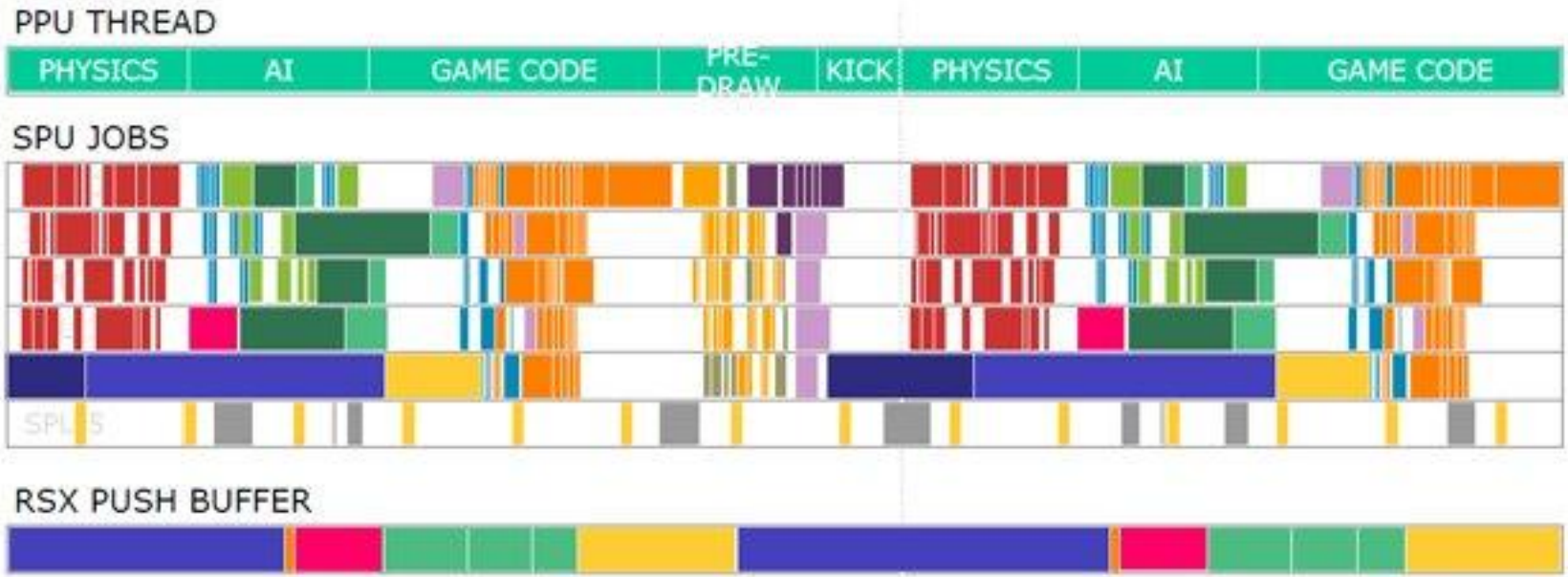
Advanced Games

Parallelism


GDC
Multithreading
the Entire
Destiny
Engine

A 10’000 view of a frame

PS3 Engine


PPU THREAD
PHYSICS
AI
GAME CODE
PRE-DRAW
KICK
PHYSICS
AI
GAME CODE
SPU JOBS
RSX PUSH BUFFER

Interleaved Frames

PPU	Update		Network	Update		Network	Update		Network	Update	
SPU		Physics	Sound	Lighting	Physics	Sound	Light	Physics	Sound	Lighting	Physics
SPU		Physics		Lighting	Physics		Light	Physics		Lighting	Physics
RSX			Render		Output	Render		Output	Render		Output
					Frame 1 Lag: 5			Frame 2 Lag: 3			Frame 3 Lag: 3

Awesomenauts

The threading scheme in Awesomenauts. While one thread updates the game, the other thread renders the *previous* frame. At the end of every frame there is a moment where all objects are copied from the gameplay thread to the render thread. The question is how to make that copying period very short.

Threads

Different Threads for Different Tasks?

Thread Pools

Threads are workers, for all your jobs

Jobs

What could be a singular task?

Who relies on this task?

What data does this task need?

What data does this task write to?

Gameplay Jobs

Update each **Entity** as a job?

Update each **Component** as a job?

Data Race

```
class Player_component{
    float Health;
    float GetHealth();
    void SetHealth(float h);
}
```

```
class Bullet_component{
    void Update(){
     if(near_player){
         float h = player->GetHealth();
         h--;
         player->SetHealth(h);
     }
    }
}
```

Data Race

```
class Player_component{
    float Health;
    mutex player_lock;
    float GetHealth();
    void SetHealth(float h);
}
```

```
class Bullet_component{
    void Update(){
     if(near_player){
         player->player_lock.lock()
         float h = player->GetHealth();
         h--;
         player->SetHealth(h);
         player->player_lock.unlock()
     }
    }
}
```

Data Race

```
class Player_component{
   public:
    void hit(float h);
   private:
    float health;
    mutex lock;
}
Player_component::hit(float h){
   std::lock_guard<std::mutex> guard(lock);
   health--;
}
```

```
class Bullet_component{
    void Update(){
     if(near_player){
         player->hit(1.0)
     }
    }
}
```

Data Race

This would work, but has major drawbacks:

- You must manually add a mutex lock to every function.
- **Deadlocks** can happen if components are tightly coupled.
- Performance goes serial if you components are too loosely coupled.

Four Solutions

Solution 1
Just Deal With/ Forget about it

Place manual locks. Works for small number of components, wherein you can guarantee no deadlocks.

👍Ship it now
☹*could* break
☹won't scale

Solution 2
Work out Dependencies, update accordingly

Build a graph somehow. Process the graph in parallel.

☹Pointless for small systems
👍Fast
👍Almost Deterministic

Solution 3
Lightly-Decouple, use Event passing

☹Messy.
☹Code bloat.
☹Hard to see code flow.

👍 Scales well to big systems
👍 Decoupling allows for more procedural/organic systems.

Solution 4
Completely-Decouple, Multiple Data States

Use message passing, keep immutable copies of old data, serialise writing of 'new' states.

☹Complicated code
👍Data Oriented design
👍You have to do this anyway for multiframe processing