ECS Architecture for Modern Military Simulators

Joseph A. Vagedes\textsuperscript{1}, Douglas D. Hodson\textsuperscript{2}, Scott L. Nykl\textsuperscript{3}, Jeremy R. Millar\textsuperscript{4}

Air Force Institute of Technology, WPAFB, OH, USA\textsuperscript{1,2,3}
Bellevue University, Bellevue NE, USA\textsuperscript{4}

emails: joey.vagedes@afit.edu\textsuperscript{1}, douglas.hodson@afit.edu\textsuperscript{2}, scott.nykl@afit.edu\textsuperscript{3}, jemillar@bellevue.edu\textsuperscript{4}

Abstract—This paper introduces the concepts associated with the Entity Component System (ECS) architecture and its current applications. It considers possible use cases in modern military simulators and provides a theoretical implementation of the ECS architectural pattern in a realistic scenario.

Keywords: entity component real-time, model, simulation

1. Introduction

The Entity Component System (ECS) architecture is a software architecture commonly used in game development, being spearheaded by popular game engines such as Unity and Unreal Engine. This architecture focuses on cache optimization’s to reduce cache misses [1], automatic parallelization done by the architecture [2], and highly decoupled code [1] [3]. These optimization’s not only increase the computational performance (allowing for more work to be done per computational cycle [a frame], or more frames per second), but also make implementing and maintaining the code base simple and quick compared to other architectures.

As most games developed on these platforms tend to be real-time systems, one can begin to see many similarities between video games and modern real-time military simulators. In addition to both being real-time systems, other similarities include the ability to take in user input to manipulate a 2D or 3D world.

A key difference however, is how the simulation is displayed to the user. In almost all game engines, there is an on-board graphics renderer provided by the architecture that is used for visualization. This may be true for some real-time military simulators, but for most, the visualization tool is a separate system. The two systems communicate via a network, mainly packets of data being sent to the dedicated renderer, generally by the User Datagram Protocol. This is a huge advantage for military simulators as rendering a display consumes a lot of processor time. By being able to delegate this to a different system, the processor can use that time elsewhere.

2. Background

The ECS architecture focuses on data-oriented programming rather than object-oriented programming. Instead of objects inheriting data and functionality from a parent object (object-oriented programming), the object will be composed of data (data-oriented programming) [4]. This is considered the “Component” in the ECS architecture while the object is now the “Entity”. Finally, the functionality becomes the “System”. Each will be explained in-depth below.

3. Entity

In some ECS architectures, the entity is an instance of the class that contains the components (remember, it is composed of components, it does not inherit them). Not only does this technique make entities large and slow to move around, but it also decreases performance. This is because the system holds an array of all entities that must be iterated over even if it does not have the required component. This makes it hard to parallelize as the software does not know what components each entity contains; thus, it cannot delegate workloads to threads without a complex system in place to handle data races.

For more advanced (and current) ECS architectures, the components are taken out of the class and the entity becomes a single unique identifier used to locate that entity’s component [5]. Generally, this unique identifier is equal to its location in the world list (e.g. if it was the first object added to the list, its unique identifier will be 0).

4. Component

As stated previously, components are the data pertaining to entities. Some common components are position, rotation, and size. Components are generally “structures/ classes” that contain multiple pieces of data that commonly work together (no functionality). It is possible to split “struct/ class” such that each data piece is its own component (e.g. split the position component into a x, y, and z component). Doing this is a trade off as it will hurt readability and understandability of the code, but it could potentially increase performance. This is because the algorithm responsible for parallelizing systems may find a more optimal way to schedule each system.

As an example, imagine a system that only changes the y position. If the component contains x, y, and z position, then all three are locked when that system runs. If a different system wanted to read the x position, it would have to wait until the first system finishes. If you split the position component into three separate components, then one system
can write to the the x position component at the same time as another system is reading y position component.

In current implementations, components are no longer in a class; they have been moved into a list containing all copies of that particular component. That list is managed by a world manager [5], [4]. Their location in the list is equivalent to their entity’s unique identifier. It is important to note that the size of each list is always equal to the amount of entities in the simulation, regardless of if every entity has that component or not. This is so that the components location in the list is always equal to the unique identifier of the entity. However, if an entity does not have a particular component, then the entry in that component array is null. Another important fact is that each entry in the list will be immediately next to each other in memory. This is what helps lower cache misses.

It is best to keep the size of each component as small as possible as the Central Processing Unit (CPU) will be able to iterate through more entries in the array before coming to a cache miss. It is vital to realize that it is not a list of pointers to components, but a list of the components themselves. If it were a list of pointers to each component, the worst case scenario could be a cache miss every time a component is accessed as the memory location of the pointer will not be near the memory location of the component that said pointer is pointing at.

5. System

Now that data (component) storage is handled for each entity, the manipulation of these components must be considered. This is done through systems. There are two parts to this; entities interacting with themselves, and entities interacting with other entities. The first is highly optimized by the ECS architecture, however the latter is much harder to implement, and must be carefully coded to ensure top performance. Only the first part will be considered, as implementing the latter is highly dependent on what is trying to be done, and there is no standard way to do it.

A system is defined very similar to a function, the only exception being that it will be ran on all entities that have the required components. Think of this as a loop that iterates over any entity that meets the requirements of the system, then runs the system on it. When creating a system, the first step is to define the components that are needed as the inputs (anything being read or written). Next, the functionality of the system is created by defining the interactions between the components. It is important to remember that the system only has access to a single instance of a component at any given time (e.g. during the first iteration of a system, the system only has access to pos_component.at(0) and cannot access po_component.at(1) or any other entry in the component list [6].

Consider a system that changes an entities location. For this, a system is defined that requests write access to the position component and read access to the velocity component. The internals of the system are coded such that it reads the velocity and adds it to the current position value (based on one time-unit of change). This system is then pushed to the game manager. During simulation execution, on every game loop, the system will be run on all entities that have both a location and velocity component and incrementally change that entities position.

It starts to become obvious as to why this architecture reduces cache misses and stalls, greatly increasing performance. This is because each system can be run quickly and efficiently over all entities (that have the required components) as all the necessary data (for all entities, not just one) is pulled into the low-level cache when the system first begins and is able to be iterated over without cache misses.

Additionally, algorithms can be developed that look at all systems in the simulation and determine what systems can be parallelized based off of the requested read/ write access of components for each system.

Consider three systems: system one reads velocity and writes position, system two reads position and writes to a collision component, and system three reads position and writes to a current city component. It can be seen that system two and three both rely on system one updating the position, however system two and three do not rely on each other. This means that system one can run on a thread; once it finishes, system two and system three can be ran on separate threads simultaneously. This greatly increases performance; with even more systems running in parallel, this architecture could potentially perform a lot of processing in a short amount of time [7].

6. Game Loop

The game loop is a loop that will run continuously until the simulation/game is terminated; it is responsible for responding to user input, executing all systems, and rendering the screen. However, since most modern-day military simulators are not responsible for rendering the screen, that can be replaced with sending data packets to the secondary system responsible for rendering. It is important to note that these loops are generally considered separate from an actual ECS design, though they are almost always used in conjunction.

A common misconception is that the game loop is a simple “while loop” that waits for a Boolean termination signal (in some instances, it can be). However, most of the time this is not true. Using a simple loop as stated above, the amount of time that passes per loop is non-deterministic as it is dependent on how much processing is being done that loop. If any of the systems are dependent on time passing at a steady pace (e.g. 60 cycles/ frames a second), then time would randomly speed up and slow down depending on how much work the processor was doing and would lead to a
non-real-time simulation [8]. [A real-life example of this is Space Invaders: The enemies speeding up as you progress in game is a bi-product of having to render less entities on the screen and was not intentional.] Now if the simulation only cared about the end product, then that would be fine, but for real-time simulations, that will not work. A final issue with this design is that it will cause a CPU to run at 100% usage the entire time the simulation is running.

This means the game loop must become slightly more complicated. There are many ways to implement a game loop, and it is a decision that must be made early as it affects how the game logic is written. Common game loops focus on setting a max frame rate that should be obtainable on most systems, then each time step is an assumed value (1 second divided by the number of frames). A disadvantage to this is that if the frame rate is set too high, or the hardware is old, users may find that time slows down as the CPU cannot process that much data [8] Another possibility is that the game loop will actually calculate how much time has passed since the last frame and feed that time to the update functions (or the systems for an ECS architecture) [1]. The issue with this implementation is that it complicates programming time-dependent systems as code must be robust enough to calculate its result based on a certain time, instead of being able to assume that the next frame is a specific increase from the last frame.

As an example, implementing functionality such as “pressing the left arrow moves the character left” would be as simple as “If the left arrow is pressed, move left by one unit” in the first game loop implementation. This is because you know time will move the same every frame. If the game loop is implemented the second way, it becomes more complicated to create the functionality. This is because a new component that stores the previous loops current time must be created. Then in the system, that time is subtracted from the actual current time, then multiplied by some velocity.

There are many other implementations of game loops out there, each with its own pros and cons. However, One thing that should almost always be done on a separate thread is the rendering (or in this case, sending data packets to the renderer) as these actions are extremely slow [9]. Delegating it to a different thread will free up more computational time that can be devoted to user input and system execution. It is common to see ECS architecture’s only implement a game loop that keeps a consistent number of frames per second. This is because it gives the developer the frame-work the ability to either use the internal game loop, or create a time component, that is then updated at the beginning of every loop by a system and is then read by other systems. This way it does not force the developer to use time-based changes for all systems.

7. Putting it All Together

As a recap, the ECS architecture causes the developer to change their programming style from object-oriented programming to data-oriented programming. Rather than inheriting functionality from a class hierarchy, that functionality is turned into components (the data) and systems (the functionality) and moved out of the class. The class itself simply becomes a unique identifier (generally an integer) and is referred to as the entity.

Overall the world manager handles the entities, their components, and the systems. The world manager has an entity list and a list for each component that exists in the simulation. The manager is responsible for determining what systems can be run in parallel and for dispatching the systems to threads. The game loop within the game manager is split into two different loops. The first is responsible for handling user input and dispatching systems while the second is responsible for sending packets to the separate rendering system. Generally, the first system will be able to run much faster than the latter due to the slowness of creating and sending packets. The main loop has a set number of frames per second, allowing the developer to assume a constant time-step per game loop and is not forced to use true time-based calculations unless wanted.

8. Advantages and Disadvantage

Many of the advantages of using the ECS architecture have already been described, but to recap, the advantages are: Automatic parallelization, reduced cache misses/ page faults, and highly decoupled code. The first two greatly increase performance as a general developer does not have the knowledge, expertise, or time required to write safe, parallelizable code that also optimizes data storage in a way that the CPU can take advantage. By having this done automatically by the architecture, it takes a significant workload away from the developer, who may not have even implemented it optimally. This architecture forces the programmer to write highly decoupled code; it creates a code base that is easy to implement, maintain, and update. The developer can easily implement additional features by creating additional components, attaching them to the entity, then create a system that interacts with these components. Since the previous code is completely decoupled, no modifications to existing software will be needed.

As for disadvantages, the first is that most developers are in the mindset of data-oriented programming, or “composition over inheritance” [4]. This is a completely different mindset that will take time for the developer to learn and to use properly. Most programmers do not focus on performance and thus have been taught to use object-oriented programming as it makes code readability and understandability much greater.

The greatest disadvantage to the ECS architecture is that communication between multiple entities can become very
difficult. The most common example has to do with physics. Two entities are in the same location, but how do they know they must bounce off of each other? A system can only know about one particular entity at a time. There are multiple solutions such as creating a messaging system for entities, but it is not currently handled by the ECS architecture and must be developed by the user [1].

9. Application

With this background in the ECS architecture, a concrete hypothetical implementation of a radar system using this architecture will be explained. The first thing to consider are the entities needed for a radar simulation. For this example, a radar and any missiles that the radar can detect will be needed; for simplicity sake, assume there are two missiles. Next, the components for each entity will need to be considered. The radar will need a sensor range, sensor rotation, detection angle, and position; the first three can be turned into a single sensor component and third will be a single component. The missiles will have a position, rotation, and state component (a Boolean that is set if the missile is detected). Below are the components:

<table>
<thead>
<tr>
<th>Component</th>
<th>Sensor</th>
<th>Missile</th>
<th>Missile</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>sensor</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rotation</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>isDetected</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

The final step is to define the systems that will modify these components. The first system defined is responsible for changing the position/rotation of a missile. This system will need write access to rotation and position (technically it needs both read and write access, but this is covered by just write access). For simplicity sake, imagine the system contains an equation that inputs a position and rotation, and outputs a new position and rotation. These new values are then saved in their respective components. The missiles need a second system that will do something if the state is set to detected. It only needs read access to the state variable. The system is simple, it merely checks to see if the missile is in the detected state; if it is, it will write to the console stating its detected.

Next, systems for the radar will be defined. This will begin to get more complicated as there will be some entity interactions. The first system will be responsible for sensor rotation. This will only need write access to the sensor component. It will rotate the sensor by 5 degrees every time it is called. The system could be expanded to include the missile detection, but for the sake of decoupling code, it will remain a second system. This system will only need read access to sensor. Within the system, a game manager function will be used that will return all entities. The system will iterate through the entity list, checking if the state component exists (only missiles have this component).

For entities where this is true, it will check the current position and determine if it is within the sensor’s range and current rotation angle. If it is, it will access the state component and set the value to true. Let’s do a recap of each system:

<table>
<thead>
<tr>
<th>Component</th>
<th>Pos</th>
<th>Rot</th>
<th>isDet</th>
<th>Sensor</th>
</tr>
</thead>
<tbody>
<tr>
<td>System 1</td>
<td>W</td>
<td>W</td>
<td></td>
<td>W</td>
</tr>
<tr>
<td>System 2</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>System 3</td>
<td></td>
<td>R</td>
<td></td>
<td>R</td>
</tr>
<tr>
<td>System 4</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Now that all entities, components, and systems are created, they are pushed into the world manager. The world manager stores the entities in an array. The array will be EntityList[Sensor, Missile, Missile]. Their unique_id is their location in the list. The components are then stored in separate lists as follows, where E means the component exists, and () means it does not:

<table>
<thead>
<tr>
<th>EntityList</th>
<th>Sensor</th>
<th>Missile</th>
<th>Missile</th>
</tr>
</thead>
<tbody>
<tr>
<td>posList</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>sensorList</td>
<td>T</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>rotationList</td>
<td>()</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>detectedList</td>
<td>()</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Next, an algorithm would determine what systems can be parallelized, but will be completed manually to see how the decisions are made. From the table above, System one (missile position update), System two (isDetected lookup), and System three (Sensor rotation change) can be run in parallel. This is due to the fact that none of these systems request write access to the same component or requests read access while another has requested write access. Those three systems will be pushed to three different threads to be worked on simultaneously. The game manager will then push system 4 to the same thread as system 3. This is because system 4 relies on the data being written by system 3, and by putting them in the same thread, it guarantees that system 3 will finish before system 4 can start.

Once all systems have completed, the architecture will take a “snapshot” of the current state, and on a separate thread, it will begin sending data packets to the rendering system. In parallel with rendering, the next loop will begin and start getting user input (and running systems once user input is handled). Below is a diagram for what is happening each loop iteration. It is important to note that the systems being run in “system execution” are also being ran in parallel where possible. The program will continue the loop sequence as seen in Figure 1 until termination.

10. Final Thoughts

Although the ECS architecture is fairly well defined in this paper, it is only a single hypothetical implementation in an area that is still being actively developed. A multitude of implementations for this architecture exist in many
different programming languages, all with slightly different approaches. This paper sought to define many of the terms and specific programming patterns that exist throughout most versions of this architecture while also presenting an implementation that the author believes takes the best of many different implementations and combines them into one.

The concept of defining which components (and read/write access) are needed for each system outside of the actual implementation of the system opens many doors for automated parallelization through existing, or new, algorithms who have been designed for optimal scheduling which could lead to even greater performance gains when using the ECS architecture.

References


Author Biographies

JOSEPH VAGEDES is a graduate student attending the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio USA currently pursuing his master’s in Computer Engineering. He received his B.S in Computer Engineering from the University of Cincinnati in 2018. His research interests include 3D interactive systems for modeling and simulation, software architecture (simplification and automation of thread-level parallelization), and software optimization for current hardware architectures.

DOUGLAS D. HODSON is a Associate Professor of Computer Engineering at the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio USA. He received a B.S. in Physics from Wright State University in 1985, and both an M.S. in Electro-Optics in 1987 and an M.B.A. in 1999 from the University of Dayton. He completed his Ph.D. at the AFIT in 2009. His research interests include computer engineering, software engineering, real-time distributed simulation, and quantum communications. He is also a DAGSI scholar and a member of Tau Beta Pi.

SCOTT L. NYKL is an Assistant Professor of Computer Science at the Air Force Institute of Technology. His areas of interest are Real Time 3D Computer Graphics, computer vision, sensor fusion, parallel processing, Interactive Virtual Worlds, and computer networking. Throughout his graduate studies, he authored a 3D Visualization Engine and created avionics-related visualizations earning several national awards including mention in Forbes’s “The Greatest Young Inventors In America”. He was a National Collegiate Inventor’s Competition finalist as well as the Institution of Navigation (ION) graduate student award recipient.

JEREMY R. MILLAR is an Adjunct Assistant Professor of Computer Science at the Air Force Institute of Technology (AFIT), an Adjunct Professor of Cybersecurity at Bellevue University, and an active duty cyber operations officer stationed at Offutt AFB, NE. He received a B.S. in Computer Science from the University of Tennessee in 2000, an M.S. in Computer Science from AFIT in 2009, and a Ph.D. from AFIT in 2017. His research interests include distributed systems, virtual environments, and machine learning.