Exploring Implementation Methods to Generate an Accurate Representation of the Night Sky for use in Virtual Reality

Anna Sikkink
asikkink@hawaii.edu
Information and Computer Sciences
University of Hawai‘i at Mānoa
Advisor: Dr. Jason Leigh

Abstract

In this paper I investigate three methods to generate an accurate representation of the night sky and determine their suitability for use in Virtual Reality (VR). Developed in Unity3D, these implementations were created with 1) GameObjects, 2) ParticleSystem, and 3) Unity's Data Oriented Technology Stack (DOTS). Comparing the performance of these three versions through a series of tests, the DOTS implementation was found to maintain target frame times across all tests, which included visualizing all stars from the Hipparcos star catalog. This project will be integrated into the next iteration of the Kilo Hōkū VR application, which will allow my development team and I to make the application more accessible as an educational tool.

Background

Kilo Hōkū VR, a virtual reality simulation of sailing on the double-hulled sailing canoe Hōkūleʻa, began as a class project in 2016, developed by Patrick Karjala, Kari Noe, Dean Lodes, and me. Our initial goal was to discover how a virtual reality (VR) environment could aid in the learning and teaching of Modern Hawaiian wayfinding. Users experience being onboard the Hōkūleʻa on the open ocean, where they can view stars and highlight constellations, and see the Hawaiian star compass in context. With these tools, users can apply Modern Hawaiian wayfinding techniques to navigate and sail the Hōkūleʻa between two Hawaiian islands. Teachers are also able to test a user's existing knowledge within the virtual environment through use of a teacher controller, which allows them to turn on or off specific features in the simulation, such as the star compass, celestial equator or meridian lines, and change the time or location of the user[1].

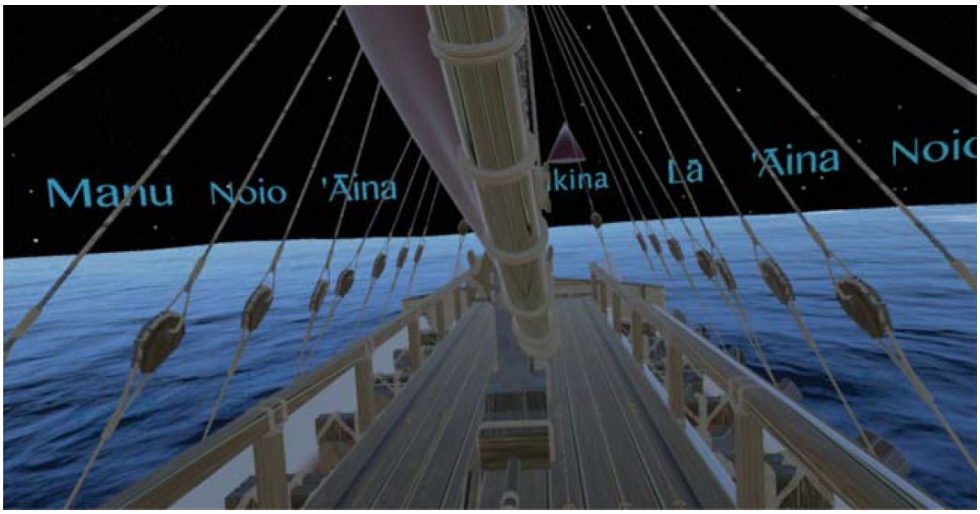


Figure 1: Kilo Hōkū VR from a user's view within virtual reality at the start of the simulation[1]

Kilo Hōkū VR was developed using the Unity3D game engine and was originally designed for the HTC Vive VR headset. The Vive headset was chosen due to the fidelity of tracking, which helped in making the simulation easier to use so that it would be more approachable, especially for users new to virtual reality environments[3]. Version 1.0 of Kilo Hōkū VR was released in

2020 on itch.io, making it available for anyone with access to the Vive to download and use[2]. Kilo Hōkū VR has since undergone continual development and experimentation as my team and I work to build the simulation into a more effective teaching tool. In collaborating with outside stakeholders to further develop the application, it became apparent that several significant changes would be needed for the simulation to become more accessible to educators.

Figure 2: a user highlighting a constellation in Kilo Hōkū VR under instructor guidance[1]

My development team determined that the most beneficial update to the application would be to port it to a more approachable VR headset. The Vive headset connects to a computer, which runs the VR application and sends the visuals out to the headset display. To track the player's movements, lighthouse boxes are mounted around the space that send out infrared lasers, which the Vive headset and controllers detect and use to generate positional information for the VR application. To run the Vive, a computer is required with the specifications of: an Intel Core i5-4590/AMD FX 8350 Processor, an NVIDIA GeForce GTX 1060 or AMD Radeon RX 480 GPU, and at least 4GB RAM[4].

The Meta Quest 2 (previously Oculus Quest 2) is not only more affordable than the Vive, but also operates as a standalone device which does not require tethering to a computer, and uses inside-out tracking which removes the need for external equipment such as lighthouses. This is favorable for classrooms as they would only need to purchase the Quest 2 headset to run the application. The lower cost of the Quest 2 also means that educators would be able to purchase more Quest 2 headsets overall, instead of needing to purchase both a headset and a computer that meets the required specifications. Additionally, there is much less setup required with the Quest 2, increasing the ease of use in a classroom setting.

However, the Quest 2's specs include a Qualcomm Snapdragon XR2 processor, Adreno 650 GPU, and 6GB of RAM[5]], which is less processing power than we would get from a computer running the Vive. Due to these hardware limitations, many adjustments needed to be made to the Kilo Hōkū VR simulation to ensure it not only runs on the hardware, but also at a high enough frame rate to keep users from getting simulation sickness. Applications running below

60 frames per second (FPS) can cause simulation sickness in VR, while applications running at 120 FPS and above has shown better user performance and VR experience overall[6].

With support and feedback from stakeholders, my development team has produced a functional Quest 2 port of Kilo Hōkū VR, with a public release planned for later this year.

Motivation

Another significant update needed for Kilo Hōkū VR was to improve the original implementation of the sky to allow for more precise interactions and improved accuracy of the sky visualization. Therefore, I chose to explore different methods of implementing a generated sky for VR for my master's capstone project. The initial implementation of the sky in Kilo Hōkū VR consisted of a sphere object with reversed normals to display a texture on the inside of the sphere. A "star field" texture was created using a high resolution full sky image from NASA's Scientific Visualization Studio, then applied to the sphere[3]. This created a virtual celestial sphere, appearing like a dome with the user and Hōkūleʻa inside at the center, from which the user would see the stars around them as if they were looking at a clear night sky.

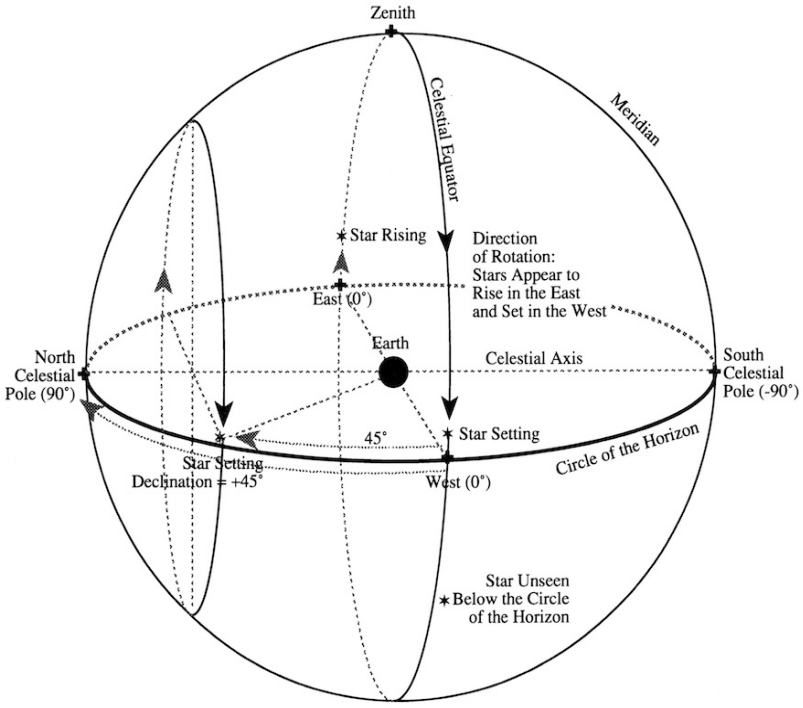
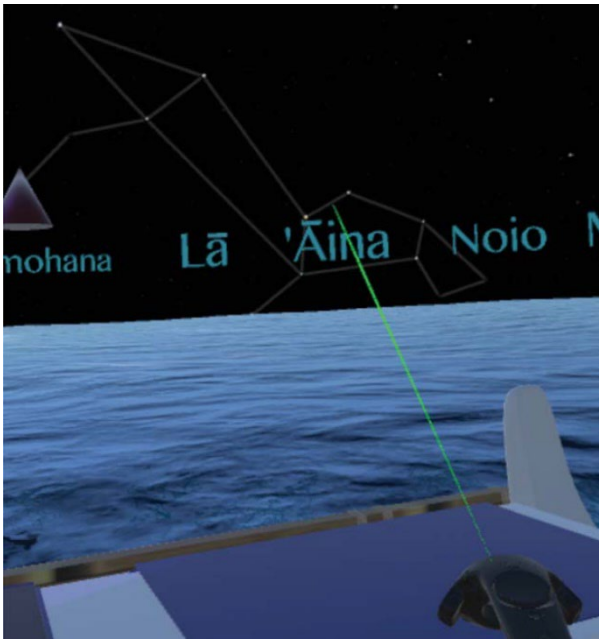


Figure 3: The concept of a celestial sphere is an imaginary sphere with the Earth at the center. Looking overhead, we see the half of the sphere as the sky as we see from Earth, appearing as a dome. The other half of the sphere is below the horizon and not visible[21]

In order to allow users to point at the sky to identify constellations and Hawaiian starlines, collider boxes were manually added around each group of stars included in a constellation. Upon triggering the collider with a raycast, which is presented as a laser pointer in the

simulation, the texture on the celestial sphere object is swapped out with another texture with the constellation lines shown for the corresponding constellation.



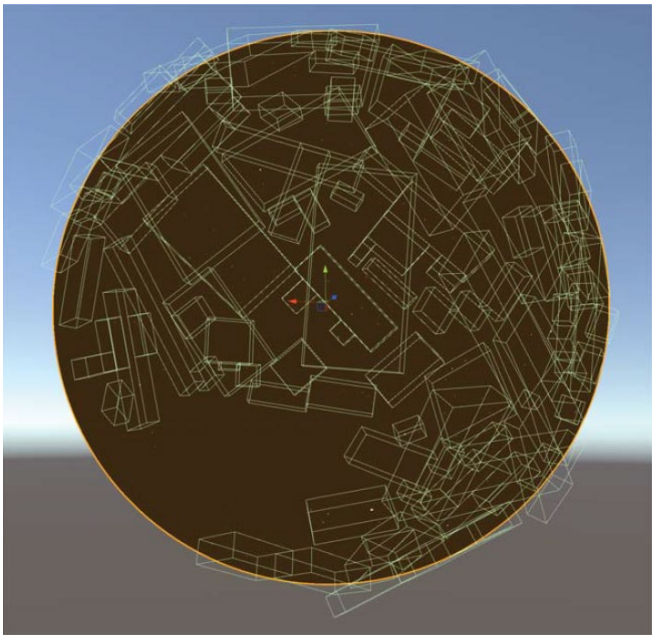


Figure 4 (left): VR view of a user highlighting a constellation in Kilo Hōkū VR
Figure 5 (right): Kilo Hōkū VR's celestial sphere with constellation colliders as seen externally[1]

This method was a simple way to create a mostly accurate sky and include the interactions we wanted, but it could not be easily improved upon in its current state. With 90 texture images (88 western constellations, the Hawaiian starlines, and the default sky with no lines or constellations highlighted), each at a resolution of 8192px by 4096px as required to keep the image clear in VR, the act of constantly swapping the individual textures on the celestial sphere is not very efficient. Adding the ability to highlight individual stars or show other cultural constellations would not be a quick addition as colliders would need to be manually added in the Unity Editor, since there isn't any positional data for the stars in the simulation.

Additionally, some visual defects are apparent in the star positions and appearances near the poles where the texture doesn't match up properly on the sphere. This is problematic because the north star (Polaris/Hōkūpaʻa) is a significant star in Hawaiian wayfinding, as part of Ka Iwikuamoʻo, "The Backbone" Hawaiian starline. Not only is Polaris/Hōkūpaʻa a quick way to orient cardinal direction since the star is nearly directly on the north celestial pole, but the Ka Iwikuamoʻo starline runs from this star across the sky to the Southern Cross constellation, also known as Hānaiakamalama, which is near the south celestial pole[11].

Figure 6 (top): Visual defects near the north celestial pole. Polaris/ Hōkūpa‘a star appears like an arrow shape, and the stars around it are stretched out into lines.
Figure 7 (bottom): The same region of sky as Figure 6, as displayed in Stellarium, demonstrating a correct visualization of these stars.

The Pleiades, known as Makali‘i, is a cluster of seven small stars. They rise before the stars in the Hawaiian starline of “Ke Ka o Makali‘i”, and are used as a visual guide during the first month of the year (November – December)[11]. Unfortunately, in Kilo Hōkū VR, they don’t display true to real life due to the texture resolution, which was reportedly confusing students who were using the app in a classroom setting. Generating these stars will improve the clarity of the visualization compared to using the texture image, resolving this issue.

Figure 8 (left): The Pleiades/ Makaliʻi as they are displayed in Kilo Hōkū VR, which appears blurred and individual stars are less distinct.
Figure 9 (right): The Pleiades star cluster as displayed in Stellarium, which is truer to reality.

Ultimately, generating the stars based off of star catalog data will help resolve these visual issues in addition to providing added possibilities to improve interactions and display other useful information in the application.

Approach

My goal with this project was to create a new virtual celestial sphere which could be utilized in Kilo Hōkū VR, one in which each star would be generated based on positional and observational data from a star catalog. With the star position data in the simulation, the stars themselves will appear clearer and accurately placed. The Hawaiian starlines, constellation lines, and colliders can also be generated from the positional data, instead also of needing to manually create them as was done in Kilo Hōkū VR 1.0, while retaining the interactivity aspects needed. This also gives us more flexibility in adding in other cultural constellations or helpful visuals in the future as well. Generating each star also makes it possible to make them individually interactable, meaning the user could point at any star and view relevant information such as star name, which constellation or Hawaiian starline it belongs to, and any other facts we want to display.

This generated starfield needs to run alongside the other elements of our Kilo Hōkū VR simulation, which include the ocean system, the Hōkūleʻa model, the star compass, the meridian and celestial equator lines, and the island models. In addition, it needs to be able to run in

virtual reality, ultimately on a less robust hardware device such as the Quest 2. To determine a suitable solution which meets the aforementioned requirements, I created a series of implementations which used different methods to generate the stars with the intent to compare them to discern the most efficient and effective method to use for the next version of Kilo Hōkū VR.

Method

In Unity, I developed three implementations of a generated starfield using different methods to create stars: stars as standard GameObjects, stars as individual particles in Unity's ParticleSystem, and stars as entities using Unity's new Data Oriented Technology Stack (DOTS).

Independent of implementation method, in order to create an accurate representation of the sky, star data from a catalog needs to be imported into the simulation. The celestial sphere in Kilo Hōkū VR is intended to be a tool for learning the Hawaiian starlines and constellations to build the visual recognition skills needed to navigate using Hawaiian wayfinding techniques. Therefore, it is most important to display the stars which are visible to the naked eye. I decided to use the Hipparcos catalog (HIP), which was updated with re-processed data in 2007 and contains 117,955 stars[13]. The HIP catalog was created from data obtained by the HIPPARCOS space astrometry satellite, with the goal of determining parallaxes of stars that were brighter than around +12.4 in apparent magnitude[19]. For context, the apparent brightest star from Earth, Sirius, is a magnitude of -1.5, and the naked eye limit is about +6.5[17], so the data from HIP provides stars with magnitudes beyond only the stars needed for visualizing in Kilo Hōkū VR.

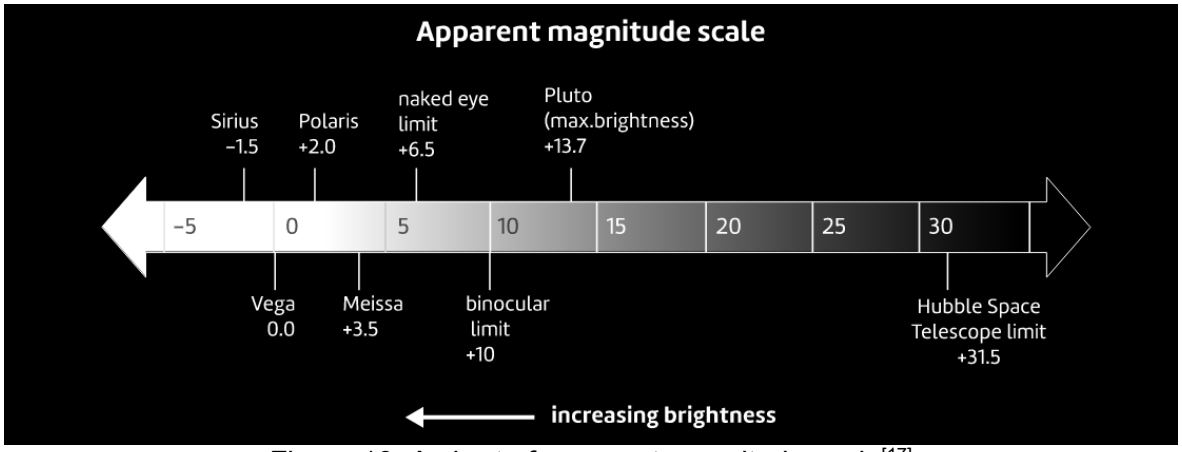


Figure 10: A chart of apparent magnitude scale[17]

The HIP catalog can be downloaded from the Vizier Catalogue Service[13] and stored as a CSV file, then read into the Unity scene via a C# script using the System.IO file reading methods. From there, the imported data gives us coordinates for each star, which are used to place the stars in their correct locations in the scene to create a virtual celestial sphere. Star coordinates are given in two numbers, Declination and Right Ascension. These are analogous to latitude and longitude coordinates on the Earth.

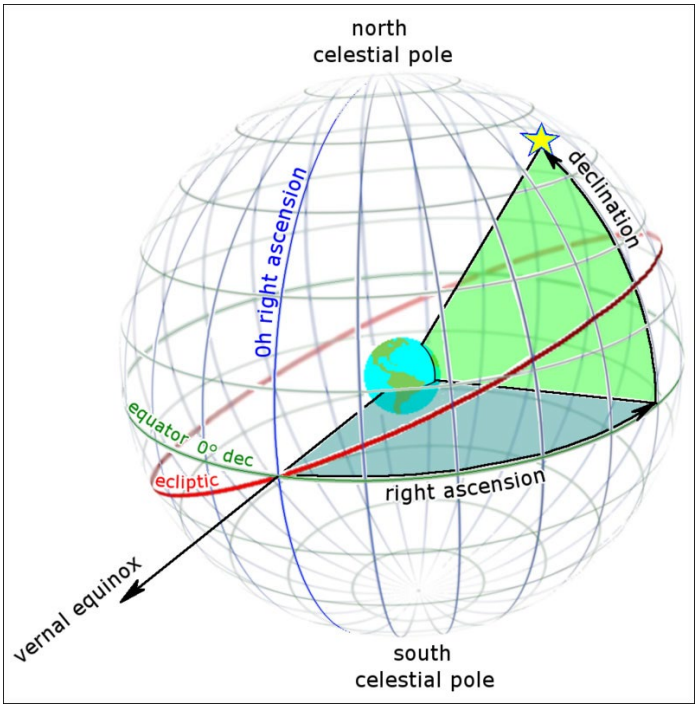


Figure 11: Right Ascension and Declination on the celestial sphere[18]

Declination is the angle of a star away from the celestial equator, an imaginary line around the middle of the celestial sphere which is on the same plane as the Earth's equator. Celestial bodies to the north of the celestial equator have positive declinations up to +90 degrees (the north celestial pole), while bodies south of the celestial equator have negative declinations up to -90 degrees (the south celestial pole)[21].

Right Ascension gives the position of a celestial body in relationship to lines from the north to south celestial poles, like longitude lines, which intersect the celestial equator at right angles. Right Ascension is given in hours and minutes, with 0 and 24 hours equal to the same point. 24 hours is used because the Earth rotates once every 24 hours, which equals one revolution of the celestial sphere. The zero point for Right Ascension is based on the position of the sun at vernal equinox[21]. In the HIP catalog, the Right Ascension has already been converted into a value in degrees.

These coordinate values can be converted into x,y,z coordinates using a spherical to cartesian conversion formula. Given a radius value (pre-defined at 1000 for this project), the Right Ascension (ra) and Declination (dec) values are plugged into the conversion formula, producing 3D coordinates (x,y,z) at which to place each star in the Unity scene.

```
x = (radius*Mathf.Sin(dec))*Mathf.Cos(ra);
y = radius*Mathf.Cos(dec);
z = (radius*Mathf.Sin(dec))*Mathf.Sin(ra);
```

Figure 12: Code used to convert star coordinates in RA and Dec into 3D coordinates.

The catalog also provides recorded magnitude values for each star. That value can be used to adjust the scale of the star in the simulation, so higher magnitude stars appear larger in the virtual celestial sphere. This magnitude value can also be used to determine which stars to display or omit based on a threshold, for example, displaying only prominent stars (above +5.5 magnitude, 2617 stars from the HIP catalog) or only stars visible to the naked eye (above +6.5 magnitude, 7982 stars). These thresholds are used later in comparing the efficacy of the different methods of implementing the virtual sky.

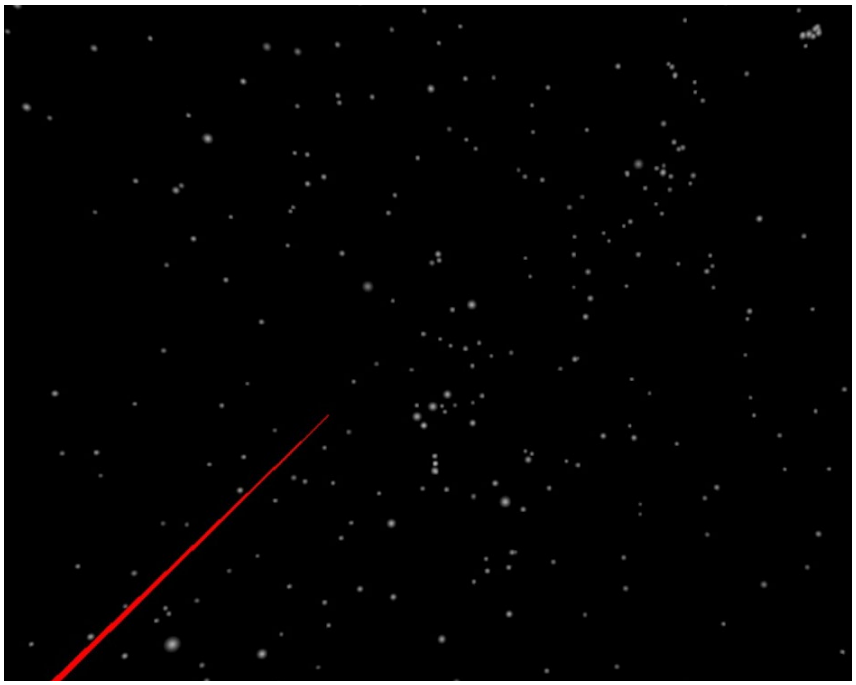


Figure 13: The stars in the constellation of Orion with a red laser pointer, as seen from inside the generated celestial sphere in VR. Stars are placed and sized based on data from the HIP catalog.

The HIP catalog was also used by Stellarium, a free, open-source planetarium software available for desktop and mobile devices[8]. Stellarium developers created a series of constellationship files, which map out which stars are connected to other stars within each constellation[9]. Since these mappings are based on HIP catalog star data, and Stellarium has given permission to use their constellationship files[10], they can also be imported into this simulation project to generate the lines to create the western constellations and Hawaiian starlines. Although these constellationship files could be created manually, being able to use the files from Stellarium saved a lot of time and allowed me to avoid re-doing work that has already been done.

```
KOM 11 32349 37279 37279 37826 37826 36850 36850 28360 28360 28380 28380 25428 25428 23015 23015 24608 24608 28360 32349
KHK 8 27989 26727 26727 27366 27366 24436 24436 25930 25930 25336 25336 27989 26727 26311 26311 25930
MAK 1 17499 17489
MEE 4 60965 59803 59803 59316 59316 61359 61359 60965
NAH 7 67301 65378 65378 62956 62956 59774 59774 54061 54061 53910 53910 58001 58001 59774
IWI 5 11767 54061 67301 69673 69673 65474 60965 65474 60221 61084
MAN 13 85927 86670 86670 87073 87073 86228 86228 84143 84143 82671 82671 82514 82514 82396 82396 81266 81266 80763 80763
NAV 8 102098 97649 97649 91262 91262 102098 95947 97165 97165 100453 100453 102488 102488 102098 102098 100453
LUP 9 677 113881 113881 113963 113963 1067 1067 677 113368 113963 746 677 3419 2081 1067 3419 113881 112731
IWA 4 8886 6686 6686 4427 4427 3179 3179 746
HAN 2 61084 60718 62434 59747
NAK 1 71683 68702
KAO 9 57632 54872 54872 54879 54879 49669 49669 49583 49583 50583 50583 50335 50335 48455 48455 47908 57632 54879
```

Figure 14: Hawaiian Starlines constellationship file from Stellarium, which links star IDs to other stars which they are connected to in the same constellation, making it possible to generate the corresponding lines in the simulation when combined with the HIP catalog star position data.

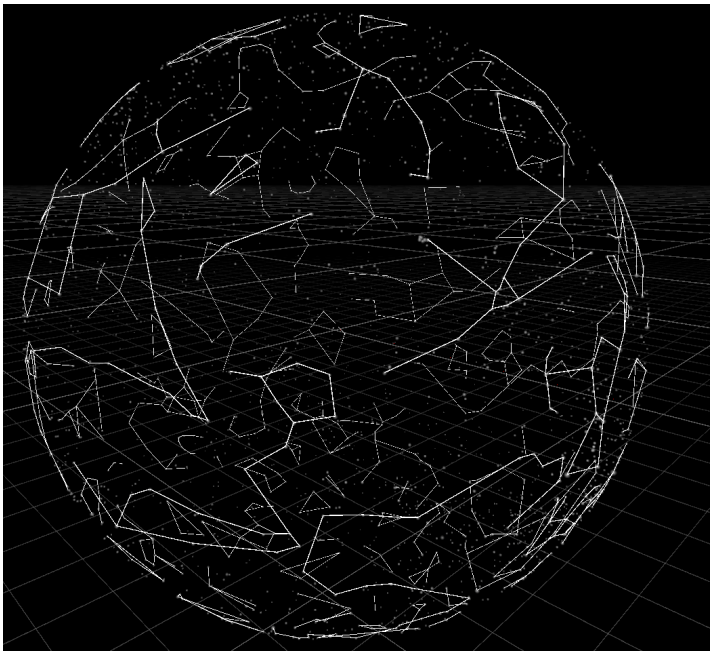
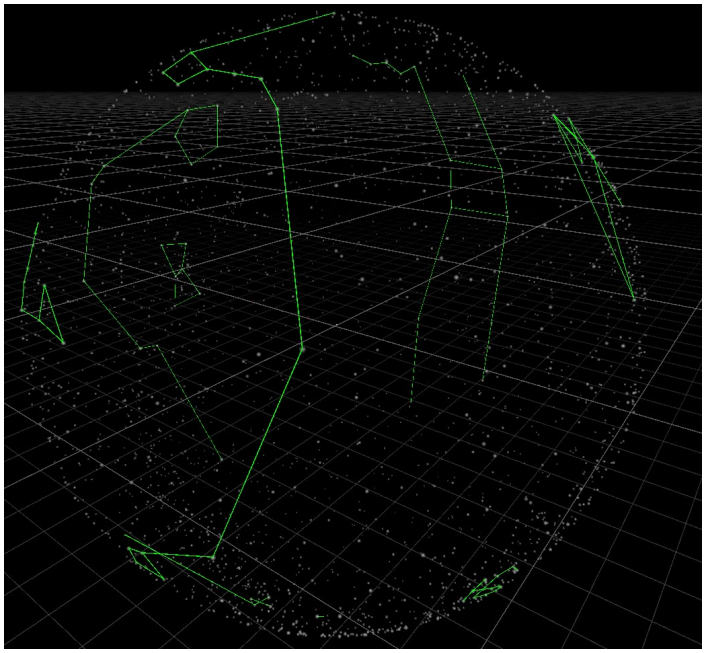



Figure 15 (left): External view of a generated celestial sphere with constellation lines.
Figure 16 (right): External view of a generated celestial sphere with Hawaiian starlines in green.

GameObject Implementation

GameObjects are the fundamental objects in Unity that represent the elements in a game or application, such as characters, props, lights, cameras, etc. Properties are given to GameObjects via components. Depending on what kind of object is needed, different combinations of components are added to a GameObject[14]. Prefabs can be created from GameObjects to be used as a template and reused in the project without having to be reconfigured.

For this implementation, each star is represented by a GameObject prefab with a sphere mesh. A star generator script was created to instantiate each star GameObject in the scene, based on the star catalog position data, and scale the GameObject using the corresponding magnitude value. Constellations and Hawaiian starlines were added to the scene using Unity's LineRenderer class based on the star position data described in the constellationship files. Colliders are generated around these lines and set to detect when the user's laser collides, displaying the constellations western name and Hawaiian name, if applicable.

ParticleSystem Implementation

Unity's built-in ParticleSystem simulates behavior for individual particles and renders many small images to create a visual effect. Each particle in the system represents a graphical element and are all simulated collectively to create impressions of a complete effect. This is useful when rendering dynamic or non-solid objects, such as smoke or fire. The built-in particle

system also allows particles to interact with Unity's underlying physics system, so collision events can be detected on individual particles[15].

The implementation of this system consisted of creating a ParticleSystem in the Unity scene, and then using a script to populate the ParticleSystem Particle array with each star, specifying its position and size. After all star particles have been added to the particle array, the SetParticles function is called on the ParticleSystem, which adds the particles to the scene and renders them. Constellation and starlines are generated in the same way as described above in the GameObject implementation.

DOTS Implementation

The final implementation uses Unity's Data Oriented Tech Stack, or DOTS, a combination of technologies and packages that provide a data-oriented design approach to building games and simulations in Unity[7]. At the time of creating this implementation, the DOTS Entities package version 1.0 (also known as Entity Component System or ECS) is in pre-release, so an official release is still in active development by Unity[25].

Since simulating the full catalog of stars is quite intensive in addition to supporting virtual reality, making the most efficient use of the CPU processing power is important. Developing with data-oriented design (DOD) can aid developers in gaining the performance needed for intensive data-focused applications. Utilizing ECS requires a shift in approach from the object-oriented programming (OOP) paradigm to data-oriented design. Developers must consider what data is needed for their application and how to best structure it in memory so the CPU can efficiently access the data while running the systems to process it. Instead of having individual objects with specific components on each (such as color or position), in DOD these components would be grouped together into arrays so that systems can iterate over the arrays to transform data[20].

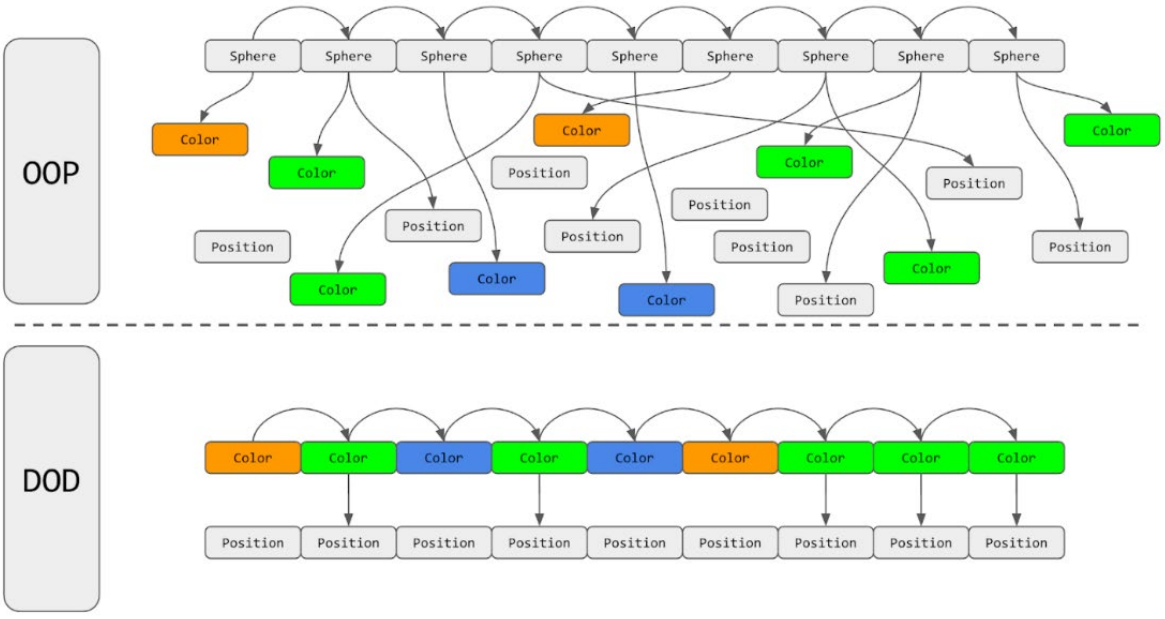


Figure 17: A comparison of how data is processed with OOP and DOD[20]

Figure 17 shows the difference between how the CPU processes an example scene with Sphere classes using OOP and DOD. The Unity DOTS best practices guide describes Figure 17 as follows: “In OOP, the code iterates over an array of Sphere classes to check the Color of each one and set the Position of the green ones. Although the array is packed with contiguous data, it only contains references to Sphere classes, and the actual data can be scattered throughout memory, resulting in cache misses. In DOD, Spheres are decomposed into Color and Position components and packed into buffers, resulting in fewer cache misses and much faster processing”[20].

In Unity’s ECS, an entity is a lightweight, unmanaged alternative to a GameObject, and is essentially a unique identified number. Data for each entity is stored as associate components, usually in the form of struct values. Aspects in ECS are also useful as they provide an object-like wrapper over an entity’s components, defining accessible and modifiable component values, and simplifying component-related code[23].

A collection of entities is known as a world. A world also owns a set of systems, which are run on the main thread, typically once per frame. These systems are used to apply logic to the entities and to transform data[23]. Another benefit of ECS is that it makes it possible for developers to take advantage of Unity’s Burst compiler for performance gains. There are two types of systems in ECS, the first is the ISystem, which is compatible with Burst and therefore provides better performance, while the other is SystemBase, which is slower than ISystem since it is not Burst compiled but allows for use of managed code[22].

Implementing the celestial sphere in ECS is a bit more involved than the other two methods described previously. Essentially, each star is generated as an entity based on the star prefab used in the GameObject implementation. To do this, a subscene must be created in the Unity scene, which will contain the entities. A parent “Starfield” entity is created using an implementation of the Baker Authoring class, which is done in a C# script. This same script contains a MonoBehavior class (Unity’s base class) that have variable declarations for StarfieldRadius (celestial sphere radius value used in converting star coordinates), StarCount (number of stars to generate), and StarPrefab (prefab to create our stars from), which can be accessed and adjusted from the Unity editor. Those values are then used in the Baker class when adding the component to the starfield.

```
public class StarfieldMono : MonoBehaviour
{
    public float StarFieldRadius;
    public int StarCount;
    public GameObject StarPrefab;


}

public class StarFieldBaker : Baker<StarfieldMono>
{
    public override void Bake(StarfieldMono authoring)
    {
        AddComponent(new StarfieldProperties
        {
            StarFieldRadius = authoring.StarFieldRadius,
            StarCount = authoring.StarCount,
            StarPrefab = GetEntity(authoring.StarPrefab)

        });

    }
```

Figure 18: Code to author the Starfield parent entity.

The Baker class creates the starfield entity and sets the values for StarfieldRadius, StarCount, and the StarPrefab using a component struct called “StarfieldProperties”. These values are made accessible by systems and other entities via an

aspect struct called “Starfield Aspect”. Another Baker class is needed to author individual star entities, which is done in a separate system script.

There are two systems at work in this implementation, the first is called “StarfieldGenerator”, which implements ECS ISystem and instantiates the star entities in the subscene. These entities are all first initialized at the zero point in the scene (0,0,0) at a scale of 1. The second system implements SystemBase. This system, called “StarPlacement”, imports the HIP catalog data, and schedules jobs to update the star entities to their correct positions in the scene and adjust the scale of star entities relative to their magnitude.

The constellation lines and Hawaiian starlines are generated using the same methods as the GameObject and ParticleSystem celestial sphere implementations. In the future, these could also be converted to use ECS for additional increased performance.

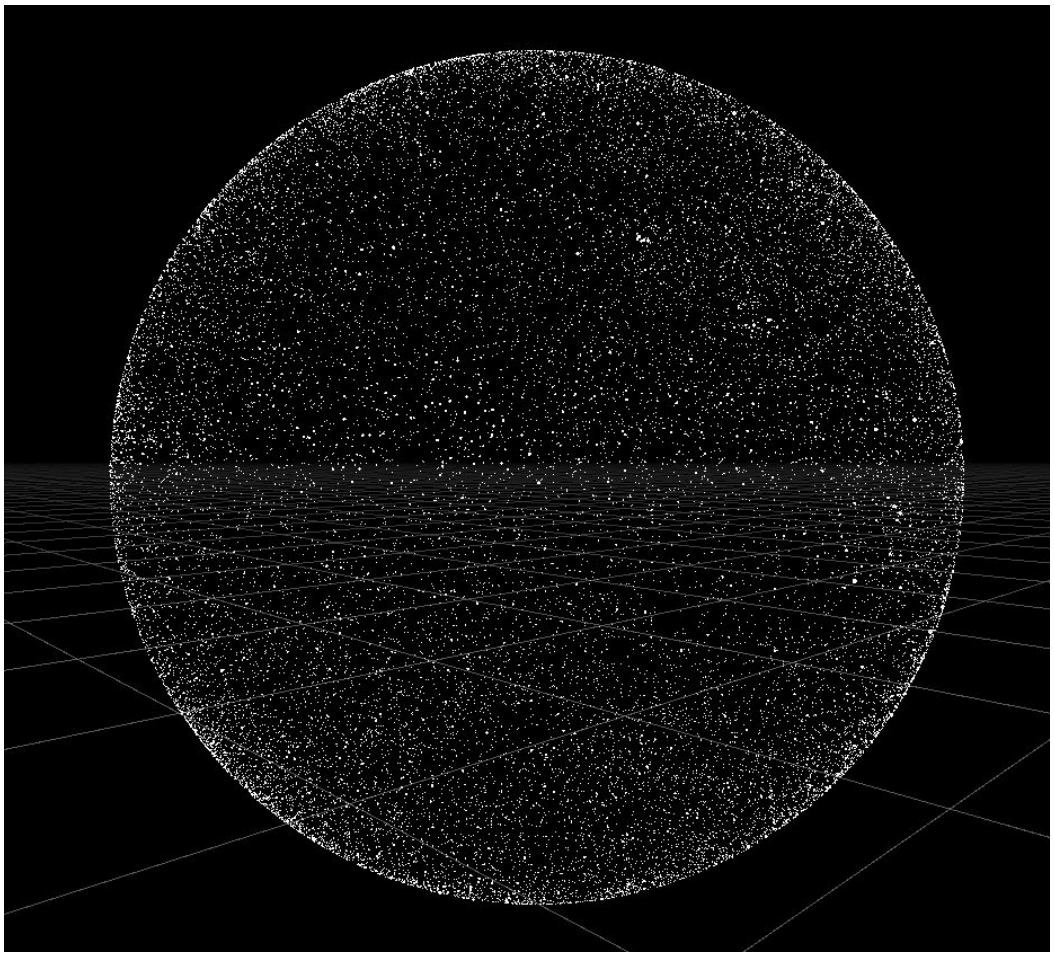


Figure 19: An exterior view of the celestial sphere generated with Unity DOTS. All ~118,000 stars from the HIP catalog are generated as entities in the scene.

Discussion

To compare the efficiency of each implementation, a series of performance tests were conducted by generating varying amounts of stars from the HIP catalog. The first test is generating only prominent stars, or stars over +5.5 in magnitude, which includes 2617 stars in the catalog. The second test generates only stars visible with the naked eye, or stars over +6.5 in magnitude, totaling 7982 stars. The last three tests generate roughly a quarter of the catalog (30,000 stars), half of the catalog (60,000 stars), and the full catalog (117955 stars).

For accurate testing, Unity's Profile Analyzer package was used to obtain data on the amount of time it takes to create each frame in milliseconds. This data was recorded over 2700 frames, which is roughly 30 seconds of time at 90 frames per second (FPS), for each of the five tests and for each implementation method.

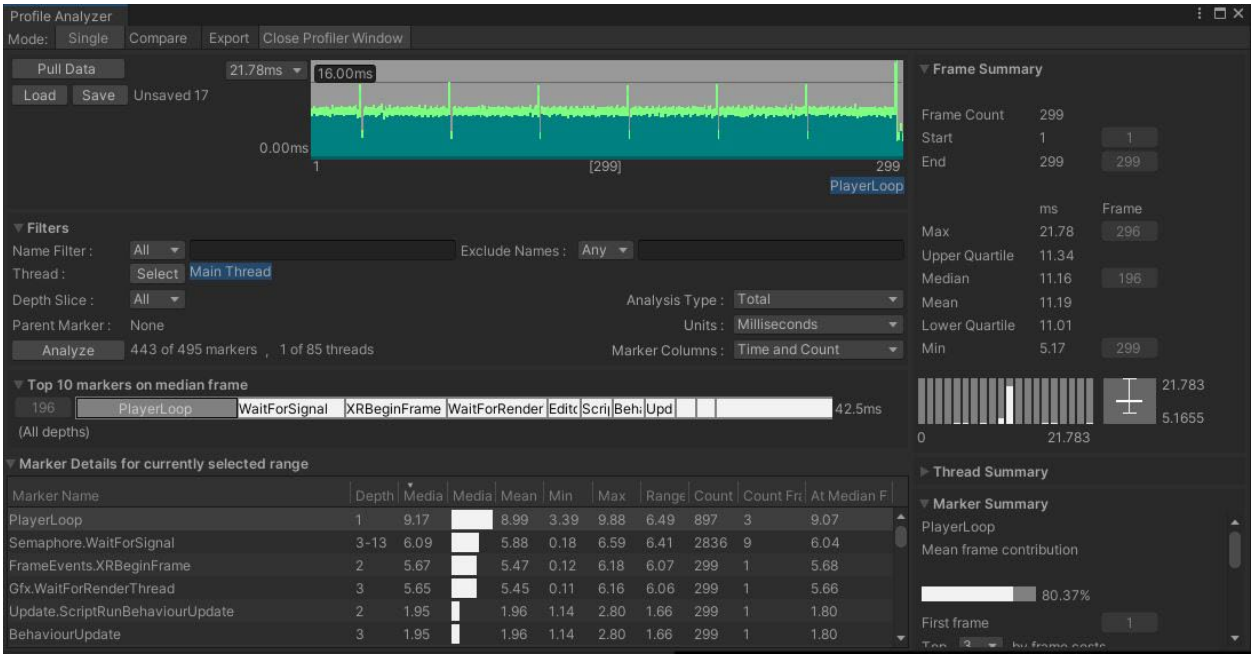


Figure 20: Unity's Profile Analyzer displays statistics for a sample of frames.

Measuring performance by frame time is important because although it's possible to have an average frame rate of 60 FPS, if a game renders 59 frames in 0.75 seconds, but the next frame takes 0.25 seconds to render, players will notice a stutter effect between the two frames[24]. This is why it's important to maintain a "time budget" based on target FPS, so users will get a smoother and more consistent experience. Since we are aiming to hit an FPS target of 90 as needed for VR, we are aiming for 11.1 miliseconds (ms) per frame[24].

The performance tests were conducted using the Vive headset to simulate VR on a computer with the following specifications: Intel i5-9600k CPU, Nvidia GeForce GTX 1080, and 16 GB of RAM. The computer was only running Unity Editor 2022.2.11f1 and the SteamVR application, which were needed to run the simulation and complete the test. Although we ultimately want to analyze the frame times to determine the efficiency of each implementation method, the

differences are stark even when viewing the average FPS for each test, as seen in Figure 21 below.

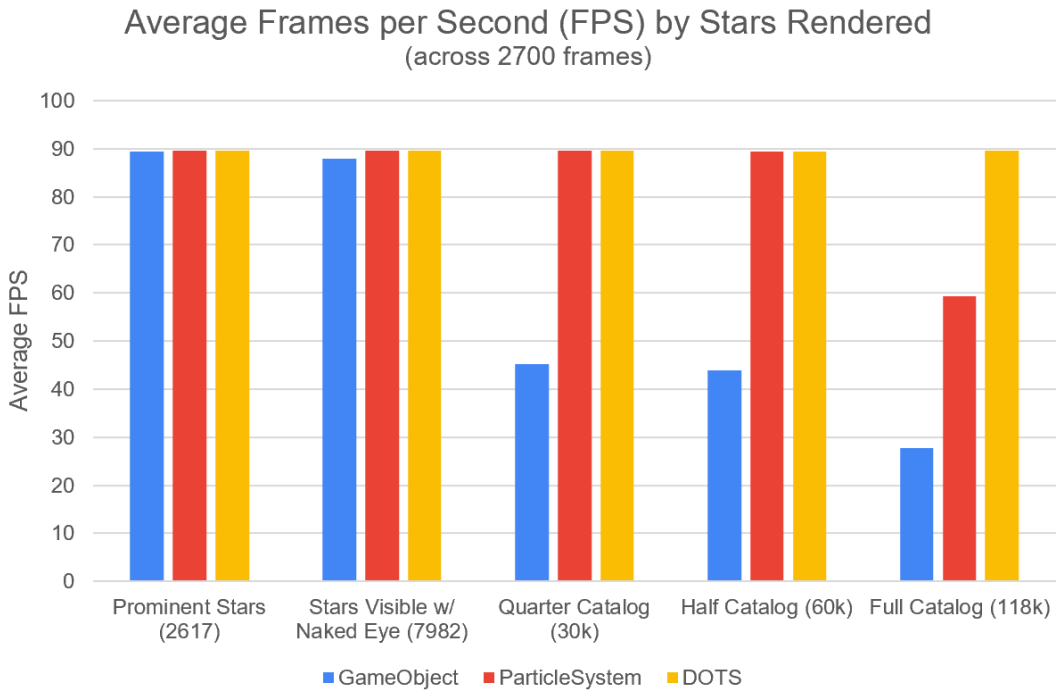
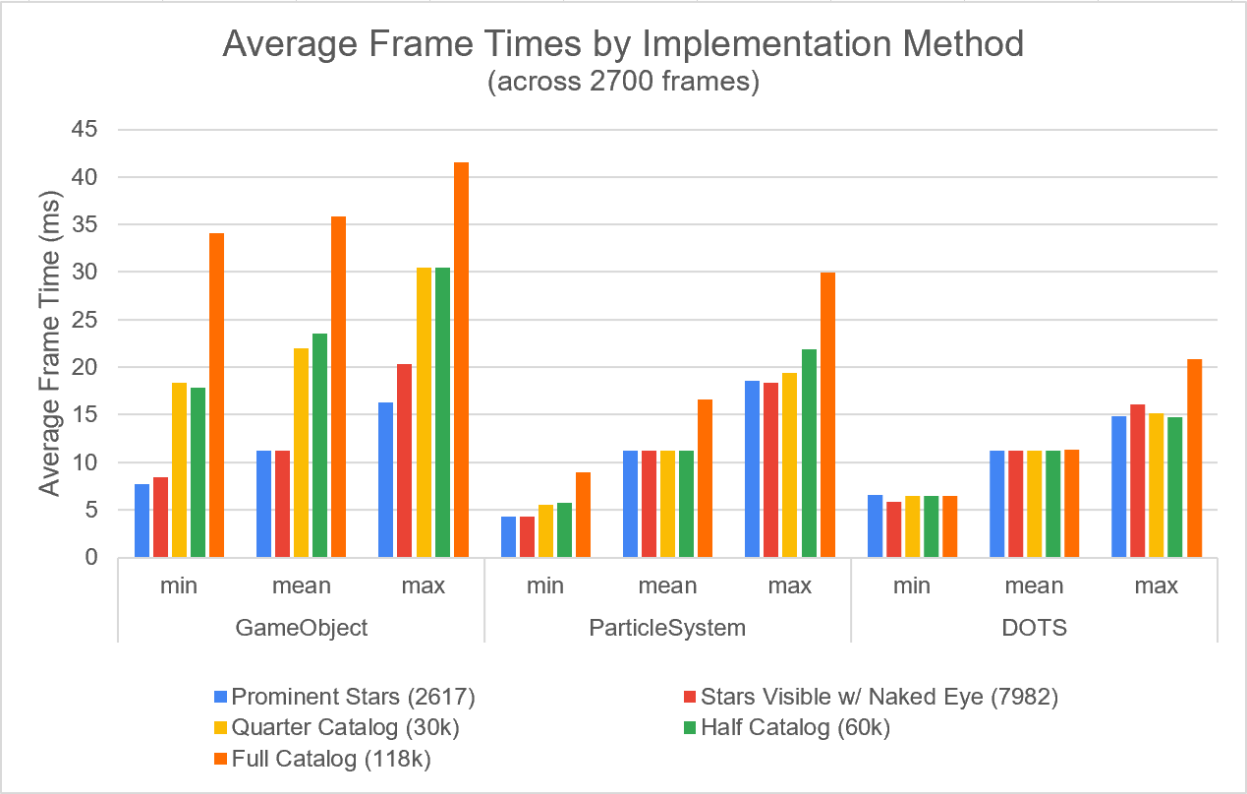


Figure 21: A chart showing the average FPS results for each implementation method, by each of the five tests completed.

When considering the entirety of the data recorded for frame times, we can see similar variation between each implementation, and can discern where each method starts struggling to maintain frame times within the target budget of 11.1 ms.

Based on the test results compared in the chart below in Figure 22, if we are generating only the prominent stars, all methods are about the same, the mean frame times are within the target budge and the implementations run at nearly 90 FPS. By the time we reach a quarter of the catalog, the GameObject method drops off drastically, with an average frame time nearly double the target. The ParticleSystem method is able to keep the target frame times until the entire catalog of stars is rendered. However, the DOTS implementation average frame times remain around our target frame time for all tests, including the full star catalog.



The results show that using Unity's Data Oriented Technology Stack, and designing the implementation from a data-oriented approach provides the required performance benefits needed to keep the application running at 90 FPS in VR, even when rendering nearly 118,000 star entities. For the specific needs of the Kilo Hōkū VR application, we don't need to render the entire catalog, and can stick to only displaying the stars that are visible with the naked eye. Therefore, we could use any of the implementation methods on comparable hardware to the specifications which were used in testing. However, if we want to use a generated celestial sphere on the Quest 2, which has less processing power, using the DOTS implementation would be most efficient based on these findings.

A couple of things to note from performing these tests: first, since they were run in the Unity Editor so the Profile Analyzer package could be utilized to record frame time data, the recorded frame times and FPS were affected by the editor itself, so the performance may be better if the project has been built and run standalone outside of the editor. In testing, I also noticed some overhead from the Unity XR packages which was affecting the frame times beyond the star rendering, which may be worth digging into to determine if there's a way to improve the time by adjusting settings in the XR package. Lastly, test results were varying depending on the direction the user was facing, this is due to the non-homogeneity of the stars in the sky. This was more evident during the half and full catalog tests, where if the user is looking at a section of the sky with a larger number of stars (such as around the milky way), frame times would increase noticeably. To try to maintain accuracy in testing, I moved the headset in a consistent pattern for all tests.

Reflection

In developing the different implementations of the virtual night sky, I learned more about how each implementation method (GameObjects, ParticleSystem, DOTS) works in Unity from a technical standpoint, and how to manipulate each of them to display specific data. I also learned how to effectively use the Profiler and Profile Analyzer in Unity to measure the performance of applications and which techniques are best to measure performance for VR applications, such as using frame times over frames per second.

Prior to this project, I hadn't had much exposure to data-oriented design, but through this project I learned why it is important and how to use it to improve application performance. Since Unity DOTS is relatively new to the Unity Engine, and the Pre-release version only just became available on November 28, 2022, many structural methods and syntax changes had been made between the experimental and pre-release versions, so a lot of information regarding DOTS was outdated outside of Unity's documentation. This made creating the DOTS implementation an interesting challenge for me, but I learned so much from the experience and have a deeper understanding of data-oriented design now. I plan to continue developing the DOTS implementation from this project as DOTS shifts to an official release, and more features are added which could be useful for Kilo Hōkū VR, like the Unity physics and netcode for entities packages.

There were a number of features I wanted to include in this project, which I am planning to implement in the future to enhance the celestial sphere beyond this DOTS implementation. Since there are a number of specific stars that are important to know how to find in Hawaiian wayfinding, such as the stars Hōkūleʻa (Arcturus) and Hikianalia (Spica), I want to make these stars interactable so users can highlight them like they can with the constellations and starlines, and display relevant information about them. I am hoping that Unity physics for entities is stabilized soon with the Unity LTS release so that this is not too cumbersome to do. At the moment, because entities and regular Unity GameObjects exist in different "realms" of a Unity project, and Unity's XR packages are based of off GameObject implementations, a lot of additional work is needed in referencing and translating entities to match user movements and to trigger interactions between entities and GameObjects.

Additional features could also be added to make the sky even more useful as a learning tool, such as adding the moon and planets as entities, as these are also important to distinguish in wayfinding and the planets were important culturally, known as hoku hele, "Traveling Stars", or hoku 'ae'a, "Wandering Stars"[21], since they move across the sky differently than the movement of the stars. Simulating weather effects which partially occlude the celestial sphere could also be helpful in testing learners' recognition skills.

In conclusion, this project has shown that a functional version of a generated celestial sphere has been proven effective and is able to run in virtual reality. By utilizing Unity DOTS, this is feasible with a large-scale data set like the HIP star catalog. My Kilo Hōkū development team and I can use the findings of this project to enhance the next version of the Kilo Hōkū VR application by replacing the original virtual celestial sphere with a generated one using the DOTS implementation techniques. We will be able to use this DOTS version to display all stars visible to the naked eye in virtual reality on the Quest 2 headset, ultimately making the application more accurate and effective as a learning tool for classrooms.

Acknowledgments

Many thanks to Dr. Jason Leigh for advising me throughout the development of this project, and for the recommendation to experiment with Unity DOTS. Additional thanks to my Kilo Hōkū VR team, Patrick Karjala, Kari Noe, and Dean Lodes, for the encouragement on this project. Lastly, I wanted to thank Johnny Thompson, Unity Developer from Turbo Makes Games, for his Unity ECS 1.0 videos, which I found very helpful in first getting acquainted with DOTS.

References

1) Patrick Karjala, Dean Lodes, Kari Noe, Anna Sikkink, Jason Leigh; Kilo Hōkū—Experiencing Hawaiian, Non-Instrument Open Ocean Navigation through Virtual Reality. *Presence: Teleoperators and Virtual Environments* 2017; 26 (3): 264–280. DOI: https://doi.org/10.1162/pres_a_00301

2) “Kilo Hōkū VR.” (n.d.). Retrieved from https://kilohokuvr.com/

3) Kari Noe, Patrick Karjala, Anna Sikkink, and Dean Lodes. 2020. A Demonstration of Kilo Hōkū-Implementing Hawaiian Star Navigation Methods in Virtual Reality. In Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems (CHI EA '20). Association for Computing Machinery, New York, NY, USA, 1–4. https://doi.org/10.1145/3334480.3383156

4) “What are the system requirements?” VIVE Support, System Requirements. (n.d.). Retrieved from https://www.vive.com/us/support/vive/category_howto/what-are-the-system-requirements.html

5) “How Powerful is Oculus Quest 2 [Comparison with the Quest, Go, the PC and Consoles]”. (n.d.). ServReality. Retrieved from https://servreality.com/news/how-powerful-oculus-quest-2-comparison-with-the-quest-go-the-pc-and-consoles

6) VR sickness: Wang, Jialin & Shi, Rongkai & Zheng, Wenxuan & Xie, Weijie & Kao, Dominic & Liang, Hai-Ning. (2023). Effect of Frame Rate on User Experience, Performance, and Simulator Sickness in Virtual Reality. IEEE Transactions on Visualization and Computer Graphics. PP. 1-11. DOI: 10.1109/TVCG.2023.3247057.

7) Bond, Jeremy Gibson. (2022). *Introduction to Game Design, Prototyping, and Development: From Concept to Playable Game with Unity and C#* (3rd ed.). Addison-Wesley Professional. ISBN-13: 9780136619949

8) Zotti, G., Hoffmann, S. M. ., Wolf, A. ., Chéreau, F. ., & Chéreau, G. . (2021). The Simulated Sky: Stellarium for Cultural Astronomy Research. Journal of Skyscape Archaeology, 6(2), 221–258. https://doi.org/10.1558/jsa.17822

9) Zotti, Georg, Wolf, Alexander. (2023). *Stellarium 23.1 User Guide*. Retrieved from https://github.com/Stellarium/stellarium/releases/download/v23.1/stellarium_user_guide-23.1-1.pdf

10) "Constellationship Licensing." (2020). GitHub – Stellarium/stellarium. Retrieved from https://github.com/Stellarium/stellarium/discussions/790

11) Polynesian Voyaging Society. (n.d.). "Hawaiian Star Lines and Names for Stars." Hawaiian Voyaging Traditions. Retrieved from https://archive.hokulea.com/ike/hookele/hawaiian_star_lines.html

12) van Leeuwen F. (2007). Validation of the new Hipparcos reduction. A&A 474 (2) 653-664. DOI: 10.1051/0004-6361:20078357

13) "Hipparcos, the New Reduction: The Astrometric Catalogue." (n.d.). CDS VizieR. Retrieved from http://vizier.nao.ac.jp/viz-bin/VizieR-3?-source=I/311/hip2

14) "GameObjects." (2017). Unity Documentation. Retrieved from https://docs.unity3d.com/Manual/GameObjects.html

15) "Particle systems." (n.d.). Unity Documentation. Retrieved from https://docs.unity3d.com/Manual/ParticleSystems.html

16) Tonkin, Stephen. (2018). "Astronomy star catalogues: which to use and when." Sky at Night Magazine. Retrieved from https://www.skyatnightmagazine.com/advice/skills/astronomy-star-catalogues-which-to-use-and-when/

17) Christian, C., & Roy, J. (2017). A Question and Answer Guide to Astronomy (2nd ed.). Cambridge: Cambridge University Press. DOI:10.1017/9781316681558

18) King, Bob. (2019). "Right Ascension & Declination: Celestial Coordinates for Beginners." Sky & Telescope. Retrieved from https://skyandtelescope.org/astronomy-resources/right-ascension-declination-celestial-coordinates/

19) Chromey, F. (2010). To Measure the Sky: An Introduction to Observational Astronomy. Cambridge: Cambridge University Press. DOI:10.1017/CBO9780511794810

20) "Part 1: Understanding data-oriented design." (2021). Unity Learn. Retrieved from https://learn.unity.com/tutorial/part-1-understand-data-oriented-design-1?courseId=60132919edbc2a56f9d439c3&uv=2020.1

21) Polynesian Voyaging Society. (n.d.). "The Celestial Sphere." Hawaiian Voyaging Traditions. Retrieved from https://archive.hokulea.com/ike/hookele/celestial_sphere.html

22) “Systems Comparison.” (2023). Unity Manual: Entities 1.0.0-pre.65. Retrieved from https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/systems-comparison.html

23) Will, Brian. “Entities and components.” (2023). GitHub – Unity-Technologies/EntitiyComponentSystemSamples. Retrieved from https://github.com/Unity-Technologies/EntityComponentSystemSamples/blob/master/Docs/entities-components.md

24) “Performance Profiling Tips for Game Developers.” (n.d.). Unity Best Practices. Retrieved from https://unity.com/how-to/best-practices-for-profiling-game-performance

25) “DOTS Roadmap” (n.d.). Unity Platform Roadmaps. Retrieved from https://unity.com/roadmap/unity-platform/dots