Lecture 8

Game Architecture Revisited
Recall: The Game Loop

60 times/s = 16.7 ms

**Update**
- Receive player input
- Process player actions
- Process NPC actions
- Interactions (e.g. physics)

**Draw**
- Cull non-visible objects
- Transform visible objects
- Draw to backing buffer
- Display backing buffer
Almost everything is in loop
except asynchronous actions
is enough for simple games

How do we organize this loop?
- Do not want spaghetti code
- Distribute over programmers
Model-View-Controller Pattern

**Controller**
- Updates model in response to events
- Updates view with model changes

**Model**
- Defines/manages the program data
- Responds to the controller requests

**View**
- Displays model to the user/player
- Provides interface for the controller

Calls the methods of
The Game Loop and MVC

- **Model**: The game state
  - Value of game resources
  - Location of game objects

- **View**: The draw phase
  - Rendering commands only
  - Major computation in update

- **Controller**: The update phase
  - Alters the game state
  - Vast majority of your code
Application Structure

Ownership

Collaboration

Architecture Revisited
Application Structure

- Collaboration
  - Must import class/interface
  - Instantiates an object OR
  - Calls the objects methods

- Ownership
  - Instantiated the object
  - Responsible for disposal
  - Superset of collaboration

Ownership

Collaboration

Root Controller

Subcontroller

Model

Subcontroller

Model
Avoid Cyclic Collaboration

Controller

Y collaborates with X

Z collaborates with

Y

X

Y collaborates with X

Collaborates with
Application Structure

Root Controller

Subcontroller → Model

Subcontroller → Model

View

Model

Model

Model

?
CUGL Views: Scene Graphs

Architecture Revisited
CUGL Views: Scene Graphs

Architecture Revisited
CUGL Views: Scene Graphs

Topic for Another Lecture

Architecture Revisited
# Model-Controller Separation (Standard)

<table>
<thead>
<tr>
<th>Model</th>
<th>Controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Store/retrieve <strong>object data</strong></td>
<td>• Process <strong>user input</strong></td>
</tr>
<tr>
<td>• Limit access (getter/setter)</td>
<td>• Determine action for input</td>
</tr>
<tr>
<td>• Preserve any invariants</td>
<td>• <strong>Example</strong>: mouse, gamepad</td>
</tr>
<tr>
<td>• Only affects this object</td>
<td>• Call action in the model</td>
</tr>
<tr>
<td>• Implements <strong>object logic</strong></td>
<td></td>
</tr>
<tr>
<td>• Complex actions on model</td>
<td></td>
</tr>
<tr>
<td>• May affect multiple models</td>
<td></td>
</tr>
<tr>
<td>• <strong>Example</strong>: attack, collide</td>
<td></td>
</tr>
</tbody>
</table>

Traditional controllers are “lightweight”
Classic Software Problem: Extensibility

• **Given**: Class with some base functionality
  • Might be provided in the language API
  • Might be provided in 3\textsuperscript{rd} party software

• **Goal**: Object with *additional* functionality
  • Classic solution is to subclass original class first
  • **Example**: Extending GUI widgets (e.g. Swing)

• But subclassing does not always work…
  • How do you extend a *Singleton* object?
Problem with Subclassing

- Games have *lots* of classes
  - Each game entity is different
  - Needs its own functionality (e.g. object methods)

- Want to avoid **redundancies**
  - Makes code hard to change
  - Common source of bugs

- Might be tempted to **subclass**
  - Common behavior in parents
  - Specific behavior in children

[Diagram showing subclassing relationships between NPC, Human, Orc, Human Warrior, Human Archer, Orc Warrior, Orc Archer, with arrows indicating redundant behavior.]
Problem with Subclassing

- Games have *lots* of classes
  - Each game entity is different
  - Needs its own functionality (e.g. object methods)
- Want to avoid *redundancies*
  - Makes code hard to change
  - Common source of bugs
- Might be tempted to *subclass*
  - Common behavior in parents
  - Specific behavior in children

Architecture Revisited
Model-Controller Separation (Standard)

Model

- Store/retrieve **object data**
  - Limit access (getter/setter)
  - Preserve any invariants
  - Only affects this object

- Implements **object logic**
  - Complex actions on model
  - May affect multiple models
  - **Example**: attack, collide

Redundant Behavior
Model-Controller Separation (Alternate)

Model

- Store/retrieve **object data**
- Limit access (getter/setter)
- Preserve any invariants
- Only affects this object

Controller

- Process **game actions**
  - Determine from input or AI
  - Find *all* objects effected
  - Apply action to objects
- Process **interactions**
  - Look at current game state
  - Look for “triggering” event
  - Apply interaction outcome

In this case, models are lightweight
Does Not Completely Solve Problem

Can I flee?

- Code **correctness** a concern
- Methods have specifications
- Must use according to spec

- Check correctness via **typing**
  - Find methods in object class
  - **Example**: orc.flee()
  - Check type of parameters
  - **Example**: force_to_flee(orc)

- **Logical** association with type
  - Even if not part of class
Issues with the OO Paradigm

- Object-oriented programming is very **noun-centric**
  - All code must be organized into classes
  - Polymorphism determines capability via type

- OO became popular with **traditional MVC pattern**
  - Widget libraries are nouns implementing view
  - Data structures (e.g. CS 2110) are all nouns
  - Controllers are not necessarily nouns, but lightweight

- Games, interactive media break this paradigm
  - View is animation (process) oriented, not widget oriented
  - Actions/capabilities only loosely connected to entities
Programming and Parts of Speech

Classes/Types are Nouns

- Methods have verb names
- Method calls are sentences
  - subject.verb(object)
  - subject.verb()
- Classes related by *is-a*
  - Indicates class a subclass of
  - **Example**: String *is-a* Object
- Objects are class *instances*

Actions are Verbs

- Capability of a game object
- Often just a simple function
  - damage(object)
  - collide(object1,object1)
- Relates to objects via *can-it*
  - **Example**: Orc *can-it* attack
  - Not necessarily tied to class
  - **Example**: swapping items
Duck Typing: Reaction to This Issue

- “Type” determined by its
  - Names of its methods
  - Names of its properties
  - If it “quacks like a duck”

- Python has this capability
  - `hasattr(<object>,<string>)`
  - True if object has attribute or method of that name

- This has many **problems**
  - Correctness is a **nightmare**

**Java:**
```
public boolean equals(Object h) {
    if (!(h instanceof Person)) {
        return false;
    }
    Person ob = (Person)h;
    return name.equals(ob.name);
}
```

**Python:**
```
def __eq__(self,ob):
    if (not hasattr(ob,'name'))
        return False
    return (self.name == ob.name)
```
Duck Typing: Reaction to This Issue

- “Type” determined by its
  - Names of its methods
  - Names of its properties
  - If it “quacks like a duck”
  - Python has this capability
    - `hasattr(<object>, <string>)`
      - True if object has attribute or method of that name
  - This has many problems
    - Correctness is a nightmare

Java:

```java
public boolean equals(Object h) {
    if (!(h instanceof Person)) {
        return false;
    }
    Person ob = (Person) h;
    return name.equals(ob.name);
}
```

Python:

```python
def __eq__(self, ob):
    if not (hasattr(ob, 'name')): return False
    return (self.name == ob.name)
```

- What do we really want?
  - Capabilities over properties
  - Extend capabilities without necessarily changing type
  - Without using new languages
- Again, use software patterns

Architecture Revisited
Possible Solution: Decorator Pattern

Request → Decorator Object → Original Object

New Functionality → Original Functionality
Java I/O Example

```java
InputStream input = System.in;

Reader reader = new InputStreamReader(input);

BufferedReader buffer = new BufferedReader(reader);
```

- Built-in console input
- Make characters easy to read
- Read whole line at a time

Most of `java.io` works this way
Alternate Solution: Delegation Pattern

Request → Original Object → Reference to delegate → Delegate Object 1

Forward Request

Inversion of the Decorator Pattern
Alternate Solution: Delegation Pattern

Inversion of the Decorator Pattern
Example: Sort Algorithms

```java
public class SortableArray extends ArrayList{

    private Sorter sorter = new MergeSorter();  // new QuickSorter();

    public void setSorter(Sorter s) { sorter = s; }

    public void sort() {
        Object[] list = toArray();
        sorter.sort(list);
        clear();
        for (o:list) { add(o); }
    }
}

public interface Sorter {
    public void sort(Object[] list);
}
```

Architecture Revisited
## Comparison of Approaches

<table>
<thead>
<tr>
<th>Decoration</th>
<th>Delegation</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Pattern applies to <em>decorator</em></td>
<td>• Applies to <em>original object</em></td>
</tr>
<tr>
<td>• Given the original object</td>
<td>• You designed object class</td>
</tr>
<tr>
<td>• Requests through decorator</td>
<td>• All requests through object</td>
</tr>
<tr>
<td>• <strong>Monolithic</strong> solution</td>
<td>• <strong>Modular</strong> solution</td>
</tr>
<tr>
<td>• Decorator has all methods</td>
<td>• Each method can have own delegate implementation</td>
</tr>
<tr>
<td>• “Layer” for more methods (e.g. Java I/O classes)</td>
<td>• Like higher-order functions</td>
</tr>
<tr>
<td>• Works on <em>any</em> object/class</td>
<td>• Limited to classes you make</td>
</tr>
</tbody>
</table>

Architecture Revisited
The Subclass Problem Revisited

Redundant Behavior

Delegates?

NPC

Human
Human Warrior
Human Archer

Orc
Orc Warrior
Orc Archer

NPC

Orc

Human

Warrior

Archer

Slot

Slot

Slot
A C++ Solution: Mix-Ins

- Orthogonal class design
  - Start with common base
  - Only one level of superclass
  - Other classes can combine

- Needs C++ templates
  - Templatize base class
  - Nest templates to mix

- Builds an inheritance tree
  - But tree details not needed
  - Stacking is commutative

```c++
class Number {
    int n;
    void set(int v) { n = v; }
    int get() const { return n; }
};

template <typename BASE>
class Undo : public BASE {
    int pre;
    void set(int v) { pre = BASE::get(); BASE::set(v); }
    void undo() { BASE::set(pre); }
};

template <typename BASE>
class Redo : public BASE {
    int post;
    void set(int v) { post = v; BASE::set(v); }
    void redo() { BASE::set(post); }
};

typedef Redo< Undo<Number> > ReUndoNumber;
```
Summary

• Games naturally fit a **specialized MVC** pattern
  • Want *lightweight* models (mainly for serialization)
  • Want *heavyweight* controllers for the game loop
  • View is specialized rendering with few widgets

• Proper design leads to unusual OO patterns
  • Subclass hierarchies are unmanageable
  • **Component-based design** better models actions
  • More advanced patterns supported by C++.