rubik Reference Manual

Version 0.0.7

Generated by Doxygen 1.2.12

Wed Jan 23 17:53:15 2002

Contents

Chapter 1

rubik Hierarchical Index

1.1 rubik Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

rubik Compound Index

2.1 rubik Compound List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

rubik Class Documentation

3.1 commandoptions Class Reference

A class for parsing command line options.

`#include <commandoptions.h>`

Public Methods

- template<typename T> void register_option (T &par, std::string long_name, char short_name, std::string des, std::string arg_name)

  Register an option with a reference to a variable.

- void register_flag (bool &par, std::string long_name, char short_name, std::string des)

  Register a flag with a reference to a `bool` *variable.*

- template<typename T> void register_argument (T &par, std::string name, std::string des)

  Register a mandatory argument with a reference to a variable.

- void process_command_line (int argc, const char *argv[ ])

  Process the arguments, options, and flags on the command line.

- ~commandoptions ()

  Cleans dynamically allocated stuff up.

3.1.1 Detailed Description

A class for parsing command line options.

By using this class, you wont have to redo the tedious task of making a command line parser every time you need one.

The command line is parsed and checked for errors. An exception will be thrown in any of these circumstances:

- an unrecognizable option or flag was encountered.

- the argument to an option could not be parsed. This means that an exception is thrown, if the type of the argument doesn't match the type of the variable bound to the argument. This is determined by using the << operator.
- too many or too few arguments are given.
- several options are put into one group. The rule is, that there can only be one option per group, as each option requires an argument. If `-f` is an option that requires a filename (most likely a `string`), and `-n` also is an option, then the following command line will make `commandoptions` throw an exception:

  ```
  $ my_program -fn filename
  ```
- an option is used without an argument.
- a single dash is found without a trailing character. Also, if two dashes are found without the name of a long option or long flag. This means that `commandoptions` currently *doesn't* follow the GNU style, which treats everything following two consecutive dashes arguments.

Here is a little program that demonstrates the use of `commandoptions`:

We start by including the standard IO stream library (which is needed by our program and not by `commandoptions`) and the `commandoptions` header:

```
#include <iostream>
#include "commandoptions"
```

We can now start our `main` function and decleare our `commandoptions` object:

```
int main(int argc, const char *argv[]) {
  commandoptions c;
```

Our little program has a number of variables that the user can control. These variables are declared next:

```
  bool beep = false;
  int no_of_hits = 3;
  std::string boink_sound = "*boink*";
  std::string ouch_sound = "*ouch*";
  std::string victim;
```

Now that the variables have been declared and initialised, we can register them with our `commandoptions` object:

```
  c.register_flag(beep, "beep", 'b', "Turn on beeping");
  c.register_option(no_of_hits, "hits", 'h', "How many times to hit victim", "NUMBER");
  c.register_option(boink_sound, "", 's', "How hitting victim sounds", "SOUND");
  c.register_option(ouch_sound, "help-cry", '\0', "Victim response", "SOUND");
  c.register_argument(victim, "victim", "");
```

We can now try to parse the command line. Since this might fail, we use a `try` block:

```
  try {
    c.process_command_line(argc, argv);
```

If the parsing fails, a `commandoptions_error` object will be thrown, and we will end up in the `catch` block. The error-object has a `commandoptions_error::what` method that we use to get an explaination of the error:

```
  } catch (commandoptions_error &ex) {
    std::cerr << "Error: " << ex.what() << endl;
```

The program should now terminate with a non-zero exitcode to indicate that an error has occured:

```
    return 1;
  }
```

If everything went well, then we can start using the variables we declared previously. They will be updated to reflect the arguments given by the user on the command line. The rest of the code is:

```
  if (no_of_hits > 0) {
    std::cout << "Now hitting " << victim << ' '
              << no_of_hits << " times:" << std::endl;
    for (int i = 0; i < no_of_hits; ++i)
      std::cout << boink_sound << ' ' << ouch_sound << std::endl;
  }
  if (beep)
    std::cout << "*beep*" << std::endl;
}
```

3.1.2 Member Function Documentation

3.1.2.1 void commandoptions::process_command_line (int *argc*, const char * *argv*[ ])

Process the arguments, options, and flags on the command line.

You should call this method after you have registered your arguments, options, and flags with the object.

See also:
register_argument, register_option, and register_flag.

3.1.2.2 template<typename T> void commandoptions::register_argument (T & *par*, std::string *name*, std::string *des*) [inline]

Register a mandatory argument with a reference to a variable.

The variable will be updated with the value found on the command line.

Parameters:

par the variable to update. The variable is tied to the flag, and it will be updated when process_command_line is called. This is done with the help of >> which means that you can make your program accept a vector as an argument, if you have implemented the `operator>>` method in your vectorclass.

des the description of the argument. The description will be part of the help displayed when the program in invoed with the `--help` option. It will be placed after the long and short name, so it should be no longer than one line.

3.1.2.3 void commandoptions::register_flag (bool & *par*, std::string *long_name*, char *short_name*, std::string *des*) [inline]

Register a flag with a reference to a `bool` variable.

The variable will be inverted if the flag is found on the command line. This means that you should give the variable a suitable default value before you call process_command_line. If the flag is present more than one time on the command line, it will be parsed as if only occured *once*.

Parameters:

par the variable to update. The variable is tied to the flag, and it will be updated when process_command_line is called.

long_name the long name of the flag. The user can then toggle the flag by invoking the program with `--long_name`. The long name will be part of the help displayed when the user invokes the program with `--help` as an option.

short_name the short name for the flag. This is a single `char`. The user can then toggle the flag by invoking the program with `-short_name`. The short name will be part of the help displayed when the program is invoked with the `--help` option.

des the description of the flag. The description will be part of the help displayed when the program in invoed with the `--help` option. It will be placed after the long and short name, so it should be no longer than one line.

3.1.2.4 template<typename T> void commandoptions::register_option (T & *par*, std::string *long_name*, char *short_name*, std::string *des*, std::string *arg_name*) [inline]

Register an option with a reference to a variable.

The variable will be updated with the argument on the command line.

Parameters:

par the variable to update. The variable is tied to the option, and it will be updated when process_command_line is called.

long_name the long name of the option. The user can then enable the option by invoking the program with `--long_name`. The long name will be part of the help displayed when the user invokes the program with `--help` as an option.

short_name the short name for the option. This is a single `char`. The user can then enable the option by invoking the program with `-short_name`. The short name will be part of the help displayed when the program is invoked with the `--help` option.

des the description of the option. The description will be part of the help displayed when the program in invoed with the `--help` option. It will be placed after the long and short name, so it should be no longer than one line.

arg_name the name of the option. This will be part of the help displayed when the program is invoked with the `-h` or `-?` options.

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

- commandoptions.h
- commandoptions.cpp

3.2 commandoptions_error Class Reference

An exception class.

`#include <commandoptions.h>`

Public Methods

- commandoptions_error (std::string s) throw ()
  Constructs a new error.
- ~commandoptions_error () throw ()
  Does nothing?
- virtual const char * what () const throw ()
  Returns the errormessage.

3.2.1 Detailed Description

An exception class.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 commandoptions_error::commandoptions_error (std::string *s*) throw () [inline]

Constructs a new error.

Parameters:
s the errormessage.

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

- commandoptions.h

3.3 RoundedCube Class Reference

Models a cube with rounded edges.

`#include <RoundedCube.h>`

Public Types

- enum axis { **Xaxis**, **Yaxis**, **Zaxis** }

  The x-, y-, and z-axis.

Public Methods

- RoundedCube (float s_length=0.8, float r_width=0.1, int r_steps=2)

  Constructs a cube with rounded edges.

- void render ()

  Draws the cube.

- void rotate (axis a, bool direction)

  Rotates the cube 90 degrees.

Public Attributes

- bool draw_normals

  Should normals be drawn? This should only be set to `true`*, if you're debugging.*

3.3.1 Detailed Description

Models a cube with rounded edges.

The cube will be made up of six squares with rounded edges in-between. The sides will be red, green, blue, orange, yellow, and white just as the original Rubik's Cube. They will only reflect little of the specular light that shines on them, but they will reflect all the diffuse light.

The rounded edges will be black, but will have more intense highlights than the sides. This gives a nice shiny effect when the cube rotates in front of a light.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 RoundedCube::RoundedCube (float *s_length* = 0.8, float *r_width* = 0.1, int *r_steps* = 2)

Constructs a cube with rounded edges.

Parameters:

s_length the length of the sides. This is only the length of the squares. To get the total width/height/depth of the cube, you'll have to add two times *r_width* as well.

r_width the width of the roundings.

r_steps the number of steps used to do the rounded edges. The edges will consist of *r_steps* rectangles — the larger the number, the finer the edge.

3.3.3 Member Function Documentation

3.3.3.1 void RoundedCube::render ()

Draws the cube.

The cube will be drawn centered around (0, 0).

3.3.3.2 void RoundedCube::rotate (axis *a*, bool *direction*)

Rotates the cube 90 degrees.

The cube will rotate by changing the color of the sides.

Parameters:

a the axis to rotate around. The axis are local to the cube.

direction if direction is `true`, then the cube will be rotated clockwise, otherwise it will be rotated counter-clockwise.

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

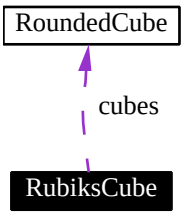
- RoundedCube.h
- RoundedCube.cpp

3.4 RubiksCube Class Reference

A Rubik's Cube.

`#include <RubiksCube.h>`

Collaboration diagram for RubiksCube:



Public Methods

- RubiksCube ()
  The constructor.
- ~RubiksCube ()
  Destructs the 27 cubes.
- void render ()
  Renders the cubes.
- void rotateX (unsigned int block, int degrees)
  Rotates the designated block around the x-axis.
- void rotateY (unsigned int block, int degrees)
  Rotates the designated block around the y-axis.
- void rotateZ (unsigned int block, int degrees)
  Rotates the designated block around the z-axis.

3.4.1 Detailed Description

A Rubik's Cube.

This class manages the 27 cubes that make up a Rubik's Cube.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 RubiksCube::RubiksCube ()

The constructor.

Memory will be allocated for the 27 cubes.

3.4.3 Member Function Documentation

3.4.3.1 void RubiksCube::render ()

Renders the cubes.

This will make the necessary calls to OpenGL to render the 27 cubes. You shouldn't call render() between calls to glBegin() ... glEnd().

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

- RubiksCube.h
- RubiksCube.cpp

Index