
Simulations and Machine Learning for Parachute Navigation

Major Qualifying Project

Advisors:

PROFESSOR OREN MANGOUBI
PROFESSOR RANDY PAFFENROTH

Sponsoring Co-Advisor:

DR. GREGORY NOETSCHER (CCDC-SC)

Written By:

GRACE MALABANTI
JOSEPH SCHEUFELE
JULIETTE SPITAEELS



WPI

A Major Qualifying Project
WORCESTER POLYTECHNIC INSTITUTE

This report represents the work of one or more WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on the web without editorial or peer review.

AUGUST 25, 2021 - APRIL 27, 2022

Submitted to the Faculty of the Worcester Polytechnic Institute in partial fulfillment of
the requirements for the Degree of Bachelor of Science in Mathematical Sciences &
Computer Science.

1. Abstract

In the military, supplies are critical, and a common method for supply deliveries are parafoil parachutes. The accuracy of these supply drops is important to the success and safety of operations and supporting American troops in their missions [1]. The most common way of tracking and navigating these parafoils is via Global Positioning System (GPS) [2]. However, signals from GPS satellites can suffer from disruptions due to interference like jamming or environmental factors. The drawbacks of GPS signaling have led to research in alternative forms of navigation that do not rely on external signals. One in particular is the area of Guided-Airdrop Vision-Based Navigation (GAVN) [1].

In recent years there has been an increase in machine learning (ML) and Data Science (DS) research [3]. Some of the most popular ML models for digital image processing are neural networks (NN), a type of deep learning model that can be trained using examples in order to make predictions on new data [4]. Deep learning algorithms have the potential to pinpoint the location of cargo parafoils by using the visual feed from a camera mounted to the parachute, as opposed to external GPS signaling. Accordingly, the research in this MQP focuses on the development of a machine learning-based alternative to GPS navigation for cargo parafoils.

In order for neural networks to make meaningful predictions, they require large amounts of training data [4]. However, collecting images from parafoil drops is expensive and resource-intensive. Therefore, we first created a virtual simulator for parachute drops. This simulator is able to generate the large quantities of labeled image data required by our proposed deep learning methods in an inexpensive manner. Next, after collecting data using our simulator, we were able to apply a variety of preprocessing methods to the images and test different neural network structures in an attempt to predict location from the images. In particular, we studied the effect of multi-task deep learning models on the performance of vision-based navigation in realistic parafoil scenarios. Additionally, we found through our experiments that multi-task learning yield more consistent error results compared to single-task networks for our vision-based navigation models.

2. Acknowledgements

Our project would not have been possible without support from our sponsors, advisors, and departments. First, we want to thank Dr. Greg Noetscher for providing us with his invaluable expertise, which guided our project throughout the entire process. Next, we want to thank DEVCOM-SC for sponsoring our work; without their support our team would not have had the amazing opportunity to learn from an interesting, impactful project. We would also like to thank our advisors Professor Randy Paffenroth and Professor Oren Mangoubi. Both our advisors guided us by providing not only their knowledge in the areas of Math, Data Science, and Computer Science, but also teaching us the process of research and collaboration. Additionally, received helpful guidance along way the way from WPI's Academic Research and Computing office and would like to thank James Kingsley for all his help learning about super-computing. Finally, we would like to thank the departments of Mathematical Sciences and Computer Science at Worcester Polytechnic Institute for preparing us for this project through the variety of classes and coursework during our undergraduate experience.

3. Executive Summary

In the military, supplies are critical, and a common method for supply deliveries are parafoil parachutes. With aid from these parachutes, crates of supplies can be dropped by aircrafts almost anywhere in the world. Sometimes these parachutes carry fragile cargo like medicine and tools, while other times multi-ton machinery is dropped from the sky. No matter what these parachutes carry, the accuracy of these supply drops is important to the success and safety of operations and supporting Soldiers in their missions.

The most common way of tracking and navigating these parafoils is via Global Positioning System (GPS). Despite GPS being the industry standard for location tracking, it has its limitations and drawbacks. Signals from GPS satellites are not always reliable and can be spoofed or jammed by adversaries, or interrupted by strong weather. These drawbacks of GPS have led to research in alternative forms of navigation that do not rely on external signals but can compete in accuracy with GPS. These methods are called Guided-Airdrop Vision-Based Navigation (GAVN) [1].

Alternative forms of navigation are an active area of research and the U.S. Army Combat Capabilities Development Command Soldier Center (DEVCOM-SC), the sponsor of the project, is one prominent organization participating in GAVN innovation. DEVCOM-SC is comprised of researchers, engineers, and technologists who work to provide the Soldier with a wide range of capabilities. One of their core domains is airdrop/aerial delivery. To this end, they have sponsored our team to aid their research and development of GAVN technologies.

In recent years there has been an increase in machine learning (ML) and data science (DS) research. One focus of this research has been advanced image processing technology. This advanced image processing technology has the potential to pinpoint the location of cargo parafoils mid-flight. This could allow for fully self-contained navigation systems which do not rely on GPS, and in turn avoid the pitfalls of external signaling. Attempts have been made to develop this technology, but they have not yet been able to perform with better accuracy than traditional GPS navigation [1].

Advanced image processing technologies are based on neural networks (NN), a kind of machine learning model that is able to make predictions after being trained on thousands, and sometimes millions, of data points. The previous MQP team began work on this project using data collected from in-the-field parachute drops. They found that these drops did not produce enough data to train machine learning models [5]. Collecting data from parachute drops is expensive, so it was clear that an alternate way to col-

lect data was required to move forward, one that was both inexpensive and versatile.

Virtual simulations currently have many training applications within the military such as soldiers in field exercises or pilots in flight drills. DEVCOM-SC wanted us to apply this technology to collect aerial imaging data from parachute drops. Our goal was to create a simulator that could replicate parachute drops and collect aerial images and their locations anywhere in the world. Our needs were met by a video game design program, the Unreal Engine (UE) in combination with a the satellite imaging plug in, Cesium [6].

To create realistic drop paths for parachute simulations we used coordinates collected from a series of in-the-field DEVCOM-SC test drops. We developed code that could either recreate a drop path in its original location, or take parachute drop coordinates from one of the real drops and transform the path to a new location, determined by a user-defined landing destination. From the simulator we were able to collect many images labels with their coordinate locations. Examples of simulated images from a variety of locations around the world are shown in Figure 3.1.



Figure 3.1: Synthetic aerial images captured using our simulator. Locations clockwise from the top left: Lunar Craters (NV), Amazon Rainforest, Yuma Proving Ground, New York City.

With this simulated data we were able to begin designing our machine learning models. Our goal was to give the neural network two images, and have it predict the distance between them. Initially, we wanted to predict the change in position of all three dimensions, but soon narrowed this focus down to predicting only the change in altitude, as it was the most meaningful measurement collected from our simulator. With this goal in mind we tried a variety of datasets of different sizes and structures, but eventually chose one that satisfied the needs of our problem. We collected data from eight simulated drops, set over the Yuma Proving Grounds, with images taken each second. Then we paired up every picture within each drop that had a difference of less than 200 meters in altitude, but greater than zero meters. The maximum restriction ensured that we did not have images that were so far apart that they had no overlapping features. The minimum restriction prevented occasional upward changes in altitude and zero difference values that were problematic for calculating percent error. This left us with a dataset consisting of about 450,000 image pairs and the initial image's altitude, latitude, and longitude. We used random samples of 75,000 samples at a time to train and test our models, and set aside 25,000 samples to ultimately be used for a final validation of our network.

We applied a series of preprocessing steps to our datasets to prepare them for machine learning applications. As a baseline, we used images that we grayscaled and consistently sized to 240×426 pixels. These steps were important for the consistency and allowed us to easily flatten the images. After these preprocessing steps were applied, each sample consisted of two images, along with the initial picture's coordinates, therefore each sample consisted of 204,483 variables.

We determined the performance of our ML models based on three different metrics, mean squared (MSE), root mean squared error (RMSE), and percent error for the difference in the true altitude from the predicted. We also implemented validation testing, to assure the accuracy of our model. We explored the effects of single-task versus multi-task neural networks. Single-task networks are a model where the loss function is guided by one error and the model returns a single output, the desired prediction value. In our case our single output network was guided by the MSE for the altitude difference between the image pair, which was also the value returned by our neural network. Multi-task neural networks, as the name implies, try to predict multiple variables at once, as opposed to a single desired value. The training of multi-task models is driven by the loss function incorporating the

multiple values of interest. In our case, we defined our multi-task loss function as average MSE of the three different dimensions: latitude, longitude, and altitude.

Our best model was a multi-task neural network [7] consisted of 9 layers: input, output and seven hidden layers. The input layer reduces the flattened images from their original size to 1024 neurons. Each consecutive layer reduces the dimensionality of the network by a factor of two until the output layer which will either convert a vector of 8 neurons to a single scalar (to represent the change in altitude for a single-task model) or a 3x1 vector (to represent the change in latitude, longitude, and altitude for the multi-task model). We set the set the hyperparameters as 750 epochs, a batch size of 128 images, and a step size of 0.001. These settings balanced our needs for precision but also training time, as these two goals conflict. We found applying pairwise equalization was a very helpful preprocessing step and Leaky ReLU was the best performing activation function. The results of the neural network experiments we conducted are displayed in Figure 3.1.

Variables			Testing Error		Validation Error	
Single or multi?	Equalized	Activation Function	RMSE	Percent Error	RMSE	Percent Error
Single	None	ReLU	25.607	0.489	78.474	1.379
Multi	None	ReLU	25.664	0.484	77.809	1.005
Single	Pairwise	ReLU	18.540	0.262	79.264	3.249
Multi	Pairwise	ReLU	18.673	0.291	18.907	0.294
Single	Pairwise	GELU	18.370	0.261	78.867	3.226
Multi	Pairwise	GELU	18.510	0.283	18.697	0.290
Single	Pairwise	Leaky ReLU	17.952	0.257	79.012	3.222
Multi	Pairwise	Leaky ReLU	17.528	0.264	17.493	0.277

Table 3.1: Report of validation error metrics given for the different network architectures we tested. Based on our cross-validation results we determined our best model was the nine layer multi-task network, with Leaky ReLU as the activation function. The best result is displayed in bold.

Contents

1	Abstract	i
2	Acknowledgements	ii
3	Executive Summary	iii
4	Introduction	1
4.1	Deliverables	3
5	Background	4
5.1	DEVCOM-SC and Previous Work	4
5.2	Simulator	6
5.3	Image Processing	9
5.4	Machine Learning	13
5.4.1	Supervised Learning	14
5.4.2	Classification and Regression	15
5.4.3	Evaluation Metrics	16
5.4.4	Cross-Validation	17
5.4.5	Random Forests	17
5.4.6	Neural Networks	19
5.4.6.1	Activation Functions	22
5.4.6.2	Training a Neural Network	25
5.4.6.3	Single-task and Multi-task Neural Networks	26
5.5	High Performance Computing	26
5.6	Weights & Biases	28
6	Methodology	29
6.1	Simulator	29
6.1.1	First Attempt Using AirSim	30
6.1.2	Final Method Using Cesium	32
6.1.3	Additional Settings	37
6.2	Dataset	39
6.2.1	Predictors and Labels	39
6.2.2	Training Subsets	41
6.2.3	Preprocessing	41
6.3	High Performance Computing	42
6.4	Neural Networks	43
6.4.1	Baseline Network	44

6.4.2	Error Metrics	46
6.4.3	Single-task and Multi-task Learning	47
6.4.4	Model Validation	47
6.4.5	Additional Architectures	47
6.4.6	Cross-Validation	48
7	Results	49
7.1	Simulator	49
7.2	Datasets	53
7.2.1	Preprocessing	53
7.2.1.1	Grayscale	53
7.2.1.2	Resizing	54
7.2.1.3	Equalization	54
7.2.2	Distance Limits	57
7.2.3	Deliverables	58
7.3	Neural Network	59
7.3.1	Epochs	59
7.3.2	Activation Functions	60
7.3.3	Layers	67
7.3.4	Cross-Validation	70
8	Future Work	72
8.1	Simulator	72
8.2	Preprocessing	73
8.3	Datasets	73
8.4	Neural Network	73
9	Conclusion	75
10	Appendices	76
10.1	Simulator Manual	76

List of Figures

3.1	Synthetic aerial images captured using our simulator. Locations clockwise from the top left: Lunar Craters (NV), Amazon Rainforest, Yuma Proving Ground, New York City.	iv
5.1	One of DEVCOM-SC's core domains is airdrop/aerial delivery. For deliveries, a plane must drop the supplies via parachute. These parachutes are unmanned and therefore are equipped with remote steering equipment to navigate them to the correct landing point [18]. This image is an example of a JPAD test drop [19].	5
5.2	The Unreal Engine with the Cesium plug-in allows for the creation of a 3D geospatial ecosystem, based on satellite imaging of Earth. This is an image of the whole globe rendered in the Unreal Engine.	7
5.3	The Unreal Engine with the Cesium plug-in allows for the creation of a 3D geospatial ecosystem, based on satellite imaging of Earth. The mountain range seen in this image is rendered with realistic sunlight, landscapes, and associated coordinates from Cesium.	8
5.4	Seen here, UE4 has a camera object that can be placed into the environment. This camera can then be programmed to follow a flight path and capture images and locations.	8
5.5	Images taken with UE4 default to color. From top left, (a) is a raw colorized image from the unreal engine, (b) is the same image grayscale, (c) is the final intensity equalized image, (d) is the histogram of the pixels with the intensity of each color (red, green, blue), (e) is the histogram of the grayscale image before equalization and (f) is the histogram of the equalized image.	10
5.6	Blurring may be used to smooth out noise within an image. Blurring is especially useful for preserving shapes and edges. Image (a) is a grayscale image of the Yuma Proving Grounds and image (b) is the same image with gaussian blurring applied.	12

- 5.7 Fourier and Wavelet compression are similar methods for reducing complexity in images. They both work by performing calculations on the images and finding the highest value coefficients that represent the image. The rest of the values can be replaced with zero. By applying either of these methods, it reduces complexity and difficulty of calculations [33]. Image (a) is an example of Fourier compression applied to 5.6a and image (b) is wavelet compression applied to the same image. . 13
- 5.8 Classification predicts, or classifies, discrete data. It assigns the data to a class that is predefined. Regression predicts continuous values by finding relationships between dependent and independent variables [39]. The image on the left is of a classification example, and the image on the right is of a regression example [40]. 15
- 5.9 A decision tree is a kind of ML method that uses linear divisions, often called cuts, to create regions for predictions. The cuts indicate decisions that are used to guide a prediction of a testing point. This method works well on both regression and classification problems [41]. This image represents the basic node and branch structure of a decision tree [44]. 18
- 5.10 Random forests are a group of decision trees that work together as an ensemble to make predictions. Each tree is slightly different since it is built with a bootstrapped sample of data and each split decides which variable to cut on from only a subset of the predictors sampled at each node [41]. This image illustrates the structure of a random forest working together to make a prediction [44]. 19
- 5.11 This figure depicts a 3 layered neural network, including the input, hidden and output layers. The input neurons of the network are denoted as x_i , the hidden neurons of the network are denoted h_i and the final output is denoted as \hat{y} . The first layer of the network depends on the raw data, such as flattened image pairs, being passed as input. Notice that the value within each neuron is passed to every neuron of the next layer via the arrows. 20

- 5.12 This figure shows a simple network made up of 2 input scalar cells, x_0 and x_1 , and one hidden scalar cell h_1 . Each of the cells that are input to h_1 are given a scalar weight, w_1 and w_2 respectively. Each neuron that is not part of the input layer also carries a unique bias value, b_i 20
- 5.13 This is a diagram of a more complex network. There is one input layer of size 3, one hidden layer of size 2, and one output layer of size 1. x_0 , x_1 , and x_2 are the inputs, $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ are the weight matrices, h_1 , and h_2 are the hidden neurons, b_1 , and b_2 are the biases, and y is the output. 21
- 5.14 The main idea of an activation function is to set a threshold for the value of h_i to either exceed and pass on its information, or not exceed and not pass on its information. The use of activation functions also allows neural networks to create non-linear prediction models [45]. 22
- 5.15 The graph of the ReLU activation function. This activation function works by setting all values less than zero to zero, and any input above zero remains the same [48]. 23
- 5.16 Leaky ReLU works very much like ReLU, however Leaky ReLU expects to be given a positive slope which will replace the input if it is less than 0 [47]. The effect of having this positive slope is that all the neurons will fire in the network. This image is a graph of the Leaky ReLU activation function [49]. 24
- 5.17 With GELU, inputs are weighted by their percentile rather than sign, like ReLU [50]. Instead of having a cut off at $x = 0$, GELU uses the cumulative distribution function of the standard normal distribution. This image is a graph of the GELU activation function [51]. 25
- 6.1 The Unreal Engine Editor window when you first create a project. The scene is empty except for a blank ground tile. The engine provides tools for adding in objects and actors, and coding in functionality. 30
- 6.2 The Unreal Engine Editor window with the first landscape we used. The scene contains mountains, roads, and lake. The scene is limited to a tile which was a drawback. 31
- 6.3 The Unreal Engine Editor window with the Cesium plug-in added in. 32

6.4	The object within the Unreal Engine with a camera attached to it. This object mimics a cargo parachute and its attached camera.	33
6.5	On the left is the spline path from a DEVCOM-SC parachute drop in Arizona. On the right is the same spline path transformed to New York City. This was done through MATLAB code and allowed us to collect realistic aerial imaging data from anywhere in the world.	35
6.6	The blueprint code to move the object along the spline.	35
6.7	The blueprint code to collect locations upon clicked the F key. When the F key is clicked a loop will begin with a user-defined delay between each iteration. Within the loop, the location of the object is collected and saved to a log. The locations is in the form of latitude, longitude, and altitude	36
6.8	The blueprint code to collect images upon clicked the F key. When the the F key is clicked a loop will begin with a user-defined delay between each iteration. Within the loop, the command to take a screenshot is called. The screenshot is taken from the camera perspective. The images are saved in the screenshots folder of the project.	37
6.9	An image captured from our simulator in Arizona. The Sun's position can be changed using a built-in parameter to simulate different times of day in any virtual location. This allows the user to change the time of day in the simulator, in order to collect more diverse imaging data. The sun in this image is set to sunset	38
6.10	The first image in the dataset. This image was captured from our simulator following a DEVCOM-SC flight path.	40
6.11	The second image in the dataset. This image was captured from our simulator following a DEVCOM-SC flight path.	41
6.12	The graph of the ReLU activation function. This activation function works by setting all values less than zero to zero, and any input above zero remains the same.	44
7.1	The image on the left is collected from a DEVCOM-SC parachute drop over the Yuma Proving Grounds in Arizona and the image on the right is collected from the same location within our simulator.	50

7.2	Synthetic aerial images captured from our simulator. Locations clockwise from the top left: Lunar Craters (NV), Amazon Rainforest, Yuma Proving Ground, New York City. The Yuma Proving ground example is taken at dusk with volumetric fog. This picture demonstrates the diverse functionality of our simulator.	51
7.3	Shown are the four different variations of the Yuma Proving ground that can be made by combining fog and time of day. Clockwise from the top left: midday without fog, midday with fog, dusk with fog, and dusk without fog. This image demonstrates the different visual effects that can be achieved using these settings.	52
7.4	The image above shows an example of histogram equalization on a grayscale image collected using our simulator. The technique increases contrast in the image.	56
7.5	This pair of images has more than a 200m difference in altitude: the first picture is very far away from the ground while another is very close. Because of this there may not be any shared features between the images.	58
7.6	The graph is an example of convergent training and testing errors due to insufficient epoch values and too few training samples available to the network. We were able to identify 750 epochs was sufficient for our final training set, which avoided graphs of this nature.	60
7.7	The testing loss for our datasets run with the ReLU activation function. The datasets show a smooth convergence to a consistent MSE of about 350.	61
7.8	The testing loss for our datasets run with the Leaky ReLU activation function. The datasets show a smooth convergence to a consistent MSE of about 325.	63
7.9	The testing loss for our datasets run with the GELU activation function. The datasets show a smooth convergence to a consistent MSE of about 340.	65

List of Tables

3.1	Report of validation error metrics given for the different network architectures we tested. Based on our cross-validation results we determined our best model was the nine layer multi-task network, with Leaky ReLU as the activation function. The best result is displayed in bold.	vi
6.1	An example of a datapoint from our input.	40
6.2	The layer architecture for our baseline multi-task NN. This model was used as the basis for our experiments before we performed experiments altering the architecture	45
6.3	The layer architecture for our baseline single-task NN. This model was used as the basis for our experiments before we performed experiments altering the architecture	45
6.4	The hyperparameters for the neural network. These parameters were set for the whole project.	46
7.1	This table demonstrates the ineffectiveness of the CLAHE method as a preprocessing method for our data. As it can be seen, the single and multi-task networks performed similarly with and without the CLAHE method implemented. Note, the RMSE in this table is an approximated based on an early method of our loss function.	55
7.2	The averaged results from the experiments comparing the effects of pairwise histogram equalization. We can see in the case of both the single and multi-task networks that the testing RMSE reduced by about 7 meters and the testing percent error decreased by about half. We also found that the cross-validation results were very inconsistent with testing for all models except the multi-task NN with equalization applied, highlighting that model as our best so far.	57
7.3	The breakdown of our four tests for the single output network using ReLU as the activation function. Averaging over the four experiments the RMSE is 18.53 meters.	61
7.4	The breakdown of our four tests for the multi-task network using ReLU as the activation function. Averaging over the four experiments the RMSE is 18.673 meters.	62

7.5	The breakdown of our four tests for the single-task network using Leaky ReLU as the activation function. Averaging over the four experiments the RMSE is 17.952 meters.	63
7.6	The breakdown of our four tests for the multi-task network using Leaky ReLU as the activation function. Averaging over the four experiments the RMSE is 17.528 meters. The italicized line for dataset 1 indicate the model we ultimately determined to be our best model.	64
7.7	The breakdown of our four tests for the single output network using GELU as the activation function. Averaging over the four experiments the RMSE is 18.370 meters.	65
7.8	The breakdown of our four tests for the multi-task network using GELU as the activation function. Averaging over the four experiments the RMSE is 18.510 meters.	66
7.9	Report of average error metrics given different activation functions used in our baseline neural network structure. The bolded line for the multi-task, pairwise normalized network that uses Leaky ReLU activation is the best network we determined based on lowest RMSE and it is most consistent with the cross-validation errors discussed in the next section.	67
7.10	The shallow layer architecture.	68
7.11	The layer architecture beginning with larger initial layers. . . .	68
7.12	The straight layer architecture.	69
7.13	Testing error metric results from running different architectures on one dataset. We saw that both the bigger input layer and straight layer models performed worse than our baseline network, but saw that the shallower network performed comparably.	69
7.14	Report of validation error metrics given for the different network architectures we tested. Based on our cross-validation results we determined our best model was the nine layer multi-task network, with Leaky ReLU as the activation function. The best result is displayed in bold.	70

4. Introduction

In the military, supplies are critical, and a common method for supply deliveries are parafoil parachutes. With aid from these parachutes, crates of supplies can be dropped by aircrafts almost anywhere in the world [1]. Sometimes these parachutes carry fragile cargo like medicine and tools, while other times multi-ton machinery is dropped from the sky. No matter what these parachutes carry, the accuracy of these supply drops is important to the success and safety of operations and supporting Soldiers in their missions [1].

The most common way of tracking and navigating these parafoils is via Global Positioning System (GPS). GPS is the standard for navigation. This technology was originally developed for, and is still used for, military applications. But today, most modern navigational technologies, like those in cars and cellphones, rely on GPS [2]. GPS works via a constellation of satellites transmitting low frequency signals that devices on Earth use to interpret their location.

Despite GPS being the industry standard for location tracking, it has its limitations and drawbacks. Signals from GPS satellites are not always reliable and can be spoofed or jammed by adversaries. Strong weather can impact signal strength and accuracy as well [1]. These drawbacks of GPS have led to research in alternative forms of navigation that do not rely on external signals but can compete in accuracy with GPS. These methods are called Guided-Airdrop Vision-Based Navigation (GAVN) [1].

Alternative forms of navigation are an active area of research, and the U.S. Army Combat Capabilities Development Command Soldier Center (DEVCOM-SC), the sponsor of the project, is one prominent organization participating in GAVN innovation [1]. DEVCOM-SC is comprised of researchers, engineers, and technologists who work to provide the Soldier with a wide range of capabilities. One of their core domains is airdrop/aerial delivery. To this end, they have sponsored our team to aid their research and development of GAVN technologies.

In recent years there has been an increase in machine learning (ML) and data science (DS) research. Despite some of the theories being centuries old, it is only in the past few decades that there has been access to enough high-quality data to put them into action [3]. Recently, some ML based technologies have achieved great accuracy and have been adopted into daily life. One example is facial recognition, which has been integrated with consumer products like cellphones [8]. We have also seen applications of image recognition in self-driving vehicles, such as Tesla Autopilot [9]. Beyond identifying

iPhone users and navigating city streets, this advanced image processing technology also has the potential to pinpoint the location of cargo parafoils mid-flight. This could allow for fully self-contained navigation systems which do not rely on GPS, and in turn avoid the pitfalls of external signaling. Attempts have been made to develop this technology, but they have not yet been able to perform with better accuracy than traditional GPS navigation [1].

Advanced image processing technologies are based on neural networks (NN), a kind of machine learning model that can make predictions after being trained on thousands, and sometimes millions, of data points [4]. A previous MQP team began work on this project using data collected from in-the-field parachute drops. They found that their models were limited by the small amount of data produced during these test drops [5]. Collecting data from parachute drops is expensive; it not only requires physical resources like jet fuel and aircrafts, and human resources to man the plane, but each drop also runs the risk of damaging valuable equipment [10]. Test drops are also limited to regions with the infrastructure to support them, so they are often done at military testing sites, in turn limiting the variety of data that is collected [11]. It was clear that an alternate way to collect data was required to move forward, one that was both inexpensive and versatile.

Virtual simulations currently have many training applications within the military such as soldiers in field exercises or pilots in flight drills [12]. DEVCOM-SC wanted to apply this technology to collect aerial imaging data from parachute drops. In partnership with DEVCOM-SC, our team constructed an application that, given a flight path, will simulate a parachute drop while collecting aerial images and their corresponding locations. This allowed us to leverage data from in-the-field drops, but modify them to be placed anywhere in the world. Moreover, we could control aspects of the drop, such as weather and time of day, allowing us to collect a variety of high-quality images without the drawbacks of physical parachute test drops.

After collecting data using our simulator, we were able to apply a variety of preprocessing methods to the images and test different kinds of neural networks in an attempt to predict differences in location from two images. Our preprocessing methods included grayscaling, image resizing, and equalization. We explored a variety of neural network structures and ultimately investigated the benefits of multi-task neural networks opposed to single-task networks. We found that a multi-task neural network with nine layers and Leaky ReLU as its activation function was our best model achieving a

validation error of 27.7%.

4.1 Deliverables

Through our work with DEVCOM-SC, we provided several contributions to the body of research surrounding GAVN technologies. We created a parachute drop simulator that is capable of automatically collecting images with their respective locations during a drop. Due to the complexity of the simulator, we found it necessary to include a user manual that would demonstrate how to build the simulator from scratch and how to operate it correctly, included as Appendix Section 10.1. We made versions of the simulator for 10 locations and 8 drop patterns. To make this variety of projects we created code that could either recreate a drop path in its original location, or take parachute drop coordinates from one of the real drops and transform the path to a new location, determined by a user-defined landing destination.

Using our simulator, we collected over 10,000 aerial images labeled with the location coordinates of the virtual parachute. We wrote a program to pair images with their labels and also pair images together to format them for the neural network. We provide code to preprocess and compile the image pairs into a dataset.

Additionally, we developed code for training a neural network to predict the difference in altitude between two images. We ran several experiments with this program and calculated performance metrics across a variety of networks and image preprocessing combinations, and provide this performance information in Section 7. Finally, we also provided our best neural network, determined by lowest error metric across testing and validation.

5. Background

Based on previous research, we were inspired to leverage machine learning as a tool for image-based navigation. This section describes the work that has guided our project, along with the descriptions of virtual simulation, machine learning techniques, preprocessing methods, and deep learning models.

5.1 DEVCOM-SC and Previous Work

Parachutes have been used for decades, with some of their earliest use dating back to 1783 [13]. While parachutes were being developed, they were first tested jumping from trees, then tall buildings for a higher altitude, and then finally from planes for the first time in 1914. Shortly after, in 1916, the parachute was adopted for use in the military, first being used by an Austrian pilot to escape from a burning plane [14].

In World War I, General William "Billy" Mitchell began planning how parachutes could be used for transporting troops and supplies. He theorized that parachutes would allow Soldiers, along with their aid, to infiltrate enemy territories without crossing the front lines [15]. Since then, there have been countless successful parachute operations. One famous example of the use of parachutes was during D-Day, when thousands of parachutes were dropped by the Allies over Normandy. [15].

Today, parachutes are still a significant aspect of military research and development. The U.S. Army Combat Capabilities Development Command (DEVCOM) is the leader in technology research and development for the Army [16]. They have eight major centers located around the world. The DEVCOM Soldier Center (DEVCOM-SC) is at the forefront of parachute research and development. DEVCOM-SC was founded in 1954 and is located in Natick, Massachusetts. According to the organization, it aims "to provide the Army with innovative science and technology solutions to optimize the performance of our Soldiers" [17]. DEVCOM-SC is comprised of researchers, engineers, and technologists who work to provide the Soldier with a wide range of capabilities. Examples of their work beyond GAVN and parachute development include clothing, life support systems, laser-protection systems and augmented reality devices [17].

One of DEVCOM-SC's core domains is airdrop/aerial delivery. This includes the delivery of both supplies and Soldiers. For the delivery of supplies, a plane is flown near the desired landing location for a package, then the supplies are dropped out of the plane and their parachutes are deployed. These

parachutes are unmanned, and therefore the packages are equipped with remote steering equipment to navigate them to the correct landing point [18]. As the parachutes are dropping, a log is recorded with the GPS coordinates of the parachute, and a variety of other information, such as the orientation of the package, is also recorded. Additionally, there is a camera attached to the package that records video as the parachute descends [18].



Figure 5.1: One of DEVCOM-SC's core domains is airdrop/aerial delivery. For deliveries, a plane must drop the supplies via parachute. These parachutes are unmanned and therefore are equipped with remote steering equipment to navigate them to the correct landing point [18]. This image is an example of a JPAD test drop [19].

DEVCOM-SC covers research and development of the package creation, parachute requirements, and airdrop precision. Airdrop precision is critical to ensure that deliveries are timely and accurate. To achieve this, a Global Positioning System (GPS) receiver is attached to steerable parachutes to locate and guide the cargo package to the desired landing point. This delivery location can then be shared with the Soldiers on the ground to retrieve the

package [18].

GPS utilizes satellites and receivers to send and retrieve signals [2]. However, the use of external signals can allow for interference. This can be in the form of jamming or spoofing. Jamming occurs when a device is used to stop the reception of signals between the satellites and receivers [20]. Spoofing occurs when the signal is interfered with and replaced with a false signal, which can relay incorrect positions [21]. Additionally, the signal can be tracked by outside users which can potentially disclose locations, in turn compromising safety [20]. For these reasons, DEVCOM-SC is looking to develop alternatives to GPS-based navigation.

The previous MQP team researched GAVN technologies to solve this problem [5]. They began with a limited set of videos collected during in-the-field parachute tests using the cameras attached to each package. The data was unlabeled, so unsupervised learning was attempted, but achieved poor results. They then created a labeled data set to use in supervised machine learning algorithms. They began testing with three feature matching algorithms [5]. To optimize these algorithms, they analyzed the most effective max distance between pictures for each and applied these constraints to the data to maximize accuracy. These three models were then utilized in an ensemble algorithm. After their creation of the algorithms and neural network, the clear constraint of the project was the lack of data. They determined a critical next step would be to generate more data for machine learning applications [5]. The process of sending up a plane and deploying a parachute is costly and time consuming, so an alternate source of synthetic data was required [10].

5.2 Simulator

In order for our team to create synthetic data, a virtual simulator was required. A simulator is an imitation of a real world process or environment [22]. Virtual simulators have been used for a variety of applications such as user training, research, and experiments. In order to be effective, virtual simulators focus on the quality of data processing, a variety of locations, and their level of realism [22].

One platform used to create virtual simulators are game engines. A game engine contains the functions for rendering worlds, scripts, animation, sound effects, simulations, and interactive game play [23]. The Unreal Engine 4

(UE4) is one of the most popular commercial game engines [23]. UE4 is a real-time 3D creation tool known for its strong performance and realistic simulations [24].

The Unreal Engine allows the use of plug-ins to aid project development. Cesium is a plug-in that creates an entire virtual globe within Unreal Engine as seen in Figure 5.2 [6]. It provides the engine with the capability to create a 3D geospatial ecosystem, based on satellite imaging of Earth. This allows the globe to be rendered with realistic sunlight, landscapes, and associated coordinates, such as the mountain range in Figure 5.3 [6]. The Unreal Engine also contains many options to enhance world development including weather and time of day [24].

A large collection of tutorials is provided by Cesium, UE4, and third parties to complete small tasks that can then be compiled to build the simulator that was required. Due to this vast quantity of information, it is difficult to recreate our project without a proper user manual, therefore one is provided in Appendix Section 10.1.



Figure 5.2: The Unreal Engine with the Cesium plug-in allows for the creation of a 3D geospatial ecosystem, based on satellite imaging of Earth. This is an image of the whole globe rendered in the Unreal Engine.



Figure 5.3: The Unreal Engine with the Cesium plug-in allows for the creation of a 3D geospatial ecosystem, based on satellite imaging of Earth. The mountain range seen in this image is rendered with realistic sunlight, landscapes, and associated coordinates from Cesium.

To now simulate a DEVCOM-SC parachute drop in this virtual environment, logs collected from in the field-test-drops were used. The Unreal Engine has options to input flight paths and connect them to objects like the camera from Figure 5.4 using project blueprints.

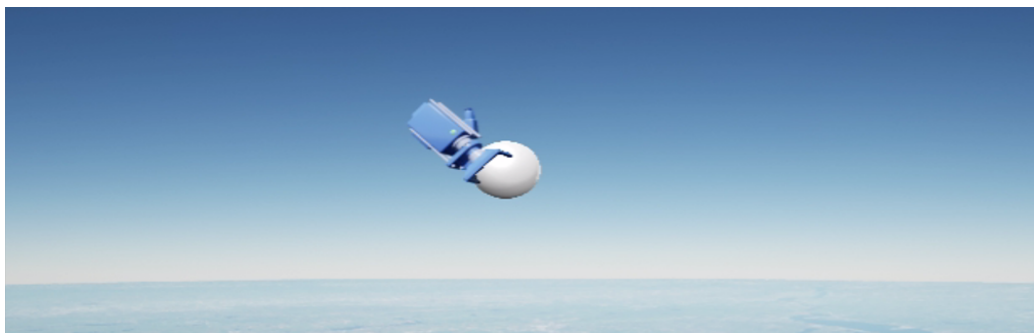


Figure 5.4: Seen here, UE4 has a camera object that can be placed into the environment. This camera can then be programmed to follow a flight path and capture images and locations.

The Unreal Engine has blueprints to program functionality in the engine. The blueprint uses a node-based interface. There are several categories of nodes. Objects within the unreal engine can be represented as nodes. This allows the developer to code features that affect or use the objects in the

game. Another category of nodes in the engine are functions [25]. Some functions included in the engine are printing text to a log file, generating numbers, and retrieving locations of object. Along with functions, there are also nodes for while and for loops. This allows a function or action to occur repeatedly until the ending criteria is met. For example, a developer could easily have a screenshot taken ten times by using a for loop [25].

While the logic for many of these nodes is pre-built, developers can edit the logic of a node and create custom nodes. One customization would be to create a delay between each loop. Now instead of a screenshot being taken right after the other, a developer could program the for loop node to wait one second between each screenshot [25].

To start a blueprint program, there are nodes that trigger code. These triggers include the game starting, certain user actions, or keyboard clicks. For example, a developer could program the loop to begin when a user clicks the L key. Snapshots from the perspective of the camera object can be taken by the user, as well as the location of the object relative to the Cesium globe. These triggers are saved to a universal log file created by UE4.

Another feature of UE4 is cinematic mode. This tool allows video to be recorded from the perspective of a moving or stationary object in the scene. This tool was key in recording high definition video of the simulated flights from the perspective of the camera object following the given flight path [26].

5.3 Image Processing

A strong understanding of how digital images are composed and can be manipulated is critical to deep learning models for image recognition. Images can be modified before they are used for training or testing. Modifications pertinent to this project include grayscaling, equalizing, blurring, resizing, compressing, and flattening.

Digital images are matrices where each element of row i and column j is a pixel. Each pixel will also have k channels where, $k = 3$ for color images and $k = 1$ for grayscale images [27]. The value of a pixel at element i, j, k represents intensity, it ranges from 0 to 255. For color images such as in Figure 5.5a, a pixel will have 3 channels that can represent the intensity of red, green and blue, which allows a pixel to take on any color in the visible light spectrum. Grayscale images such as that in Figure 5.5b only require one channel per pixel, and the value represents the intensity of white. Therefore,

a grayscale image will require three times less data to represent it or store it [27].

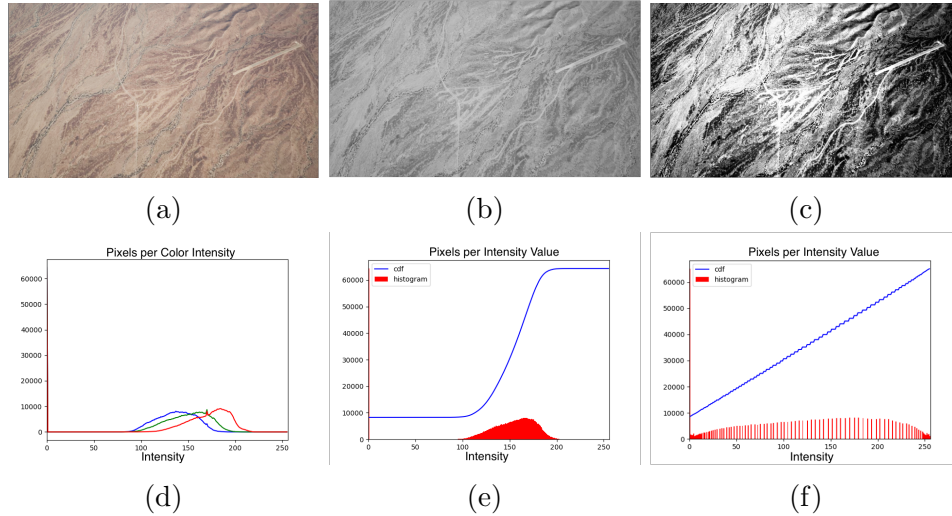


Figure 5.5: Images taken with UE4 default to color. From top left, (a) is a raw colorized image from the unreal engine, (b) is the same image grayscaled, (c) is the final intensity equalized image, (d) is the histogram of the pixels with the intensity of each color (red, green, blue), (e) is the histogram of the grayscaled image before equalization and (f) is the histogram of the equalized image.

An image's intensity histogram is an important window into its properties [28]. It is created by counting the number of pixels within each channel that belong to each intensity value from 0 to 255. Examples of intensity histograms include Figures 5.5d, 5.5e, and 5.5f. Figure 5.5d has three superimposed histograms, one for each color channel. Figure 5.5e has only one channel and the intensity of white appears to be similar to the intensities of the color channels seen in Figure 5.5d. Figure 5.5f is the histogram of the grayscaled image when the contrast has been improved through equalization [28].

The effects of grayscaleing are displayed in the difference between Figures 5.5a and 5.5b. Grayscaleing is the process of replacing the three channels of a color image with a single channel that represents the intensity of white. It

is defined by the equation [29]:

$$RGBtoGray : Y = \beta_1 R + \beta_2 G + \beta_3 B$$

Grayscale inherently reduces the information contained in the images. Sometimes this is ideal, for example, a parafoil may only possess a camera that captures grayscale images. Therefore, the images captured by the parafoil drop simulator must also be grayscale, otherwise the deep learning model may not work when exposed to real world conditions. Other benefits of grayscale include a dimension reduction from three channels to one. Reducing the dimensions will make training a deep learning model less expensive computationally, time wise, and with respect to available memory, in turn allowing more images to be used for training models [29].

Deep learning models can ideally receive unprocessed images and be able to make inferences based on features found within the images. To make features in an image more visible, one can improve the contrast between pixel intensities. This process is called histogram equalization [30].

The effects of histogram equalization can be seen in the differences between the Figures 5.5b and 5.5c and their respective histograms, Figures 5.5e and 5.5f. It accomplishes this improvement by increasing the range of intensities within the image. The increase is done by mapping the original distribution to a wider and more uniform distribution. This remapping should be done with the cumulative distribution function (cdf) [30]. For a histogram H where $H(i)$ is the number of pixels with intensity i , the cdf of $H(i)$ is defined as:

$$H'(i) = \sum_{0 \leq j < i} H(j).$$

Figure 5.5e shows the original distribution of color intensity in red and the cdf in blue and 5.5f shows the new distribution of intensity values are more equally distributed across the possible range of values. This change is also represented by the cdf line (blue) straightening [30].

Histogram equalization can also be performed using a sliding window across the image. This algorithm is called the CLAHE method [31]. The main idea to this method is that histogram equalization on the whole image may boost the brightness and wash away some information. Therefore, it may be more useful to perform histogram equalization across separate portions of the image at a time[31].

In some cases, histogram equalization may create noise within the image. Therefore, blurring can be used to smooth out the noise. Blurring is especially useful for preserving shapes and edges [32]. Blurring is done by sliding a window, or kernel, across the image, calculating the average pixel intensity, and changing the intensity of each pixel within the window to that average. Looking at the two images of the runway shown in Figure 5.6. Image (a) is an original image and image (b) is the same picture with blurring performed via the Gaussian method [32].

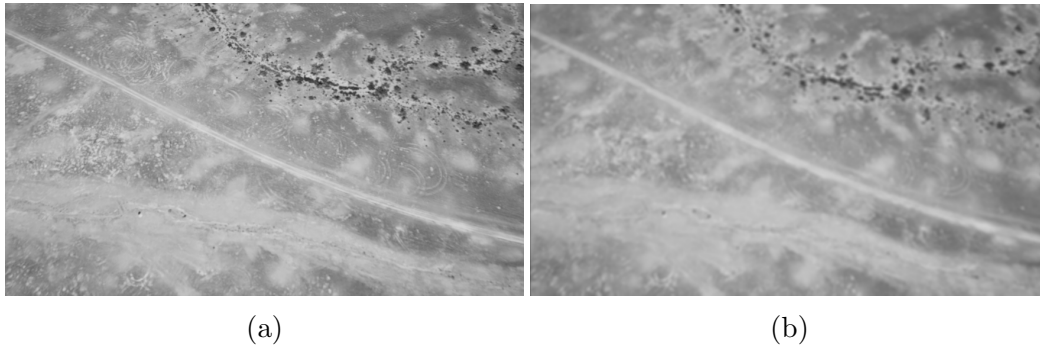


Figure 5.6: Blurring may be used to smooth out noise within an image. Blurring is especially useful for preserving shapes and edges. Image (a) is a grayscale image of the Yuma Proving Grounds and image (b) is the same image with gaussian blurring applied.

Since images are arrays of pixel values, with each entry representing one pixel, high resolution images are very large, complicated matrices. There are many ways of reducing their size and simplifying the computations that are involved with training the neural network. Some methods include reducing resolution, Fourier compression, and Wavelet compression.

Image resolution can be reduced by averaging the values of multiple pixels into one larger pixel. Common values for image resolution are 144 by 256 pixels, 240 by 426 pixels, 360 by 480 pixels, and 480 by 848 pixels. This step was also important to our process since it made images of different sizes consistent in their dimensions for linear algebra purposes.

Fourier and Wavelet compression are similar methods for reducing complexity in images. They both work by performing calculations on the images and finding the highest value coefficients that represent the image. The rest of the values can be replaced with zero. The effects of each method

are depicted in Figure 5.7. By applying either of these methods, it reduces complexity and difficulty of calculations. Fourier follows a specifically defined algorithm, but Wavelet gets its name from the variety of waves that can be used to calculate the coefficients. Common wavelets used for feature detection and image processing are Haar and Daubechies [33].

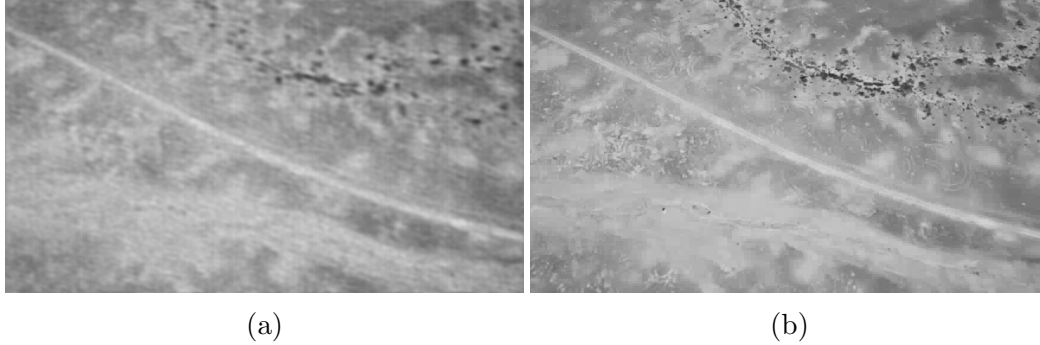


Figure 5.7: Fourier and Wavelet compression are similar methods for reducing complexity in images. They both work by performing calculations on the images and finding the highest value coefficients that represent the image. The rest of the values can be replaced with zero. By applying either of these methods, it reduces complexity and difficulty of calculations [33]. Image (a) is an example of Fourier compression applied to 5.6a and image (b) is wavelet compression applied to the same image.

One more important image preprocessing step is flattening. Individual grayscale images are represented as two dimensional (2D) matrices, flattening an image converts a 2D matrix into a vector [34]. It accomplishes this task by appending each row to the end of the previous row, repeat until a single vector remains. Once an image pair is flattened, it becomes a one dimensional representation of the data, which is the appropriate structure for a sample in machine learning models [34].

5.4 Machine Learning

Machine learning is a popular field in artificial intelligence, which refers to predictive models trained using data. There are many kinds of machine learning models, like random forests and neural networks [35].

Neural networks have become the industry leader in machine learning models, but they are part of a subcategory of machine learning called deep learning [36]. This name relates to the special structure of neural networks which consists of a series of layers that combine predictors through a series of nonlinear calculations in order to yield an output of the desired size.

The machine learning models we explored, along with other important information necessary for understanding the process of machine learning will be explained in the section, including supervised learning, classification verses regression, evaluation metrics, and cross-validation.

The general idea of any ML model is that some function of interest, f , exists in nature, with some noise, notated as:

$$y = f(x) + \epsilon$$

The goal of machine a learning method is to approximate f as closely as possible, despite not knowing what its true structure is. We do this by finding some \hat{f} such that:

$$\hat{y} = \hat{f}(x) + \epsilon$$

This can be done in high dimensions, such that $\mathbf{x} \in R^{n \times p}$, and \mathbf{y} and $\hat{\mathbf{y}}$ are matrices such that $\mathbf{y} \in R^{n \times k}$ and $\hat{\mathbf{y}} \in R^{n \times k}$. The matrix \mathbf{x} represents the samples, where n is the number of samples and p is the number of predictors. As for \mathbf{y} and $\hat{\mathbf{y}}$, k is the desired dimension of variables to predict. Oftentimes $k = 1$, but k can also be a larger value.

5.4.1 Supervised Learning

There are two main categories of machine learning problems: supervised and unsupervised learning. The most notable difference between the two is supervised learning utilizes labeled data while unsupervised learning does not [37]. Therefore, they have different applications for which they are best suited.

Supervised learning relies on labeled datasets to train algorithms. The algorithm makes predictions and then accuracy can be determined by comparison predictions to true values [38]. In our case, we have our labelled datasets comprised of images and their locations. An algorithm can be given a pair of pictures as input and predict the positional difference between the image as an output. The correct location can then be to gauge the prediction accuracy of the model.

Now that we have our labeled data and the type of machine learning problem chosen. We now have to decide what type of problem we want to solve. We would determine this by deciding what we want the algorithm to predict based off of at the images its given. The main types of problems are classification and regression[37].

5.4.2 Classification and Regression

There are two ways to predict data using labeled data in a supervised learning algorithm: classification and regression. Classification predicts, or classifies, discrete data. It assigns the data to a class that is predefined. Regression predicts continuous values by finding relationships between dependent and independent variables [39]. These differences can be seen in Figure 5.8.

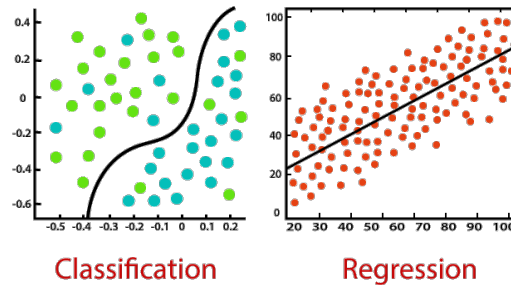


Figure 5.8: Classification predicts, or classifies, discrete data. It assigns the data to a class that is predefined. Regression predicts continuous values by finding relationships between dependent and independent variables [39]. The image on the left is of a classification example, and the image on the right is of a regression example [40].

Classification is used to divide data into classes based on different parameters of the data [38]. One example of a classification model that can be done with our data would be predicting the country the image was taken in. The predefined class is the country. Classification models include K-nearest neighbors, support vector machines, and logistic regression [41].

Regression maps input values to a continuous output variable [39]. The values of latitude, longitude, and altitude are continuous output variables. There are several values our model could predict. One that we considered were guessing the difference of latitude, latitude, and longitude between two

pictures. Another option would be guessing one of the values of location of one picture. We moved forward with a regression problem to have the model predict the difference in altitude between two images.

Regression models include simple linear regression and polynomial regression [41]. Some models can be used for either classification or regression problems. This means they are capable of outputting either a numerical or categorical response. Examples of these include decision trees, random forests [41], and neural networks [42].

5.4.3 Evaluation Metrics

One cannot train a machine learning model without first deciding how they will evaluate it. Without an evaluation metric, there is no way to directly compare different models, hyperparameters, or datasets. Three popular evaluation metrics for regression are mean squared (MSE), root mean squared error (RMSE), and percent error.

The equation for MSE is as follows where n is the number of datapoints in a batch and y_i is the true value and \hat{y}_i is the predicted value.

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

MSE and RMSE are very similar, but RMSE, as the name implies, square roots MSE in order to return the value to its original units.

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$$

Percent error is calculated by finding the difference between a prediction and the true value, then dividing by the true value.

$$\text{Percent Error} = \left| \frac{y_i - \hat{y}_i}{y_i} \right|$$

Percent error is a measure of error with respect to the magnitude of the value. These three values, MSE, RMSE, and percent error, typically grow and shrink together. All three metrics share the rule that smaller values indicate a better model, with 0 being perfect.

5.4.4 Cross-Validation

Cross-validation (CV) is the important part in the machine learning process. It is a way of ensuring that an algorithm predicts to the expected accuracy. Cross-validation can help catch mistakes like over-fitting or data snooping. There are many ways of performing cross-validation, such as k-fold, bootstrap, and leave-one-out cross-validation [41]. Though the details of each method vary, they all have the common goal of selecting a subset of the data to use to train the model on, while preserving the rest of the data to use to validate the models' accuracy [41].

One of the most popular methods of cross-validation is validation sets [43]. This is where one portion of the data, typically 20-30% of samples, are set aside for testing purposes. Then the remaining data, 70-80%, is used to train the models. Doing this method multiple times at each level of the validation process is known as hierarchical cross-validation [43]. In our process we used hierarchical cross-validation to train our neural network. This means we first took a sample of our data, in our case 25,000 samples, and set it aside to test on at the very end, to ensure our final model worked as expected. This set of samples is called the validation set. Then we used the remaining data to train and test our neural network, using 80% of our data as training samples and 20% as testing samples.

The validation set is crucial in verifying the accuracy of a ML model. The percent error and MSE collected from training the neural network is compared to the results of cross-validation. The best model is determined by the lowest average RMSE and percent error.

5.4.5 Random Forests

Prior to the rise in neural networks, random forests were the leading predictive models in machine learning. Random forests are simply a collection of decision trees built through bootstrap aggregation, which work together in an ensemble [41]. A decision tree is a kind of ML method that uses linear divisions, often called cuts, to create regions for predictions. The cuts indicate decisions that are used to guide a prediction of a testing point. This method works well on both regression and classification problems [41].

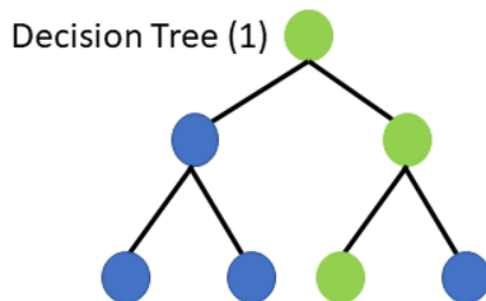


Figure 5.9: A decision tree is a kind of ML method that uses linear divisions, often called cuts, to create regions for predictions. The cuts indicate decisions that are used to guide a prediction of a testing point. This method works well on both regression and classification problems [41]. This image represents the basic node and branch structure of a decision tree [44].

Random forests are a group of decision trees that work together to make predictions. Each tree is slightly different since it is built with a bootstrapped sample of data. Bootstrapping is simply taking n random samples, with replacement, from a group of size n [41]. This allows some points to be included multiple times, while others are not included at all. Each point has approximately a 60% chance of being included in any of the samples. Additionally, each split decides which variable to cut on from only a subset of the predictors sampled at each node. This randomization creates variety since it causes each to be slightly different [41]. Then these trees can work together as an ensemble and either average their predictions (for regression) or vote on grouping (for classification). This method of having many trees, each trained on less data, has proved to be more powerful than a single tree trained, on the whole training sample [41].

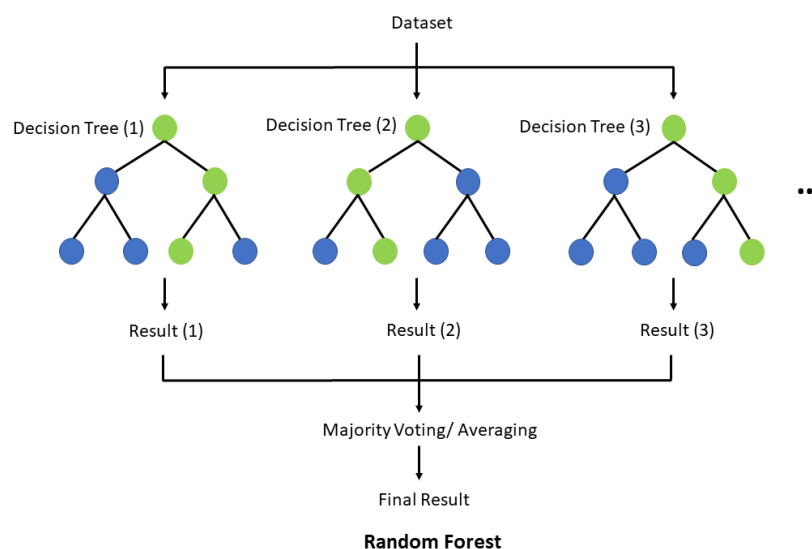


Figure 5.10: Random forests are a group of decision trees that work together as an ensemble to make predictions. Each tree is slightly different since it is built with a bootstrapped sample of data and each split decides which variable to cut on from only a subset of the predictors sampled at each node [41]. This image illustrates the structure of a random forest working together to make a prediction [44].

5.4.6 Neural Networks

A neural network is a computational model inspired by the composition of the human brain. Like the brain being composed of interconnected nerve cells, a network is composed of interconnected neurons. In general, these neurons are grouped together into layers [4]. Figure 5.11 depicts a 3 layered neural network, including the input, hidden and output layers. The input neurons of the network are denoted as x_i , the hidden neurons of the network are denoted h_i and the final output is denoted as \hat{y} . The first layer of the network depends on the raw data, such as flattened image pairs, being passed as input. Notice that the value within each neuron is passed to every neuron of the next layer via the arrows.

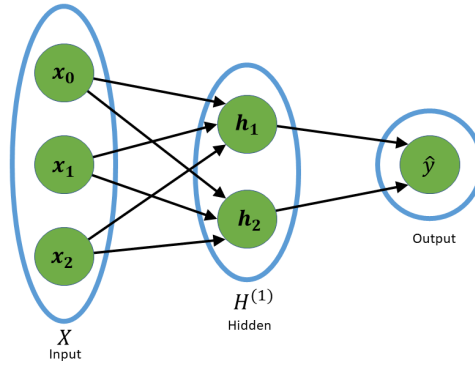


Figure 5.11: This figure depicts a 3 layered neural network, including the input, hidden and output layers. The input neurons of the network are denoted as x_i , the hidden neurons of the network are denoted h_i and the final output is denoted as \hat{y} . The first layer of the network depends on the raw data, such as flattened image pairs, being passed as input. Notice that the value within each neuron is passed to every neuron of the next layer via the arrows.

Along the arrows that connect each neuron are weights. Figure 5.12 shows a snippet of a network of 2 input scalar cells, x_0 and x_1 , and one hidden scalar cell h_1 . Each of the cells that are input to h_1 are given a scalar weight, w_1 and w_2 respectively. Each neuron that is not part of the input layer also carries a unique bias value, b_i [4].

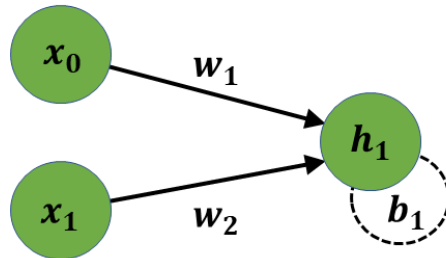


Figure 5.12: This figure shows a simple network made up of 2 input scalar cells, x_0 and x_1 , and one hidden scalar cell h_1 . Each of the cells that are input to h_1 are given a scalar weight, w_1 and w_2 respectively. Each neuron that is not part of the input layer also carries a unique bias value, b_i .

The calculation of h_1 is shown in the following equation:

$$x_0 \times w_1 + x_1 \times w_2 + b_1 = h_1$$

The product is taken between the values of the input cells and their weights. Each product will be summed together, and finally the bias is added. To simplify the notation for calculating the output of larger networks, each layer and their respective weights can be vectorized as shown below:

$$\mathbf{X}_{1,n} = [x_0 \ x_1], \mathbf{W}_{n,h}^{(1)} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}, \mathbf{b}_{1,h}^{(1)} = [b_1]$$

\mathbf{X} represents the input vector, made up of n features. $\mathbf{W}^{(i)}$ represents the weight matrix for layer i . It is made up of n weights for each previous neuron and h hidden layers. Each column of $\mathbf{W}^{(i)}$ represents one hidden neuron of the layer. To represent the bias vector of layer i we have $b^{(i)}$, each of the h elements of $b^{(i)}$ is the scalar bias relative to each hidden neuron of layer i . \mathbf{H}^i is made up of the calculated neurons of hidden layer i . The vectorized form of calculating this simple network would be:

$$\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)} = \mathbf{H}^{(1)}$$

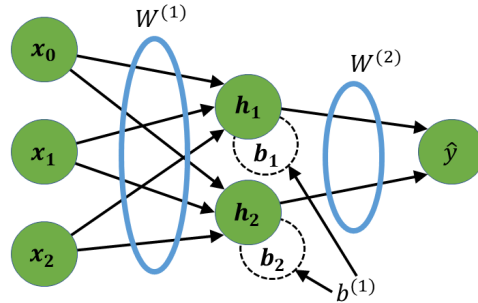


Figure 5.13: This is a diagram of a more complex network. There is one input layer of size 3, one hidden layer of size 2, and one output layer of size 1. x_0 , x_1 , and x_2 are the inputs, $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ are the weight matrices, h_1 , and h_2 are the hidden neurons, b_1 , and b_2 are the biases, and \hat{y} is the output.

Figure 5.13 shows a diagram of a more complex network. There is one input layer of size 3, one hidden layer of size 2, and one output layer of size

1. The vectorized equation calculating the output of this network is:

$$\mathbf{W}^{(2)}(\mathbf{XW}^{(1)} + \mathbf{b}^{(1)}) + \mathbf{b}^{(2)} = \hat{y}$$

5.4.6.1 Activation Functions

Another critical aspect of a neural network is the activation function. The main idea of an activation function is to set a threshold for the value of h_i to either exceed and pass on its information, or not exceed and not pass on its information. The use of activation functions also allows neural networks to create non-linear prediction models [45]. In addition, activation functions determine which weights and biases will be updated as the network is learning. Figure 5.14 shows a more in-depth perspective on how activation functions fit into the calculation, in this case, the activation function is denoted as f .

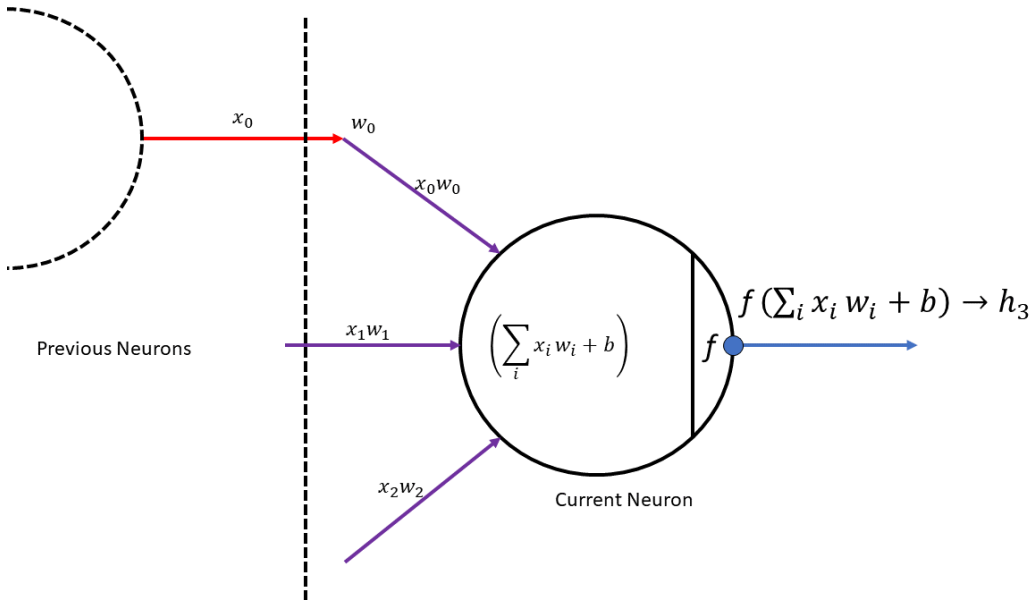


Figure 5.14: The main idea of an activation function is to set a threshold for the value of h_i to either exceed and pass on its information, or not exceed and not pass on its information. The use of activation functions also allows neural networks to create non-linear prediction models [45].

Examples of activation functions include the Rectified Linear Unit (ReLU),

leaky ReLU, and the Gaussian Error Linear Units (GELU) [46]. Figures 5.15, 5.16 and 5.17 are the graphs of the ReLU, Leaky ReLU, and GELU respectively. While their graphs look similar, the crucial difference is how these functions handle negative values and values near 0. Depending on the task, different activation functions should be tested to determine which works best [46].

Rectified Linear is the most popular activation function currently. One of its best aspects is that it is very computationally simple [47]. It works by setting all values less than zero to zero, and any input above zero remains the same. The main drawback of ReLU is that during the backward propagation phase, the neurons that do not fire will also not have their weights updated [47].

$$ReLU(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$$

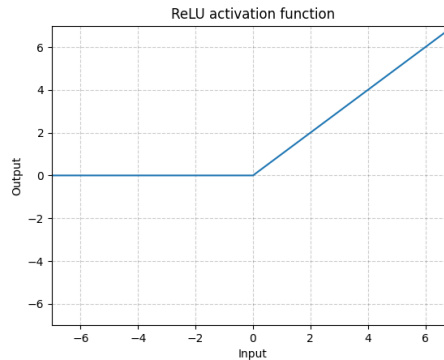


Figure 5.15: The graph of the ReLU activation function. This activation function works by setting all values less than zero to zero, and any input above zero remains the same [48].

Leaky ReLU works very much like ReLU, however Leaky ReLU expects to be given a positive slope which will replace the input if it is less than 0 [47]. The effect of having this positive slope is that all the neurons will fire in the network. This fact allows all the weights to be updated after each training round. However, the neurons that fire with an input of less than 0 will have a minuscule effect on the output compared to the neurons that

fired with an output greater than 0 [47].

$$\text{LeakyReLU}(x) = \begin{cases} x & x \geq 0 \\ 0.01x & x < 0 \end{cases}$$

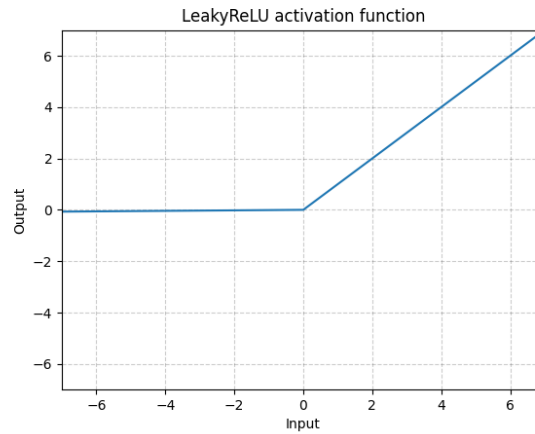


Figure 5.16: Leaky ReLU works very much like ReLU, however Leaky ReLU expects to be given a positive slope which will replace the input if it is less than 0 [47]. The effect of having this positive slope is that all the neurons will fire in the network. This image is a graph of the Leaky ReLU activation function [49].

With GELU, inputs are weighted by their percentile rather than sign, like ReLU [50]. Some research has shown that GELU has improved performance for problems related to computer vision. Instead of having a cut off at $x = 0$, GELU uses the cumulative distribution function of the standard normal distribution. Different distribution functions can be used instead, however there is limited agreement on which work best [50].

$$\text{GELU}(x) = x * \Phi(x)$$

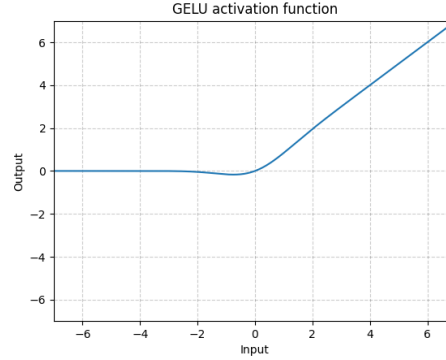


Figure 5.17: With GELU, inputs are weighted by their percentile rather than sign, like ReLU [50]. Instead of having a cut off at $x = 0$, GELU uses the cumulative distribution function of the standard normal distribution. This image is a graph of the GELU activation function [51].

5.4.6.2 Training a Neural Network

To train a neural network, the model learns by making predictions on the full dataset repeatedly [52]. Each time a network trains over the whole training dataset it is called an epoch. Usually for computational purposes, the dataset is broken into batches. For example, when trying to predict the difference in altitude between two images with a dataset of 75,000 training pairs, one could make batches of size 128. With a batch size of 128, the network will make 128 predictions at once, and will find the average error across those predictions. These errors are calculated using a loss function.

An example of a loss function would be mean squared error. It is calculated by taking the difference between the prediction and the ground truth and squaring it to remove negatives. If the network is predicting 3 numbers in the case of multi-task learning, the output \hat{y} and truth y will be vectors of size 3×1 . If the network is predicting 1 number in the case of single-task learning, the output \hat{y} and truth y will be scalars. The equation for mean squared error is:

$$MSE(\hat{y}, y) = \frac{1}{mn} * \sum (y - \hat{y})^2.$$

Where \hat{y} is the prediction, y is the truth, n is the preferred batch size, and m is the size of the output.

The main idea behind back propagation is to calculate the partial derivatives, or gradients, of a chosen loss function with respect to each of the weights and biases. Those gradients are used to update the weights in order to reduce the error [52]. Having enough data becomes the limiting factor with back propagation, which is partially why many epochs can be used instead of larger datasets.

5.4.6.3 Single-task and Multi-task Neural Networks

Single-task neural networks are designed to make a single prediction [7] such as the altitude change between two images. Multi-task neural networks are designed to make multiple predictions in tandem. In the case of parafoil navigation, an example of multi-task learning would be: given two image pairs, predict the longitude, latitude and altitude change between the images. In cases where the difficulty in predicting each task is similar, splitting a multi-task network into single-task networks, one for each prediction, may be more useful. In cases where the difficulty in predicting each task is not similar (i.e. predicting altitude being easier than predicting latitude and longitude), using the multi-task network may be more suitable for predicting the factor with the least difficulty [7].

Designing a multi-task network can be done by combining the loss functions of the different values of interest. By combining the different error metrics into a single value, it can be used to guide the neural network's training and potentially improve the predictions of the variable of interest [7].

5.5 High Performance Computing

Our team made many decisions regarding the makeup of the neural networks. In order to understand some of the costs and benefits that went into those decisions one must understand the basic components of a computer, their limitations and how a high-performance computing system can be used for ML applications. The basic components that are of concern while training a neural network are central processing unit (CPU), random access memory (RAM), graphics processing unit (GPU), and hard disk.

Data on a computer is stored in a hierarchy based on most recent use [53]. For instance, one high resolution image might be 2 MB in size, to

store a dataset of 10000 images (20 GB) one would save those images to a hard disk. Hard disks are the lowest layer of the hierarchy because they have persistence and are voluminous [53]. This means that they can preserve thousands of gigabytes of data even when the computer is turned off. When our team needs to view one of the saved images and apply grayscaling, a copy of the image will be transferred from the hard disk to the RAM [53]. Having the luxury of data persistence comes at the cost of read and write times that are orders of magnitude slower than RAM. RAM is faster because it is non-persistent and closely connected to the CPU. RAM is faster, but it is also more monetarily expensive and therefore it is a resource that must be thoughtfully conserved [53]. Once we have finished viewing the image and altering it, the computer will no longer need the image on the RAM, and will update the image on the hard disk to be a copy of the image on RAM, then it will mark the image in RAM for removal in order to keep the RAM as clean as possible [53].

The natural tendency when training neural networks is to want more data and a larger network. Since neural networks are made up of many matrix multiplications, it is essentially the same type of independent computations being done repetitively. Calculations are done with a CPU [53]. A CPU will make millions of calculations per second, but it is very restricted in how many computations it can make in parallel. The load that is placed on a CPU for training neural networks is very high. To mitigate this stress, researchers adopted a graphics processing unit (GPU) to meet their needs of completing many, simple calculations in parallel. GPUs are a derivative of the CPU, but their goal is to maximize parallel computing [53]. Due to this, they are orders of magnitude faster than CPUs at completing tasks such as matrix multiplication. The drawback that comes with using GPUs for calculations such as matrix multiplication is that a GPU has its own fixed size dedicated RAM and both matrices must be able to fit onto it [53].

A retail computer may be a good device for a student to learn how to train a neural network, but they are not optimized for that task. A high-performance computing (HPC) system is required for professionals. HPC systems are designed to provide an entire company with computational resources [54]. One could picture an HPC system as hundreds of regular computers linked together to form a computing cluster. Instead of being made up of a single machine, the HPC system links together many servers [54]. A server is a computer that does not need a display. A regular computer is used to remotely connect to a server via the command terminal and the secure

shell (ssh) protocol. This allows the user to run code on the server, but be writing the code from another computer. Since an HPC system is offered to an entire company, special software is written that allots resources to those who request them [54]. For example, an HPC system could be made up of 500 CPU cores, 1000 GB of RAM, 100 GPUs, and 1000 TB of hard disk.

5.6 Weights & Biases

Weights & Biases (W&B) is a ML operation management platform [55]. Its features include tracking experiments, reporting, and data visualization. Each time we run data through our neural networks, W&B logs information such as training and testing predictions, ground truths, and our calculated error metrics at both the batch and epoch levels [55].

With the information from (W&B), we can create graphs and select x and y axis from several data metrics saved from our network to build an informative dashboard about model performance. Visualizing the data allows us to learn about the performance of the network. It can help spot over fitting, under fitting, and convergence rates.

Weights & Biases also aids in organization. Different neural network runs can be tagged in categories [55]. Then graphs can be created to compare the data within that category. Such as comparing the data of networks that used different types of image preprocessing [55].

6. Methodology

Our first step in completing the project was to design a virtual simulator that could be used to collect high-quality, labeled images for training our machine learning model. Once developed, we ran our simulator to collect our desired images and organized them into a dataset that paired images and labeled them with their positional differences. In order to create these datasets, we had to apply preprocessing methods to our image data. Next, we designed and trained machine learning models that could predict the differences based on the datasets. Finally, we evaluated our models using a series of error metrics and cross validated our models using a validation set to ensure their reported accuracy.

6.1 Simulator

Due to the large quantity of data required to train neural networks and the limited amount of in-the-field data available from DEVCOM-SC, our first step of the project was to develop a simulator so we could create our own synthetic data. We decided that a virtual simulator would be the best method since it had the greatest opportunity for variety of data, while also being an inexpensive solution. We began by researching platforms which we could use to create a simulator, knowing we would need features such as image capture, custom landscapes, and flight capabilities. Our unique set of requirements was met by a video game design software, the Unreal Engine (UE4). Exact instructions for building and using our simulator are provided in the manual we wrote, included as Appendix Section 10.1.

An Unreal Engine project begins with an empty environment, seen in Figure 6.1 that can be filled with a wide variety of elements, effects, and movements. Projects also have logic for visual effects that simulate our world, like gravity and light. There are options to bring in templates and pre-designed aspects from the Unreal Engine Marketplace, like mountainscapes or city scenes. There are also the tools in place for the user to build their own unique scene [6].

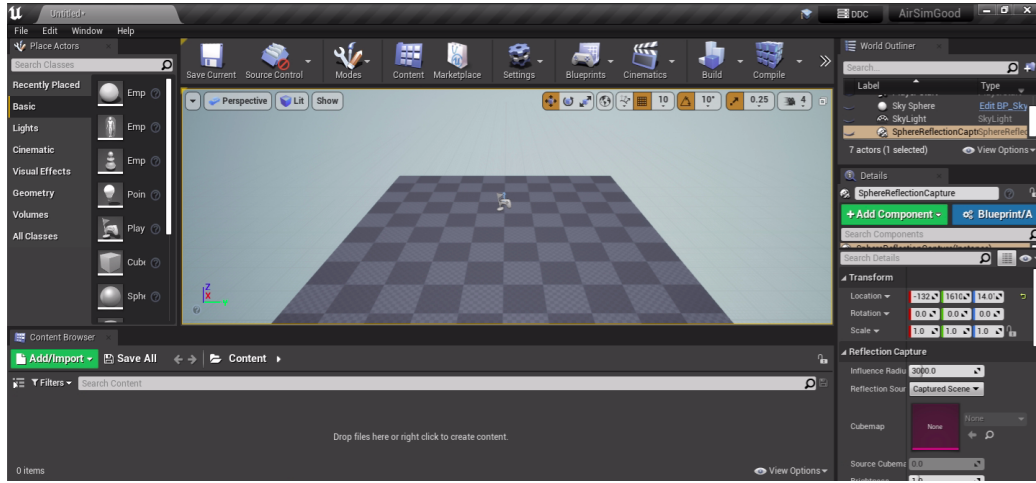


Figure 6.1: The Unreal Engine Editor window when you first create a project. The scene is empty except for a blank ground tile. The engine provides tools for adding in objects and actors, and coding in functionality.

Since the projects made in UE4 are local to specific machines, an important tool we utilized was a test environment on WPI's virtual desktop infrastructure. This allowed all team members to share projects and associated files through a remote login system. This was critical for our ability to collaborate in designing the simulator, as well as later running it to collect data.

6.1.1 First Attempt Using AirSim

We first began building our simulator in the Unreal Engine by using a Marketplace template which featured a highly realistic section of world terrain. This pre-built environment featured a snowy mountain landscape and was ideal for understanding the basics of Unreal Engine [56]. It contained many landmarks that we expected would be useful in later feature detection such as mountain peaks, trees, lakes, and roads. The landscape was built as a limited tile so there were edges where the world terrain cut off, as seen in Figure 6.2, which was a drawback.

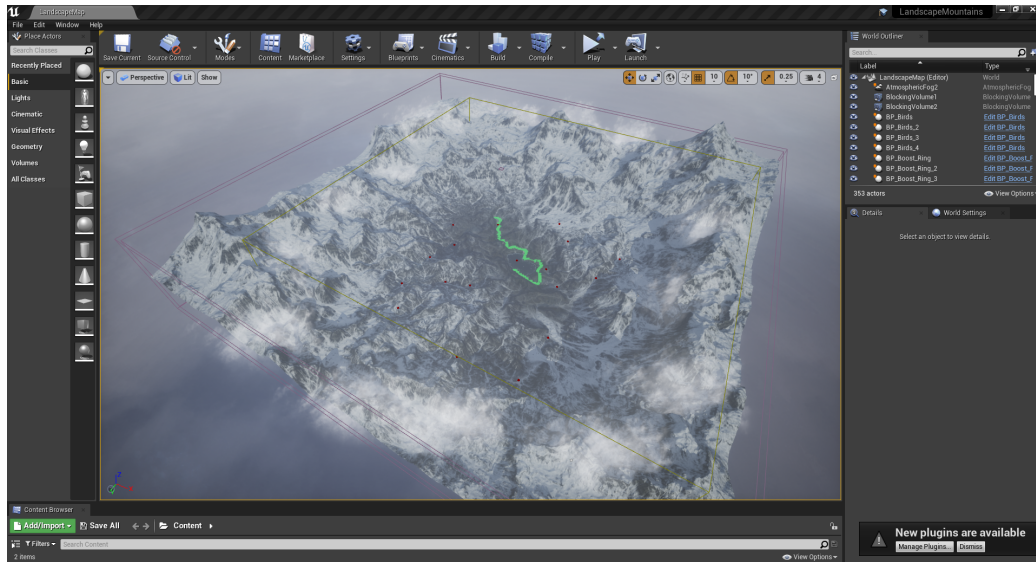


Figure 6.2: The Unreal Engine Editor window with the first landscape we used. The scene contains mountains, roads, and lake. The scene is limited to a tile which was a drawback.

Another tool we explored early on in the process was AirSim. AirSim is an open source simulator for drones and cars [57]. Using this tool, we were able to easily add a drone to our project that could fly through the simulated environment according to user controls. AirSim also has a feature to record videos or collect images from the virtual drone's perspective, mimicking the functionality of a camera attached to the DEVCOM-SC parachutes.

For our purposes, a simulator would need to have the capability to collect both aerial images from the drone's perspective and the coordinates of where the image was taken. We wanted to record the location of each picture in latitude, longitude, and altitude, the same measurements that are recorded by GPS. However, AirSim and the Unreal Engine had conflicting coordinate systems, complicating the task of location tracking.

After exploring the discrepancy, we discovered that AirSim uses an origin-based coordinate system. This fact prevented us from having a consistent basis to convert the drone locations to the desired measures of latitude, longitude, and altitude. After several attempts to convert location measurements between AirSim and the Unreal Engine, we decided it was best to pursue a different solution to bring in an aircraft that could simulate a parachute

drop.

6.1.2 Final Method Using Cesium

We researched alternative programs to use to aid in the building of our simulator in the Unreal Engine. In researching different tools, we found Cesium, a plugin for the Unreal Engine. Cesium provides an accurate digital rendering of the globe. The graphics in Cesium are built using satellite imaging. Cesium also tracks objects in the virtual environment according to WSG-84 coordinates[58]: these are represented as latitude, longitude, and altitude measured with respect to an ellipsoidal Earth [59]. Cesium met both our needs for meaningful coordinates and an accurate, virtual globe for aerial image collection. To implement Cesium, first the plug-in had to be downloaded and installed in a new Unreal Engine project. Then we could import a virtual Earth and Sun to create our simulated environment, as seen in Figure 6.3.

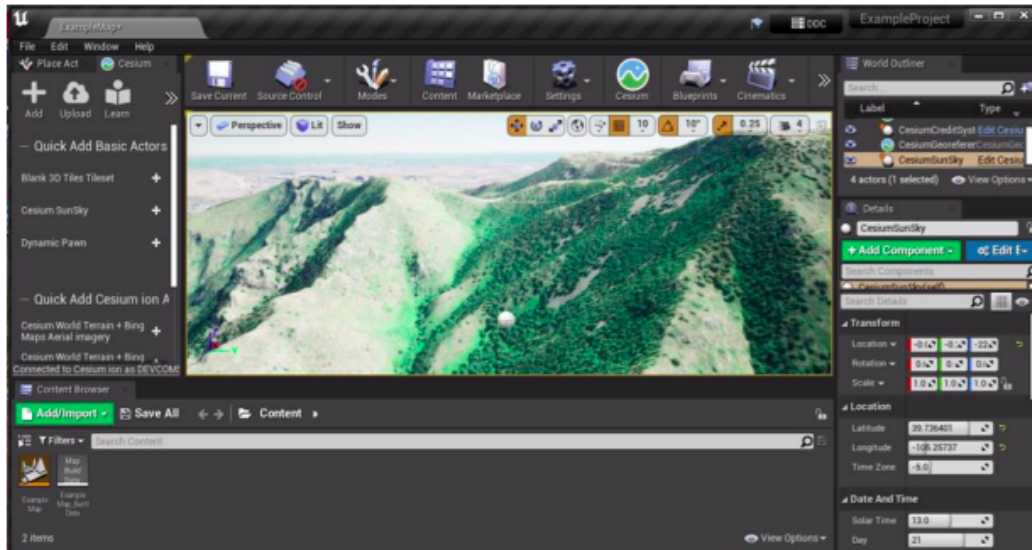


Figure 6.3: The Unreal Engine Editor window with the Cesium plug-in added in.

Once the environment was ready, we needed to decide on a virtual object that could both simulate the movements of a parachute falling and collect

images from the perspective of the object. AirSim had these prebuilt capabilities but again was incompatible with the coordinate system of the rest of the Unreal Engine environment. Instead of using AirSim, we created an object actor that could be paired with a camera in a parent-child relationship. The camera angle could be adjusted by altering the object's pitch, roll, and yaw. This allowed us to position the camera to best simulate the real-life positioning of the camera on parafoil cargo packages.

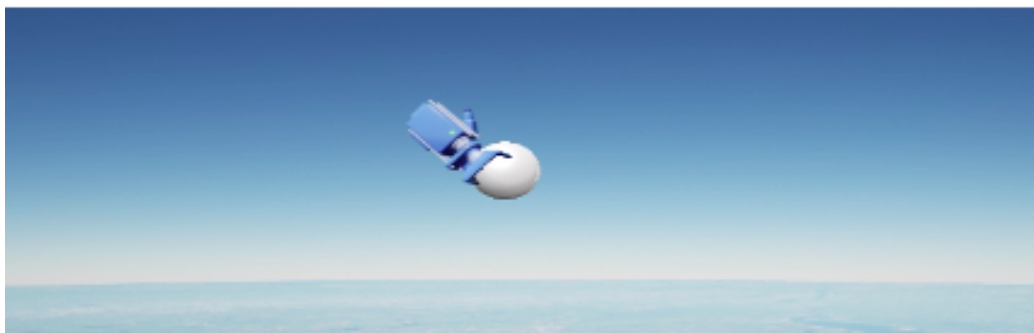


Figure 6.4: The object within the Unreal Engine with a camera attached to it. This object mimics a cargo parachute and its attached camera.

Now that the object was created with accurate camera positioning, we needed to program the object to fall in a way similar to real-life parachutes. Cesium has the capability to take in a list of coordinates to create a spline path for an object to move along. This meant we could give the object a series of coordinates and then program the parachute to follow the path of the spline between those points. The coordinates we used to create parachute drop paths were extracted from eight DEVCOM-SC parachute test drops that had taken place at the Yuma Proving Grounds, an Army testing site located in Arizona. Each drop has an associated drop log which collects data about the parachute's decent.

We were able to isolate our desired position information using MATLAB parser code, provided by our sponsor. The parser made the information available as .m files (MATLAB specific files) and we could then build an additional parser program that could clean and write these locations to a CSV. Cleaning was an important step since some of the drops included missing readings which were replaced with all 0 values. These needed to be removed for our flight path file to work correctly, or else our camera object would

be directed towards the geographic point 0,0,0. Additionally, some of the logs included a portion of time where the location was being tracked in the airplane before the package was dropped. For each log we graphed the path to make sure we removed any portion that was inconsistent with a falling parafoil. The cleaned coordinates were saved to a CSV file in a specified format compatible with the Unreal Engine. Once we had the file, we could pair it with the camera object to recreate any of the original test drops in our virtual simulator.

Based on the parachute drop logs, we were also able to develop further MATLAB code for transforming the flight paths to new locations. This program allowed users to input a desired location for the parachute to land, along with the real-life flight path they would like to follow, and would return a CSV in the necessary format to be used to define a spline path in the Unreal Engine. It worked by calculating a difference vector for each of the dimensions, by subtracting it from a vector containing all entries of the final position. This resulted in a difference vector that could be added to a vector that contains entries of the new desired landing position. The same idea applied for longitude and altitude, with some adjustments for the different measures of altitude from meters above mean sea level (used while measuring the real-life drops) to meters from ellipsoid (the standard altitude measure in Cesium).

An example of this is starting with a real-life flight drop from Arizona, we can choose a new starting coordinate in New York City (NYC) as our desired landing point and transform the entire spline path to the new location. The resulting path will be written to a CSV file, then a user can upload the file to their desired Unreal Engine project and pair it with the camera object to simulate the drop in NYC. The two splines can be seen in Figure 6.5. With this added functionality we would be able to collect realistic aerial imaging data from anywhere in the world.

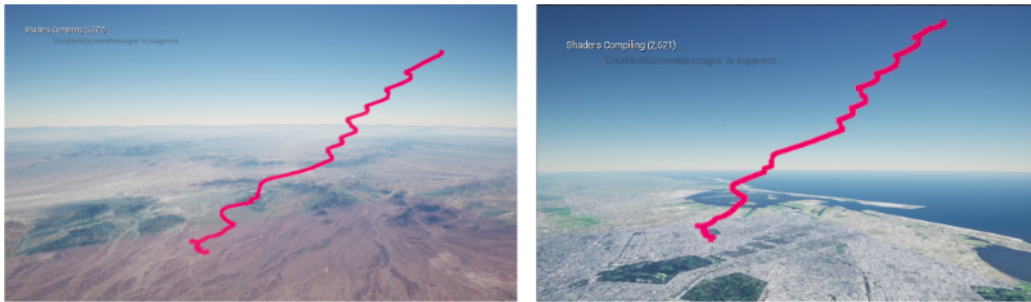


Figure 6.5: On the left is the spline path from a DEVCOM-SC parachute drop in Arizona. On the right is the same spline path transformed to New York City. This was done through MATLAB code and allowed us to collect realistic aerial imaging data from anywhere in the world.

Now that we had the path's coordinate points, we wanted our camera object to follow, we programmed a command to begin the flight of the object along the spline with the blueprint in Figure 6.6:

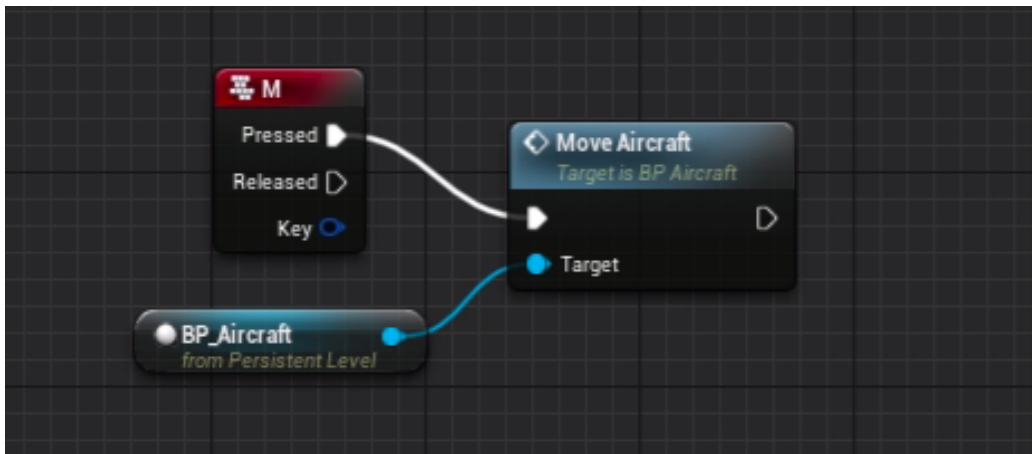


Figure 6.6: The blueprint code to move the object along the spline.

This allowed the object to follow along the path of a parachute and have an associated camera follow the descent. Next, we needed to figure out how to automatically capture images from the camera's perspective and save the corresponding coordinates. Our first solution used programmed key commands that were manually pressed each time the user wanted to take a picture.

The F9 keyboard click command is automatically linked with the screenshot command in the Unreal Engine terminal. With that fact in mind, we programmed the same keystroke to capture the WSG-84 coordinates of the object. This allowed the user to use one keystroke, the F9 key, in order to capture a screenshot. The screenshots are saved in the photos folder of the project, and the coordinates of the camera are saved as text to the project's log file.

This manual method, however, would not be sufficient for large scale data collection as it required a user to sit and continuously click the key for the duration of the flight. Not only that, but we needed to take pictures at regular intervals, and this would be a difficult task for a human to manually complete. Therefore, the next feature we created was an automated function to collect data. To do this, we used key nodes such as loops and timers. Figures 6.7 and 6.8 show the blueprints to collect images and locations respectively at a time interval set by the user.

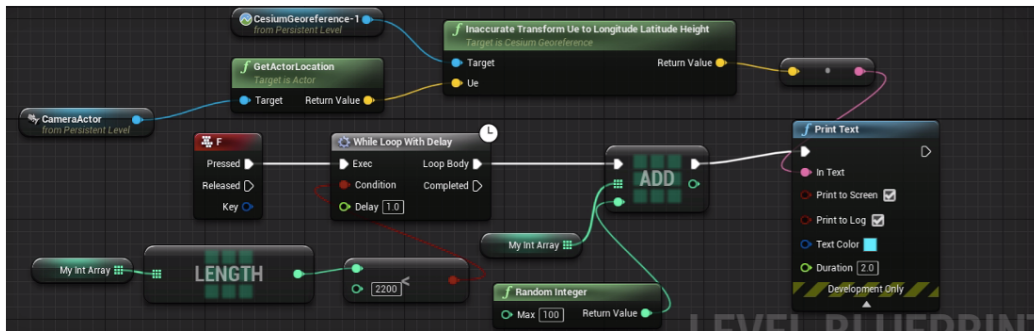


Figure 6.7: The blueprint code to collect locations upon clicked the F key. When the F key is clicked a loop will begin with a user-defined delay between each iteration. Within the loop, the location of the object is collected and saved to a log. The locations is in the form of latitude, longitude, and altitude

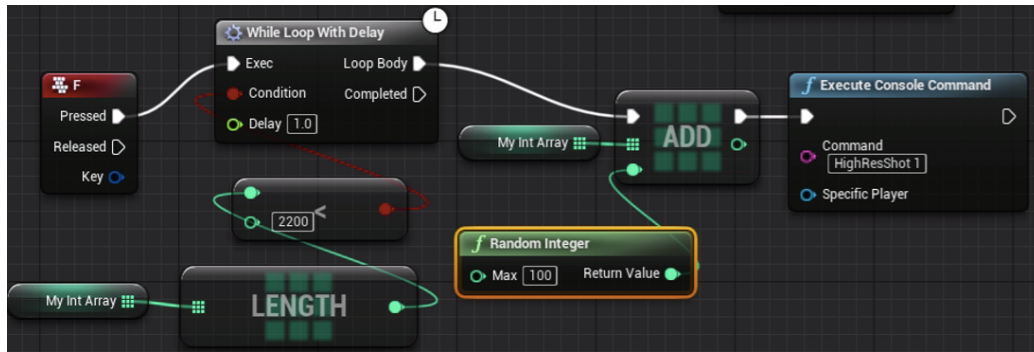


Figure 6.8: The blueprint code to collect images upon clicked the F key. When the the F key is clicked a loop will begin with a user-defined delay between each iteration. Within the loop, the command to take a screenshot is called. The screenshot is taken from the camera perspective. The images are saved in the screenshots folder of the project.

The simulator now had the capability to replicate flight drops and autonomously capture data. Next, we learned to use the Unreal Engine’s cinematic feature to record videos of the simulated drops. In addition to being able to collect aerial images, the cinematic tools in UE4 made it possible to record videos. We created project in a variety of locations such as Greenland, New York City, and the Amazon. Then we ran each simulator and utilized cinematic mode in UE4 to record the view of the camera. These videos were used to demonstrate the capabilities of the simulator.

6.1.3 Additional Settings

As mentioned previously, the Sun is a controllable element within Cesium. The Sun’s position can be changed using a built-in parameter to simulate different times of day in any celestial location. This allows the user to change the time of day in the simulator, in order to collect more diverse imaging data. The changing placement mimics sunrise, sunset, and dusk accurately to a location and season on the Earth. An example of the simulator at sunset can be found in Figure 6.9

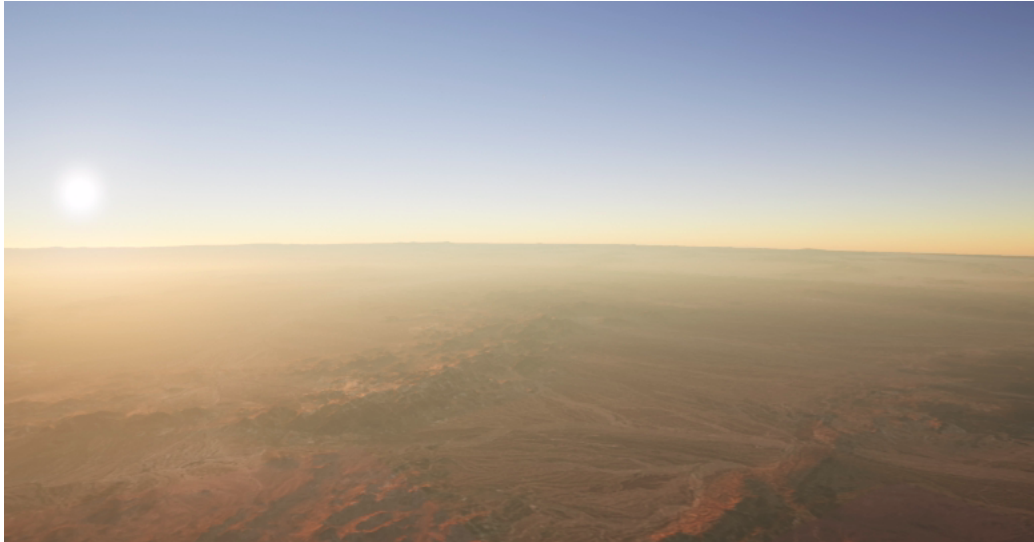


Figure 6.9: An image captured from our simulator in Arizona. The Sun's position can be changed using a built-in parameter to simulate different times of day in any virtual location. This allows the user to change the time of day in the simulator, in order to collect more diverse imaging data. The sun in this image is set to sunset

Weather is another feature within the Unreal Engine. We chose to experiment with the different types of fog in our simulator; volumetric fog, exponential height fog, atmospheric fog, and sky atmosphere. The different fog options vary in densities and altitudes, but we found volumetric fog gave the best effect for our simulation needs. The Unreal Engine also has the capability to program and develop more weather types, but we limited our development to fog as a proof of concept for weather capabilities.

More detailed information of our simulator development, including step by step instructions for building a new simulator project or running a project we built are provided in the Simulator Manual, Appendix Section 10.1. Once completed, we used our simulator to collect the information for creating our dataset of labeled images.

6.2 Dataset

Once we had the necessary information gathered from running our simulator, we needed to structure it in a way that could be used for machine learning applications. This included matching images with their corresponding locations, pairing images and calculating their differences in position, applying preprocessing techniques, and splitting data into training, testing, and validation sets.

6.2.1 Predictors and Labels

From the simulator we were able to collect many images with the location saved to the UE4 project log. We created a parser for the logs such that we could match each image up with its recorded location. Our goal was to give the neural network two images, and have it predict the distance between them. So in addition, our parser also paired images and calculated the difference in location in each dimension (latitude, longitude, and altitude). We also included the location for the first image as predictors in our dataset. Initially, we wanted to predict the change in position of all three dimensions, but soon we soon narrowed this focus down to predicting only the change in altitude, as it was the most meaningful measurement collecting using our simulator.

With this goal in mind, we tried a variety of datasets of different sizes and structures, but eventually chose one that satisfied the needs of our problem. We took data from eight drops, set over the Yuma Proving Grounds, with images taken each second. Then we paired up every picture within each drop that had a difference of less than 200 meters in altitude, but greater than zero meters. The maximum restriction ensure that we did not have images that were so far apart that they had not overlapping features. The minimum restriction prevented occasional upward changes in altitude and zero difference values that were problematic for calculating percent error. This left us with a dataset of about 450,000 image pairs, labels with the initial image's altitude, latitude, and longitude, and the difference between pictures in the same three dimensions. An example of a datapoint can be seen in Table 6.1. We set aside 20 percent of this data to ultimately be used for a final validation of our network, leaving us with about 380,000 samples to use for training and testing our network. Our training data had a mean

difference in altitude of -97.788 meters (the negative indicates decent) and a standard deviation for the difference in altitude of 57.307 meters. Since each of our samples were labeled with their true response values (change in altitude, change in longitude, and change in latitude), we were working with a supervised learning problem.

Image 1	Lon	Lat	Alt	Image 2	ΔLon	ΔLat	ΔAlt
Figure 6.10	-114.3	33.31	3801.94	Figure 6.11	0.001	0.002	-61.13

Table 6.1: An example of a datapoint from our input.



Figure 6.10: The first image in the dataset. This image was captured from our simulator following a DEVCOM-SC flight path.

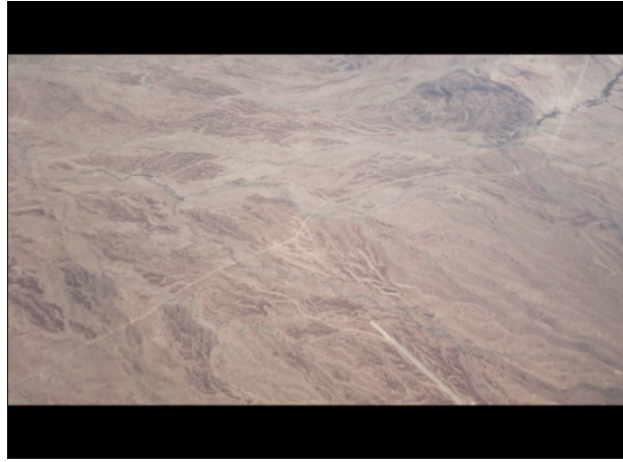


Figure 6.11: The second image in the dataset. This image was captured from our simulator following a DEVCOM-SC flight path.

6.2.2 Training Subsets

Given hardware constraints we quickly realized that training on this entire dataset of high-quality images would not be possible, so instead we took random samples from this larger dataset to train our neural network. Through a series of cross-validations, we settled that 75,000 pairs of labeled images balanced our needs for converging errors and training time. We took four random samples of labeled image pairs from the larger dataset, without replacement, and used these four subsets to run our model training.

6.2.3 Preprocessing

We applied a series of preprocessing steps to our datasets to prepare them for machine learning applications. As a baseline, we used images that we grayscaled and consistently sized to 240×426 pixels. These preprocessing techniques were important for the consistency and ease in our next step of creating our datasets, flattening. Flattening images is an important step in image processing, and is made easier by grayscaling. Grayscaled images are simply an array of pixel values, whereas color images consist of three arrays of pixel values, one for each color: red, blue, and green. The problem with images being structured as arrays are that neural networks, and most other machine learning algorithms, depend on each sample being formatted as a

vector. In order to modify the image to fit this constraint, each row of the image array is appended to the previous, resulting in the necessary vector. For our images, since they were each consistently sized to 240×426 , an image could be flattened and transformed to a $1 \times 102,240$ vector. Therefore each sample consisted of two images, along with the initial picture's coordinates, it means each sample consisted of 204,483 variables.

6.3 High Performance Computing

Companies like Amazon [60], Microsoft [61], and Google [62] have data centers all over the world and can offer limited HPC services to anyone for free. Our team used the Google Colab service. The service allows users to collaboratively work on python notebook files. It allows users to run code on their servers, and they offer limited GPU capabilities as well [63]. However, we quickly discovered that more resources would be necessary to process our high dimensional data and train complex, deep learning models.

WPI maintains an HPC system, called Turing that our project team used for this project. The software that is used to manage Turing's resources is called SLURM (Simple Linux Utility for Resource Management) [54]. To make requests through SLURM one must either use the 'sinteractive' or 'sbatch' functions. Both functions allow the user to specify the number of CPUs, the amount of RAM in gigabytes, the number and type of GPU, and how long to request these resources for. Typically, the longer resources are requested for, the longer it will take to be granted them. The sinteractive function is useful for debugging a program and thus is usually used to request resources for a short amount of time since the user must be on standby to begin using the resources once they have been allocated. The sbatch function will run in the background waiting for resources and will automatically run a specified program once granted resources. Sbatch is useful for running programs that have already been debugged and require resources for a long time.

Our team required a workflow that allowed us to write code and easily run a test directly on the Turing system while also being shared simultaneously among our team members. We experimented with several coding environments to improve our workflow. We tried environments like Git and Jupyter Notebooks. We relied on a Git repository that was shared between the Turing file system and our home computers. This allowed our team to

write code on our computers and then upload them to the Turing computer to be run through the ssh connection. Another environment included hosting a Jupyter Notebook server on Turing. This was useful in allowing the team to create files directly on the server and to run them without having to transfer files or make shell commands. The issue we encountered was with the size of our code base expected to increase rapidly, we needed to be able to create standard python files to improve modularity of code. Visual Studio Code was the platform that offered our team everything we needed. Visual Studio Code's remote connection feature allowed our team to effectively develop code directly on Turing with no need to have separate connections of transfer files [64].

On challenge we faced was finding an efficient method for moving the training and testing data from the central processing unit (CPU) to graphics processing unit (GPU). In order to do this, we needed code to instruct PyTorch to move specific data from the RAM on the server to the memory on the GPU. Since all calculations that require a GPU must have all the data involved on the GPU's memory, a balancing act is created between the size of the neural network and the size of the training and testing batches. There are a few ways to accomplish this goal, we first decided to use P100 GPUs, which reduced our training and testing computation time tenfold compared to the CPU. We also initially transferred an entire dataset to the GPU at once. This method was very wasteful, because you only need the current batch to be on the GPU. When we made the change to only transfer one batch of training and testing data at a time, we were able to use much larger neural networks which improved our results. As the size of our dataset grew, our batch size went from 16 image pairs to 128 image pairs. The P100 GPUs we were using no longer supported that many image pairs to be transferred at once. We therefore needed to start requesting A100 GPUs, which satisfied our groups needs for the remainder of the project.

6.4 Neural Networks

Throughout our project our team experiments with different neural network architectures and hyperparameters. Our very first network was simple and consisted of just three basic layers: an input layer, one hidden layer, and then an output layer. The input was the size of two flattened images and the three coordinate values of the first picture (102,243) by 1024. The output

was of size three to reflect the latitude, longitude and altitude predictions. The hidden layer converts the 1024 neurons to 16, and the output layer converts the 16 neurons to either a scalar or 3x1 vector. The activation function for used in the hidden layer was rectified linear (ReLU). The graph of the function is seen in Figure 6.12.

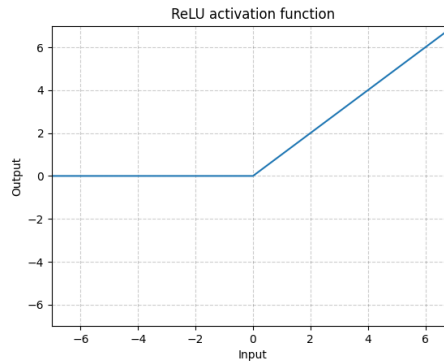


Figure 6.12: The graph of the ReLU activation function. This activation function works by setting all values less than zero to zero, and any input above zero remains the same.

The loss metric used was mean squared error. Given this network outputs three variables, we designed our loss function to be the averaged across the mean squared error (MSE) of each position variable. This initial structure inspired our comparison between multi-task and single-task networks, as we moved away from the goal of predicting all three dimensions to only altitude.

6.4.1 Baseline Network

Our next network was deeper and consisted of 9 layers: input, output and seven hidden layers. The input layer reduces the flattened images from their original size to 1024 neurons. Each consecutive layer reduces the dimensionality of the network by a factor of two until the output layer which will either convert a vector of 8 neurons to a single scalar (to represent the change in altitude for a single task model) or a 3x1 vector (to represent the change in latitude, longitude, and altitude for the multi-task model). The philosophy of testing this network was that a slow reduction of information

between layers will preserve the important information and concentrate it like a funnel.

Layer	Input Size	Output Size
Input: 1	204,480	1024
Hidden: 2	1024	512
Hidden: 3	512	256
Hidden: 4	256	128
Hidden: 5	128	64
Hidden: 6	64	32
Hidden: 7	32	16
Hidden: 8	16	8
Output: 9	8	3

Table 6.2: The layer architecture for our baseline multi-task NN. This model was used as the basis for our experiments before we performed experiments altering the architecture

Layer	Input Size	Output Size
Input: 1	204,480	1024
Hidden: 2	1024	512
Hidden: 3	512	256
Hidden: 4	256	128
Hidden: 5	128	64
Hidden: 6	64	32
Hidden: 7	32	16
Hidden: 8	16	8
Hidden: 9	8	1

Table 6.3: The layer architecture for our baseline single-task NN. This model was used as the basis for our experiments before we performed experiments altering the architecture

We also set our hyperparameters for the duration of the project at this step; we decided to use 750 epochs, batch sizes of 128 images, and step size

of 0.001. These setting balanced our needs for precision but also training time, as these two goals conflict. This model became our baseline for testing a variety of preprocessing methods, new activation functions, and modified architectures.

Neural Network Hyperparameters	
Epochs	750
Batch Size	128
Step Size	0.001

Table 6.4: The hyperparameters for the neural network. These parameters were set for the whole project.

6.4.2 Error Metrics

We used three different important metrics, mean squared (MSE), root mean squared error (RMSE), and percent error for the difference in the real altitude from the predicted. The equation for MSE is as follows where n is the number of datapoints in a batch and y_i is the true value and \hat{y}_i is the predicted value.

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

MSE and RMSE are very similar, but RMSE, as the name implies, square roots MSE in order to return the value to its original units.

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}.$$

Percent error is calculated by finding the difference between a prediction and the true value, then dividing by the true value.

$$\text{Percent Error} = \left| \frac{y_i - \hat{y}_i}{\hat{y}_i} \right|$$

Percent error is a measure of error with respect to the magnitude of the value. These three values, MSE, RMSE, and percent error, typically grow and shrink together. All three metrics share the rule that smaller values indicate a better model, with 0 being perfect.

6.4.3 Single-task and Multi-task Learning

Another structural change we explored for our neural networks was the effects of single-task verses multi-task neural networks. Single-task networks are a traditional model where the loss function is guided by one error and the model returns a single output, the desired prediction value. In our case our single-task network was guided by the MSE for the altitude difference between the image pair, as seen below, which was also the value returned by our NN.

$$MSE_{alt} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

Multi-task neural networks, as the name implies, try to predict multiple variables at once, opposed to a single desired value. This is mainly driven by the loss function incorporating the multiple values of interest. In our case we defined our multi-task loss function as average MSE of the three different dimensions, latitude, longitude, and altitude. This function was precisely defined as:

$$MSE_{multi} = \frac{1}{3}(MSE_{alt} + MSE_{lat} + MSE_{lon})$$

We wanted to see if predicting the three dimensions in tandem could help improve the accuracy of our single value of interest, the change in altitude.

6.4.4 Model Validation

To ensure our basic neural networks functions as we expected, we created a dataset that consisted of the same picture 200 times. It was labeled accurately as a repeat, meaning that the differences in each dimension were always 0. We ran this dataset through our models to ensure that the error converged to zero, and in turn checked that the model functioned as expected. This was an important check for our code when we started made changes to our code and received unexpected results, we could help debug using this method.

6.4.5 Additional Architectures

From our baseline network we had a variety of changes we could make to our model to test their effect on our models' accuracy. First, we tried

changing the activation functions we used, testing both Leaky ReLU and GELU. The equation for Leaky ReLU is $f(x) = \begin{cases} x & x \geq 0 \\ 0.01x & x < 0 \end{cases}$ so it is different from ReLU since it avoids nodes from becoming inactive when set to zero [47]. GELU is an activation function defined as $f(x) = x\Phi(x)$ [50]. We decided to test this activation function as it has been shown to perform well for image processing models.

We also explored adding larger input layer, to reduce the dimension of our data more slowly. Similarly, we tried adding layers that did not reduce the dimension at all, just applied our activation function an extra time during the training process. Additionally, we tried making a shorter network consisting of only seven layers, to check if a less complex network would result in similar accuracy to our baseline, nine layer model.

6.4.6 Cross-Validation

Beyond calculating training errors, we also conducted cross-validation using data that had not been used in the training or testing of the network. Calculating validation error is an important part of the hierarchical cross-validation process. Running this cross-validation yields our validation error metrics, which are a final check of each network's accuracy. Validation error metrics consistent with training errors shows that models have performed as expected, whereas high validation errors that are inconsistent with testing errors can indicate an overfitted model or another underlying issue. Our validation errors use the same metrics as our testing error, MSE, RMSE, and percent error, so they can easily be compared to the testing and training errors.

7. Results

This section details the results from our project. This includes the final simulator and its capabilities, the final structure of the datasets used to create our machine learning model, and the final summary of the different neural networks we tried along with their associated error reports and comparisons.

7.1 Simulator

We developed a parachute dropping simulator with many adjustable parameters. The simulator was developed using the Unreal Engine and the Cesium plug-in. The simulator contains the entire globe, built from satellite images. The simulator has four main functions: loading in parachute drop paths, a parachute object with an attached camera that descends along the path, automated data collection of images and locations, and modifiable features like time of day and weather.

The parachute drop paths can either be parsed from a real-life parachute drop log or generated in new locations using our MATLAB code. We parsed eight parachute drop logs, provided by DEVCOM-SC, and used them as the model to build new drops at new locations in the world. This original drop data was collected from tests at the Yuma Proving Ground in Arizona. A side-by-side comparison of an image from DEVCOM-SC and our simulator over Yuma Proving Ground can be seen in Figure 7.1. The simulator can be modified and saved as different projects, we created one for each original drop and nine drops in new locations around the globe, representing diverse terrain.

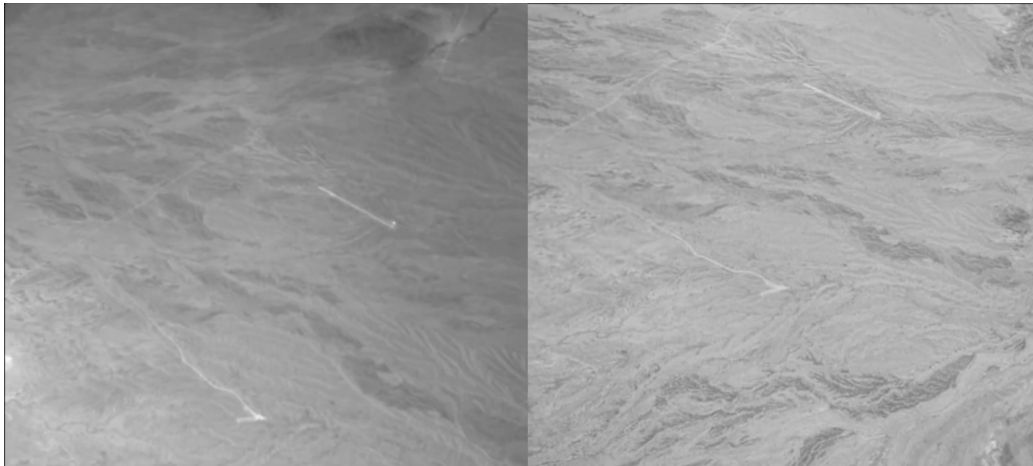


Figure 7.1: The image on the left is collected from a DEVCOM-SC parachute drop over the Yuma Proving Grounds in Arizona and the image on the right is collected from the same location within our simulator.

We created eight projects, based on the provided drop data, by loading in cleaned and properly formatted data to projects. We created MATLAB code for cleaning and formatting drops in their location for use in UE4 projects. These projects are now ready to be used and collect data, and were the locations we used to build our dataset.

We also built MATLAB code that takes in an ending location and will recreate a drop path pattern ending in the location specified by the user, mimicking one of the real-life drops, also specified by the user. This allows for drops to be created anywhere in the world. We created additional project using transformed drops in the following locations: the Amazon Rainforest, the Arctic, Greenland, the Lunar Craters, the Mississippi River Valley, New York City, the Sahara Desert, Sweden, and the Cranland Airport. We chose these locations since they represented a wide variety of different features, for example Sweden represented a coastal area with many islands whereas the Amazon Rainforest consisted of a densely forested landscape. We found that some snowy locations, like Greenland, did not offer usable data since the whole terrain was featureless and white. Examples of images from four of our simulation projects can be found in Figure 7.2. The figure shows the Lunar Craters (NV), Amazon Rainforest, Yuma Proving Ground, and New York City.



Figure 7.2: Synthetic aerial images captured from our simulator. Locations clockwise from the top left: Lunar Craters (NV), Amazon Rainforest, Yuma Proving Ground, New York City. The Yuma Proving ground example is taken at dusk with volumetric fog. This picture demonstrates the diverse functionality of our simulator.

Our simulator allowed us to collect labeled image data. We programmed functionality into our simulator to automate the collection of images and their associated locations throughout a parachute drop, at user specified time intervals, all with the click of a button. More instructions on using the simulator are provided in our manual, Appendix Section 10.1. We collected over 9,000 images that we used to build our dataset, set in the Yuma Proving Grounds. We also collected an additional set of images from a drop set in New York City, as a way to test the versatility of our simulator. In addition, we collected videos of the simulator in action, as an additional way of demonstrating its functionality.

To showcase the additional features that can be added and modified to the simulator we created three more projects that demonstrate time of day and weather settings. The simulator can have the sun set to anytime of the day, so one project showed the capability to make convincing dusk scenes. Weather is another feature within our simulation, and we were also able to add realistic fog to our projects. Therefore, the second project demonstrated the fog. The third combination we tried was an example of fog at night time to show how the two features can work together for a realistic visual effect. These combinations of visual effects in our simulated environment can be

seen in Figure 7.3.

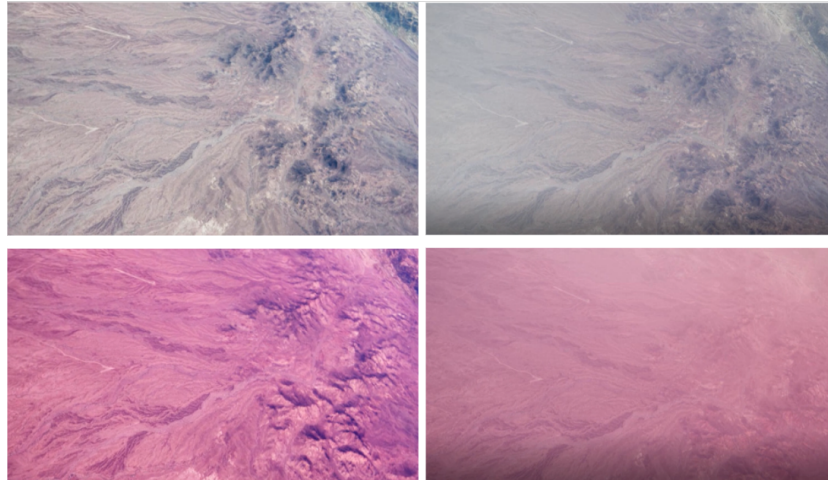


Figure 7.3: Shown are the four different variations of the Yuma Proving ground that can be made by combining fog and time of day. Clockwise from the top left: midday without fog, midday with fog, dusk with fog, and dusk without fog. This image demonstrates the different visual effects that can be achieved using these settings.

We created four tools to aid in using the simulator. First we created a parser code which allows users to input a parachute drop log and the parser will clean and extract relevant position information for simulating the flight in the Unreal Engine, along with properly formatting it to be compatible with the requirements of our simulator. This is separate from the parser provided to us from DEVCOM-SC. The second tool is a MATLAB code to generate a flight drop, based on a real testing drop path, given a desired landing location. The third tool we created was python parser code to extract the location information from project logs and pair it with the associated images, to result in the basic dataset for the simulators. We later developed further python code to construct more complicated datasets from one or more drops. The last tool we created is an instruction manual we wrote to create, edit, and use the simulator. The manual includes how to recreate the process we developed to build a virtual simulator from start to finish. It also explains how to perform the main functionality of the simulator including: how to use the parsers and MATLAB code, how to load in flight paths, record videos, capture data, add fog, change time of day, and troubleshoot common errors.

The full manual can be found in appendix section 10.1.

7.2 Datasets

We built a dataset which we subdivided to be used for the training, testing, and validation of our deep learning models. This section will detail the specifics of our final datasets, such as image processing techniques we used, and their effects on our final results.

7.2.1 Preprocessing

Image preprocessing was a crucial step in preparing our data for ML applications. In this section we will describe our results from grayscaling, resizing, and equalization in the context of creating meaningful datasets.

7.2.1.1 Grayscale

The first preprocessing method we applied to the images in our dataset was grayscaling. The first reason we did this is because the current cameras mounted on parafoils are in grayscale, so this modification better mimics the real-life data that we are attempting to simulate. We also did this for the purposes of reducing dimensionality. Colored images consist of three times as much information as grayscale pictures, since each pixel has three assigned values: red, blue, and green. Alternately, in grayscale, each pixel only has one value, which represent its intensity. This can be done by implementing the equation provided in the OpenCV python package [29]:

$$RGBtoGray : Y = 0.299R + 0.587G + 0.114B.$$

We never ran our model on colored images but have good reason to believe this modification resulted in lower errors. For one, it reduces the size of the data, allowing us to include up to three times as much data in our training before running out of memory. Additionally, to train a model through 750 epochs, it took between 14 and 20 hours, so higher dimensional data would have increased the training time for models based on datasets with the same number of pictures, potentially resulting in us decreasing our number of epochs or reducing the sample size to improve computational time.

We predict that either of these trade-offs would have resulted in worse model performance compared to our grayscaled dataset and corresponding model.

7.2.1.2 Resizing

Resizing was another necessary step in model performance. First, by making all the images the same size, we provided consistency between images. This was important since it allowed us to combine flattened images into a larger matrix, the necessary structure for all ML models, including neural networks. This also allowed us to reduce the dimensionality of our data, similar to converting from color to grayscale. Our raw images were collected at a higher resolution than necessary, approximately 1280 by 650 pixels, whereas the reduced images were consistently sized as 240 by 426. This means resizing a pair of images brought the dimension of the data from 1,664,000 variables down to only 204,480, which is less than one-eighth of the original size.

Again, similar to grayscaling, this reduction in dimensionality also significantly increased the number of images we could use for training. It also significantly improved computational time of our model. We did not perform experiments to verify this conclusions, but believe them due to the structure and nature of the models.

7.2.1.3 Equalization

We investigated multiple methods of equalizing images. One method we tried was Contrast Limited Adaptive Histogram Equalization (CLAHE), which equalizes pixel values in smaller sub-windows and is a function in the OpenCV package [29]. This is often the standard for image equalization, but we found it to perform equal to or worse than non-equalized images, as demonstrated in the data in Table 7.1. We believe this is due to the window equalization feature. Imagine two images are similar, but taken at slightly different angles. The centers of the pictures will look almost identical, but the edges might have some differences. This could result in the windows in each picture capturing different sections of features, and then the equalization could assign different pixel values to features that were originally the same.

Variables			Testing Error	
Single or multi?	Equalized	Activation Function	Approximate RMSE	Percent Error
Single	None	ReLU	58.656	61.241
Multi	None	ReLU	28.513	29.770
Single	CLAHE	ReLU	58.683	61.270
Multi	CLAHE	ReLU	34.609	36.134

Table 7.1: This table demonstrates the ineffectiveness of the CLAHE method as a preprocessing method for our data. As it can be seen, the single and multi-task networks performed similarly with and without the CLAHE method implemented. Note, the RMSE in this table is an approximated based on an early method of our loss function.

This problem led us to two important realizations. First, we need to use an equalization method that applies to the whole picture at once, not sub-sections. We also need to equalize pairs of images together, not individually, as we had first done. This is for a similar reason as to why we believe the CLAHE method was not effective for our problem; if two images have many features in common, but one contains a very dark patch which is missing from the other, then equalizing the images separately, even if it is over the whole image, can result in pixel values that were originally the same to be assigned to new values that are different.

Therefore, our final method of equalization was to first concatenate the images, one on top of the other, and then apply histogram equalization to the entire image. This method helped ensure consistency in values between pairs of images, while also allowing equalization to improve contrast.



Figure 7.4: The image above shows an example of histogram equalization on a grayscale image collected using our simulator. The technique increases contrast in the image.

The result of our new equalization method showed clear improvement in our error metrics. Run on our basic network, with ReLU as the activation function, we can see in the case of both the single- and multi-task networks that the testing RMSE reduced by about 7 meters and the testing percent error decreased by about half. We also found that the cross-validation results were very inconsistent with testing for all models except the multi-task NN with equalization applied, highlighting that model as our best so far. We ran this experiment for the four randomly sampled datasets, and the averaged results are provided in Table 7.2.

Variables			Testing Error		Validation Error	
Single or multi?	Equalized	Activation Function	RMSE	Percent Error	RMSE	Percent Error
Single	None	ReLU	25.607	0.489	78.474	1.379
Multi	None	ReLU	25.664	0.484	77.809	1.005
Single	Pairwise	ReLU	18.540	0.262	79.264	3.249
Multi	Pairwise	ReLU	18.673	0.291	18.907	0.294

Table 7.2: The averaged results from the experiments comparing the effects of pairwise histogram equalization. We can see in the case of both the single and multi-task networks that the testing RMSE reduced by about 7 meters and the testing percent error decreased by about half. We also found that the cross-validation results were very inconsistent with testing for all models except the multi-task NN with equalization applied, highlighting that model as our best so far.

7.2.2 Distance Limits

We decided to implement a maximum distance between images in a pair when building our dataset. We chose to add a upper limit since we wanted to make sure that images were not selected from so far apart in the drop that it could cause them to have no reasonable relationship. For example, if an image was taken at the very beginning of the flight path and showed an aerial view of the landscape with key features, and another was taken just 40 meters above the ground and a mile away, then there may not be any shared features between the images. An example of this scenario is provided in Figure 7.5. Therefore, we selected a maximum difference in altitude of 200 meters when building our dataset.

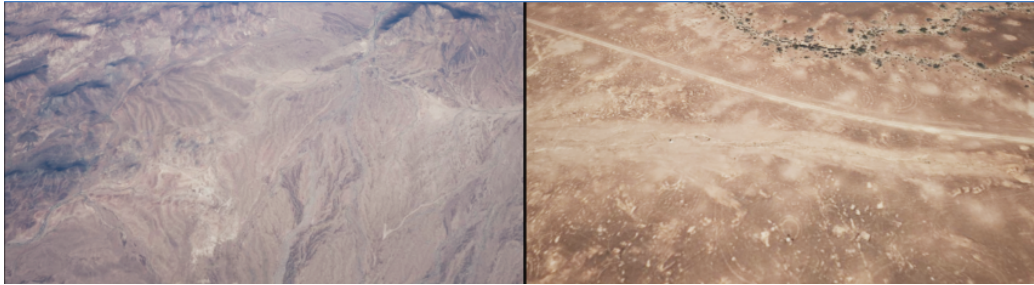


Figure 7.5: This pair of images has more than a 200m difference in altitude: the first picture is very far away from the ground while another is very close. Because of this there may not be any shared features between the images.

We also added a minimum distance of 0, which cut out occasional positive differences in data, and problematic 0 values for percent error calculations. This condition left us with about 450,000 valid pairs of images from the eight simulations set in the Yuma Proving Grounds to use for our training, testing, and cross-validation.

7.2.3 Deliverables

We used four final datasets for training our neural networks. They were processed from comma-separated values (CSV) files and converted to PyTorch files. Each of the four datasets were made by randomly choosing 75,000 image pair samples, without replacement, from the larger set of qualified training pairs. Each sample consisted of the three dimensions of the first picture in the pair (latitude and longitude, measured in degrees, and altitude, measured in meters about sea level) and the pair of images, both grayscaled, resized to 240 by 426, and pairwise equalized using histogram equalization. Each sample was labeled with the difference in latitude, longitude, and altitude, measured again in degrees and meters above sea level, respectively.

Our CSV files include only the image locations, not the final preprocessed and flattened images, but was a helpful step for us to validate our dataset before transforming it into a PyTorch (.pt) file.

7.3 Neural Network

Neural Networks can be modified by adjusting the parameters used to train the model. There are also features of the architecture that can be changed to modify the network in the hopes of improving prediction accuracy. In our neural network we adjusted the number of epochs used in training, tested ReLU, Leaky ReLU, and GELU to find a good activation function for our model, tried different layer architectures, and determined performance based on a set of error metrics.

7.3.1 Epochs

When choosing the number of epochs we would use to train our networks, we needed to balance the need for repetition in training to the time required for extensive testing. We settled on using 750 epochs throughout our experiments. This value was well past the knee in the testing error curve, but we were still observing minimal improvements up to this point, as shown in Figure 7.6. To train our models through 750 epochs in batches of 128 took approximately 15 hours per network, so we decided this value balanced our needs for precision and also time. This value of epochs was chosen in partnership with our dataset size of 75,000 image pairs, as smaller datasets did not always converge.



Figure 7.6: The graph is an example of convergent training and testing errors due to insufficient epoch values and too few training samples available to the network. We were able to identify 750 epochs was sufficient for our final training set, which avoided graphs of this nature.

7.3.2 Activation Functions

We tested three different activation functions on our baseline, nine layer network: ReLU, Leaky ReLU, and GELU. ReLU is a popular activation function for initial testing of neural networks, so we began there. In the last decade ReLU has proven to result in good accuracy and favorable computation time compared to other activation functions [65]. The average RMSE for the multi-task normalized function on the baseline multi-task network using ReLU is 18.673 meters and for the single task network the average RMSE for the experiments was 18.540 meters, and a specific breakdown of errors for each of the four training sets are provided below in Figures 7.7, 7.3, and 7.4.

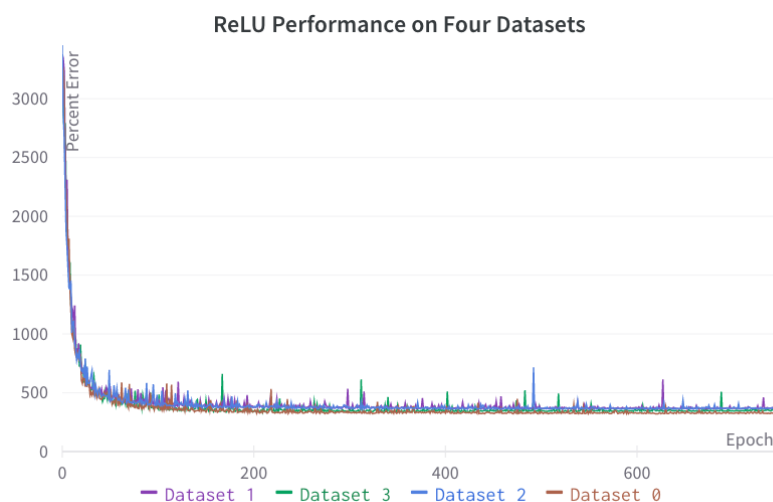


Figure 7.7: The testing loss for our datasets run with the ReLU activation function. The datasets show a smooth convergence to a consistent MSE of about 350.

Normalized, Single-task with ReLU			
	MSE	RMSE	Percent Error
Dataset 0	335.662	18.321	0.240
Dataset 1	348.166	18.659	0.308
Dataset 2	362.548	19.041	0.259
Dataset 3	328.477	18.124	0.240
Average	343.713	18.536	0.262

Table 7.3: The breakdown of our four tests for the single output network using ReLU as the activation function. Averaging over the four experiments the RMSE is 18.53 meters.

Normalized, Multi-task with ReLU			
	MSE	RMSE	Percent Error
Dataset 0	327.976	18.110	0.254
Dataset 1	346.760	18.621	0.376
Dataset 2	365.185	19.110	0.262
Dataset 3	354.875	18.838	0.273
Average	348.699	18.670	0.291

Table 7.4: The breakdown of our four tests for the multi-task network using ReLU as the activation function. Averaging over the four experiments the RMSE is 18.673 meters.

We then considered Leaky ReLU as the next activation function to test. Leaky ReLU is different from ReLU since it keeps weights from being classified as 0, meaning that neurons are not inactivated without opportunity to be reintroduced into the network at later layers [47]. The results from Leaky ReLU in our baseline, multi-task network was an average RMSE of 17.528 and the single-task network has an average RMSE of 17.952, which was slightly improved from our model using ReLU. As seen in Figure 7.8, Leaky ReLU was more prone to large spikes in error during training, though. The breakdown of each of the four individual experiments' results are provided in Figures 7.5 and 7.6.

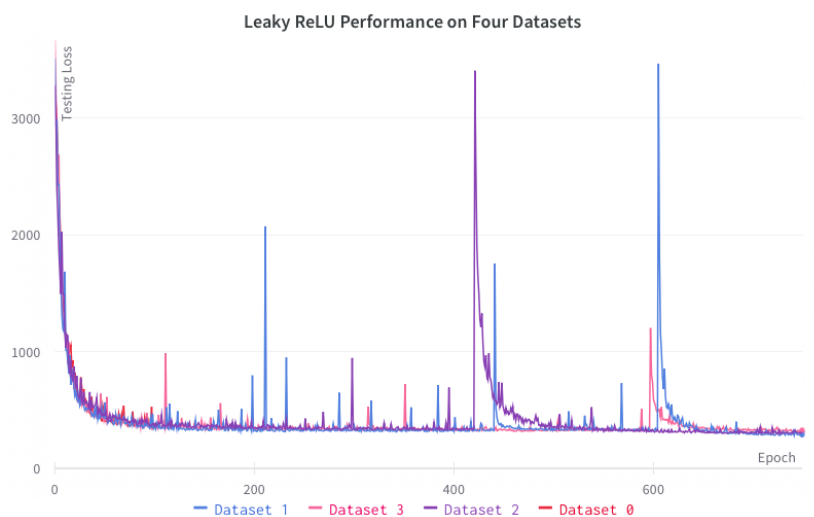


Figure 7.8: The testing loss for our datasets run with the Leaky ReLU activation function. The datasets show a smooth convergence to a consistent MSE of about 325.

Normalized, Single-task with Leaky ReLU			
	MSE	RMSE	Percent Error
Dataset 0	260.932	16.153	0.233
Dataset 1	330.028	18.167	0.310
Dataset 2	358.393	18.931	0.225
Dataset 3	339.739	18.432	0.262
Average	322.273	17.921	0.257

Table 7.5: The breakdown of our four tests for the single-task network using Leaky ReLU as the activation function. Averaging over the four experiments the RMSE is 17.952 meters.

Normalized, Multi-task with Leaky ReLU			
	MSE	RMSE	Percent Error
Dataset 0	302.196	17.384	0.275
Dataset 1	287.701	16.962	0.274
Dataset 2	311.733	17.656	0.259
Dataset 3	327.277	18.091	0.248
Average	307.227	17.523	0.264

Table 7.6: The breakdown of our four tests for the multi-task network using Leaky ReLU as the activation function. Averaging over the four experiments the RMSE is 17.528 meters. The italicized line for dataset 1 indicate the model we ultimately determined to be our best model.

Finally, we tested out a third activation function, Gaussian Error Linear Units (GELU). GELU has been research and shown as a powerful activation function for image processing purposes [42], so we put it to the test in our problem. Our results were slightly worse compared to Leaky ReLU, the previous front runner, but did perform better than ReLU. The results from GELU in our baseline, multi-task network was an average RMSE of 18.510 and the single task network has an average RMSE of 18.370. The breakdown of each individual experiments' results are provided in Figures 7.9, 7.7, and 7.8. Figure 7.9 shows a smooth convergence of error, similar to ReLU, and different from Leaky ReLU.

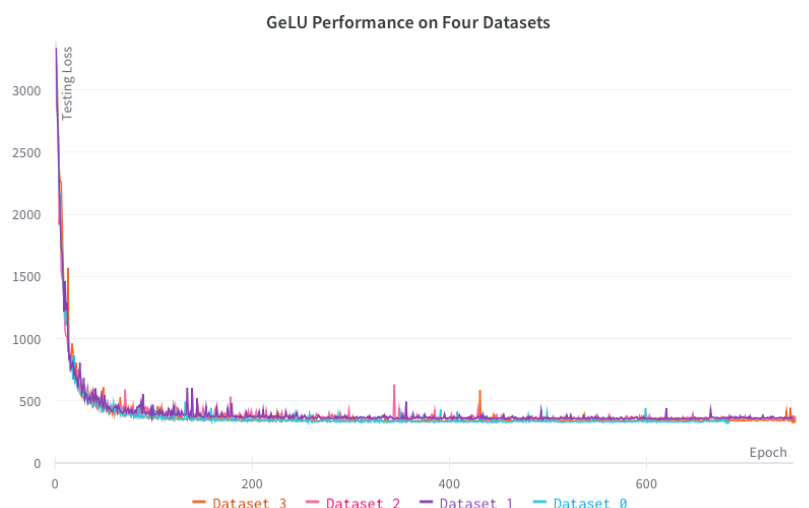


Figure 7.9: The testing loss for our datasets run with the GELU activation function. The datasets show a smooth convergence to a consistent MSE of about 340.

Normalized, Single-task with GELU			
	MSE	RMSE	Percent Error
Dataset 0	313.538	17.707	0.251
Dataset 1	323.208	17.978	0.279
Dataset 2	362.224	19.032	0.269
Dataset 3	350.790	18.729	0.243
Average	337.440	18.362	0.261

Table 7.7: The breakdown of our four tests for the single output network using GELU as the activation function. Averaging over the four experiments the RMSE is 18.370 meters.

Normalized, Multi-task with GELU			
	MSE	RMSE	Percent Error
Dataset 0	313.538	17.707	0.251
Dataset 1	360.078	18.976	0.362
Dataset 2	357.839	18.917	0.283
Dataset 3	339.036	18.413	0.238
Average	342.623	18.503	0.283

Table 7.8: The breakdown of our four tests for the multi-task network using GELU as the activation function. Averaging over the four experiments the RMSE is 18.510 meters.

The results of these activation function tests, along with comparisons to the histogram equalized experiments are provided in the table below. The averaged error metrics show that Leaky ReLU had the lowest error based on both RMSE and percent error, but not a clear better model between single- and multi-task. The single-task network yielded a higher RMSE but a lower percent error compared to the multi-task network which has a lower RMSE but higher percent error. We differentiated the multi-task network as our best model, since it performed much better than the single task counterpart in cross-validation testing as discussed in an upcoming portion of the Results, Section 7.3.4. Moreover, it is interesting to point out that while Leaky ReLU performed best in the final comparison of error metrics, it had the least smooth convergence of the different activation functions we tested. The full table of results can be seen for the testing errors in Table 7.9.

Variables			Testing Error	
Single or multi?	Equalized	Activation Function	RMSE	Percent Error
Single	Pairwise	ReLU	18.540	0.262
Multi	Pairwise	ReLU	18.673	0.291
Single	Pairwise	GELU	18.370	0.261
Multi	Pairwise	GELU	18.510	0.283
Single	Pairwise	Leaky ReLU	17.952	0.257
Multi	Pairwise	Leaky ReLU	17.528	0.264

Table 7.9: Report of average error metrics given different activation functions used in our baseline neural network structure. The bolded line for the multi-task, pairwise normalized network that uses Leaky ReLU activation is the best network we determined based on lowest RMSE and it is most consistent with the cross-validation errors discussed in the next section.

7.3.3 Layers

We explored a series of architecture designs when working to improve our neural network’s accuracy. In very early stages we used a network as simple as three layers: an input layer, one hidden layer, and an output layer. Soon we revised to a more complex network, with nine layers: an input layer, seven hidden layers, and an output layer. This network was used as a baseline for deciding the other metric we used, then we changed the number and size of layers as our last architecture change during testing. Results from the baseline network structure were detailed in previous sections.

We first tested a series of three new networks compared to our baseline, with other parameters held constant. Based on our previous experiments we continued to use histogram equalized pictures pairs and continued further testing using Leaky ReLU as our activation function. One was a shallower version, which we called our shorter layer structure; we tested this to see if the results were comparable to the baseline, then we know a less complex model would be preferable in terms of over fitting and computation time. A visual of this structure is provided in Figure 7.10.

We also tried two architectures with modified layer compared to our baseline model. One had layers that always decreased in dimension, but started at a larger value for the first input layer. We called this model our larger layer structure. The other model had some layers that did not reduce dimen-

sion, but applied the activation function. This model was called out straight layer function. Diagrams of these model structures are provided in Tables 7.11 and 7.12 respectively.

Layer	Input Size	Output Size
Input: 1	204,480	2048
Hidden: 2	2048	1024
Hidden: 3	1024	512
Hidden: 4	512	128
Hidden: 5	128	32
Hidden: 6	32	8
Output: 7	8	single: 1 multi:3

Table 7.10: The shallow layer architecture.

Layer	Input Size	Output Size
Input: 1	204,480	4096
Hidden: 2	4096	2048
Hidden: 3	2048	512
Hidden: 4	512	256
Hidden: 5	256	128
Hidden: 6	128	64
Hidden: 6	64	32
Hidden: 6	32	16
Hidden: 6	16	8
Output: 7	8	single: 1 multi:3

Table 7.11: The layer architecture beginning with larger initial layers.

Layer	Input Size	Output Size
Input: 1	204,480	2048
Hidden: 2	2048	1024
Hidden: 3	1024	256
Hidden: 4	256	256
Hidden: 5	256	64
Hidden: 6	64	32
Hidden: 6	32	8
Output: 7	8	single: 1 multi:3

Table 7.12: The straight layer architecture.

We tested these new models on only one of our datasets initially to see if there was promise to any of our new models. We saw that both the larger layer and straight layer single- and multi-task models performed worse than our averaged baseline network, when measuring error using RMSE. The single-task shorter model also performed worse than our baseline. But we found that the shorter multi-task network consistently performed slightly better than our baseline network, since both RMSE and percent error measures for this trial were lower than the average measures for our baseline model. This indicated that a less complex model could have the potential to perform better than our baseline model. Table 7.13 shows the error metrics from one dataset test of the shallow structure.

Variables		New Architectures		
Single or Multi?	Layer Structure	MSE	RMSE	Percent Error
Single	Straight	318.320	17.842	0.342
Multi	Straight	322.673	17.963	0.001
Single	Larger	329.667	18.157	0.284
Multi	Larger	630.166	25.103	0.588
Single	Shorter	334.532	18.290	0.300
Multi	Shorter	300.133	17.324	0.250

Table 7.13: Testing error metric results from running different architectures on one dataset. We saw that both the bigger input layer and straight layer models performed worse than our baseline network, but saw that the shallower network performed comparably.

7.3.4 Cross-Validation

We completed the last step of our hierarchical cross-validation using our validation set. This was a test of 25,000 image pairs that has not been used in the training of our model. We found that there was consistency between our testing and validation errors for our multi-task neural networks, but very large validation errors (typically around 300%) for our single task neural networks. We cross validated the saved neural networks; in a few cases our HPC resources expired before the model could be saved, and therefore cross-validated, but the table of averaged testing and validation errors in Table 7.14 are based on all the saved models we had available. Based on our cross-validation results we determined our best model was the nine layer multi-task network, with Leaky ReLU as the activation function.

Variables			Testing Error		Validation Error	
Single or multi?	Equalized	Activation Function	RMSE	Percent Error	RMSE	Percent Error
Single	None	ReLU	25.607	0.489	78.474	1.379
Multi	None	ReLU	25.664	0.484	77.809	1.005
Single	Pairwise	ReLU	18.540	0.262	79.264	3.249
Multi	Pairwise	ReLU	18.673	0.291	18.907	0.294
Single	Pairwise	GELU	18.370	0.261	78.867	3.226
Multi	Pairwise	GELU	18.510	0.283	18.697	0.290
Single	Pairwise	Leaky ReLU	17.952	0.257	79.012	3.222
Multi	Pairwise	Leaky ReLU	17.528	0.264	17.493	0.277

Table 7.14: Report of validation error metrics given for the different network architectures we tested. Based on our cross-validation results we determined our best model was the nine layer multi-task network, with Leaky ReLU as the activation function. The best result is displayed in bold.

This inconsistency between the single- and multi-task neural networks could imply that multi-task neural network has a smoother loss landscape than that of a single-task network [66]. This could have important implications for reducing training time for neural networks, since smoother loss landscapes allow for larger step sizes to be effective. We believe some of our inconsistency between testing and validation error could be due to step size we used, 0.001, being too large for our learning rate.

Additionally, we tried testing out best performing network in a new location to see how it would perform in a different environment. We chose a

drop based in New York City, as we thought the urban environment was very different from that of the terrain in the Yuma Proving Ground. We created a validation set of 25,000 image pair collected from a drop in NYC, consistent in structure to our other datasets used. We found that our best network performed significantly worse when cross-validated on this new terrain, with a RMSE of 79.33 and a percent error of 358%.

In order to compare our network against another model we first tried a different ML algorithm, a random forest. Our random forest consisted of 100 trees, and predicted the change in altitude between grayscaled and resized two images. To train and test this algorithm we used a smaller set of 5,000 image pairs, divided into a training and testing set of 80% and 20%, respectively. This baseline model resulted in a RMSE of 55.74 meters.

8. Future Work

Future teams that work on this project have the opportunity to continue work on the simulator, refine the preprocessing techniques, explore new datasets, and make improvements to the neural network.

8.1 Simulator

One task for future MQP teams is to continue work on our simulator, including improving the User Interface (UI), testing in more diverse environments, adding specialized camera features, and increasing accuracy in location measures.

Our current simulator requires users to customize setting in editor mode of the Unreal Engine. Specifications like flight path and timing of pictures can be made by the user, but there is opportunity to package these inputs in a more user friendly fashion. In addition, the projects could be configured in a headless fashion to allow for faster, easier data collection.

The variety within the environments and their features can also be expanded on in next years' projects. For this year's data collection we used pictures only from midday and in clear weather. We have the capability to precisely change the time of day and add fog effects in the current simulator, but more could be done to mimic real world conditions. This could include adding more diverse weather options, like rain and snow, and improving the nighttime simulator we attempted.

In addition, in real world applications, the military uses a variety of cameras to collect the information they need. Infrared (IR) imaging is often used instead of full spectrum imaging [67]. A great improvement to our simulator would be introducing a virtual version of specialized camera types, especially IR capabilities.

Another potential area of work comes out of a major constraint we faced while working with our simulated data; location values in our simulator only go to the thousands place in decimal specificity. In terms of altitude, which was measured in meter, that was adequate precision, but for latitude and longitude, which are measured in degrees, it was a limited amount of information. The programming of this precision was written into the Unreal Engines blueprint nodes, and required more specialized knowledge of the program to change. A future team could explore the back end programming of UE4 to modify this precision setting and then collect more meaningful latitude and longitude data.

8.2 Preprocessing

Our team explored many preprocessing methods, but ran out of time to try all the methods we would have liked. We expect dimension reduction to drastically improve our model accuracy, and it could be done through methods like PCA, Wavelet and Fourier transforms, and auto encoders. Next years' teams can implement these techniques to test if they have positive effect on the model accuracy and computation speed. This would also reduce the size of the dataset for future teams, and in turn allow them to train with more images and use different neural network architecture that was incompatible with our high dimensional data. They also have the opportunity to try additional methods like, Gaussian blurring [32] or ORB Edge Detection [68], to test their effects in this complex problem.

8.3 Datasets

During our project we focused our problem in on one specific area, the Yuma Proving Grounds, during clear weather, and during midday lighting conditions. Our simulator has created the opportunity to use pictures from all over the world, and add in a variety of conditions that better simulate real world drops. Next years' teams could develop plans for expanding the datasets to include a greater variety of training images for better prediction performance in different conditions. We believe there could be potential to train the model on a subset of diverse terrains, that when combined, could cover the whole feature space of the globe, similar to the concept of eigenfaces in facial recognition field [69].

8.4 Neural Network

Our neural network could also be improved upon by future teams. We researched and tested a series of activation functions and architectures for our model, but there are infinite structures that could be implemented for the problem. There is opportunity to incorporate convolutional layers and test their effects, explore new activation functions, and different sizes and combinations of layers. Future teams could also modify values like batch size and step size to see if they could improve the convergence rate the network.

Based on the non-smooth convergence of our NN that used Leaky ReLU activation function, we believe that a smaller steps size could benefit our model, but at the cost of increased computation time. In turn, increasing the batch size may be one way to decrease the training time.

9. Conclusion

The goal of our project was to aid DEVCOM-SC in development of GAVN technology. In the military, supplies are critical, and a common method for supply deliveries are parafoil parachutes. The accuracy of these supply drops are important to the success and safety of operations and supporting American troops in their missions. While GPS is the most common way to track these parachutes its drawbacks have led DEVCOM-SC to pursue GAVN technology.

Our team took a series of steps to research a machine learning based alternative to GPS navigation for cargo parafoils. First, to achieve the task of obtaining large quantities of high-quality image data, we created a virtual simulator for parachute drops. After collecting data using our simulator, we were able to apply a variety of preprocessing methods to the images and test different neural network structures in an attempt to predict location from the images.

From our work, DEVCOM-SC now has the ability to generate accurate synthetic aerial images from anywhere in the world in different weather conditions and times of day. This is an invaluable aid to neural network development, since it allows for fast, easy, and inexpensive data collection necessary for model training, testing, and cross-validation. We also were able to provide insights on the performance effects of activation functions, network architecture, and preprocessing methods.

The major conclusion our team came to is about the comparison between single- and multi-task learning. From running our experiments, we've found that it could be possible that multi-task neural networks smooth the loss landscape. This conclusion could explain our discrepancy in error between the testing and validation sets.

The availability of unlimited simulated data allowed our team meaningfully apply ML models, and in turn showed additional evidence for the possibility of effective GAVN technology that could outperform traditional GPS signaling. To build on our work, future teams can improve the User Interface, generate data and build models based on more diverse environments, add specialized camera features, and increase accuracy in location measures. For the neural network, there is opportunity to incorporate convolutional layers and test their effects, explore new activation functions, and different sizes and combinations of layers. With the data, code, and performance reports, there is the structure and information to build a better performing neural network and continue research in the area of GAVN as a feasible alternative to GPS.

10. Appendices

10.1 Simulator Manual



BUILDING A PARACHUTE DROP SIMULATOR

ABSTRACT

A tutorial on how to Install Cesium onto an Unreal project based on tutorial available here:

<https://cesium.com/learn/unreal/unreal-quickstart/>

Grace Malabanti, Joe Scheufele, and Juliette Spitaels

Table of Contents

Table of Contents	1
Prerequisites:.....	2
Set Up Unreal Engine and Cesium	2
Step 1: Install the Cesium for Unreal Plugin	2
Step 2: Create the Project and Level.....	3
Step 3: Connect to Cesium.....	9
Step 4: Create a Globe.....	10
Step 5: Add Lighting with CesiumSunSky.....	11
Creating a Flight Path	13
Step 1: Adjust the Unreal Level	13
Step 2: Add the PlaneTrack Class.....	14
Step 3: Bring in Real-World Flight Data	22
Step 4: Add Positions to the Flight Track.....	26
Step 5: Add the Aircraft	32
Step 6: Adding a Camera	40
Environment Customizations	41
Time of Day:.....	41
Fog:.....	42
Handling the Data	42
Manual Data Collection	42
Automatic Data Collection	46
Data Organization	55
Alternate Drops	56
User Guide	63
Open and Load Files	63
Flying and Collecting Data.....	65
Accessing Data	65
Recording a Drop Video	66
Troubleshooting.....	72
Cesium World Disappears	72
Frequently Asked Questions.....	77

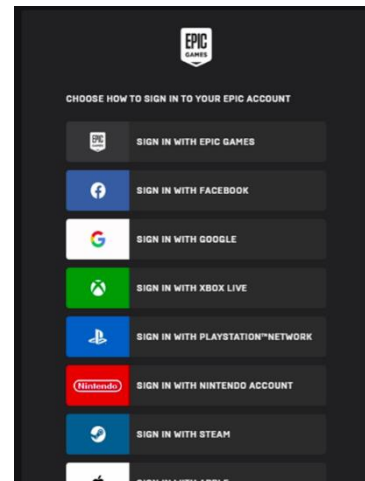
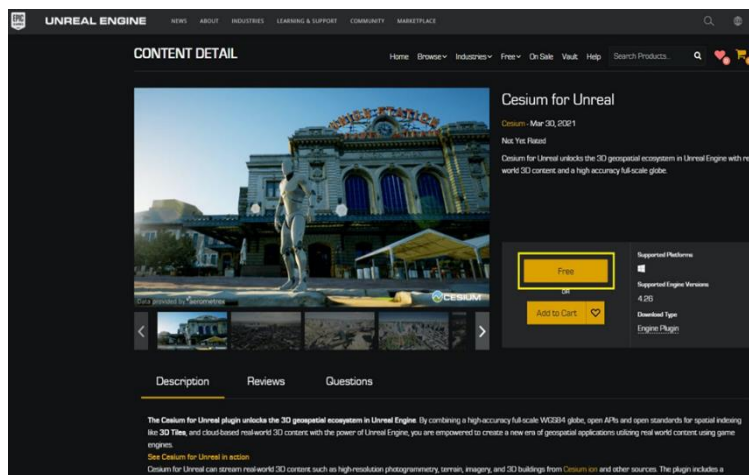
Prerequisites:

- An installed version of Unreal Engine (at least 4.26 or later). For instructions on how to install Unreal Engine, visit the Unreal Engine download page and refer to the Installing Unreal Engine guide for detailed instructions: [Installing Unreal Engine | Unreal Engine Documentation](#)
- A Cesium ion account to stream terrain and building assets into Unreal Engine. Sign up for a free Cesium ion account if you don't already have one: [Sign Up](#)
- A recent version of Visual Studio, which is a licensed software. Information about downloading this for Unreal Engine here: [Setting Up Visual Studio for Unreal Engine](#)

Set Up Unreal Engine and Cesium

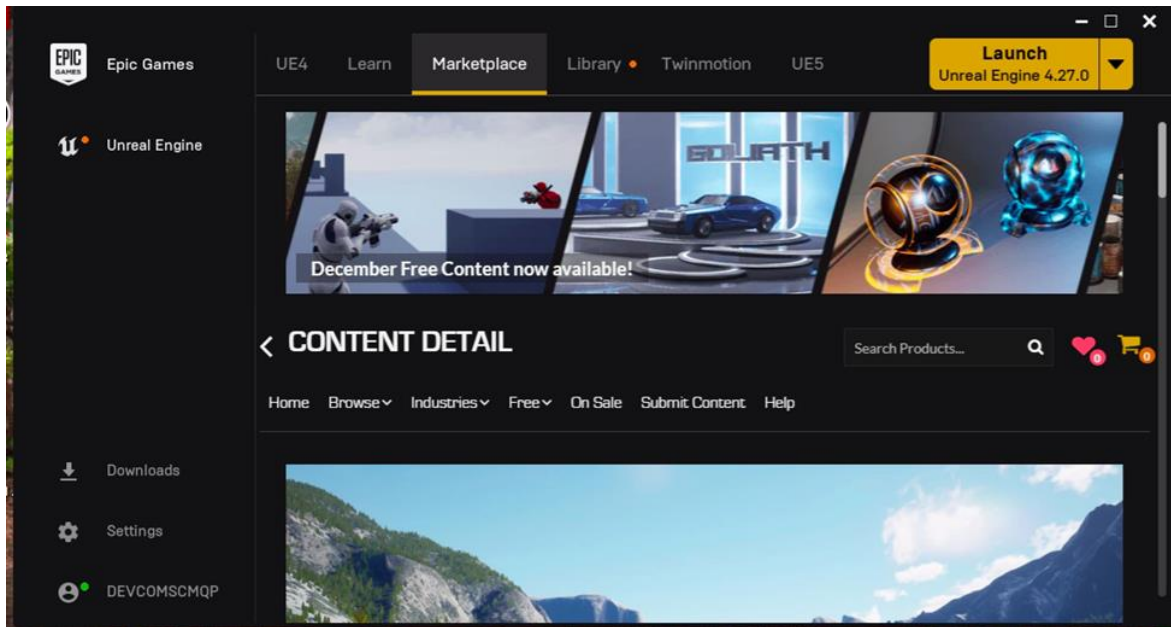
Step 1: Install the Cesium for Unreal Plugin

Open the Cesium for Unreal plugin page on the Unreal Engine Marketplace: [Cesium for Unreal in Code Plugins - UE Marketplace](#). Sign in if needed and click on the **Free** button to install the plugin on your Unreal Engine account. If authorized, use the DEVCOMSCMQP@gmail.com account to sign in with Google.

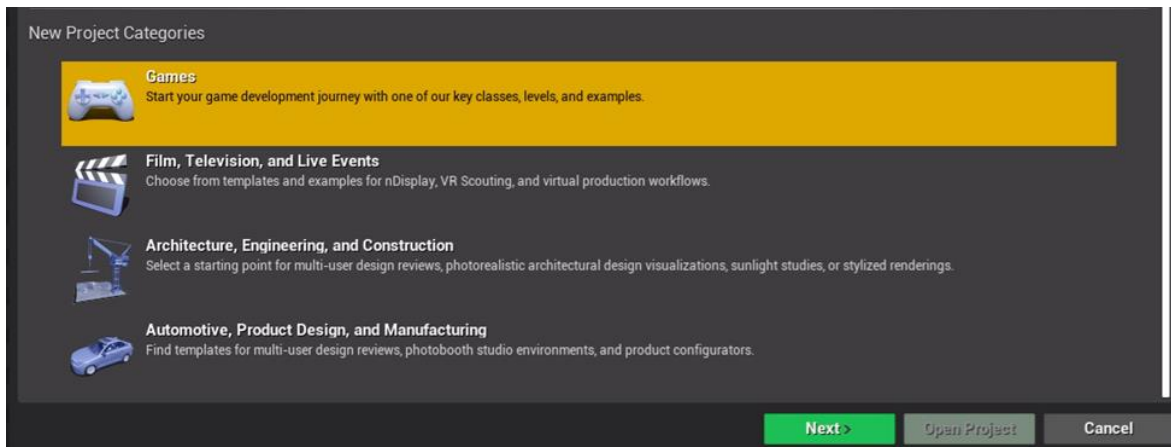


Step 2: Create the Project and Level

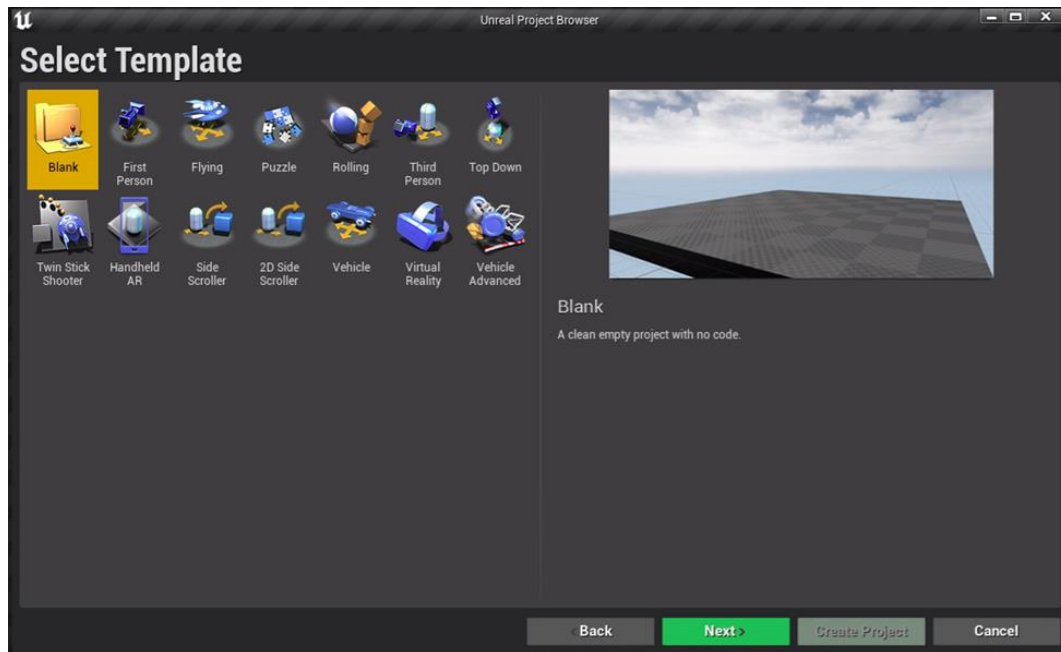
Click the **Launch** button in Unreal Engine to create a new project.



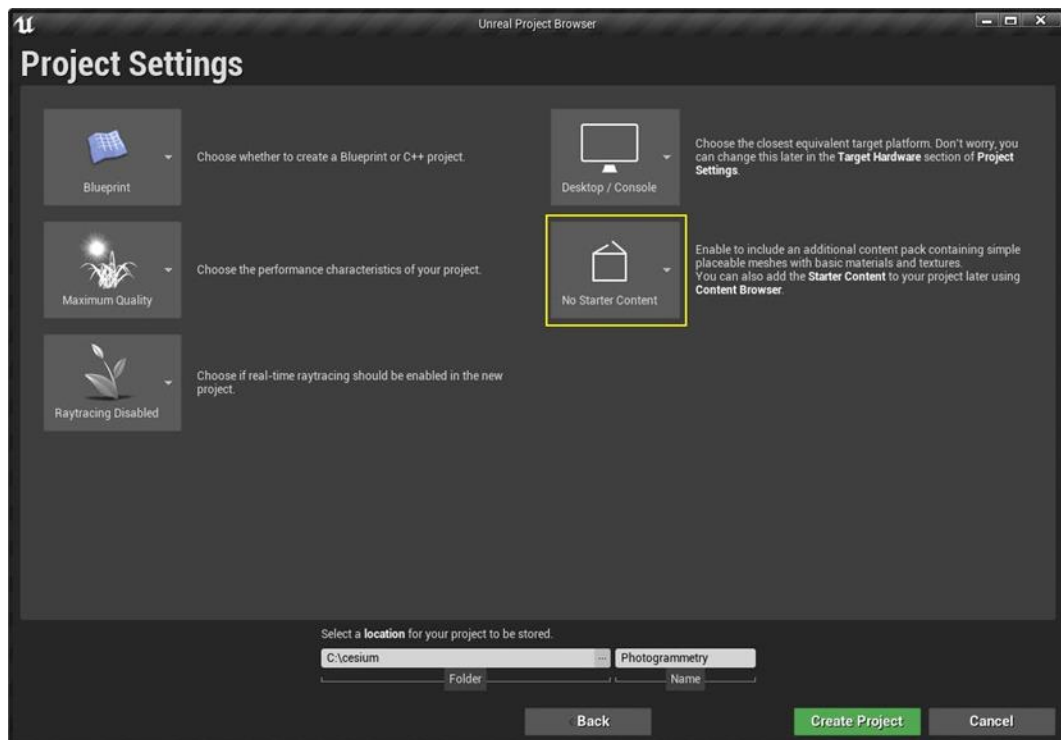
Select **Game** as the New Project Category and click the green **Next** button.



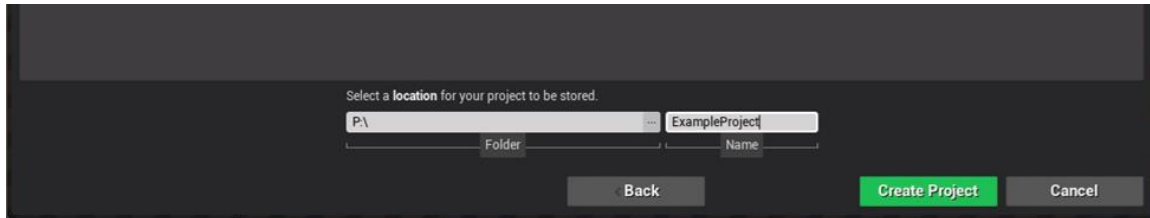
Now choose **Blank** as the Template. Again, click the green **Next** button.



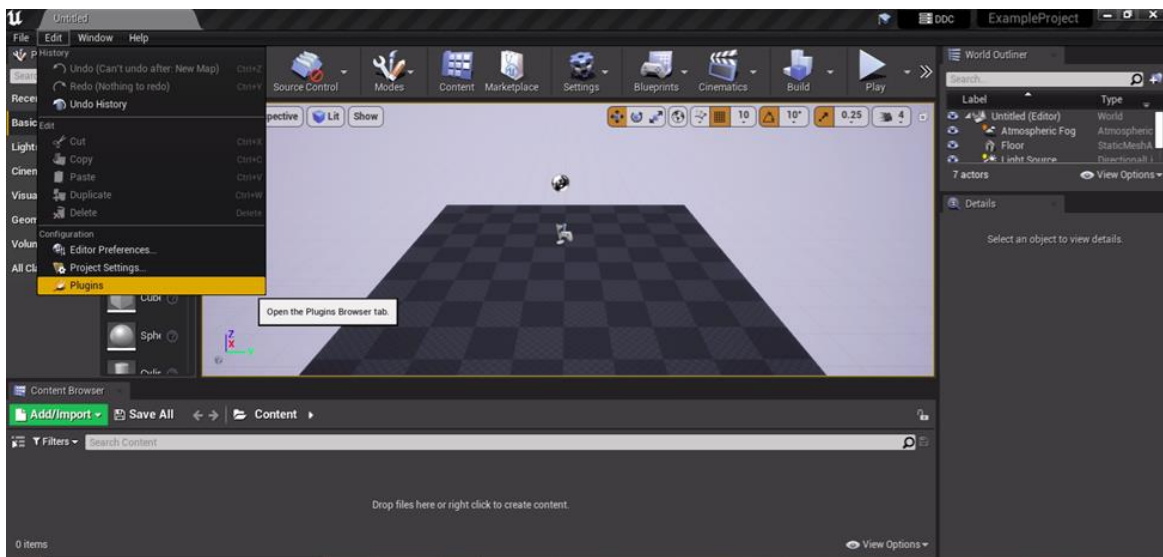
Choose **No Starter Content** to avoid cluttering the level.



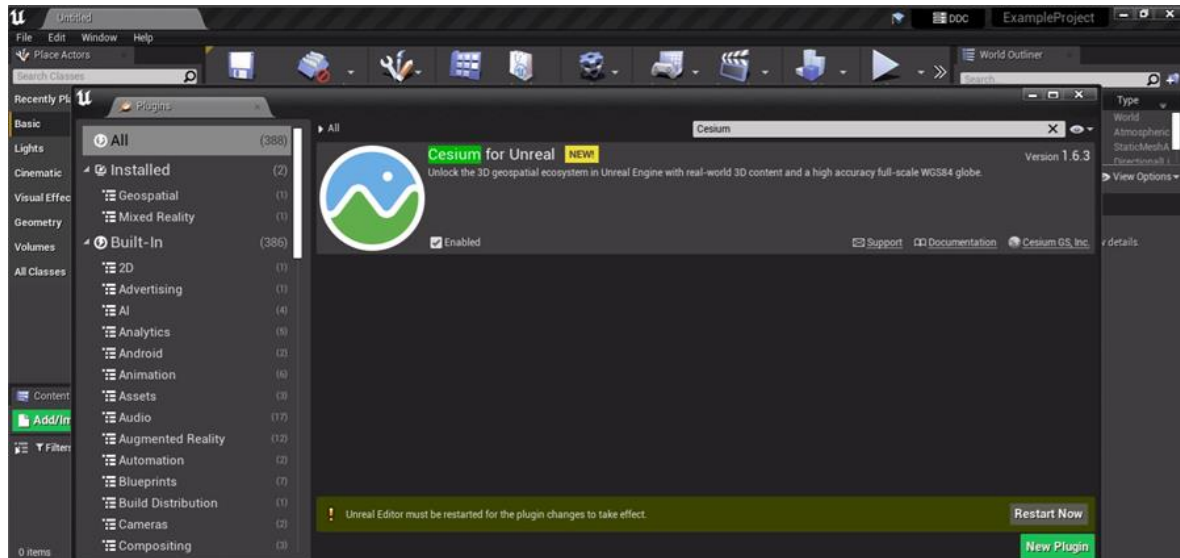
Name your project in the field at the bottom of the window. Here, you will also select the location where your file will save. Note, these files are about 4 GB, so select a location that will accommodate that size. Additionally, if using the remote desktop, ensure you are saving to a permanent location, like the P drive (pictured below). Then click the green **Create Project** button.



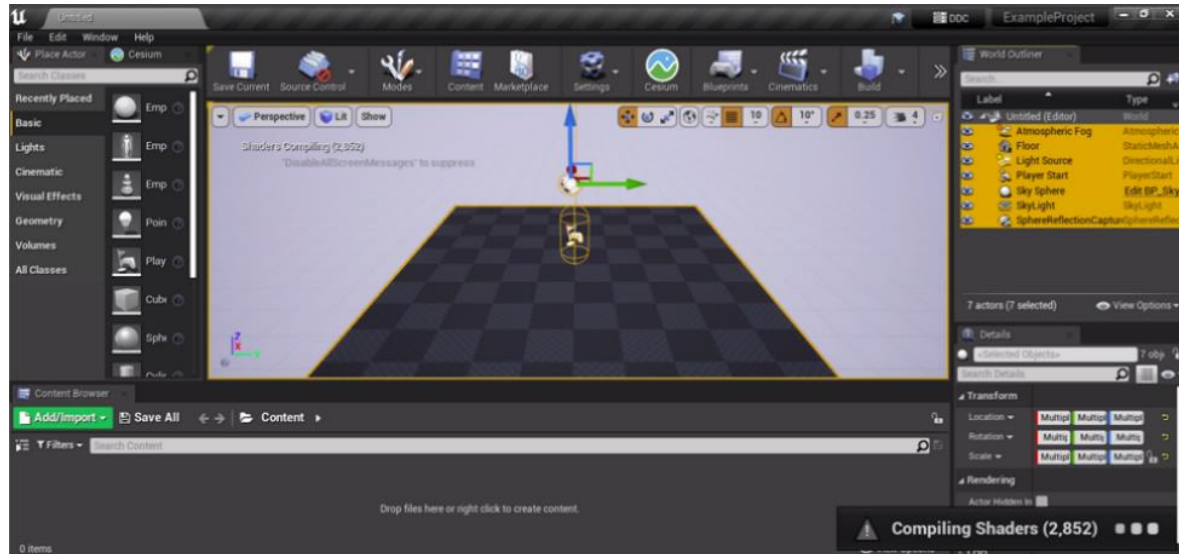
Next you will activate the Cesium for Unreal plugin. Go to Edit → Plugins in the top left of the editor window.



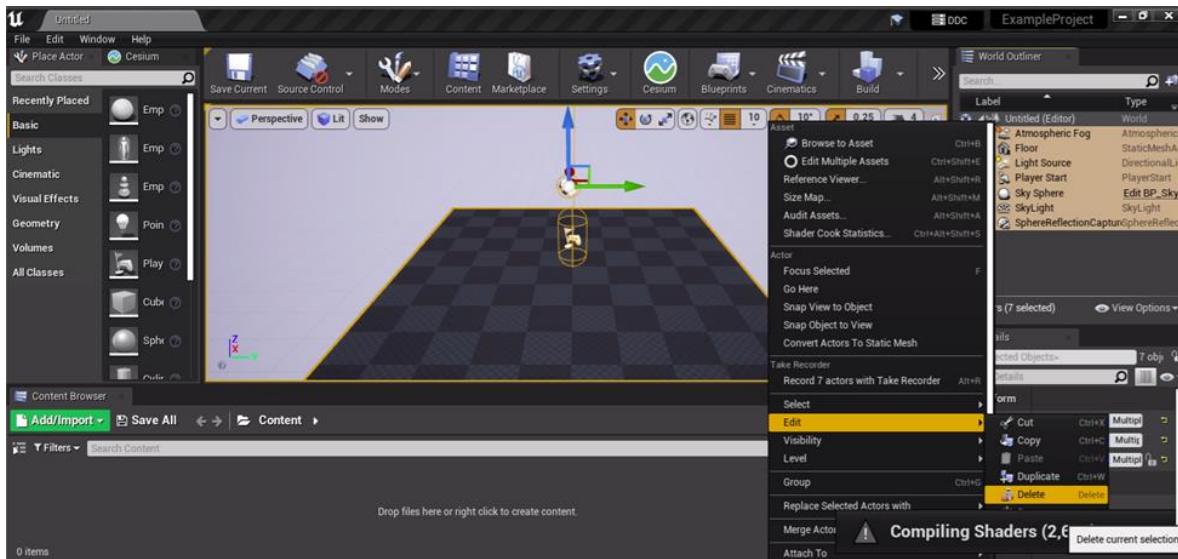
In the new Plugins window, search for “Cesium” in the search bar at the top-right corner. Click the **Enabled** checkbox for the plugin, so it is checked. A yellow banner will likely appear at the bottom, asking you to restart the engine. Click the **Restart Now** button. Your project will automatically reopen, and you can now exit out of the plug-ins window



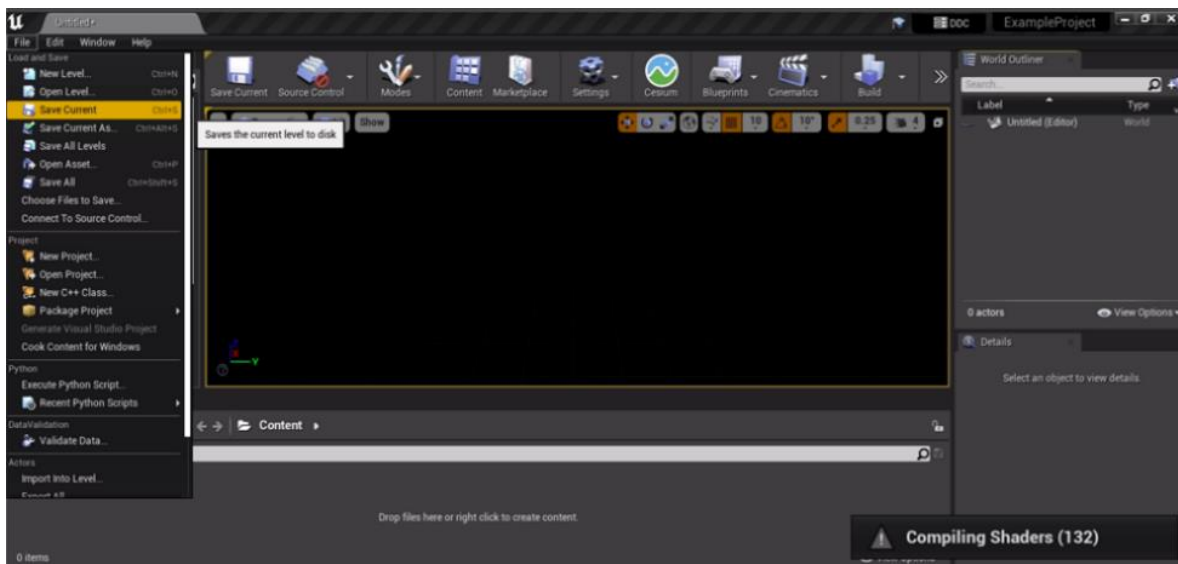
The World Outliner on the right side of the editor lists all the objects you have in your scene. Select all of them except the **Untitled (Editor)** by using the shift-click command.



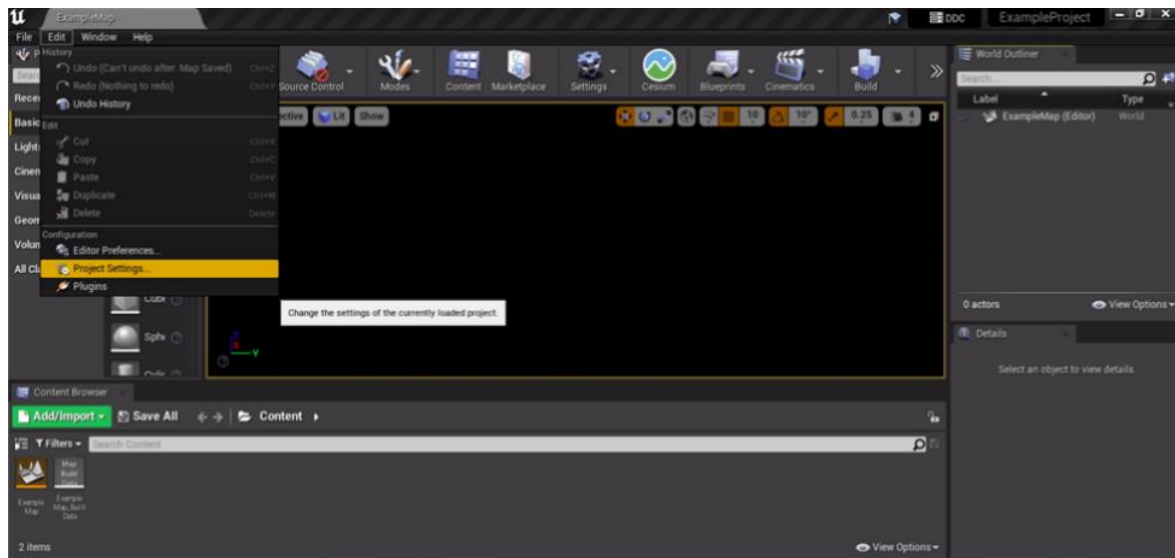
Right click on your selected objects → Edit → Delete, to remove all the objects.



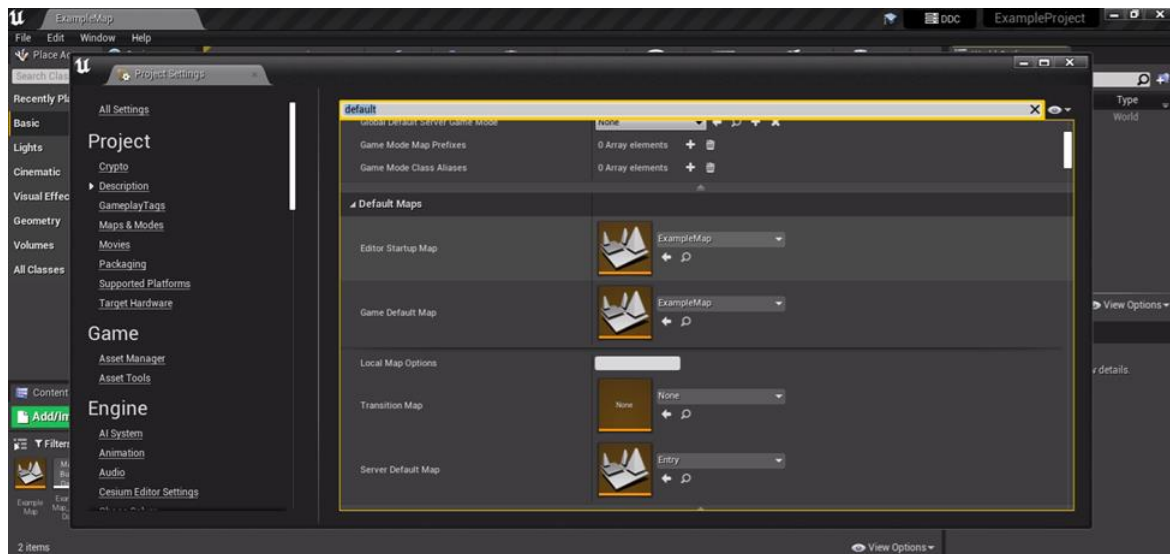
A message will pop-up warning about dependencies. Click **Yes All** to clear the message and delete the objects. You should now have a completely black viewport in the editor. Now you want to save your level by first clicking File → Save Current in the top left of the editor window.



Name your level in the bottom bar of the popup window, here we used the name “ExampleMap.” Theme click the grey **Save** button. This will save a file inside of your project. Go to Edit → Project Settings in the top left of the editor.

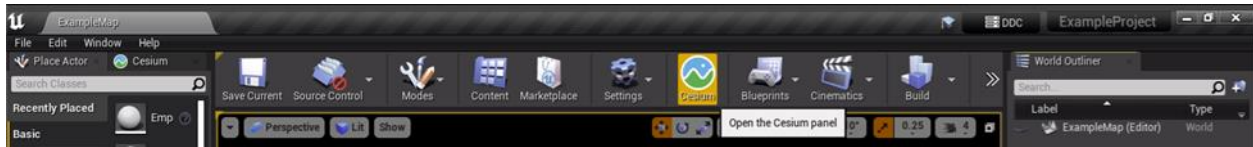


In the new popup window, search for “default.” Set the level you saved above as the Editor Startup Map and the Game Default Map. This ensures your level will be re-opened automatically when Unreal Editor is restarted. Once set, close the Project Settings window, it will automatically save your changes.

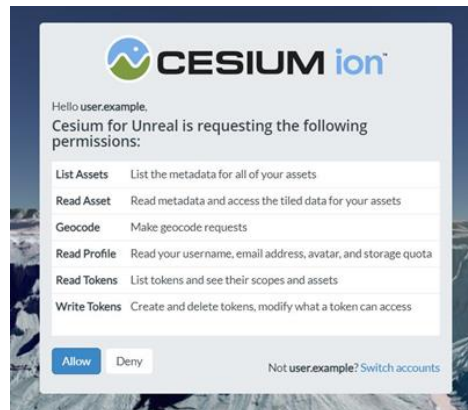


Step 3: Connect to Cesium

Open the Cesium panel by clicking the icon in the toolbar.



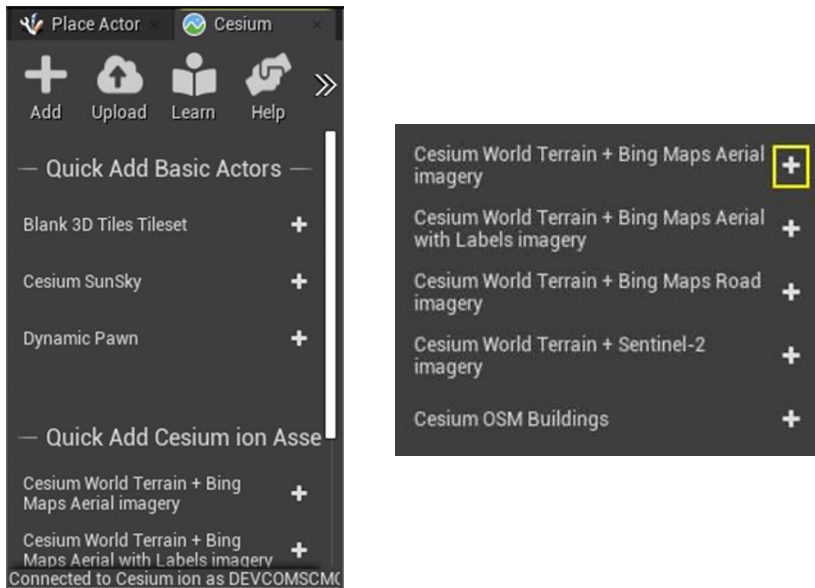
The Cesium panel will show up on the left side of the editor window. Connect to Cesium ion with the Connect button. A pop-up browser will show, asking you to allow Cesium for Unreal to access your assets with the currently logged-in account from Cesium ion. Select **Allow**. You may be asked to login again. If so, select the first option, **Sign in with Epic Games**.



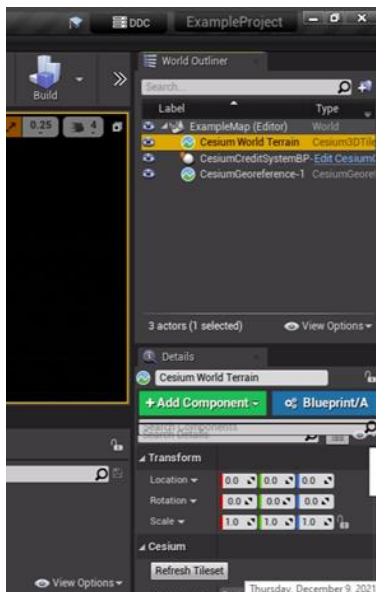
Now head back to Unreal Engine to continue with the next steps.

Step 4: Create a Globe

In this section, you will begin to populate the scene with assets from Cesium ion. Now that you've connected to Cesium, the window will look different, displaying many options for editing. From the Quick Add Cesium Ion Assets section at the bottom of the window, click the plus button next to **Cesium World Terrain + Bing Maps Aerial imagery**.

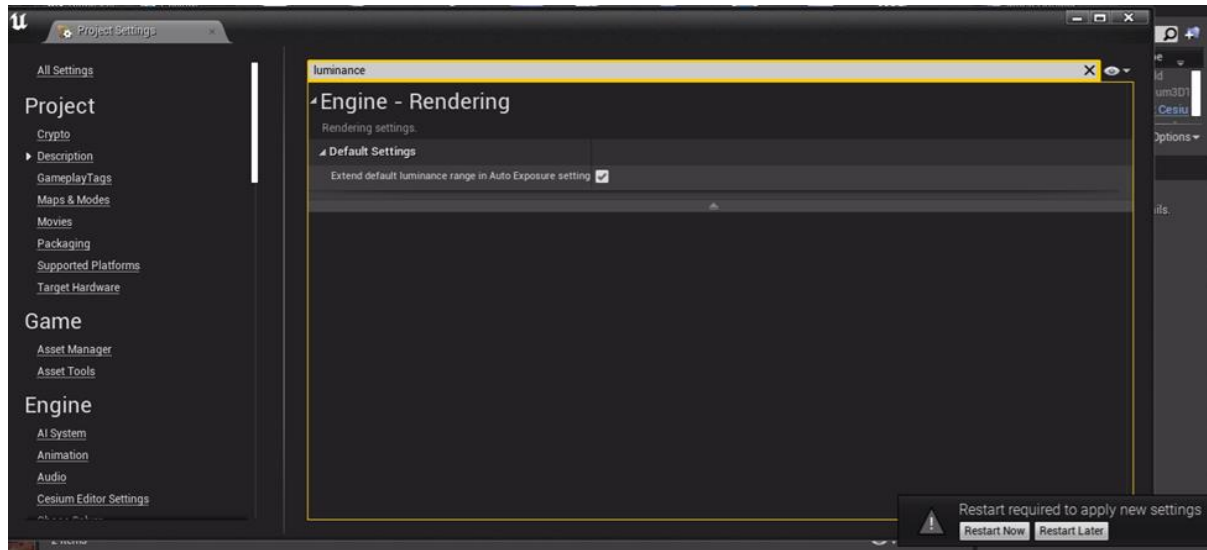


This step will generate new Cesium World Terrain and CesiumGeoreference actors in the World Outliner, on the left. You should also see a white sphere with arrows in your editing viewport.



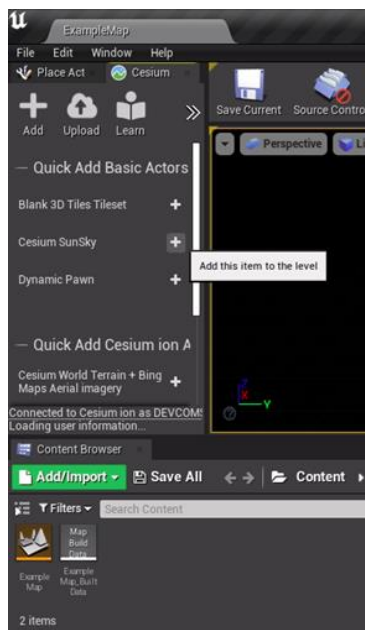
Step 5: Add Lighting with CesiumSunSky

The CesiumSunSky Actor controls the sun in a project. To begin, go to Edit → Project Settings. In the pop-up, search for “luminance.” Make sure the option “Extend default luminance range in Auto Exposure settings” is enabled.

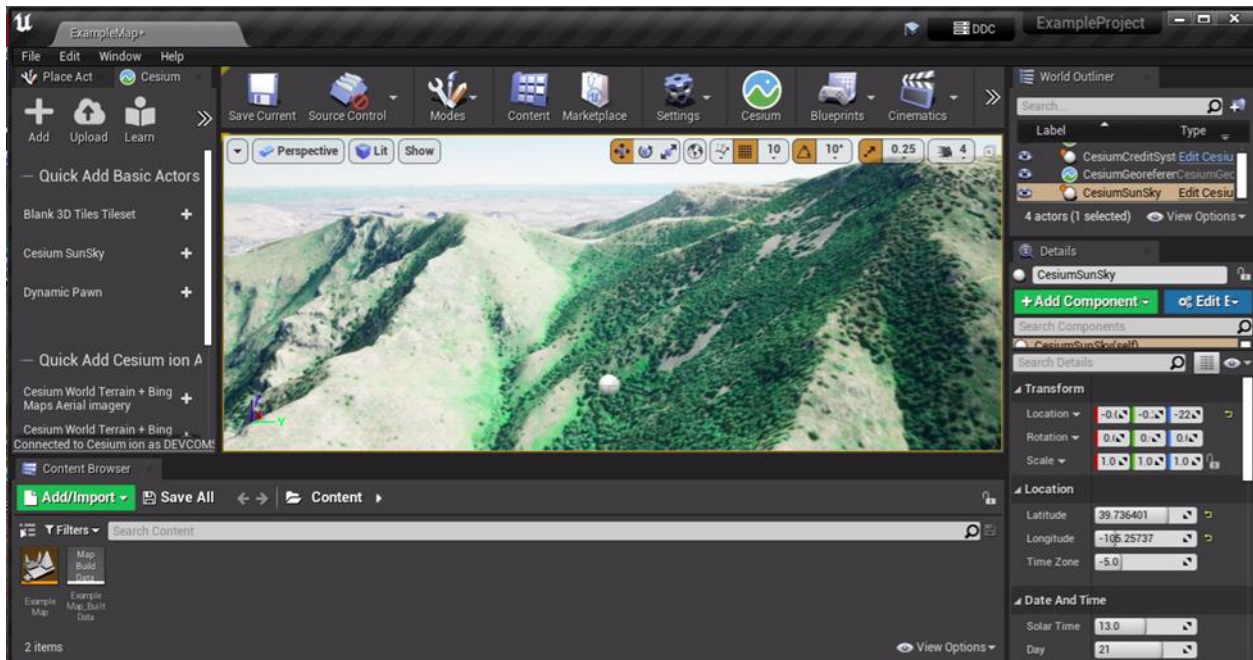


Restart Unreal Engine after enabling this option, as indicated by the popup in the corner.

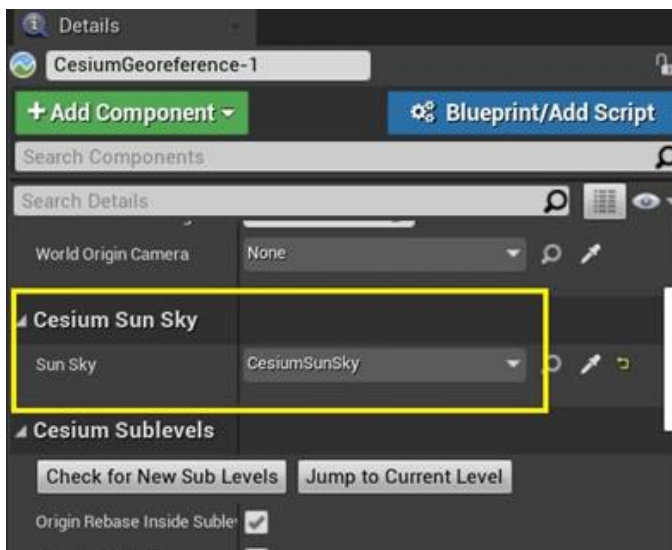
Return to the Cesium panel on the left. Add a CesiumSunSky actor to the scene by clicking the plus button.



The scene may appear white for a moment as the auto exposure adjusts. When it has settled, you should see a sky and atmosphere lighting the terrain.



Note: There may be a black strip near the horizon. To fix this, select the CesiumGeoreference actor, look for SunSky in the Details panel, and ensure it is set to your new CesiumSunSky actor.



Creating a Flight Path

(Based on tutorial available here: <https://cesium.com/learn/unreal/unreal-flight-tracker/>)

In this section we will import a flight path and create a spline for an object and camera to fly along. For this tutorial, we will recreate a parachute test drop in Arizona.

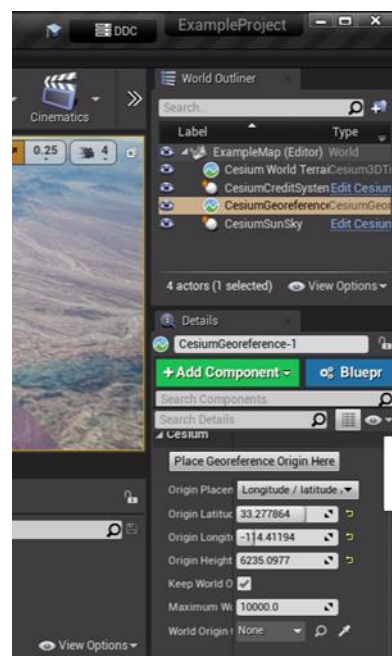
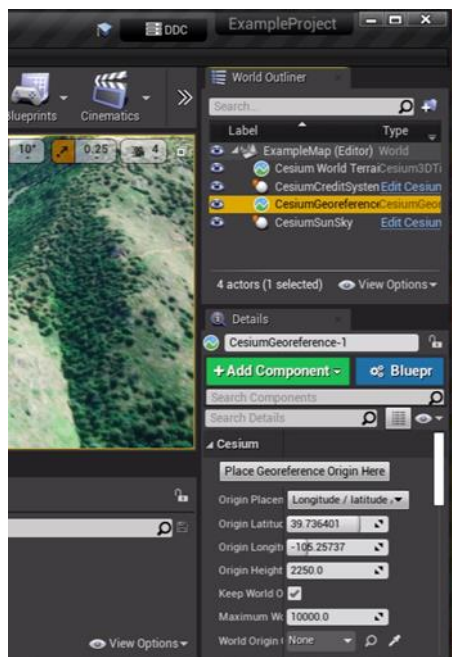
Step 1: Adjust the Unreal Level

To position your view to the parachute test site. In the World Outliner on the left, select the CesiumGeoreference object. And you want to change your location in this tab to be:

Origin Latitude = 33.277864

Origin Longitude = -114.41194

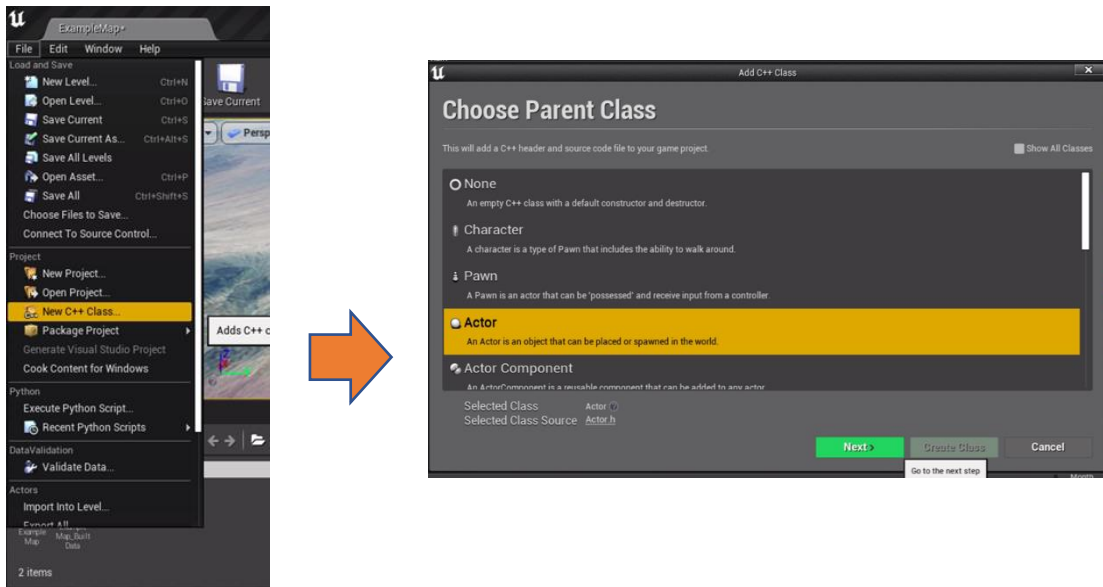
Origin Height = 6235.0977



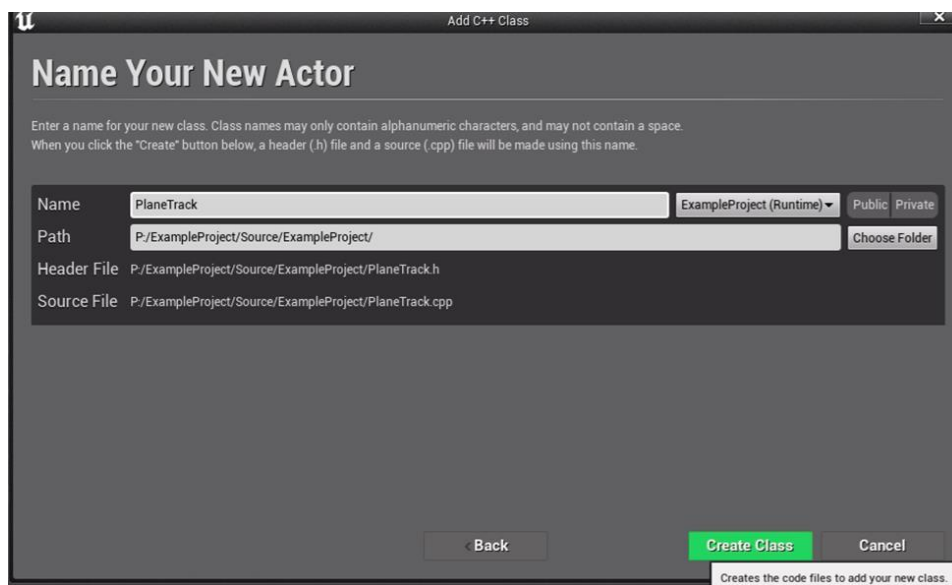
Step 2: Add the PlaneTrack Class

The PlaneTrack class will contain logic to process the drop data and generate position points for the spline which represents the parachute's path.

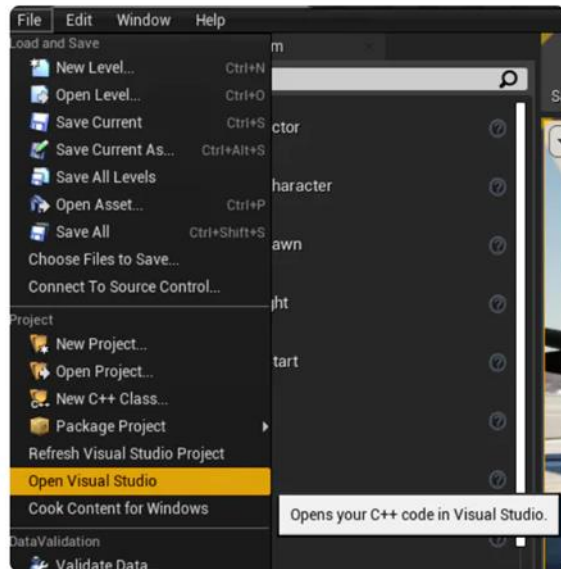
Add a new C++ class by going to File → New C++ Class... at the top left corner of the Unreal Editor. Select Actor as the parent class. And click the green **Next** button.



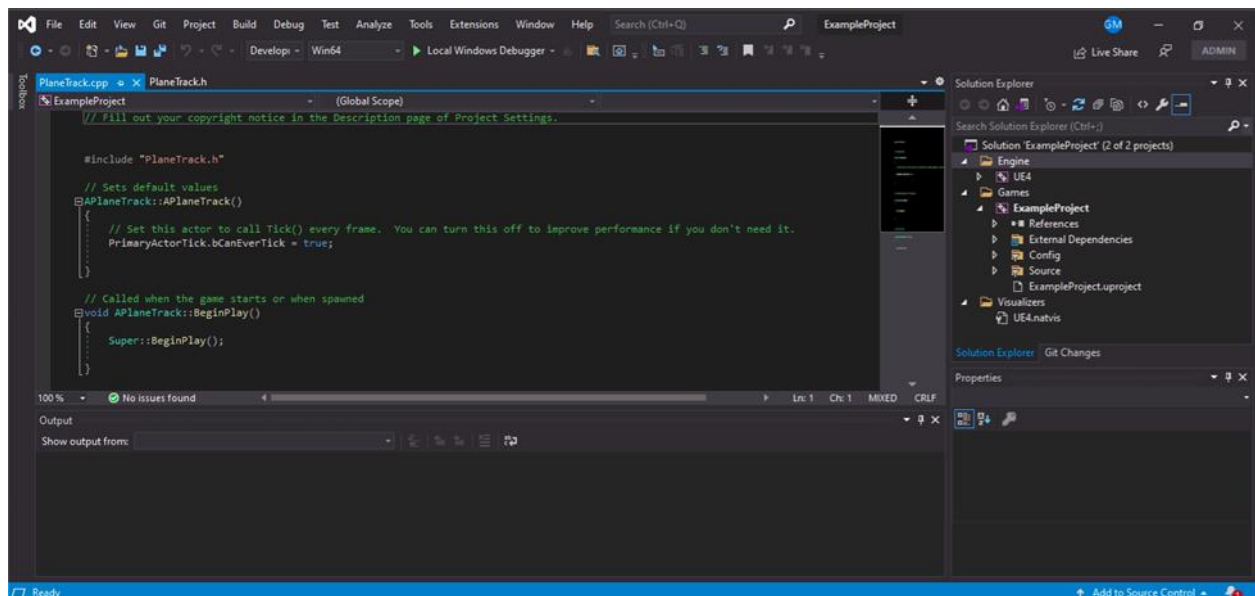
On the following page, set the name of the new class to "PlaneTrack". Then click the green **Create Class** button. Visual Studio should automatically open the file.



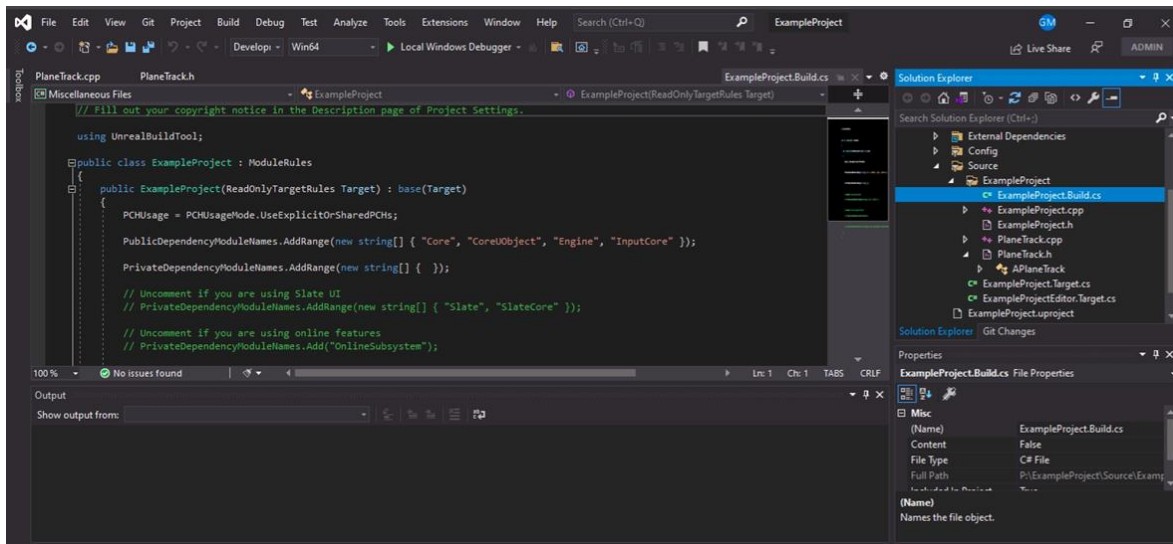
If Visual Studio does not open, navigate to File → Open Visual Studio. You will need to log in using your licensed account.



Visual Studio will open with associated Unreal Engine code.



On the right Panel, click Games → Source → YourProjectName there will be a file with the file type '.Build.cs', named according to your project. Click that file to open the code.



Add the following code snippet to the project, below the line:

```
"PrivateDependencyModuleNames.AddRange(new string[] { });";
```

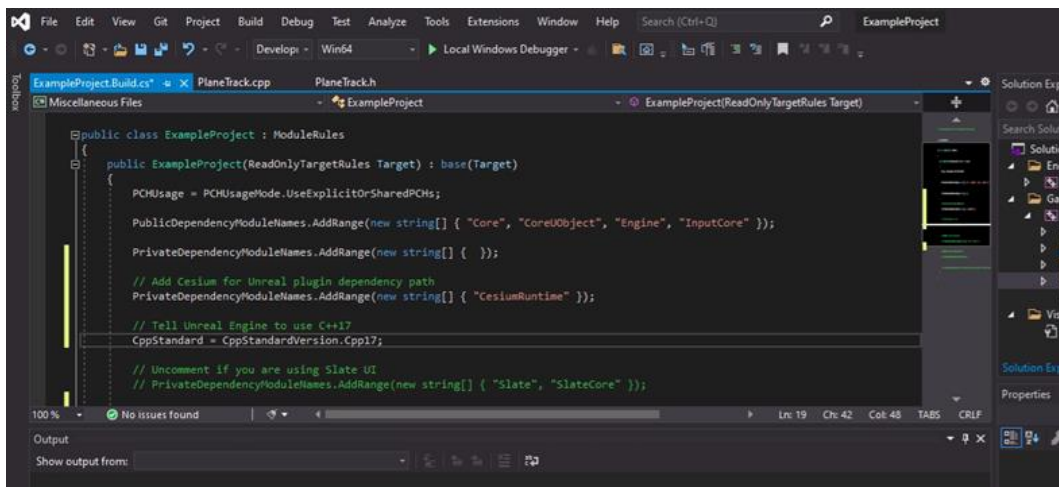
```
// Add Cesium for Unreal plugin dependency path
```

```
PrivateDependencyModuleNames.AddRange(new string[] { "CesiumRuntime" });
```

```
// Tell Unreal Engine to use C++17
```

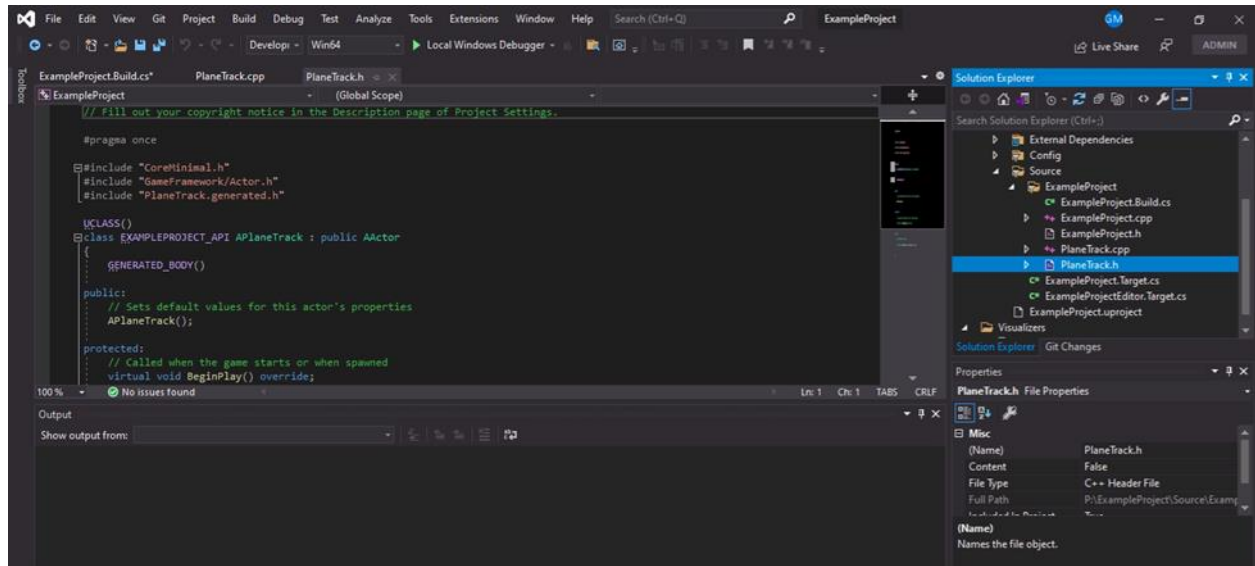
```
CppStandard = CppStandardVersion.Cpp17;
```

After you paste in the code, it should look like this:



Next you need to add some member variables to the PlaneTrack class to store the drop data, spline, and convert the data to the appropriate coordinate system.

Now open a new .h file, PlaneTrack.h, located in the source folder.



At the top of the file, import the necessary libraries by copying and pasting in the code below:

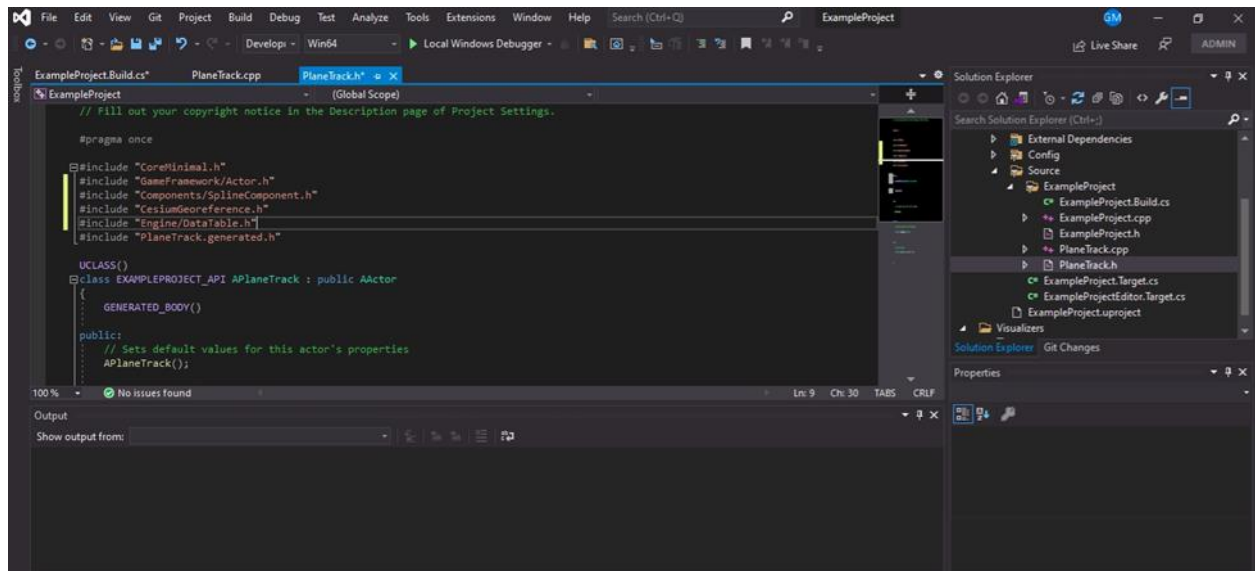
```
// Add import paths. Make sure they go above the PlaneTrack.generated.h line

#include "Components/SplineComponent.h"

#include "CesiumGeoreference.h"

#include "Engine/DataTable.h"
```

Once finished it should look like this:



You will also need to add the below code snippet within the class under APlaneTrack();

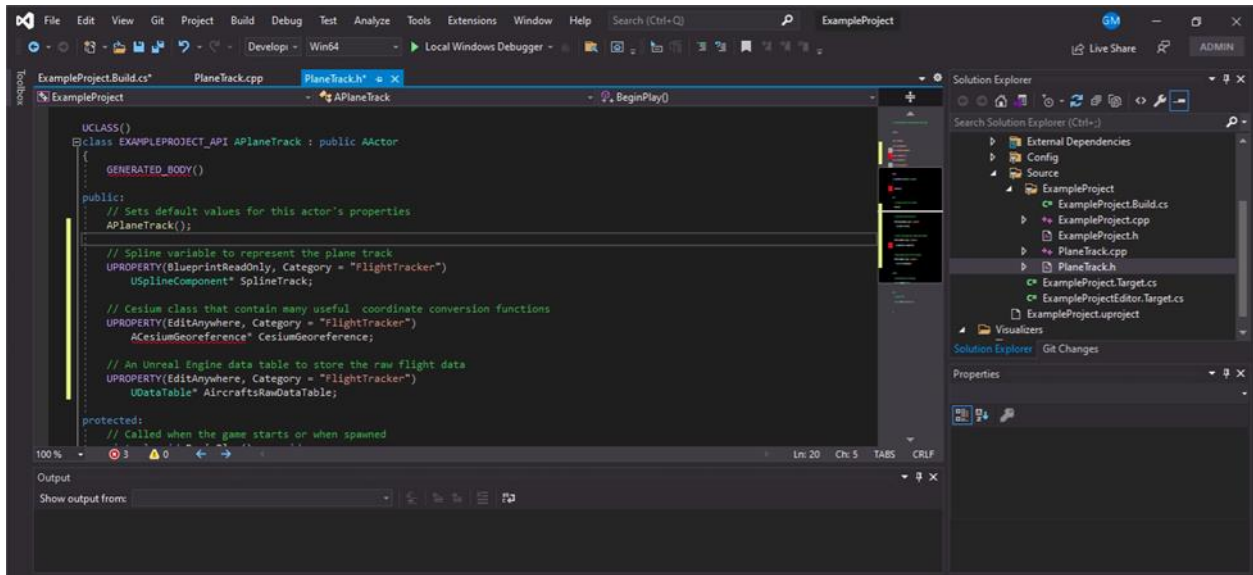
```
public:

    // Spline variable to represent the plane track
    UPROPERTY(BlueprintReadOnly, Category = "FlightTracker")
    USplineComponent* SplineTrack;

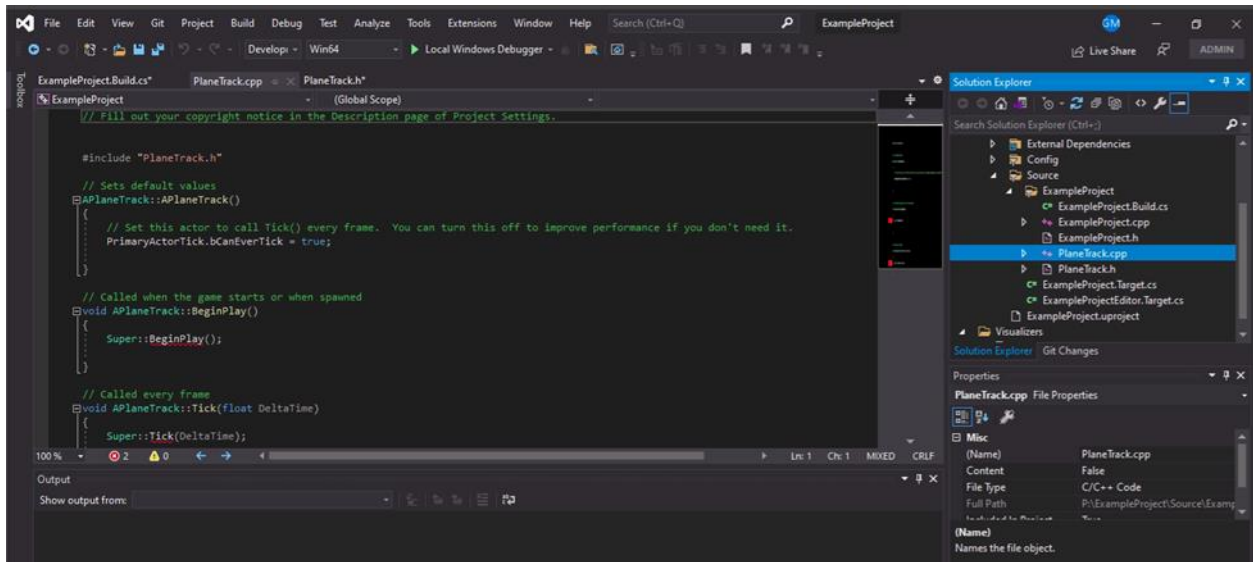
    // Cesium class that contain many useful coordinate conversion functions
    UPROPERTY(EditAnywhere, Category = "FlightTracker")
    ACesiumGeoreference* CesiumGeoreference;

    // An Unreal Engine data table to store the raw flight data
    UPROPERTY(EditAnywhere, Category = "FlightTracker")
    UDataTable* AircraftsRawDataTable;
```

Once completed your code should look like this:



Navigate to and open the file `PlaneTrack.h`. It should initially look like the image below. Navigate to the `APlaneTrack::APlaneTrack()` line in the code. This is known as a constructor, and we will modify it in the next step.



Change the contents of the APlaneTrack constructor by replacing the existing code with the code below:

```
APlaneTrack::APlaneTrack()

{

    // Set this actor to call Tick() every frame. You can turn this off to //improve performance if
    you don't need it.

    PrimaryActorTick.bCanEverTick = true;


    // Initialize the track

    SplineTrack = CreateDefaultSubobject<USplineComponent>(TEXT("SplineTrack"));

    // This lets us visualize the spline in Play mode

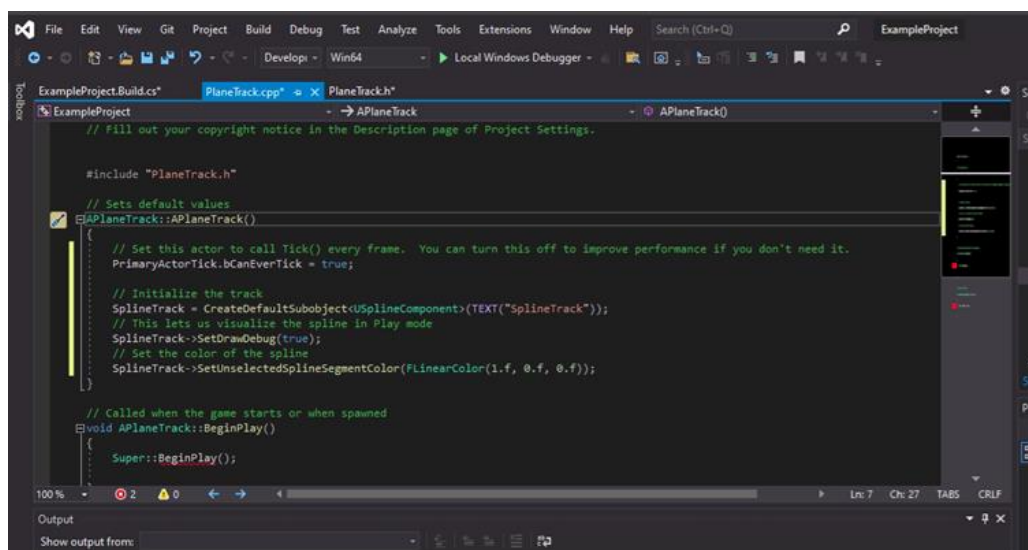
    SplineTrack->SetDrawDebug(true);

    // Set the color of the spline

    SplineTrack->SetUnselectedSplineSegmentColor(FLinearColor(1.f, 0.f, 0.f));

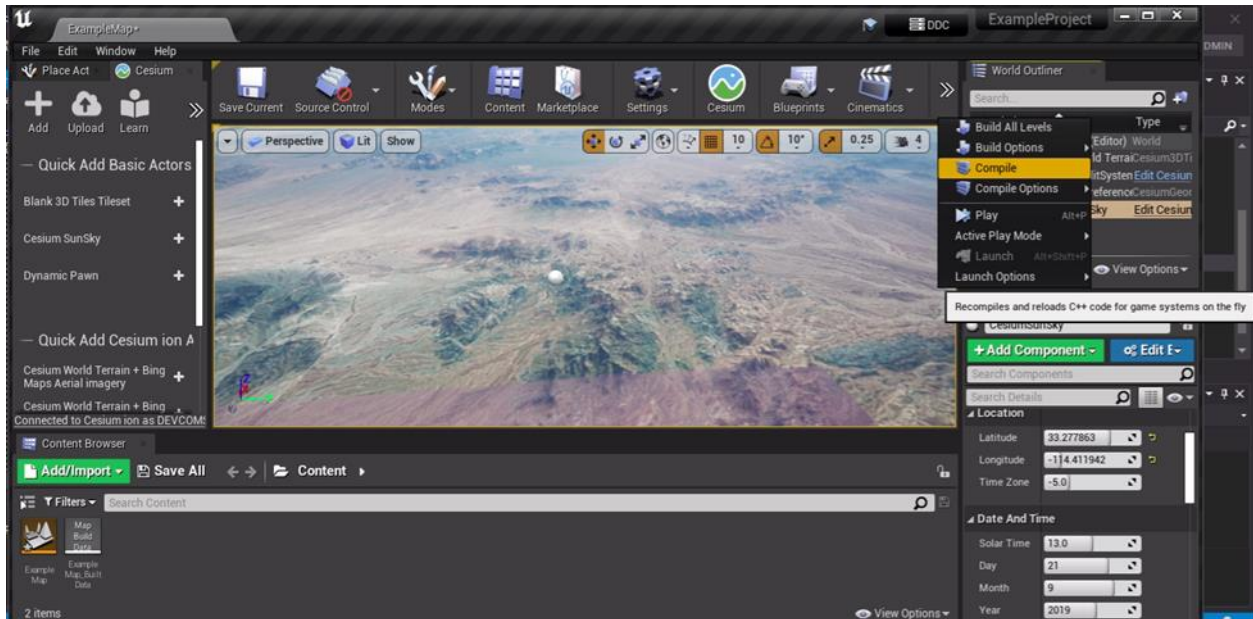
}
```

Once you've made this change your file should look like this:



Now save each of the C++ files you edited (there should be three) in the Visual Studio Editor. Do this by clicking the blue save button in the top left and saving to the default location.

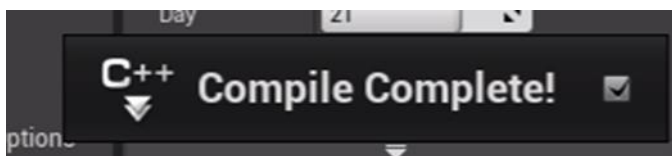
After you save your files, return to the Unreal Engine Editor, and click the **Compile** button at the top tool panel.



Depending on screen size, you may see the icon in the toolbar here:



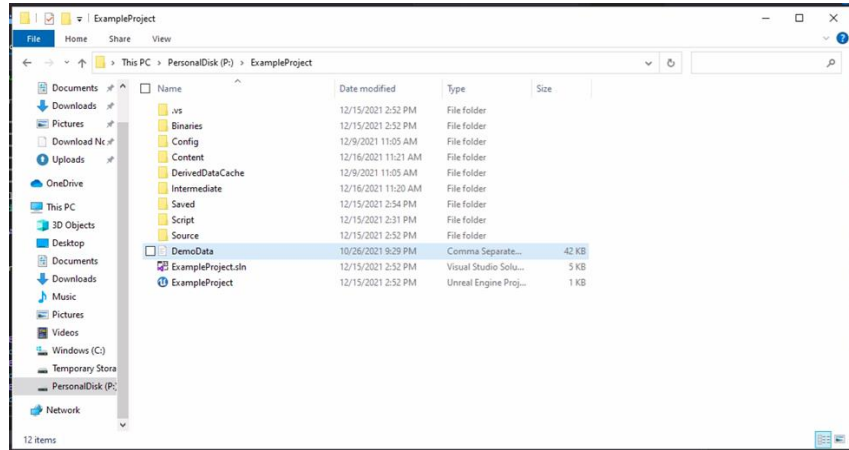
If the code is well-formatted and correct, you will see the “Compile Complete” message at the bottom right corner of the Unreal Engine main editor. In our instructions, compiling the code refers to this step.



Step 3: Bring in Real-World Flight Data

Next, you are going to use parachute test-drop data for this demo. The height in Cesium for Unreal is in meters relative to the WGS84 ellipsoid. The data is provided in the required format, and available on the shared drive and here: [DemoData.csv](#). Then download the data and drop a copy of the CSV file into your project folder.

	A	B	C	D
1	id	latitude	longitude	height
2	1	33.27786	-114.412	6235.098
3	2	33.278	-114.412	6202.197
4	3	33.27805	-114.412	6191.097
5	4	33.27811	-114.412	6180.897
6	5	33.27817	-114.412	6169.897
7	6	33.27823	-114.412	6160.198
8	7	33.27829	-114.412	6152.398
9	8	33.27837	-114.412	6145.198
10	9	33.27848	-114.412	6138.398
11	10	33.27859	-114.412	6131.499
12	11	33.27871	-114.412	6124.899
13	12	33.27884	-114.412	6118.099
14	13	33.27896	-114.412	6111.799
15	14	33.2791	-114.412	6105.399
16	15	33.27922	-114.412	6098.999
17	16	33.27935	-114.412	6092.9
18	17	33.27947	-114.412	6086.7
19	18	33.27959	-114.412	6080.6
20	19	33.27971	-114.412	6074.3
21	20	33.27983	-114.412	6068.1
22	21	33.27995	-114.412	6061.901
23	22	33.28008	-114.412	6055.401
24	23	33.28022	-114.412	6049.101
25	24	33.28036	-114.412	6042.801
26	25	33.28051	-114.412	6036.501
27	26	33.28065	-114.412	6030.401
28	27	33.28079	-114.412	6024.101
29	28	33.28094	-114.412	6018.101



For the PlaneTrack class to access the data to perform coordinate conversions, you will use the Unreal Engine DataTable to store the data inside of the project. In this step, you will create a data structure to represent the structure of the drop data.

In the PlaneTrack.h file, insert this snippet of code directly below the imports to define the flight database structure.

```
USTRUCT(BlueprintType)

struct FAircraftRawData : public FTableRowBase

{

    GENERATED_USTRUCT_BODY()

    public:

        FAircraftRawData()

            : Latitude(0.0)

            , Longitude(0.0)

            , Height(0.0)

        {}

        UPROPERTY(EditAnywhere, Category = "FlightTracker")

            double Latitude;

        UPROPERTY(EditAnywhere, Category = "FlightTracker")

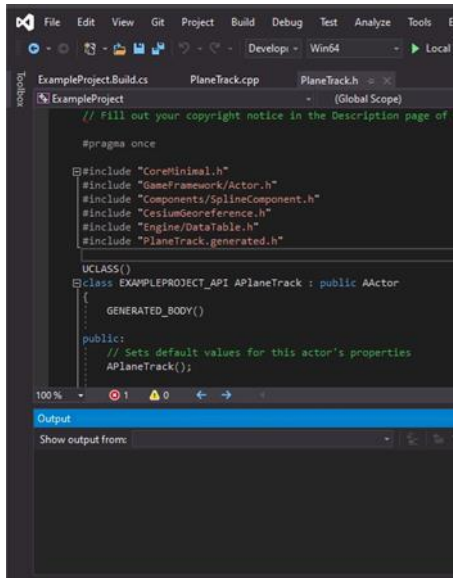
            double Longitude;

        UPROPERTY(EditAnywhere, Category = "FlightTracker")

            double Height;

};
```

Before you add the code, it should look like:



```
// Fill out your copyright notice in the Description page of Project Settings

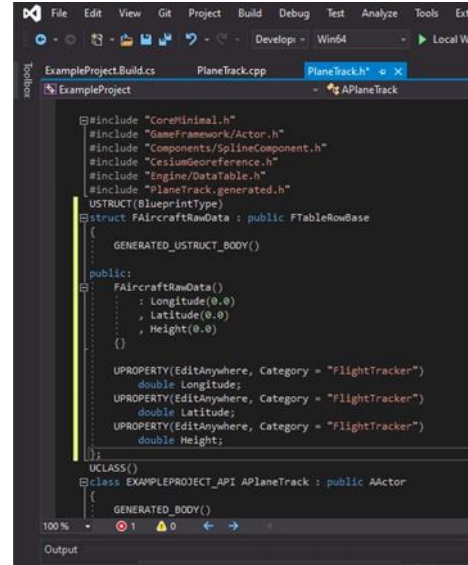
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "Components/SplineComponent.h"
#include "CesiumGeoreference.h"
#include "Engine/DataTable.h"
#include "PlaneTrack.generated.h"

UCLASS()
class EXAMPLEPROJECT_API APlaneTrack : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    APlaneTrack();
}
```

After you add the code it should look like:



```
#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "Components/SplineComponent.h"
#include "CesiumGeoreference.h"
#include "Engine/DataTable.h"
#include "PlaneTrack.generated.h"

USTRUCT(BlueprintType)
struct FAircraftRawData : public FTableRowBase
{
    GENERATED_USTRUCT_BODY()

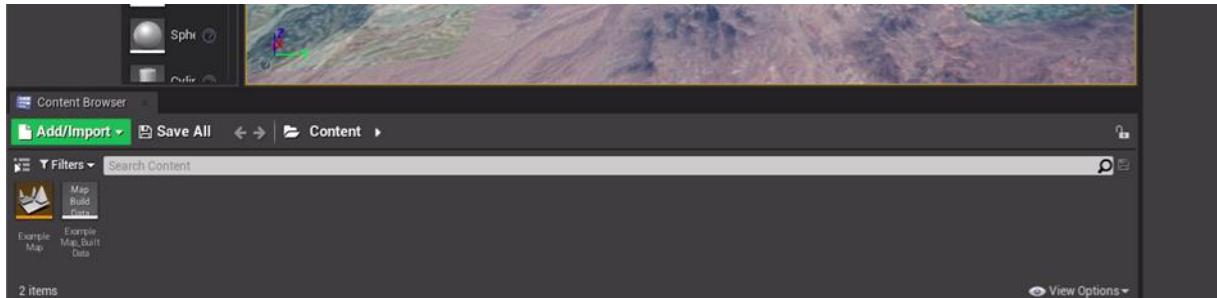
public:
    FAircraftRawData()
        : Longitude(0.0)
        , Latitude(0.0)
        , Height(0.0)
    {}

    UPROPERTY(EditAnywhere, Category = "FlightTracker")
    double Longitude;
    UPROPERTY(EditAnywhere, Category = "FlightTracker")
    double Latitude;
    UPROPERTY(EditAnywhere, Category = "FlightTracker")
    double Height;
};

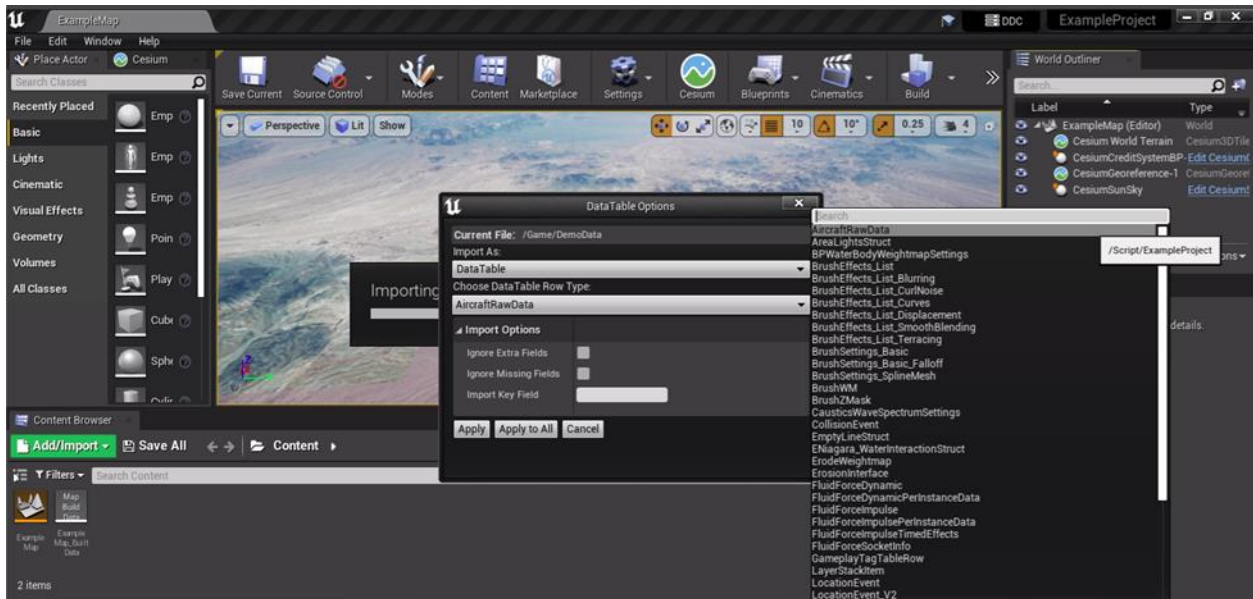
UCLASS()
class EXAMPLEPROJECT_API APlaneTrack : public AActor
{
    GENERATED_BODY()
}
```

Next, save and compile the code. Remember, if the code is well-formatted and correct, you will see the “Compile Complete” message at the bottom right corner of the Unreal Engine main editor.

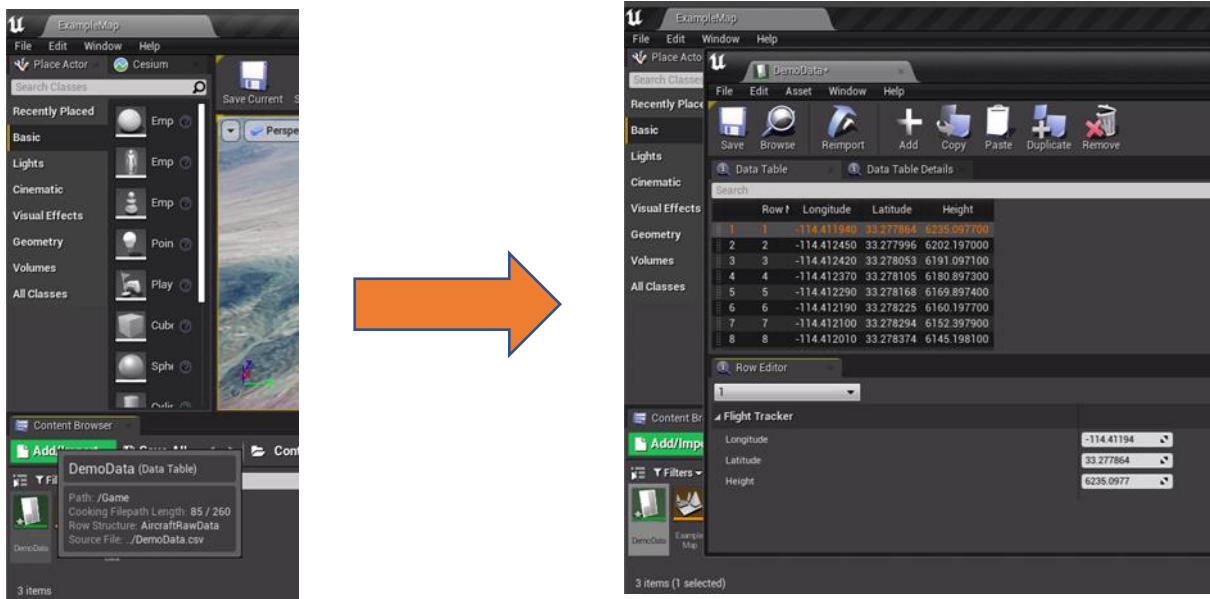
Next you will drag-and-drop the .csv data file into the Unreal Engine Content Browser. This is the panel at the bottom of your editor.



After you drop in the file, a pop-up window will appear. In the **Choose DataTable Row Type** dropdown select **AircraftRawData** (typically the first option):



Click **Apply**. Then to check the data uploaded correctly, double-click on the file in the Content Browser to open the data table:



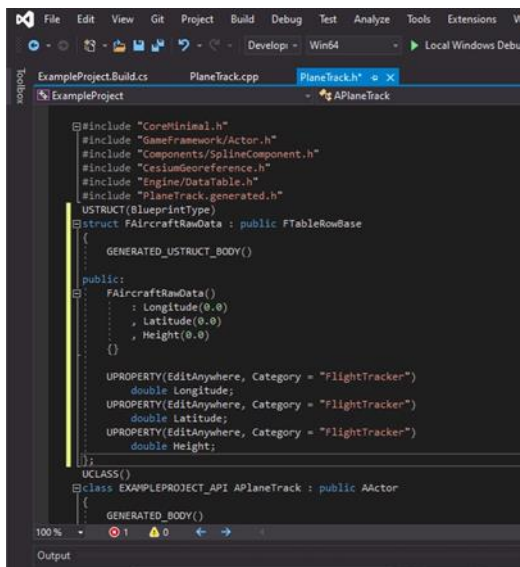
Step 4: Add Positions to the Flight Track

In this step, you will add some more code to the PlaneTrack class to complete the rest of the functionality needed to create the spline path.

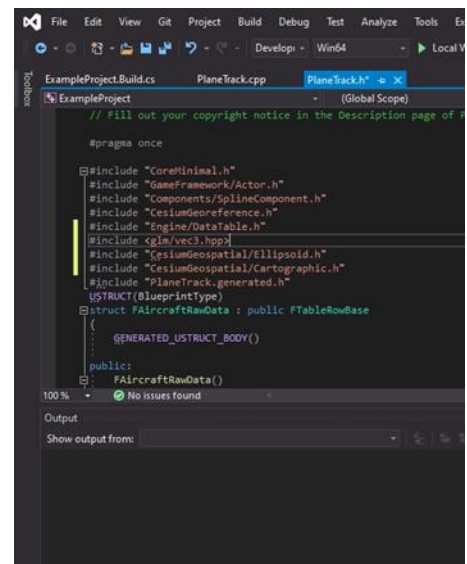
Add the following imports to the PlaneTrack.h file, above the PlaneTrack.Generated.h line.

```
// Imports should be placed above the PlaneTrack.Generated.h line.  
  
#include <glm/vec3.hpp>  
  
#include "CesiumGeospatial/Ellipsoid.h"  
  
#include "CesiumGeospatial/Cartographic.h"
```

Before you add the code it should look like:



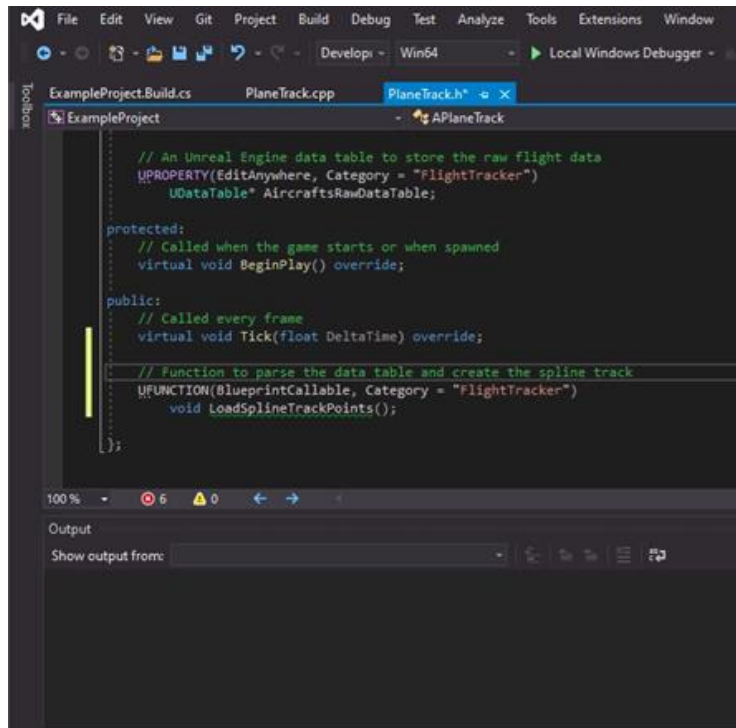
After it should look like:



While still in PlaneTrack.h, add the following function at the end of the APlaneTrack class definition.

```
public:  
  
// Function to parse the data table and create the spline track  
UFUNCTION(BlueprintCallable, Category = "FlightTracker")  
  
void LoadSplineTrackPoints();
```

After, it should look like:



The screenshot shows the Visual Studio Code editor with the 'PlaneTrack.h' file open. The file contains C++ code for an Unreal Engine actor. The code includes a data table property, a BeginPlay override, a Tick override, and a UFUNCTION for loading spline track points.

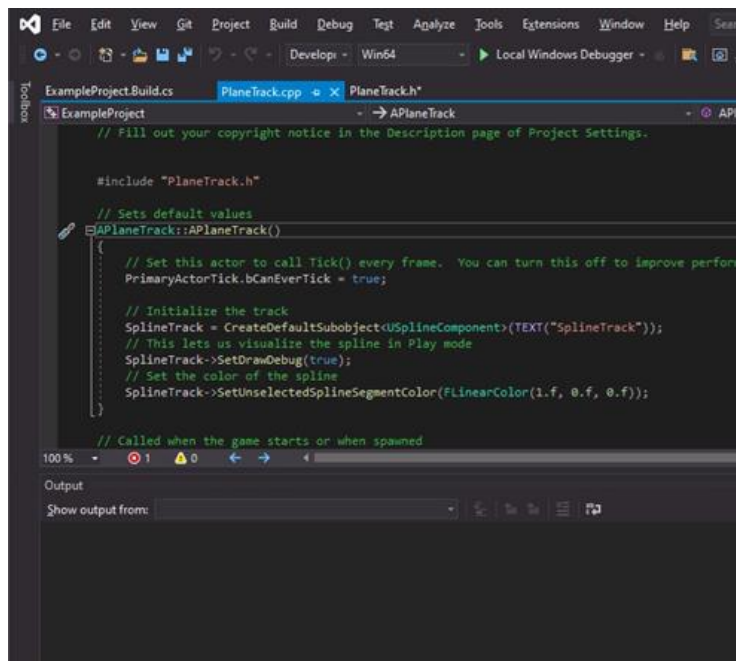
```
// An Unreal Engine data table to store the raw flight data
UPROPERTY(EditAnywhere, Category = "FlightTracker")
UDataTable* AircraftsRawDataTable;

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    // Function to parse the data table and create the spline track
    UFUNCTION(BlueprintCallable, Category = "FlightTracker")
    void LoadSplineTrackPoints();
};
```

Switch to PlaneTrack.cpp file. Before you make any changes, it should look like:



The screenshot shows the Visual Studio Code editor with the 'PlaneTrack.cpp' file open. The code includes the header file, sets default values for the actor, initializes the spline track, and implements the BeginPlay and Tick methods.

```
// Fill out your copyright notice in the Description page of Project Settings.

#include "PlaneTrack.h"

// Sets default values
APlaneTrack::APlaneTrack()
{
    // Set this actor to call Tick() every frame. You can turn this off to improve performance
    PrimaryActorTick.bCanEverTick = true;

    // Initialize the track
    SplineTrack = CreateDefaultSubobject<USplineComponent>(TEXT("SplineTrack"));
    // This lets us visualize the spline in Play mode
    SplineTrack->SetDrawDebug(true);
    // Set the color of the spline
    SplineTrack->SetUnselectedSplineSegmentColor(FLinearColor(1.f, 0.f, 0.f));
}

// Called when the game starts or when spawned
```

Switch to PlaneTrack.cpp. Add the following code snippet at the bottom of the file to create the body of LoadSplineTrackPoints:

```
void APlaneTrack::LoadSplineTrackPoints()
{
    if (this->AircraftsRawDataTable != nullptr && this->CesiumGeoreference != nullptr)
    {
        int32 PointIndex = 0;
        for (auto& row : this->AircraftsRawDataTable->GetRowMap())
        {
            FAircraftRawData* Point = (FAircraftRawData*)row.Value;
            // Get row data point in lat/long/alt and transform it into UE4 points
            double PointLatitude = Point->Latitude;
            double PointLongitude = Point->Longitude;
            double PointHeight = Point->Height;

            // Compute the position in UE coordinates
            glm::dvec3 UECoords = this->CesiumGeoreference-
>TransformLongitudeLatitudeHeightToUe(glm::dvec3(PointLongitude, PointLatitude,
PointHeight));
            FVector SplinePointPosition = FVector(UECoords.x, UECoords.y, UECoords.z);
            this->SplineTrack->AddSplinePointAtIndex(SplinePointPosition, PointIndex,
ESplineCoordinateSpace::World, false);

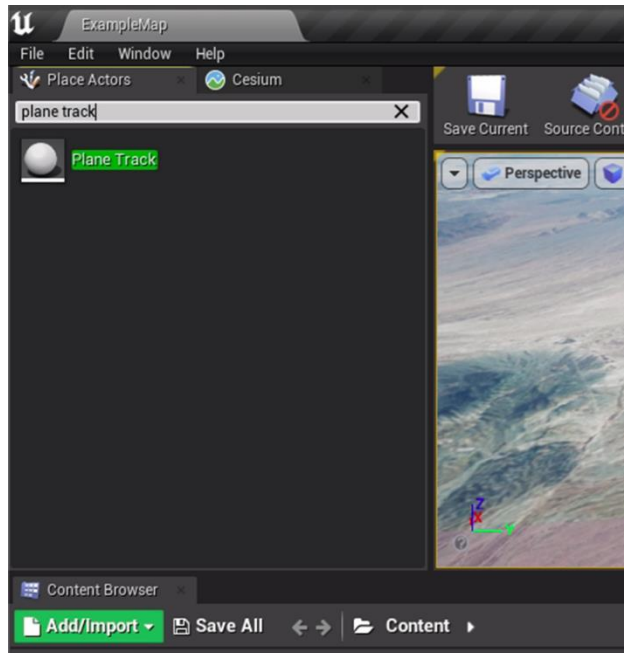
            // Get the up vector at the position to orient the aircraft
            const CesiumGeospatial::Ellipsoid& Ellipsoid =
CesiumGeospatial::Ellipsoid::WGS84;
            glm::dvec3 upVector =
Ellipsoid.geodeticSurfaceNormal(CesiumGeospatial::Cartographic(FMath::DegreesToRadians(PointLongitude), FMath::DegreesToRadians(PointLatitude),
FMath::DegreesToRadians(PointHeight)));

            // Compute the up vector at each point to correctly orient the plane
            glm::dvec4 ecefUp(upVector, 0.0);
            const glm::dmat4& ecefToUnreal = this->CesiumGeoreference-
>GetEllipsoidCenteredToUnrealWorldTransform();
            glm::dvec4 unrealUp = ecefToUnreal * ecefUp;
            this->SplineTrack->SetUpVectorAtSplinePoint(PointIndex, FVector(unrealUp.x,
unrealUp.y, unrealUp.z), ESplineCoordinateSpace::World, false);

            PointIndex++;
        }
        this->SplineTrack->UpdateSpline();
    }
}
```

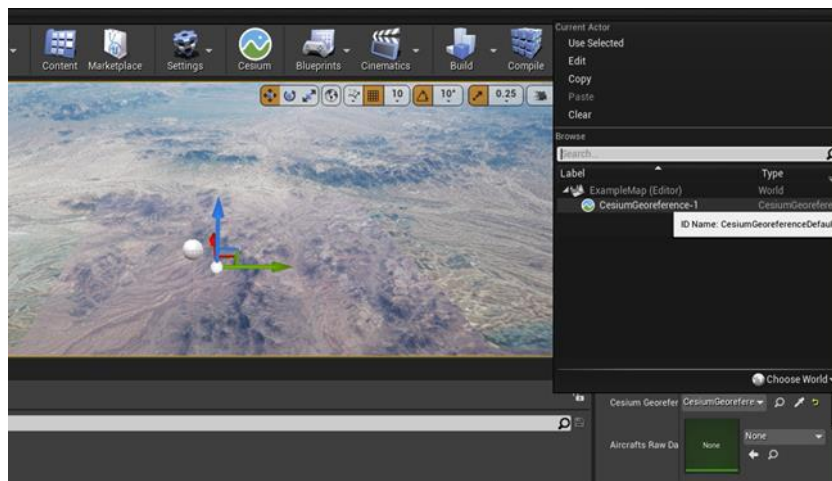
Save and compile the code. Make sure you get the “Compile Completed” notification.

Navigate back to the Unreal Engine to add the flight track to the scene. In the Place Actors panel, search for "Plane Track"

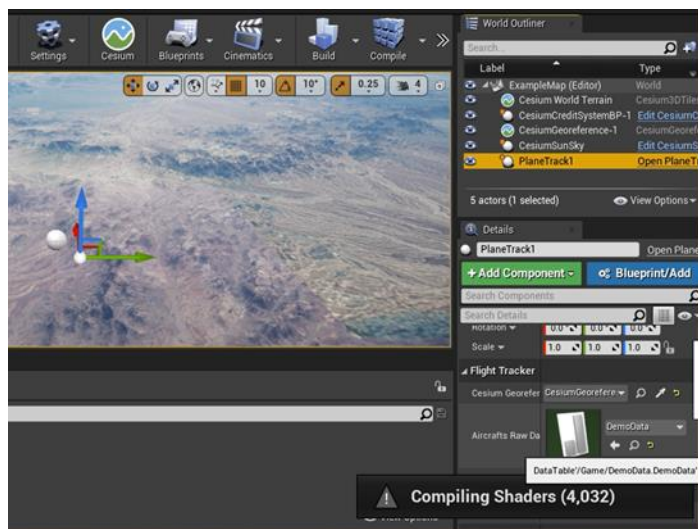


Click and drag the actor into the viewport, then drop it into the scene.

Select the PlaneTrack actor. In the Details panel, look for the Flight Tracker category. Set the Cesium Georeference variable to the Cesium Georeference variable in your scene



Then set the Aircrafts Raw Data Table variable to the data table added in step 3.

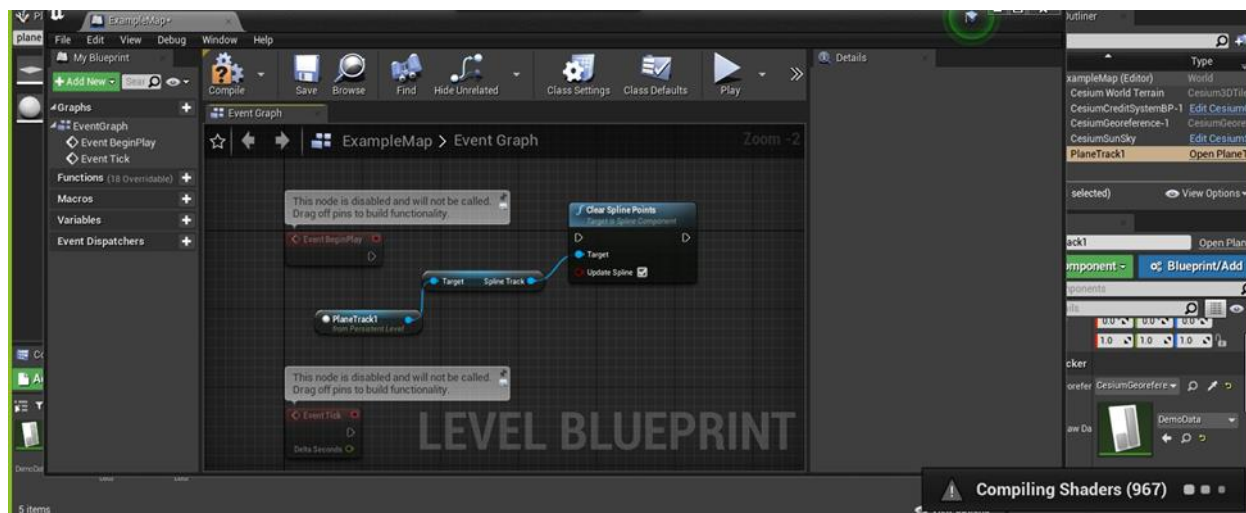


Navigate to and click on the Open Level Blueprint button in the toolbar at the top of the screen.

Near the Event BeginPlay node, Drag-and-drop the PlaneTrack actor from the World Outliner.

Right click on the PlaneTrack actor to bring up a menu of additional nodes, which are the code building blocks of Blueprint. Search for the Clear Spline Points function node and add it by clicking.

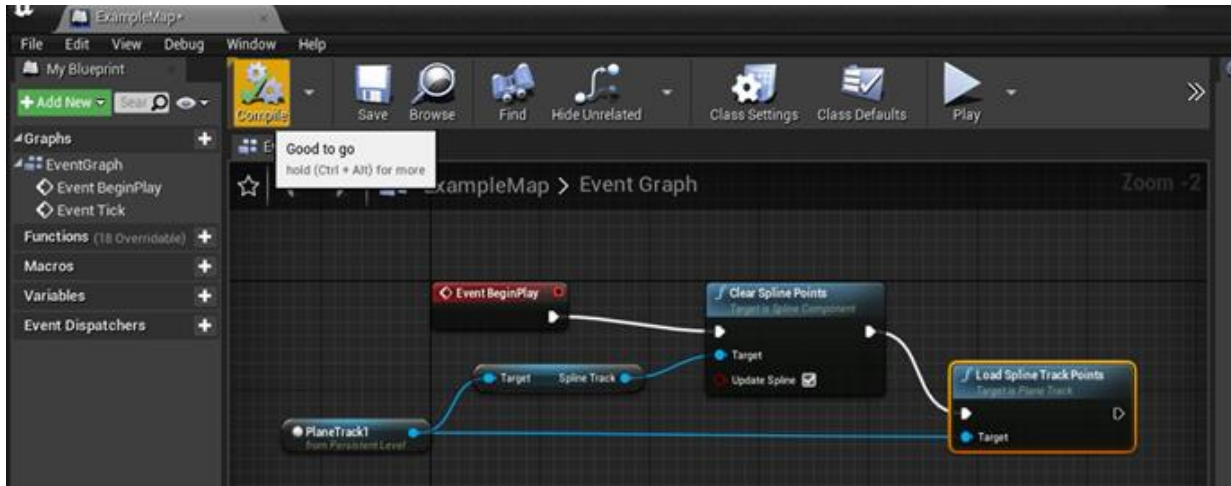
Connect the blocks by clicking and dragging the arrow from the PlaneTrack node to the Clear Spline Points node, and an additional node, Target, will automatically populate to help connect them.



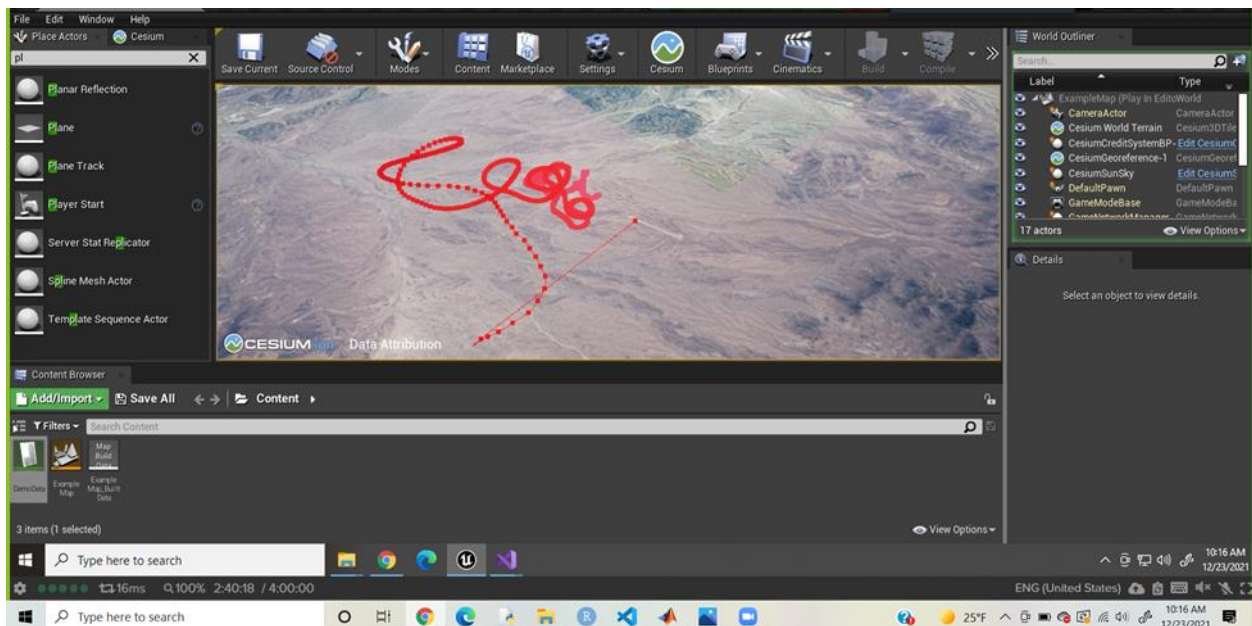
From the PlaneTrack object node, drag another connection and search for "Load Spline Track Points."

Connect the Clear Spline Points and the Load Spline Track Points nodes, and connect the Event BeginPlay node to the Clear Splint Points node. The final Blueprint network looks like this:

Click Compile at the top left corner of the Blueprint Editor, the following is the completed connections.



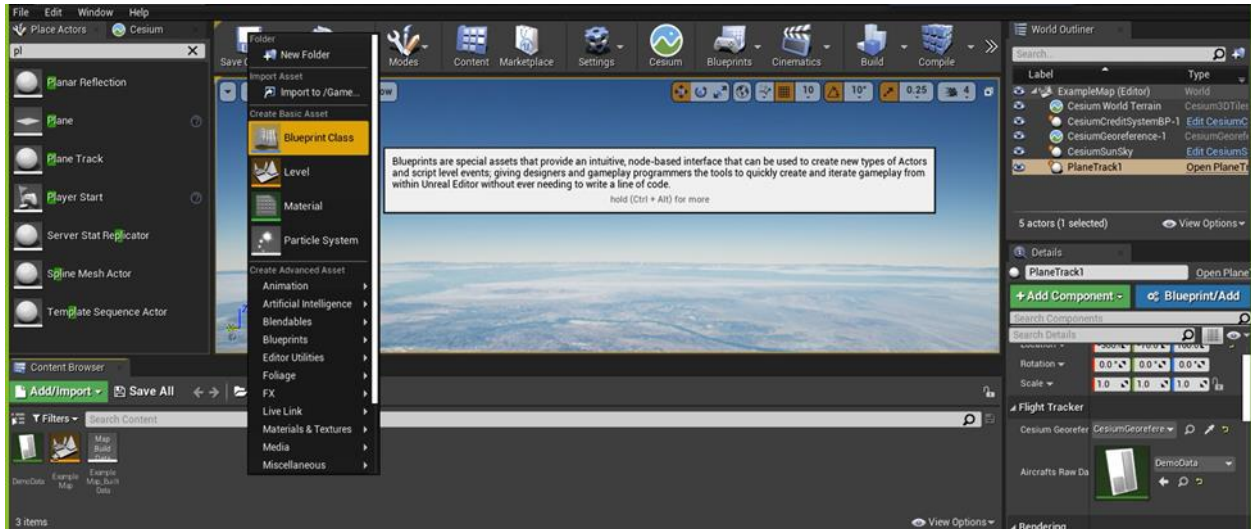
To check if everything is set up correctly, click the Play button in the top panel of the main editor. Since spline visualization is off by default, you can turn it on by selecting the viewport, pressing the key (usually under the Esc key) on your keyboard and entering in the ShowFlag.Splines 1 command. You should be able to see the data points connected by a spline curve that starts in the air and spirals towards the ground.



Step 5: Add the Aircraft

The final step to complete the flight tracker is adding an actor that follows the spline path.

Right-click in a blank area of the Content Browser and select Blueprint Class.



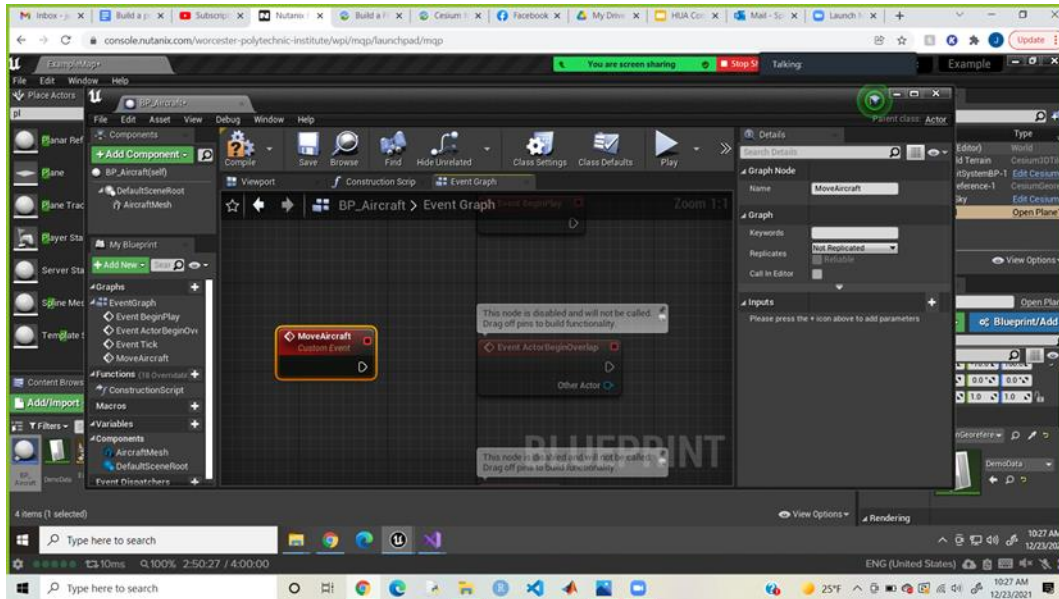
When prompted to pick the Parent Class, select Actor. Name the class “BP_Aircraft” (BP stands for Blueprint).

Double-click on the new Blueprint class to access the [Blueprint Editor](#). Click on the green **Add Component** button at the top left corner

Search for “Static Mesh”. Add it to the component list and name it “AircraftMesh”.

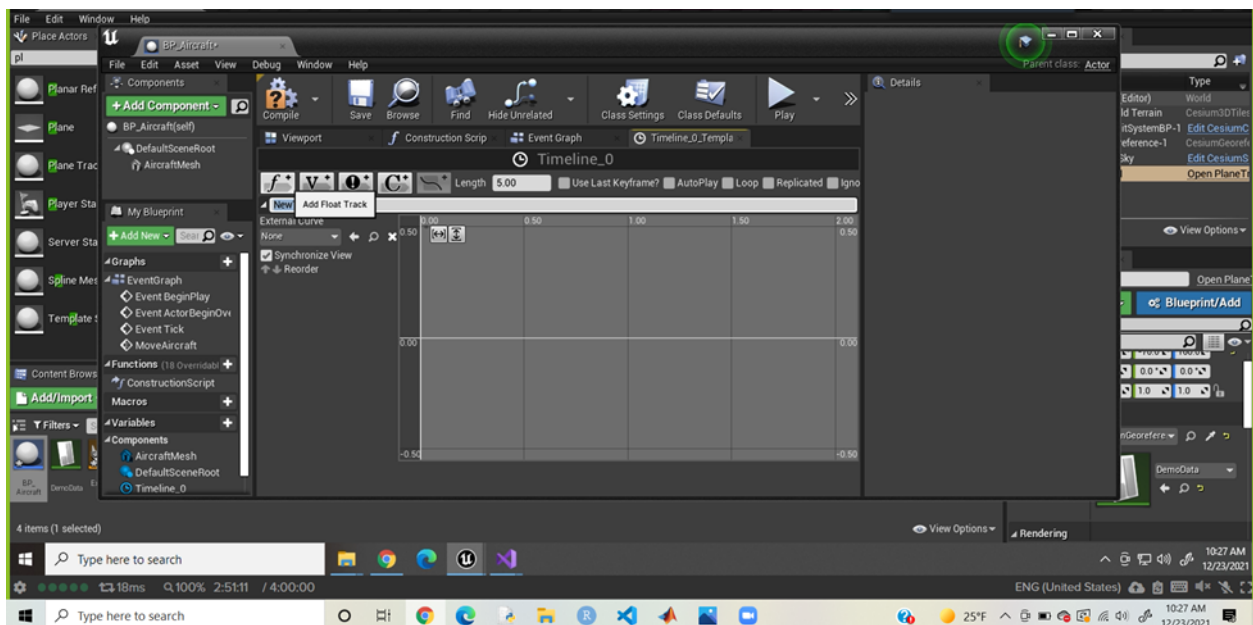
Head to the Event Graph by clicking the Event Graph tab at the top of the Blueprint Editor.

Right-click anywhere in the Event Graph and search for “Custom Event”. Name this event “MoveAircraft”. The following is what your editor should look like:



Right-click again in the Event Graph and search for “Add Timeline”.

Double-click on the Timeline node to open the Timeline Editor. Then create a float curve by clicking on the **Add Float Track** button at the top left corner of the editor and give it a name. In this tutorial, the Float Track is called “Alpha”.

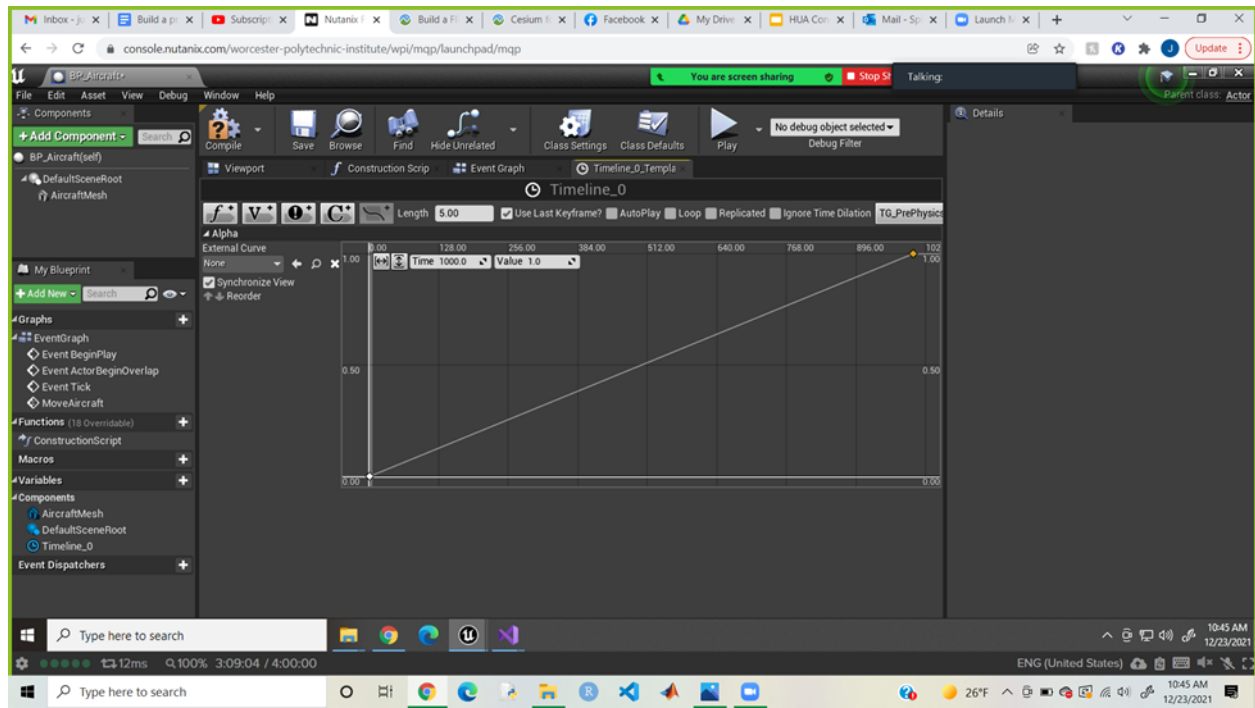


Add keyframes to the timeline by right-clicking on the curve and select Add key to Curve.

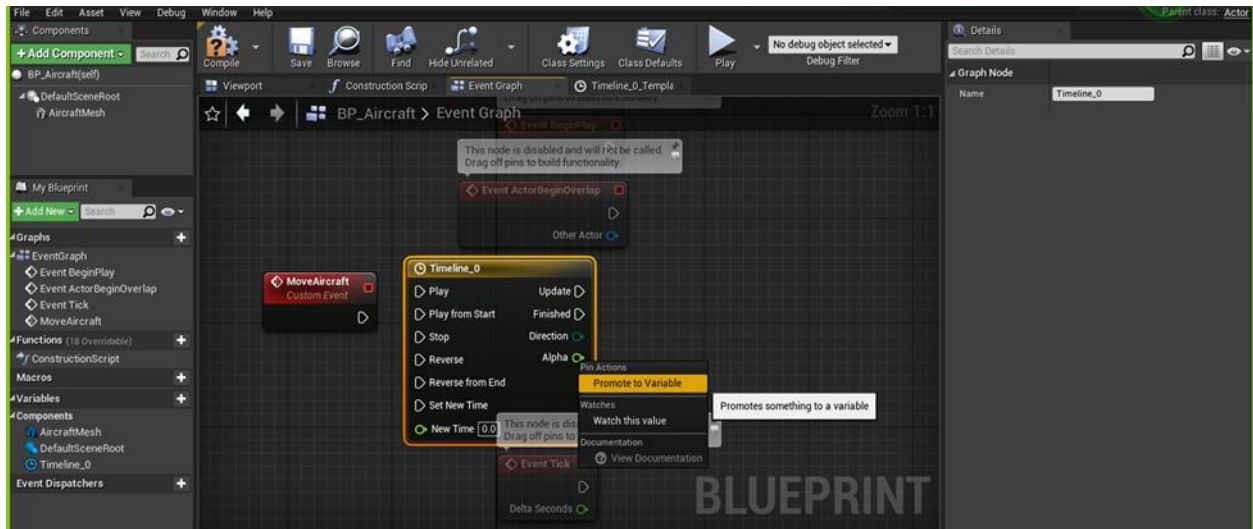
The first key point is at Time = 0, Value = 0. To assign precise values to a key point, select it and type in the values at the top left corner. Then add a second key point by again right-clicking on the curve and select Add key to Curve. This key point should be set at Time = 1, Value = 1000:

To adjust the view so you can see the whole curve, click the **Zoom for Fit Horizontal** and **Zoom to Fit Vertical** buttons in the top left-hand corner of the viewport.

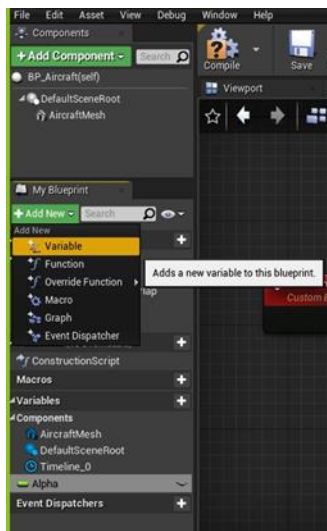
Then, at the top of the timeline window, set **Length** as 5 and check **Use Last Keyframe** if it is not already checked. Then, in the toolbar, select **Save**.



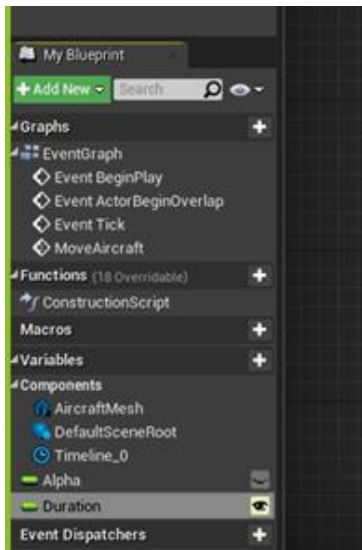
Head back to the Event Graph tab and right-click the Alpha output pin of the Timeline node and select Promote to variable. This will add a new variable to the left panel under Components:



Navigate to the My Blueprint panel, click the **Add New** button, and select **Variable** from the drop-down menu. Name the variable "Duration." Give it a type of float.



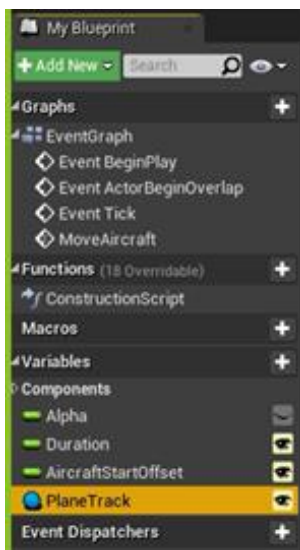
On the right side in the My Blueprint panel, click on the closed eye symbol next to duration to open the eye and make it public and editable in the main Unreal Engine editor.



Similarly, add the following remaining variables to the Plane Blueprint class:

- **AircraftStartOffset**: type `float`; public visibility; The Slider Range and Value Range should be between 0 and 1, since the timeline goes between 0 and 1. These variables can be edited in the Details panel in the Blueprint Editor.
- **PlaneTrack**: type `PlaneTrack` with Object Reference; public visibility.

The final Components list looks like this:



Compile the Blueprint.

The screenshot displays a Blueprint graph with the following nodes and connections:

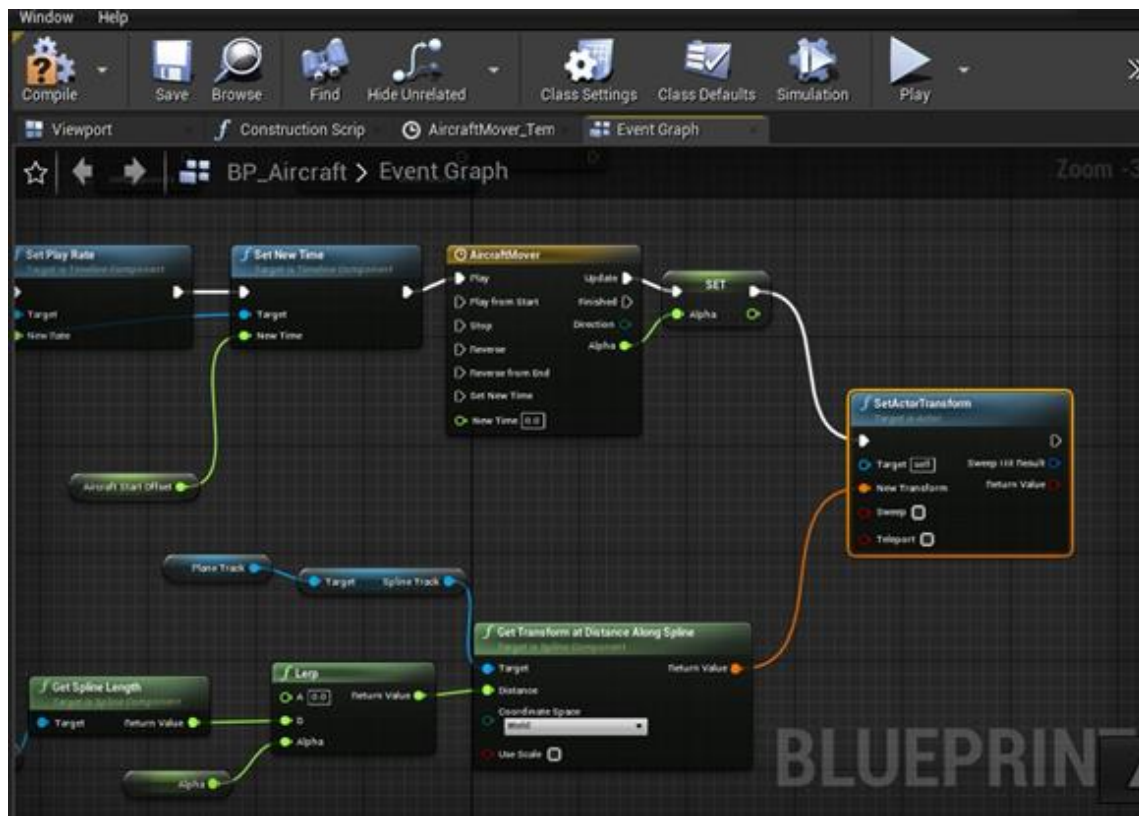
- Plane Track** (Input) connects to **Target**.
- Target** connects to **Spline Track**.
- Spline Track** connects to **Get Spline Length**.
- Get Spline Length** (Return Value) connects to **Leap**.
- Alpha** (Input) connects to **Leap**.
- Leap** (Return Value) connects to **Get Transform at Distance Along Spline**.
- Get Transform at Distance Along Spline** (Target) connects to the final output.

The **Get Transform at Distance Along Spline** node has the following settings:

- Target**: Spline Component
- Distance**: (Input from Leap)
- Coordinate Space**: World
- Use Scale**: (Checked)

The word "BLUEPRINT" is visible in the bottom right corner of the image.

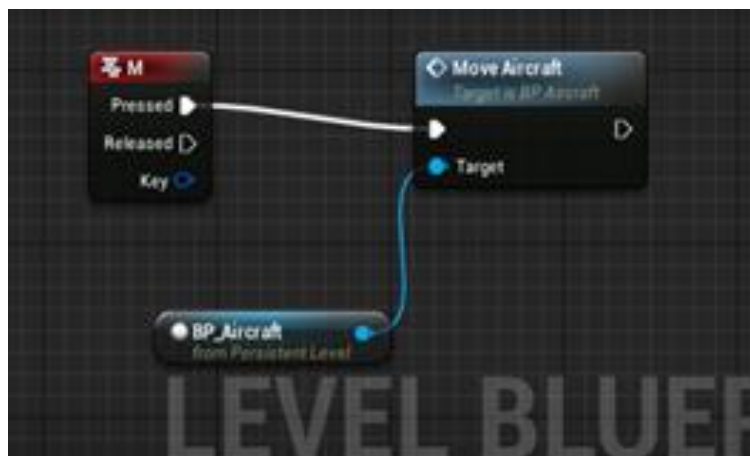
Lastly, connect the two Blueprint networks together with the Set Actor Transform node:



Compile the Blueprint.

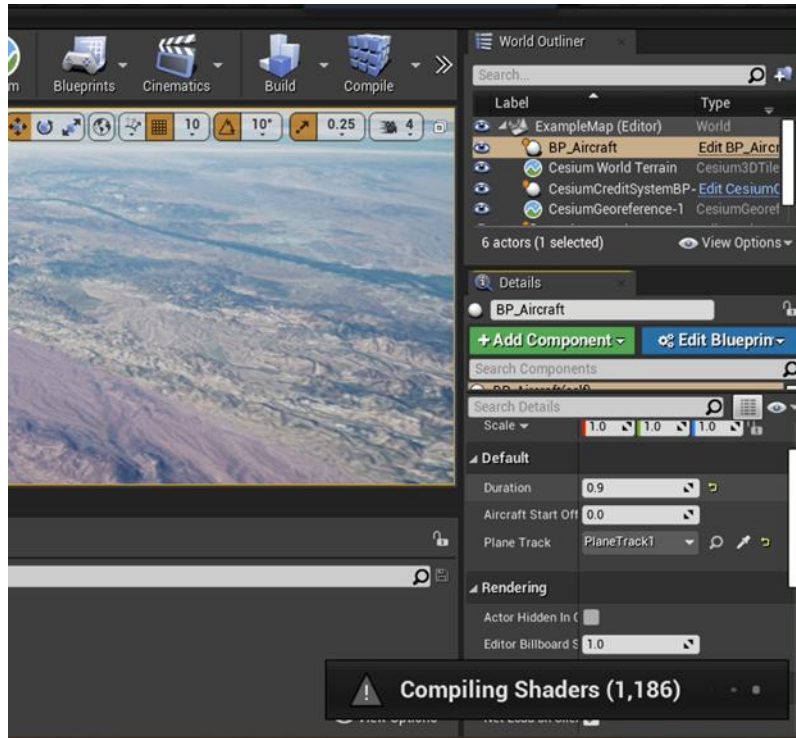
Navigate back to the main editor. Add your aircraft object to the scene by drag-and-dropping the BP_Aircraft Blueprint into the scene.

Head back to the Level Blueprint. In the Event Graph, add a Keyboard node (M is used here). Connect this Keyboard node to the Move Aircraft event as follow:



Compile the Blueprint.

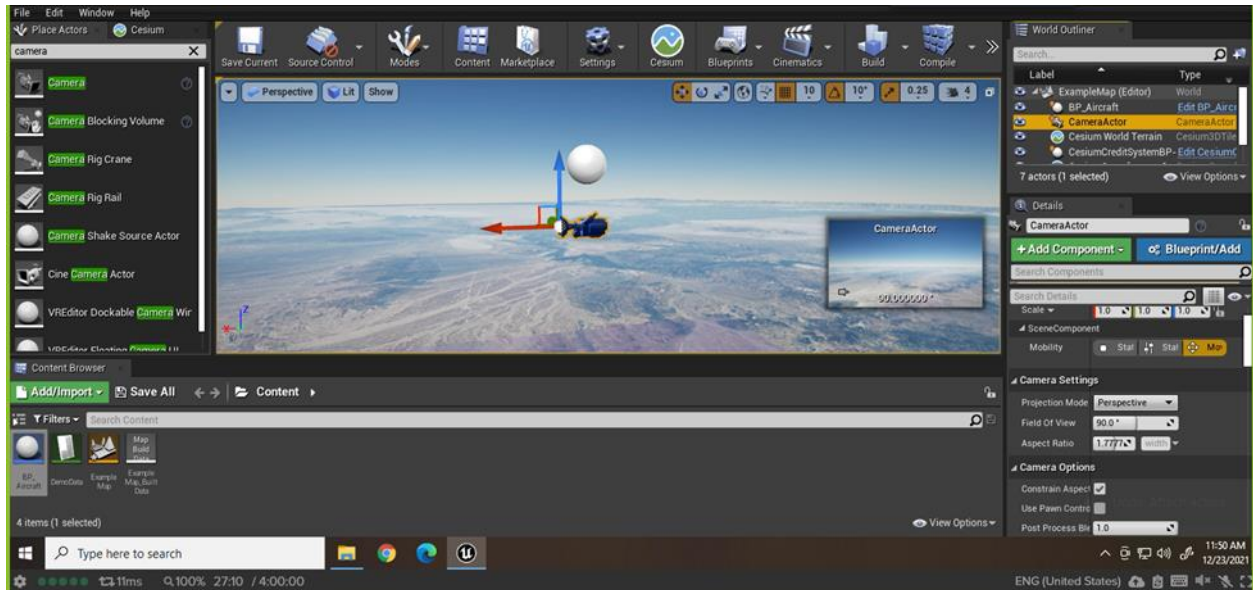
With the BP_Aircraft actor selected, head to the Details panel. Set PlaneTrack to PlaneTrack1 from the drop down. Set Duration to 0.9. Leave AirplaneOffset at 0.0. The smaller the duration the slower the flight.



With the viewport focused on the aircraft, click the Play button and press M (or the key that you chose to connect the Move Aircraft event to) to start the flight.

Step 6: Adding a Camera

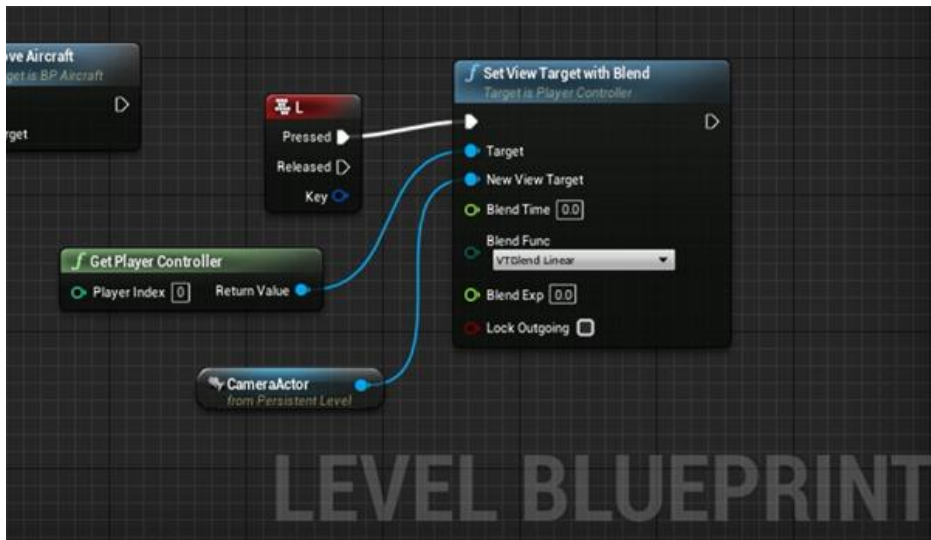
Next, we will mount a camera to the parachute so we can follow the parachute as it drops. Using the Place Actor panel, add a Camera Actor to the scene.



In the World Outliner panel on the right, drag-and-drop the Camera Actor onto the BP_Aircraft actor so that it becomes BP_Aircraft actor's child. Then, click on the Camera Actor in the World Outline. In the Details panel, navigate to Transform. Change the green value of Rotation to -45.



Next, open the Level Blueprint. Add a custom keyboard event to change the view to the camera as seen below.

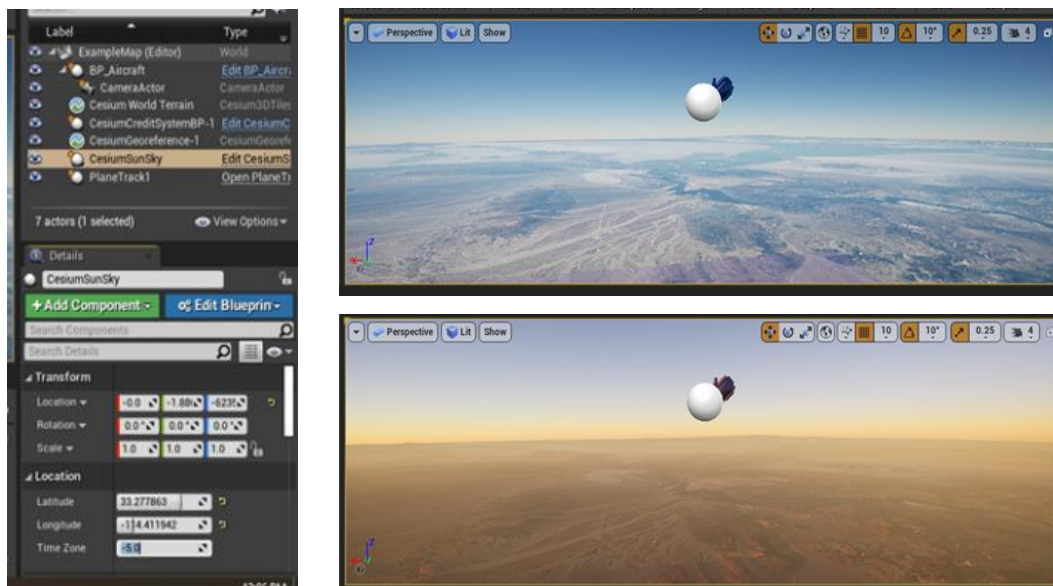


Save and compile the blueprint.

Environment Customizations

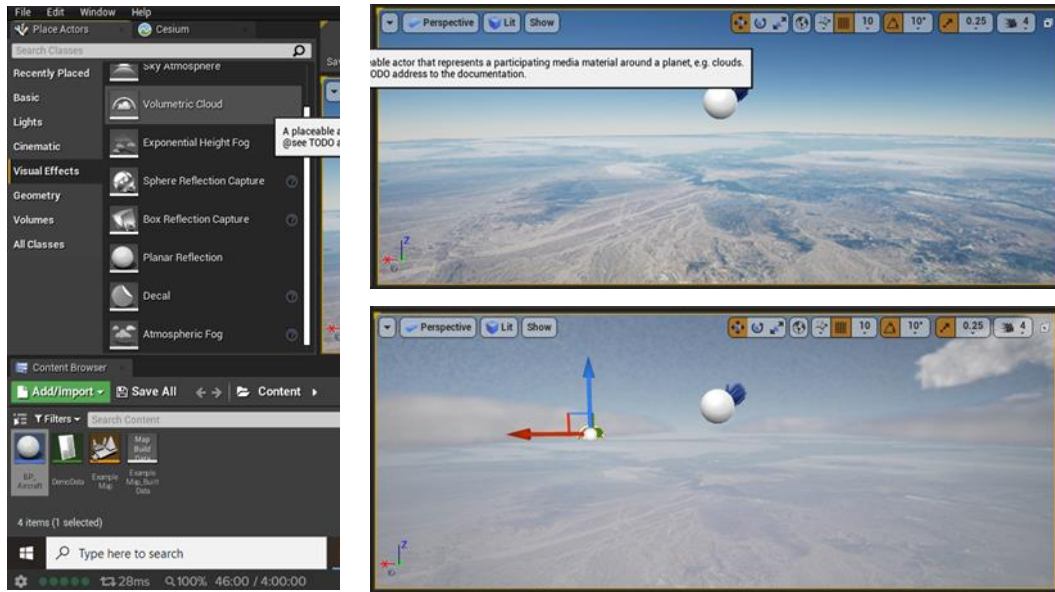
Time of Day

You can customize what time of day the game is in. Click on CesiumSunSky and in the details panel navigate to **Timezone**, there you can type in a precise value for time or click and drag the slide to adjust time of day.



Fog

There are multiple kinds of fog that can be added to a scene: Volumetric Cloud (our preference), Atmospheric Fog, and Exponential Height Fog. In the Place Actors panel select **Visual Effects** and you will see the various types of fog. Click and drag the desired fog into the viewport.



Handling the Data

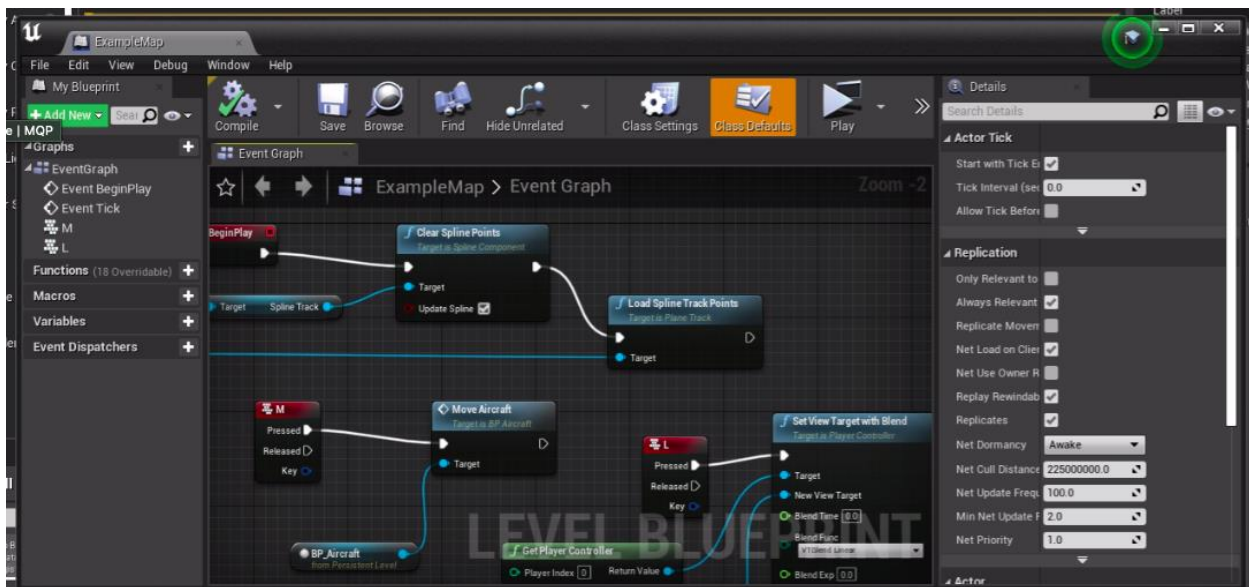
The goal of the project is to be able to collect data in the form of images taken from the drop and their corresponding location (latitude, longitude, and altitude). There are several ways to collect this data explained below.

Manual Data Collection

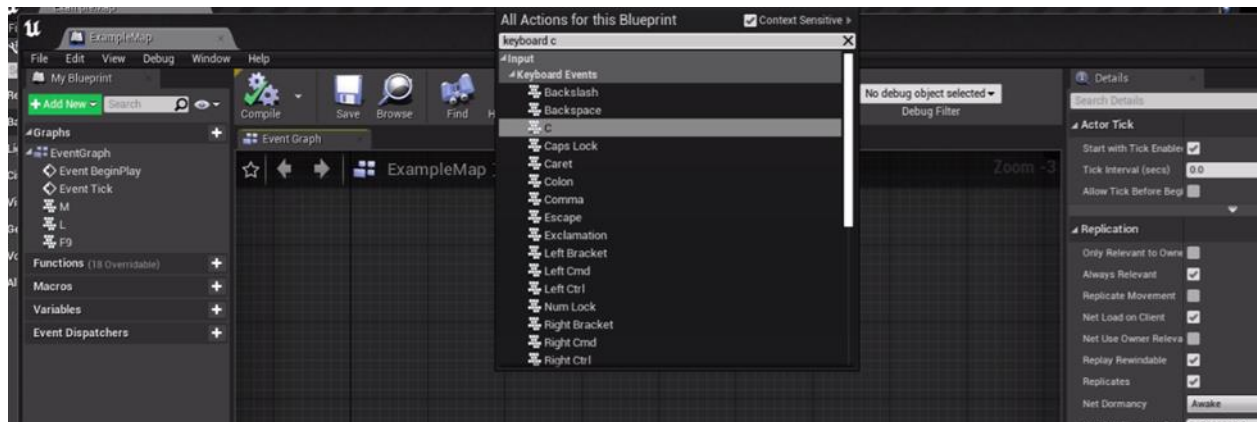
We will create a button command to take a screenshot and capture its location when we press the C key. This allows you to capture a screenshot and location at varying frequencies and however many times you wish.

To begin, open the **Level Blueprint**.

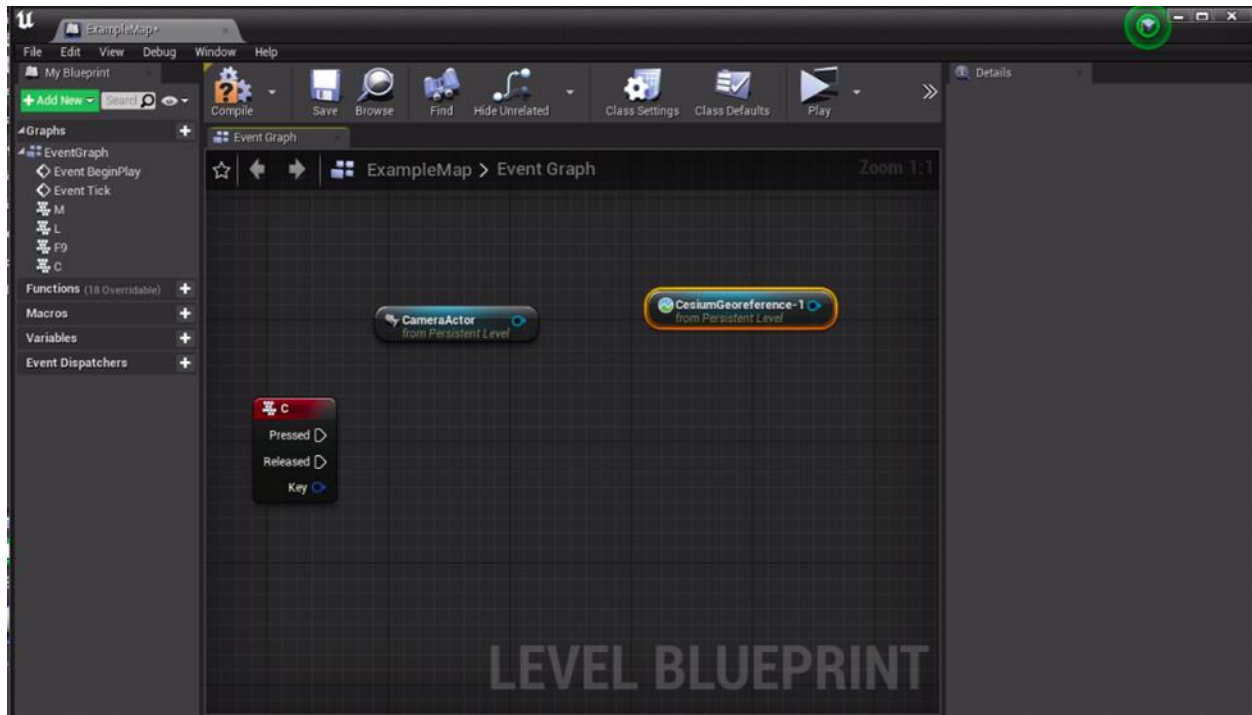
It will look like this:



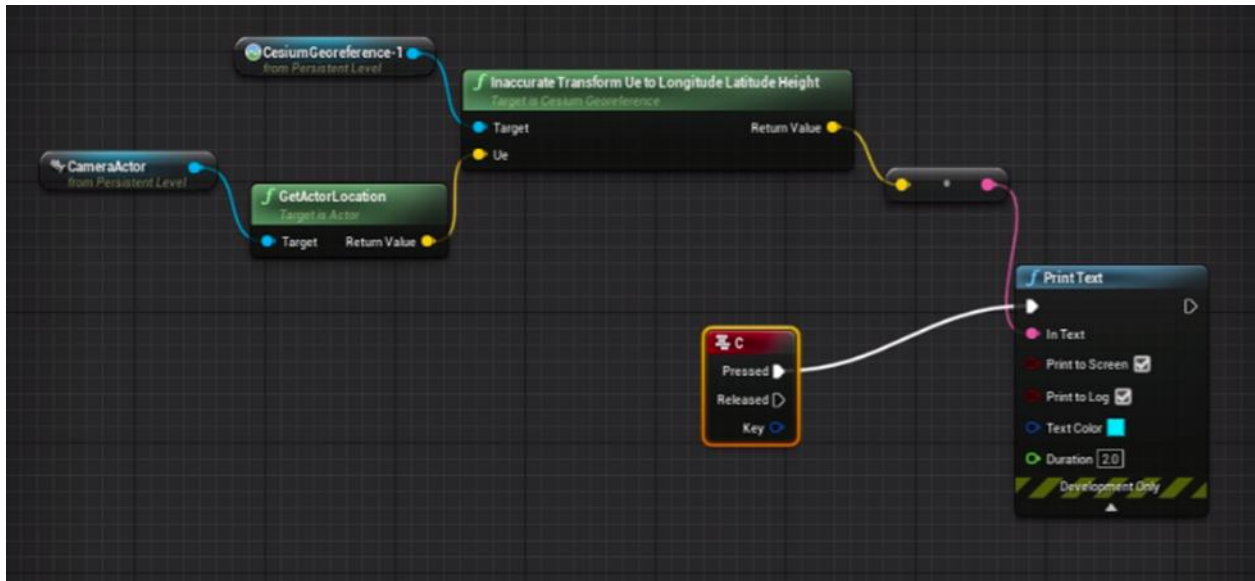
Right click and drag the background to move to a clear part of the blueprint. Then, Right click in the open space and search for 'Keyboard C'. This will be the trigger key for an image and location datapoint.



Back in the main window editor, click and drag 'CesiumGeoReference-1' from the World Outliner to the blueprint window. Click and drag in the 'CameraActor' in as well. They will appear as shown below:

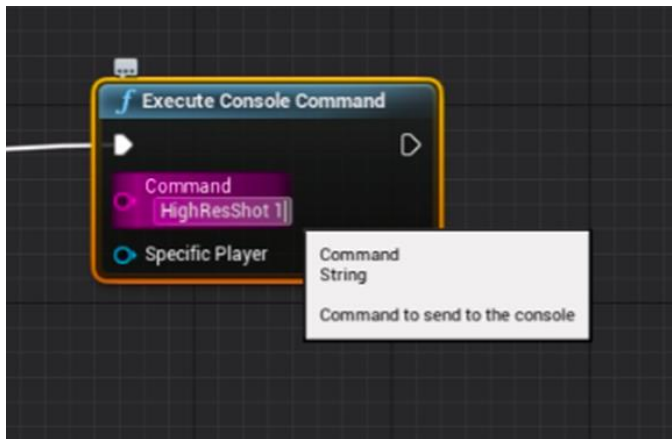


Right-click in the level blueprint to search and add the rest of the following nodes. Your final blueprint should like this:



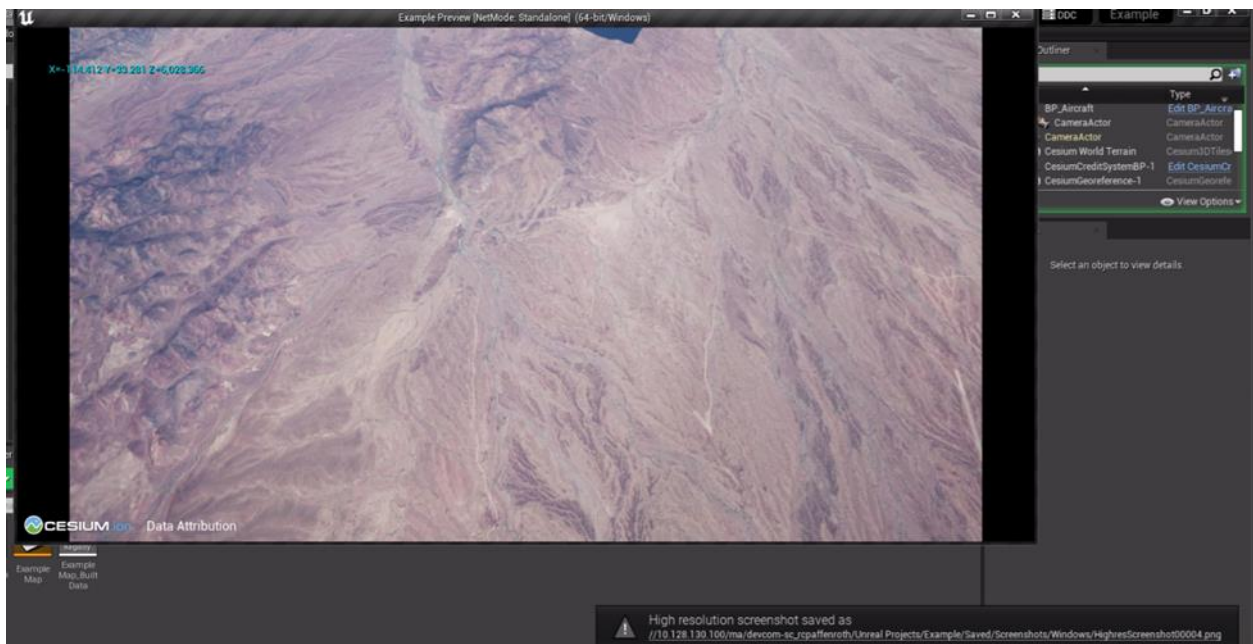
Make sure to click **Save** and then **Compile** at the top left corner of the blueprint window. This code builds the location collection function.

Next, we will build the code to capture screenshots with the same key, C. Click and drag the background to move to a clear spot in the blueprint. Right-click and search for 'Keyboard C' again. Then right-click and search for 'execute console command.' Connect the two nodes. In the Command text box within 'Execute Console Command' type in 'HighResShot 1'



Click **Save** then **Compile** in the top left corner of the blueprint window. Return to the main editor and click **Save** there too. Then to test that it is working. Hit the play button in the top right corner to open the play window.

Then click 'L' to move to the camera view and then click 'M' to begin the drop. To take a screenshot click 'C' and you will see a message in the bottom right and top right corners seen here:

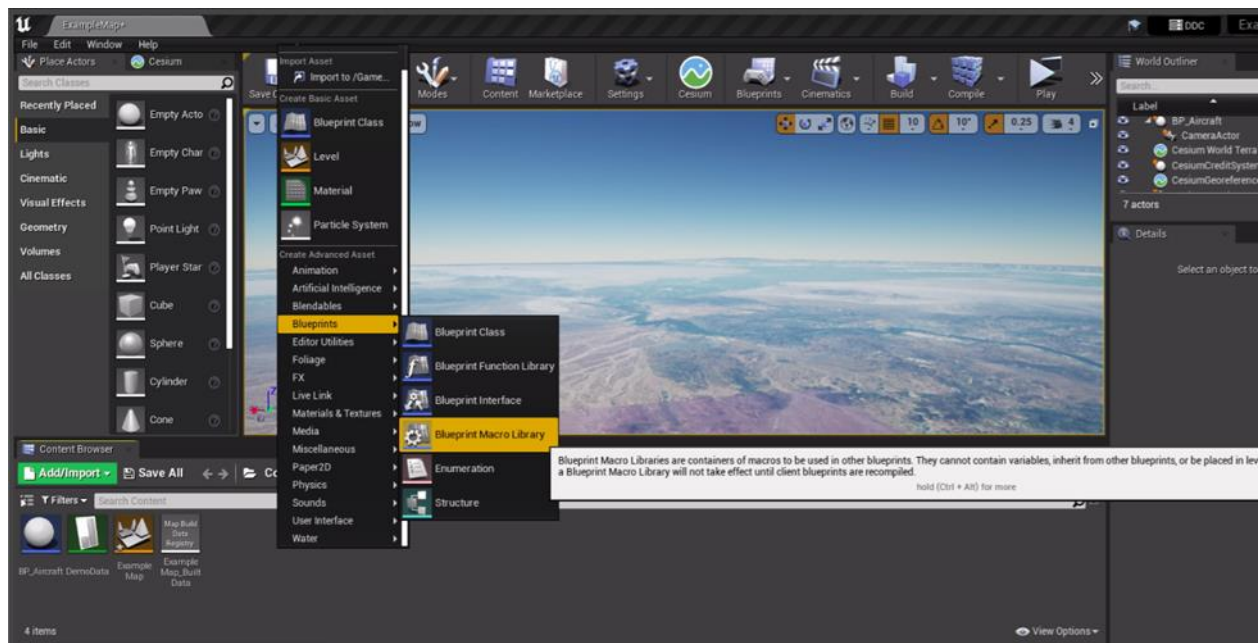


The message in the bottom right will indicate where the images are saved to. You'll see small blue text in the top left corner of the play window, in the previous image, that indicates the location of the parachute at the time the picture was taken.

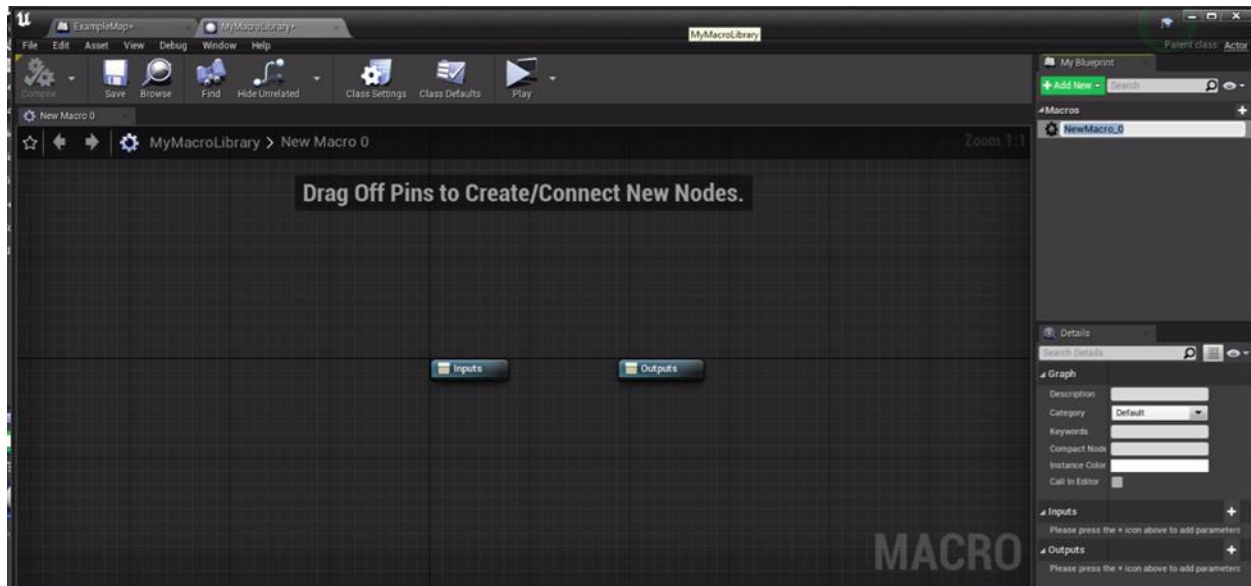
Automatic Data Collection

You can also add in a blueprint to automatically collect data at selected intervals.

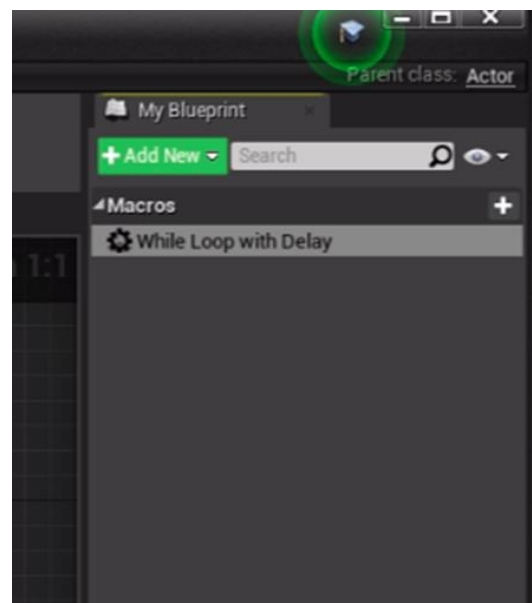
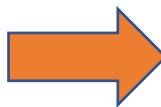
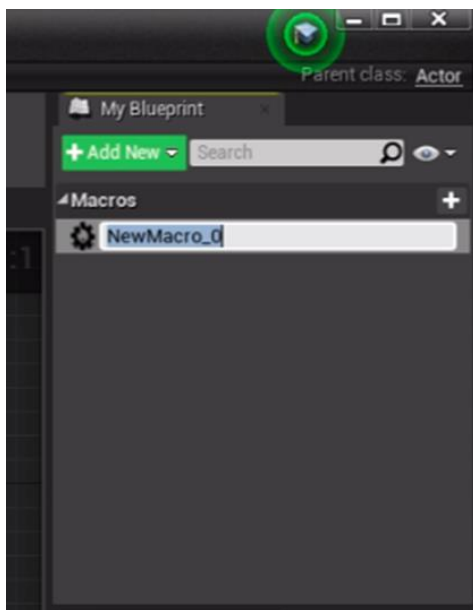
To do this, first open the level blueprint. In the blueprint, click and drag the background to move to a clear part of the blueprint. Right click and search for 'Keyboard F' and select it. Next we have to customize a while loop. Navigate back to the main editor page. On the bottom of the editor, right click in the Content Browser and go to Blueprints → Blueprint Macro Library.



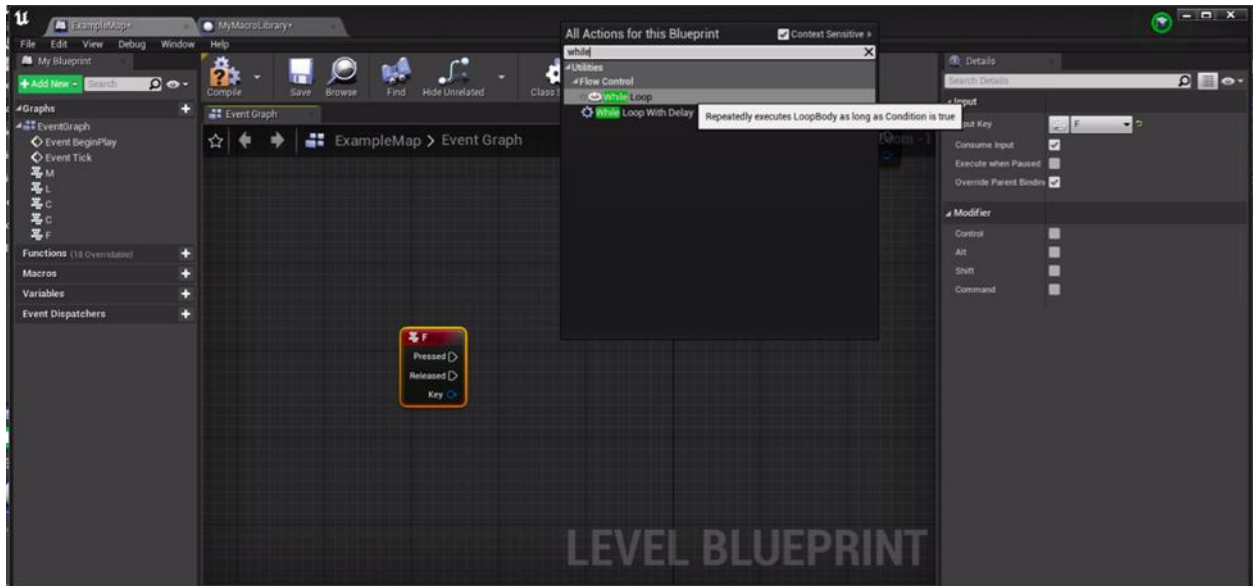
A dialog window will pop up, select **Actor** as the parent class. An object will be added to the content browser, rename it 'MyMacroLibrary'. Then double click on **MyMacroLibrary** and it will open a new tab in the level blueprint window.



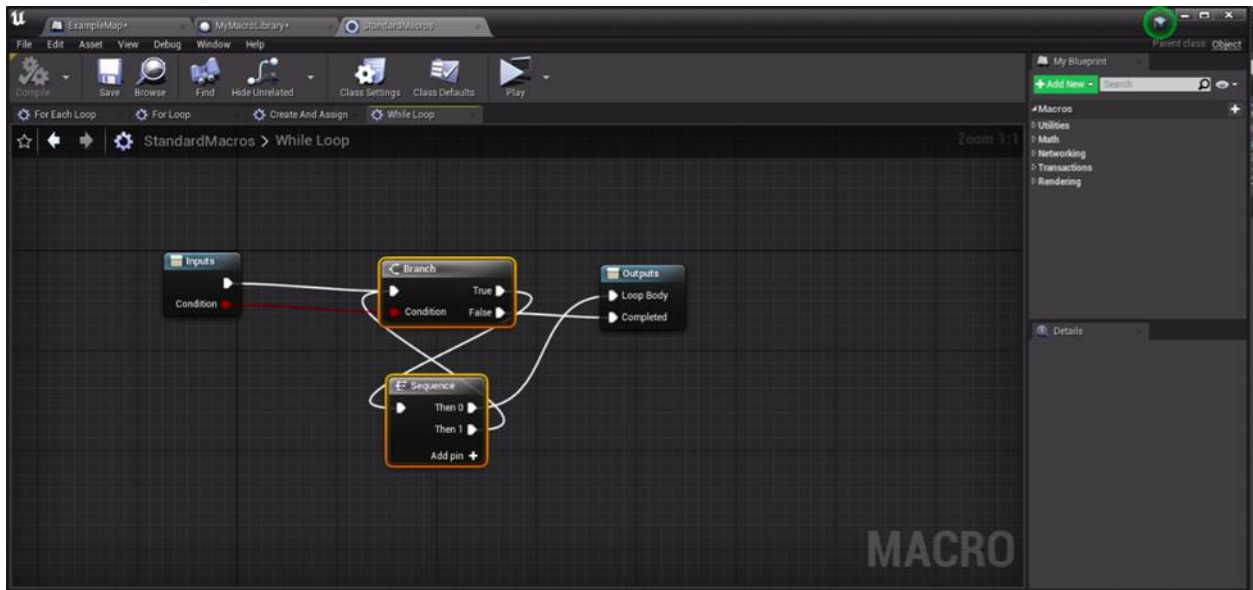
In the top right corner, there is a Blueprint tab. This custom node is considered a macro. By default, it is named 'NewMacro_0', but rename it to 'While Loop with Delay'



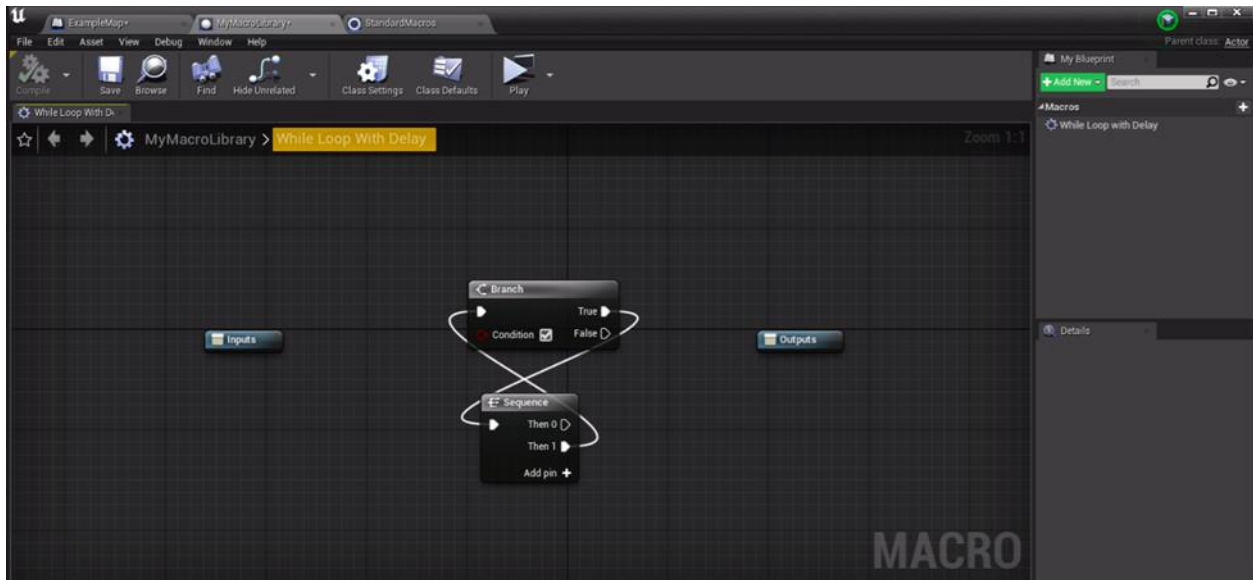
Then navigate back to the ExampleMap tab at the top of the window. Right click in the blueprint and search for 'While Loop'



The While Loop node will appear. Double-click on it to open a new tab with the blueprint for this function. Drag and select everything between the Input and Output nodes and copy them:



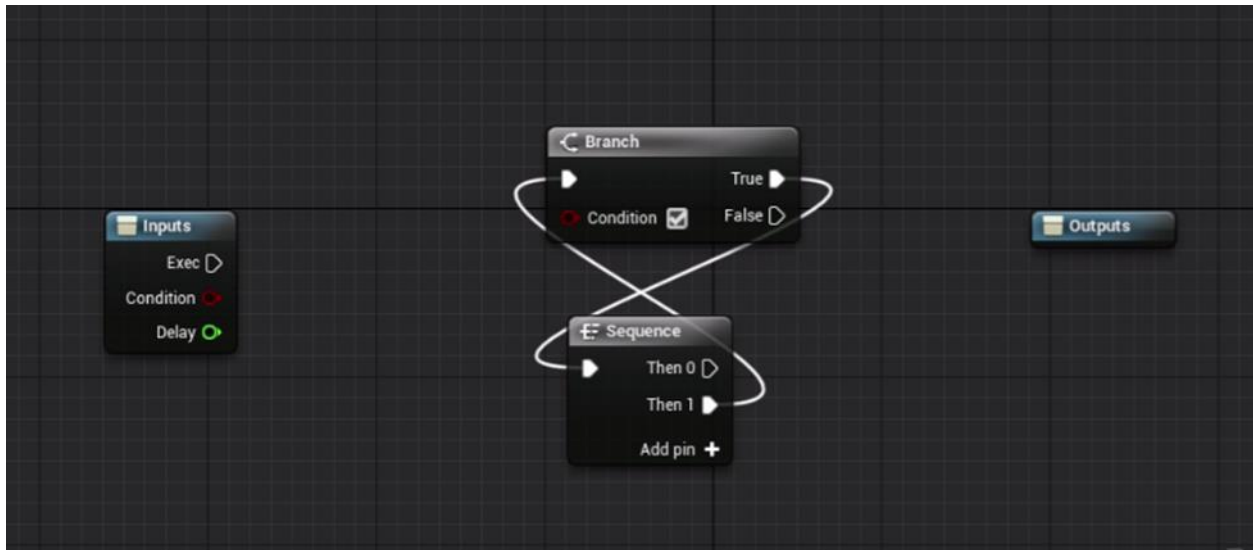
Then navigate to the MyMacroLibrary tab at the top of the window. Drag the Input and Output nodes further away from each other to create more space between them. Then paste in the node we copied earlier.



Click **While Loop with Delay** in the Blueprint tab on the right-hand side. Then navigate to the Details panel below it. Click **New Parameter** next to inputs. Rename the new input as 'Exec' and set the type to Exec with the white identifier. Then we will add two more inputs. The next one will be named 'Condition' with type Boolean with the red identifier. The last one will be named 'Delay' with type Float with the green identifier.



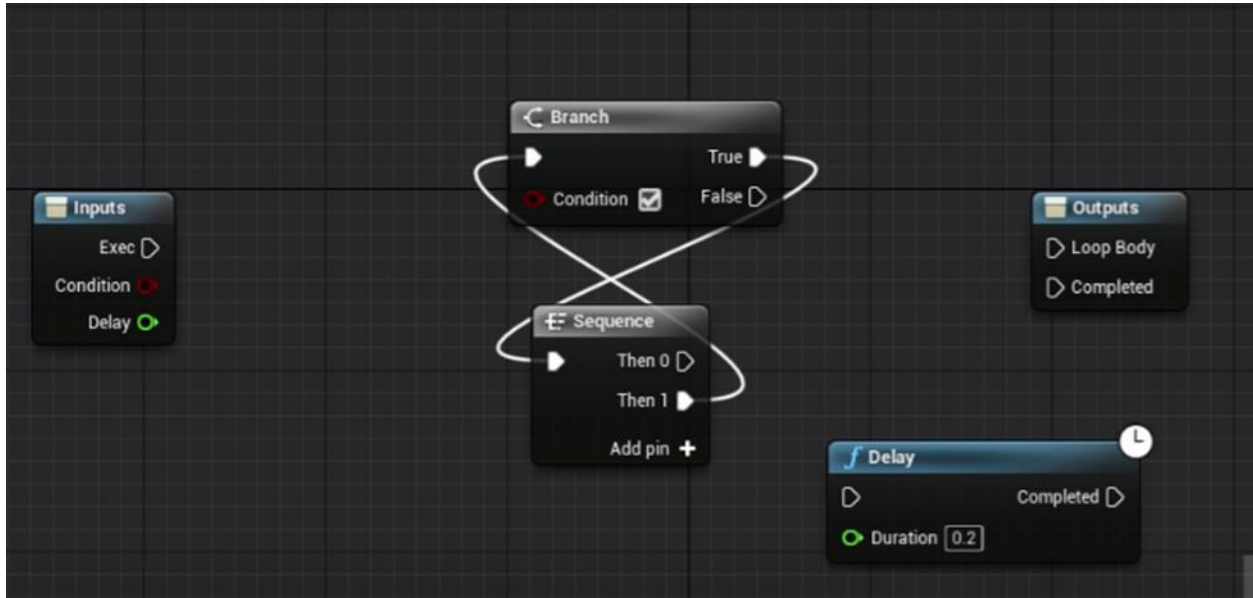
The inputs will automatically appear on the input node:



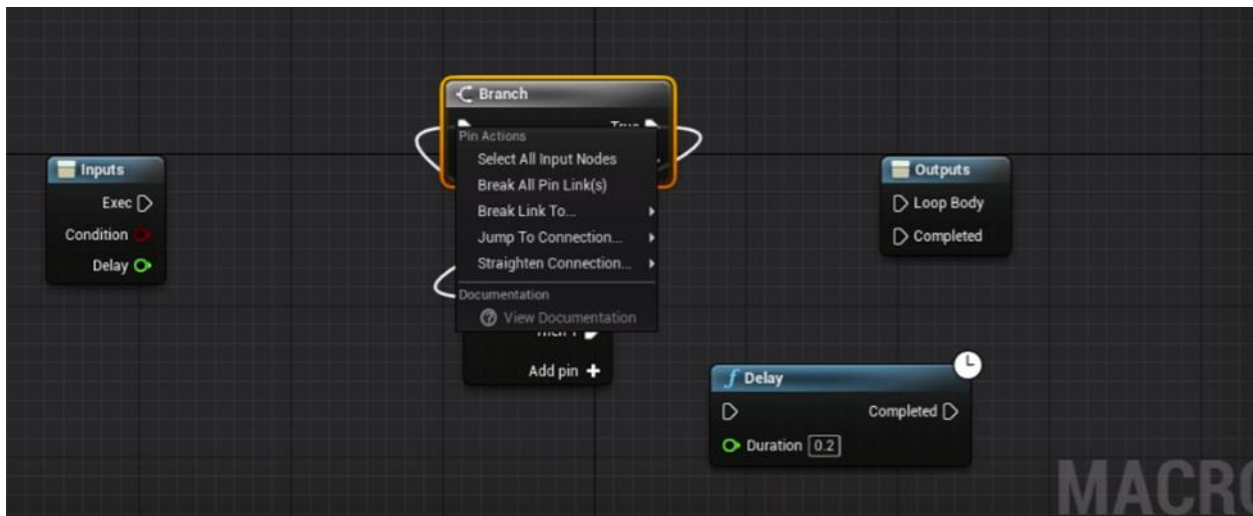
Now we will add parameters to the output node, which is also in the Details panel. The first one will be named 'Loop Body' with type Exec with the white identifier. The second one will be named 'Completed' with the type of Exec with the white identifier. The parameters will automatically appear of the output node:



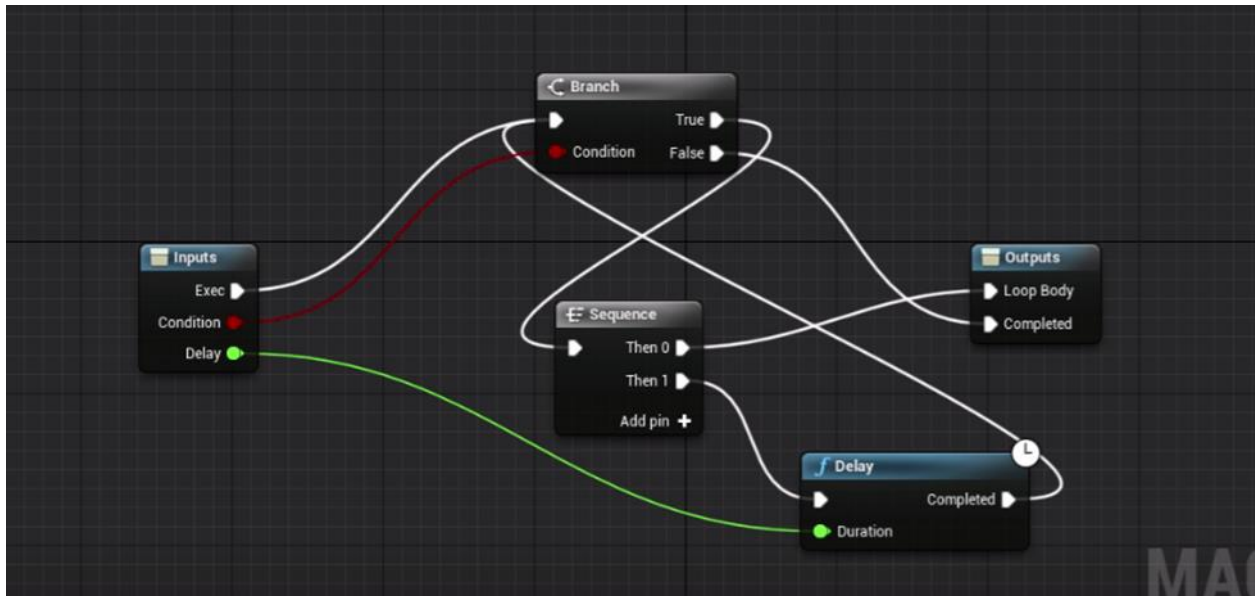
To the right of the Sequence node, right-click and search for Delay and select it.



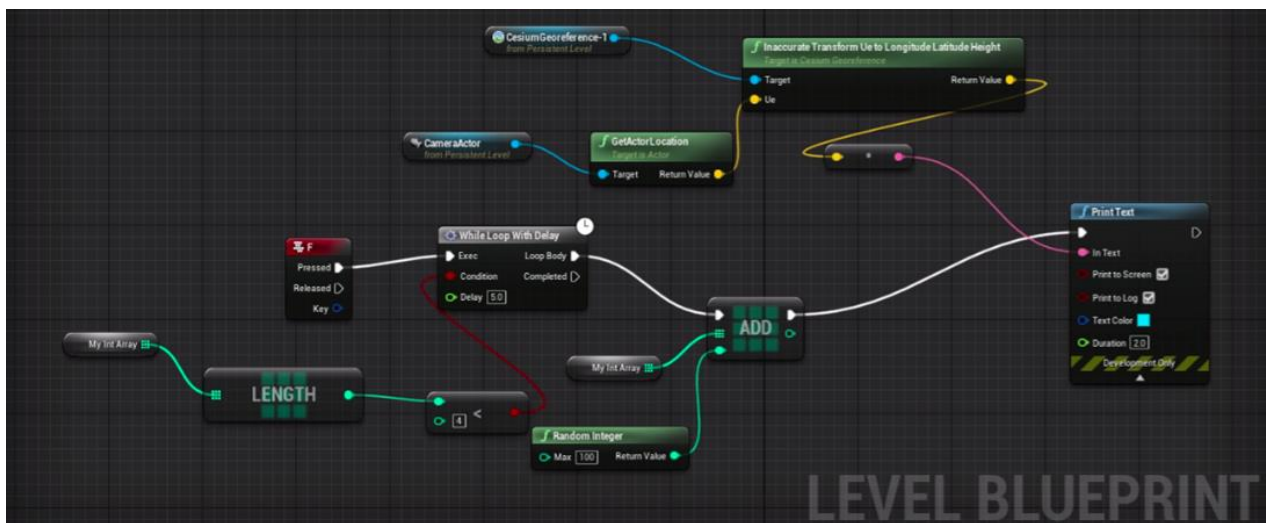
Next, we will break all the connections. Right click on the input pin for Branch and select **Break All Pin Link(s)**. Repeat this for the input pin next to False on the Branch node.



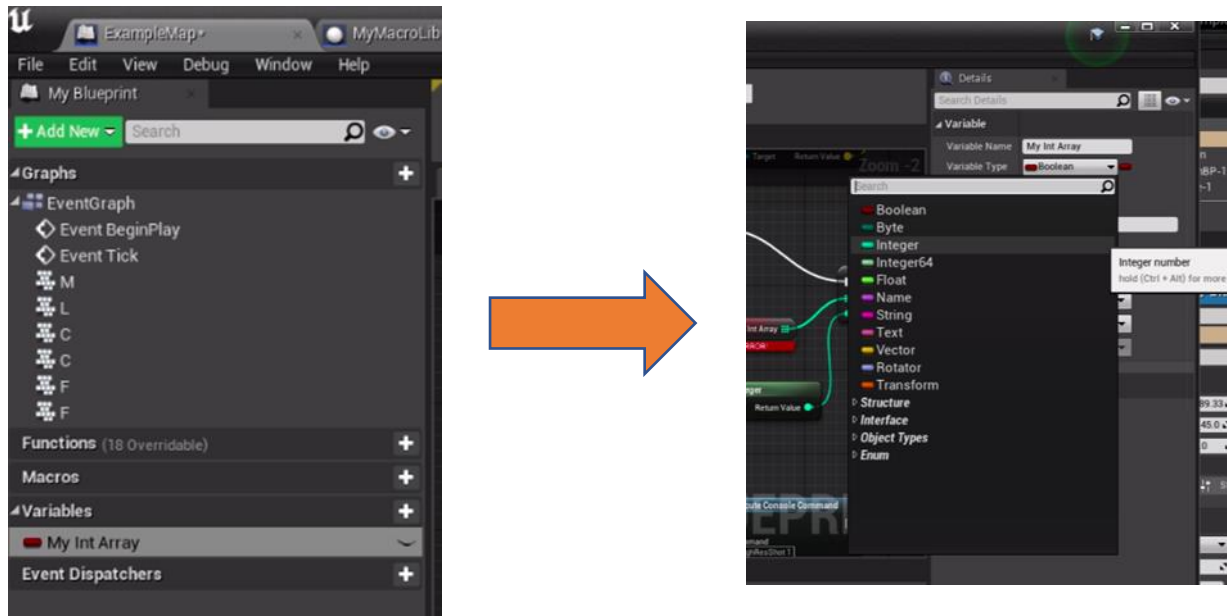
Then reconnect the nodes as seen below: Then click save in the top right.



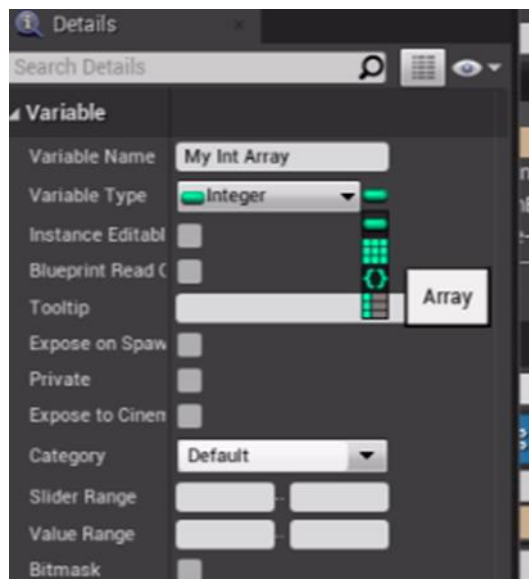
Navigate back to the ExampleMap tab and delete the While Loop node we had inserted earlier. Right-Click and search for 'While Loop with Delay' and select it. Complete the rest of the blueprint as follows:



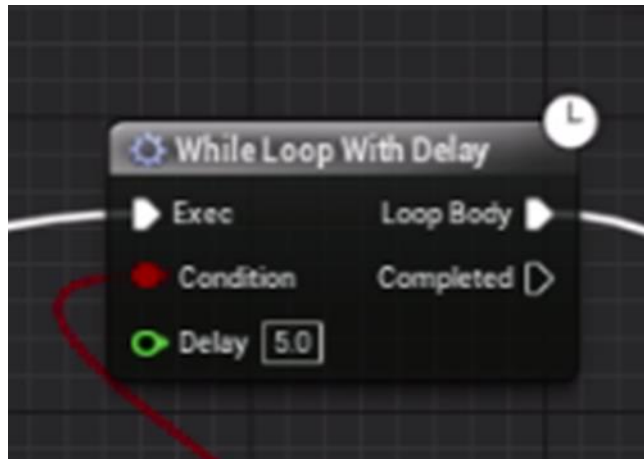
To create the array 'My Int Array' you must add a variable in the left-hand side and name it 'My Int Array'. Then on the right-hand side the details panel will appear, set the type as integer.



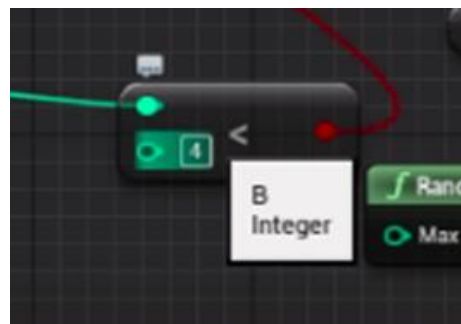
Next to the variable type, select the blue pill shaped icon and a drop down will appear. Select the 3x3 square icon to set it as an array. This variable can now be dragged and dropped into the blueprint.



This code will log the location every x seconds. To increase the time between collections, change the value of Delay in the While Loop with Delay. The picture shows it set to a 5 second delay. You can also adjust how many pictures it takes.

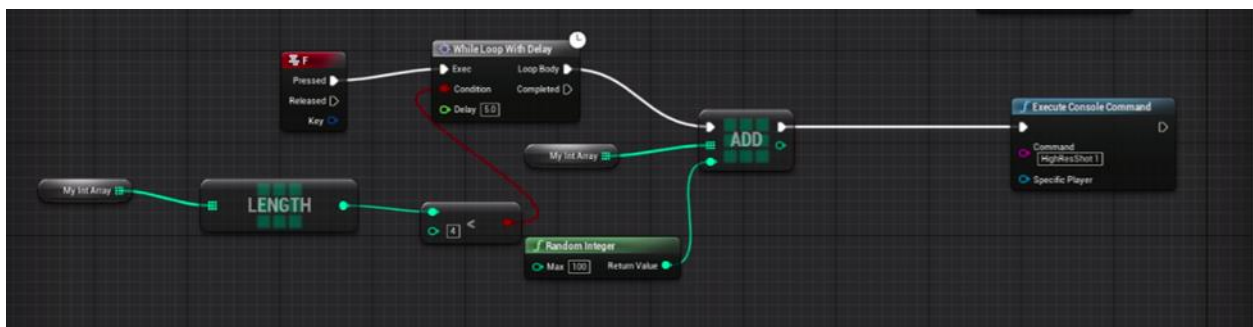


In the node shown below, the value entered is one more than the number of pictures taken. So, it is set to take 3 images.



Next, we will add in the nodes to take screenshots every time the location is logged.

Create the following section of the Blueprint. Make sure the keyboard control is the same as the location, in this case “Keyboard F.” Also make sure the values for the Delay and Number of Images are the same as the values for the location that we set above.

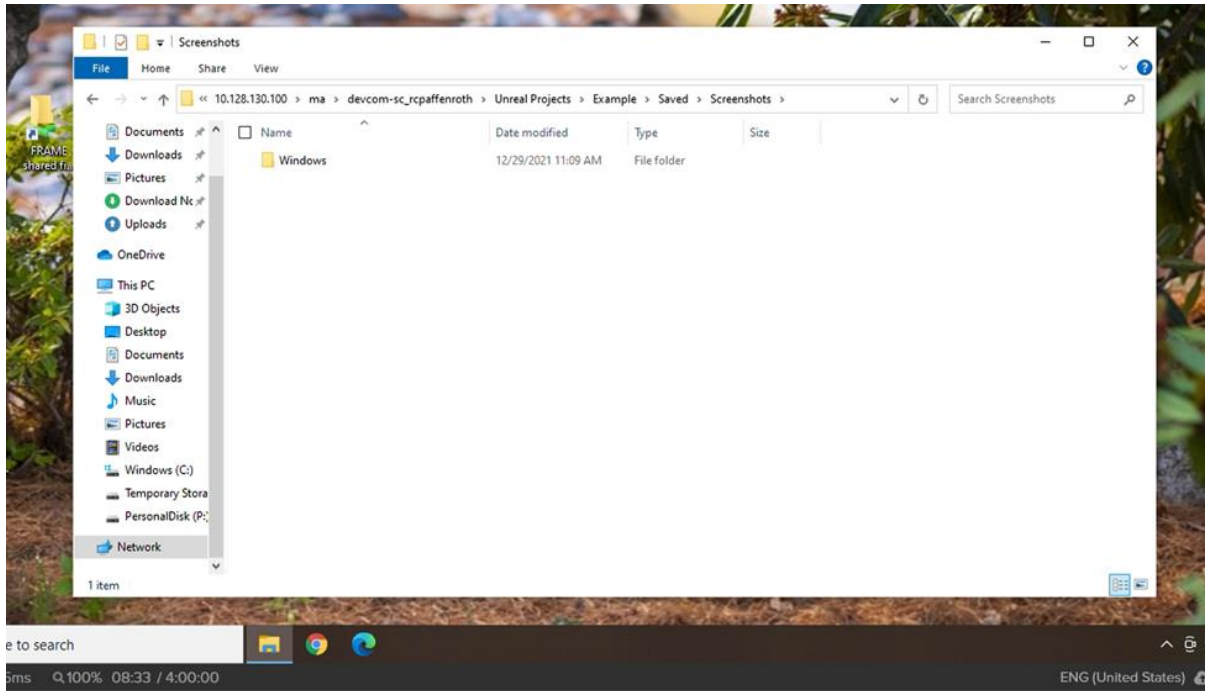


Click **Save** and **Compile** in the top right corner.

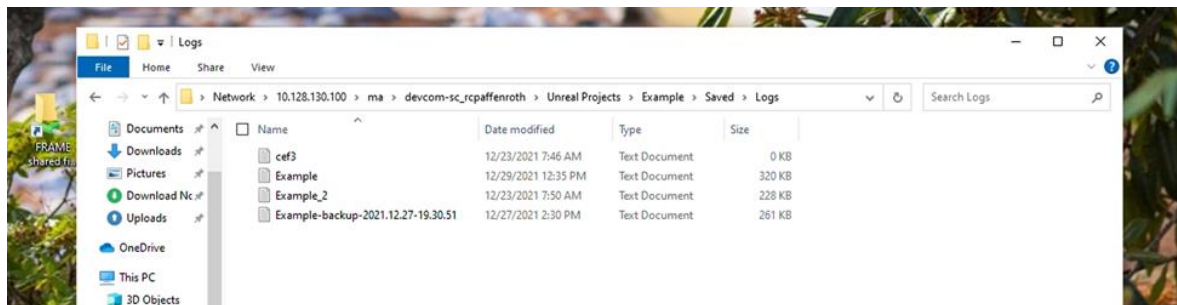
Data Organization

Organizing the data from the drops requires two pieces of information from the project file.

First you need to locate the folder where the screenshots were saved. This will be found at <yourProject>/Saved/Screenshots/Windows. This entire Windows folder can be moved or copied, since we will run a program that sorts through the whole folder.

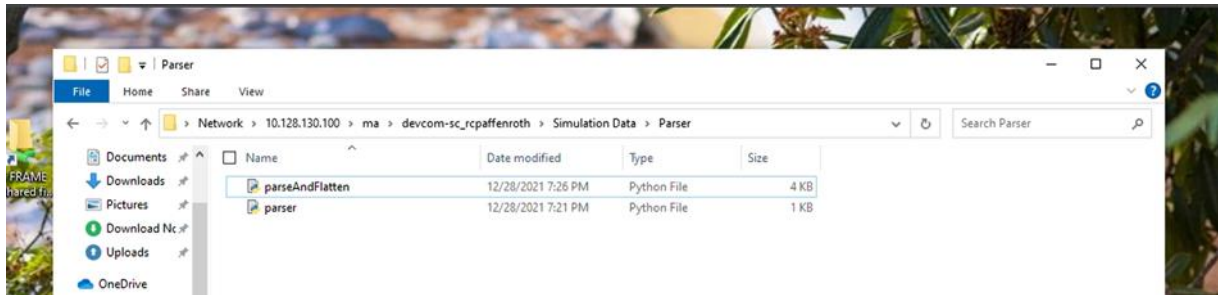


Next you will need to find the log file, which is where the location data for the images is saved. This should be saved in the folder <yourProject>/Saved/Logs. Sometimes multiple logs will be produced for a single run of the project, so to quickly check you have the correct one, within a log file you can search for 'x=' (by using the ctrl-f keyboard shortcut) and check that there are instances of that written into the log.



We advise you to make a new folder somewhere with ample space and move both the log file and the Windows folder containing the screenshots there.

Now, to parse the locations from the log, use the parser code we made, located on the desktop folder WPI Research Storage, in the Simulation Data/Parser folder.



The folder contains two important python scripts, the first of which is a basic parser, named `parser.py`, that compiles the locations of each picture in order and writes it to a csv file. These match up with each image in the Windows folder, in order from earliest to latest screenshot.

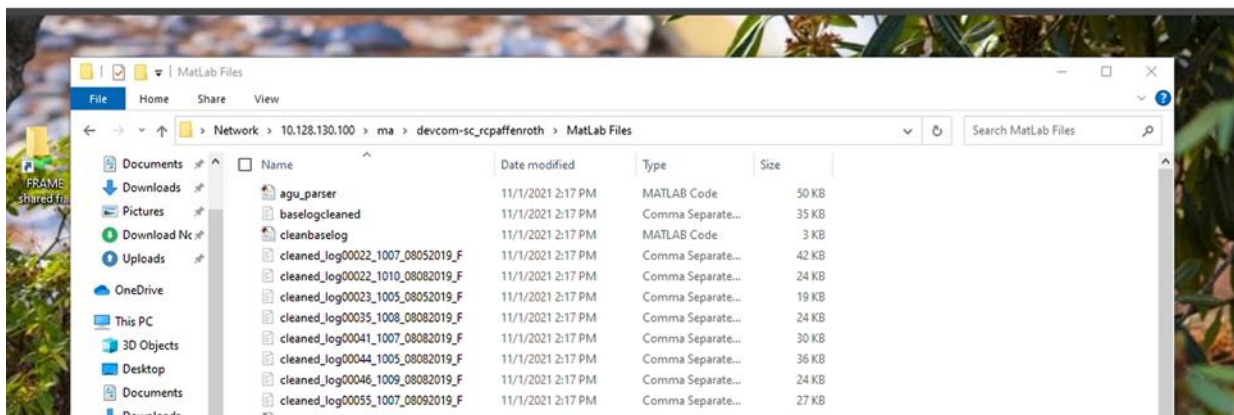
The other file, `parseAndFlatten.py`, also parses the log file and organizes locations, but also matches it to a flattened version of the associated image, along with the flatten first image from the series. It writes this data out as a pickle file, which is like a more efficient CSV file that is unviewable to users.

Before running either of these files, review the file paths written into the program to check they match with your new folder that contains the log and screenshots you want to parse, and the written files are named the way you'd like. Once this is confirmed, run the file.

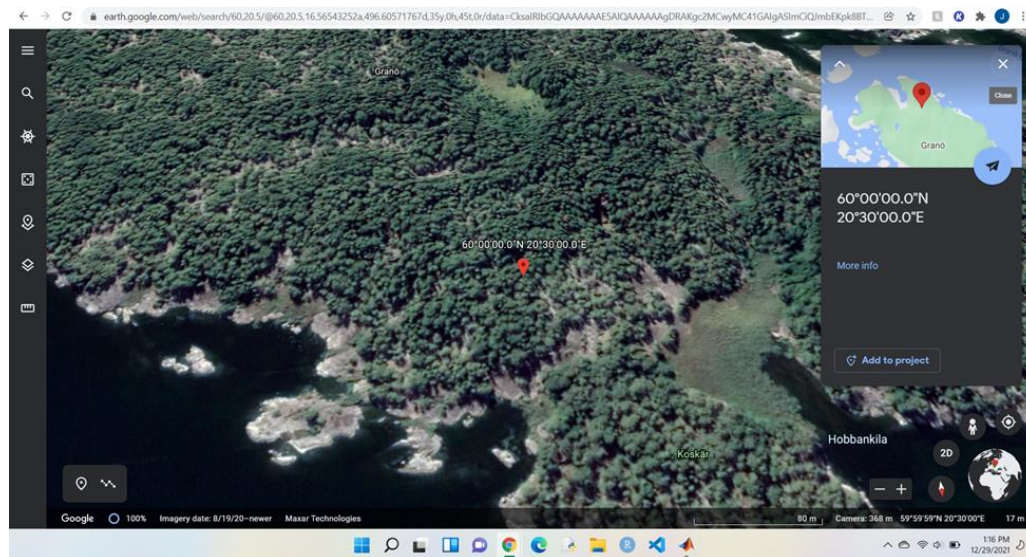
Alternate Drops

We have written code to easily build new CSV files for drops anywhere in the world. The code will follow the path of a real drop, which you specify, and the parachute will land where you choose.

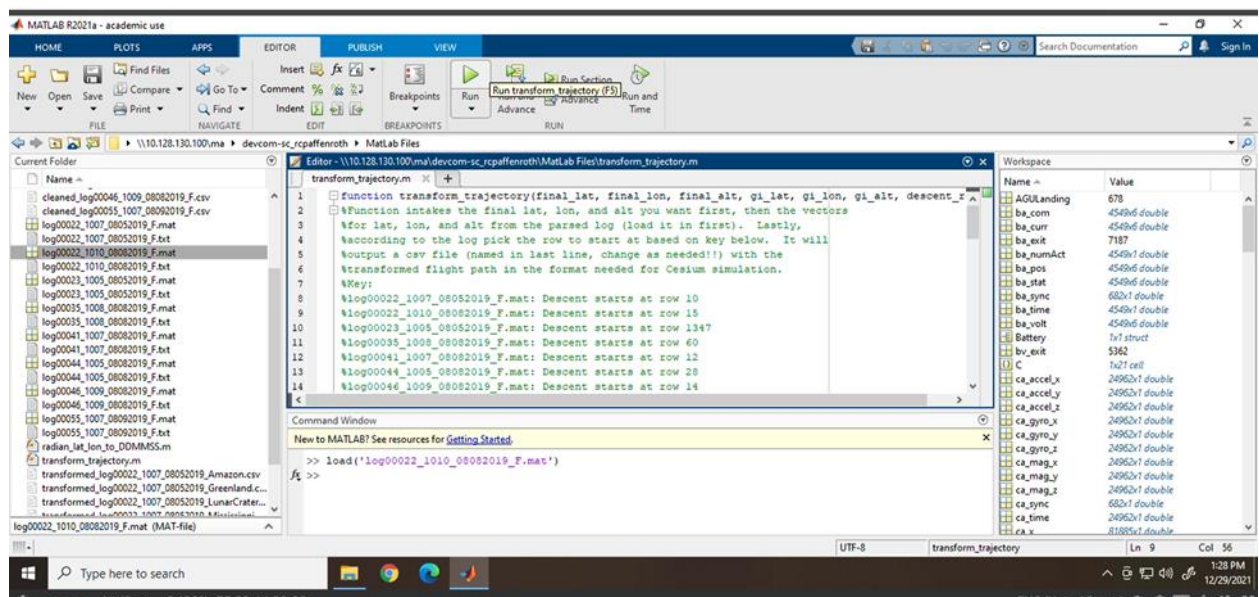
The files you will need to do this are found in the MATLAB Files folder located in the WPI Research Storage folder.



The file `transform_trajectory.m` contains code for creating data for a drop file based on a real drop path. The output is a CSV file compatible with our Unreal Engine simulator. The functions first three inputs are the final latitude, final longitude, and final altitude you want. These values can be found easily using Google Earth; your cursor's location is given in the bottom right-hand corner of the window. For example, if I'd like my drop to land in Sweden at (60, 20.5), I can type that into Google Earth, hover over the pin, and find the height of the ground there, 17 meters. You can also just scan the map for a location and similarly use your cursor to find a landing point.



Next, you'll need to know which drop path you'd like to follow, since the next three inputs are then the vectors for latitude, longitude, and altitude from the parsed drop log. These are easily available to the function by simply loading the desired file first, then using the given names of `gi_lat`, `gi_lon`, and `gi_alt` when you run the function. Eight pre-parsed logs are provided in the MATLAB Files Folder already, identified by their ".mat" ending.



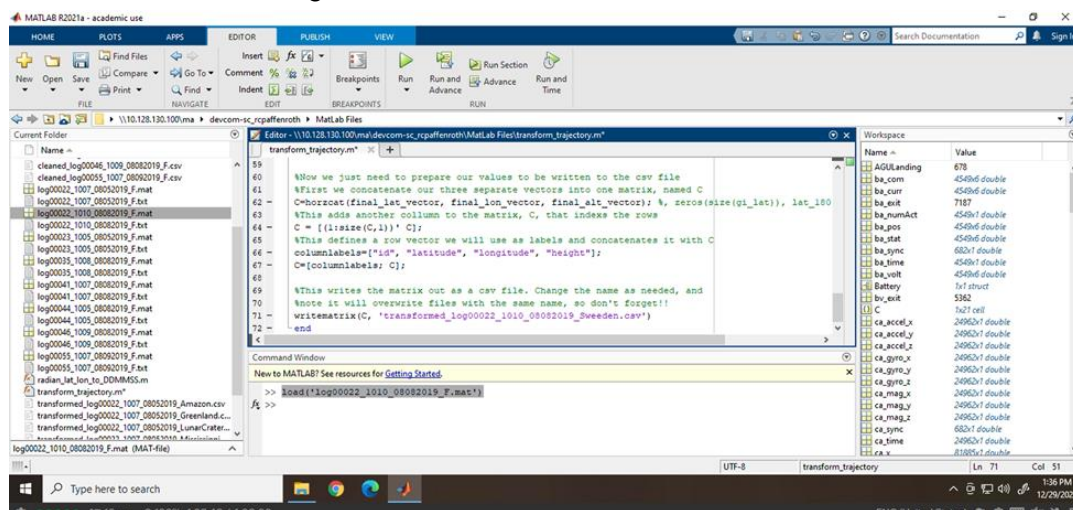
So, if you wanted to follow the trajectory of the drop log00022_1010_08082019_F, you would load that file either by double clicking the .mat file in the left pane or by running the line "load('log00022_1010_08082019_F.mat')", as shown above.

The last input for the function is the row to start at based on the key below. This simply discard values that were recorded while the parachute package was still in the plane:

Key:	
log00022_1007_08052019_F.mat:	Descent starts at row 10
log00022_1010_08082019_F.mat:	Descent starts at row 15
log00023_1005_08052019_F.mat:	Descent starts at row 1347
log00035_1008_08082019_F.mat:	Descent starts at row 60
log00041_1007_08082019_F.mat:	Descent starts at row 12
log00044_1005_08082019_F.mat:	Descent starts at row 28
log00046_1009_08082019_F.mat:	Descent starts at row 14
log00055_1007_08092019_F.mat:	Descent starts at row 19

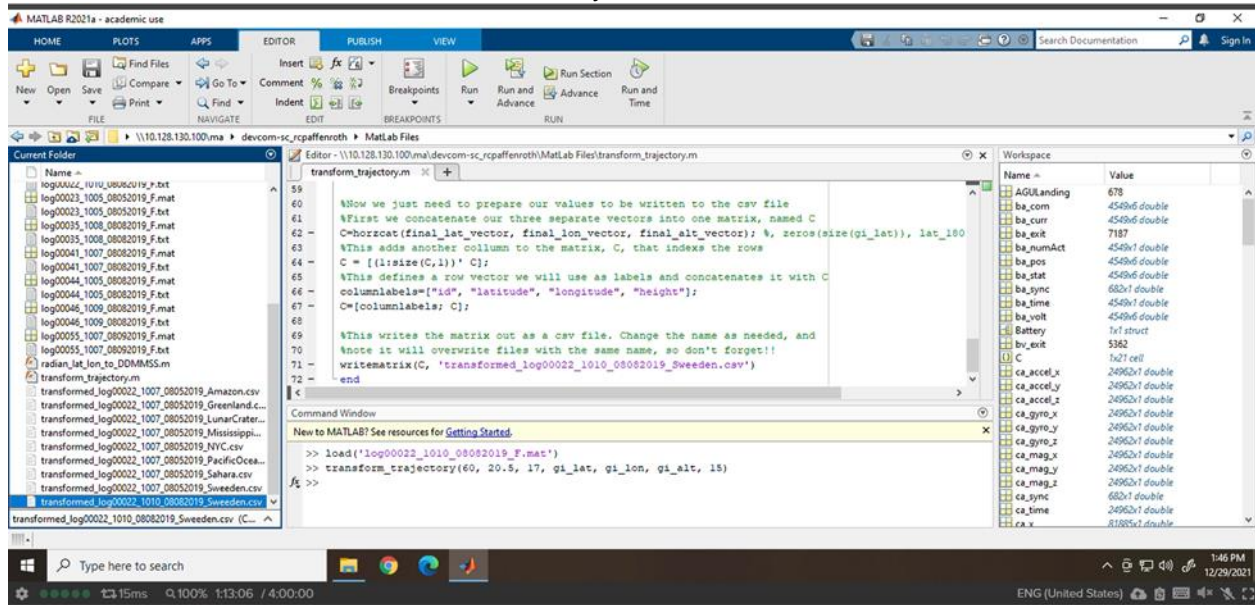
In the case of this example using the path from log00022_1010_08082019_F, you would input 15.

The last thing you need to do before running the file is setting the name of the output csv file you'd like. You can do this in line 71 of the file. You can change the name to anything you'd like, but make sure to maintain the .csv ending.



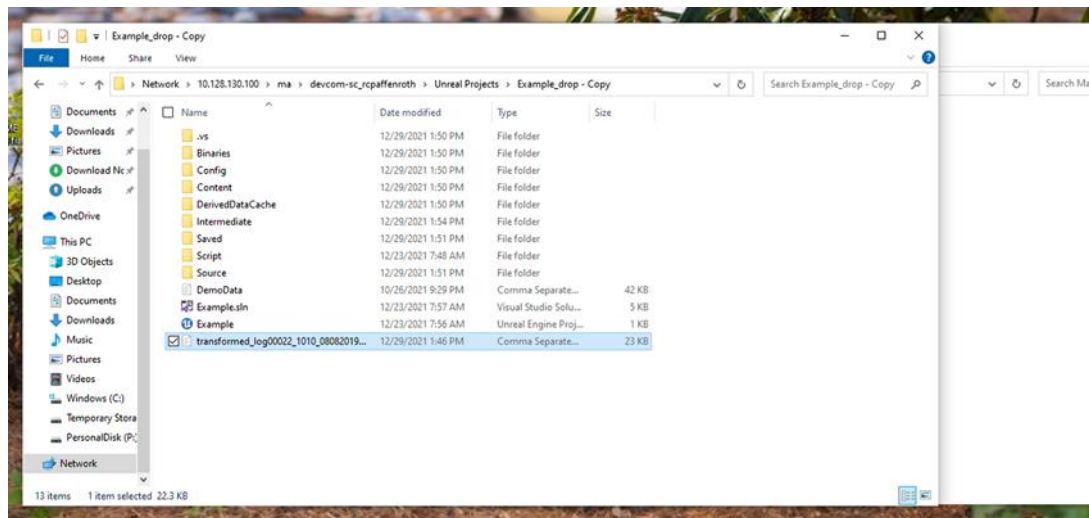
After making this change, make sure you click the **Save** button in the top left of the application.

After making this change, you could run a transform of the drop to the specified location in Sweden by typing the line: `transform_trajectory(60, 20.5, 17, gi_lat, gi_lon, gi_alt, 15)` The desired csv file will be written into the MATLAB Files folder, and you can use it in a simulation.

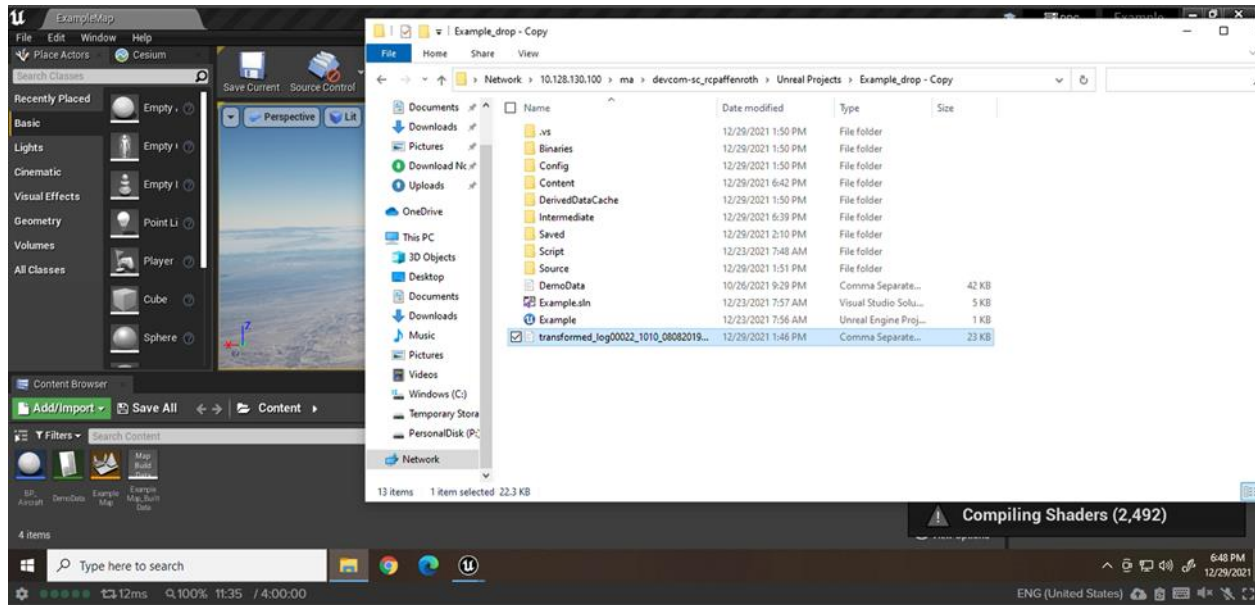


Note: the file `cleanbaselog.m` works very similarly, but instead of transforming the location, it simply cleans the path or a drop and writes it to a csv in the appropriate structure to be used in the simulator.

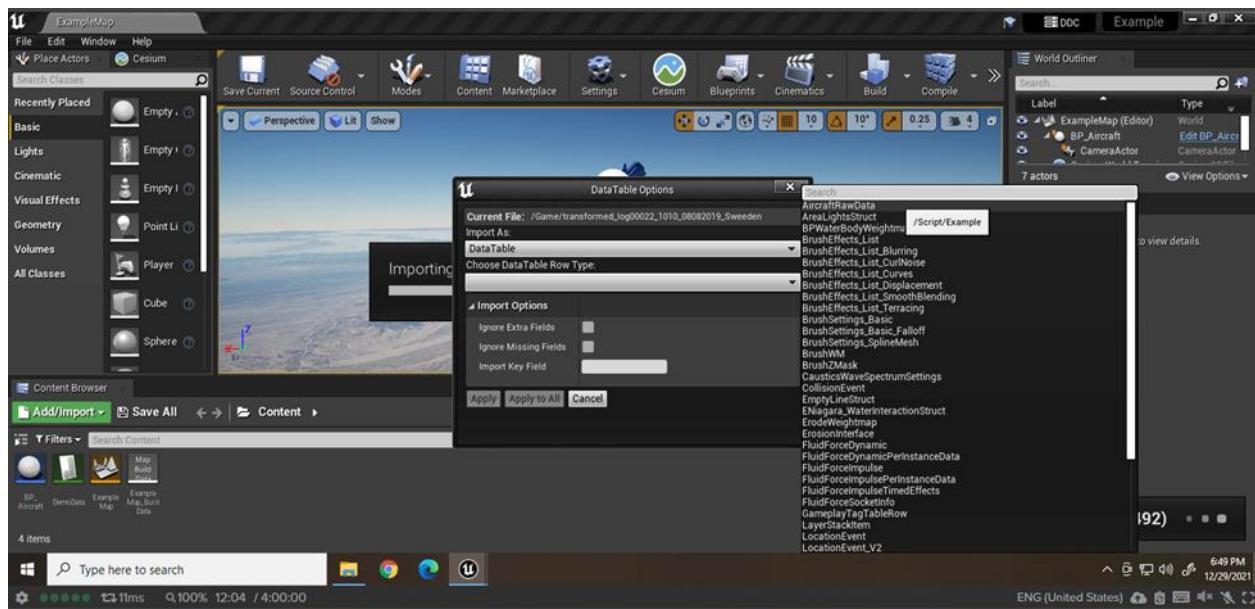
Now we will load the new CSV file into our simulation. First, move the new CSV file to the project folder you're using to make this simulation.



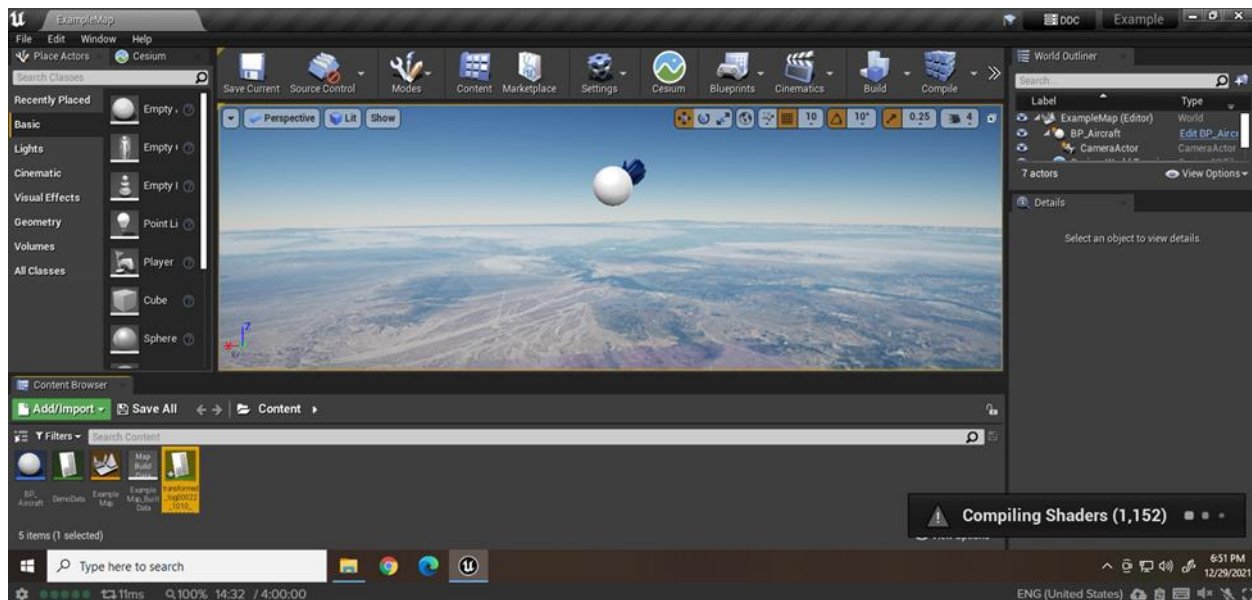
Next you will drag-and-drop the .csv data file into the Unreal Engine Content Browser. This is the panel at the bottom of your editor.



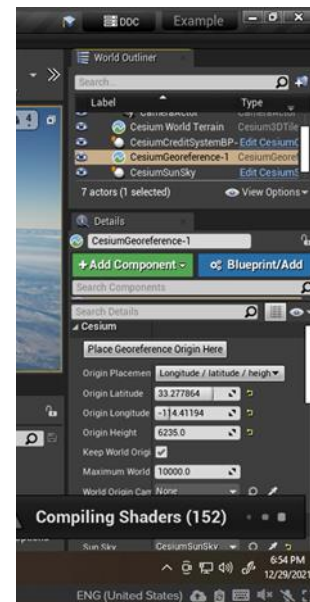
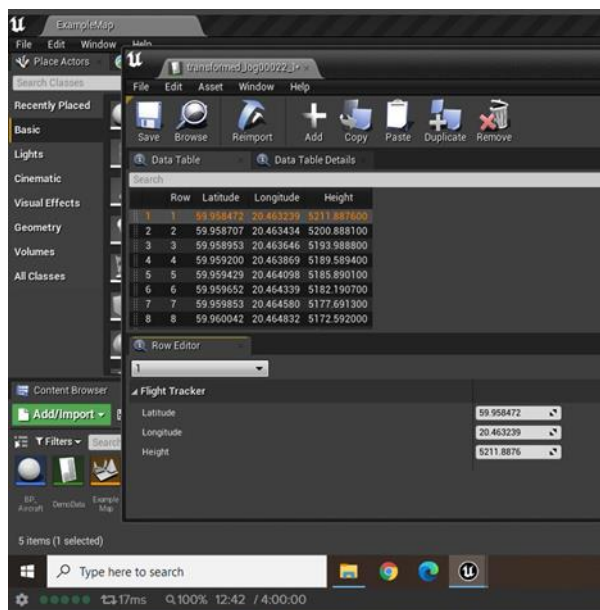
After you drop in the file, a pop-up window will appear. In the **Choose DataTable Row Type** dropdown select **AircraftRowData** (typically the first option):



Click **Apply**. Then to check the data uploaded correctly, double-click on the file in the Content Browser to open the data table:



Note the first line of the new data table, next you will input these values to position your view to the drop location. In the World Outliner on the left, select the CesiumGeoreference object.

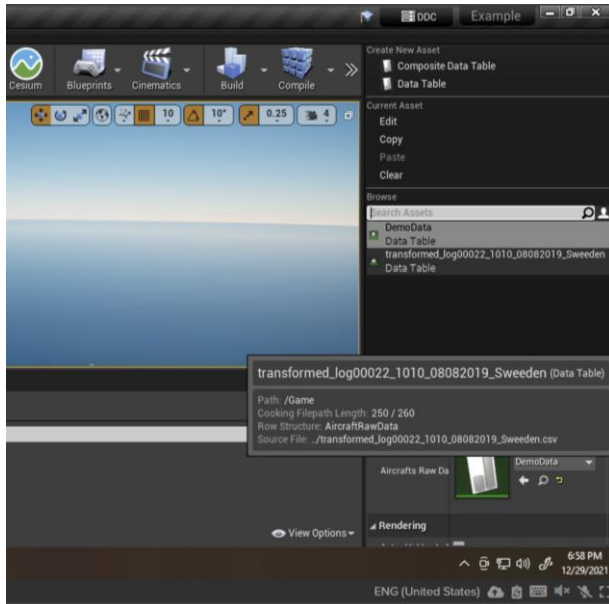


Here you want to change your location in this tab to be the first coordinated in the data file, in this case:

Origin Latitude = 59.958472; Origin Longitude = 20.463239; Origin Height = 5211.887600

If the scene appears dark, adjust the time of day by selecting the CesiumSunSky actor in the World outliner Panel and adjusting the **Time Zone** slider.

Select the PlaneTrack actor. In the Details panel, then set the Aircrafts Raw Data Table variable to the data table you added.



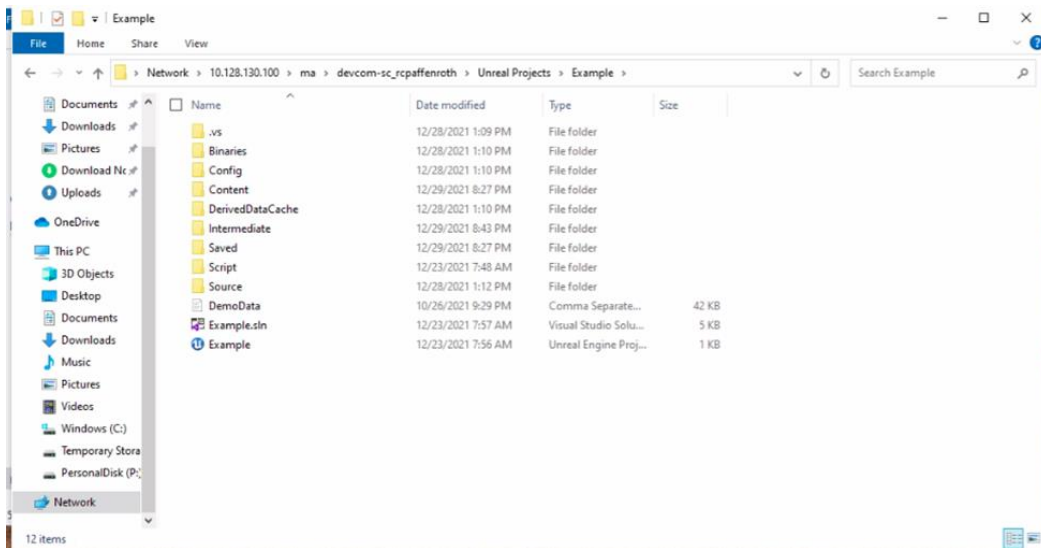
Finally, **Save** the project. Now you will have a simulated drop in the new location according to your CSV file.

User Guide

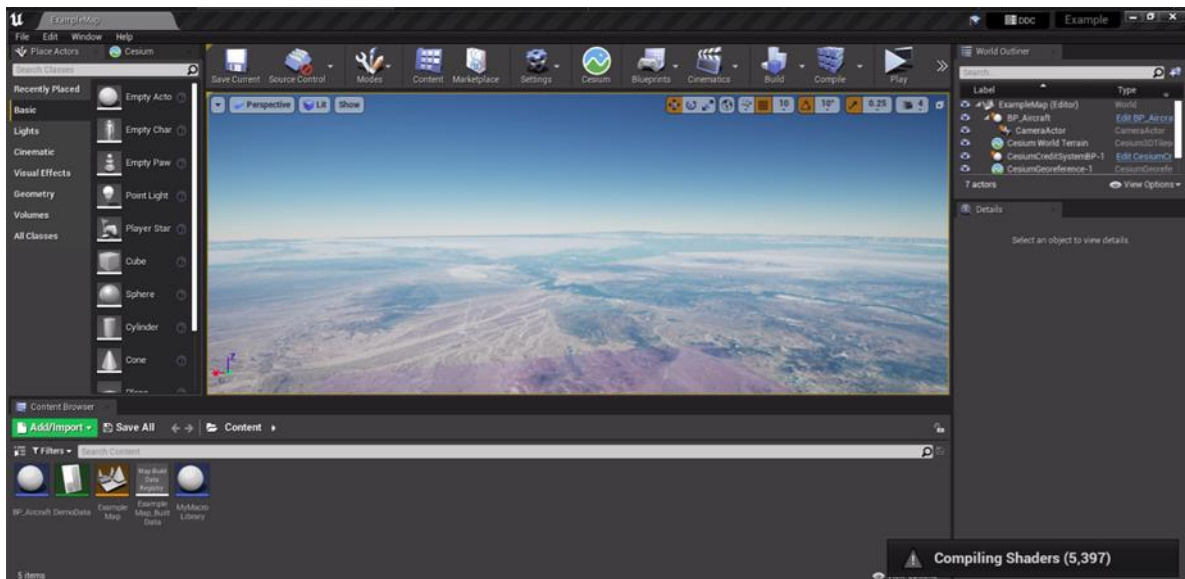
These are instructions on how to open and run a project, load in flight files, and collect data. We will be using our project named 'Example and load in a new location, fly the drone, and collect data.

Open and Load Files

Each Unreal Engine project is saved with its own folder. Open the project folder and double click on the <project_name>, in this case named Example. The file type will be Unreal Engine Project. It will take a couple minutes to load the project.

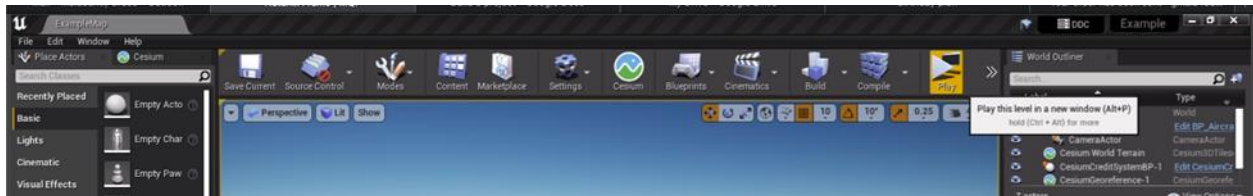


Once the project opens it will look like this:

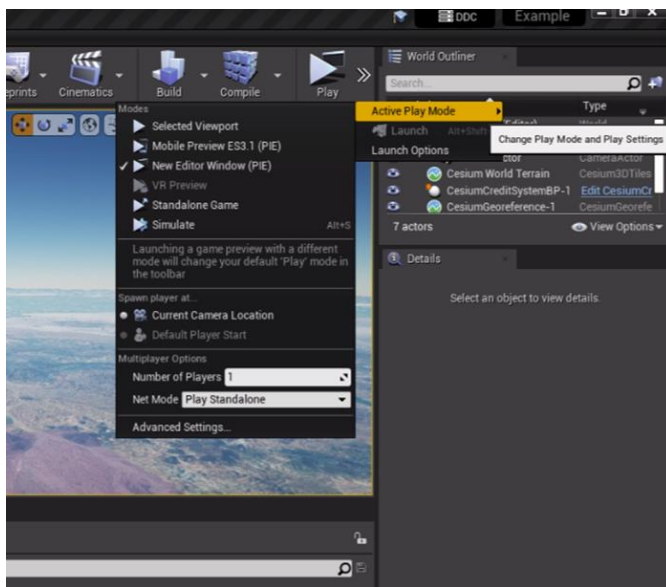


Note: The 'Compiling Shaders' message in the bottom right is letting the user know that the editor is loading in the scenery and lighting. For best performance we recommend waiting for all the shaders to compile before playing the project. This can take several minutes.

Now we'll play our project. Our premade projects are preloaded with different drop patterns to follow. If you would like to change the drop file referend XYZ in the manual. In the top bar there are several icons. Locate the **Play** button. If you cannot see it, you may need to increase the size of the window.



There are a variety of ways to play and display the simulation which can be found under **Active Play Mode**. We recommend playing in the **New Editor Window (PIE)**, click on it to start playing the project.

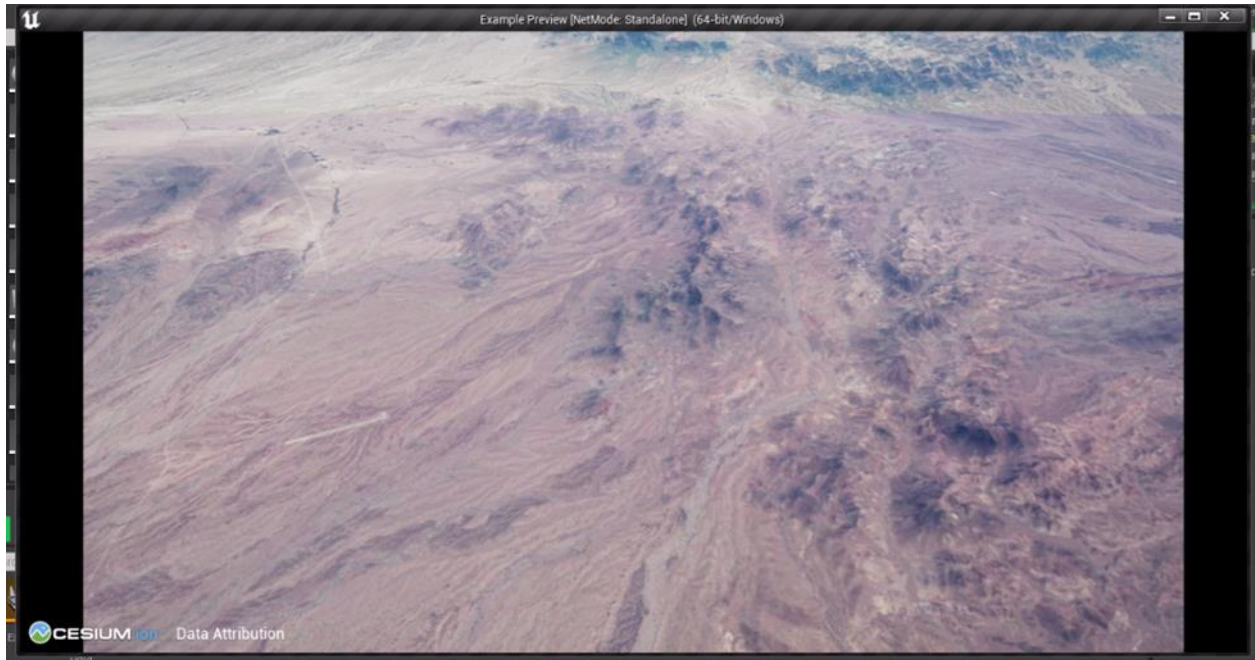


A new window will appear running the project. You can click anywhere in the window to begin mouse control. You can move your mouse around to look around in the simulated world. The view is a general viewport view.

Flying and Collecting Data

There are several built in key triggers to control the drone in the simulation:

L - Mounts the camera to the drone and changes the view to the drone perspective, once in this perspective you can no longer control the view with your mouse. The view will face downwards as seen below.



M - Begins the flight

C - Manually capture a screenshot and location, can be clicked multiple times

F - Begins an autonomous collect of screenshots and locations that will run during the whole flight

To start the flight first click L to change perspective then M to begin moving. Then you can click C or F to collect data depending on how you would like to collect it. Drops typically last about 20 minutes.

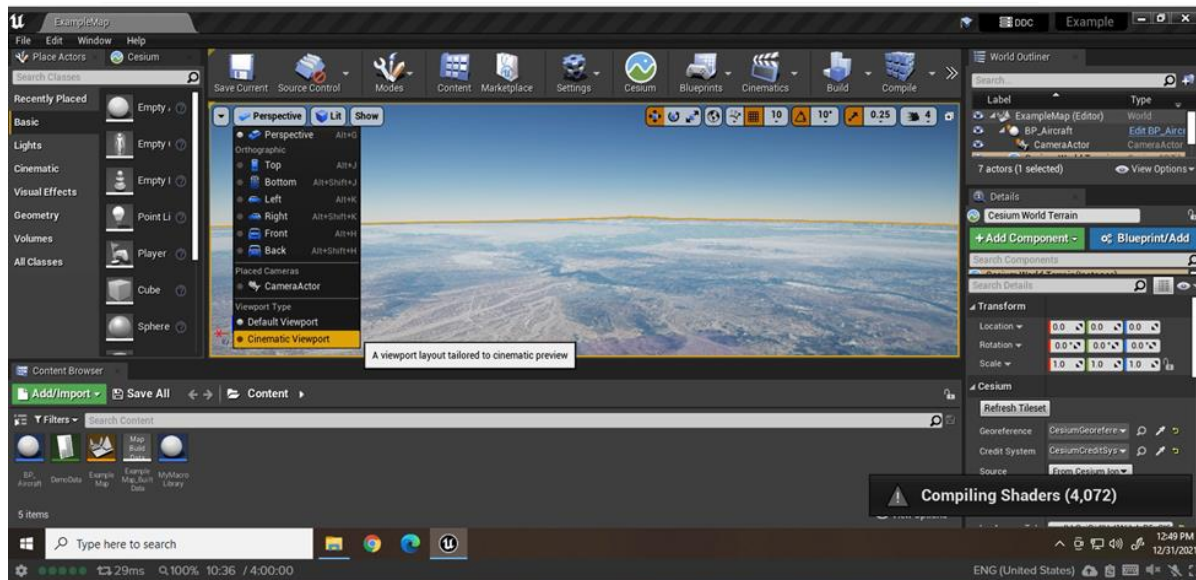
Accessing Data

Once the drop has completed you can close the window and all the data collected will be saved. To access the screenshots collected navigate back to Project Folder -> Saved -> Screenshots -> Windows. To access the log files containing the locations navigate to Project Folder -> Saved -> Logs.

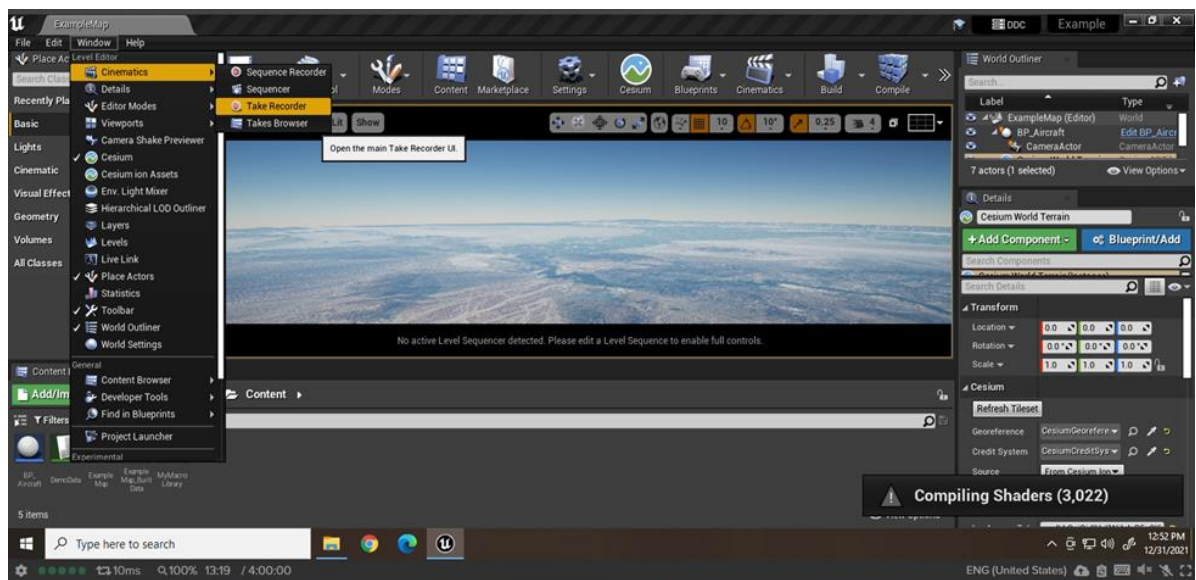
Recording a Drop Video

To record a video of a drop, the first step is to run the drop once and record how long the simulation takes. This is a key step, since in the recording step you will not be able to observe the movement while you record. In this example, I will be working with a project where I know I want to record thirty seconds of flight.

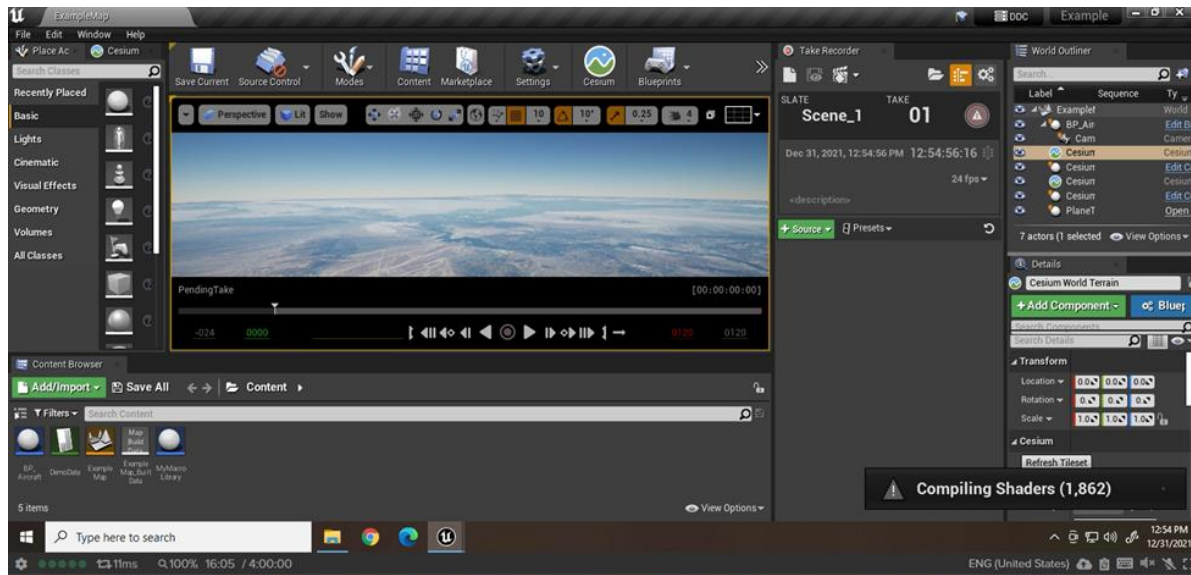
First, in the top left corner of the viewport select Perspective → Cinematic Viewport.



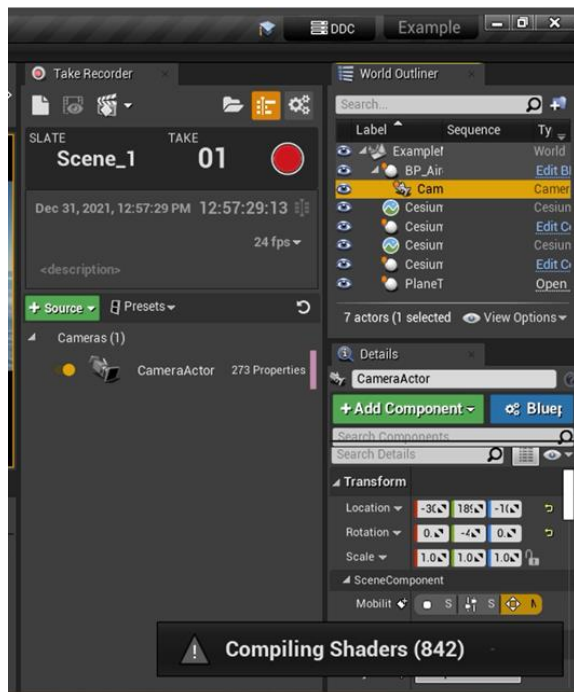
Next, bring up the recording panel by going to the top left corner of the Unreal Editor and selecting Window → Cinematics → Take Recorder.



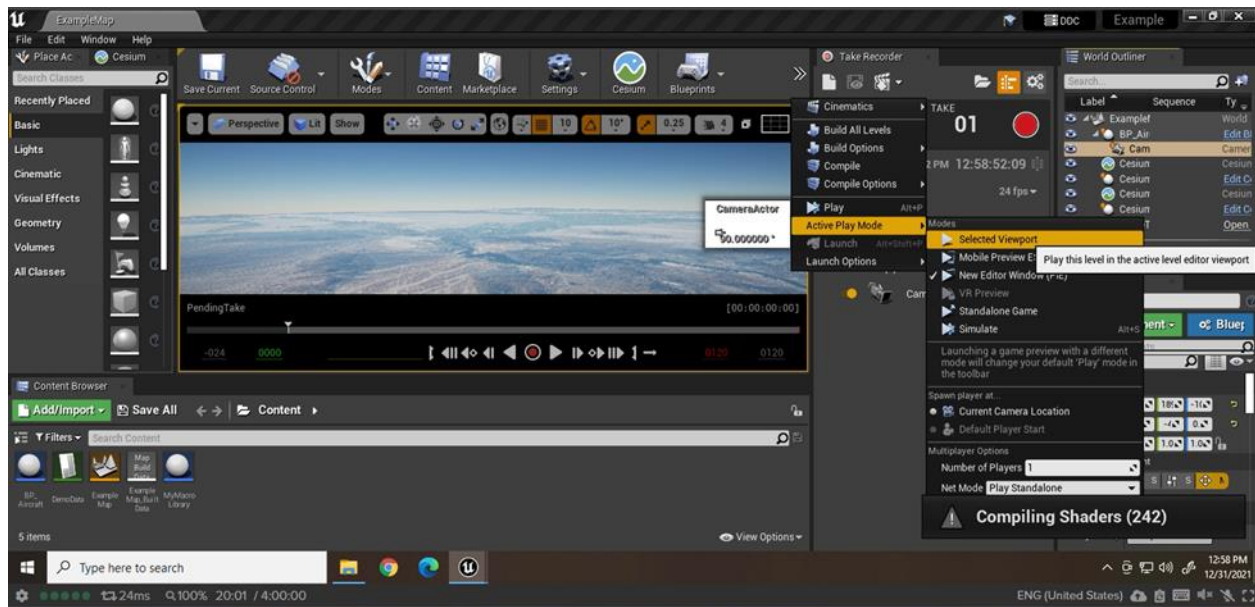
This will open a popup window, which you can minimize, and underneath there will be panel at the left, titled take recorder.



Click and drag your CameraActor from the World Outliner into the bottom section of the take recorder panel. This will tell the program which perspective to record from.

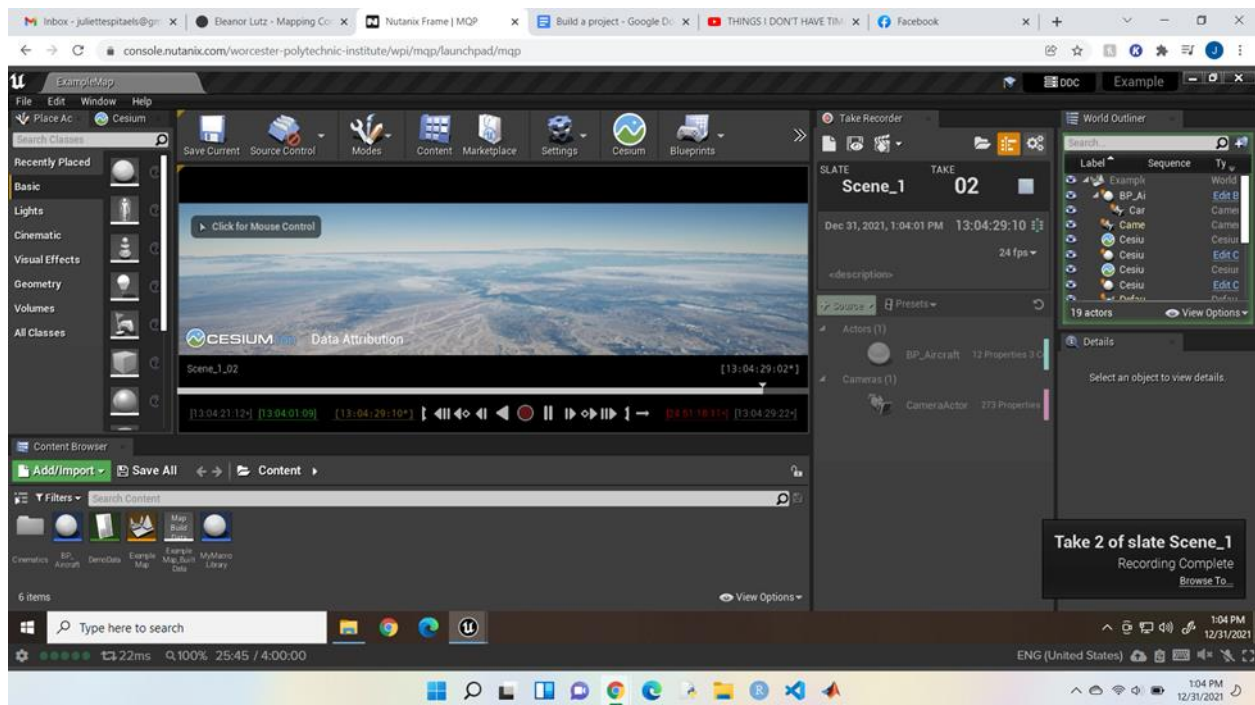


Now play the project, making sure you're using "selected viewport" Active Play Mode.



Now you can press the large red record button in the Take Recorder panel. This brings back up the popup, which again you can minimize, and below will be a three second countdown.

To start the movement of the parachute, click into the viewport and press the M trigger key. **Note: Make sure you do not press L to mount the camera, or else it will not record properly.**

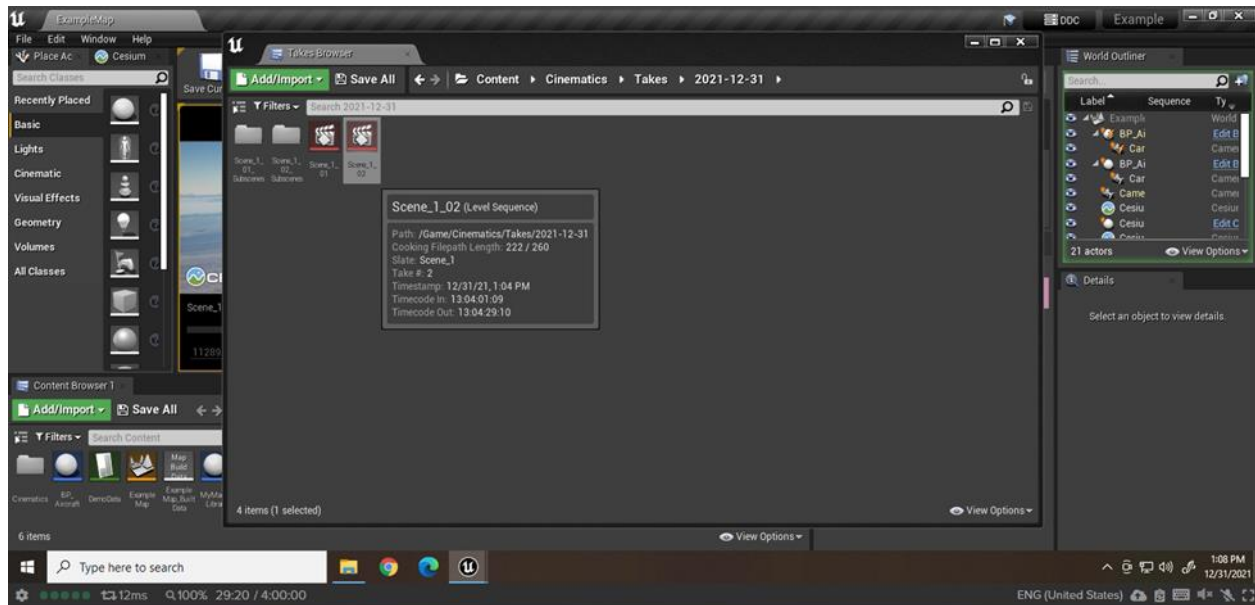


The screenshot shows the 'Take Recorder' menu in the Unreal Engine interface. The menu is open, displaying options such as 'Cinematics', 'Build All Levels', 'Build Options', 'Compile', 'Compile Options', 'Pause', 'Stop' (highlighted in yellow), and 'Possess or Eject Player'. The 'Stop' option is highlighted in yellow, and the 'Possess or Eject Player' option is visible below it. The background shows the Unreal Engine interface with a 'Take Recorder' button and a 'Camera' button.

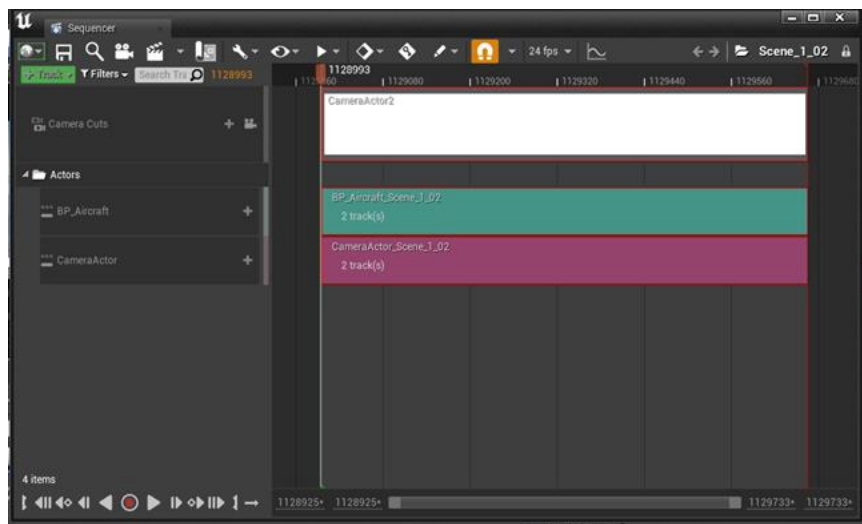
The screenshot displays the Unreal Engine interface with the following elements:

- Take Recorder (Top Left):** Shows 'Scene_1' at '03' with a red recording indicator. The time is 'Dec 31, 2021, 1:06:52 PM' and '13:06:52:00'. The frame rate is '24 fps'. There is a description field containing '<description>'. Below are buttons for '+ Source', 'Presets', and a refresh icon.
- World Outliner (Top Right):** Contains a search bar and a list of objects. A tooltip 'Show/Hide the Takes Browser' points to a button above the list. The list includes 'Example', 'BP_Ai', 'Car', 'Came', 'Cesium', 'Cesium', 'Cesium', 'Cesium', 'Cesium', and 'Default'. On the right, there are links for 'World', 'Edit B', 'Camer', 'Camer', 'Cesium', 'Edit C', 'Cesium', 'Edit C', and 'Default'.
- Cameras (Bottom Left):** Shows 'Cameras (1)' with a list containing 'CameraActor' with '273 Properties'.
- Details (Bottom Right):** A panel titled 'Details' with a tooltip 'Select an object to view details.'

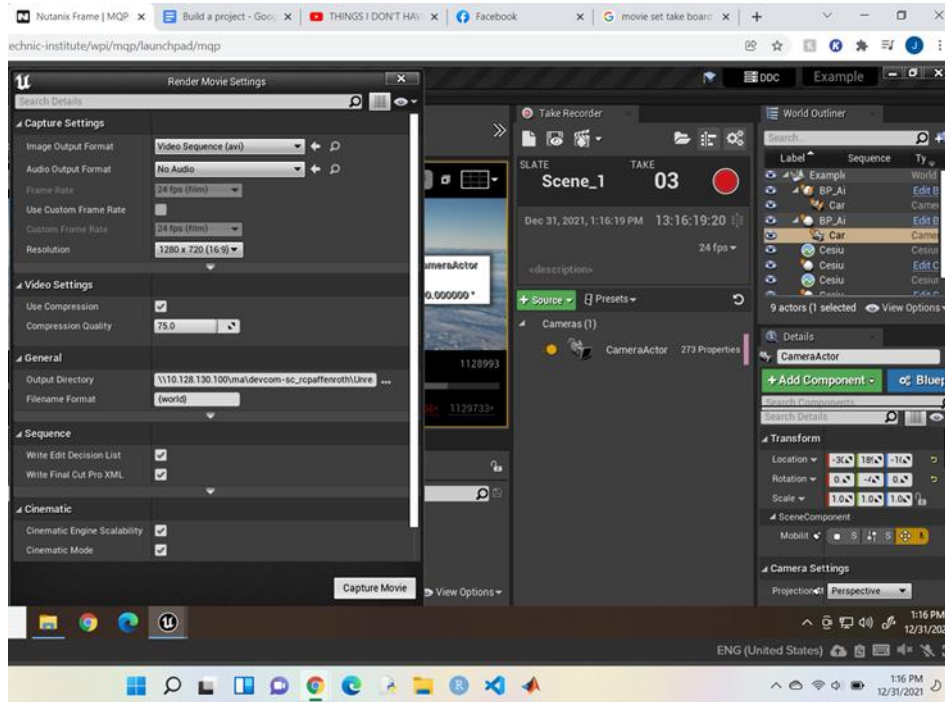
Select the take you recorded, from the popup. Note: I ran my recording twice, which is why I have two different takes to choose from. The takes are numbered with the most recent recording having the highest number.



Selecting your take will open another popup window, titled Sequencer. In this window you will select the render to video button on the top left, which is the button that looks like the clapperboard.



This will open yet another window, where you can review the settings used for the recording. **If your shaders are not finished compiling for the project, WAIT. Continuing past this step without compiled shaders often leads to the engine crashing and not saving your takes.** If your shaders are compiled, at the bottom left of this window click the gray Capture Movie button. If you're prompted to save your project at this step, do so.



Completing the previous step will cause a preview window to open somewhere on your screen.

Once your preview disappears, it means your video has finished rendering. To view it, go into <your project folder → Saved → VideoCaptures. There you will see your video saved according to your level name. If you render multiple videos, numbers will append to the end to differentiate the files.

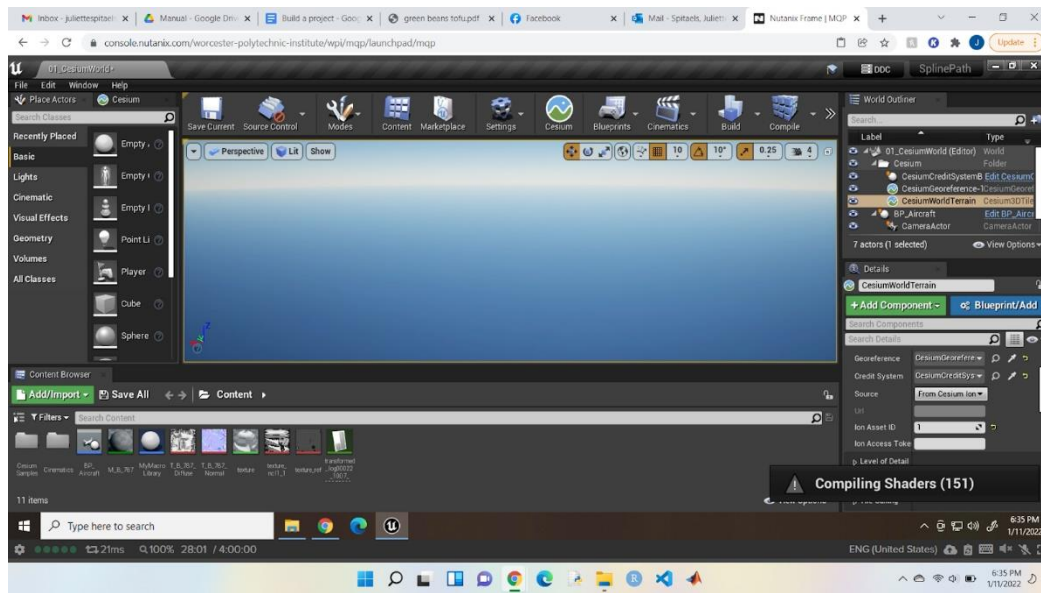
To check your video saved properly, click it to view.

It's a good idea to **Save** your project after this step. Now you can record more videos or close the Unreal Editor if you're finished.

Troubleshooting

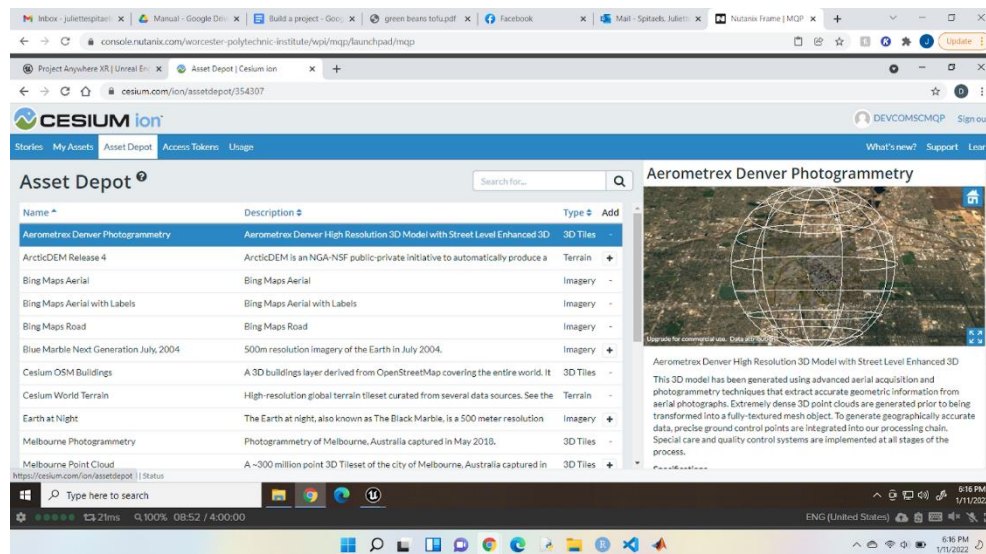
Cesium World Disappears

At times you may encounter an issue with tokens, which are unique identifiers for access to streaming Cesium imaging. If your globe ever appears as a blank blue ball, these steps can be followed to fix the issue in affected projects.

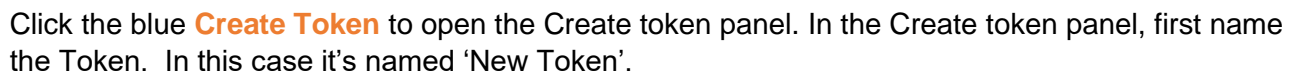


For further detail, see tutorial: docs.unrealengine.com/4.27/en-US/Resources/Showcases/ProjectAnywhereXR/#projectsetup

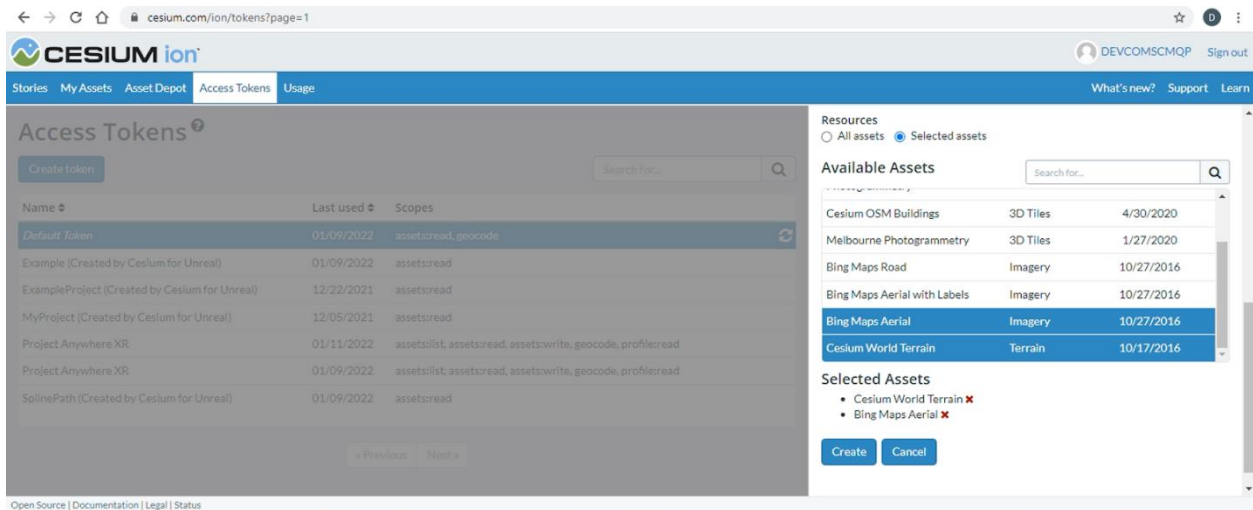
Log in to your Cesium Ion account and go to the Asset Depot tab, in the top left of the window.



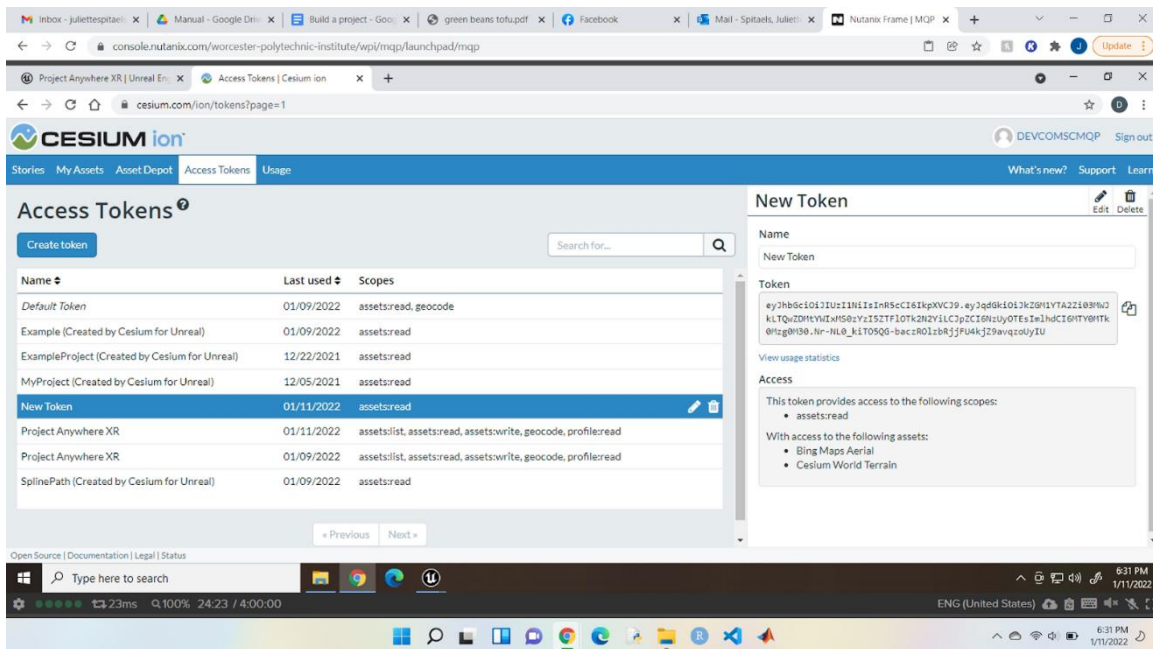
Go to the **Access Tokens** tab, also located in the top left of the window.



Under Resources, select the radio button for Selected assets to show the list of Available Assets. In the Available Assets list, select the following assets: Bing Maps Aerial; Cesium world terrain

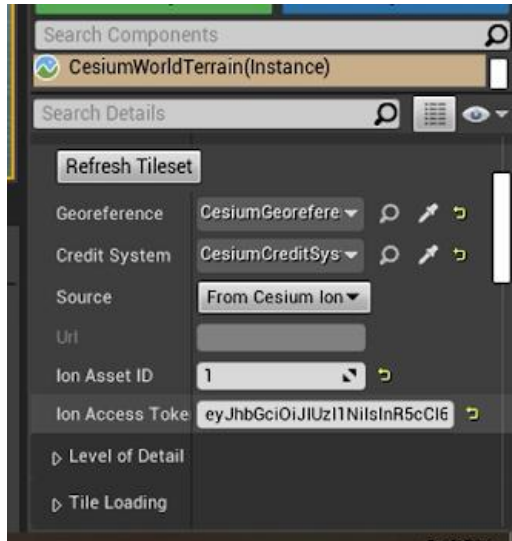


Click the blue **Create** button to create the token. Now select your new token from the list of tokens, it should look like:

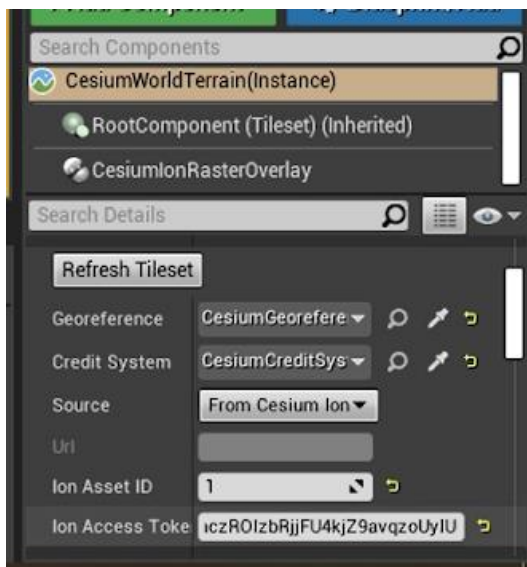


Copy the text under Token by clicking the copy icon at the left of the code. Now open your desired project. In the Unreal Editor's World Outliner, select the following Cesium World Terrain Actor.

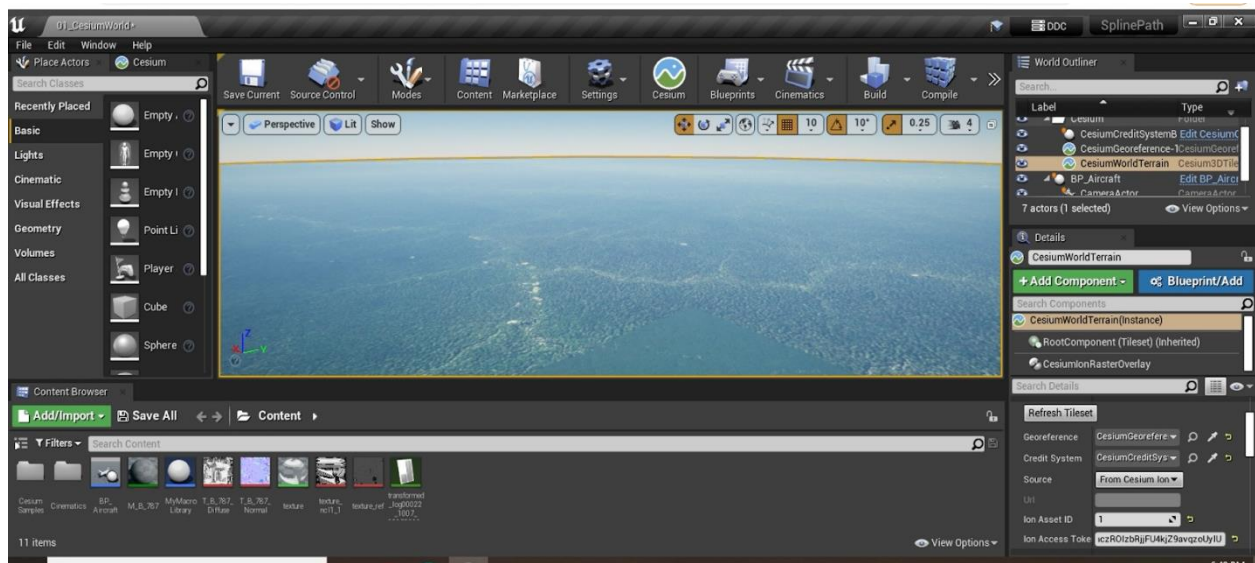
In the Details panel find the Ion Access Token property. Delete the existing token and replace it with the new token you generated.



Next, click and drag to expand the components panel, which is just above the details search bar. Select the 'CesiumRasterOverlay' from the list. In the panel, find the 'Ion Access Token' property. Again, delete the existing token and replace it with the new token you generated.



Now your globe should show land masses and other features again.



To finish, press the Save button in the Unreal Engine toolbar.

Frequently Asked Questions

-

Bibliography

- [1] C. Dever, L. Hamilton, R. Truax, L. Wholey, K. Bergeron, and G. Noetscher, “Guided-Airdrop Vision-Based Navigation,” in *24th AIAA Aerodynamic Decelerator Systems Technology Conference*. DOI: 10.2514/6.2017-3723. [Online]. Available: <https://arc.aiaa.org/doi/abs/10.2514/6.2017-3723>.
- [2] *Satellite Navigation - GPS - How It Works — Federal Aviation Administration*. [Online]. Available: https://www.faa.gov/about/office_org/headquarters_offices/ato/service_units/techops/navservices/gnss/gps/howitworks (visited on 03/16/2022).
- [3] M. I. Jordan and T. M. Mitchell, “Machine learning: Trends, perspectives, and prospects,” *Science*, vol. 349, no. 6245, pp. 255–260, 2015. DOI: 10.1126/science.aaa8415. eprint: <https://www.science.org/doi/pdf/10.1126/science.aaa8415>. [Online]. Available: <https://www.science.org/doi/abs/10.1126/science.aaa8415>.
- [4] M. A. Nielsen, “Neural Networks and Deep Learning,” en, 2015. [Online]. Available: <http://neuralnetworksanddeeplearning.com> (visited on 03/01/2022).
- [5] Z. R. Fitzgibbon, R. S. Veetekat, and K. J. Fabrizio, “Feature Recognition from Aerial Imaging,” Worcester Polytechnic Institute, Tech. Rep., 2021.
- [6] Cesium, *Cesium for Unreal – Cesium*. [Online]. Available: <https://cesium.com/platform/cesium-for-unreal/> (visited on 02/09/2022).
- [7] R. Caruana, “Multitask Learning,” en, *Machine Learning*, vol. 28, no. 1, pp. 41–75, Jul. 1997, ISSN: 1573-0565. DOI: 10.1023/A:1007379606734. [Online]. Available: <https://doi.org/10.1023/A:1007379606734> (visited on 04/20/2022).
- [8] S. Khan, M. H. Javed, E. Ahmed, S. A. A. Shah, and S. U. Ali, “Facial Recognition using Convolutional Neural Networks and Implementation on Smart Glasses,” in *2019 International Conference on Information Science and Communication Technology (ICISCT)*, Mar. 2019, pp. 1–6. DOI: 10.1109/CISCT.2019.8777442.
- [9] *Artificial Intelligence & Autopilot*, en-us. [Online]. Available: <https://www.tesla.com/AI> (visited on 03/16/2022).
- [10] G. Noetscher, personal communication, Sep. 2021.
- [11] G. Noetscher, personal communication, Apr. 8, 2022.

- [12] J. V. Iyengar, G. H. Hargett, and J. D. Hargett, “Military development and applications of simulation systems,” en, p. 13, 1999.
- [13] H. c. Editors, *First parachute jump is made over Paris*, en. [Online]. Available: <https://www.history.com/this-day-in-history/the-first-parachutist> (visited on 04/13/2022).
- [14] *Aerial Delivery of Supplies*. [Online]. Available: https://qmmuseum.lee.army.mil/wwii/aerial_supplies.htm (visited on 04/13/2022).
- [15] *Top parachute facts — National Army Museum*, en. [Online]. Available: <https://www.nam.ac.uk/explore/top-parachute-facts> (visited on 04/13/2022).
- [16] *U.S. Army Combat Capabilities Development Command*, en. [Online]. Available: <https://www.army.mil/devcom> (visited on 04/13/2022).
- [17] *DEVCOM Soldier Center*, en. [Online]. Available: https://www.army.mil/article/157840/devcom_soldier_center (visited on 04/13/2022).
- [18] *High-performance ram-air canopy with improved direction*, en, Section: technology, May 2020. [Online]. Available: <https://techlinkcenter.org/technologies/high-performance-ram-air-canopy-with-improved-directional-steering/39231617-4bd6-4743-9b2c-af872f458d61> (visited on 04/13/2022).
- [19] *Draper Advances Capabilities of the Military’s Autonomous Airdrop System to Enable Operations in GPS Denied Environments*, en. [Online]. Available: <https://www.draper.com/news-releases/draper-advances-capabilities-militarys-autonomous-airdrop-system-enable-operations> (visited on 04/19/2022).
- [20] J. Skillings, *GPS at risk: Those signals are more vulnerable than you realize*, en. [Online]. Available: <https://www.cnet.com/science/gps-at-risk-those-signals-are-more-vulnerable-than-you-realize/> (visited on 04/13/2022).
- [21] *Protecting GPS From Spoofers Is Critical to the Future of Navigation*, en, Section: Telecommunications, Jul. 2016. [Online]. Available: <https://spectrum.ieee.org/gps-spoofing> (visited on 04/13/2022).
- [22] *Cutting Edge Training & Simulation Software Platform - Unreal Engine*. [Online]. Available: <https://www.unrealengine.com/en-US/solutions/simulation> (visited on 04/13/2022).

- [23] X. Chen, M. Wang, and Q. Wu, “Research and development of virtual reality game based on unreal engine 4,” in *2017 4th International Conference on Systems and Informatics (ICSAI)*, Hangzhou: IEEE, Nov. 2017, pp. 1388–1393, ISBN: 978-1-5386-1107-4. DOI: 10.1109/ICSAI.2017.8248503. [Online]. Available: <http://ieeexplore.ieee.org/document/8248503/> (visited on 02/09/2022).
- [24] U. Engine, *An Overview of the Unreal Engine Product - Unreal Engine*. [Online]. Available: <https://www.unrealengine.com/en-US/unreal> (visited on 02/09/2022).
- [25] *Introduction to Blueprints*, en-US. [Online]. Available: <https://docs.unrealengine.com/5.0/en-US/introduction-to-blueprints-visual-scripting-in-unreal-engine/> (visited on 04/13/2022).
- [26] *Cinematics and Sequencer*, en-US. [Online]. Available: <https://docs.unrealengine.com/5.0/en-US/cinematics-and-movie-making-in-unreal-engine/> (visited on 04/13/2022).
- [27] *OpenCV: Basic Operations on Images*. [Online]. Available: https://docs.opencv.org/4.x/d3/df2/tutorial_py_basic_ops.html (visited on 04/19/2022).
- [28] *OpenCV: Histograms - 1 : Find, Plot, Analyze !!!* [Online]. Available: https://docs.opencv.org/4.x/d1/db7/tutorial_py_histogram_begins.html (visited on 04/13/2022).
- [29] *OpenCV: Color conversions*. [Online]. Available: https://docs.opencv.org/3.4/de/d25/imgproc_color_conversions.html (visited on 03/22/2022).
- [30] *OpenCV: Histograms - 2: Histogram Equalization*. [Online]. Available: https://docs.opencv.org/4.x/d5/daf/tutorial_py_histogram_equalization.html (visited on 04/13/2022).
- [31] *OpenCV: Histograms - 2: Histogram Equalization*. [Online]. Available: https://docs.opencv.org/4.x/d5/daf/tutorial_py_histogram_equalization.html (visited on 04/13/2022).
- [32] *OpenCV: Smoothing Images*. [Online]. Available: https://docs.opencv.org/4.x/d4/d13/tutorial_py_filtering.html (visited on 04/14/2022).

- [33] S. L. Brunton and J. N. Kutz, “Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control,” in *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*, Cambridge University Press, 2019, pp. 47–77, ISBN: 978-1-108-42209-3. [Online]. Available: <http://databookuw.com/>.
- [34] *Torch.flatten — PyTorch 1.11.0 documentation*. [Online]. Available: <https://pytorch.org/docs/stable/generated/torch.flatten.html> (visited on 04/19/2022).
- [35] *Machine Learning: What it is and why it matters*, en. [Online]. Available: https://www.sas.com/en_us/insights/analytics/machine-learning.html (visited on 02/09/2022).
- [36] *A Concise History of Neural Networks — by Jaspreet — Towards Data Science*. [Online]. Available: <https://towardsdatascience.com/a-concise-history-of-neural-networks-2070655d3fec> (visited on 03/01/2022).
- [37] J. Delua, *Supervised vs. Unsupervised Learning: What’s the Difference?* — IBM, Mar. 2021. [Online]. Available: <https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning> (visited on 02/09/2022).
- [38] S. R., “Comparison of Supervised and Unsupervised Learning Algorithms for Pattern Classification,” *International Journal of Advanced Research in Artificial Intelligence*, vol. 2, pp. 34, 35, 2013. DOI: 10.1.1.278.5274. [Online]. Available: <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.278.5274&rep=rep1&type=pdf#page=41>.
- [39] G. M. Fitzmaurice, “Regression,” en, *Diagnostic Histopathology*, vol. 22, no. 7, pp. 271–278, Jul. 2016, ISSN: 17562317. DOI: 10.1016/j.mpdhp.2016.06.004. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S1756231716300627> (visited on 02/09/2022).
- [40] *Regression vs Classification in Machine Learning - Javatpoint*, en. [Online]. Available: <https://www.javatpoint.com/regression-vs-classification-in-machine-learning> (visited on 04/19/2022).

- [41] G. James, Daniela Witten, Trevor Hastie, and Rob Tibshirani, “An Introduction to Statistical Learning,” in *An Introduction to Statistical Learning with Applications in R*, 2nd ed., Springer, 2017, ISBN: 978-1-4614-7138-7. [Online]. Available: <https://www.statlearning.com/>.
- [42] X. Zhang, D. Chang, W. Qi, and Z. Zhan, “A study on different functionalities and performances among different activation functions across different ANNs for image classification,” *Journal of Physics: Conference Series*, vol. 1732, no. 1, p. 012 026, Jan. 2021. DOI: 10 . 1088/1742-6596/1732/1/012026. [Online]. Available: <https://doi.org/10.1088/1742-6596/1732/1/012026>.
- [43] J. Barry-Straume, A. Tschannen, D. Engels, and E. Fine, “An evaluation of training size impact on validation accuracy for optimized convolutional neural networks,” vol. 1, no. 4, 2018. [Online]. Available: <https://scholar.smu.edu/datasciencereview/vol1/iss4/12/>.
- [44] TseKiChun, *English: Random forest explain*, Nov. 2021. [Online]. Available: https://commons.wikimedia.org/wiki/File:Random_forest_explain.png (visited on 04/27/2022).
- [45] V. Jain, *Everything you need to know about “Activation Functions” in Deep learning models*, en, Dec. 2019. [Online]. Available: <https://towardsdatascience.com/everything-you-need-to-know-about-activation-functions-in-deep-learning-models-84ba9f82c253> (visited on 04/19/2022).
- [46] *Activation Functions Explained - GELU, SELU, ELU, ReLU and more*, en, Aug. 2019. [Online]. Available: <https://mlfromscratch.com/activation-functions-explained/> (visited on 04/19/2022).
- [47] A. L. Maas, A. Y. Hannun, and A. Y. Ng, “Rectifier Nonlinearities Improve Neural Network Acoustic Models,” en, p. 6,
- [48] *ReLU — PyTorch 1.11.0 documentation*. [Online]. Available: <https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html> (visited on 04/19/2022).
- [49] *LeakyReLU — PyTorch 1.11.0 documentation*. [Online]. Available: <https://pytorch.org/docs/stable/generated/torch.nn.LeakyReLU.html> (visited on 04/19/2022).

- [50] D. Hendrycks and K. Gimpel, “Gaussian Error Linear Units (GELUs),” *arXiv:1606.08415 [cs]*, Jul. 2020, arXiv: 1606.08415. [Online]. Available: <http://arxiv.org/abs/1606.08415> (visited on 04/19/2022).
- [51] *GELU — PyTorch 1.11.0 documentation*. [Online]. Available: <https://pytorch.org/docs/stable/generated/torch.nn.GELU.html> (visited on 04/19/2022).
- [52] S. D. Silva, *The Maths behind Back Propagation*, en, Mar. 2020. [Online]. Available: <https://towardsdatascience.com/the-maths-behind-back-propagation-cf6714736abf> (visited on 04/19/2022).
- [53] *Computer Hardware*. [Online]. Available: <https://web.stanford.edu/class/cs101/hardware-1.html> (visited on 04/19/2022).
- [54] “Slurm documentation,” Slurm. (), [Online]. Available: <https://slurm.schedmd.com/documentation.html>.
- [55] *W&B Company*. [Online]. Available: <http://wandb.ai/site/company> (visited on 04/19/2022).
- [56] “Landscape mountains,” Unreal Engine. (), [Online]. Available: <https://www.unrealengine.com/marketplace/en-US/product/landscape-mountains>.
- [57] Microsoft. “Airsim,” AirSim. (), [Online]. Available: <https://microsoft.github.io/AirSim/>.
- [58] “Cesium GeographicProjection,” Cesium. (), [Online]. Available: <https://cesium.com/learn/cesiumjs/ref-doc/GeographicProjection.html>.
- [59] GISGeography. “World geodetic system (WGS84),” GIS Geography. (Jun. 8, 2021), [Online]. Available: <https://gisgeography.com/wgs84-world-geodetic-system/>.
- [60] Amazon. “Our data centers,” Amazon Web Services. (), [Online]. Available: <https://aws.amazon.com/compliance/data-center/data-centers/>.
- [61] Microsoft. “Azure global infrastructure,” Azure. (), [Online]. Available: <https://azure.microsoft.com/en-us/global-infrastructure/>.
- [62] Google. “About google data centers,” Google Data Centers. (), [Online]. Available: <https://www.google.com/about/datacenters/>.

- [63] GoogleColab. “Welcome to colab,” Google Colab. (), [Online]. Available: https://colab.research.google.com/?utm_source=scs-index.
- [64] “VS code remote development,” Visual Studio Code. (), [Online]. Available: <https://code.visualstudio.com/docs/remote/remote-overview>.
- [65] P. Ramachandran, B. Zoph, and Q. V. Le, *Searching for activation functions*, 2017. DOI: 10.48550/ARXIV.1710.05941. [Online]. Available: <https://arxiv.org/abs/1710.05941>.
- [66] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, “Visualizing the loss landscape of neural nets,” in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS’18, Montréal, Canada: Curran Associates Inc., 2018, pp. 6391–6401.
- [67] J. Ratches, R. Vollmerhausen, and R. Driggers, “Target acquisition performance modeling of infrared imaging systems: Past, present, and future,” English (US), *IEEE Sensors Journal*, vol. 1, no. 1, pp. 31–40, 2001, ISSN: 1530-437X. DOI: 10.1109/JSEN.2001.923585.
- [68] *OpenCV: Feature Matching*. [Online]. Available: https://docs.opencv.org/3.4/dc/dc3/tutorial_py_matcher.html (visited on 04/19/2022).
- [69] M. Turk and A. Pentland, “Face recognition using eigenfaces,” in *Proceedings. 1991 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1991, pp. 586–591. DOI: 10.1109/CVPR.1991.139758.