Lecture 6

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
The Game Loop

- 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
Structure of a CUGL Application

Main

Application

Scene

Models

Root Node

Scene

Models

Root Node
Structure of a CUGL Application

- Main
- Application
  - Memory policy (future lecture)
  - App Configuration

- Scene
  - Models
  - Root Node

- Scene
  - Models
  - Root Node
Structure of a CUGL Application

Main

Application

Scene

Models

Root Node

Scene

Models

Root Node
Structure of a CUGL Application

- Main
- Application
  - Scene
    - Models
    - Root Node
  - Scene
    - Models
    - Root Node
- Controller(s)
- View

Architecture Revisited
The Application Class

**onStartup()**
- Handles the game assets
  - Attaches the asset loaders
  - Loads immediate assets
- Starts any global singletons
  - **Example**: AudioChannels
- Creates any player modes
  - But does not launch *yet*
  - Waits for assets to load
  - Like **GDXRoot** in 3152

**update()**
- Called each animation frame
- Manages gameplay
  - Converts input to actions
  - Processes NPC behavior
  - Resolves physics
  - Resolves other interactions
- Updates the scene graph
  - Transforms nodes
  - Enables/disables nodes
The Application Class

**onStartup()**

- Handles the game assets
- Attaches the asset loaders
- Loads immediate assets
- Starts any entities
- Creates any player modes
- But does not launch yet
- Waits for assets to load
- Like GDXRoot in 3152

**update()**

- Called each animation frame
- Manages gameplay
- Converts input to actions
- Resolves NPC behavior
- Resolves other interactions
- Updates the scene graph
- Transforms nodes
- Enables/disables nodes

**onShutdown()**
cleans this up

Does not draw! Handled separately
Application Structure

Ownership

Collaboration

Model

Scene Controller

Subcontroller

Subcontroller

View

Model

Model

Model

Architecture Revisited
Application Structure

Ownership

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

- **Ownership**
  - Instantiated the object
  - Responsible for disposal
  - Superset of collaboration
Avoid Cyclic Collaboration

Y collaborates with X

Controller

Z collaborates

Y collaborates with X
Scene Structure

Scene Controller

Subcontroller

Subcontroller

View

Model

Model

Model

Architecture Revisited
CUGL Views: Scene Graphs

![Scene Graph Diagram]

- Scene
  - Root Node
    - Node
      - Node
      - Node
      - Node
    - Node
  - Model
  - Model
CUGL Views: Scene Graphs

Architecture Revisited
CUGL Views: Scene Graphs

Topic for Another Lecture
## 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

### Controller
- **Process** user input
  - Determine action for input
  - **Example**: mouse, gamepad
  - Call action in the model

---

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 3rd 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?
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

Problem with Subclassing

![Diagram showing subclassing in game design]

Human
- Human Warrior
- Human Archer

Orc
- Orc Warrior
- Orc Archer

NPC
- Redundant Behavior
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
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

- 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

Can I *flee*?
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:
```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)
```
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)
```

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*
Possible Solution: Decorator Pattern

New Functionality

Request

Decorator Object

Original Functionality

Original Object
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 → Delegate Object 1

Reference to delegate

Forward Request

Inversion of the Decorator Pattern
Alternate Solution: Delegation Pattern

Alternative Solution: Delegation Pattern

Request

Original Object

Reference to delegate

Delegate Object 2

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); }
    }
}
```

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

Architecture Revisited
Comparison of Approaches

## Decoration

- Pattern applies to *decorator*
  - Given the original object
  - Requests through decorator

  **Monolithic** solution
  - Decorator has all methods
  - “Layer” for more methods (e.g. Java I/O classes)

- Works on *any* object/class

## Delegation

- Applies to *original object*
  - You designed object class
  - All requests through object

  **Modular** solution
  - Each method can have own delegate implementation
  - Like higher-order functions

- Limited to classes you make
The Subclass Problem Revisited

Delegates?

NPC

Human
- Human Warrior
- Human Archer

Orc
- Orc Warrior
- Orc Archer

Redundant Behavior

NPC

Human
- Slot
- Slot

Orc
- Slot

Warrior
- Archer
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

- CUGL view is handled in scene graphs

- Proper design leads to unusual OO patterns
  - Subclass hierarchies are unmanageable
  - Component-based design better models actions