

Tutorial 2: Debugging

Anton Gerdelan <gerdela@scss.tcd.ie>

Debugging Tools

- Pencil and paper (diagrams of, manual step-through)
- `printf()`, `assert()`
- Text editor, terminal
- Eyes, brain
- Debugger program - `gdb`, `lldb`, the one in visual studio
- A good front-end is nice

The Problem with Linux

- Debuggers are convenient for following the path code takes - functions, recursion, templates...
- Linux has a major dearth of visual debuggers
- gdb and lldb have a command line interface (okay for getting a **backtrace** after crash ... clunky otherwise)
- DDD, Eclipse, Code::Blocks, CGDB - none great

“Alright I’ll Just Make My Own Then!”

```

//EXTERMINATOR\\ by Anton Gerdelan @capnramses (B)reakpoint (R)un (spacebar/N)ext (W)atch (G)DB
src/main.cpp line (461/840)
438 update_controller_states ();
439 // update any camera effects
440 update_camera_effects (delta);
441 update_logic_steps (delta);
442
443 if (was_action_pressed (g_wipe_screen_action)) {
444     start_fist ();
445 }
446 } // endif vic/defeat/no panels open
447 } // endif !menuopen
448
449 //*****
450 // START OF PLAYER 1 VIEW
451 //*****
452 vec3 cam_pos = g_cam.world_pos;
453 // the +1 is to offset by half a tile
454 int centre_x = (int)((cam_pos.v[0] + 1.0f) / 2.0f);
455 int centre_z = (int)((cam_pos.v[2] + 1.0f) / 2.0f);
456
457 // shadow mapping pass
458
459 // note:
460 // this is about 72 draw calls or ~12 per shadow direction
461 if (g_shadows_enabled && g_cam.is_dirty) {
462     //glCullFace (GL_FRONT);
463     for (int i = 0; i < 6; i++) {
464         // NB. also sets viewport and clears fb depth
465         bind_shadow_fb ((Shad_Dir)i);
466         draw_manifold_around_depth_only (cam_pos.v[0], cam_pos.v[2],
467             SHADOW_CASTER_MAX_TILES_AWAY);
468         assert (render_props_around_depth_only (centre_x, centre_z,
469             SHADOW_CASTER_MAX_TILES_AWAY));
470         draw_snakes_depth_only ();
471     }
472     //glCullFace (GL_BACK);
473 }
474
475 // end of shadow mapping pass
476
477 /* draw count stats at start of treasure test level
478 w/o mmenu open and w/menu open
479 before manifold 48 0 -- this is b/c cam_dirty is always true when updating char
480 after manifold 16 16
481 after splats 0 0
482 after props 158 158
483 after sprites 1 1
484 after particles 1 1
485 after fb 0 0
486 after guis 3 3
    
```

Behold!
cam_pos {2
centre_z 1
cam_pos.v[0] 2
cam_pos.v[1] 10
cam_pos.v[2] 2

and despair!
#0 main (argc=1, argv=0x7fffffffde98) at src/main.c:55

crongdor

CHOOSE THY LEVEL!

- INTRODUCTION
- BIGPROPS
- TABLE
- WEBS
- WINCHESTER
- BRIDGE
- TORCHES
- ALL
- THREE_DOORS
- SNAKE
- PILLARTEST
- BOWLDER
- AI_TEST
- FALL_TEST
- PLATS
- PORTAL
- TREASURE_TEST
- HAMMER_TIME
- DARTS
- EIM1
- BOULDERS
- LAMPSTEST2
- ANTON2
- ANTON3
- TOWER
- PALACE
- DAGUJA
- LAMPS

INTRODUCTION

AND SO BEGINS THE TALE OF **CRONGDOR** -
BARBARIAN, RAIDER, THIEF.

“oh...that’s why there aren’t any decent ones...”

Enter: VS Code

- Visual Studio Code is very nice (simple, quick, visual)
- type in “**code**” in terminal to open
- install+enable the “**C/C++ extension**” (puzzle piece icon)
- you get a text editor
- **breakpoints, watch-list, stack trace,**
git integration,
- you don't have advanced views; **memory inspect, asm,**
etc.

Setup

- *File->"Open Folder"* gives you a 'project' view
- Open or create a new main.c file
- first let us compile the program (you can use the built-in terminal or an external one)
- `gcc -o my_demo main.c -g -std=c99` (Linux)
- `clang -o my_demo main.c -g -std=c99` (OS X or Linux)
- `mingw32-gcc my_demo main.c -g -std=c99` (Windows with mingw gcc)
- **-g** means "include debugging symbols in my program"
- you won't be able to debug without them

Start Debugging

- View->Debug
- Click in the margin (left of line numbers) to set a breakpoint (red circle)
- the *cog* button takes you to settings file `.vscode/launch.json`
 - which debugger to use, **enter your program name** to run, etc.

Stepping

- *Green arrow* starts program in debug mode
- Program pauses on breakpoint line
- Use **step over** (next line)
step in (next line but also go *into* functions)
step out (run until we leave the function)
continue (green arrow)
stop (red square)
- local variables' values (in scope)
- hover mouse over a local variable
- right click variable to **add to watch list**
- **call stack** of functions open with line that call the next function

Useful For

- Checking why variables or output do not have the value you expected - “what am I missing?”
- Check the flow of execution (esp. other peoples spaghetti code)
- Why lead up to it crashing?
 - “binary search” your breakpoints around

The command-line version

- After compiling with debug symbols
- `gdb ./my_demo` (or `lldb ./my_demo` for OS X)
- This enters a gdb terminal session
- type **run**
- after it crashes type **bt** to get a backtrace
 - which line caused the segmentation fault and what leads up to that
- type **q** to quit gdb

Alternatives

- On Apple install Xcode in Appstore to get all the programming tools (clang, lldb, etc.)
- Xcode and Visual Studio are pretty handy (but very bloaty with all the project settings)
- I still prefer gcc on Windows (but that's me)
- Some people like Code::Blocks (in lab) and QT Creator on Linux

Other Useful Tools

- **Profilers** {gprof (linux), Instruments (Xcode), VTune}
- **Static Analysis** - very handy code mistake finder - try Clang's static analysis tool on your code files.
- **Memory** leak checker - Valgrind
- **Friends** - code review / tips / sanity check