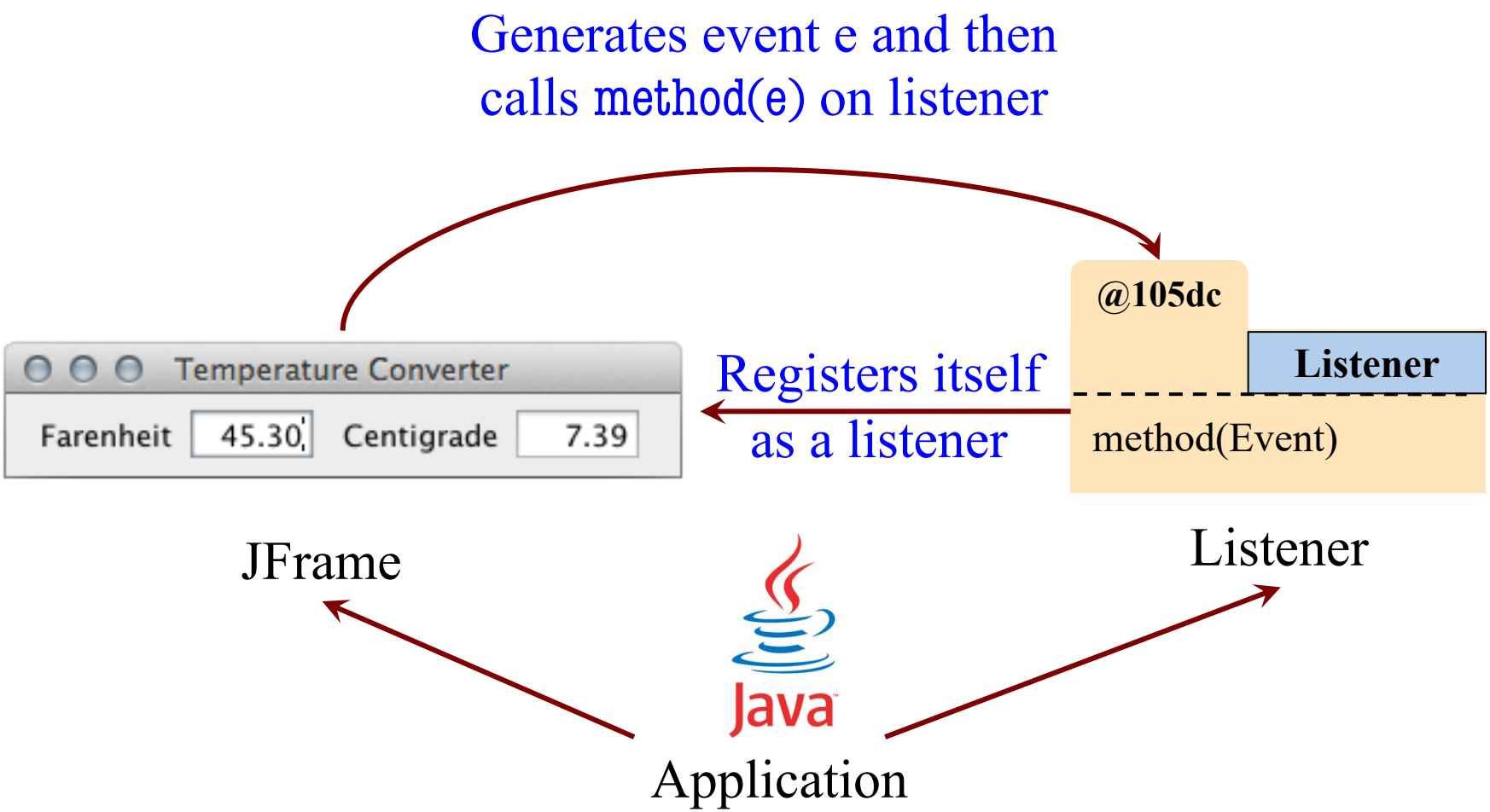
the gamedesigninitiative at cornell university

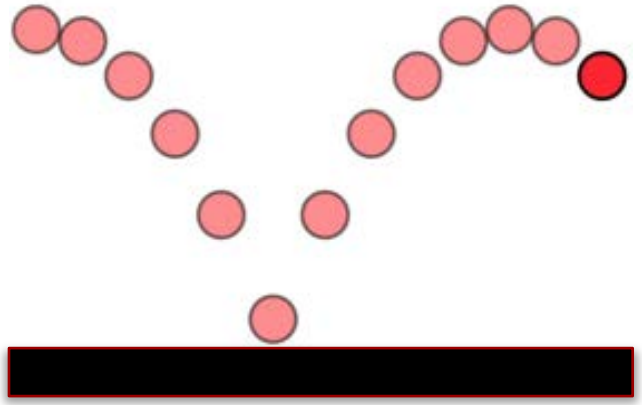
Game Loop

2110-Level Apps are Event Driven

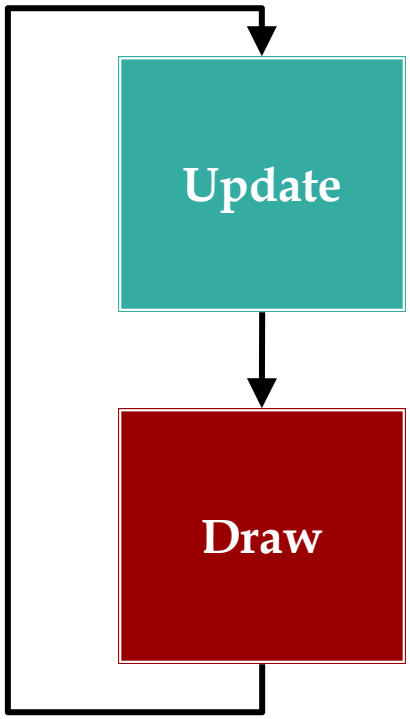


Limitations of the Event Model

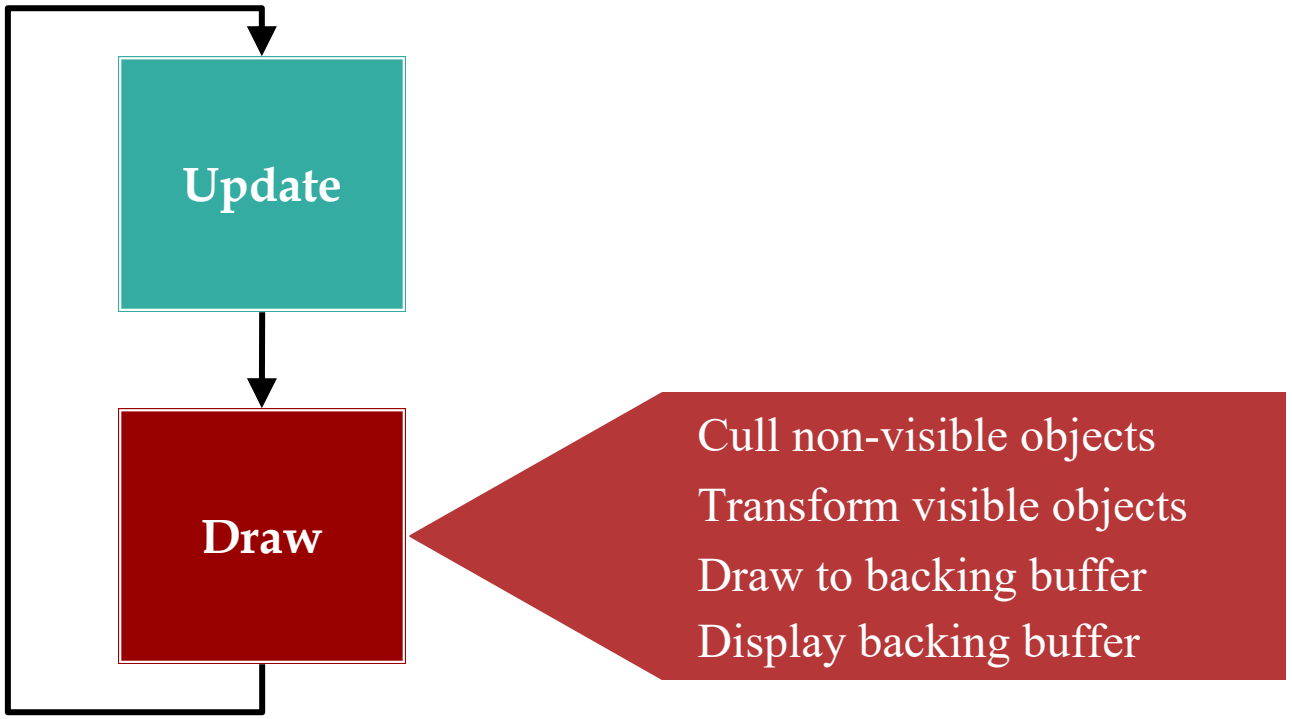
- Program only reacts to user input
  - Nothing changes if user does nothing
  - Desired behavior for productivity apps
- Games continue without input
  - Character animation
  - Clock timers
  - Enemy AI
  - Physics Simulations



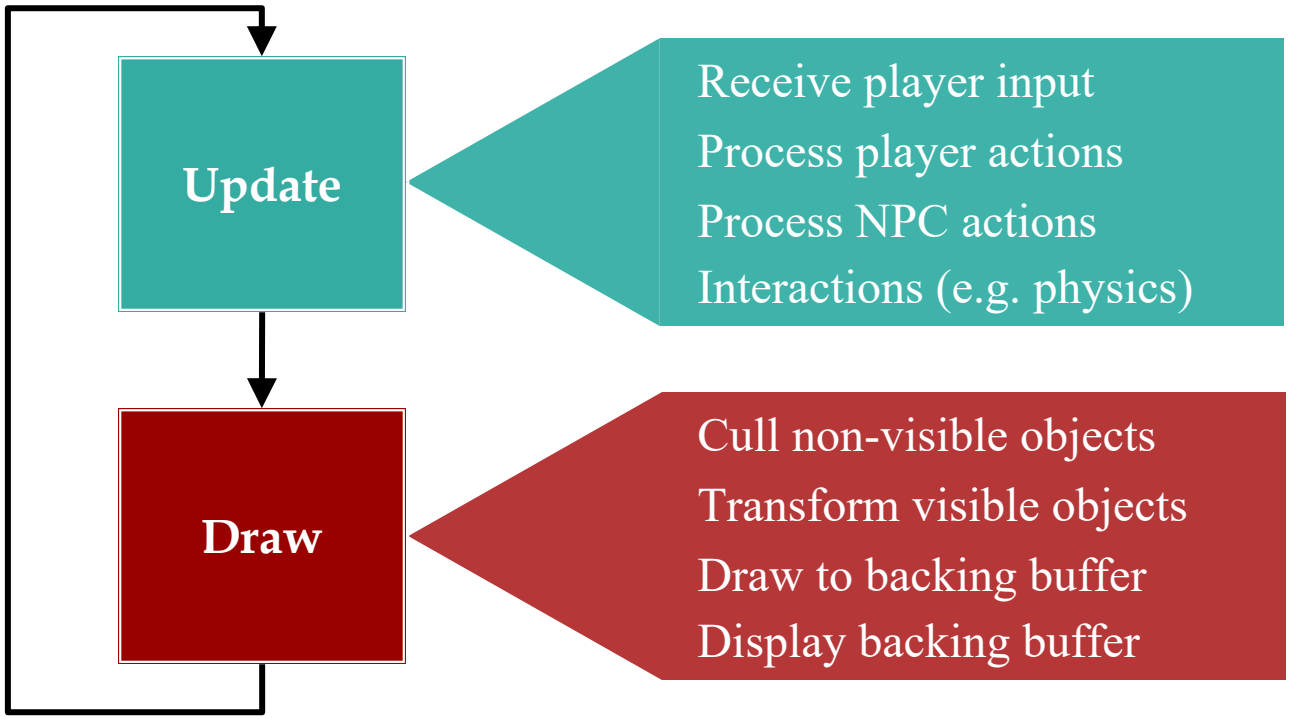
The Game Loop



The Game Loop

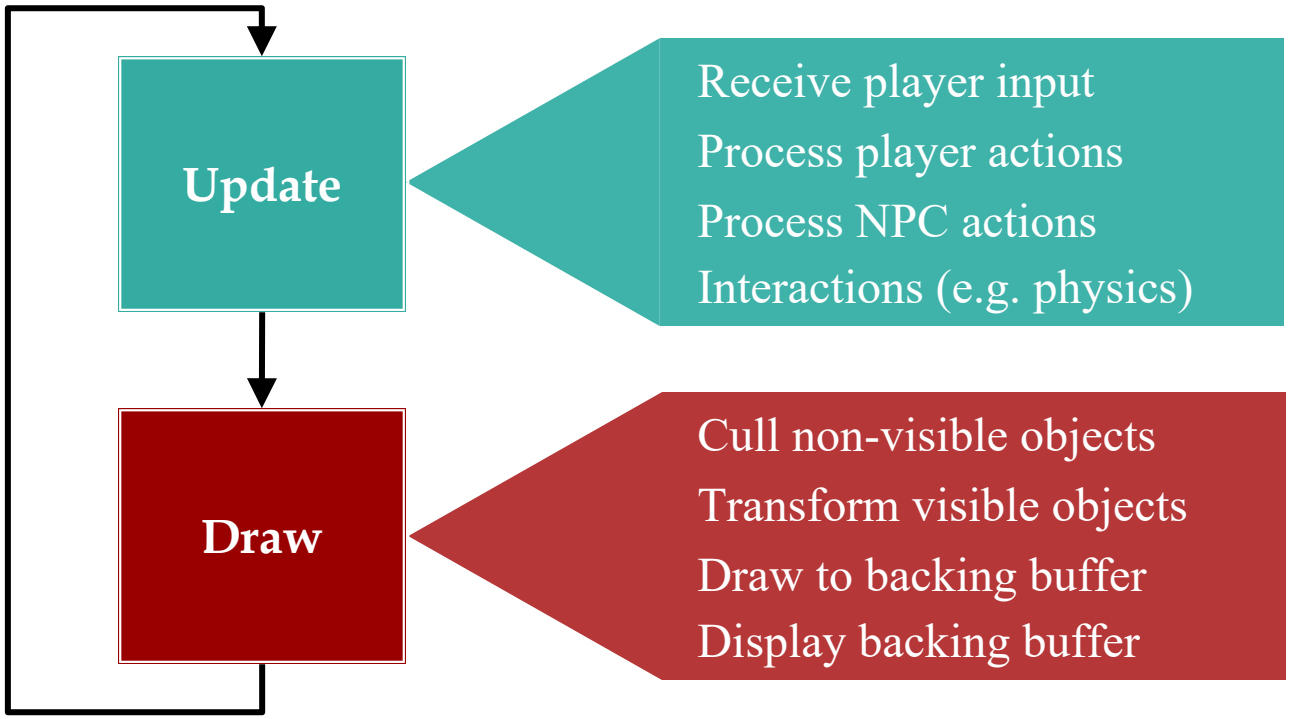


The Game Loop



The Game Loop

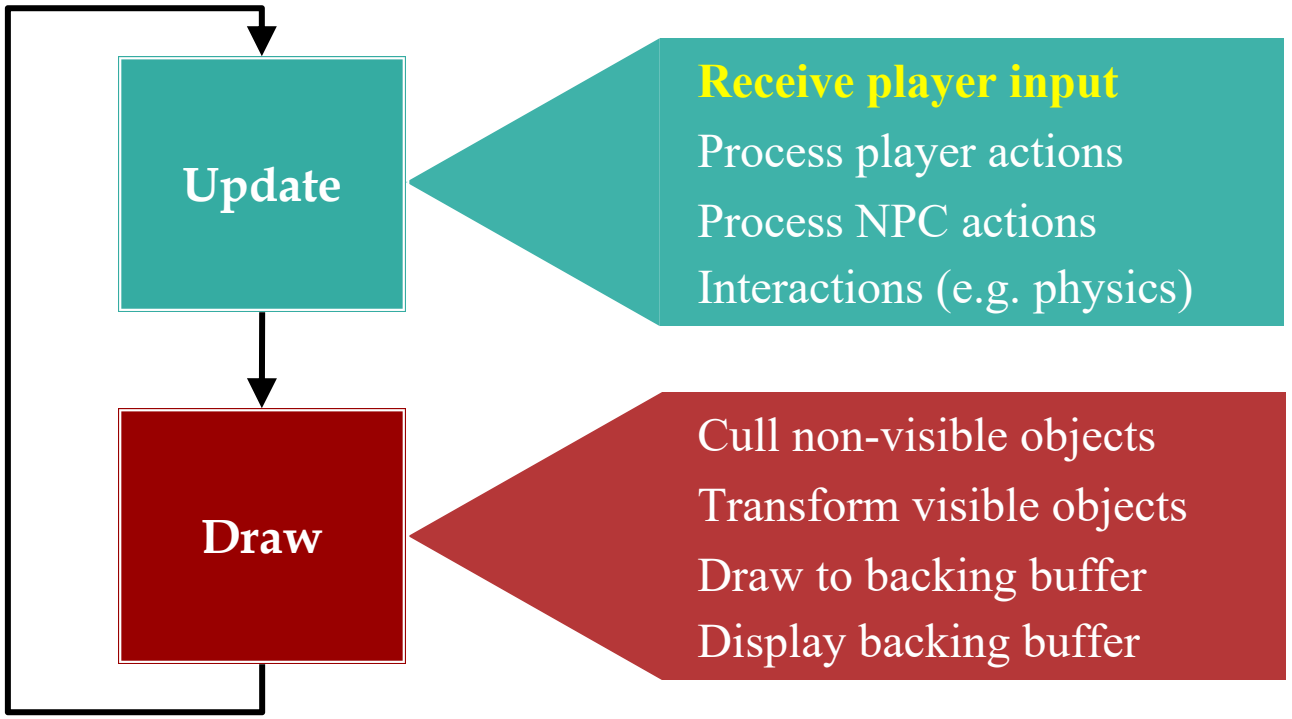
60 times/s
=
16.7 ms



Few Words on Drawing

- Drawing needs to be **fast**!
  - Do as little computation as possible
  - But draw as few objects as possible
- Is this a contradiction?
  - Need to compute what to draw
  - So drawing *less* has extra overhead
- **Rule**: do **not** modify game state in draw
  - Any extra computation is local-only

The Game Loop



Player Input

- Traditional input is event-driven
  - Events capture state of controller
  - OS/VM generates events for you
  - Listeners react to events

- Game loop uses **polling** for input
  - Ask for controller state at start of loop
  - **Example**: What is joystick position?
  - If no change, do no actions that loop

Problem with Polling

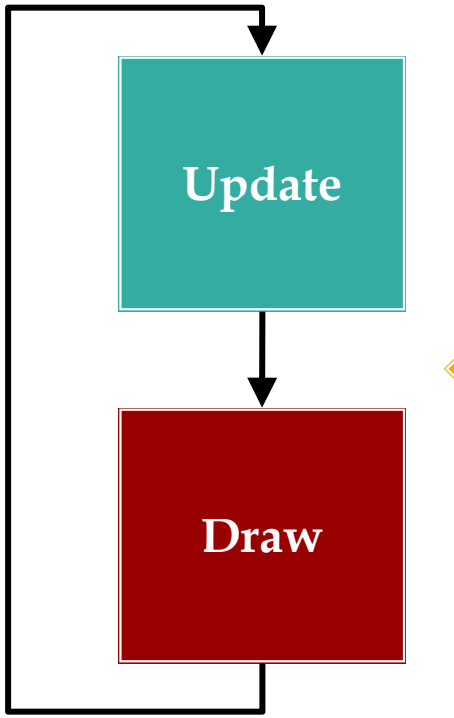
- Only one event per update loop
  - Multiple events are lost
  - **Example**: Fast typing
- Captures state at beginning
  - Short events are lost
  - **Example**: Fast clicks

- Event-driven mostly avoids these problems
  - Captures all events as they happen
  - But capture still has a frame-rate **resolution**

Combining Input Approaches

- LibGDX input is extremely flexible
  - Every input type supports events OR polling
- **Polling**: `Input` interface
  - Access it through the static class GDX.Input
  - Allows you to read the input state right now
- **Events**: `InputProcessor` interface
  - Register it with the appropriate input device
  - Works exactly like Swing listeners

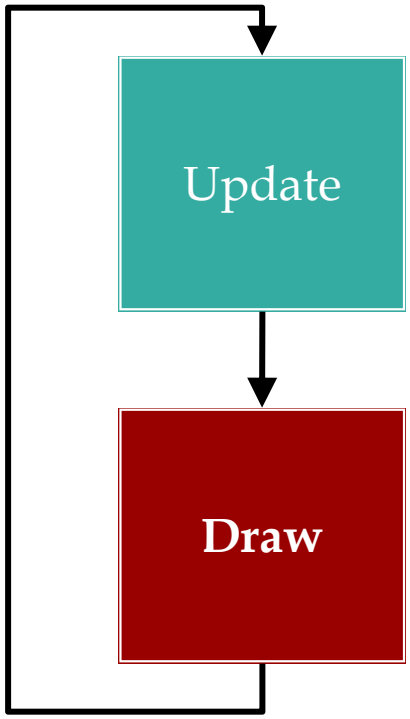
Problem: Timing



How do these fit together?

```
public class MyProcessor implements
      InputProcessor {


   public void keyTyped(char c) {
      //  Do something with input
   }

}
```

Problem: Timing



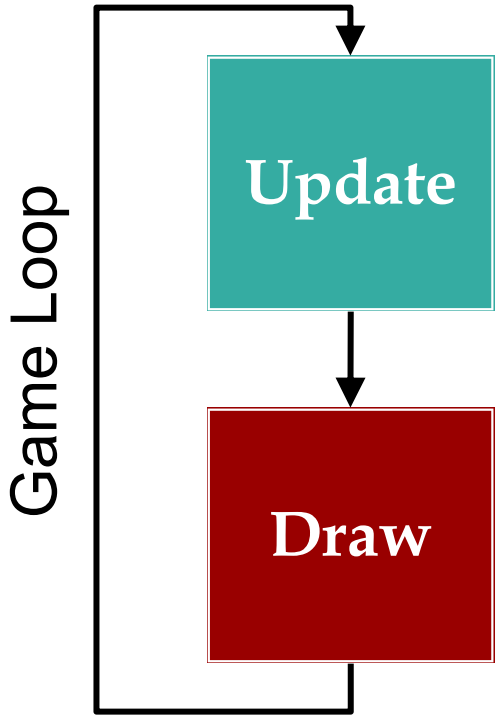
How do these fit together?

```
public class MyProcessor implements
      InputProcessor {

   public void keyTyped(char c) {
      //  Do something with input
   }

}
```

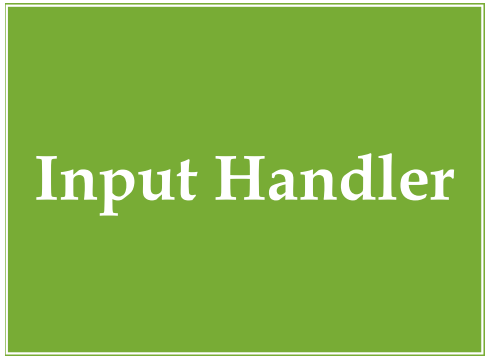
Unclear exactly when it is invoked

Classic Producer-Consumer Problem

Consumer



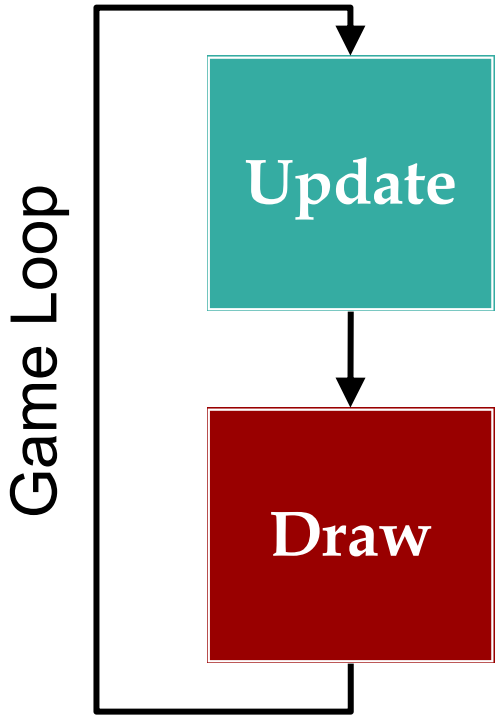
Producer



Classic Producer-Consumer Problem

Consumer

Producer



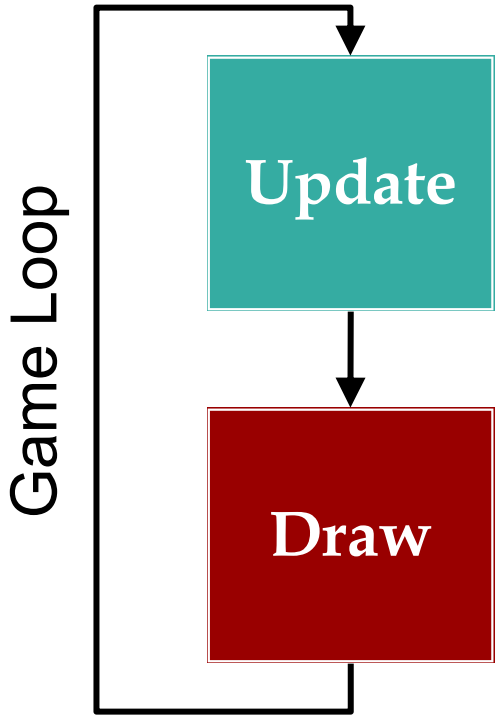
Buffer

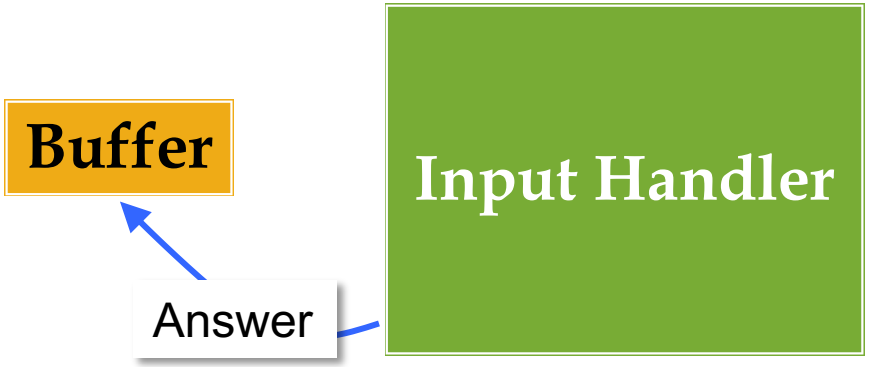
Input Handler

Classic Producer-Consumer Problem

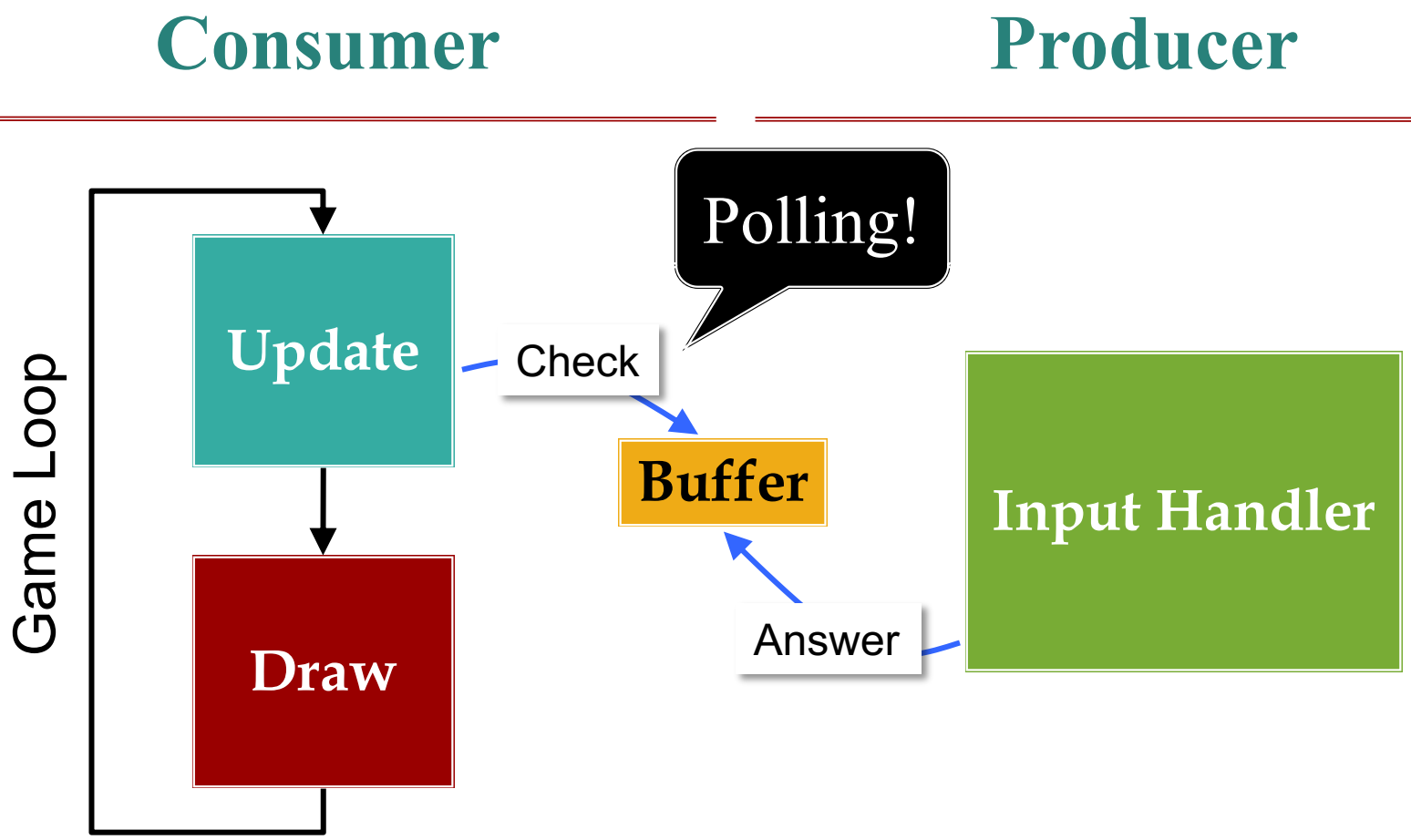
Consumer

Producer

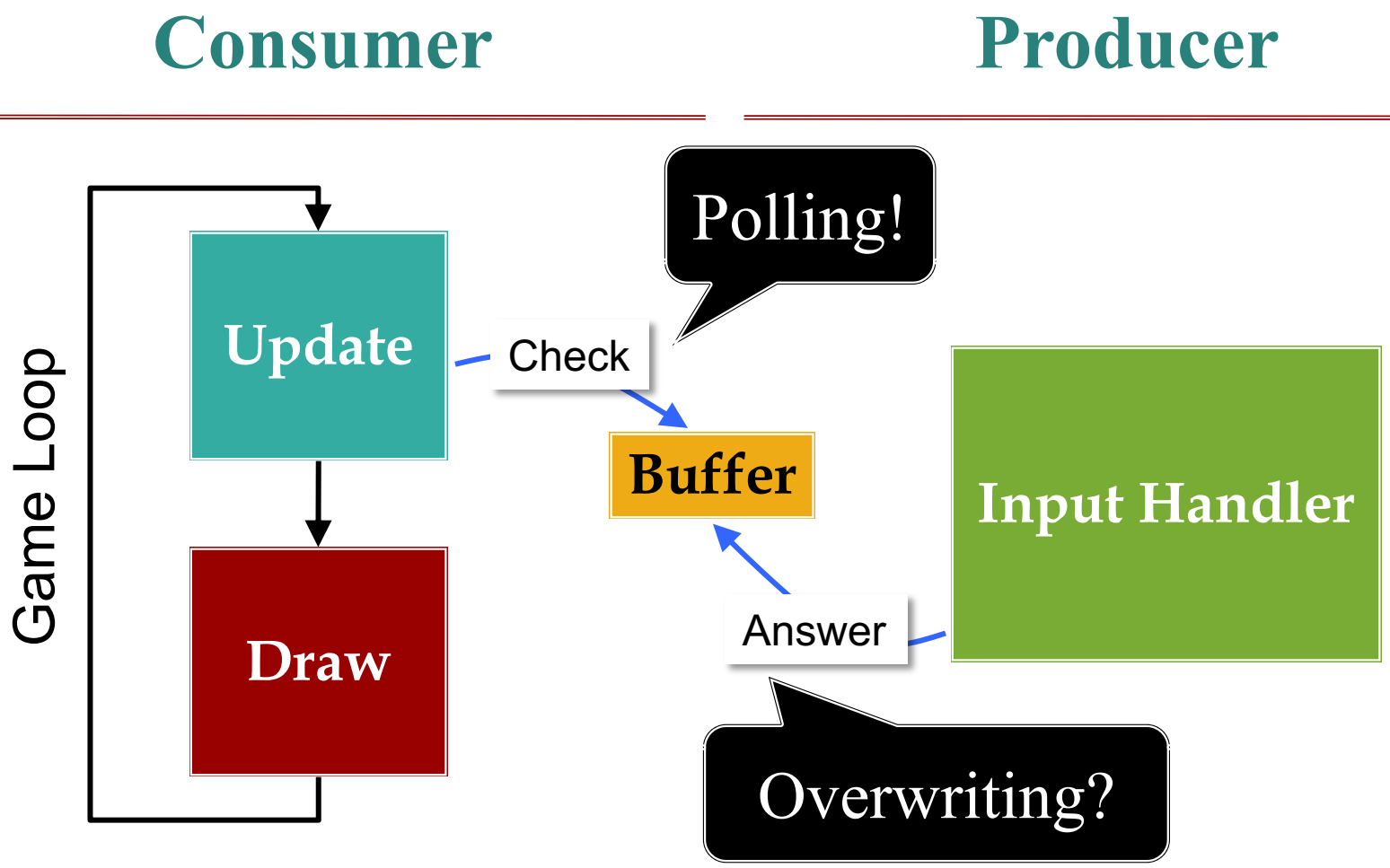




Classic Producer-Consumer Problem

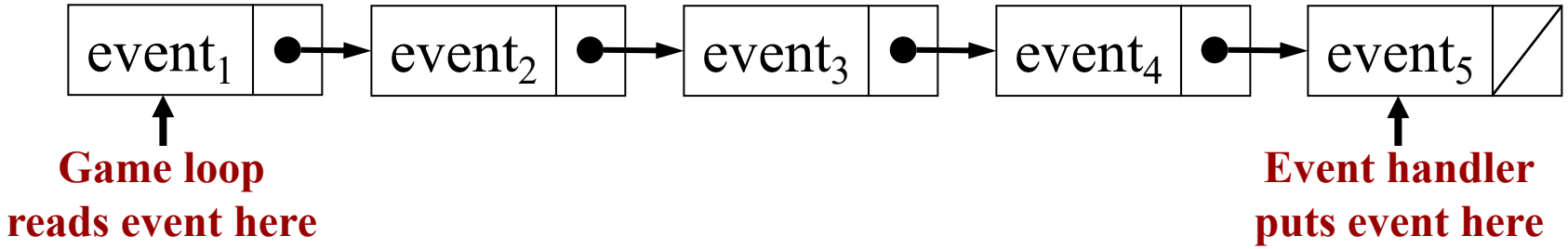


Classic Producer-Consumer Problem



Buffering Input

- If overwriting an issue, need an event queue
  - Input processor writes at end of the queue
  - Game loop reads from the front of queue



- Generally requires multiple threads
  - Event handler is (usually) OS/VM provided thread
  - Game loop itself is an additional thread

Event Handlers: Really Necessary?

- Most of the time: **No**
  - Frame rate is short: 16.7 ms
  - Most events are > 16.7 ms
  - Event loss not catastrophic

- Buffering is sometimes undesirable
  - Remembers every action ever done
  - But may take a longer time to process
  - If takes too long, just want to abort

Picking the Right Input

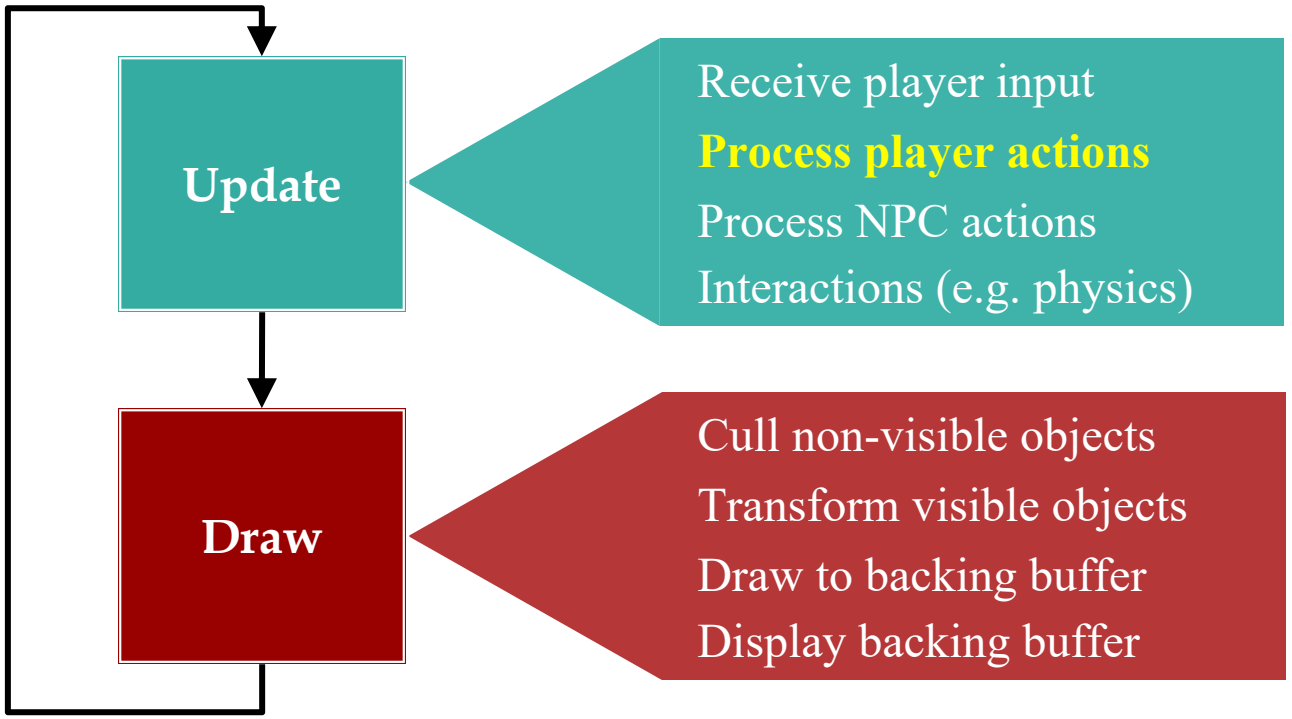
Polling

- When game loop is explicit
  - Actively animating screen
  - Must time input correctly
- **Example**: playing the game

Event Driven

- When game loop is implicit
  - Art assets are largely static
  - Nothing to do if no input
- **Example**: a menu screen

The Game Loop



Player Actions

- Actions alter the game state
  - Can alter player state: movement
  - Can alter opponent state: damage
- Player actions correspond to user input
  - Choice is determined by input controller
  - Else action is performed by computer
- These are your game **verbs**!

Abstract Actions from Input

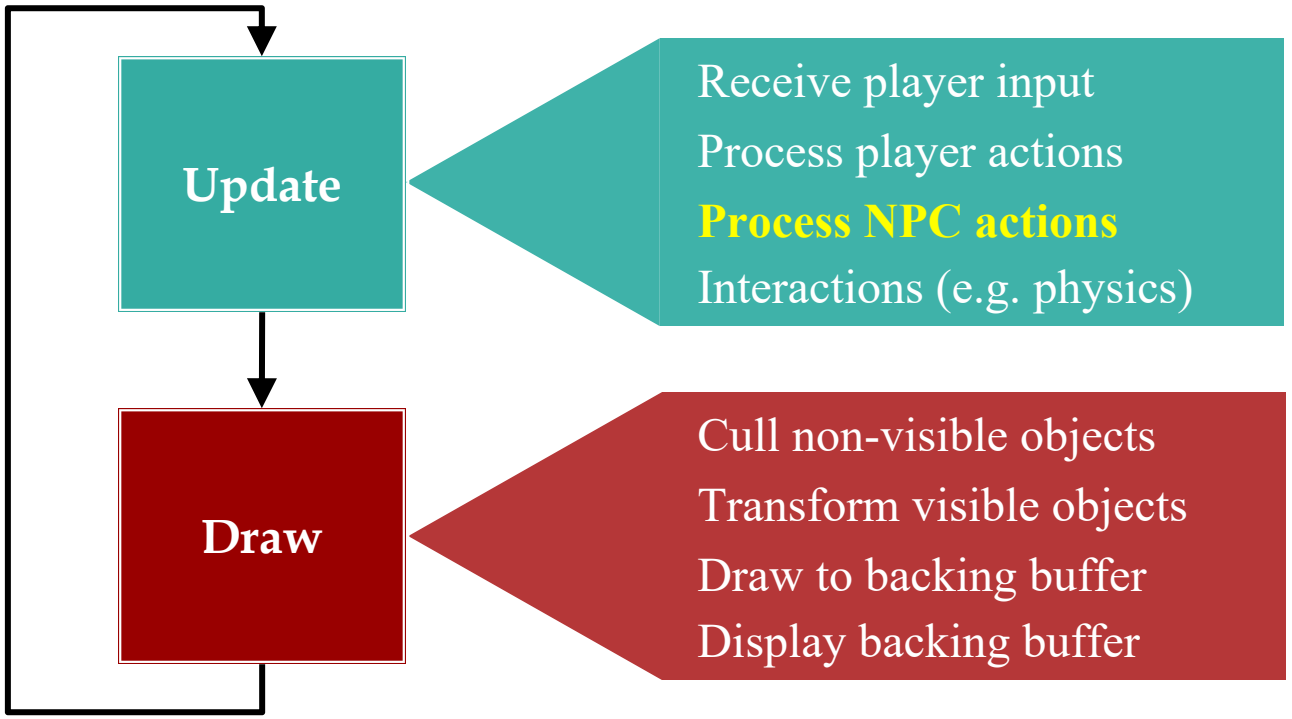
- **Actions**: functions that modify game state
  - `move(dx,dy)` modifies `x`, `y` by `dx`, `dy`
  - `attack(o)` attacks opponent `o`
- Input controller **maps** input to actions
  - Read input state from controller
  - Pick an action and call that function
- Input handler should never alter state directly!

Abstract Actions from Input

- **Actions**: functions that modify game state
  - `move(dx,dy)` modifies `x`, `y` by `dx`, `dy`
  - `attack(o)` attacks opponent `o`
- Input controller **maps** input to actions
  - Read input state from controller
  - Pick an action and call that function
- Input handler should never alter state directly!

Design versus **Implementation**

The Game Loop



NPC: Non-Player Character

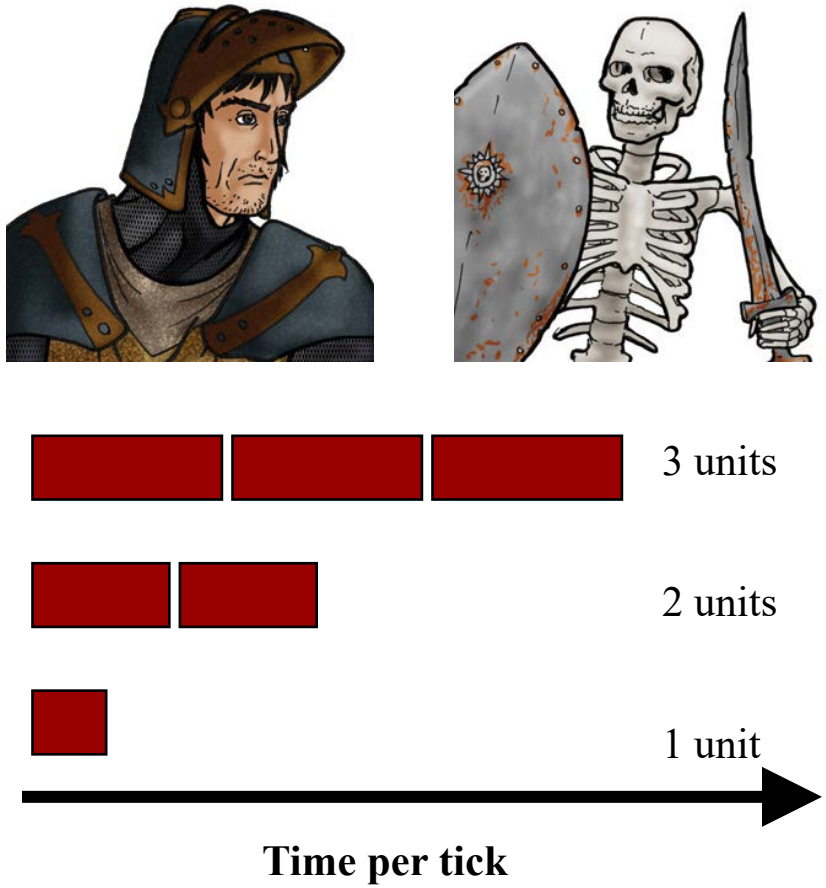
- NPC is an intelligent computer-controlled entity
  - Unlike a physics object, it can act, not just interact
  - Sometimes called an *agent*
- NPCs have their own actions/verbs
  - But no input controller to choose
- Work on sense-think-act cycle
  - **Sense**: perceive the world around it
  - **Think**: choose an action to perform
  - **Act**: update the game state

Act versus Sense-Think

- Act should be *very* fast!
  - Function to update state
  - **Example**: apply velocity
  - Exactly like the player
- Sense-think unique to NPC
  - The *hard* computation
  - Focus of AI lectures
- Multiplayer: Replace sense-think with human decision

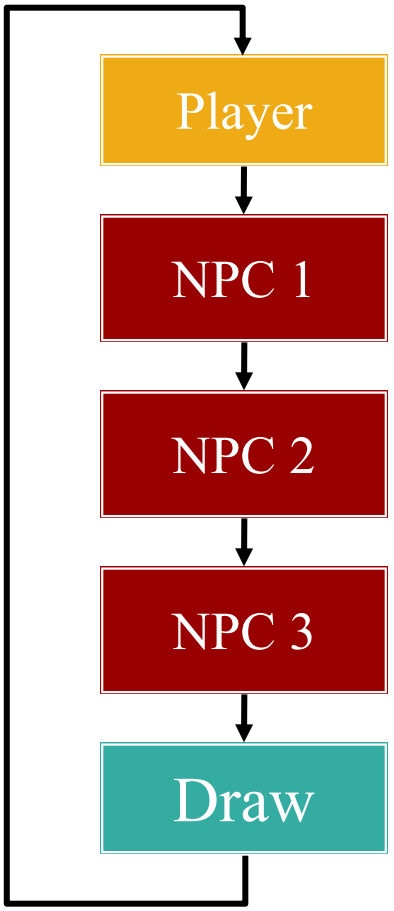
Problem with Sensing

- Sensing may be slow!
  - Consider *all* objects
- Example: morale
  - n knights, n skeletons
  - Knights fear skeletons
  - Proportional to # seen
- Count skeletons in view
  - O(n) to count skeletons
  - O(n^2) for all units



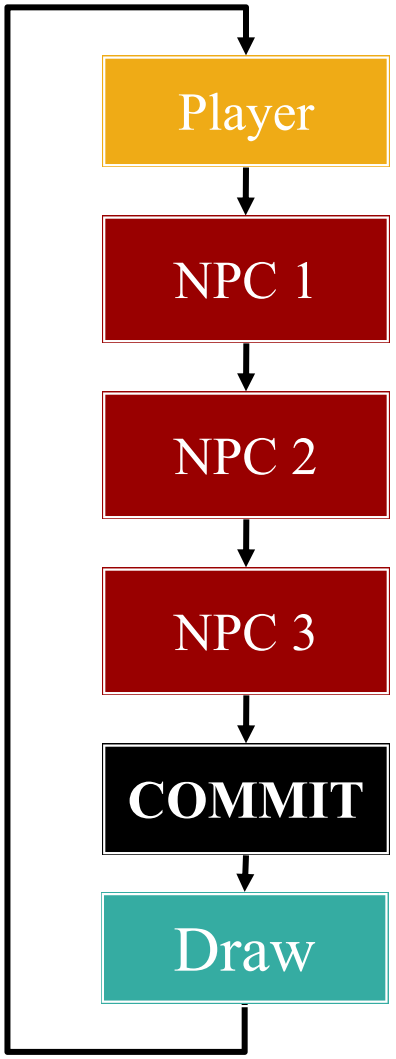
Processing NPCs

- Naïve solution: sequentially
- **Problem**: NPCs react too fast!
  - Each reads the actions of previous
  - Even before drawn on screen!

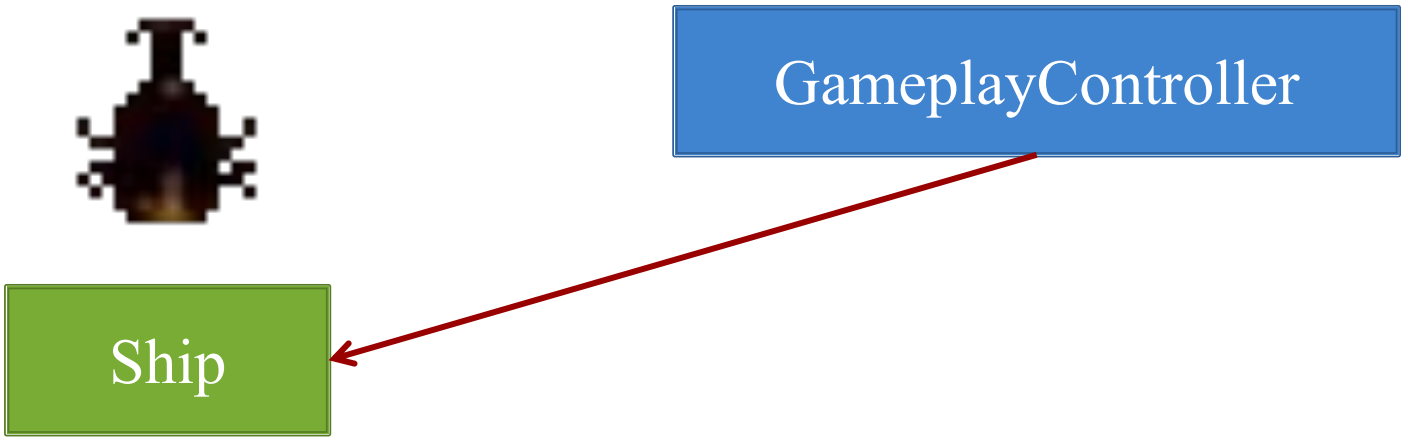


Processing NPCs

- Naïve solution: sequentially
- **Problem**: NPCs react too fast!
  - Each reads the actions of previous
  - Even before drawn on screen!
- **Idea**: only react to what can see
  - *Choose* actions, but don't perform
  - Once all chosen, then perform
  - Another reason to abstract actions



Processing Actions in Lab 3



- Decides whether to shoot
- Stores intent in the object
- But **DOES NOT** shoot

- Waits until objects commit
- Checks intent in Ship object
- Performs action for intent

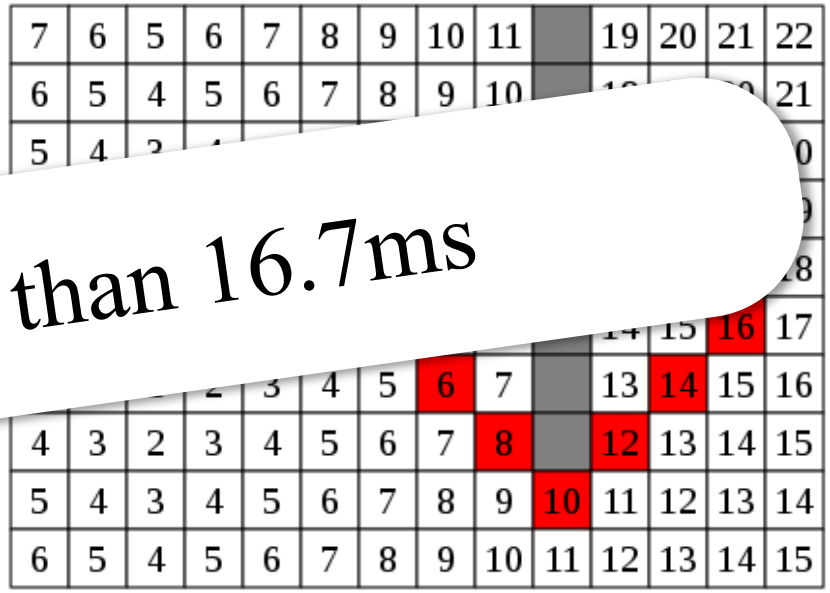
Problem: Pathfinding

- Focus of Game Lab 2
  - Crucial if top view
  - Major area of research
- Potentially very slow
  - *n* NPCs, *g* grid squares
  - Dijkstra: $O(g^2)$
  - For each NPC: $O(ng^2)$
- **Moving obstacles?**

7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18	19	20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

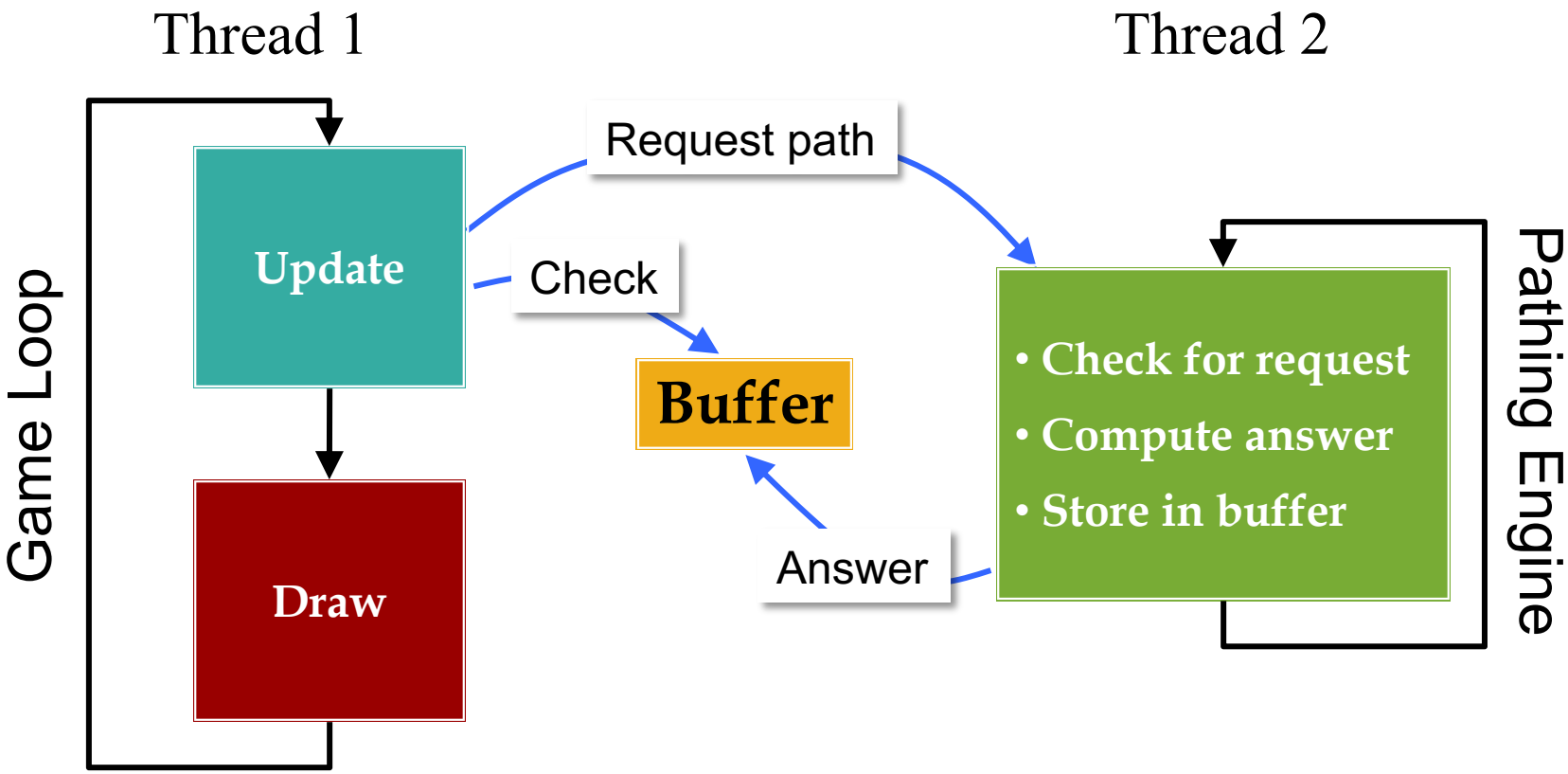
Problem: Pathfinding

- Focus of Game Lab 2
  - Crucial if top view
  - Major area of research
- Potentially very l
  -
  -
  - For each NPC: O(ng^2)
- **Moving obstacles?**



Often more than 16.7ms

Asynchronous Pathfinding

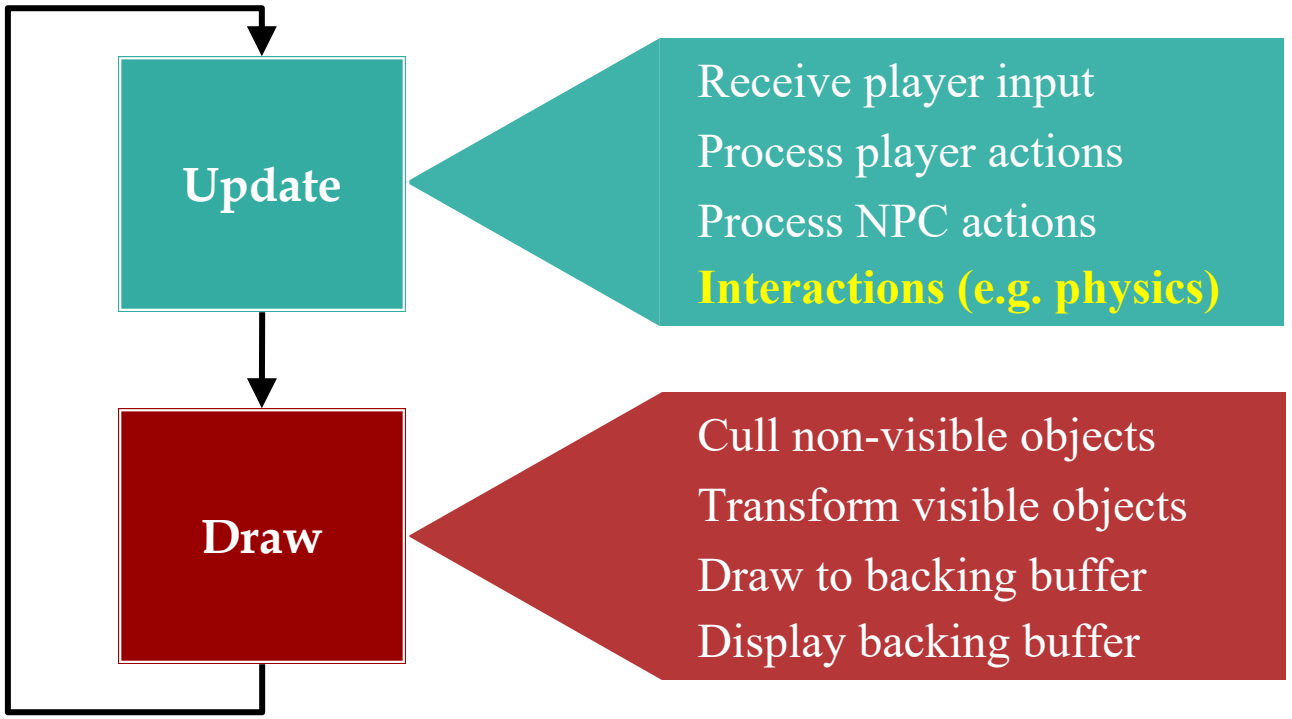


Looks like input buffering!

Asynchronous Pathfinding

- NPCs do not get answer right away
  - Check every loop until answered
  - Remember request; do not ask again
- What to do until then?
  - Act, but don't think!
  - If nothing, **fake** something
  - "Stomping Feet" in RTSs

The Game Loop



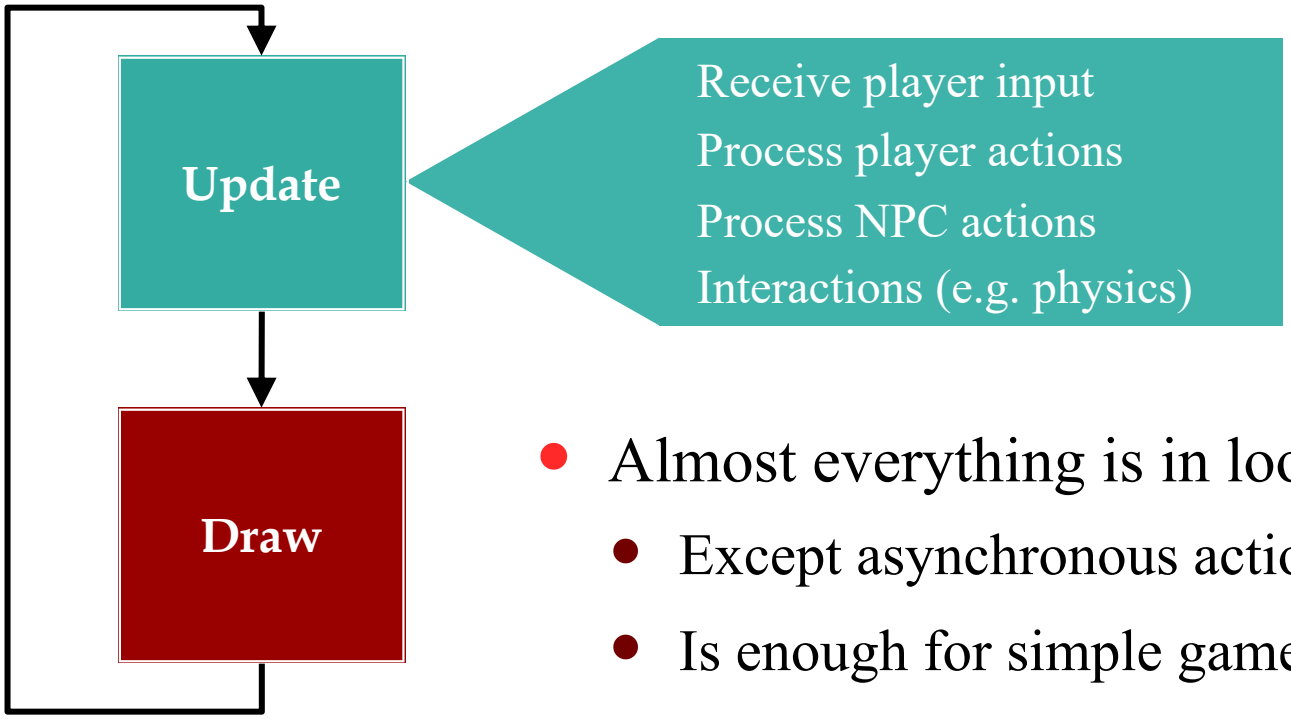
Purpose of a Physics Engine

- Moving objects about the screen
  - **Kinematics**: Without regard to external forces
  - **Dynamics**: The effect of forces on the screen
- Collisions between objects
  - **Collision detection**: Did a collision occur?
  - **Collision resolution**: What do we do?
- More on this issue later (~Spring Break)

Physics Engines: Two Levels

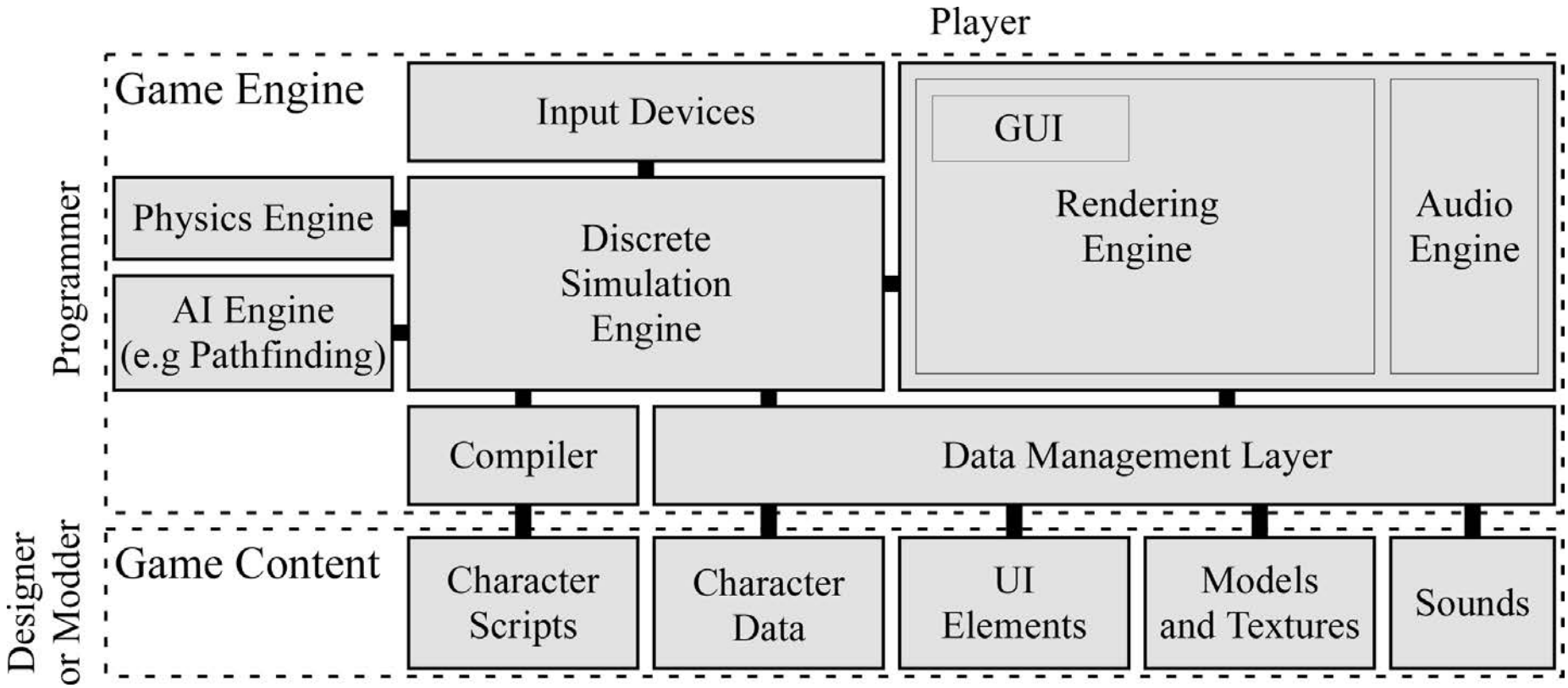
- **White Box**: Engine corrects movement errors
  - Update object state ignoring physics
  - Physics engine nudges object until okay
- **Black Box**: Engine handles everything
  - Do not move objects or update state
  - Give forces, mass, velocities, etc. to engine
  - Engine updates to state that is *close enough*

The Game Loop

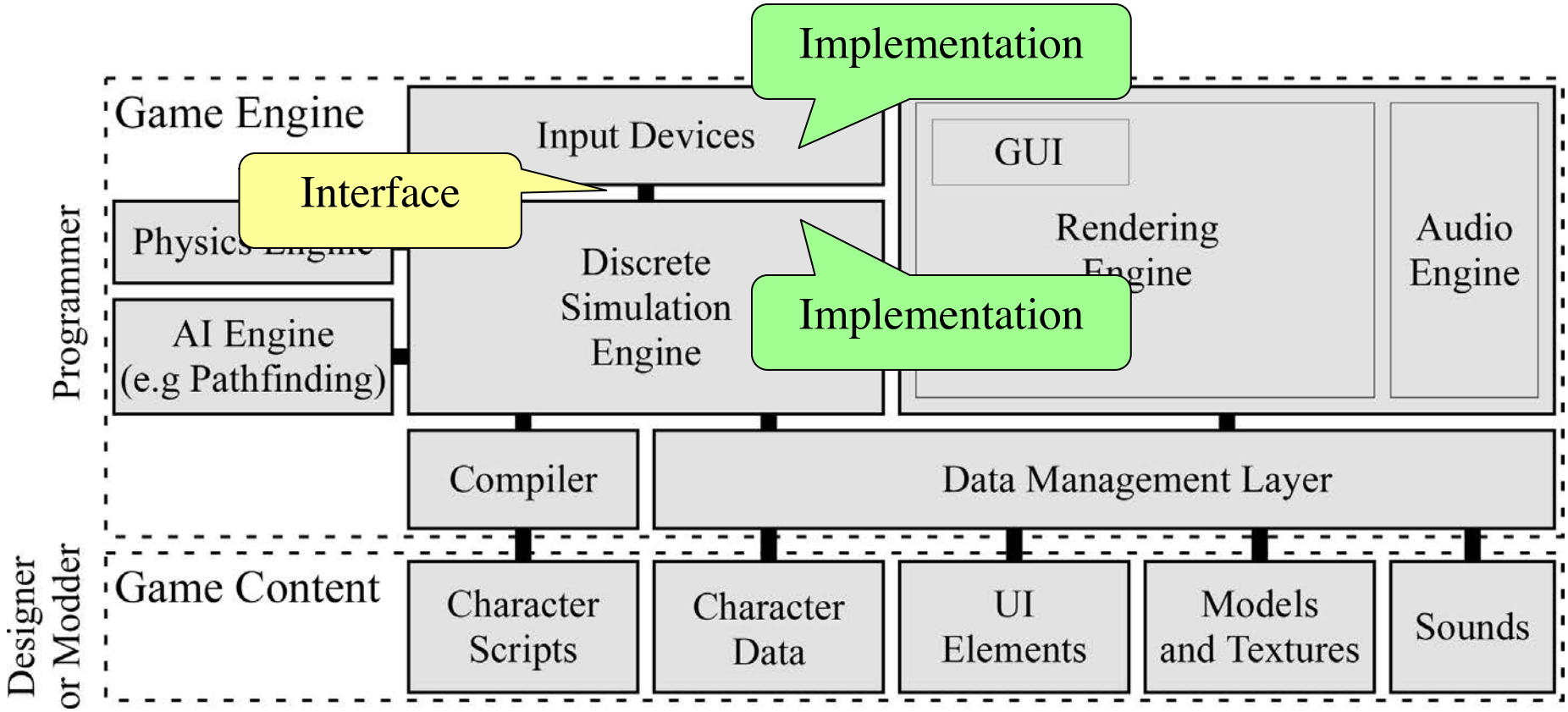


- Almost everything is in loop
  - Except asynchronous actions
  - Is enough for simple games
- How do we organize this loop?
  - Do not want spaghetti code
  - Distribute over programmers

Architecture: Organizing Your Code



Architecture: Organizing Your Code



Where Did This Come From?