accu 2022

HOW CODE FAILS IN THE REAL WORLD

JAMES TURNER

Background

- James Turner, james@[flightgear.org | kdab.com]
- C++ developer for 25+ years
- Consultant developer at KDAB
- Contributed to open-source projects for 20+ years
  - Assorted feature work, especially around UI/UX
  - Build and release engineering
- Often the point of contact for end-user problem reports

FlightGear

- Legacy C++ codebase
  - Open-source, many contributors
  - Runs on all major desktop platforms
- Extensively data-driven via XML and scripts
- Loads user-generated content
  - Including scripts
- Content downloaded during runtime
- OpenGL
- Free-form threading

5 Stages of a user-reported failure

- Denial : ‘that can’t happen’
- Anger : ‘how the <expletive> does that happen’
- Bargaining : ‘if you change X, does it still happen?’
- Depression : ‘I hate computers’
- Acceptance : ‘Ohhhhhhhh. I hate computers. Fix is pushed.’

What I thought going into this

- Users report the software is ‘more unstable than the last version’
- Getting good feedback is hard
- On macOS & Linux, some users would send backtraces
  - Much easier to fix the issues
- Let’s integrate a crash reporter and life will be better!

Crash-reporting technology pieces

- Crash-reporting library or process
  - deployed with your application
- Build you app in release mode with debug symbols
  - If using CMake, use RelWithDebInfo with care
- Extract / archive those symbols (PDBs, dSYM etc)
- Strip the code before packaging & deployment
- Automate this on CI (Jenkins)

… And achieve nothing …

- CrashRpt
    - Simple, Windows only
- HTTP POST to an end-point you supply
    - To a directory on my DreamHost
- Directory full of zipped MiniDumps

… And the pieces that make it useful

- Aggregation backend (web service)
- Symbolication with vendor symbols (Microsoft, Intel, AMD, nVidia)
- Correlating symbol artefacts to builds / releases
- Annotating runs with meta-data
- Statistical grouping based on metadata

Crash context

- What was the user doing?
- What important configuration is set?
- What anomalous things have already occurred?

Well, if we already have a reporting backend, let’s collect this information as the program runs. When a crash occurs, we can include it in the report, and hopefully make any trends clearer.

Practical Notes

- For FlightGear, I'm using Sentry.io
  - Also used at KDAB by some customers
- APIs for many languages
  - native backend wraps Crashpad or Breakpad
- Command-line tool to upload symbols, define releases, etc
- Offers various self-hosting and hosted solutions
- I can heartily recommend it (#notsponsored)
  - They accept pull requests and respond to bug reports

The ‘typical’ crash

- Expected to find code as shown on the right
- Steps to reproduce are reliable
- Code-read in the problem area
- Trivial to fix with crash trace

```
auto myPtr = getFoo();
myPtr->engageRotor();

Airport* a;
if (a && a->getTower()->getPosition()) { …
```

An anthology of crashes

- Computers are slow
- Users are very impatient (or UI is bad)
- Archaic hardware
- Weird system configurations
- File-systems (especially on Windows) fail in all kinds of ways
  - %$@!$# OneDrive
- malloc() does actually fail

Impatience

- Database built on first run
  - Takes 1-5 minutes
- Progress dialog runs on the main thread
- Rebuild task uses a worker thread internally

```
DatabaseRebuildTask t;
t.start();

ProgressDialog d;
d.setUpdateCallback([&t, &d]() {
   if (t.isDone()) d.close();

d.setProgress(t.getPercentComplete
());
});
d.exec();

// continue with application startup
// do something using the DB
// crash, DB is not built yet?!
```

More Impatience

- Custom UI for menubar, inside the window
- macOS uses native menubar
- GUI is initialised during early startup
- Splash screen blocks window event interaction during loading

Still more impatience

- Code as shown
- Crash preparing some stuff
  - Twice, what?
- Multiple clicks on the button before the window actually closes

```
Startup::onLaunchMainThing
{
    prepareSomeStuff();
    closeStartupWindow();
}

…
StartupWindow s;
s.exec();
…
// cool, startup is done, continue with
main thing
```

Malloc, etc

- C++ (eg, STL) throws std::bad_alloc
- Caught in various places, hard to debug
- set_new_handler to the recuse!
  - Explicitly log an error report (and backtrace)

Slowness, network style

- HTTP check for new update on startup
- Reports back to startup GUI
- HTTP requests are ref-counted, cleaned up once done
- For some users, timeout after a long time
- Startup GUI is gone

Solution: add proper cancellation API to HTTP requests, so a cancelled request doesn’t report failure when cancelled

Drivers ☹

- Check available OpenGL versions
  - Give some clear user feedback if we can’t run
- Check when first window is created
  - NOPE
- Attempt to create an offscreen context
- Check the version which is returned
- Still crashes on some ancient Intel drivers
- Delicate ordering of calls *seems* to fix most cases

(Your) over-confidence is your weakness

- “Cool, the number of crash reports is manageable. How about non-crashing failures”
- Record stack-trace in constructor of our base exception class

Oooops.

100x increase ☹

Non-crash failures

- Downloaded files
  - Just broken (malformed XML)
  - replaced with firewall / proxy error HTML
  - appearing as 0-bytes
- UI leading to broken setups
- Gross configuration errors (non-parseable files)
- Non-supported OpenGL surface / texture formats
  - Quite a few of these do crash however :-)

DontReadMe.txt

- Content (aircraft) downloaded as a Zip
- Relative paths inside the content (textures, UI files) referenced relative to the directory name
- GitHub sets an automatic directory name
  - Based on the branch
- Download page, readme, etc:
  - ‘You must rename the directory to *foobar*’
- All files are ‘not found’

SQLITE_BUSY

- SQLite allows concurrent processes to access the DB
- Users (un-)intentionally launch multiple copies
- APIs return BUSY to indicate you should retry
  - Fine, DB exec wrapper does a loop+sleep+back-off
- Still getting occasional BUSY errors?!
- Query prepare call on startup can also fail

Old paths

- List of add-on paths
  - Added by user from file picker
  - Saved / loaded to persistent preferences
  - Validate paths on load
- GUI view of paths
  - Initialise from list of paths, skip missing paths
- Invalid paths persist internally forever ☹

Conclusions

Trend Analysis

- First version containing an issue is invaluable
  - Eventually ☹
- Correlation of tag data gives clues
- Uptake of new versions
- Session duration, % of failed sessions
- Other analytical data
  - Rapidly crosses into wider domains

Surfacing errors to UI

- Collecting errors makes it clearer which ones matter
- UX work to surface errors
  - Understandable
  - Actionable
  - Non-annoying
- Easier to justify to developers (or management) based on collated data

Privacy, etc

- Don’t want to record any personal data
- Use a UUID generated on first-run to cluster issues by user
- Sentry strips usernames from file paths, etc
  - Does not record IPs or even region
- First-run UI consents the crash-reporter

Missing Features?

- Sentry-Native can’t do user input on crash submission
- Questionable how much this would add
- Capturing last rendered frame would be great
  - Except for all the trouble it brings

Lessons learned

- Intuition is usually wrong
- Any reporting will be very informative
  - ‘Do something, and measure it’
- Iterative process
  - Add tracing data incrementally
  - Faster release cycle helps