Iterative Development

Motivation

- Last thing you want to do is write critical code near end of a project
  - Induces huge stress on team
  - Introduces all kinds of "interesting" bugs that break working code
- Testing always gets cut in crunch
  - Makes problem even worse!
- *Planning* can help avoid writing critical code in alpha or beta phases

Wishes versus Reality

- Most games you make are smaller/less than you originally envisioned
  - Design was bigger than implementation
  - Or, tested/working implementation bigger than what made it into game
- That's ok → expect it
- So, how do we know when a game is "done"?

How Do We Estimate Progress?

Example:

- Jo is a programmer
- She estimates it will take 10 days to implement Smart Trap
- She is 4 days into implementation
- Is Smart Trap 40% complete? … maybe
  - We may not see it "snap shut" until day 9
- She's good, → finishes in 8 days total
  - Yay, we are ahead!
- Later, decide to add functionality to Smart Trap (e.g., trap large bad guys, too)
  - Takes 4 days
- Boo, now we're behind!

What's the Point?

- Most things get revisited multiple times during development
  - Fix bugs, modify functionality, etc.
  - "Refactoring" your code
    - Note, refactoring easier with clear, easy-to-understand code!
  - Expect this! Despite your careful planning …
- So, the "40% done" estimate looks pretty sketchy…
- Need way to account for time without driving project into trouble (and into panic)

Incremental Delivery

- Milestones are good things!
  - They let us get things "done"
- Milestones can have downside
  - If you miss one, people notice, action taken
  - Especially management people
- Developer's view
  - Milestones (or plans, in general) are just best guesses for how implementation will evolve
- Management's view
  - Schedules are contracts with developers
  - *Promising* certain things at certain times
- Different views cause problems
  - Developers: panic, pressure, long hours
  - Managers: justification for financial pressure

Milestones (1 of 2)

- Despite problems, necessary
  - Without milestones, unlikely to get done
- Unrealistic milestones mean work not done on time, no matter financial importance
  - Remember, are *best* guesses
- Managers need to know estimates of developers and key makers along the way
  - Plan financial/time links accordingly
- External milestones coarser
  - Tie to publishers, marketers, etc.
- Internal milestones have finer granularity
  - Used by team members

Milestones (2 of 2)

- Think of development plan as black box
  - Managers have specific "interface" to box
    - "Give me the latest build"
    - "Give me the latest (high-level) schedule"
- Clearly, this is too simplistic/wishful thinking
  - Managers just want to know more (and need to, to do their jobs better)
- But view as development plan as "black box" helps separate job roles better

There is More than Meets the Eye

- For many, "if I can't see it, it is not important"
  - AI takes time to build (and you don't see it)
  - Network code to balance players is an optimization (and you don't see it)
- Developers receive less "credit" for unseen code than for things that can be seen
- Good managers will probe deeper to see what is really going on
  - Requires technical ability (knowledge)
  - This is one reason Game Designer needs technical knowledge!

Iteration

- Make frequent working builds
  - "We don't go home Friday until a working build checked in."
  - Frequency (daily or weekly) depends upon project
- If management asks for latest build, give one from last week
  - Resist desire to show latest-and-greatest
    - Won't always be bug free, ready to show
  - People will always expect it and leads to unrealistic expectations

Internal Scheduling

- Give detailed design document
  - Make list of all objects (e.g. players, items, NPCs…) that need to be built
  - Mark each as *one* of:
    - Core – base, fundamental functionality
    - Required – needed for working, playable game
    - Desired – icing on the cake, make game special but not required
- End result:
  - List of features sorted by importance
- Note, doing this planning gets easier the more you do!

Internal Scheduling Structure

- Could start from top of milestone list → Work down and when time runs out, then done
  - Produces whole lot of "complete" pieces, but no whole that works together
  - Makes management (and others) nervous since cannot see it "coming together"
- Better way → since list made in Object-Oriented (OO) fashion, start building objects!

OO Iterative Development – Object Versions (1 of 2)

- Create a *Stub* version of each object
  - Complete, but empty
  - Perhaps just print out message
- *Basic* version
  - Placeholder with some properties present
  - Set attributes, minimal functionality
- *Nominal* version
  - Commercial viable implementation
  - Most functionality in place
  - Tested
- *Optimal* version
  - State of the art
  - All polish present
  - Thoroughly tested

Stub

```
// Player.h
class Player {
  public:
    Player();
    ~Player();
};

// Player.cpp
#include "Player.h"
Player::Player(){ }
Player::~Player(){ }
```

Nice feature about above development plan? Game will "build" even after Basic version!

OO Iterative Development – Object Versions (2 of 2)

- Some objects (classes) will be simpler
  - Fewer iterations (e.g. `Position` class)
- Some will be more complex
  - More iterations (e.g. `WorldManager` class)
- Can say have shippable game when every object at least in *Nominal* version
  - Working definition of "Good Enough"
- A complete game is one where all objects are at *Optimal* level

OO Iterative Development – Overall

- But, seems like need to write 3 versions of every object!
  - Yes, but would probably do that anyway with revisions
- Approach
  - Starting with core, then required, then desired, implement *Stub* versions of all objects
  - Starting with core, then required, implement *Nominal* versions
  - Code is now *releasable*
- Only now start to work on desired
- This is breadth-first approach
  - Better than "let's do the cool bits first!"
  - Always have build-able game
  - Near-continuous growth
  - Can easily show refinement
  - Throughout, better handle on how "complete" game is

Scheduling - Naive

	Feature	Null	Base	Nominal	Optimal
Core	F1	1	13	25	37
	F2	2	14	26	38
	F3	3	15	27	39
	F4	4	16	28	40
Required	F5	5	17	29	41
	F6	6	18	30	42
	F7	7	19	31	43
	F8	8	20	32	44
Desired	F9	9	21	33	45
	F10	10	22	34	46
	F11	11	23	35	47
	F12	12	24	36	48

Scheduling – Better (single programmer)

	Feature	Null	Base	Nominal	Optimal
Core	F1	1	13	22	37
	F2	2	14	23	38
	F3	3	15	24	39
	F4	4	16	25	40
Required	F5	5	17	26	41
	F6	6	18	27	42
	F7	7	19	28	43
	F8	8	20	29	44
Desired	F9	9	21	32	45
	F10	10	30	33	46
	F11	11	31	34	47
	F12	12	35	36	48

- Note! This is just one example → Alternate could be to finish Core Nominal before Base Required
- Point is to "zig-zag" to bottom corner, with optimal last

Scheduling – Better (multiple programmers)

	Feature	Null	Base	Nominal	Optimal
Core	F1	1A	7A	11B	19A
	F2	1B	7B	12A	19B
	F3	2A	8A	12B	20A
	F4	2B	8B	13A	20B
Required	F5	3A	9A	13B	21A
	F6	3B	9B	14A	21B
	F7	4A	10A	14B	22A
	F8	4B	10B	15A	22B
Desired	F9	5A	11A	16B	23A
	F10	5B	15B	17A	23B
	F11	6A	16A	17B	24A
	F12	6B	18A	18B	24B

Team Work

- Make sure to use skills of each team member well
- Keep everyone busy
  - No waiting, if possible
- Communication vital!
  - Every programmer should be aware of what others are doing
  - Code reviews (for sharing implementation details as much as writing solid code)
  - Joint status meetings (*Daily*! Even if brief)
  - Documentation (documented code, documented milestones and status, documented bug list)

Scheduling with Iteration

- Shift:
  - FROM: When will it be finished?
  - TO: When will it be good enough?
- "Finished" is meaningless, anyway
- Have definition of "good enough" now!
- Bad estimation often comes from top-down dissection
  - No accounting for learning curve, code revision, or integration
- Iterative development
  - Total time equals sum of the Stub, Base, Nominal, and Optimal levels

Consider Saucer Shoot

- Core
  - Ability for player to move ship, fire bullets
  - Saucers move
- Required
  - Bullets destroy saucers
  - Saucers respawn
  - Explosions
  - Animated Sprites
  - Game difficulty progresses and game ends
- Desired
  - Stars
  - Game start screen, game end screen
  - Score

Consider Dragonfly

- (Note, your development did separate 2a, 2b, 2c "mini-projects")
- Core
  - Log file management
  - Game loop with timing
  - Game objects with updates
- Required
  - User-input
  - User-defined events
  - Graphics support
  - Collisions
- Desired
  - Animated Sprites
  - Efficient scene management (e.g. for collision detection)
  - Multi-colored Sprites
  - Camera control

Is "optimal" done for any classes?
Probably not – would need all Desired done first!
But have *Nominal* version of classes for all

Group Exercise

- Split into Project 3 Teams
- Make list for *your* game, with one feature in each list
  - Core
  - Required
  - Desired
- Provide high-level class name(s) associated with each