Exploring Game Industry Technological Solutions to Simulate Large-Scale Autonomous Entities within a Virtual Battlespace

Kyle McCullough, Raymond New, Noah Nam, Ryan McAlinden

USC - Institute of Creative Technologies

Los Angeles, CA

{mccullough, new, nam, mcalinden}@ict.usc.edu

ABSTRACT

In support of Synthetic Training Environment (STE) goals of a virtual battlespace, we explored experimental technologies in the commercial game industry to determine the feasibility of running one million autonomous entities while maintaining performant simulation and fidelity. In conventional software architectural methods, the number of entities represented do not exceed hundreds (or at most a few thousand) due to the expense of simulating relatively complex behaviors for each entity. Even when using group behaviors to provide the illusion of large quantities of simulated characters, these simulations fail to provide behavioral fidelity at a higher level. To tackle this problem, the Unity game engine was chosen to simulate this virtual battlespace, and specifically to explore the Unity-specific Entity Component System (ECS), Job System and Burst Compiler experimental technologies. Unlike traditional methods of Artificial Intelligence (AI) representation, the ECS uses a data-driven approach (not object-oriented one) to represent large numbers of identically-structured data. While it is an unfamiliar departure from usual game object representation, its format enforces low-level contiguous storage of the data in memory, dramatically decreasing in-cache misses and preventing loss of data representation compared to random location storage. This results in far quicker iteration and access through large numbers of data blocks in memory. Also, the Unity Job System allowed for better distribution of the computational workload among multi-core processors, while the Burst Compiler improved the performance of the overall system by translating the .NET C# code into highly optimized native code. The work has shown promising results with its current implementation allowing up to 150,000 autonomous, simple-behavior entities to be simulated while running at relatively performant standards (20-30 frames per second (FPS) on average). Further optimizations can still increase the number of entities, but eventually, we will consider using a distributed systems solution to simulate a million entities in this virtual battlespace.

ABOUT THE AUTHORS

Kyle McCullough is currently the lead Programmer at USC-ICT working on the One World Terrain project. Previously, he was a creative director and writer for the Video Games Industry, most recently for Ubisoft's 'Transference', winning *Best Interactive Narrative VR Experience* at Raindance 2018. His research work involves advanced prototype systems development, utilizing AI and 3D visualization to increase realism at all levels of large-scale dynamic simulation environments. He has a B.F.A. from New York University. Email: mccullough@ict.usc.edu

Raymond New is a senior programmer at the USC-ICT working on various projects including the One World Terrain. He acquired his BS. in Computer Science from the California State University Long Beach in 2007. He has been developing games and simulation software for the institute for over 12 years, including the Dark Networks game for the Naval Post-Graduate School which won the Silver Medal Award at a Serious Games competition in 2018. Email: new@ict.usc.edu

Noah Nam is currently a programmer at USC-ICT working on the One World Terrain project. His work involves implementing Unity's Entity Component System to enable simulating many entities within a large-scale dynamic simulation environment. He has a B.S. in Computer Science from Drexel University.
Email: nam@ict.usc.edu

Ryan McAlinden is the Associate Director for Digital Training and Instruction at USC-ICT. He rejoined ICT in 2013 after a three-year post as a senior scientist at the NATO Communications & Information Agency (NCIA) in The Hague, Netherlands. There he led the provision of operational analysis support to the International Security Assistance Force (ISAF) Headquarters in Kabul, Afghanistan. Prior to joining NCIA, Ryan worked as a computer scientist at USC-ICT from 2002 through 2009. He has a B.S. from Rutgers University and M.S. in computer science from USC. Email: mcalinden@ict.usc.edu

Exploring Game Industry Technological Solutions to Simulate Large-Scale Autonomous Entities within a Virtual Battlespace

Kyle McCullough, Raymond New, Noah Nam, Ryan McAlinden

USC - Institute of Creative Technologies

Los Angeles, CA

{mccullough,new,nam,mcalinden }@ict.usc.edu

INTRODUCTION

Utilizing existing and traditional methods of AI Agent creation and run-time deployment is an incredibly taxing and complicated process at the hardware level, especially in relation to the consumption of a computer's central processing unit (CPU) and memory resources. CPU and random-access memory (RAM), even on high-powered machines, are quickly overloaded and blocked when accounting for even the simplest scripted behaviors of Autonomous Agents. This can happen even when the total number of agents only reaches into the hundreds.

These traditional Object Oriented methods often require illusory solutions such as defining group behaviors to drive the entities at higher numbers, removing the simulatory benefit of individual fidelity, and therefore greatly decreasing the simulated realism. Even when utilizing methods like Group Behaviors, taxation on the hardware resources fails to fall within an acceptable performant range when the number of simulated entities reaches a few thousand at most.

This paper seeks to identify a working methodology and path towards a goal of high fidelity, complex, and fully autonomous individual AI Agents deployed at a global scale and in this case tested within virtual battle space environments. This means attempting to achieve one million dynamic and independent *living* entities in a real-world environment utilizing actual geo-specific terrain, weather, atmosphere, and other real-world parameters.

We chose to create a baseline utilizing a single desktop workstation, valued at under $5000 and suitable for 3D visualization. This allowed us to analyze our efforts based on common and accessible hardware, with the knowledge that we could scale up components to increase performance, even ultimately into the cloud for near unlimited scalability. We chose the game engine Unity, after exploring a number of software options, due to their new and experimental inclusion at their software core of a "Data Oriented Tech Stack" (DOTS), built up from the Entity Component System (ECS) architectural pattern, along with two additional features that support the system further, a Job System and Burst Compiler. All of which we will explore in detail later in this paper.

Though there is some research that delves deep into using ECS as a pattern of development within Unity, contrasting inheritance versus composition (Holopainen, 2016), there are no papers that single out increasing the number of Autonomous Agents, and there is nothing recent to make use of Unity's experimental foundation for ECS. There are a few hobbyist and commercial initiatives that have attempted to find limits for Autonomous Agents using traditional methods and implementing either Unity's built-in NavMesh solution (Brackeys, 2018) or, the method we used for this work, A* Pathfinding (Granberg, 2018) which is a common algorithm for determining point-to-point navigation in gaming and has a well-supported library for Unity. Though these initiatives have made an effort to test against the limitations of the engine and hardware they have not attempted to test against the concepts of optimization presented by the Unity DOTS and ECS Framework as we have sought to do in this paper.

Our previous traditional work towards achieving one million individually controlled entities as a target goal lead to significant roadblocks, though we were initially able to simulate around 1000 simple entities without the need for group behavior scripting. After optimizing this as far as possible utilizing object-oriented programming (OOP) methods we were able to almost double that achieving around 2000 Agents however this ran at only about 15-20 FPS.

Our current work utilizing the experimental Entity Component System has allowed us to simulate a few magnitudes more, currently around 150,000 Agents on a single machine, running at closer to 30 FPS without distributed

processing or intensive optimization, and utilizing a hybrid ECS architecture. Though we used Unity and some of these features are specific to the Unity Data Oriented Tech Stack, the broader concepts and optimizations found in the ECS architectural pattern would work in a number of runtimes.

AI IN THE COMMERCIAL VIDEO GAME INDUSTRY

Artificial Intelligence as a concept has been adrift in the mind of humanity from the days of antiquity. However, it was the dawn of computation and information theory, coupled with the numerous breakthroughs in neurology and cognitive sciences spread across the 1930-50's that unveiled the true potential of an electromechanical brain. A machine that would be able to make decisions, draw conclusions and ultimately even dream and create art. The scientists responsible for creating the numerous computational innovations of this time were certain they would be able to achieve the goals of what we now know as AGI, or Artificial General Intelligence, within a decade. Though once they began to direct their research efforts to the problem, even with many millions of dollars of funding, it was soon realized as far more difficult as they had anticipated. The complexities of Intelligence and Consciousness are still mysteries today, even as computing superpower and storage begins to near that of the human brain.

Video Gaming at its nascent stages, in the 1960s and 1970s would ultimately help to shape a new future for many of today's AI methods and technologies across such disparate fields as autonomous vehicles, factory data simulations, robotics, city planning, thermostats, and disaster scenario analysis.

At the origin of Video Games, competitive games requiring two players, quickly led towards the idea of a solo player, competing "against the computer" which gave rise to the need for a description of ever more complicated machine learning algorithms and higher fidelity of computer controlled competition. The rise of Arcade Games led to quick, iterative leaps in the AI patterns, with the games requiring increasingly difficult and more complex "enemies" to challenge the player. The Pac-Man series being a wonderful example wherein the original game was deterministic, where at any given time the state of the enemy agent was determined by distance from the player (Gallagher & Ryan, 2003), the latter Ms. Pac-Man was nondeterministic, where occasional randomization lead to the erratic and unpredictable positions and movements of the collectibles and enemies (Delooze & Viner, 2009).

Video game based Autonomous Intelligent Agents are more generally oriented towards the appearance of intelligence oftentimes utilizing assigned group behaviors in place of individual utility. This paradigm is the norm for Video Games and most explicit within scenario based simulations, due to technology, reality, and budget constraints. However many of the core components of these agents are found across a number of AI applications, especially those with discrete physical analogs such as pathfinding, approximating for missing information, decision trees, state machines, and in more advanced simulations abstractions such as learning, nerve response, and memory.

IBM's natural language AI, Watson, when applied to the game of *Jeopardy!* (Tesauro, Gondek, Lenchner, Fan, & Prager, 2013), is a good example of an AI that maps a number of human intelligence qualities to its neural networks, while at its core it is still bound by the rules and limits of a rather simple game. In *Jeopardy!* linguistic data is applied to known answers, with Watson working backward to determine potential questions and weighting their probability. Watson is capable of beating top human players at a game utilizing a human created language, though Watson, narrowly bound by that game, would not know how to react under fire at the front lines in a battlefield. Conversely, adding just one complex layer such as linguistic communication to a fully simulated battlefield AI entity requires an exponential increase in hardware and software resources. When the idea of actualized simulated warfare within a battlespace is involved, we need to be able to understand what it would take to simulate millions of independent high fidelity Agents within a world where the rules and options begin to trend towards infinity. This simultaneity is at the root of the problem presented and explored in our research and paper.

Traditional Methods of Creating Autonomous Agents

The classic method of incorporating AI entities follows directly in line with the common practice of Object Oriented Programming, in which components and scripts are predefined and attached to a Unity GameObject, which is a container within Unity for holding components. The GameObject's are then placed within a scene's hierarchy. This GameObject is generally saved as a Prefab, Unity's form of an "archetype", which allows you to store a single instance of a complete GameObject for reuse where any modification to the Prefab will affect all subsequent

instances of it. The prefab is then instantiated a given number of times depending on the need for agent presence. This methodology is common in Unity, both for gaming and using Unity as a more generalized simulation solution. There is a good bit of research as far as utilizing Unity purely for simulation, with researchers even creating toolkits to take further advantage of Unity's platform (Juliani et al., 2018). Unity's programming structure has a number of benefits including its direct link to, and familiar coding practices of OOP, though it can quickly become cumbersome, as its actual hierarchical application at runtime is more in line with a human's thinking than a computer's processing.

GameObjects
GameObjects are essentially a container to hold a collection of components that can be placed hierarchically in a scene. They represent everything within the scene itself and are commonly made up of additional children GameObjects increasing the complexity. For instance, a parent GameObject may contain a renderer component with a mesh to represent a volumetric 3D model of a soldier, while a number of additional GameObjects positioned hierarchically as its children could contain logical components that define the soldier's behaviors, collision actions, or locomotion.

Components
The Unity Editor comes with a large number of built-in components; these can represent physical objects within a scene visually using components such as mesh renderers. However, they can also express physical aspects of the scene non-visually as audio players or physics colliders, and can even represent behaviors and actions through custom C# scripts. An example of a C# script as a custom behavior would be the logic that controls a characters movement based on user input, through mouse clicks, joysticks, or the keyboard.

Experimental Technologies for Creating Autonomous Agents

Using Unity3D as our runtime environment allowed us to discover the upper limits of our OOP approach as well as to experiment with Unity's newest preview initiative to rebuild their core foundation, known as the "Data Oriented Technology Stack" (DOTS). DOTS seeks to make multithreaded programming simpler and more performant with only minor modifications to existing code, and provides simple frameworks for creating new code. DOTS includes three new features within the Unity core that work in conjunction to increase performance. With some effort, we were able to move away from GameObjects and transition a lot of our existing code into the ECS pattern. It is worth noting that ECS is commonly considered to be of a higher quality insofar as its software metrics, especially related to readability, maintenance and re-use. However, there is research to suggest that the internal complexity metrics, such as *Percentage of Comments*, *Methods per Class*, and *Statements per Method*, are actually similar to traditional models (Reilly & Chalmers, 2013).

ECS
The Entity Component System, or ECS, is an architectural pattern that makes up the primary foundation with which Unity's developers have exposed the DOTS framework, and separates the codebase into 3 distinct areas. Entities, which are simply identifiers that do not have any functionality on their own. Components, which are the data, they can be tied to the identifiers but also have no inherent behaviors. Finally, the Systems, they take the data solely from the needed components and transform it, resulting in behaviors both behind the scenes and on-screen. The consequence of ECS is data oriented, highly optimized code that favors composition as opposed to inheritance. While the ECS framework does work independently, the greatest optimization occurs in conjunction with Unity's additional experimental offerings, the C# Jobs System and Burst Compiler.

Job System
A Job in computing is a "unit of work, or unit of execution (that performs said work)" (Wikipedia, 2018). Unity exposes a system for scheduling this work through its IJob Interface in the C# Job System. The greatest benefits of the Jobs System are realized through the code being allowed to run multi-threaded across all of the computer's available cores quickly, efficiently and safely. The Job System also exposes Unity's native C++ Job System directly to the C# code so that custom game scripts are run as jobs directly with Unity's core engine code. Unity's Job System provides protection against many of the normal issues presented when writing code for multithreading such as the bugs that can be caused by race conditions (Unity Technologies, 2018).

Burst Compiler

The Burst Compiler is a new Low Level Virtual Machine (LLVM) based compiler within Unity. LLVM is an open-source project that originated at University of Illinois and has sought to bridge the gap between high-level code and native binary code by providing a toolchain and compiler libraries (LLVM Foundation, n.d.). In the case of Unity DOTS, the burst compiler uses LLVM and takes the highly optimized code written through the ECS and Jobs process to create highly performant system specific native code optimized directly for the target system. This type of optimization would normally need to be achieved through time consuming and intensive native specific optimizations and tuning, or complex code written in a language that can act directly on the processor such as C or C++.

CURRENT WORK

In order to best understand the benefits and consequences of both methodologies, we chose to create a baseline in the current research based upon a simple agent type that we could easily extend in future work to determine how various levels of complexity directly affect our evaluation methods. To set our baseline we designed a simple reflex agent with a model-based intelligence. These agents were given a "Move To" command that took input from a human user and passed the destination as a target to all of the individuals within the agent group. By basing our initial tests on a single common AI utility, we could easily understand the measures of both the traditional OOP and experimental ECS methods without overburdening the structure of our research. We built the framework with a high extensibility so that we can easily add and modify additional AI utility for the agents without having to recreate the foundational elements of the simulation.

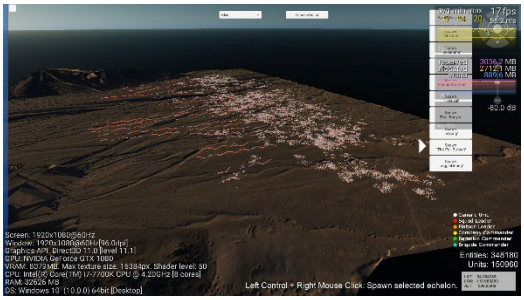

Figure 1. Test on One World Terrain

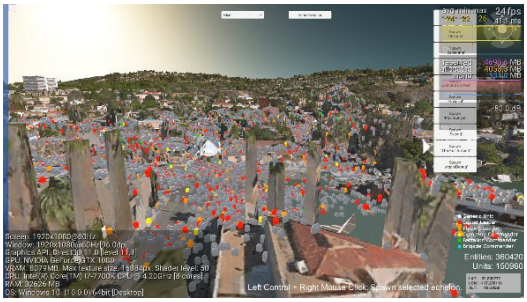

Figure 2. Test on MS Bing Data

We utilized highly detailed drone capture data from the ICT One World Terrain project (see Figure 1), which utilizes UAS collected imagery to produce game-engine ready usable terrains (Spicer, McAlinden, & Conover, 2016). We also used terrain generated through Microsoft Bing sources to establish secondary analysis (see Figure 2). Both of the data sources allowed us to analyze the baseline on a real-world setting, so that we could make the most efficient and applicable use of our mobility utility.

The data model used for mobility was the A* algorithmic method, through the third-party A* Pathfinding API (Granberg, 2017) for Unity, written by Aron Granberg. Using this library allowed us to create a navigable area through a pre-calculated navigational graph created from the terrain mesh. By utilizing the defined graph our agents were able to navigate both successfully and realistically across the terrain to the destination point.

The goal of our initial research was to understand how far we could optimize a single workstation, though we understood when we began that ultimately we would probably need to move to distributed processing in order to achieve the million agent target. The workstation utilized for our testing was running Windows 10 64-bit, with an Intel Quad Core i7-7700K CPU @ 4.20 GHz, 32 GB RAM, and an NVIDIA GeForce GTX 1080.

EXPERIMENTAL TECH COMPARE CONTRAST WITH TRADITIONAL SOLUTIONS

Traditional Method Implementation

We created each Agent as a GameObject, attaching the required components necessary to enable it to achieve our testing goals of realistic navigation from starting point to a target destination. This included a mesh renderer with a simple capsule mesh and capsule collider, roughly approximating the volume of a human agent.

We added the necessary MonoBehaviour (Unity's base class) scripts to each GameObject, which were necessary to facilitate the Agent's mobility and allow access to the Unity-specific functions related to a GameObject, notably Start() and Update(). Start() running once when the script is first enabled and Update() being called once per frame. These scripts made use of the A* library and were quite simple in their structure. Simply passing a target point and requesting a calculated path then moving the agent along the path until the target point was reached.

We created a prefab of this Agent GameObject and verified the viability, testing with just one agent, later instantiating a larger and larger number of Agents from this prefab. We initially began our research with a small set of agents, and begin to analyze and enhance performance, through use of the Unity3D profiler and logs. This common method for runtime optimization and profiling quickly allowed us to find "hotspots" in the code or processes, which we were able to focus on and optimize even further.

Traditional Method Results

We initially hit a maximum of around 1,000 entities, which we were able to optimize in both the code and by modifications to the A* pathing in order to bring that number to around 2,000 unique entities. This was only 0.2% of our target goal of one million entities, and it is worth again noting that for this baseline these were simple agents with only one utility mobility behavior for moving the agent to a user selected target point. The resource limitations were met almost immediately in the traditional OOP methodology. Adding any additional behaviors or fidelity would ultimately increase the requirements resulting in additional bottlenecks and framerate reduction.

What Worked

The Unity environment has been updated and solidified since Version 1.0 in 2005 for superior tractability and stability amongst game engines and simulation architectures. The OOP and GameObject/Monobehaviour approach is well documented and has vast community support that allows for quick prototyping and a high production rate of game systems. The familiarity and ease of modeling complex architectures using the traditional methodology of GameObjects and MonoBehaviours with OOP allowed us to rapidly assemble our test scenario and left the AI Agents cleanly decoupled from the virtual environment simulation. The methodologies, libraries, and approaches we took are all well documented across both the gaming and simulation industries, and we were able to utilize and build upon pre-existing and ongoing research for the One World Terrain project as well as related Game Engine Optimization work involving custom mesh types, asset bundles, and texture compression formats (Liles, Ortiz, & Caruthers, 2019).

What Didn't Work

The number one penalty against the simulation experiment was that the entirety of the code was running on the main thread, which is shared by many other parts of the system resulting in highly inefficient use of CPU cores and resources. Also, the use of C# and .NET libraries in Unity meant that particular language features, such as automatic garbage collection, wound up consuming additional memory and computational cycles, adding to the performance issues when compared to other native languages, such as C++ that are commonly used by competing game engines.

There were a number of other weaknesses with this method when utilizing a large number of Agents. A primary weak point was a prevalence of in-cache misses due to the conventional system of allocating data in memory randomly, as well as corrupted data representations that were also prevalent enough to be noticeable, especially at higher levels of autonomous deployments.

Even when attempting to factor in the benefit of additional computational power via a distributed systems implementation we would fall short of reaching our goal of a baseline simulation with 1,000,000 Agents.

Experimental Method Implementation

In order to test with ECS we had to modify our existing traditional approach and begin to look at our OOP foundation in a different manner. We separated the existing GameObjects that contained all of the data and behavior for our Agents into their component pieces. The Agent entity became a singular container, empty of any pre-assigned data or behavior.

We needed to understand the various systems we were making use of and determine how best to transition the existing MonoBehaviours to systems. This would be easier for Unity specific features such as locomotion which are better supported, while for a third party library such as A* Pathfinding it would be quite a bit more difficult.

Experimental Method Results

Through utilizing ECS and Job System, we were able to see a large increase in the number of Agents we could deploy to the simulation at any given time. We were able to increase from 2,000 to 150,000 Agents at a steady and usable framerate much closer to our performance target of 30 FPS. The multithreading greatly increased the amount of work our test scene could achieve in one frame as we were able to visualize in the Unity Profiler (see Figure 3). This was a 7400% increase and brought us to 15% of our target goal without any additional optimizations, while still subject to the limitations of our singular hardware system. It is worth noting that each Agent actually requires a number of Entities for functionality and visualization so we were actually running 350,000 entities in the simulation total, including those required for pathfinding and terrain generation and calculation. In our future work, we discuss how we can further optimize this number via distributed processing.

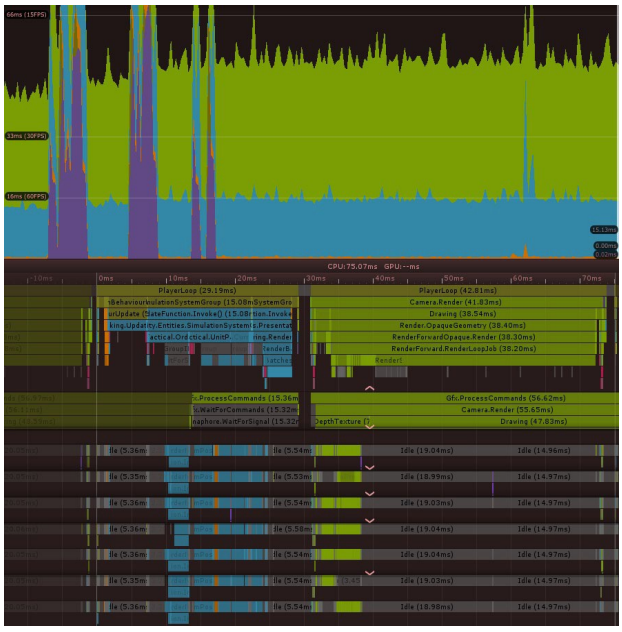

Figure 3. Distribution of work within one frame

What Worked

The largest benefits to our ability to simulate such a large number of Agents came through multithreading. This was relatively simple to integrate as a strong understanding of multithreading is not required due to the strict guidelines Unity integrates into the use of the exposed multithreading methods.

We were able to cut down greatly on Cache Misses as well, due to the ECS architecture, which by its nature packs related data closely together in memory, which also greatly reduces lookup times. Researchers with Intel studied this structure extensively (Ferreira & Geig, 2018) and their data (see Figure 4) highlights the beneficial difference in the memory structures of ECS architecture, which we were able to reproduce effectively.

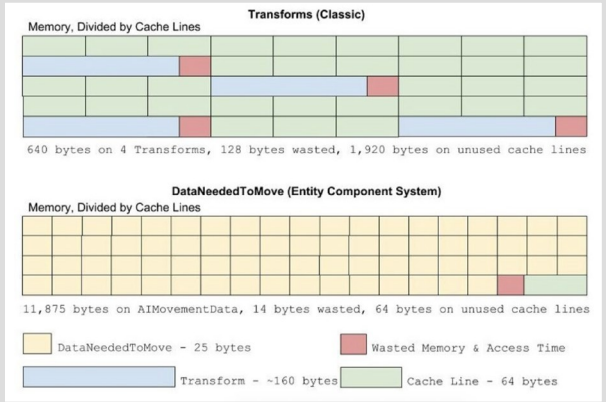

Figure 4. Memory Structure. Adapted from "Get Started with the Unity* Entity Component System" by Ferreira, C., & Geig, M. 2018. Copyright 2019 Intel Corporation

Another large benefit came through our use of Burst compliant code, which made assumptions about memory in order to create highly efficient, system specific assembly code at compilation. Though notably we were not able to

integrate this as much as we would have liked mainly due to the Hybrid ECS approach we had to take in order to utilize functions such as the Unity user interface (UI) and third-party A* libraries which are not yet supported.

One additional benefit of note is that by its nature the ECS system decouples implementation and data, meaning that we were able to disable individual systems for rapid debugging without breaking code.

What Did Not Work

However, we did run into a number of issues when implementing the ECS system into our codebase. The most notable of these is the fact that Unity's ECS foundation is still in early development. The exposed features and best practices for their use were constantly changing, requiring frequent refactoring to keep our code working. This is the nature of experimental technology, but it greatly slowed down the speed at which we were able to implement and optimize for ECS. A number of necessary features were completely unsupported by ECS when we began our research, including booleans and physics, which have since been added. However, there is still no ECS UI support, which means that even without implementing a third party library we forced to implement a Hybrid approach. This prevented us from attaining the full benefits of a pure ECS system. Even a single OOP system that was not converted to ECS could take longer to run than all of the ECS systems put together. In our case, by simply disabling the UI that we used to show Military Symbol Identifiers we were able to increase FPS while subsequently decreasing RAM utilization (see Table 1) two specific metrics we attempted to optimize throughout this work.

Table 1. Resource Use According to UI Status

UI Status	RAM ALLOC	FPS AVG	FPS MAX
Enabled	3208.4 MB	16	17
Disabled	2712.1 MB	17	20

We also ran into a number of issues when attempting to implement the Burst compiler, as its optimization stems from its restrictiveness it can only work with blittable value data and will not accept reference types. We had to convert a number of instance classes to value based structs in order to take advantage of the Burst compiler, though in some areas of the code, this was not yet possible due to limitations of the experimental system and we lost optimization in those.

An area of research we encountered, but is beyond the scope of our initial experiment, is to understand the makeup of higher fidelity agents in regards to the addition of entities within the ECS. As we discovered, running a single Agent was not a single entity, but instead each agent was comprised of the necessary parts to function reliably at our chosen fidelity. It would be important to note the cost of higher fidelities in their addition of entities, or to even understand how other types of agents would tax the ECS by necessitating additional entities. For instance, it would be important to attempt to understand how many entities would be required to model an accurate non-human agent, such as a ground vehicle or helicopter within this architecture. What additional toll would weaponry and flight dynamics take on the system, and how would that affect or limit the target goal of one million autonomous agents? These are important questions to begin to understand and our baseline with ECS shows that there are appropriate ways to begin to tackle these questions, but it seems fruitful only when the ECS can be implemented in a non-hybrid fashion, so that requirements like physics are not a significant bottleneck.

In addition, it is worth noting that ECS entities do not appear in the Unity hierarchy, so they can only be created/modified within scripts or via utilizing the hybrid approach, in which a "Convert GameObject to Entity" script is attached to a GameObject. This is a steep departure from the traditional method of visualizing GameObjects within the editor and made it difficult to gain a full understanding of how our scripts were actually working without implementing custom UI and debug functionalities to display the necessary data and values for analysis. However, there is an ECS debug viewer the displays a list of all running ECS systems, and each system's affected entities, though it is not as fully featured as the regular hierarchy and it is difficult to identify the individual entities.

Finally, the ECS system is sparse, often changing, and sometimes outdated, this made it difficult to quickly learn ECS and be able to achieve the greatest benefits from it. While this is again a symptom of an experimental architecture with Unity3D and a foundational change, it was tough to create, optimize and support the full framework we were building.

APPLICATIONS

Benefits to Virtual Battlefield and Simulations

Individual autonomy, communication, fidelity, and quantity of AI Agents are just a small number of the qualities necessary to create fully realized battlefield simulation for understanding, planning, and analysis. A number of initiatives benefit from Artificial Intelligence in a virtual battlespace, we will look at just a few in this paper in order to understand how to best realize our intentions and how they've been influenced by current goals.

AI in War Gaming
War-games are a consistent and important first line of war strategy. Attempting to simulate and understand the near and far future of the battlefield has been useful to military strategists since its origins in Prussia in the late 1700's. Computer-based and computer-assisted professional war-games allow today's military to gain a greater understanding of potential battlefield outcomes and reactions, and the ability to determine realistic unit movements at all levels of organization from fire teams to field armies. Being able to simulate a single side's field army group at a realistic level requires individual agents numbering up to 1,000,000. This is our current target goal for this research. Adding complex behavioral fidelity improves the quality of the simulation, ultimately leading to a realistically complex battlespace environment where increasing the complexity through additional factors and actions of warfare allow indeterminate and multi-faceted war-gaming operations to be run and analyzed.

AI in Training
The Army's Synthetic Training Environment has a number of components that are required to bring it online and make it truly affective and sustainable in its effort. A critical factor to this training will be the presence of AI elements in regards to Autonomous Agents. Realistic training will require incredibly large numbers of computer controlled and directed entities, both enemy and friendly, in order to achieve actualized battlefield fidelities and realism. To be able to properly augment and enhance each individual entity with the intelligence and utility required to achieve realistic fidelity is a task within itself, but progress towards that goal must first be researched by creating baselines and understanding the current and realistic limitations of such an effort. Our work hopes to begin to define the scope and a few paths towards achieving the million-agent goal, while also collecting the data necessary to begin understanding the more difficult tasks of behavioral fidelity required for realistic simulation.

AI in Operational Planning
Operational planning on the battlefield is generally done in a way in which individual units are grouped together and assigned as singular entities. While this is helpful for a representation of the battlefield, it misses a number of items that the work in this paper attempts to correct. Troop movement and visibility becomes much more granular when each Agent can be solely represented as opposed to an approximated cluster. The effects of terrain trafficability can be realistically modeled and simulated as well, accounting for a level of detail down to the footsteps of each individual soldier. The representations are numerous but can be summed up in the idea that by raising the quality and fidelity of a simulation you only succeed at simulating reality when the number of Agents represented is as close as possible to the real world human presence. Otherwise regardless of how detailed a group behavior is, or how much attribution and relational data you have provided to a 3D model dataset, you are still grossly underestimating the reality.

Other Simulation Uses
Any large-scale simulation could make use of this paper's work by simply adjusting the agent behaviors to fit the scenario. The most direct analog would be in a large city planning environment, where the impacts of large scale construction projects would have an effect on the daily flow of traffic, vehicular and pedestrian. By tweaking agent behavior to describe these things adequately within any given city terrain dataset you could quickly create baselines of 100,000 Agents navigating the current environment. Then, run that analysis under any number of road closures or creation scenarios, quickly discovering the most optimized routes for detours and best times of day in which

construction operation would have reduced impact on citizens. Large events often bring in hundreds of thousands of visitors to areas that are not generally suited to such an influx. The event planners could utilize this work to simulate and determine with a high fidelity what the best way to manage such a crowd may be. In the reverse, these systems could be used to determine egress and evacuation in large-scale emergency simulations where current systems either render these events at a low fidelity or make use of misrepresentative group behaviors to approximate the outcome of the scenarios.

Taking this idea a step further you would be able to simulate a large scale logistical or production environment, where the Agents can be modified to represent factory workers, whether human or autonomous, and even factory products and production components. It would be possible to represent and develop organizational and production systems with a very high level of detail at a very high level of visual fidelity. In a large warehouse simulation environment, the stored objects themselves could have an internal knowledge to consistently update their on-shelf neighbors and positions as they're shuffled from place to place, allowing for a detailed reporting and extensive analysis to take place.

FUTURE WORK

Advanced Terrain Models

We used a relatively simple terrain model for our baseline testing in this work. An area large enough to support the amount of units we had hoped to deploy, while also being simplistic enough to not create additional bottlenecks we would need to work through in this initial research. In further work, we plan to implement more complex models that use ICT's advanced and fully automated classification and segmentation research. (Chen et al., 2019) This would give the benefit of additional ground-truth to the simulation including providing the AI with knowledge of existing roads, geo-typical trees, and other material and object attributions so that the processing required would be decreased while also implementing advanced navigation and trafficability, even within a complex urban or battlefield environment. This would improve the fidelity and behavior of the Autonomous Agents without tasking the workstation to process these improvements, as they would be provided inherently through the richly attributed, classified, and segmented terrains.

Distributed Processing

After reaching the maximum limits with our hybrid ECS setup on a singular machine, we can begin to explore the potential of distributed processing. On past research initiatives at ICT we have used multiple machines to handle large simulation environments to good effect. Notably, the Densely Populated Urban Environment, or DPUE, effort in which we clustered four machines together so that each one maintained responsibility for a geographical segment within the simulation. The individual machines handled location and state data and then passed calculations back to a master machine that handled the rendering and visualization of the environment. This would be a good starting point for this current work, by segmenting the load over an area so that the machines could split handling the Agents. Difficulty would increase alongside fidelity as we improved the agent quality, adding communication and interaction would especially be difficult, as coordinating those systems across multiple machines would present problems with the ECS implementation. Notably, because ECS is still in its experimental stages within Unity, the required networking features are not yet ready for public use. However, we do know that Unity is actively working on both networking and physics implementations of ECS in the Unity Editor, which would be requirements for this type of distributed processing.

Cloud Implementation

Once we have optimized and hit the hardware limits of our single machine, an additional next step would be to experiment with a cloud-based implementation. This would remove the restrictions set in place by any singular hardware or localized cluster, as we would be able to access the processing power of hundreds of servers at one time, working in tandem to provide resources to the simulation. As the Army's Synthetic Training Environment, STE, may potentially live in the cloud, it would be beneficial to all of the processes it needs access to for the virtual proximity to be lessened. The issues with scalability that are encountered in on premise hardware are generally eliminated with the cloud, as it takes no time at all to expand the resource pool as opposed to ordering and setting up physical equipment. Another benefit of this implementation is that we would be able to focus our efforts entirely on development and optimization of the software, removing any constraints present in physical hardware maintenance and support. The costs would generally be lower as well, especially when factoring in the complexity and power required implementing such a massive system. However, since we would not own the hardware or network beyond

our own premises, we could be subject to downtime or feature loss due to factors out of our control, either at the data center or anywhere along the network lines connecting us.

CONCLUSION

Many other game engines support a version of Entity Component System, but Unity is taking a new approach with their Data Oriented Tech Stack, integrating and streamlining the components directly into their software core. It definitely seems that the industry is heading in this direction, with notable examples as Improbable's SpatialOS even spearheading the ECS distributed cloud approach to create massive online worlds. Our initial work has definitely shown the advantages to the ECS Framework as related to optimization, but the experimental aspects and workload required to transition existing code mean it's worth considering if it is a necessity in all cases. However, in our specific case, within the matter of scale and large simulation, it seems to be a definite choice for the architecture, especially as the experimental nature is temporary and Unity is committed to creating a solid and stable ECS production pipeline.

ACKNOWLEDGEMENTS

The project/effort/work depicted here was or is sponsored by the U.S. Army Research Laboratory (ARL) under University Affiliated Research Center (UARC) contract number W911NF-14-D-0005. Statements and opinions expressed and content included do not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred.

REFERENCES (WIP)

Brackeys. (2018, March 04). How many AI Agents can Unity handle? Retrieved from https://www.youtube.com/watch?v=G9Otw12OUvE

Chen, M., Feng, A., McCullough, K., Prasad, P.B., McAlinden, R., Soibelman, L., (2019). Fully Automated Photogrammetric Data Segmentation and Object Information Extraction Approach for Creating Simulation Terrain. *Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC)*

Delooze, L. L., & Viner, W. R. (2009). Fuzzy Q-learning in a nondeterministic environment: Developing an intelligent Ms. Pac-Man agent. *2009 IEEE Symposium on Computational Intelligence and Games*.

Ferreira, C., & Geig, M. (2018, May 31). Get Started with the Unity Entity Component System (ECS), C# Job System, and Burst Compiler. Retrieved from https://software.intel.com/en-us/articles/get-started-with-the-unity-entity-component-system-ecs-c-sharp-job-system-and-burst-compiler

Gallagher, M. & Ryan., A. (2003). Learning to Play Pac-Man: An Evolutionary, Rule-based Approach. *Proceeding of the 2003 Congress on Evolutionary Computation (CEC)*.

Granberg, A. (2017, April 9) *A* Pathfinding Project.* Retrieved from http://arongranberg.com/astar/

Granberg, A. (2018, January 07). RTS Demo. Retrieved from https://www.youtube.com/watch?v=wBUdMXeeej0&feature=youtu.be

Holopainen, T. (2016). *Object-oriented programming with Unity: Inheritance versus composition.* (Unpublished Bachelor's thesis). JAMK University of Applied Sciences. Jyväskylä, Finland.

Job (computing). (2018, February 07). Retrieved from https://en.wikipedia.org/wiki/Job_(computing)

Juliani, A., Berges, V.P., Vckay, E., Gao, Y., Henry, H., Mattar, M., and Lange, D. (2018) Unity: A general platform for intelligent agents. arXiv:1809.02627, 2018.

Liles, L. & Ortiz, J. (2019). Geospecific 3D Terrain Data Optimization Solutions for Game Engines. *Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC) 2019.*

LLVM Foundation. The LLVM Compiler Infrastructure Project. (n.d.). Retrieved from http://llvm.org/

Reilly, C., & Chalmers, K. (2013). Game physics analysis and development — a quality-driven approach using the Entity Component Pattern. *The Computer Games Journal, 2*(2), 125-153.

Spicer, R., McAlinden, R., & Conover, D. (2016). Producing Usable Simulation Terrain Data from UAS-Collected Imagery. *Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC)*

Tesauro, G., Gondek, D. C., Lenchner, J., Fan, J., & Prager, J. M. (2013). Analysis of Watsons Strategies for Playing Jeopardy! *Journal of Artificial Intelligence Research, 47*, 205-251.

Unity Technologies. (2018). The safety system in the C# Job System. Retrieved from https://docs.unity3d.com/Manual/JobSystemSafetySystem.html