# Random Things About Code

Unity Training Academy 2018-2019, #2
Aras Pranckevičius




Generative Art – Made with Unity

## Caveat Emptor

- This is going to be totally random!
- Without any structure!
- ...so yeah :)

# Random Intro

A long time ago in a galaxy far, far away...

- I thought these are awesome:
  - C++
  - STL, Boost
  - Object Oriented Programming
  - Design Patterns
- Had hair!

unity

Now...

- Most of these are... ”not that good”
  - C++: too complicated
  - STL/Boost: too complicated, over-engineered
  - OOP: only useful for UI widgets
  - Design Patterns: BS to sell books/courses
- No hair :(

Some things which I like: Futurist Programming

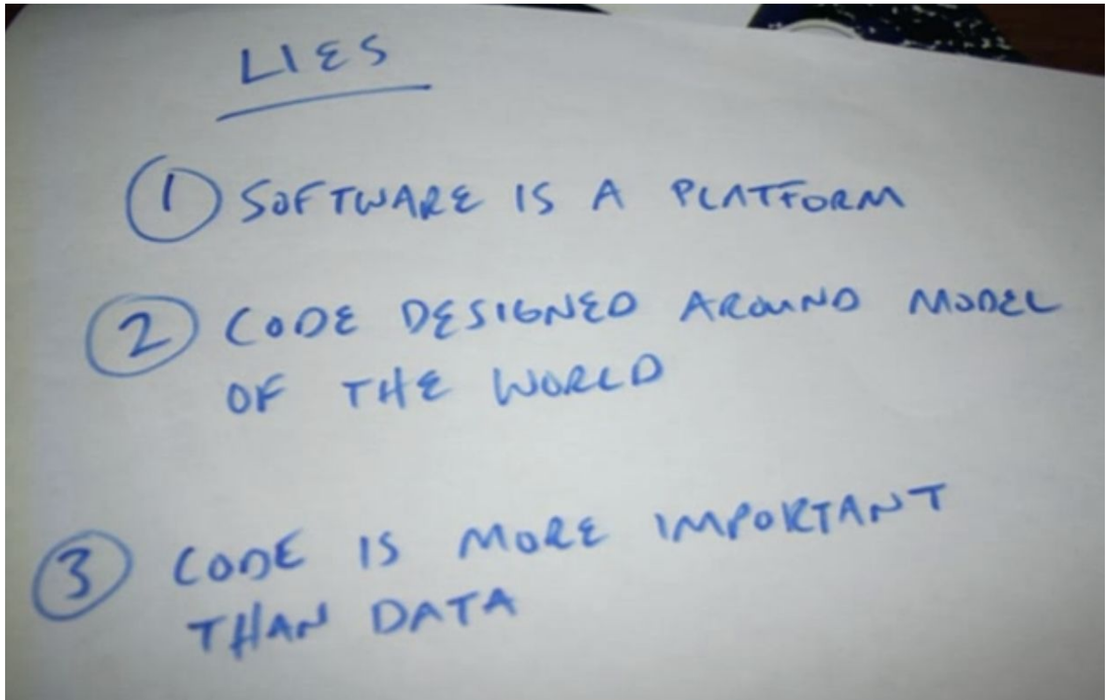
- “Futurist Programming” by Nick Porcino
- http://nickporcino.com/meshula-net-archive/posts/post168.html

Some things which I like: Futurist Programming

- No Compromise, No Waste Programming
  - The program is the BEST at what it does
  - The program is FAST
  - The footprint is SMALL
  - The code is CLEAR
  - The program is BUG-FREE
  - Abstractions must SIMPLIFY
  - The unnecessary is ELIMINATED
  - NO COMPROMISES in the name of Extensibility, Modularity, Structured Programming, Reusable Code, Top Down Design, Standards, Object Oriented Design, or Agility.

Some things which I like: Three Big Lies

- “Three Big Lies” by Mike Acton
  - https://www.gdcvault.com/play/1012200/Three-Big-Lies-Typical-Design
  - https://cellperformance.beyond3d.com/articles/2008/03/three-big-lies.html
  - Fun fact: Mike is at Unity now, working on ECS and stuff!




unity

Some things which I like: Three Big Lies

- Software is a platform
- Code designed around the model of the world
- Code is more important than data

Code is just means to solve a problem

- “Solve a problem” is key
  - What *exact* problem is your code solving?
  - Is that an *actual* problem someone has?
- Don’t get too attached to your code (or anything…)
  - Some lovely code will get thrown away
  - Some nasty code will live forever

# “Future Proof”

Prediction is very difficult, especially about the future.

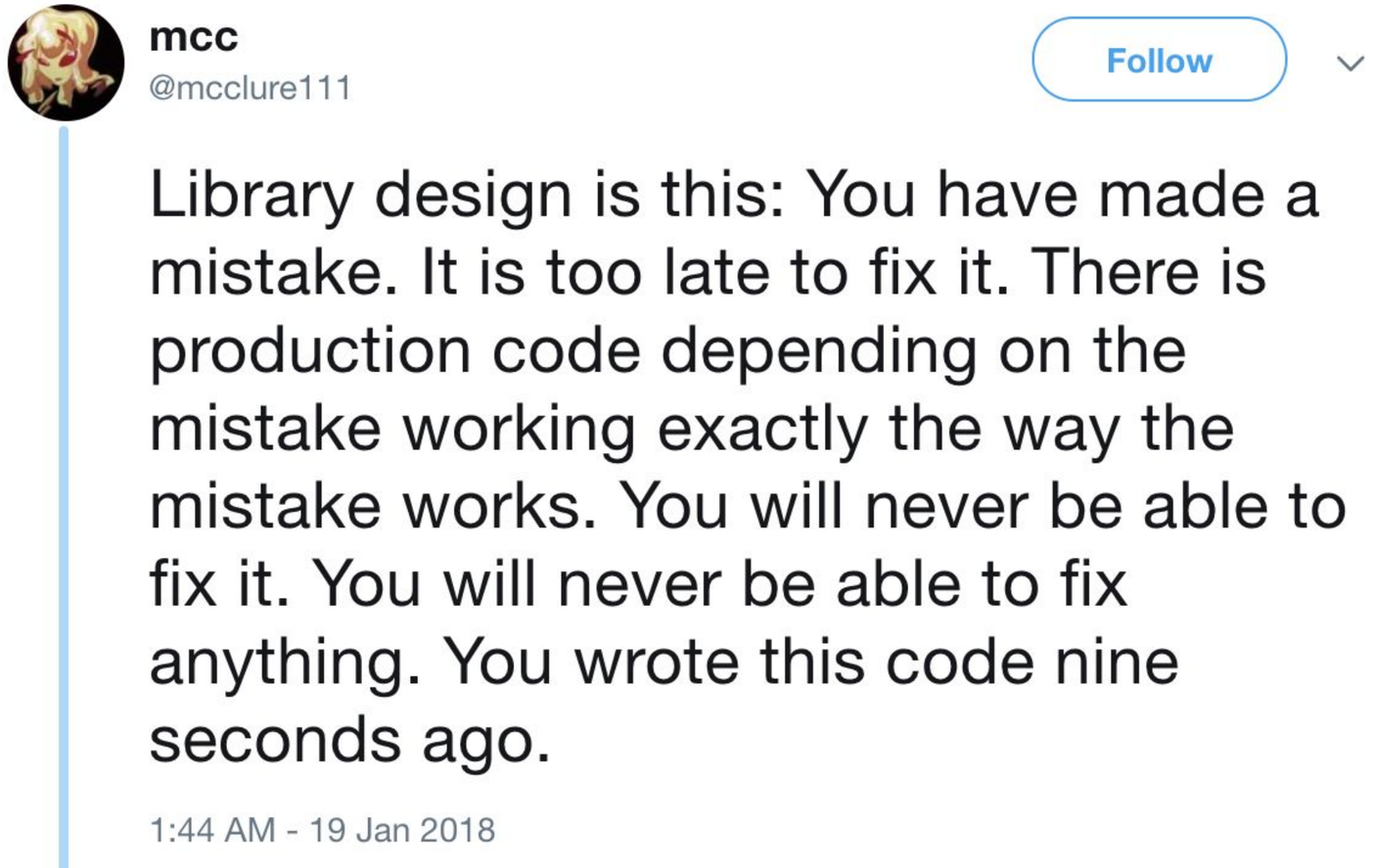
Niels Bohr

## Plan ahead just enough

- Enough to not put yourself into a corner
- Simplest thing that solves *today's* problem
- It can and *will* change in the future
  - Often in ways you could not have predicted

Exception: Public APIs

- Public APIs of a platform (like Unity) live 10+ years
  - https://twitter.com/mcclure111/status/954137509843398656

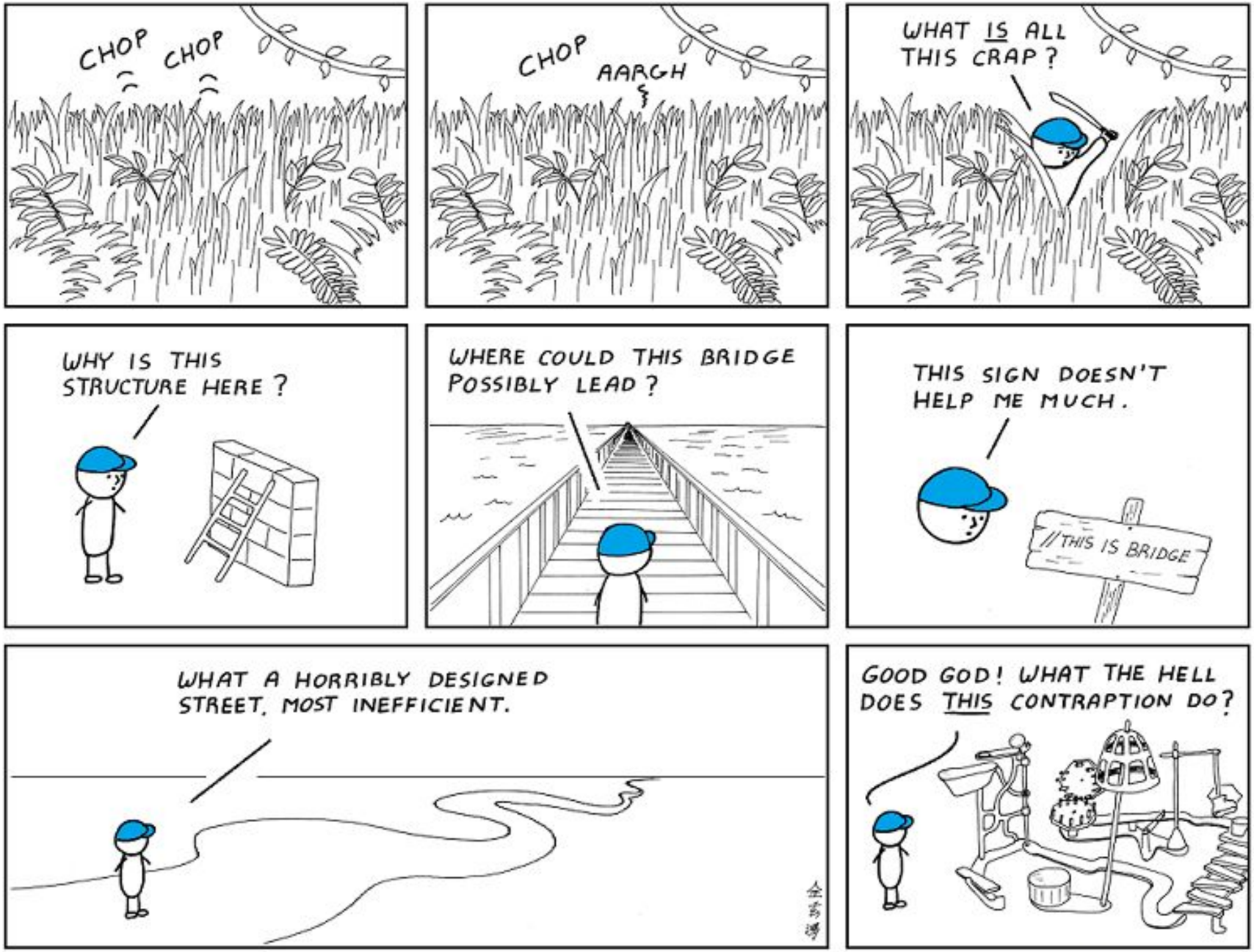


Future Proof is wishful thinking

- I don’t think I ever saw “future proof” plan work out
  - Often you don’t know future requirements
  - “I’ll make a renderer interface and implementation for DX11, and later on will learn Vulkan and just make an impl for that”
  - If you don’t know Vulkan yet, you have no idea about proper interface

Future Proofing result is often this

- https://abstrusegoose.com/432



## When should I build an abstraction?

- “When you have three things” is good rule
  - Have done 3 things separately,
  - Suspect they might have something in common,
  - Factor out common functionality/interface/…
- Duplicating code is sometimes ok!
  - http://bitsquid.blogspot.com/2011/01/managing-coupling.html

# Navigating large codebases

Large codebases often are...

- Fairly old (Unity: some parts 14yo)
- Little or no documentation
- Grew organically
- Some places no one remembers what/why/how

Reaction can often be “aaarg what is this?!”

- Most of it is there for a reason
- Tempting to say “this sucks, burn it, start over”
  - Often not a good idea
  - It *must* be solving some problems quite well,
  - ...or otherwise you would not be working on it!
- Maybe you would have done it differently
  - Original authors would have done it differently too!

Assume authors are not stupid

- If something looks strange/weird/wrong:
  - 30% there is a good (non obvious) reason for it
  - 30% there *was* a good reason for it
  - 30% there is no good reason, and code is indeed stupid
  - 10% Ph'nglui mglw'nafh C~~thulhu~~ R'lyeh ~~wgahnagl fhtagn~~

Don’t go cowboy refactor/cleanup

- My messup on PSI: Syberian Conflict (2005)
  - Contracted on physics/explosions system
  - Saw a bunch of “ugh” code
  - Refactored/cleaned up the heck out of it
  - Did not realize I was only working in “test” solution without rest of game
  - Broke builds for everyone else
    - *(this was 2005, we had no branches or CI)*

## Figuring out a small thing

- Pick a small thing (feature etc.) to figure out
- Search whole codebase for API/message/…
  - Search in whole solution in IDE
  - ripgrep https://github.com/BurntSushi/ripgrep is insanely fast
- Breakpoint in debugger
  - Step through/into from there
  - Take notes and build a mental picture of things

- If you get stuck, *do* ask!
  - First spend 10 minutes figuring it out yourself of course
  - Don’t get stuck for longer than a day
  - Many people happy to answer questions & explain things
  - Protip: Ctrl+K in Slack, type topic, find channels that sound related

- Pick a larger system to build high level view of
  - Can you figure it out from file/class/functions/interface layout?
  - Does it map well to common approaches?
    - Game Engine Architecture book
    - Real-Rime Rendering book
    - Real-Time Collision Detection book
  - Are there any docs, talks, presentations about it?
  - Ask someone to walk you through it (~1 hour)
    - Again, many people are happy to!

- Debugging is finding where/why a problem is
  - “What could possibly have caused this?”
  - “What is different in this case vs the one that works?”
  - Binary search / Divide & conquer
  - Hypothesis, test, repeat
  - Source control *(next)* can be useful

- Use Source Control effectively
  - Don’t mash up unrelated changes
  - Write *detailed* commit messages, explaining *why*
    - “Fixed some stuff” *will* bite you in the arse 4 years later
  - Learn VCS annotate/blame/log functions
    - Who changed this, when and why? Can you ask them?

- Having automated tests will save you 1000 times
  - 3 months/years in the future when changes have to be made
  - It might not even be *you* making changes
  - ...or it might be you, having forgotten *everything* about the code

## Useful workflows: One Thing At A Time

- Small enough “I’m making progress” tasks
  - 1-2 hours each; put them into Trello/Favro/Stickies/Dropbox
  - Helps with accomplishment & focus
- When stuff does not work:
  - What *exactly* have you changed from when it was working?
    - Version control with small, isolated changes!
  - Look at that instead of randomly plowing forward

## Useful workflows: Know Your Tools

- Learn a good IDE/debugger
  - Including plugins that might help you (Resharper, VisualAssist)
  - Keyboard shortcuts
- Learn other tools
  - Source control, profilers, grepping, regex, …

# Navigating a large organization

## Navigating an organization

- Companies are structured *very* differently
- Culture within a company can be different too
  - Or in different locations / departments of the same company!
- Unity (in R&D) is fairly flexible, relaxed, not overly hierarchical, and can feel chaotic at times
  - The chaos allows awesome things to happen, but can be intimidating or confusing

- Know your team *(well duh)*
- But also wander off into other areas
  - e.g. join other Slack channels of interest
  - Answer questions you know the answer to
  - Participate in discussions

- Interact with people
  - Hackweeks, Unites, team offsites, townhalls, …
  - Yes, it can be hard for many of us
  - Often worth the effort though

- Make you/team/work be visible
  - Did something cool? Tell others about it
    - Some teams work on years on great stuff, and never send an update “hey look, we did this!” to anyone
  - This does sound like marketing, because it is
  - *“Others know about you”* is much better than

    *“No one has any idea you even exist”*

# Ask me 5 or more questions!