the zoidberg engine

Search this site

Welcome to the zoidberg engine Infosphere
Project Definition
Resources
Game Ideas
Documentation
The Team
To-Dos
Project updates
Files
Release Dates
Sitemap

136
days since
Project Due Date

Recent site activity

removed stuff
edited by Joe Balough

The Team
edited by rmb5299

removed stuff
edited by Joe Balough

cliCreator
edited by Joe Balough

Testing Framework
edited by Joe Balough

View All

Documentation >

Code Standards

This page is here to dictate how you should write your code for the zoidberg engine to ensure that it will be consistent with the existing code base.

So we don't have to go through all our code and try to figure out what everything does after the code is done, all functions and files should be documented with Doxygen styled comments detailing the function, parameters, and authors.

Contents

File Naming

Files should be named using a short all lower-case name that describes that files basic function. This will usually simply be the name of the class contained within (for example `level.cpp` and `level.h` containing the `level` class). If the file does not implement a class, it should be named something short but descriptive (like `main.cpp` or `util.h` and `util.cpp`).

Comments

There are three types of comments.

- **Doxygen Comments** provide documentation to Doxygen and are formatted like a multiline C-style comment with a second star on the first line. They should be formatted like this:

```
/**
 * This is a multiline comment to describe a function or something.
 * yay!
 *
 * @author Joe Balough
 */
```

  - Use a space to keep the stars aligned on the left. All text should be spaced off of the stars by one space character.
  - All of these comments will be put into the Doxygen documentation for the function it is contained within.
  - You can use the following Doxygen commands to specify special items:
    - `@file [filename]`
      - The name of this file
    - `@brief [description]`
      - A brief description of what you're documenting. This is usually followed by two line returns and a lengthy description of what you're documenting (see below).
    - `@see [thing]`
      - Tells Doxygen to add a link to the specified thing, can be a class, file, etc.
    - `@param [type] [name] [description]`
      - Describes a parameter to a function, should be one of these for each parameter
    - `@return [type] [description]`
      - Describes what is returned by a function
    - `@author [name]`
      - Indicates who wrote this bit of code

- **Single-line Comments** provide a quick and simple comment to the code. Code should be well commented so that anybody else can simply look at the code and understand what is going on. Use this comment style to document functions *in the source file* or in a function to describe what is happening. Use these comments for multi-line comments up to three lines long please.

```
// This is a comment that will span
// onto more than just one line.
```

- Put one space between the `//` characters and the comment text.

- **Multi-line Comments** simply provide a longer comment, these should only be used in code segments when absolutely necessary. Comments should be kept brief and thus, the Single-line comments are preferred.

```
/* this does this
 * text
 */
```

  - Use the same spacing as with the Doxygen comments.

Header Files

At the very top of any and all header files (and any source file that has no corresponding header file like main.cpp) should go a Doxygen comment describing the current file. It starts with the file name, describes the file's overall function (first in brief then in depth), and lists the authors and contributors. Here's an example for the level class header (level.h):

```
/**
 * @file level.h
 *
 * @brief The level class manages all data needed for a level to run.
 *
 * This file contains the level class. It manages all the data that would be needed
 * in order to run its associated level. The 'main game loop' is actually run from
 * this class. The level class manages the local OAM copy, passing SpriteEntries
 * to its children objects. It calls upon the assets class for parsing and loading
 * level layout and background data. When running the main loop for this level,
 * the update() function is called which will in turn update all of the objects
 * currently present in the level.
 *
 * @see object.h
 * @see asset.h
 * @author Joe Balough
 */
```

Following that goes the license notice. This tells the person looking at the code that it is licensed under the GNU GPL v.3.

```
/*
 *  Copyright (c) 2010 zoidberg engine
 *
 *  This file is part of the zoidberg engine.
 *
 *  The zoidberg engine is free software: you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation, either version 3 of the License, or
 *  (at your option) any later version.
 *
 *  The zoidberg engine is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with the zoidberg engine.  If not, see <http://www.gnu.org/licenses/>.
 */
```

Now comes the include once block. We wrap the header code in a `#ifndef ... #endif` block. The naming convention goes `[FILENAME]_[EXT]_INCLUDED`, all characters upper-case

```
#ifndef LEVEL_H_INCLUDED
#define LEVEL_H_INCLUDED

...

#endif
```

Note: *all* files end with an empty line.

Inside this `#ifndef ... #endif` block goes the following

1. Preprocessor macros and variables. These are always spelled using only upper-case letters and always start with `ZBE_[name]`

```
#define ZBE_NO_MATRICES -1
#define ZBE_NO_SPRITES -1
```

2. Preprocessor include directives. Note: If it isn't obvious why this file is being included, add a short comment explaining that file's inclusion.

```
#include <nds.h>
#include <vector>
#include "object.h" //addSprite function needs this
#include "hero.h"  //addSprite can add a hero to the mix
```

3. Any calls to tell the processor to use a specific namespace.

```
using namespace std;
```

4. Class definitions. The naming convention for classes is to use a simple descriptive name that conforms to the variable naming rules detailed below. All classes should be preceded with a Doxygen comment block explaining its function and its author.

```
/**
 * The level class
 *
 * The level class manages all the necessary objects for this level.
 *
 * @author Joe Balough
 */
class level
{
 ...
};
```

Functions and Variables

Functions and variables follow the same rules for naming.

- Use a short descriptive phrase to describe the variable / function. `noReallyLongFunctionNamesPlease();`
- Capitalization should follow a variation on camel case where the first character is lower-case.
- All functions and any variable defined to be a member of a class should also include Doxygen documentation.
  - In the header files, functions should name the function, provide a short description of what it does, a list of the parameters and what they are, and the author.

```
/**
 * initOAM function
 *
 * This function takes a freshly allocated OAMTable by reference and
 * initializes it by clearing the option bits for all the sprites and
 * setting all the matrices to be the identity matrix.
 *
 * @param OAMTable &oam
 *   Passed by reference, the OAMTable to initialize
 * @author Joe Balough
 */
void initOAM(OAMTable &oam);
```

  - In the source files, the comment can be a summary of what is in the header documentation.

```
// Takes an OAMTable and initializes it
void initOAM(OAMTable &oam)
```

  - Variables should simply provide a brief description of what they are used for. These can be in standard C++-style comments.

```
// The level's local copy of the OAM Table
OAMTable oam;
```

Block Formatting

Curly braces go on the line following the function declaration and are then followed by a new line. The closing curly brace should also be on its own line:

```
void myClass::myFunction(int param1, bool param2)
{
    //code code code
}
```

Indentation

Indentation levels should be marked with a single *tab character*. You may have to set this up in your IDE:

- In Code::Blocks -- under Settings > Editor > General Settings enable "Use TAB character"

Control functions

When inserting `if, for, while,` etc. blocks, there should be a space between the function and its options:

```
for(unsigned int i = 0; i < objects.size(); i++)
{
    //code code code
}
```

Assignment and Comparison

Put a spaces in between the variable name, operator, and value:

```
int i = 0;
```

Pointers

When defining a pointer put the star character (*) touching the variable name.

```
touchPosition *touch = NULL;
```

Function Parameters

All passed function parameters should be separated by a space after the comma.

```
dmaCopyHalfWords(SPRITE_DMA_CHANNEL, oam.oamBuffer, OAM, SPRITE_COUNT *
sizeof(SpriteEntry));
```