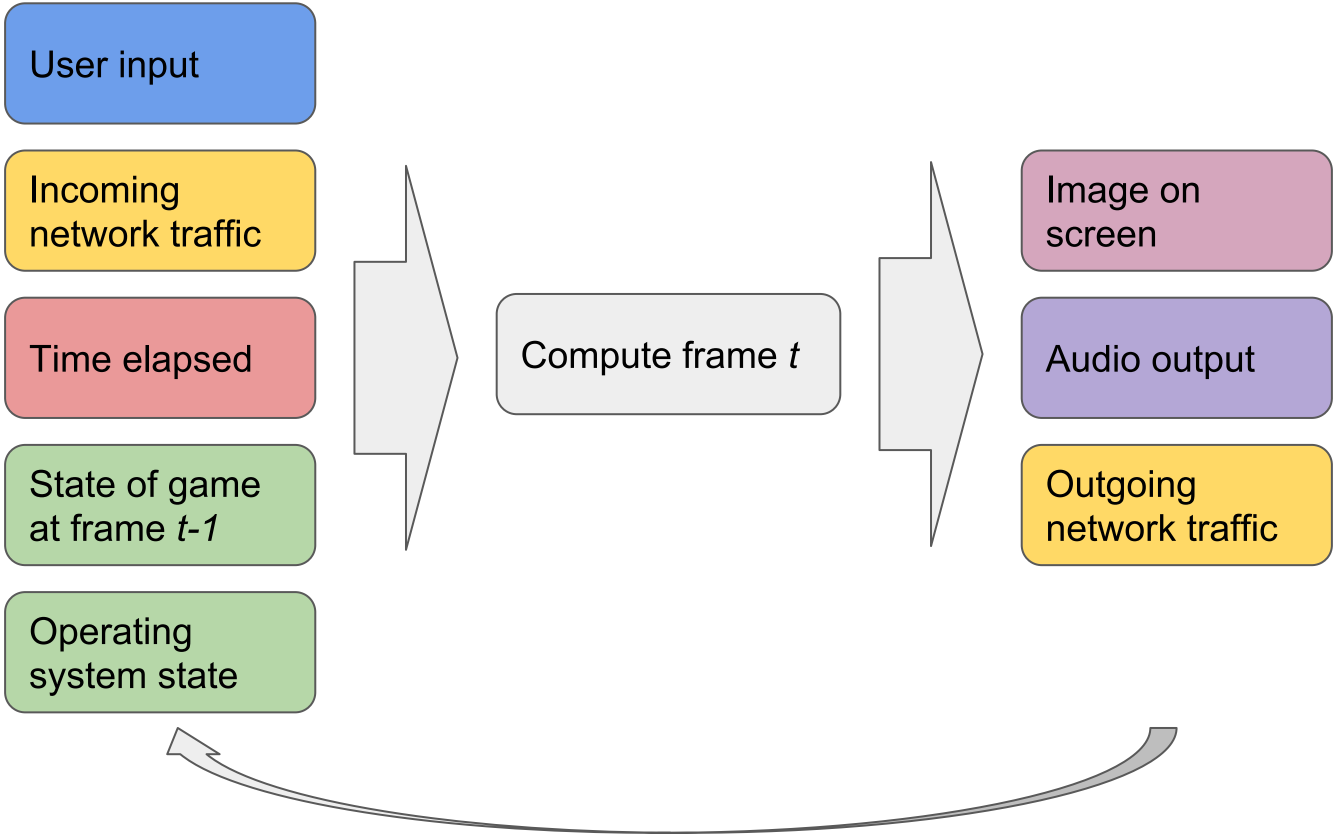
Game Architecture

Main Loop

Today’s Agenda

- What’s this main loop thing?
- A simple main loop
- Handling time
- Handling ‘modes’
- Multicore main loop
- Job systems
- First homework assignment

What’s this main loop thing?



User input

Incoming network traffic

Time elapsed

State of game at frame *t-1*

Operating system state

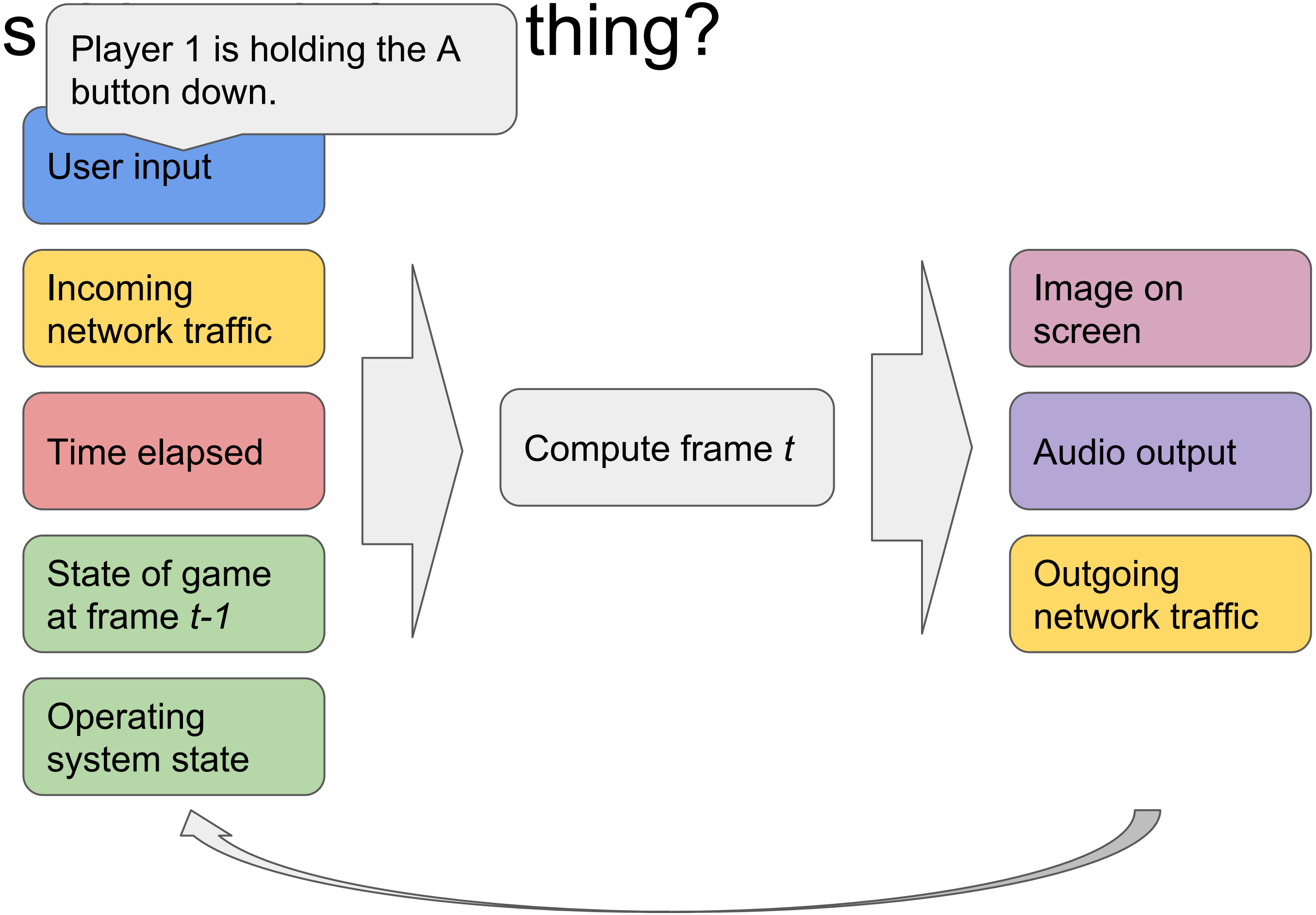
Compute frame *t*

Image on screen

Audio output

Outgoing network traffic

What’s thing?



Player 1 is holding the A button down.

User input

Incoming network traffic

Time elapsed

State of game at frame *t-1*

Operating system state

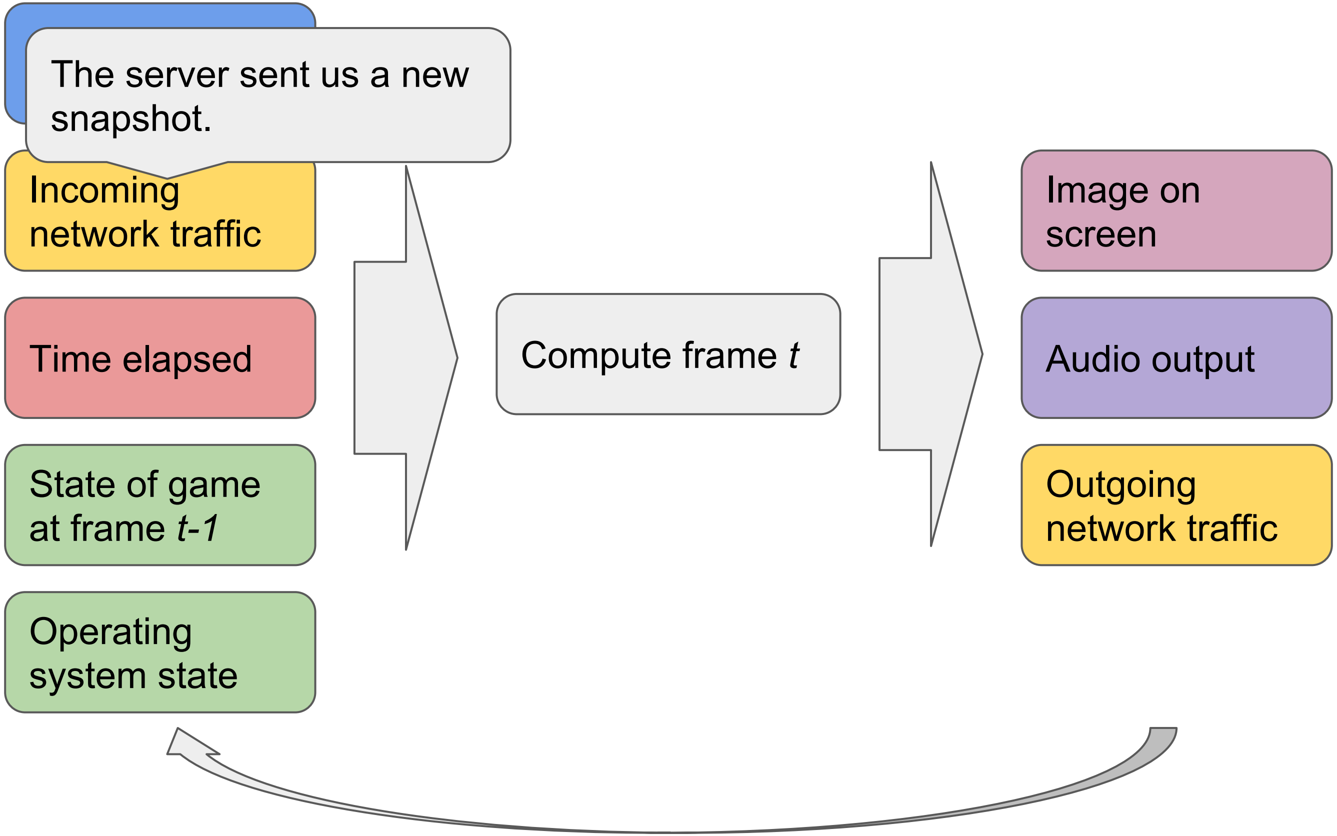
Compute frame *t*

Image on screen

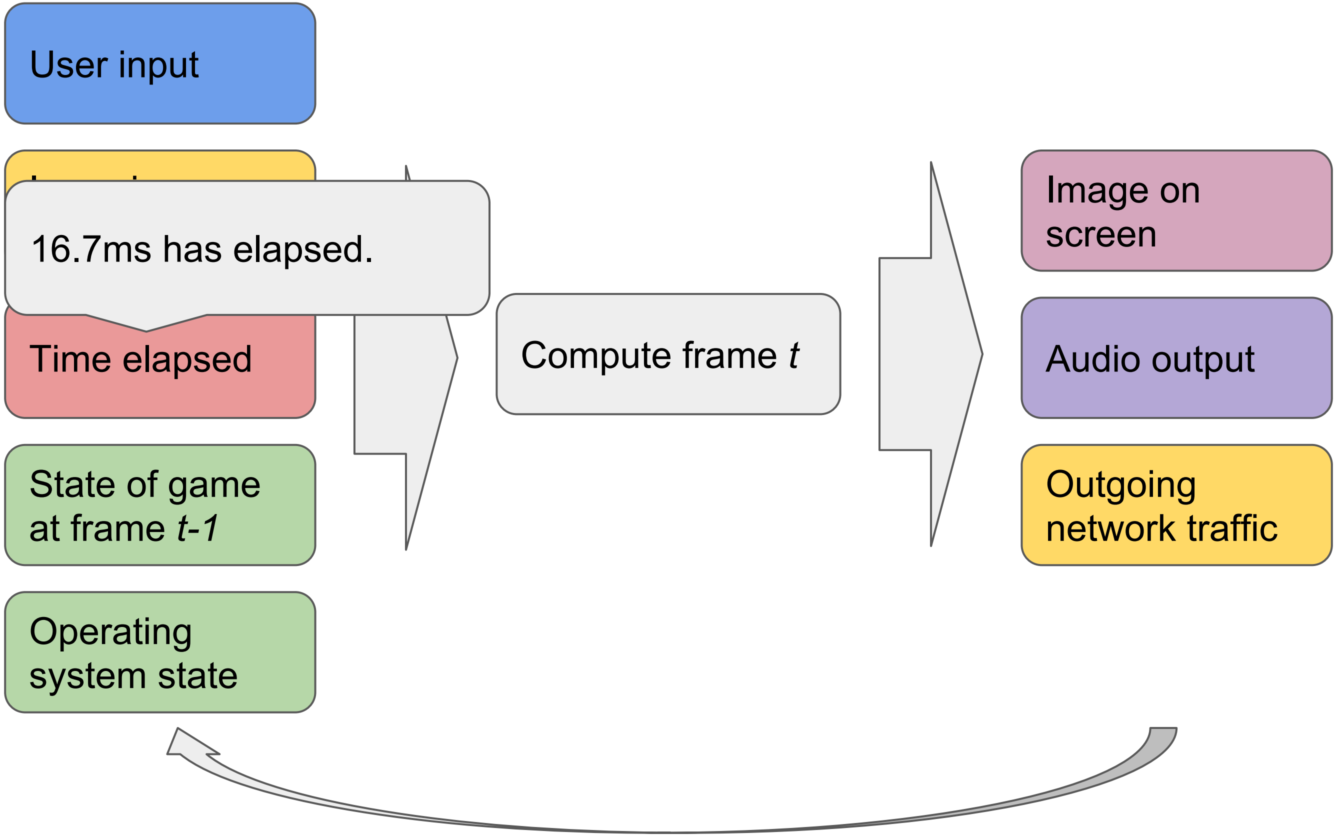
Audio output

Outgoing network traffic

What’s this main loop thing?



What’s this main loop thing?



User input

16.7ms has elapsed.

Time elapsed

State of game at frame *t-1*

Operating system state

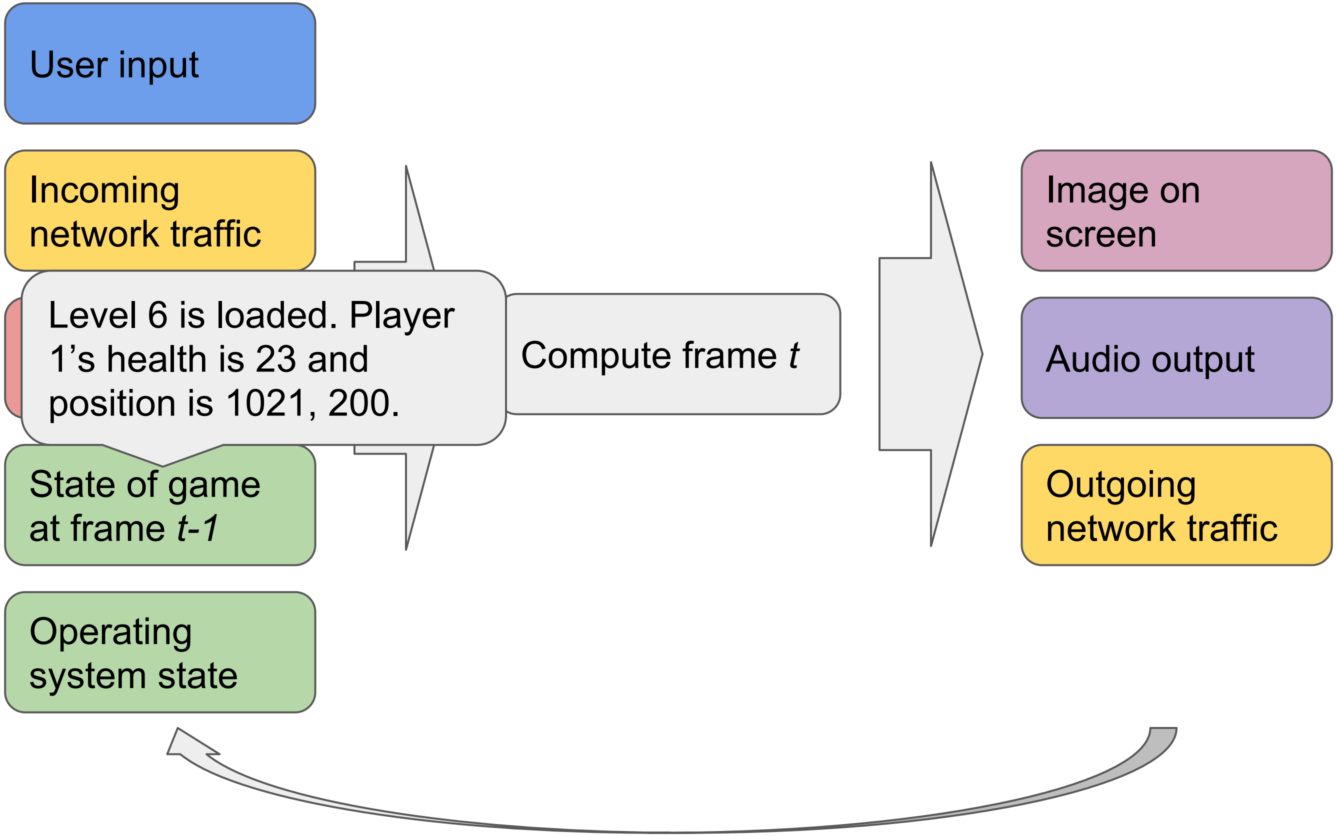
Compute frame *t*

Image on screen

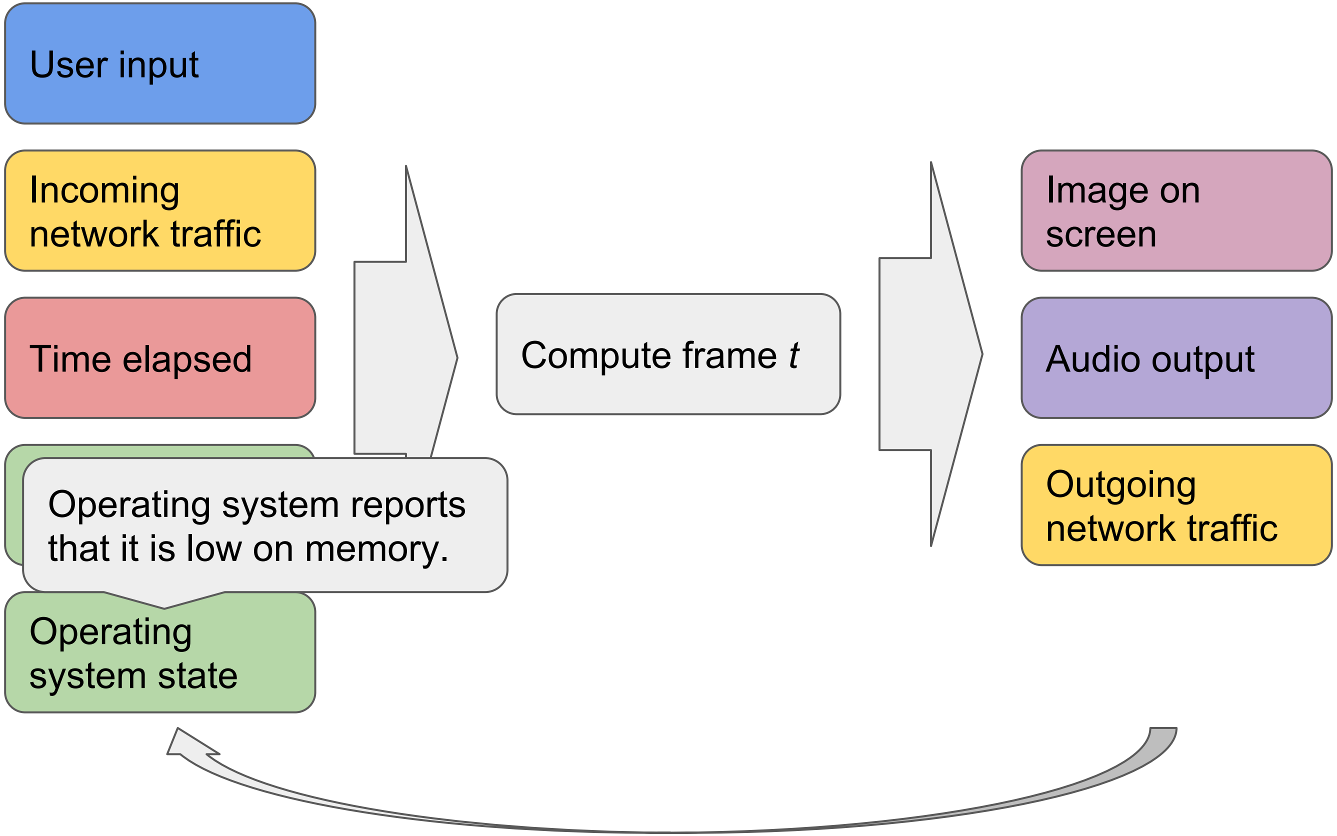
Audio output

Outgoing network traffic

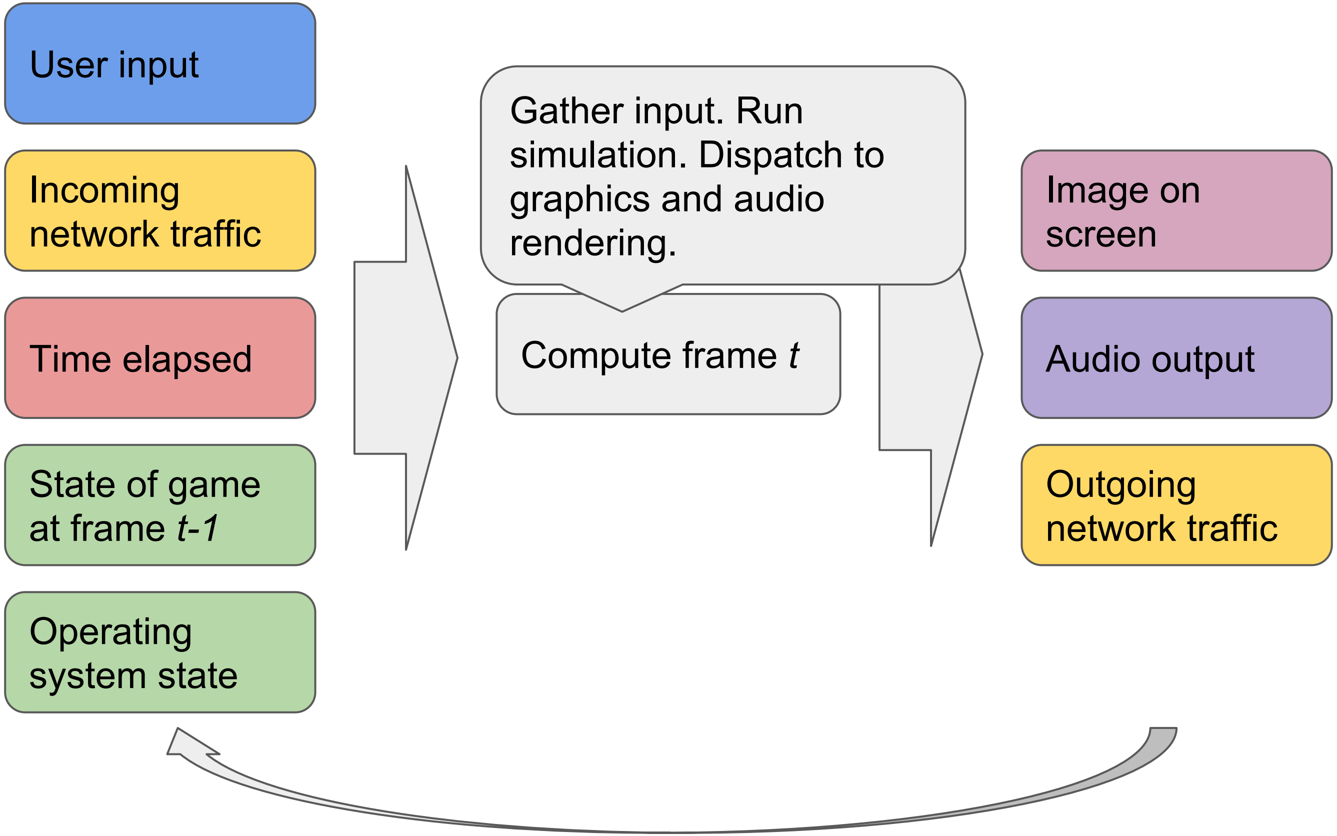
What’s this main loop thing?



What’s this main loop thing?



What’s this main loop thing?



User input

Incoming network traffic

Time elapsed

State of game at frame *t-1*

Operating system state

Gather input. Run simulation. Dispatch to graphics and audio rendering.

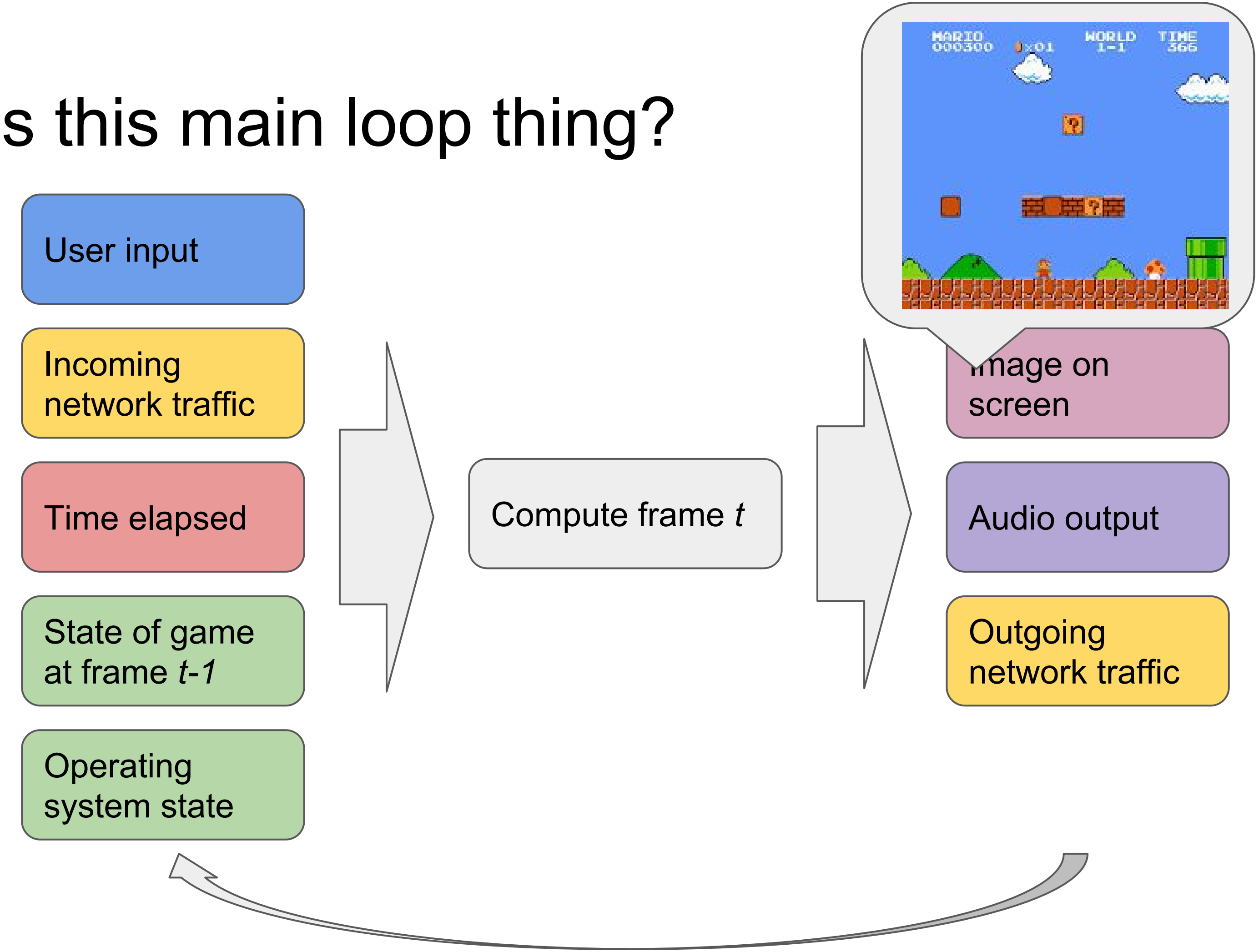
Compute frame *t*

Image on screen

Audio output

Outgoing network traffic

What’s this main loop thing?



User input

Incoming network traffic

Time elapsed

State of game at frame *t-1*

Operating system state

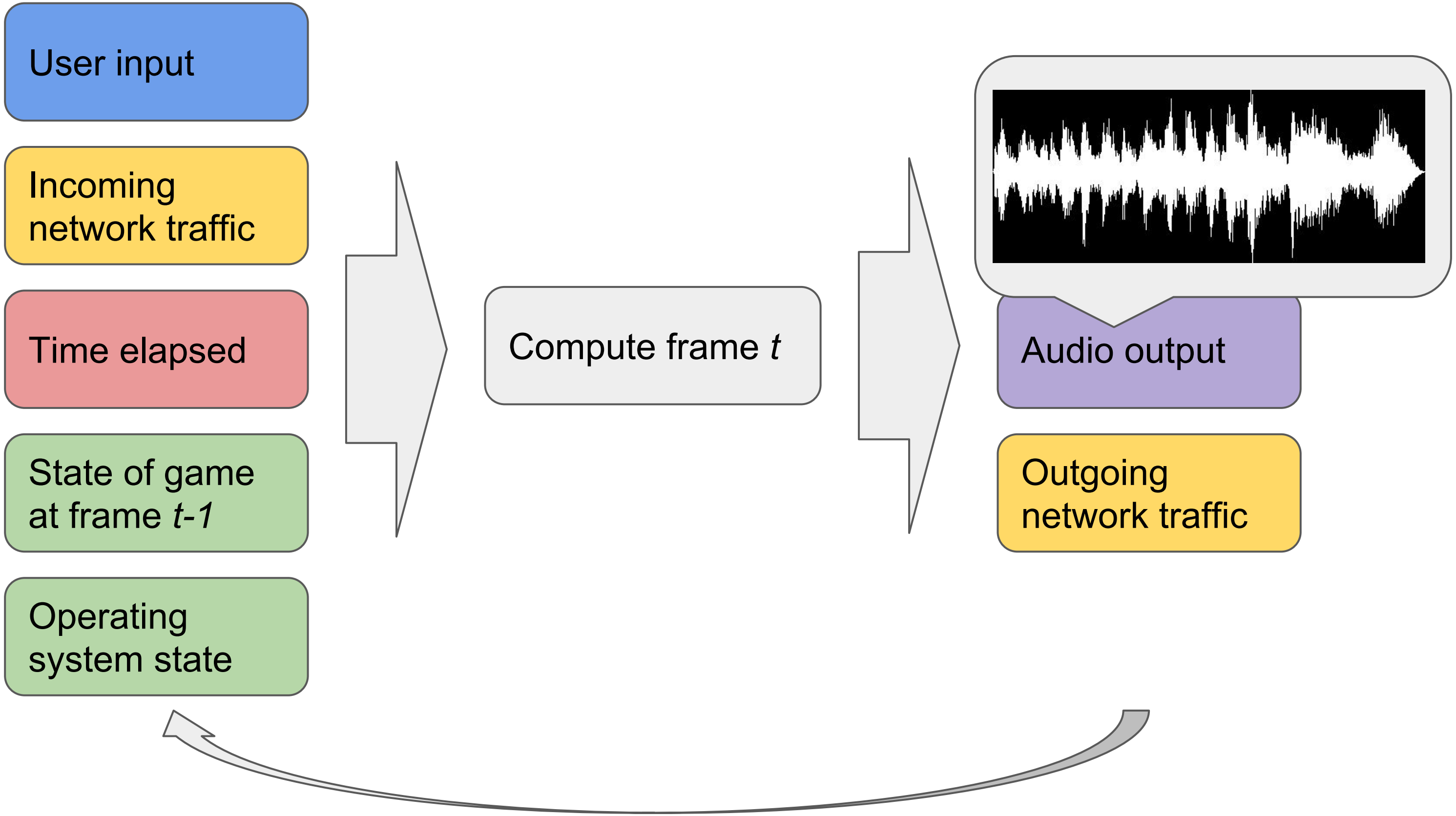
Compute frame *t*

Image on screen

Audio output

Outgoing network traffic

What’s this main loop thing?



User input

Incoming network traffic

Time elapsed

State of game at frame *t-1*

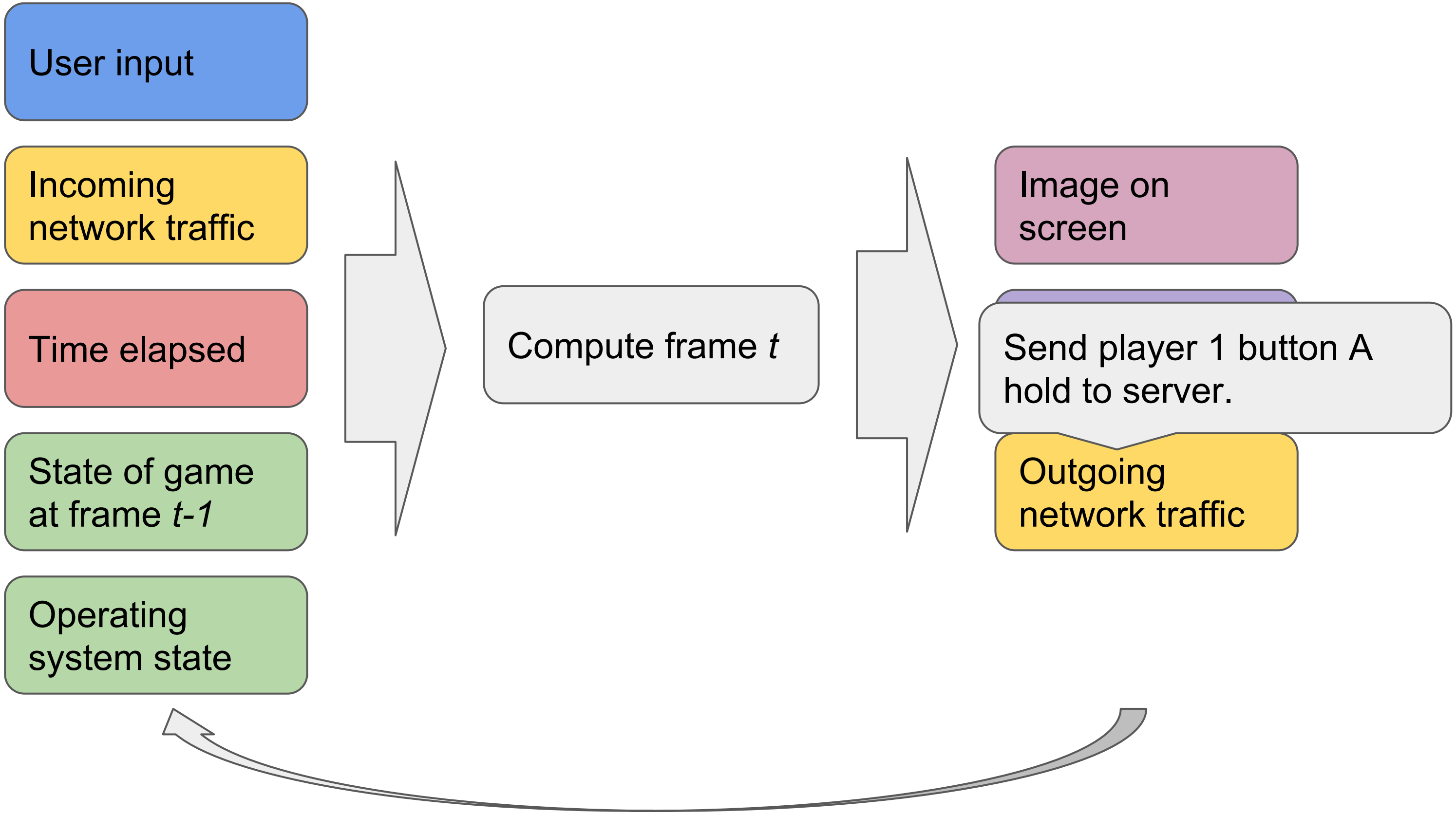
Operating system state

Compute frame *t*

Audio output

Outgoing network traffic

What’s this main loop thing?



User input

Incoming network traffic

Time elapsed

State of game at frame *t-1*

Operating system state

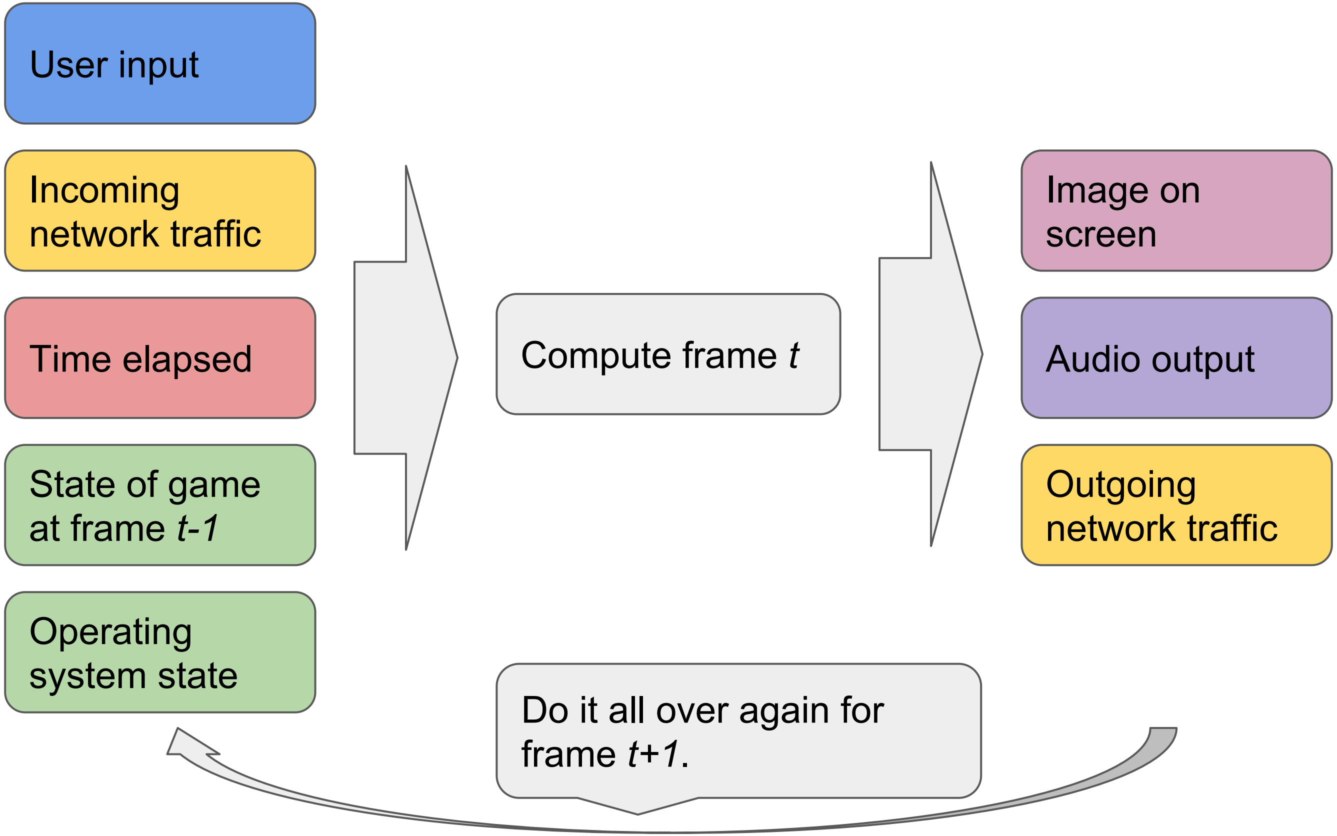
Compute frame *t*

Image on screen

Send player 1 button A hold to server.

Outgoing network traffic

What’s this main loop thing?



User input

Incoming network traffic

Time elapsed

State of game at frame *t-1*

Operating system state

Compute frame *t*

Image on screen

Audio output

Outgoing network traffic

Do it all over again for frame *t+1*.

Simple main loop

```
while (true) {
    time_update();
    input_update();

    if (os_handle_events() == k_exit) break;

    sim_update();

    graphics_update();
    audio_update();
}
```

Time

- Importance and relevance
- Common terms
- Measuring time
- Time transformed
- Game systems using time
- Delta time considerations

Time

- **Importance and relevance**
- Common terms
- Measuring time
- Time transformed
- Game systems using time
- Delta time considerations

For many types of games, we aim to:

- Minimize lag between user input and action on screen
- Maximize sampling rate so the user can better perceive game environment (smoothness)

Proper treatment of time is critical to achieving these goals.

Time

- Importance and relevance
- **Common terms**
- Measuring time
- Time transformed
- Game systems using time
- Delta time considerations

- **Frame rate** measured in **Hertz**.
- **Frame time**, or **delta time**, measured in **milliseconds**.
- **V-sync**, or synchronizing game update with monitor refresh rate*.
- **Frame rate spike**, or sudden change in frame rate resulting in poor game feel.
- **Clock wrap**, or the current time overflowing its storage and cycling back to zero.
- **Clock drift**, or a timer slowly diverging from the true time (faster or slower).

Time

- Importance and relevance
- Common terms
- **Measuring time**
- Time transformed
- Game systems using time
- Delta time considerations

- Often:
  - High resolution timer that ticks once per CPU cycle (e.g. 3 billion times per second on a 3Ghz CPU).
  - Store absolute high resolution time 64-bit integers.
  - Store small durations in 32-bit floats.
- Be aware of precision limitations.
- Watch for drift between processors on multicore systems.

Time

- Importance and relevance
- Common terms
- Measuring time
- **Time transformed**
- Game systems using time
- Delta time considerations

Hierarchy of timers:

- Wall clock time
- Application time
- Gameplay time
- Animation time

Use bias and scale to transform between timers. Some lower level timers can be artificially scaled, paused, etc as needed by game design.

Time

- Importance and relevance
- Common terms
- Measuring time
- Time transformed
- **Game systems using time**
- Delta time considerations

Don’t have to update all systems every frame:

- Graphics: once per frame
- AI: 5Hz?
- Physics: 120Hz?

For systems that update, need to know how much time to simulate. Often just use delta time since for last frame.

This isn’t without its issues...

Time

- Importance and relevance
- Common terms
- Measuring time
- Time transformed
- Game systems using time
- **Delta time considerations**

1. Last frame’s delta time is just a guess of this frame’s actual delta time.
2. Systems that require high frequency update can be difficult. The slower the frame, the more updates they require, making the frame still slower…
3. Mobile platforms have a battery. Running at full speed burns battery faster.

Consider:

- Clamping delta time to acceptable range
- Using floating mean to estimate delta time, or even fixed time step

Modes

Games have a lot of modes of operation. Some introduced by us:

- Startup sequence
- Frontend menu
- Loading a level
- Playing level
- At pause menu…

Some forced upon us by operating system:

- Running in foreground or background
- Running with full system resources, running in constrained way

Modes 2

- Certain game systems only run in certain modes
- Structure of main loop varies mode-to-mode

A good main loop cleanly describes mode specific behavior and transitions.

Modes: One Approach

Stack based state machine:

- Frame computation, `sim_update()`, is actually a FSM with stack
- States can be activated (pushed) and deactivated (popped)
- State at the top of stack is executed each frame

```
void sim_update()
{
        switch (top_state)
        {
        case k_state_menu: do_menu(); break;
        case k_state_play: do_gameplay();
break;
        case k_state_load: do_load(); break;
        case k_input_unplugged: do_unplugged();
        }
}

void do_gameplay()
{
        if (is_input_unplugged())
        {
                state_push(k_input_unplugged);
                return;
        }
        // ...
}
```

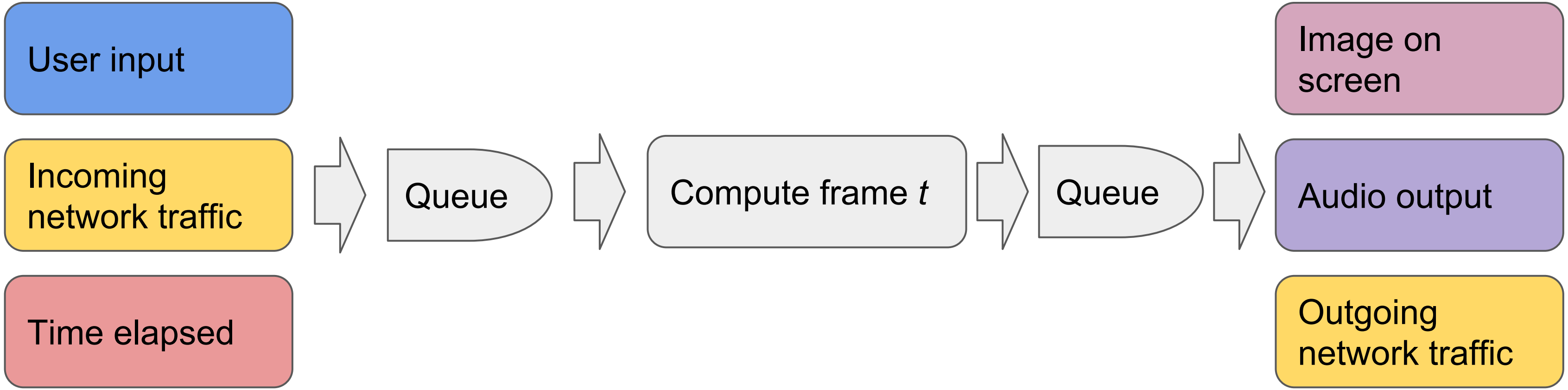
Multicore

- Most games run on multicore CPUs
- Common core counts today range from 2 to 8+
- We probably want to use those cores
- We have a few options:
    - Preserve the high level sequential nature of the main loop, go wide within sim and output
    - Go fully asynchronous and process all phases of main loop in parallel
    - A hybrid approach

Go Wide

- What it looks like:
  - Inside sim phase:
    - Run game logic in parallel
    - Solve physics in parallel
  - Inside output phase:
    - Build command buffers in parallel
    - Evaluate audio in parallel
- Advantage:
  - Isolates multicore usage -- less code has to be thread safe
  - Possibly easier to comprehend
- Disadvantage:
  - Somewhat limited ability to scale
  - Bubbles in computation in a single task/job define the critical path

Go Fully Asynchronous



User input

Incoming network traffic

Time elapsed

Queue

Compute frame *t*

Queue

Image on screen

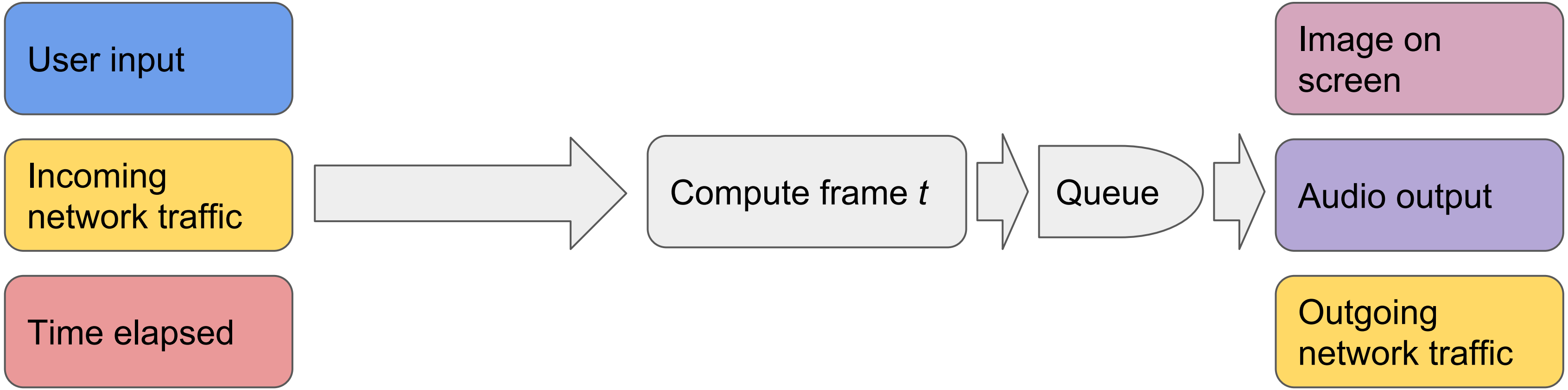
Audio output

Outgoing network traffic

Go Fully Asynchronous

- What it looks like:
  - Queues between phases
    - Input frame t, sim frame t-1, output frame t-2
  - Wide computation within phases
- Advantage:
  - Scales up and down reasonably well
- Disadvantage:
  - All code is potentially running in parallel
- For example: Destiny

Hybrid Approach



User input

Incoming network traffic

Time elapsed

Compute frame *t*

Queue

Image on screen

Audio output

Outgoing network traffic

Hybrid Approach

- What it looks like:
  - Queues between phases sim and output
  - Ad-hoc wide computation within phases
- Middle ground between wide only, and fully async
- For example: Skylanders SuperChargers

Doing the Work

Assuming we want to do something multicore, how do we implement it?

Doing the Work

Assuming we want to do something multicore, how do we implement it?

- Spin up some purpose built threads
  - Works great for I/O bound work, but doesn’t scale to arbitrary core counts

Doing the Work

Assuming we want to do something multicore, how do we implement it?

- Spin up some purpose built threads
    - Works great for I/O bound work, but doesn’t scale to arbitrary core counts
- Divide work into units (jobs, tasks) and schedule across available cores
    - Job queue
        - Dispatch work FIFO order

Doing the Work

Assuming we want to do something multicore, how do we implement it?

- Spin up some purpose built threads
  - Works great for I/O bound work, but doesn’t scale to arbitrary core counts
- Divide work into units (jobs, tasks) and schedule across available cores
  - Job queue
    - Dispatch work FIFO order
  - Task graph
    - Account for dependencies between work in a DAG of work
    - Schedule only when dependencies have been fulfilled

Doing the Work

Assuming we want to do something multicore, how do we implement it?

- Spin up some purpose built threads
  - Works great for I/O bound work, but doesn’t scale to arbitrary core counts
- Divide work into units (jobs, tasks) and schedule across available cores
  - Job queue
    - Dispatch work FIFO order
  - Task graph
    - Account for dependencies between work in a DAG of work
    - Schedule only when dependencies have been fulfilled
  - Job system
    - Dispatch work FIFO order, but allow jobs to wait on other jobs
    - Dependencies are implicit in control flow

Doing the Work

Assuming we want to do something multicore, how do we implement it?

- Spin up some purpose built threads
    - Works great for I/O bound work, but doesn’t scale to arbitrary core counts
- Divide work into units (jobs, tasks) and schedule across available cores
    - Job queue
        - Dispatch work FIFO order
    - Task graph
        - Account for dependencies between work in a DAG of work
        - Schedule only when dependencies have been fulfilled
    - **Job system** ☺
        - **Dispatch work FIFO order, but allow jobs to wait on other jobs**
        - **Dependencies are implicit in control flow**

Job System

- "Parallelizing the Naughty Dog Engine Using Fibers", Christian Gyrling
- Components
  - One thread per core
  - Fiber pool sufficient for one fiber per queued/active job
  - Job queue
  - Waiting list
- Threads are the compute resource, fibers are the context
- User adds work to shared job queue
- User waits for job by pushing self to wait list, switching to scheduler fiber
- Scheduler:
  - Pops ready work off wait list and switches to its fiber
  - Pops work off job queue, assigns fiber, and switches to that fiber

Job System

```
void child_job_func(void* data() { /* ... */ }

void parent_job_func()
{
    job_decl decls[100];
    decls[...].function = child_job_func;
    decls[...].data = child_job_data;

    int counter;
    job_run(decls, 100, &counter);
    job_wait(&counter);
}
```

First homework

- Sequential main loop with wide sim phase
- Use job system to simulate all game objects in parallel
  - For homework, you will write a key component, the thread-safe queue that holds work
- Simple component-based entity system
  - Discussed more next class
- Let’s walk through the code together...

Summary

- Structure of the main loop
- Time
- Modes
- Multicore options and a job system

End Lecture

http://gdcvault.com/play/1022186/Parallelizing-the-Naughty-Dog-Engine