# Entity-Component System (ECS), Perfect for 3D Development

## What is ECS

ECS architecture is a common and desirable pattern in 3D and game development that follows the **composition over inheritance and hierarchy** principle.

The benefits of ECS include:

1. Greater flexibility when defining objects by mixing and matching reusable parts.
2. Eliminates the problems of long inheritance chains with complex interwoven functionality.
3. Promotes clean design via decoupling, encapsulation, modularization, reusability.
4. Most scalable way to build a VR application in terms of complexity.
5. Proven architecture for 3D and VR development.
6. Allows for extending new features (possibly sharing them as community components).

On the 2D Web, we lay out elements that have fixed behavior in a hierarchy. 3D and VR is different; there are infinite types of possible objects that have unbounded behavior. ECS provides a manageable pattern to construct types of objects.

Below are great introductory materials to ECS architecture.

* [*Entity-component-system* on Wikipedia](https://en.wikipedia.org/wiki/Entity%E2%80%93component%E2%80%93system)
* [*What is an Entity System?* by Adam Martin](http://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/)
* [*Decoupling Patterns — Component* on Game Programming Patterns](http://gameprogrammingpatterns.com/component.html)
* [*Evolve Your Hierarchy* by Mick West](http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/)

A well-known game engine implementing ECS is Unity.

## Concept

A basic definition of ECS involves:

* [**Entities**](https://aframe.io/docs/1.3.0/core/entity.html) are container objects into which components can be attached. Entities are the base of all objects in the scene. Without components, entities neither do nor render anything, like empty **<div>**s.
* [**Components**](https://aframe.io/docs/1.3.0/core/component.html) are reusable modules or data containers that can be attached to entities to provide appearance, behavior, and/or functionality. Components are like plug-and-play for objects. All logic is implemented through components, and we define different types of objects by mixing, matching, and configuring components. Like alchemy!
* [**Systems**](https://aframe.io/docs/1.3.0/core/systems.html) provide global scope, management, and services for classes of components. Systems are often optional, but we can use them to separate logic and data; systems handle the logic, components act as data containers.

## Example

Some abstract examples of different types of entities built from composing together different components:

* **Box = Position + Geometry + Material**
* **Light Bulb = Position + Light + Geometry + Material + Shadow**
* **Sign = Position + Geometry + Material + Text**
* **VR Controller = Position + Rotation + Input + Model + Grab + Gestures**
* **Ball = Position + Velocity + Physics + Geometry + Material**
* **Player = Position + Camera + Input + Avatar + Identity**

As another abstract example, imagine we want to build a car entity by assembling components:

* We can attach a **material** component that has properties such as “colour” or “shininess” that affects the appearance of the car.
* We can attach an **engine** component that has properties such as “horsepower” or “weight” that affects the functionality of the car.
* We can attach a **tire** component that has properties such as “number of tires” or “steering angle” that affects the behavior of the car.

So, we can create different types of cars by varying the properties of the **material**, **engine**, and **tire** component. The **material**, **engine**, and **tire** components don’t have to know about each other and can even be used in isolation for other cases. We could mix and match them to create even different types of vehicles:

* To create a *boat* entity: remove the **tire** component.
* To create a *motorcycle* entity: change **tire** component’s number of tires to 2, configure the **engine** component to be smaller.
* To create an *airplane* entity: attach **wing** and **jet** components.

Contrast this to traditional inheritance where if we wanted to extend an object, we would have to create a large class that tries to handle everything or an inheritance chain.