CMPS 20 – Game Design Experience

Winter 2012

Staff

- Professor: Arnav Jhala ([jhala@cs.ucsc.edu](mailto:jhala@cs.ucsc.edu))
  - Office Hours: Tuesdays by appointment (can sign up online – tinyurl.com/meetarnav)
- Teaching Assistants: Sherol Chen, Trevor Sarratt
- Readers/Tutors: Jon Gill (XNA programming), Joaquin Nadal (XNA programming), Kimberly Shannon (team issues), Audrey Fabian (grading)
Resources

• Website
  – [http://courses.soe.ucsc.edu/courses/cmgs20/Winter12/01](http://courses.soe.ucsc.edu/courses/cmgs20/Winter12/01)
  – Discussion forum:
    • Piazza (piazza.com)
  – Submission and Grading:
    • eCommons (ecommons.ucsc.edu)

Overview

Design fundamentals, Game Appreciation, Game Studies, History.

Programming, Language, Data Structures, Dev Environments, Debugging.

Mostly narrow, played for fun, as consumer

Teamwork, ideation through deployment, finishing project.

Engines, advanced rendering, efficiency, new interfaces

Broad, played for fun and for study, as designer, producer, developer

Computational Cinematics Studio

[http://games.soe.ucsc.edu/ccs](http://games.soe.ucsc.edu/ccs)
Course Objectives

• Learn basic principles of game programming using a set of libraries
  – Main game loop, display of 2D sprites and 3D objects
  – Content pipeline, Art Integration
  – Collision detection, scrolling game worlds, shaders
  – Audio
• Learn basic game AI techniques
  – Simple behaviors
• Learn basic principles of object-oriented design
  – Subdividing a project into classes
  – Unified Modeling Language structure diagrams
  – Software design patterns
• Develop increased proficiency in programming
  – C# language, coding focused assignments
• Learn techniques for working as a team
  – Quarter-long game project developed in 4 person team

Grades

• Homework: 40% (3 assignments)
  – Assignment 1: XNA setup – 10%
  – Assignment 2: Debugging – 15%
  – Assignment 3: New game features – 15%
• Midterm exam: 20%
• Term project: 40%, broken down as follows
  – Team formation document: 1%
  – Game concept document: 2%
  – Technical design document: 7%
  – First playable prototype: 10%
  – Public playtesting beta version: 10%
  – Final game project: 15%
Reading Material

• Textbooks
  – Learning XNA 4.0 by Aaron Reed, O’Reilly publishers
  – Programming C# 4.0 by Jesse Liberty and Donald Xie, O’Reilly publishers, 2010
  – Available at campus bookstore and online through the library

• Reference Materials
  – Articles that are uploaded on class website
  – Links to XNA and C# development forums, tutorials, etc.

Project

• Work in teams of 4 to create a fully playable computer game
  – Developed on XNA platform in C# (covered in class)
  – XNA provides libraries and art content (meshes, textures, etc.) is freely available online
  – Created games can run on Xbox 360, PC, and Zune
  – Mentors will be assigned to each team

• Final Deliverables
  – Working game executable
  – Manual
  – Wiki page that contains:
    • Design document
    • Screenshots
    • Videos
    • Development logs (daily logs for each team member)
    • Link to subversion code repository on GForge (UCSC)
• Theme (REQUIRED)
  – Imagine Cup guidelines (imaginecup.com)
  – Stem Challenge (http://www.stemchallenge.org)
  – UCSC SOE Advising
  – UCSC Orientation
  – Documentary game
    • Darfur is dying
    • Cat and the Coup

• Phases
  – Team Formation (including Mentor Assignment) – Week 2
  – Game Concept Document – Week 4
  – Production Schedule Document – Week 5
  – Technical Design Document (including prototypes)
    – Week 7
  – Playable Prototype – Week 8
  – Public Playtest beta – Week 9
  – Final Game – Week 10
Introduction to XNA

- Programming
- Software Development
- Game Development

XNA Game Studio Express

- XNA GSE is a series of libraries for creating 2D and 3D computer games
  - Uses C# as the primary programming language
  - Integrated with Visual Studio C# Express
    - Also now the full version of Visual Studio
  - Games can run under Windows or on Xbox 360
  - It is possible to create professional games using this toolkit
- Example games:
You write your game in C#  
  – Using features in XNA Framework  
• Runs on top of common language runtime ("Managed Code")

You provide  

Provided for you

- 2D & 3D graphics support  
  – Access to HLSL (High level shader language)  
    • Pixel and vertex shaders  
• Audio support  
  – XACT cross-platform audio tool  
• Controller and keyboard input  
  – Xbox 360 controller  
• Font support  
• Content Pipeline  
• Game save storage  
• Networking  
• … and much more
• Follow instructions on pages linked from:
  – Microsoft DreamSpark ([https://www.dreamspark.com](https://www.dreamspark.com))
  – Also found on Tools page of course website

• Install Visual Studio
  – Visual Studio is an integrated development environment (editor/debugger/compiler)
  – Unless you currently use Visual Studio, you want “Visual C# 2008 Express”
    • XNA GSE will work with Visual Studio 2008 Professional if you have that installed instead

• Install XNA Game Studio 4.0
  – You want version 4.0, the latest version

---

**XNA Creator’s Club**

• XNA Creator’s Club Website
  – Community website for XNA GSE
  – Multiple complete games with source code
  – Many tutorials, and code examples
  – Very active discussion forums

• Creator’s Club Subscriptions
  – Can put game on Xbox 360
  – Access to premium content

• Trial membership - free
  – Available through Dream Spark or MSDNAA
    • Allows you to put game on Xbox 360
XNA Community Web Sites

• XNA Team Blog
  – blogs.msdn.com/xna/
  – Announcements from the XNA dev. Team

• Ziggyware
  – www.ziggyware.com
  – Developer-oriented XNA news
  – Recent contest for XNA tutorial articles
    • Winner: Skeel Keng-Siang Lee’s Introduction To Soft Body Physics

• XNA Development
  – www.xnadevelopment.com
  – XNA tutorials. See also the Links page for links to other quality XNA websites

Controllers

• XNA Game Studio Express allows you to use Xbox 360 controllers
  – Normal Xbox 360 controller is Bluetooth wireless, and is not recognized by the Windows Bluetooth manager
  – Hence, when developing game under Windows, won’t be able to test control scheme (bad)

• To create a game using Xbox 360 controller, need to:
  – Buy a corded Windows Xbox 360 controller (~$35 + shipping)
    • Google for “xbox 360 controller windows” for multiple online vendors
  – OR, buy an Xbox 360 wireless gaming receiver (~$20 + shipping)
    • allows wireless controller to work with Windows
  – Should buy now, so you have it ready for when you start programming

• Can also create a game that uses keyboard input
  – Would need to change control scheme to port to Xbox 360
This week’s work

- Visit Creators Club and DreamSpark websites
- Download and install
  - Visual Studio C# 2010
  - XNA Game Studio
- Compile and run a sample game
- Play around with starter kits
- Read Chapter 1 (Getting Started) in XNA 4.0
- Read in Programming C# 4.0
  - Chapter 1 (C# 4.0)
  - Chapter 2 (Getting Started: “Hello World”)
  - Chapter 3 (C# Language Fundamentals)
- Try one of the example code samples from the book for yourself in Visual C# 2010
  - Get familiar with Visual studio environment
- Book is available online, via O’Reilly Safari
  - http://proquest.safaribooksonline.com/9780596527433
  - Use on-campus computer to access

Upcoming Assignments

- Individual
  - XNA setup and drawing sprites
  - Alchemy game
  - http://www.onlinegameslobby.com/Games/21568/alchemy---evolution
Before we jump in...
Goals of the C# language

• A simple, modern, general-purpose object-oriented language
• Software robustness and programmer productivity
  – Strong type checking, array bounds checking, detection of use of uninitialized variables, source code portability, automatic garbage collection
• Useable in creating software components
• Ease of learning by programmers familiar with C++ and Java
• Usable for embedded and large system programming
• Strong performance, but not intended to compete with C or assembly language

Brief history of C#

• Originated by Microsoft as a response to Java
  – Initial public release in 2000
• Language name inspired by musical note
  – A “step above” C/C++ (and Java)
  – Linux wags: D♭ (D-flat, same note, different name)
• Lead designers: Anders Hejlsberg, Scott Wiltamuth
  – Hejlsberg experience: Turbo Pascal, Borland Delphi, J++
• C# standardized via ECMA and ISO
  – However, Microsoft retains architectural control
Key language features

- Unified object system
  - Everything type is an object, even primitives
- Single inheritance
- Interfaces
  - Specify methods & interfaces, but no implementation
- Structs
  - A restricted, lightweight (efficient) type
- Delegates
  - Expressive typesafe function pointer
  - Useful for strategy and observer design patterns
- Preprocessor directives

Hello World example

class Hello
{
    static void Main()
    {
        // Use the system console object
        System.Console.WriteLine("Hello, World!");
    }
}

Creates a new object type (class) called Hello.
It contains a single method, called Main.
Main contains one line, which writes "Hello, World!" on the display.
The method that performs this action is called WriteLine.
The WriteLine method belongs to the System.Console object.
The keyword "static" means that the method Main can be called even if there is no current instance of the class. It's a class method, not an instance method.
The line beginning with // is a comment, and does not execute.
Syntax

- Case-sensitive
- Whitespace has no meaning
  - Sequences of space, tab, linefeed, carriage return
- Semicolons are used to terminate statements (;)
- Curly braces {} enclose code blocks
- Comments:
  - /* comment */
  - // comment
  - /// <comment_in_xml>
    - Automatic XML commenting facility

Classes and Objects

- A class combines together
  - Data
    - Class variables
  - Behavior
    - Methods
- A key feature of object-oriented languages
  - Procedural languages, such as C, did not require clustering of data and behavior
- Class/instance distinction
  - Class defines variables & methods
  - Need to create instances of the class, called objects, to use variables & methods
  - Exception: static methods and variables
  - Analogy: a jelly bean mold (class) can be used to create a large number of jelly beans (objects, instances of the class)
Defining a class

[attributes] [access-modifiers] class identifier [:base-class [,interface(s)]]
{ class-body }

Simple example:

```csharp
class A
{
    int num = 0; // a simple variable

    A (int initial_num) { num = initial_num; } // set initial value of num
}
```

• Attributes: used to add metadata to a class
  – Can safely be ignored
• Access modifiers: one of
  – public, private, protected, internal, protected internal
• Base-class
  – Indicates (optional) parent for inheritance
• Interfaces
  – Indicates (optional) interfaces that supply method signatures that need to be implemented in the class
• Class-body
  – Code for the variables and methods of the class

Inheritance

• Operationally
  – If class B inherits from base class A, it gains all of the variables and methods of A
  – Class B can optionally add more variables and methods
  – Class B can optionally change the methods of A
• Uses
  – Reuse of class by specializing it for a specific context
  – Extending a general class for more specific uses
• Interfaces
  – Allow reuse of method definitions of interface
  – Subclass must implement method definitions
Inheritance Example

class A
{
    public void display_one()
    {
        System.Console.WriteLine("I come from A");
    }
}

class B : A
{
    public void display_two()
    {
        System.Console.WriteLine("I come from B, child of A");
    }
}

class App
{
    static void Main()
    {
        A a = new A();       // Create instance of A
        B b = new B();       // Create instance of B
        a.display_one();    // I come from A
        b.display_one();    // I come from A
        b.display_two();    // I come from B, child of A
    }
}

What today’s class will cover...

• More about classes
  – Data hiding
• Types System in C#
• Conditionals
• Properties
• Arrays
• Loops
Visibility

- A class is a container for data and behavior
- Often want to control over which code:
  - Can read & write data
  - Can call methods
- Access modifiers:
  - Public
    - No restrictions. Members visible to any method of any class
  - Private
    - Members in class A marked private only accessible to methods of class A
    - Default visibility of class variables (but is good to state this explicitly)
  - Protected
    - Members in class A marked protected accessible to methods of class A and subclasses of A.

Visibility Example

```csharp
class A
{
    public int num_slugs;
    protected int num_trees;
    ...
}

class B : A
{
    private int num_tree_sitters;
    ...
}

class C
{
    ...
}
```

- **Class A can see:**
  - num_slugs: is public
  - num_trees: is protected, but is defined in A
- **Class B can see:**
  - num_slugs: is public in A
  - num_trees: is protected in parent A
  - num_tree_sitters: is private, but is defined in B
- **Class C can see:**
  - num_slugs: is public in A
  - Can’t see:
    - num_trees: protected in A
    - num_tree_sitters: private in B
Constructors

- Use “new” to create a new object instance
  - This causes the “constructor” to be called
- A constructor is a method called when an object is created
  - C# provides a default constructor for every class
    - Creates object but takes no other action
  - Typically classes have explicitly provided constructor
- Constructor
  - Has same name as the class
  - Can take arguments
  - Usually public, though not always
    - Singleton design pattern makes constructor private to ensure only one object instance is created

Type System

- Value types
  - Directly contain data
  - Cannot be null
  - Allocated on the stack
- Reference types
  - Contain references to objects
  - May be null
  - Allocated on the heap

```csharp
int i = 123;
string s = "Hello world";
```
Predefined Types

- C# predefined types
  - Reference object, string
  - Signed sbyte, short, int, long
  - Unsigned byte, ushort, uint, ulong
  - Character char (2 byte, Unicode)
  - Floating-point float, double, decimal
  - Logical bool

- Predefined types are simply aliases for system-provided types
  - For example, int == System.Int32

Unusual types in C#

- Bool
  - Holds a boolean value, “true” or “false”
  - Integer values do not equal to boolean values
    - 0 does not equal false
    - There is no built-in conversion from integer to boolean
Unified type system

• All types ultimately inherit from object
  – Classes, enums, arrays, delegates, structs, ...
• An implicit conversion exists from any type to type object

Unified Type System (Boxing)

• Boxing
  – Process of converting a value type to the type object
  – Wraps value inside a System.Object and stores it on the managed heap
    • Can think of this as allocating a “box”, then copying the value into it
• Unboxing
  – Extracts the value type from the object
  – Checks type of box, copies value out

```csharp
int i = 123;
object o = (object)i;
int j = (int)o;
```
Variables

Variables must be initialized or assigned to before first use

Class members take a visibility operator beforehand (private by default)

Constants cannot be changed

Examples:

```csharp
int numberOfSlugs = 0;
string name;
float myFloat = 0.5f;
bool hotOrNot = true;
```

Also constants:

```csharp
const int freezingPoint = 32;
```

Enumerations

Base type can be any integral type (ushort, long) except for char

Defaults to int

Must cast to int to display in Writeln

Example: `(int)g.gradeA`
Conditionals

if (expression)
  statement1
[else
  statement2]

Example:

if (i < 5) {
  System.Console.WriteLine("i is smaller than 5");
} else {
  System.Console.WriteLine("i is greater than or equal to 5");
}

• C# supports C/C++/Java syntax for “if” statement
• Expression must evaluate to a bool value
  – No integer expressions here
• == means “equal to” for boolean comparison
  – if (i == 5)    // if i equals 5
  – if (i = 5)     // error, since i = 5 is not a boolean expression

Switch statement

switch (expression)
{
  case constant-expression:
    statement(s);
  jump-statement
  [default: statement(s);]
}

Example:

const int raining = 1;
const int snowing = 0;
int weather = snowing;
switch (weather) {
  case snowing:
    System.Console.WriteLine("It is snowing!");
    break;
  case raining:
    System.Console.WriteLine("I am wet!");
    break;
  default:
    System.Console.WriteLine("Weather OK");
    break;
}

• Alternative to if
• Typically use break
• Can use goto to continue to another case
• Classes support information hiding
  – Data members of a class can be declared as private
    • Hides implementation details of the class
  – Create accessor methods to access data
    • Typically get__() and set___(), which are public
      • “Getters and setters”
  – Other classes access data via get() and set() method
    • So long as the interface to get and set stay the same, the class can
      change how it represents its data
    • Information hiding permits implementations to change without
      affecting using classes
  – But, is tedious to always access data via accessors
    • x = foo.getX() is more tedious than x = foo.x;
    • Wouldn’t it be great to preserve benefits of accessors, while also
      retaining the syntax of direct access?

C# Properties

• Provide procedural access to data
  – Like accessors
• But have syntax similar to direct variable access
  – foo.X instead of foo.getX();
• Minor feature, but provides substantial improvement in readability and fluidity of
  programming

Travis thumbs his nose at private property
www.flickr.com/photos/sillygwailo/492070138/
C# Property Syntax

[access-modifiers] return-type property-name
{
    get
    {
        ... sequence of statements ending in a return (or throw)
    }
    set
    {
        ... sequence of statements
    }
}

- Get accessor returns value of same type as “return type”
- Set accessors have implicit parameter named “value”
  - Use to set internal data representation
- Properties can be public, private, protected
  - Public: any class can access, private: only that class, protected: that class and children
- By convention, property names have initial capital (“X” to access “x”)

C# Property Example

public class GameInfo
{
    private string name;

    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
}

// Test code
GameInfo g = new GameInfo();

// Call set accessor
g.Name = "Radiant Silvergun";

// Call get accessor
System.Console.Write(g.Name);
Automatic Properties

• Very often, properties are straightforward getters and setters
  – Get accessor just reads value of one variable
  – Set accessor just writes the value of one variable
• Creating properties in this situation is very mechanical
• With automatic properties, the compiler creates these straightforward get and set accessors

C# Automatic Property Example

```csharp
public class GameInfo
{
    public string Name {get; set;}
}

// Test code
GameInfo g = new GameInfo();

// Call set accessor
g.Name = "Radiant Silvergun";

// Call get accessor
System.Console.Write(g.Name);
```

This property behaves the same as the first property example, two slides ago.
A private class variable, private String name, is automatically created.
Arrays

• Array is an indexed collection of objects
  – Index means you use array[i] syntax to access members
  – Recall that types like int, string, float, etc. are objects
  – Can even have arrays of arrays
• Unlike C++, in C# array is an object
• Arrays have many useful properties and methods
  – Properties:
    • Length: length of array
    • Rank: number of dimensions of array
  – Methods:
    • Sort(): sorts values in one dimensional array
    • BinarySearch(): searches one dimensional array
    • Clear(): sets range of elements to 0 or null reference
    • Reverse(): reverses elements in one dimensional array
    • … and many others

Declaring an Array

```csharp
type[] array name;
Example:
int[] numbers;
numbers = new int[3];
```

- Technically, just creates a variable (numbers) that will hold a reference to an array-of-integers object.
- Creates an array-of-integers object, of length 3, which are initialized to 0.

• Array numbering follows C conventions
  – First element is numbers[0]
  – Upper bound is 2, so 3rd element is numbers[2]
Arrays of Reference Types

public class GameInfo
{
    public string gameName;
}

GameInfo[] myGArray = new GameInfo[2];

• Creating a “new” array of a reference type
  – Just creates a series of null references
  – Need to assign object instances to array elements

Arrays of Reference Types (2)

public class GameInfo
{
    public string gameName;
}

GameInfo[] myGArray = new GameInfo[2];

GameInfo A = new GameInfo();
GameInfo B = new GameInfo();

myGArray[0] = A;
myGArray[1] = B;

There are only two instances of class GameInfo.
There are four reference variables that point to GameInfo.

A, B
myGArray[0]
myGArray[1]
Initializing Arrays

- Both syntaxes have identical behavior
- Can also initialize reference types:

```java
int[] anIntArray = new int[3] { 2, 4, 6 }

OR

int[] anIntArray = { 2, 4, 6 }

string[] aStringArray = { "The", "Quick", "Brown", "Fox" }

AClass[] AClassArray = { new AClass(), new AClass(), new AClass() }
```

Two Dimensional Arrays

- Can have two (and more) dimensional arrays
- Also possible to initialize
  - Implicitly sets bounds of the array

```java
// Create a 4 x 3 array

int[,] myTwoDimArray = new int[2, 3];

myTwoDimArray = new int[,]{
    { 0, 1, 2 },
    { 3, 4, 5 },
    { 6, 7, 8 },
    { 9, 10, 11 }
};
```
Sorting Arrays

string[] aStringArray = { "Cherry", "Apple", "Banana", "Peach" };

// Sort elements
Array.Sort(aStringArray);
// Elements now: Apple, Banana, Cherry, Peach

Array.Reverse(aStringArray);
// Elements now: Peach, Cherry, Banana, Apple

• Call Array.Sort(), passing your array to sort a one dimensional array
• Call Array.Reverse() to reverse elements

Looping in C#

• C# has four looping constructs
  – for
    • for (j = 0; j < 5; ++j) { ... }
    • Classic loop syntax in C-like languages
    • Possibility of off-by-one errors in array indexing
  – foreach
    • foreach (int j in intArray)
    • Eliminates array indexing errors, no need to create index variable before statement
  – while
    • while (j < 5) { ...; j++; }
    • Loop until an event occurs
  – do ... while
    • do {...; j++;} while (j < 5)
    • Uncommon, perform action, then do condition check
Foreach Statement

foreach ( type identifier in array-or-collection )
{ ... }

• Iterates through all elements in an array, or collection type
• Creates identifier just for the scope of the statements inside the foreach
  — Holds the current element within the array/collection
• Very convenient when it can be used

```
string[] aStringArray = { "Cherry", "Apple", "Banana", "Peach" };

// Sort elements
Array.Sort( aStringArray );

foreach (string s in aStringArray)
    System.Console.Write( "{0} : ", s);

// Output: "Apple : Banana : Cherry : Peach : "
```

List

• Arrays have problem that you must know how many elements you want in advance
  — This is not always known
• List class is collection with variable size
  — Dynamically increases in size if needed
  — When an array reaches its capacity, need to create new array, and copy all elements from old array to new array
  • Ugh!
Creating a List

List<type> listname

Example:

List<string> stringList = new List<string>(); // Create list of string. Don’t forget()

stringList.Add("Quick");
stringList.Add("Brown");
stringList.Add("Fox");

foreach (string s in myStringList) // Lists work with foreach
    System.Console.Write("{0} ", s);

• Add elements with Add() method
• Clear() removes all elements from list
• Remove() removes first element from list
• Sort() sorts the list
• Count property: number of elements in list

Queue, Stack, Dictionary

• C# provides queue, stack, and dictionary
• Queue: first-in, first-out
  • Enqueue(), Dequeue(), Peek()
• Stack: last-in, first-out
  • Push(), Pop(), Peek()
• Dictionary
  – Holds set of key, value pairs
  – Permits lookup of a value given the key
  – Example use: extensible character attribute system
    • Keys: strings, names of attribute
    • Value: int, value of specific attribute
• Chapter 4 (Classes and Objects)
  Chapter 9 (Arrays, Indexers, and Collections) in
  *Programming C#*