Parallel Architecture for Games

Objectives

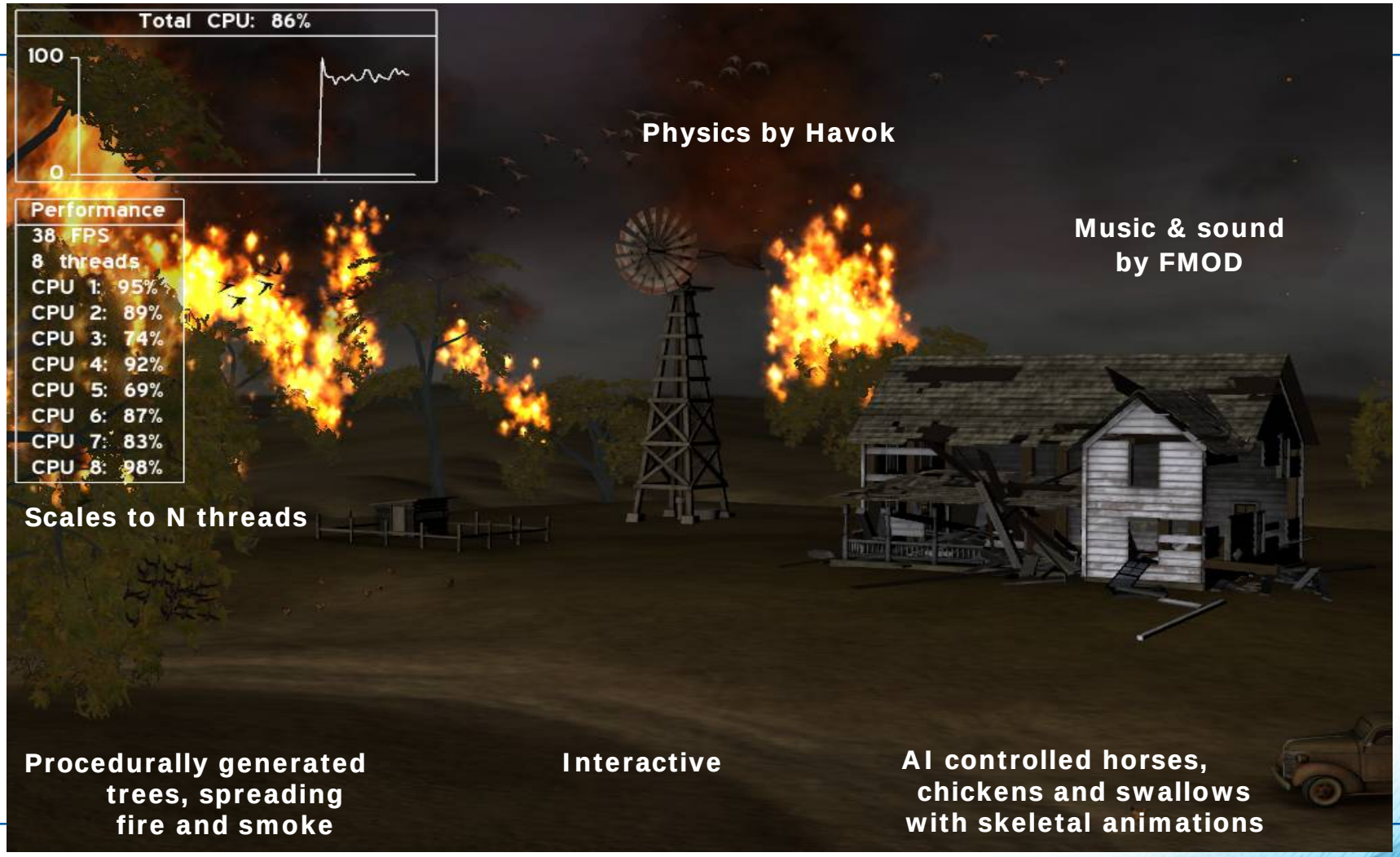
At the end of this session, you will be able to:

- Understand how change synchronization is handled in the parallel architecture framework and how it contributes to the scalable game framework.
- Recognize how common interfaces are used to interact between Systems (AI, Physics, Graphics, etc).
- Comprehend the components and tasks that need to come together to deliver each frame for a particular workload.

Agenda

- **Introduction** — What is this technology demo
- **Framework** — How the framework was designed to support parallel computing
- **Challenges** — Potential issues and solutions of a highly threaded game engine
- **Summary** — Curriculum application and Conclusions

Smoke Demo



Optimize Available CPUs With Parallel Architecture For Games

Framework built to support N-threads

Tech Demo uses real game technologies (Havok, FMOD, Ogre3D, DirectX9)

Partitioned and configurable to produce multiple demos

Why Parallel Architecture for Games?

Performance

- The parallel framework was designed to support threading functionality that can maximize CPU usage and FPS throughput

Prototype

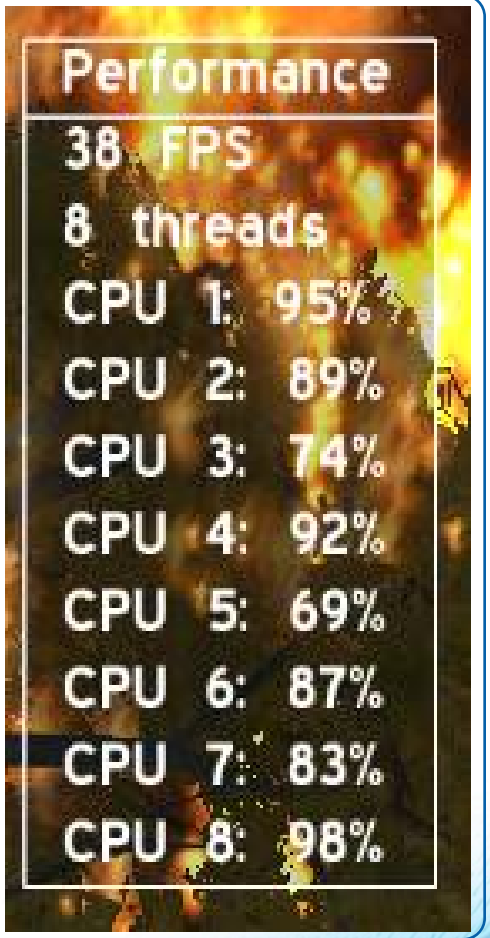
- Partitioned and configurable
  - Described with simple XML files
- Explore new game technologies (e.g., procedural fire/smoke)

Explore

- Examine threading techniques
- Understand interaction between systems

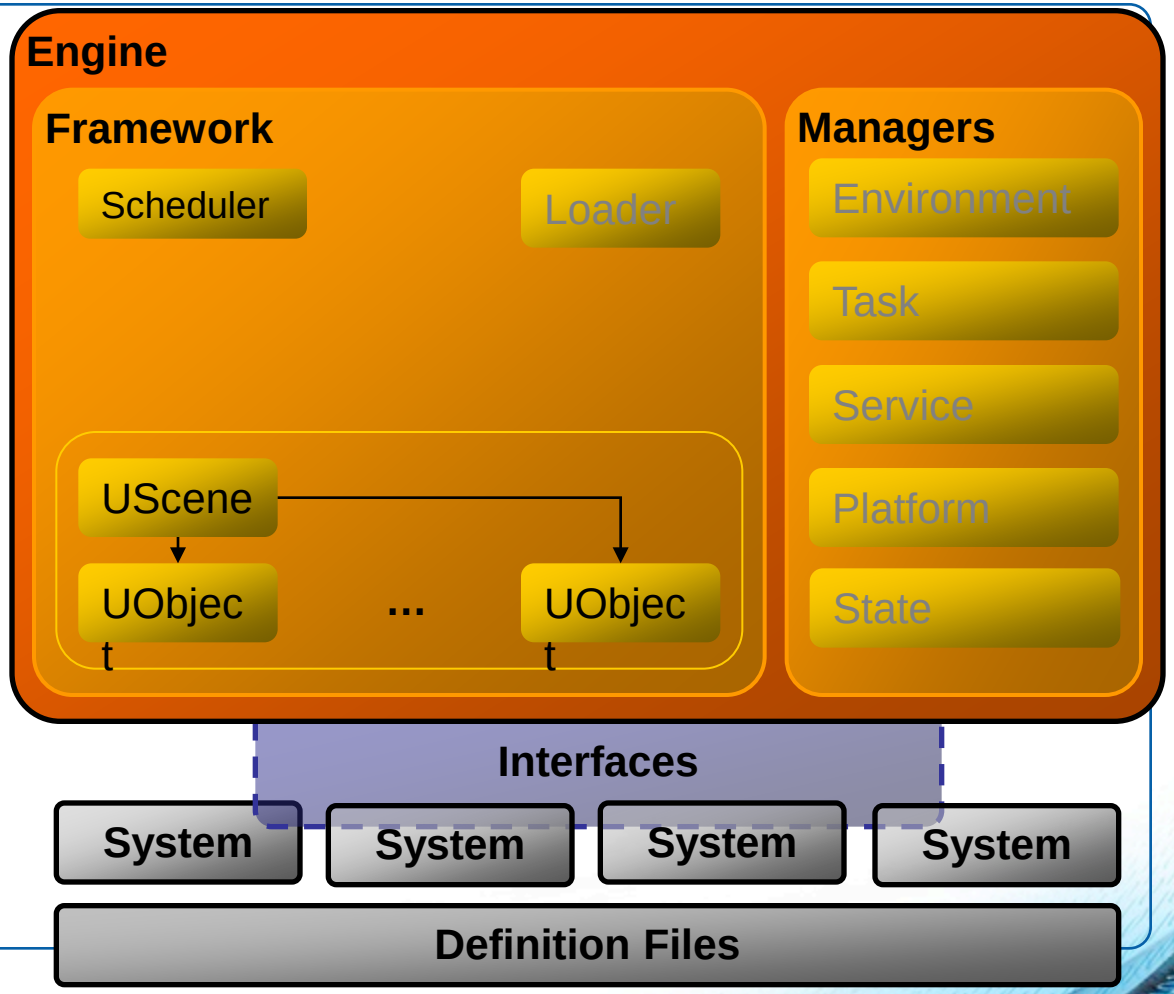
Teach

- Share source, demos, samples, workloads, white papers



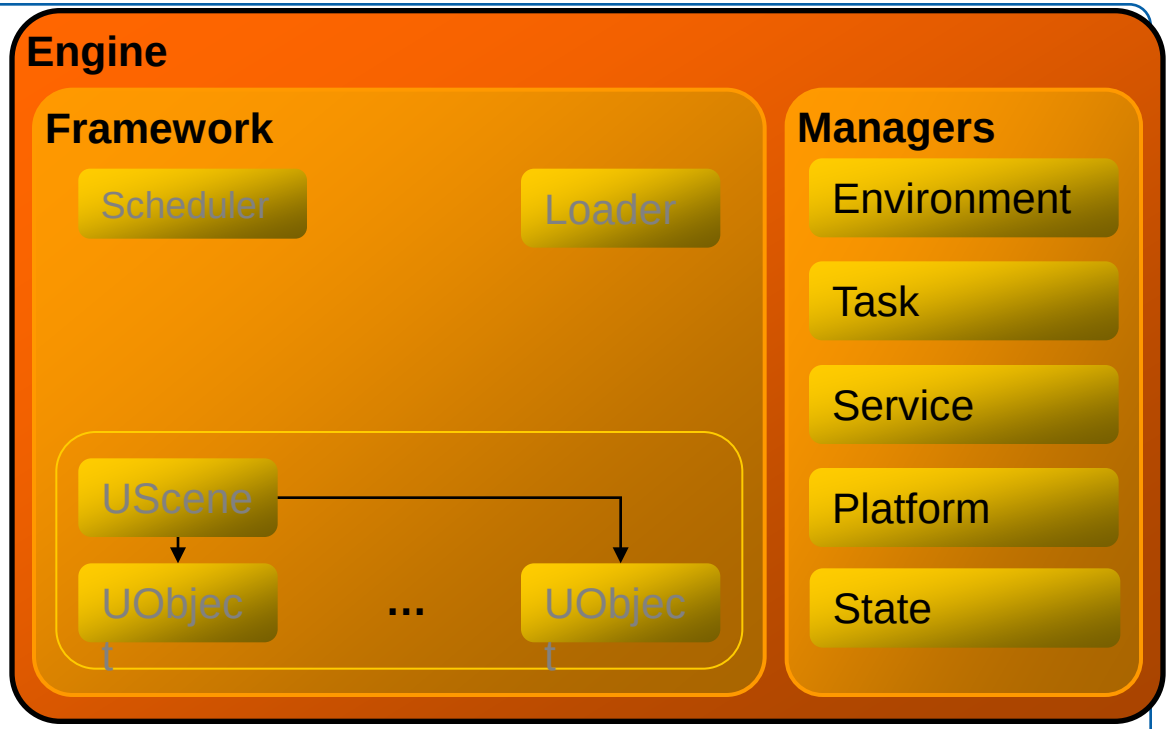
The Engine and Framework

- Scheduler manages System jobs
- Data structured to support independent processing
- System modularity (through interfaces)
- Systems are specific to the demo (e.g. AI, physics, etc.)



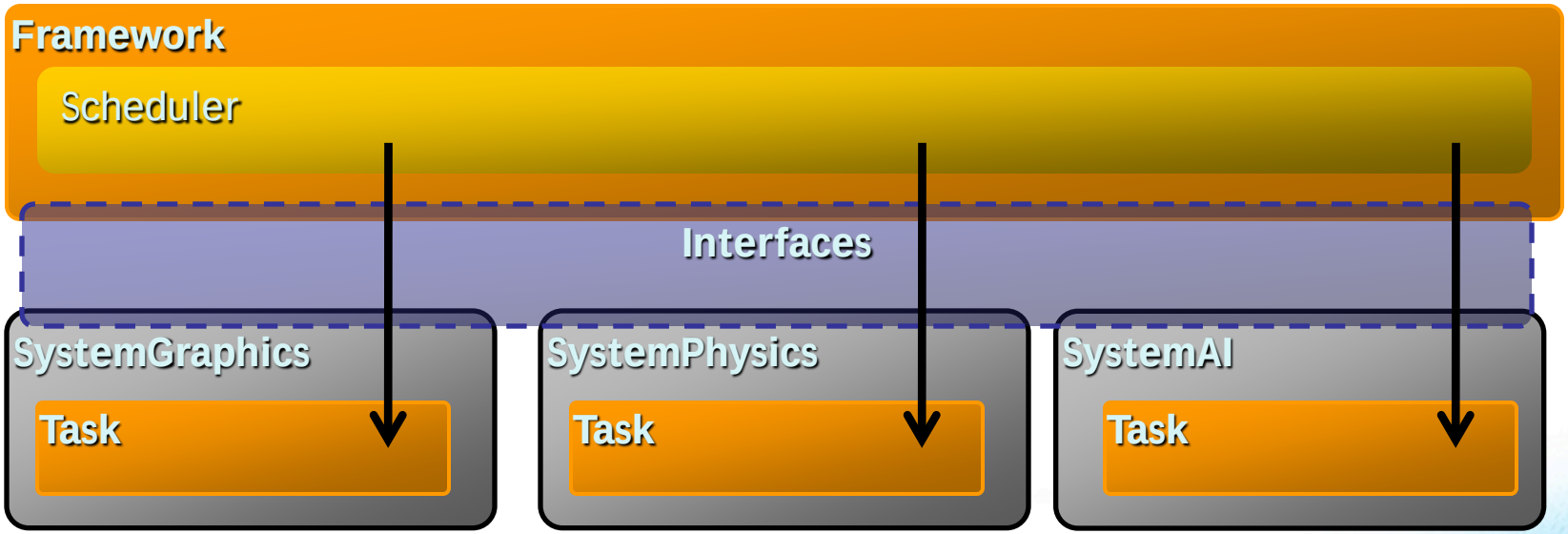
Manager Components

- Service Manager used to propagate data between Systems
- State Manager (change control) minimizes thread synchronization



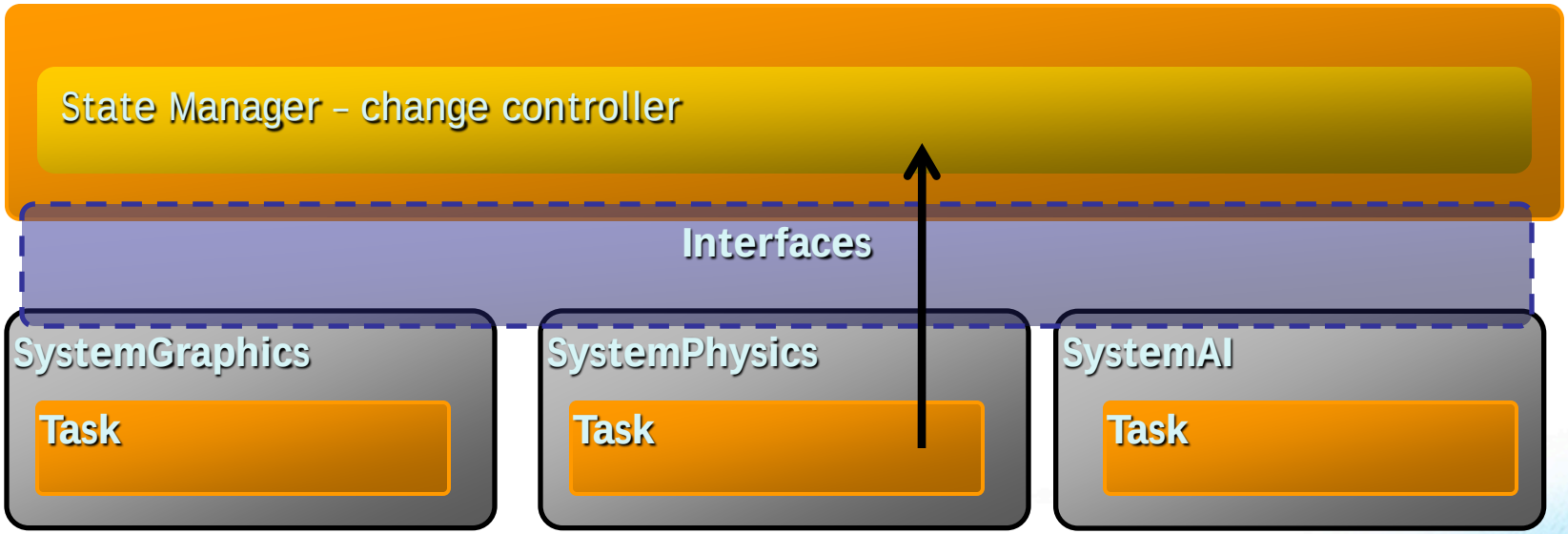
Scheduler Manages System Jobs

- Each System is a DLL
- Each component implements the proper interfaces to interact with the Framework (interfaces are just pure virtual base classes)
- The Scheduler will invoke the task for each system once per frame



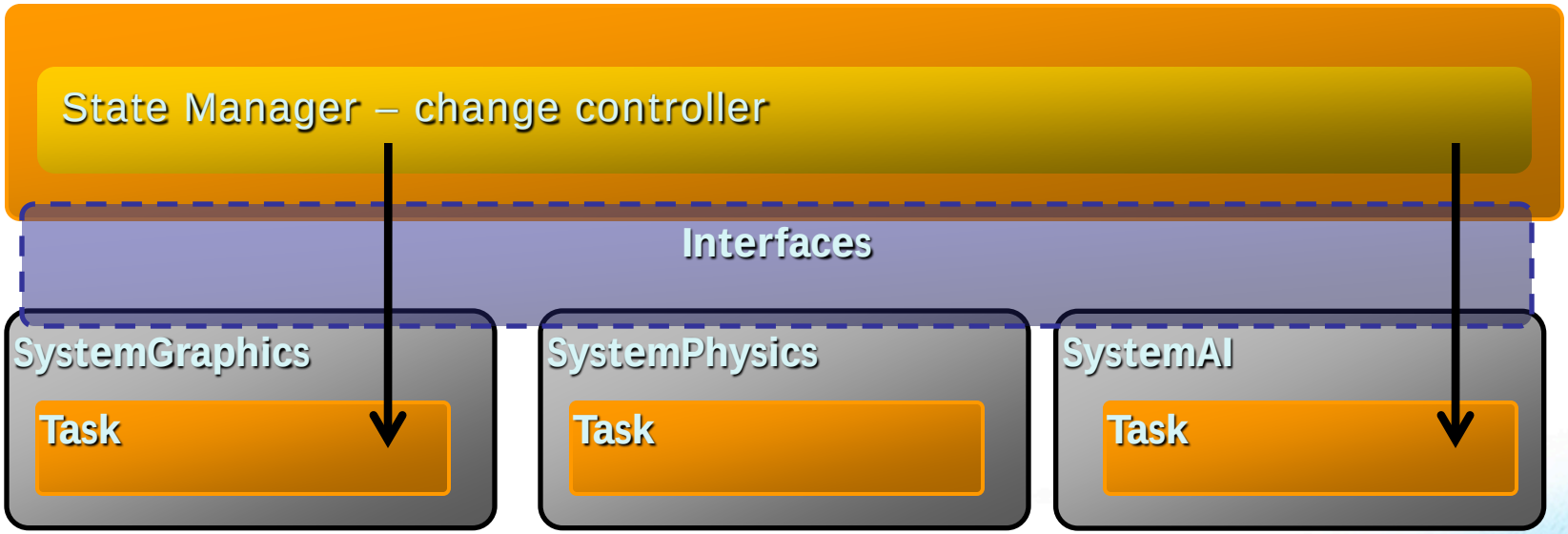
Change Controller Minimizes Thread Synchronization…

- Data is shared between the Systems with the change controller
- If one System changes an Object within a System…



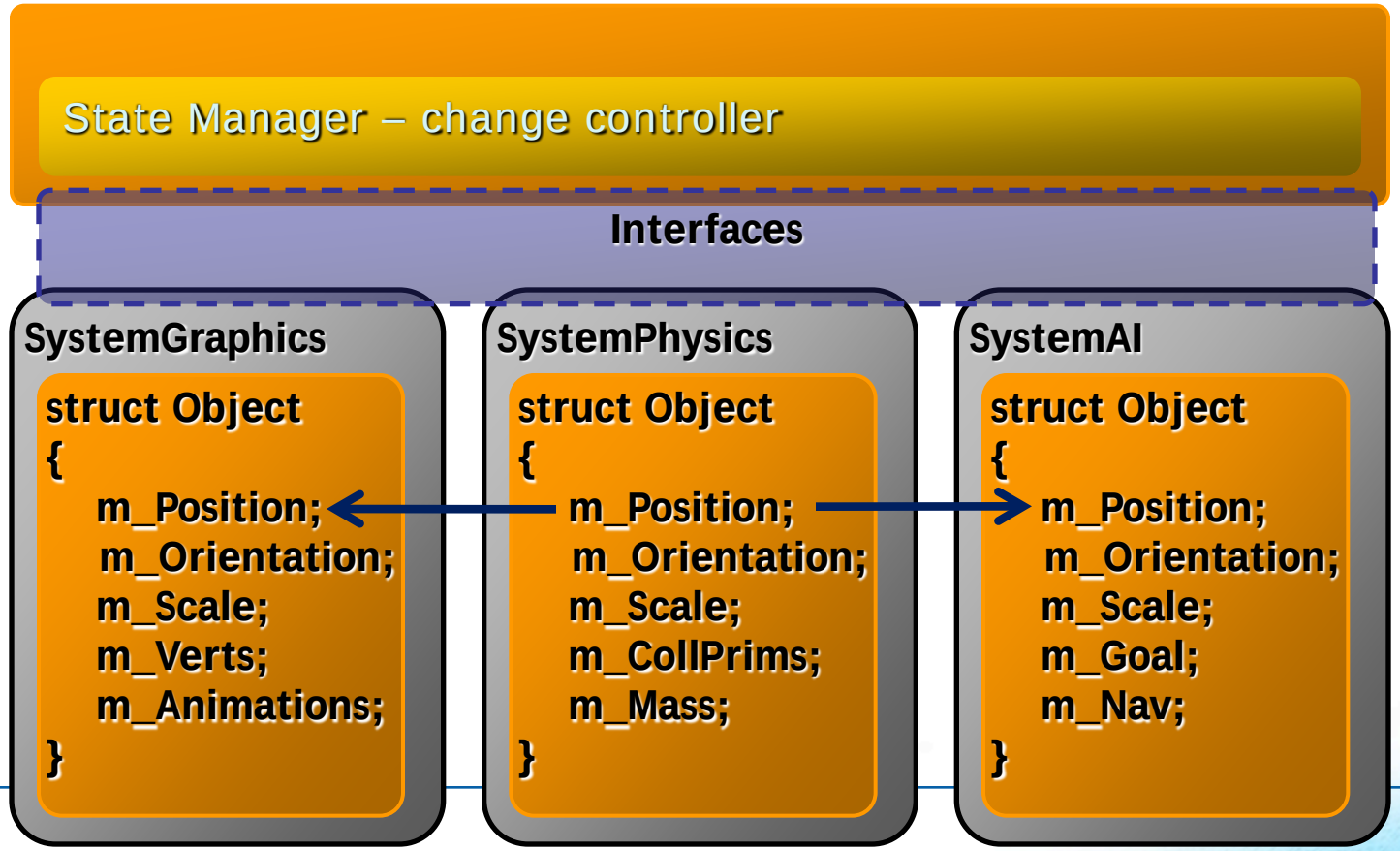
Change Controller Minimizes Thread Synchronization

- Data is shared between the Systems with the change controller
- If one System changes an Object within a System… all Systems with a matching Object will be notified about the change
- This allows Systems to work independently



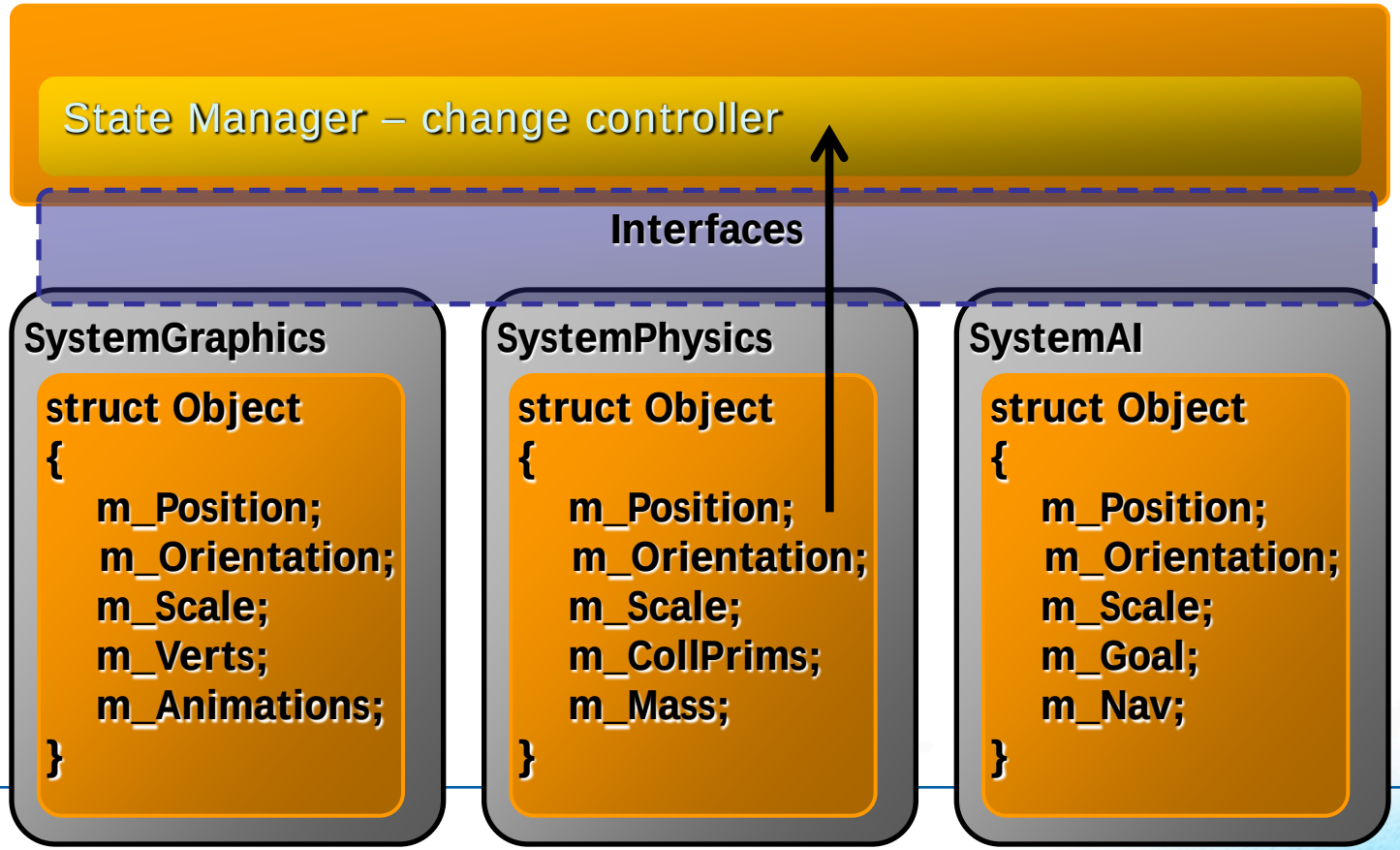
Data Structured To Support Independent Processing

- Each System subscribes to desired changes during initialization



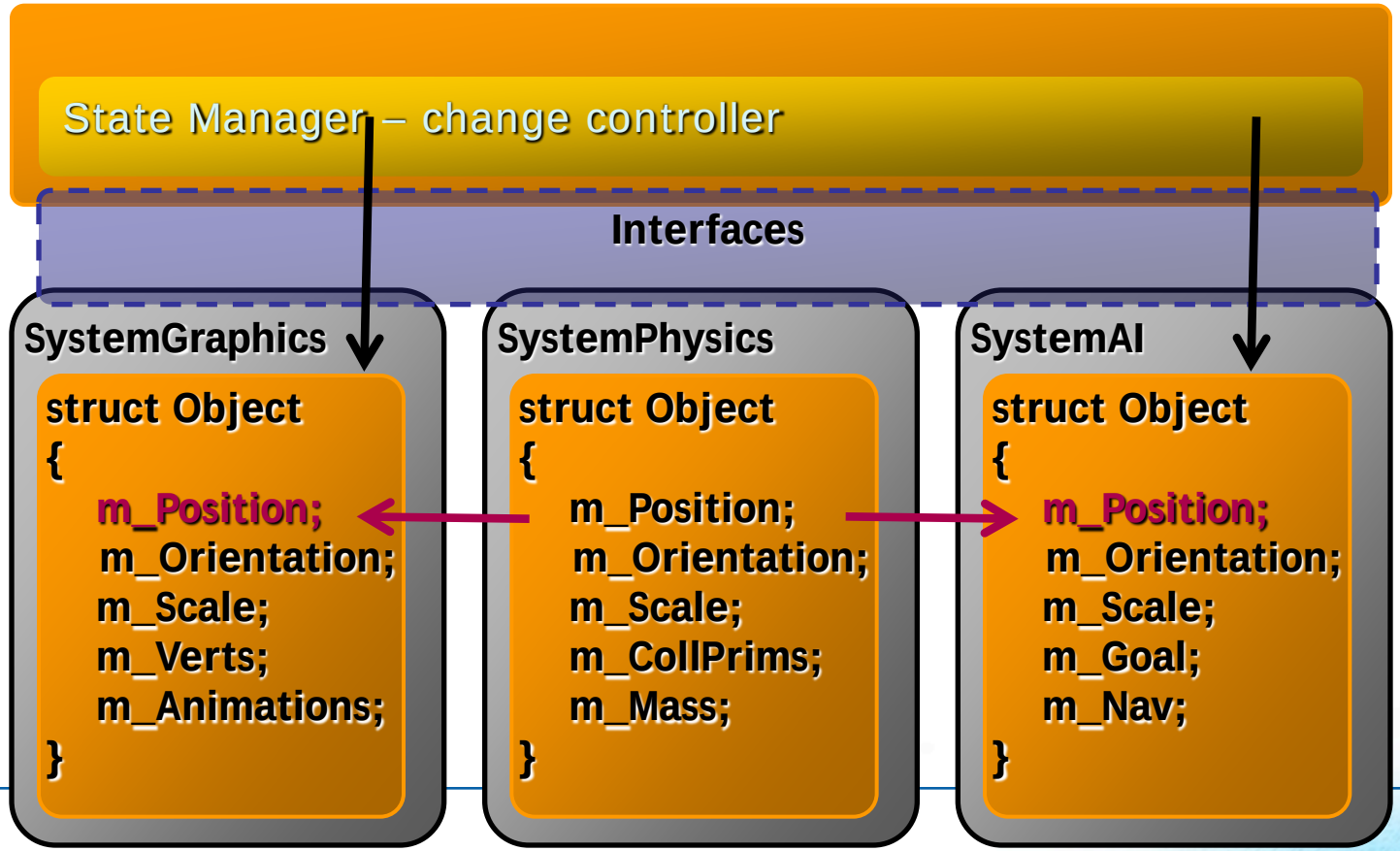
Each System Stores A Copy Of The Data It Needs

- When a System modifies its data, it tells the change controller



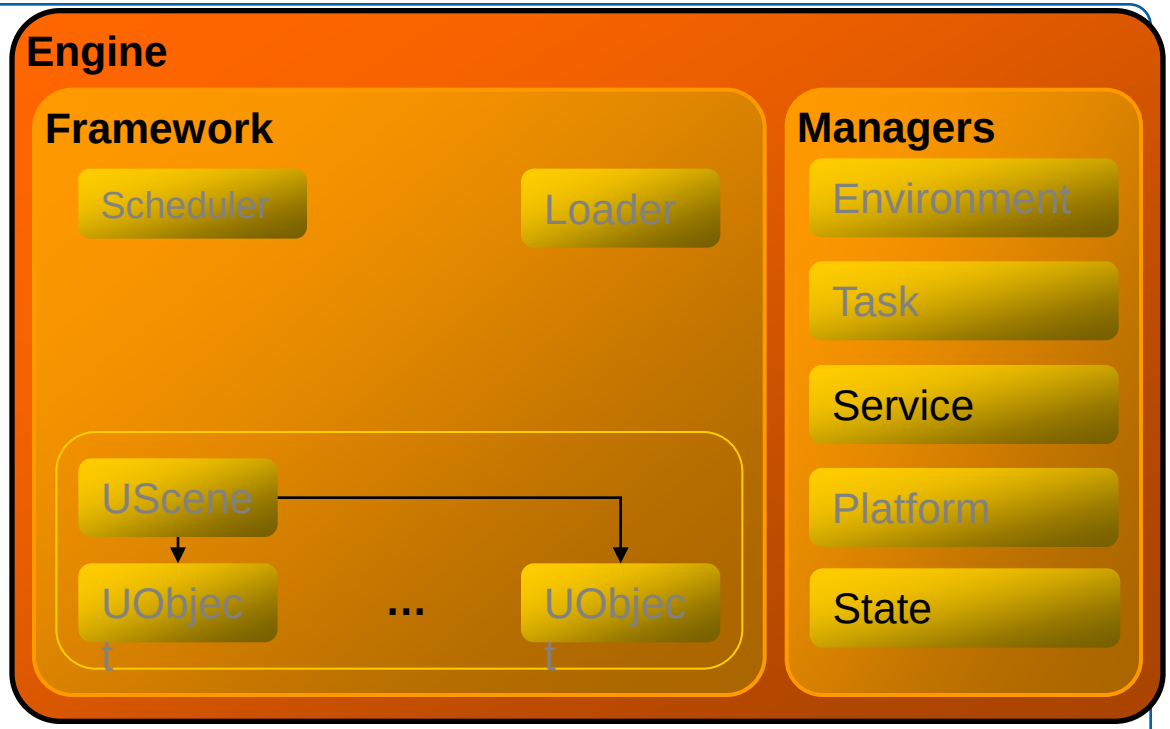
Systems Copy Data From Other Systems As Needed

- When a System is told about a data change, it will get all of the required information from the changing System



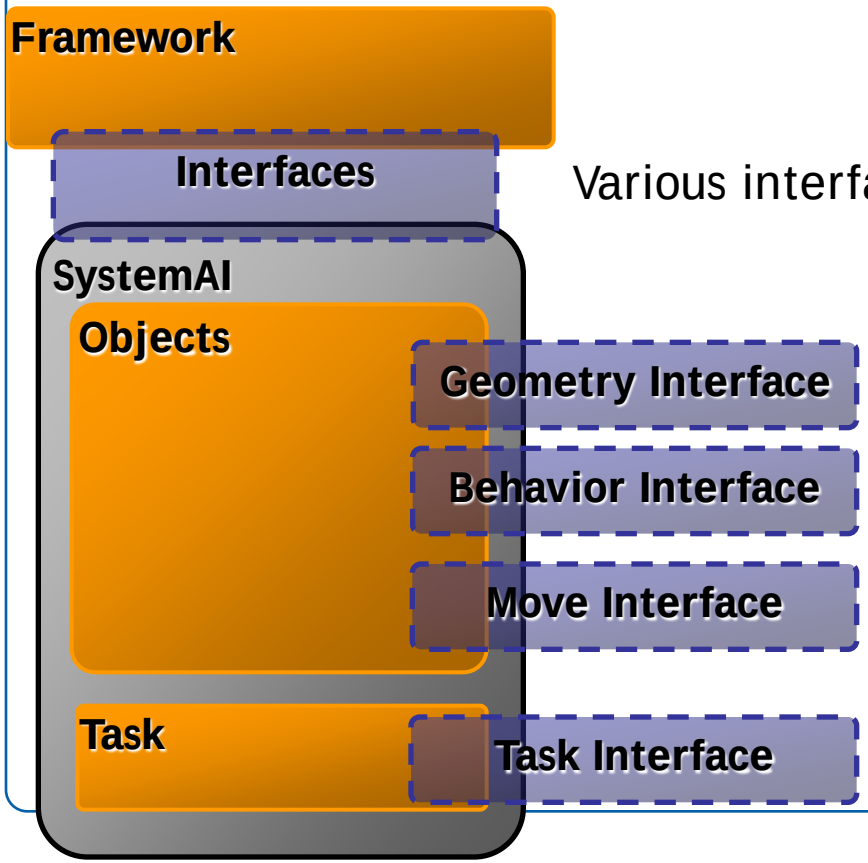
Manager Components Continued

- Service Manager used to propagate data between Systems
- State Manager (change control) minimizes thread synchronization but imparts frame latency



System Modularity - Through Interfaces

- Interfaces are used to build up the interaction between Systems. This allows for efficient modularity.



Various interfaces to work within the Framework

Change positions, etc.

Change behavior (generic state)

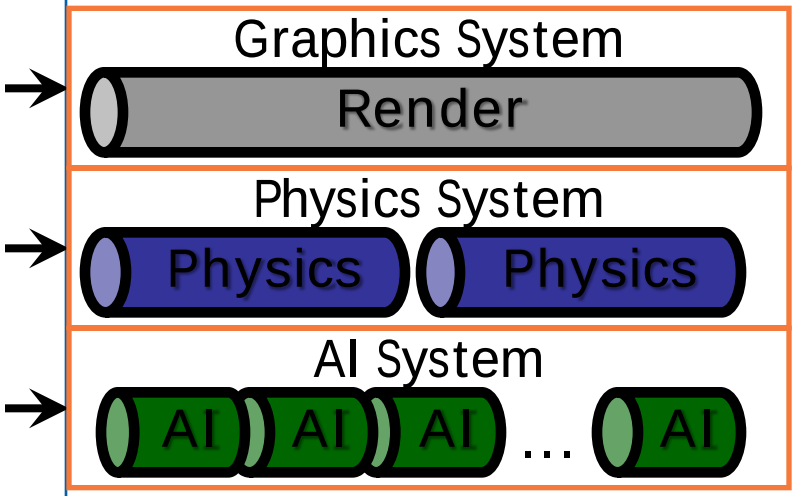
Moving object (velocity, etc.)

Update and do work

Put It All Together And See A Frame

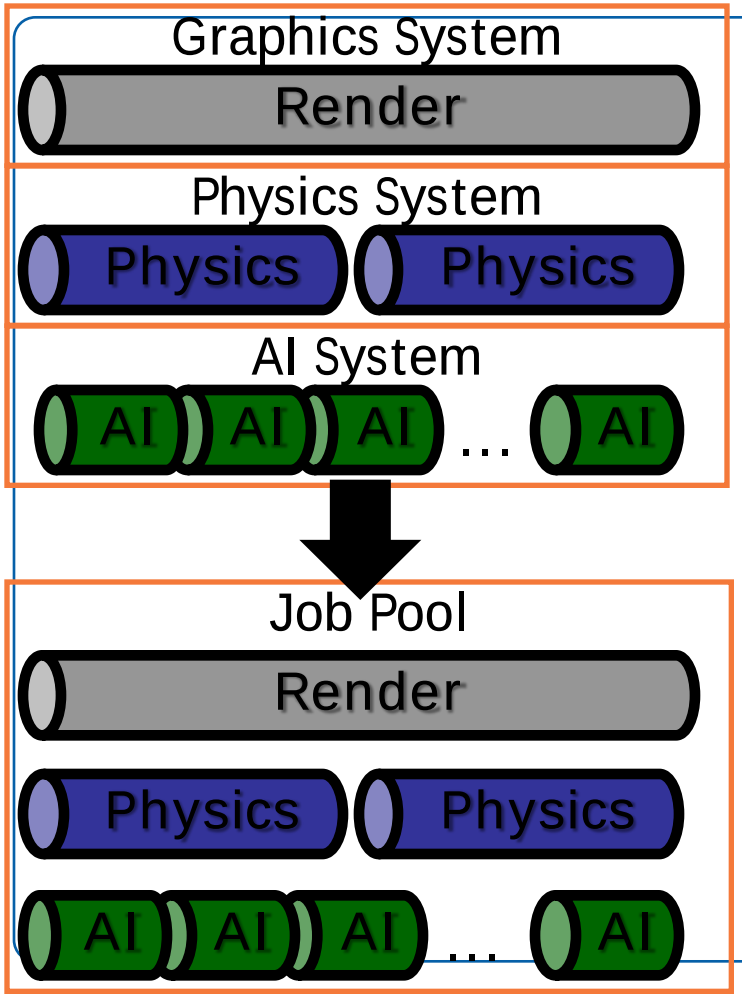
1. At the start of a frame, Systems run and subdivide tasks
2. Sub-tasks are added to a pool
3. Worker threads process sub-tasks
4. Tasks post changes as needed
5. Changes sent to observers at the end of the frame

1. Systems Subdivide



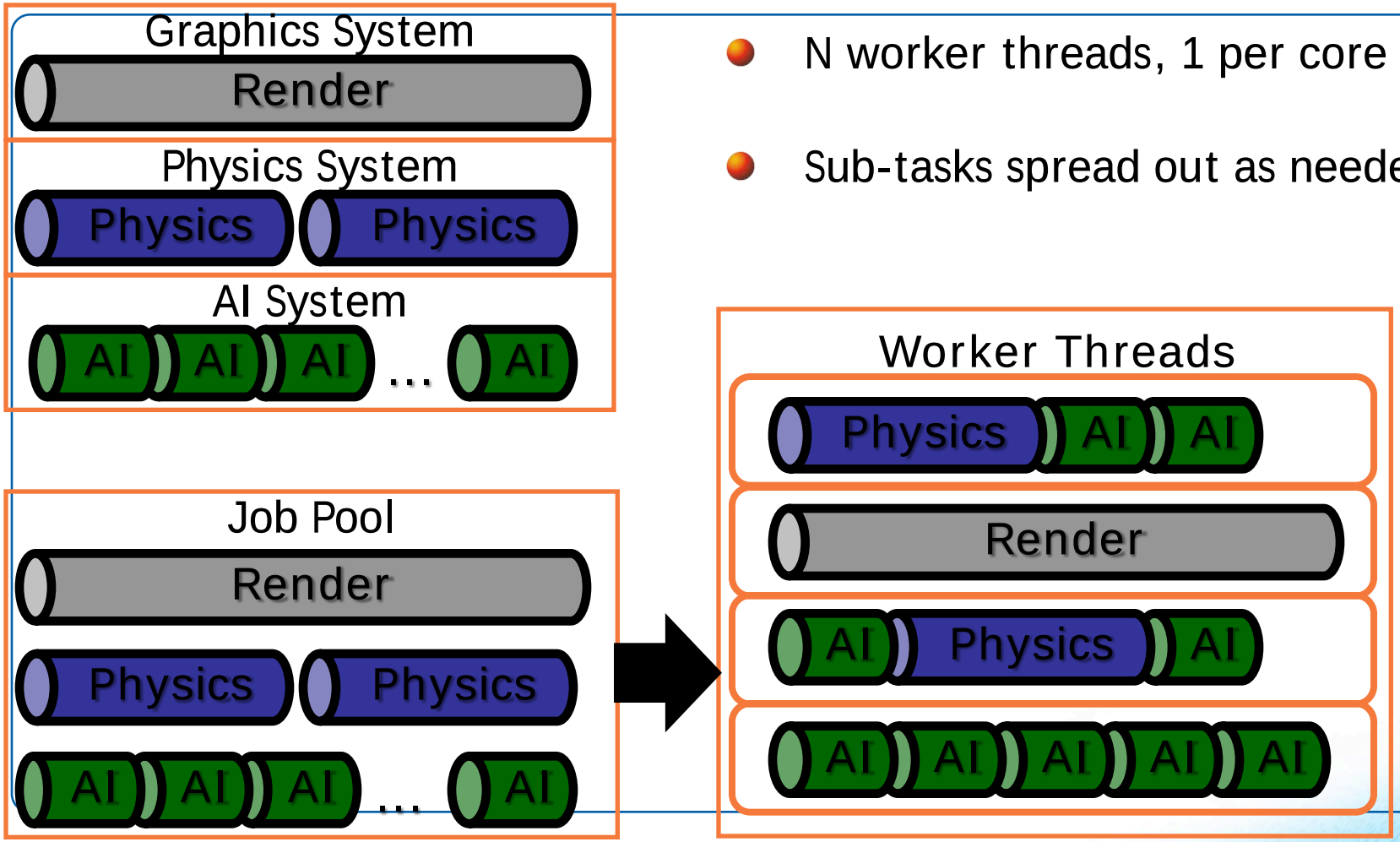
- Scheduler invokes each system per frame
- Systems subdivide work into sub-tasks
- Systems can subdivide work based on a "natural" granularity
- Good middleware makes this easy

2. Add Sub-Tasks To A Pool



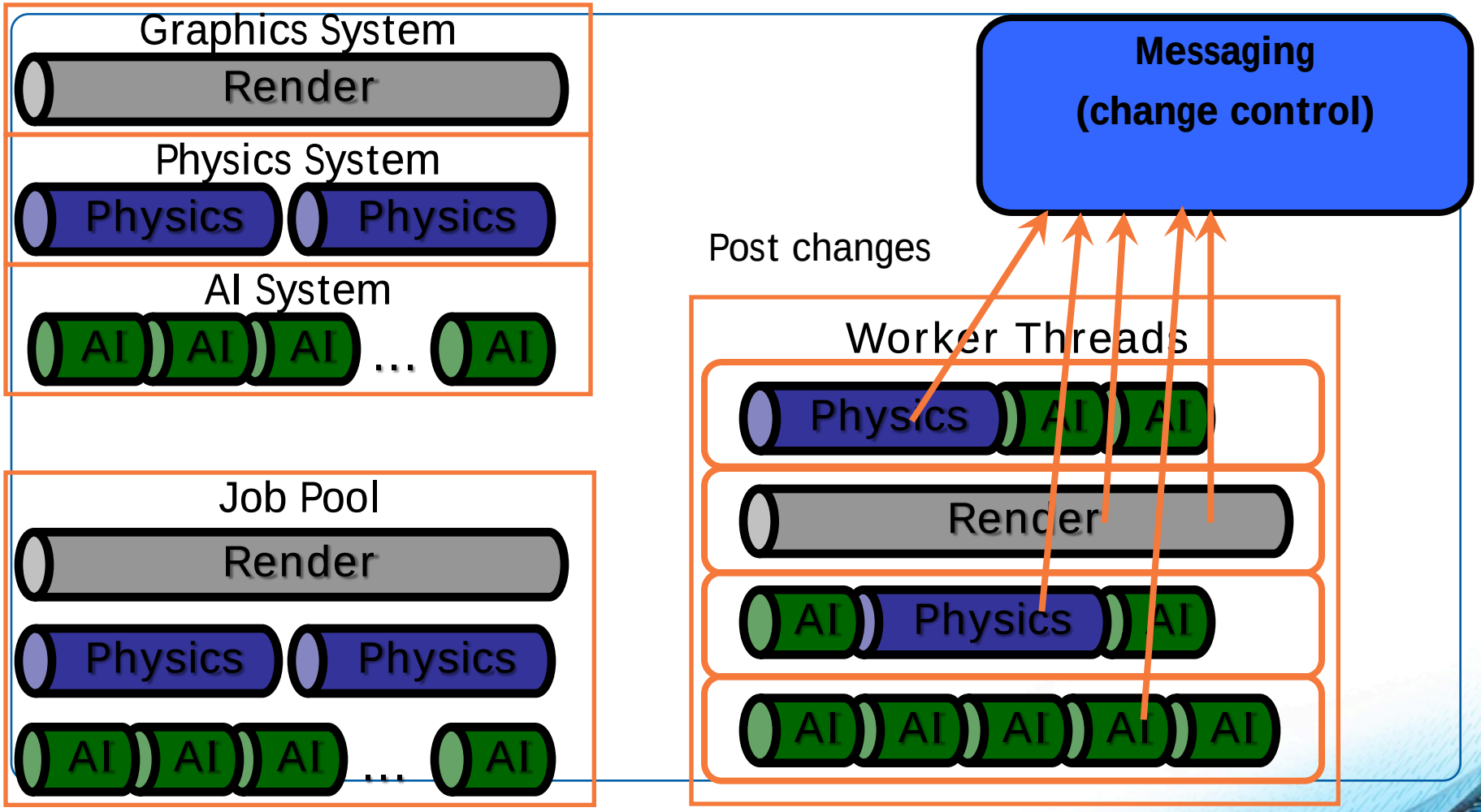
- All sub-tasks in single job pool

3. Worker Threads Process Sub-Tasks

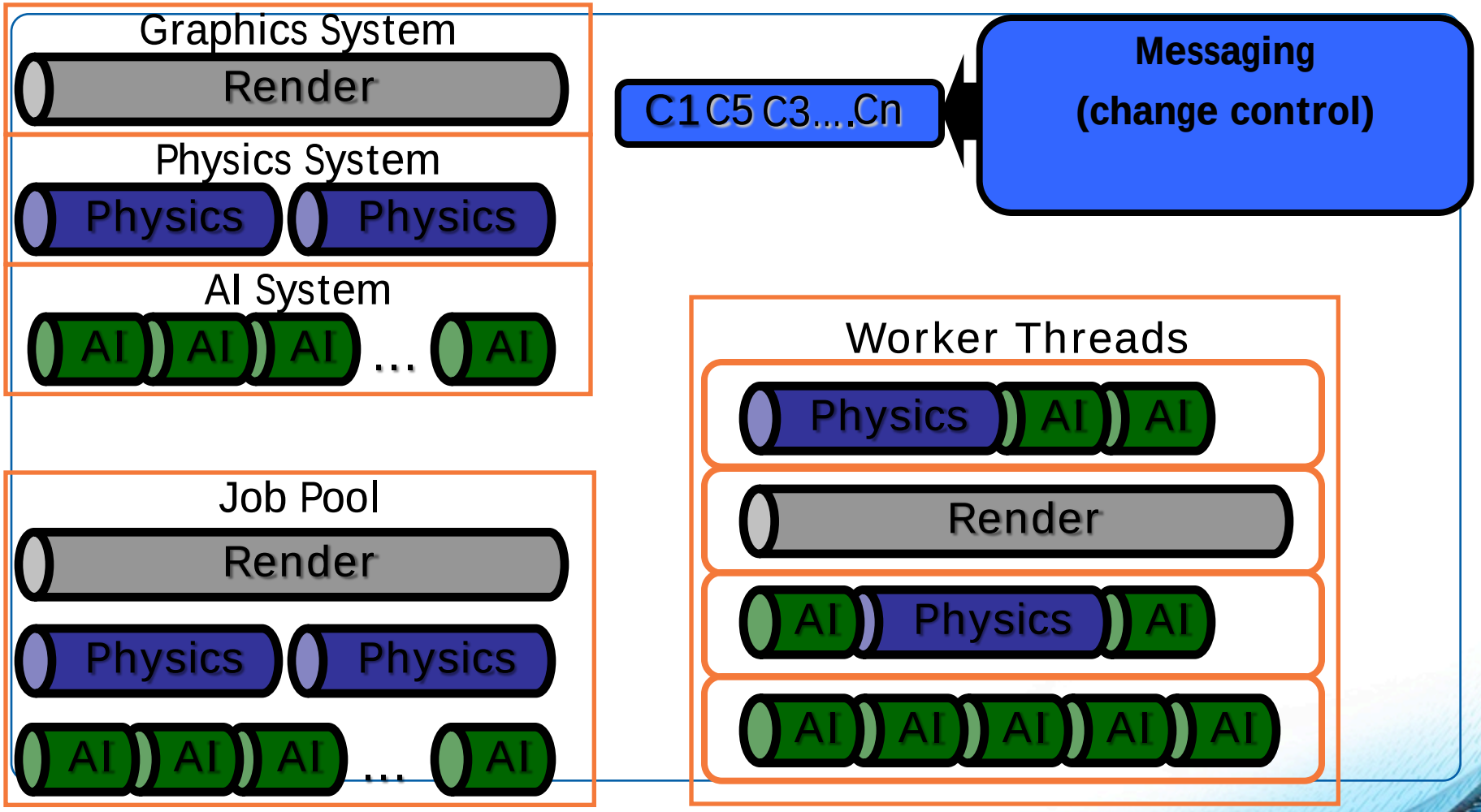


- N worker threads, 1 per core
- Sub-tasks spread out as needed

4. Tasks Post Changes



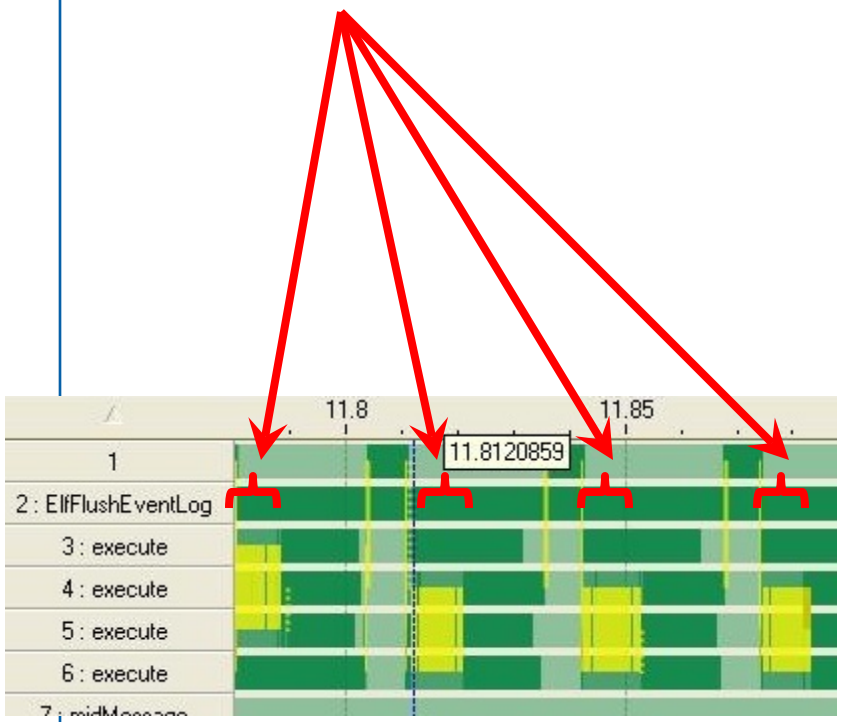
Worker Threads Post Changes To The Change Queue

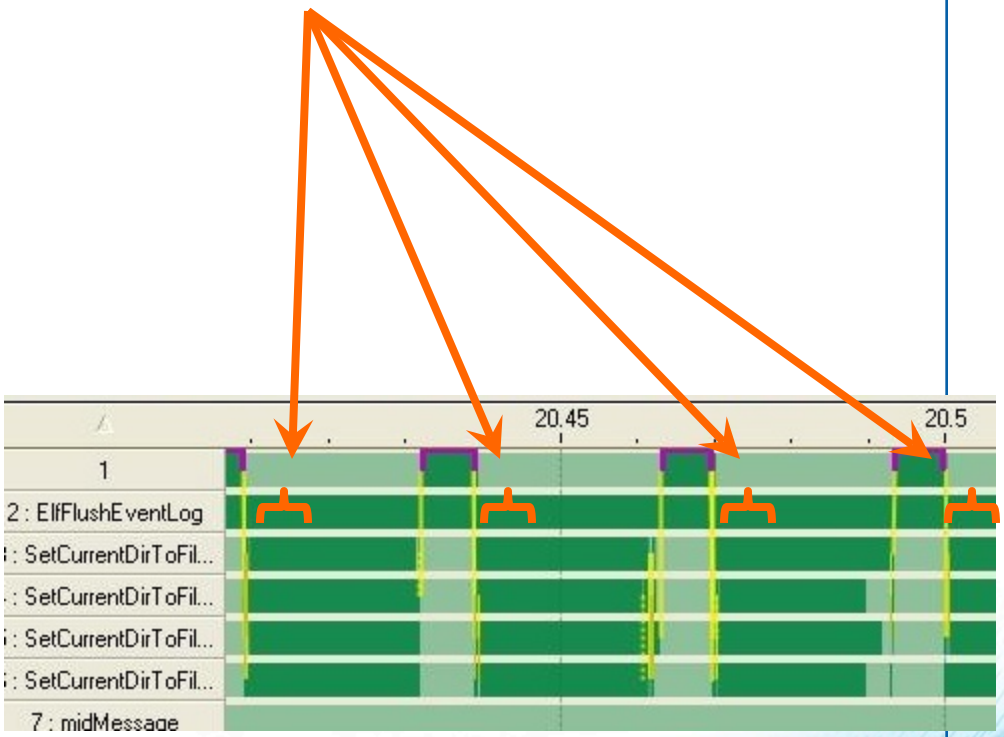


Minimizing Serialization Is A Challenge

Heavy contention on global sync object within the change controller

Per-thread sync within the change controller

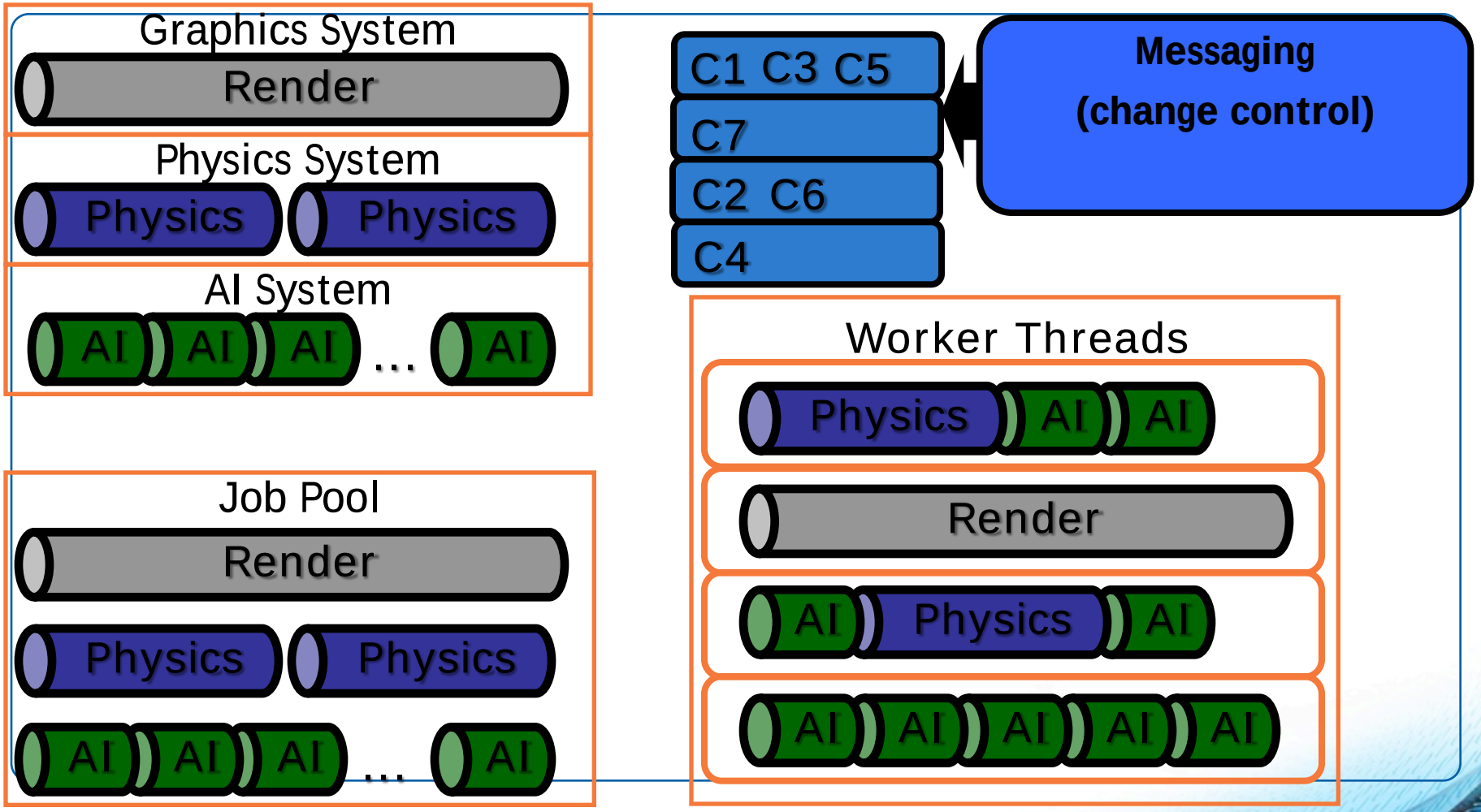




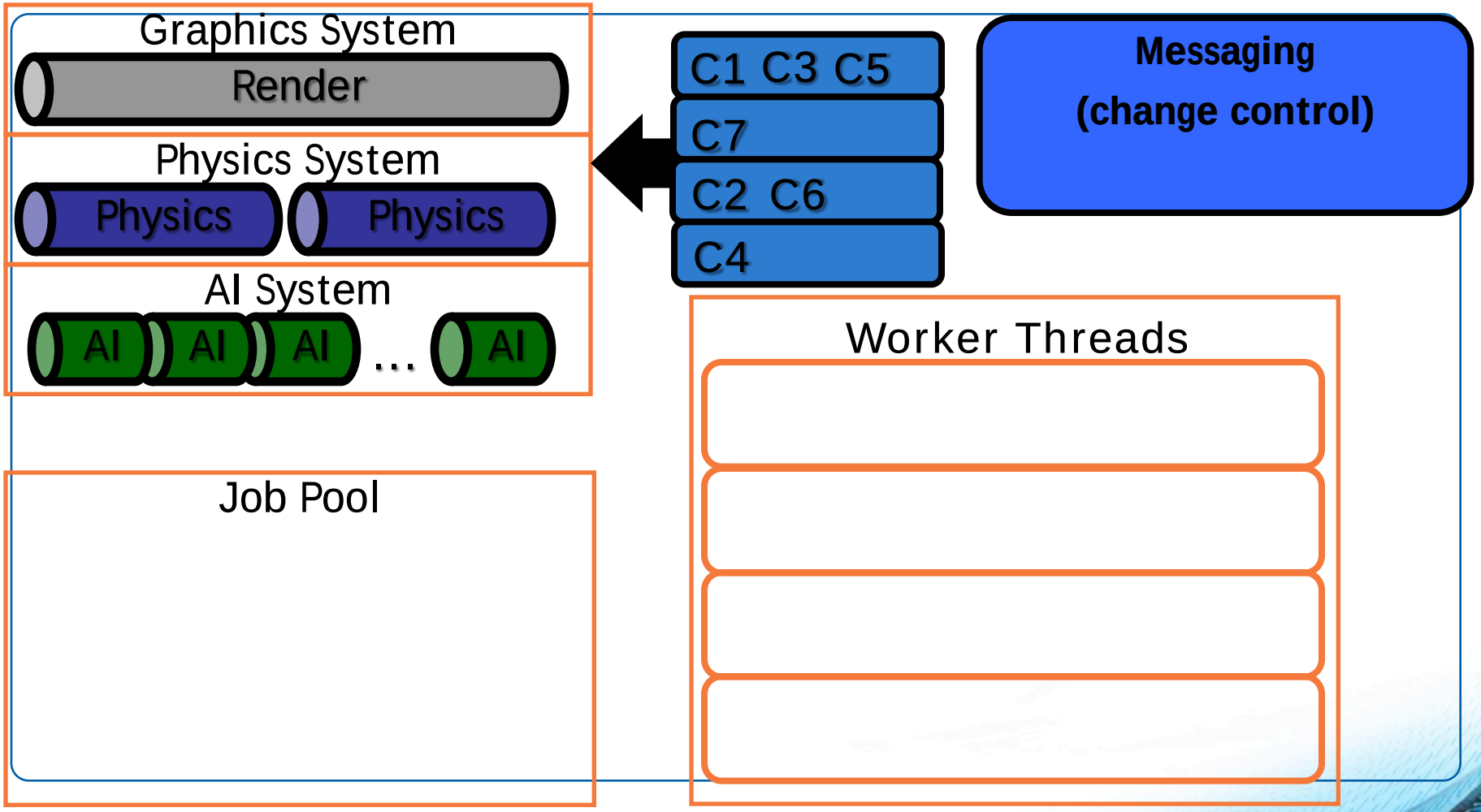
Before optimization

After optimization

Worker Threads Now Have Unique Change Queues



5. Changes Are Sent To Observers



Lessons Learned

- Do:
  - Keep systems modular to help agile development
  - Evaluate the features of your middleware for integration
  - If possible, support multiple threading subsystems to ease debugging
- Don't:
  - Ignore thread interaction between systems, especially middleware
  - No one method works for everybody

Curriculum Requirements And Application

- The following courses define the suggested pre-requisites for this material – most should be at the advanced level:
  - **Operating Systems**: Basic issues in concurrency, deadlock control, synchronization scheduling, memory management, protection and access control, inter-process communication, and structured design.
  - **Computer Graphics**: Introduction to graphics display hardware and applications, interactive techniques, 2D scan conversion, 2D & 3D transformations, 3D viewing, visible surface algorithms, illumination models, smooth shading, ray tracing, shadows, transparency, texture mapping.
  - **3D Graphics and Rendering**: The process of creating images from 3D models. Includes transformations, shading, lighting, rasterization, texturing, and other topics
  - **Parallel Programming**: Exploration of parallel programming paradigms, parallel computing architectures, contemporary and historical examples and their impact, context with parallel algorithms.
- Course where this material directly applies:
  - **Game Engine Development**: principles of developing game engines targeted at modern PC hardware.

Teaching Strategies

- Students will be taught how the Parallel Architecture for Games engine operates; as well as being able to both manipulate configuration files and add new objects to an existing scene using the source code provided (in a Windows based environment).

- Additional Student Projects
  - A short project; replacing existing middleware components (Havok), with similar functionality (Newton physics or ODE) to show the pros and cons of this endeavor.
  - A project to optimize and improve the serial portion of this architecture (data synchronization). In addition improve the inherent data latency issues with change control could also be explored.
  - A semester project to create a totally new game with new objects (with AI interactions) and show how this project performs based on threading tools (with optimum performance) and the number of CPU cores.

Summary And Key Take Away

- Why was Smoke Developed?
- Think and breathe Parallel Computing - It's ubiquitous!

Backup

Example: Configuration File

Scene Definition file for an instance of a bird

```
<Object Name="Bird0">
  <Properties SystemType="Geometry">
     <Property Name="Position" Value1="-3200" Value2="1000" Value3="145"/>
     <Property Name="Orientation" Value1="0.0" Value2="0.0" Value3="0.0"
     Value4="1.0"/>
      <Property Name="Scale" Value1="15.0" Value2="15.0" Value3="15.0"/>
   </Properties>
   <Properties SystemType="Graphics" ObjectType="MeshAnimated">
      <Property Name="Mesh" Value1="swallow.mesh"/>
      <Property Name="Skeleton" Value1="swallow.skeleton"/>
      <Property Name="Animation" Value1="flying" Value2="Flying"/>
   </Properties>
   <Properties SystemType="AI">
   </Properties>
</Object>
```