

Project in Computer Science-D7034E

SIMULATION FRAMEWORK FOR EVENT CAMERAS ON AIRSIM

By Sergio Serrano Hernández

serser-1@student.ltu.se

LULEÅ
TEKNISKA
UNIVERSITET

INDEX:

| | |
|--|----------|
| Introduction: | 2 |
| 1. Timeline and milestones: | 3 |
| 1.1. First proposal. Early October: | 3 |
| 1.2. Second proposal. Mid February: | 5 |
| 2. The Final software set up: | 6 |
| 3. Tutorial on how to replicate the work done: | 7 |
| 3.1. Install the main framework: | 7 |
| 3.2. Unreal Engine 4.25: | 8 |
| 3.3. AirSim: | 9 |
| 3.4. Event camera ROS wrapper: | 9 |
| 3.5. Final steps once everything is installed: | 10 |
| 3.6. Dynamic movement of the drone and objects in Unreal Engine: | 16 |

INTRODUCTION:

As of today, we live in a world in which image processing is evolving at such a pace that would have never been possible to even imagine. Algorithms are being developed constantly and every single day improvements are made.

Object detection, path planning, or obstacle avoidance are names that are quite often heard of lately, and are drastically improving our lives as time goes on. Even though these things are used for plenty of useful reasons, I would like to highlight their use in the field of computer vision, where, for example, we can use object detection for doing tasks like face detection, face recognition, or vehicle counting, amongst many other tasks.

When it comes to obstacle avoidance in fast moving drones, a low processing time is key to success, as an obstacle will not wait for the drone to make a decision in which way it is going to move. In a world in which computers and electronic devices are getting faster and faster every year with every new release, computer vision algorithms can be used and give a result as fast as the processing unit (CPU or GPU) gets the data it needs to calculate that result. That means that whenever a computer vision algorithm receives an input, it will give an output result in a small fraction of a second, being this processing time usually faster (it might be slower in some cases) than the rate with which it receives the input images from the camera, being then, the refresh rate of the conventional cameras a bottleneck for these kinds of algorithms.

Event cameras have the outstanding feature of capturing changes in brightness in an asynchronous way, meaning that they do not have to wait for a clock signal to cast an event, removing the bottleneck that the conventional cameras introduced to obstacle avoidance algorithms, and in general, making everything to be calculated in a faster way, as the algorithms do not have to wait for another input image every time they finish an iteration since events are constantly casted asynchronously, and therefore, in a faster way than the actual calculations by the algorithms are done.

Nowadays, it is unwise to create an algorithm from scratch and testing it on a real drone before going through a simulation framework, which usually represents reality with a really high degree of fidelity, in order to avoid crashes or things being broken in case of an unexpected behavior of our algorithm.

The software set up that is most often used at *Luleå University of Technology* when it comes to drone navigating purposes, is **Linux** and **ROS**, and the simulation environment used for it is **Gazebo**.

Unfortunately, using **Gazebo** with event cameras does not appear to be an easy task, and despite having the possibility of doing it with some third-party plugins, using

Microsoft AirSim turned to be the option to go as it has a built-in event camera simulator.

So now, the real challenge is to use an event camera in simulation, and that uses the **Ubuntu + ROS** framework in an **AirSim** environment.

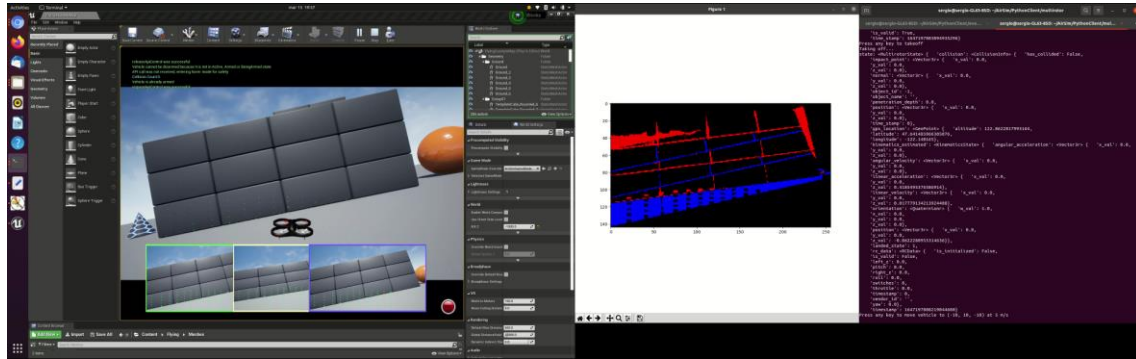


Figure 1: Event image captured by a simulated drone.

1. TIMELINE AND MILESTONES:

In this section, I am going to write about how the project has been developing up to the day of today. There have been several changes in both the main goal and the approach to follow to achieve that main goal, but the most important part of it, being that my work should be something related with event cameras, has always been present.

1.1. FIRST PROPOSAL. EARLY OCTOBER:

After some discussions and meetings with George and Christoforos, we agreed to work on an event camera-based project by Microsoft including machine learning but not **ROS**. We had that idea parked until the end of January, when I started working on it. After many tries of installing a lot of dependencies it turned out not to work for some reason that we still don't know, which was quite disappointing as we liked the idea of the event camera-based project, but at least we had some work done already just by knowing that for some reason that project was not feasible.

1.1.1. WINDOWS APPROACH:

I state this try here not to commit the same errors in the future.

The whole Microsoft project is based in **Linux**, so that is not a good point to start with if we are considering working with **Windows**. Basically, mostly all the dependencies from the “*requirements.txt*” file can’t be installed as there are some versioning issues when using **Windows**.

After tweaking some dependencies to be able to run the software, at some point the simulation does not work because of lacking some environment parameters that are actually given to the program, so we didn’t find a solution for this.

I've succesfully trained the evae with the gates and got a good amount of weights, but when it comes to training the rl_policie, I execute this line in the command promt

```
python train.py --obs_type event_stream --data_len 3 --tc --rep_weights  
C:\Users\username\event-vae-rl\event_vae\weights --ls 128 , and instead of it running  
straight away I get one error asking for more parameters: TypeError: __init__() missing 2 required positional arguments: 'stack' and 'tc' .
```

Figure 2: Error issue posted on github.

1.1.2. LINUX VIRTUAL MACHINE APPROACH:

Taking advantage of a **Manjaro** virtual machine that I already had installed, I tried to install all the dependencies and software needed there, which was a little bit more successful this time as I didn’t have any dependencies errors. Sadly, for using **Unreal Engine** it is needed to have an actual graphics card, which is not the case when you are using a virtual machine as the graphics card is virtual as well, so I was not able to use **Unreal Engine**, at least **4.27**, so it might be possible with **4.24** as it also supports **OpenGL** which is supported as well by virtual machine’s graphic cards. As I said, I was not able to do it but maybe it is possible to forward graphic card capabilities to a virtual machine in a way that this approach would work, but at least I could not find such a way. I also experienced problems with space as **Unreal Engine** takes more than 100Gb to be installed, so I had to resize the virtual machine several times.

1.2. SECOND PROPOSAL. MID FEBRUARY:

After quite a long time trying to replicate the first project in my computer, it was time to move on as the work was stuck and I could not afford more delays.

That first motivation that we had for doing something event-camera-related remained, so after considering several options, I chose to go with an event camera-based collision avoidance project.

We decided first that I would try to simulate the project itself in the simulation tool, and then if we had time, we would try to implement it in real life.

The main goal was achieving to make **ROS** publish events from a simulated event camera within the simulation tool, and then as a further work, those published events could be used in an algorithm.

1.2.1. SIMULATION APPROACH WITH GAZEBO:

Gazebo, as the simulation framework that comes with **ROS**, is very convenient for a lot of types of simulation as well as for the huge variety of sensors that it comes with.



Figure 3: **Gazebo** logotype

There are some plugins for using an event camera on gazebo, but they seemed a little bit tricky to implement, at least for a beginner with that framework like me, and I also got told that **Microsoft AirSim** is the most straight forward approach for simulating an event camera. So, I decided to park **Gazebo** even though it seems to be a nice tool that I will definitely use in the future.

1.2.2. SIMULATION APPROACH WITH AIRSIM:

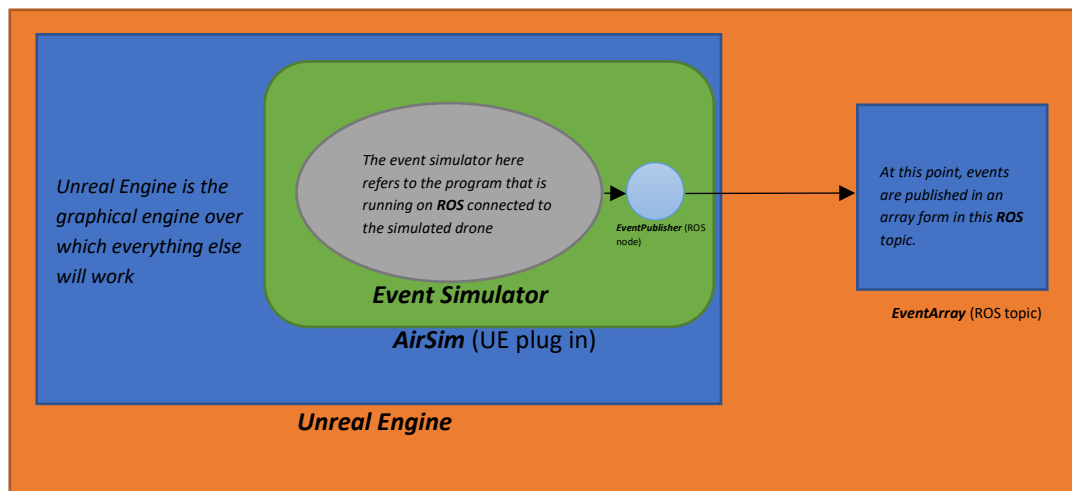
After some work already done with the other approaches, we decided that the first and most important thing that I should achieve with my project would be to understand and being able to use the event simulator provided by Microsoft, as well as publishing the events over **ROS**.

Once we achieve that, as a further work, then we could try to implement a navigation algorithm with that used the event camera and test if works properly.

In the next section I am going to go through this approach as is the one that I am finally using, serving as a tutorial on how to replicate it.

This work is based on the following framework: **Ubuntu 20.04 + ROS Noetic**.

With this approach I have successfully achieved to publish the events captured by a simulated event camera on a drone in **AirSim**, also showed that the events change throwing a ball towards the drone, and making the drone move in an environment with objects.



Block diagram of the solution

2. THE FINAL SOFTWARE SET UP:

Now that the task has been explained, it is time to talk about what is needed when it comes to software to replicate this work.

The final software setup that is being used currently is **Ubuntu 20.04** together with **ROS Noetic**, the graphics engine used is **Unreal Engine 4.25**, and on top of that we are using the **AirSim** plugin which is the actual flying simulator.

3. TUTORIAL ON HOW TO REPLICATE THE WORK DONE:

3.1. INSTALLING THE MAIN SOFTWARE:

In case that you don't have it already, you will need to install **Ubuntu 20.04 + ROS Noetic**. In this section I am going to go through how to install both.

3.1.1. INSTALLING UBUNTU 20.04:

With the help of a computer and a flash drive (with at least 8gb available), we can install **Ubuntu 20.04**. There are several approaches for this but I am going to go through the one that I used coming from a **Windows 10** installation.

You can download the latest version of **Ubuntu** [here](#), I cannot assure that this approach will work with newer versions when they start coming out since this project has too many dependencies involved that may cause incompatibility issues, so in case that it does not work I would recommend looking for the 20.04 image.

Once **Ubuntu** is downloaded, the next thing you are going to download is **Rufus**, which is my to-go application for creating bootable flash drives. You can download it [here](#).

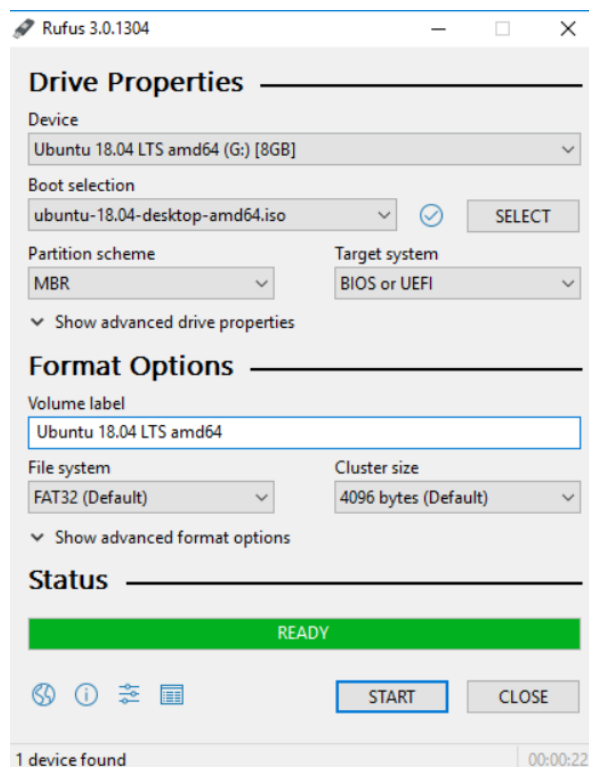


Figure 4: **Rufus** main menu.

In figure 4, you can see the main menu of **Rufus**, on the device tab you select your flash drive, and on the boot selection tab you select the operating system image that you have downloaded.

3.1.2. INSTALLING ROS NOETIC:

To install **ROS Noetic** you only have to follow this tutorial, and do the specific steps for Noetic: <http://wiki.ros.org/Installation/Ubuntu>.

In case you are not familiar yet with how **ROS** works following the beginner tutorials is more than recommendable: <http://wiki.ros.org/ROS/Tutorials>.

3.2. UNREAL ENGINE 4.25:

The graphics engine that we will be using for running **AirSim** over it is **Unreal Engine 4.25**, and in order to install it you will need to have around 100Gb of free disk space just for that purpose, so it is recommended to have around from 150 to 200Gb of free disk space for replicating this project as it requires a lot of dependencies along the way.

To install **Unreal Engine 4.25**, you have to clone the 4.25 branch from **GitHub**, but for being able to do so first you have to follow the steps in the “1 – Required Setup” section in this [tutorial](#) in order to have access to that repository, since it is a private one. Then once those steps are done and you have access to that repository, to clone that specific branch is as easy as executing this command on a terminal:

```
git clone https://github.com/EpicGames/UnrealEngine.git --branch=4.25
```

And once that is cloned, run the following commands to finish the installation of it:

```
cd UnrealEngine  
./Setup.sh  
./GenerateProjectFiles.sh  
make
```

Once that is done, **Unreal Engine** is ready to go, but it will take a while to install everything, depending on your computer's specifications.

3.3. AIRSIM:

AirSim is the simulator that we are going to be using for this project, and it runs over **Unreal Engine**, that is why we had to install it in the previous step.

Installing **AirSim** is pretty easy, but I am going to sum up the steps here, so open a new terminal and paste the following commands:

```
git clone https://github.com/Microsoft/AirSim.git
cd AirSim
./setup.sh
./build.sh
```

Now in case you want to create your own environment for your vehicles in **Unreal Engine**, you will have to do so on **Windows** as it is not possible to create them on **Linux**, however there is an already created environment which comes with **AirSim** that you can use for your simulations and is good enough for this project, so you will not need to create a new environment to replicate this work.

3.4. EVENT CAMERA ROS WRAPPER:

AirSim comes with a built-in **ROS** wrapper that we can use to communicate with the simulated drone making it receive and publish messages. It comes with several topics and message types preconfigured however it does not have any native event camera topic for publishing and receiving events.

Also, **AirSim** comes with an event simulator that works pretty nice, so what I have done is to adapt that simulator to the **ROS** wrapper so it publishes the array of events that the event camera is detecting with a timestamp, a position x and y which refers to pixel coordinates, and a polarity, which can be either true or false.

To get the adapted event simulator, you can clone this [github folder](#) and place it in the folder "*AirSim/ros/src*", this will replace your current *airsim_ros_pkgs* folder but you do not have to worry about that since I did not delete anything, I just added my work in there. This new folder includes everything you need to make the Event publisher work, including the messages, the scripts and the launch files, so what you would have to do is to build (just execute the command "*catkin_make*" in the "*AirSim/ros*" folder) that workspace before running it.

3.5. FINAL STEPS ONCE EVERYTHING IS INSTALLED:

Once we have all the dependencies installed, as well as the previous repository built, now we will be able to finally replicate the project and do further work on it.

3.5.1. OPENING UNREAL EDITOR:

Open a terminal in the following directory:

“~/UnrealEngine/Engine/Binaries/Linux” and there execute the following command:

```
./UE4Editor
```

That will open the main menu of **Unreal Engine**, now you have to open the Blocks environment, in case it is the first time you open it you will have to follow these steps:

- Go to **Unreal Engine** installation folder and start Unreal by running

```
./Engine/Binaries/Linux/UE4Editor
```
- When **Unreal Engine** prompts for opening or creating project, select Browse and choose “AirSim/Unreal/Environments/Blocks” (or your [custom Unreal](#) project).
- Alternatively, the project file can be passed as a commandline argument. For Blocks:

```
./Engine/Binaries/Linux/UE4Editor  
<AirSim_path>/Unreal/Environments/Blocks/Blocks.uproject
```

- If you get prompts to convert project, look for More Options or Convert-In-Place option. If you get prompted to build, choose Yes. If you get prompted to disable **AirSim** plugin, choose No.

3.5.2. SETTING UP ROS:

First of all, execute in a terminal:

```
roscore
```

Then, in a new terminal, go to “/AirSim/ros”, and execute:

```
source devel/setup.bash
```

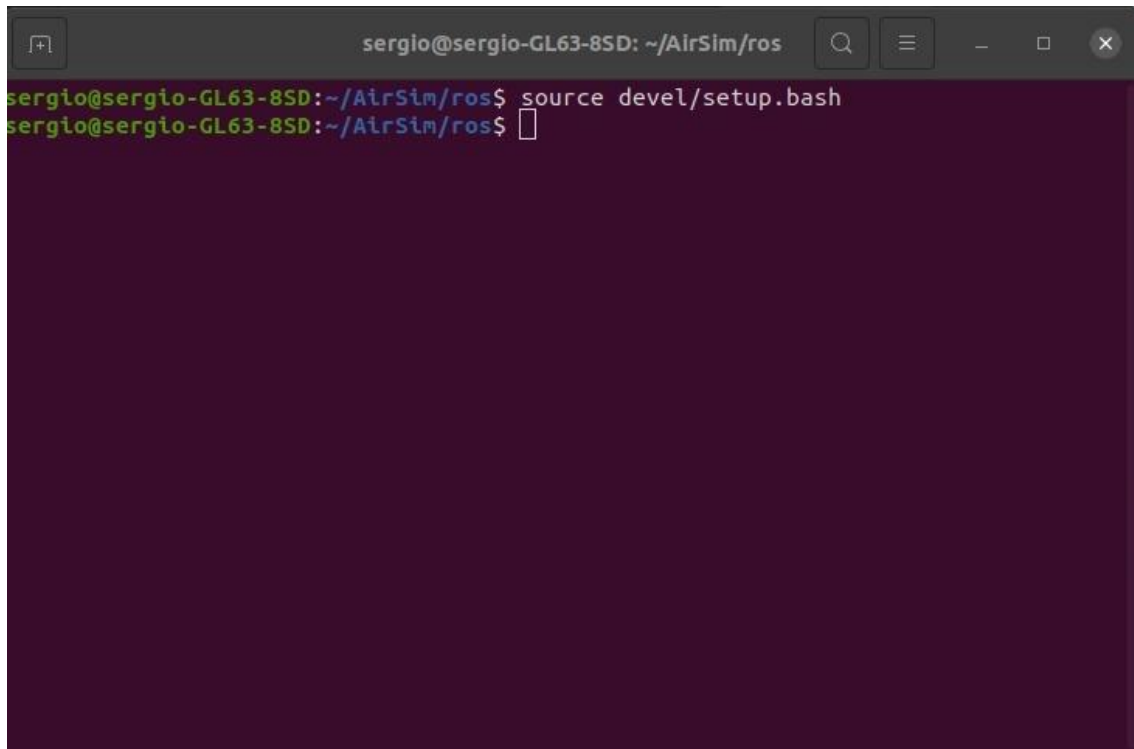
A terminal window with a dark background and light-colored text. The window title bar shows 'sergio@sergio-GL63-8SD: ~/AirSim/ros'. The terminal content shows two lines of text: 'sergio@sergio-GL63-8SD:~/AirSim/ros\$ source devel/setup.bash' and 'sergio@sergio-GL63-8SD:~/AirSim/ros\$' followed by a cursor. The terminal window has standard Linux window controls (minimize, maximize, close) and a search icon.

Figure 5: sourcing the catkin workspace.

3.5.3. USING THE EVENT SIMULATOR:

First of all, we click play in the **Unreal Engine** window (in my case my drone appears with a depth camera and some more other features, if it is the first time you use **AirSim**, most likely you will not have those things, but they are not important for this approach, you will also probably have to choose between car and drone simulation, just choose drone simulation).

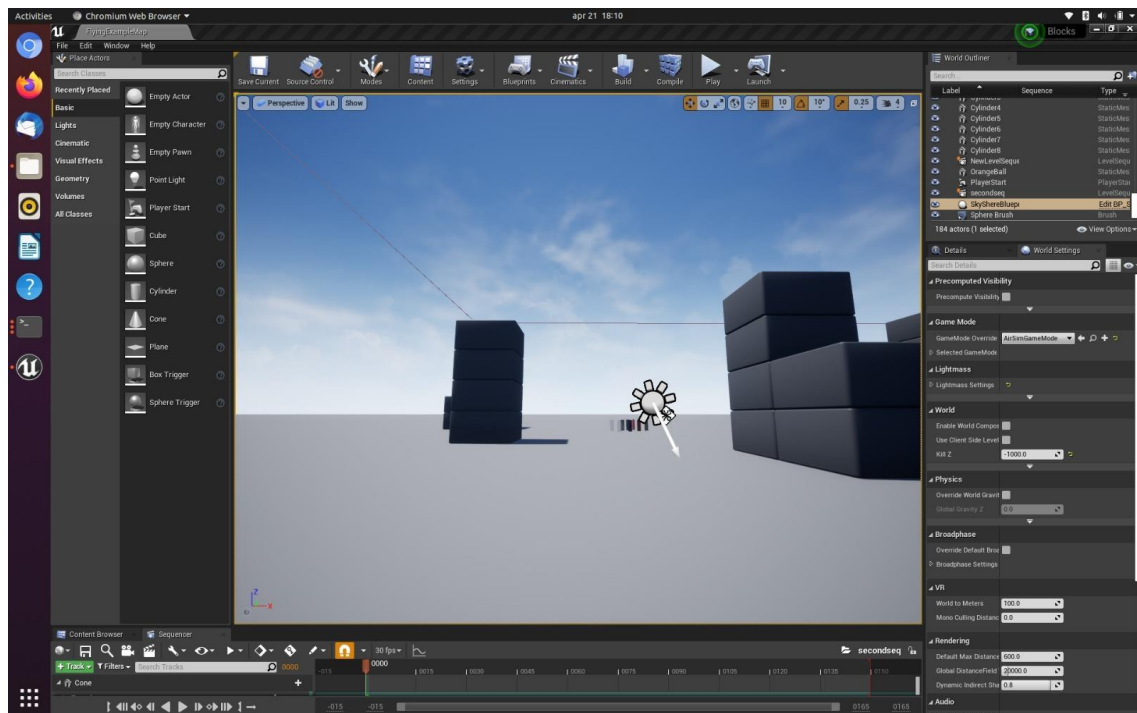


Figure 6: Play button appears in the docker right above the simulation, being the second icon starting from the right.

Once that is running, on the same terminal we used to source the workspace on the previous step, we run:

```
roslaunch airsimsim_ros_pkgs EventROS.py
```

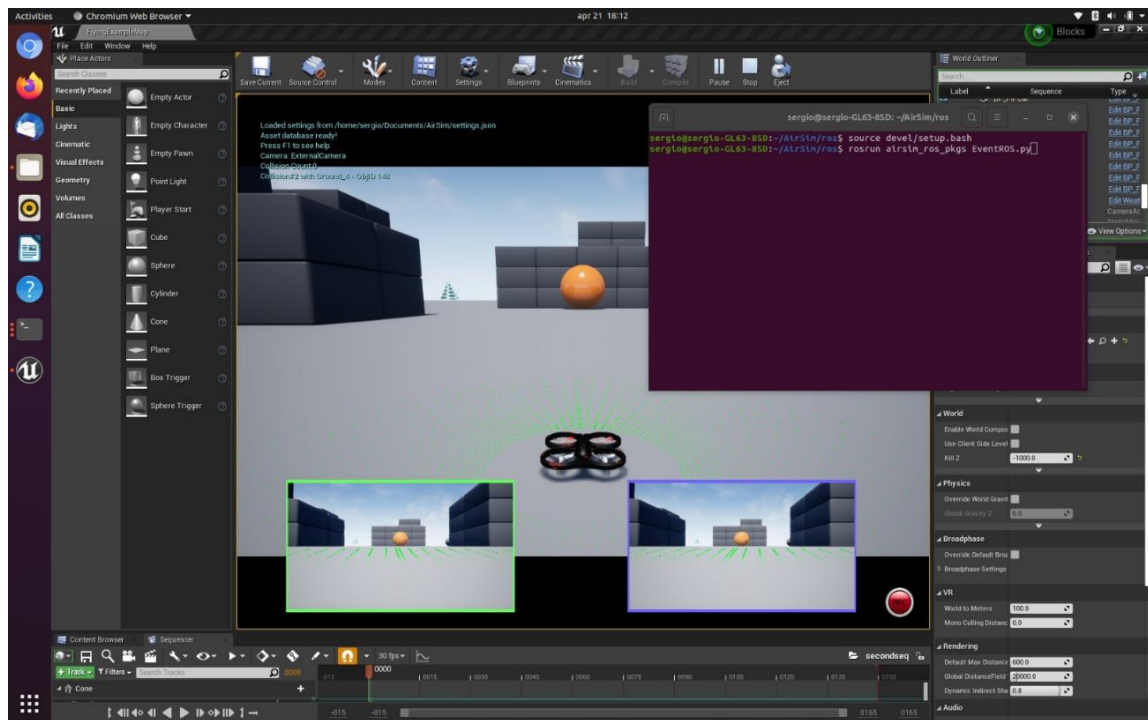


Figure 7: `rosrun airsims_ros_pkgs EventROS.py` command written.

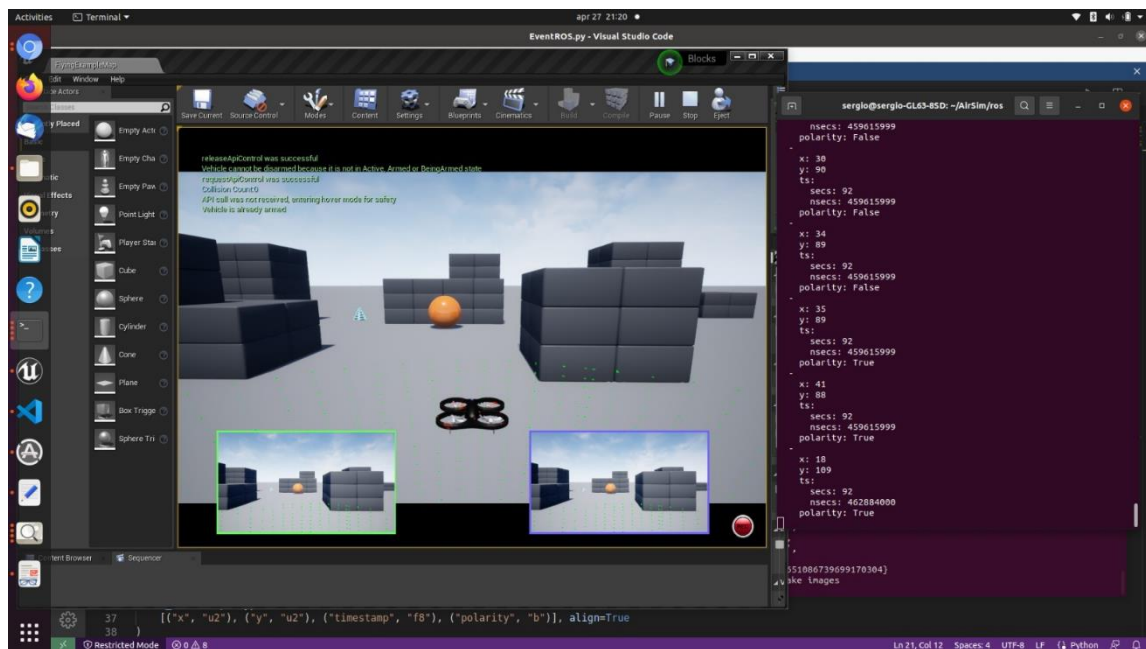
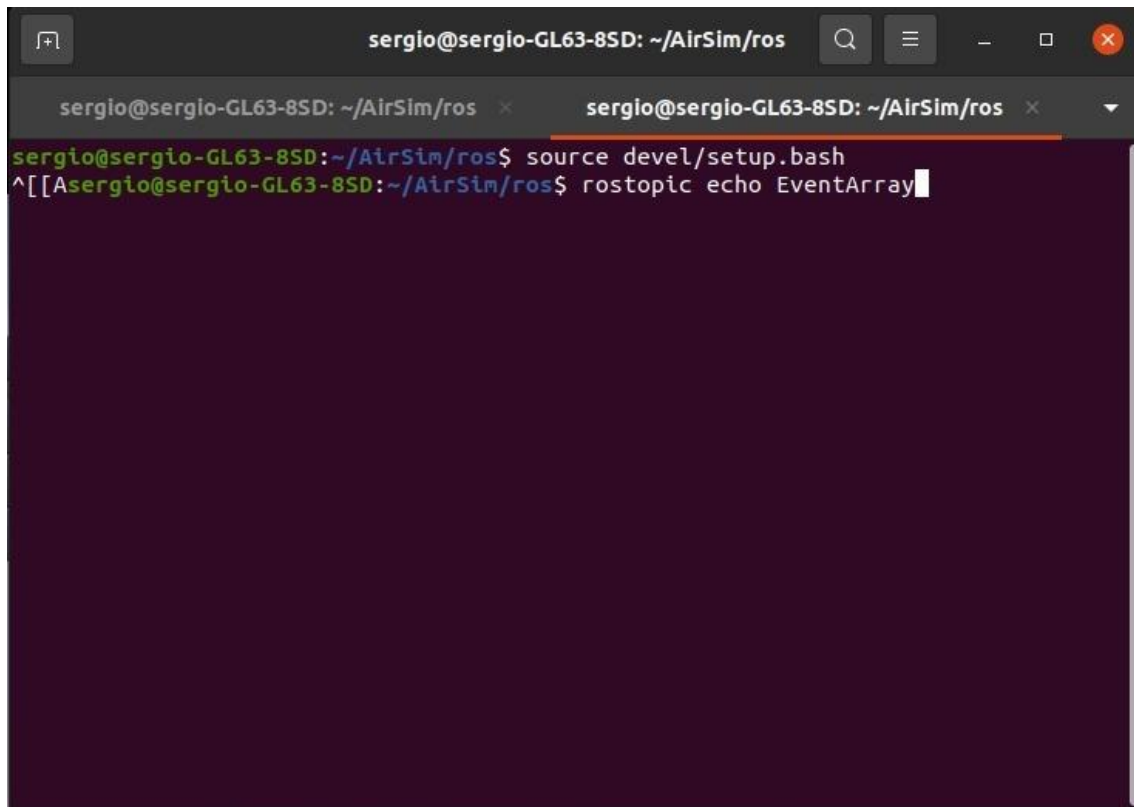


Figure 8: Event simulator running and displaying events on the terminal.

3.5.4. SEEING THE EVENTS PUBLISHED WITH ROS:

Once we have the Event simulator and “roscore” running, we have to open a new terminal in the same location as the previous one (“/AirSim/ros”), we source it so **ROS** knows where to find the previously built messages, and then we execute:

```
rostopic echo EventArray
```

A screenshot of a terminal window with a dark background. The window title bar shows 'sergio@sergio-GL63-8SD: ~/AirSim/ros'. There are two tabs open, both with the same title. The terminal shows the following commands and their outputs: 'sergio@sergio-GL63-8SD:~/AirSim/ros\$ source devel/setup.bash' followed by a prompt '^[[A', and then 'sergio@sergio-GL63-8SD:~/AirSim/ros\$ rostopic echo EventArray' followed by a cursor. The terminal is mostly empty below the second command.

```
sergio@sergio-GL63-8SD: ~/AirSim/ros
sergio@sergio-GL63-8SD:~/AirSim/ros$ source devel/setup.bash
^[[Asergio@sergio-GL63-8SD:~/AirSim/ros$ rostopic echo EventArray
```

Figure 9: Commands written for echoing the topic.

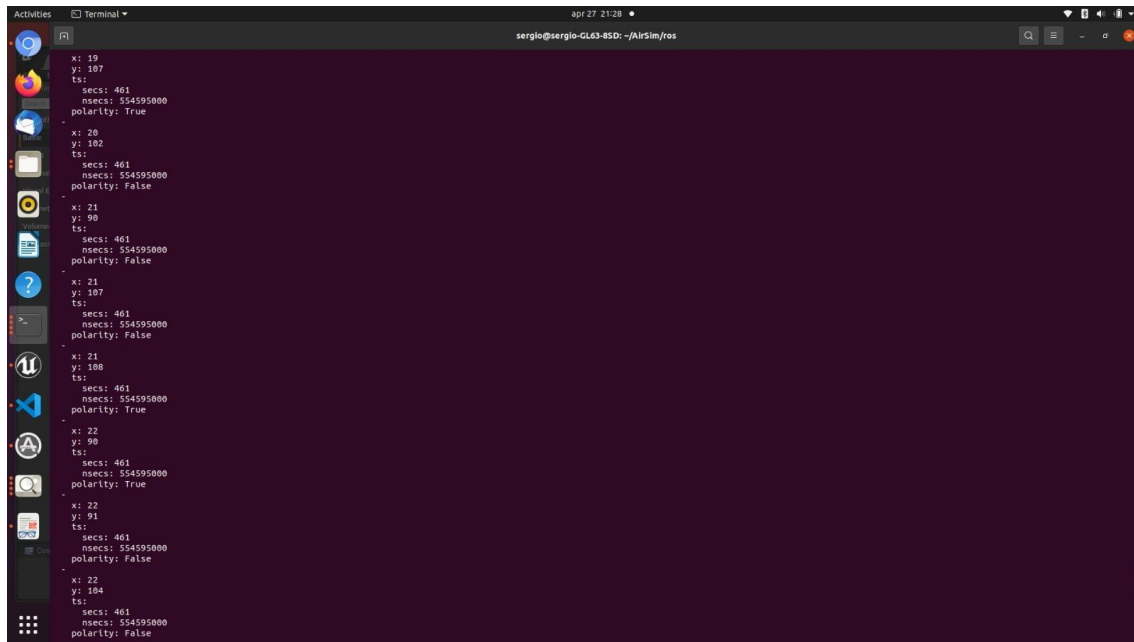


Figure 10: Events published in the topic EventArray.

Up to this point, that would be all to make the Event simulator work, the events are being published and it is possible to see them with “*rostopic echo*”, on the following section I will just add how to make the drone move in a simple predetermined way, and how to create animations on **Unreal Engine** so you can actually move objects that can be detected by the simulated event camera, and that you will see in the published events, showing that the event publisher is actually working.

3.6. DYNAMIC MOVEMENT OF THE DRONE AND OBJECTS IN UNREAL ENGINE:

I am not going to write a section on how to move the drone in a predetermined path because there is no point in doing so as it is too short, in spite of a section I will just put it here, and then I will write a section on how to create animated cinematics in **Unreal Engine**.

So, in case we want to see the drone moving, we will just use a flight example provided by **AirSim** which is really easy to use. On a new terminal we go to the following location: “~/AirSim/PythonClient/multirotor”, and there we execute the following command:

```
python3 hello_drone.py
```

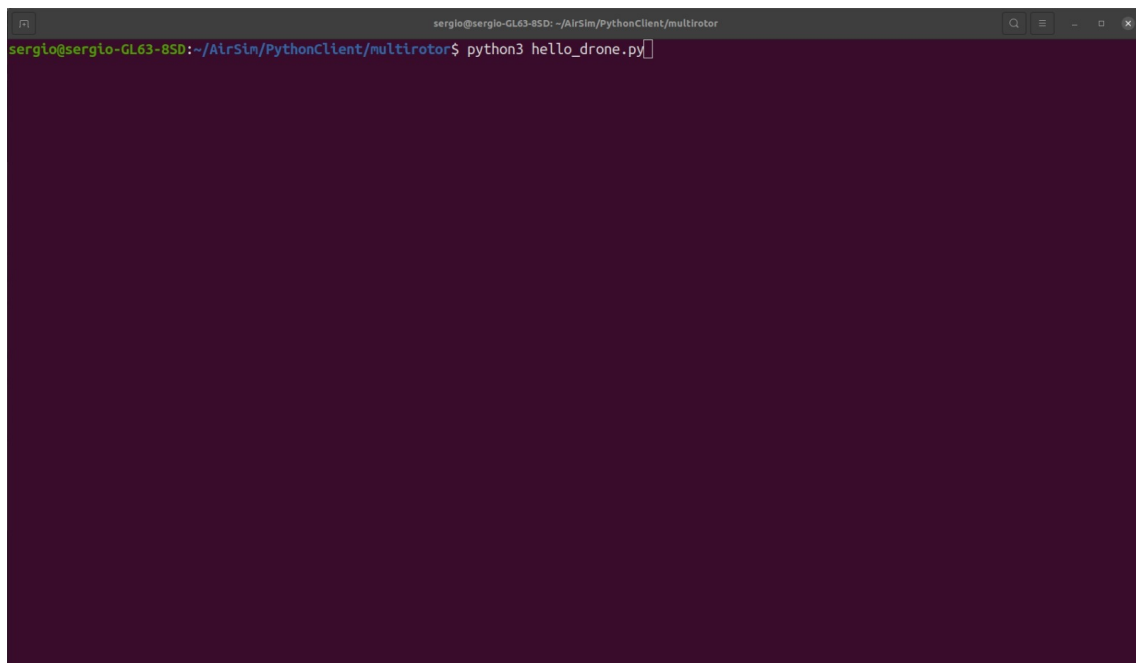


Figure 11: Flight example by **AirSim**.

3.6.1. CREATING AN ANIMATED SEQUENCE:

First of all, we will create a new cinematic, for that, follow these steps:

- Click on cinematics, then in the menu that will pop up click on add level sequence:

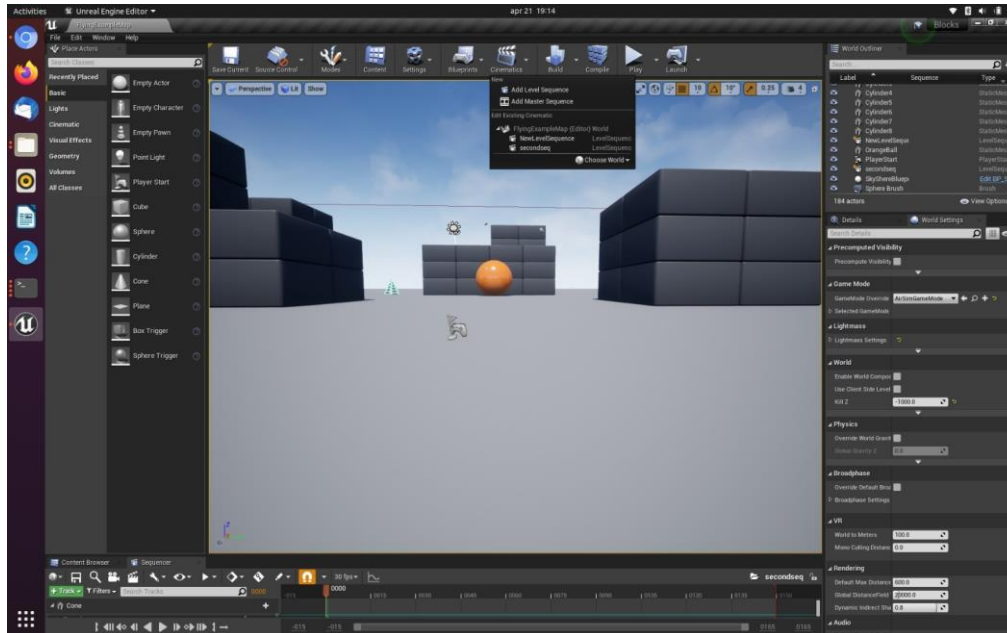


Figure 12: Cinematics menu.

- Then a menu in which you can give a name to your cinematic and save it in a specific location will appear, just give it a name and save it wherever you want.

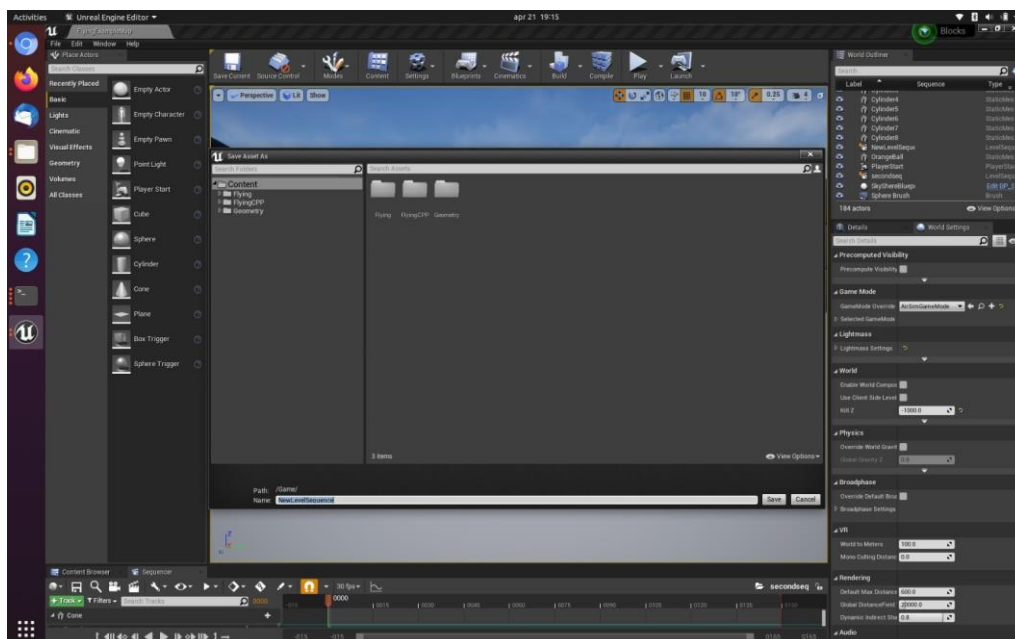


Figure 13: Save Asset As menu.

- The sequencer will prompt, and in case it does not prompt go to “cinematics->”name of the sequence””.

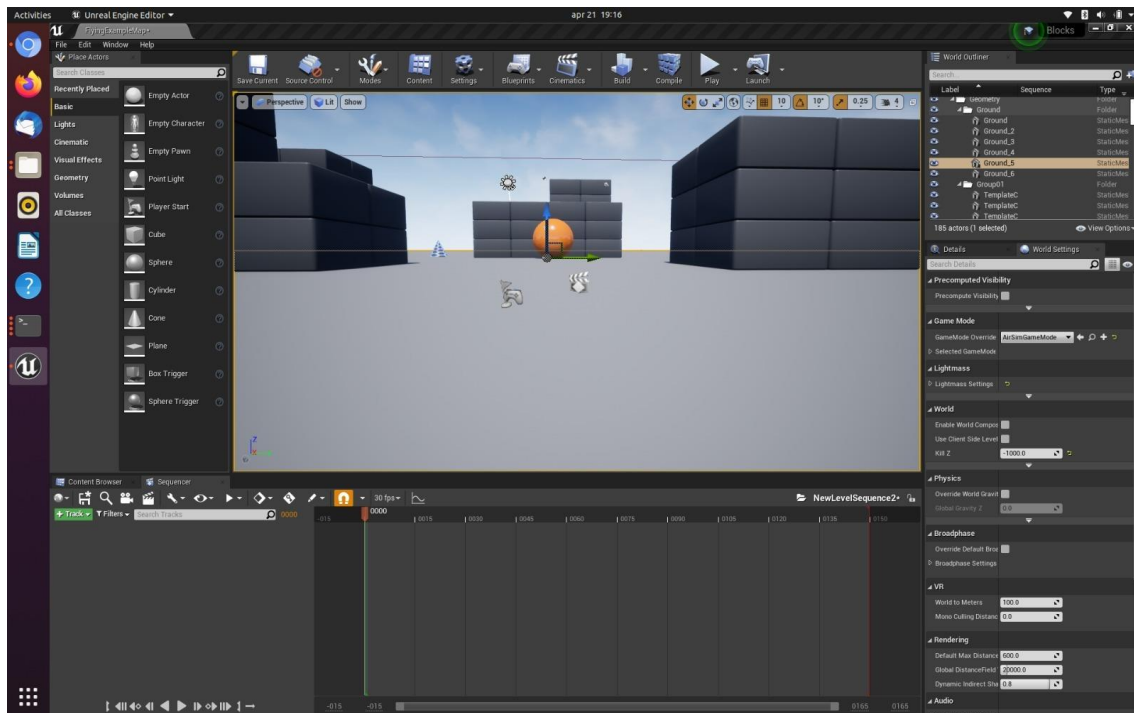


Figure 14: Sequencer.

- Then click on “+Track -> Actor to sequencer -> ”whichever actor you want to animate””

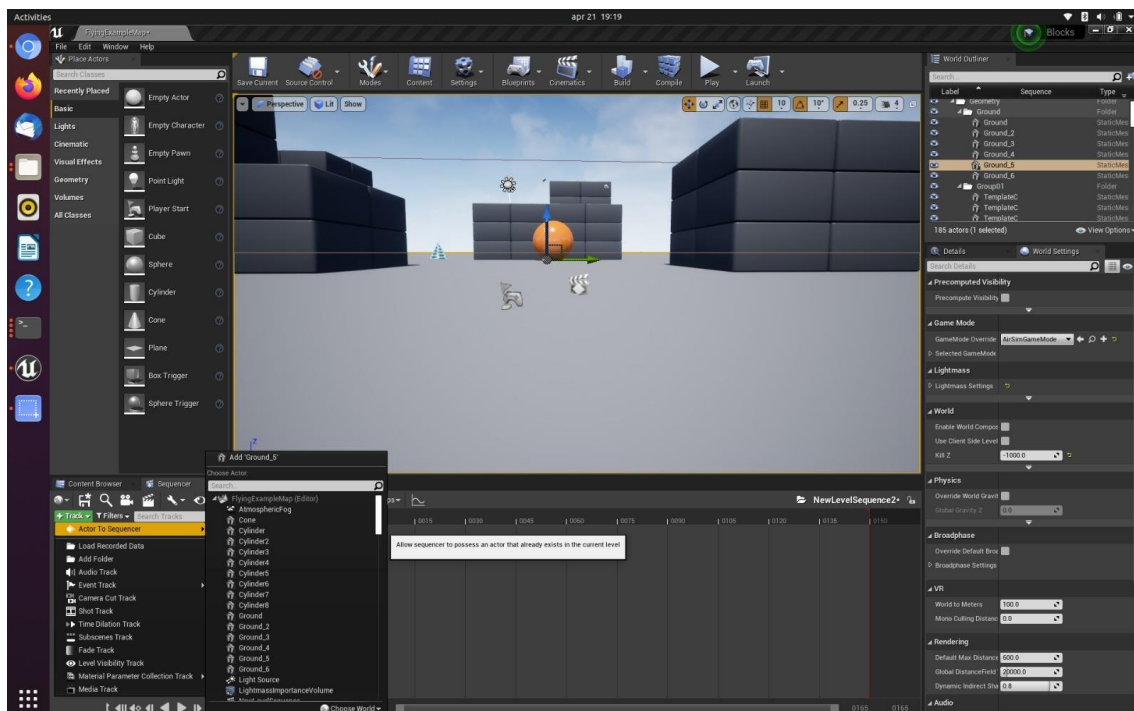


Figure 15: Selecting an actor for the sequence.

- Now you will only have to move the actor and save the keyframes in the timeline pressing the space bar. If it is easier, just follow [this tutorial until the minute 5:09](#).

3.6.2. ADDING THE TRIGGERING ACTION:

Once the sequence is created, click on “blueprints -> Open Level Blueprints”

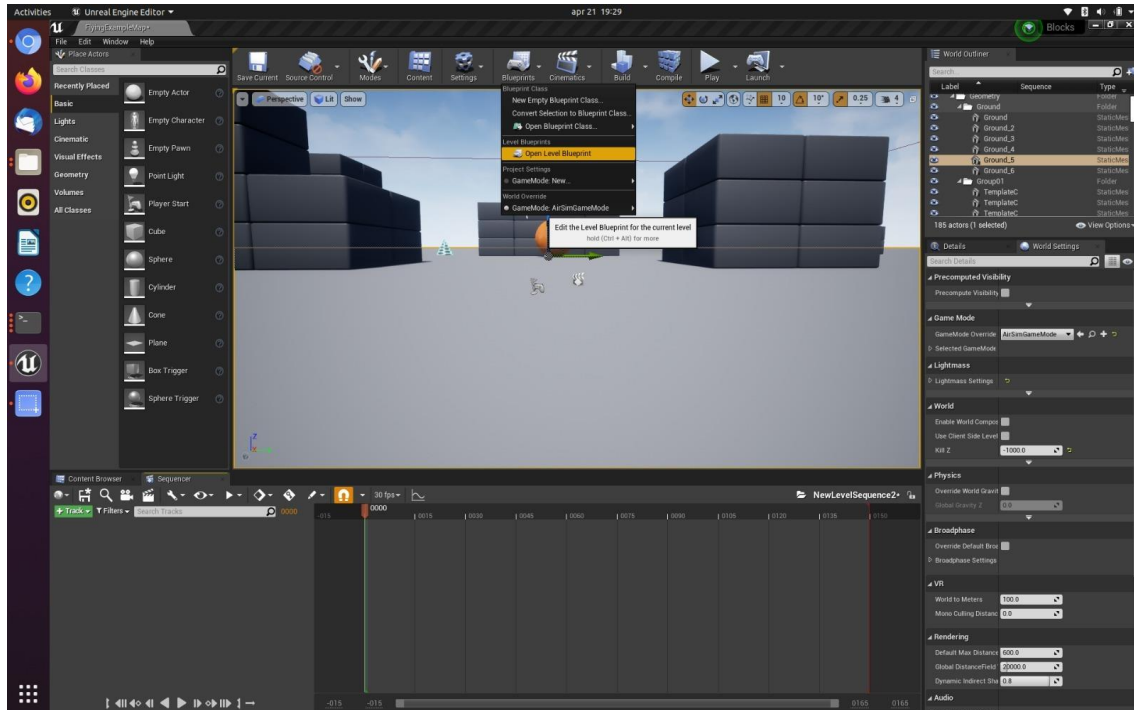


Figure 15: Opening level blueprint.

Now we will add the following blocks:

- keyboard trigger (spacebar for example).
- Right after keystroke -> "Create Level Sequence Player" [Here we have to choose which sequence to play in the parameter "level sequence"].
- Right after, connected to "Exec", search for sequence play and add "Play (SequencePlayer)", two new blocks will appear, connect "Out Actor" to the unconnected target that appears with the sequence player.

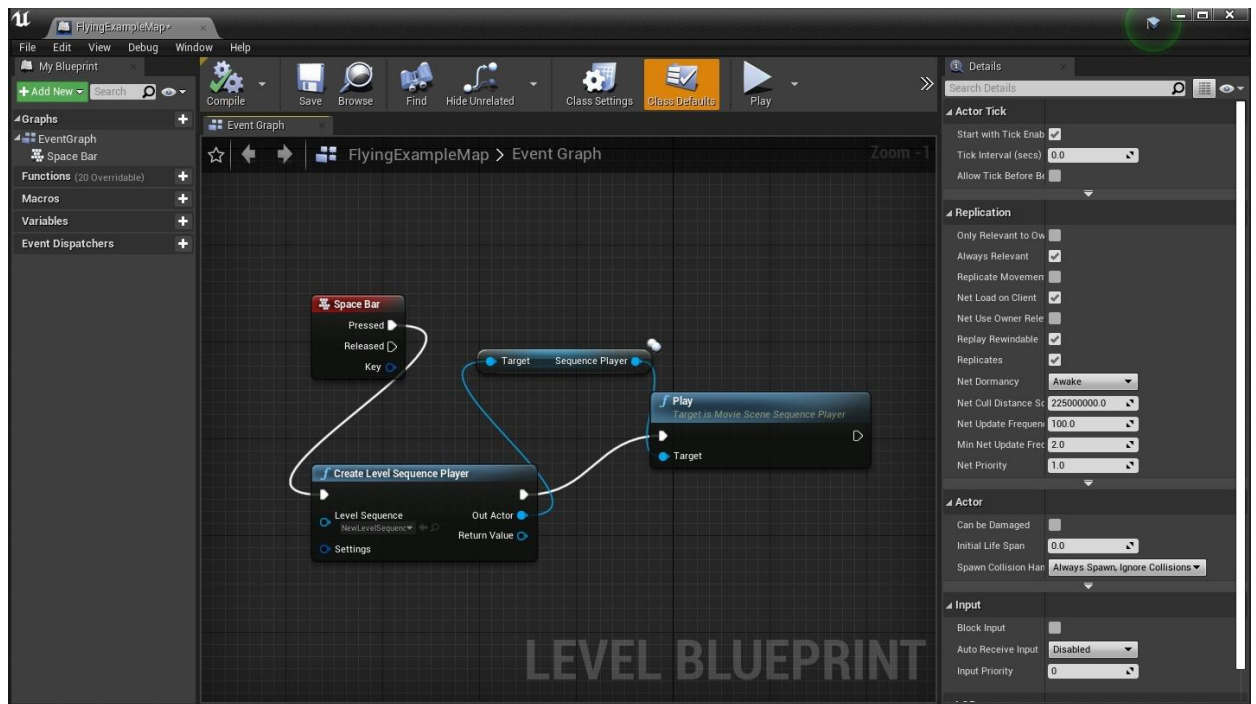


Figure 16: All the previous blocks connected.

Now after that is done, compile and save, and every time that the keystroke is pressed while the simulation is running the cinematic will be played (if it implies that an object moves towards the event camera it will be detected, and you will be able to see so on the published events over **ROS**).