Assignment 6 Recitation

Concurrency

The Threading Model

- Multiple lines of execution, one set of shared memory
- As opposed to a process
- To handle this without errors, we require...

Concurrency Primitives

- Focus on three types:
  - Threads
  - Mutexes
  - Semaphores
- Already covered high level in lecture
- Now built into C++11
- Makes your life easier! (as compared to POSIX or other libraries)

Threads

- With C++11, threads can run any function
- The function name is the first input to the thread constructor
- The function arguments follow, in the same order in the constructor as in the function

Threads

- An example:

```
void thread_adder(int a, int b) {
	std::cout << a << std::endl;
	std::cout << b << std::endl;
	std::cout << a +  b << std::endl;
}

int a = 1;
int b = 2;
std::thread t(thread_adder, a, b);
t.join()
```

Threads

- A side note: joining
  - Blocks execution on the current thread until the joined thread has finished
  - As opposed to detaching, where both threads are free to run simultaneously

Mutexes

- Use when only one thread should use a resource at a time

```
std::mutex* m = new std::mutex();      // Create a mutex.
m->lock();                             // Lock mutex.
m->unlock();                           // Unlock mutex.
```

Mutex Example

```
void a(std::mutex* m) {
        m->lock();
        std::cout << "In a!" << std::endl;
}

void b(std::mutex* m) {
        m->lock();
        std::cout << "In b!" << std::endl;
        m->unlock();
}
```

```
//Lock shared by a, b
std::mutex m;
a(&m);
b(&m);
```

Deadlock!

```
void a(std::mutex* m) {
        m->lock();
        std::cout << “In a!” << std::endl;
}

void b(std::mutex* m) {
        m->lock();
        std::cout << “In b!” << std::endl;
        m->unlock();
}
```

```
//Lock shared by a, b
std::mutex m;
a(&m);
b(&m);
```

Deadlock fixed!

```
void a(std::mutex* m) {
        m->lock();
        std::cout << "In a!" << std::endl;
        m->unlock();
}
void b(std::mutex* m) {
        m->lock();
        std::cout << "In b!" << std::endl;
        m->unlock();
}
```

```
//Lock shared by a, b
std::mutex m;
a(&m);
b(&m);
```

Another Mutex Example

```
void fill(std::vector<int>* vec) {
	for (int i = 0; i < 2000; ++i) {
		vec->push_back(i);
	}
}
```

Another Mutex Example

```
vector<int> nums;
std::thread a(fill, &nums);
std::thread b(fill, &nums);
a.join();
b.join();
std::cout << "size of nums: " << nums.size() << std::endl;
```

Race conditions!

```
vector<int> nums;
std::thread a(fill, &nums);
std::thread b(fill, &nums);
a.join();
b.join();
std::cout << “size of nums: ” << nums.size() << std::endl;
```

A quick fix

```
std::mutex lock;
...
void fill(std::vector<int>* vec, std::mutex* m) {
        for (int i = 0; i < 2000; ++i) {
            m->lock();
            vec->push_back(i);
            m->unlock();
        }
}
```

Semaphores

- Can be thought of as generalized mutexes
  - Instead of only 0 or 1 threads holding the lock, 0..n threads can hold the lock
- Not available in C++11 :(

```
Semaphore* s = new Semaphore(2);
s->dec();
s->dec(); // Semaphore count is now 0, more decs will block
s->inc(); // Semaphore now has space
s->dec();
s->value(); // Returns 0
```

Consumer Producer Queues

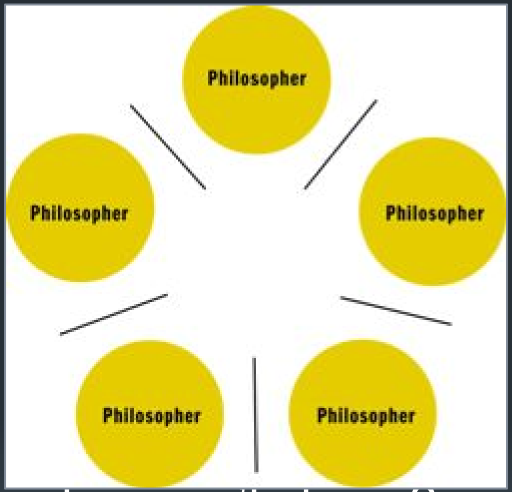
- Some threads produce data
- Some threads read/consume data
- These consumers and producers need to share a fixed size buffer
- Two potential problems
  - Empty buffer, consumer must wait
  - Full buffer, producers must wait
- Apply semaphores?

Consumer Producer Queues

- What do we need to track?
  - Number of slots filled.
  - Number of slots empty
  - Use a semaphore for each value
- Producers
  - Decrement empty slots semaphore
  - Add value
  - Increment filled slots semaphore
- Consumers
  - Decrement filled slots semaphore
  - Use value
  - Increment empty slots semaphore

Dining philosophers

- 5 silent hungry philosophers
- One bowl of spaghetti
- 5 forks placed as shown
- Philosophers alternate eating and thinking
- Must have both forks to left+right to eat
- Cannot grab forks for each other
- Replace forks after eating
- How to design behavior such that each philosopher won't starve?



Proposal

- Think until the right fork is free; when it is, pick it up
- Think until the left fork is free; when it is, pick it up
- When both forks are held, eat for a fixed amount of time
- Then, put the left fork down
- Then, put the right fork down
- Repeat

Deadlock

- All start picking up the right fork at the same time
- Wait forever for the left fork to be ready!

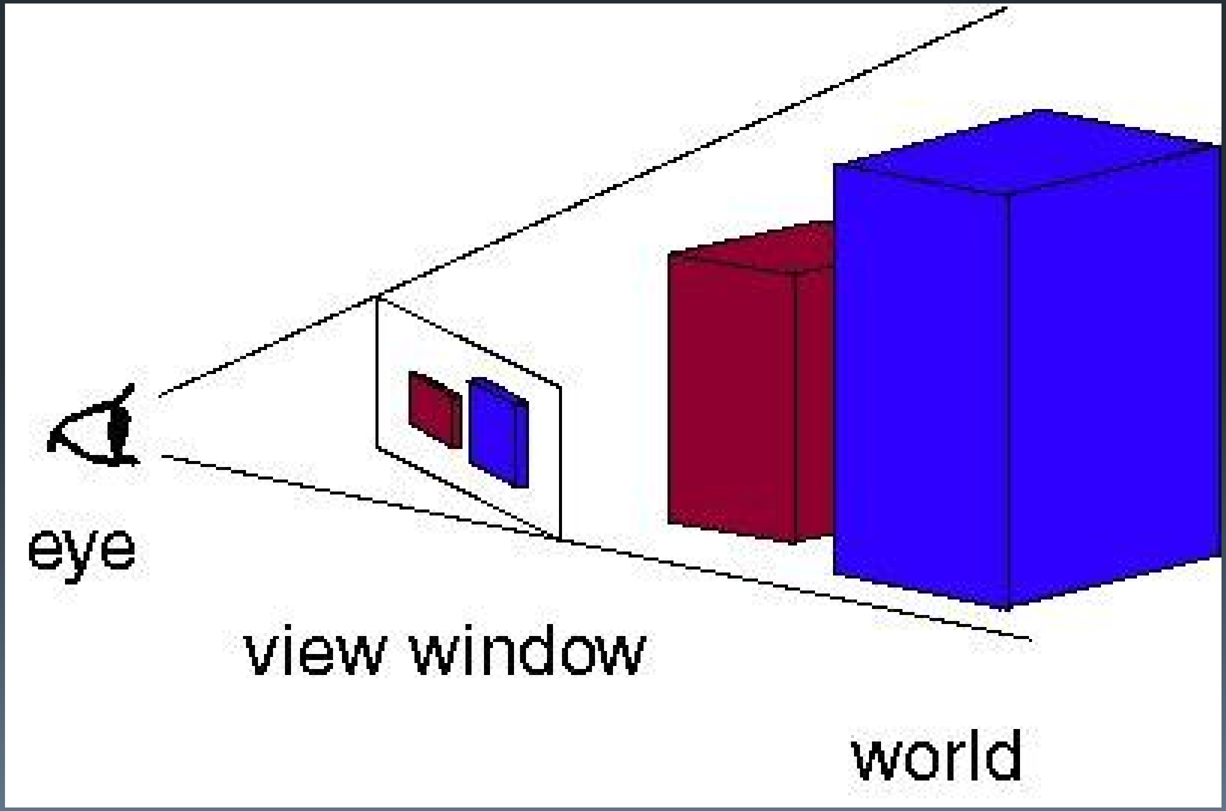
Round Robin

- Since they are silent, imagine a third party helper
- Tells the philosophers when to eat
- Only allows **one** philosopher to eat
- Deadlock?
- Implement using primitives mentioned above

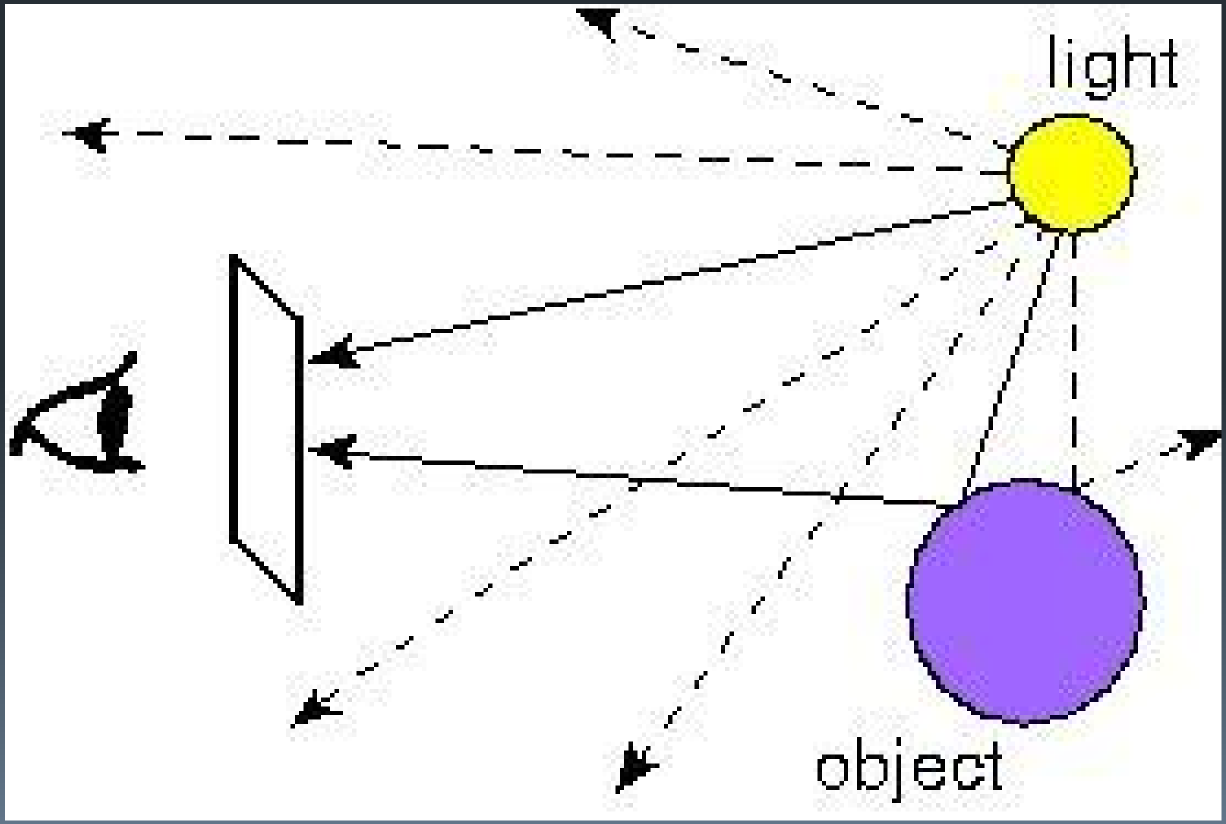
Homework notes:

- Each philosopher runs in its own thread!
- Every philosopher must eventually eat!

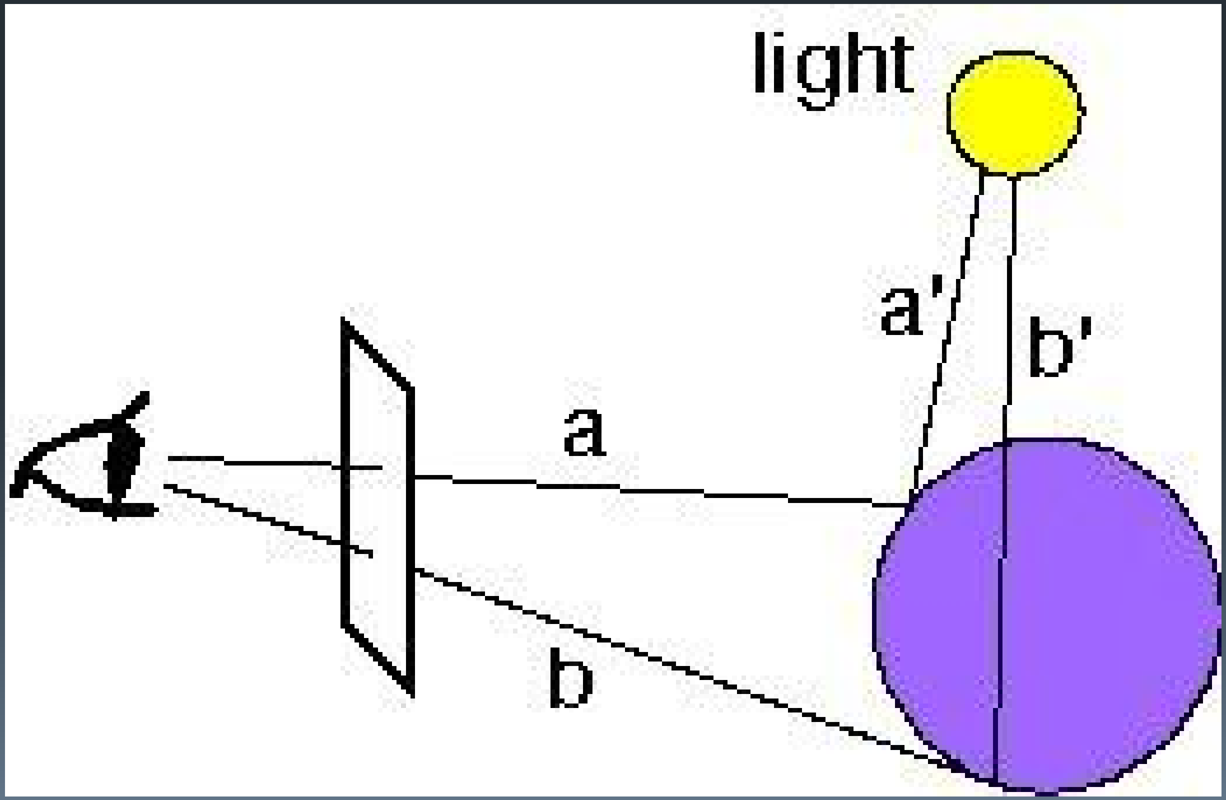
Raytracing!



Raytracing!



Raytracing!



Questions?