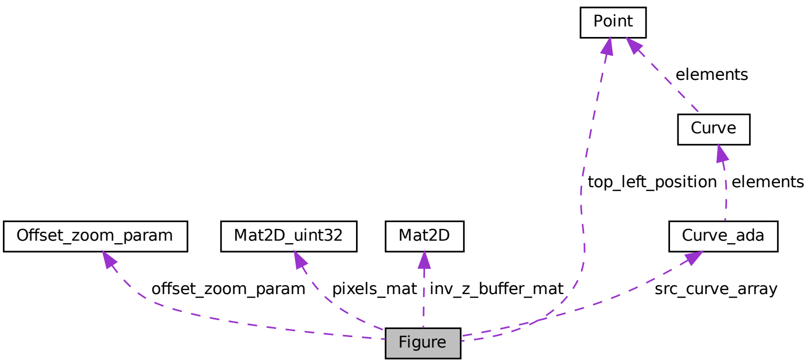
The documentation for this struct was generated from the following file:

- src/include/Almog_Draw_Library.h

3.4 Figure Struct Reference

```
#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

3.4.1 Detailed Description

Definition at line 118 of file Almog_Draw_Library.h.

3.4.2 Member Data Documentation

3.4.2.1 background_color

```
uint32_t Figure::background_color
```

Definition at line 134 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), and adl_curves_plot_on_figure().

3.4.2.2 inv_z_buffer_mat

```
Mat2D Figure::inv_z_buffer_mat
```

Definition at line 133 of file Almog_Draw_Library.h.

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

3.4.2.3 max_x

```
float Figure::max_x
```

Definition at line 124 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().

3.4.2.4 max_x_pixel

```
int Figure::max_x_pixel
```

Definition at line 120 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().

3.4.2.5 max_y

```
float Figure::max_y
```

Definition at line 126 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().

3.4.2.6 max_y_pixel

```
int Figure::max_y_pixel
```

Definition at line 122 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().

3.4.2.7 min_x

```
float Figure::min_x
```

Definition at line 123 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().

3.4.2.8 min_x_pixel

```
int Figure::min_x_pixel
```

Definition at line 119 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().

3.4.2.9 min_y

```
float Figure::min_y
```

Definition at line 125 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().

3.4.2.10 min_y_pixel

```
int Figure::min_y_pixel
```

Definition at line 121 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().

3.4.2.11 offset_zoom_param

```
Offset_zoom_param Figure::offset_zoom_param
```

Definition at line 129 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().

3.4.2.12 pixels_mat

```
Mat2D_uint32 Figure::pixels_mat
```

Definition at line 132 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().

3.4.2.13 src_curve_array

```
Curve_ada Figure::src_curve_array
```

Definition at line 130 of file Almog_Draw_Library.h.

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

3.4.2.14 to_draw_axis

```
bool Figure::to_draw_axis
```

Definition at line 135 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), and adl_curves_plot_on_figure().

3.4.2.15 to_draw_max_min_values

```
bool Figure::to_draw_max_min_values
```

Definition at line 136 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), and adl_curves_plot_on_figure().

3.4.2.16 top_left_position

```
Point Figure::top_left_position
```

Definition at line 131 of file Almog_Draw_Library.h.

Referenced by adl_figure_alloc(), and adl_figure_copy_to_screen().

3.4.2.17 x_axis_head_size

```
int Figure::x_axis_head_size
```

Definition at line 127 of file Almog_Draw_Library.h.

Referenced by adl_axis_draw_on_figure(), and adl_max_min_values_draw_on_figure().

3.4.2.18 y_axis_head_size

```
int Figure::y_axis_head_size
```

Definition at line 128 of file Almog_Draw_Library.h.

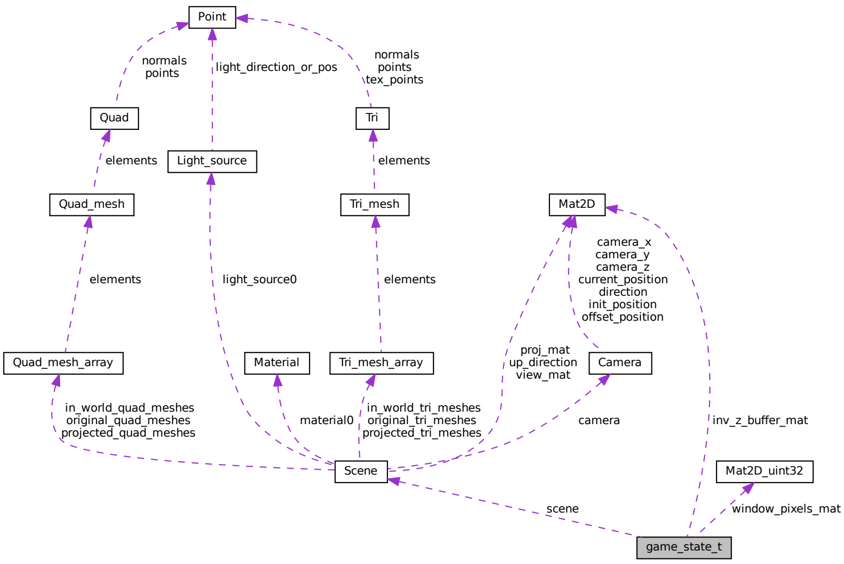
Referenced by adl_axis_draw_on_figure(), and adl_max_min_values_draw_on_figure().

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

- src/include/Almog_Draw_Library.h

3.5 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
- SDL_Surface ∗ window_surface
- SDL_Texture ∗ window_texture
- Mat2D_uint32 window_pixels_mat
- Mat2D inv_z_buffer_mat
- Scene scene

3.5.1 Detailed Description

Definition at line 38 of file display.c.

3.5.2 Member Data Documentation

3.5.2.1 a_was_pressed

```
int game_state_t::a_was_pressed
```

Definition at line 55 of file display.c.

Referenced by main().

3.5.2.2 const_fps

```
float game_state_t::const_fps
```

Definition at line 42 of file display.c.

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

3.5.2.3 d_was_pressed

```
int game_state_t::d_was_pressed
```

Definition at line 56 of file display.c.

Referenced by main().

3.5.2.4 delta_time

```
float game_state_t::delta_time
```

Definition at line 40 of file display.c.

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

3.5.2.5 e_was_pressed

```
int game_state_t::e_was_pressed
```

Definition at line 57 of file display.c.

Referenced by main().

3.5.2.6 elapsed_time

```
float game_state_t::elapsed_time
```

Definition at line 41 of file display.c.

Referenced by main(), and update_window().

3.5.2.7 fps

```
float game_state_t::fps
```

Definition at line 43 of file display.c.

Referenced by main(), and update_window().

3.5.2.8 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().

3.5.2.9 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().

3.5.2.10 inv_z_buffer_mat

```
Mat2D game_state_t::inv_z_buffer_mat
```

Definition at line 69 of file display.c.

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

3.5.2.11 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().

3.5.2.12 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().

3.5.2.13 q_was_pressed

```
int game_state_t::q_was_pressed
```

Definition at line 58 of file display.c.

Referenced by main().

3.5.2.14 renderer

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

Definition at line 63 of file display.c.

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

3.5.2.15 s_was_pressed

```
int game_state_t::s_was_pressed
```

Definition at line 54 of file display.c.

Referenced by main().

3.5.2.16 scene

```
Scene game_state_t::scene
```

Definition at line 71 of file display.c.

Referenced by check_window_mat_size(), destroy_window(), process_input_window(), render(), setup(), setup_window(), and update().

3.5.2.17 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().

3.5.2.18 to_clear_renderer

```
int game_state_t::to_clear_renderer
```

Definition at line 50 of file display.c.

Referenced by main(), and render_window().

3.5.2.19 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().

3.5.2.20 to_render

```
int game_state_t::to_render
```

Definition at line 45 of file display.c.

Referenced by main(), and process_input_window().

3.5.2.21 to_update

```
int game_state_t::to_update
```

Definition at line 46 of file display.c.

Referenced by main(), and process_input_window().

3.5.2.22 w_was_pressed

```
int game_state_t::w_was_pressed
```

Definition at line 53 of file display.c.

Referenced by main().

3.5.2.23 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().

3.5.2.24 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(), update(), and update_window().

3.5.2.25 window_pixels_mat

```
Mat2D_uint32 game_state_t::window_pixels_mat
```

Definition at line 68 of file display.c.

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

3.5.2.26 window_surface

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

Definition at line 65 of file display.c.

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

3.5.2.27 window_texture

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

Definition at line 66 of file display.c.

Referenced by destroy_window().

3.5.2.28 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(), update(), and update_window().

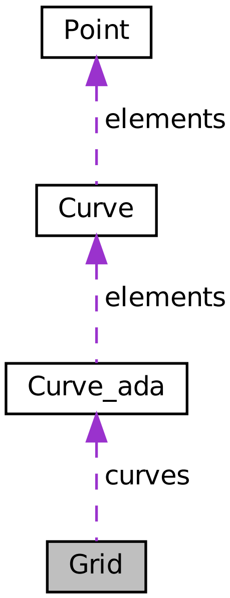
The documentation for this struct was generated from the following file:

- src/include/display.c

3.6 Grid Struct Reference

```
#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]

3.6.1 Detailed Description

Definition at line 139 of file Almog_Draw_Library.h.

3.6.2 Member Data Documentation

3.6.2.1 curves

`Curve_ada Grid::curves`

Definition at line 140 of file Almog_Draw_Library.h.

Referenced by adl_cartesian_grid_create(), adl_grid_draw(), and ae_grid_project_world2screen().

3.6.2.2 de1

`float Grid::de1`

Definition at line 147 of file Almog_Draw_Library.h.

Referenced by adl_cartesian_grid_create().

3.6.2.3 de2

`float Grid::de2`

Definition at line 148 of file Almog_Draw_Library.h.

Referenced by adl_cartesian_grid_create().

3.6.2.4 max_e1

`float Grid::max_e1`

Definition at line 142 of file Almog_Draw_Library.h.

Referenced by adl_cartesian_grid_create().

3.6.2.5 max_e2

```
float Grid::max_e2
```

Definition at line 144 of file Almog_Draw_Library.h.

Referenced by adl_cartesian_grid_create().

3.6.2.6 min_e1

```
float Grid::min_e1
```

Definition at line 141 of file Almog_Draw_Library.h.

Referenced by adl_cartesian_grid_create().

3.6.2.7 min_e2

```
float Grid::min_e2
```

Definition at line 143 of file Almog_Draw_Library.h.

Referenced by adl_cartesian_grid_create().

3.6.2.8 num_samples_e1

```
int Grid::num_samples_e1
```

Definition at line 145 of file Almog_Draw_Library.h.

Referenced by adl_cartesian_grid_create().

3.6.2.9 num_samples_e2

```
int Grid::num_samples_e2
```

Definition at line 146 of file Almog_Draw_Library.h.

Referenced by adl_cartesian_grid_create().

3.6.2.10 plane

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

Definition at line 149 of file Almog_Draw_Library.h.

Referenced by adl_cartesian_grid_create().

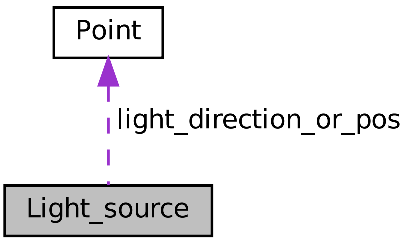
The documentation for this struct was generated from the following file:

- src/include/Almog_Draw_Library.h

3.7 Light_source Struct Reference

```
#include <Almog_Engine.h>
```

Collaboration diagram for Light_source:



Public Attributes

- Point light_direction_or_pos
- float light_intensity

3.7.1 Detailed Description

Definition at line 161 of file Almog_Engine.h.

3.7.2 Member Data Documentation

3.7.2.1 light_direction_or_pos

```
Point Light_source::light_direction_or_pos
```

Definition at line 162 of file Almog_Engine.h.

Referenced by ae_quad_calc_light_intensity(), ae_quad_project_world2screen(), ae_scene_init(), and ae_tri_calc_light_intensity().

3.7.2.2 light_intensity

```
float Light_source::light_intensity
```

Definition at line 163 of file Almog_Engine.h.

Referenced by ae_quad_calc_light_intensity(), ae_scene_init(), and ae_tri_calc_light_intensity().

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

- src/include/Almog_Engine.h

3.8 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

3.8.1 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.

3.8.2 Member Data Documentation

3.8.2.1 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(), ae_point_to_mat2D(), ae_projection_mat_set(), ae_quad_calc_normal(), ae_tri_calc_normal(), ae_z_buffer_copy_to_screen(), 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(), and render_window().

3.8.2.2 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().

3.8.2.3 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(), ae_point_to_mat2D(), ae_projection_mat_set(), ae_quad_calc_normal(), ae_tri_calc_normal(), ae_z_buffer_copy_to_screen(), 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(), and render_window().

3.8.2.4 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:

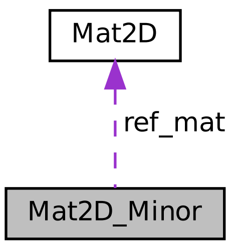
- src/include/Matrix2D.h

3.9 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

3.9.1 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.

3.9.2 Member Data Documentation

3.9.2.1 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().

3.9.2.2 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().

3.9.2.3 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().

3.9.2.4 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().

3.9.2.5 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().

3.9.2.6 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:

- src/include/Matrix2D.h

3.10 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

3.10.1 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.

3.10.2 Member Data Documentation

3.10.2.1 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().

3.10.2.2 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().

3.10.2.3 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().

3.10.2.4 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:

- src/include/Matrix2D.h

3.11 Material Struct Reference

```
#include <Almog_Engine.h>
```

Public Attributes

- float specular_power_alpha
- float c_ambi
- float c_diff
- float c_spec

3.11.1 Detailed Description

Definition at line 166 of file Almog_Engine.h.

3.11.2 Member Data Documentation

3.11.2.1 c_ambi

```
float Material::c_ambi
```

Definition at line 168 of file Almog_Engine.h.

Referenced by ae_quad_calc_light_intensity(), ae_scene_init(), and ae_tri_calc_light_intensity().

3.11.2.2 c_diff

```
float Material::c_diff
```

Definition at line 169 of file Almog_Engine.h.

Referenced by ae_quad_calc_light_intensity(), ae_scene_init(), and ae_tri_calc_light_intensity().

3.11.2.3 c_spec

```
float Material::c_spec
```

Definition at line 170 of file Almog_Engine.h.

Referenced by ae_quad_calc_light_intensity(), ae_scene_init(), and ae_tri_calc_light_intensity().

3.11.2.4 specular_power_alpha

```
float Material::specular_power_alpha
```

Definition at line 167 of file Almog_Engine.h.

Referenced by ae_quad_calc_light_intensity(), ae_scene_init(), and ae_tri_calc_light_intensity().

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

- src/include/Almog_Engine.h

3.12 Offset_zoom_param Struct Reference

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

Public Attributes

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

3.12.1 Detailed Description

Definition at line 40 of file Almog_Draw_Library.h.

3.12.2 Member Data Documentation

3.12.2.1 mouse_x

```
int Offset_zoom_param::mouse_x
```

Definition at line 44 of file Almog_Draw_Library.h.

3.12.2.2 mouse_y

```
int Offset_zoom_param::mouse_y
```

Definition at line 45 of file Almog_Draw_Library.h.

3.12.2.3 offset_x

```
float Offset_zoom_param::offset_x
```

Definition at line 42 of file Almog_Draw_Library.h.

Referenced by adl_line_draw(), and adl_point_draw().

3.12.2.4 offset_y

```
float Offset_zoom_param::offset_y
```

Definition at line 43 of file Almog_Draw_Library.h.

Referenced by adl_line_draw(), and adl_point_draw().

3.12.2.5 zoom_multiplier

```
float Offset_zoom_param::zoom_multiplier
```

Definition at line 41 of file Almog_Draw_Library.h.

Referenced by adl_line_draw(), and adl_point_draw().

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

- src/include/Almog_Draw_Library.h

3.13 Point Struct Reference

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

Public Attributes

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

3.13.1 Detailed Description

Definition at line 50 of file Almog_Draw_Library.h.

3.13.2 Member Data Documentation

3.13.2.1 w

```
float Point::w
```

Definition at line 54 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(), adl_tri_fill_Pinedas_rasterizer_interpolate_normal(), ae_point_normalize_xyz(), ae_point_project_view2screen(), ae_point_project_world2view(), ae_quad_calc_light_intensity(), ae_quad_clip_with_plane(), ae_quad_get_average_normal(), ae_quad_get_average_point(), ae_quad_transform_to_view(), ae_scene_init(), ae_tri_calc_light_intensity(), ae_tri_clip_with_plane(), ae_tri_get_average_normal(), ae_tri_get_average_point(), ae_tri_project_world2screen(), and ae_tri_transform_to_view().

3.13.2.2 x

```
float Point::x
```

Definition at line 51 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(), adl_tri_fill_Pinedas_rasterizer_interpolate_normal(), ae_mat2D_to_point(), ae_point_normalize_xyz(), ae_point_project_view2screen(), ae_point_project_world2view(), ae_point_to_mat2D(), ae_print_points(), ae_print_tri(), ae_quad_get_average_normal(), ae_quad_get_average_point(), ae_quad_project_world2screen(), ae_quad_transform_to_view(), ae_scene_init(), ae_signed_dist_point_and_plane(), ae_tri_clip_with_plane(), ae_tri_get_average_normal(), ae_tri_get_average_point(), ae_tri_mesh_get_from_obj_file(), ae_tri_mesh_get_from_stl_file(), ae_tri_mesh_normalize(), ae_tri_mesh_rotate_Euler_xyz(), ae_tri_mesh_set_bounding_box(), ae_tri_mesh_translate(), ae_tri_project_world2screen(), and ae_tri_transform_to_view().

3.13.2.3 y

```
float Point::y
```

Definition at line 52 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(), adl_tri_fill_Pinedas_rasterizer_interpolate_normal(), ae_point_normalize_xyz(), ae_point_project_view2screen(), ae_point_project_world2view(), ae_point_to_mat2D(), ae_print_points(), ae_print_tri(), ae_quad_get_average_normal(), ae_quad_get_average_point(), ae_quad_project_world2screen(), ae_quad_transform_to_view(), ae_scene_init(), ae_signed_dist_point_and_plane(), ae_tri_clip_with_plane(), ae_tri_get_average_normal(), ae_tri_get_average_point(), ae_tri_mesh_get_from_obj_file(), ae_tri_mesh_get_from_stl_file(), ae_tri_mesh_normalize(), ae_tri_mesh_rotate_Euler_xyz(), ae_tri_mesh_set_bounding_box(), ae_tri_mesh_translate(), ae_tri_project_world2screen(), and ae_tri_transform_to_view().

3.13.2.4 z

```
float Point::z
```

Definition at line 53 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(), adl_tri_fill_Pinedas_rasterizer_interpolate_normal(), ae_point_normalize_xyz(), ae_point_project_view2screen(), ae_point_project_world2view(), ae_point_to_mat2D(), ae_print_points(), ae_print_tri(), ae_quad_get_average_normal(), ae_quad_get_average_point(), ae_quad_project_world2screen(), ae_quad_transform_to_view(), ae_scene_init(), ae_signed_dist_point_and_plane(), ae_tri_compare(), ae_tri_get_average_normal(), ae_tri_get_average_point(), ae_tri_mesh_get_from_obj_file(), ae_tri_mesh_get_from_stl_file(), ae_tri_mesh_normalize(), ae_tri_mesh_rotate_Euler_xyz(), ae_tri_mesh_set_bounding_box(), ae_tri_mesh_translate(), ae_tri_project_world2screen(), and ae_tri_transform_to_view().

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

- src/include/Almog_Draw_Library.h

3.14 Quad Struct Reference

```
#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]

3.14.1 Detailed Description

Definition at line 91 of file Almog_Draw_Library.h.

3.14.2 Member Data Documentation

3.14.2.1 colors

```
uint32_t Quad::colors[4]
```

Definition at line 94 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), adl_quad2tris(), adl_quad_fill_interpolate_color_mean_value(), ae_quad_clip_with_plane(), and ae_tri_mesh_get_from_quad_mesh().

3.14.2.2 light_intensity

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

Definition at line 96 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(), ae_quad_calc_light_intensity(), ae_quad_project_world2screen(), and ae_tri_mesh_get_from_quad_mesh().

3.14.2.3 normals

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

Definition at line 93 of file Almog_Draw_Library.h.

Referenced by ae_quad_calc_light_intensity(), ae_quad_get_average_normal(), ae_quad_project_world2screen(), ae_quad_set_normals(), and ae_tri_mesh_get_from_quad_mesh().

3.14.2.4 points

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

Definition at line 92 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_m adl_quad_fill_interpolate_normal_mean_value(), ae_quad_calc_light_intensity(), ae_quad_calc_normal(), ae_quad_clip_with_plane(), ae_quad_get_average_point(), ae_quad_project_world2screen(), ae_quad_set_normals(), ae_quad_transform_to_view(), and ae_tri_mesh_get_from_quad_mesh().

3.14.2.5 to_draw

```
bool Quad::to_draw
```

Definition at line 95 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(), ae_quad_project_world2screen(), and ae_tri_mesh_get_from_quad_mesh().

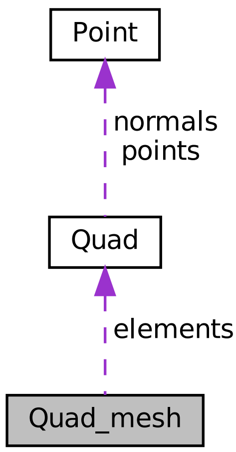
The documentation for this struct was generated from the following file:

- src/include/Almog_Draw_Library.h

3.15 Quad_mesh Struct Reference

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

Collaboration diagram for Quad_mesh:



Public Attributes

- size_t length
- size_t capacity
- Quad * elements

3.15.1 Detailed Description

Definition at line 111 of file Almog_Draw_Library.h.

3.15.2 Member Data Documentation

3.15.2.1 capacity

```
size_t Quad_mesh::capacity
```

Definition at line 113 of file Almog_Draw_Library.h.

3.15.2.2 elements

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

Definition at line 114 of file Almog_Draw_Library.h.

Referenced by adl_quad_mesh_draw(), adl_quad_mesh_fill(), adl_quad_mesh_fill_interpolate_color(), adl_quad_mesh_fill_interpolat
ae_quad_mesh_project_world2screen(), ae_quad_project_world2screen(), ae_scene_free(), and ae_tri_mesh_get_from_quad_mesh

3.15.2.3 length

```
size_t Quad_mesh::length
```

Definition at line 112 of file Almog_Draw_Library.h.

Referenced by adl_quad_mesh_draw(), adl_quad_mesh_fill(), adl_quad_mesh_fill_interpolate_color(), adl_quad_mesh_fill_interpolat
ae_quad_mesh_project_world2screen(), ae_quad_project_world2screen(), and ae_tri_mesh_get_from_quad_mesh().

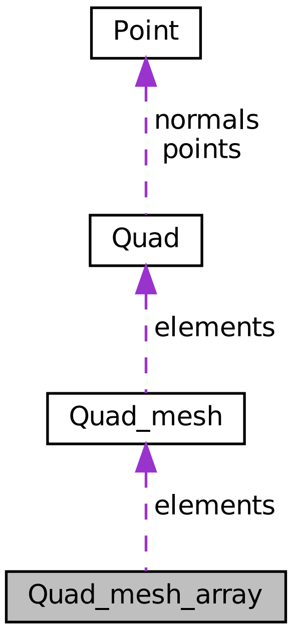
The documentation for this struct was generated from the following file:

- src/include/Almog_Draw_Library.h

3.16 Quad_mesh_array Struct Reference

```
#include <Almog_Engine.h>
```

Collaboration diagram for Quad_mesh_array:



Public Attributes

- size_t length
- size_t capacity
- Quad_mesh ∗ elements

3.16.1 Detailed Description

Definition at line 137 of file Almog_Engine.h.

3.16.2 Member Data Documentation

3.16.2.1 capacity

```
size_t Quad_mesh_array::capacity
```

Definition at line 139 of file Almog_Engine.h.

3.16.2.2 elements

```
Quad_mesh* Quad_mesh_array::elements
```

Definition at line 140 of file Almog_Engine.h.

Referenced by ae_scene_free().

3.16.2.3 length

```
size_t Quad_mesh_array::length
```

Definition at line 138 of file Almog_Engine.h.

Referenced by ae_scene_free().

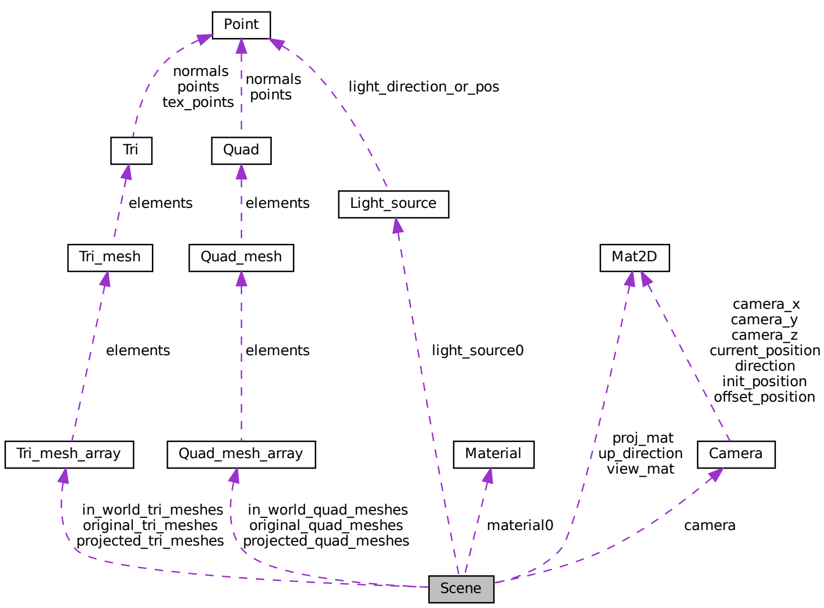
The documentation for this struct was generated from the following file:

- src/include/Almog_Engine.h

3.17 Scene Struct Reference

```
#include <Almog_Engine.h>
```

Collaboration diagram for Scene:



Public Attributes

- Tri_mesh_array in_world_tri_meshes
- Tri_mesh_array projected_tri_meshes
- Tri_mesh_array original_tri_meshes
- Quad_mesh_array in_world_quad_meshes
- Quad_mesh_array projected_quad_meshes
- Quad_mesh_array original_quad_meshes
- Camera camera
- Mat2D up_direction
- Mat2D proj_mat
- Mat2D view_mat
- Light_source light_source0
- Material material0

3.17.1 Detailed Description

Definition at line 173 of file Almog_Engine.h.

3.17.2 Member Data Documentation

3.17.2.1 camera

```
Camera Scene::camera
```

Definition at line 182 of file Almog_Engine.h.

Referenced by ae_camera_free(), ae_camera_init(), ae_camera_reset_pos(), ae_line_project_world2screen(), ae_quad_calc_light_intensity(), ae_quad_project_world2screen(), ae_scene_init(), ae_tri_calc_light_intensity(), ae_tri_project_world2screen(), check_window_mat_size(), process_input_window(), and update().

3.17.2.2 in_world_quad_meshes

```
Quad_mesh_array Scene::in_world_quad_meshes
```

Definition at line 178 of file Almog_Engine.h.

Referenced by ae_scene_free().

3.17.2.3 in_world_tri_meshes

`Tri_mesh_array Scene::in_world_tri_meshes`

Definition at line 174 of file Almog_Engine.h.

Referenced by ae_scene_free(), render(), setup(), and update().

3.17.2.4 light_source0

`Light_source Scene::light_source0`

Definition at line 187 of file Almog_Engine.h.

Referenced by ae_quad_calc_light_intensity(), ae_quad_project_world2screen(), ae_scene_init(), and ae_tri_calc_light_intensity().

3.17.2.5 material0

`Material Scene::material0`

Definition at line 188 of file Almog_Engine.h.

Referenced by ae_quad_calc_light_intensity(), ae_scene_init(), and ae_tri_calc_light_intensity().

3.17.2.6 original_quad_meshes

`Quad_mesh_array Scene::original_quad_meshes`

Definition at line 180 of file Almog_Engine.h.

Referenced by ae_scene_free().

3.17.2.7 original_tri_meshes

`Tri_mesh_array Scene::original_tri_meshes`

Definition at line 176 of file Almog_Engine.h.

Referenced by ae_scene_free(), and setup().

3.17.2.8 proj_mat

```
Mat2D Scene::proj_mat
```

Definition at line 184 of file Almog_Engine.h.

Referenced by ae_scene_free(), ae_scene_init(), and update().

3.17.2.9 projected_quad_meshes

```
Quad_mesh_array Scene::projected_quad_meshes
```

Definition at line 179 of file Almog_Engine.h.

Referenced by ae_scene_free().

3.17.2.10 projected_tri_meshes

```
Tri_mesh_array Scene::projected_tri_meshes
```

Definition at line 175 of file Almog_Engine.h.

Referenced by ae_scene_free(), render(), setup(), and update().

3.17.2.11 up_direction

```
Mat2D Scene::up_direction
```

Definition at line 183 of file Almog_Engine.h.

Referenced by ae_scene_free(), ae_scene_init(), and update().

3.17.2.12 view_mat

```
Mat2D Scene::view_mat
```

Definition at line 185 of file Almog_Engine.h.

Referenced by ae_scene_free(), ae_scene_init(), and update().

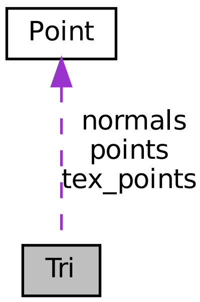
The documentation for this struct was generated from the following file:

- src/include/Almog_Engine.h

3.18 Tri Struct Reference

```
#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]

3.18.1 Detailed Description

Definition at line 79 of file Almog_Draw_Library.h.

3.18.2 Member Data Documentation

3.18.2.1 colors

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

Definition at line 83 of file Almog_Draw_Library.h.

Referenced by adl_quad2tris(), adl_tri_fill_Pinedas_rasterizer_interpolate_color(), ae_tri_clip_with_plane(), ae_tri_mesh_flip_normals(), ae_tri_mesh_get_from_obj_file(), ae_tri_mesh_get_from_quad_mesh(), and ae_tri_mesh_get_from_stl_file().

3.18.2.2 light_intensity

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

Definition at line 85 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(), ae_tri_calc_light_intensity(), ae_tri_mesh_flip_normals(), ae_tri_mesh_get_from_obj_file(), ae_tri_mesh_get_from_quad_mesh(), ae_tri_mesh_get_from_stl_file(), and ae_tri_project_world2screen().

3.18.2.3 normals

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

Definition at line 82 of file Almog_Draw_Library.h.

Referenced by ae_tri_calc_light_intensity(), ae_tri_get_average_normal(), ae_tri_mesh_flip_normals(), ae_tri_mesh_get_from_quad_ ae_tri_mesh_get_from_stl_file(), ae_tri_project_world2screen(), and ae_tri_set_normals().

3.18.2.4 points

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

Definition at line 80 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(), ae_print_tri(), ae_tri_calc_light_intensity(), ae_tri_calc_normal(), ae_tri_clip_with_plane(), ae_tri_compare(), ae_tri_create(), ae_tri_get_average_point(), ae_tri_mesh_flip_normals(), ae_tri_mesh_get_from_obj_file(), ae_tri_mesh_get_from_quad_mesh(), ae_tri_mesh_get_from_stl_file(), ae_tri_mesh_normalize(), ae_tri_mesh_rotate_Euler_xyz(), ae_tri_mesh_set_bounding_box(), ae_tri_mesh_translate(), ae_tri_project_world2screen(), ae_tri_set_normals(), and ae_tri_transform_to_view().

3.18.2.5 tex_points

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

Definition at line 81 of file Almog_Draw_Library.h.

Referenced by ae_tri_clip_with_plane(), ae_tri_mesh_flip_normals(), and ae_tri_project_world2screen().

3.18.2.6 to_draw

```
bool Tri::to_draw
```

Definition at line 84 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_inte
adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_normal(), ae_print_tri(), ae_tri_mesh_flip_normals(), ae_tri_mesh_get_from_obj_file
ae_tri_mesh_get_from_quad_mesh(), ae_tri_mesh_get_from_stl_file(), and ae_tri_project_world2screen().

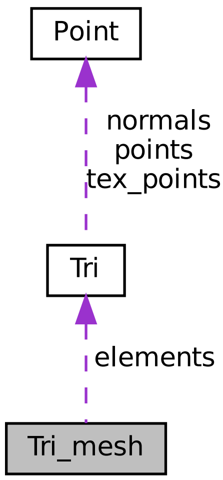
The documentation for this struct was generated from the following file:

- src/include/Almog_Draw_Library.h

3.19 Tri_mesh Struct Reference

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

Collaboration diagram for Tri_mesh:



Public Attributes

- size_t length
- size_t capacity
- Tri ∗ elements

3.19.1 Detailed Description

Definition at line 102 of file Almog_Draw_Library.h.

3.19.2 Member Data Documentation

3.19.2.1 capacity

```
size_t Tri_mesh::capacity
```

Definition at line 104 of file Almog_Draw_Library.h.

3.19.2.2 elements

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

Definition at line 105 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(), adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_normal(), ae_print_tri_mesh(), ae_scene_free(), ae_tri_mesh_append_copy(), ae_tri_mesh_flip_normals(), ae_tri_mesh_normalize(), ae_tri_mesh_project_world2screen(), ae_tri_mesh_rotate_Euler_xyz(), ae_tri_mesh_set_bounding_box(), ae_tri_mesh_set_normals(), ae_tri_mesh_translate(), and ae_tri_project_world2screen().

3.19.2.3 length

```
size_t Tri_mesh::length
```

Definition at line 103 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(), adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_normal(), ae_print_tri_mesh(), ae_tri_mesh_append_copy(), ae_tri_mesh_create_copy(), ae_tri_mesh_flip_normals(), ae_tri_mesh_normalize(), ae_tri_mesh_project_world2screen(), ae_tri_mesh_rotate_Euler_xyz(), ae_tri_mesh_set_bounding_box(), ae_tri_mesh_set_normals(), ae_tri_mesh_translate(), ae_tri_project_world2screen(), render(), and setup().

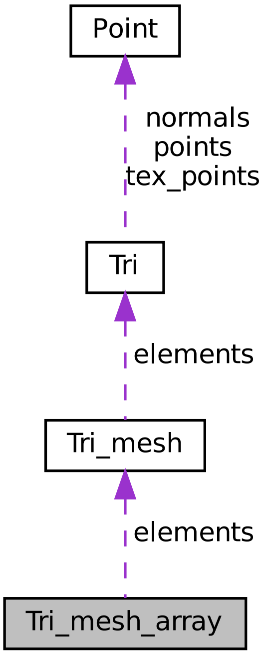
The documentation for this struct was generated from the following file:

- src/include/Almog_Draw_Library.h

3.20 Tri_mesh_array Struct Reference

```
#include <Almog_Engine.h>
```

Collaboration diagram for Tri_mesh_array:



Public Attributes

- size_t length
- size_t capacity
- Tri_mesh ∗ elements

3.20.1 Detailed Description

Definition at line 128 of file Almog_Engine.h.

3.20.2 Member Data Documentation

3.20.2.1 capacity

```
size_t Tri_mesh_array::capacity
```

Definition at line 130 of file Almog_Engine.h.

3.20.2.2 elements

```
Tri_mesh* Tri_mesh_array::elements
```

Definition at line 131 of file Almog_Engine.h.

Referenced by ae_scene_free(), render(), setup(), and update().

3.20.2.3 length

```
size_t Tri_mesh_array::length
```

Definition at line 129 of file Almog_Engine.h.

Referenced by ae_scene_free(), render(), setup(), and update().

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

- src/include/Almog_Engine.h

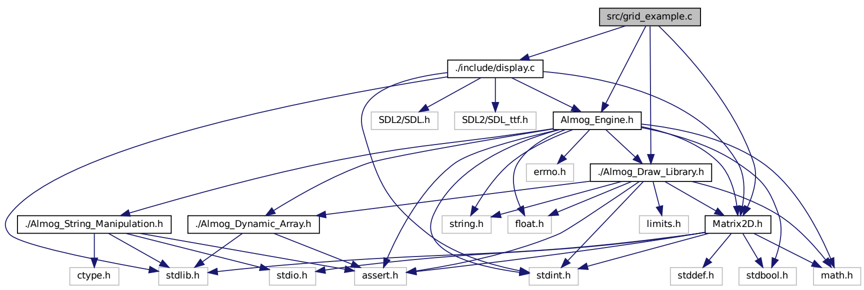
Chapter 4

File Documentation

4.1 src/grid_example.c File Reference

```
#include "./include/display.c"
#include "./include/Matrix2D.h"
#include "./include/Almog_Draw_Library.h"
#include "./include/Almog_Engine.h"
```

Include dependency graph for grid_example.c:



Macros

- #define SETUP
- #define UPDATE
- #define RENDER
- #define MATRIX2D_IMPLEMENTATION
- #define ALMOG_DRAW_LIBRARY_IMPLEMENTATION
- #define ALMOG_ENGINE_IMPLEMENTATION

Functions

- void setup (game_state_t *game_state)
- void update (game_state_t *game_state)
- void render (game_state_t *game_state)

Variables

- Grid grid
- Grid grid_proj

4.1.1 Macro Definition Documentation

4.1.1.1 ALMOG_DRAW_LIBRARY_IMPLEMENTATION

```
#define ALMOG_DRAW_LIBRARY_IMPLEMENTATION
```

Definition at line 7 of file grid_example.c.

4.1.1.2 ALMOG_ENGINE_IMPLEMENTATION

```
#define ALMOG_ENGINE_IMPLEMENTATION
```

Definition at line 9 of file grid_example.c.

4.1.1.3 MATRIX2D_IMPLEMENTATION

```
#define MATRIX2D_IMPLEMENTATION
```

Definition at line 5 of file grid_example.c.

4.1.1.4 RENDER

```
#define RENDER
```

Definition at line 3 of file grid_example.c.

4.1.1.5 SETUP

```
#define SETUP
```

Definition at line 1 of file grid_example.c.

4.1.1.6 UPDATE

```
#define UPDATE
```

Definition at line 2 of file grid_example.c.

4.1.2 Function Documentation

4.1.2.1 render()

```
void render (
            game_state_t * game_state )
```

Definition at line 32 of file grid_example.c.

References ADL_DEFAULT_OFFSET_ZOOM, adl_grid_draw(), grid_proj, and game_state_t::window_pixels_mat.

4.1.2.2 setup()

```
void setup (
            game_state_t * game_state )
```

Definition at line 14 of file grid_example.c.

References adl_cartesian_grid_create(), game_state_t::const_fps, grid, and grid_proj.

4.1.2.3 update()

```
void update (
            game_state_t * game_state )
```

Definition at line 23 of file grid_example.c.

References ae_grid_project_world2screen(), ae_projection_mat_set(), ae_view_mat_set(), Camera::aspect_ratio, Scene::camera, Camera::fov_deg, grid, grid_proj, Scene::proj_mat, game_state_t::scene, Scene::up_direction, Scene::view_mat, game_state_t::window_h, game_state_t::window_w, Camera::z_far, and Camera::z_near.

4.1.3 Variable Documentation

4.1.3.1 grid

```
Grid grid
```

Definition at line 12 of file grid_example.c.

Referenced by adl_cartesian_grid_create(), adl_grid_draw(), setup(), and update().

4.1.3.2 grid_proj

```
Grid grid_proj
```

Definition at line 13 of file grid_example.c.

Referenced by render(), setup(), and update().

4.2 grid_example.c

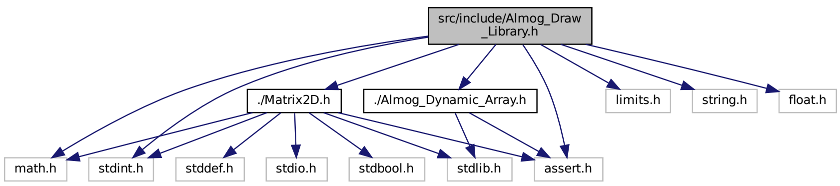
```
00001 #define SETUP
00002 #define UPDATE
00003 #define RENDER
00004 #include "./include/display.c"
00005 #define MATRIX2D_IMPLEMENTATION
00006 #include "./include/Matrix2D.h"
00007 #define ALMOG_DRAW_LIBRARY_IMPLEMENTATION
00008 #include "./include/Almog_Draw_Library.h"
00009 #define ALMOG_ENGINE_IMPLEMENTATION
00010 #include "./include/Almog_Engine.h"
00011
00012 Grid grid;
00013 Grid grid_proj;
00014 void setup(game_state_t *game_state)
00015 {
00016     // game_state->to_limit_fps = 0;
00017     game_state->const_fps = 500;
00018
00019     grid      = adl_cartesian_grid_create(-1, 1, -2, 2, 10, 20, "XZ", 1);
00020     grid_proj = adl_cartesian_grid_create(-1, 1, -2, 2, 10, 20, "XZ", 1);
00021 }
00022
00023 void update(game_state_t *game_state)
00024 {
00025     ae_projection_mat_set(game_state->scene.proj_mat, game_state->scene.camera.aspect_ratio,
      game_state->scene.camera.fov_deg, game_state->scene.camera.z_near, game_state->scene.camera.z_far);
00026     ae_view_mat_set(game_state->scene.view_mat, game_state->scene.camera,
      game_state->scene.up_direction);
00027
00028     ae_grid_project_world2screen(game_state->scene.proj_mat, game_state->scene.view_mat, grid_proj,
      grid, game_state->window_w, game_state->window_h, &(game_state->scene));
00029
00030 }
00031
00032 void render(game_state_t *game_state)
00033 {
00034     adl_grid_draw(game_state->window_pixels_mat, grid_proj, 0xffffffff, ADL_DEFAULT_OFFSET_ZOOM);
00035
00036 }
```

4.3 src/include/Almog_Draw_Library.h File Reference

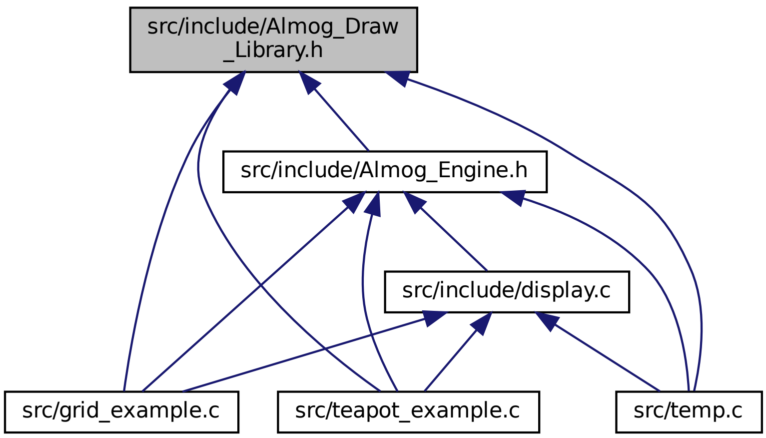
Immediate-mode 2D/3D raster helpers for drawing onto Mat2D_uint32 pixel buffers.

```
#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
- struct Point
- struct Curve
- struct Curve_ada
- struct Tri

- struct Quad
- struct Tri_mesh
- struct Quad_mesh
- struct Figure
- struct Grid

Macros

- #define ADL_ASSERT 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.

4.3.1 Detailed Description

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

Conventions

- Pixel buffer: Mat2D_uint32 with elements encoded as ARGB 0xAARRGGBB.
- Coordinates: x grows to the right, y grows downward; origin is the top-left corner of the destination buffer.
- Depth: Functions that accept inv_z_buffer perform a depth test using inverse-Z (larger values are closer). The buffer stores doubles.
- Transform: Most drawing functions accept an Offset_zoom_param describing a pan/zoom transform that is applied about the screen center. Use ADL_DEFAULT_OFFSET_ZOOM for identity.
- Colors: Unless noted otherwise, colors are ARGB in 0xAARRGGBB format.
- Alpha: adl_point_draw alpha-blends source over destination and writes an opaque result (A = 255) to the pixel buffer.

This header contains function declarations and optional implementations (guarded by ALMOG_DRAW_LIBRARY↩ _IMPLEMENTATION).

Definition in file Almog_Draw_Library.h.

4.3.2 Macro Definition Documentation

4.3.2.1 ADL_ASSERT

```
#define ADL_ASSERT assert
```

Definition at line 37 of file Almog_Draw_Library.h.

4.3.2.2 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 243 of file Almog_Draw_Library.h.

4.3.2.3 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 247 of file Almog_Draw_Library.h.

4.3.2.4 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 244 of file Almog_Draw_Library.h.

4.3.2.5 ADL_DEFAULT_OFFSET_ZOOM

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

Definition at line 264 of file Almog_Draw_Library.h.

4.3.2.6 ADL_FIGURE_AXIS_COLOR

```
#define ADL_FIGURE_AXIS_COLOR 0xff000000
```

Definition at line 257 of file Almog_Draw_Library.h.

4.3.2.7 ADL_FIGURE_HEAD_ANGLE_DEG

```
#define ADL_FIGURE_HEAD_ANGLE_DEG 30
```

Definition at line 256 of file Almog_Draw_Library.h.

4.3.2.8 ADL_FIGURE_PADDING_PRECENTAGE

```
#define ADL_FIGURE_PADDING_PRECENTAGE 20
```

Definition at line 252 of file Almog_Draw_Library.h.

4.3.2.9 ADL_MAX_CHARACTER_OFFSET

```
#define ADL_MAX_CHARACTER_OFFSET 10
```

Definition at line 259 of file Almog_Draw_Library.h.

4.3.2.10 ADL_MAX_FIGURE_PADDING

```
#define ADL_MAX_FIGURE_PADDING 70
```

Definition at line 253 of file Almog_Draw_Library.h.

4.3.2.11 ADL_MAX_HEAD_SIZE

```
#define ADL_MAX_HEAD_SIZE 15
```

Definition at line 255 of file Almog_Draw_Library.h.

4.3.2.12 ADL_MAX_POINT_VAL

```
#define ADL_MAX_POINT_VAL 1e5
```

Definition at line 242 of file Almog_Draw_Library.h.

4.3.2.13 ADL_MAX_SENTENCE_LEN

```
#define ADL_MAX_SENTENCE_LEN 256
```

Definition at line 261 of file Almog_Draw_Library.h.

4.3.2.14 ADL_MAX_ZOOM

```
#define ADL_MAX_ZOOM 1e3
```

Definition at line 262 of file Almog_Draw_Library.h.

4.3.2.15 ADL_MIN_CHARACTER_OFFSET

```
#define ADL_MIN_CHARACTER_OFFSET 5
```

Definition at line 260 of file Almog_Draw_Library.h.

4.3.2.16 ADL_MIN_FIGURE_PADDING

```
#define ADL_MIN_FIGURE_PADDING 20
```

Definition at line 254 of file Almog_Draw_Library.h.

4.3.2.17 adl_offset2d

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

Definition at line 2227 of file Almog_Draw_Library.h.

4.3.2.18 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 265 of file Almog_Draw_Library.h.

4.3.2.19 BLUE_hexARGB

```
#define BLUE_hexARGB 0xFF0000FF
```

Definition at line 232 of file Almog_Draw_Library.h.

4.3.2.20 CURVE

```
#define CURVE
```

Definition at line 59 of file Almog_Draw_Library.h.

4.3.2.21 CURVE_ADA

```
#define CURVE_ADA
```

Definition at line 69 of file Almog_Draw_Library.h.

4.3.2.22 CYAN_hexARGB

```
#define CYAN_hexARGB 0xFF00FFFF
```

Definition at line 234 of file Almog_Draw_Library.h.

4.3.2.23 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 237 of file Almog_Draw_Library.h.

4.3.2.24 GREEN_hexARGB

```
#define GREEN_hexARGB 0xFF00FF00
```

Definition at line 231 of file Almog_Draw_Library.h.

4.3.2.25 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 157 of file Almog_Draw_Library.h.

4.3.2.26 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 154 of file Almog_Draw_Library.h.

4.3.2.27 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 160 of file Almog_Draw_Library.h.

4.3.2.28 is_left_edge

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

Definition at line 239 of file Almog_Draw_Library.h.

4.3.2.29 is_top_edge

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

Definition at line 238 of file Almog_Draw_Library.h.

4.3.2.30 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 240 of file Almog_Draw_Library.h.

4.3.2.31 POINT

```
#define POINT
```

Definition at line 49 of file Almog_Draw_Library.h.

4.3.2.32 PURPLE_hexARGB

```
#define PURPLE_hexARGB 0xFFFF00FF
```

Definition at line 233 of file Almog_Draw_Library.h.

4.3.2.33 QUAD

```
#define QUAD
```

Definition at line 90 of file Almog_Draw_Library.h.

4.3.2.34 QUAD_MESH

```
#define QUAD_MESH
```

Definition at line 110 of file Almog_Draw_Library.h.

4.3.2.35 RED_hexARGB

```
#define RED_hexARGB 0xFFFF0000
```

Definition at line 230 of file Almog_Draw_Library.h.

4.3.2.36 RGB_hexRGB

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

Definition at line 163 of file Almog_Draw_Library.h.

4.3.2.37 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 166 of file Almog_Draw_Library.h.

4.3.2.38 TRI

```
#define TRI
```

Definition at line 78 of file Almog_Draw_Library.h.

4.3.2.39 TRI_MESH

```
#define TRI_MESH
```

Definition at line 101 of file Almog_Draw_Library.h.

4.3.2.40 YELLOW_hexARGB

```
#define YELLOW_hexARGB 0xFFFFFF00
```

Definition at line 235 of file Almog_Draw_Library.h.

4.3.3 Function Documentation

4.3.3.1 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 2247 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.

4.3.3.2 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 451 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().

4.3.3.3 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 2077 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().

4.3.3.4 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 2446 of file Almog_Draw_Library.h.

References ada_append, ada_init_array, Grid::curves, Grid::de1, Grid::de2, grid, 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.

Referenced by setup().

4.3.3.5 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 519 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().

4.3.3.6 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 1360 of file Almog_Draw_Library.h.

References adl_point_draw().

4.3.3.7 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 1382 of file Almog_Draw_Library.h.

References adl_point_draw().

4.3.3.8 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 2163 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.

4.3.3.9 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 2198 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.

4.3.3.10 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 2014 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.

4.3.3.11 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 2057 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.

4.3.3.12 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 2724 of file Almog_Draw_Library.h.

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

Referenced by render().

4.3.3.13 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 1986 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().

4.3.3.14 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 316 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().

4.3.3.15 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 1798 of file Almog_Draw_Library.h.

Referenced by adl_2Dscalar_interp_on_figure(), and adl_curves_plot_on_figure().

4.3.3.16 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 1878 of file Almog_Draw_Library.h.

References HexARGB_RGB_VAR.

Referenced by adl_linear_sRGB_to_okLch().

4.3.3.17 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 1945 of file Almog_Draw_Library.h.

References adl_linear_sRGB_to_okLab(), and PI.

Referenced by adl_interpolate_ARGBcolor_on_okLch().

4.3.3.18 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 403 of file Almog_Draw_Library.h.

References adl_line_draw().

Referenced by adl_grid_draw().

4.3.3.19 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 423 of file Almog_Draw_Library.h.

References adl_line_draw().

Referenced by adl_quad_draw().

4.3.3.20 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 2103 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().

4.3.3.21 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 1913 of file Almog_Draw_Library.h.

References RGBA_hexARGB.

Referenced by adl_okLch_to_linear_sRGB().

4.3.3.22 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 1964 of file Almog_Draw_Library.h.

References adl_okLab_to_linear_sRGB(), and PI.

Referenced by adl_interpolate_ARGBcolor_on_okLch().

4.3.3.23 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 283 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().

4.3.3.24 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 1818 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.

4.3.3.25 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 943 of file Almog_Draw_Library.h.

References adl_lines_loop_draw(), and Quad::points.

Referenced by adl_quad_mesh_draw().

4.3.3.26 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 961 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().

4.3.3.27 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 1149 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(), and adl_quad_mesh_fill_interpolate_color().

4.3.3.28 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 1055 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().

4.3.3.29 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 1253 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.

4.3.3.30 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 1277 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.

4.3.3.31 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 1334 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.

4.3.3.32 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 1304 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.

4.3.3.33 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).
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 906 of file Almog_Draw_Library.h.

References adl_line_draw().

Referenced by adl_character_draw().

4.3.3.34 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 925 of file Almog_Draw_Library.h.

References adl_line_draw().

Referenced by adl_character_draw().

4.3.3.35 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 882 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().

4.3.3.36 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 1778 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().

4.3.3.37 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 1402 of file Almog_Draw_Library.h.

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

Referenced by adl_tri_mesh_draw().

4.3.3.38 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 1425 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, Point::w, Point::x, Point::y, and Point::z.

Referenced by adl_tri_mesh_fill_Pinedas_rasterizer().

4.3.3.39 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 1506 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, Point::w, Point::x, Point::y, and Point::z.

Referenced by adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_color().

4.3.3.40 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 1597 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, Point::w, Point::x, Point::y, and Point::z.

Referenced by adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_normal().

4.3.3.41 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 1679 of file Almog_Draw_Library.h.

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

4.3.3.42 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 1701 of file Almog_Draw_Library.h.

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

4.3.3.43 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 1725 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, and Tri::to_draw.

4.3.3.44 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 1750 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, and Tri::to_draw.

Referenced by render().

4.4 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);
```

```c

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);
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;

    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);
}

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];
    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];
    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;
```

```c
    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);
```

```c

    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;
            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;
            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;
            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;
            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;
            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;
            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;
            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;
            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;
            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;
            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)) {
```

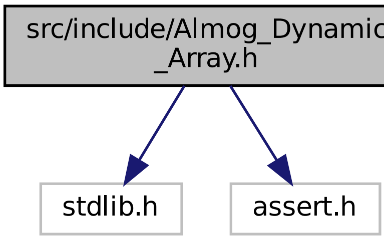
```
        for (int e1_index = 0; e1_index <= num_samples_e1; e1_index++) {
            Curve curve;
            ada_init_array(Point, curve);
            Point point_max = {0}, point_min = {0};

            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;
            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*/
```

4.5 src/include/Almog_Dynamic_Array.h File Reference

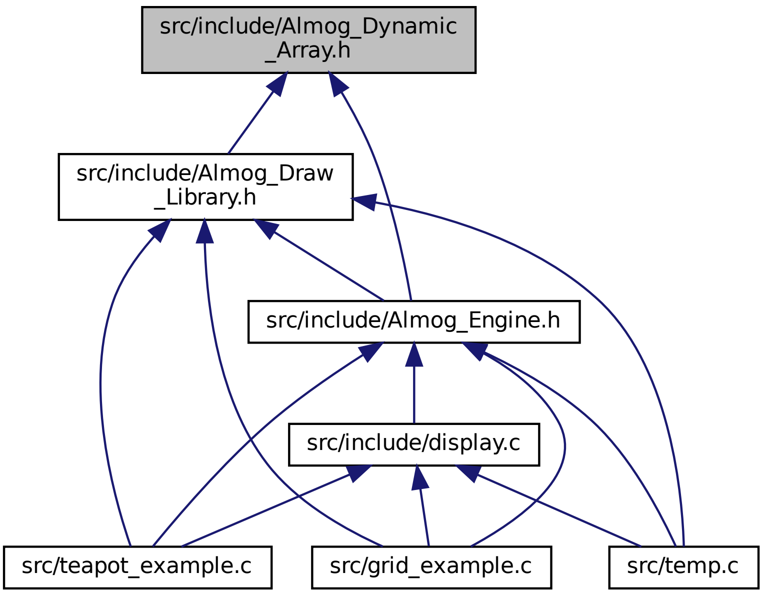
Header-only C macros that implement a simple dynamic array.

```
#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.

4.5.1 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.

4.5.2 Macro Definition Documentation

4.5.2.1 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.

4.5.2.2 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.

4.5.2.3 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.

4.5.2.4 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.

4.5.2.5 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.

4.5.2.6 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.

4.5.2.7 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.

4.5.2.8 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.

4.5.2.9 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.

4.5.2.10 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.

4.5.2.11 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.

4.6 Almog_Dynamic_Array.h

```
00001
00051 #ifndef ALMOG_DYNAMIC_ARRAY_H_
00052 #define ALMOG_DYNAMIC_ARRAY_H_
00053
00054 #include <stdlib.h>
00055 #include <assert.h>
00056
00057
00064 #define ADA_INIT_CAPACITY 10
00065
00073 #ifndef ADA_MALLOC
00074 #define ADA_MALLOC malloc
00075 #endif /*ADA_MALLOC*/
00076
00084 #ifndef ADA_REALLOC
00085 #define ADA_REALLOC realloc
00086 #endif /*ADA_REALLOC*/
00087
00095 #ifndef ADA_ASSERT
00096 #define ADA_ASSERT assert
00097 #endif /*ADA_ASSERT*/
00098
00099 /* typedef struct {
00100     size_t length;
00101     size_t capacity;
00102     int* elements;
00103 } ada_int_array; */
00104
00120 #define ada_init_array(type, header) do {                                       \
00121         header.capacity = ADA_INIT_CAPACITY;                                        \
00122         header.length = 0;                                                      \
00123         header.elements = (type *)ADA_MALLOC(sizeof(type) * header.capacity);   \
00124         ADA_ASSERT(header.elements != NULL);                                    \
00125     } while (0)
00126
```

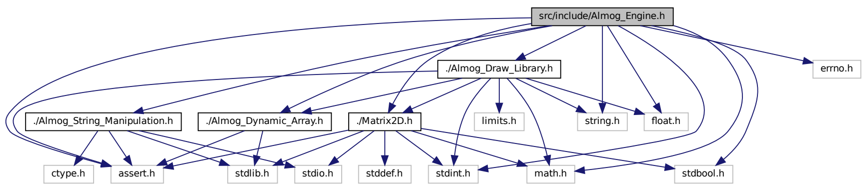
```c
#define ada_resize(type, header, new_capacity) do { \
        type *ada_temp_pointer = (type *)ADA_REALLOC((void *)(header.elements), new_capacity*sizeof(type)); \
        if (ada_temp_pointer == NULL) { \
            exit(1); \
        } \
        header.elements = ada_temp_pointer; \
        ADA_ASSERT(header.elements != 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_*/
```

4.7 src/include/Almog_Engine.h File Reference

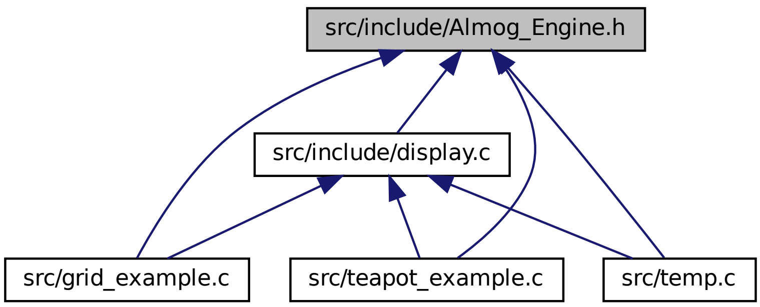
Software 3D rendering and scene utilities for meshes, camera, and projection.

```
#include "./Almog_Dynamic_Array.h"
#include "./Matrix2D.h"
#include "./Almog_Draw_Library.h"
#include "./Almog_String_Manipulation.h"
#include <assert.h>
#include <math.h>
#include <stdbool.h>
#include <float.h>
#include <stdint.h>
#include <errno.h>
#include <string.h>
```

Include dependency graph for Almog_Engine.h:



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



Classes

- struct Tri_mesh_array
- struct Quad_mesh_array
- struct Camera
- struct Light_source
- struct Material
- struct Scene

Macros

- #define AE_ASSERT assert
- #define STL_HEADER_SIZE 80
- #define STL_NUM_SIZE 4
- #define STL_SIZE_FOREACH_TRI 50
- #define STL_ATTRIBUTE_BITS_SIZE 2
- #define ARGB_hexARGB(a, r, g, b) 0x01000000l∗(uint8_t)(a) + 0x00010000∗(uint8_t)(r) + 0x00000100∗(uint8←_t)(g) + 0x00000001∗(uint8_t)(b)
- #define AE_MAX_POINT_VAL 1e5
- #define ae_assert_point_is_valid(p)
- #define ae_assert_tri_is_valid(tri)
- #define ae_assert_quad_is_valid(quad)
- #define ae_point_normalize_xyz_norma(p, norma)
- #define ae_point_calc_norma(p) sqrt(((p).x ∗ (p).x) + ((p).y ∗ (p).y) + ((p).z ∗ (p).z))
- #define ae_point_add_point(p, p1, p2)
- #define ae_point_sub_point(p, p1, p2)
- #define ae_point_dot_point(p1, p2) (((p1).x ∗ (p2).x) + ((p1).y ∗ (p2).y) + ((p1).z ∗ (p2).z))
- #define ae_point_mult(p, const)
- #define ae_points_equal(p1, p2) (p1).x == (p2).x && (p1).y == (p2).y && (p1).z == (p2).z
- #define TRI_MESH_ARRAY
- #define QUAD_MESH_ARRAY
- #define AE_PRINT_TRI(tri) ae_print_tri(tri, #tri, 0)
- #define AE_PRINT_MESH(mesh) ae_print_tri_mesh(mesh, #mesh, 0)

Enumerations

- enum Lighting_mode { AE_LIGHTING_FLAT , AE_LIGHTING_SMOOTH , AE_LIGHTING_MODE_LENGTH }

Functions

- Tri ae_tri_create (Point p1, Point p2, Point p3)

  Create a triangle from three points.
- void ae_tri_mesh_create_copy (Tri_mesh ∗des, Tri ∗src_elements, size_t len)

  Append copies of triangles to a destination Tri_mesh (resets destination length first).
- void ae_camera_init (Scene ∗scene, int window_h, int window_w)

  Initialize the camera part of a Scene.
- void ae_camera_free (Scene ∗scene)

  Free camera-related allocations in a Scene.
- Scene ae_scene_init (int window_h, int window_w)

  Create and initialize a Scene.
- void ae_scene_free (Scene ∗scene)

  Free all resources owned by a Scene.
- void ae_camera_reset_pos (Scene ∗scene)

  Reset camera orientation and position to initial state.
- void ae_point_to_mat2D (Point p, Mat2D m)

  Write a Point into a Mat2D vector.
- Point ae_mat2D_to_point (Mat2D m)

  Read a 3x1 Mat2D vector into a Point.
- Tri_mesh ae_tri_mesh_get_from_obj_file (char ∗file_path)

Load a triangle mesh from a Wavefront OBJ file.
- Tri_mesh ae_tri_mesh_get_from_stl_file (char ∗file_path)

  Load a triangle mesh from a binary STL file.
- Tri_mesh ae_tri_mesh_get_from_file (char ∗file_path)

  Load a triangle mesh from a file (OBJ or STL).
- void ae_tri_mesh_appand_copy (Tri_mesh_array ∗mesh_array, Tri_mesh mesh)

  Append a copy of a Tri_mesh into a Tri_mesh_array.
- Tri_mesh ae_tri_mesh_get_from_quad_mesh (Quad_mesh q_mesh)

  Convert a Quad_mesh into a Tri_mesh.
- void ae_print_points (Curve p)

  Print a list of points to stdout.
- void ae_print_tri (Tri tri, char ∗name, size_t padding)

  Print a triangle to stdout.
- void ae_print_tri_mesh (Tri_mesh mesh, char ∗name, size_t padding)

  Print all triangles in a mesh to stdout.
- Point ae_point_normalize_xyz (Point p)

  Normalize a point's xyz to unit length.
- void ae_tri_set_normals (Tri ∗tri)

  Compute and set per-vertex normals for a triangle.
- Point ae_tri_get_average_normal (Tri tri)

  Compute the average of the three vertex normals of a triangle.
- Point ae_tri_get_average_point (Tri tri)

  Compute the average of the three vertices of a triangle.
- void ae_tri_calc_normal (Mat2D normal, Tri tri)

  Compute the face normal of a triangle.
- void ae_tri_mesh_translate (Tri_mesh mesh, float x, float y, float z)

  Translate a triangle mesh by (x, y, z).
- void ae_tri_mesh_rotate_Euler_xyz (Tri_mesh mesh, float phi_deg, float theta_deg, float psi_deg)

  Rotate a triangle mesh using XYZ Euler angles (degrees).
- void ae_tri_mesh_set_bounding_box (Tri_mesh mesh, float ∗x_min, float ∗x_max, float ∗y_min, float ∗y_max, float ∗z_min, float ∗z_max)

  Compute the axis-aligned bounding box of a triangle mesh.
- void ae_tri_mesh_normalize (Tri_mesh mesh)

  Normalize mesh coordinates to [-1, 1], centered at origin.
- void ae_tri_mesh_flip_normals (Tri_mesh mesh)

  Flip triangle winding and recompute per-vertex normals.
- void ae_tri_mesh_set_normals (Tri_mesh mesh)

  Recompute per-vertex normals for all triangles in a mesh.
- void ae_quad_set_normals (Quad ∗quad)

  Compute and set per-vertex normals for a quad.
- Point ae_quad_get_average_normal (Quad quad)

  Compute the average of the four vertex normals of a quad.
- Point ae_quad_get_average_point (Quad quad)

  Compute the average of the four vertices of a quad.
- void ae_quad_calc_normal (Mat2D normal, Quad quad)

  Compute the face normal of a quad using the first three vertices.
- void ae_curve_copy (Curve ∗des, Curve src)

  Copy a Curve (ADA array of points).
- void ae_tri_calc_light_intensity (Tri ∗tri, Scene ∗scene, Lighting_mode lighting_mode)

  Compute per-vertex lighting intensity for a triangle.
- void ae_quad_calc_light_intensity (Quad ∗quad, Scene ∗scene, Lighting_mode lighting_mode)

Compute per-vertex lighting intensity for a quad.
- Point ae_line_itersect_plane (Mat2D plane_p, Mat2D plane_n, Mat2D line_start, Mat2D line_end, float ∗t)

  Intersect a line segment with a plane.
- int ae_line_clip_with_plane (Point start_in, Point end_in, Mat2D plane_p, Mat2D plane_n, Point ∗start_out, Point ∗end_out)

  Clip a line segment against a plane.
- float ae_signed_dist_point_and_plane (Point p, Mat2D plane_p, Mat2D plane_n)

  Signed distance from a point to a plane.
- int ae_tri_clip_with_plane (Tri tri_in, Mat2D plane_p, Mat2D plane_n, Tri ∗tri_out1, Tri ∗tri_out2)

  Clip a triangle against a plane.
- int ae_quad_clip_with_plane (Quad quad_in, Mat2D plane_p, Mat2D plane_n, Quad ∗quad_out1, Quad ∗quad_out2)

  Clip a quad against a plane.
- void ae_projection_mat_set (Mat2D proj_mat, float aspect_ratio, float FOV_deg, float z_near, float z_far)

  Build a perspective projection matrix.
- void ae_view_mat_set (Mat2D view_mat, Camera camera, Mat2D up)

  Build a right-handed view matrix from a Camera and up vector.
- Point ae_point_project_world2screen (Mat2D view_mat, Mat2D proj_mat, Point src, int window_w, int window_h)

  Project a point from world space directly to screen space.
- Point ae_point_project_world2view (Mat2D view_mat, Point src)

  Transform a point from world space to view space.
- Point ae_point_project_view2screen (Mat2D proj_mat, Point src, int window_w, int window_h)

  Project a view-space point to screen space.
- void ae_line_project_world2screen (Mat2D view_mat, Mat2D proj_mat, Point start_src, Point end_src, int window_w, int window_h, Point ∗start_des, Point ∗end_des, Scene ∗scene)

  Project and near-clip a world-space line segment to screen space.
- Tri ae_tri_transform_to_view (Mat2D view_mat, Tri tri)

  Transform a triangle from world space to view space.
- Tri_mesh ae_tri_project_world2screen (Mat2D proj_mat, Mat2D view_mat, Tri tri, int window_w, int window↩_h, Scene ∗scene, Lighting_mode lighting_mode)

  Project a single world-space triangle to screen space with clipping.
- void ae_tri_mesh_project_world2screen (Mat2D proj_mat, Mat2D view_mat, Tri_mesh ∗des, Tri_mesh src, int window_w, int window_h, Scene ∗scene, Lighting_mode lighting_mode)

  Project a triangle mesh from world to screen space with clipping.
- Quad ae_quad_transform_to_view (Mat2D view_mat, Quad quad)

  Transform a quad from world space to view space.
- Quad_mesh ae_quad_project_world2screen (Mat2D proj_mat, Mat2D view_mat, Quad quad, int window_w, int window_h, Scene ∗scene, Lighting_mode lighting_mode)

  Project a single world-space quad to screen space with clipping.
- void ae_quad_mesh_project_world2screen (Mat2D proj_mat, Mat2D view_mat, Quad_mesh ∗des, Quad_mesh src, int window_w, int window_h, Scene ∗scene, Lighting_mode lighting_mode)

  Project a quad mesh from world to screen space with clipping.
- void ae_curve_project_world2screen (Mat2D proj_mat, Mat2D view_mat, Curve ∗des, Curve src, int window_w, int window_h, Scene ∗scene)

  Project and clip a polyline (Curve) from world to screen space.
- void ae_curve_ada_project_world2screen (Mat2D proj_mat, Mat2D view_mat, Curve_ada ∗des, Curve_ada src, int window_w, int window_h, Scene ∗scene)

  Project and clip an array of polylines from world to screen space.
- void ae_grid_project_world2screen (Mat2D proj_mat, Mat2D view_mat, Grid des, Grid src, int window_w, int window_h, Scene ∗scene)

  Project and clip all polylines in a Grid from world to screen.

- void ae_tri_swap (Tri ∗v, int i, int j)
  Swap two triangles in an array.
- bool ae_tri_compare (Tri t1, Tri t2)
  Compare two triangles for sorting by depth.
- void ae_tri_qsort (Tri ∗v, int left, int right)
  Quicksort an array of triangles by depth.
- double ae_linear_map (double s, double min_in, double max_in, double min_out, double max_out)
  Linearly map a scalar from one range to another.
- void ae_z_buffer_copy_to_screen (Mat2D_uint32 screen_mat, Mat2D inv_z_buffer)
  Visualize an inverse-z buffer by writing a grayscale image.

4.7.1 Detailed Description

Software 3D rendering and scene utilities for meshes, camera, and projection.

A small, header-driven 3D engine providing:

- Scene and camera setup (projection/view matrices, Euler navigation).
- Triangle and quad mesh loading (OBJ/ASCII+binary STL), normalization, transforms, and per-vertex/face normals.
- Back-face culling, near-plane and screen-space polygon clipping.
- Perspective projection (world->view->screen) and line/grid helpers.
- Basic Phong-like lighting (ambient, diffuse, specular) with flat/smooth modes.
- Simple z-buffer visualization utility.

Inspiration This code is heavily inspired by the 3D engine of 'OneLoneCoder' in C++. You can find the source code in: `https://github.com/OneLoneCoder/Javidx9/tree/master/ConsoleGameEngine/BiggerProjects/Engine3D` . featured in this video of his: `https://youtu.be/ih20l3pJoeU?si=CzQ8rjk5ZEOlqEHN`.

Note

- Depends on Almog_Dynamic_Array.h, Matrix2D.h, Almog_Draw_Library.h, and Almog_String_Manipulation.h for math, containers, and I/O utilities.

- All public functions are prefixed with 'ae_'.

- Define ALMOG_ENGINE_IMPLEMENTATION in exactly one translation unit to compile the function bodies.

Definition in file Almog_Engine.h.

4.7.2 Macro Definition Documentation

4.7.2.1 AE_ASSERT

```
#define AE_ASSERT assert
```

Definition at line 44 of file Almog_Engine.h.

4.7.2.2 ae_assert_point_is_valid

```
#define ae_assert_point_is_valid(
            p )
```

Value:

```
AE_ASSERT(isfinite((p).x) && isfinite((p).y) && isfinite((p).z) && isfinite((p).w));    \
AE_ASSERT((p).x > -AE_MAX_POINT_VAL && (p).x < AE_MAX_POINT_VAL);
                    \
AE_ASSERT((p).y > -AE_MAX_POINT_VAL && (p).y < AE_MAX_POINT_VAL);
                    \
AE_ASSERT((p).z > -AE_MAX_POINT_VAL && (p).z < AE_MAX_POINT_VAL);
                    \
AE_ASSERT((p).w > -AE_MAX_POINT_VAL && (p).w < AE_MAX_POINT_VAL);
```

Definition at line 89 of file Almog_Engine.h.

4.7.2.3 ae_assert_quad_is_valid

```
#define ae_assert_quad_is_valid(
            quad )
```

Value:

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

Definition at line 97 of file Almog_Engine.h.

4.7.2.4 ae_assert_tri_is_valid

```
#define ae_assert_tri_is_valid(
            tri )
```

Value:

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

Definition at line 94 of file Almog_Engine.h.

4.7.2.5 AE_MAX_POINT_VAL

```
#define AE_MAX_POINT_VAL 1e5
```

Definition at line 88 of file Almog_Engine.h.

4.7.2.6 ae_point_add_point

```
#define ae_point_add_point(
            p,
            p1,
            p2 )
```

Value:

```
(p).x = (p1).x + (p2).x;  \
(p).y = (p1).y + (p2).y;                           \
(p).z = (p1).z + (p2).z;                           \
(p).w = (p1).w + (p2).w
```

Definition at line 105 of file Almog_Engine.h.

4.7.2.7 ae_point_calc_norma

```
#define ae_point_calc_norma(
            p ) sqrt(((p).x * (p).x) + ((p).y * (p).y) + ((p).z * (p).z))
```

Definition at line 104 of file Almog_Engine.h.

4.7.2.8 ae_point_dot_point

```
#define ae_point_dot_point(
            p1,
            p2 ) (((p1).x * (p2).x) + ((p1).y * (p2).y) + ((p1).z * (p2).z))
```

Definition at line 113 of file Almog_Engine.h.

4.7.2.9 ae_point_mult

```
#define ae_point_mult(
            p,
            const )
```

Value:

```
(p).x *= const; \
(p).y *= const;                     \
(p).z *= const
```

Definition at line 114 of file Almog_Engine.h.

4.7.2.10 ae_point_normalize_xyz_norma

```
#define ae_point_normalize_xyz_norma(
            p,
            norma )
```

Value:

```
(p).x = (p).x / norma;  \
(p).y = (p).y / norma;                      \
(p).z = (p).z / norma
```

Definition at line 101 of file Almog_Engine.h.

4.7.2.11 ae_point_sub_point

```
#define ae_point_sub_point(
            p,
            p1,
            p2 )
```

Value:

```
(p).x = (p1).x - (p2).x;  \
(p).y = (p1).y - (p2).y;                          \
(p).z = (p1).z - (p2).z;                          \
(p).w = (p1).w - (p2).w
```

Definition at line 109 of file Almog_Engine.h.

4.7.2.12 ae_points_equal

```
#define ae_points_equal(
            p1,
            p2 ) (p1).x == (p2).x && (p1).y == (p2).y && (p1).z == (p2).z
```

Definition at line 117 of file Almog_Engine.h.

4.7.2.13 AE_PRINT_MESH

```
#define AE_PRINT_MESH(
            mesh ) ae_print_tri_mesh(mesh, #mesh, 0)
```

Definition at line 267 of file Almog_Engine.h.

4.7.2.14 AE_PRINT_TRI

```
#define AE_PRINT_TRI(
              tri ) ae_print_tri(tri, #tri, 0)
```

Definition at line 266 of file Almog_Engine.h.

4.7.2.15 ARGB_hexARGB

```
#define ARGB_hexARGB(
              a,
              r,
              g,
              b ) 0x01000000l*(uint8_t)(a) + 0x00010000*(uint8_t)(r) + 0x00000100*(uint8_t)(g)
+ 0x00000001*(uint8_t)(b)
```

Definition at line 83 of file Almog_Engine.h.

4.7.2.16 QUAD_MESH_ARRAY

```
#define QUAD_MESH_ARRAY
```

Definition at line 136 of file Almog_Engine.h.

4.7.2.17 STL_ATTRIBUTE_BITS_SIZE

```
#define STL_ATTRIBUTE_BITS_SIZE 2
```

Definition at line 71 of file Almog_Engine.h.

4.7.2.18 STL_HEADER_SIZE

```
#define STL_HEADER_SIZE 80
```

Definition at line 59 of file Almog_Engine.h.

4.7.2.19 STL_NUM_SIZE

```
#define STL_NUM_SIZE 4
```

Definition at line 63 of file Almog_Engine.h.

4.7.2.20 STL_SIZE_FOREACH_TRI

```
#define STL_SIZE_FOREACH_TRI 50
```

Definition at line 67 of file Almog_Engine.h.

4.7.2.21 TRI_MESH_ARRAY

```
#define TRI_MESH_ARRAY
```

Definition at line 127 of file Almog_Engine.h.

4.7.3 Enumeration Type Documentation

4.7.3.1 Lighting_mode

```
enum Lighting_mode
```

Enumerator

AE_LIGHTING_FLAT	
AE_LIGHTING_SMOOTH	
AE_LIGHTING_MODE_LENGTH	

Definition at line 120 of file Almog_Engine.h.

4.7.4 Function Documentation

4.7.4.1 ae_camera_free()

```
void ae_camera_free (
            Scene * scene )
```

Free camera-related allocations in a Scene.

Frees all Mat2D objects owned by scene->camera (init_position, current_position, offset_position, direction, camera_x/y/z).

Parameters

scene	Scene whose camera resources will be freed.

Definition at line 366 of file Almog_Engine.h.

References Scene::camera, Camera::camera_x, Camera::camera_y, Camera::camera_z, Camera::current_position, Camera::direction, Camera::init_position, mat2D_free(), and Camera::offset_position.

Referenced by ae_scene_free().

4.7.4.2 ae_camera_init()

```
void ae_camera_init (
            Scene * scene,
            int window_h,
            int window_w )
```

Initialize the camera part of a Scene.

Sets perspective parameters (z_near, z_far, fov, aspect_ratio), allocates camera matrices/vectors, and sets initial position and orientation. The aspect ratio is computed as window_h / window_w.

Parameters

scene	Scene whose camera will be initialized.
window_h	Window height in pixels.
window_w	Window width in pixels.

Definition at line 320 of file Almog_Engine.h.

References Camera::aspect_ratio, Scene::camera, Camera::camera_x, Camera::camera_y, Camera::camera_z, Camera::current_position, Camera::direction, Camera::fov_deg, Camera::init_position, mat2D_alloc(), MAT2D_AT, mat2D_copy(), mat2D_fill(), Camera::offset_position, Camera::pitch_offset_deg, Camera::roll_offset_deg, Camera::yaw_offset_deg, Camera::z_far, and Camera::z_near.

Referenced by ae_scene_init().

4.7.4.3 ae_camera_reset_pos()

```
void ae_camera_reset_pos (
            Scene * scene )
```

Reset camera orientation and position to initial state.

Resets roll/pitch/yaw to zero, clears offset_position, restores camera basis vectors to identity, and copies current_position from init_position.

Parameters

scene	Scene containing the camera to reset.

Definition at line 472 of file Almog_Engine.h.

References Scene::camera, Camera::camera_x, Camera::camera_y, Camera::camera_z, Camera::current_position, Camera::init_position, MAT2D_AT, mat2D_copy(), mat2D_fill(), Camera::offset_position, Camera::pitch_offset_deg, Camera::roll_offset_deg, and Camera::yaw_offset_deg.

Referenced by process_input_window().

4.7.4.4 ae_curve_ada_project_world2screen()

```
void ae_curve_ada_project_world2screen (
            Mat2D proj_mat,
            Mat2D view_mat,
            Curve_ada * des,
            Curve_ada src,
            int window_w,
            int window_h,
            Scene * scene )
```

Project and clip an array of polylines from world to screen space.

Applies ae_curve_project_world2screen to each element in src and writes into the corresponding element in des. Arrays must be the same length.

Parameters

proj_mat	Projection matrix (4x4).
view_mat	View matrix (4x4).
des	Output array of curves (each overwritten).
src	Input array of world-space curves.
window_w	Screen width in pixels.
window_h	Screen height in pixels.
scene	Scene (camera for near plane).

Definition at line 3670 of file Almog_Engine.h.

References ae_curve_project_world2screen(), Curve_ada::elements, and Curve_ada::length.

4.7.4.5 ae_curve_copy()

```
void ae_curve_copy (
            Curve * des,
            Curve src )
```

Copy a Curve (ADA array of points).

Clears destination length and appends all points from src.

Parameters

des	Destination curve (modified/grown as needed).
src	Source curve.

Definition at line 1461 of file Almog_Engine.h.

References ada_append, Curve::elements, and Curve::length.

Referenced by ae_curve_project_world2screen().

4.7.4.6 ae_curve_project_world2screen()

```
void ae_curve_project_world2screen (
            Mat2D proj_mat,
            Mat2D view_mat,
            Curve * des,
            Curve src,
            int window_w,
            int window_h,
            Scene * scene )
```

Project and clip a polyline (Curve) from world to screen space.

Projects each segment with near-plane clipping and screen-edge clipping. Segments fully outside are removed. The destination curve is overwritten.

Note

This solution is not prefect. It sometimes delete one more edge then necessary, but I think that it won't brake.

Parameters

proj_mat	Projection matrix (4x4).
view_mat	View matrix (4x4).
des	Output curve (overwritten; ADA array grown as needed).
src	Input world-space curve.
window_w	Screen width in pixels.
window_h	Screen height in pixels.
scene	Scene (camera for near plane).

Definition at line 3558 of file Almog_Engine.h.

References ada_remove, ae_curve_copy(), ae_line_clip_with_plane(), ae_line_project_world2screen(), ae_points_equal, Curve::elements, Curve::length, mat2D_alloc(), MAT2D_AT, mat2D_fill(), and mat2D_free().

Referenced by ae_curve_ada_project_world2screen(), and ae_grid_project_world2screen().

4.7.4.7 ae_grid_project_world2screen()

```
void ae_grid_project_world2screen (
            Mat2D proj_mat,
            Mat2D view_mat,
            Grid des,
            Grid src,
            int window_w,
            int window_h,
            Scene * scene )
```

Project and clip all polylines in a Grid from world to screen.

Applies ae_curve_project_world2screen to each curve in the grid.

Parameters

proj_mat	Projection matrix (4x4).
view_mat	View matrix (4x4).
des	Output grid (curves overwritten).
src	Input world-space grid.
window_w	Screen width in pixels.
window_h	Screen height in pixels.
scene	Scene (camera for near plane).

Definition at line 3691 of file Almog_Engine.h.

References ae_curve_project_world2screen(), Grid::curves, Curve_ada::elements, and Curve_ada::length.

Referenced by update().

4.7.4.8 ae_line_clip_with_plane()

```
int ae_line_clip_with_plane (
            Point start_in,
            Point end_in,
            Mat2D plane_p,
            Mat2D plane_n,
```

```
Point * start_out,
Point * end_out )
```

Clip a line segment against a plane.

Returns the portion of the line segment [start_in, end_in] that lies on or inside the plane (signed distance >= 0). plane_n is normalized inside the function.

Parameters

start_in	Input start point (world or view space).
end_in	Input end point.
plane_p	Plane reference point (3x1).
plane_n	Plane normal (3x1).
start_out	Output clipped start point (if visible).
end_out	Output clipped end point (if visible).

Returns

int 0 if fully outside, 1 if fully or partially inside (outputs are valid), -1 on error.

Definition at line 1720 of file Almog_Engine.h.

References ae_assert_point_is_valid, ae_line_itersect_plane(), ae_point_to_mat2D(), ae_signed_dist_point_and_plane(), mat2D_alloc(), mat2D_free(), and mat2D_normalize.

Referenced by ae_curve_project_world2screen(), and ae_line_project_world2screen().

4.7.4.9 ae_line_itersect_plane()

```
Point ae_line_itersect_plane (
            Mat2D plane_p,
            Mat2D plane_n,
            Mat2D line_start,
            Mat2D line_end,
            float * t )
```

Intersect a line segment with a plane.

Computes intersection of segment [line_start, line_end] with the plane defined by point plane_p and normal plane_n. plane_n is normalized inside the function. The parameter t in [0, 1] is returned via out-parameter.

Note

The Mat2D objects line_start and line_end are temporarily modified internally; pass copies if you must preserve their values.

Parameters

plane_p	Plane reference point (3x1).
plane_n	Plane normal (3x1).
line_start	Segment start point (3x1).
line_end	Segment end point (3x1).
t	Output parametric distance along the segment (0=start, 1=end).

Returns

Point Intersection point in 3D.

Definition at line 1680 of file Almog_Engine.h.

References ae_mat2D_to_point(), mat2D_add(), mat2D_alloc(), mat2D_dot_product(), mat2D_fill(), mat2D_free(), mat2D_mult(), mat2D_normalize, and mat2D_sub().

Referenced by ae_line_clip_with_plane(), ae_quad_clip_with_plane(), and ae_tri_clip_with_plane().

4.7.4.10 ae_line_project_world2screen()

```
void ae_line_project_world2screen (
            Mat2D view_mat,
            Mat2D proj_mat,
            Point start_src,
            Point end_src,
            int window_w,
            int window_h,
            Point * start_des,
            Point * end_des,
            Scene * scene )
```

Project and near-clip a world-space line segment to screen space.

Transforms the segment to view space, clips it against the near plane at z = z_near + 0.01, then projects to screen space. If fully clipped, both outputs are set to the sentinel (-1, -1, 1, 1).

Parameters

view_mat	View matrix (4x4).
proj_mat	Projection matrix (4x4).
start_src	World-space start point.
end_src	World-space end point.
window↔_w	Screen width in pixels.
window↔_h	Screen height in pixels.
start_des	Output screen-space start point (or sentinel).
end_des	Output screen-space end point (or sentinel).
scene	Scene (used for near plane distance).

Definition at line 2933 of file Almog_Engine.h.

References ae_line_clip_with_plane(), ae_point_project_view2screen(), ae_point_project_world2view(), Scene::camera, mat2D_alloc(), MAT2D_AT, mat2D_fill(), mat2D_free(), and Camera::z_near.

Referenced by ae_curve_project_world2screen().

4.7.4.11 ae_linear_map()

```
double ae_linear_map (
            double s,
            double min_in,
            double max_in,
            double min_out,
            double max_out )
```

Linearly map a scalar from one range to another.

Computes min_out + (s - min_in) ∗ (max_out - min_out) / (max_in - min_in).

Parameters

s	Input scalar.
min_in	Input range minimum.
max_in	Input range maximum.
min_out	Output range minimum.
max_out	Output range maximum.

Returns

double Mapped scalar.

Definition at line 3770 of file Almog_Engine.h.

Referenced by ae_z_buffer_copy_to_screen().

4.7.4.12 ae_mat2D_to_point()

```
Point ae_mat2D_to_point (
            Mat2D m )
```

Read a 3x1 Mat2D vector into a Point.

Reads x, y, z from m(0..2,0) and returns a Point with w=1.

Parameters

m	Source matrix (3x1).

Returns

Point The corresponding point with w=1.

Definition at line 522 of file Almog_Engine.h.

References MAT2D_AT, and Point::x.

Referenced by ae_line_itersect_plane(), ae_quad_calc_light_intensity(), ae_quad_set_normals(), ae_tri_calc_light_intensity(), and ae_tri_set_normals().

4.7.4.13 ae_point_normalize_xyz()

```
Point ae_point_normalize_xyz (
            Point p )
```

Normalize a point's xyz to unit length.

Divides x, y, z by their Euclidean norm. w is preserved unchanged.

Parameters

p	Input point.

Returns

Point Unit-length point (xyz), with original w.

Definition at line 976 of file Almog_Engine.h.

References ae_point_calc_norma, Point::w, Point::x, Point::y, and Point::z.

Referenced by ae_quad_calc_light_intensity(), ae_quad_get_average_normal(), ae_scene_init(), ae_tri_calc_light_intensity(), and ae_tri_get_average_normal().

4.7.4.14 ae_point_project_view2screen()

```
Point ae_point_project_view2screen (
            Mat2D proj_mat,
            Point src,
            int window_w,
            int window_h )
```

Project a view-space point to screen space.

Applies the projection matrix, performs perspective divide if $|w| > 1e\text{-}3$, maps normalized device coords to pixel coordinates: x_screen = (x_ndc + 1) $*$ 0.5 $*$ window_w y_screen = (y_ndc + 1) $*$ 0.5 $*$ window_h

z is z_ndc, w is the clip-space w (or 1 if the original w $\sim$ 0).

Parameters

proj_mat	Projection matrix (4x4).
src	View-space point.
window↩_w	Screen width in pixels.
window↩_h	Screen height in pixels.

Returns

Point Screen-space point.

Definition at line 2870 of file Almog_Engine.h.

References ae_assert_point_is_valid, mat2D_alloc(), MAT2D_AT, mat2D_dot(), mat2D_free(), Point::w, Point::x, Point::y, and Point::z.

Referenced by ae_line_project_world2screen(), ae_point_project_world2screen(), ae_quad_project_world2screen(), and ae_tri_project_world2screen().

4.7.4.15 ae_point_project_world2screen()

```
Point ae_point_project_world2screen (
            Mat2D view_mat,
            Mat2D proj_mat,
            Point src,
            int window_w,
            int window_h )
```

Project a point from world space directly to screen space.

Combines ae_point_project_world2view and ae_point_project_view2screen.

Parameters

view_mat	View matrix (4x4).
proj_mat	Projection matrix (4x4).
src	World-space point.
window↩_w	Screen width in pixels.
window↩_h	Screen height in pixels.

Returns

Point Screen-space point (x,y in pixels). z is post-projection z/w, w is clip-space w.

Definition at line 2806 of file Almog_Engine.h.

References ae_point_project_view2screen(), and ae_point_project_world2view().

4.7.4.16 ae_point_project_world2view()

```
Point ae_point_project_world2view (
            Mat2D view_mat,
            Point src )
```

Transform a point from world space to view space.

Multiplies [x y z 1] by view_mat (row-vector convention in this code). Returns the resulting view-space point; w should be 1.

Parameters

view_mat	View matrix (4x4).
src	World-space point.

Returns

Point View-space point (w=1).

Definition at line 2824 of file Almog_Engine.h.

References AE_ASSERT, ae_assert_point_is_valid, mat2D_alloc(), MAT2D_AT, mat2D_dot(), mat2D_free(), Point::w, Point::x, Point::y, and Point::z.

Referenced by ae_line_project_world2screen(), and ae_point_project_world2screen().

4.7.4.17 ae_point_to_mat2D()

```
void ae_point_to_mat2D (
            Point p,
            Mat2D m )
```

Write a Point into a Mat2D vector.

Writes p into m. m must be either 3x1 or 1x3. Only x, y, z are written.

Parameters

p	Source point (x, y, z used; w ignored).
m	Destination matrix (3x1 or 1x3).

Definition at line 498 of file Almog_Engine.h.

References Mat2D::cols, MAT2D_AT, MATRIX2D_ASSERT, Mat2D::rows, Point::x, Point::y, and Point::z.

Referenced by ae_line_clip_with_plane(), ae_quad_calc_normal(), ae_quad_clip_with_plane(), ae_quad_project_world2screen(), ae_quad_set_normals(), ae_tri_calc_normal(), ae_tri_clip_with_plane(), ae_tri_project_world2screen(), and ae_tri_set_normals().

4.7.4.18 ae_print_points()

```
void ae_print_points (
            Curve p )
```

Print a list of points to stdout.

Each point is printed as: "point i: (x, y, z)".

Parameters

p	Curve of points to print.

Definition at line 925 of file Almog_Engine.h.

References Curve::elements, Curve::length, Point::x, Point::y, and Point::z.

4.7.4.19 ae_print_tri()

```
void ae_print_tri (
            Tri tri,
            char * name,
            size_t padding )
```

Print a triangle to stdout.

Prints the triangle's vertices and draw flag, with an optional name and indentation padding (spaces).

Parameters

tri	Triangle to print.
name	Label to print before the triangle.
padding	Number of leading spaces for indentation.

Definition at line 942 of file Almog_Engine.h.

References Tri::points, Tri::to_draw, Point::x, Point::y, and Point::z.

Referenced by ae_print_tri_mesh().

4.7.4.20 ae_print_tri_mesh()

```
void ae_print_tri_mesh (
            Tri_mesh mesh,
            char * name,
            size_t padding )
```

Print all triangles in a mesh to stdout.

Each triangle is printed via ae_print_tri with the given padding.

Parameters

mesh	Triangle mesh to print.
name	Label for the mesh.
padding	Number of leading spaces for indentation.

Definition at line 958 of file Almog_Engine.h.

References ae_print_tri(), Tri_mesh::elements, and Tri_mesh::length.

4.7.4.21 ae_projection_mat_set()

```
void ae_projection_mat_set (
            Mat2D proj_mat,
            float aspect_ratio,
            float FOV_deg,
            float z_near,
            float z_far )
```

Build a perspective projection matrix.

proj_mat must be 4x4. FOV is in degrees. The matrix maps view-space to clip space consistent with the engine's pipeline; z is mapped using z_far/(z_far - z_near).

Parameters

proj_mat	Output 4x4 projection matrix.
aspect_ratio	aspect = window_h / window_w.
FOV_deg	Vertical field of view in degrees (must be $>$ 0).
z_near	Near clipping plane distance ($>$ 0).
z_far	Far clipping plane distance ($>$ z_near).

Definition at line 2682 of file Almog_Engine.h.

References AE_ASSERT, Mat2D::cols, MAT2D_AT, mat2D_fill(), PI, and Mat2D::rows.

Referenced by ae_scene_init(), and update().

4.7.4.22 ae_quad_calc_light_intensity()

```
void ae_quad_calc_light_intensity (
            Quad * quad,
            Scene * scene,
            Lighting_mode lighting_mode )
```

Compute per-vertex lighting intensity for a quad.

Same model as ae_tri_calc_light_intensity, applied to four vertices. Results are clamped to [0, 1].

Parameters

quad	Quad to update (quad->light_intensity[i] is written).
scene	Scene providing light and material parameters.
lighting_mode	Flat or smooth lighting mode.

Definition at line 1580 of file Almog_Engine.h.

References AE_LIGHTING_FLAT, AE_LIGHTING_SMOOTH, ae_mat2D_to_point(), ae_point_add_point, ae_point_dot_point, ae_point_mult, ae_point_normalize_xyz(), ae_point_sub_point, ae_quad_get_average_normal(), ae_quad_get_average_point(), Material::c_ambi, Material::c_diff, Material::c_spec, Scene::camera, Camera::current_position, Light_source::light_direction_or_pos, Quad::light_intensity, Light_source::light_intensity, Scene::light_source0, Scene::material0, Quad::normals, Quad::points, Material::specular_power_alpha, and Point::w.

Referenced by ae_quad_project_world2screen().

4.7.4.23 ae_quad_calc_normal()

```
void ae_quad_calc_normal (
            Mat2D normal,
            Quad quad )
```

Compute the face normal of a quad using the first three vertices.

normal must be a 3x1 vector. The function writes the normalized cross product of (p1 - p0) x (p2 - p0) into normal.

Parameters

normal	Output 3x1 vector for the face normal.
quad	Input quad.

Definition at line 1428 of file Almog_Engine.h.

References AE_ASSERT, ae_assert_quad_is_valid, ae_point_to_mat2D(), Mat2D::cols, mat2D_alloc(), mat2D_calc_norma(), mat2D_cross(), mat2D_free(), mat2D_mult(), mat2D_sub(), Quad::points, and Mat2D::rows.

4.7.4.24 ae_quad_clip_with_plane()

```
int ae_quad_clip_with_plane (
            Quad quad_in,
            Mat2D plane_p,
            Mat2D plane_n,
            Quad * quad_out1,
            Quad * quad_out2 )
```

Clip a quad against a plane.

Splits or discards the quad quad_in against the plane defined by (plane_p, plane_n). plane_n is normalized inside the function.

Parameters

quad_in	Input quad.
plane_p	Plane reference point (3x1).
plane_n	Plane normal (3x1).
quad_out1	First output quad (if any). When output count is 2, this holds one of the resulting polygons (possibly as a quad composed from intersections).
quad_out2	Second output quad (if split).

Returns

int Number of output polygons: 0 (culled), 1, or 2. Returns -1 on error.

Definition at line 2181 of file Almog_Engine.h.

References ae_assert_quad_is_valid, ae_line_itersect_plane(), ae_point_to_mat2D(), ae_signed_dist_point_and_plane(), Quad::colors, mat2D_alloc(), mat2D_free(), mat2D_normalize, Quad::points, and Point::w.

Referenced by ae_quad_mesh_project_world2screen(), and ae_quad_project_world2screen().

4.7.4.25 ae_quad_get_average_normal()

```
Point ae_quad_get_average_normal (
            Quad quad )
```

Compute the average of the four vertex normals of a quad.

Averages the four vertex normals and normalizes the result.

Parameters

quad	Input quad.

Returns

Point The averaged, normalized normal.

Definition at line 1379 of file Almog_Engine.h.

References ae_point_normalize_xyz(), Quad::normals, Point::w, Point::x, Point::y, and Point::z.

Referenced by ae_quad_calc_light_intensity(), and ae_quad_project_world2screen().

4.7.4.26 ae_quad_get_average_point()

```
Point ae_quad_get_average_point (
            Quad quad )
```

Compute the average of the four vertices of a quad.

Parameters

quad	Input quad.

Returns

Point The average point (x, y, z, w are simple averages).

Definition at line 1403 of file Almog_Engine.h.

References Quad::points, Point::w, Point::x, Point::y, and Point::z.

Referenced by ae_quad_calc_light_intensity().

4.7.4.27 ae_quad_mesh_project_world2screen()

```
void ae_quad_mesh_project_world2screen (
            Mat2D proj_mat,
            Mat2D view_mat,
            Quad_mesh * des,
            Quad_mesh src,
            int window_w,
            int window_h,
            Scene * scene,
            Lighting_mode lighting_mode )
```

Project a quad mesh from world to screen space with clipping.

Iterates over all quads, applies near-plane and screen-edge clipping, and writes results into des. Quads can be split by clipping, so des may end up with more elements than src.

Parameters

proj_mat	Projection matrix (4x4).
view_mat	View matrix (4x4).
des	Output mesh (cleared and filled; ADA array grown as needed).
src	Input world-space quad mesh.
window_w	Screen width in pixels.
window_h	Screen height in pixels.
scene	Scene (camera/light/material).
lighting_mode	Flat or smooth lighting mode.

Definition at line 3434 of file Almog_Engine.h.

References ada_append, ada_insert_unordered, ada_remove_unordered, ae_assert_quad_is_valid, ae_quad_clip_with_plane(), ae_quad_project_world2screen(), Quad_mesh::elements, Quad_mesh::length, mat2D_alloc(), MAT2D_AT, mat2D_fill(), and mat2D_free().

4.7.4.28 ae_quad_project_world2screen()

```
Quad_mesh ae_quad_project_world2screen (
            Mat2D proj_mat,
            Mat2D view_mat,
            Quad quad,
            int window_w,
            int window_h,
            Scene * scene,
            Lighting_mode lighting_mode )
```

Project a single world-space quad to screen space with clipping.

Computes lighting and visibility, transforms to view space, clips against near plane, and projects to screen space. A quad may produce one or two quads after clipping.

Parameters

proj_mat	Projection matrix (4x4).
view_mat	View matrix (4x4).
quad	World-space quad.
window_w	Screen width in pixels.
window_h	Screen height in pixels.
scene	Scene (camera/light/material).
lighting_mode	Flat or smooth lighting mode.

Returns

Quad_mesh An ADA array of resulting screen-space quads. Caller must free result.elements.

Definition at line 3321 of file Almog_Engine.h.

References ada_append, ada_init_array, ae_assert_quad_is_valid, ae_point_project_view2screen(), ae_point_to_mat2D(), ae_quad_calc_light_intensity(), ae_quad_clip_with_plane(), ae_quad_get_average_normal(), ae_quad_transform_to_view(), Scene::camera, Camera::current_position, Quad_mesh::elements, Quad_mesh::length, Light_source::light_direction_or_pos, Quad::light_intensity, Scene::light_source0, mat2D_alloc(), MAT2D_AT, mat2D_fill(), mat2D_free(), mat2D_sub(), Quad::normals, Quad::points, Quad::to_draw, Point::x, Point::y, Point::z, and Camera::z_near.

Referenced by ae_quad_mesh_project_world2screen().

4.7.4.29 ae_quad_set_normals()

```
void ae_quad_set_normals (
            Quad * quad )
```

Compute and set per-vertex normals for a quad.

For each vertex, computes the cross product of adjacent edges and normalizes the result. Results are stored in quad->normals[i].

Parameters

quad	Quad whose normals will be computed and written.

Definition at line 1336 of file Almog_Engine.h.

References ae_assert_quad_is_valid, ae_mat2D_to_point(), ae_point_to_mat2D(), mat2D_alloc(), mat2D_copy(), mat2D_cross(), mat2D_free(), mat2D_normalize, mat2D_sub(), Quad::normals, and Quad::points.

4.7.4.30 ae_quad_transform_to_view()

```
Quad ae_quad_transform_to_view (
            Mat2D view_mat,
            Quad quad )
```

Transform a quad from world space to view space.

Applies view_mat to each vertex (homogeneous multiply with w=1). Returns the transformed quad; normals are not changed.

Parameters

view_mat	View matrix (4x4).
quad	World-space quad.

Returns

Quad View-space quad.

Definition at line 3271 of file Almog_Engine.h.

References AE_ASSERT, ae_assert_quad_is_valid, mat2D_alloc(), MAT2D_AT, mat2D_dot(), mat2D_free(), Quad::points, Point::w, Point::x, Point::y, and Point::z.

Referenced by ae_quad_project_world2screen().

4.7.4.31 ae_scene_free()

```
void ae_scene_free (
            Scene * scene )
```

Free all resources owned by a Scene.

Frees camera, matrices, and any allocated meshes in the in_world, projected, and original mesh arrays (triangles and quads). Does not free the Scene struct itself when passed by pointer.

Parameters

scene	Scene to free.

Note

Assumes the game_state was initialized with zeros.

Definition at line 429 of file Almog_Engine.h.

References ae_camera_free(), Tri_mesh::elements, Quad_mesh::elements, Tri_mesh_array::elements, Quad_mesh_array::elements, Scene::in_world_quad_meshes, Scene::in_world_tri_meshes, Tri_mesh_array::length, Quad_mesh_array::length, mat2D_free(), Scene::original_quad_meshes, Scene::original_tri_meshes, Scene::proj_mat, Scene::projected_quad_meshes, Scene::projected_tri_meshes, Scene::up_direction, and Scene::view_mat.

Referenced by destroy_window().

4.7.4.32 ae_scene_init()

```
Scene ae_scene_init (
            int window_h,
            int window_w )
```

Create and initialize a Scene.

Initializes camera, up direction, default light and material, and allocates projection and view matrices. Caller must release resources with ae_scene_free.

Parameters

window↩_h	Window height in pixels.
window↩_w	Window width in pixels.

Returns

Scene An initialized scene object.

Definition at line 388 of file Almog_Engine.h.

References ae_camera_init(), ae_point_normalize_xyz(), ae_projection_mat_set(), ae_view_mat_set(), Camera::aspect_ratio, Material::c_ambi, Material::c_diff, Material::c_spec, Scene::camera, Camera::fov_deg, Light_source::light_direction_or_pos, Light_source::light_intensity, Scene::light_source0, mat2D_alloc(), MAT2D_AT, mat2D_fill(), Scene::material0, Scene::proj_mat, Material::specular_power_alpha, Scene::up_direction, Scene::view_mat, Point::w, Point::x, Point::y, Point::z, Camera::z_far, and Camera::z_near.

Referenced by setup_window().

4.7.4.33 ae_signed_dist_point_and_plane()

```
float ae_signed_dist_point_and_plane (
            Point p,
            Mat2D plane_p,
            Mat2D plane_n )
```

Signed distance from a point to a plane.

Computes dot(n, p) - dot(n, plane_p). The normal is not normalized internally; pass a normalized plane_n for distances in consistent units.

Parameters

p	Point to evaluate.
plane↩_p	Plane reference point (3x1).
plane↩_n	Plane normal (3x1).

Returns

float Signed distance (>=0 is on the "inside" of the plane).

Definition at line 1807 of file Almog_Engine.h.

References ae_assert_point_is_valid, MAT2D_AT, Point::x, Point::y, and Point::z.

Referenced by ae_line_clip_with_plane(), ae_quad_clip_with_plane(), and ae_tri_clip_with_plane().

4.7.4.34 ae_tri_calc_light_intensity()

```
void ae_tri_calc_light_intensity (
            Tri * tri,
            Scene * scene,
            Lighting_mode lighting_mode )
```

Compute per-vertex lighting intensity for a triangle.

Implements a Phong-like model with ambient, diffuse, and specular terms, using material0 and light_source0 from the scene. When lighting_mode is AE_LIGHTING_FLAT, the average normal and triangle centroid are used for all vertices; when AE_LIGHTING_SMOOTH, each vertex normal and position is used. For directional light, light_↩ direction_or_pos.w == 0; for point light, w != 0. Results are clamped to [0, 1].

Parameters

tri	Triangle to update (tri->light_intensity[i] is written).
scene	Scene providing light and material parameters.
lighting_mode	Flat or smooth lighting mode.

Definition at line 1487 of file Almog_Engine.h.

References AE_LIGHTING_FLAT, AE_LIGHTING_SMOOTH, ae_mat2D_to_point(), ae_point_add_point, ae_point_dot_point, ae_point_mult, ae_point_normalize_xyz(), ae_point_sub_point, ae_tri_get_average_normal(), ae_tri_get_average_point(), Material::c_ambi, Material::c_diff, Material::c_spec, Scene::camera, Camera::current_position, Light_source::light_direction_or_pos, Tri::light_intensity, Light_source::light_intensity, Scene::light_source0, Scene::material0, Tri::normals, Tri::points, Material::specular_power_alpha, and Point::w.

Referenced by ae_tri_project_world2screen().

4.7.4.35 ae_tri_calc_normal()

```
void ae_tri_calc_normal (
            Mat2D normal,
            Tri tri )
```

Compute the face normal of a triangle.

normal must be a 3x1 vector. The function writes the normalized cross product of (p1 - p0) x (p2 - p0) into normal.

Parameters

normal	Output 3x1 vector for the face normal.
tri	Input triangle.

Definition at line 1086 of file Almog_Engine.h.

References AE_ASSERT, ae_assert_tri_is_valid, ae_point_to_mat2D(), Mat2D::cols, mat2D_alloc(), mat2D_calc_norma(), mat2D_cross(), mat2D_free(), mat2D_mult(), mat2D_sub(), Tri::points, and Mat2D::rows.

Referenced by ae_tri_project_world2screen().

4.7.4.36 ae_tri_clip_with_plane()

```
int ae_tri_clip_with_plane (
            Tri tri_in,
            Mat2D plane_p,
            Mat2D plane_n,
            Tri * tri_out1,
            Tri * tri_out2 )
```

Clip a triangle against a plane.

Splits or discards the triangle tri_in against the plane defined by (plane_p, plane_n). plane_n is normalized inside the function.

Parameters

tri_in	Input triangle.
plane←↩ _p	Plane reference point (3x1).
plane←↩ _n	Plane normal (3x1).
tri_out1	First output triangle (if any).
tri_out2	Second output triangle (if split).

Returns

int Number of output triangles: 0 (culled), 1, or 2. Returns -1 on error.

Definition at line 1838 of file Almog_Engine.h.

References ae_assert_tri_is_valid, ae_line_itersect_plane(), ae_point_to_mat2D(), ae_signed_dist_point_and_plane(), Tri::colors, mat2D_alloc(), mat2D_free(), mat2D_normalize, Tri::points, Tri::tex_points, Point::w, Point::x, and Point::y.

Referenced by ae_tri_mesh_project_world2screen(), and ae_tri_project_world2screen().

4.7.4.37 ae_tri_compare()

```
bool ae_tri_compare (
            Tri t1,
            Tri t2 )
```

Compare two triangles for sorting by depth.

Returns true if t1 should come before t2 when sorting by the maximum z of their vertices (descending order).

Parameters

t1	First triangle.
t2	Second triangle.

Returns

bool true if t1 precedes t2, false otherwise.

Definition at line 3725 of file Almog_Engine.h.

References Tri::points, and Point::z.

Referenced by ae_tri_qsort().

4.7.4.38 ae_tri_create()

```
Tri ae_tri_create (
            Point p1,
            Point p2,
            Point p3 )
```

Create a triangle from three points.

Parameters

p1	First vertex (world space).
p2	Second vertex (world space).
p3	Third vertex (world space).

Returns

Tri The created triangle with vertices set. Other fields are left uninitialized.

Definition at line 278 of file Almog_Engine.h.

References Tri::points.

4.7.4.39 ae_tri_get_average_normal()

```
Point ae_tri_get_average_normal (
            Tri tri )
```

Compute the average of the three vertex normals of a triangle.

Averages the three vertex normals and normalizes the result.

Parameters

tri	Input triangle.

Returns

Point The averaged, normalized normal (w averaged but unused).

Definition at line 1041 of file Almog_Engine.h.

References ae_point_normalize_xyz(), Tri::normals, Point::w, Point::x, Point::y, and Point::z.

Referenced by ae_tri_calc_light_intensity().

4.7.4.40 ae_tri_get_average_point()

```
Point ae_tri_get_average_point (
            Tri tri )
```

Compute the average of the three vertices of a triangle.

Parameters

tri	Input triangle.

Returns

Point The average point (x, y, z, w are simple averages).

Definition at line 1062 of file Almog_Engine.h.

References Tri::points, Point::w, Point::x, Point::y, and Point::z.

Referenced by ae_tri_calc_light_intensity().

4.7.4.41 ae_tri_mesh_append_copy()

```
void ae_tri_mesh_append_copy (
            Tri_mesh_array * mesh_array,
            Tri_mesh mesh )
```

Append a copy of a Tri_mesh into a Tri_mesh_array.

Creates a deep copy of mesh (triangles by value) and appends it to mesh_array (ADA array of meshes).

Parameters

mesh_array	Destination mesh array to append into.
mesh	Source triangle mesh to copy.

Definition at line 851 of file Almog_Engine.h.

References ada_append, ada_init_array, Tri_mesh::elements, and Tri_mesh::length.

Referenced by setup().

4.7.4.42 ae_tri_mesh_create_copy()

```
void ae_tri_mesh_create_copy (
            Tri_mesh * des,
            Tri * src_elements,
            size_t len )
```

Append copies of triangles to a destination Tri_mesh (resets destination length first).

Appends len triangles from src_elements into the destination ADA array pointed by des.

Parameters

des	Destination triangle mesh (ADA array). Will grow as needed.
src_elements	Source array of triangles to copy from.
len	Number of triangles to copy from src_elements.

Definition at line 299 of file Almog_Engine.h.

References ada_append, and Tri_mesh::length.

4.7.4.43 ae_tri_mesh_flip_normals()

```
void ae_tri_mesh_flip_normals (
            Tri_mesh mesh )
```

Flip triangle winding and recompute per-vertex normals.

Swaps vertex order to invert winding, copies attributes accordingly, and recomputes normals.

Parameters

mesh	Mesh to flip (modified in place).

Definition at line 1283 of file Almog_Engine.h.

References ae_tri_set_normals(), Tri::colors, Tri_mesh::elements, Tri_mesh::length, Tri::light_intensity, Tri::normals, Tri::points, Tri::tex_points, and Tri::to_draw.

4.7.4.44 ae_tri_mesh_get_from_file()

```
Tri_mesh ae_tri_mesh_get_from_file (
            char * file_path )
```

Load a triangle mesh from a file (OBJ or STL).

Dispatches to ae_tri_mesh_get_from_obj_file or ae_tri_mesh_get_from_stl_file based on file extension.

Parameters

file_path	Path to the file (.obj, .stl, .STL).

Returns

Tri_mesh The loaded triangle mesh. Caller must free mesh.elements when done.

Definition at line 816 of file Almog_Engine.h.

References ae_tri_mesh_get_from_obj_file(), ae_tri_mesh_get_from_stl_file(), asm_get_word_and_cut(), ASM_MAX_LEN_LINE, and asm_str_in_str().

Referenced by setup().

4.7.4.45 ae_tri_mesh_get_from_obj_file()

```
Tri_mesh ae_tri_mesh_get_from_obj_file (
            char * file_path )
```

Load a triangle mesh from a Wavefront OBJ file.

Supports vertex positions (v). Face lines (f) with 3 or 4 vertices are parsed. Texture coordinates and normals in the file are ignored (a warning is printed once if present). Quads are triangulated as (0,1,2) and (2,3,0). Colors are set to white and to_draw is set to true.

Parameters

file_path	Path to the OBJ file.

Returns

Tri_mesh The loaded triangle mesh. Caller must free mesh.elements when done.

Definition at line 540 of file Almog_Engine.h.

References ada_append, ada_init_array, asm_get_line(), asm_get_next_word_from_line(), asm_get_word_and_cut(), asm_length(), ASM_MAX_LEN_LINE, asm_str_in_str(), Tri::colors, Curve::elements, Tri::light_intensity, Tri::points, Tri::to_draw, Point::x, Point::y, and Point::z.

Referenced by ae_tri_mesh_get_from_file().

4.7.4.46 ae_tri_mesh_get_from_quad_mesh()

```
Tri_mesh ae_tri_mesh_get_from_quad_mesh (
            Quad_mesh q_mesh )
```

Convert a Quad_mesh into a Tri_mesh.

Splits each quad into two triangles: (0,1,2) and (2,3,0), copying per-vertex attributes (points, colors, normals, light intensities).

Parameters

q_mesh	Input quad mesh.

Returns

Tri_mesh Resulting triangle mesh. Caller must free mesh.elements when done.

Definition at line 875 of file Almog_Engine.h.

References ada_append, ada_init_array, Tri::colors, Quad::colors, Quad_mesh::elements, Quad_mesh::length, Tri::light_intensity, Quad::light_intensity, Tri::normals, Quad::normals, Tri::points, Quad::points, Tri::to_draw, and Quad::to_draw.

4.7.4.47 ae_tri_mesh_get_from_stl_file()

```
Tri_mesh ae_tri_mesh_get_from_stl_file (
            char * file_path )
```

Load a triangle mesh from a binary STL file.

Reads binary STL (little-endian). Per-triangle normals from the file are negated to match the engine's convention and copied to each vertex normal. Colors are set to white and to_draw is set to true.

Parameters

file_path	Path to the binary STL file.

Returns

Tri_mesh The loaded triangle mesh. Caller must free mesh.elements when done.

Definition at line 743 of file Almog_Engine.h.

References ada_append, ada_init_array, Tri::colors, Tri::light_intensity, Tri::normals, Tri::points, STL_ATTRIBUTE_BITS_SIZE, STL_HEADER_SIZE, STL_NUM_SIZE, Tri::to_draw, Point::x, Point::y, and Point::z.

Referenced by ae_tri_mesh_get_from_file().

4.7.4.48 ae_tri_mesh_normalize()

```
void ae_tri_mesh_normalize (
            Tri_mesh mesh )
```

Normalize mesh coordinates to [-1, 1], centered at origin.

Uniformly scales and recenters the mesh so that the largest axis fits exactly into [-1, 1]. Other axes are scaled proportionally. Updates all vertices in place.

Parameters

mesh	Triangle mesh to normalize (modified in place).

Definition at line 1244 of file Almog_Engine.h.

References ae_tri_mesh_set_bounding_box(), Tri_mesh::elements, Tri_mesh::length, Tri::points, Point::x, Point::y, and Point::z.

Referenced by setup().

4.7.4.49 ae_tri_mesh_project_world2screen()

```
void ae_tri_mesh_project_world2screen (
            Mat2D proj_mat,
            Mat2D view_mat,
            Tri_mesh * des,
            Tri_mesh src,
            int window_w,
            int window_h,
            Scene * scene,
            Lighting_mode lighting_mode )
```

Project a triangle mesh from world to screen space with clipping.

Iterates over all triangles, applies near-plane and screen-edge clipping (top/right/bottom/left), and writes results into des. Triangles can be split by clipping, so des may end up with more elements than src.

Parameters

proj_mat	Projection matrix (4x4).
view_mat	View matrix (4x4).
des	Output mesh (cleared and filled; ADA array grown as needed).
src	Input world-space triangle mesh.
window_w	Screen width in pixels.
window_h	Screen height in pixels.
scene	Scene (camera/light/material).
lighting_mode	Flat or smooth lighting mode.

Definition at line 3151 of file Almog_Engine.h.

References ada_append, ada_insert_unordered, ada_remove_unordered, ae_assert_tri_is_valid, ae_tri_clip_with_plane(), ae_tri_project_world2screen(), Tri_mesh::elements, Tri_mesh::length, mat2D_alloc(), MAT2D_AT, mat2D_fill(), and mat2D_free().

Referenced by update().

4.7.4.50 ae_tri_mesh_rotate_Euler_xyz()

```
void ae_tri_mesh_rotate_Euler_xyz (
            Tri_mesh mesh,
            float phi_deg,
            float theta_deg,
            float psi_deg )
```

Rotate a triangle mesh using XYZ Euler angles (degrees).

Applies DCM = Cz(psi_deg) $*$ Cy(theta_deg) $*$ Cx(phi_deg) to each vertex. Recomputes per-vertex normals afterward.

Parameters

mesh	Triangle mesh to rotate (modified in place).
phi_deg	Rotation about X axis, degrees.
theta_deg	Rotation about Y axis, degrees.
psi_deg	Rotation about Z axis, degrees.

Definition at line 1143 of file Almog_Engine.h.

References ae_tri_mesh_set_normals(), Tri_mesh::elements, Tri_mesh::length, mat2D_alloc(), MAT2D_AT, mat2D_dot(), mat2D_free(), mat2D_set_rot_mat_x(), mat2D_set_rot_mat_y(), mat2D_set_rot_mat_z(), Tri::points, Point::x, Point::y, and Point::z.

Referenced by setup().

4.7.4.51 ae_tri_mesh_set_bounding_box()

```
void ae_tri_mesh_set_bounding_box (
            Tri_mesh mesh,
            float * x_min,
            float * x_max,
            float * y_min,
            float * y_max,
            float * z_min,
            float * z_max )
```

Compute the axis-aligned bounding box of a triangle mesh.

Writes min/max for x, y, z across all vertices in the mesh.

Parameters

mesh	Input triangle mesh.
x_min	Output minimum x.
x_max	Output maximum x.
y_min	Output minimum y.
y_max	Output maximum y.
z_min	Output minimum z.
z_max	Output maximum z.

Definition at line 1206 of file Almog_Engine.h.

References Tri_mesh::elements, Tri_mesh::length, Tri::points, Point::x, Point::y, and Point::z.

Referenced by ae_tri_mesh_normalize().

4.7.4.52 ae_tri_mesh_set_normals()

```
void ae_tri_mesh_set_normals (
            Tri_mesh mesh )
```

Recompute per-vertex normals for all triangles in a mesh.

Calls ae_tri_set_normals on each triangle.

Parameters

mesh	Mesh to update (modified in place).

Definition at line 1321 of file Almog_Engine.h.

References ae_tri_set_normals(), Tri_mesh::elements, and Tri_mesh::length.

Referenced by ae_tri_mesh_rotate_Euler_xyz().

4.7.4.53 ae_tri_mesh_translate()

```
void ae_tri_mesh_translate (
            Tri_mesh mesh,
            float x,
            float y,
            float z )
```

Translate a triangle mesh by (x, y, z).

Adds the given offsets to each vertex in the mesh.

Parameters

mesh	Triangle mesh to translate (modified in place).
x	X-axis offset.
y	Y-axis offset.
z	Z-axis offset.

Definition at line 1121 of file Almog_Engine.h.

References Tri_mesh::elements, Tri_mesh::length, Tri::points, Point::x, Point::y, and Point::z.

4.7.4.54 ae_tri_project_world2screen()

```
Tri_mesh ae_tri_project_world2screen (
            Mat2D proj_mat,
            Mat2D view_mat,
            Tri tri,
            int window_w,
            int window_h,
            Scene * scene,
            Lighting_mode lighting_mode )
```

Project a single world-space triangle to screen space with clipping.

Computes lighting, back-face visibility, transforms to view space, clips against near plane, and projects to screen space. If clipping splits the triangle, multiple triangles may be returned.

Parameters

proj_mat	Projection matrix (4x4).
view_mat	View matrix (4x4).
tri	World-space triangle.
window_w	Screen width in pixels.
window_h	Screen height in pixels.
scene	Scene (camera for near plane, light/material for lighting).
lighting_mode	Flat or smooth lighting mode.

Returns

Tri_mesh An ADA array of resulting screen-space triangles. Caller must free result.elements.

Definition at line 3038 of file Almog_Engine.h.

References ada_append, ada_init_array, ae_assert_tri_is_valid, ae_point_project_view2screen(), ae_point_to_mat2D(), ae_tri_calc_light_intensity(), ae_tri_calc_normal(), ae_tri_clip_with_plane(), ae_tri_transform_to_view(), Scene::camera, Camera::current_position, Tri_mesh::elements, Tri_mesh::length, Tri::light_intensity, mat2D_alloc(), MAT2D_AT, mat2D_fill(), mat2D_free(), mat2D_sub(), mat2D_transpose(), Tri::normals, Tri::points, Tri::tex_points, Tri::to_draw, Point::w, Point::x, Point::y, Point::z, and Camera::z_near.

Referenced by ae_tri_mesh_project_world2screen().

4.7.4.55 ae_tri_qsort()

```
void ae_tri_qsort (
            Tri * v,
            int left,
            int right )
```

Quicksort an array of triangles by depth.

Sorts v[left..right] using ae_tri_compare (descending by max z).

Parameters

v	Array of triangles to sort.
left	Left index (inclusive).
right	Right index (inclusive).

Definition at line 3742 of file Almog_Engine.h.

References ae_tri_compare(), and ae_tri_swap().

4.7.4.56 ae_tri_set_normals()

```
void ae_tri_set_normals (
            Tri * tri )
```

Compute and set per-vertex normals for a triangle.

For each vertex, computes the cross product of the adjacent edges around that vertex and normalizes it. Results are stored in tri->normals[i].

Parameters

tri	Triangle whose normals will be computed and written.

Definition at line 998 of file Almog_Engine.h.

References ae_assert_tri_is_valid, ae_mat2D_to_point(), ae_point_to_mat2D(), mat2D_alloc(), mat2D_copy(), mat2D_cross(), mat2D_free(), mat2D_normalize, mat2D_sub(), Tri::normals, and Tri::points.

Referenced by ae_tri_mesh_flip_normals(), and ae_tri_mesh_set_normals().

4.7.4.57 ae_tri_swap()

```
void ae_tri_swap (
            Tri * v,
            int i,
            int j )
```

Swap two triangles in an array.

Parameters

v	Array of triangles.
i	Index of first element.
j	Index of second element.

Definition at line 3706 of file Almog_Engine.h.

Referenced by ae_tri_qsort().

4.7.4.58 ae_tri_transform_to_view()

```
Tri ae_tri_transform_to_view (
            Mat2D view_mat,
            Tri tri )
```

Transform a triangle from world space to view space.

Applies view_mat to each vertex (homogeneous multiply with w=1). Returns the transformed triangle; normals are not changed.

Parameters

view_mat	View matrix (4x4).
tri	World-space triangle.

Returns

Tri View-space triangle.

Definition at line 2988 of file Almog_Engine.h.

References AE_ASSERT, ae_assert_tri_is_valid, mat2D_alloc(), MAT2D_AT, mat2D_dot(), mat2D_free(), Tri::points, Point::w, Point::x, Point::y, and Point::z.

Referenced by ae_tri_project_world2screen().

4.7.4.59 ae_view_mat_set()

```
void ae_view_mat_set (
            Mat2D view_mat,
            Camera camera,
            Mat2D up )
```

Build a right-handed view matrix from a Camera and up vector.

Computes camera basis (right, up, forward) from yaw/pitch/roll offsets and direction, applies offset_position along those axes to update current_position, then zeroes offset_position. Writes the resulting 4x4 view matrix.

Note

Although camera is passed by value, its Mat2D members (e.g. current_position, offset_position, camera_↩ x/y/z) are modified in place due to internal pointer semantics of Mat2D.

Parameters

view_mat	Output 4x4 view matrix.
camera	Camera state (basis vectors and positions updated).
up	World up direction (3x1).

Definition at line 2716 of file Almog_Engine.h.

References Camera::camera_x, Camera::camera_y, Camera::camera_z, Camera::current_position, Camera::direction, mat2D_add(), mat2D_alloc(), MAT2D_AT, mat2D_calc_norma(), mat2D_copy(), mat2D_cross(), mat2D_dot(), mat2D_dot_product(), mat2D_fill(), mat2D_free(), mat2D_mult(), mat2D_set_DCM_zyx(), mat2D_sub(), mat2D_transpose(), Camera::offset_position, Camera::pitch_offset_deg, Camera::roll_offset_deg, and Camera::yaw_offset_deg.

Referenced by ae_scene_init(), and update().

4.7.4.60 ae_z_buffer_copy_to_screen()

```
void ae_z_buffer_copy_to_screen (
            Mat2D_uint32 screen_mat,
            Mat2D inv_z_buffer )
```

Visualize an inverse-z buffer by writing a grayscale image.

Finds the min positive and max inverse-z in inv_z_buffer, maps the range to [0.1, 1.0], and writes an RGB grayscale value into screen_mat at each pixel. Values <= 0 are clamped to the minimum positive.

Parameters

screen_mat	Output RGB image (Mat2D_uint32) 0xRRGGBB per pixel.
inv_z_buffer	Input inverse-z values (Mat2D of doubles).

Definition at line 3785 of file Almog_Engine.h.

References ae_linear_map(), Mat2D::cols, MAT2D_AT, MAT2D_AT_UINT32, RGB_hexRGB, and Mat2D::rows.

4.8 Almog_Engine.h

```
#ifndef ALMOG_ENGINE_H_
#define ALMOG_ENGINE_H_

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

#ifndef ALMOG_STRING_MANIPULATION_IMPLEMENTATION
#define ALMOG_STRING_MANIPULATION_IMPLEMENTATION
#endif
#include "./Almog_String_Manipulation.h"

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

#include <math.h>
#include <stdbool.h>
#include <float.h>
#include <stdint.h>
#include <errno.h>
#include <string.h>

#ifndef PI
#define PI M_PI
#endif

#ifndef STL_HEADER_SIZE
#define STL_HEADER_SIZE 80
#endif

#ifndef STL_NUM_SIZE
#define STL_NUM_SIZE 4
#endif

#ifndef STL_SIZE_FOREACH_TRI
#define STL_SIZE_FOREACH_TRI 50
#endif

#ifndef STL_ATTRIBUTE_BITS_SIZE
#define STL_ATTRIBUTE_BITS_SIZE 2
#endif

#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
#define ARGB_hexARGB(a, r, g, b) 0x01000000l*(uint8_t)(a) + 0x00010000*(uint8_t)(r) +
    0x00000100*(uint8_t)(g) + 0x00000001*(uint8_t)(b)
#ifndef RGB_hexRGB
#define RGB_hexRGB(r, g, b) (int)(0x010000*(int)(r) + 0x000100*(int)(g) + 0x000001*(int)(b))
#endif

#define AE_MAX_POINT_VAL 1e5
#define ae_assert_point_is_valid(p) AE_ASSERT(isfinite((p).x) && isfinite((p).y) && isfinite((p).z) &&
    isfinite((p).w));    \
        AE_ASSERT((p).x > -AE_MAX_POINT_VAL && (p).x < AE_MAX_POINT_VAL);
                                                 \
```

```
        AE_ASSERT((p).y > -AE_MAX_POINT_VAL && (p).y < AE_MAX_POINT_VAL);        \
        AE_ASSERT((p).z > -AE_MAX_POINT_VAL && (p).z < AE_MAX_POINT_VAL);        \
        AE_ASSERT((p).w > -AE_MAX_POINT_VAL && (p).w < AE_MAX_POINT_VAL);
#define ae_assert_tri_is_valid(tri) ae_assert_point_is_valid((tri).points[0]);  \
        ae_assert_point_is_valid((tri).points[1]);                              \
        ae_assert_point_is_valid((tri).points[2])
#define ae_assert_quad_is_valid(quad) ae_assert_point_is_valid((quad).points[0]);   \
        ae_assert_point_is_valid((quad).points[1]);                                 \
        ae_assert_point_is_valid((quad).points[2]);                                 \
        ae_assert_point_is_valid((quad).points[3])
#define ae_point_normalize_xyz_norma(p, norma) (p).x = (p).x / norma;  \
        (p).y = (p).y / norma;                                         \
        (p).z = (p).z / norma
#define ae_point_calc_norma(p) sqrt(((p).x * (p).x) + ((p).y * (p).y) + ((p).z * (p).z))
#define ae_point_add_point(p, p1, p2) (p).x = (p1).x + (p2).x;  \
        (p).y = (p1).y + (p2).y;                                \
        (p).z = (p1).z + (p2).z;                                \
        (p).w = (p1).w + (p2).w
#define ae_point_sub_point(p, p1, p2) (p).x = (p1).x - (p2).x;  \
        (p).y = (p1).y - (p2).y;                                \
        (p).z = (p1).z - (p2).z;                                \
        (p).w = (p1).w - (p2).w
#define ae_point_dot_point(p1, p2) (((p1).x * (p2).x) + ((p1).y * (p2).y) + ((p1).z * (p2).z))
#define ae_point_mult(p, const) (p).x *= const; \
        (p).y *= const;                         \
        (p).z *= const
#define ae_points_equal(p1, p2) (p1).x == (p2).x && (p1).y == (p2).y && (p1).z == (p2).z


typedef enum {
    AE_LIGHTING_FLAT,
    AE_LIGHTING_SMOOTH,
    AE_LIGHTING_MODE_LENGTH
} Lighting_mode;

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

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

typedef struct {
    Mat2D init_position;
    Mat2D current_position;
    Mat2D offset_position;
    Mat2D direction;
    float z_near;
    float z_far;
    float fov_deg;
    float aspect_ratio;
    float roll_offset_deg;
    float pitch_offset_deg;
    float yaw_offset_deg;
    Mat2D camera_x;
    Mat2D camera_y;
    Mat2D camera_z;
} Camera;

typedef struct {
    Point light_direction_or_pos;
    float light_intensity;
} Light_source;

typedef struct {
    float specular_power_alpha;
    float c_ambi;
    float c_diff;
    float c_spec;
} Material;

typedef struct {
    Tri_mesh_array in_world_tri_meshes;
    Tri_mesh_array projected_tri_meshes;
```

```
    Tri_mesh_array original_tri_meshes;

    Quad_mesh_array in_world_quad_meshes;
    Quad_mesh_array projected_quad_meshes;
    Quad_mesh_array original_quad_meshes;

    Camera camera;
    Mat2D up_direction;
    Mat2D proj_mat;
    Mat2D view_mat;

    Light_source light_source0;
    Material material0;
} Scene;

Tri         ae_tri_create(Point p1, Point p2, Point p3);
void        ae_tri_mesh_create_copy(Tri_mesh *des, Tri *src_elements, size_t len);

void        ae_camera_init(Scene *scene, int window_h, int window_w);
void        ae_camera_free(Scene *scene);
Scene       ae_scene_init(int window_h, int window_w);
void        ae_scene_free(Scene *scene);
void        ae_camera_reset_pos(Scene *scene);

void        ae_point_to_mat2D(Point p, Mat2D m);
Point       ae_mat2D_to_point(Mat2D m);

Tri_mesh    ae_tri_mesh_get_from_obj_file(char *file_path);
Tri_mesh    ae_tri_mesh_get_from_stl_file(char *file_path);
Tri_mesh    ae_tri_mesh_get_from_file(char *file_path);
void        ae_tri_mesh_append_copy(Tri_mesh_array *mesh_array, Tri_mesh mesh);
Tri_mesh    ae_tri_mesh_get_from_quad_mesh(Quad_mesh q_mesh);

void        ae_print_points(Curve p);
void        ae_print_tri(Tri tri, char *name, size_t padding);
void        ae_print_tri_mesh(Tri_mesh mesh, char *name, size_t padding);

Point       ae_point_normalize_xyz(Point p);
void        ae_tri_set_normals(Tri *tri);
Point       ae_tri_get_average_normal(Tri tri);
Point       ae_tri_get_average_point(Tri tri);
void        ae_tri_calc_normal(Mat2D normal, Tri tri);
void        ae_tri_mesh_translate(Tri_mesh mesh, float x, float y, float z);
void        ae_tri_mesh_rotate_Euler_xyz(Tri_mesh mesh, float phi_deg, float theta_deg, float
      psi_deg);
void        ae_tri_mesh_set_bounding_box(Tri_mesh mesh, float *x_min, float *x_max, float *y_min,
      float *y_max, float *z_min, float *z_max);
void        ae_tri_mesh_normalize(Tri_mesh mesh);
void        ae_tri_mesh_flip_normals(Tri_mesh mesh);
void        ae_tri_mesh_set_normals(Tri_mesh mesh);
void        ae_quad_set_normals(Quad *quad);
Point       ae_quad_get_average_normal(Quad quad);
Point       ae_quad_get_average_point(Quad quad);
void        ae_quad_calc_normal(Mat2D normal, Quad quad);
void        ae_curve_copy(Curve *des, Curve src);

void        ae_tri_calc_light_intensity(Tri *tri, Scene *scene, Lighting_mode lighting_mode);
void        ae_quad_calc_light_intensity(Quad *quad, Scene *scene, Lighting_mode lighting_mode);

Point       ae_line_itersect_plane(Mat2D plane_p, Mat2D plane_n, Mat2D line_start, Mat2D line_end,
      float *t);
int         ae_line_clip_with_plane(Point start_in, Point end_in, Mat2D plane_p, Mat2D plane_n, Point
      *start_out, Point *end_out);
float       ae_signed_dist_point_and_plane(Point p, Mat2D plane_p, Mat2D plane_n);
int         ae_tri_clip_with_plane(Tri tri_in, Mat2D plane_p, Mat2D plane_n, Tri *tri_out1, Tri
      *tri_out2);
int         ae_quad_clip_with_plane(Quad quad_in, Mat2D plane_p, Mat2D plane_n, Quad *quad_out1, Quad
      *quad_out2);

void        ae_projection_mat_set(Mat2D proj_mat,float aspect_ratio, float FOV_deg, float z_near,
      float z_far);
void        ae_view_mat_set(Mat2D view_mat, Camera camera, Mat2D up);
Point       ae_point_project_world2screen(Mat2D view_mat, Mat2D proj_mat, Point src, int window_w, int
      window_h);
Point       ae_point_project_world2view(Mat2D view_mat, Point src);
Point       ae_point_project_view2screen(Mat2D proj_mat, Point src, int window_w, int window_h);
void        ae_line_project_world2screen(Mat2D view_mat, Mat2D proj_mat, Point start_src, Point
      end_src, int window_w, int window_h, Point *start_des, Point *end_des, Scene *scene);
Tri         ae_tri_transform_to_view(Mat2D view_mat, Tri tri);
Tri_mesh    ae_tri_project_world2screen(Mat2D proj_mat, Mat2D view_mat, Tri tri, int window_w, int
      window_h, Scene *scene, Lighting_mode lighting_mode);
void        ae_tri_mesh_project_world2screen(Mat2D proj_mat, Mat2D view_mat, Tri_mesh *des, Tri_mesh
      src, int window_w, int window_h, Scene *scene, Lighting_mode lighting_mode);
Quad        ae_quad_transform_to_view(Mat2D view_mat, Quad quad);
Quad_mesh   ae_quad_project_world2screen(Mat2D proj_mat, Mat2D view_mat, Quad quad, int window_w, int
      window_h, Scene *scene, Lighting_mode lighting_mode);
void        ae_quad_mesh_project_world2screen(Mat2D proj_mat, Mat2D view_mat, Quad_mesh *des,
```

```
    Quad_mesh src, int window_w, int window_h, Scene *scene, Lighting_mode lighting_mode);
void        ae_curve_project_world2screen(Mat2D proj_mat, Mat2D view_mat, Curve *des, Curve src, int
    window_w, int window_h, Scene *scene);
void        ae_curve_ada_project_world2screen(Mat2D proj_mat, Mat2D view_mat, Curve_ada *des,
    Curve_ada src, int window_w, int window_h, Scene *scene);
void        ae_grid_project_world2screen(Mat2D proj_mat, Mat2D view_mat, Grid des, Grid src, int
    window_w, int window_h, Scene *scene);

void        ae_tri_swap(Tri *v, int i, int j);
bool        ae_tri_compare(Tri t1, Tri t2);
void        ae_tri_qsort(Tri *v, int left, int right);
double      ae_linear_map(double s, double min_in, double max_in, double min_out, double max_out);
void        ae_z_buffer_copy_to_screen(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer);

#endif /* ALMOG_ENGINE_H_ */

#ifdef ALMOG_ENGINE_IMPLEMENTATION
#undef ALMOG_ENGINE_IMPLEMENTATION

#define AE_PRINT_TRI(tri) ae_print_tri(tri, #tri, 0)
#define AE_PRINT_MESH(mesh) ae_print_tri_mesh(mesh, #mesh, 0)

Tri ae_tri_create(Point p1, Point p2, Point p3)
{
    Tri tri;

    tri.points[0] = p1;
    tri.points[1] = p2;
    tri.points[2] = p3;

    return tri;
}

void ae_tri_mesh_create_copy(Tri_mesh *des, Tri *src_elements, size_t len)
{
    Tri_mesh temp_des = *des;
    temp_des.length = 0;
    for (size_t i = 0; i < len; i++) {
        ada_append(Tri, temp_des, src_elements[i]);
    }
    *des = temp_des;
}

void ae_camera_init(Scene *scene, int window_h, int window_w)
{
    scene->camera.z_near       = 0.1;
    scene->camera.z_far        = 1000;
    scene->camera.fov_deg      = 60;
    scene->camera.aspect_ratio = (float)window_h / (float)window_w;

    scene->camera.init_position = mat2D_alloc(3, 1);
    mat2D_fill(scene->camera.init_position, 0);
    MAT2D_AT(scene->camera.init_position, 2, 0) = -4;

    scene->camera.current_position = mat2D_alloc(3, 1);
    mat2D_copy(scene->camera.current_position, scene->camera.init_position);

    scene->camera.offset_position = mat2D_alloc(3, 1);
    mat2D_fill(scene->camera.offset_position, 0);

    scene->camera.roll_offset_deg = 0;
    scene->camera.pitch_offset_deg = 0;
    scene->camera.yaw_offset_deg = 0;

    scene->camera.direction = mat2D_alloc(3, 1);
    mat2D_fill(scene->camera.direction, 0);
    MAT2D_AT(scene->camera.direction, 2, 0) = 1;

    scene->camera.camera_x = mat2D_alloc(3, 1);
    mat2D_fill(scene->camera.camera_x, 0);
    MAT2D_AT(scene->camera.camera_x, 0, 0) = 1;

    scene->camera.camera_y = mat2D_alloc(3, 1);
    mat2D_fill(scene->camera.camera_y, 0);
    MAT2D_AT(scene->camera.camera_y, 1, 0) = 1;

    scene->camera.camera_z = mat2D_alloc(3, 1);
    mat2D_fill(scene->camera.camera_z, 0);
    MAT2D_AT(scene->camera.camera_z, 2, 0) = 1;
}

void ae_camera_free(Scene *scene)
{
    mat2D_free(scene->camera.init_position);
    mat2D_free(scene->camera.current_position);
    mat2D_free(scene->camera.offset_position);
    mat2D_free(scene->camera.direction);
```

```
    mat2D_free(scene->camera.camera_x);
    mat2D_free(scene->camera.camera_y);
    mat2D_free(scene->camera.camera_z);
}

Scene ae_scene_init(int window_h, int window_w)
{
    Scene scene = {0};
    ae_camera_init(&(scene), window_h, window_w);

    scene.up_direction = mat2D_alloc(3, 1);
    mat2D_fill(scene.up_direction, 0);
    MAT2D_AT(scene.up_direction, 1, 0) = 1;

    scene.light_source0.light_direction_or_pos.x = 0.5;
    scene.light_source0.light_direction_or_pos.y = 1;
    scene.light_source0.light_direction_or_pos.z = 1;
    scene.light_source0.light_direction_or_pos.w = 0;
    scene.light_source0.light_direction_or_pos =
  ae_point_normalize_xyz(scene.light_source0.light_direction_or_pos);
    scene.light_source0.light_intensity = 1;

    scene.material0.specular_power_alpha = 1;
    scene.material0.c_ambi = 0.2;
    scene.material0.c_diff = 0.6;
    scene.material0.c_spec = 0.2;

    scene.proj_mat = mat2D_alloc(4, 4);
    ae_projection_mat_set(scene.proj_mat, scene.camera.aspect_ratio, scene.camera.fov_deg,
  scene.camera.z_near, scene.camera.z_far);

    scene.view_mat = mat2D_alloc(4, 4);
    ae_view_mat_set(scene.view_mat, scene.camera, scene.up_direction);

    return scene;
}

void ae_scene_free(Scene *scene)
{
    ae_camera_free(scene);
    mat2D_free(scene->up_direction);
    mat2D_free(scene->proj_mat);
    mat2D_free(scene->view_mat);

    for (size_t i = 0; i < scene->in_world_tri_meshes.length; i++) {
        free(scene->in_world_tri_meshes.elements[i].elements);
    }
    for (size_t i = 0; i < scene->projected_tri_meshes.length; i++) {
        free(scene->projected_tri_meshes.elements[i].elements);
    }
    for (size_t i = 0; i < scene->original_tri_meshes.length; i++) {
        free(scene->original_tri_meshes.elements[i].elements);
    }
    if (scene->in_world_tri_meshes.elements) free(scene->in_world_tri_meshes.elements);
    if (scene->projected_tri_meshes.elements) free(scene->projected_tri_meshes.elements);
    if (scene->original_tri_meshes.elements) free(scene->original_tri_meshes.elements);

    for (size_t i = 0; i < scene->in_world_quad_meshes.length; i++) {
        free(scene->in_world_quad_meshes.elements[i].elements);
    }
    for (size_t i = 0; i < scene->projected_quad_meshes.length; i++) {
        free(scene->projected_quad_meshes.elements[i].elements);
    }
    for (size_t i = 0; i < scene->original_quad_meshes.length; i++) {
        free(scene->original_quad_meshes.elements[i].elements);
    }
    if (scene->in_world_quad_meshes.elements) free(scene->in_world_quad_meshes.elements);
    if (scene->projected_quad_meshes.elements) free(scene->projected_quad_meshes.elements);
    if (scene->original_quad_meshes.elements) free(scene->original_quad_meshes.elements);
}

void ae_camera_reset_pos(Scene *scene)
{
    scene->camera.roll_offset_deg = 0;
    scene->camera.pitch_offset_deg = 0;
    scene->camera.yaw_offset_deg = 0;

    mat2D_fill(scene->camera.offset_position, 0);

    mat2D_fill(scene->camera.camera_x, 0);
    MAT2D_AT(scene->camera.camera_x, 0, 0) = 1;
    mat2D_fill(scene->camera.camera_y, 0);
    MAT2D_AT(scene->camera.camera_y, 1, 0) = 1;
    mat2D_fill(scene->camera.camera_z, 0);
    MAT2D_AT(scene->camera.camera_z, 2, 0) = 1;

    mat2D_copy(scene->camera.current_position, scene->camera.init_position);
```

```
}

void ae_point_to_mat2D(Point p, Mat2D m)
{
    MATRIX2D_ASSERT((3 == m.rows && 1 == m.cols) || (1 == m.rows && 3 == m.cols));

    if (3 == m.rows) {
        MAT2D_AT(m, 0, 0) = p.x;
        MAT2D_AT(m, 1, 0) = p.y;
        MAT2D_AT(m, 2, 0) = p.z;
    }
    if (3 == m.cols) {
        MAT2D_AT(m, 0, 0) = p.x;
        MAT2D_AT(m, 0, 1) = p.y;
        MAT2D_AT(m, 0, 2) = p.z;
    }
}

Point ae_mat2D_to_point(Mat2D m)
{
    Point res = {.x = MAT2D_AT(m, 0, 0), .y = MAT2D_AT(m, 1, 0), .z = MAT2D_AT(m, 2, 0), .w = 1};
    return res;
}

Tri_mesh ae_tri_mesh_get_from_obj_file(char *file_path)
{
    char current_line[ASM_MAX_LEN_LINE], current_word[ASM_MAX_LEN_LINE],
 current_num_str[ASM_MAX_LEN_LINE];
    char file_name[ASM_MAX_LEN_LINE], file_extention[ASM_MAX_LEN_LINE], mesh_name[ASM_MAX_LEN_LINE];
    int texture_warning_was_printed = 0;

    strncpy(file_name, file_path, ASM_MAX_LEN_LINE);
    strncpy(file_extention, file_name, ASM_MAX_LEN_LINE);

    /* check if file is an obj file*/
    asm_get_word_and_cut(file_name, file_extention, '.');
    asm_get_word_and_cut(file_name, file_extention, '.');
    if (strncmp(file_extention, ".obj", ASM_MAX_LEN_LINE)) {
        fprintf(stderr, "%s:%d: [Error] unsupported file format: '%s'\n", __FILE__, __LINE__,
 file_name);
        exit(1);
    }

    strncpy(mesh_name, file_name, ASM_MAX_LEN_LINE);
    while(asm_length(mesh_name)) {
        asm_get_word_and_cut(current_word, mesh_name, '/');
    }

    strncpy(mesh_name, current_word, ASM_MAX_LEN_LINE);

    strncpy(current_word, ".", ASM_MAX_LEN_LINE);
    strncat(file_name, ".obj", ASM_MAX_LEN_LINE/2);
    strncat(current_word, file_name, ASM_MAX_LEN_LINE/2);

    FILE *fp_input = fopen(current_word, "rt");
    if (fp_input == NULL) {
        fprintf(stderr, "%s:%d: [Error] failed to open input file: '%s', %s\n", __FILE__, __LINE__,
 current_word, strerror(errno));
        exit(1);
    }

    // strncpy(output_file_name, "./build/", ASM_MAX_LEN_LINE);
    // strncat(output_file_name, mesh_name, ASM_MAX_LEN_LINE/2);
    // strncat(output_file_name, ".c", ASM_MAX_LEN_LINE/2);
    // FILE *fp_output = fopen(output_file_name, "wt");
    // if (fp_input == NULL) {
    //     fprintf(stderr, "%s:%d: [Error] failed to open output file: '%s'. %s\n", __FILE__,
 __LINE__, output_file_name, strerror(errno));
    //     exit(1);
    // }

    /* parsing data from file */
    Curve points = {0};
    ada_init_array(Point, points);
    Tri_mesh mesh;
    ada_init_array(Tri, mesh);

    int line_len;

    while ((line_len = asm_get_line(fp_input, current_line)) != -1) {
        asm_get_next_word_from_line(current_word, current_line, ' ');
        if (!strncmp(current_word, "v", 1)) {
            Point p;
            asm_get_word_and_cut(current_word, current_line, ' ');
            asm_get_word_and_cut(current_word, current_line, ' ');
            p.x = atof(current_word);
            asm_get_word_and_cut(current_word, current_line, ' ');
```

```c
            p.y = atof(current_word);
            asm_get_word_and_cut(current_word, current_line, ' ');
            p.z = atof(current_word);
            // printf("current word: %s\n", current_word);
            ada_append(Point, points, p);
            // break;
        }
        if (!strncmp(current_word, "f", 1)) {
            Tri tri1 = {0}, tri2 = {0};

            // printf("line: %s\nword: %s, %d\n", current_line, current_word, atoi(current_word));
            asm_get_word_and_cut(current_word, current_line, ' ');
            // printf("line: %s\nword: %s, %d\n", current_line, current_word, atoi(current_word));

            int number_of_spaces = asm_str_in_str(current_line, " ");
            // printf("%d\n", number_of_spaces);
            // exit(1);
            if (!(number_of_spaces == 3 || number_of_spaces == 4 || number_of_spaces == 5)) {
                fprintf(stderr, "%s:%d: [Error] there is unsupported number of vertices for a face: %d\n", __FILE__, __LINE__, number_of_spaces);
                exit(1);
            }
            if (number_of_spaces == 3) {
                /* there are 3 vertices for the face. */
                asm_get_word_and_cut(current_word, current_line, ' ');
                // printf("line: %s\nword: %s, %d\n", current_line, current_word, atoi(current_word));
                int number_of_backslash = asm_str_in_str(current_word, "/");
                if (number_of_backslash == 0) {
                    tri1.points[0] = points.elements[atoi(current_word)-1];
                    asm_get_word_and_cut(current_word, current_line, ' ');
                    tri1.points[1] = points.elements[atoi(current_word)-1];
                    asm_get_word_and_cut(current_word, current_line, ' ');
                    tri1.points[2] = points.elements[atoi(current_word)-1];
                }
                if (number_of_backslash == 2) {
                    if (!texture_warning_was_printed) {
                        fprintf(stderr, "%s:%d [Warning] texture and normals data ignored of file at - '%s'\n", __FILE__, __LINE__, file_path);
                        texture_warning_was_printed = 1;
                    }

                    asm_get_word_and_cut(current_num_str, current_word, '/');
                    // printf("line: %s\nword: %s\nnum str: %s, %d\n", current_line, current_word, current_num_str, atoi(current_num_str));
                    tri1.points[0] = points.elements[atoi(current_num_str)-1];

                    asm_get_word_and_cut(current_word, current_line, ' ');
                    asm_get_word_and_cut(current_num_str, current_word, '/');
                    // printf("line: %s\nword: %s\nnum str: %s, %d\n", current_line, current_word, current_num_str, atoi(current_num_str));
                    tri1.points[1] = points.elements[atoi(current_num_str)-1];

                    asm_get_word_and_cut(current_word, current_line, ' ');
                    asm_get_word_and_cut(current_num_str, current_word, '/');
                    // printf("line: %s\nword: %s\nnum str: %s, %d\n", current_line, current_word, current_num_str, atoi(current_num_str));
                    tri1.points[2] = points.elements[atoi(current_num_str)-1];
                }

                tri1.to_draw = true;
                tri1.light_intensity[0] = 1;
                tri1.light_intensity[1] = 1;
                tri1.light_intensity[2] = 1;
                tri1.colors[0] = 0xFFFFFFFF;
                tri1.colors[1] = 0xFFFFFFFF;
                tri1.colors[2] = 0xFFFFFFFF;

                ada_append(Tri, mesh, tri1);
                // AE_PRINT_TRI(tri1);
            } else if (number_of_spaces == 5 || number_of_spaces == 4) {
                /* there are 4 vertices for the face. */
                /* sometimes there is a space in the end */
                asm_get_word_and_cut(current_word, current_line, ' ');
                // printf("line: %s\nword: %s, %d\n", current_line, current_word, atoi(current_word));
                int number_of_backslash = asm_str_in_str(current_word, "/");
                if (number_of_backslash == 0) {
                    tri1.points[0] = points.elements[atoi(current_word)-1];
                    asm_get_word_and_cut(current_word, current_line, ' ');
                    tri1.points[1] = points.elements[atoi(current_word)-1];
                    asm_get_word_and_cut(current_word, current_line, ' ');
                    tri1.points[2] = points.elements[atoi(current_word)-1];
                }
                if (number_of_backslash == 2 || number_of_backslash == 1) {
                    if (!texture_warning_was_printed) {
                        fprintf(stderr, "%s:%d [Warning] texture and normals data ignored of file at - '%s'\n", __FILE__, __LINE__, file_path);
                        texture_warning_was_printed = 1;
```

```
                    }

                    asm_get_word_and_cut(current_num_str, current_word, '/');
                    // printf("line: %s\nword: %s\nnum str: %s, %d\n", current_line, current_word, current_num_str, atoi(current_num_str));
                    tri1.points[0] = points.elements[atoi(current_num_str)-1];
                    tri2.points[2] = points.elements[atoi(current_num_str)-1];

                    asm_get_word_and_cut(current_word, current_line, ' ');
                    asm_get_word_and_cut(current_num_str, current_word, '/');
                    // printf("line: %s\nword: %s\nnum str: %s, %d\n", current_line, current_word, current_num_str, atoi(current_num_str));
                    tri1.points[1] = points.elements[atoi(current_num_str)-1];

                    asm_get_word_and_cut(current_word, current_line, ' ');
                    asm_get_word_and_cut(current_num_str, current_word, '/');
                    // printf("line: %s\nword: %s\nnum str: %s, %d\n", current_line, current_word, current_num_str, atoi(current_num_str));
                    tri1.points[2] = points.elements[atoi(current_num_str)-1];
                    tri2.points[0] = points.elements[atoi(current_num_str)-1];

                    asm_get_word_and_cut(current_word, current_line, ' ');
                    asm_get_word_and_cut(current_num_str, current_word, '/');
                    // printf("line: %s\nword: %s\nnum str: %s, %d\n", current_line, current_word, current_num_str, atoi(current_num_str));
                    tri2.points[1] = points.elements[atoi(current_num_str)-1];
                }

                tri1.to_draw = true;
                tri1.light_intensity[0] = 1;
                tri1.light_intensity[1] = 1;
                tri1.light_intensity[2] = 1;
                tri1.colors[0] = 0xFFFFFFFF;
                tri1.colors[1] = 0xFFFFFFFF;
                tri1.colors[2] = 0xFFFFFFFF;

                tri2.to_draw = true;
                tri2.light_intensity[0] = 1;
                tri2.light_intensity[1] = 1;
                tri2.light_intensity[2] = 1;
                tri2.colors[0] = 0xFFFFFFFF;
                tri2.colors[1] = 0xFFFFFFFF;
                tri2.colors[2] = 0xFFFFFFFF;

                ada_append(Tri, mesh, tri1);
                ada_append(Tri, mesh, tri2);
                // AE_PRINT_TRI(tri1);
                // AE_PRINT_TRI(tri2);
            }
            // exit(2);
        }
    }

    return mesh;
}

Tri_mesh ae_tri_mesh_get_from_stl_file(char *file_path)
{
    FILE *file;
    file = fopen(file_path, "rb");
    if (file == NULL) {
        fprintf(stderr, "%s:%d: [Error] failed to open input file: '%s', %s\n", __FILE__, __LINE__, file_path, strerror(errno));
        exit(1);
    }

    char header[STL_HEADER_SIZE];
    fread(header, STL_HEADER_SIZE, 1, file);
    // dprintSTRING(header);

    uint32_t num_of_tri;
    fread(&num_of_tri, STL_NUM_SIZE, 1, file);
    // dprintINT(num_of_tri);

    Tri_mesh mesh;
    ada_init_array(Tri, mesh);
    for (size_t i = 0; i < num_of_tri; i++) {
        Tri temp_tri = {0};

        fread(&(temp_tri.normals[0].x), STL_NUM_SIZE, 1, file);
        fread(&(temp_tri.normals[0].y), STL_NUM_SIZE, 1, file);
        fread(&(temp_tri.normals[0].z), STL_NUM_SIZE, 1, file);

        temp_tri.normals[0].x = - temp_tri.normals[0].x;
        temp_tri.normals[0].y = - temp_tri.normals[0].y;
        temp_tri.normals[0].z = - temp_tri.normals[0].z;

```

```
        temp_tri.normals[1] = temp_tri.normals[0];
        temp_tri.normals[2] = temp_tri.normals[0];

        fread(&(temp_tri.points[0].x), STL_NUM_SIZE, 1, file);
        fread(&(temp_tri.points[0].y), STL_NUM_SIZE, 1, file);
        fread(&(temp_tri.points[0].z), STL_NUM_SIZE, 1, file);

        fread(&(temp_tri.points[1].x), STL_NUM_SIZE, 1, file);
        fread(&(temp_tri.points[1].y), STL_NUM_SIZE, 1, file);
        fread(&(temp_tri.points[1].z), STL_NUM_SIZE, 1, file);

        fread(&(temp_tri.points[2].x), STL_NUM_SIZE, 1, file);
        fread(&(temp_tri.points[2].y), STL_NUM_SIZE, 1, file);
        fread(&(temp_tri.points[2].z), STL_NUM_SIZE, 1, file);

        fseek(file, STL_ATTRIBUTE_BITS_SIZE, SEEK_CUR);

        temp_tri.to_draw = true;
        temp_tri.light_intensity[0] = 1;
        temp_tri.light_intensity[1] = 1;
        temp_tri.light_intensity[2] = 1;
        temp_tri.colors[0] = 0xFFFFFFFF;
        temp_tri.colors[1] = 0xFFFFFFFF;
        temp_tri.colors[2] = 0xFFFFFFFF;

        // ae_tri_set_normals(&temp_tri);

        ada_append(Tri, mesh, temp_tri);
    }

    return mesh;
}

Tri_mesh ae_tri_mesh_get_from_file(char *file_path)
{
    char file_extention[ASM_MAX_LEN_LINE], temp_word[ASM_MAX_LEN_LINE];

    strncpy(file_extention, file_path, ASM_MAX_LEN_LINE);

    int num_of_dots;
    while ((num_of_dots = asm_str_in_str(file_extention, ".")) >= 1) {
        asm_get_word_and_cut(temp_word, file_extention, '.');
    }

    if (!(!strncmp(file_extention, "obj", 3) || !strncmp(file_extention, "STL", 3) || !strncmp(file_extention, "stl", 3))) {
        fprintf(stderr, "%s:%d: [Error] unsupported file format: '%s'\n", __FILE__, __LINE__, file_path);
        exit(1);
    }

    if (!strncmp(file_extention, "STL", 3) || !strncmp(file_extention, "stl", 3)) {
        return ae_tri_mesh_get_from_stl_file(file_path);
    } else if (!strncmp(file_extention, "obj", 3)) {
        return ae_tri_mesh_get_from_obj_file(file_path);
    }

    Tri_mesh null_mesh = {0};
    return null_mesh;
}

void ae_tri_mesh_append_copy(Tri_mesh_array *mesh_array, Tri_mesh mesh)
{
    Tri_mesh_array temp_mesh_array = *mesh_array;
    Tri_mesh temp_mesh;
    ada_init_array(Tri, temp_mesh);
    for (size_t i = 0; i < mesh.length; i++) {
        ada_append(Tri, temp_mesh, mesh.elements[i]);
    }
    ada_append(Tri_mesh, temp_mesh_array, temp_mesh);


    *mesh_array = temp_mesh_array;
}

Tri_mesh ae_tri_mesh_get_from_quad_mesh(Quad_mesh q_mesh)
{
    Tri_mesh t_mesh;
    ada_init_array(Tri, t_mesh);

    for (size_t q_index = 0; q_index < q_mesh.length; q_index++) {
        Quad current_q = q_mesh.elements[q_index];
        Tri temp_t = {.to_draw = current_q.to_draw};

        temp_t.points[0] = current_q.points[0];
        temp_t.colors[0] = current_q.colors[0];
        temp_t.normals[0] = current_q.normals[0];
```

```
        temp_t.light_intensity[0] = current_q.light_intensity[0];
        temp_t.points[1] = current_q.points[1];
        temp_t.colors[1] = current_q.colors[1];
        temp_t.normals[1] = current_q.normals[1];
        temp_t.light_intensity[1] = current_q.light_intensity[1];
        temp_t.points[2] = current_q.points[2];
        temp_t.colors[2] = current_q.colors[2];
        temp_t.normals[2] = current_q.normals[2];
        temp_t.light_intensity[2] = current_q.light_intensity[2];

        ada_append(Tri, t_mesh, temp_t);

        temp_t.points[0] = current_q.points[2];
        temp_t.colors[0] = current_q.colors[2];
        temp_t.normals[0] = current_q.normals[2];
        temp_t.light_intensity[0] = current_q.light_intensity[2];
        temp_t.points[1] = current_q.points[3];
        temp_t.colors[1] = current_q.colors[3];
        temp_t.normals[1] = current_q.normals[3];
        temp_t.light_intensity[1] = current_q.light_intensity[3];
        temp_t.points[2] = current_q.points[0];
        temp_t.colors[2] = current_q.colors[0];
        temp_t.normals[2] = current_q.normals[0];
        temp_t.light_intensity[2] = current_q.light_intensity[0];

        ada_append(Tri, t_mesh, temp_t);
    }

    return t_mesh;
}

void ae_print_points(Curve p)
{
    for (size_t i = 0; i < p.length; i++) {
        printf("point %3zu: (%5f, %5f, %5f)\n", i, p.elements[i].x, p.elements[i].y, p.elements[i].z);
    }
}

void ae_print_tri(Tri tri, char *name, size_t padding)
{
    printf("%*s%s:\n", (int) padding, "", name);
    printf("%*s    (%f, %f, %f)\n%*s    (%f, %f, %f)\n%*s    (%f, %f, %f)\n", (int) padding, "", tri.points[0].x, tri.points[0].y, tri.points[0].z, (int) padding, "", tri.points[1].x, tri.points[1].y, tri.points[1].z, (int) padding, "", tri.points[2].x, tri.points[2].y, tri.points[2].z);
    printf("%*s    draw? %d\n", (int)padding, "", tri.to_draw);
}

void ae_print_tri_mesh(Tri_mesh mesh, char *name, size_t padding)
{
    char tri_name[256];
    printf("%*s%s:\n", (int) padding, "", name);
    for (size_t i = 0; i < mesh.length; i++) {
        snprintf(tri_name, 256, "tri %zu", i);
        ae_print_tri(mesh.elements[i], tri_name, 4);
    }
}

Point ae_point_normalize_xyz(Point p)
{
    Point res = {0};

    float norma = ae_point_calc_norma(p);

    res.x = p.x / norma;
    res.y = p.y / norma;
    res.z = p.z / norma;
    res.w = p.w;

    return res;
}

void ae_tri_set_normals(Tri *tri)
{
    ae_assert_tri_is_valid(*tri);

    Mat2D point  = mat2D_alloc(3, 1);
    Mat2D to_p   = mat2D_alloc(3, 1);
    Mat2D from_p = mat2D_alloc(3, 1);
    Mat2D normal = mat2D_alloc(3, 1);

    for (int i = 0; i < 3; i++) {
        int current_index  = i;
        int next_index     = (i + 1) % 3;
        int previous_index = (i - 1 + 3) % 3;
        ae_point_to_mat2D(tri->points[current_index], point);
        ae_point_to_mat2D(tri->points[next_index], from_p);
```

```
        ae_point_to_mat2D(tri->points[previous_index], to_p);

        mat2D_sub(from_p, point);
        mat2D_sub(point, to_p);

        mat2D_copy(to_p, point);

        mat2D_cross(normal, to_p, from_p);
        // mat2D_cross(normal, from_p, to_p);
        mat2D_normalize(normal);

        tri->normals[current_index] = ae_mat2D_to_point(normal);
    }

    mat2D_free(point);
    mat2D_free(to_p);
    mat2D_free(from_p);
    mat2D_free(normal);
}

Point ae_tri_get_average_normal(Tri tri)
{
    Point normal0 = tri.normals[0];
    Point normal1 = tri.normals[1];
    Point normal2 = tri.normals[2];

    Point res;
    res.x = (normal0.x + normal1.x + normal2.x) / 3;
    res.y = (normal0.y + normal1.y + normal2.y) / 3;
    res.z = (normal0.z + normal1.z + normal2.z) / 3;
    res.w = (normal0.w + normal1.w + normal2.w) / 3;

    return ae_point_normalize_xyz(res);
}

Point ae_tri_get_average_point(Tri tri)
{
    Point point0 = tri.points[0];
    Point point1 = tri.points[1];
    Point point2 = tri.points[2];

    Point res;
    res.x = (point0.x + point1.x + point2.x) / 3;
    res.y = (point0.y + point1.y + point2.y) / 3;
    res.z = (point0.z + point1.z + point2.z) / 3;
    res.w = (point0.w + point1.w + point2.w) / 3;

    return res;
}

void ae_tri_calc_normal(Mat2D normal, Tri tri)
{
    AE_ASSERT(3 == normal.rows && 1 == normal.cols);
    ae_assert_tri_is_valid(tri);

    Mat2D a = mat2D_alloc(3, 1);
    Mat2D b = mat2D_alloc(3, 1);
    Mat2D c = mat2D_alloc(3, 1);

    ae_point_to_mat2D(tri.points[0], a);
    ae_point_to_mat2D(tri.points[1], b);
    ae_point_to_mat2D(tri.points[2], c);

    mat2D_sub(b, a);
    mat2D_sub(c, a);

    mat2D_cross(normal, b, c);

    mat2D_mult(normal, 1/mat2D_calc_norma(normal));

    mat2D_free(a);
    mat2D_free(b);
    mat2D_free(c);
}

void ae_tri_mesh_translate(Tri_mesh mesh, float x, float y, float z)
{
    for (size_t i = 0; i < mesh.length; i++) {
        for (int j = 0; j < 3; j++) {
            mesh.elements[i].points[j].x += x;
            mesh.elements[i].points[j].y += y;
            mesh.elements[i].points[j].z += z;
        }
    }
}

void ae_tri_mesh_rotate_Euler_xyz(Tri_mesh mesh, float phi_deg, float theta_deg, float psi_deg)
```

```
{
    Mat2D RotZ = mat2D_alloc(3,3);
    mat2D_set_rot_mat_z(RotZ, psi_deg);
    Mat2D RotY = mat2D_alloc(3,3);
    mat2D_set_rot_mat_y(RotY, theta_deg);
    Mat2D RotX = mat2D_alloc(3,3);
    mat2D_set_rot_mat_x(RotX, phi_deg);
    Mat2D DCM = mat2D_alloc(3,3);
    // mat2D_fill(DCM,0);
    Mat2D temp = mat2D_alloc(3,3);
    // mat2D_fill(temp,0);
    mat2D_dot(temp, RotY, RotZ);
    mat2D_dot(DCM, RotX, temp); /* I have a DCM */

    Mat2D src_point_mat = mat2D_alloc(3,1);
    Mat2D des_point_mat = mat2D_alloc(3,1);

    for (size_t i = 0; i < mesh.length; i++) {
        for (int j = 0; j < 3; j++) {
            // mat2D_fill(src_point_mat, 0);
            // mat2D_fill(des_point_mat, 0);
            Point des;
            Point src = mesh.elements[i].points[j];

            MAT2D_AT(src_point_mat, 0, 0) = src.x;
            MAT2D_AT(src_point_mat, 1, 0) = src.y;
            MAT2D_AT(src_point_mat, 2, 0) = src.z;

            mat2D_dot(des_point_mat, DCM, src_point_mat);

            des.x = MAT2D_AT(des_point_mat, 0, 0);
            des.y = MAT2D_AT(des_point_mat, 1, 0);
            des.z = MAT2D_AT(des_point_mat, 2, 0);

            mesh.elements[i].points[j] = des;
        }
    }

    ae_tri_mesh_set_normals(mesh);

    mat2D_free(RotZ);
    mat2D_free(RotY);
    mat2D_free(RotX);
    mat2D_free(DCM);
    mat2D_free(temp);
    mat2D_free(src_point_mat);
    mat2D_free(des_point_mat);
}

void ae_tri_mesh_set_bounding_box(Tri_mesh mesh, float *x_min, float *x_max, float *y_min, float
  *y_max, float *z_min, float *z_max)
{
    float xmin = FLT_MAX, xmax = FLT_MIN;
    float ymin = FLT_MAX, ymax = FLT_MIN;
    float zmin = FLT_MAX, zmax = FLT_MIN;

    float x, y, z;

    for (size_t t = 0; t < mesh.length; t++) {
        for (size_t p = 0; p < 3; p++) {
            x = mesh.elements[t].points[p].x;
            y = mesh.elements[t].points[p].y;
            z = mesh.elements[t].points[p].z;
            if (x > xmax) xmax = x;
            if (x < xmin) xmin = x;
            if (y > ymax) ymax = y;
            if (y < ymin) ymin = y;
            if (z > zmax) zmax = z;
            if (z < zmin) zmin = z;
        }
    }
    *x_min = xmin;
    *x_max = xmax;
    *y_min = ymin;
    *y_max = ymax;
    *z_min = zmin;
    *z_max = zmax;
}

void ae_tri_mesh_normalize(Tri_mesh mesh)
{
    float xmax, xmin, ymax, ymin, zmax, zmin;
    ae_tri_mesh_set_bounding_box(mesh, &xmin, &xmax, &ymin, &ymax, &zmin, &zmax);

    for (size_t t = 0; t < mesh.length; t++) {
        for (size_t p = 0; p < 3; p++) {
            float x, y, z;
```

```
            x = mesh.elements[t].points[p].x;
            y = mesh.elements[t].points[p].y;
            z = mesh.elements[t].points[p].z;

            float xdiff = xmax-xmin;
            float ydiff = ymax-ymin;
            float zdiff = zmax-zmin;
            float max_diff = fmax(xdiff, fmax(ydiff, zdiff));
            float xfactor = xdiff/max_diff;
            float yfactor = ydiff/max_diff;
            float zfactor = zdiff/max_diff;

            x = (((x - xmin) / (xdiff)) * 2 - 1) * xfactor;
            y = (((y - ymin) / (ydiff)) * 2 - 1) * yfactor;
            z = (((z - zmin) / (zdiff)) * 2 - 1) * zfactor;

            mesh.elements[t].points[p].x = x;
            mesh.elements[t].points[p].y = y;
            mesh.elements[t].points[p].z = z;
        }
    }
}

void ae_tri_mesh_flip_normals(Tri_mesh mesh)
{
    for (size_t i = 0; i < mesh.length; i++) {
        Tri res_tri, tri = mesh.elements[i];

        res_tri.to_draw = tri.to_draw;

        res_tri.colors[0]          = tri.colors[2];
        res_tri.light_intensity[0] = tri.light_intensity[2];
        res_tri.normals[0]         = tri.normals[2];
        res_tri.points[0]          = tri.points[2];
        res_tri.tex_points[0]      = tri.tex_points[2];

        res_tri.colors[1]          = tri.colors[1];
        res_tri.light_intensity[1] = tri.light_intensity[1];
        res_tri.normals[1]         = tri.normals[1];
        res_tri.points[1]          = tri.points[1];
        res_tri.tex_points[1]      = tri.tex_points[1];

        res_tri.colors[2]          = tri.colors[0];
        res_tri.light_intensity[2] = tri.light_intensity[0];
        res_tri.normals[2]         = tri.normals[0];
        res_tri.points[2]          = tri.points[0];
        res_tri.tex_points[2]      = tri.tex_points[0];

        ae_tri_set_normals(&res_tri);

        mesh.elements[i] = res_tri;
    }
}

void ae_tri_mesh_set_normals(Tri_mesh mesh)
{
    for (size_t i = 0; i < mesh.length; i++) {
        ae_tri_set_normals(&(mesh.elements[i]));
    }
}

void ae_quad_set_normals(Quad *quad)
{
    ae_assert_quad_is_valid(*quad);

    Mat2D point  = mat2D_alloc(3, 1);
    Mat2D to_p   = mat2D_alloc(3, 1);
    Mat2D from_p = mat2D_alloc(3, 1);
    Mat2D normal = mat2D_alloc(3, 1);

    for (int i = 0; i < 4; i++) {
        int current_index  = i;
        int next_index     = (i + 1) % 4;
        int previous_index = (i - 1 + 4) % 4;
        ae_point_to_mat2D(quad->points[current_index], point);
        ae_point_to_mat2D(quad->points[next_index], from_p);
        ae_point_to_mat2D(quad->points[previous_index], to_p);

        mat2D_sub(from_p, point);
        mat2D_sub(point, to_p);

        mat2D_copy(to_p, point);

        mat2D_cross(normal, to_p, from_p);
        mat2D_normalize(normal);

        quad->normals[current_index] = ae_mat2D_to_point(normal);
```

```
    }

    mat2D_free(point);
    mat2D_free(to_p);
    mat2D_free(from_p);
    mat2D_free(normal);

}

Point ae_quad_get_average_normal(Quad quad)
{
    Point normal0 = quad.normals[0];
    Point normal1 = quad.normals[1];
    Point normal2 = quad.normals[2];
    Point normal3 = quad.normals[3];

    Point res;
    res.x = (normal0.x + normal1.x + normal2.x + normal3.x) / 4;
    res.y = (normal0.y + normal1.y + normal2.y + normal3.y) / 4;
    res.z = (normal0.z + normal1.z + normal2.z + normal3.z) / 4;
    res.w = (normal0.w + normal1.w + normal2.w + normal3.w) / 4;

    res = ae_point_normalize_xyz(res);

    return res;
}

Point ae_quad_get_average_point(Quad quad)
{
    Point point0 = quad.points[0];
    Point point1 = quad.points[1];
    Point point2 = quad.points[2];
    Point point3 = quad.points[3];

    Point res;
    res.x = (point0.x + point1.x + point2.x + point3.x) / 4;
    res.y = (point0.y + point1.y + point2.y + point3.y) / 4;
    res.z = (point0.z + point1.z + point2.z + point3.z) / 4;
    res.w = (point0.w + point1.w + point2.w + point3.w) / 4;

    return res;
}

void ae_quad_calc_normal(Mat2D normal, Quad quad)
{
    AE_ASSERT(3 == normal.rows && 1 == normal.cols);
    ae_assert_quad_is_valid(quad);

    Mat2D a = mat2D_alloc(3, 1);
    Mat2D b = mat2D_alloc(3, 1);
    Mat2D c = mat2D_alloc(3, 1);

    ae_point_to_mat2D(quad.points[0], a);
    ae_point_to_mat2D(quad.points[1], b);
    ae_point_to_mat2D(quad.points[2], c);

    mat2D_sub(b, a);
    mat2D_sub(c, a);

    mat2D_cross(normal, b, c);

    mat2D_mult(normal, 1/mat2D_calc_norma(normal));

    mat2D_free(a);
    mat2D_free(b);
    mat2D_free(c);
}

void ae_curve_copy(Curve *des, Curve src)
{
    Curve temp_des = *des;
    temp_des.length = 0;

    for (size_t i = 0; i < src.length; i++) {
        ada_append(Point, temp_des, src.elements[i]);
    }

    *des = temp_des;
}

void ae_tri_calc_light_intensity(Tri *tri, Scene *scene, Lighting_mode lighting_mode)
{
    /* based on the lighting model described in: 'Alexandru C. Telea-Data Visualization_ Principles
   and Practice-A K Peters_CRC Press (2014)' Pg.29 */
    Point L = {0};
    Point r = {0};
    Point v = {0};
```

```
    Point mL = {0};
    Point pml = {0};
    Point mLn2n = {0};
    Point ave_norm = ae_tri_get_average_normal(*tri);
    Point camera_pos = ae_mat2D_to_point(scene->camera.current_position);

    float c_ambi = scene->material0.c_ambi;
    float c_diff = scene->material0.c_diff;
    float c_spec = scene->material0.c_spec;
    float alpha  = scene->material0.specular_power_alpha;

    switch (lighting_mode) {
    case AE_LIGHTING_FLAT:
        for (int i = 0; i < 3; i++) {
            if (scene->light_source0.light_direction_or_pos.w == 0) {
                L = scene->light_source0.light_direction_or_pos;
                L = ae_point_normalize_xyz(L);
                mL = L;
                ae_point_mult(mL, -1);
            } else {
                Point l = scene->light_source0.light_direction_or_pos;
                Point p = tri->points[i];
                ae_point_sub_point(pml, p, l);
                pml = ae_point_normalize_xyz(pml);
                L = pml;
                L.w = 0;
                mL = L;
                ae_point_mult(mL, -1);
            }

            ae_point_sub_point(v, camera_pos, ae_tri_get_average_point(*tri));
            float mL_dot_norm = ae_point_dot_point(mL, ave_norm);
            mLn2n = ave_norm;
            ae_point_mult(mLn2n, 2 * mL_dot_norm);
            ae_point_add_point(r, L, mLn2n);

            tri->light_intensity[i] = c_ambi + scene->light_source0.light_intensity * (c_diff *
      fmaxf(mL_dot_norm, 0) + c_spec * powf(fmaxf(ae_point_dot_point(r, v), 0), alpha));
        }
        break;
    case AE_LIGHTING_SMOOTH:
        for (int i = 0; i < 3; i++) {
            if (scene->light_source0.light_direction_or_pos.w == 0) {
                L = scene->light_source0.light_direction_or_pos;
                L = ae_point_normalize_xyz(L);
                mL = L;
                ae_point_mult(mL, -1);
            } else {
                Point l = scene->light_source0.light_direction_or_pos;
                Point p = tri->points[i];
                ae_point_sub_point(pml, p, l);
                pml = ae_point_normalize_xyz(pml);
                L = pml;
                L.w = 0;
                mL = L;
                ae_point_mult(mL, -1);
            }
            ae_point_sub_point(v, camera_pos, tri->points[i]);
            float mL_dot_norm = ae_point_dot_point(mL, tri->normals[i]);
            mLn2n = tri->normals[i];
            ae_point_mult(mLn2n, 2 * mL_dot_norm);
            ae_point_add_point(r, L, mLn2n);

            tri->light_intensity[i] = c_ambi + scene->light_source0.light_intensity * (c_diff *
      fmaxf(mL_dot_norm, 0) + c_spec * powf(fmaxf(ae_point_dot_point(r, v), 0), alpha));
        }
        break;
    default:
        for (int i = 0; i < 3; i++) {
            tri->light_intensity[i] = 1;
        }
        break;
    }

    for (int i = 0; i < 3; i++) {
        tri->light_intensity[i] = fminf(1, fmaxf(0, tri->light_intensity[i]));
    }
}

void ae_quad_calc_light_intensity(Quad *quad, Scene *scene, Lighting_mode lighting_mode)
{
    /* based on the lighting model described in: 'Alexandru C. Telea-Data Visualization_ Principles
      and Practice-A K Peters_CRC Press (2014)' Pg.29 */
    Point L = {0};
    Point r = {0};
    Point v = {0};
    Point mL = {0};
```

```
    Point pml = {0};
    Point mLn2n = {0};
    Point ave_norm = ae_quad_get_average_normal(*quad);
    Point camera_pos = ae_mat2D_to_point(scene->camera.current_position);

    float c_ambi = scene->material0.c_ambi;
    float c_diff = scene->material0.c_diff;
    float c_spec = scene->material0.c_spec;
    float alpha  = scene->material0.specular_power_alpha;

    switch (lighting_mode) {
    case AE_LIGHTING_FLAT:
        for (int i = 0; i < 4; i++) {
            if (scene->light_source0.light_direction_or_pos.w == 0) {
                L = scene->light_source0.light_direction_or_pos;
                L = ae_point_normalize_xyz(L);
                mL = L;
                ae_point_mult(mL, -1);
            } else {
                Point l = scene->light_source0.light_direction_or_pos;
                Point p = quad->points[i];
                ae_point_sub_point(pml, p, l);
                pml = ae_point_normalize_xyz(pml);
                L = pml;
                L.w = 0;
                mL = L;
                ae_point_mult(mL, -1);
            }

            ae_point_sub_point(v, camera_pos, ae_quad_get_average_point(*quad));
            float mL_dot_norm = ae_point_dot_point(mL, ave_norm);
            mLn2n = ave_norm;
            ae_point_mult(mLn2n, 2 * mL_dot_norm);
            ae_point_add_point(r, L, mLn2n);

            quad->light_intensity[i] = c_ambi + scene->light_source0.light_intensity * (c_diff * fmaxf(mL_dot_norm, 0) + c_spec * powf(fmaxf(ae_point_dot_point(r, v), 0), alpha));
        }
        break;
    case AE_LIGHTING_SMOOTH:
        for (int i = 0; i < 4; i++) {
            if (scene->light_source0.light_direction_or_pos.w == 0) {
                L = scene->light_source0.light_direction_or_pos;
                L = ae_point_normalize_xyz(L);
                mL = L;
                ae_point_mult(mL, -1);
            } else {
                Point l = scene->light_source0.light_direction_or_pos;
                Point p = quad->points[i];
                ae_point_sub_point(pml, p, l);
                pml = ae_point_normalize_xyz(pml);
                L = pml;
                L.w = 0;
                mL = L;
                ae_point_mult(mL, -1);
            }
            ae_point_sub_point(v, camera_pos, quad->points[i]);
            float mL_dot_norm = ae_point_dot_point(mL, quad->normals[i]);
            mLn2n = quad->normals[i];
            ae_point_mult(mLn2n, 2 * mL_dot_norm);
            ae_point_add_point(r, L, mLn2n);

            quad->light_intensity[i] = c_ambi + scene->light_source0.light_intensity * (c_diff * fmaxf(mL_dot_norm, 0) + c_spec * powf(fmaxf(ae_point_dot_point(r, v), 0), alpha));
        }
        break;
    default:
        for (int i = 0; i < 4; i++) {
            quad->light_intensity[i] = 1;
        }
        break;
    }

    for (int i = 0; i < 4; i++) {
        quad->light_intensity[i] = fminf(1, fmaxf(0, quad->light_intensity[i]));
    }
}

Point ae_line_itersect_plane(Mat2D plane_p, Mat2D plane_n, Mat2D line_start, Mat2D line_end, float *t)
{
    mat2D_normalize(plane_n);
    float plane_d = - mat2D_dot_product(plane_n, plane_p);
    float ad = mat2D_dot_product(line_start, plane_n);
    float bd = mat2D_dot_product(line_end, plane_n);
    *t  = (- plane_d - ad) / (bd - ad);
    mat2D_sub(line_end, line_start);
    Mat2D line_start_to_end = line_end;
```

```
    mat2D_mult(line_start_to_end, *t);
    Mat2D line_to_intersection = line_start_to_end;

    Mat2D intersection_p = mat2D_alloc(3, 1);
    mat2D_fill(intersection_p, 0);
    mat2D_add(intersection_p, line_start);
    mat2D_add(intersection_p, line_to_intersection);

    Point ans_p = ae_mat2D_to_point(intersection_p);

    mat2D_free(intersection_p);

    return ans_p;
}

int ae_line_clip_with_plane(Point start_in, Point end_in, Mat2D plane_p, Mat2D plane_n, Point
  *start_out, Point *end_out)
{
    ae_assert_point_is_valid(start_in);
    ae_assert_point_is_valid(end_in);

    mat2D_normalize(plane_n);

    /* if the signed distance is positive, the point lies on the "inside" of the plane */
    Point inside_points[2];
    Point outside_points[2];
    int inside_points_count = 0;
    int outside_points_count = 0;

    /* calc signed distance of each point of tri_in */
    float d0 = ae_signed_dist_point_and_plane(start_in, plane_p, plane_n);
    float d1 = ae_signed_dist_point_and_plane(end_in, plane_p, plane_n);
    float t;

    // float epsilon = 1e-3;
    float epsilon = 0;
    if (d0 >= epsilon) {
        inside_points[inside_points_count++] = start_in;
    } else {
        outside_points[outside_points_count++] = start_in;
    }
    if (d1 >= epsilon) {
        inside_points[inside_points_count++] = end_in;
    } else {
        outside_points[outside_points_count++] = end_in;
    }

    /* classifying the triangle points */
    if (outside_points_count == 2) {
        return 0;
    } else if (inside_points_count == 2) {
        *start_out = start_in;
        *end_out = end_in;
        return 1;
    } else if (d0 >= epsilon && d1 < epsilon) {
        Mat2D line_start = mat2D_alloc(3, 1);
        Mat2D line_end   = mat2D_alloc(3, 1);

        *start_out = inside_points[0];

        ae_point_to_mat2D(inside_points[0], line_start);
        ae_point_to_mat2D(outside_points[0], line_end);
        *end_out = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);

        mat2D_free(line_start);
        mat2D_free(line_end);

        ae_assert_point_is_valid(*start_out);
        ae_assert_point_is_valid(*end_out);

        return 1;
    } else if (d1 >= epsilon && d0 < epsilon) {
        Mat2D line_start = mat2D_alloc(3, 1);
        Mat2D line_end   = mat2D_alloc(3, 1);

        *end_out = inside_points[0];

        ae_point_to_mat2D(inside_points[0], line_start);
        ae_point_to_mat2D(outside_points[0], line_end);
        *start_out = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);

        mat2D_free(line_start);
        mat2D_free(line_end);

        ae_assert_point_is_valid(*start_out);
        ae_assert_point_is_valid(*end_out);

```

```
        return 1;
    }
    return -1;
}

float ae_signed_dist_point_and_plane(Point p, Mat2D plane_p, Mat2D plane_n)
{
    ae_assert_point_is_valid(p);

    // mat2D_normalize(plane_n);
    // Mat2D p_mat2D = mat2D_alloc(3, 1);
    // ae_point_to_mat2D(p, p_mat2D);

    // float res = mat2D_dot_product(plane_n, p_mat2D) - mat2D_dot_product(plane_n, plane_p);

    float res = MAT2D_AT(plane_n, 0, 0) * p.x + MAT2D_AT(plane_n, 1, 0) * p.y + MAT2D_AT(plane_n, 2, 0) * p.z - (MAT2D_AT(plane_n, 0, 0) * MAT2D_AT(plane_p, 0, 0) + MAT2D_AT(plane_n, 1, 0) * MAT2D_AT(plane_p, 1, 0) + MAT2D_AT(plane_n, 2, 0) * MAT2D_AT(plane_p, 2, 0));

    // mat2D_free(p_mat2D);

    return res;
}

int ae_tri_clip_with_plane(Tri tri_in, Mat2D plane_p, Mat2D plane_n, Tri *tri_out1, Tri *tri_out2)
{
    ae_assert_tri_is_valid(tri_in);

    mat2D_normalize(plane_n);

    /* if the signed distance is positive, the point lies on the "inside" of the plane */
    Point inside_points[3];
    Point outside_points[3];
    int inside_points_count = 0;
    int outside_points_count = 0;
    Point tex_inside_points[3];
    Point tex_outside_points[3];
    int tex_inside_points_count = 0;
    int tex_outside_points_count = 0;

    /* calc signed distance of each point of tri_in */
    float d0 = ae_signed_dist_point_and_plane(tri_in.points[0], plane_p, plane_n);
    float d1 = ae_signed_dist_point_and_plane(tri_in.points[1], plane_p, plane_n);
    float d2 = ae_signed_dist_point_and_plane(tri_in.points[2], plane_p, plane_n);
    float t;

    // float epsilon = 1e-3;
    float epsilon = 0;
    if (d0 >= epsilon) {
        inside_points[inside_points_count++] = tri_in.points[0];
        tex_inside_points[tex_inside_points_count++] = tri_in.tex_points[0];
    } else {
        outside_points[outside_points_count++] = tri_in.points[0];
        tex_outside_points[tex_outside_points_count++] = tri_in.tex_points[0];
    }
    if (d1 >= epsilon) {
        inside_points[inside_points_count++] = tri_in.points[1];
        tex_inside_points[tex_inside_points_count++] = tri_in.tex_points[1];
    } else {
        outside_points[outside_points_count++] = tri_in.points[1];
        tex_outside_points[tex_outside_points_count++] = tri_in.tex_points[1];
    }
    if (d2 >= epsilon) {
        inside_points[inside_points_count++] = tri_in.points[2];
        tex_inside_points[tex_inside_points_count++] = tri_in.tex_points[2];
    } else {
        outside_points[outside_points_count++] = tri_in.points[2];
        tex_outside_points[tex_outside_points_count++] = tri_in.tex_points[2];
    }

    /* classifying the triangle points */
    if (inside_points_count == 0) {
        return 0;
    } else if (inside_points_count == 3) {
        *tri_out1 = tri_in;
        return 1;
    } else if (inside_points_count == 1 && outside_points_count == 2 && d2 >= epsilon) {
        Mat2D line_start = mat2D_alloc(3, 1);
        Mat2D line_end   = mat2D_alloc(3, 1);

        *tri_out1 = tri_in;
        // tri_out1->colors[0] = 0xFF0000;
        // tri_out1->colors[1] = 0xFF0000;
        // tri_out1->colors[2] = 0xFF0000;

        (*tri_out1).points[0] = inside_points[0];
        (*tri_out1).tex_points[0] = tex_inside_points[0];
```

```

        ae_point_to_mat2D(inside_points[0], line_start);
        ae_point_to_mat2D(outside_points[0], line_end);
        (*tri_out1).points[1] = ae_line_intersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*tri_out1).points[1].w = t * (outside_points[0].w - inside_points[0].w) + inside_points[0].w;
        (*tri_out1).tex_points[1].x = t * (tex_outside_points[0].x - tex_inside_points[0].x) + tex_inside_points[0].x;
        (*tri_out1).tex_points[1].y = t * (tex_outside_points[0].y - tex_inside_points[0].y) + tex_inside_points[0].y;

        ae_point_to_mat2D(inside_points[0], line_start);
        ae_point_to_mat2D(outside_points[1], line_end);
        (*tri_out1).points[2] = ae_line_intersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*tri_out1).points[2].w = t * (outside_points[1].w - inside_points[0].w) + inside_points[0].w;
        (*tri_out1).tex_points[2].x = t * (tex_outside_points[1].x - tex_inside_points[0].x) + tex_inside_points[0].x;
        (*tri_out1).tex_points[2].y = t * (tex_outside_points[1].y - tex_inside_points[0].y) + tex_inside_points[0].y;

        mat2D_free(line_start);
        mat2D_free(line_end);

        /* fixing color ordering */
        uint32_t temp_color = tri_out1->colors[2];
        tri_out1->colors[2] = tri_out1->colors[1];
        tri_out1->colors[1] = tri_out1->colors[0];
        tri_out1->colors[0] = temp_color;

        ae_assert_tri_is_valid(*tri_out1);

        return 1;
    } else if (inside_points_count == 1 && outside_points_count == 2 && d1 >= epsilon) {
        Mat2D line_start = mat2D_alloc(3, 1);
        Mat2D line_end   = mat2D_alloc(3, 1);

        *tri_out1 = tri_in;
        // tri_out1->colors[0] = 0xFF0000;
        // tri_out1->colors[1] = 0xFF0000;
        // tri_out1->colors[2] = 0xFF0000;

        (*tri_out1).points[0] = inside_points[0];
        (*tri_out1).tex_points[0] = tex_inside_points[0];

        ae_point_to_mat2D(inside_points[0], line_start);
        ae_point_to_mat2D(outside_points[0], line_end);
        (*tri_out1).points[1] = ae_line_intersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*tri_out1).points[1].w = t * (outside_points[0].w - inside_points[0].w) + inside_points[0].w;
        (*tri_out1).tex_points[1].x = t * (tex_outside_points[0].x - tex_inside_points[0].x) + tex_inside_points[0].x;
        (*tri_out1).tex_points[1].y = t * (tex_outside_points[0].y - tex_inside_points[0].y) + tex_inside_points[0].y;

        ae_point_to_mat2D(inside_points[0], line_start);
        ae_point_to_mat2D(outside_points[1], line_end);
        (*tri_out1).points[2] = ae_line_intersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*tri_out1).points[2].w = t * (outside_points[1].w - inside_points[0].w) + inside_points[0].w;
        (*tri_out1).tex_points[2].x = t * (tex_outside_points[1].x - tex_inside_points[0].x) + tex_inside_points[0].x;
        (*tri_out1).tex_points[2].y = t * (tex_outside_points[1].y - tex_inside_points[0].y) + tex_inside_points[0].y;

        mat2D_free(line_start);
        mat2D_free(line_end);

        /* fixing color ordering */
        uint32_t temp_color = tri_out1->colors[2];
        tri_out1->colors[2] = tri_out1->colors[1];
        tri_out1->colors[1] = tri_out1->colors[0];
        tri_out1->colors[0] = temp_color;

        temp_color = tri_out1->colors[2];
        tri_out1->colors[2] = tri_out1->colors[0];
        tri_out1->colors[0] = temp_color;

        ae_assert_tri_is_valid(*tri_out1);

        return 1;
    } else if (inside_points_count == 1 && outside_points_count == 2 && d0 >= epsilon) {
        Mat2D line_start = mat2D_alloc(3, 1);
        Mat2D line_end   = mat2D_alloc(3, 1);

        *tri_out1 = tri_in;
        // tri_out1->colors[0] = 0xFF0000;
        // tri_out1->colors[1] = 0xFF0000;
        // tri_out1->colors[2] = 0xFF0000;

        (*tri_out1).points[0] = inside_points[0];
```

```
        (*tri_out1).tex_points[0] = tex_inside_points[0];

        ae_point_to_mat2D(inside_points[0], line_start);
        ae_point_to_mat2D(outside_points[0], line_end);
        (*tri_out1).points[1] = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*tri_out1).points[1].w = t * (outside_points[0].w - inside_points[0].w) + inside_points[0].w;
        (*tri_out1).tex_points[1].x = t * (tex_outside_points[0].x - tex_inside_points[0].x) +
tex_inside_points[0].x;
        (*tri_out1).tex_points[1].y = t * (tex_outside_points[0].y - tex_inside_points[0].y) +
tex_inside_points[0].y;

        ae_point_to_mat2D(inside_points[0], line_start);
        ae_point_to_mat2D(outside_points[1], line_end);
        (*tri_out1).points[2] = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*tri_out1).points[2].w = t * (outside_points[1].w - inside_points[0].w) + inside_points[0].w;
        (*tri_out1).tex_points[2].x = t * (tex_outside_points[1].x - tex_inside_points[0].x) +
tex_inside_points[0].x;
        (*tri_out1).tex_points[2].y = t * (tex_outside_points[1].y - tex_inside_points[0].y) +
tex_inside_points[0].y;

        mat2D_free(line_start);
        mat2D_free(line_end);

        ae_assert_tri_is_valid(*tri_out1);

        return 1;
    } else if (inside_points_count == 2 && outside_points_count == 1 && d2 < epsilon) {
        Mat2D line_start = mat2D_alloc(3, 1);
        Mat2D line_end   = mat2D_alloc(3, 1);

        *tri_out1 = tri_in;
        // tri_out1->colors[0] = 0x00FF00;
        // tri_out1->colors[1] = 0x00FF00;
        // tri_out1->colors[2] = 0x00FF00;

        *tri_out2 = tri_in;
        // tri_out2->colors[0] = 0x0000FF;
        // tri_out2->colors[1] = 0x0000FF;
        // tri_out2->colors[2] = 0x0000FF;

        (*tri_out1).points[0] = inside_points[0];
        (*tri_out1).tex_points[0] = tex_inside_points[0];
        (*tri_out1).points[1] = inside_points[1];
        (*tri_out1).tex_points[1] = tex_inside_points[1];
        ae_point_to_mat2D(inside_points[0], line_start);
        ae_point_to_mat2D(outside_points[0], line_end);
        (*tri_out1).points[2] = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*tri_out1).points[2].w = t * (outside_points[0].w - inside_points[0].w) + inside_points[0].w;
        (*tri_out1).tex_points[2].x = t * (tex_outside_points[0].x - tex_inside_points[0].x) +
tex_inside_points[0].x;
        (*tri_out1).tex_points[2].y = t * (tex_outside_points[0].y - tex_inside_points[0].y) +
tex_inside_points[0].y;

        (*tri_out2).points[0] = inside_points[1];
        (*tri_out2).tex_points[0] = tex_inside_points[1];
        ae_point_to_mat2D(inside_points[1], line_start);
        ae_point_to_mat2D(outside_points[0], line_end);
        (*tri_out2).points[1] = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*tri_out2).points[1].w = t * (outside_points[0].w - inside_points[1].w) + inside_points[1].w;
        (*tri_out2).tex_points[1].x = t * (tex_outside_points[0].x - tex_inside_points[1].x) +
tex_inside_points[1].x;
        (*tri_out2).tex_points[1].y = t * (tex_outside_points[0].y - tex_inside_points[1].y) +
tex_inside_points[1].y;
        (*tri_out2).points[2] = (*tri_out1).points[2];
        (*tri_out2).tex_points[2] = (*tri_out1).tex_points[2];

        mat2D_free(line_start);
        mat2D_free(line_end);

        /* fixing color ordering */
        uint32_t temp_color = tri_out2->colors[2];
        tri_out2->colors[2] = tri_out2->colors[0];
        tri_out2->colors[0] = tri_out2->colors[1];
        tri_out2->colors[1] = temp_color;

        ae_assert_tri_is_valid(*tri_out1);
        ae_assert_tri_is_valid(*tri_out2);

        return 2;
    } else if (inside_points_count == 2 && outside_points_count == 1 && d1 < epsilon) {
        Mat2D line_start = mat2D_alloc(3, 1);
        Mat2D line_end   = mat2D_alloc(3, 1);

        *tri_out1 = tri_in;
        // tri_out1->colors[0] = 0x00FF00;
        // tri_out1->colors[1] = 0x00FF00;
        // tri_out1->colors[2] = 0x00FF00;
```

```

        *tri_out2 = tri_in;
        // tri_out2->colors[0] = 0x0000FF;
        // tri_out2->colors[1] = 0x0000FF;
        // tri_out2->colors[2] = 0x0000FF;

        (*tri_out1).points[0] = inside_points[0];
        (*tri_out1).tex_points[0] = tex_inside_points[0];
        (*tri_out1).points[1] = inside_points[1];
        (*tri_out1).tex_points[1] = tex_inside_points[1];
        ae_point_to_mat2D(inside_points[0], line_start);
        ae_point_to_mat2D(outside_points[0], line_end);
        (*tri_out1).points[2] = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*tri_out1).points[2].w = t * (outside_points[0].w - inside_points[0].w) + inside_points[0].w;
        (*tri_out1).tex_points[2].x = t * (tex_outside_points[0].x - tex_inside_points[0].x) + tex_inside_points[0].x;
        (*tri_out1).tex_points[2].y = t * (tex_outside_points[0].y - tex_inside_points[0].y) + tex_inside_points[0].y;

        (*tri_out2).points[0] = inside_points[1];
        (*tri_out2).tex_points[0] = tex_inside_points[1];
        ae_point_to_mat2D(inside_points[1], line_start);
        ae_point_to_mat2D(outside_points[0], line_end);
        (*tri_out2).points[1] = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*tri_out2).points[1].w = t * (outside_points[0].w - inside_points[1].w) + inside_points[1].w;
        (*tri_out2).tex_points[1].x = t * (tex_outside_points[0].x - tex_inside_points[1].x) + tex_inside_points[1].x;
        (*tri_out2).tex_points[1].y = t * (tex_outside_points[0].y - tex_inside_points[1].y) + tex_inside_points[1].y;
        (*tri_out2).points[2] = (*tri_out1).points[2];
        (*tri_out2).tex_points[2] = (*tri_out1).tex_points[2];

        mat2D_free(line_start);
        mat2D_free(line_end);

        /* fixing color ordering */
        uint32_t temp_color = tri_out1->colors[2];
        tri_out1->colors[2] = tri_out1->colors[1];
        tri_out1->colors[1] = temp_color;

        temp_color = tri_out2->colors[2];
        tri_out2->colors[2] = tri_out2->colors[0];
        tri_out2->colors[0] = tri_out2->colors[1];
        tri_out2->colors[1] = temp_color;
        temp_color = tri_out2->colors[1];
        tri_out2->colors[1] = tri_out2->colors[0];
        tri_out2->colors[0] = temp_color;

        ae_assert_tri_is_valid(*tri_out1);
        ae_assert_tri_is_valid(*tri_out2);

        return 2;
    } else if (inside_points_count == 2 && outside_points_count == 1 && d0 < epsilon) {
        Mat2D line_start = mat2D_alloc(3, 1);
        Mat2D line_end   = mat2D_alloc(3, 1);

        *tri_out1 = tri_in;
        // tri_out1->colors[0] = 0x00FF00;
        // tri_out1->colors[1] = 0x00FF00;
        // tri_out1->colors[2] = 0x00FF00;

        *tri_out2 = tri_in;
        // tri_out2->colors[0] = 0x0000FF;
        // tri_out2->colors[1] = 0x0000FF;
        // tri_out2->colors[2] = 0x0000FF;

        (*tri_out1).points[0] = inside_points[0];
        (*tri_out1).tex_points[0] = tex_inside_points[0];
        (*tri_out1).points[1] = inside_points[1];
        (*tri_out1).tex_points[1] = tex_inside_points[1];
        ae_point_to_mat2D(inside_points[0], line_start);
        ae_point_to_mat2D(outside_points[0], line_end);
        (*tri_out1).points[2] = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*tri_out1).points[2].w = t * (outside_points[0].w - inside_points[0].w) + inside_points[0].w;
        (*tri_out1).tex_points[2].x = t * (tex_outside_points[0].x - tex_inside_points[0].x) + tex_inside_points[0].x;
        (*tri_out1).tex_points[2].y = t * (tex_outside_points[0].y - tex_inside_points[0].y) + tex_inside_points[0].y;

        (*tri_out2).points[0] = inside_points[1];
        (*tri_out2).tex_points[0] = tex_inside_points[1];
        ae_point_to_mat2D(inside_points[1], line_start);
        ae_point_to_mat2D(outside_points[0], line_end);
        (*tri_out2).points[1] = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*tri_out2).points[1].w = t * (outside_points[0].w - inside_points[1].w) + inside_points[1].w;
        (*tri_out2).tex_points[1].x = t * (tex_outside_points[0].x - tex_inside_points[1].x) + tex_inside_points[1].x;
```

```
        (*tri_out2).tex_points[1].y = t * (tex_outside_points[0].y - tex_inside_points[1].y) +
    tex_inside_points[1].y;
        (*tri_out2).points[2] = (*tri_out1).points[2];
        (*tri_out2).tex_points[2] = (*tri_out1).tex_points[2];

        mat2D_free(line_start);
        mat2D_free(line_end);

        /* fixing color ordering */
        uint32_t temp_color = tri_out1->colors[2];
        tri_out1->colors[2] = tri_out1->colors[0];
        tri_out1->colors[0] = tri_out1->colors[1];
        tri_out1->colors[1] = temp_color;

        temp_color = tri_out2->colors[2];
        tri_out2->colors[2] = tri_out2->colors[1];
        tri_out2->colors[1] = tri_out2->colors[0];
        tri_out2->colors[0] = temp_color;

        ae_assert_tri_is_valid(*tri_out1);
        ae_assert_tri_is_valid(*tri_out2);

        return 2;
    }
    return -1;
}

int ae_quad_clip_with_plane(Quad quad_in, Mat2D plane_p, Mat2D plane_n, Quad *quad_out1, Quad
    *quad_out2)
{
    ae_assert_quad_is_valid(quad_in);

    mat2D_normalize(plane_n);

    /* if the signed distance is positive, the point lies on the "inside" of the plane */
    Point inside_points[4];
    Point outside_points[4];
    int inside_points_count = 0;
    int outside_points_count = 0;

    /* calc signed distance of each point of tri_in */
    float d0 = ae_signed_dist_point_and_plane(quad_in.points[0], plane_p, plane_n);
    float d1 = ae_signed_dist_point_and_plane(quad_in.points[1], plane_p, plane_n);
    float d2 = ae_signed_dist_point_and_plane(quad_in.points[2], plane_p, plane_n);
    float d3 = ae_signed_dist_point_and_plane(quad_in.points[3], plane_p, plane_n);
    float t;

    // float epsilon = 1e-3;
    float epsilon = 0;
    if (d0 >= epsilon) {
        inside_points[inside_points_count++] = quad_in.points[0];
    } else {
        outside_points[outside_points_count++] = quad_in.points[0];
    }
    if (d1 >= epsilon) {
        inside_points[inside_points_count++] = quad_in.points[1];
    } else {
        outside_points[outside_points_count++] = quad_in.points[1];
    }
    if (d2 >= epsilon) {
        inside_points[inside_points_count++] = quad_in.points[2];
    } else {
        outside_points[outside_points_count++] = quad_in.points[2];
    }
    if (d3 >= epsilon) {
        inside_points[inside_points_count++] = quad_in.points[3];
    } else {
        outside_points[outside_points_count++] = quad_in.points[3];
    }

    /* classifying the triangle points */
    if (inside_points_count == 0) {
        return 0;
    } else if (inside_points_count == 4) {
        *quad_out1 = quad_in;
        return 1;
    } else if (inside_points_count == 1 && outside_points_count == 3 && d1 >= epsilon) {
        Mat2D line_start = mat2D_alloc(3, 1);
        Mat2D line_end   = mat2D_alloc(3, 1);

        *quad_out1 = quad_in;
        *quad_out2 = quad_in;

        (*quad_out1).points[1] = quad_in.points[1];

        ae_point_to_mat2D(quad_in.points[1], line_start);
        ae_point_to_mat2D(quad_in.points[2], line_end);
```

```
        (*quad_out1).points[2] = ae_line_intersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out1).points[2].w = t * (quad_in.points[2].w - quad_in.points[1].w) + quad_in.points[1].w;
        (*quad_out1).colors[2] = quad_in.colors[2];

        ae_point_to_mat2D(quad_in.points[1], line_start);
        ae_point_to_mat2D(quad_in.points[0], line_end);
        (*quad_out1).points[0] = ae_line_intersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out1).points[0].w = t * (quad_in.points[0].w - quad_in.points[1].w) + quad_in.points[1].w;
        (*quad_out1).colors[0] = quad_in.colors[0];

        (*quad_out1).points[3].x = ((*quad_out1).points[0].x + (*quad_out1).points[2].x) / 2;
        (*quad_out1).points[3].y = ((*quad_out1).points[0].y + (*quad_out1).points[2].y) / 2;
        (*quad_out1).points[3].z = ((*quad_out1).points[0].z + (*quad_out1).points[2].z) / 2;
        (*quad_out1).points[3].w = ((*quad_out1).points[0].w + (*quad_out1).points[2].w) / 2;
        (*quad_out1).colors[3] = quad_in.colors[3];

        mat2D_free(line_start);
        mat2D_free(line_end);

        ae_assert_quad_is_valid(*quad_out1);

        return 1;
    } else if (inside_points_count == 1 && outside_points_count == 3 && d2 >= epsilon) {
        Mat2D line_start = mat2D_alloc(3, 1);
        Mat2D line_end   = mat2D_alloc(3, 1);

        *quad_out1 = quad_in;
        *quad_out2 = quad_in;

        (*quad_out1).points[2] = quad_in.points[2];

        ae_point_to_mat2D(quad_in.points[2], line_start);
        ae_point_to_mat2D(quad_in.points[3], line_end);
        (*quad_out1).points[3] = ae_line_intersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out1).points[3].w = t * (quad_in.points[3].w - quad_in.points[2].w) + quad_in.points[2].w;
        (*quad_out1).colors[3] = quad_in.colors[3];

        ae_point_to_mat2D(quad_in.points[2], line_start);
        ae_point_to_mat2D(quad_in.points[1], line_end);
        (*quad_out1).points[1] = ae_line_intersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out1).points[1].w = t * (quad_in.points[1].w - quad_in.points[2].w) + quad_in.points[2].w;
        (*quad_out1).colors[1] = quad_in.colors[1];

        (*quad_out1).points[0].x = ((*quad_out1).points[3].x + (*quad_out1).points[1].x) / 2;
        (*quad_out1).points[0].y = ((*quad_out1).points[3].y + (*quad_out1).points[1].y) / 2;
        (*quad_out1).points[0].z = ((*quad_out1).points[3].z + (*quad_out1).points[1].z) / 2;
        (*quad_out1).points[0].w = ((*quad_out1).points[3].w + (*quad_out1).points[1].w) / 2;
        (*quad_out1).colors[0] = quad_in.colors[0];

        mat2D_free(line_start);
        mat2D_free(line_end);

        ae_assert_quad_is_valid(*quad_out1);

        return 1;
    } else if (inside_points_count == 1 && outside_points_count == 3 && d3 >= epsilon) {
        Mat2D line_start = mat2D_alloc(3, 1);
        Mat2D line_end   = mat2D_alloc(3, 1);

        *quad_out1 = quad_in;
        *quad_out2 = quad_in;

        (*quad_out1).points[3] = quad_in.points[3];

        ae_point_to_mat2D(quad_in.points[3], line_start);
        ae_point_to_mat2D(quad_in.points[0], line_end);
        (*quad_out1).points[0] = ae_line_intersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out1).points[0].w = t * (quad_in.points[0].w - quad_in.points[3].w) + quad_in.points[3].w;
        (*quad_out1).colors[0] = quad_in.colors[0];

        ae_point_to_mat2D(quad_in.points[3], line_start);
        ae_point_to_mat2D(quad_in.points[2], line_end);
        (*quad_out1).points[2] = ae_line_intersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out1).points[2].w = t * (quad_in.points[2].w - quad_in.points[3].w) + quad_in.points[3].w;
        (*quad_out1).colors[2] = quad_in.colors[2];

        (*quad_out1).points[1].x = ((*quad_out1).points[2].x + (*quad_out1).points[0].x) / 2;
        (*quad_out1).points[1].y = ((*quad_out1).points[2].y + (*quad_out1).points[0].y) / 2;
        (*quad_out1).points[1].z = ((*quad_out1).points[2].z + (*quad_out1).points[0].z) / 2;
        (*quad_out1).points[1].w = ((*quad_out1).points[2].w + (*quad_out1).points[0].w) / 2;
        (*quad_out1).colors[1] = quad_in.colors[1];
```

```

        mat2D_free(line_start);
        mat2D_free(line_end);

        ae_assert_quad_is_valid(*quad_out1);

        return 1;
    } else if (inside_points_count == 1 && outside_points_count == 3) {
        Mat2D line_start = mat2D_alloc(3, 1);
        Mat2D line_end   = mat2D_alloc(3, 1);

        *quad_out1 = quad_in;
        *quad_out2 = quad_in;

        (*quad_out1).points[0] = inside_points[0];

        ae_point_to_mat2D(inside_points[0], line_start);
        ae_point_to_mat2D(outside_points[0], line_end);
        (*quad_out1).points[1] = ae_line_intersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out1).points[1].w = t * (outside_points[0].w - inside_points[0].w) + inside_points[0].w;

        ae_point_to_mat2D(inside_points[0], line_start);
        ae_point_to_mat2D(outside_points[1], line_end);
        (*quad_out1).points[2] = ae_line_intersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out1).points[2].w = t * (outside_points[1].w - inside_points[0].w) + inside_points[0].w;

        ae_point_to_mat2D(inside_points[0], line_start);
        ae_point_to_mat2D(outside_points[2], line_end);
        (*quad_out1).points[3] = ae_line_intersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out1).points[3].w = t * (outside_points[2].w - inside_points[0].w) + inside_points[0].w;

        mat2D_free(line_start);
        mat2D_free(line_end);

        ae_assert_quad_is_valid(*quad_out1);

        return 1;
    } else if (inside_points_count == 2 && outside_points_count == 2 && d2 < epsilon && d1 < epsilon) {
        Mat2D line_start = mat2D_alloc(3, 1);
        Mat2D line_end   = mat2D_alloc(3, 1);

        *quad_out1 = quad_in;

        (*quad_out1).points[0] = quad_in.points[3];
        (*quad_out1).colors[0] = quad_in.colors[3];
        (*quad_out1).points[1] = quad_in.points[0];
        (*quad_out1).colors[1] = quad_in.colors[0];

        ae_point_to_mat2D(quad_in.points[0], line_start);
        ae_point_to_mat2D(quad_in.points[1], line_end);
        (*quad_out1).points[2] = ae_line_intersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out1).points[2].w = t * (quad_in.points[1].w - quad_in.points[0].w) + quad_in.points[0].w;
        (*quad_out1).colors[2] = quad_in.colors[1];

        ae_point_to_mat2D(quad_in.points[3], line_start);
        ae_point_to_mat2D(quad_in.points[2], line_end);
        (*quad_out1).points[3] = ae_line_intersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out1).points[3].w = t * (quad_in.points[2].w - quad_in.points[3].w) + quad_in.points[3].w;
        (*quad_out1).colors[3] = quad_in.colors[2];

        mat2D_free(line_start);
        mat2D_free(line_end);

        ae_assert_quad_is_valid(*quad_out1);

        return 2;
    } else if (inside_points_count == 2 && outside_points_count == 2 && d0 < epsilon && d1 < epsilon) {
        Mat2D line_start = mat2D_alloc(3, 1);
        Mat2D line_end   = mat2D_alloc(3, 1);

        *quad_out1 = quad_in;

        (*quad_out1).points[0] = quad_in.points[2];
        (*quad_out1).colors[0] = quad_in.colors[2];
        (*quad_out1).points[1] = quad_in.points[3];
        (*quad_out1).colors[1] = quad_in.colors[3];

        ae_point_to_mat2D(quad_in.points[2], line_start);
        ae_point_to_mat2D(quad_in.points[1], line_end);
        (*quad_out1).points[3] = ae_line_intersect_plane(plane_p, plane_n, line_start, line_end, &t);
```

```
        (*quad_out1).points[3].w = t * (quad_in.points[1].w - quad_in.points[2].w) + quad_in.points[2].w;
        (*quad_out1).colors[3] = quad_in.colors[1];

        ae_point_to_mat2D(quad_in.points[3], line_start);
        ae_point_to_mat2D(quad_in.points[0], line_end);
        (*quad_out1).points[2] = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out1).points[2].w = t * (quad_in.points[0].w - quad_in.points[3].w) + quad_in.points[3].w;
        (*quad_out1).colors[2] = quad_in.colors[0];

        mat2D_free(line_start);
        mat2D_free(line_end);

        ae_assert_quad_is_valid(*quad_out1);

        return 2;
    } else if (inside_points_count == 2 && outside_points_count == 2 && d0 < epsilon && d3 < epsilon) {
        Mat2D line_start = mat2D_alloc(3, 1);
        Mat2D line_end   = mat2D_alloc(3, 1);

        *quad_out1 = quad_in;

        (*quad_out1).points[0] = quad_in.points[1];
        (*quad_out1).colors[0] = quad_in.colors[1];
        (*quad_out1).points[1] = quad_in.points[2];
        (*quad_out1).colors[1] = quad_in.colors[2];

        ae_point_to_mat2D(quad_in.points[2], line_start);
        ae_point_to_mat2D(quad_in.points[3], line_end);
        (*quad_out1).points[2] = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out1).points[2].w = t * (quad_in.points[3].w - quad_in.points[2].w) + quad_in.points[2].w;
        (*quad_out1).colors[2] = quad_in.colors[3];

        ae_point_to_mat2D(quad_in.points[1], line_start);
        ae_point_to_mat2D(quad_in.points[0], line_end);
        (*quad_out1).points[3] = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out1).points[3].w = t * (quad_in.points[1].w - quad_in.points[3].w) + quad_in.points[3].w;
        (*quad_out1).colors[3] = quad_in.colors[0];

        mat2D_free(line_start);
        mat2D_free(line_end);

        ae_assert_quad_is_valid(*quad_out1);

        return 1;
    } else if (inside_points_count == 2 && outside_points_count == 2) {
        Mat2D line_start = mat2D_alloc(3, 1);
        Mat2D line_end   = mat2D_alloc(3, 1);

        *quad_out1 = quad_in;

        (*quad_out1).points[0] = inside_points[0];
        (*quad_out1).points[1] = inside_points[1];
        ae_point_to_mat2D(inside_points[1], line_start);
        ae_point_to_mat2D(outside_points[0], line_end);
        (*quad_out1).points[2] = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out1).points[2].w = t * (outside_points[0].w - inside_points[1].w) + inside_points[1].w;

        ae_point_to_mat2D(inside_points[0], line_start);
        ae_point_to_mat2D(outside_points[1], line_end);
        (*quad_out1).points[3] = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out1).points[3].w = t * (outside_points[1].w - inside_points[0].w) + inside_points[0].w;

        mat2D_free(line_start);
        mat2D_free(line_end);

        ae_assert_quad_is_valid(*quad_out1);

        return 1;
    } else if (inside_points_count == 3 && outside_points_count == 1 && d0 < epsilon) {
        Mat2D line_start = mat2D_alloc(3, 1);
        Mat2D line_end   = mat2D_alloc(3, 1);

        *quad_out1 = quad_in;
        *quad_out2 = quad_in;

        ae_point_to_mat2D(quad_in.points[3], line_start);
        ae_point_to_mat2D(quad_in.points[0], line_end);
        (*quad_out1).points[0] = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out1).points[0].w = t * (quad_in.points[0].w - quad_in.points[3].w) + quad_in.points[3].w;
```

```
        // adl_interpolate_ARGBcolor_on_RGB((quad_in).colors[3], (quad_in).colors[0], t,
&((*quad_out1).colors[0]));

        (*quad_out2).points[1] = quad_in.points[1];
        // (*quad_out2).colors[1] = quad_in.colors[1];

        ae_point_to_mat2D(quad_in.points[1], line_start);
        ae_point_to_mat2D(quad_in.points[0], line_end);
        (*quad_out2).points[0] = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out2).points[0].w = t * (quad_in.points[0].w - quad_in.points[1].w) +
quad_in.points[1].w;
        // adl_interpolate_ARGBcolor_on_RGB((quad_in).colors[1], (quad_in).colors[0], t,
&((*quad_out2).colors[0]));

        (*quad_out2).points[2] = (*quad_out1).points[0];
        // (*quad_out2).colors[2] = (*quad_out1).colors[0];

        (*quad_out2).points[3].x = ((*quad_out2).points[2].x + (*quad_out2).points[0].x) / 2;
        (*quad_out2).points[3].y = ((*quad_out2).points[2].y + (*quad_out2).points[0].y) / 2;
        (*quad_out2).points[3].z = ((*quad_out2).points[2].z + (*quad_out2).points[0].z) / 2;
        (*quad_out2).points[3].w = ((*quad_out2).points[2].w + (*quad_out2).points[0].w) / 2;
        // adl_interpolate_ARGBcolor_on_RGB((*quad_out2).colors[2], (*quad_out2).colors[0], 0.5f,
&((*quad_out2).colors[3]));


        mat2D_free(line_start);
        mat2D_free(line_end);

        ae_assert_quad_is_valid(*quad_out1);

        return 2;
    } else if (inside_points_count == 3 && outside_points_count == 1 && d1 < epsilon) {
        Mat2D line_start = mat2D_alloc(3, 1);
        Mat2D line_end   = mat2D_alloc(3, 1);

        *quad_out1 = quad_in;
        *quad_out2 = quad_in;

        (*quad_out1).points[0] = quad_in.points[0];
        (*quad_out1).colors[0] = quad_in.colors[0];
        (*quad_out1).points[2] = quad_in.points[2];
        (*quad_out1).colors[2] = quad_in.colors[2];
        (*quad_out1).points[3] = quad_in.points[3];
        (*quad_out1).colors[3] = quad_in.colors[3];

        ae_point_to_mat2D(quad_in.points[2], line_start);
        ae_point_to_mat2D(quad_in.points[1], line_end);
        (*quad_out1).points[1] = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out1).points[1].w = t * (quad_in.points[1].w - quad_in.points[2].w) +
quad_in.points[2].w;
        (*quad_out1).colors[1] = (*quad_out1).colors[1];
        // adl_interpolate_ARGBcolor_on_RGB((quad_in).colors[2], (quad_in).colors[1], t,
&((*quad_out1).colors[1]));

        (*quad_out2).points[0] = quad_in.points[0];
        (*quad_out2).colors[0] = quad_in.colors[0];
        (*quad_out2).points[3] = (*quad_out1).points[1];
        (*quad_out2).colors[3] = (*quad_out1).colors[3];

        ae_point_to_mat2D(quad_in.points[0], line_start);
        ae_point_to_mat2D(quad_in.points[1], line_end);
        (*quad_out2).points[1] = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out2).points[1].w = t * (quad_in.points[1].w - quad_in.points[0].w) +
quad_in.points[0].w;
        (*quad_out2).colors[1] = quad_in.colors[1];
        // adl_interpolate_ARGBcolor_on_RGB((quad_in).colors[0], (quad_in).colors[1], t,
&((*quad_out2).colors[1]));

        (*quad_out2).points[2].x = ((*quad_out2).points[1].x + (*quad_out2).points[3].x) / 2;
        (*quad_out2).points[2].y = ((*quad_out2).points[1].y + (*quad_out2).points[3].y) / 2;
        (*quad_out2).points[2].z = ((*quad_out2).points[1].z + (*quad_out2).points[3].z) / 2;
        (*quad_out2).points[2].w = ((*quad_out2).points[1].w + (*quad_out2).points[3].w) / 2;
        // adl_interpolate_ARGBcolor_on_RGB((*quad_out2).colors[1], (*quad_out2).colors[3], 0.5f,
&((*quad_out2).colors[2]));


        mat2D_free(line_start);
        mat2D_free(line_end);

        ae_assert_quad_is_valid(*quad_out1);

        return 2;
    } else if (inside_points_count == 3 && outside_points_count == 1 && d2 < epsilon) {
        Mat2D line_start = mat2D_alloc(3, 1);
        Mat2D line_end   = mat2D_alloc(3, 1);

        *quad_out1 = quad_in;
```

```
        *quad_out2 = quad_in;

        (*quad_out1).points[0] = quad_in.points[0];
        (*quad_out1).colors[0] = quad_in.colors[0];
        (*quad_out1).points[1] = quad_in.points[1];
        (*quad_out1).colors[1] = quad_in.colors[1];
        (*quad_out1).points[3] = quad_in.points[3];
        (*quad_out1).colors[3] = quad_in.colors[3];

        ae_point_to_mat2D(quad_in.points[1], line_start);
        ae_point_to_mat2D(quad_in.points[2], line_end);
        (*quad_out1).points[2] = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out1).points[2].w = t * (quad_in.points[2].w - quad_in.points[1].w) + quad_in.points[1].w;
        (*quad_out1).colors[2] = (*quad_out1).colors[2];
        // adl_interpolate_ARGBcolor_on_RGB((quad_in).colors[2], (quad_in).colors[1], t, &((*quad_out1).colors[1]));

        (*quad_out2).points[3] = quad_in.points[3];
        (*quad_out2).colors[3] = quad_in.colors[3];
        (*quad_out2).points[0] = (*quad_out1).points[2];
        (*quad_out2).colors[0] = (*quad_out1).colors[0];

        ae_point_to_mat2D(quad_in.points[3], line_start);
        ae_point_to_mat2D(quad_in.points[2], line_end);
        (*quad_out2).points[2] = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out2).points[2].w = t * (quad_in.points[2].w - quad_in.points[3].w) + quad_in.points[3].w;
        (*quad_out2).colors[2] = quad_in.colors[2];
        // adl_interpolate_ARGBcolor_on_RGB((quad_in).colors[0], (quad_in).colors[1], t, &((*quad_out2).colors[1]));

        (*quad_out2).points[1].x = ((*quad_out2).points[2].x + (*quad_out2).points[0].x) / 2;
        (*quad_out2).points[1].y = ((*quad_out2).points[2].y + (*quad_out2).points[0].y) / 2;
        (*quad_out2).points[1].z = ((*quad_out2).points[2].z + (*quad_out2).points[0].z) / 2;
        (*quad_out2).points[1].w = ((*quad_out2).points[2].w + (*quad_out2).points[0].w) / 2;
        // adl_interpolate_ARGBcolor_on_RGB((*quad_out2).colors[1], (*quad_out2).colors[3], 0.5f, &((*quad_out2).colors[2]));


        mat2D_free(line_start);
        mat2D_free(line_end);

        ae_assert_quad_is_valid(*quad_out1);

        return 2;
    } else if (inside_points_count == 3 && outside_points_count == 1 && d3 < epsilon) {
        Mat2D line_start = mat2D_alloc(3, 1);
        Mat2D line_end   = mat2D_alloc(3, 1);

        *quad_out1 = quad_in;
        *quad_out2 = quad_in;

        (*quad_out1).points[0] = quad_in.points[0];
        (*quad_out1).colors[0] = quad_in.colors[0];
        (*quad_out1).points[1] = quad_in.points[1];
        (*quad_out1).colors[1] = quad_in.colors[1];
        (*quad_out1).points[2] = quad_in.points[2];
        (*quad_out1).colors[2] = quad_in.colors[2];

        ae_point_to_mat2D(quad_in.points[0], line_start);
        ae_point_to_mat2D(quad_in.points[3], line_end);
        (*quad_out1).points[3] = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out1).points[3].w = t * (quad_in.points[3].w - quad_in.points[0].w) + quad_in.points[0].w;
        (*quad_out1).colors[3] = (*quad_out1).colors[3];
        // adl_interpolate_ARGBcolor_on_RGB((quad_in).colors[2], (quad_in).colors[1], t, &((*quad_out1).colors[1]));

        (*quad_out2).points[2] = quad_in.points[2];
        (*quad_out2).colors[2] = quad_in.colors[2];
        (*quad_out2).points[1] = (*quad_out1).points[3];
        (*quad_out2).colors[1] = (*quad_out1).colors[1];

        ae_point_to_mat2D(quad_in.points[2], line_start);
        ae_point_to_mat2D(quad_in.points[3], line_end);
        (*quad_out2).points[3] = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out2).points[3].w = t * (quad_in.points[3].w - quad_in.points[2].w) + quad_in.points[2].w;
        (*quad_out2).colors[3] = quad_in.colors[3];
        // adl_interpolate_ARGBcolor_on_RGB((quad_in).colors[0], (quad_in).colors[1], t, &((*quad_out2).colors[1]));

        (*quad_out2).points[0].x = ((*quad_out2).points[3].x + (*quad_out2).points[1].x) / 2;
        (*quad_out2).points[0].y = ((*quad_out2).points[3].y + (*quad_out2).points[1].y) / 2;
        (*quad_out2).points[0].z = ((*quad_out2).points[3].z + (*quad_out2).points[1].z) / 2;
        (*quad_out2).points[0].w = ((*quad_out2).points[3].w + (*quad_out2).points[1].w) / 2;
```

```c
        // adl_interpolate_ARGBcolor_on_RGB((*quad_out2).colors[1], (*quad_out2).colors[3], 0.5f, &((*quad_out2).colors[2]));


        mat2D_free(line_start);
        mat2D_free(line_end);

        ae_assert_quad_is_valid(*quad_out1);

        return 2;
    } else if (inside_points_count == 3 && outside_points_count == 1) {
        Mat2D line_start = mat2D_alloc(3, 1);
        Mat2D line_end   = mat2D_alloc(3, 1);

        *quad_out1 = quad_in;

        (*quad_out1).points[0] = inside_points[0];
        (*quad_out1).points[1] = inside_points[1];
        (*quad_out1).points[2] = inside_points[2];
        ae_point_to_mat2D(inside_points[2], line_start);
        ae_point_to_mat2D(outside_points[0], line_end);
        (*quad_out1).points[3] = ae_line_itersect_plane(plane_p, plane_n, line_start, line_end, &t);
        (*quad_out1).points[3].w = t * (outside_points[0].w - inside_points[2].w) + inside_points[2].w;

        mat2D_free(line_start);
        mat2D_free(line_end);

        ae_assert_quad_is_valid(*quad_out1);

        return 1;
    }
    return -1;
}

void ae_projection_mat_set(Mat2D proj_mat,float aspect_ratio, float FOV_deg, float z_near, float z_far)
{
    AE_ASSERT(4 == proj_mat.cols);
    AE_ASSERT(4 == proj_mat.rows);
    AE_ASSERT(FOV_deg && "FOV needs to bee bigger then zero");

    mat2D_fill(proj_mat, 0);

    float field_of_view = 1.0f / tanf(0.5f * FOV_deg * PI / 180);
    float z_normalization = z_far / (z_far - z_near);

    MAT2D_AT(proj_mat, 0, 0) = aspect_ratio * field_of_view;
    MAT2D_AT(proj_mat, 1, 1) = field_of_view;
    MAT2D_AT(proj_mat, 2, 2) = z_normalization;
    MAT2D_AT(proj_mat, 2, 3) = 1;
    MAT2D_AT(proj_mat, 3, 2) = - z_normalization * z_near;
}

void ae_view_mat_set(Mat2D view_mat, Camera camera, Mat2D up)
{
    Mat2D DCM = mat2D_alloc(3,3);
    Mat2D DCM_trans = mat2D_alloc(3,3);
    mat2D_set_DCM_zyx(DCM, camera.yaw_offset_deg, camera.pitch_offset_deg, camera.roll_offset_deg);
    mat2D_transpose(DCM_trans, DCM);

    Mat2D temp_vec = mat2D_alloc(3, 1);
    Mat2D camera_direction = mat2D_alloc(3, 1);

    /* rotating camera_direction */
    mat2D_dot(camera_direction, DCM_trans, camera.direction);

    /* calc new forward direction */
    Mat2D new_forward = mat2D_alloc(3, 1);
    mat2D_copy(new_forward, camera_direction);
    mat2D_mult(new_forward, 1.0 / mat2D_calc_norma(new_forward));

    /* calc new up direction */
    mat2D_copy(temp_vec, new_forward);
    mat2D_mult(temp_vec, mat2D_dot_product(up, new_forward));
    Mat2D new_up = mat2D_alloc(3, 1);
    mat2D_copy(new_up, up);
    mat2D_sub(new_up, temp_vec);
    mat2D_mult(new_up, 1.0 / mat2D_calc_norma(new_up));

    /* calc new right direction */
    Mat2D new_right = mat2D_alloc(3, 1);
    mat2D_cross(new_right, new_up, new_forward);
    mat2D_mult(new_right, 1.0 / mat2D_calc_norma(new_right));

    mat2D_copy(camera.camera_x, new_right);
    mat2D_copy(camera.camera_y, new_up);
```

```
    mat2D_copy(camera.camera_z, new_forward);

    /* adding offset to init_position */
    // mat2D_add(camera_pos, camera.offset_position);

    mat2D_copy(temp_vec, camera.camera_x);
    mat2D_mult(temp_vec, MAT2D_AT(camera.offset_position, 0, 0));
    mat2D_add(camera.current_position, temp_vec);
    mat2D_copy(temp_vec, camera.camera_y);
    mat2D_mult(temp_vec, MAT2D_AT(camera.offset_position, 1, 0));
    mat2D_add(camera.current_position, temp_vec);
    mat2D_copy(temp_vec, camera.camera_z);
    mat2D_mult(temp_vec, MAT2D_AT(camera.offset_position, 2, 0));
    mat2D_add(camera.current_position, temp_vec);

    mat2D_fill(camera.offset_position, 0);

    MAT2D_AT(view_mat, 0, 0) = MAT2D_AT(new_right, 0, 0);
    MAT2D_AT(view_mat, 0, 1) = MAT2D_AT(new_up, 0, 0);
    MAT2D_AT(view_mat, 0, 2) = MAT2D_AT(new_forward, 0, 0);
    MAT2D_AT(view_mat, 0, 3) = 0;
    MAT2D_AT(view_mat, 1, 0) = MAT2D_AT(new_right, 1, 0);
    MAT2D_AT(view_mat, 1, 1) = MAT2D_AT(new_up, 1, 0);
    MAT2D_AT(view_mat, 1, 2) = MAT2D_AT(new_forward, 1, 0);
    MAT2D_AT(view_mat, 1, 3) = 0;
    MAT2D_AT(view_mat, 2, 0) = MAT2D_AT(new_right, 2, 0);
    MAT2D_AT(view_mat, 2, 1) = MAT2D_AT(new_up, 2, 0);
    MAT2D_AT(view_mat, 2, 2) = MAT2D_AT(new_forward, 2, 0);
    MAT2D_AT(view_mat, 2, 3) = 0;
    MAT2D_AT(view_mat, 3, 0) = - mat2D_dot_product(camera.current_position, new_right);
    MAT2D_AT(view_mat, 3, 1) = - mat2D_dot_product(camera.current_position, new_up);
    MAT2D_AT(view_mat, 3, 2) = - mat2D_dot_product(camera.current_position, new_forward);
    MAT2D_AT(view_mat, 3, 3) = 1;


    mat2D_free(temp_vec);
    mat2D_free(new_forward);
    mat2D_free(new_up);
    mat2D_free(new_right);
    mat2D_free(DCM);
    mat2D_free(DCM_trans);
    mat2D_free(camera_direction);
}

Point ae_point_project_world2screen(Mat2D view_mat, Mat2D proj_mat, Point src, int window_w, int window_h)
{
    Point view_point = ae_point_project_world2view(view_mat, src);
    Point screen_point = ae_point_project_view2screen(proj_mat, view_point, window_w, window_h);

    return screen_point;
}

Point ae_point_project_world2view(Mat2D view_mat, Point src)
{
    ae_assert_point_is_valid(src);

    Mat2D src_point_mat = mat2D_alloc(1,4);
    Mat2D des_point_mat = mat2D_alloc(1,4);

    Point des_point = {0};

    MAT2D_AT(src_point_mat, 0, 0) = src.x;
    MAT2D_AT(src_point_mat, 0, 1) = src.y;
    MAT2D_AT(src_point_mat, 0, 2) = src.z;
    MAT2D_AT(src_point_mat, 0, 3) = 1;

    mat2D_dot(des_point_mat, src_point_mat, view_mat);

    double w = MAT2D_AT(des_point_mat, 0, 3);
    AE_ASSERT(w == 1);
    des_point.x = MAT2D_AT(des_point_mat, 0, 0) / w;
    des_point.y = MAT2D_AT(des_point_mat, 0, 1) / w;
    des_point.z = MAT2D_AT(des_point_mat, 0, 2) / w;
    des_point.w = w;

    mat2D_free(src_point_mat);
    mat2D_free(des_point_mat);

    return des_point;

}

Point ae_point_project_view2screen(Mat2D proj_mat, Point src, int window_w, int window_h)
{
    ae_assert_point_is_valid(src);

```

```
    Mat2D src_point_mat = mat2D_alloc(1,4);
    Mat2D des_point_mat = mat2D_alloc(1,4);
    Point des;

    MAT2D_AT(src_point_mat, 0, 0) = src.x;
    MAT2D_AT(src_point_mat, 0, 1) = src.y;
    MAT2D_AT(src_point_mat, 0, 2) = src.z;
    MAT2D_AT(src_point_mat, 0, 3) = 1;

    mat2D_dot(des_point_mat, src_point_mat, proj_mat);

    double w = MAT2D_AT(des_point_mat, 0, 3);
    if (fabs(w) > 1e-3) {
        des.x = MAT2D_AT(des_point_mat, 0, 0) / w;
        des.y = MAT2D_AT(des_point_mat, 0, 1) / w;
        des.z = MAT2D_AT(des_point_mat, 0, 2) / w;
        des.w = w;
    } else {
        // des.x = MAT2D_AT(des_point_mat, 0, 0);
        // des.y = MAT2D_AT(des_point_mat, 0, 1);
        // des.z = MAT2D_AT(des_point_mat, 0, 2);
        // des.w = 1;

        des.x = 0;
        des.y = 0;
        des.z = 0;
        des.w = 1;
    }

    mat2D_free(src_point_mat);
    mat2D_free(des_point_mat);

    /* scale into view */
    des.x += 1;
    des.y += 1;

    des.x *= 0.5f * window_w;
    des.y *= 0.5f * window_h;

    return des;
}

void ae_line_project_world2screen(Mat2D view_mat, Mat2D proj_mat, Point start_src, Point end_src, int window_w, int window_h, Point *start_des, Point *end_des, Scene *scene)
{
    Point start_view_point = ae_point_project_world2view(view_mat, start_src);
    Point end_view_point = ae_point_project_world2view(view_mat, end_src);

    Mat2D z_p = mat2D_alloc(3, 1);
    Mat2D z_n = mat2D_alloc(3, 1);
    mat2D_fill(z_p, 0);
    mat2D_fill(z_n, 0);
    MAT2D_AT(z_p, 2, 0) = scene->camera.z_near+0.01;
    MAT2D_AT(z_n, 2, 0) = 1;

    Point clipped_start_view_point = {0}, clipped_end_view_point = {0};
    int rc = ae_line_clip_with_plane(start_view_point, end_view_point, z_p, z_n, &clipped_start_view_point, &clipped_end_view_point);

    if (rc == -1) {
        fprintf(stderr, "%s:%d: [error] problem with clipping lines\n", __FILE__, __LINE__);
        exit(1);
    } else if (rc == 0) {
        clipped_start_view_point = (Point){-1,-1,1,1};
        clipped_end_view_point = (Point){-1,-1,1,1};
        start_view_point = clipped_start_view_point;
        end_view_point = clipped_end_view_point;

        *start_des = start_view_point;
        *end_des = end_view_point;
        return;

    } else if (rc == 1) {
        start_view_point = clipped_start_view_point;
        end_view_point = clipped_end_view_point;
    }


    Point start_screen_point = ae_point_project_view2screen(proj_mat, start_view_point, window_w, window_h);
    Point end_screen_point = ae_point_project_view2screen(proj_mat, end_view_point, window_w, window_h);

    mat2D_free(z_p);
    mat2D_free(z_n);

    *start_des = start_screen_point;
```

```
    *end_des = end_screen_point;

}

Tri ae_tri_transform_to_view(Mat2D view_mat, Tri tri)
{
    ae_assert_tri_is_valid(tri);

    Mat2D src_point_mat = mat2D_alloc(1,4);
    Mat2D des_point_mat = mat2D_alloc(1,4);

    Tri des_tri = tri;

    for (int i = 0; i < 3; i++) {
        MAT2D_AT(src_point_mat, 0, 0) = tri.points[i].x;
        MAT2D_AT(src_point_mat, 0, 1) = tri.points[i].y;
        MAT2D_AT(src_point_mat, 0, 2) = tri.points[i].z;
        MAT2D_AT(src_point_mat, 0, 3) = 1;

        mat2D_dot(des_point_mat, src_point_mat, view_mat);

        double w = MAT2D_AT(des_point_mat, 0, 3);
        AE_ASSERT(w == 1);
        des_tri.points[i].x = MAT2D_AT(des_point_mat, 0, 0) / w;
        des_tri.points[i].y = MAT2D_AT(des_point_mat, 0, 1) / w;
        des_tri.points[i].z = MAT2D_AT(des_point_mat, 0, 2) / w;
        des_tri.points[i].w = w;
    }

    mat2D_free(src_point_mat);
    mat2D_free(des_point_mat);

    ae_assert_tri_is_valid(des_tri);

    return des_tri;
}

Tri_mesh ae_tri_project_world2screen(Mat2D proj_mat, Mat2D view_mat, Tri tri, int window_w, int window_h, Scene *scene, Lighting_mode lighting_mode)
{
    ae_assert_tri_is_valid(tri);

    Mat2D tri_normal = mat2D_alloc(3, 1);
    Mat2D temp_camera2tri = mat2D_alloc(3, 1);
    Mat2D camera2tri = mat2D_alloc(1, 3);
    Mat2D dot_product = mat2D_alloc(1, 1);
    Tri des_tri = tri;

    ae_point_to_mat2D(tri.points[0], temp_camera2tri);
    mat2D_sub(temp_camera2tri, scene->camera.current_position);
    mat2D_transpose(camera2tri, temp_camera2tri);

    /* calc lighting intensity of tri */
    #if 1
        ae_tri_calc_light_intensity(&des_tri, scene, lighting_mode);
    #else
    for (int i = 0; i < 3; i++) {
        ae_point_to_mat2D(tri.normals[i], tri_normal);
        MAT2D_AT(dot_product, 0, 0) = MAT2D_AT(light_direction, 0, 0) * MAT2D_AT(tri_normal, 0, 0) + MAT2D_AT(light_direction, 1, 0) * MAT2D_AT(tri_normal, 1, 0) + MAT2D_AT(light_direction, 2, 0) * MAT2D_AT(tri_normal, 2, 0);
        des_tri.light_intensity[i] = fmaxf(0.2, fminf(1, MAT2D_AT(dot_product, 0, 0)));
    }
    #endif

    /* calc if tri is visible to the camera */
    ae_tri_calc_normal(tri_normal, tri);
    // ae_point_to_mat2D(tri.normals[0], tri_normal);
    MAT2D_AT(dot_product, 0, 0) = MAT2D_AT(camera2tri, 0, 0) * MAT2D_AT(tri_normal, 0, 0) + MAT2D_AT(camera2tri, 0, 1) * MAT2D_AT(tri_normal, 1, 0) + MAT2D_AT(camera2tri, 0, 2) * MAT2D_AT(tri_normal, 2, 0);
    if (MAT2D_AT(dot_product, 0, 0) < 0) {
        des_tri.to_draw = true;
    } else {
        des_tri.to_draw = false;
    }

    /* transform tri to camera view */
    tri = ae_tri_transform_to_view(view_mat, tri);

    // Tri_mesh temp_tri_array;
    // ada_init_array(Tri, temp_tri_array);
    // ada_append(Tri, temp_tri_array, tri);
    /* clip tir */
    Tri clipped_tri1 = {0};
    Tri clipped_tri2 = {0};
    Mat2D z_plane_p = mat2D_alloc(3, 1);
```

```
    Mat2D z_plane_n = mat2D_alloc(3, 1);
    mat2D_fill(z_plane_p, 0);
    mat2D_fill(z_plane_n, 0);
    MAT2D_AT(z_plane_p, 2, 0) = scene->camera.z_near+0.01;
    MAT2D_AT(z_plane_n, 2, 0) = 1;

    int num_clipped_tri = ae_tri_clip_with_plane(tri, z_plane_p, z_plane_n, &clipped_tri1, &clipped_tri2);
    Tri_mesh temp_tri_array;
    ada_init_array(Tri, temp_tri_array);
    if (num_clipped_tri == -1) {
        fprintf(stderr, "%s:%d: [error] problem with clipping triangles\n", __FILE__, __LINE__);
        exit(1);
    } else if (num_clipped_tri == 0) {
        ;
    } else if (num_clipped_tri == 1) {
        ae_assert_tri_is_valid(clipped_tri1);
        ada_append(Tri, temp_tri_array, clipped_tri1);
    } else if (num_clipped_tri == 2) {
        ae_assert_tri_is_valid(clipped_tri1);
        ae_assert_tri_is_valid(clipped_tri2);
        ada_append(Tri, temp_tri_array, clipped_tri1);
        ada_append(Tri, temp_tri_array, clipped_tri2);
    }
    mat2D_free(z_plane_p);
    mat2D_free(z_plane_n);

    for (size_t temp_tri_index = 0; temp_tri_index < temp_tri_array.length; temp_tri_index++) {
        /* project tri to screen */
        for (int i = 0; i < 3; i++) {
            des_tri.points[i] = ae_point_project_view2screen(proj_mat, temp_tri_array.elements[temp_tri_index].points[i], window_w, window_h);

            if (des_tri.points[i].w) {
                des_tri.tex_points[i].x /= des_tri.points[i].w;
                des_tri.tex_points[i].y /= des_tri.points[i].w;
                des_tri.tex_points[i].z /= des_tri.points[i].w;
                des_tri.tex_points[i].w  = des_tri.points[i].w;
            }

        }
        ae_assert_tri_is_valid(des_tri);
        temp_tri_array.elements[temp_tri_index] = des_tri;
    }


    mat2D_free(tri_normal);
    mat2D_free(temp_camera2tri);
    mat2D_free(camera2tri);
    mat2D_free(dot_product);

    return temp_tri_array;
}

void ae_tri_mesh_project_world2screen(Mat2D proj_mat, Mat2D view_mat, Tri_mesh *des, Tri_mesh src, int window_w, int window_h, Scene *scene, Lighting_mode lighting_mode)
{
    Tri_mesh temp_des = *des;
    temp_des.length = 0;

    size_t i;
    for (i = 0; i < src.length; i++) {
        Tri_mesh temp_tri_array = ae_tri_project_world2screen(proj_mat, view_mat, src.elements[i], window_w, window_h, scene, lighting_mode);

        for (size_t tri_index = 0; tri_index < temp_tri_array.length; tri_index++) {
            Tri temp_tri = temp_tri_array.elements[tri_index];
            ada_append(Tri, temp_des, temp_tri);
        }

        free(temp_tri_array.elements);
    }

    /* clip tir */
    int offset = 0;
    Mat2D top_p = mat2D_alloc(3, 1);
    Mat2D top_n = mat2D_alloc(3, 1);
    mat2D_fill(top_p, 0);
    mat2D_fill(top_n, 0);
    MAT2D_AT(top_p, 1, 0) = 0 + offset;
    MAT2D_AT(top_n, 1, 0) = 1;

    Mat2D bottom_p = mat2D_alloc(3, 1);
    Mat2D bottom_n = mat2D_alloc(3, 1);
    mat2D_fill(bottom_p, 0);
    mat2D_fill(bottom_n, 0);
    MAT2D_AT(bottom_p, 1, 0) = window_h - offset;
```

```
    MAT2D_AT(bottom_n, 1, 0) = -1;

    Mat2D left_p = mat2D_alloc(3, 1);
    Mat2D left_n = mat2D_alloc(3, 1);
    mat2D_fill(left_p, 0);
    mat2D_fill(left_n, 0);
    MAT2D_AT(left_p, 0, 0) = 0 + offset;
    MAT2D_AT(left_n, 0, 0) = 1;

    Mat2D right_p = mat2D_alloc(3, 1);
    Mat2D right_n = mat2D_alloc(3, 1);
    mat2D_fill(right_p, 0);
    mat2D_fill(right_n, 0);
    MAT2D_AT(right_p, 0, 0) = window_w - offset;
    MAT2D_AT(right_n, 0, 0) = -1;

    for (int plane_number = 0; plane_number < 4; plane_number++) {
        for (int tri_index = 0; tri_index < (int)(temp_des.length); tri_index++) {
            if (temp_des.length == 0) {
                break;
            }
            // if (temp_des.elements[tri_index].to_draw == false) {
            //     ada_remove_unordered(Tri, temp_des, tri_index);
            //     tri_index--;
            //     tri_index = (int)fmaxf((float)tri_index, 0.0f);
            //     continue;
            // }
            Tri clipped_tri1 = {0};
            Tri clipped_tri2 = {0};
            int num_clipped_tri;
            switch (plane_number) {
                case 0:
                    num_clipped_tri = ae_tri_clip_with_plane(temp_des.elements[tri_index], top_p, top_n, &clipped_tri1, &clipped_tri2);
                break;
                case 1:
                    num_clipped_tri = ae_tri_clip_with_plane(temp_des.elements[tri_index], right_p, right_n, &clipped_tri1, &clipped_tri2);
                break;
                case 2:
                    num_clipped_tri = ae_tri_clip_with_plane(temp_des.elements[tri_index], bottom_p, bottom_n, &clipped_tri1, &clipped_tri2);
                break;
                case 3:
                    num_clipped_tri = ae_tri_clip_with_plane(temp_des.elements[tri_index], left_p, left_n, &clipped_tri1, &clipped_tri2);
                break;
            }
            if (num_clipped_tri == -1) {
                fprintf(stderr, "%s:%d: [error] problem with clipping triangles\n", __FILE__, __LINE__);
                exit(1);
            } else if (num_clipped_tri == 0) {
                ada_remove_unordered(Tri, temp_des, tri_index);
                tri_index--;
                tri_index = (int)fmaxf((float)tri_index, 0.0f);
            } else if (num_clipped_tri == 1) {
                ae_assert_tri_is_valid(clipped_tri1);
                temp_des.elements[tri_index] = clipped_tri1;
            } else if (num_clipped_tri == 2) {
                ae_assert_tri_is_valid(clipped_tri1);
                ae_assert_tri_is_valid(clipped_tri2);
                temp_des.elements[tri_index] = clipped_tri1;
                ada_insert_unordered(Tri, temp_des, clipped_tri2, tri_index+1);
            }
        }
    }

    // if (temp_des.length > 2) {
    //     ae_qsort_tri(temp_des.elements, 0, temp_des.length-1);
    // }

    mat2D_free(top_p);
    mat2D_free(top_n);
    mat2D_free(bottom_p);
    mat2D_free(bottom_n);
    mat2D_free(left_p);
    mat2D_free(left_n);
    mat2D_free(right_p);
    mat2D_free(right_n);

    *des = temp_des;
}

Quad ae_quad_transform_to_view(Mat2D view_mat, Quad quad)
{
    ae_assert_quad_is_valid(quad);
```

```
    Mat2D src_point_mat = mat2D_alloc(1,4);
    Mat2D des_point_mat = mat2D_alloc(1,4);

    Quad des_quad = quad;

    for (int i = 0; i < 4; i++) {
        MAT2D_AT(src_point_mat, 0, 0) = quad.points[i].x;
        MAT2D_AT(src_point_mat, 0, 1) = quad.points[i].y;
        MAT2D_AT(src_point_mat, 0, 2) = quad.points[i].z;
        MAT2D_AT(src_point_mat, 0, 3) = 1;

        mat2D_dot(des_point_mat, src_point_mat, view_mat);

        double w = MAT2D_AT(des_point_mat, 0, 3);
        AE_ASSERT(w == 1);
        des_quad.points[i].x = MAT2D_AT(des_point_mat, 0, 0) / w;
        des_quad.points[i].y = MAT2D_AT(des_point_mat, 0, 1) / w;
        des_quad.points[i].z = MAT2D_AT(des_point_mat, 0, 2) / w;
        des_quad.points[i].w = w;
    }

    mat2D_free(src_point_mat);
    mat2D_free(des_point_mat);

    ae_assert_quad_is_valid(des_quad);

    return des_quad;
}

Quad_mesh ae_quad_project_world2screen(Mat2D proj_mat, Mat2D view_mat, Quad quad, int window_w, int window_h, Scene *scene, Lighting_mode lighting_mode)
{
    ae_assert_quad_is_valid(quad);

    Mat2D quad_normal = mat2D_alloc(3, 1);
    Mat2D camera2quad = mat2D_alloc(3, 1);
    Mat2D dot_product = mat2D_alloc(1, 1);
    Quad des_quad = quad;

    /* calc lighting intensity of tri */
    #if 1
        ae_quad_calc_light_intensity(&des_quad, scene, lighting_mode);
    #else
    for (int i = 0; i < 4; i++) {
        ae_point_to_mat2D(quad.normals[i], quad_normal);
        MAT2D_AT(dot_product, 0, 0) = scene->light_source0.light_direction_or_pos.x * MAT2D_AT(quad_normal, 0, 0) + scene->light_source0.light_direction_or_pos.y * MAT2D_AT(quad_normal, 1, 0) + scene->light_source0.light_direction_or_pos.z * MAT2D_AT(quad_normal, 2, 0);
        des_quad.light_intensity[i] = fmaxf(0.2, fminf(1, MAT2D_AT(dot_product, 0, 0)));
    }
    #endif

    /* calc if quad is visible to the camera */
    bool visible = 0;
    #if 1
    for (int i = 0; i < 4; i++) {
        ae_point_to_mat2D(quad.points[i], camera2quad);
        mat2D_sub(camera2quad, scene->camera.current_position);

        ae_point_to_mat2D(quad.normals[i], quad_normal);
        MAT2D_AT(dot_product, 0, 0) = MAT2D_AT(camera2quad, 0, 0) * MAT2D_AT(quad_normal, 0, 0) + MAT2D_AT(camera2quad, 1, 0) * MAT2D_AT(quad_normal, 1, 0) + MAT2D_AT(camera2quad, 2, 0) * MAT2D_AT(quad_normal, 2, 0);
        visible = visible || (MAT2D_AT(dot_product, 0, 0) < 0);
    }
    #else
    ae_point_to_mat2D(quad.points[0], camera2quad);
    mat2D_sub(camera2quad, scene->camera.current_position);

    Point ave_norm = ae_quad_get_average_normal(quad);
    ae_point_to_mat2D(ave_norm, quad_normal);
    MAT2D_AT(dot_product, 0, 0) = MAT2D_AT(camera2quad, 0, 0) * MAT2D_AT(quad_normal, 0, 0) + MAT2D_AT(camera2quad, 1, 0) * MAT2D_AT(quad_normal, 1, 0) + MAT2D_AT(camera2quad, 2, 0) * MAT2D_AT(quad_normal, 2, 0);
    visible = MAT2D_AT(dot_product, 0, 0) < 0;
    #endif

    if (visible) {
        des_quad.to_draw = true;
    } else {
        des_quad.to_draw = false;
    }

    /* transform quad to camera view */
    quad = ae_quad_transform_to_view(view_mat, quad);

```

```
    /* clip quad */
    Quad  clipped_quad1 = {0}, clipped_quad2 = {0};
    Mat2D z_plane_p = mat2D_alloc(3, 1);
    Mat2D z_plane_n = mat2D_alloc(3, 1);
    mat2D_fill(z_plane_p, 0);
    mat2D_fill(z_plane_n, 0);
    MAT2D_AT(z_plane_p, 2, 0) = scene->camera.z_near+0.01;
    MAT2D_AT(z_plane_n, 2, 0) = 1;

    int num_clipped_quad = ae_quad_clip_with_plane(quad, z_plane_p, z_plane_n, &clipped_quad1, &clipped_quad2);
    Quad_mesh temp_quad_array;
    ada_init_array(Quad, temp_quad_array);
    if (num_clipped_quad == -1) {
        fprintf(stderr, "%s:%d: [error] problem with clipping quad\n", __FILE__, __LINE__);
        exit(1);
    } else if (num_clipped_quad == 0) {
        ;
    } else if (num_clipped_quad == 1) {
        ae_assert_quad_is_valid(clipped_quad1);
        ada_append(Quad, temp_quad_array, clipped_quad1);
    } else if (num_clipped_quad == 2) {
        ae_assert_quad_is_valid(clipped_quad1);
        ae_assert_quad_is_valid(clipped_quad2);
        ada_append(Quad, temp_quad_array, clipped_quad1);
        ada_append(Quad, temp_quad_array, clipped_quad2);
    }
    mat2D_free(z_plane_p);
    mat2D_free(z_plane_n);

    for (size_t temp_quad_index = 0; temp_quad_index < temp_quad_array.length; temp_quad_index++) {
        /* project quad to screen */
        for (int i = 0; i < 4; i++) {
            des_quad.points[i] = ae_point_project_view2screen(proj_mat, temp_quad_array.elements[temp_quad_index].points[i], window_w, window_h);

        }
        ae_assert_quad_is_valid(des_quad);
        temp_quad_array.elements[temp_quad_index] = des_quad;
    }


    mat2D_free(quad_normal);
    mat2D_free(camera2quad);
    mat2D_free(dot_product);

    return temp_quad_array;
}

void ae_quad_mesh_project_world2screen(Mat2D proj_mat, Mat2D view_mat, Quad_mesh *des, Quad_mesh src, int window_w, int window_h, Scene *scene, Lighting_mode lighting_mode)
{
    Quad_mesh temp_des = *des;
    temp_des.length = 0;

    size_t i;
    for (i = 0; i < src.length; i++) {
        Quad_mesh temp_quad_array = ae_quad_project_world2screen(proj_mat, view_mat, src.elements[i], window_w, window_h, scene, lighting_mode);

        for (size_t quad_index = 0; quad_index < temp_quad_array.length; quad_index++) {
            Quad temp_quad = temp_quad_array.elements[quad_index];
            ada_append(Quad, temp_des, temp_quad);
        }

        free(temp_quad_array.elements);
    }


    /* clip quad */
    int offset = 0;
    Mat2D top_p = mat2D_alloc(3, 1);
    Mat2D top_n = mat2D_alloc(3, 1);
    mat2D_fill(top_p, 0);
    mat2D_fill(top_n, 0);
    MAT2D_AT(top_p, 1, 0) = 0 + offset;
    MAT2D_AT(top_n, 1, 0) = 1;

    Mat2D bottom_p = mat2D_alloc(3, 1);
    Mat2D bottom_n = mat2D_alloc(3, 1);
    mat2D_fill(bottom_p, 0);
    mat2D_fill(bottom_n, 0);
    MAT2D_AT(bottom_p, 1, 0) = window_h - offset;
    MAT2D_AT(bottom_n, 1, 0) = -1;

    Mat2D left_p = mat2D_alloc(3, 1);
    Mat2D left_n = mat2D_alloc(3, 1);
```

```
    mat2D_fill(left_p, 0);
    mat2D_fill(left_n, 0);
    MAT2D_AT(left_p, 0, 0) = 0 + offset;
    MAT2D_AT(left_n, 0, 0) = 1;

    Mat2D right_p = mat2D_alloc(3, 1);
    Mat2D right_n = mat2D_alloc(3, 1);
    mat2D_fill(right_p, 0);
    mat2D_fill(right_n, 0);
    MAT2D_AT(right_p, 0, 0) = window_w - offset;
    MAT2D_AT(right_n, 0, 0) = -1;

    for (int plane_number = 0; plane_number < 4; plane_number++) {
        for (int quad_index = 0; quad_index < (int)(temp_des.length); quad_index++) {
            if (temp_des.length == 0) {
                break;
            }
            // if (temp_des.elements[quad_index].to_draw == false) {
            //     ada_remove_unordered(Quad, temp_des, quad_index);
            //     quad_index--;
            //     quad_index = (int)fmaxf((float)quad_index, 0.0f);
            //     continue;
            // }
            Quad clipped_quad1 = {0}, clipped_quad2 = {0};
            int num_clipped_quad;
            switch (plane_number) {
                case 0:
                    num_clipped_quad = ae_quad_clip_with_plane(temp_des.elements[quad_index], top_p, top_n, &clipped_quad1, &clipped_quad2);
                break;
                case 1:
                    num_clipped_quad = ae_quad_clip_with_plane(temp_des.elements[quad_index], right_p, right_n, &clipped_quad1, &clipped_quad2);
                break;
                case 2:
                    num_clipped_quad = ae_quad_clip_with_plane(temp_des.elements[quad_index], bottom_p, bottom_n, &clipped_quad1, &clipped_quad2);
                break;
                case 3:
                    num_clipped_quad = ae_quad_clip_with_plane(temp_des.elements[quad_index], left_p, left_n, &clipped_quad1, &clipped_quad2);
                break;
            }
            if (num_clipped_quad == -1) {
                fprintf(stderr, "%s:%d: [error] problem with clipping quads\n", __FILE__, __LINE__);
                exit(1);
            } else if (num_clipped_quad == 0) {
                ada_remove_unordered(Quad, temp_des, quad_index);
                quad_index--;
                quad_index = (int)fmaxf((float)quad_index, 0.0f);
            } else if (num_clipped_quad == 1) {
                ae_assert_quad_is_valid(clipped_quad1);
                temp_des.elements[quad_index] = clipped_quad1;
            } else if (num_clipped_quad == 2) {
                ae_assert_quad_is_valid(clipped_quad1);
                ae_assert_quad_is_valid(clipped_quad2);
                temp_des.elements[quad_index] = clipped_quad1;
                ada_insert_unordered(Quad, temp_des, clipped_quad2, quad_index+1);
                quad_index++;
            }
        }
    }

    mat2D_free(top_p);
    mat2D_free(top_n);
    mat2D_free(bottom_p);
    mat2D_free(bottom_n);
    mat2D_free(left_p);
    mat2D_free(left_n);
    mat2D_free(right_p);
    mat2D_free(right_n);

    *des = temp_des;
}


void ae_curve_project_world2screen(Mat2D proj_mat, Mat2D view_mat, Curve *des, Curve src, int window_w, int window_h, Scene *scene)
{
    ae_curve_copy(des, src);
    Curve temp_des = *des;
    /* set planes */
    int offset = 50;
    Mat2D top_p = mat2D_alloc(3, 1);
    Mat2D top_n = mat2D_alloc(3, 1);
    mat2D_fill(top_p, 0);
    mat2D_fill(top_n, 0);
```

```
    MAT2D_AT(top_p, 1, 0) = 0 + offset;
    MAT2D_AT(top_n, 1, 0) = 1;
    Mat2D bottom_p = mat2D_alloc(3, 1);
    Mat2D bottom_n = mat2D_alloc(3, 1);
    mat2D_fill(bottom_p, 0);
    mat2D_fill(bottom_n, 0);
    MAT2D_AT(bottom_p, 1, 0) = window_h - offset;
    MAT2D_AT(bottom_n, 1, 0) = -1;
    Mat2D left_p = mat2D_alloc(3, 1);
    Mat2D left_n = mat2D_alloc(3, 1);
    mat2D_fill(left_p, 0);
    mat2D_fill(left_n, 0);
    MAT2D_AT(left_p, 0, 0) = 0 + offset;
    MAT2D_AT(left_n, 0, 0) = 1;
    Mat2D right_p = mat2D_alloc(3, 1);
    Mat2D right_n = mat2D_alloc(3, 1);
    mat2D_fill(right_p, 0);
    mat2D_fill(right_n, 0);
    MAT2D_AT(right_p, 0, 0) = window_w - offset;
    MAT2D_AT(right_n, 0, 0) = -1;

    for (size_t point_index = 0; point_index < temp_des.length-1; point_index++) {
        Point start_src_point = src.elements[point_index];
        Point end_src_point = src.elements[point_index+1];

        Point start_des_point = {0}, end_des_point = {0};

        ae_line_project_world2screen(view_mat, proj_mat, start_src_point, end_src_point, window_w,
window_h, &start_des_point, &end_des_point, scene);


        Point clipped_start_des_point = {0}, clipped_end_des_point = {0};

        for (int plane_number = 0; plane_number < 4; plane_number++) {
            int rc;
            switch (plane_number) {
                case 0:
                    rc = ae_line_clip_with_plane(start_des_point, end_des_point, top_p, top_n,
&clipped_start_des_point, &clipped_end_des_point);
                break;
                case 1:
                    rc = ae_line_clip_with_plane(start_des_point, end_des_point, right_p, right_n,
&clipped_start_des_point, &clipped_end_des_point);
                break;
                case 2:
                    rc = ae_line_clip_with_plane(start_des_point, end_des_point, bottom_p, bottom_n,
&clipped_start_des_point, &clipped_end_des_point);
                break;
                case 3:
                    rc = ae_line_clip_with_plane(start_des_point, end_des_point, left_p, left_n,
&clipped_start_des_point, &clipped_end_des_point);
                break;
            }
            if (rc == -1) {
                fprintf(stderr, "%s:%d: [error] problem with clipping lines\n", __FILE__, __LINE__);
                exit(1);
            } else if (rc == 0) {
                clipped_start_des_point = (Point){-1,-1,1,1};
                clipped_end_des_point = (Point){-1,-1,1,1};
                start_des_point = clipped_start_des_point;
                end_des_point = clipped_end_des_point;
                temp_des.elements[point_index] = start_des_point;
                temp_des.elements[point_index+1] = end_des_point;
            } else if (rc == 1) {
                start_des_point = clipped_start_des_point;
                end_des_point = clipped_end_des_point;
                temp_des.elements[point_index] = start_des_point;
                temp_des.elements[point_index+1] = end_des_point;
            }
        }

    }

    Point default_point = (Point){-1,-1,1,1};
    for (int i = 0; i < (int)temp_des.length; i++) {
        if (ae_points_equal(temp_des.elements[i], default_point)) {
            ada_remove(Point, temp_des, i);
            i--;
        }
    }

    *des = temp_des;

    mat2D_free(top_p);
    mat2D_free(top_n);
    mat2D_free(bottom_p);
    mat2D_free(bottom_n);
```

```
    mat2D_free(left_p);
    mat2D_free(left_n);
    mat2D_free(right_p);
    mat2D_free(right_n);
}

void ae_curve_ada_project_world2screen(Mat2D proj_mat, Mat2D view_mat, Curve_ada *des, Curve_ada src, int window_w, int window_h, Scene *scene)
{
    if (src.length == 0) return;
    for (size_t curve_index = 0; curve_index < src.length; curve_index++) {
        ae_curve_project_world2screen(proj_mat, view_mat, &(des->elements[curve_index]), src.elements[curve_index], window_w, window_h, scene);
    }
}

void ae_grid_project_world2screen(Mat2D proj_mat, Mat2D view_mat, Grid des, Grid src, int window_w, int window_h, Scene *scene)
{
    if (src.curves.length == 0) return;
    for (size_t curve_index = 0; curve_index < src.curves.length; curve_index++) {
        ae_curve_project_world2screen(proj_mat, view_mat, &(des.curves.elements[curve_index]), src.curves.elements[curve_index], window_w, window_h, scene);
    }
}

void ae_tri_swap(Tri *v, int i, int j)
{
    Tri temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

bool ae_tri_compare(Tri t1, Tri t2)
{
    float t1_z_max = fmaxf(t1.points[0].z, fmaxf(t1.points[1].z, t1.points[2].z));
    float t2_z_max = fmaxf(t2.points[0].z, fmaxf(t2.points[1].z, t2.points[2].z));

    return t1_z_max > t2_z_max;
}

void ae_tri_qsort(Tri *v, int left, int right)
{
    int i, last;

    if (left >= right)                  /* do nothing if array contains */
        return;                         /* fewer than two elements */
    ae_tri_swap(v, left, (left + right) / 2);  /* move partition elem */
    last = left;                        /* to v[0] */
    for (i = left + 1; i <= right; i++) /* partition */
        if (ae_tri_compare(v[i], v[left]))
            ae_tri_swap(v, ++last, i);
    ae_tri_swap(v, left, last); /* restore partition elem */
    ae_tri_qsort(v, left, last - 1);
    ae_tri_qsort(v, last + 1, right);
}

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

void ae_z_buffer_copy_to_screen(Mat2D_uint32 screen_mat, Mat2D inv_z_buffer)
{
    double max_inv_z = 0;
    double min_inv_z = DBL_MAX;
    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 = ae_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(screen_mat, i, j) = color;
        }
    }
```

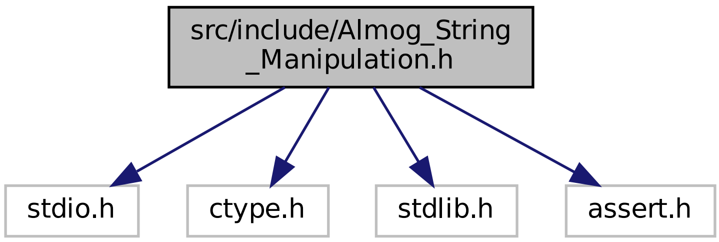
```
}

#endif /* ALMOG_ENGINE_IMPLEMENTATION */
```

4.9 src/include/Almog_String_Manipulation.h File Reference

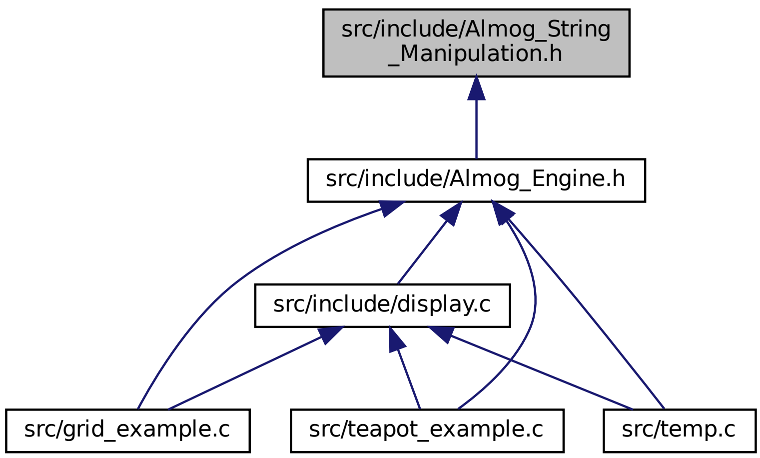
Lightweight string and line manipulation helpers.

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

Include dependency graph for Almog_String_Manipulation.h:



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



Macros

- #define ASM_MAXDIR 100

  Generic maximum directory length constant (not used by the functions in this header but available to callers).
- #define ASM_MAX_LEN_LINE (int)1e3

  Maximum number of characters read by asm_get_line (excluding the terminating null).
- #define asm_dprintSTRING(expr) printf(#expr " = %s\n", expr)

  Debug print a C string expression as "expr = value\n".
- #define asm_dprintCHAR(expr) printf(#expr " = %c\n", expr)

  Debug print a character expression as "expr = c\n".
- #define asm_dprintINT(expr) printf(#expr " = %d\n", expr)

  Debug print an integer expression as "expr = n\n".
- #define asm_dprintSIZE_T(expr) printf(#expr " = %zu\n", expr)

  Debug print a size_t expression as "expr = n\n".

Functions

- int asm_get_line (FILE ∗fp, char ∗dst)

  Read a single line from a stream into a buffer.
- int asm_length (char ∗str)

  Compute the length of a null-terminated C string.
- int asm_get_next_word_from_line (char ∗dst, char ∗src, char seperator)

  Extract the next word from a line without modifying the source.
- void asm_copy_array_by_indesies (char ∗target, int start, int end, char ∗src)

  Copy a substring [start, end) from src into target and null-terminate.
- int asm_get_word_and_cut (char ∗dst, char ∗src, char seperator)

  Get the next word and cut the source string at that point.
- int asm_str_in_str (char ∗src, char ∗word2search)

  Count occurrences of a substring within a string.
- int asm_strncmp (const char ∗s1, const char ∗s2, const int N)

  Compare up to N characters for equality (boolean result).

4.9.1 Detailed Description

Lightweight string and line manipulation helpers.

This single-header module provides small utilities for working with C strings:

- Reading a single line from a FILE stream
- Measuring string length
- Extracting the next "word" (token) from a line using a separator
- Cutting the extracted word from the source buffer
- Copying a substring by indices
- Counting occurrences of a substring
- A boolean-style strncmp (returns 1 on equality, 0 otherwise)

Usage

- In exactly one translation unit, define ALMOG_STRING_MANIPULATION_IMPLEMENTATION before including this header to compile the implementation.
- In all other files, include the header without the macro to get declarations only.

Notes and limitations

- All destination buffers must be large enough; functions do not grow or allocate buffers.
- asm_get_line enforces MAX_LEN_LINE characters (not counting the terminating '\0'). Longer lines cause a fatal error via exit(1).
- asm_strncmp differs from the standard C strncmp: this version returns 1 if equal and 0 otherwise.
- These functions are not locale-aware unless otherwise noted (isspace is used for whitespace handling).

Definition in file Almog_String_Manipulation.h.

4.9.2 Macro Definition Documentation

4.9.2.1 asm_dprintCHAR

```
#define asm_dprintCHAR(
              expr ) printf(#expr " = %c\n", expr)
```

Debug print a character expression as "expr = c\n".

Parameters

expr	An expression that yields a character promoted to int.

Definition at line 72 of file Almog_String_Manipulation.h.

4.9.2.2 asm_dprintINT

```
#define asm_dprintINT(
              expr ) printf(#expr " = %d\n", expr)
```

Debug print an integer expression as "expr = n\n".

Parameters

expr	An expression that yields an int.

Definition at line 79 of file Almog_String_Manipulation.h.

4.9.2.3 asm_dprintSIZE_T

```
#define asm_dprintSIZE_T(
              expr ) printf(#expr " = %zu\n", expr)
```

Debug print a size_t expression as "expr = n\n".

Parameters

expr	An expression that yields a size_t.

Definition at line 86 of file Almog_String_Manipulation.h.

4.9.2.4 asm_dprintSTRING

```
#define asm_dprintSTRING(
              expr ) printf(#expr " = %s\n", expr)
```

Debug print a C string expression as "expr = value\n".

Parameters

expr	An expression that yields a pointer to char (const or non-const).

Definition at line 65 of file Almog_String_Manipulation.h.

4.9.2.5 ASM_MAX_LEN_LINE

```
#define ASM_MAX_LEN_LINE (int)1e3
```

Maximum number of characters read by asm_get_line (excluding the terminating null).

If an input line exceeds this value before encountering '
' or EOF, asm_get_line prints an error to stderr and terminates the process with exit(1).

Definition at line 58 of file Almog_String_Manipulation.h.

4.9.2.6 ASM_MAXDIR

```
#define ASM_MAXDIR 100
```

Generic maximum directory length constant (not used by the functions in this header but available to callers).

Definition at line 47 of file Almog_String_Manipulation.h.

4.9.3 Function Documentation

4.9.3.1 asm_copy_array_by_indesies()

```
void asm_copy_array_by_indesies (
            char * target,
            int start,
            int end,
            char * src )
```

Copy a substring [start, end) from src into target and null-terminate.

Copies characters with indices i = start, start+1, ..., end-1 from src into target, then writes a terminating '\0'.

Parameters

target	Destination buffer. Must be large enough to hold (end - start) characters plus the null terminator.
start	Inclusive start index within src (0-based).
end	Exclusive end index within src (must satisfy end >= start).
src	Source string buffer.

Warning

No bounds checking is performed. The caller must ensure valid indices and sufficient target capacity.

Note

This routine supports in-place "left-shift" usage where target == src and start > 0 (used by asm_get_word_↵
and_cut).

Definition at line 232 of file Almog_String_Manipulation.h.

Referenced by asm_get_word_and_cut().

4.9.3.2 asm_get_line()

```
int asm_get_line (
            FILE * fp,
            char * dst )
```

Read a single line from a stream into a buffer.

Reads characters from the FILE stream until a newline ('
') or EOF is encountered. The newline, if present, is not copied. The result is always null-terminated.

Parameters

fp	Input stream (must be non-NULL).
dst	Destination buffer. Must have capacity of at least MAX_LEN_LINE + 1 bytes.

Returns

Number of characters stored in dst (excluding the terminating null).

Return values

-1	EOF was encountered before any character was read.

Note

If the line exceeds MAX_LEN_LINE characters before a newline or EOF, the function prints an error and calls exit(1).

An empty line returns 0 (not -1).

Definition at line 119 of file Almog_String_Manipulation.h.

References ASM_MAX_LEN_LINE.

Referenced by ae_tri_mesh_get_from_obj_file().

4.9.3.3 asm_get_next_word_from_line()

```
int asm_get_next_word_from_line (
            char * dst,
            char * src,
            char seperator )
```

Extract the next word from a line without modifying the source.

Skips leading whitespace in src (as determined by isspace), then copies characters into dst until one of the following is seen: the separator, a newline ('
'), or the string terminator ('\0'). The copied word in dst is null-terminated and is never empty on success.

Special case:

- If the very first character in src (at index 0, without leading whitespace) is the separator, '
  ', or '\0', that single character is returned as a one-character "word".

Parameters

dst	Destination buffer for the extracted word. Must be large enough to hold the token plus the null terminator.
src	Source C string to parse (not modified by this function).
seperator	Separator character to stop at (spelling as in the API).

Returns

The number of characters consumed from src (i.e., the index of the first unconsumed character).

Return values

-1	No word was found (e.g., only whitespace before a delimiter or end-of-string).

Note

The source buffer is not altered. To both extract and advance/cut the source, see asm_get_word_and_cut.

Definition at line 182 of file Almog_String_Manipulation.h.

Referenced by ae_tri_mesh_get_from_obj_file(), and asm_get_word_and_cut().

4.9.3.4 asm_get_word_and_cut()

```
int asm_get_word_and_cut (
            char * dst,
            char * src,
            char seperator )
```

Get the next word and cut the source string at that point.

Extracts the next word from src (per asm_get_next_word_from_line semantics) into dst. On success, src is modified in-place to remove the consumed prefix. The new src begins at the stopping character (the separator, newline, or terminator).

Example: For src = "abc,def", separator = ','

- dst becomes "abc"
- src becomes ",def" (note the leading separator remains)

Parameters

dst	Destination buffer for the extracted word (large enough for the token and terminating null).
src	Source buffer. Modified in-place if a word is found.
seperator	Separator character to stop at (spelling as in the API).

Returns

1 if a word was extracted and src adjusted, 0 otherwise.

Definition at line 260 of file Almog_String_Manipulation.h.

References asm_copy_array_by_indesies(), asm_get_next_word_from_line(), and asm_length().

Referenced by ae_tri_mesh_get_from_file(), and ae_tri_mesh_get_from_obj_file().

4.9.3.5 asm_length()

```
int asm_length (
            char * str )
```

Compute the length of a null-terminated C string.

Parameters

str	Null-terminated string (must be non-NULL).

Returns

The number of characters before the terminating null byte.

Definition at line 146 of file Almog_String_Manipulation.h.

Referenced by ae_tri_mesh_get_from_obj_file(), asm_get_word_and_cut(), and asm_str_in_str().

4.9.3.6 asm_str_in_str()

```
int asm_str_in_str (
            char * src,
            char * word2search )
```

Count occurrences of a substring within a string.

Counts how many times word2search appears in src. Occurrences may overlap.

Parameters

src	The string to search in (must be null-terminated).
word2search	The substring to find (must be null-terminated).

Returns

The number of (possibly overlapping) occurrences found.

Definition at line 285 of file Almog_String_Manipulation.h.

References asm_length(), and asm_strncmp().

Referenced by ae_tri_mesh_get_from_file(), and ae_tri_mesh_get_from_obj_file().

4.9.3.7 asm_strncmp()

```
int asm_strncmp (
            const char * s1,
            const char * s2,
            const int N )
```

Compare up to N characters for equality (boolean result).

Returns 1 if the first N characters of s1 and s2 are all equal; otherwise returns 0. Unlike the standard C strncmp, which returns 0 on equality and a non-zero value on inequality/order, this function returns a boolean-like result (1 == equal, 0 == different).

Parameters

s1	First string (may be shorter than N).
s2	Second string (may be shorter than N).
N	Number of characters to compare.

Returns

1 if equal for the first N characters, 0 otherwise.

Definition at line 310 of file Almog_String_Manipulation.h.

Referenced by asm_str_in_str().

4.10 Almog_String_Manipulation.h

```

#ifndef ALMOG_STRING_MANIPULATION_H_
#define ALMOG_STRING_MANIPULATION_H_

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <assert.h>

#define ASM_MAXDIR 100

#define ASM_MAX_LEN_LINE (int)1e3

#define asm_dprintSTRING(expr) printf(#expr " = %s\n", expr)

#define asm_dprintCHAR(expr) printf(#expr " = %c\n", expr)
```

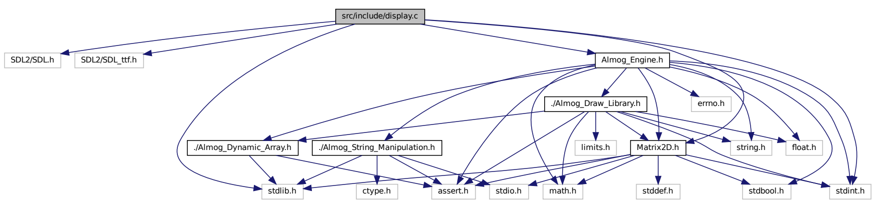
```c

#define asm_dprintINT(expr) printf(#expr " = %d\n", expr)

#define asm_dprintSIZE_T(expr) printf(#expr " = %zu\n", expr)

int asm_get_line(FILE *fp, char *dst);
int asm_length(char *str);
int asm_get_next_word_from_line(char *dst, char *src, char seperator);
void asm_copy_array_by_indesies(char *target, int start, int end, char *src);
int asm_get_word_and_cut(char *dst, char *src, char seperator);
int asm_str_in_str(char *src, char *word2search);
int asm_strncmp(const char *s1, const char *s2, const int N);

#endif /*ALMOG_STRING_MANIPULATION_H_*/

#ifdef ALMOG_STRING_MANIPULATION_IMPLEMENTATION
#undef ALMOG_STRING_MANIPULATION_IMPLEMENTATION


int asm_get_line(FILE *fp, char *dst)
{
    int i = 0;
    char c;

    while ((c = fgetc(fp)) != '\n' && c != EOF) {
        dst[i] = c;
        i++;
        if (i >= ASM_MAX_LEN_LINE) {
            fprintf(stderr, "ERROR: line too long\n");
            exit(1);
        }
    }
    dst[i] = '\0';
    if (c == EOF && i == 0) {
        return -1;
    }
    return i;
}

int asm_length(char *str)
{
    char c;
    int i = 0;

    while ((c = str[i]) != '\0') {
        i++;
    }
    return i++;
}

int asm_get_next_word_from_line(char *dst, char *src, char seperator)
{
    int i = 0, j = 0;
    char c;

    while (isspace((c = src[i]))) {
        i++;
    }

    while ((c = src[i]) != seperator &&
                     c != '\n'&&
                     c != '\0') {
                       dst[j] = src[i];
                       i++;
                       j++;
    }

    if ((c == seperator ||
         c == '\n'||
         c == '\0') && i == 0) {
            dst[j++] = c;
            i++;
    }

    dst[j] = '\0';

    if (j == 0) {
        return -1;
    }
    return i;

}

void asm_copy_array_by_indesies(char *target, int start, int end, char *src)
{
    int j = 0;
    for (int i = start; i < end; i++) {
```

```
        target[j] = src[i];
        j++;
    }
    target[j] = '\0';
}

int asm_get_word_and_cut(char *dst, char *src, char seperator)
{
    int last_pos;

    if (src[0] == '\0') {
        return 0;
    }
    last_pos = asm_get_next_word_from_line(dst, src, seperator);
    if (last_pos == -1) {
        return 0;
    }
    asm_copy_array_by_indesies(src, last_pos, asm_length(src), src);
    return 1;
}

int asm_str_in_str(char *src, char *word2search)
{
    int i = 0, num_of_accur = 0;
    while (src[i] != '\0') {
        if (asm_strncmp(src+i, word2search, asm_length(word2search))) {
            num_of_accur++;
        }
        i++;
    }
    return num_of_accur;
}

int asm_strncmp(const char *s1, const char *s2, const int N)
{
    int i = 0;
    while (i < N) {
        if (s1[i] == '\0' && s2[i] == '\0') {
            break;
        }
        if (s1[i] != s2[i] || (s1[i] == '\0') || (s2[i] == '\0')) {
            return 0;
        }
        i++;
    }
    return 1;
}


#endif /*ALMOG_STRING_MANIPULATION_IMPLEMENTATION*/

```

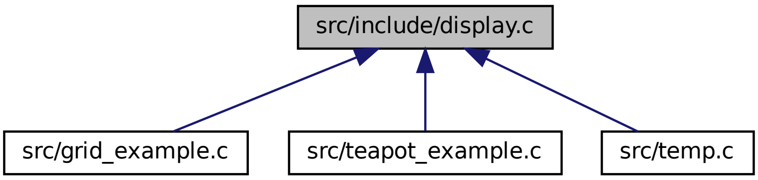
4.11 src/include/display.c File Reference

```
#include <SDL2/SDL.h>
#include <SDL2/SDL_ttf.h>
#include "Matrix2D.h"
#include <stdlib.h>
#include <stdint.h>
#include "Almog_Engine.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)

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 ()

4.11.1 Macro Definition Documentation

4.11.1.1 dprintCHAR

```
#define dprintCHAR(
            expr ) printf(#expr " = %c\n", expr)
```

Definition at line 25 of file display.c.

4.11.1.2 dprintD

```
#define dprintD(
            expr ) printf(#expr " = %g\n", expr)
```

Definition at line 27 of file display.c.

4.11.1.3 dprintINT

```
#define dprintINT(
            expr ) printf(#expr " = %d\n", expr)
```

Definition at line 26 of file display.c.

4.11.1.4 dprintSIZE_T

```
#define dprintSIZE_T(
            expr ) printf(#expr " = %zu\n", expr)
```

Definition at line 28 of file display.c.

4.11.1.5 dprintSTRING

```
#define dprintSTRING(
            expr ) printf(#expr " = %s\n", expr)
```

Definition at line 24 of file display.c.

4.11.1.6 FPS

```
#define FPS 100
```

Definition at line 17 of file display.c.

4.11.1.7 FRAME_TARGET_TIME

```
#define FRAME_TARGET_TIME (1000 / FPS)
```

Definition at line 21 of file display.c.

4.11.1.8 WINDOW_HEIGHT

```
#define WINDOW_HEIGHT (9 * 80)
```

Definition at line 13 of file display.c.

4.11.1.9 WINDOW_WIDTH

```
#define WINDOW_WIDTH (16 * 80)
```

Definition at line 9 of file display.c.

4.11.2 Function Documentation

4.11.2.1 check_window_mat_size()

```
void check_window_mat_size (
            game_state_t * game_state )
```

Definition at line 361 of file display.c.

References Camera::aspect_ratio, Scene::camera, 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::scene, 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().

4.11.2.2 copy_mat_to_surface_RGB()

```
void copy_mat_to_surface_RGB (
            game_state_t * game_state )
```

Definition at line 376 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().

4.11.2.3 destroy_window()

```
void destroy_window (
            game_state_t * game_state )
```

Definition at line 316 of file display.c.

References ae_scene_free(), game_state_t::inv_z_buffer_mat, mat2D_free(), mat2D_free_uint32(), game_state_t::renderer, game_state_t::scene, game_state_t::window, game_state_t::window_pixels_mat, game_state_t::window_surface, and game_state_t::window_texture.

Referenced by main().

4.11.2.4 fix_framerate()

```
void fix_framerate (
            game_state_t * game_state )
```

Definition at line 333 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().

4.11.2.5 initialize_window()

```
int initialize_window (
            game_state_t * game_state )
```

Definition at line 137 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().

4.11.2.6 main()

```
int main ( )
```

Definition at line 88 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, 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::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, and WINDOW_WIDTH.

4.11.2.7 process_input_window()

```
void process_input_window (
            game_state_t * game_state )
```

Definition at line 186 of file display.c.

References ae_camera_reset_pos(), Scene::camera, game_state_t::game_is_running, game_state_t::left_button_pressed, MAT2D_AT, Camera::offset_position, Camera::pitch_offset_deg, game_state_t::previous_frame_time, Camera::roll_offset_deg, game_state_t::scene, game_state_t::space_bar_was_pressed, game_state_t::to_render, and game_state_t::to_update.

Referenced by main().

4.11.2.8 render()

```
void render (
            game_state_t * game_state )
```

Definition at line 32 of file grid_example.c.

References ADL_DEFAULT_OFFSET_ZOOM, adl_grid_draw(), adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_normal(), Tri_mesh_array::elements, grid_proj, Scene::in_world_tri_meshes, game_state_t::inv_z_buffer_mat, Tri_mesh::length, Tri_mesh_array::length, Scene::projected_tri_meshes, game_state_t::scene, and game_state_t::window_pixels_mat.

Referenced by render_window().

4.11.2.9 render_window()

```
void render_window (
            game_state_t * game_state )
```

Definition at line 295 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().

4.11.2.10 setup()

```
void setup (
            game_state_t * game_state )
```

Definition at line 14 of file grid_example.c.

References ada_append, ada_init_array, adl_cartesian_grid_create(), ae_tri_mesh_append_copy(), ae_tri_mesh_get_from_file(), ae_tri_mesh_normalize(), ae_tri_mesh_rotate_Euler_xyz(), ASM_MAX_LEN_LINE, game_state_t::const_fps, Tri_mesh_array::elements, grid, grid_proj, Scene::in_world_tri_meshes, Tri_mesh::length, Tri_mesh_array::length, Scene::original_tri_meshes, Scene::projected_tri_meshes, game_state_t::scene, and game_state_t::to_limit_fps.

Referenced by setup_window().

4.11.2.11 setup_window()

```
void setup_window (
            game_state_t * game_state )
```

Definition at line 170 of file display.c.

References ae_scene_init(), game_state_t::inv_z_buffer_mat, mat2D_alloc(), mat2D_alloc_uint32(), game_state_t::scene, 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().

4.11.2.12 update()

```
void update (
            game_state_t * game_state )
```

Definition at line 23 of file grid_example.c.

References ae_grid_project_world2screen(), AE_LIGHTING_FLAT, ae_projection_mat_set(), ae_tri_mesh_project_world2screen(), ae_view_mat_set(), Camera::aspect_ratio, Scene::camera, Tri_mesh_array::elements, Camera::fov_deg, grid, grid_proj, Scene::in_world_tri_meshes, Tri_mesh_array::length, Scene::proj_mat, Scene::projected_tri_meshes, game_state_t::scene, Scene::up_direction, Scene::view_mat, game_state_t::window_h, game_state_t::window_w, Camera::z_far, and Camera::z_near.

Referenced by update_window().

4.11.2.13 update_window()

```
void update_window (
            game_state_t * game_state )
```

Definition at line 267 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().

4.12 display.c

```c
#include <SDL2/SDL.h>
#include <SDL2/SDL_ttf.h>
#include "Matrix2D.h"
#include <stdlib.h>
#include <stdint.h>
#include "Almog_Engine.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;

    SDL_Surface *window_surface;
    SDL_Texture *window_texture;

    Mat2D_uint32 window_pixels_mat;
    Mat2D inv_z_buffer_mat;

    Scene scene;
} 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.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;
    }

    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);

    game_state->scene = ae_scene_init(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) {
                    MAT2D_AT(game_state->scene.camera.offset_position, 1, 0) -= 0.05;
                }
                if (event.key.keysym.sym == SDLK_s) {
                    MAT2D_AT(game_state->scene.camera.offset_position, 1, 0) += 0.05;
                }
                if (event.key.keysym.sym == SDLK_d) {
                    MAT2D_AT(game_state->scene.camera.offset_position, 0, 0) += 0.05;
                }
                if (event.key.keysym.sym == SDLK_a) {
                    MAT2D_AT(game_state->scene.camera.offset_position, 0, 0) -= 0.05;
                }
                if (event.key.keysym.sym == SDLK_e) {
                    MAT2D_AT(game_state->scene.camera.offset_position, 2, 0) += 0.05;
                }
                if (event.key.keysym.sym == SDLK_q) {
                    MAT2D_AT(game_state->scene.camera.offset_position, 2, 0) -= 0.05;
                }
                if (event.key.keysym.sym == SDLK_LEFT) {
                    game_state->scene.camera.pitch_offset_deg -= 1;
                }
                if (event.key.keysym.sym == SDLK_RIGHT) {
                    game_state->scene.camera.pitch_offset_deg += 1;
                }
                if (event.key.keysym.sym == SDLK_UP) {
                    game_state->scene.camera.roll_offset_deg += 1;
                    if (game_state->scene.camera.roll_offset_deg > 89) {
                        game_state->scene.camera.roll_offset_deg = 89;
                    }
                }
                if (event.key.keysym.sym == SDLK_DOWN) {
                    game_state->scene.camera.roll_offset_deg -= 1;
                    if (game_state->scene.camera.roll_offset_deg < -89) {
                        game_state->scene.camera.roll_offset_deg = -89;
                    }
                }
                if (event.key.keysym.sym == SDLK_r) {
                    ae_camera_reset_pos(&(game_state->scene));
                }
                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);
        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);
    mat2D_free(game_state->inv_z_buffer_mat);
    ae_scene_free(&(game_state->scene));

    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();
```

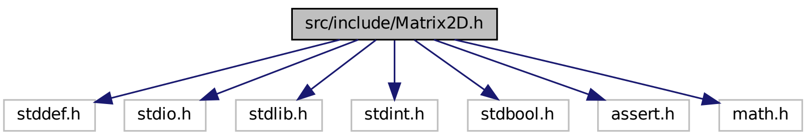
```
}

#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->scene.camera.aspect_ratio = (float)(game_state->window_h) / (float)(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);
}


```

4.13 src/include/Matrix2D.h File Reference

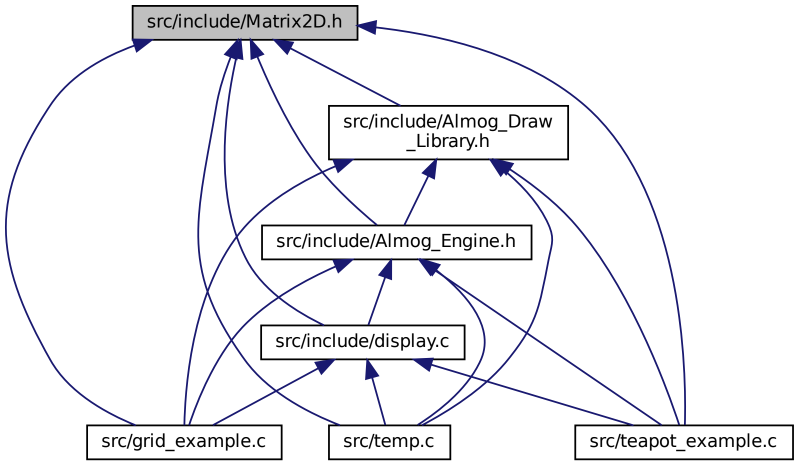
A single-header C library for simple 2D matrix operations on doubles and uint32_t, including allocation, basic arithmetic, linear algebra, and helpers (LUP, inverse, determinant, DCM, etc.).

```
#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.

4.13.1 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.

4.13.2 Macro Definition Documentation

4.13.2.1 __USE_MISC

```
#define __USE_MISC
```

Definition at line 151 of file Matrix2D.h.

4.13.2.2 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.

4.13.2.3 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.

4.13.2.4 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.

4.13.2.5 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.

4.13.2.6 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.

4.13.2.7 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.

4.13.2.8 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.

4.13.2.9 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.

4.13.2.10 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.

4.13.2.11 PI

```
#define PI M_PI
```

Definition at line 154 of file Matrix2D.h.

4.13.3 Function Documentation

4.13.3.1 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(), ae_line_itersect_plane(), and ae_view_mat_set().

4.13.3.2 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.

4.13.3.3 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.

4.13.3.4 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.

4.13.3.5 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(), ae_camera_init(), ae_curve_project_world2screen(), ae_line_clip_with_plane(), ae_line_itersect_plane(), ae_line_project_world2screen(), ae_point_project_view2screen(), ae_point_project_world2view(), ae_quad_calc_normal(), ae_quad_clip_with_plane(), ae_quad_mesh_project_world2screen(), ae_quad_project_world2screen(), ae_quad_set_normals(), ae_quad_transform_to_view(), ae_scene_init(), ae_tri_calc_normal(), ae_tri_clip_with_plane(), ae_tri_mesh_project_world2screen(), ae_tri_mesh_rotate_Euler_xyz(), ae_tri_project_world2screen(), ae_tri_set_normals(), ae_tri_transform_to_view(), ae_view_mat_set(), check_window_mat_size(), mat2D_det(), mat2D_invert(), mat2D_set_DCM_zyx(), mat2D_solve_linear_sys_LUP_decomposition(), and setup_window().

4.13.3.6 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().

4.13.3.7 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.

Referenced by ae_quad_calc_normal(), ae_tri_calc_normal(), and ae_view_mat_set().

4.13.3.8 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().

4.13.3.9 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(), ae_camera_init(), ae_camera_reset_pos(), ae_quad_set_normals(), ae_tri_set_normals(), ae_view_mat_set(), mat2D_det(), mat2D_invert(), and mat2D_LUP_decomposition_with_swap().

4.13.3.10 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.

4.13.3.11 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.

Referenced by ae_quad_calc_normal(), ae_quad_set_normals(), ae_tri_calc_normal(), ae_tri_set_normals(), and ae_view_mat_set().

4.13.3.12 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().

4.13.3.13 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.

4.13.3.14 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().

4.13.3.15 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(), ae_point_project_view2screen(), ae_point_project_world2view(), ae_quad_transform_to_view(), ae_tri_mesh_rotate_Euler_xyz(), ae_tri_transform_to_view(), ae_view_mat_set(), mat2D_set_DCM_zyx(), and mat2D_solve_linear_sys_LUP_decomposition().

4.13.3.16 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.

Referenced by ae_line_itersect_plane(), and ae_view_mat_set().

4.13.3.17 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(), ae_camera_init(), ae_camera_reset_pos(), ae_curve_project_world2screen(), ae_line_itersect_plane(), ae_line_project_world2screen(), ae_projection_mat_set(), ae_quad_mesh_project_world2screen(), ae_quad_project_world2screen(), ae_scene_init(), ae_tri_mesh_project_world2screen(), ae_tri_project_world2screen(), ae_view_mat_set(), mat2D_invert(), mat2D_LUP_decomposition_with_swap(), and mat2D_solve_linear_sys_LUP_decomposition().

4.13.3.18 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.

4.13.3.19 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().

4.13.3.20 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(), ae_camera_free(), ae_curve_project_world2screen(), ae_line_clip_with_plane(), ae_line_itersect_plane(), ae_line_project_world2screen(), ae_point_project_view2screen(), ae_point_project_world2view(), ae_quad_calc_normal(), ae_quad_clip_with_plane(), ae_quad_mesh_project_world2screen(), ae_quad_project_world2screen(), ae_quad_set_normals(), ae_quad_transform_to_view(), ae_scene_free(), ae_tri_calc_normal(), ae_tri_clip_with_plane(), ae_tri_mesh_project_world2screen(), ae_tri_mesh_rotate_Euler_xyz(), ae_tri_project_world2screen(), ae_tri_set_normals(), ae_tri_transform_to_view(), ae_view_mat_set(), check_window_mat_size(), destroy_window(), mat2D_det(), mat2D_invert(), mat2D_set_DCM_zyx(), and mat2D_solve_linear_sys_LUP_decomposition().

4.13.3.21 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().

4.13.3.22 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.

4.13.3.23 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.

4.13.3.24 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().

4.13.3.25 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().

4.13.3.26 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.

4.13.3.27 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.

4.13.3.28 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.

4.13.3.29 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().

4.13.3.30 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.

4.13.3.31 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().

4.13.3.32 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.

4.13.3.33 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.

Referenced by ae_line_itersect_plane(), ae_quad_calc_normal(), ae_tri_calc_normal(), and ae_view_mat_set().

4.13.3.34 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().

4.13.3.35 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().

4.13.3.36 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.

4.13.3.37 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.

4.13.3.38 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.

4.13.3.39 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.

4.13.3.40 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().

4.13.3.41 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().

4.13.3.42 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().

Referenced by ae_view_mat_set().

4.13.3.43 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().

4.13.3.44 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 ae_tri_mesh_rotate_Euler_xyz(), and mat2D_set_DCM_zyx().

4.13.3.45 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 ae_tri_mesh_rotate_Euler_xyz(), and mat2D_set_DCM_zyx().

4.13.3.46 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(), ae_tri_mesh_rotate_Euler_xyz(), and mat2D_set_DCM_zyx().

4.13.3.47 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.

4.13.3.48 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(), ae_line_itersect_plane(), ae_quad_calc_normal(), ae_quad_project_world2screen(), ae_quad_set_normals(), ae_tri_calc_normal(), ae_tri_project_world2screen(), ae_tri_set_normals(), and ae_view_mat_set().

4.13.3.49 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.

4.13.3.50 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().

4.13.3.51 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.

4.13.3.52 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().

4.13.3.53 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.

Referenced by ae_tri_project_world2screen(), and ae_view_mat_set().

4.13.3.54 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().

4.14 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

#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);
```

```c
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);
        }
    }
}

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;
}
```

```c

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);
```

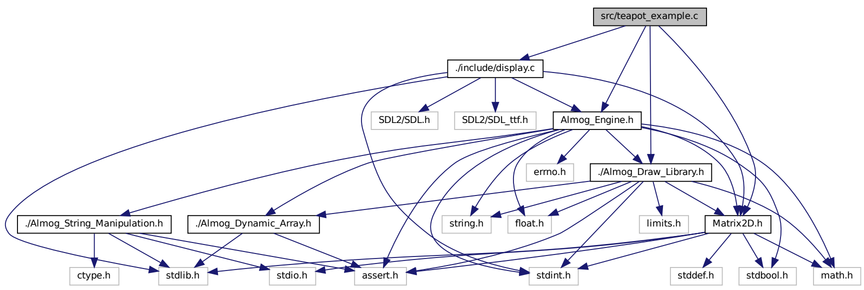
```

    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
```

4.15 src/teapot_example.c File Reference

```
#include "./include/display.c"
#include "./include/Matrix2D.h"
#include "./include/Almog_Draw_Library.h"
#include "./include/Almog_Engine.h"
```

Include dependency graph for teapot_example.c:



Macros

- #define SETUP
- #define UPDATE

- #define RENDER
- #define MATRIX2D_IMPLEMENTATION
- #define ALMOG_DRAW_LIBRARY_IMPLEMENTATION
- #define ALMOG_ENGINE_IMPLEMENTATION

Functions

- void setup (game_state_t *game_state)
- void update (game_state_t *game_state)
- void render (game_state_t *game_state)

4.15.1 Macro Definition Documentation

4.15.1.1 ALMOG_DRAW_LIBRARY_IMPLEMENTATION

```
#define ALMOG_DRAW_LIBRARY_IMPLEMENTATION
```

Definition at line 7 of file teapot_example.c.

4.15.1.2 ALMOG_ENGINE_IMPLEMENTATION

```
#define ALMOG_ENGINE_IMPLEMENTATION
```

Definition at line 9 of file teapot_example.c.

4.15.1.3 MATRIX2D_IMPLEMENTATION

```
#define MATRIX2D_IMPLEMENTATION
```

Definition at line 5 of file teapot_example.c.

4.15.1.4 RENDER

```
#define RENDER
```

Definition at line 3 of file teapot_example.c.

4.15.1.5 SETUP

```
#define SETUP
```

Definition at line 1 of file teapot_example.c.

4.15.1.6 UPDATE

```
#define UPDATE
```

Definition at line 2 of file teapot_example.c.

4.15.2 Function Documentation

4.15.2.1 render()

```
void render (
            game_state_t * game_state )
```

Definition at line 61 of file teapot_example.c.

References ADL_DEFAULT_OFFSET_ZOOM, adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_normal(), Tri_mesh_array::elements, Scene::in_world_tri_meshes, game_state_t::inv_z_buffer_mat, Tri_mesh::length, Tri_mesh_array::length, Scene::projected_tri_meshes, game_state_t::scene, and game_state_t::window_pixels_mat.

4.15.2.2 setup()

```
void setup (
            game_state_t * game_state )
```

Definition at line 12 of file teapot_example.c.

References ada_append, ada_init_array, ae_tri_mesh_append_copy(), ae_tri_mesh_get_from_file(), ae_tri_mesh_normalize(), ae_tri_mesh_rotate_Euler_xyz(), Tri_mesh_array::elements, Scene::in_world_tri_meshes, Tri_mesh::length, Tri_mesh_array::length, Scene::original_tri_meshes, Scene::projected_tri_meshes, game_state_t::scene, and game_state_t::to_limit_fps.

4.15.2.3 update()

```
void update (
              game_state_t * game_state )
```

Definition at line 50 of file teapot_example.c.

References AE_LIGHTING_FLAT, ae_projection_mat_set(), ae_tri_mesh_project_world2screen(), ae_view_mat_set(), Camera::aspect_ratio, Scene::camera, Tri_mesh_array::elements, Camera::fov_deg, Scene::in_world_tri_meshes, Tri_mesh_array::length, Scene::proj_mat, Scene::projected_tri_meshes, game_state_t::scene, Scene::up_direction, Scene::view_mat, game_state_t::window_h, game_state_t::window_w, Camera::z_far, and Camera::z_near.

4.16 teapot_example.c

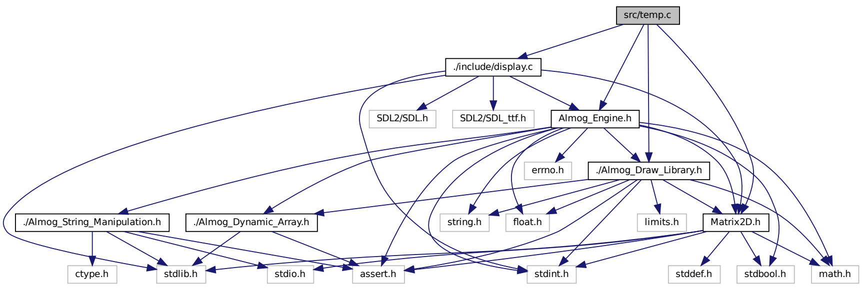
```
00001 #define SETUP
00002 #define UPDATE
00003 #define RENDER
00004 #include "./include/display.c"
00005 #define MATRIX2D_IMPLEMENTATION
00006 #include "./include/Matrix2D.h"
00007 #define ALMOG_DRAW_LIBRARY_IMPLEMENTATION
00008 #include "./include/Almog_Draw_Library.h"
00009 #define ALMOG_ENGINE_IMPLEMENTATION
00010 #include "./include/Almog_Engine.h"
00011
00012 void setup(game_state_t *game_state)
00013 {
00014     game_state->to_limit_fps = 0;
00015
00016     ada_init_array(Tri_mesh, game_state->scene.original_tri_meshes);
00017     ada_init_array(Tri_mesh, game_state->scene.in_world_tri_meshes);
00018     ada_init_array(Tri_mesh, game_state->scene.projected_tri_meshes);
00019
00020     char file_path[MAX_LEN_LINE];
00021     strncpy(file_path, "./teapot.stl", MAX_LEN_LINE);
00022
00023     Tri_mesh teapot_mesh = ae_tri_mesh_get_from_file(file_path);
00024     // ae_tri_mesh_flip_normals(teapot_mesh);
00025     ada_append(Tri_mesh, game_state->scene.original_tri_meshes, teapot_mesh);
00026
00027     printf("[INFO] number of meshes: %zu\n", game_state->scene.original_tri_meshes.length);
00028     size_t sum = 0;
00029     for (size_t i = 0; i < game_state->scene.original_tri_meshes.length; i++) {
00030         printf("[INFO] mesh number %zu: %zu\n", i,
       game_state->scene.original_tri_meshes.elements[i].length);
00031         sum += game_state->scene.original_tri_meshes.elements[i].length;
00032     }
00033     printf("[INFO] total number of triangles: %zu\n", sum);
00034
00035
00036     for (size_t i = 0; i < game_state->scene.original_tri_meshes.length; i++) {
00037         ae_tri_mesh_normalize(game_state->scene.original_tri_meshes.elements[i]);
00038     }
00039     for (size_t i = 0; i < game_state->scene.original_tri_meshes.length; i++) {
00040         ae_tri_mesh_append_copy(&(game_state->scene.in_world_tri_meshes),
       game_state->scene.original_tri_meshes.elements[i]);
00041         ae_tri_mesh_append_copy(&(game_state->scene.projected_tri_meshes),
       game_state->scene.original_tri_meshes.elements[i]);
00042         game_state->scene.projected_tri_meshes.elements[i].length = 0;
00043     }
00044
00045     ae_tri_mesh_rotate_Euler_xyz(game_state->scene.in_world_tri_meshes.elements[0], -90, 0, 180);
00046
00047     // ae_translate_mesh(game_state->scene.in_world_tri_meshes.elements[0], 0, 0, 2);
00048 }
00049
00050 void update(game_state_t *game_state)
00051 {
00052     ae_projection_mat_set(game_state->scene.proj_mat, game_state->scene.camera.aspect_ratio,
       game_state->scene.camera.fov_deg, game_state->scene.camera.z_near, game_state->scene.camera.z_far);
00053     ae_view_mat_set(game_state->scene.view_mat, game_state->scene.camera,
       game_state->scene.up_direction);
00054
00055     for (size_t i = 0; i < game_state->scene.in_world_tri_meshes.length; i++) {
00056         ae_tri_mesh_project_world2screen(game_state->scene.proj_mat, game_state->scene.view_mat,
       &(game_state->scene.projected_tri_meshes.elements[i]),
       game_state->scene.in_world_tri_meshes.elements[i], game_state->window_w, game_state->window_h,
       &(game_state->scene), AE_LIGHTING_FLAT);
```

```
    }

}

void render(game_state_t *game_state)
{
    for (size_t i = 0; i < game_state->scene.projected_tri_meshes.length; i++) {
        adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_normal(game_state->window_pixels_mat,
     game_state->inv_z_buffer_mat, game_state->scene.projected_tri_meshes.elements[i], 0xffffffff,
     ADL_DEFAULT_OFFSET_ZOOM);
    }

    for (size_t i = 0; i < game_state->scene.in_world_tri_meshes.length; i++) {
        game_state->scene.projected_tri_meshes.elements[i].length = 0;
    }
}
```

4.17 src/temp.c File Reference

```
#include "./include/display.c"
#include "./include/Matrix2D.h"
#include "./include/Almog_Draw_Library.h"
#include "./include/Almog_Engine.h"
```

Include dependency graph for temp.c:



Macros

- #define SETUP
- #define UPDATE
- #define RENDER
- #define MATRIX2D_IMPLEMENTATION
- #define ALMOG_DRAW_LIBRARY_IMPLEMENTATION
- #define ALMOG_ENGINE_IMPLEMENTATION

Functions

- void setup (game_state_t *game_state)
- void update (game_state_t *game_state)
- void render (game_state_t *game_state)

4.17.1 Macro Definition Documentation

4.17.1.1 ALMOG_DRAW_LIBRARY_IMPLEMENTATION

```
#define ALMOG_DRAW_LIBRARY_IMPLEMENTATION
```

Definition at line 7 of file temp.c.

4.17.1.2 ALMOG_ENGINE_IMPLEMENTATION

```
#define ALMOG_ENGINE_IMPLEMENTATION
```

Definition at line 9 of file temp.c.

4.17.1.3 MATRIX2D_IMPLEMENTATION

```
#define MATRIX2D_IMPLEMENTATION
```

Definition at line 5 of file temp.c.

4.17.1.4 RENDER

```
#define RENDER
```

Definition at line 3 of file temp.c.

4.17.1.5 SETUP

```
#define SETUP
```

Definition at line 1 of file temp.c.

4.17.1.6 UPDATE

```
#define UPDATE
```

Definition at line 2 of file temp.c.

4.17.2 Function Documentation

4.17.2.1 render()

```
void render (
            game_state_t * game_state )
```

Definition at line 63 of file temp.c.

References ADL_DEFAULT_OFFSET_ZOOM, adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_normal(), Tri_mesh_array::elements, game_state_t::inv_z_buffer_mat, Tri_mesh_array::length, Scene::projected_tri_meshes, game_state_t::scene, and game_state_t::window_pixels_mat.

Referenced by render_window().

4.17.2.2 setup()

```
void setup (
            game_state_t * game_state )
```

Definition at line 12 of file temp.c.

References ada_append, ada_init_array, ae_tri_mesh_append_copy(), ae_tri_mesh_get_from_file(), ae_tri_mesh_normalize(), ae_tri_mesh_rotate_Euler_xyz(), ASM_MAX_LEN_LINE, Tri_mesh_array::elements, Scene::in_world_tri_meshes, Tri_mesh::length, Tri_mesh_array::length, Scene::original_tri_meshes, Scene::projected_tri_meshes, game_state_t::scene, and game_state_t::to_limit_fps.

Referenced by setup_window().

4.17.2.3 update()

```
void update (
            game_state_t * game_state )
```

Definition at line 49 of file temp.c.

References AE_LIGHTING_FLAT, ae_projection_mat_set(), ae_tri_mesh_project_world2screen(), ae_view_mat_set(), Camera::aspect_ratio, Scene::camera, Tri_mesh_array::elements, Camera::fov_deg, Scene::in_world_tri_meshes, Tri_mesh_array::length, Scene::proj_mat, Scene::projected_tri_meshes, game_state_t::scene, Scene::up_direction, Scene::view_mat, game_state_t::window_h, game_state_t::window_w, Camera::z_far, and Camera::z_near.

Referenced by update_window().

4.18 temp.c

```
#define SETUP
#define UPDATE
#define RENDER
#include "./include/display.c"
#define MATRIX2D_IMPLEMENTATION
#include "./include/Matrix2D.h"
#define ALMOG_DRAW_LIBRARY_IMPLEMENTATION
#include "./include/Almog_Draw_Library.h"
#define ALMOG_ENGINE_IMPLEMENTATION
#include "./include/Almog_Engine.h"

void setup(game_state_t *game_state)
{
    game_state->to_limit_fps = 0;

    ada_init_array(Tri_mesh, game_state->scene.original_tri_meshes);
    ada_init_array(Tri_mesh, game_state->scene.in_world_tri_meshes);
    ada_init_array(Tri_mesh, game_state->scene.projected_tri_meshes);

    char file_path[ASM_MAX_LEN_LINE];
    strncpy(file_path, "./teapot.stl", ASM_MAX_LEN_LINE);

    Tri_mesh tri_mesh = ae_tri_mesh_get_from_file(file_path);

    ada_append(Tri_mesh, game_state->scene.original_tri_meshes, tri_mesh);

    printf("[INFO] number of meshes: %zu\n", game_state->scene.original_tri_meshes.length);
    size_t sum = 0;
    for (size_t i = 0; i < game_state->scene.original_tri_meshes.length; i++) {
        printf("[INFO] mesh number %zu: %zu\n", i,
  game_state->scene.original_tri_meshes.elements[i].length);
        sum += game_state->scene.original_tri_meshes.elements[i].length;
    }
    printf("[INFO] total number of triangles: %zu\n", sum);


    for (size_t i = 0; i < game_state->scene.original_tri_meshes.length; i++) {
        ae_tri_mesh_normalize(game_state->scene.original_tri_meshes.elements[i]);
    }
    for (size_t i = 0; i < game_state->scene.original_tri_meshes.length; i++) {
        ae_tri_mesh_append_copy(&(game_state->scene.in_world_tri_meshes),
  game_state->scene.original_tri_meshes.elements[i]);
        ae_tri_mesh_append_copy(&(game_state->scene.projected_tri_meshes),
  game_state->scene.original_tri_meshes.elements[i]);
        game_state->scene.projected_tri_meshes.elements[i].length = 0;
    }

    ae_tri_mesh_rotate_Euler_xyz(game_state->scene.in_world_tri_meshes.elements[0], -90, 0, 180);

}

void update(game_state_t *game_state)
{
    // MAT2D_PRINT(game_state->scene.camera.current_position);
    // MAT2D_PRINT(game_state->scene.light_direction);

    ae_projection_mat_set(game_state->scene.proj_mat, game_state->scene.camera.aspect_ratio,
  game_state->scene.camera.fov_deg, game_state->scene.camera.z_near, game_state->scene.camera.z_far);
    ae_view_mat_set(game_state->scene.view_mat, game_state->scene.camera,
  game_state->scene.up_direction);

    for (size_t i = 0; i < game_state->scene.in_world_tri_meshes.length; i++) {
        ae_tri_mesh_project_world2screen(game_state->scene.proj_mat, game_state->scene.view_mat,
  &(game_state->scene.projected_tri_meshes.elements[i]),
  game_state->scene.in_world_tri_meshes.elements[i], game_state->window_w, game_state->window_h,
  &(game_state->scene), AE_LIGHTING_FLAT);
    }

}

void render(game_state_t *game_state)
{
    for (size_t i = 0; i < game_state->scene.projected_tri_meshes.length; i++) {
        adl_tri_mesh_fill_Pinedas_rasterizer_interpolate_normal(game_state->window_pixels_mat,
  game_state->inv_z_buffer_mat, game_state->scene.projected_tri_meshes.elements[i], 0xffffffff,
  ADL_DEFAULT_OFFSET_ZOOM);
    }
}
```

Index