3D Model Binary Vision System

Coding Standards and Quality Document

Flap_Jacks

Rani Arraf
Quinn du Piesanie
Jacobus Janse van Rensburg
Steven Visser
Marcus Werren

Contents

1 Introduction

This is the Coding Standards and Quality document for Flap_Jacks. Flap_Jacks is one of the teams belonging to the Software Engineering module at the University of Pretoria. Flap_Jacks has been tasked with engineering a system that can be used at a dental practice, to keep track of patients, their bookings and their consultations. The core aspect of this project however, is that the system is able to generate a 3D model of the patients teeth in real time display this on the system.

This document has been written in order to try and maintain the quality of the code that is used on this project, making it readable, and to give it a high usability & maintainability for new developers of the system. If a program is written well, while following this standards document, errors will be less likely to occur, and those that do will be easy to trace correct.

2 Coding Standards

2.1 Naming Conventions

All files should be named to describe exactly what it will be used for. For example, the file that holds the database models should be called 'Models.js'

All Classes and object classes should be named to represent what they are designed after or designed to store, for example, the class that stores all of the information about a car should be called CarDetails.

All methods should be named after the function that they were implemented to do. For example, a function that changes the details of an object should be called UpdateObject.

All Variables should be meaningful. This means that they must describe what the variable will hold. For example, a variable holding the todays date will be called currentDate.

2.2 Indentation

Indentation is very important when it comes to the readability of your code. Indentation can be used for many things, however, we use it in this project for showing the scope of functions, classes, conditional (if-else) statements and loop blocks. The main idea behind the indentation here is to show the scope of a code block, while keeping the code neat and readable at the same time.

When indenting, it is common practice to use either 4 spaces or a tab. This must be kept consistent throughout the program once it has been decided on.

Below is an example of what proper indentation of a conditional statement should look like:

```
if(indentation == 'tab')
{
        return "This line is indented correctly";
}
else
{
return "This line is not indented";
}
```

2.3 Comments

We shall use inline comments as a way to describe what is happening at key points within a function. Where possible, comments will be used on the line above

2.4 Braces

When coding, we use braces to format the code, as another measure to keep it clean, orderly and readable. Every developer uses braces differently, however we have to keep consistent within the same program. For this, we have decided that all braces should be on their own line, as shown below:

```
if( )
{

}
```

Also, it is known that many programmers prefer to neglect the use of braces when it comes to single line conditional statements, however we have opted to always use braces, as this keeps the code consistent and readable.

2.5 Spacing

Ideally, to keep the code as readable as possible, we would like to use a space between all mathematical operators (=, +, -, *, /, %) and preceding and succeeding text.

When writing conditional statements, we always put the else of an if-else on its own line to keep the separation of code clear.

2.6 Variables

We do not allow multiple variables to be declared on the same line, as this can cause clutter and confusion. Even though it bulks up the code, each variable should be declared on a new line.

2.7 Statements

In order to keep a uniform and clean approach to programming, keeping a the limit of the number of programming statements per line as 1. This is to keep the code from getting cluttered and making it hard to read.

Example:

This is wrong:
```
app.route('/getDoctorsBookings').post(Model.getDoctorsBookings);
```

This is correct:
```
app.route('/getDoctorsBookings')
        .post(Model.getDoctorsBookings);
```

2.8 Methods & Classes

Before the declaration of each method, function or class, there must be a block comment that says the author of the function and the functions purpose.

To make it easy to read the file and to easily distinguish the beginning and end of a function or class, a commented row of '=' equal signs must be used in between every declaration.

Example:

```
//========================================
//Function developed by: Timmy
//This function will do nothing
function nothing()
{

}
//========================================
```

2.9 Files

At the start of each source file, the following details should always be provided by the author in a block comment:

- The authors name

- The purpose of the file as a whole
- The purpose of the functions within the file

Example:

```
//File created by: Timmy
//This file was created to hold all of the functions that control the API
//The functions allow the creation, deletion, modification of database entries
```

2.10 HTML Development

As we all know, HTML source files can very easily become disorganised and messy. As a way to keep the file readable, we will make use of indentation. All Elements children elements should be indented one level further than their parent, and there should be a blank line between all element tags. Also, all tags should be on their own lines.

Example:

```
<body>
        <div>
            <button>Click me! </button>
        </div>
</body>
```

3 Github Repository

Click here to view our repository.

Our Github repository follows a very strict separation of concerns guideline.At the top level, we have Application and documentation folders and a README File.

3.1 Application

This directory contains all of the code that is used to run the application. All code is separated into to their own directories, including the API for the backend and the web application that is viewed in the browser.

3.1.1 API

The API directory contains 3 more sub-directories. These are the Controller, Model and route. These are used to contain separate parts of the API that function together to build most of the backend of the system.

3.1.2 Website

This directory is split into 5 directories. These include CSS, HTML, JS, Logs and Rendering. The code in these files are used to build the front end that the user will view in the front end on the browser.

3.1.3 sfmAlogorithm

This contains all of the code that pertains to the generation algorithm for the system. All images and outputs that are generated will be stored in this directory, along with all of the code that is used to execute the algorithm.

3.1.4 Test

Tihs contains all of the unit and integration tests for our application. ALl tests are executed from here and this is where the results will be stored.

3.2 Documentation

This directory contains all of the documentation that was created for the 3D Binary Vision System project. This includes following:

- User Manual
- Coding Standards
- Technical Installation Manual
- System Requirements Specification
- Testing Policy Document

3.3 Read Me!

This is a simple .md file that contains all of the information about our repository and the developers who worked on the project. It contains links to all of the documentation that pertains to this project as well as the github.io pages of each developer. Finally it contains the contributions of each developer.