

Planning for Debugging Day

Yaser Zhian

March 2010

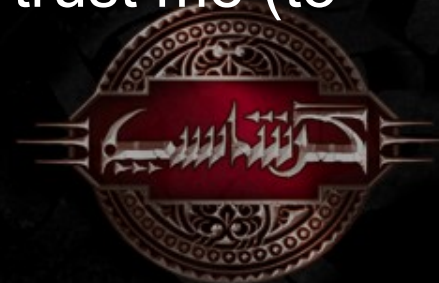
Fanafzar Game Studios

Programmer, “Garshasp” Project



What this is about

- The things that you should do at the beginning of a game (or similar) project to simplify debugging, adding features, optimization, build and release.
- We are talking about the technical and code aspect of this problem, but some of the ideas are generalizable to the rest of the team.
- I can only give you some tips, not a streamlined process. That's your job.
- Seasoned programmers and developers will definitely know what I'm talking about and will agree with most of this. Less-experienced programmers have to trust me (to some extent.)



Process, Process, Process (1/3)

- `Do { Conceptualize -> Build -> Import -> Integrate -> Test } while (0 == 0);`
- Iteration time is the most important and underestimated factor in game development process.
- Ideally, you should never have to exit the game to change anything (code, assets, configuration, scripts,...)
- More realistically, you should never have to exit the game to change anything except for code.



Process, Process, Process (2/3)

- Data-driven design and code is king!
- Keep engineers out of the loop. Seriously!
- Code reviews are good things, even informal ones.
- Agile philosophy is fantastic for small game developers.



Process, Process, Process (3/3)

- Automate as much as you can, as early as possible, and keep adapting. Tools, tools, tools!
- Transparency and discoverability
- Source control, asset control, versioning, branches, tags and issue tracking are not just fancy words!



Naming and Directory Structure

- A name is a unifying concept. Helps develop common views and more efficient and less error-prone communication.
- ??? (No one knows what you need!)
- When choosing a name, consider how that name looks to these entities:
 - Yourself, in 6 months
 - Other people
 - Scripts and tools



Build System (1/3)

- Quick, configurable, dependable, minimal.
- Think about header and source layout, very carefully and very deliberately.
- The compiler is your friend. Each compile-time error is potentially one less runtime or logical error you'd have to chase down.
- Ideally, wrong code should not compile. (Static assertion, type traits, template metaprogramming, type system design, etc.)



Build System (2/3)

- Code build time is important. Make it as short as possible.
 - Use precompiled headers.
 - Modularize your code (into libraries.)
 - Use abstract interfaces (but consider the performance costs of polymorphism.)
 - Eliminate unnecessary inter-class dependencies.
 - Use forward definitions.



Build System (3/3)

- Build variations
 - Full debug
 - Optimized debug
 - Profiled release
 - Developer release (with debug information)
 - Retail
 - and a couple more...



Code



Decisions to be Made

- Coding Style
- Exceptions
- RTTI
- Templates vs. Polymorphism
- Smart pointers vs. Handles vs. Raw pointers
- RAI
- Lifetime and Ownership Management (and a bag full of problems!)
- Intrusive code instrumentation
- String format
- and much much much more...



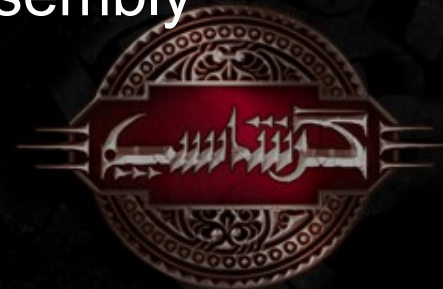
Love Thy Compiler (1/2)

- Use EVERY little obscure compiler and platform facility that can possibly help.
- But wrap them up, even if you don't care about portability (in case of a change in compilers/libraries, or if you want to disable a feature, etc.)
- Memory is NOT fast (except on an SPU!) Love your cache too.
- Don't be afraid of assembly. Learn to read through what your compiler produces. Love doesn't necessarily imply trust!



Love Thy Compiler (2/2)

- Learn all about compiler `#pragmas` and intrinsics.
- Use intrinsics instead of inline assembly, because `asm` usually hinders compiler optimizations.
- Learn about pointer aliasing, restricted pointers and every other hint that you can give the compiler to produce better (faster) code.
- Don't be afraid of assembly. Sometimes compilers generate really bad code. Hand-crafted assembly is and always will be an option.
- But not for the faint of heart. Don't drop to assembly lightly, specially on PC.



Type System (1/3)

- Consider implementing your own containers (STL is too frivolous with memory.)
- Type metadata and reflection
- Try not to use common type keywords (int, short, float, etc.) Typedef them all.
 - i32, u32, i16, u64, i8, u8, byte, ch8, ch16, ...
 - f32, f64, v128
- Or use `stdint.h`, which is a standard header but missing from Visual C++ (still!) There are free and conforming implementations available though.



Type System (2/3)

- Types that are equal physically, but not semantically:
 - Use different classes with no assignment operators (or at least correct ones.) For example: Degree **vs.** Radian, NormalizedVector3 **vs.** Vector3
 - Hungarian Notation (commonly misapplied, e.g. in Win32 headers and docs!)
 - `degAngle = radAngle;` is most probably wrong and the good thing is that it indeed looks **WRONG**.



Type System (3/3)

- Strings
 - ASCII is dead (mostly!)
 - At least differentiate between these usages:
 - File names and paths.
 - User input and text shown to the user.
 - Internal entity names, IDs and the like.
 - Text file and script contents, etc.
 - And many more.
 - Realize that some strings are (mostly) constant during their lifetime. (Optimization opportunity!)



Pointers

- Don't use `T *` to represent a pointer to `T`. At least `typedef` something like `TPtr` and use that instead.
- Don't use `T *` to represent an array. Use a `struct` with size information (`templated` for arrays with known sizes at compile-time, or a member `size_t` (or smaller data if suitable) if not.)
- `NULL` is not the only invalid pointer value (`0xcdcdcdcd`, `0xcafebabe`, `0xdeadcd0de`, ...) Don't use comparison with `NULL` to detect this. Use a macro.
- Don't just use `NULL` for passing optional values. You can use `boost::optional`, or at the very least abstract out the concept.



Standard Library Independence

- File Handling and asynchronous I/O
- Directory handling
- Random numbers
- Math
- Time handling
- ...



Debugging

Consider the idea that the purpose of writing code is not running it. Write code for debugging.

- Logging
- Assertion
- Context metadata
- Crash handling and crash dump
- Visualizing data



Debug: Logging

- Do NOT overwhelm!
- Collect as much as possible, show as little as needed.
- A good logging library should be:
 - runtime controllable
 - able to send logs to multiple targets
 - able to filter what is sent where based on audience, source, target, etc.
 - fast, fast, fast
 - as non-intrusive as possible
 - Employed from day 0.



Debug: Assertion

- You do use them, don't you?!
- Never use the standard assert!
- Assert whenever you can, but nowhere you shouldn't. (Hint: don't assert when a configuration file is not found; make a new one!)



Debug: Context metadata

- Should be totally and partially disable-able.
- Including, but not limited to:
 - Machine name, user name, code revision, build number, build parameters, time, environment, frame number, etc.
 - Where does the execution flow come from? (Stack trace)
 - Where are we now? File, line, function, instance.
 - Which subsystem called this piece of code? Graphics? AI? Animation? Sound?
 - Why did it call this? Was it initializing? Loading data? Looking for a collision? Walking the scene graph? Visiting AI actors?
 - How long did the function take to execute?
 - How Many times was it called in this frame?
 - How many memory allocations did it perform?
 - ... (use your imagination!)



Debug: Crash handling and crash dump

- If you have to fail, fail as soon as possible and as loud as possible.
- Gather as much data as you possibly can: place, time, what happened before, what was going to happen after, context, hardware environment, software environment, user settings, etc.
- Use a “Black Box” type construct. Collect some data all the time, but store it only when crashing.
- Consider automatically submitting this info to your issue tracker (during development) or emailing it to your team (after release.)



Debug: Visualizing Data

- There is much more going on in a game than graphics (which can be seen already!)
 - Lots of data involved.
 - No clear picture from mere numbers.
 - Hard to understand in a single slice of time.
- Add visual representations for non-visual objects: physics, bounding volumes, AI algorithms, behavioral decisions, etc.
- Real-time graphs and bars too!



Memory

- NEVER use naked new/delete in real code.
- NEVER EVER use naked new/delete in real code!
- Write your own memory manager, even if it is just a wrapper over new/delete.
- Don't forget malloc() and friends.
- Don't forget third-party libraries.
- Always be conscious about who uses how much memory. Always.
- Always override the memory management in third party libraries (including STL) unless you have reason not to.
- Writing a memory bug detector is quite easy. Write one or find one.
- Memory allocation is usually a source of performance bottlenecks as well as bugs.



In-game console

- Useful for debugging and interactive experimentation.
- Tied-in to the scripting system.
- Make it like a good OS shell.
- Can think about it as a higher-level debugger (inter-object relationships, etc.)
- Give access to everything you can, but consider that the designers and artists will have to use it too.



Serialization

- Useful for much more than saving/loading or network communication.
- Template magic? Some form of IDL? Übermacros?
- Consider two interchangeable formats: one optimized binary and one human readable.
- Make everything you can serializable.
- Consider interactions with source control.
- Plan for evolution of data structures over the course of development.



Runtime Asset management

- Everything other than code that constitutes the game.
- Consider going beyond simple files.
- Consider treating asset management more like writing a DBMS, than a file system.
- Consider integrating asset fingerprinting, change management, versioning, change detection, hot reloading, etc.
- Consider asynchronous operations.



Time

- Do NOT underestimate time management.
- Never just use direct calls to OS to get the time for game subsystems.
- Familiarize yourself with higher-resolution timers (`RDTSC`, `QueryPerformanceCounter()`, `gettimeofday()`, ...)
- Familiarize yourself with multi-core issues, power management issues, Windows issues(!), user issues,...
- Consider the difference between wall clock, game clock, camera clock, level clock, animation clock, etc.
- The more time you put in time, the better the times you'll have when the time of debugging time comes!



Screen capture and Game Replay

- Capturing still images is a must.
- Capturing video is a big plus.
- Capturing metadata (DEM files, anyone?) to be able to replay a scenario is a huge plus.
- The game replay subsystem can be implemented with varying levels of complexity and functionality. Even a simple system can have considerable benefits.



Runtime Statistics

- Please invest in this!
- You must know your own game. This is one very good way.
- Anything from frame-rate and tri-count to number of memory allocations and file accesses per frame, per second or per-level.
- Find interesting ways to display these data.



Automated Testing

- Full unit testing is the Holy Grail!
- Testing of games and features is usually very time consuming. Any automation helps.
- Any change in code or assets can have undesired side-effects: regression testing.
- You can even have automatic screen-shot comparisons!
- Auto test non-visual parts, if you can't manage more sophisticated methods.



Questions?

"I know that you believe that you understood what you think I said, but I am not sure you realize that what you heard is not what I meant."



We are hiring!

jobs@fanafzar.com

