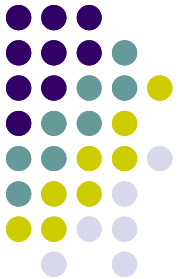
# Original Pitch



Game Performance Optimization

Ed Keenan

November 3, 2009

# In the Beginning…

There were Monkeys

# Soon they began to learn

# Use tools

# Eventually Typewriters

# Since they can type….

They could program

# Code monkey was born

# It’s not bad…

Code Monkeys get money

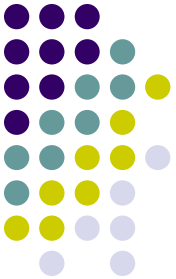
# Imagine Ninjas

- A warrior specially trained in a variety of unorthodox arts of war.

# Roles of Ninjas

- ***Sabotage***
  - Deliberate act of weakening the enemy
- ***Espionage***
  - Obtain information that is secret
- ***Scouting***
  - Exploring to gain information
- ***Assassination***
  - Targeted Killing
- ***Guerilla Warfare***
  - Uses ambush and mobility in attacking vulnerable targets

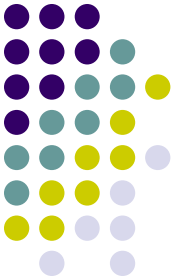
# The Enemy



- ***Resource & Performance*** grabbing issues in ***Game Development***
- Using the Ninja duties
  - Gather recon on performance issues ***Scouting***
  - Disrupt old behavior through ***Sabotage***
  - Gather information through testing ***Espionage***
  - Eliminate performance spikes ***Assassination***
  - Refactor and attack weak points ***Guerilla Warfare***

# Code Ninja is born

# Game Optimization



- Topics using actual System Game Code
  - Extended matrix instruction set
  - Dynamic memory usages
  - Increasing run-time systems to very large scale
  - C++ language enhancements and extensions
  - Streaming & File I/O
  - Profiling and metrics
- Large final project:
  - Refactor existing Particle System to improve performance and minimize resource usage

# Become a Code Ninja

GAM 391/491 Topics
Game Performance Optimization

Game performance and optimization are one of the ***MOST*** important issues that modern game console developers face

- Topics using actual System Game Code
  - Extended matrix instruction set
  - Dynamic memory usages
  - Increasing run-time systems to very large scale
  - C++ language enhancements and extensions
  - Streaming & File I/O
  - Profiling and metrics
- Large final project:
  - Refactor existing Particle System to improve performance and minimize resource usage

# Become a Code Ninja

**GAM 391/491 Topics**
**Game Performance Optimization**

Prereqs: C++, Data Structures
Linear Algebra, Graphics, Hardware Knowledge is a plus but not req'd

Ed Keenan ekeenan2@cdm.depaul.edu
Former Executive Technology Director of Midway Games

# Class Overview

Optimized C++

CSC 361 / CSC 461

Ed Keenan

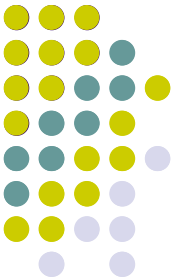
30 March 2023

13.0.9.15.7 Mayan Long Count

# Overview

- Contacts
- Philosophy
- Syllabus
- Details
- Software Development
- First Assignments

# Class Tools



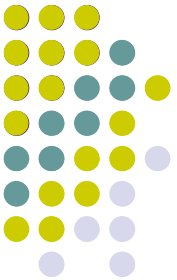
- Google Docs
  - Ideas or Concepts that come up in class
  - Capture them
  - We can go into deeper discussions in Piazza from this document.
    - Google Doc - Shared notes

# Class Tools

- Slido.com
  - Polling/Cloud/Quiz application
  - Interactions during Zoom class

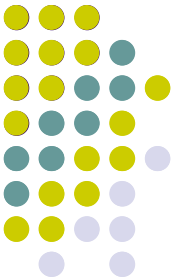
- Slido.com
  - Code: **3155898**
    - Slido also has mobile application
      - https://app.sli.do/event/79tEXMrASHpqZ7FP4M9TP9

# Course Tools



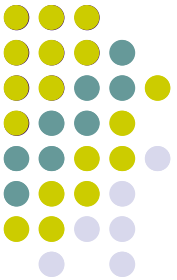
- D2L
  - Using this only for
    - Grading
    - Holding links to YouTube lectures
    - That's it
      - I hate it
      - Do not use it

# Piazza



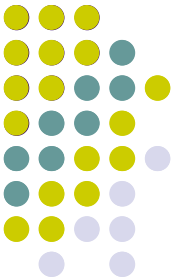
- Tool for primary communication
  - All communication
    - You are require to follow all post
    - Due dates, modifications, etc live there
  - No Emails
    - Use Piazza
  - Sensitive posts
    - Keep private piazza post

# Perforce



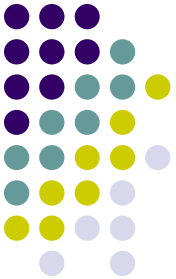
- Source Code repository
  - All the assignments
  - Lecture notes
  - Submission on Perforce
- All assignments have copyright
  - You cannot post any assignment or code on GitHub

# Visual Studio



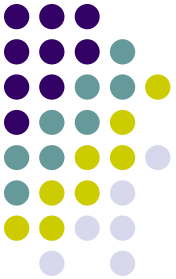
- Microsoft
  - Big Evil Empire…
    - Really good at software
    - Dogfood their product
  - Visual studio has superior IDE
    - C++, C# (Java), Python
    - You will start to see the wisdom of this choice
  - Enterprise is a $7000 tool
    - Its free to students
    - Many advance features

# Prereqs



- You are ready for C++
  - Data Structures in Java or C++
    - CSC 301, CSC 383, CSC 393, CSC 403
  - Computer Systems I
    - CSC 373,  CSC 406
  - Python
    - Doesn’t help you AT all
    - Useful for scripting, AI, data science
    - Not for C++

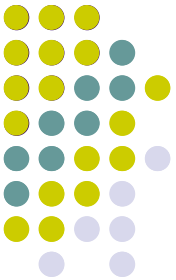
# Syllabus



→ Syllabus

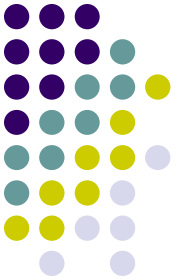
Behold its glory!

# I think…



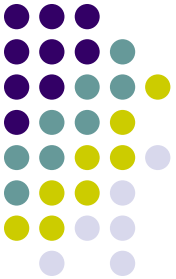
- Optimized software development is: specialized ***APPLIED*** software engineering
  - You can learn more if you
    - practice and experiment often
    - work individually
    - learn new material
    - iterate on previous work
    - compete with peers
    - take a class from Keenan

# Interactive Class



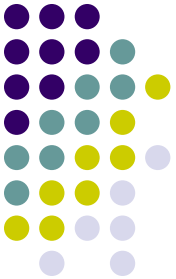
- Participation is a Big part of Class
  - Expect to be involved
  - Not a passive Class
- It's a Programming Class
  - We **program**
  - We **PROGRAM**
  - We ***PROGRAM***

# Interactive Class



- We will have:
  - Code reviews
    - Oh yes…
    - ***Yum Yum***
  - Debate
  - Demonstrations
  - Discussions
  - Challenges
- We are all learning
  - It's OK to make mistakes.
  - The only mistake is ***NOT*** trying
- Remember:
  - I am a ***professional***

# Season Programmers

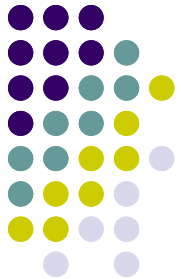


- Movie
  - Cadillac DTS

# Thank You!

Did I mention it's a programming class?

# Learn C++



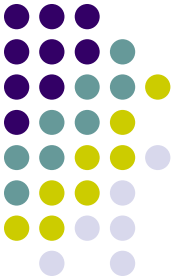
Optimized C++

Ed Keenan

# Goals

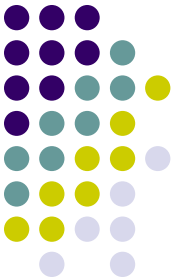
- Quick and Dirty C++ Bootcamp
  - Everyone knows Java
  - Everyone should know a little C (pointers)
- Let's get going fast

# Quick and Dirty



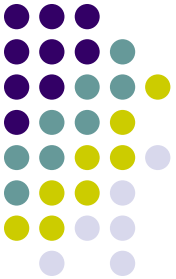
- Highlighting issues in C++
  - Look up in Reference Book
  - Start a separate thread for each item

# Big four



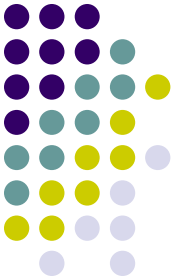
- Four functions are default created by the compiler
  - Default Constructor
    - ***Dog();***
  - Copy Constructor
    - ***Dog(const Dog &);***
  - Assignment Operator
    - ***Dog & operator = (const Dog &);***
  - Destructor
    - ***~Dog()***
- There is actually 6 with C++11 – more later in quarter

# Big four – DEFINE them



- Do not EVER use the default implementations implicitly (Pick ONE)
  - Define them yourself
  - Specify that you want to use them
- Copy Constructor – example
  - Dog(const Dog &) = default;
  - Dog(const Dog &) = delete;
  - Dog(const Dog &) { // your implementation }

# Heap vs Stack



- You can instantiate a class on the Stack or on the Heap
  - Determines who owns the memory
  - Different responsibility for the programmer

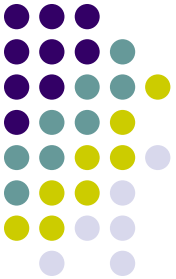
- Stack

```
void foo()
{
   Dog fido;
   fido.x = 5;
}
```

- Heap

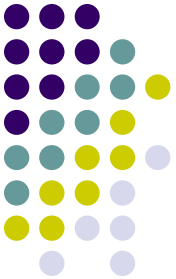
```
void foo()
{
   Dog *pDog = new Dog();
   pDog->x = 5;
}
```

# Stack is Faster than Heap



- Object is instantiated process.
  - Allocate Space in RAM
  - Initialize the Object
    - Constructor (one of the types)
- Heap
  - Call new() to allocate space
  - Extremely **SLOW** (100-1000s cycles)
  - Initialize object
- Stack
  - Create space (move the frame pointer)
  - Very **FAST** (5-20 cycles)
  - Initialize Object

# Java / C#



- Can only declare objects on Heap

**Java – Heap**

```
void foo()
{
   Dog pDog = new Dog();
   pDog.x = 5;
}
```

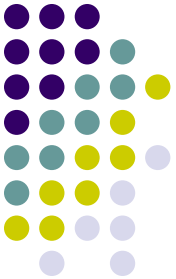
**C++ - Stack**

```
void foo()
{
   Dog fido;
   fido.x = 5;
}
```

**C++ - Heap**

```
void foo()
{
   Dog *pDog = new Dog();
   pDog->x = 5;
}
```

# Does Java use Pointers?



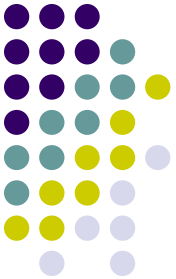
- Yes you been using pointers all along…
  - They just disguise it

**Java – Heap**

```
void foo()
{
   Dog pDog = new Dog();
   pDog.x = 5;
}
```

- pDog is Reference in Java
  - Or is it
- pDog is actually a pointer to memory on the Heap.
- Why no ->?
  - To make you feel better… it's the same code under the hood
  - pDog.x = 5;
    - is really pDog->x = 5;

# References are Pointers



- Pointer are references, References are pointers
  - That's the truth (same in Java and C#)
- References are:

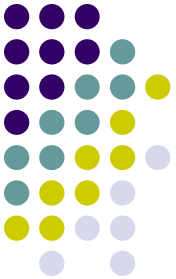
```
Dog & R
```

  is the same as

```
Dog * const P
```

- Difference
  - Difference is that R is guaranteed to be pointing to a Dog object, where pointer P may not be pointing to a Dog.
  - With pointers you can change P, but references are constant pointers that prevent the ability to change the address.
  - Syntax sugar on accessing.
    - references uses:
      - (dot) instead of `->` (arrow)

# Value, Reference, Pointer



- Prototype:

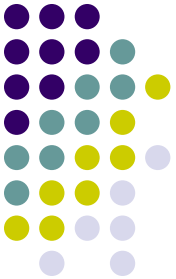
```
void foo( Dog dog )
```

- Calling:

```
Dog fido;
foo(fido);
```

- Discussion:
  - If Dog is 1000 Bytes, then 1000 Bytes are copied to foo function.
  - If fido is modified in foo, no effect in the calling function

# Value, Reference, Pointer



- Prototype:

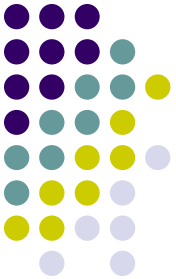
```
void foo( Dog *dog )
```

- Calling:

```
Dog fido;
foo(&fido);
```

- Discussion:
  - If Dog is 1000 Bytes, only the pointer is copied 4 bytes to the foo function.
  - If fido is modified in foo, it changes the value in the calling function

# Value, Reference, Pointer



- Prototype:

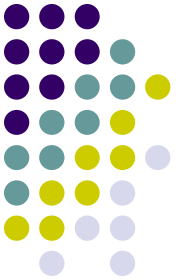
```
void foo( Dog &dog )
```

- Calling:

```
Dog fido;
foo(fido);
```

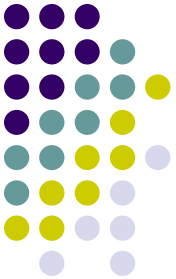
- Discussion:
  - If Dog is 1000 Bytes, only the reference(pointer) is copied 4 bytes to the foo function.
  - If fido is modified in foo, it changes the value in the calling function

# Printf



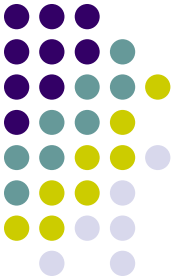
- Learn and embrace printf()
  - Its faster and easier
  - *Do not* use ***cout()***
    - That's n00b move
- Even Java now has a printf()
- Formatting contest
  - Try a complex formatting task
    - printf() and cout()
  - No contest!

# Headers – Prototypes?



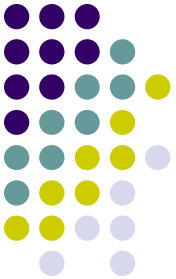
- C++ separates prototypes from the body of the code.
  - Headers are processed in the preprocessor phase.
    - Effectively including every #include into the *.cpp file.
    - So it becomes a very large file that is processed
- Function needs its prototype defined in the header
  - Body to be defined in a separate location
  - Separation of duties
- Executable code does not carry any symbol info
  - Essentially pure machine code
    - describe the interface of code
    - separate from the code itself
  - description is in the header file

# 1970s yeah!



- Headers artifact from the 70's.
  - computers had very little memory
  - keeping the entire module in memory just wasn't possible
  - Compiler start reading the file at the top
    - proceed linearly through the source code
  - Doesn't have to consider other translation units, reads the code from top to bottom.
  - Legacy
    - Similar reason why there is little endian today

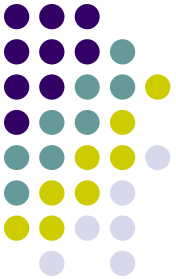
# Headers - Guards



- Since headers can be included into any file
  - Headers can also be nested
    - Headers including other headers
  - Same prototype can be included multiple times
  - Header guards prevent this from happening
- Do not rely on pragmas
  - Use Preprocessor directives:

```
#ifndef  HEADER_NAME_H
#define HEADER_NAME_H
      // header goes here
#endif
```

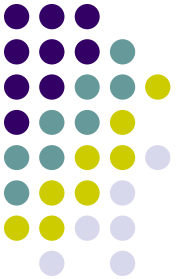
# Scope of classes



- Access specifier keywords
  - public
  - protected
  - private
- Control the access of the methods and data
  - Defined in the header / prototype definition
- Can be used on individual methods or as a list

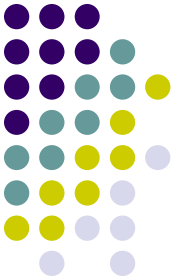
```
private void foo();
public:
        float getX();
        float getY();
```

# Java - one Type inheritance



- In Java
  - Everything is public
    - Boo… we need more flavors
- In C++
  - more flavors of inheritance
  - multiple inheritance
  - Guiding Rule…
    - *"With great power comes great responsibility"*
      - Spider Man
    - *"Let's be careful out there"*
      - Hill street blues

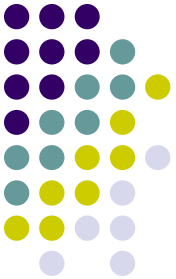
# Access of inheritance



```
class A
{
public:
    int x;
protected:
    int y;
private:
    int z;
};
```

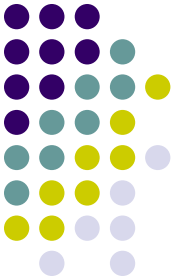
```
class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A
{
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A    // 'private' is default for classes
{
    // x is private
    // y is private
    // z is not accessible from D
};
```

# No Garbage collection



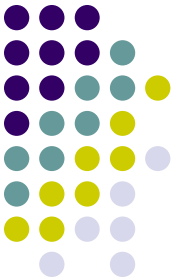
- Memory manage languages (Java, C#)
  - They can leak – it's a bear to find and fix
  - User loses control when clean-up happens
    - No hiccups for memory clean up
    - Inefficient reuse of memory
- C++ / C no memory management
  - If you call new()
    - You have to somewhere call a corresponding delete()
  - Same malloc() – free() combo

# C++ Cheat sheet (vs Java)



- Memory management – needed
  - Will leak memory if not released correctly
- Several types of inheritance
  - public, private, protected and multiple inheritance
- Objects can be declared on stack
  - Very fast allocation
- Access
  - Pointers are explicit
  - Reference are constant pointers
- Header vs Code separation

# Miscellaneous



- Material covered as we go through class
  - Const
  - Defines
  - Preprocessor
  - Initializer list
  - No protection Arrays
  - Memory Leak
  - Virtual, abstract, override, final
  - No interfaces

# Thank You!

- Easy?