Architectural Patterns for Games

Pipe and Filter

This pattern connects a sequence of components (filters), each of which processes data and passes it along a pipeline to the next. It's typically used in systems with linear data processing stages. Each filter performs a clearly defined task, and filters are easy to replace or extend. In games, this can be used to build the game loop as a pipeline — reading input, handling collisions, updating AI, then rendering.
It works best when your system can be broken down into **independent processing steps** that follow one another.

Layered Architecture

The system is split into layers of abstraction, where each layer only interacts with the one directly below it. Higher layers build on lower ones, which enables incremental development. In game AI, for example, a layered approach might divide movement into obstacle avoidance, local exploration, and global strategy. Each layer adds more complexity and refines the behavior.
Use this when the problem can be naturally decomposed into **increasing levels of abstraction** or **incremental refinement**.

Blackboard

A blackboard is a shared memory space where independent components (often called experts) read and write information. Each expert posts its findings to the blackboard and reacts when certain data becomes available. An arbiter often controls access to avoid conflict. In games, this is useful for coordinating AI behaviors where agents contribute ideas or observations, like multiple strategies for a tank in a battlefield game.
Pick this when multiple modules need to **collaborate without direct communication**, and **react to shared information** dynamically.

Hierarchical Task Trees

Game behavior is structured as a tree of tasks where higher-level tasks decompose into lower-level ones. Tasks can be scheduled, aborted, or retried, and dependencies between them are explicit. This makes it easier to represent complex objectives like "attack the castle" as a series of ordered sub-goals (e.g. break wall, kill archers, storm gate).
Use this when your game includes **goal-driven AI or scripted sequences** that can be broken down into subtasks.

Entity-Component System (ECS)

Instead of using deep class hierarchies, ECS promotes composition. An entity is just a unique ID with a collection of components (e.g., health, position, behavior). Systems operate over entities with specific components, enabling flexible and dynamic behavior. This allows behaviors to be added, changed, or removed at runtime. It avoids the rigidity of traditional inheritance and simplifies managing different types of game objects.
Choose ECS when your game involves **many types of entities with shared features** and you want **maximum flexibility and runtime control**.