Министерство цифрового развития, связи и массовых коммуникаций РФ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет
телекоммуникаций и информатики» (СибГУТИ)

Кафедра Прикладной Математики и Кибернетики (ПМиК)

Лабораторная работа №5

по дисциплине «Визуальное Программирование и Человеко-Машинное Взаимодействие**»**

GOMS-анализ приложения Dependency Walker

Выполнил: студент 3 курса
ИВТ, гр. ИП-111
Прилепа М. К.
Проверила:
доцент ПМиК, Мерзлякова Е.Ю.

Новосибирск 2023

Оглавление

Описание пользователя. Описание программы

Программа предназначена для **не совсем обычного** пользователя ПК, обладающего навыками связывания библиотек. Приложение не русифицировано, потому подробное изучение интерфейса, а также поиск необходимых параметров и функций могут быть затруднены для человека, который не знает английского языка.

Самая главная цель этой программы – показать все системные и не системные зависимости любой библиотеки или exe-шник с другими библиотеками.

Я же эту программу использую в основном для того, чтобы из Release-сборки Qt-приложения собрать отдельный, не зависимый exe-шник. Конечно, библиотеки никуда не денутся, но зато они будут хотя бы рядом с ним и никуда не денутся при передаче через архив на компьютер, где даже нет QtCreator. Примером такого приложения является PlanetVPN, что и использовался для обхода ограничения по IP, не дающее залогинится в установщике QtCreator. Такая вот петля вышла, что обе вещи от друг друга зависят, хотя я не утверждаю, что в мире существует только один единственный VPN и он обязательно написан на Qt.

Задачи интерфейса

1.) Выбор любого exe-шника, либо библиотеки.
2.) Ожидание конца процесса сканирования без заламывания UI-потока. На самом деле я спутал это с курсором мыши, что имел иконку вращения. Если нажать на приложение во время его раздумий, будет видно заламывания UI-потока в виде надписи “(Не отвечает…)” возле заголовка виджета.
3.) Показывание всех зависимостей, которые даже не мог представить сам подопытный, пока не попал в эту программу (образно говоря).

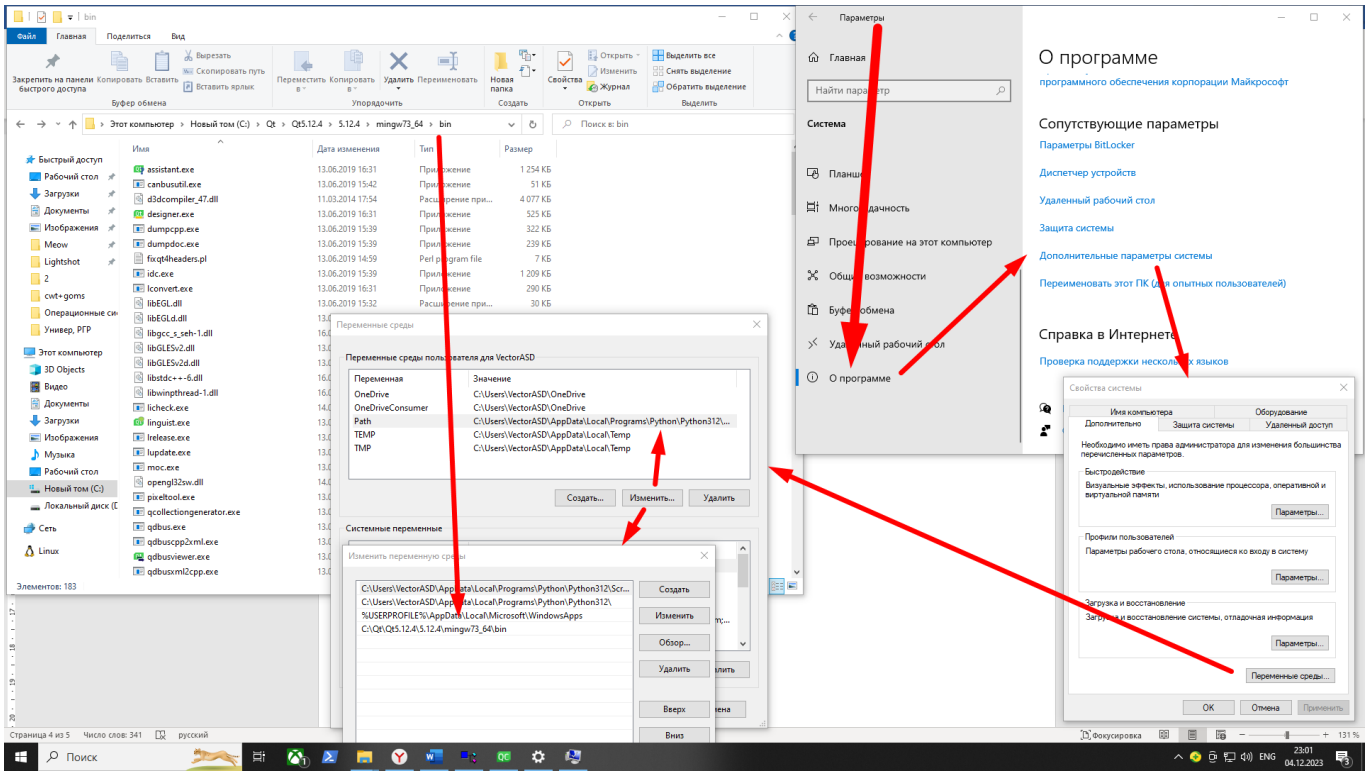
Задача пользователя. Решение. История

Задача №1

Цель:

В курсовом проекте, написанном на Qt, найти все библиотеки, которые нужны для функционирования этого Qt.

Есть небольшая оговорка, что в системную переменную окружения PATH должен быть заранее указан путь компилятора, который используется для сборки курсовой работы. Желательно, чтобы режим дебага был выключен. Без этого условия даже не получится запустить exe-шник вручную, т.е. без самого QtCreator.



GOMS-анализ допускает проведение дополнительных действий до самого основного анализа, ибо действия никак не связаны с программой, которую требуется проанализировать.

Цель я разобью на несколько подцелей:

1.) Добраться до папки D:\Meow\Documents\build-lab7-Desktop_Qt_5_12_4_MinGW_64_bit-Release\release; и скрыть файл lab7.exe (так уж вышло, что 7 лабораторная стала продолжение курсовой);

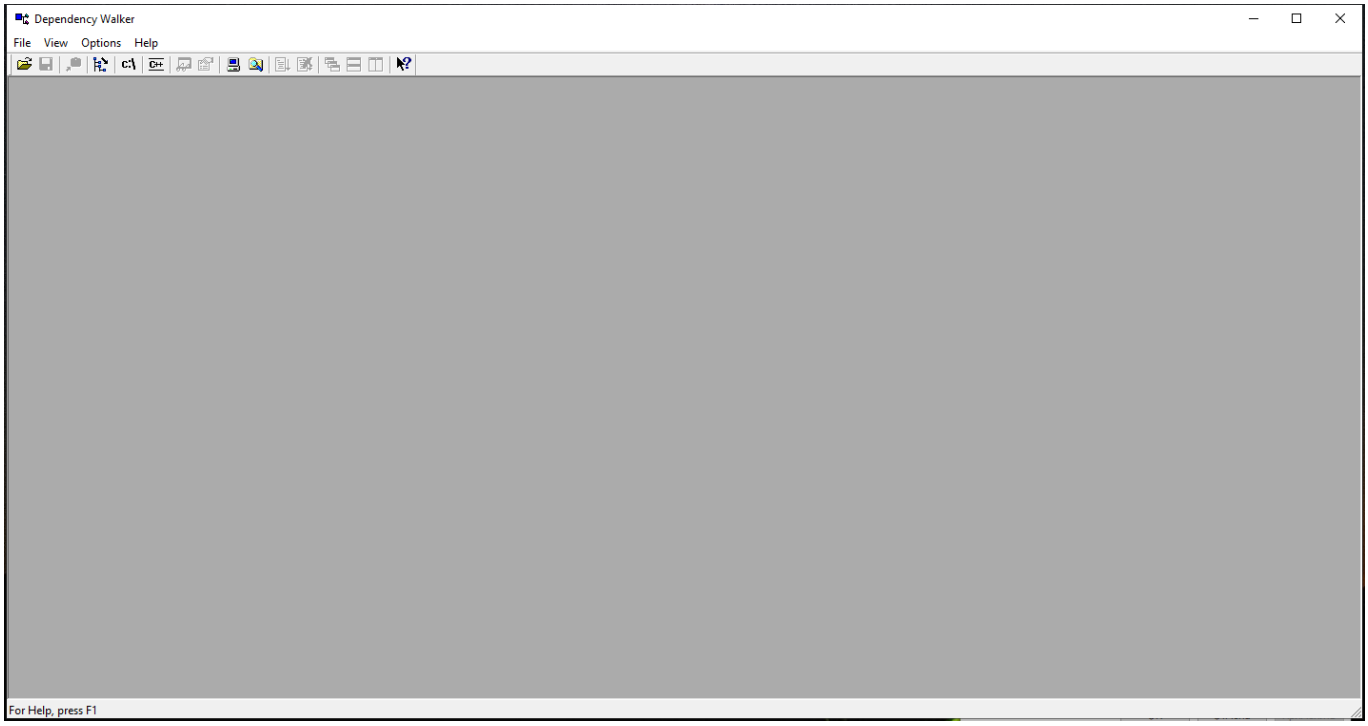
2.) Ждать, пока программа родит результат.

3.) Проанализировать.

Теперь пора каждой подцели выписать методы:

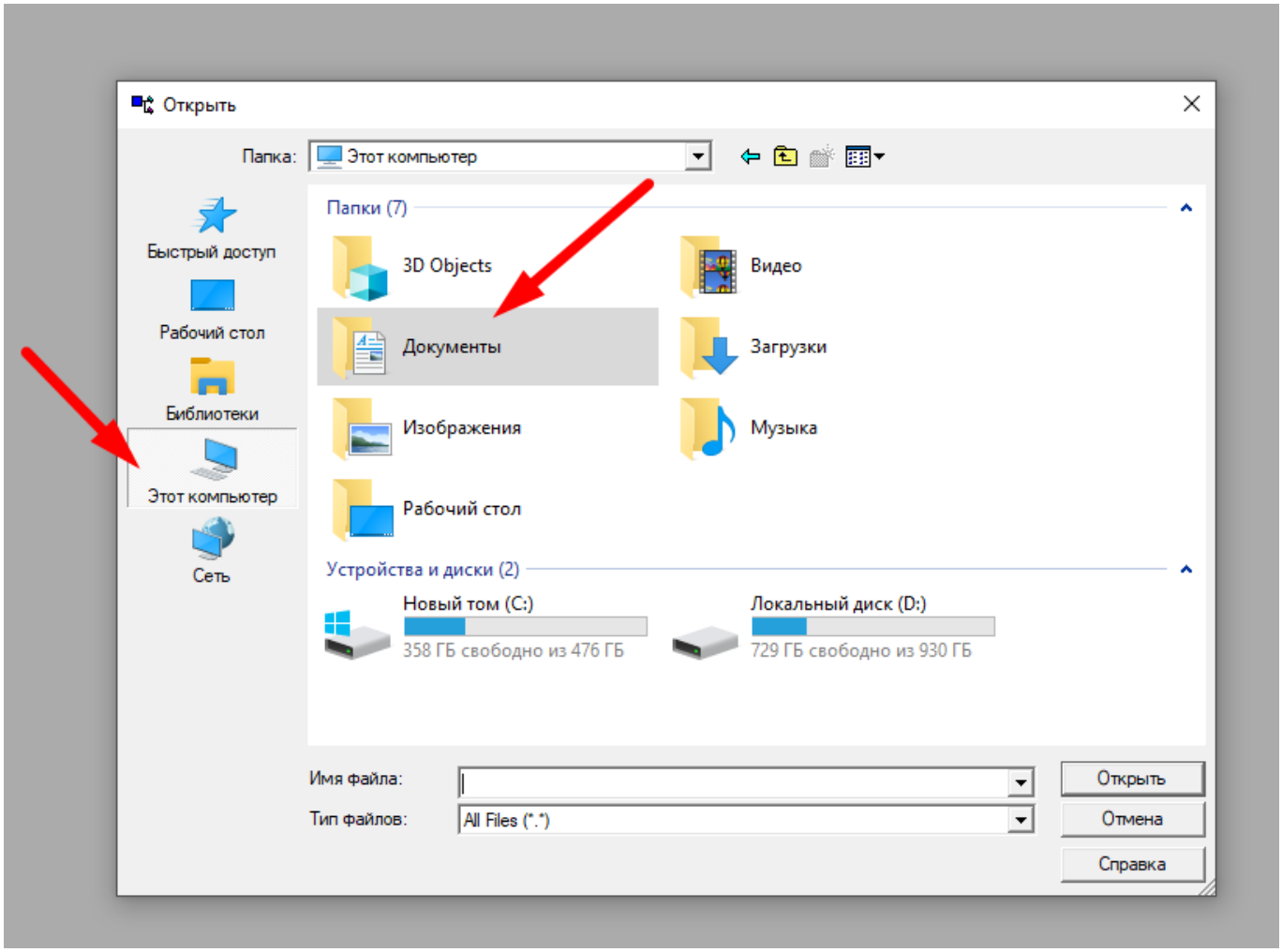
Подцель №1 (открытие папки):

Вот что видит пользователь:



1. Ему сначала нужно подумать, потом увидеть кнопку “File”, навести туда курсор, тыкнуть. Пользователей заранее знает, что дальше последует кнопка “Open…”, так что думать дополнительно не нужно, он снова наводит и снова тыкает. Ожидание компьютера здесь не учитывается, т.к. ощущается оно, как моментальное. (МРВРВ)

2. Тут можно схитрить, что уже открылась нужная папка, но допустим, что он до этого был не в ней. Напоминаю, что путь составляет D:\Meow\Documents\build-lab7-Desktop_Qt_5_12_4_MinGW_64_bit-Release\release, есть кнопка “Этот компьютер”, после чего мы увидим это:



Напоминаю, что пользователь опытный и делает это с самых пелёнок, но перед рядом всех операций перемещений мыши и кликов нужно подумать, чтобы просто сориентироваться в пространстве. Сначала кнопка “Этот компьютер”, потом папка “Documents”, потом “build-lab7-Desktop_Qt_5_12_4_MinGW_64_bit-Release”, потом “release” и наконец-то файл-exe-шник (важно, что только тут клик двойной). На всякий случай я перед длинным названием файла поставлю мыслительный процесс, т.к. допускается, что пользователь опытный, а файлов с подобными названиями накопилось очень много. (МРВРВ МРВРВРВВ)

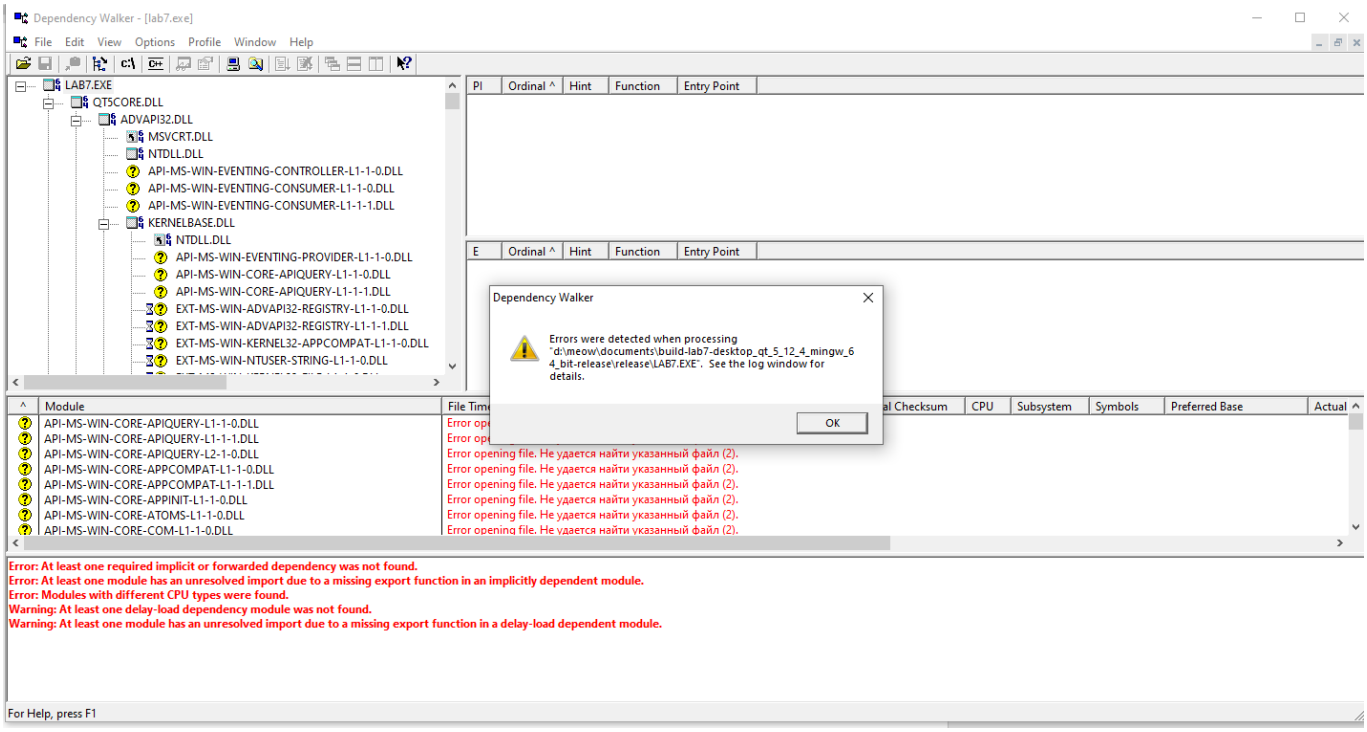
Подцель №2 (ожидание):

Это всего лишь ожидание, но оно довольно-таки небыстрое. Секундомер показал 9 секунд. У меня очень мощный компьютер, но в анализ включается не только около 3 Гб папка самого компилятора (они все примерно одинаково весят, да и это даже не компиляторы, а то, что выдали компиляторы при сборке Qt-запчастей, вот и потом вес почти одинаковый), но и самой операционной системы, вернее, папки C:\Windows\System32, что весит прям почти ровно 5 Гб. На самом деле программа не знает пути папок заранее, а обращается к ним только после полного анализа

Path-пути окружения и из-за причинно-следственной связи между библиотеками. Также, построение самого дерева библиотек тоже является частично проблемой интерфейса. R = 9 секундам и на миллисекунду, наверное, не больше и не меньше… По крайней мере завершение сопровождается звуком открытием окна предупреждения и я именно в этот момент увидел 9 секунд в секундомере. (R = 9 с.)

Подцель №3 (исследование):

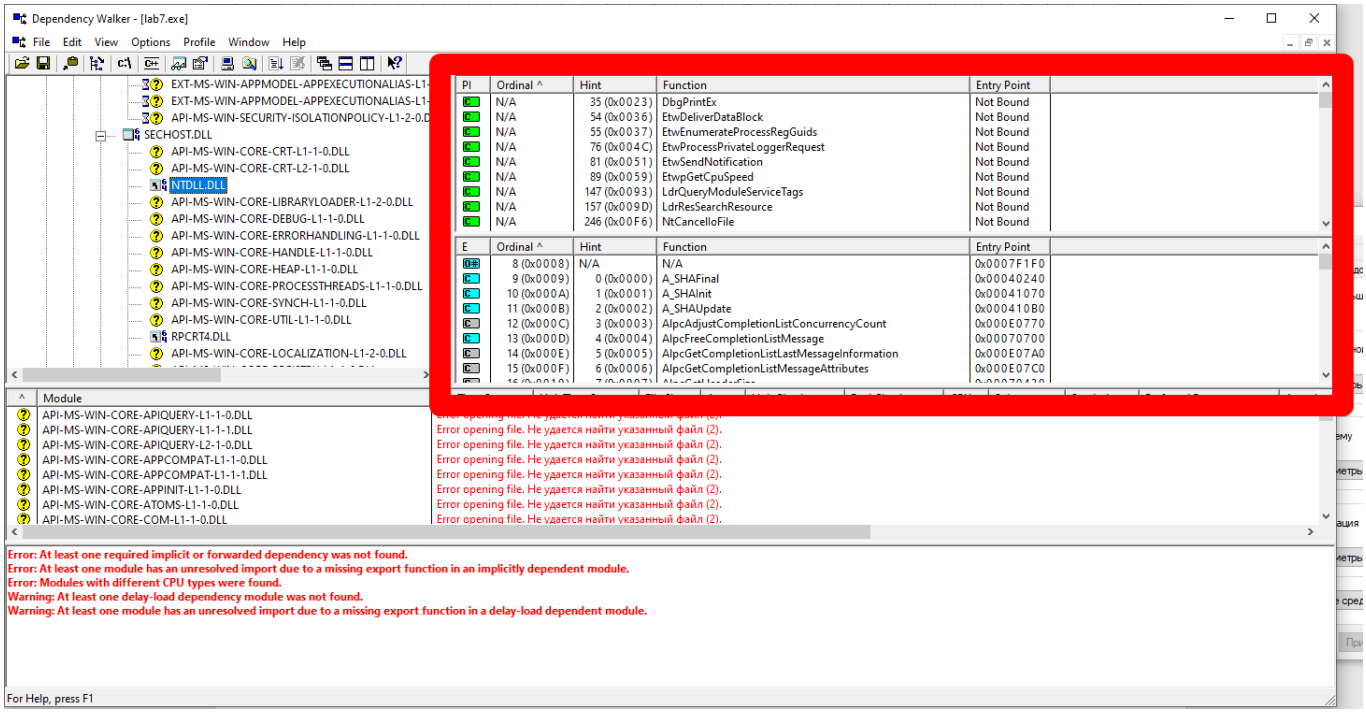
1.) Первым методом будет анализ того, что приложение не идеально и **всегда** анализ завершается звуком. В случае Qt это аксиома, т.к. между моим курсовым проектом, где уже много чего понаписано, и проектом Hello World exe-шник хоть весит по-разному, но время анализа (9 секунд) от этого не меняется, и начинка в виде почему-то ненайденных некоторых библиотек тоже не меняется, хоть это никак и не приводит к поломке работы exe-шника. Вот что видит пользователь:



Сначала он думает, видит ошибку, двигает курсором на кнопку “ОК” и вышвыривает компьютер в окошко, вернее, нажимает кнопку. (MPB)

2.) Ну и сам процесс анализа. Ибо наш пользователь ещё тот мастер-ломастер, он знает, что выгоднее навести 1 раз и кликнуть 1 раз, а потом перевести руку на клавиатуру и тыкать постоянно стрелочку вниз (Qt::Key_Down) вместо того, что каждый раз наводить курсор и тыкать кнопки. Допустим, что пользователь это сделал ***100 раз*** и угомонился. После каждого действия ещё дополнительное раздумье, ведь ему нужно просмотреть то, что справа… Между этими дополнительными раздумьями и следующим кликом на стрелочку вниз (палец уже нацелен туда ещё с первого раза), ему придётся сделать кучу действий, не касающихся самого приложения, так что триллион лишних М я включать не буду. (первый просмотр библиотеки = (MPBM), остальные 99 = (H)+99(KM)).

Я уверен, что мыслительные процессы все нужны, т.к. справа пишется постоянно это:



Ещё нужно учитывать, что вращение глаз между правой таблицей и названием библиотеки, что слева, тоже присутствует, но GOMS не учитывает вращение глаз.

Итог всех подцелей и их методов:

(MPBPB) + (MPBPB MPBPBPBB) + (R = 9 с.) + (MPB) + (MPBM) + ((H)+99(KM)) =

= 107M + 9P + 10B + R + H + 99K = 107 * 1.35 + 9 * 1.1 + 10 * 0.2 + 9 + 0.4 + 99 * 0.2 =

185,55 секунд. Т.е., у пользователя уйдёт **именно на приложение** 3 минуты, чтобы исследовать 100 библиотек. Действия с ними за пределами приложения в проводнике не входят в данный анализ. Обратите внимание, что думательный процесс, что пользователь нажмёт 99-раз клавишу нижней стрелки, а перед этим пододвинет курсор и сменит мышь на клавиатуру, состоится только 1 раз.

Задача №2

Цель:

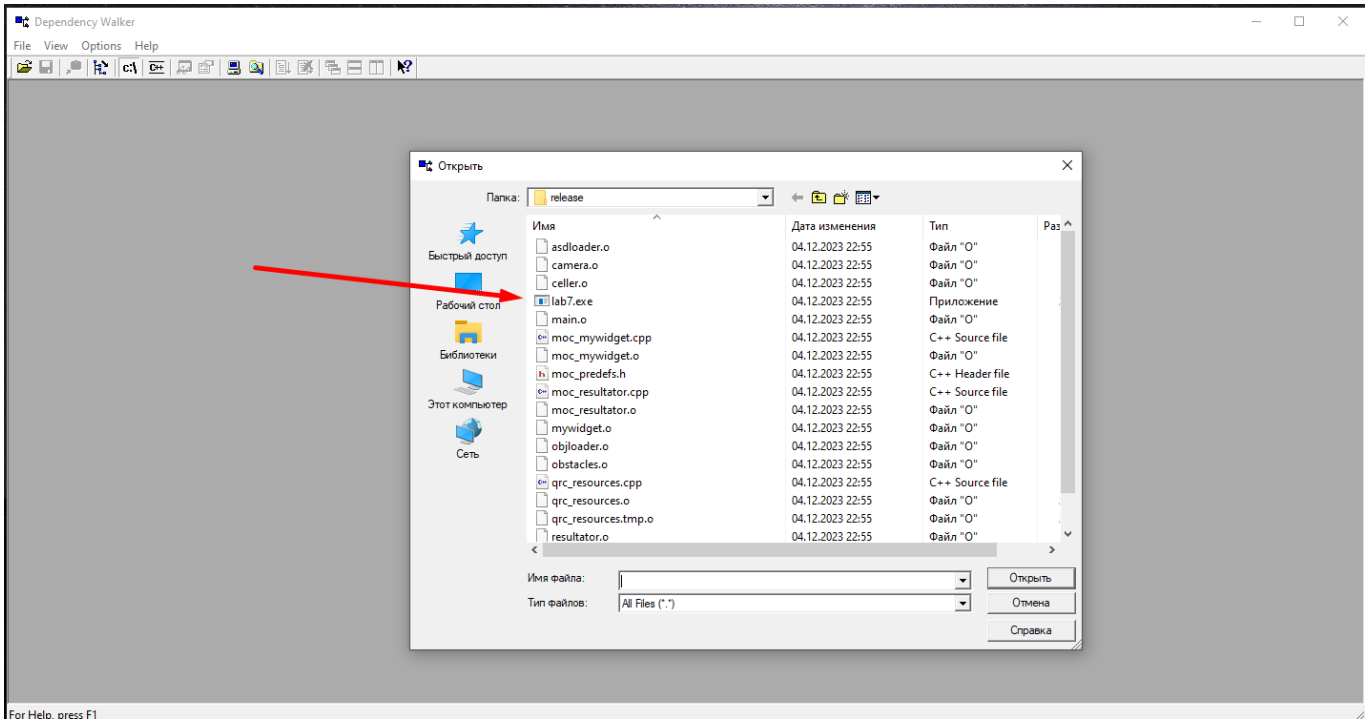
Давайте что-то более поэкзотичнее. Нужно узнать название функции и что вызывает её, когда вы изменяете размер окна.

Вот её подцели:

1.) Как и в прошлой подзадаче, нужно вскрыть exe-шник (подождать 9 сек.).
2.) Включить профилировщик и подождать (снова засеку секундомер для другой R).
3.) Потянуть виджет и посмотреть внутри исследуемого приложения.

Подцель №1 (вскрытие exe-шника):

1. В этот раз я допущу, что мы делаем эту подзадачу сразу после предыдущей, так что после раздумий, наведения курсора и нажатие кнопок “File” и “Open…” мы сразу увидим:



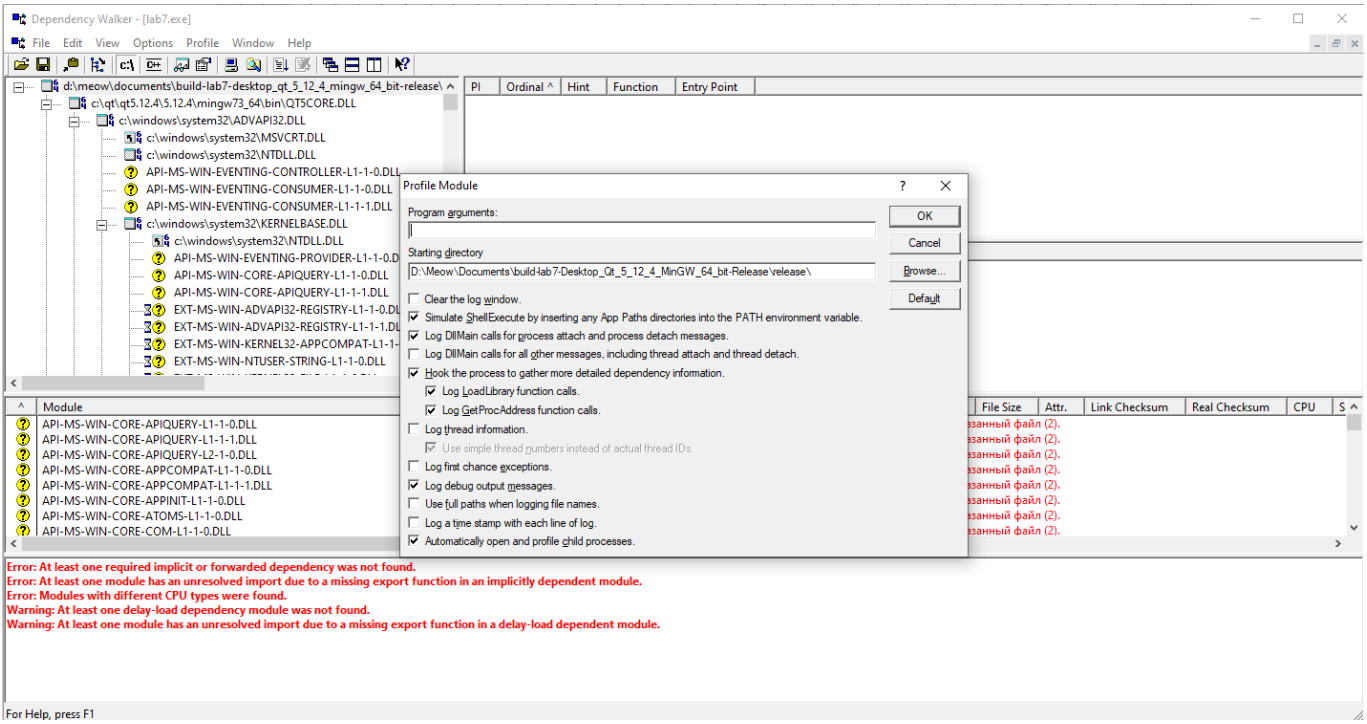
Далее, остаётся всего лишь поломать голову, повернуть курсор к “lab7.exe”, тыкнуть почему-то не двойной клик (ему так просто хочется), пододвинуть мышь к кнопке “Открыть” и тыкнуть. (MPBPB MPBPB)

2. Ожидание анализа в практически-константные 9 секунд тоже учитывается.

(R = 9 с.)

Подцель №2 (включаем профилировщик и снова ждём):

1. Думаем, видим после звука то же самое предупреждение, что не все зависимости удалось найти (для Qt это аксиома), двигаем курсор на “Ок” и тыкаем. ~~Далее, снова думаем (крутим глазами, желательно одновременно двумя в разные стороны), двигаем курсор к кнопке “Profile” и тыкаем Start Profiling…~~ Нет, я передумал, лучше пододвинуть руку на клавиатуру и просто тыкнуть F7-клавишу, пользователь же умный в отличие от меня (до этого момента) и точно знает с пелёнок лучше таблицы умножения, что там забиндена F7. (MPB**H**K)
2. Вот что пользователь увидел:

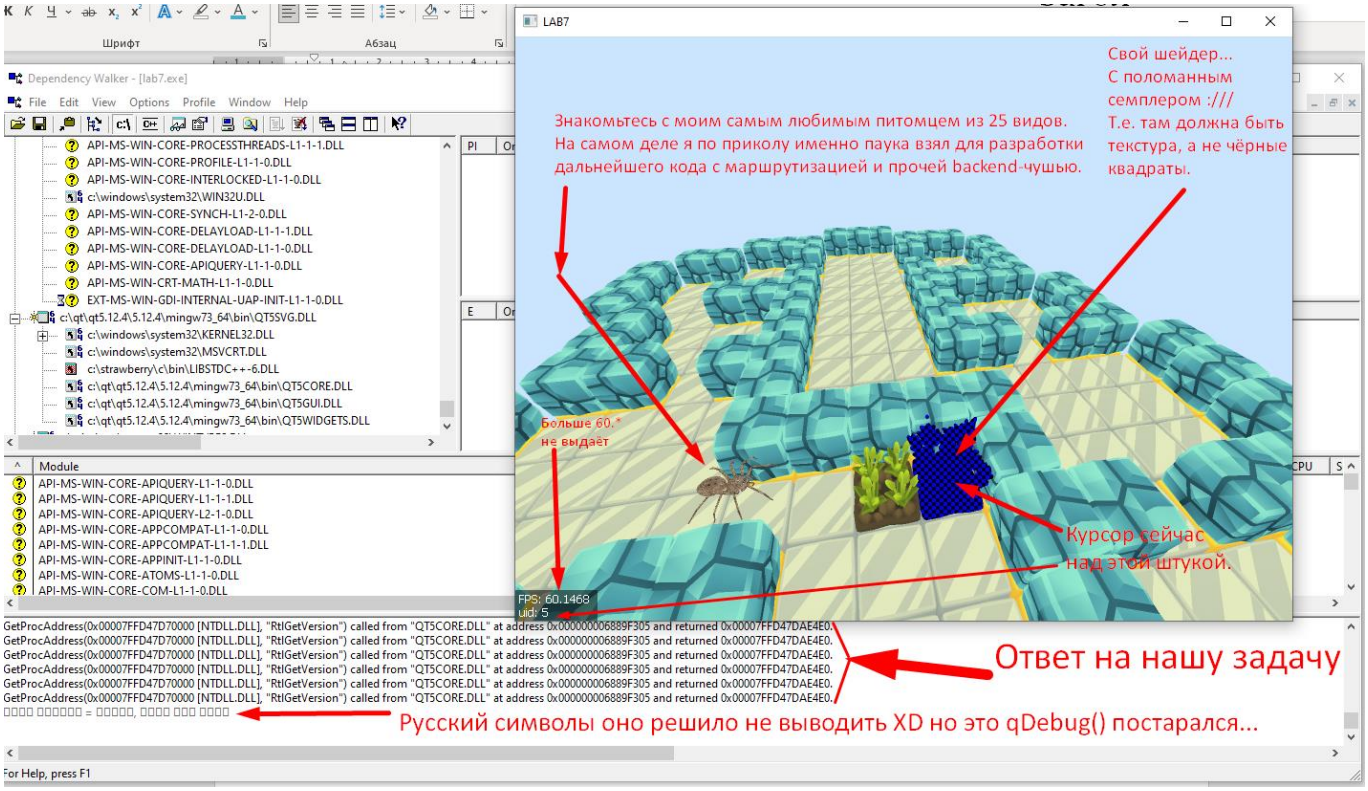


Пользователю стало дурно лицезреть сие шедевр, и он просто подумал всего лишь немножечко, вернул руку с клавиатуры на мышь, пододвинул курсор к кнопке “ОК” и раздавил её. Ещё нужно учесть ожидание. Секундомер показал 7.5 секунд. Теперь $R_2 = 7.5$.

(M**H**PBR_2, $R_2 = 7.5$)

3. Осталось всего лишь подумать, переместить курсор к левому-нижнему уголку (подойдёт любой уголок и даже не уголок, лишь-бы было что тянуть), потянуть (на это отдельная буква данного GOMS-анализа - D), ну и подумать глазами. (MPDM)

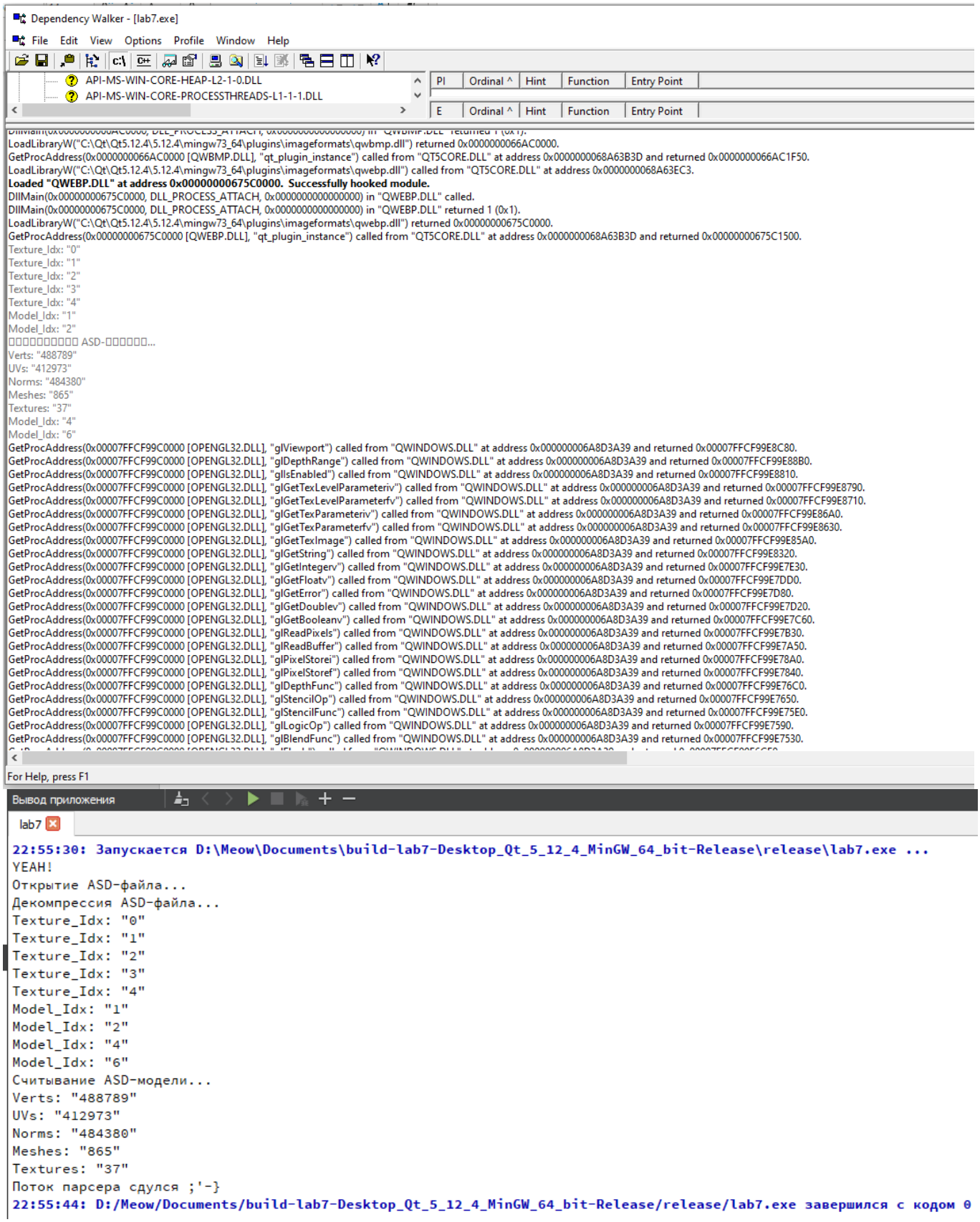
Вот что пользователь в конце нашего пути увидел:



И о да, я проспойлерил, как уже выглядит курсовая в данный момент! Ну и куча надписей… Многообещающий спойлер. В итоге мы увидели эту информацию в консоли: QT5Core.dll обращается по адресу 0x6889F305 к методу **RtlGetVersion** внутри библиотеки NTDLL.dll (начало которой по адресу 0x7FFD4770000), а результат этого метода (вызов ассемблерной команды-ret) должен счётчик команд привести к 0x7FFD47DAE4E0, что и есть точка вызова функции в самой QT5Core.dll в виде ассемблерной команды call + размер, что занимает эта команда, чтобы возврат был уже после вызова call.

Не знаю зачем нам эта информация, хотя, для инъекции она хорошо сойдёт, ведь подобная информация хороша, ведь приложение я скомпилил без дебага, а межбиблиотечное взаимодействие полезно раскрывать. Так можно и steam-плагин взломать в какой-то игре самому…

Конечно, этот скриншот не относится к GOMS-анализу:



Обратите внимание, что это место есть ничто иное, как перехватчик stdout-потока. Т.е. всё то, что мне писал QtCreator в себе из-за затрагивания qDebug(), теперь это работает спокойно в профилировщике данной программы. Ну и было полезно узнать, что все gl-команды я оказывается использую из библиотеки OpenGL32.dll. Профилировщик название библиотеки быстрее мне даст, чем мой собственный поиск среди 1000 системных и не системных библиотек, к примеру, **glViewport**-функции.

Нужно учесть, что это вызов получателя метода, а не самого метода. Скорее всего последний адрес после “and returned” означает как раз место, куда была вернут этот метод, как адрес библиотеки + адрес метода в библиотеке.

Итог всех подцелей и их методов:

(MPBPB MPBPB) + (R = 9 с.) + (MPBHK) + (MHPBR_2, $R_2 = 7.5$) + (MPDM) =

= 6M + 7P + 6B + R + R_2 + K + D + 2H = 6 * 1.35 + 7 * 1.1 + 6 * 0.2 + 9 + 7.5 + 0.2 + 2 + 2 * 0.4 = **36,5** секунд. В целом правдивый результат. Если бы я в R пихал что попало (в данном случае значение по умолчанию 0.2 секунды), то анализ казался бы не правильным по сравнению с реальной практикой. Опять же, учитывается только взаимодействие с UI и только.

Заключение

По сути, заключение, уже написал в конце каждой (из двух) задаче.

Спасибо за внимание!