How the structure and design of my game can assist in maintenance and capacity for extension (D2)

When creating a game it is important to fully design how the game will function before starting development. Rushing into development will mean that the base code of the game will be substandard and will affect the performance and ease of future development. In this document I will be detailing how to design and structure a games code, while showing how I have done this in my code.

Comments

A comment is a visual way to organise the code into sections, describe a complex line of codes functionality and explain/justify why you have used that code for that function. These comments have absolutely zero influence on the finished program and can only be seen in the code, this is because the compiler or interpreter knows to ignore the comments.

While comments are mostly useful, overused they can make code messy and needlessly long. They should only be used to explain the function of very complex code, rather than simple functions that the reader should already be able understand just from the code itself. For example explaining that “A + B = C” is “adding A to B which will output C” is unnecessary.

But one thing the reader won’t easily be able to determine is why you choose to solve the problem using that specific method.

Comments are especially helpful when large teams are all working on the same code. They help to explain the needed functionality, and to plan out where certain sections of code will be placed. Comments also help when you need to track down a specific bug stored in a long piece of code because comments will help categorise the code better. However this can be done more effectively by using modularisation, which I will discuss later.

They are also effective when going back to the code after several months, by refreshing your memory of how parts of the code work. This makes it very helpful when revisiting the code to make extensions to the game.

Parts of work derived from my Unit 6 Software Development work.
Since not much of my code needed advanced explanations and I was the sole developer, I didn’t comment too often during the development of my game. I mainly used it to organise very long files, such as my initialisation script (Figure 1).

**Variable Names & Naming Conventions**

Variable names might seem not at all important, since some are only used once. However a well named variable can save a lot time when referring to/calling it, save you from having to use comments to explain what it is and just generally improve the readability of your code.

One of the most important factors when creating a variable name is that it describes what the variable is storing and that it is specific (not too broad), for example calling a variable that stores character names “CharacterNames” rather than just “Names”. Another important thing to consider when naming variables is the length, because you might have to retype this name many times throughout your code, but length is still a secondary concern and a variable name should not sacrifice its descriptiveness for shortness.

A naming convention is just a single style of naming files and then not changing it during the code. When working on a project with other people it is important to set a convention for everyone to follow, otherwise the code will become fractured and harder to read. An example of this is having character information all laid out in variables with the same prefix, such as “CharacterHitPoints”, “CharacterNames” and “CharacterAttacks”. There is also what style you will use to write variables, such as CamelCase “ExampleVariable” where every word is capitalised, you could also put underscores in between words “Example_variable” or a combination of the two”.

During the development of my game I have followed strict naming conventions by putting a prefix before all assets used in my game. For example all my sprites are prefixed with “spr_”, objects with “obj_” and so on... (Figure 2)

I have also tried to name all my variables and assets with sensible and descriptive names (Figure 1).

Parts of work derived from my Unit 6 Software Development work.
Indenting & Layout

The layout of the code is vital in improving its readability and for making future extensions easier to integrate into the existing code. It also helps a coder spot simple mistakes such as missing brackets or semicolons, that could be easily missed if your code looked like the code in the bottom of this image. (Figure 3)

The main reason the code looks cleaner at the top is due to indenting. Indenting should be used when a piece of code is a decedent of another. Such as the code contained within the { brackets of an if statement, for example:

```
if (1 + 1 = 2) {
    Indented Code
    if (2 + 2 = 4) {
        Further Indented Code
    }
}
None Indented Code
```

There are many different styles of indenting and how to layout brackets, above I demonstrated the 1TBS (One True Bracing Style) which is the style my code is written in. It is important that the same style is maintained throughout the code, otherwise the readability of the code will be affected.

I have maintained the 1TBS throughout my code and made sure it is indented correctly. (Figure 4)

```
if (argument0 == 1) {
    var Object_Name = object_get_name(object_index);

    switch (Object_Name) {
    case "obj_Lightning_Tower":
        obj_Control.Selected_Range = string(global.Lightning_Tower_Range);
        obj_Control.Selected_Speed = "1.5s";
        obj_Control.Selected_Damage = string(global.Lightning_Projectile_Damage) + " HP";
        break;
    case "obj_Earthquake_Tower":
        obj_Control.Selected_Range = string(global.Earthquake_Tower_Range);
        obj_Control.Selected_Speed = "1s";
        obj_Control.Selected_Damage = string(global.Earthquake_Projectile_Damage) + " HP";
        break;
    }
} else if (argument0 == 0) {
    obj_Control.Selected_Range = "N/A";
    obj_Control.Selected_Speed = "N/A";
    obj_Control.Selected_Damage = "N/A";
}
```

Parts of work derived from my Unit 6 Software Development work.
Big game studios with massive codebases also have to use indenting to make their mountains of code readable. For example here is some code from Sid Meier’s Civilization V: (Figure 5)

```lua
Figure 5

Modularisation

On a basic level, modularisation is the process of cutting up code into smaller individual modules. This can involve being put into separate files or in different functions in one piece of code. There is generally no improvement to the performance of how the code runs, there is actually probably a slight decrease because of the time it takes to open another file. (Although this would be a few milliseconds, so not a big concern) So the main advantage of this process is the productivity gains from splitting the code up into more manageable chunks.

Modularisation is similar to OOP (Object Orientated Programming”, however there are a few main differences. The main one being that OOP uses classes/objects to split things up, whereas modularisation uses methods/functions. OOP could be considered a form of modularisation, although there is not a clear consensus on this fact.

One of the main benefits of modularising the code is that if a problem occurs, it should be a lot easier to narrow down where the problem occurred. For example if there is an error on level 2 with paths, it is pretty obvious that the error is most likely relating the Level 2 path module. Without modular code you would then have to locate the piece of code responsible on line 6592/10000, whereas with modularisation you load up the level 2 path file and you have significantly lowered the amount of code you need to search through.

Parts of work derived from my Unit 6 Software Development work.
Another benefit is when working in teams multiple programmers can work on the code at the same time. With one file there is a lot more danger of corruption or accidental deletion if several people are trying to edit the file. However with modularisation all the modules can be their own files, so the corruption problem is not an issue.

The last major benefit is when it comes time to expand the game, there is a reduced chance of breaking the current version. This is because if new content is being created, it will be stored in its own new modules. While these new modules can invoke/call the already existing modules, they shouldn’t be able to break them since they are self-contained (can tamper with the actual code of the modules).

My game is developed in GameMaker, which is modular by design. All objects, sounds, sprites, etc... are all contained in their own files/modules. This makes it very easy to debug a problem when it occurs, as GameMaker’s error message tells me which module is responsible. I have also used scripts for pieces of code that are going to be used more than once by different modules. (Figure 2)

For most triple “A” game titles where the code base can be more than 100,000+ lines of code, modularisation is vital. An example of one of these games is Sid Meier’s Civilization V, however it is hard to show an example of the modularisation in this game, because most of the core code in the game is encrypted. However, all the texture files and assets are not encrypted, so the modularisation can be seen. (Figure 6)