GLFW

Users Guide

Document release 1.4

SUMMARY

This document describes the general usage of the GLFW 2.4 API. Most of the API functions are described here, but for a complete list of functions you should refer to the *GLFW Reference Manual*.

TRADEMARKS

TABLE OF CONTENTS

1. INTRODUCTION

GLFW is a portable API (Application Program Interface) that handles operating system specific tasks related to OpenGL programming. While OpenGL in general is portable, easy to use and often results in tidy and compact code, the operating system specific mechanisms that are required to set up and manage an OpenGL window are quite the opposite. GLFW tries to remedy this by providing the following functionality:

- Opening and maintaining an OpenGL window.
- Keyboard, mouse and joystick input.
- A high precision timer.
- Multi threading support.
- Support for querying and using OpenGL extensions.
- Image file loading support.

All this functionality is implemented as a set of easy-to-use functions, which makes it possible to write an OpenGL application framework in just a few lines of code.

The GLFW API is completely operating system and platform independent, which makes it very simple to port GLFW based OpenGL applications to a variety of platforms. Currently supported platforms are:

- Microsoft Windows® 95/98/ME/NT/2000/XP/.NET Server.
- Unix® or Unix-like systems[1] running the X Window System™, e.g. Linux®, IRIX® and Solaris™.

[1] For threading support GLFW uses the POSIX™ threading package (pthread), which is supported by most modern Unix-like systems.

2. GETTING STARTED

In this chapter you will learn how to write a simple OpenGL application using GLFW. We start by initializing GLFW, then we open a window and read some user keyboard input.

2.1 Initializing GLFW

Before using any of the GLFW functions, it is necessary to call **glfwInit**. It initializes internal working variables which are used by other GLFW functions. The C syntax is:

```
int glfwInit( void )
```

glfwInit returns *GL_TRUE* if initialization succeeded, or *GL_FALSE* if it failed.

When your application is done using GLFW, typically at the very end of the program, you should call **glfwTerminate**, which makes a clean up and places GLFW in a non-initialized state (i.e. it is necessary to call **glfwInit** again before using any GLFW functions). The C syntax is:

```
void glfwTerminate( void )
```

Among other things, **glfwTerminate** closes the OpenGL window unless it was closed manually, and kills any running threads that were created using GLFW.

2.2 Opening An OpenGL Window

Opening an OpenGL window is done with the function **glfwOpenWindow**. The function takes nine arguments, which are used to describe the following properties of the window to open:

- Window dimensions (width and height) in pixels.
- Color and alpha buffer depth.
- Depth buffer (Z-buffer) depth.
- Stencil buffer depth.
- Fullscreen or windowed mode.

The C language syntax for **glfwOpenWindow** is:

```
int glfwOpenWindow( int width, int height,
    int redbits, int greenbits, int bluebits,
    int alphabits, int depthbits, int stencilbits,
    int mode )
```

glfwOpenWindow returns *GL_TRUE* if the window was opened correctly, or *GL_FALSE* if GLFW failed to open the window.

GLFW tries to open a window which best matches the requested parameters. Some parameters may be omitted by setting them to zero, which will result in GLFW either using a default value, or the related functionality to be disabled. For instance, if *width* and *height* are both zero, GLFW will use a window resolution of 640x480. If *depthbits* is zero, the opened window may not have a depth buffer.

The *mode* argument is used to specify if the window is to be a s.c. fullscreen window, or a regular window.

If *mode* is *GLFW_FULLSCREEN*, the window will cover the entire screen and no window borders will be visible. If possible, the video mode will be changed to the mode that closest matches the *width*, *height*, *redbits*, *greenbits*, *bluebits* and *alphabits* arguments. Furthermore, the mouse pointer will be hidden, and screensavers are prohibited. This is usually the best mode for games and demos.

If *mode* is *GLFW_WINDOW*, the window will be opened as a normal window on the desktop. The mouse pointer will not be hidden, and screensavers are allowed to be activated.

To close the window, you can either use **glfwTerminate**, as described earlier, or you can use the more explicit approach by calling **glfwCloseWindow**, which has the C syntax:

```
    void glfwCloseWindow( void )
```

2.3 Using Keyboard Input

GLFW provides several means for receiving user input, which will be discussed in more detail later on in this manual. One of the simplest ways of checking for keyboard input is to use the function **glfwGetKey**:

```
    int glfwGetKey( int key )
```

It queries the current status of individual keyboard keys. The argument *key* specifies which key to check, and it can be either an uppercase printable ISO 8859-1 (Latin 1) character (e.g. 'A', '3' or '.'), or a special key identifier (see the *GLFW Reference Manual* for a list of special key identifiers). **glfwGetKey** returns *GLFW_PRESS* (or 1) if the key is currently held down, or *GLFW_RELEASE* (or 0) if the key is not being held down. For example:

```
    A_pressed = glfwGetKey( ‘A’ );
    esc_pressed = glfwGetKey( GLFW_KEY_ESC );
```

In order for **glfwGetKey** to have any effect, you need to poll for input events on a regular basis. This can be done in one of two ways:

1. Implicitly by calling **glfwSwapBuffers** often.
2. Explicitly by calling **glfwPollEvents** often.

In general you do not have to care about this, since you will normally call **glfwSwapBuffers** to swap front and back rendering buffers every animation frame anyway. If, however, this is not the case, you should call **glfwPollEvents** in the order of 10-100 times per second in order for GLFW to maintain an up to date input state.

2.4 Putting It Together: A Minimal GLFW Application

Now that you know how to initialize GLFW, open a window and poll for keyboard input, let us exemplify this with a simple OpenGL program. In the following example error-checking has been omitted for the sake of brevity:

```
#include <GL/glfw.h>

int main( void )
{
    int running = GL_TRUE;

    // Initialize GLFW
    glfwInit();

    // Open an OpenGL window
    if( !glfwOpenWindow( 300,300, 0,0,0,0,0,0, GLFW_WINDOW ) )
    {
        glfwTerminate();
        return 0;
    }

    // Main loop
    while( running )
    {
        // OpenGL rendering goes here...
        glClear( GL_COLOR_BUFFER_BIT );

        // Swap front and back rendering buffers
        glfwSwapBuffers();

        // Check if ESC key was pressed or window was closed
        running = !glfwGetKey( GLFW_KEY_ESC ) &&
                  glfwGetWindowParam( GLFW_OPENED );
    }

    // Close window and terminate GLFW
    glfwTerminate();

    // Exit program
    return 0;
}
```

The program opens a 300x300 window and runs in a loop until the escape key is pressed, or the window was closed. All the OpenGL “rendering” that is done in this example is to clear the window, using the **glClear** function.

3. WINDOW OPERATIONS

In this chapter, you will learn more about window related GLFW functionality, including: setting and getting window properties, buffer swap control and video mode querying.

3.1 Setting Window Properties

In the previous chapter the **glfwOpenWindow** function was described, which specifies the sizes of the color, alpha, depth and stencil buffers. It is also possible to request an accumulator buffer and/or auxiliary buffers by using the **glfwOpenWindowHint** function:

```
    void glfwOpenWindowHint( int target, int hint )
```

The *target* argument can be one of the constants listed in the table below, and *hint* is the value to assign to the specified target.

Name	**Description**
GLFW_REFRESH_RATE	Vertical monitor refresh rate in Hz (only used for fullscreen windows).
GLFW_ACCUM_RED_BITS	Number of bits for the red channel of the accumulator buffer.
GLFW_ACCUM_GREEN_BITS	Number of bits for the green channel of the accumulator buffer.
GLFW_ACCUM_BLUE_BITS	Number of bits for the blue channel of the accumulator buffer.
GLFW_ACCUM_ALPHA_BITS	Number of bits for the alpha channel of the accumulator buffer.
GLFW_AUX_BUFFERS	Number of auxiliary buffers.
GLFW_STEREO	Specify if stereo rendering should be supported (can be GL_TRUE or GL_FALSE).

Table 1: Parameters which are possible to set using glfwOpenWindowHint.

For a hint to have any effect, the **glfwOpenWindowHint** function must be called before opening the window with the **glfwOpenWindow** function.

To request an accumulator buffer, set the *GLFW_ACCUM_x_BITS* targets to values greater than zero (usually eight or sixteen bits per component). To request auxiliary buffers, set the *GLFW_AUX_BUFFERS* target to a value greater than zero. To request a stereo rendering capable window, set the *GLFW_STEREO* target to *GL_TRUE*. The *GLFW_REFRESH_RATE* target should be used with caution, since it may result in suboptimal operation, or even a blank or damaged screen.

Besides the parameters that are given with the **glfwOpenWindow** and **glfwOpenWindowHint** functions, a few more properties of a window can be changed after the window has been opened, namely the window title, window size, and window position.

To change the window title of an open window, use the **glfwSetWindowTitle** function:

```
void glfwSetWindowTitle( const char *title )
```

title is a null terminated ISO 8859-1 (8-bit Latin 1) string that will be used as the window title. It will also be used as the application name (for instance in the application list when using ALT+TAB under Windows, or as the icon name when the window is iconified under X11). The default window name is “GLFW Window”, which will be used unless **glfwSetWindowTitle** is called after the window has been opened.

To change the size of a window, call **glfwSetWindowSize**:

```
void glfwSetWindowSize( int width, int height )
```

Where *width* and *height* are the new dimensions of the window.

To change the position of a window, call **glfwSetWindowPos**:

```
void glfwSetWindowPos( int x, int y )
```

Where *x* and *y* are the new desktop coordinates of the window. This function does not have any effect when in fullscreen mode.

3.2 Getting Window Properties

When opening a window, the opened window will not necessarily have the requested properties, so you should always check the parameters that your application relies on (e.g. number of stencil bits) using **glfwGetWindowParam**, which has the C syntax:

```
int glfwGetWindowParam( int param )
```

The argument *param* can be one of the tokens listed in Table 2, and the return value is an integer holding the requested value.

Name	**Return value**
GLFW_OPENED	*GL_TRUE* if window is opened, else *GL_FALSE*.
GLFW_ACTIVE	*GL_TRUE* if window is selected and active, else *GL_FALSE*.
GLFW_ICONIFIED	*GL_TRUE* if window is iconified, else *GL_FALSE*.
GLFW_ACCELERATED[2]	*GL_TRUE* if window is hardware accelerated.
GLFW_RED_BITS	Number of bits for the red color component.
GLFW_GREEN_BITS	Number of bits for the green color component.
GLFW_BLUE_BITS	Number of bits for the blue color component.
GLFW_ALPHA_BITS	Number of bits for the alpha buffer.
GLFW_DEPTH_BITS	Number of bits for the depth buffer.
GLFW_STENCIL_BITS	Number of bits for the stencil buffer.
GLFW_REFRESH_RATE[3]	Vertical refresh rate in Hz.
GLFW_ACCUM_RED_BITS	Number of bits for the red channel of the accumulator buffer.
GLFW_ACCUM_GREEN_BITS	Number of bits for the green channel of the accumulator buffer.
GLFW_ACCUM_BLUE_BITS	Number of bits for the blue channel of the accumulator buffer.
GLFW_ACCUM_ALPHA_BITS	Number of bits for the alpha channel of the accumulator buffer.
GLFW_AUX_BUFFERS	Number of auxiliary buffers.
GLFW_STEREO	*GL_TRUE* if stereo rendering is supported, else *GL_FALSE*.

Table 2: Parameters which are possible to query using glfwGetWindowParam.

Another useful function is **glfwSetWindowSizeCallback**, which specifies a user function that will be called every time the window size has changed. The C syntax is:

```
    void glfwSetWindowSizeCallback( GLFWwindowsizefun cbfun )
```

The user function *fun* should be of the type:

```
    void GLFWCALL fun( int width, int height )
```

The first argument passed to the user function is the width of the window, and the second argument is the height of the window. Here is an example of how to use a window size callback function:

[2] *GLFW_ACCELERATED* is only supported under Windows.
[3] *GLFW_REFRESH_RATE* is only supported under Windows and Xfree86. Other systems will return zero when *GLFW_REFRESH_RATE* is selected.

```
int WinWidth, WinHeight;

void GLFWCALL WindowResize( int width, int height )
{
    WinWidth  = width;
    WinHeight = height;
}

int main( void )
{
    ...
    glfwSetWindowSizeCallback( WindowResize );
    ...
}
```

Using a callback function for getting the window size is mostly useful for windowed applications, since the window size may be changed at any time by the user. It can also be used to determine the actual fullscreen resolution.

An alternative to using a callback function for getting the window size, is to use the function **glfwGetWindowSize**:

```
void glfwGetWindowSize( int *width, int *height )
```

The *width* and *height* arguments are filled out with the current window dimensions.

3.3 Buffer Swapping

GLFW windows are always double buffered. That means that you have two rendering buffers; a *front* buffer and a *back* buffer. The front buffer is the buffer that is being displayed, and the back buffer is not displayed. OpenGL lets you select which of these two buffers you want to render to (with the **glDrawBuffer** command), but the default (and preferred) rendering buffer is the back buffer. This way you will avoid flickering and artifacts caused by graphics being only partly drawn at the same time as the video raster beam is displaying the graphics on the monitor.

When an entire frame has been rendered to the back buffer, it is time to swap the back and the front buffers in order to display the rendered frame, and begin rendering a new frame. This is done with the command **glfwSwapBuffers**. The C syntax is:

```
void glfwSwapBuffers( void )
```

Besides swapping the front and back rendering buffers, **glfwSwapBuffers** also calls **glfwPollEvents** (this behavior can be disabled by calling **glfwDisable** with the argument *GLFW_AUTO_POLL_EVENTS*). This is to ensure frequent polling of events, such as keyboard and mouse input, and window reshaping events.

Sometimes it can be useful to select when the buffer swap will occur. With the function **glfwSwapInterval** it is possible to select the minimum number of vertical retraces the video raster line should do before swapping the buffers:

```
void glfwSwapInterval( int interval )
```

If *interval* is zero, the swap will take place immediately when **glfwSwapBuffers** is called, without waiting for a vertical retrace (also known as “vsync off”). Otherwise at least *interval* retraces will pass between each buffer swap (also known as “vsync on”). Using a swap interval of zero can be useful for benchmarking purposes, when it is not desirable to measure the time it takes to wait for the vertical retrace. However, a swap interval of 1 generally gives better visual quality.

It should be noted that not all OpenGL implementations and hardware supports this function, in which case **glfwSwapInterval** will have no effect. Sometimes it is only possible to affect the swap interval through driver settings (e.g. the display settings under Windows, or as an environment variable setting under Unix).

3.4 Querying Video Modes

Although GLFW generally does a good job at selecting a suitable video mode for you when you open a fullscreen window, it is sometimes useful to know exactly which modes are available on a certain system. For example, you may want to present the user with a list of video modes to select from. To get a list of available video modes, you can use the function **glfwGetVideoModes**:

```
int glfwGetVideoModes( GLFWvidmode *list, int maxcount )
```

The argument *list* is a vector of *GLFWvidmode* structures, and *maxcount* is the maximum number of video modes that your vector can hold. **glfwGetVideoModes** will return the actual number of video modes detected on the system.

The *GLFWvidmode* structure looks like this:

```
typedef struct {
   int Width, Height;     // Video resolution
   int RedBits;           // Red bits per pixel
   int GreenBits;         // Green bits per pixel
   int BlueBits;          // Blue bits per pixel
} GLFWvidmode;
```

Here is an example of retrieving all available video modes:

```
int nummodes;
GLFWvidmode list[ 200 ];
nummodes = glfwGetVideoModes( list, 200 );
```

The returned list is sorted, first by color depth (*RedBits* + *GreenBits* + *BlueBits*), and then by resolution (*Width*Height*), with the lowest resolution, fewest bits per pixel mode first.

To get the desktop video mode, use the function **glfwGetDesktopMode**:

```
void glfwGetDesktopMode( GLFWvidmode *mode )
```

The function returns the resolution and color depth of the user desktop in the *mode* structure. Note that the user desktop mode is independent of the current video mode if a GLFW fullscreen window has been opened.

4. INPUT HANDLING

In this chapter you will learn how to use keyboard, mouse and joystick input, using either polling or callback functions.

4.1 Event Polling

The first thing to know about input handling in GLFW is that all keyboard and mouse input is collected by checking for input events. This has do be done manually by calling either **glfwPollEvents** or **glfwSwapBuffers** (which implicitly calls **glfwPollEvents** for you). Normally this does not have to be a concern, since **glfwSwapBuffers** is called every frame, which should be often enough (about 10-100 times per second for a normal OpenGL application). One exception is when rendering is paused, and then the program waits for input to begin animation again. In this case **glfwPollEvents** has to be called repeatedly until any new input events arrive.

If it is not desirable that **glfwPollEvents** is called implicitly from **glfwSwapBuffers**, call **glfwDisable** with the argument *GLFW_AUTO_POLL_EVENTS*.

Note that for joystick input event polling is not needed, since all relevant joystick state is gathered every time a joystick function is called.

4.2 Keyboard Input

GLFW gives three options for getting keyboard input:

- Manually polling the state of individual keys.
- Automatically receive new key state for any key, using a callback function.
- Automatically receive characters, using a callback function.

Depending on what the keyboard input will be used for, either of the methods may be more suitable. The main difference between the two last options is that while characters are affected by modifier keys (such as shift), key state is independent of any modifier keys. Also, special keys (such as function keys, cursor keys and modifier keys) are not reported to the character callback function.

To check if a key is held down or not at any given moment, use the function **glfwGetKey**:

```
int glfwGetKey( int key )
```

It queries the current status of individual keyboard keys. The argument *key* specifies which key to check, and it can be either an uppercase ASCII character, or a special key identifier. **glfwGetKey** returns *GLFW_PRESS* (or 1) if the key is currently held down, or *GLFW_RELEASE* (or 0) if the key is not being held down.

In most situations, it may be useful to know if a key has been pressed and released between two calls to **glfwGetKey** (especially if the animation is fairly slow, which may allow the user to press and release a key between two calls to **glfwGetKey**). This can be accomplished by enabling *sticky keys*, which is done by calling **glfwEnable** with the argument *GLFW_STICKY_KEYS*, as in the following example:

```
    glfwEnable( GLFW_STICKY_KEYS );
```

When sticky keys are enabled, a key will not be released until it is checked with **glfwGetKey**. To disable sticky keys, call **glfwDisable** witht the argument *GLFW_STICKY_KEYS*. Then all keys that are not currently held down will be released, and future key releases will take place immediately when the user releases the key, without waiting for **glfwGetKey** to be called. By default sticky keys are disabled.

Sticky keys are often very useful and should be used in most cases. There is however a danger involved with enabling sticky keys, and that is that keys which are pressed by the user and are not checked with **glfwGetKey,** may remain pressed for a very long time. A typical situation where this may be dangerous is in a program that consists of two or more sections (e.g. a menu section and a game section). If the first section enables sticky keys but does not check for keys which the second section checks for, there is a potential of recording many key presses in the first section which will be detected in the second section. To avoid this problem, always disable sticky keys before leaving a section of a program.

An alternative to using **glfwGetKey** is to register a keyboard input callback function with **glfwSetKeyCallback**:

```
    void glfwSetKeyCallback( GLFWkeyfun cbfun )
```

The argument *fun* is a pointer to a callback function. The callback function shall take two integer arguments. The first is the key identifier, and the second is the new key state, which can be *GLFW_PRESS* or *GLFW_RELEASE*. To unregister a callback function, call **glfwSetKeyCallback** with *fun* = *NULL*.

A callback function can be useful in some situations. For instance it can replace multiple **glfwGetKey** calls with a *switch/case* statement.

If the keyboard is to be used as a text input device (e.g. in a user dialog) rather than as a set of independent buttons, a character callback function is more suitable. To register a character callback function, use **glfwSetCharCallback**:

```
    void glfwSetCharCallback( GLFWcharfun cbfun )
```

The argument *fun* is a pointer to a callback function. The callback function shall take two integer arguments. The first is a Unicode character code, and the second is *GLFW_PRESS* if the key that generated the character was pressed, or *GLFW_RELEASE* if it was released. To unregister a callback function, call **glfwSetCharCallback** with *fun* = *NULL*.

The Unicode character set is an international standard for encoding characters. It is much more comprehensive than seven or eight bit character sets (e.g. US-ASCII and Latin 1), and includes characters for most written languages in the world. It should be noted that Unicode character codes 0 to 255 are the same as for ISO 8859-1 (Latin 1), so as long as a proper range check is performed on the Unicode character code, it can be used just as an eight bit Latin 1 character code (which can be useful if full Unicode support is not possible).

4.2.1 Key repeat

By default, GLFW does not report key repeats when a key is held down. To activate key repeat, call **glfwEnable** with the argument *GLFW_KEY_REPEAT*:

```
glfwEnable( GLFW_KEY_REPEAT );
```

This will let a registered key or character callback function receive key repeat events when a key is held down.

4.2.2 Special system keys

On most systems there are some special system keys that are normally not intercepted by an application. For instance, under Windows it is possible to switch programs by pressing ALT+TAB, which brings up a list of running programs to select from. In certain situations it can be desirable to prevent such special system keys from interfering with the program. With GLFW it is possible to do by calling **glfwDisable** with the argument *GLFW_SYSTEM_KEYS*:

```
glfwDisable( GLFW_SYSTEM_KEYS );
```

By doing so, most system keys will have no effect and will not interfere with your program. System keys can be re-enabled by calling **glfwEnable** with the argument *GLFW_SYSTEM_KEYS*. By default, system keys are enabled.

4.3 Mouse Input

Just like for keyboard input, mouse input can be realized with either polling or callback functions.

4.3.1 Mouse position

To read the mouse position, you can use the function **glfwGetMousePos**:

```
void glfwGetMousePos( int *x, int *y )
```

The arguments *x* and *y* point to integer variables that will be updated with the current absolute mouse position. An alternative is to use a callback function instead, which can be set with **glfwSetMousePosCallback**:

```
void glfwSetMousePosCallback( GLFWmouseposfun cbfun )
```

The function that *fun* points to will be called every time the mouse position changes. The first argument to the callback function is the mouse x position, and the second argument is the mouse y position.

4.3.2 Mouse buttons

To query the state of a mouse button, call **glfwGetMouseButton**:

```
int glfwGetMouseButton( int button )
```

The argument *button* can be one of the following mouse button identifiers: *GLFW_MOUSE_BUTTON_LEFT*, *GLFW_MOUSE_BUTTON_RIGHT* or *GLFW_MOUSE_BUTTON_MIDDLE*. **glfwGetMouseButton** will return *GLFW_PRESS* (or 1) if the corresponding mouse button is held down, otherwise it will return *GLFW_RELEASE* (or 0).

Just as it is possible to make keys “sticky”, it is also possible to make mouse buttons appear as held down until the button is checked for with **glfwGetMouseButton**. To enable sticky mouse buttons, call **glfwEnable** with the argument *GLFW_STICKY_MOUSE_BUTTONS*. To disable sticky mouse buttons, call **glfwDisable** with the same argument.

When sticky mouse buttons are enabled, a mouse button will not be released until it is checked with **glfwGetMouseButton**. To disable sticky mouse buttons, call **glfwDisable** witht the argument *GLFW_STICKY_MOUSE_BUTTONS*. Then all mouse buttons that are not currently held down will be released, and future mouse button releases will take place immediately when the user releases the mouse button, without waiting for **glfwGetMouseButton** to be called. By default sticky mouse buttons are disabled.

There is also a callback function for mouse button activities, which can be set with **glfwSetMouseButtonCallback**:

```
void glfwSetMouseButtonCallback( GLFWmousebuttonfun fun )
```

The argument *fun* specifies a function that will be called whenever a mouse button is pressed or released, or *NULL* to unregister a callback function. The first argument to the callback function is a mouse button identifier, and the second is either *GLFW_PRESS* or *GLFW_RELEASE*, depending on the new state of the corresponding mouse button.

4.3.3 Mouse wheel

Some mice have a mouse wheel, which can be thought of as a third mouse axis. To get the position of the mouse wheel, call **glfwGetMouseWheel**:

```
int glfwGetMouseWheel( void )
```

The function returns an integer that represents the position of the mouse wheel. When the user turns the wheel, the wheel position will increase or decrease.

It is also possible to register a callback function for mouse wheel events with the **glfwSetMouseWheelCallback** function:

```
void glfwSetMouseWheelCallback( GLFWmousewheelfun fun )
```

The argument *fun* specifies a function that will be called whenever a mouse wheel is moved, or *NULL* to unregister a callback function. The argument to the callback function is the position of the mouse wheel.

4.3.4 Hiding the mouse cursor

It is possible to hide the mouse cursor with the function call:

```
glfwDisable( GLFW_MOUSE_CURSOR );
```

Hiding the mouse cursor has three effects:

1. The cursor becomes invisible.
2. The cursor is guaranteed to be confined to the window.
3. Mouse coordinates are not limited to the window size.

To show the mouse cursor again, call **glfwEnable** with the argument *GLFW_MOUSE_CURSOR*:

```
glfwEnable( GLFW_MOUSE_CURSOR );
```

By default the mouse cursor is hidden if a window is opened in fullscreen mode, otherwise it is not hidden.

4.4 Joystick Input

GLFW has support for up to sixteen joysticks, and an infinite[4] number of axes and buttons per joystick. Unlike keyboard and mouse input, joystick input does not need an opened window, and **glfwPollEvents** or **glfwSwapBuffers** does not have to be called in order for joystick state to be updated.

4.4.1 Joystick capabilities

First, it is often necessary to determine if a joystick is connected, and what its capabilities are. To get this information the function **glfwGetJoystickParam** can be used:

```
int glfwGetJoystickParam( int joy, int param )
```

[4]There are of course actual limitations posed by the underlying hardware, drivers and operation system.

The *joy* argument specifies which joystick to retrieve the parameter from, and it should be *GLFW_JOYSTICK_n*, where *n* is in the range 1 to 16. The *param* argument specifies which parameter to retrieve. To determine if a joystick is connected, *param* should be *GLFW_PRESENT*, which will cause the function to return *GL_TRUE* if the joystick is connected, or *GL_FALSE* if it is not. To determine the number of axes or buttons that are supported by the joystick, *param* should be *GLFW_JOYSTICK_AXES* or *GLFW_JOYSTICK_BUTTONS*, respectively.

4.4.2 Joystick position

To get the current axis positions of the joystick, the **glfwGetJoystickPos** is used:

```
int glfwGetJoystickPos( int joy, float *pos, int numaxes )
```

As with **glfwGetJoystickParam**, the *joy* argument specifies which joystick to retrieve information from. The *numaxes* argument specifies how many axes to return, and the *pos* argument specifies an array in which all the axis positions are stored. The function returns the actual number of axes that were returned, which could be less than *numaxes* if the joystick does not support all the requested axes, or if the joystick is not connected.

For instance, to get the position of the first two axes (the X and Y axes) of joystick 1, the following code can be used:

```
float  Position[ 2 ];

glfwGetJoystickPos( GLFW_JOYSTICK_1, Position, 2 );
```

After this call, the first element of the *Position* array will hold the X axis position of the joystick, and the second element will hold the Y axis position. In this example we do not use the information about how many axes were really returned.

The position of an axis can be in the range -1.0 to 1.0, where positive values represent right, forward or up directions, while negative values represent left, back or down directions. If a requested axis is not supported by the joystick, the corresponding array element will be set to zero. The same goes for the situation when the joystick is not connected (all axes are treated as unsupported).

4.4.3 Joystick buttons

A function similar to the **glfwGetJoystickPos** function is available for querying the state of joystick buttons, namely the **glfwGetJoystickButtons** function:

```
int glfwGetJoystickButtons( int joy, unsigned char *buttons,
                            int numbuttons )
```

The function works just like the **glfwGetJoystickAxis** function, except that it returns the state of joystick buttons instead of axis positions. Each button in the array specified by the *buttons* argument can be either *GLFW_PRESS* or *GLFW_RELEASE*, telling if the corresponding button is currently held down or not. Unsupported buttons will have the value *GLFW_RELEASE*.

5. TIMING

5.1 High Resolution Timer

In most applications, it is useful to know exactly how much time has passed between point A and point B in a program. A typical situation is in a game, where you need to know how much time has passed between two rendered frames in order to calculate the correct movement and physics etc. Another example is when you want to benchmark a certain piece of code.

GLFW provides a high-resolution timer, which reports a double precision floating point value representing the number of seconds that have passed since **glfwInit** was called. The timer is accessed with the function **glfwGetTime**:

```
double glfwGetTime( void )
```

The precision of the timer depends on which computer and operating system you are running, but it is almost guaranteed to be better than 10 ms, and in most cases it is much better than 1 ms (on a modern PC you can get resolutions in the order of 1 us).

It is possible to set the value of the high precision timer using the **glfwSetTime** function:

```
void glfwSetTime( double time )
```

The argument *time* is the time, in seconds, that the timer should be set to.

5.2 Sleep

Sometimes it can be useful to put a program to sleep for a short time. It can be used to reduce the CPU load in various situations. For this purpose, there is a function called **glfwSleep**, which has the following C syntax:

```
void glfwSleep( double time )
```

The function will put a thread[5] to sleep for the time specified with the argument *time*, which has the unit seconds. When **glfwSleep** is called, the calling thread will be put in waiting state, and thus will not consume any CPU time.

Note that there is generally a minimum sleep time that will be recognized by the operating system, which is usually coupled to the task-switching interval. This minimum time is often in the range 5 – 20 milliseconds, and it is not possible to make a thread sleep for less than that time. Specifying a very small sleep time may result in **glfwSleep** returning immediately, without putting the thread to sleep.

[5] Note that unlike the standard Unix **sleep** function, which puts an entire process to sleep, **glfwSleep** will only put the calling thread to sleep. (For a single threaded program, both functions give the same result).

6. OPENGL EXTENSION SUPPORT

One of the benefits of OpenGL is that it is extensible. Independent hardware vendors (IHVs) may include functionality in their OpenGL implementations which exceed that of the OpenGL standard.

An extension is defined by:

A) An extension name (e.g. *GL_ARB_multitexture*).
B) New OpenGL tokens (e.g. *GL_TEXTURE1_ARB*).
C) New OpenGL functions (e.g. **glActiveTextureARB**).

A list of official extensions, together with their definitions, can be found at the *OpenGL Extension Registry* (http://oss.sgi.com/projects/ogl-sample/registry/).

To use a certain extension, the following steps must be performed:

1. A *compile time* check for the support of the extension.
2. A *run time* check for the support of the extension.
3. Fetch *function pointers* for the extended OpenGL functions (done at run time).

How this is done using GLFW is described in the following sections.

6.1 Compile Time Check

The compile time check is necessary to perform in order to know if the compiler include files have defined the necessary tokens. It is very easy to do. The include file GL/gl.h will define a constant with the same name as the extension, if all the extension tokens are defined. Here is an example of how to check for the extension *GL_ARB_multitexture*:

```
#ifdef GL_ARB_multitexture
     // Extension is supported by the include files
#else
     // Extension is not supported by the include files
     // Update your <GL/gl.h> file!
#endif
```

6.2 Run Time Check

Even if the compiler include files have defined all the necessary tokens, the target system may not support the extension (perhaps it has a different graphic card with a different OpenGL implementation, or it has an older driver). That is why it is necessary to do a run time check for the extension support as well. This is done with the GLFW function **glfwExtensionSupported**, which has the C syntax:

```
     int glfwExtensionSupported( const char *extension )
```

The argument *extension* is a null terminated string with the extension name. **glfwExtensionSupported** returns *GL_TRUE* if the extension is supported, otherwise it returns *GL_FALSE*.

Let us extend the previous example of checking for support of the extension *GL_ARB_multitexture*. This time we add a run time check, and a variable which we set to *GL_TRUE* if the extension is supported, or *GL_FALSE* if it is not supported.

```
int multitexture_supported;

#ifdef GL_ARB_multitexture
     // Check if extension is supported at run time
     multitexture_supported =
         glfwExtensionSupported( “GL_ARB_multitexture” );
#else
     // Extension is not supported by the include files
     // Update your <GL/gl.h> file!
     multitexture_supported = GL_FALSE;
#endif
```

Now it is easy to check for the extension within the program, simply do:

```
     if( multitexture_supported )
     {
         // Use multi texturing
     }
     else
     {
         // Use some other solution (or fail)
     }
```

6.3 Fetching Function Pointers

Some extensions (not all) require the use of new OpenGL functions, which are not necessarily defined by your link libraries. Thus it is necessary to get the function pointers dynamically at run time. This is done with the GLFW function **glfwGetProcAddress**:

```
     void * glfwGetProcAddress( const char *procname )
```

The argument *procname* is a null terminated string holding the name of the OpenGL function. **glfwGetProcAddress** returns the address to the function if the function is available, otherwise *NULL* is returned.

Obviously, fetching the function pointer is trivial. For instance, if we want to obtain the pointer to **glActiveTextureARB**, we simply call:

```
     glActiveTextureARB = glfwGetProcAddress( “glActiveTextureARB” );
```

However, there are many possible naming and type definition conflicts involved with such an operation, which may result in compiler warnings or errors.

My proposed solution is the following:

- Do *not* use the function name for the variable name. Use something similar (perhaps with a prefix or suffix), and then use *#define* to map the function name to your variable.
- The standard type definition naming convention for function pointers is *PFNxxxxPROC*, where *xxxx* is the uppercase version of the function name (e.g. *PFNGLACTIVETEXTUREARBPROC*). Either make sure that a compatible gl.h and/or glext.h file is used by your compiler and rely on it to do the type definitions for you, or use a custom type definition naming convention (e.g. *xxxx_T* or something) and do the type definitions yourself.

Here is an example of how to do it (here we use our own function pointer type defintion):

```
// Type definition of the function pointer
typedef void (APIENTRY * GLACTIVETEXTUREARB_T) (GLenum texture);

// Function pointer
GLACTIVETEXTUREARB_T _ActiveTextureARB;
#define glActiveTextureARB _ActiveTextureARB

int multitexture_supported;

#ifdef GL_ARB_multitexture
     // Check if extension is supported at run time
     if( glfwExtensionSupported( “GL_ARB_multitexture” ) )
     {
         // Get the function pointer
         glActiveTextureARB = (GLACTIVETEXTUREARB_T)
             glfwGetProcAddress( “glActiveTextureARB” );

         multitexture_supported = GL_TRUE;
     }
     else
     {
         multitexture_supported = GL_FALSE;
     }
#else
     // Extension is not supported by the include files
     multitexture_supported = GL_FALSE;
#endif
```

Please note that the code example is not 100% complete. First of all, the *GL_ARB_multitexture* extension defines many more functions than the single function that the code example defines. Secondly, checking if an extension is supported using **glfwExtensionSupported** is not enough to ensure that the corresponding functions will be valid. You also need to check if the function pointers returned by **glfwGetProcAddress** are non-*NULL* values.

6.3.1 About function pointer type definitions

To make a function pointer type definition, you need to know the function prototype. This can often be found in the extension definitions (e.g. at the *OpenGL Extension Registry*). All the functions that are defined for an extensions are listed with their C prototype definitions under the section *New Procedures and Functions* in the extension definition.

For instance, if we look at the definition of the *GL_ARB_texture_compression* extension, we find a list of new functions. One of the functions looks like this:

```
void GetCompressedTexImageARB(enum target, int lod, void *img);
```

Like in most official OpenGL documentation, all the *GL* and *gl* prefixes have been left out. In other words, the real function prototype would look like this:

```
void glGetCompressedTexImageARB(GLenum target, GLint lod, void *img);
```

All we have to do to turn this prototype definition into a function pointer type definition, is to replace the function name with *(APIENTRY * xxxx_T)*, where *xxxx* is the uppercase version of the name (according to the proposed naming convention). The keyword *APIENTRY* is needed to be compatible between different platforms. The GLFW include file glfw.h always makes sure that *APIENTRY* is properly defined, regardless of which platform the program is compiled on.

In other words, for the function **glGetCompressedTexImageARB** we get:

```
typedef void (APIENTRY * GLGETCOMPRESSEDTEXIMAGEARB_T)
             (GLenum target, GLint level, void *img);
```

7. IMAGE AND TEXTURE IMPORT

In many, if not most, OpenGL applications you want to use pre-generated 2D images for surface textures, light maps, transparency maps etc. Typically these images are stored with a standard image format in a file, which requires the program to decode and load the image(s) from file(s), which can require much work from the programmer.

To make life easier for OpenGL developers, GLFW has built-in support for loading images from files.

7.1 Texture Loading

To load a texture from a file, you can use the function **glfwLoadTexture2D**:

```
int glfwLoadTexture2D( const char *name, int flags )
```

This function reads a 2D image from a Truevision Targa format file (.TGA) with the name given by *name*, and uploads it to texture memory. It is similar to the OpenGL function **glTexImage2D**, except that the image data is read from a file instead of from main memory, and all the pixel format and data storage flags are handled automatically.

The *flags* argument can be used to control how the texture is loaded. If *flags* is zero, the origin of the texture will be the lower left corner, and only one mipmap level is loaded. If *flags* is *GLFW_ORIGIN_UL_BIT* the origin of the texture will be the upper left corner. If *flags* is *GLFW_BUILD_MIPMAPS_BIT*, all mipmap levels will be generated and uploaded to texture memory. To combine *GLFW_ORIGIN_UL_BIT* and *GLFW_BUILD_MIPMAPS_BIT*, or them togeter like this: *flags* = *GLFW_ORIGIN_UL_BIT* | *GLFW_BUILD_MIPMAPS_BIT*.

Here is an example of how to upload a texture from a file to OpenGL texture memroy, and configure the texture for trilinear interpolation (assuming an OpenGL window has been opened successfully).

```
    GLuint  texid;

    // Generate texture object for one texture
    glGenTextures( 1, &texid );

    // Bind texture object
    glBindTexture( GL_TEXTURE_2D, texid );

    // Load texture from file, and build all mipmap levels
    glfwLoadTexture2D( "mytexture.tga", GLFW_BUILD_MIPMAPS_BIT );

    // Use trilinear interpolation for minification
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                     GL_LINEAR_MIPMAP_LINEAR );

    // Use bilinear interpolation for magnification
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                     GL_LINEAR );

    // Enable texturing
    glEnable( GL_TEXTURE_2D );
```

As you can see, **glfwLoadTexture2D** is very easy to use. Since it can also automatically create mipmaps when required, it is also a very powerful function.

7.2 Image Loading

In certain cases it may be useful to be able to load an image into client memory (application memory), without directly uploading the image to OpenGL texture memory. For example, one may wish to retain a copy of the texture in local memory for future use. Another example is when the image is not to be used as a texture at all, e.g. if it is to be used as a height map.

GLFW also offers the possibility to load an image to application memory, using the **glfwReadImage** function:

```
    int glfwReadImage( const char *name, GLFWimage *img, int flags )
```

The function reads the image given by the argument *name*, and upon success stores the relevant image information and pixel data in the *GLFWimage* structure *img*. The *GLFWimage* structure is defined as:

```
typedef struct {
    int Width, Height;    // Image dimensions
    int Format;           // OpenGL pixel format
    int BytesPerPixel;    // Number of bytes per pixel
    unsigned char *Data;  // Pointer to pixel data
} GLFWimage;
```

Data points to the loaded pixel data. If the function loaded the image successfully, *GL_TRUE* is returned, otherwise *GL_FALSE* is returned.

Possible flags for the *flags* argument are *GLFW_ORIGIN_UL_BIT* and *GLFW_NO_RESCALE_BIT*. *GLFW_ORIGIN_UL_BIT* works as described for the **glfwLoadTexture2D** function. If the *GLFW_NO_RESCALE_BIT* flag is set, the image will not be rescaled to the closest larger 2^m x 2^n resolution, which is otherwise the default action for images with non-power-of-two dimenstions.

When an image that was loaded with the **glfwReadImage** function is not used anymore (e.g. when it has been uploaded to texture memory), you should use the function **glfwFreeImage** to free the allocated memory:

```
void glfwFreeImage( GLFWimage *img )
```

8. MULTI THREADING

Multi threading may not seem to be related to OpenGL, and thus it may seem to be out of the scope of GLFW to provide multi threading support. The initial intent of GLFW was to provide the very basic functionality needed to create an OpenGL application, but as GLFW grew to be a platform for portable OpenGL applications, it felt natural to include an operating system independent multi threading layer in GLFW. Hopefully this will make GLFW more attractive for advanced OpenGL developers, as well as inspire more programmers to use multi threading.

In this chapter you will learn about multi threading and how you can use GLFW to create multi threaded applications.

8.1 Why Use Multi Threading?

Multi threading is not a new technology, neither is it an advanced technology. In fact, multi threading could be found as early as 1985 in consumer computers, namely the Amiga, whose operating system implemented preemptive multi threading. During the early and mid 90’s, consumer level operating systems emerged for Intel based PCs that supported multi threading. Still, over a decade later, many programmers, especially game programmers, feel reluctant to using threading in their applications. Why?

There are probably many reasons that one can think of to avoid multi threading, but most of them are based on ignorance and myths. The foremost reason for not using multi threading is probably that it requires a new way of parallel thinking, but once accepted, threaded programming can take your program to new performance levels and solve many problematic timing and synchronization issues.

In the following sections a few key reasons for using multi threaded programming will be presented.

8.1.1 Take advantage of multi processor systems

If an application is divided into several threads that can execute concurrently, these threads will automatically execute on separate processors on a SMP (symmetric multi-processing) system. Multi processor platforms are becoming increasingly common, and the price to pay is generally not much higher than for a single processor system. A multi processor system is especially appealing if you consider that in general, for the price of the fastest processor available you can get two processors which perform only slightly worse.

To take a few examples (date: 16/10-2001):

- You get two 1.3 GHz AMD Atlons for the price of one 1.5 GHz AMD Athlon.
- You get three 1.6 GHz Pentium 4 CPUs for the price of one 2.0 GHz Pentium 4.
- You get four 1.3 GHz AMD Athlons for the price of one 2.0 GHz Pentium 4.

The problem with SMP systems is of course, that if an application is not multi threaded, only one of the available processors will actually be used. This is probably the most important aspect of multi threading. For SMP systems to be really useful, programs *must* be multi threaded.

If more programs are multi threaded, more SMP systems will be sold and the price for such systems will drop, and then more multi threaded programs will be written... Let us write multi threaded applications!

8.1.2 Avoid unnecessary waiting

In many situations, an application is placed in a wait state, waiting for a task to complete. Examples of such situations are: waiting for a file to load from disk, waiting for a vertical retrace (when using a double buffered display, such as a GLFW OpenGL window), waiting for a display to be cleared or data to be sent to the graphic card.

Some or all of these operations can be done asynchronously, if the conditions are right and the operating system supports it, but a simple and efficient way of avoiding unnecessary waits is to use multi threading. If there are several active threads in an application, a thread that was waiting for CPU time can start running as soon as another thread enters a wait state. This will speed up an application on both single and multi processor systems.

8.1.3 Improve real time performance

It is a known fact that an application becomes more responsive and exhibits less timing problems if different jobs are assigned to separate threads.

A typical example is streaming audio: when an audio buffer is empty, it has to be filled with new sound again within a limited amount of time, or strange sound loops or clicks may be the result. If a program is displaying graphics, loading files and playing audio at the same time (a typical game), it is very difficult to *guarantee* that the program will update the audio buffers in time if everything is performed in a single thread. On the other hand, if the audio buffer is updated from a separate thread, it becomes a very simple task.

8.2 How To Use Multi Threading

Threads are sometimes referred to as “lightweight processes”, which gives you a clue of what they are. In general, every program runs as a *process*, which has its own memory space and its own set of resources, such as opened files etc. As a consequence, each process is coupled with a fairly large set of state. When the processor changes the execution from one process to another process, all this state has to be changed too (this is often referred to as a *context switch*), which can be quite costly.

In contrast, a *thread* is a separate execution path *within* a process, which shares the same memory area and resources. This means that very little state has to be changed when switching execution between different threads (basically only the stack pointer and the processor registers). It also means that data exchange between threads is very simple, and there is little or no overhead in exchanging data, since program variables and data areas can be shared between threads.

Writing threaded applications may be very awkward before you get used to it, but there are a few key rules that are fairly simple to follow:

1. *ALWAYS* assure exclusive access to data that is shared between threads!
2. Make sure that threads are synchronized properly!
3. *NEVER* busy wait!

…and for OpenGL applications:

4. Calling OpenGL commands from more than one thread is asking for trouble!

In the following sections you will learn how to use the functionality of GLFW to create threads and meet these rules, and hopefully you will find that it is not very difficult to write a multi threaded application.

8.3 Creating Threads

Creating a thread in GLFW is very simple. You just call the function **glfwCreateThread**:

```
GLFWthread glfwCreateThread( GLFWthreadfun fun, void *arg )
```

The argument *fun* is a pointer to a function which will be executed by the new thread, and *arg* is an argument that is passed to the thread. **glfwCreateThread** returns a positive thread ID number if the thread was created succesfully, or a negative number if the thread could not be created.

When the thread function returns, the thread will die. In most cases, you want to know when the thread has finished. A thread can wait for another thread to die with the command **glfwWaitThread**:

```
int glfwWaitThread( GLFWthread ID, int waitmode )
```

The argument *ID* is the thread handle that was obtained when creating the thread. If *waitmode* is *GLFW_NOWAIT*, **glfwWaitThread** will return immediately with the value *GL_TRUE* if the thread died, or *GL_FALSE* if it is still alive. This can be useful if you only want to check if the thread is alive. If *waitmode* is *GLFW_WAIT*, **glfwWaitThread** will wait until the specified thread has died. Regardless of what *waitmode* is, **glfwWaitThread** will return immediately if the thread does not exist (e.g. if the thread has already died or if *ID* is an invalid thread handle).

In some situations, you may want to brutally kill a thread without waiting for it to finish. This can be done with **glfwDestroyThread**:

```
     void glfwDestroyThread( GLFWthread ID )
```

It should be noted that **glfwDestroyThread** is a very dangerous operation, which may interrupt a thread in the middle of an important operation, which can result in lost data or deadlocks (when a thread is waiting for a condition to be raised, which can never be raised). In other words, do not use this function unless you *really* have to do it, and if you *really* know what you are doing (and what the thread that you are killing is doing)!

To sum up what we have learned so far, here is an example program which will print “Hello world!” (error checking has been left out for brevity):

```
#include <stdio.h>
#include <GL/glfw.h>

void GLFWCALL HelloFun( void *arg )
{
    printf( “Hello ” );
}

int main( void )
{
    GLFWthread thread;

    glfwInit();
    thread = glfwCreateThread( HelloFun, NULL );
    glfwWaitThread( thread, GLFW_WAIT );
    printf( “world!\n” );
    glfwTerminate();

    return 0;
}
```

The program starts by initializing GLFW, as always, and then it goes on by creating a thread that will execute the function *HelloFun*. The main thread then waits for the created thread to do its work and finish. Finally the main thread prints “world!”, terminates GLFW and exits. The result is that “Hello world!” will be printed in the console window.

That was easy! Multi threading *is* easy!

You may have noticed that we have already used a simple form of thread synchronization, by waiting for the child thread to die *before* we print “world!”. If we would have placed the wait command *after* the print command, there would be no way of knowing which word would be printed first (“Hello” or “world!”). Our program would then suffer from a *race condition*, which is a term used to describe a situation where two (or more) threads are competing to complete a task first.

Later on you will learn how to do advanced thread synchronization using condition variables, which let threads wait for certain *conditions* before continuing execution.

8.4 Data Sharing Using Mutex Objects

Remember the first rule of multi threading (*Alwaus assure exclusive access to data that is shared between threads*)? In many situations you need to protect a certain data area while reading or modifying it, so that other threads do not start changing or reading the data while you are only half way through.

For instance, consider that you have a vector *vec*, and a variable *N* telling how many elements there are in the vector. What happens if thread A adds an element to the vector at the same time as thread B is removing an element from the vector? The following scenario is possible:

```
Thread A:      N ++;
Thread B:      x = vec[ N-1 ];
Thread B:      N --;
Thread A:      vec[ N-1 ] = y;
```

We have created a possible race condition. The result in this case is that thread B reads an invalid element from the vector, and thread A overwrites an already existing element. Not good!

The solution is to only let one thread have access to the vector at a time. This is done with *mutex* objects (mutex stands for mutual exclusion). The proper use of mutexes eliminates race conditions. To create a mutex object in GLFW, you use the function **glfwCreateMutex**:

```
GLFWmutex glfwCreateMutex( void )
```

glfwCreateMutex returns *NULL* if a mutex object could not be created, otherwise a mutex handle is returned. To destroy a mutex object which is no longer in use, call **glfwDestroyMutex**:

```
void glfwDestroyMutex( GLFWmutex mutex )
```

Mutex objects by themselves do not contain any useful data. They act as a lock to any arbitrary data. Any thread can lock access to the data using the function **glfwLockMutex**:

```
void glfwLockMutex( GLFWmutex mutex )
```

The argument *mutex* is the mutex handle that was obtained when creating the mutex. **glfwLockMutex** will block the calling thread until the specified mutex is available (which will be immediately, if no other thread has locked it).

Once a mutex has been locked, no other thread is allowed to lock the mutex. Only one thread at a time can get access to the mutex, and only the thread that has locked the mutex may use or manipulate the data which the mutex protects. To unlock a mutex, the thread calls **glfwUnlockMutex**:

```
void glfwUnlockMutex( GLFWmutex mutex )
```

As soon as **glfwUnlockMutex** has been called, other threads may lock it again.

Here is the scenario with the two threads trying to access the same vector again, but this time they use a mutex object (*vecmutex*):

```
Thread A:     glfwLockMutex( vecmutex );
Thread A:     N ++;
Thread B:     glfwLockMutex( vecmutex );
Thread A:     vec[ N-1 ] = y;
Thread A:     glfwUnlockMutex( vecmutex );
Thread B:     x = vec[ N-1 ];
Thread B:     N --;
Thread B:     glfwUnlockMutex( vecmutex );
```

In this example, thread A successfully obtains a lock on the mutex and directly starts modifying the vector data. Next, thread B *tries* to get a lock on the mutex, but is placed on hold since thread A has already locked the mutex. Thread A is free to continue its work, and when it is done it unlocks the mutex. *Now* thread B locks the mutex and gains exclusive access to the vector data, performs its work, and unlocks the mutex.

The race condition has been avoided, and the code performs as expected.

That was easy! Multi threading *is* easy!

8.5 Thread Synchronization Using Condition Variables

Now you know how to create threads and how to safely exchange data between threads, but there is one thing left to solve for multi threaded programs: conditional waits. Very often it is necessary for one thread to wait for a condition that will be satisfied by another thread.

For instance, a thread A may need to wait for both thread B and thread C to finish a certain task before it can continue. For starters, we can create a mutex protecting a variable holding the number of completed threads:

```
GLFWmutex mutex;
int threadsdone;
```

Now, thread B and C will lock the mutex and increase the *threadsdone* variable by one when they are done, and then unlock the mutex again. Thread A can lock the mutex and check if *threadsdone* is 2.

If we assume that *mutex* has been created successfully, the code for the three threads (A, B and C) could be the following:

```
Thread A: Wait for both thread B and C to finish.
     do
     {
         glfwLockMutex( mutex );
         done = (threadsdone == 2);
         glfwUnlockMutex( mutex );
     }
     while( !done );
```

```
Thread B and C: Tell thread A that I am done.
     glfwLockMutex( mutex );
     threadsdone ++;
     glfwUnlockMutex( mutex );
```

The problem is that when thread A discovers that thread B and C are not done, it needs to check *threadsdone* over and over again until *threadsdone* is 2. We have created a busy waiting loop! (Rule 3: Never busy wait!) Not good!

The method will work without a doubt, but thread A will consume a lot of CPU power doing nothing. What we need is a way for thread A to halt until thread B or C tells it to reevaluate the conditions again. This is exactly what condition variables do.

GLFW supports three condition variable operations: *wait, signal* and *broadcast*. One or more threads may *wait* to be woken up on a condition, and one ore more threads may *signal* or *broadcast* a condition. The difference between signal and broadcast is that broadcasting a condition wakes up all waiting threads (in an unspecified order, which is decided by task scheduling rules), while signaling a condition only wakes up one waiting thread (again, which one is unspecified).

An important property of condition variables, which separates them from other signaling objects such as events, is that only *currently* waiting threads are affected by a condition. A condition is "forgotten" as soon as it has been signaled or broadcasted. That is why a condition variable is always associated with a mutex, which protects additional condition information, such as the "done" variable construct described above.

This may all be confusing at first, but you will see that condition variables are both simple and powerful. They can be used to construct more abstract objects such as *semaphores*, *events* or *gates* (which is why GLFW does not support semaphores natively).

Before we go on by solving the busy waiting scenario, let us go through the GLFW condition variable functions. Just like for mutexes, you can create and destroy condition variable objects. The functions for doing this are:

```
     GLFWcond glfwCreateCond( void )
     void glfwDestroyCond( GLFWcond cond )
```

glfwCreateCond returns *NULL* if a condition variable object could not be created, otherwise a condition variable handle is returned. To destroy a condition variable that is no longer in use, call **glfwDestroyCond**.

To wait for a condition variable, you use **glfwWaitCond**, which has the C syntax:

```
     void glfwWaitCond( GLFWcond cond, GLFWmutex mutex,
                        double timeout )
```

When **glfwWaitCond** is called, the locked mutex specified by *mutex* will be unlocked, and the thread will be placed in a wait state until it receives the condition *cond*. As soon as the waiting thread is woken up, the mutex *mutex* will be locked again. If *timeout* is *GLFW_INFINITY*, **glfwWaitCond** will wait until the condition *cond* is received. If *timemout* is a positive time (in seconds), **glfwWaitCond** will wait until the condition *cond* is received or the specified time has passed.

To signal or broadcast a condition variable, you use the functions **glfwSignalCond** and **glfwBroadcastCond**:

```
     void glfwSignalCond( GLFWcond cond )
     void glfwBroadcastCond( GLFWcond cond )
```

glfwSignalCond will wake up one thread that is waiting for the condition *cond*. **glfwBroadcastCond** will wake up all threads that are waiting for the condition *cond*.

Now that we have the tools, let us see what we can do to resolve the busy waiting situation. First, we add a condition variable to our data set:

```
     GLFWcond  cond;
     GLFWmutex mutex;
     int threadsdone;
```

If we assume that *mutex* and *cond* have been created successfully, the code for the three threads (A, B and C) could be the following:

`Thread A: Wait for both thread B and C to finish.`

```
     glfwLockMutex( mutex );
     do
     {
      done = (threadsdone == 2);
      if( !done )
      {
            glfwWaitCond( cond, mutex, GLFW_INFINITY );
      }
     }
     while( !done );
     glfwUnlockMutex( mutex );
```

`Thread B and C: Tell thread A that I am done.`

```
     glfwLockMutex( mutex );
     threadsdone ++;
     glfwUnlockMutex( mutex );
     glfwSignalCond( cond );
```

With the addition of a condition variable (bold lines), the busy waiting loop turned into a nice condition waiting loop, and thread A no longer wastes any CPU time. Also note that the mutex locking and unlocking is moved outside of the waiting loop. This is because **glfwWaitCond** effectively performs the necessary mutex locking and unlocking for us.

That was easy! Multi threading *is* easy!

8.6 Is GLFW Thread Safe?

The current version of GLFW (v2.4) is *not* 100% thread safe. In other words, certain GLFW API functions may cause conflicts if called from different threads. To avoid conflicts, only the following GLFW API functions should be regarded as thread safe (i.e. they can be called from any thread at any time):

1. All functions that deal with threads, mutexes and condition variables (e.g. **glfwCreateThread**, **glfwLockMutex** etc).
2. Timing function **glfwSleep**.

All other GLFW API function calls should either be done from a single thread, or be synchronized with mutexes and condition variables to serialize the calls.

8.7 Conclusion

Now you know all you need to know to build advanced multi threaded programs. Of course, there is a long way to go before you feel comfortable with multi threading, and there are many pitfalls to fall into.

Hopefully this short introduction has inspired you and helped you on the way to use multi threading. Multi threading is a huge subject, and there are many articles, books and tutorials written about it. There are many things to think about when designing threaded programs, which have not been covered here. Some of them are:

- How to optimally partition a computationally heavy task into several threads?
- Which tasks of a program are suitable to put in separate threads?
- Which standard/non-standard library calls are thread safe?

A hint is that the GLFW threading interface is very closely related to the POSIX threading package (pthread), so if you find any documentation or article that deals with POSIX threads, you can easily translate it into GLFW threads.