A guide to the sourcecode of **Danger from the Deep**

Contents

Chapter 1

About this document

What is this document about? It should give you a general overview of the sourcecode of Danger from the Deep, the general concepts and ideas. Most projects of today come with a code-related documentation (like Doxygen). This is generally a handy thing, when developing or extending code - Danger from the Deep has it too (many things are still missing). But it helps you only when you know where to look and *how* to extend the code. Even if the code would have a documentation for *every* function, you would not know where to start. Code-based documentation can be the final step to make project documentation complete, but it is not useful as first step into the project. It is often misused as excuse to not creating a more general documentation.
Many open source projects out there have a more or less well managed Doxygen-like code documentation. But what is it good for? It is of use only for those that already know how to use the project - as reference. But where to start if you are new to the project? You are lost. At that point example code may help or researching for information on the internet. Or to consult a more general documentation, if available.
For this project, only the latter option is possible (or you enter IRC/forum and ask us directly, what you should do anyway if you want to join). This document tries to solve the problem described above and tries to give interested people an overview of the project.
Of course one could enhance the Doxygen-based documentation while learning more about the project. Such a person would be very welcome.

Chapter 2

General idea

This is a hobby project. All developers do the work in their spare time, no one is paid for it. The code has grown for years (since around january 2003) and many people contributed to it. While we worked on the project, we changed many things: interfaces, implementations and algorithms. We tried many things. While developing the code, our knowledge developed as well. Thus, the style of writing code and using C++ changed as well. As a result of all these facts, you will find out that the code is not as uniform as code could be.
However, i can tell you from real life experience (working as software engineer), that this is true for the most code on this planet. Considered that fact, the source code of Danger from the Deep could be much worse. The code is made after the object oriented paradigm about software construction. Its extensible, mostly readable and understandable. Beside aiming for functionality we aim for speed too, as we are working on a game with realtime effects. For that reason we use *C++*, as it features object orientation, high execution speed and flexibility as well.
The game is written in portable code and can be compiled for many platforms (up to six working at end of 2006). The text resources inside are not hardcoded, so the game supports multiple languages (currently seven are implemented at end of 2006). Now consider that we are working on a realtime 3d game with many great visual effects, multi platform, multi language and many more features.

2.1 Joining

If *you* want to join the team, then you are welcome. We need more developers. But let me tell you one warning word first. This is not meant to offend you, but to make you think. Please read on.
People tend to transfer their view of coding to new projects, trying to adapt the project to their style. You may be a smart and brilliant coder, you may believe that your way of doing things is the best, no matter what way the project has chosen before you joined. We don't want to hinder you to start hacking on the code, extending it with more features or fixing bugs et cetera. Any help or contribution is appreciated. But…there comes the warning.
But…the point is, if you do so, check carefully what you do and how you do it. The feature you

want do implement may have been already implemented, somewhere more or less hidden in the code. The aspect you want to add may collide with some other parts of the code. The functions you add may conflict with existing interfaces. The way you want to implement a feature, may be only one alternative of many, and possible not the ideal one, and so on.
This is not meant to offend you, the contributor of new code. I write this to avoid double work. To avoid wasting your time with writing code that already exists or wasting time of others by breaking existing code. If you want to contribute, I stronlgy recommend to get a copy of the code and try to understand what is going on inside it first. One step of that process is to read this document so you are on the right path.

2.2 History

The game arised from a graphic test for OpenGL under Linux. I wanted to try out OpenGL graphics under Linux and finally wrote some tile based water rendering code. Later i added a simple model loader and had the very basics of a submarine simulation. As i am a great fan of Silent Service and Aces of the Deep (guess where the name comes from) and considered the fact that there is or was no submarine simulation for Linux, i began to create one.
The project started in January 2003 and the code evolved since that time. I improved my knowledge about C++, OpenGL and about software development in general. Other developers joined the project and added more ideas and development ideas and styles. Often things were just tried out and thrown away later. This all lead to code that seems a bit chaotic, more like evolution and less planned. There are many places where this can be seen. Interfaces that are not fully implemented, or various ways doing similar things around the code.

2.3 Used libraries

SDL, SDL_image, fftw, SDL_mixer, SDL_net

2.4 Language: C++

speed, object orientation, platform independence, opengl, flexibility (multiple inheritance)

2.4.1 About resource and memory management

I bet you have been told that there are two models of handling resources like files, memory and more: the *C*-like way, where you have to do everything on your own and the *Java*-like way, where some entity called “garbage collector” cleans up unused memory for you. The same is true for other resources, except that even such an collector could not know when to close files and you have to close them at the right time like with *C*.

The *C*-way is problematic, because you have to do every tiny bit of work for yourself and you have to do it over and over again, which is highly error prone.
On the other hand there is the clean OOP-way (object oriented programming), where memory doesn't need to be freed by the the developer, but the runtime environment does it for you. Of course that environment can't know when to close files, so you have to do this still on your own. When code can throw exceptions, you have to encapsulate file handling and close them in special code blocks (the *final* keyword in *Java*) or similar things.
In fact, there is a third way. The only language i know that offers this way is *C++*. You can do it the hard way like *C* but you should not. *C++* is not *C*. The language follows a very different paradigm. And *C++* is not like *Java*. There are critical differences. Do the things in *C++* like they should be done in this language, and not like *C* and not like *Java*. The *C++* paradigm is not that hard to learn and brings you some surprising advantages.
The key mechanism is that local variables can be objects. Local variables are local to the block they are defined in (from opening curly brace to closing curly brace). If the variable is an object of some class, the constructor of that class is called on defining the variable, and the destructor is called automatically when leaving the block. Constructors and destructors are more or less normal functions, so you can put any code in them. The object you declare does not even have to have attributes with it, its constructor and destructor is called anyway.
I hope you already see what enormous chances this offers. Automatic call of a function when leaving a block? No matter how the block is left - by either normal code execution or break or return or exception? This fits perfectly to closing a file or freeing memory.
By letting *C++* handle the local variables automatically and by attaching resources to such local variables you can make nearly *every* resource management automatic! Never ever again you lose some memory or forget to close files. Resources are automatically freed when an error occurs and the code is stopped by an exception. Stable, reliable design without extra work by the developer - doesn't that sound great?
It *is* great, but the price is language complexity and some extra work on declaring such helper classes. But the first problem vanishs once you are used to the language, and the second problem is tiny compared to the advantages you can enjoy with the method.
The *C++* standard library (STL) already makes use of such techniques and has fully automatic memory management. Full speed of *C* with the easy coding like *Java* (sometimes even easier). That is one of the reasons why we have chosen to use *C++* for the project.
The automatisms can be seen in the standard classes like `std::vector` or `std::auto_ptr`, and are also used by some Danger from the Deep classes like the `ptrset` or the `ptrvector`. The concept is used for files too, realized as "streams" in *C++*. It can be extended very well to multi-threaded programming as well, like writing a class that locks a mutex in constructor and unlocks it in its destructor. That way mutex locking can be done automatically by declaring such a locker object inside a code block. Upon leaving the block, the mutex is automatically unlocked, removing a very nasty and common source of error with multi-threaded programming. Many more applications are possible.
In short one can say: if you use `new` and `delete` to allocate memory blocks, you do something wrong.

Use vectors instead. You should only create single objects with new and should attach them directly to an object that can manage the memory automatically, so the object can not get lost, whatever happens (exceptions can break normal code flow, be careful!). Thus, you should not be forced to use `delete` too often. If you even would use `malloc` or `free` anywhere, you are very wrong and should expect harsh comments by the developers. These functions are exclusivly *C* and do not mix with the *C++* ones. As the *C++*-FAQ "lite" by Marshall Cline abbreviates it: arrays are *evil*, unmanaged pointers are *evil* - in general. Check if you could use a *reference* instead of a pointer, whereever you would like to use pointers.
I suggest reading the mentioned FAQ at http://www.parashift.com/c++-faq-lite/

2.5 Error handling

Currently the code treats errors as fatal events, reports it via message to command line and quits the game. Checks are done using the **system::myassert()** function, but this is embarassing and ugly. This mixes code test related assertions with error checks. C++ exceptions are the definite thing to use here, but are used very rarely yet. I discovered only recently how useful they are.
The whole code (that means each place with a **system::myassert() call** has to be checked wether it is an error check (in that case replaced by an descriptive exception) or a development check (in that case a normal **assert()** could be used). For exceptions one should define classes that heir from **std::exception** and contain an error string. A hierarchy of exceptions could be created for a finer grained error handling and control. Look at file **error.h** for an example definition.
The general idea about exceptions is to report *what* went wrong, not where. Exceptions are made to handle errors at runtime. If you want to know where the error was raised to ease development, add a descriptive text to the exception or print it in the log. Each exception can be given a report text that is for the user or developer. With some tricky macros you can append the line and source file name to the text as well (see file `error.h`). The set of exceptions brought by the C++ standard library is a good starting point and these exceptions already cover many possible errors, like `invalid_argument`, `runtime_error` and so on. The constructor of an exceptions gets an user defined string, where a developer can state the nature of the error.
For example you could check in some function, if a given pointer is non-zero:

```
if (!myptr)
        throw std::invalid_argument("myfunction: Null-pointer given");
```

(Of course if you expect some pointer to have a valid value, you maybe could use a reference instead). All exceptions form an inheritance tree, a hierachy. Because of that one can categorize errors and handle them regarding to category. This allows a very flexible error management. I suggest you to learn something more about C++ exceptions if you are new to the language. The C-style of error reporting via return values is *not* the way to do it. But do not confuse error handling by value reporting. If a function should do a simple thing that can fail, it would be ok to return a bool for

that function and not throw an exception in case of error. It depends on the severity and if the caller can handle that malfunction directly.
Good examples where to use exceptions are situations where errors are not expected, where normal code execution is assumed. As their name already tells, exceptions are designed to handle exceptional situations. You want to load an image and can't read the file? Then throw an exception. The exception will be e.g. passed upwards to the mission loader, which can react on it. For example it could present an error message to the user: "couldn't load mission, because of ..." or similar things.
Final note: because exceptions break normal control flow of programs, you have to write your code in a way that no memory is lost or resource is kept. If you do a thing that needs to be cleaned up when leaving the current block of code, you need to embrace the code with a try/catch clause and clean up in the catch-path. You can make your life much easier, if you use built-in types that do this automatically. For example use auto-pointers (`std::auto_ptr`) instead of plain C-pointers or streams (`std::ifstream`) instead of file-pointers. A very handy thing is to add a class that does the cleanup in its destructor (and the initialization in the constructor) and to instantiate an object of that class as local variable. The compiler then will automagically do all the clean-up work for you.

Chapter 3

Structure of the project

Class diagram:

3.1 Main files and interfaces

At the writing of this document the games has several dozen header and source files (classes). It takes some work to learn which file does what and how. As a short guide I explain the main files and interfaces here.

3.1.1 Where it all starts

Check the file `subsim.cpp`. This file contains the `main` function of the game. There game objects are created, user interfaces are initialized and so on. You need to learn about the other interfaces first to understand that file.

3.1.2 State and display

The central aspect is playing a game. The code is divided in two categories. One for handling the state of the game, that is storing all objects, their data, simulating physics, environment and so on. The other part is the presentation of that data: visualization, sound processing, user input.

3.1.3 Hierachy of state classes

The central class for a game instance is the class `game`. That class describes an instance of a game at any time and holds all other objects needed to represent a game's state. The class `game` holds all objects of the game's world like airplaines, ships, submarines, grenades, torpedoes and so on. Every object in the game that is a physical entity is of class `sea_object` or one of its heirs.
Heirs of `sea_object` are e.g. `ship`, `airplane`, `gun_shell` and so on. Heirs of `ship` are e.g. `submarine` and `torpedo`. Now you see what we need multiple inheritance for.

The game object and all incorporated objects are the data representation of the simulated word. Their contents are stored in savegames. If you do something about simulating the world or its content, you modify any object attached to the game object.

3.1.4 Hierarchy of presentation classes

On the other side, there are the classes used for user input and graphical output (well, sound and music as well). I describe the user interface here, that is merely the graphical interface.
The central class of the user interface is called so: `user_interface`. Everything needed to render the game or environment is attached to it. There are implementations depending on the type of object that the player controls: `airplane_interface`, `ship_interface` and `submarine_interface`, where at the moment only the latter is funtional. The interface object also holds classes for displaying the sky, sun, moon, the ocean environment and the environment of the player controlled vessel.
The in-game user interface is partitioned in multiple screens or displays, also known as stations. For example the various compartments of a ship or submarine: the bridge, the engine room, the map chart et cetera. The base class for such a screen is `user_display`. Every display heirs from it, and there are many.

3.2 Basic classes and types

3.2.1 Mathematical helper classes

There are some basic classes that implement various mathematical constructs as template classes, so one can instantiate them for any needed data type. Predefined names for `int`, `float` and `double` exist. These base classes are for vectors (`vector2`, `vector3` and `vector4`), matrices (`matrix4`) and quaternions (`quaternion`). Operator overloading is used so that one can write equations and formulas in a common style. The code is rather self explanatory, so have a look.
There is also a helper class for (nautical) angles: `angle`, that implements wrapping of values at 360 degrees and some other useful things.
And finally a helper class for fixed point arithmetic data types: `fixed`. Fixed point arithmetic is still faster for some cases and the precision is still high enough for some tasks related to 3D rendering (do not use them for physics though, nor `float`, but `double` for physics).

3.2.2 Other helper classes

bspline color data error filehelper fixed objcache ptrset

3.2.3 Rendering and system related related classes

model.h sound.h texture.h font.h network.h color.h image.h system.h

3.2.4 Components of the game state and simulation

ai.h airplane.h game.h torpedo.h submarine.h gun$_shell.hconvoy.hsea_object.hsensors.hship.hdepth_charge.h$

3.2.5 Components of the graphical user interface

sub$_bridge_display.hairplane_interface.hfreeview_display.hsub_control_popup.hsub_damage_display.hsub_gauges_dis$

3.2.6 Special 3D rendering related classes

ocean$_wave_generator.hparticle.htriangulate.hperlinnoise.hcoastmap.hsky.hwater.hwater_sse.hmake_mesh.h$

3.2.7 OpenGL GUI classes

widget.h

3.2.8 Miscellaneous classes

tokencodes.h token.h tokenizer.h binstream.h gldebug.h parser.h global$_data.hcfg.hhighscorelist.hkeys.htexts$

3.3 The game: state and display, the classes `game` and `user_interfa`

The code (functionality, data structures and classes) of the game is partitioned in two parts: one managing the state and the other managing the display or user interface. The central class for the state data is the class `game` with all its dependant classes (`sea_object` and its heirs and so on). All data you need to describe a game's state is stored there. If you want to save games or do network play, all data you have to access is hold by that class.
On the other side there is the user interface. Its heirs and associated classes manage everything from state presentation (visual and acustical - graphic and sound) to user interaction (input). This part requests data from the `game` class for presentation. Parts of the user interface for example are the classes for the stations, the environmental rendering (classes `water`, `sky`, `moon`, `user_display` and many more).
To simulate the game one only needs to change the class `game` and its corelated data. This class knows nothing about the user interface. Because of that you can use one instance of the `user_interface` class with any instance of class `game` without the need to reconstruct the user interface, which is a costly process. You can instead simply exchange the game object, as it is done for loading games or as it would be needed for network play or switching missions.
Note that at the moment there is a reference to `user_interface` in class `game`, that is needed for simulation. The main loop of the game is currently in `game::run()`, which needs to know the user interface instance. However it is planned to resolve that situation and move the main loop

code outside of class `game`. Then we have the real partition between two parts, as described above. However if you plan to add features, always keep this partition in mind.
And why do we do that? Doesn't it make thing more complicated? No, it isn't a real limitation, but on the other hand splits code in two clean domains. Physical simulation and rendering are two independent parts. It would be a lot more complex if state simulation would interact with the user interface in every possible situation.
Note that events causing feedback to the user (like playing an explosion sound after a collision) need some extra work. You can't simply call an "play explosion" function of the user interface (where the sound is played) from class `game` (where you detect the collision). Instead you have to remember such events in class game and let the user interface request the events and react on them.
This partitioning brings problems as you see, but also has its advantages. You can simulate the game without need to render it. This allows dedicated servers. Or it allows to exchange the renderer or many more things.

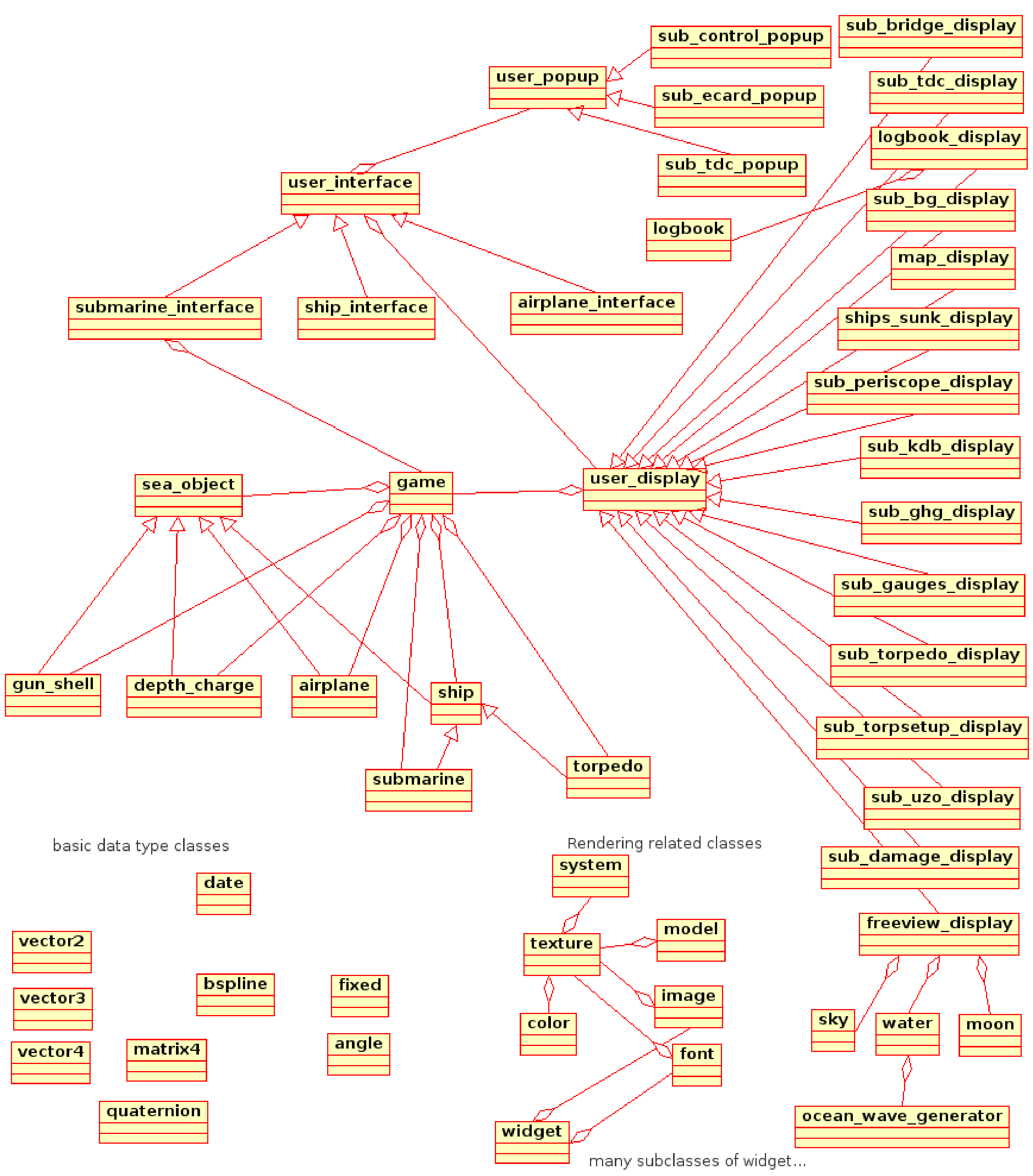

Figure 3.1: Class diagram of Danger from the Deep

Chapter 4

Coding style

4.1 Introduction

This is a topic that can cause more heat than light when discussed. Everyone has and likes his own style. Anyway, this here is the style we use in the source code of Danger from the Deep. You have been warned.
It is desireable to use *one* style throughout the whole code to enhance readability. But creating working code is of course more important than keeping style consistent at any cost. If you create new code and have the choice, you should use the game's style to enhance readability though. Code that was contributed by other people may have a different style, this is the case in some parts of current code.
This style originated from the ideas of the coding style of the Linux Kernel (originally written by Linus Torvalds). Although everyone likes his own style and mostly there are arguments pro or contra a certain style, this one is reasonable. See below.

4.2 Indentation

People differ on how many spaces to indent. In my opinion everything less than four is a pain. I use eight, because this will really help one to see **how** indentation works (and even after sitting many hours in front of the monitor). The simple solution to this problem is: use **TABs** for indentation. Everyone can set his favorite tab width in his editor then. There is another reason: indenting that much prevents you from nesting your code to deep, which is a good help to avoid too complex code.

4.3 Placing (curly) braces

Yes, the old C discussion. I place braces at the end of the line of the command. Want an example? here it comes:

Conditional commands:

```
if (x == 3) {
        do_something();
} else if (y == 4) {
        do_other();
} else {
        do_another();
}

switch (a) {
case 3:
        do_abc();
        break;
case 4:
        do_efg();
}
```

Various loops:

```
for (unsigned i = 0; i < 3; ++i) {
        b += z;
}

do {
    do_nothing();
} while (z == 5);

while (true) {
        do_something();
}
```

Functions and classes:

```
void foo(int i)
{
        do_func_code();
}

class xyz : public abc
{
 private:
        int a;
 public:
        xyz();
};
```

Why this style? Kernighan and Ritchie used it and when XEmacs is set to this style it produces this kind of style and its auto-formatting works best with it. Seriously, this style leads to code that wastes less space (especially saves lines!) without losing readability. I've seen much code that places each brace in its own line at the cost that the code is much longer. The longer it gets, the less fits on one screen and one needs to scroll, which makes it more difficult to follow the execution path of the code and to understand it.

4.4 Naming

I prefer using lowerspace characters and underscores. Why? humans are more trained to read lowerspace characters and underscores are a better optical partition than mixing lowerspace and upperspace characters. Example: read this fast and see what is easier. Lowerspace **this_variable_has_a_very_lo**
Upperspace **thisVariableHasAVeryLongName**. Horrible. Looks crowded to me. If you write code for Danger from the Deep, please follow the rules and use underscores and not mixed-case names.
The same goes for classes, members and functions. All should be in lowerspace form. Methods that return some data from a class should be named **get_xyz()** when **xyz** is the name of the member. This is no hard rule, but it should be followed for consistency.
The most important thing, and the only rule i urge you to really follow is to **not use hungarian notation** nor anything similar. This is a very weird thing some big, greedy software company introduced and makes code **hard to read**. The compiler will check the type of the variables anyway, and if you state that you would need this kind of notation to show what your variables are or do, you did something wrong before. If you can't tell from the name of a variable what it does, then this is the real problem. Adding the type to the name doesn't fix it.
Cause of the real problem: maybe you also have too many variables so that you want to add the type to distinguish between them? Another problem is the reason: don't make functions or classes too long or too complicated!

4.5 Closing words

These rules are no strict "you must follow them" rules. I prefer working code is added to the game rather than keeping the style conform. But it can help to read and understand the code, so please try to keep the style.

Chapter 5

What needs to be done

Beside the official “To Do” lists or fixing reported bugs, what needs to be done about the sourcecode?

- Add Doxygen documentation for important interfaces, classes and functions
- Check for readability of code and comments
- Check every place where a “fixme” was strayed out if it is still valid or can be fixed easily
- Clean up the code - remove obvious bugs or resolve badly readable code
- Extend this codeguide with growing knowledge

Most of these topics are valid for any software project around, but can be an easy starting point for anyone who wants to join the development team. The work can be a bit boring though.

Chapter 6

Water rendering

6.1 Introduction

As you have read earlier, the project raised around a water rendering test with OpenGL. Since that time many monthes have passed and many lines of code or comments have been written and also scrapped about that topic.
It is not only important to give you general information about the code and the project, but also to archive experience about certain topics. We have researched some aspects or experimented around them. This is true for historical technical stuff like sonar, torpedoes, submarine and ship types, convoy routes and tactics and many more things. Some documentation is available externally.
We also researched and experimented about certain techniques of rendering the environment you can see in Danger from the Deep. Is is of some value to centrally store the gained knowledge. This chapter is about water rendering, a very interesting topic, and one of the most important rendering topics around a submarine simulation.

6.2 Requirements

The most common scenario for Danger from the Deep are actions on the open ocean. So we need to render a realistic ocean environment with all types of weather. This includes:

- Ocean surface with realistic form (waves)
- Correct rendering of water color
- Extra effects of water: foam, spray, reflections
- Interaction of water with objects or coast
- Influence of water surface to ship movement (tide, rolling)

- Underwater renderering for outside views or torpedo-cameras

At the moment not all of these topics have been implemented and some are more important than others. What is most important is the rendering of a realistically shaped surface of water with correct lighting. This is already implemented rather well, but many improvements have been planned by various developers and will be implemented later.

6.3 Shape of ocean

Ocean waves show a certain pattern, that looks like it is repeating or self-similar. The surface is rather rough and very detailed. It varies from small waves in the near to large waves that can be seen in the distance of many kilometers up to the horizon. We have to compute 3D shape data of the ocean surface that is realistic and on the other hand fits to memory. Ocean waves can be represented by a height field if we leave out breaking waves, were wave tops collapse. So for every point $P = (x, y)$ on the ocean surface we have exactly one height value $h = f(x, y)$ where f is a function that can compute these height values. The task is now to compute f or find a good approximation.
There are many possible ways: simple plasma-like noise (Perlin-Noise), fractals, statistical data et cetera. To compute any function f in realtime we will certainly need to limit the area in the XY-plane where f can generate values for. The obvious idea is to create a function that generates tileable data, that is, $f(x, y) = f(x', y')$ where $x = x'$ modulo N and $y = y'$ modulo N for some value N. This is true for most noise functions.
We tried out some alternatives, but what gives the best result by far is a statistically based model that computes random frequencies with a gauss distribution and converts these to the spatial domain via a Fourier transform. This is the famous model presented in the paper by Jerry Tessendorf about ocean rendering and referenced hier as the FFT-approach.
We use a function to generate tileable height values of a certain resolution $N * N$ where N is a power of 2 for computational simplicity and because of the constraints of the model. A tile of size $M * M$ in square meters is generated. Detail per meter is of course $D := N/M$. Larger numbers of N cause more detailed waves, but consume enormous amounts of memory when precomputed. Larger values of M are important to create realistic waves, but lead to a bigger N needed. Since waves are changing shape we need to compute an animation over time. If we want to precompute the height values, we need a function that is not only tileable in the spatial domain but also in the temporal domain.
This can be done with the FFT-approach also very easily, but leads to even more memory consumption. Today's computers have enough memory though. Some feasible values are $N = 128$, $M = 256$ and $A = 256$ where A is the number of animation frames. We animate the waves with 25 frames per second. Although waves are represented as height field, much better results can be achieved, when the waves are not rendered with regular grid in XY-plane but with certain displacements.
Thus we need 3 coefficients for each wave data sample: (x, y, z). We use IEEE 754 float values for the coefficients, so the memory consumption is: $N * N * A * 3 * 4$ bytes (each float value has 4 bytes). This gives for the values mentioned above: $2^7 * 2^7 * 2^8 * 3 * 2^2 = 2^{(7+7+8+2)} * 3 = 2^{24} * 3 = 48$ megabytes. That is fully acceptable for today's computers.

There are many possible ways to spare some memory, like computing less frames per second and interpolating in temporal domain or by computing the animation in real-time. The latter idea has the advantage that waves don't need to loop over time, leading to a more realistical appearance. The disadvantage is that the computation needs critical amounts of CPU time.

6.4 Rendering of the geometry

As we have seen, water has incredible detail from close waves to the horizon. It just doesn't fit to the approach of rendering flat triangles. We would need to render many, many triangles per frame. But modern video cards can do this. And there are tricks like simulating geometry (normal mapping). But we need to build triangles of the height data and render them on the screen. For any near-realistic approach we need many triangles, certainly more than 10,000 per frame.
Now there are several ways how to generate triangles from the height data and how. We tried out many different things, but two are of most interest. The first techique is called "projected grid" and is currently in use. It's benefit is that we have triangles with similar area all around the screen, no matter in which direction we look or in what angle. Near waves show same detail per screen area than far waves. The disadvantages are that the computation of the triangle's vertices are very costly. The second technique is called "geoclipmap"-rendering and is a modification of the so called geoclipmap-rendering algorithm. It is a level-of-detail approach, where several levels with fix triangle sizes are rendered from close to far view. Computations are much easier, but triangle areas do vary more and we need more triangles per frame to get similar results.
The first technique is currently implemented and works rather well, but has reached its limits. Also some minor bugs remain.
The second technique can do more and can be extended to render more surface detail on faster video cards. It can be supported by the video card better. On future cards it can be computed fully on the video card. Research in that area goes on, a test implementation is also done and available in CVS.

6.5 Lighting and color

6.5.1 Normal mapping

6.6 Surface effects

foam, spray

6.7 Interaction

tide, rolling, coast.