CS 354R: Computer Game Technology

Component-based Software Design
Fall 2017
Game Engine Architecture

- Good practices lay a good foundation
Inheritance-based Architecture

- Deadly diamond
- Hard to maintain
- Messy structure
- Potential memory penalties
Problem

• Entities inherently occupy multiple domains
• Domains should remain as isolated as possible

• How can we incorporate both of these principles?
Component-based Architecture

- Break domains into component classes
- Entity acts as a container of components
Potential Domains

- Domains can be a single component or broken into multiple components:
  - Input
  - Graphics (Rendering + Animations)
  - Physics (Collision + Forces)
  - Sound
  - GUI
  - AI (Sensing + Thinking + Acting)
  - Game Logic
Abstract Base Classes

- Components as abstract base classes
- Implemented via interfaces
GameObjects Without Inheritance

- No need for GameObject inheritance
- Instantiate GameObjects based on selected components:

```cpp
GameObject * createPlayer1() {  return new GameObject(new Player1InputComponent(),  new Player1PhysicsComponent(),  new Player1GraphicsComponent());
}
```
Pure Component-based Design

• If we take this model of container classes with components to its extreme, what happens?
Entity Component System

- A specific form of component-based architecture
- Entity is only a container and id
- No methods/fields embedded in the entity
- Data-driven with database storage

How can these systems communicate?

(http://www.alecmce.com/)
Communication Between Classes

• Direct Reference
  • Components have references to relevant components
  • Example:
    • GraphicsComponent contains a reference to Physics Component
    • Updates sprite upon collision

• Message Passing
  • Component sends message to container class
  • Container class broadcasts message to its components
  • Same idea as event-driven programming
Discuss

- Consider your current projects
- How could they be reengineered to better take advantage of component-based principles?
- What sorts of advantages and disadvantages might arise?
Component Advantages

- Understandable
- Maintainable
- Flexible
- Can have memory advantage
Component Disadvantages

- More upfront work
- Requires inter-component communication
- Can have memory disadvantage
Managing Memory

- Smart layout of data will avoid cache misses
- Cache hits lead to massive performance gains
- Arrays are flat with fast access
- Vectors allow for flexibility in array size
Restructuring the Engine Loop

- GameObjects contained in vector at game manager level
  
  ```cpp
class Engine {
  std::vector<GameObject>

  void update();
};
```

- GameObject components stored in vector within component’s domain:
  
  ```cpp
class Physics : public System {
  std::vector<PhysicsComponent>;

  void update();
};
```
Handle-based Relationships

- Index into system’s component array using a handle

```c
struct GameObject {
    unsigned m_handle;
    unsigned m_components[eNumComponents];
};
```

- Lightweight bookkeeping
- Can expand system to translate handles into pointers
  - Pointers handled on the fly
  - Prevents dangling pointers
Hybrid Solutions Possible

- Pure component-based systems might be over-engineering
- Always design for the problem
- Usual software principles:
  - Take time to plan before writing code
  - If a system is difficult to conceptualize, the current approach might be wrong
  - Leave time to rework existing code
Last Thoughts on Data Structures

• There is no one correct solution

• Individual preference is a good place to start…
  • But be flexible and adapt to the problem

• Don’t over-engineer or prematurely optimize…
  • But keep data storage and caching in mind

• Try different approaches
• Take multiple passes to refactor
References


- Randy Gaul. Component Based Engine Design <http://www.randygaul.net/2013/05/20/component-based-engine-design/>