

CMPS 20: Game Design Experience

Imagine Cup
Team Formation
XNA Input

January 14, 2010

Arnav Jhala

Adapted from Jim Whitehead's slides

Announcements

- Team Formation (Details are on the website)
 - Team member names and bio
 - Contact Information
 - Means of communication
 - Location of meetings
- Once teams are formed I will randomly assign more members in teams of <4 students (combine two 2 person teams)
- This information must be submitted, typewritten, on a piece of paper. After submission of this assignment, each team member will be expected to know this information, so you should take this opportunity to add phone numbers to cell phones, email addresses to email applications, etc.
- Details of sections are now on the webpage
- Link to Moodle class page is now on the main class homepage
- Email TA David Seagal or me for Moodle related issues

Kenny Spade: Imagine Cup

Upcoming Project Deliverable

- Game Concept Document
 - A compelling document that sells your game concept
 - Title page
 - Title of game, name of group, name of team members, sample artwork
 - Overview page
 - Table at top: game genre, platform (PC/XBox), team size
 - Key points section
 - Bulleted list of important elements of gameplay
 - Goal of game, what makes game unique, main characters, main fictional elements
 - Sample artwork image to give feel of the game
 - Biographies
 - True, pocket biographies of each team member (1-2 paragraphs each) stressing experience that makes you a strong game designer
 - 1-3 pages giving a textual description of the game
 - Fictional background, brief description of characters, goal of player in game, how does player interact with the game, brief description of levels, game audience, other important elements as needed.
 - 1-2 pages of sample conceptual artwork
 - Hand-drawn sketches are fine
- See template and evaluation criteria on course website

Game Input

- XNA Framework supports three input sources
 - Xbox 360 controller
 - Wired controller under Windows
 - Wireless or wired for Xbox 360
 - Up to 4 at a time
 - Keyboard
 - Good default for Windows games
 - Xbox 360 also supports USB keyboards
 - Mouse
 - Windows only (no Xbox 360 support)
- Poll for input
 - Every clock tick, check the state of your input devices
 - Generally works OK for 1/60th second ticks



Digital vs Analog Controls

- Input devices have two types of controls
- **Digital**
 - Reports only two states: **on** or **off**
 - Keyboard: keys
 - Controller A, B, X, Y, Back, Start, D-Pad
- **Analog**
 - Report a **range of values**
 - Xbox 360 controller: -1.0f to 1.0f
 - Mouse: mouse cursor values (in pixels)

Input Type Overview

Input Device	Digital Buttons	Analog Controls	Vibration	Win?	Xbox?	Number
Xbox 360 Controller	14	4	Yes	Yes (wired or wireless with adapter)	Yes (wireless or wired)	4
Keyboard	>100	0	No	Yes	Yes	1
Mouse	5	3	No	Yes	No	1

Xbox 360 Controller Input

- Every clock tick
 - Get state of controller
 - Call GetState() on GamePad class
 - Pass in PlayerIndex
 - PlayerIndex.One, PlayerIndex.Two, ...
 - Corresponds to lit region in “ring of light”
 - Returns a GamePadState structure
 - Check if controller is connected
 - IsConnected boolean in GamePadState
 - Check GamePadState for current state of digital and analog inputs
 - Recall that update() is called every clock tick
 - Get input in update(), or a method called from it

GamePad Class

```
public static class GamePad
{
    public static GamePadCapabilities GetCapabilities(PlayerIndex playerIndex);
    public static GamePadState GetState(PlayerIndex playerIndex);
    public static GamePadState GetState(PlayerIndex playerIndex, GamePadDeadZone
deadZoneMode);
    public static bool SetVibration(PlayerIndex playerIndex, float leftMotor, float rightMotor);
}
```

- A **static** class
 - Do not need to create an instance to use
 - All methods are static
- **GetState**
 - Retrieve current state of all inputs on one controller
- **SetVibration**
 - Use to make controller vibrate
- **GetCapabilities**
 - Determine which input types are supported.
 - Can check for voice support, and whether controller is connected.

C# Structs

- A struct in C# is a lightweight alternative to a class
- Similar to class
 - Can have constructors, properties, methods, fields, operators, nested types, indexers
- Different from class
 - Struct does not support inheritance, or destructors
 - Is a **value** type (classes are reference types)
- Rule of thumb:
 - Use structs for types that are small, simple, similar in behavior to built-in types

GamePadState Struct

```
public struct GamePadState
{
    public static bool operator !=(GamePadState left, GamePadState right);
    public static bool operator ==(GamePadState left, GamePadState right);
    public GamePadButtons Buttons { get; }
    public GamePadDPad DPad { get; }
    public bool IsConnected { get; }
    public int PacketNumber { get; }
    public GamePadThumbSticks ThumbSticks { get; }
    public GamePadTriggers Triggers { get; }
}
```

- Properties for reading state of the GamePad
 - Digital Input: Buttons, DPad
 - Analog Input: ThumbSticks, Triggers
 - Check connection state: IsConnected
 - PacketNumber
 - Number increases when gamepad state changes
 - Use to check if player has changed gamepad state since last tick

GamePadButtons Struct (Buttons)

```
GamePadState m_pad; // create GamePadState struct
m_pad = GamePad.GetState(PlayerIndex.One); // retrieve current controller state
if (m_pad.Buttons.A == ButtonState.Pressed) // do something if A button pressed
if (m_pad.Buttons.LeftStick == ButtonState.Pressed) // do something if left stick button pressed
if (m_pad.Buttons.Start == ButtonState.Pressed) // do something if start button pressed
```

- Properties for retrieving current button state
 - A, B, X, Y
 - Start, Back
 - LeftStick, RightStick
 - When you press straight down on each joystick, is a button press
 - LeftShoulder, RightShoulder
- Possible values are given by ButtonState enumeration
 - Released – button is up
 - Pressed – button is down

GameDPad Struct (DPad)

```
GamePadState m_pad; // create GamePadState struct
m_pad = GamePad.GetState(PlayerIndex.One); // retrieve current controller state
if (m_pad.DPad.Up == ButtonState.Pressed) // do something if DPad up button
pressed|
if (m_pad.DPad.Left == ButtonState.Pressed) // do something if DPad left button
pressed
```

- Properties for retrieving current DPad button state
 - Up, Down, Left, Right
- Possible values are given by ButtonState enumeration
 - Released – button is up
 - Pressed – button is down

GamePadThumbsticks Struct (Thumbsticks)

```
GamePadState m_pad; // create GamePadState struct
m_pad = GamePad.GetState(PlayerIndex.One); // retrieve current controller state
if (m_pad.Thumbsticks.Left.X > 0.0f) // do something if Left joystick pressed to
right|
if (m_pad.Thumbsticks.Right.Y < 0.0f) // do something if Right joystick pressed
down
```

- Each thumbstick has X, Y position
 - Ranges from -1.0f to 1.0f
 - Left (-1.0f), Right (1.0f), Up (1.0f), Down (-1.0f)
 - 0.0f indicates not being pressed at all
 - Represented as a Vector2
- So, have
 - Left.X, Left.Y, Right.X, Right.Y

Joystick Dead Zone

- Joysticks typically have tiny deflection to left/right or up/down
 - Leads to “drift” if uncompensated
- **Dead-zone**
 - Region around 0.0f that is interpreted as not-moving
 - Allows joysticks to have a small amount of deflection without leading to drift
- Three ways to handle this
 - From GamePadDeadZone enumeration
 - **IndependentAxes:** X & Y each have separate dead zone (default)
 - **Circular:** X & Y combined before processing dead zone
 - **None:** No processing, application must determine

GamePadTriggers Struct (Triggers)

```
GamePadState m_pad; // create GamePadState struct
m_pad = GamePad.GetState(PlayerIndex.One); // retrieve current controller state
if (m_pad.Triggers.Left != 0.0f) // do something if Left trigger pressed down
if (m_pad.Triggers.Right >= 0.95f) // do something if Right trigger pressed all the way down
```

- Each trigger ranges from 0.0f to 1.0f
 - Not pressed: 0.0f
 - Fully pressed: 1.0f
 - Represented as a float
- Have left and right triggers
 - Properties: Left, Right
- *Demonstration of XNA Input Reporter utility*
 - *creators.xna.com/en-US/utilities/inputreporter*

Controller Vibration

- Can set the vibration level of the gamepad
 - Call `SetVibration()` on `GamePad` class
 - Pass controller number, left vibration, right vibration
 - Left motor is low frequency
 - Right motor is high-frequency
- Turn vibration full on, both motors
 - `GamePad.SetVibration(PlayerIndex.One, 1.0f, 1.0f);`
- Turn vibration off, both motors
 - `GamePad.SetVibration(PlayerIndex.One, 0f, 0f);`

Keyboard Input

- Every clock tick, poll state of keyboard
 - Call `GetState()` on `Keyboard` class
 - `KeyboardState keyState = Keyboard.GetState()`
 - `Keyboard` is a static class
 - Check if a specific key is pressed
 - `if (keyState.IsKeyDown(Keys.keyname)) ...`
 - Also, `IsKeyUp(Keys.keyname)`
 - `Keys` is an enumeration of keynames
 - Provides low-level access to keypress information
 - Can determine if right/left Shift key pressed, for example
 - Also, `GetPressedKeys()`
 - Returns array of keys currently pressed
 - If length of array is zero, no keys currently pressed

Mouse Input

- Every clock tick, poll state of the mouse
 - Call GetState on Mouse class
 - `MouseState mouseState = Mouse.GetState();`
 - Mouse is a static class
- MouseState contains a series of properties
 - X, Y : position of mouse (int)
 - LeftButton, MiddleButton, RightButton, XButton1, XButton2
 - Either Released or Pressed (ButtonState enumeration)
 - ScrollWheelValue (int)
 - Scroll wheel represents cumulative change over lifetime of the game

Wrapper Class

- What if you want to use the controller, if present, and the keyboard if not?
- Create an input wrapper class
 - Checks both controller and keyboard input
 - Has a series of properties to set/get current direction state
- Example:
 - If controller connected AND controller DPad up arrow pressed
 - Set wrapper's up property to true
 - Else if keyboard Up key pressed
 - Set wrapper's up property to true
 - Else
 - Set wrapper's up property to false
- Rest of application checks input wrapper class up property

Reading

- Read Chapter 3 (User Input and Collision Detection) in *XNA 3.0*
- Download and try XNA Input Recorder demo, if you have an Xbox 360 controller
- Try creating a simple XNA program to collect keypress input

Procedural programming paradigm

- Programs are composed of a set of procedures
 - Also known as functions, methods, etc.
 - Specify a sequence of statements
- Each procedure specifies a set of steps in the overall computation
 - Few firm criteria for how to decompose a problem into procedures
- Focus is on determining the steps in the computation, and the sequence of those steps
- Some data is passed as parameters, much data is left as global variables
 - Data, and the procedures that operate on them, may be in different text files

Object-oriented programming paradigm

- Programs are composed of objects
- An object is composed of
 - Data
 - Methods that act on that data
- An object is expected to correspond to a significant abstraction in the application
- It is possible to specialize objects, via inheritance
 - E.g., square can inherit from the more generic shape
- Class/object distinction
 - Specification of a class, vs. instances of a class

Ways Procedural Designs are Expressed in C#

- Pure procedural
 - All methods are static, and found in the same class as main(). There is only one class.
 - Class variables act as global variables.
- Pure procedural with data abstraction
 - All methods are static, and found in the same class as main(). There are multiple classes, but other classes only have member variables, no methods (except a constructor).
 - Object-orientation only used to cluster related global variables together
- Procedural, with minor object-orientation
 - Design is strongly procedural, but classes have a small number of methods defined on them
- In the vast majority of cases (and until you have substantial experience as a software designer), you should avoid use of procedural design in object-oriented software.
- *Demonstration of some of these from homework submissions*

Bad Code Smells

- Once code has an object-oriented design, can then focus on improving its design
 - If the design is procedural, can't even begin to do this
- Refactoring literature has notion of “code smells”
 - “If it stinks, change it” (M. Fowler, *Refactoring*)
 - A characteristic of a design that is a strong indicator it has poor structure, and should be refactored
 - Code smells are rules of thumb
 - It's not always straightforward that a bad smell must lead to a refactoring. Have to use judgement.
 - Still, as new designers, bad code smells likely mean you should change your code.

Code Smells: Duplicated Code

- Duplicated code (code clones)
 - The same, or very similar code, appears in many places
 - Problem
 - A bug fix in one code clone may not be propagated to all
 - Makes code larger than it needs to be
 - Example from homework
 - Adjacency checks to print warnings (“I smell a Wumpus”, etc.)
 - Fix: *extract method* refactoring
 - Create new method that encapsulates duplicated code
 - Replace code clones with method call

Code smells: Long Method

- Long method
 - A method that has too many lines of code
 - How long is too long? Depends.
 - Over 20 is usually a bad sign. Under 10 lines is typically good.
 - Still, no hard and fast rules.
 - Problem
 - The longer a method, the harder it is to understand, change, and reuse
 - Example from homework
 - Shooting logic, Main
 - Fix: extract method
 - Take chunks of code from inside long method, and make a new method
 - Call new method inside the now-not-so-long method.

Code smells: Feature Envy

- Feature Envy
 - A method in one class uses primarily data and methods from another class to perform its work
 - Seems “envious” of the capabilities of the other class
 - Problem:
 - Indicates abstraction fault.
 - Ideally want data, and actions on that data, to live in the same class.
 - Feature Envy indicates the method was incorrectly placed in the wrong class
 - Fix:
 - Move method
 - Move the method with feature envy to the class containing the most frequently used methods and data items

Code smells: Large class

- Large class
 - A class is trying to do too much
 - Many instance variables
 - Many methods
- Problem:
 - Indicates abstraction fault
 - There is likely more than one concern embedded in the code
 - Or, some methods belong on other classes
 - Associated with duplicated code
- Fix:
 - *Extract class* refactoring
 - Take a subset of the instance variables and methods and create a new class with them
 - This makes the initial (long) class shorter
 - *Move method* refactoring
 - Move one or more methods to other classes
- Example from homework:
 - Class containing Main() tends to have too much game logic

Code smells: switch statements

- Switch statements
 - The cases in a switch statement contain logic for different types of instances of the same class
 - In object-oriented code, this indicates new subclasses should be created
- Problem
 - The same switch/case structure appears in many places
- Fix
 - Create new subclasses
 - Extract method to move case block logic into methods on the new subclasses

Code smells: Data class

- Data class
 - A class that has only class variables, getter/setter methods/properties, and nothing else
 - Is just acting as a data holder
- Problem
 - Typically, other classes have methods with feature envy
 - That is, there are usually other methods that primarily manipulate data in the data class
 - Indicates these methods should really be on the data class
 - Can indicate the design is really procedural
- Fix
 - Examine methods that use data in the data class, and use move method refactoring to shift methods

Homework

- Read Chapter 4 in Learning XNA 3.0 on Object-Oriented Design
- Read Chapter 1 (Refactoring, a First Example) in Martin Fowler, *Refactoring* book
 - Will post link on forum, and put link in syllabus on website
- Examine your Wumpus source code for code smells