TinyWindow

0.3

Generated by Doxygen 1.8.7

Mon Nov 2 2015 03:32:38

Contents

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

windowManager::tWindow . . . . . ??
windowManager . . . . . ??

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

TinyWindow.h . . . . . ??

Chapter 3

Data Structure Documentation

3.1 windowManager::tWindow Struct Reference

Public Member Functions

- tWindow ()

Data Fields

- const char * name
- GLuint iD
- GLuint colourBits
- GLuint depthBits
- GLuint stencilBits
- GLboolean keys [256+1+54]
- GLboolean mouseButton [2+1]
- GLuint resolution [2]
- GLuint position [2]
- GLuint mousePosition [2]
- GLboolean shouldClose
- GLboolean inFocus
- GLboolean initialized
- GLboolean contextCreated
- GLboolean isCurrentContext
- GLuint currentState
- GLuint currentWindowStyle
- onKeyEvent_t keyEvent
- onMouseButtonEvent_t mouseButtonEvent
- onMouseWheelEvent_t mouseWheelEvent
- onDestroyedEvent_t destroyedEvent
- onMaximizedEvent_t maximizedEvent
- onMinimizedEvent_t minimizedEvent
- onFocusEvent_t focusEvent
- onMovedEvent_t movedEvent
- onResizeEvent_t resizeEvent
- onMouseMoveEvent_t mouseMoveEvent
- Window windowHandle
- GLXContext context
- XVisualInfo * visualInfo

- GLint * attributes
- XSetWindowAttributes setAttributes
- GLbitfield decorators
- Atom AtomState
- Atom AtomHidden
- Atom AtomFullScreen
- Atom AtomMaxHorz
- Atom AtomMaxVert
- Atom AtomClose
- Atom AtomActive
- Atom AtomDemandsAttention
- Atom AtomFocused
- Atom AtomCardinal
- Atom AtomIcon
- Atom AtomHints
- Atom AtomWindowType
- Atom AtomWindowTypeDesktop
- Atom AtomWindowTypeSplash
- Atom AtomWindowTypeNormal
- Atom AtomAllowedActions
- Atom AtomActionResize
- Atom AtomActionMinimize
- Atom AtomActionShade
- Atom AtomActionMaximizeHorz
- Atom AtomActionMaximizeVert
- Atom AtomActionClose
- Atom AtomDesktopGeometry

3.1.1 Detailed Description

3.1.2 Constructor & Destructor Documentation

3.1.2.1 windowManager::tWindow::tWindow ( ) [inline]

< the window is in its default state

< the default window style for the respective platform

```
02074             {
02075                 name = nullptr;
02076                 iD = NULL;
02077                 colourBits = NULL;
02078                 depthBits = NULL;
02079                 stencilBits = NULL;
02080                 shouldClose = GL_FALSE;
02081                 currentState = WINDOWSTATE_NORMAL;
02082 
02083                 keyEvent = nullptr;
02084                 mouseButtonEvent = nullptr;
02085                 mouseWheelEvent = nullptr;
02086                 destroyedEvent = nullptr;
02087                 maximizedEvent = nullptr;
02088                 minimizedEvent = nullptr;
02089                 focusEvent = nullptr;
02090                 movedEvent = nullptr;
02091                 resizeEvent = nullptr;
02092                 mouseMoveEvent = nullptr;
02093 
02094                 initialized = GL_FALSE;
02095                 contextCreated = GL_FALSE;
02096                 currentWindowStyle = WINDOWSTYLE_DEFAULT;
02097 
02098 #if defined( __linux )
02099                 context = 0;
02100 #endif
02101             }
```

3.1.3 Field Documentation

3.1.3.1 Atom windowManager::tWindow::AtomActionClose

atom for allowing the window to be closed

3.1.3.2 Atom windowManager::tWindow::AtomActionMaximizeHorz

atom for allowing the window to be maximized horizontally

3.1.3.3 Atom windowManager::tWindow::AtomActionMaximizeVert

atom for allowing the window to be maximized vertically

3.1.3.4 Atom windowManager::tWindow::AtomActionMinimize

atom for allowing the window to be minimized

3.1.3.5 Atom windowManager::tWindow::AtomActionResize

atom for allowing the window to be resized

3.1.3.6 Atom windowManager::tWindow::AtomActionShade

atom for allowing the window to be shaded

3.1.3.7 Atom windowManager::tWindow::AtomActive

atom for the active window

3.1.3.8 Atom windowManager::tWindow::AtomAllowedActions

atom for allowed window actions

3.1.3.9 Atom windowManager::tWindow::AtomCardinal

atom for cardinal coordinates

3.1.3.10 Atom windowManager::tWindow::AtomClose

atom for closing the window

3.1.3.11 Atom windowManager::tWindow::AtomDemandsAttention

atom for when the window demands attention

3.1.3.12 Atom windowManager::tWindow::AtomDesktopGeometry

atom for Desktop Geometry

3.1.3.13 Atom windowManager::tWindow::AtomFocused

atom for the focused state of the window

3.1.3.14 Atom windowManager::tWindow::AtomFullScreen

atom for the full screen state of the window

3.1.3.15 Atom windowManager::tWindow::AtomHidden

atom for the current hidden state of the window

3.1.3.16 Atom windowManager::tWindow::AtomHints

atom for the window decorations

3.1.3.17 Atom windowManager::tWindow::AtomIcon

atom for the icon of the window

3.1.3.18 Atom windowManager::tWindow::AtomMaxHorz

atom for the maximized horizontally state of the window

3.1.3.19 Atom windowManager::tWindow::AtomMaxVert

atom for the maximized vertically state of the window

3.1.3.20 Atom windowManager::tWindow::AtomState

atom for the state of the window

3.1.3.21 Atom windowManager::tWindow::AtomWindowType

atom for the type of window

3.1.3.22 Atom windowManager::tWindow::AtomWindowTypeDesktop

atom for the desktop window type

3.1.3.23 Atom windowManager::tWindow::AtomWindowTypeNormal

atom for the normal splash screen window type

3.1.3.24 Atom windowManager::tWindow::AtomWindowTypeSplash

atom for the splash screen window type

3.1.3.25 GLint* windowManager::tWindow::attributes

attributes of the window. RGB, depth, stencil, etc

3.1.3.26 GLuint windowManager::tWindow::colourBits

color format of the window. (defaults to 32 bit color)

3.1.3.27 GLXContext windowManager::tWindow::context

the handle to the GLX rendering context

3.1.3.28 GLboolean windowManager::tWindow::contextCreated

whether the OpenGL context has been successfully created

3.1.3.29 GLuint windowManager::tWindow::currentState

The current state of the window. these states include Normal, Minimized, Maximized and Full screen

3.1.3.30 GLuint windowManager::tWindow::currentWindowStyle

the current style of the window

3.1.3.31 GLbitfield windowManager::tWindow::decorators

enabled window decorators

3.1.3.32 GLuint windowManager::tWindow::depthBits

Size of the Depth buffer. (defaults to 8 bit depth)

3.1.3.33 onDestroyedEvent_t windowManager::tWindow::destroyedEvent

this is the callback to be used when the window has been closed in a non-programmatic fashion

3.1.3.34 onFocusEvent_t windowManager::tWindow::focusEvent

this is the callback to be used when the window has been given focus in a non-programmatic fashion

3.1.3.35 GLuint windowManager::tWindow::iD

ID of the Window. (where it belongs in the window manager)

3.1.3.36 GLboolean windowManager::tWindow::inFocus

Whether the Window is currently in focus(if it is the current window be used)

3.1.3.37 GLboolean windowManager::tWindow::initialized

whether the window has been successfully initialized

3.1.3.38 GLboolean windowManager::tWindow::isCurrentContext

whether the window is the current window being drawn to

3.1.3.39 onKeyEvent_t windowManager::tWindow::keyEvent

this is the callback to be used when a key has been pressed

3.1.3.40 GLboolean windowManager::tWindow::keys[256+1+54]

Record of keys that are either pressed or released in the respective window

3.1.3.41 onMaximizedEvent_t windowManager::tWindow::maximizedEvent

this is the callback to be used when the window has been maximized in a non-programmatic fashion

3.1.3.42 onMinimizedEvent_t windowManager::tWindow::minimizedEvent

this is the callback to be used when the window has been minimized in a non-programmatic fashion

3.1.3.43 GLboolean windowManager::tWindow::mouseButton[2+1]

Record of mouse buttons that are either presses or released

3.1.3.44 onMouseButtonEvent_t windowManager::tWindow::mouseButtonEvent

this is the callback to be used when a mouse button has been pressed

3.1.3.45 onMouseMoveEvent_t windowManager::tWindow::mouseMoveEvent

this is a callback to be used when the mouse has been moved

3.1.3.46 GLuint windowManager::tWindow::mousePosition[2]

Position of the Mouse cursor relative to the window co-ordinates

3.1.3.47 onMouseWheelEvent_t windowManager::tWindow::mouseWheelEvent

this is the callback to be used when the mouse wheel has been scrolled.

3.1.3.48 onMovedEvent_t windowManager::tWindow::movedEvent

this is the callback to be used the window has been moved in a non-programmatic fashion

3.1.3.49 const char* windowManager::tWindow::name

Name of the window

3.1.3.50 GLuint windowManager::tWindow::position[2]

Position of the Window relative to the screen co-ordinates

3.1.3.51 onResizeEvent_t windowManager::tWindow::resizeEvent

this is a callback to be used when the window has been resized in a non-programmatic fashion

3.1.3.52 GLuint windowManager::tWindow::resolution[2]

Resolution/Size of the window stored in an array

3.1.3.53 XSetWindowAttributes windowManager::tWindow::setAttributes

the attributes to be set for the window

3.1.3.54 GLboolean windowManager::tWindow::shouldClose

Whether the Window should be closing

3.1.3.55 GLuint windowManager::tWindow::stencilBits

Size of the stencil buffer, (defaults to 8 bit)

3.1.3.56 XVisualInfo* windowManager::tWindow::visualInfo

the handle to the Visual Information. similar purpose to PixelformatDesriptor

3.1.3.57 Window windowManager::tWindow::windowHandle

the X11 handle to the window. I wish they didn't name the type 'Window'

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

- TinyWindow.h

3.2 windowManager Class Reference

```
#include <TinyWindow.h>
```

Data Structures

- struct tWindow

Public Member Functions

- windowManager ()
- ~windowManager (void)

Static Public Member Functions

- static void ShutDown (void)
- static windowManager * AddWindow (const char *windowName, GLuint width=1280, GLuint height=720, GLuint colourBits=8, GLuint depthBits=8, GLuint stencilBits=8)
- static GLuint GetNumWindows (void)
- static GLboolean GetMousePositionInScreen (GLuint &x, GLuint &y)
- static GLuint * GetMousePositionInScreen (void)
- static GLboolean SetMousePositionInScreen (GLuint x, GLuint y)
- static GLuint * GetScreenResolution (void)
- static GLboolean GetScreenResolution (GLuint &width, GLuint &Height)
- static GLboolean GetWindowResolutionByName (const char *windowName, GLuint &width, GLuint &height)
- static GLboolean GetWindowResolutionByIndex (GLuint windowIndex, GLuint &width, GLuint &height)
- static GLuint * GetWindowResolutionByName (const char *windowName)
- static GLuint * GetWindowResolutionByIndex (GLuint windowIndex)
- static GLboolean SetWindowResolutionByName (const char *windowName, GLuint width, GLuint height)
- static GLboolean SetWindowResolutionByIndex (GLuint windowIndex, GLuint width, GLuint height)
- static GLboolean GetWindowPositionByName (const char *windowName, GLuint &x, GLuint &y)
- static GLboolean GetWindowPositionByIndex (GLuint windowIndex, GLuint &x, GLuint &y)
- static GLuint * GetWindowPositionByName (const char *windowName)
- static GLuint * GetWindowPositionByIndex (GLuint windowIndex)
- static GLboolean SetWindowPositionByName (const char *windowName, GLuint x, GLuint y)
- static GLboolean SetWindowPositionByName (GLuint windowIndex, GLuint x, GLuint y)
- static GLboolean GetMousePositionInWindowByName (const char *windowName, GLuint &x, GLuint &y)
- static GLboolean GetMousePositionInWindowByIndex (GLuint windowIndex, GLuint &x, GLuint &y)
- static GLuint * GetMousePositionInWindowByName (const char *windowName)
- static GLuint * GetMousePositionInWindowByIndex (GLuint windowIndex)
- static GLboolean SetMousePositionInWindowByName (const char *windowName, GLuint x, GLuint y)
- static GLboolean SetMousePositionInWindowByIndex (GLuint windowIndex, GLuint x, GLuint y)
- static GLboolean WindowGetKeyByName (const char *windowName, GLuint key)
- static GLboolean WindowGetKeyByIndex (GLuint windowIndex, GLuint key)
- static GLboolean GetWindowShouldCloseByName (const char *windowName)
- static GLboolean GetWindowShouldCloseByIndex (GLuint windowIndex)
- static GLboolean WindowSwapBuffersByName (const char *windowName)
- static GLboolean WindowSwapBuffersByIndex (GLuint windowIndex)
- static GLboolean MakeWindowCurrentContextByName (const char *windowName)
- static GLboolean MakeWindowCurrentContextByIndex (GLuint windowIndex)
- static GLboolean GetWindowIsFullScreenByName (const char *windowName)
- static GLboolean GetWindowIsFullScreenByIndex (GLuint windowIndex)
- static GLboolean SetFullScreenByName (const char *windowName, GLboolean newState)
- static GLboolean SetFullScreenByIndex (GLuint windowIndex, GLboolean newState)
- static GLboolean GetWindowIsMinimizedByName (const char *windowName)
- static GLboolean GetWindowIsMinimizedByIndex (GLuint windowIndex)
- static GLboolean MinimizeWindowByName (const char *windowName, GLboolean newState)
- static GLboolean MinimizeWindowByIndex (GLuint windowIndex, GLboolean newState)
- static GLboolean GetWindowIsMaximizedByName (const char *windowName)
- static GLboolean GetWindowIsMaximizedByIndex (GLuint windowIndex)
- static GLboolean MaximizeWindowByName (const char *windowName, GLboolean newState)
- static GLboolean MaximizeWindowByIndex (GLuint windowIndex, GLboolean newState)

- static const char * GetWindowNameByIndex (GLuint windowIndex)
- static GLuint GetWindowIndexByName (const char *windowName)
- static GLboolean SetWindowTitleBarByName (const char *windowName, const char *newTitle)
- static GLboolean SetWindowTitleBarByIndex (GLuint windowIndex, const char *newName)
- static GLboolean SetWindowIconByName (const char *windowName, const char *icon, GLuint width, GLuint height)
- static GLboolean SetWindowIconByIndex (GLuint windowIndex, const char *icon, GLuint width, GLuint height)
- static GLboolean GetWindowIsInFocusByName (const char *windowName)
- static GLboolean GetWindowIsInFocusByIndex (GLuint windowIndex)
- static GLboolean FocusWindowByName (const char *windowName, GLboolean newState)
- static GLboolean FocusWindowByIndex (GLuint windowIndex, GLboolean newState)
- static GLboolean RestoreWindowByName (const char *windowName)
- static GLboolean RestoreWindowByIndex (GLuint windowIndex)
- static GLboolean Initialize (void)
- static GLboolean IsInitialized (void)
- static void PollForEvents (void)
- static void WaitForEvents (void)
- static GLboolean RemoveWindowByName (const char *windowName)
- static GLboolean RemoveWindowByIndex (GLuint windowIndex)
- static GLboolean SetWindowStyleByName (const char *windowName, GLuint windowStyle)
- static GLboolean SetWindowStyleByIndex (GLuint windowIndex, GLuint windowStyle)
- static GLboolean EnableWindowDecoratorsByName (const char *windowname, GLbitfield decorators)
- static GLboolean EnableWindowDecoratorsByIndex (GLuint windowIndex, GLbitfield decorators)
- static GLboolean DisableWindowDecoratorByName (const char *windowName, GLbitfield decorators)
- static GLboolean DisableWindowDecoratorByIndex (GLuint windowIndex, GLbitfield decorators)
- static GLboolean SetWindowOnKeyEventByName (const char *windowName, onKeyEvent_t onKey)
- static GLboolean SetWindowOnKeyEventByIndex (GLuint windowIndex, onKeyEvent_t onKey)
- static GLboolean SetWindowOnMouseButtonEventByName (const char *windowName, onMouseButton↩ Event_t onMouseButton)
- static GLboolean SetWindowOnMouseButtonEventByIndex (GLuint windowIndex, onMouseButtonEvent_↩ t onMouseButton)
- static GLboolean SetWindowOnMouseWheelEventByName (const char *windowName, onMouseWheel↩ Event_t onMouseWheel)
- static GLboolean SetWindowOnMouseWheelEventByIndex (GLuint windowIndex, onMouseWheelEvent_↩ t onMouseWheel)
- static GLboolean SetWindowOnDestroyedByName (const char *windowName, onDestroyedEvent_t on↩ Destroyed)
- static GLboolean SetWindowOnDestroyedByIndex (GLuint windowIndex, onDestroyedEvent_t onDestroyed)
- static GLboolean SetWindowOnMaximizedByName (const char *windowName, onMaximizedEvent_t on↩ Maximized)
- static GLboolean SetWindowOnMaximizedByIndex (GLuint windowIndex, onMaximizedEvent_t on↩ Maximized)
- static GLboolean SetWindowOnMinimizedByName (const char *windowName, onMinimizedEvent_t on↩ Minimized)
- static GLboolean SetWindowOnMinimizedByIndex (GLuint windowIndex, onMinimizedEvent_t onMinimized)
- static GLboolean SetWindowOnFocusByName (const char *windowName, onFocusEvent_t onFocus)
- static GLboolean SetWindowOnFocusByIndex (GLuint windowIndex, onFocusEvent_t onFocus)
- static GLboolean SetWindowOnMovedByName (const char *windowName, onMovedEvent_t onMoved)
- static GLboolean SetWindowOnMovedByIndex (GLuint windowIndex, onMovedEvent_t onMoved)
- static GLboolean SetWindowOnResizeByName (const char *windowName, onResizeEvent_t onResize)
- static GLboolean SetWindowOnResizeByIndex (GLuint windowIndex, onResizeEvent_t onResize)
- static GLboolean SetWindowOnMouseMoveByName (const char *windowName, onMouseMoveEvent_t on↩ MouseMove)
- static GLboolean SetWindowOnMouseMoveByIndex (GLuint windowIndex, onMouseMoveEvent_t on↩ MouseMove)

Static Private Member Functions

- static tWindow * GetWindowInList (const char *windowName)
- static tWindow * GetWindowInList (GLuint windowIndex)
- static GLboolean IsValid (const char *stringParameter)
- static GLboolean IsValid (onKeyEvent_t onKeyPressed)
- static GLboolean IsValid (onMouseWheelEvent_t onMouseWheelEvent)
- static GLboolean IsValid (onMaximizedEvent_t onMaximized)
- static GLboolean IsValid (onFocusEvent_t onFocus)
- static GLboolean IsValid (onMovedEvent_t onMoved)
- static GLboolean IsValid (onMouseMoveEvent_t onMouseMove)
- static GLboolean WindowExists (GLuint windowIndex)
- static windowManager * GetInstance (void)
- static void InitializeWindow (tWindow *window)
- static void InitializeGL (tWindow *window)
- static void ShutdownWindow (tWindow *window)
- static GLboolean DoesExistByName (const char *windowName)
- static GLboolean DoesExistByIndex (GLuint windowIndex)
- static tWindow * GetWindowByName (const char *windowName)
- static tWindow * GetWindowByIndex (GLuint windowIndex)
- static tWindow * GetWindowByHandle (Window windowHandle)
- static tWindow * GetWindowByEvent (XEvent currentEvent)
- static GLboolean Linux_Initialize (void)
- static void InitializeAtomics (tWindow *window)
- static void Linux_InitializeWindow (tWindow *window)
- static GLboolean Linux_InitializeGL (tWindow *window)
- static void Linux_ShutdownWindow (tWindow *window)
- static void Linux_Shutdown (void)
- static void Linux_Fullscreen (tWindow *window)
- static void Linux_Minimize (tWindow *window)
- static void Linux_Maximize (tWindow *window)
- static void Linux_Restore (tWindow *window)
- static void Linux_Focus (tWindow *window, GLboolean newFocusState)
- static void Linux_SetMousePosition (tWindow *window)
- static void Linux_SetWindowPosition (tWindow *window)
- static void Linux_SetWindowResolution (tWindow *window)
- static void Linux_ProcessEvents (XEvent currentEvent)
- static void Linux_PollForEvents (void)
- static void Linux_WaitForEvents (void)
- static void Linux_SetMousePositionInScreen (GLuint x, GLuint y)
- static Display * GetDisplay (void)
- static const char * Linux_GetEventType (XEvent currentEvent)
- static GLuint Linux_TranslateKey (GLuint keySymbol)
- static void Linux_EnableDecorators (tWindow *window, GLbitfield decorators)
- static void Linux_DisableDecorators (tWindow *window, GLbitfield decorators)
- static void Linux_SetWindowStyle (tWindow *window, GLuint windowStyle)
- static void Linux_SetWindowIcon (tWindow *window, const char *icon, GLuint width, GLuint height)
- static GLXFBConfig GetBestFrameBufferConfig (tWindow *givenWindow)

Private Attributes

- std::list< tWindow * > windowList
- GLuint screenResolution [2]
- GLuint screenMousePosition [2]
- GLboolean isInitialized
- const Display * currentDisplay
- XEvent currentEvent

Static Private Attributes

- static windowManager * instance = nullptr

3.2.1 Detailed Description

3.2.2 Constructor & Destructor Documentation

3.2.2.1 windowManager::windowManager ( ) [inline]

```
00346 {}
```

3.2.2.2 windowManager::~windowManager (void) [inline]

shutdown and delete all windows in the manager

```
00352     {
00353         if ( !GetInstance()->windowList.empty() )
00354         {
00355 #if defined( _MSC_VER )
00356             for each( auto CurrentWindow in GetInstance()->windowList )
00357 #else
00358             for ( auto CurrentWindow : GetInstance()->windowList )
00359 #endif
00360             {
00361                 delete CurrentWindow;
00362             }
00363             GetInstance()->windowList.clear();
00364         }
00365     }
```

3.2.3 Member Function Documentation

3.2.3.1 static windowManager* windowManager::AddWindow (const char * *windowName*, GLuint *width* = 1280, GLuint *height* = 720, GLuint *colourBits* = 8, GLuint *depthBits* = 8, GLuint *stencilBits* = 8) [inline], [static]

use this to add a window to the manager. returns a pointer to the manager which allows for the easy creation of multiple windows < if the window is being used without being initialized

```
00397     {
00398         if ( GetInstance()->IsInitialized() )
00399         {
00400             if ( IsValid( windowName ) )
00401             {
00402                 tWindow* newWindow = new tWindow;
00403                 newWindow->name = windowName;
00404                 newWindow->resolution[0] = width;
00405                 newWindow->resolution[1] = height;
00406                 newWindow->colourBits = colourBits;
00407                 newWindow->depthBits = depthBits;
00408                 newWindow->stencilBits = stencilBits;
00409
00410                 GetInstance()->windowList.push_back( newWindow );
00411                 newWindow->iD = GetNumWindows() - 1;
00412
00413                 InitializeWindow( newWindow );
00414
00415                 return GetInstance();
00416             }
00417             return nullptr;
00418         }
00419
00420         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00421         return nullptr;
00422     }
```

3.2.3.2 static GLboolean windowManager::DisableWindowDecoratorByIndex (GLuint *windowIndex*, GLbitfield *decorators*) [inline], [static]

< if the window is being used without being initialized

```
01759     {
01760         if ( GetInstance()->IsInitialized() )
01761         {
01762             if ( DoesExistByIndex( windowIndex ) )
01763             {
01764 #if defined( _WIN32 ) || defined( _WIN64 )
01765                 Windows_DisableDecorators( GetWindowByIndex( windowIndex ), decorators );
01766 #else
01767                 Linux_DisableDecorators( GetWindowByIndex(
      windowIndex ), decorators );
01768 #endif
01769                 return FOUNDATION_OK;
01770             }
01771             return FOUNDATION_ERROR;
01772         }
01773         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01774         return FOUNDATION_ERROR;
01775     }
```

3.2.3.3 static GLboolean windowManager::DisableWindowDecoratorByName (const char * *windowName*, GLbitfield *decorators*) [inline], [static]

< if the window is being used without being initialized

```
01741     {
01742         if ( GetInstance()->IsInitialized() )
01743         {
01744             if ( DoesExistByName( windowName ) )
01745             {
01746 #if defined( _WIN32 ) || defined( _WIN64 )
01747                 Windows_DisableDecorators( GetWindowByName( windowName ), decorators );
01748 #else
01749                 Linux_DisableDecorators( GetWindowByName( windowName
      ), decorators );
01750 #endif
01751                 return FOUNDATION_OK;
01752             }
01753             return FOUNDATION_ERROR;
01754         }
01755         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01756         return FOUNDATION_ERROR;
01757     }
```

3.2.3.4 static GLboolean windowManager::DoesExistByIndex (GLuint *windowIndex*) [inline], [static], [private]

< if an invalid window index was given

```
02352     {
02353         if ( GetInstance()->IsInitialized() )
02354         {
02355             if ( windowIndex <= ( GetInstance()->windowList.size() - 1 ) )
02356             {
02357                 return FOUNDATION_OK;
02358             }
02359
02360             PrintErrorMessage(
      TINYWINDOW_ERROR_INVALID_WINDOW_INDEX );
02361             return FOUNDATION_ERROR;
02362         }
02363
02364         return FOUNDATION_ERROR;
02365     }
```

3.2.3.5 static GLboolean windowManager::DoesExistByName (const char * *windowName*) [inline], [static], [private]

< if an invalid window name was given

```
02327     {
02328         if ( GetInstance()->IsInitialized() )
02329         {
02330             if ( IsValid( windowName ) )
02331             {
02332 #if defined( _MSC_VER )
02333                 for each( auto window in GetInstance()->windowList )
02334 #else
02335                 for ( auto window : GetInstance()->windowList )
02336 #endif
02337                 {
02338                     if( !strcmp( window->name, windowName ) )
02339                     {
02340                         return GL_TRUE;
02341                     }
02342                 }
02343             }
02344             PrintErrorMessage(
      TINYWINDOW_ERROR_INVALID_WINDOW_NAME );
02345             return GL_FALSE;
02346         }
02347 
02348         return GL_FALSE;
02349     }
```

3.2.3.6 static GLboolean windowManager::EnableWindowDecoratorsByIndex (GLuint *windowIndex*, GLbitfield *decorators*) [inline], [static]

< if the window is being used without being initialized

```
01722     {
01723         if ( GetInstance()->IsInitialized() )
01724         {
01725             if ( DoesExistByIndex( windowIndex ) )
01726             {
01727 #if defined( _WIN32 ) || defined( _WIN64 )
01728                 Windows_EnableDecorators( GetWindowByIndex( windowIndex ), decorators );
01729 #else
01730                 Linux_EnableDecorators( GetWindowByIndex( windowIndex
      ), decorators );
01731 #endif
01732                 return FOUNDATION_OK;
01733             }
01734             return FOUNDATION_ERROR;
01735         }
01736         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01737         return FOUNDATION_ERROR;
01738     }
```

3.2.3.7 static GLboolean windowManager::EnableWindowDecoratorsByName (const char * *windowname*, GLbitfield *decorators*) [inline], [static]

< if the window is being used without being initialized

```
01704     {
01705         if ( GetInstance()->IsInitialized() )
01706         {
01707             if ( DoesExistByName( windowname ) )
01708             {
01709 #if defined( _WIN32 ) || defined( _WIN64 )
01710                 Windows_EnableDecorators( GetWindowByName( windowname ), decorators );
01711 #else
01712                 Linux_EnableDecorators( GetWindowByName( windowname ),
      decorators );
01713 #endif
01714                 return FOUNDATION_OK;
01715             }
```

```
01716             return FOUNDATION_ERROR;
01717         }
01718         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01719         return FOUNDATION_ERROR;
01720     }
```

3.2.3.8 static GLboolean windowManager::FocusWindowByIndex (GLuint *windowIndex*, GLboolean *newState*) [inline], [static]

< if a window tries to use a graphical function without a context

```
01530     {
01531         if ( GetInstance()->IsInitialized() )
01532         {
01533             if ( DoesExistByIndex( windowIndex ) )
01534             {
01535 #if defined( _WIN32 ) || defined( _WIN64 )
01536                 Windows_Focus( GetWindowByIndex( windowIndex ), newState );
01537 #else
01538                 Linux_Focus( GetWindowByIndex( windowIndex ), newState );
01539 #endif
01540                 return FOUNDATION_OK;
01541             }
01542             return FOUNDATION_ERROR;
01543         }
01544         PrintErrorMessage( TINYWINDOW_ERROR_NO_CONTEXT );
01545         return FOUNDATION_ERROR;
01546     }
```

3.2.3.9 static GLboolean windowManager::FocusWindowByName (const char * *windowName*, GLboolean *newState*) [inline], [static]

< if the window is being used without being initialized

```
01512     {
01513         if ( GetInstance()->IsInitialized() )
01514         {
01515             if ( DoesExistByName( windowName ) )
01516             {
01517 #if defined( _WIN32 ) || defined( _WIN64 )
01518                 Windows_Focus( GetWindowByName( windowName ), newState );
01519 #else
01520                 Linux_Focus( GetWindowByName( windowName ), newState );
01521 #endif
01522                 return FOUNDATION_OK;
01523             }
01524             return FOUNDATION_ERROR;
01525         }
01526         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01527         return FOUNDATION_ERROR;
01528     }
```

3.2.3.10 static GLXFBConfig windowManager::GetBestFrameBufferConfig (tWindow * *givenWindow*) [inline], [static], [private]

```
04919     {
04920         const GLint visualAttributes[] =
04921         {
04922             GLX_X_RENDERABLE, GL_TRUE,
04923             GLX_DRAWABLE_TYPE, GLX_WINDOW_BIT,
04924             GLX_X_VISUAL_TYPE, GLX_TRUE_COLOR,
04925             GLX_RED_SIZE, givenWindow->colourBits,
04926             GLX_GREEN_SIZE, givenWindow->colourBits,
04927             GLX_BLUE_SIZE, givenWindow->colourBits,
04928             GLX_ALPHA_SIZE, givenWindow->colourBits,
04929             GLX_DEPTH_SIZE, givenWindow->depthBits,
04930             GLX_STENCIL_SIZE, givenWindow->stencilBits,
04931             GLX_DOUBLEBUFFER, GL_TRUE,
04932             None
```

```
04933         };
04934
04935         GLint frameBufferCount;
04936         GLuint bestBufferConfig, bestNumSamples = 0;
04937         GLXFBConfig* configs = glXChooseFBConfig( GetDisplay(), 0, visualAttributes, &
      frameBufferCount );
04938
04939         for ( GLuint currentConfig = 0; currentConfig < frameBufferCount; currentConfig++ )
04940         {
04941             XVisualInfo* visualInfo = glXGetVisualFromFBConfig( GetDisplay(), configs[
      currentConfig] );
04942
04943             if ( visualInfo )
04944             {
04945                 //printf( "%i %i %i\n", VisInfo->depth, VisInfo->bits_per_rgb, VisInfo->colormap_size );
04946                 GLint samples, sampleBuffer;
04947                 glXGetFBConfigAttrib( GetDisplay(), configs[currentConfig], GLX_SAMPLE_BUFFERS, &
      sampleBuffer );
04948                 glXGetFBConfigAttrib( GetDisplay(), configs[currentConfig], GLX_SAMPLES, &samples
       );
04949
04950                 if ( sampleBuffer && samples > -1 )
04951                 {
04952                     bestBufferConfig = currentConfig;
04953                     bestNumSamples = samples;
04954                 }
04955             }
04956
04957             XFree( visualInfo );
04958         }
04959
04960         GLXFBConfig BestConfig = configs[bestBufferConfig];
04961
04962         XFree( configs );
04963
04964         return BestConfig;
04965     }
```

3.2.3.11 static Display* windowManager::GetDisplay (void) [inline], [static], [private]

```
04318     {
04319         return GetInstance()->currentDisplay;
04320     }
```

3.2.3.12 static windowManager* windowManager::GetInstance (void) [inline], [static], [private]

```
02286     {
02287         if ( windowManager::instance == nullptr )
02288         {
02289             windowManager::instance = new windowManager();
02290             return windowManager::instance;
02291         }
02292
02293         else
02294         {
02295             return windowManager::instance;
02296         }
02297     }
```

3.2.3.13 static GLboolean windowManager::GetMousePositionInScreen (GLuint & *x*, GLuint & *y*) [inline], [static]

return the mouse position in screen co-ordinates < if the window is being used without being initialized

```
00442     {
00443         if ( GetInstance()->IsInitialized() )
00444         {
00445             x = GetInstance()->screenMousePosition[0];
00446             y = GetInstance()->screenMousePosition[1];
00447             return FOUNDATION_OK;
00448         }
00449
00450         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00451         return FOUNDATION_ERROR;
00452     }
```

3.2.3.14 static GLuint* windowManager::GetMousePositionInScreen (void) [inline], [static]

return the mouse position in screen co-ordinates < if the window is being used without being initialized

```
00457     {
00458         if ( GetInstance()->IsInitialized() )
00459         {
00460             return GetInstance()->screenMousePosition;
00461         }
00462 
00463         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00464         return nullptr;
00465     }
```

3.2.3.15 static GLboolean windowManager::GetMousePositionInWindowByIndex (GLuint *windowIndex*, GLuint & *x*, GLuint & *y*) [inline], [static]

return the mouse position relative to the given window's co-ordinates by setting X and Y < if the window is being used without being initialized

```
00808     {
00809         if ( GetInstance()->IsInitialized() )
00810         {
00811             if ( DoesExistByIndex( windowIndex ) )
00812             {
00813                 x = GetWindowByIndex( windowIndex )->
      mousePosition[0];
00814                 y = GetWindowByIndex( windowIndex )->
      mousePosition[1];
00815                 return FOUNDATION_OK;
00816             }
00817             return FOUNDATION_ERROR;
00818         }
00819         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00820         return FOUNDATION_ERROR;
00821     }
```

3.2.3.16 static GLuint* windowManager::GetMousePositionInWindowByIndex (GLuint *windowIndex*) [inline], [static]

return the mouse Position relative to the given window's co-ordinates as an array < if the window is being used without being initialized

```
00843     {
00844         if ( GetInstance()->IsInitialized() )
00845         {
00846             if ( DoesExistByIndex( windowIndex ) )
00847             {
00848                 return GetWindowByIndex( windowIndex )->
      mousePosition;
00849             }
00850             return nullptr;
00851         }
00852         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00853         return nullptr;
00854     }
```

3.2.3.17 static GLboolean windowManager::GetMousePositionInWindowByName (const char * *windowName*, GLuint & *x*, GLuint & *y*) [inline], [static]

return the mouse Position relative to the given window's co-ordinates by setting X and Y < if the window is being used without being initialized

```
00790     {
00791         if ( GetInstance()->IsInitialized() )
00792         {
00793             if ( DoesExistByName( windowName ) )
00794             {
00795                 x = GetWindowByName( windowName )->mousePosition[0];
00796                 y = GetWindowByName( windowName )->mousePosition[1];
00797                 return FOUNDATION_OK;
00798             }
00799             return FOUNDATION_ERROR;
00800         }
00801         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00802         return FOUNDATION_ERROR;
00803     }
```

3.2.3.18 static GLuint* windowManager::GetMousePositionInWindowByName (const char * *windowName*) [inline], [static]

return the mouse Position relative to the given window's co-ordinates as an array < if the window is being used without being initialized

```
00827     {
00828         if ( GetInstance()->IsInitialized() )
00829         {
00830             if ( DoesExistByName( windowName ) )
00831             {
00832                 return GetWindowByName( windowName )->
      mousePosition;
00833             }
00834             return FOUNDATION_ERROR;
00835         }
00836         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00837         return nullptr;
00838     }
```

3.2.3.19 static GLuint windowManager::GetNumWindows (void) [inline], [static]

return the total amount of windows the manager has < if the window is being used without being initialized

```
00428     {
00429         if ( GetInstance()->IsInitialized() )
00430         {
00431             return GetInstance()->windowList.size();
00432         }
00433
00434         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00435         return FOUNDATION_ERROR;
00436     }
```

3.2.3.20 static GLuint* windowManager::GetScreenResolution (void) [inline], [static]

return the Resolution of the current screen < if the window is being used without being initialized

```
00492     {
00493         if ( GetInstance()->IsInitialized() )
00494         {
00495 #if defined( _WIN32 ) || defined( _WIN64 )
00496             RECT screen;
00497             HWND desktop = GetDesktopWindow();
00498             GetWindowRect( desktop, &screen );
00499
00500             GetInstance()->screenResolution[0] = screen.right;
00501             GetInstance()->screenResolution[1] = screen.bottom;
00502             return GetInstance()->screenResolution;
00503
00504 #else
00505             GetInstance()->screenResolution[0] = WidthOfScreen(
      XDefaultScreenOfDisplay( GetInstance()->currentDisplay ) );
```

```
00506             GetInstance()->screenResolution[1] = HeightOfScreen(
      XDefaultScreenOfDisplay( GetInstance()->currentDisplay ) );
00507
00508             return GetInstance()->screenResolution;
00509 #endif
00510         }
00511
00512         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
       );
00513         return nullptr;
00514     }
```

3.2.3.21 static GLboolean windowManager::GetScreenResolution (GLuint & *width*, GLuint & *Height*) [inline], [static]

return the Resolution of the current screen < if the window is being used without being initialized

```
00519     {
00520         if ( GetInstance()->IsInitialized() )
00521         {
00522 #if defined( _WIN32 ) || defined( _WIN64 )
00523             RECT screen;
00524             HWND desktop = GetDesktopWindow();
00525             GetWindowRect( desktop, &screen );
00526             width = screen.right;
00527             Height = screen.bottom;
00528 #else
00529             width = WidthOfScreen( XDefaultScreenOfDisplay( GetInstance()->
      currentDisplay ) );
00530             Height = HeightOfScreen( XDefaultScreenOfDisplay( GetInstance()->
      currentDisplay ) );
00531
00532             GetInstance()->screenResolution[0] = width;
00533             GetInstance()->screenResolution[1] = Height;
00534 #endif
00535             return FOUNDATION_OK;
00536         }
00537
00538         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
       );
00539         return FOUNDATION_ERROR;
00540     }
```

3.2.3.22 static tWindow* windowManager::GetWindowByEvent (XEvent *currentEvent*) [inline], [static], [private]

```
03464     {
03465         switch( currentEvent.type )
03466         {
03467             case Expose:
03468             {
03469                 return GetWindowByHandle( currentEvent.xexpose.window );
03470             }
03471
03472             case DestroyNotify:
03473             {
03474                 return GetWindowByHandle( currentEvent.xdestroywindow.window )
      ;
03475             }
03476
03477             case CreateNotify:
03478             {
03479                 return GetWindowByHandle( currentEvent.xcreatewindow.window );
03480             }
03481
03482             case KeyPress:
03483             {
03484                 return GetWindowByHandle( currentEvent.xkey.window );
03485             }
03486
03487             case KeyRelease:
03488             {
03489                 return GetWindowByHandle( currentEvent.xkey.window );
03490             }
03491
03492             case ButtonPress:
03493             {
```

```
03494                 return GetWindowByHandle( currentEvent.xbutton.window );
03495             }
03496
03497             case ButtonRelease:
03498             {
03499                 return GetWindowByHandle( currentEvent.xbutton.window );
03500             }
03501
03502             case MotionNotify:
03503             {
03504                 return GetWindowByHandle( currentEvent.xmotion.window );
03505             }
03506
03507             case FocusIn:
03508             {
03509                 return GetWindowByHandle( currentEvent.xfocus.window );
03510             }
03511
03512             case FocusOut:
03513             {
03514                 return GetWindowByHandle( currentEvent.xfocus.window );
03515             }
03516
03517             case ResizeRequest:
03518             {
03519                 return GetWindowByHandle( currentEvent.xresizerequest.window )
    ;
03520             }
03521
03522             case ConfigureNotify:
03523             {
03524                 return GetWindowByHandle( currentEvent.xconfigure.window );
03525             }
03526
03527             case PropertyNotify:
03528             {
03529                 return GetWindowByHandle( currentEvent.xproperty.window );
03530             }
03531
03532             case GravityNotify:
03533             {
03534                 return GetWindowByHandle( currentEvent.xgravity.window );
03535             }
03536
03537             case ClientMessage:
03538             {
03539                 return GetWindowByHandle( currentEvent.xclient.window );
03540             }
03541
03542             case VisibilityNotify:
03543             {
03544                 return GetWindowByHandle( currentEvent.xvisibility.window );
03545             }
03546
03547             default:
03548             {
03549                 return nullptr;
03550             }
03551         }
03552     }
```

3.2.3.23 static tWindow* windowManager::GetWindowByHandle (Window *windowHandle*) [inline], [static], [private]

```
03452     {
03453         for( auto iter : GetInstance()->windowList )
03454         {
03455             if ( iter->windowHandle == windowHandle )
03456             {
03457                 return iter;
03458             }
03459         }
03460         return nullptr;
03461     }
```

3.2.3.24 static tWindow* windowManager::GetWindowByIndex (GLuint *windowIndex*) [inline], [static], [private]

```
02386     {
```

```
02387         if ( windowIndex <= GetInstance()->windowList.size() - 1 )
02388         {
02389             return GetWindowInList( windowIndex );
02390         }
02391         return nullptr;
02392     }
```

3.2.3.25 static tWindow* windowManager::GetWindowByName (const char * *windowName*) [inline], [static], [private]

```
02368     {
02369 #if defined( _MSC_VER )
02370             for each( auto window in GetInstance()->windowList )
02371 #else
02372             for( auto window : GetInstance()->windowList )
02373 #endif
02374             {
02375                 if ( !strcmp( window->name, windowName ) )
02376                 {
02377                     return window;
02378                 }
02379             }
02380 
02381             return nullptr;
02382     }
```

3.2.3.26 static GLuint windowManager::GetWindowIndexByName (const char * *windowName*) [inline], [static]

< if the window is being used without being initialized

```
01392     {
01393         if ( GetInstance()->IsInitialized() )
01394         {
01395             if ( DoesExistByName( windowName ) )
01396             {
01397                 return GetWindowByName( windowName )->iD;
01398             }
01399             return FOUNDATION_ERROR;
01400         }
01401         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01402         return FOUNDATION_ERROR;
01403     }
```

3.2.3.27 static tWindow* windowManager::GetWindowInList (const char * *windowName*) [inline], [static], [private]

```
02194     {
02195         if ( IsValid( windowName ) )
02196         {
02197 #if defined( _MSC_VER )
02198             for each ( auto window in GetInstance()->windowList )
02199 #else
02200             for( auto window : GetInstance()->windowList )
02201 #endif
02202             {
02203                 if( window->name == windowName )
02204                 {
02205                     return window;
02206                 }
02207             }
02208 
02209             return nullptr;
02210         }
02211 
02212         return nullptr;
02213     }
```

3.2.3.28 static tWindow* windowManager::GetWindowInList (GLuint *windowIndex*) [inline], [static], [private]

```
02216     {
02217         if ( WindowExists( windowIndex ) )
02218         {
02219 #if defined( _MSC_VER )
02220             for each ( auto window in GetInstance()->windowList )
02221             {
02222                 if ( window->iD == windowIndex )
02223                 {
02224                     return window;
02225                 }
02226             }
02227 #else
02228             for ( auto window : GetInstance()->windowList )
02229             {
02230                 if ( window->iD == windowIndex )
02231                 {
02232                     return window;
02233                 }
02234             }
02235 #endif
02236
02237             return nullptr;
02238         }
02239
02240         return nullptr;
02241     }
```

3.2.3.29 static GLboolean windowManager::GetWindowIsFullScreenByIndex (GLuint *windowIndex*) [inline], [static]

return whether the given window is in fullscreen mode < the window is currently full screen

< if a window tries to use a graphical function without a context

```
01088     {
01089         if ( GetInstance()->IsInitialized() )
01090         {
01091             if ( DoesExistByIndex( windowIndex ) )
01092             {
01093                 return ( GetWindowByIndex( windowIndex )->currentState ==
      WINDOWSTATE_FULLSCREEN );
01094             }
01095
01096             return FOUNDATION_ERROR;
01097         }
01098         PrintErrorMessage( TINYWINDOW_ERROR_NO_CONTEXT );
01099         return FOUNDATION_ERROR;
01100     }
```

3.2.3.30 static GLboolean windowManager::GetWindowIsFullScreenByName (const char * *windowName*) [inline], [static]

return whether the given window is in fullscreen mode < the window is currently full screen

< if a window tries to use a graphical function without a context

```
01071     {
01072         if ( GetInstance()->IsInitialized() )
01073         {
01074             if ( DoesExistByName( windowName ) )
01075             {
01076                 return ( GetWindowByName( windowName )->currentState ==
      WINDOWSTATE_FULLSCREEN );
01077             }
01078
01079             return FOUNDATION_ERROR;
01080         }
01081         PrintErrorMessage( TINYWINDOW_ERROR_NO_CONTEXT );
01082         return FOUNDATION_ERROR;
01083     }
```

3.2.3.31 static GLboolean windowManager::GetWindowIsInFocusByIndex (GLuint *windowIndex*) [inline], [static]

< if the window is being used without being initialized

```
01498     {
01499         if ( GetInstance()->IsInitialized() )
01500         {
01501             if ( DoesExistByIndex( windowIndex ) )
01502             {
01503                 return GetWindowByIndex( windowIndex )->inFocus;
01504             }
01505             return FOUNDATION_ERROR;
01506         }
01507 
01508         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01509         return FOUNDATION_ERROR;
01510     }
```

3.2.3.32 static GLboolean windowManager::GetWindowIsInFocusByName (const char * *windowName*) [inline], [static]

< if the window is being used without being initialized

```
01485     {
01486         if ( GetInstance()->IsInitialized() )
01487         {
01488             if ( DoesExistByName( windowName ) )
01489             {
01490                 return GetWindowByName( windowName )->inFocus;
01491             }
01492             return FOUNDATION_ERROR;
01493         }
01494         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01495         return FOUNDATION_ERROR;
01496     }
```

3.2.3.33 static GLboolean windowManager::GetWindowIsMaximizedByIndex (GLuint *windowIndex*) [inline], [static]

return whether the given window is currently maximized < the window is currently maximized

< if the window is being used without being initialized

```
01304     {
01305         if ( GetInstance()->IsInitialized() )
01306         {
01307             if ( DoesExistByIndex( windowIndex ) )
01308             {
01309                 return ( GetWindowByIndex( windowIndex )->currentState ==
     WINDOWSTATE_MAXIMIZED );
01310             }
01311             return FOUNDATION_ERROR;
01312         }
01313         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01314         return FOUNDATION_ERROR;
01315     }
```

3.2.3.34 static GLboolean windowManager::GetWindowIsMaximizedByName (const char * *windowName*) [inline], [static]

return whether the current window is currently maximized < the window is currently maximized

< if the window is being used without being initialized

```
01287     {
01288         if ( GetInstance()->IsInitialized() )
01289         {
01290             if ( DoesExistByName( windowName ) )
01291             {
01292                 return ( GetWindowByName( windowName )->currentState ==
      WINDOWSTATE_MAXIMIZED );
01293             }
01294 
01295             return FOUNDATION_ERROR;
01296         }
01297         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01298         return FOUNDATION_ERROR;
01299     }
```

3.2.3.35 static GLboolean windowManager::GetWindowIsMinimizedByIndex (GLuint *windowIndex*) [inline], [static]

returns whether the given window is minimized < the window is currently minimized

< if the window is being used without being initialized

```
01196     {
01197         if ( GetInstance()->IsInitialized() )
01198         {
01199             if ( DoesExistByIndex( windowIndex ) )
01200             {
01201                 return ( GetWindowByIndex( windowIndex )->currentState ==
      WINDOWSTATE_MINIMIZED );
01202             }
01203             return FOUNDATION_ERROR;
01204         }
01205         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01206         return FOUNDATION_ERROR;
01207     }
```

3.2.3.36 static GLboolean windowManager::GetWindowIsMinimizedByName (const char * *windowName*) [inline], [static]

returns whether the given window is minimized < the window is currently minimized

< if the window is being used without being initialized

```
01180     {
01181         if ( GetInstance()->IsInitialized() )
01182         {
01183             if ( DoesExistByName( windowName ) )
01184             {
01185                 return ( GetWindowByName( windowName )->currentState ==
      WINDOWSTATE_MINIMIZED );
01186             }
01187             return FOUNDATION_ERROR;
01188         }
01189         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01190         return FOUNDATION_ERROR;
01191     }
```

3.2.3.37 static const char* windowManager::GetWindowNameByIndex (GLuint *windowIndex*) [inline], [static]

gets and sets for window name and index < if the window is being used without being initialized

```
01379     {
01380         if ( GetInstance()->IsInitialized() )
01381         {
01382             if ( DoesExistByIndex( windowIndex ) )
01383             {
01384                 return GetWindowByIndex( windowIndex )->name;
```

```
01385             }
01386             return FOUNDATION_ERROR;
01387         }
01388         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01389         return FOUNDATION_ERROR;
01390     }
```

3.2.3.38 static GLboolean windowManager::GetWindowPositionByIndex (GLuint *windowIndex*, GLuint & *x*, GLuint & *y*) [inline], [static]

return the Position of the given window relative to screen co-ordinates by setting X and Y < if the window is being used without being initialized

```
00688     {
00689         if ( GetInstance()->IsInitialized() )
00690         {
00691             if ( DoesExistByIndex( windowIndex ) )
00692             {
00693                 x = GetWindowByIndex( windowIndex )->position[0];
00694                 y = GetWindowByIndex( windowIndex )->position[1];
00695                 return FOUNDATION_OK;
00696             }
00697             return FOUNDATION_ERROR;
00698         }
00699         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00700         return FOUNDATION_ERROR;
00701     }
```

3.2.3.39 static GLuint* windowManager::GetWindowPositionByIndex (GLuint *windowIndex*) [inline], [static]

return the Position of the given window relative to screen co-ordinates as an array < if the window is being used without being initialized

```
00725     {
00726         if ( GetInstance()->IsInitialized() )
00727         {
00728             if ( DoesExistByIndex( windowIndex ) )
00729             {
00730                 return GetWindowByIndex( windowIndex )->position;
00731             }
00732             return nullptr;
00733         }
00734         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00735         return nullptr;
00736     }
```

3.2.3.40 static GLboolean windowManager::GetWindowPositionByName (const char * *windowName*, GLuint & *x*, GLuint & *y*) [inline], [static]

return the Position of the given window relative to screen co-ordinates by setting X and Y < if the window is being used without being initialized

```
00669     {
00670         if ( GetInstance()->IsInitialized() )
00671         {
00672             if ( DoesExistByName( windowName ) )
00673             {
00674                 x = GetWindowByName( windowName )->position[0];
00675                 y = GetWindowByName( windowName )->position[1];
00676                 return FOUNDATION_OK;
00677             }
00678
00679             return FOUNDATION_ERROR;
00680         }
00681         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00682         return FOUNDATION_ERROR;
00683     }
```

3.2.3.41 static GLuint* windowManager::GetWindowPositionByName (const char * *windowName*) [inline], [static]

return the Position of the given window relative to screen co-ordinates as an array < if the window is being used without being initialized

```
00707     {
00708         if ( GetInstance()->IsInitialized() )
00709         {
00710             if ( DoesExistByName( windowName ) )
00711             {
00712                 return GetWindowByName( windowName )->position;
00713             }
00714 
00715             return nullptr;
00716         }
00717 
00718         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00719         return nullptr;
00720     }
```

3.2.3.42 static GLboolean windowManager::GetWindowResolutionByIndex (GLuint *windowIndex,* GLuint & *width,* GLuint & *height*) [inline], [static]

return the Resolution of the given window by setting width and height < if the window is being used without being initialized

```
00564     {
00565         if ( GetInstance()->IsInitialized() )
00566         {
00567             if ( DoesExistByIndex( windowIndex ) )
00568             {
00569                 width = GetWindowByIndex( windowIndex )->
      resolution[0];
00570                 height = GetWindowByIndex( windowIndex )->
      resolution[1];
00571 
00572                 return FOUNDATION_OK;
00573             }
00574             return FOUNDATION_ERROR;
00575         }
00576 
00577         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00578         return FOUNDATION_ERROR;
00579     }
```

3.2.3.43 static GLuint* windowManager::GetWindowResolutionByIndex (GLuint *windowIndex*) [inline], [static]

return the Resolution of the Given Window as an array of doubles < if the window is being used without being initialized

```
00602     {
00603         if ( GetInstance()->IsInitialized() )
00604         {
00605             if ( DoesExistByIndex( windowIndex ) )
00606             {
00607                 return GetWindowByIndex( windowIndex )->
      resolution;
00608             }
00609             return nullptr;
00610         }
00611 
00612         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00613         return nullptr;
00614     }
```

3.2.3.44 static GLboolean windowManager::GetWindowResolutionByName (const char * *windowName*, GLuint & *width*, GLuint & *height*) [inline], [static]

return the Resolution of the given window by setting width and height < if the window is being used without being initialized

```
00546     {
00547         if ( GetInstance()->IsInitialized() )
00548         {
00549             if ( DoesExistByName( windowName ) )
00550             {
00551                 width = GetWindowByName( windowName )->resolution[0];
00552                 height = GetWindowByName( windowName )->
      resolution[1];
00553                 return FOUNDATION_ERROR;
00554             }
00555             return FOUNDATION_ERROR;
00556         }
00557         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00558         return FOUNDATION_ERROR;
00559     }
```

3.2.3.45 static GLuint* windowManager::GetWindowResolutionByName (const char * *windowName*) [inline], [static]

return the Resolution of the given Window as an array of doubles < if the window is being used without being initialized

```
00585     {
00586         if ( GetInstance()->IsInitialized() )
00587         {
00588             if ( DoesExistByName( windowName ) )
00589             {
00590                 return GetWindowByName( windowName )->resolution;
00591             }
00592             return nullptr;
00593         }
00594
00595         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00596         return nullptr;
00597     }
```

3.2.3.46 static GLboolean windowManager::GetWindowShouldCloseByIndex (GLuint *windowIndex*) [inline], [static]

return whether the given window should be closing < if the window is being used without being initialized

```
00959     {
00960         if ( GetInstance()->IsInitialized() )
00961         {
00962             if ( DoesExistByIndex( windowIndex ) )
00963             {
00964                 return GetWindowByIndex( windowIndex )->
      shouldClose;
00965             }
00966             return FOUNDATION_ERROR;
00967         }
00968
00969         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00970         return FOUNDATION_ERROR;
00971     }
```

3.2.3.47 static GLboolean windowManager::GetWindowShouldCloseByName (const char * *windowName*) [inline], [static]

return whether the given window should be closing < if the window is being used without being initialized

```
00942     {
00943         if ( GetInstance()->IsInitialized() )
00944         {
00945             if ( DoesExistByName( windowName ) )
00946             {
00947                 return GetWindowByName( windowName )->shouldClose;
00948             }
00949             return FOUNDATION_ERROR;
00950         }
00951 
00952         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00953         return FOUNDATION_ERROR;
00954     }
```

3.2.3.48 static GLboolean windowManager::Initialize (void) [inline], [static]

```
01588     {
01589         GetInstance()->isInitialized = GL_FALSE;
01590 #if defined( _WIN32 ) || defined( _WIN64 )
01591         return Windows_Initialize();
01592 #else
01593         return Linux_Initialize();
01594 #endif
01595     }
```

3.2.3.49 static void windowManager::InitializeAtomics (tWindow * *window*) [inline], [static], [private]

```
03574     {
03575         GLuint display = windowManager::GetDisplay();
03576         window->AtomState = XInternAtom( display, "_NET_WM_STATE", GL_FALSE );
03577         window->AtomFullScreen = XInternAtom( display, "_NET_WM_STATE_FULLSCREEN", GL_FALSE );
03578         window->AtomMaxHorz = XInternAtom( display, "_NET_WM_STATE_MAXIMIZED_HORZ", GL_FALSE );
03579         window->AtomMaxVert = XInternAtom( display, "_NET_WM_STATE_MAXIMIZED_VERT", GL_FALSE );
03580         window->AtomClose = XInternAtom( display, "WM_DELETE_WINDOW", GL_FALSE );
03581         window->AtomHidden = XInternAtom( display, "_NET_WM_STATE_HIDDEN", GL_FALSE );
03582         window->AtomActive = XInternAtom( display, "_NET_ACTIVE_WINDOW", GL_FALSE );
03583         window->AtomDemandsAttention = XInternAtom( display, "_NET_WM_STATE_DEMANDS_ATTENTION", GL_FALSE );
03584         window->AtomFocused = XInternAtom( display, "_NET_WM_STATE_FOCUSED", GL_FALSE );
03585         window->AtomCardinal = XInternAtom( display, "CARDINAL", GL_FALSE );
03586         window->AtomIcon = XInternAtom( display, "_NET_WM_ICON", GL_FALSE );
03587         window->AtomHints = XInternAtom( display, "_MOTIF_WM_HINTS", GL_TRUE );
03588 
03589         window->AtomWindowType = XInternAtom( display, "_NET_WM_WINDOW_TYPE", GL_FALSE );
03590         window->AtomWindowTypeDesktop = XInternAtom( display, "_NET_WM_WINDOW_TYPE_UTILITY", GL_FALSE );
03591         window->AtomWindowTypeSplash = XInternAtom( display, "_NET_WM_WINDOW_TYPE_SPLASH", GL_FALSE );
03592         window->AtomWindowTypeNormal = XInternAtom( display, "_NET_WM_WINDOW_TYPE_NORMAL", GL_FALSE );
03593 
03594         window->AtomAllowedActions = XInternAtom( display, "_NET_WM_ALLOWED_ACTIONS", GL_FALSE );
03595         window->AtomActionResize = XInternAtom( display, "WM_ACTION_RESIZE", GL_FALSE );
03596         window->AtomActionMinimize = XInternAtom( display, "_WM_ACTION_MINIMIZE", GL_FALSE );
03597         window->AtomActionShade = XInternAtom( display, "WM_ACTION_SHADE", GL_FALSE );
03598         window->AtomActionMaximizeHorz = XInternAtom( display, "_WM_ACTION_MAXIMIZE_HORZ", GL_FALSE );
03599         window->AtomActionMaximizeVert = XInternAtom( display, "_WM_ACTION_MAXIMIZE_VERT", GL_FALSE );
03600         window->AtomActionClose = XInternAtom( display, "_WM_ACTION_CLOSE", GL_FALSE );
03601 
03602         window->AtomDesktopGeometry = XInternAtom( display, "_NET_DESKTOP_GEOMETRY", GL_FALSE );
03603     }
```

3.2.3.50 static void windowManager::InitializeGL (tWindow * *window*) [inline], [static], [private]

```
02309     {
02310 #if defined( _WIN32 ) || defined( _WIN64 )
02311         Windows_InitializeGL( window );
02312 #else
02313         Linux_InitializeGL( window );
02314 #endif
02315     }
```

3.2.3.51 static void windowManager::InitializeWindow (tWindow * *window*) [inline], [static], [private]

```
02300     {
```

```
02301 #if defined( _WIN32 ) || defined( _WIN64 )
02302         Windows_InitializeWindow( window );
02303 #else
02304         Linux_InitializeWindow( window );
02305 #endif
02306     }
```

3.2.3.52 static GLboolean windowManager::IsInitialized (void) [inline], [static]

```
01598     {
01599         return GetInstance()->isInitialized;
01600     }
```

3.2.3.53 static GLboolean windowManager::IsValid (const char * *stringParameter*) [inline], [static], [private]

```
02245     {
02246         return ( stringParameter != nullptr );
02247     }
```

3.2.3.54 static GLboolean windowManager::IsValid (onKeyEvent_t *onKeyPressed*) [inline], [static], [private]

```
02250     {
02251         return ( onKeyPressed != nullptr );
02252     }
```

3.2.3.55 static GLboolean windowManager::IsValid (onMouseWheelEvent_t *onMouseWheelEvent*) [inline], [static], [private]

```
02255     {
02256         return ( onMouseWheelEvent != nullptr );
02257     }
```

3.2.3.56 static GLboolean windowManager::IsValid (onMaximizedEvent_t *onMaximized*) [inline], [static], [private]

```
02260     {
02261         return ( onMaximized != nullptr );
02262     }
```

3.2.3.57 static GLboolean windowManager::IsValid (onFocusEvent_t *onFocus*) [inline], [static], [private]

```
02265     {
02266         return ( onFocus != nullptr );
02267     }
```

3.2.3.58 static GLboolean windowManager::IsValid (onMovedEvent_t *onMoved*) [inline], [static], [private]

```
02270     {
02271         return ( onMoved != nullptr );
02272     }
```

3.2.3.59 static GLboolean windowManager::IsValid (onMouseMoveEvent_t *onMouseMove*) [inline], [static], [private]

```
02275     {
02276         return ( onMouseMove != nullptr );
02277     }
```

3.2.3.60 static void windowManager::Linux_DisableDecorators (tWindow * *window,* GLbitfield *decorators*) [inline], [static], [private]

< the close button decoration of the window

< the maximize button decoration pf the window

< the minimize button decoration of the window

< the minimize button decoration of the window

< the maximize button decoration pf the window

< the minimize button decoration of the window

< the icon decoration of the window

< The title bar decoration of the window

< the border decoration of the window

< the sizable border decoration of the window

```
04775     {
04776         if ( decorators & DECORATOR_CLOSEBUTTON )
04777         {
04778             //I hate doing this but it is neccessary to keep functionality going.
04779             GLboolean minimizeEnabled, maximizeEnabled;
04780 
04781             if ( decorators & DECORATOR_MAXIMIZEBUTTON )
04782             {
04783                 maximizeEnabled = GL_TRUE;
04784             }
04785 
04786             if ( decorators & DECORATOR_MINIMIZEBUTTON )
04787             {
04788                 minimizeEnabled = GL_TRUE;
04789             }
04790 
04791             window->currentWindowStyle &= ~LINUX_DECORATOR_CLOSE;
04792 
04793             if ( maximizeEnabled )
04794             {
04795                 window->currentWindowStyle |= LINUX_DECORATOR_MAXIMIZE;
04796             }
04797 
04798             if ( minimizeEnabled )
04799             {
04800                 window->currentWindowStyle |= LINUX_DECORATOR_MINIMIZE;
04801             }
04802 
04803             window->decorators = 1;
04804         }
04805 
04806         if ( decorators & DECORATOR_MINIMIZEBUTTON )
04807         {
04808             window->currentWindowStyle &= ~LINUX_DECORATOR_MINIMIZE;
04809             window->decorators = 1;
04810         }
04811 
04812         if ( decorators & DECORATOR_MAXIMIZEBUTTON )
04813         {
04814             GLboolean minimizeEnabled;
04815 
04816             if ( decorators & DECORATOR_MINIMIZEBUTTON )
04817             {
04818                 minimizeEnabled = GL_TRUE;
04819             }
04820 
04821             window->currentWindowStyle &= ~LINUX_DECORATOR_MAXIMIZE;
04822 
04823             if ( minimizeEnabled )
```

```
04824             {
04825                 window->currentWindowStyle |= LINUX_DECORATOR_MINIMIZE;
04826             }
04827 
04828             window->decorators = 1;
04829         }
04830 
04831         if ( decorators & DECORATOR_ICON )
04832         {
04833             //Linux ( at least cinammon ) doesnt have icons in the window. only in the taskbar icon
04834         }
04835 
04836         //just need to set it to 1 to enable all decorators that include title bar
04837         if ( decorators & DECORATOR_TITLEBAR )
04838         {
04839             window->decorators = LINUX_DECORATOR_BORDER;
04840         }
04841 
04842         if ( decorators & DECORATOR_BORDER )
04843         {
04844             window->decorators = 0;
04845         }
04846 
04847         if ( decorators & DECORATOR_SIZEABLEBORDER )
04848         {
04849             window->decorators = 0;
04850         }
04851 
04852         long hints[5] = { LINUX_FUNCTION | LINUX_DECORATOR, window->
    currentWindowStyle, window->decorators, 0, 0 };
04853 
04854         XChangeProperty( GetDisplay(), window->windowHandle, window->AtomHints, XA_ATOM, 32,
04855             PropModeReplace, ( unsigned char* )hints, 5 );
04856 
04857         XMapWindow( GetDisplay(), window->windowHandle );
04858     }
```

3.2.3.61 static void windowManager::Linux_EnableDecorators (tWindow * *window*, GLbitfield *decorators*) [inline], [static], [private]

< the close button decoration of the window

< the minimize button decoration of the window

< the maximize button decoration pf the window

< the icon decoration of the window

< The title bar decoration of the window

< the border decoration of the window

< the sizable border decoration of the window

```
04726     {
04727         if ( decorators & DECORATOR_CLOSEBUTTON )
04728         {
04729             window->currentWindowStyle |= LINUX_DECORATOR_CLOSE;
04730             window->decorators = 1;
04731         }
04732 
04733         if ( decorators & DECORATOR_MINIMIZEBUTTON )
04734         {
04735             window->currentWindowStyle |= LINUX_DECORATOR_MINIMIZE;
04736             window->decorators = 1;
04737         }
04738 
04739         if ( decorators & DECORATOR_MAXIMIZEBUTTON )
04740         {
04741             window->currentWindowStyle |= LINUX_DECORATOR_MAXIMIZE;
04742             window->decorators = 1;
04743         }
04744 
04745         if ( decorators & DECORATOR_ICON )
04746         {
04747             //Linux ( at least cinammon ) doesnt have icons in the window. only in the taskbar icon
04748         }
04749 
04750         //just need to set it to 1 to enable all decorators that include title bar
04751         if ( decorators & DECORATOR_TITLEBAR )
```

```
04752         {
04753             window->decorators = 1;
04754         }
04755 
04756         if ( decorators & DECORATOR_BORDER )
04757         {
04758             window->decorators = 1;
04759         }
04760 
04761         if ( decorators & DECORATOR_SIZEABLEBORDER )
04762         {
04763             window->decorators = 1;
04764         }
04765 
04766         long hints[5] = { LINUX_FUNCTION | LINUX_DECORATOR, window->
      currentWindowStyle, window->decorators, 0, 0 };
04767 
04768         XChangeProperty( GetDisplay(), window->windowHandle, window->AtomHints, XA_ATOM, 32,
04769             PropModeReplace, ( unsigned char* )hints, 5 );
04770 
04771         XMapWindow( GetDisplay(), window->windowHandle );
04772     }
```

3.2.3.62 static void windowManager::Linux_Focus (tWindow * *window,* GLboolean *newFocusState*) [inline], [static], [private]

```
03779     {
03780         if( newFocusState )
03781         {
03782             XMapWindow( windowManager::GetDisplay(), window->windowHandle );
03783         }
03784 
03785         else
03786         {
03787             XUnmapWindow( windowManager::GetDisplay(), window->windowHandle );
03788         }
03789     }
```

3.2.3.63 static void windowManager::Linux_Fullscreen (tWindow * *window*) [inline], [static], [private]

< the window is currently full screen

```
03725     {
03726         XEvent currentEvent;
03727         memset( &currentEvent, 0, sizeof( currentEvent ) );
03728 
03729         currentEvent.xany.type = ClientMessage;
03730         currentEvent.xclient.message_type = window->AtomState;
03731         currentEvent.xclient.format = 32;
03732         currentEvent.xclient.window = window->windowHandle;
03733         currentEvent.xclient.data.l[0] = window->currentState ==
      WINDOWSTATE_FULLSCREEN;
03734         currentEvent.xclient.data.l[1] = window->AtomFullScreen;
03735 
03736         XSendEvent( windowManager::GetDisplay(),
03737             XDefaultRootWindow( windowManager::GetDisplay() ),
03738             0, SubstructureNotifyMask, &currentEvent );
03739     }
```

3.2.3.64 static const char* windowManager::Linux_GetEventType (XEvent *currentEvent*) [inline], [static], [private]

```
04323     {
04324         switch ( currentEvent.type )
04325         {
04326         case MotionNotify:
04327         {
04328             return "Motion Notify Event\n";
04329         }
04330 
04331         case ButtonPress:
04332         {
04333             return "Button Press Event\n";
```

```
04334         }
04335
04336         case ButtonRelease:
04337         {
04338             return "Button Release Event\n";
04339         }
04340
04341         case ColormapNotify:
04342         {
04343             return "Color Map Notify event \n";
04344         }
04345
04346         case EnterNotify:
04347         {
04348             return "Enter Notify Event\n";
04349         }
04350
04351         case LeaveNotify:
04352         {
04353             return "Leave Notify Event\n";
04354         }
04355
04356         case Expose:
04357         {
04358             return "Expose Event\n";
04359         }
04360
04361         case GraphicsExpose:
04362         {
04363             return "Graphics expose event\n";
04364         }
04365
04366         case NoExpose:
04367         {
04368             return "No Expose Event\n";
04369         }
04370
04371         case FocusIn:
04372         {
04373             return "Focus In Event\n";
04374         }
04375
04376         case FocusOut:
04377         {
04378             return "Focus Out Event\n";
04379         }
04380
04381         case KeymapNotify:
04382         {
04383             return "Key Map Notify Event\n";
04384         }
04385
04386         case KeyPress:
04387         {
04388             return "Key Press Event\n";
04389         }
04390
04391         case KeyRelease:
04392         {
04393             return "Key Release Event\n";
04394         }
04395
04396         case PropertyNotify:
04397         {
04398             return "Property Notify Event\n";
04399         }
04400
04401         case ResizeRequest:
04402         {
04403             return "Resize Property Event\n";
04404         }
04405
04406         case CirculateNotify:
04407         {
04408             return "Circulate Notify Event\n";
04409         }
04410
04411         case ConfigureNotify:
04412         {
04413             return "configure Notify Event\n";
04414         }
04415
04416         case DestroyNotify:
04417         {
04418             return "Destroy Notify Request\n";
04419         }
04420
```

```
04421         case GravityNotify:
04422         {
04423             return "Gravity Notify Event \n";
04424         }
04425 
04426         case MapNotify:
04427         {
04428             return "Map Notify Event\n";
04429         }
04430 
04431         case ReparentNotify:
04432         {
04433             return "Reparent Notify Event\n";
04434         }
04435 
04436         case UnmapNotify:
04437         {
04438             return "Unmap notify event\n";
04439         }
04440 
04441         case MapRequest:
04442         {
04443             return "Map request event\n";
04444         }
04445 
04446         case ClientMessage:
04447         {
04448             return "Client Message Event\n";
04449         }
04450 
04451         case MappingNotify:
04452         {
04453             return "Mapping notify event\n";
04454         }
04455 
04456         case SelectionClear:
04457         {
04458             return "Selection Clear event\n";
04459         }
04460 
04461         case SelectionNotify:
04462         {
04463             return "Selection Notify Event\n";
04464         }
04465 
04466         case SelectionRequest:
04467         {
04468             return "Selection Request event\n";
04469         }
04470 
04471         case VisibilityNotify:
04472         {
04473             return "Visibility Notify Event\n";
04474         }
04475 
04476         default:
04477         {
04478             return 0;
04479         }
04480         }
04481     }
```

3.2.3.65 static GLboolean windowManager::Linux_Initialize (void) [inline], [static], [private]

< Linux: if cannot connect to X11 server

```
03555     {
03556         GetInstance()->currentDisplay = XOpenDisplay( 0 );
03557 
03558         if( !GetInstance()->currentDisplay )
03559         {
03560             PrintErrorMessage(
      TINYWINDOW_ERROR_LINUX_CANNOT_CONNECT_X_SERVER );
03561             return FOUNDATION_ERROR;
03562         }
03563 
03564         GetInstance()->screenResolution[0] = WidthOfScreen( XScreenOfDisplay(
      GetInstance()->currentDisplay,
03565             DefaultScreen( GetInstance()->currentDisplay ) ) );
03566 
03567         GetInstance()->screenResolution[1] = HeightOfScreen( XScreenOfDisplay(
      GetInstance()->currentDisplay,
```

```
03568             DefaultScreen( GetInstance()->currentDisplay ) ) );
03569         GetInstance()->isInitialized = GL_TRUE;
03570         return FOUNDATION_OK;
03571     }
```

3.2.3.66 static GLboolean windowManager::Linux_InitializeGL (tWindow * *window*) [inline], [static], [private]

< if the window already has an OpenGL context

```
03669     {
03670         if( !window->context )
03671         {
03672             window->context = glXCreateContext(
03673                     windowManager::GetDisplay(),
03674                 window->visualInfo,
03675                 0,
03676                 GL_TRUE );
03677
03678             if( window->context )
03679             {
03680                 glXMakeCurrent( GetDisplay(),
03681                     window->windowHandle,
03682                     window->context );
03683
03684                 XWindowAttributes l_Attributes;
03685
03686                 XGetWindowAttributes( GetDisplay(),
03687                     window->windowHandle, &l_Attributes );
03688                 window->position[0] = l_Attributes.x;
03689                 window->position[1] = l_Attributes.y;
03690
03691                 window->contextCreated = GL_TRUE;
03692                 return FOUNDATION_OK;
03693             }
03694         }
03695
03696         else
03697         {
03698             PrintErrorMessage(
      TINYWINDOW_ERROR_EXISTING_CONTEXT );
03699             return FOUNDATION_ERROR;
03700         }
03701
03702         return FOUNDATION_ERROR;
03703     }
```

3.2.3.67 static void windowManager::Linux_InitializeWindow (tWindow * *window*) [inline], [static], [private]

< Linux: if cannot connect to X11 server

< Linux: if visual information given was invalid

< Linux: when X11 fails to create a new window

```
03606     {
03607         window->attributes = new GLint[5]{
03608             GLX_RGBA,
03609             GLX_DOUBLEBUFFER,
03610             GLX_DEPTH_SIZE,
03611             window->depthBits,
03612             None};
03613
03614         window->decorators = 1;
03615         window->currentWindowStyle |= LINUX_DECORATOR_CLOSE |
      LINUX_DECORATOR_MAXIMIZE | LINUX_DECORATOR_MINIMIZE |
      LINUX_DECORATOR_MOVE;
03616
03617         if ( !windowManager::GetDisplay() )
03618         {
03619             PrintErrorMessage(
      TINYWINDOW_ERROR_LINUX_CANNOT_CONNECT_X_SERVER );
03620             exit( 0 );
03621         }
```

```
03622
03623         //window->VisualInfo = glXGetVisualFromFBConfig( GetDisplay(), GetBestFrameBufferConfig( window )
      );
03624
03625         window->visualInfo = glXChooseVisual( windowManager::GetDisplay(), 0,
      window->attributes );
03626
03627         if ( !window->visualInfo )
03628         {
03629             PrintErrorMessage(
      TINYWINDOW_ERROR_LINUX_INVALID_VISUALINFO );
03630             exit( 0 );
03631         }
03632
03633         window->setAttributes.colormap = XCreateColormap( GetDisplay(),
03634             DefaultRootWindow( GetDisplay() ),
03635             window->visualInfo->visual, AllocNone );
03636
03637         window->setAttributes.event_mask = ExposureMask | KeyPressMask
03638             | KeyReleaseMask | MotionNotify | ButtonPressMask | ButtonReleaseMask
03639             | FocusIn | FocusOut | Button1MotionMask | Button2MotionMask | Button3MotionMask |
03640             Button4MotionMask | Button5MotionMask | PointerMotionMask | FocusChangeMask
03641             | VisibilityChangeMask | PropertyChangeMask | SubstructureNotifyMask;
03642
03643         window->windowHandle = XCreateWindow( windowManager::GetDisplay(),
03644             XDefaultRootWindow( windowManager::GetDisplay() ), 0, 0,
03645             window->resolution[0], window->resolution[1],
03646             0, window->visualInfo->depth, InputOutput,
03647             window->visualInfo->visual, CWColormap | CWEventMask,
03648             &window->setAttributes );
03649
03650         if( !window->windowHandle )
03651         {
03652             PrintErrorMessage(
      TINYWINDOW_ERROR_LINUX_CANNOT_CREATE_WINDOW );
03653             exit( 0 );
03654         }
03655
03656         XMapWindow( GetDisplay(), window->windowHandle );
03657         XStoreName( GetDisplay(), window->windowHandle,
03658             window->name );
03659
03660         InitializeAtomics( window );
03661
03662         XSetWMProtocols( GetDisplay(), window->windowHandle, &window->AtomClose, GL_TRUE );
03663
03664         Linux_InitializeGL( window );
03665         return GL_TRUE;
03666     }
```

3.2.3.68 static void windowManager::Linux_Maximize (tWindow * *window*) [inline], [static], [private]

< the window is currently maximized

```
03756     {
03757         XEvent currentEvent;
03758         memset( &currentEvent, 0, sizeof( currentEvent ) );
03759
03760         currentEvent.xany.type = ClientMessage;
03761         currentEvent.xclient.message_type = window->AtomState;
03762         currentEvent.xclient.format = 32;
03763         currentEvent.xclient.window = window->windowHandle;
03764         currentEvent.xclient.data.l[0] = ( window->currentState ==
      WINDOWSTATE_MAXIMIZED );
03765         currentEvent.xclient.data.l[1] = window->AtomMaxVert;
03766         currentEvent.xclient.data.l[2] = window->AtomMaxHorz;
03767
03768         XSendEvent( windowManager::GetDisplay(),
03769             XDefaultRootWindow( windowManager::GetDisplay() ),
03770             0, SubstructureNotifyMask, &currentEvent );
03771     }
```

3.2.3.69 static void windowManager::Linux_Minimize (tWindow * *window*) [inline], [static], [private]

< the window is currently minimized

```
03742     {
```

```
03743         if( window->currentState == WINDOWSTATE_MINIMIZED )
03744         {
03745             XIconifyWindow( windowManager::GetDisplay(),
03746                 window->windowHandle, 0 );
03747         }
03748 
03749         else
03750         {
03751             XMapWindow( windowManager::GetDisplay(), window->windowHandle );
03752         }
03753     }
```

3.2.3.70 static void windowManager::Linux_PollForEvents (void) [inline], [static], [private]

```
04285     {
04286         //if there are any events to process
04287         if( XEventsQueued( GetInstance()->GetDisplay(), QueuedAfterReading ) )
04288         {
04289             XNextEvent( GetInstance()->currentDisplay, &
      GetInstance()->currentEvent );
04290 
04291             XEvent currentEvent = GetInstance()->
      currentEvent;
04292 
04293             Linux_ProcessEvents( currentEvent );
04294         }
04295     }
```

3.2.3.71 static void windowManager::Linux_ProcessEvents (XEvent *currentEvent*) [inline], [static], [private]

< the key is currently up

< the key is currently up

< the key is currently up

< the key is currently up

< the key is currently up

< the key is currently down

< the key is currently down

< the key is currently down

< the key is currently down

< the key is currently down

< the left mouse button

< the mouse button is currently down

< the left mouse button

< the mouse button is currently down

< the middle mouse button / ScrollWheel

< the mouse button is currently down

< the middle mouse button / ScrollWheel

< the mouse button is currently down

< the right mouse button

< the mouse button is currently down

< the right mouse button

< the mouse button is currently down

< the mouse wheel down

< the mouse button is currently down

< the mouse wheel up

< the mouse wheel up

< the mouse button is currently down

< the mouse wheel up

< the left mouse button

< the mouse button is currently up

< the left mouse button

< the mouse button is currently up

< the middle mouse button / ScrollWheel

< the mouse button is currently up

< the middle mouse button / ScrollWheel

< the mouse button is currently up

< the right mouse button

< the mouse button is currently up

< the right mouse button

< the mouse button is currently up

< the mouse wheel down

< the mouse button is currently down

< the mouse wheel up

< the mouse button is currently down

set the screen mouse position to match the event

```
03820     {
03821         tWindow* window = GetWindowByEvent( currentEvent );
03822 
03823         switch ( currentEvent.type )
03824         {
03825             case Expose:
03826             {
03827                 break;
03828             }
03829 
03830             case DestroyNotify:
03831             {
03832                 //  printf( "blarg" );
03833 
03834                 if ( IsValid( window->destroyedEvent ) )
03835                 {
03836                     window->destroyedEvent();
03837 
03838                 }
03839 
03840                 printf( "Window was destroyed\n" );
03841                 ShutdownWindow( window );
03842 
03843                 break;
03844 
03845             }
03846 
03847             /*case CreateNotify:
03848             {
03849             printf( "Window was created\n" );
03850             l_Window->InitializeGL();
03851 
03852             if( IsValid( l_Window->m_OnCreated ) )
03853             {
03854             l_Window->m_OnCreated();
03855             }
```

```
03856
03857             break;
03858             } */
03859
03860             case KeyPress:
03861             {
03862                 GLuint functionKeysym = XKeycodeToKeysym(
03863                     GetInstance()->currentDisplay,
    currentEvent.xkey.keycode, 1 );
03864
03865                 if ( functionKeysym <= 255 )
03866                 {
03867                     window->keys[functionKeysym] = KEYSTATE_DOWN;
03868                     if ( IsValid( window->keyEvent ) )
03869                     {
03870                         window->keyEvent( functionKeysym, KEYSTATE_DOWN );
03871                     }
03872                 }
03873
03874                 else
03875                 {
03876                     window->keys[Linux_TranslateKey( functionKeysym )] =
    KEYSTATE_DOWN;
03877
03878                     if ( IsValid( window->keyEvent ) )
03879                     {
03880                         window->keyEvent( Linux_TranslateKey( functionKeysym ),
    KEYSTATE_DOWN );
03881                     }
03882                 }
03883
03884                 break;
03885             }
03886
03887             case KeyRelease:
03888             {
03889                 GLboolean isRetriggered = GL_FALSE;
03890                 if ( XEventsQueued( GetInstance()->currentDisplay,
    QueuedAfterReading ) )
03891                 {
03892                     XEvent nextEvent;
03893                     XPeekEvent( GetInstance()->currentDisplay, &nextEvent );
03894
03895                     if ( nextEvent.type == KeyPress &&
03896                         nextEvent.xkey.time == currentEvent.xkey.time &&
03897                         nextEvent.xkey.keycode == currentEvent.xkey.keycode )
03898                     {
03899                         GLuint functionKeysym = XKeycodeToKeysym( GetInstance()->
    currentDisplay,
03900                         nextEvent.xkey.keycode, 1 );
03901
03902                         XNextEvent( GetInstance()->currentDisplay, &
    currentEvent );
03903                         window->keyEvent( Linux_TranslateKey( functionKeysym ),
    KEYSTATE_DOWN );
03904                         isRetriggered = GL_TRUE;
03905                     }
03906                 }
03907
03908                 if ( !isRetriggered )
03909                 {
03910                     GLuint functionKeysym = XKeycodeToKeysym( GetInstance()->
    currentDisplay,
03911                         currentEvent.xkey.keycode, 1 );
03912
03913                     if ( functionKeysym <= 255 )
03914                     {
03915                         window->keys[functionKeysym] = KEYSTATE_UP;
03916
03917                         if ( IsValid( window->keyEvent ) )
03918                         {
03919                             window->keyEvent( functionKeysym, KEYSTATE_UP );
03920                         }
03921                     }
03922
03923                     else
03924                     {
03925                         window->keys[Linux_TranslateKey( functionKeysym )] =
    KEYSTATE_UP;
03926
03927                         if ( IsValid( window->keyEvent ) )
03928                         {
03929                             window->keyEvent( Linux_TranslateKey( functionKeysym ),
    KEYSTATE_UP );
03930                         }
03931                     }
03932
```

```
03933                     if ( IsValid( window->keyEvent ) )
03934                     {
03935                         window->keyEvent( Linux_TranslateKey( functionKeysym ),
      KEYSTATE_UP );
03936                     }
03937                 }
03938
03939                 break;
03940             }
03941
03942             case ButtonPress:
03943             {
03944                 switch ( currentEvent.xbutton.button )
03945                 {
03946                 case 1:
03947                 {
03948                     window->mouseButton[MOUSE_LEFTBUTTON] =
      MOUSE_BUTTONDOWN;
03949
03950                     if ( IsValid( window->mouseButtonEvent ) )
03951                     {
03952                         window->mouseButtonEvent( MOUSE_LEFTBUTTON,
      MOUSE_BUTTONDOWN );
03953                     }
03954                     break;
03955                 }
03956
03957                 case 2:
03958                 {
03959                     window->mouseButton[MOUSE_MIDDLEBUTTON] =
      MOUSE_BUTTONDOWN;
03960
03961                     if ( IsValid( window->mouseButtonEvent ) )
03962                     {
03963                         window->mouseButtonEvent( MOUSE_MIDDLEBUTTON,
      MOUSE_BUTTONDOWN );
03964                     }
03965                     break;
03966                 }
03967
03968                 case 3:
03969                 {
03970                     window->mouseButton[MOUSE_RIGHTBUTTON] =
      MOUSE_BUTTONDOWN;
03971
03972                     if ( IsValid( window->mouseButtonEvent ) )
03973                     {
03974                         window->mouseButtonEvent( MOUSE_RIGHTBUTTON,
      MOUSE_BUTTONDOWN );
03975                     }
03976                     break;
03977                 }
03978
03979                 case 4:
03980                 {
03981                     window->mouseButton[MOUSE_SCROLL_UP] =
      MOUSE_BUTTONDOWN;
03982
03983                     if ( IsValid( window->mouseWheelEvent ) )
03984                     {
03985                         window->mouseWheelEvent( MOUSE_SCROLL_DOWN );
03986                     }
03987                     break;
03988                 }
03989
03990                 case 5:
03991                 {
03992                     window->mouseButton[MOUSE_SCROLL_DOWN] =
      MOUSE_BUTTONDOWN;
03993
03994                     if ( IsValid( window->mouseWheelEvent ) )
03995                     {
03996                         window->mouseWheelEvent( MOUSE_SCROLL_DOWN );
03997                     }
03998                     break;
03999                 }
04000
04001                 default:
04002                 {
04003                     //need to add more mmouse buttons
04004                     break;
04005                 }
04006                 }
04007
04008                 break;
04009             }
04010
```

```
04011             case ButtonRelease:
04012             {
04013                 switch ( currentEvent.xbutton.button )
04014                 {
04015                 case 1:
04016                 {
04017                     //the left mouse button was released
04018                     window->mouseButton[MOUSE_LEFTBUTTON] =
      MOUSE_BUTTONUP;
04019
04020                     if ( IsValid( window->mouseButtonEvent ) )
04021                     {
04022                         window->mouseButtonEvent( MOUSE_LEFTBUTTON,
      MOUSE_BUTTONUP );
04023                     }
04024                     break;
04025                 }
04026
04027                 case 2:
04028                 {
04029                     //the middle mouse button was released
04030                     window->mouseButton[MOUSE_MIDDLEBUTTON] =
      MOUSE_BUTTONUP;
04031
04032                     if ( IsValid( window->mouseButtonEvent ) )
04033                     {
04034                         window->mouseButtonEvent( MOUSE_MIDDLEBUTTON,
      MOUSE_BUTTONUP );
04035                     }
04036                     break;
04037                 }
04038
04039                 case 3:
04040                 {
04041                     //the right mouse button was released
04042                     window->mouseButton[MOUSE_RIGHTBUTTON] =
      MOUSE_BUTTONUP;
04043
04044                     if ( IsValid( window->mouseButtonEvent ) )
04045                     {
04046                         window->mouseButtonEvent( MOUSE_RIGHTBUTTON,
      MOUSE_BUTTONUP );
04047                     }
04048                     break;
04049                 }
04050
04051                 case 4:
04052                 {
04053                     //the mouse wheel was scrolled up
04054                     window->mouseButton[MOUSE_SCROLL_UP] =
      MOUSE_BUTTONDOWN;
04055                     break;
04056                 }
04057
04058                 case 5:
04059                 {
04060                     //the mouse wheel wasscrolled down
04061                     window->mouseButton[MOUSE_SCROLL_DOWN] =
      MOUSE_BUTTONDOWN;
04062                     break;
04063                 }
04064
04065                 default:
04066                 {
04067                     //need to add more mouse buttons
04068                     break;
04069                 }
04070                 }
04071                 break;
04072             }
04073
04074             //when the mouse/pointer device is moved
04075             case MotionNotify:
04076             {
04077                 //set the windows mouse position to match the event
04078                 window->mousePosition[0] =
04079                     currentEvent.xmotion.x;
04080
04081                 window->mousePosition[1] =
04082                     currentEvent.xmotion.y;
04083
04084                 ///set the screen mouse position to match the event
04085                 GetInstance()->screenMousePosition[0] =
      currentEvent.xmotion.x_root;
04086                 GetInstance()->screenMousePosition[1] =
      currentEvent.xmotion.y_root;
04087
```

```
04088                 if ( IsValid( window->mouseMoveEvent ) )
04089                 {
04090                     window->mouseMoveEvent( currentEvent.xmotion.x,
04091                         currentEvent.xmotion.y, currentEvent.xmotion.x_root,
04092                         currentEvent.xmotion.y_root );
04093                 }
04094                 break;
04095             }
04096
04097             //when the window goes out of focus
04098             case FocusOut:
04099             {
04100                 window->inFocus = GL_FALSE;
04101                 if ( IsValid( window->focusEvent ) )
04102                 {
04103                     window->focusEvent(
04104                         window->inFocus );
04105                 }
04106                 break;
04107             }
04108
04109             //when the window is back in focus ( use to call restore callback? )
04110             case FocusIn:
04111             {
04112                 window->inFocus = GL_TRUE;
04113
04114                 if ( IsValid( window->focusEvent ) )
04115                 {
04116                     window->focusEvent( window->inFocus );
04117                 }
04118                 break;
04119             }
04120
04121             //when a request to resize the window is made either by
04122             //dragging out the window or programmatically
04123             case ResizeRequest:
04124             {
04125                 window->resolution[0] = currentEvent.xresizerequest.width;
04126                 window->resolution[1] = currentEvent.xresizerequest.height;
04127
04128                 glViewport( 0, 0,
04129                     window->resolution[0],
04130                     window->resolution[1] );
04131
04132                 if ( IsValid( window->resizeEvent ) )
04133                 {
04134                     window->resizeEvent( currentEvent.xresizerequest.width,
04135                         currentEvent.xresizerequest.height );
04136                 }
04137
04138                 break;
04139             }
04140
04141             //when a request to configure the window is made
04142             case ConfigureNotify:
04143             {
04144                 glViewport( 0, 0, currentEvent.xconfigure.width,
04145                     currentEvent.xconfigure.height );
04146
04147                 //check if window was resized
04148                 if ( ( GLuint )currentEvent.xconfigure.width != window->resolution[0]
04149                     || ( GLuint )currentEvent.xconfigure.height != window->resolution[1] )
04150                 {
04151                     if ( IsValid( window->resizeEvent ) )
04152                     {
04153                         window->resizeEvent( currentEvent.xconfigure.width,
      currentEvent.xconfigure.height );
04154                     }
04155
04156                     window->resolution[0] = currentEvent.xconfigure.width;
04157                     window->resolution[1] = currentEvent.xconfigure.height;
04158                 }
04159
04160                 //check if window was moved
04161                 if ( ( GLuint )currentEvent.xconfigure.x != window->position[0]
04162                     || ( GLuint )currentEvent.xconfigure.y != window->position[1] )
04163                 {
04164                     if ( IsValid( window->movedEvent ) )
04165                     {
04166                         window->movedEvent( currentEvent.xconfigure.x,
      currentEvent.xconfigure.y );
04167                     }
04168
04169                     window->position[0] = currentEvent.xconfigure.x;
04170                     window->position[1] = currentEvent.xconfigure.y;
04171                 }
04172                 break;
```

```
04173             }
04174
04175             case PropertyNotify:
04176             {
04177                 //this is needed in order to read from the windows WM_STATE Atomic
04178                 //to determine if the property notify event was caused by a client
04179                 //iconify event( minimizing the window ), a maximise event, a focus
04180                 //event and an attention demand event. NOTE these should only be
04181                 //for eventts that are not triggered programatically
04182
04183                 Atom type;
04184                 GLint format;
04185                 ulong numItems, bytesAfter;
04186                 unsigned char*  properties = nullptr;
04187
04188                 XGetWindowProperty( windowManager::GetDisplay(),
      currentEvent.xproperty.window,
04189                     window->AtomState,
04190                     0, LONG_MAX, GL_FALSE, AnyPropertyType,
04191                     &type, &format, &numItems, &bytesAfter,
04192                     & properties );
04193
04194                 if (  properties && ( format == 32 ) )
04195                 {
04196                     //go through each property and match it to an existing Atomic state
04197                     for ( GLuint currentItem = 0; currentItem < numItems; currentItem++ )
04198                     {
04199                         long currentProperty = ( ( long* )(  properties ) )[currentItem];
04200
04201                         if ( currentProperty == window->AtomHidden )
04202                         {
04203                             //window was minimized
04204                             if ( IsValid( window->minimizedEvent ) )
04205                             {
04206                                 //if the minimized callback for the window was set
04207                                 window->minimizedEvent();
04208                             }
04209                         }
04210
04211                         if ( currentProperty == window->AtomMaxVert ||
04212                             currentProperty == window->AtomMaxVert )
04213                         {
04214                             //window was maximized
04215                             if ( IsValid( window->maximizedEvent ) )
04216                             {
04217                                 //if the maximized callback for the window was set
04218                                 window->maximizedEvent();
04219                             }
04220                         }
04221
04222                         if ( currentProperty == window->AtomFocused )
04223                         {
04224                             //window is now in focus. we can ignore this is as FocusIn/FocusOut does this
      anyway
04225                         }
04226
04227                         if ( currentProperty == window->AtomDemandsAttention )
04228                         {
04229                             //the window demands attention like a celebrity
04230                             printf( "window demands attention \n" );
04231                         }
04232                     }
04233                 }
04234
04235                 break;
04236             }
04237
04238             case GravityNotify:
04239             {
04240                 //this is only supposed to pop up when the parent of this window( if any ) has something
      happen
04241                 //to it so that this window can react to said event as well.
04242                 break;
04243             }
04244
04245             //check for events that were created by the TinyWindow manager
04246             case ClientMessage:
04247             {
04248                 const char* atomName = XGetAtomName( windowManager::GetDisplay(),
      currentEvent.xclient.message_type );
04249                 if ( IsValid( atomName ) )
04250                 {
04251                     //printf( "%s\n", l_AtomName );
04252                 }
04253
04254                 if ( ( Atom )currentEvent.xclient.data.l[0] == window->AtomClose )
04255                 {
```

```
04256                     printf( "window closed\n" );
04257                     window->shouldClose = GL_TRUE;
04258                     if( IsValid( window->destroyedEvent ) )
04259                     {
04260                         window->destroyedEvent();
04261                     }
04262                     ShutdownWindow( window );
04263 
04264                     break;
04265 
04266                 }
04267 
04268                 //check if fullscreen
04269                 if ( ( Atom )currentEvent.xclient.data.l[1] == window->AtomFullScreen )
04270                 {
04271                     break;
04272                 }
04273                 break;
04274 
04275             }
04276 
04277             default:
04278             {
04279                 return;
04280             }
04281         }
04282     }
```

3.2.3.72 static void windowManager::Linux_Restore (tWindow * *window*) [inline], [static], [private]

```
03774     {
03775         XMapWindow( windowManager::GetDisplay(), window->windowHandle );
03776     }
```

3.2.3.73 static void windowManager::Linux_SetMousePosition (tWindow * *window*) [inline], [static], [private]

```
03792     {
03793         XWarpPointer(
03794             windowManager::GetInstance()->
      currentDisplay,
03795             window->windowHandle, window->windowHandle,
03796             window->position[0], window->position[1],
03797             window->resolution[0], window->resolution[1],
03798             window->mousePosition[0], window->mousePosition[1] );
03799     }
```

3.2.3.74 static void windowManager::Linux_SetMousePositionInScreen (GLuint *x*, GLuint *y*) [inline], [static], [private]

```
04308     {
04309         XWarpPointer( GetInstance()->currentDisplay, None,
04310             XDefaultRootWindow( GetInstance()->currentDisplay ), 0, 0,
04311             GetScreenResolution()[0],
04312             GetScreenResolution()[1],
04313             x, y );
04314     }
```

3.2.3.75 static void windowManager::Linux_SetWindowIcon (tWindow * *window*, const char * *icon*, GLuint *width*, GLuint *height*) [inline], [static], [private]

< Linux: when the function has not yet been implemented on the Linux in the current version of the API

```
04913     {
04914         //sorry :(
04915         PrintErrorMessage(
      TINYWINDOW_ERROR_LINUX_FUNCTION_NOT_IMPLEMENTED );
04916     }
```

3.2.3.76 static void windowManager::Linux_SetWindowPosition (tWindow * *window*) [inline], [static], [private]

```
03802     {
03803         XWindowChanges windowChanges;
03804 
03805         windowChanges.x = window->position[0];
03806         windowChanges.y = window->position[1];
03807 
03808         XConfigureWindow(
03809             windowManager::GetDisplay(),
03810             window->windowHandle, CWX | CWY, &windowChanges );
03811     }
```

3.2.3.77 static void windowManager::Linux_SetWindowResolution (tWindow * *window*) [inline], [static], [private]

```
03814     {
03815         XResizeWindow( windowManager::GetDisplay(),
03816             window->windowHandle, window->resolution[0], window->resolution[1] );
03817     }
```

3.2.3.78 static void windowManager::Linux_SetWindowStyle (tWindow * *window,* GLuint *windowStyle*) [inline], [static], [private]

< the default window style for the respective platform

< the window has no decorators but the window border and title bar

< the window has no decorators

< if the window style gives is invalid

```
04861     {
04862         switch ( windowStyle )
04863         {
04864         case WINDOWSTYLE_DEFAULT:
04865         {
04866             window->decorators = ( 1L << 2 );
04867             window->currentWindowStyle = LINUX_DECORATOR_MOVE |
      LINUX_DECORATOR_CLOSE |
04868                 LINUX_DECORATOR_MAXIMIZE |
      LINUX_DECORATOR_MINIMIZE;
04869             long Hints[5] = { LINUX_FUNCTION | LINUX_DECORATOR, window->
      currentWindowStyle, window->decorators, 0, 0 };
04870 
04871             XChangeProperty( GetDisplay(), window->windowHandle, window->AtomHints, XA_ATOM, 32,
      PropModeReplace,
04872                 ( unsigned char* )Hints, 5 );
04873 
04874             XMapWindow( GetDisplay(), window->windowHandle );
04875             break;
04876         }
04877 
04878         case WINDOWSTYLE_BARE:
04879         {
04880             window->decorators = ( 1L << 2 );
04881             window->currentWindowStyle = ( 1L << 2 );
04882             long Hints[5] = { LINUX_FUNCTION | LINUX_DECORATOR, window->
      currentWindowStyle, window->decorators, 0, 0 };
04883 
04884             XChangeProperty( GetDisplay(), window->windowHandle, window->AtomHints, XA_ATOM, 32,
      PropModeReplace,
04885                 ( unsigned char* )Hints, 5 );
04886 
04887             XMapWindow( GetDisplay(), window->windowHandle );
04888             break;
04889         }
04890 
04891         case WINDOWSTYLE_POPUP:
04892         {
04893             window->decorators = 0;
04894             window->currentWindowStyle = ( 1L << 2 );
04895             long Hints[5] = { LINUX_FUNCTION | LINUX_DECORATOR, window->
      currentWindowStyle, window->decorators, 0, 0 };
```

```
04896
04897             XChangeProperty( GetDisplay(), window->windowHandle, window->AtomHints, XA_ATOM, 32,
    PropModeReplace,
04898                  ( unsigned char* )Hints, 5 );
04899
04900             XMapWindow( GetDisplay(), window->windowHandle );
04901             break;
04902         }
04903
04904         default:
04905         {
04906             PrintErrorMessage(
    TINYWINDOW_ERROR_INVALID_WINDOWSTYLE );
04907             break;
04908         }
04909         }
04910     }
```

3.2.3.79 static void windowManager::Linux_Shutdown (void) [inline], [static], [private]

```
03720     {
03721         XCloseDisplay( GetInstance()->currentDisplay );
03722     }
```

3.2.3.80 static void windowManager::Linux_ShutdownWindow (tWindow * *window*) [inline], [static], [private]

< the window is currently full screen

```
03706     {
03707         if( window->currentState == WINDOWSTATE_FULLSCREEN )
03708         {
03709             RestoreWindowByName( window->name );
03710         }
03711
03712         glXDestroyContext( windowManager::GetDisplay(), window->context );
03713         XUnmapWindow( windowManager::GetDisplay(), window->windowHandle );
03714         XDestroyWindow( windowManager::GetDisplay(), window->windowHandle );
03715         window->windowHandle = 0;
03716         window->context = 0;
03717     }
```

3.2.3.81 static GLuint windowManager::Linux_TranslateKey (GLuint *keySymbol*) [inline], [static], [private]

< the fist key that is not a char

< the Escape key

< the fist key that is not a char

< the Home key

< the fist key that is not a char

< the ArrowLeft key

< the fist key that is not a char

< the ArrowRight key

< the fist key that is not a char

< the ArrowUp key

< the fist key that is not a char

< the ArrowDown key

< the fist key that is not a char

< the PageUp key

< the fist key that is not a char

< the PageDown key

< the fist key that is not a char

< the End key

< the fist key that is not a char

< the PrintScreen key

< the fist key that is not a char

< the insert key

< the fist key that is not a char

< the NumLock key

< the fist key that is not a char

< the Keypad Multiply key

< the fist key that is not a char

< the Keypad Add key

< the fist key that is not a char

< the Keypad Subtract key

< the fist key that is not a char

< the Keypad Period/Decimal key

< the fist key that is not a char

< the KeyPad Divide key

< the fist key that is not a char

< the Keypad 0 key

< the fist key that is not a char

< the Keypad 1 key

< the fist key that is not a char

< the Keypad 2 key

< the fist key that is not a char

< the Keypad 3 key

< the fist key that is not a char

< the Keypad 4 key

< the fist key that is not a char

< the Keypad 5 key

< the fist key that is not a char

< the Keypad 6 key

< the fist key that is not a char

< the Keypad 7 key

< the fist key that is not a char

< the keypad 8 key

< the fist key that is not a char

< the Keypad 9 key

< the fist key that is not a char

< the F1 key

< the fist key that is not a char

< the F2 key

< the fist key that is not a char

< the F3 key

< the fist key that is not a char

< the F4 key

< the fist key that is not a char

< the F5 key

< the fist key that is not a char

< the F6 key

< the fist key that is not a char

< the F7 key

< the fist key that is not a char

< the F8 key

< the fist key that is not a char

< the F9 key

< the fist key that is not a char

< the F10 key

< the fist key that is not a char

< the F11 key

< the fist key that is not a char

< the F12 key

< the fist key that is not a char

< the left Shift key

< the fist key that is not a char

< the right Shift key

< the fist key that is not a char

< the right Control key

< the fist key that is not a char

< the left Control key

< the fist key that is not a char

< the CapsLock key

< the fist key that is not a char

< the left Alternate key

< the fist key that is not a char

< the right Alternate key

```
04485     {
```

```
04486         switch ( keySymbol )
04487         {
04488         case XK_Escape:
04489         {
04490             return KEY_ESCAPE;
04491         }
04492
04493         case XK_Home:
04494         {
04495             return KEY_HOME;
04496         }
04497
04498         case XK_Left:
04499         {
04500             return KEY_ARROW_LEFT;
04501         }
04502
04503         case XK_Right:
04504         {
04505             return KEY_ARROW_RIGHT;
04506         }
04507
04508         case XK_Up:
04509         {
04510             return KEY_ARROW_UP;
04511         }
04512
04513         case XK_Down:
04514         {
04515             return KEY_ARROW_DOWN;
04516         }
04517
04518         case XK_Page_Up:
04519         {
04520             return KEY_PAGEUP;
04521         }
04522
04523         case XK_Page_Down:
04524         {
04525             return KEY_PAGEDOWN;
04526         }
04527
04528         case XK_End:
04529         {
04530             return KEY_END;
04531         }
04532
04533         case XK_Print:
04534         {
04535             return KEY_PRINTSCREEN;
04536         }
04537
04538         case XK_Insert:
04539         {
04540             return KEY_INSERT;
04541         }
04542
04543         case XK_Num_Lock:
04544         {
04545             return KEY_NUMLOCK;
04546         }
04547
04548         case XK_KP_Multiply:
04549         {
04550             return KEY_KEYPAD_MULTIPLY;
04551         }
04552
04553         case XK_KP_Add:
04554         {
04555             return KEY_KEYPAD_ADD;
04556         }
04557
04558         case XK_KP_Subtract:
04559         {
04560             return KEY_KEYPAD_SUBTRACT;
04561         }
04562
04563         case XK_KP_Decimal:
04564         {
04565             return KEY_KEYPAD_PERIOD;
04566         }
04567
04568         case XK_KP_Divide:
04569         {
04570             return KEY_KEYPAD_DIVIDE;
04571         }
04572
```

```
04573         case XK_KP_0:
04574         {
04575             return KEY_KEYPAD_0;
04576         }
04577
04578         case XK_KP_1:
04579         {
04580             return KEY_KEYPAD_1;
04581         }
04582
04583         case XK_KP_2:
04584         {
04585             return KEY_KEYPAD_2;
04586         }
04587
04588         case XK_KP_3:
04589         {
04590             return KEY_KEYPAD_3;
04591         }
04592
04593         case XK_KP_4:
04594         {
04595             return KEY_KEYPAD_4;
04596         }
04597
04598         case XK_KP_5:
04599         {
04600             return KEY_KEYPAD_5;
04601         }
04602
04603         case XK_KP_6:
04604         {
04605             return KEY_KEYPAD_6;
04606         }
04607
04608         case XK_KP_7:
04609         {
04610             return KEY_KEYPAD_7;
04611         }
04612
04613         case XK_KP_8:
04614         {
04615             return KEY_KEYPAD_8;
04616         }
04617
04618         case XK_KP_9:
04619         {
04620             return KEY_KEYPAD_9;
04621         }
04622
04623         case XK_F1:
04624         {
04625             return KEY_F1;
04626         }
04627
04628         case XK_F2:
04629         {
04630             return KEY_F2;
04631         }
04632
04633         case XK_F3:
04634         {
04635             return KEY_F3;
04636         }
04637
04638         case XK_F4:
04639         {
04640             return KEY_F4;
04641         }
04642
04643         case XK_F5:
04644         {
04645             return KEY_F5;
04646         }
04647
04648         case XK_F6:
04649         {
04650             return KEY_F6;
04651         }
04652
04653         case XK_F7:
04654         {
04655             return KEY_F7;
04656         }
04657
04658         case XK_F8:
04659         {
```

```
04660             return KEY_F8;
04661         }
04662
04663         case XK_F9:
04664         {
04665             return KEY_F9;
04666         }
04667
04668         case XK_F10:
04669         {
04670             return KEY_F10;
04671         }
04672
04673         case XK_F11:
04674         {
04675             return KEY_F11;
04676         }
04677
04678         case XK_F12:
04679         {
04680             return KEY_F12;
04681         }
04682
04683         case XK_Shift_L:
04684         {
04685             return KEY_LEFTSHIFT;
04686         }
04687
04688         case XK_Shift_R:
04689         {
04690             return KEY_RIGHTSHIFT;
04691         }
04692
04693         case XK_Control_R:
04694         {
04695             return KEY_RIGHTCONTROL;
04696         }
04697
04698         case XK_Control_L:
04699         {
04700             return KEY_LEFTCONTROL;
04701         }
04702
04703         case XK_Caps_Lock:
04704         {
04705             return KEY_CAPSLOCK;
04706         }
04707
04708         case XK_Alt_L:
04709         {
04710             return KEY_LEFTALT;
04711         }
04712
04713         case XK_Alt_R:
04714         {
04715             return KEY_RIGHTALT;
04716         }
04717
04718         default:
04719         {
04720             return 0;
04721         }
04722         }
04723     }
```

3.2.3.82 static void windowManager::Linux_WaitForEvents (void) [inline], [static], [private]

```
04297     {
04298         //even if there aren't any events to process
04299         XNextEvent( GetInstance()->currentDisplay, &
      GetInstance()->currentEvent );
04300
04301         XEvent currentEvent = GetInstance()->currentEvent;
04302
04303         Linux_ProcessEvents( currentEvent );
04304     }
```

3.2.3.83 static GLboolean windowManager::MakeWindowCurrentContextByIndex (GLuint *windowIndex*) [inline], [static]

make the given window be the current OpenGL Context to be drawn to < if a window tries to use a graphical function without a context

```
01045     {
01046         if ( GetInstance()->IsInitialized() )
01047         {
01048             if ( DoesExistByIndex( windowIndex ) )
01049             {
01050 #if defined( _WIN32 ) || defined( _WIN64 )
01051                 wglMakeCurrent( GetWindowByIndex( windowIndex )->deviceContextHandle,
01052                     GetWindowByIndex( windowIndex )->glRenderingContextHandle );
01053 #else
01054                 glXMakeCurrent( GetDisplay(), GetWindowByIndex( windowIndex )->
      windowHandle,
01055                     GetWindowByIndex( windowIndex )->context );
01056 #endif
01057                 return FOUNDATION_OK;
01058             }
01059 
01060             return FOUNDATION_ERROR;
01061         }
01062 
01063         PrintErrorMessage( TINYWINDOW_ERROR_NO_CONTEXT );
01064         return FOUNDATION_ERROR;
01065     }
```

3.2.3.84 static GLboolean windowManager::MakeWindowCurrentContextByName (const char * *windowName*) [inline], [static]

make the given window be the current OpenGL Context to be drawn to < if the window is being used without being initialized

```
01021     {
01022         if ( GetInstance()->IsInitialized() )
01023         {
01024             if ( DoesExistByName( windowName ) )
01025             {
01026 
01027 #if defined( _WIN32 ) || defined( _WIN64 )
01028                 wglMakeCurrent( GetWindowByName( windowName )->deviceContextHandle,
01029                     GetWindowByName( windowName )->glRenderingContextHandle );
01030 #else
01031                 glXMakeCurrent( windowManager::GetDisplay(),
      GetWindowByName( windowName )->windowHandle,
01032                     GetWindowByName( windowName )->context );
01033 #endif
01034                 return FOUNDATION_OK;
01035             }
01036             return FOUNDATION_ERROR;
01037         }
01038         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
       );
01039         return FOUNDATION_ERROR;
01040     }
```

3.2.3.85 static GLboolean windowManager::MaximizeWindowByIndex (GLuint *windowIndex*, GLboolean *newState*) [inline], [static]

toggle the maximization state of the current window < if the window is being used without being initialized

```
01357     {
01358         if ( GetInstance()->IsInitialized() )
01359         {
01360             if ( DoesExistByIndex( windowIndex ) )
01361             {
01362 #if defined( _WIN32 ) || defined( _WIN64 )
01363                 Windows_Maximize( GetWindowByIndex( windowIndex ), newState );
01364 #else
01365                 Linux_Maximize( GetWindowByIndex( windowIndex ) );
```

```
01366 #endif
01367                 return FOUNDATION_OK;
01368             }
01369             return FOUNDATION_ERROR;
01370         }
01371         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01372         return FOUNDATION_ERROR;
01373     }
```

3.2.3.86 static GLboolean windowManager::MaximizeWindowByName (const char * *windowName,* GLboolean *newState*) [inline], [static]

toggle the maximization state of the current window < the window is currently maximized

< the window is in its default state

< if the window is being used without being initialized

```
01321     {
01322         if ( GetInstance()->IsInitialized() )
01323         {
01324             if ( DoesExistByName( windowName ) )
01325             {
01326                 if ( newState )
01327                 {
01328                     GetWindowByName( windowName )->currentState =
      WINDOWSTATE_MAXIMIZED;
01329 #if defined( _WIN32 ) || defined( _WIN64 )
01330                     Windows_Maximize( GetWindowByName( windowName ), newState );
01331 #else
01332                     Linux_Maximize( GetWindowByName( windowName ) );
01333 #endif
01334                     return FOUNDATION_OK;
01335                 }
01336
01337                 else
01338                 {
01339                     GetWindowByName( windowName )->currentState =
      WINDOWSTATE_NORMAL;
01340 #if defined( _WIN32 ) || defined( _WIN64 )
01341                     Windows_Maximize( GetWindowByName( windowName ), newState );
01342 #else
01343                     Linux_Maximize( GetWindowByName( windowName ) );
01344 #endif
01345                     return FOUNDATION_OK;
01346                 }
01347             }
01348             return FOUNDATION_ERROR;
01349         }
01350         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01351         return FOUNDATION_ERROR;
01352     }
```

3.2.3.87 static GLboolean windowManager::MinimizeWindowByIndex (GLuint *windowIndex,* GLboolean *newState*) [inline], [static]

toggle the minimization state of the window < the window is currently minimized

< the window is in its default state

< if the window is being used without being initialized

```
01250     {
01251         if ( GetInstance()->IsInitialized() )
01252         {
01253             if ( DoesExistByIndex( windowIndex ) )
01254             {
01255                 if ( newState )
01256                 {
01257                     GetWindowByIndex( windowIndex )->
      currentState = WINDOWSTATE_MINIMIZED;
01258 #if defined( _WIN32 ) || defined( _WIN64 )
01259                     Windows_Minimize( GetWindowByIndex( windowIndex ), newState );
```

```
01260 #else
01261                     Linux_Minimize( GetWindowByIndex( windowIndex ) );
01262 #endif
01263                     return FOUNDATION_OK;
01264                 }
01265
01266                 else
01267                 {
01268                     GetWindowByIndex( windowIndex )->
      currentState = WINDOWSTATE_NORMAL;
01269 #if defined( _WIN32 ) || defined( _WIN64 )
01270                     Windows_Minimize( GetWindowByIndex( windowIndex ), newState );
01271 #else
01272                     Linux_Minimize( GetWindowByIndex( windowIndex ) );
01273 #endif
01274                     return FOUNDATION_OK;
01275                 }
01276             }
01277             return FOUNDATION_ERROR;
01278         }
01279         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01280         return FOUNDATION_ERROR;
01281     }
```

3.2.3.88 static GLboolean windowManager::MinimizeWindowByName (const char * *windowName*, GLboolean *newState*) [inline], [static]

toggle the minimization state of the given window < the window is currently minimized

< the window is in its default state

< if a window tries to use a graphical function without a context

```
01213     {
01214         if ( GetInstance()->IsInitialized() )
01215         {
01216             if ( DoesExistByName( windowName ) )
01217             {
01218                 if ( newState )
01219                 {
01220                     GetWindowByName( windowName )->currentState =
      WINDOWSTATE_MINIMIZED;
01221
01222 #if defined( _WIN32 ) || defined( _WIN64 )
01223                     Windows_Minimize( GetWindowByName( windowName ), newState );
01224 #else
01225                     Linux_Minimize( GetWindowByName( windowName ) );
01226 #endif
01227                     return FOUNDATION_OK;
01228                 }
01229
01230                 else
01231                 {
01232                     GetWindowByName( windowName )->currentState =
      WINDOWSTATE_NORMAL;
01233 #if defined( _WIN32 ) || defined( _WIN64 )
01234                     Windows_Minimize( GetWindowByName( windowName ), newState );
01235 #else
01236                     Linux_Minimize( GetWindowByName( windowName ) );
01237 #endif
01238                     return FOUNDATION_OK;
01239                 }
01240             }
01241             return FOUNDATION_ERROR;
01242         }
01243         PrintErrorMessage( TINYWINDOW_ERROR_NO_CONTEXT );
01244         return FOUNDATION_ERROR;
01245     }
```

3.2.3.89 static void windowManager::PollForEvents (void) [inline], [static]

< if the window is being used without being initialized

```
01604     {
01605         if ( GetInstance()->IsInitialized() )
```

```
01606         {
01607 #if defined( _WIN32 ) || defined( _WIN64 )
01608             GetInstance()->Windows_PollForEvents();
01609 #else
01610             GetInstance()->Linux_PollForEvents();
01611 #endif
01612         }
01613 
01614         else
01615         {
01616             PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01617         }
01618     }
```

3.2.3.90 static GLboolean windowManager::RemoveWindowByIndex (GLuint *windowIndex*) [inline], [static]

< if the window is being used without being initialized

```
01652     {
01653         if ( GetInstance()->IsInitialized() )
01654         {
01655             if ( DoesExistByIndex( windowIndex ) )
01656             {
01657                 ShutdownWindow( GetWindowByIndex( windowIndex ) );
01658                 return FOUNDATION_OK;
01659             }
01660             return FOUNDATION_ERROR;
01661         }
01662         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01663         return FOUNDATION_ERROR;
01664     }
```

3.2.3.91 static GLboolean windowManager::RemoveWindowByName (const char * *windowName*) [inline], [static]

< if the window is being used without being initialized

```
01638     {
01639         if ( GetInstance()->IsInitialized() )
01640         {
01641             if ( DoesExistByName( windowName ) )
01642             {
01643                 ShutdownWindow( GetWindowByName( windowName ) );
01644                 return FOUNDATION_OK;
01645             }
01646             return FOUNDATION_ERROR;
01647         }
01648         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
    );
01649         return FOUNDATION_ERROR;
01650     }
```

3.2.3.92 static GLboolean windowManager::RestoreWindowByIndex (GLuint *windowIndex*) [inline], [static]

< if a window tries to use a graphical function without a context

```
01568     {
01569         if ( GetInstance()->IsInitialized() )
01570         {
01571             if ( WindowExists( windowIndex ) )
01572             {
01573 #if defined( _WIN32 ) || defined( _WIN64 )
01574                 Windows_Restore( GetWindowByIndex( windowIndex ) );
01575 #else
01576                 Linux_Restore( GetWindowByIndex( windowIndex ) );
01577 #endif
01578                 return FOUNDATION_OK;
01579             }
01580             return FOUNDATION_ERROR;
```

```
01581         }
01582         PrintErrorMessage( TINYWINDOW_ERROR_NO_CONTEXT );
01583         return FOUNDATION_ERROR;
01584     }
```

3.2.3.93 static GLboolean windowManager::RestoreWindowByName (const char * *windowName*) [inline], [static]

< if the window is being used without being initialized

```
01550     {
01551         if ( GetInstance()->IsInitialized() )
01552         {
01553             if ( DoesExistByName( windowName ) )
01554             {
01555 #if defined( _WIN32 ) || defined( _WIN64 )
01556                 Windows_Restore( GetWindowByName( windowName ) );
01557 #else
01558                 Linux_Restore( GetWindowByName( windowName ) );
01559 #endif
01560                 return FOUNDATION_OK;
01561             }
01562             return FOUNDATION_ERROR;
01563         }
01564         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01565         return FOUNDATION_ERROR;
01566     }
```

3.2.3.94 static GLboolean windowManager::SetFullScreenByIndex (GLuint *windowIndex*, GLboolean *newState*) [inline], [static]

< the window is currently full screen

< the window is in its default state

< if the window is being used without being initialized

```
01143     {
01144         if ( GetInstance()->IsInitialized() )
01145         {
01146             if ( DoesExistByIndex( windowIndex ) )
01147             {
01148                 if ( newState )
01149                 {
01150                     GetWindowByIndex( windowIndex )->
      currentState = WINDOWSTATE_FULLSCREEN;
01151 #if defined( _WIN32 ) || defined( _WIN64 )
01152                     Windows_FullScreen( GetWindowByIndex( windowIndex ) );
01153 #else
01154                     Linux_Fullscreen( GetWindowByIndex( windowIndex ) );
01155 #endif
01156                     return FOUNDATION_OK;
01157                 }
01158
01159                 else
01160                 {
01161                     GetWindowByIndex( windowIndex )->
      currentState = WINDOWSTATE_NORMAL;
01162 #if defined( _WIN32 ) || defined( _WIN64 )
01163                     Windows_FullScreen( GetWindowByIndex( windowIndex ) );
01164 #else
01165                     Linux_Fullscreen( GetWindowByIndex( windowIndex ) );
01166 #endif
01167                     return FOUNDATION_OK;
01168                 }
01169             }
01170             return FOUNDATION_ERROR;
01171         }
01172         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01173         return FOUNDATION_ERROR;
01174     }
```

3.2.3.95 static GLboolean windowManager::SetFullScreenByName (const char * *windowName,* GLboolean *newState*) [inline], [static]

toggle the given window's full screen mode < the window is currently full screen

< the window is in its default state

< if the window is being used without being initialized

```
01106     {
01107         if ( GetInstance()->IsInitialized() )
01108         {
01109             if ( DoesExistByName( windowName ) )
01110             {
01111                 if ( newState )
01112                 {
01113                     GetWindowByName( windowName )->currentState =
      WINDOWSTATE_FULLSCREEN;
01114 #if defined( _WIN32 ) || defined( _WIN64 )
01115                     Windows_FullScreen( GetWindowByName( windowName ) );
01116 #else
01117                     Linux_Fullscreen( GetWindowByName( windowName ) );
01118 #endif
01119
01120                     return FOUNDATION_OK;
01121                 }
01122
01123                 else
01124                 {
01125                     GetWindowByName( windowName )->currentState =
      WINDOWSTATE_NORMAL;
01126 #if defined( _WIN32 ) || defined( _WIN64 )
01127                     Windows_FullScreen( GetWindowByName( windowName ) );
01128 #else
01129                     Linux_Fullscreen( GetWindowByName( windowName ) );
01130 #endif
01131                     return FOUNDATION_OK;
01132                 }
01133             }
01134             return FOUNDATION_ERROR;
01135         }
01136         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01137         return FOUNDATION_ERROR;
01138     }
```

3.2.3.96 static GLboolean windowManager::SetMousePositionInScreen (GLuint *x,* GLuint *y*) [inline], [static]

set the position of the mouse cursor relative to screen co-ordinates < if the window is being used without being initialized

```
00471     {
00472         if ( GetInstance()->IsInitialized() )
00473         {
00474             GetInstance()->screenMousePosition[0] = x;
00475             GetInstance()->screenMousePosition[1] = y;
00476
00477 #if defined( _WIN32 ) || defined( _WIN64 )
00478             Windows_SetMousePositionInScreen();
00479 #else
00480             Linux_SetMousePositionInScreen( x, y );
00481 #endif
00482             return FOUNDATION_OK;
00483         }
00484         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00485         return FOUNDATION_ERROR;
00486     }
```

3.2.3.97 static GLboolean windowManager::SetMousePositionInWindowByIndex (GLuint *windowIndex,* GLuint *x,* GLuint *y*) [inline], [static]

set the mouse Position of the given window's co-ordinates < if the window is being used without being initialized

```
00884     {
00885         if ( GetInstance()->IsInitialized() )
00886         {
00887             if ( DoesExistByIndex( windowIndex ) )
00888             {
00889                 GetWindowByIndex( windowIndex )->mousePosition[0] = x;
00890                 GetWindowByIndex( windowIndex )->mousePosition[1] = y;
00891 #if defined( _WIN32 ) || defined( _WIN64 )
00892                 Windows_SetMousePosition( GetWindowByIndex( windowIndex ) );
00893 #else
00894                 Linux_SetMousePosition( GetWindowByIndex( windowIndex
       ) );
00895 #endif
00896                 return FOUNDATION_OK;
00897             }
00898             return FOUNDATION_ERROR;
00899         }
00900         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
       );
00901         return FOUNDATION_ERROR;
00902     }
```

3.2.3.98 static GLboolean windowManager::SetMousePositionInWindowByName (const char * *windowName*, GLuint *x*, GLuint *y*) [inline], [static]

set the mouse Position of the given window's co-ordinates < if the window is being used without being initialized

```
00860     {
00861         if ( GetInstance()->IsInitialized() )
00862         {
00863             if ( DoesExistByName( windowName ) )
00864             {
00865                 GetWindowByName( windowName )->mousePosition[0] = x;
00866                 GetWindowByName( windowName )->mousePosition[1] = y;
00867 #if defined( _WIN32 ) || defined( _WIN64 )
00868                 Windows_SetMousePosition( GetWindowByName( windowName ) );
00869 #else
00870                 Linux_SetMousePosition( GetWindowByName( windowName )
      );
00871 #endif
00872                 return FOUNDATION_OK;
00873             }
00874             return FOUNDATION_ERROR;
00875         }
00876
00877         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
       );
00878         return FOUNDATION_ERROR;
00879     }
```

3.2.3.99 static GLboolean windowManager::SetWindowIconByIndex (GLuint *windowIndex*, const char * *icon*, GLuint *width*, GLuint *height*) [inline], [static]

< if the window is being used without being initialized

```
01464     {
01465         if ( GetInstance()->IsInitialized() )
01466         {
01467             if ( DoesExistByIndex( windowIndex ) && IsValid( icon ) )
01468             {
01469 #if defined( _WIN32 ) || defined( _WIN64 )
01470                 Windows_SetWindowIcon( GetWindowByIndex( windowIndex ), icon, width, height
       );
01471 #else
01472                 Linux_SetWindowIcon( GetWindowByIndex( windowIndex ),
      icon, width, height );
01473 #endif
01474                 return FOUNDATION_OK;
01475             }
01476
01477             return FOUNDATION_ERROR;
01478         }
01479         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
       );
01480         return FOUNDATION_ERROR;
01481     }
```

3.2.3.100 static GLboolean windowManager::SetWindowIconByName (const char * *windowName*, const char * *icon*, GLuint *width*, GLuint *height*) [inline], [static]

< if the window is being used without being initialized

```
01445     {
01446         if ( GetInstance()->IsInitialized() )
01447         {
01448             if ( DoesExistByName( windowName ) && IsValid( icon ) )
01449             {
01450 #if defined( _WIN32 ) || defined( _WIN64 )
01451                 Windows_SetWindowIcon( GetWindowByName( windowName ), icon, width, height );
01452 #else
01453                 Linux_SetWindowIcon( GetWindowByName( windowName ), icon,
      width, height );
01454 #endif
01455                 return FOUNDATION_OK;
01456             }
01457             return FOUNDATION_ERROR;
01458         }
01459 
01460         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
     );
01461         return FOUNDATION_ERROR;
01462     }
```

3.2.3.101 static GLboolean windowManager::SetWindowOnDestroyedByIndex (GLuint *windowIndex*, onDestroyedEvent_t *onDestroyed*) [inline], [static]

< if the window is being used without being initialized

```
01881     {
01882         if ( GetInstance()->IsInitialized() )
01883         {
01884             if ( DoesExistByIndex( windowIndex ) )
01885             {
01886                 GetWindowByIndex( windowIndex )->destroyedEvent = onDestroyed
    ;
01887                 return FOUNDATION_OK;
01888             }
01889             return FOUNDATION_ERROR;
01890         }
01891         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
     );
01892         return FOUNDATION_ERROR;
01893     }
```

3.2.3.102 static GLboolean windowManager::SetWindowOnDestroyedByName (const char * *windowName*, onDestroyedEvent_t *onDestroyed*) [inline], [static]

< if the window is being used without being initialized

```
01867     {
01868         if ( GetInstance()->IsInitialized() )
01869         {
01870             if ( DoesExistByName( windowName ) )
01871             {
01872                 GetWindowByName( windowName )->destroyedEvent = onDestroyed;
01873                 return FOUNDATION_OK;
01874             }
01875             return FOUNDATION_ERROR;
01876         }
01877         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
     );
01878         return FOUNDATION_ERROR;
01879     }
```

3.2.3.103 static GLboolean windowManager::SetWindowOnFocusByIndex (GLuint *windowIndex*, onFocusEvent_t *onFocus*) [inline], [static]

< if the window is being used without being initialized

```
01968     {
01969         if ( GetInstance()->IsInitialized() )
01970         {
01971             if ( DoesExistByIndex( windowIndex ) )
01972             {
01973                 GetWindowByIndex( windowIndex )->focusEvent = onFocus;
01974                 return FOUNDATION_OK;
01975             }
01976             return FOUNDATION_ERROR;
01977         }
01978         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01979         return FOUNDATION_ERROR;
01980     }
```

3.2.3.104 static GLboolean windowManager::SetWindowOnFocusByName (const char * *windowName,* onFocusEvent_t *onFocus*) [inline],[static]

< if the window is being used without being initialized

```
01954     {
01955         if ( GetInstance()->IsInitialized() )
01956         {
01957             if ( DoesExistByName( windowName ) )
01958             {
01959                 GetWindowByName( windowName )->focusEvent = onFocus;
01960                 return FOUNDATION_OK;
01961             }
01962             return FOUNDATION_ERROR;
01963         }
01964         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01965         return FOUNDATION_ERROR;
01966     }
```

3.2.3.105 static GLboolean windowManager::SetWindowOnKeyEventByIndex (GLuint *windowIndex,* onKeyEvent_t *onKey*) [inline],[static]

< if the window is being used without being initialized

```
01794     {
01795         if ( GetInstance()->IsInitialized() )
01796         {
01797             if ( DoesExistByIndex( windowIndex ) )
01798             {
01799                 GetWindowByIndex( windowIndex )->keyEvent = onKey;
01800                 return FOUNDATION_OK;
01801             }
01802             return FOUNDATION_ERROR;
01803         }
01804         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01805         return FOUNDATION_ERROR;
01806     }
```

3.2.3.106 static GLboolean windowManager::SetWindowOnKeyEventByName (const char * *windowName,* onKeyEvent_t *onKey*) [inline],[static]

< if the window is being used without being initialized

```
01779     {
01780         if ( GetInstance()->IsInitialized() )
01781         {
01782             if ( DoesExistByName( windowName ) )
01783             {
01784                 GetWindowByName( windowName )->keyEvent = onKey;
01785                 return FOUNDATION_OK;
01786             }
01787
01788             return FOUNDATION_ERROR;
```

```
01789         }
01790         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01791         return FOUNDATION_ERROR;
01792     }
```

3.2.3.107 static GLboolean windowManager::SetWindowOnMaximizedByIndex (GLuint *windowIndex,* onMaximizedEvent_t *onMaximized*) [inline], [static]

< if the window is being used without being initialized

```
01910     {
01911         if ( GetInstance()->IsInitialized() )
01912         {
01913             if ( DoesExistByIndex( windowIndex ) )
01914             {
01915                 GetWindowByIndex( windowIndex )->maximizedEvent = onMaximized
      ;
01916                     return FOUNDATION_OK;
01917             }
01918             return FOUNDATION_ERROR;
01919         }
01920         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01921         return FOUNDATION_ERROR;
01922     }
```

3.2.3.108 static GLboolean windowManager::SetWindowOnMaximizedByName (const char * *windowName,* onMaximizedEvent_t *onMaximized*) [inline], [static]

< if the window is being used without being initialized

```
01896     {
01897         if ( GetInstance()->IsInitialized() )
01898         {
01899             if ( DoesExistByName( windowName ) )
01900             {
01901                 GetWindowByName( windowName )->maximizedEvent = onMaximized;
01902                 return FOUNDATION_OK;
01903             }
01904             return FOUNDATION_ERROR;
01905         }
01906         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01907         return FOUNDATION_ERROR;
01908     }
```

3.2.3.109 static GLboolean windowManager::SetWindowOnMinimizedByIndex (GLuint *windowIndex,* onMinimizedEvent_t *onMinimized*) [inline], [static]

< if the window is being used without being initialized

```
01939     {
01940         if ( GetInstance()->IsInitialized() )
01941         {
01942             if ( DoesExistByIndex( windowIndex ) )
01943             {
01944                 GetWindowByIndex( windowIndex )->minimizedEvent = onMinimized
      ;
01945                 return FOUNDATION_OK;
01946             }
01947             return FOUNDATION_ERROR;
01948         }
01949         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01950         return FOUNDATION_ERROR;
01951     }
```

3.2.3.110 static GLboolean windowManager::SetWindowOnMinimizedByName (const char * *windowName*, onMinimizedEvent_t *onMinimized*) [inline], [static]

< if the window is being used without being initialized

```
01925     {
01926         if ( GetInstance()->IsInitialized() )
01927         {
01928             if ( DoesExistByName( windowName ) )
01929             {
01930                 GetWindowByName( windowName )->minimizedEvent = onMinimized;
01931                 return FOUNDATION_OK;
01932             }
01933             return FOUNDATION_ERROR;
01934         }
01935         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01936         return FOUNDATION_ERROR;
01937     }
```

3.2.3.111 static GLboolean windowManager::SetWindowOnMouseButtonEventByIndex (GLuint *windowIndex*, onMouseButtonEvent_t *onMouseButton*) [inline], [static]

< if the window is being used without being initialized

```
01823     {
01824         if ( GetInstance()->IsInitialized() )
01825         {
01826             if ( DoesExistByIndex( windowIndex ) )
01827             {
01828                 GetWindowByIndex( windowIndex )->
      mouseButtonEvent = onMouseButton;
01829                 return FOUNDATION_OK;
01830             }
01831             return FOUNDATION_ERROR;
01832         }
01833         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01834         return FOUNDATION_ERROR;
01835     }
```

3.2.3.112 static GLboolean windowManager::SetWindowOnMouseButtonEventByName (const char * *windowName*, onMouseButtonEvent_t *onMouseButton*) [inline], [static]

< if the window is being used without being initialized

```
01809     {
01810         if ( GetInstance()->IsInitialized() )
01811         {
01812             if ( DoesExistByName( windowName ) )
01813             {
01814                 GetWindowByName( windowName )->mouseButtonEvent =
      onMouseButton;
01815                 return FOUNDATION_OK;
01816             }
01817             return FOUNDATION_ERROR;
01818         }
01819         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01820         return FOUNDATION_ERROR;
01821     }
```

3.2.3.113 static GLboolean windowManager::SetWindowOnMouseMoveByIndex (GLuint *windowIndex*, onMouseMoveEvent_t *onMouseMove*) [inline], [static]

< if the window is being used without being initialized

```
02055     {
02056         if ( GetInstance()->IsInitialized() )
02057         {
02058             if ( DoesExistByIndex( windowIndex ) )
02059             {
02060                 GetWindowByIndex( windowIndex )->mouseMoveEvent = onMouseMove
    ;
02061                 return FOUNDATION_OK;
02062             }
02063             return FOUNDATION_ERROR;
02064         }
02065         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
     );
02066         return FOUNDATION_ERROR;
02067     }
```

3.2.3.114 static GLboolean windowManager::SetWindowOnMouseMoveByName (const char * *windowName*, onMouseMoveEvent_t *onMouseMove*) [inline], [static]

< if the window is being used without being initialized

```
02041     {
02042         if ( GetInstance()->IsInitialized() )
02043         {
02044             if ( DoesExistByName( windowName ) )
02045             {
02046                 GetWindowByName( windowName )->mouseMoveEvent = onMouseMove;
02047                 return FOUNDATION_OK;
02048             }
02049             return FOUNDATION_ERROR;
02050         }
02051         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
     );
02052         return FOUNDATION_ERROR;
02053     }
```

3.2.3.115 static GLboolean windowManager::SetWindowOnMouseWheelEventByIndex (GLuint *windowIndex*, onMouseWheelEvent_t *onMouseWheel*) [inline], [static]

< if the window is being used without being initialized

```
01852     {
01853         if ( GetInstance()->IsInitialized() )
01854         {
01855             if ( DoesExistByIndex( windowIndex ) )
01856             {
01857                 GetWindowByIndex( windowIndex )->mouseWheelEvent =
    onMouseWheel;
01858                 return FOUNDATION_OK;
01859             }
01860             return FOUNDATION_ERROR;
01861         }
01862         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
     );
01863         return FOUNDATION_ERROR;
01864     }
```

3.2.3.116 static GLboolean windowManager::SetWindowOnMouseWheelEventByName (const char * *windowName*, onMouseWheelEvent_t *onMouseWheel*) [inline], [static]

< if the window is being used without being initialized

```
01838     {
01839         if ( GetInstance()->IsInitialized() )
01840         {
01841             if ( DoesExistByName( windowName ) )
01842             {
01843                 GetWindowByName( windowName )->mouseWheelEvent = onMouseWheel
    ;
01844                 return FOUNDATION_OK;
```

```
01845             }
01846             return FOUNDATION_ERROR;
01847         }
01848         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01849         return FOUNDATION_ERROR;
01850     }
```

3.2.3.117 static GLboolean windowManager::SetWindowOnMovedByIndex (GLuint *windowIndex,* onMovedEvent_t *onMoved*) [inline], [static]

< if the window is being used without being initialized

```
01997     {
01998         if ( GetInstance()->IsInitialized() )
01999         {
02000             if ( DoesExistByIndex( windowIndex ) )
02001             {
02002                 GetWindowByIndex( windowIndex )->movedEvent = onMoved;
02003                 return FOUNDATION_OK;
02004             }
02005             return FOUNDATION_ERROR;
02006         }
02007         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
02008         return FOUNDATION_ERROR;
02009     }
```

3.2.3.118 static GLboolean windowManager::SetWindowOnMovedByName (const char * *windowName,* onMovedEvent_t *onMoved*) [inline], [static]

< if the window is being used without being initialized

```
01983     {
01984         if ( GetInstance()->IsInitialized() )
01985         {
01986             if ( DoesExistByName( windowName ) )
01987             {
01988                 GetWindowByName( windowName )->movedEvent = onMoved;
01989                 return FOUNDATION_OK;
01990             }
01991             return FOUNDATION_ERROR;
01992         }
01993         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01994         return FOUNDATION_ERROR;
01995     }
```

3.2.3.119 static GLboolean windowManager::SetWindowOnResizeByIndex (GLuint *windowIndex,* onResizeEvent_t *onResize*) [inline], [static]

< if the window is being used without being initialized

```
02026     {
02027         if ( GetInstance()->IsInitialized() )
02028         {
02029             if ( DoesExistByIndex( windowIndex ) )
02030             {
02031                 GetWindowByIndex( windowIndex )->resizeEvent = onResize;
02032                 return FOUNDATION_OK;
02033             }
02034             return FOUNDATION_ERROR;
02035         }
02036         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
02037         return FOUNDATION_ERROR;
02038     }
```

3.2.3.120 static GLboolean windowManager::SetWindowOnResizeByName (const char * *windowName*, onResizeEvent_t *onResize*) [inline], [static]

< if the window is being used without being initialized

```
02012     {
02013         if ( GetInstance()->IsInitialized() )
02014         {
02015             if ( DoesExistByName( windowName ) )
02016             {
02017                 GetWindowByName( windowName )->resizeEvent = onResize;
02018                 return FOUNDATION_OK;
02019             }
02020             return FOUNDATION_ERROR;
02021         }
02022         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
02023         return FOUNDATION_ERROR;
02024     }
```

3.2.3.121 static GLboolean windowManager::SetWindowPositionByName (const char * *windowName*, GLuint *x*, GLuint *y*) [inline], [static]

set the Position of the given window relative to screen co-ordinates < if the window is being used without being initialized

```
00742     {
00743         if ( GetInstance()->IsInitialized() )
00744         {
00745             if ( DoesExistByName( windowName ) )
00746             {
00747                 GetWindowByName( windowName )->position[0] = x;
00748                 GetWindowByName( windowName )->position[1] = y;
00749 #if defined( _WIN32 ) || defined( _WIN64 )
00750                 Windows_SetWindowPosition( GetWindowByName( windowName ) );
00751 #else
00752                 Linux_SetWindowPosition( GetWindowByName( windowName
      ) );
00753 #endif
00754                 return FOUNDATION_OK;
00755             }
00756             return FOUNDATION_ERROR;
00757         }
00758
00759         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00760         return FOUNDATION_ERROR;
00761     }
```

3.2.3.122 static GLboolean windowManager::SetWindowPositionByName (GLuint *windowIndex*, GLuint *x*, GLuint *y*) [inline], [static]

set the position of the given window relative to screen co-ordinates < if the window is being used without being initialized

```
00766     {
00767         if ( GetInstance()->IsInitialized() )
00768         {
00769             if ( DoesExistByIndex( windowIndex ) )
00770             {
00771                 GetWindowByIndex( windowIndex )->position[0] = x;
00772                 GetWindowByIndex( windowIndex )->position[1] = y;
00773 #if defined( _WIN32 ) || defined( _WIN64 )
00774                 Windows_SetWindowPosition( GetWindowByIndex( windowIndex ) );
00775 #else
00776                 Linux_SetWindowPosition( GetWindowByIndex(
      windowIndex ) );
00777 #endif
00778                 return FOUNDATION_OK;
00779             }
00780         }
00781
```

```
00782         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00783         return FOUNDATION_ERROR;
00784     }
```

3.2.3.123 static GLboolean windowManager::SetWindowResolutionByIndex (GLuint *windowIndex,* GLuint *width,* GLuint *height*) [inline], [static]

set the Size/Resolution of the given window < if the window is being used without being initialized

```
00644     {
00645         if ( GetInstance()->IsInitialized() )
00646         {
00647             if ( WindowExists( windowIndex ) )
00648             {
00649                 GetWindowByIndex( windowIndex )->resolution[0] = width;
00650                 GetWindowByIndex( windowIndex )->resolution[1] = height;
00651
00652 #if defined( _WIN32 ) || defined( _WIN64 )
00653                 Windows_SetWindowResolution( GetWindowByIndex( windowIndex ) );
00654 #else
00655                 Linux_SetWindowResolution(
      GetWindowByIndex( windowIndex ) );
00656 #endif
00657                 return FOUNDATION_OK;
00658             }
00659             return FOUNDATION_ERROR;
00660         }
00661         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00662         return FOUNDATION_ERROR;
00663     }
```

3.2.3.124 static GLboolean windowManager::SetWindowResolutionByName (const char * *windowName,* GLuint *width,* GLuint *height*) [inline], [static]

set the Size/Resolution of the given window < if the OpenGL context for the window is invalid

```
00620     {
00621         if ( GetInstance()->IsInitialized() )
00622         {
00623             if ( DoesExistByName( windowName ) )
00624             {
00625                 GetWindowByName( windowName )->resolution[0] = width;
00626                 GetWindowByName( windowName )->resolution[1] = height;
00627 #if defined( _WIN32 ) || defined( _WIN64 )
00628                 Windows_SetWindowResolution( GetWindowByName( windowName ) );
00629 #else
00630                 Linux_SetWindowResolution(
      GetWindowByName( windowName ) );
00631 #endif
00632                 return FOUNDATION_OK;
00633             }
00634             return FOUNDATION_ERROR;
00635         }
00636
00637         PrintErrorMessage( TINYWINDOW_ERROR_INVALID_CONTEXT
      );
00638         return FOUNDATION_ERROR;
00639     }
```

3.2.3.125 static GLboolean windowManager::SetWindowStyleByIndex (GLuint *windowIndex,* GLuint *windowStyle*) [inline], [static]

< if the window is being used without being initialized

```
01685     {
01686         if ( GetInstance()->IsInitialized() )
01687         {
01688             if ( DoesExistByIndex( windowIndex ) )
```

```
01689             {
01690 #if defined( _WIN32 ) || defined( _WIN64 )
01691                 Windows_SetWindowStyle( GetWindowByIndex( windowIndex ), windowStyle );
01692 #else
01693                 Linux_SetWindowStyle( GetWindowByIndex( windowIndex ),
      windowStyle );
01694 #endif
01695                 return FOUNDATION_OK;
01696             }
01697             return FOUNDATION_ERROR;
01698         }
01699         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01700         return FOUNDATION_ERROR;
01701     }
```

3.2.3.126 static GLboolean windowManager::SetWindowStyleByName (const char * *windowName,* GLuint *windowStyle*) [inline], [static]

< if the window is being used without being initialized

```
01667     {
01668         if ( GetInstance()->IsInitialized() )
01669         {
01670             if ( DoesExistByName( windowName ) )
01671             {
01672 #if defined( _WIN32 ) || defined( _WIN64 )
01673                 Windows_SetWindowStyle( GetWindowByName( windowName ), windowStyle );
01674 #else
01675                 Linux_SetWindowStyle( GetWindowByName( windowName ),
      windowStyle );
01676 #endif
01677                 return FOUNDATION_OK;
01678             }
01679             return FOUNDATION_ERROR;
01680         }
01681         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01682         return FOUNDATION_ERROR;
01683     }
```

3.2.3.127 static GLboolean windowManager::SetWindowTitleBarByIndex (GLuint *windowIndex,* const char * *newName*) [inline], [static]

< if the window is being used without being initialized

```
01425     {
01426         if ( GetInstance()->IsInitialized() )
01427         {
01428             if ( DoesExistByIndex( windowIndex ) && IsValid( newName ) )
01429             {
01430 #if defined( _WIN32 ) || defined( _WIN64 )
01431                 SetWindowText( GetWindowByIndex( windowIndex )->windowHandle, newName );
01432 #else
01433                 XStoreName( GetDisplay(), GetWindowByIndex( windowIndex )->
      windowHandle, newName );
01434 #endif
01435                 return FOUNDATION_OK;
01436             }
01437             return FOUNDATION_ERROR;
01438         }
01439
01440         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01441         return FOUNDATION_ERROR;
01442     }
```

3.2.3.128 static GLboolean windowManager::SetWindowTitleBarByName (const char * *windowName,* const char * *newTitle*) [inline], [static]

< if the window is being used without being initialized

```
01407     {
01408         if ( GetInstance()->IsInitialized() )
01409         {
01410             if ( DoesExistByName( windowName ) && IsValid( newTitle ) )
01411             {
01412 #if defined( _WIN32 ) || defined( _WIN64 )
01413                 SetWindowText( GetWindowByName( windowName )->windowHandle, newTitle );
01414 #else
01415                 XStoreName( GetDisplay(), GetWindowByName( windowName )->
      windowHandle, newTitle );
01416 #endif
01417                 return FOUNDATION_OK;
01418             }
01419             return FOUNDATION_ERROR;
01420         }
01421         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
       );
01422         return FOUNDATION_ERROR;
01423     }
```

3.2.3.129 static void windowManager::ShutDown (void) [inline], [static]

use this to shutdown the window manager when your program is finished

```
00371     {
00372 #if defined( _MSC_VER )
00373         for each ( auto CurrentWindow in GetInstance()->windowList )
00374         {
00375             delete CurrentWindow;
00376         }
00377 #endif
00378
00379 #if defined( CURRENT_OS_LINUX )
00380         for ( auto CurrentWindow : GetInstance()->windowList )
00381         {
00382             delete CurrentWindow;
00383         }
00384
00385         XCloseDisplay( GetInstance()->currentDisplay );
00386 #endif
00387
00388         GetInstance()->windowList.clear();
00389         delete instance;
00390     }
```

3.2.3.130 static void windowManager::ShutdownWindow (tWindow * *window*) [inline], [static], [private]

```
02318     {
02319 #if defined( _WIN32 ) || defined( _WIN64 )
02320         Windows_ShutdownWindow( window );
02321 #else
02322         Linux_ShutdownWindow( window );
02323 #endif
02324     }
```

3.2.3.131 static void windowManager::WaitForEvents (void) [inline], [static]

< if the window is being used without being initialized

```
01620     {
01621         if ( GetInstance()->IsInitialized() )
01622         {
01623 #if defined( _WIN32 ) || defined( _WIN64 )
01624             GetInstance()->Windows_WaitForEvents();
01625 #else
01626             GetInstance()->Linux_WaitForEvents();
01627 #endif
01628         }
01629
01630         else
01631         {
```

```
01632             PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01633         }
01634     }
```

3.2.3.132 static GLboolean windowManager::WindowExists (GLuint *windowIndex*) [inline], [static], [private]

```
02280     {
02281         return ( windowIndex <= GetInstance()->windowList.size() - 1 );
02282     }
```

3.2.3.133 static GLboolean windowManager::WindowGetKeyByIndex (GLuint *windowIndex,* GLuint *key*) [inline], [static]

returns the current state of the given key relative to the given window < if the window is being used without being initialized

```
00925     {
00926         if ( GetInstance()->IsInitialized() )
00927         {
00928             if ( DoesExistByIndex( windowIndex ) )
00929             {
00930                 return GetWindowByIndex( windowIndex )->keys[key];
00931             }
00932             return FOUNDATION_ERROR;
00933         }
00934         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00935         return FOUNDATION_ERROR;
00936     }
```

3.2.3.134 static GLboolean windowManager::WindowGetKeyByName (const char * *windowName,* GLuint *key*) [inline], [static]

returns the current state of the given key relative to the given window < if the window is being used without being initialized

```
00908     {
00909         if ( GetInstance()->IsInitialized() )
00910         {
00911             if ( DoesExistByName( windowName ) )
00912             {
00913                 return GetWindowByName( windowName )->keys[key];
00914             }
00915
00916             return FOUNDATION_ERROR;
00917         }
00918         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
    );
00919         return FOUNDATION_ERROR;
00920     }
```

3.2.3.135 static GLboolean windowManager::WindowSwapBuffersByIndex (GLuint *windowIndex*) [inline], [static]

swap the draw buffers of the given window < if the window is being used without being initialized

```
00999     {
01000         if ( GetInstance()->IsInitialized() )
01001         {
01002             if ( DoesExistByIndex( windowIndex ) )
01003             {
01004 #if defined( _WIN32 ) || defined( _WIN64 )
01005                 SwapBuffers( GetWindowByIndex( windowIndex )->deviceContextHandle );
```

```
01006 #else
01007                 glXSwapBuffers( GetDisplay(), GetWindowByIndex( windowIndex )->
      windowHandle );
01008 #endif
01009                 return FOUNDATION_OK;
01010             }
01011             return FOUNDATION_ERROR;
01012         }
01013         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
       );
01014         return FOUNDATION_ERROR;
01015     }
```

3.2.3.136 static GLboolean windowManager::WindowSwapBuffersByName (const char * *windowName*) [inline], [static]

swap the draw buffers of the given window < if the window is being used without being initialized

```
00977     {
00978         if ( GetInstance()->IsInitialized() )
00979         {
00980             if ( DoesExistByName( windowName ) )
00981             {
00982 #if defined( _WIN32 ) || defined( _WIN64 )
00983                 SwapBuffers( GetWindowByName( windowName )->deviceContextHandle );
00984 #else
00985                 glXSwapBuffers( GetDisplay(), GetWindowByName( windowName )->
      windowHandle );
00986 #endif
00987                 return FOUNDATION_OK;
00988             }
00989             return FOUNDATION_ERROR;
00990         }
00991
00992         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
       );
00993         return FOUNDATION_ERROR;
00994     }
```

3.2.4 Field Documentation

3.2.4.1 const Display* windowManager::currentDisplay [private]

3.2.4.2 XEvent windowManager::currentEvent [private]

3.2.4.3 windowManager * windowManager::instance = nullptr [static], [private]

3.2.4.4 GLboolean windowManager::isInitialized [private]

3.2.4.5 GLuint windowManager::screenMousePosition[2] [private]

3.2.4.6 GLuint windowManager::screenResolution[2] [private]

3.2.4.7 std::list< tWindow*> windowManager::windowList [private]

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

- TinyWindow.h

Chapter 4

File Documentation

4.1 TinyWindow.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <list>
#include <limits.h>
#include <string.h>
```

Data Structures

- class windowManager
- struct windowManager::tWindow

Macros

- #define DEFAULT_WINDOW_WIDTH 1280
- #define DEFAULT_WINDOW_HEIGHT 720
- #define KEYSTATE_DOWN 1
- #define KEYSTATE_UP 0
- #define KEY_ERROR -1
- #define KEY_FIRST 256 + 1
- #define KEY_F1 KEY_FIRST
- #define KEY_F2 KEY_FIRST + 1
- #define KEY_F3 KEY_FIRST + 2
- #define KEY_F4 KEY_FIRST + 3
- #define KEY_F5 KEY_FIRST + 4
- #define KEY_F6 KEY_FIRST + 5
- #define KEY_F7 KEY_FIRST + 6
- #define KEY_F8 KEY_FIRST + 7
- #define KEY_F9 KEY_FIRST + 8
- #define KEY_F10 KEY_FIRST + 9
- #define KEY_F11 KEY_FIRST + 10
- #define KEY_F12 KEY_FIRST + 11
- #define KEY_CAPSLOCK KEY_FIRST + 12
- #define KEY_LEFTSHIFT KEY_FIRST + 13
- #define KEY_RIGHTSHIFT KEY_FIRST + 14
- #define KEY_LEFTCONTROL KEY_FIRST + 15

- #define KEY_RIGHTCONTROL KEY_FIRST + 16
- #define KEY_LEFTWINDOW KEY_FIRST + 17
- #define KEY_RIGHTWINDOW KEY_FIRST + 18
- #define KEY_LEFTALT KEY_FIRST + 19
- #define KEY_RIGHTALT KEY_FIRST + 20
- #define KEY_ENTER KEY_FIRST + 21
- #define KEY_PRINTSCREEN KEY_FIRST + 22
- #define KEY_SCROLLLOCK KEY_FIRST + 23
- #define KEY_NUMLOCK KEY_FIRST + 24
- #define KEY_PAUSE KEY_FIRST + 25
- #define KEY_INSERT KEY_FIRST + 26
- #define KEY_HOME KEY_FIRST + 27
- #define KEY_END KEY_FIRST + 28
- #define KEY_PAGEUP KEY_FIRST + 28
- #define KEY_PAGEDOWN KEY_FIRST + 30
- #define KEY_ARROW_DOWN KEY_FIRST + 31
- #define KEY_ARROW_UP KEY_FIRST + 32
- #define KEY_ARROW_LEFT KEY_FIRST + 33
- #define KEY_ARROW_RIGHT KEY_FIRST + 34
- #define KEY_KEYPAD_DIVIDE KEY_FIRST + 35
- #define KEY_KEYPAD_MULTIPLY KEY_FIRST + 36
- #define KEY_KEYPAD_SUBTRACT KEY_FIRST + 37
- #define KEY_KEYPAD_ADD KEY_FIRST + 38
- #define KEY_KEYPAD_ENTER KEY_FIRST + 39
- #define KEY_KEYPAD_PERIOD KEY_FIRST + 40
- #define KEY_KEYPAD_0 KEY_FIRST + 41
- #define KEY_KEYPAD_1 KEY_FIRST + 42
- #define KEY_KEYPAD_2 KEY_FIRST + 43
- #define KEY_KEYPAD_3 KEY_FIRST + 44
- #define KEY_KEYPAD_4 KEY_FIRST + 45
- #define KEY_KEYPAD_5 KEY_FIRST + 46
- #define KEY_KEYPAD_6 KEY_FIRST + 47
- #define KEY_KEYPAD_7 KEY_FIRST + 48
- #define KEY_KEYPAD_8 KEY_FIRST + 49
- #define KEY_KEYPAD_9 KEY_FIRST + 50
- #define KEY_BACKSPACE KEY_FIRST + 51
- #define KEY_TAB KEY_FIRST + 52
- #define KEY_DELETE KEY_FIRST + 53
- #define KEY_ESCAPE KEY_FIRST + 54
- #define KEY_LAST KEY_ESCAPE
- #define MOUSE_BUTTONUP 0
- #define MOUSE_BUTTONDOWN 1
- #define MOUSE_LEFTBUTTON 0
- #define MOUSE_RIGHTBUTTON 1
- #define MOUSE_MIDDLEBUTTON 2
- #define MOUSE_LAST MOUSE_MIDDLEBUTTON + 1
- #define MOUSE_SCROLL_DOWN 0
- #define MOUSE_SCROLL_UP 1
- #define WINDOWSTYLE_BARE 1
- #define WINDOWSTYLE_DEFAULT 2
- #define WINDOWSTYLE_POPUP 3
- #define WINDOWSTATE_NORMAL 0
- #define WINDOWSTATE_MAXIMIZED 1
- #define WINDOWSTATE_MINIMIZED 2
- #define WINDOWSTATE_FULLSCREEN 3

- #define DECORATOR_TITLEBAR 0x01
- #define DECORATOR_ICON 0x02
- #define DECORATOR_BORDER 0x04
- #define DECORATOR_MINIMIZEBUTTON 0x08
- #define DECORATOR_MAXIMIZEBUTTON 0x010
- #define DECORATOR_CLOSEBUTTON 0x20
- #define DECORATOR_SIZEABLEBORDER 0x40
- #define LINUX_DECORATOR_BORDER 1L << 1
- #define LINUX_DECORATOR_MOVE 1L << 2
- #define LINUX_DECORATOR_MINIMIZE 1L << 3
- #define LINUX_DECORATOR_MAXIMIZE 1L << 4
- #define LINUX_DECORATOR_CLOSE 1L << 5
- #define FOUNDATION_ERROR 0
- #define FOUNDATION_OK 1
- #define TINYWINDOW_ERROR_NO_CONTEXT 0
- #define TINYWINDOW_ERROR_INVALID_WINDOW_NAME 1
- #define TINYWINDOW_ERROR_INVALID_WINDOW_INDEX 2
- #define TINYWINDOW_ERROR_INVALID_WINDOW_STATE 3
- #define TINYWINDOW_ERROR_INVALID_RESOLUTION 4
- #define TINYWINDOW_ERROR_INVALID_CONTEXT 5
- #define TINYWINDOW_ERROR_EXISTING_CONTEXT 6
- #define TINYWINDOW_ERROR_NOT_INITIALIZED 7
- #define TINYWINDOW_ERROR_ALREADY_INITIALIZED 8
- #define TINYWINDOW_ERROR_INVALID_TITLEBAR 9
- #define TINYWINDOW_ERROR_INVALID_EVENT 10
- #define TINYWIDNOW_ERROR_WINDOW_NOT_FOUND 11
- #define TINYWINDOW_ERROR_INVALID_WINDOWSTYLE 12
- #define TINYWINDOW_ERROR_INVALID_WINDOW 13
- #define TINYWINDOW_ERROR_FUNCTION_NOT_IMPLEMENTED 14
- #define TINYWINDOW_ERROR_LINUX_CANNOT_CONNECT_X_SERVER 15
- #define TINYWINDOW_ERROR_LINUX_INVALID_VISUALINFO 16
- #define TINYWINDOW_ERROR_LINUX_CANNOT_CREATE_WINDOW 17
- #define TINYWINDOW_ERROR_LINUX_FUNCTION_NOT_IMPLEMENTED 18
- #define TINYWINDOW_ERROR_WINDOWS_CANNOT_CREATE_WINDOW 19
- #define TINYWINDOW_ERROR_WINDOWS_CANNOT_INITIALIZE 20
- #define TINYWINDOW_ERROR_WINDOWS_FUNCTION_NOT_IMPLEMENTED 21
- #define TINYWINDOW_WARNING_NOT_CURRENT_CONTEXT 0
- #define TINYWINDOW_WARNING_NO_GL_EXTENSIONS 1
- #define LINUX_FUNCTION 1
- #define LINUX_DECORATOR 2

Typedefs

- typedef void(* onKeyEvent_t)(GLuint key, GLboolean keyState)
- typedef void(* onMouseButtonEvent_t)(GLuint button, GLboolean buttonState)
- typedef void(* onMouseWheelEvent_t)(GLuint wheelDirection)
- typedef void(* onDestroyedEvent_t)(void)
- typedef void(* onMaximizedEvent_t)(void)
- typedef void(* onMinimizedEvent_t)(void)
- typedef void(* onFocusEvent_t)(GLboolean inFocus)
- typedef void(* onMovedEvent_t)(GLuint x, GLuint y)
- typedef void(* onResizeEvent_t)(GLuint width, GLuint height)
- typedef void(* onMouseMoveEvent_t)(GLuint windowX, GLuint windowY, GLuint screenX, GLuint screenY)

Functions

- static void PrintWarningMessage (GLuint warningNumber)
- static void PrintErrorMessage (GLuint errorNumber)

4.1.1 Macro Definition Documentation

4.1.1.1 #define DECORATOR_BORDER 0x04

the border decoration of the window

4.1.1.2 #define DECORATOR_CLOSEBUTTON 0x20

the close button decoration of the window

4.1.1.3 #define DECORATOR_ICON 0x02

the icon decoration of the window

4.1.1.4 #define DECORATOR_MAXIMIZEBUTTON 0x010

the maximize button decoration pf the window

4.1.1.5 #define DECORATOR_MINIMIZEBUTTON 0x08

the minimize button decoration of the window

4.1.1.6 #define DECORATOR_SIZEABLEBORDER 0x40

the sizable border decoration of the window

4.1.1.7 #define DECORATOR_TITLEBAR 0x01

The title bar decoration of the window

4.1.1.8 #define DEFAULT_WINDOW_HEIGHT 720

4.1.1.9 #define DEFAULT_WINDOW_WIDTH 1280

4.1.1.10 #define FOUNDATION_ERROR 0

4.1.1.11 #define FOUNDATION_OK 1

4.1.1.12 #define KEY_ARROW_DOWN KEY_FIRST + 31

the ArrowDown key

4.1.1.13 #define KEY_ARROW_LEFT KEY_FIRST + 33

the ArrowLeft key

4.1.1.14 #define KEY_ARROW_RIGHT KEY_FIRST + 34

the ArrowRight key

4.1.1.15 #define KEY_ARROW_UP KEY_FIRST + 32

the ArrowUp key

4.1.1.16 #define KEY_BACKSPACE KEY_FIRST + 51

the Backspace key

4.1.1.17 #define KEY_CAPSLOCK KEY_FIRST + 12

the CapsLock key

4.1.1.18 #define KEY_DELETE KEY_FIRST + 53

the Delete key

4.1.1.19 #define KEY_END KEY_FIRST + 28

the End key

4.1.1.20 #define KEY_ENTER KEY_FIRST + 21

the Enter/Return key

4.1.1.21 #define KEY_ERROR -1

the key pressed is considered invalid

4.1.1.22 #define KEY_ESCAPE KEY_FIRST + 54

the Escape key

4.1.1.23 #define KEY_F1 KEY_FIRST

the F1 key

4.1.1.24 #define KEY_F10 KEY_FIRST + 9

the F10 key

4.1.1.25 #define KEY_F11 KEY_FIRST + 10

the F11 key

4.1.1.26 #define KEY_F12 KEY_FIRST + 11

the F12 key

4.1.1.27 #define KEY_F2 KEY_FIRST + 1

the F2 key

4.1.1.28 #define KEY_F3 KEY_FIRST + 2

the F3 key

4.1.1.29 #define KEY_F4 KEY_FIRST + 3

the F4 key

4.1.1.30 #define KEY_F5 KEY_FIRST + 4

the F5 key

4.1.1.31 #define KEY_F6 KEY_FIRST + 5

the F6 key

4.1.1.32 #define KEY_F7 KEY_FIRST + 6

the F7 key

4.1.1.33 #define KEY_F8 KEY_FIRST + 7

the F8 key

4.1.1.34 #define KEY_F9 KEY_FIRST + 8

the F9 key

4.1.1.35 #define KEY_FIRST 256 + 1

the fist key that is not a char

4.1.1.36 #define KEY_HOME KEY_FIRST + 27

the Home key

4.1.1.37 #define KEY_INSERT KEY_FIRST + 26

the insert key

4.1.1.38 #define KEY_KEYPAD_0 KEY_FIRST + 41

the Keypad 0 key

4.1.1.39 #define KEY_KEYPAD_1 KEY_FIRST + 42

the Keypad 1 key

4.1.1.40 #define KEY_KEYPAD_2 KEY_FIRST + 43

the Keypad 2 key

4.1.1.41 #define KEY_KEYPAD_3 KEY_FIRST + 44

the Keypad 3 key

4.1.1.42 #define KEY_KEYPAD_4 KEY_FIRST + 45

the Keypad 4 key

4.1.1.43 #define KEY_KEYPAD_5 KEY_FIRST + 46

the Keypad 5 key

4.1.1.44 #define KEY_KEYPAD_6 KEY_FIRST + 47

the Keypad 6 key

4.1.1.45 #define KEY_KEYPAD_7 KEY_FIRST + 48

the Keypad 7 key

4.1.1.46 #define KEY_KEYPAD_8 KEY_FIRST + 49

the keypad 8 key

4.1.1.47 #define KEY_KEYPAD_9 KEY_FIRST + 50

the Keypad 9 key

4.1.1.48 #define KEY_KEYPAD_ADD KEY_FIRST + 38

the Keypad Add key

4.1.1.49 #define KEY_KEYPAD_DIVIDE KEY_FIRST + 35

the KeyPad Divide key

4.1.1.50 #define KEY_KEYPAD_ENTER KEY_FIRST + 39

the Keypad Enter key

4.1.1.51 #define KEY_KEYPAD_MULTIPLY KEY_FIRST + 36

the Keypad Multiply key

4.1.1.52 #define KEY_KEYPAD_PERIOD KEY_FIRST + 40

the Keypad Period/Decimal key

4.1.1.53 #define KEY_KEYPAD_SUBTRACT KEY_FIRST + 37

the Keypad Subtract key

4.1.1.54 #define KEY_LAST KEY_ESCAPE

the last key to be supported

4.1.1.55 #define KEY_LEFTALT KEY_FIRST + 19

the left Alternate key

4.1.1.56 #define KEY_LEFTCONTROL KEY_FIRST + 15

the left Control key

4.1.1.57 #define KEY_LEFTSHIFT KEY_FIRST + 13

the left Shift key

4.1.1.58 #define KEY_LEFTWINDOW KEY_FIRST + 17

the left Window key

4.1.1.59 #define KEY_NUMLOCK KEY_FIRST + 24

the NumLock key

4.1.1.60 #define KEY_PAGEDOWN KEY_FIRST + 30

the PageDown key

4.1.1.61 #define KEY_PAGEUP KEY_FIRST + 28

the PageUp key

4.1.1.62 #define KEY_PAUSE KEY_FIRST + 25

the pause/break key

4.1.1.63 #define KEY_PRINTSCREEN KEY_FIRST + 22

the PrintScreen key

4.1.1.64 #define KEY_RIGHTALT KEY_FIRST + 20

the right Alternate key

4.1.1.65 #define KEY_RIGHTCONTROL KEY_FIRST + 16

the right Control key

4.1.1.66 #define KEY_RIGHTSHIFT KEY_FIRST + 14

the right Shift key

4.1.1.67 #define KEY_RIGHTWINDOW KEY_FIRST + 18

the right Window key

4.1.1.68 #define KEY_SCROLLLOCK KEY_FIRST + 23

the ScrollLock key

4.1.1.69 #define KEY_TAB KEY_FIRST + 52

the Tab key

4.1.1.70 #define KEYSTATE_DOWN 1

the key is currently up

4.1.1.71 #define KEYSTATE_UP 0

the key is currently down

4.1.1.72 #define LINUX_DECORATOR 2

4.1.1.73 #define LINUX_DECORATOR_BORDER 1L << 1

4.1.1.74 #define LINUX_DECORATOR_CLOSE 1L << 5

4.1.1.75 #define LINUX_DECORATOR_MAXIMIZE 1L << 4

4.1.1.76 #define LINUX_DECORATOR_MINIMIZE 1L << 3

4.1.1.77 #define LINUX_DECORATOR_MOVE 1L << 2

4.1.1.78 #define LINUX_FUNCTION 1

4.1.1.79 #define MOUSE_BUTTONDOWN 1

the mouse button is currently down

4.1.1.80 #define MOUSE_BUTTONUP 0

the mouse button is currently up

4.1.1.81 #define MOUSE_LAST MOUSE_MIDDLEBUTTON + 1

the last mouse button to be supported

4.1.1.82 #define MOUSE_LEFTBUTTON 0

the left mouse button

4.1.1.83 #define MOUSE_MIDDLEBUTTON 2

the middle mouse button / ScrollWheel

4.1.1.84 #define MOUSE_RIGHTBUTTON 1

the right mouse button

4.1.1.85 #define MOUSE_SCROLL_DOWN 0

the mouse wheel up

4.1.1.86 #define MOUSE_SCROLL_UP 1

the mouse wheel down

4.1.1.87 #define TINYWIDNOW_ERROR_WINDOW_NOT_FOUND 11

if the window was not found in the window manager

4.1.1.88 #define TINYWINDOW_ERROR_ALREADY_INITIALIZED 8

if the window was already initialized

4.1.1.89 #define TINYWINDOW_ERROR_EXISTING_CONTEXT 6

if the window already has an OpenGL context

4.1.1.90 #define TINYWINDOW_ERROR_FUNCTION_NOT_IMPLEMENTED 14

if the function has not yet been implemented in the current version of the API

4.1.1.91 #define TINYWINDOW_ERROR_INVALID_CONTEXT 5

if the OpenGL context for the window is invalid

4.1.1.92 #define TINYWINDOW_ERROR_INVALID_EVENT 10

if the given event callback was invalid

4.1.1.93 #define TINYWINDOW_ERROR_INVALID_RESOLUTION 4

if an invalid window resolution was given

4.1.1.94 #define TINYWINDOW_ERROR_INVALID_TITLEBAR 9

if the Title-bar text given was invalid

4.1.1.95 #define TINYWINDOW_ERROR_INVALID_WINDOW 13

4.1.1.96 #define TINYWINDOW_ERROR_INVALID_WINDOW_INDEX 2

if an invalid window index was given

4.1.1.97 #define TINYWINDOW_ERROR_INVALID_WINDOW_NAME 1

if an invalid window name was given

4.1.1.98 #define TINYWINDOW_ERROR_INVALID_WINDOW_STATE 3

if an invalid window state was given

4.1.1.99 #define TINYWINDOW_ERROR_INVALID_WINDOWSTYLE 12

if the window style gives is invalid

4.1.1.100 #define TINYWINDOW_ERROR_LINUX_CANNOT_CONNECT_X_SERVER 15

Linux: if cannot connect to X11 server

4.1.1.101 #define TINYWINDOW_ERROR_LINUX_CANNOT_CREATE_WINDOW 17

Linux: when X11 fails to create a new window

4.1.1.102 #define TINYWINDOW_ERROR_LINUX_FUNCTION_NOT_IMPLEMENTED 18

Linux: when the function has not yet been implemented on the Linux in the current version of the API

4.1.1.103 #define TINYWINDOW_ERROR_LINUX_INVALID_VISUALINFO 16

Linux: if visual information given was invalid

4.1.1.104 #define TINYWINDOW_ERROR_NO_CONTEXT 0

if a window tries to use a graphical function without a context

4.1.1.105 #define TINYWINDOW_ERROR_NOT_INITIALIZED 7

if the window is being used without being initialized

4.1.1.106 #define TINYWINDOW_ERROR_WINDOWS_CANNOT_CREATE_WINDOW 19

Windows: when Win32 cannot create a window

4.1.1.107 #define TINYWINDOW_ERROR_WINDOWS_CANNOT_INITIALIZE 20

Windows: when Win32 cannot initialize

4.1.1.108 #define TINYWINDOW_ERROR_WINDOWS_FUNCTION_NOT_IMPLEMENTED 21

Windows: when a function has yet to be implemented on the Windows platform in the current version of the API

4.1.1.109 #define TINYWINDOW_WARNING_NO_GL_EXTENSIONS 1

if your computer does not support any OpenGL extensions

4.1.1.110 #define TINYWINDOW_WARNING_NOT_CURRENT_CONTEXT 0

if using calling member functions of a window that is not the current window being drawn to

4.1.1.111 #define WINDOWSTATE_FULLSCREEN 3

the window is currently full screen

4.1.1.112 #define WINDOWSTATE_MAXIMIZED 1

the window is currently maximized

4.1.1.113 #define WINDOWSTATE_MINIMIZED 2

the window is currently minimized

4.1.1.114 #define WINDOWSTATE_NORMAL 0

the window is in its default state

4.1.1.115 #define WINDOWSTYLE_BARE 1

the window has no decorators but the window border and title bar

4.1.1.116 #define WINDOWSTYLE_DEFAULT 2

the default window style for the respective platform

4.1.1.117 #define WINDOWSTYLE_POPUP 3

the window has no decorators

4.1.2 Typedef Documentation

4.1.2.1 typedef void(* onDestroyedEvent_t)(void)

To be called when the window is being destroyed

4.1.2.2 typedef void(* onFocusEvent_t)(GLboolean inFocus)

To be called when the window has gained event focus

4.1.2.3 typedef void(* onKeyEvent_t)(GLuint key, GLboolean keyState)

To be called when a key event has occurred

4.1.2.4 typedef void(* onMaximizedEvent_t)(void)

To be called when the window has been maximized

4.1.2.5 typedef void(* onMinimizedEvent_t)(void)

To be called when the window has been minimized

4.1.2.6 typedef void(* onMouseButtonEvent_t)(GLuint button, GLboolean buttonState)

To be called when a Mouse button event has occurred

4.1.2.7 typedef void(* onMouseMoveEvent_t)(GLuint windowX, GLuint windowY, GLuint screenX, GLuint screenY)

To be called when the mouse has been moved within the window

4.1.2.8 typedef void(* onMouseWheelEvent_t)(GLuint wheelDirection)

To be called when a mouse wheel event has occurred.

4.1.2.9 typedef void(* onMovedEvent_t)(GLuint x, GLuint y)

To be called when the window has been moved

4.1.2.10 typedef void(* onResizeEvent_t)(GLuint width, GLuint height)

To be called when the window has been resized

4.1.3 Function Documentation

4.1.3.1 static void PrintErrorMessage (GLuint *errorNumber*) [static]

< if a window tries to use a graphical function without a context

< if an invalid window name was given

< if an invalid window index was given

< if an invalid window state was given

< if an invalid window resolution was given

< if the OpenGL context for the window is invalid

< if the window already has an OpenGL context

< if the window is being used without being initialized

< if the window was already initialized

< if the Title-bar text given was invalid

< if the given event callback was invalid

< if the window was not found in the window manager

< if the window style gives is invalid

< if the function has not yet been implemented in the current version of the API

< Linux: if cannot connect to X11 server

< Linux: if visual information given was invalid

< Linux: when X11 fails to create a new window

< Linux: when the function has not yet been implemented on the Linux in the current version of the API

< Windows: when Win32 cannot create a window

< Windows: when a function has yet to be implemented on the Windows platform in the current version of the API

```
00205 {
00206     switch ( errorNumber )
00207     {
00208         case TINYWINDOW_ERROR_NO_CONTEXT:
00209         {
00210             printf( "Error: An OpenGL context must first be created( initialize the window ) \n" );
00211             break;
00212         }
00213 
00214         case TINYWINDOW_ERROR_INVALID_WINDOW_NAME:
00215         {
00216             printf( "Error: invald window name \n" );
00217             break;
00218         }
00219 
00220         case TINYWINDOW_ERROR_INVALID_WINDOW_INDEX:
00221         {
00222             printf( "Error: invalid window index \n" );
00223             break;
00224         }
00225 
00226         case TINYWINDOW_ERROR_INVALID_WINDOW_STATE:
00227         {
00228             printf( "Error: invalid window state \n" );
00229             break;
00230         }
00231 
00232         case TINYWINDOW_ERROR_INVALID_RESOLUTION:
00233         {
```

```
00234             printf( "Error: invalid resolution \n" );
00235             break;
00236         }
00237
00238         case TINYWINDOW_ERROR_INVALID_CONTEXT:
00239         {
00240             printf( "Error: Failed to create OpenGL context \n" );
00241             break;
00242         }
00243
00244         case TINYWINDOW_ERROR_EXISTING_CONTEXT:
00245         {
00246             printf( "Error: context already created \n" );
00247             break;
00248         }
00249
00250         case TINYWINDOW_ERROR_NOT_INITIALIZED:
00251         {
00252             printf( "Error: Window manager not initialized \n" );
00253             break;
00254         }
00255
00256         case TINYWINDOW_ERROR_ALREADY_INITIALIZED:
00257         {
00258             printf( "Error: window has already been initialized \n" );
00259             break;
00260         }
00261
00262         case TINYWINDOW_ERROR_INVALID_TITLEBAR:
00263         {
00264             printf( "Error: invalid title bar name ( cannot be null or nullptr ) \n" );
00265             break;
00266         }
00267
00268         case TINYWINDOW_ERROR_INVALID_EVENT:
00269         {
00270             printf( "Error: invalid event callback given \n" );
00271             break;
00272         }
00273
00274         case TINYWIDNOW_ERROR_WINDOW_NOT_FOUND:
00275         {
00276             printf( "Error: window was not found \n" );
00277             break;
00278         }
00279
00280         case TINYWINDOW_ERROR_INVALID_WINDOWSTYLE:
00281         {
00282             printf( "Error: invalid window style given \n" );
00283             break;
00284         }
00285
00286         case TINYWINDOW_ERROR_INVALID_WINDOW:
00287         {
00288             printf( "Error: invalid window given \n" );
00289             break;
00290         }
00291
00292         case TINYWINDOW_ERROR_FUNCTION_NOT_IMPLEMENTED:
00293         {
00294             printf( "Error: I'm sorry but this function has not been implemented yet :( \n" );
00295             break;
00296         }
00297
00298         case TINYWINDOW_ERROR_LINUX_CANNOT_CONNECT_X_SERVER:
00299         {
00300             printf( "Error: cannot connect to X server \n" );
00301             break;
00302         }
00303
00304         case TINYWINDOW_ERROR_LINUX_INVALID_VISUALINFO:
00305         {
00306             printf( "Error: Invalid visual information given \n" );
00307             break;
00308         }
00309
00310         case TINYWINDOW_ERROR_LINUX_CANNOT_CREATE_WINDOW:
00311         {
00312             printf( "Error: failed to create window \n" );
00313             break;
00314         }
00315
00316         case TINYWINDOW_ERROR_LINUX_FUNCTION_NOT_IMPLEMENTED
    :
00317         {
00318             printf( "Error: function not implemented on linux platform yet. sorry :( \n" );
00319             break;
```

```
00320         }
00321
00322         case TINYWINDOW_ERROR_WINDOWS_CANNOT_CREATE_WINDOW:
00323         {
00324             printf( "Error: failed to create window \n" );
00325             break;
00326         }
00327
00328         case TINYWINDOW_ERROR_WINDOWS_FUNCTION_NOT_IMPLEMENTED
      :
00329         {
00330             printf( "Error: function not implemented on Windows platform yet. sorry ;( \n" );
00331             break;
00332         }
00333
00334         default:
00335         {
00336             printf( "Error: unspecified Error \n" );
00337             break;
00338         }
00339     }
00340 }
```

4.1.3.2 static void PrintWarningMessage (GLuint *warningNumber*) [static]

< if your computer does not support any OpenGL extensions

< if using calling member functions of a window that is not the current window being drawn to

```
00180 {
00181     switch ( warningNumber )
00182     {
00183         case TINYWINDOW_WARNING_NO_GL_EXTENSIONS:
00184         {
00185             printf( "Warning: no OpenGL extensions available \n" );
00186             break;
00187         }
00188
00189         case TINYWINDOW_WARNING_NOT_CURRENT_CONTEXT:
00190         {
00191             printf( "Warning: window not the current OpenGL context being rendered to \n" );
00192             break;
00193         }
00194
00195         default:
00196         {
00197             printf( "Warning: unspecified warning \n" );
00198             break;
00199         }
00200     }
00201 }
```

[twoside]book

fixltx2e calc doxygen graphicx [utf8]inputenc makeidx multicol multirow warntextcomp textcomp [nointegrals]wasysym [table]xcolor

[T1]fontenc mathptmx [scaled=.90]helvet courier amssymb sectsty

geometry a4paper,top=2.5cm,bottom=2.5cm,left=2.5cm,right=2.5cm

fancyhdr

natbib [titles]tocloft

ifpdf [pdftex,pagebackref=true]hyperref

TinyWindow

0.3

Generated by Doxygen 1.8.7

Mon Nov 2 2015 03:32:38

Contents

Chapter 5

Data Structure Index

5.1 Data Structures

Here are the data structures with brief descriptions:

windowManager::tWindow . . . . . ??
windowManager . . . . . ??

Chapter 6

File Index

6.1 File List

Here is a list of all files with brief descriptions:

TinyWindow.h . . . . . ??

Chapter 7

Data Structure Documentation

7.1 windowManager::tWindow Struct Reference

Public Member Functions

- tWindow ()

Data Fields

- const char * name
- GLuint iD
- GLuint colourBits
- GLuint depthBits
- GLuint stencilBits
- GLboolean keys [256+1+54]
- GLboolean mouseButton [2+1]
- GLuint resolution [2]
- GLuint position [2]
- GLuint mousePosition [2]
- GLboolean shouldClose
- GLboolean inFocus
- GLboolean initialized
- GLboolean contextCreated
- GLboolean isCurrentContext
- GLuint currentState
- GLuint currentWindowStyle
- onKeyEvent_t keyEvent
- onMouseButtonEvent_t mouseButtonEvent
- onMouseWheelEvent_t mouseWheelEvent
- onDestroyedEvent_t destroyedEvent
- onMaximizedEvent_t maximizedEvent
- onMinimizedEvent_t minimizedEvent
- onFocusEvent_t focusEvent
- onMovedEvent_t movedEvent
- onResizeEvent_t resizeEvent
- onMouseMoveEvent_t mouseMoveEvent
- Window windowHandle
- GLXContext context
- XVisualInfo * visualInfo

- GLint * attributes
- XSetWindowAttributes setAttributes
- GLbitfield decorators
- Atom AtomState
- Atom AtomHidden
- Atom AtomFullScreen
- Atom AtomMaxHorz
- Atom AtomMaxVert
- Atom AtomClose
- Atom AtomActive
- Atom AtomDemandsAttention
- Atom AtomFocused
- Atom AtomCardinal
- Atom AtomIcon
- Atom AtomHints
- Atom AtomWindowType
- Atom AtomWindowTypeDesktop
- Atom AtomWindowTypeSplash
- Atom AtomWindowTypeNormal
- Atom AtomAllowedActions
- Atom AtomActionResize
- Atom AtomActionMinimize
- Atom AtomActionShade
- Atom AtomActionMaximizeHorz
- Atom AtomActionMaximizeVert
- Atom AtomActionClose
- Atom AtomDesktopGeometry

7.1.1 Detailed Description

7.1.2 Constructor & Destructor Documentation

7.1.2.1 windowManager::tWindow::tWindow ( ) [inline]

< the window is in its default state

< the default window style for the respective platform

```
02074             {
02075                 name = nullptr;
02076                 iD = NULL;
02077                 colourBits = NULL;
02078                 depthBits = NULL;
02079                 stencilBits = NULL;
02080                 shouldClose = GL_FALSE;
02081                 currentState = WINDOWSTATE_NORMAL;
02082 
02083                 keyEvent = nullptr;
02084                 mouseButtonEvent = nullptr;
02085                 mouseWheelEvent = nullptr;
02086                 destroyedEvent = nullptr;
02087                 maximizedEvent = nullptr;
02088                 minimizedEvent = nullptr;
02089                 focusEvent = nullptr;
02090                 movedEvent = nullptr;
02091                 resizeEvent = nullptr;
02092                 mouseMoveEvent = nullptr;
02093 
02094                 initialized = GL_FALSE;
02095                 contextCreated = GL_FALSE;
02096                 currentWindowStyle = WINDOWSTYLE_DEFAULT;
02097 
02098 #if defined( __linux )
02099                 context = 0;
02100 #endif
02101             }
```

7.1.3 Field Documentation

7.1.3.1 Atom windowManager::tWindow::AtomActionClose

atom for allowing the window to be closed

7.1.3.2 Atom windowManager::tWindow::AtomActionMaximizeHorz

atom for allowing the window to be maximized horizontally

7.1.3.3 Atom windowManager::tWindow::AtomActionMaximizeVert

atom for allowing the window to be maximized vertically

7.1.3.4 Atom windowManager::tWindow::AtomActionMinimize

atom for allowing the window to be minimized

7.1.3.5 Atom windowManager::tWindow::AtomActionResize

atom for allowing the window to be resized

7.1.3.6 Atom windowManager::tWindow::AtomActionShade

atom for allowing the window to be shaded

7.1.3.7 Atom windowManager::tWindow::AtomActive

atom for the active window

7.1.3.8 Atom windowManager::tWindow::AtomAllowedActions

atom for allowed window actions

7.1.3.9 Atom windowManager::tWindow::AtomCardinal

atom for cardinal coordinates

7.1.3.10 Atom windowManager::tWindow::AtomClose

atom for closing the window

7.1.3.11 Atom windowManager::tWindow::AtomDemandsAttention

atom for when the window demands attention

7.1.3.12 Atom windowManager::tWindow::AtomDesktopGeometry

atom for Desktop Geometry

7.1.3.13 Atom windowManager::tWindow::AtomFocused

atom for the focused state of the window

7.1.3.14 Atom windowManager::tWindow::AtomFullScreen

atom for the full screen state of the window

7.1.3.15 Atom windowManager::tWindow::AtomHidden

atom for the current hidden state of the window

7.1.3.16 Atom windowManager::tWindow::AtomHints

atom for the window decorations

7.1.3.17 Atom windowManager::tWindow::AtomIcon

atom for the icon of the window

7.1.3.18 Atom windowManager::tWindow::AtomMaxHorz

atom for the maximized horizontally state of the window

7.1.3.19 Atom windowManager::tWindow::AtomMaxVert

atom for the maximized vertically state of the window

7.1.3.20 Atom windowManager::tWindow::AtomState

atom for the state of the window

7.1.3.21 Atom windowManager::tWindow::AtomWindowType

atom for the type of window

7.1.3.22 Atom windowManager::tWindow::AtomWindowTypeDesktop

atom for the desktop window type

7.1.3.23 Atom windowManager::tWindow::AtomWindowTypeNormal

atom for the normal splash screen window type

7.1.3.24 Atom windowManager::tWindow::AtomWindowTypeSplash

atom for the splash screen window type

7.1.3.25 GLint* windowManager::tWindow::attributes

attributes of the window. RGB, depth, stencil, etc

7.1.3.26 GLuint windowManager::tWindow::colourBits

color format of the window. (defaults to 32 bit color)

7.1.3.27 GLXContext windowManager::tWindow::context

the handle to the GLX rendering context

7.1.3.28 GLboolean windowManager::tWindow::contextCreated

whether the OpenGL context has been successfully created

7.1.3.29 GLuint windowManager::tWindow::currentState

The current state of the window. these states include Normal, Minimized, Maximized and Full screen

7.1.3.30 GLuint windowManager::tWindow::currentWindowStyle

the current style of the window

7.1.3.31 GLbitfield windowManager::tWindow::decorators

enabled window decorators

7.1.3.32 GLuint windowManager::tWindow::depthBits

Size of the Depth buffer. (defaults to 8 bit depth)

7.1.3.33 onDestroyedEvent_t windowManager::tWindow::destroyedEvent

this is the callback to be used when the window has been closed in a non-programmatic fashion

7.1.3.34 onFocusEvent_t windowManager::tWindow::focusEvent

this is the callback to be used when the window has been given focus in a non-programmatic fashion

7.1.3.35 GLuint windowManager::tWindow::iD

ID of the Window. (where it belongs in the window manager)

7.1.3.36 GLboolean windowManager::tWindow::inFocus

Whether the Window is currently in focus(if it is the current window be used)

7.1.3.37 GLboolean windowManager::tWindow::initialized

whether the window has been successfully initialized

7.1.3.38 GLboolean windowManager::tWindow::isCurrentContext

whether the window is the current window being drawn to

7.1.3.39 onKeyEvent_t windowManager::tWindow::keyEvent

this is the callback to be used when a key has been pressed

7.1.3.40 GLboolean windowManager::tWindow::keys[256+1+54]

Record of keys that are either pressed or released in the respective window

7.1.3.41 onMaximizedEvent_t windowManager::tWindow::maximizedEvent

this is the callback to be used when the window has been maximized in a non-programmatic fashion

7.1.3.42 onMinimizedEvent_t windowManager::tWindow::minimizedEvent

this is the callback to be used when the window has been minimized in a non-programmatic fashion

7.1.3.43 GLboolean windowManager::tWindow::mouseButton[2+1]

Record of mouse buttons that are either presses or released

7.1.3.44 onMouseButtonEvent_t windowManager::tWindow::mouseButtonEvent

this is the callback to be used when a mouse button has been pressed

7.1.3.45 onMouseMoveEvent_t windowManager::tWindow::mouseMoveEvent

this is a callback to be used when the mouse has been moved

7.1.3.46 GLuint windowManager::tWindow::mousePosition[2]

Position of the Mouse cursor relative to the window co-ordinates

7.1.3.47 onMouseWheelEvent_t windowManager::tWindow::mouseWheelEvent

this is the callback to be used when the mouse wheel has been scrolled.

7.1.3.48 onMovedEvent_t windowManager::tWindow::movedEvent

this is the callback to be used the window has been moved in a non-programmatic fashion

7.1.3.49 const char* windowManager::tWindow::name

Name of the window

7.1.3.50 GLuint windowManager::tWindow::position[2]

Position of the Window relative to the screen co-ordinates

7.1.3.51 onResizeEvent_t windowManager::tWindow::resizeEvent

this is a callback to be used when the window has been resized in a non-programmatic fashion

7.1.3.52 GLuint windowManager::tWindow::resolution[2]

Resolution/Size of the window stored in an array

7.1.3.53 XSetWindowAttributes windowManager::tWindow::setAttributes

the attributes to be set for the window

7.1.3.54 GLboolean windowManager::tWindow::shouldClose

Whether the Window should be closing

7.1.3.55 GLuint windowManager::tWindow::stencilBits

Size of the stencil buffer, (defaults to 8 bit)

7.1.3.56 XVisualInfo* windowManager::tWindow::visualInfo

the handle to the Visual Information. similar purpose to PixelformatDesriptor

7.1.3.57 Window windowManager::tWindow::windowHandle

the X11 handle to the window. I wish they didn't name the type 'Window'

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

- TinyWindow.h

7.2 windowManager Class Reference

```
#include <TinyWindow.h>
```

Data Structures

- struct tWindow

Public Member Functions

- windowManager ()
- ~windowManager (void)

Static Public Member Functions

- static void ShutDown (void)
- static windowManager * AddWindow (const char *windowName, GLuint width=1280, GLuint height=720, GLuint colourBits=8, GLuint depthBits=8, GLuint stencilBits=8)
- static GLuint GetNumWindows (void)
- static GLboolean GetMousePositionInScreen (GLuint &x, GLuint &y)
- static GLuint * GetMousePositionInScreen (void)
- static GLboolean SetMousePositionInScreen (GLuint x, GLuint y)
- static GLuint * GetScreenResolution (void)
- static GLboolean GetScreenResolution (GLuint &width, GLuint &Height)
- static GLboolean GetWindowResolutionByName (const char *windowName, GLuint &width, GLuint &height)
- static GLboolean GetWindowResolutionByIndex (GLuint windowIndex, GLuint &width, GLuint &height)
- static GLuint * GetWindowResolutionByName (const char *windowName)
- static GLuint * GetWindowResolutionByIndex (GLuint windowIndex)
- static GLboolean SetWindowResolutionByName (const char *windowName, GLuint width, GLuint height)
- static GLboolean SetWindowResolutionByIndex (GLuint windowIndex, GLuint width, GLuint height)
- static GLboolean GetWindowPositionByName (const char *windowName, GLuint &x, GLuint &y)
- static GLboolean GetWindowPositionByIndex (GLuint windowIndex, GLuint &x, GLuint &y)
- static GLuint * GetWindowPositionByName (const char *windowName)
- static GLuint * GetWindowPositionByIndex (GLuint windowIndex)
- static GLboolean SetWindowPositionByName (const char *windowName, GLuint x, GLuint y)
- static GLboolean SetWindowPositionByName (GLuint windowIndex, GLuint x, GLuint y)
- static GLboolean GetMousePositionInWindowByName (const char *windowName, GLuint &x, GLuint &y)
- static GLboolean GetMousePositionInWindowByIndex (GLuint windowIndex, GLuint &x, GLuint &y)
- static GLuint * GetMousePositionInWindowByName (const char *windowName)
- static GLuint * GetMousePositionInWindowByIndex (GLuint windowIndex)
- static GLboolean SetMousePositionInWindowByName (const char *windowName, GLuint x, GLuint y)
- static GLboolean SetMousePositionInWindowByIndex (GLuint windowIndex, GLuint x, GLuint y)
- static GLboolean WindowGetKeyByName (const char *windowName, GLuint key)
- static GLboolean WindowGetKeyByIndex (GLuint windowIndex, GLuint key)
- static GLboolean GetWindowShouldCloseByName (const char *windowName)
- static GLboolean GetWindowShouldCloseByIndex (GLuint windowIndex)
- static GLboolean WindowSwapBuffersByName (const char *windowName)
- static GLboolean WindowSwapBuffersByIndex (GLuint windowIndex)
- static GLboolean MakeWindowCurrentContextByName (const char *windowName)
- static GLboolean MakeWindowCurrentContextByIndex (GLuint windowIndex)
- static GLboolean GetWindowIsFullScreenByName (const char *windowName)
- static GLboolean GetWindowIsFullScreenByIndex (GLuint windowIndex)
- static GLboolean SetFullScreenByName (const char *windowName, GLboolean newState)
- static GLboolean SetFullScreenByIndex (GLuint windowIndex, GLboolean newState)
- static GLboolean GetWindowIsMinimizedByName (const char *windowName)
- static GLboolean GetWindowIsMinimizedByIndex (GLuint windowIndex)
- static GLboolean MinimizeWindowByName (const char *windowName, GLboolean newState)
- static GLboolean MinimizeWindowByIndex (GLuint windowIndex, GLboolean newState)
- static GLboolean GetWindowIsMaximizedByName (const char *windowName)
- static GLboolean GetWindowIsMaximizedByIndex (GLuint windowIndex)
- static GLboolean MaximizeWindowByName (const char *windowName, GLboolean newState)
- static GLboolean MaximizeWindowByIndex (GLuint windowIndex, GLboolean newState)

- static const char * GetWindowNameByIndex (GLuint windowIndex)
- static GLuint GetWindowIndexByName (const char *windowName)
- static GLboolean SetWindowTitleBarByName (const char *windowName, const char *newTitle)
- static GLboolean SetWindowTitleBarByIndex (GLuint windowIndex, const char *newName)
- static GLboolean SetWindowIconByName (const char *windowName, const char *icon, GLuint width, GLuint height)
- static GLboolean SetWindowIconByIndex (GLuint windowIndex, const char *icon, GLuint width, GLuint height)
- static GLboolean GetWindowIsInFocusByName (const char *windowName)
- static GLboolean GetWindowIsInFocusByIndex (GLuint windowIndex)
- static GLboolean FocusWindowByName (const char *windowName, GLboolean newState)
- static GLboolean FocusWindowByIndex (GLuint windowIndex, GLboolean newState)
- static GLboolean RestoreWindowByName (const char *windowName)
- static GLboolean RestoreWindowByIndex (GLuint windowIndex)
- static GLboolean Initialize (void)
- static GLboolean IsInitialized (void)
- static void PollForEvents (void)
- static void WaitForEvents (void)
- static GLboolean RemoveWindowByName (const char *windowName)
- static GLboolean RemoveWindowByIndex (GLuint windowIndex)
- static GLboolean SetWindowStyleByName (const char *windowName, GLuint windowStyle)
- static GLboolean SetWindowStyleByIndex (GLuint windowIndex, GLuint windowStyle)
- static GLboolean EnableWindowDecoratorsByName (const char *windowname, GLbitfield decorators)
- static GLboolean EnableWindowDecoratorsByIndex (GLuint windowIndex, GLbitfield decorators)
- static GLboolean DisableWindowDecoratorByName (const char *windowName, GLbitfield decorators)
- static GLboolean DisableWindowDecoratorByIndex (GLuint windowIndex, GLbitfield decorators)
- static GLboolean SetWindowOnKeyEventByName (const char *windowName, onKeyEvent_t onKey)
- static GLboolean SetWindowOnKeyEventByIndex (GLuint windowIndex, onKeyEvent_t onKey)
- static GLboolean SetWindowOnMouseButtonEventByName (const char *windowName, onMouseButton↩ Event_t onMouseButton)
- static GLboolean SetWindowOnMouseButtonEventByIndex (GLuint windowIndex, onMouseButtonEvent_↩ t onMouseButton)
- static GLboolean SetWindowOnMouseWheelEventByName (const char *windowName, onMouseWheel↩ Event_t onMouseWheel)
- static GLboolean SetWindowOnMouseWheelEventByIndex (GLuint windowIndex, onMouseWheelEvent_↩ t onMouseWheel)
- static GLboolean SetWindowOnDestroyedByName (const char *windowName, onDestroyedEvent_t on↩ Destroyed)
- static GLboolean SetWindowOnDestroyedByIndex (GLuint windowIndex, onDestroyedEvent_t onDestroyed)
- static GLboolean SetWindowOnMaximizedByName (const char *windowName, onMaximizedEvent_t on↩ Maximized)
- static GLboolean SetWindowOnMaximizedByIndex (GLuint windowIndex, onMaximizedEvent_t on↩ Maximized)
- static GLboolean SetWindowOnMinimizedByName (const char *windowName, onMinimizedEvent_t on↩ Minimized)
- static GLboolean SetWindowOnMinimizedByIndex (GLuint windowIndex, onMinimizedEvent_t onMinimized)
- static GLboolean SetWindowOnFocusByName (const char *windowName, onFocusEvent_t onFocus)
- static GLboolean SetWindowOnFocusByIndex (GLuint windowIndex, onFocusEvent_t onFocus)
- static GLboolean SetWindowOnMovedByName (const char *windowName, onMovedEvent_t onMoved)
- static GLboolean SetWindowOnMovedByIndex (GLuint windowIndex, onMovedEvent_t onMoved)
- static GLboolean SetWindowOnResizeByName (const char *windowName, onResizeEvent_t onResize)
- static GLboolean SetWindowOnResizeByIndex (GLuint windowIndex, onResizeEvent_t onResize)
- static GLboolean SetWindowOnMouseMoveByName (const char *windowName, onMouseMoveEvent_t on↩ MouseMove)
- static GLboolean SetWindowOnMouseMoveByIndex (GLuint windowIndex, onMouseMoveEvent_t on↩ MouseMove)

Static Private Member Functions

- static tWindow * GetWindowInList (const char *windowName)
- static tWindow * GetWindowInList (GLuint windowIndex)
- static GLboolean IsValid (const char *stringParameter)
- static GLboolean IsValid (onKeyEvent_t onKeyPressed)
- static GLboolean IsValid (onMouseWheelEvent_t onMouseWheelEvent)
- static GLboolean IsValid (onMaximizedEvent_t onMaximized)
- static GLboolean IsValid (onFocusEvent_t onFocus)
- static GLboolean IsValid (onMovedEvent_t onMoved)
- static GLboolean IsValid (onMouseMoveEvent_t onMouseMove)
- static GLboolean WindowExists (GLuint windowIndex)
- static windowManager * GetInstance (void)
- static void InitializeWindow (tWindow *window)
- static void InitializeGL (tWindow *window)
- static void ShutdownWindow (tWindow *window)
- static GLboolean DoesExistByName (const char *windowName)
- static GLboolean DoesExistByIndex (GLuint windowIndex)
- static tWindow * GetWindowByName (const char *windowName)
- static tWindow * GetWindowByIndex (GLuint windowIndex)
- static tWindow * GetWindowByHandle (Window windowHandle)
- static tWindow * GetWindowByEvent (XEvent currentEvent)
- static GLboolean Linux_Initialize (void)
- static void InitializeAtomics (tWindow *window)
- static void Linux_InitializeWindow (tWindow *window)
- static GLboolean Linux_InitializeGL (tWindow *window)
- static void Linux_ShutdownWindow (tWindow *window)
- static void Linux_Shutdown (void)
- static void Linux_Fullscreen (tWindow *window)
- static void Linux_Minimize (tWindow *window)
- static void Linux_Maximize (tWindow *window)
- static void Linux_Restore (tWindow *window)
- static void Linux_Focus (tWindow *window, GLboolean newFocusState)
- static void Linux_SetMousePosition (tWindow *window)
- static void Linux_SetWindowPosition (tWindow *window)
- static void Linux_SetWindowResolution (tWindow *window)
- static void Linux_ProcessEvents (XEvent currentEvent)
- static void Linux_PollForEvents (void)
- static void Linux_WaitForEvents (void)
- static void Linux_SetMousePositionInScreen (GLuint x, GLuint y)
- static Display * GetDisplay (void)
- static const char * Linux_GetEventType (XEvent currentEvent)
- static GLuint Linux_TranslateKey (GLuint keySymbol)
- static void Linux_EnableDecorators (tWindow *window, GLbitfield decorators)
- static void Linux_DisableDecorators (tWindow *window, GLbitfield decorators)
- static void Linux_SetWindowStyle (tWindow *window, GLuint windowStyle)
- static void Linux_SetWindowIcon (tWindow *window, const char *icon, GLuint width, GLuint height)
- static GLXFBConfig GetBestFrameBufferConfig (tWindow *givenWindow)

Private Attributes

- std::list< tWindow * > windowList
- GLuint screenResolution [2]
- GLuint screenMousePosition [2]
- GLboolean isInitialized
- const Display * currentDisplay
- XEvent currentEvent

Static Private Attributes

- static windowManager * instance = nullptr

7.2.1 Detailed Description

7.2.2 Constructor & Destructor Documentation

7.2.2.1 windowManager::windowManager ( ) [inline]

```
00346 {}
```

7.2.2.2 windowManager::~windowManager (void) [inline]

shutdown and delete all windows in the manager

```
00352     {
00353         if ( !GetInstance()->windowList.empty() )
00354         {
00355 #if defined( _MSC_VER )
00356             for each( auto CurrentWindow in GetInstance()->windowList )
00357 #else
00358             for ( auto CurrentWindow : GetInstance()->windowList )
00359 #endif
00360             {
00361                 delete CurrentWindow;
00362             }
00363             GetInstance()->windowList.clear();
00364         }
00365     }
```

7.2.3 Member Function Documentation

7.2.3.1 static windowManager* windowManager::AddWindow (const char * *windowName*, GLuint *width* = 1280, GLuint *height* = 720, GLuint *colourBits* = 8, GLuint *depthBits* = 8, GLuint *stencilBits* = 8) [inline], [static]

use this to add a window to the manager. returns a pointer to the manager which allows for the easy creation of multiple windows < if the window is being used without being initialized

```
00397     {
00398         if ( GetInstance()->IsInitialized() )
00399         {
00400             if ( IsValid( windowName ) )
00401             {
00402                 tWindow* newWindow = new tWindow;
00403                 newWindow->name = windowName;
00404                 newWindow->resolution[0] = width;
00405                 newWindow->resolution[1] = height;
00406                 newWindow->colourBits = colourBits;
00407                 newWindow->depthBits = depthBits;
00408                 newWindow->stencilBits = stencilBits;
00409
00410                 GetInstance()->windowList.push_back( newWindow );
00411                 newWindow->iD = GetNumWindows() - 1;
00412
00413                 InitializeWindow( newWindow );
00414
00415                 return GetInstance();
00416             }
00417             return nullptr;
00418         }
00419
00420         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00421         return nullptr;
00422     }
```

7.2.3.2 static GLboolean windowManager::DisableWindowDecoratorByIndex (GLuint *windowIndex*, GLbitfield *decorators*) [inline], [static]

< if the window is being used without being initialized

```
01759     {
01760         if ( GetInstance()->IsInitialized() )
01761         {
01762             if ( DoesExistByIndex( windowIndex ) )
01763             {
01764 #if defined( _WIN32 ) || defined( _WIN64 )
01765                 Windows_DisableDecorators( GetWindowByIndex( windowIndex ), decorators );
01766 #else
01767                 Linux_DisableDecorators( GetWindowByIndex(
      windowIndex ), decorators );
01768 #endif
01769                 return FOUNDATION_OK;
01770             }
01771             return FOUNDATION_ERROR;
01772         }
01773         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01774         return FOUNDATION_ERROR;
01775     }
```

7.2.3.3 static GLboolean windowManager::DisableWindowDecoratorByName (const char * *windowName*, GLbitfield *decorators*) [inline], [static]

< if the window is being used without being initialized

```
01741     {
01742         if ( GetInstance()->IsInitialized() )
01743         {
01744             if ( DoesExistByName( windowName ) )
01745             {
01746 #if defined( _WIN32 ) || defined( _WIN64 )
01747                 Windows_DisableDecorators( GetWindowByName( windowName ), decorators );
01748 #else
01749                 Linux_DisableDecorators( GetWindowByName( windowName
      ), decorators );
01750 #endif
01751                 return FOUNDATION_OK;
01752             }
01753             return FOUNDATION_ERROR;
01754         }
01755         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01756         return FOUNDATION_ERROR;
01757     }
```

7.2.3.4 static GLboolean windowManager::DoesExistByIndex (GLuint *windowIndex*) [inline], [static], [private]

< if an invalid window index was given

```
02352     {
02353         if ( GetInstance()->IsInitialized() )
02354         {
02355             if ( windowIndex <= ( GetInstance()->windowList.size() - 1 ) )
02356             {
02357                 return FOUNDATION_OK;
02358             }
02359
02360             PrintErrorMessage(
      TINYWINDOW_ERROR_INVALID_WINDOW_INDEX );
02361             return FOUNDATION_ERROR;
02362         }
02363
02364         return FOUNDATION_ERROR;
02365     }
```

7.2.3.5 static GLboolean windowManager::DoesExistByName (const char * *windowName*) [inline], [static], [private]

< if an invalid window name was given

```
02327     {
02328         if ( GetInstance()->IsInitialized() )
02329         {
02330             if ( IsValid( windowName ) )
02331             {
02332 #if defined( _MSC_VER )
02333                 for each( auto window in GetInstance()->windowList )
02334 #else
02335                 for ( auto window : GetInstance()->windowList )
02336 #endif
02337                 {
02338                     if( !strcmp( window->name, windowName ) )
02339                     {
02340                         return GL_TRUE;
02341                     }
02342                 }
02343             }
02344             PrintErrorMessage(
      TINYWINDOW_ERROR_INVALID_WINDOW_NAME );
02345             return GL_FALSE;
02346         }
02347 
02348         return GL_FALSE;
02349     }
```

7.2.3.6 static GLboolean windowManager::EnableWindowDecoratorsByIndex (GLuint *windowIndex*, GLbitfield *decorators*) [inline], [static]

< if the window is being used without being initialized

```
01722     {
01723         if ( GetInstance()->IsInitialized() )
01724         {
01725             if ( DoesExistByIndex( windowIndex ) )
01726             {
01727 #if defined( _WIN32 ) || defined( _WIN64 )
01728                 Windows_EnableDecorators( GetWindowByIndex( windowIndex ), decorators );
01729 #else
01730                 Linux_EnableDecorators( GetWindowByIndex( windowIndex
      ), decorators );
01731 #endif
01732                 return FOUNDATION_OK;
01733             }
01734             return FOUNDATION_ERROR;
01735         }
01736         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01737         return FOUNDATION_ERROR;
01738     }
```

7.2.3.7 static GLboolean windowManager::EnableWindowDecoratorsByName (const char * *windowname*, GLbitfield *decorators*) [inline], [static]

< if the window is being used without being initialized

```
01704     {
01705         if ( GetInstance()->IsInitialized() )
01706         {
01707             if ( DoesExistByName( windowname ) )
01708             {
01709 #if defined( _WIN32 ) || defined( _WIN64 )
01710                 Windows_EnableDecorators( GetWindowByName( windowname ), decorators );
01711 #else
01712                 Linux_EnableDecorators( GetWindowByName( windowname ),
      decorators );
01713 #endif
01714                 return FOUNDATION_OK;
01715             }
```

```
01716             return FOUNDATION_ERROR;
01717         }
01718         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01719         return FOUNDATION_ERROR;
01720     }
```

7.2.3.8 static GLboolean windowManager::FocusWindowByIndex (GLuint *windowIndex*, GLboolean *newState*) [inline], [static]

< if a window tries to use a graphical function without a context

```
01530     {
01531         if ( GetInstance()->IsInitialized() )
01532         {
01533             if ( DoesExistByIndex( windowIndex ) )
01534             {
01535 #if defined( _WIN32 ) || defined( _WIN64 )
01536                 Windows_Focus( GetWindowByIndex( windowIndex ), newState );
01537 #else
01538                 Linux_Focus( GetWindowByIndex( windowIndex ), newState );
01539 #endif
01540                 return FOUNDATION_OK;
01541             }
01542             return FOUNDATION_ERROR;
01543         }
01544         PrintErrorMessage( TINYWINDOW_ERROR_NO_CONTEXT );
01545         return FOUNDATION_ERROR;
01546     }
```

7.2.3.9 static GLboolean windowManager::FocusWindowByName (const char * *windowName*, GLboolean *newState*) [inline], [static]

< if the window is being used without being initialized

```
01512     {
01513         if ( GetInstance()->IsInitialized() )
01514         {
01515             if ( DoesExistByName( windowName ) )
01516             {
01517 #if defined( _WIN32 ) || defined( _WIN64 )
01518                 Windows_Focus( GetWindowByName( windowName ), newState );
01519 #else
01520                 Linux_Focus( GetWindowByName( windowName ), newState );
01521 #endif
01522                 return FOUNDATION_OK;
01523             }
01524             return FOUNDATION_ERROR;
01525         }
01526         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01527         return FOUNDATION_ERROR;
01528     }
```

7.2.3.10 static GLXFBConfig windowManager::GetBestFrameBufferConfig (tWindow * *givenWindow*) [inline], [static], [private]

```
04919     {
04920         const GLint visualAttributes[] =
04921         {
04922             GLX_X_RENDERABLE, GL_TRUE,
04923             GLX_DRAWABLE_TYPE, GLX_WINDOW_BIT,
04924             GLX_X_VISUAL_TYPE, GLX_TRUE_COLOR,
04925             GLX_RED_SIZE, givenWindow->colourBits,
04926             GLX_GREEN_SIZE, givenWindow->colourBits,
04927             GLX_BLUE_SIZE, givenWindow->colourBits,
04928             GLX_ALPHA_SIZE, givenWindow->colourBits,
04929             GLX_DEPTH_SIZE, givenWindow->depthBits,
04930             GLX_STENCIL_SIZE, givenWindow->stencilBits,
04931             GLX_DOUBLEBUFFER, GL_TRUE,
04932             None
```

```
04933         };
04934
04935         GLint frameBufferCount;
04936         GLuint bestBufferConfig, bestNumSamples = 0;
04937         GLXFBConfig* configs = glXChooseFBConfig( GetDisplay(), 0, visualAttributes, &
      frameBufferCount );
04938
04939         for ( GLuint currentConfig = 0; currentConfig < frameBufferCount; currentConfig++ )
04940         {
04941             XVisualInfo* visualInfo = glXGetVisualFromFBConfig( GetDisplay(), configs[
      currentConfig] );
04942
04943             if ( visualInfo )
04944             {
04945                 //printf( "%i %i %i\n", VisInfo->depth, VisInfo->bits_per_rgb, VisInfo->colormap_size );
04946                 GLint samples, sampleBuffer;
04947                 glXGetFBConfigAttrib( GetDisplay(), configs[currentConfig], GLX_SAMPLE_BUFFERS, &
      sampleBuffer );
04948                 glXGetFBConfigAttrib( GetDisplay(), configs[currentConfig], GLX_SAMPLES, &samples
       );
04949
04950                 if ( sampleBuffer && samples > -1 )
04951                 {
04952                     bestBufferConfig = currentConfig;
04953                     bestNumSamples = samples;
04954                 }
04955             }
04956
04957             XFree( visualInfo );
04958         }
04959
04960         GLXFBConfig BestConfig = configs[bestBufferConfig];
04961
04962         XFree( configs );
04963
04964         return BestConfig;
04965     }
```

7.2.3.11 static Display* windowManager::GetDisplay (void) [inline], [static], [private]

```
04318     {
04319         return GetInstance()->currentDisplay;
04320     }
```

7.2.3.12 static windowManager* windowManager::GetInstance (void) [inline], [static], [private]

```
02286     {
02287         if ( windowManager::instance == nullptr )
02288         {
02289             windowManager::instance = new windowManager();
02290             return windowManager::instance;
02291         }
02292
02293         else
02294         {
02295             return windowManager::instance;
02296         }
02297     }
```

7.2.3.13 static GLboolean windowManager::GetMousePositionInScreen (GLuint & *x*, GLuint & *y*) [inline], [static]

return the mouse position in screen co-ordinates < if the window is being used without being initialized

```
00442     {
00443         if ( GetInstance()->IsInitialized() )
00444         {
00445             x = GetInstance()->screenMousePosition[0];
00446             y = GetInstance()->screenMousePosition[1];
00447             return FOUNDATION_OK;
00448         }
00449
00450         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00451         return FOUNDATION_ERROR;
00452     }
```

7.2.3.14 static GLuint* windowManager::GetMousePositionInScreen (void) [inline], [static]

return the mouse position in screen co-ordinates < if the window is being used without being initialized

```
00457     {
00458         if ( GetInstance()->IsInitialized() )
00459         {
00460             return GetInstance()->screenMousePosition;
00461         }
00462 
00463         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00464         return nullptr;
00465     }
```

7.2.3.15 static GLboolean windowManager::GetMousePositionInWindowByIndex (GLuint *windowIndex*, GLuint & *x*, GLuint & *y*) [inline], [static]

return the mouse position relative to the given window's co-ordinates by setting X and Y < if the window is being used without being initialized

```
00808     {
00809         if ( GetInstance()->IsInitialized() )
00810         {
00811             if ( DoesExistByIndex( windowIndex ) )
00812             {
00813                 x = GetWindowByIndex( windowIndex )->
      mousePosition[0];
00814                 y = GetWindowByIndex( windowIndex )->
      mousePosition[1];
00815                 return FOUNDATION_OK;
00816             }
00817             return FOUNDATION_ERROR;
00818         }
00819         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00820         return FOUNDATION_ERROR;
00821     }
```

7.2.3.16 static GLuint* windowManager::GetMousePositionInWindowByIndex (GLuint *windowIndex*) [inline], [static]

return the mouse Position relative to the given window's co-ordinates as an array < if the window is being used without being initialized

```
00843     {
00844         if ( GetInstance()->IsInitialized() )
00845         {
00846             if ( DoesExistByIndex( windowIndex ) )
00847             {
00848                 return GetWindowByIndex( windowIndex )->
      mousePosition;
00849             }
00850             return nullptr;
00851         }
00852         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00853         return nullptr;
00854     }
```

7.2.3.17 static GLboolean windowManager::GetMousePositionInWindowByName (const char * *windowName*, GLuint & *x*, GLuint & *y*) [inline], [static]

return the mouse Position relative to the given window's co-ordinates by setting X and Y < if the window is being used without being initialized

```
00790     {
00791         if ( GetInstance()->IsInitialized() )
00792         {
00793             if ( DoesExistByName( windowName ) )
00794             {
00795                 x = GetWindowByName( windowName )->mousePosition[0];
00796                 y = GetWindowByName( windowName )->mousePosition[1];
00797                 return FOUNDATION_OK;
00798             }
00799             return FOUNDATION_ERROR;
00800         }
00801         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00802         return FOUNDATION_ERROR;
00803     }
```

7.2.3.18 static GLuint* windowManager::GetMousePositionInWindowByName (const char * *windowName*) [inline], [static]

return the mouse Position relative to the given window's co-ordinates as an array < if the window is being used without being initialized

```
00827     {
00828         if ( GetInstance()->IsInitialized() )
00829         {
00830             if ( DoesExistByName( windowName ) )
00831             {
00832                 return GetWindowByName( windowName )->
     mousePosition;
00833             }
00834             return FOUNDATION_ERROR;
00835         }
00836         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00837         return nullptr;
00838     }
```

7.2.3.19 static GLuint windowManager::GetNumWindows (void) [inline], [static]

return the total amount of windows the manager has < if the window is being used without being initialized

```
00428     {
00429         if ( GetInstance()->IsInitialized() )
00430         {
00431             return GetInstance()->windowList.size();
00432         }
00433
00434         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00435         return FOUNDATION_ERROR;
00436     }
```

7.2.3.20 static GLuint* windowManager::GetScreenResolution (void) [inline], [static]

return the Resolution of the current screen < if the window is being used without being initialized

```
00492     {
00493         if ( GetInstance()->IsInitialized() )
00494         {
00495 #if defined( _WIN32 ) || defined( _WIN64 )
00496             RECT screen;
00497             HWND desktop = GetDesktopWindow();
00498             GetWindowRect( desktop, &screen );
00499
00500             GetInstance()->screenResolution[0] = screen.right;
00501             GetInstance()->screenResolution[1] = screen.bottom;
00502             return GetInstance()->screenResolution;
00503
00504 #else
00505             GetInstance()->screenResolution[0] = WidthOfScreen(
     XDefaultScreenOfDisplay( GetInstance()->currentDisplay ) );
```

```
00506             GetInstance()->screenResolution[1] = HeightOfScreen(
      XDefaultScreenOfDisplay( GetInstance()->currentDisplay ) );
00507
00508             return GetInstance()->screenResolution;
00509 #endif
00510         }
00511
00512         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
       );
00513         return nullptr;
00514     }
```

7.2.3.21 static GLboolean windowManager::GetScreenResolution (GLuint & *width*, GLuint & *Height*) [inline], [static]

return the Resolution of the current screen < if the window is being used without being initialized

```
00519     {
00520         if ( GetInstance()->IsInitialized() )
00521         {
00522 #if defined( _WIN32 ) || defined( _WIN64 )
00523             RECT screen;
00524             HWND desktop = GetDesktopWindow();
00525             GetWindowRect( desktop, &screen );
00526             width = screen.right;
00527             Height = screen.bottom;
00528 #else
00529             width = WidthOfScreen( XDefaultScreenOfDisplay( GetInstance()->
      currentDisplay ) );
00530             Height = HeightOfScreen( XDefaultScreenOfDisplay( GetInstance()->
      currentDisplay ) );
00531
00532             GetInstance()->screenResolution[0] = width;
00533             GetInstance()->screenResolution[1] = Height;
00534 #endif
00535             return FOUNDATION_OK;
00536         }
00537
00538         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
       );
00539         return FOUNDATION_ERROR;
00540     }
```

7.2.3.22 static tWindow* windowManager::GetWindowByEvent (XEvent *currentEvent*) [inline], [static], [private]

```
03464     {
03465         switch( currentEvent.type )
03466         {
03467             case Expose:
03468             {
03469                 return GetWindowByHandle( currentEvent.xexpose.window );
03470             }
03471
03472             case DestroyNotify:
03473             {
03474                 return GetWindowByHandle( currentEvent.xdestroywindow.window )
      ;
03475             }
03476
03477             case CreateNotify:
03478             {
03479                 return GetWindowByHandle( currentEvent.xcreatewindow.window );
03480             }
03481
03482             case KeyPress:
03483             {
03484                 return GetWindowByHandle( currentEvent.xkey.window );
03485             }
03486
03487             case KeyRelease:
03488             {
03489                 return GetWindowByHandle( currentEvent.xkey.window );
03490             }
03491
03492             case ButtonPress:
03493             {
```

```
03494                 return GetWindowByHandle( currentEvent.xbutton.window );
03495             }
03496
03497             case ButtonRelease:
03498             {
03499                 return GetWindowByHandle( currentEvent.xbutton.window );
03500             }
03501
03502             case MotionNotify:
03503             {
03504                 return GetWindowByHandle( currentEvent.xmotion.window );
03505             }
03506
03507             case FocusIn:
03508             {
03509                 return GetWindowByHandle( currentEvent.xfocus.window );
03510             }
03511
03512             case FocusOut:
03513             {
03514                 return GetWindowByHandle( currentEvent.xfocus.window );
03515             }
03516
03517             case ResizeRequest:
03518             {
03519                 return GetWindowByHandle( currentEvent.xresizerequest.window )
    ;
03520             }
03521
03522             case ConfigureNotify:
03523             {
03524                 return GetWindowByHandle( currentEvent.xconfigure.window );
03525             }
03526
03527             case PropertyNotify:
03528             {
03529                 return GetWindowByHandle( currentEvent.xproperty.window );
03530             }
03531
03532             case GravityNotify:
03533             {
03534                 return GetWindowByHandle( currentEvent.xgravity.window );
03535             }
03536
03537             case ClientMessage:
03538             {
03539                 return GetWindowByHandle( currentEvent.xclient.window );
03540             }
03541
03542             case VisibilityNotify:
03543             {
03544                 return GetWindowByHandle( currentEvent.xvisibility.window );
03545             }
03546
03547             default:
03548             {
03549                 return nullptr;
03550             }
03551         }
03552     }
```

7.2.3.23 static tWindow* windowManager::GetWindowByHandle (Window *windowHandle*) [inline], [static], [private]

```
03452     {
03453         for( auto iter : GetInstance()->windowList )
03454         {
03455             if ( iter->windowHandle == windowHandle )
03456             {
03457                 return iter;
03458             }
03459         }
03460         return nullptr;
03461     }
```

7.2.3.24 static tWindow* windowManager::GetWindowByIndex (GLuint *windowIndex*) [inline], [static], [private]

```
02386     {
```

```
02387         if ( windowIndex <= GetInstance()->windowList.size() - 1 )
02388         {
02389             return GetWindowInList( windowIndex );
02390         }
02391         return nullptr;
02392     }
```

7.2.3.25 static tWindow* windowManager::GetWindowByName (const char * *windowName*) [inline], [static], [private]

```
02368     {
02369 #if defined( _MSC_VER )
02370             for each( auto window in GetInstance()->windowList )
02371 #else
02372             for( auto window : GetInstance()->windowList )
02373 #endif
02374             {
02375                 if ( !strcmp( window->name, windowName ) )
02376                 {
02377                     return window;
02378                 }
02379             }
02380 
02381             return nullptr;
02382     }
```

7.2.3.26 static GLuint windowManager::GetWindowIndexByName (const char * *windowName*) [inline], [static]

< if the window is being used without being initialized

```
01392     {
01393         if ( GetInstance()->IsInitialized() )
01394         {
01395             if ( DoesExistByName( windowName ) )
01396             {
01397                 return GetWindowByName( windowName )->iD;
01398             }
01399             return FOUNDATION_ERROR;
01400         }
01401         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01402         return FOUNDATION_ERROR;
01403     }
```

7.2.3.27 static tWindow* windowManager::GetWindowInList (const char * *windowName*) [inline], [static], [private]

```
02194     {
02195         if ( IsValid( windowName ) )
02196         {
02197 #if defined( _MSC_VER )
02198             for each ( auto window in GetInstance()->windowList )
02199 #else
02200             for( auto window : GetInstance()->windowList )
02201 #endif
02202             {
02203                 if( window->name == windowName )
02204                 {
02205                     return window;
02206                 }
02207             }
02208 
02209             return nullptr;
02210         }
02211 
02212         return nullptr;
02213     }
```

7.2.3.28 static tWindow* windowManager::GetWindowInList (GLuint *windowIndex*) [inline], [static], [private]

```
02216     {
02217         if ( WindowExists( windowIndex ) )
02218         {
02219 #if defined( _MSC_VER )
02220             for each ( auto window in GetInstance()->windowList )
02221             {
02222                 if ( window->iD == windowIndex )
02223                 {
02224                     return window;
02225                 }
02226             }
02227 #else
02228             for ( auto window : GetInstance()->windowList )
02229             {
02230                 if ( window->iD == windowIndex )
02231                 {
02232                     return window;
02233                 }
02234             }
02235 #endif
02236
02237             return nullptr;
02238         }
02239
02240         return nullptr;
02241     }
```

7.2.3.29 static GLboolean windowManager::GetWindowIsFullScreenByIndex (GLuint *windowIndex*) [inline], [static]

return whether the given window is in fullscreen mode < the window is currently full screen

< if a window tries to use a graphical function without a context

```
01088     {
01089         if ( GetInstance()->IsInitialized() )
01090         {
01091             if ( DoesExistByIndex( windowIndex ) )
01092             {
01093                 return ( GetWindowByIndex( windowIndex )->currentState ==
      WINDOWSTATE_FULLSCREEN );
01094             }
01095
01096             return FOUNDATION_ERROR;
01097         }
01098         PrintErrorMessage( TINYWINDOW_ERROR_NO_CONTEXT );
01099         return FOUNDATION_ERROR;
01100     }
```

7.2.3.30 static GLboolean windowManager::GetWindowIsFullScreenByName (const char * *windowName*) [inline], [static]

return whether the given window is in fullscreen mode < the window is currently full screen

< if a window tries to use a graphical function without a context

```
01071     {
01072         if ( GetInstance()->IsInitialized() )
01073         {
01074             if ( DoesExistByName( windowName ) )
01075             {
01076                 return ( GetWindowByName( windowName )->currentState ==
      WINDOWSTATE_FULLSCREEN );
01077             }
01078
01079             return FOUNDATION_ERROR;
01080         }
01081         PrintErrorMessage( TINYWINDOW_ERROR_NO_CONTEXT );
01082         return FOUNDATION_ERROR;
01083     }
```

7.2.3.31 static GLboolean windowManager::GetWindowIsInFocusByIndex (GLuint *windowIndex*) [inline], [static]

< if the window is being used without being initialized

```
01498     {
01499         if ( GetInstance()->IsInitialized() )
01500         {
01501             if ( DoesExistByIndex( windowIndex ) )
01502             {
01503                 return GetWindowByIndex( windowIndex )->inFocus;
01504             }
01505             return FOUNDATION_ERROR;
01506         }
01507 
01508         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01509         return FOUNDATION_ERROR;
01510     }
```

7.2.3.32 static GLboolean windowManager::GetWindowIsInFocusByName (const char * *windowName*) [inline], [static]

< if the window is being used without being initialized

```
01485     {
01486         if ( GetInstance()->IsInitialized() )
01487         {
01488             if ( DoesExistByName( windowName ) )
01489             {
01490                 return GetWindowByName( windowName )->inFocus;
01491             }
01492             return FOUNDATION_ERROR;
01493         }
01494         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01495         return FOUNDATION_ERROR;
01496     }
```

7.2.3.33 static GLboolean windowManager::GetWindowIsMaximizedByIndex (GLuint *windowIndex*) [inline], [static]

return whether the given window is currently maximized < the window is currently maximized

< if the window is being used without being initialized

```
01304     {
01305         if ( GetInstance()->IsInitialized() )
01306         {
01307             if ( DoesExistByIndex( windowIndex ) )
01308             {
01309                 return ( GetWindowByIndex( windowIndex )->currentState ==
     WINDOWSTATE_MAXIMIZED );
01310             }
01311             return FOUNDATION_ERROR;
01312         }
01313         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01314         return FOUNDATION_ERROR;
01315     }
```

7.2.3.34 static GLboolean windowManager::GetWindowIsMaximizedByName (const char * *windowName*) [inline], [static]

return whether the current window is currently maximized < the window is currently maximized

< if the window is being used without being initialized

```
01287     {
01288         if ( GetInstance()->IsInitialized() )
01289         {
01290             if ( DoesExistByName( windowName ) )
01291             {
01292                 return ( GetWindowByName( windowName )->currentState ==
      WINDOWSTATE_MAXIMIZED );
01293             }
01294 
01295             return FOUNDATION_ERROR;
01296         }
01297         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01298         return FOUNDATION_ERROR;
01299     }
```

7.2.3.35 static GLboolean windowManager::GetWindowIsMinimizedByIndex (GLuint *windowIndex*) [inline], [static]

returns whether the given window is minimized < the window is currently minimized

< if the window is being used without being initialized

```
01196     {
01197         if ( GetInstance()->IsInitialized() )
01198         {
01199             if ( DoesExistByIndex( windowIndex ) )
01200             {
01201                 return ( GetWindowByIndex( windowIndex )->currentState ==
      WINDOWSTATE_MINIMIZED );
01202             }
01203             return FOUNDATION_ERROR;
01204         }
01205         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01206         return FOUNDATION_ERROR;
01207     }
```

7.2.3.36 static GLboolean windowManager::GetWindowIsMinimizedByName (const char * *windowName*) [inline], [static]

returns whether the given window is minimized < the window is currently minimized

< if the window is being used without being initialized

```
01180     {
01181         if ( GetInstance()->IsInitialized() )
01182         {
01183             if ( DoesExistByName( windowName ) )
01184             {
01185                 return ( GetWindowByName( windowName )->currentState ==
      WINDOWSTATE_MINIMIZED );
01186             }
01187             return FOUNDATION_ERROR;
01188         }
01189         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01190         return FOUNDATION_ERROR;
01191     }
```

7.2.3.37 static const char* windowManager::GetWindowNameByIndex (GLuint *windowIndex*) [inline], [static]

gets and sets for window name and index < if the window is being used without being initialized

```
01379     {
01380         if ( GetInstance()->IsInitialized() )
01381         {
01382             if ( DoesExistByIndex( windowIndex ) )
01383             {
01384                 return GetWindowByIndex( windowIndex )->name;
```

```
01385             }
01386             return FOUNDATION_ERROR;
01387         }
01388         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01389         return FOUNDATION_ERROR;
01390     }
```

7.2.3.38 static GLboolean windowManager::GetWindowPositionByIndex (GLuint *windowIndex*, GLuint & *x*, GLuint & *y*) [inline], [static]

return the Position of the given window relative to screen co-ordinates by setting X and Y < if the window is being used without being initialized

```
00688     {
00689         if ( GetInstance()->IsInitialized() )
00690         {
00691             if ( DoesExistByIndex( windowIndex ) )
00692             {
00693                 x = GetWindowByIndex( windowIndex )->position[0];
00694                 y = GetWindowByIndex( windowIndex )->position[1];
00695                 return FOUNDATION_OK;
00696             }
00697             return FOUNDATION_ERROR;
00698         }
00699         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00700         return FOUNDATION_ERROR;
00701     }
```

7.2.3.39 static GLuint* windowManager::GetWindowPositionByIndex (GLuint *windowIndex*) [inline], [static]

return the Position of the given window relative to screen co-ordinates as an array < if the window is being used without being initialized

```
00725     {
00726         if ( GetInstance()->IsInitialized() )
00727         {
00728             if ( DoesExistByIndex( windowIndex ) )
00729             {
00730                 return GetWindowByIndex( windowIndex )->position;
00731             }
00732             return nullptr;
00733         }
00734         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00735         return nullptr;
00736     }
```

7.2.3.40 static GLboolean windowManager::GetWindowPositionByName (const char * *windowName*, GLuint & *x*, GLuint & *y*) [inline], [static]

return the Position of the given window relative to screen co-ordinates by setting X and Y < if the window is being used without being initialized

```
00669     {
00670         if ( GetInstance()->IsInitialized() )
00671         {
00672             if ( DoesExistByName( windowName ) )
00673             {
00674                 x = GetWindowByName( windowName )->position[0];
00675                 y = GetWindowByName( windowName )->position[1];
00676                 return FOUNDATION_OK;
00677             }
00678
00679             return FOUNDATION_ERROR;
00680         }
00681         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00682         return FOUNDATION_ERROR;
00683     }
```

7.2.3.41 static GLuint* windowManager::GetWindowPositionByName (const char * *windowName*) [inline], [static]

return the Position of the given window relative to screen co-ordinates as an array < if the window is being used without being initialized

```
00707     {
00708         if ( GetInstance()->IsInitialized() )
00709         {
00710             if ( DoesExistByName( windowName ) )
00711             {
00712                 return GetWindowByName( windowName )->position;
00713             }
00714 
00715             return nullptr;
00716         }
00717 
00718         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00719         return nullptr;
00720     }
```

7.2.3.42 static GLboolean windowManager::GetWindowResolutionByIndex (GLuint *windowIndex,* GLuint & *width,* GLuint & *height*) [inline], [static]

return the Resolution of the given window by setting width and height < if the window is being used without being initialized

```
00564     {
00565         if ( GetInstance()->IsInitialized() )
00566         {
00567             if ( DoesExistByIndex( windowIndex ) )
00568             {
00569                 width = GetWindowByIndex( windowIndex )->
      resolution[0];
00570                 height = GetWindowByIndex( windowIndex )->
      resolution[1];
00571 
00572                 return FOUNDATION_OK;
00573             }
00574             return FOUNDATION_ERROR;
00575         }
00576 
00577         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00578         return FOUNDATION_ERROR;
00579     }
```

7.2.3.43 static GLuint* windowManager::GetWindowResolutionByIndex (GLuint *windowIndex*) [inline], [static]

return the Resolution of the Given Window as an array of doubles < if the window is being used without being initialized

```
00602     {
00603         if ( GetInstance()->IsInitialized() )
00604         {
00605             if ( DoesExistByIndex( windowIndex ) )
00606             {
00607                 return GetWindowByIndex( windowIndex )->
      resolution;
00608             }
00609             return nullptr;
00610         }
00611 
00612         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00613         return nullptr;
00614     }
```

7.2.3.44 static GLboolean windowManager::GetWindowResolutionByName (const char * *windowName*, GLuint & *width*, GLuint & *height*) [inline], [static]

return the Resolution of the given window by setting width and height < if the window is being used without being initialized

```
00546     {
00547         if ( GetInstance()->IsInitialized() )
00548         {
00549             if ( DoesExistByName( windowName ) )
00550             {
00551                 width = GetWindowByName( windowName )->resolution[0];
00552                 height = GetWindowByName( windowName )->
      resolution[1];
00553                 return FOUNDATION_ERROR;
00554             }
00555             return FOUNDATION_ERROR;
00556         }
00557         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00558         return FOUNDATION_ERROR;
00559     }
```

7.2.3.45 static GLuint* windowManager::GetWindowResolutionByName (const char * *windowName*) [inline], [static]

return the Resolution of the given Window as an array of doubles < if the window is being used without being initialized

```
00585     {
00586         if ( GetInstance()->IsInitialized() )
00587         {
00588             if ( DoesExistByName( windowName ) )
00589             {
00590                 return GetWindowByName( windowName )->resolution;
00591             }
00592             return nullptr;
00593         }
00594
00595         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00596         return nullptr;
00597     }
```

7.2.3.46 static GLboolean windowManager::GetWindowShouldCloseByIndex (GLuint *windowIndex*) [inline], [static]

return whether the given window should be closing < if the window is being used without being initialized

```
00959     {
00960         if ( GetInstance()->IsInitialized() )
00961         {
00962             if ( DoesExistByIndex( windowIndex ) )
00963             {
00964                 return GetWindowByIndex( windowIndex )->
      shouldClose;
00965             }
00966             return FOUNDATION_ERROR;
00967         }
00968
00969         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00970         return FOUNDATION_ERROR;
00971     }
```

7.2.3.47 static GLboolean windowManager::GetWindowShouldCloseByName (const char * *windowName*) [inline], [static]

return whether the given window should be closing < if the window is being used without being initialized

```
00942     {
00943         if ( GetInstance()->IsInitialized() )
00944         {
00945             if ( DoesExistByName( windowName ) )
00946             {
00947                 return GetWindowByName( windowName )->shouldClose;
00948             }
00949             return FOUNDATION_ERROR;
00950         }
00951 
00952         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00953         return FOUNDATION_ERROR;
00954     }
```

7.2.3.48 static GLboolean windowManager::Initialize (void) [inline], [static]

```
01588     {
01589         GetInstance()->isInitialized = GL_FALSE;
01590 #if defined( _WIN32 ) || defined( _WIN64 )
01591         return Windows_Initialize();
01592 #else
01593         return Linux_Initialize();
01594 #endif
01595     }
```

7.2.3.49 static void windowManager::InitializeAtomics (tWindow * *window*) [inline], [static], [private]

```
03574     {
03575         GLuint display = windowManager::GetDisplay();
03576         window->AtomState = XInternAtom( display, "_NET_WM_STATE", GL_FALSE );
03577         window->AtomFullScreen = XInternAtom( display, "_NET_WM_STATE_FULLSCREEN", GL_FALSE );
03578         window->AtomMaxHorz = XInternAtom( display, "_NET_WM_STATE_MAXIMIZED_HORZ", GL_FALSE );
03579         window->AtomMaxVert = XInternAtom( display, "_NET_WM_STATE_MAXIMIZED_VERT", GL_FALSE );
03580         window->AtomClose = XInternAtom( display, "WM_DELETE_WINDOW", GL_FALSE );
03581         window->AtomHidden = XInternAtom( display, "_NET_WM_STATE_HIDDEN", GL_FALSE );
03582         window->AtomActive = XInternAtom( display, "_NET_ACTIVE_WINDOW", GL_FALSE );
03583         window->AtomDemandsAttention = XInternAtom( display, "_NET_WM_STATE_DEMANDS_ATTENTION", GL_FALSE );
03584         window->AtomFocused = XInternAtom( display, "_NET_WM_STATE_FOCUSED", GL_FALSE );
03585         window->AtomCardinal = XInternAtom( display, "CARDINAL", GL_FALSE );
03586         window->AtomIcon = XInternAtom( display, "_NET_WM_ICON", GL_FALSE );
03587         window->AtomHints = XInternAtom( display, "_MOTIF_WM_HINTS", GL_TRUE );
03588 
03589         window->AtomWindowType = XInternAtom( display, "_NET_WM_WINDOW_TYPE", GL_FALSE );
03590         window->AtomWindowTypeDesktop = XInternAtom( display, "_NET_WM_WINDOW_TYPE_UTILITY", GL_FALSE );
03591         window->AtomWindowTypeSplash = XInternAtom( display, "_NET_WM_WINDOW_TYPE_SPLASH", GL_FALSE );
03592         window->AtomWindowTypeNormal = XInternAtom( display, "_NET_WM_WINDOW_TYPE_NORMAL", GL_FALSE );
03593 
03594         window->AtomAllowedActions = XInternAtom( display, "_NET_WM_ALLOWED_ACTIONS", GL_FALSE );
03595         window->AtomActionResize = XInternAtom( display, "WM_ACTION_RESIZE", GL_FALSE );
03596         window->AtomActionMinimize = XInternAtom( display, "_WM_ACTION_MINIMIZE", GL_FALSE );
03597         window->AtomActionShade = XInternAtom( display, "WM_ACTION_SHADE", GL_FALSE );
03598         window->AtomActionMaximizeHorz = XInternAtom( display, "_WM_ACTION_MAXIMIZE_HORZ", GL_FALSE );
03599         window->AtomActionMaximizeVert = XInternAtom( display, "_WM_ACTION_MAXIMIZE_VERT", GL_FALSE );
03600         window->AtomActionClose = XInternAtom( display, "_WM_ACTION_CLOSE", GL_FALSE );
03601 
03602         window->AtomDesktopGeometry = XInternAtom( display, "_NET_DESKTOP_GEOMETRY", GL_FALSE );
03603     }
```

7.2.3.50 static void windowManager::InitializeGL (tWindow * *window*) [inline], [static], [private]

```
02309     {
02310 #if defined( _WIN32 ) || defined( _WIN64 )
02311         Windows_InitializeGL( window );
02312 #else
02313         Linux_InitializeGL( window );
02314 #endif
02315     }
```

7.2.3.51 static void windowManager::InitializeWindow (tWindow * *window*) [inline], [static], [private]

```
02300     {
```

```
02301 #if defined( _WIN32 ) || defined( _WIN64 )
02302         Windows_InitializeWindow( window );
02303 #else
02304         Linux_InitializeWindow( window );
02305 #endif
02306     }
```

7.2.3.52 static GLboolean windowManager::IsInitialized (void) [inline], [static]

```
01598     {
01599         return GetInstance()->isInitialized;
01600     }
```

7.2.3.53 static GLboolean windowManager::IsValid (const char * *stringParameter*) [inline], [static], [private]

```
02245     {
02246         return ( stringParameter != nullptr );
02247     }
```

7.2.3.54 static GLboolean windowManager::IsValid (onKeyEvent_t *onKeyPressed*) [inline], [static], [private]

```
02250     {
02251         return ( onKeyPressed != nullptr );
02252     }
```

7.2.3.55 static GLboolean windowManager::IsValid (onMouseWheelEvent_t *onMouseWheelEvent*) [inline], [static], [private]

```
02255     {
02256         return ( onMouseWheelEvent != nullptr );
02257     }
```

7.2.3.56 static GLboolean windowManager::IsValid (onMaximizedEvent_t *onMaximized*) [inline], [static], [private]

```
02260     {
02261         return ( onMaximized != nullptr );
02262     }
```

7.2.3.57 static GLboolean windowManager::IsValid (onFocusEvent_t *onFocus*) [inline], [static], [private]

```
02265     {
02266         return ( onFocus != nullptr );
02267     }
```

7.2.3.58 static GLboolean windowManager::IsValid (onMovedEvent_t *onMoved*) [inline], [static], [private]

```
02270     {
02271         return ( onMoved != nullptr );
02272     }
```

7.2.3.59 static GLboolean windowManager::IsValid (onMouseMoveEvent_t *onMouseMove*) [inline], [static], [private]

```
02275     {
02276         return ( onMouseMove != nullptr );
02277     }
```

7.2.3.60 static void windowManager::Linux_DisableDecorators (tWindow * *window,* GLbitfield *decorators*) [inline], [static], [private]

< the close button decoration of the window

< the maximize button decoration pf the window

< the minimize button decoration of the window

< the minimize button decoration of the window

< the maximize button decoration pf the window

< the minimize button decoration of the window

< the icon decoration of the window

< The title bar decoration of the window

< the border decoration of the window

< the sizable border decoration of the window

```
04775     {
04776         if ( decorators & DECORATOR_CLOSEBUTTON )
04777         {
04778             //I hate doing this but it is neccessary to keep functionality going.
04779             GLboolean minimizeEnabled, maximizeEnabled;
04780 
04781             if ( decorators & DECORATOR_MAXIMIZEBUTTON )
04782             {
04783                 maximizeEnabled = GL_TRUE;
04784             }
04785 
04786             if ( decorators & DECORATOR_MINIMIZEBUTTON )
04787             {
04788                 minimizeEnabled = GL_TRUE;
04789             }
04790 
04791             window->currentWindowStyle &= ~LINUX_DECORATOR_CLOSE;
04792 
04793             if ( maximizeEnabled )
04794             {
04795                 window->currentWindowStyle |= LINUX_DECORATOR_MAXIMIZE;
04796             }
04797 
04798             if ( minimizeEnabled )
04799             {
04800                 window->currentWindowStyle |= LINUX_DECORATOR_MINIMIZE;
04801             }
04802 
04803             window->decorators = 1;
04804         }
04805 
04806         if ( decorators & DECORATOR_MINIMIZEBUTTON )
04807         {
04808             window->currentWindowStyle &= ~LINUX_DECORATOR_MINIMIZE;
04809             window->decorators = 1;
04810         }
04811 
04812         if ( decorators & DECORATOR_MAXIMIZEBUTTON )
04813         {
04814             GLboolean minimizeEnabled;
04815 
04816             if ( decorators & DECORATOR_MINIMIZEBUTTON )
04817             {
04818                 minimizeEnabled = GL_TRUE;
04819             }
04820 
04821             window->currentWindowStyle &= ~LINUX_DECORATOR_MAXIMIZE;
04822 
04823             if ( minimizeEnabled )
```

```
04824             {
04825                 window->currentWindowStyle |= LINUX_DECORATOR_MINIMIZE;
04826             }
04827 
04828             window->decorators = 1;
04829         }
04830 
04831         if ( decorators & DECORATOR_ICON )
04832         {
04833             //Linux ( at least cinammon ) doesnt have icons in the window. only in the taskbar icon
04834         }
04835 
04836         //just need to set it to 1 to enable all decorators that include title bar
04837         if ( decorators & DECORATOR_TITLEBAR )
04838         {
04839             window->decorators = LINUX_DECORATOR_BORDER;
04840         }
04841 
04842         if ( decorators & DECORATOR_BORDER )
04843         {
04844             window->decorators = 0;
04845         }
04846 
04847         if ( decorators & DECORATOR_SIZEABLEBORDER )
04848         {
04849             window->decorators = 0;
04850         }
04851 
04852         long hints[5] = { LINUX_FUNCTION | LINUX_DECORATOR, window->
    currentWindowStyle, window->decorators, 0, 0 };
04853 
04854         XChangeProperty( GetDisplay(), window->windowHandle, window->AtomHints, XA_ATOM, 32,
04855             PropModeReplace, ( unsigned char* )hints, 5 );
04856 
04857         XMapWindow( GetDisplay(), window->windowHandle );
04858     }
```

7.2.3.61 static void windowManager::Linux_EnableDecorators (tWindow * *window*, GLbitfield *decorators*) [inline], [static], [private]

< the close button decoration of the window

< the minimize button decoration of the window

< the maximize button decoration pf the window

< the icon decoration of the window

< The title bar decoration of the window

< the border decoration of the window

< the sizable border decoration of the window

```
04726     {
04727         if ( decorators & DECORATOR_CLOSEBUTTON )
04728         {
04729             window->currentWindowStyle |= LINUX_DECORATOR_CLOSE;
04730             window->decorators = 1;
04731         }
04732 
04733         if ( decorators & DECORATOR_MINIMIZEBUTTON )
04734         {
04735             window->currentWindowStyle |= LINUX_DECORATOR_MINIMIZE;
04736             window->decorators = 1;
04737         }
04738 
04739         if ( decorators & DECORATOR_MAXIMIZEBUTTON )
04740         {
04741             window->currentWindowStyle |= LINUX_DECORATOR_MAXIMIZE;
04742             window->decorators = 1;
04743         }
04744 
04745         if ( decorators & DECORATOR_ICON )
04746         {
04747             //Linux ( at least cinammon ) doesnt have icons in the window. only in the taskbar icon
04748         }
04749 
04750         //just need to set it to 1 to enable all decorators that include title bar
04751         if ( decorators & DECORATOR_TITLEBAR )
```

```
04752         {
04753             window->decorators = 1;
04754         }
04755 
04756         if ( decorators & DECORATOR_BORDER )
04757         {
04758             window->decorators = 1;
04759         }
04760 
04761         if ( decorators & DECORATOR_SIZEABLEBORDER )
04762         {
04763             window->decorators = 1;
04764         }
04765 
04766         long hints[5] = { LINUX_FUNCTION | LINUX_DECORATOR, window->
      currentWindowStyle, window->decorators, 0, 0 };
04767 
04768         XChangeProperty( GetDisplay(), window->windowHandle, window->AtomHints, XA_ATOM, 32,
04769             PropModeReplace, ( unsigned char* )hints, 5 );
04770 
04771         XMapWindow( GetDisplay(), window->windowHandle );
04772     }
```

7.2.3.62 static void windowManager::Linux_Focus (tWindow * *window,* GLboolean *newFocusState*) [inline], [static], [private]

```
03779     {
03780         if( newFocusState )
03781         {
03782             XMapWindow( windowManager::GetDisplay(), window->windowHandle );
03783         }
03784 
03785         else
03786         {
03787             XUnmapWindow( windowManager::GetDisplay(), window->windowHandle );
03788         }
03789     }
```

7.2.3.63 static void windowManager::Linux_Fullscreen (tWindow * *window*) [inline], [static], [private]

< the window is currently full screen

```
03725     {
03726         XEvent currentEvent;
03727         memset( &currentEvent, 0, sizeof( currentEvent ) );
03728 
03729         currentEvent.xany.type = ClientMessage;
03730         currentEvent.xclient.message_type = window->AtomState;
03731         currentEvent.xclient.format = 32;
03732         currentEvent.xclient.window = window->windowHandle;
03733         currentEvent.xclient.data.l[0] = window->currentState ==
      WINDOWSTATE_FULLSCREEN;
03734         currentEvent.xclient.data.l[1] = window->AtomFullScreen;
03735 
03736         XSendEvent( windowManager::GetDisplay(),
03737             XDefaultRootWindow( windowManager::GetDisplay() ),
03738             0, SubstructureNotifyMask, &currentEvent );
03739     }
```

7.2.3.64 static const char* windowManager::Linux_GetEventType (XEvent *currentEvent*) [inline], [static], [private]

```
04323     {
04324         switch ( currentEvent.type )
04325         {
04326         case MotionNotify:
04327         {
04328             return "Motion Notify Event\n";
04329         }
04330 
04331         case ButtonPress:
04332         {
04333             return "Button Press Event\n";
```

```
04334         }
04335
04336         case ButtonRelease:
04337         {
04338             return "Button Release Event\n";
04339         }
04340
04341         case ColormapNotify:
04342         {
04343             return "Color Map Notify event \n";
04344         }
04345
04346         case EnterNotify:
04347         {
04348             return "Enter Notify Event\n";
04349         }
04350
04351         case LeaveNotify:
04352         {
04353             return "Leave Notify Event\n";
04354         }
04355
04356         case Expose:
04357         {
04358             return "Expose Event\n";
04359         }
04360
04361         case GraphicsExpose:
04362         {
04363             return "Graphics expose event\n";
04364         }
04365
04366         case NoExpose:
04367         {
04368             return "No Expose Event\n";
04369         }
04370
04371         case FocusIn:
04372         {
04373             return "Focus In Event\n";
04374         }
04375
04376         case FocusOut:
04377         {
04378             return "Focus Out Event\n";
04379         }
04380
04381         case KeymapNotify:
04382         {
04383             return "Key Map Notify Event\n";
04384         }
04385
04386         case KeyPress:
04387         {
04388             return "Key Press Event\n";
04389         }
04390
04391         case KeyRelease:
04392         {
04393             return "Key Release Event\n";
04394         }
04395
04396         case PropertyNotify:
04397         {
04398             return "Property Notify Event\n";
04399         }
04400
04401         case ResizeRequest:
04402         {
04403             return "Resize Property Event\n";
04404         }
04405
04406         case CirculateNotify:
04407         {
04408             return "Circulate Notify Event\n";
04409         }
04410
04411         case ConfigureNotify:
04412         {
04413             return "configure Notify Event\n";
04414         }
04415
04416         case DestroyNotify:
04417         {
04418             return "Destroy Notify Request\n";
04419         }
04420
```

```
04421         case GravityNotify:
04422         {
04423             return "Gravity Notify Event \n";
04424         }
04425 
04426         case MapNotify:
04427         {
04428             return "Map Notify Event\n";
04429         }
04430 
04431         case ReparentNotify:
04432         {
04433             return "Reparent Notify Event\n";
04434         }
04435 
04436         case UnmapNotify:
04437         {
04438             return "Unmap notify event\n";
04439         }
04440 
04441         case MapRequest:
04442         {
04443             return "Map request event\n";
04444         }
04445 
04446         case ClientMessage:
04447         {
04448             return "Client Message Event\n";
04449         }
04450 
04451         case MappingNotify:
04452         {
04453             return "Mapping notify event\n";
04454         }
04455 
04456         case SelectionClear:
04457         {
04458             return "Selection Clear event\n";
04459         }
04460 
04461         case SelectionNotify:
04462         {
04463             return "Selection Notify Event\n";
04464         }
04465 
04466         case SelectionRequest:
04467         {
04468             return "Selection Request event\n";
04469         }
04470 
04471         case VisibilityNotify:
04472         {
04473             return "Visibility Notify Event\n";
04474         }
04475 
04476         default:
04477         {
04478             return 0;
04479         }
04480         }
04481     }
```

7.2.3.65 static GLboolean windowManager::Linux_Initialize (void) [inline], [static], [private]

< Linux: if cannot connect to X11 server

```
03555     {
03556         GetInstance()->currentDisplay = XOpenDisplay( 0 );
03557 
03558         if( !GetInstance()->currentDisplay )
03559         {
03560             PrintErrorMessage(
      TINYWINDOW_ERROR_LINUX_CANNOT_CONNECT_X_SERVER );
03561             return FOUNDATION_ERROR;
03562         }
03563 
03564         GetInstance()->screenResolution[0] = WidthOfScreen( XScreenOfDisplay(
      GetInstance()->currentDisplay,
03565             DefaultScreen( GetInstance()->currentDisplay ) ) );
03566 
03567         GetInstance()->screenResolution[1] = HeightOfScreen( XScreenOfDisplay(
      GetInstance()->currentDisplay,
```

```
03568             DefaultScreen( GetInstance()->currentDisplay ) ) );
03569         GetInstance()->isInitialized = GL_TRUE;
03570         return FOUNDATION_OK;
03571     }
```

7.2.3.66 static GLboolean windowManager::Linux_InitializeGL (tWindow * *window*) [inline], [static], [private]

< if the window already has an OpenGL context

```
03669     {
03670         if( !window->context )
03671         {
03672             window->context = glXCreateContext(
03673                     windowManager::GetDisplay(),
03674                 window->visualInfo,
03675                 0,
03676                 GL_TRUE );
03677
03678             if( window->context )
03679             {
03680                 glXMakeCurrent( GetDisplay(),
03681                     window->windowHandle,
03682                     window->context );
03683
03684                 XWindowAttributes l_Attributes;
03685
03686                 XGetWindowAttributes( GetDisplay(),
03687                     window->windowHandle, &l_Attributes );
03688                 window->position[0] = l_Attributes.x;
03689                 window->position[1] = l_Attributes.y;
03690
03691                 window->contextCreated = GL_TRUE;
03692                 return FOUNDATION_OK;
03693             }
03694         }
03695
03696         else
03697         {
03698             PrintErrorMessage(
      TINYWINDOW_ERROR_EXISTING_CONTEXT );
03699             return FOUNDATION_ERROR;
03700         }
03701
03702         return FOUNDATION_ERROR;
03703     }
```

7.2.3.67 static void windowManager::Linux_InitializeWindow (tWindow * *window*) [inline], [static], [private]

< Linux: if cannot connect to X11 server

< Linux: if visual information given was invalid

< Linux: when X11 fails to create a new window

```
03606     {
03607         window->attributes = new GLint[5]{
03608             GLX_RGBA,
03609             GLX_DOUBLEBUFFER,
03610             GLX_DEPTH_SIZE,
03611             window->depthBits,
03612             None};
03613
03614         window->decorators = 1;
03615         window->currentWindowStyle |= LINUX_DECORATOR_CLOSE |
      LINUX_DECORATOR_MAXIMIZE | LINUX_DECORATOR_MINIMIZE |
      LINUX_DECORATOR_MOVE;
03616
03617         if ( !windowManager::GetDisplay() )
03618         {
03619             PrintErrorMessage(
      TINYWINDOW_ERROR_LINUX_CANNOT_CONNECT_X_SERVER );
03620             exit( 0 );
03621         }
```

```
03622
03623         //window->VisualInfo = glXGetVisualFromFBConfig( GetDisplay(), GetBestFrameBufferConfig( window )
      );
03624
03625         window->visualInfo = glXChooseVisual( windowManager::GetDisplay(), 0,
      window->attributes );
03626
03627         if ( !window->visualInfo )
03628         {
03629             PrintErrorMessage(
      TINYWINDOW_ERROR_LINUX_INVALID_VISUALINFO );
03630             exit( 0 );
03631         }
03632
03633         window->setAttributes.colormap = XCreateColormap( GetDisplay(),
03634             DefaultRootWindow( GetDisplay() ),
03635             window->visualInfo->visual, AllocNone );
03636
03637         window->setAttributes.event_mask = ExposureMask | KeyPressMask
03638             | KeyReleaseMask | MotionNotify | ButtonPressMask | ButtonReleaseMask
03639             | FocusIn | FocusOut | Button1MotionMask | Button2MotionMask | Button3MotionMask |
03640             Button4MotionMask | Button5MotionMask | PointerMotionMask | FocusChangeMask
03641             | VisibilityChangeMask | PropertyChangeMask | SubstructureNotifyMask;
03642
03643         window->windowHandle = XCreateWindow( windowManager::GetDisplay(),
03644             XDefaultRootWindow( windowManager::GetDisplay() ), 0, 0,
03645             window->resolution[0], window->resolution[1],
03646             0, window->visualInfo->depth, InputOutput,
03647             window->visualInfo->visual, CWColormap | CWEventMask,
03648             &window->setAttributes );
03649
03650         if( !window->windowHandle )
03651         {
03652             PrintErrorMessage(
      TINYWINDOW_ERROR_LINUX_CANNOT_CREATE_WINDOW );
03653             exit( 0 );
03654         }
03655
03656         XMapWindow( GetDisplay(), window->windowHandle );
03657         XStoreName( GetDisplay(), window->windowHandle,
03658             window->name );
03659
03660         InitializeAtomics( window );
03661
03662         XSetWMProtocols( GetDisplay(), window->windowHandle, &window->AtomClose, GL_TRUE );
03663
03664         Linux_InitializeGL( window );
03665         return GL_TRUE;
03666     }
```

7.2.3.68 static void windowManager::Linux_Maximize (tWindow * *window*) [inline], [static], [private]

< the window is currently maximized

```
03756     {
03757         XEvent currentEvent;
03758         memset( &currentEvent, 0, sizeof( currentEvent ) );
03759
03760         currentEvent.xany.type = ClientMessage;
03761         currentEvent.xclient.message_type = window->AtomState;
03762         currentEvent.xclient.format = 32;
03763         currentEvent.xclient.window = window->windowHandle;
03764         currentEvent.xclient.data.l[0] = ( window->currentState ==
      WINDOWSTATE_MAXIMIZED );
03765         currentEvent.xclient.data.l[1] = window->AtomMaxVert;
03766         currentEvent.xclient.data.l[2] = window->AtomMaxHorz;
03767
03768         XSendEvent( windowManager::GetDisplay(),
03769             XDefaultRootWindow( windowManager::GetDisplay() ),
03770             0, SubstructureNotifyMask, &currentEvent );
03771     }
```

7.2.3.69 static void windowManager::Linux_Minimize (tWindow * *window*) [inline], [static], [private]

< the window is currently minimized

```
03742     {
```

```
03743         if( window->currentState == WINDOWSTATE_MINIMIZED )
03744         {
03745             XIconifyWindow( windowManager::GetDisplay(),
03746                 window->windowHandle, 0 );
03747         }
03748 
03749         else
03750         {
03751             XMapWindow( windowManager::GetDisplay(), window->windowHandle );
03752         }
03753     }
```

7.2.3.70 static void windowManager::Linux_PollForEvents (void) [inline], [static], [private]

```
04285     {
04286         //if there are any events to process
04287         if( XEventsQueued( GetInstance()->GetDisplay(), QueuedAfterReading ) )
04288         {
04289             XNextEvent( GetInstance()->currentDisplay, &
      GetInstance()->currentEvent );
04290 
04291             XEvent currentEvent = GetInstance()->
      currentEvent;
04292 
04293             Linux_ProcessEvents( currentEvent );
04294         }
04295     }
```

7.2.3.71 static void windowManager::Linux_ProcessEvents (XEvent *currentEvent*) [inline], [static], [private]

< the key is currently up

< the key is currently up

< the key is currently up

< the key is currently up

< the key is currently up

< the key is currently down

< the key is currently down

< the key is currently down

< the key is currently down

< the key is currently down

< the left mouse button

< the mouse button is currently down

< the left mouse button

< the mouse button is currently down

< the middle mouse button / ScrollWheel

< the mouse button is currently down

< the middle mouse button / ScrollWheel

< the mouse button is currently down

< the right mouse button

< the mouse button is currently down

< the right mouse button

< the mouse button is currently down

< the mouse wheel down

< the mouse button is currently down

< the mouse wheel up

< the mouse wheel up

< the mouse button is currently down

< the mouse wheel up

< the left mouse button

< the mouse button is currently up

< the left mouse button

< the mouse button is currently up

< the middle mouse button / ScrollWheel

< the mouse button is currently up

< the middle mouse button / ScrollWheel

< the mouse button is currently up

< the right mouse button

< the mouse button is currently up

< the right mouse button

< the mouse button is currently up

< the mouse wheel down

< the mouse button is currently down

< the mouse wheel up

< the mouse button is currently down

set the screen mouse position to match the event

```
03820     {
03821         tWindow* window = GetWindowByEvent( currentEvent );
03822 
03823         switch ( currentEvent.type )
03824         {
03825             case Expose:
03826             {
03827                 break;
03828             }
03829 
03830             case DestroyNotify:
03831             {
03832                 //  printf( "blarg" );
03833 
03834                 if ( IsValid( window->destroyedEvent ) )
03835                 {
03836                     window->destroyedEvent();
03837 
03838                 }
03839 
03840                 printf( "Window was destroyed\n" );
03841                 ShutdownWindow( window );
03842 
03843                 break;
03844 
03845             }
03846 
03847             /*case CreateNotify:
03848             {
03849             printf( "Window was created\n" );
03850             l_Window->InitializeGL();
03851 
03852             if( IsValid( l_Window->m_OnCreated ) )
03853             {
03854             l_Window->m_OnCreated();
03855             }
```

```
03856
03857             break;
03858             } */
03859
03860             case KeyPress:
03861             {
03862                 GLuint functionKeysym = XKeycodeToKeysym(
03863                     GetInstance()->currentDisplay,
    currentEvent.xkey.keycode, 1 );
03864
03865                 if ( functionKeysym <= 255 )
03866                 {
03867                     window->keys[functionKeysym] = KEYSTATE_DOWN;
03868                     if ( IsValid( window->keyEvent ) )
03869                     {
03870                         window->keyEvent( functionKeysym, KEYSTATE_DOWN );
03871                     }
03872                 }
03873
03874                 else
03875                 {
03876                     window->keys[Linux_TranslateKey( functionKeysym )] =
    KEYSTATE_DOWN;
03877
03878                     if ( IsValid( window->keyEvent ) )
03879                     {
03880                         window->keyEvent( Linux_TranslateKey( functionKeysym ),
    KEYSTATE_DOWN );
03881                     }
03882                 }
03883
03884                 break;
03885             }
03886
03887             case KeyRelease:
03888             {
03889                 GLboolean isRetriggered = GL_FALSE;
03890                 if ( XEventsQueued( GetInstance()->currentDisplay,
    QueuedAfterReading ) )
03891                 {
03892                     XEvent nextEvent;
03893                     XPeekEvent( GetInstance()->currentDisplay, &nextEvent );
03894
03895                     if ( nextEvent.type == KeyPress &&
03896                         nextEvent.xkey.time == currentEvent.xkey.time &&
03897                         nextEvent.xkey.keycode == currentEvent.xkey.keycode )
03898                     {
03899                         GLuint functionKeysym = XKeycodeToKeysym( GetInstance()->
    currentDisplay,
03900                         nextEvent.xkey.keycode, 1 );
03901
03902                         XNextEvent( GetInstance()->currentDisplay, &
    currentEvent );
03903                         window->keyEvent( Linux_TranslateKey( functionKeysym ),
    KEYSTATE_DOWN );
03904                         isRetriggered = GL_TRUE;
03905                     }
03906                 }
03907
03908                 if ( !isRetriggered )
03909                 {
03910                     GLuint functionKeysym = XKeycodeToKeysym( GetInstance()->
    currentDisplay,
03911                         currentEvent.xkey.keycode, 1 );
03912
03913                     if ( functionKeysym <= 255 )
03914                     {
03915                         window->keys[functionKeysym] = KEYSTATE_UP;
03916
03917                         if ( IsValid( window->keyEvent ) )
03918                         {
03919                             window->keyEvent( functionKeysym, KEYSTATE_UP );
03920                         }
03921                     }
03922
03923                     else
03924                     {
03925                         window->keys[Linux_TranslateKey( functionKeysym )] =
    KEYSTATE_UP;
03926
03927                         if ( IsValid( window->keyEvent ) )
03928                         {
03929                             window->keyEvent( Linux_TranslateKey( functionKeysym ),
    KEYSTATE_UP );
03930                         }
03931                     }
03932
```

```
03933                     if ( IsValid( window->keyEvent ) )
03934                     {
03935                         window->keyEvent( Linux_TranslateKey( functionKeysym ),
      KEYSTATE_UP );
03936                     }
03937                 }
03938
03939                 break;
03940             }
03941
03942             case ButtonPress:
03943             {
03944                 switch ( currentEvent.xbutton.button )
03945                 {
03946                 case 1:
03947                 {
03948                     window->mouseButton[MOUSE_LEFTBUTTON] =
      MOUSE_BUTTONDOWN;
03949
03950                     if ( IsValid( window->mouseButtonEvent ) )
03951                     {
03952                         window->mouseButtonEvent( MOUSE_LEFTBUTTON,
      MOUSE_BUTTONDOWN );
03953                     }
03954                     break;
03955                 }
03956
03957                 case 2:
03958                 {
03959                     window->mouseButton[MOUSE_MIDDLEBUTTON] =
      MOUSE_BUTTONDOWN;
03960
03961                     if ( IsValid( window->mouseButtonEvent ) )
03962                     {
03963                         window->mouseButtonEvent( MOUSE_MIDDLEBUTTON,
      MOUSE_BUTTONDOWN );
03964                     }
03965                     break;
03966                 }
03967
03968                 case 3:
03969                 {
03970                     window->mouseButton[MOUSE_RIGHTBUTTON] =
      MOUSE_BUTTONDOWN;
03971
03972                     if ( IsValid( window->mouseButtonEvent ) )
03973                     {
03974                         window->mouseButtonEvent( MOUSE_RIGHTBUTTON,
      MOUSE_BUTTONDOWN );
03975                     }
03976                     break;
03977                 }
03978
03979                 case 4:
03980                 {
03981                     window->mouseButton[MOUSE_SCROLL_UP] =
      MOUSE_BUTTONDOWN;
03982
03983                     if ( IsValid( window->mouseWheelEvent ) )
03984                     {
03985                         window->mouseWheelEvent( MOUSE_SCROLL_DOWN );
03986                     }
03987                     break;
03988                 }
03989
03990                 case 5:
03991                 {
03992                     window->mouseButton[MOUSE_SCROLL_DOWN] =
      MOUSE_BUTTONDOWN;
03993
03994                     if ( IsValid( window->mouseWheelEvent ) )
03995                     {
03996                         window->mouseWheelEvent( MOUSE_SCROLL_DOWN );
03997                     }
03998                     break;
03999                 }
04000
04001                 default:
04002                 {
04003                     //need to add more mmouse buttons
04004                     break;
04005                 }
04006                 }
04007
04008                 break;
04009             }
04010
```

```
04011             case ButtonRelease:
04012             {
04013                 switch ( currentEvent.xbutton.button )
04014                 {
04015                 case 1:
04016                 {
04017                     //the left mouse button was released
04018                     window->mouseButton[MOUSE_LEFTBUTTON] =
      MOUSE_BUTTONUP;
04019
04020                     if ( IsValid( window->mouseButtonEvent ) )
04021                     {
04022                         window->mouseButtonEvent( MOUSE_LEFTBUTTON,
      MOUSE_BUTTONUP );
04023                     }
04024                     break;
04025                 }
04026
04027                 case 2:
04028                 {
04029                     //the middle mouse button was released
04030                     window->mouseButton[MOUSE_MIDDLEBUTTON] =
      MOUSE_BUTTONUP;
04031
04032                     if ( IsValid( window->mouseButtonEvent ) )
04033                     {
04034                         window->mouseButtonEvent( MOUSE_MIDDLEBUTTON,
      MOUSE_BUTTONUP );
04035                     }
04036                     break;
04037                 }
04038
04039                 case 3:
04040                 {
04041                     //the right mouse button was released
04042                     window->mouseButton[MOUSE_RIGHTBUTTON] =
      MOUSE_BUTTONUP;
04043
04044                     if ( IsValid( window->mouseButtonEvent ) )
04045                     {
04046                         window->mouseButtonEvent( MOUSE_RIGHTBUTTON,
      MOUSE_BUTTONUP );
04047                     }
04048                     break;
04049                 }
04050
04051                 case 4:
04052                 {
04053                     //the mouse wheel was scrolled up
04054                     window->mouseButton[MOUSE_SCROLL_UP] =
      MOUSE_BUTTONDOWN;
04055                     break;
04056                 }
04057
04058                 case 5:
04059                 {
04060                     //the mouse wheel wasscrolled down
04061                     window->mouseButton[MOUSE_SCROLL_DOWN] =
      MOUSE_BUTTONDOWN;
04062                     break;
04063                 }
04064
04065                 default:
04066                 {
04067                     //need to add more mouse buttons
04068                     break;
04069                 }
04070                 }
04071                 break;
04072             }
04073
04074             //when the mouse/pointer device is moved
04075             case MotionNotify:
04076             {
04077                 //set the windows mouse position to match the event
04078                 window->mousePosition[0] =
04079                     currentEvent.xmotion.x;
04080
04081                 window->mousePosition[1] =
04082                     currentEvent.xmotion.y;
04083
04084                 ///set the screen mouse position to match the event
04085                 GetInstance()->screenMousePosition[0] =
      currentEvent.xmotion.x_root;
04086                 GetInstance()->screenMousePosition[1] =
      currentEvent.xmotion.y_root;
04087
```

```
04088                 if ( IsValid( window->mouseMoveEvent ) )
04089                 {
04090                     window->mouseMoveEvent( currentEvent.xmotion.x,
04091                         currentEvent.xmotion.y, currentEvent.xmotion.x_root,
04092                         currentEvent.xmotion.y_root );
04093                 }
04094                 break;
04095             }
04096
04097             //when the window goes out of focus
04098             case FocusOut:
04099             {
04100                 window->inFocus = GL_FALSE;
04101                 if ( IsValid( window->focusEvent ) )
04102                 {
04103                     window->focusEvent(
04104                         window->inFocus );
04105                 }
04106                 break;
04107             }
04108
04109             //when the window is back in focus ( use to call restore callback? )
04110             case FocusIn:
04111             {
04112                 window->inFocus = GL_TRUE;
04113
04114                 if ( IsValid( window->focusEvent ) )
04115                 {
04116                     window->focusEvent( window->inFocus );
04117                 }
04118                 break;
04119             }
04120
04121             //when a request to resize the window is made either by
04122             //dragging out the window or programmatically
04123             case ResizeRequest:
04124             {
04125                 window->resolution[0] = currentEvent.xresizerequest.width;
04126                 window->resolution[1] = currentEvent.xresizerequest.height;
04127
04128                 glViewport( 0, 0,
04129                     window->resolution[0],
04130                     window->resolution[1] );
04131
04132                 if ( IsValid( window->resizeEvent ) )
04133                 {
04134                     window->resizeEvent( currentEvent.xresizerequest.width,
04135                         currentEvent.xresizerequest.height );
04136                 }
04137
04138                 break;
04139             }
04140
04141             //when a request to configure the window is made
04142             case ConfigureNotify:
04143             {
04144                 glViewport( 0, 0, currentEvent.xconfigure.width,
04145                     currentEvent.xconfigure.height );
04146
04147                 //check if window was resized
04148                 if ( ( GLuint )currentEvent.xconfigure.width != window->resolution[0]
04149                     || ( GLuint )currentEvent.xconfigure.height != window->resolution[1] )
04150                 {
04151                     if ( IsValid( window->resizeEvent ) )
04152                     {
04153                         window->resizeEvent( currentEvent.xconfigure.width,
      currentEvent.xconfigure.height );
04154                     }
04155
04156                     window->resolution[0] = currentEvent.xconfigure.width;
04157                     window->resolution[1] = currentEvent.xconfigure.height;
04158                 }
04159
04160                 //check if window was moved
04161                 if ( ( GLuint )currentEvent.xconfigure.x != window->position[0]
04162                     || ( GLuint )currentEvent.xconfigure.y != window->position[1] )
04163                 {
04164                     if ( IsValid( window->movedEvent ) )
04165                     {
04166                         window->movedEvent( currentEvent.xconfigure.x,
      currentEvent.xconfigure.y );
04167                     }
04168
04169                     window->position[0] = currentEvent.xconfigure.x;
04170                     window->position[1] = currentEvent.xconfigure.y;
04171                 }
04172                 break;
```

```
04173             }
04174
04175             case PropertyNotify:
04176             {
04177                 //this is needed in order to read from the windows WM_STATE Atomic
04178                 //to determine if the property notify event was caused by a client
04179                 //iconify event( minimizing the window ), a maximise event, a focus
04180                 //event and an attention demand event. NOTE these should only be
04181                 //for eventts that are not triggered programatically
04182
04183                 Atom type;
04184                 GLint format;
04185                 ulong numItems, bytesAfter;
04186                 unsigned char*  properties = nullptr;
04187
04188                 XGetWindowProperty( windowManager::GetDisplay(),
      currentEvent.xproperty.window,
04189                     window->AtomState,
04190                     0, LONG_MAX, GL_FALSE, AnyPropertyType,
04191                     &type, &format, &numItems, &bytesAfter,
04192                     & properties );
04193
04194                 if (  properties && ( format == 32 ) )
04195                 {
04196                     //go through each property and match it to an existing Atomic state
04197                     for ( GLuint currentItem = 0; currentItem < numItems; currentItem++ )
04198                     {
04199                         long currentProperty = ( ( long* )(  properties ) )[currentItem];
04200
04201                         if ( currentProperty == window->AtomHidden )
04202                         {
04203                             //window was minimized
04204                             if ( IsValid( window->minimizedEvent ) )
04205                             {
04206                                 //if the minimized callback for the window was set
04207                                 window->minimizedEvent();
04208                             }
04209                         }
04210
04211                         if ( currentProperty == window->AtomMaxVert ||
04212                             currentProperty == window->AtomMaxVert )
04213                         {
04214                             //window was maximized
04215                             if ( IsValid( window->maximizedEvent ) )
04216                             {
04217                                 //if the maximized callback for the window was set
04218                                 window->maximizedEvent();
04219                             }
04220                         }
04221
04222                         if ( currentProperty == window->AtomFocused )
04223                         {
04224                             //window is now in focus. we can ignore this is as FocusIn/FocusOut does this
      anyway
04225                         }
04226
04227                         if ( currentProperty == window->AtomDemandsAttention )
04228                         {
04229                             //the window demands attention like a celebrity
04230                             printf( "window demands attention \n" );
04231                         }
04232                     }
04233                 }
04234
04235                 break;
04236             }
04237
04238             case GravityNotify:
04239             {
04240                 //this is only supposed to pop up when the parent of this window( if any ) has something
      happen
04241                 //to it so that this window can react to said event as well.
04242                 break;
04243             }
04244
04245             //check for events that were created by the TinyWindow manager
04246             case ClientMessage:
04247             {
04248                 const char* atomName = XGetAtomName( windowManager::GetDisplay(),
      currentEvent.xclient.message_type );
04249                 if ( IsValid( atomName ) )
04250                 {
04251                     //printf( "%s\n", l_AtomName );
04252                 }
04253
04254                 if ( ( Atom )currentEvent.xclient.data.l[0] == window->AtomClose )
04255                 {
```

```
04256                     printf( "window closed\n" );
04257                     window->shouldClose = GL_TRUE;
04258                     if( IsValid( window->destroyedEvent ) )
04259                     {
04260                         window->destroyedEvent();
04261                     }
04262                     ShutdownWindow( window );
04263 
04264                     break;
04265 
04266                 }
04267 
04268                 //check if fullscreen
04269                 if ( ( Atom )currentEvent.xclient.data.l[1] == window->AtomFullScreen )
04270                 {
04271                     break;
04272                 }
04273                 break;
04274 
04275             }
04276 
04277             default:
04278             {
04279                 return;
04280             }
04281         }
04282     }
```

7.2.3.72 static void windowManager::Linux_Restore (tWindow * *window*) [inline], [static], [private]

```
03774     {
03775         XMapWindow( windowManager::GetDisplay(), window->windowHandle );
03776     }
```

7.2.3.73 static void windowManager::Linux_SetMousePosition (tWindow * *window*) [inline], [static], [private]

```
03792     {
03793         XWarpPointer(
03794             windowManager::GetInstance()->
      currentDisplay,
03795             window->windowHandle, window->windowHandle,
03796             window->position[0], window->position[1],
03797             window->resolution[0], window->resolution[1],
03798             window->mousePosition[0], window->mousePosition[1] );
03799     }
```

7.2.3.74 static void windowManager::Linux_SetMousePositionInScreen (GLuint *x*, GLuint *y*) [inline], [static], [private]

```
04308     {
04309         XWarpPointer( GetInstance()->currentDisplay, None,
04310             XDefaultRootWindow( GetInstance()->currentDisplay ), 0, 0,
04311             GetScreenResolution()[0],
04312             GetScreenResolution()[1],
04313             x, y );
04314     }
```

7.2.3.75 static void windowManager::Linux_SetWindowIcon (tWindow * *window*, const char * *icon*, GLuint *width*, GLuint *height*) [inline], [static], [private]

< Linux: when the function has not yet been implemented on the Linux in the current version of the API

```
04913     {
04914         //sorry :(
04915         PrintErrorMessage(
      TINYWINDOW_ERROR_LINUX_FUNCTION_NOT_IMPLEMENTED );
04916     }
```

7.2.3.76 static void windowManager::Linux_SetWindowPosition (tWindow * *window*) [inline], [static], [private]

```
03802     {
03803         XWindowChanges windowChanges;
03804 
03805         windowChanges.x = window->position[0];
03806         windowChanges.y = window->position[1];
03807 
03808         XConfigureWindow(
03809             windowManager::GetDisplay(),
03810             window->windowHandle, CWX | CWY, &windowChanges );
03811     }
```

7.2.3.77 static void windowManager::Linux_SetWindowResolution (tWindow * *window*) [inline], [static], [private]

```
03814     {
03815         XResizeWindow( windowManager::GetDisplay(),
03816             window->windowHandle, window->resolution[0], window->resolution[1] );
03817     }
```

7.2.3.78 static void windowManager::Linux_SetWindowStyle (tWindow * *window*, GLuint *windowStyle*) [inline], [static], [private]

< the default window style for the respective platform

< the window has no decorators but the window border and title bar

< the window has no decorators

< if the window style gives is invalid

```
04861     {
04862         switch ( windowStyle )
04863         {
04864         case WINDOWSTYLE_DEFAULT:
04865         {
04866             window->decorators = ( 1L << 2 );
04867             window->currentWindowStyle = LINUX_DECORATOR_MOVE |
      LINUX_DECORATOR_CLOSE |
04868                 LINUX_DECORATOR_MAXIMIZE |
      LINUX_DECORATOR_MINIMIZE;
04869             long Hints[5] = { LINUX_FUNCTION | LINUX_DECORATOR, window->
      currentWindowStyle, window->decorators, 0, 0 };
04870 
04871             XChangeProperty( GetDisplay(), window->windowHandle, window->AtomHints, XA_ATOM, 32,
      PropModeReplace,
04872                 ( unsigned char* )Hints, 5 );
04873 
04874             XMapWindow( GetDisplay(), window->windowHandle );
04875             break;
04876         }
04877 
04878         case WINDOWSTYLE_BARE:
04879         {
04880             window->decorators = ( 1L << 2 );
04881             window->currentWindowStyle = ( 1L << 2 );
04882             long Hints[5] = { LINUX_FUNCTION | LINUX_DECORATOR, window->
      currentWindowStyle, window->decorators, 0, 0 };
04883 
04884             XChangeProperty( GetDisplay(), window->windowHandle, window->AtomHints, XA_ATOM, 32,
      PropModeReplace,
04885                 ( unsigned char* )Hints, 5 );
04886 
04887             XMapWindow( GetDisplay(), window->windowHandle );
04888             break;
04889         }
04890 
04891         case WINDOWSTYLE_POPUP:
04892         {
04893             window->decorators = 0;
04894             window->currentWindowStyle = ( 1L << 2 );
04895             long Hints[5] = { LINUX_FUNCTION | LINUX_DECORATOR, window->
      currentWindowStyle, window->decorators, 0, 0 };
```

```
04896
04897             XChangeProperty( GetDisplay(), window->windowHandle, window->AtomHints, XA_ATOM, 32,
    PropModeReplace,
04898                  ( unsigned char* )Hints, 5 );
04899
04900             XMapWindow( GetDisplay(), window->windowHandle );
04901             break;
04902         }
04903
04904         default:
04905         {
04906             PrintErrorMessage(
    TINYWINDOW_ERROR_INVALID_WINDOWSTYLE );
04907             break;
04908         }
04909         }
04910     }
```

7.2.3.79 static void windowManager::Linux_Shutdown (void) [inline], [static], [private]

```
03720     {
03721         XCloseDisplay( GetInstance()->currentDisplay );
03722     }
```

7.2.3.80 static void windowManager::Linux_ShutdownWindow (tWindow * *window*) [inline], [static], [private]

< the window is currently full screen

```
03706     {
03707         if( window->currentState == WINDOWSTATE_FULLSCREEN )
03708         {
03709             RestoreWindowByName( window->name );
03710         }
03711
03712         glXDestroyContext( windowManager::GetDisplay(), window->context );
03713         XUnmapWindow( windowManager::GetDisplay(), window->windowHandle );
03714         XDestroyWindow( windowManager::GetDisplay(), window->windowHandle );
03715         window->windowHandle = 0;
03716         window->context = 0;
03717     }
```

7.2.3.81 static GLuint windowManager::Linux_TranslateKey (GLuint *keySymbol*) [inline], [static], [private]

< the fist key that is not a char

< the Escape key

< the fist key that is not a char

< the Home key

< the fist key that is not a char

< the ArrowLeft key

< the fist key that is not a char

< the ArrowRight key

< the fist key that is not a char

< the ArrowUp key

< the fist key that is not a char

< the ArrowDown key

< the fist key that is not a char

< the PageUp key

< the fist key that is not a char

< the PageDown key

< the fist key that is not a char

< the End key

< the fist key that is not a char

< the PrintScreen key

< the fist key that is not a char

< the insert key

< the fist key that is not a char

< the NumLock key

< the fist key that is not a char

< the Keypad Multiply key

< the fist key that is not a char

< the Keypad Add key

< the fist key that is not a char

< the Keypad Subtract key

< the fist key that is not a char

< the Keypad Period/Decimal key

< the fist key that is not a char

< the KeyPad Divide key

< the fist key that is not a char

< the Keypad 0 key

< the fist key that is not a char

< the Keypad 1 key

< the fist key that is not a char

< the Keypad 2 key

< the fist key that is not a char

< the Keypad 3 key

< the fist key that is not a char

< the Keypad 4 key

< the fist key that is not a char

< the Keypad 5 key

< the fist key that is not a char

< the Keypad 6 key

< the fist key that is not a char

< the Keypad 7 key

< the fist key that is not a char

< the keypad 8 key

< the fist key that is not a char

< the Keypad 9 key

< the fist key that is not a char

< the F1 key

< the fist key that is not a char

< the F2 key

< the fist key that is not a char

< the F3 key

< the fist key that is not a char

< the F4 key

< the fist key that is not a char

< the F5 key

< the fist key that is not a char

< the F6 key

< the fist key that is not a char

< the F7 key

< the fist key that is not a char

< the F8 key

< the fist key that is not a char

< the F9 key

< the fist key that is not a char

< the F10 key

< the fist key that is not a char

< the F11 key

< the fist key that is not a char

< the F12 key

< the fist key that is not a char

< the left Shift key

< the fist key that is not a char

< the right Shift key

< the fist key that is not a char

< the right Control key

< the fist key that is not a char

< the left Control key

< the fist key that is not a char

< the CapsLock key

< the fist key that is not a char

< the left Alternate key

< the fist key that is not a char

< the right Alternate key

```
04485     {
```

```
04486         switch ( keySymbol )
04487         {
04488         case XK_Escape:
04489         {
04490             return KEY_ESCAPE;
04491         }
04492
04493         case XK_Home:
04494         {
04495             return KEY_HOME;
04496         }
04497
04498         case XK_Left:
04499         {
04500             return KEY_ARROW_LEFT;
04501         }
04502
04503         case XK_Right:
04504         {
04505             return KEY_ARROW_RIGHT;
04506         }
04507
04508         case XK_Up:
04509         {
04510             return KEY_ARROW_UP;
04511         }
04512
04513         case XK_Down:
04514         {
04515             return KEY_ARROW_DOWN;
04516         }
04517
04518         case XK_Page_Up:
04519         {
04520             return KEY_PAGEUP;
04521         }
04522
04523         case XK_Page_Down:
04524         {
04525             return KEY_PAGEDOWN;
04526         }
04527
04528         case XK_End:
04529         {
04530             return KEY_END;
04531         }
04532
04533         case XK_Print:
04534         {
04535             return KEY_PRINTSCREEN;
04536         }
04537
04538         case XK_Insert:
04539         {
04540             return KEY_INSERT;
04541         }
04542
04543         case XK_Num_Lock:
04544         {
04545             return KEY_NUMLOCK;
04546         }
04547
04548         case XK_KP_Multiply:
04549         {
04550             return KEY_KEYPAD_MULTIPLY;
04551         }
04552
04553         case XK_KP_Add:
04554         {
04555             return KEY_KEYPAD_ADD;
04556         }
04557
04558         case XK_KP_Subtract:
04559         {
04560             return KEY_KEYPAD_SUBTRACT;
04561         }
04562
04563         case XK_KP_Decimal:
04564         {
04565             return KEY_KEYPAD_PERIOD;
04566         }
04567
04568         case XK_KP_Divide:
04569         {
04570             return KEY_KEYPAD_DIVIDE;
04571         }
04572
```

```
04573         case XK_KP_0:
04574         {
04575             return KEY_KEYPAD_0;
04576         }
04577
04578         case XK_KP_1:
04579         {
04580             return KEY_KEYPAD_1;
04581         }
04582
04583         case XK_KP_2:
04584         {
04585             return KEY_KEYPAD_2;
04586         }
04587
04588         case XK_KP_3:
04589         {
04590             return KEY_KEYPAD_3;
04591         }
04592
04593         case XK_KP_4:
04594         {
04595             return KEY_KEYPAD_4;
04596         }
04597
04598         case XK_KP_5:
04599         {
04600             return KEY_KEYPAD_5;
04601         }
04602
04603         case XK_KP_6:
04604         {
04605             return KEY_KEYPAD_6;
04606         }
04607
04608         case XK_KP_7:
04609         {
04610             return KEY_KEYPAD_7;
04611         }
04612
04613         case XK_KP_8:
04614         {
04615             return KEY_KEYPAD_8;
04616         }
04617
04618         case XK_KP_9:
04619         {
04620             return KEY_KEYPAD_9;
04621         }
04622
04623         case XK_F1:
04624         {
04625             return KEY_F1;
04626         }
04627
04628         case XK_F2:
04629         {
04630             return KEY_F2;
04631         }
04632
04633         case XK_F3:
04634         {
04635             return KEY_F3;
04636         }
04637
04638         case XK_F4:
04639         {
04640             return KEY_F4;
04641         }
04642
04643         case XK_F5:
04644         {
04645             return KEY_F5;
04646         }
04647
04648         case XK_F6:
04649         {
04650             return KEY_F6;
04651         }
04652
04653         case XK_F7:
04654         {
04655             return KEY_F7;
04656         }
04657
04658         case XK_F8:
04659         {
```

```
04660             return KEY_F8;
04661         }
04662
04663         case XK_F9:
04664         {
04665             return KEY_F9;
04666         }
04667
04668         case XK_F10:
04669         {
04670             return KEY_F10;
04671         }
04672
04673         case XK_F11:
04674         {
04675             return KEY_F11;
04676         }
04677
04678         case XK_F12:
04679         {
04680             return KEY_F12;
04681         }
04682
04683         case XK_Shift_L:
04684         {
04685             return KEY_LEFTSHIFT;
04686         }
04687
04688         case XK_Shift_R:
04689         {
04690             return KEY_RIGHTSHIFT;
04691         }
04692
04693         case XK_Control_R:
04694         {
04695             return KEY_RIGHTCONTROL;
04696         }
04697
04698         case XK_Control_L:
04699         {
04700             return KEY_LEFTCONTROL;
04701         }
04702
04703         case XK_Caps_Lock:
04704         {
04705             return KEY_CAPSLOCK;
04706         }
04707
04708         case XK_Alt_L:
04709         {
04710             return KEY_LEFTALT;
04711         }
04712
04713         case XK_Alt_R:
04714         {
04715             return KEY_RIGHTALT;
04716         }
04717
04718         default:
04719         {
04720             return 0;
04721         }
04722         }
04723     }
```

7.2.3.82 static void windowManager::Linux_WaitForEvents (void) [inline], [static], [private]

```
04297     {
04298         //even if there aren't any events to process
04299         XNextEvent( GetInstance()->currentDisplay, &
      GetInstance()->currentEvent );
04300
04301         XEvent currentEvent = GetInstance()->currentEvent;
04302
04303         Linux_ProcessEvents( currentEvent );
04304     }
```

7.2.3.83 static GLboolean windowManager::MakeWindowCurrentContextByIndex (GLuint *windowIndex*) [inline], [static]

make the given window be the current OpenGL Context to be drawn to < if a window tries to use a graphical function without a context

```
01045     {
01046         if ( GetInstance()->IsInitialized() )
01047         {
01048             if ( DoesExistByIndex( windowIndex ) )
01049             {
01050 #if defined( _WIN32 ) || defined( _WIN64 )
01051                 wglMakeCurrent( GetWindowByIndex( windowIndex )->deviceContextHandle,
01052                     GetWindowByIndex( windowIndex )->glRenderingContextHandle );
01053 #else
01054                 glXMakeCurrent( GetDisplay(), GetWindowByIndex( windowIndex )->
      windowHandle,
01055                     GetWindowByIndex( windowIndex )->context );
01056 #endif
01057                 return FOUNDATION_OK;
01058             }
01059 
01060             return FOUNDATION_ERROR;
01061         }
01062 
01063         PrintErrorMessage( TINYWINDOW_ERROR_NO_CONTEXT );
01064         return FOUNDATION_ERROR;
01065     }
```

7.2.3.84 static GLboolean windowManager::MakeWindowCurrentContextByName (const char * *windowName*) [inline], [static]

make the given window be the current OpenGL Context to be drawn to < if the window is being used without being initialized

```
01021     {
01022         if ( GetInstance()->IsInitialized() )
01023         {
01024             if ( DoesExistByName( windowName ) )
01025             {
01026 
01027 #if defined( _WIN32 ) || defined( _WIN64 )
01028                 wglMakeCurrent( GetWindowByName( windowName )->deviceContextHandle,
01029                     GetWindowByName( windowName )->glRenderingContextHandle );
01030 #else
01031                 glXMakeCurrent( windowManager::GetDisplay(),
      GetWindowByName( windowName )->windowHandle,
01032                     GetWindowByName( windowName )->context );
01033 #endif
01034                 return FOUNDATION_OK;
01035             }
01036             return FOUNDATION_ERROR;
01037         }
01038         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
       );
01039         return FOUNDATION_ERROR;
01040     }
```

7.2.3.85 static GLboolean windowManager::MaximizeWindowByIndex (GLuint *windowIndex*, GLboolean *newState*) [inline], [static]

toggle the maximization state of the current window < if the window is being used without being initialized

```
01357     {
01358         if ( GetInstance()->IsInitialized() )
01359         {
01360             if ( DoesExistByIndex( windowIndex ) )
01361             {
01362 #if defined( _WIN32 ) || defined( _WIN64 )
01363                 Windows_Maximize( GetWindowByIndex( windowIndex ), newState );
01364 #else
01365                 Linux_Maximize( GetWindowByIndex( windowIndex ) );
```

```
01366 #endif
01367                 return FOUNDATION_OK;
01368             }
01369             return FOUNDATION_ERROR;
01370         }
01371         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01372         return FOUNDATION_ERROR;
01373     }
```

7.2.3.86 static GLboolean windowManager::MaximizeWindowByName (const char * *windowName,* GLboolean *newState*) [inline], [static]

toggle the maximization state of the current window < the window is currently maximized

< the window is in its default state

< if the window is being used without being initialized

```
01321     {
01322         if ( GetInstance()->IsInitialized() )
01323         {
01324             if ( DoesExistByName( windowName ) )
01325             {
01326                 if ( newState )
01327                 {
01328                     GetWindowByName( windowName )->currentState =
      WINDOWSTATE_MAXIMIZED;
01329 #if defined( _WIN32 ) || defined( _WIN64 )
01330                     Windows_Maximize( GetWindowByName( windowName ), newState );
01331 #else
01332                     Linux_Maximize( GetWindowByName( windowName ) );
01333 #endif
01334                     return FOUNDATION_OK;
01335                 }
01336
01337                 else
01338                 {
01339                     GetWindowByName( windowName )->currentState =
      WINDOWSTATE_NORMAL;
01340 #if defined( _WIN32 ) || defined( _WIN64 )
01341                     Windows_Maximize( GetWindowByName( windowName ), newState );
01342 #else
01343                     Linux_Maximize( GetWindowByName( windowName ) );
01344 #endif
01345                     return FOUNDATION_OK;
01346                 }
01347             }
01348             return FOUNDATION_ERROR;
01349         }
01350         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01351         return FOUNDATION_ERROR;
01352     }
```

7.2.3.87 static GLboolean windowManager::MinimizeWindowByIndex (GLuint *windowIndex,* GLboolean *newState*) [inline], [static]

toggle the minimization state of the window < the window is currently minimized

< the window is in its default state

< if the window is being used without being initialized

```
01250     {
01251         if ( GetInstance()->IsInitialized() )
01252         {
01253             if ( DoesExistByIndex( windowIndex ) )
01254             {
01255                 if ( newState )
01256                 {
01257                     GetWindowByIndex( windowIndex )->
      currentState = WINDOWSTATE_MINIMIZED;
01258 #if defined( _WIN32 ) || defined( _WIN64 )
01259                     Windows_Minimize( GetWindowByIndex( windowIndex ), newState );
```

```
01260 #else
01261                     Linux_Minimize( GetWindowByIndex( windowIndex ) );
01262 #endif
01263                     return FOUNDATION_OK;
01264                 }
01265
01266                 else
01267                 {
01268                     GetWindowByIndex( windowIndex )->
      currentState = WINDOWSTATE_NORMAL;
01269 #if defined( _WIN32 ) || defined( _WIN64 )
01270                     Windows_Minimize( GetWindowByIndex( windowIndex ), newState );
01271 #else
01272                     Linux_Minimize( GetWindowByIndex( windowIndex ) );
01273 #endif
01274                     return FOUNDATION_OK;
01275                 }
01276             }
01277             return FOUNDATION_ERROR;
01278         }
01279         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01280         return FOUNDATION_ERROR;
01281     }
```

7.2.3.88 static GLboolean windowManager::MinimizeWindowByName (const char * *windowName*, GLboolean *newState*) [inline], [static]

toggle the minimization state of the given window < the window is currently minimized

< the window is in its default state

< if a window tries to use a graphical function without a context

```
01213     {
01214         if ( GetInstance()->IsInitialized() )
01215         {
01216             if ( DoesExistByName( windowName ) )
01217             {
01218                 if ( newState )
01219                 {
01220                     GetWindowByName( windowName )->currentState =
      WINDOWSTATE_MINIMIZED;
01221
01222 #if defined( _WIN32 ) || defined( _WIN64 )
01223                     Windows_Minimize( GetWindowByName( windowName ), newState );
01224 #else
01225                     Linux_Minimize( GetWindowByName( windowName ) );
01226 #endif
01227                     return FOUNDATION_OK;
01228                 }
01229
01230                 else
01231                 {
01232                     GetWindowByName( windowName )->currentState =
      WINDOWSTATE_NORMAL;
01233 #if defined( _WIN32 ) || defined( _WIN64 )
01234                     Windows_Minimize( GetWindowByName( windowName ), newState );
01235 #else
01236                     Linux_Minimize( GetWindowByName( windowName ) );
01237 #endif
01238                     return FOUNDATION_OK;
01239                 }
01240             }
01241             return FOUNDATION_ERROR;
01242         }
01243         PrintErrorMessage( TINYWINDOW_ERROR_NO_CONTEXT );
01244         return FOUNDATION_ERROR;
01245     }
```

7.2.3.89 static void windowManager::PollForEvents (void) [inline], [static]

< if the window is being used without being initialized

```
01604     {
01605         if ( GetInstance()->IsInitialized() )
```

```
01606         {
01607 #if defined( _WIN32 ) || defined( _WIN64 )
01608             GetInstance()->Windows_PollForEvents();
01609 #else
01610             GetInstance()->Linux_PollForEvents();
01611 #endif
01612         }
01613 
01614         else
01615         {
01616             PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01617         }
01618     }
```

7.2.3.90 static GLboolean windowManager::RemoveWindowByIndex (GLuint *windowIndex*) [inline], [static]

< if the window is being used without being initialized

```
01652     {
01653         if ( GetInstance()->IsInitialized() )
01654         {
01655             if ( DoesExistByIndex( windowIndex ) )
01656             {
01657                 ShutdownWindow( GetWindowByIndex( windowIndex ) );
01658                 return FOUNDATION_OK;
01659             }
01660             return FOUNDATION_ERROR;
01661         }
01662         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01663         return FOUNDATION_ERROR;
01664     }
```

7.2.3.91 static GLboolean windowManager::RemoveWindowByName (const char * *windowName*) [inline], [static]

< if the window is being used without being initialized

```
01638     {
01639         if ( GetInstance()->IsInitialized() )
01640         {
01641             if ( DoesExistByName( windowName ) )
01642             {
01643                 ShutdownWindow( GetWindowByName( windowName ) );
01644                 return FOUNDATION_OK;
01645             }
01646             return FOUNDATION_ERROR;
01647         }
01648         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
    );
01649         return FOUNDATION_ERROR;
01650     }
```

7.2.3.92 static GLboolean windowManager::RestoreWindowByIndex (GLuint *windowIndex*) [inline], [static]

< if a window tries to use a graphical function without a context

```
01568     {
01569         if ( GetInstance()->IsInitialized() )
01570         {
01571             if ( WindowExists( windowIndex ) )
01572             {
01573 #if defined( _WIN32 ) || defined( _WIN64 )
01574                 Windows_Restore( GetWindowByIndex( windowIndex ) );
01575 #else
01576                 Linux_Restore( GetWindowByIndex( windowIndex ) );
01577 #endif
01578                 return FOUNDATION_OK;
01579             }
01580             return FOUNDATION_ERROR;
```

```
01581         }
01582         PrintErrorMessage( TINYWINDOW_ERROR_NO_CONTEXT );
01583         return FOUNDATION_ERROR;
01584     }
```

7.2.3.93 static GLboolean windowManager::RestoreWindowByName (const char * *windowName*) [inline], [static]

< if the window is being used without being initialized

```
01550     {
01551         if ( GetInstance()->IsInitialized() )
01552         {
01553             if ( DoesExistByName( windowName ) )
01554             {
01555 #if defined( _WIN32 ) || defined( _WIN64 )
01556                 Windows_Restore( GetWindowByName( windowName ) );
01557 #else
01558                 Linux_Restore( GetWindowByName( windowName ) );
01559 #endif
01560                 return FOUNDATION_OK;
01561             }
01562             return FOUNDATION_ERROR;
01563         }
01564         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01565         return FOUNDATION_ERROR;
01566     }
```

7.2.3.94 static GLboolean windowManager::SetFullScreenByIndex (GLuint *windowIndex*, GLboolean *newState*) [inline], [static]

< the window is currently full screen

< the window is in its default state

< if the window is being used without being initialized

```
01143     {
01144         if ( GetInstance()->IsInitialized() )
01145         {
01146             if ( DoesExistByIndex( windowIndex ) )
01147             {
01148                 if ( newState )
01149                 {
01150                     GetWindowByIndex( windowIndex )->
      currentState = WINDOWSTATE_FULLSCREEN;
01151 #if defined( _WIN32 ) || defined( _WIN64 )
01152                     Windows_FullScreen( GetWindowByIndex( windowIndex ) );
01153 #else
01154                     Linux_Fullscreen( GetWindowByIndex( windowIndex ) );
01155 #endif
01156                     return FOUNDATION_OK;
01157                 }
01158
01159                 else
01160                 {
01161                     GetWindowByIndex( windowIndex )->
      currentState = WINDOWSTATE_NORMAL;
01162 #if defined( _WIN32 ) || defined( _WIN64 )
01163                     Windows_FullScreen( GetWindowByIndex( windowIndex ) );
01164 #else
01165                     Linux_Fullscreen( GetWindowByIndex( windowIndex ) );
01166 #endif
01167                     return FOUNDATION_OK;
01168                 }
01169             }
01170             return FOUNDATION_ERROR;
01171         }
01172         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01173         return FOUNDATION_ERROR;
01174     }
```

7.2.3.95 static GLboolean windowManager::SetFullScreenByName (const char * *windowName,* GLboolean *newState*) [inline], [static]

toggle the given window's full screen mode < the window is currently full screen

< the window is in its default state

< if the window is being used without being initialized

```
01106     {
01107         if ( GetInstance()->IsInitialized() )
01108         {
01109             if ( DoesExistByName( windowName ) )
01110             {
01111                 if ( newState )
01112                 {
01113                     GetWindowByName( windowName )->currentState =
      WINDOWSTATE_FULLSCREEN;
01114 #if defined( _WIN32 ) || defined( _WIN64 )
01115                     Windows_FullScreen( GetWindowByName( windowName ) );
01116 #else
01117                     Linux_Fullscreen( GetWindowByName( windowName ) );
01118 #endif
01119
01120                     return FOUNDATION_OK;
01121                 }
01122
01123                 else
01124                 {
01125                     GetWindowByName( windowName )->currentState =
      WINDOWSTATE_NORMAL;
01126 #if defined( _WIN32 ) || defined( _WIN64 )
01127                     Windows_FullScreen( GetWindowByName( windowName ) );
01128 #else
01129                     Linux_Fullscreen( GetWindowByName( windowName ) );
01130 #endif
01131                     return FOUNDATION_OK;
01132                 }
01133             }
01134             return FOUNDATION_ERROR;
01135         }
01136         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01137         return FOUNDATION_ERROR;
01138     }
```

7.2.3.96 static GLboolean windowManager::SetMousePositionInScreen (GLuint *x,* GLuint *y*) [inline], [static]

set the position of the mouse cursor relative to screen co-ordinates < if the window is being used without being initialized

```
00471     {
00472         if ( GetInstance()->IsInitialized() )
00473         {
00474             GetInstance()->screenMousePosition[0] = x;
00475             GetInstance()->screenMousePosition[1] = y;
00476
00477 #if defined( _WIN32 ) || defined( _WIN64 )
00478             Windows_SetMousePositionInScreen();
00479 #else
00480             Linux_SetMousePositionInScreen( x, y );
00481 #endif
00482             return FOUNDATION_OK;
00483         }
00484         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00485         return FOUNDATION_ERROR;
00486     }
```

7.2.3.97 static GLboolean windowManager::SetMousePositionInWindowByIndex (GLuint *windowIndex,* GLuint *x,* GLuint *y*) [inline], [static]

set the mouse Position of the given window's co-ordinates < if the window is being used without being initialized

```
00884     {
00885         if ( GetInstance()->IsInitialized() )
00886         {
00887             if ( DoesExistByIndex( windowIndex ) )
00888             {
00889                 GetWindowByIndex( windowIndex )->mousePosition[0] = x;
00890                 GetWindowByIndex( windowIndex )->mousePosition[1] = y;
00891 #if defined( _WIN32 ) || defined( _WIN64 )
00892                 Windows_SetMousePosition( GetWindowByIndex( windowIndex ) );
00893 #else
00894                 Linux_SetMousePosition( GetWindowByIndex( windowIndex
       ) );
00895 #endif
00896                 return FOUNDATION_OK;
00897             }
00898             return FOUNDATION_ERROR;
00899         }
00900         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
       );
00901         return FOUNDATION_ERROR;
00902     }
```

7.2.3.98 static GLboolean windowManager::SetMousePositionInWindowByName (const char * *windowName*, GLuint *x*, GLuint *y*) [inline], [static]

set the mouse Position of the given window's co-ordinates < if the window is being used without being initialized

```
00860     {
00861         if ( GetInstance()->IsInitialized() )
00862         {
00863             if ( DoesExistByName( windowName ) )
00864             {
00865                 GetWindowByName( windowName )->mousePosition[0] = x;
00866                 GetWindowByName( windowName )->mousePosition[1] = y;
00867 #if defined( _WIN32 ) || defined( _WIN64 )
00868                 Windows_SetMousePosition( GetWindowByName( windowName ) );
00869 #else
00870                 Linux_SetMousePosition( GetWindowByName( windowName )
      );
00871 #endif
00872                 return FOUNDATION_OK;
00873             }
00874             return FOUNDATION_ERROR;
00875         }
00876
00877         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
       );
00878         return FOUNDATION_ERROR;
00879     }
```

7.2.3.99 static GLboolean windowManager::SetWindowIconByIndex (GLuint *windowIndex*, const char * *icon*, GLuint *width*, GLuint *height*) [inline], [static]

< if the window is being used without being initialized

```
01464     {
01465         if ( GetInstance()->IsInitialized() )
01466         {
01467             if ( DoesExistByIndex( windowIndex ) && IsValid( icon ) )
01468             {
01469 #if defined( _WIN32 ) || defined( _WIN64 )
01470                 Windows_SetWindowIcon( GetWindowByIndex( windowIndex ), icon, width, height
       );
01471 #else
01472                 Linux_SetWindowIcon( GetWindowByIndex( windowIndex ),
      icon, width, height );
01473 #endif
01474                 return FOUNDATION_OK;
01475             }
01476
01477             return FOUNDATION_ERROR;
01478         }
01479         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
       );
01480         return FOUNDATION_ERROR;
01481     }
```

7.2.3.100 static GLboolean windowManager::SetWindowIconByName (const char * *windowName,* const char * *icon,* GLuint *width,* GLuint *height*) [inline], [static]

< if the window is being used without being initialized

```
01445     {
01446         if ( GetInstance()->IsInitialized() )
01447         {
01448             if ( DoesExistByName( windowName ) && IsValid( icon ) )
01449             {
01450 #if defined( _WIN32 ) || defined( _WIN64 )
01451                 Windows_SetWindowIcon( GetWindowByName( windowName ), icon, width, height );
01452 #else
01453                 Linux_SetWindowIcon( GetWindowByName( windowName ), icon,
      width, height );
01454 #endif
01455                 return FOUNDATION_OK;
01456             }
01457             return FOUNDATION_ERROR;
01458         }
01459 
01460         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01461         return FOUNDATION_ERROR;
01462     }
```

7.2.3.101 static GLboolean windowManager::SetWindowOnDestroyedByIndex (GLuint *windowIndex,* onDestroyedEvent_t *onDestroyed*) [inline], [static]

< if the window is being used without being initialized

```
01881     {
01882         if ( GetInstance()->IsInitialized() )
01883         {
01884             if ( DoesExistByIndex( windowIndex ) )
01885             {
01886                 GetWindowByIndex( windowIndex )->destroyedEvent = onDestroyed
    ;
01887                 return FOUNDATION_OK;
01888             }
01889             return FOUNDATION_ERROR;
01890         }
01891         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01892         return FOUNDATION_ERROR;
01893     }
```

7.2.3.102 static GLboolean windowManager::SetWindowOnDestroyedByName (const char * *windowName,* onDestroyedEvent_t *onDestroyed*) [inline], [static]

< if the window is being used without being initialized

```
01867     {
01868         if ( GetInstance()->IsInitialized() )
01869         {
01870             if ( DoesExistByName( windowName ) )
01871             {
01872                 GetWindowByName( windowName )->destroyedEvent = onDestroyed;
01873                 return FOUNDATION_OK;
01874             }
01875             return FOUNDATION_ERROR;
01876         }
01877         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01878         return FOUNDATION_ERROR;
01879     }
```

7.2.3.103 static GLboolean windowManager::SetWindowOnFocusByIndex (GLuint *windowIndex,* onFocusEvent_t *onFocus*) [inline], [static]

< if the window is being used without being initialized

```
01968     {
01969         if ( GetInstance()->IsInitialized() )
01970         {
01971             if ( DoesExistByIndex( windowIndex ) )
01972             {
01973                 GetWindowByIndex( windowIndex )->focusEvent = onFocus;
01974                 return FOUNDATION_OK;
01975             }
01976             return FOUNDATION_ERROR;
01977         }
01978         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01979         return FOUNDATION_ERROR;
01980     }
```

7.2.3.104 static GLboolean windowManager::SetWindowOnFocusByName (const char * *windowName,* onFocusEvent_t *onFocus*) [inline],[static]

< if the window is being used without being initialized

```
01954     {
01955         if ( GetInstance()->IsInitialized() )
01956         {
01957             if ( DoesExistByName( windowName ) )
01958             {
01959                 GetWindowByName( windowName )->focusEvent = onFocus;
01960                 return FOUNDATION_OK;
01961             }
01962             return FOUNDATION_ERROR;
01963         }
01964         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
    );
01965         return FOUNDATION_ERROR;
01966     }
```

7.2.3.105 static GLboolean windowManager::SetWindowOnKeyEventByIndex (GLuint *windowIndex,* onKeyEvent_t *onKey*) [inline],[static]

< if the window is being used without being initialized

```
01794     {
01795         if ( GetInstance()->IsInitialized() )
01796         {
01797             if ( DoesExistByIndex( windowIndex ) )
01798             {
01799                 GetWindowByIndex( windowIndex )->keyEvent = onKey;
01800                 return FOUNDATION_OK;
01801             }
01802             return FOUNDATION_ERROR;
01803         }
01804         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
    );
01805         return FOUNDATION_ERROR;
01806     }
```

7.2.3.106 static GLboolean windowManager::SetWindowOnKeyEventByName (const char * *windowName,* onKeyEvent_t *onKey*) [inline],[static]

< if the window is being used without being initialized

```
01779     {
01780         if ( GetInstance()->IsInitialized() )
01781         {
01782             if ( DoesExistByName( windowName ) )
01783             {
01784                 GetWindowByName( windowName )->keyEvent = onKey;
01785                 return FOUNDATION_OK;
01786             }
01787
01788             return FOUNDATION_ERROR;
```

```
01789         }
01790         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01791         return FOUNDATION_ERROR;
01792     }
```

7.2.3.107 static GLboolean windowManager::SetWindowOnMaximizedByIndex (GLuint *windowIndex,* onMaximizedEvent_t *onMaximized*) [inline], [static]

< if the window is being used without being initialized

```
01910     {
01911         if ( GetInstance()->IsInitialized() )
01912         {
01913             if ( DoesExistByIndex( windowIndex ) )
01914             {
01915                 GetWindowByIndex( windowIndex )->maximizedEvent = onMaximized
      ;
01916                     return FOUNDATION_OK;
01917             }
01918             return FOUNDATION_ERROR;
01919         }
01920         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01921         return FOUNDATION_ERROR;
01922     }
```

7.2.3.108 static GLboolean windowManager::SetWindowOnMaximizedByName (const char * *windowName,* onMaximizedEvent_t *onMaximized*) [inline], [static]

< if the window is being used without being initialized

```
01896     {
01897         if ( GetInstance()->IsInitialized() )
01898         {
01899             if ( DoesExistByName( windowName ) )
01900             {
01901                 GetWindowByName( windowName )->maximizedEvent = onMaximized;
01902                 return FOUNDATION_OK;
01903             }
01904             return FOUNDATION_ERROR;
01905         }
01906         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01907         return FOUNDATION_ERROR;
01908     }
```

7.2.3.109 static GLboolean windowManager::SetWindowOnMinimizedByIndex (GLuint *windowIndex,* onMinimizedEvent_t *onMinimized*) [inline], [static]

< if the window is being used without being initialized

```
01939     {
01940         if ( GetInstance()->IsInitialized() )
01941         {
01942             if ( DoesExistByIndex( windowIndex ) )
01943             {
01944                 GetWindowByIndex( windowIndex )->minimizedEvent = onMinimized
      ;
01945                 return FOUNDATION_OK;
01946             }
01947             return FOUNDATION_ERROR;
01948         }
01949         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01950         return FOUNDATION_ERROR;
01951     }
```

7.2.3.110 static GLboolean windowManager::SetWindowOnMinimizedByName (const char * *windowName*, onMinimizedEvent_t *onMinimized*) [inline], [static]

< if the window is being used without being initialized

```
01925     {
01926         if ( GetInstance()->IsInitialized() )
01927         {
01928             if ( DoesExistByName( windowName ) )
01929             {
01930                 GetWindowByName( windowName )->minimizedEvent = onMinimized;
01931                 return FOUNDATION_OK;
01932             }
01933             return FOUNDATION_ERROR;
01934         }
01935         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01936         return FOUNDATION_ERROR;
01937     }
```

7.2.3.111 static GLboolean windowManager::SetWindowOnMouseButtonEventByIndex (GLuint *windowIndex*, onMouseButtonEvent_t *onMouseButton*) [inline], [static]

< if the window is being used without being initialized

```
01823     {
01824         if ( GetInstance()->IsInitialized() )
01825         {
01826             if ( DoesExistByIndex( windowIndex ) )
01827             {
01828                 GetWindowByIndex( windowIndex )->
     mouseButtonEvent = onMouseButton;
01829                 return FOUNDATION_OK;
01830             }
01831             return FOUNDATION_ERROR;
01832         }
01833         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01834         return FOUNDATION_ERROR;
01835     }
```

7.2.3.112 static GLboolean windowManager::SetWindowOnMouseButtonEventByName (const char * *windowName*, onMouseButtonEvent_t *onMouseButton*) [inline], [static]

< if the window is being used without being initialized

```
01809     {
01810         if ( GetInstance()->IsInitialized() )
01811         {
01812             if ( DoesExistByName( windowName ) )
01813             {
01814                 GetWindowByName( windowName )->mouseButtonEvent =
     onMouseButton;
01815                 return FOUNDATION_OK;
01816             }
01817             return FOUNDATION_ERROR;
01818         }
01819         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01820         return FOUNDATION_ERROR;
01821     }
```

7.2.3.113 static GLboolean windowManager::SetWindowOnMouseMoveByIndex (GLuint *windowIndex*, onMouseMoveEvent_t *onMouseMove*) [inline], [static]

< if the window is being used without being initialized

```
02055     {
02056         if ( GetInstance()->IsInitialized() )
02057         {
02058             if ( DoesExistByIndex( windowIndex ) )
02059             {
02060                 GetWindowByIndex( windowIndex )->mouseMoveEvent = onMouseMove
      ;
02061                 return FOUNDATION_OK;
02062             }
02063             return FOUNDATION_ERROR;
02064         }
02065         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
       );
02066         return FOUNDATION_ERROR;
02067     }
```

7.2.3.114 static GLboolean windowManager::SetWindowOnMouseMoveByName (const char * *windowName*, onMouseMoveEvent_t *onMouseMove*) [inline], [static]

< if the window is being used without being initialized

```
02041     {
02042         if ( GetInstance()->IsInitialized() )
02043         {
02044             if ( DoesExistByName( windowName ) )
02045             {
02046                 GetWindowByName( windowName )->mouseMoveEvent = onMouseMove;
02047                 return FOUNDATION_OK;
02048             }
02049             return FOUNDATION_ERROR;
02050         }
02051         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
       );
02052         return FOUNDATION_ERROR;
02053     }
```

7.2.3.115 static GLboolean windowManager::SetWindowOnMouseWheelEventByIndex (GLuint *windowIndex*, onMouseWheelEvent_t *onMouseWheel*) [inline], [static]

< if the window is being used without being initialized

```
01852     {
01853         if ( GetInstance()->IsInitialized() )
01854         {
01855             if ( DoesExistByIndex( windowIndex ) )
01856             {
01857                 GetWindowByIndex( windowIndex )->mouseWheelEvent =
      onMouseWheel;
01858                 return FOUNDATION_OK;
01859             }
01860             return FOUNDATION_ERROR;
01861         }
01862         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
       );
01863         return FOUNDATION_ERROR;
01864     }
```

7.2.3.116 static GLboolean windowManager::SetWindowOnMouseWheelEventByName (const char * *windowName*, onMouseWheelEvent_t *onMouseWheel*) [inline], [static]

< if the window is being used without being initialized

```
01838     {
01839         if ( GetInstance()->IsInitialized() )
01840         {
01841             if ( DoesExistByName( windowName ) )
01842             {
01843                 GetWindowByName( windowName )->mouseWheelEvent = onMouseWheel
      ;
01844                 return FOUNDATION_OK;
```

```
01845             }
01846             return FOUNDATION_ERROR;
01847         }
01848         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01849         return FOUNDATION_ERROR;
01850     }
```

7.2.3.117 static GLboolean windowManager::SetWindowOnMovedByIndex (GLuint *windowIndex,* onMovedEvent_t *onMoved*) [inline], [static]

< if the window is being used without being initialized

```
01997     {
01998         if ( GetInstance()->IsInitialized() )
01999         {
02000             if ( DoesExistByIndex( windowIndex ) )
02001             {
02002                 GetWindowByIndex( windowIndex )->movedEvent = onMoved;
02003                 return FOUNDATION_OK;
02004             }
02005             return FOUNDATION_ERROR;
02006         }
02007         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
02008         return FOUNDATION_ERROR;
02009     }
```

7.2.3.118 static GLboolean windowManager::SetWindowOnMovedByName (const char * *windowName,* onMovedEvent_t *onMoved*) [inline], [static]

< if the window is being used without being initialized

```
01983     {
01984         if ( GetInstance()->IsInitialized() )
01985         {
01986             if ( DoesExistByName( windowName ) )
01987             {
01988                 GetWindowByName( windowName )->movedEvent = onMoved;
01989                 return FOUNDATION_OK;
01990             }
01991             return FOUNDATION_ERROR;
01992         }
01993         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01994         return FOUNDATION_ERROR;
01995     }
```

7.2.3.119 static GLboolean windowManager::SetWindowOnResizeByIndex (GLuint *windowIndex,* onResizeEvent_t *onResize*) [inline], [static]

< if the window is being used without being initialized

```
02026     {
02027         if ( GetInstance()->IsInitialized() )
02028         {
02029             if ( DoesExistByIndex( windowIndex ) )
02030             {
02031                 GetWindowByIndex( windowIndex )->resizeEvent = onResize;
02032                 return FOUNDATION_OK;
02033             }
02034             return FOUNDATION_ERROR;
02035         }
02036         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
02037         return FOUNDATION_ERROR;
02038     }
```

7.2.3.120 static GLboolean windowManager::SetWindowOnResizeByName (const char * *windowName*, onResizeEvent_t *onResize*) [inline], [static]

< if the window is being used without being initialized

```
02012     {
02013         if ( GetInstance()->IsInitialized() )
02014         {
02015             if ( DoesExistByName( windowName ) )
02016             {
02017                 GetWindowByName( windowName )->resizeEvent = onResize;
02018                 return FOUNDATION_OK;
02019             }
02020             return FOUNDATION_ERROR;
02021         }
02022         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
02023         return FOUNDATION_ERROR;
02024     }
```

7.2.3.121 static GLboolean windowManager::SetWindowPositionByName (const char * *windowName*, GLuint *x*, GLuint *y*) [inline], [static]

set the Position of the given window relative to screen co-ordinates < if the window is being used without being initialized

```
00742     {
00743         if ( GetInstance()->IsInitialized() )
00744         {
00745             if ( DoesExistByName( windowName ) )
00746             {
00747                 GetWindowByName( windowName )->position[0] = x;
00748                 GetWindowByName( windowName )->position[1] = y;
00749 #if defined( _WIN32 ) || defined( _WIN64 )
00750                 Windows_SetWindowPosition( GetWindowByName( windowName ) );
00751 #else
00752                 Linux_SetWindowPosition( GetWindowByName( windowName
      ) );
00753 #endif
00754                 return FOUNDATION_OK;
00755             }
00756             return FOUNDATION_ERROR;
00757         }
00758
00759         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00760         return FOUNDATION_ERROR;
00761     }
```

7.2.3.122 static GLboolean windowManager::SetWindowPositionByName (GLuint *windowIndex*, GLuint *x*, GLuint *y*) [inline], [static]

set the position of the given window relative to screen co-ordinates < if the window is being used without being initialized

```
00766     {
00767         if ( GetInstance()->IsInitialized() )
00768         {
00769             if ( DoesExistByIndex( windowIndex ) )
00770             {
00771                 GetWindowByIndex( windowIndex )->position[0] = x;
00772                 GetWindowByIndex( windowIndex )->position[1] = y;
00773 #if defined( _WIN32 ) || defined( _WIN64 )
00774                 Windows_SetWindowPosition( GetWindowByIndex( windowIndex ) );
00775 #else
00776                 Linux_SetWindowPosition( GetWindowByIndex(
      windowIndex ) );
00777 #endif
00778                 return FOUNDATION_OK;
00779             }
00780         }
00781
```

```
00782         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00783         return FOUNDATION_ERROR;
00784     }
```

7.2.3.123 static GLboolean windowManager::SetWindowResolutionByIndex (GLuint *windowIndex*, GLuint *width*, GLuint *height*) [inline], [static]

set the Size/Resolution of the given window < if the window is being used without being initialized

```
00644     {
00645         if ( GetInstance()->IsInitialized() )
00646         {
00647             if ( WindowExists( windowIndex ) )
00648             {
00649                 GetWindowByIndex( windowIndex )->resolution[0] = width;
00650                 GetWindowByIndex( windowIndex )->resolution[1] = height;
00651
00652 #if defined( _WIN32 ) || defined( _WIN64 )
00653                 Windows_SetWindowResolution( GetWindowByIndex( windowIndex ) );
00654 #else
00655                 Linux_SetWindowResolution(
      GetWindowByIndex( windowIndex ) );
00656 #endif
00657                 return FOUNDATION_OK;
00658             }
00659             return FOUNDATION_ERROR;
00660         }
00661         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00662         return FOUNDATION_ERROR;
00663     }
```

7.2.3.124 static GLboolean windowManager::SetWindowResolutionByName (const char * *windowName*, GLuint *width*, GLuint *height*) [inline], [static]

set the Size/Resolution of the given window < if the OpenGL context for the window is invalid

```
00620     {
00621         if ( GetInstance()->IsInitialized() )
00622         {
00623             if ( DoesExistByName( windowName ) )
00624             {
00625                 GetWindowByName( windowName )->resolution[0] = width;
00626                 GetWindowByName( windowName )->resolution[1] = height;
00627 #if defined( _WIN32 ) || defined( _WIN64 )
00628                 Windows_SetWindowResolution( GetWindowByName( windowName ) );
00629 #else
00630                 Linux_SetWindowResolution(
      GetWindowByName( windowName ) );
00631 #endif
00632                 return FOUNDATION_OK;
00633             }
00634             return FOUNDATION_ERROR;
00635         }
00636
00637         PrintErrorMessage( TINYWINDOW_ERROR_INVALID_CONTEXT
      );
00638         return FOUNDATION_ERROR;
00639     }
```

7.2.3.125 static GLboolean windowManager::SetWindowStyleByIndex (GLuint *windowIndex*, GLuint *windowStyle*) [inline], [static]

< if the window is being used without being initialized

```
01685     {
01686         if ( GetInstance()->IsInitialized() )
01687         {
01688             if ( DoesExistByIndex( windowIndex ) )
```

```
01689             {
01690 #if defined( _WIN32 ) || defined( _WIN64 )
01691                 Windows_SetWindowStyle( GetWindowByIndex( windowIndex ), windowStyle );
01692 #else
01693                 Linux_SetWindowStyle( GetWindowByIndex( windowIndex ),
      windowStyle );
01694 #endif
01695                 return FOUNDATION_OK;
01696             }
01697             return FOUNDATION_ERROR;
01698         }
01699         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01700         return FOUNDATION_ERROR;
01701     }
```

7.2.3.126 static GLboolean windowManager::SetWindowStyleByName (const char * *windowName*, GLuint *windowStyle*) [inline], [static]

< if the window is being used without being initialized

```
01667     {
01668         if ( GetInstance()->IsInitialized() )
01669         {
01670             if ( DoesExistByName( windowName ) )
01671             {
01672 #if defined( _WIN32 ) || defined( _WIN64 )
01673                 Windows_SetWindowStyle( GetWindowByName( windowName ), windowStyle );
01674 #else
01675                 Linux_SetWindowStyle( GetWindowByName( windowName ),
      windowStyle );
01676 #endif
01677                 return FOUNDATION_OK;
01678             }
01679             return FOUNDATION_ERROR;
01680         }
01681         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01682         return FOUNDATION_ERROR;
01683     }
```

7.2.3.127 static GLboolean windowManager::SetWindowTitleBarByIndex (GLuint *windowIndex*, const char * *newName*) [inline], [static]

< if the window is being used without being initialized

```
01425     {
01426         if ( GetInstance()->IsInitialized() )
01427         {
01428             if ( DoesExistByIndex( windowIndex ) && IsValid( newName ) )
01429             {
01430 #if defined( _WIN32 ) || defined( _WIN64 )
01431                 SetWindowText( GetWindowByIndex( windowIndex )->windowHandle, newName );
01432 #else
01433                 XStoreName( GetDisplay(), GetWindowByIndex( windowIndex )->
      windowHandle, newName );
01434 #endif
01435                 return FOUNDATION_OK;
01436             }
01437             return FOUNDATION_ERROR;
01438         }
01439
01440         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01441         return FOUNDATION_ERROR;
01442     }
```

7.2.3.128 static GLboolean windowManager::SetWindowTitleBarByName (const char * *windowName*, const char * *newTitle*) [inline], [static]

< if the window is being used without being initialized

```
01407     {
01408         if ( GetInstance()->IsInitialized() )
01409         {
01410             if ( DoesExistByName( windowName ) && IsValid( newTitle ) )
01411             {
01412 #if defined( _WIN32 ) || defined( _WIN64 )
01413                 SetWindowText( GetWindowByName( windowName )->windowHandle, newTitle );
01414 #else
01415                 XStoreName( GetDisplay(), GetWindowByName( windowName )->
      windowHandle, newTitle );
01416 #endif
01417                 return FOUNDATION_OK;
01418             }
01419             return FOUNDATION_ERROR;
01420         }
01421         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
       );
01422         return FOUNDATION_ERROR;
01423     }
```

7.2.3.129 static void windowManager::ShutDown (void) [inline], [static]

use this to shutdown the window manager when your program is finished

```
00371     {
00372 #if defined( _MSC_VER )
00373         for each ( auto CurrentWindow in GetInstance()->windowList )
00374         {
00375             delete CurrentWindow;
00376         }
00377 #endif
00378
00379 #if defined( CURRENT_OS_LINUX )
00380         for ( auto CurrentWindow : GetInstance()->windowList )
00381         {
00382             delete CurrentWindow;
00383         }
00384
00385         XCloseDisplay( GetInstance()->currentDisplay );
00386 #endif
00387
00388         GetInstance()->windowList.clear();
00389         delete instance;
00390     }
```

7.2.3.130 static void windowManager::ShutdownWindow (tWindow * *window*) [inline], [static], [private]

```
02318     {
02319 #if defined( _WIN32 ) || defined( _WIN64 )
02320         Windows_ShutdownWindow( window );
02321 #else
02322         Linux_ShutdownWindow( window );
02323 #endif
02324     }
```

7.2.3.131 static void windowManager::WaitForEvents (void) [inline], [static]

< if the window is being used without being initialized

```
01620     {
01621         if ( GetInstance()->IsInitialized() )
01622         {
01623 #if defined( _WIN32 ) || defined( _WIN64 )
01624             GetInstance()->Windows_WaitForEvents();
01625 #else
01626             GetInstance()->Linux_WaitForEvents();
01627 #endif
01628         }
01629
01630         else
01631         {
```

```
01632             PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
01633         }
01634     }
```

7.2.3.132 static GLboolean windowManager::WindowExists (GLuint *windowIndex*) [inline], [static], [private]

```
02280     {
02281         return ( windowIndex <= GetInstance()->windowList.size() - 1 );
02282     }
```

7.2.3.133 static GLboolean windowManager::WindowGetKeyByIndex (GLuint *windowIndex,* GLuint *key*) [inline], [static]

returns the current state of the given key relative to the given window < if the window is being used without being initialized

```
00925     {
00926         if ( GetInstance()->IsInitialized() )
00927         {
00928             if ( DoesExistByIndex( windowIndex ) )
00929             {
00930                 return GetWindowByIndex( windowIndex )->keys[key];
00931             }
00932             return FOUNDATION_ERROR;
00933         }
00934         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
      );
00935         return FOUNDATION_ERROR;
00936     }
```

7.2.3.134 static GLboolean windowManager::WindowGetKeyByName (const char * *windowName,* GLuint *key*) [inline], [static]

returns the current state of the given key relative to the given window < if the window is being used without being initialized

```
00908     {
00909         if ( GetInstance()->IsInitialized() )
00910         {
00911             if ( DoesExistByName( windowName ) )
00912             {
00913                 return GetWindowByName( windowName )->keys[key];
00914             }
00915
00916             return FOUNDATION_ERROR;
00917         }
00918         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
    );
00919         return FOUNDATION_ERROR;
00920     }
```

7.2.3.135 static GLboolean windowManager::WindowSwapBuffersByIndex (GLuint *windowIndex*) [inline], [static]

swap the draw buffers of the given window < if the window is being used without being initialized

```
00999     {
01000         if ( GetInstance()->IsInitialized() )
01001         {
01002             if ( DoesExistByIndex( windowIndex ) )
01003             {
01004 #if defined( _WIN32 ) || defined( _WIN64 )
01005                 SwapBuffers( GetWindowByIndex( windowIndex )->deviceContextHandle );
```

```
01006 #else
01007                 glXSwapBuffers( GetDisplay(), GetWindowByIndex( windowIndex )->
      windowHandle );
01008 #endif
01009                 return FOUNDATION_OK;
01010             }
01011             return FOUNDATION_ERROR;
01012         }
01013         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
       );
01014         return FOUNDATION_ERROR;
01015     }
```

7.2.3.136 static GLboolean windowManager::WindowSwapBuffersByName (const char * *windowName*) [inline], [static]

swap the draw buffers of the given window < if the window is being used without being initialized

```
00977     {
00978         if ( GetInstance()->IsInitialized() )
00979         {
00980             if ( DoesExistByName( windowName ) )
00981             {
00982 #if defined( _WIN32 ) || defined( _WIN64 )
00983                 SwapBuffers( GetWindowByName( windowName )->deviceContextHandle );
00984 #else
00985                 glXSwapBuffers( GetDisplay(), GetWindowByName( windowName )->
      windowHandle );
00986 #endif
00987                 return FOUNDATION_OK;
00988             }
00989             return FOUNDATION_ERROR;
00990         }
00991
00992         PrintErrorMessage( TINYWINDOW_ERROR_NOT_INITIALIZED
       );
00993         return FOUNDATION_ERROR;
00994     }
```

7.2.4 Field Documentation

7.2.4.1 const Display* windowManager::currentDisplay [private]

7.2.4.2 XEvent windowManager::currentEvent [private]

7.2.4.3 windowManager * windowManager::instance = nullptr [static], [private]

7.2.4.4 GLboolean windowManager::isInitialized [private]

7.2.4.5 GLuint windowManager::screenMousePosition[2] [private]

7.2.4.6 GLuint windowManager::screenResolution[2] [private]

7.2.4.7 std::list< tWindow*> windowManager::windowList [private]

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

- TinyWindow.h

Chapter 8

File Documentation

8.1 TinyWindow.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <list>
#include <limits.h>
#include <string.h>
```

Data Structures

- class windowManager
- struct windowManager::tWindow

Macros

- #define DEFAULT_WINDOW_WIDTH 1280
- #define DEFAULT_WINDOW_HEIGHT 720
- #define KEYSTATE_DOWN 1
- #define KEYSTATE_UP 0
- #define KEY_ERROR -1
- #define KEY_FIRST 256 + 1
- #define KEY_F1 KEY_FIRST
- #define KEY_F2 KEY_FIRST + 1
- #define KEY_F3 KEY_FIRST + 2
- #define KEY_F4 KEY_FIRST + 3
- #define KEY_F5 KEY_FIRST + 4
- #define KEY_F6 KEY_FIRST + 5
- #define KEY_F7 KEY_FIRST + 6
- #define KEY_F8 KEY_FIRST + 7
- #define KEY_F9 KEY_FIRST + 8
- #define KEY_F10 KEY_FIRST + 9
- #define KEY_F11 KEY_FIRST + 10
- #define KEY_F12 KEY_FIRST + 11
- #define KEY_CAPSLOCK KEY_FIRST + 12
- #define KEY_LEFTSHIFT KEY_FIRST + 13
- #define KEY_RIGHTSHIFT KEY_FIRST + 14
- #define KEY_LEFTCONTROL KEY_FIRST + 15

- #define KEY_RIGHTCONTROL KEY_FIRST + 16
- #define KEY_LEFTWINDOW KEY_FIRST + 17
- #define KEY_RIGHTWINDOW KEY_FIRST + 18
- #define KEY_LEFTALT KEY_FIRST + 19
- #define KEY_RIGHTALT KEY_FIRST + 20
- #define KEY_ENTER KEY_FIRST + 21
- #define KEY_PRINTSCREEN KEY_FIRST + 22
- #define KEY_SCROLLLOCK KEY_FIRST + 23
- #define KEY_NUMLOCK KEY_FIRST + 24
- #define KEY_PAUSE KEY_FIRST + 25
- #define KEY_INSERT KEY_FIRST + 26
- #define KEY_HOME KEY_FIRST + 27
- #define KEY_END KEY_FIRST + 28
- #define KEY_PAGEUP KEY_FIRST + 28
- #define KEY_PAGEDOWN KEY_FIRST + 30
- #define KEY_ARROW_DOWN KEY_FIRST + 31
- #define KEY_ARROW_UP KEY_FIRST + 32
- #define KEY_ARROW_LEFT KEY_FIRST + 33
- #define KEY_ARROW_RIGHT KEY_FIRST + 34
- #define KEY_KEYPAD_DIVIDE KEY_FIRST + 35
- #define KEY_KEYPAD_MULTIPLY KEY_FIRST + 36
- #define KEY_KEYPAD_SUBTRACT KEY_FIRST + 37
- #define KEY_KEYPAD_ADD KEY_FIRST + 38
- #define KEY_KEYPAD_ENTER KEY_FIRST + 39
- #define KEY_KEYPAD_PERIOD KEY_FIRST + 40
- #define KEY_KEYPAD_0 KEY_FIRST + 41
- #define KEY_KEYPAD_1 KEY_FIRST + 42
- #define KEY_KEYPAD_2 KEY_FIRST + 43
- #define KEY_KEYPAD_3 KEY_FIRST + 44
- #define KEY_KEYPAD_4 KEY_FIRST + 45
- #define KEY_KEYPAD_5 KEY_FIRST + 46
- #define KEY_KEYPAD_6 KEY_FIRST + 47
- #define KEY_KEYPAD_7 KEY_FIRST + 48
- #define KEY_KEYPAD_8 KEY_FIRST + 49
- #define KEY_KEYPAD_9 KEY_FIRST + 50
- #define KEY_BACKSPACE KEY_FIRST + 51
- #define KEY_TAB KEY_FIRST + 52
- #define KEY_DELETE KEY_FIRST + 53
- #define KEY_ESCAPE KEY_FIRST + 54
- #define KEY_LAST KEY_ESCAPE
- #define MOUSE_BUTTONUP 0
- #define MOUSE_BUTTONDOWN 1
- #define MOUSE_LEFTBUTTON 0
- #define MOUSE_RIGHTBUTTON 1
- #define MOUSE_MIDDLEBUTTON 2
- #define MOUSE_LAST MOUSE_MIDDLEBUTTON + 1
- #define MOUSE_SCROLL_DOWN 0
- #define MOUSE_SCROLL_UP 1
- #define WINDOWSTYLE_BARE 1
- #define WINDOWSTYLE_DEFAULT 2
- #define WINDOWSTYLE_POPUP 3
- #define WINDOWSTATE_NORMAL 0
- #define WINDOWSTATE_MAXIMIZED 1
- #define WINDOWSTATE_MINIMIZED 2
- #define WINDOWSTATE_FULLSCREEN 3

- #define DECORATOR_TITLEBAR 0x01
- #define DECORATOR_ICON 0x02
- #define DECORATOR_BORDER 0x04
- #define DECORATOR_MINIMIZEBUTTON 0x08
- #define DECORATOR_MAXIMIZEBUTTON 0x010
- #define DECORATOR_CLOSEBUTTON 0x20
- #define DECORATOR_SIZEABLEBORDER 0x40
- #define LINUX_DECORATOR_BORDER 1L << 1
- #define LINUX_DECORATOR_MOVE 1L << 2
- #define LINUX_DECORATOR_MINIMIZE 1L << 3
- #define LINUX_DECORATOR_MAXIMIZE 1L << 4
- #define LINUX_DECORATOR_CLOSE 1L << 5
- #define FOUNDATION_ERROR 0
- #define FOUNDATION_OK 1
- #define TINYWINDOW_ERROR_NO_CONTEXT 0
- #define TINYWINDOW_ERROR_INVALID_WINDOW_NAME 1
- #define TINYWINDOW_ERROR_INVALID_WINDOW_INDEX 2
- #define TINYWINDOW_ERROR_INVALID_WINDOW_STATE 3
- #define TINYWINDOW_ERROR_INVALID_RESOLUTION 4
- #define TINYWINDOW_ERROR_INVALID_CONTEXT 5
- #define TINYWINDOW_ERROR_EXISTING_CONTEXT 6
- #define TINYWINDOW_ERROR_NOT_INITIALIZED 7
- #define TINYWINDOW_ERROR_ALREADY_INITIALIZED 8
- #define TINYWINDOW_ERROR_INVALID_TITLEBAR 9
- #define TINYWINDOW_ERROR_INVALID_EVENT 10
- #define TINYWIDNOW_ERROR_WINDOW_NOT_FOUND 11
- #define TINYWINDOW_ERROR_INVALID_WINDOWSTYLE 12
- #define TINYWINDOW_ERROR_INVALID_WINDOW 13
- #define TINYWINDOW_ERROR_FUNCTION_NOT_IMPLEMENTED 14
- #define TINYWINDOW_ERROR_LINUX_CANNOT_CONNECT_X_SERVER 15
- #define TINYWINDOW_ERROR_LINUX_INVALID_VISUALINFO 16
- #define TINYWINDOW_ERROR_LINUX_CANNOT_CREATE_WINDOW 17
- #define TINYWINDOW_ERROR_LINUX_FUNCTION_NOT_IMPLEMENTED 18
- #define TINYWINDOW_ERROR_WINDOWS_CANNOT_CREATE_WINDOW 19
- #define TINYWINDOW_ERROR_WINDOWS_CANNOT_INITIALIZE 20
- #define TINYWINDOW_ERROR_WINDOWS_FUNCTION_NOT_IMPLEMENTED 21
- #define TINYWINDOW_WARNING_NOT_CURRENT_CONTEXT 0
- #define TINYWINDOW_WARNING_NO_GL_EXTENSIONS 1
- #define LINUX_FUNCTION 1
- #define LINUX_DECORATOR 2

Typedefs

- typedef void(* onKeyEvent_t)(GLuint key, GLboolean keyState)
- typedef void(* onMouseButtonEvent_t)(GLuint button, GLboolean buttonState)
- typedef void(* onMouseWheelEvent_t)(GLuint wheelDirection)
- typedef void(* onDestroyedEvent_t)(void)
- typedef void(* onMaximizedEvent_t)(void)
- typedef void(* onMinimizedEvent_t)(void)
- typedef void(* onFocusEvent_t)(GLboolean inFocus)
- typedef void(* onMovedEvent_t)(GLuint x, GLuint y)
- typedef void(* onResizeEvent_t)(GLuint width, GLuint height)
- typedef void(* onMouseMoveEvent_t)(GLuint windowX, GLuint windowY, GLuint screenX, GLuint screenY)

Functions

- static void PrintWarningMessage (GLuint warningNumber)
- static void PrintErrorMessage (GLuint errorNumber)

8.1.1 Macro Definition Documentation

8.1.1.1 #define DECORATOR_BORDER 0x04

the border decoration of the window

8.1.1.2 #define DECORATOR_CLOSEBUTTON 0x20

the close button decoration of the window

8.1.1.3 #define DECORATOR_ICON 0x02

the icon decoration of the window

8.1.1.4 #define DECORATOR_MAXIMIZEBUTTON 0x010

the maximize button decoration pf the window

8.1.1.5 #define DECORATOR_MINIMIZEBUTTON 0x08

the minimize button decoration of the window

8.1.1.6 #define DECORATOR_SIZEABLEBORDER 0x40

the sizable border decoration of the window

8.1.1.7 #define DECORATOR_TITLEBAR 0x01

The title bar decoration of the window

8.1.1.8 #define DEFAULT_WINDOW_HEIGHT 720

8.1.1.9 #define DEFAULT_WINDOW_WIDTH 1280

8.1.1.10 #define FOUNDATION_ERROR 0

8.1.1.11 #define FOUNDATION_OK 1

8.1.1.12 #define KEY_ARROW_DOWN KEY_FIRST + 31

the ArrowDown key

8.1.1.13 #define KEY_ARROW_LEFT KEY_FIRST + 33

the ArrowLeft key

8.1.1.14 #define KEY_ARROW_RIGHT KEY_FIRST + 34

the ArrowRight key

8.1.1.15 #define KEY_ARROW_UP KEY_FIRST + 32

the ArrowUp key

8.1.1.16 #define KEY_BACKSPACE KEY_FIRST + 51

the Backspace key

8.1.1.17 #define KEY_CAPSLOCK KEY_FIRST + 12

the CapsLock key

8.1.1.18 #define KEY_DELETE KEY_FIRST + 53

the Delete key

8.1.1.19 #define KEY_END KEY_FIRST + 28

the End key

8.1.1.20 #define KEY_ENTER KEY_FIRST + 21

the Enter/Return key

8.1.1.21 #define KEY_ERROR -1

the key pressed is considered invalid

8.1.1.22 #define KEY_ESCAPE KEY_FIRST + 54

the Escape key

8.1.1.23 #define KEY_F1 KEY_FIRST

the F1 key

8.1.1.24 #define KEY_F10 KEY_FIRST + 9

the F10 key

8.1.1.25 #define KEY_F11 KEY_FIRST + 10

the F11 key

8.1.1.26 #define KEY_F12 KEY_FIRST + 11

the F12 key

8.1.1.27 #define KEY_F2 KEY_FIRST + 1

the F2 key

8.1.1.28 #define KEY_F3 KEY_FIRST + 2

the F3 key

8.1.1.29 #define KEY_F4 KEY_FIRST + 3

the F4 key

8.1.1.30 #define KEY_F5 KEY_FIRST + 4

the F5 key

8.1.1.31 #define KEY_F6 KEY_FIRST + 5

the F6 key

8.1.1.32 #define KEY_F7 KEY_FIRST + 6

the F7 key

8.1.1.33 #define KEY_F8 KEY_FIRST + 7

the F8 key

8.1.1.34 #define KEY_F9 KEY_FIRST + 8

the F9 key

8.1.1.35 #define KEY_FIRST 256 + 1

the fist key that is not a char

8.1.1.36 #define KEY_HOME KEY_FIRST + 27

the Home key

8.1.1.37 #define KEY_INSERT KEY_FIRST + 26

the insert key

8.1.1.38 #define KEY_KEYPAD_0 KEY_FIRST + 41

the Keypad 0 key

8.1.1.39 #define KEY_KEYPAD_1 KEY_FIRST + 42

the Keypad 1 key

8.1.1.40 #define KEY_KEYPAD_2 KEY_FIRST + 43

the Keypad 2 key

8.1.1.41 #define KEY_KEYPAD_3 KEY_FIRST + 44

the Keypad 3 key

8.1.1.42 #define KEY_KEYPAD_4 KEY_FIRST + 45

the Keypad 4 key

8.1.1.43 #define KEY_KEYPAD_5 KEY_FIRST + 46

the Keypad 5 key

8.1.1.44 #define KEY_KEYPAD_6 KEY_FIRST + 47

the Keypad 6 key

8.1.1.45 #define KEY_KEYPAD_7 KEY_FIRST + 48

the Keypad 7 key

8.1.1.46 #define KEY_KEYPAD_8 KEY_FIRST + 49

the keypad 8 key

8.1.1.47 #define KEY_KEYPAD_9 KEY_FIRST + 50

the Keypad 9 key

8.1.1.48 #define KEY_KEYPAD_ADD KEY_FIRST + 38

the Keypad Add key

8.1.1.49 #define KEY_KEYPAD_DIVIDE KEY_FIRST + 35

the KeyPad Divide key

8.1.1.50 #define KEY_KEYPAD_ENTER KEY_FIRST + 39

the Keypad Enter key

8.1.1.51 #define KEY_KEYPAD_MULTIPLY KEY_FIRST + 36

the Keypad Multiply key

8.1.1.52 #define KEY_KEYPAD_PERIOD KEY_FIRST + 40

the Keypad Period/Decimal key

8.1.1.53 #define KEY_KEYPAD_SUBTRACT KEY_FIRST + 37

the Keypad Subtract key

8.1.1.54 #define KEY_LAST KEY_ESCAPE

the last key to be supported

8.1.1.55 #define KEY_LEFTALT KEY_FIRST + 19

the left Alternate key

8.1.1.56 #define KEY_LEFTCONTROL KEY_FIRST + 15

the left Control key

8.1.1.57 #define KEY_LEFTSHIFT KEY_FIRST + 13

the left Shift key

8.1.1.58 #define KEY_LEFTWINDOW KEY_FIRST + 17

the left Window key

8.1.1.59 #define KEY_NUMLOCK KEY_FIRST + 24

the NumLock key

8.1.1.60 #define KEY_PAGEDOWN KEY_FIRST + 30

the PageDown key

8.1.1.61 #define KEY_PAGEUP KEY_FIRST + 28

the PageUp key

8.1.1.62 #define KEY_PAUSE KEY_FIRST + 25

the pause/break key

8.1.1.63 #define KEY_PRINTSCREEN KEY_FIRST + 22

the PrintScreen key

8.1.1.64 #define KEY_RIGHTALT KEY_FIRST + 20

the right Alternate key

8.1.1.65 #define KEY_RIGHTCONTROL KEY_FIRST + 16

the right Control key

8.1.1.66 #define KEY_RIGHTSHIFT KEY_FIRST + 14

the right Shift key

8.1.1.67 #define KEY_RIGHTWINDOW KEY_FIRST + 18

the right Window key

8.1.1.68 #define KEY_SCROLLLOCK KEY_FIRST + 23

the ScrollLock key

8.1.1.69 #define KEY_TAB KEY_FIRST + 52

the Tab key

8.1.1.70 #define KEYSTATE_DOWN 1

the key is currently up

8.1.1.71 #define KEYSTATE_UP 0

the key is currently down

8.1.1.72 #define LINUX_DECORATOR 2

8.1.1.73 #define LINUX_DECORATOR_BORDER 1L << 1

8.1.1.74 #define LINUX_DECORATOR_CLOSE 1L << 5

8.1.1.75 #define LINUX_DECORATOR_MAXIMIZE 1L << 4

8.1.1.76 #define LINUX_DECORATOR_MINIMIZE 1L << 3

8.1.1.77 #define LINUX_DECORATOR_MOVE 1L << 2

8.1.1.78 #define LINUX_FUNCTION 1

8.1.1.79 #define MOUSE_BUTTONDOWN 1

the mouse button is currently down

8.1.1.80 #define MOUSE_BUTTONUP 0

the mouse button is currently up

8.1.1.81 #define MOUSE_LAST MOUSE_MIDDLEBUTTON + 1

the last mouse button to be supported

8.1.1.82 #define MOUSE_LEFTBUTTON 0

the left mouse button

8.1.1.83 #define MOUSE_MIDDLEBUTTON 2

the middle mouse button / ScrollWheel

8.1.1.84 #define MOUSE_RIGHTBUTTON 1

the right mouse button

8.1.1.85 #define MOUSE_SCROLL_DOWN 0

the mouse wheel up

8.1.1.86 #define MOUSE_SCROLL_UP 1

the mouse wheel down

8.1.1.87 #define TINYWIDNOW_ERROR_WINDOW_NOT_FOUND 11

if the window was not found in the window manager

8.1.1.88 #define TINYWINDOW_ERROR_ALREADY_INITIALIZED 8

if the window was already initialized

8.1.1.89 #define TINYWINDOW_ERROR_EXISTING_CONTEXT 6

if the window already has an OpenGL context

8.1.1.90 #define TINYWINDOW_ERROR_FUNCTION_NOT_IMPLEMENTED 14

if the function has not yet been implemented in the current version of the API

8.1.1.91 #define TINYWINDOW_ERROR_INVALID_CONTEXT 5

if the OpenGL context for the window is invalid

8.1.1.92 #define TINYWINDOW_ERROR_INVALID_EVENT 10

if the given event callback was invalid

8.1.1.93 #define TINYWINDOW_ERROR_INVALID_RESOLUTION 4

if an invalid window resolution was given

8.1.1.94 #define TINYWINDOW_ERROR_INVALID_TITLEBAR 9

if the Title-bar text given was invalid

8.1.1.95 #define TINYWINDOW_ERROR_INVALID_WINDOW 13

8.1.1.96 #define TINYWINDOW_ERROR_INVALID_WINDOW_INDEX 2

if an invalid window index was given

8.1.1.97 #define TINYWINDOW_ERROR_INVALID_WINDOW_NAME 1

if an invalid window name was given

8.1.1.98 #define TINYWINDOW_ERROR_INVALID_WINDOW_STATE 3

if an invalid window state was given

8.1.1.99 #define TINYWINDOW_ERROR_INVALID_WINDOWSTYLE 12

if the window style gives is invalid

8.1.1.100 #define TINYWINDOW_ERROR_LINUX_CANNOT_CONNECT_X_SERVER 15

Linux: if cannot connect to X11 server

8.1.1.101 #define TINYWINDOW_ERROR_LINUX_CANNOT_CREATE_WINDOW 17

Linux: when X11 fails to create a new window

8.1.1.102 #define TINYWINDOW_ERROR_LINUX_FUNCTION_NOT_IMPLEMENTED 18

Linux: when the function has not yet been implemented on the Linux in the current version of the API

8.1.1.103 #define TINYWINDOW_ERROR_LINUX_INVALID_VISUALINFO 16

Linux: if visual information given was invalid

8.1.1.104 #define TINYWINDOW_ERROR_NO_CONTEXT 0

if a window tries to use a graphical function without a context

8.1.1.105 #define TINYWINDOW_ERROR_NOT_INITIALIZED 7

if the window is being used without being initialized

8.1.1.106 #define TINYWINDOW_ERROR_WINDOWS_CANNOT_CREATE_WINDOW 19

Windows: when Win32 cannot create a window

8.1.1.107 #define TINYWINDOW_ERROR_WINDOWS_CANNOT_INITIALIZE 20

Windows: when Win32 cannot initialize

8.1.1.108 #define TINYWINDOW_ERROR_WINDOWS_FUNCTION_NOT_IMPLEMENTED 21

Windows: when a function has yet to be implemented on the Windows platform in the current version of the API

8.1.1.109 #define TINYWINDOW_WARNING_NO_GL_EXTENSIONS 1

if your computer does not support any OpenGL extensions

8.1.1.110 #define TINYWINDOW_WARNING_NOT_CURRENT_CONTEXT 0

if using calling member functions of a window that is not the current window being drawn to

8.1.1.111 #define WINDOWSTATE_FULLSCREEN 3

the window is currently full screen

8.1.1.112 #define WINDOWSTATE_MAXIMIZED 1

the window is currently maximized

8.1.1.113 #define WINDOWSTATE_MINIMIZED 2

the window is currently minimized

8.1.1.114 #define WINDOWSTATE_NORMAL 0

the window is in its default state

8.1.1.115 #define WINDOWSTYLE_BARE 1

the window has no decorators but the window border and title bar

8.1.1.116 #define WINDOWSTYLE_DEFAULT 2

the default window style for the respective platform

8.1.1.117 #define WINDOWSTYLE_POPUP 3

the window has no decorators

8.1.2 Typedef Documentation

8.1.2.1 typedef void(* onDestroyedEvent_t)(void)

To be called when the window is being destroyed

8.1.2.2 typedef void(* onFocusEvent_t)(GLboolean inFocus)

To be called when the window has gained event focus

8.1.2.3 typedef void(* onKeyEvent_t)(GLuint key, GLboolean keyState)

To be called when a key event has occurred

8.1.2.4 typedef void(* onMaximizedEvent_t)(void)

To be called when the window has been maximized

8.1.2.5 typedef void(* onMinimizedEvent_t)(void)

To be called when the window has been minimized

8.1.2.6 typedef void(* onMouseButtonEvent_t)(GLuint button, GLboolean buttonState)

To be called when a Mouse button event has occurred

8.1.2.7 typedef void(* onMouseMoveEvent_t)(GLuint windowX, GLuint windowY, GLuint screenX, GLuint screenY)

To be called when the mouse has been moved within the window

8.1.2.8 typedef void(* onMouseWheelEvent_t)(GLuint wheelDirection)

To be called when a mouse wheel event has occurred.

8.1.2.9 typedef void(* onMovedEvent_t)(GLuint x, GLuint y)

To be called when the window has been moved

8.1.2.10 typedef void(* onResizeEvent_t)(GLuint width, GLuint height)

To be called when the window has been resized

8.1.3 Function Documentation

8.1.3.1 static void PrintErrorMessage (GLuint *errorNumber*) [static]

< if a window tries to use a graphical function without a context

< if an invalid window name was given

< if an invalid window index was given

< if an invalid window state was given

< if an invalid window resolution was given

< if the OpenGL context for the window is invalid

< if the window already has an OpenGL context

< if the window is being used without being initialized

< if the window was already initialized

< if the Title-bar text given was invalid

< if the given event callback was invalid

< if the window was not found in the window manager

< if the window style gives is invalid

< if the function has not yet been implemented in the current version of the API

< Linux: if cannot connect to X11 server

< Linux: if visual information given was invalid

< Linux: when X11 fails to create a new window

< Linux: when the function has not yet been implemented on the Linux in the current version of the API

< Windows: when Win32 cannot create a window

< Windows: when a function has yet to be implemented on the Windows platform in the current version of the API

```
00205 {
00206     switch ( errorNumber )
00207     {
00208         case TINYWINDOW_ERROR_NO_CONTEXT:
00209         {
00210             printf( "Error: An OpenGL context must first be created( initialize the window ) \n" );
00211             break;
00212         }
00213 
00214         case TINYWINDOW_ERROR_INVALID_WINDOW_NAME:
00215         {
00216             printf( "Error: invald window name \n" );
00217             break;
00218         }
00219 
00220         case TINYWINDOW_ERROR_INVALID_WINDOW_INDEX:
00221         {
00222             printf( "Error: invalid window index \n" );
00223             break;
00224         }
00225 
00226         case TINYWINDOW_ERROR_INVALID_WINDOW_STATE:
00227         {
00228             printf( "Error: invalid window state \n" );
00229             break;
00230         }
00231 
00232         case TINYWINDOW_ERROR_INVALID_RESOLUTION:
00233         {
```

```
00234             printf( "Error: invalid resolution \n" );
00235             break;
00236         }
00237
00238         case TINYWINDOW_ERROR_INVALID_CONTEXT:
00239         {
00240             printf( "Error: Failed to create OpenGL context \n" );
00241             break;
00242         }
00243
00244         case TINYWINDOW_ERROR_EXISTING_CONTEXT:
00245         {
00246             printf( "Error: context already created \n" );
00247             break;
00248         }
00249
00250         case TINYWINDOW_ERROR_NOT_INITIALIZED:
00251         {
00252             printf( "Error: Window manager not initialized \n" );
00253             break;
00254         }
00255
00256         case TINYWINDOW_ERROR_ALREADY_INITIALIZED:
00257         {
00258             printf( "Error: window has already been initialized \n" );
00259             break;
00260         }
00261
00262         case TINYWINDOW_ERROR_INVALID_TITLEBAR:
00263         {
00264             printf( "Error: invalid title bar name ( cannot be null or nullptr ) \n" );
00265             break;
00266         }
00267
00268         case TINYWINDOW_ERROR_INVALID_EVENT:
00269         {
00270             printf( "Error: invalid event callback given \n" );
00271             break;
00272         }
00273
00274         case TINYWIDNOW_ERROR_WINDOW_NOT_FOUND:
00275         {
00276             printf( "Error: window was not found \n" );
00277             break;
00278         }
00279
00280         case TINYWINDOW_ERROR_INVALID_WINDOWSTYLE:
00281         {
00282             printf( "Error: invalid window style given \n" );
00283             break;
00284         }
00285
00286         case TINYWINDOW_ERROR_INVALID_WINDOW:
00287         {
00288             printf( "Error: invalid window given \n" );
00289             break;
00290         }
00291
00292         case TINYWINDOW_ERROR_FUNCTION_NOT_IMPLEMENTED:
00293         {
00294             printf( "Error: I'm sorry but this function has not been implemented yet :( \n" );
00295             break;
00296         }
00297
00298         case TINYWINDOW_ERROR_LINUX_CANNOT_CONNECT_X_SERVER:
00299         {
00300             printf( "Error: cannot connect to X server \n" );
00301             break;
00302         }
00303
00304         case TINYWINDOW_ERROR_LINUX_INVALID_VISUALINFO:
00305         {
00306             printf( "Error: Invalid visual information given \n" );
00307             break;
00308         }
00309
00310         case TINYWINDOW_ERROR_LINUX_CANNOT_CREATE_WINDOW:
00311         {
00312             printf( "Error: failed to create window \n" );
00313             break;
00314         }
00315
00316         case TINYWINDOW_ERROR_LINUX_FUNCTION_NOT_IMPLEMENTED
    :
00317         {
00318             printf( "Error: function not implemented on linux platform yet. sorry :( \n" );
00319             break;
```

```
00320         }
00321
00322         case TINYWINDOW_ERROR_WINDOWS_CANNOT_CREATE_WINDOW:
00323         {
00324             printf( "Error: failed to create window \n" );
00325             break;
00326         }
00327
00328         case TINYWINDOW_ERROR_WINDOWS_FUNCTION_NOT_IMPLEMENTED
      :
00329         {
00330             printf( "Error: function not implemented on Windows platform yet. sorry ;( \n" );
00331             break;
00332         }
00333
00334         default:
00335         {
00336             printf( "Error: unspecified Error \n" );
00337             break;
00338         }
00339     }
00340 }
```

8.1.3.2 static void PrintWarningMessage (GLuint *warningNumber*) [static]

< if your computer does not support any OpenGL extensions

< if using calling member functions of a window that is not the current window being drawn to

```
00180 {
00181     switch ( warningNumber )
00182     {
00183         case TINYWINDOW_WARNING_NO_GL_EXTENSIONS:
00184         {
00185             printf( "Warning: no OpenGL extensions available \n" );
00186             break;
00187         }
00188
00189         case TINYWINDOW_WARNING_NOT_CURRENT_CONTEXT:
00190         {
00191             printf( "Warning: window not the current OpenGL context being rendered to \n" );
00192             break;
00193         }
00194
00195         default:
00196         {
00197             printf( "Warning: unspecified warning \n" );
00198             break;
00199         }
00200     }
00201 }
```