COMP 8851

Borna Noureddin, Ph.D.

Entity Component Systems

Object-oriented games

- Everything (player, enemy, tank, bullet, light, sound effect, etc) defined by its own class
- Often great deal of similarity between some classes: inheritance (eg, Tank and Car can derive from Vehicle class)
- Intuitive and relatively effective
- Can add Truck by inheriting from Vehicle
- Vehicle might inherit from PhysicalObject, as can Projectile

Object-oriented games

- Design requires detailed design of class hierarchy
- Hierarchy of increasingly abstract classes
- If hierarchy planned out well before implementation, can build large, complex game leveraging inheritance
- But, the deeper the class hierarchy, the more brittle and fragile it becomes:
  - Requirements change after implementation begins
  - Need to add/remove functionality to some abstract classes
  - Changes will affect all subclasses (even if they don't actually need changes)
  - End up with messy code additions pushed up towards root

Entity component games

- Core principles of good software design is modularity
- Many benefits of modularity, including flexibility
- Can replace one piece without having to change everything, especially if rest of system talks to it the same way (i.e., new replacement piece conforms to same interface)

Entity component games

- Entity represents concrete “thing” (e.g., Tank)
  - has no Tank-specific logic
  - actually barely any logic at all (really just an ID)
  - Real magic: the components it contains
- Component is a module of functionality (attribute?)
  - things that Entities have (e.g., Position, Velocity, RenderableMesh, PhysicalBody, etc.)
- Entity little more than bag of Components
- Entity has no explicit knowledge of what parts it contains
- => All entities can be treated the same way by the rest of the game

Entity component games

- Possible because components take care of themselves, regardless of which entity they belong to
- Example:
  - RenderableMesh component contains capability to render a 3D model
  - Model assigned to component
  - Component assigned to entity
  - Entity calls generic Draw() function on *all* its components, without needing to know what it does
  - RenderableMesh draws itself to display
- All the entity needs to do is call some generic update function on each of its components each frame, and each component will do its own thing

Entity component games - Pros

- Components are generic, perform single role, same way, regardless of parent entity
  - RenderableMesh of a Tank object would draw itself the same way as that of a Car object
  - Only difference: shape of mesh assigned to each component
- Different types of entities can be manufactured easily by plugging different reusable components into empty entity

Entity component games - Pros

- Great for maintaining flexibility during and after development
  - Changes to entities typically involve changing 1 or 2 components in isolation
  - No need to change any unrelated components or pollute other entities
  - New functionality can be added with independent addition of new components

Entity component games - Cons

- There are inherent relationships between components which requires them to be coupled in some way or other
- E.g., RenderableMesh knows how to render, but not where without consulting Position component

Entity component games - Cons

- Velocity component not much good without being able to update Position component
  - Possible solution: push ‘shared data’ up into the Entity itself
  - all components can read and write to position variable
  - leads to analogous problem to that of the traditional OO hierarchy: slippery slope
  - movement also needs to know how much health entity has
  - either communicate with Health component or push health up into entity
  - and so on….

Entity component games - Cons

- Another solution: allow components to hold references to each other and communicate directly
- Multiple drawbacks
  - couples components very tightly (difficult to reuse)
  - references need to be assigned when entity is created, but without entity's direct intervention (non-trivial problem)
  - could implement some elaborate system of runtime inter-component dependency resolution and injection: extremely complex

Entity component games - Cons

- Alternative: attach message dispatching system to each entity
- Allows components to fire events when something interesting happens, and handle other events of interest fired by other components
- Decouples components (nice), but comes at cost of increased complexity and persistent performance penalty

Entity component games - Cons

- Even if all components hooked up and working together, completely encapsulating logic related to each component fraught with danger of components becoming bloated with functionality
- Theoretically, Physics component should handle all physical interaction between its parent entity and rest of the world, but does the knowledge of the rest of the world really belong in a component?
- Maybe split out physics calculations into a centralized physics manager: which components have logic and which don't, and how much?