Program a Game Engine from Scratch

Mark Claypool

Chapter 5 - Debugging

This document is part of the book "Dragonfly – Program a Game Engine from Scratch", (Version 9.0). Information online at: `http://dragonfly.wpi.edu/book/`

Chapter 5

Debugging

This chapter provides general guidelines to help in debugging (Section 5.1) as well as lists common bugs that are encountered when developing Dragonfly (Section 5.2).

5.1 Debugging Help

Debugging is a methodical process for removing mistakes in a program. The topic of debugging is so important that there are whole sets of tools to help out. These tools, often called "debuggers", can help the programmer trace code, print out intermediate values of variables, and profile performance, among other things. Many development environments are especially designed to both write and debug code. Examples of these *integrated development environments* (or *IDEs*) include *Eclipse* and Microsoft *Visual Studio*. Knowing how to use a good debugger, whether inside an IDE or externally, can be extremely helpful for finding bugs.

However, despite the presence of powerful debugging tools, debugging can still be frustrating for all programmers. Beginners often do not know how to proceed when there is a mistake in the code. Even advanced programmers can get "stuck" when facing a particularly nasty bug. Part of the frustration in debugging stems from not knowing exactly how long it takes to find and fix a programming error. While a good programmer may know how long it takes to find and fix a bug on average – this can be useful for coming up with an estimated time to fix all known bugs – the variance in the average time can be high. This means the time to find and fix a specific bug (maybe the one facing the programmer right now!) may take a lot more time than the time to find and fix an average bug. Compound that with the fact that sometimes a fix to a bug will cause a "ripple" effect, other bugs appearing in code that was previously working, and there is a lot of potential for frustration.

Despite these drawbacks, there are methodologies, as well as tips and guidelines that can help manage programming bugs. An approach to debugging is provided in Section 5.1.1, presented as a five-step debugging process. Certain methods can even help prevent bugs in the first place, as described in Section 5.1.2. Lastly, some tips and guidelines can provide an approach to debugging, given in Section 5.1.3. While this book is designed for building a game engine, most of the debugging advice provided holds for debugging game engines as well as game code.

5.1.1 Five-step Debugging Process

Step 1: Reproduce Problem Consistently. The first step to debugging is to find cases where the bug *always* occurs. A vague problem like "sometimes the engine will crash after the boss is killed" may indicate there is a problem, but does not help much towards finding a solution. Steps should be identified that always produce the bug. For example, to reproduce the engine crash: 1) start on the boss level, 2) use rockets for all attacks, and 3) kill the boss while in armor mode. This provides a systematic way to reproduce the problem. Even better if the problem can be reproduced with user (e.g., game player) input.

Step 2: Collect Clues. Once the bug can be reproduced, information on where the bug is in the code needs to gathered. These "clues" suggest where the problem might be, but they do not necessarily point out the problem. For example, a clue may be that the game engine crashes when a projectile goes outside the game world boundary. This suggests the out of bounds event may not be handled properly by the projectile. However, some clues may be false. For example, the out of bounds event may be handled just fine by the projectile, but the real problem is the movement of the projectile when it is not within the game world boundaries.

Step 3: Pinpoint Error. Determining where, exactly, the bug is can be much of the challenge, especially in a large body of code spread over many files. Code inspection should not be undertaken too early – instead determining where, exactly, the bug occurs should be pursued first. A divide-and-conquer approach can be used, whereby large sections of execution can be eliminated from consideration. For example, suppose a game engine crashes in an unknown location, but a clue is when the experience points of a unit become negative. A message can print out the value of the experience points, say, half-way through the game. If the experience points are still positive, that means the bug is after the current block of code and the previous code execution does not need to be looked at further. Logfile messages can be valuable at this stage, providing details on execution of the game and engine as it progresses as well as values of variables and object attributes. With a debugger, breakpoints can be set at select locations to check values at key parts in the code.

Step 4: Repair Problem. The exact solution to the bug depends upon the problem. Late in development, changing fundamental data structures may not be a good idea. For example, if a static array used by the engine is hitting limits, changing the array to a linked list may not be wise since a lot of the code might refer to the array, perhaps even individual elements – all code that would need to be changed. The seemingly simple change (an array to a linked list) may cause "ripple" effects, with bugs then springing up elsewhere. Ideally, any fix to the code would be done by the original coder who likely has the most insights into the original design rationale for the implementation. In fixing the bug, considering other, potential bugs is wise. For example, if a bug occurs when a missile projectile leaves the game world, it may be that a similar bug occurs when an arrow projectile leaves the game world.

Step 5: Test Solution. After a repair, the solution should be tested to ensure the problem is really fixed. This step may seem obvious, but sometimes a programmer can be sure they have a fix, but in fact be wrong, either because the error was not pinpointed correctly or the solution has bugs of its own. If possible, an independent tester should verify that the solution works. In addition, not only should the repaired code be tested, but also code that uses the repaired code as there may be "ripple" effects, causing bugs to appear where there were none previously. For example, if a solution to an out of memory bug is to make static arrays smaller, this may cause other parts of the code to stop working when they worked fine with the previous, large, memory allocations.

5.1.2 Prevention

While debugging – both the art and the science of debugging – is a necessary and important skill for any programmer, it is equally important to take steps to prevent bugs in the first place.

A consistent coding style and variable naming help with structure and readability, especially if coupled with a consistent indentation of code blocks. Variable names that are descriptive (e.g., `mouse_x` instead of just `x`) is also useful. Because of their unique challenges, prefacing pointers with `p_` (e.g., `Object *p_object`) is recommended.

Comments should be used to explain more complicated aspects of code, both for the original programmer and for following programmers that may be re-factoring the code or fixing bugs.

All variables should be initialized. While many compilers warn if an initialized variable is being referenced, depending upon where in the code this happens (e.g., via a pointer), it may not be detected. A good practice is to initialize all variables when declared and to initialize all attributes of an object in the constructor.

Adding infrastructure and even tools to help verify working code can help prevent bugs, or at least detect them very early. For example, spending the time developing a logfile manager (Section 4.3.5 on page 59) provides a valuable tool for displaying attributes. Other helpful indicators may be visual, showing on the game objects (e.g., collision "boxes" around Objects with large Sprites) or on the game screen. These indicators can be removed once a game (or engine) is finished.

Avoiding replicating identical code can make it easier to fix bugs and prevent them from persisting. For example, an engine programmer may have code that removes Objects from a list, and then replicate that code to remove ViewObjects from a list. If there is a bug in the Object list code, and it is fixed, it will still persist in the ViewObject list code. Isolating the code that is common to both in a function or method is preferable.

5.1.3 Debugging Tips

Fix one thing at a time. After adding some code and running a test case, there may be multiple bugs revealed. While it is tempting to try to fix more than one at a time, doing so, and still having bugs, makes it difficult to verify that any progress has been made. Instead, fixing one bug (and testing to make sure it is fixed!), then proceeding to other bugs, is advised.

Start with a simple case that works. Then, add more complexity, one step at a time, until the bug occurs. This will provide the simplest case for reproducing the bug and make it easier to proceed with finding and fixing it.

Question assumptions. Do not necessarily assume that even "mature" code works. Test cases done earlier may not have been thorough or were not operated under stress where timing and memory resources were vulnerable.

Minimize randomness. A bug revealing itself may depend upon a random number, say, that determines which way a bad guy moves. Removing this randomness, by fixing the seed or even hard coding a choice, will make the bug easier to find.

Break complex calculations into steps. It may be tempting to put a complex, mathematical equation into one line (and C++ can allow for some *very* complicated, single-line statements), but separating out the components into a single line each (with multiple variables, if needed) can help reveal bugs (and trace code). This holds for other coding shortcuts.

Check boundary conditions. The bulk of code doing a loop or accessing an array may work just fine, but the boundary conditions – at the beginning or the end – sometimes are the problem. These can be visually inspected, sometimes tracing a small example by hand, in order to fix.

Use a debugger. Debuggers are tools specifically designed to help find bugs. Every good programmer should be adept at one (or more) debuggers and make use of a debugger at appropriate times during development.

Check code that was most recently changed. While a bug may appear most anywhere during development, often the culprit is code that was most recently added. This holds even when the bug appears to be elsewhere.

Take a break. Sometimes, a programmer cannot see the problem when too close to it. Taking a break, whether a break from the project completely or simple moving on to another part of the program, then coming back sometime later can provide a fresh perspective, uncovering the bug quickly.

Explain the bug to someone else. Talking through the bug to another person can allow him/her to provide insights. In addition, often the very act of explaining a bug will reveal the problem to the one doing the explaining.

Debug with a partner. The partner may have new techniques or ways of approaching problem solving. Debugging with a partner has many of the same advantages of peer-programming and peer code review. Plus, having a partner provides for a built-in excuse to explain the bug to someone else.

Get outside help. There may be tech support for consoles, a third party library, examples on the Web, online forums on the Web, libraries with books and other materials and even professors and students that can help. Programmers should not hesitate to take advantage of the many resources that might be at their disposal.

5.2 Common Bugs in Dragonfly

While just about any error is possible when developing a game engine (and game), this section contains some of the more common errors observed in Dragonfly development. Note,

the "Message" sections are meant to be illustrative representations of compiler output, but may not exactly match the errors produced by all compilers or systems.

5.2.1 Undefined Constructor in Base Class

Message:

```
GameManager.cpp: undefined reference to 'Manager::Manager()'
```

The above message, or messages like it, stem from not declaring a constructor in a parent class (e.g., Manager), even if it not used. For example, while the intent may not be for there to ever be an instance of a Manager – because specific game managers, such as the DisplayManager, are supposed to be derived from the Manager class – the Manager still needs a defined constructor, even if it is just empty. The base constructor gets called automatically when constructing an object in a derived class.

5.2.2 Vtable Errors

Message:

```
Object.o: In function 'Object::Object()':
Object.cpp:(.text+0x14): undefined reference to 'vtable for Object'
```

The above type of message arises when a method is declared as `virtual` (e.g., `eventHandler()` for class Object), but is never defined. The confusing part is the error description does not define what method, exactly, is missing. The solution is to be sure that all `virtual` methods are defined (such as the previously mentioned `eventHandler()`), even if they are expected to be re-defined in a derived class (e.g., a game object).

5.2.3 Time System Call Errors

Message (Linux):

```
Clock.cpp: undefined reference to 'clock_gettime'
```

For Linux, the above message occurs during linking if the real-time library *is not* linked in via `-lrt`. It should be.

Message (Mac):

```
ld: library not found for -lrt
```

For Mac, the above message occurs during linking if the real-time library *is* linked in via `-lrt`. It should not be.

If using a Makefile included with the Saucer Shoot tutorial (Chapter 3), the lines for the appropriate development platform need to be commented/uncommented.

Message (Mac):

```
function 'delta':
error:'CLOCK_REALTIME' undeclared (first use in this function)
error: (Each undeclared identifier is reported only once
error: for each function it appears in.)
```

For Mac, the above message occurs during compilation if `clock_gettime()` is used instead of `gettimeofday()` `gettimeofday()`.

See Section 4.4.2 on page 66 for details on system calls for game loop timing.

5.2.4 Header Definition Errors

Message:

```
ObjectListIterator.h: In member function ‘void ObjectListIterator::first()’:
ObjectListIterator.h:28:26: error: expected ‘;’ before ‘}’ token
```

The above message can occur if a method body is defined in the header file during multi-file compilation – in this case, `first()` in `ObjectListIterator.h`. The problem arises since ObjectListIterator has both Object and ObjectList as forward references, indicating to the compiler it is seen and can be referred to but has not yet been defined. By defining code (such as `first()` in this case, but it can happen for any method), the compiler complains. The solution is to not have any method definitions in the header file (`.h`) for a class, especially one that is forward referenced. Instead, all method definitions should be put in the corresponding class `.cpp` file.

5.2.5 Personality Errors

Message:

```
lib/libdragonfly.a(GameManager.o):GameManager.cpp:
          undefined reference to ‘___gxx_personality_sj0’
```

This message typically only occurs when trying to link in a pre-compiled Dragonfly library (such as downloaded from the Dragonfly Web page) where the libraries are incompatible. In this case, the “personality” error stems from the compiler and library being used are different than those used to compile Dragonfly (e.g., `libdragonfly.a`). The solution is to refer to the Web site for exactly the versions of the libraries used to compile Dragonfly, and make sure to have only those installed.

5.2.6 Syntax Errors

Message:

```
WorldManager.cpp: In function ‘int startUp()’:
WorldManager.cpp: error: ‘deletions’ was not declared in this scope
```

The above error (and many, many that look similar) stems from defining `startUp()` in `WorldManager.cpp` without including the `WorldManager::` in front (i.e., it should be declared `int WorldManager::startUp()`). This error in semantics is surprisingly common since to the eye, the syntax looks fine (and actually, in terms of syntax, it often is) so the compiler does not provide helpful debugging messages. The solution is to look carefully at the method names in the `.cpp` files and make sure to pre-pend the class names in front of them.

5.2.7 Makefile/Project File Errors

Message:

```
*** No rule to make target 'libdragonfly.a', needed by 'game'.
Target 'all' not remade because of errors.
```

or message:

```
game.cpp: fatal error: GameManager.h: No such file or directory
```

The above errors (and others like them) stem from having one or more paths set incorrectly in the `Makefile` or project. The path to Dragonfly (`libdragonfly.a`) needs to be defined for linking, while the locations of the Dragonfly header files (e.g., `Object.h`, `GameManager.h`, ...) need to defined for compilation. Errors such as the above suggest one or more of these variables is set incorrectly. The solution is to be sure the Dragonfly library and header files are in the locations specified.

Similar messages can appear if dependencies are generated on a different machine than the machine being compiled (say, because the code was copied from one machine to another). Typing `make depend` and `make clean` (Linux) or doing a build clean (Windows) after adding a new class definition (or after transferring the source code to a different machine) can fix this problem.

Message:

```
make -k
Makefile:68: *** missing separator.  Stop.
```

The above error can occur if the line (line 68 in this example) with the system command is not preceded by a tab. The basic `Makefile` is composed of:

```
target: dependencies
[tab] system command
```

Where the `[tab]` above *must* be a tab and not spaces. This is often overlooked when a `Makefile` is entered by hand using an example printed in a book. The line with a tab can look the same as one with spaces, but the latter results in the "missing separator" error above when trying to compile. The fix is to make sure all system commands under the target have a tab instead of spaces.

5.2.8 Segmentation Faults

Message:

```
Segmentation fault.
```

The above message may be seen for a variety of reasons, and nearly all of them stem from the same problem – a program has attempted to access an area of memory that was not allowed to access. This can happen when using incorrect control strings with input (e.g., `printf()` or `scanf()`), accessing beyond the bounds of an array (remember that in

C++, an array of size `n` is numbered from 0 to `n-1`), or dereferencing a pointer or address that has not been assigned properly. What can be particularly frustrating is that the error message does not say which of these is the problem, nor even what line of code has caused the error. The solution is to first narrow down where the error occurs. This is most easily done with a debugger. An example using the GNU debugger, `gdb`, is as follows:

1. Compile program with `-g` flag (e.g., `g++ -g mycode.cpp`).
2. Run `gdb` on the compiled program. (e.g., `gdb game.exe`).
3. Type `run` and, when it faults, type `backtrace` to have the compiler indicate where in the code stack, the fault occurred.

For a step-by-step example, see:

`http://www.unknownroad.com/rtfm/gdbtut/gdbsegfault.html`

The above steps do not necessarily indicate what, exactly, the problem is since a pointer problem can spill over into "good" memory space. However, in many cases is does provide direct information on the variable or memory access that is the problem.

For a little more code exploration within `gdb`, the commands `up` and `down` can be used to move up and down the code stack, using `print` to display values of variables.

5.2.9 Redeclaration of Classes

Message:

```
Redeclaration of class ClassName...
```

The above error (with the actual class name instead of ClassName) typically occurs if the header file, say `ClassName.h`, has been included from multiple source files. This error occurs most often for utility-type classes (and functions) that are used by multiple other classes (e.g., the LogManager). The fix is typically to use conditional compilation directives for the compiler pre-processor. See Section 4.3.4 on page 58 for details. If conditional compilation directives are in use, they should be checked that the names used in the `#ifndef` and `#define` statements are identical.

5.2.10 Static Arrays Too Large

Message:

```
relocation truncated to fit: R_X86_64_32 ...
```

The above error (or something similar) occurs if there is too much static memory allocated at compile time. This typically happens when arrays are given too large a maximum. For Dragonfly, the likely culprit is a large maximum given to ObjectList. The solution is to make arrays smaller or use dynamically-sized lists (see Section 4.5.2.3 on 86).

5.2.11 Deleting Statically Allocated Memory

Message:

```
*** double free or corruption error: 0x0000000001d3e010 ***
```

The above error, or one like it, happens when a variable is deleted (via `delete`) that was not allocated via dynamically `new`. This error can also occur if dynamically allocated memory is deleted twice. Or, this can also happen if an individual unit of an array that was dynamically allocated is deleted (e.g., `delete array[10]`).

The solution to all these errors is to make sure there is only one `delete` call for each `new` and to make sure memory is not deleted from parts of a dynamically allocated chunk.

Although this discussion has used `new` and `delete` as examples since they are common in C++, the same holds for `malloc()`, `calloc()` and `realloc()` paired with `free()`.

5.2.12 Sprite is Invisible

Output:

(Nothing – a Sprite that has been created in a sprite file does not show up when loaded)

The return value of the call to the ResourceManager `loadSprite()` method should be checked – it returns 0 if the sprite is successfully loaded, otherwise it returns -1. In most cases of a sprite display bug, the Sprite did not successfully load.

The directory path to the sprite file should be verified to be given properly to `loadSprite()`. For example, the Saucer Shoot game (Section 3.3) presumes the sprite files are in a sub-directory labeled `sprites/` that is one level down from the game's directory:

```
0  RM.loadSprite("sprites/saucer-spr.txt", "saucer");
```

The location of the sprite file (`saucer-spr.txt` in this example) should be verified and fixed, as appropriate.

If the sprite file is in the right place, the file content should be checked for errors. When a sprite file is parsed in `loadSprite()`, any line that has an error produces a message in the logfile and the loading stops (returning -1). Errors can occur for a wide variety of reasons: width and height parameters are incorrect, color is not one of the recognized colors, number of frames does not match number of `end` delimiters, and missing `eof`. A common error that is hard to see when creating a sprite file in an editor is to have an incorrect number of blank spaces at the end of a line. For example, a three line sprite frame may look like:

```
0 \
1 ~==-
2 /
```

Each line needs 4 characters, but visually it cannot be verified if line 0 and line 2 have the correct number (3) of spaces. The Dragonfly logfile should help determine this by providing some detail as to what line is at fault and why (e.g., too many or too few characters on line XX).

If the Sprite loads properly but still does not display, the physical location (x,y) of the Object should be checked to be sure it lies within the camera view. An Object that is drawn

outside of the view (i.e., off the visible screen) does not generate an error, but it is not seen either.

Lastly, if a Sprite loads, and the Object's position is within the camera view, the visibility attribute of the Object should be checked to verify that it is visible. In Dragonfly, Objects are visible by default, but can be turned invisible by the method `setVisible()`, as needed. This assumes, of course, that the optional invisibility is supported (see Section 4.17.1.2 on page 236).

5.2.13 Collisions Do Not Occur

Output:

(Two objects collide but a collision event is not generated.)

This error could occur for a variety of reasons. A quick checklist includes checking that:

1. The Object Boxes are, in fact, overlapping (colliding).
2. Both Objects are solid (`HARD` or `SOFT`).
3. The game code calls WorldManager `moveObject()` and does not call Object `setPosition()`. The latter just changes the position of the Object but does *not* check for collisions.

5.2.14 Errors using LogManager writeLog()

Output:

error: cannot pass objects of non-trivially-copyable type 'std::string {aka struct std::basic_string < char > }' through '...'

This compilation error comes when trying to print a string with `writeLog()` via `%s`, e.g., `std::string s; writeLog("%s", s);`. The string must be converted to a C-string via the `c_str()` method, e.g., `std::string s; writeLog("%s", s.c_str());`.

5.2.15 Errors compiling SFML

In developing with SFML, a compilation error message may be:

Message:

```
<SFML/Graphics.hpp>: fatal error: No such file or directory
```

In the above message, the system header file `<SFML/Graphics.hpp>` cannot be found by the compiler. Be sure SFML is properly installed for the development platform. Once that is confirmed, verify that the include path for the compiler is properly set to include the SFML directory with the header files.

Once that error is solved, when the compiler tries to link, a message may be:

```
DisplayManager.cpp: undefined reference to 'sf::RenderWindow()'
DisplayManager.cpp: undefined reference to 'sf::VideoMode()'
```

and other errors pertaining to calls with the 'sf::' prefix. In these second messages, the SFML header files are in place and resolved, but the linker does not find the needed SFML libraries. Check that the necessary SFML libraries are also in place. Also, check that the compiler is linking in all the needed SFML libraries:

```
sfml-graphics sfml-window sfml-system sfml-audio
```

Also, verify that the include path for the compiler is properly set to include the SFML directory with the libraries. Note, this is typically a different setting in an IDE than the setting for the include files. In Microsoft Visual Studio, this may involve copying the SFML libraries (.dlls) to the directory where the executable is running.

5.2.16 Errors running SFML

If the engine with SFML will compile, but does not appear to display output properly, try testing some simple SFML code outside of Dragonfly completely. For example, Listing 4.71 provides the minimum needed SFML output functionality needed to develop Dragonfly. If compiled and run successfully, it provides some confidence that SFML is properly installed and the development environment is setup to use it. Remember, instead of typing the example in by code (which, admittedly is not too hard) all Listings are available for download on the Dragonfly Dragonfly book Web page (`http://dragonfly.wpi.edu/book/`).

Output:
error: debug assertion

If this error occurs when developing on Windows, check that there is a call to `resetBuffer()` that needs to be made to any `sf::SoundBuffer`. This call should be placed in the Sound destructor (`~Sound()`). See Section 4.14.2 on page 193.