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 \(\Rightarrow\) 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, \(\Rightarrow\) 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 (as in add functionality to Dragonfly)
    – Note, refactoring easier with clean, 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
  – Guidelines for when certain things will be ready
• 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 of 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”
  – At 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 also 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

```cpp
// 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 – 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 nominal versions of desired
- This is breadth-first approach
  - Better than "let's do the cool bits first!"
  - Always have buildable game
  - Near-continuous growth
  - Can easily show refinement
  - Throughout, better handle on how "complete" game is

Scheduling - Naive

<table>
<thead>
<tr>
<th>Feature</th>
<th>Null</th>
<th>Base</th>
<th>Nominal</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>F1</td>
<td>11</td>
<td>22</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>F2</td>
<td>14</td>
<td>26</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>F3</td>
<td>12</td>
<td>24</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>F4</td>
<td>24</td>
<td>35</td>
<td>47</td>
</tr>
</tbody>
</table>

Scheduling – Better (single programmer)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Null</th>
<th>Base</th>
<th>Nominal</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>F1</td>
<td>1A</td>
<td>7A</td>
<td>11B</td>
</tr>
<tr>
<td></td>
<td>F2</td>
<td>1B</td>
<td>7B</td>
<td>12A</td>
</tr>
<tr>
<td></td>
<td>F3</td>
<td>2A</td>
<td>8A</td>
<td>128A</td>
</tr>
<tr>
<td></td>
<td>F4</td>
<td>2B</td>
<td>8B</td>
<td>120A</td>
</tr>
</tbody>
</table>

Scheduling – Better (multiple programmers)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Null</th>
<th>Base</th>
<th>Nominal</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>F1</td>
<td>5A</td>
<td>11A</td>
<td>18B</td>
</tr>
<tr>
<td></td>
<td>F2</td>
<td>5B</td>
<td>11B</td>
<td>182A</td>
</tr>
<tr>
<td></td>
<td>F3</td>
<td>6A</td>
<td>16A</td>
<td>178B</td>
</tr>
</tbody>
</table>

**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.*
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)
  - Peer-debugging (as needed)
  - 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
  - Saucers move
  - Player can move ship, fire bullets
  - Allows you to refine interface mechanic early!
- Required
  - Bullets destroy saucers
  - Saucers respawn
  - Explosions
  - Animated Sprites
  - Game difficulty progresses and game ends
- Desired
  - Stars
  - Game start screen, game end screen
  - Points

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, colored Sprites
    - HUD objects
    - Camera control
    -Efficient scene management (e.g. for collision detection)

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