Designers are from Venus, Programmers are from Uranus

Designer/Programmer Interaction

Who and What am I?

- sports, m
- Gameplay, AI, Design

Why This Talk? …Process

Old School design loop:

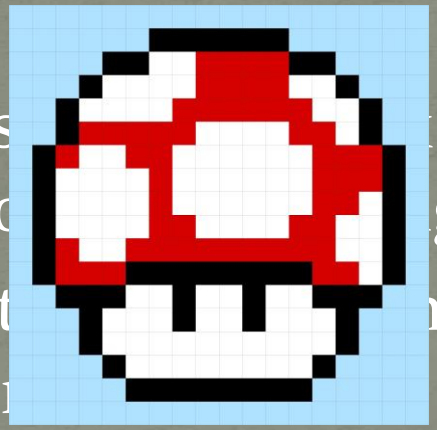
- Design writes specifications doc
- Programmers go off and implement
- Playtest
- Repeat, maybe

New School design loop:

- Systems designer writes doc
- Programmers implement the tool content designers use
- Programmers support designers with tool and feature changes
- Rinse and repeat until ship. Then keep doing it

Why This Talk? …Features

- Typical roshambo: designers, programmers, artists
- Staff is separated by career tracks, game features are not
- Increas games
  tight co gn anc
- These t be esp
  harmo



Things to Cover

Basic differences

- Process differences
- Work motivations
- Measure of succes

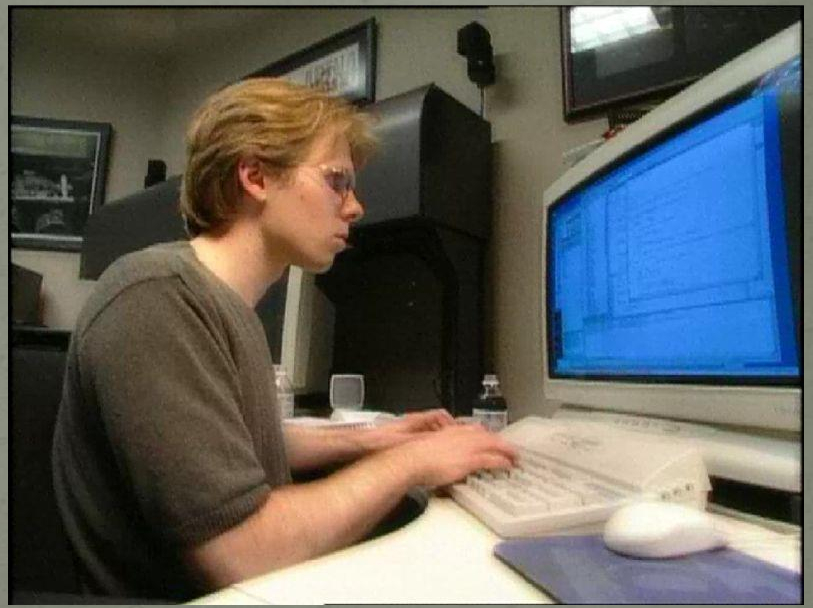
Working together

- Establishing Connection
- Communication
- Ongoing techniques

Bridge Methods

- Technical Designers
- Dual class CD

Basic Differences



Differing Process

Design

Free form
Collaborative
Largely discovered
Very situational

VS

Programming

Structured
Abstract
Systematic
Typically more isolated

Mixed Team?

Differing Goals and Motivations

Design

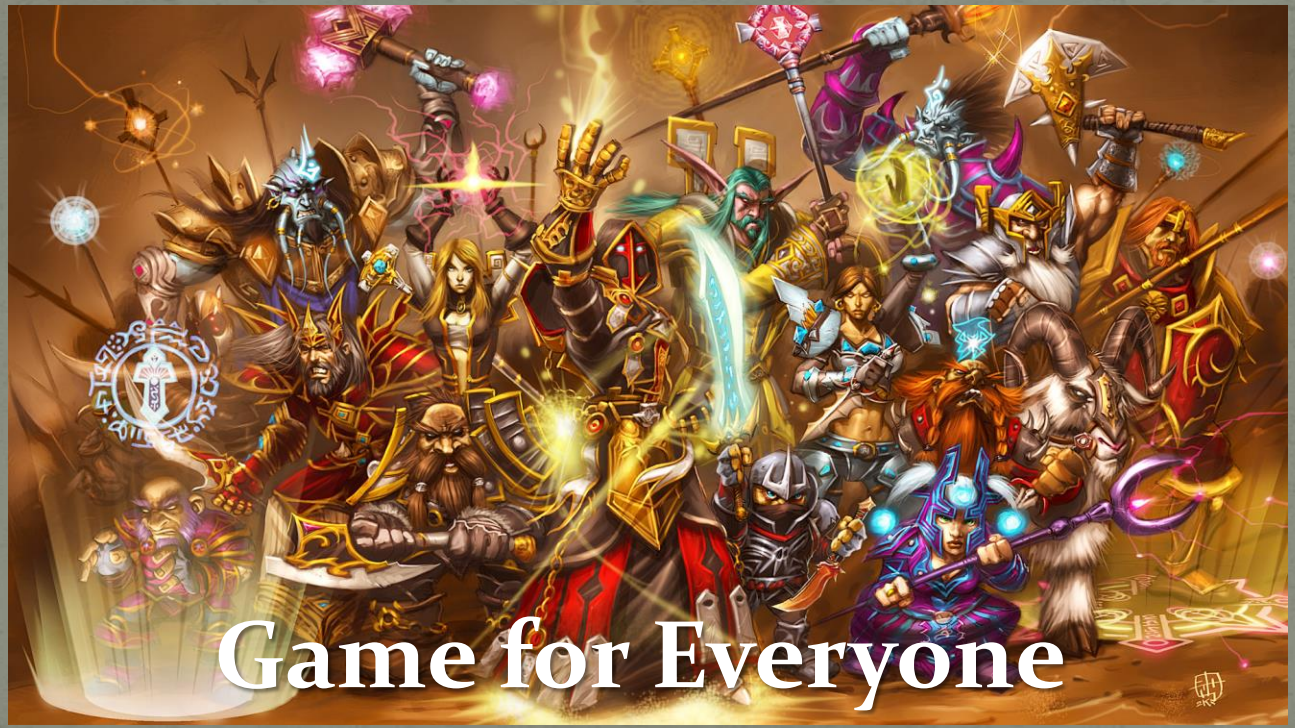
FUN
A consistent player contract
Simplicity

VS

Programming

Performance
Scalability
Maintenance
Simplicity

Design Pillars



Design Pillars

Design Pillars

Design Pillars

Design Pillars

Design Pillars

Programming Pillars

Solve the Problems You Have

An engineering tendency is to expand the requirements of a problem to solve other problems that we can imagine we might have one day. In many cases this is unnecessary. The original problem might get redesigned. The feature might get cut. Sometimes the added requirements are good, but the implementation was too early to know how it really should have been done.

Solve the problems you have. Wait until you know you need a feature before you implement it. The code will be faster, smaller, done sooner, easier to understand, and easier to maintain.

Fast Doesn't Mean Sloppy

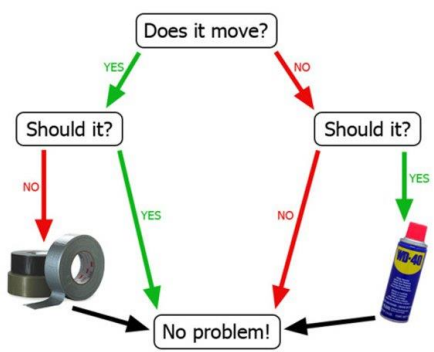
You can spend two weeks to do it right, or one week to do it fast and then two more weeks fixing the bugs over the next six months. Spend the two weeks.

"Hack" is not a dirty word. A hack is a robust solution constructed with very little code. It might not be as fast as we need it some day. It might not deliver all the "would be nice" features. However, what a hack does, it does well. The whole point is to not support it again until some time in the future, so if it's not robust or does not do what it's supposed to do every time then you've defeated the purpose of taking the expedient path.

Programming Pillars

Keep It Simple

Know How It Broke



Sometimes a bug seems to magically go away. This doesn't mean it actually did. If we don't know how it happened in the first place then we can't be sure it won't come back, or that we have the proper solution.

If a random change fixes the bug, revert the change until you know *why* it fixes the problem.

If a bug magically goes away, trace the previous revision until you know what should have been done to fix the bug, then compare that to what was actually done and see if it was the right fix.

Programming Pillars

It Better Work

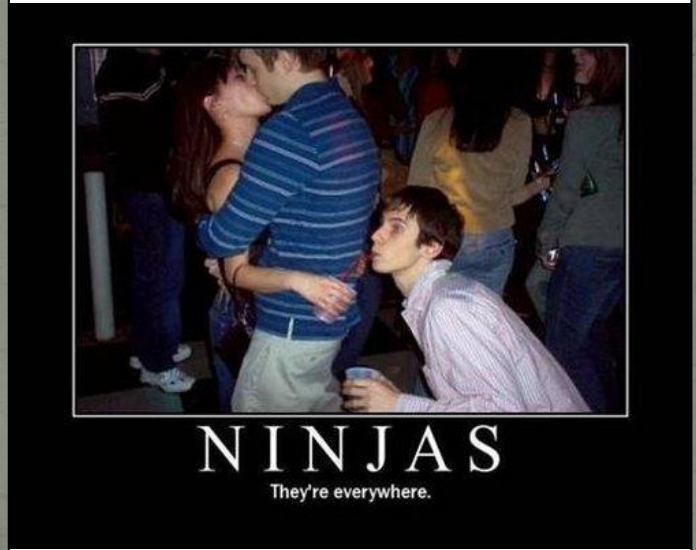
Compile it.

Run it.

Use it.

Don't give it to someone else unless you're sure it compiles, runs, and performs the way it's expected to perform. If your recipient doesn't know your system then committing incomplete or incorrect code wastes time as they try to debug what you should have already debugged.

Leave No Tracks



Everything we write should look like one person wrote it all.

Programming Pillars

Play the Game You're Making

We make games we want to play, and we play the games we make. We build a game to be as fun as it can be for us, and we assume anyone else like us will like the game the way we've made it.

Sometimes our tasks seem to bury us, but we have to use the product to engineer it properly, and to give feedback to the designers for things we could engineer better. Always make time to play the game regardless of your workload.

Use the Right Tool

Every tool has things it does well. Every tool can be used wrong.

Use the right programming language, the right algorithm, the right data structure, the right amount of customizability, the right amount of flexibility, the right balance of speed optimization versus memory optimization.

Know who and what you're supporting. Know your minimum requirements. Know how long you expect to use the system. All of these things help determine the right tool.

Goals and Motivations → WIN?

Design

FUN
A consistent player contract
Simplicity

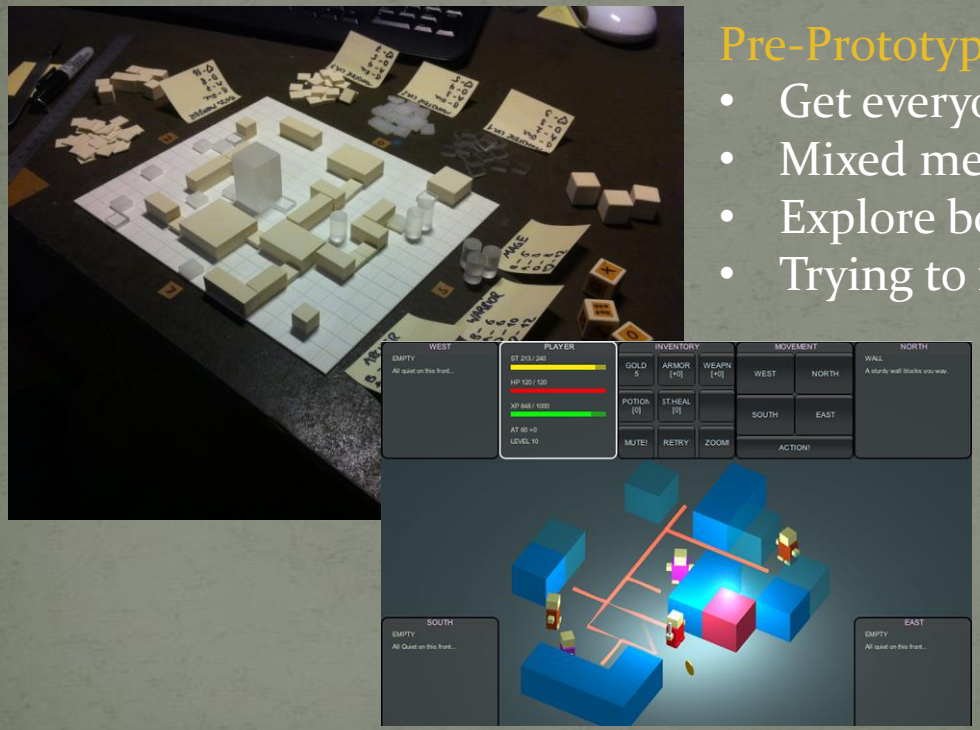
VS

Programming

Performance
Scalability
Maintenance
Simplicity

Working Together

In the Beginning



Pre-Prototype

- Get everyone on board and on same page
- Mixed media
- Explore boundaries of design space
- Trying to find exceptions creates clarity

Prototype

- Pushback on abstraction
- Now have more info about performance, etc
- Watch for overengineering and possible trust fails

Establishing connection

Communication Clarity

- Vocabulary
  - Consistency is king
- Naming systems
  - Allowing for flexibility at definition
- Be EXPICIT
  - Priorities:
    - Must Have/Would Be Nice
    - Post Ship
  - Estimates
    - Don't give snap estimates
    - Ranges show certainty

Other Communication

- Meetings
  - Work chunk size
  - Social setting
- Ongoing communiqués
  - Email/IM/phone/in person
  - Times of day, “focus time”
- Feedback and suggestions
  - Foster an “open for ideas” zone
  - What are you trying to fix?

Other Communication

- Documentation
  - Design docs
  - Process
  - Content/Scripts
- Bug reports
  - Everything is broke, Nothing Works

Trouble spots

- Work flow
  - Personal preferences
  - Pet peeves
- Discovery
  - Big changes
  - Reprioritization
- Crunch

Trust

- The “contract”

Bridge Methods

Technical Designer

- Needs twice the mentoring/management.
- Can suffer from “green programmer” problems
  - Unapproved checkins, hacks, easter eggs
  - Poorly optimized code or potentially exploitable
- Take care with overlapping influence
- Should not be the main designer. Dual nature makes them not focus as shrewdly on player’s needs
- ABSOLUTELY THE FUTURE OF GAME DEVELOPMENT
  - sortof

Dual class Creative Director

- Knows enough to push back on both sides, and provide mediation
- Ownership
- Need to manage both sides

Points to ponder

- Programmers and designers are very different people
- Best systems come from melding design and engineering
- Build a strong developer relationship
  - Communication is the heart
  - Trust is the soul
- Cross discipline staff can be valuable, if…
- Remember, we all want the same thing, to make something great

Questions?

Brian.Schwab@gmail.com