Component Entity System

Table of Fucking Contents

[Overview 2](#_Toc302347954)

[Concrete Example System 2](#_Toc302347955)

[Advantages 3](#_Toc302347956)

[Possible Issues 4](#_Toc302347957)

[Technical Stuff 5](#_Toc302347958)

[Managers and Systems 6](#_Toc302347959)

[EntityManager 6](#_Toc302347960)

[PlayerManager 6](#_Toc302347961)

[EnemyManager 6](#_Toc302347962)

[InputManager 6](#_Toc302347963)

[PhysicsManager 6](#_Toc302347964)

[SpriteManager 6](#_Toc302347965)

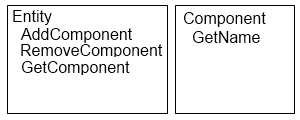
[SoundManager 6](#_Toc302347966)

[EnemySpawner 6](#_Toc302347967)

[References 7](#_Toc302347968)

# Overview

OOP isn’t very good at implementing the “global” parts of a game that operate on everything or have to be invoked from everywhere. A Component Entity System (CES) can deal with these systems and does so outside the realm of Entities and Components. Each system runs continuously and performs global operations on every Entity that contains a Component of the same aspect as that system. Entities represent all the unique, different “things” in your game world and have no data or methods in themselves. What do they do then? They contain a unique identifier and Components. Components are used to label an Entity as possessing a particular aspect, property, or behavior. So, there are no such things as trucks and goblins, only Components that enable their respective behaviors. Generally, you want to start with isolated components, and only skew toward coupling them to other components as is needed for component interactions and as you note performance problems (through profiling).



The Entity is simply a container for Components.



Entities can share Components, have different ones for the same purpose, and contain Components that other Entities do not.

## Concrete Example System

The game update cycle goes something like this: your world manager says "update the world", and that update the world function likely has some specific order in which it updates components - physics needs to come before rendering, and so on - and for each class of component, it gathers a list of all entities with that component, and says "update your physics." So the entities run Update() on their list of components of that type. Not everything will necessarily update that way - rendering probably keeps track of meshes directly and renders them directly, with rendering components just updating their mesh's information during their Update() step, and the actual physics simulation portion of physics would do likewise - but much will.

You have a mobile animated creature. It has a skinned mesh component, and a mobile physics component, and a movement AI component. The AI component calculates a movement vector and puts it into the entity's data store, the mobile physics component takes the movement vector data from the entity and physically moves the entity along (and adds current velocity data to the entity), and the skinned mesh component determines if the object is moving based on velocity and either does a walking animation or a standing animation depending.

You have a second entity that is a trigger physics component and an explosion AI component. The trigger physics component defines a bound you can walk through; it just waits for a collision, which sends a message to the attached-to entity when anything collides with it. The explosion AI component waits until it gets a collision message, waits 5 seconds, and then does a collection (by asking the entity manager) for any nearby entities - and to each, sends a physical impulse message.

The player, walking in the circle, eventually enters the explosion region, and then 5 seconds later gets the physical impulse message. Its render and AI components ignore it, but the physics component recognizes that message, and applies the requested physical impulse - launching the player up and away. The player lands, and then continues walking in a circle as defined by the other two Components.

## Advantages

* Better encapsulation of functionality (just like methods should “do one thing and do it well”)
* It’s very, very extensible. Any Entity can do anything; just add the Component for it. This makes it almost like a dynamic inheritance tree.
  + Avoids deep chains like: Entity > DrawableEntity > PhysicsEntity > MoveableEntity > Player > Lumberjack
* CES can result in far fewer classes written because functionality is shared
* You don’t need to have full knowledge of the system or all its capabilities before coding it.
* No need to update *everything* related to an entity at the same time
  + Can separate logic, rendering, physics, etc. into update **phases**
* Components are isolated, so they can be tested individually
* Easy to add or remove behaviors to or from an Entity at runtime
* Spatial locality of reference: if the system is implemented in a property-centric manner, all Components of a certain type are in contiguous memory, which greatly increases access time when iterating through them
* Useful for visual editors; can easily add Components to world objects
* In a traditional hierarchy using inheritance, you often find yourself needing to use multiple inheritance (possibly leading to DDoD) or duplicating large chunks of code (with slight variations) to give two different objects the behavior they need. Components are reusable and negate the need to fall back on these bad practices.
* A traditional object model will bloat your objects with lots of data and functionality that all of them may not need or ever use. With components, each Entity only contains what it *needs*.
* Consider a situation where you have 100 tigers on the playing field. In traditional OOP each of those 100 Tiger objects has a copy of each method that can be invoked on that unit, though most compilers use optimization to share those methods behind the scenes and avoid wasting memory. In a CES, there are no unique or even shared methods, as Entities and Components do not contain any. Instead there is an external system for each aspect that contains all the methods for any Entity with a Component that marks it as using that system.

## Possible Issues

* Inter-component interactions and **coupling**
  + Needs to be fast
  + Can simply get a pointer to another component (eats up the CPU)
  + Components can store references to other Components they need to access, allowing them to call each other’s member functions directly
  + Can use events to perform tasks or retrieve information, but this may lead to synchronization issues
* Making components so large or specific they cannot be reused by other entities
* Components have such small functionality that the **overhead** of managing and processing them is greater than the cost of the features they provide
* Circular dependencies: data should flow from higher to lower levels
  + Example: Physics should not talk directly to the A.I.
* Objects may get out of sync with each other
  + You need to enforce the order in which Components are added or updated
* Duplication of state for certain pairs of Components like Physics and Rendering
* Overkill for simpler games
* Multiplayer is complex
  + All game object creation can be handled on the server, which then tells the clients that need to know about game objects to create them as well. If two players are on opposite ends of the map (not sharing a screen), they don’t need to know about the objects that the other player is seeing.

# Technical Stuff

Entities are just containers for components and have a uniqueID.

Each entity has a list of components that can be found by type when needed:

- this.GetComponent<ComponentType>();

- <http://en.wikipedia.org/wiki/C_Sharp_4.0#Dynamic_member_lookup>

Each entity should hold at most one component of each type.

What to do if all your enemy units are the same? What would be the best way to create an new instance of this object without always redefining the parameters?

* Use the Flyweight Pattern: <http://c2.com/cgi/wiki?FlyweightPattern>
* Or … use the prototype design pattern: <http://en.wikipedia.org/wiki/Prototype_pattern>
  + Basically just use clone() and set parameters

Store pools per Component type and iterate each type independently to perform updates.

How do you create the Entities with Components on them?

* Parse XML files defining things like: Player, Enemy, PhysicsRules, etc.

<entity name="TrapTrigger">

<component name="physics" width=200 height=100 /component>

<component name="logic" timer=2 damage=50 /component>

</entity>

* OR… initialize them at compile time when loading the game

# Managers and Systems

The basic idea for systems is that certain Components identify their parent Entities as subscribing to certain systems. So, each system continuously runs its own internal methods against those Components one at a time. For example, the Rendering system wakes up every 16 milliseconds, renders every Entity that has a RenderableComponent, and goes back to sleep. Generally, you only add a new Component type because you’ve added a new system that needs to store/attach data to the Entities. Also, only one Component of any given type should be attached to each Entity.

## GameRules

Handles code related to gameplay: chance to hit, applying damage, awarding XP, etc.

This system can also handle victory conditions and other major gameplay events.

## EntityManager

-

## PlayerManager

-

## EnemyManager

-

## Camera

-

## InputManager

Constantly polls the input device (mouse/keyboard or gamepad), sets internal flags accordingly, and changes the state of all Entities with the InputComponent.

## PhysicsManager

Handles collisions between Entities, explosions, that sort of thing.

## SpriteManager

Handles drawing sprites to the screen and animating them.

The Animation system constantly watches for anything that would trigger a new animation, or cause an existing animation to change (e.g. player changes direction mid-step), and updates the data of each affected Entity (i.e. those with the AnimatableComponent).

The only difference between Entities with the related Component is the argument they pass in when attaching the Component, namely the animation/texture filename.

## SoundManager

-

## EnemySpawner

-

## HUD

Draws the main display (static stuff on the screen) and screen overlays.

# References

Component-based Software Engineering. <http://en.wikipedia.org/wiki/Component-based_software_engineering>

The Game Entity – Part III, Components. Carr-Robb-John, Michael A. <http://altdevblogaday.com/2011/08/10/the-game-entity-%E2%80%93-part-iii-components/>

Game Engines 101: The Entity/Component Model. Fox, Megan. <http://www.gamasutra.com/blogs/MeganFox/20101208/6590/Game_Engines_101_The_EntityComponent_Model.php>

Entity Systems are the Future of MMOG Development. Martin, Adam. <http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/>

Evolve Your Hierarchy. West, Mick. <http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/>

Component Based Entity System Design. <http://www.gamedev.net/topic/577635-component-based-entity-system-design---phase-2-post-30/>

Adopting a Component Model for Game Development. Schearer, Alex. <http://www.anotherearlymorning.com/2010/09/adopting-a-component-model-for-game-development/>

Introduction to Dungeon Siege Architecture – Game Objects. <http://garage.gaspowered.com/?q=su_301#gos>

Prefer Composition over Inheritance. Itterheim, Steffen. <http://www.learn-cocos2d.com/2010/06/prefer-composition-inheritance/>

Game Object Structure: Inheritance vs. Aggregation. Wilson, Kyle. <http://www.gamearchitect.net/Articles/GameObjects1.html>

Game Programming Patterns: Component. Nystrom, Robert. <http://gameprogrammingpatterns.com/component.html>