

Code Complexity and Coverage in the Godot Engine

Assignment 3 in DD2480

Teo Nordström

teon@kth.se

Rifat Kazi

rifat@kth.se

Linus Svensson

linussv@kth.se

Syam Kumar Vemana

vemana@kth.se

February 2025

1 Project

Name: The Godot Game Engine

URL: <https://github.com/godotengine/godot>

OUR REPO URL: <https://github.com/Teonord/godot-DD2480>

Godot is a open-source, free to use Game Engine for 2D and 3D projects. It is the engine of choice for many popular Indie games, such as Buckshot Roulette, Brotato and Luck be a Landlord.

2 Onboarding Experience

Godot, as a very open-source friendly project, has a detailed documentation stating pretty much everything one could need to start helping with developement of the engine¹. After fiddling a bit with WSL and not having any luck, it was realized that building straight in Windows was much easier. All one needed to do was to install the Python Software Construction Tool called SCons through PIP, and then run it with the command prompt inside of the Godot folder. This folder has all the files necessary for SCons to understand what and how to compile, and it installs all dependencies automatically. It also detected our operating system and compiled immediately as an executable for Windows, which meant it was easy to test that the compiled version worked. First compilation takes quite a while, but the compilation finishes with no major errors.

For running the tests, we just had to compile the project with the command `scons tests=yes`, and then run the generated executable with a special flag by calling `bin\godot.windows.tools.x86_64.exe`

¹<https://docs.godotengine.org/en/latest/contributing/development/compiling/index.html>

--test. More information about the testing can be found in the Godot Docs². Running the tests show some errors, but these seem to be purposeful as all 1171 tests run with success, though one test is skipped.

3 Complexity

Linus: I have chosen to look at `get_euler()` function in `core/math/basis.cpp`, start line 457. It is a method that computes and returns Euler angles from a given 3x3 matrix. When running with Lizard we get a CCN of 29. We also get 29, or 30 if including the function start itself. The function has a total of 141 lines of code, which means that it on average contains a path on every 5 lines. Most of these are single line `if()` containers or `case()`. According to our best guess, the tool (Lizard) does not take the function start in consideration. The documentation of the function is not clear about the different possible outcomes.

Teo: The part that I focused on is `intersects_transformed` from `/core/math/rect2.cpp`. It is a method that checks whether a transformed rectangle in 2D space would intersect with the current rectangle. When running with Lizard, this function gets a CCN of 26, which is unlike manually counted 25. This may be because Lizard counts the `#ifdef` as an extra `if` statement. To get our value, we still need to assume that compiler pre-directive `MATH_CHECKS` is defined. With a total of 133 lines of code, the function is quite cyclomatically complex, with a new path on average every ≈ 5 lines. However, almost all of these are just leading to the same points and furthering the program to the next jump, so they are single line `if` containers. There are no exceptions in this function, but if there are faulty arguments or problems in the called methods that we counted the CCN would rise dramatically. The documentation³ does not mention this function at all.

Rifat: I chose to look at `et_closest_points_between_segments` from `/core/math/geometry_3d.cpp`. It tries to find the closes point between two vectors. The closest points are then stored in the location given by two pointers that are passed in as arguments. Running lizard on `geometry_3d.cpp` gives CCN of 28 on the function. My manual count yields 29 CCN. To calculate the CCN i made a control flow graph and counted edges and nodes, although I didn't draw the ternary expressions i just counted them manually. The function i choose is a public function although it doesn't have any documentation as it is only used as a helper function for this function https://docs.godotengine.org/en/stable/classes/class_geometry3d.html#class-geometry3d-method-get-closest-points-between-segments. which is fairly well defined.

²https://docs.godotengine.org/en/stable/contributing/development/core_and_modules/unit_testing.html

³https://docs.godotengine.org/en/stable/classes/class_rect2.html

Syam:

I chose to analyze `AABB::intersection` from `aabb.cpp`, which determines whether two axis-aligned bounding boxes intersect. The function originally had a CCN of 20 due to multiple number of `if-else` conditions and repeated logic for handling each axis separately. Each axis was checked independently, resulting in redundant conditional statements that made the function difficult to read and maintain. This approach also led to unnecessary code repetition, increasing the risk of errors.

For `SteamTracker::SteamTracker` in `stream_tracker.cpp`, the constructor had a CCN of 16 due to extensive conditional checks for OS-specific Steam API paths and dynamic library loading. The constructor has inline conditional logic that handled multiple cases, making it overly complex. Each operating system had its own block of logic, interwoven with function calls, leading to high complexity and reduced readability. This made it difficult to modify or extend the function without introducing potential errors.

4 Refactoring

Linus: The `get_euler()` function contains much code replication. Each euler order follows a similar structure with minor variations. Thus, we could refactor by introducing a helper function with the repeated logic for getting the euler angles while handling singularities (gimbal lock cases). This helper function would take the relevant matrix value, row indices and sign adjustments to calculate the angles. By doing this, we would replace six very similar code segments with a single function call per euler order. With this, `get_euler()` would go from a CCN of 29 (or 30), to a CCN of 29 (or 30) - 22, since there are a total of 22 `if` statements and extra `&&` within the logic that will be moved to the helper function. Although the actual helper function would of course add cyclomatic complexity. This refactoring would result in much more readable and less cyclomatic complex code in `get_euler()`. However, there might be a slight performance impact by introducing extra function calls, although it should be minimal. I decided not to implement my refactor.

Teo: The `intersects_transformed` has several `goto`-points, which not only is generally considered bad practice but can also be easily replaced by functions. So, to start the refactor, we will put each jump destination into a separate helper function. We can also move the first `goto` check, as the code is very similar to that used in the jump segments. Now, instead of one function with a CCN of 25, we have 6 functions with the main function having a CCN of 9 and all others having a CCN of 5. This gives a TOTAL CCN 9 higher than the original, which makes sense considering we are adding 5 functions and adding a return point in an `if` statement for each of these, while not removing any paths. Each function now tests for one individual aspect necessary for intersection and returns true if intersection is possible, with the final function being the one who decides if intersection actually is happening or not. The impact of this refactoring is slightly more readable code, and much improved cyclomatic complexity - however, calling several functions does have an ever so slight performance impact from having to keep track of arguments and returns.

These functions in specific all only require a pointer as an argument, however, which hopefully alleviates this. The git diff for the refactoring can be found here: <https://github.com/Teonord/godot-DD2480/commit/c4ad089dd60a0a7cc39fb175d788bde492e2c902>.

Rifat: **Planning:** The reason the CCN is so high on `geometry_3d::get_closest_points_between_segments` is because it has so many if and else statements, and then on top of that it has a lot of ternary operations which in themselves are if else statements. Easiest way to reduce the CCN for the function is to just split it up to different functions, one way to do that would be maybe to have the computation for parallel vectors in one function and non-parallel in another function. Although maybe it doesn't yield the most satisfactory result. The ternary operators in the functions do the same thing though, so I could possibly swap out that for a function instead. Drawback of first solution would be that it would be same code and it would also just be another function and more code. The drawback of both solutions is that we add another function albeit private, but con would be potentially less code, and less complexity in the primary function.

Implementation: I decided to only implement the second solution: a helper function to swap out the ternary operators that were essentially all doing the same thing (limiting the value between a floor and ceiling), that took the CCN of the original function from 29 to 12, the helper function has a CCN of 3. Together they have a CCN of 15. Individually the function has 16 less CCN from original which is a reduction of $\approx 57\%$. And if you add the CCN of the helper function (called clamp) to it is a reduction of $\approx 46\%$. The git diff of the refactoring can be found here: <https://github.com/Teonord/godot-DD2480/pull/26/commits/8fe20f796e14cfe87234539cde1ec38a3b0b3f3b>

Drawback of the solution is potentially that it is slower, but you could probably do inline instead of a function to make it faster, although I'm not completely sure about how the compiler would compile this and if it actually slower. Another drawback is we add another function that only really has one function.

Syam:

The `AABB::intersection` function previously used a series of nested if-else conditions to determine intersection boundaries, leading to high complexity. The function was refactored by replacing these conditionals with a loop iterating over three axes, ensuring that the logic remains compact and easier to follow. Instead of checking each axis separately, the loop generalizes the conditions, significantly reducing redundancy. This refactoring decreases the CCN from 20 to 11, a **45% reduction**, improving both efficiency and maintainability. The changes are documented in <https://github.com/Teonord/godot-DD2480/pull/35>.

For `SteamTracker::SteamTracker`, the original implementation contained OS-specific conditional checks and function calls within the constructor, making it difficult to read and modify. To address this, the logic was split into three dedicated helper functions: `get_steam_library_path()` for path retrieval, `load_steam_library()` for handling dynamic library loading, and `initialize_steam_api()` for Steam API initialization. This modularization ensures that each function is responsible for a single aspect of the initialization process, making future changes easier. While the CCN remains

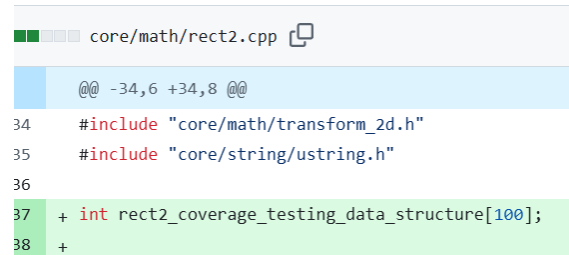
at 16, the structured separation of concerns significantly improves readability and maintainability, making the function easier to work with. Changes documented in <https://github.com/Teonord/godot-DD2480/pull/16/files>.

5 Coverage

5.1 Our Own Coverage Tool

To gather data around coverage, we used a simple 100 object list of ints that would be global when running tests. This will keep track of which branches has been taken by simply adding one to an index when the code passes through the point defined by that index. This is then printed after the tests have been run, giving us full coverage knowledge about our functions (assuming they are instrumented correctly).

For the `intersects_transformed` method, this is how the instrumentation looked.



```
core/math/rect2.cpp
@@ -34,6 +34,8 @@
34  #include "core/math/transform_2d.h"
35  #include "core/string/ustring.h"
36
37 + int rect2_coverage_testing_data_structure[100];
38 +
```

Figure 1: Adding of data structure for `Rect2::intersects_transformed` coverage testing.

```

28 core/math/rect2.cpp
108 108 }
109 109
110 110 bool Rect2::intersects_transformed(const Transform2D &p_xform, const Rect2 &p_rect) const {
111 + rect2_coverage_testing_data_structure[1]++;
112 +
113 113 #ifdef MATH_CHECKS
114 114 if (unlikely(size.x < 0 || size.y < 0 || p_rect.size.x < 0 || p_rect.size.y < 0)) {
115 + if (unlikely(size.x < 0)) rect2_coverage_testing_data_structure[2]++;
116 + if (unlikely(size.y < 0)) rect2_coverage_testing_data_structure[3]++;
117 + if (unlikely(p_rect.size.x < 0)) rect2_coverage_testing_data_structure[4]++;
118 + if (unlikely(p_rect.size.y < 0)) rect2_coverage_testing_data_structure[5]++;
119 ERR_PRINT("Rect2 size is negative, this is not supported. Use Rect2.abs() to get a Rect2 with a positive size.");
120 }
121 #endif
122 122
123 123 @@ -127,15 +133,19 @@ bool Rect2::intersects_transformed(const Transform2D &p_xform, const Rect2 &p_re
124 //base rect2 first (faster)
125 125
126 126 if (xf_points[0].y > position.y) {
127 + rect2_coverage_testing_data_structure[6]++;
128 goto next1;
129 }
130 130 if (xf_points[1].y > position.y) {
131 + rect2_coverage_testing_data_structure[7]++;
132 goto next1;
133 }
134 134 if (xf_points[2].y > position.y) {
135 + rect2_coverage_testing_data_structure[8]++;
136 goto next1;
137 }

```

Figure 2: This is a sample of the coverage calculation done within the intersects_transformed function.

5.2 Evaluation

- How detailed is your coverage measurement?
 - Our coverage measurement is as detailed as we want to make it. It can cover every path, if we make it do so. If there is a ternary condition or an OR within an if, all that is required is to create another check for each one. This can give us details about exactly which branches are tested and which are not.
- What are the limitations of your own tool?
 - An issue with our tool is that it potentially introduces extra if statements that will be necessary to add values when the path is within a ternary condition or an or statement in an if. This would technically make the function more cyclomatically complex. However, this is only done for testing, and these checks would be removed in the case of developing production code. It is certainly something worth considering, however. There's also no testing done for the entire paths, only which branches that the tests cross on their way to the end. This means we only know that one execution has gotten into a specific branch, not which branches led us to there.
- Are the results of your tool consistent with existing coverage tools?
 - Because of the size of Godot, it was hard to manage to get an existing coverage tool to

```

latest status: SUCCESS
Path 0 was covered 0 times.
Path 1 was covered 11 times.
Path 2 was covered 0 times.
Path 3 was covered 0 times.
Path 4 was covered 0 times.
Path 5 was covered 0 times.
Path 6 was covered 6 times.
Path 7 was covered 1 times.
Path 8 was covered 2 times.
Path 9 was covered 0 times.
Path 10 was covered 6 times.
Path 11 was covered 0 times.
Path 12 was covered 0 times.
Path 13 was covered 0 times.
Path 14 was covered 2 times.
Path 15 was covered 4 times.
Path 16 was covered 0 times.
Path 17 was covered 0 times.
Path 18 was covered 6 times.
Path 19 was covered 0 times.
Path 20 was covered 0 times.
Path 21 was covered 0 times.
Path 22 was covered 0 times.
Path 23 was covered 0 times.
Path 24 was covered 0 times.

```

Figure 3: intersects_transformed code coverage after the addition of four new tests.

work within the program. However, we have found nothing that would point towards our program getting any different responses than they would get.

6 Coverage Improvement

Linus: For the `get_euler()` function almost all branches were already tested. The only branch that was not tested was `default`. Although, there were also lacking tests for scenarios that should be invalid (not visiting any branch). Two tests, aimed to reach the `default` branch, were implemented. One with an invalid input and one with a correct input that is corrupted before use. Two tests for invalid scenarios were also added. One test with a `nullptr` as input and one with an uninitialized order.

Teo: For the method `intersects_transformed`, there was actually no coverage at all when we begun. This meant that increasing coverage was easy - testing in the first place would improve coverage wildly. For this function a total of five test cases were added. One of these did not give correct answers on obviously correct input data, which makes us believe that the function does not work properly when `Rect2s` are scaled. This one had to be commented out. There are, however, still four fully functional test cases with a total of eleven assertions. Once the code is run again with our code coverage shown, the output looks like this:

The test cases can be found in the GitHub repository commit 109bb3⁴ in file `tests/core/math/test_rect2.h`.

Rifat: For the method `geometry_3d::get_closest_points_between_segments` very few branches are tested. In the original code only 5 out of 17 (29%) branches are tested. After adding four of

⁴<https://github.com/Teonord/godot-DD2480/pull/4/commits/109bb3992010c5a35cea79a345d4520b525d39ba>

my own test cases for the function the coverage went to 11 out of 17 (65%) of branches. So some improvement were gained. What exactly each branch is doing was kind of hard to understand though, so to improve the coverage I was kind of going with my gut feeling and thinking of different ways you can setup two vectors instead of looking at what the math in the function was doing, which too be honest isn't really the best approach, but it did yield an improvement. The test cases I added can be found on <https://github.com/Teonord/godot-DD2480/pull/25/files#diff-916614ddbefa1d6062fa9e05d4870421f32c9aeaccd9a17c823d029723936d76>.

Syam:

For `AABB::intersection`, there were already existing test cases, making it difficult to increase coverage significantly. However, minor improvements were made by adding edge cases, such as AABBs that barely touch or have zero volume in certain dimensions.

The overall increase in coverage was small due to the already well-tested nature of the function. Updates are documented in <https://github.com/Teonord/godot-DD2480/pull/30/files>.

For `SteamTracker`, there were no existing test cases, meaning adding tests led to significant improvements in coverage. The new test cases covered various scenarios such as OS-specific Steam API loading, handling of missing libraries, and ensuring correct initialization of functions. This introduced new execution paths that were previously untested, significantly improving the robustness of the module. Test cases are documented in <https://github.com/Teonord/godot-DD2480/pull/35/files>.

7 Self-Assessment: Way of Working

We kept on working with GitHub issues as we have before. Every group member was expected to count and work on coverage for one function each. All other tasks were collaborated on, as in previous assignments. The rest of the Essence standard self-assessment can be found in a separate PDF.

8 Overall Experience

Linus: I feel that there are not many tasks during studies that teach or require you to provide further work on an existing large project. Which I presume will be a very likely thing that we all have to do when we actually work. I find this task to be an interesting experience that I believe will be very useful later on.

Teo: Working in a large project like this was equal parts fun and pain. There's a lot of code to look through which is very interesting, and the documentation was generally really well written which was a good help. The painful part was mostly compiling, and occasionally understanding what a function was expecting from us. The compiling took a long time on my computer, and it wasn't bettered by the fact that nearly the entire codebase had to be re-compiled occasionally if the

compilation failed because of an error. However, I have learned to not be quite as scared of working at these already developed programs, as there always seems to be room for improvement.

Rifat: Personally this was an interesting experience, I've always wanted to contribute to a open source project (or more specifically path of building community fork). But I've never really looked into how to do it, and now I have some general knowledge of how to go about doing it. When it comes to this project specifically, it is probably the first time I've had to modify and look into code written by someone else on something I initially had no clue on what it was doing or how.

This is also the first time I'm working on a big program that takes a long time to compile, normally I will every once in a while just do random things at see if things work. But this forced me to think through my changes before trying to compile as not doing so would just waste time.

Syam: Refactoring reduced CCN for `AABB::intersection` by 45% and improved modularity in `SteamTracker`. Coverage improvements were minimal for `AABB::intersection` due to existing test cases but significant for `SteamTracker`, where new test cases introduced entirely new paths. These improvements enhance readability, maintainability, and ensure robust test coverage, making future changes easier to implement without introducing unnecessary complexity. And initially I choose few fuctions which even with many new test cases did not give any coverage. So, a little of my time was wasted. Apart from that i enjoyed my time.