Lecture 17

Debugging Strategies
There are Two Main Strategies

- **Confirmation**
  - Confirm everything you believe to be true
  - Find the thing that is not actually true
  - In worse case, have to look at every line of code

- **Binary Search**
  - Identify where the code is working properly
  - Identify where the code is not working properly
  - Limit confirmation to the space in between
There are Two Main Strategies

- **Confirmation**
  - Confirm everything you believe to be true
  - Find the thing that is not actually true
  - In worse case, have to look at every line of code

- **Binary Search**
  - Identify where the code is working properly
  - Identify where the code is not working properly
  - Limit confirmation to the space in between

Everything else is a fancy tool to do this.
The Challenge of Finding Errors

- **Access errors** are the hardest
  - Refer to object in memory
  - Object is deleted somehow
  - Refer to attribute of object
  - May/may not cause crash

- Remember the 1110 rule
  - Error found != error cause
  - Cause is somewhere before

- Must work up the *call stack*
  - Part of the *binary search*
The Challenge of Finding Errors

- **Access errors** are the hardest
  - Refer to object in memory
  - Object is deleted somehow
  - Refer to attribute of object
  - May/may not cause crash

- Remember the 1110 rule
  - Error found != error cause
  - Cause is somewhere before

- Must work up the **call stack**
  - Part of the **binary search**

- “Deletion” is not immediate
  - Marks it for deletion
  - Will be deleted later

- Can still access object
  - Data corrupted as recycled
Primitive Confirmation Tools

• **Assertions** *(CCAssert)*
  - Check that your assumption is true
  - Crash the code if it is not
  - **Goal**: Make error closer to the crash

• **Logging** *(CCLog)*
  - Print out a variable value to check it
  - Alternatively print out a trace of program flow
  - **Goal**: View the internal program state
Primitive Confirmation Tools

- **CCAssert** (test, statement)
  - Test is any boolean statement
  - Statement must be char* or wchar*
  - **Ex**: `CCAssert(myptr != nullptr, "myptr is null")`

- **CCLog** (statement, v1, v2, v3...)
  - Uses same syntax as printf()
  - Need to use char* to display string names
  - **Ex**: `CCLog("Node is %s", node->getName().c_str())`
Problems with Logging

• **Verbose**
  • Code with print every animation frame
  • Way too much information to sort through
  • Most game designers will log to a file

• **Distortionary**
  • Logging and other I/O is a blocking operation
  • Will change the thread behavior of your app
  • Can cause errors to appear/disappear
Advanced Tools

• Breakpoints
  • Stop the execution of the code
  • Can continue running from that point
  • Can continue one step at a time

• Watches
  • Look at the value of an individual variable
  • Can drill down into object attributes
  • But only works when variable is in scope
Advanced Tools

- **Memory Dumps**
  - Look at a raw memory location
  - Does not require a variable to be in scope
  - Good way to look at heap for corruption

- **Thread Monitors**
  - Stack traces for all running threads
  - All threads are frozen by a breakpoint
  - Allows you to compare state across threads
XCode Tools
XCode Tools
XCode Tools

Breakpoint

Watches

Game Audio
XCode Tools

- Thread Manager
- Breakpoint
- Watches

Game Audio
XCode Tools

Memory Dump
Visual Studio Tools

Game Audio
Visual Studio Tools

Game Audio
Visual Studio Tools

Game Audio
Visual Studio Tools

Memory Dump

Breakpoint

Watches

Game Audio
Visual Studio Tools

- Memory Dump
- Breakpoint
- Call Stack
- Watches

Game Audio
Visual Studio Tools

Memory Dump

Breakpoint

Call Stack

Watches

Threads have a separate window
Breakpoint Strategies

• **Break early**
  • Break before the error, to check everything is okay
  • Step forward and watch how the code changes

• **Break infrequently**
  • If you always break, cannot initialize or animate anything
  • Design special conditionals for your breakpoint

• **Break on deletion**
  • Put breakpoints inside of all your destructors
  • Allows you to track accidental deletion
Problems with Code Stepping

- Code stepping is not “thread safe”
  - Will never leave your current thread
  - Have to choose “continue” instead of “step”

- Makes it very difficult to find thread errors
  - May miss when a variable changes state
  - We cannot find Win32 AudioEngine bug

- Solution: Rely heavily on assertions
  - Assert every variable shared across threads
  - Assert them everywhere they may change
Case Study: JSON Loading

- Problem was a thread *race condition*
  - Appeared on Windows, but not OS X
  - Because of particular Windows thread schedule
  - But technically unsafe on all platforms

- Found by putting *breakpoints in destructors*
  - Models getting deleted immediately after creation
  - Watched the reference counts to find problem
  - Autorelease pool was deleting before `retain()`
Case Study: b2BlockAllocator

- **Memory address** problem in Box2D engine
  - Problem was because we put Box2D in a DLL
  - Required stepping through the allocation process
  - Required memory dumps to view the heap

- Problem with the *static global variables*
  - DLLs have a distinct global space
  - BlockAllocator was initialized inside of the DLL
  - When it was used outside the DLL, not initialized