

# Optimizing the Development Pipeline

## Tools, Technology, Process

Scott Bilas

*Game Camp 2008, Oslo, Norway*

***(Cell Phones?)***



# About Me

- Engineer at Loose Cannon Studios
- 15 years experience in pro software
  - 50% hard core games (Sierra Studios, Gas Powered Games, Loose Cannon Studios).
  - 40% casual (Edmark, Oberon Games).
  - 10% non-game academic and web (CNDE, iCat).
- Coding career generally focused on architecture, engine systems, process, tools, content pipeline.
  - I've spent a lot of time working on the problems we're going into today.
  - *I love this stuff!@#*



# Games keep getting bigger!

- Whether hard core, casual, independent...
  - Bigger, richer, more complex.
  - More content, bigger budgets, bigger teams.
  - Higher technical requirements from games (multiplayer, massively connected games, XBLA).
- Places a bigger burden on the production team.
  - We have to spend more time, more money, or get more efficient, to keep up.
  - I'm going to talk about getting more efficient: let's spend that limited development time better!

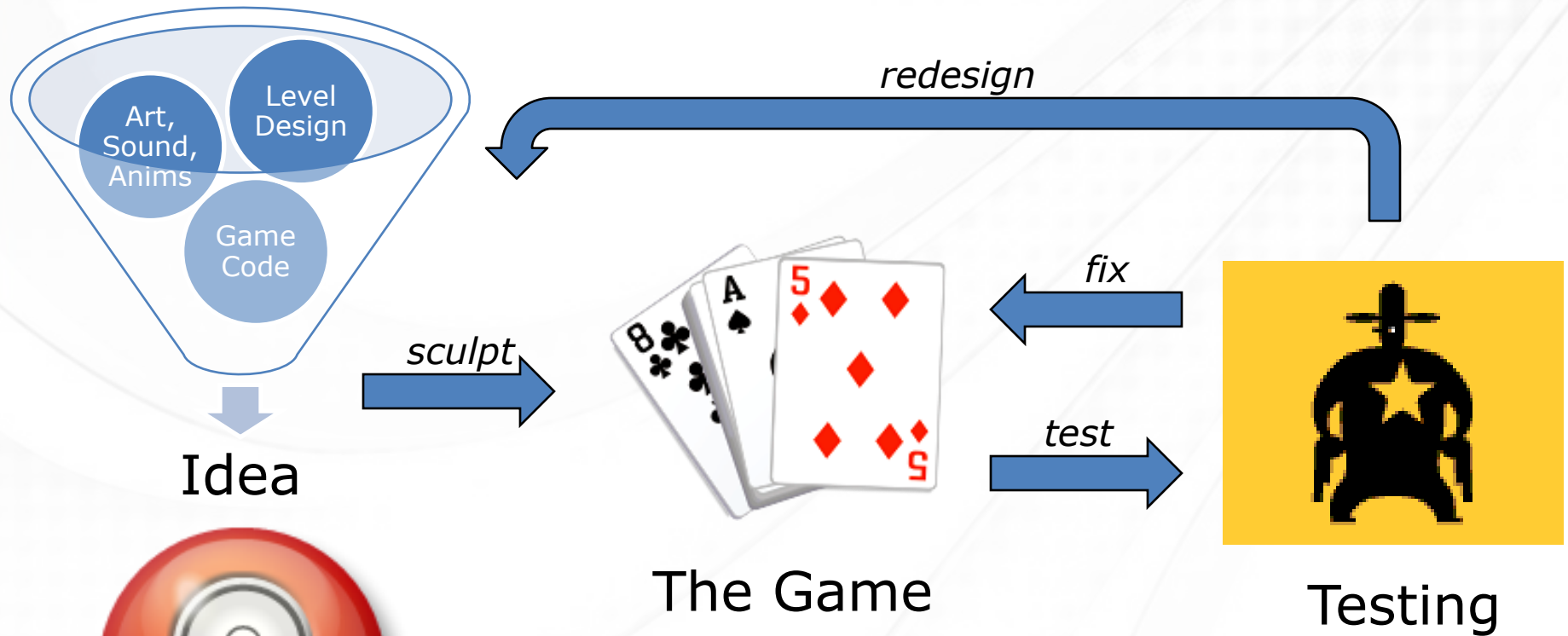
# About This Talk

- What's a development pipeline?
  - "Pipeline" is an overloaded term, but I mean the process of getting ideas into the game.
  - I'm using the term "idea" loosely - could be art, audio, game design, level design, a feature or game screen. *Something tangible.*
- "Optimizing the Development Pipeline"?
  - This talk is not about how to make something fun, or pretty, or fast.
  - It is about enabling a team to make something fun, or pretty, or fast.
- This is generally an engineering talk, hope that's ok...
  - I hope I don't assume too much game development knowledge on your part.

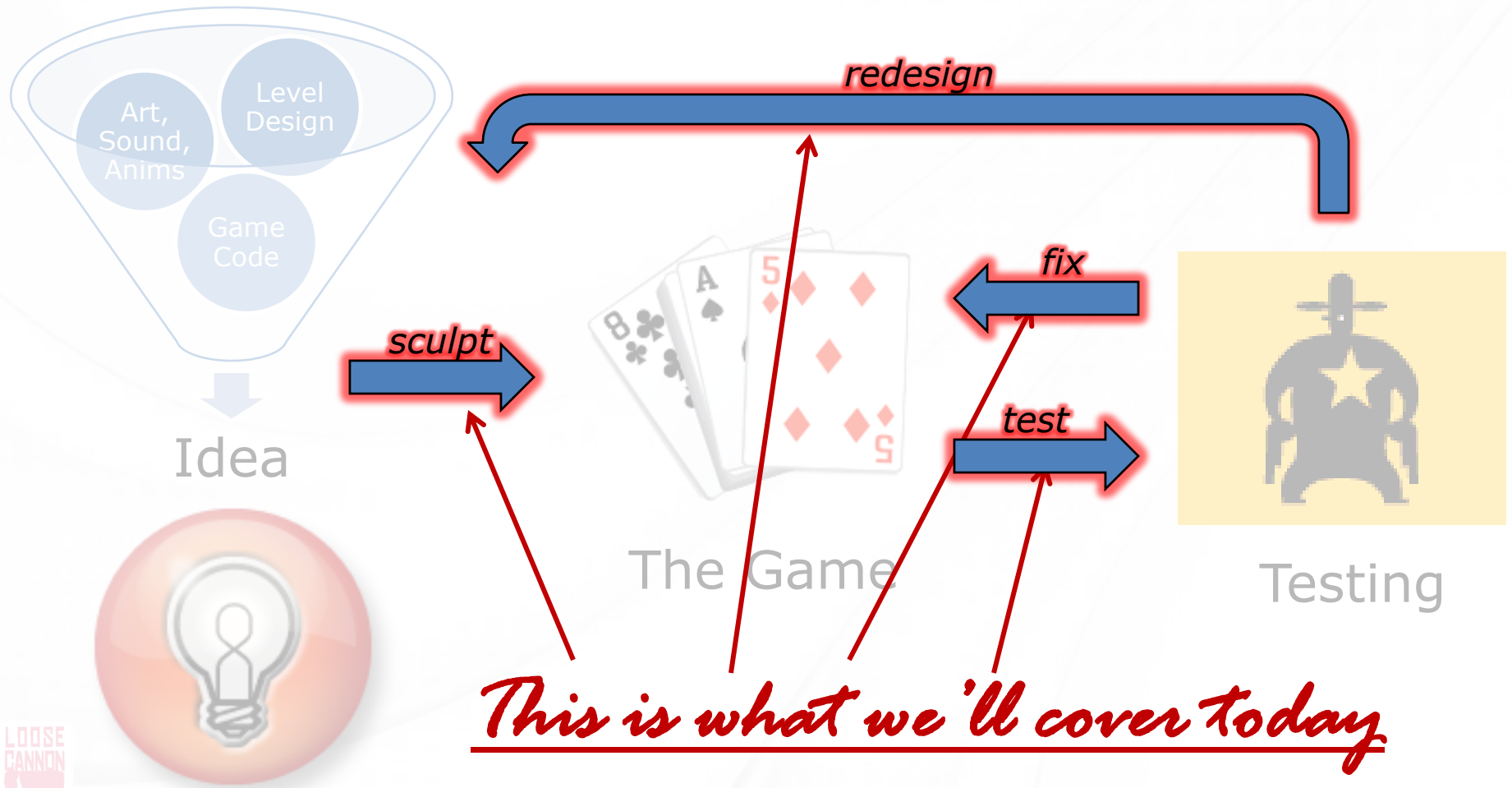
# Talk Outline

- The four stages of the development pipeline
  - *Creation, Integration, Testing, Tweaking.*
- What it means to “optimize” the pipeline
  - Making it faster, safer, more accurate.
- Specific tools, processes, and workflows to improve each area of the pipeline.
  - Identifying problems and weaknesses.
  - Proposing solutions to them.
- Some parts very technical, some parts high level.

# Simplified Pipeline



# Simplified Pipeline



# Stages of the Development Pipeline

- **Creation**
  - Building game assets. Code, anims, levels, music, etc.
- **Integration**
  - Getting the game assets into the game so other people can play with them. Typically complex and volatile.
- **Testing**
  - Testing the game assets, where testers = anyone.
- **Tweaking**
  - Making changes based on testing. Could be bug fixing, small adjustments, or even some redesign.

*Everyone participates in all or most stages – not just artists making art, level designers making levels, etc.*

# Sample Run Through the Pipeline

- Creation
  - Artist makes a new character and previews it in a tool to see what it will look like in the game.
- Integration
  - They check it into source control, possibly working with a content engineer to add it to the game.
- Testing
  - The art director reviews the work in the next build as part of a regular approval pass.
- Tweaking
  - Feedback is given to the artist about their work and they make some changes to some of the art, previewing it live in the game build.

# Key Characteristics of an Effective Pipeline

## “Fast, Safe, and Accurate”

- **Fast**

- Get assets into the game fast. Minimize the time between thinking of an idea, creating it, and seeing it work in the game.
- Engineers are really slow; keep them out of the loop as much as possible.

- **Safe**

- Set up for success by being proactive about bugs. Work defensively, anticipate problems.
- Get in the right state of mind for preventing bugs – not just in our own work but in the work of others.

# Key Characteristics of an Effective Pipeline

- Accurate
  - Knowing when something isn't right by verifying content as it loads, and telling the developer how to fix it easily with proper feedback.
  - The game shouldn't crash when something is wrong (such as spelling the name of an object wrong), yet it should still:
    - Warn with useful info about how to fix it, and...
    - ...do so loudly so the developer can't miss the warning.

# **Stage: Creation**

## **[Building game assets.]**



# Stage: Creation

- Creation requirements
  - *What rules are there for the assets I create?*
  - Formal documentation via wikis are great here
  - But typically, it's informal "tribal knowledge"
    - Explained by the leads and peers
  - Engine/tool verification routines are a great form of documentation
    - "Doctor, it hurts when I do this!"
    - Find out rules via experimentation and testing
    - (More on this in the Integration section)
- Authoring tools
  - *What tools do I use to create those assets?*

# Stage: Creation

- Physical structure
  - *Where do I put my assets once they are created so they can be integrated into the game?*
- Ideally everyone has everything they need to know to make their stuff
  - Realistically many things don't get explained until it's too late, or the rules change without notice
  - This happens at many levels of development

# Examples of Requirements

## *Stage: Creation*

- Low level
  - Object x is not thread-safe
  - XML data loads are not cached
  - Game object collection can't be modified while iterators outstanding
- Engine
  - Power of two texture for 3D game - art
  - Filename can't have spaces in it - general
  - No more than four simultaneous audio emitters - level design
- Game-specific
  - Naming conventions for files - general
  - Texture mapping requirements (all weapons on single 256x256)
  - Every character requires X shirts, Y pants, Z hats, Q emotes
  - Configuration formats - level design
- Game-general
  - Layered text for language translation
  - X megabytes total size
- Unfortunately common scenario: "hidden" rules



# Authoring Tools

## *Stage: Creation*

- These make the assets
  - Some we buy like Photoshop, Flash, Visual Studio, Torque 2D, or weird console tools
  - Some home grown like a level editor, 3D exporter, or voiceover message database
  - Includes coordination and workflow tools like source control, integration/approval worksheets, schedules
- Where do you put a file when it's been created? What should it be named? What should its format be?
  - Good use of source control is the foundation of an effective team
  - Source control is a key communication tool - what changed, when did it change, and why

# Source Control

## Stage: Creation

- Check *everything* into version control
  - Lose nothing, single point of backup
  - Guarantees consistency across all machines.
  - Guarantees you can build it all again years from now.
- Be meticulous about organizing projects
  - Helps when a year down the road your publisher wants to take your game onto a different platform, or another language, or modify it to take advantage of a new API.
- Branches are great
  - Especially when toolset and game development moving along simultaneously.
  - Branch engine and toolset as game goes beta
  - Can be difficult with Subversion

## **Stage: Integration**

**[Getting the game assets into the game  
so other people can play with them.]**



# Authoring Tools vs. Game

## *Stage: Integration*

- Source content is often different from in-game
  - Rotated scaled JPG'd in game renders different from in Photoshop
  - 3D models look very different (especially lighting)
  - Text rendered in-game almost always different
    - Depends on the rendering method and library
  - 2D layout can have subtle differences from the layout tool depending on engine settings
  - Computer monitors vs. crappy televisions
- Many special rules that the importer/exporter and in-game loader can enforce
  - “Don’t do that!” i.e. 2000-wide bitmap or unregistered font or bad naming

# Authoring Tools vs. Game

## *Stage: Integration*

- Need a way to minimize difference between appearance in tools vs. game
- Previewer app is common, great solution
  - On Dungeon Siege, export *forced* you to confirm a preview first (export-and-preview in one click).
- Ideal is the game itself as previewer, assuming minimal delay to bring up the assets in-game

Ultimate goal is to avoid having to wait for engineer to integrate assets, especially small changes

# Quick Detour: The Three Build Types

## “Debug”, “Production”, and “Release”

- **Production**

- Optimized, content safeties turned on.
- This is what the test and dev teams generally work with.
- All dev features available like the console, advanced reporting features.

- **Debug**

- Unoptimized, code and content safeties turned on, easiest to debug.
- This is what the engineers generally work with (due to reduced performance).
  - Buy very fast machines to compensate.
  - On consoles, you’re just screwed, deal with it...

# The Release Build

- This build is the only kind we send outside.
  - Goes to final distribution, beta testers, press, etc.
- Super optimized, all dev features disabled.
  - No computery sounding errors, no confusing developer features, no cheat keys.
  - Big dev stuff compiles out completely so the final exe is smaller in size.
  - Use a `#ifdef RELEASE` for this.
- How to balance what is enabled in release vs. production builds?
  - Turn off all logging? Or permit some critical logging?
  - Small amount of diagnostics can help debug issue found in the field by customer support.
    - Have them copy log files into an email, etc.

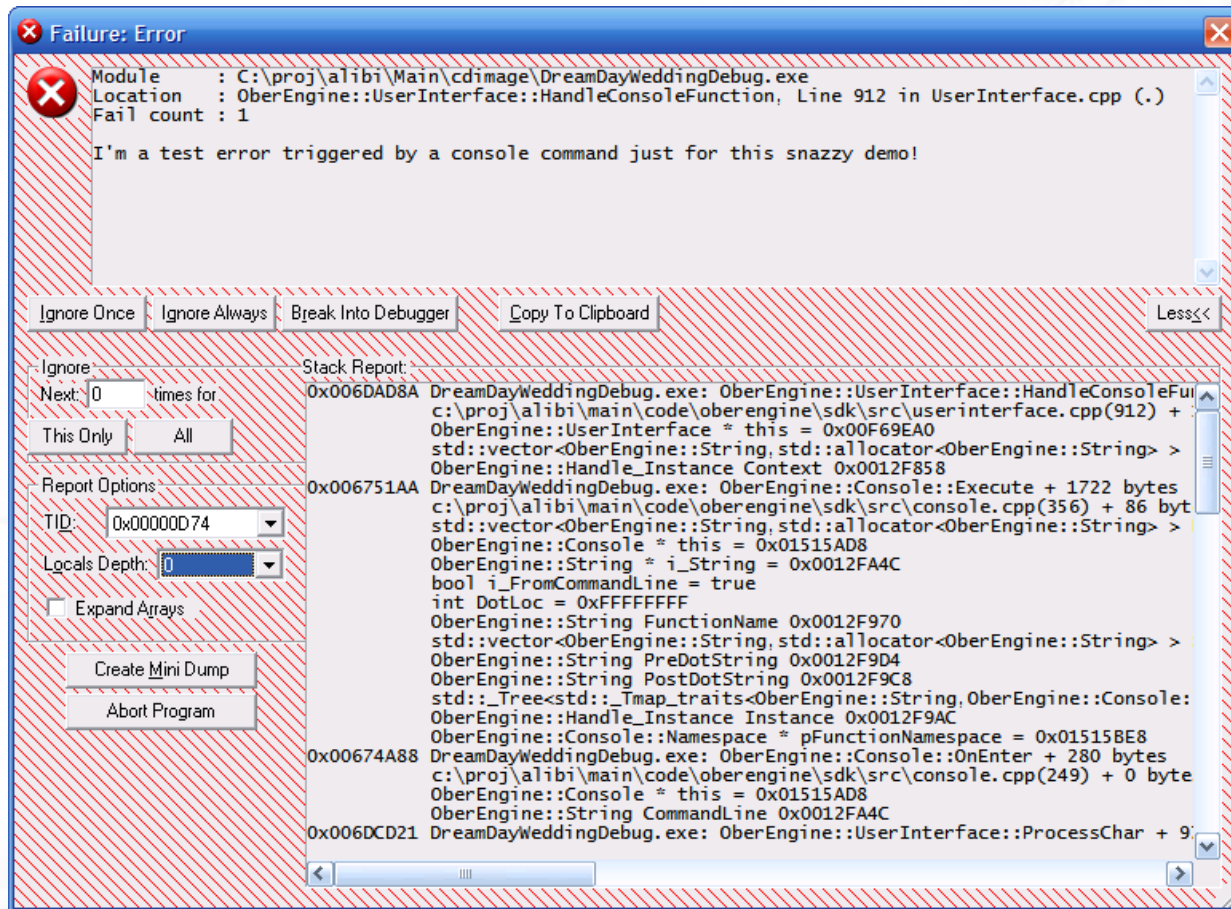
# Verify As We Integrate

- Multiple benefits
  - Content is more likely to work right in game
  - Educate developers about rules over time w/ immediate feedback
  - Keep people up to date on changing rules
  - Catch errors early
- Write validation *as you write code*
  - Cost of validating incoming data is light compared to cost of reading from disk, so do it heavily
    - No equivalent to static code analysis, no standardized tools
  - Validate everything that's obvious right away.
  - More complicated rules to validate can come later, especially in response to bugs found.

# Developer Feedback Systems

- We verified and found a problem, now what?
  - Tell the developer!!
- “Developer Feedback Systems” refers to the set of dialogs, log files, visual cues, etc. that the game uses to inform of errors and status
- Errors should include: message, location, extra stuff (stack, misc. useful game variables) as an ignorable popup
  - “Ignore always” is a useful/dangerous feature
  - Can use John Robbins’s SUPERASSERT from Bugslayer library to bootstrap

# Demo: Developer Feedback Systems



(Error dialog code based on John Robbins's "Bugslayer" library)

# Problem: Too Much or Bad Feedback

## *Developer Feedback Systems*

- How many games have log files and scrolling consoles filled with stuff almost nobody pays attention to?
  - "Initializing blah..." "Loaded xyz widgets..."
- People get numb, undermines entire system.
  - Errors that come up and don't ever get fixed make people doubt the importance of other errors
  - You want people to treat errors seriously
- How to fix?
  - Filters that are enabled by default, configured in ini file
    - Same as junk mail – make it opt-in for people who care
  - `#ifdef MY_NAME` in a locally modified "mydefs.h"
  - Tack on initialization time and memory used, e.g. "Starting up system x..." then "Done! [30.2s and 1023KB used]."

# Problem: No Attention Paid to Feedback

## *Developer Feedback Systems*

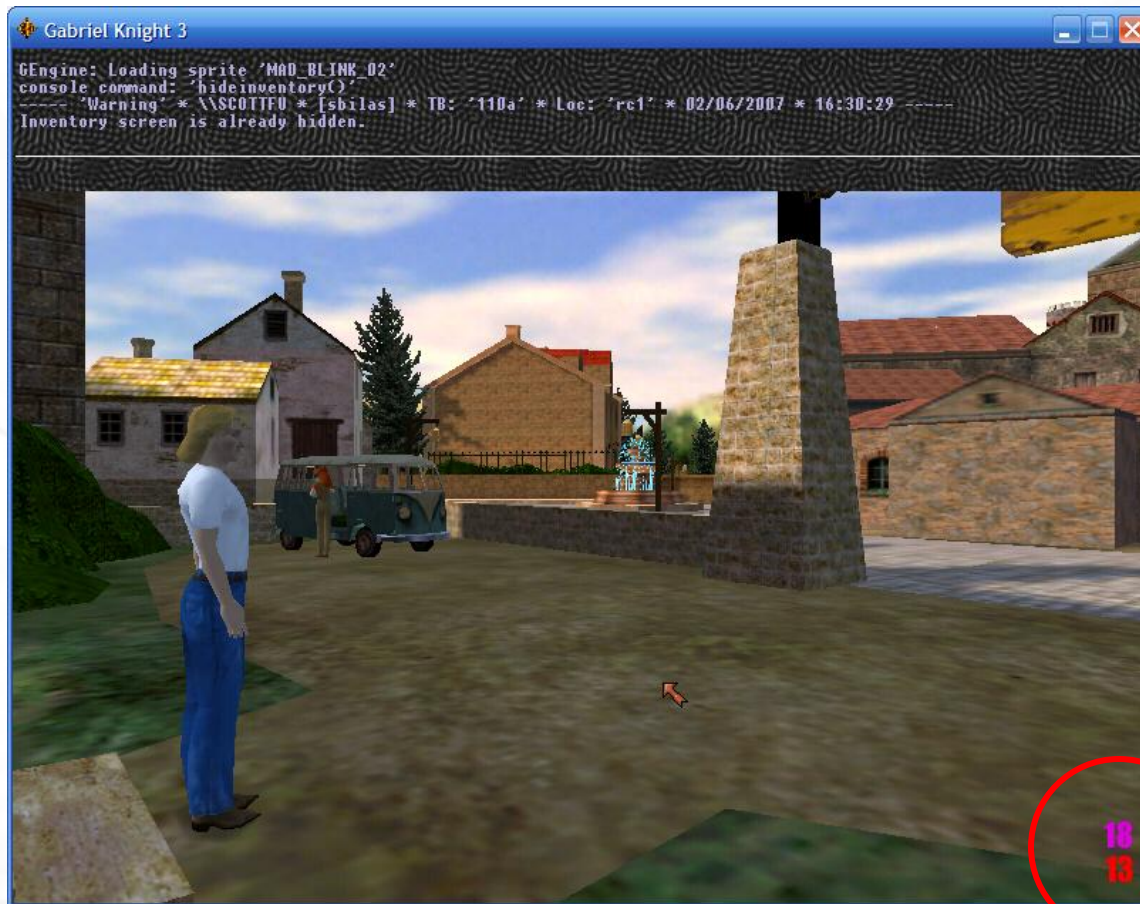
- Content creators want to get work done and into the game.
- If a dialog comes up with an incomprehensible error, they:
  - Ignore it, hit OK without even reading, get amnesia
  - Assume it's not for them and ignore it
  - If they know it's not for them, they'll assume that it's someone else's problem to report (“I'm not a tester”)
  - Assume it's supposed to work that way, as if the error is a status message
- People tend to learn patterns and go by them forever until forced to change

# Solution: Better Feedback

## *Developer Feedback Systems*

- Reducing the “too much feedback” problem helps here, so does team discipline, but we need to improve usability.
- Better error messages that actually mean something and give direction
  - Bad: “Handle is null”, Good: “Object Foo is missing texture ‘foo.png’”
  - Bad: “Invalid name”, Good: “Can’t find name ‘%s’ in database, referenced on line 123 of foo.xml – check spelling?”
- Automatic logging of all errors so we can catch ignored errors
  - This lets people work fast without losing critical diagnostic information

# Keep People Honest :)



Show number of errors and warnings ignored onscreen

# Feedback Philosophy

## *Developer Feedback Systems*

- Code by its nature must be perfect except where specifically permit sloppiness – ASSERT everything
- Content assumed trying to crash the game – verify heavily, complain loudly, continue with tolerance and grace
  - An error in art should not get in the way of audio integration
  - A bug in unrelated font rendering should not stop animation testing

# Control The Noise

## *Developer Feedback Systems*

- Strict categorization infos vs. warnings vs. errors vs. fatals
  - **Infos**: Off by default in console output, but always logged. Only for status to help diagnose in postmortems.
  - **Warnings**: Technical errors that do not cause stability but should be fixed by ship time. Log, but not with popup.
  - **Errors**: Serious errors that may cause destabilization, proceed at own risk, should be fixed as soon as possible. Do popup dialog.
  - **Fatals**: Unrecoverable, nuke entire site from orbit. Do crash dump if need info about how fatal occurred.
- “Tell Bob about this” in messages can get attention.
- “Retry” ability for script compile to enable just-in-time fix.
- Important: be gentle! Don’t get upset with apathy!!
  - Baby steps.

# Special Topic: Assertions

## *Developer Feedback Systems*

- Validate incoming parameters
  - Make sure that you're getting what you expect from whoever calls your function
- Validate object states
  - Object initialized properly? Not being modified while outer collection is iterated on? Not in the middle of a shutdown?
- Document intent
  - Asserts are better than comment blocks which get out of date
    - Live code is forced to be kept up to date
  - Asserts spell out in code exactly what conditions the parameters must satisfy
- Assertions can seriously affect debug build performance
  - Fear not, buy faster computers for engineers!
  - Also, must strike a balance (for example, `std::vector::iterator` bounds checking)
  - `#if MY_DEBUG_FEATURE` for optional deep validation

# Stage: Testing

[Testing the game assets.]



# Stage: Testing

- This is a broad topic
- Here, we focus on enabling better testing and diagnosis of problems
  - *Mostly this is about saving time*
- Outline – some catalogs
  - Defensive coding techniques (to prevent errors)
  - Diagnostic coding techniques (to enable quicker, easier diagnosis of errors)
- But first, write this down: “**Read John’s book**”
  - Debugging Applications for Microsoft .NET and Microsoft Windows by John Robbins of Wintellect (Microsoft Press)

# Naming Standards and Conventions

## *Defensive Coding*

- A big source of bugs is using API's incorrectly
- Use standards for naming, structural requirements of interfaces, etc.
  - Many options, whatever you do be consistent
  - Consistency = met expectations = documentation
  - Lack of standards leads to bugs
- Names are documentation that never get out of date because they're "live code"
  - Example: for direction caller will use, parameter prefixes "in", "out", or "io".
  - Example: for handing off ownership, parameter prefix "take", function prefix "Create" or "New", "Release"
- \$, \$\$, \$\$\$ for note, improvement, fix before ship
  - Better than TODO, FIXME, etc, includes shippability, grepping includes higher levels, guaranteed to be consistent across engineers
  - Grep before ship for \$\$\$ for stuff everybody forgot

# General Techniques

## *Defensive Coding*

- Enable memory leak reporting
  - Does not catch “runtime leaks” only total leaks, need to hook memory manager (described later) for that
- Enable corruption detection
  - Uses special pre- and post- alloc/free “bad fill”
    - Know the bad fill codes for your compiler and platform – 0xCD, 0xDD, etc.
  - When running in debugger, Windows uses special debug memory manager
- Asserts as discussed before
- Proactive content validation as discussed before.
- Get vs. Query
  - GetXyz() = “I am assuming this to be valid, give error if not”
  - QueryXyz() = “Give me back a pointer that may be null, optionally reporting an error too”
  - Minimize burden of game code error reporting or it won’t happen.
- Ansi string class for “dev” strings, Unicode string class for “screen” strings
  - Makes it very clear what strings need to go through the translated strings database and which don't.
  - 100% Unicode engine – bad idea

# General Techniques

## *Defensive Coding*

- Static assert for things the compiler can check
- Don't typedef ints to be handles, use custom types

```
#define ASSERT_STATIC( x ) \  
    typedef int $_Compile_Time_Assert_${!!(x)}  
    ASSERT_STATIC( 1 );
```

```
#define DECLARE_HANDLE(name) \  
    struct name##_ { int unused; }; \  
    typedef struct name##_ *name
```

# Diagnostic Coding

- This part is about enabling fast, accurate diagnostics
  - Have as much info as possible for rare crashes
  - Minimize sending bugs back to repro for more info
  - Avoid bad conclusions by guaranteeing *correct and relevant* info
- The Usual Suspects
  - Console
  - Scripting language, properly integrated
  - Verifiable data specification language (XML, + XSL in your dreams)
  - Proper logging, feedback system

# The Build ID

## *Diagnostic Coding*

- Create a build naming convention (“build ID”) and stick to it
- Use it everywhere
  - Names of folders for build
  - Names of zip files distributed to partners, test teams
  - Names of crash files generated
- I like projectcodename-buildtype-changelist and sometimes add -yyyy\_mm\_dd
- Put the build ID and build date in the title bar of the game when in non-release builds
  - Really helps people give you the correct build number and type
  - Screen shots will automatically include it

# Tracking Memory Manager

## *Diagnostic Coding*

- Hook the memory manager in debug builds!
- #define new/delete/malloc/free/etc. to include file/line
  - Careful of “placement new” (especially in STL)
- Expand memory allocs to include next/prev to make a ring
  - Now can do runtime memory usage analysis with onscreen overlay (usage per file, module, function... sort by size, num allocs...)
  - Include non-mem allocs in system (textures – esp. important with UMA, sound buffers, 3rd party library usage) for higher level total resource reporting
  - Include stats from OS on total physical and virtual usage to see what's missing.
  - Include general counts (such as game objects instantiated) to help give more clues when finding memory or performance problems
- Advanced: use fast 5-level stack walk, store in expanded memory block, resolve leaks with partial stacks using PDB
  - Helps when leak includes a non-empty general collection class
  - Can add a break-on-stack feature if serial ID break doesn't work

# In-Game Profiler

## *Diagnostic Coding*

- Commercial profilers only solve part of the problem
  - Lots of overhead in using them
  - Requires install of profiler to test system
  - Not set up for frame-based real-time analysis
    - Can't diagnose a hitch or dropout in frame rate
- Solution: build hierarchical in-game profiler
  - Identify major game systems, wrap entry/exit
  - Automate with macros
  - Overlay graph on screen
  - Important: include pausable scrolling historical record
    - While testing, get a hitch in frame rate, pause and look at the historical graph to figure out what happened

# Demo: In-Game Profiler

group	unit	inst	avg	min	max
Misc	%cpu	15	15	14	16
Render UI	%cpu	2	2	2	2
Update UI	%cpu	2	2	2	2
Render Effects	%cpu	2	2	2	2
Update Effects	%cpu	2	2	2	2
Render Terrain	%cpu	10	10	9	10
Light Terrain	%cpu	3	3	3	3
Render Models	%cpu	18	18	18	20
Light Models	%cpu	4	4	4	4
Deform Models	%cpu	7	7	7	8
Animate Models	%cpu	3	3	3	3
Render Misc	%cpu	27	27	27	28
AI Query	%cpu	1	1	1	1
AI Misc	%cpu	6	6	6	6
60DB	%cpu	2	2	2	2
Load Texture	%cpu	4	4	4	4
Frame Time	msec	38	38	38	39
Primary Thread	%cpu	100	100	100	100

Blue: Models, LtBlue: Terrain, Teal: UI  
 Red: Effects, DkGreen: Physics, LtGreen: 60DB  
 LtOrange: AI, Cyan: Loading, Yellow: Net  
 DkBlue: Skrit, DkRed: Fuel, Orange: Triggers  
 DkPurple: 3DMisc

Instantaneous CPU stats and color legend

Window-averaged CPU stats as pie chart



Historical CPU usage (one column per frame)



# Recording Feedback

## *Diagnostic Coding*

- Log everything to file even ignored stuff.
  - Critical to diagnosis of complicated problems.
  - Pre-init for error dialog should flush all logs.
  - Don't overwrite old logs, always append or start new file, include Build ID in filename and log session header
    - Prevents “oops I ran the game again” nuking critical logs
- Have a “submit all” btn
  - Save minidump, bmp of back buffer, copies of prefs and save games, and most recent log in Build ID named zip or folder on desktop
- Automated reporting of crashes via minidumps
- Careful of DRM interaction with minidumps

# Miscellaneous Tricks

## *Diagnostic Coding*

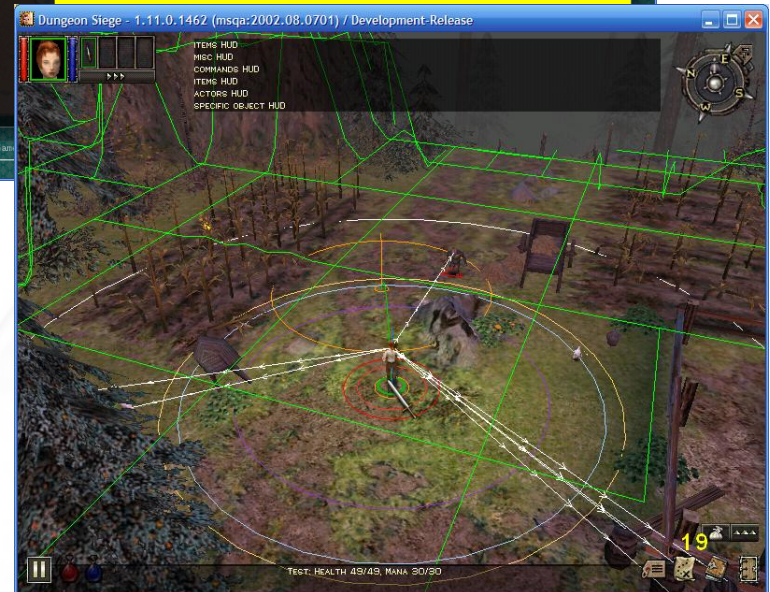
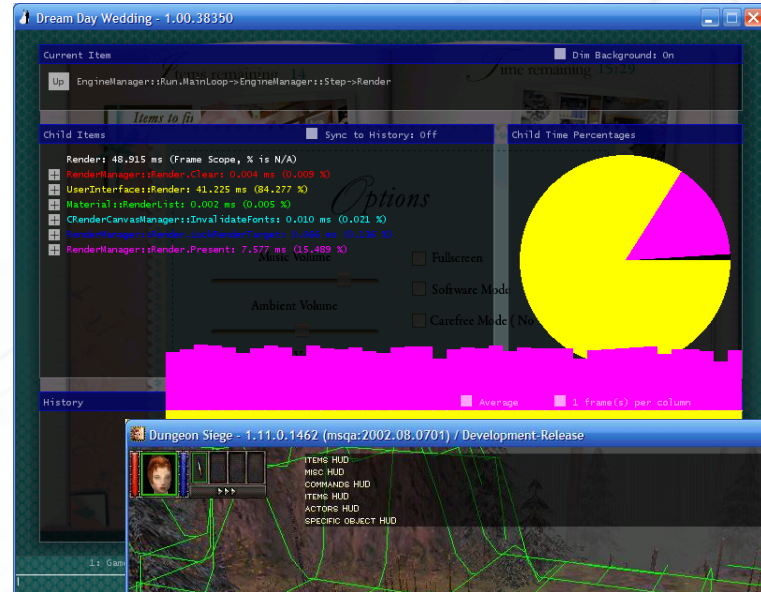
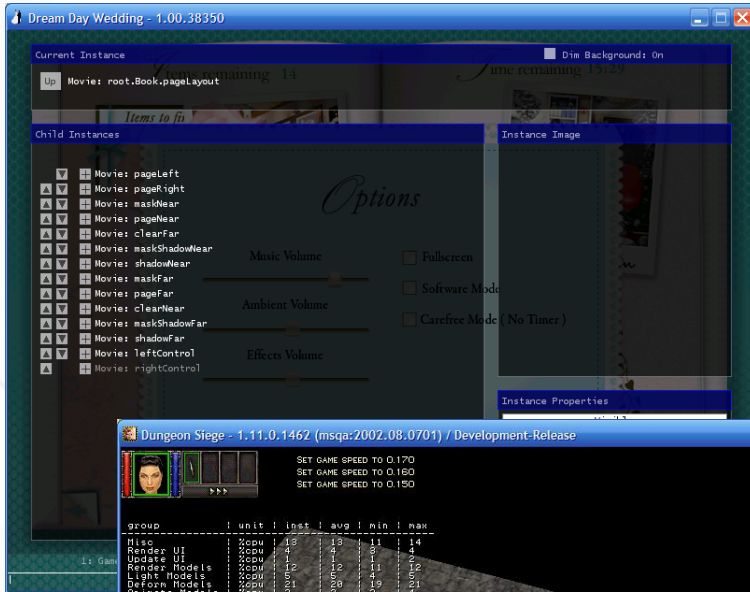
- Need a process in place to feedback issues with tools to engineers so can add new verification code
  - Must get rest of team to think “never again!”
- In-game inspection tools to walk visual object tree
- Watchdog thread for diagnosing hangs
  - Hold down special keys, thread wakes up, freezes all threads, does minidump and error report
- Ability to radically change speed of game, including go to single-step mode so that you can really focus in on when something is wrong
  - Requires several layers of "time" managed by engine
- End-user data gathering and data mining tools
  - Oberon provides this as a service to 2nd party devs



# Levels of Debugging

- Levels
  - **Light (90%)**: inspect relevant code/content from bug report
  - **Medium (9%)**: look at minidump, review logs in detail
  - **Heavy (1%)**: full reset of assumptions, painstaking removal of variables, drop into disassembly, etc.
- Know which to use and when, or can waste time
  - Jumping into the disassembler can be fun and familiar but is usually too microscopic
  - For really tough problems, heavy debugging is better than giving up and calling it a “random bug”
- Get and *confirm* the relevant info from reporter first

# Demo: Debug HUDs



# **Stage: Tweaking**

**[Making changes based on testing.]**



# Getting There Fast

## *Tweaking*

**“Optimize getting to what needs tweaking.”**

All that time wasted getting into the game at the spot you want adds up very fast.

- Trying to get the game in a particular state in order to reproduce a bug
- Getting to a particular game mode or screen in order to work on a module of the game that you're developing
- Needing to fast forward through some game progress in order to pretend you've been playing for an hour.
- Especially want to eliminate repetitive repro's.

# Command Line Flags

## *Getting There Fast*

- Use an interface system to register commands
  - Command line queries scattered through code is bad
  - Require that docs be included per command
  - Have some naming standards to avoid namespace clutter
- Support reading command line options from ini file
  - Lots easier during development so you're not juggling long command lines.
  - Can use to standardize options for entire teams when developing or testing.
- Flag to run startup scripts can do more complicated setup
  - Startup script can do complicated development setups, like "start at level 3, add in four green guys and eight yellow guys, set my score to 50000, give me x y and z upgrades".
  - Another way to do this is with saved games.

## Other Tools

### *Getting There Fast*

- Console commands
- Direct script entry through console
  - Support ctrl-v from clipboard – paste test scripts in
- Development-only hotkeys
  - Can be good and bad
- Construction mode (more on this later)
- BTW: `#ifndef RELEASE` all of this

# About Save Game Files

## *Tweaking*

- Save game files can be useful for tweaking
- Prefer a text format for dev builds
  - Easy to manually review save files when debugging.
  - Easy to generate or modify save files.
    - Need to change a score or # lives for testing? Just edit the save game and reload.
- Can pack extra debug-only info into save files to help postmortems
- Optional: encrypt for release builds
  - But have release always able to read text to help testing

# About Save Game Files

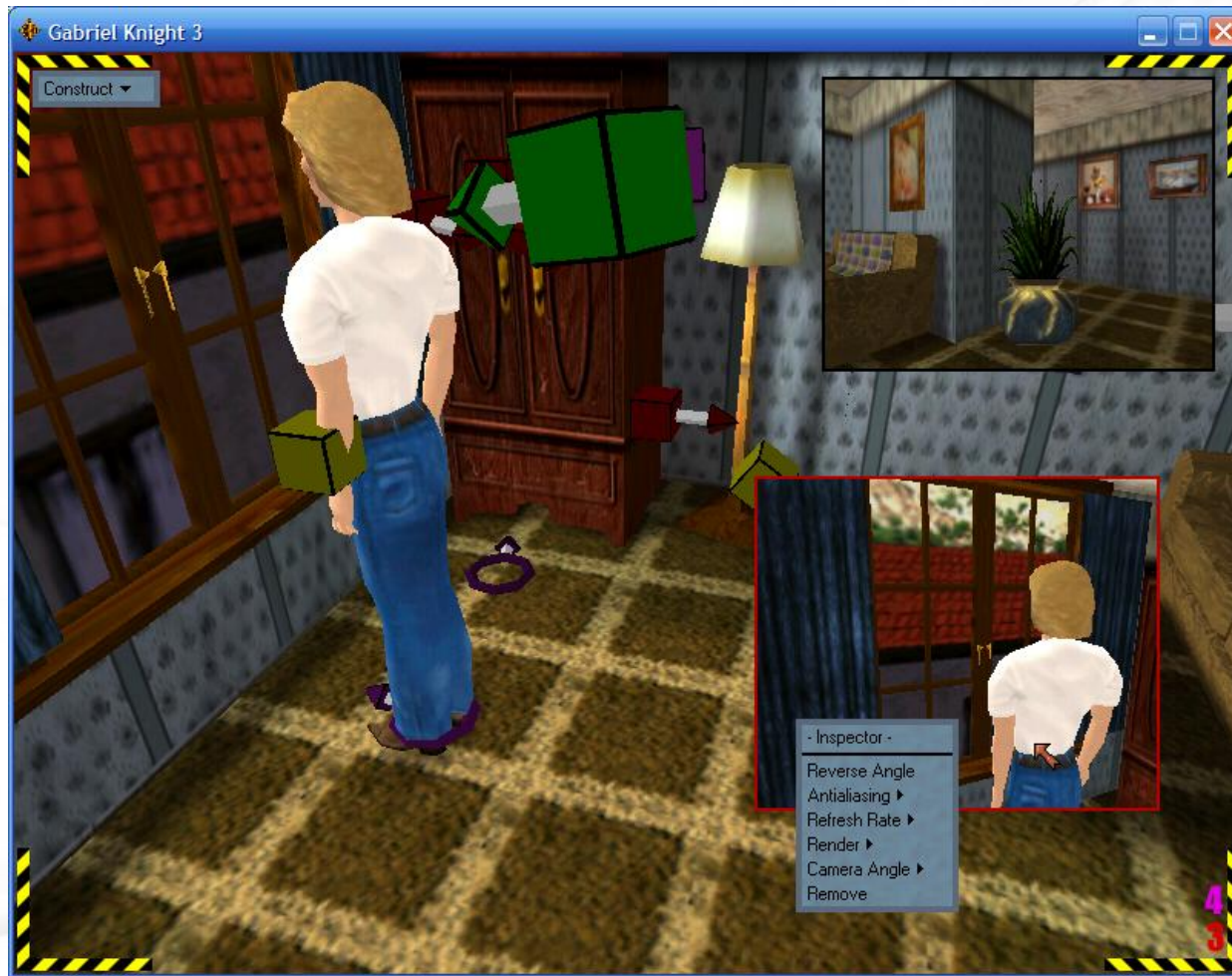
## *Tweaking*

- Be tolerant when loading save games
  - Have reasonable defaults for all values
  - Ignore things that you don't need (too much data is not an error condition), etc.
  - Permits easy forward migration
- Support a dev-only "load save game x" as a command line option or console/script command.
  - Makes directed testing lots easier, permits some test automation.
  - Great when pre-generating save files at different game stages for test team
- Store "min version number required" in save game file
  - Use when want to force save game invalidation due to breaking change

# Tweaking It Live

- Doing creation and tweaking in-game is ideal, guarantees WYSIWYG
  - Gabriel Knight 3 “construction mode”
  - Dungeon Siege game window embedded in editor, mostly shared codebase
- For externally modified files, support live reload
  - Alt-tab away, tweak, alt-tab back, "reload ui".
  - Save lots of time by avoiding exiting and restarting

# Demo: Construction Mode



# Conclusion

- We talked about a huge palette of possible features and processes for a great dev pipeline.
- Key word is “possible”
  - Every project, team, culture has its own needs.
  - Production manager and team leads should push for pipeline improvements based on need
  - Avoid over-solving – these games are still small.
  - Can take years to build up a full-featured pipeline
- The pipeline should be evolved, not dumped onto people. Allow time to adjust.

# Thank You!

- Slides will be posted to:

<http://scottbilas.com>

- Contact information:

**Scott Bilas**

[scottbilas@gmail.com](mailto:scottbilas@gmail.com)

