Chapter 3.1
Teams and Processes

Being a programmer.

Programming Teams

- Up into the 1990's, even for very large games, the programmers developed the whole game (art and sounds too!)
- Shift from technical improvements to game-play
- Now for the most part, programmers write code to support designers and artists (who are the real content creators).
- Note: Mobile games are the exception.

Programming Areas

- Game code
  - Anything related directly to the game: camera, AI, displays, messaging system,
- Game engine
  - Any code that can be reused between different games (classes and libraries). Isolate game from the hardware allows focus on program logic
- Tools
  - In house tools (level editors, visual sound effects)
  - Plug-ins for off-the-shelf tools (Maya, 3ds max, Photoshop).

Team Organization

- Programmers often have a background in Computer Science or sciences
- They usually specialize in some area (AI, graphics, networking) but must know about all other areas.
- Teams usually have a lead programmer.
- They sometimes have a project lead for each of the major areas. (AI, graphics)
- “Technical”, “game-play” programmers are tool specialists.

Skills and Personalities

- Successful teams have a mix of personalities and skills:
  - Experience vs. new ideas
  - Methodical vs. visionaryqwe
- The ability for members of a team to work together is extremely important.
- Not unheard of (in business as well as games) to have “personality mapping” done.

Methodologies

- A methodology describes the procedures followed during development to create a game.
- Every company has a methodology (way of doing things), even if they don't explicitly think about it.
- With billions of dollars at stake, large companies have formally defined methodologies.

Methodologies: Code and Fix

- Unfortunately very common
- Little or no planning, reacting to events
- Poor quality, unreliability of finished product, cancellations (Duke Nukem)
- “Crunch time” and “death spiral” normal
- “Flying by the seat of pants” OK, for small projects but dangerous with years long $$.
- “Stupid Lazy”

Methodologies: Waterfall

- Very well-defined steps in development
- Lots of planning ahead of time
- Great for creating a detailed milestone schedule
- Good for “movie games” and other end to end defined projects.
- Doesn't react well to changes
- 'Exploratory' game development is too unpredictable for this approach

Methodologies: Iterative

- Multiple development cycles during a single project
  - Each delivering a new set of functionality
- The game could ship at any moment
- Allows for planning but also for changes
- Allow for effective team allocation, team planning.

Methodologies: Agile Methods

- Deal with the unexpected
- Very short iterations and goals
  - 2-3 weeks
- Iterate based on feedback of what was learned so far, open too last minute changes.
- Very good visibility of state of game
- Difficult for publishers or even developers to adopt because it's relatively new
- In use by “crash teams” at large studios.

Common Practices

- Version Control Systems (VCS)
  - Also called Revision Control (RCS) or Source Code Management (SCM)
  - Database with all the files and history.
  - Only way to work properly with a team.
  - Centralized location, “checkout”
  - History for each file helps manage bugs.
  - Branching and merging can be very useful.
  - Used for source code as well as game assets.

Common Practices

- Coding standards
  - Set of coding rules for the whole team to follow
  - Improves readability and maintainability of the code
  - Easier to work with other people's code
  - They vary a lot from place to place
    - Get used to different styles
    - Sometimes details go to 100's of pages.

Common Practices

- Automated builds
  - Dedicated build server builds the game routinely (nightly) from scratch
  - Takes the source code and creates an executable
  - Also takes assets (tools, etc.) and builds them into game-specific format
  - Build must never break or there is a problem.
  - Build may be part of a “testing”/”QA”/”debugging” process loop.

Quality

- Code reviews
  - Another programmer (or group of programmers) reads over some code to try to find problems
  - Sometimes done before code is committed to version control
  - Can be beneficial if done correctly
  - Requires a specialists eye (continuity experts in film).

Quality

- Asserts and crashes
  - Use asserts any time the game could crash or something could go very wrong
  - An assert is a controlled crash (writes to standard error)
  - Much easier to debug and fix (and turn off)
  - Happens right where the problem occurred
  - Effective in catching “program errors” not “user errors”

```
/* assert example */
#include <stdio.h>
#include <assert.h>
void print_number(int* myInt) {
    assert (myInt!=NULL);
    printf ("%d\n",*myInt); }
int main () {
  int a=10;
  int * b = NULL;
  int * c = NULL;
  b=&a;
  print_number (b);
  print_number (c);
  Return 0; }
```

Quality

- Unit tests
  - With very large code-bases, it's difficult to make changes without breaking features
  - Unit tests make sure nothing changes
  - Test very small bits of functionality in isolation (math library, graphics unit)
  - Build them and run them frequently
  - Good test harness is essential (CppUnit, CppUnitLite)

Quality

- Acceptance test (or functional tests)
  - High level tests that exercise lots of functionality (AI, level loading, objectives)
  - They usually run the whole game checking for specific features (scripts conduct the actual gameplay, PTFB)
  - Having them automated means they can run very frequently (with every build)

Quality

- Bug database
  - Keep a list of all bugs, a description, their status, priority and sometimes programmer
  - Team uses it to know what to fix next
  - Gives an idea of how far the game is from shipping
  - Doesn't prevent bugs, just helps fix them more efficiently

Leveraging Existing Code

- A lot of code that games use is the same
- It's a total waste of time to write it over and over
- Instead, spend your time in what's going to make your game unique
- Avoid Not Invented Here (NIH) syndrome!

Leveraging Existing Code

- Reuse code from previous project
  - Easier in a large company if you have an engine and tools group
- Use freeware code and tools
  - No support
  - Make sure license allows it

Leveraging Existing Code

- Middleware
  - Companies provide with components used in game development
    - physics, animation, graphics, etc
- Commercial game engines
  - You can license the whole engine and tools and a single package
  - Good if you're doing exactly that type of game

Platforms

- PCs
  - Includes Windows, Linux, and Macs
  - Can have very powerful hardware
  - Easier to patch and allow for user content
  - Need to support a wide range of hardware and drivers
  - Games need to play nice with other programs and the operating system

Platforms

- Game consoles
  - Current generation
    - Wii, Xbox 360, PS3
  - Fixed set of hardware – never changes
  - Usually use custom APIs
  - More limited resources (but well defined)
  - Currently much better sales than PC games (although that changes over time)

Platforms

- Handhelds and mobiles
  - Limited hardware (although rapidly improving)
  - Programming often done in lower-level languages (C, C++ or even assembly)
  - Much smaller projects, teams, and budgets
  - Emerging market
  - Separate lecture

Platforms

- Browser and downloadable games
  - Small games – mostly 2D
  - Need to be downloaded quickly (broadband speed question)
  - Run on the PC itself (on any browser usually)

Platforms

- Multi-platform development
  - The closer the platforms, the easier the development
  - Use abstraction layers to hide platform-specific code
  - Choice
    - Target the minimum common denominator for platforms (easy, cheap), vs. do the best you can in each platform (more expensive and time consuming)