CMPS 20: Game Design Experience

Inheritance
Polymorphism

Collision Detection

January 21, 2010
Arnav Jhala

Adapted from Jim Whitehead's slides
Classification

• Classification is the act of assigning things to categories of things.
  – The is-a relationship

• Examples:
  – A Volkswagen Jetta is a (kind of) car.
  – A hot dog is a (kind of) food.

• Classification is at the heart of object-oriented modeling.
  – An object-oriented class represents a category
    • Class car, Class food
  – Object instances (in OO) are instances of categories
Developing classes

• The process of taking a set of real world objects and developing its associated category is called **abstraction**.

• Example:
  – If I give you the set of food items:
    • Cheese whizz, hot dog, corn chips, bran flakes, chicken
  – The act by which you create the category *food* and call these all examples of food, is **abstraction**.

• The process of abstraction is used to determine what classes should be in your software
Hierarchies of Categories

• Sometimes there are situations where you have multiple levels of categories
  – Category1 is a (kind of) Category2 is a (kind of) Category3
  – Example
    • A cell phone is a (kind of) telephone
    • A wired phone is a (kind of) telephone
    • An iPhone is a (kind of) cell phone
    • Telephone is an abstract category

• In software, may want to represent things like telephone, but never have direct instances
  – Only want instances of sub-categories
  – Example: want to represent telephones, but do not want instances of telephone. Only want instances of cell phone, or wired phone.
  – In software, we would like telephone to provide details about all telephones...
    • All phones have a telephone number, can make and receive calls
  – ...but not to give cell phone or wired-phone specific information
    • Cell phones have text messaging, wired phones do not
Representing Abstract Categories

- Two ways to represent abstract categories
  - Interface
    - Class properties
    - Method names and parameter lists, but no method implementations of any methods.
  - Abstract Class
    - Class variables
    - Can have a mixture of:
      - Methods that are fully implemented
        » Might (or might not) be overridden in subclasses
      - Methods that only have a name and a parameter list
        » Abstract method
        » Must be implemented in subclasses
Interface

• Describes a set of methods, properties that must be implemented by all subclasses
  – But, no implementation details are given
  – Subclasses can implement the methods and properties as they see fit
    • So long as they match the types of the properties, method return values, and method parameters

• Describes the external boundary of class
  – What other classes need to know to use its features
  – Provides no implementation detail
Defining an interface

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

Simple example:

interface ITelephone
{
    public string PhoneNum {get; set;}   // Phone number property, can read
                                           and write

    void display_phone_num();             // Output phone number
}

• Naming convention:
  – Put capital “I” in front of interface name
Using an Interface

interface ITelephone
{
    public string PhoneNum {get; set;}  // Phone number property, can read and write
    void display_phone_num();           // Output phone number
}

public cellphone : ITelephone
{
    public string PhoneNum {get; set;}  // Use automatic property to implement interface property

    public void display_phone_num()
    {
        System.Console.WriteLine("cell phone number is \{0\}", PhoneNum);
    }
}

• Syntactically, looks like inheritance
• Must implement all aspects of interface
Abstract Classes

• Sometimes you want to:
  – Define the information passing interface for a class
    • Can do this with an interface
  – **And** provide implementations for some simple methods that all subclasses are likely to use, while only providing interfaces for some other methods which subclasses must implement
    • Cannot use an interface for this
    • Define an abstract class instead

• An abstract class looks like a regular class, except
  – Some methods are marked **abstract** and some are marked or **virtual**
Abstract vs Virtual Methods

• An abstract method
  – Only the name, return type, and parameter list is provided
  – There is no implementation – subclasses must implement
  – Must use override keyword in implementing method

• A virtual method
  – The entire method is implemented
    • Have name, return type, parameter list, and implementing statements
  – Use of virtual keyword signifies that subclasses are welcome to override
  – Subclasses may provide new implementation, but are not required to do so
    • If they do, must use override keyword
Example of Abstract and Virtual Methods

```csharp
class Telephone
{
    public string PhoneNum {get; set;} // Phone number property, can read and write
    virtual void display_phone_num() // Output phone number, might be overridden by subclasses
    {
        System.Console.WriteLine("Telephone number is {0}", PhoneNum);
    }

    abstract bool call(string num_to_call); // Abstract class, must be implemented by subclasses
}

public cellphone : Telephone
{
    public override void display_phone_num() // Overrides implementation in Telephone
    {
        System.Console.WriteLine("Cell phone number is {0}", PhoneNum);
    }

    public override bool call(string num_to_call) // Implements abstract method given in Telephone
    {
        … // implementation of calling logic
    }
}
```
Overriding methods

• Given: class B is a subclass of class A
  – class B: A { ... }

• Methods in B can override (re-implment) any method in A
  – Do not need to use override keyword for this, just do it
  – Only need to use override when method in A is marked virtual, or abstract
  – Acts as a way of forcing programmer to focus on the intention of the person developing the parent class (A)
Interface vs Abstract Class

• Interface
  – Pro
    • A class can inherit multiple interfaces
    • Interfaces can inherit (specialize) other interfaces
      – Can have rich hierarchies of interfaces
    • Can create containers with interfaces
      – Hold instances of any kind of subclass
    • Can create references to interfaces
      – Can refer to instances of any kind of subclass
  – Con
    • Cannot provide any implementation, even for simple, generic methods
Interface vs Abstract Class (cont’d)

• Abstract class
  – Pro
    • Can provide implementation of some methods, while leaving others to be implemented by subclasses
    • Allows deferring some, but not all, implementation decisions to subclass
    • Useful when there needs to be a fixed call order among methods
    • Can implement method that defines call order, but leave implementation of called methods to subclasses
    • Template Method pattern
  – Con
    • Subclass can only inherit one Abstract class
    • Abstract class must be top of inheritance hierarchy
    • Parent class may make some implementation decisions subclass cannot easily change
Interface vs Abstract class (cont’d)

- C# code tends to prefer interfaces over abstract classes

- For game code
  - Can define interfaces for specific roles of objects in game
    - ICollidable (for objects that can collide with one another)
      - Player, Enemy, Bullets can all have distinct implementations, but still inherit from ICollidable
    - Then, use List<ICollidable> to hold all collidable objects
      - One list can hold objects of type Player, Enemy, and Bullet, even though their implementations are very different
    - Collision detection then only uses methods and properties in ICollidable interface
Broad vs Narrow Sweep

• With many small objects in large playfield
  – Each object only has the potential to collide with nearby objects

• Broad sweep
  – Perform a quick pass over $n$ objects to determine which pairs have potential to intersect, $p$

• Narrow sweep
  – Perform $p \times p$ check of object pairs that have potential to intersect

• Dramatically reduces # of checks
Broad sweep approaches

• Grid
  – Divide playfield into a grid of squares
  – Place each object in its square
  – Only need to check contents of square against itself and neighboring squares
  – See http://www.harveycartel.org/metanet/tutorials/tutorialB.html for example

• Space partitioning tree
  – Subdivide space as needed into rectangles
  – Use tree data structure to point to objects in space
  – Only need to check tree leaves
  – Quadtree, Binary Space Partition (BSP) tree

• Application-specific
  – 2D-shooter: only need to check for collision against ship
  – Do quick y check for broad sweep

Point Quadtree (Wikipedia)
Space Partition Grid

- Subdivide space into a grid
  - Each cell in the grid holds a list of the objects contained in the cell
  - Objects that overlap cells are placed in multiple cells
  - One approach:

```csharp
class Grid {
    // The collision grid
    // Grid is a 2D array, where each element holds a list
    // of IGameObject
    List<IGameObject>[,] grid;
    int grid_cols;  // # of grid columns
    int grid_rows;  // # of grid rows
    int cell_x;     // x width of each grid cell
    int cell_y;     // y width of each grid cell
    ...

    This is a relatively memory intensive approach.
```
Grid Cell Size Issues

• Grid too fine
  – Cell size similar to object size
  – Many objects will overlap cells
  – Slows update and collision checks

• Grid too large
  – Few overlaps (good)
  – Larger number of n x n checks within a cell

• Grid too coarse with respect to object complexity
  – Complexity of object geometry makes pairwise comparison too hard
  – Solution: make grid size smaller, objects broken up into smaller pieces

• Grid too large and too fine
  – Can happen if object sizes vary widely
Grid Collision Test

- Within each cell
  - Compare each object in cell with every other object in cell
  - Pairwise comparison, but only within cells

```c
// Simple cell compare – does twice as many compares as necessary
void cell_collide(int row, int col) {
    foreach (IGameObject g in grid[row,col]) {
        foreach (IGameObject h in grid[row,col]) {
            if (g == h) continue;
            if (object_test(g, h) == true) {
                g.collided = true;
                h.collided = true;
            }
        }
    }
}
```
Object Update

- When an object moves, it may change cell location
- Need to compute cell(s) holding object at new location
  - May be more than one due to overlap
- If new cell, add object to cell
  - This is $O(1)$, since only adding to end of list
- If leaving existing cell, need to remove from list
  - Naïve approach: $O(\# \text{ elems in cell})$, since need to traverse lists to find object to remove
    - A problem, since fast moving objects may mean large fraction of objects need to update each tick. Approaches $O(n^2)$
  - Better approach: have each game object maintain pointer back to its location in list in each cell
    - But, requires you to write your own linked list class
Collision Detection

- Collision detection
  - Determining if two objects intersect (true or false)
  - Example: did bullet hit opponent?

- Collision resolution (collision determination)
  - Determining when two objects came into contact
    - At what point during their motion did they intersect?
    - Example: Two balls needing to bounce off each other
  - Determining where two objects intersect
    - Which points on the objects touch during a collision?
    - Example: Pong: where ball hits paddle is important

- Complexity of answering these questions increases in order given
  - If < when < where
Key to collision detection: scaling

- **Key constraint:** only 16.667ms per clock tick
  - Limited time to perform collision detection
- **Object shapes in 2D games can be quite complex**
  - Naïve: check two objects pixel-by-pixel for collisions
- **Many objects can potentially collide**
  - Naïve: check every object pair for collisions
- **Drawback of naïve approach**
  - Takes too long
  - Doesn’t scale to large numbers of objects
- **Approaches for scaling**
  - Reduce # of collision pairs
  - Reduce cost of each collision check
Naïve Collision Detection: $n \times n$ checking

- **Assume**
  - $n$ objects in game
  - All $n$ objects can potentially intersect with each other

- **Conceptually easiest approach**
  - Check each object against every other object for intersections
  - Leads to $(n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2} = O(n^2)$ checks
  - Done poorly, can be exactly $n^2$ checks
  - Example: 420 objects leads to 87,990 checks, per clock tick
Broad vs Narrow Sweep

• With many small objects in large playfield
  – Each object only has the potential to collide with nearby objects

• Broad sweep
  – Perform a quick pass over n objects to determine which pairs have potential to intersect, p

• Narrow sweep
  – Perform p x p check of object pairs that have potential to intersect

• Dramatically reduces # of checks
Broad sweep approaches

- **Grid**
  - Divide playfield into a grid of squares
  - Place each object in its square
  - Only need to check contents of square against itself and neighboring squares
  - See http://www.harveycartel.org/metanet/tutorials/tutorialB.html for example

- **Space partitioning tree**
  - Subdivide space as needed into rectangles
  - Use tree data structure to point to objects in space
  - Only need to check tree leaves
  - Quadtree, Binary Space Partition (BSP) tree

- **Application-specific**
  - 2D-shooter: only need to check for collision against ship
  - Do quick y check for broad sweep
Reducing Cost of Checking Two Objects for Collision

- General approach is to substitute a bounding volume for a more complex object
- Desirable properties of bounding volumes:
  - Inexpensive intersection test
  - Tight fitting
  - Inexpensive to compute
  - Easy to rotate and transform
  - Low memory use

Megaman X1 (Capcom). White boxes represent bounding volumes.
Common Bounding Volumes

- Circle/Sphere
- Axis-Aligned Bounding Box (AABB)
- Oriented Bounding Box (OBB)
- Convex Hull

Better bounds, better culling
Faster test, less memory

- Most introductory game programming texts call AABBs simply “bounding boxes”
Circle Bounding Box

• Simple storage, easy intersection test
• Rotationally invariant

```c
struct Point {
    int x;
    int y;
};

struct circle {
    Point c;   // center
    int r;       // radius
};

bool circle_intersect(circle a, circle b) {
    Point d;   // d = b.c – a.c
    d.x = a.c.x – b.c.x;
    d.y = a.c.y – b.c.y;

    int dist2 = d.x*d.x + d.y*d.y;  // d dot d
    int radiusSum = a.r + b.r;

    if (dist2 <= radiusSum * radiusSum) {
        return true;
    } else {
        return false;
    }
}
```

Compare Euclidean distance between circle centers against sum of circle radii.
Axis-Aligned Bounding Boxes (AABBs)

- Three common representations
  - Min-max
    
    ```
    struct AABB {
        Point min;
        Point max;
    }
    ```

  - Min-widths
    
    ```
    struct AABB {
        Point min;
        int dx;   // x width
        int dy;   // y width
    }
    ```

  - Center-radius
    
    ```
    struct AABB {
        Point c;
        int rx;   // x radius
        int ry;   // y radius
    }
    ```

Can easily be extended to 3D
Axis Aligned Bounding Box Intersection (min-max)

- Two AABBs intersect only if they overlap on both axes

```cpp
def IntersectAABB(AABB a, AABB b):
    if (a.max.x < b.min.x or a.min.x < b.max.x) return False;
    if (a.max.y < b.min.y or a.min.y < b.max.y) return False;
    return True;
```
Two AABBs intersect only if they overlap on both axes

bool IntersectAABB(AABB a, AABB b) {
    int t;
    t = a.min.x - b.min.x;
    if (t > b.dx || -t > a.dx) return false;
    t = a.min.y - b.min.y;
    if (t > b.dy || -t > a.dy) return false;
    return true;
}

// Note: requires more operations than
// min-max case (2 more subtractions, 2 more negations)
### AABB Intersection (center-radius)

**• Two AABBs intersect only if they overlap on both axes**

```cpp
bool IntersectAABB(AABB a, AABB b) {
    if (Abs(a.c.x - b.c.x) > (a.r.dx + b.r.dx)) return false;
    if (Abs(a.c.y - b.c.y) > (a.r.dy + b.r.dy)) return false;
    return true;
}
```

// Note: Abs() typically single instruction on modern processors