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
    - 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

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

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

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

• **CUAssert**(test, statement, v1, v2, v3...)  
  • Test is any boolean statement  
  • Remainder of arguments act like printf()  
  • **Ex:** CUAssert(index > 0, "index is %d", index)
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
XCode Tools

- Breakpoint
- Thread Manager
- Watches
XCode Tools

Memory Dump
Visual Studio Tools
Visual Studio Tools
Visual Studio Tools

Breakpoint

Watches
Visual Studio Tools

- **Memory Dump**
- **Breakpoint**
- **Call Stack**
- **Watches**
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 had many problems in an old AudioEngine

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

- Problem in Cocos2d-x, an older engine
  - Not a C++11 compliant engine
  - Did not support smart pointers (or anything)
  - Instead all game objects had reference counting
- Manual reference counting leads to mistakes
  - Only slightly better than manual deletion
  - Even Apple has abandoned this in Objective-C
- But very instructive for debugging memory
Aside: Reference Counting

- Every object has a **counter**
  - Tracks number of “owners”
  - No owners = memory leak
  - Increment when get reference
- Often an explicit method call
  - Historically called `retain()`
- Decrement when reference lost
  - Method call is `release()`
  - If makes count 0, delete it
Scene Graphs the Old Way

// create a new instance
Node* node = Node::create();
node->retain();

// Add the node to scene graph
scene->addChild(node);

// Release the local reference
node->release();

// Remove from scene graph
scene->removeChild(node);
Scene Graphs the Old Way

// create a new instance
Node* node = Node::create();
node->retain();

// Add the node to scene graph
scene->addChild(node);

// Release the local reference
node->release();

// Remove from scene graph
scene->removeChild(node);
	node is deleted

Custom allocator

Reference count 1

Reference count 1

Reference count 2

Reference count 1

Reference count 0
// create a new instance
Node* node = Node::create();
node->retain();

// Add the node to scene graph
scene->addChild(node);

// Do not release the local reference

// Remove from scene graph
scene->removeChild(node);

Memory Leak!
Case Study: JSON Loading

- Problem was a thread *race condition*
  - Appeared on Windows, but not MacOS
  - 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
  - There was a stray release() 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
Summary

- Two main strategies to debugging
  - **Confirmation**: Make sure code does what you think
  - **Binary Search**: Find where confirmation wrong

- Primitive tools in code on all platforms
  - Logging with CULog
  - Assertions with CUAssert

- Advanced tools in professional IDEs
  - Breakpoints and Watches
  - Thread Monitors (to see call stack)
  - Memory Dumps