Software Requirements Specification
for
The Walking Game

Star Team

Version: 1.0
Date: 10/31/2014

Revision History

Revision #	Date	Description	Author
1.0	10/31/2014	Initial revision	Samuel I. Gunadi

1 Introduction

The members of Star team are Samuel I. Gunadi, Roberto J. Kondurura, and Ryan Elegant. Star team has a goal to fulfill the requirements of the software engineering course at Universitas Pelita Harapan. The project website can be found at `https://github.com/samuelgunadi/thewalkinggame`.

1.1 Purpose

This specification establishes the functional, performance, and development requirements for The Walking Game project. This document is intended to aid by Star Team in the development process and also by

1.2 Scope

The project will have a Wavefront OBJ file parser; people 3D models, textures, and walking animations; textures with alpha blending; a textured floor; collision detection; simple command interpreter and scripting; nameplates; and an MMORPG style third-person camera.

The animation technique used in The Walking Games will not be skeletal animation; instead, it will be by loading one .obj for every key frame.

This project aims to educate and entertain users—educate, by studying the source code, and entertain, by playing and interacting through the console. This project will mostly benefits the team as part of their Computer Graphics and Software Engineering course assignments.

1.3 Definitions, acronyms, and abbreviations

SRS — Software Requirements Specification

OpenGL — Open Graphics Library

OBJ — Wavefront OBJ File Format

GLEW — The OpenGL Extension Wrangler Library

PDF — Portable Document Format

IDE — Integrated Development Environment

MMORPG — Massively Multiplayer Online Role-Playing Game

1.4 References

[1] "IEEE Recommended Practice for Software Requirements Specifications," *IEEE Std 830-1998*, pp. 1–40, Oct 1998.

[2] "IEEE Standard for Configuration Management in Systems and Software Engineering," *IEEE Std 828-2012 (Revision of IEEE Std 828-2005)*, pp. 1–71, March 2012.

[3] "Systems and software engineering – Life cycle processes–Requirements engineering," *ISO/IEC/IEEE 29148:2011(E)*, pp. 1–94, Dec 2011.

[4] P. Tripathy and K. Naik, *Software Evolution and Maintenance: A Practitioner's Approach*. Hoboken, New Jersey: John Wiley & Sons, 2014.

[5] K. Naik and P. Tripathy, *Software Testing and Quality Assurance: Theory and Practice*. Hoboken, New Jersey: John Wiley & Sons, 2008.

[6] I. Sommerville, *Software Engineering*, 9th ed. Boston, Massachusetts: Addison-Wesley, 2010.

[7] R. Pressman and B. Maxim, *Software Engineering: A Practitioner's Approach*, 8th ed. New York, New York: McGraw-Hill Education, 2014.

1.5 Overview

Section 1 identifies the scope of this document, the purpose of this document, lists the definitions, acronyms and references, and explains how the SRS is organized. Section 2 describes the general factors that affect the product and its requirements. This section does not state specific requirements. Instead, it provides a background for those requirements, which are defined in detail in Section 3 of the SRS, and makes them easier to understand. Section 3 contains all of the software requirements to a level of detail sufficient to enable designers to design a system to satisfy those requirements, and testers to test that the system satisfies those requirements.

2 Overall description

2.1 Product perspective

2.1.1 User interfaces

The Walking Game includes an interface resembling a RPG game with a console to provide additional functionalities and aid in testing and debugging process.

2.1.2 Hardware interfaces

The Walking Game should run on any computer hardware meeting the following criteria:

- Dual-core CPU
- OpenGL 4.3 compatible GPU with 2 GB memory
- 2 GB free hard disk space.
- Mouse with 3 buttons
- Keyboard with US layout
- 2 GB of RAM

2.1.3 Software interfaces

The Walking Game integrates several external software to provide functionality:

1. GLFW. GLFW is used for creating windows with OpenGL contexts and receiving input and events. GLFW is multi-platform and supports Windows, OS X and many Unix-like systems. GLFW is licensed under the zlib/libpng license.
2. GLEW. GLEW is a cross-platform open-source C/C++ extension loading library. GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform. OpenGL core and extension functionality is exposed in a single header file. GLEW has been tested on a variety of operating systems, including Windows, Linux, Mac OS X, FreeBSD, Irix, and Solaris.
3. GLM. GLM is a header only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specification and released under the MIT license.

2.2 Product functions

These are the main functional areas:

- **OBJ file parser.** OBJ files are generated by content creation tools and then parsed and loaded into GPU memory.
- **Texture loader.** Image files are generated by content creation tools and then parsed and loaded into GPU memory.
- **Character management.** The scene includes characters that are rendered with animations, textures, and geometry data, then the characters are labeled with nameplates and animated.
- **Command interpreter.** The console commands are inputted through the console and appropriate actions are executed.
- **Third-person camera.** The camera enables the user to view the scenes.

2.3 User characteristics

No special knowledge or skills should be assumed on the part of the users. Although the user is expected to have experience in C/C++ to interact with the console.

2.4 Constraints

The Walking Game may experience hardware limitations constrain for graphics requirements if installed in an incompatible computer.

2.5 Assumptions and Dependencies

The Walking Game is written with portability in mind and should run on Windows, OS X, and Linux.

3 Specific requirements

3.1 External interface requirements

3.1.1 User interfaces

All interaction with the user is done via console and main window.

3.1.2 Hardware Interfaces

User interacts using computer display, mouse, and keyboard.

3.1.3 Software interfaces

The system should be capable of running on Windows, OS X, and Linux.

3.2 Classes

3.2.1 Character

This class shall render a character and update its animation when it's walking.

3.2.2 Nameplate

This class shall label a character and render a text above the character.

3.2.3 Texture

This class shall aid managing textures.

3.2.4 Mesh

This class shall be capable of importing OBJ and uploading the geometrical data into the GPU. This class should be able to export and import the geometrical data into binary format to speed up the loading process.

3.2.5 Plane

This class shall represent a geometric plane.

3.2.6 Floor

This class shall render a textured plane.

3.2.7 Frustum

This class shall represent a frustum used by the camera.

3.2.8 Camera

This class shall provide camera functionalities to a scene.

3.2.9 Third-person camera

This class shall add the camera functionalities to follow a character.

3.2.10 Scene

This class shall abstract the scene.

3.2.11 Main scene

This class shall encapsulate the objects to be rendered.

3.2.12 Main event handler

This class shall handle keyboard and mouse inputs from user.

3.2.13 Shader program

This class shall encapsulate an OpenGL shader program.

3.2.14 Light

This class shall represent a light.

3.2.15 Material

This class shall represent a Phong shader material.

3.2.16 Thread pool

This class shall enable asynchronous tasks to be executed concurrently.

3.3 Performance requirements

No performance requirements have been specified by the clients.

3.4 Design constraints

No design constraints have been identified.

3.5 Software system attributes

3.5.1 Reliability

1. system should handle errors when processing console commands.
2. The system should not crash during initialization and running.

3.5.2 Availability

1. The system should recover corrupt files.

3.5.3 Security

1. The system should provide logs to identify errors.

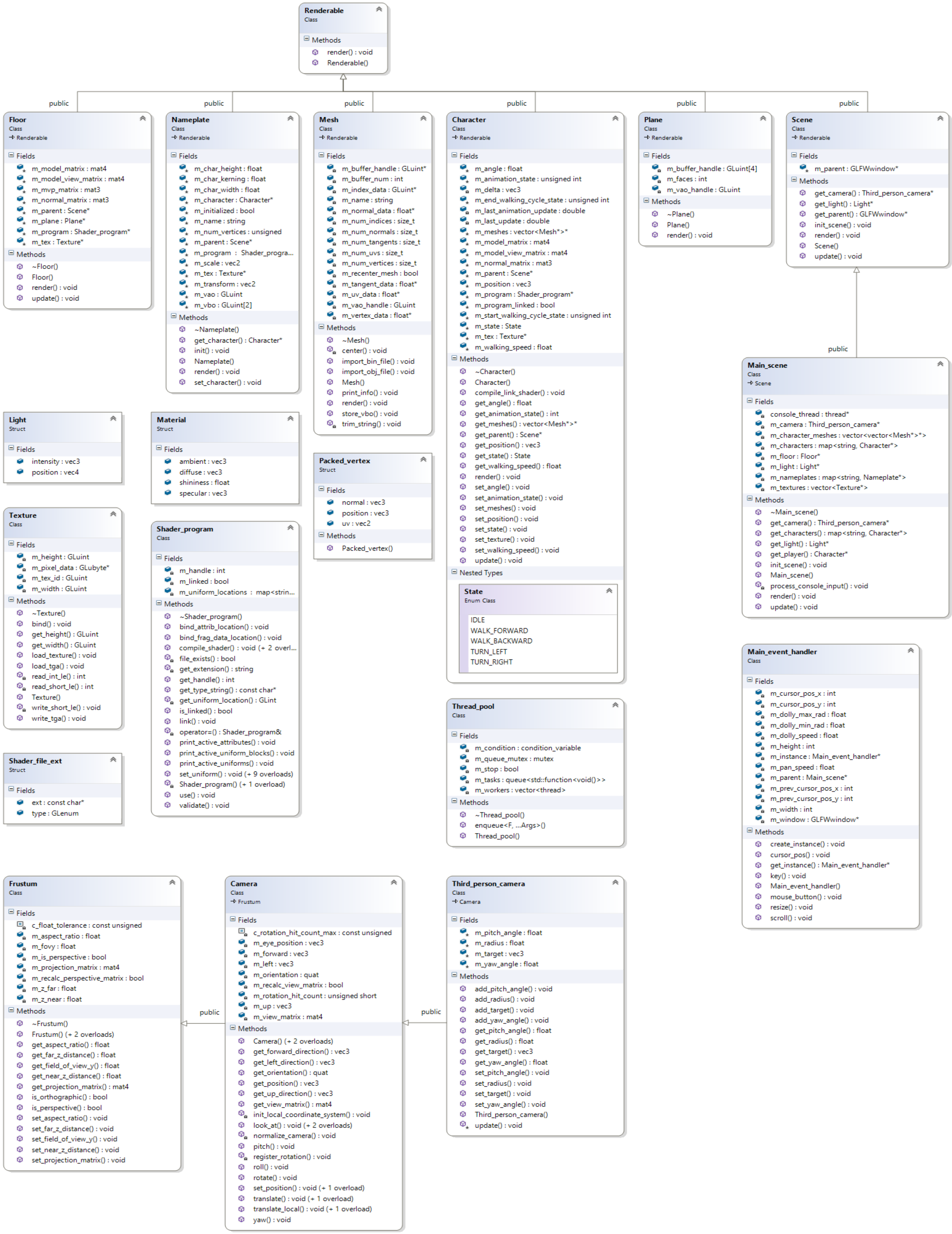
3.6 Other requirements

No other requirements have been identified.

4 Supporting information

4.1 Table of contents

4.2 Appendix A: Class diagram


Renderable
Class
Methods
render() : void
Renderable()
public
public
public
public
public
public
Floor
Class
Renderable
Fields
m_model_matrix : mat4
m_model_view_matrix : mat4
m_mvp_matrix : mat3
m_normal_matrix : mat3
m_parent : Scene*
m_plane : Plane*
m_program : Shader_program*
m_tex : Texture*
Methods
~Floor()
Floor()
render() : void
update() : void
Nameplate
Class
Renderable
Fields
m_char_height : float
m_char_kerning : float
m_char_width : float
m_character : Character*
m_initialized : bool
m_name : string
m_num_vertices : unsigned
m_parent : Scene*
m_program : Shader_progra...
m_scale : vec2
m_tex : Texture*
m_transform : vec2
m_vao : GLuint
m_vbo : GLuint[2]
Methods
~Nameplate()
get_character() : Character*
init() : void
Nameplate()
render() : void
set_character() : void
Mesh
Class
Renderable
Fields
m_buffer_handle : GLuint*
m_buffer_num : int
m_index_data : GLuint*
m_name : string
m_normal_data : float*
m_num_indices : size_t
m_num_normals : size_t
m_num_tangents : size_t
m_num_uvs : size_t
m_num_vertices : size_t
m_recenter_mesh : bool
m_tangent_data : float*
m_uv_data : float*
m_vao_handle : GLuint
m_vertex_data : float*
Methods
~Mesh()
center() : void
import_bin_file() : void
import_obj_file() : void
Mesh()
print_info() : void
render() : void
store_vbo() : void
trim_string() : void
Character
Class
Renderable
Fields
m_angle : float
m_animation_state : unsigned int
m_delta : vec3
m_end_walking_cycle_state : unsigned int
m_last_animation_update : double
m_last_update : double
m_meshes : vector<Mesh*>*
m_model_matrix : mat4
m_model_view_matrix : mat4
m_normal_matrix : mat3
m_parent : Scene*
m_position : vec3
m_program : Shader_program*
m_program_linked : bool
m_start_walking_cycle_state : unsigned int
m_state : State
m_tex : Texture*
m_walking_speed : float
Methods
~Character()
Character()
compile_link_shader() : void
get_angle() : float
get_animation_state() : int
get_meshes() : vector<Mesh*>*
get_parent() : Scene*
get_position() : vec3
get_state() : State
get_walking_speed() : float
render() : void
set_angle() : void
set_animation_state() : void
set_meshes() : void
set_position() : void
set_state() : void
set_texture() : void
set_walking_speed() : void
update() : void
Nested Types
State
Enum Class
IDLE
WALK_FORWARD
WALK_BACKWARD
TURN_LEFT
TURN_RIGHT
Plane
Class
Renderable
Fields
m_buffer_handle : GLuint[4]
m_faces : int
m_vao_handle : GLuint
Methods
~Plane()
Plane()
render() : void
Scene
Class
Renderable
Fields
m_parent : GLFWwindow*
Methods
get_camera() : Third_person_camera*
get_light() : Light*
get_parent() : GLFWwindow*
init_scene() : void
render() : void
Scene()
update() : void
public
Main_scene
Class
Scene
Fields
console_thread : thread*
m_camera : Third_person_camera*
m_character_meshes : vector<vector<Mesh*>*>
m_characters : map<string, Character*>
m_floor : Floor*
m_light : Light*
m_nameplates : map<string, Nameplate*>
m_textures : vector<Texture*>
Methods
~Main_scene()
get_camera() : Third_person_camera*
get_characters() : map<string, Character*>
get_light() : Light*
get_player() : Character*
init_scene() : void
Main_scene()
process_console_input() : void
render() : void
update() : void
Light
Struct
Fields
intensity : vec3
position : vec4
Material
Struct
Fields
ambient : vec3
diffuse : vec3
shininess : float
specular : vec3
Packed_vertex
Struct
Fields
normal : vec3
position : vec3
uv : vec2
Methods
Packed_vertex()
Texture
Class
Fields
m_height : GLuint
m_pixel_data : GLubyte*
m_tex_id : GLuint
m_width : GLuint
Methods
~Texture()
bind() : void
get_height() : GLuint
get_width() : GLuint
load_texture() : void
load_tga() : void
read_int_le() : int
read_short_le() : int
Texture()
write_short_le() : void
write_tga() : void
Shader_program
Class
Fields
m_handle : int
m_linked : bool
m_uniform_locations : map<strin...
Methods
~Shader_program()
bind_attrib_location() : void
bind_frag_data_location() : void
compile_shader() : void (+ 2 overl...
file_exists() : bool
get_extension() : string
get_handle() : int
get_type_string() : const char*
get_uniform_location() : GLint
is_linked() : bool
link() : void
operator=() : Shader_program&
print_active_attributes() : void
print_active_uniform_blocks() : void
print_active_uniforms() : void
set_uniform() : void (+ 9 overloads)
Shader_program() (+ 1 overload)
use() : void
validate() : void
Shader_file_ext
Struct
Fields
ext : const char*
type : GLenum
Thread_pool
Class
Fields
m_condition : condition_variable
m_queue_mutex : mutex
m_stop : bool
m_tasks : queue<std::function<void()>>
m_workers : vector<thread>
Methods
~Thread_pool()
enqueue<F, ...Args>()
Thread_pool()
Main_event_handler
Class
Fields
m_cursor_pos_x : int
m_cursor_pos_y : int
m_dolly_max_rad : float
m_dolly_min_rad : float
m_dolly_speed : float
m_height : int
m_instance : Main_event_handler*
m_pan_speed : float
m_parent : Main_scene*
m_prev_cursor_pos_x : int
m_prev_cursor_pos_y : int
m_width : int
m_window : GLFWwindow*
Methods
create_instance() : void
cursor_pos() : void
get_instance() : Main_event_handler*
key() : void
Main_event_handler()
mouse_button() : void
resize() : void
scroll() : void
Frustum
Class
Fields
c_float_tolerance : const unsigned
m_aspect_ratio : float
m_fovy : float
m_is_perspective : bool
m_projection_matrix : mat4
m_recalc_perspective_matrix : bool
m_z_far : float
m_z_near : float
Methods
~Frustum()
Frustum() (+ 2 overloads)
get_aspect_ratio() : float
get_far_z_distance() : float
get_field_of_view_y() : float
get_near_z_distance() : float
get_projection_matrix() : mat4
is_orthographic() : bool
is_perspective() : bool
set_aspect_ratio() : void
set_far_z_distance() : void
set_field_of_view_y() : void
set_near_z_distance() : void
set_projection_matrix() : void
public
Camera
Class
Frustum
Fields
c_rotation_hit_count_max : const unsigned
m_eye_position : vec3
m_forward : vec3
m_left : vec3
m_orientation : quat
m_recalc_view_matrix : bool
m_rotation_hit_count : unsigned short
m_up : vec3
m_view_matrix : mat4
Methods
Camera() (+ 2 overloads)
get_forward_direction() : vec3
get_left_direction() : vec3
get_orientation() : quat
get_position() : vec3
get_up_direction() : vec3
get_view_matrix() : mat4
init_local_coordinate_system() : void
look_at() : void (+ 2 overloads)
normalize_camera() : void
pitch() : void
register_rotation() : void
roll() : void
rotate() : void
set_position() : void (+ 1 overload)
translate() : void (+ 1 overload)
translate_local() : void (+ 1 overload)
yaw() : void
public
Third_person_camera
Class
Camera
Fields
m_pitch_angle : float
m_radius : float
m_target : vec3
m_yaw_angle : float
Methods
add_pitch_angle() : void
add_radius() : void
add_target() : void
add_yaw_angle() : void
get_pitch_angle() : float
get_radius() : float
get_target() : vec3
get_yaw_angle() : float
set_pitch_angle() : void
set_radius() : void
set_target() : void
set_yaw_angle() : void
Third_person_camera()
update() : void