**CIS 422**

**Developer Documentation**

**Panda Engine**

This document was derived from EyeDraw Programmer’s Documentation by Anthony Hornof, Rob Hoselton and Anna Cavender

PDF: <https://www.cs.uoregon.edu/Classes/10W/cis422/Handouts/EyeDrawDocumentation.pdf>

Online: <https://www.cs.uoregon.edu/research/cm-hci/EyeDraw/>

[**1 Introduction**](#_ba24094dv0n0) **5**

[1.1 Glossary](#_gx450atnwqbq) 5

[**2. Developer Use**](#_mpuj4dxqx8vg) **5**

[2.1 Introduction](#_qclfh2112j6k) 5

[2.2 Source Files](#_igpbx7x9ixqe) 6

[2.2.1 Component Source Files](#_2wu4igajvjf) 6

[2.2.1.1 Shader.h](#_nxoaj265vha6) 6

[2.2.1.1.1 Shader](#_605otn29d82w) 6

[2.2.1.1.2 Data Members](#_ll025xk4ax1r) 6

[2.2.1.1.3 Enumerations](#_yau2799zovby) 6

[2.2.1.1.4 Public Members](#_hwmx634006r8) 7

[2.2.1.1.5 Private Members](#_yxl4jtof8b8d) 7

[2.2.1.2 Shader.cpp](#_7vve7ancl5hr) 7

[2.2.1.2.1 Constructor](#_abed3q49aemd) 7

[2.2.1.2.1 Destructor](#_yhmx3nnyp71t) 7

[2.2.1.2.1 Update](#_e56m2axxpkms) 7

[2.2.1.2.1 Bind](#_idai6brpxpc) 8

[2.2.1.2.1 Createshader](#_qrpbmkvnxy2n) 8

[2.2.1.2.1 VerifyShader](#_iud5a5m5k4m2) 8

[2.2.1.3 Mesh.h](#_t83l6jxikt3m) 8

[2.2.1.3.1 Vertex](#_7d6x687sdmrb) 8

[2.2.1.3.1 Public Members](#_wcztzq2lbvc2) 8

[2.2.1.3.1 Private Members](#_xkala8a9ohbl) 9

[2.2.1.4 Mesh.cpp](#_yez59thbqbl2) 9

[2.2.1.4.1 Constructors](#_nwaufmy3033i) 9

[2.2.1.4.2 InitMesh](#_rhufgk9m3wjj) 9

[2.2.1.4.3 DrawMesh](#_60l1i9grcctk) 10

[2.2.1.5 Texture.h](#_wsmxpxpjo8s1) 10

[2.2.1.5.1 Public Data Members](#_w5uvnue54pd0) 10

[2.2.1.5.2 Private Data Members](#_mvn7rl82jeo5) 10

[2.2.1.6 Texture.cpp](#_bv4o395q66f0) 10

[2.2.1.6.1 Constructor](#_fk1otesbtp5n) 10

[2.2.1.6.2 Bind](#_ff1nk7k7r9jp) 10

[2.2.1.7 Display.h](#_6sm2hvz2sgzg) 11

[2.2.1.7.1 Public Members](#_fiwrh3o4afq0) 11

[2.2.1.7.1 Private Members](#_wgzolvkys6t5) 11

[2.2.1.8 Display.cpp](#_uxy8hq8ftzf) 12

[2.2.1.8.1 Constructor](#_ihm7rceng97z) 12

[2.2.1.8.2 set\_attr](#_quu3c1nlicts) 12

[2.2.1.8.3 Event Listeners](#_4h21is8culdm) 12

[2.2.1.9 Camera.h](#_83du353kt8sq) 12

[2.2.1.9.1 Public Members](#_ws2cdatiubz6) 13

[2.2.1.9.2 Private Members](#_lf6o914yt5ha) 13

[2.2.1.10 Camera.cpp](#_k9lkdu9bizww) 13

[2.2.1.9.2 Constructor](#_vfqvw1kvotis) 13

[2.2.1.9.3 GetProjection](#_xs0oi7wkw0m) 13

[2.2.1.10 Transform.h](#_fxphh6ddso05) 13

[2.2.1.10.1 GetProjection](#_6jzjak4dvii) 13

[2.2.1.10.2 Public Members](#_qofrfdcwacwi) 14

[2.2.1.11 Transform.cpp](#_vlsmqoh4yd7a) 14

[2.2.1.11.1 Transform::MatModel()](#_obc9y5u8tobm) 14

[2.2.1.12 GameLoop.h](#_rm5mfzi9iyma) 14

[2.2.1.12.1 Public Members](#_h03t9lsfjvjo) 14

[2.2.1.12.1 Private Members](#_gdocuati6o4d) 15

[2.2.1.13 GameLoop.cpp](#_n4z56o3l7xcd) 15

[2.2.1.14 GameManager.h](#_xriahp5e628j) 15

[2.2.1.14.1 Public Members](#_w41gk77rkhao) 15

[2.2.1.14.2 Private/Protected Members](#_tafdsaw5q1b2) 16

[2.2.1.15 GameManager.cpp](#_gtkg4yne90e0) 16

[2.2.1.16 EventDispatcher.hpp](#_1fugli8oepma) 16

[2.2.1.16.1 Public Members](#_efcg3d2mp7iu) 16

[2.2.1.16.2 Private/Protected Members](#_me6jd9saycpj) 17

[2.2.1.16 EventDispatcher.cpp](#_mgthd8vn9uny) 17

[2.2.1.17 Listeners.hpp](#_r2u0xogkz4bg) 17

[2.2.1.17.1 Display](#_n8y4595wtrqh) 18

[2.2.1.17.2 TimedEventDetails](#_gii17ghv3v9y) 18

[2.2.1.17.3 MouseButtonEventDetails](#_k0vmxocr6fu) 18

[2.2.1.17.4 KeyboardEventDetails](#_lf6b4t9d8nit) 18

[2.2.1.18 GameOptions.hpp](#_h72pzhh2dima) 18

[2.2.1.18.1 WindowOptions](#_r8lwc3rbzyhr) 18

[2.2.1.18.2 Public Members](#_nsxpyn15nqrx) 18

[2.2.1.18.3 Private Members](#_sgegy58zf533) 18

[2.2.1.18.4 GameOptions](#_tuaadmo5q488) 18

[2.2.1.18.5 Public Members](#_n6t8ywa259n0) 19

[2.2.1.18.6](#_6wo5gkyoof4) 19

[2.2.1.19 Settings.hpp](#_fwy6x27sjg2c) 19

[2.2.1.19.2 Public Members](#_t0819d6r8zfi) 19

[2.2.1.20 Utils.hpp](#_3ixmekp2zazu) 19

[2.2.1.20.1 Public Static Members](#_2v5u9j72u2s9) 19

[2.2.1.20.1 Public Members](#_5k0vb2xgcqvp) 19

[2.3 Third Party Libraries](#_z2pun539pwgo) 20

# 

# 1 Introduction

## 1.1 Glossary

**Vertex** a point that terminates a line

**Vector** a collection of vertices

**3D** Containing height, width and depth

**Linear** **Combination** an expression constructed from a set of terms by multiplication

**Matrix** a rectangular array of data

**Shader** a program which calculates rendering effects on graphics hardware

**Uniform** a variable inside a shader able to be accessed outside a shader

**Texture** a 2D image bound to all or a portion of a 3D model

**Mesh** a collection of vertices to create a 3D model

**OBJFile** a standard format for storing vertices

**Cmake** a free cross platform build system

# 2. Developer Use

## 2.1 Introduction

This document covers the source code for Panda Engine, an open source 3D game engine developed in C++ for University of Oregon class CIS422 Software Methodology.

This document assumes the reader has fundamental knowledge of C++ ,common programming paradigms, and a knowledge of the GNU/Linux operating system.

The document makes no assumption on the readers knowledge of third party libraries used in the development of Panda Engine. Knowledge of OpenGL, SDL2, and GLM are highly recommended for a more accurate idea of the inner workings of the system.

This document provides an overview of each source file, giving the reader information on the inner workings of each file. This document is intended to assist future and current developers creating new features or editing existing features.

## 2.2 Source Files

## 2.2.1 Component Source Files

### 2.2.1.1 Shader.h

#### 2.2.1.1.1 Shader

This class handles how shaders communicate with the system. The class parses a given shader file attempts to link and compile the file. The class handles basic linking and compiling errors, based off the standards of the OpenGL shading language GLSL.

#### 2.2.1.1.2 Data Members

* GLuint prog
* GLuint nshaders[N\_SHADER]
* GLuint nuniform[NUNIFORM]

All data members are an OpenGL datatype of unsigned it. Prog is a reference to the shader program passed in through the construct and parsed by the class. Nshaders is an array of GLuints used to hold all shaders. The array is initialized using an enum which defines all shader types. Nuniform is an array of GLuints to hold all uniforms. The array is initialized using an enum which defines all uniforms.

#### 2.2.1.1.3 Enumerations

* SHADERTYPE
* UNIFORM

The enumeration SHADERTYPE represents the types of shaders the engine supports. The current values are: VERTEX, and FRAGMENT. These values represent the vertex shaders and fragment shaders.

* **Vertex** **Shader**

A shader which has an effect on all or a select amount of vertices within the system

* **Fragment** **Shader**

A shader which has an effect on all or a select amount of pixels within the system

The enumeration UNIFORM represents the types of uniforms the engine supports. The current values are TRANSFORM. This allows a shader to communicate data outside of the shader class. The vertex shader takes advantage of this value using it to perform transformations on screen to simulate movement in a 3D space.

#### 2.2.1.1.4 Public Members

* Shader(const std::string& fname)
* ~Shader()
* void Bind()
* void Update(const Transform& t, const Camera& camera)

#### 2.2.1.1.5 Private Members

* std::string loadshader(const std::string fname);
* GLuint createshader(const std::string& file, GLenum type);
* void verifyshader(GLuint shader, GLuint flag, bool isprog, const std::string& error)

### 2.2.1.2 Shader.cpp

This file contains the logic for all functions and data members within the Shader.hpp file.

#### 2.2.1.2.1 Constructor

The constructor takes a shader filename as a parameter. The system then attempts to create an OpenGL program containing the shader information. To create the program the text within the file is parsed and then checked for validity. If the file is valid a shader program is created if a file is not valid an error is created and logged.

The system then stores the newly created program in the private data member nshaders. The stored data is then used by OpenGL with the glAttachShader function. The program is then compiled and linked with OpenGL, and variables within the shader programs are linked with OpenGL.

#### 2.2.1.2.1 Destructor

The destructor iterates over the private data member nshaders, freeing all memory used to store the shaders. The system then calls the OpenGL function glDeleteProgram to free the memory OpenGL has allocated for the shader program.

#### 2.2.1.2.1 Update

The update function is used to access the vertex shader variable which controls the current view position. The accessed variable is changed through matrix multiplication.

#### 2.2.1.2.1 Bind

The bind function calls the OpenGL function glUseProgram, giving it the private member prog. This OpenGL function accesses computer hardware and feeds the information contained within the created shader program.

#### 2.2.1.2.1 Createshader

The create shader function loads the contents of a file and stores the contents into a string. The contents of the string are then compiled through the glCompileShader program. The compiled shader is then verified. If the verification is successful the shader program is returned

#### 2.2.1.2.1 VerifyShader

This function takes the shader program and validates it using the OpenGL function glGetProgramiv. If the shader is verified through this function, the function exits. If an error is found the function outputs an error.

### 2.2.1.3 Mesh.h

This file contains the information to create a mesh in the context of OpenGL. A mesh is a set of vertices in some dimensional space represented by a matrix or vector.

#### 2.2.1.3.1 Vertex

This nested class is the representation of each mesh detail. The class itself contains only the data pos, and tex. These are vectors representing either the position of the vertices in some dimensional space, or the texture mapping within the same dimensional space. The class uses standard data hiding practices.

#### 2.2.1.3.1 Public Members

* Mesh(Vertex\* vert, unsigned int nvert, unsigned int\* indeces, unsigned int nindeces);
* Mesh(const std::string& fname)
* DrawMesh()
* ~Mesh();

This class contains two constructors. The first constructor takes a Vertex pointer an unsigned int pointer and an unsigned int. It uses this information to create create the vertices representation in some dimensional space. The second constructor takes a file name and utilizes the OBJloader class to create the mesh from an OBJ file. The destructor frees all allocated memory used during the creation of the mesh. The draw function transfers the information from the mesh to the GPU.

#### 2.2.1.3.1 Private Members

* enum{ POSITION\_VB, TEXTURE\_VB,INDEX, NBUFF };
* void InitMesh(const IndexedModel& model);
* GLuint vertArr;
* GLuint vertArrBuf[NBUFF];
* unsigned int ndraw;

The anonymous enum is used to indicate which data point is being sent to the GPU in a more human readable format. The initMesh function is used to translate the contents of an OBJ file into a format OpenGL may use. The vertArr and vertArrBuff and variables to store the OpenGL representation of data. Ndraw is a variable representing the number of items OpenGL shall draw.

### 2.2.1.4 Mesh.cpp

This file contains all the logic and implementation details used to create the mesh representation.

#### 2.2.1.4.1 Constructors

* Mesh(const std::string& fname)

This constructor takes a filename as a parameter. The file is then converted to an OBJModel object. The translated file is then passed to helper function initmesh.

* Mesh(Vertex\* vert, unsigned int nvert,unsigned int\* indeces, unsigned int nindeces)

This constructor parses the contents of the Vertex array passed in. The contents are held within two vectors in a format OpenGL expects. An IndexedModel is then created using the contents of these vectors and initmesh is called.

#### 2.2.1.4.2 InitMesh

The initmesh function utilizes OpenGL functions to create a representation of data the GPU may draw. The data must be given to OpenGL sequentially. The data is given in the following order: position, texture, and indices.

#### 2.2.1.4.3 DrawMesh

The draw mesh function binds the private data member vertArr as the data for the GPU to draw. The function then specifies the draw format (triangles by default). The data is then freed so the GPU may perform other tasks

### 2.2.1.5 Texture.h

The texture file handles loading image data and mapping the data to a space in a display window. The following image formats are supported: jpg, png, bmp.

#### 2.2.1.5.1 Public Data Members

* Textures(const std::string&fname)
* ~Textures()
* Bind(unsigned int textnum)

The constructor takes a filename, which is the image to load. The destructor handles deallocating any resources used during the creation of a texture. The bind function uses OpenGL functions to communicate the parsed image data to the GPU.

#### 2.2.1.5.2 Private Data Members

* GLuint texturehandle

The private data member used as a pointer to the parsed image data.

### 2.2.1.6 Texture.cpp

#### 2.2.1.6.1 Constructor

The constructor attempts to load the filename passed in as a parameter. The file is then loaded by the third party library stbi image. If the file has been loaded without error, the height, width and bits per pixel are loaded into a pointer. The constructor then allocates space for the texture and binds the texture into an OpenGL buffer. This buffer is then sent to the GPU with the parsed information.

#### 2.2.1.6.2 Bind

The bind function sends the information including texture type to the GPU.

### 2.2.1.7 Display.h

The display class handles the creation of a display window and the initialization of OpenGL and SDL. On initialization the class creates all necessary components to communicate with the hardware.

#### 2.2.1.7.1 Public Members

* Display(Engine::GameLoop\* g, std::string title, int w, int h);
* ~Display();
* void SwapDisp();
* void Clear(float r, float g, float b, float a);
* float GetHeightf(){return static\_cast<float>(this->sheight);}
* float GetWidthf(){return static\_cast<float>(this->swithd);}
* void markDirty(){this->isDirty = true;};
* void ShowWindow();
* void HideWindow();
* void RegisterWindowListener(Listener::GameGUIListener\* list, Events::Priority p);
* void UnregisterWindowListener(Listener::GameGUIListener\* list);
* void ExecuteGUIEvent(Events::WindowEventDetails\* details);
* Management::WindowOptions\* GetWindowOptions(){return this->windOpts;}
* std::map<Events::Priority, std::unordered\_set<Listener::GameGUIListener\*>\* > gameGUIListeners{};
* bool isShown;
* high\_resolution\_clock::time\_point lastFrame;
* high\_resolution\_clock::time\_point startingTime;
* bool isDirty;
* Management::WindowOptions\* windOpts;

The constructor takes a string used as a title, window size and a game loop pointer. The constructor creates the window and binds the window to the gameloop pointer passed in. The swap display function switches the current screen the end user is displayed. The clear function swaps colors the screen with a red, green, blue and alpha value. The class follows standard data hiding techniques with getters and setters.

The class contains event managers to register, unregister and listen for events within the window. The window class contains high resolution timers to set the frame rate to a user specified number. The bool isDirty represent if a screen has been drawn on, with a value of true meaning it has already been drawn on.

#### 2.2.1.7.1 Private Members

* Int swidth
* int sheight;
* std::string title;
* Engine::GameLoop\* game;
* void set\_attr();
* SDL\_Window\* window;
* SDL\_GLContext glcontext;

The private members represent the current height, width and title of the window. The private members also represent the current OpenGL context, which is a communication line between the GPU and the program.

### 2.2.1.8 Display.cpp

#### 2.2.1.8.1 Constructor

The constructor first sets the isdirty flag to false, then registers game event listeners. Any user defined settings are then created through a window options manager. OpenGL preferences are then set through a private function set\_attr.

A window is then created and bound as an OpenGL window along with an OpenGL context. OpenGL is then checked for errors. If no errors exist, OpenGL settings are then set and the window is bound to the game loop parameter.

#### 2.2.1.8.2 set\_attr

This function sets OpenGL options. The following options are set: size of the Red, Green, Blue and alpha bits, the buffer size of a window, the depth information stored, and setting the use of double buffers (two screens).

### 2.2.1.8.3 Event Listeners

* ExecuteGUIEvent(Events::WindowEventDetails\* eventDetails)
* UnregisterWindowListener(Listener::GameGUIListener\* list)
* RegisterWindowListener(Listener::GameGUIListener\* list, Events::Priority p)

The ExecuteGUIEvent function reacts to events within the screen. If an event occurs, the function executes an eventdetails option defined by the user.

The unregisterWindowListener function removes an event listener from the window.

The RegisterWindowListener function adds an event to the window with a priority.

### 2.2.1.9 Camera.h

The camera class handles all aspects of the view a user is displayed on the OpenGL window. The class uses the glm library to handle matrix mathematics used in calculating the way the objects are displayed.

#### 2.2.1.9.1 Public Members

* Camera(const glm::vec3& pos , float fov, float aspect, float near, float far);
* glm::mat4 GetProjection() const;

The constructor creates a view using the position, the field of view, the aspect ratio of the screen, and the farthest/nearest points to display to the user. The function GetProjection matrix math to create and return the projection a camera should use.

#### 2.2.1.9.2 Private Members

* glm::mat4 perspective;
* glm::vec3 position;
* glm::vec3 forward;
* glm::vec3 up

The private members are all vectors representing either the forward view, the upward view, the position of the camera, or the current view perspective.

### 2.2.1.10 Camera.cpp

#### 2.2.1.9.2 Constructor

The constructor translates each parameter into a perspective returned by the glm function perspective. The position of the camera is then set as the position passed as the parameter.

#### 2.2.1.9.3 GetProjection

The function calculates the view projection based off the product of perspective and the value returned by the glm function lookat.

### 2.2.1.10 Transform.h

This class contains the positional, rotational and scalar vectors. The class updates all vertices within a scene. It combines each data member and returns the transformation matrix each vertex should run through.

#### 2.2.1.10.1 GetProjection

* glm::vec3 pos
* glm::vec3 rot
* glm::vec3 scl

Each data member represents a column vector, either position, rotation, or scale. These may be adjusted to change the position, rotation and scale.

#### 2.2.1.10.2 Public Members

* glm::vec3& GetPos()
* glm::vec3& GetRot()
* glm::vec3& GetScl()
* SetPos(const glm::vec3& p)
* SetRot(const glm::vec3& r)
* SetScl(const glm::vec3& s)
* glm::mat4 MatModel()

This class contains standard C++ getters and setters conforming to the standard of data hiding. The getters and setters differ slightly as they getters return a reference to a vector, and the settlers take a constant reference as a parameter. This choice was to speed up the calculations. C++ by default copies parameters. Copying the parameter would impact performance significantly therefore these are passed as a reference.

### 2.2.1.11 Transform.cpp

This class takes the three vectors passed in from the constructor and sets the private position,rotation and scale vectors to this value.

#### 2.2.1.11.1 Transform::MatModel()

The function takes the private variables and combines the variables into a transform matrix. Due to the nature of three dimensional matrix, the x,y, and z coordinates must be calculated separately then combined.

### 2.2.1.12 GameLoop.h

This class handles all the game logic including: creating a window, starting a game, pausing a game, stopping a game, aborting a game and managing/creating the events.

#### 2.2.1.12.1 Public Members

* GameLoop();
* ~GameLoop();
* void CreateWindow(std::string title, int w, int h);
* void StartGame(bool startPaused)
* void PauseGame()
* void StopGame(bool forced)
* void AbortGame
* Management::GameOptions\* getOptions()

The game loop handles the creation of all components of the game system. The create window calls the window function checking the minimum and maximum size. The window created is set to hidden by default. The pause and resume game set the current state of the paused boolean to true or false depending on if the current state of the game.

#### 2.2.1.12.1 Private Members

* void Run();
* void initback();
* Display \*display
* bool gameRunning;
* std::chrono::high\_resolution\_clock::time\_point startingTime;
* bool gamePaused;
* high\_resolution\_clock::time\_point lastTick;
* Management::GameOptions \*options;

The run function is the main game loop. All game logic happens within this private function. The initback function is used to communicate and initialized SDL. The starting time is used to keep track of the current frame rate the game is running at.

### 2.2.1.13 GameLoop.cpp

This function contains an example game loop in the run function. The run function contains a SDL\_Event listener.This event listener interprets the peripheral device input and interprets the input based off logic contained within an event dispatcher. A high resolution clock is used to track the frame rate the game runs at. This clock is used to synchronize all events. At the end of the loop, a time delta is created and is used in the synchronization process.

### 2.2.1.14 GameManager.h

The game manager class is used to manage and dispatch events to the game. The game manager handles starting,stopping, pausing, resuming and forcing quit the game. The class also handles the game options and thread creation.

#### 2.2.1.14.1 Public Members

* friend void jumperThread(GameManager\* gm)
* static GameManager\* &instance(void)
* void CreateWindow(std::string title, int w, int h)
* void StartGame(bool startPaused)
* void StopGame(bool forced)
* void PauseGame(void)
* void ResumeGame(void)
* void Abort(std::string reason)
* void WaitForGameEnd(void)
* Management::GameOptions\* GetOptions()
* static std::mutex\* io\_mutex

The jumperthread serves as a point for threads to reach the run function from the main game loop. The gamemanager ensures only one instance is handles at a time, ensuring thread safe activity. The class contains functions to manage the state of the game.

#### 2.2.1.14.2 Private/Protected Members

* std::thread\* mainGameThread
* void Run();
* bool gamePaused
* GameManager() : startingTime(std::chrono::high\_resolution\_clock::now()), lastTick(std::chrono::high\_resolution\_clock::now()
* bool gameRunning
* bool gameSetup;
* std::chrono::high\_resolution\_clock::time\_point startingTime;
* high\_resolution\_clock::time\_point lastTick;
* Management::GameOptions\* options;

The private data members keep track of the current state of the game thread. These are used to either pause the game state, re run the game state, or abort the game state. The private members make use of the high resolution clock to synchronize all activity.

### 2.2.1.15 GameManager.cpp

This class allows mutex access for other threads created by the game manager. The run function creates an event dispatcher listening for events while the game state is not aborted. The start game function sets the state to running, creates the synchronization clock and creates a new thread.

### 2.2.1.16 EventDispatcher.hpp

This class handles the registration, management and execution of event listeners. The class implements a priority system to allow for some sort of ordering of importance in event execution.

#### 2.2.1.16.1 Public Members

* static EventDispatcher\* &instance(){static EventDispatcher\* e\_d
* void RegisterEventListener( GameEventsListener\* l,Priority p);
* void UnregisterEventListener( GameEventsListener\* listener);
* void RegisterTickListener(GameTickListener\* l,Priority p);
* void UnregisterTickListener( GameTickListener\* listener);
* void RegisterKeyListener(GameKeyboardListener\* l,Priority p);
* void UnregisterKeyListener( GameKeyboardListener\* listener);
* void RegisterMouseListener(GameMouseListener\* l,Priority p);
* void UnregisterMouseListener( GameMouseListener\* listener);
* bool RegisterUserDefinedEvent(const string eventName);
* void RegisterUserDefinedListener( GenericEventListener\* lis, string eventName, Priority p);
* void UnregisterUserdefinedListener(GenericEventListener\* lis, string eventName);
* bool ExecuteUserDefinedEvents( string eventName, EventDetails\* details);

The public members of this class all handle event registration and removing registered events. The event types the system currently supports are: SDL events, game tick, keyboard input, mouse input, and custom user defined events.

#### 2.2.1.16.2 Private/Protected Members

* std::map<Priority, std::unordered\_set<GameEventsListener\*>\* > gameEventListeners{};
* std::map<Priority, std::unordered\_set<GameTickListener\*>\* > gameTickListeners{};
* std::map<Priority, std::unordered\_set<GameKeyboardListener\*>\* > gameKeyListeners{};
* std::map<Priority, std::unordered\_set<GameMouseListener\*>\* > gameMouseListeners{};
* std::unordered\_set<std::string> userEventNames{};
* std::map<std::string, std::map<Priority, std::unordered\_set<GenericEventListener\*>\* >\* > userEventListeners;
* void ExecuteGameEvent(GameEvent event, EventDetails\* details);
* void ExecuteTickEvent(TimedEventDetails\* details);
* void ExecuteKeyEvent(KeyboardEventDetails\* details);
* void ExecuteMouseEvent(MouseButtonEventDetails\* details);

The data members of this class all involve the storage of events. For speed, the registered events are stored in a std::map. When the event listener senses an event, the executeevent function is called according to the type of event.

### 2.2.1.16 EventDispatcher.cpp

This class implements the functions and data members described in the EventDispatcher.hpp file. The majority of these functions deal with either storing a user defined event, or looking up the value of a user defined event. This class implements basic logging for errors if an event is not found, but attempts to recover from this error and should not crash.

### 2.2.1.17 Listeners.hpp

This header class is to provide base classes for a game script to register event handlers/

interceptors for game events executed by the game engine. This file is made of multiple classes, each handling the different event types.

#### 2.2.1.17.1 Display

The display class handles events regarding the display. The class contains an enum which represents the priority of the event with 5 being the highest and 1 being the lowest. The class stores and event id and string which is used to search for the event.

#### 2.2.1.17.2 TimedEventDetails

The timed event details class handles events regarding a time elapse event. These events are used to handle game ticks, or time within a game.

#### 2.2.1.17.3 MouseButtonEventDetails

The mousebuttonevent class handles events regarding a mouse peripheral device. These events include mouse button clicks, mouse motion events, and current mouse position within the OpenGL screen.

#### 2.2.1.17.4 KeyboardEventDetails

The keyboardevent class handles events regarding a keyboard peripheral device. These events include a key press, release, the value of the key, and if the key has been repeated.

### 2.2.1.18 GameOptions.hpp

This header file is for configuring one of the game option, game rate. The It is generated for each singular event and windows.

#### 2.2.1.18.1 WindowOptions

Stores the target frame refresh rate for a specified window. Allows for setting and getting the current target.

#### 2.2.1.18.2 Public Members

* short getFrameRateTarget
* void setFrameRateTarget(short target)

#### 2.2.1.18.3 Private Members

* short frameRateTarget

#### 2.2.1.18.4 GameOptions

Stores the target game logic “tick”/refresh rate for a specified game loop. Allows for setting and getting the current target.

#### 2.2.1.18.5 Public Members

* short getTickRateTarget
* void setTickRateTarget(short target)

#### 2.2.1.18.6

* short tickRateTarget

### 2.2.1.19 Settings.hpp

This header file is for defining global variables to be used throughout the application. Most are defaults if the value isn’t specified on object creation.

#### 2.2.1.19.2 Public Members

* int Settings::Window::window\_width
* int Settings::Window::window\_height
* bool Settings::Window::is\_fullscreen
* bool Settings::Window::is\_center
* std::string Settings::Shader::default\_dir
* int Settings::Sound::volume
* std::string Settings::Multimedia::texture\_dir
* std::string Settings::Multimedia::model\_dir
* std::string Settings::Multimedia::music\_dir
* std::string Settings::Logging::error\_dir
* bool Settings::Logging::is\_time
* bool Settings::Debug::display\_fps
* bool Settings::Debug::console\_write

### 2.2.1.20 Utils.hpp

This header class is to provide a class that is capable of performing calculations relating to timing of events.

#### 2.2.1.20.1 Public Static Members

* Utilities\* instance()

This function returns a instance to the utilities object, it ensures that only one such object exists at any given time.

#### 2.2.1.20.1 Public Members

* long long getMillisFrom(high\_resolution\_clock::time\_point \*then, high\_resolution\_clock::time\_point \*now)
* long long getMillisWaitTime(short rate)

The functions above will determine the delta time between any two time points and the required delta time to be below a certain rate.

## 2.3 Third Party Libraries

The system makes use of multiple third party libraries. The following third party libraries are used in the system.

OpenGL - <https://www.opengl.org/documentation/>

OpenGL is used as a graphic library. This library allows the system to communicate with system hardware.

SDL2 - <https://wiki.libsdl.org/FrontPage>

SDL2 is used as a system library. SDL2 makes the process of communicating with Operating System features such as keyboards, mouse events, and sound easier.

GLM - <https://glm.g-truc.net/0.9.8/api/index.html>

GLM is used as a mathematics library. The library contains many common mathematical functions used in 3D graphic processing

Stbimage - <https://github.com/nothings/stb>

A c/c++ image loading library used to load image data of textures.

OBJloader - <https://github.com/Bly7/OBJ-Loader>

The object loader used to load object files. The code contained is based off this implementation of obj loader.

## 