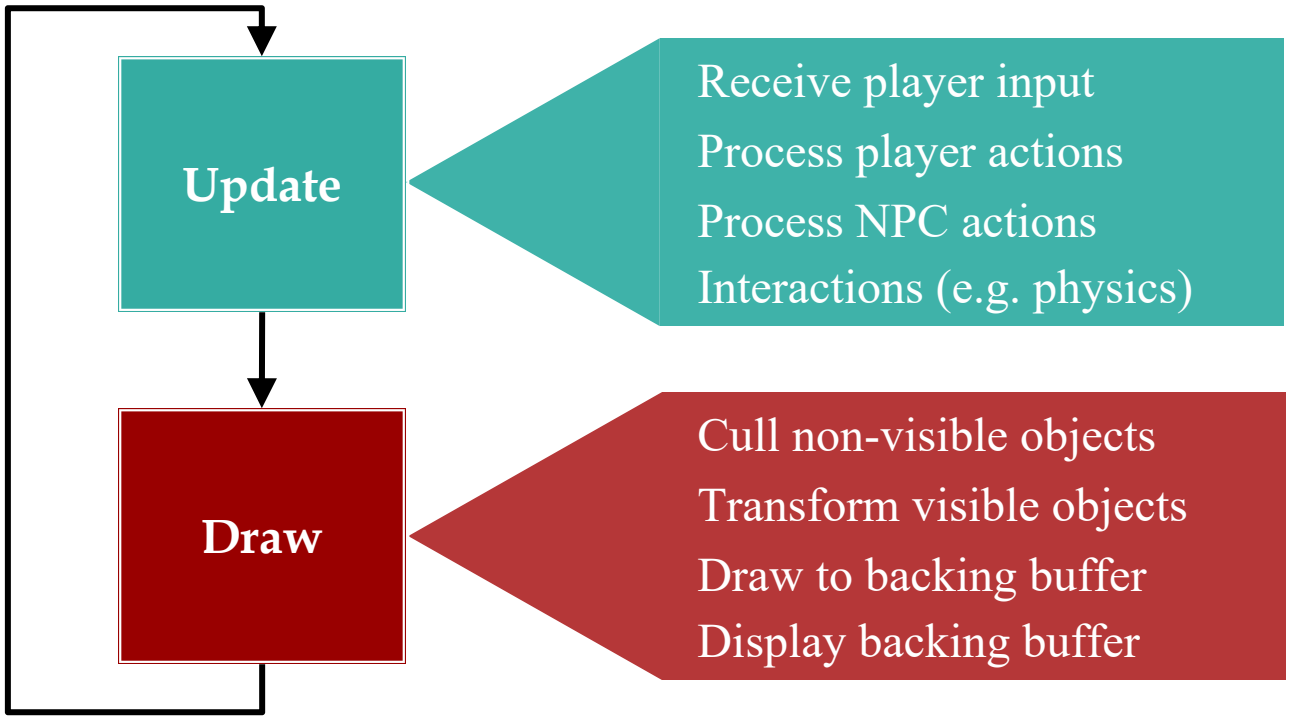
the gamedesigninitiative at cornell university

Lecture 4

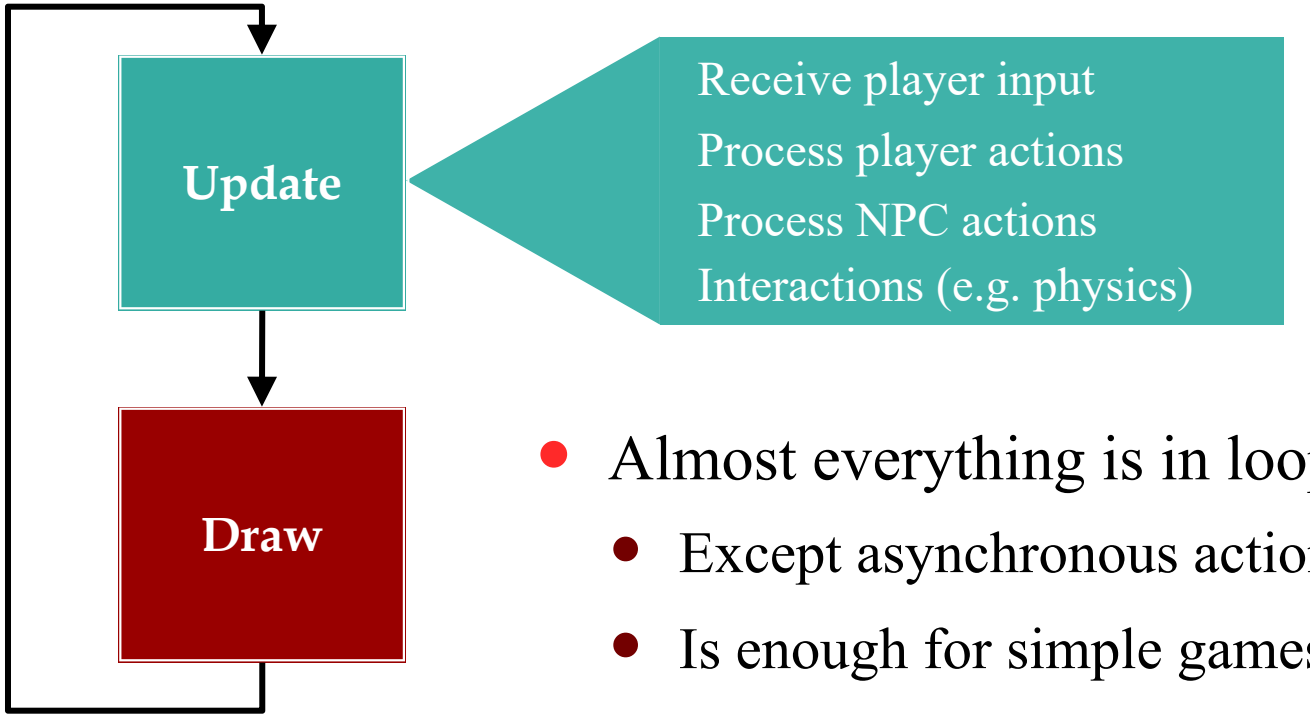
# Game Architecture Revisited

# Recall: The Game Loop

60 times/s
=
16.7 ms

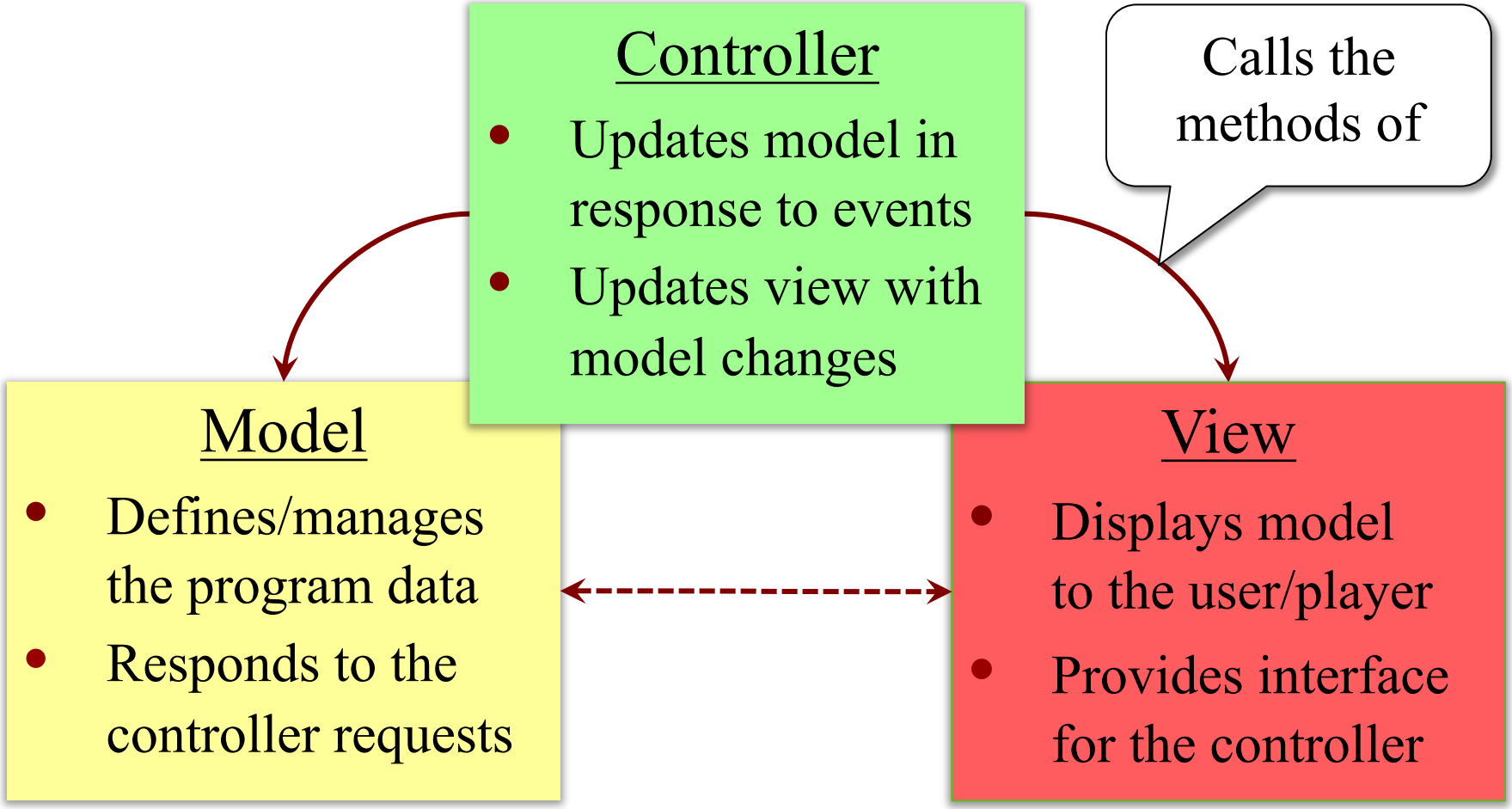


# Recall: The Game Loop



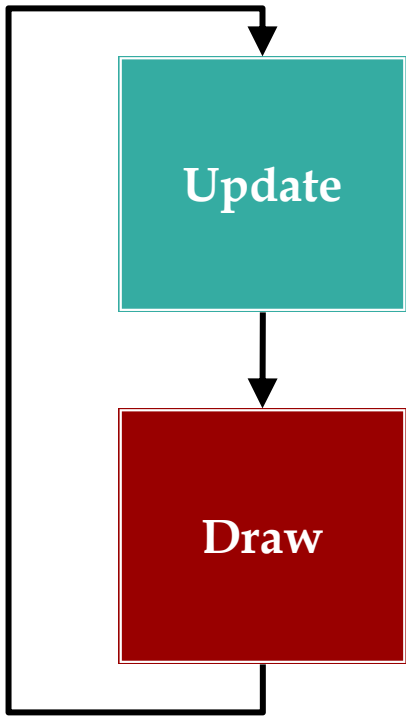

- Almost everything is in loop
  - Except asynchronous actions
  - Is enough for simple games
- How do we organize this loop?
  - Do not want spaghetti code
  - Distribute over programmers

# Model-View-Controller Pattern

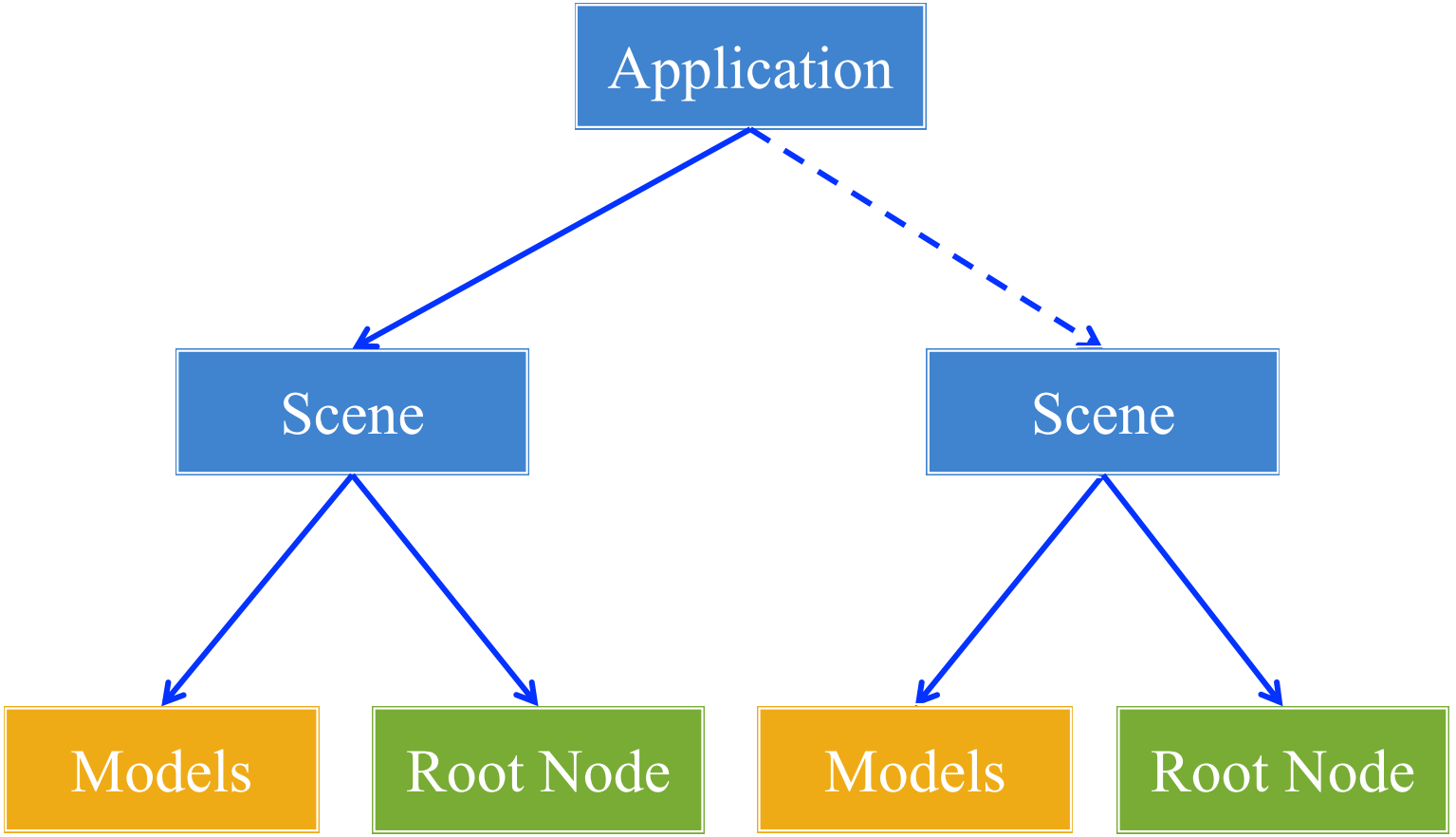


# The Game Loop and MVC

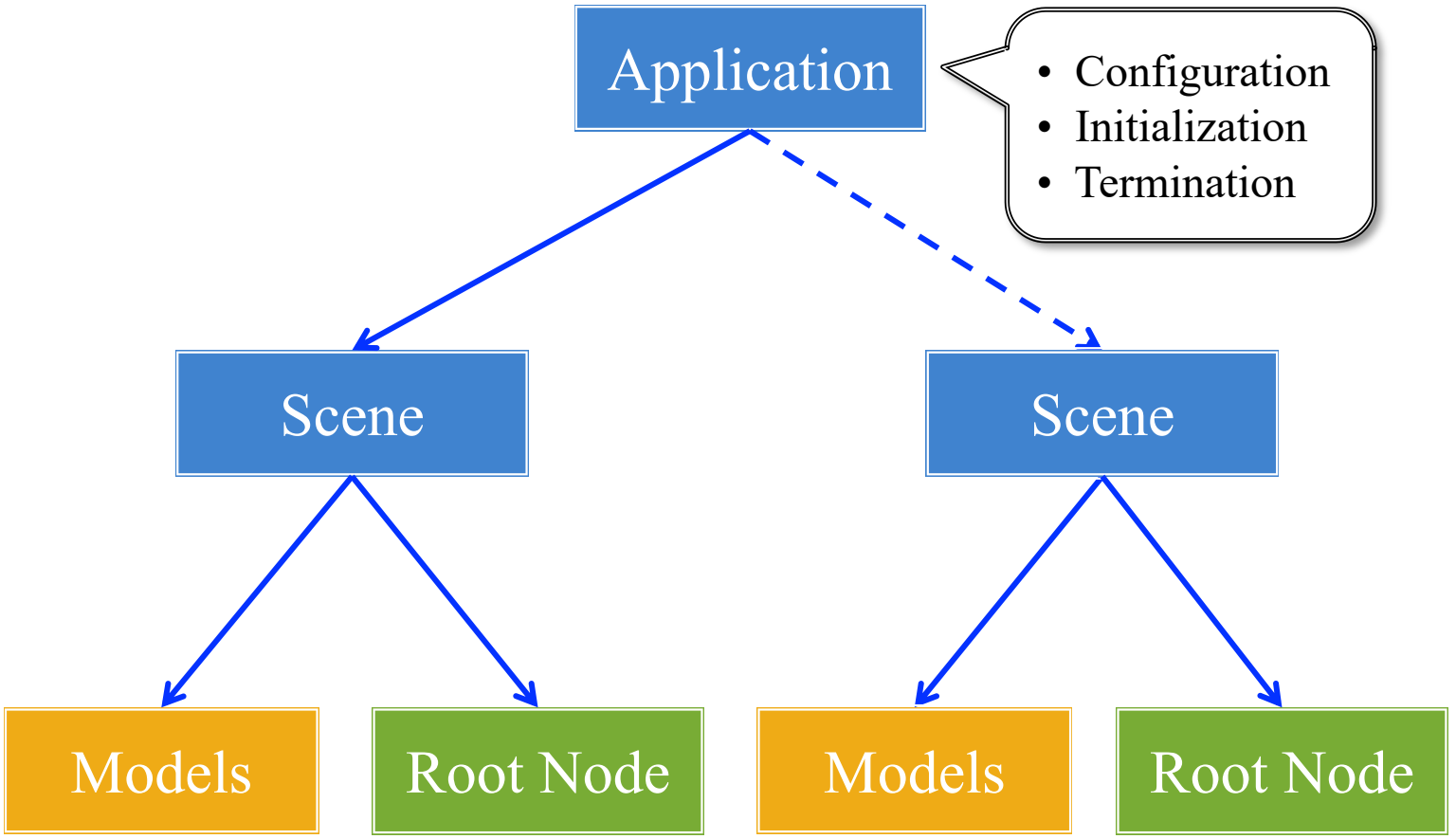
- Model: The game state
  - Value of game resources
  - Location of game objects
- View: The draw phase
  - Rendering commands only
  - Major computation in update
- Controller: The update phase
  - Alters the game state
  - Vast majority of your code



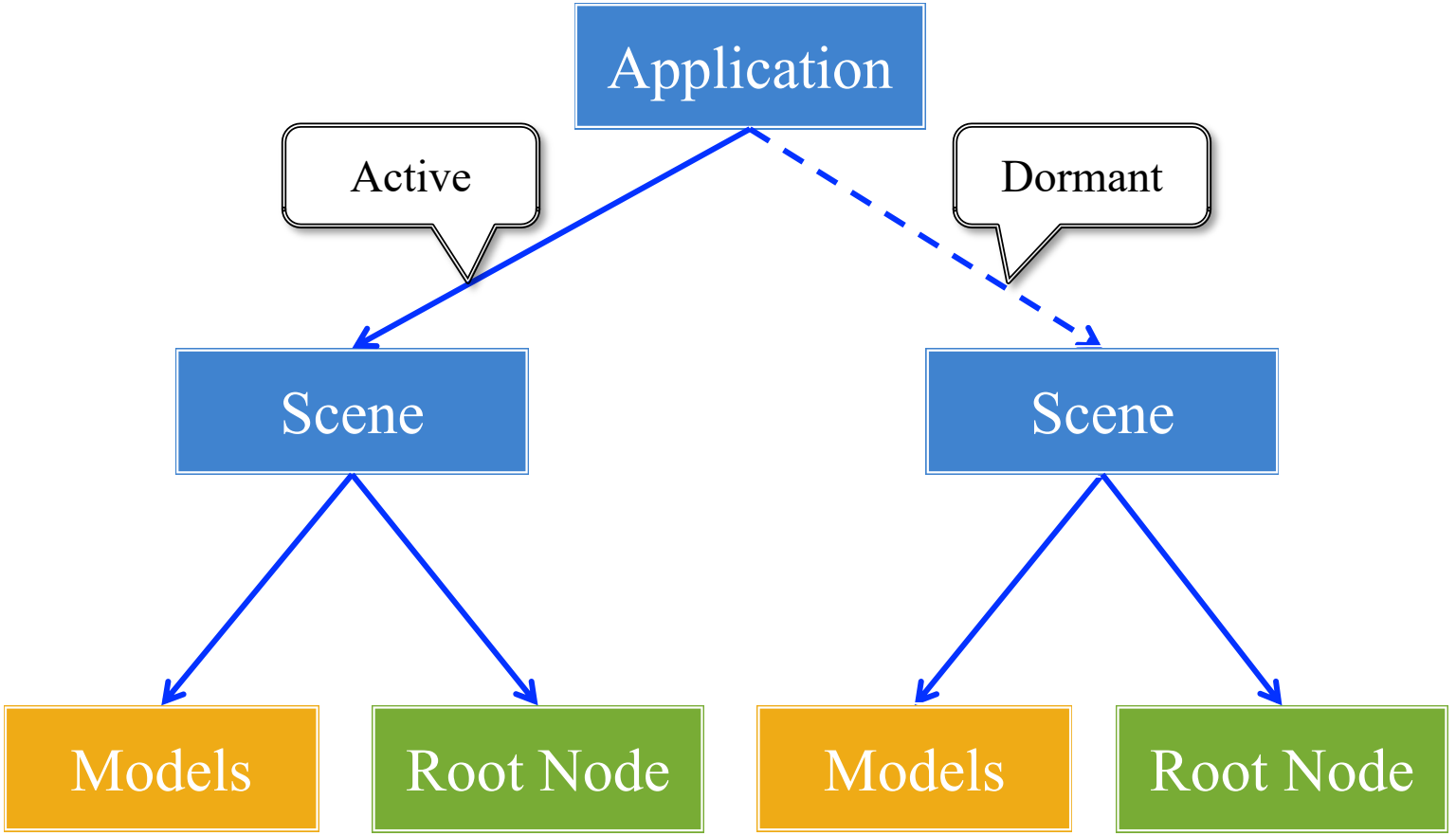
# Structure of a CUGL Application



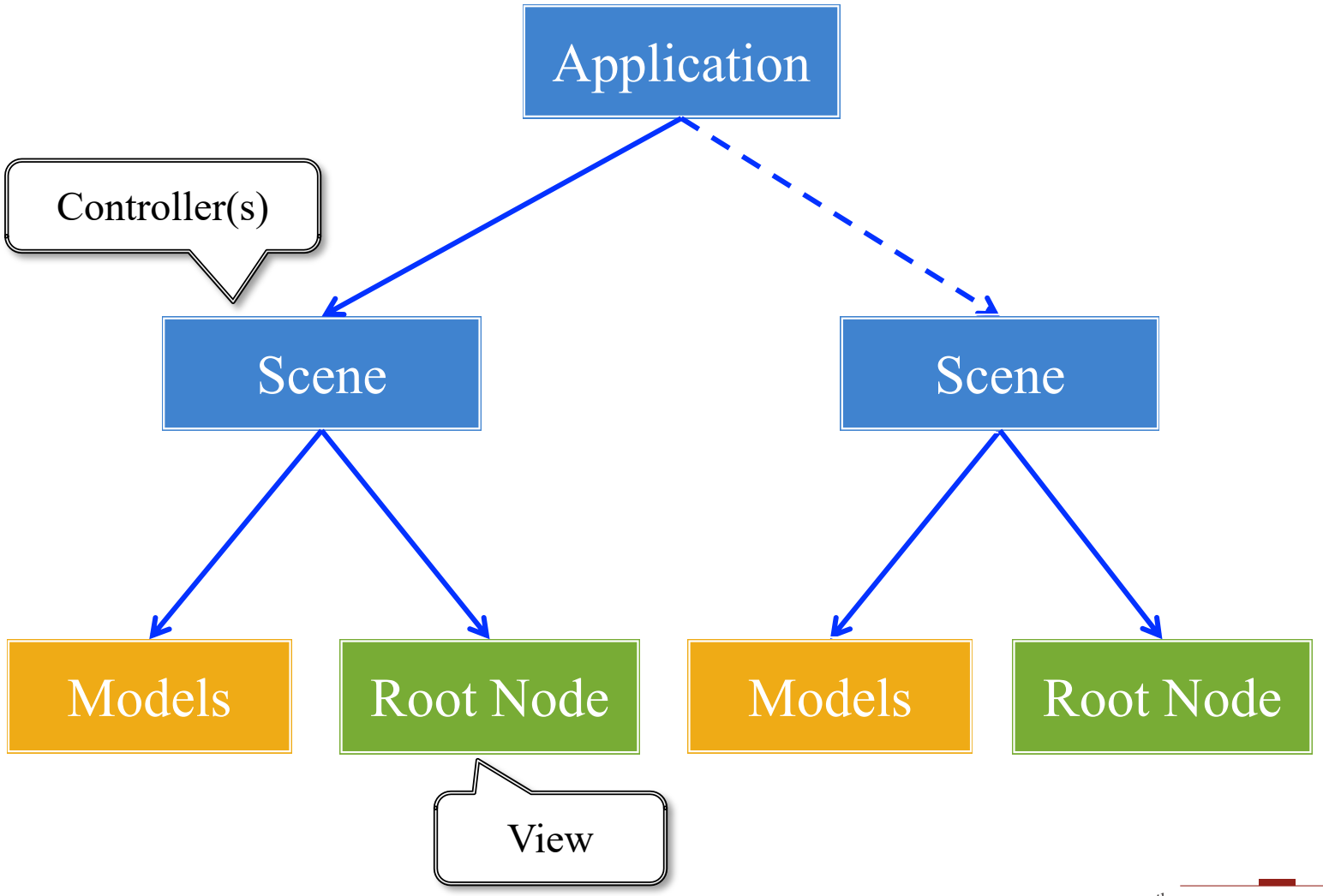
# Structure of a CUGL Application



# Structure of a CUGL Application



# Structure of a CUGL Application



# The Application Class

- CUGL and SDL3 do not use a `main()` function
  - They register specific callbacks with SDL3
  - This is better for asynchronous event support
- CUGL does this through the `Application` class
  - The `on`-methods (`onStartup`, etc.) are the callbacks
  - As are the method `update()` and `draw()`
- But you must *register* your Application **subclass**
  - You do this through the `CU_ROOTCLASS` macro
  - This creates a unique pointer and sends it to SDL3

# The Application Class

- CUGL and SDL3 do not use a `main()` function
  - They register specific callbacks with SDL3
  - This is better for asynchronous event support
- CUGL does this through the `Application` class
  - The `on`-methods (`onStartup`, etc.) are the callbacks
  - As are the method `update()` and `draw()`
- But you must *register* your Application **subclass**

This happens at compile time, not runtime!

# The Application Lifestages

## The Constructor

- Effectively your main()
  - Called before anything else
  - The backend not initialized!
  - Cannot use SDL3 functions
- Defines the app settings
  - The application name
  - The initial window size
  - Things SDL3 needs to start
- Nothing else!

## onStartup()

- **After** backend is initialized
- Loads the game assets
  - Attaches the asset loaders
  - Loads immediate assets
- Starts any global singletons
  - **Example**: `AudioEngine`
- Creates any scenes
  - But does not launch ***yet***
  - Waits for assets to load

# The Application Lifestages

## 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

## draw()

- Called each animation frame
- Delegates to `Scene` class
- `Scene` draws the scene graph
  - Activates the pipeline
  - Sends data to the pipeline
- Mostly done for you
  - Scene has all this code
  - Only override if you need a custom graphics pipeline

# The Application Lifestages

## onShutdown()

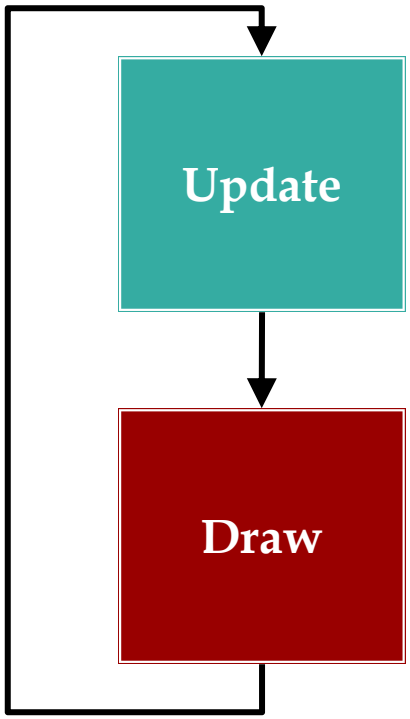
- Called when quitting
  - Allows you to clean up
  - Delete all objects
  - Deactivate all services
- You **must** do this
  - Might leave zombie threads
  - SDL3 will crash if don't
- But generally easy
  - Set shared pointers to null

## Anything Else?

- Other on-methods exist
  - `onSuspend`/`onResume`
  - `onLowMemory`
  - `onResize`
- Default for them is okay
  - But might need to override
  - Will see this later
- See documentation for more

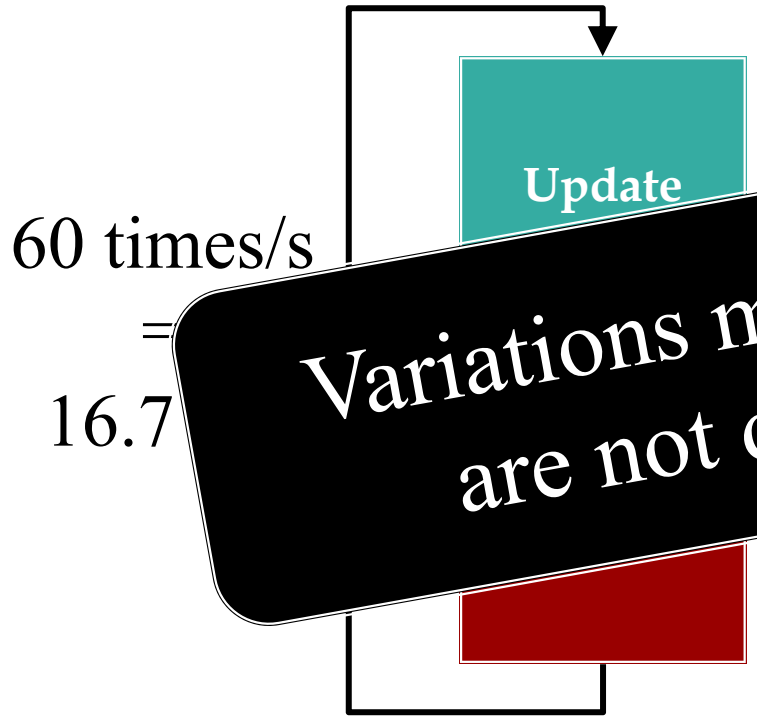
# Problems With the Game Loop

60 times/s
=
16.7 ms




- 16.7 ms **not guaranteed**!
  - Even for optimized code
  - Result of external factors
- **Regularly** see minor jitter
  - "In-between" code
  - Potential Vsync delay
- **Occasional** major jitter
  - Dynamic library loading
  - Cost of debugging tools

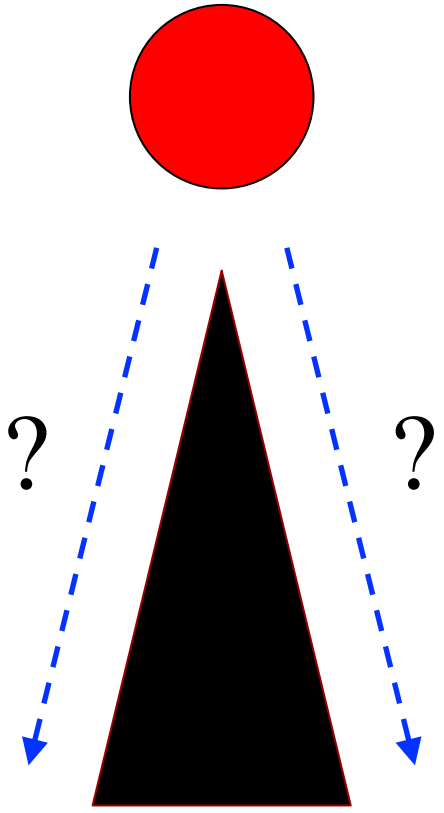
# Problems With the Game Loop




Variations mean simulations are not deterministic!

- 16.7 ms **not guaranteed**!
  - Even for optimized code
  - [illegible] factors
  - [illegible] r jitter
  - [illegible]otential Vsync delay
- **Occasional** major jitter
  - Dynamic library loading
  - Cost of debugging tools

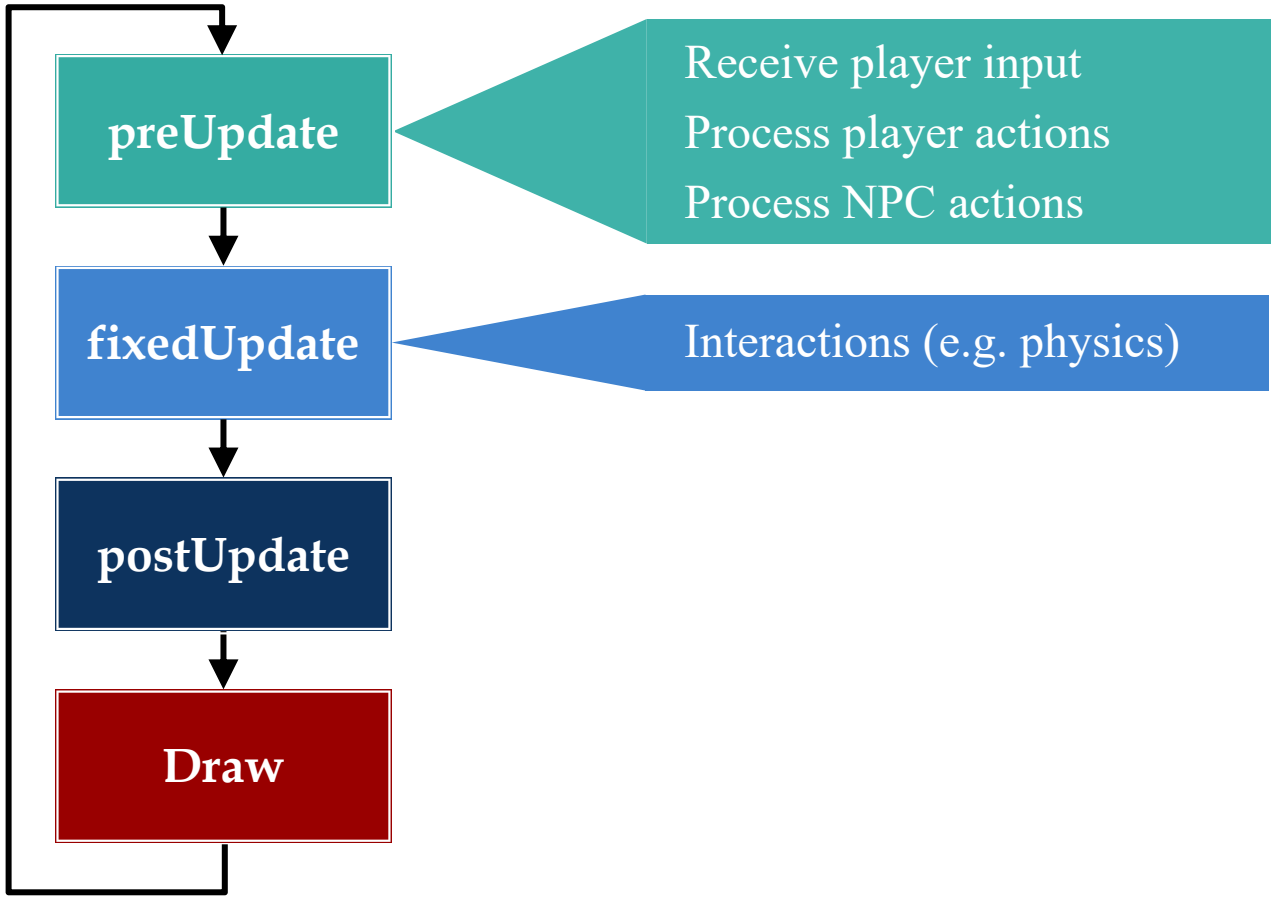
# Physics and Non-Determinism



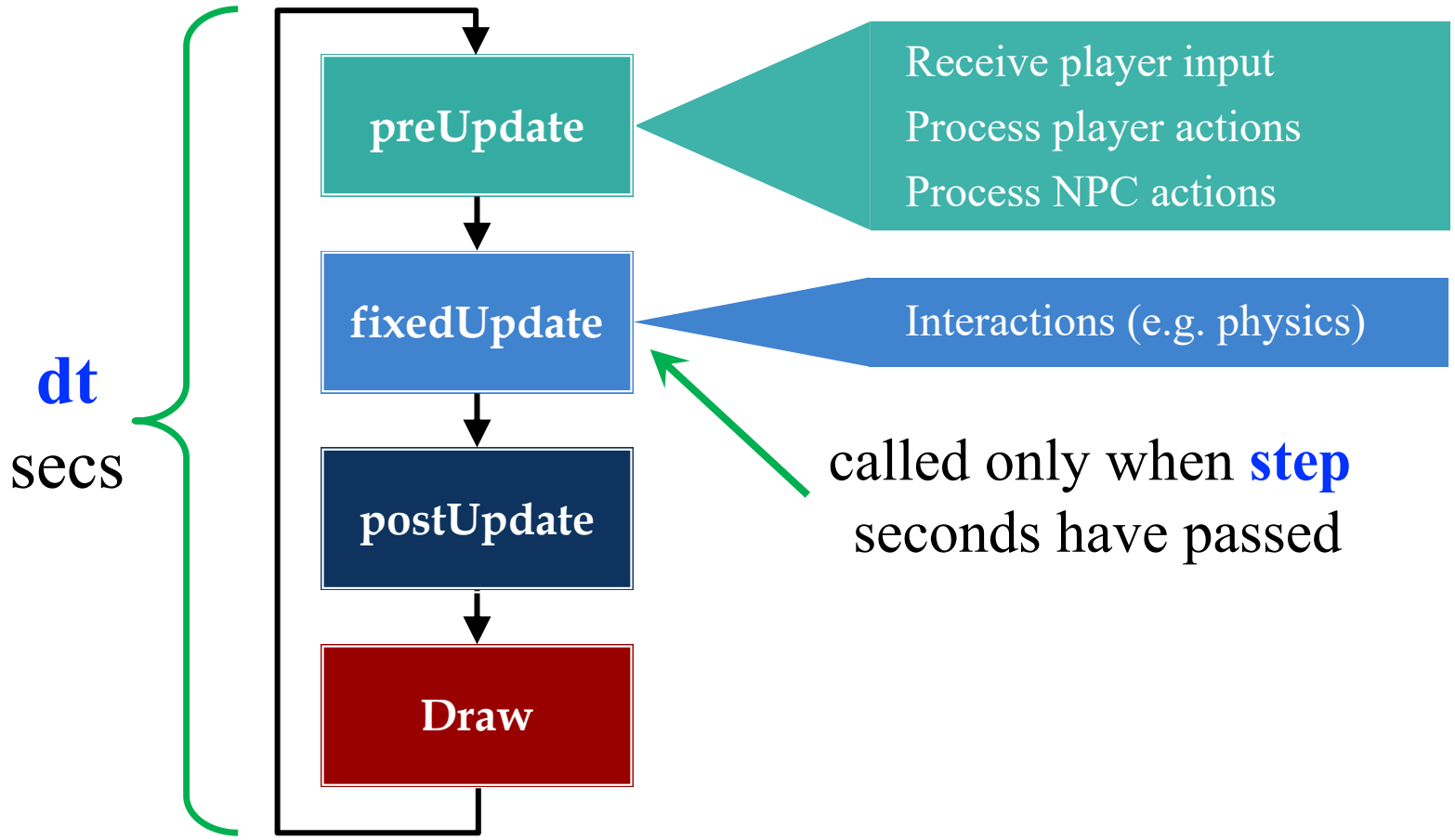
# How To Guarantee Determinism?

- Need to **decouple simulation** from other code
  - Cannot be delayed by drawing
  - Cannot be affected by OS externalities
- Put this on a **separate thread**?
  - Thread management still has some overhead
  - Have to **synchronize** with input/drawing thread (bad!)
- Create a **separate logical loop**?
  - Simulation loop runs at its own fixed rate
  - Draw method simply draws what it has so far

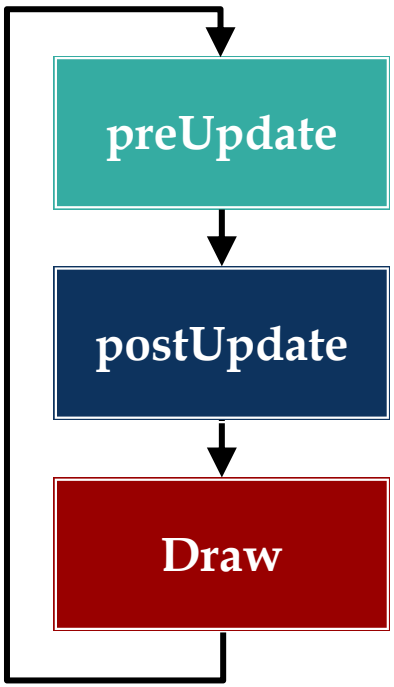
# The Game Loop Revisited

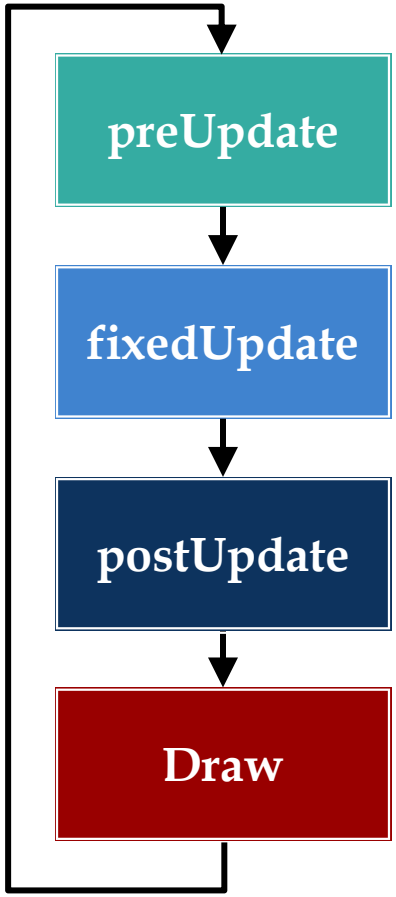


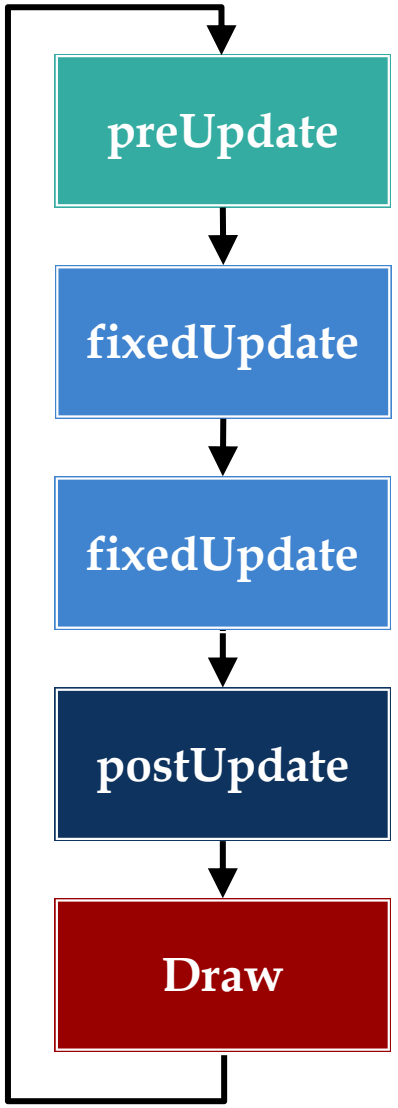
# The Game Loop Revisited



# These Are All Possible

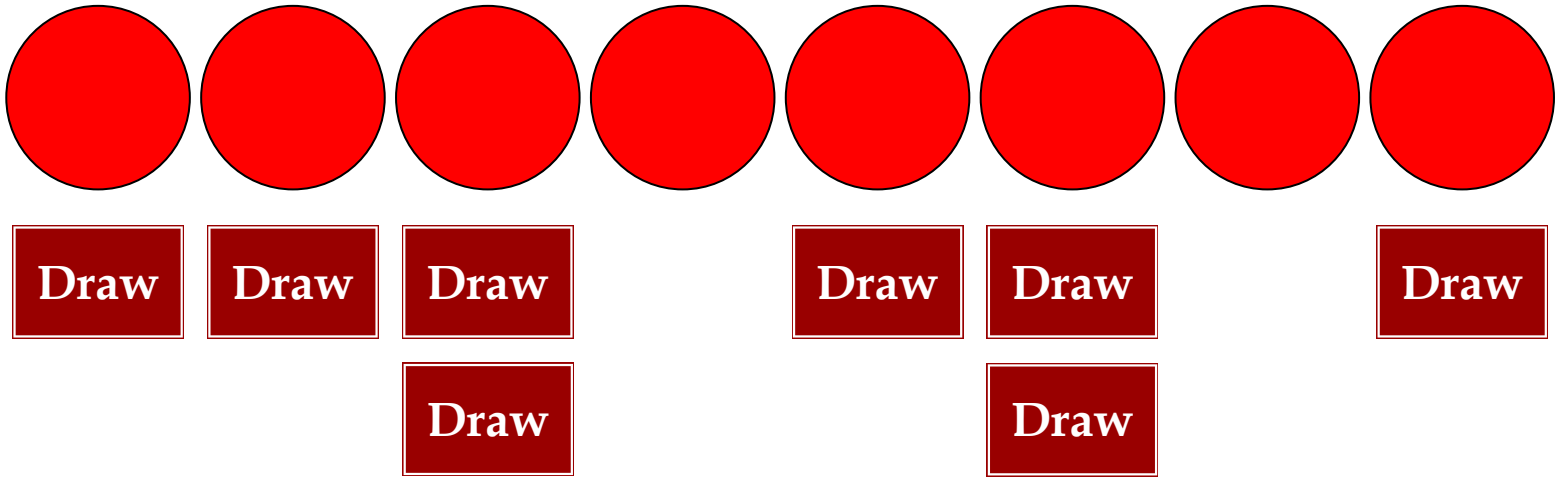


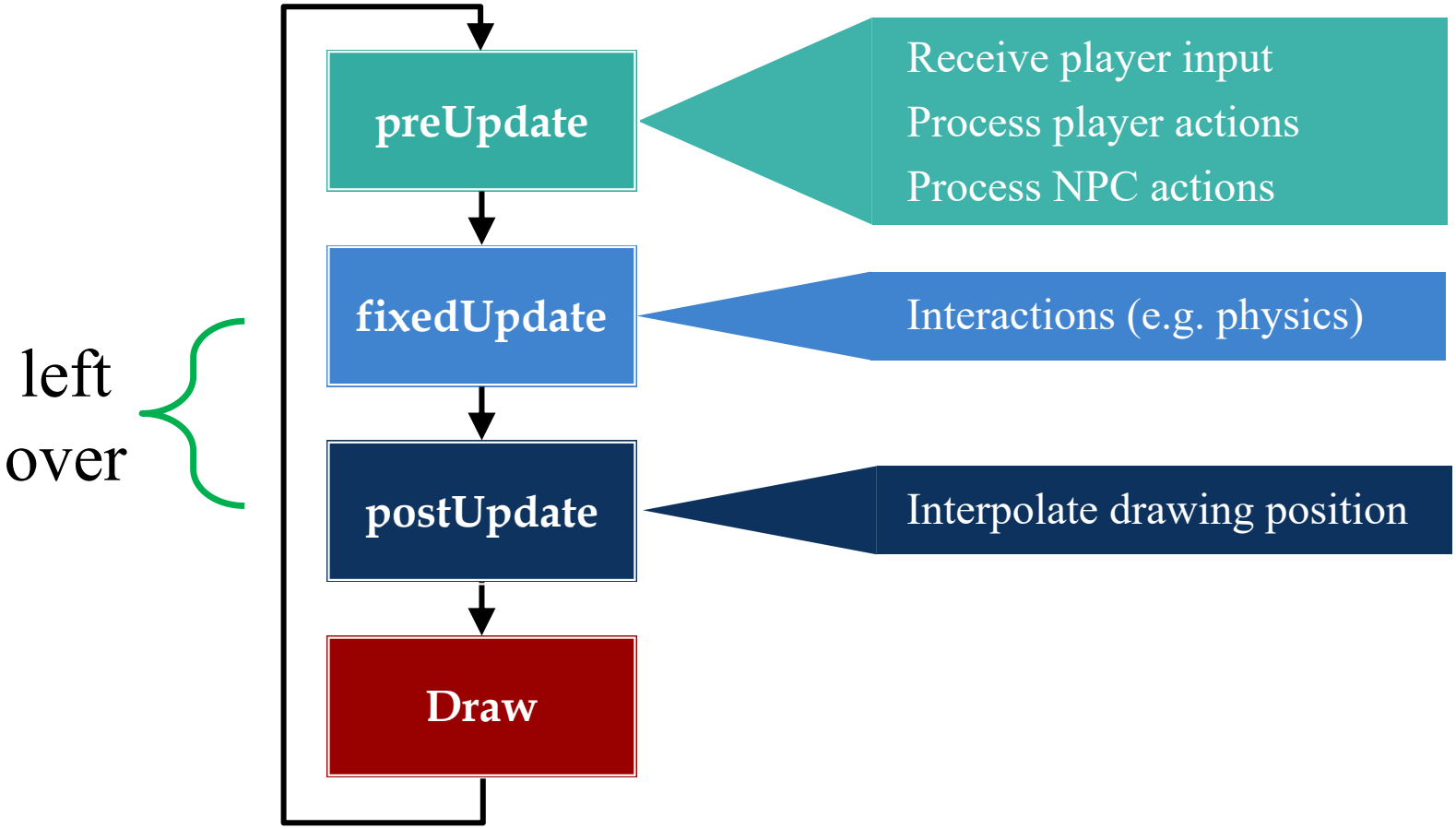



# Problem: Jerky Motion

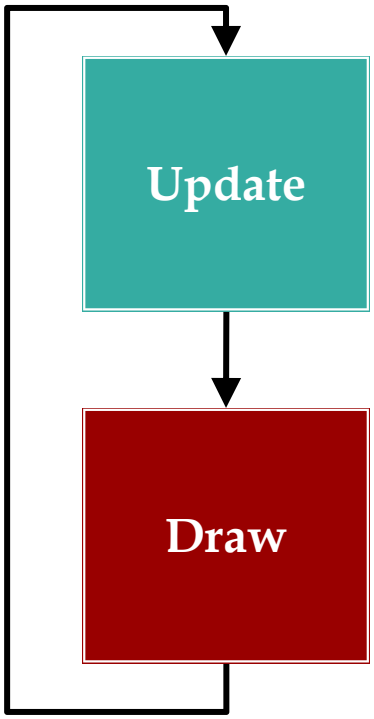
Each Image is a result of `fixedUpdate`



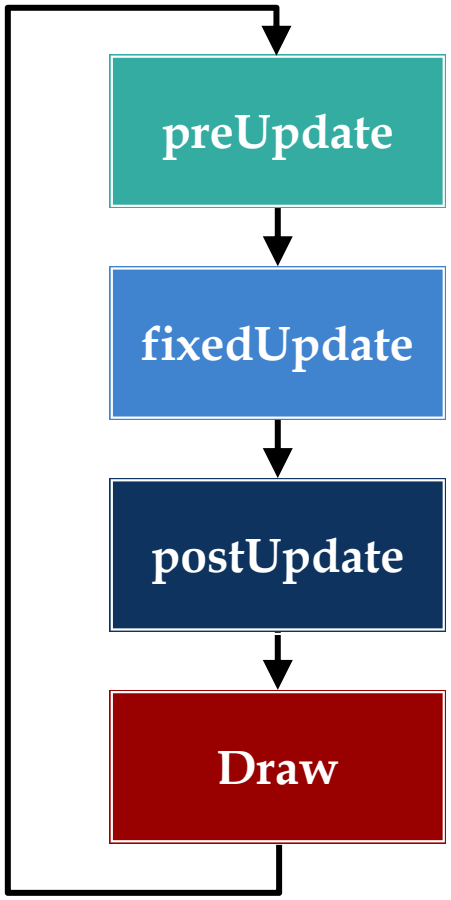
# The Game Loop Revisited



# CUGL Supports Both Loops

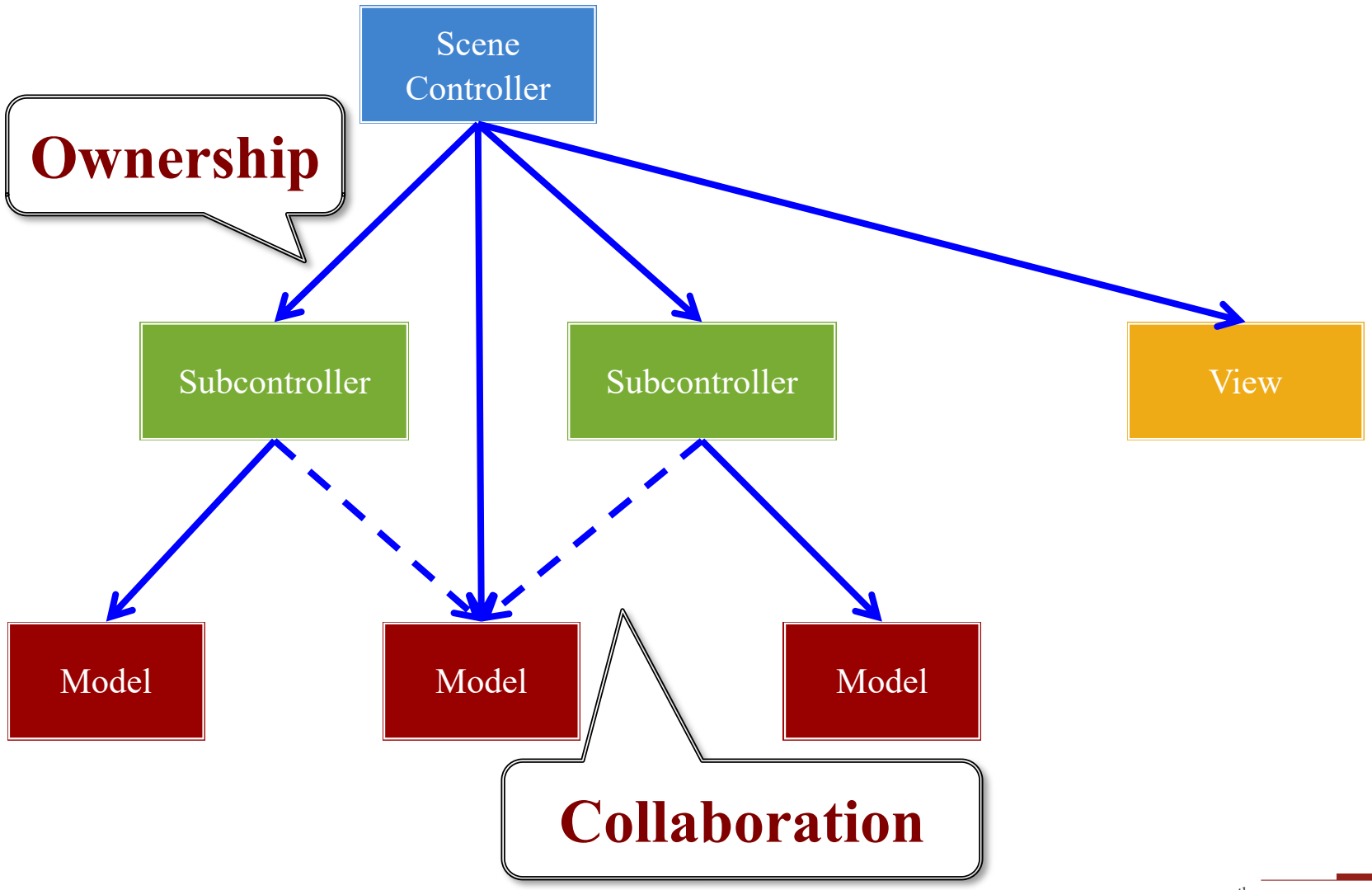



setDeterministic(false)

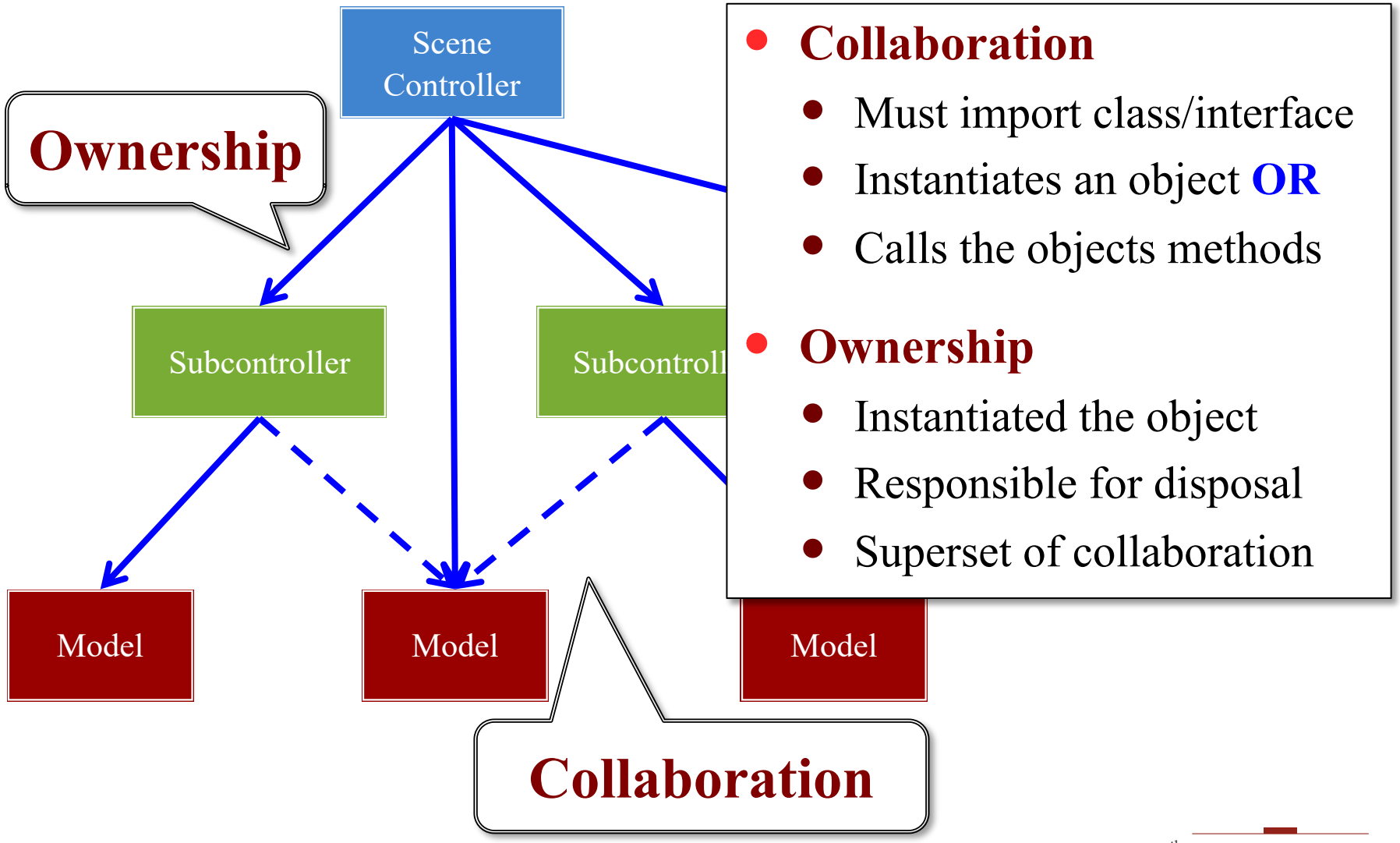



setDeterministic(true)

# Scene Structure

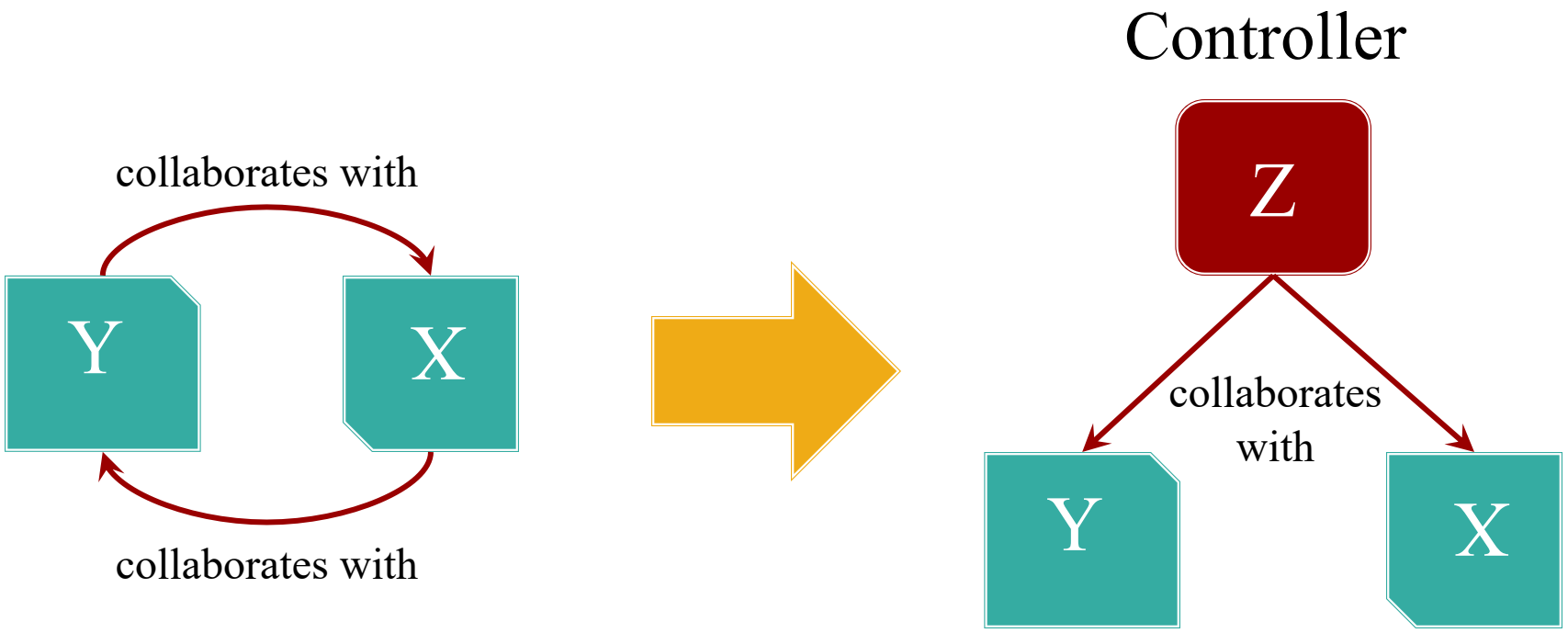


# Scene Structure

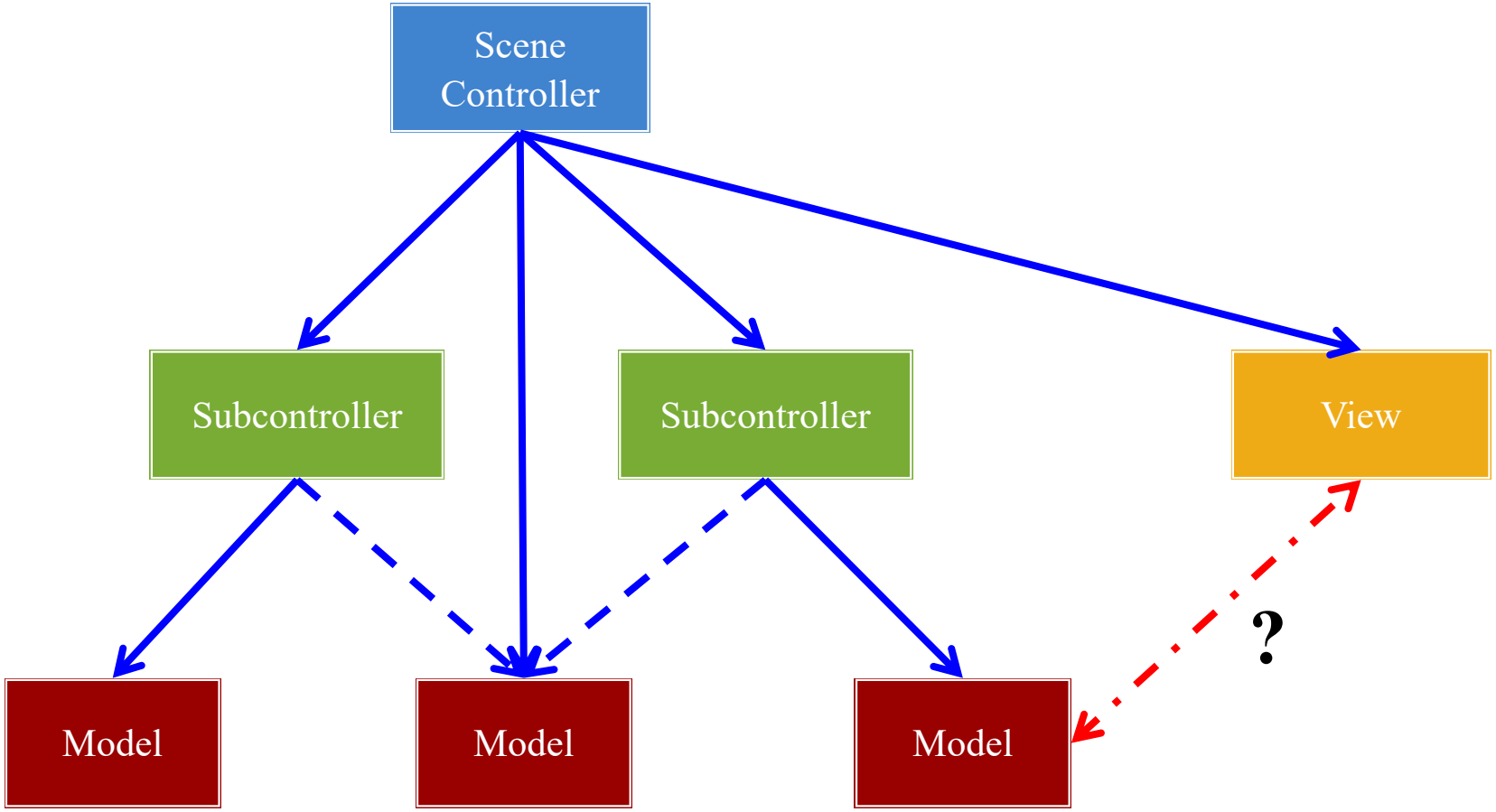



- **Collaboration**
  - Must import class/interface
  - Instantiates an object **OR**
  - Calls the objects methods
- **Ownership**
  - Instantiated the object
  - Responsible for disposal
  - Superset of collaboration

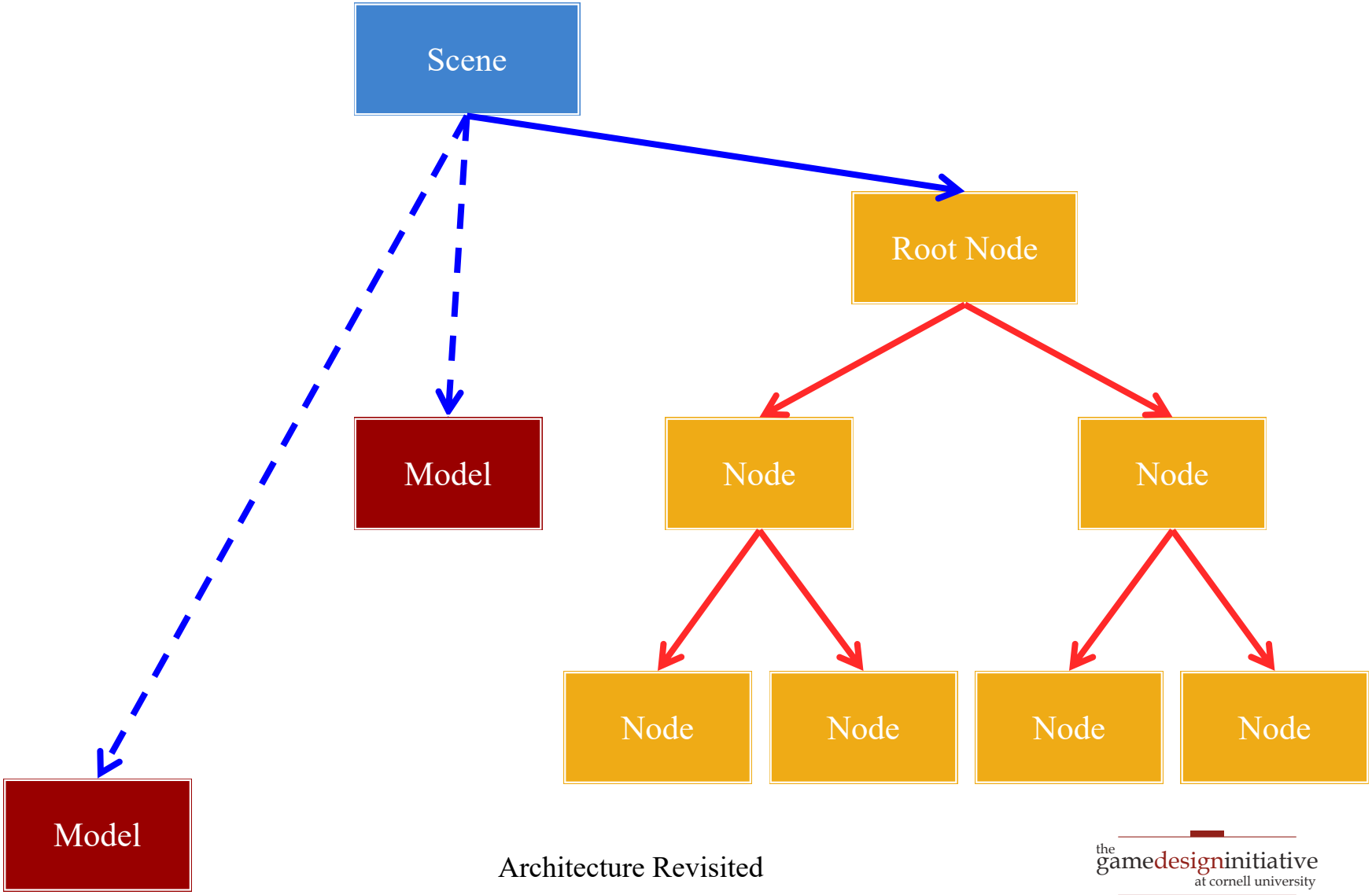
# Avoid Cyclic Collaboration



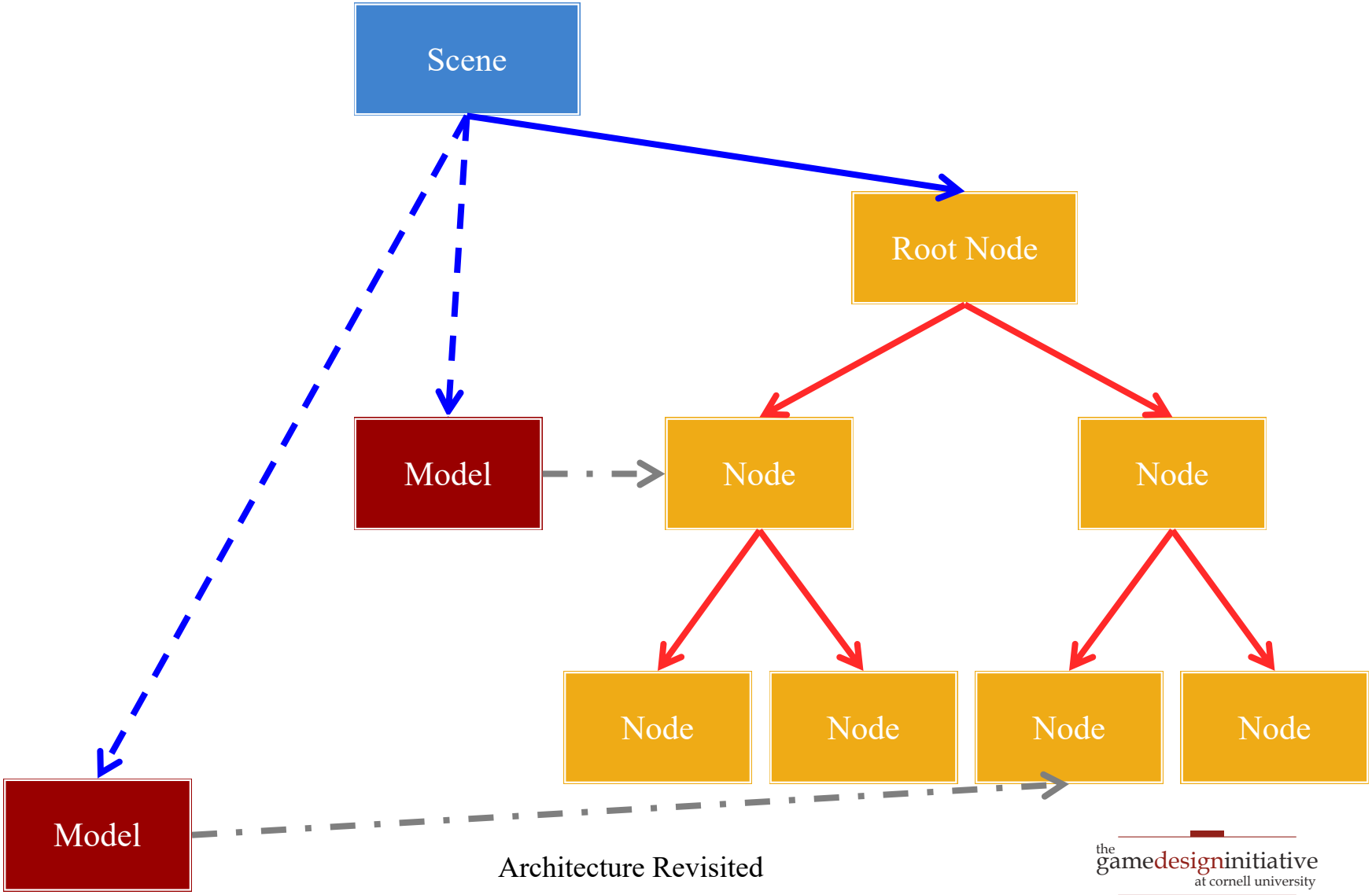
# Scene Structure



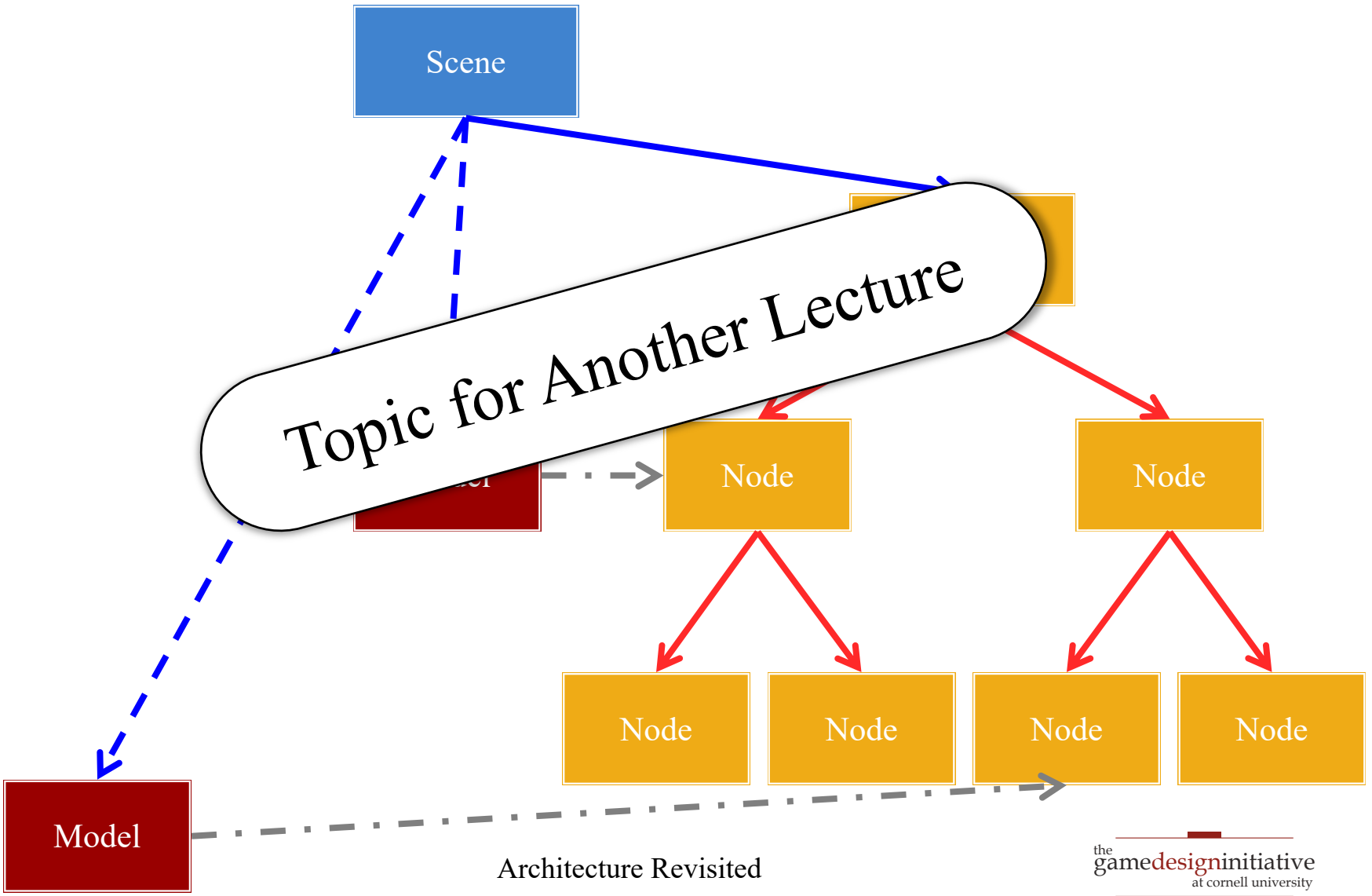
# CUGL Views: Scene Graphs



# CUGL Views: Scene Graphs



# CUGL Views: Scene Graphs



# 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

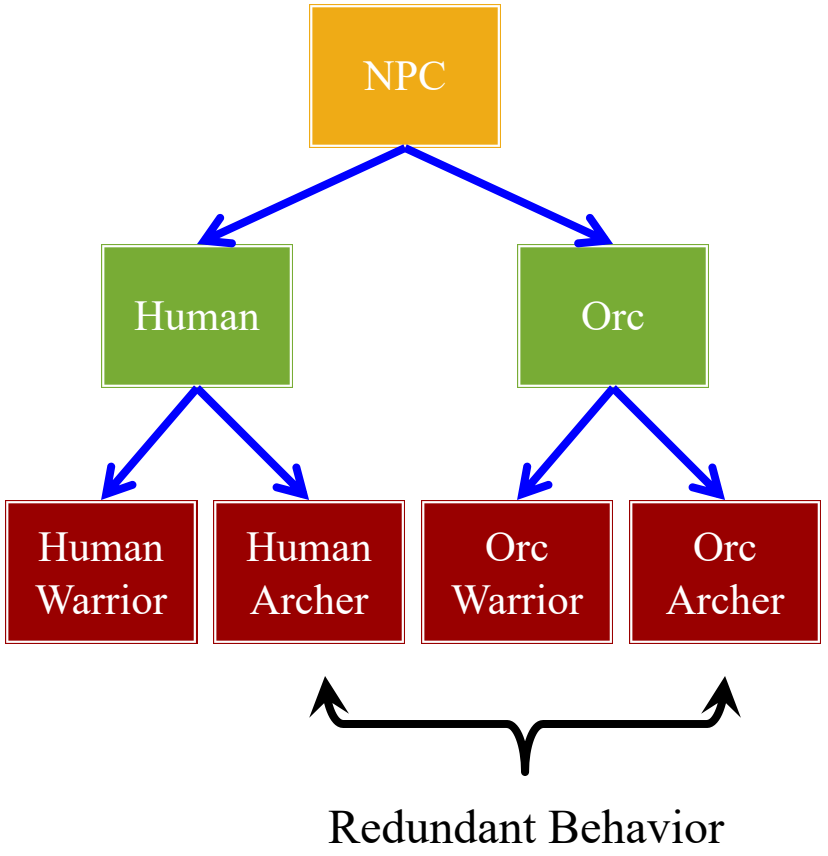
Traditional controllers are "lightweight"

# 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?

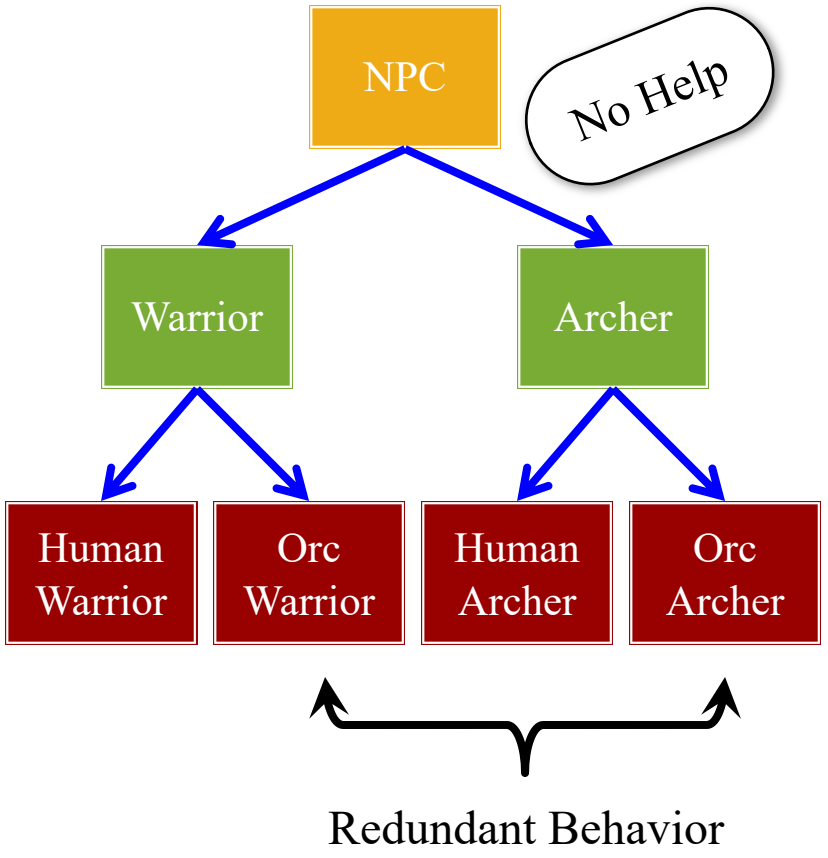
# Problem with Subclassing

- 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



# Problem with Subclassing

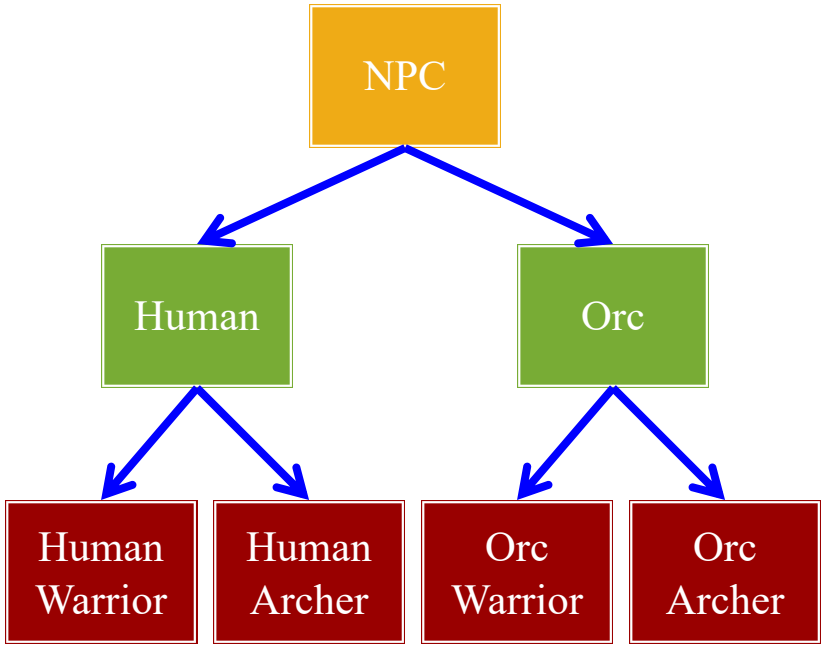
- Games have *lots* of classes
  - Each game entity is different
  - Needs its own functionality (e.g. object methods)
- Want to avoid **redundancies**
  - Makes code hard to change
  - Common source of bugs
- Might be tempted to **subclass**
  - Common behavior in parents
  - Specific behavior in children



# 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

In this case, models are lightweight

## 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

# Entity-Component Model

- Keep models as **pure data**
  - Essentially C-structs with no methods
  - Allows you to store as cache-friendly arrays
- **Components** provide the model **methods**
  - Designed as separate controller classes/modules
  - Models can mix-and-match components
- **Problem:** very different way of architecting code
  - Pays off in tight behavior loops (cache-optimal)
  - Less pay-off otherwise

# What is the Problem We Want to Solve?

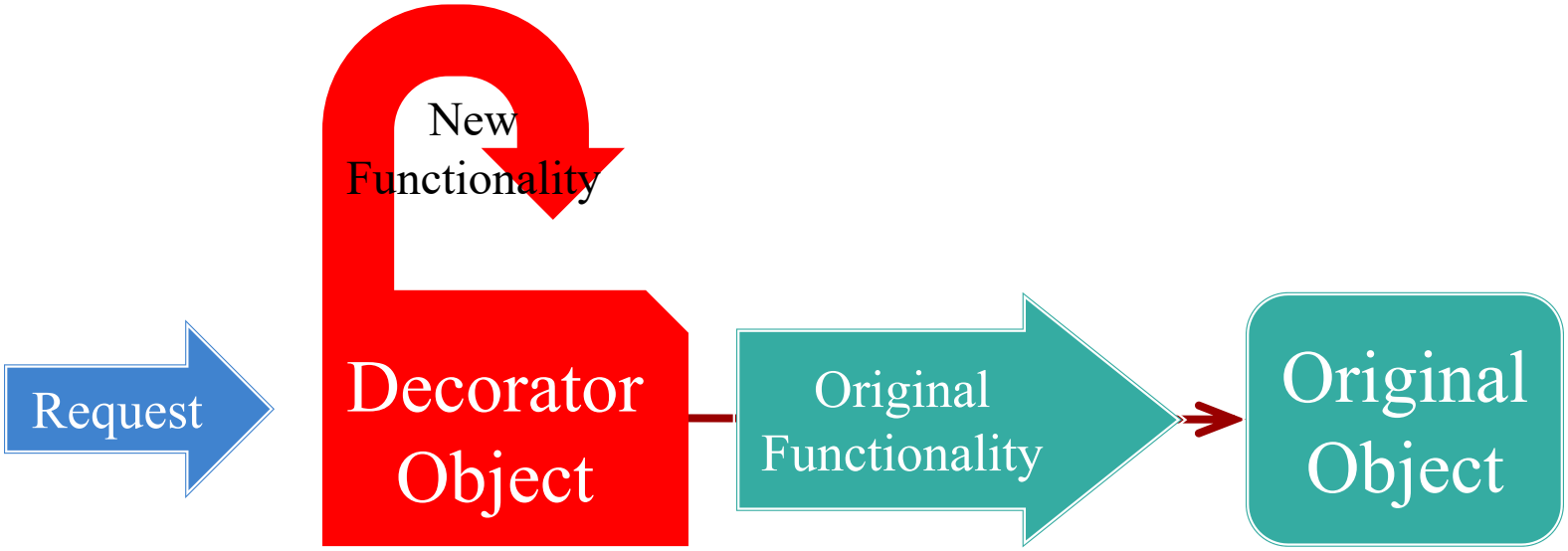
## Classes/Types are Nouns

- 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-do*
  - **Example**: Orc `can-do` attack
  - Not necessarily tied to class
  - **Example**: swapping items

# Possible Solution: Decorator Pattern



# Java I/O Example

```
InputStream input = System.in;
```

Built-in console input

```
Reader reader = new InputStreamReader(input);
```

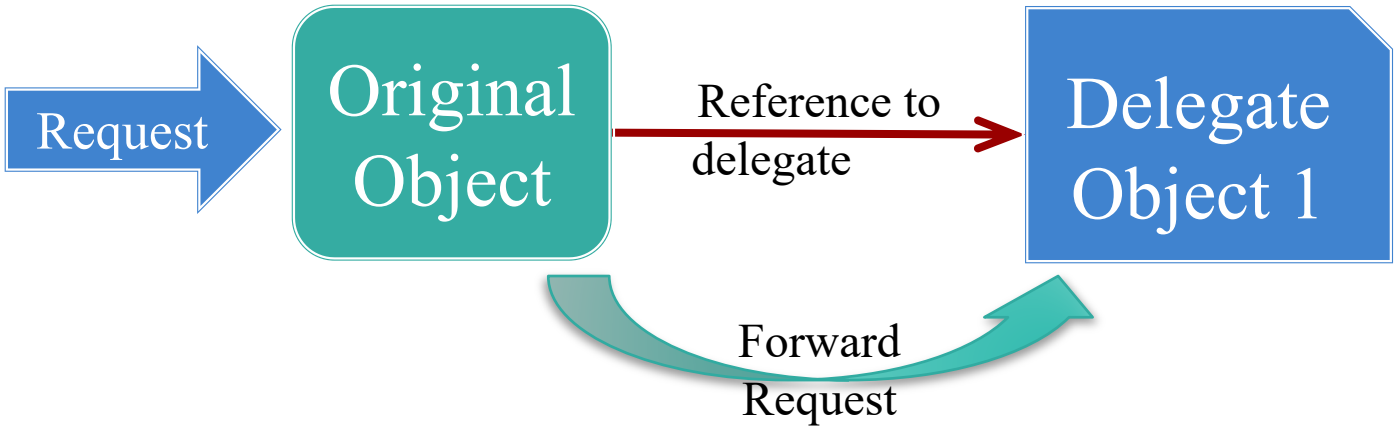
Make characters easy to read

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

Read whole line at a time

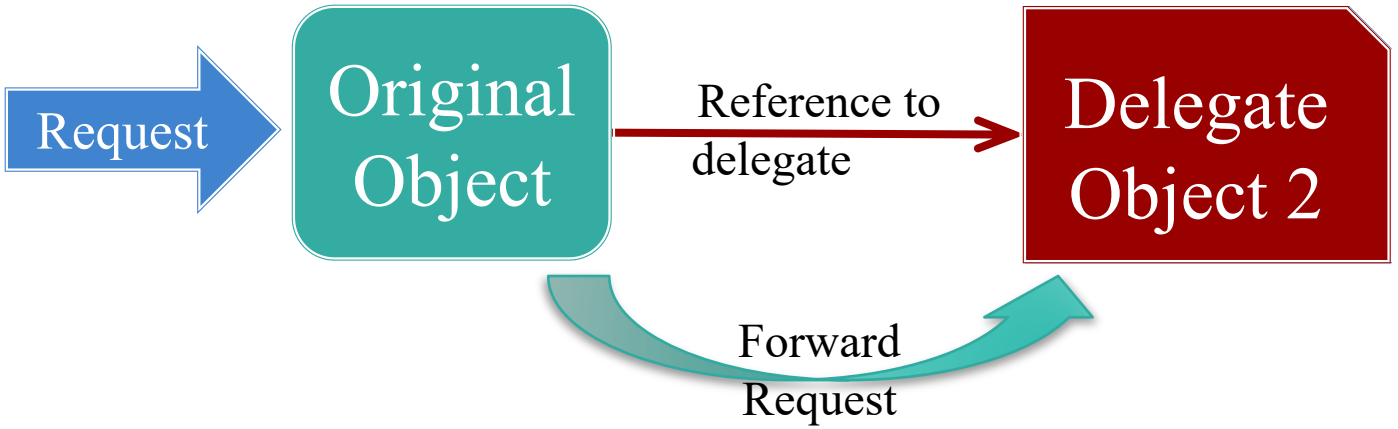
Most of java.io works this way

# Alternate Solution: Delegation Pattern




*Inversion* of the Decorator Pattern

# Alternate Solution: Delegation Pattern




*Inversion* of the Decorator Pattern

# Comparison of Approaches

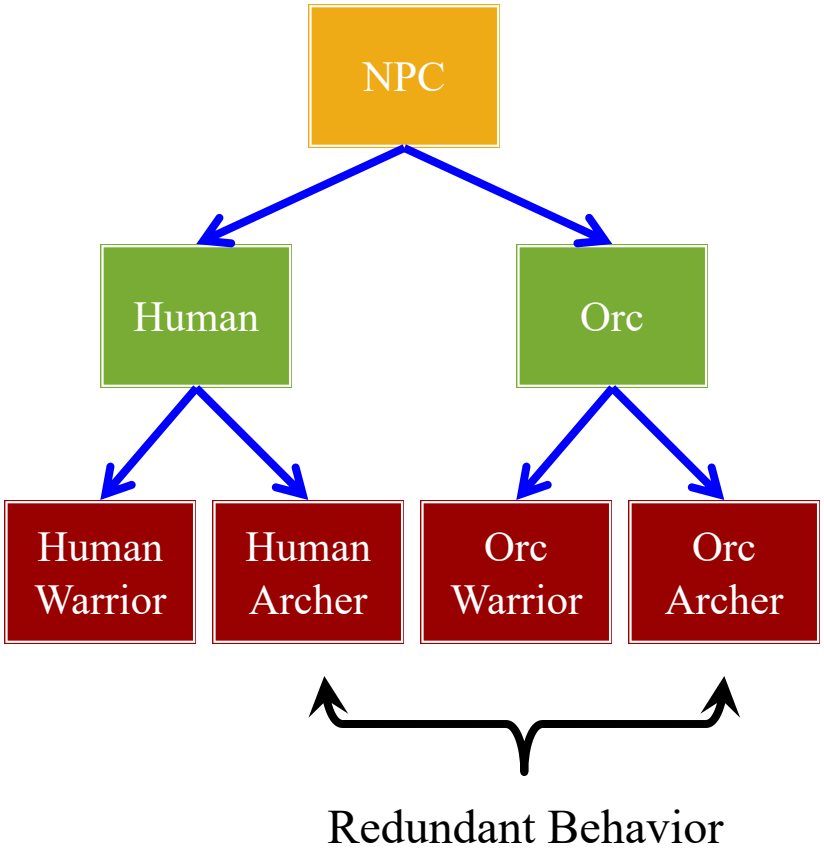
## Decoration

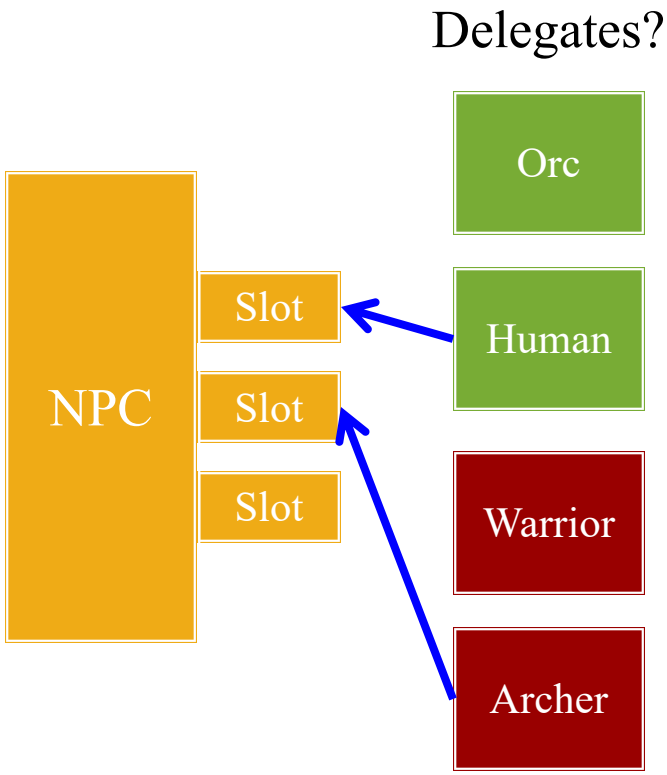
- Pattern applies to *decorator*
  - Given the original object
  - Requests through decorator
- **Monolithic** solution
  - Decorator has all methods
  - "Layer" for more methods (e.g. Java I/O classes)
- Works on *any* object/class

## 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



# 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